

Telelogic **Rhapsody**

User Guide



IBM®

Rhapsody®

User Guide



Before using the information in this manual, be sure to read the “Notices” section of the Help or the PDF available from **Help > List of Books**.

This edition applies to Telelogic Rhapsody 7.4 and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 1997, 2008.

US Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Introduction to Rhapsody	1
Rhapsody Features	2
Rhapsody Tools	3
The Rhapsody Browser	4
The Favorites Browser	5
Graphic Editors	6
Code Generator	8
Animator	8
Utilities	8
Third-party Interfaces	9
Rhapsody Windows	10
Browser	12
Diagram Drawing Area	12
Diagram Navigator	16
Output Window	17
Active Code View	26
Welcome Screen	27
Rhapsody Project Tools	28
Browser Filter	28
Standard Project Tools	29
Generating and Running Code Tools	30
Managing and Arranging Windows	31
Favorites Toolbar	32
Creating New Diagrams	33
VBA Toolbar	35
Animation Toolbar	35
Drawing Toolbar	37
Common Drawing Tools	37
Zoom Toolbar	38
Format Toolbar	39
Layout Toolbar	40
Free Shapes Toolbar	41
Using the Features Dialog Box	43
Invoking the Dialog Box	43

Table of Contents

Applying Changes	44
Canceling Changes	44
Pinning the Dialog Box	44
Hiding the Buttons on the Features Dialog Box	45
Docking the Features Dialog Box	45
Opening Multiple Instances of the Dialog Box.	45
Properties Tab.	46
Displaying Feature Tabs as Stand-alone Dialog Boxes	48
Hiding Tabs.	49
Hyperlinks	52
Creating Hyperlinks	52
Following a Hyperlink	55
Editing a Hyperlink	55
Deleting a Hyperlink	58
New Rhapsody Diagram Dialog Box	59
New Rhapsody Project Dialog Box	60
Import Dialog Box	61
Importing Source Code Dialog Box	63
UML Design Essentials	65
Unified Modeling Language	65
UML Diagrams	65
UML Views	66
Diagrams in Rhapsody	68
Partially Constructive Diagrams	68
Fully Constructive Diagrams	68
Specifying a Model with Rhapsody	69
Development Methodology.	70
Analysis	70
Design	72
Implementation	73
Testing	73
Classes and Types	75
Creating a Class	75
Defining the Features of a Class	76
Defining the Characteristics of a Class	76
Defining the Attributes of a Class	79
Defining Class Operations	84
Defining Class Ports	98

Defining Relations	99
Defining Class Tags	103
Defining Class Properties	103
Adding a Class Derivation	104
Defining Class Behavior	106
Generating, Editing, and Roundtripping Class Code	106
Generating Class Code	106
Editing Class Code	107
Roundtripping Class Code	107
Opening the Main Diagram for a Class	108
Setting Display Options	109
General Tab Display Options	110
Displaying Attributes and Operations	113
Removing or Deleting a Class	115
Ports	115
Partial Specification of Ports	116
Considerations	116
Creating a Port	118
Specifying the Features of a Port	119
Viewing Ports in the Browser	125
Connecting Ports	125
Using Rapid Ports	125
Selecting Which Ports to Display in the Diagram	128
Deleting a Port	129
Programming with the Port APIs - RiC++	129
Port Code Generation - RiC	133
Port Code Generation - Java	133
Composite Types	135
Creating Enumerated Types	137
Creating Language Types	138
Creating Structures	139
Creating Typedefs	140
Creating Unions	141
Properties	142
Language-Independent Types	144
Changing the Type Mapping	145
Changing the Order of Types in the Generated Code	147
Using Fixed-point Variables	148
Defining Fixed-point Variables	148
Operations Permitted for Fixed-point Variables	149

Table of Contents

Restrictions on Use of Fixed-point Variables	149
Fixed-point Conversion Macros	150
Java Enums	151
Adding a Java Enum to a Model	151
Defining Constants for a Java Enum	151
Adding Java Enums to an OMD	152
Code Generation	152
Creating Java Enums with the Rhapsody API	152
Template Classes, Generic Classes	153
Creating a Template Class	153
Using Template Classes as Generalizations	158
Creating an Operations Template	158
Creating a Functions Template	159
Instantiating a Template Class	160
Code Generation and Templates	161
Template Limitations	161
Eclipse Platform Integration	163
Platform Integration Prerequisites	163
Confirming Your Rhapsody Platform Integration within Eclipse	164
Guided Tour of the Rhapsody Platform Integration within Eclipse	165
Rhapsody Modeling Perspective	166
Rhapsody Debug Perspective	167
Rhapsody Perspectives in Eclipse	168
Working with Eclipse Projects	170
Creating a New Rhapsody Project within Eclipse	170
Opening a Rhapsody Project in Eclipse	172
Adding New Elements	176
Filtering Out File Types	177
Exporting Eclipse Source Code to a Rhapsody Project	179
Rhapsody Unit Import	182
Source Code Import (Reverse Engineering)	185
Search and Replace in Models	187
Accessing the Rhapsody Search Facility in Eclipse	188
Generating and Editing Code	191
Checking the Model	191
Generating Code	192
Dynamic Model-Code Associativity	199
Editing Code	200
Building, Debugging, and Animating	206

Building Your Eclipse Project	206
Debugging Your Eclipse Project	208
Rhapsody Animation in Eclipse	209
Eclipse Configuration Management	212
Parallel Development	212
Configuration Management and Rhapsody Unit View	213
Sharing a Rhapsody Model	215
Performing Team (CM) Operations	218
Rhapsody DiffMerge Facility in Eclipse	220
Generating Rhapsody Reports	221
Working with Projects	223
Project Elements	223
Creating and Managing Projects	224
Creating a Project	224
Profiles	225
Opening a Rhapsody Project	227
Search and Replace Facility	227
Locating and Listing Specific Items in a Model	232
Editing a Project	233
Using IDF for a Rhapsody in C Project	236
Saving a Project	237
Renaming a Project	238
Closing a Project	238
Creating and Loading Backup Projects	239
Archiving a Project	240
Creating Table and Matrix Views	241
Working with Multiple Projects	262
Opening Multiple Projects	262
Setting the Active Project	263
Copying and Referencing Elements among Projects	263
Moving Elements among Projects	267
Closing All Open Projects	268
Managing Project Lists	269
Project Limitations	271
Naming Conventions and Guidelines	272
Guidelines for Naming Model Elements	272
Standard Prefixes	273
Using Project Units	274
Unit Characteristics and Guidelines	275
Separating a Project into Units	276

Table of Contents

Modifying Units	277
Saving Individual Units	277
Loading/Unloading Units	277
Saving Packages in Separate Directories	279
Using Environment Variables with Reference Units	281
Using Workspaces	282
Creating a Custom Rhapsody Workspace	282
Adding Units to a Workspace	282
Unloaded Units	283
Opening a Project with Workspace Information	283
Controlling Workspace Window Preferences	283
Project Files and Directories	284
Parallel Project Development	286
Unit Types	286
DiffMerge Tool Functions	287
Project Migration and Multi-Language Projects	288
Opening Models from a Different Language Version	288
Multi-Language Projects	289
Using the Rhapsody Workflow Integration with Eclipse	292
Converting a Rhapsody Configuration to Eclipse	293
Importing Eclipse Projects into Rhapsody	293
Creating a New Eclipse Configuration	294
Troubleshooting Your Eclipse Installation with Rhapsody	295
Switching Between Eclipse and Wind River Workbench	296
Rhapsody Tags for the Eclipse Configuration	296
Rhapsody Features Settings for Eclipse	296
Eclipse Workbench Properties	297
Editing Rhapsody Code using Eclipse	298
Locating Implementation Code in Eclipse	298
Opening an Existing Eclipse Configuration	298
Disassociating an Eclipse Project from Rhapsody	299
Workflow Integration with Eclipse Limitations	299
Domain-specific Projects and the NetCentric Profile	300
Defining a Service Package for the Service Provider	302
Creating a WSDL Specification	303
Exporting a WSDL Specification File	303
Using the Schedulability, Performance, and Time (SPT) Profile	304
Overview	304
How to Use the SPT Model Provided with Rhapsody	304
Co-Debugging with Tornado	307
Preparing the Tornado IDE	307

IDE Operation in Rhapsody	307
Co-Debugging with the Tornado Debugger	308
IDE Properties	309
Creating Rhapsody SDL Blocks	310
Using Model Elements	313
Using the Rhapsody Browser	313
Opening the Rhapsody Browser	314
Setting the Display Mode	314
Basic Browser Icons	316
Filtering the Browser	320
Opening the Browse From Here Browser	322
Deleting Items from a Browser	324
Using the Favorites Browser	325
Creating your Favorites List	326
Removing Items from your Favorites List	326
Showing and Hiding the Favorites Browser	327
Favorites Browser Limitations	327
Adding Elements	328
Settings in the Browser	328
Adding a Rhapsody Profile Manually	329
Editing Component Elements	330
Configurations	330
Configuration Files	330
Locating an Element on a Diagram	331
Example of Locate On Diagram from Code View	333
Locate On Diagram Rules	334
Limitations	334
Using Package Elements	335
Functions	336
Creating a Global Function	336
Objects	336
Variables	337
Dependencies	338
Constraints	338
Classes	338
Types	338
Receptions	338
Events	338
Actors	339
Use Cases	339

Table of Contents

Nodes	339
Files	339
Adding Diagram Elements	340
Element Identification and Paths	341
Labeling Elements	342
Adding a Label to an Element	342
Removing a Label from an Element	344
Label Mode	344
Moving Elements	345
Copying Elements	345
Renaming Elements	346
Displaying Stereotypes of Model Elements	347
Creating Graphical Elements	348
Associating an Image File with a Model Element	349
Displaying Associated Images	349
Restoring Image Size, Proportions	350
Modifying, Replacing, and Removing Associated Image Files	350
Supported Image Formats	350
Compatibility with Previous Image Association Mechanism	351
Controlled Files and Image Association	351
Deleting Elements	352
Reordering Elements in the Browser	353
Smart Drag-and-Drop	354
Searching in the Model	358
Finding Element References	358
Advanced Search and Replace Features	359
Using the Auto Replace Feature	360
Searching for Elements	361
Searching in Field Types	362
Previewing in the Search and Replace Facility	364
Controlled Files	365
Creating a Controlled File	366
Browsing to a Controlled File	368
Controlled File Features	371
Troubleshooting Controlled Files	375
Controlled Files Limitations	376
Printing Rhapsody Diagrams	379
Selecting Which Diagrams to Print	379

Diagram Print Settings	381
Using Page Breaks	382
Exporting Rhapsody Diagrams	383
Navigating Between DOORS and Rhapsody	384
Adding Annotations to Diagrams	384
Creating Annotations	385
Editing Annotation Text	387
Defining the Features of an Annotation	387
Converting Notes to Comments	389
Anchoring Annotations	389
Display Options for Annotations	392
Deleting an Annotation	394
Using Annotations with Other Tools	395
Annotation Limitations	395
Using Profiles	396
Projects without Profiles	397
Types of Profiles	399
Converting Packages and Profiles	399
Profile Properties	399
Using a Profile to Enable Access to Your Custom Help File	400
Defining Stereotypes	406
Associating Stereotypes with an Element	407
Associating Stereotypes with a New Term Element	407
Order of Stereotypes in List	408
Associating a Stereotype with a Bitmap	408
Deleting Stereotypes	409
Stereotype Inheritance	410
Special Stereotypes	410
Using Tags to Add Element Information	411
Defining a Stereotype Tag	411
Defining a Global Tag	413
Defining a Tag for an Individual Element	414
Adding a Value to a Tag	415
Deleting a Tag	416
The Internal Editor	417
Window Properties	417
The Color/Font Tab	418
The Language/Tabs Tab	420
The Keyboard Tab	421
The Misc Tab	425

Table of Contents

Mouse Actions	427
Undo and Redo	427
Searching	428
Bookmarks	428
Printing	429
Graphic Editors	431
Creating New Diagrams	431
Creating Statecharts and Activity Diagrams	432
Creating all Other Diagram Types	433
Opening Existing Diagrams	434
Deleting Diagrams	434
Automatically Populating Diagrams	435
Property Settings for the Diagram Editor	439
Setting Diagram Fill Color	440
Creating Elements	440
Repetitive Drawing Mode	441
Drawing Boxes	441
Drawing Arrows	442
Naming Boxes and Arrows	444
Drawing Freestyle Shapes	444
Placing Elements Using the Grid	450
Using Autoscroll	451
Selecting Elements	452
Selecting Elements Using the Mouse	452
Selecting Elements Using the Edit Menu	452
Selection Handles	453
Selecting Multiple Elements	454
Editing Elements	455
Resizing Elements	456
Moving Control Points	456
Moving Elements	457
Maintaining Line Shape when Moving or Stretching Elements	457
Changing the Format of a Single Element	458
Copying Formatting from one Element to Another	461
Changing the Format of a Metaclass	462
Making an Element's Formatting the Default Formatting	466
Copying an Element	466

Arranging Elements	469
Removing an Element from the View	471
Deleting an Element from the Model	471
Editing Text	472
Displaying Compartments	473
Selecting Items to Display	473
Displaying Stereotype of Items in List	474
Zooming	474
Zooming In and Zooming Out	475
Scaling a Diagram	475
Panning a Diagram	475
Undoing a Zoom	475
Specifying the Specification or Structured View	476
Using the Bird's Eye (Diagram Navigator)	477
Showing/Hiding the Bird's Eye Window	477
Navigating to a Specific Area of a Diagram	477
Using the Bird's Eye to Enlarge/Shrink the Visible Area	477
Scrolling/Zooming in Drawing Area	478
Changing the Appearance of the Viewport	478
General Characteristics of the Bird's Eye Window	478
Completing Relations	479
Using IntelliVisor	479
Activating IntelliVisor	480
IntelliVisor Information	480
Customizing Rhapsody	487
Adding Helpers	488
Using the Helpers Dialog Box	489
Creating a Link to a Helper Application	490
Adding a VBA Macro	497
Creating Your Own Profile	499
Creating a New Stereotype for the New Profile	501
Re-using Your Customized Profile	501
Adding New Element Types	502
New Terms and Their Properties	502
Availability of Out-of-the-Box Model Elements	503
Creating a Customized Diagram	506
Adding Customized Diagrams to the Diagrams Toolbar	507
Creating a Customized Diagram Element	507
Adding Customized Diagram Elements	509

Table of Contents

Diagram Types	509
Diagram Elements.....	510
Use Case Diagrams	515
Use Case Diagrams Overview	515
Creating Use Case Diagram Elements	516
Use Case Diagram Drawing Toolbar.....	516
System Boundary Box.....	517
Use Cases.....	517
Actors	520
Packages	523
Associations	524
Generalizations	524
Dependencies	525
Sequences	525
Object Model Diagrams	527
Object Model Diagrams Overview	527
Creating Object Model Diagram Elements	528
Object Model Diagram Drawing Toolbar.....	528
Objects	529
Classes	542
Composite Classes	543
Packages	544
Inheritance	545
Realization	547
Associations	548
Links	561
Dependencies	568
Actors	573
Flows and FlowItems	574
Files.....	584
Attributes, Operations, Variables, Functions, and Types	595
Flow Ports.....	596
External Elements.....	598
Implementing Base Classes	610
Namespace Containment	616
Activity Diagrams	619
Activity Diagram Features	619
Advanced Features	621

Actions	621
Creating Activity Diagram Elements	622
Activity Diagram Drawing Icons	622
Drawing an Action	623
Modifying the Features of an Action	625
Displaying an Action	626
Action Blocks	626
Subactivities	629
Termination States	629
Object Nodes	631
Adding Call Behaviors	633
Activity Flows or Transitions	633
Connectors	635
Synchronization Bars	637
Swimlanes	641
Adding Calls to Behaviors	645
Adding Action Pins / Activity Parameters to Diagrams	647
Local Termination Semantics	649
Code Generation	651
Functor Classes	651
Limitations and Specified Behavior	653
Flow Charts	655
Defining Algorithms with Flow Charts	655
Flow Charts Similarity to Activity Diagrams	656
Creating Flow Chart Elements	657
Flow Chart Drawing Icons	657
Actions	658
Action Blocks	661
Termination States	664
Send Action State elements	665
Activity Flows	666
Connectors	668
Code Generation	669
Limitations and Specified Behavior	669
Sequence Diagrams	671
Sequence Diagram Layout	672
Names Pane	673
Message Pane	674
Analysis Versus Design Mode	674

Table of Contents

Showing Unrealized Messages	675
Realizing a Selected Element	675
Creating Sequence Diagram Elements	676
Sequence Diagram Drawing Icons	676
Creating a System Border	678
Creating an Instance Line	679
Creating a Message	683
Creating a Reply Message	693
Drawing a Create Arrow	696
Creating a Destroy Arrow	696
Creating a Condition Mark	696
Creating a Timeout	697
Creating a Cancelled Timeout	697
Creating an Actor Line	698
Specifying a Time Interval	698
Creating a Dataflow	699
Creating a Partition Line	701
Creating an Interaction Occurrence	701
Creating Interaction Operators	704
Creating Execution Occurrences	706
Shifting Diagram Elements with the Mouse	707
Display Options	708
Sequence Diagrams in the Browser	709
Sequence Comparison	711
Sequence Comparison Algorithm	712
Using Sequence Comparison	713
Setting Sequence Comparison Options	715
The General Tab	716
The Message Selection Tab	718
The Instance Groups Tab	722
The Message Groups Tab	727
Statecharts	733
States	734
Statechart Drawing Icons	735
Drawing a State	736
State Name Guidelines	737
Modifying the Features of a State	738
Display Options for States	740

Termination States	741
Transitions	744
Creating a Statechart Transition	744
Modifying the Features of a Transition	745
Types of Transitions	746
Selecting a Trigger Transition	748
Transition Labels	749
Triggers	749
Guards	755
Actions	757
Events and Operations	758
Sending Events Across Address Spaces	759
Setting Properties	759
API for Sending Events across Address Spaces	760
Functions for Serialization/Unserialization	761
Send Action State Elements	763
Defining Send Action State Elements	763
Display Options	764
Graphical Behavior	764
Code Generation	764
Default Transitions	765
And Lines	766
Connectors	767
Condition Connectors	768
History Connectors	769
Junction Connectors	770
Diagram Connectors	770
Termination Connectors	771
EnterExit Points	771
Submachines	773
Creating a Submachine	773
Opening a Submachine or Parent Statechart	773
Creating a Deep Transition	773
Merging a Sub-Statechart into its Parent Statechart	774
Statechart Semantics	775
Single Message Run-to-Completion Processing	775
Enabled Transitions	776
Transition Selection	776
Transition Execution	778
Active Classes without Statecharts	778

Table of Contents

Single-Action Statecharts	778
Inherited Statecharts	780
Types of Inheritance	781
Inheritance Color Coding	781
Inheritance Rules	781
Overriding Inheritance Rules	785
Overriding Textual Information	786
Refining the Hierarchy of Reactive Classes	787
IS_IN Query	790
Message Parameters	792
Modeling Continuous Time Behavior	794
Interrupt Handlers	794
Inlining Statechart Code	795
Panel Diagrams	797
Panel Diagram Features	797
General Procedure for Using the Panel Diagram	799
Creating Panel Diagram Elements	800
Panel Diagram Drawing Tools	800
Bubble Knob Control	801
Gauge Control	802
Meter Control	803
Level Indicator Control	804
Matrix Display Control	805
Digital Display Control	806
LED Control	807
On/Off Switch Control	808
Push Button Control	809
Button Array Control	810
Text Box Control	811
Slider Control	812
Binding a Control Element to a Model Element	813
More about Binding a Control Element	815
Changing the Settings for a Control Element	817
Changing the Properties for a Control Element	819
Properties for a Bubble Knob Control	820
Properties for a Gauge Control	822
Properties for a Meter Control	825
Properties for a Level Indicator Control	828

Properties for a Matrix Display Control	830
Properties for a Digital Display Control	830
Properties for a LED Control	831
Properties for a On/Off Switch Control	832
Properties for a Slider Control	833
Setting the Value Bindings for a Button Array Control	835
Changing the Display Name for a Control Element	836
Limitations	837
Structure Diagrams	839
Structure Diagram Drawing Icons	840
Composite Classes	840
Objects	840
Ports	841
Links	841
Dependencies	841
Flows	841
Creating an Object	842
Modifying the Features of an Object	843
Changing the Order of Objects	845
Supported Rhapsody Functionality in Objects	846
External Files	846
Collaboration Diagrams	847
Collaboration Diagrams Overview	847
Collaboration Diagram Toolbar	849
Classifier Roles	850
Multiple Objects	851
Actors	851
Links	852
Modifying the Features of a Link	853
Changing the Underlying Association	855
Link Messages and Reverse Link Messages	855
Component Diagrams	857
Component Diagram Uses	858

Component Diagram Drawing Icons	859
Elements of a Component Diagram	860
Components	860
Files	862
Folders	868
Dependencies	871
Component Interfaces and Realizations	871
Flows	872
Setting Component Configurations in the Browser	873
Modifying and Working with Components	873
Active Component	874
Configurations	874
Making Permanent Changes to the Main File	884
Creating Components under a Package	885
Deployment Diagrams	887
Deployment Diagram Drawing Toolbar	888
Nodes	889
Creating a Node	889
Changing the Owner of a Node	890
Designating a CPU Type	891
Modifying the Features of a Node	891
Component Instances	892
Adding a Component Instance	892
Moving a Component Instance	894
Modifying the Features of a Component Instance	894
Dependencies	895
Adding a Dependency Using the Toolbar	895
Adding a Dependency Using the Browser	896
Flows	896
Assigning a Package to a Deployment Diagram	897
Checks	899
Checker Features	899
Using the Checks Tab	900
Specifying Which Checks to Perform	902
Checking the Model	903
Checks Tab Limitations	903

User-defined Checks	904
How to Create User-defined Checks	904
How to Remove User-defined Checks	905
How to Deploy User-defined Checks	906
External Checks Limitations	906
List of Rhapsody Checks	907
Basic Code Generation Concepts	921
Code Generation Overview	921
The Code Toolbar	923
Generating Code	923
Incremental Code Generation	923
Smart Generation of Packages	924
Generating Code Guidelines	925
Dynamic Model-Code Associativity	925
Generating Makefiles	925
Aborting Code Generation	926
Targets	927
Building the Target	927
Deleting Old Objects Before Building Applications	928
Running the Executable	929
Shortcut for Creating an Executable	929
Instrumentation	929
Stopping Model Execution	929
Generating Code for Individual Elements	930
Using the Code Menu	930
Using the Browser	930
Using an Object Model Diagram	930
Results of Code Generation	931
Output Messages	931
Locating and Fixing Compilation Errors	931
Viewing and Editing the Generated Code	932
Setting the Scope of the Code View Editor	932
Adding Line Numbers	933
Editing Code	934
Locating Model Elements	934
Regenerating Code in the Editor	935
Associating Files with an Editor	935
Using an External Editor	936
Viewing Generated Operations	936

Deleting Redundant Code Files	937
Generating Code for Actors	937
Selecting Actors Within a Component	938
Limitations on Actors' Characteristics	938
Generating Code for Component Diagrams	939
Cross-Package Initialization	941
Class Code Structure	943
Class Header File	943
Implementation Files	949
Changing the Order of Operations/Functions in Generated Code	953
Using Code-Based Documentation Systems	955
Template Properties	955
Sample Usage	956
Wrapping Code with #ifdef-#endif	960
Overloading Operators	960
Using Anonymous Instances	965
Creating Anonymous Instances	965
Deleting Anonymous Instances	966
Deleting Components of a Composite	966
Using Relations	967
To-One Relations	967
To-Many Relations	967
Ordered To-Many Relations	968
Qualified To-Many Relations	968
Random Access To-Many Relations	969
Support for Static Architectures	970
Properties for Static Memory Allocation	971
Static Memory Allocation Algorithm	973
Static Memory Allocation Conditions	974
Static Memory Allocation Limitations	974
Using Standard Operations	975
Applications for Standard Operations	975
Creating Standard Operations	977
Statechart Serialization	981
Generating Methods for Serialization	981
Serialization Properties	981
Methods Provided for Implementing Serialization	982
Components-based Development in RiC	984
Action Language for Code Generation	985

C Optimization	986
Backward Compatibility	987
Limitations	987
Customizing C Code Generation	989
Code Customization Concepts	989
Customizing Code Generation	990
Viewing the Simplified Model	991
Customizing the Generation of the Simplified Model	991
Properties Used for Simplification	991
Customizing the Code Writer	992
Customizing the RiC Rules	993
Deploying the Changed Rules	996
Reverse Engineering	997
Using the Reverse Engineering Tool	998
Initializing the Reverse Engineering Dialog Box	1008
Excluding Particular Files	1009
Reverse Engineering Tool Limitations	1009
Visualization of External Elements	1010
Defining Preprocessor Symbols	1011
Adding a Preprocessing Symbol	1012
Analyzing #include Files	1019
Mapping Classes to Types and Packages	1024
Specifying Directory Structures	1029
Specifying Reference Classes	1030
Reference Classes	1032
Locating a Directory that Contains Reference Classes	1033
Miscellaneous Options	1034
Modeling Classes as Rhapsody Types	1037
Reflect Data Members	1042
Error Handling	1045
Updating Existing Packages	1046
Command-Line Interface for Populate Object Model Diagrams	1047
Populate Object Model Diagrams Limitations	1047
Message Reporting	1048
Code Respect and Reverse Engineering for Rhapsody in C and C++	1050
Reverse Engineering for C++	1050

Reverse Engineering for Rhapsody in Java	1050
Reverse Engineering Other Constructs	1051
Unions	1051
Enumerated Types	1051
Comments	1052
Limitations for Comments	1052
Macro Collection	1053
Collected Macro File	1053
Code Generation	1054
Controlling Macro Collection	1054
Code Generation of Imported Macros	1055
Limitations	1055
Backward Compatibility Issues	1056
Results of Reverse Engineering	1056
Lost Constructs	1057
Roundtripping	1059
Supported Elements	1060
Roundtripping Limitations	1060
Dynamic Model-code Associativity (DMCA)	1061
The Roundtripping Process	1062
Automatic and Forced Roundtripping	1062
Roundtripping Classes	1062
Modifying Code Segments for Roundtripping	1063
Recovering Lost Roundtrip Annotations	1064
Roundtripping Classes	1065
Roundtripping Packages	1067
Roundtripping Deletion of Elements from the Code	1069
Roundtripping for C++	1070
Roundtripping for Java	1071
Roundtripping Properties	1071
Code Respect	1073
Activating the Code Respect Feature	1074
Where Code Respect Information is Defined	1075
Making SourceArtifacts Appear on the Rhapsody Browser	1076
Manually Adding a SourceArtifact	1077
Reverse Engineering and SourceArtifacts	1077
Roundtripping and SourceArtifacts	1077

Code Generation and SourceArtifacts	1078
Configuration Management and SourceArtifacts	1078
Animation	1079
Animation Overview	1080
Animation Features	1080
General Procedure to Prepare for Animation	1080
Creating a Component	1081
Setting the Component Features	1082
Creating a Configuration	1083
Setting the Instrumentation Mode	1084
Running the Animated Model	1085
Running on the Host	1085
Running on a Remote Target	1086
Opening a Port Automatically	1087
Testing an Application on a Remote Target	1087
Testing a Library	1088
Partially Animating a Model (C/C++)	1088
Setting Elements for Partial Animation	1089
Partial Animation Considerations	1089
Partially Animated Sequence Diagrams	1090
Ending an Animation Session	1091
Animation Toolbar	1092
Creating Initial Instances	1093
Break Command	1094
Command Prompt	1094
Generating Events Using the Animation Command Bar	1094
Events with Arguments	1095
Generating Events Using the Command History List	1096
Threads	1097
Thread View	1097
Setting the Thread Focus	1097
Names of Threads	1098
Notes on Multiple Threads	1098
Active Thread Properties	1099
Breakpoints	1100
Defining Breakpoints	1101
Enabling and Disabling Breakpoints	1104
Deleting Breakpoints	1104

Table of Contents

Event Generator	1105
Events History List	1106
Calling Animation Operations	1107
Scheduling and Threading Issues	1109
Using Partial Animation	1109
Restrictions	1109
Animation Modes	1111
Silent Mode	1111
Watch Mode	1111
Viewing the Model	1112
Call Stack	1113
Event Queue	1113
Animated Browser	1114
Animated Sequence Diagrams	1114
Animated Statecharts	1124
Instance Names	1126
Names of Class Instances	1126
Names of Component Instances	1126
Navigation Expressions	1127
Names of Special Objects	1127
Animation Scripts	1127
Sample Script	1128
Running Scripts Automatically	1129
Black-Box Animation	1130
Animation Properties	1130
Example	1131
Using the Properties for Black-Box Testing	1134
Instance Line Pop-Up Menu	1135
Behavior and Restrictions	1135
Animation Hints	1136
Exception Handling	1136
If Animation and Application are Out of Sync	1136
Passing Complex Parameters	1137
Combining Animation Settings in the Same Model	1137
Limitations	1137
Guidelines for Writing Serialization Functions	1138
AnimSerializeOperation	1138
AnimUnserializeOperation	1140
Running an Animated Application Without Rhapsody	1141

Tracing	1143
Tracer Capabilities	1143
Starting a Trace Session	1144
Controlling Tracer Operation	1144
Accessing Tracer Commands	1145
Using a File to Enter Tracer Commands	1145
Using Threads	1146
Tracer Commands	1147
break	1147
CALL	1150
display	1152
GEN	1152
go	1153
help	1154
input	1154
LogCmd	1155
output	1156
quit	1156
resume	1157
set focus	1157
show	1158
suspend	1161
timestamp	1161
trace	1161
watch	1165
Tracer Messages by Subject	1166
Ending a Trace Session	1168
Managing Web-enabled Devices	1169
Using Web-enabled Devices	1169
Setting Model Elements as Web-Manageable	1170
Limitations on Web-Enabling Elements	1171
Selecting Elements to Expose to the Internet	1172
Connecting to the Web Site from the Internet	1174
Navigating to the Model through a Web Browser	1174
The Web GUI Pages	1176
Viewing and Controlling a Model via the Internet	1181
Customizing the Web Interface	1182
Adding Web Files to a Rhapsody Model	1182

Table of Contents

Accessing Web Services Provided with Rhapsody	1183
Adding Rhapsody Functionality to Your Web Design	1188
Customizing the Rhapsody Web Server	1192
Reports	1195
ReporterPLUS	1195
Launching ReporterPLUS	1196
ReporterPLUS Templates	1196
Generating Reports Using Existing Templates	1200
Viewing Reports Online	1200
Generating a List of Specific Items	1201
Using the System Model Template	1201
Using the Internal Reporting Facility	1203
Producing an Internal Report	1203
Setting the RTF Character Set	1205
Using the Internal Report Output	1205
Java-specific Issues	1207
Generation of Javadoc Comments	1207
Including Javadoc Comments in Rhapsody-generated Code	1207
Changing the Appearance of Javadoc Comments in Generated Code	1208
Enabling/Disabling Javadoc Comment Generation	1208
"Built-in" Keywords	1209
Description Templates in JavaDocProfile	1209
Multiple Appearance of Javadoc Tags	1209
Adding New Javadoc Tags	1210
Javadoc Handling in Reverse Engineering and Roundtripping	1211
Javadoc Troubleshooting	1211
Static Import	1212
Adding Static Imports to a Model	1212
Reverse Engineering / Roundtripping and Static Import Statements	1212
Code Generation Checks	1213
Static Blocks	1213
Adding Static Blocks to Classes in a Model	1213
Changing a Static Block to an Operation	1213
Reverse Engineering / Roundtripping and Static Blocks	1214
Generating JAR Files	1214
Java 5 Annotations	1215
Creating a JavaAnnotation type	1216
Using a JavaAnnotation type	1217
Using a JavaAnnotation within a model	1218

Code Generation and Java 5 Annotations	1220
Reverse Engineering and Java 5 Annotations	1220
Limitations for Java 5 Annotations	1221
Java Reference Model.	1221
Systems Engineering with Rhapsody	1223
Installing and Launching Systems Engineering.	1223
Creating a SysML Profile Project	1225
SysML Profile Features	1226
SysML Profile Packages	1227
Harmony Process and Toolkit	1228
Harmony Process Summary	1228
Creating a Harmony Project	1230
Harmony Toolkit	1231
Systems Engineering Requirements in Rhapsody	1240
Analysis and Requirements using Gateway	1241
Creating Rhapsody Requirements Diagrams	1243
Creating Specialized Requirement Types	1246
Requirements Tabular View	1247
Creating Use Case Diagrams	1249
Boundary Box and the Environment	1249
Actors and Systems Design in Use Cases	1249
Use Case Features for Systems Engineering	1250
Associating Actors with Use Cases	1252
Defining Requirements in Use Case Diagrams	1252
Tracing Requirements in Use Case Diagrams	1252
Dependencies between Requirements and Use Cases	1253
Defining Flow in a Use Case Diagram	1253
Defining the Stereotype of a Dependency	1253
Activity Modeling in SysML	1254
Action Types in SysML	1254
SysML Activity Diagrams	1254
Creating an Activity Diagram	1255
Setting Activity Diagram Properties	1255
Activity Diagram Drawing Icons for Systems Engineering	1256
Drawing Action States	1257
Drawing a Default Flow	1258
Drawing a Subactivity	1258
Drawing Transitions	1258
Drawing Transitions Between States	1259
Drawing Swimlanes	1259

Table of Contents

Drawing a Fork Synchronization	1259
Drawing a Join Synchronization	1260
Creating a Sequence Diagram from an Activity Diagram	1260
Creating a Design Structure	1261
Creating a Block Definition Diagram	1262
Block Definition Diagram Drawing Tools	1263
Block Definition Diagram Graphics	1265
Creating an Internal Block Diagram	1267
Internal Block Diagram Drawing Icons	1268
Drawing the Parts	1268
Drawing Standard Ports and Links	1269
Specifying the Port Contract and Attributes	1269
Parametric Diagrams	1270
Parametric Diagram Drawing Icons	1271
Creating the Constraint Block	1272
Creating the Parametric Diagram	1273
Binding Constraint Properties Together	1273
Adding Equations	1274
Implementation Using the Action Language	1275
Basic Syntax Rules	1275
Frequently Used Statements	1276
Reserved Words	1276
Assignment and Arithmetic Operations	1277
Defining an Action using the Action Language	1277
Checking Action Language Entries	1278
Action Language Reference	1279
System Validation	1284
Creating a Component	1284
Setting the Component Features	1285
Creating a Configuration	1285
Preparing to Web-enable the Model	1286
Creating a Web-Enabled Configuration	1286
Selecting Elements to Web-enable	1288
Connecting to the Web-enabled Model	1289
Navigating to the Model through a Web Browser	1289
Viewing and Controlling a Model	1290
Sending Events to Your Model	1290
Importing System Architect's DoDAF Diagrams	1291
Mapping the Import Scope	1291
Importing the SA Elements	1292
Converting Imported Data into a Rhapsody Diagram	1293

Post Processing Mechanism for SA Users	1294
Generating a Imported Elements Report.	1294
Integration with Teamcenter Systems Engineering	1295
UML or SysML	1295
Prerequisites for Working with Rhapsody	1297
Importing a Rhapsody Model into Teamcenter	1297
Creating a Rhapsody Model from Existing Teamcenter Project	1298
Modifying Shared Elements from within Teamcenter	1298
Limitations	1299
 Rhapsody DoDAF Add-on	 1301
Rhapsody DoDAF Add-on and Profile	1301
DoDAF Views	1302
Operational View	1303
Systems View	1303
Technical View	1304
All Views	1304
Products Included in the Rhapsody DoDAF Add-on	1305
DoDAF Add-on Helper Utilities.	1309
Setup DoDAF Packages	1311
Create OV-2 from Mission Objective.	1311
Create OV-6c from Mission Objective.	1311
Update OV-2 from OV-6c	1311
Generate Service Based OV-3 Matrix.	1311
Generate SV-3 Matrix	1311
Generate SV-5 Summary Matrix.	1311
Generate SV-5 Full Matrix.	1312
DoDAF Report Generator	1312
Configuring a Rhapsody Project for DoDAF.	1314
Creating a Rhapsody DoDAF Project	1314
Diagrams Toolbar for a Rhapsody DoDAF Project	1317
DoDAF Tags	1319
Generating the OV-3 Operational Information Exchange Matrix.	1321
Generating the DoDAF Report from the Architecture Model.	1323
Limitations	1325
Troubleshooting	1326
Verifying the Rhapsody DoDAF Add-on Installation	1326
Manually Adding the DoDAF Helpers	1327
Correcting Messages that Appear as Mission Objectives.	1330
View, Caption, or Table of Figures is Missing from Document	1332

Rhapsody MODAF Add-on	1333
Rhapsody MODAF Pack	1334
MODAF Viewpoints	1335
All Views Viewpoint	1337
Strategic Viewpoint	1337
Operational Viewpoint	1337
Systems Viewpoint	1338
Acquisition Viewpoint	1338
Technical Viewpoint	1338
Views Included in the Rhapsody MODAF Pack	1339
Configuring a Rhapsody Project for MODAF	1347
Creating a Rhapsody MODAF Project	1347
Customizing the Rhapsody Table and Matrix Views for MODAF	1352
Creating Stereotypes and Using Tags	1353
About Creating Table/Matrix Views in MODAF	1354
Creating Documentation for Your MODAF Project With ReporterPLUS	1360
Setting Up ReporterPLUS	1360
Document Structure	1361
Generating a MODAF document	1362
Troubleshooting ReporterPLUS and MODAF	1364
Using the Dependencies Linker	1366
Troubleshooting the Dependencies Linker	1367
General Troubleshooting	1368
Verifying the Rhapsody MODAF Add-on Installation	1368
Finding Missing Icons on Drawing toolbar	1369
Checking Your MODAF Model	1369
Rhapsody's Automotive Industry Tools	1371
AUTOSAR Modeling	1371
The AUTOSAR Workflow	1371
Creating an AUTOSAR Project	1372
Creating AUTOSAR Diagrams	1372
Checking an AUTOSAR Model	1372
Import/Export from/to AUTOSAR XML Format	1373
The AutomotiveC Profile	1374
Extended Execution Model	1374
Automotive-specific Stereotypes	1378
AutomotiveC Properties	1381
Fixed-point Variable Support	1381
Simulink and StateMate Block Integration Capabilities	1381

StateMateBlock in Rhapsody	1383
Preparing a StateMateBlock for Rhapsody	1383
Creating the StateMateBlock in Rhapsody	1384
Connecting and Synchronizing StateMate and Rhapsody	1385
Troubleshooting StateMate with Rhapsody	1386
DOORS Interface	1387
Installation Requirements	1388
Using DOORS Version 7.0	1388
Solaris-Specific Information	1388
Using Rhapsody with DOORS	1389
Navigating from DOORS to Rhapsody	1390
Creating the DOORS Project	1390
Invoking the DOORS Interface	1390
Mapping Requirements to Imported Elements	1402
Ending a DOORS Session	1403
Other Information	1403
Summary	1404
Importing Rose Models	1405
Importing a Rose Model	1406
Selecting Elements to Import	1407
Setting Import Options	1408
Starting the Import	1409
Incremental Import of Rational Rose Models	1410
Mapping Rules	1412
Importing Association Classes	1417
XMI Exchange Tools	1419
Using XMI in Rhapsody Development	1419
Exporting a Model to XMI	1420
Examining the Exported File	1421
Importing an XMI File to Rhapsody	1422
More Information	1423
Integrating Simulink Components	1425

Importing Simulink Components	1426
Integration of the Simulink-generated Code	1427
Troubleshooting Simulink Integration	1428
Creating Simulink S-functions with Rhapsody	1429
Using Rhapsody in Conjunction with Simulink	1429
Creating a Simulink S-function	1429
S-function Creation: Behind the Scenes	1430
Timing and S-Functions	1430
Limitations	1430
Using the Rhapsody Command-line Interface (CLI)	1433
RhapsodyCL	1433
Interactive Mode	1434
Socket Mode	1434
Command-line Syntax	1435
Switches	1435
Commands	1435
Order of Commands	1436
Including Commands in a Script File	1437
Exiting after Use of Command-line Options	1437
Return Codes	1437
Examples	1439
Command-line Switches	1440
Command-line Commands	1442
Rhapsody Shortcuts	1449
Accelerator Keys	1449
Mnemonics	1449
Keyboard Modifiers	1451
Standard Windows Keyboard Interaction	1451
Rhapsody Accelerator Keys	1451
Application Accelerators	1451
Accelerators and Modifier Usage in Diagrams	1453
Code Editor Accelerators	1455
Useful Rhapsody Windows Shortcuts	1456

Changing Settings to Show the Mnemonic Underlining 1458

Technical Support and Documentation 1459

Contacting Telelogic Rhapsody Support 1459

 Accessing the Automated Problem Report Form 1460

 Automatically Generated Problem Reports 1461

 Calling Telelogic Rhapsody Technical Support. 1461

Contacting IBM Rational Software Support 1462

Accessing the Rhapsody Documentation. 1463

Rhapsody Glossary 1465

Index 1525

Introduction to Rhapsody

Welcome to Telelogic Rhapsody®!

Systems engineers and software developers use Rhapsody to create either embedded or real-time systems. However, Rhapsody goes beyond defining requirements and designing a software solution. Rhapsody actually implements the solution from design diagrams and automatically generating ANSI-compliant code that is optimized for the most widely used target environments.

With Rhapsody, you have the ability to analyze the intended behavior of the application much earlier in the development cycle by generating code from UML and SysML diagrams and testing the application as you create it. Rhapsody can be used for any of the following:

- ◆ Reactivity—Statecharts and events
- ◆ Time-based behavior—Timeouts
- ◆ Multi-threaded architectures—Active classes and protected classes
- ◆ Real-time environments—Direct support for several real-time, operating systems (RTOS)

Rhapsody is semantically complete. Most items that you draw in Rhapsody UML diagrams, such as objects or events, have precise meaning in the underlying model. *Objects* are the structural building blocks of a system.

Rhapsody translates these diagrams into source code in one of four high-level languages: C++, C, Ada, or Java. Rhapsody then allows you to edit the generated code and dynamically roundtrip the changes back into the model and its graphical views.

The *Rhapsody User Guide* provides detailed information and procedures for engineers, architects, designers, and developers using Rhapsody.

Rhapsody Features

Rhapsody includes the following features:

- ◆ UML[®], SysML[™], and Functional C design modeling environment with Domain-Specific Language (DSL) support including DoDAF*, MODAF*, and AUTOSAR*.
- ◆ Predefined *profiles* supplying a coherent set of tags, stereotypes, and constraints for a specific type of project. For more information, see [Profiles](#).
- ◆ MathWorks Simulink[®] Interface, SDL Interface, and Statemate[®] Interface available with the Interfaces Add-on enables you to validate your complete architecture while using the best-in-class tools for control engineering, protocol development, and functional system design.
- ◆ Model verification with full model simulation and execution.
- ◆ Static checking to ensure that the design is consistent.
- ◆ Full application generation of C, C++, Java, and Ada in an integrated design environment.
- ◆ Easily customizable real-time framework that separates high-level application design from platform-specific implementation details with numerous adapters available such as VxWorks, Windows CE, and Integrity. A full list is available in the Rhapsody release notes, and in addition you can create your own adapter.
- ◆ Requirements modeling and traceability features with integration to leading requirements management tools such as DOORS^{®*} or text-based tools such as Microsoft[®] Word.
- ◆ Easily integrate and create models from your existing C, C++, Java, and Ada code into the modeling environment using reverse engineering and code visualization.
- ◆ Integration with leading IDEs such as Eclipse, Wind River[®] Workbench, and Green Hills[®] Multi[®].
- ◆ Dynamic model-code associativity enabling design to be done using either code or diagrams providing maximum flexibility while ensuring the two remain synchronized.
- ◆ Improved test productivity and early detection of defects using Rhapsody[®] TestConductor[™] to automate tedious testing tasks, define tests with code and graphically with sequence diagrams, statecharts, activity diagrams and flowcharts; and execute the tests interactively or in batch mode.
- ◆ XMI* (XML Metadata Interchange) and IBM[®] Rational Rose^{®*} importing for integration of legacy systems and reuse of existing code.
- ◆ Full Configuration Management Interface* support with advanced graphical difference and merging capabilities for use with tools such as Synergy[™] or IBM[®] Rational[®] Clearcase[®].
- ◆ Customization to meet your specific development needs using the Java API.

- ◆ Generation of documentation using a range of tools, from a simple RTF report generator to the full customization with Rhapsody[®] ReporterPLUS[™]*.

* Capabilities are provided by optional add-ons.

Rhapsody Tools

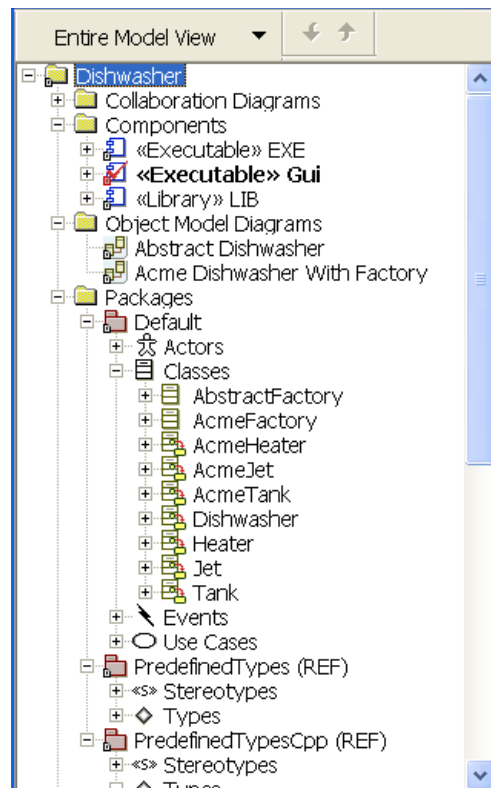
Rhapsody consists of the following set of tools that interact with each other to give you a complete software design environment:

- ◆ [The Rhapsody Browser](#)
- ◆ [The Favorites Browser](#)
- ◆ [Graphic Editors](#)
- ◆ [Code Generator](#)
- ◆ [Animator](#)
- ◆ [Utilities](#), including reverse engineering, Web-enabling devices, XMI generation, COM and CORBA[®] support, and Visual Basic[®] for Applications
- ◆ [Third-party Interfaces](#) such as Eclipse, Rose import, and CM tools

The Rhapsody Browser

The Rhapsody browser provides the most comprehensive display of the system, giving you a clear overview of your entire model. Views filter the display to optimize usability for a particular task. During an animation session, the browser dynamically displays object instances as the model executes. See [Browser](#) and [Browser Filter](#) for more details about the uses of the browser and [Using the Rhapsody Browser](#) for details on the browser's model display features.

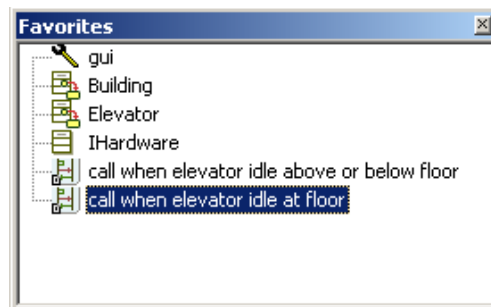
The following figure shows the Rhapsody browser:



The Favorites Browser

Rhapsody models can become very large, making it difficult to find commonly used model elements in the Rhapsody browser. To help you manage large and complex projects, and to be able to focus on and easily access model elements of particular interest to you, you can create a favorites list that displays in the Favorites browser. Like the favorites functionality for a Web browser, in Rhapsody, you can create a list of model elements you want to concentrate on that is displayed in the Favorites browser.

The following figure shows a sample Favorites browser:



Your favorites list is saved in the `<projectname>.rpw` file, as well as the position and visibility of the Favorites browser, so that when you open the project the next time, your settings are automatically in place. Note that when multiple projects are loaded, the Favorites browser shows the favorites list for the *active project*.

For more information about the Favorites browser, see [Using the Favorites Browser](#).

Graphic Editors

The graphic editors enable you to analyze, design, and construct the system using UML diagrams. Diagrams enable you to observe the model from several different perspectives, like turning a cube in your hand to view its different sides. Depending on its focus, a diagram might show only a subset of the total number of classes, objects, relationships, or behaviors in the model. Together, the diagrams represent a complete design.

Rhapsody adds the objects created in diagrams to the Rhapsody project, if they do not already exist. Conversely, Rhapsody removes elements from the project when they are deleted from a diagram. However, you can also add existing elements to diagrams that do not need to be added to the project, and remove elements from a diagram without deleting them from the model repository.

- ◆ **Use case diagram editor** provides tools for creating use case diagrams, which show the use cases of the system and the actors that interact with them. See [Use Case Diagrams](#).
- ◆ **Object model diagram editor** provides tools for creating object model diagrams, which are logical views showing the static structure of the classes and objects in an object-oriented software system and the relationships between them. See [Object Model Diagrams](#).
- ◆ **Sequence diagram editor** provides tools for creating sequence diagrams, which show interactions between objects in the form of messages passed between the objects over time. If you run animated code with the Animator, you can watch messages being passed between objects as the model runs. See [Sequence Diagrams](#).
- ◆ **Collaboration diagram editor** provides tools for creating collaboration diagrams, which describe how different kinds of objects and associations are used to accomplish a particular task. Collaboration diagrams and sequence diagrams are both interaction diagrams that show sequences. Sequence diagrams have a time component, whereas collaboration diagrams do not.

Like sequence diagrams, collaboration diagrams show the message flow between different classes. However, collaboration diagrams emphasize object relationships whereas sequence diagrams emphasize the order of the message flow. For a particular task or interaction, a collaboration diagram can also show the individual objects that are created, destroyed, or exist continuously for the duration of the task. See [Collaboration Diagrams](#).

- ◆ **Statechart editor** provides tools for creating statecharts, which define the behaviors of individual classes in the system.

Statecharts show the states of a class in a given context, events that can cause transitions from one state to another, and actions that result from state transitions. Rhapsody generates function bodies from information entered into statecharts. If you run animated code with the animator, you can watch an object change states as it reacts to various messages, events, and triggered operations that you generate. See [Statecharts](#)

- ◆ **Activity diagram editor** provides tools for creating activity diagrams. Activity diagrams show the lifetime behavior of an object, or the procedure that is executed by an operation

in terms of a process flow, rather than as a set of reactions to incoming events. When a system is not event-driven, use activity diagrams rather than statecharts to specify behavior. See [Activity Diagrams](#).

- ◆ **Component diagram editor** provides tools for creating component diagrams, which show the dependencies among software components, such as library or executable components. Component diagrams can also show component dependencies, such as the files (or other units) that are contained by a component, or the connections or interfaces among components. See [Component Diagrams](#).
- ◆ **Deployment diagram editor** provides tools for creating deployment diagrams, which show the run-time physical architecture of the system. The physical architecture of a running system consists of the configuration of run-time processing elements and the software components, processes, and objects that live on them. A deployment diagram graphs the nodes in the system, representing various processors, connected by communication associations. See [Deployment Diagrams](#).
- ◆ **Structure diagram editor** provides tools for creating structure diagrams, which model the structure of a composite classes. See [Structure Diagrams](#).

Note

All the diagrams use UML notation.

The FunctionalC profile has these diagrams available to construct a C model:

- ◆ Use case diagram
- ◆ Statechart
- ◆ Build diagram
- ◆ Call Graph diagram
- ◆ Flow Chart
- ◆ Message diagram
- ◆ File diagram

Code Generator

The code generator synthesizes complete production-quality code from the model to free you from low-level coding activities. Rhapsody generates code primarily from OMDs and statecharts, but also from activity and other diagrams. Allowing the tool to generate code automatically for you lets you concentrate on higher-level system analysis and design tasks. See [Basic Code Generation Concepts](#) for detailed information.

Animator

The animator lets you debug and verify your software at the design level rather than the compiler level. See [Animation](#) for detailed information.

Utilities

In addition to the core UML-based design features, Rhapsody provides a number of utilities to assist with development including the following:

- ◆ Dynamic reverse engineering (see [Basic Code Generation Concepts](#))
- ◆ Roundtrip (see [Basic Code Generation Concepts](#))
- ◆ Reverse engineering (see [Reverse Engineering](#))
- ◆ Check model (see [Checks](#))
- ◆ Web-enabling of Rhapsody models (see [Managing Web-enabled Devices](#))
- ◆ Standard and customizable report generation with Rhapsody Reporter and ReporterPLUS (see [Reports](#) and the *ReporterPLUS Guide* available from the List of Books on the Help menu)
- ◆ Import of model elements from libraries and external source files (refer to the *Rhapsody API Reference Manual*)
- ◆ **Add to Model** and multiuser collaboration (refer to the *Rhapsody Team Collaboration Guide*)
- ◆ Component download (refer to the *Rhapsody Team Collaboration Guide*)
- ◆ Web Collaboration (refer to the *Rhapsody Team Collaboration Guide*)
- ◆ DiffMerge (see [Parallel Project Development](#) and also the *Rhapsody Team Collaboration Guide*)

Third-party Interfaces

- ◆ Configuration management tools (refer to the *Rhapsody Team Collaboration Guide*)
- ◆ Support for the Microsoft Source Code Control (SCC) standard (refer to the *Rhapsody Team Collaboration Guide*)
- ◆ Import of models created in other tools, such as Rational[®] Rose (see [Importing Rose Models](#))
- ◆ Integrated VBA development environment and macro editor (refer to the *Rhapsody API Reference Manual*)
- ◆ IDE interface to the Tornado[™] development environment (see [Co-Debugging with Tornado](#))
- ◆ Code editors (such as CodeWright[™])
- ◆ Source debuggers (in addition to IDEs)
- ◆ Eclipse (see [Using the Rhapsody Workflow Integration with Eclipse](#))

Rhapsody Windows

When creating or editing a project, the Rhapsody workspace has the following windows:

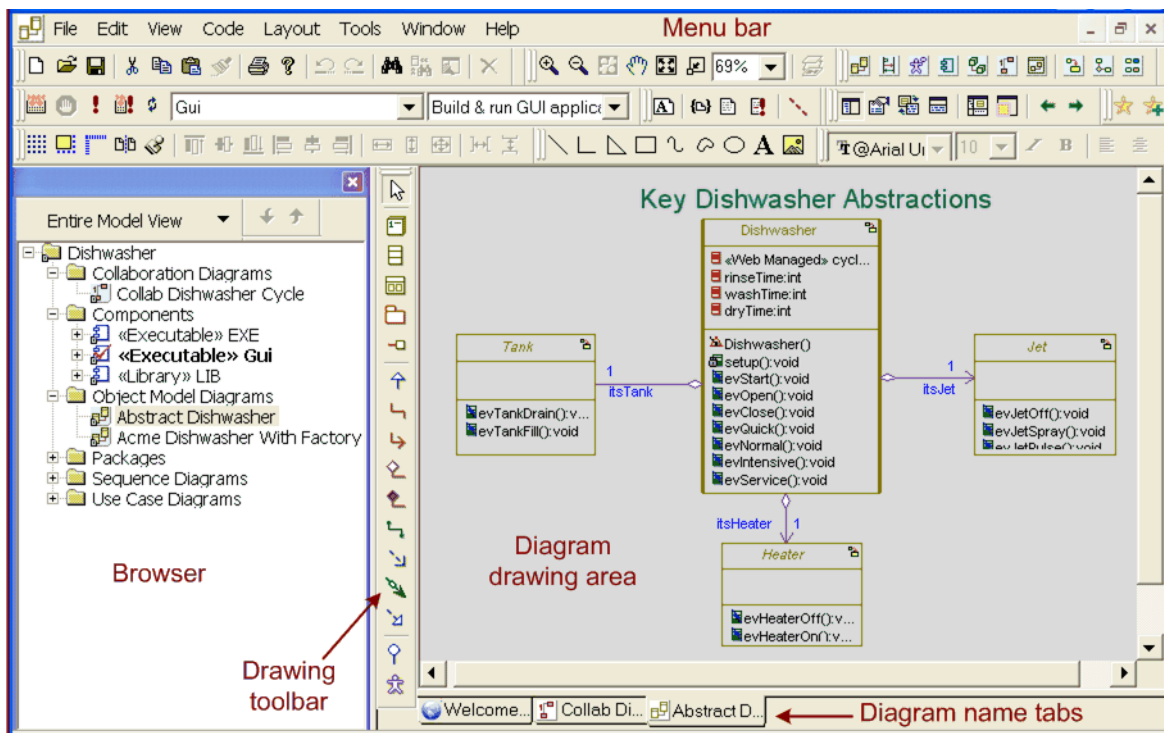
- ◆ **Menu bar** lists the primary functions as File, Edit, View, Code, Tools, Layout, Windows, and Help. Many of the Rhapsody functions available on the menus are also accessible from [Rhapsody Shortcuts](#) and icons across the top of the Rhapsody interface, as described in [Rhapsody Project Tools](#).
- ◆ **Browser** (see [Browser](#)) displays the contents of the project and has several views to choose from.
- ◆ **Diagram drawing area** (see [Diagram Drawing Area](#)) contains the diagram editor windows, which can be moved and resized.
- ◆ **Drawing toolbar** icons in the center of the window. Different icons are displayed depending on the type of diagram displayed in the drawing area. Refer to the descriptions of the diagrams for explanations of each diagram's drawing icons.
- ◆ **Diagram Navigator** (see [Diagram Navigator](#)) provides a bird's eye view of the diagram that is currently displayed.
- ◆ **Output window** (see [Output Window](#)) has several tabs, each displaying different types of Rhapsody output including search results.

In addition, you can open two windows from the View menu:

- ◆ **Features dialog box** (see [Using the Features Dialog Box](#)) displays details of selected model element. By default, it appears as a floating dialog box, but you can dock it to the main window in any position.
- ◆ **Active code view** (see [Active Code View](#)) generates and displays code for the selected model element.

When you open Rhapsody for the first time, the [Welcome Screen](#) displays.

The following figure shows the default arrangement of the Rhapsody windows.



Note the following:

- ◆ You can reposition each window within the Rhapsody workspace to suit your preferred work style.
- ◆ To dock or undock a window quickly, double-click the title bar.
- ◆ To reposition a window, click the title bar and drag-and-drop the window to the desired location.
- ◆ The very bottom of the main window shows the status bar, which displays the current mode (for example, GE MODE) and the date and time. To toggle the display, select **View > Status Bar**.

Browser

The browser displays a hierarchy of your project, and provides easy access to the elements and diagrams it contains. Several view options enable you to filter the display. [Using the Rhapsody Browser](#) provides detailed descriptions of browser elements and views.

By default, the browser is docked at the upper, left corner of the Rhapsody main window.

- ◆ To open the browser, select **View > Browser**.
- ◆ Select **Tools > Browser** open multiple instances of the browser to simplify the process of navigating between elements. This feature is particularly useful during animation.

In addition, you may select multiple elements in the browser and perform any of these operations on all of them:

- ◆ Move them to another browser element by dragging and dropping them over the target element
- ◆ Copy them to another browser element by dragging and dropping them over the target element
- ◆ Delete all of them at the same time

Diagram Drawing Area

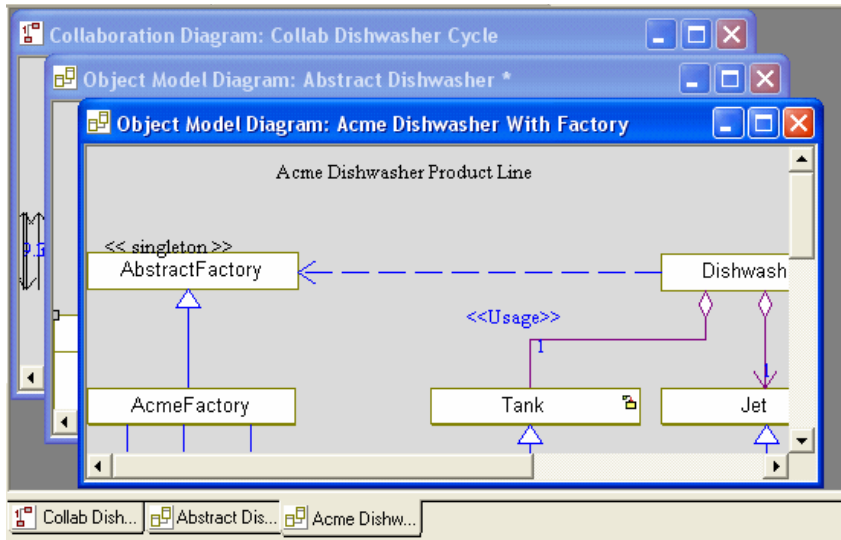
The drawing area displays the graphic editors and code editors. Editors can be moved and resized within the drawing area. When you open more than one editor, tabs are displayed at the bottom of the drawing area so you can easily move between the open diagrams or generated code files. In addition, you can tile or cascade the windows that contain the different diagrams.

Each diagram includes a title bar, which contains the name of the diagram and its type. A modified diagram has an asterisk (*) added to the end of its name in the title bar.

You can turn off tabs in the drawing area by clearing the check mark next to **Workbar Mode** in the View menu.

See [Graphic Editors](#) for a description of the graphic editor windows. See [The Internal Editor](#) for a description of the default code editor.

The following figure shows the drawing area with cascading windows (**Window > Cascade**).



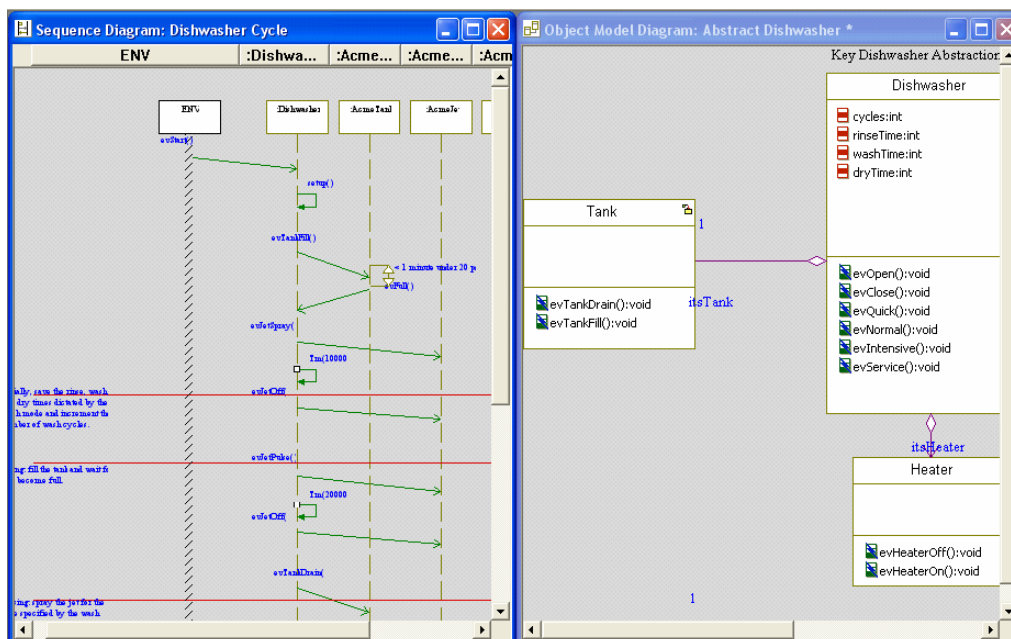
Maintaining the Window Content

When you resize the drawing area (for example, to increase the available drawing space), some of the diagram might move out of the visible area of the window. The part of the diagram displayed in the window is called the *viewport*. You can specify whether to display the viewport regardless of window manipulation using two different methods:

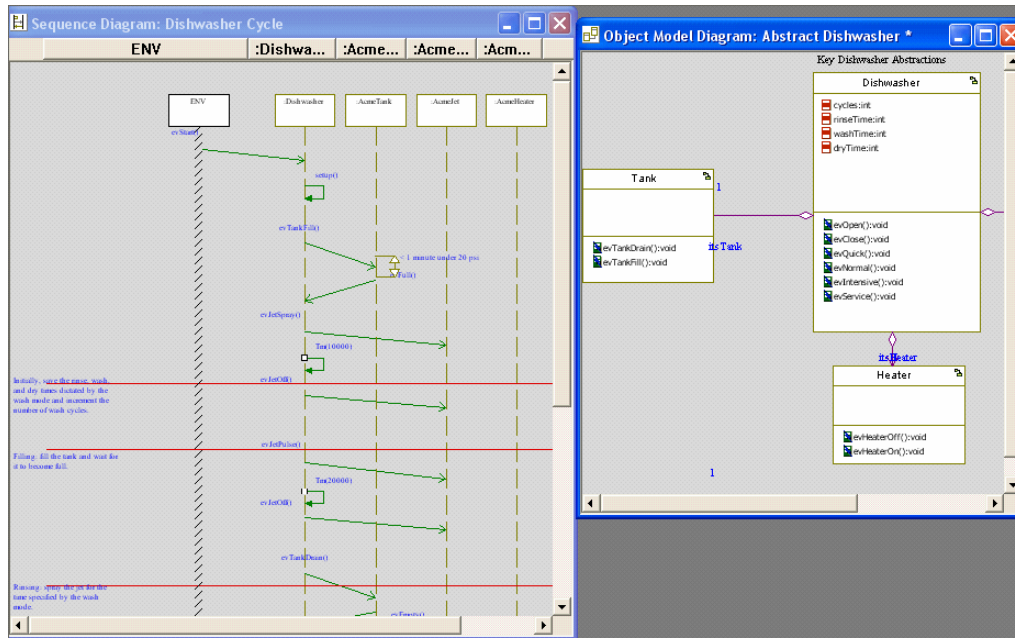
- ◆ Select **View > Maintain Window Content**. A check mark is displayed when this functionality is enabled.
- ◆ Set the property `General::Graphics::MaintainWindowContent` to `Checked`.

When this functionality is enabled, the elements are scaled according to the zoom factor so you see the same elements in the window, regardless of scaling.

For example, the following figure shows the effect of selecting **Window > Tile Vertically** with **Maintain Window Content** turned off. Note that some of the information in the diagrams is hidden from view.



The following figure shows the same operation, with **Maintain Window Content** enabled. Note that the same information is displayed, even though you changed the size of the window.



Note that the following operations change the window size:

- ◆ Maximize/restore
- ◆ Tile
- ◆ Cascade
- ◆ Manual resizing by dragging the edge of the window

Diagram Navigator

The Diagram Navigator provides a bird's eye view of the diagram that is currently being viewed. This can be very useful when dealing with very large diagrams, allowing you to view specific areas of the diagram in the drawing area, while, at the same time, maintaining a view of the diagram in its entirety.

The Diagram Navigator contains a depiction of the entire diagram being viewed, and a rectangle viewport that indicates which portion of the diagram is currently visible in the drawing area.

For detailed information about using the Diagram Navigator window, see [Using the Bird's Eye \(Diagram Navigator\)](#).

Output Window

The Output window is where Rhapsody displays various output messages. Tabs on the Output window enable you to easily navigate among the different types of output messages:

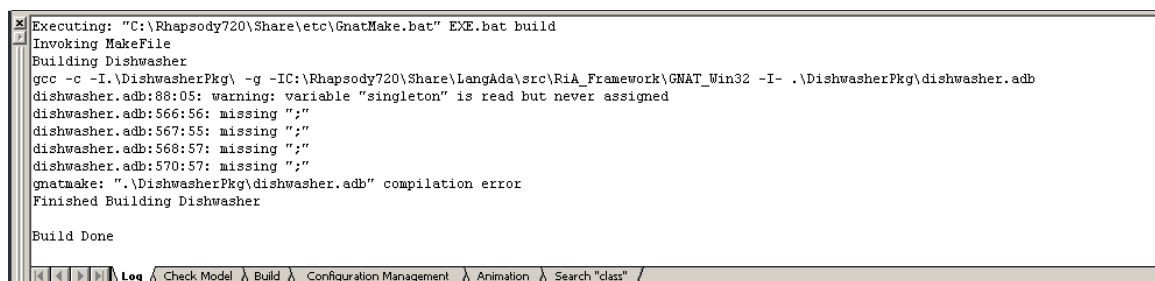
- ◆ The [Log Tab](#) shows all the messages from all the other tabs of the Output window (except for **Search Results**) in text—meaning non-tabular—format.
- ◆ The [Build Tab](#) shows the messages related to building an application in tabular format.
- ◆ The [Check Model Tab](#) shows the messages related to checking the code for a model in tabular format.
- ◆ The [Configuration Management Tab](#) shows the messages related to configuration management actions for a model in text format.
- ◆ The [Animation Tab](#) shows the message related to animating a model in text format.
- ◆ The [Search Results Tab](#) shows the results from searches of your model in tabular format. Note that this tab may not appear until you perform a search.

By default, the Output window is located at the bottom portion of the main Rhapsody window. Also by default, when you generate, build, or run an application; do a search, a CM action, or a check model, Rhapsody opens the Output window. To manually open the Output window, choose **View > Output Window**.

Log Tab

The **Log** tab serves as a console log. It shows all the messages from all the other tabs of the Output window (except for **Search Results**) in text—meaning non-tabular—format. The messages that appear on the **Check Model**, **Build**, **Configuration Management**, and **Animation** tabs appear on the **Log** tab too, but always in text format. For the check model and build functions, you can view their messages on the **Check Model** and **Build** tabs in tabular format or on the **Log** tab in text format, depending on your preference.

The following figure shows the **Log** tab after a build function has been performed. Notice that the output is in text format.



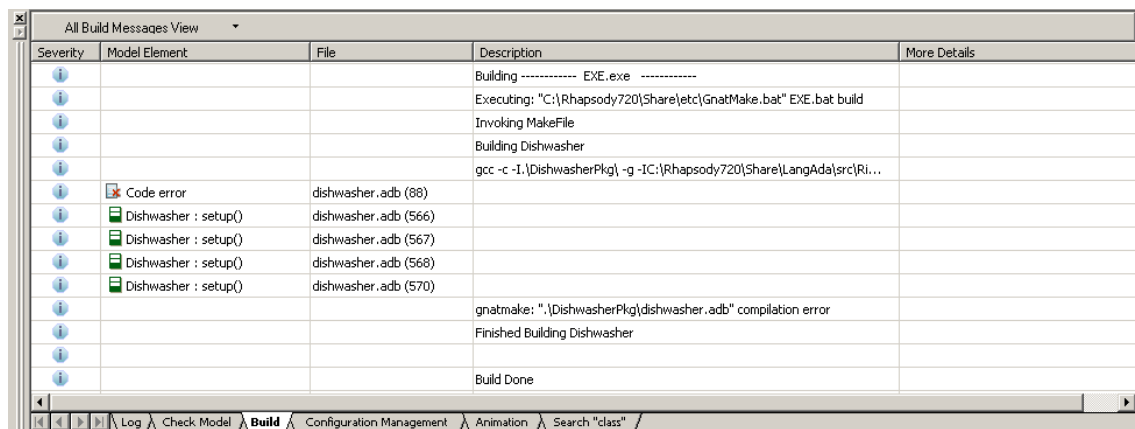
```

Executing: "C:\Rhapsody720\Share\etc\GnatMake.bat" EXE.bat build
Invoking MakeFile
Building Dishwasher
gcc -c -I.\DishwasherPkg\ -g -IC:\Rhapsody720\Share\LangAda\src\RIa_Framework\GNAT_Win32 -I- .\DishwasherPkg\dishwasher.adb
dishwasher.adb:88:05: warning: variable "singleton" is read but never assigned
dishwasher.adb:566:56: missing ";"
dishwasher.adb:567:56: missing ";"
dishwasher.adb:568:57: missing ";"
dishwasher.adb:570:57: missing ";"
gnatmake: ".\DishwasherPkg\dishwasher.adb" compilation error
Finished Building Dishwasher

Build Done
  
```

Note that you can right-click on the **Log** tab to use the **Clear**, **Copy**, **Paste**, and **Hide** commands.

The following figure shows the **Build** tab for the same build function. As you can see the messages provide the same type of information, though the presentation is in a tabular format.



Severity	Model Element	File	Description	More Details
			Building ----- EXE.exe -----	
			Executing: "C:\Rhapsody720\Share\etc\GnatMake.bat" EXE.bat build	
			Invoking MakeFile	
			Building Dishwasher	
			gcc -c -I.\DishwasherPkg\ -g -IC:\Rhapsody720\Share\LangAda\src\RI...	
	Code error	dishwasher.adb (88)		
	Dishwasher : setup()	dishwasher.adb (566)		
	Dishwasher : setup()	dishwasher.adb (567)		
	Dishwasher : setup()	dishwasher.adb (568)		
	Dishwasher : setup()	dishwasher.adb (570)		
			gnatmake: ".\DishwasherPkg\dishwasher.adb" compilation error	
			Finished Building Dishwasher	
			Build Done	

On the **Log**, **Check Model**, and **Build** tabs, you can double-click an item on the tab and, if possible, Rhapsody will open either the relevant model element (for example, the Features dialog box for an association that may be causing an error) or to the code source. From whichever opens, you can make corrections or view the item more closely.




Check Model Tab

Rhapsody analyses and organizes the results of checking the code for a model and displays the results on the **Check Model** tab, as shown in the following figure. Check messages are grouped by a severity hierarchy, and provide you with the location, domain, and integrity for an item, where possible.

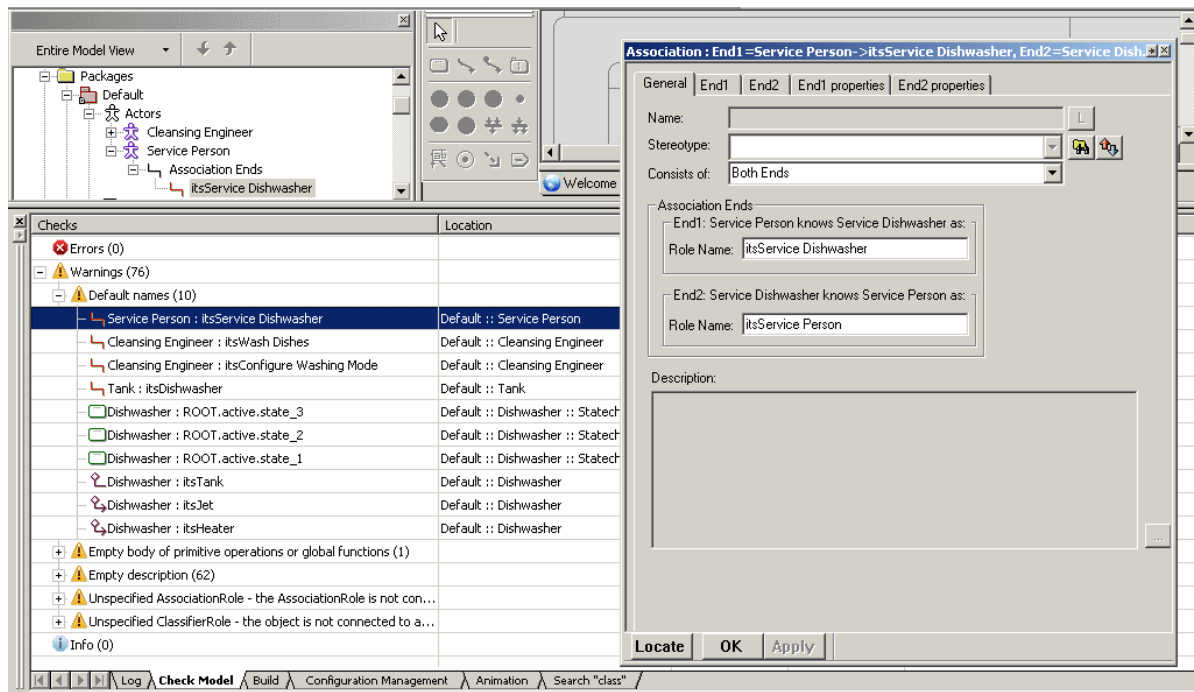
Before generating code, Rhapsody automatically performs certain checks for the correctness and completeness of the model. In addition, you can perform selected checks at any time during the design process. To check the code for a model, choose **Tools > Check Model** and then the name of the configuration for the model. For more information about checking the code for a model, see [Checks](#).

Checks	Location	Domain	Integrity
Errors (0)			
Warnings (76)			
Default names (10)		Common	Complete
Service Person : itsService Dishwasher	Default :: Service Person		
Cleansing Engineer : itsWash Dishes	Default :: Cleansing Engineer		
Cleansing Engineer : itsConfigure Washing Mode	Default :: Cleansing Engineer		
Tank : itsDishwasher	Default :: Tank		
Dishwasher : ROOT.active.state_3	Default :: Dishwasher :: Statechart		
Dishwasher : ROOT.active.state_2	Default :: Dishwasher :: Statechart		
Dishwasher : ROOT.active.state_1	Default :: Dishwasher :: Statechart		
Dishwasher : itsTank	Default :: Dishwasher		
Dishwasher : itsJet	Default :: Dishwasher		
Dishwasher : itsHeater	Default :: Dishwasher		
Empty body of primitive operations or global functions (1)		Class Model	Complete
Empty description (62)		Common	Complete
Unspecified AssociationRole - the AssociationRole is not con...		Class Model	Correct
Unspecified ClassifierRole - the object is not connected to a...		Class Model	Correct
Info (0)			

The following table explains what type of information you see on the **Check Model** tab.

Column	Explanation
Checks	<p>Shows the check results tree grouped by three levels and in the hierarchy shown as follows:</p> <p>Level 1, Severity: Where the elements listed are grouped under the three optional severity levels as follows:</p> <ul style="list-style-type: none">  Errors  Warnings  Info <p>The grouping is determined by the severity level of each added check. If there is a + icon next to a label, click it to expand or collapse the list. In addition, to the right of each severity level name is the number of problems in the security level. As shown in the above figure, there are 76 warnings.</p> <p>Level 2, Checks: Where each check that produces errors/problems when it is executed is shown in the list indented under its relevant severity level (for example, Default Names under Warnings in the above figure). A check is considered a group that holds under it all its related problems. Each check in the list shows also its domain and integrity properties. In addition, to the right of each check name is the number of problems the check contains. As shown in the above figure, there are 10 Default Name warnings.</p> <p>Level 3: Check Elements, where each problem found when performing a specific check is shown as a list item under the relevant check name. Problems are represented by model elements and shown with the relevant name, type icon, and model location path (for example, ServicePerson: itsServiceDishwasher, the first item in the Default Names check group in the Warnings severity level, as shown in the above figure).</p>
Location	Shows, for each problem found, the location of an element in the model.
Domain	Shows, for each check in the list, its domain property. This includes domains that are from user-defined checks.
Integrity	Shows, for each check in the list, its integrity property value.

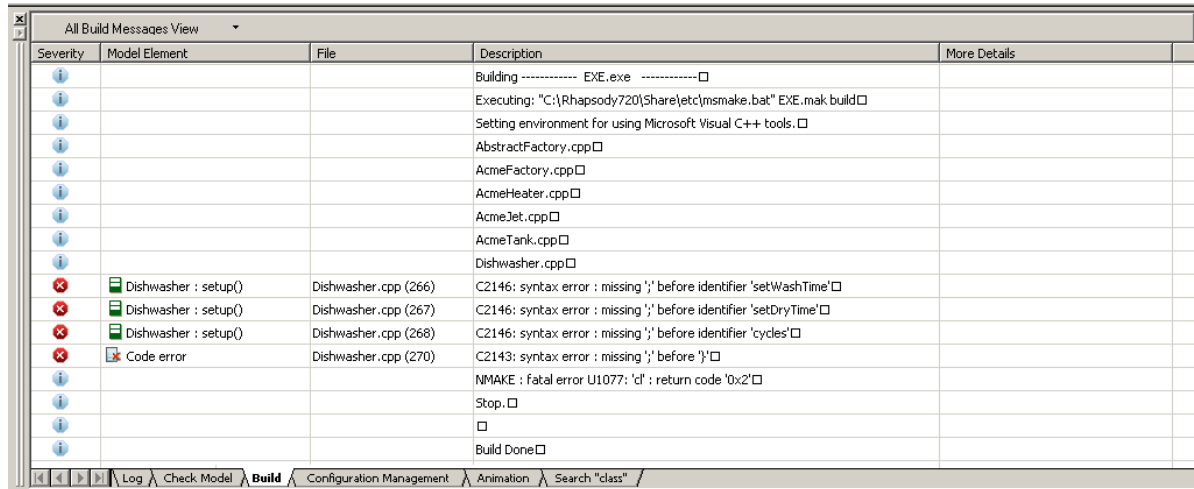
You can double-click an item on the **Check Model** tab and if possible, Rhapsody will bring you to the relevant model element (for example, the Features dialog box for an association, as shown in the following figure) or to the code on which you can make corrections or view the item more closely. Note that if you click a level heading, you expand or collapse the list for the level.






Note that you can right-click on the **Check Model** tab to use the **Copy All** and **Clear All** commands.

Build Tab

The **Build** tab shows the messages related to building an application, as shown in the following figure.



This tabular view shows the following types of information:

Column	Explanation
Severity	<p> Error messages. Errors appear when the model fails to build. You must fix errors before the model can be built. Rhapsody parses the information provided by the compiler to develop the list of error messages. Note that there can be two types of error (and warning) messages: model element and code error. Model element-type errors (and warnings) are those that Rhapsody can correspond to specific model elements in a project. Code error-type errors are those that Rhapsody cannot find any corresponding model element.</p> <p> Warning messages. Warnings have no effect on whether the model is built, but you should review them and address them if necessary as they may have an effect on whether the model builds as expected. Rhapsody parses the information provided by the compiler to develop the list of warning messages.</p> <p> Informational messages. These messages are messages that are not warnings or errors and they have no effect on the building of the model.</p>
Model Element	Applicable to error and warning messages only, shows the Rhapsody model element and its applicable Rhapsody icon. If no related model element is found, the error is assumed to be a code error-type error.
File	Applicable to error and warning messages only, shows the file name and line number where an error/warning was found.
Description	Descriptions are provided by the compiler.
More Details	Applicable to error and warning messages only, and only if available, show more details as provided by the compiler.

By default, you see all the messages for a build, as shown in the above figure. If this is not the case, you can select **All Build Messages View** from the drop-down menu in the upper-left corner of the Output window for the **Build** tab.

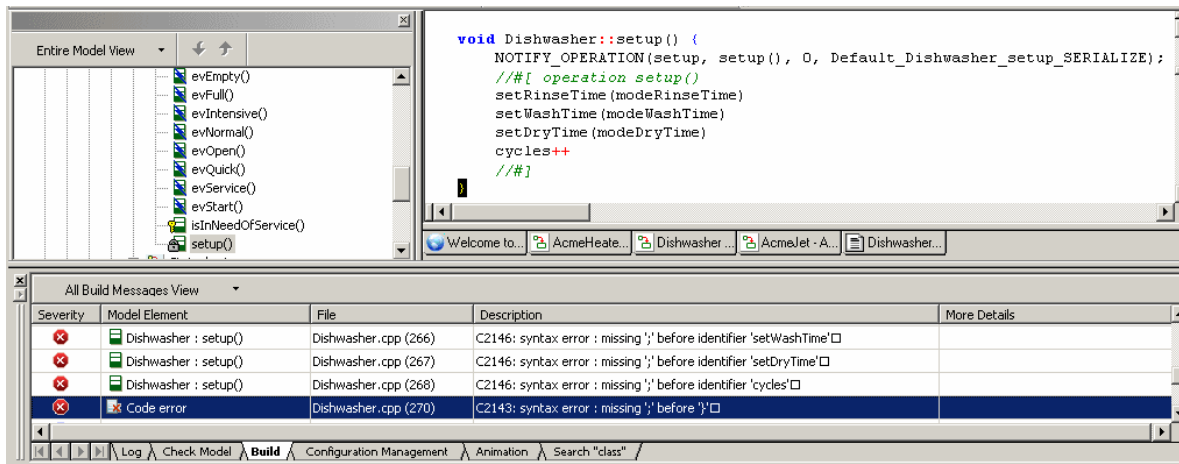
You can use the drop-down menu in the upper-left corner of the Output window for the **Build** tab to choose other views. The following figure shows the **Build** tab with **Model Element Messages** selected, which shows only items with a severity of Error or Warning that also have references to a model element. In addition, information messages, code error-type errors and warnings are filtered out.

Severity	Model Element	File	Description	More Details
✘	Dishwasher : setup()	Dishwasher.cpp (266)	C2146: syntax error : missing ';' before identifier 'setWashTime' □	
✘	Dishwasher : setup()	Dishwasher.cpp (267)	C2146: syntax error : missing ';' before identifier 'setDryTime' □	
✘	Dishwasher : setup()	Dishwasher.cpp (268)	C2146: syntax error : missing ';' before identifier 'cycles' □	

The following figure shows the **Build** tab with **All Errors and Warnings View** selected, which shows only error and warning messages (so that information messages are filtered out).

Severity	Model Element	File	Description	More Details
✘	Dishwasher : setup()	Dishwasher.cpp (266)	C2146: syntax error : missing ';' before identifier 'setWashTime' □	
✘	Dishwasher : setup()	Dishwasher.cpp (267)	C2146: syntax error : missing ';' before identifier 'setDryTime' □	
✘	Dishwasher : setup()	Dishwasher.cpp (268)	C2146: syntax error : missing ';' before identifier 'cycles' □	
✘	Code error	Dishwasher.cpp (270)	C2143: syntax error : missing ';' before '}' □	

Note that you can double-click an item on the tab and, if possible, Rhapsody will open either the relevant model element (for example, the Features dialog box for an association that may be causing an error) or to the code source. From whichever opens, you can make corrections or view the item more closely. In the following figure, double-clicking the “Code Error” item on the **Build** tab in the lower portion of the figure, opens the code for that item in the upper-right portion of the figure.



Note that you can right-click on the **Build** tab to use the **Copy All** and **Clear All** commands.

Note

By default, the **Build** tab displays after you run a build. If you want the **Log** tab to automatically display instead after a build, you can set the `CG::General::ShowLogViewAfterBuild` property to `Checked`.

Supported Compilers

The compilers from Microsoft, Java, and Cygwin are fully supported, which means that Rhapsody is able to analyze the output from their compilers and show the correct severity levels for their messages. For all other compilers, their output will only show Informational messages.

Configuration Management Tab

The **Configuration Management** tab shows messages related to configuration management actions for a model, as shown in the following figure. For more information about configuration management tools in Rhapsody, refer to the *Rhapsody Team Collaboration Guide*.



Note that you can right-click on the **Configuration Management** tab to use the **Clear**, **Copy**, **Paste**, and **Hide** commands.

Animation Tab

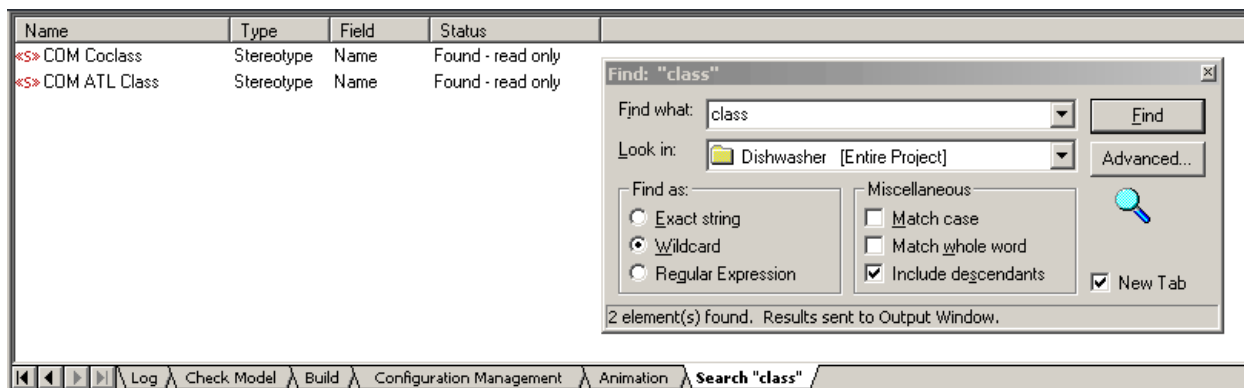
The **Animation** tab shows messages related to animating a model, as shown in the following figure. For more information about animation in Rhapsody, see [Animation](#).



Note that you can right-click on the **Animation** tab to use the **Clear**, **Copy**, **Paste**, and **Hide** commands.

Search Results Tab

The **Search Results** tab shows results from searches of your model, as shown on the left side of the following figure. Note that this tab may not appear until you perform a search (for example, choose **Edit > Search** and select the **New Tab** check box, as shown in the right side of the following figure). For more information about doing searches in Rhapsody, see [Searching Models](#).



Active Code View

The Active Code View window displays code for an element selected in the browser. Whenever you make changes to the model, Rhapsody regenerates the code and updates it in the Active Code View window.

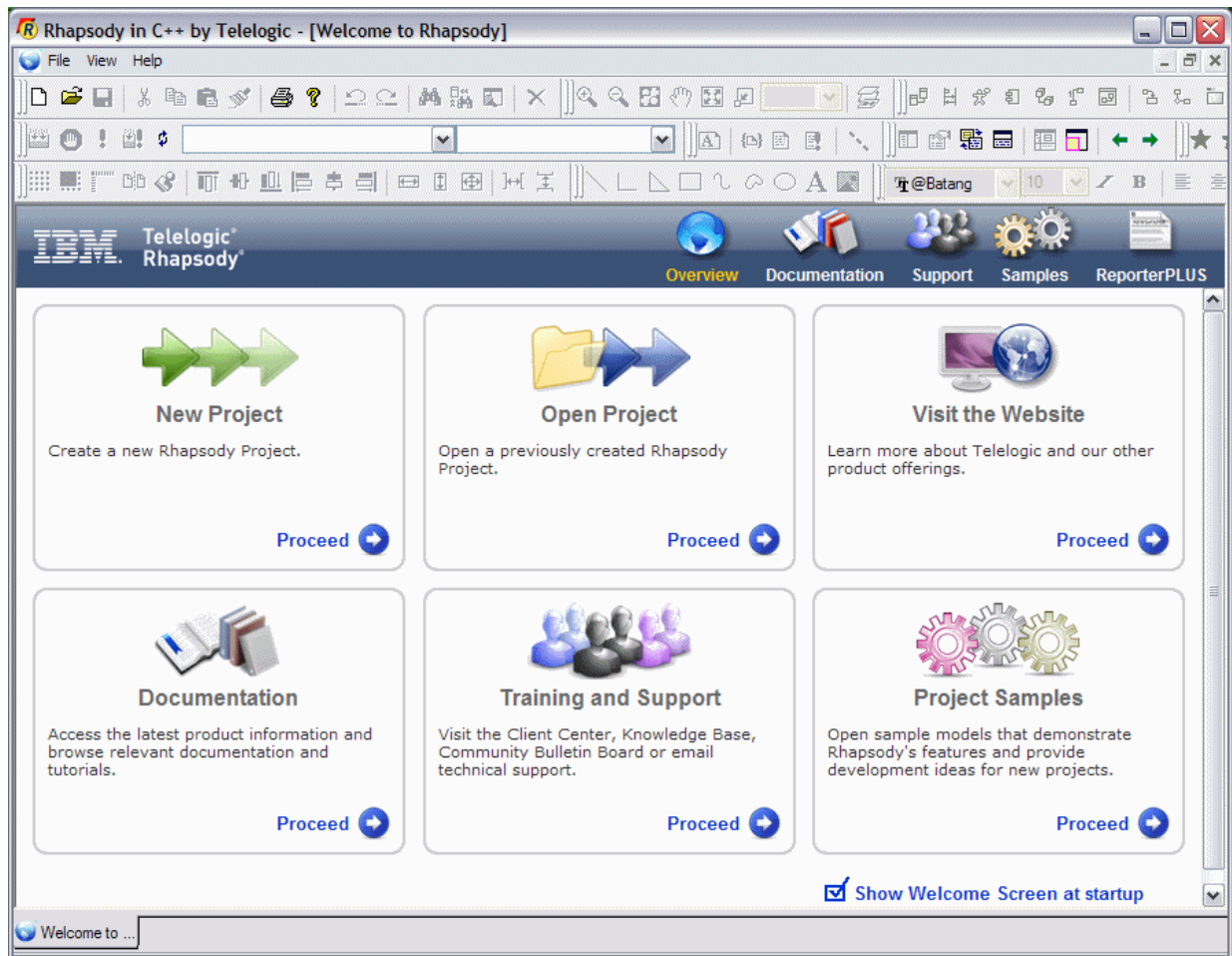


To open the Active Code View window, select **View > Active Code View**. Alternatively, click the **Active Code View** tool on the Windows toolbar.

The Active Code View window has two tabs, Specification and Implementation, which display the specification code and the implementation code, respectively.

Welcome Screen

The Welcome Screen, shown below, displays when you run Rhapsody for the first time. It provides links to help you get started quickly. The Welcome screen displays each time you start Rhapsody unless you clear the *Show Welcome Screen at startup* check box at the bottom of the screen.



You can view the Welcome Screen at any time from the **Help > Welcome Screen** option. If you have closed it and want to restore the display-on-startup setting, follow these steps:

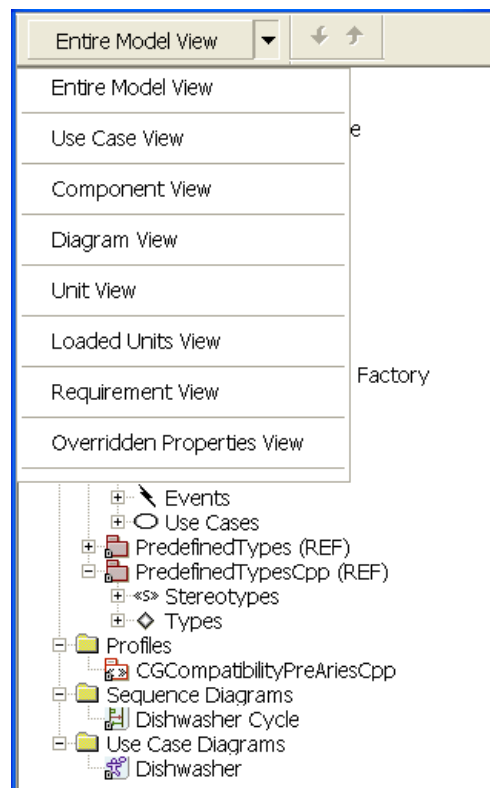
1. Select **Help > Welcome Screen** from the menu to display the screen.
2. Select the **Show Welcome Screen at startup** check box.

Rhapsody Project Tools

The Rhapsody project tools allow you to perform model design and development tasks using groups of toolbar buttons in the browser, drawing area, and output window. If an icon is disabled or not displayed, the operation represented by the button is not available for the currently displayed project items.

Browser Filter

The browser filter lets you display only the elements relevant to your current task. Click the down arrow at the top of the browser to display the menu of filter options, as shown in the following figure:



In the above figure, the filter is set to **Entire Model View** (default). See [Filtering the Browser](#) for detailed information about the other options.

You may also display the browser filter using the **View > Toolbars > Browser Filter** menu options.

Standard Project Tools

The **Standard** toolbar includes the following buttons:



New. Creates a new project. This button executes the same command as **File > New**.



Open. Opens an existing project. This button executes the same command as **File > Open**.



Save. Saves the current project. This button executes the same command as **File > Save**.



Cut. Cuts the selection to the clipboard. This button executes the same command as **Edit > Cut**.



Copy. Copies the selection to the clipboard. This button executes the same command as **Edit > Copy**.



Paste. Pastes the contents of the clipboard. This button executes the same command as **Edit > Paste**.



Format Painter. Used for copying formatting from one element to another element in the same diagram.



Print. Prints the active view. This button executes the same command as **File > Print**.



About. Opens the About Rhapsody dialog box, which displays the product release number, build number, serial number, contact information, and copyright information for your version of Rhapsody. You can also choose **Help > About Rhapsody** to open the dialog box. In addition, when you have the About Rhapsody dialog box open, you can click the **License** button to open the License Details dialog box. This dialog box shows you the host details for your machine (for example, the Host Name and Host ID) as well as your license information. You may need to know all this information when calling for technical support or upgrading your software.

Notice that you can resize the width of the License Details dialog box.



Undo. Undoes the last operation you performed in the model. This button executes the same command as **Edit > Undo**.



Redo. Reverses the undo command. This button executes the same command as **Edit > Redo**.



Search. Opens the Search dialog box, which enables you to search for a term in the model. This button executes the same command as **Edit > Search**.



References. Opens a list of elements that reference the selected element. This button executes the same command as right-clicking the selected element in the browser and selecting **References** from the pop-up menu.



Locate In Browser. Locates the selected element in the browser.

This button executes the same command as the **Locate** button in the Features dialog box and **Edit > Locate in Browser**.



Delete. Deletes the current selection from model. This button executes the same command as **Edit > Delete**.

Generating and Running Code Tools

The **Code** toolbar provides quick access to frequently used Code menu options. To display or hide this toolbar, select **View > Toolbars > Code**.

The **Code** toolbar includes the following buttons:



Make. Builds the active configuration. You must generate code before you can build the configuration.

This button executes the same command as **Code > Generate > Build run XXX.exe**.



Stop Make/Execution. Stops the make process or the execution while it is in progress. This button executes the same command as **Code > Stop**.



Run Executable. Runs the executable image. This button executes the same command as **Code > Run XXX.exe**.



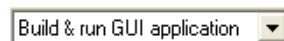
Generate/Make/Run (GMR). Generates code, builds the configuration, and runs the executable image. This button executes the same command as **Code > Generate/Make/Run**.



Disable/Enable Dynamic Model Code Associativity. Disables or enables dynamic model-code associativity. The button appears as two connected arrows when DMCA is active, and two disconnected arrows when DMCA is inactive. See [Deleting Redundant Code Files](#)



Current Component. Contains a list of all components in the project. To change the active component, select it from the drop-down list.



Current Configuration. Contains a list of all configurations in the active component. To change the active configuration, select it from the drop-down list.

Managing and Arranging Windows

The **Windows** toolbar provides quick access to Rhapsody windows, such as the browser and the Features dialog box. You can also access these commands from the View menu. To display or hide this toolbar, select **View > Toolbars > Windows**.

The **Windows** toolbar includes the following buttons:



Browser. Toggles between showing and hiding the browser.



Show/Hide Features. Toggles between showing and hiding the Features dialog box for the current element.



Show/Hide Active Code View. Toggles between showing and hiding the Active Code View window.



Show/Hide Output Window. Toggles between showing and hiding the Output window.



Toggle Arrange Options. Toggles between two standard desktop arrangements.

Alternatively, select **Window > Arrange Options**.

Use **Windows > Arrange Icons** to manipulate the arrangement of the desktop.



Toggles between showing and hiding the **Bird's Eye window**. You can also press **Alt + F5** to perform the same operation. For more information about this feature, see [Using the Bird's Eye \(Diagram Navigator\)](#).



Back displays the previously displayed window. This operation is also available using the **Window > Back** option.



Forward displays the window in the opposite direction from **Back**. This operation is also available using the **Window > Forward** option.

Note

The Back and Forward navigation is not available on Linux. This is for Windows systems only

Favorites Toolbar

The **Favorites** toolbar provides quick access to the Favorites browser. To display or hide this toolbar, select **View > Toolbars > Favorites**.

The **Favorites** toolbar includes the following buttons:



Show/Hide Favorites. Toggles between showing and hiding the Favorites browser.



Add to Favorites. Select a model element and then click this button to add what you selected to your Favorites browser.

For more information about the Favorites browser, see [Using the Favorites Browser](#).

Creating New Diagrams

The **Diagrams** toolbar provides quick access to the graphic editors, where diagrams are created and edited. The [Profiles](#) used to create the new projects control which diagrams are available for the projects. The available diagrams are represented as buttons on the toolbar across the top of the Rhapsody window. To hide or display this toolbar, select **View > Toolbars > Diagrams**.

To create a new diagram, follow these steps:

1. Select the **Tools** menu and select the type of diagram you want to create, or click the diagram icon at the top of the Rhapsody window.
2. The Open dialog box for the selected diagram displays. Highlight the portion of the project with which the diagram will be associated.
3. Click **New**.
4. In the New Diagram dialog box, enter the **Name** of the new diagram.
5. If you want to populate the new diagram automatically with existing model elements, click the **Populate Diagram** check box.
6. Click **OK** to generate the new diagram.

The following are all of the diagram types with their buttons, as displayed on the toolbar:



Object Model Diagram shows the logical views of the static structure of the classes and objects in an object-oriented software system and the relationships between them. This diagram is available for the majority of profiles.



Sequence Diagram shows the interactions between objects in the form of messages passed between the objects over time. This diagram is available for the majority of profiles.



Use Case Diagram shows the use cases of the system and the actors that interact with them. This diagram is available for the majority of profiles and also the FunctionalC profile.



Component Diagram shows the dependencies among software components, such as library or executable components. This diagram is available for the majority of profiles.



Deployment Diagram shows the run-time physical architecture of the system. This diagram is available for the majority of profiles.



Collaboration Diagram describes how different kinds of objects and associations are used to accomplish a particular task. This diagram is available for the majority of profiles.



Structure Diagram shows the architecture of the composite classes that define the model structure. This diagram is available for the majority of profiles.



Open Statechart defines the behaviors of individual classes in the system. This diagram is available for the majority of profiles and also the FunctionalC profile.



Open Activity Diagram shows the lifetime behavior of an object, or the procedure that is executed by an operation in terms of a process flow, rather than as a set of reactions to incoming events. This diagram is available for the majority of profiles.



Open Panel Diagram provides you with a convenient way to demonstrate a user device. During animation or Webify, you can use a panel diagram to activate and monitor your user application. This diagram is available for Rhapsody in C, Rhapsody in C++, and Rhapsody in Java projects.\



Build Diagram shows how the software is to be built. This diagram is primarily associated with the FunctionalC profile.



Call Graph Diagram shows the relationship of function calls as well as the relationship of data. This diagram is primarily associated with the FunctionalC profile.



File Diagram shows how files interact with one another (typically how the `#include` structure is created). This diagram is primarily associated with the FunctionalC profile.



Message Diagram shows how the files functionality may interact through messaging (synchronous function calls or asynchronous communication). This diagram is primarily associated with the FunctionalC profile.



Flow Chart (uses the same icon as the activity diagram) for a function or class the chart shows the operational flow and code generation. This diagram is primarily associated with the FunctionalC profile.

VBA Toolbar

The **VBA** toolbar provides quick access to VBA options, which can also be accessed by selecting **Tools > VBA**. To display or hide this toolbar, select **View > Toolbars > VBA**.

The **VBA** toolbar includes the following buttons:



VBA Editor. Opens the VBA editor.



Show Properties. Opens the VBA properties dialog box.



Show Macros Dialog. Opens the Macros dialog box so you can create VBA macros.



Design Mode. Enables you to run VBA in design mode.

Animation Toolbar

The **Animation** toolbar is enabled during an animation session. To display or hide this toolbar, select **View > Toolbars > Animation**. See [Animation](#) for detailed information about animation.

The **Animation** toolbar includes the following buttons:



Go Step. Advances the application a single step.



Go. Advances the application until it terminates or reaches a breakpoint.



Go Idle. Advances the application until it reaches an idle state where it is waiting for timeouts or events from GUI threads.



Go Event. Advances the application until the next event is dispatched or until the executable reaches an idle state.



Animation Break. Interrupts a model that is executing and suspends the clock.



Command Prompt. Opens the command bar where you can type animation and trace commands.



Quit Animation. Ends the animation session and terminates the executable.



Breakpoints. Opens the Breakpoints dialog box, where you can control breakpoints.



Event Generator. Opens the Events dialog box, where you can generate events using the Event Generator.

Drawing Toolbar

The **Drawing** toolbar provides access to tools used in creating and editing diagrams in the graphic editors. Each graphic editor has a unique drawing toolbar. To display or hide the **Drawing** toolbar for the current diagram, select **View > Toolbars > Drawing**.

To view the complete set of drawing tools available for all the different diagrams, select **View > Toolbars > All Drawing**.

For more information about modeling toolbars, see [Graphic Editors](#).

Common Drawing Tools

The **Common Drawing** toolbar enables you to add requirements, comments, and other annotations to any diagram. To display or hide this toolbar, select **View > Toolbars > Common Drawing**.

The **Common Drawing** toolbar contains the following buttons:



Note. Creates a documentation note. Click the button and use the mouse to draw the note in the diagram.

This is the type of note available with previous versions of Rhapsody. The note appears in the diagram, but not in the browser.



Constraint. Creates a constraint. This button executes the same command as **Edit > Add New > Constraint**.



Comment. Creates a comment. This button executes the same command as **Edit > Add New > Comment**.



Requirement. Creates a requirement. This button executes the same command as **Edit > Add New > Requirement**.



Anchor Constraint/Comment/Requirement to Item. Creates an anchor for a constraint, comment, or requirement.

See [Adding Annotations to Diagrams](#) for information about the tools available in this toolbar.

Zoom Toolbar

The **Zoom** toolbar contains the zoom tools you use with all the different diagram types. These tools are also available in the **View > Zoom/Pan** menu. To display or hide this toolbar, select **View > Toolbars > Zoom**.

The **Zoom** toolbar includes the following buttons:



Zoom In. Zooms in on a diagram. Click this button and click in a graphic editor window to increase the view by 25%.



Zoom Out. Zooms out on a diagram. Click this button and click in a graphic editor window to decrease the view by 25%.



Zoom to Selection. Select an element in a diagram and click this button to zoom into the selected section of the diagram.

Alternately, you can click **Zoom In** and hold down the left mouse button to draw a selection box around the part of the diagram you want to zoom in on.



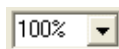
Pan. Moves the diagram in the drawing area so you can see portions of the diagram that do not fit in the current viewing area.



Scale to Fit. Resizes the active diagram to fit within the graphic editor window.



Undo Zoom. Reverses the last zoom action.



Scale Percentage. The options on this drop-down list resize the active diagram scale by the selected percentage.



Specification/Structured View. Displays either the specification or structured view of the active diagram.

See [Zooming](#) for detailed information.

Format Toolbar

The **Format** toolbar provides tools that affect the display of text in your diagrams, such as font, size, color, and so on. In addition, you can access these options by selecting **Edit > Format > Change > Font**.

To display or hide this toolbar, select **View > Toolbars > Format**.

The **Format** toolbar includes the following buttons:



Font Type. Specifies the font style (“type face”) used for text. Use the drop-down list to select a different font.



Text Size. Specifies the size used for text. Use the drop-down list to select a different size.



Italics. Changes the selected text to italic font.



Bold. Changes the selected text to boldface font.



Left Justify. Left-justifies the selected text.

This tool is enabled only in fields that support RTF, including the **Description** for elements and annotation elements (comment, requirement, and constraint).



Center Justify. Centers the selected text.

This tool is enabled only in fields that support RTF, including the **Description** for elements and annotation elements (comment, requirement, and constraint).

This tool is enabled only in fields that support RTF, including the **Description** for elements and annotation elements (comment, requirement, and constraint).



Right Justify. Right-justifies the selected text.

This tool is enabled only in fields that support RTF, including the **Description** for elements and annotation elements (comment, requirement, and constraint).



Bullet. Creates a bulleted list.

This tool is enabled when you are editing the **Description** for an element.



Font Color. Specifies the color to use for the text or label of the selected element. For example, if you select a state and use this tool to change the color to red, the name of the selected state is displayed in red.

This tool performs the same action as right-clicking and element and selecting **Format** from the pop-up menu.



Line Color. Specifies the color to use for the selected line element. For example, if you select a state and use this tool to change the line color to blue, the text box for the state will be displayed in blue.

This tool performs the same action as right-clicking and element and selecting **Format** from the pop-up menu.



Fill Color. Specifies the color to use as fill color for the selected element. For example, if you select a state and use this tool to change the color to yellow, the selected state will be filled with yellow.

This tool performs the same action as right-clicking and element and selecting **Format** from the pop-up menu.

Layout Toolbar

The **Layout** toolbar provides quick access to tools that help you with the layout of elements in your diagram, including a grid, page breaks, rulers, and so on. To display or hide this toolbar, select **View > Toolbars > Layout**.

The **Layout** toolbar includes the following buttons:



Grid. Displays or hides the grid in the drawing area. This is equivalent to selecting **Layout > Grid > Grid**.



Snap to Grid. Automatically aligns new elements to the closest grid points. This is equivalent to selecting **Layout > Grid > Snap to Grid**.



Rulers. Displays or hides the rulers in the drawing area. This is equivalent to selecting **Layout > Show Rulers**.



Page Breaks. Displays or hides the page breaks in your diagram. This is equivalent to selecting **Layout > View Page Breaks**.

Page boundaries are denoted by dashed lines.



Stamp Mode. Turns repetitive drawing mode on or off. This is equivalent to selecting **Tools > Repetitive Drawing Mode**.



Align Top. Aligns the selected elements to the top of the element with the gray selection handles. This is equivalent to selecting **Layout > Align > Top**.



Align Middle. Aligns the selected elements to the middle (along the horizontal) of the element with the gray selection handles. This is equivalent to selecting **Layout > Align > Middle**.



Align Bottom. Aligns the selected elements to the bottom of the element with the gray selection handles. This is equivalent to selecting **Layout > Align > Bottom**.



Align Left. Aligns the selected elements to the left side of the element with the gray selection handles. This is equivalent to selecting **Layout > Align > Left**.



Align Center. Aligns the elements so they are aligned to the center, vertical line of the element with the gray selection handles. This is equivalent to selecting **Layout > Align > Center**.



Align Right. Aligns the selected elements to the right side of the element with the gray selection handles. This is equivalent to selecting **Layout > Align > Right**.



Same width. Resizes all the selected elements so they are the same width as the element with the gray selection handles. This is equivalent to selecting **Layout > Make Same Size > Same Width**.



Same height. Resizes all the selected elements so they are the same height as the element with the gray selection boxes. This is equivalent to selecting **Layout > Make Same Size > Same Height**.



Same size. Resizes all the selected elements so they are the same size as the element with the gray selection boxes. This is equivalent to selecting **Layout > Make Same Size > Same Size**.



Space across. Spaces the selected elements so they are equidistant (across) from the element with the gray selection handles. This is equivalent to selecting **Layout > Space Evenly > Space Across**.



Space down. Spaces the selected elements so they are equidistant (down) from the element with the gray selection handles. This is equivalent to selecting **Layout > Space Evenly > Space Down**.

Free Shapes Toolbar

The **Free Shapes** toolbar enables you to draw elements freehand in a diagram. To display or hide this toolbar, select **View > Toolbars > Free Shapes**. See [Graphic Editors](#) for detailed information on these tools.

The **Free Shapes** toolbar includes the following buttons:



Line. Draws a straight line between the selected endpoints.



Polyline. Draws a polyline using multiple points.



Polygon. Draws a polygon.



Rectangle. Draws a rectangle.



Polycurve. Draws a curve.



Closed Curve. Draws a closed, curved shape within the bounds of the specified shape.



Ellipse. Draws a circle or ellipse.



Text. Draws free text.



Image. Enables you to import an image into the diagram.

Using the Features Dialog Box

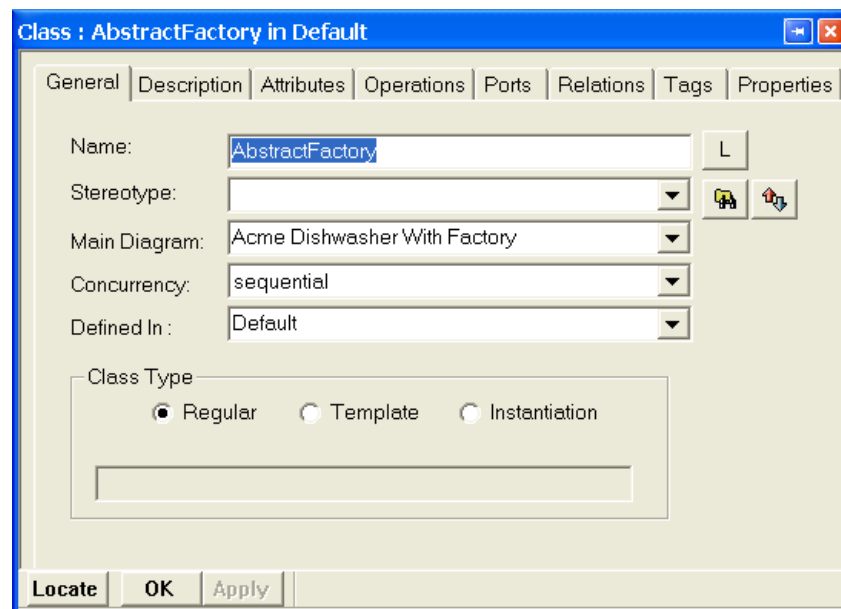
The Features dialog box enables you to edit the features of each element in the Rhapsody model.

Invoking the Dialog Box

To open the Features dialog box, use any of the following methods:

- ◆ Double-click an element in the browser (except diagrams).
- ◆ Right-click an element and select **Features** from the pop-up menu.
- ◆ Select an element in the browser and press **Alt + Enter**.
- ◆ Select an element and select **View > Features**.

The Features dialog box lists different fields depending on the element type. For example, the following figure shows the features of a class.



Applying Changes

When you have made changes that need to be applied to the project, an asterisk (*) is displayed in the title bar of the Features dialog box. Use one of these methods to save the changes.

- ◆ Press **Ctrl + Enter**.
- ◆ Click the **Apply** button on the Features dialog box.
- ◆ Change focus to another window or tab.
- ◆ Initiate an external activity, such as generating code, saving the project, or generating a report.

To apply changes and close the Features dialog box, click **OK**.



Canceling Changes

To cancel changes made to the Features dialog box, press the **Esc** key. Alternatively, you can close the dialog box without applying changes.

Note that changes *cannot* be canceled once they have been applied to the model.

Pinning the Dialog Box

The Features dialog box can be “pinned” to a specific element to keep the information for that element displayed while examining other elements’ features. To pin a Features dialog box to an element, follow these steps:

1. Highlight an element in the browser or a diagram.
2. Display the Features dialog box for that element.
3. Click the  icon in the upper right corner of the dialog box. Note that the icon changes to this  icon to indicate that this element’s dialog box will now remain displayed.

In pinned mode, the features displayed in the pinned dialog box remain displayed and accessible from all of the dialog box tabs even when a different element is selected. Therefore, you may display and pin two or three Features dialog boxes to compare the information for the elements.

When you no longer need to see the features of the element displayed, you may click the pin icon again to disconnect it from the element or simply close the dialog box.

Hiding the Buttons on the Features Dialog Box

At the bottom of the Features dialog box, there are three buttons: **Locate**, **OK**, and **Apply**.

To remove these buttons from view, right-click the title bar for the Features dialog box and uncheck the **Features Toolbar** option.

Docking the Features Dialog Box

By default, the Features dialog box is a floating window. It can be positioned anywhere on the screen, or docked to the Rhapsody work area.

To dock the Features dialog box in the Rhapsody window, do one of the following:

- ◆ Double-click the title bar of the Features dialog box. The dialog box will jump to the location where it was last docked. To dock the dialog box in a different location, click the title bar and drag the dialog box to the desired location.
- ◆ Right-click the title bar and select **Enable Docking by Drag** to display a check mark and drag the dialog box to the desired location.

To undock the Features dialog box, do one of the following:

- ◆ Double-click the title bar of the Features dialog box.
- ◆ Click the title bar and hold down **Ctrl** while dragging to a new location.
- ◆ Right-click the title bar and select **Enable Docking by Drag** to remove the check mark and drag the window to the desired location. The dialog box is no longer docked with the main window.

Opening Multiple Instances of the Dialog Box

You can open multiple Features dialog boxes in the Rhapsody workspace. Using this functionality, you can easily compare the features of two different elements and quickly copy text from one Features dialog box to another.

To open more than one Features dialog box, right-click an element and select **Features in New Window**.

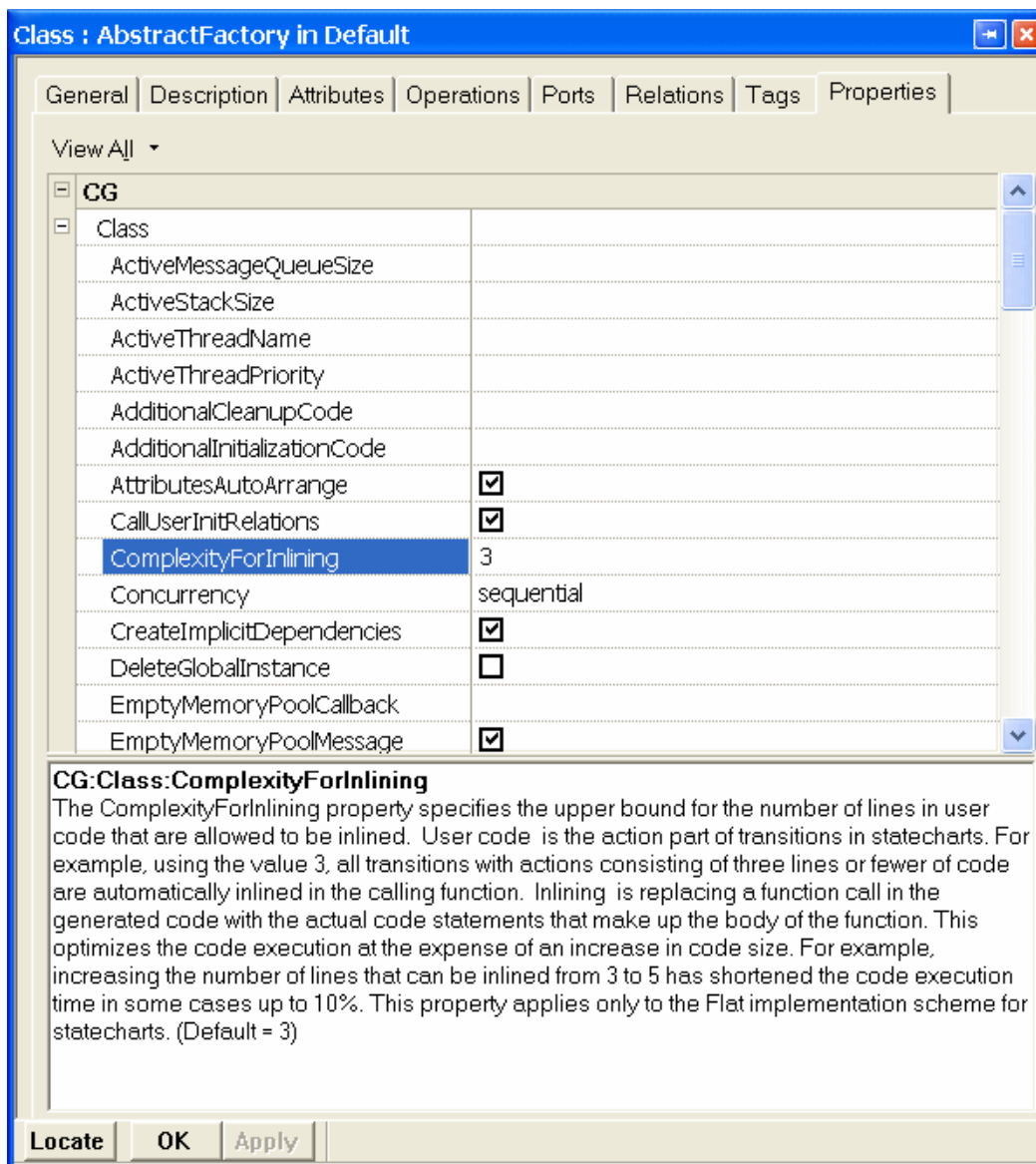
When opened as a new window, the Features dialog box remains focused on the same element, even when you change the browser or graphic editor selection. Any changes made to that element from another view (such as the browser or a diagram editor) are automatically updated in the Features dialog box. This enables you to keep track of a particular element's features while working with other parts of the model.

When you have an open Features dialog box that is focused on a particular element, you can locate that element in the browser by clicking the **Locate** button at the bottom of the dialog box.

Alternatively, you can locate the item by selecting the **Locate in Browser** tool from the standard toolbar.

Properties Tab

The **Properties** tab of the Features dialog box displays the properties for the currently selected item (selected in the browser or in a diagram) or all of the properties for a model (select **File > Project Properties**).



Properties Definitions Display

The **Properties** tab of the Features dialog box includes a definition display, below the list of properties, as shown previously. This area displays the selected property's definition. Each time a new property is selected, its definition displays below.

Definitions are displayed for all three of the property levels: subject, metaclass, and the individual property. (Refer to the *Rhapsody Properties Reference Manual* for more information about this hierarchical structure.) To examine the complete list of property definitions, refer to the *Rhapsody Property Definitions* PDF file available from the *List of Books*. This list can be searched along with the other PDF versions of the Rhapsody documentation to locate specific property definitions and the procedures that use them.

Searching for Properties

The **Properties** tab contains a drop-down menu (shown below) to filter the properties displayed in the dialog box.



Of course, the **All** option displays all of the available properties for your selection. You may display only the **Overridden** properties for the model or only the properties you overrode locally with the **Locally Overridden** option. The **Common** properties are those most often changed for the selected type of item.

The **Filter** menu option allows you to search for specific properties by entering a filter string. Rhapsody displays only the properties that contain the text you entered. When you are filtering the properties and select the **Match property description** check box, Rhapsody searches the property definitions for the string you entered, as well as the property name.

Note

The Filter mechanism is not case-sensitive and does not allow the use of wildcards or expressions.

If you enter more than one word as the Filter, Rhapsody performs an “or” search and displays all of the properties that contain any one of the words entered. To limit the search to only the definitions containing the entire phrase, enclose the words in the search string within quotation marks.

As long as the Features dialog box (or standalone properties dialog box) remains open, the selection you made from the menu (filter text and check box setting) is retained. When the Features dialog box is closed, these are reset.

Displaying Feature Tabs as Stand-alone Dialog Boxes

The Features dialog box contains a number of tabs that are common to almost all types of elements:

- ◆ **General**
- ◆ **Descriptions**
- ◆ **Relations**
- ◆ **Tags**
- ◆ **Properties**

For each of these tabs, you also have the option of displaying the information in a dockable stand-alone dialog box. To do so, follow these steps:

1. Select an element in the Rhapsody browser or in a diagram.
2. Select the relevant menu item in the View menu, for example, **View > Description**.

Once the stand-alone dialog box is displayed, you can dock the dialog box. To do so, follows these steps:

1. Right-click the title bar for the title bar and select **Enable Docking by Drag**.
Note that a check mark appears to the left of the command on the pop-up menu.
2. Drag the dialog box to one of the borders or other docking locations in the Rhapsody window.
Notice that upon reaching one of these locations, the outline of the dialog box changes to reflect the area the dialog box occupies when docked.

Note

When one of these dialog boxes is docked, it continues to display the information in the same manner as it does when it is “pinned,” as described in [Pinning the Dialog Box](#).

To undock a dialog box, do one of the following:

- ◆ To undock without disabling the docking capability, drag the dialog box to one of the non-docking locations in the Rhapsody window.
- ◆ To disable docking and undock, follow these steps:

- a. Right-click the title bar for the dialog box and select **Enable Docking by Drag**. Note that the check mark to the left of the command no longer appears.
- b. Drag the dialog box anywhere in the Rhapsody window.

Hiding Tabs

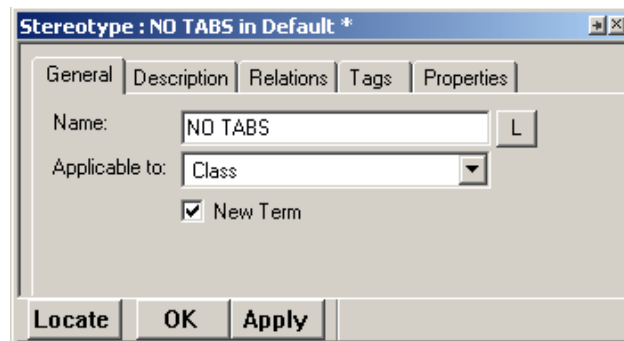
If you normally do not use one of the tabs on the Features dialog box for a particular metaclass, you can hide it. To do this, you have to create a **New Term** stereotype that sets the `HideTabsInFeaturesDialog` property to hide one or more tabs on the Features dialog box. Then you would apply this stereotype to a model element of that metaclass.

Note

This feature is used exclusively for elements with the **New Term** stereotype.

To hide one or more tabs on the Features dialog box for a particular metaclass, follow these steps:

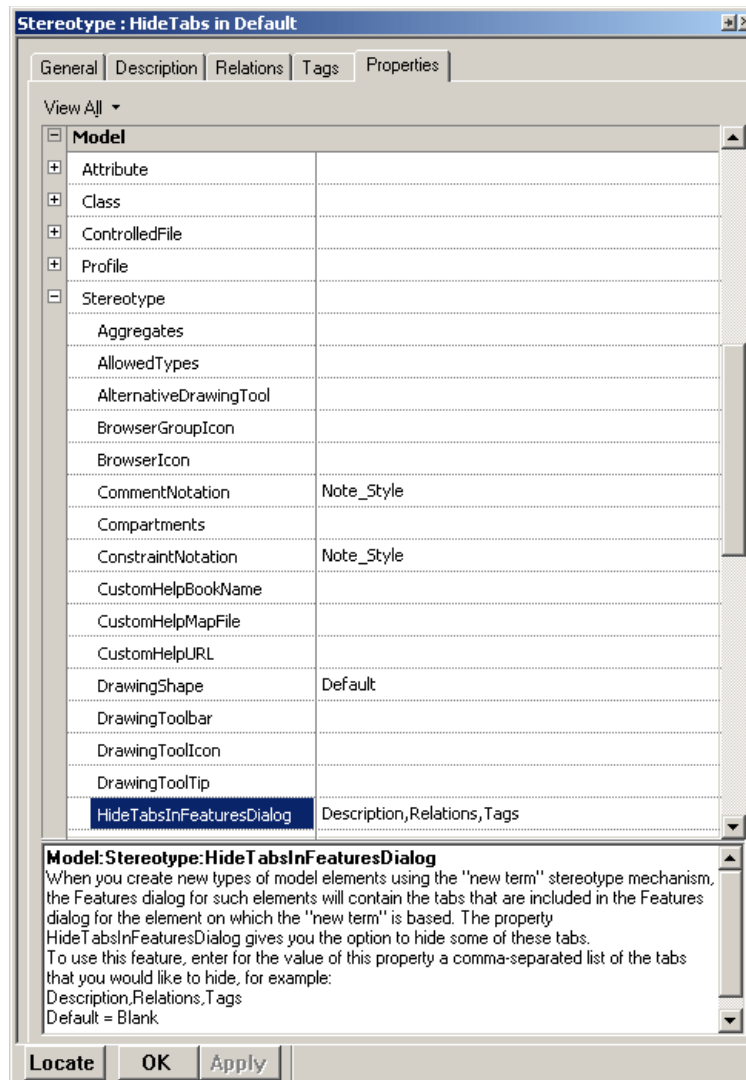
1. Display your model in Rhapsody.
2. In the Rhapsody browser, create a stereotype as a **New Term**.
 - a. See [Defining Stereotypes](#) to learn how to create a stereotype.
 - b. Be sure to select one metaclass in the **Applicable To** box and select the **New Term** check box on the **General** tab of the Features dialog box for your stereotype, as shown in the following figure:



- c. On the **Properties** tab for the stereotype, locate the `Model::Stereotype::HideTabsInFeaturesDialog` property.

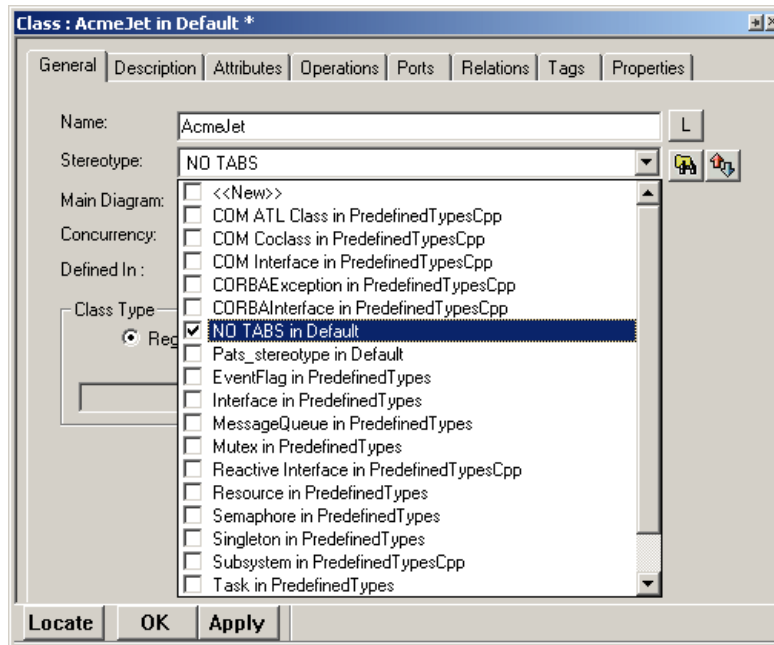
- d. Click the box to the right of the property name and type the names of the tabs, separated by a comma, that you want to hide; for example, `Description,Relations,Tags`, as shown in the following figure:

Note: You cannot hide the **General** tab.



- e. Click **OK** to close the Features dialog box for the stereotype.

- Apply the stereotype to a model element of the metaclass. For example, if you selected the **Class** metaclass (in the **Applicable To** box on the **General** tab of the Features dialog box for the stereotype), then you can apply this stereotype (select it from the **Stereotype** drop-down menu on the **General** tab of the Features dialog box for the class) to any classes you currently have in your model or any that you create, as shown in the following figure:



Note: When you define a stereotype as a **New Term**, it is given its own category in the Rhapsody browser, and any elements to which this stereotype is applied are displayed under this category.

If you want to display a previously hidden tab, delete the name of that tab from the list you entered in the `Model::Stereotype::HideTabsInFeaturesDialog` property.

Hyperlinks

Rhapsody supports both *internal hyperlinks*, which point to Rhapsody model elements, and *external hyperlinks*, which point to a URL or file.

In addition, you can:

- ◆ Use the DiffMerge tool to compare models to locate differences in diagrams and to merge models that contain hyperlinks.
 - Note:** You can edit a description that uses hyperlinks or RTF format in the DiffMerge tool if it is from the left or right side of the comparison, but you cannot edit a description from a merge.
- ◆ Export hyperlinks using the Rhapsody COM API.
 - Note:** You cannot create or modify hyperlinks using the COM API.
- ◆ Report on hyperlinks using ReporterPLUS.
- ◆ Find references to hyperlinks using the Show References feature.

Creating Hyperlinks

You can create hyperlinks inside the **Description** of an element or the browser, as described in the following sections.

Note

Hyperlinks created in the **Description** are *not* model elements, and can neither be viewed in the browser nor accessed by the COM API.

Using the Description Tab


A typical use for the **Description** tab of the Features dialog box is to enter a description for whatever Rhapsody element you currently have open. For example, if you have the Features dialog box open for a class, you can enter a detailed description for the class on the **Description** tab. You can do the same on the **Description** tab for an attribute, an event, a package, and so forth.

In addition, you can create a hyperlink within the description of an element. To do so, follow these steps:

1. Open the Features dialog box for the element.
2. Select the **Description** tab.
3. Right-click in the open field to display the pop-up menu. If you want to replace pre-existing text with a hyperlink, select the text before right-clicking.

4. Select **Hyperlink** from the pop-up menu. The Hyperlink dialog box opens. Use this dialog box to specify the hyperlink text and target.

The **Text to display** field specifies the text for the hyperlink. The possible values are as follows:

- ◆ **Free text**—Display the specified text as the hyperlink text.
- ◆ **Target name**—Display the full path of the target as the hyperlink text.
- ◆ **Target label**—Display the label of the target as the hyperlink text. This option is available only for internal hyperlinks that have labels.
- ◆ **Tag value**—Displays the value for the tag. Note that this value is available only when you select a tag as the hyperlink target. See [Using Tag Values in Hyperlinks](#) for an example that uses this field.
- ◆ The **Link target** field specifies the target file, Web page, or model element. You can specify the target by typing the target in the text field, using the drop-down list to select the model element in the model, or clicking the Ellipses button  to open a new window so you can navigate to the target file.

Note: You can include a relative path in the hyperlink target. If you use a relative path, the base directory is the one where the `<Project name>.rpy` file is located.

5. Click **OK** to create the hyperlink and close the dialog box. The hyperlink is displayed in the Description area as blue, underlined text. Note that this type of hyperlink is *not* displayed in the browser.

Using the Browser

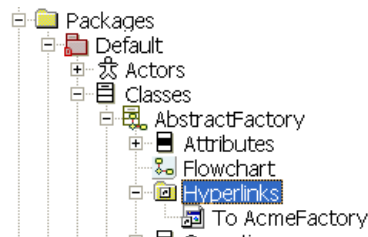
To create a hyperlink in the browser, follow these steps:

1. Right-click the element to which to add the hyperlink and select **Add New > Hyperlink**. Alternatively, right-click the **Hyperlinks** category and select **Add New Hyperlink**.

Rhapsody creates a new hyperlink in the browser.

2. Open the Features dialog box for the new hyperlink.
3. Specify the hyperlink display text in the **Text to display** field.
4. Specify the hyperlink target in the **Link target** field by typing the path, using the drop-down list, or using the navigation window.
5. If desired, specify a stereotype or description.
6. Click **OK** to create the hyperlink and close the dialog box.

The hyperlink is added to the `Hyperlinks` category under the owner element, as shown in the following figure.



To improve readability, there are different icons for the different targets, such as the following:

- ◆ Word files
- ◆ Classes
- ◆ URLs

You can drag-and-drop hyperlinks from the **Hyperlinks** category of one element to that of another. Similarly, you can copy hyperlinks from the `Hyperlinks` category of one element to that of another by dragging-and-dropping and pressing **Ctrl**, or using the Copy and Paste shortcuts.

Following a Hyperlink

To follow a hyperlink, double left-click it. The corresponding file, dialog box, or URL is displayed.

Alternatively, you can use the **Open Hyperlink** option in the pop-up menu.

Editing a Hyperlink

You can edit a hyperlink using the Features dialog box or from within the Description area, depending on the type of hyperlink.

Note

You cannot rename a hyperlink directly from the browser—you must invoke the Features dialog box.

Use the Features dialog box to change the features of the hyperlink, including its text display and target.

A hyperlink has the following features:

- ◆ **Name**—Specifies the name of the element. The default name is `hyperlink_n`, where *n* is an incremental integer starting with 0.
- ◆ **Text to display**—Specifies the text for the hyperlink. The possible values are **Free text**, **Target name**, **Target label**, and **Tag value**. See [Using the Description Tab](#) for more information on these options.
- ◆ **Link target**—Specifies the target file, Web page, or model element.
- ◆ **Description**—Describes the hyperlink.

Editing the Hyperlink in the Description Area

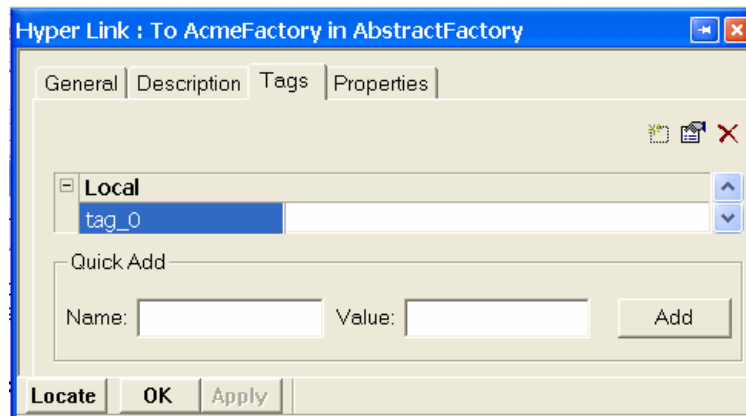
To edit a hyperlink in the Description area, follow these steps:

1. Open the Features dialog box for the element.
2. Select the **Description** tab.
3. Right-click the hyperlink in the text and select **Edit Hyperlink**.
4. In the Hyperlink dialog box (see [Using the Description Tab](#)), edit the link as desired.
5. Click **OK** to apply your changes and close the dialog box.

Using Tag Values in Hyperlinks

You can display the value of a tag in a hyperlink. To add a tag value to a hyperlink, follow these steps:


1. Wherever you want to create the hyperlink, right-click and select **Features**.
2. In the Features dialog box, select the **Tags** tab. The **Quick Add**, shown below, allows you to enter the name of the hyperlink and its value. If the tag does not have a value, the value «empty» is displayed.

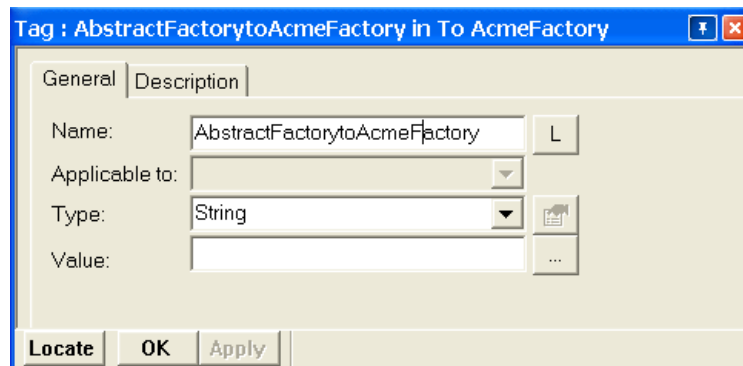


3. Click **OK** to place the tag value in the hyperlink and close the dialog box.

Changing the Tag Value

To change the value of the tag, follow these steps:

1. Click the tag value hyperlink or click the New icon  in the Tag dialog box to invoke this Features dialog box.



2. Replace the existing value with the new value.
3. Click **OK** to apply your changes and close the dialog box.

Limitations

Note the following limitations:

- ◆ You can select tags as hyperlink targets, which are available in the Rhapsody browser. For example, if you have the tag `color` in a profile that is applicable to *all* classes, you cannot see the tag `color` under a given class instance in the browser. The Rhapsody browser shows only local or overridden tags; however, these tags are shown in the **Tags** tab of the Features dialog box for the class.
- ◆ If you override a tag value in a package, the tag is considered to be local because it is tied to that specific element. If you have a hyperlink to the local tag and subsequently delete the tag, the reference will be unresolved.

Deleting a Hyperlink

Delete a hyperlink using one of the following methods:

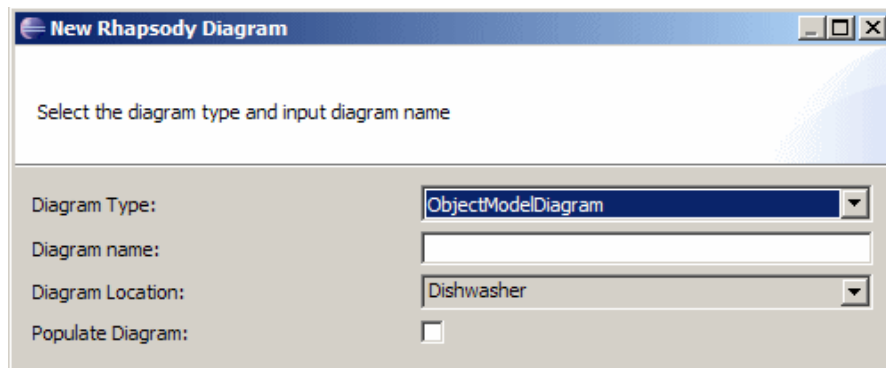
- ◆ In the text area of the **Description** tab, right-click the link and select **Remove Hyperlink** from the pop-up menu, or use the backspace key or **Delete** icon.
- ◆ In the browser, highlight the hyperlink and select **Delete from Model** from the pop-up menu or click the **Delete** icon.

New Rhapsody Diagram Dialog Box

This procedure is relevant for Eclipse users.

To create a Rhapsody diagram, follow these steps:

1. On the New Rhapsody Diagram dialog box, select a diagram type from the drop-down list, as shown in the following figure:



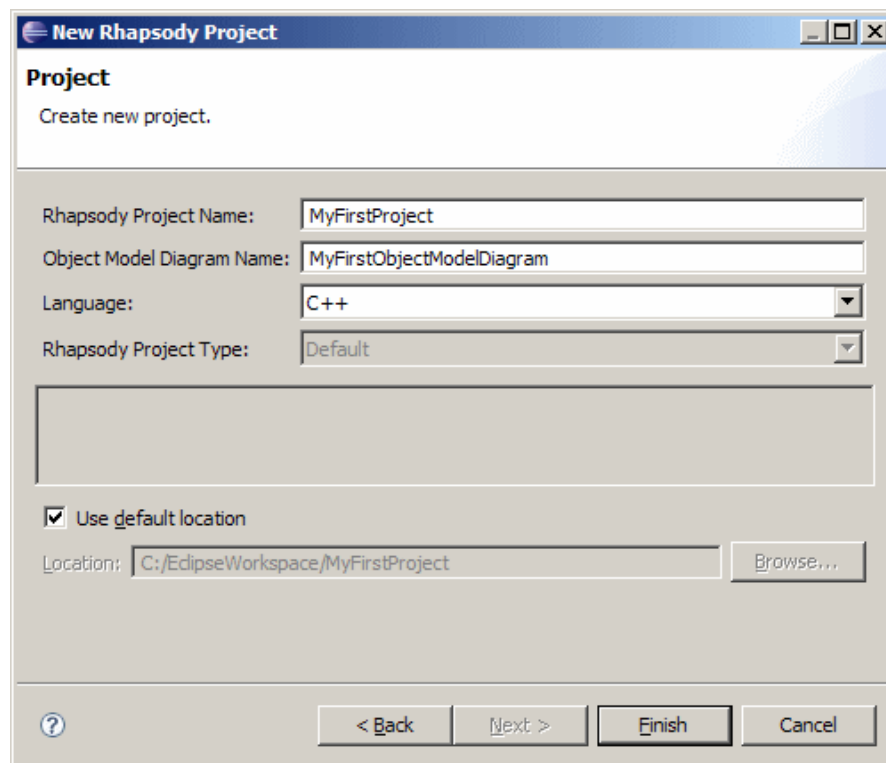
2. Enter a name for the diagram.
3. If available, select a location for the diagram from the drop-down list.
4. If available, if you want to populate the new diagram automatically with existing model elements. Click the **Populate Diagram** check box.
5. Click **Finish**.

New Rhapsody Project Dialog Box

This procedure is relevant for Eclipse users.

To create a Rhapsody project, follow these steps:

1. On the New Rhapsody Project dialog box, enter a name for your Rhapsody project, as shown in the following figure:



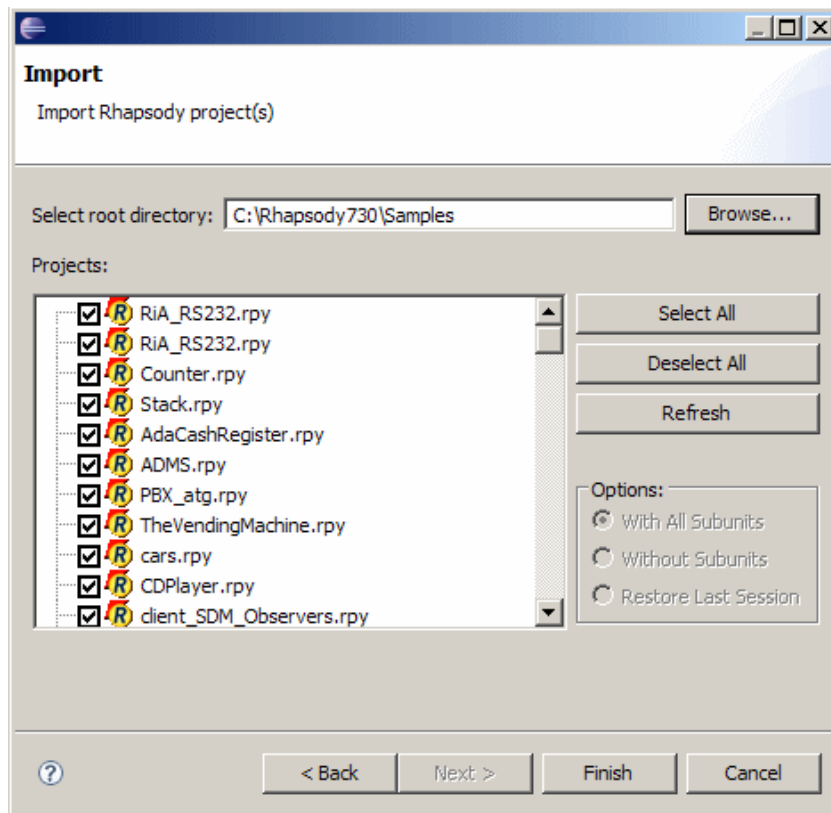
2. Optionally, enter a name for your first object model diagram.
3. Select the language for your project from the drop-down list.
4. If available, select the Rhapsody project type from the drop-down list.
5. If you want to designate a location for your project other than your default location, clear the Use default location check box and browse to your preferred location.
6. Click **Finish**.
7. If the directory for your project is not already created, click **Yes** when you are asked if you want to create it.

Import Dialog Box

This procedure is relevant for Eclipse users.

To import a project, follow these steps:

1. On the Import dialog box, browse to your select root directory, as shown in the following figure:



2. Select the project(s) you want to import.
 - ◆ Click **Select All** to select all the projects listed.
 - ◆ Click **Deselect All** to clear the check boxes for all the selected projects.
 - ◆ Click **Refresh** to refresh your list.
3. If available, select your options selection:
 - ◆ **With All Subunits.** Select this option button to load all units in the project, ignoring workspace information. For information on workspaces, see [Using Workspaces](#).

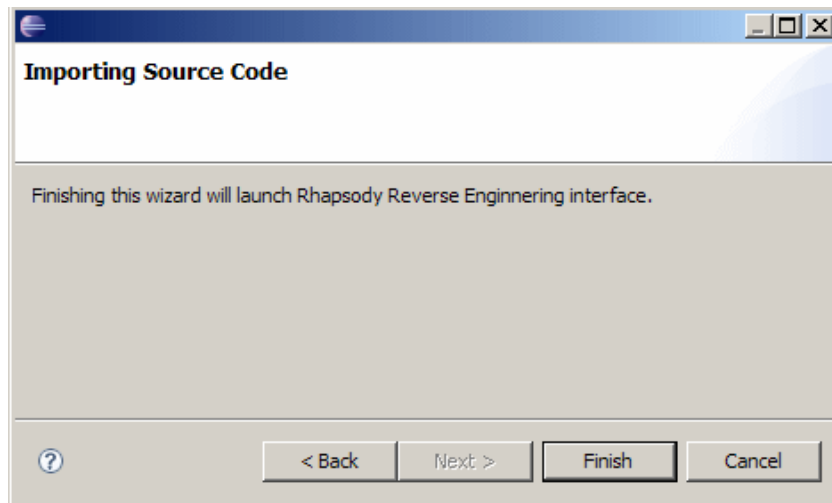
- ◆ **Without Subunits.** Select this option button to prevent loading any project units. All project units will be loaded as stubs.
 - ◆ **Restore Last Session.** Select this option button if you would like to load only those units that were open during your last Rhapsody session.
4. Click **Finish**.

Importing Source Code Dialog Box

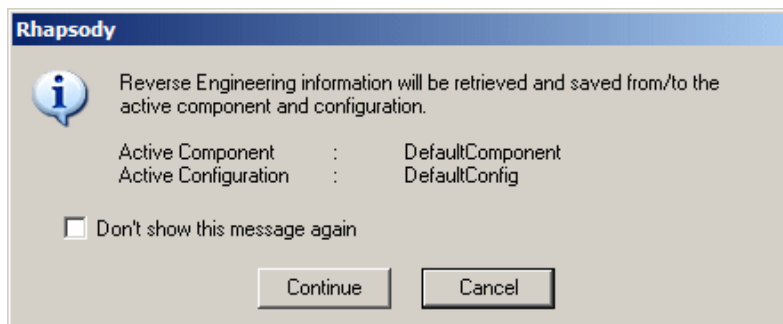
This procedure is relevant for Eclipse users.

To import source code, follow these steps:

1. On the Importing Source Code dialog box, as shown in the following figure, click the **Finish** button:



2. On the message box that appears, as shown on the following figure, click **Continue** to open the Reverse Engineering dialog box.



3. See [Using the Reverse Engineering Tool](#).

UML Design Essentials

This section provides an overview of how to use UML in Rhapsody to design your models.

Unified Modeling Language

The Unified Modeling Language (UML) is a third-generation modeling language for describing complex systems. The Object Management Group[®] (OMG[®]) adopted the UML as the industry standard for describing object-oriented systems in the fall of 1997. For more information on the OMG, see their Web site at <http://www.omg.org>.

UML defines a set of diagrams by which you can specify the objects, messages, relationships, and constraints in your system. Each diagram emphasizes a different aspect or view of the system elements. For example, a UML sequence diagram focuses on the message flow between objects during a particular scenario, whereas an object model diagram defines classes, their operations, relations, and other elements.

UML Diagrams

The UML specification includes the following diagrams:

- ◆ **Use case diagram**—Shows typical interactions between the system being designed and external users or actors. Rhapsody can generate code for actors in use case diagrams to be used for testing a model.
- ◆ **Object model diagram**—Shows the static structure of a system: the objects in the system and their associations and operations, and the relationships between classes and any constraints on those relationships.
- ◆ **Sequence diagram**—Shows the message flow of objects over time for a particular scenario.
- ◆ **Collaboration diagram**—Provides the same information as a sequence diagram but emphasizes structure, whereas a sequence diagram emphasizes time.
- ◆ **Statechart**—Defines all the states that an object can occupy and the messages or events that cause the object's transition from one state to another.
- ◆ **Activity diagram**—Specifies a workflow or process for classes, use cases, and operations. Activity diagrams provide similar information to statecharts, but are better for linear step-

by-step processes, whereas statecharts are better suited for non-linear or event-driven processes.

- ◆ **Component diagram**—Describes the organization of the software units and the dependencies among these units.
- ◆ **Deployment diagram**—Depicts the nodes in the final system architecture and the connections between them. Nodes include processors that execute software components, and the devices that those components control.
- ◆ **Structure diagram**—Models the structure of a composite class; any class or object that has an OMD can have a structure diagram. Object model diagrams focus more on the specification of classes, whereas structure diagrams focus on the objects used in the model.

In addition, a Flow Chart is available in the Rhapsody product. You can use a flow chart to describe a function or class operation and for code generation.

UML Views

Rhapsody enables you to draw UML diagrams that provide different views of your system. By editing the UML diagrams in Rhapsody to create increasingly complex views, you can add layers of perspective, detail, and specificity to your model until you have a complete solution.

Structural Views

Structural views show model elements and their relationships to each other. Model elements include classes, use cases, components, and actors; their relationships include dependencies, inheritances, associations, aggregation, and composition.

The following UML diagrams provide structural views:

- ◆ Use case diagram
- ◆ Object model diagram
- ◆ Structure diagrams
- ◆ Component diagram
- ◆ Deployment diagram

Dynamic Behavior Views

Dynamic behavior views describe the system behavior. This includes state behavior, such as the different states a class occupies, state transitions, forks and joins, and actions within a state; and interactions, such as the collaborations occurring between classes during a particular scenario.

The following UML diagrams provide dynamic behavior views of the model:

- ◆ Statechart
- ◆ Activity diagram
- ◆ Sequence diagram
- ◆ Collaboration diagram

Model Management Views

Model management views show the hierarchical organization of the model. Object model diagrams provide a model management view.

Diagrams in Rhapsody

Rhapsody includes a graphic editor for each of the UML diagrams, enabling you to create detailed views of your model. The graphic editors not only capture the design of your system, but also generate implementation code.

Note

Rhapsody diagrams have varying levels of code generation ability. Model elements and implementation code can also be created from the browser.

Because Rhapsody maintains a tight model-code associativity, you can easily generate updated code when you make changes to the model. You can also edit code directly and bring those changes into the model via the roundtrip feature. See [Basic Code Generation Concepts](#) for more information about model-code associativity.

Partially Constructive Diagrams

Partially constructive diagrams generate code for some, but not all of the elements in the diagram. Partially constructive diagrams include the following:

- ◆ Use case diagrams
- ◆ Sequence diagrams
- ◆ Collaboration diagrams

Fully Constructive Diagrams

Fully constructive diagrams generate code for every element in the diagram. Fully constructive diagrams include the following:

- ◆ Object model diagrams
- ◆ Component diagrams
- ◆ Statecharts
- ◆ Activity diagrams

Specifying a Model with Rhapsody

To create a working model, you must create at a minimum an object model diagram. An object model diagram generates the code necessary for a minimally functioning model.

A properly designed implementation, however, includes at a minimum object model diagrams, statecharts or activity diagrams, and component diagrams. Object model diagrams and statecharts can be considered to be design diagrams, because they are most often used in the design phase of a project. Other diagrams are more helpful in other phases. For example, use case and sequence diagrams are useful in the requirements analysis phase, where use case diagrams document structural requirements and sequence diagrams document behavioral requirements.

Development Methodology

A development methodology is a combination of a *process*, a tool, and a modeling language. Rhapsody is a UML-compliant modeling tool that is process-neutral and supports the most common phases of any good development methodology. However, Rhapsody is particularly well-suited to an iterative process in which you build a number of model prototypes, test, debug, reanalyze, and then rebuild the model any number of times, all within a single development environment.

The ROPES™ process is an example of an iterative process that illustrates the use of Rhapsody and the UML across all typical process phases and activities. The following sections provide a general overview of the phases involved in the ROPES process—including the subtasks involved in general analysis, design, implementation, and test phases—and the Rhapsody tools appropriate for each phase. The modeling products' Web site (<http://modeling.telelogic.com>) contains detailed information on ROPES.

Analysis

In the analysis phase, you define a problem, its possible solutions, and their characteristics.

Requirements Analysis

Begin with the requirements analysis to identify the system requirements. What are the primary system functions or system usages? Use case diagrams can capture these along with the external actors that interact with the system.

Describe the expected behavior of the system as a whole by creating a series of “black-box” sequence diagrams. In these, you will define the sequence of messages between external actors and the system as a whole. You can create a number of sequence diagrams for each use case, where each sequence diagram represents one scenario that could occur while carrying out that use case. You can also use collaboration diagrams to specify the expected behavior of the system.

Use the black-box sequence diagrams as the basis for creating statecharts, which realize all possible scenarios. Statecharts specify the behavior of each object, or object implementation, as opposed to sequence diagrams, which concentrate on requirements-based scenarios. Sequence diagrams also serve as the primary test data in the testing phase; in later stages, use them to test whether your system as a whole responds properly to the external messages that come into it. You can also use activity diagrams to realize all possible scenarios.

Object Analysis

While you are capturing system requirements, you should also define the entities and structural relationships that will exist in the application you are creating and its domain or environment. This should result in a structural (static) model of the system—a logical object model of the system.

Determine the subsystems of your system. What are their responsibilities and relationships? These subsystems become the basis of the packages, or collections of classes, within your system.

Determine the key objects or classes in these subsystems and define their responsibilities, descriptions, and their relations to other classes. Use object model diagrams to create these classes and their relations. Using sequence diagrams and statecharts, define the behavior and interactions of these essential objects.

You can also use the code generation and animation tools to execute and debug these higher-level analysis models.

Design

In the analysis phase, you came up with several possible solutions to your problem. In the design phase, you choose one of those solutions and define how it will be implemented. As with the analysis phase, the design phase has more than one component. Just as the analysis phase should conclude with some result—a full set of use cases and a logical object model of the system—the design phase should also provide results: task and deployment models, and more refined logical object models.

Architectural Design

Define the major architectural pieces of the system—what are the high-level parts? Define what the system domains are and which key classes fit in each domain. Which are your composite classes? In this analysis, you should also map classes, packages, and components to the relevant physical parts of your system—the processors and devices. Define which libraries and executables are necessary in your model. You are creating the task and deployment models for your system. To help with this, you can apply UML design patterns as appropriate for your system.

Mechanistic Design

Continue to detail the internal workings of your system, breaking it down into smaller pieces and more classes, if necessary. Use “white-box” sequence diagrams to depict the class interactions within the system. Define the collaborations that are required to realize certain core cases by creating collaboration diagrams. Add to your model the “glue” objects that are used in the UML design patterns that you use. Again, you can use the code generation and animation tools to debug and test the model at this point. Your end result should be a more refined set of logical object models.

Detailed Design

Continue to fill in the details of your design. Get your individual classes working; fully define their constraints, internal data structures, and message passing behavior. Use activity diagrams and statecharts to define correct behavior. At this stage, you will probably begin typing in extra code in the implementation boxes in various diagrams. Use component diagrams to define the physical artifacts of your system and to include the libraries, executables, or legacy code you have deemed necessary for your model. Make low-level decisions about implementations, such as choosing static or dynamic instantiations. This should result in a more refined logical object model (or models) of your system.

Implementation

The implementation phase is essentially the code generation and unit testing phase. Using Rhapsody, write the code that is not generated automatically, such as the bodies of non-statechart operations. These include constructors, destructors, object methods, and global functions. Use the animation and tracing tools to test and debug sections of code and to make decisions about any optimization trade-offs.

Testing

In the testing phase, you determine not only whether your model is working, but whether it meets the requirements that you set in the analysis phase. Your end result should be a working system.

Rhapsody includes the following features to assist with the testing phase:

- ◆ **Animator**—Enables you to create test scripts to apply external test stimuli to the system.
- ◆ **Tracer**—Enables you to perform white, gray, and black-box regression testing. It also provides performance testing based on timing annotations or on a simulated time facility.
- ◆ **Sequence diagram comparison**—Automatically compares requirement sequences with implementation sequences.

Classes and Types

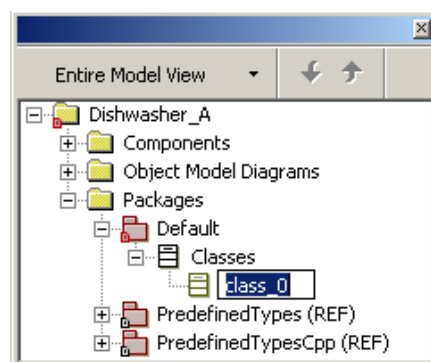
Classes provide a specification (blueprint) for *objects*, which are self-contained, uniquely identified, run-time entities that consist of both data and operations that manipulate this data. Classes can contain attributes, operations, events, relations, components, super classes, types, actors, use cases, diagrams, and other classes. The Rhapsody browser icon for a class is a three-compartment box with the top, or name, compartment filled in. For an example of this icon, see [Defining the Attributes of a Class](#).

Creating a Class

In general, to create a class, in the Rhapsody browser, do one of the following:

- ◆ Right-click the **Classes** category to which you want to add a class and select **Add New Class**.
- ◆ Right-click a package and select **Add New > Class**.
- ◆ Select a package and select **Edit > Add New > Class** from the menu bar.

Rhapsody creates a new class and names it `class_n`, where n is greater than or equal to 0. The new class is located in the browser under the **Classes** category, and is selected, as shown in the following figure, so that you can rename it.



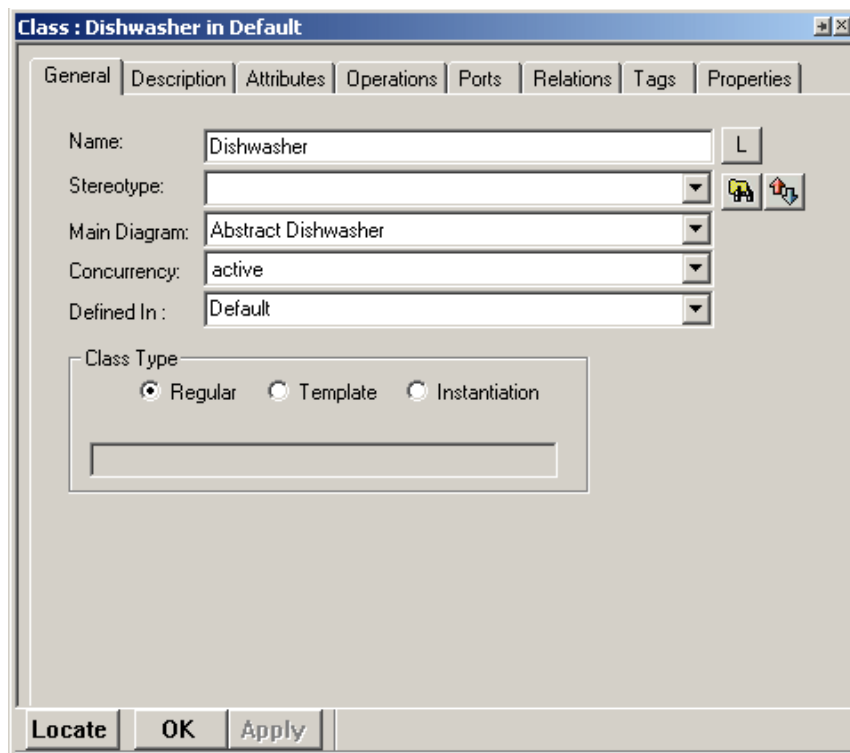
See [Object Model Diagrams](#) for information on creating classes in OMDs.

Defining the Features of a Class

Use the Features dialog box to define and modify a class. You can also use it to re-arrange the order of attributes and operations, control the display of attributes and operations, create templates, and so forth. To open the Features dialog box for a class, double-click it on the Rhapsody browser, or right-click it and select **Features** from the pop-up menu.



Defining the Characteristics of a Class

The **General** tab of the Features dialog box, as shown in the following figure, enables you to define the characteristics of a class.



On the **General** tab, you define the general features for a class through the various controls on the tab.

- ◆ In the **Name** box you specify the name of the element. The default name is `class_n`, where `n` is an incremental integer starting with 0. To enter a detailed description of the class, use the **Description** tab.
- ◆ You use the **L** button to open the Name and Label dialog box to specify the label for the element, if any. See [Labeling Elements](#) for information on creating labels.

- ◆ In the **Stereotype** drop-down list box you specify the stereotype of the element, if any. See [Defining Stereotypes](#) for information on creating stereotypes.
 - To select from a list of current stereotypes in the project, click the Select Stereotype icon .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order icon .

Note: The COM stereotypes are constructive; that is, they affect code generation. Refer to the *Rhapsody COM Development Guide* for more information about these components.

- ◆ In the **Main Diagram** drop-down list box you specify the diagram (from the ones available) that contains the most complete view of the class.
- ◆ In the **Concurrency** drop-down list box you specify the concurrency. The possible values are as follows:
 - **Active** means the class runs on its own thread.
 - **Sequential** means the class runs on the system thread.
- ◆ In the **Defined In** drop-down list box you specify the owner for the class. Every class lives inside either a package or another class.

Note: A class not explicitly drawn in a package belongs to the default package of the diagram. If the diagram is not explicitly assigned to a package, the diagram belongs to the default package of the project.

- ◆ In the **Class Type** area you specify the class type. The possible values are as follows:
 - **Regular** creates a class.
 - **Template** creates a template. To specify the necessary arguments, use the **Template Parameters** tab that appears once you select the **Template** radio button. For more information, see [Creating a Template Class](#).
 - **Instantiation** creates an instantiation of a template. To specify the necessary arguments, use the **Template Instantiation** tab that appears once you select the **Instantiation** radio button. For more information, see [Instantiating a Template Class](#).

Note: To create an instance of a class, select the **Instantiation** radio button and select the template that the instance is from. For example, if you have a template class **A** and create **B** as an instance of that class, this means that **B** is created as an instance of class **A** at run time.

Selecting Nested Classes in Dialog Boxes

Every primary model element is uniquely identified by a path in the following form:

`<ns1>::<ns2>::...<nsn>::<name>`

In this syntax, `ns` can be either a package or a class. Primary model elements are packages, classes, types, and diagrams. Classes can contain only other classes, stereotyped classes (such as actors), and types.





You can select a nested element in a dialog box by entering its name in either of the following formats:

- ◆ `<name> in <ns1>::<ns2>::...<nsn>`
- ◆ `<ns1>::<ns2>::...<nsn>::<name>`

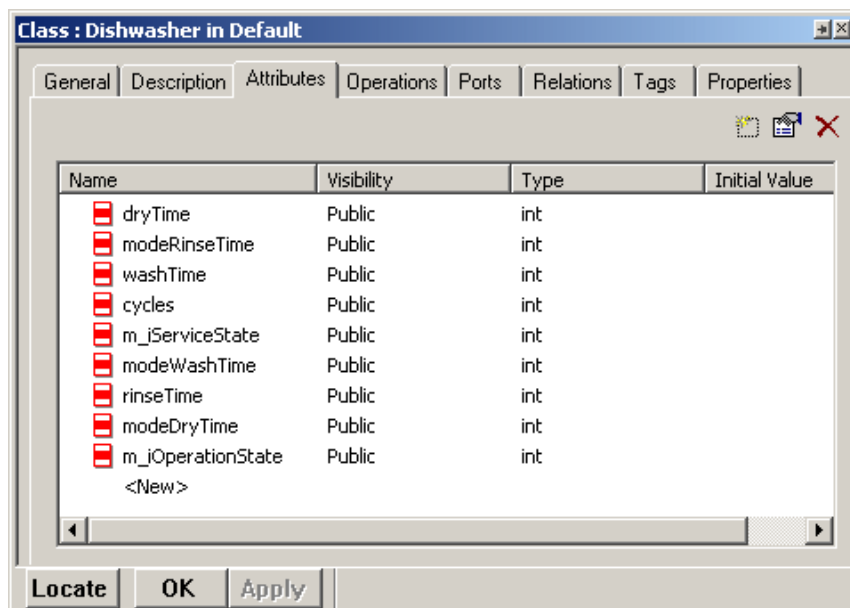
Defining the Attributes of a Class

Attributes are the data members of a class. Rhapsody automatically generates accessor (`get`) and mutator (`set`) methods for attributes, so you do not need to define them yourself.

The Rhapsody browser icon for attributes is a three-compartment class box with the middle compartment filled in:


-  The icon for the **Attributes** category is black.
-  The icon for an individual attribute is red.
-  The icon for a protected attribute is overlaid with a key.
-  The icon for a private attribute is overlaid with a padlock.

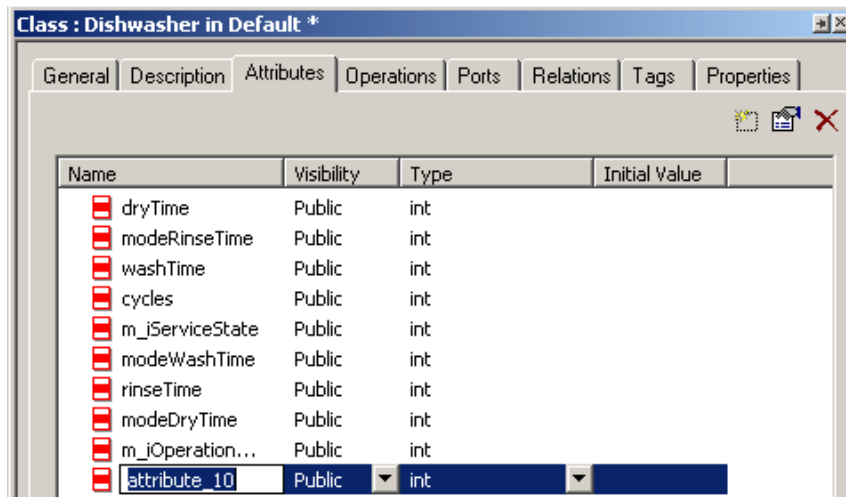
The **Attributes** tab of the Features dialog box contains a list of the attributes that belong to the class, as shown in the following figure.



The **Attributes** tab enables you to perform the following tasks:


- ◆ Add a new attribute.

To create a new attribute, either click the <New> row in the list of attributes, or click the New icon  in the upper, right corner of the dialog box. The new row is filled in with the default values, as shown in the following figure.




- ◆ **Modify an existing attribute.**

To modify an attribute, you can use any of the following methods:

- Select the attribute and change the value name and/or change its parameters from the drop-down list boxes.
- Select the attribute and click the Invoke Feature Dialog icon  to open the Features dialog box for the attribute and make your changes there. You can also double-click the attribute name or icon next to the name to open the Features dialog box.

- ◆ **Delete an attribute.**

To delete an attribute from the model, select the attribute and click the Delete icon .

- ◆ **View the attribute values.**


To view the values for an attribute, open the Features dialog box for it.

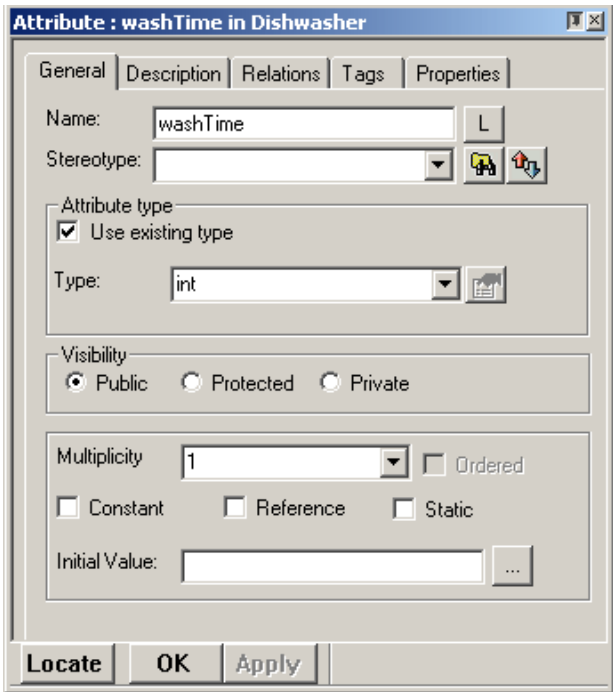
You can use the following keyboard shortcuts within an editable list:

- ◆ **Arrow** keys to move between rows and columns.
- ◆ **Enter** key to start or stop editing in a text box, or to make a selection in a combo box.

- ◆ **Insert** key to insert a new element below the selected element.
- ◆ **Delete** key to delete the selected element.
- ◆ **Esc** key to cancel editing.

Defining the Features of an Attribute



When you click the Invoke Features Dialog icon  or double-click an attribute, the Attribute dialog box opens, as shown in the following figure. This dialog box is also displayed when you select an attribute in the browser and may have different options than shown in here.



The screenshot shows the 'Attribute : washTime in Dishwasher' dialog box. It has five tabs: General, Description, Relations, Tags, and Properties. The 'General' tab is selected. The 'Name' field contains 'washTime'. There is an 'L' button next to it. The 'Stereotype' field is empty. Below it, there is a section for 'Attribute type' with a checked box for 'Use existing type' and a 'Type' dropdown set to 'int'. The 'Visibility' section has radio buttons for 'Public' (selected), 'Protected', and 'Private'. The 'Multiplicity' is set to '1' with an 'Ordered' checkbox. There are checkboxes for 'Constant', 'Reference', and 'Static', all of which are unchecked. An 'Initial Value' field is empty. At the bottom, there are 'Locate', 'OK', and 'Apply' buttons.

On the **General** tab, you define the general features for a attribute through the various controls on the tab.

- ◆ In the **Name** box you specify the name of the attribute. The default name is `attribute_n`, where n is an incremental integer starting with 0. To enter a detailed description of the attribute, use the **Description** tab.
- ◆ You use the **L** button to open the Name and Label dialog box to specify the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ In the **Stereotype** drop-down list box you specify the stereotype of the attribute, if any. See [Defining Stereotypes](#) for information on creating stereotypes.

- To select from a list of current stereotypes in the project, click the Select Stereotype icon .
- To sort the order of the selected stereotypes, click the Change Stereotype Order icon .

Note: The COM stereotypes are constructive; that is, they affect code generation. Refer to the *Rhapsody COM Development Guide* for more information about these components.

- ◆ In the **Attribute type** area you specify the attribute type. There are two ways to specify the type:
 - Select the **Use existing type** check box to select a predefined or user-defined type or class. Use the **Type** drop-down list box to select from among the Rhapsody predefined types, and any types and classes you have created in this project. Or to define a new type, delete the value in the **Type** drop-down list box to enable the Invoke Feature Dialog icon and click it to open the Type dialog box.

See [Composite Types](#) for detailed information on creating types.


- Clear the **Use existing type** check box if there is no defined type. A **C++[Java] Declaration** box appears in which you can give the attribute a declaration appropriate for your language edition. See [Modifying Data Types](#).
- ◆ In the **Visibility** area you specify the type of access (visibility) for the accessor/mutator generated for the attribute: **Public**, **Protected**, or **Private**.

When you generate code, each attribute is generated into three entities:

- The data member itself
- An accessor (`get`) method for retrieving the data value
- A mutator (`set`) method for setting the data value

Note: The visibility setting affects only the visibility of the accessor and mutator methods, not of the data member itself. The data member is always protected, regardless of the access setting.

- ◆ In the **Multiplicity** box (enabled when appropriate) you specify the multiplicity of the attribute. If this is greater than 1, use the **Ordered** check box to specify whether the order of the reference type items is significant. The modifier choices are as follows:
 - **Constant** specifies whether the attribute is read-only (check box is selected) or modifiable (check box is cleared).
 - **Reference** specifies whether the attribute is referenced as a reference, such as a pointer (*) or an address (&) in C++.
 - **Static** creates a static attribute, which belongs to the class as a whole rather than to individual objects. See [Initializing Static Attributes](#).

- ◆ In the **Initial Value** box you specify the initial value for the attribute. To access the text editor, click the Ellipses icon .

Note: Throughout the Rhapsody interface, the Ellipses button invokes a text editor.

Modifying Data Types

To create or edit a user-defined data type, follow these steps:

1. Open the Features dialog box for the attribute.
2. In the **Attribute type** area, clear the **Use existing type** check box.
3. Type a declaration for the new type in the **C++[Java] Declaration** box using the proper syntax. Note the following:
 - ◆ You can omit the semicolon at the end of the line; Rhapsody automatically adds one if it is not present.
 - ◆ Substitute `%s` for the name of the type in the declaration. For example:


```
typedef unsigned char %s[100]
```

This translates to the following declaration in the generated code:

```
typedef unsigned char msg_t[100];
```
4. Add a description for the type on the **Description** tab.
5. Click **OK** to apply your changes and dismiss the Attribute dialog box.

Rhapsody adds it to the **Types** category under the package to which the class belongs, rather than under the class itself.

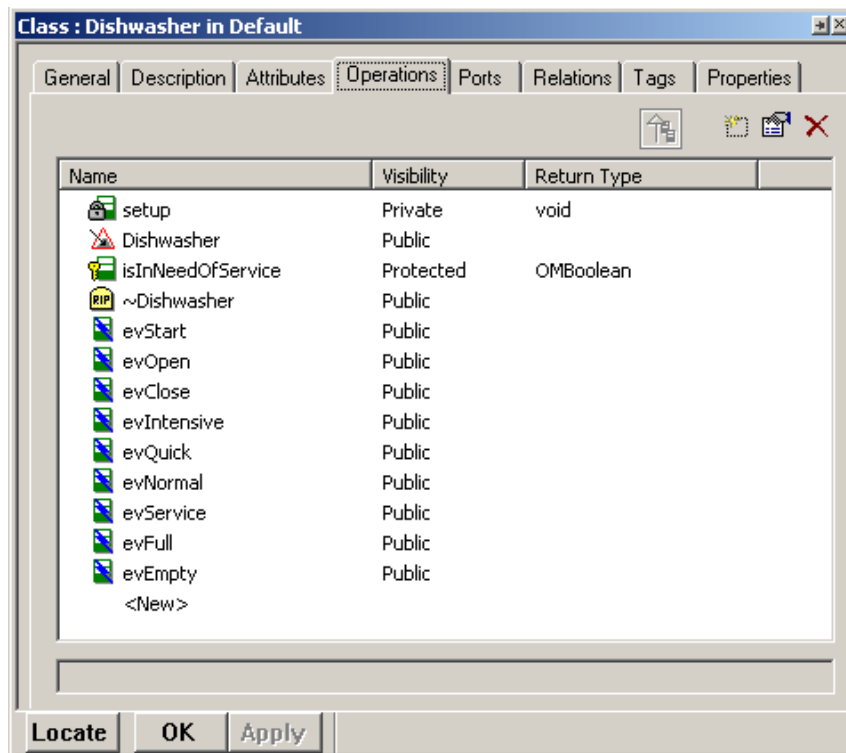
Initializing Static Attributes

If you select the **Static** check box on the Features dialog box for an attribute, use the **Initial Value** box to enter an initial value. You can invoke a text editor for entering initialization code by clicking the Ellipses icon  associated with the box.

See [Generating Code for Static Attributes](#) for information on code generation for static attributes.

Defining Class Operations

The **Operations** tab of Features dialog box for a class enables you to add, edit, or remove operations from the model or from the current OMD view. It contains a list of all the operations belonging to the class, as shown in the following figure.



Rhapsody enables you to create sets of standard operations for classes and events. For more information, see [Using Standard Operations](#).

You can create the following types of operations:

- ◆ [Primitive Operations](#)
- ◆ [Receptions](#)
- ◆ [Triggered Operations](#)
- ◆ [Constructors](#)
- ◆ [Destructors](#)

Primitive Operations

A *primitive operation* is one whose body you define yourself instead of letting Rhapsody generate it for you from a statechart.

Creating a Primitive Operation

To create a primitive operation using the Features dialog box for a class, follow these steps:

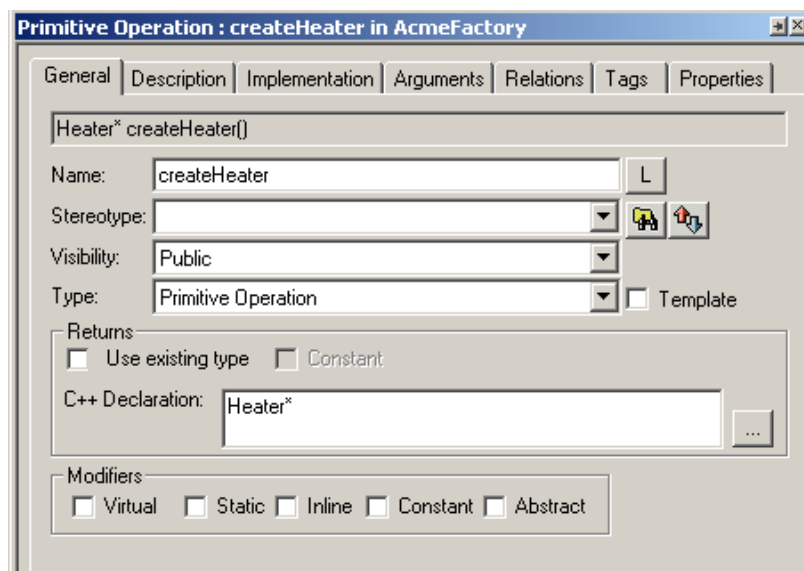
1. On the Rhapsody browser, double-click a class to open its Features dialog box.
2. On the **Operations** tab, either click the <New> row in the list of operations or click the New icon in the upper, right corner of the dialog box and select **PrimitiveOperation** from the pop-up menu. The new row is filled in with the default values.
3. By default, Rhapsody names the new primitive operation `Message_n`, where *n* is greater than or equal to 0. Type the new name for the operation in the **Name** column.
4. Change the other default values as necessary.
5. Click **OK**.

Alternatively, you can create a primitive operation through the use of the Rhapsody browser, as follows:



1. Select the class, actor, operation, or use case node to which to add the operation on the Rhapsody browser.
2. Do either of the following:
 - ◆ Right-click and select **Add New > Operation** from the pop-up menu.
 - ◆ Choose **Edit > Add New > Operation** from the menu bar.
3. Rename the operation.

Defining the Features of a Primitive Operation

The Features dialog box for a primitive operation enables you to change the features for it, including its return values, arguments, and modifiers. The following figure shows the **General** tab of the Features dialog box for a primitive operation.




On this **General** tab, you define the general features for a primitive operation through the various controls on the tab. Notice that the primitive operation's signature is displayed at the top of the **General** tab of the Features dialog box.

- ◆ In the **Name** box you specify the name of the element. The default name is `Message_n`, where n is an incremental integer starting with 0. To enter a detailed description of the operation, use the **Description** tab.
- ◆ You use the **L** button to open the Name and Label dialog box to specify the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ In the **Stereotype** drop-down list box you specify the stereotype of the attribute, if any. See [Defining Stereotypes](#) for information on creating stereotypes.
 - To select from a list of current stereotypes in the project, click the Select Stereotype icon .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order icon .

Note: The COM stereotypes are constructive; that is, they affect code generation. Refer to the *Rhapsody COM Development Guide* for more information about these components.

- ◆ In the **Visibility** drop-down list box you specify the visibility of the primitive operation: **Public**, **Protected**, or **Private**.
- ◆ In the **Type** drop-down list box you specify the operation type. For primitive operations, this box is set to **Primitive Operation**. If this is a template class, select the **Template** check box. To specify the necessary arguments for the template, use the **Template Parameters** tab that appears once you select the **Template** check box. For more information, see [Creating a Template Class](#).
- ◆ In the **Returns** area you specify the return type of a function.
 - If the function will return a defined type, select the **Use existing type** check box and select the return type from the **Type** drop-down list box that appears once you select the check box. Or to define a new type, delete the value in the **Type** drop-down list box to enable the Invoke Feature Dialog icon and click it to open the Type dialog box.

See [Composite Types](#) for detailed information on creating types.
 - If you want to use a type that is not defined, clear the **Use existing type** check box. A **C++[Java] Declaration** box appears. Enter the code as you want it to appear in the return statement. To access the text editor, click the Ellipses icon .
- ◆ In the **Modifiers** area you specify the modifiers of the operation. The possible values are **Virtual**, **Static**, **Inline**, **Constant**, or **Abstract**, but the available modifier types vary according to the type of operation.

Receptions

A *reception* specifies the ability of a class to react to a certain event (called a *signal* in the UML). The name of the reception is the same as the name of the event; therefore, you cannot change the name of a reception directly. Receptions are displayed under the **Operations** category for the class.

Creating a Reception Using the Features Dialog Box

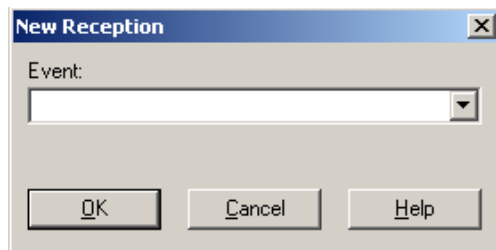
To create a reception using the Features dialog box, follow these steps:

1. On the Rhapsody browser, double-click a class to open its Features dialog box.
2. On the **Operations** tab, either click the <New> row in the list of operations or click the New icon in the upper, right corner of the dialog box and select **Reception** from the pop-up menu. The new row is filled in with the default values.
3. Type the name of the reception in the **Event** box on the New Reception dialog box. If Rhapsody cannot find an event with the given name, a confirmation box opens, prompting you to create a new event. Click **Yes** to create a new event and the specified reception.
4. Open the Features dialog box for the new reception operation you just created and set its other values as necessary.
5. Click **OK**.

Creating a Reception Using the Browser

To create a reception using the Rhapsody browser, follow these steps:

1. In the browser, select a class, actor, operation, or use case node.
2. Select **Add New > Reception** from the pop-up menu. The New Reception dialog box opens, as shown in the following figure.



3. Type the name of the new reception and click **OK**.
4. The following happens depending on what Rhapsody finds:

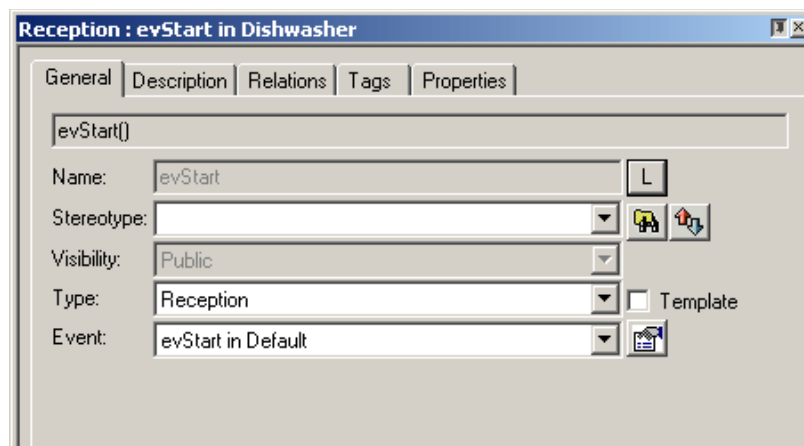
- If Rhapsody finds an event with the specified name, it creates the new reception and displays it in the browser.
- If Rhapsody cannot find an event with the given name, a confirmation box opens, prompting you to create a new event. Click **Yes** to create a new event and the specified reception.

Note the following:

- ◆ When you add a new reception with a new name to a class, an event of that name is added to the package. If you specify an existing event name, the reception simply points to that event.
- ◆ Receptions are inherited. Therefore, if you give a trigger to a transition with a reception name that does not exist in the class but does exist in its base class, Rhapsody does not create a new reception.



Defining the Features of a Reception

The Features dialog box for a reception, as shown in the following figure, enables you to change the features of a reception, including its type and the event to which the reception reacts.



On this **General** tab, you define the general features for a reception through the various controls on the tab. Notice that the reception's signature is displayed at the top of the **General** tab of the Features dialog box.

- ◆ In the **Name** box you specify the name of the reception. The default name is `event_n`, where n is an incremental integer starting with 0. To enter a detailed description of the reception, use the **Description** tab.
- ◆ If the **Name** box is inaccessible, as shown in the above figure, click the **L** button to open the Name and Label dialog box to change the name, if any. See [Labeling Elements](#) for information on creating labels.

- ◆ In the **Stereotype** drop-down list box you specify the stereotype of the attribute, if any. See [Defining Stereotypes](#) for information on creating stereotypes.
 - To select from a list of current stereotypes in the project, click the Select Stereotype icon .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order icon .
- Note:** The COM stereotypes are constructive; that is, they affect code generation. Refer to the *Rhapsody COM Development Guide* for more information about these components.
- ◆ In the **Visibility** drop-down list box you specify the visibility of the reception (**Public**, **Protected**, or **Private**), if available.
 - ◆ In the **Type** drop-down list box you specify the operation type. For receptions, this box is set to **Reception**.
 - ◆ In the **Event** drop-down list box you specify the event to which the reception reacts. To view or modify the features of the event itself, click the Invoke Feature Dialog icon.

Deleting Receptions

You cannot delete receptions in the following cases:

- ◆ The reception is used by the statechart of the class.
- ◆ The reception is used by a derived statechart of a class that does not have its own reception.

Triggered Operations

A *triggered operation* can trigger a state transition in a statechart, just like an event. The body of the triggered operation is executed as a result of the transition being taken. See [Triggered Operations](#) for more information.

Constructors

Constructors are called when a class is instantiated, generally to initialize data members with values relevant to that object.

Rhapsody has the following constructor icons:



The Rhapsody browser icon for a constructor is a red triangle with a black arrow.



The icon for a protected constructor is overlaid with a key.



The icon for a private constructor is overlaid with a padlock.

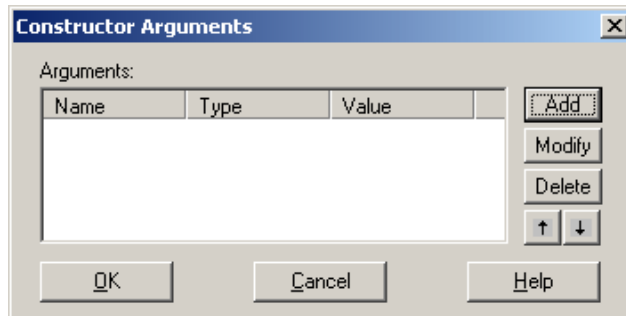
Creating a Constructor

To create a constructor, follow these steps:

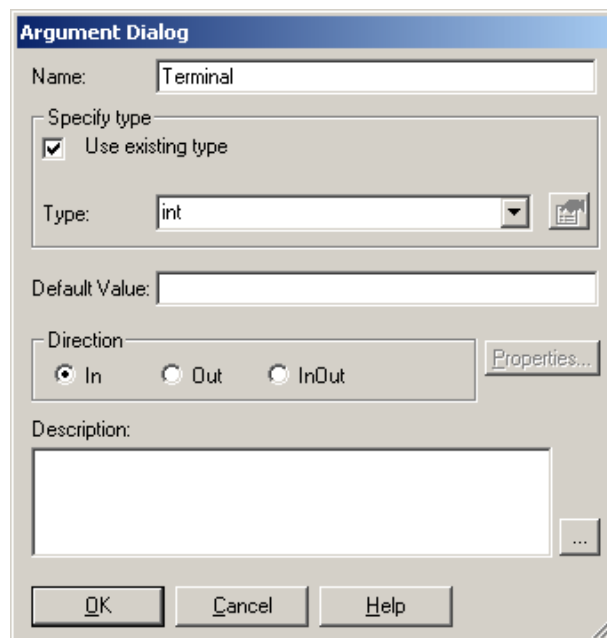
1. Depending on if you want to use the Features dialog box or the Rhapsody browser:
 - ◆ On the Rhapsody browser, double-click a class to open its Features dialog box and on the **Operations** tab, either click the <New> row in the list of operations or click the New icon in the upper, right corner of the dialog box and select **Constructor** from the pop-up menu.
 - ◆ On the Rhapsody browser, right-click either the class or the **Operations** category under the class and select **Add New > Constructor**.

Note: Alternatively, you can invoke this dialog box by highlighting the appropriate element in a diagram, right-clicking, and selecting **New Constructor** from the pop-up menu.

2. The Constructor Arguments dialog box opens, as shown in the following figure.

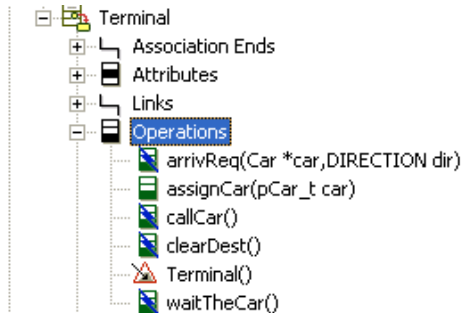


3. Click **Add**. The Argument Dialog box opens, as shown in the following figure.



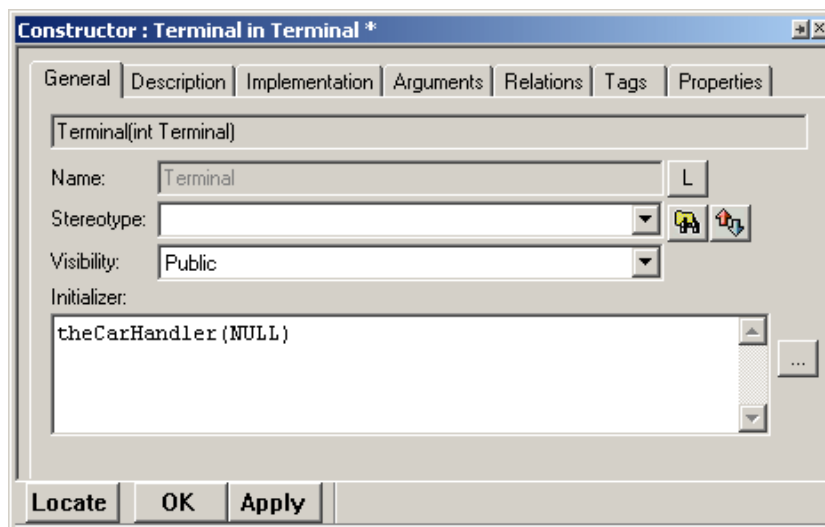
4. Type in a name for the new constructor and change the default values as necessary.
5. Click **OK** to close the Argument Dialog box.
6. Click **OK** to close the Constructor Argument dialog box.

The new constructor is listed under the **Operations** category for the class in the Rhapsody browser, as shown for the Terminal class operation in the following figure.





Defining Constructor Features

The Features dialog box enables you to change the features of a constructor, including its arguments and initialization code. Double-click the constructor in the Rhapsody browser to open its Features dialog box, as shown in the following figure.




On this **General** tab, you define the general features for a constructor through the various controls on the tab. Notice that the constructor's signature is displayed at the top of the **General** tab of the Features dialog box.

- ◆ In the **Name** box you specify the name of the constructor. The default name is the name of the class it creates. To enter a detailed description of the constructor, use the **Description** tab.

- ◆ If the **Name** box is inaccessible, as shown in the above figure, click the **L** button to open the Name and Label dialog box to change the name, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ In the **Stereotype** drop-down list box you specify the stereotype of the attribute, if any. See [Defining Stereotypes](#) for information on creating stereotypes.
 - To select from a list of current stereotypes in the project, click the Select Stereotype icon .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order icon .

Note: The COM stereotypes are constructive; that is, they affect code generation. Refer to the *Rhapsody COM Development Guide* for more information about these components.

- ◆ In the **Visibility** drop-down list box you specify the visibility of the reception (**Public**, **Protected**, or **Private**), if available. The default value is **Public**.
- ◆ In the **Initializer** box you enter code if you want to initialize class attributes or super classes in the constructor initializer. To access the text editor, click the Ellipses icon .

For example, to initialize a class attribute called `a` to `5`, type the following code:

```
a(5)
```

Note: In C++, this assignment is generated into the following code in the class implementation file to initialize the data member in the constructor initializer rather than in the constructor body:

```
//-----  
// A.cpp  
//-----  
A::A() : a(5) {  
    //#[operation A()  
    //#]  
};
```

Note: You must initialize `const` data members in the constructor initializer rather than in the constructor body.

To enter code for any initializations that you want to perform in the constructor body rather than in the constructor initializer, use the **Implementation** tab. You can create and initialize objects participating in relationships within a constructor's body. You can pass arguments to these objects if they have overloaded constructors using, for example:

```
new relatedClass(3)
```

This code in the body of the class constructor calls the constructor for the related class and passes it a value of 3. The related class must have a conversion constructor that accepts a parameter. The constructor of the related class then performs its initialization using the passed-in value.

Destroyers

A *destructor* is called when an object is destroyed, for example, to de-allocate memory that was dynamically allocated for an attribute during construction.

Rhapsody has the following destructor icons:



The Rhapsody browser icon for a destructor is a tombstone (RIP = Rest In Peace).



The icon for a protected destructor is overlaid with a key.



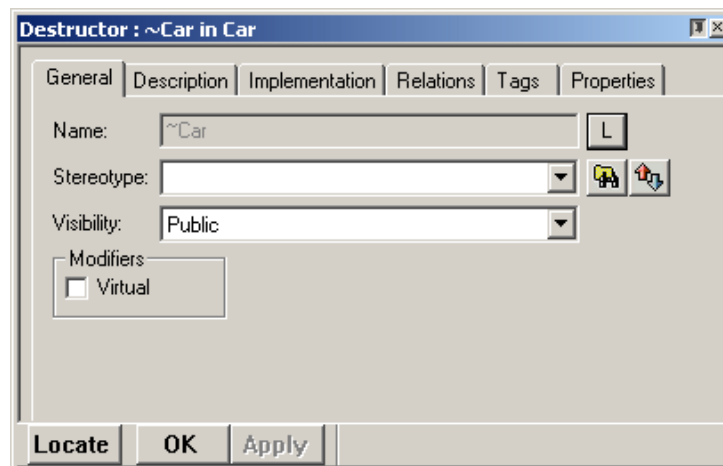
The icon for a private destructor is overlaid with a padlock.

Creating a Destructor



To create a destructor, follow the instructions for [Creating a Primitive Operation](#), but for the type, select **Destructor** from the pop-up menu.

Modifying the Features of a Destructor

The Features dialog box, as shown in the following figure, enables you to change the features of a destructor, including its visibility and modifier.



On this **General** tab, you define the general features for a destructor through the various controls on the tab.

- ◆ In the **Name** box you specify the name of the destructor. By definition, destructors have the same name as the class, preceded by a tilde (~) symbol. To enter a detailed description of the reception, use the **Description** tab.
- ◆ If the **Name** box is inaccessible, as shown in the above figure, click the **L** button to open the Name and Label dialog box to change the name, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ In the **Stereotype** drop-down list box you specify the stereotype of the attribute, if any. See [Defining Stereotypes](#) for information on creating stereotypes.
 - To select from a list of current stereotypes in the project, click the Select Stereotype icon .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order icon .

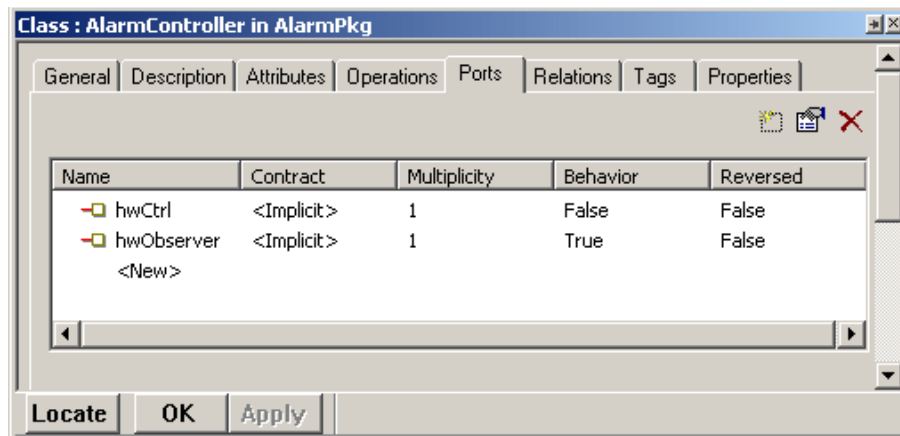
Note: The COM stereotypes are constructive; that is, they affect code generation. Refer to the *Rhapsody COM Development Guide* for more information about these components.

- ◆ In the **Visibility** drop-down list box you specify the visibility of the reception (**Public**, **Protected**, or **Private**). By default, destructors are **Public**.
- ◆ In the **Modifiers** area you specify the modifiers of the destructor. Select **Virtual**, if desired.

To type code for the body of the destructor, use the **Implementation** tab.

Defining Class Ports

Use the **Ports** tab, as shown in the following figure, to create, modify, and delete class ports.



The **Ports** tab contains the following columns:

- ◆ **Name** specifies the name of the port.
- ◆ **Contract** specifies the contract of the port. See [The Contract Tab](#) for more information about contracts. The possible values are as follows:
 - **Implicit** means that the contract is a “hidden” interface that exists only as the contract of the port.
 - **Explicit** mean that the contract is an explicit interface in the model. An explicit contract can be reused so several ports can have the same contract.
- ◆ **Multiplicity** specifies the multiplicity of the port. The default value is 1.
- ◆ **Behavior** specifies whether the port is a behavioral port, which means that the messages of the provided interface are forwarded to the owner class. If it is *non-behavioral*, the messages are sent to one of the internal parts of the class.
- ◆ **Reversed** specifies whether the interfaces are reversed, so the provided interfaces become the required interfaces and vice versa.

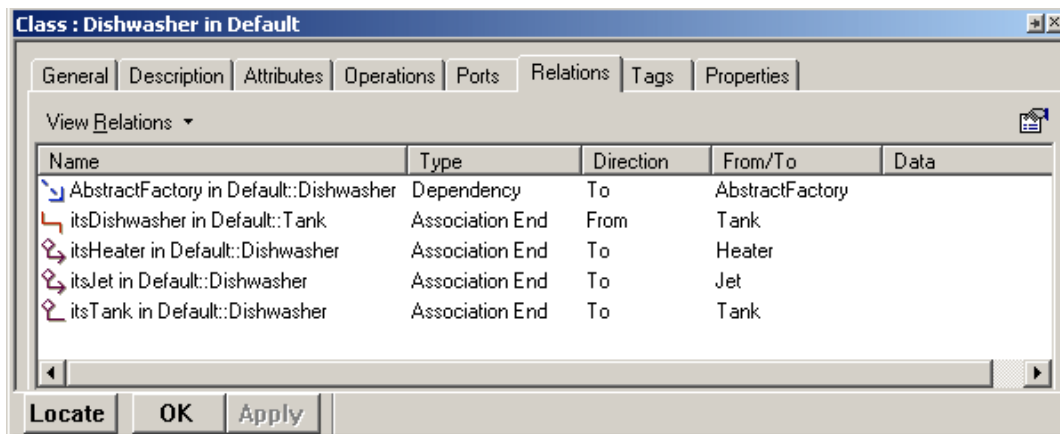
See [Defining the Attributes of a Class](#) for instructions on how to use the interface on this tab to create, modify, or delete a port.

See [Ports](#) for detailed information about specifying ports.

Defining Relations

The term “relations” refers to all the relationships you can define between elements in the model (not just classes)—associations, dependencies, generalization, flows, and links. (In previous versions of Rhapsody, the term referred to all the different kinds of associations.)

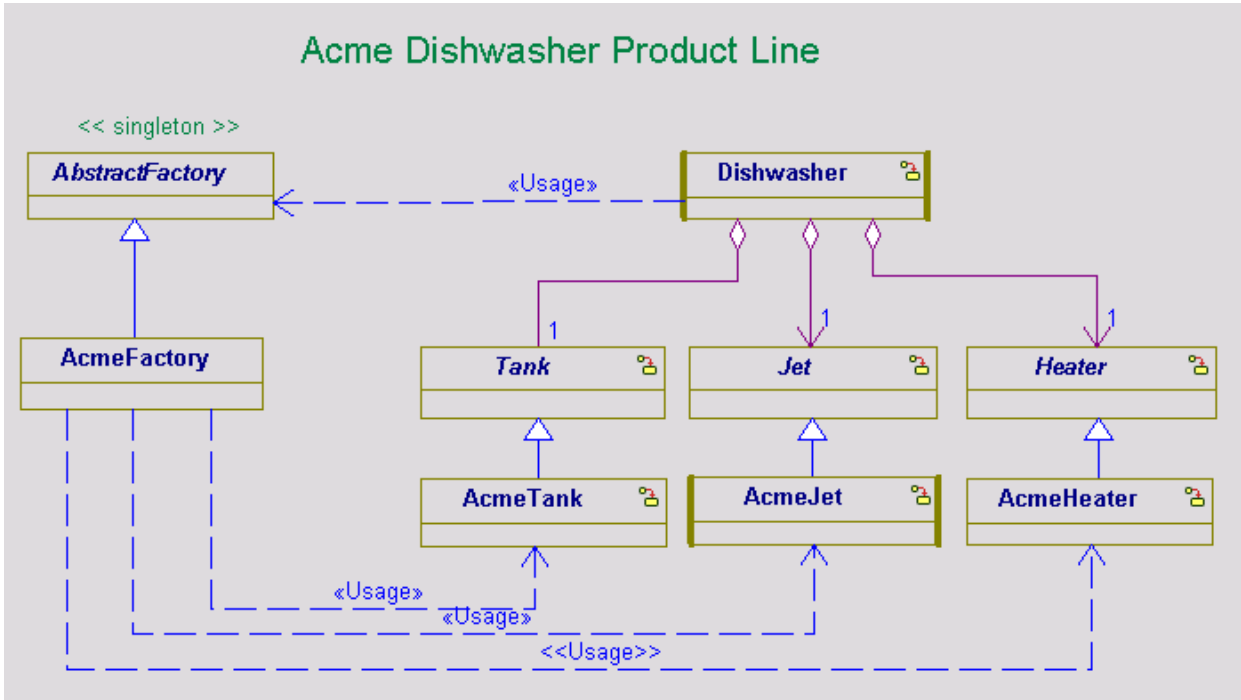
The **Relations** tab lists all the relationships (dependencies, associations, and so on) the class is engaged with, as shown in the following figure.



The **Relations** tab contains the following columns:

- ◆ **Name** specifies the name of the relation.
- ◆ **Type** specifies the relation type (for example, **Association End** and **Dependency**).
- ◆ **Direction** specifies the direction of the relationship (**From** or **To**).
- ◆ **From/To** specifies the target of the relationship. For example, as shown in the figure, the dependency `Dishwasher.AbstractFactory` goes from the `Dishwasher` class to the `AbstractFactory`.
- ◆ **Data** specifies any additional data used by the relationship.

For example, if you have a port that provides the interface `MyInterface`, the **Data** column would list `Provided Interface`.



See [Associations](#) for more information on relationships.

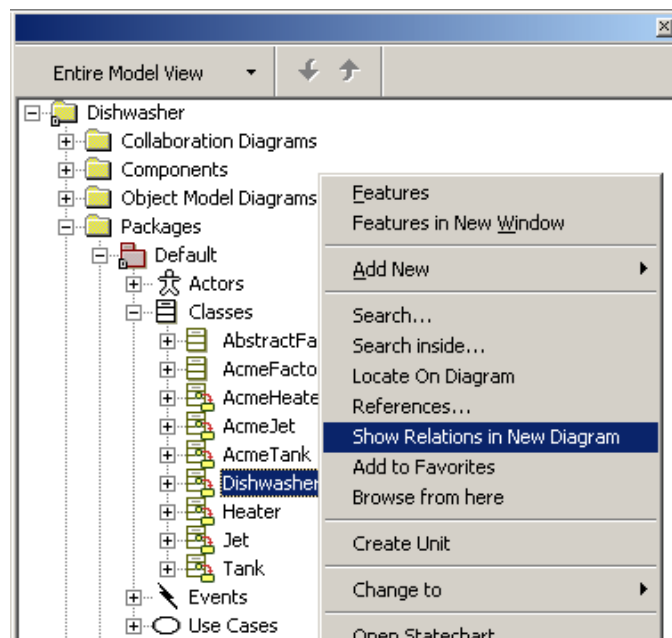
Showing All Relations for a Class, Object, or Package in a Diagram

To fully understand the meaning or purpose of a class, Rhapsody lets you create an object model diagram that shows a class and its relations to all the other elements in a project.

Note that you can show the relations for a class, an object, and a package. You would use the same procedure as noted below, except that you would select that particular element (for example, an object) instead.

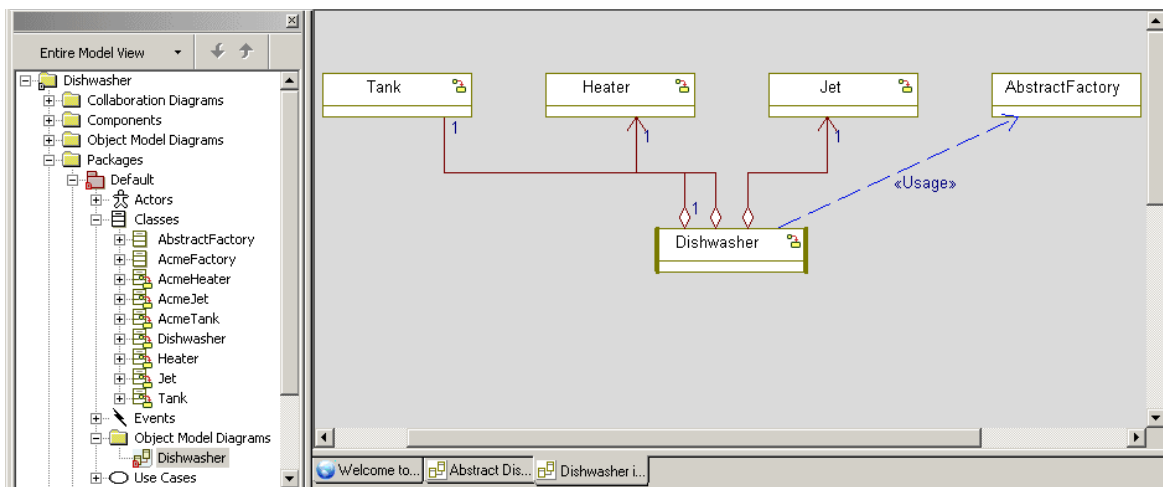
To show all the relations of a class, follow these steps:

1. Right-click a class in the Rhapsody browser and select **Show Relations in New Diagram** from the pop-up menu, as shown in the following figure.



2. Notice the following:

- ◆ Rhapsody created an object model diagram that shows all the relations for the selected class, as shown on right side of the following figure.
- ◆ Rhapsody also named the new object model diagram from the name of the class you selected and it created an **Object Model Diagram** category to hold the new diagram within the package where the class resides, as shown in the left side of figure. (Notice that there is no **Object Model Diagram** category in the browser shown in step 1.)



Note the following:

- ◆ The **Show Relations in New Diagram** pop-up menu command is available for classes, objects, and packages from the Rhapsody browser as well as on a diagram. For both the browser and on a diagram, you would right-click the element (for example, an object) and select **Show Relations in New Diagram**.
 - Note:** For a diagram you can select multiple elements (for example, two classes). Use **Ctrl+Click** to make multiple selections and then right-click one of the selected elements to open the pop-up menu and click **Show Relations in New Diagram**.
- ◆ The location of the new object model diagram created by **Show Relations in New Diagram** depends on whether you invoked the command from the browser or a diagram:
 - When invoked from the browser, **Show Relations in New Diagram** creates the diagram in the same location as the selected element and places it in an **Object Model Diagram** category (which Rhapsody creates if the category is not already available).
 - When invoked from a diagram, **Show Relations in New Diagram** creates the object model diagram in the same location in which the source diagram resides.

- ◆ The new object model diagram created by **Show Relations in New Diagram** will be named the same name as the class, object, or package from which it was created. If there is already an object model diagram with that name, a number will be appended to the name (for example, `Dishwasher_9`).
- ◆ See also [Automatically Populating Diagrams](#).

Defining Class Tags

The **Tags** tab lists the available tags for this class. See [Using Profiles](#) for detailed information on tags.

Defining Class Properties

The **Properties** tab lets you set and edit class properties and displays the definitions for individual properties with their default values. Refer to the *Rhapsody Properties Reference Manual* for information about the organization of Rhapsody's properties and their uses. The definition of an individual property displays at the bottom of the Features dialog box when a property is highlighted in the **Property** tab. To examine the complete list of property definitions, refer to the *Rhapsody Property Definitions* PDF file available from the *List of Books*. This list can be searched along with the other PDF versions of the Rhapsody documentation to locate specific property definitions and the procedures that use them.

Adding a Class Derivation

A class *derivation* is modeled as a dependency relationship with a stereotype of <<derive>>. Code is not generated from that relationship.

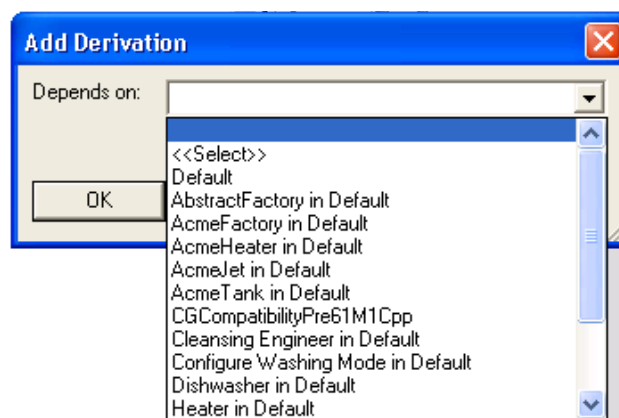
The <<derive>> stereotype specifies a derivation relationship among model elements that are usually, but not necessarily, of the same type. A derived dependency specifies the following:

- ◆ The client may be computed from the supplier.
- ◆ Then mapping specifies the computation.

The client may be implemented for design reasons, such as efficiency, even though it is logically redundant.

To create a class derivation, follow these steps:

1. Highlight the class for which you are creating a derivation.
2. Right-click and select **Add New > Derivation**.
3. In the dialog box, select from the **Depends on** drop-down menu, as shown in the following figure.



The items in the drop-down menu are the following:

- ◆ Elements currently in the model.
 - ◆ Profiles for the development language, as defined in [Opening a Rhapsody Project](#).
 - ◆ <<Select>> displays another dialog box with model browser to allow you to make a selection that is not listed in the drop-down menu.
4. Make your selection and click **OK**. The selected element is then listed under the Class' Derivation in the browser.

Making a Class an Instance

To make a class an instance in an OMD, select the **Instance** tab in the Features dialog box.

If the class represents an instance, check the option **This box is also an instance**. This is equivalent to giving the class an instance name in the OMD. If this box is checked, the following boxes become active:

- ◆ **Instance Name**—Specifies the name of the instance.
- ◆ **Multiplicity**—Specifies the number (or range) of times to instantiate the class.

The multiplicity indicates the number of instances that can participate at either end of a relation. Multiplicity can be shown in terms of a fixed number, a range, or an asterisk (*), meaning any number of instances including zero.

Defining Class Behavior

To define the behavior of a class, you give it either a statechart or an activity diagram. follow these steps:

1. In the OMD, right-click the class.
2. Select either **New Statechart** or **New Activity Diagram** from the pop-up menu.

See [Statecharts](#) or [Activity Diagrams](#) for more information on these diagrams.

Generating, Editing, and Roundtripping Class Code

Rhapsody enables you to generate code and invoke a text editor for editing the generated code directly from within an OMD. The following sections describe these tasks in detail.

Generating Class Code

To generate code for a single class, follow these steps:

1. Right-click the class, then select **Generate** from the pop-up menu.
2. If a directory for the configuration that the class belongs to has not yet been created, Rhapsody asks if you want to create the directory. The configuration directory is under the component directory. Click **Yes**.
3. Rhapsody creates the configuration directory and generates the class code to it. An output window opens at the bottom of the Rhapsody window for the display of code generation messages.

Editing Class Code

To edit code once it has been generated, right-click the class and select **Edit Code** from the pop-up menu. By default, the code generated for the class opens in the Rhapsody internal text editor. If both a specification and an implementation file were generated for the class, both files open, with the specification file in the foreground.

To set Rhapsody to open the editor associated with the file extension instead of the internal editor, follow these steps:

1. Select **File > Project Properties**.
2. Set the `General::Model::ClassCodeEditor` property to `Associate`.
3. Click **OK**.

Roundtripping Class Code

When generating code, Rhapsody places all user code for method bodies and transition code written in statecharts between special annotation symbols. The symbols are as follows:

Language	Body Annotation Symbols
Ada	--+[<ElementType> <ElementName> --+]
C	/*#[<ElementType> <ElementName> */ /*#]*/
C++ and Java	//#[<ElementType> <ElementName> //#]

For example, the following `Initialize()` operation for the `Connection` class in the PBX sample contains user code that was entered in the Implementation field of the Operation dialog box. The user code is placed between the annotation symbols when code is generated for the class:

```
void Connection::Initialize() {
    //#[ operation Initialize()
    DigitsDialed = 0;
    Digits[0] = 0;
    Digits[1] = 0;
    Busy = FALSE;
    Extension = 0;
    //#]
}
```

You can edit the code between the *annotation symbols* in a text editor and then roundtrip your changes back into the model. The roundtripped edits will be retained upon the next code generation. This is how Rhapsody keeps the code and the model in sync to provide model-code associativity.

Note

Any text edits made outside the annotation symbols may be lost with the next code generation. For more information, see [Deleting Redundant Code Files](#).

To roundtrip code changes back into the model, follow these steps:

1. Edit the generated class code between the `/// and /// annotation symbols.`
2. In the browser or diagram, right-click the class containing the code that you just edited.
3. From the resulting pop-up menu, select **Roundtrip**.

If you view the Implementation box of the specification dialog box for the operation (or the statechart for the class if you edited transition code), you can see that your text edits were added to the model.

Opening the Main Diagram for a Class

You can specify a main diagram for a class in the Features dialog box. This is usually the diagram that shows the most important information for a class. For example, in the PBX sample, the PBX diagram is specified as the main diagram for the `Connection` class. The main diagram for a class must be either an object model diagram (OMD) or a use case diagram (UCD).

1. In the OMD, right-click the class.
2. Select **Open Main Diagram** from the pop-up menu.

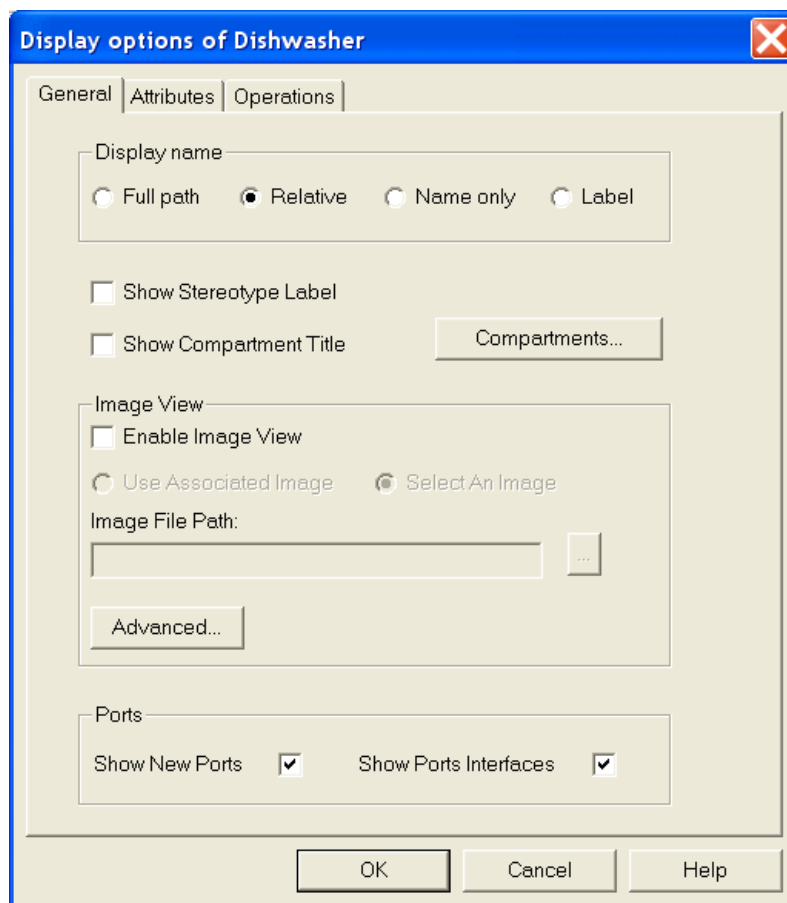
Alternatively, select the element in the browser, then select **Tools > Main Diagram**.

If a main diagram is not specified for the class, nothing happens. Otherwise, the specified main diagram opens.

Setting Display Options

Rhapsody allows a great deal of flexibility in how elements are displayed. Display options relate to how the element name, stereotype, and compartments are displayed in the diagram.

For example, to change the display options for a class, right-click the class in the diagram and select **Display Options** from the pop-up menu. The Display options for the selected class opens, as shown in the following figure.



General Tab Display Options

The first tab of the Display Options dialog box enables you to set general display options, including the class name and stereotype. The **General** tab contains the following controls:

- ◆ **Display name**—Specifies how the class name is displayed. A class is always inside at least one package, but the package can be nested inside other packages, and the class can also be nested inside one or more classes. The class name displayed in the OMD can show multiple levels of nesting in the class name.

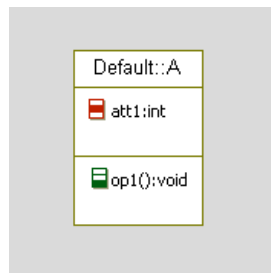
The possible display options are as follows:

- **Full path**—The full path name includes the entire class nesting scheme in the following format:

```
<p1>::<p2>::...::<pn>::<c1>::<c2>::... ::<classname>
```

In this syntax, *p* [*n*] are packages and *c* [*n*] are classes.

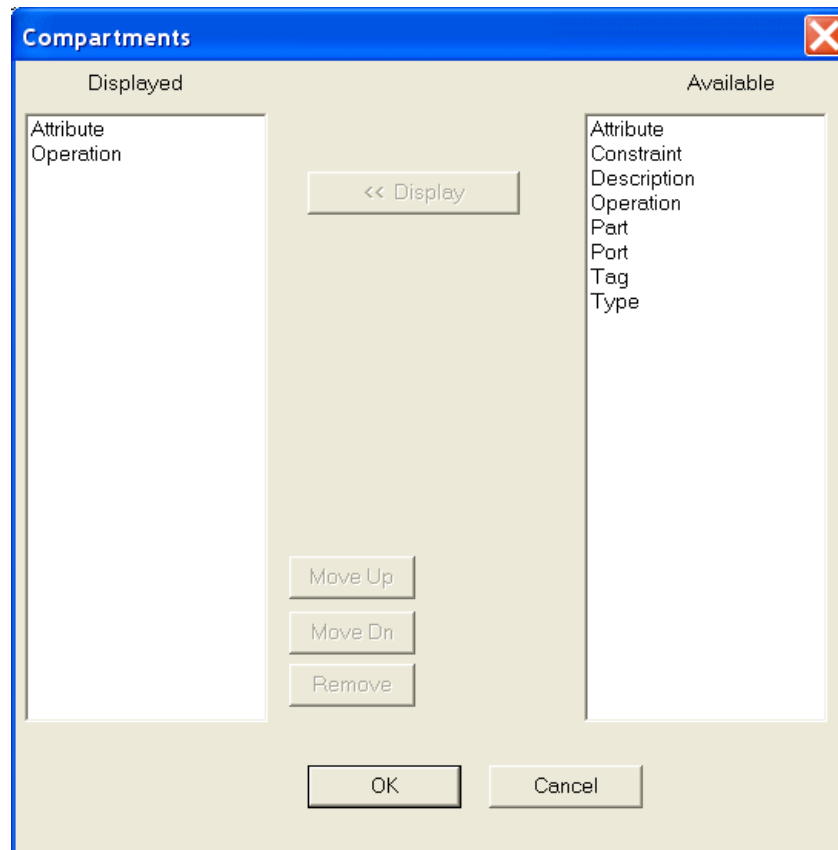
For example, the full path for a class *A* that belongs to the `Default` package would be displayed as follows:



- **Relative**—The relative name shows nesting of a class inside other classes, depending on its context within the diagram.

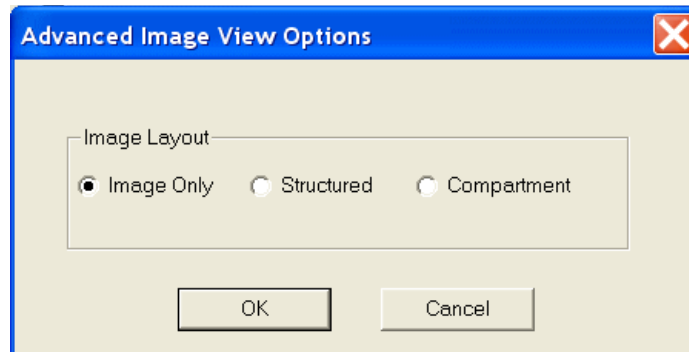
For example, if class *A* contains class *B*, then inside of *A*, *B*'s relative name is *B*, but outside of *A*, *B*'s relative name is *A* : *B*.

- **Name only**—This option displays only the class name without any path information.
- **Label**—This option displays the label for the class.
- ◆ **Show Stereotype Label** indicates whether or not to display the class stereotype as text at the bottom of the class box between guillemet symbols (for example, «Interface»).
- ◆ **Show Compartment Label** indicates whether or not to display the labels of available compartments. If you check the box, you then select the compartment for which you want the labels displayed from the list in the **Available** column of this dialog box. Highlight your selections and click the **Display** button to move them into the **Displayed** column.



- ◆ **Image View**—Specifies how the image is displayed. Check the box “Enable Image View” and either of the following options:
 - **Use Associated Image.** This option uses the default Rhapsody provided image for the object selected in Rhapsody.
 - **Select An Image.** If this option is selected, Rhapsody displays the **Image File Path** for you to locate it on your computer and click **OK**.

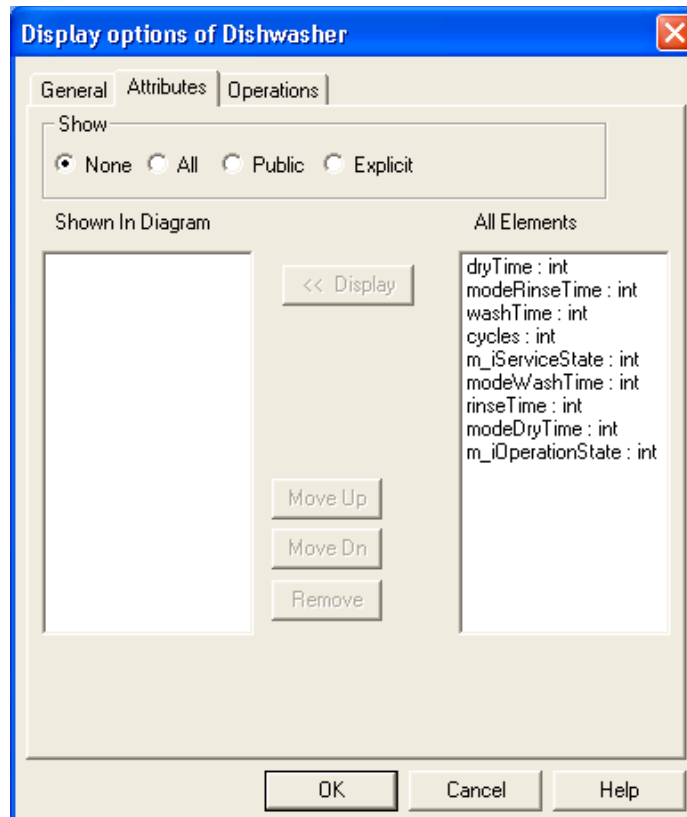
- ◆ **Advanced.** Clicking the Advanced option displays a dialog box with these three options:
 - Select **Image Only** to display the just the image.
 - Select **Structured** to see the picture in a separate compartment below the object's name compartment.
 - Select **Compartment** to see the picture in a separate compartment of its own between the object name compartment and bottom compartment.



- ◆ **Ports**—Specifies whether to show new ports and their interfaces. Rhapsody allows you to specify which ports to display in the diagram using the **Show New Ports** functionality. See [Ports](#) for more information.

Displaying Attributes and Operations

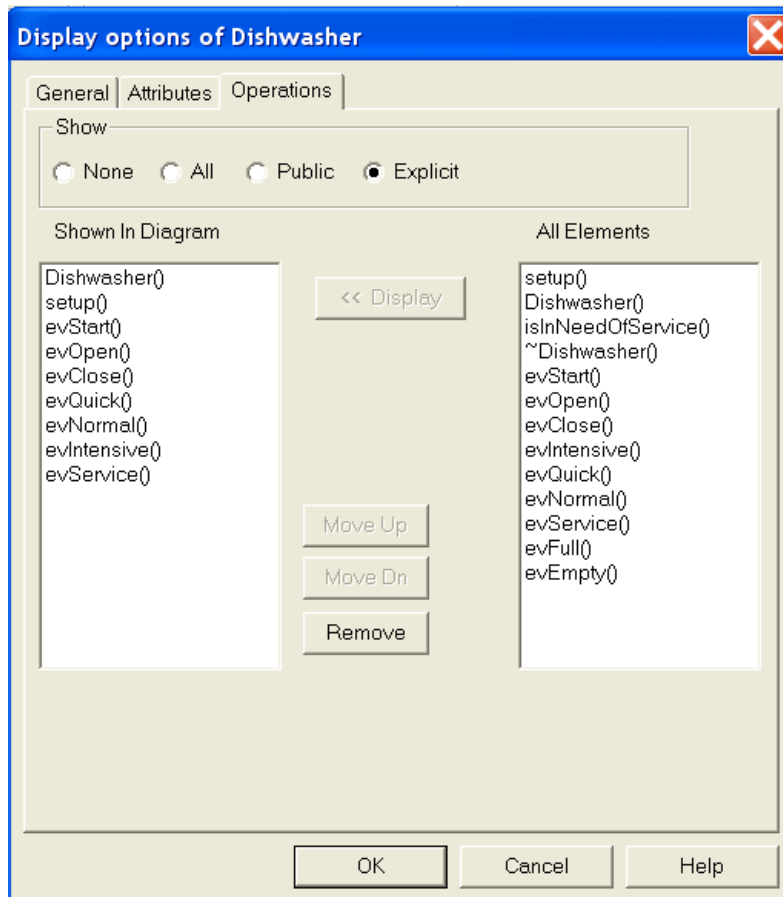
The **Attributes** tab in the Display options dialog box enables you to select which, if any, attributes to display in the diagram. The following figure shows the **Attributes** tab for the `Dishwasher` class.



To specify which elements to displayed in the diagram, follow these steps:

1. Highlight the element in the **All Elements** list.
2. Click the **Display** button to move the element to the **Shown in Diagram** list.
3. Repeat for each element or simply click the **All** button to select all of the elements in the list and display them.
4. You may move elements up and down in the **Shown In Diagram** list or remove them using the other three buttons in this dialog box.
5. Click **OK** to save your selections.

Similarly, the **Operations** tab allows you to select which, if any, operations to display in the diagram.



Removing or Deleting a Class

You can remove a class from the current view (diagram) or delete the class entirely from the model. Removing a class from the view does *not* delete it from the model.

To remove a class from the diagram, right-click the class and select **Remove from View** from the pop-up menu.

To delete the class entirely from the model, do one the following:

- ◆ Right-click the class, then select **Delete from Model** from the pop-up menu.
- ◆ Select the class, then click **Delete** in the main toolbar.

Note

When you delete a class, all of its objects are also deleted.

Ports

A *port* is a distinct interaction point between a class and its environment or between (the behavior of) a class and its internal parts. A port allows you to specify classes that are independent of the environment in which they are embedded. The internal parts of the class can be completely isolated from the environment and vice versa.

A port can have the following interfaces:

- ◆ **Required interfaces**—Characterize the requests that can be made from the port's class (via the port) to its environment (external objects). A required interface is denoted by a socket notation.
- ◆ **Provided interfaces**—Characterize the requests that could be made from the environment to the class via the port. A provided interface is denoted by a lollipop notation.

These interfaces are specified using a *contract*, which by itself is a provided interface. See [The Contract Tab](#) for more information.

If a port is *behavioral*, the messages of the provided interface are forwarded to the owner class; if it is *non-behavioral*, the messages are sent to one of the internal parts of the class. Classes can distinguish between events of the same type if they are received from different ports.

Note

Refer to the *HomeAlarmwithPorts* example (under `Rhapsody/Samples`) for an example of a model that uses ports.

Partial Specification of Ports

If you specify ports without any contract (for example, an implicit contract with no provided and required interfaces), Rhapsody assumes that the port relays events using the code generator. You could link two such ports and the objects would be able to exchange events via these ports.

However, Rhapsody will notify you during code generation (with warnings or informational messages) because the specification is still incomplete.

Considerations

Ports are interaction points through which objects can send or receive messages (primitive operations, triggered operations, and events).

Ports in UML have a type, which in Rhapsody is called a *contract*. A contract of a port is like a class for an object.

If a port has a contract (for example, interface \mathbb{I}), the port provides \mathbb{I} by definition. If you want the port to provide an additional interface (for example, interface \mathbb{J}), then, according to UML, \mathbb{I} must inherit \mathbb{J} (because a port can have only one type). In the case of Rhapsody, this inheritance is created automatically once you add \mathbb{J} to the list of provided interfaces (again, this is a port with an explicit contract \mathbb{I}). According to the UML standard, if \mathbb{I} and \mathbb{J} are unrelated, you must specify a new interface to be the contract and have this interface inherit both \mathbb{I} and \mathbb{J} .

Implicit Port Contracts

Some found that enforcing a specification of a special interface as the port's contract to be artificial, so Rhapsody provides the notion for an *implicit contract*. This means that if the contract is implicit, you can specify a *list* of provided and required interfaces that are not related to each other, whereas the contract interface remains implicit (no need to explicitly define a special interface to be the port's contract in the model).

Working with implicit contracts has pros and cons. If the port is connected to other ports that provide and require only subsets of its provided and required interfaces, it is more natural to work with implicit contracts. However, if the port is connected to another port that is exactly "reversed" (see the check box in the port's Features dialog box) or if other ports provide and require the same set of interfaces, it makes sense to work with explicit contracts. This is similar to specifying objects separately from the classes, or objects with implicit classes in the case when only a single object of this type or class exists in the system.

Rapid Ports


Rapid ports are ports that have no provided and required interfaces (which means that the contract is implicit, because a port with an explicit contract, by definition, provides a contract interface). These ports relay any events that come through them. The notion of rapid ports is Rhapsody-

specific, and enables users to do rapid prototyping using ports. This functionality is especially beneficial to users who specify behavior using statecharts—without the need to elaborate the contract at the early stages of the analysis or design.

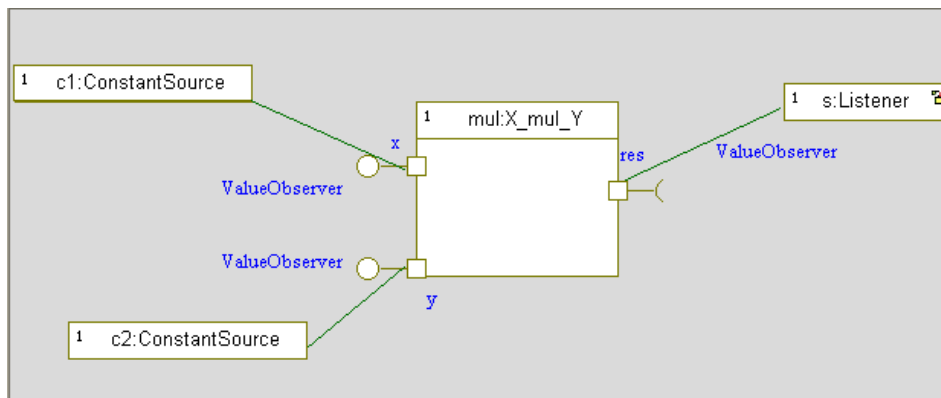
See [Using Rapid Ports](#) for more information on rapid ports.

Creating a Port

To create a port in an object model diagram, follow these steps:

1. Click the **Create port** icon .
2. Click the class boundary to place the port. A text box opens so you can name the new port.
3. Type the name for the port, then press **Enter** to dismiss the box. Note that the port label uses the convention `portName{ [multiplicity] }`. For example:
 - a. `p`
 - b. `p[5]`
 - c. `p[*]`

The new port appears as a small square on the boundary of its class, as shown in the following figure.



Alternatively, you can create a port in the following ways:

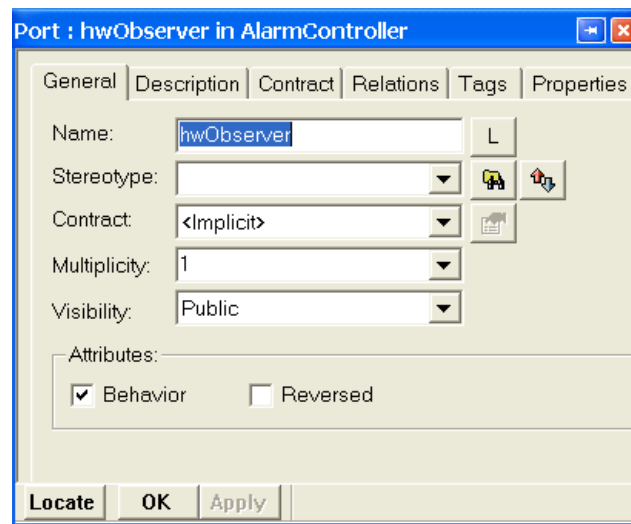
- ◆ Use the **Ports** tab of the Features dialog box for the class. See [Defining Class Ports](#) for more information.
- ◆ Right-click the class in the browser, then select **Add New > Port** from the pop-up menu.

Specifying the Features of a Port



As with all elements, you use the Features dialog box to specify the features that define a port. The Features dialog box for a port includes five tabs: **General**, **Contract**, **Relations**, **Tags**, and **Properties**.

The General Tab

The **General** tab defines basic information for the port, as shown in the following figure.



On the **General** tab, you define the general features for a port through the various controls on the tab.

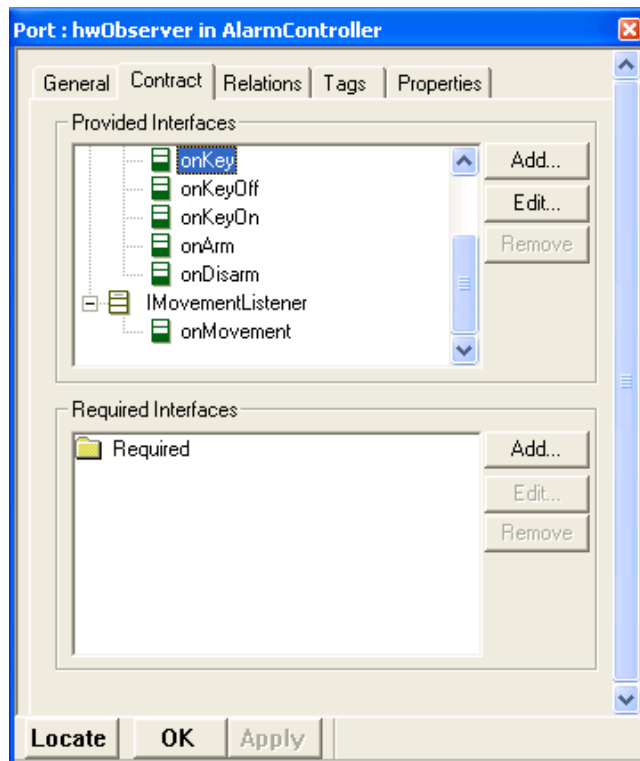
- ◆ In the **Name** box you specify the name of the port. The default name is `port_n`, where *n* is an incremental integer starting with 0. To enter a detailed description of the attribute, use the **Description** tab.
- ◆ You use the **L** button to open the Name and Label dialog box to specify the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ In the **Stereotype** drop-down list box you specify the stereotype of the port. See [Defining Stereotypes](#) for information on creating stereotypes.
 - To select from a list of current stereotypes in the project, click the Select Stereotype icon .
 - To sort the order of the selected stereotypes, click the Change Stereotype Order icon .

Note: The COM stereotypes are constructive; that is, they affect code generation. Refer to the *Rhapsody COM Development Guide* for more information about these components.

- ◆ In the **Contract** drop-down list box you specify the port contact. The drop-down list contains the following possible values:
 - <**Implicit**> means the contract is implicit.
 - <**New**> enables you to define a new contract. If you select this value, Rhapsody displays a separate class Features dialog box so you can define the new interface.
 - A list of classes with a stereotype that includes the word “interface.”The arrow button next to the **Contract** box opens the Features dialog box for the contract. However, this button is disabled if the contract is implicit.
- ◆ In the **Multiplicity** drop-down list box you specify the multiplicity of the port. The multiplicity is included in the port label if it is greater than 1.
- ◆ In the **Visibility** drop-down list box you specify the port’s visibility (**Public**, **Protected**, or **Private**). The default value is **Public**.
- ◆ In the **Attributes** area you specify the port attributes:
 - If you select the **Behavior** check box, messages sent to the port are relayed to the owner class. By default, the check box is cleared.
 - If you select the **Reversed** check box, the provided interfaces become the required interfaces, and the required interfaces become the provided interfaces.

The Contract Tab

The **Contract** tab enables you to specify the port contract, as shown in the following figure. The contract specifies the provided and required interfaces through relations to other interfaces.



There are two types of contract:

- ◆ **Explicit** means the contract is an explicit interface in the model. An explicit contract can be reused so several ports can have the same contract.
- ◆ **Implicit** means the contract is a “hidden” interface that exists only as the contract of the port.

For both provided and required interfaces, three buttons are available:

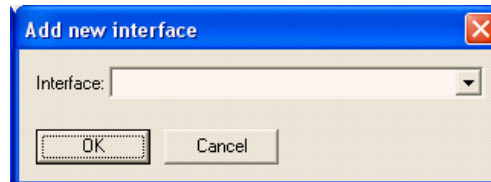
- ◆ **Add** to add a new interface to the list of available interfaces. For provided interfaces, this means that the contract inherits the selected interface; for required interfaces, this means that the contract has a new dependency stereotyped «Usage» towards the interface.
- ◆ **Edit** to open the Features dialog box for the selected element so you can modify it.
- ◆ **Remove** to remove the relation with the contract for the selected interface.

Note that if you enabled the **Reversed** check box on the **General** tab, the bottom of the **Contract** tab displays a message in red stating that the contract is reversed.

Specifying the Port Contract

To specify the contract information for the port, follow these steps:

1. To specify the provided interfaces, select the `Provided` folder icon, then click the **Add** button in the top group box. The Add new interface dialog box opens, as shown in the following figure.



2. Either type in the new name of the interface, or use the drop-down list to specify the interface.
3. Click **OK** to apply your changes and close the dialog box.
4. You return to the **Contract** tab, which now lists the provided interface you just specified.
5. To specify the required interface, click the **Required** folder, then select **Add**. The Add New Interface dialog box opens.

6. Specify the required interface, then click **OK** to apply your changes and close the dialog box.

Note: If a provided interface (including the contract) has an association to another interface, the other interface is a required interface.

7. Click **OK** to dismiss the Features dialog box.

Note: If an interface provided by a port inherits from another interface, then by definition, the port also provides the base interface. This means that if you wish to remove the base interface from the contract, you must remove the generalization between the two interfaces. (Before removing such an interface, Rhapsody will notify you that the generalization will also be removed.)

Display Options for Ports

The owning class or object specifies whether ports and their interfaces are displayed. By default, new ports and their interfaces are displayed.

To disable these default settings, follow these steps:

1. In the diagram, right-click the class or object that owns the port.
2. Select **Display Options** from the pop-up menu.
3. Disable the **Show Ports Interfaces** or **Show New Ports** check box, as desired.
4. Click **OK** to apply your changes and close the dialog box.

You can specify how the port name and stereotype are displayed using the Display Options dialog box for the port itself.

See [Selecting Which Ports to Display in the Diagram](#) for more information on displaying ports.

The Tags Tab

The **Tags** tab lists the tags available for the port. See [Using Profiles](#) for detailed information on tags.

The Properties Tab

The **Properties** tab lets you set and edit the properties that affect your model. The definition of each property displays at the bottom of the Features dialog box when you highlight a property in the list.

The following table shows the properties (under the `ObjectModelGe` subject) that control how ports are displayed.

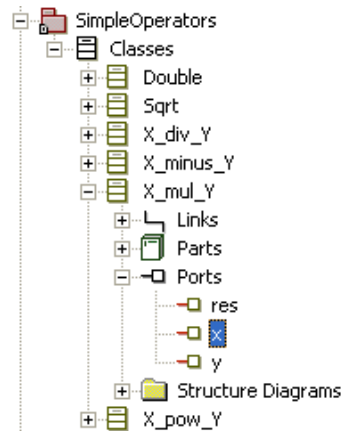
Metaclass	Property	Description
Class/Object	ShowPorts	Determines whether ports are displayed in OMDs
	ShowPortsInterfaces	Determines whether ports are displayed in OMDs
Port	color	Specifies the color used to draw the port
	FillColor	Specifies the default fill color for the port
	name_color	Specifies the default color of the port name
	UseFillColor	Specifies whether to use the fill color for the port

Note that you cannot selectively show ports in diagrams—either all the ports are displayed, or none of them are.

Refer to the *Rhapsody Properties Reference Manual* for explanations of how properties are grouped and used in Rhapsody. To examine the complete list of property definitions, refer to the *Rhapsody Property Definitions* PDF file available from the *List of Books*. This list can be searched along with the other PDF versions of the Rhapsody documentation to locate specific property definitions and the procedures that use them.

Viewing Ports in the Browser

Ports are displayed in the browser under the appropriate class, as shown in the following figure.



Connecting Ports

To exchange messages using ports, you must specify links between their objects. To do that, draw a link to the ports as described in [Links](#).

You can specify a link between a part (or a port of a part) to a port belonging to the enclosing class.

You cannot specify associations via ports. In addition, you cannot specify a link between classes, even if the ends are connected to ports.

Using Rapid Ports

Rhapsody supports *rapid ports*: you can simply draw ports, connect them via links, create a statechart, and the ports will exchange events without any additional information. In addition, if a port is not connected to any of its class's internal parts, the code generator assumes it is a behavioral port and messages will be relayed to or from the class. In rapid mode, the classes must be reactive because Rhapsody assumes that events are exchanged.

Rapid ports would be useful in the following situations:

- ◆ In component-based design. For example, when you have a class to be reused in different systems and has a behavior of its own (not that of one of its parts) that provides and requires the interfaces of the port's contract.
- ◆ The class has a statechart in which the triggers of the transitions are based on the ports through which the events were received. In other words, because the statechart is able to

distinguish between the ports through which an event was sent, it could react differently to the same events based on which port the event came from.

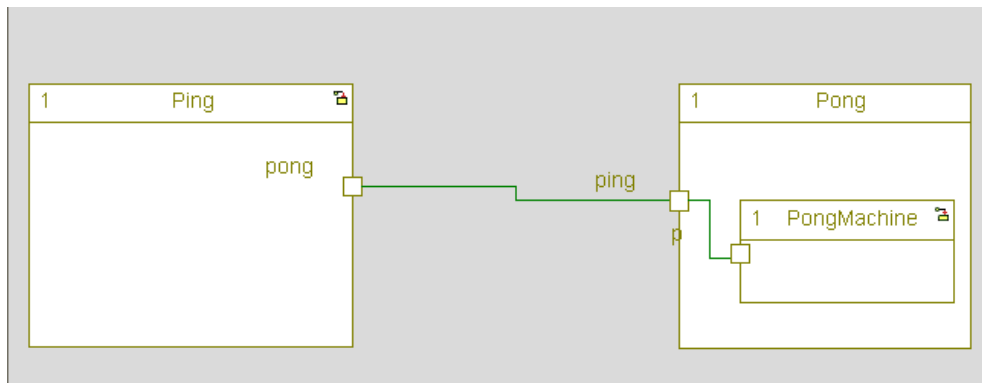
Note

Once you specify the contract on a port, you must specify the contract on all the ports that are connected to it. Otherwise, the code generator will issue a warning that there is a mismatch in the contracts and the links will not be created.

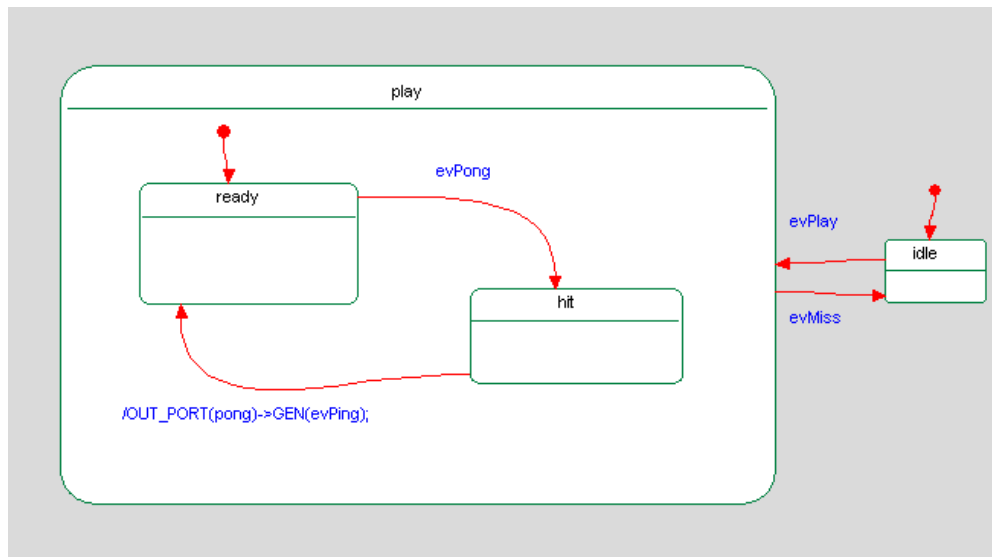
Rhapsody uses the values of the following framework properties to implement the rapid ports:

- ◆ `DefaultProvidedInterfaceName`—Specifies the interface that must be implemented by the “in” part of a rapid port.
- ◆ `DefaultReactivePortBase`—Stores the base class for the generic rapid port (or default reactive port). This base class relays all events
- ◆ `DefaultRequiredInterfaceName`—Specifies the interface that must be implemented by the “out” part of a rapid port
- ◆ `DefaultReactivePortIncludeFile`—Specifies the include files that are referenced in the generated file that implements the class with the rapid ports

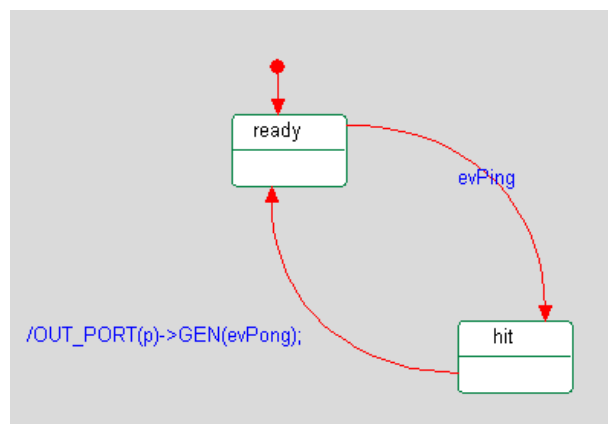
Consider the following figure, which shows an OMD that uses rapid ports.



The following figure shows the statechart for the `Ping` class.



The following figure shows the statechart for the `PongMachine` part.



During animation, the two objects exchange `evPing` and `evPong` events.

Selecting Which Ports to Display in the Diagram

If you right-click a class and select the **Ports** option from the pop-up menu, the following options are available:

- ◆ **New Port** creates a new port on the specified class.
- ◆ **Show All Ports** shows all the ports that currently exist in the specified model class.
- ◆ **Hide Ports** hides all the ports that are currently displayed by the specified model class. However, ports that are created later will be displayed.

Using this show/hide functionality in conjunction with the **Show New Ports** display option for the owning class, you can show and hide ports as desired to simplify your model. See [General Tab Display Options](#) for more information on display options for classes.

If you set **Show New Ports** mode to on, each new port that is added to the class is also displayed in the diagram class. Ports created before this graphic class, or while the **Show New Ports** feature is off, are *not* synthesized in the diagram—unless they were created using the graphic editor **New Port** option.

If **Show New Ports** mode is off, any ports created after disabling this mode will not be displayed.

Showing all Ports

To show all the ports in the diagram, enable the **Show New Ports** option in the Display Options dialog box for the owning class, and select **Ports > Show All Ports** from the pop-up menu for the class.

Showing New Ports Only

To show only new ports in the diagram, enable the **Show New Ports** option in the Display Options dialog box for the owning class, and select **Ports > Hide Ports** from the pop-up menu for the class.

Hiding All Ports

To hide all ports in the diagram, disable the **Show New Ports** option in the Display Options dialog box for the owning class, and select **Ports > Hide Ports** from the pop-up menu for the class.

Deleting a Port

To remove a specific port from a class, use the **Delete from Model** or **Remove from View** option in the pop-up menu for the owning class.

Programming with the Port APIs - RiC++

The following sections describe the APIs you use to program using ports. The topics are as follows:

- ◆ [Basic API Tasks](#)
- ◆ [Intermediate-Level Tasks](#)
- ◆ [Advanced-Level Tasks](#)

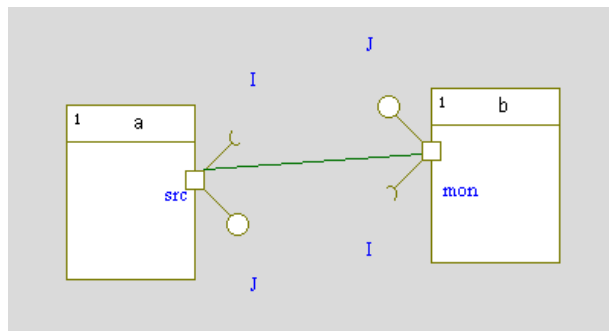
Basic API Tasks

This section describes the basic APIs used to exchange messages with and instantiate ports.

Note

The following example is not complete; it is simply a reference for the subsequent table of API calls.

Consider the following example:



The following table shows the calls to use to perform the specified tasks.

Task	Call
Call an operation.	<code>OUT_PORT(src)->f();</code>
Send an event from a to b using the ports.	<code>OUT_PORT(src)->GEN(evt);</code>
Listen for an event from port <code>src</code> to port <code>mon</code> .	<code>evt[IS_PORT(mon)]/ doYourStuff();</code>

You could also use the `OPORT` macro, which is equivalent to `OUT_PORT`.

Communicating with Ports with Multiplicity

The following table shows the calls to use if the multiplicity of the ports is 10 and you want to communicate with the ports using index 5.

Task	Call
Call an operation.	<code>OUT_PORT_AT(src, 5)->f();</code>
Send an event from a to b using the ports.	<code>OUT_PORT_AT(src, 5)->GEN(evt);</code>
Listen for an event from port <code>src</code> to port <code>mon</code> .	<code>evt[IS_PORT_AT(mon, 5)]/ doYourStuff();</code>

You could also use the `OPORT_AT` macro, which is equivalent to `OUT_PORT_AT`.

Intermediate-Level Tasks

This section describes the intermediate-level APIs used when programming with ports. You use these APIs whenever the code generator cannot create the links on its own, including the following cases:

- ◆ An external source file is used to initialize the system and you must write the code to create the links between the objects.
- ◆ The port multiplicities are not definite (for example, *).
- ◆ The port multiplicities do not match across the links. This could happen when ranges of multiplicities are used (for example, 1..10).

Connecting Objects Via Ports

If you are using an external application (such as the MFC GUI) where the links are created at run time, you can link objects with ports specified by Rhapsody using calls similar to the following:

```
a.getSrc()->setItsJ(b.getMon()->getItsJ());  
b.getMon()->setItsI(a.getSrc()->getItsI());
```

To link the objects, follow these steps:

1. Create a temporary package that creates the links for you.
2. Copy the .cpp file for the new package to the correct class.
3. Modify the code as needed.

Linking Objects Via Ports with Multiplicity

Using the example in the previous API illustration if the multiplicity of both ports is 10, you would link the objects as follows:

```
for (int i=0; i<10; ++i) {  
    a.getSrcAt(i)->setItsJ(b.getMonAt(i)->getItsJ());  
    b.getMonAt(i)->setItsI(a.getSrcAt(i)->getItsI());  
}
```

Advanced-Level Tasks

This section describes the advanced-level APIs used when programming with ports. You use these APIs when the code generator cannot determine how to instantiate the ports. This situation occurs when the port multiplicity is `*`.

Creating Ports Programmatically

By default, ports are created by value. However, if at design time you do not know how many ports there will be (multiplicity of `*`), you can create the ports programmatically.

For example, to instantiate 10 of the `src` ports, use the following call:

```
for (int i=0; i<10; ++i) {
    // instantiate and add to the container of the owner
    newSrc();
}
```

Linking Behavioral Ports to Their Owning Instance

Similarly, if you do not know what the multiplicity of the behavioral port at design time, you can specify it programmatically.

Behavioral ports are connected to their owning instance using the method `connect [ClassName]`. For example, to connect behavioral port `p` to its owner object `a` (of type `A`), use the following call:

```
a.getP()->connectA(a);
```

If the ports in previous API illustration are behavioral, you would use the following code:

```
for (int i=0; i<10; ++i) {
    newSrc();
    //hooks the class so it takes care of the messages
    getSrcAt(i)->connectA(this);
}
```

For more efficiency, use the following code:

```
for (int i=0; i<10; ++i) {
    newSrc()->connectA(this);
}
```

Port Code Generation - RiC

In RiC, code can be generated for the following types of ports:

- ◆ Rapid ports (see [Using Rapid Ports](#))
- ◆ Standard ports where the provided and required interfaces contain only event receptions

Action Language for Sending Events - RiC

The following macros are used for working with ports and events:

- ◆ Generating and sending an event via a port:

– `riCGEN_PORT([pointer to port], [event])`

Examples:

```
riCGEN_PORT (me->myPort, myEvent ())
```

For ports with multiplicity greater than one:

```
riCGEN_PORT (me->myPort [2], myEvent ())
```

- ◆ Detecting the input port through which an event has been sent:

– `riCIS_PORT([pointer to port])`

Examples:

```
riCIS_PORT (me->myPort)
```

This returns True if the event currently being handled by the RiCReactive (instantiated as me) was sent via the port myPort.

For ports with multiplicity greater than one:

```
riCIS_PORT (me->myPort [2])
```

Port Code Generation - Java

The following operations are used for working with ports and events in Java:

- ◆ Calling an operation:

– for port called MyPort and operation called foo:

```
getMyPort ().foo ();
```

– for port called MyPort, operation called foo, and multiplicity greater than 1:

```
getMyPortAt(port index).foo(), for example,  
getMyPortAt (2) .foo ();
```

- ◆ Generating and sending an event via a port:

– for port called MyPort and event called evt:

```
getMyPort ().gen (new evt ());
```

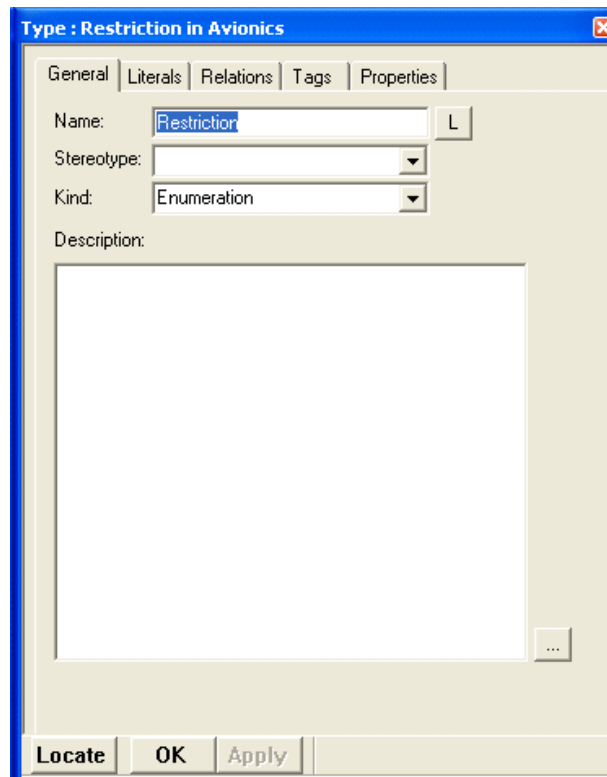
- for port called MyPort, event called e2, and multiplicity greater than 1:
getMyPortAt(*port index*).gen(new e2()), for example,
getMyPortAt(2).gen(new e2());
- ◆ Detecting the input port through which an event has been sent:
 - for port called MyPort:
isPort(getMyPort())
 - for port called MyPort, and multiplicity greater than 1: isPort(getMyPort(*port index*)), for example,
isPort(getMyPort(3))

Composite Types

Rhapsody enables you to create composite types that are modeled using structural features instead of verbatim, language-specific text. In addition, Rhapsody includes classes wherever types are used to increase the maintainability of models: if you change the name of a class, the change is propagated automatically throughout all of the references to it.

To create a composite type, follow these steps:

1. Right-click a package or the `Types` category, then select **Add New > Type**.
2. Edit the default name for the type.
3. Invoke the Features dialog box for the new type. The Type dialog box opens, as shown in the following figure.



4. If desired, specify a stereotype for the type.
5. Specify the kind of data type using the **Kind** drop-down list. The possible values are as follows:

- a. **Enumeration**—The new type is an enumerated type. Specify the enumerated values on the **Literals** tab. See [Creating Enumerated Types](#) for more information.
- b. **Language**—The new type is a language-specific construct. This is the default value. See [Creating Language Types](#) for more information.
- c. **Structure**—The new type is a structure, which is a data record. See [Creating Structures](#) for more information.
- d. **Typedef**—The new type is a typedef. See [Creating Typedefs](#) for more information.
- e. **Union**—The new type is a union, which is an overlay definition of a data record. See [Creating Unions](#) for more information.

Refer to the appropriate data type to continue the creation process.

The following table shows the mapping of composite types to the different languages.

Type Kind	Ada	C and C++	Java
Language	As in previous versions	As in previous versions	As in previous versions
Structure	Not supported	<code>struct</code> ¹	N/A
Union	Not supported	<code>union</code>	N/A
Enumeration	Enumeration types	<code>enum</code>	N/A
Typedef	Subtypes (in simple cases) or subtype	<code>typedef</code>	N/A

1. The generated struct is a simple C-style struct that contains only public data members.

Code generation analyzes the types to automatically generate:

- ◆ Dependencies in the code (`#include`)
- ◆ Type descriptions
- ◆ Field descriptions

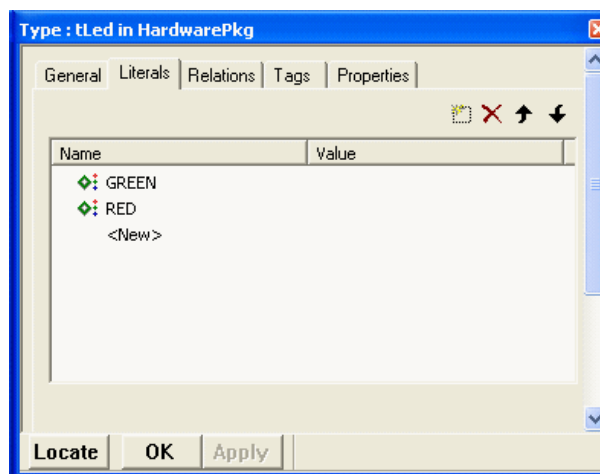
Each field in a structure and union has an attribute annotation.

Creating Enumerated Types

If you selected **Enumeration** as the **Kind**, continue the creation process as follows:

1. Select the **Literals** tab.
2. Select the **<New>** line, then type the name for the enumerated value.
3. Repeat for each value of the enumerated type.

The following figure shows values for an enumerated type.



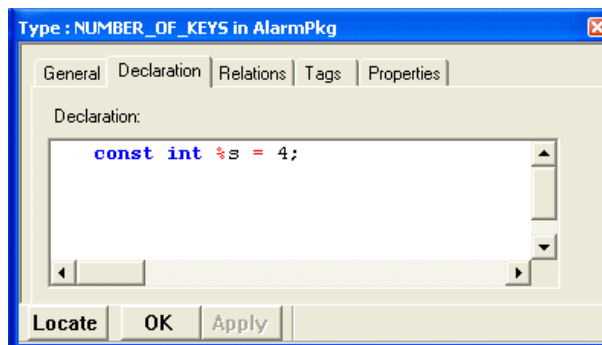
4. Click **OK** to apply your changes and close the dialog box.

Creating Language Types

If you selected **Language** as the **Kind**, continue the creation process as follows:

1. Select the **Declaration** tab.
2. Type the declaration statement in the **Declaration** text box. Use the expression `%s` as a placeholder for the type name in the declaration.

The following figure shows an example of a type of kind **Language**.



3. Click **OK** to apply your changes and close the dialog box.

Using %s

The Rhapsody code generator substitutes `%s` in type declarations with the type name. This automates the update of declarations when you rename a type.

To escape the `%s` characters, type a backslash (`\`) character before the `%s`. For example:

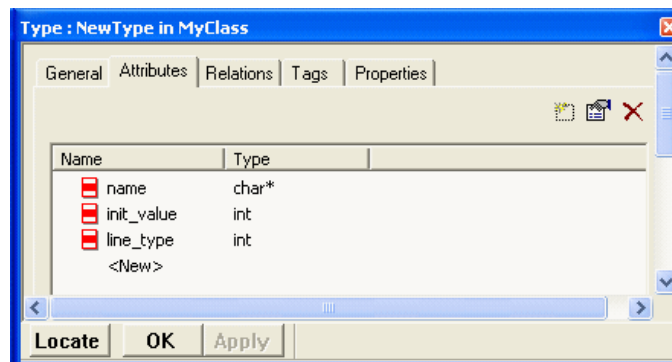
```
#define PRINT printf("\%s\n", myString())
```


Creating Structures

If you selected **Structure** as the **Kind**, continue the creation process as follows:

1. Select the **Attributes** tab.
2. Select the **<New>** line, then type the name for the member.
3. Use the **Type** drop-down list to select the type of the member. Note that the type can be another composite type.
4. Repeat Steps 2 and 3 for each structure member.

The following figure shows an example of a type of kind **Structure**.



5. Click **OK** to apply your changes and close the dialog box.

Note

Bit fields (for example, `int a :1`) are not supported. You can model them using language types. See [Creating Language Types](#) for more information.

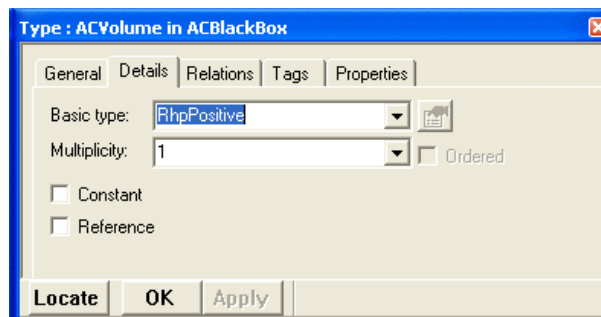
Creating Typedefs

If you selected **Typedef** as the **Kind**, continue the creation process as follows:

1. Select the **Details** tab.
2. Specify the typedef in the **Basic Type** box, or use the drop-down list to select the type. Note that the **Basic Type** *cannot* be an implicit type.
Note: If you select a type defined within the model, the arrow button next to the **Basic Type** box is enabled. Click the arrow button to invoke the Features dialog box for that class.
3. Specify the multiplicity in the **Multiplicity** box. The default value is 1.
Note: If the multiplicity is a value higher than 1, the **Ordered** check box is enabled. Click this check box if the order of the reference type items is significant.
4. If the typedef is defined as a constant (is read-only, such as the `const` qualifier in C++), enable the **Constant** check box; if the typedef is modifiable, leave the check box disabled (empty).
5. If the typedef is referenced as a reference (such as a pointer (*) or a C++ reference (&), enable the **Reference** check box.

The implementation of the reference is set by the property `<lang>_CG::Type::ReferenceImplementationPattern`. Refer to the definition of this property in the Features dialog box.

The following figure shows the **Details** tab.



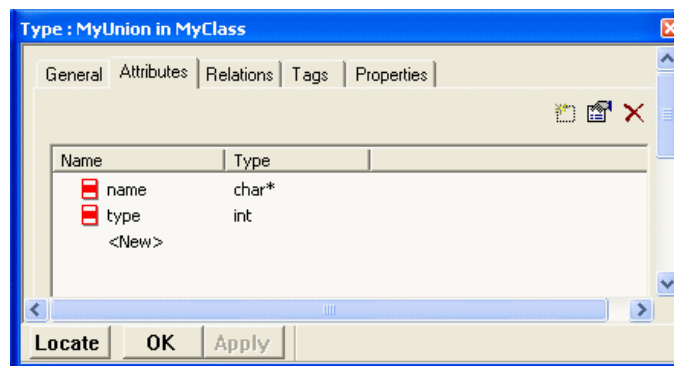
6. Click **OK** to apply your changes and close the dialog box.

Creating Unions

If you selected **Union** as the **Kind**, continue the creation process as follows:

1. Select the **Attributes** tab.
2. Select the **<New>** line, then type the name for the member.
3. Use the **Type** drop-down list to select the type of the member. Note that the type can be another composite type.

The following figure shows an example of a type of kind **Union**.



4. Click **OK** to apply your changes and close the dialog box.

Note

Ada variant record keys and conditions are not supported.

Properties

The following table lists the properties that support composite types.

Subject and Metaclass	Property	Description
CG subject		
Attribute/Type	Implementation	<p>The Implementation property enables you to specify how Rhapsody generates code for a given element (for example, as a simple array, collection, or list). (Default = Default)</p> <p>When this property is set to Default and the multiplicity is bounded (not *) and the type of the attribute is not a class, code is generated without using the container properties (as in previous versions of Rhapsody).</p> <p>Note that Rhapsody generates a single accessor and mutator for an attribute, as opposed to relations, which can have several accessors and mutators. In smart generation mode, a setter is not generated when the attribute is Constant and either:</p> <ul style="list-style-type: none"> • The attribute is not a Reference. • or The multiplicity of the attribute is 1. • or The <code>CG::Attribute::Implementation</code> property is set to <code>EmbeddedScalar</code> or <code>EmbeddedFixed</code>.
<ContainerType> subject		
<ImplementationType>	Various properties	Contain the keywords <code>\$constant</code> and <code>\$reference</code> to support the Constant and Reference modifiers
<ImplementationType>	FullTypeDefinition	Specifies the <code>typedef</code> implementation template
<lang>_CG subject		
Attribute	MutatorGenerate	Specifies whether mutators are generated for attributes
Attribute/Type	ReferenceImplementationPattern	Specifies how the Reference option is mapped to code
Class/Type	In	Specifies how code is generated when the type is used with an argument that has the modifier <code>In</code>

Subject and Metaclass	Property	Description
	InOut	Specifies how code is generated when the type is used with an argument that has the modifier InOut
	Out	Specifies how code is generated when the type is used with an argument that has the modifier Out
	ReturnType	Specifies how code is generated when the type is used as a return type
	TriggerArgument	Is used for mapping event and triggered operation arguments to code instead of the In, InOut, and Out properties
Type	EnumerationAsTypedef	Specifies whether the generated enum should be wrapped by a typedef. This property is applicable to enumeration types in C and C++.
	StructAsTypedef	Specifies whether the generated enum should be wrapped by a typedef. This property is applicable to structure types in C and C++.
	UnionAsTypedef	Specifies whether the generated union should be wrapped by a typedef. This property is applicable to union types in C and C++.

Language-Independent Types

Rhapsody enables you to build static models using language-independent, predefined types—with no dependency on the implementation language.

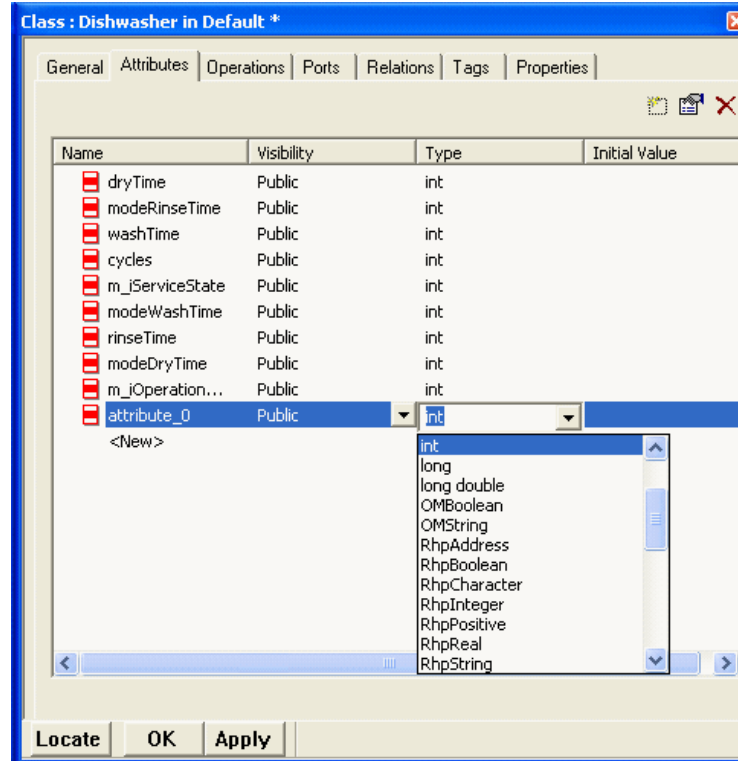
The types are defined in the following files (under `<Rhapsody>\Share\<lang>\oxf`):

- ◆ Ada—`RiA_Types` package
- ◆ C—`RiCTypes.h`
- ◆ C++—`rawtypes.h`
- ◆ Java—The types are converted during code generation, based on properties defined in the `PredefinedTypes` package loaded by Rhapsody (under `<Rhapsody>\Share\Properties\PredefinedTypes.sbs`).

The following table shows the mapping between the predefined types and the language implementation types.

Model Type	Ada	C	C++	Java
<code>RhpInteger</code>	<code>integer</code>	<code>int</code>	<code>int</code>	<code>int</code>
<code>RhpUnlimitedNatural</code>	<code>long_integer</code>	<code>long</code>	<code>long</code>	<code>long</code>
<code>RhpPositive</code>	<code>unsigned</code>	<code>unsigned int</code>	<code>unsigned int</code>	<code>int</code>
<code>RhpReal</code>	<code>long_float</code>	<code>double</code>	<code>double</code>	<code>double</code>
<code>RhpCharacter</code>	<code>character</code>	<code>char</code>	<code>char</code>	<code>char</code>
<code>RhpString</code>	<code>string</code>	<code>char*</code>	<code>OMString</code>	<code>String</code>
<code>RhpBoolean</code>	<code>boolean</code>	<code>RiCBoolean</code>	<code>bool</code>	<code>boolean</code>
<code>RhpVoid</code>	Used in procedure declaration only	<code>void</code>	<code>void</code>	<code>void</code>
<code>RhpAddress</code>	<code>address</code>	<code>void*</code>	<code>void*</code>	<code>Object</code>

When you create attributes or operations, these language-independent types are included in the **Types** drop-down list, as shown in the following figure.

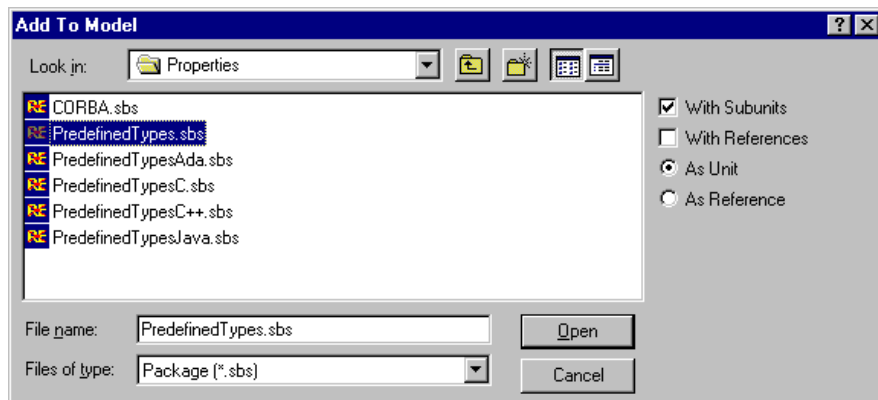


Changing the Type Mapping

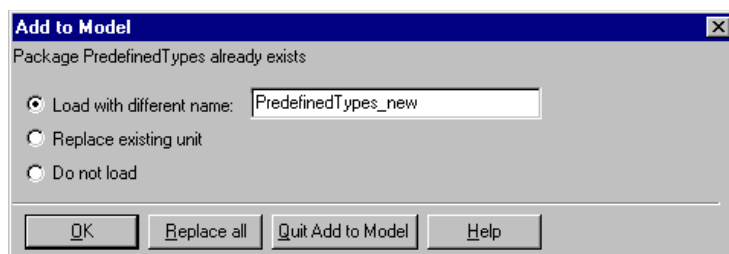
The file `PredefinedTypes.sbs` contains the set of predefined types included in Rhapsody. This file is opened automatically when you create a new model or open an existing one.

If you previously changed this file, you can merge your changes by following these steps:

1. Open the model or create a new one.
2. Select **File > Add to Model**. The Add To Model dialog box opens.
3. Navigate to the `<Rhapsody>\Share\Properties` directory.
4. Set the **Files of type** box to **Package (*.sbs)**.
5. Select the file `PredefinedTypes.sbs`, as shown on the following figure.



6. Click **Open**.
7. Because the package already exists in the model, a dialog box opens so you can add the package to the model under a new name, such as `PredefinedTypes_new`.



8. Click **OK**.
9. Change the property `<lang>_CG::Type::LanguageMap` to... TBS
10. Save the modified package.
11. Close the model.
12. Run DiffMerge on your original file and the new one, which is located in the `<project name>_rpy` directory.
13. Add your changes to the `.sbs` file located in the `Share\Properties` directory.

Note

The following behavior and restrictions apply to the language-dependent types.

- ◆ In Rhapsody in J, language-independent types are supported only as modeling constructs—you cannot use them in actions or operation bodies.
- ◆ This feature does not apply to COM and CORBA.

Changing the Order of Types in the Generated Code

Types are generated in code in the order in which they appear in the browser. This can be a problem if one type depends on another that is defined later.

For example, you can define a type `FirstType` as:

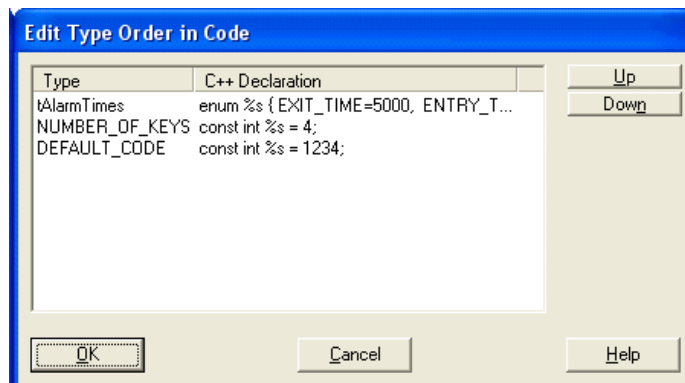
```
typedef SecondType %s
```

Next, define a type `SecondType` as:

```
typedef int %s
```

These two types defined in this order would result in a compilation error. To avoid this kind of error, you can control the order in which types are generated using the Edit Type Order in Code dialog box, which is accessible via the pop-up menu for the Types category for an individual package or a class. To edit the order of types, follow these steps:

1. Right-click the **Types** category icon (or a package or a class).
2. From the pop-up menu, select **Edit Type Order**. The Edit Type Order in Code dialog box opens, as shown in the following figure.



3. Select the type you want to move.
4. Click **Up** to generate the type earlier or **Down** to generate it later.
5. Click **OK** to apply your changes and close the dialog box.

Using Fixed-point Variables

For target systems that do not include floating-point capabilities, Rhapsody in C provides an option to use fixed-point variables.

This is done by scaling integer variables so that they can represent non-integral values. Rhapsody uses the *2 factorial* approach to achieve this. For example, setting the bit that usually represents 2^0 to represent 2^{-3} (.125).

For each such variable, the user specifies the word-size and the precision of the variable. The specific steps involved are described in [Defining Fixed-point Variables](#).

Defining Fixed-point Variables

The elements required for defining fixed-point variables are included in a profile called *FixedPoint*. This profile contains:

- ◆ Predefined types representing 8, 16, and 32-bit fixed-point variables: *FXP_8Bit_T*, *FXP_16Bit_T*, *FXP_32Bit_T*. (These are the only types that can be used with fixed-point operations.)
- ◆ A "new term" stereotype, applicable to attributes, called *Fixed-Point*, with a tag called *FXP_Shift* which is used to define the scale of the fixed-point variable.

The word-size is determined by the type chosen, while the shift to use is determined by the value entered for the tag *FXP_Shift*.

The profile uses a file called *FixedPoint.h*, which contains:

- ◆ Typedefs representing the predefined fixed-point variable types
- ◆ Macros that are used for carrying out operations on fixed-point variables.

The file is "included" into the generated code where fixed-point variables are generated.

To define a fixed-point variable:

1. Add the *FixedPoint* profile to your project as a reference.
2. In the browser, right-click the element that will contain the fixed-point variables.
3. When the context menu is displayed, select **Add New > FixedPoint > FixedPointVar**.
4. Name the new variable.
5. Open the features dialog for the new variable, and select one of the fixed-point variable types (*FXP_8Bit_T*, *FXP_16Bit_T*, or *FXP_32Bit_T*).

6. Set the shift to use by providing a value for the tag *FXP_Shift* (default value is 4). (The variable that was created already has the *Fixed-Point* stereotype applied to it.)

Operations Permitted for Fixed-point Variables

The following operations can be performed on fixed-point variables:

- ◆ Arithmetic: addition, subtraction, multiplication, division
- ◆ Assignment (=)
- ◆ Relational operators (<, >, <=, >=, ==, !=)

To carry out the above operations, you use the relevant macros that are contained in *FixedPoint.h*. For example to add fixed-point variables, use the macro *FXP_ASSIGN_SUM*. (Note that some of the macros in this file are macros that are called by the operation macros. These macros should not be called directly.)

Restrictions on Use of Fixed-point Variables

Keep the following points in mind when working with fixed-point variables:

- ◆ The supported operations can only be performed on fixed-point variables, and not on the result of fixed-point calculations. For example, the following is not permitted:
`FXP_ASSIGN_SUM(FXP_ASSIGN_SUM(varA,varB),varC)`.
- ◆ Operations can be performed on fixed-point variables only. If you try to use one of the operations with a combination of fixed-point and ordinary variables, compilation errors will result.
- ◆ The shift specified can range from 0 to (word size - 1). Rhapsody does not check that the shift you entered for the variable is within this range.
- ◆ When calling a function that takes a fixed-point variable as an argument, make sure that the variable provided to the function has the same fixed-point characteristics (word size and shift) as the defined argument.
- ◆ When calling a function that returns a fixed-point variable, make sure that the return value is being assigned to a variable that has the same fixed-point characteristics (word size and shift) as the defined return type.
- ◆ Programmers must take into account that operations on fixed-point variables can result in an arithmetic overflow.
- ◆ Programmers must take into account that operations on fixed-point variables can result in a loss of precision.

Fixed-point Conversion Macros

Rhapsody provides the following macros for converting to/from fixed-point variables:

- ◆ *FXP2INT(FPvalue, FPshift)* - Converts a fixed-point variable to an integer
- ◆ *FXP2DOUBLE(FPvalue, FPshift)* - Converts a fixed-point variable to a double
- ◆ *DOUBLE2FXP(Dvalue, FPshift)* - Converts a double to a fixed-point variable

These macros can be used in conjunction with the macros that require fixed-point variables as arguments, for example:

```
FXP_ASSIGN_EXT(myFixedPointVar, FXP_16Bit_T, 4, DOUBLE2FXP(3.5, 1), 1);
```

Java Enums

Rhapsody allows use to include Java enums (introduced in Java 5.0) in your models.

Adding a Java Enum to a Model

To add a Java enum to your model:

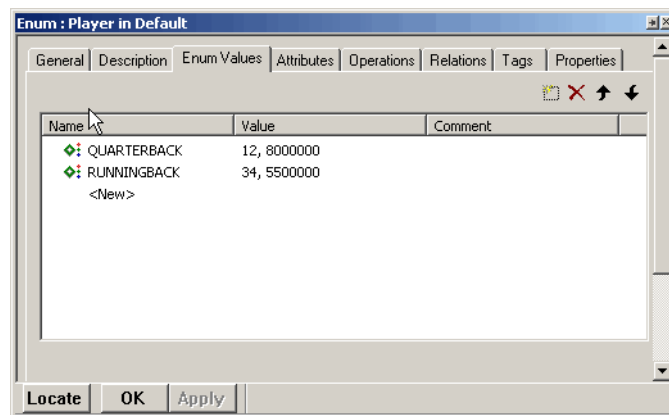
1. Select the package to which you would like to add the enum, and right-click to display the context menu.
2. From the context menu, select **Add New > Enum**.

Enums are displayed as their own category in the Rhapsody browser.

Defining Constants for a Java Enum

To define constants for an enum:

1. Double-click the relevant enum in the browser, or select Features from the context menu.
2. Select the **Enum Values** tab.



3. Click **<New>** in the list.
4. Click the Name column to change the default name assigned by Rhapsody.
5. Click the Value column to define the argument values that will be used to instantiate this instance of the enum (if you are providing more than one argument, separate them with commas).
6. Optionally, add comments for the constants you have defined.

7. Click **OK** to close the Features dialog box.

After they have been defined, enum constants appear underneath the relevant enum in the browser.

Note

Rhapsody does not support the definition of anonymous classes that extend enum constants.

Adding Java Enums to an OMD

To add a Java enum to an Object Model Diagram, drag the enum from the browser to the diagram.

Code Generation

Rhapsody generates Java code for the enums you have defined.

The reverse engineering feature does not support Java enums.

Creating Java Enums with the Rhapsody API

The following lines of code will add a new enum to the selected package, and define two constants for the new enum:

```
Dim p As RPPackage
Dim c As RPClass
Dim a1 As RPAttribute
Dim a2 As RPAttribute

Set p = getSelectedElement
Set c = p.addNewAggr("Enum", "SampleEnum")
Set a1 = c.addNewAggr("EnumValue", "FIRST_CONSTANT")
Set a2 = c.addNewAggr("EnumValue", "SECOND_CONSTANT")
```

Template Classes, Generic Classes

Rhapsody allows you to include generic design elements in your models. Specifically, you can:

- ◆ Create and use template classes (C++)
- ◆ Create and use generic classes (Java)
- ◆ Create template functions (C++)
- ◆ Create generic methods (Java)

The terminology used for these concepts differs slightly between C++ and Java. In this section, we will use the UML terms *template class* and *template operation* to represent the generic elements in both C++ and Java.

In general, the procedures described in this section apply to both C++ and Java. Where there are language-dependent differences, these differences are noted.

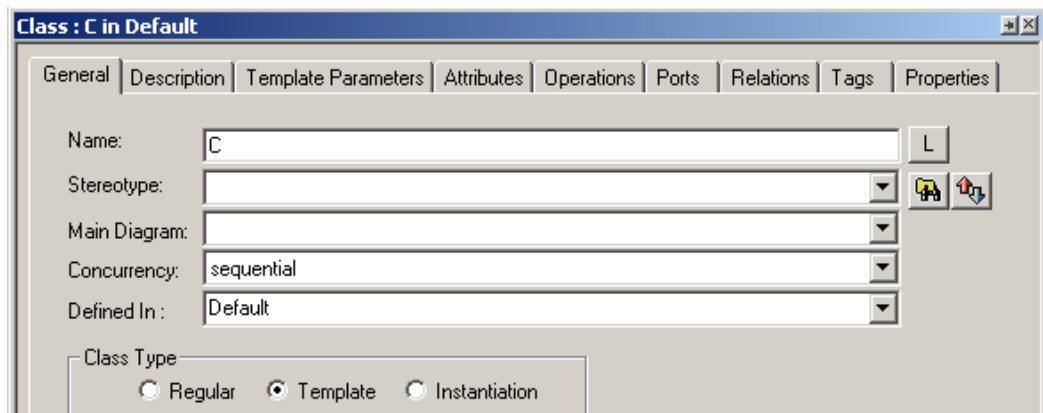
Creating a Template Class

You can use a class to create a template class. In addition, some template parameters may be specified as specific types and a specialized function to create a specialization or new class/function with content that is unrelated to the original template.

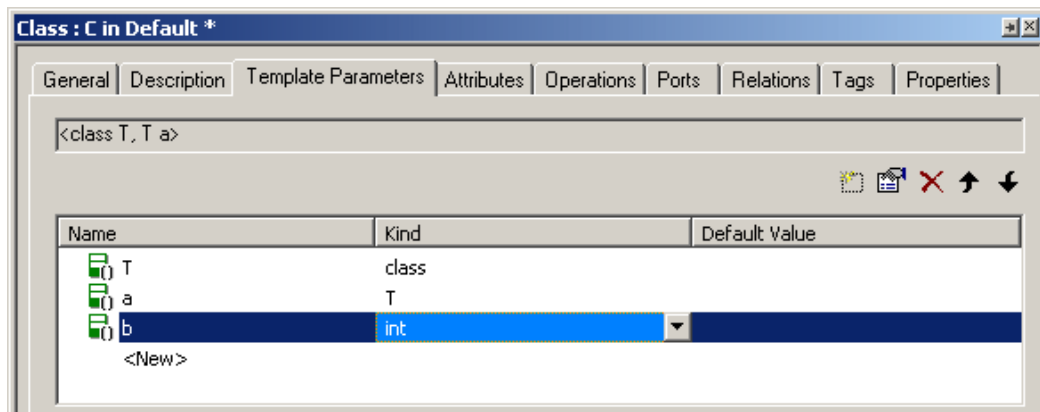
Note that you can use the DiffMerge tool to locate and merge template information.


To create a class template, follow these steps:

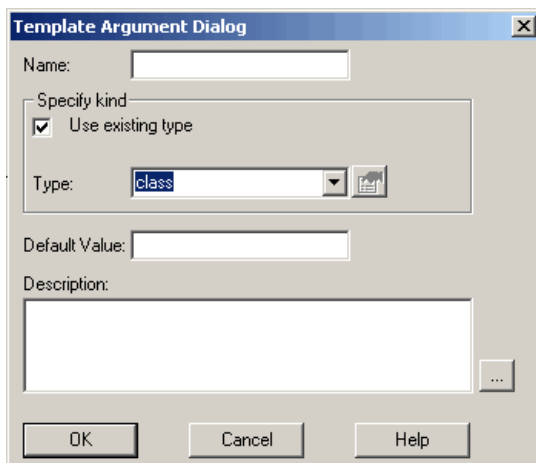
1. Double-click the class in the Rhapsody browser to open its Features dialog box.
2. On the **General** tab, in the **Class Type** area, select the **Template** radio button. Notice that the **Template Parameters** tab appears, as shown in the following figure.



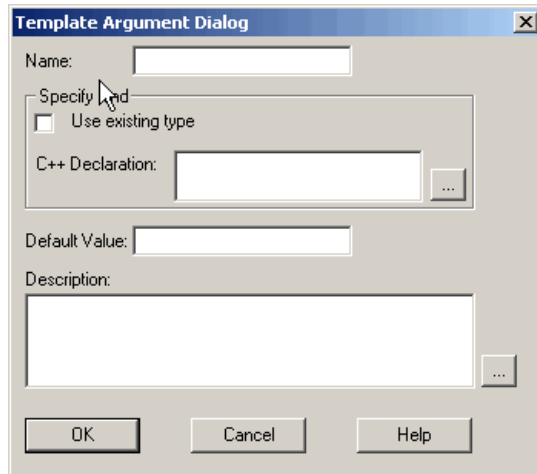
3. On the **Template Parameters** tab, click <New>.
4. Type a name to replace the default name that Rhapsody creates as <class_n>. See [Template Limitations](#) for guidelines for these names.
5. Accept the default type or select another one from the **Kind** drop-down list, as shown in the following figure.





6. To add arguments for the template, click the Invoke Feature Dialog icon  to open the Template Argument dialog box. Note the following for the Template Argument dialog box:
 - a. If you select the **Use existing type** check box, as shown in the following figure, you can change the type and enter a description. In C++, you can also provide a default value for the template argument.



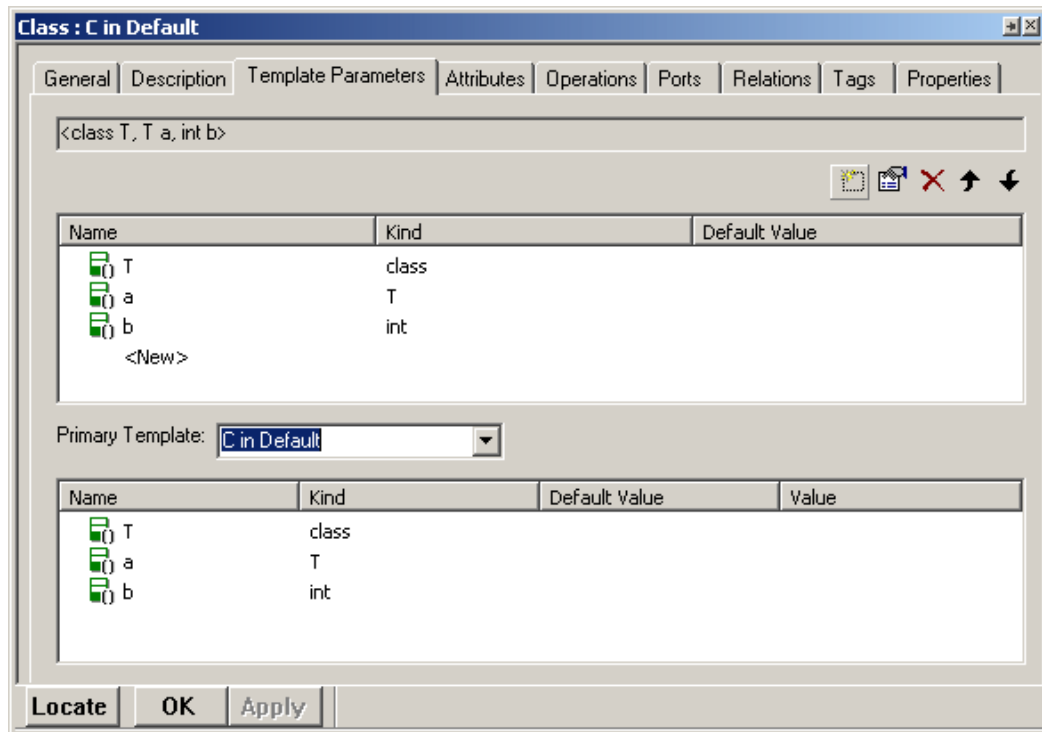
- b. If you clear the **Use existing type** check box, as shown in the following figure, you can enter code that further refines the argument type, for example a pointer to a type or an array of a certain type. When entering code in the **C++[Java] Declaration** box, you can also refer to other arguments that have been defined.



- c. Click **OK** to close the Template Argument dialog box and return to the **Template Parameters** tab.
7. Add more templates as needed by clicking **<New>** on the **Template Parameters** tab.
8. To determine the argument order on the **Template Parameters** tab, use the Move Item Up  and Move Item Down  icons.

9. If there is a primary template that you want to use, select it in the **Primary Template** drop-down list box. This box contains template(s) for which this class is a specialization. Its parameters to be instantiated appear in the box below the **Primary Template** drop-down list box, as shown in the following figure.

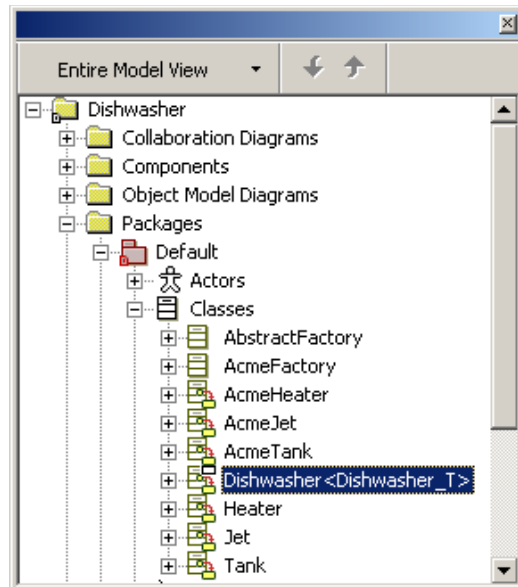
You can define specialization parameters only if you select a template as a primary class.



Note: When you try to delete a template that has specialization, Rhapsody warns you that the template has references. If you do delete the template, such specialization will generate an error when you check a model.

10. Click **OK**.

The template is listed in the browser in the **Classes** category, as shown in the following figure.



Once you have created the template class, you can begin using it directly in your code.

You can create templates in other situations. For example, you can:

- ◆ Re-use any type defined for a template parameter as a type within the template.
- ◆ Use the template class as a generalization, as described in [Using Template Classes as Generalizations](#).
- ◆ Create an operation template, as described in [Creating an Operations Template](#).
- ◆ Create a function template, as described in [Creating a Functions Template](#).

See also [Instantiating a Template Class](#).

Using Template Classes as Generalizations

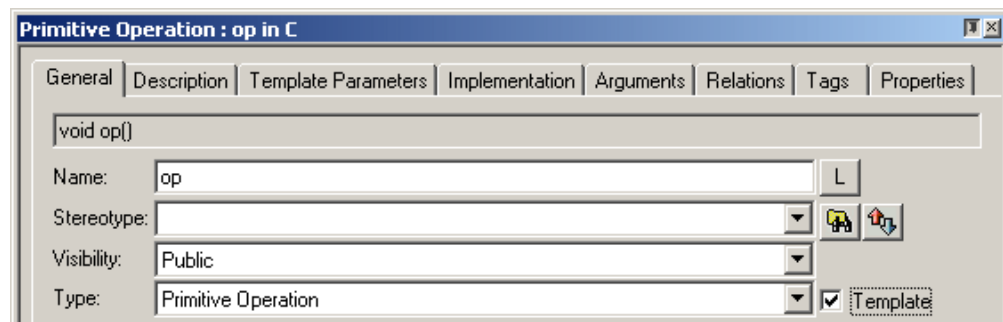
To use a template class as a generalization, follow these steps:

1. Create a class in an OMD, or in the Rhapsody browser.
2. Create the generalization by adding a super class in the browser or by drawing a generalization connector from the new class to the template class.
3. Open the Features dialog box of the generalization by using the context menu of the super class in the browser or the generalization connector in the OMD.
4. On the **Template Instantiation** tab of the Features dialog box, provide a value for each of the arguments listed by selecting an item from the **Value** drop-down list or entering a new value.

Creating an Operations Template

To create an operation template, follow these steps:

1. Create an operation in a class.
2. Open the Features dialog box for the operation.
3. On the **General** tab, in the **Class Type** area, select the **Template** check box, as shown in the following figure. Notice that the **Template Parameters** tab appears.



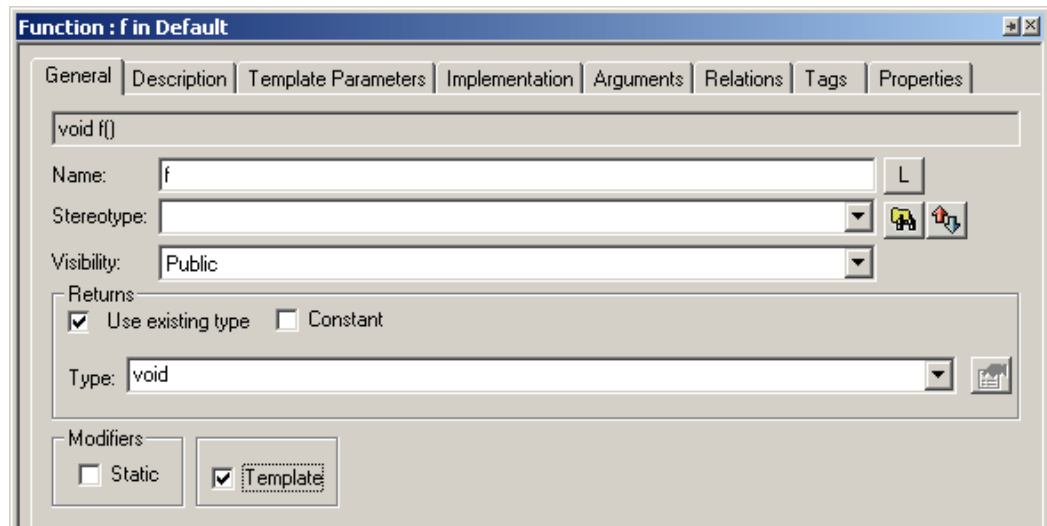
4. Set your template parameters for your operation on the **Template Parameters** tab. For detailed instructions, see [Creating a Template Class](#).

Once you have created the template operation you can begin using it in your code.

Creating a Functions Template

To create a functions template, follow these steps:

1. Create a function and open its Features dialog box.
2. On the **General** tab, select the **Template** check box, as shown in the following figure. Notice that the **Template Parameters** tab appears.



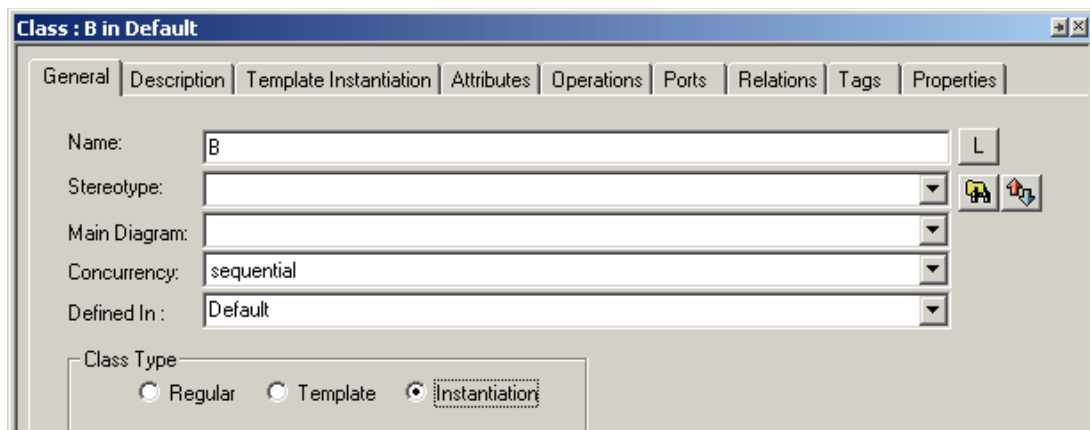
3. Set your template parameters for your function on the **Template Parameters** tab. For detailed instructions, see [Creating a Template Class](#).

Once you have created the template operation you can begin using it in your code.

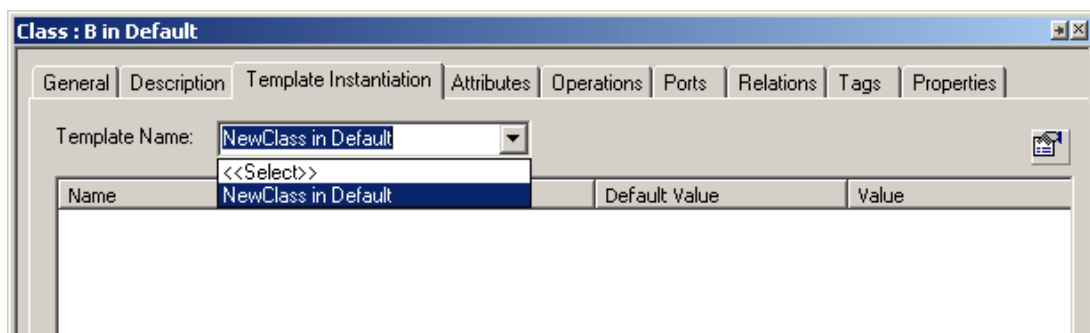
Instantiating a Template Class

To instantiate a template class, follow these steps:

1. Create a class in an OMD, or in the Rhapsody browser.
2. Open the Features dialog box for the class.
3. On the **General** tab, in the **Class Type** area, select the **Instantiation** radio button. Notice that the **Template Instantiation** tab appears, as shown in the following figure.



4. On the **Template Instantiation** tab, select a template from the **Template Name** drop-down list box, as shown in the following figure.



5. To view/modify the parameters for a template, double-click the template name or click the Invoke Feature Dialog icon to open the Template Instantiation Argument Dialog box. Click **OK** to return to the **Template Instantiation** tab.

When code is generated, the template instantiation is represented by a typedef in C++ and by a class in Java.

Code Generation and Templates

The creation of templates and specializations are supported in code generation. If both the template and its specialization are in the same package, they are generated into the same file. In the file, the template is generated before its specialization to ensure that the code runs as expected. A check is added to warn that the template and template specialization are in different packages.

Note

If a nested class or attribute is marked as a template parameter, it is not generated.

Template Limitations

The following are the limitations for templates:

- ◆ If there is a template parameter named “T” in an operation/function, the user cannot assign a class named “T” to the owner of the operation/function.
- ◆ If there are more than one operation/function with a template parameter of the same name under the same owner, renaming the created nested class renames all of the parameters with this name.
- ◆ For templates in a Java project, the following are additional limitations:
 - Wildcards are not supported.
 - Bounded wildcards are not supported.
 - Generic methods are not supported.

Eclipse Platform Integration

This section describes the **Rhapsody Platform Integration**, which lets software developers work on a Rhapsody project within the Eclipse platform. This integration is currently available only for C, C++, or Java development in a Windows environment.

If you want to work in the Rhapsody interface and use some Eclipse features, you can use the **Workflow Integration**, which is the other plug-in implementation. In this integration, software developers work in the Rhapsody product and invoke Rhapsody menu options to use some Eclipse features. You can also navigate between the two environments. This integration can be used for C, C++, and Java development in either Windows or Linux environments. Both Eclipse and Rhapsody must be open when you are using this integration. See [Working with Projects](#) for information about this implementation.

Note

Refer to the *Rhapsody Installation Guide* for Eclipse-specific installation and set-up instructions.

Platform Integration Prerequisites

The following software needs to be installed to create a fully functioning Eclipse platform integration with Rhapsody:

- ◆ Eclipse Europa
- ◆ CDT plug-in for C and C++ application development
- ◆ JDT plug-in for Java development
- ◆ Compilers required for the development language or languages you are using
- ◆ Rhapsody Developer Edition Multi-Language

The Platform Integration of Rhapsody for Eclipse requires a Multi-Language Rhapsody license.

Note

The stand-alone version of Rhapsody and the Rhapsody Platform Integration within Eclipse both use the same repository so that you can switch between the two interfaces if you want.

Confirming Your Rhapsody Platform Integration within Eclipse

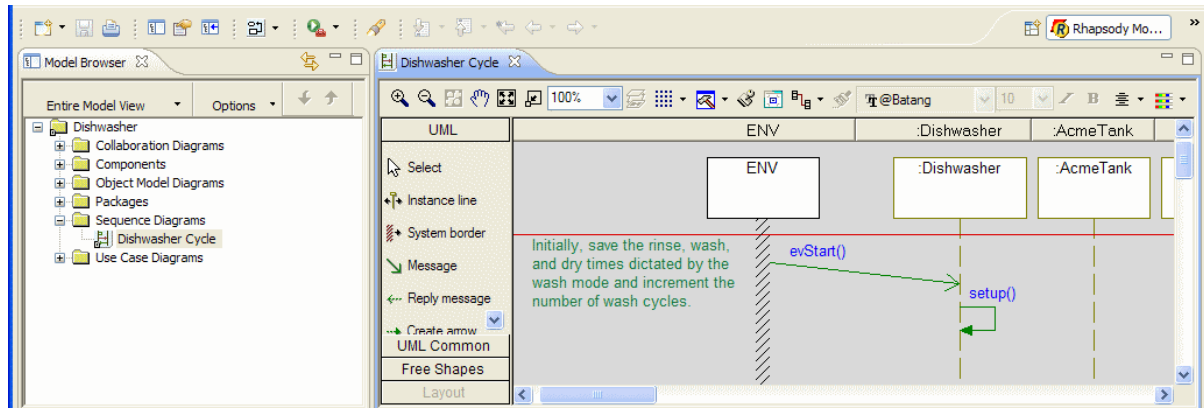
To confirm that you have the Rhapsody Platform Integration within Eclipse, in Eclipse, choose **Help > About Eclipse SDK**. As illustrated with the following figure, you should see a Rhapsody icon. If this icon does not appear, you are not set up for the Rhapsody Platform Integration. Refer to the Eclipse set-up instructions in the *Rhapsody Installation Guide* to set up for this integration.



For descriptions of the areas in the Rhapsody interface, see [Guided Tour of the Rhapsody Platform Integration within Eclipse](#).

Guided Tour of the Rhapsody Platform Integration within Eclipse

The standard Rhapsody interface elements displayed in Eclipse have the same features as in the stand-alone version except that the icons associated with a specific window are displayed at the top of the window, as shown in the following figure:



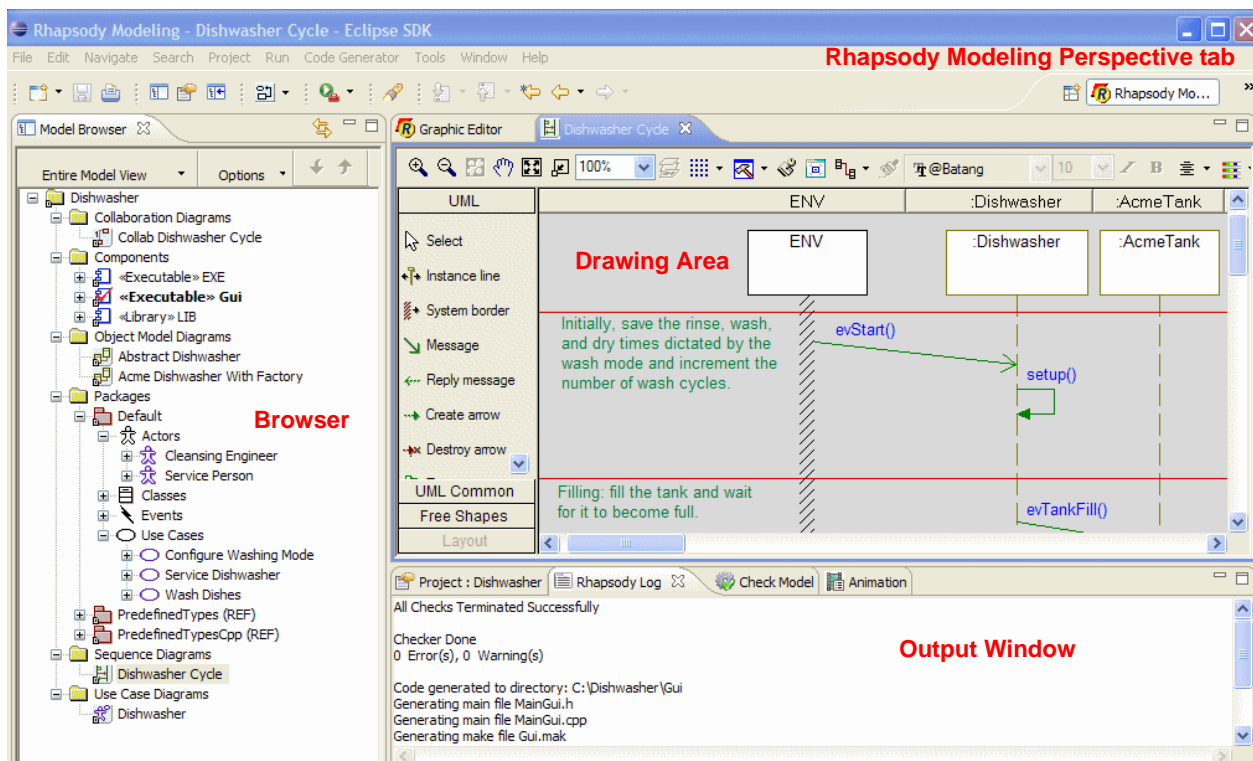
The Rhapsody Platform Integration within Eclipse adds two Rhapsody perspectives on tabs in the upper right corner of the Eclipse IDE:

- ◆ [Rhapsody Modeling Perspective](#)
- ◆ [Rhapsody Debug Perspective](#)

Rhapsody Modeling Perspective

The Rhapsody Modeling Perspective displays the main Rhapsody interface components, as illustrated in the following figure:

- ◆ Browser (Model Browser tab in Eclipse)
- ◆ Diagram Drawing Area
- ◆ Output window and Features dialog box

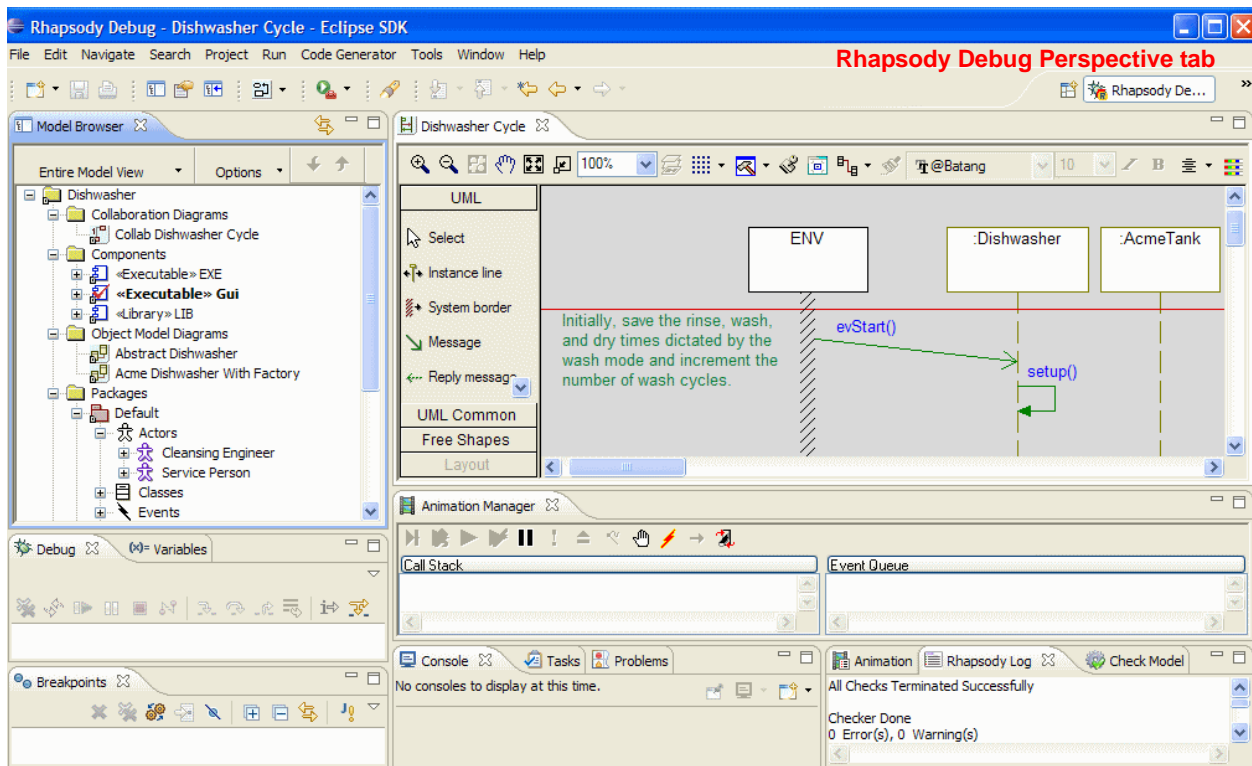


The Rhapsody menu options and drawing capabilities have been added to the Eclipse code editing, interface customization, and other capabilities.

Rhapsody Debug Perspective

The Rhapsody Debug perspective displays a number of windows, as shown in the following figure. When you open the Rhapsody Debug perspective, the following windows may open in the Eclipse IDE:

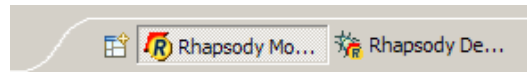
- ◆ Debug
- ◆ Variables
- ◆ Breakpoints
- ◆ Animation Manager
- ◆ Console
- ◆ Tasks
- ◆ Problems
- ◆ Animation
- ◆ Rhapsody Log
- ◆ Check Model



Developers can then use the Eclipse code level debugger and Rhapsody’s design level debugging with animation and breakpoints for a thorough and efficient debugging strategy.

Rhapsody Perspectives in Eclipse

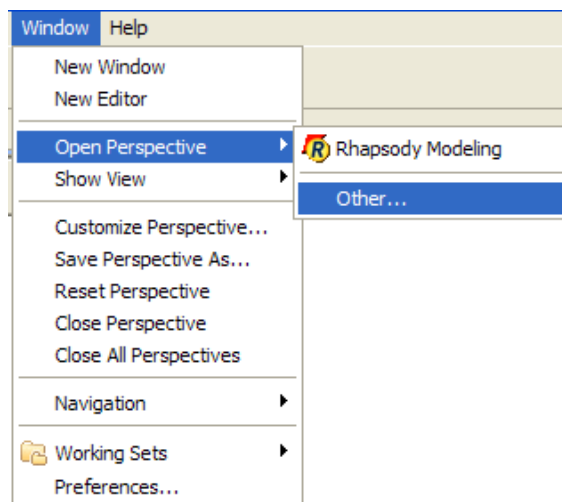
When you create a Rhapsody project in Eclipse, the *Rhapsody Modeling* perspective (highlighted in the following figure) is automatically displayed and is set as the open Eclipse perspective.



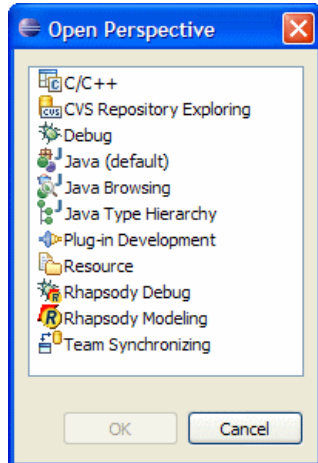
If you have the tabs displayed as shown in the figure above, you can click a tab to go to another perspective.

To open a different view perspective manually, follow these steps:

1. Choose **Window > Open Perspective > Other**, as shown in the following figure:



2. On the Open Perspective dialog box, as shown in the following figure, select another perspective, such as **Rhapsody Debug**, and click **OK**.

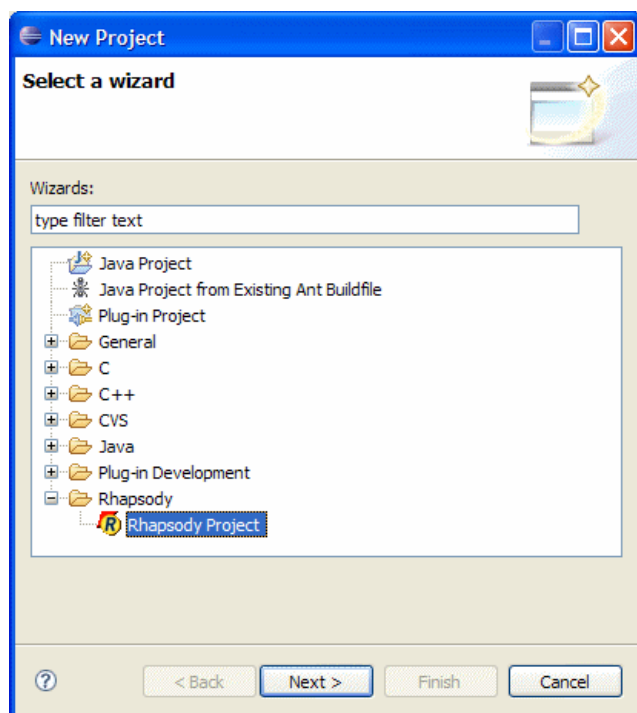


Working with Eclipse Projects

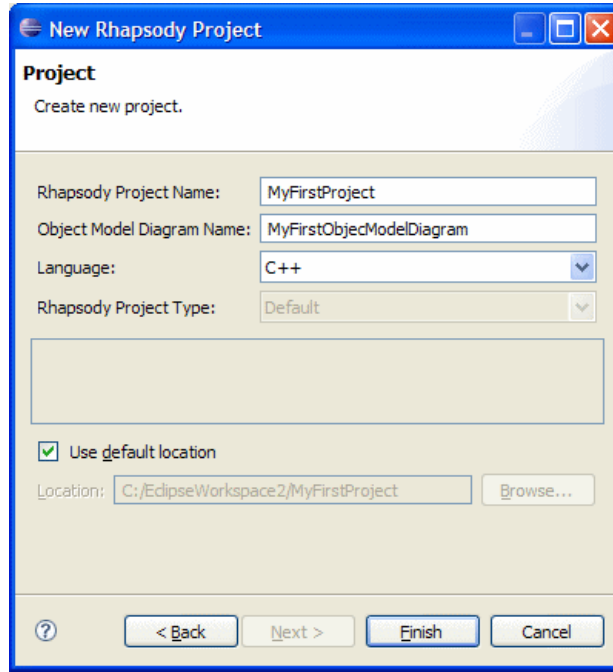
The Eclipse plug-in platform integration provides Rhapsody and Eclipse features for software developers.

Creating a New Rhapsody Project within Eclipse

1. To create a new Rhapsody project within Eclipse, choose **File > New > Project**.
2. In the New Project dialog box, select **Rhapsody Project** and click **Next**.



3. Type a name for your Rhapsody project and select the language (C, C++, or Java) from the drop-down list. In addition, you can type the name for your first object model diagram.



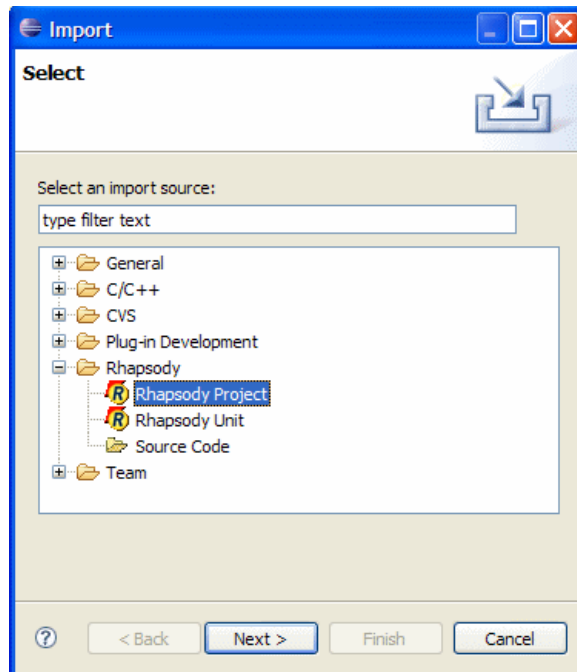
4. If you want to change the location of where you want to store your project, clear the **Use default location** check box and then use the **Browse** button to navigate to another location.
5. Click **Finish** on the New Rhapsody Project dialog box when you are done defining your new Rhapsody project.
6. If the directory for your new project has not yet been created, click **Yes** when you are asked if you want to create it.

Rhapsody creates a new project in the work area you specified and opens the new project.

Opening a Rhapsody Project in Eclipse

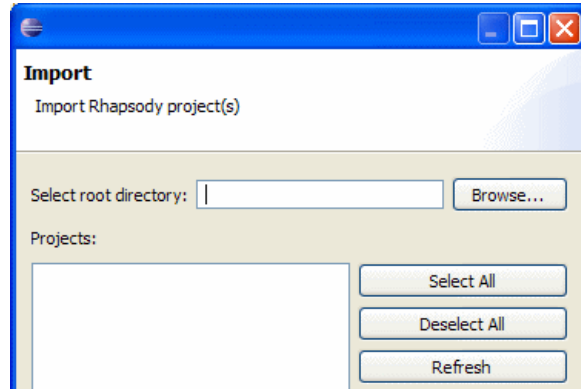
To open an existing Rhapsody project in Eclipse, follow these steps:

1. In Eclipse, choose **File > Import** to open the Import dialog box.
2. Expand the **Rhapsody** folder and select **Rhapsody Project**, as shown in the following figure:

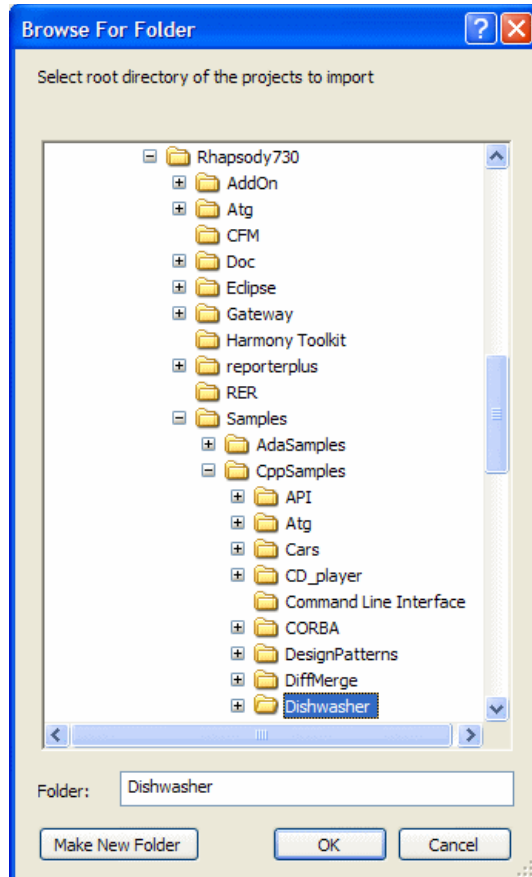


3. Click **Next**.

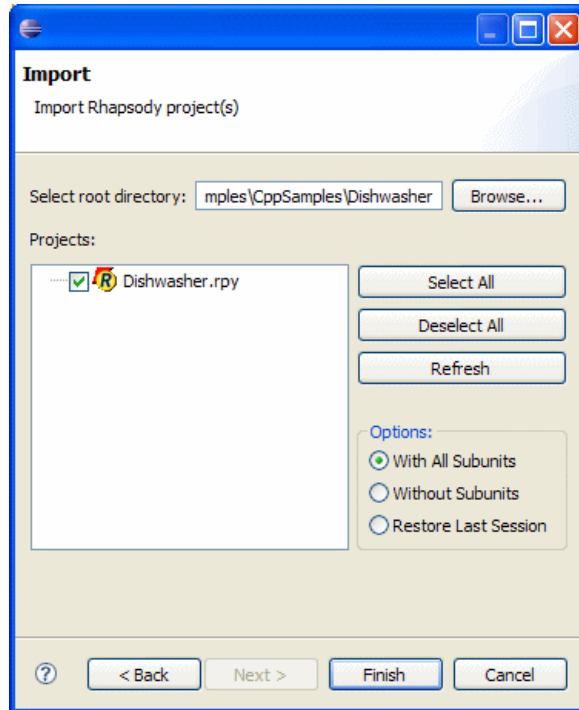
4. On the Import dialog box, as shown in the following figure, click the **Browse** button to open the Browse for Folder dialog box.



5. In the Browse For Folder dialog box, select the folder that contains the Rhapsody project that you want to import and click **OK**.



6. On the Import dialog box, select the project that you want to open and click **Finish** to open the project in Eclipse.



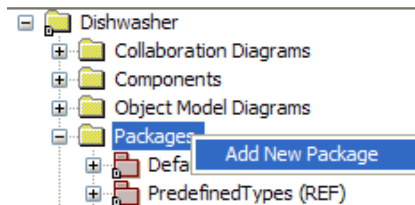
Adding New Elements

To add new elements to your project including diagrams, packages, and files, follow these steps:

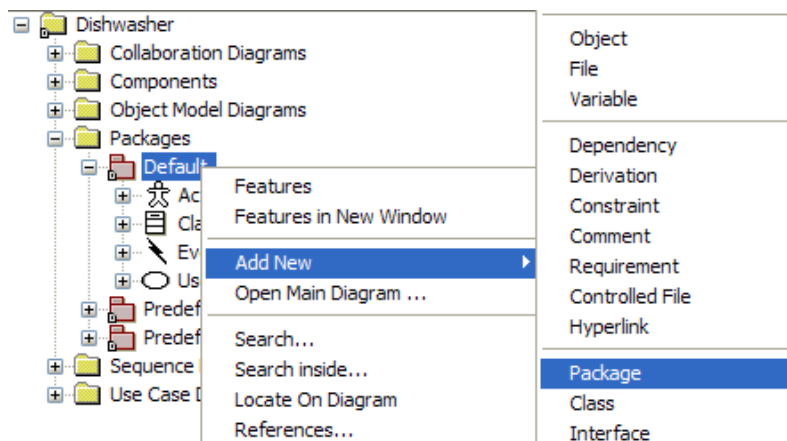
1. With Eclipse Model Browser displayed, right-click an element for which you want to add an element.
2. Select **Add New [element]** or **Add New > [element type]** from the pop-up menu depending on what type of element you selected. (See also [Filtering Out File Types](#).)

The following figures show how you can add an element to your project. While this example shows adding a package. The same method is true for other elements, such as diagrams, files, actors, operations, and so forth.

If you want to create a main-level package, right-click **Package**, as shown in the following figure, and select **Add New Package**.



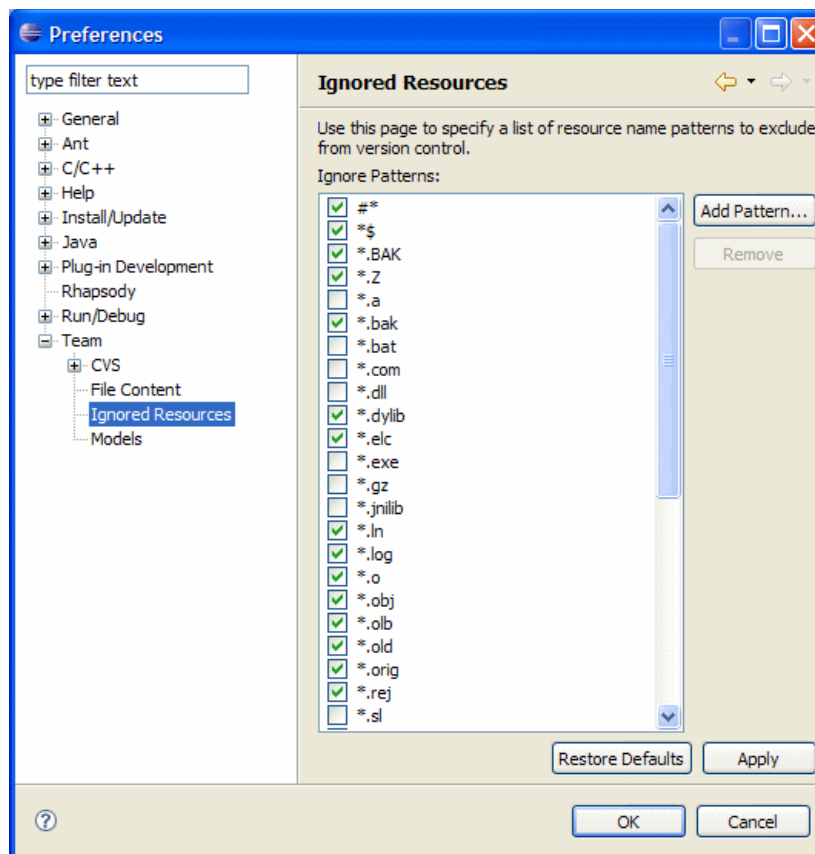
If you want to create a subpackage (a package within a package), right-click a package and select **Add New > Package**, as shown in the following figure. As you can see from this figure, you can add other elements (such an object, dependence, a class, and so forth) as well.



Filtering Out File Types

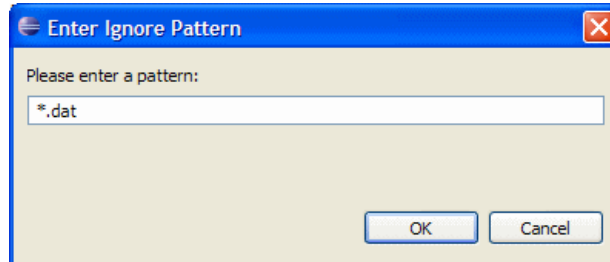
To prevent Eclipse from presenting unwanted Rhapsody project file types when adding files, follow these steps:

1. In Eclipse, choose **Window > Preferences**.
2. In the Preferences dialog box, expand the **Team** section of the tree structure.
3. Select **Ignored Resources** to display the list of file extensions that can be set to be ignored, as shown in following figure:

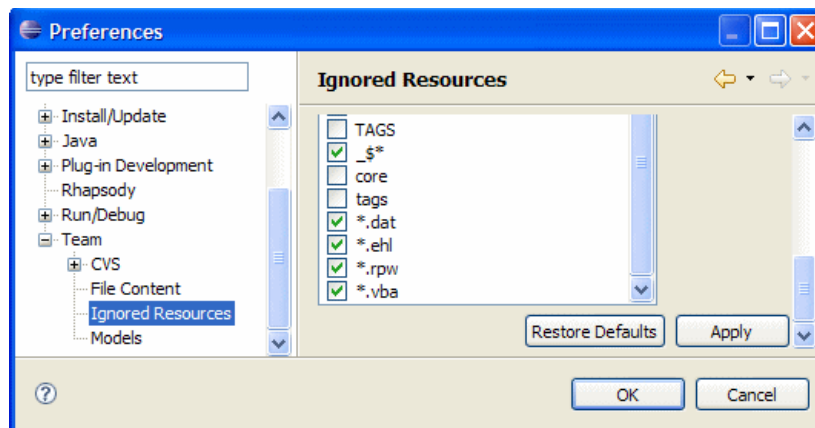


4. If the list does not contain the file types you want to filter out, click **Add Pattern**.

5. In the Add Ignore Pattern dialog box, type the file extension using the format shown, as shown in the following figure, and click **OK** to add the extension to the “ignored” list.



6. Using that method, you may want to enter the Rhapsody *.dat, *.ehl, *.rpw, and *.vba file extensions, as shown in the following figure:

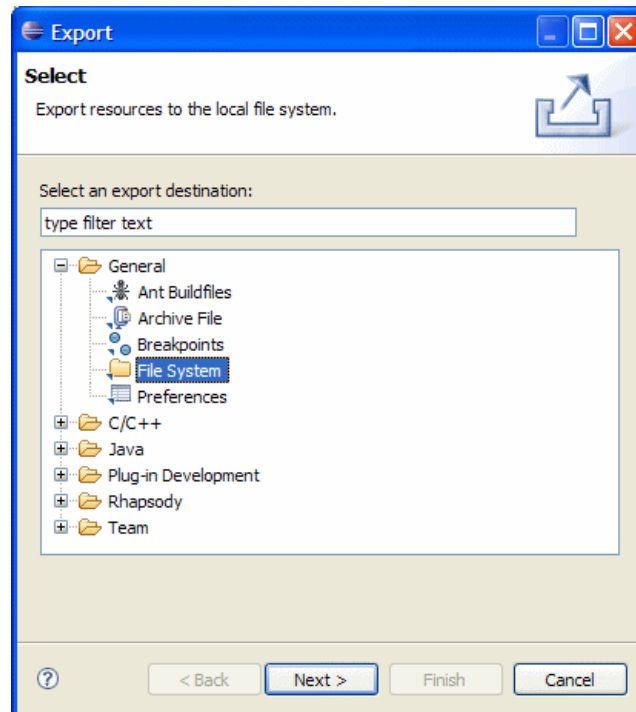


7. Click **OK** to save your changes and close the Preferences dialog box.

Exporting Eclipse Source Code to a Rhapsody Project

To export an Eclipse source code project from Eclipse to Rhapsody, follow these steps:

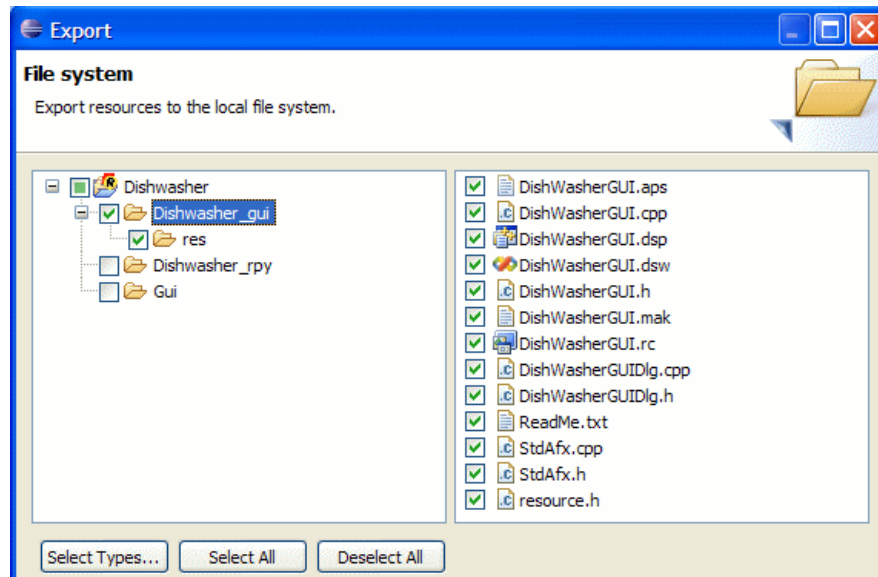
1. With the model open in Eclipse, choose **File > Export**.
2. Expand the **General** folder and select **File System** from the Export dialog box, as shown in the following figure, and click **Next**.



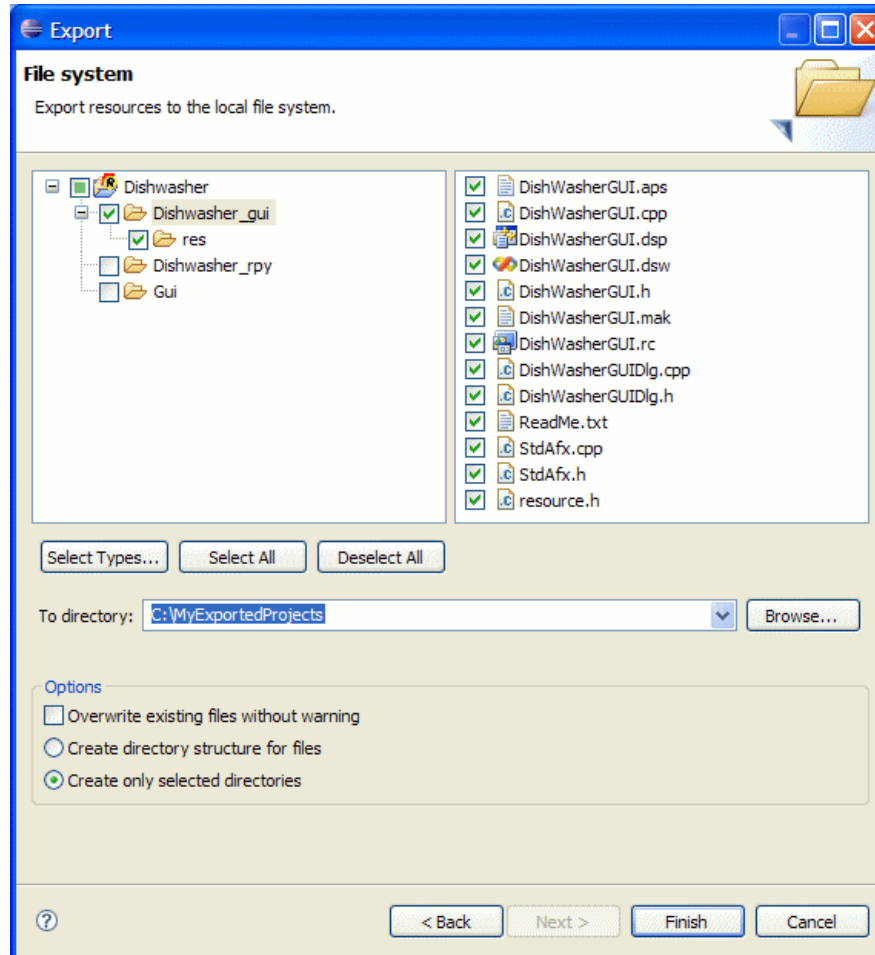
3. In the Export dialog box, in the list in the left box, as shown in the following figure:

- ◆ Select the top-level project to select everything in the project, or
- ◆ Select a subfolder folder to select only those items in that subfolder.

Note: You can select specific files to export from the list on the right. You can also use **Filter Types** to select specific file extensions to be exported.



4. Use the **Browse** button to navigate to a location directory for the exported files, as shown in the following figure:



5. Select any of the **Options** that apply to this operation.
6. Click **Finish** to complete the export operation.

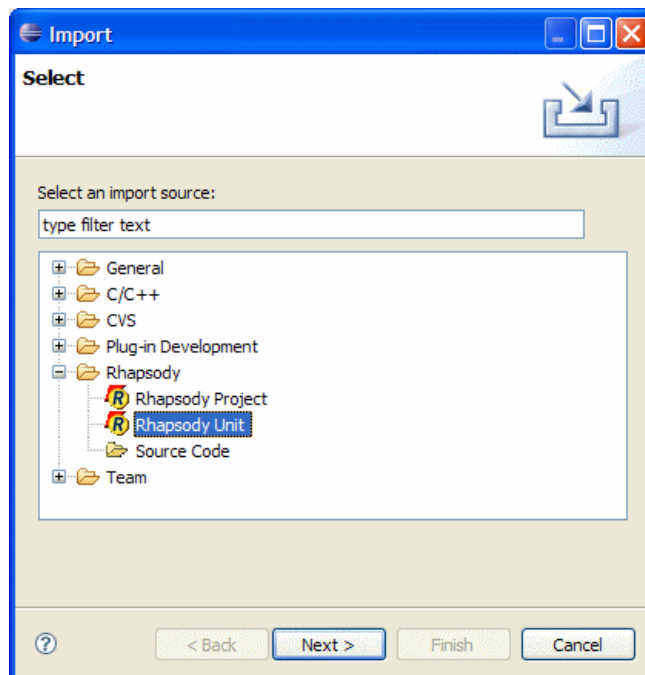
Note: If any problems were encountered during the operation, an Export Problems message box appears. Click the **Details** button for more information.

7. To see your exported files, go to the location directory for your exported files.

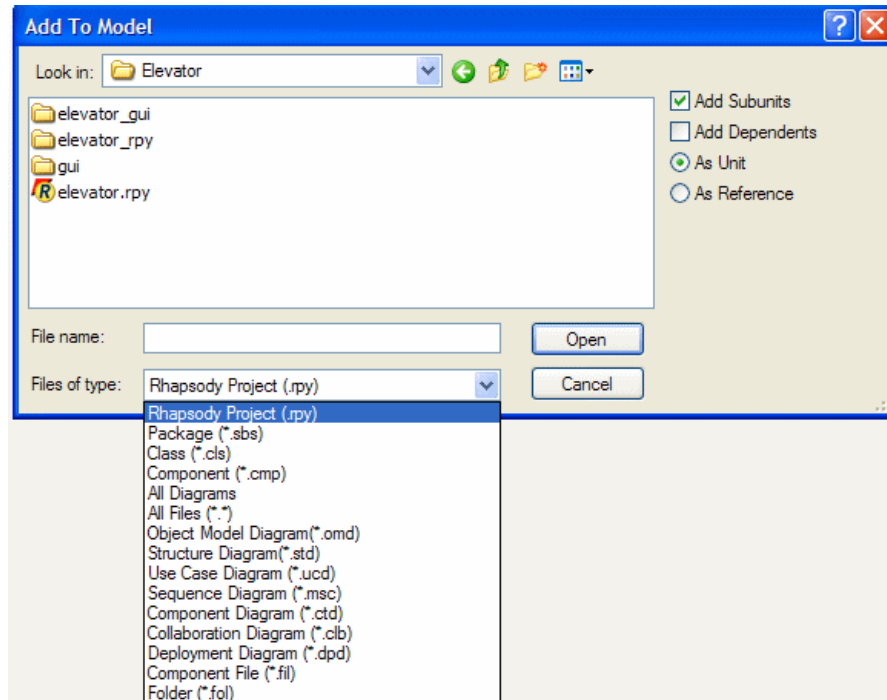
Rhapsody Unit Import

To import Rhapsody units into Eclipse, follow these steps:

1. In Eclipse, set the project for which you want to import units as the *active configuration*. Right-click the project and select **Set as Active Project**.
2. Choose **File > Import** to open the **Import** dialog box.
3. Select **Rhapsody Unit**, as shown in the following figure, and click **Next**.

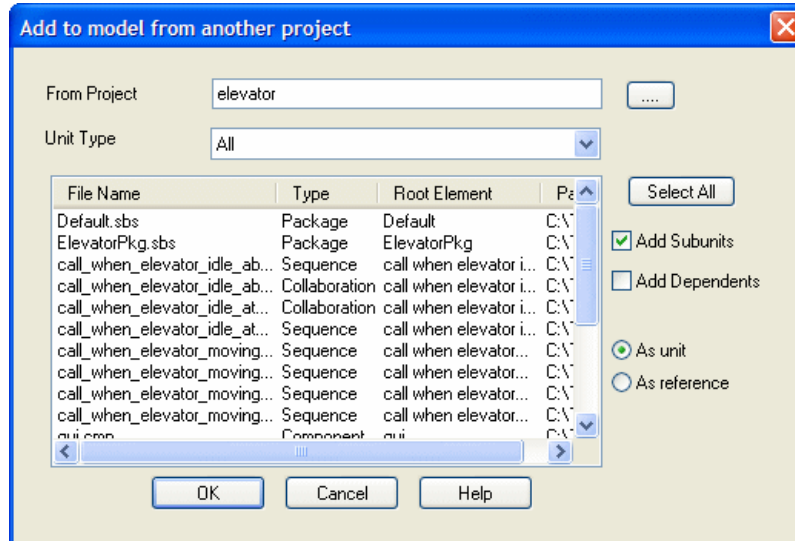


4. In the Add to Model dialog box, you can select a single file type (unit type) that you want to import by using the **Files of Type** drop-down menu, as shown in the following figure, or you can select the Rhapsody project (.rpy) to select all of the units in the project:



5. Click **Open**.

6. In the Add To Model From Another Project dialog box, select the units you want and make your applicable selections to the check boxes and radio buttons on the right.

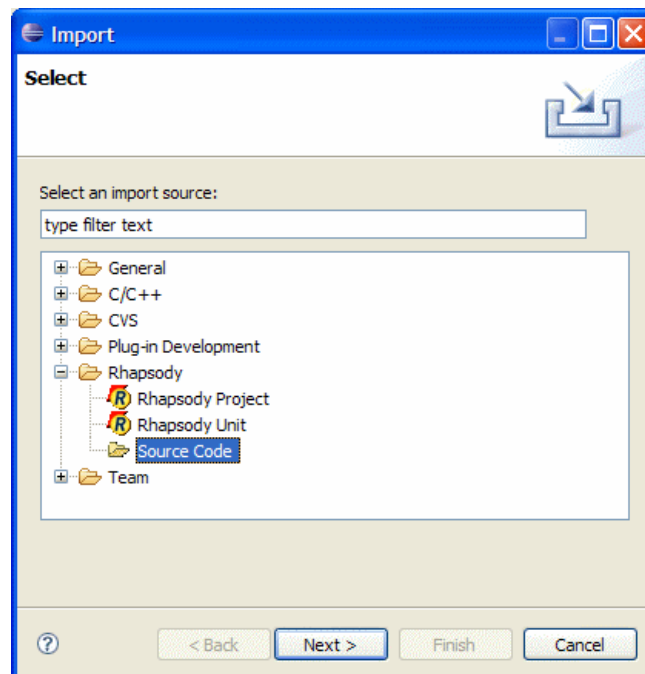


7. Click **OK** to import the units.
8. After the import is completed, a dialog box displays the status of the import. Click **Finish** to close the dialog box and return to Eclipse.

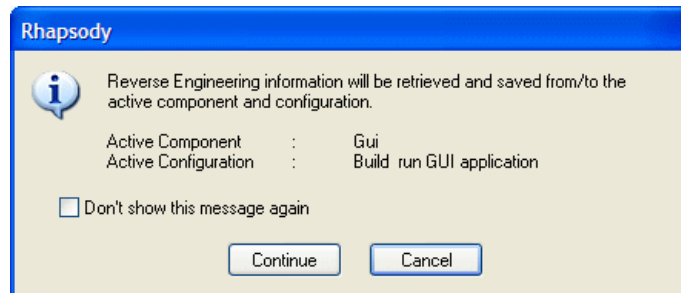
Source Code Import (Reverse Engineering)

To import source code, following these steps:



1. In Eclipse, set the project and the component as the active configuration.
 - a. Right-click the project and select **Set as Active Project**.
 - b. Right-click the component and select **Set as Active Component**.
2. In your Eclipse project, choose **File > Import** to open the Import dialog box.
3. Select **Source Code**, as shown in the following figure, and click **Next**.

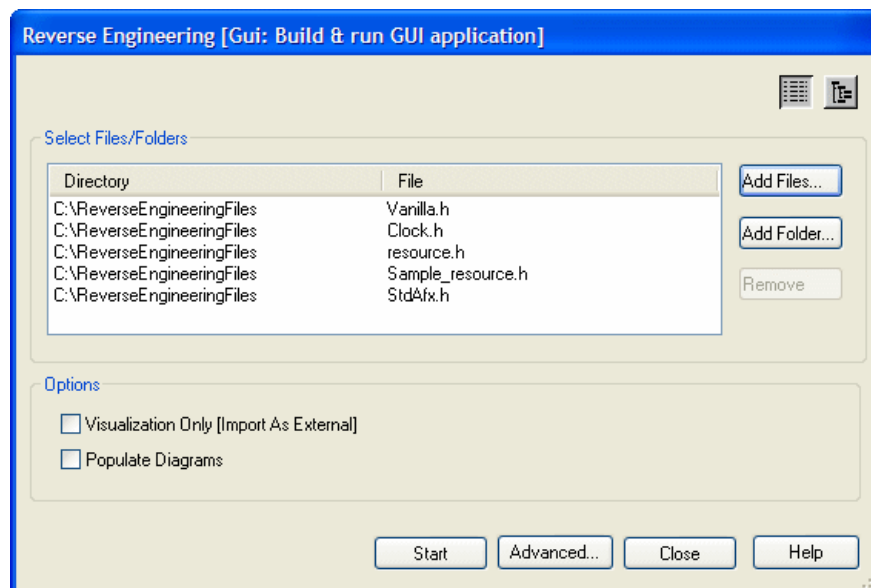


4. When the message dialog box opens, confirm that you want to launch the Rhapsody Reverse Engineering interface, click **Finish**.
5. When asked to confirm that you want the reverse engineered code to be saved to the active component and configuration, as shown in the following figure, click **Continue**.

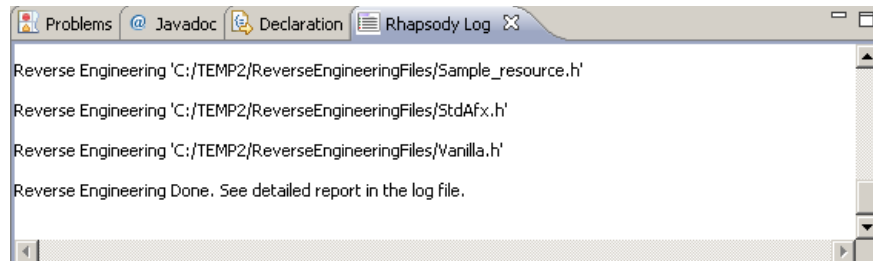


6. On the Reverse Engineering dialog box, click **Add Files** or **Add Folder** to add items to be reverse engineered.
7. After selecting the items to reverse engineer, click **Start**.

Note: You have a choice of a flat view, as shown in the following figure, or a tree view for the selected items. To toggle between the views, click the Flat View button  or the Tree View button .



8. Confirm that you want to continue with the reverse engineering process, click **Yes**.
9. Click **Finish**. The reverse engineering messages display in the **Rhapsody Log** window, as shown in the following figure:



Note

For more information about reverse engineering, see [Reverse Engineering](#).

Search and Replace in Models

The Rhapsody search facility is available to use in Eclipse for these operations:

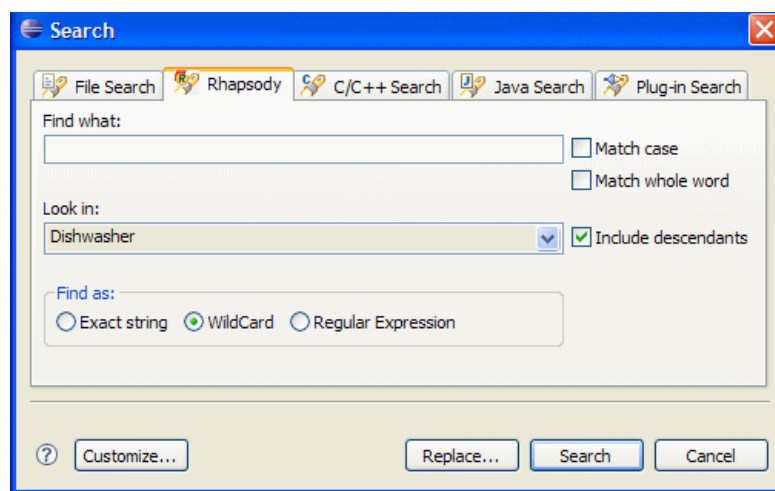
- ◆ Perform standard search-and-replace operations
- ◆ Search for the following:
 - unresolved elements in a model
 - unloaded elements in a model
 - units in the model
 - both unresolved elements and unresolved units
- ◆ Work with the search results

For more detailed instructions for the Rhapsody Search and Replace facility, see [Search and Replace Facility](#).

Accessing the Rhapsody Search Facility in Eclipse

To access Rhapsody search in Eclipse, follow these steps:

1. With your Rhapsody model open in Eclipse, choose **Search > Search** to open the Search dialog box, which by default opens with the File Search tab in focus. (You can also right-click an element in the model browser and select **Search**.)
2. For the Rhapsody search facility, click the **Rhapsody** tab, as shown in the following figure. (You can also right-click an element on the model browser and select **Search inside**.)



Customizing the Search Criteria

In the Rhapsody search facility you can use any of the standard search facilities. You can also customize your search using the following buttons.

- ◆ **Exact string** allows a non-regular expression search. When selected, the search looks for the string entered into the search field (such as `char*`).
- ◆ **Wildcard** allows wildcard characters in the search field such as `*` and produces results during the search operation that include additional characters. For example, the search `*dishwasher` matches class `dishwasher` and attribute `itsdishwasher`.
- ◆ **Regular Expression** allows the use of Unix-style regular expressions. For example, `itsdishwasher` can be located using the search term `[s]dishwasher`.

Displaying the Search Results

The search results display in the Search tab of the Eclipse output window, as shown in the following figure:

The screenshot shows the Eclipse Search window titled 'Searching 'Dishwasher''. The window displays a table of search results with four columns: Name, Type, Field, and Status. The results are as follows:

Name	Type	Field	Status
Dishwasher	Project	Name	Found
2 in Default::AcmeTank	Transition	Transition label	Found
3 in Default::AcmeTank	Transition	Transition label	Found
4 in Default::AcmeTank	Transition	Transition label	Found
5 in Default::AcmeTank	Transition	Transition label	Found
Dishwasher in Default	Class	Name	Found
m_iServiceState in Default::Dishwasher	Attribute	Description	Found
m_iOperationState in Default::Dishwasher	Attribute	Description	Found
StatechartOfDishwasher in Default::Dishwasher	Statechart	Name	Found
drying in Default::Dishwasher.StatechartOfDishwasher.ROOT.active.state_1.doorClosed	State	Exit Action	Found
filling in Default::Dishwasher.StatechartOfDishwasher.ROOT.active.state_1.doorClosed	State	Entry Action	Found
Dishwasher() in Default::Dishwasher	Constructor	Body	Found
Dishwasher() in Default::Dishwasher	Constructor	Name	Found
~Dishwasher() in Default::Dishwasher	Destructor	Name	Found
itsDishwasher in Default::Tank	Association End	Name	Found
2 in Default::Tank	Transition	Transition label	Found
3 in Default::Tank	Transition	Transition label	Found
4 in Default::Tank	Transition	Transition label	Found
5 in Default::Tank	Transition	Transition label	Found
Service Dishwasher in Default	Use Case	Description	Found
Service Dishwasher in Default	Use Case	Name	Found
itsService Dishwasher in Default::Service Person	Association End	Name	Found
Abstract Dishwasher	Object Model Diagram	Name	Found
Acme Dishwasher With Factory	Object Model Diagram	Name	Found
Dishwasher Cycle	Sequence Diagram	Name	Found

Working with Search Results

After locating elements using the Search facility, you can perform the following operations in the Search window:

- ◆ Sort items
- ◆ Check the references for each item
- ◆ Delete
- ◆ Load

To sort the items in the list, click the heading for the column to sort according to information in that column.

To work with an item located in the search, follow these steps:

1. Double-click an item in the list to open the its features window and highlight its location on the model browser.
2. Make any required changes from these entry points.

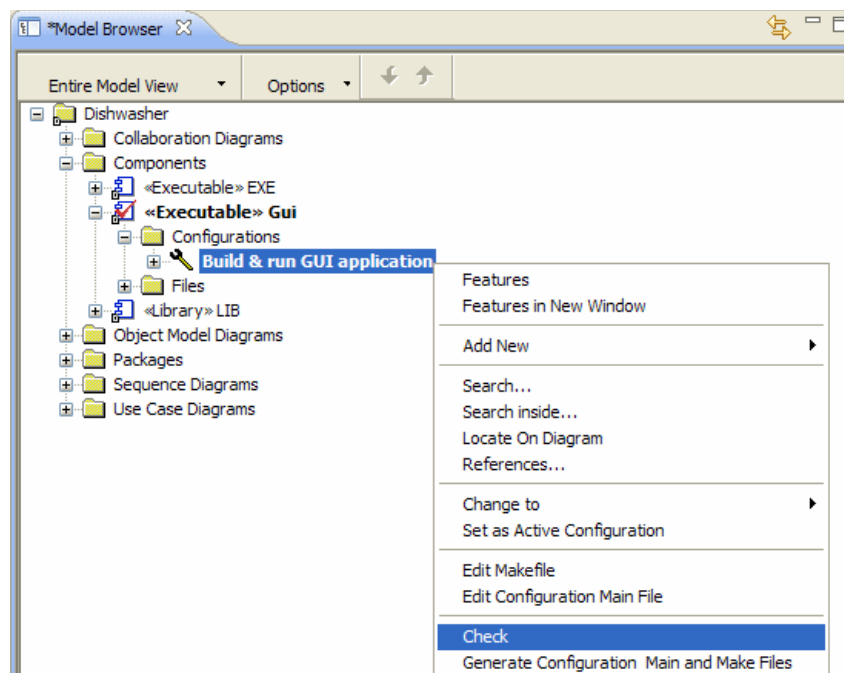
Generating and Editing Code

You can check your model, generate code, and edit the resulting code using Eclipse facilities. For detailed instructions describing Rhapsody code generation, see [Basic Code Generation Concepts](#).

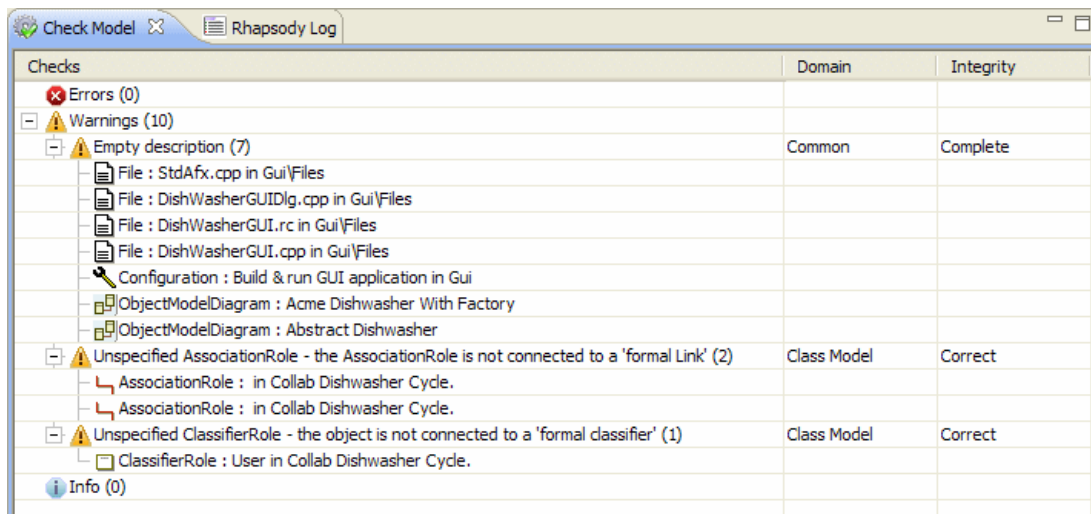
Checking the Model

To launch the check model process, use one of these methods:

- ◆ Choose **Tools > Check Model** and select one of these options:
 - [name of active configuration]
 - **Selected Elements**
 - **Configure**
- ◆ Right-click the active configuration and select **Check**, as shown in the following figure:



The results display in the Check Model tab of the Output window.



As with search results, you can double-click an item on the Check Model tab to open the Features dialog box for the item and highlight it in the model browser.

Generating Code

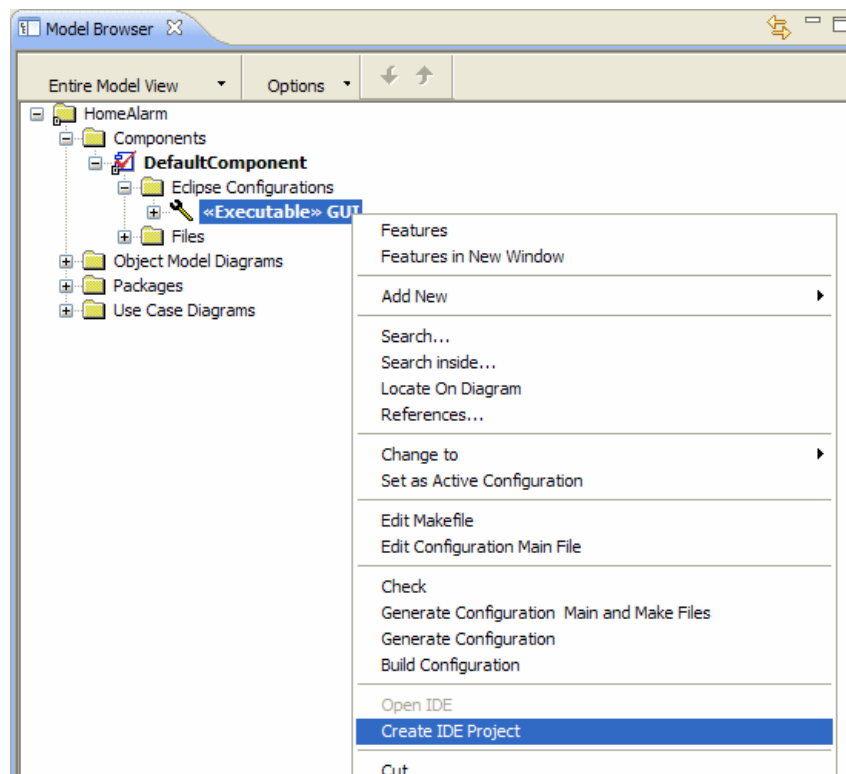
Rhapsody uses an Eclipse IDE project for code generation. Before you can generate code, you must perform the following tasks:

- ◆ Create an Eclipse IDE project
- ◆ Associate the Rhapsody Eclipse configuration with the Eclipse IDE project

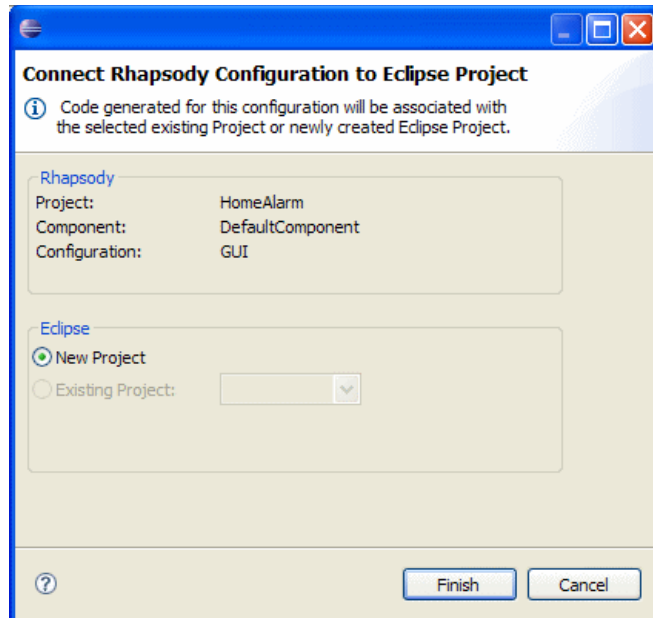
Creating an Eclipse IDE Project

To create an IDE project to use for Rhapsody code generation, follow these steps:

1. In the model browser, right-click an Eclipse configuration and select **Create IDE Project**, as shown in the following figure:

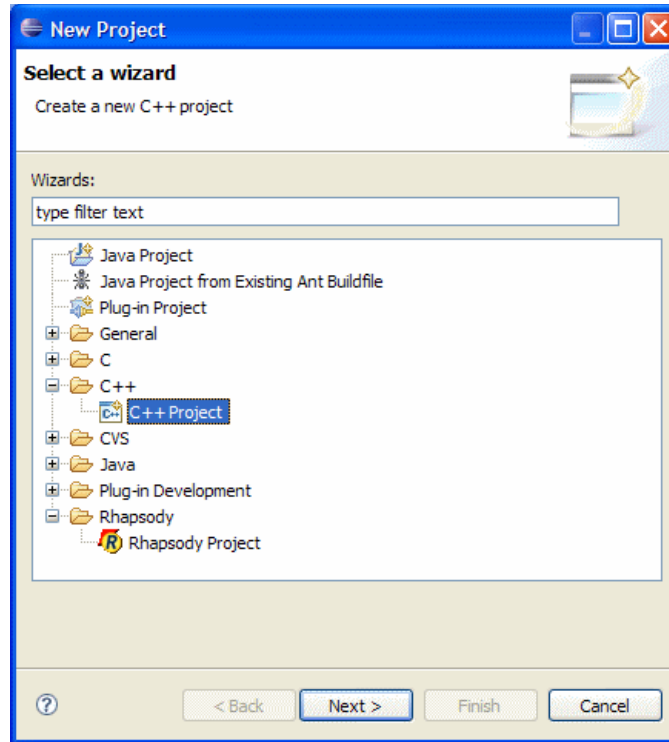


2. If you have an existing Eclipse project in your work area, that project is listed in the **Existing Project** drop-down list. However, you need to create a special Eclipse IDE project, so you should select the **New Project** radio button, as shown in the following figure:



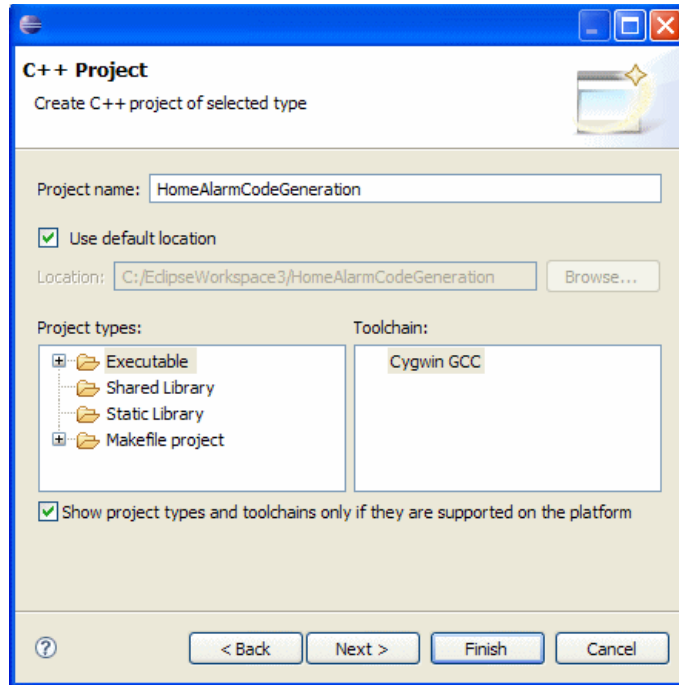
3. Click **Finish**.

4. In the New Project dialog box, select the project type based on your environment. The following figure below shows a C++ project selection.

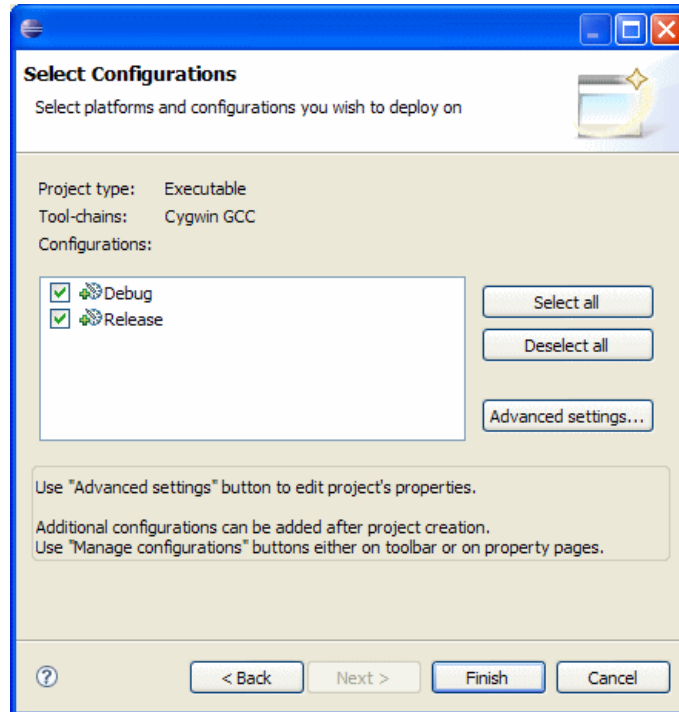


5. Click **Next**.

6. Enter a project name on the Project dialog box, as shown in the following figure, and click **Next**.



7. If necessary, make a configuration selection on the Select Configurations dialog box, and then click **Finish**.



8. If your project type selection is different from your current perspective, an Open Associated Perspective dialog box opens to give you an opportunity to switch perspectives; click **Yes**. If you want to keep using your currently active perspective, click **No**.

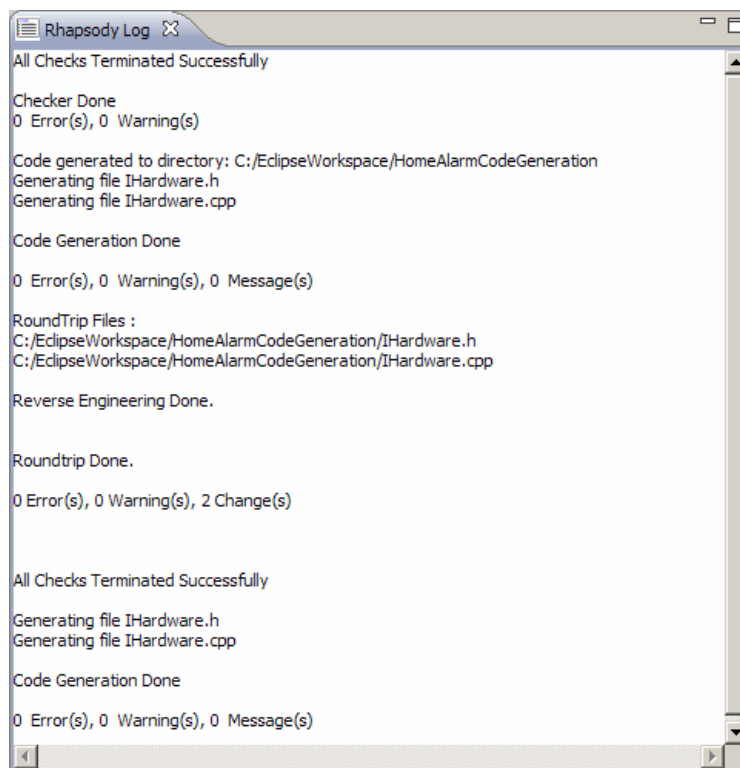
Generating Code

After setting up the IDE project to receive the generated code, choose **Code Generator > Generate** and one of these options:

- ◆ **[Active Configuration]** (In the following figure, the active configuration is called GUI.)
- ◆ **Selected classes**
- ◆ **[Active Configuration] with Dependencies**
- ◆ **Entire Project**

The system generates the requested code and displays messages in a log file, as shown in the following figure:

Note: To show a complete log report, the following figure shows a very short report. Typically, especially for the first code generation, there may be more messages for each group (for example, the listing of code generated may be longer).



Dynamic Model-Code Associativity

Rhapsody lets you work in the model or code and maintain synchronization between each so that changes in one are reflected in the other automatically. This is called Dynamic Model-Code Associativity (DMCA).

To select the DMCA option you want to use, follow these steps:

1. Choose **Code Generator > Dynamic Model Code Associativity**.
2. From the submenu, select one of these commands:
 - ◆ **Bidirectional** - changes made to the code or model are synchronized with the other.
 - ◆ **Roundtrip** - changes made in the code are automatically synchronized with the model.
 - ◆ **Code Generation** - changes made in the model are updated in the code automatically.
 - ◆ **None**

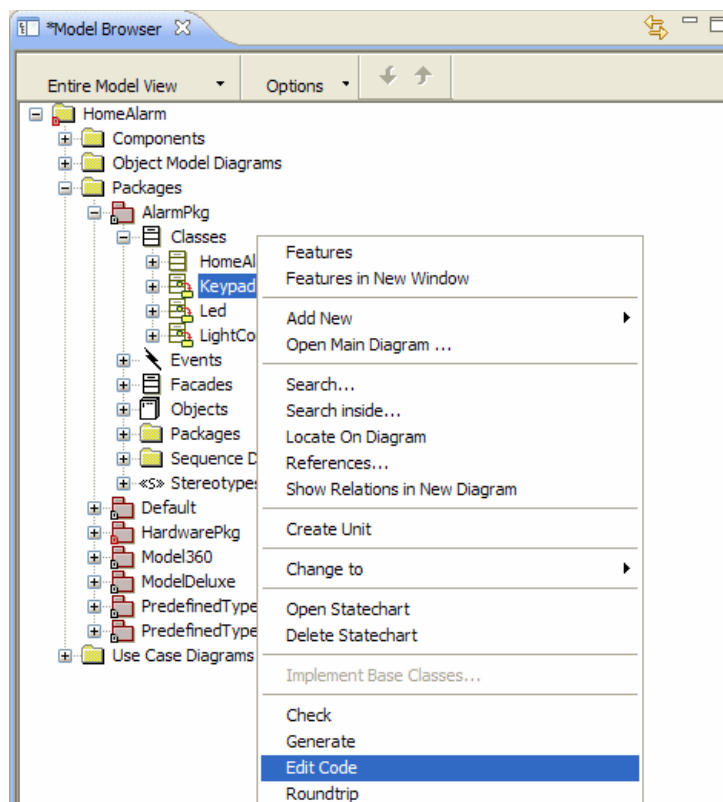
Editing Code

You can edit your Rhapsody code using the Eclipse editor.

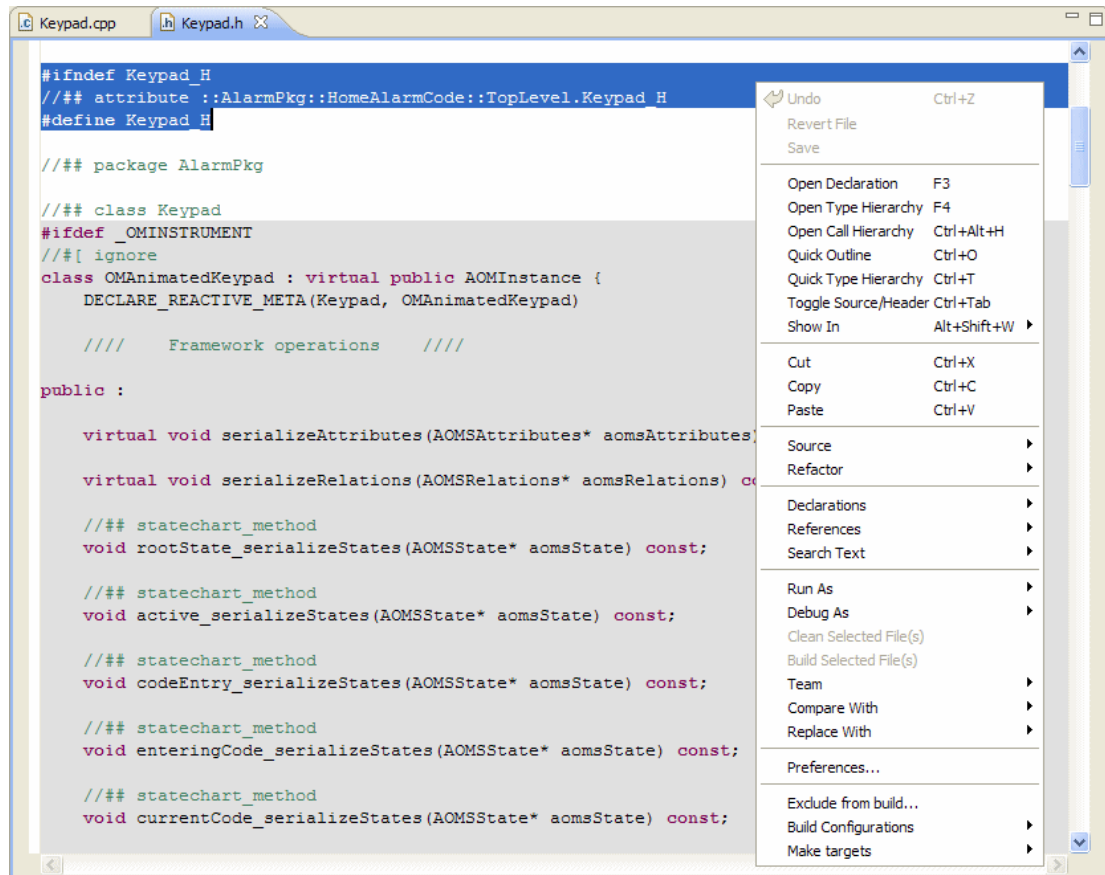
Launching the Eclipse Code Editor from the Browser or Diagram

To display generated code for a specific element in the code editor, follow these steps:

1. In the model browser, right-click an item and select **Edit Code**, as shown in the following figure, to launch the corresponding source code in the Eclipse code editor.



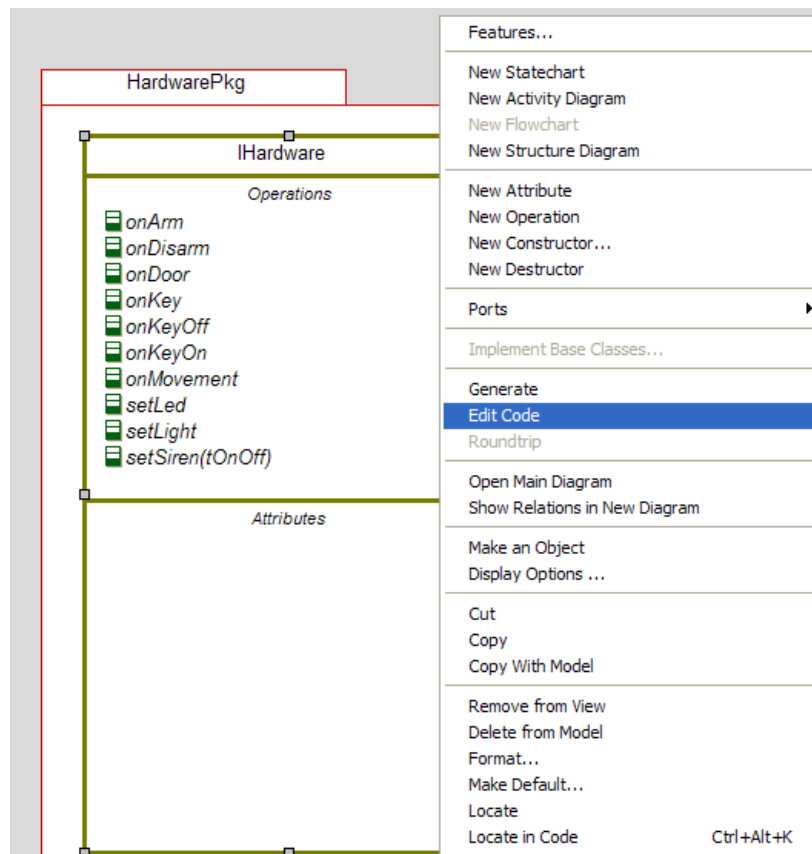
- On the Eclipse code editor, highlight any items of interest and right-click to display the editing menu, as shown in the following figure:



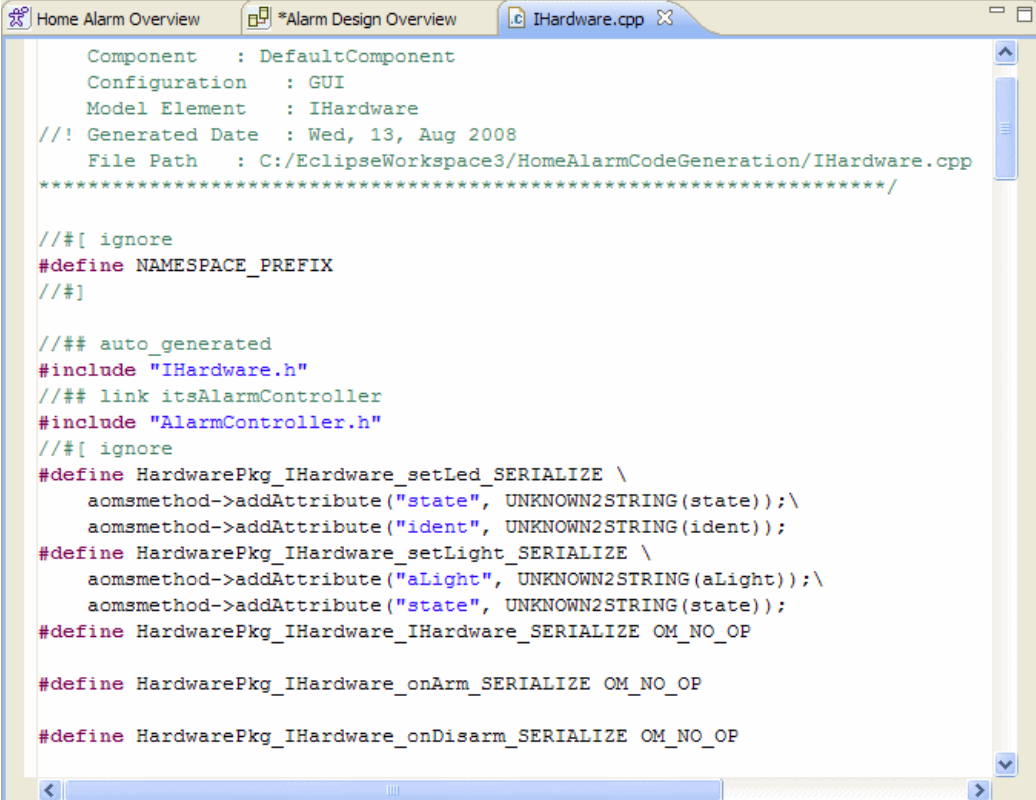
Editing Code from a Diagram Element

To launch the Eclipse code editor from a diagram element, follow these steps:

1. Open the diagram.
2. Right-click an item in the diagram and select **Edit Code**, as shown in the following figure:



3. The Eclipse code editor opens, as shown in the following figure:



```
Component      : DefaultComponent
Configuration   : GUI
Model Element  : IHardware
//! Generated Date : Wed, 13, Aug 2008
File Path      : C:/EclipseWorkspace3/HomeAlarmCodeGeneration/IHardware.cpp
*****/

//#[ ignore
#define NAMESPACE_PREFIX
//#]

//## auto_generated
#include "IHardware.h"
//## link itsAlarmController
#include "AlarmController.h"
//#[ ignore
#define HardwarePkg_IHardware_setLed_SERIALIZE \
    aomsmethod->addAttribute("state", UNKNOWN2STRING(state));\
    aomsmethod->addAttribute("ident", UNKNOWN2STRING(ident));
#define HardwarePkg_IHardware_setLight_SERIALIZE \
    aomsmethod->addAttribute("aLight", UNKNOWN2STRING(aLight));\
    aomsmethod->addAttribute("state", UNKNOWN2STRING(state));
#define HardwarePkg_IHardware_IHardware_SERIALIZE OM_NO_OP

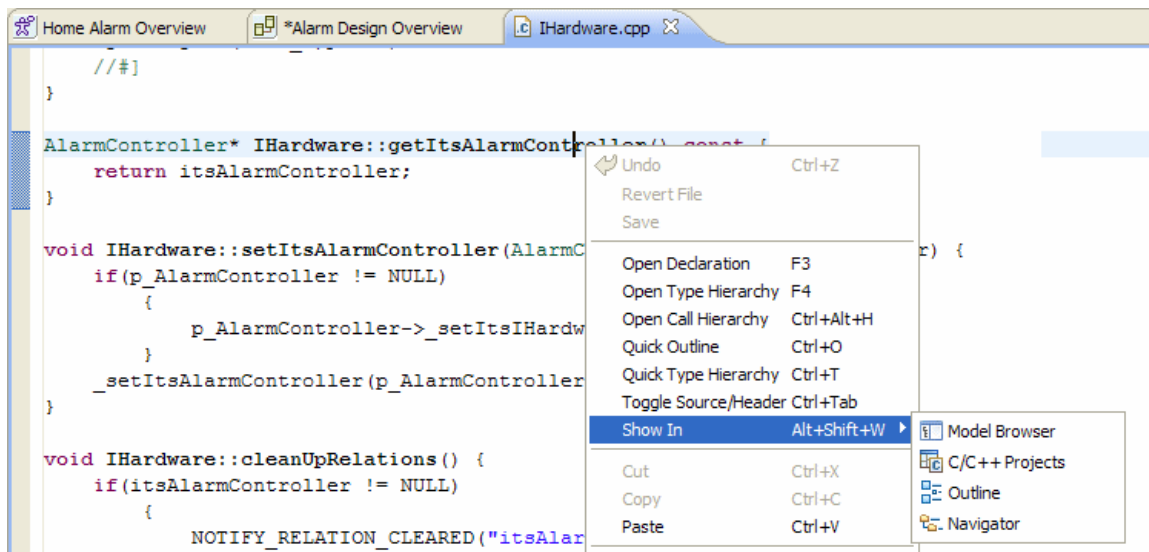
#define HardwarePkg_IHardware_onArm_SERIALIZE OM_NO_OP

#define HardwarePkg_IHardware_onDisarm_SERIALIZE OM_NO_OP
```

Locating an Element in the Browser from the Editor


If you are editing code and need to see an item in the project, usually in the model browser, follow these steps:

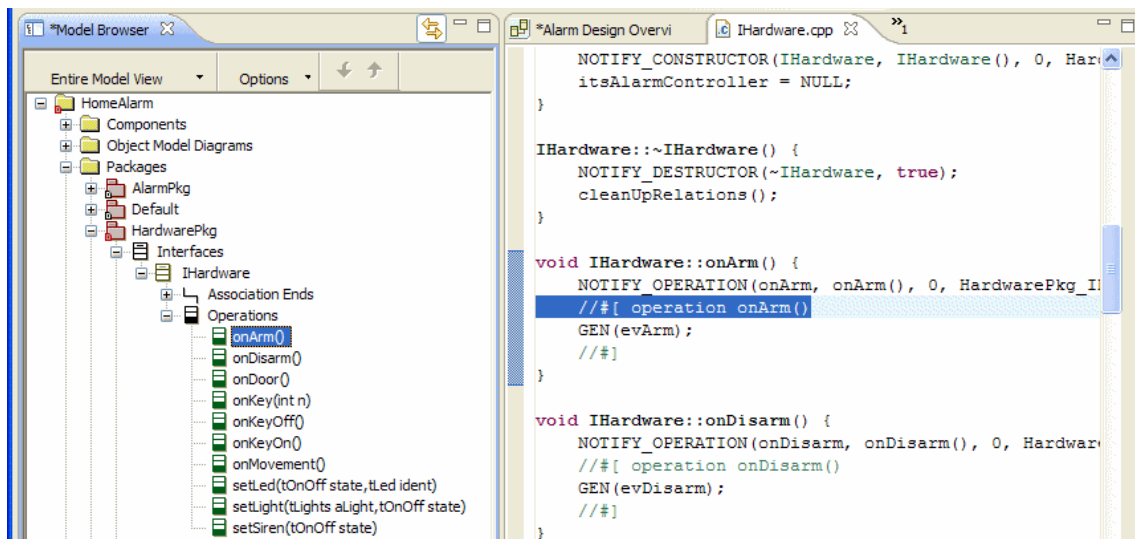
1. Highlight the item in the code.
2. Right-click and select **Show in** and then one of the options, as shown in the following figure. The option listed on this submenu are controlled by the type of project displayed. This option list is for a C++ project:
 - ◆ Model Browser
 - ◆ C/C++ Projects
 - ◆ Outline
 - ◆ Navigator



Viewing Code Associated with a Model Element

To see the code automatically for a selected model element, follow these steps:

1. Click the **Link with Editor** button  located at the top of the Eclipse model browser.
2. Click an item in the model browser and notice that the item's code is immediately displayed and highlighted in the Eclipse code editor, as shown in the following figure:



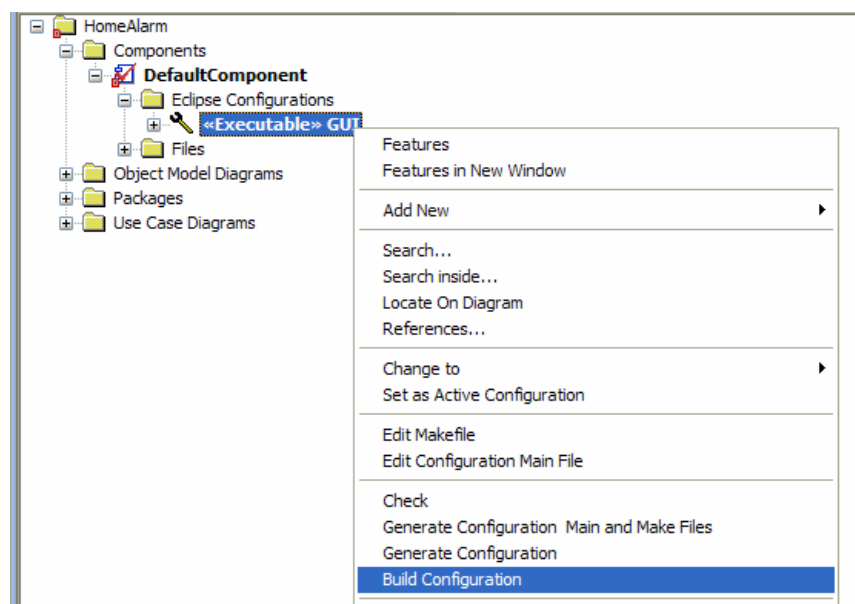
Building, Debugging, and Animating

After checking your model and generating code, you can build your project.

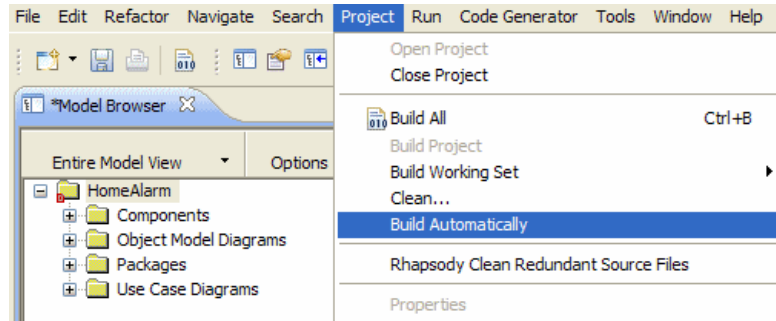
Building Your Eclipse Project

You can use any of these methods to build your Eclipse project:

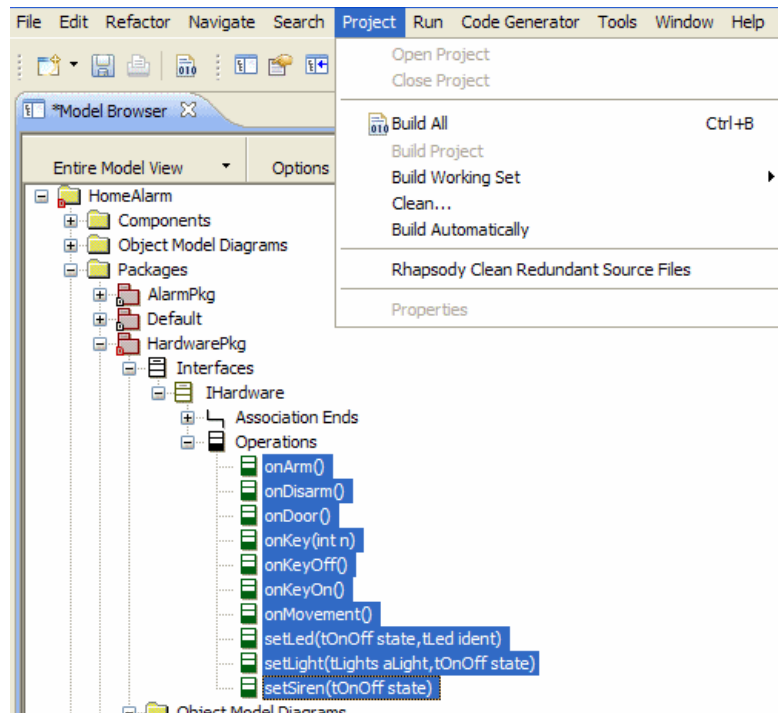
- ◆ In the model browser, right-click the active configuration for your Eclipse project and select **Build Configuration**, as shown in the following figure:



- ◆ Choose **Project > Build Automatically** to build from Eclipse using Java or C/C++ build tools, as shown in the following figure:

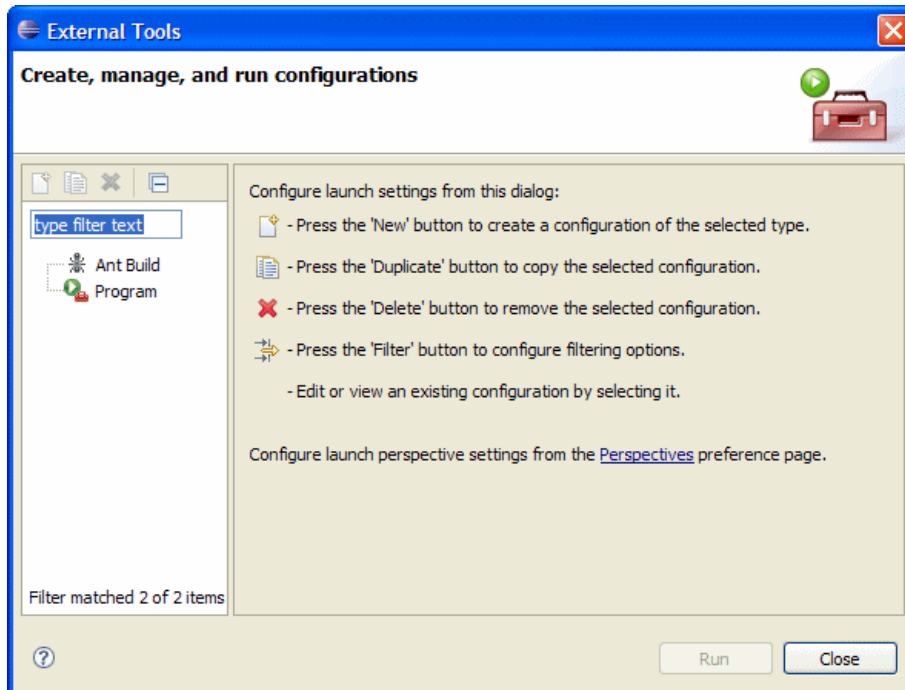


- ◆ Select an element or group of elements in the model browser and choose the **Project** menu to display this menu and select a build option.



Debugging Your Eclipse Project

After you have built your project, you can use the Eclipse debugging facilities with the Rhapsody Debug perspective. Choose the **Run** menu and then any of the debugging tools you usually use in Eclipse. The External Tools dialog box, as shown in the following figure, provide access to programs used to create, manage, and run configuration you built project.



Rhapsody Animation in Eclipse

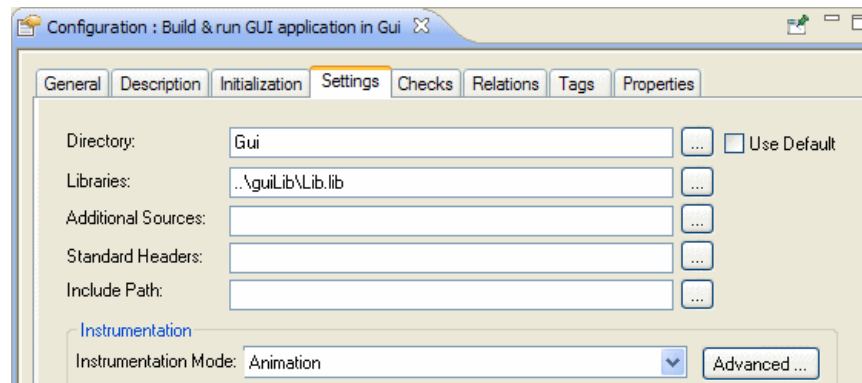
When running animated Rhapsody applications, Eclipse switches to the Rhapsody Debug perspective. You can debug an animated application using both Rhapsody Animation functionality and Eclipse Debugging tools.

For detailed instructions, see [Animation](#).

Preparing for Animation

Before you can begin animation you must prepare for it, follow these steps:

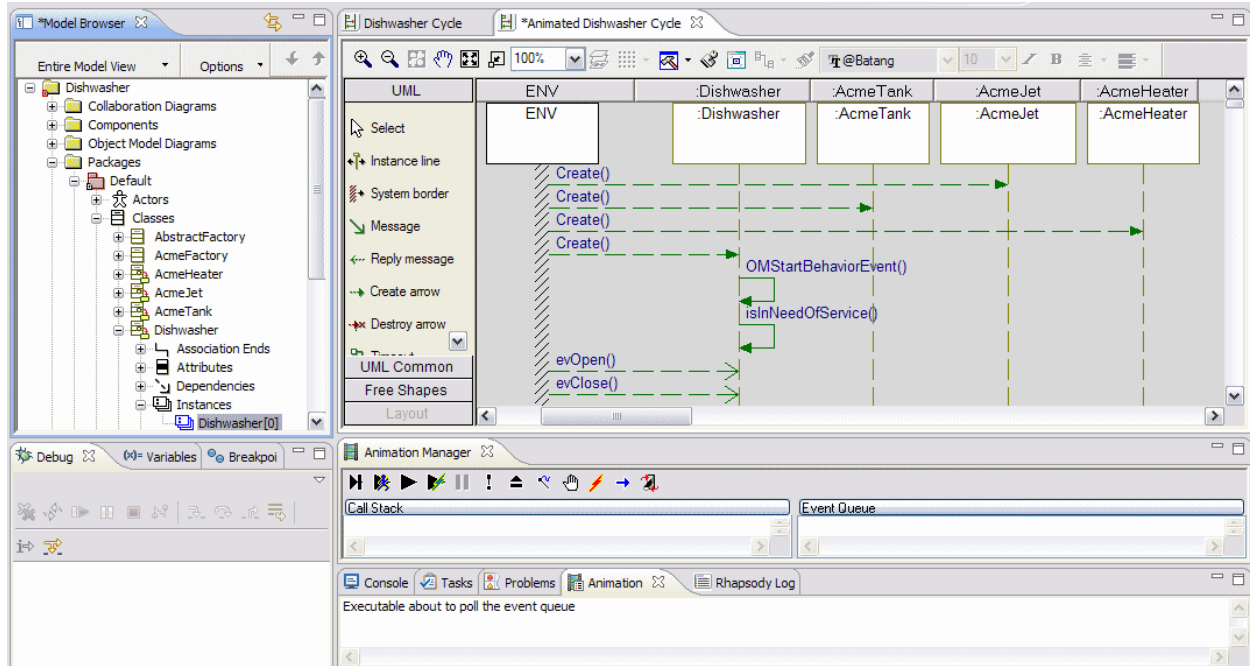
1. In the model browser for the project, expand the `Components` folder.
2. Right-click the configuration you want to animate in the configuration folder and open the **Features** window.
3. On the **Settings** tab, in the **Instrumentation Mode** box, select **Animation**, as shown in the following figure:



4. Click **Apply**.
5. Right-click the configuration you want to animate in the configuration folder and select **Generate Configuration**.

Running Animation

Eclipse displays the Animation Manager windows and toolbar (at the top of the Animation Manager window, as shown in the following figure) to perform all of the Rhapsody standard animation tasks, watch the animation, and note the messages on the Animation Log tab.



Debugging Animated Applications

You may want to use the animated applications for debugging. The Java example below, shows the moment that a breakpoint is hit. The breakpoints are listed in the lower left window with the application output to the right.

The screenshot shows the following components:

- Model Browser:** Displays the project structure, including 'Operations' like `changeCode()`, `evKey(int n)`, `evKeyOff()`, `evKeyOn()`, `isCodeCorrect() const`, `isCodeEntered() const`, and `Keypad()`.
- Keypad.java:** Shows the implementation of `changeCode()`. A breakpoint is set at line 157, which is `oldCode = newCode;`. The code includes `animInstance().not` calls and a `finally` block.
- StatechartOfKeypad:** A UML statechart diagram. The `idle` state transitions to `reprogramming` on `evKeyOn [isIn(different)]/ changeCode()`. The `reprogramming` state contains `waitOldCode` and `waitNewCode`. Transitions include `evKeyOn [isIn(notEntered)]` and `evKeyOff [isIn(notEntered)]`.
- Animation Manager:** Shows the current call stack and event queue. The call stack is `Call Stack: @GuiHomeAlarm01->itsAlarmController->theKeypad`.
- Breakpoints:** Lists several breakpoints, including `Keypad [line: 157] - changeCode()`, `AnimStepper [entry] - resumeThreadByDebugger`, and `AnimStepper [entry] - suspendThreadByDebugger`.
- Console:** Shows the output of the application, including a message: `Breakpoint (\\JavaHomeAlarm\src\alarmK... Debugger Breakpoint on thread @GuiHor...`

Eclipse Configuration Management

The Rhapsody Eclipse plug-in integrates a Rhapsody model into the Eclipse environment, enabling software developers to streamline their workflow with the benefit of working within the same development environment. You can work in the code or model in a single development environment. This enables you to use Rhapsody's modeling capabilities or to modify the code using the Eclipse editor, while maintaining synchronization between both and easily navigating from one to the other.

Parallel Development

When many developers are working in distributed teams, they often need to work in parallel. These teams use a configuration management (CM) tool, such as ClearCase, to archive project units. However, not all files may be checked into CM during development.

Developers in the team need to see the differences between an archived version of a unit and another version of the same unit that may need to be merged. To accomplish these tasks, they need to see the graphical differences between the two versions, as well as the differences in the code.

A Rhapsody unit is any project or portion of a project that can be saved as a separate file. The following are some examples of Rhapsody units with the file extensions for the unit types:

- ◆ Class (.cls)
- ◆ Package (.sbs)
- ◆ Component (.cmp)
- ◆ Project (.rpy)
- ◆ Any Rhapsody diagram

Note

The illustrations in this section show the use of the ClearCase Eclipse Team plug-in. Different Team plug-ins (for example, Synergy) may have different graphical user interfaces and menus.

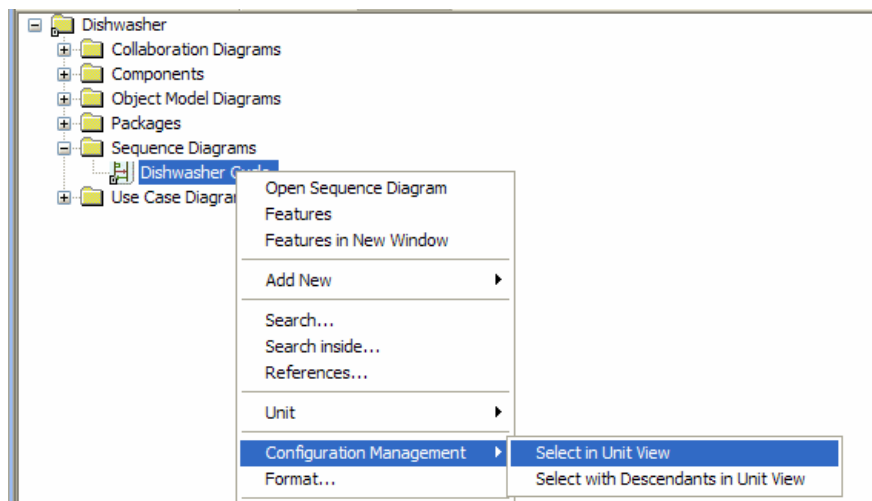
Configuration Management and Rhapsody Unit View

For the Eclipse plug-in, your individual Team plug-ins handle configuration management operations.

The Rhapsody Unit View provides a hierarchical view of Rhapsody model resources as per the model structure. Use this view to perform configuration management operations.

Navigating to the Unit View

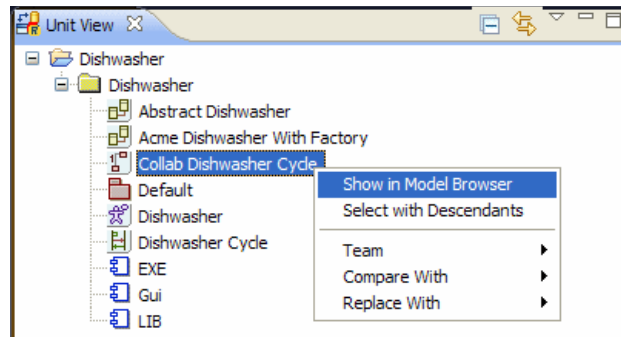
To navigate to the Unit View, on the model browser, right-click an element and select **Configuration Management > Select in Unit View**, as shown in the following figure:



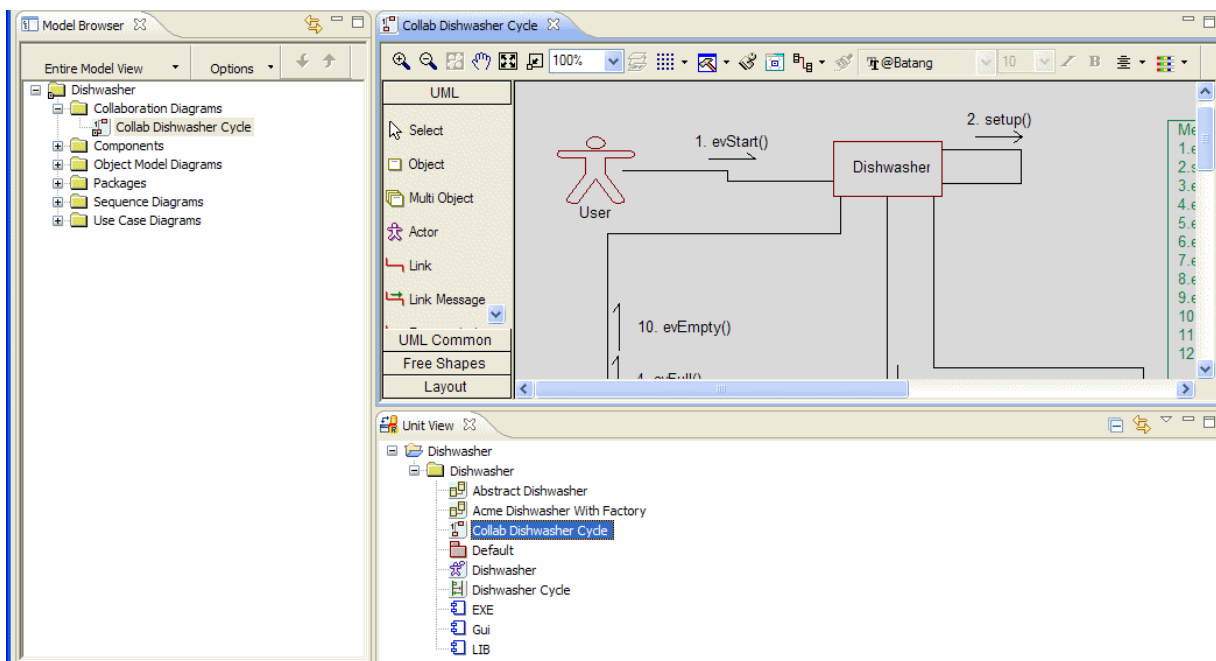
Notice also in the above figure that from the model browser, you can choose **Select with Descendants in Unit View**. This menu command selects all the descendants for Unit View.

Navigating to the Model Browser

To navigate to the model browser from the Unit View, right-click an element and select **Show in Model Browser**, as shown in the following figure:



The following figure shows the result of selecting Show in **Model Browser** in the above example.

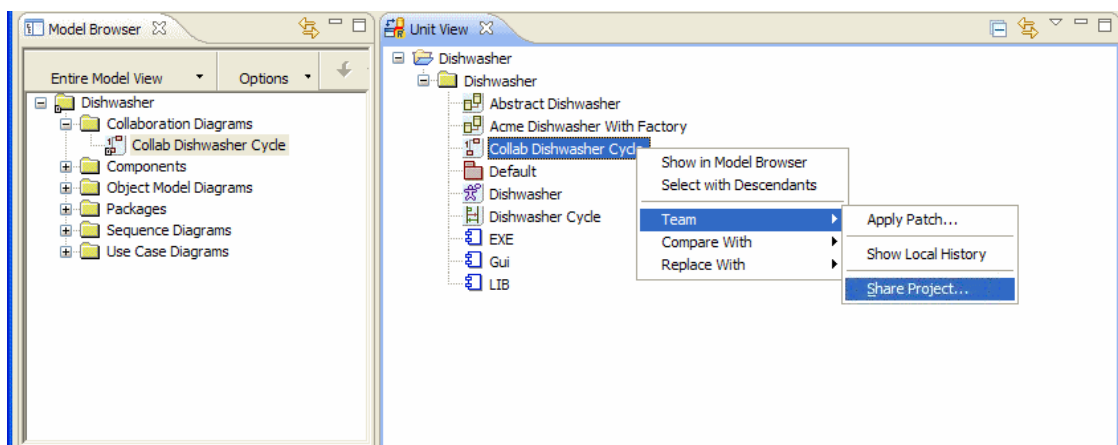


Sharing a Rhapsody Model

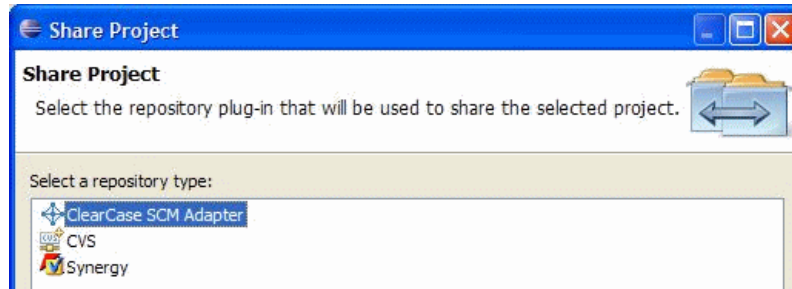
Team members can share a Rhapsody model using configuration management.

To share a Rhapsody model using configuration management, follow these steps:

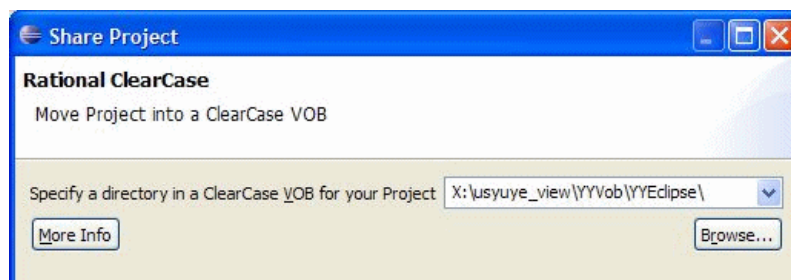
1. If you have multiple Rhapsody models loaded, make whichever project you want to share be the active one.
2. For the active Rhapsody project, at the top-level project node in the Unit View, choose **Team > Share Project**, as shown in the following figure:



3. On the Share Project dialog box, select the repository plug-in that you want to use to share the selected project, as shown in the following figure, and click **Next**.

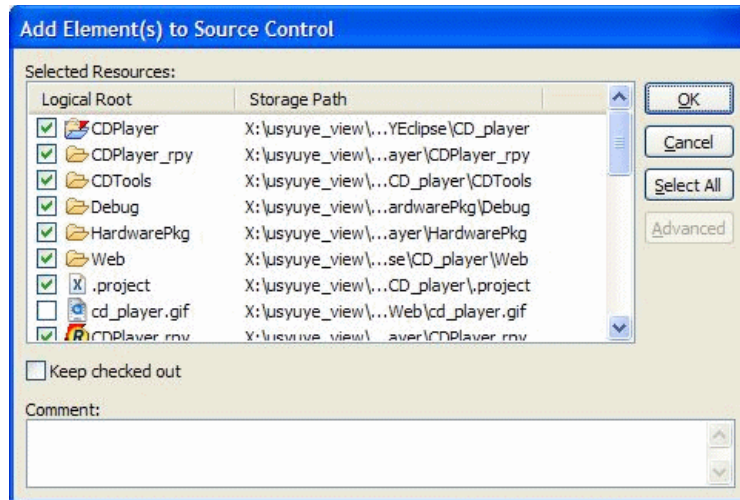


4. On the next dialog box (which shows the name of the configuration management tool you selected in the previous step), enter the required information. For example, if you are using Rational ClearCase, specify a ClearCase VOB for your project, as shown in the following figure:



5. Click **Finish**.

6. Address any other dialog boxes that may open. For example, for ClearCase, decide which elements you want to add to source control by clearing or selecting the applicable check boxes, as shown in the following figure:

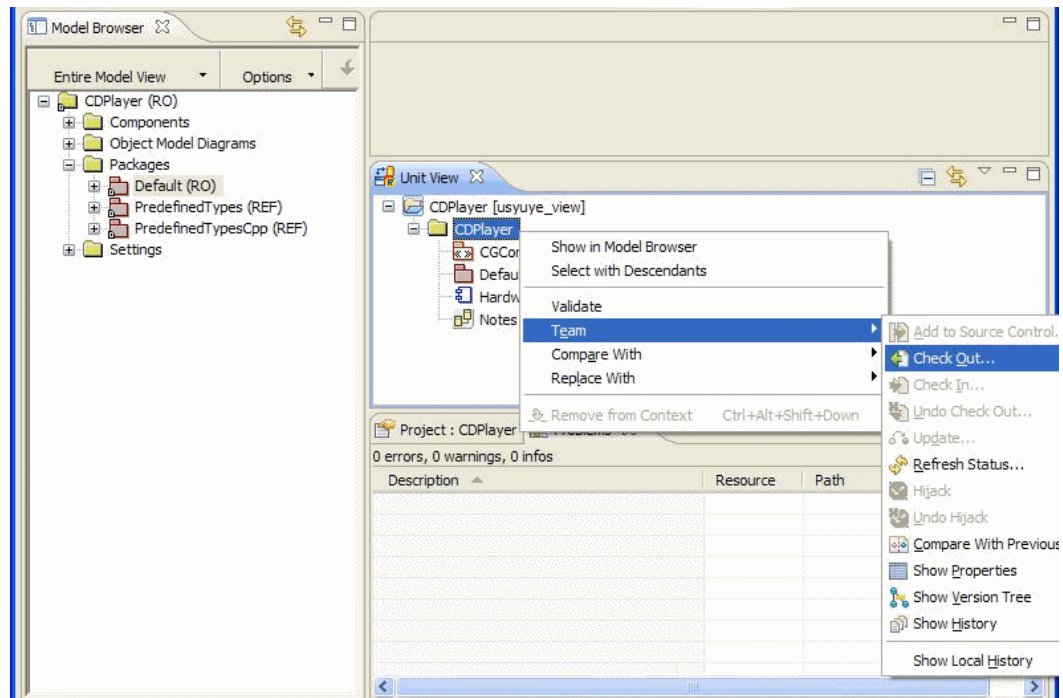


7. Click **OK**.
8. Restart Eclipse.

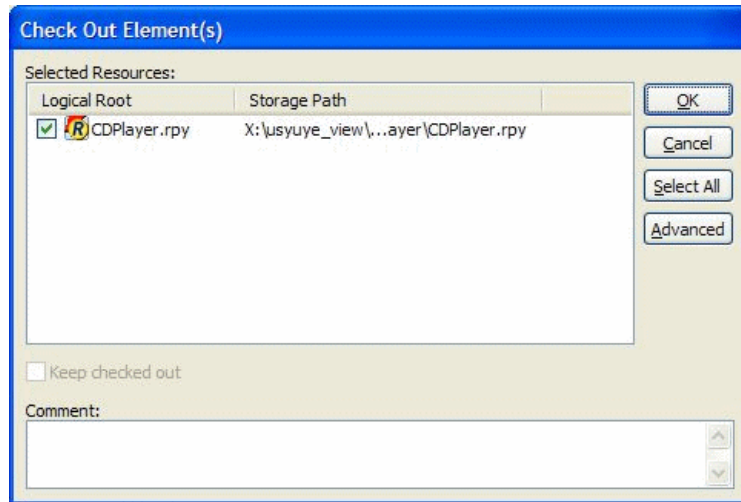
Performing Team (CM) Operations

To perform Team (configuration management) operations, follow these steps:

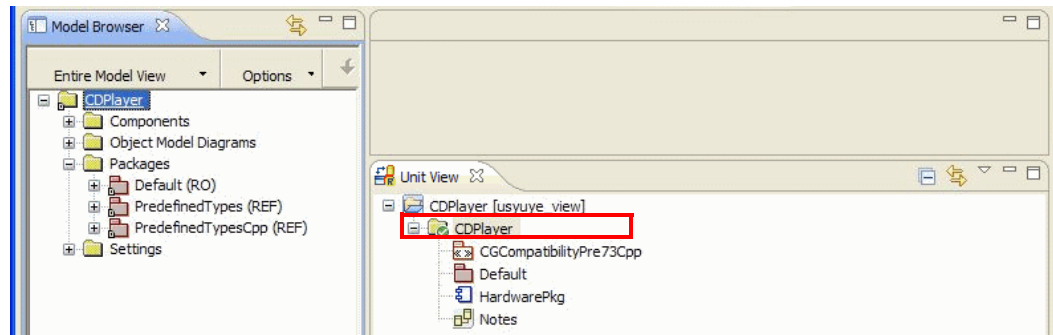
1. Before performing Team operations (such as Check In), save your Rhapsody model in the model browser.
2. For a Rhapsody project in Unit View, right-click the element you want, choose **Team** > [*configuration management operation*]; for example, **Team** > **Check Out**, as shown in the following figure:



- On the dialog box that opens, confirm the element(s) that you want to perform the configuration management operation by clearing or selecting the applicable check boxes, and then click **OK**, as shown in the following figure:

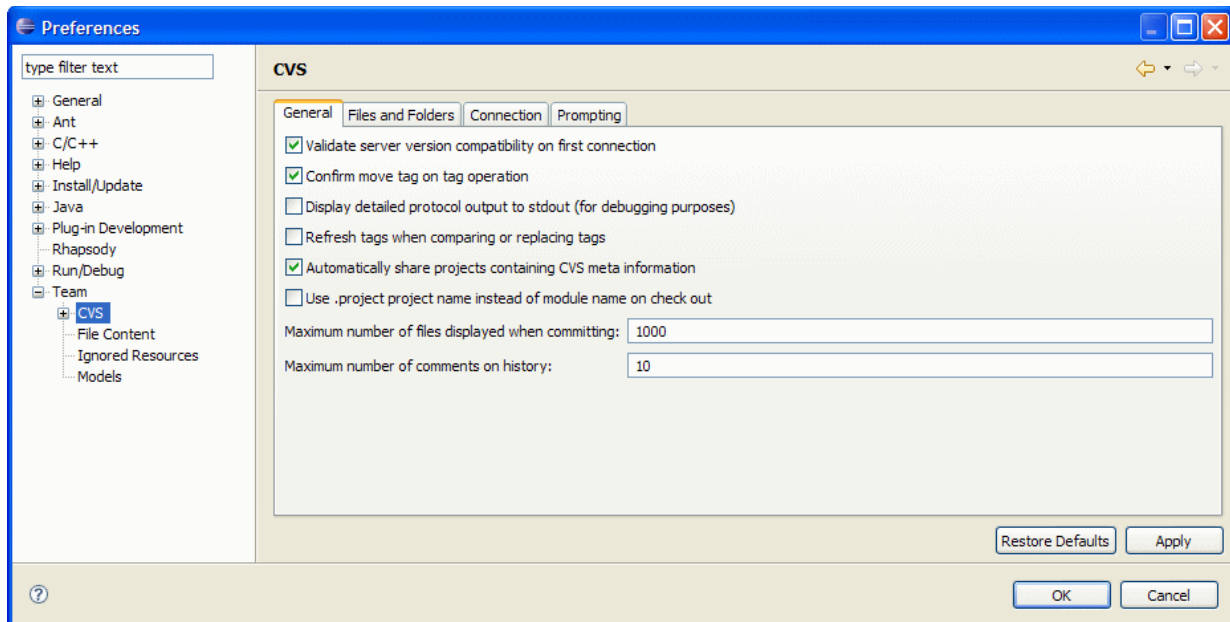


- The following figure shows the element that was checked out from the above steps. Notice the white check mark within a green background that denotes an element that is checked out.



Note

You can set Team plug-in related preferences through the Eclipse Preferences dialog box (choose **Windows > Preferences**), as shown in the following figure:



Rhapsody DiffMerge Facility in Eclipse

The DiffMerge tool can be operated inside and/or outside your CM software to access the units in the repository. It can compare two units or two units with a base (original) unit. The units being compared only need to be stored as separate files in directories and accessible from the PC running the DiffMerge tool.

In addition to the comparison and merge functions, this tool provides these capabilities:

- ◆ Graphical comparison of any type of Rhapsody diagram
- ◆ Consecutive walk-through of all of the differences in the units
- ◆ Generate a Difference Report for a selected element including graphical elements
- ◆ Print diagrams, a Difference Report, Merge Activity Log, and a Merge Report

Refer to the Rhapsody *Team Collaboration Guide* (available from **Help > Rhapsody List of Books**) for detailed DiffMerge instructions.

Generating Rhapsody Reports

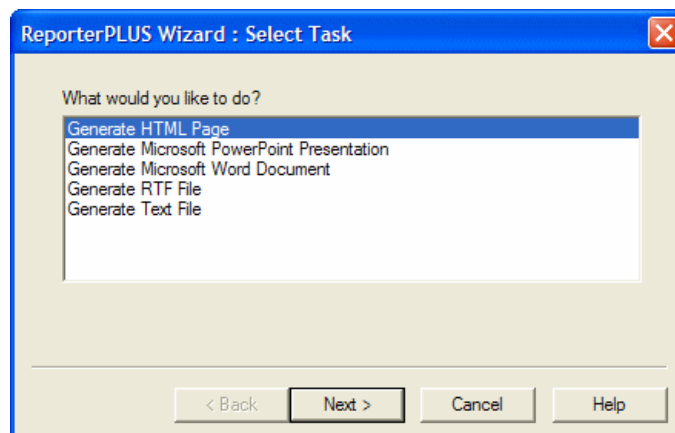
To generate reports from the model, use Rhapsody's ReporterPLUS documentation tool to create Microsoft Word, Microsoft PowerPoint, HTML, RTF, and text documents. The ReporterPLUS pre-defined templates extract text and diagrams from a model, arrange the text and diagrams in a document, and format the document.


Note

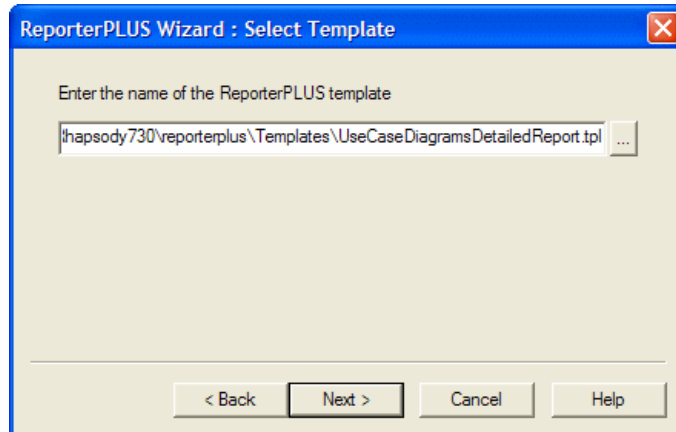
For more information about ReporterPLUS, refer to the *ReporterPLUS Guide* available from **Help > Rhapsody List of Books**.

To generate a report from the Rhapsody model, follow these steps:

1. Choose **Tools > ReporterPLUS > Report on all model elements** or **Report on selected package**.
2. On the ReporterPLUS Wizard, as shown in the following figure, select the output format you want and click **Next**.



3. Use the Browse  button to select a template from the template directory and click **Next**.



4. On the Confirmation dialog box, review the report criteria, and then click **Finish** to produce the report.
5. On the Generate Document dialog box:
 - ◆ Enter a document name.
 - ◆ Browse to where you want to locate the file(s) that will be produced.
 - ◆ Click the **Generate** button to generate your document.
6. Wait while your document is generated. ReporterPLUS spends some time loading the template and the model. Then it analyzes the model and the model element relationships.
7. When available, click **Yes** to open your report.

Working with Projects

A Rhapsody project includes the UML diagrams, packages, and code generation configurations that specify the model and the code generated from it. The term *project* is equivalent to *model* in Rhapsody.

This section provides an overview of a Rhapsody project, the components of a project, and the procedures to create, edit, and store projects.

Project Elements

A project consists of *elements* that define your model, such as packages, classes and diagrams. The browser displays these elements in a hierarchical structure that matches the organization of your model. Rhapsody uses these elements to generate code for your final application.

A Rhapsody project has the following top-level elements:

- ◆ **Components**—Contain configurations and files.
- ◆ **Packages and profiles**—Packages contain other packages, components, actors, use cases, classes (C++/J), object types, events, types (C/C++), functions (C,C++), objects, dependencies, constraints, variables, sequence diagrams, OMDs, collaboration diagrams, UCDs, and deployment diagrams.

A profile “hosts” domain-specific tags and stereotypes.

- ◆ **UML diagrams**—For example, use case, object model, sequence, collaboration, component, and deployment diagrams.

See [Using Model Elements](#) for more information about project elements.

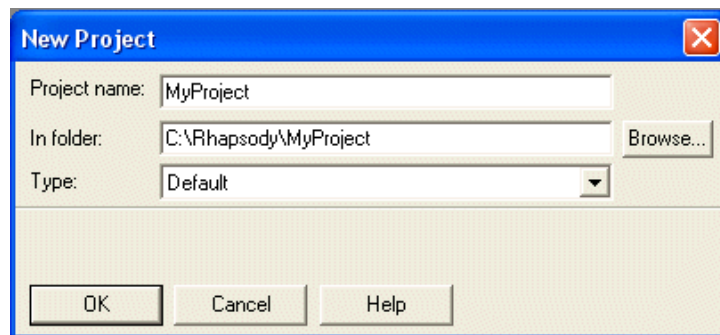
Creating and Managing Projects

When working in Rhapsody, you need to know the basic procedures for using a project. You can create a new project, or work with an existing project. You can edit your project and save the changes. You can create an archive of your project to easily exchange files with another engineer or with customer support. You can have Rhapsody create autosave files to periodically store your unsaved changes, and automatically create backup projects of previously saved versions. This section contains instructions for these and other procedures necessary for using Rhapsody.

Creating a Project

When you create a new project, Rhapsody creates a folder containing the project files in the location you specify. The name you choose for your new project is used to name project files and folders, and appears at the top level of the project hierarchy in the browser. Rhapsody provides several default elements in the new project to get you started, such as a default component and configuration. Before you begin, create a project folder in your file system to hold all of your Rhapsody projects.

1. With Rhapsody running, create the new project by either selecting **File > New**, or clicking the **New project** button on the main toolbar.
2. Replace the default project name (Project) with `<your project name>` in the **Project name** field. Enter a new directory name in the **In folder** field or **Browse** to find an existing directory. Your dialog box should be similar to the following figure:



3. The Default **Type** provides all of the basic UML structures for most Rhapsody projects. However, you can select one of the specialized [Profiles](#), that provide a predefined, coherent set of tags, stereotypes, and constraints for specific project types.
4. Click **OK**. If the directory does not exist, Rhapsody asks if you want to create it. Click **Yes** to create the new project directory.

Rhapsody creates a new project in the `<your project name>` subdirectory and opens the new project. The project name in that directory is `<your project name>.rpy`.

Profiles

A predefined Rhapsody profile becomes part of your project in one of these ways:

- ◆ You select an available profile from the Type pull-down menu, as described in the [Creating a Project](#) section.
- ◆ You manually add a specialized profile to your project from the Share\Profiles directory, as described in the [Adding a Rhapsody Profile Manually](#) section.
- ◆ Rhapsody assigns a starting-point profile based on your project settings and development environment.

The predefined profiles available to you depend on the system language and add-on products licensed for Rhapsody.

- ◆ **AdaCodeGeneration** is the default Ada code generation profile.
- ◆ **AutomotiveC** includes the capabilities provided in the following automotive industry environments:
 - FixedPoint arithmetic
 - MainLoop (no-OS)
 - OSEK21 and Basic/Extended OSEK task stereotypes
 - ExtendedC_OXF

[The AutomotiveC Profile](#) also loads the StateBlock and SimulinkInC profiles. See [Rhapsody's Automotive Industry Tools](#) for more information.
- ◆ **AUTOSAR_20** and **AUTOSAR_21** create automotive components in accordance with the AUTOSAR development process using the ECU, Internal Behavior, SW Component, System, and Topology diagrams. For more information, see [AUTOSAR Modeling](#). The separate AUTOSAR profiles support the related AUTOSAR versions (2.0 and 2.1).

Note: AUTOSAR can only be used in C language projects.
- ◆ **Default** provides all of the basic UML structures for most Rhapsody projects.
- ◆ **DoDAF** is the Rhapsody profile for DoDAF v1.0. See [Rhapsody DoDAF Add-on and Profile](#) for more information.
- ◆ **FixedPoint** profile contains predefined types representing 8, 16, and 32-bit fixed-point variables: FXP_8Bit_T, FXP_16Bit_T, FXP_32Bit_T. See [Defining Fixed-point Variables](#) for more information.
- ◆ **FunctionalC** profile tailors *Rhapsody in C* for the C coder, allowing the user to functionally model an application using familiar constructs such as files, functions, call graphs and flow charts.
- ◆ **Harmony** creates a project based on the Harmony (SE) Systems Engineering Process. See [Harmony Process and Toolkit](#) for more information.

- ◆ **IDFProfile** uses the code generation settings for the RiC IDF (see [Using IDF for a Rhapsody in C Project](#) for more information).
- ◆ **MODAF** is the Rhapsody profile for MODAF v1.1. See [Rhapsody MODAF Add-on](#) for more information.
- ◆ **MISRA98** controls the code generation settings to comply with the MISRA-C 1998 standard.
- ◆ **NetCentric** imports Web-services Definition Language (WSDL) files to design and generate the services model. See [Domain-specific Projects and the NetCentric Profile](#) for more information. (This profile requires a separate license.)
- ◆ **RespectProfile** can be used for C and C++ project to preserve the structure of the code and preserves this structure when code is regenerated from the Rhapsody model. Meaning that code generated in Rhapsody resembles the original. For more information about handling regenerated code, see the [Code Respect and Reverse Engineering for Rhapsody in C and C++](#) section.
- ◆ **RoseSkin** is used by Rose Import to set format and other settings to resemble Rose look-and-feel.
- ◆ **SDL** facilitates importing SDL Suite models into Rhapsody SDL Blocks.
- ◆ **Simulink** and **SimulinkInC** allow integration of MATLAB Simulink models into Rhapsody as Simulink Blocks (*Simulink* profile is for C++).
- ◆ **SPARK** is Rhapsody's Ada SPARK profile.
- ◆ **SPT** (Scheduling, Performance, and Time) is an implementation of the SPT standard (OMG standard) that specifies the method to add timing analysis data to model elements.
- ◆ **StatemateBlock** creates a new block/class allowing a Statemate model to become part of a Rhapsody architecture. This profile is only available for Rhapsody in C and requires a licensed version of Statemate 4.2 MR2 or greater with a license for the Statemate MicroC code generator. For more information, see [StatemateBlock in Rhapsody](#).
- ◆ **SysML** supports both UML and SysML model diagrams for systems engineering. This profile is Rhapsody's implementation of the OMG SysML profile. See [Systems Engineering with Rhapsody](#) for more information.
- ◆ **TestingProfile** is an implementation of the OMG Testing Profile. The TestingProfile is for use with Rhapsody TestConductor. For more information about this profile, see the documentation provided for Rhapsody TestConductor (for example, with Rhapsody open, choose **Help > List of Books** and then click the TestConductor User Guide link).

Opening a Rhapsody Project

To open an existing Rhapsody project with all units loaded, follow these steps:

1. Select **File > Open**, or click the **Open** button on the standard toolbar. The Open dialog box is displayed.

Note: To open one of the last-opened projects, select it from the list of projects that appear on the **File** menu just above the **Exit** command.

2. In the **Look in** field, browse to the location of the project.
3. Select the `.rpy` file, or type the name of the project file in the **File name** field.
4. Select the **With All Subunits** check box. This causes Rhapsody to load all units in the project, ignoring workspace information. For information on workspaces, see [Using Workspaces](#).
5. Click **Open**. The entire Rhapsody project opens.

Search and Replace Facility


Engineers and developers can use Rhapsody's Search and Replace facility for simple search operations and to manage large projects and expedite collaboration work. The search results display in the [Output Window](#) with the other tabbed information.

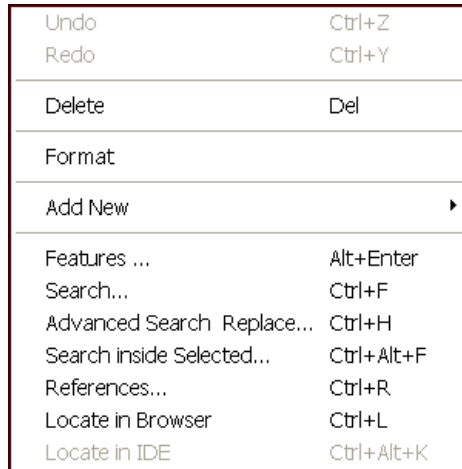
This facility provides the following capabilities:

- ◆ Perform quick searches
- ◆ Locate unresolved elements in a model
- ◆ Locate unloaded elements in a model
- ◆ Identify only the units in the model
- ◆ Search for both unresolved elements and unresolved units
- ◆ Perform simple operations on the search results
- ◆ Create a new tab in the Output window to display another set of search results
- ◆ For more detailed instructions for the Search and Replace facility, see [Searching in the Model](#).

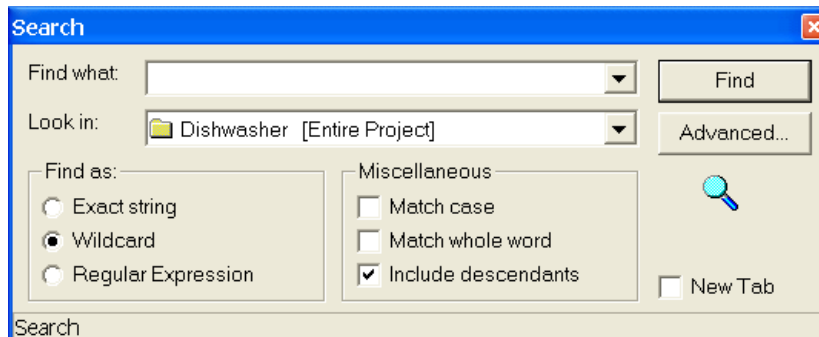
Searching Models

To search models, follow these steps:

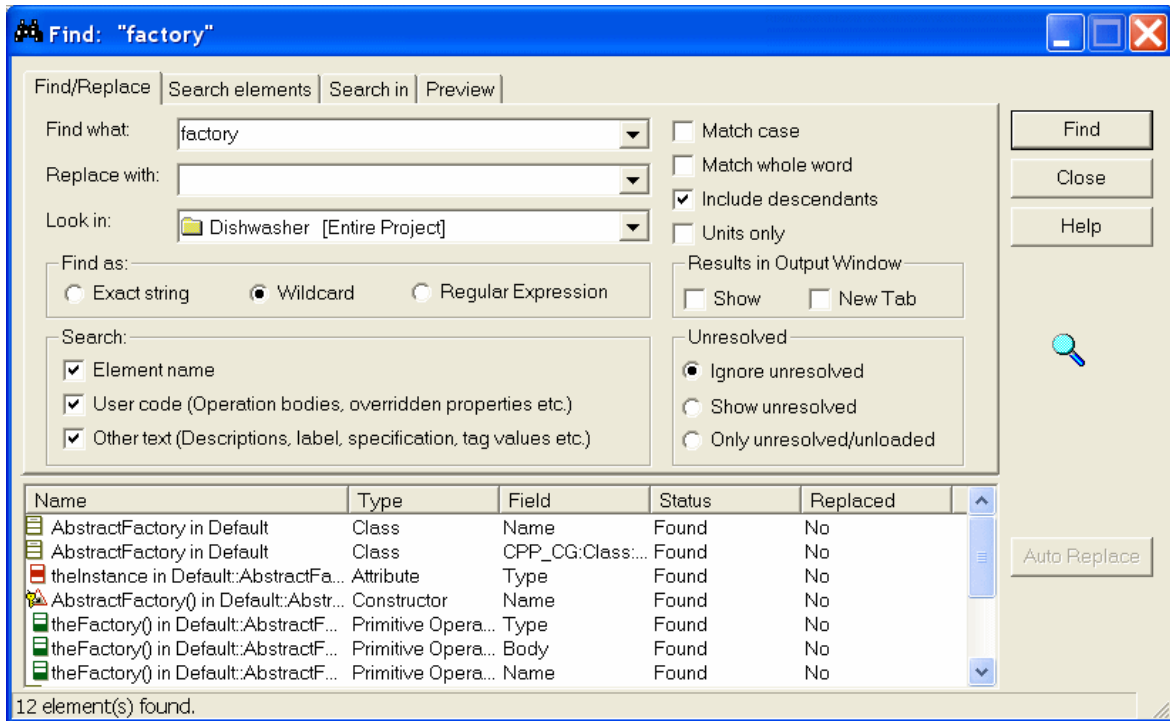
1. With the model displayed in Rhapsody, there are three methods to launch the Search facility: select the **Edit** menu (as shown in the following figure) and then select the **Search** option, click the binoculars button , or press **Ctrl + F**.



2. The Search dialog box (as shown in the following figure) and perform a quick search. Type the search criteria into the **Find what** field and click **Find**. The results display in the Output window. The search criteria displays on the Output window's **Search** tab.



- To display the more detailed search dialog box, select **Edit > Advanced Search Replace** or click the **Advanced** button in the Search dialog box (above). Both methods display this dialog box. The advanced search dialog box (as shown in the following figure) provides the Unresolved and Units only search features.



- The advanced search capabilities include the following:
 - Exact string** permits a non-regular expression search. When selected the search looks for the string entered into the search field (such as char*)
 - Wildcard** permits wildcard characters in the search field such as "*" produces results during the search operation that include additional characters. For example, the search **dishwasher* matches class *dishwasher* and attribute *itsdishwasher*.
 - Regular Expression** allows the use of Unix style regular expressions. For example, *itsdishwasher* can be located using the search term **dishwasher*.
- If after performing one search you want another **Search** tab with additional search results displayed in the Output window, check the **New Tab** box in the **Results in Output Window** area. Perform the next search.

Working with Search Results

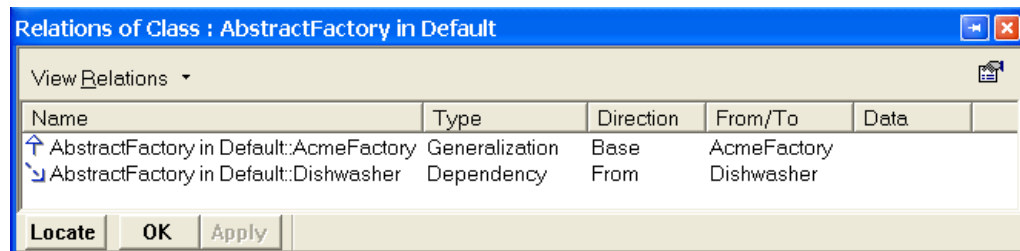
After locating elements using the Search facility, you can perform these operations in the Search dialog box or in the Output window:

- ◆ Sort items
- ◆ Check the references for each item
- ◆ Delete
- ◆ Load

To sort items in the list, click the heading above the column to sort according to display that feature of each item in the list.

To examine the *references* for an item in the search results, follow these steps:

1. Highlight an item in the search results list.
2. Right-click to display the pop-up menu.
3. Select **References** and examine the information displayed in the dialog box, as shown in the following figure:



To delete an item located in search process, follow these steps:

1. Highlight an item in the search results list.
2. Right-click to display the pop-up menu.
3. Select **Delete from Model**. The system displays a message for you to confirm the deletion.
4. Click **Yes** to complete the deletion process.

If you have located an unloaded item in the search results and want to load it into the model, right-click the item. Load the item in the same manner as it is loaded from within the browser.

Replacing

If you want to replace item names or other terminology throughout the model, follow these steps:

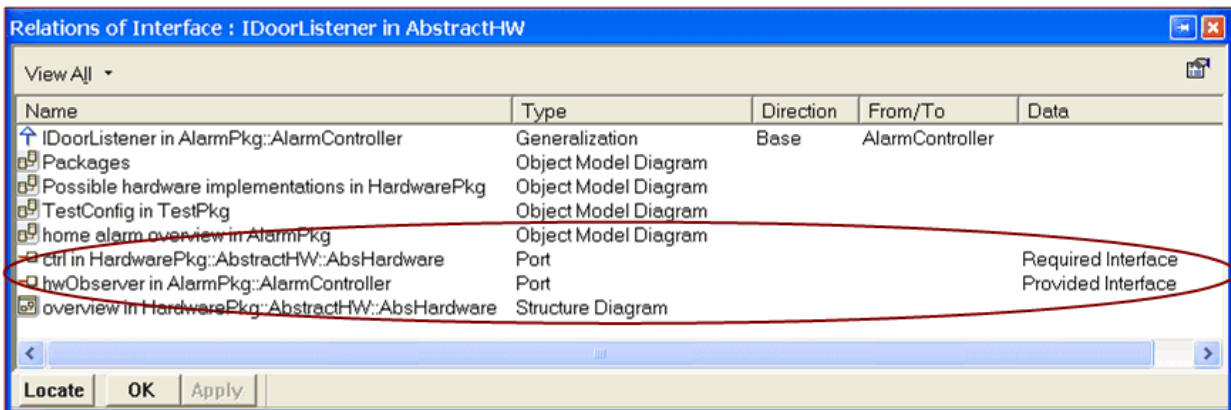
1. Display the **Advanced Search**.
2. Enter the current terminology in the **Find what** field.
3. Enter the new terminology into the **Replace with** field.
4. Make any additional selections to limit the search and replace process.
5. Click **Find** and approve or skip the possible replacements.

Locating and Listing Specific Items in a Model

During development you may need to examine a specific feature or create a list of items in the model. This is one of the situations when the Rhapsody [Application Accelerators](#) keys are useful.

In this example, the developer wants a list of the ports that are used by a specific interface. To display a list of specific items in a Rhapsody model, follow these steps:

1. Display the model in Rhapsody.
2. If you need to locate the section you want to examine, you can use [Search and Replace Facility](#) to find it. Then in the browser, highlight the item for which you need more information.
3. In this example, press the **Ctrl-R** (for relationships) accelerator keys to list the relations with the `IDoorListener` interface.
4. You can sort the displayed list by the **Name** or **Type** by clicking the heading of the column. In this example, the items are sorted by Type to show the ports grouped together.



Note

You can also use ReporterPLUS to generate a report for a selected section of your model. Refer to the *Rhapsody ReporterPLUS Guide* for more information.

Editing a Project

You build projects in Rhapsody by creating and defining elements using either the browser or the graphic editors. Elements include components, packages, classes, operations, events, diagrams, and so on.

Adding Elements

To add elements from the browser, follow these steps:

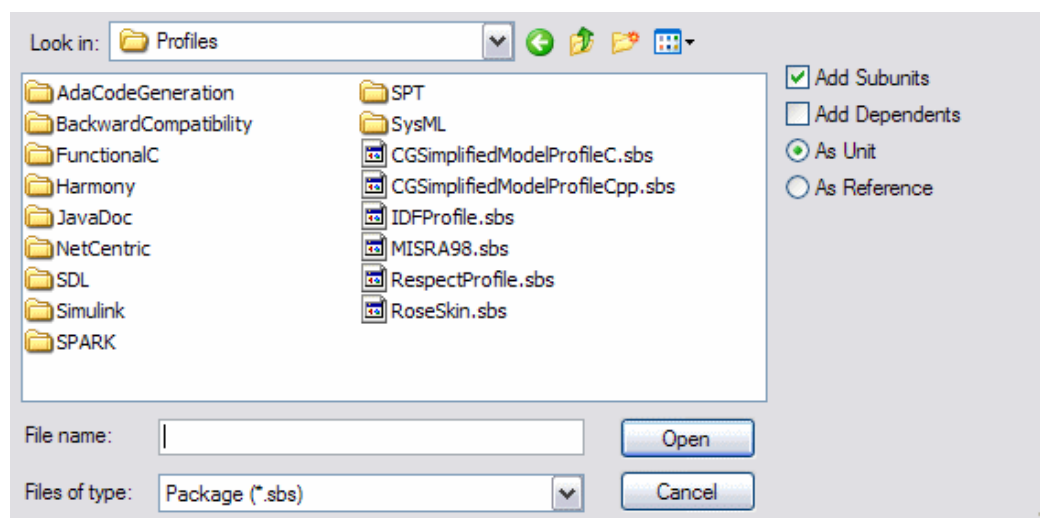
1. Right-click an element, then select **Add New** from the pop-up menu. A submenu that lists all the elements that can be added at the current location in the project hierarchy is displayed.
2. Select the element you want to add.

For detailed information about adding elements from the browser, see [Using Model Elements](#). To add elements from a graphic editor, use the drawing tools to draw the element in the diagram. For information on using the graphic editors, see [Graphic Editors](#).

Adding a Rhapsody Profile Manually

To add another predefined Rhapsody profile manually to your project, open the existing project and follow these steps:

1. Select the **File > Add to Model** option. Navigate to your Rhapsody installation directory and open the Share/Profiles folder.
2. In the Add to Model dialog box, change the **Files of Type** field to display `Package` (`*.sbs`), as shown in the following figure. This lists all of your available profiles either as separate `.sbs` files or in folders.



3. Select a profile's `.sbs` file and click **Open**. The system checks to be certain that the selected profile is compatible with the language being used in the existing project.

Rhapsody lists the newly added profile in the `Profiles` section of the browser. If you are not familiar with the profile, open the profile in the browser to examine its characteristics.

Editing in the Features Dialog Box

Each element in the project has features that can be edited using the Features dialog box. An element's features include things like its name, description, type, and implementation code.

- ◆ Double-click the element.
- ◆ Right-click the element, then select **Features** from the pop-up menu.
- ◆ Select an element in the browser, then type **Alt + Enter**.

For more information on using the Features dialog box, see [Using the Features Dialog Box](#).

Undo and Redo

Rhapsody allows you to undo the last 20 operations, and to redo the operation that was most recently undone.

To undo the last operation, do one of the following:

- ◆ Select **Undo** from the Edit menu.
- ◆ Click the **Undo** tool.

To redo an operation, do one of the following:

- ◆ Select **Redo** from the Edit menu.
- ◆ Click the **Redo** tool.

The **Undo** option does not become active until you perform at least one operation that can be undone. Similarly, the **Redo** option is not active until you have used the **Undo** command at least once.

By default, Rhapsody allows you to undo the last 20 operations, but you can set this value in the property `General::Model::UndoBufferSize`. Setting this property to a value of zero disables the **Undo/Redo** feature.

You cannot use the **Undo** command after large operations that affect the file system. The undo operation buffer is cleared and the **Undo** and **Redo** tools are deactivated. The following operations **cannot** be undone:

- ◆ Saving, opening, or closing a project
- ◆ Automatic diagram layout
- ◆ Roundtripping code with the “generated code in browser” option
- ◆ Loading a unit into a workspace
- ◆ Configuration management operations
- ◆ Importing Rose models
- ◆ Reverse engineering
- ◆ Adding a file unit to the model (via the **File > Add to Model** command)
- ◆ Code generation with the “generated code in browser” option


Using IDF for a Rhapsody in C Project

For some systems developed using Rhapsody in C, the OXF provided with Rhapsody is not appropriate because the systems require a solution with a smaller footprint. To provide a solution for these environments, a limited framework called IDF (Interrupt-Driven Framework) is also provided with Rhapsody.

Rhapsody provides a base IDF model that can be adapted for different target systems. Also included is a sample adaptation for *Microsoft NT*, which illustrates the use of the IDF.

To use this sample model to learn about the IDF, follow these steps:

1. Start the development version of Rhapsody in C and select **File > Open** and browse to locate the IDF model `Share\LangC\idf\model\idf.rpy`.

2. With that project open, click the GMR button  to generate and make the generic configuration automatically displayed above the window, as shown here.



This generates all core files and `idfFiles.list` with dependencies and rules in the `Share\LangC\idf` directory.

3. Using the same method, open another IDF model, `Share\LangC\idf\Adapters\Microsoft\MicrosoftNT`.
4. Generate and make this model with the GMR button. This builds the library `msidf.lib` in the directory `Share\LangC\lib`.

Note

For more information about IDF and using these models, refer to *Using Rhapsody's Interrupt-Driven Framework (IDF) White Paper* available from the Rhapsody Help menu (choose **Help > List of Books**).

Saving a Project

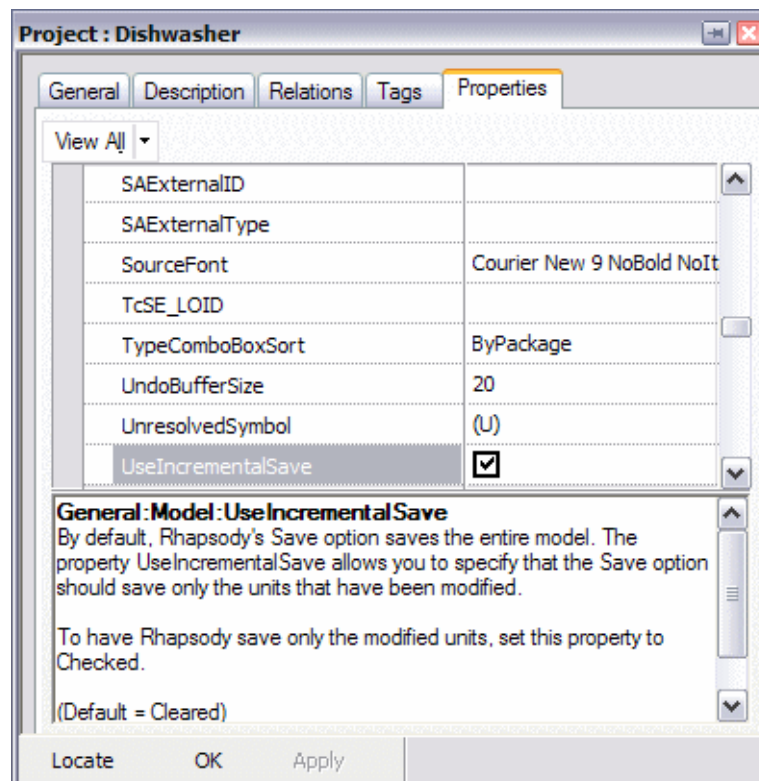
You can save a project in its current location using the **File > Save** option or click the **Save** button in the toolbar. The **Save** command saves all modified files in the project repository. To save the project to a new location with the **Save As** option, follow these steps:

1. Select **File > Save As**. The Save As dialog box opens.
2. Use the **Save In** field to locate the folder where you would like to save the project.
3. Type a name for the project file in the dialog box. The file extension `.rpy` (for *repository*) denotes that the file is a Rhapsody model.
4. Click **Save**. All project files are saved in the project repository.

Incremental Save

If you want to save only the modified project units and not the entire project, follow these steps:

1. Select **File > Project Properties** and select the **Properties** tab.
2. Navigate to the `General::Model::UseIncrementalSave` property. This property should be checked, as shown in the following figure, to use the incremental save feature.



Autosave

By default, Rhapsody performs an incremental autosave every 10 minutes to back up changes made between saves. Modified units are saved in the autosave folder, along with any units that have an older time stamp than the project file. Modifications to the project file are saved in the `<Project>_auto.rpy` file. When you save the project, the autosave files are cleared.

The autosave feature saves files in flat format. All unit files reside in the `<Project>_auto_rpy` folder, regardless of the directory structure of the original model.

To change the autosave interval, use the `General::Model::AutoSaveInterval` property. The interval is measured in minutes. To disable the autosave feature, set the `AutoSaveInterval` property to 0.

Renaming a Project

To change the name or location of the project, use the **Save As** command. Do not attempt to edit the project file directly. When you save the project under a different name, the name of the project folder is updated in the browser.

Closing a Project

1. Select **File > Close**.
2. If you have unsaved changes, Rhapsody asks if you would like to save your changes before closing the project. Select one of the following options:
 - ◆ **Yes**—Save the changes.
 - ◆ **No**—Discard the changes.
 - ◆ **Cancel**—Cancel the operation and return to Rhapsody.

To exit from Rhapsody, select **File > Exit**.

Note

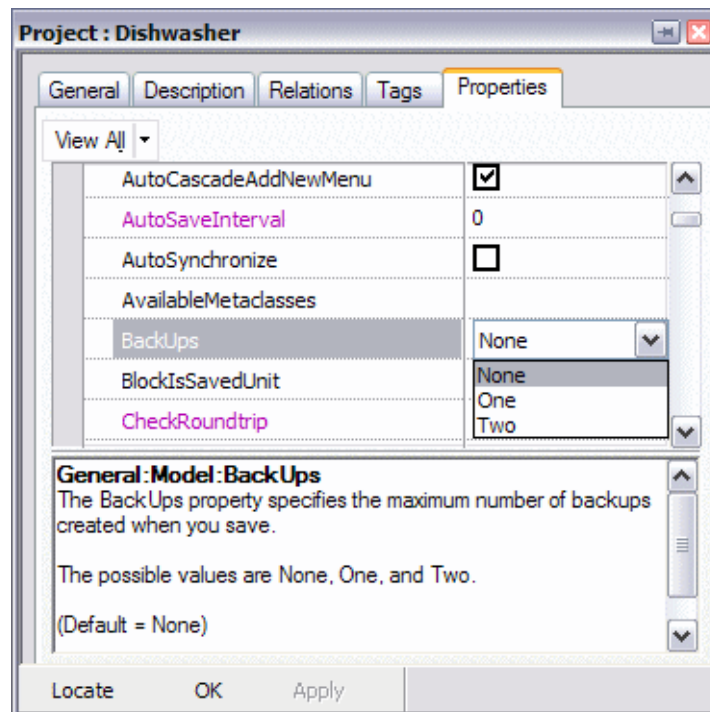
To close all the diagrams opened for a project, choose **Window > Close All**.

Creating and Loading Backup Projects

Rhapsody can create backups of your model every time you save your project, allowing you to revert to a previously saved version if you encounter a problem. To use the automatic backup feature, set the `General::Model::BackUps` property to the number of backup projects you want Rhapsody to create. The options are `None` (default), `One`, and `Two`. Leave the default value of `None` if you do not want Rhapsody to create backups.

To set up automatic backups for your project, follow these steps:

1. In the browser, right-click the `<project name>` at the top of the browser list.
2. Select **Features** from the pop-up menu.
3. Click the **Properties** tab at the top of the dialog box that then displays.
4. Click the **All** radio button to display all of the properties for this project.
5. Expand the **General** and **Model** property lists and locate the **BackUps** property (below).



6. Select **One** or **Two** from the pull-down menu. With this setting, Rhapsody creates up to one or two backups of every project in the project directory.
7. Click **OK** to save this property change.

After this change, saving a project more than once creates `<projectname>_bak2.rpy` contains the most recent backup and the previous version in `<projectname>_bak1.rpy`. To *restore* an earlier version of a project, you can open either of these backup files.

1. Open the `<Project>_bak2.rpy` (or the `<Project>_bak1.rpy` file) in Rhapsody. Do not try to rename the backup file directly.
2. Save the project as `<Project>.rpy` using the **File > Save As** command.

Archiving a Project

At times, you might need to archive a project to send it to another developer or to Customer Support. To create a complete archive, include the following files and directories:

- ◆ `<Project>.rpy` file
- ◆ `<Project>_rpy` directory with all subdirectories
- ◆ `<Component>` directories
- ◆ Any external source files (`.h` and `.cpp`) needed to compile the project

Creating Table and Matrix Views

Rhapsody provides these additional methods to view model data:


- ◆ *Table view* performs a query on a selected element type and displays a detailed list of its various attributes and relations.
- ◆ *Matrix view* displays queries showing the relations among selected model elements.

These views provide the following development capabilities:

- ◆ Define and run dynamic queries of model content
- ◆ Provide easy display and analysis of requirements
- ◆ Produce exportable and printable tables and data lists

Basic Method to Create Views from Layouts

To create either the table or matrix view, follow these general steps:

1. Highlight any area of the model in the Rhapsody browser where you want to store a table or matrix layout (query design).
2. Define a *layout* for the table or matrix (as described in the following sections) and save it in the selected browser location.
3. Define a *view* of the model data using the previously defined layout. This view allows you to also define the *scope* of the view's query.
4. In the browser, double-click the defined view to display the results of the query in the drawing area.
5. To edit the data displayed in a view, use the view's Features dialog box and click the **Refresh**  button to update the view.

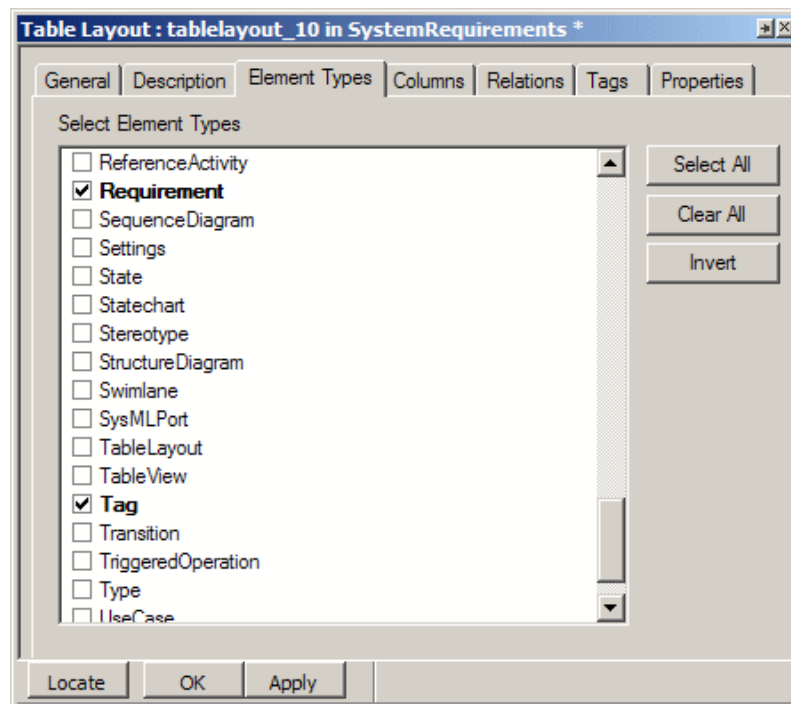
Note: Data displayed in views cannot be edited directly in a view. You must use a view's Features dialog box to make your edits.

6. To export the data, right-click the data in the drawing area and select **Copy**.

Creating a Table Layout

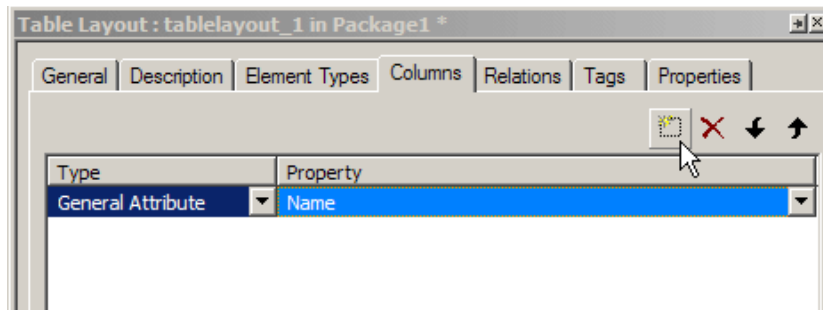
To design the structure for your model query as a table layout, follow these steps:

1. Right-click the package in the Rhapsody browser where you want to create and store your table layout and select **Add New > Table Layout**.
2. In the browser, enter a name for this table design.
You may want to include the word “layout” in the name to help identify your defined layouts from their generated views.
3. Double-click the new layout in the browser to open its Features dialog box.
4. On the **Element Types** tab, as shown in the following figure, select the element type(s) you want to be displayed in the table:

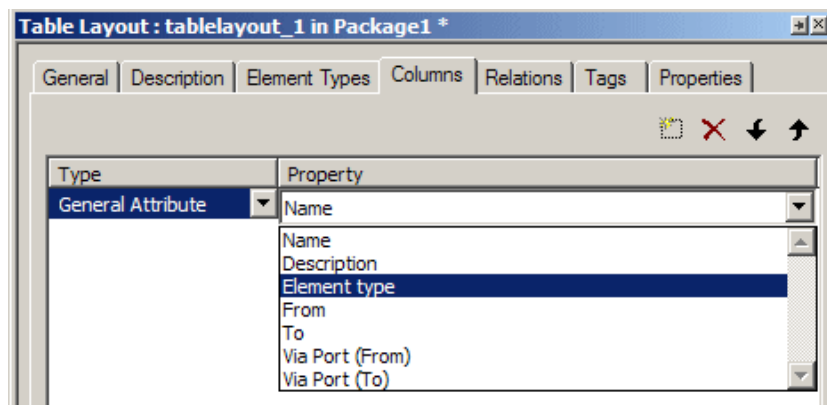


5. Click **Apply** to save your selections without closing the Features dialog box.

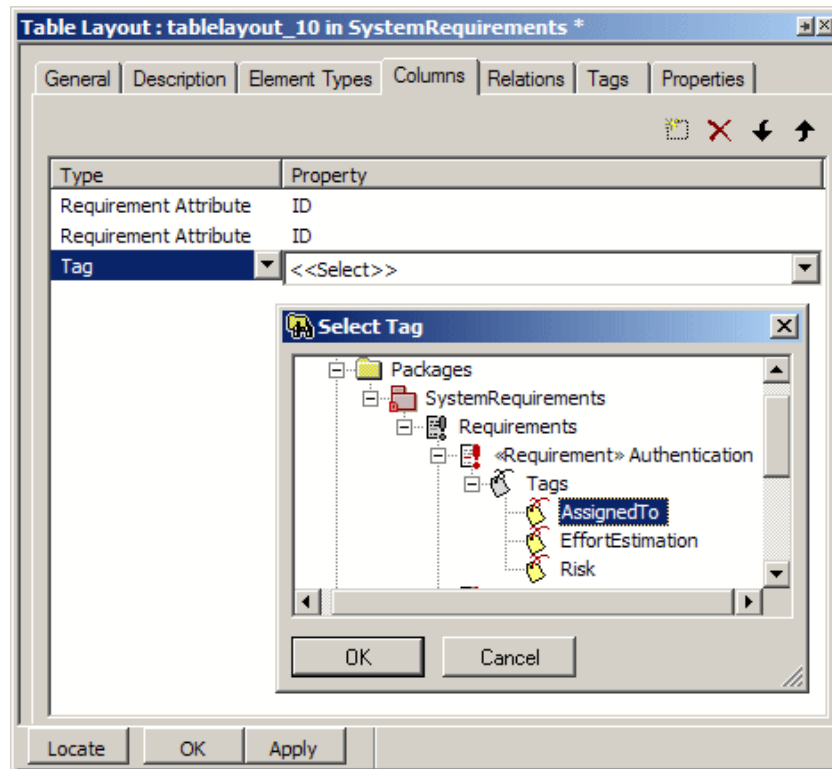
6. On the **Columns** tab, click the **New**  button to create a new row in the table layout, as shown in the following figure:







7. For the row, select a **Type** and **Property** from the corresponding drop-down menus for your query purpose:
 - ♦ The **General Attribute** type may use one of the following properties (also shown in the following figure) to define it:
 - **Name** displays the name of an element.
 - **Description** displays the description for an element (if there is one).
 - **Element type** displays the element type of an element.
 - **From** displays where an element is from.
 - **To** displays where an element goes to.
 - **Via Port (From)** displays the port from which a relation is connected. Typically used along with **Via Port (To)**. For more information, see [Including Ports and Multiple Relations](#).
 - **Via Port (To)** displays the port to which a relation is connected. Typically used along with **Via Port (From)**. For more information, see [Including Ports and Multiple Relations](#).



- ◆ The **Requirement Attribute** type may have either an **ID** or **Specification** property.
- ◆ The **Flow Attribute** type may have only **Flow items** as its property.
- ◆ The **Tag** type has the <<Select>> property. Click it to open the Select Tag dialog box in which you can identify the information for the tag, as shown in the following figure. Click **OK** to save the tag selection.

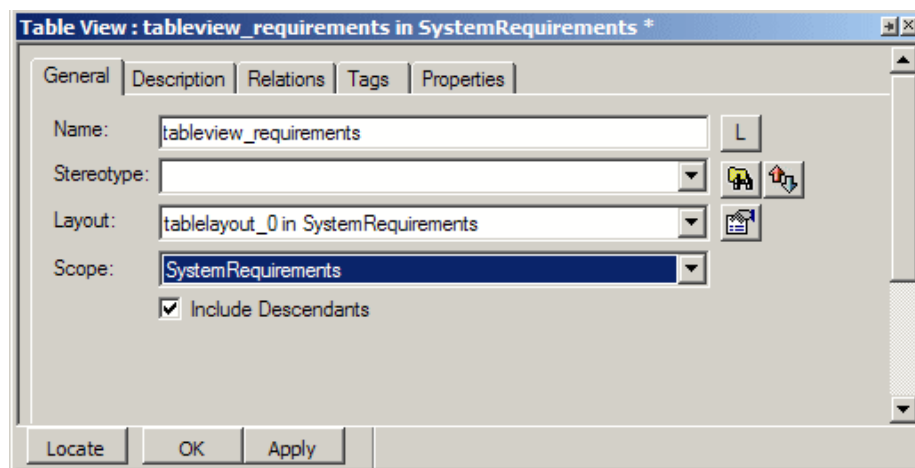


8. Click the **New**  button to add each row for your table. Use the Move Item Up and Move Item Down buttons   to arrange the order of the rows in your table layout.
9. Optionally, to remove a row from the layout, select it on the **Columns** tab and click the Delete button .
10. When you have completed your design layout, click **OK** to save your work and close the Features dialog box.

Creating a Table View

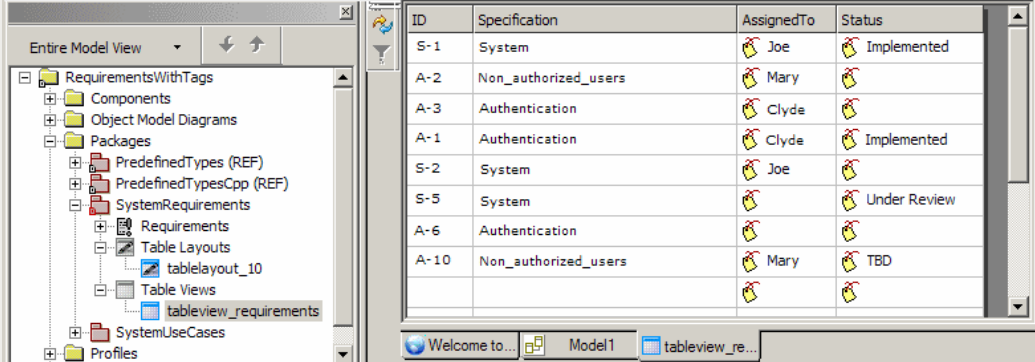
After you have created one or more table layouts, follow these steps to generate a table view of the model data based on your layout design:

1. Right-click a package on the Rhapsody browser to which you want to add a table view and select **Add New > Table View**.
2. In the browser, right-click the new table view and select **Features** to open its Features dialog box.
3. Enter a **Name** for the view.
4. Select the name of a previously created table layout from the **Layout** drop-down list.
5. Select the scope for this view by selecting a package from the **Scope** drop-down list, as shown in the following figure:




6. Clear or select the **Include Descendants** check box, as shown in the figure above, depending on if you want to exclude or include the descendants for the selected scope. For more information, see [Including/Excluding Descendants](#).
7. Click **OK** to save your table view.

- In the browser, double-click the table view name to generate the results of the data query. The query analyzes data for the selected package and all of its nested packages. The following figure shows a table view of requirements with the AssignedTo and Status tags.



ID	Specification	AssignedTo	Status
S-1	System	Joe	Implemented
A-2	Non_authorized_users	Mary	
A-3	Authentication	Clyde	
A-1	Authentication	Clyde	Implemented
S-2	System	Joe	
S-5	System		Under Review
A-6	Authentication		
A-10	Non_authorized_users	Mary	TBD

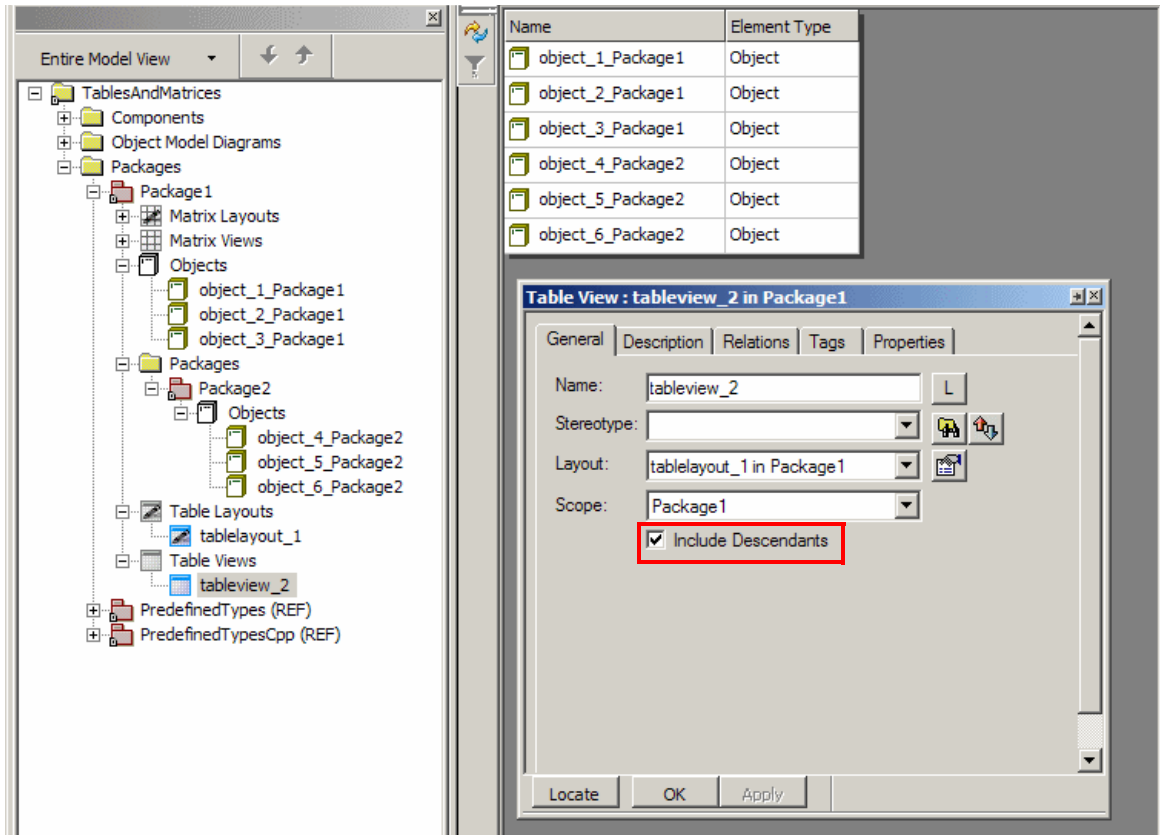
- To make changes to the displayed data, double-click a row in the table view and make the changes in the Features dialog box for that element.
- Depending on if you are done making changes:
 - Click **Apply** to save your changes in the Features dialog box and then click the **Refresh**  button to display the new data in the view.
 - If you are done making changes, click **OK** to close the Features dialog box.

Note

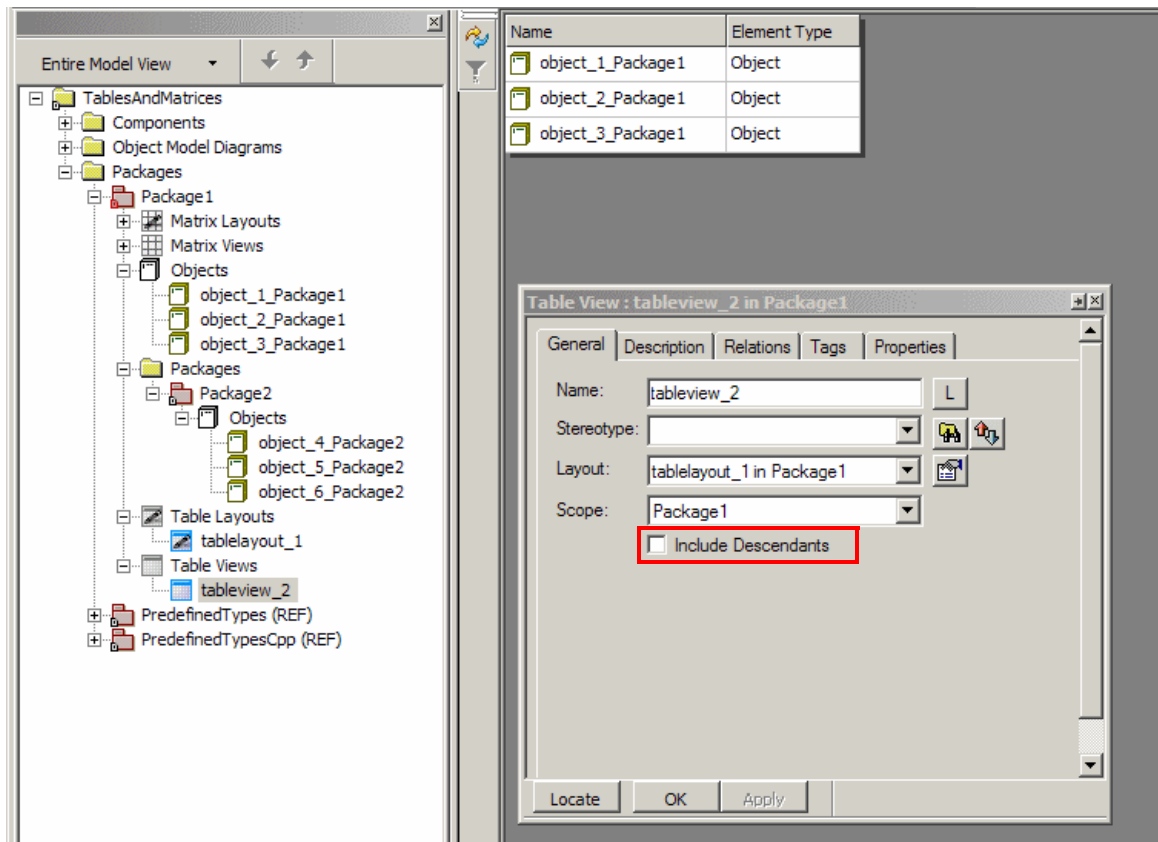
You can also return to the table layout and make changes to the design and redisplay the data in the view using **Refresh**.

Including/Excluding Descendants

By default, table and matrix views include descendants in their scope because the **Include Descendants** check box on the **General** tab of the table or matrix view's Features dialog box is selected by default. The following example shows a model with a main package (Package1) and a subpackage (Package2) within it. Both packages have objects. In this example, when you produce the table view whose scope is the main package, by default the view shows all descendants.



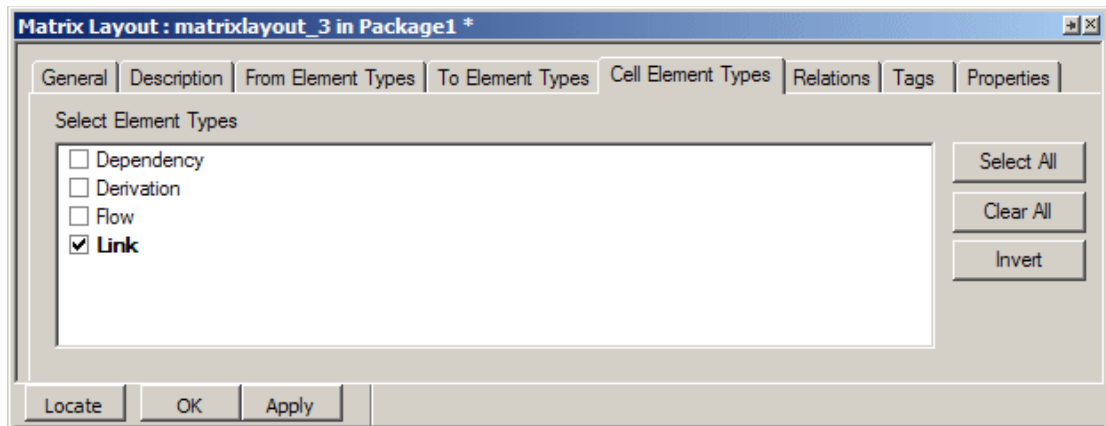
To exclude descendants from appearing in your table or matrix view, you have to clear the **Include Descendants** check box on the **General** tab of the Features dialog box for the particular view, as shown in the following figure for a table view. Then (refresh the view if necessary), the view shows without descendants, as shown in the following figure.



Creating a Matrix Layout

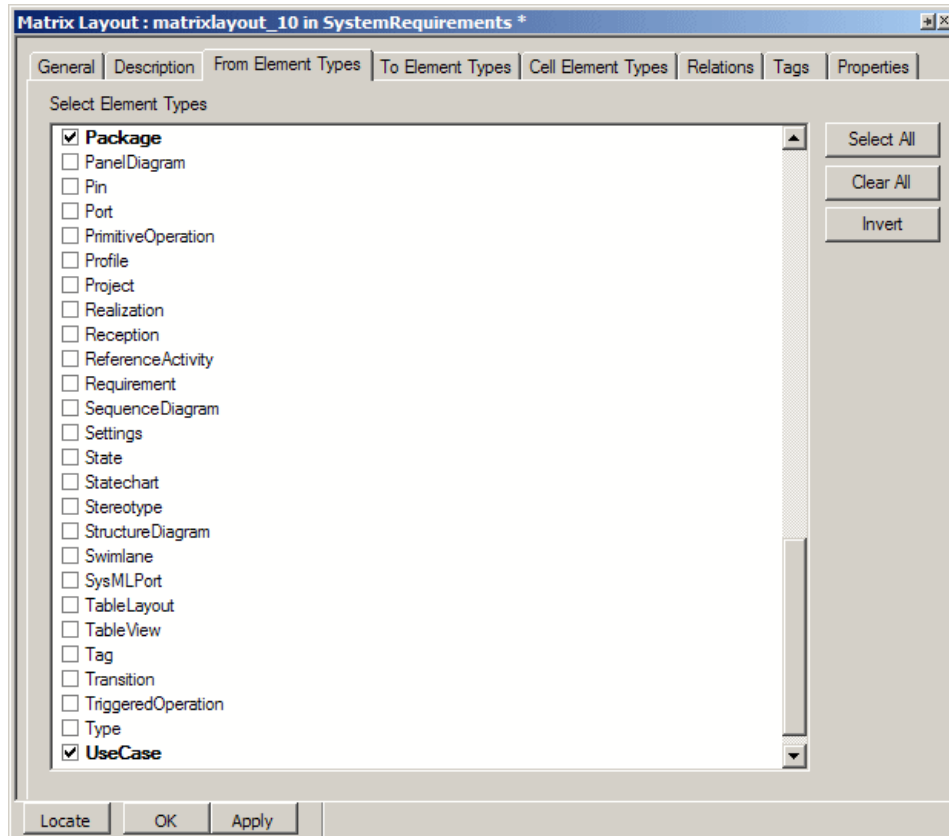
To create a matrix layout to analyze the attributes of model elements, follow these steps:

1. Right-click a package on the Rhapsody browser where you want to add a matrix layout and select **Add New > Matrix Layout**.
2. In the browser, enter a name for this new matrix design.
You may want to include the word “layout” in the name to help identify your defined layouts from their generated views.
3. Double-click the new matrix layout to open its Features dialog box.
4. On the **Cell Element Types** tab select only one of the possible element types. *Cell element type* is the relation type between the “From” and “To” elements to be displayed in matrix cells. Elements of that type will be displayed down the left side of the matrix view to identify a row of data.



5. Click **Apply** to save your selections and keep the Features dialog box open.

6. On the **From Element Types** tab, select the element type(s) from which the connection should be identified.

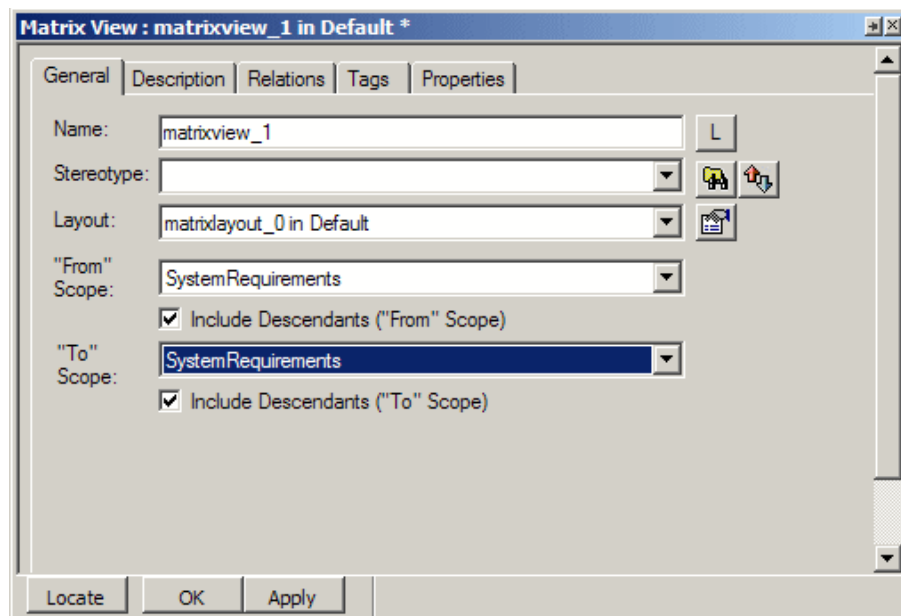


7. Click **Apply** to save your selections.
8. On the **To Element Types** tab, select the element type(s) to which the connection should be identified.
9. Click **OK** to save the layout and close the Features dialog box.

Creating a Matrix View

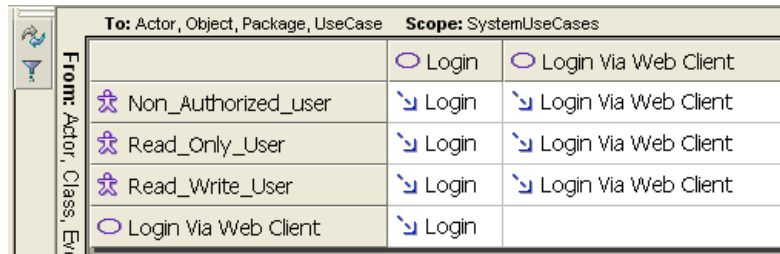
To create a matrix view to analyze the attributes of the model elements you identified in the matrix layout, follow these steps:

1. Right-click the package on the Rhapsody browser to which you want to create and store a matrix view and select **Add New > Matrix View**.
2. In the browser, right-click the new matrix view and select **Features** to open its Features dialog box.
3. Enter a **Name** for the new matrix view.
4. Select the name of a previously created matrix layout from the **Layout** drop-down list.
5. Select the scope for this view by selecting a package or project in the **“From” Scope** and in the **“To” Scope** drop-down lists, as shown in the following figure:




6. Clear or select the **Include Descendants** check box, as shown in the figure above, depending on if you want to exclude or include the descendants for the selected scope. For more information, see [Including/Excluding Descendants](#).
7. Click **OK** to save your matrix view.

8. In the browser, double-click the matrix view name to generate the results of the data query. The query analyzes data for the selected package and all of its nested packages, as shown in the following figure:



	To: Actor, Object, Package, UseCase	Scope: SystemUseCases
		○ Login ○ Login Via Web Client
☆	Non_Authorized_User	↳ Login ↳ Login Via Web Client
☆	Read_Only_User	↳ Login ↳ Login Via Web Client
☆	Read_Write_User	↳ Login ↳ Login Via Web Client
○	Login Via Web Client	↳ Login

9. To make changes to the displayed data, double-click a cell in the matrix view and make the changes in the Features dialog box for that element.
10. Depending on if you are done making changes:
 - ◆ Click **Apply** to save your changes in the Features dialog box and then click the **Refresh**  button to display the new data in the view.
 - ◆ If you are done making changes, click **OK** to close the Features dialog box.

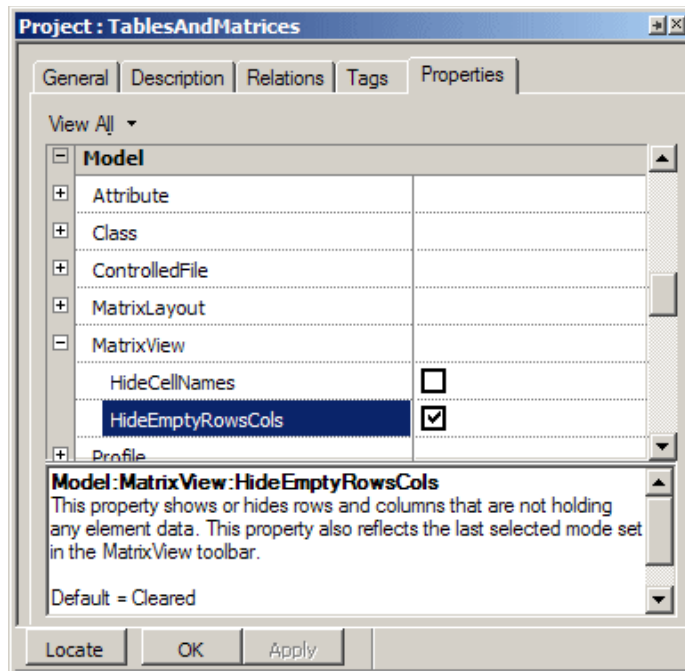
Note

You can also return to the matrix layout and make changes to the design. To redisplay the data in the view, use **Refresh**.

Filtering Out Rows and Columns without Data

If you want to remove the rows in a matrix view that do not contain data, follow these steps:

1. Open the Features dialog box for the view or choose **File > Project Properties** to display the properties for your project.
2. On the **Properties** tab, locate the `Model::MatrixView::HideEmptyRowsCols` property and select its check box, as shown in the following figure:



3. Click **OK** to save the change.
4. When you display a view, click the **Toggle empty rows filter**  button to show only the rows containing data.

If you select the `Model::MatrixView:HideCellNames` property, the content of a cell is displayed only with an icon.

Including Ports and Multiple Relations

By default, the table view and matrix view show relations between objects—even if ports are involved—and display multiple relations between elements if they exist. This means that you can easily communicate your design information in a tabular or matrix format.

The `Model::TableLayout::ShowContainerElementForPorts` property controls this capability. By default, this property is set to `Checked`. This means that for table views, if ports are encountered when searching for connections of relations, Rhapsody displays the port's parent element instead of the port itself. For matrix views, Rhapsody automatically looks at child ports of elements and finds the relations to other elements.

If you do not want this property enabled, you would clear the check box for it.

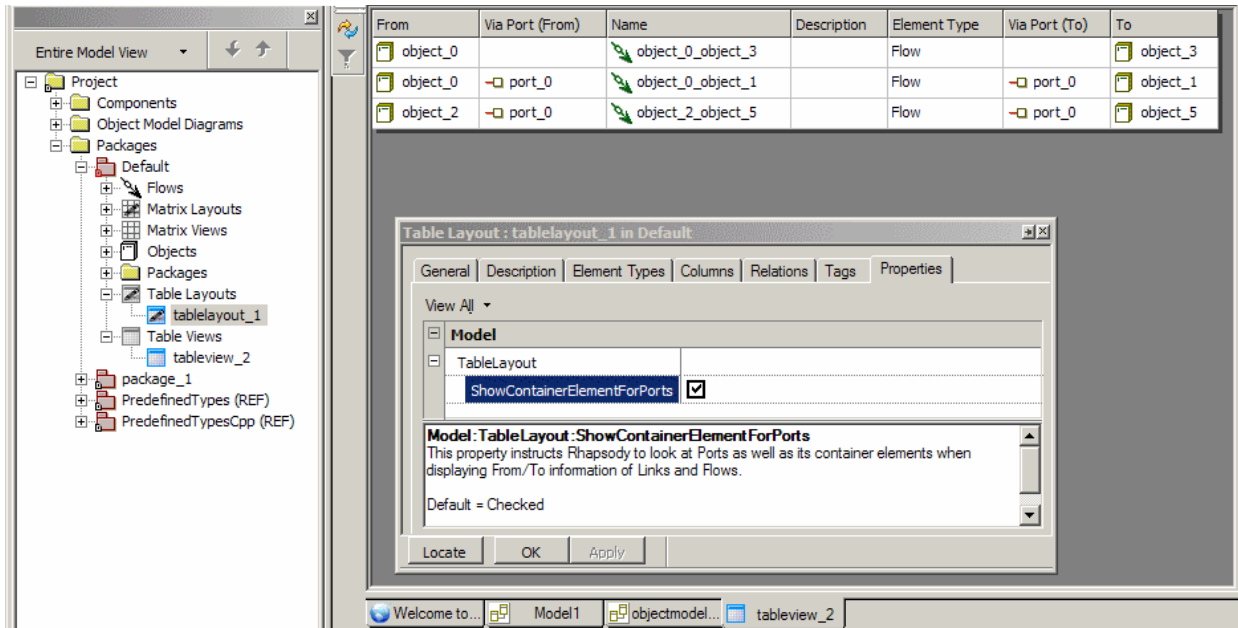
To make changes to this property, follow these steps:

1. Open the Features dialog box for the table view or matrix view.
2. On the **Properties** tab, locate the `Model:TableLayout:ShowContainerElementForPorts` property.
3. Clear or select the check box next to the property name.
 - ◆ Clear it to disable this property.
 - ◆ Select it if you want to show relations between objects—even if ports are involved—and display multiple relations between elements if they exist. This is the default.
4. Click **OK**.

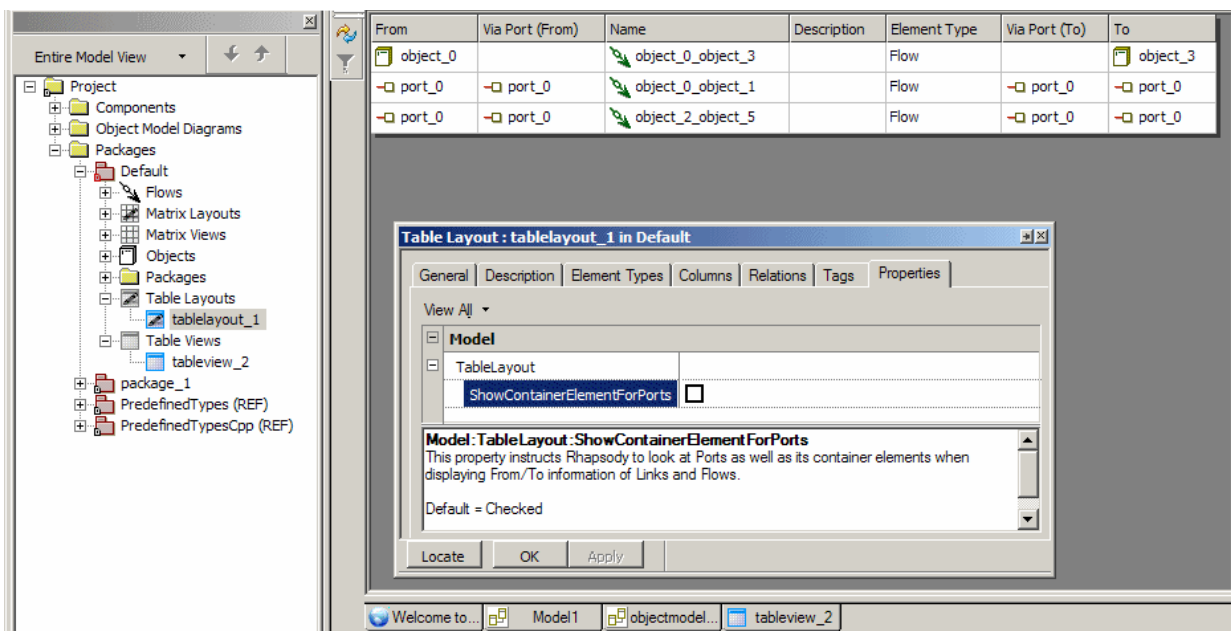
Note

When there are multiple relations, options such as **Features**, **Locate**, and so forth are disabled.

The following figure shows an example where a table view shows port relations. Notice that the table shows all the created flows with the object, instead of the ports, in the **From/To** columns. It also shows the ports in the **Via Port (From)** and **Via Port (To)** columns.



The following figure shows what the table looks like when relations are not shown. Notice that the **From/To** columns do not show the parent for the ports.

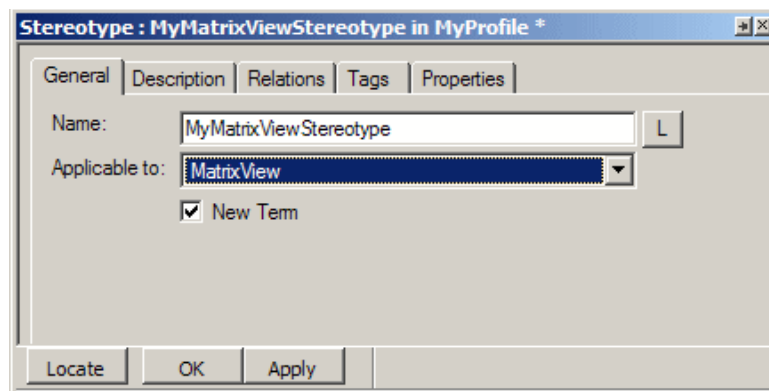


Setting up an Initial Layout for Table and Matrix Views

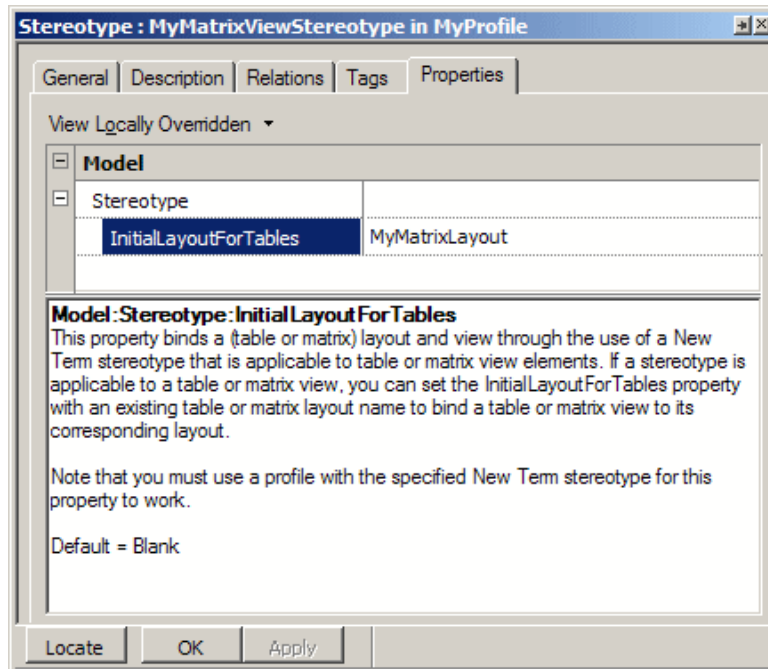
You can bind a view and layout for a table or matrix through the use of a New Term stereotype that has the `Model::Stereotype::InitialLayoutForTables` property set for this purpose for a Rhapsody profile. Once you apply the stereotype to a table or matrix layout, it lets you set the initial layout for a table or matrix view.

To apply a stereotype to a table or matrix layout for a profile, follow these steps:

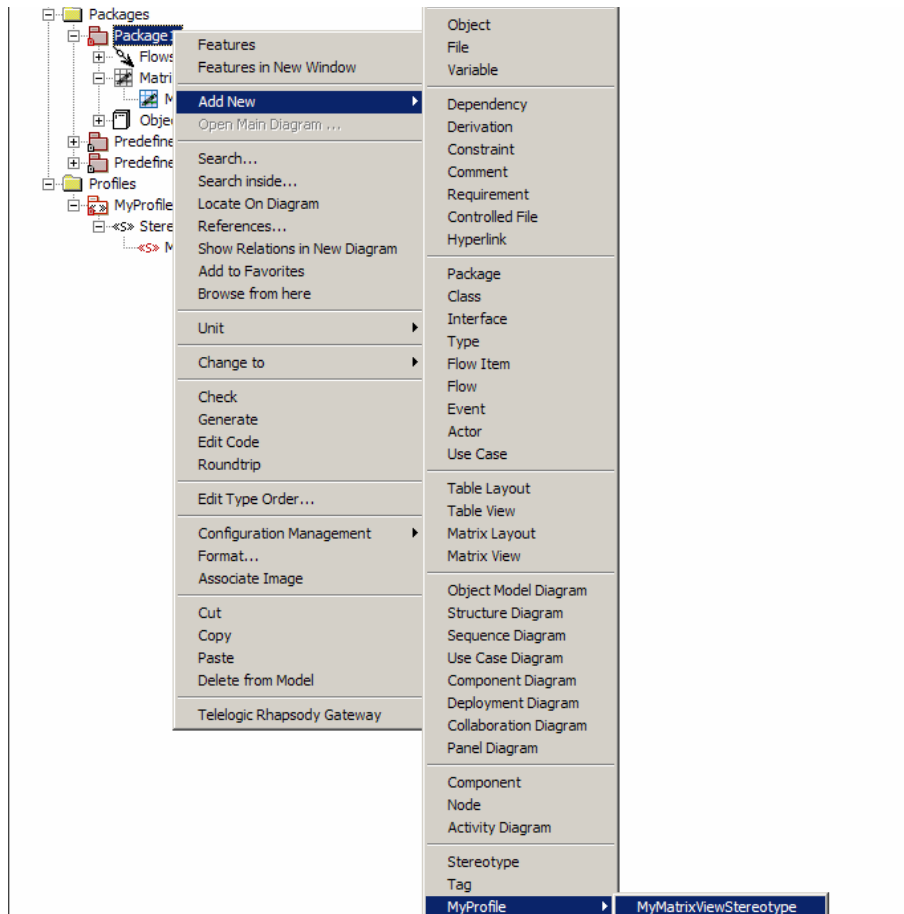
1. Create a profile. For more information, see [Creating Your Own Profile](#).
2. Create a table or matrix layout and design it as you want. See [Creating a Table Layout](#) and [Creating a Matrix Layout](#).
3. In your profile, create a stereotype and define it as a New Term and to what it is applicable to (table view or matrix view, as shown in the following figure) and click **Apply** to save but not close the dialog box. For information on stereotypes, see [Defining Stereotypes](#).



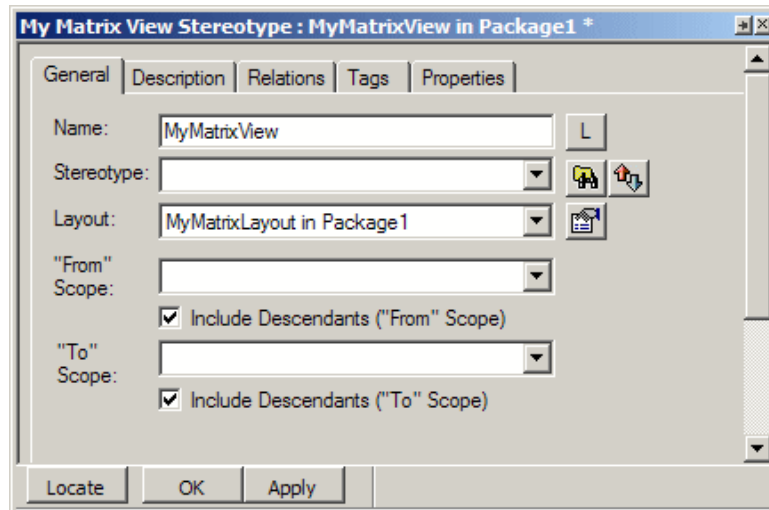
4. On the Properties tab, locate the `Model::Stereotype::InitialLayoutForTables` property and enter the name of the layout you created in step 2, as shown in the following figure, and click **OK** to close the dialog box.



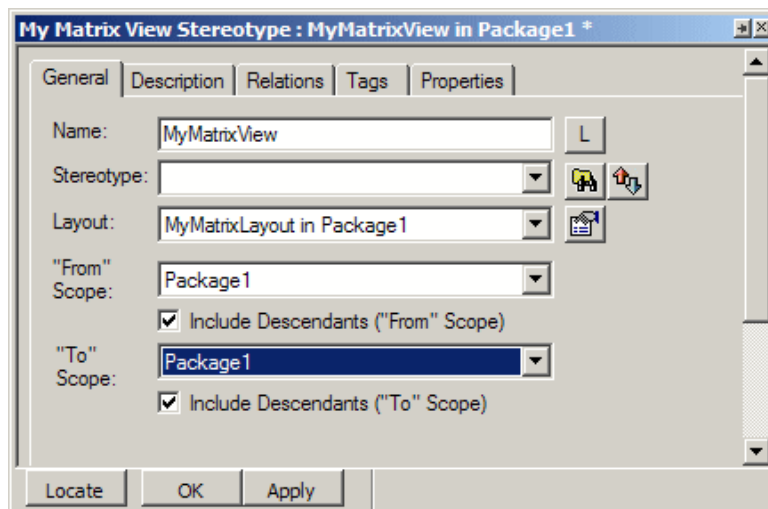
5. Create a table or matrix view through the profile, select **Add New > [name of profile] > [matrix view]**, as shown in the following figure:



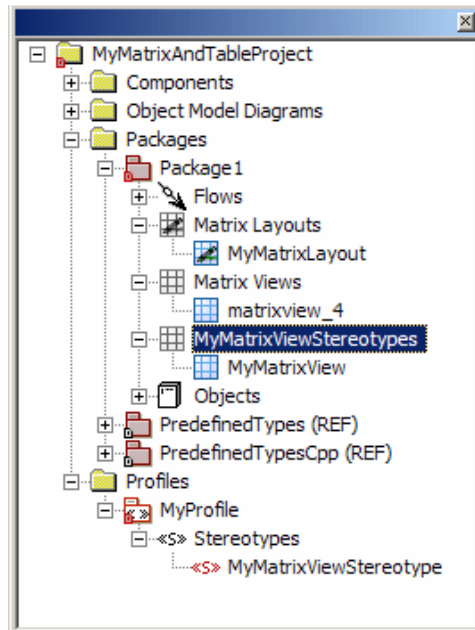
- Open the Features dialog box for the table or matrix view you created in the previous step and notice that the layout has already been identified (because of the applicable stereotype), as shown in the following figure:



- Complete your design of the view by filling in the **Scope** boxes, as shown in the following figure, and click **OK**.



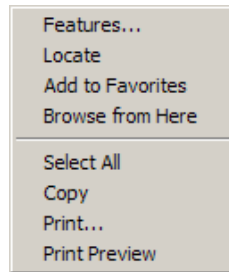
8. Look at the Rhapsody browser and notice that the view you just created is listed in a View category (in our example, **MyMatrixViewStereotypes**, as shown in the following figure) other than the general **Matrix Views** category, which only lists those views that were created without a stereotype (using **Add New > Matrix View**):



9. When you double-click the table or matrix view (**MyMatrixView** in our example above), it shows the query results from the layout that has the specified stereotype.

Managing Table or Matrix Data

After creating a table or matrix, you have additional options to manage a view and its data. Right-click the view in the drawing area to display a pop-up menu, as shown in the following figure:



This pop-up menu has the following standard Rhapsody options: Use along with **Via Port (From)**, select this property to

- ◆ **Features** to open the Features dialog box. For more information, see [Using the Features Dialog Box](#).
- ◆ **Locate** to find the location of the element on the Rhapsody browser.
- ◆ **Add to Favorites** to add to your favorites list in Rhapsody. For more information, see [Using the Favorites Browser](#).
- ◆ **Browse from Here** to open a Rhapsody browser that contains a more-focused Rhapsody browser from whichever point you are at. For more information, see [Opening a Browse From Here Browser](#).

The remaining options are standard Windows options. You may find the **Copy** option particularly useful to export the view's data into Microsoft Excel and then save it as a `.csv` file.

Working with Multiple Projects

Rhapsody allows you to have more than one project open at a time. When you have more than one project open, you can use the Rhapsody browser to copy and move elements from one project to another.

The following terms are used in Rhapsody in the context of working with multiple projects:

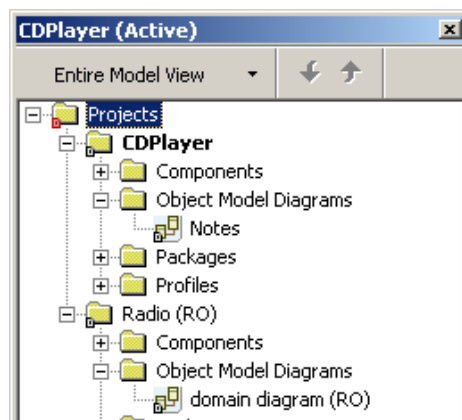
- ◆ *Projects* is the top-level container (or folder) for all open projects in a Rhapsody session (saved as an .rpl file). The **Projects** folder contains the list of projects, which can be saved and reopened.
- ◆ *Active Project* is the project that you can currently be modify. This is also the project to which Rhapsody commands, such as for code generation, will be applied (unless the command opens a dialog box that allows you to specify which project to use).

Opening Multiple Projects

To open multiple projects, follow these steps:

1. Open a project by selecting **File > Open**.
2. For each additional project you want to open, select **File > Insert Project**.

After the second project opens, the Rhapsody browser displays a project list node called **Projects** that is one level above the open projects, as shown in the following figure:



Note

Rhapsody does not permit opening two projects with the same name.

Setting the Active Project

When you select **File > Insert Project**, the project you select automatically becomes the active project.

Once you have more than one project open, you can make any project the active project as follows:

1. Right-click the project name in the Rhapsody browser.
2. Select **Set as Active Project** from the pop-up menu.

Similar to the display of the active component and active configuration, the name of the active project displays in bold in the browser.

You can modify an active project's model elements only. (Rhapsody displays **RO**, for read-only, next to all project names in the browser that are in non-active projects.)

If you make a project active without first saving changes made to the previously active project, the system asks you if you want to save those changes. If you click the **No** button, your changes to the previously active project are not yet lost. This is because when you close all the projects, the system asks if you want to save the project list and all its projects. If you click the **Yes** button, all changes made to all projects that have not been saved are saved at this time.

In general, Rhapsody commands are applied only to the active project. However, the commands **Search** and **Locate in Browser** can be applied across all open projects.

Copying and Referencing Elements among Projects

When you have more than one project open, you can use the Rhapsody browser to copy elements from one project to another using either of these methods:

- ◆ **Referencing** an element in another project only “links” that element to the original element in the original project
- ◆ **Copying** creates a element in another project that is the same as the original

Note

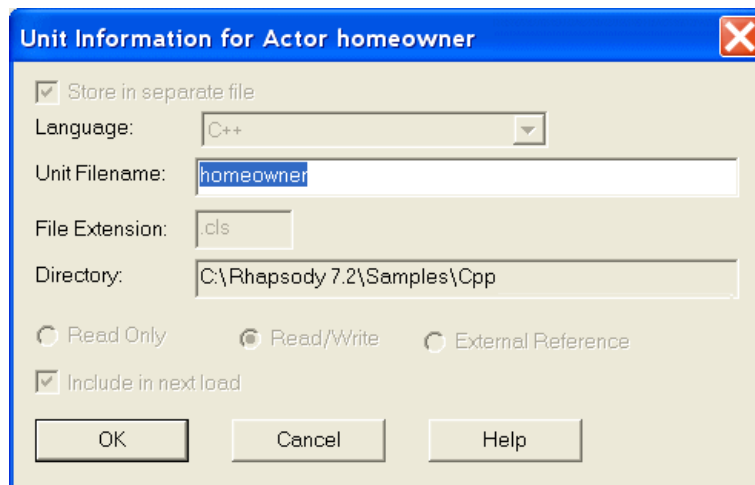
Copying and referencing can only be done within the browser. You cannot drag an element from one project to a diagram in another project.

You can use either the standard Windows copying techniques, as described in [Copying Elements to Other Projects](#) or the Shift key method, as described in [Using the Shift Key to Copy, Reference, or Move Elements](#).

Creating References

Only elements that have been saved as units can be referenced in other projects. To create a saved unit from an element, follow these steps:

1. In the active project, highlight the element in the browser and right-click.
2. Select the **Create Unit** option.
3. Type the **Unit Filename** if you want a different name from the displayed name.

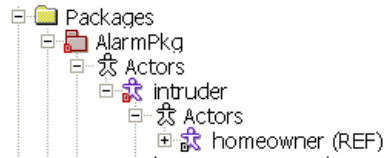


4. Click **OK** to create the unit and the icon of the new unit is marked with a red box, as shown in the following figure. In this example, the `homeowner` element is now a unit.



5. In the project that is going to receive the reference, right-click the project name.
6. Select **Set as Active Project** from the pop-up menu.
7. In the browser of the now active project, highlight the element that needs to reference the new unit and right-click.
8. Select the **Create Unit** option to change the receiving element into a unit.

9. In the original project, highlight the unit that is being referenced. Click and drag that unit to the active project's new unit. When the reference is established, the (REF) symbol appears next to the referenced element, as shown in the following figure:



Copying Elements to Other Projects

To copy an element to another project, follow these steps:

1. In the browser, right-click the project name in the Rhapsody browser from which you are copying an element.
2. Select **Set as Active Project** from the pop-up menu.
3. Highlight the element you want to copy into the other project and select **Edit > Copy** or right-click and select **Copy**.
4. Right-click the project name to which you are copying the element.
5. Select **Set as Active Project** from the pop-up menu.
6. Highlight the folder where you want to store the copied element and select **Edit > Paste**. If the copied element has the same name as an existing element in the new project, the name is appended with “_copy,” as shown in the following figure.



Copied elements with the same name as existing elements should be renamed to avoid confusion.

Using the Shift Key to Copy, Reference, or Move Elements

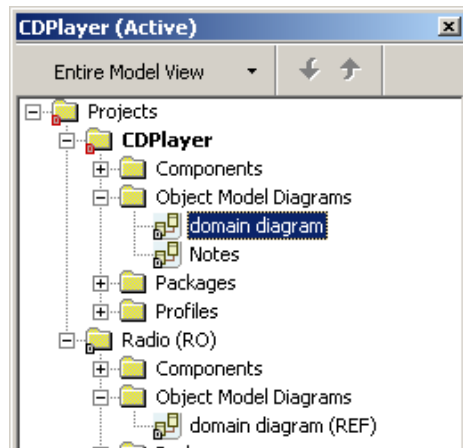
To use the pop-up menu to copy, reference, or move a unit, follow these steps:

1. Be certain that the element has been saved as a unit, as described in [Creating References](#).
2. In the non-active project, press and hold the **Shift** key, and click-and-drag the unit you want to copy, reference, or move to the target project.
3. On the pop-up menu that appears, select the applicable command:
 - ◆ **Copy here**
 - ◆ **Reference here, or**
 - ◆ **Move here - leave a reference**

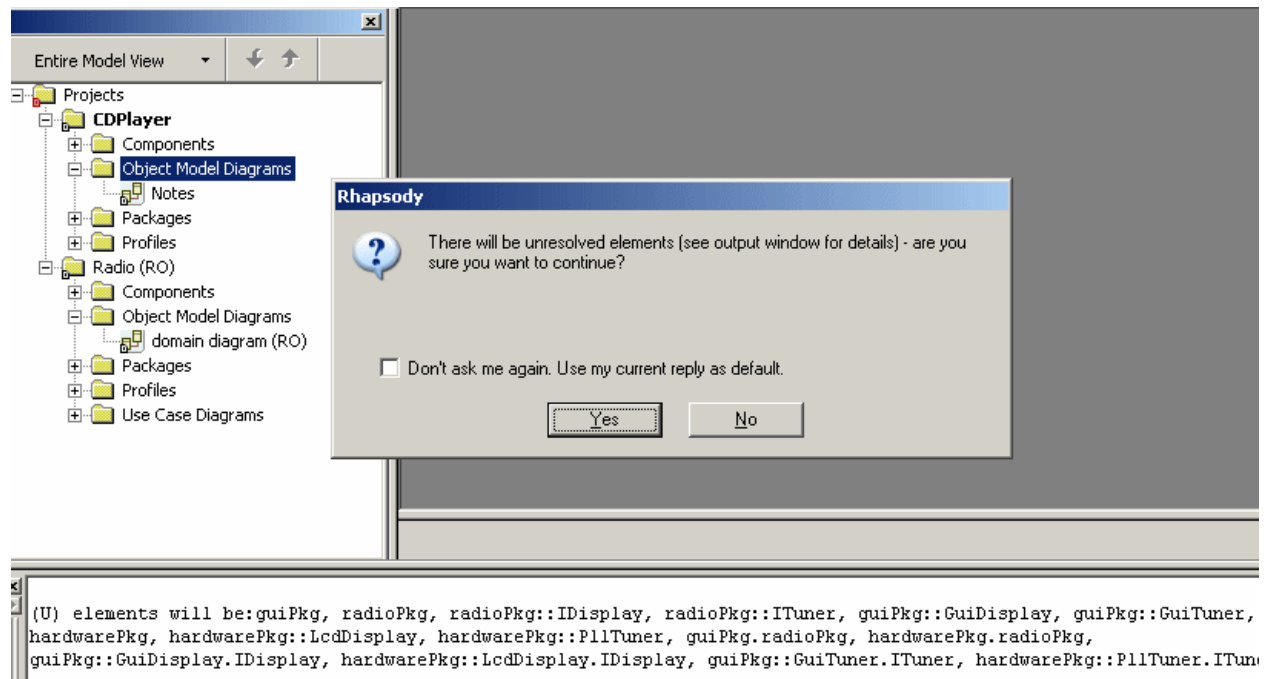
Moving Elements among Projects

To move an element from one project to another, follow these steps:

1. If the element to be moved is not a saved unit, save it as a unit.
2. Press and hold the **Alt** key, then click-and-drag the element to the other project, as shown in the following figure:



3. Rhapsody checks if there will be unresolved elements in the target project as a result of the move. If Yes, a message box appears along with the Output window, as shown here:



4. If you click **Yes**, the unit moves to the target project.

Note

The original unit, that was moved, now has a (REF) tag in the source project because the unit has been moved and the moved unit is now the unit of record in the active project.

Closing All Open Projects

To close all of the projects that are currently open, select **File > Close**.

Managing Project Lists

When a number of projects are open at the same time, you can save the list of projects as a Rhapsody project list (.rpl) file.

Saving Projects in a Project List File

To save the project list, select **File > Save All**.

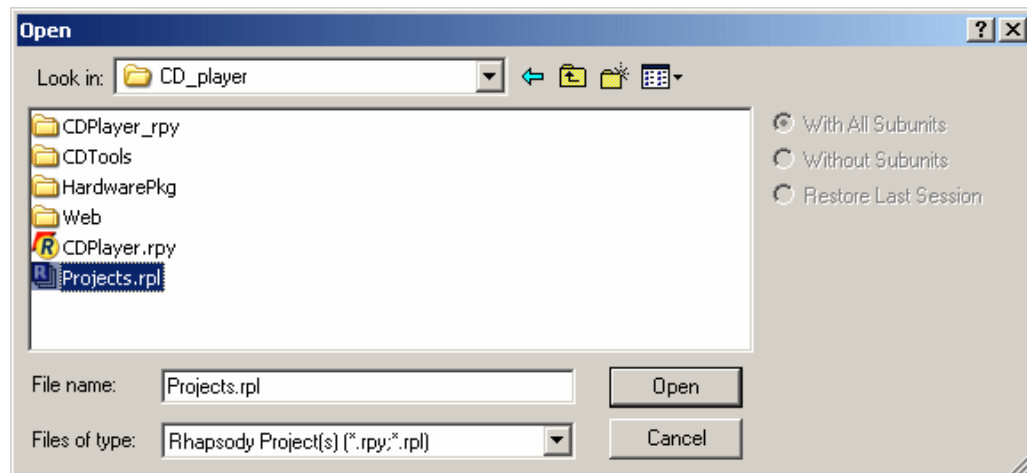
A new .rpl file is created in the current active project folder. The name of the project list file will be `Projects.rpl`. (If a file with this name already exists in the folder, a number will be added to the end of the file name, for example `Project1.rpl`.)

The current active project is saved as an attribute of the project list, so when you reopen a project list, the active project will be the project that was active when the project list was last saved.

Opening a Project File List

After a project file list has been saved, you can re-open all the projects in a project list as follows:

1. Select **File > Open** from the menu.
2. Select the relevant project list file (the Open dialog box displays all .rpy and .rpl files), as shown in the following figure:



3. Click **Open**.

Adding a Project to a Project List File

If you open a project list file, its contents are updated each time you select **Save All**. Therefore, to add another project to a project list, follow these steps:

1. Select **File > Insert Project** and choose the project to add.
2. Decide if the just added project or another one should be the current active project.
3. Select **File > Save All**.

Removing a Project from a Project List File

To remove a project from a list of files, follow these steps:

1. In the Rhapsody browser, select the project to remove.
Note that you cannot remove the current active project.
2. Press the **Delete** key on your keyboard.
3. Click **Yes** to confirm your action.
4. Select **File > Save All**.

Project Limitations

The following items identify multiple project work and display limitations.

New Project

You cannot add a new project to a list of open projects. When you select **File > New**, if there are changes to be saved, you will be asked if you want to save before closing, then all open projects will be closed, and the project list will be saved before the New Project dialog box appears.

Placement of GUI Elements

Information regarding the placement of elements such as toolbars and the browser window are stored in the `rhapsody.ini` file, and are, therefore, uniform for all projects and project lists.

Information regarding the placement of elements such as windows are stored in a project's workspace file (`.rpw`). Therefore, these elements will change, depending on which project in the list is currently the active project.

References Dialog Box

The References dialog box includes the references for all of the projects in the project list, and not just those for the active project.

DiffMerge

DiffMerge does not support the comparison of project lists (that is, groups of projects).

Configuration Management

Rhapsody does not support the configuration management of project lists but you can use your configuration management tool directly for a project file list.

Properties

When viewing properties, Rhapsody always displays the property values of the selected project. However, for all properties that affect more than one project, Rhapsody uses the settings of the active project.

Components and Configurations

When you select a different project to be the active project, the **Code** toolbar displays the active component and configuration for that project. The list of components/configurations available is also updated accordingly.

VBA Editor and the Active project

When you open the VBA editor, you see only the items belonging to the active project.

Naming Conventions and Guidelines

To assist all members of your team in understanding the purpose of individual items in the model, it is a good idea to define naming conventions. These conventions help team members to read the diagram quickly and remember the model element names easily.

Note

Remember that usually the names used in the Rhapsody models are going to be automatically written into the generated code. Therefore, the names should be simple and clearly label all of the elements, and they should not use any special characters.

Guidelines for Naming Model Elements

The names of the model elements should follow these guidelines:

- ◆ Class names begin with an upper case letter, such as `System`.
- ◆ Operations and attributes begin with lower case letters, such as `restartSystem`.
- ◆ Upper case letters separate concatenated words, such as `checkStatus`.
- ◆ The same name should not be used for different elements in the model because it will cause code generation problems. For example, you should not have a class, an interface, and a package with the same name of `Dishwasher`.
- ◆ Note the following about special characters:
 - Do not include special characters in an element's *name* if the element is used for code generation.

Note: You can use special characters in the *labels* for model elements. See [Labeling Elements](#).

- The following elements are the only ones for which you can include special characters: Dependencies, Stereotypes, Flows, Links, Configurations, Table layouts, Table views, Matrix layouts, Matrix views, Requirements, Actors, Use Cases, and all diagrams.

Note: You can use the `General::Model::NamesRegExp` property to control what special characters are allowed. For detailed information on a property, see the definition displayed in the **Properties** tab of the Features dialog box or examine the complete list of property definitions in the *Rhapsody Property Definitions* PDF file available from **Help > List of Books**.

- Note that while you can use spaces (but not special characters) in the names for actors, it is not recommended.

Standard Prefixes

Lower and upper case prefixes are useful for model elements. The following is a list of common prefixes with examples of each:

- ◆ Event names = “ev” (evStart)
- ◆ Trigger operations = “op” (opPress)
- ◆ Condition operations = “is” (isPressed)
- ◆ Interface classes = “I” (IHardware)

Using Project Units

In Rhapsody, a *unit* is any element of a project that is saved in a separate file. You can partition your model into units down to the class level. Creating units simplifies collaboration in team environments. With this feature, you have explicit control over file names and modification rights, and you can check unit files in and out of a configuration management system.

Note

Association ends and ports cannot be saved as units.

The project and all packages are always units. The following table lists other project elements that can be units.

Element	File Extension	Unit by Default?
Actors	.cls	No
Components	.cmp	Yes
Packages	.sbs	Yes
Classes	.cls	No
Implicit objects (parts)	.cls	No
Files	.cls	No
Diagrams (except statecharts and activity diagrams)	Block definition diagrams (* .omd)	No
	Component diagrams (* .ctd)	No
	Collaboration diagrams (* .clb)	No
	Deployment diagrams (* .dpd)	No
	Internal block diagrams	No
	Object model diagrams (* .omd)	No
	Sequence diagrams (* .msc)	No
	Structure diagrams (* .std)	No
	Use case diagrams (* .ucd)	No

Unit Characteristics and Guidelines

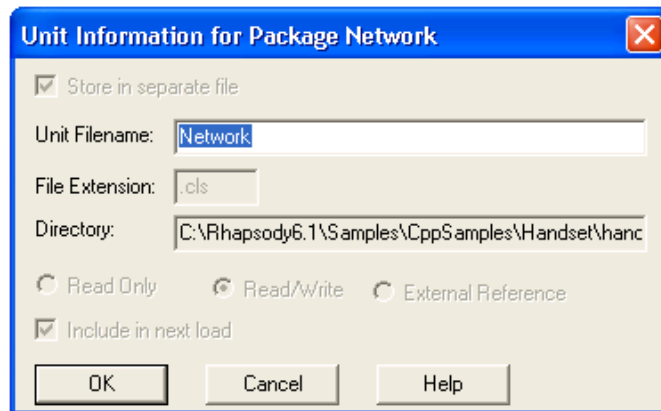
The following unit characteristics and guidelines may help you use units effectively in your projects:

- ◆ A unit can be part of only one Rhapsody model. Therefore, a unit can be modified in only one Rhapsody model.
- ◆ A unit can be referenced as often as necessary.
- ◆ A unit may contain many subunits.
- ◆ Each model element is identified by a Global Unique Identifier (GUID), so each unit is unique in its model.
- ◆ Only model elements that must be shared should be changed into units.
- ◆ A unit's name should be the same as its model element name. This simplifies the association between a unit and its model element.
- ◆ Changing many model elements to units may slow Rhapsody processes.
- ◆ To create diagrams automatically as units, change the `General::Model::DiagramIsSavedUnit` property to be `Checked`. The default is `Unchecked`.

Separating a Project into Units

If you need to share units in one project with another or you need to control some model elements in a configuration management system, you may need to divide the project into units. Keep in mind the [Unit Characteristics and Guidelines](#) and follow these steps to change elements to units:

1. Right-click the element, then select **Create Unit** from the pop-up menu. The Unit dialog box for the element opens with fields filled in, as shown in the following figure.



2. By default, the **Store in separate file** check box is selected.
3. Edit the default **Unit Filename**, if desired. Rhapsody assigns the appropriate file extension for you in the next field.
4. You change the **Directory** for the new unit.
5. In the access privileges radio button area, you may want to change the default **Read/Write** selection for the new unit.
6. The **Include in next load** selection is automatically checked since you are adding a unit.
7. Click **OK** to create the unit and close the dialog box.

The unit in the browser is now marked with a small red icon. Save the unit and the icon turns black.

Refer to the *Rhapsody Team Collaboration Guide* for more information on units.

Modifying Units

To modify a unit, follow these steps:

1. In the browser, right-click the unit to modify, then select **Unit > Edit Unit** from the pop-up menu. The Unit Information for Package dialog box opens.
2. Change the settings as desired.
3. Click **OK** to apply your changes and close the dialog box.

To reduce the amount of time required to save a project, Rhapsody marks all modified units and enables you to save those units without saving the entire project. To indicate that a unit has changed, the unit icon in the browser changes from black to red.

Saving Individual Units

To save a selected unit, follow these steps:

1. Select the unit in the browser.
2. Right-click the unit and select **Unit > Save Unit**.

Loading/Unloading Units

When you open a Rhapsody project, the Open dialog box allows you to specify whether or not Rhapsody should load the subunits contained in the project. If you need to locate the unloaded units in a project, you can use either of these tools:

- ◆ [Filtering the Browser](#) for the Unloaded Units
- ◆ [Search and Replace Facility](#)

If you elected to load the project without its subunits, you can later load individual subunits with these steps:

1. Right-click the subunit in the browser to display the menu.
2. From the context menu, select **Load [unit name]** or select **Load [unit name] with Subunits** to load the unit together with all its subunits.

To unload a unit, follow these steps:

1. Right-click the subunit in the browser to display the context menu.
2. From the context menu, select **Unit > Unload [unit name]**.

Units that are not currently loaded are indicated by *(U)* before the unit name in the browser.

If you do want to unload a unit, but you want to prevent it from being loaded the next time you open the project:

1. In the browser, right-click the unit to open the context menu, and select **Unit > Edit Unit**.
2. When the Unit Information dialog box is displayed, clear the *Include in next load* check box.

Loading Units from Last Session

If you would like to load only those units that were open during your last Rhapsody session, select the *Restore Last Session* radio button when you open the project with the Open dialog box.

If you open the project by selecting its name from the MRU list (under *File*), Rhapsody will also only load those units that were open when you completed your last session.

Note

This applies also to the *Include in next load* check box. If you cleared this check box, Rhapsody will refrain from loading the unit only if you select the *Restore Last Session* radio button when you open the project, or open the project from the MRU list.

Saving Packages in Separate Directories

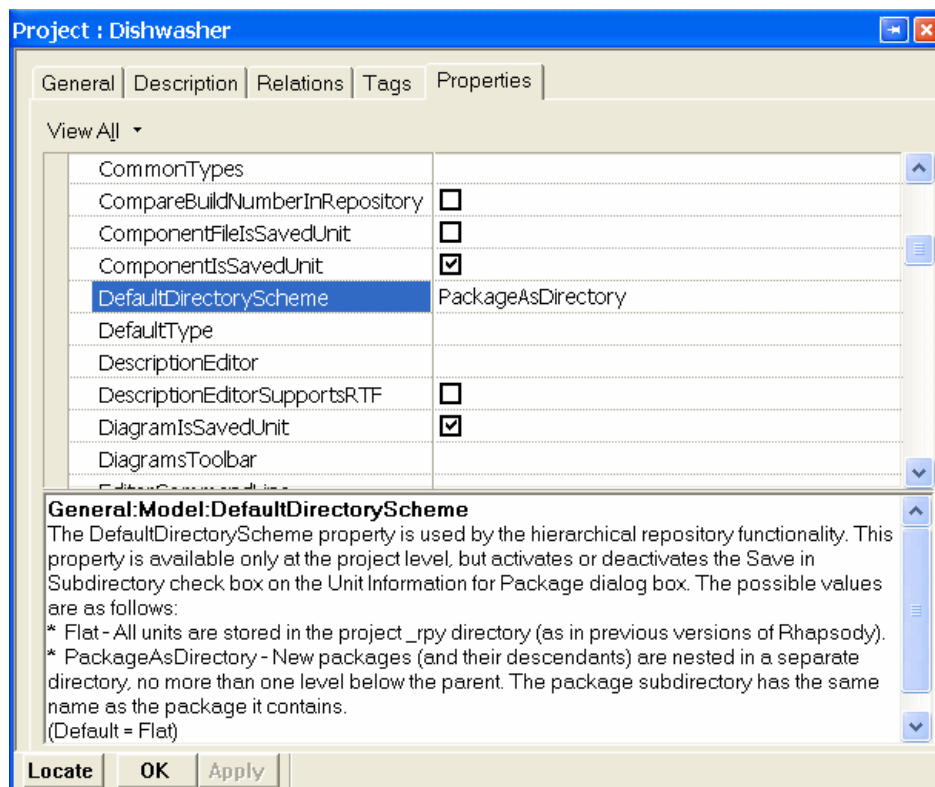
To assist with configuration management and improve project organization, you may want to store packages in separate subdirectories within a parent folder. Rhapsody has two directory schemes:

- ◆ In *flat* mode, all package files are stored in the project directory, regardless of their location in the project hierarchy.
- ◆ In *hierarchical* mode, a package is stored in a subdirectory one level below its parent. It is possible to have a hybrid project, where some packages are stored in flat mode, and others are organized in a hierarchy of folders.

To set the default so that new packages are stored in separate directories, follow these steps:

1. Select the <ProjectName> at the top of the browser hierarchy.
2. Right-click and select Features.
3. Locate the General::Model properties and select the DefaultDirectoryScheme property.

Change the value from flat to PackageAsDirectory, as shown in the following figure:



Flat Mode

In flat mode, Rhapsody stores all package files in one directory. This is usually the project directory.

If you are changing modes from hierarchical to flat, Rhapsody maintains the existing directory structure, but does not add any new subdirectories. New packages are stored within the existing structure beneath the directory of their closest parent.

To create a new model that will be in one file, after you create the project, set the following properties' check boxes to be Cleared at the project level:

- ◆ General::Model::BlockIsSavedUnit
- ◆ General::Model::ClassIsSavedUnit
- ◆ General::Model::ComponentIsSavedUnit
- ◆ General::Model::DiagramIsSavedUnit
- ◆ General::Model::FileIsSavedUnit
- ◆ General::Model::ObjectIsSavedUnit
- ◆ General::Model::PackageIsSavedUnit

Hierarchical Mode

In hierarchical mode, you can save a package in a unique subdirectory one level below the directory of its parent. All units contained in the package are saved in its subdirectory, along with the package (.sbs) file. Nested packages are further divided into subdirectories.

Consider the example of a project `Home` that contains the package `Family`, which contains the package `Pets`. With each package in its own directory, the path of the `Pets.sbs` file would be:

```
../Home/Family/Pets/Pets.sbs
```

Note

When changing from flat mode to hierarchical mode, Rhapsody does not automatically create folders for existing packages. Instead, it creates a folder for each *new* package within the existing directory structure.

1. Right-click the package and select **Unit > Edit Unit**. The Unit Information for Package dialog box opens.
2. Select the **Store in separate Directory** check box (available only for packages). The name of the separate directory has the same name as the unit.
3. Click **OK**.

Rhapsody creates the new directory and moves the package, along with all of its subunits, into the new folder.

Changing a Hierarchical Model to a Flat Model

To change an existing model from hierarchical mode to flat mode, write a VBA script that iterates over the entire model and, for each `IRPUnit`, calls the `setSeparateSaveUnit(true)` method. The only unit that should not activate this method is the project. Refer to the *Rhapsody API Reference Manual* for more information on using VBA with Rhapsody.

Using Environment Variables with Reference Units

If you have a reference unit in your model (added using the option **Add to Model As Reference**), you can edit its location using the **Directory** field of the Unit Information dialog box, and use an environment variable as part of that location.

Note

When you add a reference to your model, Rhapsody adds the packages as top-level packages by default. However, you can move the reference packages so they become nested packages.

For example, you can use a relative path (`.. \`) or the environment variable `$ENV_VAR`. If you set the `General::Model::EnvironmentVariables` property to include the path of this environment variable, Rhapsody parses and executes that environment variable when it opens the project, and then searches for the reference unit in the specified location.

Note

If you use relative paths, note that the path is relative to the `_rpy` folder—*not* where the `.rpy` file is located.

Using Workspaces

Workspaces enable you to work with selected units of a project without having to open the entire model. This feature supports component-based development and collaboration among teams. It also reduces the time required for routine operations, such as saving and code generation, by enabling you to load only the units currently under development.

In addition, workspaces save viewing preferences, including window size, position, status of feature windows, and the scaling or zoom factor of open diagrams.

Rhapsody automatically saves workspace information in a separate file named `<Project>.rpw`. Rhapsody saves the `.rpw` file whenever a project is closed, regardless of whether you save the project itself.

Creating a Custom Rhapsody Workspace

1. Select **File > Open**, or click the **Open** button on the standard toolbar. The Open dialog box is displayed.
2. In the **Look in** field, browse to the location of the project, then select the `.rpy` file.

Alternatively, type the name of the project file in the **File name** field.
3. Select the **Without Subunits** check box. This prevents Rhapsody from loading any project units. All project units will be loaded as stubs.
4. Click **Open**. The project file opens with no units loaded. This empty project acts as a starting point for you to create your workspace.
5. Add units to your workspace to customize it for your needs.

Adding Units to a Workspace

1. Select the unit in the browser.
2. Right-click the unit, then select **Load Unit** from the pop-up menu.
3. Select the unit in the browser.
4. Right-click the unit, then select **Load Unit with Subunits**.

Unloaded Units

Units of your project that have not been loaded into your workspace are marked with the letter “U.” This designation means that the unit is a *stub* unit, and was either excluded from the project intentionally, or was not found when Rhapsody attempted to load the unit.

In diagrams, a “U” located at the destination end of a relation, dependency, or generalization means that the target is an unresolved element. In this case, the originator of the relation, dependency, or generalization is loaded, but the unresolved element has been either intentionally or accidentally excluded from the project.

You can use the [Advanced Search and Replace Features](#) to locate any unloaded units and load them.

Opening a Project with Workspace Information

To open a workspace, follow these steps:

1. Select **File > Open**, or click the **Open** button on the standard toolbar. The Open dialog box is displayed.
2. In the **Look in** field, browse to the location of the project, then select the `.rpy` file. Alternatively, type the name of the project file in the **File name** field.
3. Select the **Restore Last Session** check box. The project opens with the workspace information that was saved during your last Rhapsody session.

To open a project without loading workspace information, see [Opening a Rhapsody Project](#).

Controlling Workspace Window Preferences

Workspaces save information about your window preferences. These preferences include window size, position, status of feature windows, and the scaling or zoom factor of open diagrams.

To prevent Rhapsody from saving graphic editor settings, set the property `OpenDiagramWithLastPlacement` under `General::Workspace` `BlockIsSavedUnit` check box to `Cleared`. With this setting, Rhapsody does not save the position of graphic editor windows. Instead, it uses the default window settings the next time you open a graphic editor.

You can prevent Rhapsody from saving any window preferences by setting the property `OpenWindowsWhenLoadingProject` under `General::Workspace` check box to `Cleared`.

Project Files and Directories

The project directory is the top-level directory for a project. It is the folder entered in the New Project dialog box when you create a new project. Rhapsody creates a number of project files with matching directories. Each project file/directory pair shares the same name.

The following table lists project files created by Rhapsody.

File Name	Description
<Project>.rpy	The project file or model repository. Requires the repository files in the <Project>_rpy directory to be a complete model.
<Project>_rpy	Directory containing unit files for the project, including: <ul style="list-style-type: none"> • Components (*.cmp) • Packages (*.sbs) • Classes (*.cls) • Use case diagrams (*.ucd) • Sequence diagrams (*.msc) • Object model diagrams (*.omd) • Component diagrams (*.ctd) • Collaboration diagrams (*.clb) • Deployment diagrams (*.dpd) • Structure diagrams (*.std) • Table of files contained in the project (filesTable.dat)
<Project>_auto.rpy	A backup file of the model, created during autosave. This file is written only if the <Project>.rpy has been modified since it was last saved.
<Project>_auto_rpy	Directory containing a backup of project files modified since the last save. All autosave files are stored in flat mode. See Saving Packages in Separate Directories for more information.
<Project>_bak1.rpy	A backup of the model, created when the project is first saved. Requires the repository files in the <Project>_bak1_rpy directory to be a complete model.
<Project>_bak1_rpy	Directory containing a backup of the project files for the <Project>_bak1.rpy repository. This folder can contain the same types of files as the <Project>_rpy folder.
<Project>_bak2.rpy	A backup of the model, created when the project is saved a second time. Contains the most recent backup of the project. Requires the repository files in the <Project>_bak2_rpy directory to be a complete model.
<Project>_bak2_rpy	Directory containing a backup of the project files for the <Project>_bak2.rpy repository. This folder can contain the same types of files as the <Project>_rpy folder.

File Name	Description
<Project>.rpw	Workspace settings file. Preserves the workspace settings for the project. See Using Workspaces for more information.
<Project>.ehl	Events history list. Stores events and breakpoints during animation.
<Project>.vba	VBA project file. Contains VBA macros, modules, user forms, and so on.
ReverseEngineering.log	A log of reverse engineering activity containing messages reported in the output window during reverse engineering.
<Project>_ATG	Directory that holds any tests created using the Rhapsody Automatic Test Generation add-on (if you installed the product).
<Project>_RTC	Directory that holds any tests created using the Rhapsody TestConductor™ add-on (if you installed the product).
load.log	A log of when various repository files were loaded into Rhapsody, including any errors that might have occurred during the loading and resolution phases.
store.log	A log recording when the project was saved.
<Component>	Directory for each component in the project. Organizes files generated for each configuration in the component. Each configuration is placed in a subdirectory that contains the source and binary files for a build.

Parallel Project Development

Many companies use Rhapsody to create large models developed by multiple users, who are often working in parallel in distributed teams. These teams may use a source control tool or configuration management (CM) software, such as ClearCase, to archive project units, but not all files may be loaded into CM during development.

Engineers in the team need to see the differences between an archived version of a unit and another version of the same unit or a similar unit that may need to be merged. To accomplish these tasks, they need to see the graphical differences between the two versions, as well as the differences in the code. However, source control software does not support graphical comparisons.

The Rhapsody DiffMerge tool supports team collaboration by showing how a design has changed between unit revisions and then merging units as needed. It performs a full comparison including graphical elements, text, and code differences.

Unit Types

A Rhapsody unit is any project or portion of a project that can be saved as a separate file. The following are some examples of Rhapsody units with the file extensions for the unit types:

- ◆ Class (.cls)
- ◆ Package (.sbs)
- ◆ Component (.cmp)
- ◆ Project (.rpy)
- ◆ Any Rhapsody diagram

DiffMerge Tool Functions

The DiffMerge tool can be operated inside and/or outside your CM software to access the units in an archive. It can be launched from inside or outside Rhapsody. It can compare two units or two units with a base (original) unit.

The units being compared only need to be stored as separate files in directories and accessible from the PC running the DiffMerge tool. In addition to the comparison and merge functions, this tool provides these capabilities:

- ◆ Graphical comparison of any type of Rhapsody diagram
- ◆ Consecutive walk-through of all of the differences in the units
- ◆ Generate a Difference Report for a selected element including graphical elements
- ◆ Print diagrams, a Difference Report, Merge Activity Log, and a Merge Report

Note

Many of the DiffMerge tool's operations can be run from a command-line interface to automate some of the tasks associated with software development (for example, to schedule nightly builds).

Project Migration and Multi-Language Projects

When you run Rhapsody, you select a specific language version of Rhapsody. This determines the language that is associated with your Rhapsody projects.

You can, however, open Rhapsody projects that were created with other language versions of Rhapsody.

In addition, Rhapsody allows a single model to contain units that are associated with different languages. A model can include units associated with C, C++, or Java. Code can then be generated in the appropriate language for each unit.

Note

While project migration is a built-in feature of Rhapsody, you can only have multi-language projects if you possess the special license required for this feature.

Opening Models from a Different Language Version

Rhapsody allows you to open models created in a different language version of Rhapsody. This is also referred to as *migration* of projects.

When you try to open a project that was created in a different language version of Rhapsody, you are notified that the project will be converted to the language of the current version, and you are asked whether you would like to continue with the conversion of the project.

Note

When you migrate a project, you do not lose any language-specific features of model elements that are not supported in the language version of Rhapsody that you are running. These language-specific characteristics will not be displayed, for example, in the Features dialog box, and any code generation will be in the language of the current version not the version with which the model was originally created. However, Rhapsody maintains this information. If, at a later stage, you reopen the model in the original language, you will once again see these language-specific characteristics.

When a project is migrated, bodies of operations and any other code entered manually in Rhapsody are not converted to the target language. If you already have such code in your model before the migration, make sure to convert the code in order to avoid compilation errors.

If you use *Add by reference* to add a unit whose language differs from that of the version of Rhapsody you are running, a non-persistent conversion is performed (since these elements are read-only). This non-persistent conversion will be performed each time you open the model.

Note

If you have a license for multi-language projects, no conversion is performed when you open a model from another language version of Rhapsody. If you would like to convert an entire project, just change the unit language at the project level. For details, see [Determining Language of a Unit in Multi-Language Projects](#).

Multi-Language Projects

Rhapsody uses units to permits projects to contain components from different development languages. Each unit is associated with a specific language.

Determining Language of a Unit in Multi-Language Projects

When you create a new unit, the Unit Information dialog box provides a drop-down list that allows you to select a specific language for the unit. The default language for a new unit is the language of its owner unit.

To change the language of an existing unit:

1. Right-click the unit in the browser, and select **Unit > Edit Unit** from the context menu.
2. When the Unit Information dialog box is displayed, select the language from the drop-down list.
3. Click **OK**.
4. When you are asked to confirm the change, click **Yes**.

If you are changing the language of a unit that contains subunits, Rhapsody will ask you if you also would like to change the language of all of the contained subunits.

Note

As is the case for project migration, if you change the language of a unit, you do not permanently lose any language-specific features of the unit. These language-specific characteristics will not be displayed, and any code generation will be in the new language. However, Rhapsody maintains this information. If, at a later stage, you switch the unit back to the original language, you will once again see these language-specific characteristics.

When you move units of one language to a package of another language, Rhapsody will inform you that they are different languages and will ask you to confirm the move.

If you try to add a unit that is associated with another language, Rhapsody will ask you to confirm the addition of the unit. You will also be prompted for confirmation if you add “by reference” a unit that is associated with another language.

Code Generation

In each of the language versions of Rhapsody, you can generate code for units in each of the three languages - C, C++, Java.

For code generation to work properly, you have to adhere to the following rules:

- ◆ To generate code for units in a certain language, the appropriate language must be specified at the component level.
- ◆ Elements included in the scope of a component must be of the same language as the component. (If other language elements are included in the scope, a warning will be issued during code generation.)

Note

If you select *All Elements* as the scope, Rhapsody will automatically include only those units whose language matches that of the component. If you choose the *Selected Elements* option, Rhapsody will only display those units whose language matches that of the component. However, if you selected specific elements, and then changed the language of the component, Rhapsody will not deselect these non-matching units. When you attempt to generate code with such a component, you will receive error messages. The same principle applies to the *Initial Instances* specified for the configuration.

Language-Specific Differences in Rhapsody

The Features dialog box differs from language to language. For each unit in your model, the appropriate Features dialog box is displayed.

Similarly, the properties displayed reflect the language of the selected unit.

Non-Unit Elements

If you try moving an element not saved as a unit to a package with a different language, the language of the element will be changed to that of the receiving package after you confirm that you want to move the element.

Reverse Engineering

The reverse engineering mechanism always uses the language of the active component. When you use the Reverse Engineering dialog box to add files to reverse engineer, the default file filter used will reflect the language of the active component, for example, *.java if the active component is associated with Java.

Miscellaneous Issues

- ◆ Rhapsody API: The interface `IRPUnit` allows recursive changing of unit language.
- ◆ ReporterPLUS can query the language of an element.
- ◆ Rhapsody's internal reporter shows the language of each saved unit.
- ◆ XMI: Language of each unit is exported and imported.
- ◆ Graphic Editor: Changes to language of a unit do not affect the depiction of the unit in the graphic editor. For example, if you change the language of a template class to C, it will still look like a template class in the graphic editor.
- ◆ DiffMerge checks for language differences.
- ◆ PredefinedTypes package: These packages are language-dependent. When you create a unit whose language differs from that of the Rhapsody version being used, the relevant package of predefined types for that language will be loaded.

Using the Rhapsody Workflow Integration with Eclipse

The Rhapsody plug-in for Eclipse has two implementations:

- ◆ **Rhapsody Workflow Integration** allows the software developer to work in Rhapsody and use some Eclipse features through Rhapsody menu options. This integration can be used for C and C++ development in either Windows or Linux environments. Both Eclipse and Rhapsody must be open when the developer is using this integration.
- ◆ **Rhapsody Platform Integration** permits developers to work on a Rhapsody project completely within Eclipse. Rhapsody does not need to be open for this implementation. This integration can be used for C, C++, or Java development in a Windows environment only. Refer to the *Eclipse Platform Integration User Guide* for detailed instructions.

The Rhapsody *Workflow integration with Eclipse* allows software developers to work on Rhapsody C or C++ projects in Eclipse version 3.3 or WindRiver's Eclipse Workbench 2.6 to perform these tasks:

- ◆ Create a new IDE (integrated development environment) configuration in Rhapsody in order to perform the following tasks:
 - work in Eclipse on a new Rhapsody project
 - attach an existing Rhapsody project to an existing Eclipse project
- ◆ Import legacy Eclipse models into Rhapsody UML models
- ◆ Make changes in Eclipse and rebuild the project automatically in both Rhapsody and Eclipse
- ◆ Use the debugging facilities in Rhapsody and Eclipse in a synchronized manner
- ◆ Use Rhapsody's reverse engineering with an active Eclipse configuration
- ◆ Disconnect an Eclipse project from the associated Rhapsody configuration

Converting a Rhapsody Configuration to Eclipse

If you created a configuration in Rhapsody, but now you want to convert it into an Eclipse configuration, follow these steps:

1. Open Rhapsody and in the Rhapsody browser highlight the configuration that you want to convert to be an Eclipse configuration.
2. Right-click and select the **Change to > Eclipse Configuration** menu options.
3. From this point, the process is the same as creating a new Eclipse configuration.
 - ◆ Outline
 - ◆ Navigator

Importing Eclipse Projects into Rhapsody

Rhapsody makes it easy for you to import your Eclipse projects quickly. When you import an Eclipse project, all elements contained in the project are reverse engineered and added as elements of a Rhapsody model. The imported elements are added in a new package which uses the name of the original Eclipse project. To import an Eclipse project, follow these steps:

1. Create a new Rhapsody project, or open an existing Rhapsody project.
2. Select **Tools > Import from Eclipse** from the main menu.
3. If Eclipse is not currently open, Rhapsody will launch it (after first asking you to confirm that you want Eclipse opened), and the Export Rhapsody Model dialog box will be displayed in Eclipse. (If the dialog does not appear, check the port settings used—**Code > IDE Options** from the main Rhapsody menu.)
4. Select the projects that you would like to export to Rhapsody. (If an Eclipse project has already been imported into Rhapsody, it will not appear in the list of available Eclipse projects.)
5. If you want the elements in the project to be added to the Rhapsody model as external elements, select the **Export as External** option.
6. If you want to fine-tune the reverse engineering options that will be used for importing your Eclipse project, select the option **Open Reverse Engineering options dialog in Rhapsody before export**.
7. Click **Finish**. All of the elements in the Eclipse project will be imported into the open Rhapsody project.
8. When the import process has been completed, look in the Rhapsody browser for a package that has the same name as the Eclipse project you imported. You will also notice that a

new component has been added to your Rhapsody model, containing an Eclipse configuration named after your Eclipse project.

Creating a New Eclipse Configuration

If the developer or designer prefers to use the Eclipse IDE (integrated development environment) to work on a project previously created in Rhapsody, follow these steps:

1. Display the existing Rhapsody C or C++ project in Rhapsody.
2. In the Rhapsody browser, right-click the Component in the Rhapsody project for which you want to create an Eclipse configuration.
3. Select **Add New > Eclipse Configuration** from the menus. The system displays a dialog box, asking whether or not the user wants to launch the IDE if it is not running.

Note: If IDE is already running, be certain that the ports for Eclipse and Rhapsody match by selecting the **Code > IDE options from the Rhapsody menus** and making any changes required. If the ports do not match, a new IDE may open even if the user meant to switch to the running IDE!

4. For the WindRiver version of Eclipse, the Workspace Launcher displays so that you can select a directory for your Eclipse project workspace. Click **OK** to save the selected directory.
5. Then the Rhapsody Project Wizard displays. It lists the name of the Rhapsody project and the component you selected for the new Eclipse configuration. Select whether you want to create a **New Project** or an **Existing Project**. Click **Finish**.
6. The New Project dialog box displays Wizard types. However, currently only the `vXworks Downloadable Kernel Module Project` is supported. Highlight it and click **Next**.
7. Type a **Project name** and select a workspace in this dialog box. Complete setting up the Workbench project, as described in the Workbench documentation.
8. The Application Development interface displays the Eclipse configuration of the selected Rhapsody component.
9. Back in Rhapsody, the browser now contains the new Eclipse configuration.

Troubleshooting Your Eclipse Installation with Rhapsody

If after installing Eclipse, working to set up a project as described previously, and you have not been successful, follow these troubleshooting steps to check your installation:

1. Check the path to Eclipse in the rhapsody.ini file to be certain that it is the actual path on your system. The following is an example of a typical path:

```
[IDE]
EclipsePath=C:\eclipse\eclipse.exe
```

Note: If it does not match your path to Eclipse, make the necessary changes.

2. Check to be certain that the `c:\cygwin\bin` is in your environment PATH variable.
3. Start Rhapsody and create or open a project.
4. Right-click the component and choose **Add New > Eclipse Configuration**.
5. You are prompted for a workspace. Then the Rhapsody Project Wizard asks you to either specify a project or create a new one. Click **Next**.
6. Specify the new project name and click **Next**.
7. Click **Finish**.
8. If the program asks if you want to associate with a C++ Perspective, click **Yes**. Now the Eclipse CDT IDE is open and linked to your Rhapsody project.
9. Switch to Rhapsody and make sure your Eclipse configuration is active. Generate code.
10. Right-click a model element in the Eclipse version of the code and select the **Locate in Rhapsody** option. This change should trigger a roundtrip in the Rhapsody version of the code if the change is not one of the [Workflow Integration with Eclipse Limitations](#).

Switching Between Eclipse and Wind River Workbench

When Rhapsody launches an Eclipse-based IDE, whether the generic Eclipse or Wind River Workbench, it checks the `rhapsody.ini` file to determine the path of the IDE.

Then entry that stores this information is called `EclipsePath` and it is located in the section `[IDE]`. During installation, you will be asked to provide the path for Eclipse, and this information is copied to the `rhapsody.ini` file, for example:

```
EclipsePath=1:\windriver\workbench-2.4\wrwb\2.4\x86-win32\bin\wrwb.exe
```

If you want to switch between the generic Eclipse and Wind River Workbench, change the value of this entry in the `rhapsody.ini` file.

Rhapsody Tags for the Eclipse Configuration

After creating an Eclipse configuration, the Rhapsody model elements are coupled with an Eclipse-based project. The definition of the Eclipse project is stored in Rhapsody in these *Tags* (displayed in the Rhapsody browser):

- ◆ **IDENAME** is the name of the type of integrated development environment (for example, Eclipse, Workbench).
- ◆ **IDEProject** is the project name entered while creating the Eclipse configuration.
- ◆ **IDEWorkspace** is the workspace directory for the Eclipse-based project.

The values of the tags are set automatically after the IDE project is created or mapped to the configuration. You do not need to modify the values of these tags, unless you change the IDE *workspace location* in the file system. If you make that change, then you must update the `IDEWorkspace` tag manually. See the next section, [Rhapsody Features Settings for Eclipse](#), for instructions to make this change if necessary.

Rhapsody Features Settings for Eclipse

To examine the features of an Eclipse configuration in Rhapsody, follow these steps:

1. Right-click the Eclipse configuration in the Rhapsody browser.
2. Select Features from the menu to display this dialog box. This version of the Features dialog box contains the **IDE**, **Tags**, **Properties**, and **Settings** tabs for use with Workbench projects.
3. The **IDE** tab provides two important features:
 - ◆ **Open in IDE** launches Eclipse with the corresponding project or simply bring the IDE forward.

- ◆ **Build configuration in IDE** sets the build to be performed via the IDE (by sending a request to the IDE)
4. However, if you want to perform the build in Rhapsody, use the **Settings** tab to make the build selections.
 5. Generally, you do not need to change the values for the **Tags**, unless you change the *IDEWorkspace directory location*. In that case, you must make the change to the path in the dialog box and click **OK**.

Eclipse Workbench Properties

To define the configuration and environment for the IDE project, the following Rhapsody properties are available in the [Rhapsody Features Settings for Eclipse](#):

Subject	Metaclass	Property Name	Default Value	Description
Eclipse	Configuration	InvokeExecutable	\$executable	Points to the executable. Keywords:\$executable - the IDE executable as read from Rhapsody.ini
Eclipse	Configuration	InvokeParameters	-data \$workspace - vmargs - DRhpClientPort=\$Rh pClientPort - DRhpServerPort=\$Rh pServerPort	Parameters for the command-line. Keywords: \$workspace as specified in the Rhapsody Tags for the Eclipse Configuration . \$RhpClientPort: the port number that Rhapsody uses to be a client to Eclipse, as specified using Rhapsody's menu Code > IDE options . \$RhpServerPort: the port number that Rhapsody uses to be a server to Eclipse
Eclipse	DefaultEnvironments	Eclipse	Cygwin	The default environment in the settings tab for generic Eclipse (CDT) projects
Eclipse	DefaultEnvironments	Workbench	WorkbenchManaged	The default environment in the settings tab for generic Eclipse (CDT) projects

Editing Rhapsody Code using Eclipse

To edit code from Rhapsody using Eclipse, follow these steps:

1. Open the Rhapsody project that contains the Eclipse configuration and make it the active configuration.
2. Launch Workbench.
3. In Rhapsody, right-click a class and select **Edit code in Eclipse** from the menu.
4. The implementation of that class is then displayed in Eclipse, and Eclipse automatically generates a file in DMCA mode in Rhapsody.
5. Edit the code as needed. The updates are recorded in the implementation of the class.

Locating Implementation Code in Eclipse

If you want to examine the implementation code for model elements or errors using Eclipse, follow these steps:

1. Open the Rhapsody project that contains the Eclipse configuration and make it the active configuration.
2. Launch Workbench.
3. In Rhapsody, right-click a model element, such as an attribute for a class in the browser, and select **Locate in Eclipse** from the menu.
4. The implementation code displays in Eclipse.

Opening an Existing Eclipse Configuration

After creating Eclipse configurations for the Rhapsody components that you want to work on in Eclipse, follow these steps to open Eclipse:

1. Start Rhapsody.
2. Select **Code > IDE Open ...** from the Rhapsody menu bar.
3. Eclipse launches. If the active configuration in the Rhapsody browser is an Eclipse configuration, that configuration is automatically displayed in Eclipse. If the active configuration is not an Eclipse configuration, the system displays a dialog box asking the *Eclipse workspace directory*. Then it displays the Eclipse configuration in the workbench interface.

Note: Only those elements that can be edited in Rhapsody can be opened for editing in Eclipse.

Disassociating an Eclipse Project from Rhapsody

To disassociate an Eclipse project from Rhapsody, follow these steps:

1. In the list of C/C++ or Java projects in Eclipse, right-click the project to display the context menu.
2. From the context menu, select **Rhapsody > Disconnect from Rhapsody**.
3. Click **OK** to confirm the disconnection.

This removes the Rhapsody characteristics from the project.

When you return to Rhapsody, you are asked whether you want to delete the corresponding Eclipse configuration from your model.

Note

In Rhapsody, if you delete the Eclipse configuration from the model, the Eclipse project will automatically be disconnected from the model.

Workflow Integration with Eclipse Limitations

When using the Rhapsody workflow integration with eclipse, operations originating in Rhapsody and continuing into Eclipse cannot be undone. In addition, the \$executable is mapped to a single IDE. To work with several IDEs, the user must set the properties.

Domain-specific Projects and the NetCentric Profile

The Rhapsody NetCentric Profile builds domain-specific projects for a Service Oriented Architecture (SOA) or to support Network Centric Warfare (DoDAF) Applications. In SOA projects, developers begin by writing the interface specifications using Web Service Description Language (WSDL), a complex XML schema. Unfortunately, WSDL requires that the data types be fully defined and contain legal XML types. WSDL also requires the developers to define the interface calls in terms of where the services are to be deployed (bindings and namespaces). These details are not always known during the initial design phase. The NetCentric profile helps the engineer bridge the gap from the design concepts to the WSDL file production.

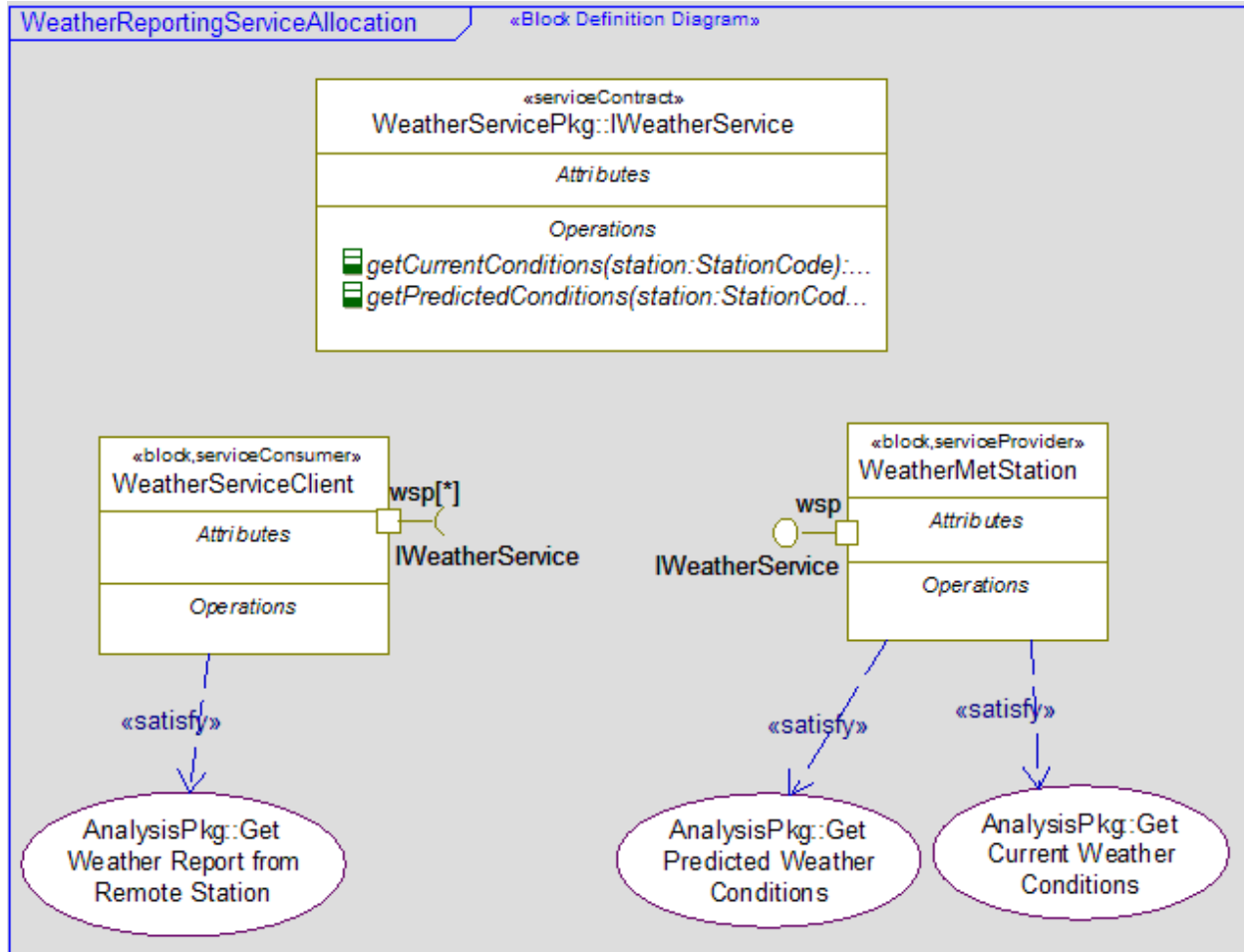
Designing a SOA or NetCentric Application Model

SOA and NetCentric Applications models have two special roles:

- ◆ Service provider or the service itself
- ◆ Service consumer is the user of the service

The service provider includes the interface specification (also called the service contract) and the related WSDL file.

The Block Definition diagram example below shows a typical collaboration in which the stereotypes indicate the service contract, service consumer, and service provider.



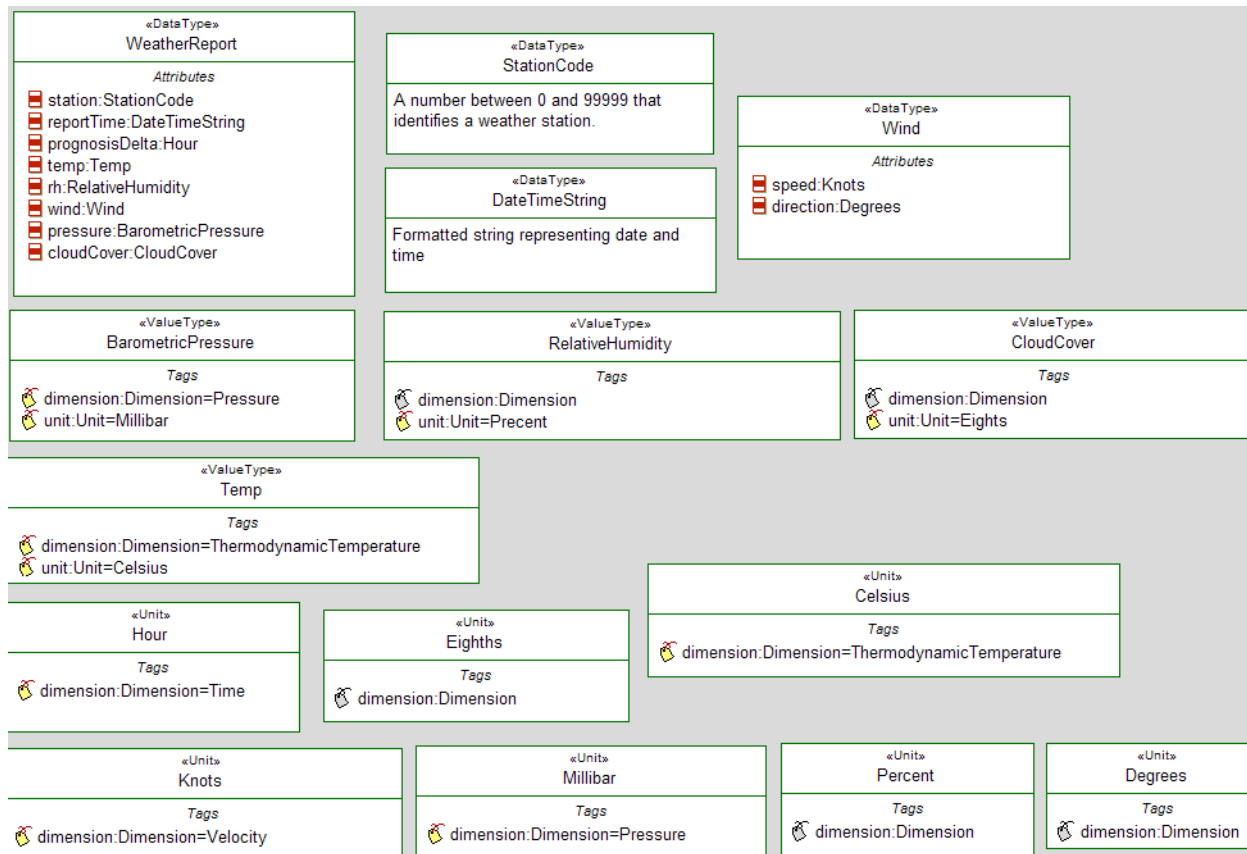
Collaborations in a top level or System of Systems model allow the systems engineers to confirm the data types, interfaces, and basic block behavior before generating the WSDL file. Executing the model ensures that both sides of the interface interpret messages the same way, thus avoiding consistency errors that otherwise would not be discovered until late in the integration phase.

Defining a Service Package for the Service Provider

The service provider is simply the code that implements the service. The service provider includes everything necessary to produce the WSDL file:

- ◆ Service contract (interface class)
- ◆ Realization of this contract (classes that implement the service)
- ◆ Data types
- ◆ WSDL file.

Rhapsody uses the stereotype `<<servicePackage>>` to indicate a UML/SysML package that contains the model elements necessary for creating the WSDL file that includes the package with the service provider and the service contract. The stereotypes `<<serviceProvider>>` and `<<serviceContract>>`, respectively, indicate these classes. To define the data types, the WSDL files also include XML schemas and XSDs. The systems engineers and designers model data types via the SysML units and value types, as shown the Block Definition diagram example below (from Rhapsody’s System samples “NetCentricWeatherService” project):



The Generate WSDL Specification tool uses these data types to create the schema information within the WSDL file.

Creating a WSDL Specification

To create a WSDL file from Rhapsody, follow these steps:

1. In a Rhapsody NetCentric project, define an interface block or class. This interface contains the methods that define the service that is called by the service consumer.
2. In the browser, select the interface block and right click.
3. Select **Change To > service Contract**. This action changes the term from `Interface` to `serviceContracts`. It also changes the stereotype from `<<Interface>>` to `<<serviceContract>>`.
4. Create the block or class that realizes the `serviceContract`. Apply the stereotype `<<serviceProvider>>` to it.
5. Add a standard port to the `<<serviceProvider>>`. Set the port to provide the service contract interface. Stereotype this port `<<servicePort>>`.
6. Define all of the data types needed by the interface.
7. Set the stereotype on the package to `<<servicePackage>>`.
8. Select **Tools > Generate WSDL Specification**.

Exporting a WSDL Specification File

To export a WSDL specification file from your Rhapsody project, follow these steps:

1. Open the NetCentric project containing the WSDL specification. It is stored as a `<<wsdlDefinitionDocument>>` stereotyped package.
2. Highlight that package in the project browser.
3. Select **Tools > Generate WSDL Specification**.
4. Enter the name and location for the output of WSDL specification file.
5. Click **OK** to export WSDL.

Using the Schedulability, Performance, and Time (SPT) Profile

Note

This functionality can only be used with Rhapsody in C++.

Overview

The SPT profile (also known as the UML Real-Time profile) is a standard UML profile adopted by the OMG in March 2002 (OMG Number: ptc/02-03-02).

The profile aims to:

- ◆ Enable the construction of models that could be used to make quantitative predictions regarding these characteristics.
- ◆ Facilitate communication of design intent between developers in a standard way.
- ◆ Enable interoperability between various analysis and design tools.

The SPT.rpy model provided in `<Rhapsody installation path>\Share\Profiles\SPT` is the Rhapsody implementation of the standard profile that you can use in any Rhapsody model.

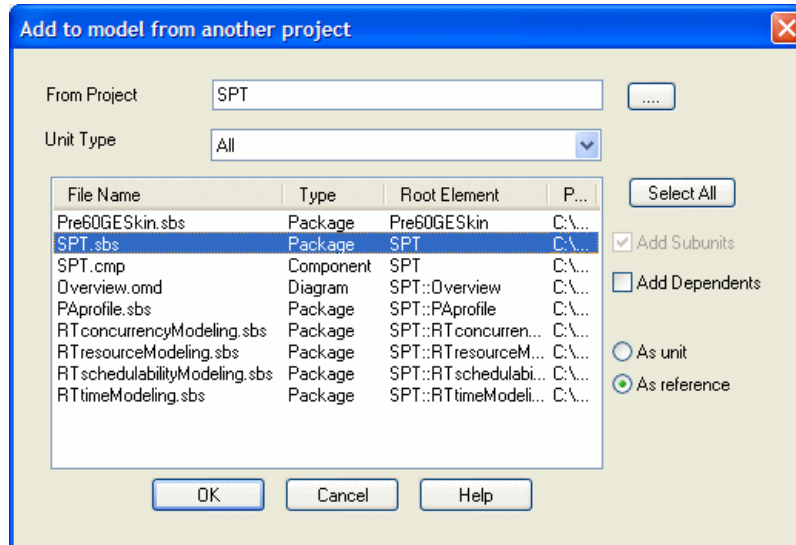
How to Use the SPT Model Provided with Rhapsody

To use the SPT profile you must enable it for use in your model and then use the stereotypes and tagged values provided by it.

Enabling the SPT Profile for use in your Model

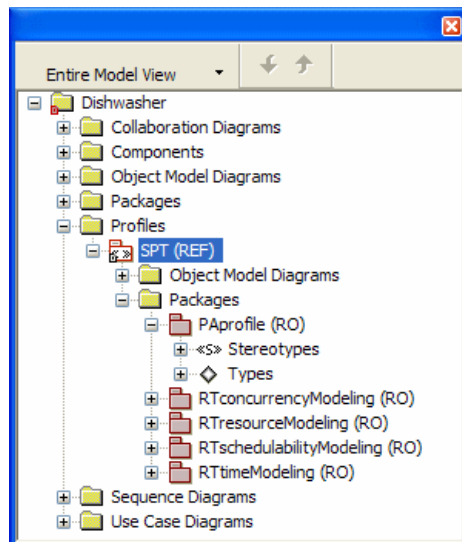
1. Open the Rhapsody model that will use the profile. For example, `MyModel.rpy`.
2. Select **File > Add to Model**.
3. On the Add to Model dialog box, navigate to the SPT model folder (for example, `<Rhapsody Installation>\Share\Profiles\SPT\`) and select **SPT.rpy**.
4. Click **Open**.
5. On the Add To Model From Another Project dialog box, select the **SPT.sbs** unit.

- Select the **As reference** radio button, as shown in the following figure:



- Click **OK**.

Rhapsody adds the SPT profile as a reference profile to your model. As a result, the stereotypes and tagged values of the profile become available, as shown in the following figure:



Using the Stereotypes and Tagged Values

1. Select the model element (for example, a class named `MyClock`).
2. Assign the stereotype you want to the model element (for example, `RTClock`).
3. Set tag values through the **Tags** tab of the Features dialog box for the element.

Changing the Profile

To change the profile, edit the SPT model.

Note

Deleting a stereotype or changing its metaclass may result in unresolved references in the models using it (that is, the models that are referencing the SPT model and have elements with this stereotype).

Co-Debugging with Tornado

Rhapsody enables you to connect to the Tornado IDE, download an executable component to the target, and perform source-code level debugging on the target while simultaneously performing design-level debugging on the Rhapsody host.

Integration of the Tornado IDE provides a seamless development workflow between Rhapsody and Tornado through the following functions:

- ◆ Downloading and reloading an image directly to the target from Rhapsody.
- ◆ Synchronizing Rhapsody breakpoints and Tornado breakpoints:
 - Tornado (gdb) is aware of Rhapsody-based breakpoints (break on state).
 - Rhapsody is alerted for source-level breakpoints from Tornado.

This appendix provides information on the following topics:

- ◆ [Preparing the Tornado IDE](#)
- ◆ [IDE Operation in Rhapsody](#)
- ◆ [Co-Debugging with the Tornado Debugger](#)
- ◆ [IDE Properties](#)

Preparing the Tornado IDE

To prepare the Tornado IDE for integration with Rhapsody, follow these steps:

1. Open Tornado, then select **Tools > TargetServer > <servername>**.
2. In Rhapsody, make sure the active configuration is set to VxWorks.

IDE Operation in Rhapsody

Rhapsody directly supports the following operations from the IDE submenu of the Code menu:

- ◆ **Connect.** Opens a connection to the IDE server. In the case of Tornado, this is the Tornado target server. This should be applied once during a session. The connection is disconnected either explicitly or when the project is closed. To connect, you must specify a target server name, typically <targetName>@<hostName>. Once specified, the name is stored for future sessions.
- ◆ **Download.** Downloads an image to the targets through the IDE. Download is enabled only if an image exists.

- ◆ **Run.** Runs the executable on the target starting from a designated entry point (VxMain in Tornado), as specified by the `<lang>_CG::<Environment>::EntryPoint` property. You can also run the executable from the Code menu or toolbar.
- ◆ **Unload.** Clears the target from all tasks and data allocated by the application. This is an important feature because RTOSes generally do not have a process concept that cleans up after termination of the application. Unload is always available and causes execution to stop if the application is running.
- ◆ **Disconnect.** Disconnects from the IDE server.

Co-Debugging with the Tornado Debugger

Before using the Tornado debugger, make sure to compile the generated file using debug flags (normally `-g`).

To use the Tornado debugger, follow these steps:

1. In Rhapsody, connect the application by selecting **Code > IDE > Connect**.
2. Download the application by selecting **Code > IDE > Download**.
3. Select **Code > IDE > Run**, or click **Run**.
4. In the **Animation** toolbar, select **Go Idle** (or **Go Step** several times) so the `τRhp` task is created.

Note: You must run the application before attaching a debugger; otherwise, there will be no tasks to which to attach the debugger.

5. In Tornado, start the debugger by selecting **Tools > Debugger**.
6. Attach the debugger to the main thread (`τRhp`) by selecting **Debug > attach**.
7. From the debugger, change directory to the generated code directory (using the `cd` command in the `gdb` prompt).
8. From the debugger, load the symbols of the executable (using the `add-symbol-file` command at the `gdb` prompt).

Now you can use `gdb` to debug the application, set breakpoints, and so on.

Before quitting animation on Rhapsody, you must detach the debugger using **Debug > Detach**. Failing to detach the debugger might block the session once Rhapsody attempts to unload the image.

Note

Do not download the executable to the target using the debugger. Rhapsody will not function properly if you use this method.

IDE Properties

The following Rhapsody properties (under `<lang>_CG::<Environment>`) determine IDE settings:

- ◆ `HasIDEInterface`

If this property is set to `Cleared`, no IDE services are attempted and IDE support is disabled. If the property is `Checked`, it is expected that the property `IDEInterfaceDLL` points to an IDE adapter that provides connection to the IDE. This property can be used to disable the IDE connection. By default, it is set to `Checked` only for the `VxWorks` environment.

- ◆ `IDEConnectParameters`

This property specifies the IDE connection parameters. If this property is defined, Rhapsody will use the connection parameters from this property instead of the `.ini` file.

- ◆ `IDEInterfaceDLL`

This property points to the IDE adapter DLL. Currently, there is no reason to modify the value of this property.

For detailed information on a property, see the definition displayed in the **Properties** tab of the Features dialog box or examine the complete list of property definitions in the *Rhapsody Property Definitions* PDF file available from the *List of Books*. That list can be searched along with the other PDF versions of the Rhapsody documentation to locate specific property definitions and the procedures that use them.

Creating Rhapsody SDL Blocks

Systems engineers often use the System Design Language (SDL) to model discrete (event driven) algorithms. The SDL Suite also generates C code for its models. Rhapsody in C++ is integrated with the SDL Suite (version 5.0 or greater) to enable system simulation based on Rhapsody and the SDL Suite's discrete behavior. Engineers can import an SDL model into Rhapsody. Rhapsody manages the imported model as a class, stereotyped with the `SDLBlock`.

Note

The naming convention for an SDL signal adds the “_” prefix to the signal's original name. This prefix can be modified by changing the `SDLSignalPrefix` property in the `Model::Profile` group.

By default the `SDLBlock` uses behavioral ports. This configuration can be changed to use a rapid port instead by selecting the `UseRapidPorts` property for the package. This property is also stored in the `Model::Profile` group that you access from the **Properties** tab of the Features dialog box.

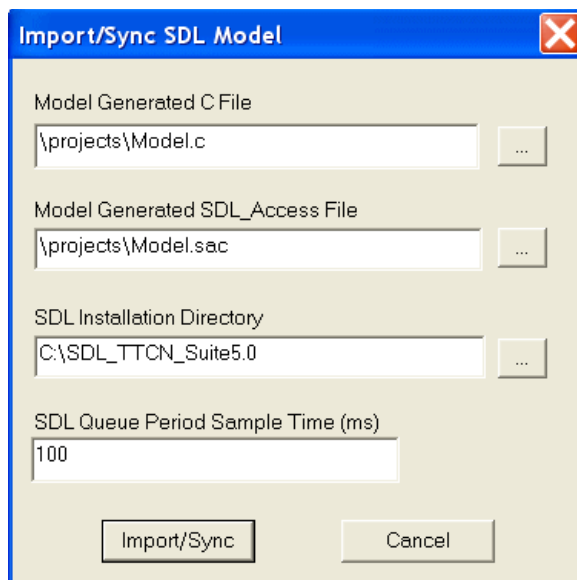
Note

The SDL models you import into Rhapsody cannot contain more than a single instance of any given process.

To import an SDL model into Rhapsody, follow these steps:

1. In the SDL Suite, open the SDL model. Mark the System level rectangle.
2. From the main menu select **Generate > Make**.
3. Select the **CAdvanced** Code Generator configuration.
4. Select the Generate environment header file check box.
5. Activate the “Make” to generate the model C file (modelname.c) and environment header file (modelname.ifc).
6. Select **SDLAccess** Code Generator configuration and activate the **Full Make** to generate the model `SDL_Access` file (modelname.sac).
7. Open Rhapsody and select **File > New**.
8. Select the **SDL_Suite** for the project **Type**.
9. Create a new block/class and select the **SDLBlock** class stereotype.
10. Right-click this block and select the **Import/Sync SDL Model** option from the menu.

11. Enter the locations of the SDL model files you created previously, as shown in the following figure.



12. Click **Import/Sync**.
13. To connect the Rhapsody block to an SDLBlock, create a user class with behavior ports and a statechart. The statechart controls the user class' sending and receiving of events to and from the SDLBlock.
14. Create objects from the SDLBlock and the Rhapsody block and connect their ports via links using the interfaces that were created by the import.
15. To create an executable, perform a code generation and build on the entire Rhapsody model. Code generation scope should contain only one SDLBlock.

Note

Since the SDLBlock is imported as a “black box,” no animation is provided with this block. There is an option to view the behavior of the SDLBlock as a wrapper via a sequence diagram. This can be done by checking the `AnimateSDLBlockBehavior` property, located in the `Model::Profile` property group.

Using Model Elements

The Rhapsody browser lists all the design elements in your model in a hierarchical, expandable tree structure, enabling you to easily navigate to any object in the model and edit its features and properties. The Rhapsody browser also takes part in animation by displaying the values of instances as they change in response to messages and events.

To help you manage large and complex Rhapsody projects, and to be able to focus on and easily access model elements of particular interest to you, you can filter the Rhapsody browser or create other browser views.

Using the Rhapsody Browser

The Rhapsody browser displays a list of project elements organized into folders. You can choose to have all elements displayed in a single folder, regardless of their position in the model, or have them displayed in subfolders based on the model hierarchy. The browser provides several views so you can filter the display of elements by different design categories. The project is the top-most folder in the Rhapsody browser. It contains the following top-level folders:

- ◆ **Components**, which contains one or more configurations and files
- ◆ **Packages**, which contains actors, classes, events, globals, diagrams, types, use cases, and other packages
- ◆ **Diagrams**, which contains any of the UML diagrams that Rhapsody supports

You can organize large projects into package hierarchies that can be viewed easily by nesting packages, components, and diagrams inside other packages, and nesting classes and types inside other classes.


Because a Rhapsody project can get quite large and complex, you may want to filter what you see on the Rhapsody browser or otherwise create other browser views. For these situations, you may want to see the following topics:

- ◆ [Filtering the Browser](#)
- ◆ [Opening the Browse From Here Browser](#)
- ◆ [Using the Favorites Browser](#)

Opening the Rhapsody Browser

By default, the Rhapsody browser is displayed the first time you open a project. In subsequent work sessions, Rhapsody consults your workspace file (`<project name>.rpw`) to determine whether to open the browser when it opens a project. For more information on workspaces, see [Controlling Workspace Window Preferences](#).

To open the Rhapsody browser manually, follow these steps:

- ◆ Click the Show/Hide Browser button  on the Rhapsody **Windows** toolbar.
- ◆ Select **View > Browser**.
- ◆ Press **Alt+0** (zero).

Setting the Display Mode

The browser has two display modes: Flat and Categories. In Flat mode, only the components, packages, and diagrams within each package have separate categories. In Categories mode, all elements are organized into categories based on their position in the project hierarchy. The default display mode is Categories mode.

1. Set focus to the browser.
2. Select **View > Browser Display Options > Organize Tree** and then choose the appropriate option:
 - ◆ **Flat** to hide the categories
 - ◆ **Categories** to display the categories

The display mode for the project is stored in the property `Browser::Settings::DisplayMode`. Set the property to `Meta-class` for Categories mode, or `Flat` for Flat mode.

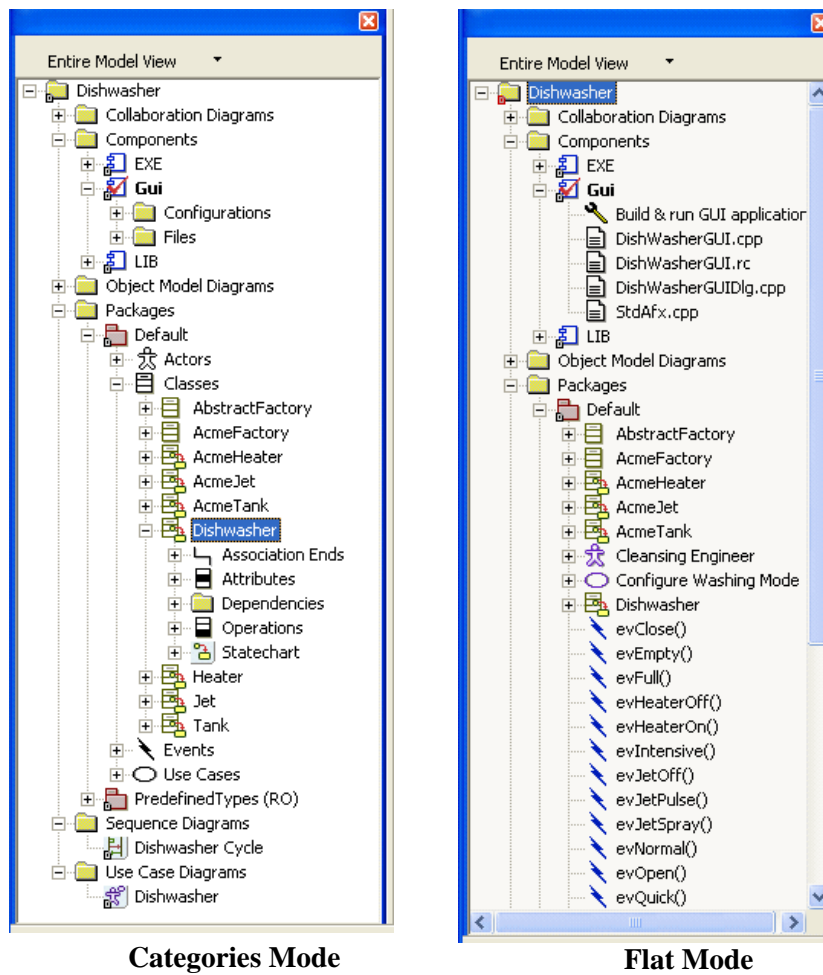
Expanding a category while in Flat mode reveals a flat list of elements of all types included under that category. For example, expanding a package category reveals a simple list of the elements contained in the package, such as actors, classes, and events, arranged alphabetically by name.

In Categories mode, each metatype appears in its own category. Individual items, such as components, diagrams, packages, classes, and actors, are displayed under the appropriate category.

Note















To display the labels defined for model elements (instead of their names), select **View > Browser Display Options > Show Labels**.

















The following figure shows the same model displayed in both browser modes.



















Basic Browser Icons

The icons before elements listed in the browser provide additional information about the elements so that you can quickly identify items you want to access. The following chart summarizes the standard browser icons, but not the icons exclusive to the speciality profiles, such as DoDAF. See the sections describing special features for the browser icons that are used.

Browser Icon	Identifies in the Rhapsody Project
	Folder used to group and organize project elements.
	Folder that is a unit. The square in the lower left corner indicates that this folder is a unit. See Using Project Units for more information.
	Folder of hyperlinks. See Hyperlinks for more information.
	Component is a physical subsystem in the form of a library or executable program or other software components such as scripts, command files, documents, or databases.
	Component that is a unit. The square in the lower left corner indicates that this component is a unit.
	Component that is a unit and is set as the active component with a red check. To be the active component, a component must be either an Executable or a Library build type. See Active Component for more information.
	Executable <i>application</i> or a <i>library</i>
	Actor represents an end user of the system, or an external component that sends information to or receives information from the system. See Actors for more information.
	Class defines the attributes and behavior of objects. See Classes for more information.
	Class with a <i>statechart</i>
	Class <i>attributes</i>
	Class <i>operations</i>
	<i>File</i> indicates an imported file.
	<i>Part</i> is a component or artifact of a system.

Browser Icon	Identifies in the Rhapsody Project
	<i>Event</i> is an asynchronous, one-way communication between two objects (such as two classes). See Events and Operations for more information.
	<i>Use case</i> captures scenarios describing how the system could be used. See Use Cases for more information.
	<i>SuperClass</i> marks a class that inherits from another class. See Inheritance for more information.
	<i>Dependency</i> notes the relationship of a dependent class to a model element or external system that must provide something required by the dependent class. See Dependencies for more information.
	<i>Constraint</i> supplies information about model elements concerning requirements, invariants in a text format.
	<i>Stereotype</i> is a type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. See Defining Stereotypes for more information.
	<i>Type</i> is a stereotype of class used to specify a domain of instances (objects) together with the operations applicable to the objects. A type cannot contain any methods. See Types for more information.
	<i>State</i> is an abstraction of the mode in which the object finds itself. It is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. See States for more information.
	<i>Default transition</i> marks the default state of an object. See Transitions for more information.
	<i>Controlled file</i> is produced in other programs, such as Word or Excel, that are added to a project for reference purposes and then controlled through Rhapsody. See Controlled Files for more information.
	<i>Association</i> defines a semantic relationship between two or more classifiers that specify connections among their instances. It represents a set of connections between the objects (or users). See Associations for more information.
	<i>Flow port</i> represents the flow of data between blocks in an object model diagram (OMD) without defining events and operations. Flowports can be added to blocks and classes in object model diagrams.
	<i>Profile</i> applies domain-specific tags and stereotypes to all packages available in the workspace. See Using Profiles for more information.
	<i>Requirement</i> is a desired feature, property, or behavior of a system. Requirements can be imported or created in Rhapsody.
	Requirement verification
	<i>Tags</i> add information to certain kinds of elements to reflect characteristics of the specific domain or platform for the modeled system. See Using Tags to Add Element Information for more information.

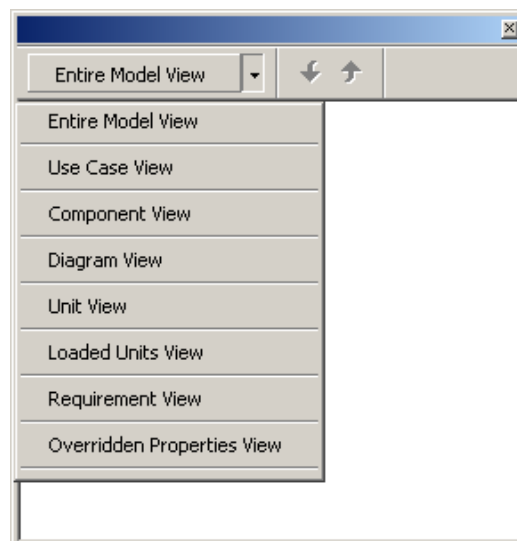
Browser Icon	Identifies in the Rhapsody Project
	Flow item
	<i>Comments</i> icon marks text added to a model element.
	<i>Table layout</i> is the design for a table view of project data. See Creating Table and Matrix Views for more information.
	<i>Table view</i> displays project data in the predefined table layout.
	<i>Matrix layout</i> is the design for a matrix view of project data.
	<i>Matrix view</i> displays project data in the predefined matrix layout.
	<i>Activity diagram</i> shows the lifetime behavior of an object, or the procedure that is executed by an operation in terms of a process flow, rather than as a set of reactions to incoming events. See Activity Diagrams for more information.
	<i>Collaboration diagram</i> displays objects, their messages, and their relationships in a particular scenario or use case. This diagram is also a unit. See Collaboration Diagrams for more information.
	<i>Component diagram</i> specifies the files and folders that components contain and defines the relations between these elements. See Component Diagrams for more information.
	<i>Deployment diagram</i> shows the configuration of run-time processing elements and the software component instances that reside on them. See Deployment Diagrams for more information.
	<i>Object model diagram</i> shows the static structure of a system: the objects in the system and their associations and operations, and the relationships between classes and any constraints on those relationships. See Object Model Diagrams for more information.
	<i>Requirements diagram</i> shows requirements imported from other software products or created in Rhapsody and illustrate the relationships between requirements and system artifacts. See Creating Rhapsody Requirements Diagrams for more information.
	<i>Sequence diagram</i> describes message exchanges within your project. See Sequence Diagrams for more information.
	<i>Statechart</i> defines the behavior of objects by specifying how they react to events or operations. See Statecharts for more information.
	Structure diagram models the structure of a composite class; any class or object that has an object model diagram can have a structure diagram. See Structure Diagrams for more information.

Browser Icon	Identifies in the Rhapsody Project
	<i>Use case diagram</i> illustrates the system's scenarios (use cases) and the actors that interact with them. The icon in this example indicates that this use case diagram is also a unit. See Use Case Diagrams for more information.

Filtering the Browser

The large size and nested hierarchy of a Rhapsody project may complicate the process of locating and working with model elements. To help you navigate the Rhapsody browser more easily, the browser has a filtering mechanism that you can use to display only the elements relevant to your current task.

To display the filter menu, as shown in the following figure, click the down arrow button at the top of the browser. Whatever view you have selected is reflected in the label to the left of the arrow button.



Select one of these views:

Note

If the browser is filtered, you can add only elements that appear in the current view.

- ◆ **Entire Model View** is the default view and it does not use a filter. It displays all model elements in the browser.
- ◆ **Use Case View** displays use cases, actors, sequence diagrams, use case diagrams, and relations among use cases and actors.
- ◆ **Component View** displays components, nodes, packages that contain components or nodes, files, folders, configurations, component diagrams, and deployment diagrams. This view helps you manage different components within your project and assists with deploying them.
- ◆ **Diagram View** filters out all elements except diagrams. It displays all the diagrams, including statecharts and activity diagrams.

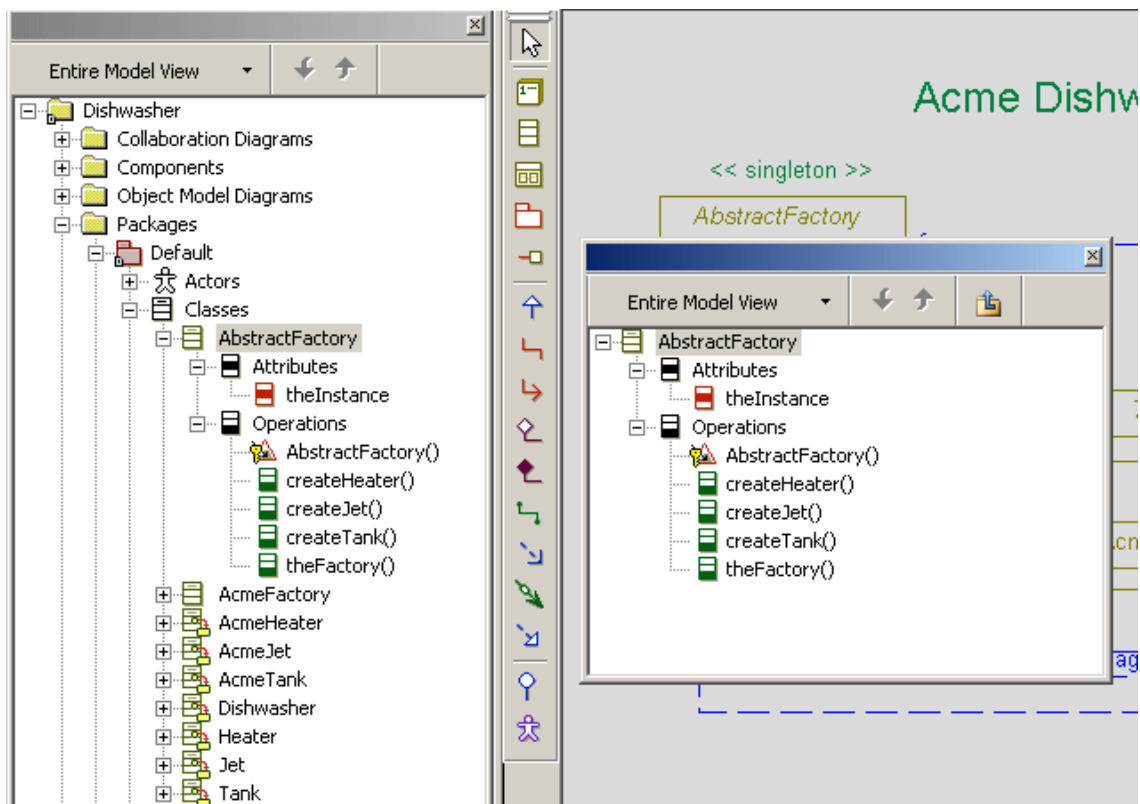
- ◆ **Unit View** displays all the elements that are also units. This view assists with configuration management. For information on creating and working with units, see [Using Project Units](#).
- ◆ **Loaded Units View** displays only the units that have not been loaded into your workspace. See [Loading/Unloading Units](#) and [Unloaded Units](#) for more information
- ◆ **Requirement View** displays only those elements with requirements.
- ◆ **Overridden Properties View** displays only those elements with overridden properties.

To learn about other browser views, see [Opening the Browse From Here Browser](#) and [Using the Favorites Browser](#).


Opening the Browse From Here Browser

Rhapsody projects can become very large and complex, making it difficult to find commonly used model elements in the Rhapsody browser. To help limit the scope of the current view of the browser, you can open a Browse From Here browser that contains the view of the browser that you want. The Browse From Here browser is similar to the main Rhapsody browser except that it typically shows a more focused area of the main Rhapsody browser.

The following figure shows the main Rhapsody browser on the left and a Browse From Here browser on the right:



The Browse From Here browser behaves just as the main Rhapsody browser does, and it has the same look-and-feel. You can drag-and-drop between the main Rhapsody browser and one or more Browse From Here browsers.

The one unique item that a Browse From Here browser has that the main Rhapsody browser does not is the Up One Level button .

For another method to help you view and access only those model elements you are most interested in, see [Filtering the Browser](#) and [Using the Favorites Browser](#). Note the following:

- ◆ The Browse From Here browser lets you view model elements in a tree structure while the Favorites browser has a flat structure.
- ◆ You can have the main Rhapsody browser, the Favorites browser, and one or more Browse From Here browser open at any time.

Opening a Browse From Here Browser


Note that you can open multiple Browse From Here browsers.

To open a Browse From Here browser, right-click a model element in the main Rhapsody browser, the Favorites browser, on a diagram, or another Browse From Here browser and select **Browse from here**.

Note

Browse From Here is not available for elements inside sequence diagrams and collaboration diagrams that are view-only elements.


Closing a Browse From Here Browser

To close a Browse From Here browser, click the Close button  for that browser.

Note that **View > Browser** is only for the main Rhapsody browser.

Navigating a Browse From Here Browser

You navigate a Browse From Here Browser as you would the main Rhapsody browser.

To set the root to the parent of the current root, click the Up One Level button . Note that, if you want, you can click this button as many times as needed to go to the project's root folder (so that your Browse From Here Browser ends up looking exactly like your main Rhapsody browser).

Browse From Here Browser Limitations

Browse From Here browsers are not saved when you close your project. Meaning that they will not be opened or available when you open your project the next time.

Deleting Items from a Browser

You can delete items from your project through the main Rhapsody browser and the Browse From Here browser.

To delete an item from your project, follow these steps:

1. Highlight the item(s) in the main Rhapsody browser or the Browse From Here browser.
2. Right-click and select **Delete from Model** from the pop-up menu, or choose **Edit > Delete** from the main menu.

The system asks for confirmation of the deletion operation.

3. Click **OK** to approve it.

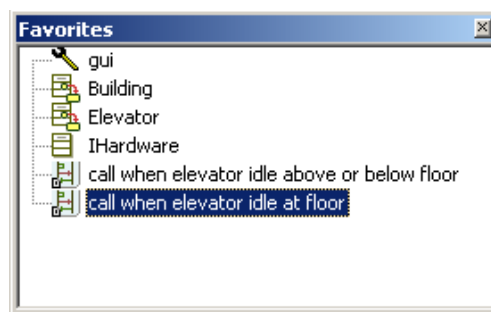
Note

Delete from Model is not available for the Favorites browser (see [Using the Favorites Browser](#)). You can highlight one or more items and choose **Edit > Delete** from the main menu to delete items from the Favorites browser (which is the same if you right-click and select **Remove from Favorites** from the pop-up menu). However, this only means that you are removing items from the Favorites browser. You cannot delete any model elements through the use of the Favorites browser.

Using the Favorites Browser

Rhapsody models can become very large, making it difficult to find commonly used model elements in the Rhapsody browser. To help you manage large and complex projects, and to be able to focus on and easily access model elements of particular interest to you, you can create a favorites list that displays in the Favorites browser. Like the favorites functionality for a Web browser, in Rhapsody, you can create a list of model elements you want to concentrate on that is displayed in the Favorites browser.

The following figure shows a sample Favorites browser:



Note that while the Favorites browser resembles the Rhapsody browser and has some of its functionality (for example, double-clicking an item on the Favorites browser will open the applicable Features dialog box, the view is refreshed when an item is renamed, it can be docked and undocked, and so forth), the Favorites browser has limited functionality (for example, there is no filtering mechanism, the elements appear as a flat list rather than a hierarchical tree structure, certain commands—such as **Cut**, **Copy**, **Paste**, **Add New**, and **Unit**—are not available, and while you can remove items from the Favorites browser, you cannot use it to delete a model element from your model).

Note


You cannot use the Favorites browser as a replacement for the Rhapsody browser.

Your favorites list is saved in the `<projectname>.rpw` file, while the visibility and position of the Favorites browser are saved in the `Rhapsody.ini` file, so that when you open the project the next time, your settings will automatically be in place. Note that when multiple projects are loaded, the Favorites browser shows the favorites list for the active project.

Creating your Favorites List

You cannot put a model element on your favorites list more than once. In addition, the Favorites browser automatically opens when you add a favorite item to it.

To create your favorites list, use any of the following methods:

- ◆ Highlight a model element in the Rhapsody browser, the Browse From Here browser, or on a diagram and click the Add to Favorites button  on the **Favorites** toolbar. (This toolbar should display by default. If it does not, choose **View > Toolbars > Favorites**).
- ◆ Highlight a model element in the Rhapsody browser, the Browse From Here browser, or on a diagram and press **Ctrl+d**.
- ◆ Right-click a model element in the Rhapsody browser or on a diagram and select **Add to Favorites**.

Note

Add to Favorites is not available for elements inside sequence diagrams and collaboration diagrams that are view-only elements.

Removing Items from your Favorites List

When you remove items from your favorites list, you are only removing them from the Favorites browser. You are not deleting them from your model.

To remove items from your favorites list, follow these steps:



1. Open the Favorites browser (see [Showing and Hiding the Favorites Browser](#)).
2. Use any of the following methods:

Note: The item is removed once you execute the action. If you change your mind, you can add the item to your favorites list again (see [Creating your Favorites List](#)).

- ◆ Right-click one or more items and select **Remove from Favorites**
- ◆ Highlight one or more items and press the **Delete** key.
- ◆ Highlight one or more items and choose **Edit > Delete**.

Showing and Hiding the Favorites Browser

To show or hide the Favorites browser, use any of the following methods:

- ◆ To open the Favorites browser, add a model element to your favorites list. See [Creating your Favorites List](#).
- ◆ Click the Show/Hide Favorites button .
- ◆ Select **View > Favorites**.
- ◆ To close the Favorites browser when it is open, click the Close button  for the browser.

Favorites Browser Limitations

Note the following limitations for the Favorites browser:

- ◆ You cannot give a name that is different from the referenced element to a favorite, like you can for a hyperlink or a favorite in a Web browser.
- ◆ Since you can put different model elements with the same name on your favorites list, there is a chance for confusion. To try to avoid this, notice the icon to the left of the model element name. The icons provide you with a clue as to what type of model item you have on your favorites list, as shown in the following figure:



- ◆ You cannot re-order the items on your favorites list.

Adding Elements

In the browser, you can add new elements to the model either from the Edit menu or from the pop-up menu. The location you select in the browser hierarchy determines which model elements you can add. The new element is added in the scope of the current selection.

- ◆ Select the project folder or a package, then select **Edit > Add New > Component**.
- ◆ Right-click the project folder or a package, then select **Add New > Component** from the pop-up menu.

Note

If the browser is filtered, you can add only elements that appear in the current view.

When you add a new association end, a dialog box opens so you can select the related model element.

1. Right-click the actor, then select **Add New > Association End** from the pop-up menu. The Add Association/Aggregation dialog box opens, as shown in the following example.



2. From the drop-down list, select the actor (or class) with which the current actor needs to communicate.
3. Click **OK** to apply your changes and close the dialog box.

When you create a new element, Rhapsody gives it a default name. You can edit the name of your new element by typing a new name directly in name field in the browser, or by opening the Features dialog box and entering a new name in the **Name** field.

Settings in the Browser

When you open an existing project in a newer version of Rhapsody, the system adds a compatibility profile in the Settings folder, as shown in this example.



See [Using Profiles](#) for more information about the Settings folder and profiles.

Adding a Rhapsody Profile Manually

To add a profile manually, open an existing project and follow these steps:

1. Select the **File > Add to Model** option. Navigate to your Rhapsody installation and open the Share/Profiles folder.
2. In the Add to Model dialog box, change the **Files of Type** field to display `Package (*.sbs)`. This lists all of the available profiles either as separate .sbs files or in folders.
3. Select a profile's .sbs file and click **Open**. The system checks to be certain that the selected profile is compatible with the language being used in the existing project.

Rhapsody lists the new profile in the **Profiles** section of the browser, as shown below.



Editing Component Elements

A *component* is a physical subsystem in the form of a library or executable program. It plays an important role in the modeling of large systems that contain several libraries and executables. For example, the Rhapsody application has several dozen components including the graphic editors, browser, code generator, and animator, all provided in the form of a library.

A component contains configurations and files. In the Rhapsody hierarchy, a component can be saved at the project level, or grouped within a package.

For instructions on editing components, see [Component Diagrams](#).

Configurations

A *configuration* specifies how the component is to be produced. For example, the configuration determines whether to compile a debug or non-debug version of the subsystem, whether it should be in the environment of the host or the target (for example, Windows versus VxWorks), and so on. For more information, see [Component Diagrams](#).

Configuration Files

Configurations manifest themselves as *files*. The ability to map logical elements to files enables you to better specify implementations for code generation, and to capture existing implementations during reverse engineering. Just as it might be desirable to map several classes into a single package, so might it be desirable to map one or more packages into a single subsystem. You control where to generate the source files for the classes (or packages) in a given subsystem, either into a single directory or separate directories.

- ◆ Which logical elements, or classes, to map into which files
- ◆ The order of the classes in the file
- ◆ Verbatim code chunks, such as macros and `#define` statements, that should be included in generated source files
- ◆ Whether to map each class to its own specification and implementation files, or to map multiple classes to the same files
- ◆ Whether specification and implementation files generated for a class, or set of classes, have the same name or different names
- ◆ Dependencies between classes
- ◆ File scope definitions for user-defined code

For more information on files, see [Component Diagrams](#).

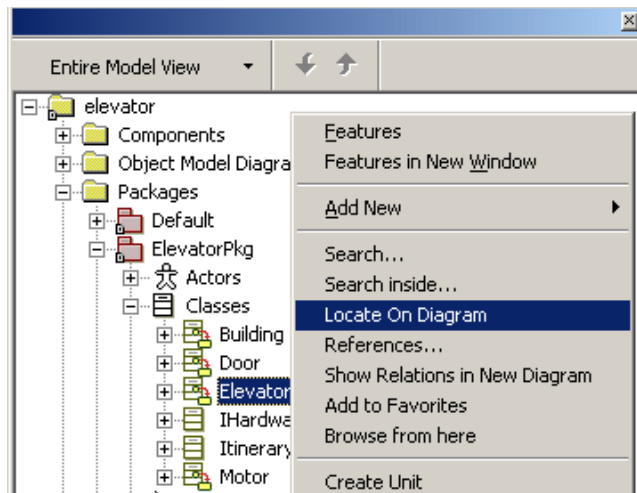
Locating an Element on a Diagram

You can use the Rhapsody browser and any code view window to quickly locate an element on a diagram by using **Locate On Diagram** from the pop-up menu.

To use the Rhapsody browser to locate an element on a diagram, follow these steps:

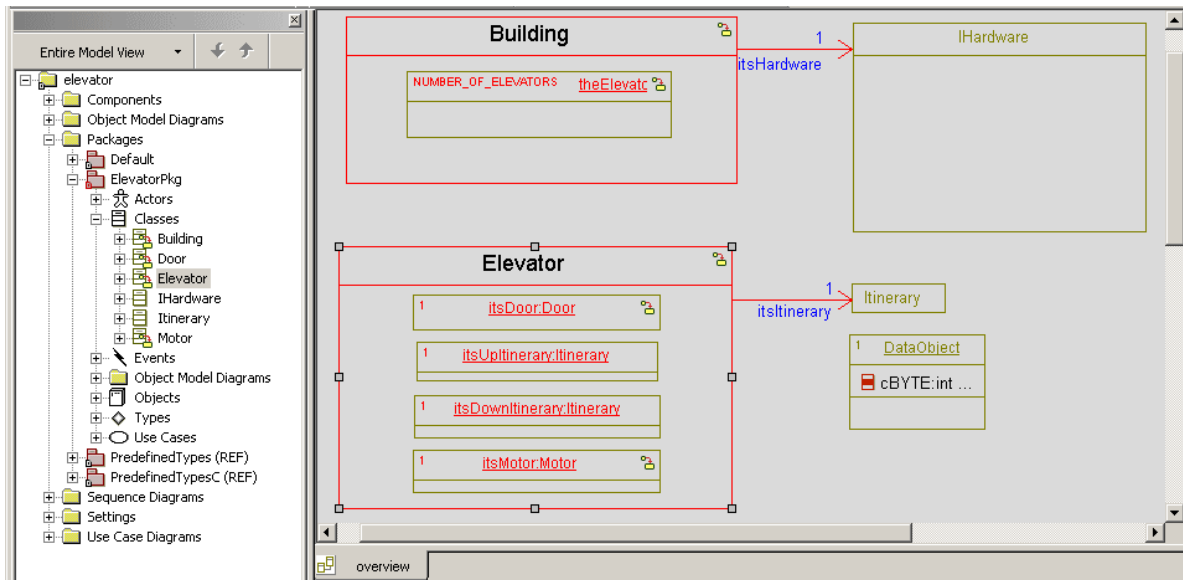
Note: These steps are the same from any code view window, except that you would select the element from the code. For an example, see [Example of Locate On Diagram from Code View](#).

1. Right-click an element (**Elevator**, as shown in the following figure on the left) in the Rhapsody browser and select **Locate On Diagram** from the pop-up menu.



2. Notice that Rhapsody opens a diagram in which the element (**Elevator**) is displayed, as shown in the following figure.

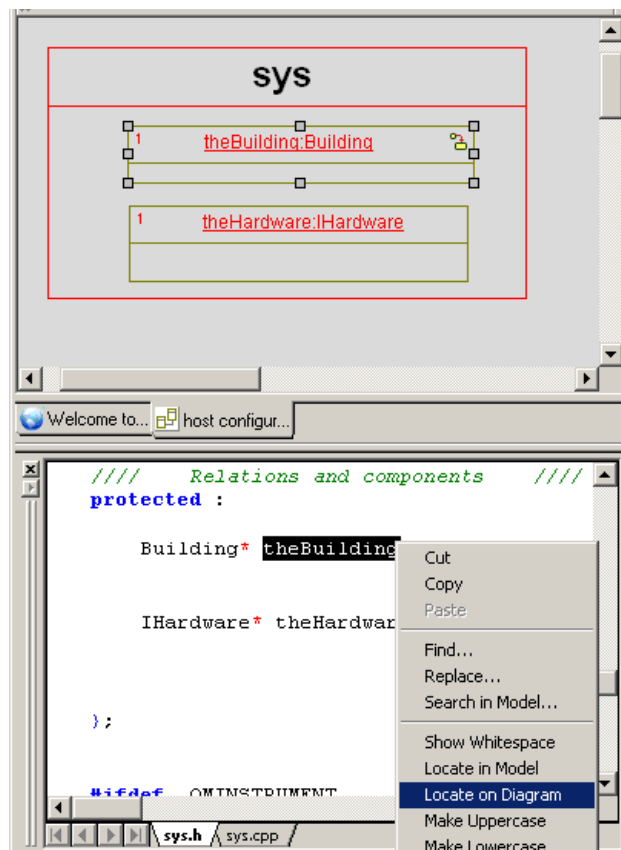
Note: If no diagram exists, a No Diagram Found message appears instead.



Example of Locate On Diagram from Code View

The following figure shows an example of selecting **Locate On Diagram** from a code view (as shown in the lower half of the figure) and the diagram it found (as shown in the upper half of the figure).

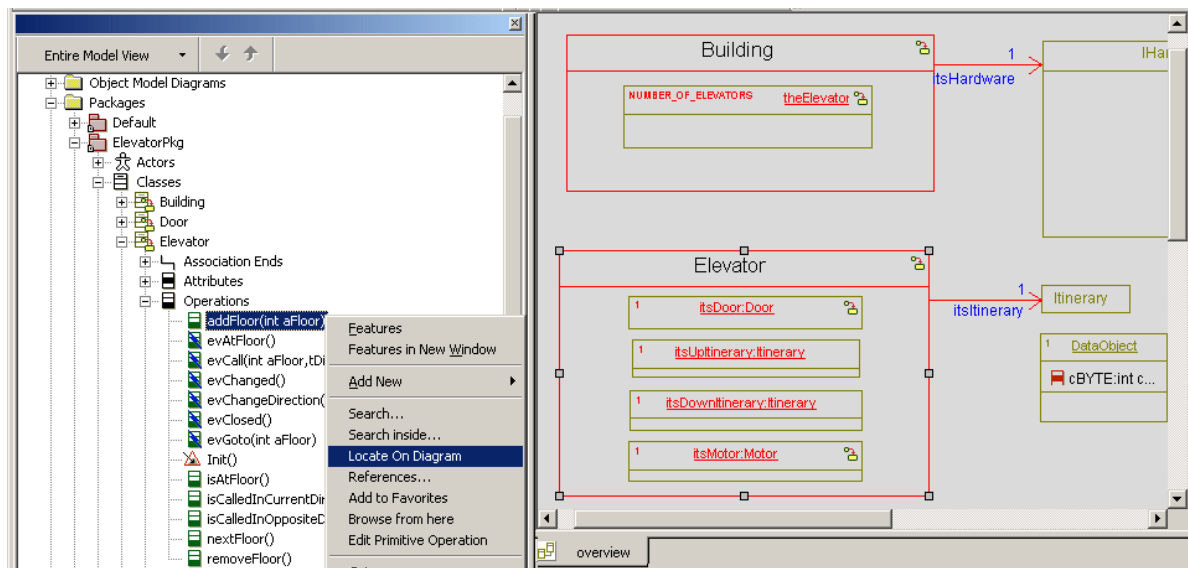
Note: The **theBuilding** element in the code view half of the following figure is selected for illustrative purposes. You can simply place your cursor within the letters of the element name or that particular line of code and right-click to display the pop-up menu and select **Locate On Diagram**.



Locate On Diagram Rules

Locate On Diagram uses the following rules to determine what to open, in order of priority:

1. Open the main diagram of the element if the main diagram is set and does show the element (as illustrated in the previous figure).
2. For an object, show the main diagram of its class if the main diagram is set and does show the object.
3. Show a randomly chosen diagram that shows the element.
4. Look for its container only if it does not find the element on the diagram. For example, if you try to locate an operation, Rhapsody looks for a diagram that shows the operation's class, as shown in the following figure.



Limitations

The ability to use **Locate On Diagram** from the code view is based on the annotations in the code. If the code is not annotated, **Locate On Diagram** will not be able to find the diagram. Note also in this case that it will not display a No Diagram Found message.

Using Package Elements

A Rhapsody project contains at least one package. *Packages* divide the system into functional domains, or *subsystems*, which can consist of objects, object types, functions, variables, and other logical artifacts. Packages do not have direct responsibilities or behavior—they are simply containers for other objects. They can be organized into hierarchies, providing the system under development with a high level of partitioning. When you create a new model, it will always include a default package, called “Default,” where model elements are saved unless you specify a different package.

Packages provide a way to group large systems into smaller, more manageable subsystems. Packages are primarily a means to group classes together into high-level units, but they can also contain diagrams and other packages.

These elements are described in detail in subsequent sections.

Classes (C++/J)	Collaboration diagrams
Comments	Sequence diagrams
Constraints	Structure diagrams
Dependencies	Object model diagrams
Flow charts	Statecharts
FlowItems	Tags
Flows	Stereotypes
Functions (C/C++)	Events
Hyperlinks	Component diagrams
Object _types (C)	Activity diagrams
Objects	Actors
Packages	Deployment diagrams
Receptions	Files
Requirements	Use case diagrams
Types (C/C++)	Nodes
Variables	Use cases

1. Right-click the project or the package category, then select **Add New > Package**.
2. Edit the name of the package using the package label in the browser.
3. Double-click the new package to open the Features dialog box.
4. Select a stereotype for the package (if necessary) from the **Stereotype** drop-down list.
5. Select an object model or use case diagram as the package’s main diagram from the **Main Diagram** drop-down list.

6. Enter a description for the package in the **Description** field.

In the generated code, this text becomes a comment located after the `#define` statements and before the `#include` statements at the top of the `.h` file for the package.

Functions

The *Functions* category lists global functions, which are visible to all classes in the same package.

Creating a Global Function

1. Right-click the name of a package or the **Functions** category in the browser.
2. Select **Add New > Function**. A new function appears under the Functions category of the selected package.
3. Edit the name of the new function in the browser.

Objects

Objects are instances of classes in the model and can be used by any class. They are typically created in OMDs, but can be added from the browser. For instructions on creating an object in an OMD, see [Object Model Diagrams](#).

1. Right-click the name of a package or the *Objects* category in the browser.
2. Select **Add New > Object**. The new object is displayed in the browser with the default name `object_n`.
3. Rename the new object as desired.

Variables

Global variables are visible to all classes in the same package.

Note

Exercise caution when using global variables. If a global variable is used for a counter in a shared library and multiple threads or processes can access the counter, inaccurate results might occur if the global counter is not protected by a mutex or semaphore.

Creating a Variable

1. Right-click the name of a package or the `Variables` category in the browser.
2. Select **Add New > Variable**.

The new global variable is displayed in the `Variables` category.

Variable Ordering in Rhapsody in C++

Consider the case where your model has variables defined directly in a package. These variables define various constants (such as `PI` and `DEGREES_PER_RADIAN`). Some of these variables are defined in terms of others, such that the dependent variable must be declared before the others for the application to compile. However, Rhapsody will not allow you to override the default alphabetical order of the variable declarations.

There are at least two ways to solve this problem:

- ◆ Define your constants using types. For example, assume a type named `PI` with the following declaration:

```
const double %s = 3.14
```

In this syntax, the `%s` is replaced with the name `PI`.

A `DEGREES_PER_RADIAN` type would have the following declaration:

```
const double %s = 180.0 / PI
```

In this syntax, the `%s` is replaced with the name `DEGREES_PER_RADIAN`.

Because Rhapsody allows you to change the order of the type declarations such that `PI` is generated first, the compilation is successful.

- ◆ Create a variable called `PI` of type `const double` with an initial value of `3.14`. Create a second variable called `DEGREES_PER_RADIAN` of type `const double` with an initial value of `180.0 / PI`. This will not compile because Rhapsody generates the `DEGREES_PER_RADIAN` variable before the `PI` variable.

On the `DEGREES_PER_RADIAN` variable, set the

`CPP_CG::Attribute::VariableInitializationFile` property to `Implementation to`

initialize the variable in the implementation file. The default setting (`Default`) causes the initialization to be put in the specification file if the type declaration begins with `const`; otherwise, it is placed in the implementation file.

Now, your application will compile correctly.

Dependencies

Dependencies are relations in which one class (the dependent) requires something provided by another (the provider). Rhapsody supports the full dependency concept as defined in the UML. Dependencies can exist between any two elements that can be displayed in the browser. Dependencies can be created in diagrams or in the browser.

Constraints

UML *constraints* provide information about model elements concerning requirements, invariants, and so on. Rhapsody captures them in text format. See [Adding Annotations to Diagrams](#) for more information.

Classes

Classes define the attributes and behavior of objects. Classes can also be created in the object model or collaboration diagram editors. See [Classes](#) for detailed information.

Types

The `Types` category displays user-defined, rather than predefined, data types. Rhapsody enables you to create types that are modeled using structural features instead of verbatim, language-specific text.

Receptions

A *reception* specifies the ability of a given class to react to a certain event (called a *signal* in the UML). Receptions are inherited. If you give a trigger to a transition with a reception name that does not exist in the class but that exists in the base class, a new reception is not created.

Events

An *event* is an asynchronous, one-way communication between two objects (such as two classes). Events inherit from the `OMEvent` abstract class, which is defined in the Rhapsody framework. Events can be created in sequence diagrams, collaboration diagrams, statecharts, or the browser.

Actors

An *actor* represents an end user of the system, or an external component that sends information to or receives information from the system. Actors can be created in UCDs, OMDs, collaboration diagrams, and the browser.

Use Cases

A *use case* captures scenarios describing how the system could be used. It usually represents this scenario at a high conceptual level. Use cases can also be created in the use case diagram editor.

Nodes

A *node* represents a computer or other computational resource used in the deployment of your application. For example, a node can represent a type of CPU. Nodes store and execute the run-time components of your application. In the model, a node can belong only to a package, *not* to another node (that is, nodes cannot be nested inside other nodes).

Files

A *file* is a graphical representation of a specification (.h) or implementation (.c) source file. This new model element enables you to use functional modeling and take advantage of the capabilities of Rhapsody (modeling, execution, code generation, and reverse engineering), without radically changing the existing files. For more information, see also [Configuration Files](#).

Adding Diagram Elements

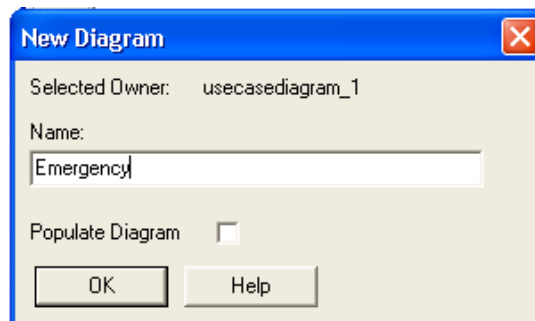
A project has separate categories in the browser for object model, sequence, use case, component, deployment, structure, and collaboration diagrams. Additionally, all of these diagrams (except collaboration diagrams) can be grouped under a package.

Note

Statecharts and activity diagrams are displayed within the class they describe. To add new statecharts and activity diagrams, select the class name in the browser instead of the package, as described below.

To add a new diagram to an existing package, follow these steps:

1. Right-click the package in the browser.
2. Select **Add New Package**.
3. Type the name of the new package in the highlighted area in the browser.
4. Right-click the new package and select the **Add New** submenu.
5. Select the type of diagram you would like to add. The New Diagram dialog box opens, as shown in this example. Here the user is adding a new use case diagram to the package.



6. Type a name for the new diagram in the **Name** field.
7. If you want to populate the new diagram automatically with existing model elements. Click the **Populate Diagram** check box.
8. Click **OK**. If you selected the **Populate Diagram** option, another dialog box displays to allowing you to select which model elements to add to the diagram. Rhapsody automatically lays out the elements in an orderly and easily comprehensible manner.
9. The new diagram appears in the drawing area of the Rhapsody window.

Element Identification and Paths

Packages and classes serve as containers for primary element definitions, in other words, recursive composites or namespaces. Each primary model element is uniquely identified by a path in the following format:

```
<ns1>::<ns2>::...::<nsn>::<name>
```

In this format, <ns> can be either a package or a class. Primary model elements are packages, classes, types, and diagrams. Names of nested elements are displayed in combo boxes in the following format:

```
<name> in <ns1>::<ns2>::...<nsn>
```

You can enter names of nested elements in combo boxes in the following format:

```
<ns1>::<ns2>::<name>
```

Labeling Elements

Element naming conventions often result in complicated and difficult-to-use names for elements, especially in large projects developed by many different individuals. For that reason, Rhapsody enables you to assign a descriptive label to an element—a label that does not have any meaning in terms of generated code, but enables you to easily reference and locate elements in diagrams and dialog boxes. A label can have any value and does not need to be unique, making it possible for you to use non-English labels, reuse labels, and use labels that include spaces on Solaris systems.

Note

Asian language text in Chinese, Korean, Taiwanese, and Japanese is supported in the element labels. The text input can be through the Features dialog box, browser, or graphic editor, and all non-ASCII characters are stored as RTF instead of plain text.

Once you have assigned a label, Rhapsody uses it in place of the element name in diagrams and dialogs by default. Reports display both the element name and the element label.

The `General::Graphics::ShowLabels` property controls whether labels are displayed in diagrams and dialogs.

Adding a Label to an Element

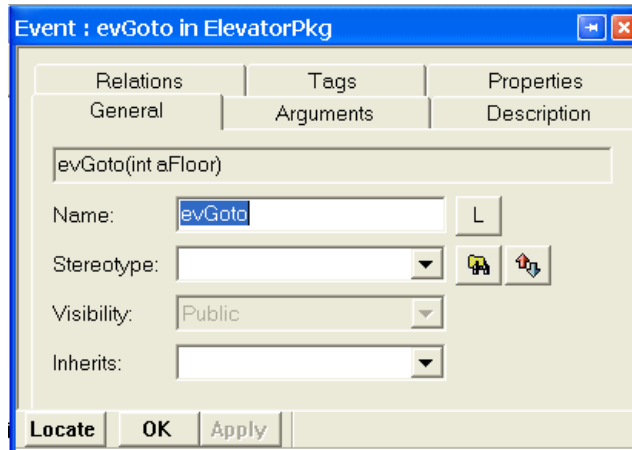
This chart lists all of the element types that can be labeled.

Packages	Association/Aggregation role names
Use cases	Events
Objects	Types
Operations	Transitions
Attributes	Constraints
States	Comments
Classes	Requirements
Actors	Files

To add a label to an element, follow these steps:

1. Right-click the selected element in the browser. (You may double-click to display the Features dialog box immediately.)

2. Select **Features** from the menu to display the Features dialog box for the selected element. In this example, the developer is labeling an event.



3. Click the **L** button next to the **Name** field. The Name and Label dialog box displays.
4. Type the text in the **Label** field. The read-only **Name** field displays the name of the element for reference.
5. Click **OK** to apply your changes and close the Name and Label dialog box.
6. Click **OK** to close the Features dialog box.

Note

Remember that these labels do not have any meaning in the generated code and are used only as references to locate elements in diagrams and dialog boxes.

Removing a Label from an Element

1. Right-click the selected element in the browser. (You may double-click to display the Features dialog box immediately.)
2. Select **Features** from the menu.
3. Click the **L** button next to the **Name** field to open the Name and Label dialog box.
4. Clear the **Label** field.
5. Click **OK** to apply your changes and close the dialog box.
6. Click **OK** to close the Features dialog box.

Alternatively, you can set the label to have the same value as the Name. In this case, the label becomes tied to the name. In this way, any changes made to the element name also affect the label.

Label Mode

Rhapsody differentiates between element names, which are used in the generated code, and element labels, which are just used to represent elements in a user-friendly way within the Rhapsody interface.

Ordinarily, you must provide an element name, even if you are providing a label, and the name must satisfy certain criteria, such as not containing spaces or special characters.

If you would like to deal exclusively with labels, you can set Rhapsody to **Label Mode**. In this work mode, you only have to provide a label for elements. Rhapsody automatically converts the label to a legal element name, replacing any characters that cannot be used in element names. The text that you entered as the label will be shown in the browser, graphic editor, and the Features dialog box.

To switch Rhapsody to Label Mode: From the main menu, select **View > Label Mode**

Note

When Label Mode is selected, Rhapsody ignores the radio button selected under **Display Name** in the Display Options dialog box. Label Mode is a per-user setting. If the project is opened on a different user's computer, it is displayed in normal mode unless that user has selected Label Mode.

Moving Elements

You can move all elements within the browser by dragging-and-dropping them from one location to another, even across packages. You may move a group of highlighted items to a new location in the project by dragging and dropping them into that location in the browser.

In addition, you can drag-and-drop packages, classes, actors, and use cases from the browser into diagrams.

If code has already been generated for a class or package, you can open the code by dragging and dropping the class or package from the browser into an open editor, such as Microsoft Word™, Visual C++, Borland C++, or Notepad. If the code exists, the standard “+” icon appears as you drag the element into the editor. If the “+” icon does not appear, most likely code has not yet been generated for the element. If the element has both a specification and an implementation file, both files are opened.

Note

Do not try to open the code for a class or package by dragging it from an object model diagram because this removes the element from the view.

Copying Elements

You can copy all elements within the browser by dragging-and-dropping. To copy an element, press the Ctrl key while dragging the element onto the new owner.

Copying an element into the same namespace, such as within its own package, creates a copy with the suffix `_copy` appended to its name. For example, copying a class `sensor` into the same package as the original creates a class named `sensor_copy`. Subsequent copies have a suffix of `_copy_<n>`, where `<n>` is an incremental integer starting with 0.

Copying an element into a different namespace creates a copy with exactly the same name.

Renaming Elements

Renaming requires a valid name that does not already exist. An appropriate message is displayed if either condition is not met, and the name reverts to its previous value.

There are two methods for renaming elements. You can edit the **Name** field of the Features dialog box, or you can edit the name directly in the browser.

1. Click once on the element name to select it.
2. Click once again to open the name field for editing. An edit box opens with the name selected, as shown in the following example.



3. Edit the name and press **Enter**, or click once outside the edit field to complete the change.

The other method you may use is the Features dialog box.

1. Open the Features dialog box for the element.
2. Type the new name in the **Name** box.
3. Click **Apply**, or anywhere outside of the Features dialog box, to apply the new name.
4. Click **OK** to close the dialog box.

Note

Do not use this method to rename the project.

Displaying Stereotypes of Model Elements

Rhapsody provides the option of displaying the stereotypes applied to a model element, alongside the name of the element in the browser.

To display elements' stereotypes in the browser, use one of these methods:

- ◆ From the menu, select **View > Browser Display Options > Show Stereotype > All**
- ◆ At the project level, modify the property `Browser::Settings::ShowStereotypes`. The possible values for this property are `No`, `Prefix`, `Suffix`.

Note

When **All** is selected, the property is assigned the value `Prefix`. When **None** is selected, the property is assigned the value `No`. If you want the stereotype name displayed after the element name, this can only be done by changing the property directly to `Suffix`.

If you select the option **First**, then for elements with more than one stereotype, only the first stereotype is displayed in the browser.

The default setting for the property `Browser::Settings::ShowStereotypes` is `Prefix`. For projects created with Rhapsody 6.0 or earlier, this property is overridden and set to `No`.

This settings does not apply to stereotypes that are defined as “new terms.” When a stereotype is defined as a “new term,” it is given its own category in the browser, and any elements to which this stereotype is applied are displayed under the new category.

Creating Graphical Elements

You may create graphical representations for elements in a model and place them in diagrams. This technique is often used to represent all visible behavior in a use case diagram. This feature is often used to represent stereotypes and tags in diagrams.

To create a graphical representation for an element that is listed in the browser, but not shown in the diagram, follow these steps:

1. Locate the element in the browser that is not currently graphically represented in a diagram. Highlight and drag the element into the diagram.
2. The element displays in the diagram, where you can use the Features dialog box to make any changes and additions to the element's description.
3. If they can be connected semantically, you may also link the new graphical element to other diagram elements in the diagram. For example, a stereotype can connect to another stereotype for inheritance.

Moving the element into the diagram functions as a move of a class or object.

Associating an Image File with a Model Element

Rhapsody allows you to associate an image file with a model element. This image can then be used to represent the element in diagrams in place of the standard graphic representation.

To associate an image file with a model element:

1. Right-click the relevant element in the browser, and from the context menu, select **Add Associated Image**.
2. When the file browser dialog is displayed, select the appropriate image file.

Displaying Associated Images

Once an image file has been associated with a model element, it can be displayed in diagrams that contain that element.

To display an element's associated image in a diagram, rather than the regular graphical representation:

1. Open the Display Options dialog box.
2. Select **Enable Image View**.
3. Choose **Use Associated Image** or, alternatively, choose **Select an Image** and click ... to open the file browser.

To enable the display of associated images at the diagram level, set the property `General::Graphics::EnableImageView` to `Checked`. The default value is `Cleared`.

When using an associated image in a diagram, there are a number of ways the image can be displayed:

- ◆ **Image Only**—displays the image instead of the regular graphic representation.
- ◆ **Structured**—displays the image within the regular graphic representation.
- ◆ **Compartment**—displays the image within a compartment together with the other items that are displayed in compartments, for example, attributes and operations.

To select one of these layout options:

1. In the Display Options dialog box, click **Advanced**.
2. When the new dialog box opens, select one of the layouts.
3. Click **OK**.

To set the image view layout at the diagram level, select the appropriate value for the property `General::Graphics::ImageViewLayout`. The default value is `ImageOnly`.

Restoring Image Size, Proportions

When **Image Only** has been selected as the image layout option, the context menu displayed when right-clicking the image contains two additional options under the menu item **Image View**—**Restore Initial Size** and **Restore Initial Proportions**.

Modifying, Replacing, and Removing Associated Image Files

Once an image has been associated with a model element, a menu item called **Associated Image** will be available in the context menu that is displayed when the element is selected in the browser. This item contains the following options:

- ◆ **Open Image**—allows you to view the image in an external viewer. The application that is opened is determined by two properties. If the property `General::Model::ImageEditor` is set to the value `AssociatedApplication`, the default application for this type of file will be launched. However, if this property is set to `ExternalImageEditor`, the application launched will be the application specified with the property `General::Model::ExternalImageEditorCommand`. The value of this property should be the command-line for launching the relevant application.
- ◆ **Replace Image**—displays a file browser, allowing you to select a new image file to associate with the model element, replacing the current associated image.
- ◆ **Remove Association**—breaks the association between the model element and its associated image file.

Supported Image Formats

The following graphic formats are supported for associated image files: jpeg, tiff, bmp, ico, emf, tga, pcx.

Compatibility with Previous Image Association Mechanism

While the procedure described in this section is the recommended way of associating an image file with a model element, Rhapsody still supports the old approach of associating a stereotype with a bitmap file. If you have defined such stereotypes, simply select **Enable Image View** in the Display Options dialog to display the image.

Controlled Files and Image Association

If you add an image file as a controlled file beneath a model element, the image file is automatically associated with the model element.

Deleting Elements

The Edit menu on the diagram editors contains **Remove** and **Delete** options. **Remove** removes an object from a particular view but not from the model, whereas **Delete** deletes the item from the entire model. In the browser, **Delete** always deletes an object from the entire model, including all diagrams.

To delete an object from the entire model while in the browser, use any of the following methods:

- ◆ Select the item. From the **Edit** menu, select **Delete**.
- ◆ Right-click the item, then select **Delete from Model** from the pop-up menu.
- ◆ Select the item and press the Delete key.

If you delete the only item in a category, the entire category disappears with it. For example, if you delete the only object model diagram, the entire Object Model Diagrams category disappears along with the specific diagram you deleted. The category does not return to the browser until you create at least one object model diagram.

Reordering Elements in the Browser

By default, model elements are ordered alphabetically within each category. However, Rhapsody also allows you to manually reorder elements within each category.

You may want to reorder elements in the browser in order to:

- ◆ move important items higher in the list
- ◆ control the order of model elements in the generated code. The order of elements in the generated code is determined by the order of display in the browser. To control the order of elements in the generated code, reorder the relevant elements in the browser before generating the code.

Note

Currently, Rhapsody allows you to rearrange all model elements except for statechart-related elements and activity diagram-related elements.

By default, the ability to reorder elements in the browser is disabled. To enable element reordering, select **View > Browser Display Options > Enable Order** from the menu.

When the reordering option is enabled, you can reorder elements, as follows:

1. Select the element you would like to move.
2. Click the Up or Down arrow in the browser toolbar to move the element.

Note

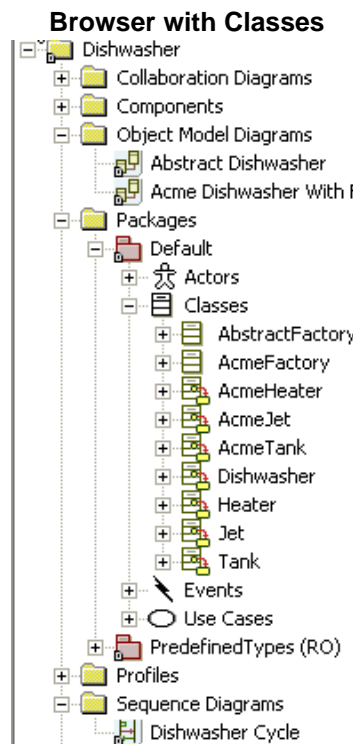
The Up and Down arrows only become enabled after you have selected a movable model element. If the element is at the top of a category, only the Down arrow will be enabled upon selecting the element. If the element is at the bottom of the category, only the Up arrow will be enabled upon selection.

Smart Drag-and-Drop

This section describes how to operate Rhapsody's smart drag-and-drop feature. It provides step-by-step instructions for performing smart drag-and-drop operations.

To drag-and-drop a class, follow these steps:

1. Expand your browser view so that all classes are viewable, as shown in the following figure.

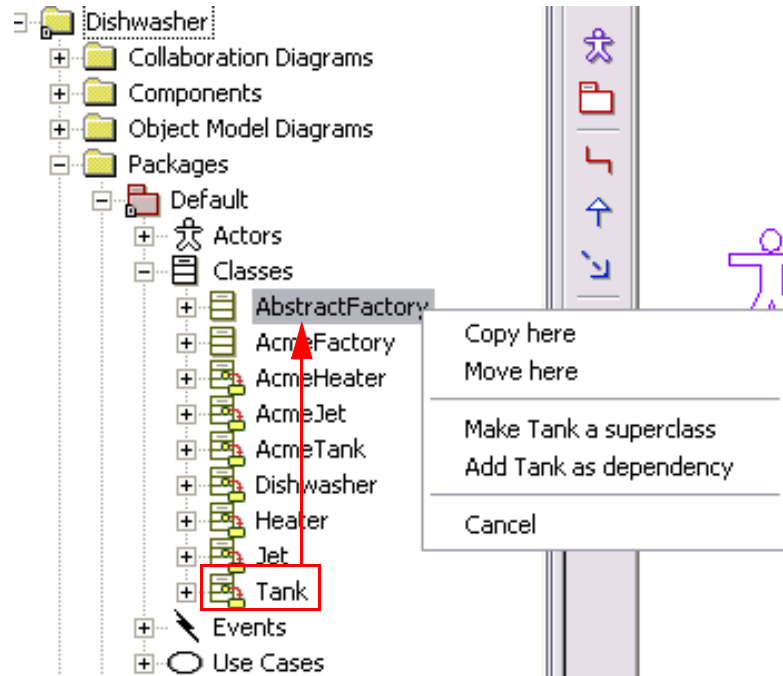


2. Click a class in the browser and while holding the mouse button down, drag the class onto another class; before releasing the mouse button press the Shift key.

Note

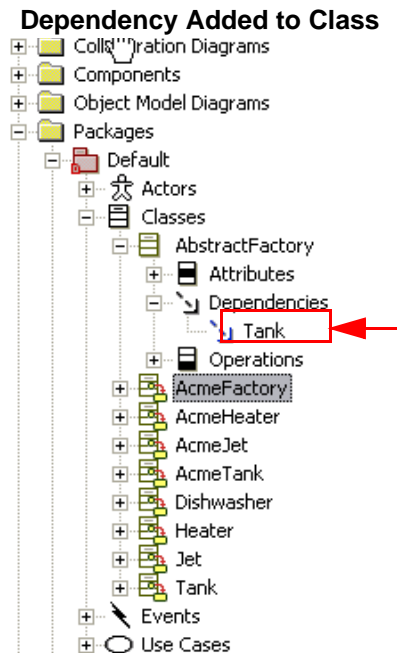
You must press and hold the Shift key before releasing the mouse button when dragging-and-dropping a class.

3. A context submenu appears. Select the desired option. In the figure below the Tank class is being dragged onto the AbstractFactory class.

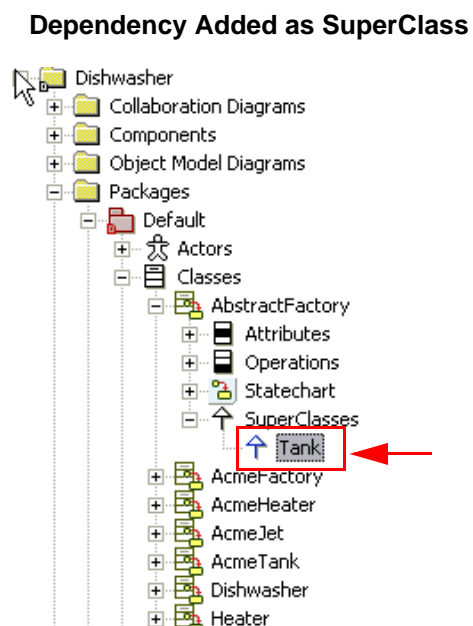


4. In this example the Tank class will be added as a dependency.

5. Once a selection is made from the submenu, the browser opens the feature added (in this case, the dependency), as shown in the following figure.

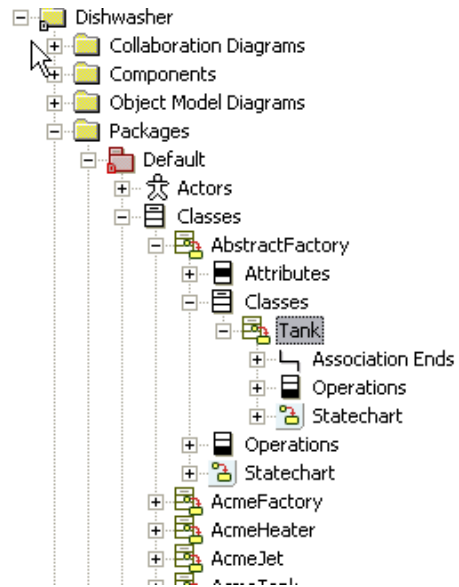


6. If the class selection (Tank) were being made a *superclass*, it would appear as shown in the figure below.



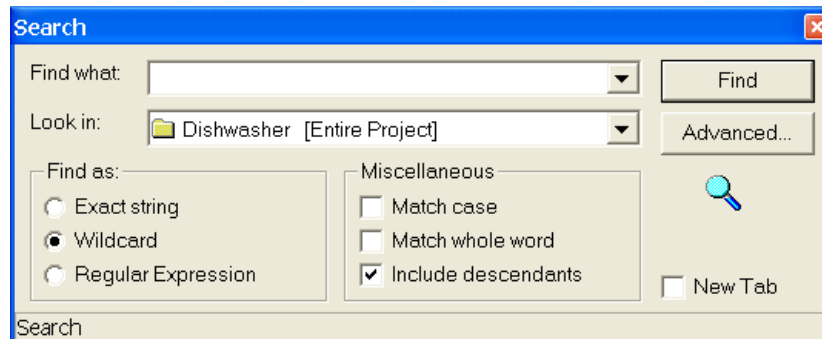
7. If the class selection (Tank) were being copied to another class (Abstract Factory), it would appear as shown in the figure below. Note that the Tank's class attributes remain intact.

Tank Class (Copied) Attributes



Searching in the Model

Engineers and developers can use Rhapsody's Search and Replace facility for simple search operations. To perform quick searches with commonly used filters such as wildcards, case, and exact string, select the **Edit > Search** option to display this dialog box.



To display new search results in a separate tab of the Output window, check the **New Tab** option and enter new search results and click **Find**.

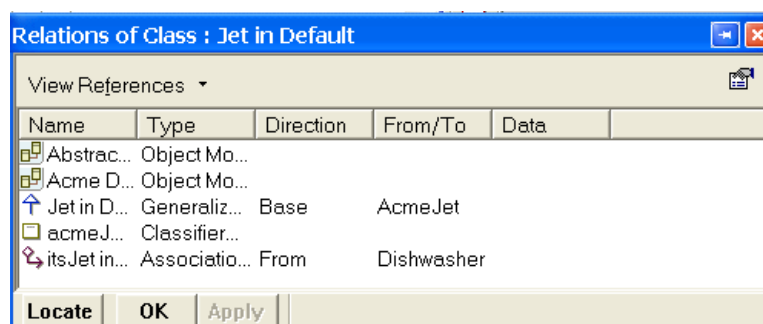
You may also highlight an item in the browser and search only in that part of the project by selecting the **Edit > Search In** option. If you want to perform more complex searches, click **Advanced** in this dialog box.

Finding Element References

The References feature enables you to find out where the specified element is used in the model.

1. Highlight an element in the browser.
2. Select **Edit >References**.

The Relations dialog box opens, listing the locations of the element in the model.



If the element is not used anywhere in the model other than in the browser, Rhapsody displays a message stating that the element is not referenced by any element.

References option returns only model elements that actually *reference* the specified element—it does not return references to the textual name of the element. To do a text-based search for the element name, use the search and replace functionality described in [Searching in the Model](#).

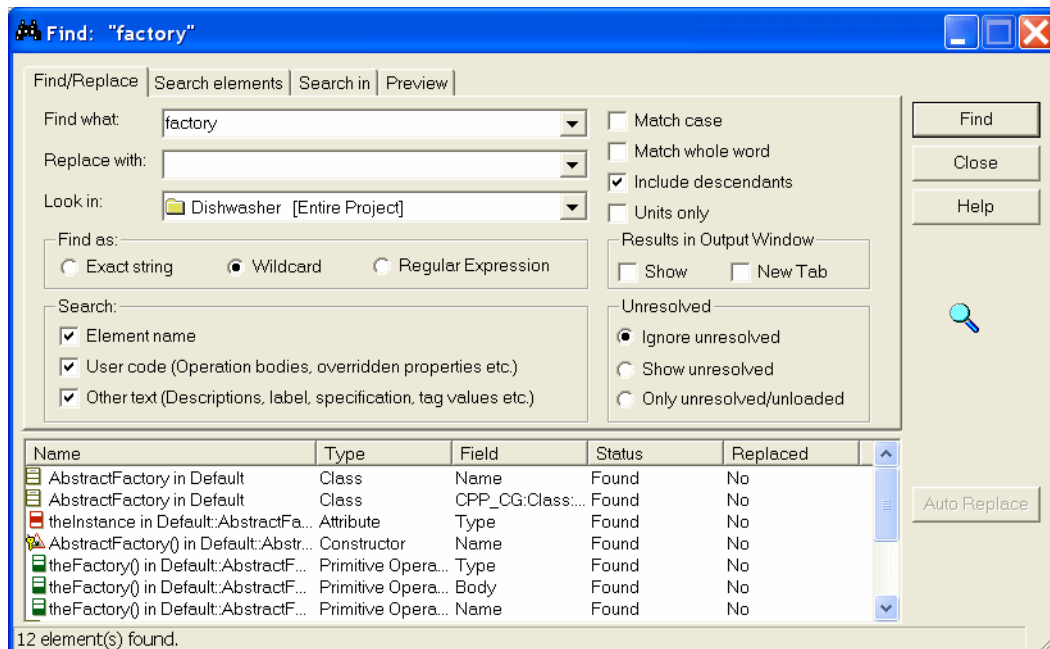
Note

You can also display the list of references by using the keyboard shortcut **CTRL+R**.

Advanced Search and Replace Features

To manage large projects and expedite collaboration work, complex searches are often needed. For those types of searches, select the **Edit > Advanced Search and Replace** option. The advanced facility provides the following additional capabilities:

- ◆ Locate unresolved elements in a model
- ◆ Locate unloaded elements in a model
- ◆ Identify only the units in the model
- ◆ Search for both unresolved elements and unresolved units
- ◆ Perform simple operations on the search results
- ◆ Create a new tab in the Output window to display another set of search results



All search results display in the Output Window with the other tabbed information. Search and replace results display at the bottom of the Find dialog box, as shown in the example. For more information about the uses of this facility, see [Advanced Search and Replace Features](#).

Using the Auto Replace Feature

To perform a search and replace operation with an automatic replace, follow these steps:

1. Select the **Edit > Advanced Search and Replace** option
2. Enter the **Find what** and **Replace with** text in those two fields.
3. Click **Auto Replace**. The changes display at the bottom of the window.

This automatic replacement also performs an automatic commit of the changes into the Rhapsody model so that you do not need to apply the changes manually. However, it does not display any ripple effects of the change.

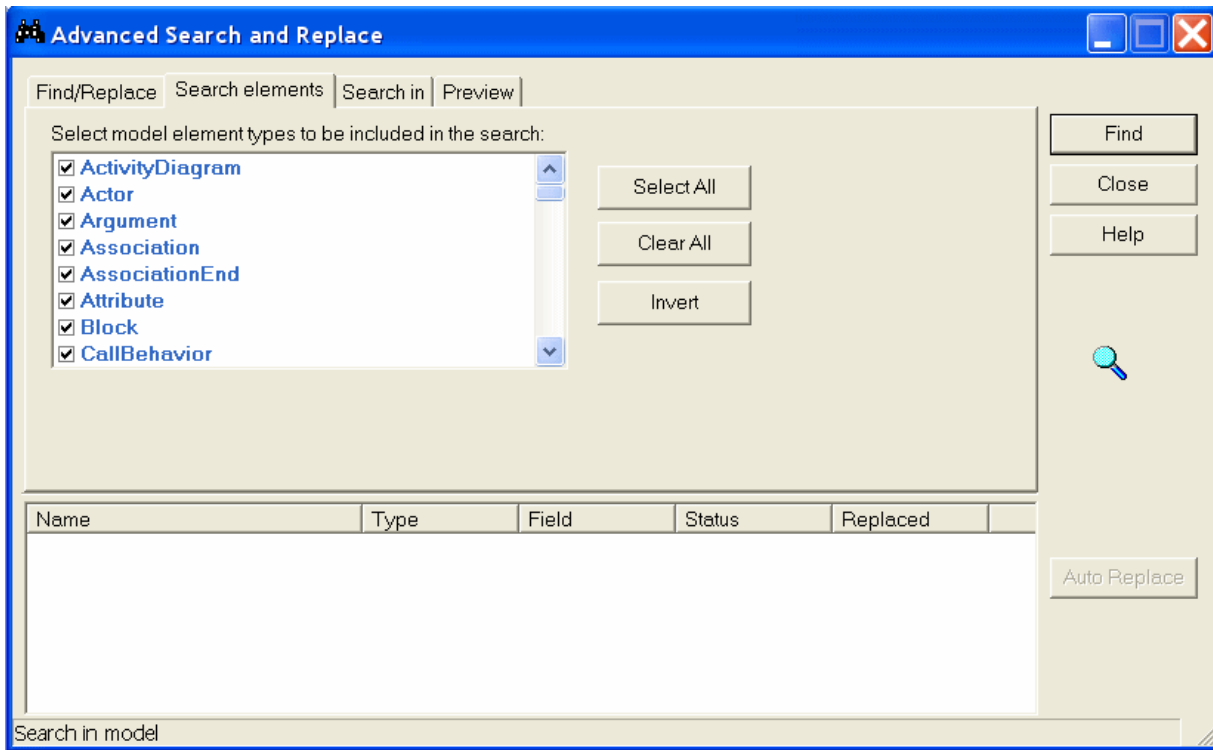
The Auto Replace feature cannot be used on the following items because they do not qualify to be replaced automatically:

- ◆ Read-only elements.
- ◆ A description that includes hyperlinks
- ◆ Stub or unresolved element names
- ◆ The name of an event reception, constructor, destructor, generalization, or hyperlink

Searching for Elements

To search for specific elements in the model, follow these steps:

1. Select the **Edit > Advanced Search and Replace** option to display the Advanced Search and Replace dialog box.
2. Click the **Search elements** tab to display this dialog box.



3. Select any or all of the element types listed in the scrolling area. Use the **Select All**, **Clear All**, or **Invert** to select element types quickly.
4. Click **Find**. The results display at the bottom of the window.
5. Highlight an item in the search results and right-click to perform either of these operations:
 - ◆ **References** displays all of the relations to the selected element.
 - ◆ **Delete from model**.
6. If you double-click an element in the list, the Features dialog box displays to allow you to see and change the element's description and other characteristics.

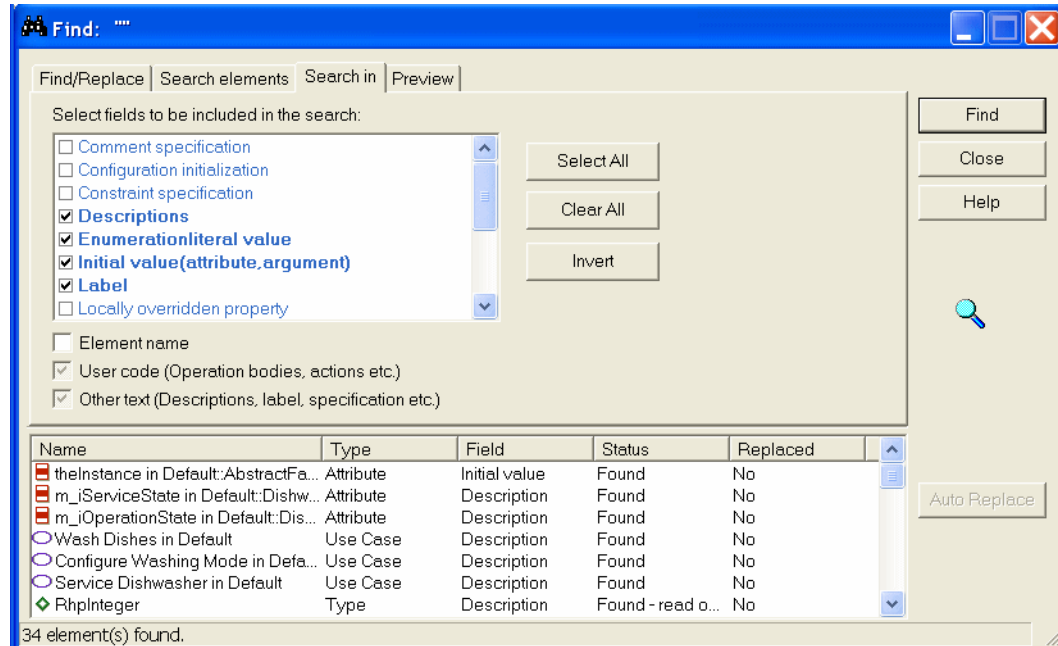
7. You may also highlight an item and perform an **Auto Replace**, as described in [Using the Auto Replace Feature](#).

Searching in Field Types

To search in specific fields within the model, follow these steps:

1. Select the **Edit > Advanced Search and Replace** option to display the Advanced Search and Replace dialog box.
2. Click the **Search in** tab.
3. Select the types of fields you want to find in the model by checking them in the list at the top of the dialog box. The possible selections are as follows:
4. The **Element name**, **User code**, and **Other text** check boxes display the fields associated with them. For example, if only **User code** is checked, only the code-related fields (Actions, Configuration initialization, Initial value, Operation bodies, and Type declarations) are checked. Similarly, if only **Other text** is checked, only the text-related fields are checked. The possible values are as follows:
 - ◆ **Comment specification**—Scans all the comment specifications for the specified string
 - ◆ **Configuration initialization**—Scan all the configuration `init` code segments for the specified string
 - ◆ **Constraint specification**—Scans all the constraint specifications for the specified string (Chinese, Korean, Taiwanese, and Japanese are supported in this field and stored as RTF.)
 - ◆ **Descriptions**—Scans all the descriptions for the specified string
 - ◆ **Enumeration literal value**—Scans the literal value of the enumeration type for the specified search string.
 - ◆ **Initial value**—Scans the initial value fields for attributes and arguments for the specified search string
 - ◆ **Label**—Scans all the labels for the specified search string
 - ◆ **Locally overridden property**—Scans the locally overridden properties for the specified search string.
 - ◆ **Multiplicity**—Scans the Multiplicity field of associations for the specified search string.
 - ◆ **Name**—Scans all the element names for the specified search string
 - ◆ **Operation bodies**—Scans operation bodies, including operations defined under classes, actors, use cases, and packages for the specified search string

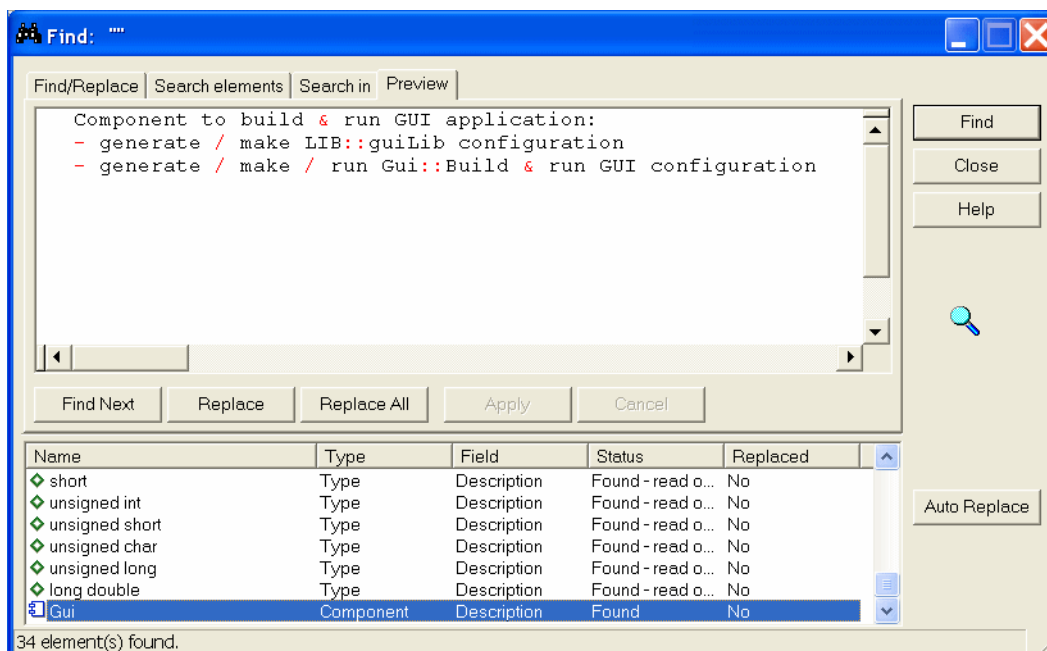
- ◆ **Requirement specifications**—Scans all requirement specifications for the specified search string. (Chinese, Korean, Taiwanese, and Japanese are supported in this field and stored as RTF.)
 - ◆ **Tag Value**—Scans the value of the tags for the specified search string.
 - ◆ **Transition Label**—Scans all transition code (action, guard, trigger) for the specified search string.
 - ◆ **Type declarations**—Scan all the user text in “on-the-fly” types for the specified search string.
5. You may also specify that the **Element name**, **User code**, and **Other text** be included or excluded from the search.
 6. Click **Find** to list the fields that meet your search criteria at the bottom of the dialog box, as shown below.



You may highlight an item in the list and replace it.

Previewing in the Search and Replace Facility

After performing a search, you may highlight an item in the search results and click the **Preview** tab. This displays the occurrence of the highlighted item, shown below, so that you can see and perhaps change the string.



Use the following buttons to make changes to the selected occurrence:

- ◆ **Find Next**—Highlights the next occurrence of the string in the same file.
- ◆ **Replace**—Replaces the highlighted occurrence of the original string with the replacement string specified on the **Find/Replace** tab.
- ◆ **Replace All**—Replaces all occurrences of the string.
- ◆ **Apply**—Applies the changes done in the **Preview** tab (editing or replacing text) to the Rhapsody model.
- ◆ **Cancel**—Cancels the search and replace.
- ◆ **Auto Replace**—Automatically replaces all occurrences of the highlighted string with a replacement string.

Note

If you make a mistake, select **Edit > Undo** to undo the changes.

Controlled Files

Rhapsody allows you to create *controlled files* and then use their features.

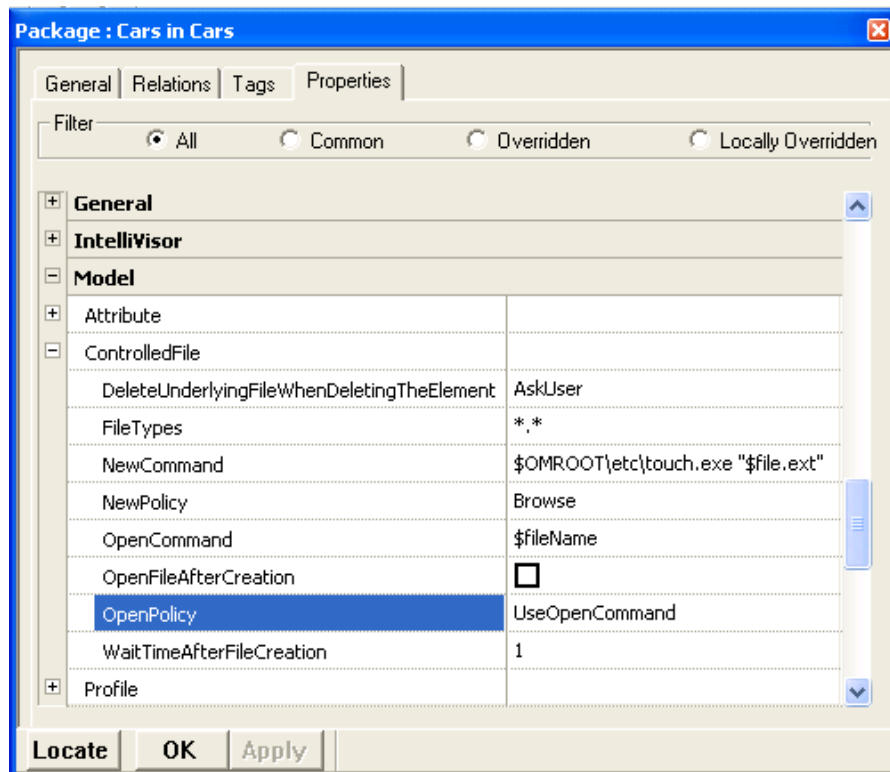
The model in this section uses the “Cars” C++ sample (available in the C:\(RhapsodyInstallationDirectory)\Samples\CppSamples\Cars) to demonstrate how to use controlled files.

- ◆ Controlled Files, such as project specifications files (e.g. word, excel files) are typically added to a project for reference purposes and can be controlled through Rhapsody.
- ◆ A controlled file can be a file of any type (.doc, .txt, .xls, etc.).
- ◆ Controlled files are added into the project from the Rhapsody browser.
- ◆ Controlled files can be added to diagrams via drag-and-drop from the browser.
- ◆ Currently, only Tag and Dependency features can be added to a controlled file.
- ◆ By default all controlled files are opened by their Windows-default programs (for example, Microsoft Excel for .xls files).
- ◆ The program(s) associated with controlled files can be changed via the **Properties** tab in the controlled files dialog box.

Creating a Controlled File

Controlled files can be created using many methods. A controlled file can be created by browsing and selecting a file from the file system or by using a command. To select a method, follow these steps:

1. To select the way a controlled file is created, double-click a package in the Rhapsody browser and select the **Properties** tab from the Features dialog box.
2. With the **All** filter selected, expand the section **Model** and expand the `ControlledFile` subsection, as shown below. The property `NewPolicy` controls the way a controlled file is created. The default selection is `SystemDefault`. The property can be set to `UseOpenCommand` or `UseRhapsodyCodeEditor`.



- ◆ **SystemDefault:** This setting lets the program (as done in hyperlink etc.), open the file according to the MIME-type mapping.
- ◆ **UseOpenCommand:** This setting means that double-clicking on the controlled file invokes the open command. The open command is set in the `OpenCommand` property (Model/ControlledFile/OpenCommand). This is a string property that expands the keyword `$fileName` to the full path of the Controlled File. In the event of property inconsistency, where the program is set to invoke the

UseOpenCommand but the command contains no setting, the SystemDefault policy will be used.

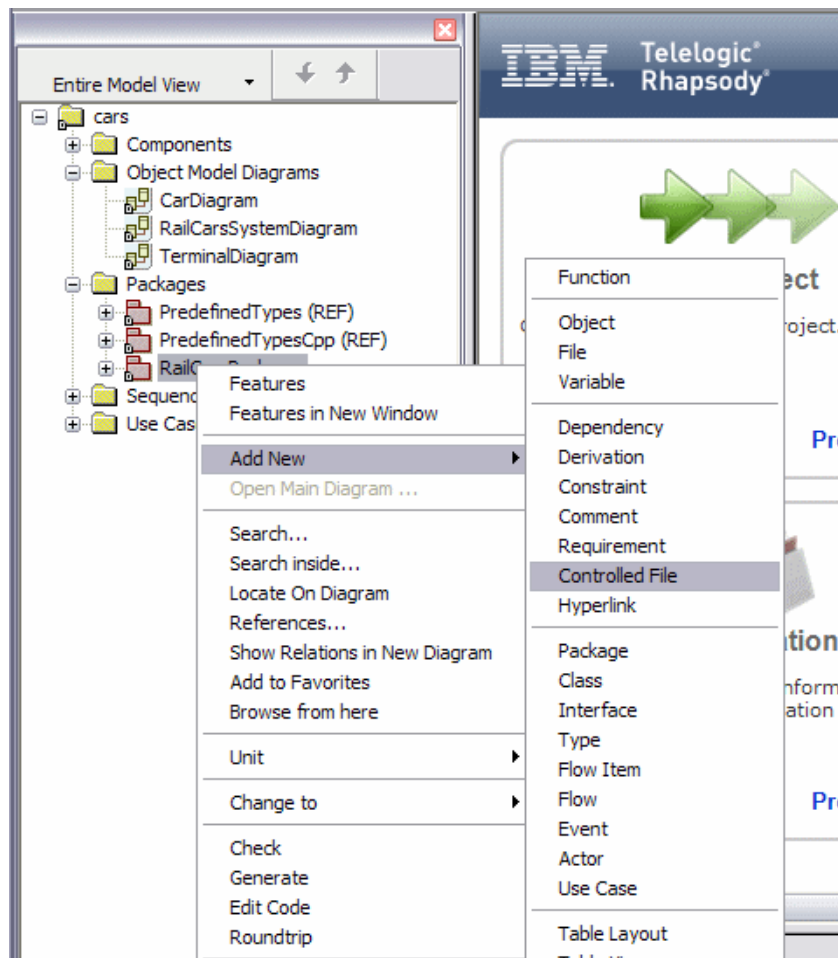
- ◆ **UseRhapsodyCodeEditor:** This setting allows the file to be opened within the internal Rhapsody code editor.

If you want to open the controlled file after it is created, set the value of the property `OpenFileAfterCreation` to yes. The file will open the file immediately after it is created.

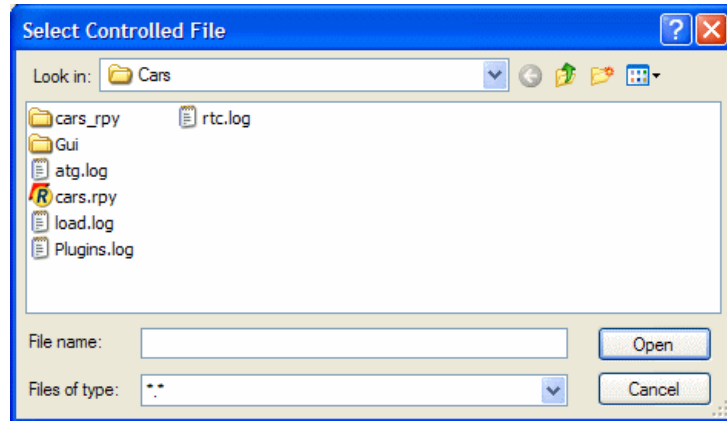
Browsing to a Controlled File

This section describes how to create a controlled file by browsing to it.

1. To create a controlled file, right-click a non-read-only element, such as a package, in the browser and select **Add New > Controlled File**, as shown in the following figure.

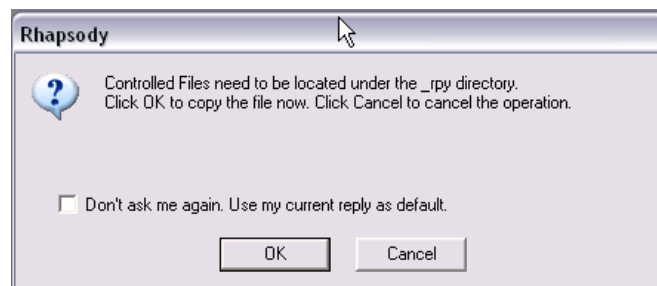


2. The Select Controlled File open window opens. Navigate to the desired file and click the **Open** button.

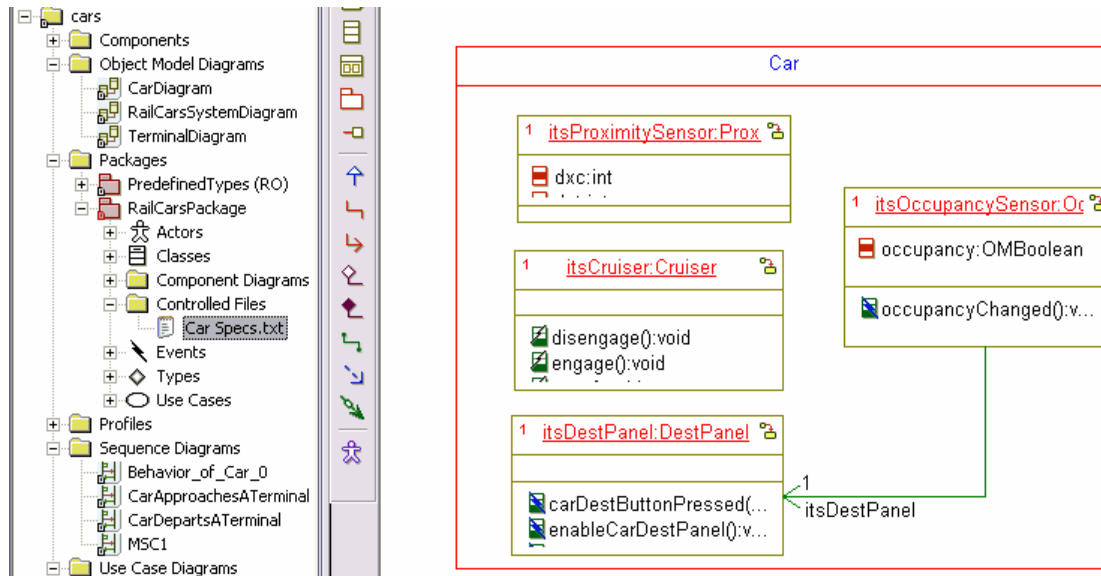


Note: Unless the controlled file being selected is located in the right location under the `_rpy` directory (`cars_rpy` in the example), a message appears asking you for permission to copy the file. If you do not want the file copied into the `_rpy` directory, you must place the original into the `_rpy` directory before selecting it from the Rhapsody browser.

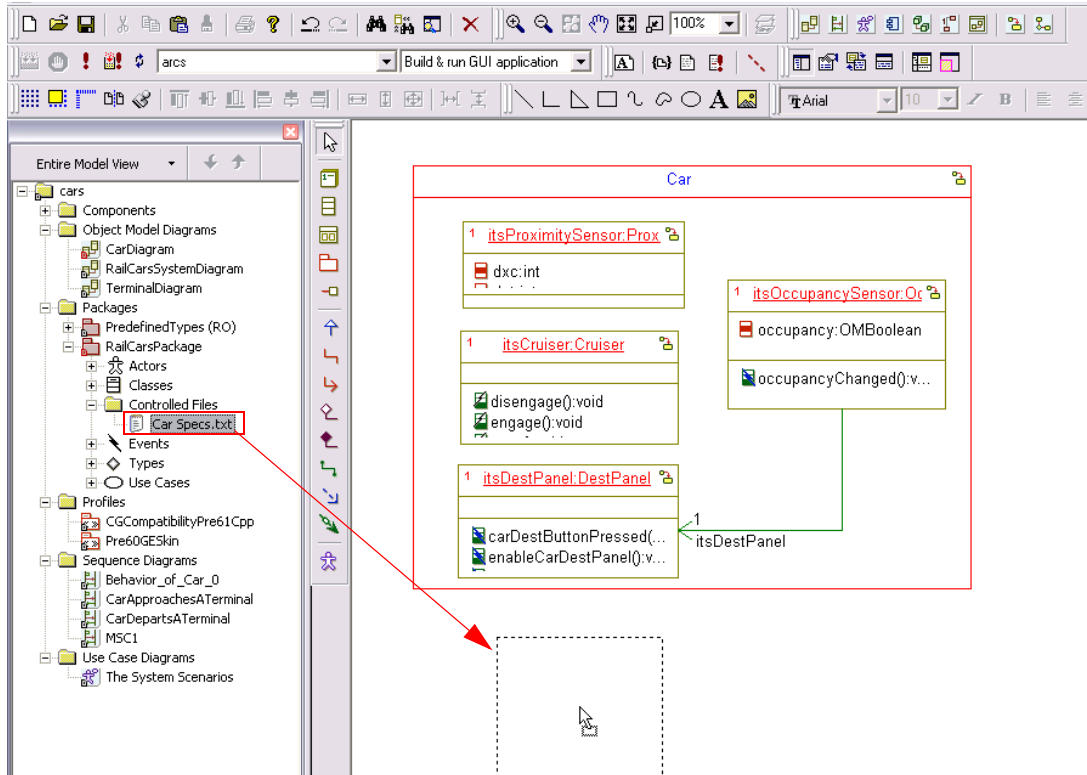
3. A dialog box appears asking you for permission to copy the file into the `_rpy` directory, as shown in the following figure. Click **OK**.



- The controlled file now appears in the Rhapsody browser, as shown in the following figure.



- Controlled files can be added to the graphic interface (right window pane) via drag-and-drop from the browser, as shown in the following figure.



Controlled File Features


Controlled files can be any standard *Windows* type, such as .doc, .txt, and .xls. When those files are opened in Rhapsody, they are displayed in their Windows-default programs, such as for Microsoft Excel for .xls files. In addition, the program(s) associated with controlled files can be changed via the **Properties** tab in the controlled files dialog box.

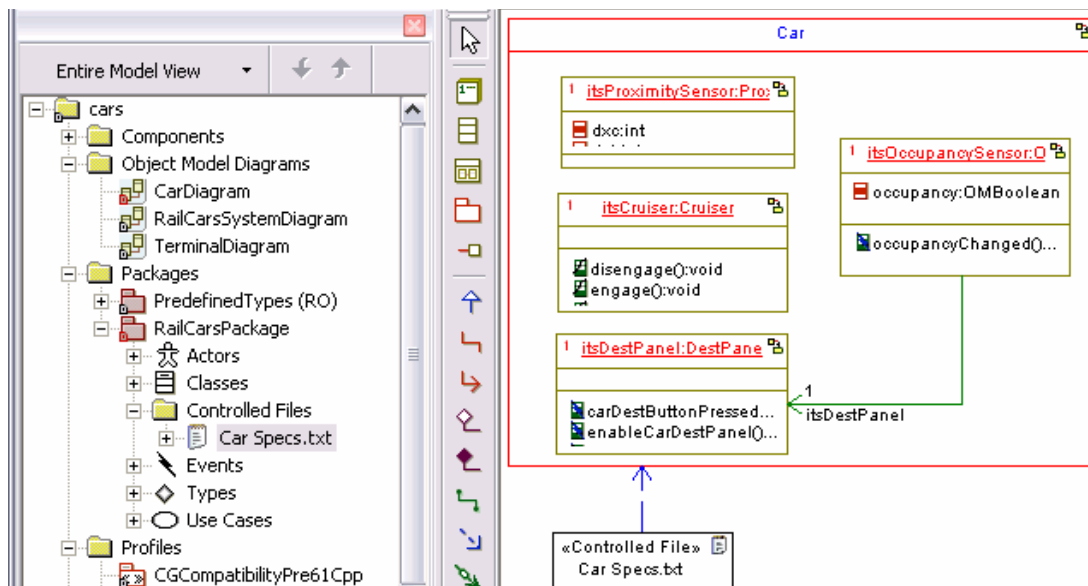
Dependencies

Dependencies can be added to controlled files to display what object depends on the controlled file or what object the controlled file is dependent upon. To add a dependency to a controlled file (via Browser), follow these steps:

1. To add a new dependency, right-click the controlled file and select **Add New >Dependency**.
2. Make a selection in the drop-down window that appears and click **OK**.
3. Type the name of the dependency, or leave the default name.
4. Double-click the dependency in the browser to modify it.

To add a dependency to a controlled file (via Graphic Pane):

1. Drag the controlled file from the browser into the graphic pane, if you have not already done so.
2. Using the dependency icon , draw a dependency line from the controlled file to the desired object, as shown in the following figure.



Tags

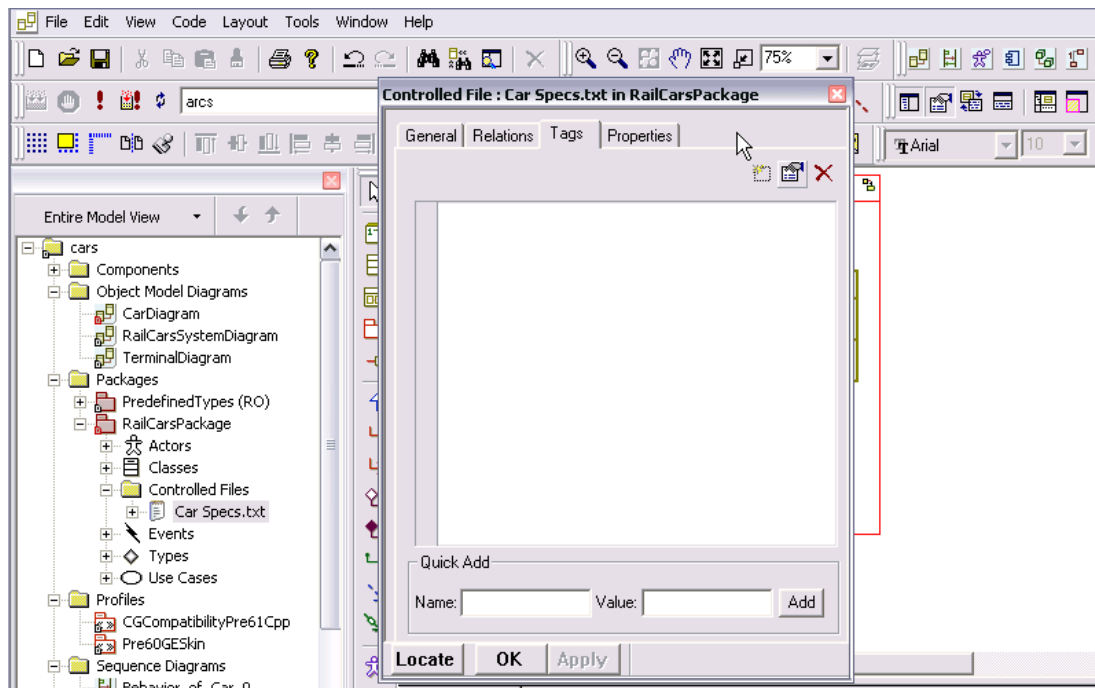
Tags can be added to controlled files for purposes such as identification. For instance, tagging all controlled files that are related to the products's specification helps differentiate those controlled files from other controlled files that may be embedded in the project.

To add a tag to a controlled file (via Browser):

1. Right-click the controlled file and select **Add New >Tag**.
2. Type a name for the tag or leave the default name.
3. Double-click the tag in the browser to modify it.

To add a tag to a controlled file (via the Features dialog box):

1. Right-click the controlled file(s) in the browser and select **Features** from the submenu that appears.
2. The Features dialog box appears. Click the **Tags** tab and enter the name of the tag in the “Quick Add” section at the bottom of the window, as shown in the following figure. Click the **Add** button.



3. Click **OK**.
4. Expand the Controlled File by clicking on the “+” next to it in the Browser.

5. The Tags listing appears. Click the “+” next to the Tags listing to see the new Tag you just created.

Configuration Management

Configuration management options are as follows:

Configuration Management Feature	Feature Description
Add to Archive	This feature is used when a user wishes to add a new file into the configuration management server for the first time. After that, other users can check in/out, lock/unlock the file.
Check In	This feature checks the file into the configuration management server so that other users can modify the file. You still have read-only access to the file, but you cannot modify it.
Check Out	This feature checks the file out of the configuration management server so that other users cannot modify the file. Other users can open the file for read-only purposes.
Lock	This feature locks the file so that other users cannot modify it in the future. Only you can unlock the file.
Unlock	This feature unlocks the files so that other users can modify the file in the future. Only you can unlock the file.

Troubleshooting Controlled Files

Below is a list of the most commonly encountered problems with their solutions:

- ◆ **Question:** “I checked out a controlled file. But I am not able to change its description, properties, etc.”

Answer: The metadata for the controlled file is stored in its parent element. So you must check out the parent element in order to change any associated information, such as the description, properties, etc.
For instance, in the example provided in [Creating a Controlled File](#), right-clicking on the parent element RailCarsPackage and choosing Configuration Management ▶ Check out, allows the controlled file attributes to be modified.

- ◆ **Question:** “I know controlled files are copied to a directory under _rpy during the creation of the controlled file, but where exactly is the controlled file copied?”

Answer: Controlled Files are copied to the same location where its parent is located on the disk. For instance, suppose you have a package called “package_1” and the repository file for it is stored like this: MyProject_rpy\package_1\package_1.sbs.
Now any controlled file you add to the package “package_1” is going to be stored in the folder: MyProject_rpy\package_1\

Answer: You have the ability to select different files from an opened Features dialog box. So when you type a different name, the system “thinks” that you are actually trying to point to a different file and returns an error message.

You can rename a controlled file from the browser by clicking two separate times on it (double-clicking on the file will open it).

- ◆ **Question:** “Is it possible to add a Controlled File to the model without copying the file to the _rpy folder?”

Answer: No, but you can add a hyperlink to the desired file instead. However, this will result in the loss of all CM (Configuration Management) capability which includes the following operations:
 - Add to archive
 - Check in
 - Check out
 - Lock
 - Unlock

- ◆ **Issue:** Configuration management (CM) operations on Controlled files fail on some occasions.
- ◆ **Details:** If you are using CM in command mode, make sure the file is located under the `_rpy` folder. In order to perform Configuration Management operations in command mode from Rhapsody, the file (other than the project itself) has to be under the `_rpy` directory. There is no current fix for this issue.

Workaround: Create a new package under the project. By default, a package is stored in separate file: do not modify that setting. Now add the problematic controlled file(s) to this package. Now you will be able to perform CM operations.

- ◆ **Issue:** Adding an existing element with a Controlled File to the Model and choosing a different name, instead of replacing the file.
- ◆ **Details:** If you perform an Add To Model of an already existing element, Rhapsody gives you the option of replacing the element or adding it using a different name. If you choose to use a different name, Rhapsody adds the element using the new name.

If both the element that already existed in the model and the element that was just added have controlled files by the same name, more than one controlled file of the same name exists in the model. But there may be only one instance of the file in the directory, which means the two controlled files in the model are actually pointing to the same file.

Controlled Files Limitations

General Limitations

- ◆ **Connecting to Archive Error:**

If your configuration management tool is ClearCase, this Connecting to Archive limitation applies to you.

If you are trying to “Connect to Archive” using the Configuration Items tool (File/Configuration Items), and your Rhapsody project contains a controlled file, the following error results:

Cannot perform Connect to Archive.

Reason: Rhapsody could not add “Project_rpy” to Source Control as the project contains at least one Controlled File. To proceed, add this directory (and its subdirectories) to Source Control using ClearCase. Note that, you do not need to Connect to Archive again.

Use ClearCase to add the directory to the appropriate source control. You now do not have to perform a “Connect to Archive.”

- ◆ **Deleting an Element with a Controlled File(s) Deletes the Controlled File From the Model Only.**

If you delete a controlled file from a Rhapsody Model, you are given the option to either delete the file completely from the hard disk drive or leave the file intact and simply delete the controlled file from the model. If you delete any other element that includes a controlled file (as a child), the controlled file is left intact on the hard disk; the option to delete the controlled file completely from the hard disk drive is not offered.

- ◆ **Moving a Controlled File - Browser vs. Graphic Editor**

If you have two packages stored in different directories (P1 and P2 for this example), and you add a controlled file to the P1 directory, the physical controlled file on the hard disk is stored in P1's directory (the same directory where P1.sbs file is located).

From the Rhapsody browser, drag the controlled file from P1 and drop it in the P2 package. Now examine the contents of the P1 and P2 directories on the hard disk. Notice that the file has moved from the P1 directory to the P2 directory.

If you repeat the above exercise in the Rhapsody Graphic Editor, the physical file is *copied* from the old directory to the new directory, instead of being completely *moved* from the old directory to the new directory.

- ◆ **Configuration Management Synchronize Dialog Box Does Not Show Controlled Files**

Configuration Management's Synchronize dialog box shows items that are modified by other users but not yet loaded into your model. This dialog does not hold true for controlled files. The synchronize dialog box does not display Controlled Files when there are newly-updated controlled files in the Configuration Management system.

- ◆ **Controlled Files Version Number Not Shown In the Configuration Management Window**

The Configuration Management window shows the version number for the items in its listing (only in command mode). But for a controlled file, it does not show the version number.

- ◆ **Foreign Language Controlled File Names are Not Supported**

If your file name has non-English characters, you cannot add it to your project as a controlled file.

- ◆ **CM Archive is Not Affected When Controlled Files are Moved/Renamed/Deleted**

When you move, rename or delete a Controlled File, its respective Configuration Management item is not updated accordingly.

DiffMerge Limitations

- ◆ **Diffmerge Does Not Compare the Contents of Two Controlled Files**

Diffmerge does not identify/indicate any content differences between two controlled files. Diffmerge does not examine the contents of controlled files as they exist on the hard disk drive. However, Diffmerge can show differences between the metadata information of two controlled files in the model.

Ex. Given two packages (Pkg1 and Pkg2) that each contain the controlled file readme.txt, if the file contents of the Pkg1 readme.txt are different from the contents of the Pkg2 readme.txt then a Diffmerge performed between Pkg1 and Pkg2 will not identify that the readme.txt files are different. However, if the Pkg1 readme.txt and Pkg2 readme.txt have different metadata such as properties or description then those differences are shown in the Diffmerge.

- ◆ **'Save merge as...' or 'merge back to rhapsody' Does Not Save Any Controlled Files**

When using Diffmerge to compare two model units that contain controlled files, performing the “Save merge as...” or “merge back to rhapsody” functions does not save the controlled files on the hard disk drive.

Ex. Using the same example as above, if you compare Pkg1 with Pkg2 and save the merge as “Pkg_Merged,” the physical file readme.txt for Pkg_Merged will not be saved on the physical hard disk drive. If you were to navigate to the file under the appropriate directory (c:\diffmerge for example), you would notice the readme.txt file does not appear in that directory.

Search & Replace Limitations

Note

After performing a search using regular expressions, look inside the **Preview** tab in the Search and Replace dialog box. The **Find Next**, **Replace**, and **Replace All** functionalities may not work for one or more selected search results.

This search and replace limitation is not limited to controlled files.

Printing Rhapsody Diagrams

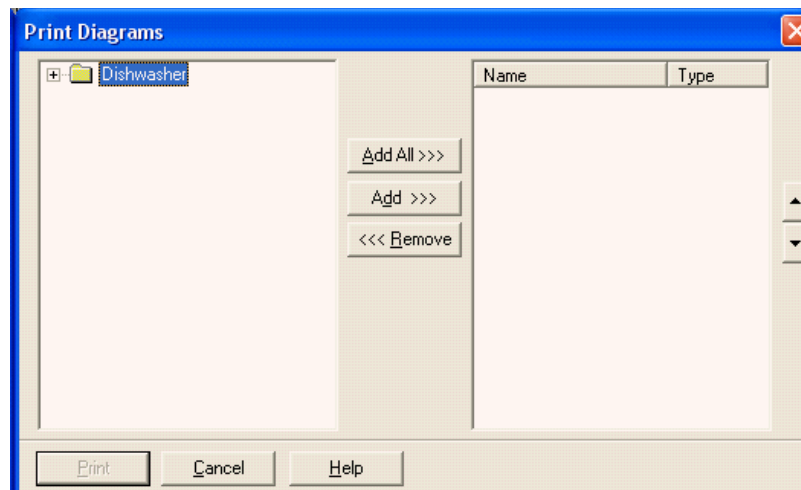
The File menu includes the following print options:

- ◆ **Print Diagrams**—Displays all the diagrams used in the model so you can easily select which ones to print (see [Selecting Which Diagrams to Print](#))
- ◆ **Print**—Displays the standard Print dialog box, which enables you to print the current diagram on the specified printer
- ◆ **Print Preview**—Displays a preview of how the diagram will look when it's printed
- ◆ **Printer Setup**—Displays the standard Print Setup dialog box, which enables you to change the printer settings, such as landscape instead of portrait mode
- ◆ **Diagram Print Settings**—Enables you to specify diagram-specific print options (see [Diagram Print Settings](#))

Selecting Which Diagrams to Print

Rhapsody enables you to print multiple diagrams.

1. Choose **File > Print Diagrams**. The Print Diagrams dialog box opens, listing the diagrams and statecharts in the current project.



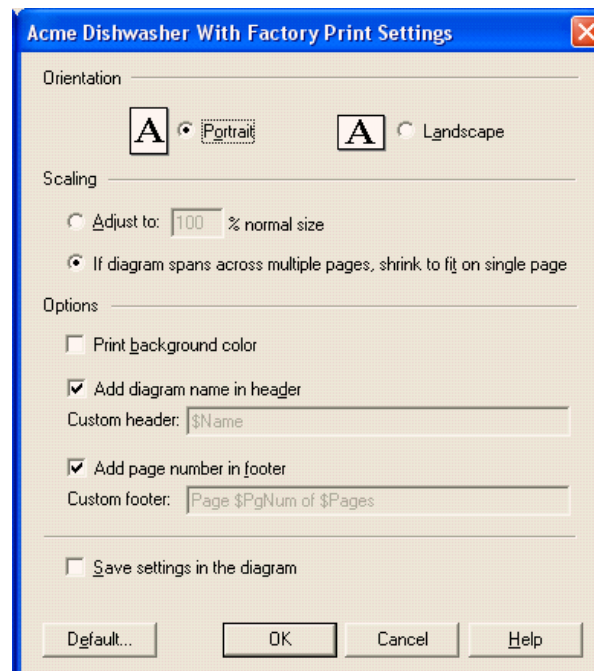
The dialog box has the following features:

- ◆ The list box on the left shows all of the diagrams in the project.
- ◆ The list box on the right shows the names and types (for example, ObjectModelDiagram) of the diagrams selected for printing.

- ◆ The buttons in the middle let you select and deselect diagrams for printing.
 - ◆ The Up/Down arrows on the far right control the order in which the diagrams are printed.
 - ◆ The buttons along the bottom edge of the dialog access the Print dialog box, or cancel printing.
2. To expand the list of diagrams in the project, click the + sign to the left of one of the diagram type names. The tree expands to show all the diagrams of that type. Note that statecharts are listed first by package, then by class.
 3. Do one of the following:
 - a. To add a diagram to the print list, select it in the list on the left and click **Add**.
 - b. To add all the diagrams in the project to the print list, click **Add All**.
 - c. To remove a diagram from the print list, select it in the list on the right and click **Remove**. Rhapsody prints the diagrams listed in the right from top to bottom.
 - d. To change the print order, select a diagram from the print list and use the up or down arrows.
 4. Click the appropriate button:
 - a. **Print**—Dismisses the Print Diagrams dialog box and opens the standard Print dialog box for your operating system. In this dialog box, you set attributes such as page size (for example, 11" x 17"), the printer to use, whether to print to a file, whether to use grayscale, and initiate the printing operation.
 - b. **Cancel**—Cancels the print operation and dismisses the dialog box.

Diagram Print Settings

The **File > Diagram Print Settings** option enables you to specify diagram-specific print settings that can be retained between Rhapsody sessions. This option is particularly useful for complex and multipage diagrams. The following figure shows the Print Settings dialog box.



The dialog box contains the following fields:

- ◆ **Orientation**—Specifies the page orientation
- ◆ **Scaling**—Specifies the scale percentage, or whether to shrink the diagram so it fits on one page.
- ◆ **Options**—Specifies whether to:
 - Print the background color.
 - Include a header or footer on the printout.
 - Retain your diagram settings across Rhapsody sessions.

By default, the header includes the name of the diagram, and the footer contains the page number. To specify a new header or footer, uncheck the appropriate check box and type the new value in the text box.

For example, to include the name of your company in the diagram header, follow these steps:

- ◆ Uncheck the **Add diagram name in header** check box.

- ◆ In the text box, type the name of your company.

Click **OK** to save your settings. Use one of the Print options to print out the diagram.

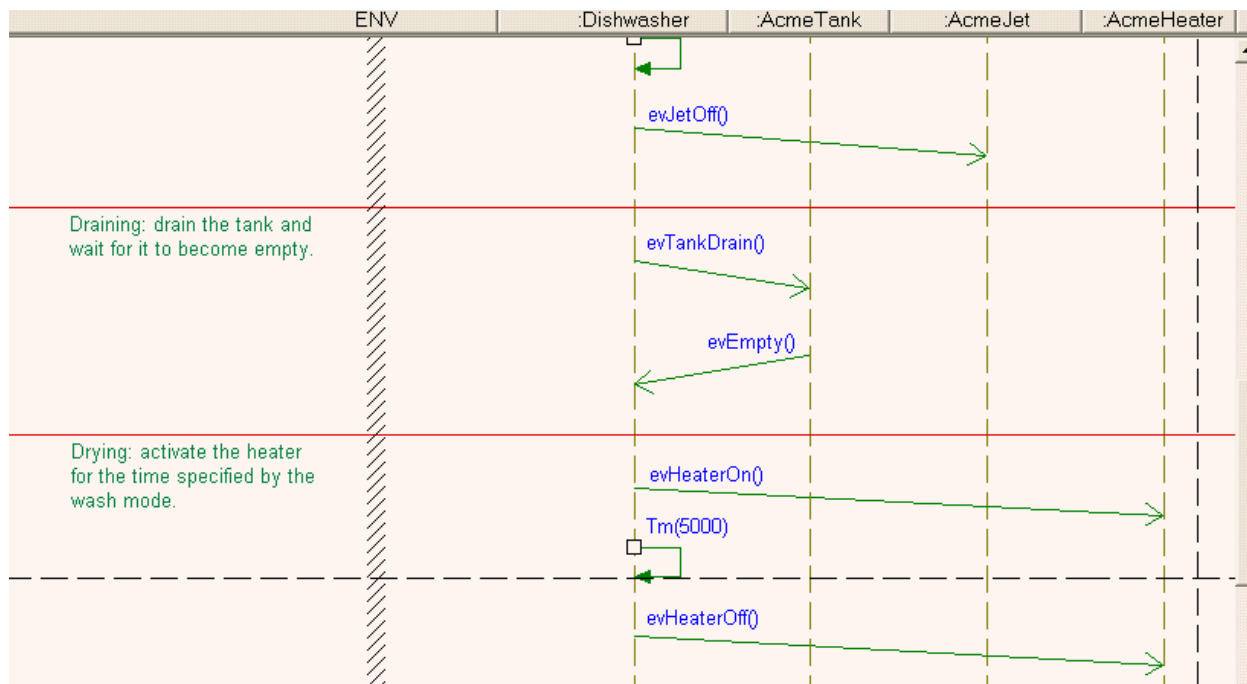
Using Page Breaks

Rhapsody can break a large diagram across several pages or scale it down so it fits on a single page, depending on your preference.

To scale the diagram so it fits on a single page, right-click in the diagram and select **Printing > Fit on One Page** from the pop-up menu. To break the diagram across pages, simply disable this option.

To view the page breaks in a diagram, do one of the following:

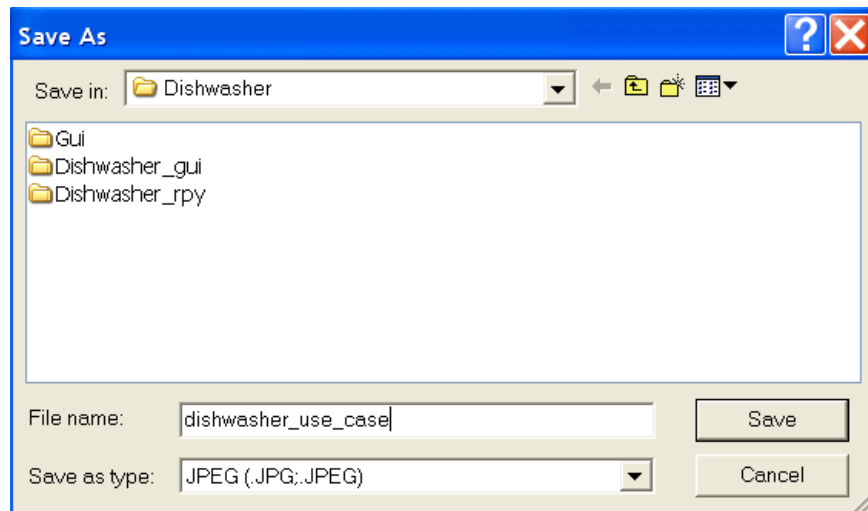
- ◆ Click the **Page Breaks** tool to toggle the display of page breaks in your diagram.
- ◆ Select **Layout > View Page Breaks**.
- ◆ Right-click in the diagram and select **Printing > View Page Breaks**. The dashed lines that represent the page boundaries are displayed, as shown in the following figure.



Exporting Rhapsody Diagrams

To export any Rhapsody diagram, follow these steps:

1. Display the diagram.
2. Right-click inside the diagram, but not on a specific item in the diagram.
3. From the displayed menu, select **Export Diagram Image**.
4. In the Save As dialog box, select a directory for the saved image in the **Save in** directory selection field.



5. In the **File name** field, type the name you want to use for the exported image file.
6. In the **Save as type** field, select one of these image types:
7. Click **Save** to complete the export process.

Navigating Between DOORS and Rhapsody

If the model has been exported to DOORS, exported elements have a **Navigate to DOORS** option on their pop-up menus. Selecting this option for an exported model element opens DOORS and takes you to the shadow element in DOORS that corresponds to the exported element. Bidirectional navigation is supported between DOORS and Rhapsody. Refer to the online help for details.

Adding Annotations to Diagrams

You can add different types of annotations to specify additional information about a model element. The annotation can add semantics (like a constraint or requirement) or can simply be informative, like a documentation note or comment.

Note

None of the annotation types generate code; they are used to improve the readability and comprehension of your model.

- ◆ **Constraint**—A condition or restriction expressed in text. It is generally a Boolean expression that must be true for an associated model element for the model to be considered well-formed. A constraint is an assertion rather than an executable mechanism. It indicates a restriction that must be enforced by the correct design of a system.

Constraints are part of the model and are, therefore, displayed in the browser.

- ◆ **Comment**—A textual annotation that does not add semantics, but contains information that might be useful to the reader and is displayed in the browser.
- ◆ **Note**—A textual annotation that does not add semantics, but contains information that might be useful to the reader. Notes are *not* displayed in the browser.
- ◆ **Requirement**—A textual annotation that describes the intent of the element. Note that a requirement modeled inside Rhapsody does not replace the usage of a dedicated requirement traceability tool, such as DOORS. Instead, a modeled requirement complements the usage of such a tool because the hierarchical modeling of requirements enables you to easily correlate each requirement to the element that addresses it.

Requirements are part of the model and are therefore displayed in the browser.

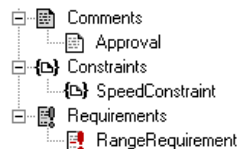
Creating Annotations

To add an annotation to a diagram, follow these steps:

1. Click the appropriate icon on the **Drawing** toolbar for the type of annotation you want to create.
2. Single-click or click-and-drag in the diagram to place the annotation at the desired location.
3. Type the note text or expression.
4. Press **Ctrl+Enter** to terminate editing.

The same annotation can apply to more than one model element and multiple annotations can apply to the same model element. You can use “ownership” instead of an anchor. For example, if a requirement is owned by a class, it is a requirement of the class.

The new annotation is displayed in the diagram, and in the browser if it is a modeled annotation. Note that documentation notes and text are *graphical annotations* and exist only in the diagram; all the other Rhapsody annotations are *modeled annotations*, and are part of the model itself. Because modeled annotations are part of the model, they can be viewed in the browser. In addition, you can move modeled annotations to new owners using the browser, and can drag-and-drop them from the browser into diagrams. Modeled annotations are displayed under separate categories in the browser by type, as shown in the following figure.



Alternatively, you can create modeled annotations in the browser, as follows:

1. Right-click the element that will own the annotations.
2. Select **Add New > Constraint, Comment, or Requirement**. The new annotations is added under the selected element.

An annotation created using the browser does not anchor the annotation to the specified model element. However, you can organize annotations such that anchoring is implied.

Creating Dependencies Between Annotations

You can show dependencies between annotations using the **Dependency** icon on the **Drawing** toolbar. Follow these steps:

1. Click the **Dependency** icon in the **Drawing** toolbar.
2. Click the edge of the dependent annotation.
3. Click the edge of the annotation on which the first annotation depends.

Creating Hierarchical Requirements

You can create hierarchical requirements by having one requirement own all the subrequirements; the owning requirement is called the “sum requirement.”

1. In the browser, right-click the component that will own the requirement and select **Add New > Requirement** from the pop-up menu.
2. Name the new requirement. This is the sum requirement.
3. Right-click the sum requirement and select **Add New > Requirement** from the pop-up menu.
4. Name the subrequirement.
5. Continue creating subrequirements as needed.
6. Create dependencies between the requirements as needed.

The following figure shows a hierarchical requirement.



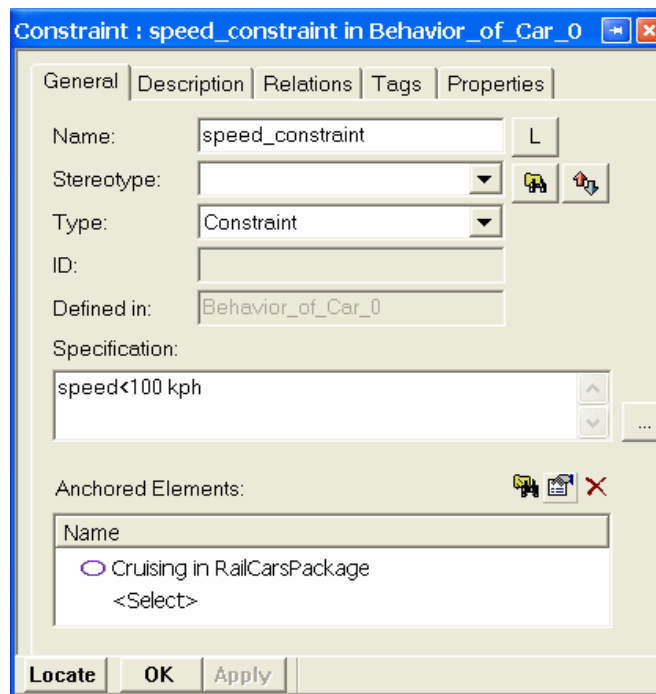
Editing Annotation Text

To edit the text of an existing annotation, do one of the following:

- ◆ Double-click the annotation.
- ◆ Right-click the annotation, then select **Features** from the pop-up menu.



Defining the Features of an Annotation

The Features dialog box enables you to change the features of an annotation, including its name type. The following figure shows the Features dialog box for an annotation (in this case, a constraint).



A constraint contains the following fields:

- ◆ **Name**—Specifies the name of the constraint. To enter a detailed description of the constraint, click the **Description** tab.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the constraint, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

- To select from a list of current stereotypes in the project, click the folder with binoculars  button.
- To sort the order of the stereotypes, click the up and down arrows  button.

Note: The COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM Development Guide* for more information about these components.

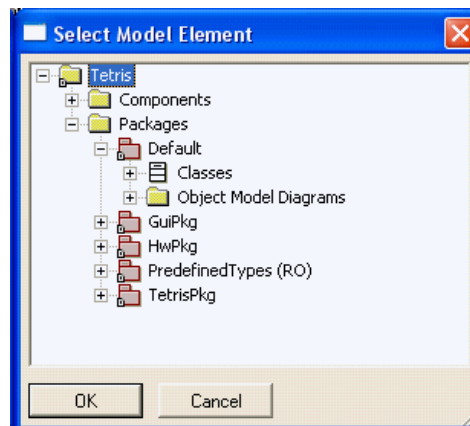
- ◆ **Type**—Specifies the annotation type. For example, you could use this field to easily change a constraint to a requirement or comment.
- ◆ **Defined in**—This read-only field specifies the owning class of the annotation.
- ◆ **Specification**—Specifies the note text or constraint expression.
- ◆ **Anchored Elements**—Lists all the elements that are anchored to this annotation.



To view the features for an anchored element, select the element in the list, then click the **Invoke Feature dialog** button.



To anchor additional elements to this annotation, click the **Select** button or the <Select> line in the list to select the elements from the hierarchical display, as shown in the following figure.



Converting Notes to Comments

As previously stated, documentation notes “live” only within diagrams: they are not displayed in the browser. To convert a note to a comment (to be displayed in the browser), follow these steps:

1. In the diagram, right-click the note.
2. Select **Convert to Comment** from the pop-up menu.

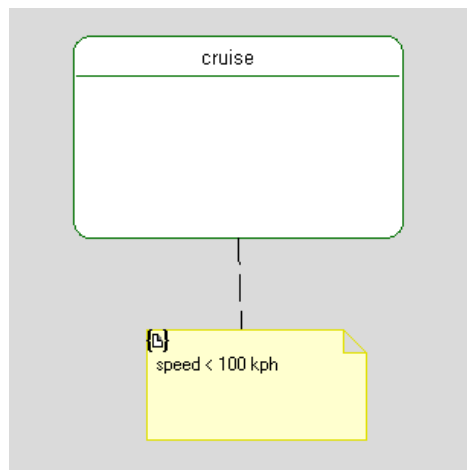
Anchoring Annotations

You can anchor a constraint, comment, requirement, or note to one or more graphical objects that represent modeling concepts (classes, use cases, and so on). You can also anchor annotations to edge elements.

Note

Although it is possible to anchor a requirement to an element, it is better to model it as a dependency.

Anchors are shown as dashed lines, as illustrated in the following figure.



To anchor an annotation to a model element, follow these steps:

1. Click the **Anchor** icon on the **Drawing** toolbar.
2. Click an edge of the annotation.
3. Move the cursor to the edge of the model element to which you want to anchor the annotation, then click to confirm.

To change the line style of an anchor, right-click the anchor line and select the appropriate option from the **Line Shape** menu.

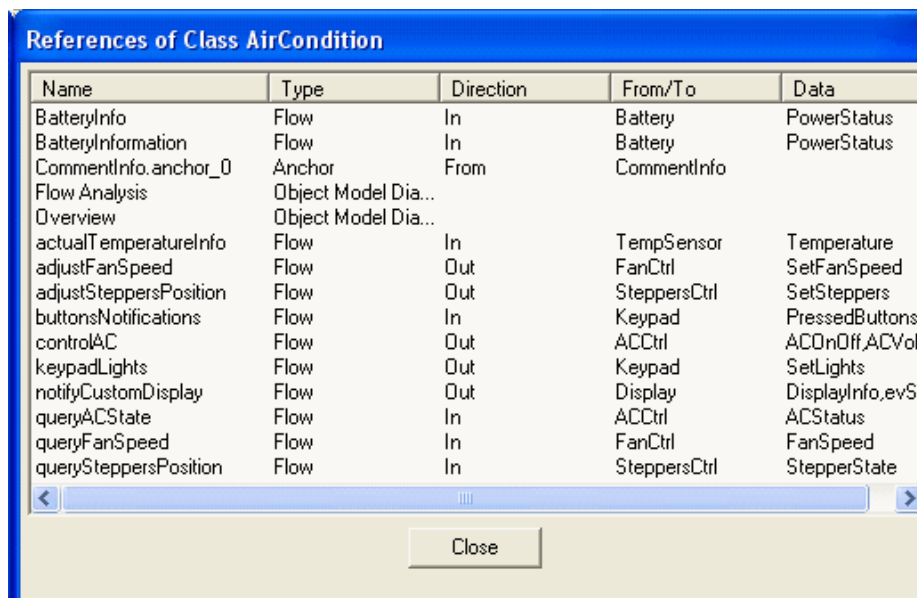
Note the following:

- ◆ There should be only one anchor between a specific annotation and a given model element.
- ◆ An annotation can be anchored to more than one element.

Finding Constraint References

To find which annotations are anchored to a particular class, follow these steps:

1. In the browser, select the class you want to query.
2. Select **References** from the pop-up menu. The References dialog box lists all the references for the specified class. Anchored annotations are labeled “Anchor” in the Type column.



Name	Type	Direction	From/To	Data
BatteryInfo	Flow	In	Battery	PowerStatus
BatteryInformation	Flow	In	Battery	PowerStatus
CommentInfo.anchor_0	Anchor	From	CommentInfo	
Flow Analysis	Object Model Dia...			
Overview	Object Model Dia...			
actualTemperatureInfo	Flow	In	TempSensor	Temperature
adjustFanSpeed	Flow	Out	FanCtrl	SetFanSpeed
adjustSteppersPosition	Flow	Out	SteppersCtrl	SetSteppers
buttonsNotifications	Flow	In	Keypad	PressedButtons
controlAC	Flow	Out	ACCtrl	ACOnOff,ACVolt
keypadLights	Flow	Out	Keypad	SetLights
notifyCustomDisplay	Flow	Out	Display	DisplayInfo,evSt
queryACState	Flow	In	ACCtrl	ACStatus
queryFanSpeed	Flow	In	FanCtrl	FanSpeed
querySteppersPosition	Flow	In	SteppersCtrl	StepperState

Deleting an Anchor

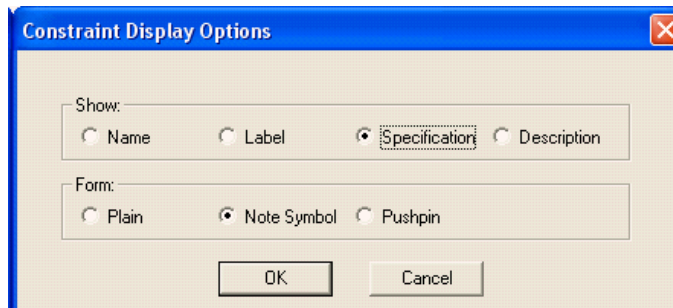
To delete an anchor, follow these steps:

1. Select the anchor.
2. Click the **Delete** tool in the main toolbar.

Display Options for Annotations

To change the display of annotations, right-click the note in the diagram and select **Display Options** from the pop-up menu. The Display Options dialog box for the selected annotation type opens, with only the relevant fields enabled.

The following figure shows the display options for a constraint.

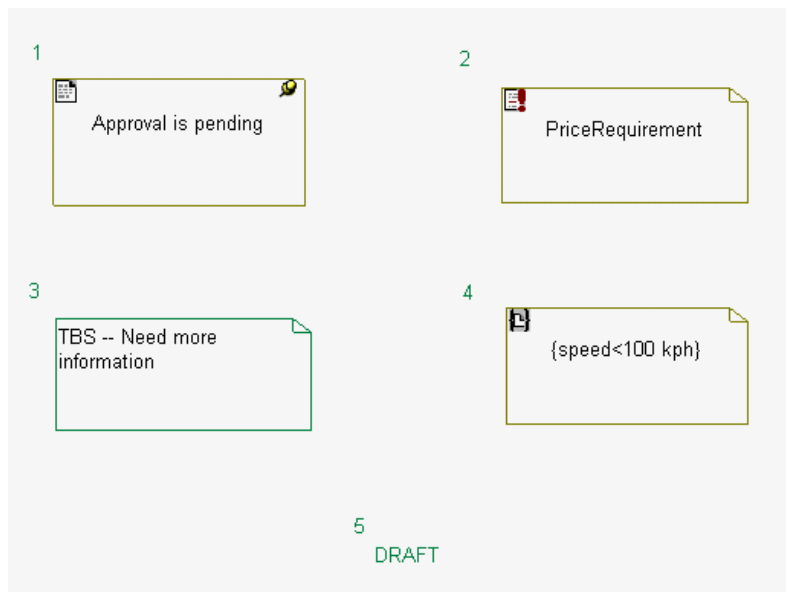


The fields are as follows:

- ◆ **Show**—Specifies which information to display for the annotation:
 - **Name**—Displays the name of the annotation
 - **Label**—Displays the annotation’s label
 - **Specification**—Displays the contents of the **Specification** field of the Features dialog box for the annotation
 - **Description**—Displays the contents of the **Description** field of the Features dialog box for the annotation
- ◆ **Form**—Specifies the graphical form of the annotation:
 - **Plain**—Shows the annotation without a border, as free-floating text
 - **Note Symbol**—Shows the annotation within a frame, with a symbol in the upper, left-hand corner to denote the annotation type
 - **Pushpin**—Displays the annotation with a rectangular border and a pushpin icon

The following figure shows some of the different display options. The annotation types and display options are as follows:

- ◆ Comment, displayed with its specification label using the pushpin style
- ◆ Requirement, displayed using its label
- ◆ Documentation note
- ◆ Constraint
- ◆ Text note



Deleting an Annotation

To delete an annotation, follow these steps:

1. In the browser, right-click the annotation to be deleted.
2. Select **Delete from Model** from the pop-up menu.

To delete an annotation from a diagram, right-click the note and do one of the following:

- ◆ Select the **Delete** icon from the toolbar.
- ◆ Select **Delete from Model** from the pop-up menu.

Using Annotations with Other Tools

The following table lists the effect of annotations on both Rhapsody and third-party tools.

Tool	Description
COM API	Annotations are supported by the COM API via the following interfaces: <ul style="list-style-type: none"> • IRPAnnotation • IRPComment • IRPConstraint • IRPRequirement Refer to the <i>Rhapsody API Reference Guide</i> for more information.
Complete Relation	When you select Layout > Complete Relations , anchors are part of the information added to the diagram.
DiffMerge	Annotations are included in difference and merge operations. Refer to the <i>Rhapsody Team Collaboration Guide</i> for more information on the DiffMerge tool.
Populate Diagram	Annotations and anchors are not supported.
References	If you use the References functionality for a modeled annotation, the tool lists the diagrams in which the specified annotation appears. When you select a diagram from the returned list, the annotation is highlighted in the diagram. If you use the References functionality for an element, the tool includes any annotations that are anchored to the specified element See Searching in the Model for more information on this functionality.
Report on model	Modeled annotations are listed by type under the owning package. Graphical annotations are not included in the list.
Rational Rose	Notes in Rose (which are not displayed in the browser) are imported as notes in RhapsodyAnnotations.
Search in model	When you search in a model, the search includes modeled annotations, including the Body section. When selected, the annotation is highlighted in the browser. See Searching in the Model for more information on this functionality.

Annotation Limitations

Note the following limitations for annotations:

- ◆ You cannot anchor an annotation to an element that is represented by a line (such as a message or transition).
- ◆ You cannot drag-and-drop an anchor from the browser to a diagram.

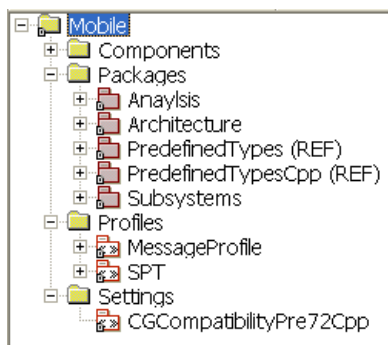
Using Profiles

A *profile* is a special kind of package that is distinguished from other packages in the browser. You specify a profile at the top level of the model. Therefore, a profile is owned by the project and affects the entire model. By default, all profiles apply to all packages available in the workspace, so their tags and stereotypes are available everywhere. A profile is very similar to any other package; however, profiles cannot be nested.

A profile “hosts” domain-specific tags and stereotypes. *Tags* enable you to add information to certain kinds of elements to reflect characteristics of the specific domain or platform for the modeled system. Tags are easily viewable in their own category in the browser and in the **Tags** tab of the Features dialog box for an element.

You can apply tags to certain cases by associating the tag definition to stereotypes in a profile or a package. In this case, the tags are visible only for those model elements that have the specified stereotype. In addition, you may apply tags to individual elements.

The browser displays all of the profiles used in a project together, as shown in this example:



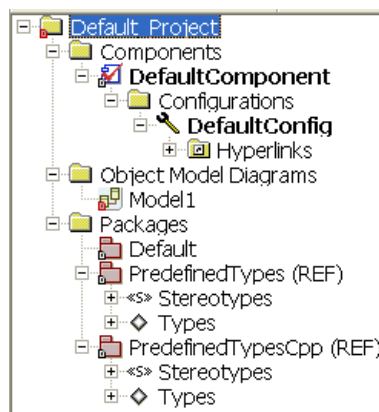
This example lists two profiles. **MessageProfile** is a user-defined (customized) profile. See [Creating Your Own Profile](#) for more information about these profiles. **SPT** was automatically added to the project because the developer specifies the Scheduling, Performance, and Time method to add timing analysis data. See [Types of Profiles](#) for more information.

Projects without Profiles

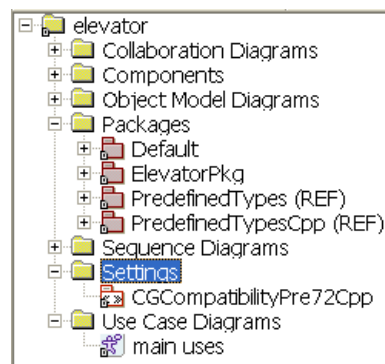
You are not required to use profiles in your Rhapsody project. When creating a new project, you may create a *Default* project, containing all of the basic UML features, by following these steps:

1. Create the new project by either selecting **File > New**, or click the **New project** icon in the main toolbar.
2. Replace the default project name (Project) with `<your project name>` in the **Project name** field. Enter a new directory name in the **In folder** field or Browse to find an existing directory.
3. In the **Type** field, select *Default*.
4. Click **OK**.

The basic structure for your new project is displayed in the browser with no Profiles folder, as shown in this example.



If you have a project that was created in an older version of Rhapsody and you begin to work on it in a newer version, Rhapsody automatically inserts the required *settings* to manage any compatibility issues, as shown in this sample project's browser.



You may note any of the following compatibility settings automatically loaded into your project:

- ◆ **CGCompatibilityPre61C** makes the code generation backwards compatible with pre-6.1 Rhapsody in C models.
- ◆ **CGCompatibilityPre61Cpp** makes the code generation backwards compatible with pre-6.1 Rhapsody in C++ models.
- ◆ **CGCompatibilityPre61Java** makes the code generation backwards compatible with pre-6.1 Rhapsody in Java models.
- ◆ **CGCompatibilityPre61M1C** makes the code generation backwards compatible with pre-6.1 maintenance release 1 Rhapsody in C models.
- ◆ **CGCompatibilityPre61M1Cpp** makes the code generation backwards compatible with pre-6.1 maintenance release 1 Rhapsody in C++ models.
- ◆ **CGCompatibilityPre70C** makes the code generation backwards compatible with pre-7.0 Rhapsody in C models.
- ◆ **CGCompatibilityPre70Cpp** makes the code generation backwards compatible with pre-7.0 Rhapsody in C++ models.
- ◆ **CGCompatibilityPre70Java** makes the code generation backwards compatible with pre-7.0 Rhapsody in Java models.
- ◆ **CGCompatibilityPre71C** makes the code generation backwards compatible with pre-7.1 Rhapsody in C models.
- ◆ **CGCompatibilityPre71Cpp** makes the code generation backwards compatible with pre-7.1 Rhapsody in C++ models.
- ◆ **CGCompatibilityPre71Java** makes the code generation backwards compatible with pre-7.1 Rhapsody in Java models.
- ◆ **CGCompatibilityPre72C** makes the code generation backwards compatible with pre-7.2 Rhapsody in C models.
- ◆ **CGCompatibilityPre72Cpp** makes the code generation backwards compatible with pre-7.2 Rhapsody in C++ models.
- ◆ **CGSimplifiedModelProfileC** provides the simplifier C code generation settings.
- ◆ **CGSimplifiedModelProfileCpp** provides the simplifier C++ code generation settings.

Types of Profiles

Rhapsody provides the following types of profiles:

- ◆ Rhapsody's predefined [Profiles](#), such as AutomotiveC and DoDAF, can be selected from the New Project dialog box when creating a new project or added to a project later, as described in [Adding a Rhapsody Profile Manually](#).
- ◆ User-defined profiles, as described in [Creating Your Own Profile](#).
- ◆ Add-on product profiles, which require an additional license, are automatically added when the engineer uses the associated add-on product, or it can be added using the **File > Add to Model** option.

Converting Packages and Profiles

You can convert any existing package into a profile and vice versa. To convert a package, right-click the package in the browser, then select **Change to > Profile**. Similarly, to convert a profile to a package, right-click the profile in the browser, then select **Change to > Package**.

Profile Properties

By default, all profiles apply to all packages available in the workspace. To associate existing profiles with new Rhapsodyannotation models, use the following properties (under `General::Profile`):

- ◆ `AutoCopied`—Specifies a comma-separated list of physical paths to profiles that will automatically be copied into new projects when they are created (using the **Add to Model, As Unit** functionality). By default, this property is an empty string.
- ◆ `AutoReferences`—Specifies a comma-separated list of physical paths to profiles that will automatically be referenced by new projects when they are created (using the **Add to Model, As Reference** functionality). By default, this property is an empty string.

Note that you can specify packages in the property values. The paths can be disks, networks, or Universal Naming Conventions (UNC), and the extension `.sbs` is optional. If Rhapsodyannotation cannot find at least one of the specified profiles, it generates an error message.

You can use profiles like any other package, including exchanging them between users and using configuration management tools with them.

Using a Profile to Enable Access to Your Custom Help File

This section shows you how you can enable access to your custom help file from within Rhapsody with the use of the **F1** key. This feature is applicable for a Rhapsody project that has a Rhapsody profile with a New Term stereotype defined for the profile. For information about profiles, see [Using Profiles](#). For information about stereotypes, see [Defining Stereotypes](#).

As referred to by Rhapsody, a custom help file consists of a help file and a map file, both of which you must create. There may be times when you have a project for which you would like to access your own help file. This may be particularly useful when there is a team working on the same project and you want them to share the same specific information. For example, your company may create its own help file to document its project terminology and project/corporate standards.

To enable the ability to access your custom help file from within Rhapsody, you have to set properties to identify and locate your custom help file and map file. Then for an element associated with the New Term stereotype in the Rhapsody profile for your Rhapsody project, when a user presses **F1** to call a help topic, your custom help file would open instead of the main Rhapsody help file.

Keep in mind that the Rhapsody product provides you with an extensive help file that always appears when there is no custom help file available.

Note

IBM is not responsible for the content of any custom help file and map file.

The creation, functioning, testing, and maintenance of a custom help file and map file are the responsibilities of the creator(s) of these files.

About Creating Your Custom Help File and Map File

A custom help file consists of a help file and a map file, both of which you create. See [About Creating Your Custom Help File](#) and [Creating Your Map File](#).

About Creating Your Custom Help File

Your custom help file must be in HTML format. (Example help file name: myhelp.htm.)

If you plan to share the custom help file with team members, place it in a shared location on a network. You identify this file and its location in the `Model::Stereotype::CustomHelpURL` property, as detailed in [Enabling Access to Your Custom Help File](#).

Creating Your Map File

The map file for your custom help file must contain one or more Rhapsody resource IDs for which the custom help is provided. A resource ID corresponds to a particular GUID element in the Rhapsody product. For example, for the **Attributes** tab, the resource ID is 161328.

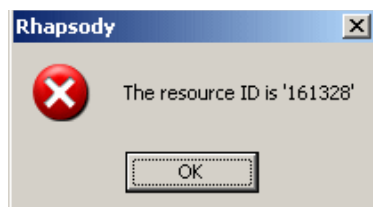
To find a Rhapsody resource ID, follow these steps:

1. Add the following line to the [General] section of the `rhapsody.ini` file and save your changes:

```
ShowActiveWindowHelpID=TRUE
```

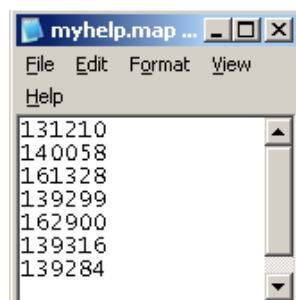
This means that now when you press **F1** to open the Rhapsody help file, you will see a dialog box with a resource ID instead, as shown in the next step.

2. In Rhapsody, press **F1** for a GUID element that you plan to associate with a New Term stereotype (for example, if your New Term stereotype is applicable to a class, the **Attributes** tab of the Features dialog box). The following figure shows the message box that shows the resource ID for the **Attributes** tab of the Features dialog box.



3. Add the resource ID number to your help map file. Each number must be on its own line. The following figure shows the contents of a map file called `myhelp.map`. (Another example of a help map file name: `myhelp.txt`.)

Note: For testing purposes as mentioned in [Enabling Access to Your Custom Help File](#), include the resource IDs shown in the following figure in your map file. You can delete these IDs later.



Note: You may find it useful to add the name of the element next to the resource ID number (for example, 161328 `Attributes` tab).

4. If you plan to share the custom help file with team members, place the help map file in a shared location on a network. You identify this file and its location in the `Model::Stereotype::CustomHelpMapFile` property, as detailed in [Enabling Access to Your Custom Help File](#).

Note: When your map file is complete, to ensure that help topics appear instead of resource ID numbers when you press **F1**, you can comment out the `ShowActiveWindowHelpID=TRUE` line in the `rhapsody.ini` file (for example, insert a semicolon or pound sign in front of the line) or you can delete the line.

Bookmarking Your Custom Help File for a Specific Resource ID

When you press **F1** to open your custom help file (when possible), it opens at the beginning of the help file.

If for a particular resource ID (for example, the **Tags** tab of the Features dialog box) you want your custom help file to open to a specific spot in the custom help file, add a bookmark to that spot in the help file. Use the standard HTML `<a name>` and `` tags.

For example, the `myhelp.htm` help file has three sections labeled: `Attributes`, `Ports`, and `Tags`. When you open the help file, you see the `Attributes` section first. If for the **Tags** tab, which is resource ID 139316, you want to open the custom help file at the `Tags` section of the help file, you would code it as follows in the `myhelp.htm` help file:

```
<a name="139316">Tags</a>
```

Then when you press **F1** on the **Tags** tab, the custom help file opens at the `Tags` section of the help file.

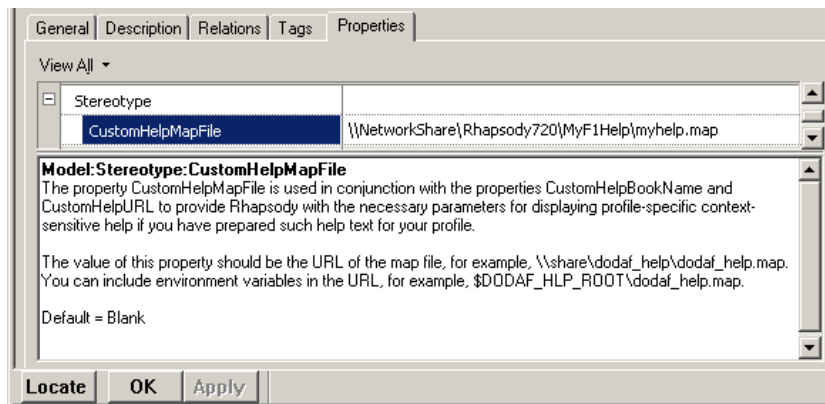
Enabling Access to Your Custom Help File

Note

These instructions assume you have created your custom help file and your map file. See [About Creating Your Custom Help File](#) and [Creating Your Map File](#).

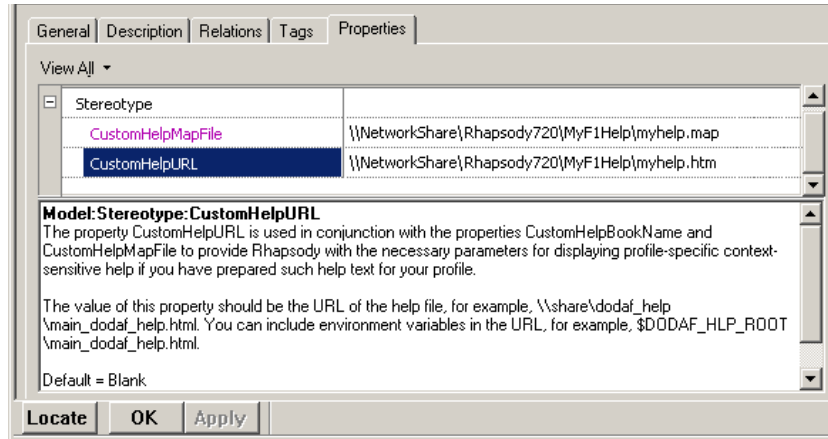
To make these instructions easier to follow, they assume that there is a fictional project called **AutomobileProject** that has a profile called **Auto2009** with a New Term stereotype called **Stereotype_Auto2009**. These profile and stereotype names are for illustrative purposes only; they are not provided in Rhapsody. (For information about creating a profile, see [Creating Your Own Profile](#). For information about stereotypes, see [Defining Stereotypes](#).) To enable access to your custom help file, follow these steps:

1. For the profile or its stereotype for the Rhapsody project on which you want to provide custom help, set the `Model::Stereotype::CustomHelpURL` and `CustomHelpMapFile` properties as follows:
 - a. Open the Features dialog box for the profile or its New Term stereotype. On the Rhapsody browser, double-click the profile name **or** stereotype name (for example, profile **Auto2009** or stereotype **Stereotype_Auto2009**).
 - b. On the **Properties** tab, select **All** from the **View** drop-down arrow (the label changes to **View All**).
 - c. Locate `Model::Stereotype::CustomHelpMapFile`. Type the path to your help map file, as shown in the example in the following figure.



Note: You can specify an environment variable as part of the URL (for example, `$DODAF_HLP_ROOT\dodaf_help.map`).

- d. Locate `Model::Stereotype::CustomHelpURL`. Type the path to your help file, as shown in the example in the following figure.



Note: You can specify an environment variable as part of the URL (for example, `$DODAF_HLP_ROOT\dodaf_help.htm`).

- e. Click **OK**.
2. Create an element using the New Term stereotype. Right-click a package in your project and select **Add New > Auto2009 > Stereotype_Auto2009**. Notice that these choices are located at the bottom of the pop-up menu.
3. Double-click the stereotype created in the previous step (for example, `stereotype_auto2009_0`) and press **F1**. If you followed these steps and used the resource IDs show in [Creating Your Map File](#), your custom help file should open.

Testing the Custom Help File

Test your custom help file before releasing it to the users of the help file. You should be sure the help file works as expected before releasing it. For example:

- ◆ If the help file is to be accessed over a network, make sure it is accessible. You may want to have someone else (besides the person who set it up) try to access the custom help file from within Rhapsody over the network before rolling out the feature to the rest of your project team.
- ◆ If you have set it so that your custom help opens at specific spots in the help file for specific resource IDs, be sure to check these particular links. See [Bookmarking Your Custom Help File for a Specific Resource ID](#).

Using the Custom Help File

Once set up, anyone who uses the profile and elements with the New Term stereotype you set up can use the **F1** key to open the custom help file when and where it is possible. When you press **F1**, Rhapsody checks if there is a custom help file for the New Term stereotype associate with the element.

- ◆ If yes, Rhapsody searches for the resource ID in your help map file and it opens the custom help file.
- ◆ If no, Rhapsody opens its main help file.

Note

The GUI element must be applicable to the New Term stereotype for the Rhapsody profile for your custom help file to work.

For example, if the **Auto2009** profile has a New Term stereotype called **Stereotype_Auto2009** and you add an element with this stereotype to a package called **Default**, when you open the Features dialog box for one of these stereotypes (for example, **stereotype_auto2009_0**) and then you press **F1**, your custom help should open (assuming you added the resource IDs for the tabs on this dialog box to your help map file).

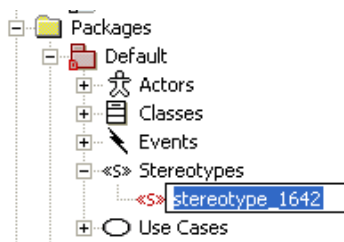
However, when you open the Features dialog box for the package and you press **F1**, you open the main Rhapsody help file because this package is not associated with the New Term stereotype call **Stereotype_Auto2009**.

Defining Stereotypes

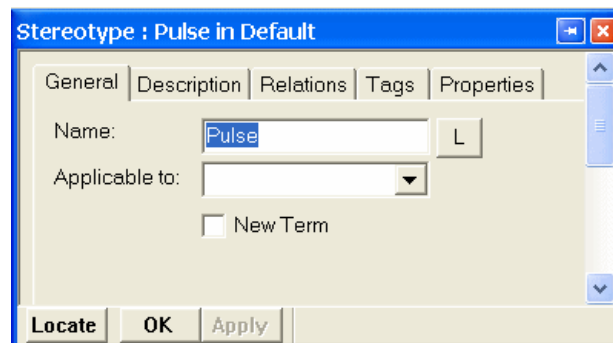
Defined stereotypes are displayed in the browser. Stereotypes can be owned by both packages and profiles.

To define a stereotype, follow these steps:

1. In the Rhapsody browser, right-click the profile or package that owns the stereotype and then select **Add New > Stereotype**. Rhapsody creates a new stereotype called `stereotype_n` under the **Stereotypes** category.



2. Enter a name for the new stereotype.
3. Double-click the new stereotype to open its **Features** dialog box, as shown in the following figure:



4. Select one or more metaclasses to which the stereotype is applicable to.
5. For profiles, if you are working in a domain with specialized terminology and notation, select the **New Term** check box to create a new metaclass. Since a term is based on an out-of-box metaclass, it functions in the same manner as its base metaclass. See [Special Stereotypes](#) for more information about new terms.
6. Click **OK** to apply your changes and close the Features dialog box for the stereotype.

7. Optionally, to set the formatting for the stereotype, right-click the new stereotype name in the browser and select **Format**. Use the Format dialog box to define the visual characteristics for the stereotype.

Associating Stereotypes with an Element

You associate stereotypes with a model element using the **Stereotype** field of the Features dialog box for that element. The **Stereotype** field includes a drop-down list of all the stereotypes defined in the profiles and packages that can extend the metaclass of that element. For example, if the element is a class, the **Stereotype** list includes all the stereotypes in all the profiles and packages that extend that class.

To associate stereotypes with an element:

1. Open the Features dialog box for the element.
2. Open the **Stereotype** drop-down list.
3. Use the check boxes to select the stereotypes you would like to apply to the element.
4. Click the arrow of the drop-down list to close the list.
5. Click **Apply** or **OK**.

Alternatively, you can select the stereotypes from a tree display, as follows:


1. Open the Features dialog box for the element.
2. Click the Browse button next to the drop-down list.
3. Find the stereotypes you would like to apply. Use **Ctrl** and **Shift** to select more than one stereotype.
4. Click **OK** to close the tree display.
5. Click **Apply** or **OK**.

Associating Stereotypes with a New Term Element

Stereotypes can also be applied to elements that are based on “new term” stereotypes. In such cases, the **Stereotype** drop-down list will not contain the stereotype on which the element is based, nor any other “new term” stereotypes.

Order of Stereotypes in List

Stereotypes can be selected for display at the top of the drop-down list. To change the order in which the selected stereotypes are displayed, follow these steps:

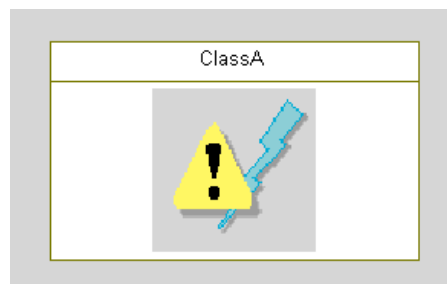
1. Click the Change Stereotype Order button , or right-click the stereotype list when it is closed and select **Edit Order** from the context menu.
2. When the list of selected stereotypes is displayed, use the up and down arrows to reorder the list.
3. Click **OK**.

Associating a Stereotype with a Bitmap

Rhapsody provides a set of predefined bitmaps in the directory `<Rhapsody_installation>\Share\PredefinedPictures`. You can associate these icons with Rhapsody stereotypes and classes.

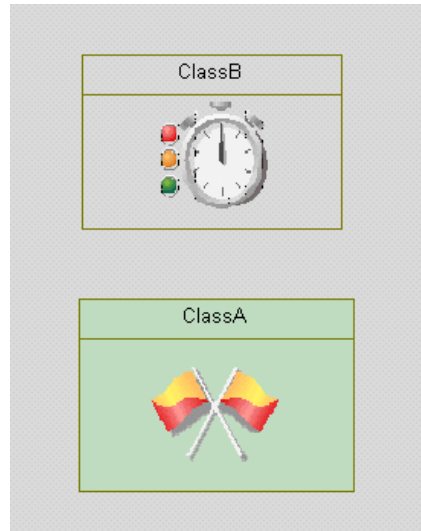
1. Define a stereotype of your own or select one of the existing stereotypes.
2. Select this stereotype for a class.
3. Rename an existing `.bmp` file (or add a new file) in the `PredefinedPictures` directory so it has the same name as the stereotype.
4. Drag-and-drop the class into an OMD.
5. Right-click the class, and select **Display Options** from the pop-up menu.
6. Under **Show stereotype**, select **Icon**; change **Display name** to **Name only**.

The bitmap is included in the class box, as shown in the following figure.



To change the bitmap background to transparent (so only the graphic itself is visible), set the property `General::Graphics::StereotypeBitmapTransparentColor` to the RGB value of the bitmap background.

The following figure shows bitmaps with transparent backgrounds.



Deleting Stereotypes

1. Select the stereotype to delete.
2. Click the **Delete** button.
3. Click **OK** to close the dialog box.

Stereotype Inheritance

Stereotype inheritance allows you to extend existing stereotypes. Stereotypes can inherit from predefined stereotypes or from stereotypes that you have created.

The derived stereotype inherits the following characteristics from its base stereotype:

- ◆ Applicability (which elements it can be applied to)
- ◆ Properties (this includes locally-overridden properties)
- ◆ Tags

While the initial values of properties for the derived stereotype are those that were inherited from the base stereotype, the values can be overridden for the derived stereotype.

You can add tags to the derived stereotype, and add elements to the list of elements to which it can be applied. To establish stereotype inheritance, follow these steps:

1. In the browser, right-click the stereotype that will be the derived stereotype.
2. From the context menu, select **Add New > Super Stereotype**

These steps can be repeated in order to create a stereotype that inherits from a number of other stereotypes.

Special Stereotypes

If a stereotype inherits from one of the “special” stereotypes, for example, usage or singleton, it inherits the special meaning of the base stereotype.

If a stereotype inherits from a “new term” stereotype, then it is also a “new term” stereotype. However, the “new term” status of the derived stereotype is removed if you do something that contradicts this status, for example, using multiple inheritance such that the derived stereotype ends up being applicable to more than one type of element.

Using Tags to Add Element Information

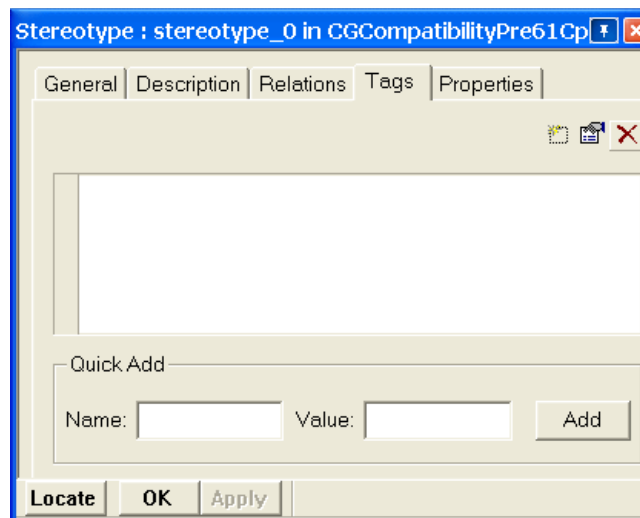
Tags are used to add information to the model base specific to the domain or platform. You can access them using the **Tags** tab of the Features dialog box for Rhapsody model elements. The **Tags** tab shows the tag definitions and values associated with the given element. You can create tags for a stereotype, metaclass, or individual element.

Defining a Stereotype Tag

When the **Tags** tab applies to a stereotype, it specifies the tag definition for all elements that use the given stereotype.

To define a new tag, follow these steps:

1. Create the profile to hold the tag if it does not already exist (see [Creating Your Own Profile](#)).
2. If it does not already exist, define a stereotype for the profile and select <<New>> (see [Defining Stereotypes](#)). The Features dialog box opens.
3. Select the **Tags** tab to display this dialog box.



4. The **Quick Add** fields allow you to define the tag's name and default value quickly and click **Add**.
5. You may want to enter a more detailed description of the tag in the area above the Quick Add.
6. Click **OK** to apply your changes and close the dialog box.

The new tag is added to the **Tags** tab. In addition, it is added to the browser, as shown in the following figure.



This sample tag is listed as `Profile::Component::<TagName>` (in this example, `Avionics::Component::RiskFactor`) because it was defined under the stereotype of the profile.

You would use this tag for components with the corresponding stereotype. For example, if you have a component named `System` with the `SafetyCritical` stereotype, its **Tag** tab would include the tag `Avionics::SafetyCritical::RiskFactor`.

Note

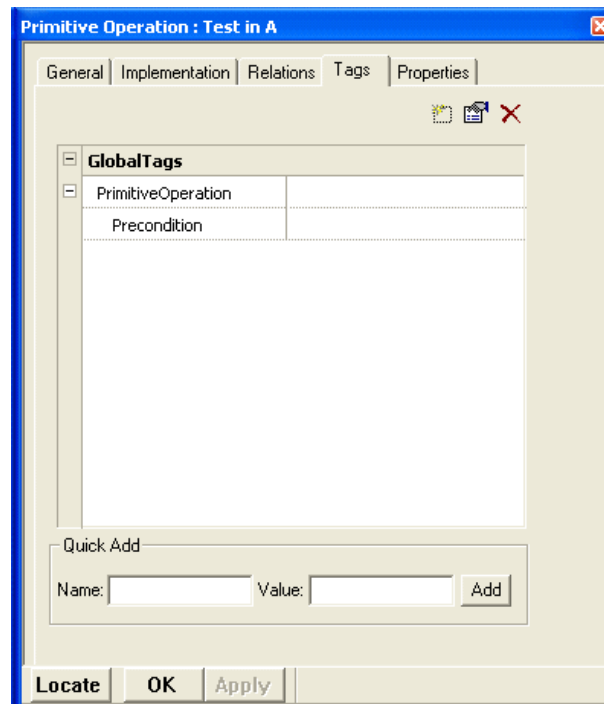
To create a new tag using the browser, simply right-click the stereotype and select **Add New > Tag** from the pop-up menu. If desired, rename the tag.

Defining a Global Tag

When the **Tags** tab applies to a metaclass, it hosts all the tag definitions that are available to all instances of a certain type (anywhere within the model)—without the need to set a stereotype. When you define a tag at the metaclass level, the **Applicable to** field is read-write so you can select the appropriate element type from the drop-down list.

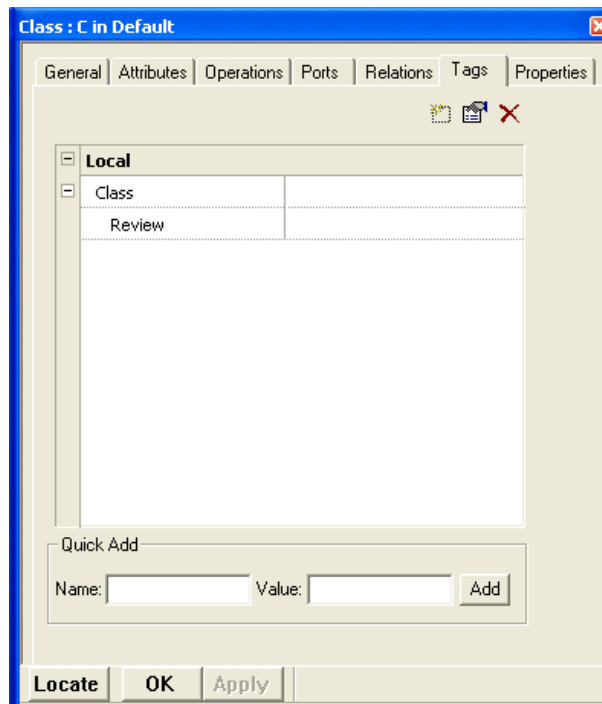
For example, you could create a new tag to specify prerequisite attributes for all primitive operations in the project by selecting the `Primitive Operation` element from the **Applicable to** drop-down list. This tag will be included automatically in the **Tags** tab of any primitive operation in the project as `<ProfileName>::<ElementType>::<TagName>`.

For example, `GlobalTags::PrimitiveOperation::Precondition`, as shown in the following figure.



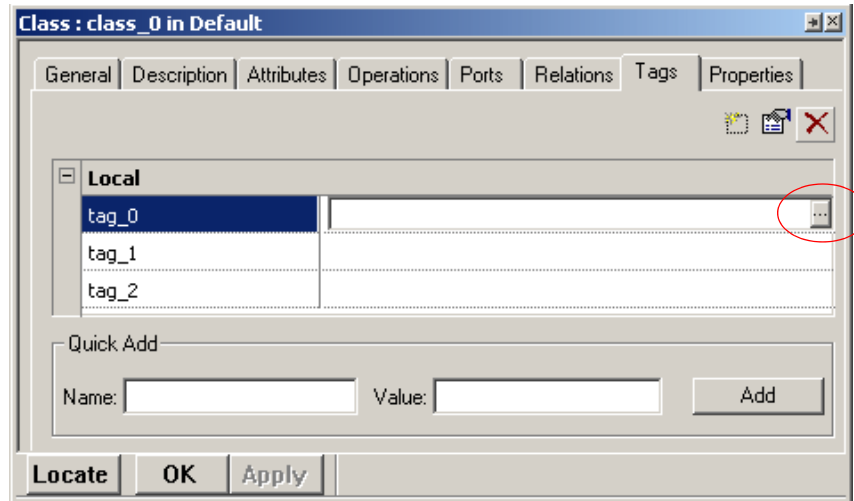
Defining a Tag for an Individual Element

You can set a tag on an individual element to flag it in some way. When you create a tag for an individual element, it is listed in the tab as `LocalTags::Class::<TagName>`. For example, `Local::Class::Review`, as shown in the following example.



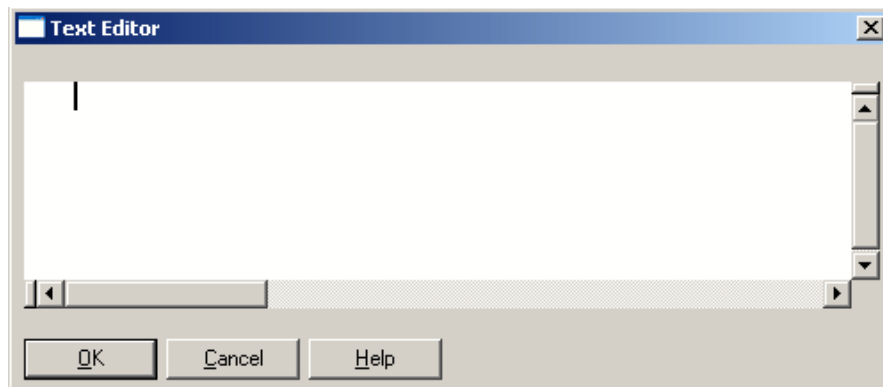
Adding a Value to a Tag

To add a value to a tag, click the Ellipsis button at the right end of the box next to the name of the tag, as shown in the following figure, to open the internal text editor.



Using the Internal Text Editor

The Rhapsody internal text editor, as shown in the following figure, is a simple text editor on which you can enter text or code (depending on the functionality of the element you are working with). When using the internal text editor is possible, Rhapsody provides you with access to it. For example, you can open the internal text editor by clicking an Ellipsis button, such as can be found on the **Tags** tab and the Properties of the Features dialog box.




Deleting a Tag

You can use the Rhapsody browser or the Features dialog box to delete a tag Select.

To delete a tag using the browser, follow these steps:

1. Right-click the tag on the browser and select **Delete from Model**.
2. Click **Yes** to confirm your requested action.

To delete a tag using the Features dialog box, follow these steps:

1. Open the Features dialog box for the element to which the tag belongs and select the **Tags** tab.
2. Select the tag you want to delete and click the **Delete** button  in the upper-right corner of the tab.

Note

If you delete a tag definition in a stereotype, it is removed from the list of tags. However, if the tag's value has been overridden, that tag will not be removed.

The Internal Editor

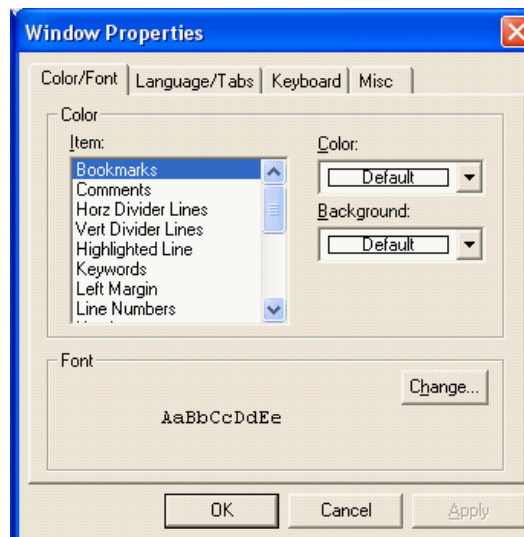
When you choose to edit code, Rhapsody launches the internal editor. This editor has a wide range of features to assist with editing.

Unlike external editors, the Rhapsody internal editor provides dynamic model-code associativity (DMCA). The DMCA feature of Rhapsody enables you to edit code and automatically roundtrip your changes back into the model. It also generates code if the model has changed. If you use an external editor, DMCA will no longer be available.

Window Properties

The Window Properties dialog box enables you to customize the Rhapsody internal editor window. Note that these window properties are completely separate from Rhapsody project properties. To invoke the dialog box, right-click anywhere within the editor window and select **Properties** from the pop-up menu.

Alternatively, press **Alt+Enter** on the keyboard. You can customize this shortcut using the **Keyboard** tab on the Window Properties dialog box. The following figure shows the Windows Properties dialog box.



The dialog box contains the following tabs:

- ◆ **Color/Font**
- ◆ **Language/Tabs**
- ◆ **Keyboard**
- ◆ **Misc**

The following sections describe how to use these tabs in detail.

The Color/Font Tab

The internal editor highlights syntax elements for easy editing. The default color settings follow standard code editing conventions. Using the **Color/Font** tab, shown in the figure, you can customize the colors and highlighting settings.

In addition to the default keywords (such as `class` and `public`), you can specify the additional language-specific keywords to be color-coded by the Rhapsody internal editor by setting the value of the property `General::Model::AdditionalLanguageKeywords` to the comma-separated list of additional keywords you want to have color-coded.

Changing the Default Colors

The following table lists the default color and highlighting settings.

Item	Foreground	Background
Bookmarks	Default	Default
Comments	Green	Default
Keywords	Blue	Default
Left Margin	White	N/A
Numbers	Teal	Default
Operators	Red	Default
Scope Keywords	Blue	Default
Strings	Purple	Default
Text	Black	Default
Window	Default	N/A

To change the colors or highlighting, follow these steps:

1. Select the **Color/Font** tab on the Windows Properties dialog box.

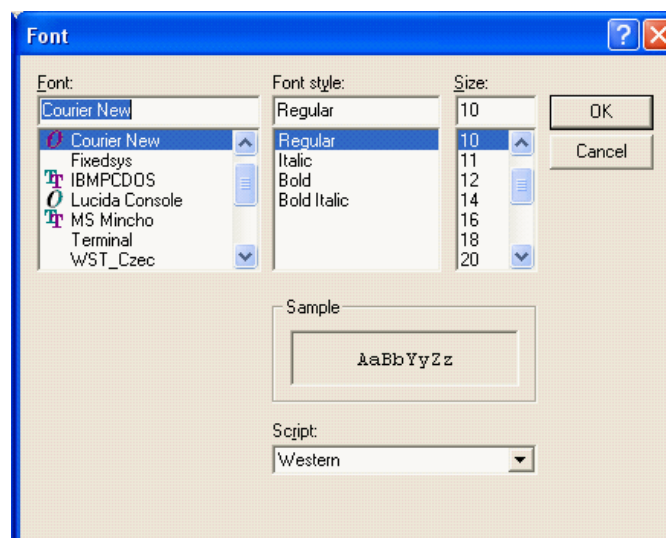
2. Select a code element from the **Item** list.
3. Choose a text color by selecting it in the **Foreground** drop-down list.
4. Choose a highlighting color by selecting it in the **Background** drop-down list. Note that background highlighting is not available for all elements.
5. Click **Apply** to apply your changes and close the dialog box.

Changing the Default Font

By default, the internal editor uses Courier New, regular, 10-point font to display text. It supports any fixed-pitch font. When you change the font, it affects the appearance of all text in the editor.

To change the font, follow these steps:

1. In the Font area of the **Color/Font** tab, click **Change**. The Font dialog box opens, as shown in the following figure.



2. As desired, select new values for the **Font**, **Font style**, and **Size** fields. You can view the effects of your changes in the Sample field.
3. Click **OK** to close the Font dialog box.
4. Click **OK** to apply your changes and dismiss the Window Properties dialog box.

The Language/Tabs Tab

The **Language/Tabs** tab, shown in the following figure, controls the indentation and tab size used by the editor.



The **Language/Tabs** tab contains the following fields:

- ◆ **Auto indentation style**—Specifies whether lines are automatically indented according to either the language scope or to the previous line in the file.

The possible values are as follows:

- **Off**—Turns off automatic indentation.
- **Follow language scoping**—Indents the code according to the language specifications. This is the default setting.
- **Copy from previous line**—Uses the indentation established in the previous line of code.
- ◆ **Tab**—Specifies how many spaces make up a tab space.
If you want the tab character converted to spaces after insertion, check the **Convert tabs to spaces while typing** box.

Note: Changes to tab size do not affect existing tab spacing. The new size applies to tabs entered after the change is made.

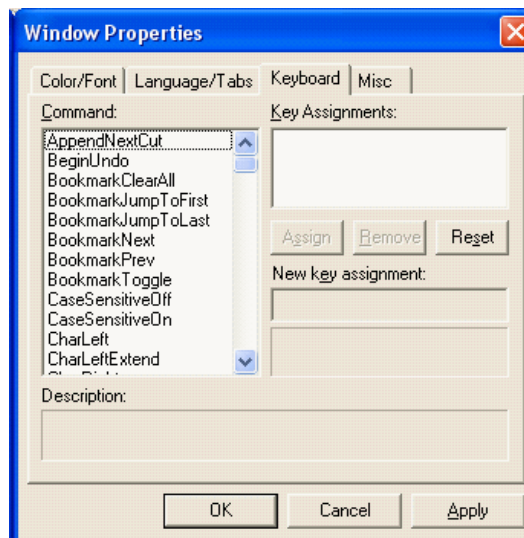
- ◆ **Language**—Specifies your programming language.

If you want any case errors corrected automatically for you, check the option **Fixup text**

case while typing language keyword. For example, if this option is enabled and you typed “WHile” as the keyword, CodeMax automatically corrects the case of the keyword to read “while.”

The Keyboard Tab

The **Keyboard** tab, shown in the following figure, allows you to create shortcuts.



The edit commands are invoked using keyboard shortcut keys. Mapping edit commands to easy-to-use and easy-to-remember keyboard shortcuts reduces the time and difficulty of editing text. Most commands have been mapped to a default shortcut. You can modify all shortcuts from the **Keyboard** tab of the Window Properties dialog box.

Custom Keyboard Mappings

The internal editor provides customizable keyboard mappings so you can create your own shortcuts for edit commands, or assign shortcuts to commands that do not have a default assigned.

To assign keyboard shortcuts, follow these steps:

1. On the **Keyboard** tab, select an edit command from the **Command** list.

If a shortcut has been assigned to this command, it is displayed in the **Key Assignments** field.

A short description of the command is displayed in the **Description** field.

2. Click in the **New Key Assignment** box to activate it for editing.
3. Use the keyboard to type the shortcut key sequence. If you make a mistake, click **Reset** and type the sequence again.
4. Click **OK** to apply your changes and close the dialog box.

Default Keyboard Mappings

The following table lists the default edit commands that have been assigned shortcut keys.

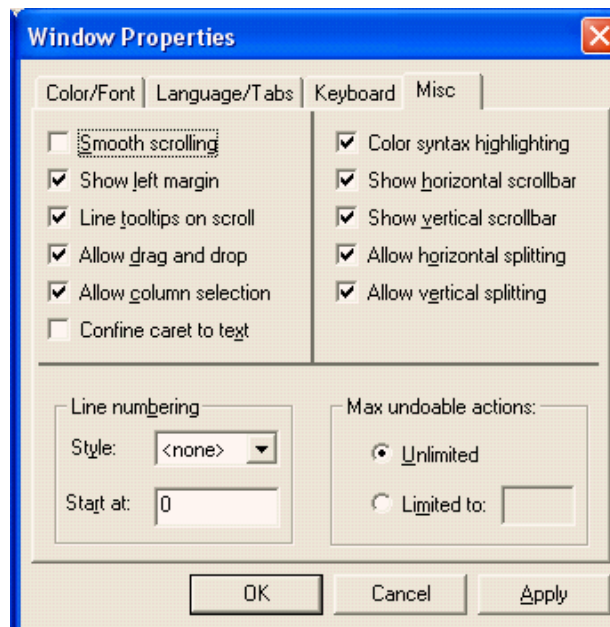
Command	Keystroke
BookmarkNext	F2
BookmarkPrev	Shift + F2
BookmarkToggle	Ctrl + F2
CharLeft	Left
CharLeftExtend	Shift + Left
CharRight	Right
CharRightExtend	Shift + Right
Copy	Ctrl + C
Copy	Ctrl + Insert
Cut	Shift + Delete
Cut	Ctrl + X
CutSelection	Ctrl + Alt + W
Delete	Delete
DeleteBack	Backspace

Command	Keystroke
DocumentEnd	Ctrl + End
DocumentEndExtend	Ctrl + Shift + End
DocumentStart	Ctrl + Home
DocumentStartExtend	Ctrl + Shift + Home
Find	Alt + F3
Find	Ctrl + F
FindNext	F3
FindNextWord	Ctrl + F3
FindPrev	Shift + F3
FindPrevWord	Ctrl + Shift + F3
FindReplace	Ctrl + Alt + F3
GoToLine	Ctrl + G
GoToMatchBrace	Ctrl +]
Home	Home
HomeExtend	Shift + Home
IndentSelection	Tab
LineCut	Ctrl + Y
LineDown	Down
LineDownExtend	Shift + Down
LineEnd	End
LineEndExtend	Shift + End
LineOpenAbove	Ctrl + Shift + N
LineUp	Up
LineUpExtend	Shift + Up
LowercaseSelection	Ctrl + U
PageDown	Next
PageDownExtend	Shift + Next
PageUp	PRIOR
PageUpExtend	Shift + Prior

Command	Keystroke
Paste	Ctrl + V
Paste	Shift + Insert
Properties	Alt + Enter
RecordMacro	Ctrl + Shift + R
Redo	Ctrl + A
SelectLine	Ctrl + Alt + F8
SelectSwapAnchor	Ctrl + Shift + X
SentenceCut	Ctrl + Alt + K
SentenceLeft	Ctrl + Alt + Left
SentenceRight	Ctrl + Alt + Right
SetRepeatCount	Ctrl + R
TabifySelection	Ctrl + Shift + T
ToggleOvertyping	Insert
ToggleWhitespaceDisplay	Ctrl + Alt + T
Undo	Ctrl + Z
Undo	Alt + Backspace
UnindentSelection	Shift + Tab
UntabifySelection	Ctrl + Shift + Space
UppercaseSelection	Ctrl + Shift + U
WindowScrollDown	Ctrl + Up
WindowScrollLeft	Ctrl + Page Up
WindowScrollRight	Ctrl + Page Down
WindowScrollUp	Ctrl + Down
WordDeleteToEnd	Ctrl + Delete
WordDeleteToStart	Ctrl + Backspace
WordLeft	Ctrl + Left
WordLeftExtend	Ctrl + Shift + Left
WordRight	Ctrl + Right
WordRightExtend	Ctrl + Shift + Right

The Misc Tab

The **Misc** tab, shown in the following figure, controls numerous miscellaneous attributes for the editor.



The following sections describe how to use some of the more interesting features available on the **Misc** tab.

Using Split Views

You can split the editor window into up to four simultaneous views of one file, where each view scrolls independently from the others. You can make changes in any view and all other views are automatically updated.

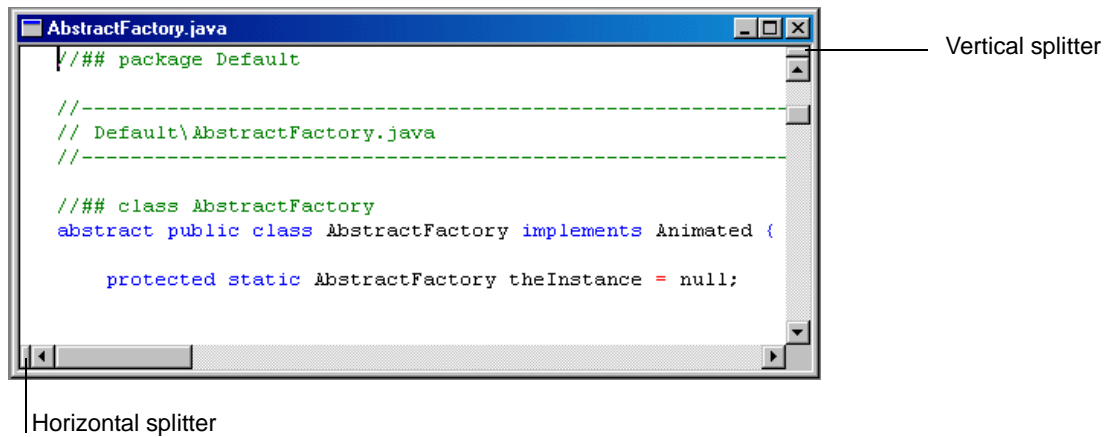
To allow splitting of the screen into two horizontal panes, check the **Allow Horizontal Splitting** box on the **Misc** tab. To allow splitting of the screen into two vertical panes, check the **Allow Vertical Splitting** box.

To split the screen horizontally:

1. Click the horizontal splitter at the left end of the horizontal scroll bar (see the figure).
2. While holding down the mouse button, drag the splitter to the right until the new pane appears.

To split the screen vertically, follow these steps:

1. Click the vertical splitter at the top of the vertical scroll bar.
2. While holding down the mouse button, drag the splitter downward until the new pane appears.



Mouse Actions

Use the mouse to select and edit text in the editor window. The following table lists the available mouse actions.

Operation	Mouse Action
Display pop-up menu.	Right-click.
Select entire line.	Click in the left margin next to the target line of text.
Select multiple lines.	Click in the left margin next to a line of text and drag the mouse up or down.
Select entire word.	Double-click anywhere in the word.
Move text (drag-and-drop)	Select text; hold down the left mouse button while dragging the selection to the new location.
Copy text (drag-and-drop)	Select text; press Ctrl and hold down the left mouse button while dragging the selection to the new location.

Undo and Redo

The internal editor allows you to undo any number of operations, and redo the previous operation.

The edit commands for undo and redo are as follows:

- ◆ **Undo**—**Alt+Backspace**, or **Ctrl+Z**
- ◆ **Redo**—**Ctrl+A**

For information on editing keyboard shortcuts for these commands, see [Custom Keyboard Mappings](#).

You can set the maximum number of undo operations from the Window Properties dialog box; the default is to allow an unlimited number of undo operations.

To set the maximum number of undo actions, follow these steps:

1. On the **Misc** tab, in the **Maximum Undoable Actions** section, select **Limited to**.
2. In the adjacent box, type the maximum number of undo actions you want to allow.
3. Click **OK**.

Searching

The internal editor contains a useful search feature that finds keywords within the current file.

To search for a keyword, follow these steps:

1. Right-click and select **Find** from the pop-up menu.
2. In the **What** box, type the search string.
3. Select one of the following options:
 - ◆ **Match whole word only**—Find instances where the search term is the whole word.
 - ◆ **Match case**—Find matching instances that have the same case as the search term.
 - ◆ **Direction**—Choose **Up** to search text above the cursor position, or **Down** to search text below the cursor position.
4. Click **Find** to find the next instance of the search term or click **Mark All** to place a bookmark in the left margin next to all instances matching the search term.

Bookmarks

The bookmark feature places a blue triangular marker in the left margin to identify a line of text.

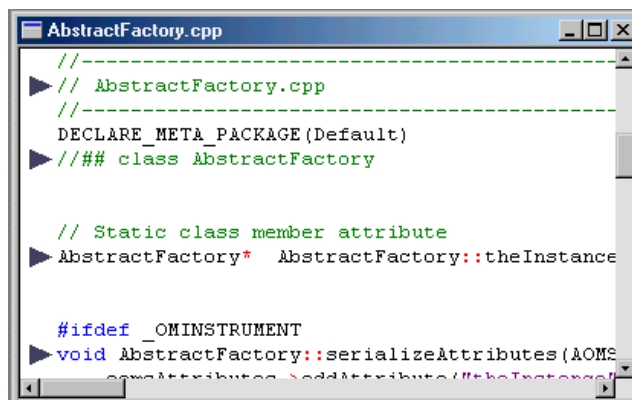
To place a bookmark in the margin, use the `BookmarkToggle` edit command. The default shortcut for `BookmarkToggle` is **Ctrl+F2**.

Repeat the `BookmarkToggle` command to remove the bookmark.

The bookmark commands are as follows:

- ◆ **BookmarkToggle**—**Ctrl+F2**
- ◆ **BookmarkNext**—**F2**
- ◆ **BookmarkPrev**—**Shift+F2**
- ◆ **BookmarkJumpToFirst**—Unassigned
- ◆ **BookmarkJumpToLast**—Unassigned
- ◆ **BookmarkClearAll**—Unassigned

For instructions on editing the keyboard shortcuts for these commands, see [Custom Keyboard Mappings](#).



```
AbstractFactory.cpp
//-----
// AbstractFactory.cpp
//-----
DECLARE_META_PACKAGE(Default)
//## class AbstractFactory

// Static class member attribute
AbstractFactory* AbstractFactory::theInstance

#ifdef _OMINSTRUMENT
void AbstractFactory::serializeAttributes(AOMS
...Attributes &addAttribute/#theInstance/
```

Printing

To print a file from the internal editor, follow these steps:

1. Make sure that the editor window containing the file you want to print is the active window in Rhapsody.
2. From the Rhapsody **File** menu, select **Print**.
3. In the Print dialog box, select the printing options, then click **Print**.

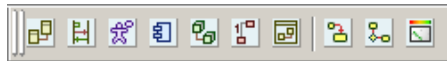
Graphic Editors

The Rhapsody graphic editors for the different UML diagrams provide the tools needed to create different views of a software model. Each graphic editor is described in detail in subsequent sections of this guide. This section describes the pop-up menus and features that are common to all the graphic editor.

Creating New Diagrams

You can create a new UML diagram using the **Diagrams** toolbar, Edit menu, Tools menu, or browser.

The following figure shows the **Diagrams** toolbar.



The following topics describe how to use these tools.

Creating Statecharts and Activity Diagrams

Statecharts and activity diagrams describe the behavior of a particular class. They can be added only at the class level. A class can have either a statechart or an activity diagram, but not both.

To create a statechart or activity diagram, follow these steps:

1. Select a class in the browser.
2. Select the appropriate diagram from the Tools menu or click the appropriate button in the **Diagrams** toolbar. The tools are as follows:



Statechart



Activity Diagram

A new diagram is displayed in the drawing area.

Alternatively, you can create new diagrams by right-clicking the class in the browser, then selecting **Add New** > <diagram name>.

Creating all Other Diagram Types

The other diagrams can be grouped under the project or a package in the project hierarchy.

To create a new diagram, follow these steps:

1. Select the appropriate diagram type from the **Tools** menu or click the appropriate button on the **Diagrams** toolbar:



Object Model Diagram



Component Diagram



Structure Diagram



Deployment Diagram



Use Case Diagram



Collaboration Diagram



Sequence Diagram



Panel Diagram

2. From the Open Diagram dialog box, select the project or a package where you would like to add the diagram.
3. Click the **New** button at the far right. The New Diagram dialog box opens.
4. Type a name for the new diagram in the **Name** box.
5. Depending on the diagram type, the New Diagram dialog box can contain the following options:
 - a. **Populate Diagram**—Automatically populates the diagram with existing model elements. This option applies to object model, use case, and structure diagrams.

See [Automatically Populating Diagrams](#) for more information.
 - b. **Operation Mode**—Specifies whether to create an *analysis* SD, which enables you to draw message sequences without adding classes and operations to the model; or a *design* SD, in which every message you create or rename is realized to an operation in the static model.

See [Sequence Diagrams](#) for more information.
6. Click **OK** to create the new diagram.

Alternatively, you can create new diagrams by right-clicking the class in the browser, then selecting **Add New** > <diagram name>.

Opening Existing Diagrams

To open an existing UML diagram, double-click the diagram in the browser. The diagram opens in the drawing area.

Alternatively, follow these steps:

1. Click the appropriate button in the **Diagrams** toolbar (see [Creating New Diagrams](#) for illustrations of the buttons). The Open Diagram dialog opens.
2. For statecharts and activity diagrams, select the class that the diagram describes from the list of available diagrams. For all other diagrams, select the diagram you want to open.
3. Click **OK**. The diagram opens in the drawing area.

As with other Rhapsody elements, the Features dialog box for the diagram enables you to edit its features, including the name, stereotype, and description. See [Using the Features Dialog Box](#) for more information.

Note

To go forward from opened diagram to opened diagram, choose **Window > Forward**. To go in the reverse direction, choose **Window > Back**. As well, you can select a currently opened diagram by its name. A list of the currently opened diagrams appear at the end of the **Window** menu.

Deleting Diagrams

In most cases, you can delete existing UML diagrams only from the browser. However, you can delete statecharts and activity diagrams from both the browser and from the Tools menu.

To delete an existing diagram, follow these steps:

1. Select the diagram from the browser.
2. Right-click and select **Delete from Model**, or press the **Delete** key.
3. Rhapsody asks you to confirm the operation. Click **Yes**.

The diagram is removed from the model.

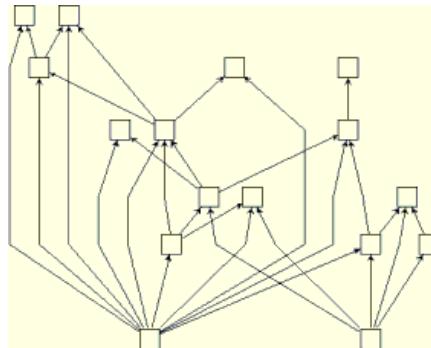
Automatically Populating Diagrams

When you create a new use case, object model, or structure diagram, you can use the **Populate Diagram** feature to populate the diagram automatically with existing model elements. You can select which model elements to add to the diagram. Rhapsody automatically lays out the elements in an orderly and easily comprehensible manner.

Once the diagram has been created, you can edit it by adding or deleting elements to tailor it to your needs. This feature is particularly useful for quickly creating diagrams after reverse engineering or importing a model. When you automatically populate a new diagram, Rhapsody offers a choice of layout style. You can select either of these styles for any relation type:

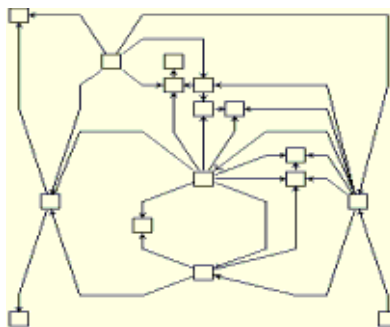
- ◆ **Hierarchical**

The elements are organized according to levels of inheritance, with lower levels inheriting from upper levels. Each layer is organized to minimize the crossing and bending of inheritance arrows and is individually centered in the diagram. You can choose this style for any type of relation. If there are no classes or inheritance, the elements might appear organized as layers, depending on the direction of the populated diagrams.



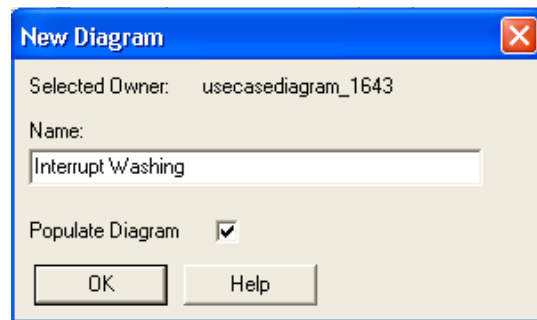
- ◆ **Orthogonal**

The entire drawing is made as compact as possible. Classes are placed to minimize the intersection, length, and bending of relation arrows. You can use this style for any relation, including inheritance relations.



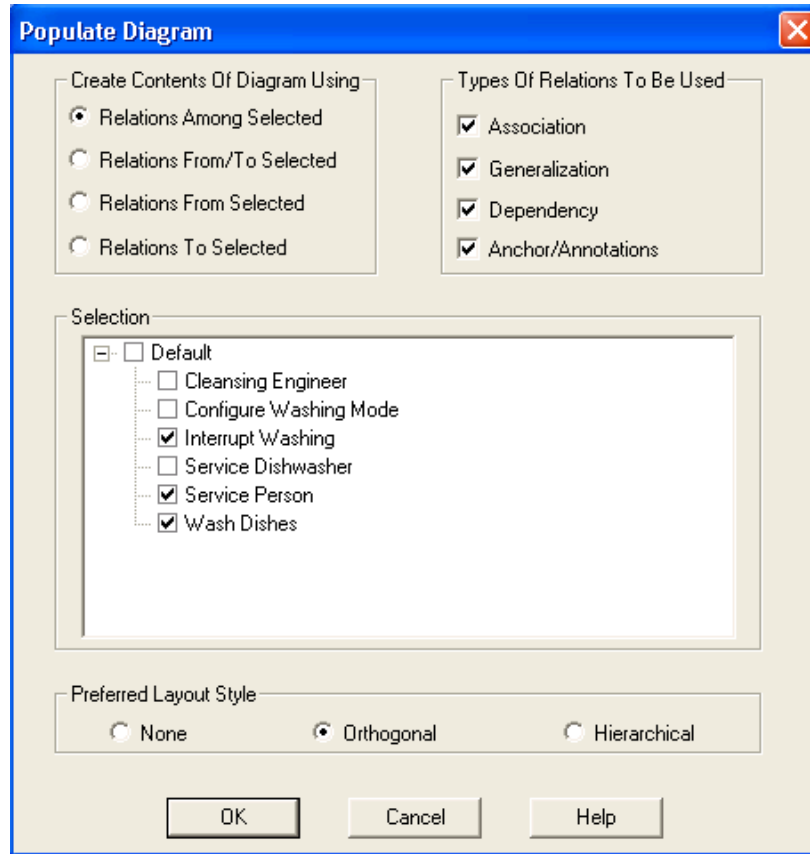
To create a new, automatically populated diagram, follow these steps:

1. In the browser, highlight an existing use case, object model, or structure diagram where you want to create another of the same element.
2. Right-click and select **Add New <element type>** and this dialog box displays.



3. Type the **Name** of the new element. This example is creating a new use case diagram.

- Click the check box for **Populate Diagram** and click **OK**. The Populate Diagram dialog box opens, as shown in the following example.



- In the **Create Contents of Diagram Using** section, indicate how you would like Rhapsody to create the contents of the diagram. The options are as follows:
 - Relations Among Selected**—Populates the diagram only with the selected elements and the relations between them
 - Relations From/To Selected**—Populates the diagram with the selected elements, their incoming and outgoing relations, and the model elements that complete these relations
 - Relations From Selected**—Populates the diagram with the selected elements, their outgoing relations, and the model elements that complete the relations
 - Relations To Selected**—Populates the diagram with the selected elements, their incoming relations, and the model elements that complete the relations

6. In the **Types of Relations to be Used** section, select which types of relations you would like Rhapsody to use when creating the contents of the diagram. The options are as follows:
 - a. OMDs and structure diagrams: **Instance, Association/Aggregation, Inheritance, Dependency, Link, and Anchor/Annotations**
 - b. UCDs: **Association, Generalization, and Dependency, and Anchor/Annotations**
7. Under **Selection**, place a check mark next to each element you want to include in the new diagram.

To select a package without selecting the elements it contains, right-click the package.
8. In the **Preferred Layout Style** section, select the type of layout you would like Rhapsody to use when creating the diagram. If you select **None**, Rhapsody automatically chooses the best layout style according to the type of relations you have chosen to display.
9. Click **OK**. The new diagram appears in the drawing area with all of the selected elements added. You can then begin to add information to the new diagram.

Note

When auto-populating a diagram, if you want Rhapsody to populate each class so it shows its attributes and operations, set the properties `ObjectModelGE::Class::ShowAttributes` and `ObjectModelGE::Class::ShowOperations` to `All` in the scope of the package or project.

Property Settings for the Diagram Editor

The properties under `General::Graphics` control how features of the diagram editors operate. The following table lists the available properties.

Property	Description
<code>AutoScrollMargin</code>	Controls how responsive the autoscrolling functionality is
<code>ClassBoxFont</code>	Specifies the default font for new class names
<code>CRTerminator</code>	Specifies how multiline fields in notes and statechart names should interpret a carriage return (CR)
<code>DeleteConfirmation</code>	Specifies whether confirmation is required before deleting a graphical element from the model
<code>ExportedDiagramScale</code>	Specifies how an exported diagram is scaled and whether it can be split into separate pages for better readability
<code>FixBoxToItsTextuals</code>	Specifies whether to resize boxes automatically to fit their text content (such as names, attributes, or operations)
<code>grid_color</code>	Specifies the default color used for the grid lines
<code>grid_snap</code>	Specifies whether the Snap to Grid feature is enabled for new diagrams, regardless of whether the grid is actually displayed
<code>grid_spacing_horizontal</code>	Specifies the spacing, in world coordinates, between grid lines along the X-axis when the grid is enabled for diagrams
<code>grid_spacing_vertical</code>	Specifies the spacing, in world coordinates, between grid lines along the Y-axis when the grid is enabled for diagrams
<code>HighlightSelection</code>	Specifies whether items should be highlighted when you move the cursor over them in a diagram
<code>LandScapeRotateOnExport</code>	Rotates an exported metafile so it can fit on a portrait page
<code>MaintainWindowContent</code>	Specifies whether the viewport (the part of a diagram displayed in the window) is kept for window resizing operations when you change the zoom level, providing additional space in the diagram in a smooth manner
<code>MarkMisplacedElements</code>	Specifies whether misplaced elements are marked in a special way. Previously, misplaced elements were shown with a small X in the upper corner
<code>MultiScaleByOne</code>	Specifies whether objects in the diagram keep the same amount of space between them when you scale them (Cleared)
<code>PrintLayoutExportScale</code>	Specifies the factor by which the Windows metafile format (WMF) files are scaled down in order to fit on one page
<code>RepeatedDrawing</code>	Specifies whether repetitive drawing mode (stamp mode) is enabled

Property	Description
ShowEdgeTracking	Specifies whether to show the “ghost” edges of a linked element when you move it
ShowLabels	Specifies whether to display labels instead of names in diagrams
StereotypeBitmap TransparentColor	Creates a “transparent” background for bitmaps associated with stereotypes (so only the graphics are displayed in the class box)
Tool_tips	Enables the display of tooltips

You can set these properties by selecting **File >Project Properties**.

For detailed information on how the `General::Graphics` properties affect the drawing of your model, refer to the definitions displayed in the **Properties** tab of the Features dialog box or examine the complete list of property definitions in the *Rhapsody Property Definitions* PDF file available from the *List of Books*. That list can be searched along with the other PDF versions of the Rhapsody documentation to locate specific property definitions and the procedures that use them.

Setting Diagram Fill Color

To set the color of the diagram background:

1. Right-click in the diagram window.
2. From the context menu, select **Diagram Fill Color**
3. Select a color.

Creating Elements

The diagram editors enable you to create a number of elements to enhance your model. To create any kind of element, you must first select one of the drawing tools in the diagram editor’s toolbar.

When Rhapsody is in drawing mode, the cursor includes a tooltip showing an icon of that element. For example, the following cursor is displayed when you are drawing a class.



The items you can draw in the diagram editors fall into two main categories: boxes and lines. In addition, Rhapsody enables you to create freestyle shapes. The following sections describe how to create these elements.

Repetitive Drawing Mode

By default, each time you want to add an element to a diagram, you must first click the appropriate icon in the **Drawing** toolbar.

In some cases, however, you may want to add a number of elements of the same type. To facilitate this, Rhapsody includes a *repetitive drawing mode*.

To enter repetitive drawing mode, click the “stamp” icon in the **Layout** toolbar. Now, after choosing a tool in the **Drawing** toolbar, you will be able to continue drawing elements of that type without selecting the tool again each time. If you choose a different tool from the toolbar, then Rhapsody will allow you to draw multiple elements of the newly selected type.

After you click the icon, Rhapsody remains in repetitive drawing mode until you turn it off. To turn off the repetitive mode, just click the "stamp" icon a second time.

Drawing Boxes

Each editor contains tools to draw boxes. The following table lists the box elements available with each editor.

Diagram Editor	Box Elements
Object model	Classes, packages, composite classes, objects, files, actors, and annotations
Use case	Use cases, actors, systems boundary boxes, packages, and annotations
Component	Components, files, folders, and annotations
Deployment	Nodes, components, and annotations
Collaboration	Objects, multi-objects, actors, and annotations
Statechart	States and annotations
Activity	Actions, subactivities, action blocks, object nodes, swimlane frames, connectors, and annotations
Structure	Composite classes, objects, and annotations

To draw a box, follow these steps:

1. Select a box tool.

2. Move the cursor to the drawing area, and do one of the following:
 - a. **Quick-Draw**—Click once to draw a box with the default size, shape, and name.
 - b. **Drag**—To draw classes, simple classes, and packages, click and drag to the opposite corner and release. If you hold down the Shift key, you create a square box.

Note that the cursor changes to the icon of whatever you are drawing (class, object, package, and so on).
3. When you complete the box, the element is given a default name. To rename it, click the text to enable editing. Type the new name, then press **Enter** or click outside the box.
4. Click anywhere outside the box to start a new box, or click the **Select** tool to stop creating boxes.

To add an existing box element to a diagram, you can simply drag-and-drop the element from the browser to the drawing area.

Drawing Arrows

Arrows connect boxes, representing the connections between different boxes in the diagram. For example, associations and dependencies are two types of arrows that can be drawn in use case diagrams.

You can draw arrows in the following ways:

- ◆ **Drag**—Click inside the source box, drag, and release the mouse inside the destination box.
- ◆ **Simple arrow clicks**—Click inside the source box and click the border of the destination box.
- ◆ **Multiple clicks**—Click inside the source box, click any number of times to mark the control points on the path of the arrow, and double-click inside the destination box. You cannot add control points to a message arrow because it allows only two clicks. Another point appears immediately to show where the next arrow will start.

Note that the cursor changes when you create arrows:

- ◆ If you click a valid element for the destination of the arrow (for example, you are drawing a transition and you click a class), the cursor changes to crosshairs in a small circle.
- ◆ If you try to connect an arrow to an invalid element, Rhapsody displays a “no entry” symbol to show that you cannot connect the arrow to that element, as shown in the following figure.



Changing the Line Shape

Rhapsody has three line shapes that you can use when you are drawing arrows in the graphic editors. You can set a unique line shape for each line; the default line shape varies by element.

The Edit menu contains the **Line Shape** option, which specifies your preferred line shape for the specified arrow. You can also access the **Line Shape** option by right-clicking an arrow or line in a diagram.

The possible values for the line shape are as follows:

- ◆ **Straight** changes the line to a straight line.
- ◆ **Spline** changes the line to a curved line.
- ◆ **Rectilinear** changes the line to a group of line segments connected at right angles.
- ◆ **Re-Route** removes excess control points to make the line more fluid.

Note

You can use the `line_style` property to change the line shape (straight, spline, rectilinear) for a line type (for example, Link, Dependency, Transition, Generalization) for a diagram type (for example, Object Model, Activity, Statechart) by project. For example, in your Handset project, you can use the `ObjectModelGe::Depends::line_style` property to set Dependency lines for Object Model diagrams to be a spline shape. You should set this property when you begin a project, as it takes effect going forward.

In addition, you can customize a line by adding/removing user-defined points to an arrow element.

To add points to an arrow element, follow these steps:

1. In the diagram, select the line you want to change.
2. Right-click and select **User Points** from the pop-up menu.
3. Depending on what you want to do:
 - ◆ To add a user-defined point, click **Add**. Note that the new point is added at the location where you accessed the pop-up menu.
 - ◆ To remove a point, click **Delete**. Rhapsody removes the point closest to the location where you accessed the pop-up menu.

Note that you can reshape the line when the cursor changes to the icon shown in the following figure:



Simply drag the line to reshape it.

Naming Boxes and Arrows

Use either of the following methods to name boxes and arrows:

- ◆ When you draw a box or a line, the cursor appears in the name field so you can edit it immediately. Type a name and press **Ctrl+Enter**. Note that names, except for transition labels, are limited to one line. To add an extra line to a transition label, press **Enter**.
- ◆ For transitions in statecharts and activity diagrams, you can select the **Name** tool. The cursor changes from an arrow to a pen. Click to select the name position, type a name, and click outside the element to finish.

To edit an existing name, do one of the following:

- ◆ Double-click the name to enable editing, and type a new name.
- ◆ Open the Features dialog box and edit the **Name** field on the **General** tab.

Note

If you rename a class box, you are renaming the class in the model.

Drawing Freestyle Shapes

The **Free Shapes** toolbar provides tools that enable you to customize your diagrams using freestyle shapes and graphic images. The toolbar includes the following tools:



Line draws a straight line between the selected endpoints.



Polyline draws a polyline using multiple points.



Polygon draws a polygon.



Rectangle draws a rectangle.



Polycurve draws a curve.



Closed Curve draws a closed, curved shape within the bounds of the specified shape.



Ellipse draws a circle or ellipse.

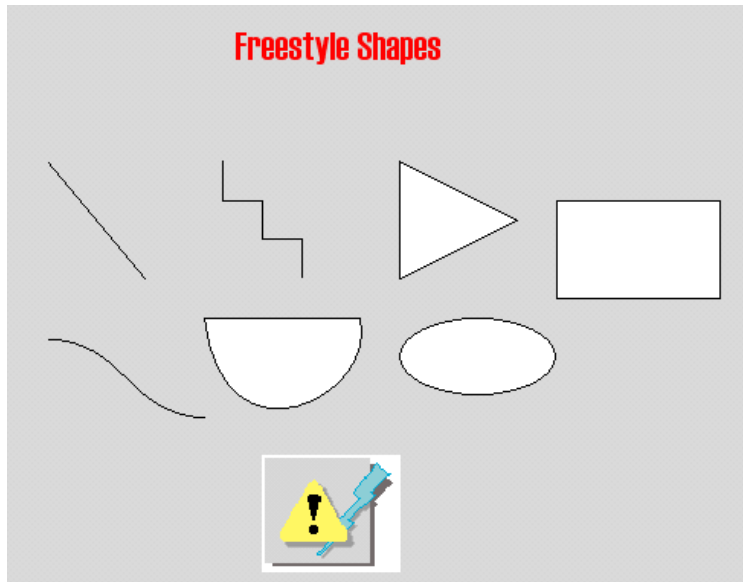


Text draws free text.



Image enables you to import an image into the diagram.

The following figure shows examples of each freestyle shape.



The following sections describe how to draw these freestyle shapes.

Drawing Lines and Polylines

To draw a simple line, follow these steps:

1. Click the **Line** tool.
2. Click in the diagram and drag the cursor away from the endpoint to create the line. The line is shown as a dashed line until you click to end the line.

If you click too early so only the square endpoint is displayed, click the endpoint and redraw the line.

To draw a polyline using several points, follow these steps:

1. Click the **Polyline** tool.
2. Click to place the first endpoint.

3. Drag the cursor away from the endpoint, clicking once to place each subsequent point in the polyline.
4. and once to place each point along the line.
5. Double-click to end the line.

Drawing Polygons

To draw a polygon, follow these steps:

1. Click the **Polygon** tool.
2. Click twice in the diagram to define two endpoints of a side, then drag and click to define the polygon.

For example, to draw a triangle, simply define one side and drag the cursor to form the triangle.

3. Click twice to complete the polygon.

Drawing Rectangles

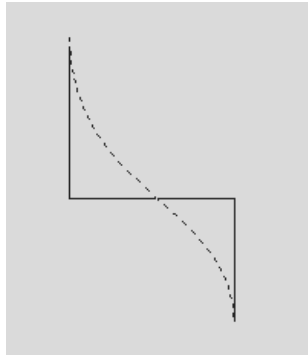
To draw a rectangle, follow these steps:

1. Click the **Rectangle** tool.
2. Click once in the diagram. By default, Rhapsody draws a square with the selection handles active.
3. Use the selection handles to create a rectangle.

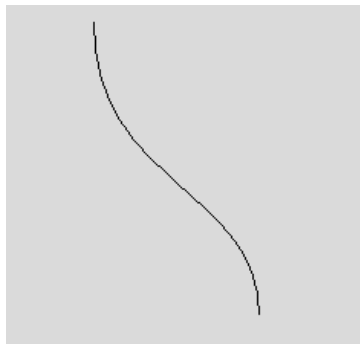
Drawing Polycurves and Closed Polycurves

To create a curve, follow these steps:

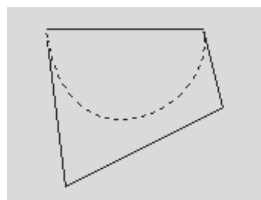
1. Click the **Polycurve** tool.
2. In the diagram, click to define several points that define the curve. As you define points (and, therefore, line segments), the resulting curve is drawn as a dashed line.



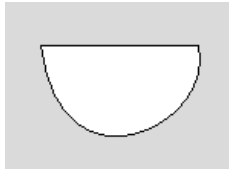
3. Double-click to create the curve.



Similarly, the **Closed Polycurve** tool creates a closed polycurve. As you define line segments, a dashed line shows the shape of the resulting closed curve.



Double-click to create the closed curve.



Drawing Ellipses and Circles

To draw an ellipse, follow these steps:

1. Click the **Ellipse** tool.
2. Click once in the diagram to place the left-most point of the ellipse.
3. Holding down the mouse button, drag the cursor to create the ellipse or circle.
4. Release the mouse button to complete the shape.

Drawing Text

To create floating text, follow these steps:

1. Click the **Text** tool.
2. Click once in the diagram to place the text box.
3. Type the desired text.
4. Press Enter for a new line; press Ctrl-Enter to place the text and dismiss the text box.

To change the text color, font, size, and so on, right-click the text and select Format from the pop-up menu. See [Changing the Format of a Single Element](#) for more information on changing text attributes.

Adding Images

To add an image to your diagram, follow these steps:

1. Click the **Image** tool. The Open dialog box is displayed.
2. Navigate to the directory that contains the image you want to add to the diagram.

Note

The Rhapsody distribution includes numerous bitmaps for common design elements, such as CDs, timeouts, displays, and so on. These images are available in `Share\PredefinedPictures` under the root installation directory.

3. Select the image to add.
4. Move the cursor to where you want to add the image, then click once to place it.

Deleting Freestyle Shapes

To delete a freestyle shape or graphic image, follow these steps:

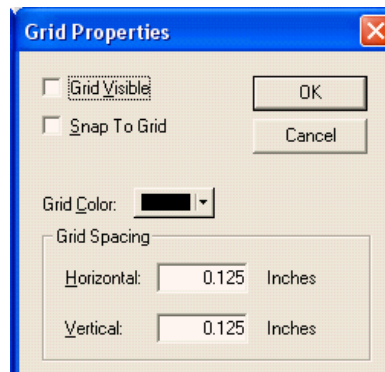
1. In the diagram, select the shape to delete.
2. Click the **Delete** tool.

Placing Elements Using the Grid

You can display a grid and rulers to assist with positioning elements in all the graphic editors except the sequence diagram editor.

To display the grid, click the **Grid** tool or select **Layout > Grid > Grid**.

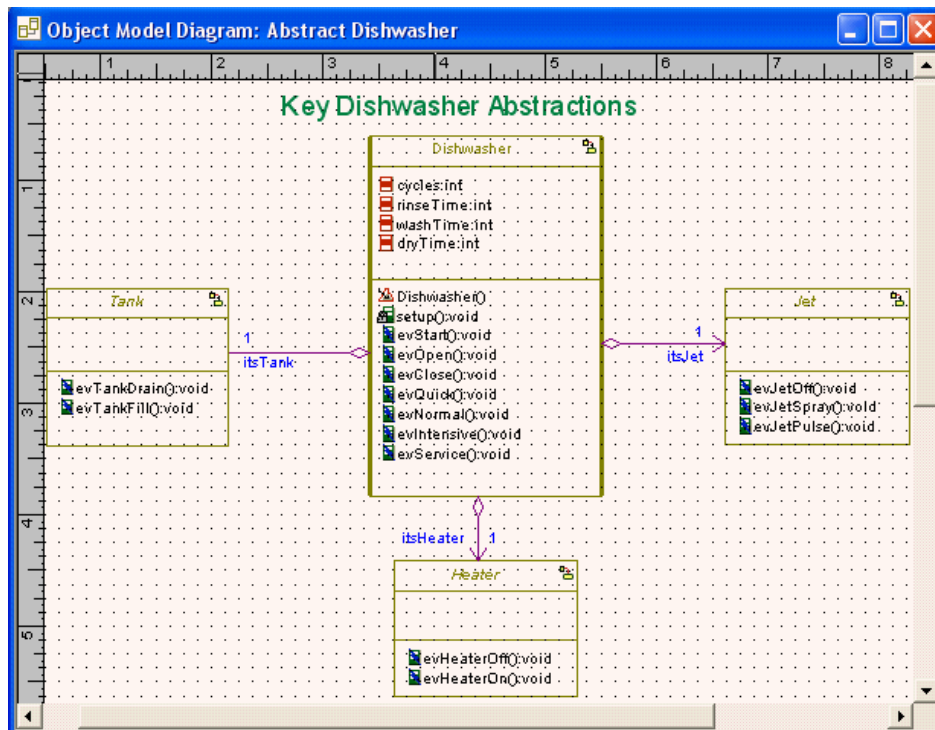
To set the attributes of the grid, select **Layout > Grid > Grid Properties**. The following figure shows the resultant dialog box.



You can set the following properties for the grid:

- ◆ **Grid Visible**—Specifies whether the grid is displayed.
- ◆ **Snap to Grid**—Specifies whether to automatically move an element you are drawing to the closest grid points. You can also set this by selecting **Layout > Grid > Snap to Grid**.
- ◆ **Grid Color**—Specifies the color used for the grid dots. The default color is black.
- ◆ **Grid Spacing**—Specifies the horizontal and vertical spacing of the grid points, in inches.

To display rulers in the drawing area, click the **Rulers** tool or select **Layout > Show Rulers**. The following figure shows an OMD with both the grid and rulers enabled.



Using Autoscroll

By default, Rhapsody automatically scrolls the diagram view while you are busy doing another operation (such as moving an existing box element or drawing new edges by dragging) that prevents you from doing the scrolling yourself. The autoscroll begins scrolling when the mouse pointer enters the autoscroll margins, which are virtual margins that define a virtual region around the drawing area (starting from the window frame and going *X* number of points into the drawing area).

You can change the size of the autoscroll margins by setting the property `General::Graphics::AutoScrollMargin`. This property defines the *X* number of points the margins enter into the drawing area. If you specify a large number for this property, the margin becomes bigger, thereby making the autoscroll more sensitive.

Set this property to 0 (no scroll region) to disable autoscrolling.

Selecting Elements

There are many ways Rhapsody enables you to select elements in diagrams. You can select an element using the mouse or using a variety of menu commands. Once you have selected an element, you can edit it depending on what kind of element it is.

Selecting Elements Using the Mouse

To select an element using the mouse, follow these steps:

1. Click the **Select** tool. When you move the mouse over the diagram, a standard mouse pointer is displayed.
2. Click an element. When you select an element, all other elements are deselected.
 - a. To select a line or an arrow, click anywhere on it.

You can select the control point of an arrow *only* by clicking and dragging. See [Clicking-and-Dragging](#) for more information.

- b. To select a box, click anywhere inside it or on its border.

Selecting Elements Using the Edit Menu

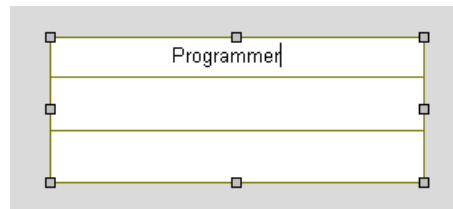
Use the **Edit > Select** option to access the following commands for making selections:

- ◆ **Select All**—Selects all elements in the diagram.
- ◆ **Select Next**—Selects the element next to the current one, when two elements are close together. This lets you easily navigate to each element in a diagram, one at a time.
- ◆ **Select by Area**—Enables you to draw a selection box around a group of elements within a container element (for example, classes in a package).
- ◆ **Select Same Type**—Selects all of the elements in the diagram that are of the same type as the element currently selected. If more than one type of element is selected, then all the elements of the different selected types will be selected.

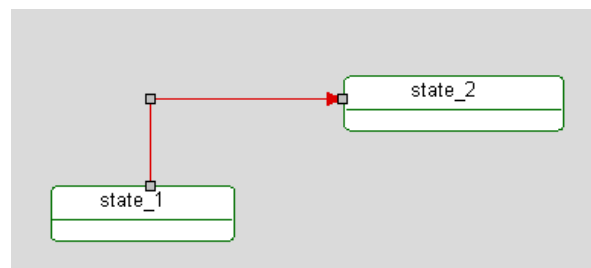
Selection Handles

When an element is selected, selection handles are displayed around its edges. The following diagrams selection handles on different elements.

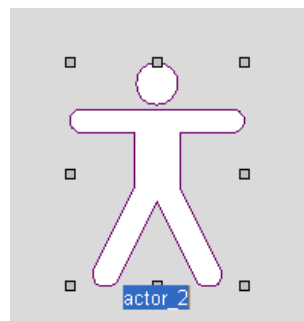
Boxes have markers on each corner and usually on each side.



Arrows and lines have a marker on each end and on each control point on the line.



Non-rectangular elements, such as use cases or actors, and groups of elements have an invisible bounding box with eight visible handles.



Note

The cursor changes depending on how the selected element will be changed. If the cursor is displayed as a four-pointed arrow, the selected element can be moved to a new location. If the cursor changes to a two-headed arrow, you can resize the element.

Selecting Multiple Elements

There are two ways to select more than one element:

- ◆ **Shift+Click**
- ◆ Clicking-and-dragging

Note that the last element selected in a multiple selection is distinguished by gray selection handles. Gray handles indicate that the element is an anchor component. Rhapsody uses anchor components as a reference for alignment operations. If you want to use a different component as the anchor, hold the **Ctrl** key down and click another element within the selection. Rhapsody transfers the gray handles from the previous element to the selected element

See [Arranging Elements](#) for more information.

Shift+Click

1. Click the first element you want to select.
2. Press and hold down the **Shift** key, then click the rest of the elements you want to select.

Note

To remove an element from the selection group, press and hold down the **Shift** key and click the element you want to remove from the selection.

Clicking-and-Dragging

To make a multiple selection using the click-and-drag method:

1. Move the mouse pointer to a blank area of the diagram.
2. Press and hold down the left mouse button.
3. Drag to surround the area that contains the elements you want to select.
4. To add more elements to the selection, hold down the Shift key while you click-and-drag again.

Editing Elements

You can edit elements using the following methods:

- ◆ **Features dialog box**—See [Using the Features Dialog Box](#) for detailed information.
- ◆ **Pop-up menu**—When you select an element in a diagram and right-click, a pop-up menu lists common operations you can perform on that element. Many of these options are element-specific, but some of the common operations are as follows:
 - **Features** or **Features in New Window**—Displays the Features dialog box for the specified element.
 - **Display Options**—Enables you to specify how elements are displayed in the diagram.
 - **Cut**—Removes the element from the view and saves it to the clipboard.
 - **Copy**—Saves a copy of the element to the clipboard and keeps it in the view.
 - **Copy with Model**—Copies an element such that when it is pasted into a diagram, a new element will be created in the model with the exact same characteristics as the original element.
 - **Remove from View**—Removes the specified element from the diagram but *not* from the model.
 - **Delete from Model**—Deletes the element from the model.
 - **Format**—Changes the format used to draw the element (color, line style, and so on). See [Changing the Format of a Single Element](#) for more information.
 - **Line Shape**—Enables you to change the shape of the line. See [Changing the Line Shape](#) for more information.
 - **User Points**—Enables you to add additional points to, or delete points from, a line element. This functionality enables you to customize the shape of the line. See [Changing the Line Shape](#) for more information. Element-specific options are described with the individual elements.
- ◆ **Manipulating the element in the diagram**—Use any of these methods to edit an element in a diagram:
 - Resize it.
 - Move its control points.
 - Move it to a new location.
 - Copy it.
 - Arrange it relative to one or more elements.
 - Remove it from the view.
 - Delete it from the model.
 - Edit any text associated with it.

Resizing Elements

You can resize an element by stretching its sides. You can resize only one element at a time.

To resize an element, follow these steps:

1. Select the element.
2. Move the mouse pointer over one of its selection handles. When the mouse pointer is over a selection handle, it changes to a double-pointed arrow. If you are editing a line, you can also move the mouse pointer over a line segment.
3. Click-and-drag the mouse until the element is the desired size and shape. If you want the element to maintain its proportions as you resize it, hold down the Shift key before you begin to click-and-drag.

If the box you are editing contains text, the text wraps to fit inside the boundaries of the new box. If you are editing a box that is connected to other lines or contains other elements, the lines and elements move and resize according to the box as you stretch it. If you hold down the Shift key, you can stretch diagonally while maintaining the element's scale.

To prevent elements from being resized when you resize their parent, press and hold the Alt key while you click-and-drag with the mouse. Alternatively, you can select the menu item **Edit > Move/Stretch Without Contained** before resizing the element.

To make Rhapsody automatically enlarge the text box of a box element to fit the size of an element name, select the **Expand to fit text** option in the pop-up menu. Alternatively, select **Layout > Expand to fit text**. Note that once you apply this feature to an element, it remains enabled until you resize the element.

To use this functionality as the default behavior, set the property `General::Graphics::FitBoxToItsTextuals` property to Checked.

Moving Control Points

If you have placed control points on arrows, you can select a control point and move it individually to change the curve of the line. This is particularly effective within statecharts where transitions are rendered as spline curves.

See the description of the **Reroute** command in [Changing the Line Shape](#) for information on removing extra control points.

Moving Elements

You can move an element by clicking on it and dragging it to a new location. You can move several elements simultaneously.

To move an element, follow these steps:

1. Select the element you want to move.
2. Move the mouse pointer over the element. The cursor changes to a four-pointed arrow, which denotes a move operation.
3. Click-and-drag the element to a new location.

Note the following:

- ◆ To prevent elements from being moved when you move their parent boxes, hold down the Alt key while you click-and-drag with the mouse. Alternatively, you can select the menu item **Edit > Move/Stretch Without Contained** before moving the element.
- ◆ If you hold down the Ctrl key when moving an element, a copy of the selected element is created.
- ◆ If you hold the Shift key down when moving an element, you can move it only horizontally or vertically.

Maintaining Line Shape when Moving or Stretching Elements

When elements are stretched or moved, Rhapsody maintains the relationship between lines and boxes as much as possible to preserve diagram layout.

When moving or stretching more than one element at the same time, lines maintain their ratio to the other elements. That is, the line stretches and moves along with the other elements. This can be problematic for straight lines, which do not remain straight.

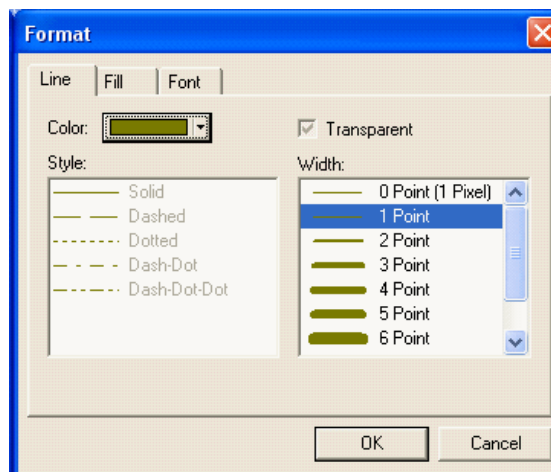
To maintain straight lines when moving or stretching boxes, move the boxes one at a time.

When moving or stretching only one box, lines that connect with vertical boundaries of the box maintain their Y-coordinate, and lines that connect with horizontal boundaries of the box maintain their X-coordinate. The coordinates change only when the box is moved to the extent that maintaining those coordinates is impossible. This way, lines that are straight remain straight whenever possible.

Changing the Format of a Single Element

To edit the format of a single element in a diagram, right-click the element and select **Format** or select **Edit > Format > Change**. Alternatively, you can use the tools in the **Format** toolbar. See [Using the Format Toolbar](#) for more information.

The Format dialog box lists the current line, fill, and font information for the selected element. The following figure shows the default line attributes for a class.

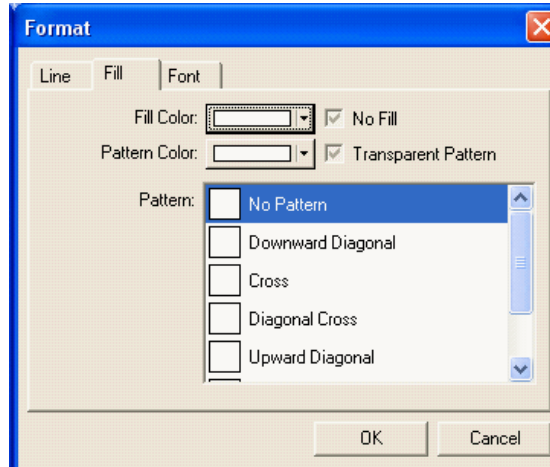


Use the **Line** tab of the dialog box to change the line color, style, or width or the lines used to draw the element.

Note

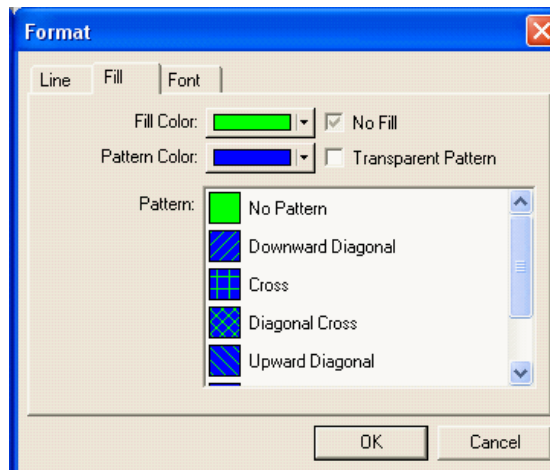
Line widths 0 and 1 are the same; however, you can specify a new line type (dashed, dotted, and so on) only for line width 0. Otherwise, the style options will remain grayed out (unavailable).

The **Fill** tab enables you to specify new fill attributes for the specified element, including the fill and pattern colors, and the pattern style. The following figure shows the **Fill** tab for a class.

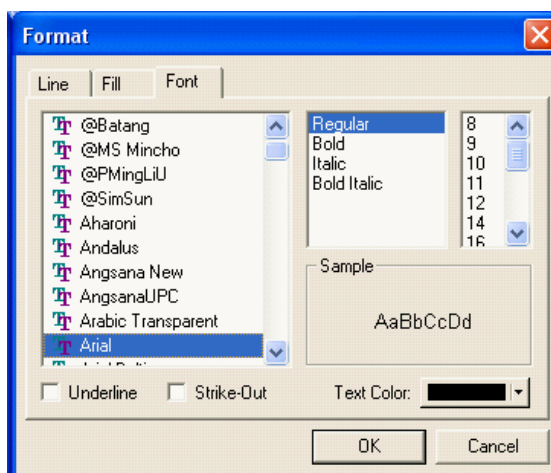


A preview of the pattern is displayed in the small box beside each pattern name.

For example, the following figure shows the preview of a fill color of green, a pattern color of blue, with the Transparent pattern option disabled.



The **Font** tab enables you to specify new font attributes for the specified element, including the font size and type, color, and whether the text should be in italics or bold, or underlined. The following figure shows the **Font** tab for a class.



Note

These settings apply only to the specified element. Therefore, if you change the attributes for an individual class, any new classes will use the default attributes. See [Changing the Format of a Single Element](#) for information on changing the attributes for a specific type of element (for example, all classes).

Using the Format Toolbar

To edit the format of a single element in a diagram, right-click the element and select one of the following icons from the toolbar:



Font Color specifies the color to use for the text or label of the selected element. For example, if you select a state and use this tool to change the color to red, the name of the selected state is displayed in red.

This tool performs the same action as right-clicking an element, selecting **Format** from the pop-up menu, and using the appropriate tab.



Line Color specifies the color to use for the selected line element. For example, if you select a state and use this tool to change the line color to blue, the box for the state will be displayed in blue.


This tool performs the same action as right-clicking an element, selecting **Format** from the pop-up menu, and using the appropriate tab.



Fill Color specifies the color to use as fill color for the selected element. For example, if you select a state and use this tool to change the color to yellow, the selected state will be filled with yellow.

This tool performs the same action as right-clicking an element, selecting **Format** from the pop-up menu, and using the appropriate tab.

Copying Formatting from one Element to Another

Rhapsody provides a Format Painter icon  to copy formatting from one element to another element in the same diagram with these steps:

1. In the diagram, click the element whose formatting you want to copy.
2. Click the Format Painter button in the **Standard** toolbar.
3. Click the element to which you would like to apply the copied formatting.

To copy formatting from one element to a group of other elements without having to click the Format Painter button each time:

1. In the diagram, click the element whose formatting you want to copy.
2. Click the Stamp Mode button in the **Layout** toolbar.
3. Click the Format Painter button
4. Click the elements to which you would like to apply the copied formatting.

Note

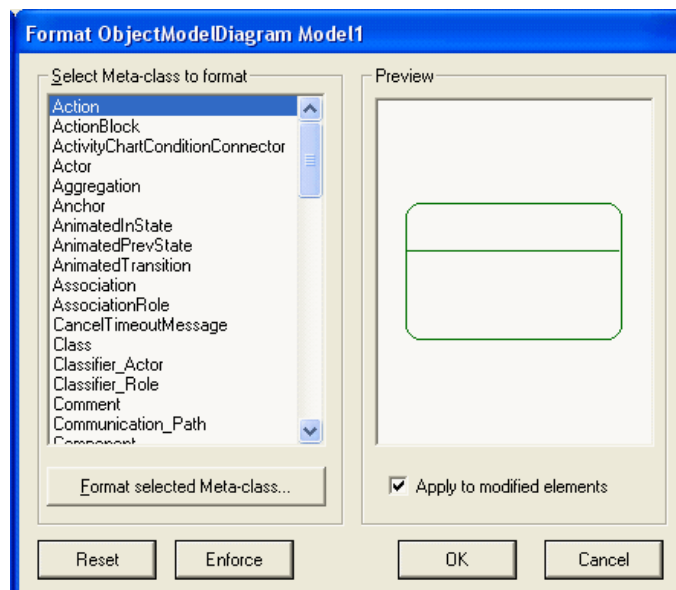
The Stamp Mode button is a toggle button. Rhapsody will remain in “stamp mode” until you click the button a second time.

Changing the Format of a Metaclass

In addition to changing the format of an individual element, you can change the format of an entire metaclass. For example, you can specify styles for all classes, all actors, all associations, and so on.

To change the default settings for an entire metaclass, follow these steps:

1. Right-click in the diagram and select **Format** from the pop-up menu. The Format dialog box opens, as shown in the following figure.



Note: The check box **Apply to modified elements** is displayed for diagrams only; for projects, packages, and stereotypes, the check box is called **Apply to sub elements**.

2. In the left pane, select the metaclass whose attributes you want to change, then click **Format selected meta-class**.

Note: The left pane will display only the metaclasses that are relevant for the type of diagram.

3. Use the Format properties dialog box to select the new line, fill, and font attributes for the metaclass (see [Changing the Format of a Single Element](#) for detailed information). The Preview pane displays how the element will look in the diagram.
4. Click one of the available buttons:
 - a. **Cancel**—Discards all of your changes.
 - b. **OK**—Saves your changes to the specified metaclasses.

If **Apply to modified elements** is enabled when you click **OK**, all the existing elements in the selected scope (of the given metaclass) are changed to the specified format (in addition to any elements that are created later). If **Apply to modified elements** is not enabled, existing elements of the specified metaclass that have individual overrides are not changed—but new elements will use the new style by default.

For example, consider the case where all actors have white fill by default, but actor A has blue fill with yellow stripes. If you change the default fill color for all actors to be green, and **Apply to modified elements** is enabled, all the actors will have white fill. However, if you disable this check box, all the actors will have white fill—except for actor A, which will keep the blue fill and yellow stripes. All subsequently created actors will use white fill.

- c. **Reset**—Removes all user-specified formats for the specified element and “rolls back” to the default values.

If **Apply to modified elements** is enabled when you click **Reset**, Rhapsody displays a confirmation dialog box that asks whether you want to reset the default values for *all* of the modified subelements in the specified diagram or project. Click **Yes** to remove all overrides; otherwise, click **No** and disable this check box to reset specific metaclass styles.

- d. **Enforce**—Forces a subelement to use the locally modified style. Note that this affects only the styles that were explicitly specified; other formats remain unchanged. It is similar to the behavior specified in [Changing the Style Scope](#).

If **Apply to modified elements** is enabled when you click **Enforce**, Rhapsody displays a confirmation dialog box that asks whether you want to force the local style on *all* the elements in the specified diagram. Click **Yes** to remove all overrides; otherwise, click **No** and disable this check box to reset specific metaclass styles. Note that if you apply **Enforce** without enabling the **Apply** check box, nothing is changed.

5. Click **OK** to dismiss the Format properties dialog box.

Use the option **Edit > Format > Un-override** to remove an overrides on boxes and line elements in diagrams.

Changing the Style Scope

Where you invoke the Format dialog box affects the scope of the formatting style:

- ◆ If you invoke the Format dialog box from within a diagram, the format change applies only to the metaclass in the *current* diagram.

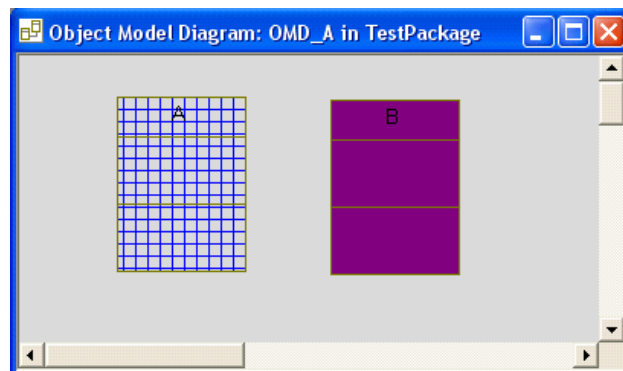
For example, if you change the format of states in the Tester statechart so they are filled with yellow, states in other statecharts will *not* be filled with yellow automatically.

- ◆ If you invoke the Format dialog box at the project level (select the project node in the browser, right-click, and select **Format** from the pop-up menu), the specified style is applied to that metaclass throughout the *entire model*.
- ◆ If you invoke the Format dialog box by selecting an individual element in a diagram, the specified style will be applied to that element only.

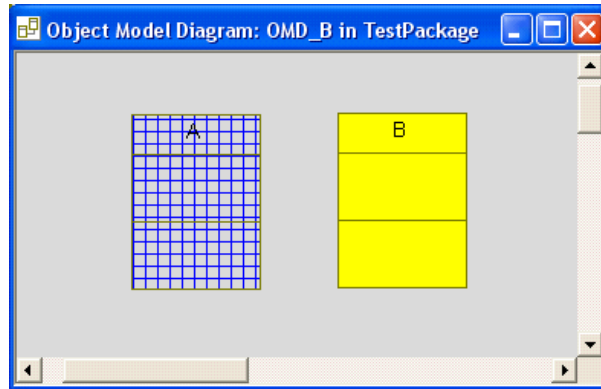
If you apply a style to an individual element and then copy it to a new diagram, it brings its style with it. Consider the following scenario:

- ◆ OMD_A uses purple fill for classes.
- ◆ Class A in OMD_A has the individual fill style of blue cross-hatching.
- ◆ Class B in OMD_A uses the default style for classes (purple fill).
- ◆ OMD_B uses yellow fill for classes.

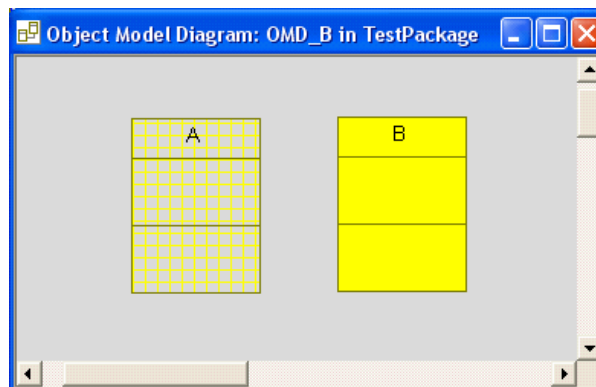
The following figure shows OMD_A.



If you copy classes A and B into OMD_B, class A keeps its individual style, but class B now uses the default local style (yellow fill). The following figure shows OMD_B.



To force the class to use the local style, click **Enforce**. In this example, class A is now forced to use yellow as its fill color. Note that the cross-hatching is still used (because there is no setting for cross-hatching in this OMD).



Making an Element's Formatting the Default Formatting

After you have applied formatting to a diagram element, you can make the element's formatting the default formatting for new elements of this type.

To make an element's formatting the default formatting for elements of that type:

1. Select the element in the diagram.
2. From the element's context menu, select **Make Default**. The Make Default dialog is displayed.
3. Select the characteristics to set as default. The available options are format, display options, and size.
4. Select the level at which you would like to set the defaults, for example, diagram level or package level.

Note

This option sets the default formatting for all new elements of the same type. For elements that already exist in the diagram, the default formatting will be applied unless the elements have been overridden. (This applies only to the formatting; the size of existing elements will not be changed, nor will the display options.)

Copying an Element

There are two different ways in which elements can be copied and pasted in a diagram:

- ◆ **Simple Copy**—another representation of the element is created on the diagram canvas.
- ◆ **Copy with Model**—a new element is created in the model and pasted into the diagram. The new model has the exact same characteristics as the original element.

Simple Copy

You can copy an element in one of three ways:

- ◆ Use the **Copy** and **Paste** commands in the Edit menu.
- ◆ Use the **Ctrl+Drag** method.
- ◆ Use the **Replicate** command in the Layout menu. Note that this applies to statecharts only.

To copy an element using the **Ctrl+Drag** method, follow these steps:

1. Select the element you want to copy.
2. Move the mouse pointer over the element.

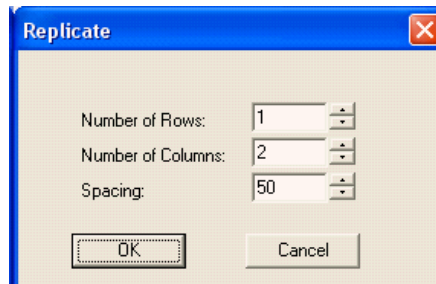
3. Press Ctrl. A small box with a plus sign in it is displayed below and to the right of the pointer.
4. Click-and-drag the element. When you release the mouse button, a copy of the element appears in the new location.

Note

In statecharts, copying creates new elements. In OMDs and UCDs, these methods copy graphic elements but do not create new elements in the model.

To copy using the **Replicate** command, follow these steps:

1. Select the element you want to copy.
2. Choose **Layout > Replicate**. The Replicate dialog is displayed, as shown in the following figure.



3. Enter the number of rows and columns you want the copied elements to use, and the spacing between them.
4. Click **OK**. Rhapsody displays the replicated elements in the diagram.

Copy with Model

This refers to the ability to copy a diagram element such that when it is pasted into a diagram, an entirely new element is created, with all of the characteristics of the original element. For example, if you use this option to copy and paste a class, a new class will be created in the model and it will contain the same attributes and operations as the original class, the same associated diagrams (such as a statechart), etc.

To create a new model element based on an existing diagram element:

1. Select the element to be copied.
2. From the main menu, select **Edit > Copy with Model**.
3. Navigate to the diagram where you would like to paste the new object.
4. From the main menu, select **Paste**. The new element will appear in the diagram as well as in the browser.
5. Rename the new item if desired. The default name will be the name of the original element with the string “_copy” appended to it.

Arranging Elements

In addition to the grid and ruler tools (described in [Placing Elements Using the Grid](#)), the **Layout** toolbar includes several tools that enable you to align elements in the drawing area.

To arrange elements, follow these steps:

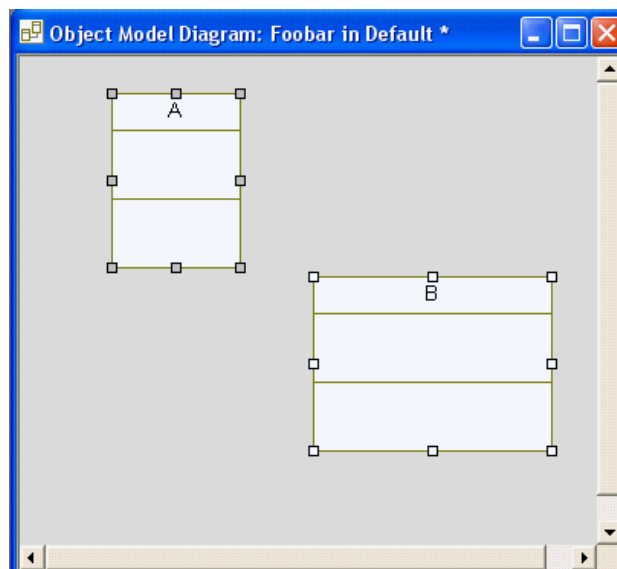
1. Select **View > Toolbars > Layout**. The **Layout** toolbar is displayed.

Note: You can dock the toolbar by clicking-and-dragging it to a window edge.

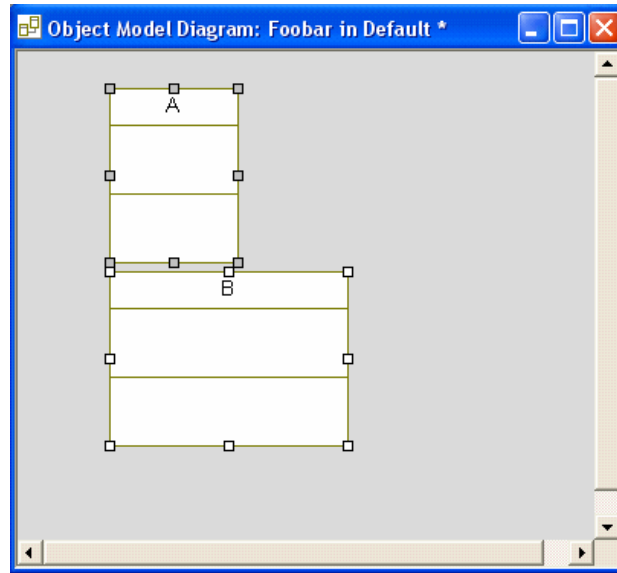
2. The tools that enable you to arrange elements are grayed out (unavailable) until you select an elements in the drawing area. In the diagram, select two or elements to arrange.
3. Select one of the layout tools. The elements are arranged according to the layout selected.

Note that the selection handles use different colors to show which element is used for the default alignment and sizing (the last element selected).

Consider the two classes in the following figure.



The selection handles on class A are gray, which denotes that Rhapsody will use the values of class A for any alignments and sizing. Therefore, if you align the left sides of class A and B, class A stays where it is and class B moves under it, as shown in the following figure.



The following alignment tools are available:



Align Top aligns the selected elements to the top of the element with the gray selection handles. This is equivalent to selecting **Layout > Align > Top**.



Align Middle aligns the selected elements to the middle (along the horizontal) of the element with the gray selection handles. This is equivalent to selecting **Layout > Align > Middle**.



Align Bottom aligns the selected elements to the bottom of the element with the gray selection handles. This is equivalent to selecting **Layout > Align > Bottom**.



Align Left aligns the selected elements to the left side of the element with the gray selection handles. This is equivalent to selecting **Layout > Align > Left**.



Align Center aligns the elements so they are aligned to the center, vertical line of the element with the gray selection handles. This is equivalent to selecting **Layout > Align > Center**.



Align Right aligns the selected elements to the right side of the element with the gray selection handles. This is equivalent to selecting **Layout > Align > Right**.



Same width resizes all the selected elements so they are the same width as the element with the gray selection handles. This is equivalent to selecting **Layout > Make Same Size > Same Width**.



Same height resizes all the selected elements so they are the same height as the element with the gray selection boxes. This is equivalent to selecting **Layout > Make Same Size > Same Height**.



Same size resizes all the selected elements so they are the same size as the element with the gray selection boxes. This is equivalent to selecting **Layout > Make Same Size > Same Size**.



Space across spaces the selected elements so they are equidistant (across) from the element with the gray selection handles. This is equivalent to selecting **Layout > Space Evenly > Space Across**.



Space down spaces the selected elements so they are equidistant (down) from the element with the gray selection handles. This is equivalent to selecting **Layout > Space Evenly > Space Down**.

Removing an Element from the View

To remove an element from the view but not from the model, follow these steps:

1. Select the element to be removed.
2. Press the Delete key. The element is removed from the view.

Alternatively, right-click the element in the diagram and select **Remove from View** from the pop-up menu.

Deleting an Element from the Model

To delete an element from both the view and the model, follow these steps:

1. Select the element to be deleted.
2. Click the **Delete** tool or press **Ctrl+Delete**. The element is deleted from both the model and the view.

Alternatively, right-click the element in the diagram and select **Delete from Model** from the pop-up menu.

Editing Text

To edit text, follow these steps:

1. Double-click any text to highlight it.
2. Edit the selection.
3. To add another line of text, press **Enter**; press **Ctrl+Enter** to end the edit.

Note that both the right mouse button and the Esc key cancel the edit.

Alternatively, for some features you can right-click the element in the drawing area and select **Edit Text** from the pop-up menu.

Displaying Compartments

One of the display options available for diagram elements is the display of contained items in visual compartments. This option is available for the following:

- ◆ Classes
- ◆ Objects
- ◆ Files

For the above elements, the following items can be displayed in compartments where applicable:

- ◆ Constraints
- ◆ Tags (both local and on the element's stereotype)
- ◆ Ports
- ◆ Parts

Selecting Items to Display

To select the items that should be displayed, follow these steps:

1. In the **Display Options** dialog box, click **Compartments**.
2. In the dialog box that is displayed, use the arrows to select the items that should be displayed, and the order in which they should be displayed.
3. Click **OK**.

In each compartment, individual items are displayed with an icon indicating the type of sub-element. If the text is too long to display, an ellipsis is displayed. When this ellipsis is clicked, you can view/edit the full text.

Note

For constraints, the content of the specification field is displayed.

The name of items can be edited in the compartments but items cannot be added or deleted.

It is not possible to modify the order in which individual items are displayed within each compartment.

User-defined terms are displayed in the same compartment as the item on which they are based, however, the icon indicates that this is a user-defined term.

Note

While this feature allows you to display/hide attributes and operations, it does not replace the attribute and operation tabs, which allow more precise display options, such as the display of only a subset of defined attributes and operations.

Displaying Stereotype of Items in List

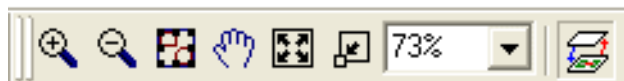
For elements listed in compartments, Rhapsody provides the option of displaying the name of stereotypes applied to the element alongside the name of the element.

To display elements' stereotypes in the compartment list, use these guidelines:

- ◆ At the diagram level or higher, modify the property `General::Graphics::ShowStereotypes`. The possible values for this property are `No`, `Prefix`, `Suffix`.
- ◆ The default setting for this property is `Prefix`.
- ◆ This property applies to all of the compartments that can be displayed.

Zooming

There are several zoom tools available on the **Zoom** toolbar, shown in the following figure.



Alternatively, you can use the **View > Zoom/Pan** menu, or use the zoom options available in the pop-up menu in the drawing area.

Note the following:

- ◆ To prevent elements from being resized when you resize their parent (for example, classes contained in a composite class), press and hold the **Alt** key while you click-and-drag with the mouse.
- ◆ If at any time the screen becomes difficult to read, you can refresh it by pressing **F5** or selecting **View > Refresh**.

Zooming In and Zooming Out

To zoom in or out on a diagram, follow these steps:

1. Click the **Zoom In** or **Zoom Out** button.
2. Move the cursor over the diagram. The cursor appears as a magnifying glass with either a plus or minus sign in it.
3. Click the diagram to enlarge or shrink it by 25%, depending on which tool you selected.

There are two ways to zoom in on a portion of a diagram:

- ◆ Click the **Zoom In** button, then hold down the left mouse button to draw a selection box around the part of the diagram you want to zoom in on.
- ◆ Select an element in the diagram, then click **Zoom to Selection** to enlarge the selected element so it takes up the entire drawing area.

Note

You remain in zoom mode until you select another tool from the toolbar.

Scaling a Diagram

To scale a diagram to a certain percentage, use the drop-down scale box. You can set the diagram scaling to a value between 10% and 500%.

Alternatively, select **View > Zoom/Pan**, then select the percentage used to scale the diagram.

To scale the diagram so the entire diagram is visible in the current window, click the **Zoom to Fit** button. When you click the button, the diagram is resized to fit in the current window. This button performs the same command as **View > Scale to Fit**.

Panning a Diagram

Click the **Pan** button to move the diagram in the drawing area so you can see portions of the diagram that do not fit in the current viewing area.

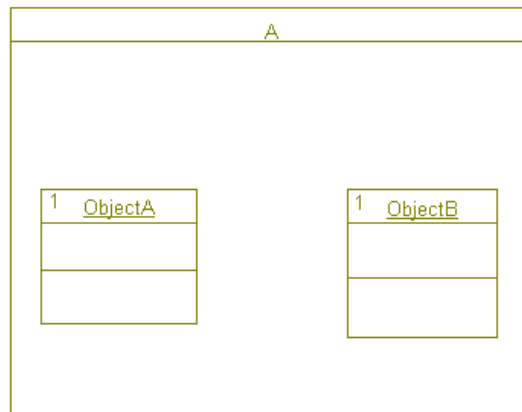
Undoing a Zoom

To undo the last zoom, click the **Undo Zoom** button, or select **View > Zoom > Undo Zoom**.

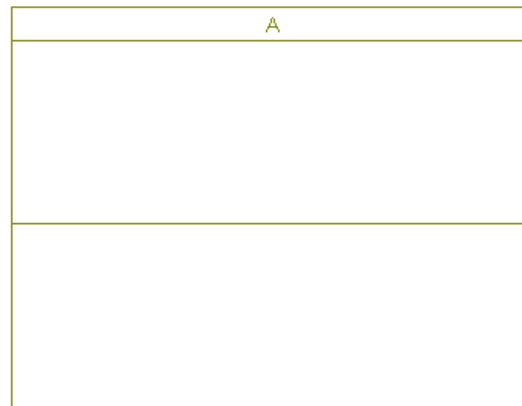
Specifying the Specification or Structured View

To show the specification or structured view of a diagram, click the **Specification/Structured View** button.

For example, suppose you have a structured (composite) class A that contains the parts ObjectA and ObjectB. The structured view looks like the following figure.



The specification view looks like the following figure.



In addition to toggling between the two views, any new classes or objects created with the selected mode will use that mode.

Note that if there is a mix of structured and specification elements, the button is grayed out.


Using the Bird's Eye (Diagram Navigator)

The Bird's Eye (Diagram Navigator) provides a bird's eye view of the diagram that is currently being viewed. This can be very useful when dealing with very large diagrams, allowing you to view specific areas of the diagram in the drawing area, while, at the same time, maintaining a view of the diagram in its entirety.

The Bird's Eye contains a depiction of the entire diagram being viewed, and a rectangle viewport that indicates which portion of the diagram is currently visible in the drawing area.

Showing/Hiding the Bird's Eye Window

To show/hide the Bird's Eye window, do one of the following:

- ◆ Select **View > Bird's Eye** from the menu bar.
- ◆ Click the Bird's Eye  icon in the **Windows** toolbar.
- ◆ Use the keyboard shortcut, **Alt+5**.
- ◆ Right-click the diagram in the drawing area, and select **Bird's Eye** from the context menu.

Navigating to a Specific Area of a Diagram

To use the Bird's Eye to move to a specific area of a diagram, do one of the following:

- ◆ Drag the viewport to the area you would like to view.
- ◆ Click and draw a “new” viewport in the Bird's Eye window over the area you would like to view.

Using the Bird's Eye to Enlarge/Shrink the Visible Area

To use the Bird's Eye to enlarge/shrink the visible area of the diagram, drag an edge or corner of the viewport to enlarge/shrink the viewport.

Enlarging the viewport has the same effect as zooming out in the drawing area. Shrinking the viewport has the same effect as zooming in the drawing area.

Note

When you drag an edge of the viewport, the viewport size will always change in both dimensions, maintaining the height/width ratio.

By default, the viewport will grow in the direction of the edge selected. If you hold down

Ctrl while dragging, however, the viewport will grow in the direction of the opposite edge as well in order to maintain the current center point of the diagram.

Scrolling/Zooming in Drawing Area

If the scroll bars are used to change the visible area of the diagram in the drawing area, the viewport will move accordingly in the Bird's Eye window.

If the zoom level is changed in the drawing area, the size of the viewport will change accordingly in the Bird's Eye window.

Changing the Appearance of the Viewport

You can modify the appearance of the viewport rectangle in the Bird's Eye window. Display characteristics that can be modified include line color, line style, line width, fill color, and fill pattern.

To display the viewport appearance dialog: Right-click anywhere in the Bird's Eye window.

General Characteristics of the Bird's Eye Window

- ◆ The Bird's Eye window is used only for changing the viewable area of a diagram. You cannot modify any diagram elements in the Bird's Eye window.
- ◆ The size and position of the Bird's Eye window is saved in the Rhapsody workspace.
- ◆ The Bird's Eye window can be resized, and it can float or be docked. The docking-related options are accessed by right-clicking the borders of the window (but not the title bar).
- ◆ The black dotted line in the Bird's Eye window represents the diagram's drawing canvas. When the canvas size is changed in the drawing area, this dotted line changes accordingly. Depending on the size of the Bird's Eye window, there may be some whitespace to the right of and below the dotted line.

Completing Relations

The Complete Relations option in the Layout menu completes relation lines in diagrams. For example, you can define relations in the browser, draw the participating classes in an OMD, then select **Complete Relations** to speed up the drawing of the relation lines.

- ◆ **All**—Completes all the relation lines
- ◆ **Selected to All**—Completes relation lines only for relations originating in the selected classes
- ◆ **Among Selected**—Completes relation lines only for relations existing between the selected classes

Using IntelliVisor

The IntelliVisor feature offers intelligent suggestions based on what you are doing to reduce:

- ◆ The number of steps required to complete a task
- ◆ The amount of time spent changing between different views (browser, graphics editor, code editor, and so on)—giving more time to actually complete the task
- ◆ The time spent in the “build and run” cycle because of fewer compilation problems resulting from wrong type usage, misspellings, and so on

The suggestions offered by IntelliVisor depend on the current context. The *context* is a model element (usually a class or a package) in which the IntelliVisor is activated. The IntelliVisor contents is defined by the scope of the context. For example, the context can be class `MyClass`; the list contents will be all the methods, attributes, and relations, including superclasses and interfaces implemented by `MyClass`, and all the global methods defined in the package containing the class.

The property `IntelliVisor::General::ActivateOnGe` specifies whether to enable IntelliVisor in the Rhapsody graphics editors. By default, IntelliVisor is enabled.

Activating IntelliVisor

When you press **Ctrl+Space** in the graphic editor, Rhapsody displays a list box with information from which to choose. You can navigate in this list box using either the arrow keys or the mouse. When you select an item from the list of suggestions and press **Enter**, that text is placed in the text box.

To dismiss the IntelliVisor list box, do one of the following:

- ◆ Press one of the following keys:
 - **Esc**
 - **Enter**
- ◆ Double-click the mouse button.
- ◆ Change the window focus.
- ◆ Press **space/Alt**.
- ◆ Click outside of the text box.

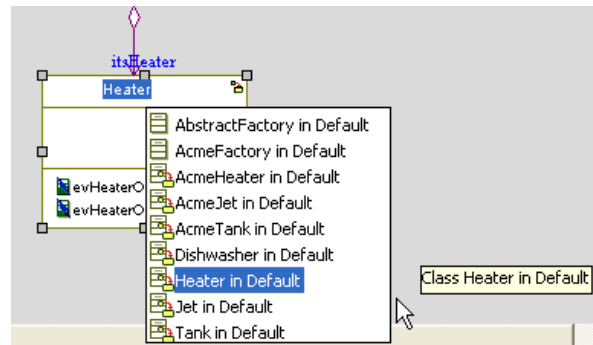
IntelliVisor Information

When you are using a graphics editor, IntelliVisor can simplify your tasks by offering shortcuts to similar model elements.

For example, if you are drawing a class in an OMD or structure diagram and invoke IntelliVisor, the list box contains the default name of the new class and all the classes that already exist in the model. If you highlight one of the classes in the list, IntelliVisor displays summary information available for that element, including:

- ◆ The element's type, name, and parent information
- ◆ The element's stereotype, if it exists
- ◆ The first few lines of the element's description, if it exists

The following figure shows information displayed by IntelliVisor.

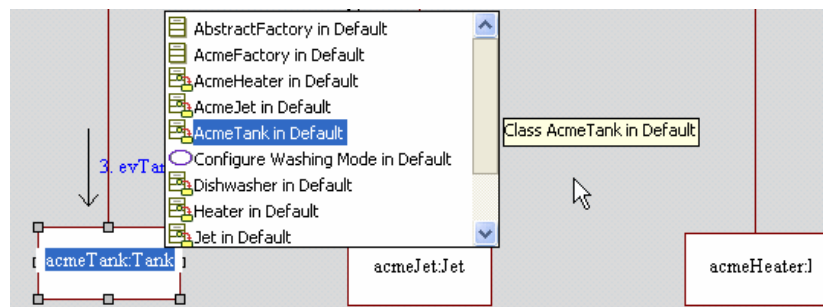


To replace the new class with an existing class, simply highlight the class in the list box and double left-click. IntelliVisor replaces the new class with the specified class.

In addition to classes, IntelliVisor can be invoked in OMDs when you are drawing actors and packages.

Collaboration Diagrams

If you invoke IntelliVisor on an object or multi-object, the list box contains all the classes in the current project. For example:

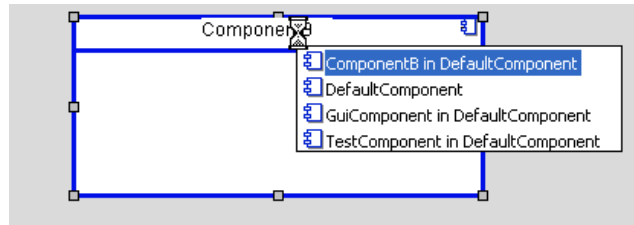


If you apply a selection from the list, IntelliVisor replaces the part after the role name. For example, `AcmeTank : Tank` will become `AcmeTank : Jet` if you select the `Jet` class in the list box.

In addition to classes, IntelliVisor can be invoked in collaboration diagrams when you are drawing actors.

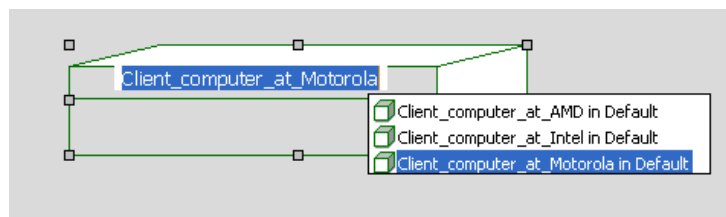
Component Diagrams

If you invoke IntelliVisor for a component in a component diagram, the list box contains all the components in the model. For example:



Deployment Diagrams

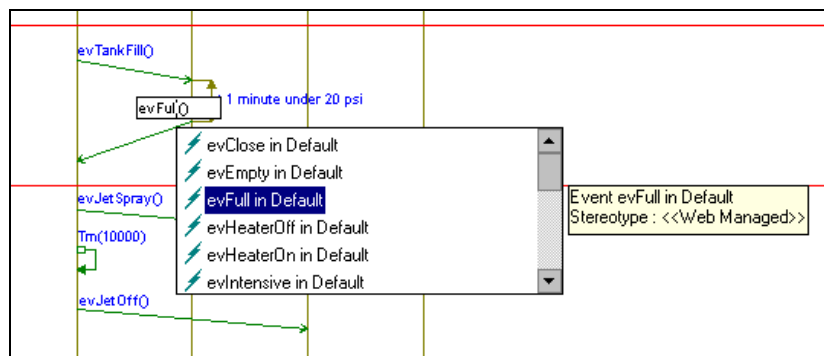
If you invoke IntelliVisor on a node in a deployment diagram, the list box contains all the nodes defined in the model. For example:



Note that you cannot activate component instances in IntelliVisor.

Sequence Diagrams

If you invoke IntelliVisor within a sequence diagram, the list box contains the events, operations, and triggered operations consumed by the target class. If there are base classes that consume events, operations, and triggered operations, they are included in the list.

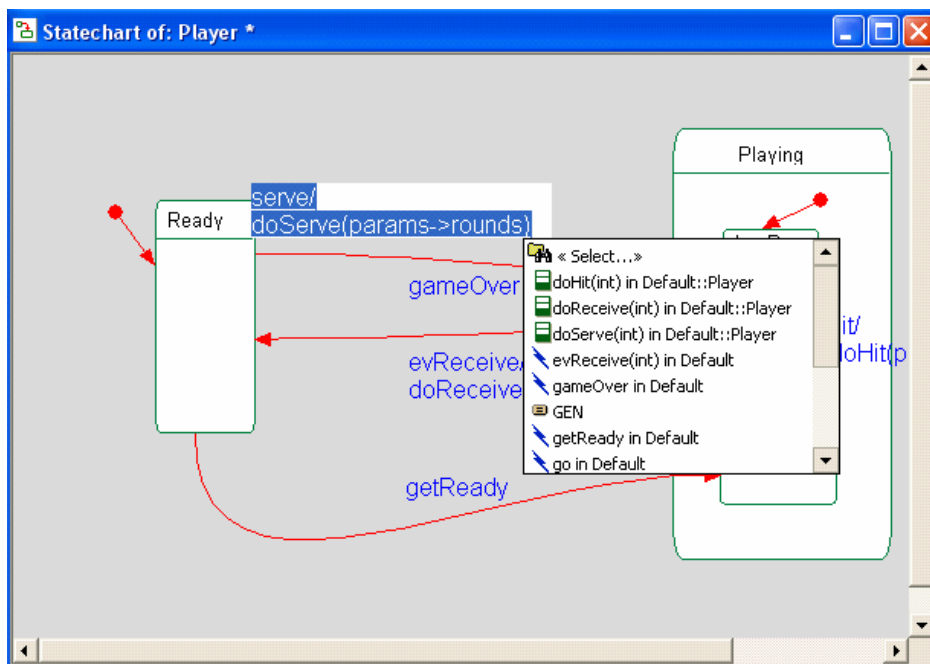


Note the following:

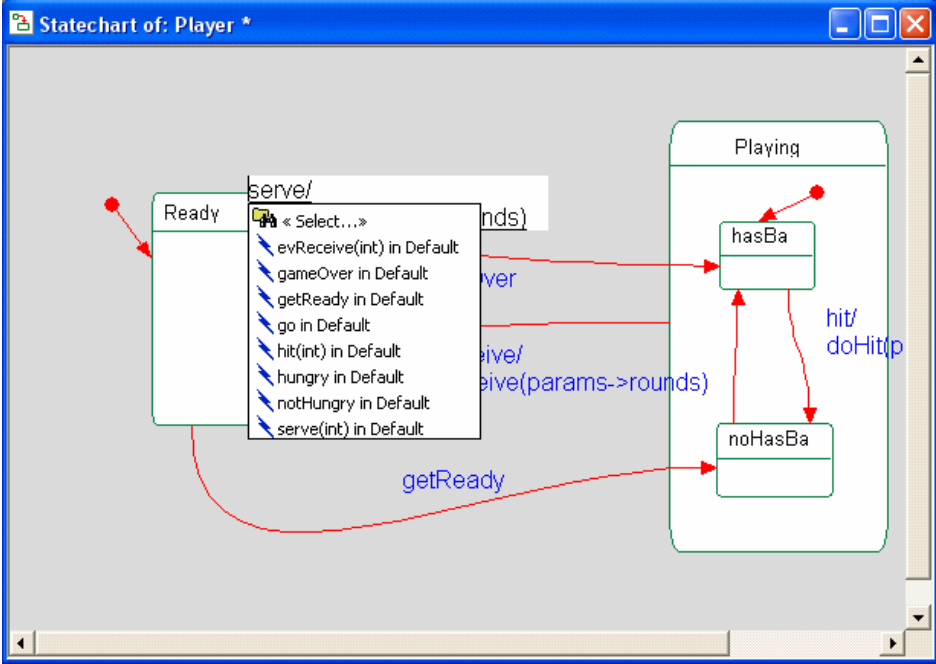
- ◆ If you select a constructor line, IntelliVisor displays the list of constructors.
- ◆ The instance line should be associated with a part class, or the IntelliVisor list box will be empty.

Statecharts and Activity Diagrams

If you invoke IntelliVisor inside a state transition trigger in a statechart or activity diagram before the “/” or “[” symbol, the list box contains all the events that can be consumed by the class, as in this example.



If you invoke IntelliVisor after the “/” or “[” symbol, the list box contains the default class content. For example:

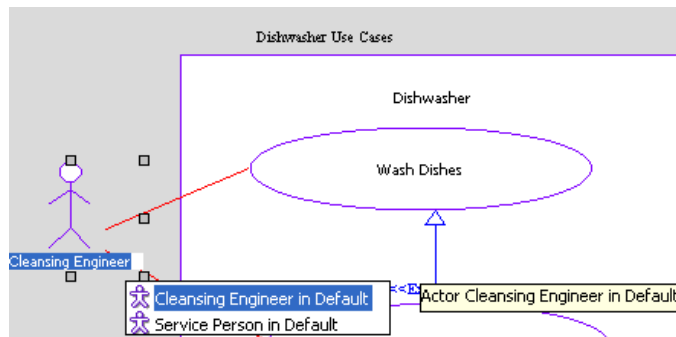


In addition, you can invoke IntelliVisor in activity diagrams to help you perform the following tasks:

- ◆ Write initialization code for actions.
- ◆ Edit the code for an activity.

Use Case Diagrams

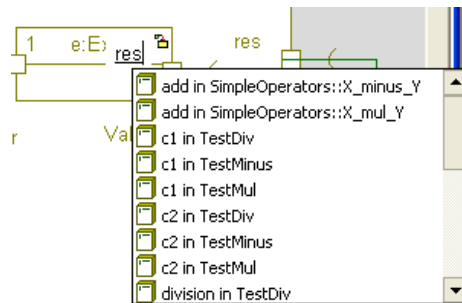
If you apply IntelliVisor on an actor, the list box contains all the actors defined in the project.



Similarly, if you invoke IntelliVisor on a use case, the list box contains all the use cases defined in the project.

Structure Diagrams

If you invoke IntelliVisor for an object in a structure diagram, the list box contains all the objects defined in the project. For example:



Similarly, if you invoke IntelliVisor on a composite class, the list box contains all the composite classes defined in the project.

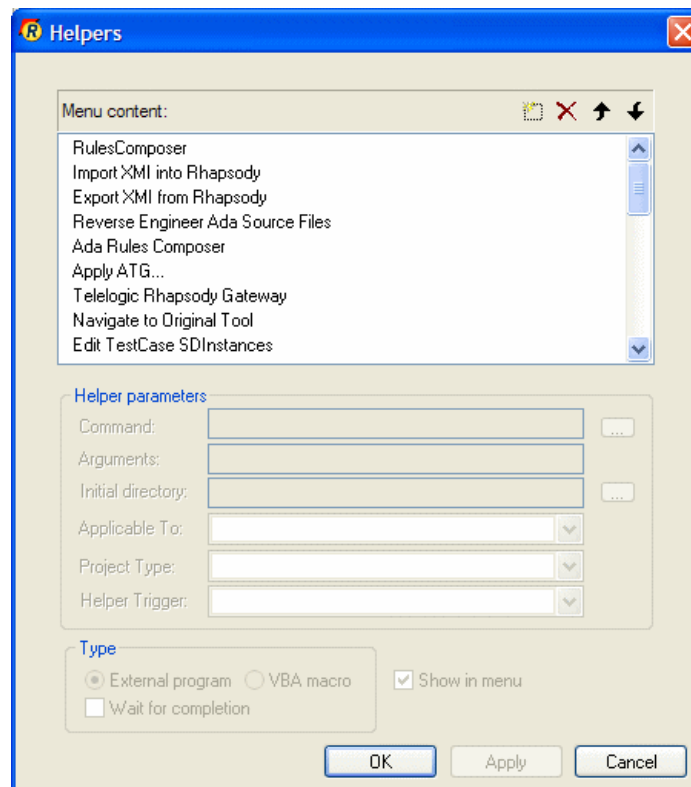
Customizing Rhapsody

You can customize Rhapsody in the following ways:

- ◆ Add helper applications (also known as helpers). Helpers are custom programs that you attach to Rhapsody to extend its functionality. They can be either external programs (executables) or Visual Basic for Applications (VBA) macros that typically use the Rhapsody COM API. They connect to a Rhapsody object via the `GetObject()` COM service. See [Adding Helpers](#).
- ◆ Create a customized profile to use in your company's models. A customized profile has the following advantages:
 - Contains terminology specific to your company
 - Forces adherence to special requirements or industry standards
 - Can be reused in other models to simplify and standardize development effortsSee [Creating Your Own Profile](#).
- ◆ Add new element types to your models. See [Adding New Element Types](#).
- ◆ Create a customized diagram. See [Creating a Customized Diagram](#).

Adding Helpers

To add a helper to the Rhapsody **Tools** menu, open a Rhapsody project and select **Tools > Customize** to open the Helpers dialog box, as shown in the following figure:



Using the Helpers Dialog Box

The following icons are available on the Helpers dialog box:



Click the New icon to create a new helper menu item.



Click the Delete icon to delete a helper menu item.



Click the Move Up icon to move up the helper item on the Rhapsody **Tools** menu.



Click the Move Down icon to move down the helper item on the Rhapsody **Tools** menu.

Use the following boxes to identify and apply your helper application:.


Command	Browse to the path to your helper application.
Arguments	Optionally, add a binding for a parameter that resolves to a run-time instance.
Initial directory	For an external program helper only, browse to the path of the default directory for the helper application.
Applicable to	From the drop-down list, select the model element(s) to associate with the helper.
Project type	From the drop-down list, select one or more profiles (for example, FunctionalC, DoDAF, SysML) to which the helper applies.
Helper trigger	From the drop-down list, select the action that will trigger this helper.

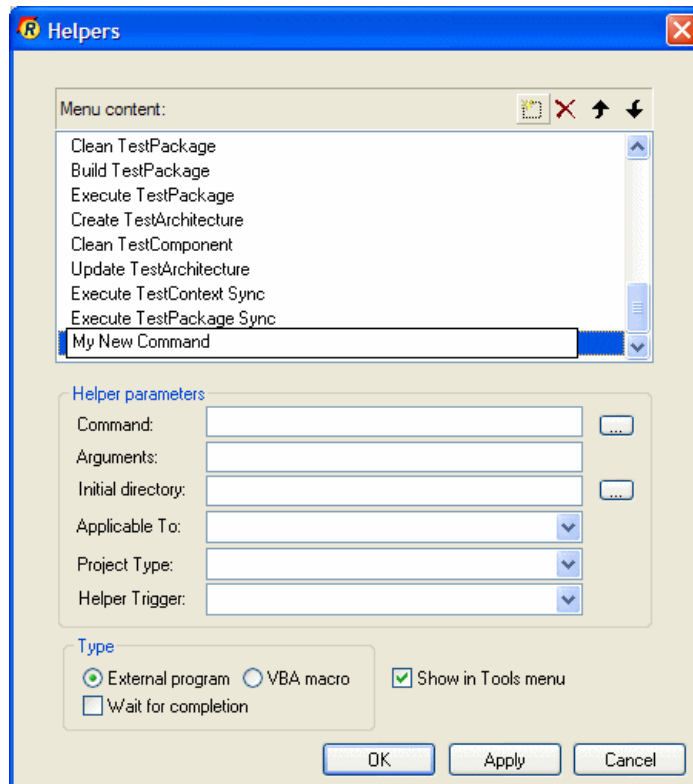
At the bottom of the dialog box, identify if your helper application is an external program helper or VBA macro helper.

See [Creating a Link to a Helper Application](#) for details on how to use the Helpers dialog box.

Creating a Link to a Helper Application


To create a link to a helper program, follow these steps:

1. On the Helpers dialog box, click the New icon  to add a blank line for a new menu item in the **Menu content** box.
2. In the blank field, type the name of the new menu item (for example, `My New Command`), as shown in the following figure:



Note: To make a shortcut key, add an ampersand character before a letter in the name. For example, `&My` makes the letter `M` a menu shortcut. You can press **Alt+M** to invoke this particular helper application once it has been created. Be sure to not use a letter that is already used as a shortcut key on the **Tools** menu or the pop-up menu for the associated model element.

3. Specify the applicable helper parameters:

- ◆ In the **Command** box, enter the command that the menu item should invoke, such as `E:\mks\mksnt\cp.exe` or click its Ellipsis button  to browse to the location of the application.
- ◆ Optionally, in the **Arguments** box, enter any arguments for the command.
- ◆ Optionally, in the **Initial Directory** box, enter an initial default directory for the program. This applies only to external programs.
- ◆ In the **Applicable To** drop-down list, specify which model element(s) to associate with the new command.

Note: If you do not specify a value for this field, the menu command for this helper application may be added to the **Tools** menu depending on what you do in step 5.

- ◆ In the **Project Type** drop-down list, select a **project profile**, as defined in [Creating a Project](#).

Note: If leave this box blank, it uses as the default the profile of the current project you have opened.

- ◆ In the **Helper Trigger** drop-down list, select the action(s) that triggers the new command.

4. Specify the helper type:

- ◆ Select the **External program** radio button if the new command is an external program, such as Microsoft Notepad.

Select the **Wait for completion** check box if you want the external program to complete running before you can continue to work in Rhapsody.

- ◆ Select the **VBA macro** radio button if the new command is a VBA macro and is defined in the `<Project>.vba` file. See [Adding a VBA Macro](#).

5. Depending on what you decided for the **Applicable To** drop-down list:

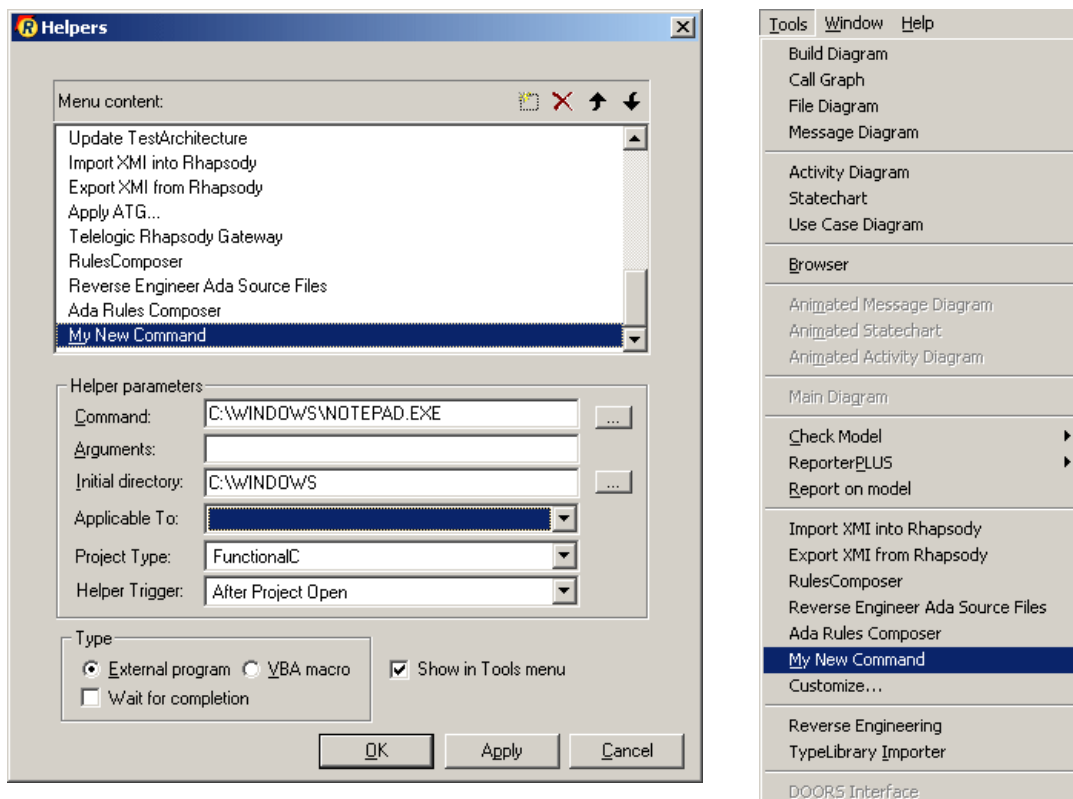
- ◆ If you did not specify an applicable element for the command, verify that the **Show in Tools menu** check box is selected. This means the new menu command for your link to a helper application displays on the **Tools** menu. If you clear this check box, there is no menu command for it on the **Tools** menu, though the link to the helper application still works once the trigger for this command is invoked.
- ◆ If you specified an applicable element for the command, verify that the **Show in Pop-up menu** check box is selected. This means the new command appears in the pop-up menu for the specified model element. If you clear this check box, there is no menu command for it on the pop-up menu for the specified model element, though the link to the helper application still works once this command is invoked.

- Click **OK** to apply your changes and close the dialog box. (You can click the **Apply** button if you want to save your changes but keep the dialog box open to continue working with it.)

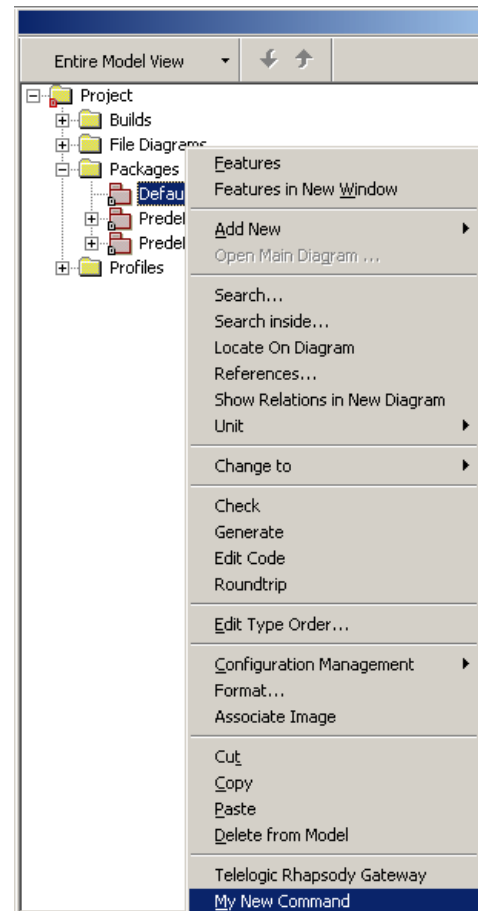
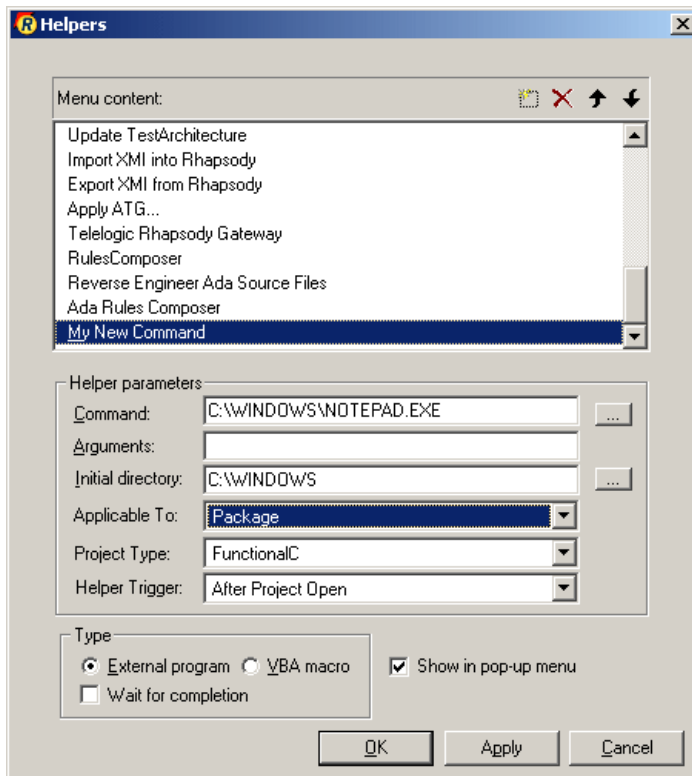
Note: Once you save and close the Helpers dialog box, the link to the helper application you just created is immediately available if the current project is within the parameters that you set for the link. For example, if the Rhapsody project you currently have open uses the FunctionalC profile and you created the My New Command helper application for this profile, then this link to the helper application is immediately available. However, if you specified the DoDAF profile (as selected in the **Project Type** drop-down list) for the My New Command link, then it will not work in your current project.

Examples of Helper Application Menu Commands

If you did not specify an applicable model element and you selected the **Show in Tools menu** check box on the Helpers dialog box, as shown in the following figure on the left, your helper application menu command is added to the **Tools** menu, as shown on the right.



If you specified an applicable model element for your command and you selected the **Show in pop-up menu** check box on the Helpers dialog box, as shown in the following figure on the left, you can right-click either the model element on the Rhapsody browser to access the menu command for the helper application, as shown on the right, or the applicable element in a graphical editor. Note that the command now does not show on the **Tools** menu.



Using a .hep File to Link to Helper Applications

You can use a .hep file to group links to helper applications that help achieve the purpose of a Rhapsody profile. Helper applications are custom programs created by you or a third-party that you can link to within the Rhapsody product. Helper applications add functionality that a profile may not have, such as the ability to query a model or write to a project. For more information about profiles, see [Using Profiles](#).

Using a .hep file is ideal for teams where all members should use the same helper applications. When your project profile and its corresponding .hep file are loaded by reference, when your team members update, they get the latest versions of these files. A .hep file is considered part of a profile and is treated as such.

To see sample .hep files, you can look at the ones provided with the Rhapsody product for certain profiles, such as AUTOSAR, DoDAF, MODAF, NetCentric, and Harmony. For example, `<Rhapsody installation path>\Share\Profiles\DoDAF` contains **DoDAF.sbs** (the profile file) and **DoDAF.hep**. The .hep files for the AUTOSAR and NetCentric profiles show .hep files with references to Java plug-ins.

If sharing links to helper applications is not an issue, or perhaps it is company policy that everyone have access to the same helper applications for all Rhapsody projects, then adding links to helper application in the `rhapsody.ini` file may suffice.

However, you may find using a .hep file more convenient:

- ◆ Easier maintenance. The `rhapsody.ini` file is typically overwritten when you get a new version of Rhapsody. Since the name of a .hep file must correspond with the name of a profile, there is less likelihood of the .hep file being overwritten.
- ◆ Less clutter on your list of menu options. The links to help applications may appear as menu options on the **Tools** menu. Typically, many people work on various projects that may use different models (and profiles) and different helper applications. Using the `rhapsody.ini` file to store all your links to helper applications may cause clutter on your list of menu options. You can use a different .hep file for each profile so that you only see the helpers needed for your project.

To use a .hep file to link to helper applications, follow these steps:

1. Open Rhapsody and use **Tools > Customize** to create one or more links to helper applications. See [Creating a Link to a Helper Application](#).

The code for your link(s) is added to the `rhapsody.ini` file.

2. Close Rhapsody.

3. Open the `rhapsody.ini` file and from the `[Helpers]` section of the file, copy the code for your help application. The following is an example of helper application code that was added to a `rhapsody.ini` file:

```
[Helpers]
...
name30=AutoCommand45
command30=C:\WINDOWS\notepad.exe
arguments30=
initialDir30=C:\WINDOWS
JavaMainClass30=
JavaClassPath30=
JavaLibPath30=
isVisible30=1
isMacro30=0
isPlugin30=0
isSynced30=0
UsingJava30=0
applicableTo30=
applicableToProfile30=Auto2009
helperTriggers30=After Project Open
isPluginCommand30=0
```

Note: Each section of link code starts with `name##`.

4. Open your `.hep` file and paste the code for the link to a helper application (what you copied in the previous step).

Note: Your `.hep` file must have the same name as the name of the profile for your Rhapsody project. For example, if the profile for your Rhapsody project is called `Auto2009`, your `.hep` file must be called `Auto2009.hep`. In addition, both the profile and the `.hep` file must reside in the same folder.

5. In the `rhapsody.ini` file, delete the code that you copied in step 3. The code to link to a helper application should only reside in the `.hep` file when you are using a `.hep` file.
6. Open Rhapsody and open your model.
7. Load the applicable profile by reference. This is the profile that has the corresponding `.hep` file; see step 4.
8. Test to make sure your link to a helper application works as expected. For example, if a link is suppose to open a helper application after you open a model (`helperTriggers30=After Project Open`, as shown in the sample code in step 3), make sure that happens.

Modifying a Link to a Helper Application

To modify a link to a helper application, follow these steps:

1. With a project open in Rhapsody, select **Tools > Customize** to open the Helpers dialog box.
2. If you want to edit the name of the helper application, double-click it in the **Menu content** box and make your changes.
3. Make other changes on the Helpers dialog box as you want. For an explanation of the controls on the Helpers dialog box, see [Creating a Link to a Helper Application](#).


Modifying a .hep File

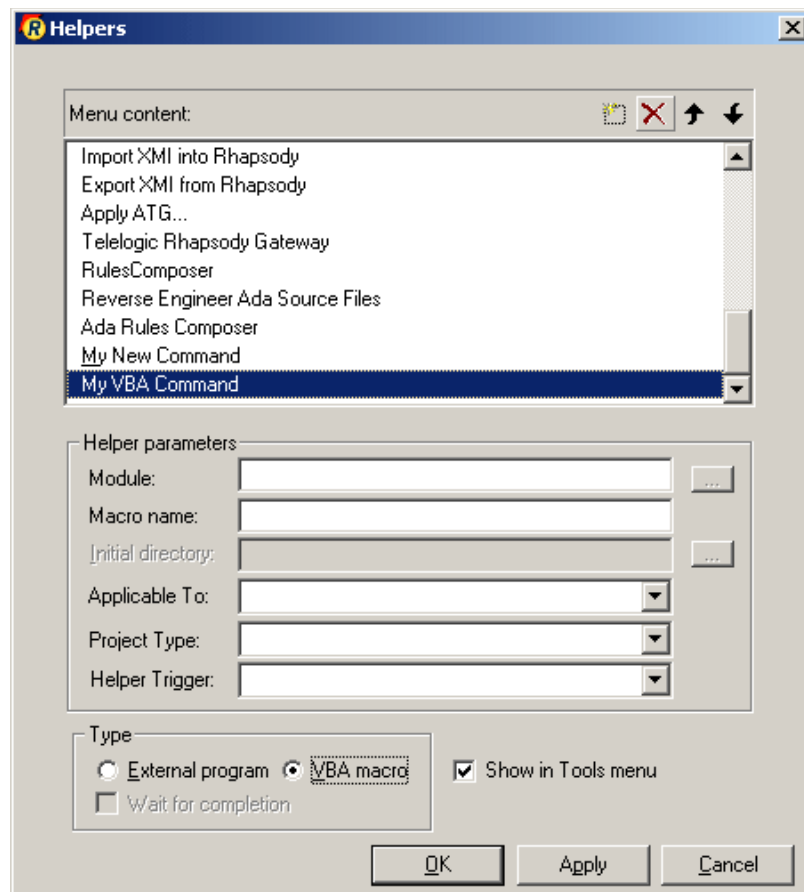
To modify a link to a .hep file by modifying the applicable .hep file, follow these steps:

1. Close the Rhapsody model.
2. Open the .hep file in a text editor (such as Microsoft Notepad) and make your changes.
3. Save your changes to the .hep file.
4. Open the Rhapsody model.
5. Test to make sure your changes work as expected.

Adding a VBA Macro

The Helpers dialog box can also be used to add a VBA macro. To add a VBA macro, follow these steps:

1. With a project open in Rhapsody, select **Tools > Customize** to open the Helpers dialog box.
2. Click the New icon  to add a blank line for a new VBA macro menu command in the **Menu content** box.
3. In the blank field, type the name of the new menu item (for example, `My VBA Command`).
4. Select the **VBA macro** radio button as the helper type. The Helpers dialog box lists VBA-specific options, as shown in the following figure.



5. Specify the applicable helper parameters:

- ◆ In the **Module** box, enter the name of the VBA module.
- ◆ In the **Macro name** box, enter the name of the VBA macro.
- ◆ In the **Applicable To** drop-down list, specify which model element(s) to associate with the new command.

Note: If you do not specify a value for this field, the menu command for this link to a helper application may be added to the **Tools** menu depending on what you do in step 6.

- ◆ In the **Project Type** drop-down list, select a project profile, as defined in [Creating a Project](#).

Note: If leave this box blank, it uses as the default the profile of the current project you have opened.

- ◆ In the **Helper Trigger** drop-down list, select the action(s) that triggers the new command.

6. Depending on what you decided for the **Applicable To** drop-down list:

- ◆ If you did not specify an applicable model element for the command, verify that the **Show in Tools menu** check box is selected. This means the new menu command for your link to a helper application displays on the **Tools** menu. If you clear this check box, there is no menu command for it on the **Tools** menu, though the link to the helper application still works once the command is invoked.
- ◆ If you specified an applicable model element for the command, verify that the **Show in Tools menu** check box is selected. This means the new command appears in the pop-up menu for the specified model element. If you clear this check box, there is no menu command for it on the pop-up menu for the specified model element, though the link to the helper application still works once the command is invoked.

For examples, see [Examples of Helper Application Menu Commands](#).

7. Click **OK** to apply your changes and close the dialog box.

Note: The helper application you just created is immediately available if the current project is within the parameters that you set for the helper application. For example, if the Rhapsody project you currently have open uses the FunctionalC profile and you created the My New Command helper application for this profile, then this helper application is immediately available. However, if you specified the DoDAF profile (as selected in the **Project Type** drop-down list) for the My New Command helper application, then it will not work in your current project.

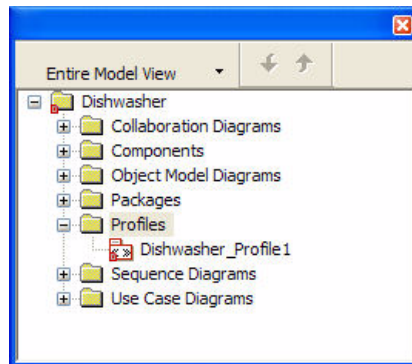
Note

It is your responsibility to add code to your VBA macro to verify that the selected object is actually the core object for your command. The COM command to get the selected element is `getSelectedElement()`.

Creating Your Own Profile

To create a customized profile, follow these steps:

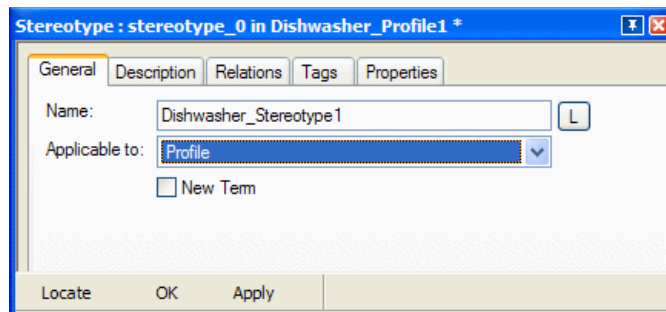
1. Create a project that you want to use as the basis for your customized profile. If you select a Rhapsody profile for this project, the characteristics of that profile are going to be used as the default values.
2. Right-click the top-level project name (for example, **Dishwasher**) and select **Add New > Profile**, and then enter a name for your profile. Notice that Rhapsody creates a **Profiles** category and places your profile within it, as shown in the following figure:



Note: If you have a package that you want to change to be a profile, right-click the package and select **Change to > Profile**. For more information, see [Converting Packages and Profiles](#) for more information.

3. Enter any information about the profile that you want your team members to know about on the **Description** tab.

4. Optionally, you can do the following:
 - a. Define global tags for your profile: Open the Features dialog box for the profile (for example, double-click the profile name) and define tags on the **Tags** tab.
 - b. Add a stereotype to your profile: On the **General** tab of the Features dialog box for your profile, select <<New>> from the **Stereotype** box. A Features dialog box opens for the stereotype on which you can name the stereotype. Notice that by default Rhapsody sets that this stereotype is applicable to a profile, as shown in the following figure. For more information about creating stereotypes, see [Defining Stereotypes](#)



After you close the Features dialog box for the stereotype and return to the Features dialog box for the profile, notice that the stereotype you just created is showing in the **Stereotype** box of the **General** tab for the profile, as shown in the following figure:



Creating a New Stereotype for the New Profile

To create a stereotype, but not immediately apply it to a profile, follow these steps:

1. Right-click your profile and select **Add New > Stereotype**. Remember to select the metaclass to which the stereotype applies.

Note: Later you can apply the stereotype to a profile by opening the Features dialog box for the profile and selecting the particular stereotype on the **General** box.

2. Once Rhapsody creates the **Stereotype** category on the browser, you can right-click the name and select **Add New Stereotype** to create more stereotypes.

Re-using Your Customized Profile

Since a profile is a package, it can be used in other projects to save time and support corporate standards easily. To share your customized profile, use these techniques:

- ◆ You may save the customized profile (package) to make it available through a CM system or in a shared area where other developers can access it.
- ◆ It can be added to an existing project using the **File > Add to Model** option to select the customized profile from its stored location.
- ◆ When multiple projects are displayed in a browser, a developer may drag and drop the customized profile from one project to a different open project to reuse it. See the instructions in [Copying and Referencing Elements among Projects](#) for more information about this process.

Note

You can set your customized profile to be automatically added to a new project either as a copy or a reference using the `AutoCopied` or `AutoReferences` properties. See [Profile Properties](#) for more information.

Adding New Element Types

The stereotype mechanism is used for introducing such new elements. In general, stereotypes are used to add new information to a model element. However, if you define a new stereotype and specify that it should be a **New Term**, it becomes a new element that can be used in models.

To add a new type of element, follow these steps:

1. Create a new stereotype. See [Defining Stereotypes](#).
2. Open the Features dialog box for the new stereotype and select one item from the **Applicable To** drop-down list. This item is the element on which the new element is based.
3. Select **New Term**.
4. Click **OK**.

Once the new term is created, it is possible to add elements of this type via the context menu in the browser.

New Terms and Their Properties

For each new term introduced, properties can be used to specify characteristics such as the type of icon to use for the term in the browser or the icon to be used on the drawing toolbar. For each of the available properties, if no value is provided, Rhapsody uses the value provided for the element on which the new term is based.

Note

Since stereotypes can be added to profiles and packages, the new terms created can be shared across models.

Availability of Out-of-the-Box Model Elements

In addition to allowing the introduction of new element types, Rhapsody allows you to hide any out-of-the-box element types that your users do not need.

The availability of *metaclasses* is determined by the `General::Model::AvailableMetaclasses` property. This property takes a comma-separated list of strings.

Note

To keep all of the out-of-the box metaclasses, leave this property blank.

To limit the availability of certain metaclasses, use this property to indicate the metaclasses that you would like to have available. The strings to use to represent the different metaclasses are as follows

- ◆ ActivityDiagram
- ◆ Actor
- ◆ Argument
- ◆ Association
- ◆ AssociationEnd
- ◆ Attribute
- ◆ Block
- ◆ Class
- ◆ ClassifierRole
- ◆ CollaborationDiagram
- ◆ CombinedFragment
- ◆ Comment
- ◆ Component
- ◆ ComponentDiagram
- ◆ ComponentInstance
- ◆ Configuration
- ◆ Connector
- ◆ Constraint
- ◆ Constructor
- ◆ ControlledFile

- ◆ DefaultTransition
- ◆ Dependency
- ◆ DeploymentDiagram
- ◆ Destructor
- ◆ EnumerationLiteral
- ◆ Event
- ◆ ExecutionOccurrence
- ◆ File
- ◆ Flow
- ◆ FlowItem
- ◆ Folder
- ◆ Generalization
- ◆ HyperLink
- ◆ InteractionOccurrence
- ◆ InteractionOperand
- ◆ Link
- ◆ Message
- ◆ Module
- ◆ Node
- ◆ Object
- ◆ ObjectModelDiagram
- ◆ Package
- ◆ Pin
- ◆ Port
- ◆ PrimitiveOperation
- ◆ Profile
- ◆ Project
- ◆ Reception
- ◆ ReferenceActivity
- ◆ Requirement
- ◆ SequenceDiagram
- ◆ State

- ◆ Statechart
- ◆ Stereotype
- ◆ StructureDiagram
- ◆ Swimlane
- ◆ SysMLPort
- ◆ Tag
- ◆ Transition
- ◆ TriggeredOperation
- ◆ Type
- ◆ UseCase
- ◆ UseCaseDiagram

Creating a Customized Diagram

In addition to allowing you to filter out certain out-of-the-box diagrams that you do not want to see, Rhapsody allows you to add customized diagrams. This is done by creating a new diagram type on the basis of one of Rhapsody's basic diagrams and adding customized diagram elements, if needed, to the list of elements available for the new type of diagram.

Note

The procedure described below for adding customized diagrams with custom elements can only be used for adding new types of diagrams. It is not possible to add new diagram element types to the standard Rhapsody diagrams.

Customized diagrams can be added at the individual model level, or they can be added to profiles so that they can be used with other models as well.

To create your customized diagram, follow these steps:

1. In the browser window, add your customized profile. See [Creating Your Own Profile](#). (While the customized diagram can be added for the current model only, usually developers and designers want to add it to a profile so that it can be reused.)
 2. Select the name of the new profile in the browser, and use the context menu to create a new *stereotype*. See [Defining Stereotypes](#).
 3. Open the Features dialog box for the new stereotype you created, and set the following values:
 - a. On the **General** tab, from the **Applicable to** drop-down list select the type of diagram that should serve as the base diagram for the new diagram type you are creating. In addition, select the **New Term** check box.
 - b. On the **Properties** tab, enter the required values for the following properties:
 - `Model::Stereotype::DrawingToolIcon` supplies the name of the .ico file that should be used as the icon for the new diagram type in the **Diagrams** toolbar.
 - `Model::Stereotype::BrowserIcon` supplies the name of the .ico file that should be used as the icon to represent the new diagram type in the browser.
- Note:** If no value is provided for `DrawingToolIcon`, the file name entered for `BrowserIcon` are used in the **Diagrams** toolbar as well. If values are not provided for either of these properties, then the icon for the base diagram is displayed both in the browser and in the **Diagrams** toolbar.
- `Model::Stereotype::DrawingToolbar` is a comma-separated list representing the elements that should be included in the drawing toolbar for this type of diagram, for example, `RpyDefault,RpySeparator,Actor,Block`.

Note: `RpyDefault` represents all the elements included in the drawing toolbar of the base diagram. If this property is left empty, only the tools from the base diagram is displayed. The toolbar can contain any drawable elements supported by the base diagram, and any new elements based on these elements.

Adding Customized Diagrams to the Diagrams Toolbar

Once you have created a customized diagram type, it is included automatically in the **Tools** menu. You also have the option of including an icon for the new diagram type in the **Diagrams** toolbar. To add the new type of diagram to the **Diagrams** toolbar:

1. Open the Features dialog box for the profile to which you added the new type of diagram.
2. On the **Properties** tab, modify the value of the `General::Model::DiagramsToolbar` property to include the name of the new diagram type in the comma-separated list, for example, `OV-1, RpySeparator, RpyDefault`. (If this property is left empty, the toolbar includes only the default icons.)

The strings to use in this list are given in [Diagram Types](#).

Creating a Customized Diagram Element

After a new diagram type have been defined, you can define new drawing elements that can be included in the drawing toolbar for the new type of diagram. This is done by basing the new element on one of the elements that is available by default in the diagram type that served as the base for the new customized diagram, as follows:

1. Select the name of the relevant profile in the browser, and use the context menu to create a new stereotype.
2. Open the Features dialog box for the new stereotype you created, and set the following values:
 - a. On the **General** tab, from the **Applicable to** drop-down list select the type of drawing element that should serve as the base element for the new diagram element type you are creating. Also, select the **New Term** check box.
 - b. On the **Properties** tab, provide values for the following properties:
 - `Model::Stereotype::DrawingToolIcon`—provide the name of the .ico file that should be used as the icon for the new drawing element when it is included in a drawing toolbar.
 - `Model::Stereotype::DrawingToolTip`—provide the text that should be displayed as a tool tip for the icon in the drawing toolbar.
 - `Model::Stereotype::DrawingShape`—if you would like to customize, to a certain degree, the appearance of the new element that you created, you can

select one of the options provided for this property, for example, you can create a new element based on **Class**, but specify that the object have "rounded corners" when displayed on a diagram.

- `Model::Stereotype::AlternativeDrawingTool`—in certain cases, a number of different out-of-the-box drawing elements are based on the same metaclass, for example, both Class and Composite Class are based on a metaclass called Class. In these cases, in addition to specifying the base metaclass in the **Applicable to** box, you must provide the name of the desired base element in the property. This property does not have to be provided for the "default" element for the metaclass. Using our example above, if you were basing the new element on the Class element, there would be no need to provide a value for this property.

The complete list of diagram elements and corresponding metaclasses is provided in [Diagram Elements](#). This table also indicates the “default” diagram element for each metaclass.

Adding Customized Diagram Elements

After customized diagram elements have been created, they can be added to one or more of the customized diagram types you have created:

1. In the browser, under **Stereotypes**, select the customized diagram to which you would like to add the custom element, and open its Features dialog box.
2. On the **Properties** tab, for the `Model::Stereotype::DrawingToolbar` property, add the name of the new element type you created to the comma-separated list representing the elements that should be included in the drawing toolbar for this type of diagram.

Then names of the elements that can be used for this list are given in [Diagram Elements](#).

Note

After defining new diagrams or diagram elements, you need to reload the model to have access to the new items or, alternatively, choose **View > Refresh New Terms**.

Diagram Types

The following list contains the strings to use for the `General::Model::DiagramsToolbar` property:

- ◆ ActivityDiagram
- ◆ CollaborationDiagram
- ◆ ComponentDiagram
- ◆ DeploymentDiagram
- ◆ ObjectModelDiagram
- ◆ SequenceDiagram
- ◆ Statechart
- ◆ StructureDiagram
- ◆ UseCaseDiagram
- ◆ RpyDefault
- ◆ RpySeparator

Diagram Elements

The following elements can be customized for the specified diagrams, as described in [Adding Customized Diagram Elements](#).

Element Name	Metaclass Name	AlternativeDrawingTool Property Required
Object Model Diagram		
Object	Object	
Class	Class	
Composite Class	Class	Yes
Package	Package	
Port	Port	
Inheritance	Generalization	
Association	AssociationEnd	
Directed Association	AssociationEnd	Yes
Composition	AssociationEnd	Yes
Aggregation	AssociationEnd	Yes
Link	Link	
Dependency	Dependency	
Flow	Flow	
Actor	Actor	
Sequence Diagram		
InstanceLine	ClassifierRole	Yes
EnvironmentLine	ClassifierRole	Yes
Message	Message	
ReplyMessage	Message	Yes
CreateMessage	Message	Yes
DestroyMessage	Message	Yes
TimeoutMessage	Message	Yes
CancelTimeoutMessage	Message	Yes
TimeIntervalMessage	Message	Yes
PartitionLine	ClassifierRole	Yes
Condition Mark	Message	Yes
ExecutionOccurrence	ExecutionOccurrence	

Element Name	Metaclass Name	AlternativeDrawingTool Property Required
InteractionOccurrence	InteractionOccurrence	
InteractionOperatorCombinedFragment	CombinedFragment	
InteractionOperand	InteractionOperand	
Use Case Diagram		
UseCase	UseCase	
Actor	Actor	
Package	Package	
Association	AssociationEnd	
Generalization	Generalization	
Dependency	Dependency	
System Border	ClassifierRole	Yes
Flow	Flow	
Collaboration Diagram		
Classifier Role	ClassifierRole	Yes
Multi Object	ClassifierRole	Yes
Classifier Actor	ClassifierRole	Yes
AssociationRole	ClassifierRole	Yes
Link Message	Message	Yes
Reverse Link Message	Message	Yes
Dependency	Dependency	
Structure Diagram		
Composite Class	Class	Yes
Object	Object	
Block	Block	
Port	Port	
Link	Link	
Dependency	Dependency	
Flow	Flow	
Deployment Diagram		
Node	Node	
Component	Component	

Element Name	Metaclass Name	AlternativeDrawingTool Property Required
Dependency	Dependency	
Flow	Flow	
Component Diagram		
Component	Component	
File	File (Component)	
Folder	Folder	
Dependency	Dependency	
Interface	Class	Yes
Realization	Cannot use as base element	
Flow	Flow	
Statechart		
State	State	
Transition	Transition	
DefaultTransition	DefaultTransition	
AndLine	Cannot use as base element	
StateChartConditionConnector	Connector	Yes
HistoryConnector	Connector	Yes
TerminationConnector	Connector	Yes
JunctionConnector	Connector	Yes
DiagramConnector	Connector	Yes
StubConnector	Connector	Yes
JoinConnector	Connector	Yes
ForkConnector	Connector	Yes
TransitionLabel	Transition	Yes
TerminationState	State	Yes
Dependency	Dependency	
Activity Diagram		
Action	State	Yes
ActionBlock	State	Yes
SubActivityState	State	Yes
ObjectNode	State	Yes

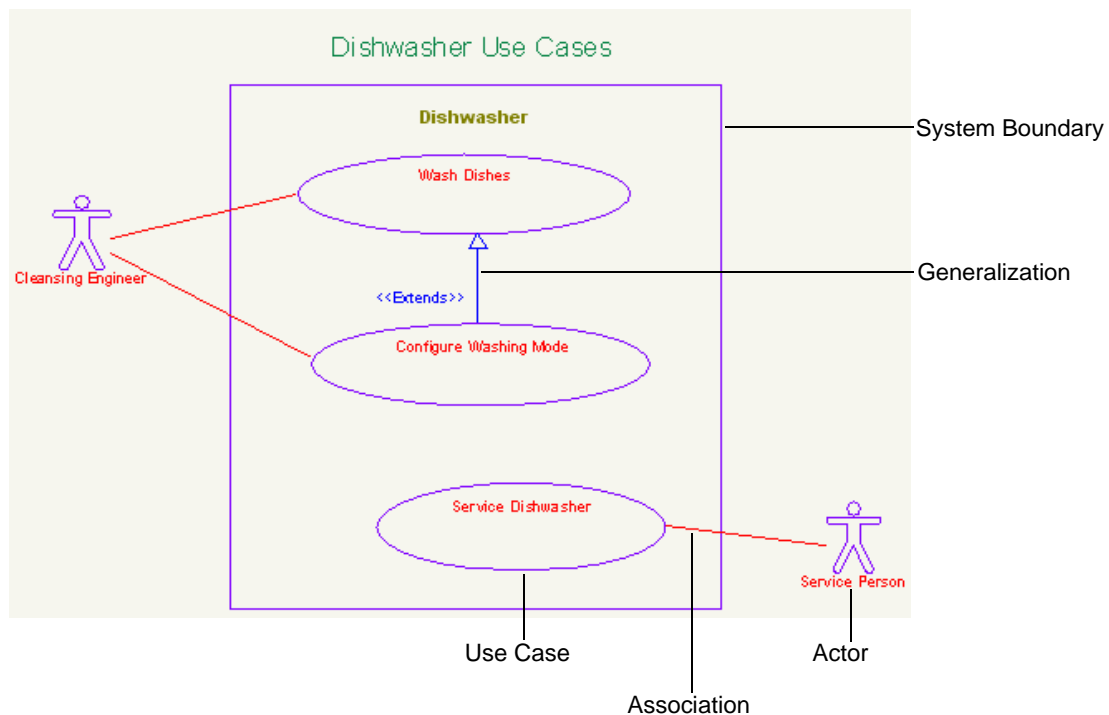
Element Name	Metaclass Name	AlternativeDrawingTool Property Required
ReferenceActivity	ReferenceActivity	
Transition	Transition	
DefaultTransition	DefaultTransition	
LoopTransition	Transition	Yes
ActivityChartConditionConnector	Connector	Yes
TerminationState	Connector	Yes
JunctionConnector	Connector	Yes
DiagramConnector	Connector	Yes
JoinConnector	Connector	Yes
ForkConnector	Connector	Yes
TransitionLabel	Transition	Yes
Swimlane Frame	Swimlane	Yes
SwimlaneDivider	Swimlane	Yes
Dependency	Dependency	
ActivityPin	Connector	Yes
ActivityParameter	Connector	Yes

Use Case Diagrams

Use case diagrams (UCDs) model relationships between one or more users (actors) and a system or class (classifier). You use them to specify requirements for system behavior. In addition, Rhapsody UCDs depict generalization relationships between use cases as defined in the UML (see [Generalizations](#)). Rhapsody does not generate code for UCDs.

Use Case Diagrams Overview

Use cases (for example, wash dishes, configure washing mode, and service dishwasher) represent the user's expectation for a system. Actors (for example, a cleansing engineer and service person) represent any external object that interacts with the system. Uses cases reside inside the system boundary, and actors reside outside it. Association lines show relationships between the use cases and the actors. For example, the actor in charge of servicing the dishwasher requires a dishwasher capable of being receiving serviced. The following UCD models these requirements.











Creating Use Case Diagram Elements

The following sections describe how to use the use case diagram drawing tools to draw the parts of a use case diagram. See [Graphic Editors](#) for basic information on diagrams, including how to create, open, and delete them.

Use Case Diagram Drawing Toolbar


The **Drawing** toolbar for a use case diagram contains the following tools:

Drawing Icon	Description
	Create Use Case icon draws a representation of a user-visible function. A use case can be large or small, but it must capture an important goal of a user for the system.
	Create Actor represents users of the system or external elements that either provide information to the system or use information provided by the system.
	Create Package groups systems or parts of a system into logical components.
	Create Association shows relationships between actors and use cases.
	Create Generalization shows how one use case is derived from another. The arrow head points to the parent use case.
	Dependency defines dependencies between an actor and a use case, between two actors, or between two use cases.
	Create Boundary box delineates the system's design scope and its external actors with the use cases inside the system boundary and the actors outside.
	Flow provides a mechanism for specifying exchange of information between system elements at a high level of abstraction. This functionality enables you to describe the flow of data and commands within a system at a very early stage, before committing to a specific design. As the system specification evolves, you can refine the abstraction to relate to the concrete implementation. See Flows and FlowItems for detailed information on flows.

System Boundary Box

The system boundary box should be the first element placed in a UCD. It distinguishes the border between use cases and actors—use cases are inside the borders of the system boundary box and actors are outside of it.

To create a system boundary box, follow these steps:


1. Click the **Create Boundary box**  icon.
2. Click once in the drawing area. Rhapsody creates a boundary box named System Boundary Box. Alternatively, click-and-drag with the mouse to draw the system boundary box.
3. If desired, edit the default name and press **Enter**.

Use Cases

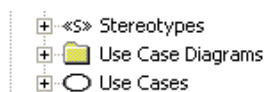
Use cases represent the externally visible behaviors, or functional aspects, of the system. They consist of the abstract, uninterpreted interactions of the system with external entities. This means that the content of use cases is not used for code generation. A use case is displayed in a UCD as an oval containing a name.

Creating a Use Case

To create a use case, follow these steps:

1. Click the **Create Use Case**  icon.
2. Click once in the diagram or click-and-drag with the mouse to draw a use case of a specific size. By default, the use case is named `usecase n` , where n is an integer greater than or equal to 0.
3. If desired, edit the default name and press **Enter**.

The new use case is displayed in both the UCD and the browser. The browser icon for a use case is an oval, as shown in the following figure.



Modifying the Features of a Use Case

The Features dialog box enables you to define the features of a use case, as shown in this example.



A use case has the following General features:

- ◆ **Name** allows you to replace the default name with the name you want for this use case.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype** specifies the stereotype of the use case, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.

- ◆ **Default Package** specifies a group for this use case.

Adding Attributes to a Use Case

Because a use case is a stereotyped class, it can have attributes. No code is generated for these attributes. Rhapsody does not display attributes in the UCD. To access attributes for a use case, use the browser.

To add a new attribute to a use case, follow these steps:

1. Select the use case in the UCD editor.
2. Right-click the use case, and then select **New Attribute** from the pop-up menu. The Attribute dialog box opens.
3. Type a name for the attribute in the **Name** field.

Use the **L** button next to the name field to assign a logical label. For more information on labels, see [Labeling Elements](#).

4. Select the **Type**, **Visibility**, and **Multiplicity** for the attribute.
5. Type a description in the **Description** tab.
6. Click **OK** to apply your changes and close the dialog box.

To add an existing attribute to a use case, follow these steps:

1. In the browser, locate the class that contains the attribute.
2. Click-and-drag the attribute to the use case in the browser. This creates a separate copy of the attribute under the use case.

Note

If you click-and-drag an attribute from one use case to another, the attribute is moved, not copied.

Adding Operations to a Use Case

Because a use case is a stereotyped class, it can have operations. To add a new operation to a use case, follow these steps:

1. Select the use case in the UCD editor.
2. Right-click the use case, and then select **New Operation**. The Primitive Operation dialog box opens.
3. Type a name for the operation in the **Name** field.
4. If desired, specify a stereotype.
5. Select the visibility of the operation.

6. Select a type for the operation in the **Type** field.
7. Select the **Return Type** for the operation.
8. Specify the operation modifiers.
9. Add any arguments using the **Arguments** section.
10. Type a description in the **Description** tab.
11. Select the **Implementation** tab.
12. Type the implementation code for the operation in the **Implementation** text box.

Note: Code is not generated for the contents of use cases. This implementation is for descriptive purposes only.
13. Click **OK** to apply your changes and close the dialog box.

Creating a Statechart or Activity Diagram for a Use Case Activity diagrams

Because use cases are stereotyped classes, it is possible to add a statechart or an activity diagram.

To add a statechart or activity diagram to a use case, follow these steps:

1. Select the use case in the UCD editor.
2. Right-click the use case, and then select either **New Statechart** or **New Activity Diagram**.

For more information on these diagrams, see [Statecharts](#) and [Activity Diagrams](#).

Actors

Actors are the external entities that interact with a use case. Typical actors that operate on real-time, embedded systems are buses (for example, Ethernet or MIB), sensors, motors, and switches. An actor is represented as a figure in UCDs.


Actors are a kind of UML classifier similar to classes and they can participate in sequences as instances. However, actors have the following constraints imposed on them:

- ◆ They cannot aggregate or compose any elements.
- ◆ They generalize only from other actors.
- ◆ They cannot be converted to classes, or vice versa.

Rhapsody can generate code for an actor, which can be used in simulation testing of the system you are building. See [Generating Code for Actors](#) for more information.

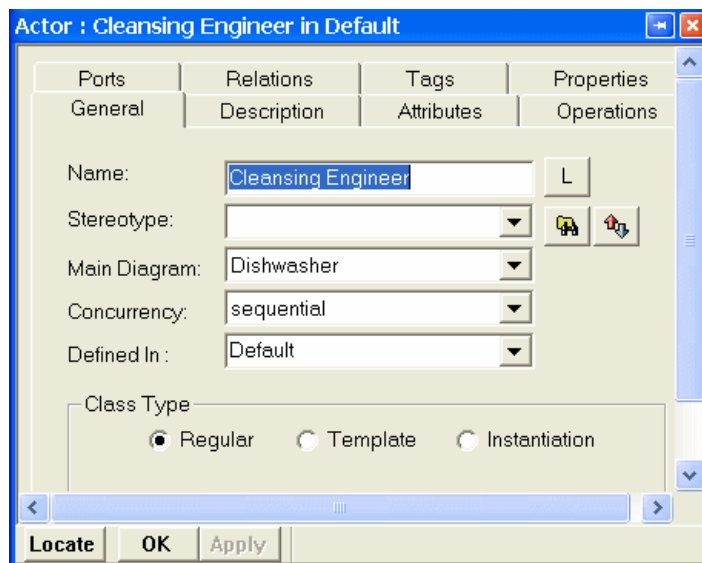
Creating an Actor

To create an actor, follow these steps:

1. Click the **Actor**  icon, and then click once in the UCD. Alternatively, click-and-drag to draw the actor.
2. Edit the default name, and then press **Enter**.

Modifying the Features of an Actor

The Features dialog box enables you to define the features of an actor, as shown in this example.



An actor has the following features:

- ◆ **Name**—Specifies the name of the element. The default name is `actor_n`, where *n* is an incremental integer starting with 0.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the actor, if any. They are enclosed in guillemets, for example `<<S1>>` and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.

- ◆ **Main Diagram**—Specifies the main diagram for the actor.

More than one UCD can contain the same use case or actor. You can select one of these diagrams to be the main diagram for the use case or actor. This is the diagram that will open when you select the **Open Main Diagram** option in the browser.

- ◆ **Concurrency**—Specifies the concurrency of the actor. The possible values are as follows:
 - **Sequential**—The element will run with other classes on a single system thread. This means you can access this element only from one active class.
 - **Active**—The element will start its own thread and run concurrently with other active classes.
- ◆ **Defined In**—Specifies which element owns this actor. An actor can be owned by a package, class, or another actor.
- ◆ **Class Type**—Specifies the class type. The possible values are as follows:
 - **Regular**—Creates a regular class.
 - **Template**—Creates a template. To specify the necessary arguments, click the **Arguments** button.
 - **Instantiation**—Creates an instantiation of a template.

To create an instance of a class, select the **Instantiation** radio button and select the template that the instance is from. For example, if you have a template class A and create B as an instance of that class, this means that B is created as an instance of class A at run time.

To specify the necessary arguments, click the **Arguments** button.

Adding Attributes and Operations

Attributes and operations are added to actors just as they are added to classes.

To add an attribute or operation to an actor, follow these steps:

1. Open the Features dialog box for the actor.
2. Select the **Attributes** tab or **Operations** tab, as appropriate.
3. Select the <New> label. A new row is displayed, with the default values filled in.
4. If needed, change the default values for the new attribute or operation.
5. Click **OK** to apply your changes and close the dialog box.

For detailed information on creating attributes and operations, see [Defining the Attributes of a Class](#).

Creating a Statechart, Activity, or Structure Diagram Activity diagrams

Because an actor is a special type of class, it can have a statechart, an activity diagram, or a structure diagram. Follow these steps to use the editor to add one of these diagrams to an actor:

1. Select the actor in the UCD editor.
2. Right-click the actor, and then select **New Statechart**, **New Activity Diagram**, or **New Structure Diagram** from the pop-up menu.

For more information on these diagrams, see [Statecharts](#), [Activity Diagrams](#), or [Structure Diagrams](#).

Generating Code for an Actor

To generate code for an actor, follow these steps:

1. Locate the active configuration in the browser.
2. Open the Features dialog box for the active configuration.
3. Select the **Initialization** tab.
4. Select **Generate Code for Actors**.


Alternatively, you can generate code for the actor by right-clicking on the actor in the UCD and selecting **Generate**.

When you generate code for the configuration, code is also generated for any actors that are part of the configuration. See [Setting Component Configurations in the Browser](#) for detailed information on configurations.

Packages

Packages logically group system components. They are represented in UCDs as a file folder.

To create a package, follow these steps:


1. Click the **Create Package**  icon, and then click once in the UCD. Alternatively, click-and-drag with the mouse to draw the package.
2. Edit the default name, and then press **Enter**.

The new package will be displayed in both the diagram and the browser (listed under Packages).

Associations

Associations represent lines of communication between actors and use cases. Use cases can associate only with actors, and vice versa.

To create an association, follow these steps:

1. Click the **Create Association**  icon.
2. Click either the actor or the use case. Note that the crosshairs change to a circle with crosshairs when you are on an element that can be part of an association.
3. Move the cursor to the target of the association and click once. If the source is an actor, the target must be a use case, and vice versa.
4. Type a name for the association, then press **Enter**.


The new association is displayed in both the UCD and the browser (under the actor's `Association Ends` category).

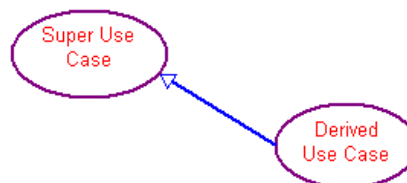
See [Modifying the Features of an Association](#) for information on modifying an association. Association features include the type, name, roles, multiplicity, qualifiers, and description.

Generalizations

UML allows for generalization as a way of factoring out commonality between use cases. In other words, it provides a means to derive one use case from another. Generalizations are allowed between use cases and actors.

To create a generalization relationship, follow these steps:


1. Click the **Create Generalization**  icon.
2. Click the derived use case.
3. Move the cursor to the closest edge of the super-use case and click once.



Dependencies

A *dependency* is a directed relationship from a client to a supplier in which the functioning of a client requires the presence of the supplier, but the functioning of the supplier does not require the presence of the client. Generalizations are allowed between any two UCD elements: use case, actor, or package.

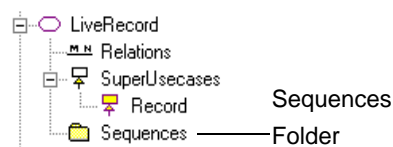
To create a dependency relationship, follow these steps:

1. Click the **Dependency**  icon.
2. Select the client element.
3. Move the cursor to the closest edge of the supplier element and click once.

You can set the dependency stereotype using the Features dialog box. See [Dependencies](#).

Sequences

UCDs assist in the analysis phase of a project. They capture hard and firm constraints at a high level. As design decisions are made, you further decompose UCDs to create more possible use cases and scenarios, or sequences, that implement the use case. Each use case has a folder in the browser containing some of its possible sequences.



Scenarios describe not only the main path through a use case, but can also include background environmental and situational descriptions to set the stage for future events. In other words, they can provide detailed definitions of preconditions for a use case. Therefore, a sequence describes the main path through a use case, whereas a variant, represented by a child use case, describes alternate paths.

For example, consider a VCR. One sequence of the InstallationAndSetup use case might be the following:

1. Unadd-on the VCR and accessories.
2. Install batteries in the remote control.
3. Connect the antenna or cable system to the VCR.
4. Set the CH3/CH4 switch.

5. Turn on the VCR and select an active channel.
6. Learn to use the TV/VCR button.
7. Test the VCR connections.

The specific sequence of steps through a particular use case is better expressed through a sequence diagram. See [Sequence Diagrams](#) for detailed information.

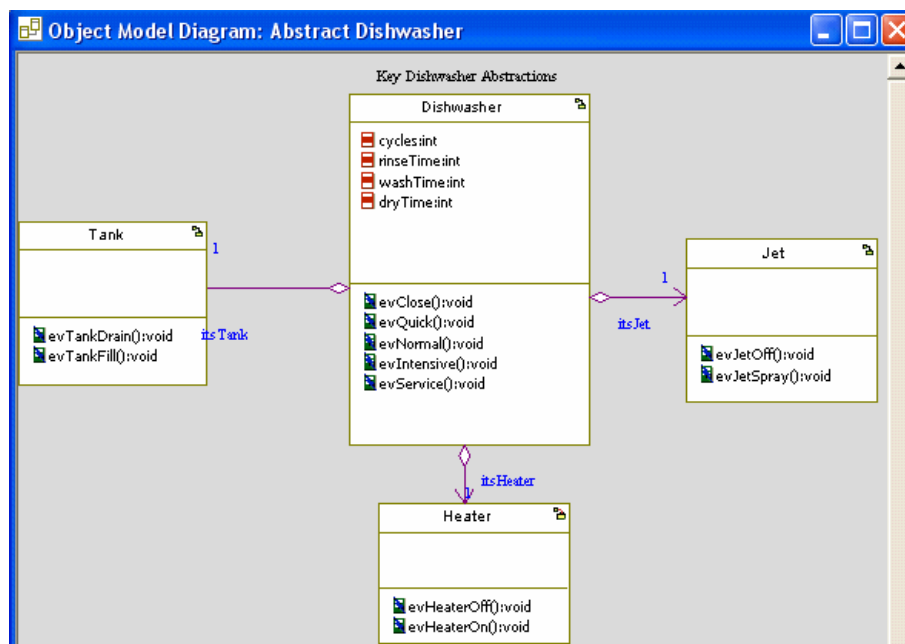
Object Model Diagrams

Object model diagrams (OMDs) specify the structure and static relationships of the classes in the system. Rhapsody OMDs are both class diagrams and object diagrams, as specified in the UML. They show the classes, objects, interfaces, and attributes in the system and the static relationships that exist between them.

Structure diagrams focus on the objects used in the model. Although you can put classes in structure diagrams and objects in the OMD, the toolbars for the diagrams are different to allow a distinction between the specification of the system and its structure. See [Structure Diagrams](#) for more information.

Object Model Diagrams Overview

More than simply being a graphical representation of the system structure, OMDs are constructive. The Rhapsody code generator directly translates the elements and relationships modeled in OMDs into source code in a number of high-level languages. The following figure shows a sample OMD.



In this diagram, the thick sidebars on the `Dishwasher` class denote that it is the active class.

You can specify and edit operations and attributes directly within class and object boxes. Simply highlight the appropriate element to make it active, then type in your changes. To open the Features dialog box for a given attribute or operation, just double-click the element within the compartment.

Note









To add a new operation or attribute, press the Insert key when the appropriate compartment is active.











Creating Object Model Diagram Elements

The following sections describe how to use the object model diagram drawing tools to draw the parts of an object model diagram. See [Graphic Editors](#) for basic information on diagrams, including how to create, open, and delete them.

Object Model Diagram Drawing Toolbar

The **Drawing** toolbar for an object model diagram includes the following tools:



Drawing Icon	Description
	Select is a pointing tool to identify parts of the diagram requiring changes or additions.
	Object is the structural building block of a system. Objects form a cohesive unit of state (data) and services (behavior). Every object has a public part and an private part. See Objects for more information.
	Class defines properties that are common to all objects of that type. See Classes for more information.
	Composite class is a container class. You can create objects and relations inside a composite class. See Composite Classes for more information.
	Package is a group of classes. See Packages for more information.
	File is available only in Rhapsody in C has an additional icon. Use it to create file model elements. A file is a graphical representation of a header (.h) or code (.c) source file. See Files for more information.
	Create Port draws connection points among objects and their environments.
	Inheritance shows the relationship between a derived class and its parent.

Drawing Icon	Description
	Association creates connections that are necessary for interaction such as messages.
	Directed association indicates the only object that can send messages to another object. See Using Directed Associations for more information.
	Aggregation specifies an association between an aggregate (whole) and a component part. See Using Aggregation Associations for more information.
	Composition defines a class that contains another part class. See Using Composition Associations for more information.
	Link creates an association between the base classes of two different objects. See Links for more information.
	Dependency creates a relationship in which the proper functioning of one element requires information provided by another element. See Dependencies for more information.
	Flow specifies the flow of data and commands within a system. See Flows and FlowItems for more information.
	Realization specifies a realization relationship between an interface and a class that implements that interface. See Realization for more information.
	Interface adds a set of operations that publicly define a behavior or way of handling something so knowledge of the internals is not needed.
	Create Actor represents an element that is external to the system. See Actors for more information.

The following sections describe how to use these tools to draw the parts of an OMD. See [Graphic Editors](#) for basic information on diagrams including how to create, open, and delete them.

Objects

Rhapsody separates objects from classes in diagrams. There are two types of objects:

-  **Objects with explicit object types.** Specify only the features that are relevant for the instance. An explicit object instantiates a “normal” class from the model.
-  **Objects with implicit types**—Enable you to specify other features that belong to classes, such as attributes, operations, and so on. An implicit object is a combination of an instance and a class. Technically, the class is hidden.
Note that Rhapsody in J does not support objects with implicit types.

An object is basically an instance of a class; however, you can create an object directly without defining a class. Objects belong to packages and *parts* belong to structured classes; the browser separates parts and objects into separate categories.

Creating an Object

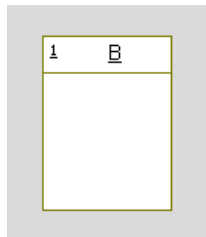
To create an object, follow these steps:

1. Click the **Object** icon in the **Drawing** toolbar.
2. Click, or click-and-drag, in the drawing area.
3. Edit the default name, then press **Enter**.

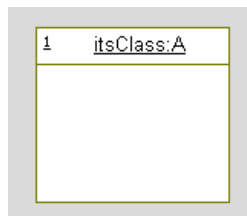
If you specify the name in the format `<ObjectName:ClassName>` (for an object with explicit type) and the class `<ClassName>` exists in the model, the new object will reference it. If it does not exist, Rhapsody prompts you to create it.

By default, Rhapsody creates objects with implicit type. In the OMD, an object is shown like a class box, with the following differences:

- ◆ The name of the object is underlined.
- ◆ The multiplicity is displayed in the upper, left-hand corner.

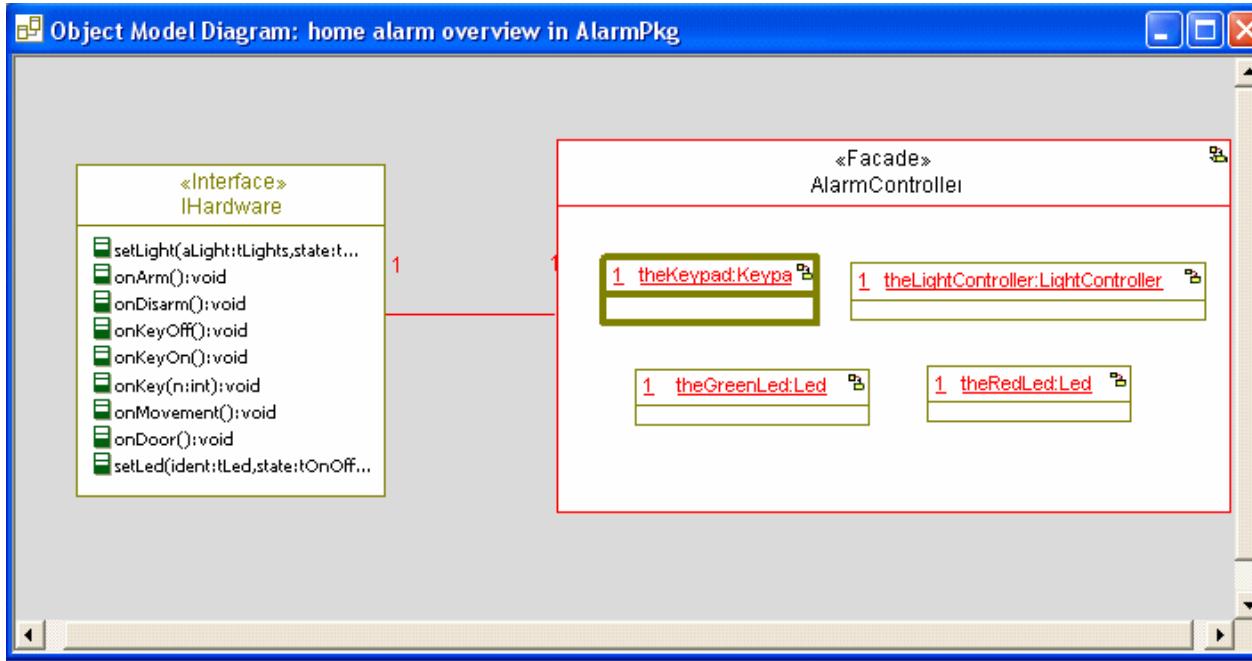


The following figure shows an object of explicit type:



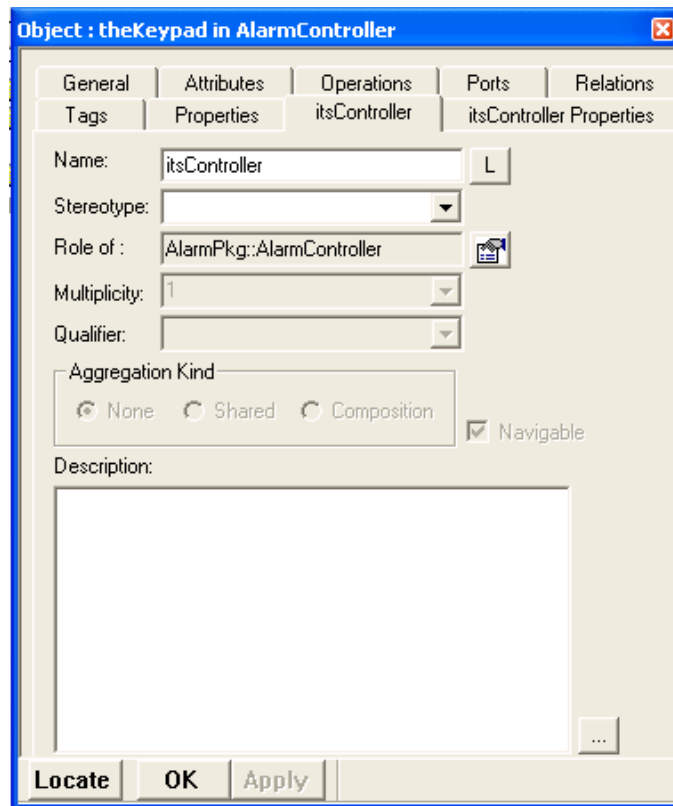
As with classes, you can display the attributes and operations in the object. See [Setting Display Options](#) for detailed information.

The following figure shows an OMD that contains parts.



Modifying the Features of an Object

The Features dialog box enables you to change the features of an object, including its concurrency and multiplicity. The following figure shows the Features dialog box for an object.



An object has the following features:

- ◆ **Name**—Specifies the name of the element. The default name is `object_n`, where *n* is an incremental integer starting with 0.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the object, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.

- ◆ **Role of**—A read-only field that specifies the class, actor, or use case that plays a role in the association.

- ◆ **Multiplicity**—Specifies the number of occurrences of this instance in the project. Common values are one (1), zero or one (0,1), or one or more (1..*).
- ◆ **Qualifier**—Shows the attributes in the related class that could function as qualifiers.
- ◆ A *qualifier* is an attribute that can be used to distinguish one object from another. For example, a PIN number can serve as a qualifier on a BankCard class. If a class is associated with many objects of another class, you can select a qualifier to differentiate individual objects. The qualifier becomes an index into the multiple relation. Adding a qualifier makes the relation a qualified association
- ◆ **Aggregation Kind**—Specifies the type of aggregation. The possible values are as follows:
 - **None**—No aggregation.
 - **Shared** (empty diamond)—Shared aggregation (whole/part relationship).
 - **Composition** (filled diamond)—Composition relationship. The instances of the class at this end contains instances of the class at the other end as a part. This part cannot be contained by other instances.
- ◆ **Navigable**—Specifies whether the association allows access to the other class. Both ends of a bi-directional association are navigable. In a directed association, the element that has the arrow head is navigable; the other end is not. See [Using Directed Associations](#) for more information.

The navigability of an association influences the appearance of the association arrow in the diagram. If one end of a symmetric association is navigable and the other is not, the association line includes an arrow head.

- ◆ **Description**—Describes the object. This field can include a hyperlink. See [Hyperlinks](#) for more information.

Converting Object Types

You can easily change the type of an object using the Features dialog box for the object.

If you convert an object with implicit type to an object with explicit type (by selecting <Explicit> in the **Type** field), a new class is created. By default, the name of the new class is <object name>_class.

If you convert an object of explicit type to an object of implicit type, the following actions occur:

- ◆ The original class is copied into the object of implicit type.
- ◆ Graphical relations are removed.
- ◆ Symmetric associations become directional.
- ◆ Links disconnect from associations.

Converting Classes to Objects

To convert a class to an object, right-click the class in the diagram and select **Make an Object** from the pop-up menu. Rhapsody converts the class to an object named `its<ClassName>:<ClassName>`. For example, if you converted class `A` to an object, the name of the object would be `itsA:A`.

Generating Code for Objects

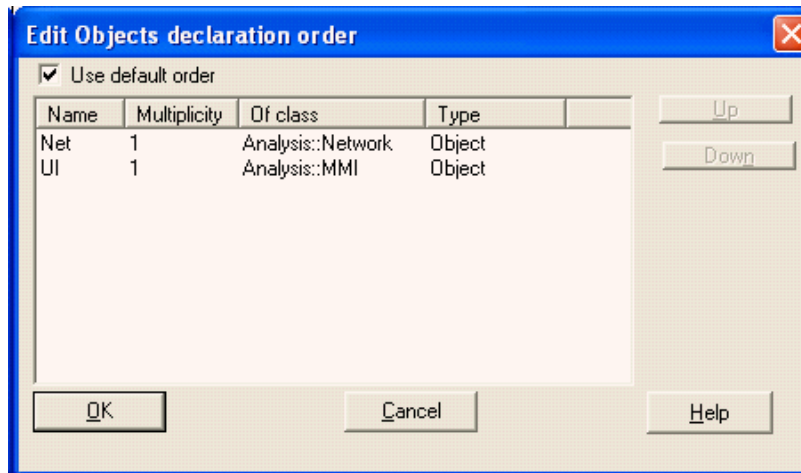
For objects with explicit type, code is generated as in previous versions of Rhapsody. The following table lists the results of generating code for objects with implicit type.

Situation	Results of Code Generation
Implicit type	During code generation, the object is mapped in two parts: <ul style="list-style-type: none">• An implicit class with the name <code><object>_C</code>.• The instance of the class in its owner (either a composite class or package). The name of the instance is <code><object></code>.
Implicit type in a package (global)	The code for the instance is generated in the package file and the code for the implicit class is generated into files named <code><object>.h</code> and <code><object>.cpp</code> .
Implicit type in a structured class (part)	The code for the instance is generated in the composite class file and the code for the implicit class is generated as a nested class of the composite (in the composite's file).
Embeddable objects	The default code scheme for code generation for objects is changed to embeddable. The default values of the following properties were changed: <ul style="list-style-type: none">• <code>CPP_CG::Class::Embeddable-Checked</code>• <code>CPP_CG::Relation::ImplementWithStaticArray-FixedAndBounded</code>

Changing the Order of Objects

To edit the order of objects, follow these steps:

1. In the browser, right-click the **Objects** category icon, then select **Edit Objects Order** from the pop-up menu. The Edit Objects declaration order dialog box opens, as shown in the following figure. It lists all the files and objects in the current package.



2. Unselect the **Use default order** check box.
3. Select the object you want to move.
4. Click **Up** to generate the object earlier or **Down** to generate it later.
5. Click **OK** to apply your changes and close the dialog box.

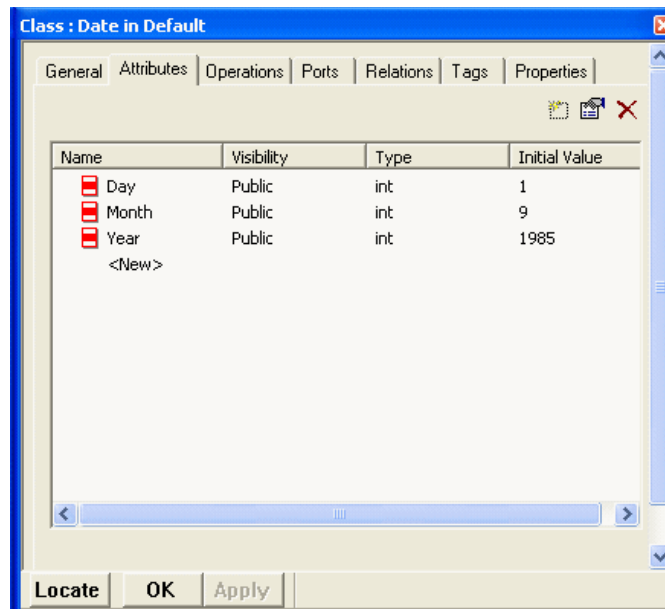
Changing the Value of an Instance

You can specify the value of attributes for instances. An *attribute value* is the value assigned to an attribute during run time. This functionality enables you to describe a possible setup of objects and parts at a certain point in their lifecycle—you can see a “snapshot” of the system, including the instances that exist and their values. To support this functionality, the Features dialog box for instances includes a new column, **Value**.

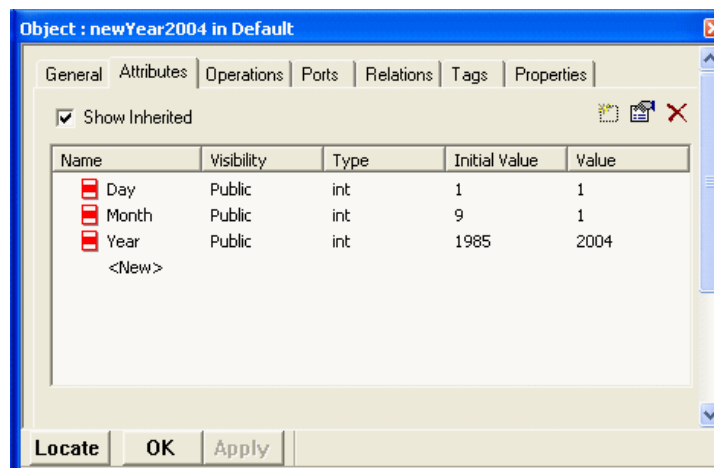
Note

Initial values are features of the attributes of the *class*, whereas instance values characterize the specific *instance* of the class (that object).

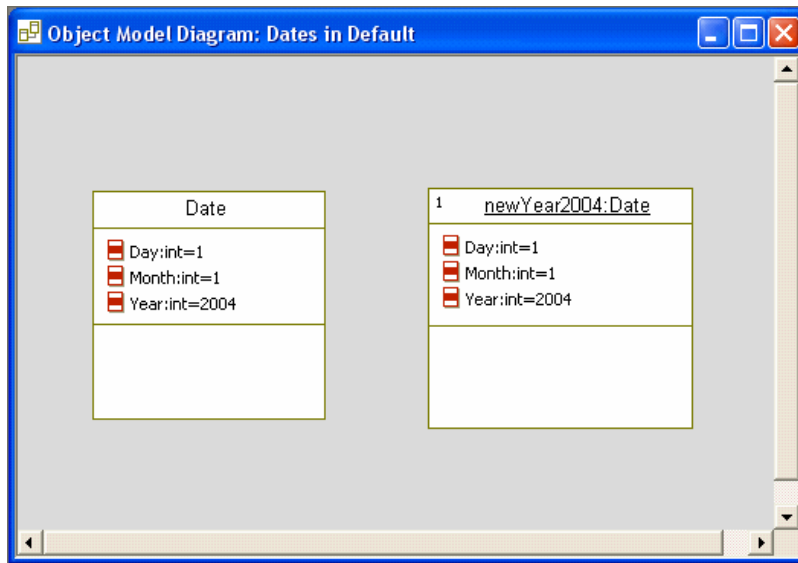
For example, consider the class, `Date`, and an object of `Date` called `newYear2004`. The class `Date` has the attributes `Day`, `Month`, and `Year`. The following figure shows the initial values for the class `Date`.



The following figure shows the attributes for object `newYear2004` of class `Date`. Note that the **Show Inherited** check box specifies whether to display the inherited attributes so you can easily modify them.



Click the **Specification View** icon to view the attributes and operations of an object in the OMD. The following OMD shows the class, `Date`, and the object of `Date` called `newYear2004`.



In the OMD, the object values are displayed using the following format:

```
[visibility]<attribute name>:<attribute type>=<value>
```

Note the following:

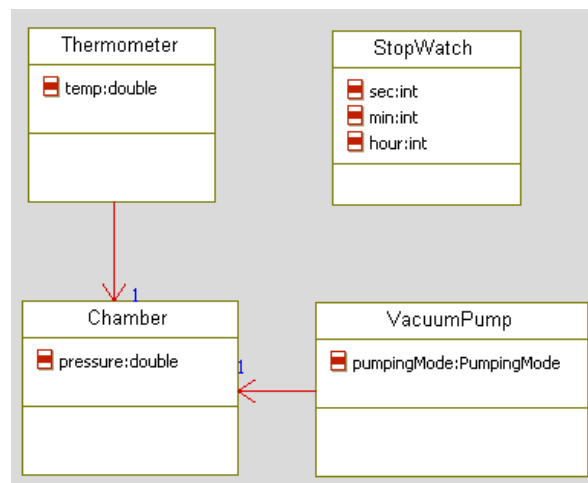
- ◆ Instance values are always displayed. To hide the entire attribute, right-click the object and select **Display Options** from the pop-up menu.
- ◆ Both the instance and the class must be “available” in the model.

Example

The following model shows how to use instance attribute values to take snapshots of a vacuum pump model at different stages in the lifecycle.

The purpose of the vacuum pump is to pump the air out of a chamber. You want to see the state of the system at various points in time—the initial value, the value after one hour, and the final value.

The following figure shows the OMD for the model.



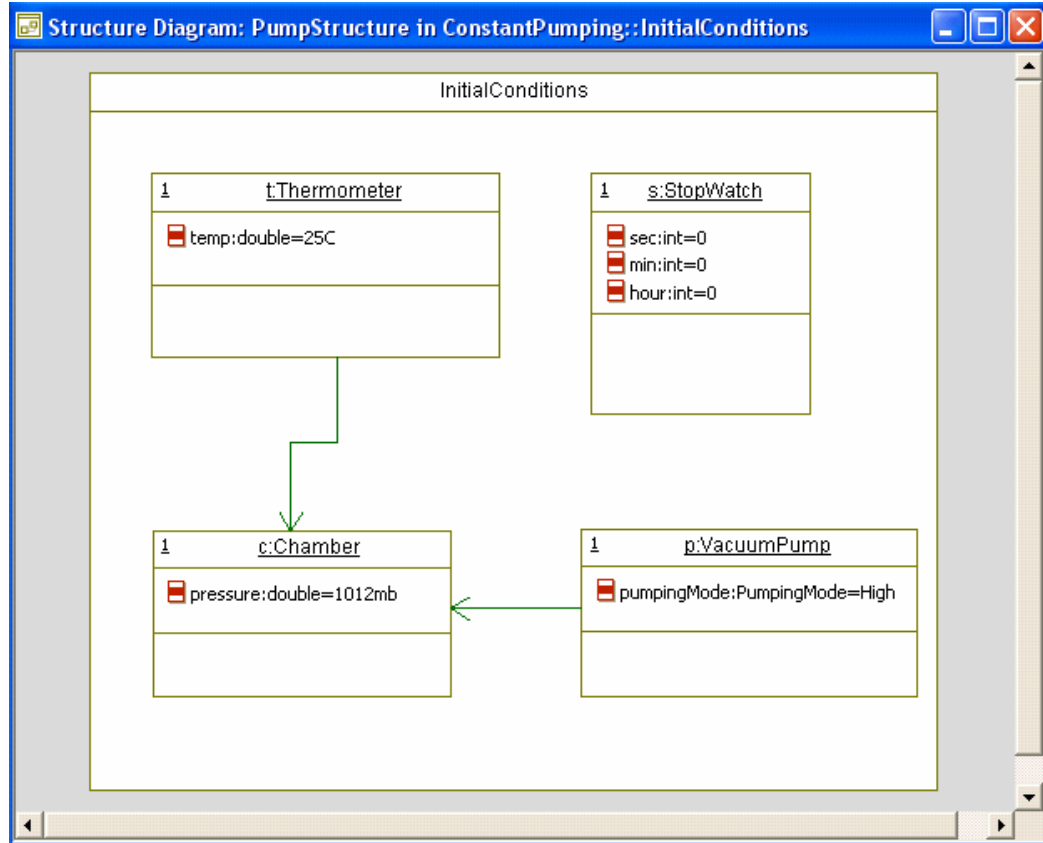
Follow these steps:

1. Create a package called `ConstantPumping`.
2. Set the `CG::Class::UseAsExternal` property to `Checked` so the package is considered external (and code will not be generated for it).

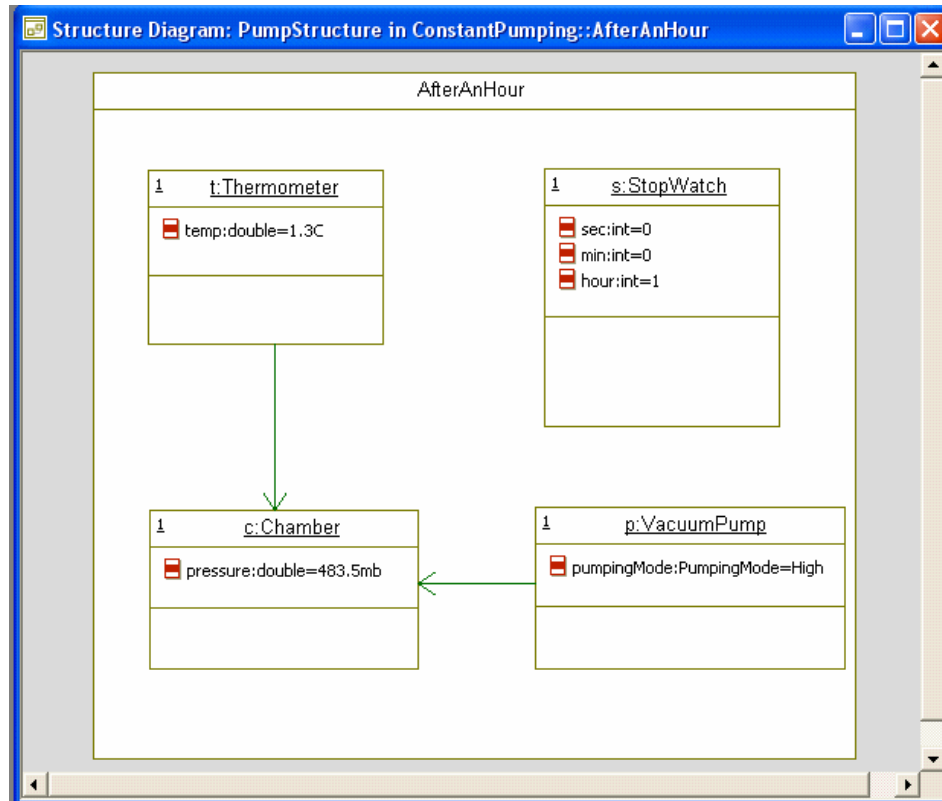
Alternatively, you can create a new stereotype for the class (`<<snapshot>>`), then set this property to `Checked`.

3. In this package, each phase is represented by a different class. For the initial conditions, create a class called `InitialConditions`.

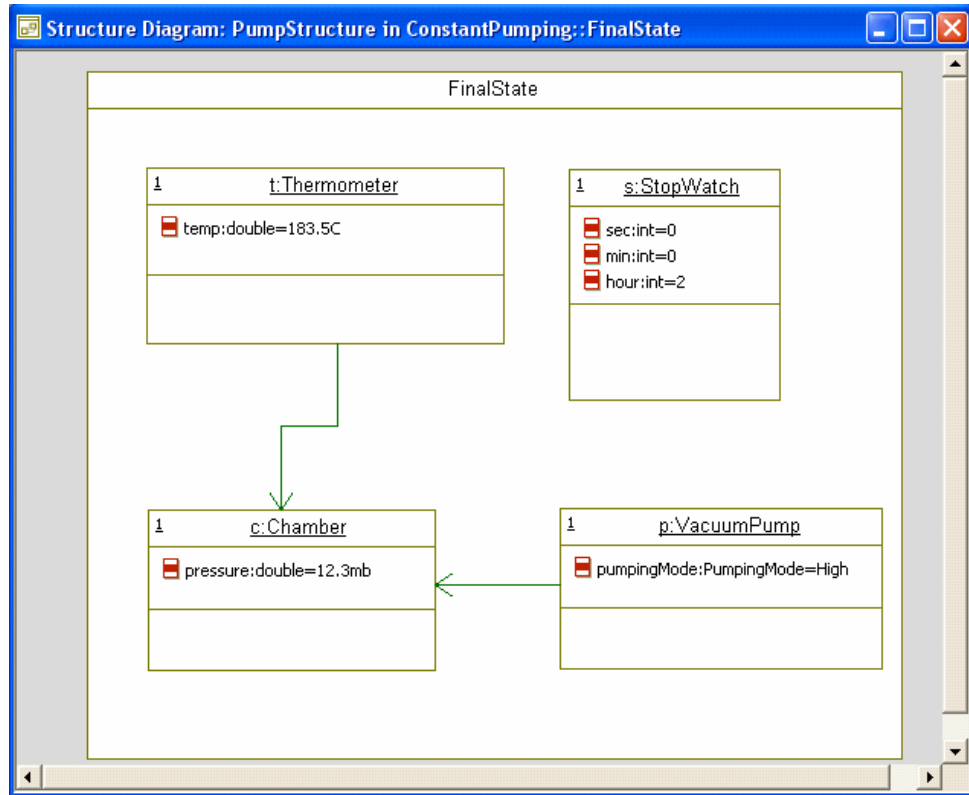
4. Add a structure diagram to `InitialConditions` and add the elements (and their attribute values) to the diagram, as shown in the following figure.



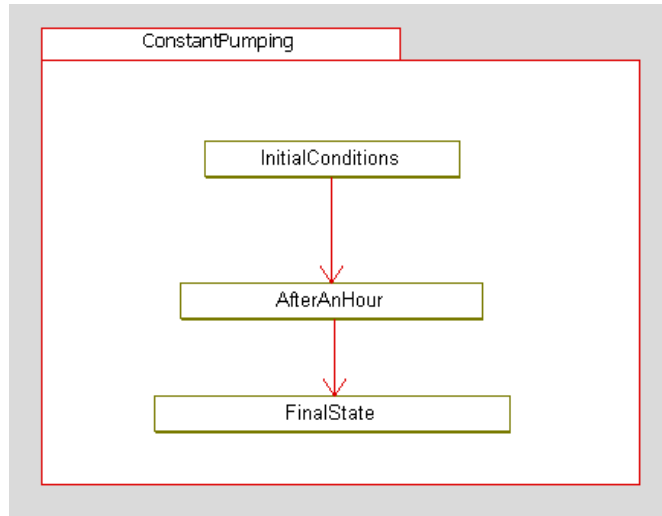
5. To show the conditions after an hour, copy the `InitialConditions` class and rename it `AfterAnHour`. Specify the attribute values for this stage in the process. The following figure shows the attribute values after the pump has been running for an hour.



6. To show the final values for the system, copy the `InitialConditions` class and rename it `FinalState`. Specify the attribute values for this stage in the process. The following figure shows the final values.



7. To show the order and transitions between snapshots, you can draw a simple OMD, as shown in the following figure.

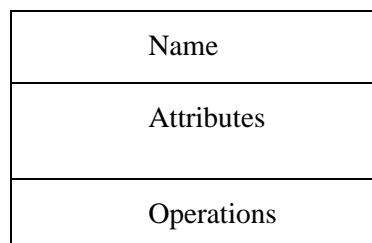


Classes

Classes can contain attributes, operations, event receptions, relations, components, superclasses, types, actors, use cases, diagrams, and other classes. The browser icon for a class is a three-compartment box with the top, or name, compartment filled in. To create a class, follow these steps:

1. In the **Drawing** toolbar, click the **Class** tool.
2. Click, or click-and-drag, in the drawing area.
3. Edit the default class name, then press **Enter**.

In the OMD, a class is shown as a rectangle with three sections, for the name, attributes, and operations. You can select and move the line separating the attributes and operations to create more space for either compartment.



If you shrink the box vertically, the operations and attributes sections disappear and the class graphic shows only the class name. The attributes and operations reappear if you enlarge the drawing.

When you rename a class in the OMD editor, the class name is changed throughout the model.

See [Classes and Types](#) for detailed information about classes.

Composite Classes

Instances in a composite class are called *parts*. To identify a component in code (actions or operations), use the expression *instance-of-composite.name-of-part*. The multiplicity of a component is relative to each instance of the composite containing it. For example, each car has one engine.

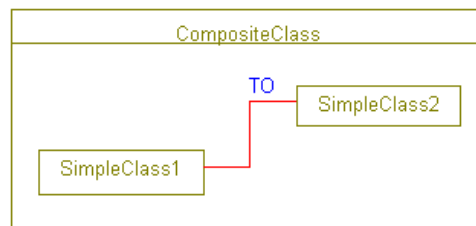
If the multiplicity is well-defined (such as 1 or 5), Rhapsody creates the components at run time, when the composite is instantiated. If an association is instantiated by a link, Rhapsody initializes the association at run time.

When a composite is destroyed, it destroys all its components.

To create a composite class, follow these steps:

1. Click the **Composite Class** tool.
2. Click in the diagram, or click-and-drag to create the composite class. The new composite class is displayed in the diagram.

Because a composite class is a container class, you can create objects and relations inside it, as shown in the following figure.



A composite class uses the same Features dialog box as objects and parts (see [Defining the Features of a Class](#)).

Another way of having the functionality of a composite class is to use a *composition*. See [Using Composition Associations](#) for more information.


Packages

In Rhapsody, every class belongs to a package. Classes drawn explicitly in a package are placed in that package in the model. Classes not drawn explicitly in a package are placed in the default package of the diagram. If you move a class to a package, it is also placed in that package in the model. If you do not connect this diagram to a package with the browser, Rhapsody assigns the diagram to the default package of the model.

For recommendations on ways of organizing a model into packages, refer to the *Rhapsody Team Collaboration Guide*.

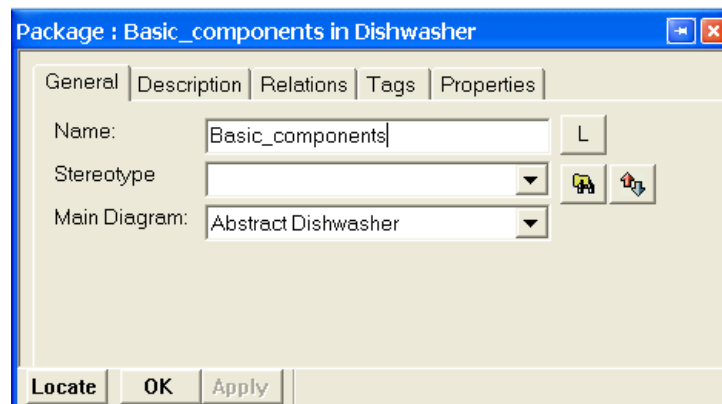
Creating a Package

To draw a package in the diagram, follow these steps:

1. Select the **Package**  icon.
2. Click once in the diagram. Now you must define the package using the Features dialog box.
3. Right-click the package and select **Features** from the menu.

Defining the Features of a Package

The Features dialog box allows you to define the characteristics of a package, such as its Name or Main Diagram, as shown in the following example.



- ◆ **Name**—Specifies the name of the package. Package names cannot contain spaces or begin with numbers.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.

- ◆ **Stereotype**—Specifies the stereotype of the package, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that package stereotypes are not constructive (see [Constructive Dependencies](#)).

- ◆ **Main Diagram**—Specifies the main diagram.

The **Description** tab allows you to write a detailed description of the package. The **Relations** tab lists all of the relationships of the package. The **Tags** tab lists the available tags for this package. The **Properties** tab enables you to define code generation properties for the package.

Inheritance


Inheritance is the “mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior.” Inheritance is also known as generalization, or a “taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed.” (Both references are from the UML Specification, v1.3.)

Creating an Inheritance

You can create inheritance by drawing an inheritance arrow between two classes or by using the browser.

Using an Inheritance Arrow

To create an inheritance arrow between two classes, follow these steps:

1. Click the **Inheritance**  icon.
2. Click in the subclass.
3. Move the cursor to the superclass and click once to end the arrow.

An inheritance arrow points from the subclass to the superclass, with a large arrowhead on the superclass end.

The browser icon for a superclass is an inheritance arrow:

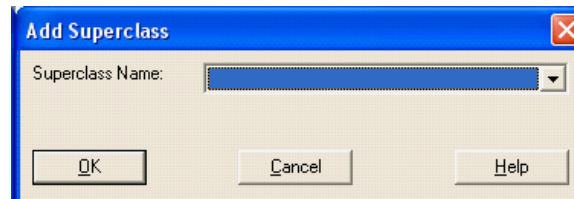
- ◆ The icon for the SuperClass category is a black arrow.
- ◆ The icon for an individual SuperClass is a blue arrow.

Double-clicking a SuperClass icon in the browser opens the Features dialog box for the superclass.

Using the Browser

To create inheritance using the browser, follow these steps:

1. Right-click a class.
2. Select **Add New > SuperClass** from the pop-up menu. The Add Superclass dialog box opens, as shown in the following figure.



3. Use the drop-down list to specify the superclass.
4. Click **OK** to apply your changes and close the dialog box.

Inheriting from an External Class

To inherit from a class that is not part of the model, set the `CG::Class::UseAsExternal` property for the superclass to `Checked`. This prevents code from being generated for the superclass.

To generate an `#include` of the superclass header file in the subclass, do one of the following:

- ◆ Add the external element to the scope of some component.
- ◆ Map the external element to a file in the component.
- ◆ Set the `CG::Class::FileName` property for the superclass to the name of its specification file (for example, `super.h`). That file is included in the source files for classes that have relations to it. If the `FileName` property is not set, no `#include` is generated.

Another way to inherit from an external class is to exclude the external class from the code generation scope. For example, if you want a class to extend the Java class `javax.swing.JTree` without actually importing it, you can follow these steps:

1. Draw a package `javax`.
2. Draw a nested package `swing` inside `javax`.
3. Draw a class `JTree` inside the `swing` package.
4. Exclude the `javax` package from the component (do not make it one of the selected elements in the browser). This prevents the component from generating code for anything in the `javax` package.

This gives the rest of the model the ability to reference the `JTree` class without generating code for it. In this way, a class in the model (for example, `MyJTree`) can inherit from `javax.swing.JTree`. If the subclass is public, the generated code is as follows:

```
import javax.swing.JTree;
...
public class MyJTree extends JTree {
    ...
}
```

If you need a class to import an entire package instead of a specific class, add a dependency (see [Dependencies](#)) with a stereotype of «Usage» to the external package, in this case `javax.swing`. The generated file will then include the following line:

```
import javax.swing.*
```

See [External Elements](#) for more information on using external elements.

Realization

Rhapsody allows you to specify a realization relationship between an interface and a class that implements that interface. This type of relationship is specified using the realization connector on the **Drawing** toolbar for object model diagrams.

Note

Realization is a "new term" based on the generalization element. This means that it is also possible to select a generalization element in a diagram, and select **Change To > Realization** from the context menu.

Code Generation for Realization Relationships

The realization connector only serves a visual purpose when used in an object model diagram. The code generation for realization relationships is not determined by the connector used between the class and the interface, but by the application of the *interface* stereotype to the class element in the diagram that represents the interface.

If you apply the *interface* stereotype to a class element, then the appropriate code will be generated for the interface and the implementing classes in Rhapsody in Java and Rhapsody in C.

For details regarding the code generation for realization relationships in C, see [Components-based Development in RiC](#).

Associations

In previous versions of Rhapsody, the term “relations” referred to all the different kinds of associations. Note that the term “relations” refers to all the relationships you can define between elements in the model (not just classes)—associations, dependencies, generalization, flows, and links.

Associations are links that allow related objects to communicate. Rhapsody supports the following types of associations:

- ◆ **Bi-directional association**—Both objects can send messages back and forth. This is also called a *symmetric association*. See [Using Bi-Directional Associations](#) for more information.
- ◆ **Directed association**—Only one of the objects can send messages to the other. See [Using Directed Associations](#) for more information.
- ◆ **Aggregation association**—Defines a whole-part relationship. See [Using Aggregation Associations](#) for more information.
- ◆ **Composition aggregation**—Defines a relationship where one class fully contains the other. See [Using Composition Associations](#) for more information.

Using Bi-Directional Associations

Bi-directional (or *symmetric*) *associations* are the simplest way that two classes can relate to each other. A bi-directional association is shown as a line between two classes, and can contain control points. The classes can be any mix of classes, simple classes, or composite classes.

When you create an association or an aggregation association between two classes and give it a role name that already exists, you have created another view of an existing relation.

In previous versions of Rhapsody, an association was described by one or two association ends. An association can be composed of the following elements:


- ◆ **Association ends**—The associated objects
- ◆ **Association element**—A view of the association as a whole
- ◆ **Association class**—An association element that has class characteristics (attributes and operations)

Note

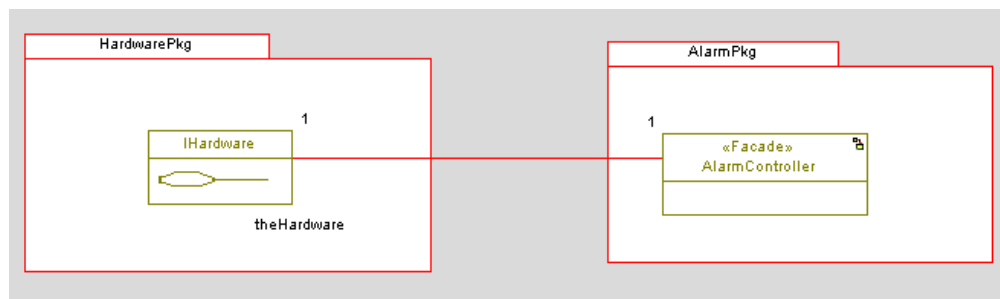
If you draw an anchor from a class to a relation (association, aggregation, or composition), it semantically implies that the class is an association class for this relation. Removing the icon changes the association class into a regular class.

Creating a Bi-Directional Association

To create a bi-directional association between classes, follow these steps:

1. Click the **Create Association** icon .
2. Click in a class.
3. Click in another class.

In this example note the bi-directional Association line between two classes.

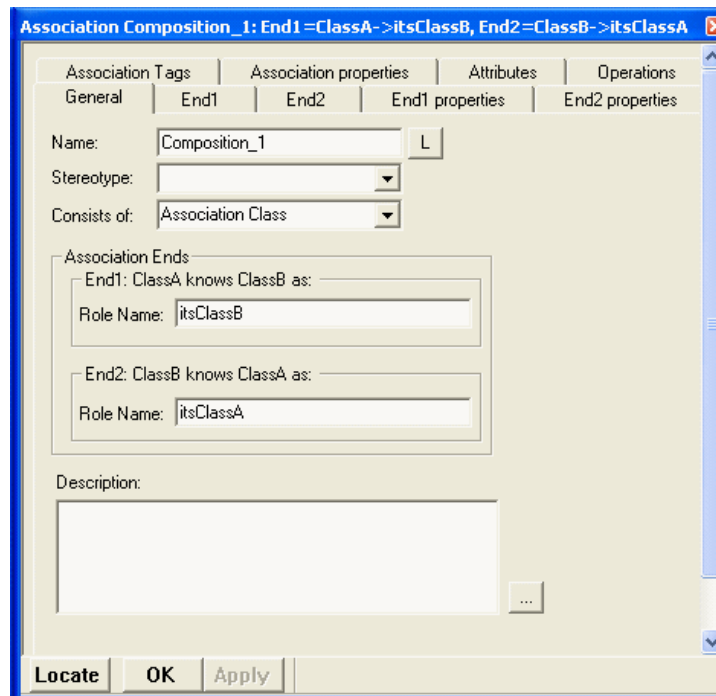


Note the following:

- ◆ Associations specify how classes relate to each other with role names. The relative numbers of objects participating is shown with multiplicity.
- ◆ You can move an association name freely.
- ◆ If you remove the class at one end of an association from the view, the association is also removed from the view. If you delete a class at one end of an association from the model, the association is also deleted.
- ◆ The role names and multiplicity are set in the Features dialog box for the association. To edit a role name or multiplicity, double-click it.
- ◆ If you move an association line from between class x and class y to between class x and class z , where z is a subclass of y , it is removed from y . But if z is a superclass of y , it remains because all relationships with a superclass are shared by their subclasses. If z and y are independent, Rhapsody moves it from y to z .

Modifying the Features of an Association

The Features dialog box enables you to change the features of an association, such as what it consists of (for example, two ends or a single end) and its association ends. The following figure shows the Features dialog box for a bi-directional association.



A bi-directional association has the following features:

- ◆ **Name**—Specifies the name of the association.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the association, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.

- ◆ **Consists of**—Specifies whether the association consists of:
 - A single association end (**End <X>**)
 - Two given ends (**Both Ends**)
 - An association element and two association ends (**Association Element**)
 - An association class and two association ends (**Association Class**)

Note: There is no representation of the association class in the diagram, nor is there code generation for the association class. The only representation of the association class is in the **Consists of** field.

- ◆ **Association Ends**—Specifies the ends of the association. If only one end is specified, the **Role Name** field for End2 field is grayed out.

Using this group box, you can change the role name of each enabled end. An *enabled end* is an end that is part of the specification of the association. The label under this field contains the type of the association end (the class to which the end is connected), the navigability of the end, and its aggregation kind. For a non-existing end, this label contains only “Role of.”

- ◆ **Description**—Describes the association. This field can include a hyperlink. See [Hyperlinks](#) for more information.

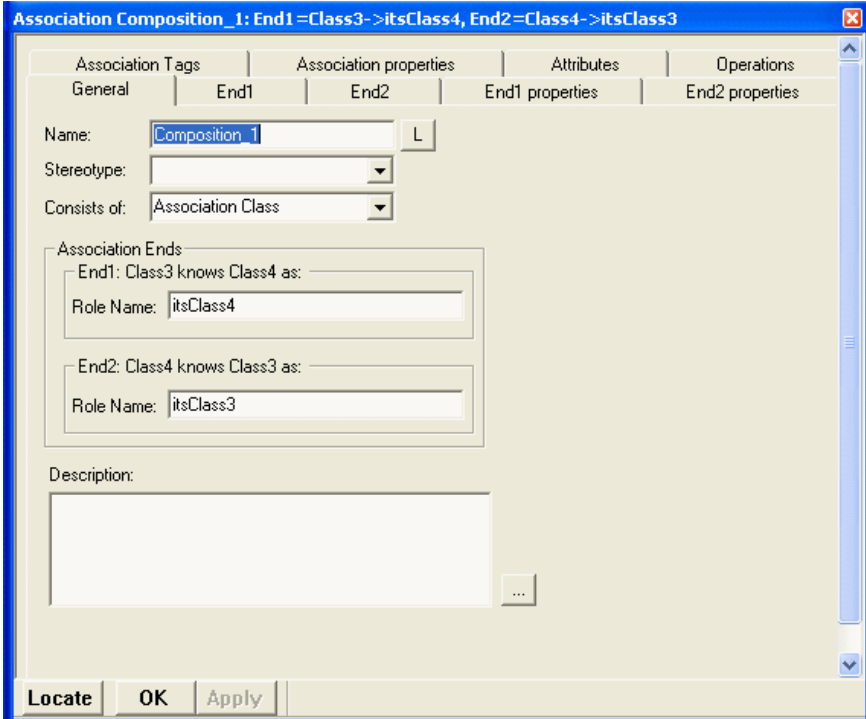
Note

If the association class or element does not exist, the **Name**, **Stereotype**, **Label**, and **Description** fields are disabled.

In addition to the **General** tab, the Features dialog box for an association contains the following tabs:

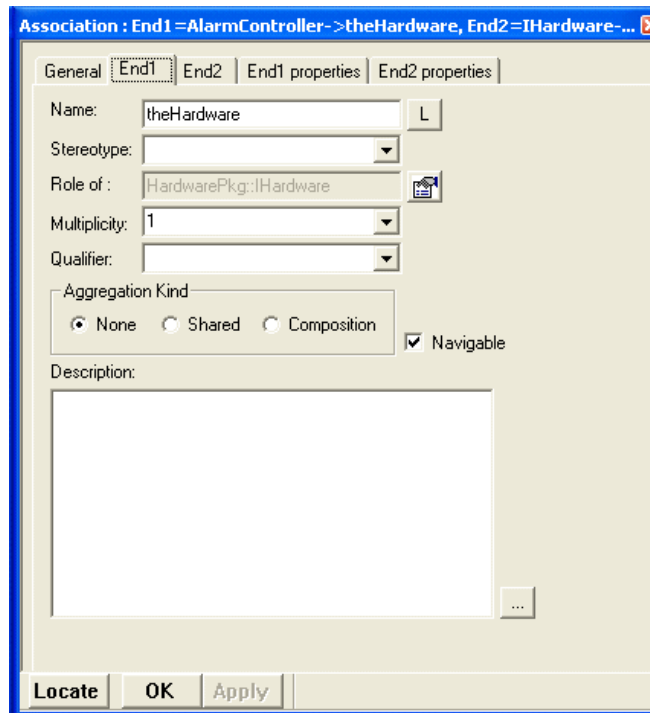
- ◆ **End1** or **End2**
- ◆ **End1 properties** or **End2 properties**

If the **Consists of** field is set to **Association Class**, the dialog box also includes tabs for attributes and operations, as shown in the following figure.



The End1 and End2 Tabs

The **End1** and **End2** tabs enable you to specify features of the individual ends of the association. The following figure shows an **End1** tab.



The **End1** and **End2** tabs contain the following fields:

- ◆ **Name**—Specifies the name of the element.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.

- ◆ **Role of**—A read-only field that specifies the class, actor, or use case that plays a role in the association.
- ◆ **Multiplicity**—Specifies the number of occurrences of this instance in the project. Common values are one (1), zero or one (0,1), or one or more (1..*).

- ◆ **Qualifier**—Shows the attributes in the related class that could function as qualifiers.

A *qualifier* is an attribute that can be used to distinguish one object from another. For example, a PIN number can serve as a qualifier on a `BankCard` class. If a class is associated with many objects of another class, you can select a qualifier to differentiate individual objects. The qualifier becomes an index into the multiple relation. Adding a qualifier makes the relation a qualified association.
- ◆ **Aggregation Kind**—Specifies the type of aggregation. The possible values are as follows:
 - **None**—No aggregation.
 - **Shared** (empty diamond)—Shared aggregation (whole/part relationship).
 - **Composition** (filled diamond)—Composition relationship. The instances of the class at this end contains instances of the class at the other end as a part. This part cannot be contained by other instances.
- ◆ **Navigable**—Specifies whether the association allows access to the other class. Both ends of a bi-directional association are navigable. In a directed association, the element that has the arrow head is navigable; the other end is not. See [Using Directed Associations](#) for more information.

The navigability of an association influences the appearance of the association arrow in the diagram. If one end of a symmetric association is navigable and the other is not, the association line includes an arrow head.
- ◆ **Description**—Describes the association. This field can include a hyperlink. See [Hyperlinks](#) for more information.

Similarly, the **End2** tab shows the features for the second end of the association.

Note that if an end is read only, its feature fields are also read-only.

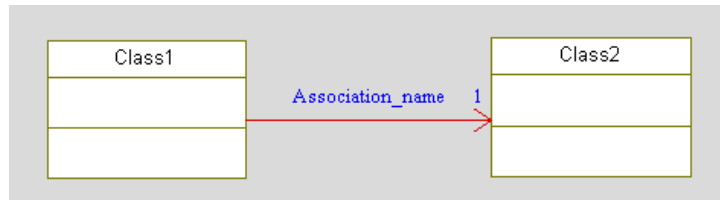
The End1 and End2 Properties Tabs

The **End1 properties** and **End2 properties** tabs enable you to specify properties for the individual ends of the association.

Refer to the *Rhapsody Properties Reference Manual* for detailed information on using the Rhapsody properties.


Using Directed Associations

In a *directed* (or *one-way*) association, only one of the objects can send messages to the other. It is shown with an arrow, as shown in the following figure.



Creating a Directed Association

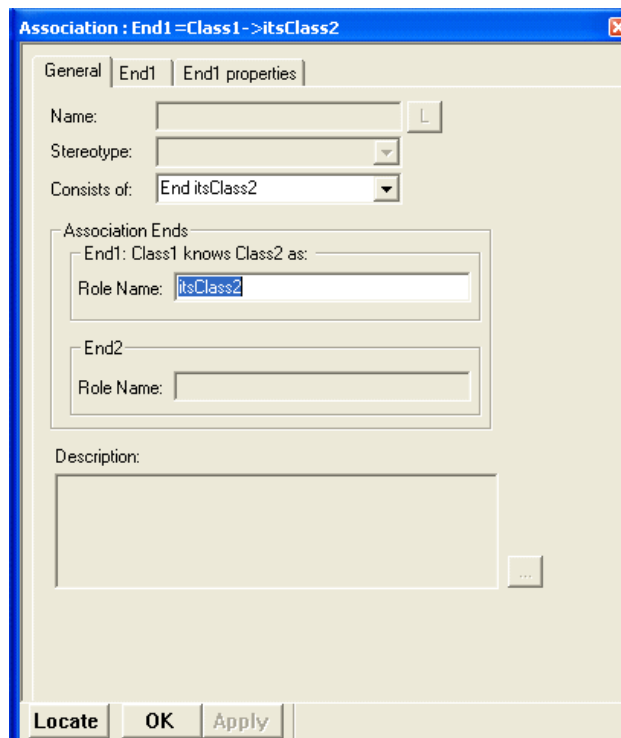
To create a directed association, follow these steps:

1. Click the **Directed Association** icon .
2. Click in the source class.
3. Click in the destination class.

Rhapsody draws an arrow between the two objects, with the arrow head pointing to the target object.

Defining the Features of a Directed Association

The Features dialog box enables you to change the features of a directed association, such as what it consists of (for example, two ends or a single end) and its association ends. The following figure shows the Features dialog box for a directed association.



The Features dialog box for a directed association is the same as the Features dialog box for a bi-directional association (see [Modifying the Features of an Association](#)), but the available tabs are different. As shown in the figure, a directed association has one role name and one multiplicity.

If the directed association is not named (as shown in the figure), the **Consists of** field is set to **End <X>** and the dialog box contains only the tabs **General**, **End1**, and **End1 properties**.

However, for a named directed association, the **Consists of** field of the Features dialog box is set to **Association Element**, which means there is one more non-navigable end. The dialog box contains the following additional tabs:

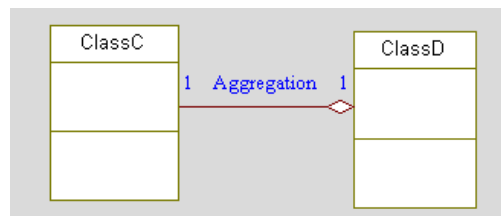
- ◆ **Association Tags**—Specifies the tags that can be applied to this association. See [Using Profiles](#) for more information on tags.
- ◆ **Association properties**—Specifies the properties that affect this association.

Using Aggregation Associations

Associations and aggregation associations are similar in usage. An association portrays a general relationship between two classes; an *aggregation association* shows a whole-part relationship. When you create an association or an aggregation association between two classes and give it a role name that already exists, you have created another view of the existing association.


An aggregation association is a whole-part relationship similar to the relationship between a composite class and a part. Other than their graphic representations, these differ mainly in that the composite/component relationship implies a whole lifetime dependency. Parts are created with their composites and are destroyed with them.

An aggregation association is shown as a line with a diamond on one end. The side with the diamond indicates the whole class, whereas the side with the line is the part class. In the following sample Aggregation Association, the diamond is placed at the first point of the aggregation:



Creating an Aggregation Association

To create an aggregation, follow these steps:

1. Click the **Aggregation** icon .
2. Click in the class that represents the whole.
3. Click in the class the represents the part.

Defining the Features of an Aggregation Association

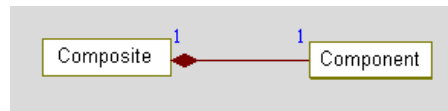
The Features dialog box for an aggregation is the same as that for a bi-directional association. See [Modifying the Features of an Association](#) for more information.

Note that for an aggregation association:

- ◆ Both ends are navigable.
- ◆ The **Aggregation Kind** value for the diamond end of an aggregation association is set to **None**; the other end must be set to **Shared**.


Using Composition Associations

Another way of drawing a composite class is to use a composition association. A *composition association* is a strong aggregation relation connecting a composite class to a part class (component). The notation is a solid diamond at the composite end of the of the relationship, as shown in the following figure.



The composite class has the sole responsibility of memory management for its part classes. As a result, a part class can be included in only one composite class at a time. A composition can contain both classes and associations.

Creating a Composite Association

1. Click the **Composition** icon .
2. Click the composite class.
3. Click the part class.
4. If desired, name the composition.
5. Press **Enter** to dismiss the text box.

Defining the Features of a Composition Association

The Features dialog box for compositions is the same as that for associations. See [Modifying the Features of an Association](#) for detailed information about the available fields.

Note that for a composition association:

- ◆ Both ends are navigable.
- ◆ The **Aggregation Kind** value for the filled-diamond end of the composition line is set to **None**; the other end must be set to **Composition**.

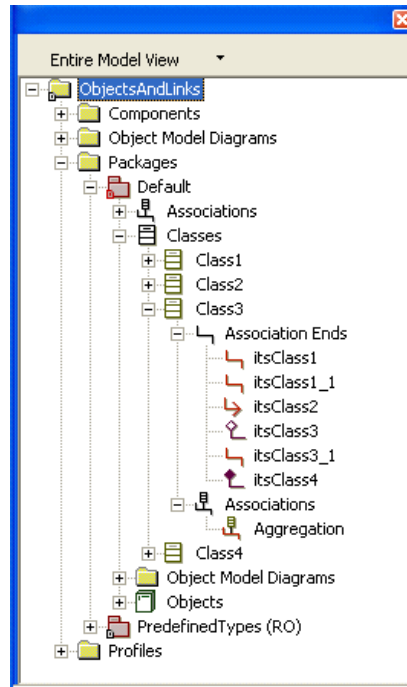
Displaying Associations in the Browser

An association can be represented in the browser as:

- ◆ A single association end
- ◆ Two association ends
- ◆ An association element and two association ends

- ◆ An association class and two association ends

Associations are listed in the browser under the category `Association Ends` under the owning class, as shown in the following figure. Note that the browser display includes separate icons for each association type.



Implementing Associations

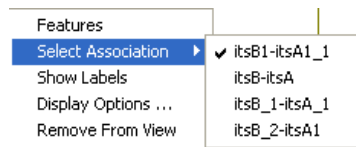
Rhapsody implements associations using containers. *Containers* are objects that store collections of other objects. To properly generate code for associations, you must specify the container set and type within that set you are using for the association.

Generally, you use the same container set for all associations. You specify the container set using the `CG::Configuration::ContainerSet` property. There are many options, depending on the language you are using. You can assign various container types, as defined in the selected container set, to specific relations. Container types include `Fixed`, `StaticArray`, `BoundedOrdered`, `BoundedUnordered`, `UnboundedOrdered`, `UnboundedUnordered`, and `Qualified`, among others. In addition, you can define your own container type called `User`. You specify the container type using the `Implementation` and `ImplementWithStaticArray` properties (under `CG::Relation`).

The Associations Menu

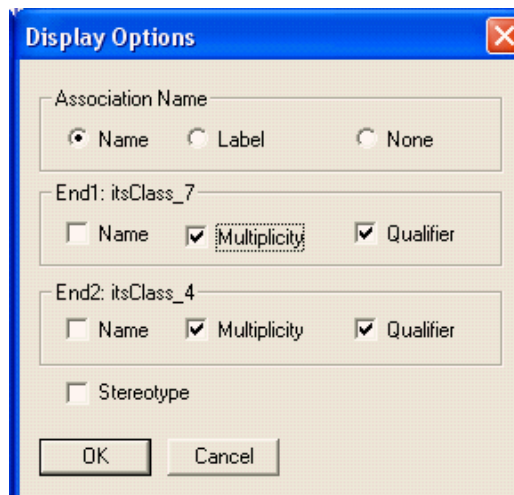
In addition to the common operations (see [Editing Elements](#)), the pop-up menu for associations includes the following options:

- ◆ **Select Association**—Lists the available associations for this class, as shown in the following figure. This functionality is useful when you have more than one association between the same elements.



See [Selecting Associations](#) for more information.

- ◆ **Show Labels**—Shows the role labels in the diagram.
- ◆ **Display Options**—Enables you to specify how associations are displayed in the diagram. By default, Rhapsody displays the name of the association, and the multiplicity and qualifiers for End1 and End2. The following figure shows the Display Options dialog box for an association.



Selecting Associations

In OMDs, you can select associations for classes that have more than one association defined between the same two classes. To do this, hold down the right mouse button over an association line to bring up the pop-up menu, then select **Select Association**.

Association names are displayed as follows:

- ◆ If the association has a name, the name is listed.
- ◆ If the association does not have a name and it is symmetric, the identifier uses the format `<role_1>-<role_2>`.
- ◆ If the association does not have a name and it is unidirectional, the identifier uses the format `-><role>`.
- ◆ If you select a different association from the pop-up list, the association line is directed to reference the selected association.

Links

A *link* is an instance of an association. In previous releases of Rhapsody, you could link objects in the model only if there were an explicit relationship between their corresponding classes. An association line between two objects meant the existence of two different model elements:

- ◆ An association between the objects' classes
- ◆ A link between the objects

Rhapsody 5.0 separates links from associations so you can have unambiguous model elements for links with a distinct notation in the diagrams. This separation enables you to:

- ◆ Specify links without having to specify the association being instantiated by the link.
- ◆ Specify features of links that are not mapped to an association.

In addition, Rhapsody supports links across packages, including code generation. To support this functionality, the default value of the property `CG::Component::InitializationScheme` was changed to `ByComponent`.

Creating a Link

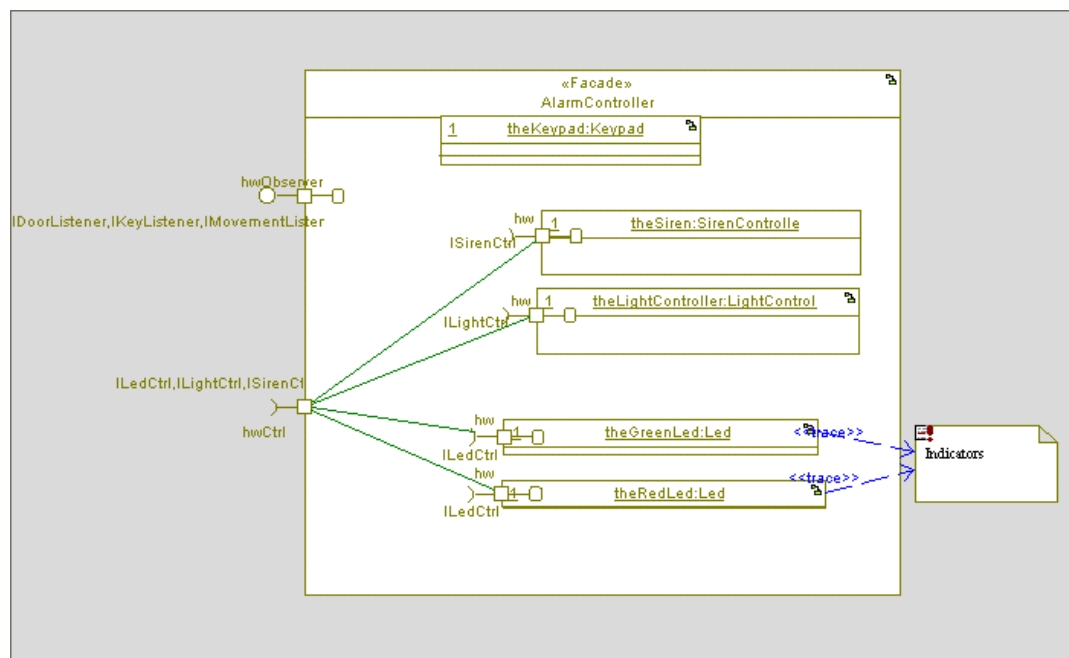
To create a link, there must be at least one association that connects one of the base classes of the type of one of the objects to a base class of the type of the second object.

To create a link, follow these steps:

1. Click the **Link** tool.
2. Click the first object.
3. Click the second object.
4. If desired, name the link and press **Enter**.

The new link is created in the diagram, and is displayed in the browser under the `Link` category. Note that you can drag-and-drop links in the browser to other classes and packages as needed; however, you cannot create links in the browser.

The following figure shows links in an OMD. Note that links shown in dark green to distinguish them from associations, which are drawn in red. In addition, the names and multiplicity of links are underlined.



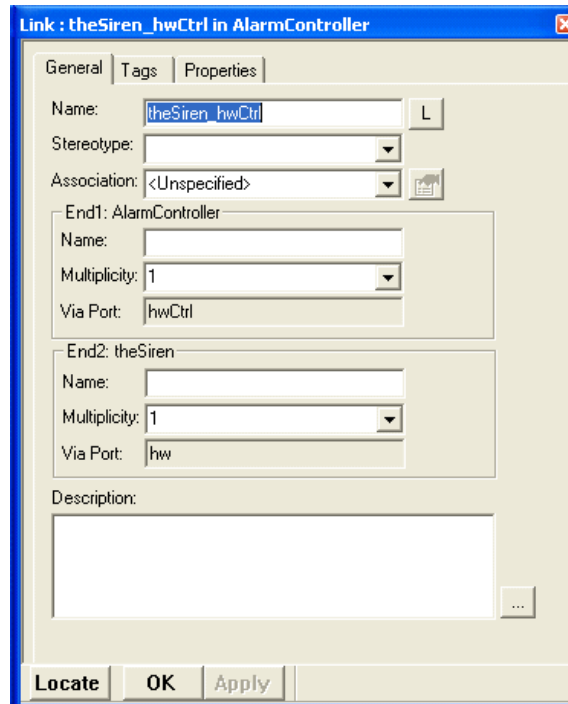
By default, the role name and multiplicity of a link are not displayed. Right-click the link and select **Display Options** from the pop-up menu to select the items you want to display. See [Using the Link Menu](#) for more information.

Note the following behavior:

- ◆ Links can be drawn between two objects or ports that belong to objects. One exception is the case when a link is drawn between a port that belongs to a composite class and its part (or a port of its part).
- ◆ When drawing a link, Rhapsody finds the association that can be instantiated by the newly drawn link and automatically maps the link to instantiate the association.
- ◆ If you draw a link between two objects with implicit type and there no associations to instantiate, Rhapsody automatically creates a new, symmetric association.

Modifying the Features of a Link

The Features dialog box enables you to change the features of a link, such as the association being instantiated by the link. The following figure shows the Features dialog box for a link.



The title bar is in the form:

Link: [end1 instance] (end1 role name) - [end2 instance] (end2 role name)

For example, a(itsA) - b(itsB).

A link has the following features:

- ◆ **Name**—Specifies the name of the link.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the link, if any. They are enclosed in guillemets, for example «S1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.

- ◆ **Association**—Specifies the association being instantiated by the link.

Rhapsody allows you to specify a link *without* having to specify the association being instantiated by the link. Until you specify the association, this field is set to <Unspecified>.

If you select <New> from the drop-down list, Rhapsody creates a new, symmetric association based on the link's data (its name and multiplicity). Note that once you specify an association for the link, you cannot change the link's role name or multiplicity (the corresponding fields of the Features dialog box are grayed out.).

To change the features of an association of which the link is an instantiation, you must invoke the Features dialog box of the association itself. Any changes you make to the association are instantly applied to links that are instantiated from it.

- ◆ **End1** and **End2**—Specifies the two ends of the link, including:
 - **Name**—The name of the link.
 - **Multiplicity**—The multiplicity of the link.
 - **Via Port**—The port used by the link, if any. This is a read-only field.
- ◆ **Description**—Describes the element. This field can include a hyperlink. See [Hyperlinks](#) for more information.

Using the Link Menu

In addition to the common operations (see [Editing Elements](#)), the pop-up menu for links includes the following options:

When you right-click a link, the pop-up menu contains the following options:

- ◆ **Select Link**—Lists the available links in the model. See [Referencing Links](#) for more information.
- ◆ **Select Association to instantiate**—Lists the associations available in the model so you can easily select one for the link to instantiate. See [Mapping a Link to an Association](#) for more information.

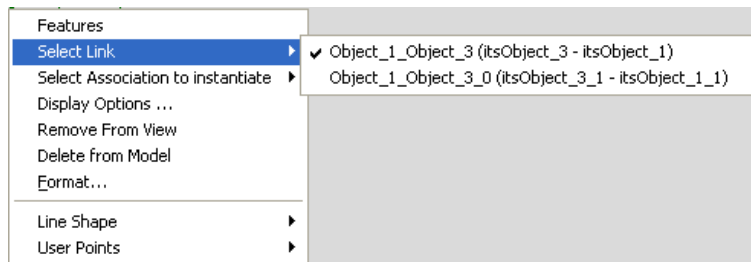
- ◆ **Show Labels**—Specifies whether to display element labels in the diagram.

Note that if you select this option and the link instantiates an association, the link ends will use the labels instead of the role names of the corresponding association ends.

- ◆ **Display Options**—Determines whether the names and multiplicities of link ends are displayed. See [Displaying Links](#) for more information.

Referencing Links

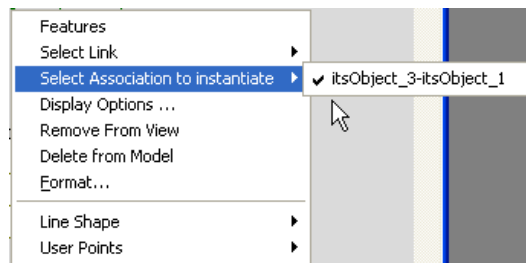
To map a link line to an existing link in the model, right-click the link and select **Select Link** from the pop-up menu, as shown in the following figure. This functionality is very useful when you have more than one link between the same elements.



To reference an existing link, select it from the submenu.

Mapping a Link to an Association

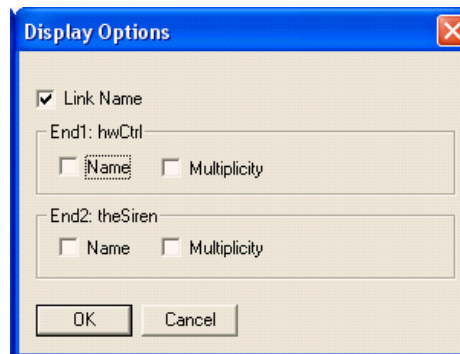
To map a link line to an existing association in the model, right-click the link and select **Select Association to instantiate** from the pop-up menu, as shown in the following figure.



To change the association, simply select a different association from the submenu. If you do this and the Active Code View is active, the corresponding code updates to reflect the change.

Displaying Links

By default, the names and multiplicities of link ends are not displayed. To change the display, right-click the link and select **Display Options** from the pop-up menu. The following figure shows the display options for links.



Note

Although the **Link Name** field is enabled by default, the “generated” link name is not shown on the diagram. However, if you change the name of the link, the new name will be displayed in the diagram.

Enable (check) the fields you want displayed in the diagram.

Using the Complete Relations Functionality

The following table shows the results of using the Complete Relations functionality with associations, generalizations, dependencies, and links.

Completing relations between these elements...	Results in...
Two classes	Rhapsody draws the associations, generalizations, and dependencies but not the links.
Two objects with implicit type	Rhapsody draws the associations, generalizations, dependencies, and links. If the link instantiates an association, the link is drawn, but the association is not.
Two objects with explicit type	Rhapsody draws only the links.

See [Completing Relations](#) for more information in the Complete Relations functionality.

Generating Code for Links

The code for run-time connection of objects is based on links. The connection code is generated when the following conditions are met:

- ◆ Links are specified with an association.
- ◆ The objects connected by the link has an owner (composite class or object) or are global (both objects are owned by a package).

If the objects are parts of a composite, the link is owned by the composite. When the objects are global, the link is owned by a package. Links across packages are initialized by the component.

- ◆ The link's package and objects are in the scope of the generated component.
- ◆ The `CG::Component::InitializationScheme` property for the component is set to `ByComponent` for links across packages.
- ◆ If more than one link exists between two objects over the same relation, Rhapsody arbitrarily chooses which link to instantiate. The packages that contain the objects are given priority in this decision.

Populating One-to-Many Associations with Objects

If you draw a one-to-many directed association between a class A and a class B, and create an object of A and several objects of B, you can connect the objects with a link that instantiates the association. The Rhapsody-generated code for such a relationship creates a container class for the one-to-many relationship in A, and creates objects of A and B. However, it does not necessarily populate A's container with the objects of B. When you model a relationship as one-to-*n*, Rhapsody instantiates *n* objects in the container. Rhapsody populates only associations with known multiplicity, and graphically shows when an association instance, or link, actually exists and when it does not.

You can populate a one-to-many container by creating the objects in source code and adding them to the container. However, you cannot model a generic one-to-many relationship and populate it with an unknown number of diagrammatically modeled objects. Therefore, it is not possible to populate a one-to-many relationship between classes drawn in one OMD with objects drawn in another OMD.

Restrictions

Note the following limitations and restrictions:

- ◆ Code generation for links across composite classes is not supported.
- ◆ You cannot make a link when one or both sides of the relation are classes (as opposed to objects).

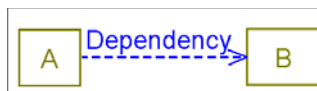
Dependencies

A *dependency* exists when the implementation or functioning of one element (class or package) requires the presence of another element. For example, if class C has an attribute a that is of class D, there is a dependency from C to D.

In the UML, a dependency is a directed relationship from a client (or clients) to a supplier stating that the client is dependent on, and affected by, the supplier. In other words, the client element requires the presence and knowledge of the supplier element. The supplier element is independent of, and unaffected by, the client element.

Creating a Dependency

A dependency arrow is a dotted line with an arrow. You can draw a dependency arrow between elements, or you can have one end attached and the other free. It can have a label, which you can move freely. If a dependency arrow is drawn to or from an element, it is attached to the element; the attached end moves with the attached border of the element.




Dependency arrows show that one thing depends on something else:

- ◆ An object that is a (logical) instantiation of another
- ◆ An object that creates or deletes another
- ◆ Constraints attached to an element
- ◆ A class that uses another class or package
- ◆ A package that uses another package or class

You can create a dependency in a diagram or in the browser.

Drawing the Dependency

To draw a dependency in the diagram, follow these steps:

1. Click the **Dependency** icon .
2. Click the dependent object.
3. Click the object on which it depends. This object also known as the *provider*.

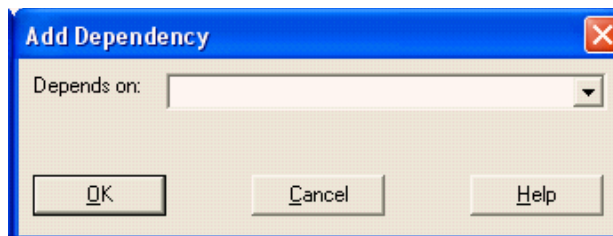
Note that you can create more than one dependency between the same two elements. For example, if you create one dependency from element *x* to element *y*, the default name of the dependency is *y*. If you create a second dependency between the same two elements, the second dependency is named *y_0* by default. To rename a dependency, do one of the following:

- ◆ Invoke the Features dialog box for the dependency, and type the new name in the **Name** field.
- ◆ In the browser, left-click the dependency whose name you want to change, and type the new name in the text box.

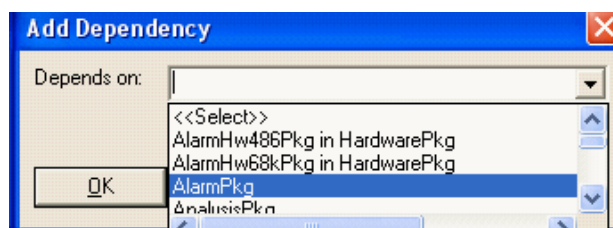
Creating the Dependency in the Browser

To create a dependency using the browser, follow these steps:

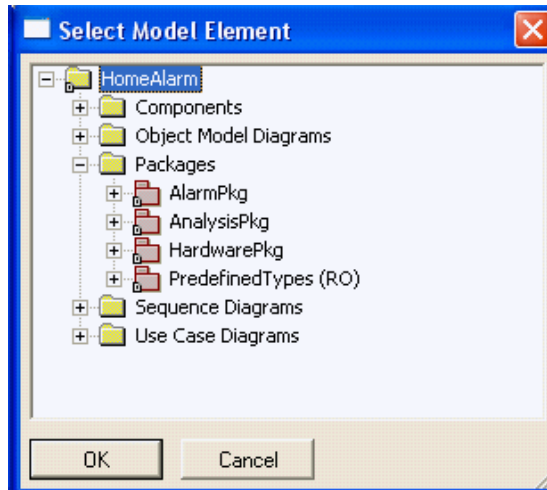
1. In the browser, right-click the element that depends on another element.
2. Select **Add New > Dependency** from the pop-up menu. The Add Dependency dialog box opens, as shown in the following example.



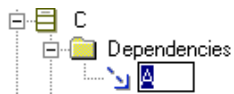
3. Use the drop-down list to select the element on which the specified element depends.



Click the <<Select>> line to open a browsable tree of the entire project, as shown in the following example.



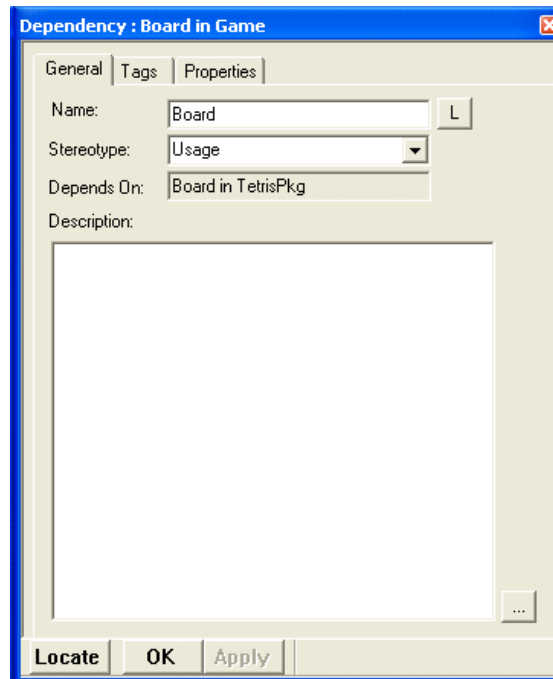
4. Highlight the appropriate element, then click **OK**.
5. Rhapsody creates the new dependency under the *Dependency* category for the dependent element, with an open text box so you can easily rename the dependency.



6. If desired, rename the dependency.

Modifying the Features of a Dependency

The Features dialog box enables you to change the features of a dependency, including its name and stereotype. The following figure shows the Features dialog box for a dependency.



A dependency has the following features:

- ◆ **Name**—Specifies the name of the dependency.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the dependency, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.

- ◆ **Depends On**—Specifies the class or package that provides information to the dependency.
- ◆ **Description**—Describes the dependency. This field can include a hyperlink. See [Hyperlinks](#) for more information.

Dependency Menu

In addition to the common operations (see [Editing Elements](#)), the pop-up menu for dependencies includes the following options:

- ◆ **Display Options**—Specifies how dependencies are displayed. The following figure shows the display options for dependencies.



- ◆ **Select Dependency**—Enables you to select a dependency. This functionality is useful when you have more than one dependency between the same elements.

Constructive Dependencies

Rhapsody supports the dependency stereotypes «Send», «Usage», and «Friend».

Note

If a class has a dependency on another class that is outside the scope of the component, Rhapsody does not automatically generate an `#include` statement for the external class. You must set the «Usage» stereotype and the `<lang>_CG::Class::SpecInclude` property for the dependent class.

Stereotypes are shown between guillemets (« . ») and are attached to the dependency line in the OMD, as shown in the following figure.



The **Properties** tab in the Features dialog box enables you to define the `UsageType` property for the dependency. This property determines how code is generated for dependencies to which a «Usage» stereotype is attached. The possible values for the `UsageType` property are as follows:

- ◆ **Specification**—An `#include` of the provider is generated in the specification file for the dependent.
- ◆ **Implementation**—An `#include` of the provider is generated in the implementation file for the dependent.
- ◆ **Existence**—A forward declaration of the provider is generated in the specification file for the dependent.


See [Defining Stereotypes](#) for more information on stereotypes.

Actors

An actor is a “coherent set of roles that users of use cases play when interacting with these use cases. An actor has one role for each use case with which it communicates.” (UML specification, version 1.3) An actor is a type of class with limited behavior. As such, it can be shown in an OMD.

Creating an Actor

To create an actor, follow these steps:

1. Click the **Actor** icon .
2. Click, or click-and-drag, in the diagram.

For a detailed explanation of actors, see [Actors](#). Note that an actor has a Features dialog box that is very similar to that of a class; see [Classes](#) for more information about the Features dialog box.

The Actor Menu

In addition to the common operations (see [Editing Elements](#)), the pop-up menu for actors includes the following options:

- ◆ **New Statechart**—Invokes the statechart editor (see [Statecharts](#))
- ◆ **New Activity Diagram**—Invokes the activity diagram editor (see [Activity Diagrams](#))
- ◆ **New Attribute**—Opens the Attribute dialog box (see [Defining the Attributes of a Class](#))
- ◆ **New Operation**—Opens the Operation dialog box (see [Defining Class Operations](#))
- ◆ **Generate**—Generates code for the actor (see [Basic Code Generation Concepts](#))
- ◆ **Edit Code**—Opens the actor code in a text editor (see [Editing Code](#))
- ◆ **Roundtrip**—Roundtrips edits made to generated code back into the model (see [The Roundtripping Process](#))
- ◆ **Open Main Diagram**—Opens the main diagram for the actor
- ◆ **Display Options**—Opens the Display Options dialog box
- ◆ **Cut**—Removes the actor from the view and saves it to the clipboard
- ◆ **Copy**—Saves a copy of the actor to the clipboard, while leaving it in the view

Flows and FlowItems

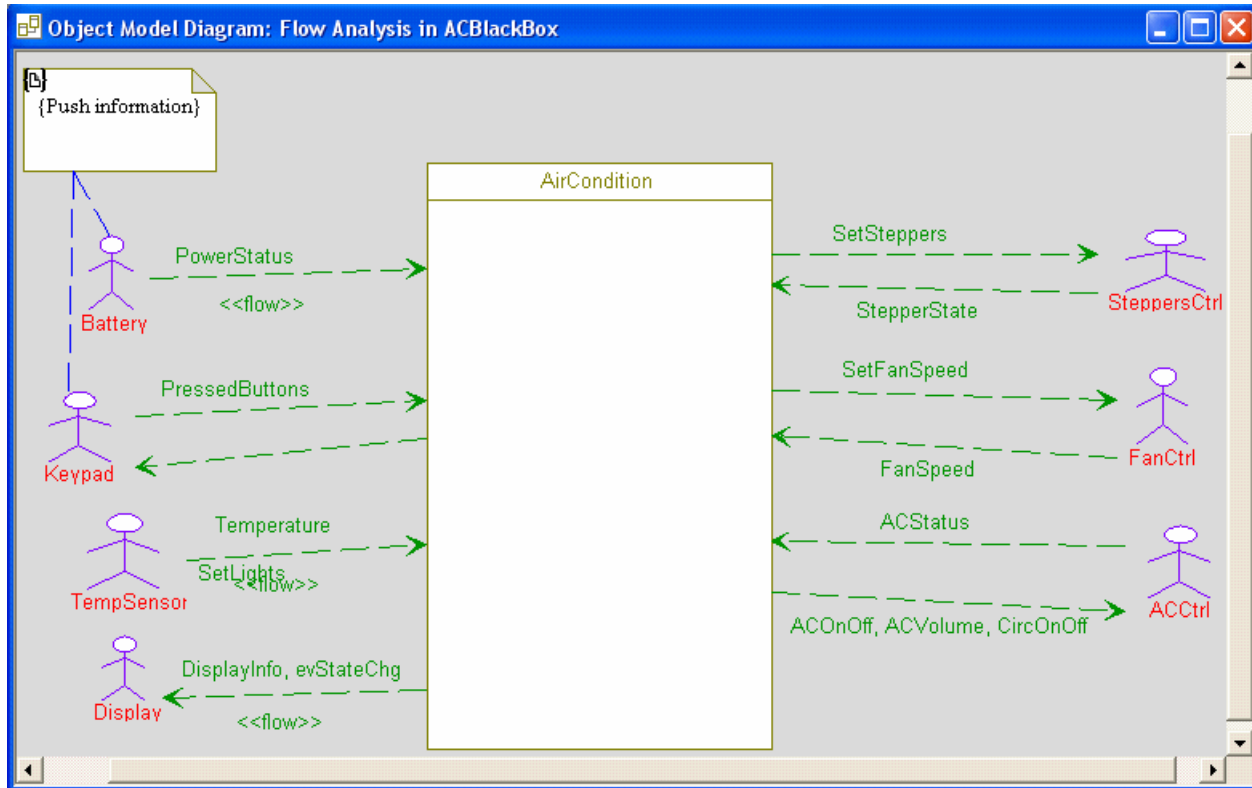
Flows and FlowItems provide a mechanism for specifying exchange of information between system elements at a high level of abstraction. This functionality enables you to describe the flow of data and commands within a system at a very early stage, before committing to a specific design. As the system specification evolves, you can refine the abstraction to relate to the concrete implementation.

Flows can convey FlowItems, classes, types, events, attributes and variables, parts and objects, and relations. You can draw flows between the following elements:

- ◆ Actors
- ◆ Classes
- ◆ Components
- ◆ Nodes
- ◆ Objects
- ◆ Packages
- ◆ Parts
- ◆ Ports
- ◆ Use cases

You can add flows to all the static diagrams supported by Rhapsody.


The flows in an object model diagram are shown below in this black-box representation of an air conditioning unit and all the actors that will interact with it. It includes all of the information that is passed either from an actor to the AC unit or from the AC unit to an actor.



Creating a Flow

Every static diagram toolbar includes an **Flow** tool, which is drawn like a link. *Static* (or *structural*) diagrams include object model diagrams, structure diagrams, use case diagrams, component diagrams, and deployment diagrams.

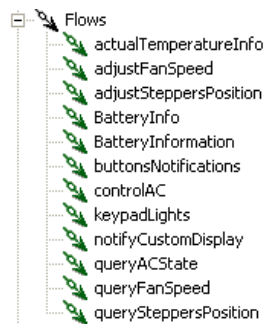
To create a flow, follow these steps:

1. In the **Drawing** toolbar, click the **Flow** icon  or choose **Edit > Add New > Flow**.
2. In the diagram, click near the first object to anchor the flow.
3. Click to place the end of the flow.
4. In the edit box, type the element conveyed by the flow, then press **Enter**.

By default, a flow is displayed as a green, dashed arrow with the keyword `«flow»` underneath it. To suppress the `«flow»` keyword, invoke the Display Options dialog box and disable the `<<flow>> keyword` check box.

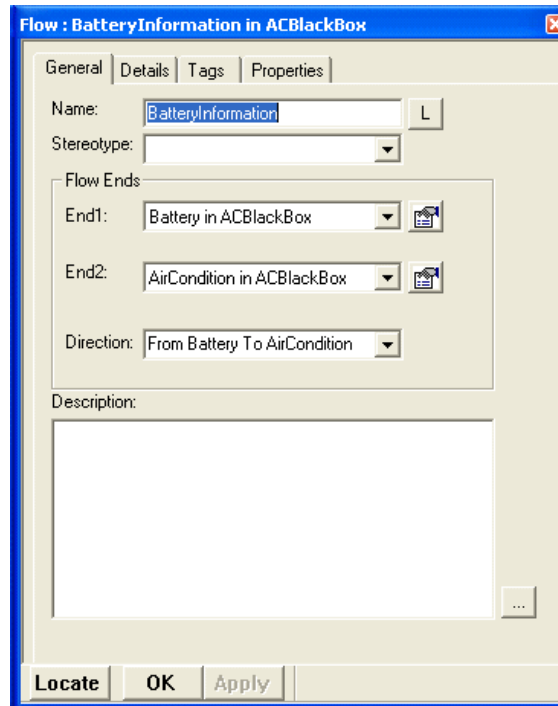
You can also control the display of the `<<flow>>` keyword for new flows by setting the `<Static diagram>::Flow::flowKeyword` Boolean property. Refer to the *Properties Reference Manual* for more information.

In the browser, flows are displayed under the `Flows` category, as shown in the following figure.



Features of a Flow

The Features dialog box enables you to change the features a flow, such as its name or flow ends. The following figure shows the **General** tab of the Features dialog box for flows.



A flow has the following features:

- ◆ **Name**—Specifies the name of the flow. By default, the flow is named using the following convention:

```
<source>_<target>[_###]
```

In this convention, the *source* and *target* are end1 and end2 of the flow, based on the direction (end1 is the source in bidirectional flows as well as flows from end1 to end2); *_###* specifies additional numbering when the name is already used.

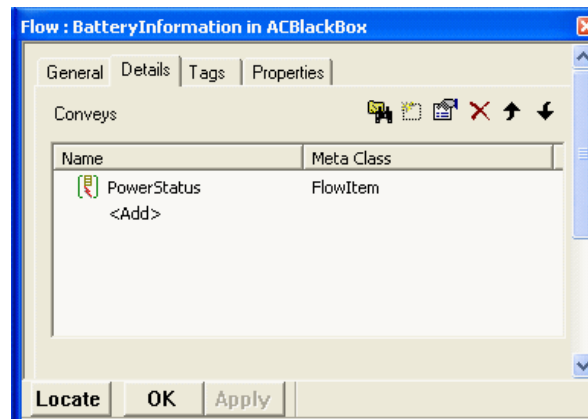
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the flow, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.

- ◆ **Flow Ends**—Specifies how the information travels, from the source (**End1**) to the target (**End2**). To change either end, use the pull-down list to select a new source or target from the selection tree.
- ◆ **Direction**—Specifies the direction in which the information flows. To invert the flow, or to make it bidirectional, use the appropriate value from the pull-down list.
- ◆ **Description**—Describes the flow. This field can include a hyperlink. See [Hyperlinks](#) for more information.

Conveyed Information

All the information elements that are conveyed on the flow are listed on the **Details** tab, as shown in the following figure.



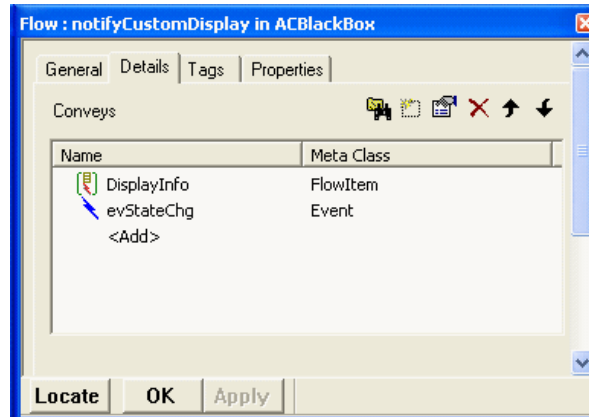
An information element can be any FlowItem, as well as elements that can realize a FlowItem—classes, events, types, attributes and variables, parts and objects, and relations.

This tab enables you to perform the following tasks on information elements:

- ◆ Add a new information element.
- ◆ Add an existing information element.
- ◆ Remove an information element.
- ◆ View the features of an information element.

See [Using FlowItems](#) for detailed information on manipulating information elements.

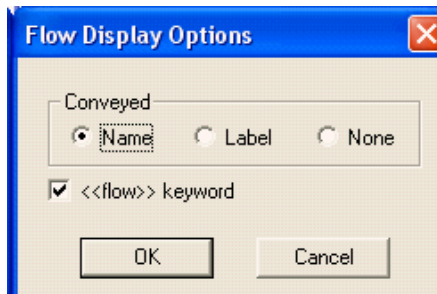
Note that you can specify multiple information elements using a comma-separated list. For example, in the OMD in the figure, the flow from the AC unit to the Display actor contains two information elements: the DisplayInfo FlowItem and the evStateChg event. The following figure shows the corresponding **Details** tab.



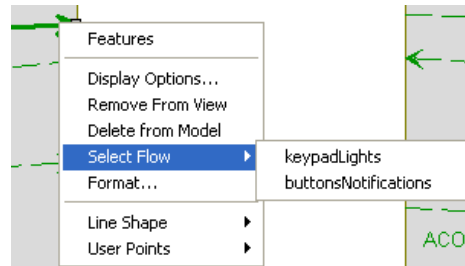
Flow Menu

In addition to the common operations (see [Editing Elements](#)), the pop-up menu for flows includes the following options:

- ◆ **Display Options**—Invokes the Display options dialog box for the flow, as shown in the following figure.



- ◆ **Select Information Flow**—Provides a list of the flows already defined between these ends so you can easily reuse flows in your model, as shown in the following figure.



Using FlowItems

A *FlowItem* is an abstraction of all kinds of information that can be exchanged between objects. It does *not* specify the structure, type, or nature of the represented data. You provide details about the information being passed by defining the classifiers that are represented by the *FlowItem*.

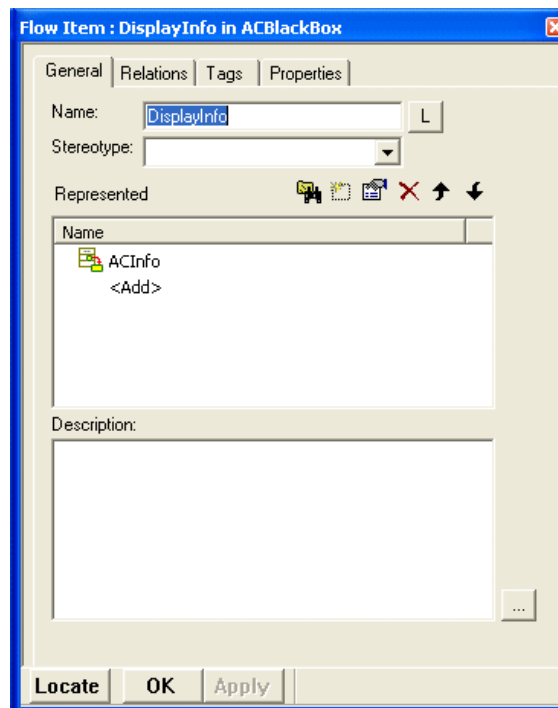
FlowItems can be decomposed into more specific FlowItems. This enables you to start in a very high level of abstraction and gradually refine it by representing the FlowItem with more specific items.

A FlowItem can represent either pure data, data instantiation, or commands (events). FlowItems can represent the following elements:

- ◆ Classes
- ◆ Types
- ◆ Events
- ◆ Other information items
- ◆ Attributes and variables
- ◆ Parts and objects
- ◆ Relations

Features of a FlowItem

To view the features of a particular information element, double-click the element in the list on the **Details** tab for the flow. The corresponding Features dialog box opens. The following figure shows the Features dialog box for a FlowItem.



A FlowItem has the following features:

- ◆ **Name**—Specifies the name of the FlowItem.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the FlowItem, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

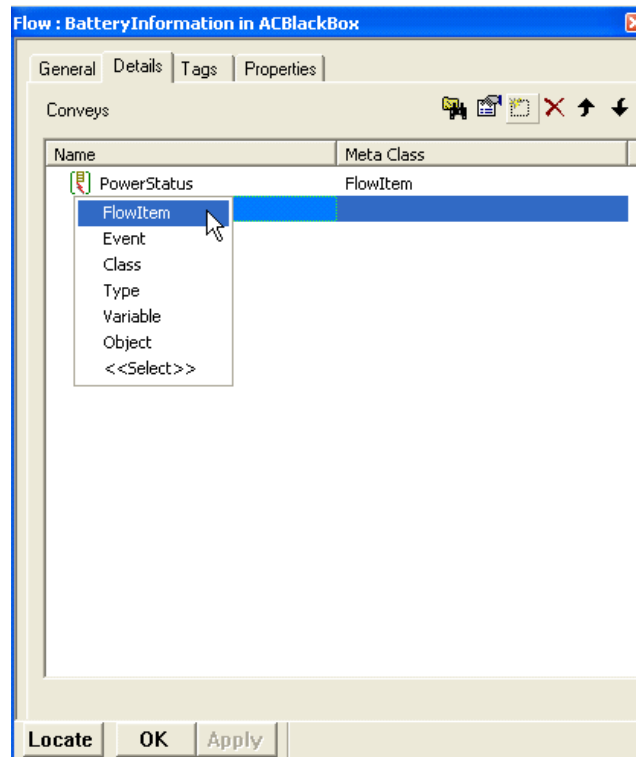
Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.

- ◆ **Represented**—Lists all the information elements that are represented by this FlowItem. In this example, the DisplayInfo FlowItem represents the ACInfo class.
- ◆ **Description**—Describes the information element. This field can include a hyperlink. See [Hyperlinks](#) for more information.

Adding a New Information Element

To add a new information element to the flow, follow these steps:

1. To create a new attribute, either click the **<Add>** row in the list of information elements, or click the **New** icon in the upper, right corner of the dialog box and select the appropriate element from the pop-up list.



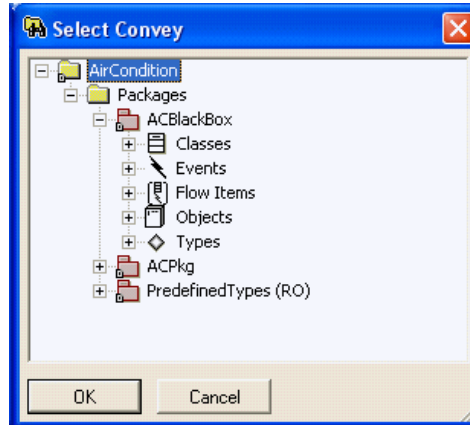
The new element is added to the list and its Features dialog box automatically opens.

2. Set the new values for the element.
3. Click **OK** to apply your changes and dismiss the Features dialog box for the new element.
4. Click **OK** to apply your changes and dismiss the Features dialog box for the flow.

Adding an Existing Information Element to the Flow

To add an existing information element to the flow, follow these steps:

1. On the **Details** tab for the flow, highlight the **<Add>** row and select the **<Select>** option in the pop-up menu. The Select Convey dialog box opens, as shown in the following figure.




2. Expand each subcategory as needed to select the information element from the tree, then click **OK**.
3. You return to the **Details** tab, where the specified information element now appears in the list of elements.
4. Click **OK** to apply your changes and close the dialog box.

Embedded Flows

In SysML notation, flows can be embedded in links. Rhapsody allows you to use this notation in object model diagrams.

Creating an Embedded Flow

To add a flow to a link, follow these steps:

1. Click the **Flow** icon .
2. Click the link to which you want the flow added.

Once the flow is created, it has the same features as an ordinary flow element, representing the flow of data between the two objects that are linked. Visually, the flow is displayed on top of the link, and it is depicted by an arrow.

To select the embedded flow element, double-click the arrow.

To move the embedded flow diagram element, drag the arrow to a new position on the link.

Changing the Flow Direction

To change the flow direction:

1. Select the flow.
2. From the context menu, select **Flip Flow Direction**.

Display Options for Embedded Flows

For an ordinary flow diagram element, the <flow> keyword is displayed alongside the flow element. To display this for embedded flows:

1. Select the flow.
2. Right-click and from the context menu, select **Display Options**.
3. Select **<flow> keyword or Stereotype** from the context menu.
4. Click **OK**.

Restrictions

Note the following restrictions and limitations:

- ◆ Flows cannot be animated.
- ◆ There is no code generation for flows.

Files

Rhapsody in C enables you to create model elements that represent files. A *file* is a graphical representation of a specification (.h) or implementation (.c) source file. This new model element enables you to use functional modeling and take advantage of the capabilities of Rhapsody (modeling, execution, code generation, and reverse engineering), without radically changing the existing files.

Note

Files are not the same as the file functionality in components that existed in previous versions of Rhapsody. To differentiate between the two, the new file is called `File in Package` and the old file element is called `File in Component`. A `File in Component` includes only references to primary model elements (package, class, and object) and shows their mapping to physical files during code generation.

A file element can include variables, functions, dependencies, types, parts, aggregate classes, and other model elements. However, nested files **are not** allowed.

Rhapsody supports the following modeling behavior for files:

- ◆ You can drag files onto object model diagrams and structure diagrams.
- ◆ If you use the FunctionalC profile, then the **File** tool is available on the **Drawing** toolbars for object model diagrams and structure diagrams.
- ◆ You can drag files onto a sequence diagram, or realize instance lines as files.
- ◆ A file can have a statechart or activity diagram.
- ◆ Files are implicit and always have a multiplicity of 1.
- ◆ Files are listed in the component scope and the initialization tree of a configuration. They have influence in the initialization tree only in the case of a **Derived** scope.
- ◆ Files can be defined as separate units, and can have configuration management performed on them. See [Using Project Units](#) for more information.
- ◆ Files can be owned by packages only.

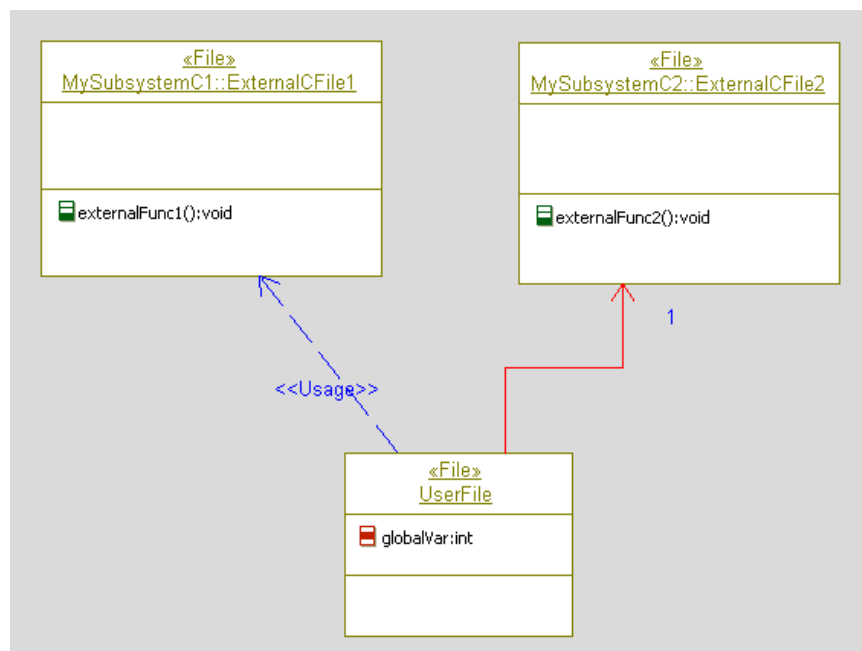
Creating a File

To create a file element, follow these steps:

1. Click the **File** icon in the **Drawing** toolbar, or select **Edit > Add New > File**.
2. Click, or click-and-drag, in the drawing area. By default, files are named `file_n`, where n is an integer greater than or equal to 0.
3. Edit the default name, then press **Enter**.

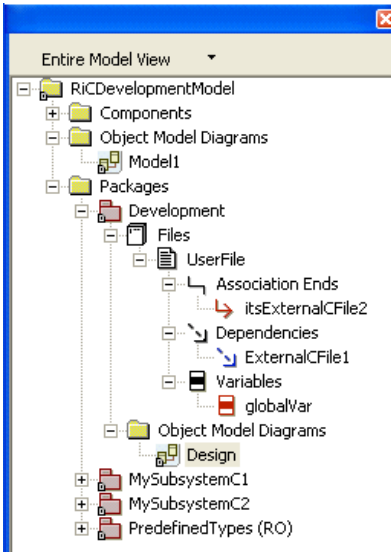
The file is shown as a box-like element in the diagram, with the `«File»` notation in the top of the box.

The following figure shows an OMD that contains files.



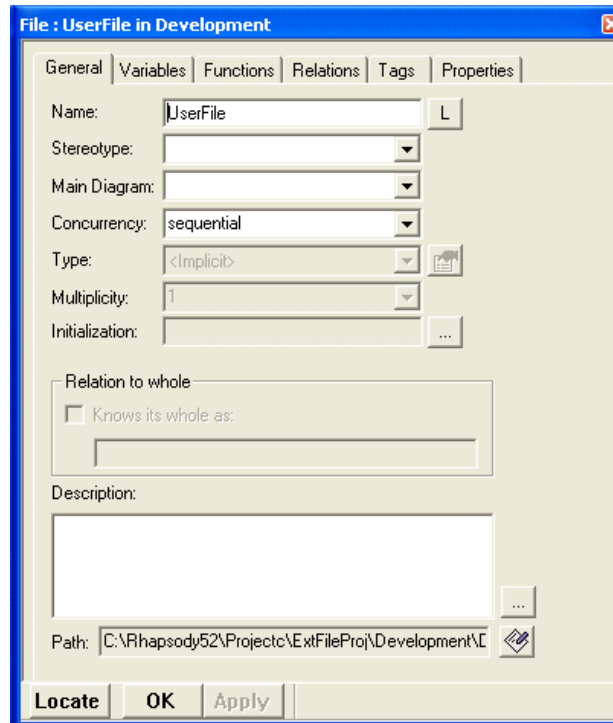
You can specify whether file variables and functions are displayed in diagrams using the **Display Options** feature. The Display Options dialog box for files is identical to that for classes, except the tab names are Variables instead of Attributes and Functions instead of Operations. See [Setting Display Options](#) for more information.

Files can be owned by packages only. File elements are listed in the browser under the `Files` category under the owning package, as shown in the following figure.



Modifying the Features of a File

The Features dialog box enables you to change the features of an file, including its name, stereotype, and main diagram. The following figure shows the Features dialog box for a file.



The **General** tab for a file is very similar to that of an object (see [Modifying the Features of an Object](#)), with the following differences:

- ◆ The **Type**, **Initialization**, **Multiplicity**, and **Relation to whole** fields are unavailable (grayed out). Multiplicity has no meaning with files, because a file is simply a file—not an object that can be instantiated. Similarly, a file cannot be an instantiation of a class—it is always implicit.
- ◆ The **Path** field is a read-only field that displays the path to the file. Click the icon to navigate directly to the specified source file.

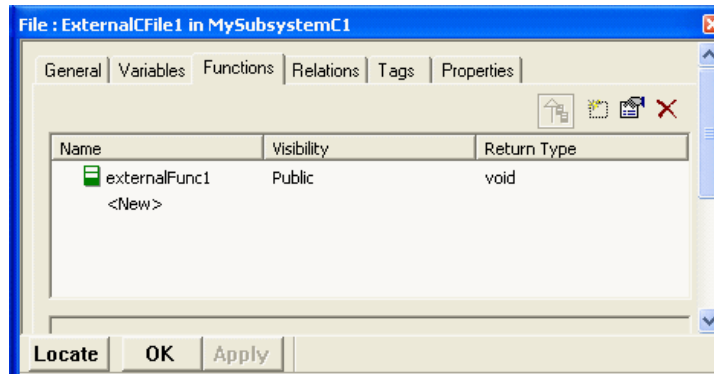
The Variables Tab

The **Variables** tab of the file Features dialog box enables you to add, edit, or remove variables from the file. It contains a list of all the variables belonging to the file. The following figure shows the **Variables** tab.



The Functions Tab

The **Functions** tab of the file Features dialog box enables you to add, edit, or remove functions from the file. It contains a list of all the functions defined in the file. The following figure shows the **Functions** tab.



Converting Files

You can easily convert a file to an object or vice versa by simply highlighting the object in the browser, then selecting **Change to** and the desired result.

To convert a file to a class, follow these steps:

1. Highlight the file in the browser, right-click, and select **Change to > Object** from the pop-up menu. The file changes to an object and moves to the `Objects` category in the browser.
2. Highlight the object in the browser, right-click, and select **Expose Class** from the pop-up menu. This create a new class with the name `<object> class`. It contains all of the content of the copied object including the attributes, operations, and statechart. This option is only available for an implicit object.

Note the following:

- ◆ If you are trying to convert an object to a file and there are aggregates that are not allowed for files, Rhapsody issues a warning message.
- ◆ Objects that are owned by another class or object cannot be converted to files.

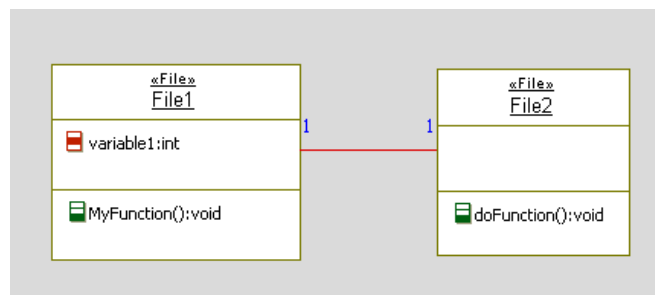
When the element has been converted, the graphical representations change in the diagrams and the converted element is moved to the appropriate category in the browser.

For information on changing the order of files, see [Changing the Order of Objects](#).

Using Associations and Dependencies

Files can be connected through associations or `«Usage»` dependencies. As standard practice, you should use associations.

For example, consider the files shown in the following figure.



Because these files are connected through bi-directional association, `File1` can call `doFunction()` directly from an operation or action on behavior.

Generating Code for Files

During code generation, files produce full production code, including behavioral code. In terms of their modeling properties, modeled files are similar to implicit singleton objects.

Note the following:

- ◆ For an active or a reactive file, Rhapsody generates a public, implicit object (singleton) that uses the active or reactive functionality. The name of the singleton is the name of the file.

Note: The singleton instance is defined in the implementation source file, not in the package source file.

- ◆ For a variable with a **Constant** modifier, Rhapsody generates a `#define` statement. For example:

```
#define MAX 66
```

The following table shows the differences between code generation of an object and a file.

Model Element	File Code	Object Code
Data member (attribute, association, or object)	A global variable	A member in the singleton struct
Function name ¹	The function name pattern is <Function>.	The name pattern for public functions is <Singleton>_<function>. The pattern for private functions is <Function>.
Function signature	The me argument is generated when required to comply with the signature of framework callback functions (for reactive behavior).	The same.
Initialization	Variables and associations are initialized directly in the definition. For example: <code>int x=5;</code> Objects are initialized in a generated Init function.	Done in the initialize function.
Type name	The name pattern for types (regardless of visibility) is <Type>.	The name pattern for public types is <Singleton>_type. The name pattern for private functions is <Type>. The name pattern can be configured using the properties <lang>_CG::Type::PublicName and PrivateName.
Visibility	Public members are declared as <code>extern</code> in the specification (.h) file and defined in the implementation (.c) file. For example: <code>extern int volume;</code> Private members are declared and defined in the implementation file as <code>static</code> . For example: <code>static int volume;</code>	Member visibility is ignored; the visibility is a result of the visibility of the struct. For example: <code>struct Ob_t { int volume; };</code>

Model Element	File Code	Object Code
<i>Auto-generated</i>		
Initialization and cleanup	<p>Only algorithmic initialization is done in the initialization method (creating parts; initializing links, behavior, and animation). The initialization and cleanup methods are created only when needed. The name of the initialization function is <file>_Init; the cleanup function is <file>_Cleanup.</p>	<p>Any initialization is done in the Init method. Init and Cleanup methods are generated by default.</p>
Framework data members	<p>Rhapsody generates a designated struct that holds only the framework members, and a single instance of the struct named <file>. The struct name is <file>_t.</p> <p>For example:</p> <pre>struct Motor_t { RiCReactive ric_reactive; }</pre>	<p>Framework members are generated as part of the object struct declaration.</p>
Call framework operations	<p>Framework operations on the file are called using the file.</p> <p>For example:</p> <pre>CGEN(Motor, ev());</pre>	<p>Framework operations on the singleton are called passing the singleton instance.</p> <p>For example:</p> <pre>CGEN(Motor, ev());</pre>
Statechart data members	<p>Statechart data members are generated as attributes of the generated structure.</p> <p>For example:</p> <pre>struct F_t { ... enum F_Enum { F_RiCNonState=0, F_ready=1} F_EnumVar; int rootState_subState; int rootState_active; };</pre>	<p>Statechart data members are generated as part of the struct.</p>
Statechart function names	<p>Public statechart functions are generated using the prefix <file>_.</p> <p>For example:</p> <pre>myFile_sIN()</pre>	<p>Use the same naming convention as any other operation.</p>

1. You can configure the name pattern for functions (for files, objects, and other elements) using the properties <lang>_CG::Operation::PublicName and PrivateName.

Using Files with Other Tools

The following table lists the effect of files on both Rhapsody and third-party tools.

Tool	Description
COM API	Files are supported by the COM API via the <code>IRPFile</code> and <code>IRPModule</code> interfaces. Refer to the <i>COM API Reference Guide</i> for more information.
Complete Relation	When you select Layout > Complete Relations , files and their relations are part of the information added to the diagram. See Completing Relations for more information on this functionality.
DiffMerge	Files are included in difference and merge operations, completely separate from objects. Refer to the <i>Team Collaboration Guide</i> for more information on the DiffMerge tool.
Populate Diagram	Files and their relations are fully supported.
References	If you use the References functionality for a file, the tool lists the owning package for the file and the diagrams in which the specified file appears. When you select a diagram from the returned list, the file is highlighted in the diagram. See Searching in the Model for more information on this functionality.
Report on model	In a report, the objects and files in the package are listed in separate groups in that order. See Reports for more information on the reporting tool.
Search in model	You can search for files in the model and select their type from the list of possible types. When selected, the file is highlighted in the browser. See Searching in the Model for more information on this functionality.
XMI Toolkit	When you export a file to XMI, it is converted to an object with a «File» stereotype. Files imported from XMI are imported into Rhapsody as files.

Attributes, Operations, Variables, Functions, and Types

In object model diagrams, attributes and operations are contained in classes. Therefore, they are not included as separate items in the **Drawing** toolbar. Similarly, variables, functions, and types are not included in the **Drawing** toolbar.

There may, however, be situations where you will want to show a higher level of detail and include attributes, operations, variables, functions, or types as individual diagram elements. Rhapsody provides a solution for these situations by allowing you to drag these elements from the browser to an OMD diagram

To add an attribute, operation, variable, function, or type to the diagram:

1. Select the relevant item in the browser.
2. Drag the item to the diagram window.

Note

Rhapsody allows these types of elements to be dragged from the browser to any of the static diagrams, not just object model diagrams.

When an item of this type is added to a diagram, the graphic element will display by default, the element name, the stereotype applied (if there is one) or the metatype of the element, and the associated image (if one has been defined).

Like all diagram elements, the features dialog for these elements can be opened by double-clicking on the element in the diagram.

The connectors provided in the **Drawing** toolbar can be used to connect individual elements of these types if the connection is semantically logical.

Once an element has been added to a diagram, the element can be added to a container element by dragging the element into the container element, for example, an attribute on the diagram can be dragged into a class.

Note

Graphic representations for these types of items can only be created by dragging them from the browser to the diagram. There is no API equivalent for this action.

Flow Ports

Flow ports allow you to represent the flow of data between objects in an object model diagram, without defining events and operations. Flow ports can be added to objects and classes in object model diagrams. They allow you to update an attribute in one object automatically when an attribute of the same type in another object changes its value.

Note

Flow ports are not supported in Rhapsody in J.

To add a flow port, follow these steps:

1. Right-click the object or class to display the context menu.
2. From the context menu, select **Ports > New Flowport**.

The method used for specifying the data that is to be sent/received via the flow port depends upon the type of flow port used - atomic or non-atomic. Non-atomic flow ports can only be used if your model uses the SysML profile. The following sections describe these two types of flow ports.

Atomic Flow Ports

Atomic flow ports can be input or output flow ports, but not bidirectional. To specify the flow direction, open the features dialog for the flow port and select the appropriate direction.

You specify the attribute that is to be sent/received via the flow port by giving the attribute and flow port the same name. If no attribute name matches the name of the flow port, a warning to this effect will be issued during code generation.

Atomic flow ports allow the flow of only a single primitive attribute.

When connecting two atomic flow ports, you have to make sure that one is an input flow port and one is an output flow port. The type of the attribute in the sending and receiving objects must match.

To connect two flow ports, use the Link connector.

Non-atomic Flow Ports

Non-atomic flow ports are available only in models that use the SysML profile.

Non-atomic flow ports can transfer a list of flow properties (a *flow specification*), which can be made up of flow properties of different types. For each flow property in the list, you indicate the direction of the flow. (Non-atomic flow ports are bi-directional.)

To define the flow properties to be sent/received via the flow port:

1. Create a flow specification.
2. Add flow properties to the flow specification. This can be done using the relevant browser context menu, or directly on the **FlowProperties** tab of the Features dialog box for the flow specification. You can also use drag-and-drop in the browser to add existing flow properties to the flow specification. (If you want to use an existing attribute, you can convert the attribute to a flow property by selecting **Change To > FlowProperty** from the attribute context menu.)
3. For each of the flow properties defined, specify the direction.
4. Create two objects and add a flow port to each.
5. In the features dialog for each of the two flow ports (the sending and receiving), set the Type to the name of the flow specification you created previously.
6. For one of the flow ports, open the features dialog and select the *Reversed* check box on the **General** tab
7. Connect the two flow ports with a link.

Updating Attribute Values

To have the value of an attribute updated when the attribute on the other end of flow is updated, you must use the function `setflowportname`, for example, if you have a flow port called `x`, you would call `setX(5)`.

When this function is called, there is also an event generated called `chflowportname`. In our example, it would be `chX`. In order to be able to react to this event, you must define an event with this name in your model.

For both of these functions, the first letter of the flow port name is upper-case even if the actual name of the flow port begins with a lower-case letter.

Note

For details regarding the use of flow ports when importing Simulink models, see [Integrating Simulink Components](#).

External Elements

Rhapsody enables you to visualize frozen legacy code or edit external code as *external elements*. This external code is code that is developed and maintained outside of Rhapsody. This code will not be regenerated by Rhapsody, but will participate in code generation of Rhapsody models that interact or interface with this external code so, for example, the appropriate `#include` statement is generated. This functionality provides easy modeling with code written outside of Rhapsody, and a better understanding of a proven system.

Rhapsody supports the following functionality for external elements:

- ◆ Reverse engineering can import elements as external.
- ◆ Reverse engineering populates the model with enough information to:
 - Model external elements in the model.
 - Enable you to open the source of the external elements, even if the element is not included in the scope of the active component.
- ◆ Rhapsody generates the correct `#include` for references to external elements.
- ◆ Elements inherit their externality from the parent. For example, if a package is external, all its aggregates are also external.
- ◆ You can add external elements to component files to define the exact location of the source code.
- ◆ Rhapsody displays external elements in the scope tree of the component.

Creating External Elements

There are two ways to create external elements:

- ◆ By reverse engineering the files (see [Using Reverse Engineering](#))
- ◆ By modeling (see [Creating External Elements by Modeling](#))

Using Reverse Engineering

How you create external elements depends on whether the code is frozen legacy code or the code is still being modified. The following sections describe each scenario:

- ◆ [Single Iteration](#)
- ◆ [Multiple Iterations](#)
- ◆ [Creating External Elements in Pre-V5.2 Models](#)

Single Iteration

Suppose you want to model external code for referencing, without regenerating it. This is appropriate for legacy code or a library that will not change.

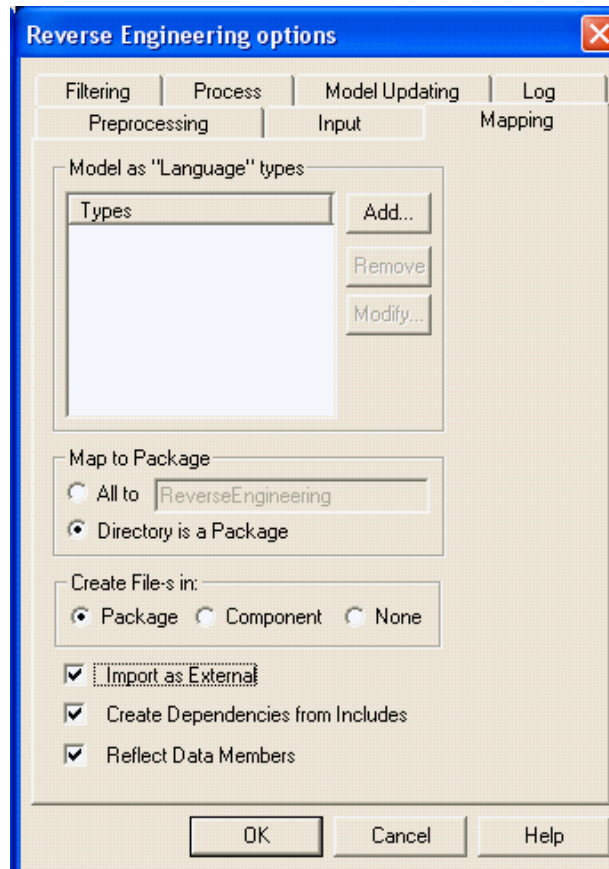
To use reverse engineering to create the external elements a single time, follow these steps:

1. Create a new model, or open an existing one.
2. Add a new component for the reverse engineered, external code (for example, `ExternalComponent`).
3. Set `ExternalComponent` to be the active component (highlight it in the browser, right-click, and select **Set as Active Component** from the pop-up menu).
4. Select **Tools > Reverse Engineering**. The Reverse Engineering dialog box opens.
5. Specify the files or folders you want to reverse engineer.
6. Click **Options**, then select the **Mapping** tab.

7. Specify the following settings:

- a. For Rhapsody in C, keep the **Package** setting for the **Create File-s in** option; for the other languages, select the appropriate setting (**Component** or **None**).
- b. Enable **Import as External**.

The following figure shows the proper settings for Rhapsody in C.



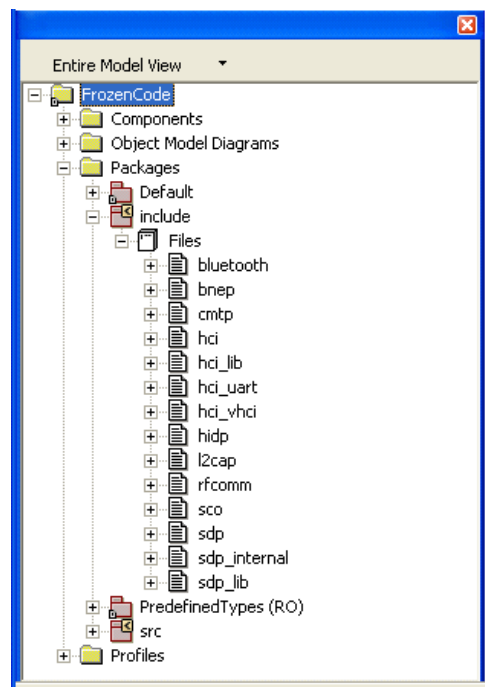
8. Set the other reverse engineering options as appropriate for your model (see [Reverse Engineering](#) for more information on the available options).
9. Click **OK** to apply your settings and close the dialog box.
10. Click **Import**. The specified files are imported into Rhapsody as external elements.

As a result of the import:

- ◆ The imported elements are added to the scope of the configuration.

- ◆ All the imported packages have the property `CG::Package::UseAsExternal` set to Checked.
- ◆ The **Include Path** or **Directory** of the Features dialog box for the configuration (in the example, `ExternalComponent`) is set to the correct include path.
- ◆ In Rhapsody in C (when the **Create File-s in** field is set to **Package**), the `CG::Configuration::GenerateDirectoryPerModelComponent` property is set to Checked for the configuration.

Note that external elements include a special icon in the upper, right corner of the regular icon, as shown in the following figure.



11. Verify the import to make sure the implementation and specification files are named correctly, the correct folders were created, and so on. Make any necessary changes.
12. Set the original component to active.
13. For the original component, create a dependency with a `«Usage»` stereotype to the `ExternalComponent`.
14. Make sure that the external elements are included in the scope of the `ExternalComponent` only.

Multiple Iterations

Suppose you want to model external code for referencing, without regenerating it—but the external code might change and the external element should be updated according to the changes in the code.

Set up your model as follows:

1. Complete the steps in the previous procedure (see [Using Reverse Engineering](#)) to create a new external model (for example, `ExternalModel`).
2. Save your model, then close it.
3. Open a new, development model.
4. Select **File > Add to Model**, then select the external model. Select **As Reference** and select all the top-most packages and the component (`ExternalModel`). The elements are imported as read-only (RO).
5. Create a dependency with a «Usage» stereotype to the `ExternalModel`.

To synchronize the code changes, follow these steps:

1. Open the external model.
2. Update the reverse engineering options as needed to include the code modifications (such as including new folders), then click **Import**.
3. Close the external model.
4. Open the development model.
5. Update the model according to the changes in the external model:
 - a. Remove references to elements deleted from the external model.
 - b. Update references to renamed elements from the external model (they become unresolved).
 - c. New elements are simply added to the model.

Creating External Elements in Pre-V5.2 Models

To add external elements to models created before Version 5.2, follow these steps:

1. Unoverride the following properties:
 - ◆ `CG::Configuration::StrictExternalElementsGeneration`
 - ◆ `CG::Component::SupportExternalElementsInScope`

These properties are automatically overridden by Rhapsody when you load an older model. Refer to the *Upgrade Guide* for more information on these properties.
2. Follow the procedure described in [Single Iteration](#).

Creating External Elements by Modeling

Alternatively, you can add external elements to the model manually. This option is used when there are very few elements to be modeled as external.

There are two ways to model the elements manually:

- ◆ Using rapid external modeling
- ◆ Using the component model

Using Rapid External Modeling

To model the elements manually using rapid external modeling, follow these steps:

1. Open an existing model or create a new one.
2. Create the external elements:
 - ◆ Create the new element to be referenced.
 - ◆ Set its `CG::Class::UseAsExternal` property to `Checked`.
 - ◆ Set its `CG::Class::FileName` property to the value expected in the `#include`. For example, `MySubsystem\C`.
3. Add the rest of the path to the **Include Path** field of the component. In the example, this would be `C:\MyProjects\Project1`.
4. Add the external elements to the scope of the component.
5. Add relations to the external elements (see [Accessing the External Element Code](#) for more information).

Using the Component Model

To model the elements manually using the component model, follow these steps:

1. Open an existing model or create a new one.
2. Add a new component for the external elements (for example, `ExternalComponent`).
3. Set the scope of the component to **Selected Elements**.
4. Create a package that will contain all the external elements, and set its `CG::Package::UseAsExternal` property to `Checked`.

Note: This step is optional; you can also add external elements to existing packages.

5. Add the package that contains the external elements to the scope of the external component. Make sure that the package is not included in the scope of other components.
6. Create a new element that will be referenced in the package.
7. Provide the following information about the source files of the external elements:
 - a. Create a hierarchy of packages as needed for the proper `#include` path. For example, suppose you want reference class `C`, which is defined in `C:\MyProjects\Project1\MySubsystem\C.h`; you would create the package `MySubsystem`.
 - b. Add a file with the necessary name to the folder and map the external element to it. You do not need to do this if the external element has the same name as the file.
 - c. Create a usage dependency to the external component.
8. Add relations to the external elements (see [Accessing the External Element Code](#) for more information).

In the generated files, the following `#include` is generated for the example element:

```
#include <MySubsystem\C.h>
```

Shortcut for Rhapsody in C

In Rhapsody in C, you can use the following shortcut in place of Steps 2 to 7 in the previous procedure:

1. Create a hierarchy of packages as needed for the proper `#include` path.

For example, suppose you want reference class `C`, which is defined in `C:\MyProjects\Project1\MySubsystem\C.h`; you would create the package `MySubsystem`.
2. Set the `CG::Package::UseAsExternal` property for the top-most package to `Checked`.
3. Create the appropriate files (see [Creating a File](#) for more information). Continuing the example, you would simply create the file `C`.
4. Create a new element that will be referenced in the file.
5. Add the rest of the path to the **Include Path** field of the component. In the example, this would be `C:\MyProjects\Project1`.
6. Set the property `CG::Configuration::GenerateDirectoryPerModelComponent` to `Checked` for the component.

Converting External Elements

You can convert external elements so they are no longer external (and therefore include them in code generation). This functionality enables you to gradually move code that was developed outside of Rhapsody to an application being developed using Rhapsody.

To convert *all* the external elements at once, follow these steps:

1. Open the model and perform the following steps:
2. Change the following properties:
 - ◆ Unoverride the property `CG::Package::UseAsExternal` for the top-most packages.
 - ◆ Unoverride the properties `Generate` and `AddToMakefile` (under `CG::File`) for the top-most folders in the external element component.
3. Add the packages and classes to the scope of the development component and remove the packages and classes from the external element component.
4. Delete the external component.
5. Generate and build the code.
6. Continue development in Rhapsody as usual.

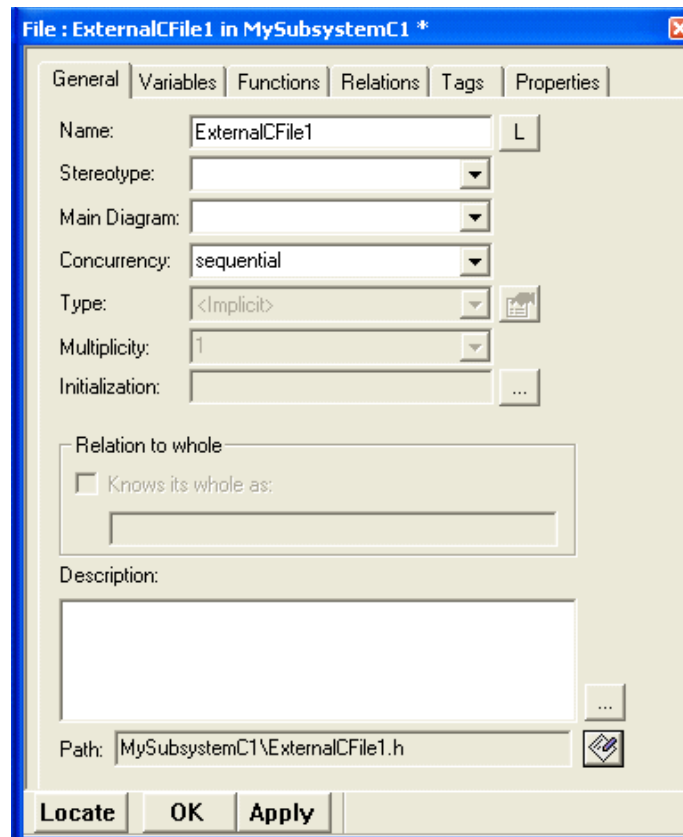
To convert specified external elements, follow these steps:

1. Create a new package for the converted elements (for example, `RedesignPackage`).
2. Add the new package to the scope of the development component.
3. Move one class or file from the external package to the `RedesignPackage`.
4. Generate, build, and test the code. Repeat as necessary.
5. Repeat Steps 3 and 4 for as many classes or files as you want to convert.
6. Continue development in Rhapsody as usual.

Viewing the Path to the Source File

The Features dialog box for an external file in Rhapsody in C includes a new field, **Path**, which shows the full path to the specification source file. The path is read-only, but you can copy it.

The following figure shows the updated dialog box.



Click the icon to navigate directly to the specified source file.

Accessing the External Element Code

To access the code in the external files, create relations to them (such as dependencies with «Usage» stereotypes, generalizations, associations, and so on). Note that the resultant source code will automatically contain the correct `#include` statements for the external elements.

In the Features dialog box for the configuration, set the scope to **Selected Elements**, and verify that the external files are not checked.

When you generate code that includes relations to external elements:

- ◆ Dependencies are converted to `#includes`.
- ◆ Generalizations are converted to inheritance and `#include`.
- ◆ Associations are converted to data members and `#includes`.
- ◆ Types are converted to type and `#include`.
- ◆ Objects and parts are instantiated.

Once you have created the external elements, you can edit the code using the Edit menu, edit options in the pop-up menu, or active code view window—just as you do for any Rhapsody code.

Adding Source Files to the Build

To add the source files of an external element to the build, follow these steps:

1. Add the component file that contains the mapping of the necessary external elements to the component.
2. Set the property `CG::File::Generate` to `Cleared`.
3. Set the property `CG::File::AddToMakefile` to `Checked`.

Generating Code for External Elements

The following table lists how Rhapsody generates code for external elements.

Element Type	Description
Package	The code generator does not generate code for an external package. However, you can map the package to a component's file or folder (and then relate it to a file or directory). You can include the package in the component scope. During code generation, a relation to a package is converted an <code>#include</code> to a file, if the package is mapped to a component's file.
Class, object, or file	The code generator does not generate code for an external class, object, or file. During code generation, a relation to a class, object, or file is converted to an <code>#include</code> or a forward declaration.
Type	The code generator does not generate code for an external type. A relation to a type is converted to an <code>#include</code> of its parent.
File (component)	A file is external if all its elements are external. If the file's <code>CG::File::Generate</code> property is set to <code>Cleared</code> , the file becomes external and code is not generated for it. To include a file in the build, set its <code>CG::File::AddToMakefile</code> property to <code>Checked</code> .

Generating Code for Relations

During code generation, Rhapsody generates either an `#include` or a forward declaration for a relation in the source file of the dependent element.

Forward Declaration (Class)

If a dependency has a `«Usage»` stereotype and the `CG::Dependency::UsageType` property is set to `Existence`, it is generated as a forward declaration. For example:

```
class ExternalClass;
```

#includes for a Class, Object, or File

External dependencies (dependencies with a «Usage» stereotype and the `CG::Dependency::UsageType` property set to `Specification/Implementation`) and implicit dependencies (such as associations and generalizations) are generated as forward declarations and `#include` statements.

To generate a local `#include` statement (for example, `#include <C.h>`), set the property `CG::File::IncludeScheme` to `LocalOnly`.

To generate a relative `#include` statement (for example, `#include <MySubsystem\C.h>`), set the `CG::File::IncludeScheme` to `RelativeToConfiguration`.

You can also use the `CG::Configuration::GenerateDirectoryPerModelComponent` and `CG::Class/Package::FileName` properties to set relative paths. Refer to the definition of this property in the Features dialog box.

Limitations

Note the following restrictions and behavior of external elements:

- ◆ External elements are not animated; they behave like the system border.
- ◆ Changes in the source files of external elements are *not* roundtripped. If necessary, use reverse engineering to update external elements.
- ◆ Only the following elements can be external: class, implicit object or file, package, and type. Components, folders, variables, and functions cannot be external.
- ◆ You cannot use **Add to Model** to add an external element as a unit. However, you can add the unit under another name, and then set that unit to be external.

Implementing Base Classes

In Rhapsody in C++ and Rhapsody in J, you can easily convey model elements defined at the interface level to the implementing class level. Using this functionality, classes automatically realize the implementing interfaces and help you synchronize the changes in the interface to the realizing classes.

There are two ways to invoke this functionality: implicitly and explicitly.

Implicit Invocation

When you connect two classes with a generalization realization, the base classes are implemented implicitly. However, a generalization will not trigger base class implementation in the following cases:

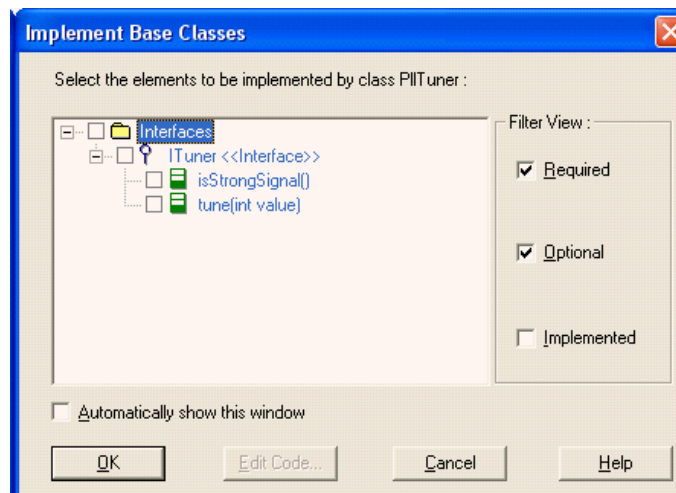
- ◆ Inheritance between two COM interfaces
- ◆ Inheritance between two CORBA interfaces
- ◆ Inheritance between two Java interfaces

If there are no operations or attributes to be overridden because of the current action, the Implement Base Classes dialog box is not displayed.

Explicit Invocation

To access this functionality explicitly, in the browser or OMD, right-click a class and select **Implement Base Classes** from the pop-up menu.

The Implement Base Classes dialog box opens, as shown in the following figure.



Implement Base Classes Dialog Box

This dialog box provides a tree-like view of all the interfaces (including methods, attributes, and stereotypes) that can be implemented by the class.

The dialog box contains three filters to control the contents of the tree view:

- ◆ **Required**—Display the operations that *must* be implemented.
- ◆ **Optional**—Display the operations that can be implemented. By default, Rhapsody displays the required and optional operations.
- ◆ **Implemented**—Display the operations that are already implemented.

Base Class Tree View

Depending on the base class, Rhapsody displays different items in the tree view. The following table shows which items are displayed.

Base Class	Items Displayed in the Tree View
C++ class	All virtuals and pure virtuals. You must implement the pure virtual methods.
Java class or Java interface	All the methods. You must implement the interfaces. The GUI takes into account the “final” option for Java methods and classes.
COM interface	All methods and attributes.
CORBA interface	All methods.

Rhapsody uses the following colors to differentiate the different method types:

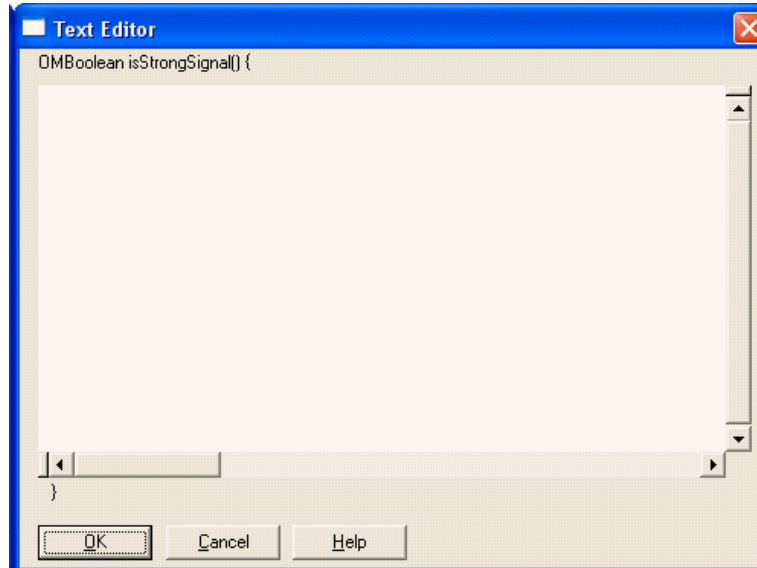
- ◆ **Blue**—Denotes a virtual method.
- ◆ **Bold, blue**—Denotes an abstract method.
- ◆ **Gray**—Denotes a method that has already been implemented.

If you try to invoke the Implement Base Classes functionality for a read-only class, Rhapsody displays a warning message informing you that the class cannot be modified. However, the Implement Base Classes dialog box opens in read-only mode so you can analyze the class. You can view the code by selecting **Edit Code**, but the **OK** button will be disabled.

Editing the Implementation Code

To change the implementation code for an operation, follow these steps:

1. Select the operation in the tree view.
2. Click the **Edit Code** button. Rhapsody displays the code in the default text editor, as shown in the following figure.

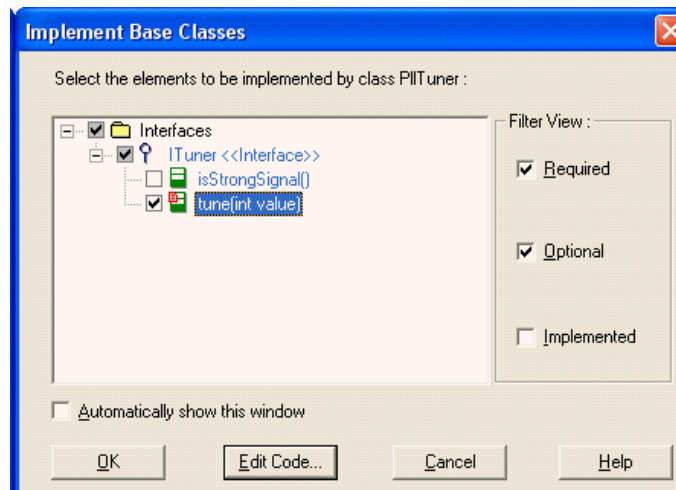


3. Type in the new implementation code in the text box.
4. Click **OK** to apply your changes and close the dialog box.

Note that if you try to edit code for an operation that has already been implemented, the text editor displays the implementation code in read-only mode.

When you edit the implementation code, Rhapsody overlays a red icon in the upper, left corner of the class icon in the tree view, as shown in the following figure.

See [The Internal Editor](#) for more information on the internal editor and its properties.



Controlling the Display of the Dialog Box

The **Automatically show this window** check box controls whether the dialog box is displayed on implicit requests. By default, this check box is not enabled, so the dialog box is displayed only when you explicitly invoke it.

If you select this check box, Rhapsody writes the following line to the [General] section of the `rhapsody.ini` file:

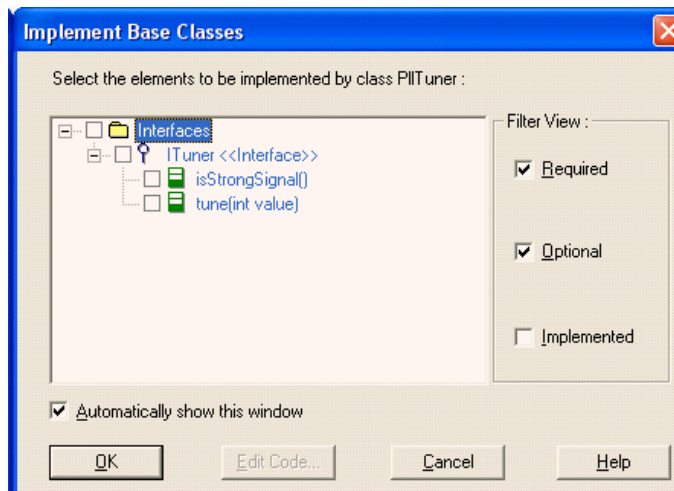
```
ImplementBaseClasses=TRUE
```

If desired, you can add this line directly to the `rhapsody.ini` file to automatically display the dialog box.

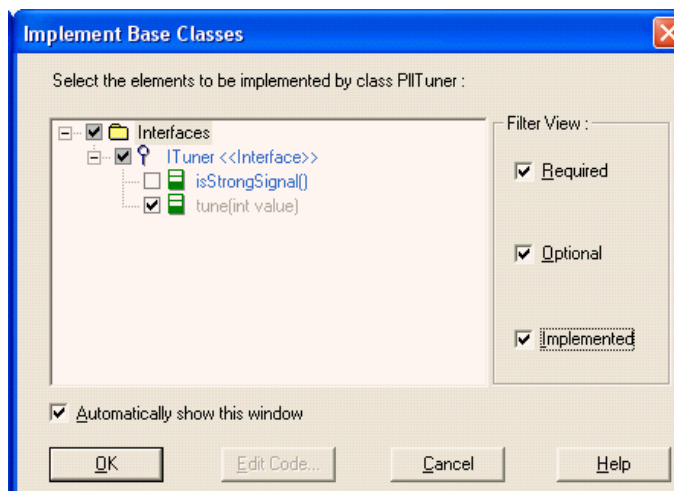
Realizing the Elements

To realize an element, select it and click **OK**.

For example, suppose you want to implement the `tune` operation. In the dialog box, select `tune`, then click **OK**.



The `PllTuner` class implements the `tune` operation and displays it in the browser. If you select **Implement Base Classes** for the `PllTuner` class again, the `tune` operation is no longer listed as a required or optional element. Click **Implemented** to see that the `tune` operation was implemented and is now displayed in gray, as shown in the following figure.



If an element has been implemented (it appears in gray in the tree and is checked), you cannot uncheck (“unrealize”) it.

Note

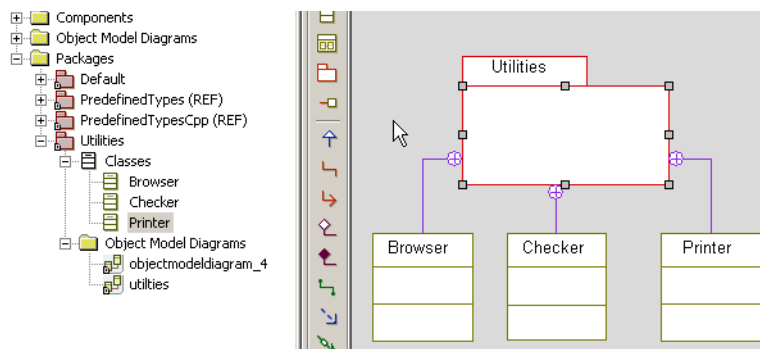
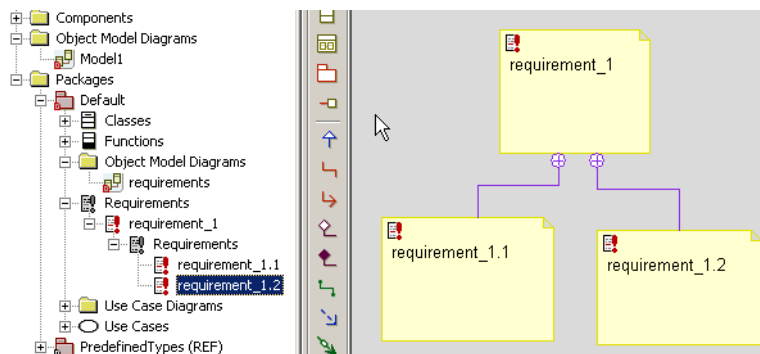
If you choose **Undo**, Rhapsody unimplements *all* the implemented classes, not just the last one, because the implementation is viewed as a single, atomic operation. For example, if you implement five elements during one operation, then select **Undo**, all five are removed.

Namespace Containment

Rhapsody allows you to display namespace containment in object model diagrams. This type of notation is also referred to as “alternative membership notation.” It depicts the hierarchical relationship between elements and the element that contains them, for example:

- ◆ requirements that contain other requirements
- ◆ packages that contain classes
- ◆ classes that contain other classes

The following screen captures illustrate the use of this notation.



Property that Controls Display of Namespace Containment

The display of namespace containment is controlled by the boolean property `ObjectModelGe:ClassDiagram:TreeContainmentStyle`, which can be set at the diagram, package, or project level. Namespace containment is displayed when the property is set to `Checked`.

The default value of the property is `Cleared`. However, in the SysML profile the default value of the property is `Checked`.

Displaying Namespace Containment

To display namespace containment in a diagram:

1. Drag the “container” element and the “contained” elements to the diagram.
2. From the menu, select **Layout > Complete Relations > All**

The hierarchical relationship between the elements will be depicted in the diagram.

Alternatively, you can select the **Populate Diagram** option when creating a new diagram. If you then select elements that have a hierarchical relationship, the diagram created will display the namespace containment for the elements.

Note

There is no drawing tool to manually draw this type of relationship on the canvas. Containment relationships between elements can only be displayed automatically based on existing relationships, using one of the methods described above.

Activity Diagrams

Activity diagrams specify a workflow, or process, for classes, use cases, and operations. As opposed to statecharts, activity diagrams are preferable when behavior is not event driven. A class (use case/operation) can have either an activity diagram or a statechart, but not both. However, a class, object, or use case may have more than one activity diagram with one of the diagrams designated as the *main behavior*.

Note

It is possible to change the main behavior between different activities within the same classifier. However, only the main behavior is inherited by the derived classes.

Activity Diagram Features

One useful application of activity diagrams is in the definition of algorithms. *Algorithms* are essentially decompositions of functions into smaller functions that specify the activities encompassed within a given process.

Note

Sequence diagrams can show algorithms of execution within objects, but activity diagrams are more useful for this purpose because they are better at showing concurrency.

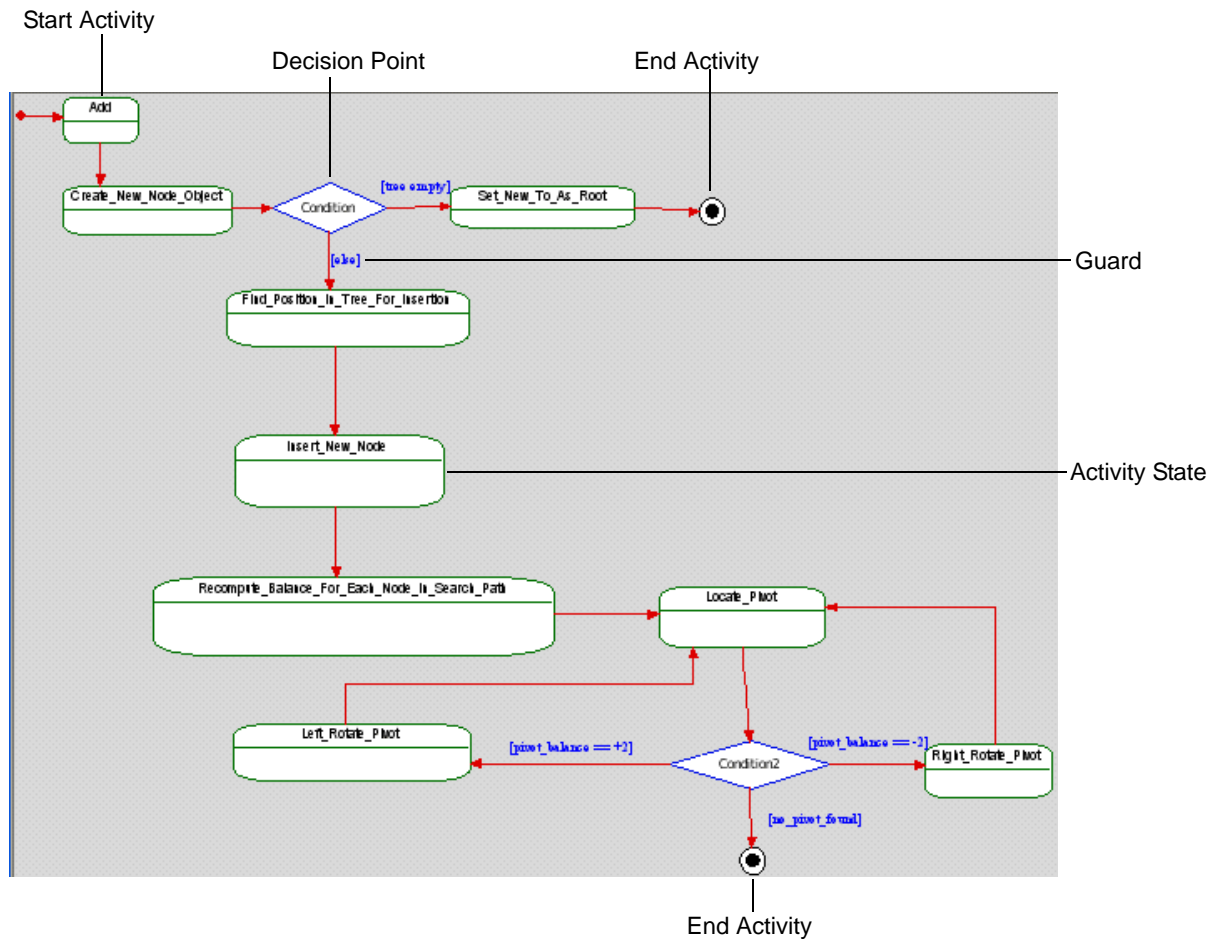
Activity diagrams have several elements in common with statecharts, including start and end activities, forks, joins, guards, and states (called *actions*). Unlike statecharts, activity diagrams have the following elements:

- ◆ **Decision points**—Show branching points in the program flow based on guard conditions.
- ◆ **Actions**—Represent function invocations with a single exit transition taken when the function completes. It is not necessary for all actions to be within the same object.
- ◆ **Action blocks**—Represent compound actions that can be decomposed into actions.
- ◆ **Subactivities**—Represent nested activity diagrams.
- ◆ **Object nodes**—Represents an object passed from the output of one state's actions to the input of another state's actions.
- ◆ **Swimlanes**—Visually organize responsibility for actions and subactions. They often correspond to organizational units in a business model.

Activity Diagrams

- ◆ **Reference activities**—References an activity in another activity chart, or to the entire activity chart itself.

The following is a sample activity diagram for an Add () operation that adds a node to a tree.



Advanced Features

You may also use these advanced features of the activity diagrams:

- ◆ Naming and renaming activity diagrams
- ◆ Include an activity diagram, but not a statechart, with a package without creating a class
- ◆ Support multiple activities in a package
- ◆ [Associating an Object Node with a Class](#)
- ◆ Create [Adding Call Behaviors](#) in the activity diagram or by dropping an operation from the browser into the diagram
- ◆ Swimlane association (representing field population) to a class (only) can be created by dragging the class from the browser to the Swimlane name compartment. For more information, see [Swimlanes](#).
- ◆ Reference an alternate activity diagram within the main behavior activity diagram

Actions

Activity diagrams are flowcharts that decompose a system into activities that correspond to states. These diagrammatic elements, called *actions*, are member function calls within a given operation. In contrast to normal states (as in statecharts), actions in activity diagrams terminate on completion of the activity, rather than as a reaction to an externally generated event.

Each action can have an entry action, and must have at least one outgoing transition. The implicit event trigger on the outgoing transition is the completion of the entry action. If the action has several outgoing transitions, each must have its own guard condition.

Actions have the following constraints:














- ◆ Outgoing transitions from actions do not include an event signature. They can include guard conditions and actions.
- ◆ Actions have non-empty entry actions.
- ◆ Actions do not have internal transitions or exit actions, nor do activities.
- ◆ Outgoing transitions on actions have no triggering events.











Creating Activity Diagram Elements

The following sections describe how to use the activity diagram drawing tools to draw the parts of an activity diagram. See [Graphic Editors](#) for basic information on diagrams, including how to create, open, and delete them.

Activity Diagram Drawing Icons


The **Drawing** toolbar for an activity diagram contains the following tools.

Drawing Icon	Description
	Action show member function calls within a given operation.
	Action Block represents compound actions that can be decomposed into actions. Action blocks can show more detail than is possible in a single, top-level action
	Subactivity represents a nested activity diagram. Subactivities represent the execution of a non-atomic sequence of steps of some duration nested within another activity.
	Object Node represents an object passed from the output of one state's actions to the input of another state's actions.
	Call Behavior represents a call to an operation of a classifier. This is only used in modeling and does not generate code for the call operation.
	Activity Flow represents a message or event that causes an object to transition from one state to another.
	Default Flow points to the state that the object, use case, or operation enters when the activity starts.
	Loop Activity Flow represents a "self transition" in that the behavior repeats within a program.
	Condition Connector shows a branching condition. A condition connector can have only one incoming transition and two or more outgoing transitions.
	Termination State signifies either local or global termination, depending on where they are placed in the diagram.
	Junction Connector joins multiple transitions into a single, outgoing transition.
	Diagram Connector connect one part of an activity diagram to another part on the same diagram. They represent another way to show looping behavior.
	Draw Join Sync Bar merges multiple incoming transitions into a single outgoing transition.

Drawing Icon	Description
	Draw Fork Sync Bar separates a single incoming transition into multiple outgoing transitions.
	Transition Label add or modify a text describing a transition.
	Swimlanes Frame organizes activity diagrams into sections of responsibility for actions and subactions.
	Swimlanes Divider icon divides the swimlane frame using vertical, solid lines to separate each swimlane (actions and subactions) from adjacent swimlanes.
	Dependency icon indicates a dependent relationship between two items in the diagram.
	Send Action State icon is represents sending actions to external entities. The Send Action State is a language-independent element, which is translated into the relevant implementation language during code generation.
	Call Operation icon represents a call to an operation of a classifier.
	Action Pin icon adds an element to represent the inputs and outputs for the relevant action or action block. An action pin can be used on a Call Operation (derived from the arguments). This icon is displayed in the Drawing toolbar when you select the Analysis Only option when defining the General features of the activity diagram. See Adding Action Pins / Activity Parameters to Diagrams for more information.
	Activity Parameter icon defines a characteristic of an action block. This icon is displayed in the Drawing toolbar when you select the Analysis Only option when defining the General features of the activity diagram. See Adding Action Pins / Activity Parameters to Diagrams for more information.
	Accept Event Action icon lets you add this element to an activity diagram so that you can connect it to an action to show the resulting action for an event. This element can specify the following: <ul style="list-style-type: none"> • Event to send • Event target • Values for event arguments This icon is displayed in the Drawing toolbar when you select the Analysis Only check box when you define the General features of the activity diagram.

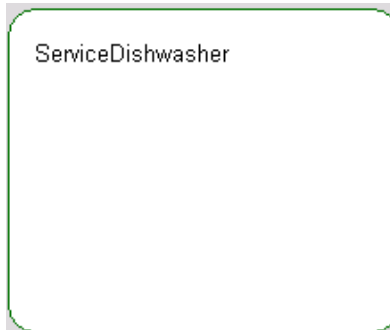
Drawing an Action

To draw an action, follow these steps:

1. Create an activity diagram.
2. Click the **Action** icon  on the **Drawing** toolbar.

3. Click or click-and-drag in the activity diagram to place the action at the desired location.
4. Type a name for the action.
5. Press **Ctrl+Enter** or click the Select arrow in the toolbar to terminate editing.

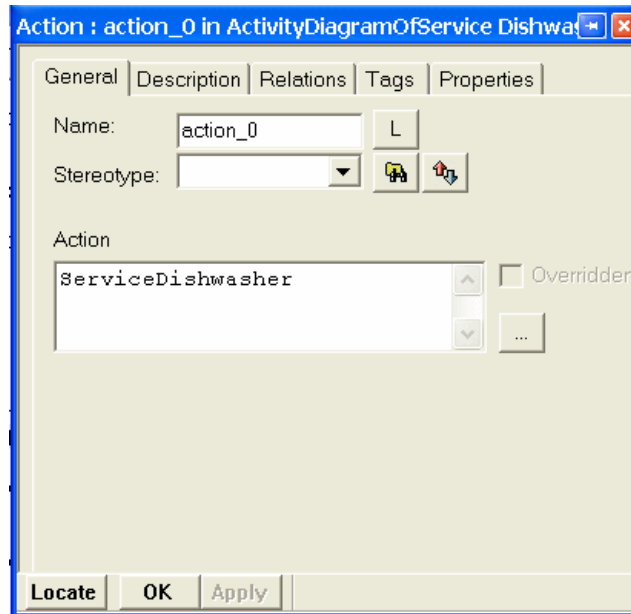
Actions have rectangles with curved edges, as shown in the following figure.





By default, the action expression, which does not need to be unique within the diagram, is displayed inside the action symbol. See [Displaying an Action](#) for information on modifying the display.

Modifying the Features of an Action

The Features dialog box enables you to add and change the features of an action, including its name and action. The following figure shows the Features dialog box for an action.



An action has the following features:

- ◆ **Name**—Specifies the name of the action. The description of the action can be entered into the text area on the **Description** tab. This description can include a hyperlink. See [Hyperlinks](#) for more information.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.
 - To select from a list of current stereotypes in the project, click the folder with binoculars  button.
 - To sort the order of the stereotypes, click the up and down arrows  button.

Note: The COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM Development Guide* for more information about these components.

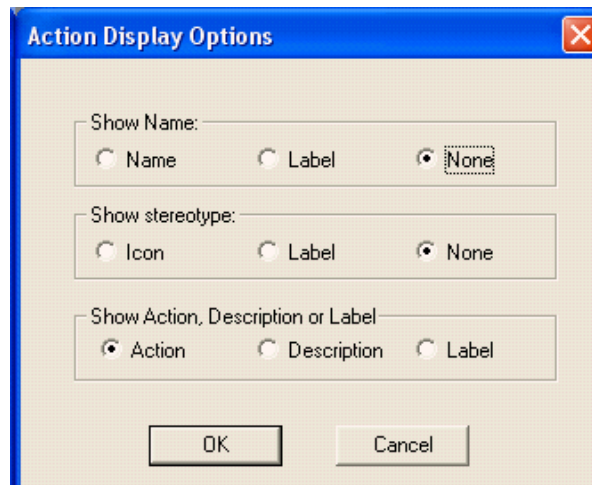
- ◆ **Action**—Specifies an action in an activity diagram. This is the text you typed into the diagram when you created the action.
The **Overridden** check box allows you to toggle the check box on and off to view the inherited information in each of the dialog box fields and decide whether to apply the information or revert back to the currently overridden information.

Displaying an Action

You can show the name, action, or description of the action in the activity diagram.

To specify which attribute to display, follow these steps:

1. Right-click the action and select **Display Options** from the pop-up menu. The Display Options menu is displayed, as shown in the following figure.



2. Select the appropriate values.

Action Blocks

Action blocks represent compound actions that can be decomposed into actions. Action blocks can show more detail than is possible in a single, top-level action. You can also use pseudocode, text, or mathematical formulas as alternative notations. Transitions inside an action block cannot cross the action block boundary.

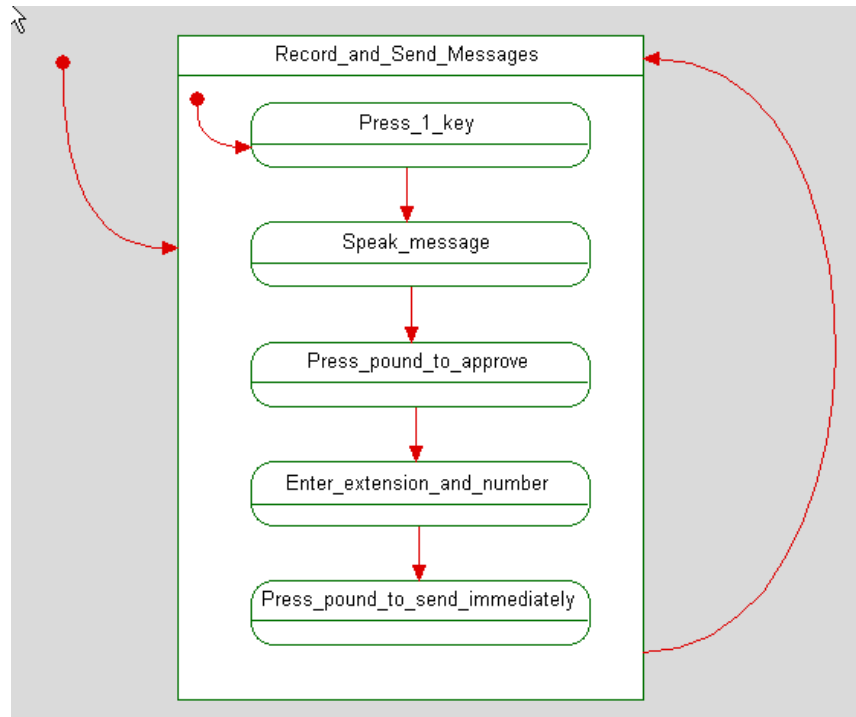
Creating an Action Block

To define the activity, draw an action block using these steps:

1. Click the **Action Block**  icon on the **Drawing** toolbar.

2. Click or click-and-drag in the activity diagram to place the action block at the desired location.
3. Draw actions and transitions inside the action block to express the activity being modeled.

Action blocks are rectangles. The name of the blocked activity is shown at the top. The `Record_and_Send_Messages` activity shown in the following sample action block that encompasses several activities.





Modifying the Features of an Action Block

The Features dialog box enables you to change the features of an action block, including its name and description. The following figure shows the Features dialog box for an action block.



An action block has the following features:

- ◆ **Name**—Specifies the name of the action block. The description of the action block can be entered into the text area on the **Description** tab. This description can include a hyperlink. See [Hyperlinks](#) for more information.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the action block, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.
 - To select from a list of current stereotypes in the project, click the folder with binoculars  button.
 - To sort the order of the stereotypes, click the up and down arrows  button.

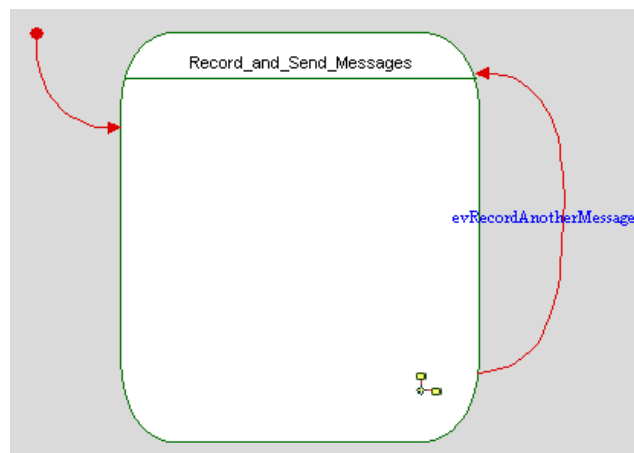
Note: The COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM Development Guide* for more information about these components.

Creating a Subactivity from an Action Block

You can convert an action block to a subactivity. This moves the contents of the block into a separate subchart, and simplifies the diagram containing the action block.

To create a subactivity from an action block, right-click the action block and select **Create Sub Activity Diagram** from the pop-up menu. Rhapsody creates a new subchart containing the former contents of the action block.

Consider the action block shown in the figure. When you create a subactivity for the action block, the parent state changes to that shown in the following figure.



The icon in the lower right corner indicates that the action block has a subchart.

Subactivities


Subactivities represent nested activity diagrams. Subactivities represent the execution of a non-atomic sequence of steps of some duration nested within another activity. Internally, a subactivity consists of a set of actions, and possibly a wait for events. In other words, a subactivity is a hierarchical action during which an associated subactivity diagram is executed. Therefore, a subactivity is a submachine state that executes an activity diagram.

The nested activity diagram must have an initial (default) state and a termination state (see [Local Termination Semantics](#)). When an input transition to the subactivity is triggered, execution begins with the initial state of the nested activity diagram. The outgoing transitions from a subactivity are enabled when the termination state of the nested activity diagram is reached (when the activity completes) or when triggering events occur on the transitions. Because states in activity diagrams normally do not have triggering events, subactivities are normally exited when their nested graph is finished.

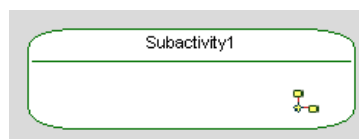
Many subactivities can invoke a single-nested activity diagram.

Creating a Subactivity

To draw a subactivity, follow these steps:

1. Click the **Subactivity**  icon on the **Drawing** toolbar.
2. Click (or click-and-drag) in the activity diagram to place the subactivity at the desired location.

A subactivity looks like an action.



Opening a Subactivity Diagram

To open a subactivity diagram, right-click the subactivity and select **Open Sub Activity Diagram** from the pop-up menu.

Termination States


UML final states are called *termination states* in Rhapsody. Termination states can signify either local or global termination, depending on where they are placed in the diagram.

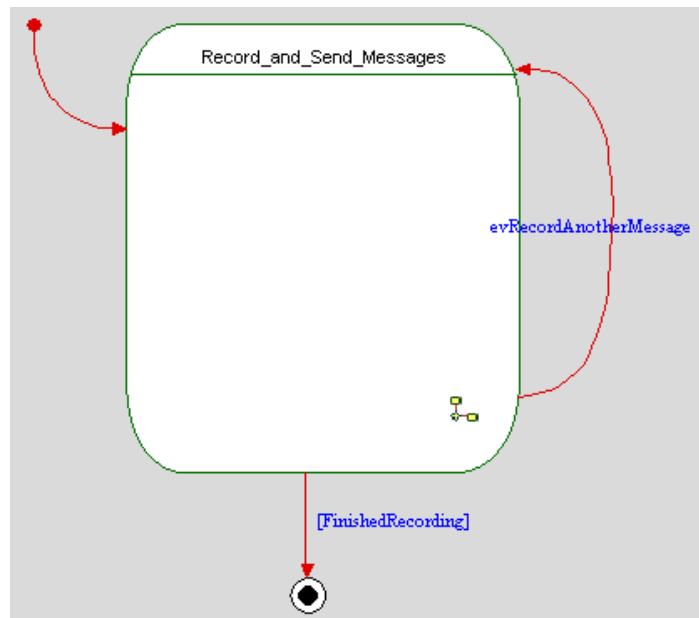
When the state is drawn inside a composite (block) state, it is considered a final state. This terminates the activity represented by the composite state, but not the instance performing the activity. See [Local Termination Semantics](#) for more information. When the state is drawn inside the top state, it is considered a termination state. This terminates the state machine, causing the instance to be destroyed.

Note

The behavior of termination states is controlled by the `LocalTerminationSemantics` property under `CG::Statechart`.

To create a termination state, follow these steps:

1. Click the **Termination State**  icon on the **Drawing** toolbar.
2. Click in the activity diagram to place the termination state at the desired location.
3. Draw a transition from any kind of state to the termination state.
4. If desired, enter a guard condition to signal the end of the activity. A termination state is drawn as a circle with a black dot in the middle, as shown in this example.



As with the other connectors, termination states and their transitions are included in the browser view.



Object Nodes

Actions operate on objects and are used by objects. These objects either initiate an action or are used by an action. Normally, actions specify calls sent between the object owning the activity diagram (which initiates actions) and the objects that are the targets of the actions. An *object node* represents an object passed from the output of one state's actions to the input of another state's actions.


Note

An object node can only be a leaf element; its meaning cannot contain other elements.

As with other states, only one object node can have the same name in the same parent state.

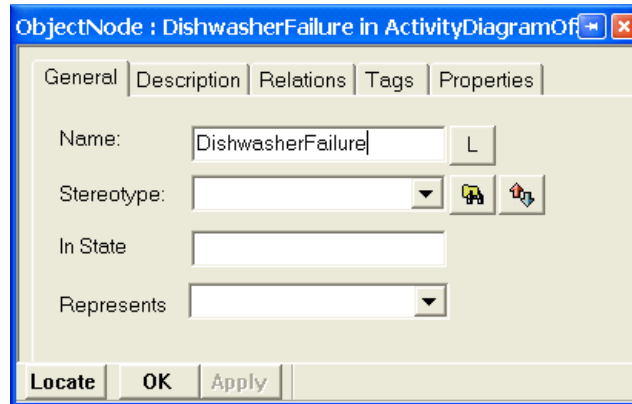
Creating an Object Node



To create an object node, follow these steps:

1. Click the **Object Node**  icon on the **Drawing** toolbar.
2. Click or click-and-drag in the activity diagram to place the node at the desired location.

Modifying the Features of an Object Node

The Features dialog box enables you to change the features of an object node, including its name and stereotype. The following figure shows the Features dialog box for an object node.



- ◆ **Name**—Specifies the name of the object node. The default name is `state_n`, where *n* is an incremental integer starting with 0. The description of the action block can be entered into the text area on the **Description** tab.
 - ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
 - ◆ **Stereotype**—Specifies the stereotype of the object node, if any. They are enclosed in guillemets, for example `«s1»` and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.
 - To select from a list of current stereotypes in the project, click the folder with binoculars  button.
 - To sort the order of the stereotypes, click the up and down arrows  button.
- Note: The COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM Development Guide* for more information about these components.
- ◆ **In State**—Specifies the required states of the object node.
 - ◆ **Represents**—Allows you to select a class/type to associate with the node from a pull-down list of project classes/types or enter the name of a new class. See [Associating an Object Node with a Class](#) for more information.

Associating an Object Node with a Class

In an activity diagram, you may associate an object node to a class/type to represent the class type using one of the following methods:

- ◆ Right-click the object node in the diagram to display the Features dialog box and select the class/type from the pull-down list in the **Represents** field.


- ◆ Dragging a class/type from the browser into the activity diagram automatically associates the class/type with the object node.

In the activity diagram, the associated node displays the name of its associated class/type in the name compartment.

Adding Call Behaviors

A *call behavior* represents a call to an operation of a classifier. This is only used in modeling and does not generate code for the call operation. However, code can be inserted manually into the **Action** field in the Call Behavior Features dialog box.

To create an call operation node, use one of these methods:

- ◆ Click the **Call Behavior** icon  on the **Drawing** toolbar and drawing the behavior operation in the diagram.
- ◆ Click or click-and-drag an operation from the browser in the activity diagram and place the behavior at the desired location.

Note

If the behavior operation dropped in the activity diagram has no association with the classifier, an empty call is created.

Activity Flows or Transitions

Activity diagrams can have activity flows, default flows, and loop activity flows on action blocks and join transitions. These transitions in activity diagrams are the same as the corresponding transitions in statecharts, with the following exceptions:

- ◆ Outgoing transitions from states can have triggers, but those from actions, action blocks, or subactivities cannot.
- ◆ Outgoing transitions from actions, action blocks, and subactivities can have only guards and actions.

Transitions exiting or entering fork or join synchronization bars have the following constraints:

- ◆ They cannot have labels.
- ◆ They must originate in, or target, either states or actions—not connectors.

Creating a Transition

To draw a transition from one state to another, follow these steps:

1. Click the **Activity Flow**  icon on the **Drawing** toolbar.

2. Click the edge of the source state.
3. Drag the cursor to the edge of the target state and release to anchor the transition.
4. If desired, enter a trigger for the transition.
5. If desired, enter a guard and action for the transition.


Completion Transitions

A transition to a termination state is called a *completion transition*. Neither final states nor termination states can have outgoing transitions. A completion transition does not have an explicit trigger, although it can have a guard condition.

Default Flow

The default flow points to the state that the object, use case, or operation enters when the activity starts.

To draw a default flow, follow these steps:


1. Click the **Default Flow** icon  on the **Drawing** toolbar.
2. Click in the activity diagram outside the default state.
3. Drag the cursor to the edge of the default state of the activity and release the mouse button.

See [Statecharts](#) for more information on default flows.

Loop Activity Flow

Loop activity flows (also known as *self transitions*) represent looping behavior in a program. Loop activity flows are often used on action blocks to indicate that the block should loop until some exit condition becomes true.


To draw a loop activity flow, follow these steps:

1. Click the **Loop Activity Flow** icon  on the **Drawing** toolbar.
2. Click the edge of any kind of state.
3. Label the loop activity flow.

See [Termination States](#) for an example of an action block with a loop activity flow. See [Statecharts](#) for more information on loop (self) transitions.

Transition Labels

To add or modify a transition label, follow these steps:

1. Click the **Transition Label** icon  on the **Drawing** toolbar.
2. Select the transition you want to label.
3. In the edit box, type the new label (or modify the existing one).
4. Press **Ctrl+Enter** or click outside the label to terminate editing.

Modifying Transitions

As with all other elements, you can modify the features of a transition using the Features dialog box. See [Modifying the Features of a Transition](#) for more information.

Connectors


Activity diagrams can have the following connectors:

- ◆ Junction connectors
- ◆ Condition
- ◆ Diagram

The following sections describe these connectors in detail.

Junction Connectors

To draw a junction connector, follow these steps:


1. Click the **Junction Connector** icon  on the **Drawing** toolbar.
2. Click in the activity diagram to place the junction at the desired location.
3. Draw transitions going into, and one transition going out of the junction.
4. Label the transitions as desired.

See [Statecharts](#) for more information on junction connectors.

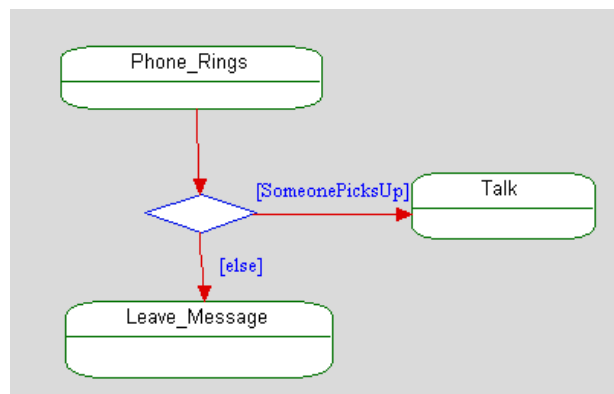
Condition Connectors

Condition connectors show branching conditions. A condition connector can have only one incoming transition and two or more outgoing transitions. The outgoing transitions are labeled with a distinct guard condition and no event trigger. A predefined guard, denoted `[else]`, can be used for no more than one outgoing transition.

To draw a condition connector, follow these steps:

1. Click the **Condition Connector** icon  on the **Drawing** toolbar.
2. Click, or click-and-drag, in the activity diagram to position the condition connector at the desired location.
3. Draw at least two states that will become targets of the outgoing transitions.
4. Draw an incoming transition from the source state to the condition connector.
5. Draw and label the outgoing transitions from the condition connector to the target states.

Condition connectors look like diamonds, as shown in the following figure.




This activity diagram shows the following behavior: When the phone rings, if someone picks up on the other end, you can talk; otherwise, you must leave a message. The condition connector represents the decision point. In other words, after the `PhoneRings()` operation, if `SomeonePicksUp` resolves to `True`, the `Talk()` operation is called. Otherwise, the `LeaveMessage()` operation is called.

Use the **Display Options** option in the pop-up menu to determine whether to display the name, label, or nothing for the condition connector.

Diagram Connectors

Diagram connectors connect one part of an activity diagram to another part on the same diagram. They represent another way to show looping behavior.

To draw a diagram connector, follow these steps:

1. Click the **Diagram Connector**  icon on the **Drawing** toolbar.
2. Click to place the source diagram connector at the desired location and label the connector.
3. Repeat to place the target diagram connector at the desired location in the diagram, and give it the same name as the source connector.

See [Statecharts](#) for more information on diagram connectors.

Synchronization Bars

Activity diagrams can include synchronization bars. A *synchronization bar* depicts either a join or a fork operation. You can draw join synchronization bars in activity diagrams for objects, use cases, and operations.


Rhapsody defines activity diagrams as meaningful only if join and fork synchronization bars are well-structured in the same sense as well-structured parentheses. In other words, they must use proper nesting. The only exception to this rule is that a join or fork connector with multiple ingoing/outgoing transitions can be used in place of a number of join or fork connectors with only two ingoing or outgoing transitions each.

Rhapsody tolerates less-than-meaningful activity charts, provided that they can be extended into meaningful ones by adding transitions (for example, a fork with transitions that never merge back).

As you draw activity diagrams, Rhapsody prevents you from drawing constructs that violate the meaningfulness of the activity diagram by displaying a “no entry” symbol.

Creating Join Synchronization Bars

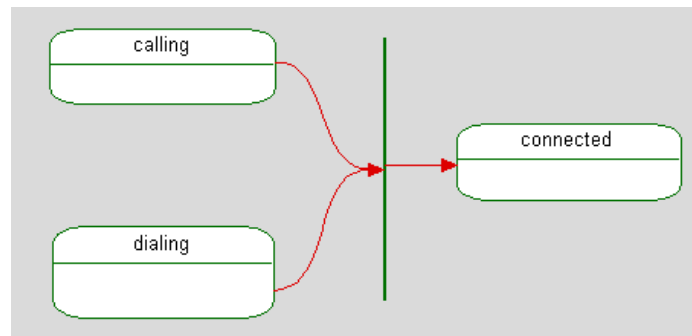
To draw a join synchronization bar, follow these steps:

1. Draw at least two states that you want to synchronize.
2. Click the **Draw Join Synch Bar** icon  on the **Drawing** toolbar.
3. Click or click-and-drag in the activity diagram to place the join synchronization bar in the desired location.

By default, the join line is drawn horizontally. To flip it, see [Rotating Synchronization Bars](#).


4. Draw incoming transitions from each of the source states to the join synchronization bar.
5. Draw a state that will be the target of the outgoing transition.
6. Draw an outgoing transition from the join synchronization bar to the target state.

A join synchronization is shown as a bar with two or more incoming transition arrows and one outgoing transition arrow, as shown in the following figure.



Creating Fork Synchronization Bars

To draw a fork synchronization bar, follow these steps:

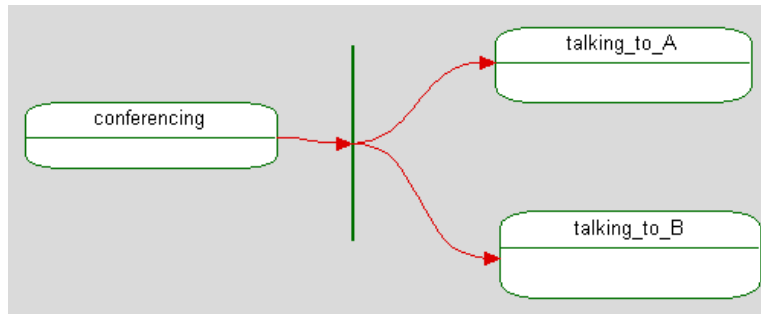
1. Click the **Draw Fork Synch Bar** icon  on the **Drawing** toolbar.
2. Click, or click-and-drag, in the activity diagram to place the fork synchronization bar at the desired location. Fork synchronization bars can be vertical or horizontal only.

By default, the join line is drawn horizontally. To flip it, see [Rotating Synchronization Bars](#).

3. Draw a source state and an incoming transition coming into the fork synchronization bar.

4. Draw the target states and the outgoing transitions from the fork synchronization bar.

A fork synchronization is shown as a heavy bar with one incoming transition and two or more outgoing transition arrows, as shown in the following figure.



Rotating Synchronization Bars

You can rotate a synchronization bar to the right (clockwise) or left (counter-clockwise).

To rotate a synchronization bar, right-click the synchronization bar and select either **Flip Right** or **Flip Left** from the pop-up menu.

Stretching Synchronization Bars

To stretch a synchronization bar, follow these steps:

1. Select the synchronization bar.
2. Click-and-drag one of the highlighted selection handles until the synchronization bar is the desired length.

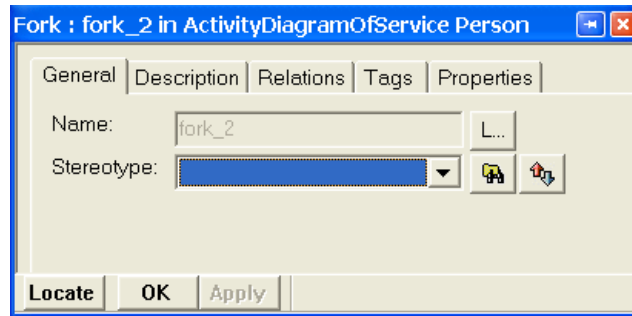
Moving Synchronization Bars

To move a synchronization bar to a new location, follow these steps:



1. Click in the middle of the synchronization bar, away from the selection handles, and drag the bar to the desired location.
2. Release the mouse button to place the synchronization bar at the new location.

Modifying Synchronization Bars

As with all other elements, you can modify the features of a synchronization bar using the Features dialog box. The following figure shows the Features dialog box for a fork synchronization bar.



The fields and buttons are as follows:

- ◆ **Name**—Specifies the name of the element. The default name is <element>_n, where *n* is an incremental integer starting with 0. Add any additional information using the **Description** tab.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.
 - To select from a list of current stereotypes in the project, click the folder with binoculars  button.
 - To sort the order of the stereotypes, click the up and down arrows  button.

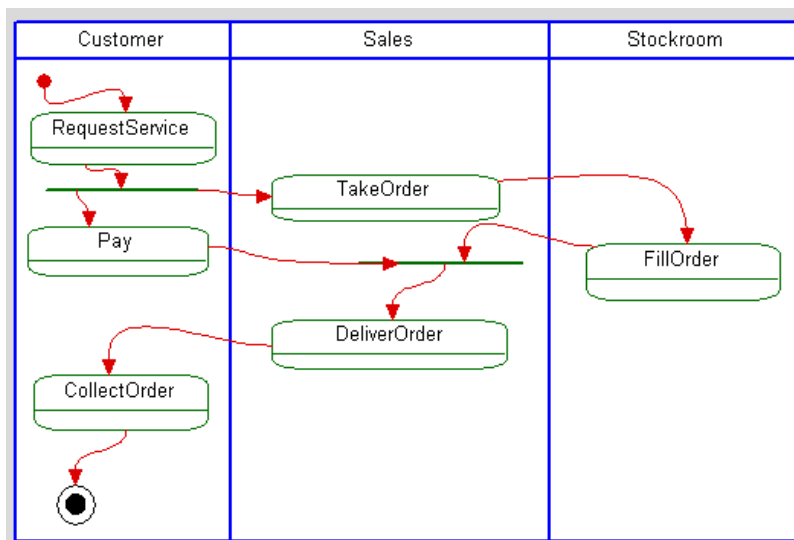
Note: The COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM Development Guide* for more information about these components.

Swimlanes

Swimlanes divide activity diagrams into sections. Each swimlane is separated from adjacent swimlanes by vertical, solid lines on both sides. The following are features of swimlanes:

- ◆ Each action is assigned to one swimlane.
- ◆ Transitions can cross lanes.
- ◆ Swimlanes do *not* change ownership hierarchy.
- ◆ Swimlane association (representing field population) to a class (only) can be created by dragging the class from the browser to the Swimlane name compartment.
- ◆ The relative ordering of swimlanes has no semantic significance.
- ◆ There is no significance to the routing of a transition path.

The following figure shows an activity diagram with swimlanes.





Creating Swimlanes

To use swimlanes in an activity diagram, you first need to create a swimlane frame. If you do not, Rhapsody generates an error message.

Note

There can be only one swimlane frame in an activity diagram. Once you have created a frame, the **Swimlane Frame** tool is grayed out.

To draw a swimlane, follow these steps:

1. Click the **Swimlane Frame** icon  on the **Drawing** toolbar.
2. The cursor turns into crosshairs. In the drawing area, click one corner to draw the swimlane frame (a box).
3. Click the **Swimlane Divider** icon  on the **Drawing** toolbar.
4. The cursor turns into a vertical bar. When it is at the desired location, left-click to place the divider. Rhapsody creates two swimlanes, named `swimlane_n` and `swimlane_n+1`, where n is an incremental integer starting at 0.

If you draw another divider, it divides the existing swimlane into two swimlanes, with the new swimlane positioned to the *left* of the divider.

Note: You cannot draw a swimlane on an existing state.

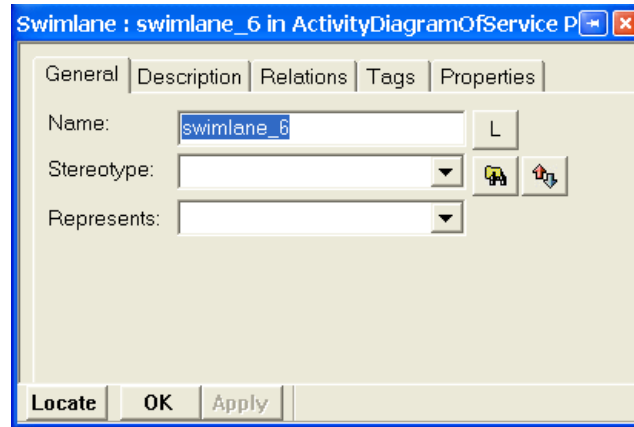
5. If desired, rename the swimlanes using the Features dialog box.

Note the following:



- ◆ Swimlanes have a minimum width. If you enlarge a swimlane, the extra space is added to the right of the swimlane. To resize a swimlane, move the divider to the left or the right.
- ◆ If a swimlane contains activity diagram elements, you cannot reduce the size of that swimlane so its divider is positioned to the left of any of those elements, because that would force the elements into a different swimlane.
- ◆ A swimlane maps into a partition of states in the activity diagram. A state symbol in a swimlane cases the corresponding state to belong to the corresponding partition.

Modifying the Features of a Swimlane

The Features dialog box enables you to change the features of a swimlane, including its name and description. The following figure shows the Features dialog box for a swimlane.

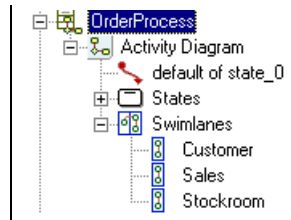


A swimlane has the following features:

- ◆ **Name**—Specifies the name of the element. The default name is `swimlane_n`, where n is an incremental integer starting with 0. Add any additional information using the **Description** tab.
 - ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
 - ◆ **Stereotype**—Specifies the stereotype of the swimlane, if any. They are enclosed in guillemets, for example `«s1»` and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.
 - To select from a list of current stereotypes in the project, click the folder with binoculars  button.
 - To sort the order of the stereotypes, click the up and down arrows  button.
- Note: The COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM Development Guide* for more information about these components.
- ◆ **Represents**—Specifies the class to which the swimlane applies.

Viewing Swimlanes in the Browser

Swimlanes are displayed in the browser under the activity diagram, as shown in the following figure.



Note that swimlane nodes cannot be deleted.

Deleting Swimlane Dividers

To delete a swimlane, follow these steps:

1. Select the divider you want to delete.
2. Click the **Delete** icon.

Deleting the Swimlane Frame

To delete the swimlane frame and all its swimlanes, follow these steps:

1. Select the frame.
2. Click the **Delete** icon.

Note that after the deletion, the frame and swimlanes are removed, but the elements they contained still exist in the activity diagram.


Swimlane Limitations

Note the following design and behavior limitations apply to swimlanes:

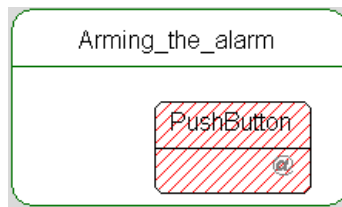
- ◆ You should not draw actions on swimlane divider lines. In other words, do not draw actions that overlap into another swimlane.
- ◆ Swimlanes cannot be inherited; therefore, inheritance graphics behave in the following way:
 - Elements drawn in swimlanes in a base class diagram are drawn as usual.
 - When there are swimlanes in a derived diagram, elements drawn in base diagrams that should be *inside* those swimlanes are positioned *outside* of them.
- ◆ Swimlanes in subactivity diagrams are not supported.
- ◆ You cannot use the browser to create or drag-and-drop swimlanes.

Adding Calls to Behaviors

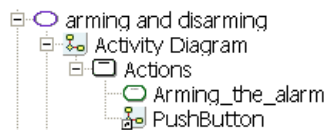
You can add a call to a behavior in another activity diagram or to the entire activity diagram. You may add calls to both activity diagrams and subactivity diagrams.

To add a call, either click the **Call Behavior**  icon on the **Drawing** toolbar, or drag-and-drop the activity (or activity diagram) from the browser into the activity diagram. Rhapsody creates the call in the activity diagram and in the browser.

The called behavior has the same name as the called object. The called behavior, `PushButton`, is marked with an at-symbol icon, as shown in this example.



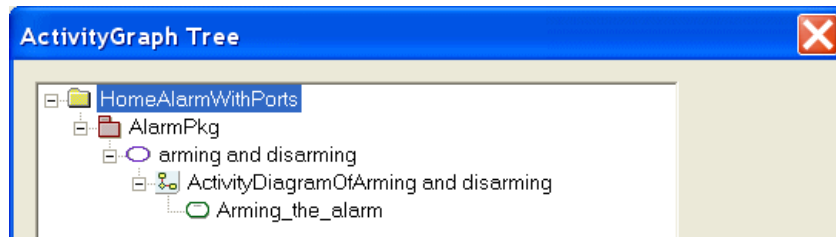
This is displayed in the project browser under the activity diagram containing the behavior.



Modifying a Called Behavior

To modify the features of a called behavior, follow these steps:

1. Highlight the called behavior in either the browser or diagram. Right-click and select **Features**.
2. Click the right arrow button beside the **Reference** field in the **General** tab.
3. The **ActivityGraph Tree** dialog box allows you to navigate to the activity diagram that the called behavior represents, as shown below.



Displaying Called Behaviors

As with most elements, use the **Display Options** option in the pop-up menu to define the display of called behaviors with one of these options:

- ◆ Show the name, label, or nothing for the activity
- ◆ Show the name, label, or icon for the stereotype

Called Behavior Limitations

Note the following behavior and restrictions:

- ◆ Called behaviors cannot be inherited.
- ◆ You cannot “undo” changes to called behaviors.
- ◆ A called behavior cannot be created in the browser.
- ◆ There is no code generation for called behaviors because the code generator ignores these calls and the transitions that go in and out of them.

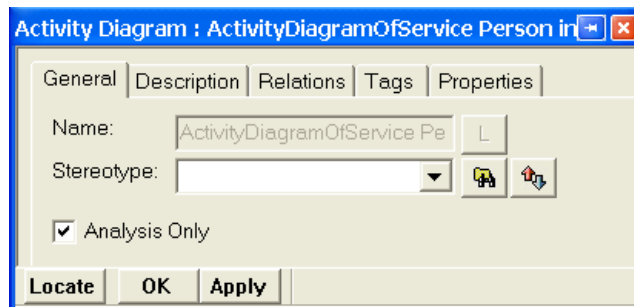
Adding Action Pins / Activity Parameters to Diagrams

Action pins can be added to actions and activity parameters can be added to action blocks in an activity diagram. These elements represent the inputs and outputs for the relevant action/action block. Action pins can also be added to subactivities.

Action pins and activity parameters are diagram elements and appear in the browser. However, there is no code generated for these elements.

To make the action pin tool available for use, follow these steps:


1. Highlight an element in the browser that needs an activity diagram for analysis purposes only.
2. Right-click the element and select the **Add New > Activity Diagram** options from the menus.
3. Right-click the *Activity Diagram* now listed in the browser.
4. Select **Features** and click the **Analysis Only** check box, as shown in this example.



5. Click **OK**. Rhapsody displays a message asking if you intend to create an analysis only activity diagram. If you click **Yes**, this diagram cannot be used for other purposes than analysis.


The system then places the Action Pin and Activity Parameter icons on the **Drawing** toolbar.

To use the action pin, follow these steps:

1. Click the small **Action Pin** icon  at the bottom of the **Drawing** toolbar.
2. Click the action to which the pin should be added.

The pin appears on the border of the action closest to the point that was clicked. The name appears alongside the pin. The default name is `pin_n`.

To add an activity parameter, follow these steps:

1. Click the **Activity Parameter** icon  at the bottom of the **Drawing** toolbar.
2. Click the action block to which the activity parameter should be added.

The activity parameter appears on the border of the action block closest to the point that was clicked. The name appears inside the activity parameter node. The default name is `parameter_n`.

Modifying Features of Action Pins / Activity Parameters

The **Features** dialog box enables you to change the following features of action pins / activity parameters, in addition to the standard fields:

- ◆ Direction (in, out, inOut)
- ◆ Argument Type (for example, int)

Graphical Characteristics of Action Pins / Activity Parameters

Action pins and activity parameter nodes have the following graphical characteristics:

- ◆ Pins/parameters always remain attached to their action/action block. However, they can be moved around the perimeter of the action/action block.
- ◆ Action pins are a fixed size. Activity parameters, however, can be resized.
- ◆ Pins/parameters cannot be copied and pasted.
- ◆ All pin/parameter operations can be undone.
- ◆ In the browser, pins/parameters appear beneath the action/action block to which they belong.
- ◆ When an action pin / activity parameter is deleted from a diagram, it is removed from both the view and the model.

Other Characteristics of Action Pins / Activity Parameters

- ◆ Appear in reports generated by the internal Rhapsody reporter.
- ◆ Appear in reports generated by ReporterPLUS.
- ◆ Are exported to DOORS.
- ◆ Are exported to XMI files using the XMI toolkit.
- ◆ Supported in DiffMerge.
- ◆ Appear in the search dialog box.
- ◆ Supported in the Rhapsody API. It returns the pins/parameters associated with an action, as well as the type of each pin/parameter.
- ◆ Are inherited as part of activity diagram inheritance, that is, if you add a pin/parameter to the activity diagram of a base class, the pin/parameter will also appear in the diagram of the class that inherits from it.

Local Termination Semantics

Local termination means that once an action block is entered, it can be exited by a null transition only if its final state has been reached. A null transition is any transition without a trigger (event or timeout). A null transition can have a guard.

Statechart Mode

The following sections describe how local termination is implemented in statechart mode for various kinds of states.

Or States

The following local termination rules apply to Or states:

- ◆ An Or state that has a termination state is completed when the termination state is reached.
- ◆ An Or state without a termination state is completed after finishing its entry action.
- ◆ An outgoing transition from an Or state can have a trigger (as with statecharts).
- ◆ An outgoing null transition (transition without a trigger) from an Or state can be taken only if the Or state is completed.
- ◆ If an Or state with a termination state has a history connector, the last state of the history connector is always the termination state, after it has been reached once.
- ◆ An Or state can be exited by any transition, including a null transition, from one of its substates (as with statecharts).

Leaf States

A leaf state is completed after finishing its entry action.

Component States

The following local termination rules apply to component states:

- ◆ Because a component state is a kind of Or state, all the local termination rules for Or states also apply to component states.
- ◆ A join transition (from several component states) is activated only if all the sources (component states) are completed.

And States

An outgoing null transition from an And state is activated only if all of its components are completed.

IS_COMPLETED() Macro

You can use the `IS_COMPLETED()` macro in a statechart to test whether a state is completed. Completion means that any of the conditions for local termination described in the previous sections are true. The macro works the same for both flat and reusable implementations of statecharts.

The `CG::Class::IsCompletedForAllStates` property specifies whether the `IS_COMPLETED()` macro can be used for all kinds of states. The default value of `Cleared` means that the macro can be used only for states that have a termination state. `Checked` means that it can be used for all states.

Activity Diagram Mode

All of the local termination rules for statechart mode also apply in activity diagram mode, with the following exceptions:

- ◆ An Or state that has a termination state is completed when the termination state is reached.
- ◆ An Or state without a termination state is never completed.

Code Generation

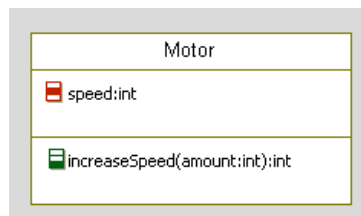
In previous releases, Rhapsody supported code generation only from activity diagrams associated with classes, not from activity diagrams associated with operations or use cases.

Rhapsody in C++ generates functor-based code for activity diagrams associated with operations. *Functor-based code* reuses the code generation functionality for activity diagrams of classes. Code is generated into a new class (called the *functor class*), which implements an activity diagram on the class level. The task of executing the *modeled operation* (an operation associated with an activity diagram) is delegated to the new class. The class that delegates the task is known as the *owner class*.

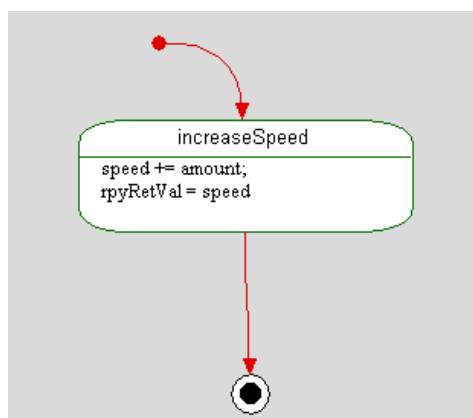
You specify whether to generate code for an activity diagram by setting the property `CPP_CG::Operation::ImplementActivityDiagram` to `Checked`. This property is set to `Cleared` by default.

Functor Classes

Consider the following class:



The following figure shows the activity diagram associated with the `increaseSpeed` modeled operation.



In the example model, `Motor` is the owner class, and `FunctorIncreaseSpeed_int` executes the modeled operation `increaseSpeed`.

This activity diagram increments the value of the class attribute `speed` by the value specified by the `amount` argument, then returns the incremental value. Note the following:

- ◆ An activity diagram should never contain code with the `return` keyword because the current implementation executes the code fragments of the diagram (that appear in actions, transitions, and so on) in contexts of different operations. Returning from those contexts (operations) does not have the same effect as returning from the body of a regular operation.

Instead, you should set the return value to the `rpRetVal` variable (shown in the previous figure), which is always generated for operations that return values.

- ◆ There are two ways an operation can finish and return control to its caller:
 - Once the diagram reaches a “stable” state (there are no more transitions to take), the operation is considered to have finished its job and it returns control to its caller.
 - The diagram reaches a termination connector at the top level.
- ◆ The actual execution of the code does not occur within the scope of the object that owns the operation. Instead, the code is executed in another object whose sole purpose is to execute the diagram. During the execution of the code in the diagram, the `this` pointer references a special-purpose object (the *functor object*) that contains the code in the diagram (in states, on transitions, and so on). To refer to the owner object, you can use the `this_` variable, which always exists for this purpose.

To make coding more natural, direct access to the attributes of the class that owns the operation (without the need for `this_`) is made possible by supplying references in the functor object. The functor class contains references that correspond to each owner class attribute and have the same name. The constructor of the functor class contains arguments to initialize each attribute; initialization is done when the owner class creates an object of the functor class.

You can control the code generation of the attribute references by setting the value of the `CPP_CG::Operation::ActivityReferenceToAttributes` property to `Checked` (the default value).

Limitations and Specified Behavior

Note the following restrictions and behavior:

- ◆ Existing inheritance within an activity diagram is supported, but you cannot create a new inherited activity diagram.
- ◆ Activity diagrams do not require “And” states to represent concurrent behavior—activity diagrams cannot include “And” states.
- ◆ Forks can end in states that are not within two orthogonal states.
- ◆ Activity diagrams for operations cannot receive events, nor can they have state nodes.
- ◆ You cannot animate the activity diagrams associated with operations. However, an animated sequence diagram for the model records the fact that a modeled operation was called. In addition, the call and object stacks record the owner object (not the functor object) as the object that receives the message.
- ◆ This feature supports only a subclass of diagrams that do not contain events (including timeout events) or triggered operations.
- ◆ If a class attribute and an argument of the modeled operation have the same name, there will be a name collision that results in a compiler error. To avoid this problem before attempting to generate code, omit the class attribute that causes the collision. This should not affect the semantics of the operation, because operation arguments hide class attributes.
- ◆ The name of the functor class will contain the signature of the modeled operation, thereby supporting overloaded operations. Complicated type names will be converted to strings appropriate for building C++ identifiers. For example, “:” characters are replaced with underscores.
- ◆ There is no support for diagram inheritance in the case of an operation that overrides a modeled operation.
- ◆ Because the code in the diagram does not execute within the scope of the modeled operation, there is no direct access to the `this` variable. Instead, access this variable using the `this_` attribute of the functor class. This is also the preferred way to invoke methods in the owner class.
- ◆ Operations with variable-length argument lists are not supported.
- ◆ Global operations with activity diagrams are not supported.

Flow Charts

A *flow chart* is a schematic representation of an algorithm or a process. In UML and Rhapsody, you can think of a flow chart as a subset of an activity diagram that is defined on methods and functions.

You can model methods and functions using flow charts in all Rhapsody programming languages. Only in Rhapsody in C and Rhapsody in C++ can readable structured code be generated from a flow chart. During code generation, for the actions defined in a flow chart Rhapsody can generate structured code for If/Then/Else, Do/While, and While/Loops.

The code generator algorithm for a flow chart can identify Loops and Ifs—the expressions for these constructs is on the guards of the action flows.

For more information about activity diagrams, see [Activity Diagrams](#).

Defining Algorithms with Flow Charts

One useful application of flow charts is in the definition of algorithms. *Algorithms* are essentially decompositions of functions into smaller functions that specify the activities encompassed within a given process.

This flow chart approach to code generation is to reduce the diagram to blocks of sequential code and then search for If/Loop patterns in those blocks. The following structured programming control structures are supported in flow charts:

- ◆ Simple If
- ◆ If/Then/Else
- ◆ While loops (where the test is at the start of the loop)
- ◆ Do/While loops (where the test is at the end of the loop)

If the algorithm does not succeed to impose the above structure, then it will need to use a GoTo.

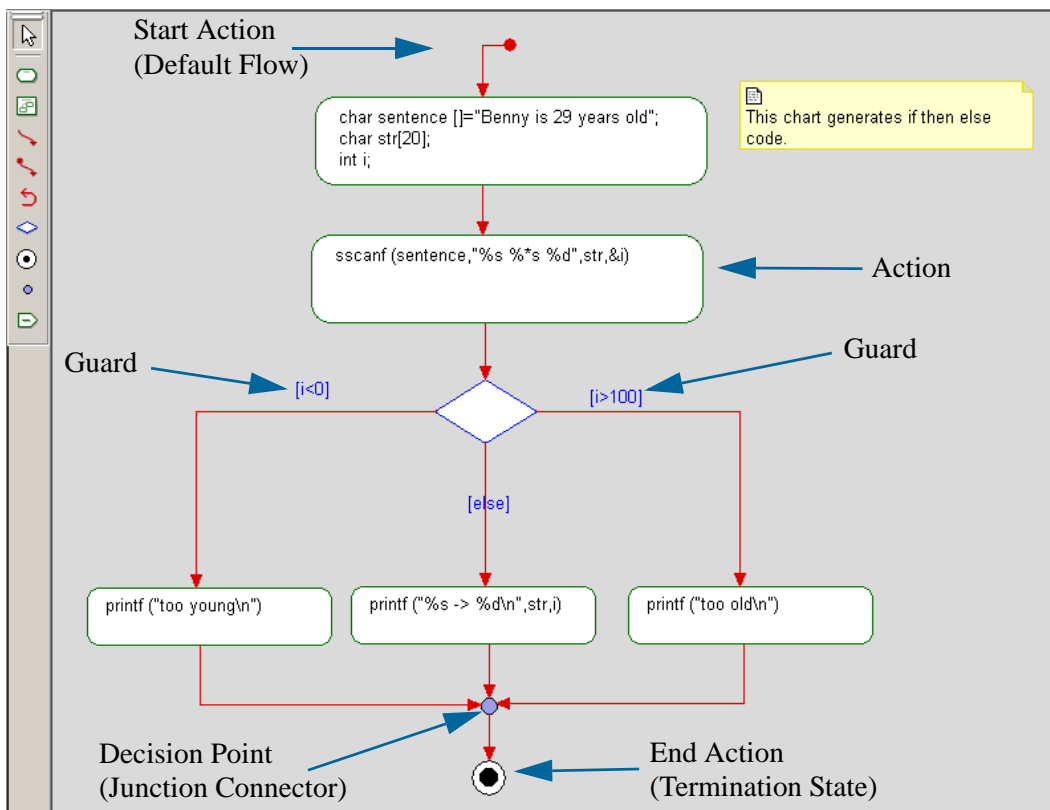
Flow Charts Similarity to Activity Diagrams

Flow charts have the following elements in common with activity diagrams including start and end activities and *actions*:

- ◆ **Decision points** that show branching points in the program flow based on guard conditions.
- ◆ **Actions** that represent function invocations with a single exit action flow taken when the function completes. It is not necessary for all actions to be within the same object.
- ◆ **Action blocks** that represent compound actions that can be decomposed into actions.

However, flow charts do NOT include And states, and flow charts for operations cannot receive events.

Rhapsody in C includes a Flowchart model located in the `<Rhapsody installation>\Samples\CSamples\Flowchart` folder. These sample flow charts show which flow chart patterns are recognized in order to generate structured code. The following example shows the main elements of a flow chart:












Creating Flow Chart Elements

The following sections describe how to use the flow chart drawing tools to draw the parts of a flow chart. See [Graphic Editors](#) for basic information on diagrams, including how to create, open, and delete them.

Flow Chart Drawing Icons

The **Drawing** toolbar for a flow chart contains the following tools.

Drawing Icon	Description
	Action represents the member function call within a given operation. See Actions for more information.
	Action Block represents compound actions that can be decomposed into actions. See Action Blocks for more information.
	Activity Flow defines the flow and its guards to supply the test for Ifs or Loops. Activity flows without guards define the default sequential flow of the flow chart. See Activity Flows for more information.
	Default Flow shows the flow origination point from an element. For flow charts, this default flow is the initial flow, and for code purposes, indicates the start of code. See Default Flow for more information.
	Loop Activity Flow represents behavior repeating in a program. See Loop Activity Flows for more information.
	Condition Connectors show branching conditions. See Condition Connectors for more information.
	Termination State provides local termination semantics. The flow chart returns at this point to the operation/function that invoked it. See Termination States for more information.
	Junction Connector combines different flows to a common target. See Junction Connectors for more information.
	Send Action State represents the sending of events to external entities. See Send Action State Elements for more information.

Actions

Flow charts decompose a system into actions that correspond to activities. These diagrammatic elements, called *actions*, are member function calls within a given operation. In contrast to normal states (as in statecharts), actions in flow charts terminate on completion of the activity, rather than as a reaction to an externally generated event.

Each action can have an entry action, and must have at least one outgoing action flow. The implicit event trigger on the outgoing action flow is the completion of the entry action. If the action has several outgoing action flows, each must have its own guard condition.

During code generation, code is derived from the actions on a flow chart.

Actions have the following constraints:

- ◆ Outgoing activity flows can include only guard conditions.
- ◆ Actions have non-empty entry actions.
- ◆ Actions do not have internal action flows or exit actions, nor do activities.
- ◆ Outgoing action flows on actions have no triggering events.

Creating a Flow Chart

You can create a flow chart on any function or class method in the same way as you can an activity diagram.


To create a flow chart, follow these steps:

1. In the browser, right-click the model element for which you want to create a flow chart, such as a function.
2. Select **Add New > Flowchart**.

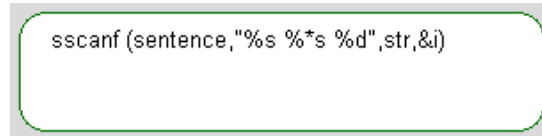
The flow chart displays in the Drawing area with a title including the selected model element.

Drawing an Action

To draw an action, follow these steps:

1. Create a flow chart.
2. Click the **Action** button  on the **Drawing** toolbar.
3. Click or click-and-drag in the flow chart to place the action where you want it.
4. Type a name for the action, then press **Ctrl+Enter** or click the Select arrow in the toolbar to terminate typing mode.

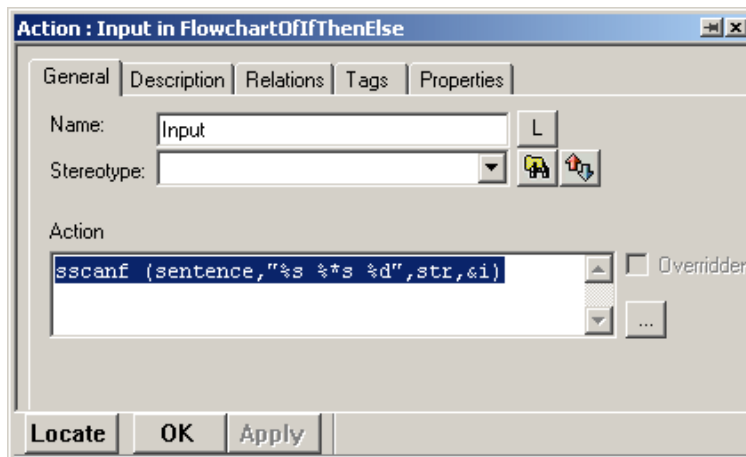
Actions have rectangles with curved edges, as shown in the following figure.



By default, the action expression, which does not need to be unique within the flow chart, is displayed inside the action symbol. See [Displaying an Action](#) for information on modifying the display.

Modifying the Features of an Action

The Features dialog box enables you to add and change the features of an action, including its name and action. The following figure shows the Features dialog box for an action.



An action has the following features:

- ◆ **Name** specifies the name of the action. The description of the action can be entered into the text area on the **Description** tab. This description can include a hyperlink. See [Hyperlinks](#) for more information.
- ◆ **L** specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype** specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that, by default, flow charts have a new term stereotype of “flowchart.”

Note also that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.

- ◆ **Action** specifies an action in a flow chart. This is the text you typed into the flow chart when you created the action.

The **Overridden** check box allows you to toggle the check box on and off to view the inherited information in each of the dialog box fields and decide whether to apply the information or revert back to the currently overridden information.

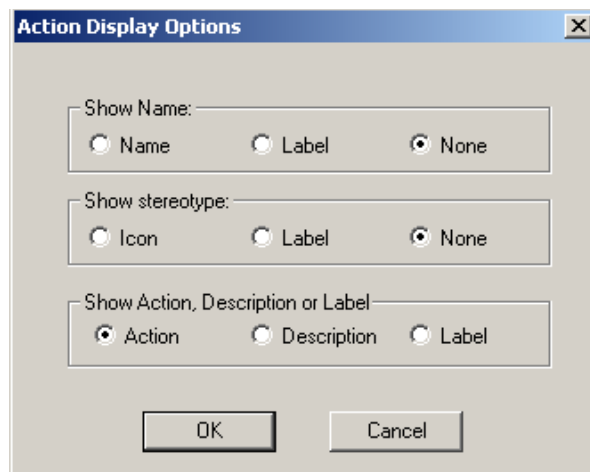
For more information about the Features dialog box, see [Using the Features Dialog Box](#).

Displaying an Action

You can show the name, action, or description of the action in the flow chart.

To specify which attribute to display, follow these steps:

1. Right-click the action and select **Display Options** from the pop-up menu. The Display Options menu is displayed, as shown in the following figure.



2. Select the appropriate values.
3. Click **OK**.


Action Blocks

Action blocks represent compound actions that can be decomposed into actions. Action blocks can show more detail than might be possible in a single, top-level action. You can also use pseudocode, text, or mathematical formulas as alternative notations.

Note that for action blocks, there must be a default flow at the start and a termination state at the end, and activity flows cannot cross the blocks boundaries to actions in the block (though they may enter and leave the block itself). The code generator will put blocks code in curly braces and this has language significance regarding variable scope and lifetime.

Creating an Action Block

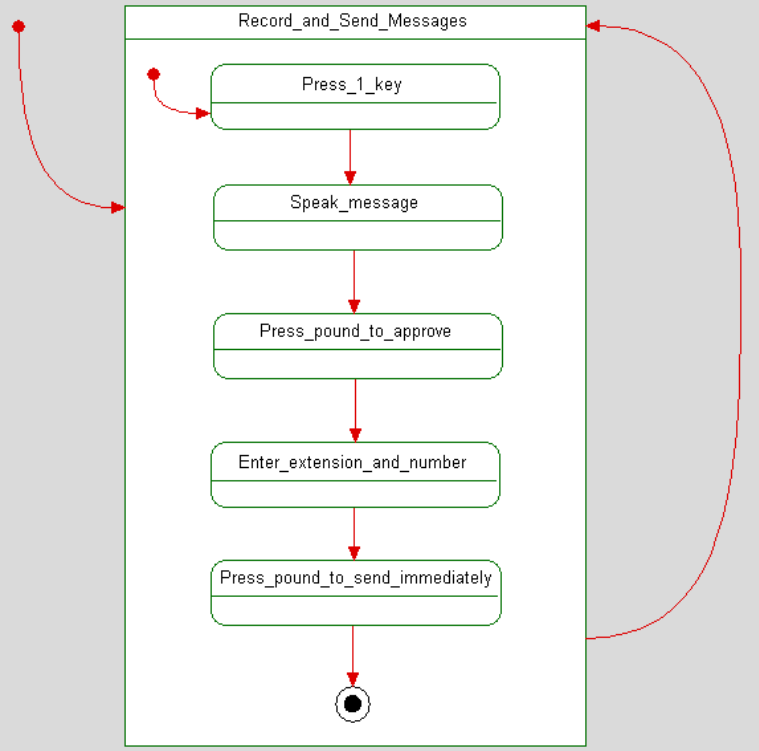
To define the activity, draw an action block using these steps:

1. Click the **Action Block** button  on the **Drawing** toolbar.
2. Click or click-and-drag in the flow chart to place the action block where you want it.
3. Draw actions and activity flows inside the action block to express the activity being modeled.

Action blocks are rectangles. The name of the blocked activity is shown at the top.

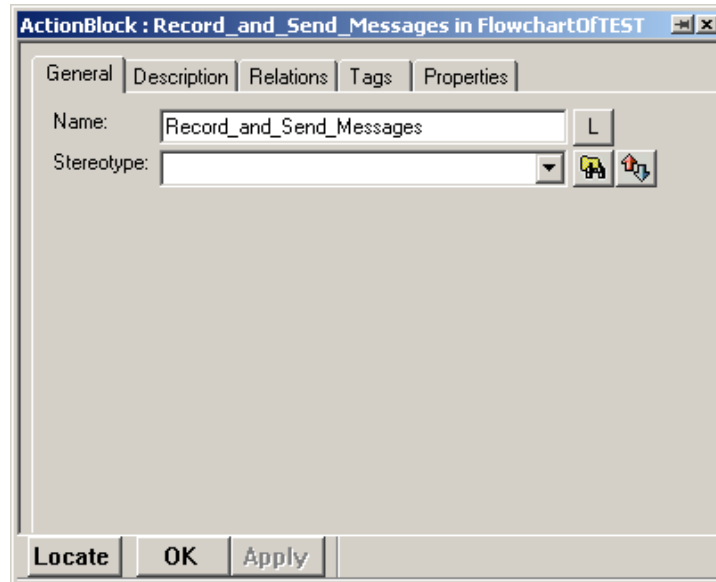
Flow Charts

The Record_and_Send_Messages activity shown in the following sample action block encompasses several activities.



Modifying the Features of an Action Block

The Features dialog box enables you to change the features of an action block, including its name and description. The following figure shows the Features dialog box for an action block.



An action block has the following features:

- ◆ **Name** specifies the name of the action block. The description of the action block can be entered into the text area on the **Description** tab. This description can include a hyperlink. See [Hyperlinks](#) for more information.
- ◆ **L** specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype** specifies the stereotype of the action block, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that, by default, flow charts have a new term stereotype of “flowchart.”

Note also that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.


Termination States

A *termination state* provides local termination semantics. The flow chart returns at this point to the operation/function that invoked it.

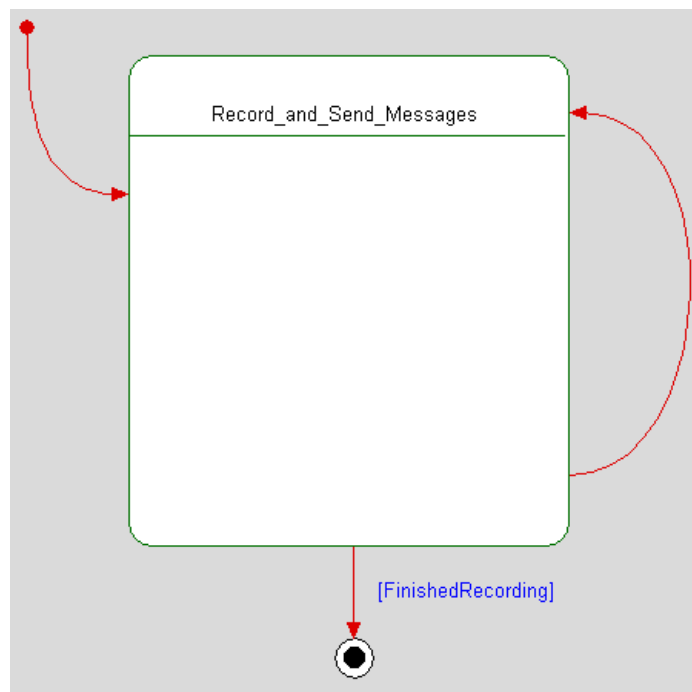
Note

The behavior of termination states is controlled by the `CG::Statechart::LocalTerminationSemantics` property. See the definition provided for the property on the applicable **Properties** tab of the Features dialog box.

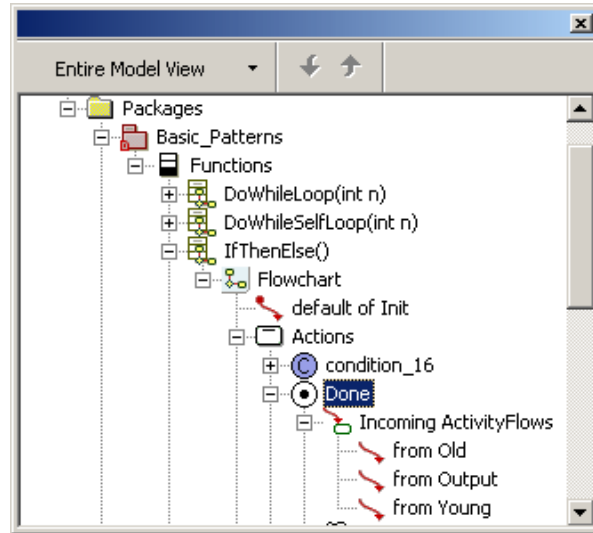
To create a termination state, follow these steps:

1. Click the **Termination State** button  on the **Drawing** toolbar.
2. Click in the flow chart to place the termination state where you want it.
3. Draw an activity flow from an action to the termination state.
4. If you want, enter a guard condition to signal the end of the activity.

A termination state looks like a circle with a black dot in its center, as shown in the following figure.



As with the other connectors, termination states and their flows are included in the Rhapsody browser.



Send Action State elements

For details regarding the use of Send Action State elements, see [Send Action State Elements](#).

Activity Flows

Flow charts can have activity flows (such as activity flows, default flows, and loop activity flows) on actions and action blocks. Activity flows define flow and their guards supply the test for Ifs or Loops. Activity flows without guards define the default sequential flow of the flow chart.

In addition, note the following:


- ◆ When an “If” flow is detected, then the activity flow with a guard defined on it is the body of the “if.”
- ◆ For all multiple exit activity flows there should always be one without a guard to define the sequential flow.

Activity flows in flow charts are the same as the corresponding transitions in activity diagrams, with the following exceptions:

- ◆ Outgoing activity flows and action blocks cannot have triggers.
- ◆ Outgoing activity flows from actions and action blocks can only have guards.

Creating an Activity Flow

To draw an activity from one action to another, follow these steps:

1. Click the **Activity Flow** button  on the **Drawing** toolbar.
2. Click the edge of the source action.
3. Drag the cursor to the edge of the target action and release to anchor the activity flow.
4. If you want, enter a guard for the activity flow.


Completion Action Flows

An action flow to a termination state is called a *completion action flow*. Termination states cannot have outgoing action flows. A completion action flow can only have a guard condition.

Default Flow

One of the action elements must be the default action flow. The flow chart flow originates from the element pointed to by the default flow. For flow charts, this default flow is the initial flow, and for code purposes, indicates the start of code.

To draw a default flow, follow these steps:


1. Click the **Default Flow** button  on the **Drawing** toolbar.
2. Click in the flow chart outside the default action.

3. Drag the cursor to the edge of the default action of the activity and release the mouse button.

Loop Activity Flows

Loop activity flows represent looping behavior in a program. Loop activity flows are often used on action blocks to indicate that the block should loop until some exit condition becomes true. A loop activity flow with a guard is in effect a Do-While statement.

To draw a loop activity flow, follow these steps:

1. Click the **Loop Activity Flow** button  on the **Drawing** toolbar.
2. Click the edge of an action.
3. Label the loop activity flow and press **Ctrl+Enter**.

See [Termination States](#) for an example of an action block with a loop activity flow.

Modifying Action Flows

As with all other elements, you can modify the features of an action flow using the Features dialog box. See [Modifying the Features of a Transition](#) for more information.

Connectors

Flow charts can have the following connectors:


- ◆ Junction
- ◆ Condition

The following sections describe these connectors in detail.

Junction Connectors

A junction connector combines different flows to a common target.

To draw a junction connector, follow these steps:


1. Click the **Junction Connector** button  on the **Drawing** toolbar.
2. Click in the flow chart to place the junction where you want it.
3. Draw flows going into, and one flow going out of the junction.
4. Label the flows if you want.

See [Statecharts](#) for more information on junction connectors.

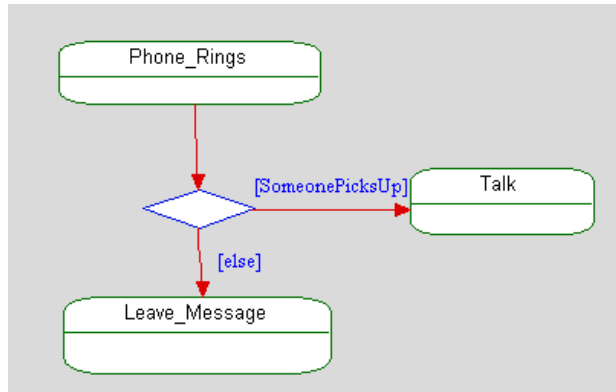
Condition Connectors

Condition connectors show branching conditions. A condition connector can have only one incoming activity and two or more outgoing activity flows. The outgoing action flows are labeled with a distinct guard condition. A predefined guard, denoted `[else]`, can be used for no more than one outgoing flow.

To draw a condition connector, follow these steps:

1. Click the **Condition Connector** button  on the **Drawing** toolbar.
2. Click, or click-and-drag, in the flow chart to position the condition connector where you want it.
3. Draw at least two actions that will become targets of the outgoing action flows.
4. Draw an incoming action flow from the source action to the condition connector.
5. Draw and label the outgoing action flows from the condition connector to the target actions.

Condition connectors look like diamonds, as shown in the following figure.



This flow chart shows the following behavior: When the phone rings, if someone picks up on the other end, you can talk; otherwise, you must leave a message. The condition connector represents the decision point. In other words, after the `PhoneRings()` operation, if `SomeonePicksUp` resolves to `True`, the `Talk()` operation is called. Otherwise, the `LeaveMessage()` operation is called.

Use the **Display Options** dialog box for the condition connector to determine whether to display its name, label, or nothing.

Code Generation

You specify whether to generate code for a flow chart by setting the `C.CG::Operation::ImplementFlowchart` property to `Checked`. This property is set to `Checked` by default. See the definition provided for the property on the applicable **Properties** tab of the Features dialog box.

Limitations and Specified Behavior

Note the following restrictions and behavior:

- ◆ You cannot animate or reverse engineer flow charts.
- ◆ Code generation from flow charts is not supported in Rhapsody in Java and Rhapsody in Ada.
- ◆ Flow chart code generation will never write the same action twice.
- ◆ This feature supports only a subclass of diagrams that do not contain events (including timeout events) or triggered operations.
- ◆ Rhapsody makes the following checks:

- If a function already has a body.
 - On guards that will be ignored because they are not part of an If/Then/Else or Loop.
 - That the flow chart and all its blocks have one and only one reachable termination state. If there are no reachable states or more than one, a message will display.
 - That there are no elements with more than one flow between them in the same direction. If there are more than one, a message will display.
 - That all the elements in the flow chart are supported. If unsupported elements are found, a message will display.
- ◆ If code generated will contain GoTos, Rhapsody will display a warning message with an indication as to which flows are causing the warning. Note the following:
 - Flow charts will normally generate structured code using If, If/Then/Else, Do, and While blocks. Rhapsody in C provides you with a Flowchart model located in the `<Rhapsody installation>\Samples\CSamples\Flowchart` folder. The Flowchart model contains a number of sample flow charts patterns that show you which ones are recognized in order to generate structured code. For example, the Flowchart model includes flow charts that show the DoWhileLoop, IfThenElse, and the WhileLoop.
 - If the code is not structured, then the flow charts will generate GoTo code. To avoid GoTo code, use the sample patterns for structured blocks as shown in the flow charts that are illustrated and documented in the Flowchart model provided with Rhapsody in C, which is located in the path noted above.

Sequence Diagrams

Sequence diagrams (SDs) describe message exchanges within your project. You can place messages in a sequence diagram as part of developing the software system. You can also run an animated sequence diagram to watch messages as they occur in an executing program.

Sequence diagrams show scenarios of message exchanges between roles played by objects. This functionality can be used in numerous ways, including analysis and design scenarios, execution traces, expected behavior in test cases, and so on.

Sequence diagrams help you understand the interactions and relationships between objects by displaying the messages that they send to each other over time. In addition, they are the key tool for viewing animated execution. When you run an animated program, its system dynamics are shown as interactions between objects and the relative timing of events.

Sequence diagrams are the most common type of interaction diagrams.

Note

Rhapsody message diagrams are based on sequence diagrams. Message diagrams, available in the FunctionalC profile, show how the files functionality may interact through messaging (through synchronous function calls or asynchronous communication). Message diagrams can be used at different levels of abstraction. At higher levels of abstractions, message diagrams show the interactions between actors, use cases, and objects. At lower levels of abstraction and for implementation, message diagrams show the communication between classes and objects.

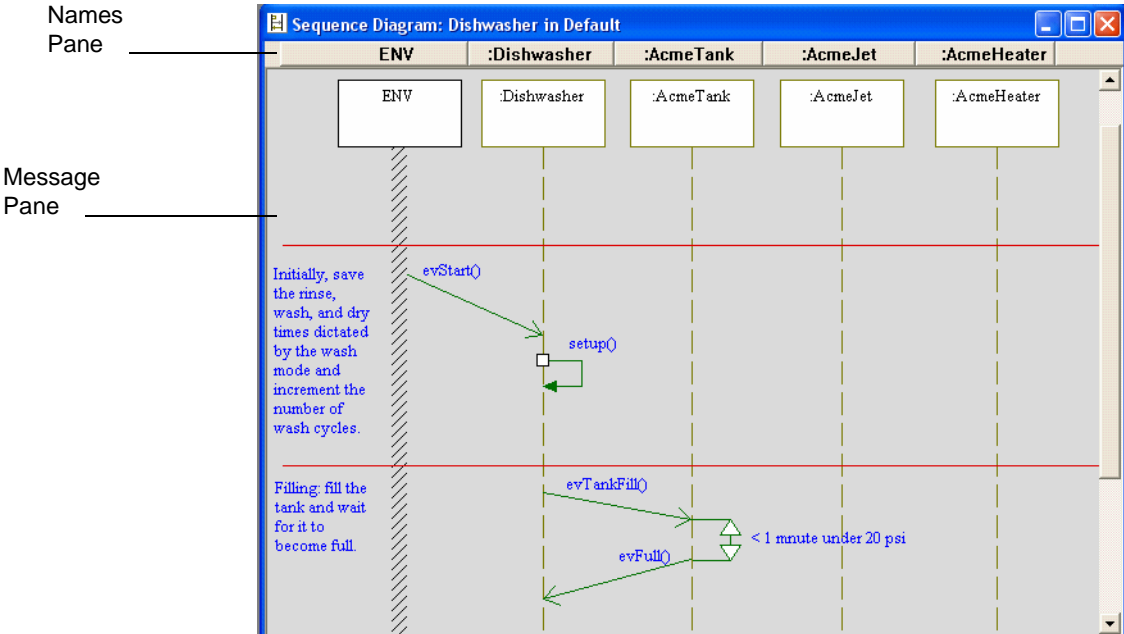
Message diagrams have an executable aspect and are a key animation tool. When you animate a model, Rhapsody dynamically builds message diagrams that record the object-to-object messaging.

For more information about the FunctionalC profile, see [Profiles](#).

Sequence Diagram Layout

A sequence diagram has two sections:

- ◆ **Names pane**—The top portion of the diagram. The name pane is a control to identify instance lines when the role names are not visible.
- ◆ **Message pane**—Shows the messages passed between instance lines.



Names Pane

The names pane contains the name of each instance line, or *classifier role*. In a sequence diagram, a *classifier role* represents an instance of a classifier. It describes a specific role played by the classifier instance to perform a particular task. A classifier role is shown as an instance line with a text header (name) with a box drawn around it. A classifier role can realize a classifier (class or actor) of the static model object.

Changing Names

Names that are too long to fit in the pane continue past the divider, running down behind the lower pane. To change the size of the names pane, click the dividing line and drag it up or down.

You can change the font and edit the names in the names pane using the pop-up menu for text items.

There are three ways to describe the name:

```
Classifier Role Name: Classifier Name
                    : Classifier Name
Classifier Role Name
```

In the first two cases, if the classifier name does not exist in the metamodel, Rhapsody asks if you want to add a new classifier to the project. The third case tells Rhapsody that you want to use an `<Unspecified>` classifier role, which means that the classifier role is not a realization of an existing classifier or actor.

Renaming Classifier Roles

If you change the name of a classifier role to a role name that already exists in the model, the classifier role is automatically realized to that classifier. For example, if you change the role name of classifier B to “Alarm” and there is a class `Alarm` in the model, this role becomes a realization of class `Alarm` and its name changes to `B : Alarm`.

If you change the name to a class that does not yet exist in the model, Rhapsody asks if you want to create that class. For example, if you type `x : X`, Rhapsody asks if you want to create the class `X`.

Message Pane

The message pane contains the elements that make up the interaction. In the object pane, system borders and instances are displayed as instance line, which are vertical lines with a box containing the role name at the top. Messages, such as events, operations, and timeouts are generally shown as horizontal and slanted arrows.

The messages appear in sequence as time advances down the diagram. The vertical distance between points in time indicates only the sequence in time and not any time scale.

Analysis Versus Design Mode

Three properties (under `SequenceDiagram::General`) to support the SD operation modes:

- ◆ `ClassCentricMode`—Specifies whether classes are realized when you draw instance lines. The possible values are as follows:
 - `Checked`—Instance names of the form `<xxx>` are treated as class names, not instance names. For example, if you create a new instance line named `c`, Rhapsody creates a class named `c` and displays it in the sequence diagram as `:c`.
 - `Cleared`—When you create an instance line, it is named `role_n` by default, which represents an anonymous instance. This is the default value.
- ◆ `RealizeMessages`—Specifies whether messages are realized when you create them. The possible values are as follows:
 - `Checked`—In Design mode, when you type in a message name, Rhapsody asks if you want to realize the message. If you answer no, the message is unspecified. For example, you could use an unrealized message to describe a message that is part of the framework (such as `takeEvent()`), without actually adding it to the model. (In analysis mode, the confirmation is controlled by the property `SequenceDiagram::General::ConfirmCreation`.)
 - `Cleared`—You can draw message lines freely, without messages from Rhapsody about realization. This is the default value.
- ◆ `CleanupRealized`—Specifies whether to delete messages in the sequence diagram if the corresponding operation is deleted. The possible values are as follows:
 - `Checked`—Delete the messages when the operation is deleted.
 - `Cleared`—Do not delete the messages when the operation is deleted. This is the default value.

For sequence diagrams produced in Rhapsody 4.0 or earlier, all three of these properties are `Cleared`.

Showing Unrealized Messages

To show a message that has not been realized, select **Edit > Select > Select Un-Realized**. The unrealized message is selected in the sequence diagram.

Realizing a Selected Element

To realize a selected element, follow these steps:

1. Select the element in the sequence diagram.
2. Select **Edit > Auto Realize**.











When you realize a message, Rhapsody creates a new message in just the manner as if you selected `<New>` in the **Realization** field in the Features dialog box. If you realize a classifier role, Rhapsody creates a class with the same name as designated in the role name, with a leading colon. For example, `Dishwasher` becomes `:Dishwasher`.










Creating Sequence Diagram Elements

The following sections describe how to use the sequence diagram tools to draw the parts of a sequence diagram. See [Graphic Editors](#) for basic information on sequence diagrams, including how to create, open, and delete them.

Sequence Diagram Drawing Icons

The **Drawing** toolbar for a sequence diagram includes the following tools.

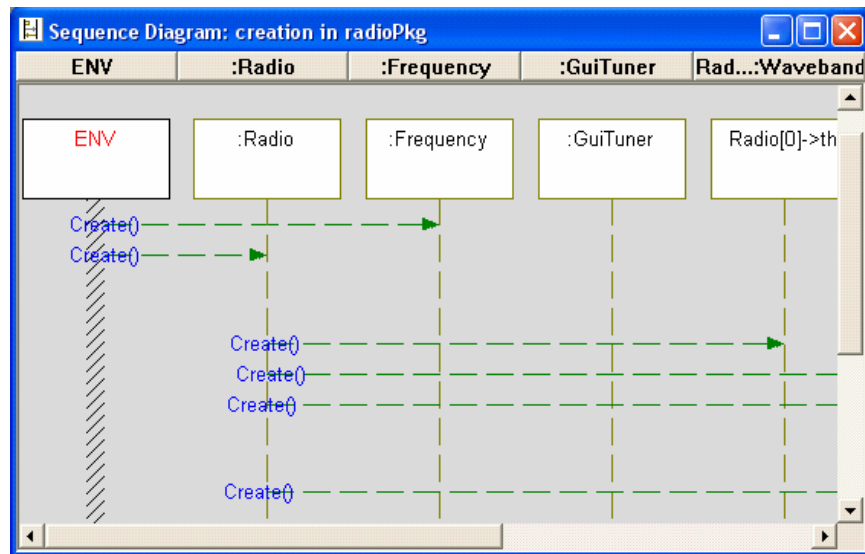
Drawing Icons	Definitions
	Instance line icon shows how an actors participates in the scenario. See Creating an Instance Line for more information.
	System border icon represents the environment. Events or operations that do not come from instance lines shown in the chart are drawn coming from the system border. See Creating a System Border for more information.
	Message icon represents an interaction between parts, or between a part and the environment. A message can be an event, a triggered operation, or a primitive operation. See Creating a Message for more information.
	Reply message icon represents the response from a message sent to a part or the environment. See Creating a Reply Message for more information.
	Create arrow icon marks when an instance is created. It can originate from the system border or another instance. It is a horizontal, dotted line from the creator object to the new object. An object can have only one "create arrow." You can label the create arrow with construction parameters. See Drawing a Create Arrow for more information.
	Destroy arrow icon marks the destruction of an object. It is a dotted line from the destroying object to the object being destroyed. It can be either a horizontal line or a message-to-self. See Creating a Destroy Arrow for more information.
	Timeout icon indicates when an event stops and may include a parameter indicating the length of the time the event is stopped. This is a type of message that is always communicating with itself. See Creating a Timeout for more information.
	Cancelled timeout icon indicates the condition when an event that has timed out should restart. See Creating a Cancelled Timeout for more information.
	Timeout interval icon can be used to create a waiting state with an event stopping for this predefined interval and then automatically restarting. See Specifying a Time Interval for more information.
	DataFlow indicates the flow of data between two objects. You may use the features dialog box to select the flowport to which it is connected on the receiving object and change the value being sent. This connection is also automatically added to the sequence diagram during animation. See Creating a Dataflow for more information.

Drawing Icons	Definitions
	Partition line icon separates phases of a scenario represented in the sequence diagram. See Creating a Partition Line for more information.
	Condition mark icon indicates that the object is in a certain condition or state at this point in the sequence. See Creating a Condition Mark for more information.
	Execution Occurrence icon shows the beginning and end of the unit of behavior (the actions performed by an operation or event) that is triggered by a specific message. See Creating Execution Occurrences for more information.
	Interaction Occurrence icon refers to another sequence from within a sequence diagram. This allows complex scenarios to be divided into smaller, reusable scenarios. See Creating an Interaction Occurrence for more information.
	Interaction Operator icon groups related elements and define specific conditions under which each group of elements occurs. See Creating Interaction Operators for more information.
	Interaction Operand Separator icon create two subgroups of elements within the sequence diagram. This may be used to create two paths that are supposed to be carried out in parallel or to define two possible paths and a condition that determines which is to be followed. See Adding an Interaction Operand Separator to an Interaction Operator for more information.
	Lost Message indicates a message sent from an instance that never arrives to its destination. This item is not supported in code generation or animation.
	Found Message indicates a message that arrives at an instance, but its target is unknown. This item is not supported in code generation or animation.
	Destruction Event indicates the destruction of the instance, such as the destroy arrow, has happened. This item is not supported in code generation or animation.


Creating a System Border

A *system border* represents the environment. Events or operations that do not come from instance lines shown in the chart are drawn coming from the system border.

A system border is a column of diagonal lines, labeled ENV. You can place a system border anywhere an instance line can be placed, but the typical locations are the far left and right edges of the chart.




To create a system border, follow these steps:

1. Click the **System border** icon .
2. Move the cursor into the drawing area, then click to place the system border. At this point, the system border is anchored in place.
3. Because the system border represents the environment, it is named ENV by default. If desired, rename the system border.

Creating an Instance Line

An instance line (or *classifier role*) is a vertical timeline labeled with the name of an instance. It represents a typical instance or class in the scenario being described. It can receive messages from or send messages to other instance lines.

In addition to creating, deleting, and modifying the name of an instance line, you can realize the instance line to a class or actor in the static model.

1. Click the **Instance line** icon .
2. Move the cursor over the diagram.
3. Move the line to a suitable location, then click to dock the line into place.
4. Type the name of a class or an instance to replace the default name.
5. If you specified design mode, Rhapsody names the instance line `:class_n` by default. If the specified class does not exist, Rhapsody asks if you want to create it. Click **OK** to create the class.
6. You can continue creating instance lines, or return to select mode by clicking the **Select** tool.

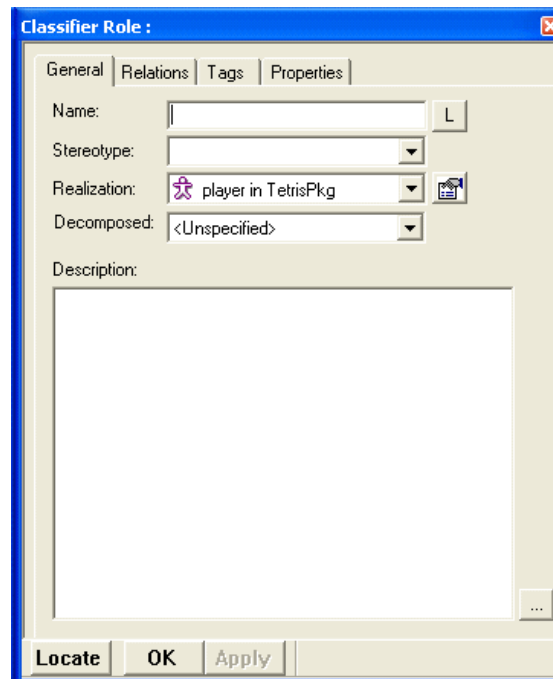
If you prefer, you can place several lines and rename them later. Rhapsody gives them default names until you rename them. See [Message Line Pop-Up Menu](#) for information on renaming instance lines. Note that the sequence diagram automatically expands the diagram as necessary as you add more instance lines.

Note

To shift one or more messages to different instance lines, select the relevant messages and press Ctrl + right arrow button, or Ctrl + left arrow button. The messages “jump” to the new source and destination instance lines. This replaces the cut and paste (or drag) functionality of messages between instance lines in the same diagram.

Modifying the Features of a Classifier Role

The Features dialog box enables you to change the features of a classifier role, including its realization. The following figure shows the Features dialog box for a classifier role.



A classifier role has the following features:

- ◆ **Name**—Specifies the name of the classifier role.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *API Development Guide* for more information about these components.

- ◆ **Realization**—Specifies the class being realized by the instance line.
- ◆ **Decomposed**—Specifies the referenced sequence diagram for the instance line, if you are using part decomposition. See [Part Decomposition](#) for more information.
- ◆ **Description**—Describes the classifier role. This field can include a hyperlink. See [Hyperlinks](#) for more information.

Names of Classifier Roles

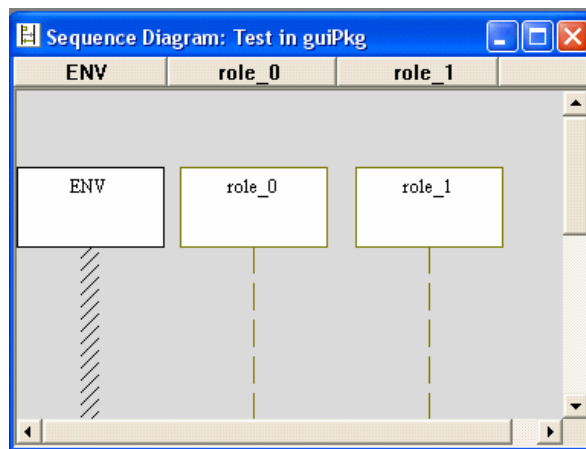
A classifier role (instance line) with a class name is a view of the class in the model. An instance line with an instance name (of the form `instance:class`) is also a view of the class in the model.

Instance lines reference classes in the model. If you rename an instance line to another class name that exists in the model, the line acts as a view to the other class in the model. If the class does not exist, Rhapsody will create it.

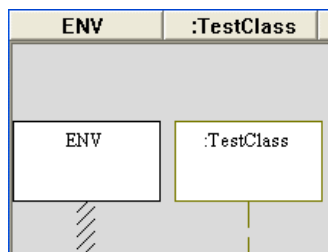
Note

The names for instance lines are resizable text frames. Text wraps to match the contour of the bounding box.

If you specified analysis mode, Rhapsody names the instance line `role_n` by default and does not prompt you for class information. The following figure shows a new instance line in an analysis SD.



If you specified design mode, Rhapsody names the instance line `:class_n` by default. If the specified class does not exist, Rhapsody asks if you want to create it. The following figure shows a new instance line in a design SD.



Note that Classifier role names are animated when their expression can be mapped to an existing object.

Examples:

We have a class `A` and an object `a:A` who is the only object of `A`. All the following names will be mapped to it:

- ◆ `:A`
- ◆ `a:A`
- ◆ `A[0]:A`
- ◆ `A[#0]:A`

Suppose that instead of `a:A`, we have a single instance of `A` as a part `itsA` of another object `b:B`.

The instance line can be named as:

- ◆ `:A`
- ◆ `b->itsA:A`
- ◆ `B[0]->itsA:A`
- ◆ `A[#0]:A`
- ◆ `B[#0]->itsA->a:A`

The same instance mappings apply as described in [Instance Names](#).

Instance Line Pop-Up Menu


- ◆ **Class**—Displays a submenu of commands for classes.
- ◆ **Open Reference Sequence Diagram**—Opens the reference sequence diagram associated with the classifier role. This option is grayed out if a reference sequence diagram does not exist for this classifier role. See [Creating an Interaction Occurrence](#) for more information.
- ◆ **Display Options**—Specifies how the element should be displayed.

Creating a Message

A *message* represents an interaction between objects, or between an object and the environment. A message can be an event, a triggered operation, or a primitive operation. In the metamodel, a message defines a specific kind of communication. The communication could be raising a signal, invoking an operation, or creating or destroying an instance.

The recipient of a message is either a class or a reactive class. Reactive classes have statecharts, whereas nonreactive classes do not. Reactive classes can receive events, triggered operations, and primitive operations. Non-reactive classes can receive only messages that are calls to primitive operations. Events are usually shown with slanted arrows to imply that they are *asynchronous* (delivery takes time). Triggered operations are shown with straight arrows to imply that they are *synchronous* (happen immediately).

To create a message, follow these steps:

1. Click the **Message** (event) icon .
2. Move the cursor over the instance lines.
Note: A plus sign appears on each instance line as you move the cursor from one to the next. This symbol indicates a potential origination point for the intended message.
3. Left-click to anchor the start of the message at the desired location, then move the cursor. A dashed line appears as a guide for the message.
4. Move the cursor lower the start of the message to create a downward-slanted diagonal line. Click to anchor the end of the message on the target object once the diagonal line has extended itself to that point.
5. If you specified design mode and the specified message is not realized in the model, Rhapsody asks if you want to realize it. Click **OK** to realize the new message.

Rhapsody creates a message with the default name `message_n()`, where *n* is an incremental integer starting with 0. Sequence diagrams automatically expand in length to accommodate new messages.

To specify the type of operation and its access level, select the **Features** option from the pop-up menu. By default, Rhapsody creates a primitive operation with public access. See [Using the Features Dialog Box](#) for more information.

Message Names

The naming convention for messages is as follows:

```
message (arguments)
```

The names for messages can be event or operation names. They can include actual parameters in parentheses, which would be expressions in the scope of the sender/caller. Message names are resizable, movable text frames. Text wraps to match the contour of the bounding box.

Note: In Rhapsody versions 4.0 and earlier, if you changed the message name, Rhapsody asked if you wanted to create a new operation with the given name.

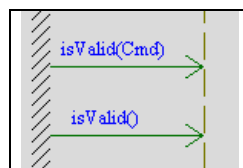
If you modify the name of an operation that exists in the model, the message is automatically realized to that operation, and Rhapsody changes its type to the type of that operation (constructor, event, and so on).

If you change the message name to a message that does not belong to the classifier, it becomes unspecified (in analysis mode).

Displaying Message Arguments

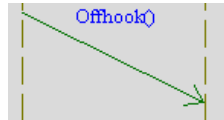
The `SequenceDiagram::General::ShowArguments` property displays or hides message arguments. By default, the `ShowArguments` property is enabled. It applies to all messages in a sequence diagram, not to individual messages.

Note that any changes you make to property settings apply only to new elements you draw after making the change, not to existing elements. For example, to have arguments displayed on one message but not on another message of the same type, set the `ShowArguments` property before drawing one message, then reset it before drawing the next message, as shown in the following figure.



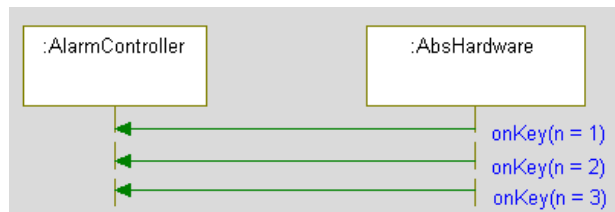
Slanted Messages

A message drawn on a slant is interpreted as an event if the target is a reactive class, and as a primitive operation if the target is a nonreactive class. A slanted message emphasizes that time passes between the sending and receiving of the message. Slanted messages can cross each other.



Horizontal Messages

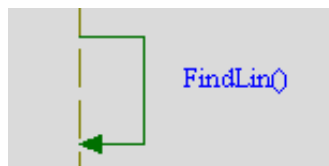
A message drawn as a horizontal line is interpreted as a triggered operation, if the target is a reactive class, and a primitive operation if the target is a nonreactive class. The horizontal line indicates that operations are synchronous.



Message-to-Self

A message-to-self is interpreted as an event if the instance is a reactive class. If the instance is a nonreactive class, a message-to-self is interpreted as a primitive operation.

A message-to-self is shown by an arrow that bends back to the sending instance. The arrow can be on either side of the instance line. If the message-to-self is a primitive operation, the arrow folds back immediately. If the message-to-self is an event, the arrow might fold back sometime later.

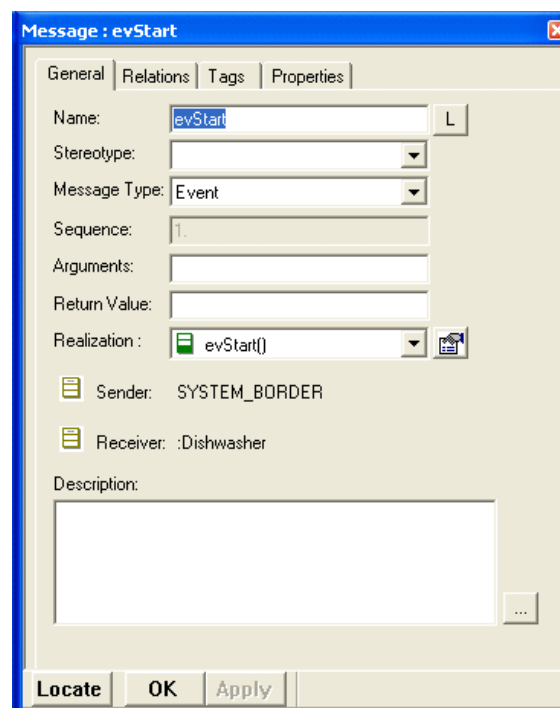


Message Line Pop-Up Menu

- ◆ **Select Message**—Enables you to select a message. See [Selecting a Message or Trigger](#) for more information.
- ◆ **Add Execution Occurrences**—Enables you to add execution occurrences.
- ◆ **Display Options**—Specifies how the element should be displayed.

Modifying the Features of a Message

The Features dialog box enables you to change the features of a message, including its type or return value. The following figure shows the Features dialog box for a message.



A message has the following features:

- ◆ **Name**—Specifies the name of the message. The default name is `Message_n`, where *n* is an incremental integer starting with 0.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the message, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *API Development Guide* for more information about these components.

- ◆ **Message Type**—Specifies the type of message—event, primitive operation, and so on.

Note that you cannot change the message type once the message has been realized.

- ◆ **Sequence**—Specifies the order number of a message. Make sure any numbering sequence you use, such as 1a., 1b., 1.2.1., 1.2.2., and so on ends with a period (.). Rhapsody needs the ending period to continue the numbering sequence automatically.

Note: In collaboration diagrams, you can modify both the format of the numbering and the starting point for the numbering. In sequence diagrams, this field is read-only—you cannot modify the numbering format or the starting point of the numbering.

- ◆ **Arguments**—Specifies the arguments for the message.
- ◆ **Return Value**—Specifies the return value of the message.
- ◆ **Realization**—Specifies the class that will be realized. This drop-down list contains the following options:
 - Existing classes in project.
 - <Unspecified>—If this is an analysis SD, this is the realization setting for the instance line.
 - <New>—Invokes the Class dialog box so you can set up the new class. See [Defining the Features of a Class](#) for more information.

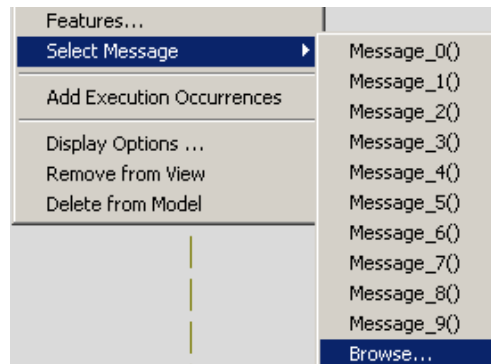
Click the **Features** button to open the Features dialog box for the class specified in the drop-down list.

In addition, this section lists the sender and receiver objects for the message.

- ◆ **Description**—Describes the message. This field can include a hyperlink. See [Hyperlinks](#) for more information.

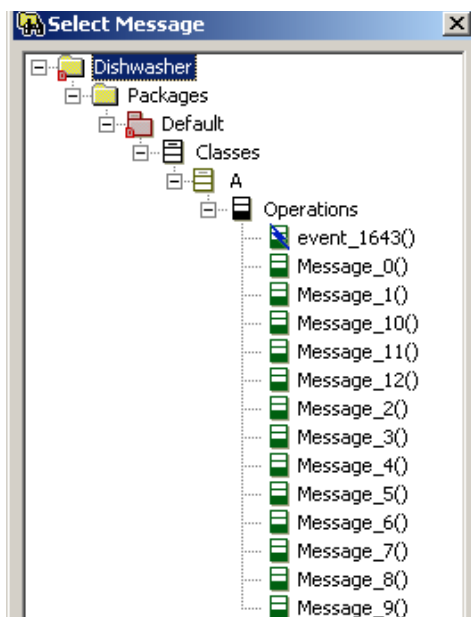
Selecting a Message or Trigger

When you choose **Select Message** or **Select Trigger**, Rhapsody displays a list of the messages or triggers provided by the target object, as shown in the following figure for messages. Notice that if there are more messages (or triggers) that can appear on the pop-up menu, a **Browse** command appears. For triggers, see also [Selecting a Trigger Transition](#).



For messages, if the target is an instance of a derived class, the list also includes those message inherited from its superclass. The target class *provides* the message, whereas the source class *requires* it. You can also think of provided messages as those to which the class responds. At the implementation level, an event is generated into statechart code that checks for the existence of that event.

The following figure shows the Select Message dialog box that opens when you choose **Select Message > Browse** for messages. It shows you all the messages/events that are available.



Cutting, Copying, and Pasting Messages

You can cut, copy, and paste messages in sequence diagrams using the standard **Ctrl+X**, **Ctrl+C**, and **Ctrl+V** keyboard shortcuts, respectively.

Moving Messages

To move a message line, follow these steps:

1. Select the message you want to move.
2. Press **Ctrl+Left arrow** to move the message to the left, or **Ctrl+Right arrow** to move it to the right.

Message Types

If you open the Features dialog box for the message, you can select the message type: primitive operation, or triggered operation, or event. Once defined, these messages are displayed in the browser, denoted by unique icons. In the browser, you can access modify a message by right-clicking on it and selecting the appropriate option from the pop-up menu.

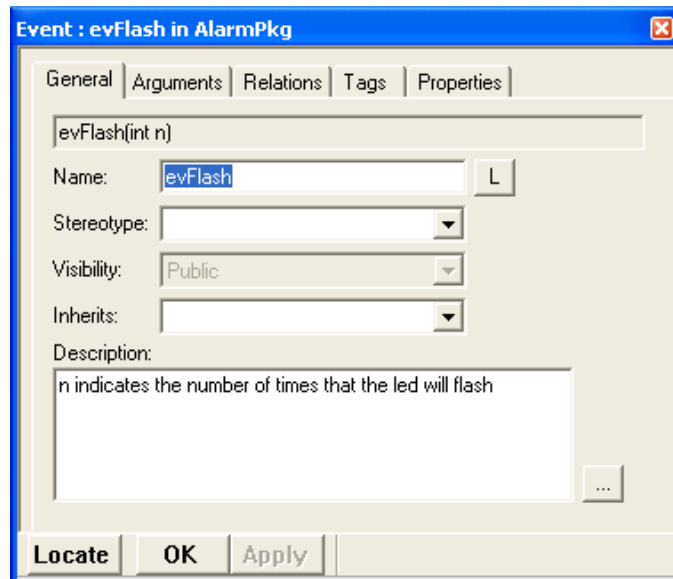
Note

Once a message has been realized, you cannot change its type.

Events

An *event* is an instantaneous occurrence that can trigger a state transition in the class that receives it. Because events are objects, they can carry information about the occurrence, such as the time at which it happened. The browser icon for an event is a lightning bolt.

The following figure shows the Features dialog box for an event.



Note

If an event arguments is of type *& (pointer reference), Rhapsody will not generate code for it.

Triggered Operations

A *triggered operation* can trigger a state transition in a statechart, just like an event. The body of the triggered operation is executed as a result of the transition being taken. The browser icon for a triggered operation is a green operation box overlaid with a lightning bolt.

Note

If an argument of a triggered operation is of type *& (pointer reference), Rhapsody will not generate code for that argument.

Operations

By default, operations are primitive operations. *Primitive operations* are those whose bodies you define yourself instead of letting Rhapsody generate them for you from a statechart.

The browser icon for operations is a three-compartment class box with the bottom compartment filled in:



The icon for the `Operations` category is black.



The icon for an individual operation is green.



The icon for a protected operation is overlaid with a key.



The icon for a private operation is overlaid with a padlock.

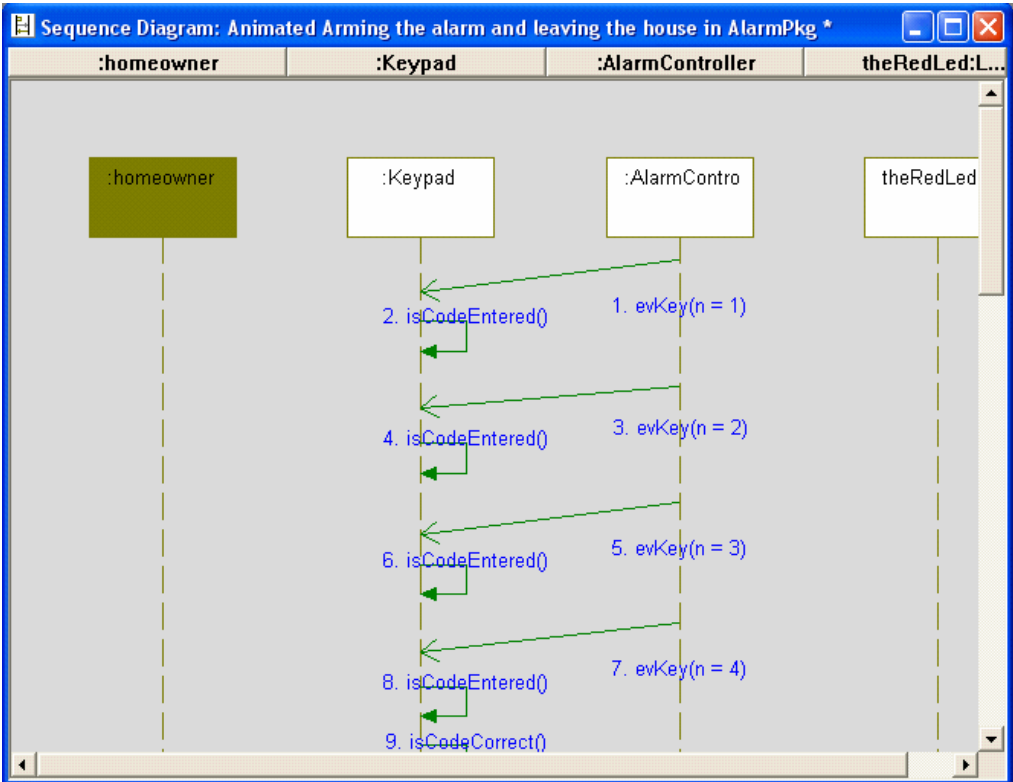
Deleting Operations

You delete an operation like any other model element in Rhapsody. However, if you delete an operation that is realized by a message in a sequence diagram, the message becomes unspecified (in both design and analysis modes).

If you delete an operation, class, or event, the corresponding message lines are not deleted automatically from the diagram. To have Rhapsody perform this cleanup automatically, set the `SequenceDiagram::General::CleanupRealized` property to `Checked`.

Viewing Sequence Numbers

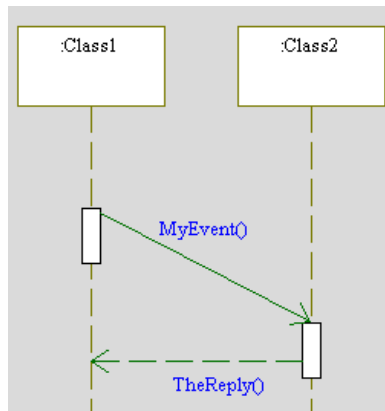
If desired, you can display the sequence number with the message name. The following figure shows a sequence diagram that includes the sequence number of each operation.



To display the sequence numbers, set the property `SequenceDiagram::General::ShowSequenceNumber` to `Checked`. By default, this property is set to `Cleared` (sequence numbers are not displayed).

Creating a Reply Message

A *reply message* can realize an operation or event reception at the *source* (unlike other messages, which realize operations at the *target*). By default, reply messages are drawn as unnamed, dashed lines



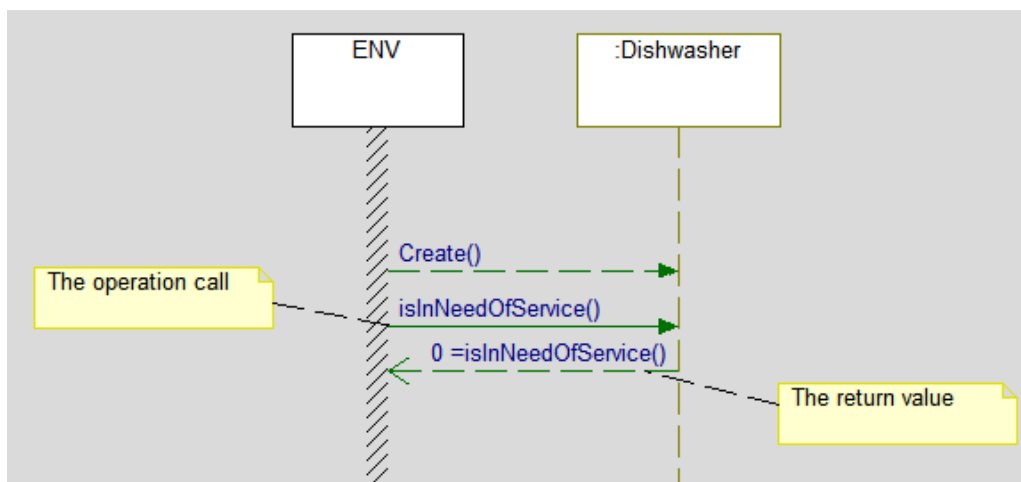
Animation of an Operation's Return Value

Note

This feature is not supported in Rhapsody in J.

To show the return value of a function as a reply message in an animated sequence diagram, you can use one of a number of predefined macros within the code of your function. This means that the return value for your function visually displays as a reply message on your sequence diagram. The same is true for a trace of a function.

The following figure shows an animated sequence diagram that draws a return value.



You can use any of the following macros depending on your situation:

- ◆ `OM_RETURN`. Use this macro in the operation's body instead of the regular "return" statement:

Examples:

```
- Int Foo(int& x) {x = 5; OM_RETURN(10);}  
- A* Foo() {OM_RETURN(newA());}
```

- ◆ **CALL.** Use this macro if you cannot change the operation code or if you want to animate return values only on specific calls to the operation. Note that this macro can handle only primitive types.

Example:

```
Int foo(int n) {return n*5;}
void callingFunction()
{
    int v;
    CALL (v, foo(10));
    // after the call v equals 50
}
```

- ◆ **CALL_INST.** Same as **CALL**, but use **CALL_INST** when the return value is of a complex type, such as a class or a union.

Example:

```
A* foo() {return new A();}
void callingFunction()
{
    A *a;
    CALL_INST(a, foo());
    // after the call a equals new A[0]
}
```

- ◆ **CALL_SER.** Use this macro when the type has a user-defined serialization function.

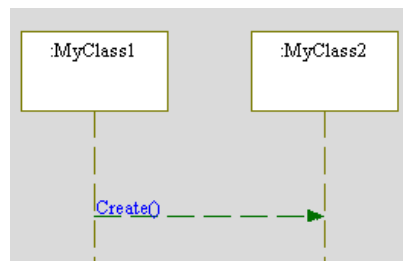
Examples:

```
- char* serializeMe(A*) {...}
- A* foo() {return new A();}
void callingFunction()
{
    A *a;
    CALL_SER(a, foo(), serializeME);}
    // after the call v equals <string that serializeMe returns>
}
```

Note that even if you choose not to embed these macros in your application, you can still see animated return values by explicitly calling an operation through the Operations dialog box. To call an operation, click the **Call operations** tool in the **Animation** toolbar.

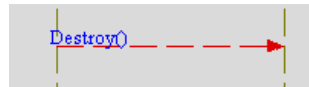
Drawing a Create Arrow

A *create arrow* marks when an instance is created. It can originate from the system border or another instance. It is a horizontal, dotted line from the creator object to the new object. Every object can have at most one create arrow. You can label the create arrow with construction parameters.



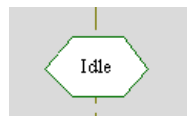
Creating a Destroy Arrow

A *destroy arrow* marks the destruction of an object. It is a dotted line from the destroying object to the object being destroyed. The destroy arrow is red and is not labeled. It can be either a horizontal line or a message-to-self.



Creating a Condition Mark

A *condition mark* (or state mark) is displayed on an instance line. A condition mark shows that the object has reached a certain condition or is in a certain state. Often, the name corresponds to a state name in the object's statechart.



1. Select the condition mark using the **Select** arrow.
2. Click-and-drag a selection handle to resize the condition mark.

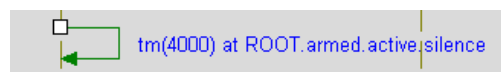
The condition mark remains centered over the instance line. If necessary, other elements on the sequence diagram are adjusted to accommodate the new size.

Creating a Timeout

The notation for timeouts is similar to the notation for events sent by an object to itself. There are two differences:

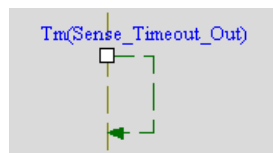
- ◆ A timeout starts with a small box.
- ◆ The name is a $\text{tm}(x)$.

The label on a timeout arrow is a parameter specifying the length of the timeout. Timeouts are always messages-to-self.



Creating a Cancelled Timeout

When designing a software system, you can establish waiting states, during which your program waits for something to occur. If the event occurs, the timeout is canceled. The sequence diagram shows this with a canceled timeout symbol. If it does not happen, the timeout wakes up the instance and resumes with some sort of error recovery process. Canceled timeouts are always messages-to-self.

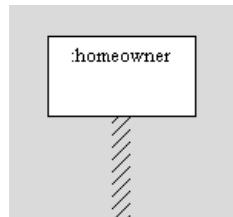


For example, on a telephone, a dial tone waits for you to dial. The telephone has a timeout set so if you do not dial, the dial tone changes to a repeating beep. If you do dial, the timeout is canceled.

A canceled timeout occurs automatically once the state on which the timeout is defined is exited. As a designer, you do not need to do anything to cancel a timeout. The framework has a call to cancel a timeout, but you do not need to use it because the code generator inserts it automatically.

Creating an Actor Line

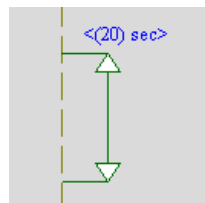
An *actor line* shows where actors affect the sequence diagram. To draw an actor line, drag-and-drop an actor from the browser to the sequence diagram. An actor is represented as an instance line with hatching.



When you want to realize a classifier, the list contains all the available classes and actors.

Specifying a Time Interval

A time interval is a vertical annotation that shows how much (real) time has passed between two points in the scenario. The name is free text; it is not constrained to be a number or unit of any kind. Time intervals can only be messages to self.

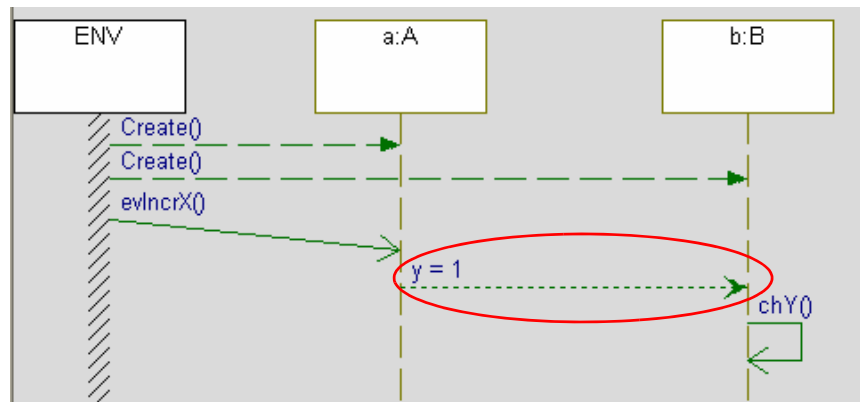


Creating a Dataflow

A dataflow in Rhapsody indicates the flow of data between two instances on a sequence diagram, as shown in the following figure. Rhapsody animation uses this notation to represent data flow between flow ports. For information on flow ports, see [Flow Ports](#).


Note

This feature is not supported in Rhapsody in J.



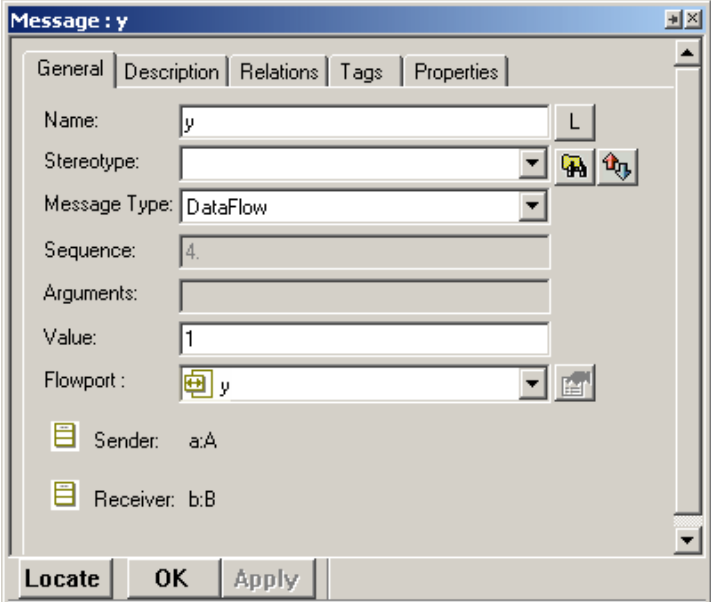
The dataflow will be realized to the flow port on the receiving instance. The name of the dataflow is the name of its realized flow port and the data that has been received. For example, for the dataflow `y = 1`, flow port `y` has received the data `1`.

Note that the dataflow arrow can be created through the following ways:

- ♦ Automatically during the animation of the sequence diagram
- ♦ Manually by drawing the dataflow on the sequence diagram (that is, click the DataFlow button  and then click the sender and receiver instances)

Sequence Diagrams

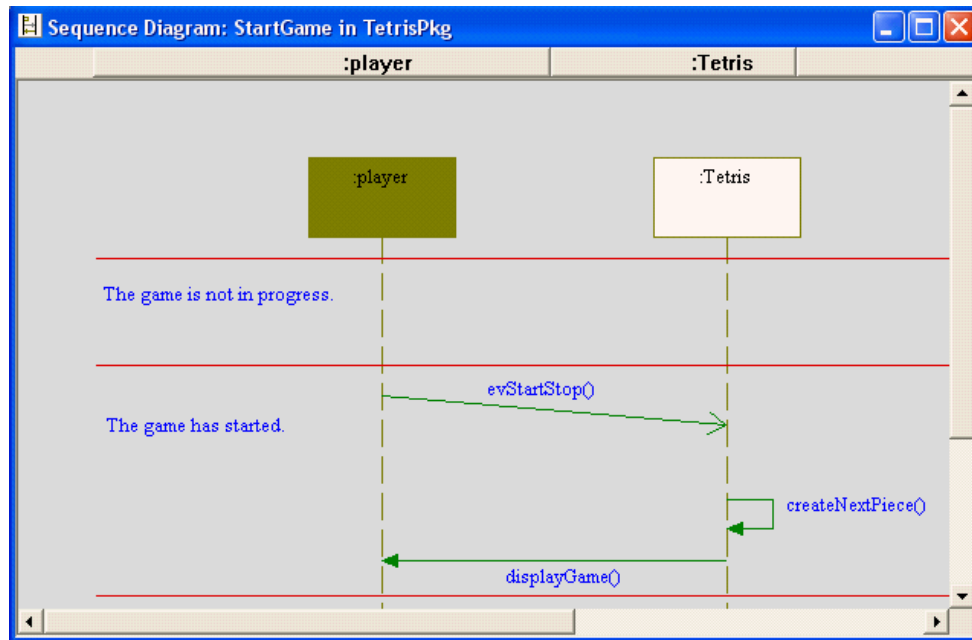
You can double-click the dataflow arrow to open its Features dialog box, as shown in the following figure, from which you can, for example, choose which flow port to connect to and change the value to send.



Creating a Partition Line

Partition lines separate phases of a scenario. They are red lines drawn across the chart and are usually used to keep parts of the scenario grouped together.

Each partition line includes a note positioned at its left end by default. You can move or resize the note, but a note is always attached to its partition line. If you move the line, the note follows. Notes contain the default text “note” until you enter text for them.



Creating an Interaction Occurrence

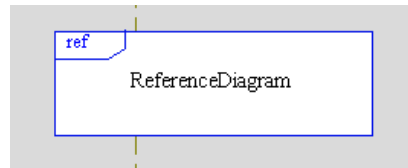
An *interaction occurrence* (or reference sequence diagram) enables you to refer to another sequence diagram from within a sequence diagram. This functionality enables you to break down complex scenarios into smaller scenarios that can be reused. Each scenario is an “interaction.”

To create an interaction occurrence, follow these steps:

1. Click the **Interaction Occurrence** tool.

Alternatively, you can use the **Add Interaction Occurrence** option in the pop-up menu.

2. Place the reference diagram on one or more instance lines to signify that those classes interact with the referenced sequence diagram. The interaction occurrence appears as a box with the “ref” label in the top corner, as shown in the following figure.



By default, when you first create the interaction occurrence (and have not yet specified which diagram it refers to), Rhapsody names it using the convention `interaction_n`, where *n* is greater than or equal to 0.

3. Right-click the interaction occurrence, then select **Features** from the pop-up menu.
4. Use the **Realization** drop-down list to specify the sequence diagram being referenced. When you select the referenced diagram, the name of the interaction occurrence is updated automatically to reflect the name of the referenced SD.
5. Click **OK** to apply your changes and close the dialog box.

You can move, rename, and delete reference sequence diagrams just like regular sequence diagrams. However, if you delete a sequence diagram that references an interaction occurrence, the interaction occurrence itself is not deleted, but becomes unassociated.

To change the default appearance of interaction occurrences, use the `SequenceDiagram::InteractionOccurrence` properties. Refer to the definition displayed in the **Properties** tab for this property.

Navigating to a Reference Sequence Diagram

To navigate to a reference sequence diagram from the current sequence diagram, right-click the interaction occurrence and select **Open Reference Sequence Diagram** from the pop-up menu. The referenced SD is displayed in the drawing area.

Interaction Occurrence Menu

- ◆ **Create Reference Sequence Diagram**—Enables you to specify the reference diagram for the interaction occurrence, if you have not yet specified one
- ◆ **Open Reference Sequence Diagram**—Opens the sequence diagram referred to by the interaction occurrence
- ◆ **Display Options**—Specifies whether labels are displayed

Part Decomposition

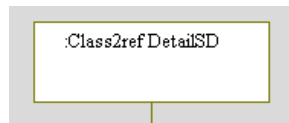
Instance line decomposition enables you to easily decompose an instance line on a sequence diagram into a series of parts. For example, if you have a composite class view in one sequence diagram and want to navigate to its parts, you can click the composite class and open a collaboration, which shows how its internal parts communicate for a particular scenario.

Part decomposition is a specialization of an interaction occurrence.

To create a part decomposition, follow these steps:

1. Invoke the Features dialog box for the instance line. The Classifier Role dialog box opens.
2. Specify the reference sequence diagram using the **Decomposed** drop-down list.
3. Click **OK** to apply your changes and close the dialog box.

In the sequence diagram, the name of the reference sequence diagram is added to the classifier role label (after the word “ref”), as shown in the following figure.



As with other interaction occurrences, navigate to the reference SD by right-clicking the instance line and selecting **Open Reference Sequence Diagram** from the pop-up menu.

Limitations

Note the following limitations for decomposition:

- ◆ UML gates are not supported: use instance lines instead.
- ◆ Animation is not supported.

Creating Interaction Operators

Interaction operators, which are included in UML 2.0, are used to group related elements in a sequence diagram. This includes the option of defining specific conditions under which each group of elements will occur.

Characteristics of Interaction Operators

Each interaction operator has a type, which determines its behavior, for example Opt, Par, or Loop.

In addition, interaction operators can include a guard to specify specific conditions under which the path will be taken.

Interaction operators can be nested where necessary.

Adding an Interaction Operator to a Diagram

To add an interaction operator to a diagram:

1. Click the Interaction Operator icon on the **Drawing** toolbar.
2. Click the canvas for a default-size interaction operator, or click and drag to draw an interaction operator of a specific size.

Setting the Type of an Interaction Operator

To set the type of an interaction operator, you can do any of the following:

- ◆ With the interaction operator selected in the diagram, click the type text in the top left corner and enter the appropriate type.
- ◆ With the interaction operator selected in the diagram, click the type text and then press **Ctrl+Space** and select a type from the list that is displayed.
- ◆ Open the Features dialog for the interaction operator, and select a type from the Type drop-down list.

Setting the Guard of an Interaction Operator

To set the guard for an interaction operator, do one of the following:

1. Click **[condition]** and enter the appropriate expression inside the brackets
2. Open the Features dialog for the interaction operator, and type in the expression under Constraints.

Note

Using **Display Options** from the context menu, the guard(s) for an interaction operator can be hidden or shown.

Adding an Interaction Operand Separator to an Interaction Operator

For certain types of interaction types, you may want to create two subgroups of elements, for example, if you have two paths that are supposed to be carried out in parallel, or you want to define two possible paths and a condition that determines which will be followed.

To create an interaction operand separator:

1. Select the Interaction Operand Separator icon on the **Drawing** toolbar.
2. Click inside the interaction operator where you would like the division to be.

If necessary, more than one separator can be added to a single interaction operator.

Note

Interaction operators (and interaction operand separators) only appear in the diagram itself. They do not appear as independent model elements in the browser, nor do they influence code generation. For this reason, when you display the context menu for an interaction operator, there is an option to Delete From View but no option for removing from the model.

Interaction Operator Types

Some of the more common types of interaction operators are described below.

- ◆ Alt—(Alternative) multiple fragments; only the one whose condition is true will execute
- ◆ Opt—(Optional) fragment executes only if the specified condition is true
- ◆ Par—(Parallel) each of the contained fragments is run in parallel
- ◆ Loop—the fragment may execute multiple times; guard indicates the basis for iteration

Creating Execution Occurrences

Execution occurrences show the beginning and end of the unit of behavior (the actions performed by an operation or event) that is triggered by a specific message.

Note

To animate a sequence diagram automatically, set the `SequenceDiagram::General::AutoLaunchAnimation` property to one of these options:

- ◆ **Always** launches the sequence diagram automatically.
- ◆ **If In Scope** launches animation only if the sequence diagram is in the active component scope.

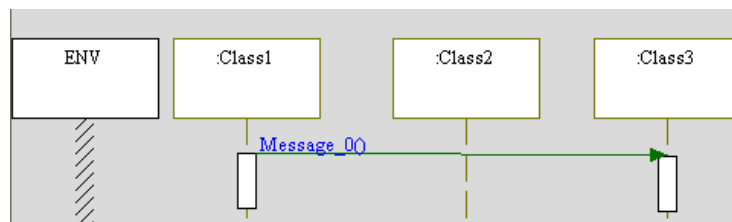
The **Never** option for this property prevents automatic animation and is the default.

There are two ways to draw interaction occurrences:

1. Select the message in the sequence diagram, right-click, and select **Add Execution Occurrences** from the pop-up menu.
2. Click the **Execution Occurrences** icon on the **Drawing** toolbar, then select the appropriate message in the sequence diagram.

You can have Rhapsody create execution occurrences automatically when you create messages by setting the property `SequenceDiagram::General::AutoCreateExecutionOccurrence` to `Checked`.

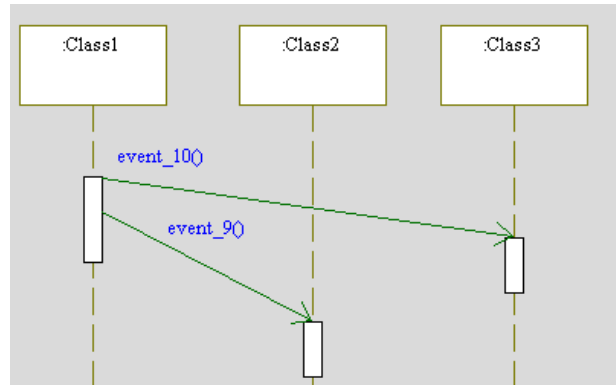
Execution occurrences are drawn at the beginning and end of the message, as shown in the following figure.



Note the following:

- ◆ If you move a message, its execution occurrences move with it.
- ◆ You can resize execution occurrences (lengthwise), but cannot move them.

- ◆ If you move a message with execution occurrences or resize them so they overlap other execution occurrences, they are “merged,” as shown in the following figure.



Deleting Execution Occurrences

You can delete execution occurrences in two ways:

- ◆ Select the execution occurrence and then click **Delete** (or use the **Delete from Model** option in the pop-up menu)
- ◆ Delete the start message (the message assigned to the execution occurrences). This automatically deletes the execution occurrences “owned” by that message.

Shifting Diagram Elements with the Mouse

You can use the following mouse actions to shift all or some of the elements in a sequence diagram.

To shift the entire diagram upward or downward:

1. Verify that an instance line is not currently selected.
2. Position the mouse in the area above the classifier role names.
3. Hold down the Shift key and drag the mouse up to shift the entire diagram upward, or drag down to shift the entire diagram downward.

To shift groups of elements upward or downward.

1. Verify that an instance line is not currently selected.
2. Position the mouse above the highest element that you want to shift.
3. Hold down the Shift key and drag the mouse up to shift the element and all the elements below it upward, or drag down to shift the element and all the elements below it downward

To shift groups of instance lines to the left or right:

1. Select any of the instance lines in the diagram.
2. Position the mouse to the left of the instance lines that you want to shift.
3. Hold down the Shift key and drag the mouse to the right to move all the elements *on the right side of the mouse cursor* to the right. Hold down the Shift key and drag the mouse to the left to move all the elements *on the right side of the mouse cursor* to the left.

Note: If you place the mouse *on an instance line* and drag while the Shift key is held down, Rhapsody will shift the instance lines on the right of that instance line but not that instance line itself. “On an instance line” includes the entire width below the box that contains the name of the classifier role.

Display Options

Most of the elements that can be added to a sequence diagram have options that you can set to affect how they are displayed in the diagram. While these options vary from element to element, the following options are common to the System Border, Instance Line, Message, and Reply Message elements:

- ◆ Select whether to display the name or the label of the element
- ◆ Show/hide any applied stereotypes

Sequence Diagrams in the Browser

The browser icon for sequence diagrams consists of two instance lines with messages passing between them. If the diagram is a unit, the icon has a small gray file overlay.

To invoke the pop-up menu for a sequence diagram, right-click the name of the diagram. The pop-up menu contains the following options:

- ◆ **Open Sequence Diagram**—Opens the selected sequence diagram in the drawing area.
- ◆ **Features**—Opens the Features dialog box for the sequence diagram.
- ◆ **Features in New Window**—Opens the Features dialog box for the sequence diagram in a separate window.
- ◆ **Add New**—Enables you to add a new dependency (see [Dependencies](#)), annotation (see [Adding Annotations to Diagrams](#)), hyperlink (see [Hyperlinks](#)), or tag (see [Using Tags to Add Element Information](#)).
- ◆ **References**—Enables you to search for references to the diagram in the model (see [Finding Element References](#)).
- ◆ **Unit**—Enables you to either make the sequence diagram a unit that you can add to a CM archive (**Save**) or modify an existing unit (**Edit Unit**).
- ◆ **Configuration Management**—Provides access to common CM operations for the sequence diagram, including Add to archive, Check In, Check Out, Lock, and Unlock.
- ◆ **Format**—Changes the format used to draw the element (color, line style, and so on). See [Changing the Format of a Single Element](#) for more information.
- ◆ **Delete from Model**—Deletes the sequence diagram from the entire model.

Sequence Comparison

During the development process, sequence diagrams (SDs) are used for the following primary purposes:

- ◆ In the early system requirements phase, they are used for use case description.
- ◆ In the implementation phase, they verify that all conditions are met in terms of communication between classes.
- ◆ In the testing phase, they capture the actual system trace.

Therefore, there is a need to facilitate comparison between SDs because, in principle at least, they should be identical. The execution message sequence should match the specification message sequence. The Sequence Diagram Comparison tool enables you to perform comparisons, for example between hypothetical and actual message sequences. You could also use this tool to compare two runs for regression testing.

If all execution SDs are identical to their corresponding specification (nonanimated) SDs, the system satisfies the requirements as captured in the use cases. However, if there are differences, you need to determine whether the specification was inaccurate or an error exists in the implementation. In both cases, you should correct the modeling error—either in the statechart or the SD—and then repeat the testing cycle to determine whether you have fixed the problem.

Sequence Comparison Algorithm

When comparing sequences, the following message parameters are used to determine whether the messages in the two SDs are identical:

- ◆ Departure time
- ◆ Arrival time
- ◆ Arguments

One simple approach involves comparing the exact position of every message and stopping at the first difference. However, this is probably too naive a comparison. For example, if there is a time offset in one SD, this kind of comparison would stop at the first message.

A more useful approach, therefore, is to take all events—message departures and arrivals—in order, and compare them without using the exact time. This kind of comparison, although simple, still shows when two SDs are essentially identical.

Because some messages can be “noise,” the comparison algorithm should also be able to decide whether a message is legitimate, and if not, mark it and continue with the comparison starting with the next message.

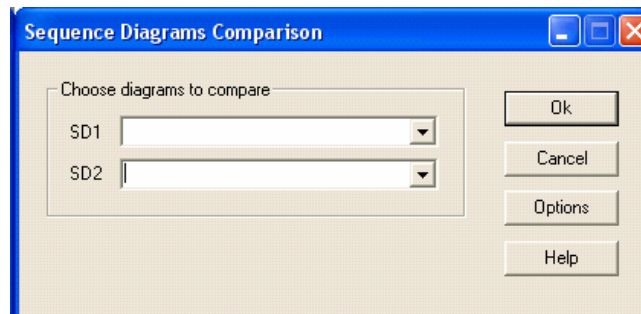
The point in comparing two SDs is not to show when one sequence is identical to another, but rather where and why they are different. Therefore, yes/no answers are not sufficient. Proper results must detail precisely what is identical and what is different. This is the approach that Rhapsody takes when comparing message sequences.

Using Sequence Comparison

Once you have saved two SDs illustrating the same scenario—for example, a specification and an execution version or two subsequent runs to test for regression—you are ready to start the sequence comparison.

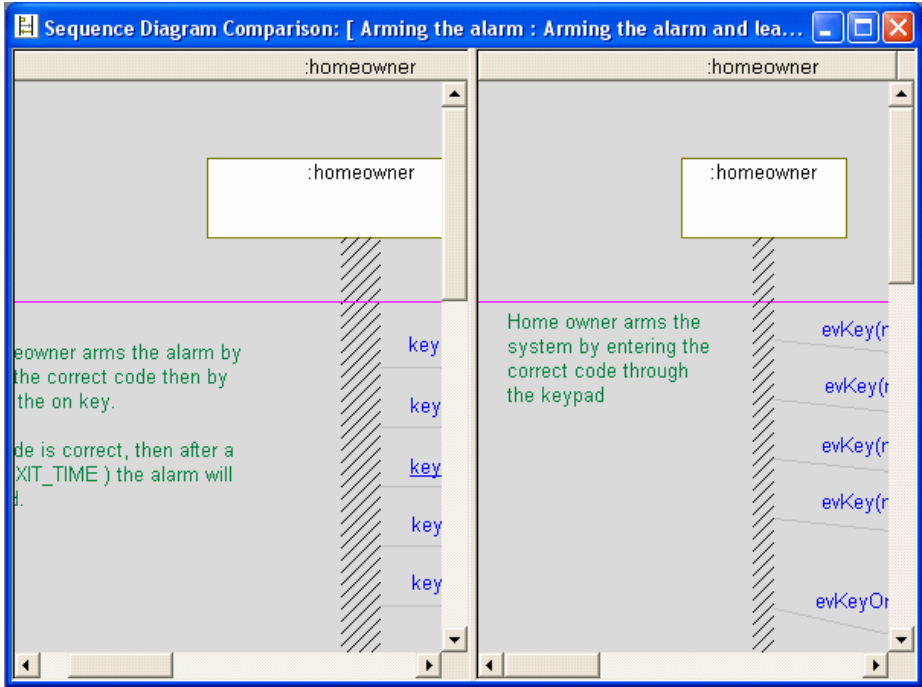
To start the comparison, follow these steps:

1. Select **Tools > Sequence Diagram Compare**. The Sequence Diagrams Comparison dialog opens, as shown in the following figure.



2. Using the **SD1** and **SD2** drop-down list controls, select two sequence diagrams to compare.
3. Set options for the sequence comparison as desired. See [Setting Sequence Comparison Options](#).
4. When all options are set and you are ready to start the comparison, click **OK**.

The result of the comparison appears as a dual-pane window with the diagram selected for SD1 on the left, and the diagram selected for SD2 on the right. Both panes are read-only. The following figure shows sample results.



The messages displayed in both panes are color-coded based on the comparison results. The following table lists the color conventions used in the comparison.

Arrow Color	Name Color	Description
Green	Blue	Message matches in both SDs
Pink	Pink	Message is missing in the other SD
Green	Pink	Message has different arguments in the other SD
Orange	Orange	Message arrives at a different time in the other SD
Gray	Gray	Message was excluded from comparison

Setting Sequence Comparison Options

Specification Sequence Diagrams show only one specific thread from the mind of the designer. Therefore, certain instances and messages will be missing. On the other hand, execution (animated) Sequence Diagrams reflect the full collaboration between objects. This is why the simple comparison between specification and execution Sequence Diagrams always fails. Rhapsody provides various options that enable you to compensate for some of the necessary differences between the two kinds of diagrams when doing a sequence comparison.

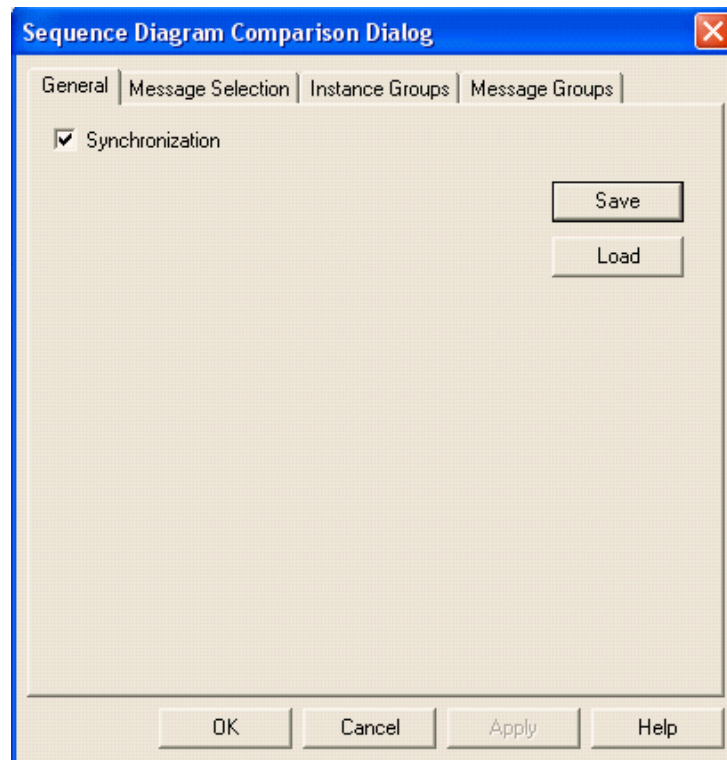
Select **Options** in the Sequence Diagrams Comparison dialog box to invoke the Sequence Diagram Comparison options dialog box. This dialog box contains the following tabs:

- ◆ [The General Tab](#)
- ◆ [The Message Selection Tab](#)
- ◆ [The Instance Groups Tab](#)
- ◆ [The Message Groups Tab](#)

The following sections describe how to use these tabs in detail.

The General Tab

The **General** tab, shown in the following example, allows you to specify whether to use synchronization and to save or upload your option settings.



The tab contains the following fields:

- ◆ **Synchronization**—Specifies whether to ignore the arrival times of messages. See [Ignoring Message Arrival Times](#) for more information.
- ◆ **Save**—Saves your option settings to a file that you can reuse. See [Saving and Loading Options Settings](#) for more information.
- ◆ **Load**—Loads your option file. See [Saving and Loading Options Settings](#) for more information.

Ignoring Message Arrival Times

Sometimes the order of arriving messages is insignificant. The **Synchronization** option enables you to ignore the arrival times of messages and consider only the order in which they are sent.

In the resulting comparison display, equivalent messages are vertically synchronized in the adjacent window panes. This helps you to locate corresponding messages in both diagrams.

To enable or disable the synchronization option, check or uncheck the **Synchronization** box in the **General** tab.

Saving and Loading Options Settings

You can save your options settings to a file and then reload them for subsequent message comparisons.

To save the settings, follow these steps:

1. Click the **Save** button on the **General** tab.
2. The Save As dialog box opens. The default name for the options file is composed of the first words of the titles of each of the diagrams being compared separated by an underscore:

`<SD2>_<SD1>.sdo`

The file extension `.sdo` stands for Sequence Diagram Options. If desired, edit the path and default name for the options file.

3. Click **OK** to save the options file.

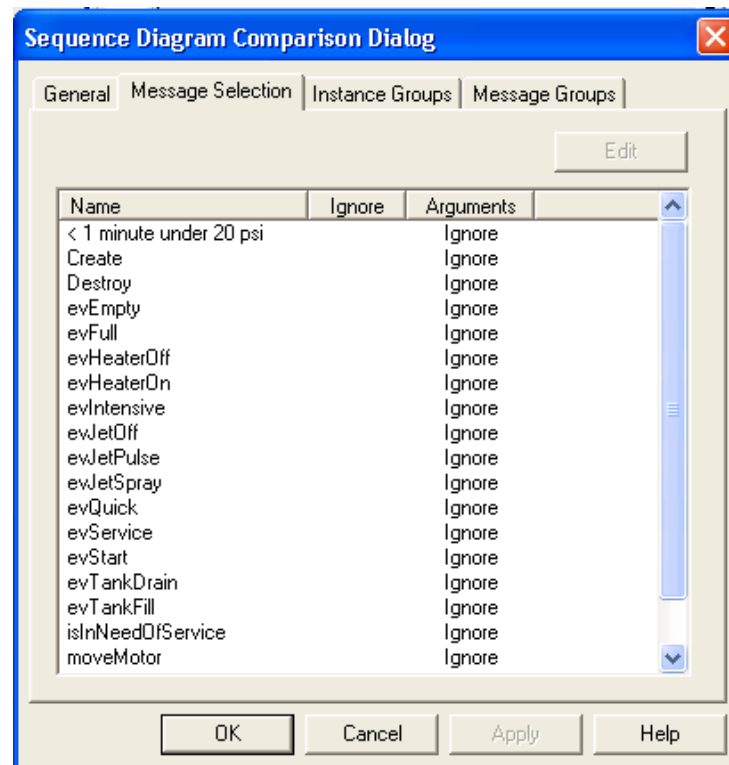
To reload your option settings, follow these steps:

1. In the **General** tab, click **Load**. The Open dialog box is displayed.
2. Select the `.sdo` file that contains your option settings.
3. Click **Open**.

The sequence comparison options are restored to the settings last saved in the file.

The Message Selection Tab

The **Message Selection** tab, shown in the following figure, enables you to select which messages to include and whether to include arguments in the comparison.



On this **Message Selection** tab, the word “Ignore” is the default setting for the Arguments column for all messages. This means that, by default, argument comparison is ignored for messages.

Using this tab, you can:

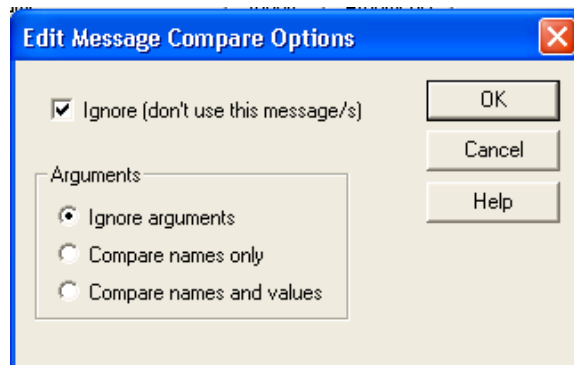
- ◆ Exclude a message from the comparison (see [Excluding a Message in the Comparison](#))
- ◆ Compare arguments (see [Comparing Arguments](#))

Excluding a Message in the Comparison

Specification SDs typically include information that is essential to a particular use case or scenario. In many cases, they exclude the initialization phase messages, whereas execution SDs include all messages. Therefore, it might be necessary ignore certain messages when doing a comparison, such as constructors. Ignored messages appear grayed out in the resulting comparison screen.

To exclude a message from the comparison, follow these steps:

1. Select the message to exclude.
2. Click **Edit**. The Edit Message Compare Options dialog box opens, as shown in the following example.



3. Click the **Ignore** box to exclude the message from the comparison.
4. The three radio buttons, allow you to specify the way to treat **Arguments** associated with the selected message.
5. Click **OK**.

Comparing Arguments

There are two options for determining whether messages are identical: the first is to compare the message names and all arguments, the second is to compare only the message names. The latter option is more useful because SDs show four different kinds of arguments:

- ◆ Unspecified arguments
- ◆ Actual values
- ◆ Formal names
- ◆ Both names and values

In specification SDs, you might not always provide complete information about message arguments. Because execution SDs record what the system actually does, they always show both argument values and names. Therefore, the message comparison ideally should not use arguments but rather focus primarily on message names.

When two messages are named identically, you can compare their arguments.

For example, consider messages called `evDigitDialed(Digit)`. They would be equivalent if you compared only their argument names (`Digit`). However, if you compared their values (`EvDigitDialed(Digit=0)`, `EvDigitDialed(Digit=1)`, and so on), their argument values would not be equivalent.

Argument comparison occurs in the following steps:

1. Find each argument.
2. Find the argument name and value.
3. Determine whether to use the name, the value, or both for the comparison.

To specify whether to use argument names or values, follow these steps:

1. In the **Message Selection** tab, select a message and click **Edit**. The Edit Message Compare Options dialog box opens.

2. Select one of the following options:

- ◆ **Compare Names Only**—Compare argument names, but ignore their values.
- ◆ **Compare Names and Values**—Compare both argument names and values.

The following are commonly used settings:

Specification SD	Execution SD	Recommended Value
Message()	Message(Arg = 1)	Ignore Arguments
Message(Arg)	Message(Arg = 1)	Compare Names Only
Message(1)	Message(Arg = 1)	Compare Names and Values

3. Click **OK**.

Depending on your selections, the following labels are displayed in the Arguments column on the **Message Selection** tab:

- ◆ **Disable**—Messages for which arguments should be ignored
- ◆ **Name**—Messages for which argument names should be compared, but not the argument values
- ◆ **Value**—Messages for which both argument names and values should be compared

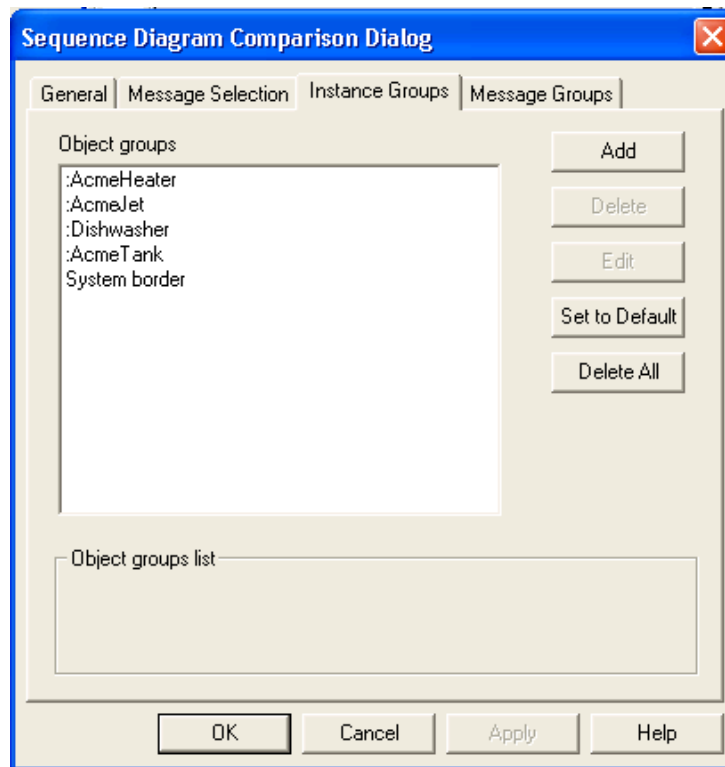
The Instance Groups Tab

In specification SDs, all messages sent by the environment come from specific objects. In execution SDs, however, these messages could potentially come from you interacting with the animation. This difference can impair the comparison.

In general, requiring a complete object match between execution and specification SDs is too rigorous a requirement. The solution is to associate objects in one SD with other objects in the other SD. Messages can then match if their source and target objects are associated in both SDs.

To associate objects with each other you create *object groups*. Object groups are, in essence, instance abstractions that bridge the gap between high-level use cases and actual implementation, or between black-box and white-box scenarios. Using object groups, you can then compare objects that do not have the same name, or compare one object to several other objects.

To view object groups, select the **Instance Groups** tab in the Sequence Diagram Comparison options dialog box. The **Instance Groups** tab, shown in the following figure, displays a list of the existing object groups in the model. There is one object group for each object—which is, by default, the only member of its own group.



The objects that belong to the group are displayed in the **Objects groups list** at the bottom of the dialog box. This means that the objects listed for SD1 are considered logically the same as those listed for SD2.

The **Instance Groups** tab enables you to perform the following operations:

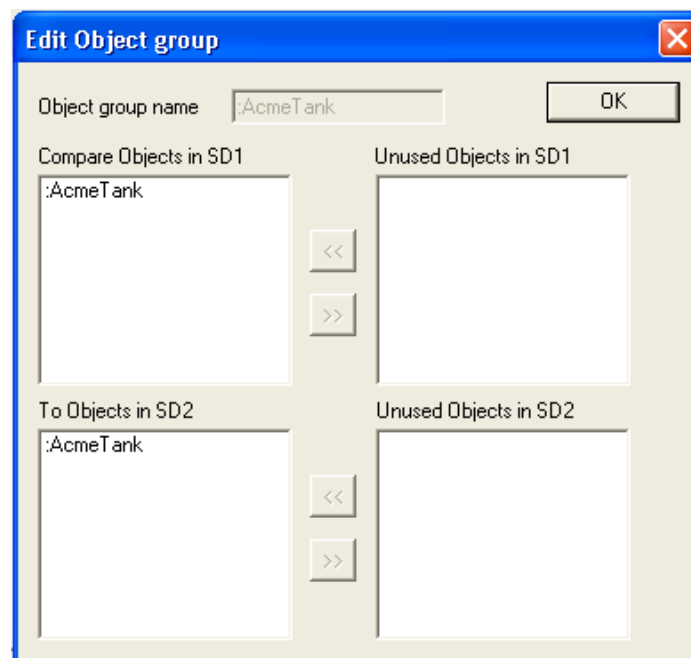
- ◆ **Add**—Creates a new object group (see [Creating Object Groups](#))
- ◆ **Delete**—Deletes an object group (see [Deleting Object Groups](#))
- ◆ **Edit**—Modifies an existing object group (see [Modifying Object Groups](#))
- ◆ **Set to Default**—Resets an object group (see [Resetting Object Groups](#))
- ◆ **Delete All**—Deletes all object groups (see [Deleting Object Groups](#))

Modifying Object Groups

If you want to associate different objects than the ones shown, either move one or more of the objects to a different object group or create a new group. In either case, you first need to remove the object you are moving from the group it is currently in, because an object can only belong to one group at a time.

To remove an object from a group, follow these steps:

1. On the **Instance Groups** tab, select an object group.
2. Click **Edit**. The Edit Object group dialog box opens.



The name of the selected object group is displayed at the top of the dialog box. The name box is grayed out because you cannot edit it here.

The Edit Object group dialog contains four boxes. The two on the left show which objects in SD1 will be associated with which objects in SD2. The two boxes on the right show which objects in each diagram are currently not assigned to any group, and are therefore available to be assigned to a group.

3. Select an object in one of the boxes on the left and click the right arrows button.

To add an object to the group, follow these steps:

1. Select an object in one of the boxes on the right, and move it with the left arrows button.
2. Click **OK**. The selected object in one diagram is now available to be added to another group.
3. On the **Instance Groups** tab, select the new group for the object and click **Edit**.
4. Select the object in the **Unused Objects in SD<number>** box in the lower, right corner and click the left arrows key to add it to the group.
5. Select the object in the **To Objects in SD<number>** box in the lower, right corner and click the right arrows button to remove it from the group.
6. Click **OK** to apply your changes and close the dialog box.

Creating Object Groups

To create a new instance group, follow these steps:

1. On the **Instance Groups** tab, click **Add**. The Edit Object Group dialog box opens. The default name for new object groups is `ClassBuffn`, where n is an integer starting with 1.
2. If desired, edit the name of the new object group.
3. To add objects to the group, move one or more unused objects from either of the boxes on the right to the corresponding box on the left.
4. Click **OK** to apply your changes and close the dialog box.

Deleting Object Groups

To delete an existing object group, follow these steps:

1. On the **Instance Groups** tab, select the object group you want to delete.
2. Click **Delete**.
3. Click **OK** to close the dialog box.

Any objects that belonged to the deleted group are now unused and available to be assigned to another object group.

To delete *all* instance groups, follow these steps:

1. On the **Instance Groups** tab, click **Delete All**.
2. Click **OK** to close the dialog box.

All objects are now unused and available to be assigned to a new object group.

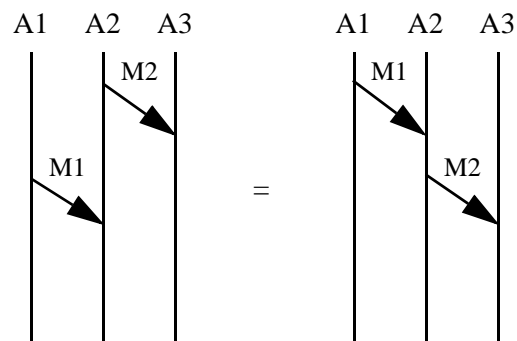
Resetting Object Groups

To set all object groups back to the default of one group per object, click **Set to Default** on the **Instance Groups** tab. An object group is added for each object with the same object in SD1 and SD2 belonging to the group.

The Message Groups Tab

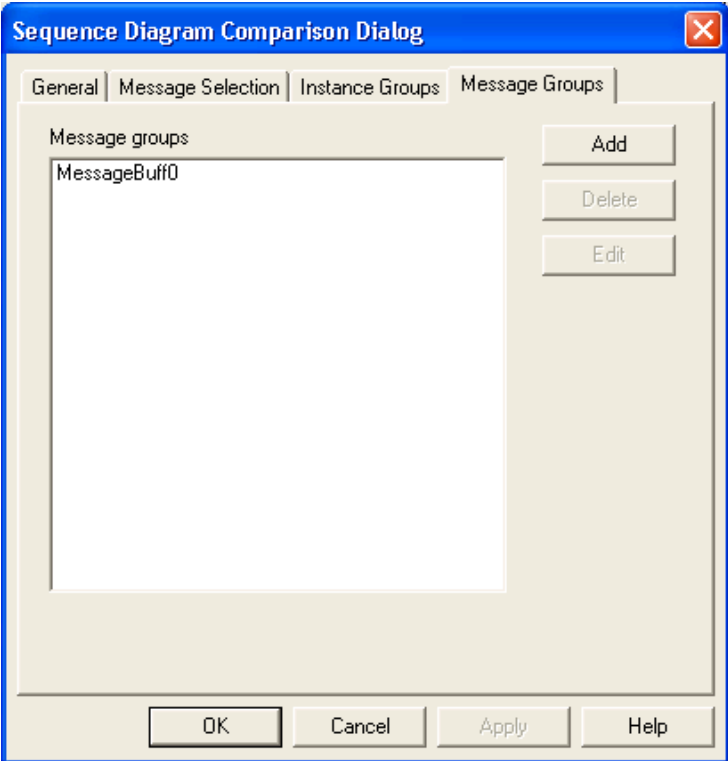
In specification SDs, you often must assume how the message queue works to determine the sequence of messages. It is highly likely that in specification SDs the order of messages will be different than the actual one specified in the statechart. An incorrect ordering assumption can result in large mismatch.

To avoid this problem, the comparison must be able to ignore the timing of messages. For example, a message M1 sent by an instance A1 after a message M2 sent by an instance A2 could match the same message sent before M2:



There can also be cases where two or more messages should be sent at the same time, but the order is not important. *Message groups* enable you to specify groups of messages for which ordering is not important. There is a match if any message in the group occurs in any order.

The **Message Groups** tab, shown in the following example, enables you to create, modify, and delete message groups.



Creating Message Groups

To create a message group, follow these steps:

1. On the **Message Groups** tab, click **Add**. The Edit Message Group dialog box opens.



The default name for new message groups is `MessageBuffn`, where n is an integer starting with 0.

2. If desired, edit the name of the new message group.

The Edit Message Group dialog contains two list boxes: the left one shows the messages that currently belong to message group, whereas the right one shows all messages in the two SDs being compared. Messages can belong to more than one message group.

Adding a Message to a Message Group

To add a message to the message group, follow these steps:

1. On the **Message Groups** tab, click **Add**.
2. Select a message from the **All Messages** list and click the left arrows button to move it to the **Messages in group** list. For multiple selections, use **Shift+Click** or **Ctrl+Click**.
3. Click **OK** to apply your changes and close the dialog box.

Removing a Message from a Message Group

To remove a message from the message group, follow these steps:

1. On the **Message Groups** tab, click **Add**.
2. Select a message from the **Messages in group** list and click the right arrows button to move it to the **All Messages** list. For multiple selections, use **Shift+Click** or **Ctrl+Click**.
3. Click **OK** to apply your changes and close the dialog box.

Determining the Message Group Members

To see which messages belong to a message group, follow these steps:

1. On the **Message Groups** tab, select a message group from the list.

The messages that belong to that group are listed at the bottom of the dialog box, as shown in this example.



2. Click **OK** to close the dialog box.

Modifying Message Groups

To modify an existing message group, follow these steps:

1. On the **Message Groups** tab, select a message group from the list and click **Edit**. The Edit Message Group dialog box opens.

2. Select a message in the **Messages in group** list and click the right arrows button to remove it from the group.

Select a message in the **All Messages** list and click the left arrows button to add it to the group.

3. Click **OK** to apply your changes and close the dialog box.

Deleting Message Groups

To delete an existing message group, follow these steps:

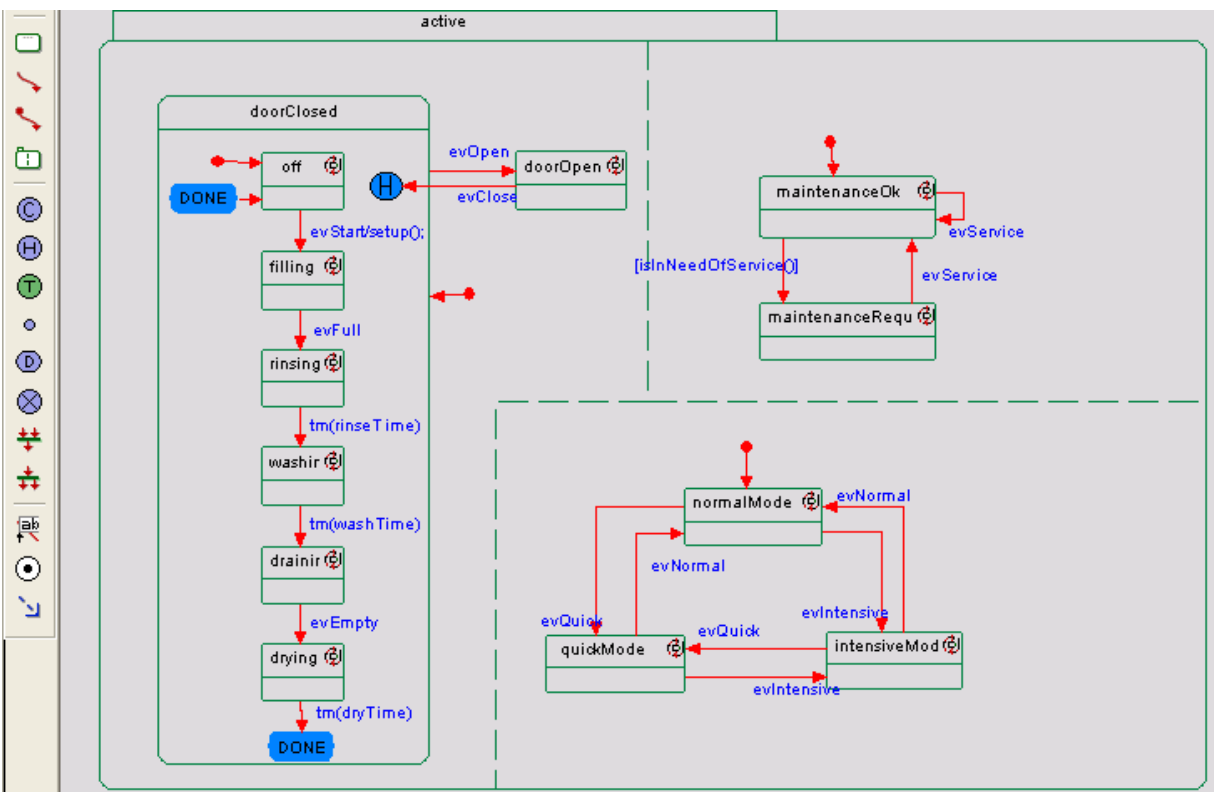
1. In the Sequence Diagram Comparison options dialog box, select the message group to delete.
2. Click **Delete**.
3. Answer **OK** to the confirmation prompt.

Statecharts

Statecharts define the behavior of objects by specifying how they react to events or operations. The reaction can be to perform a transition between states and possibly to execute some actions. When running in animation mode, Rhapsody highlights the transitions between states.

Statecharts define the run-time behavior of instances of a class. A state in a statechart is an abstraction of the mode in which the object finds itself. A message triggers a transition from one state to another. A message can be either an event or a triggered operation. An object can receive both kinds of messages when sent from other objects. An object can always receive events it sends to itself (self-messages). In Rhapsody, statecharts are part of the object-oriented paradigm. The more complicated classes can have statecharts; simpler classes do not require them.

You can use operations and attributes in classes with statecharts to define guards and actions, as in the following example.

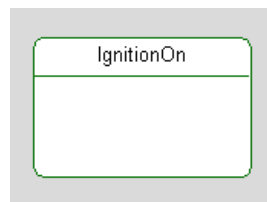


States

A *state* is a graphical representation of the status of an object. It typically reflects a certain set of its internal data (attributes) and relations. In statecharts, states can be broken down hierarchically as follows:

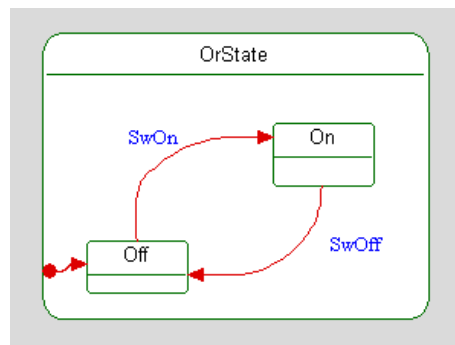
- ◆ **Basic (leaf) state**—A state that does not have any substates.

In the following example, the state represents a car ignition that is turned on.

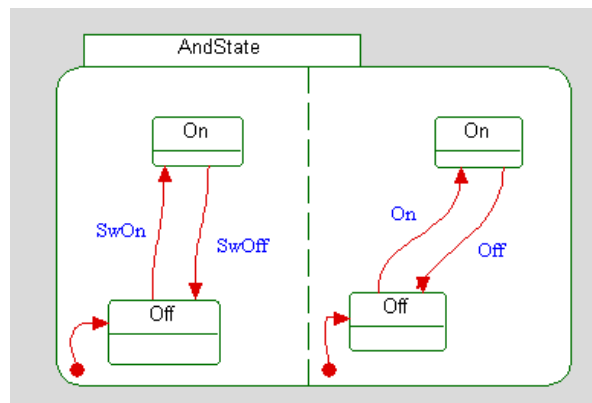


- ◆ **Or state**—A state that can be broken down into exclusive substates. This means that the object is exclusively in one or the other of its substates.

In the following example, there are possible two states—On and Off.



- ◆ **And state**—An object is in each of its substates concurrently. Each of the concurrent substates is called an *orthogonal component*. You can convert an Or state to an And state by dividing it with an And line. See [And Lines](#) for more information.











You set the statechart implementation in the **Settings** tab of the Configuration dialog box in the browser.

Statechart Drawing Icons

The **Drawing** toolbar for a statechart includes the following tools:


Drawing Icon	Description
	State indicates the current condition of an object, such as On or Off. See States for more information.
	Transition represents a message or event that cause an object to transition from one state to another. See Transitions for more information.
	Default connector shows the default state of an object when first instantiated. See Default Transitions for more information.
	And line separates the orthogonal components of an And state. There can be two or more orthogonal components in a given And state and each behaves independently of the others. See And Lines for more information.
	Condition connector shows the branches on transitions, based on Boolean conditions called guards. See Condition Connectors for more information.
	History connector stores the most recent active configuration of a state. A transition to a history connector restores this configuration. See History Connectors for more information.
	Termination connector ends the life of the object. See Termination Connectors for more information.
	Junction connector joins multiple transitions into a single, outgoing transition. See Junction Connectors for more information.

Drawing Icon	Description
	Diagram connector joins physically distant transition segments. Matching names on the source and target diagram connectors define the jump from one segment to the next. See Diagram Connectors for more information.
	EnterExit point represents the entry to / exit from sub-statecharts. See EnterExit Points for more information.
	Draw Join Sync Bar merges multiple incoming transitions into a single outgoing transition. See Activity Diagrams for more information.
	Draw Fork Sync Bar separates a single incoming transition into multiple outgoing transitions. See Activity Diagrams for more information.
	Transition Label add or modify a text describing a transition. See Activity Diagrams for more information.
	Termination State signifies either local or global termination, depending on where they are placed in the diagram. See Activity Diagrams for more information.
	Dependency icon indicates a dependent relationship between two items in the diagram. See Activity Diagrams for more information.
	Send Action State represents the sending of events to external entities. See Send Action State Elements for more information.

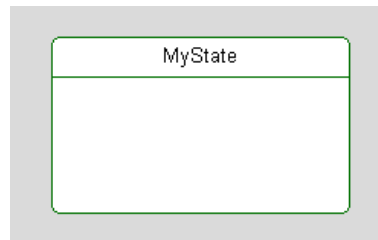
The following sections describe how to use these tools to draw the parts of a statechart. See [Graphic Editors](#) for basic information on diagrams, including how to create, open, and delete them.

Drawing a State

To draw a state, follow these steps:

1. Click the **State**  icon in the **Drawing** toolbar.
2. Click-and-drag or click in the drawing area to create a state with a default name of `state_n`, where n is an incremental integer starting with 0.
3. If desired, change the state name, then press **Enter**.

States include a standard name compartment, as shown in this example.



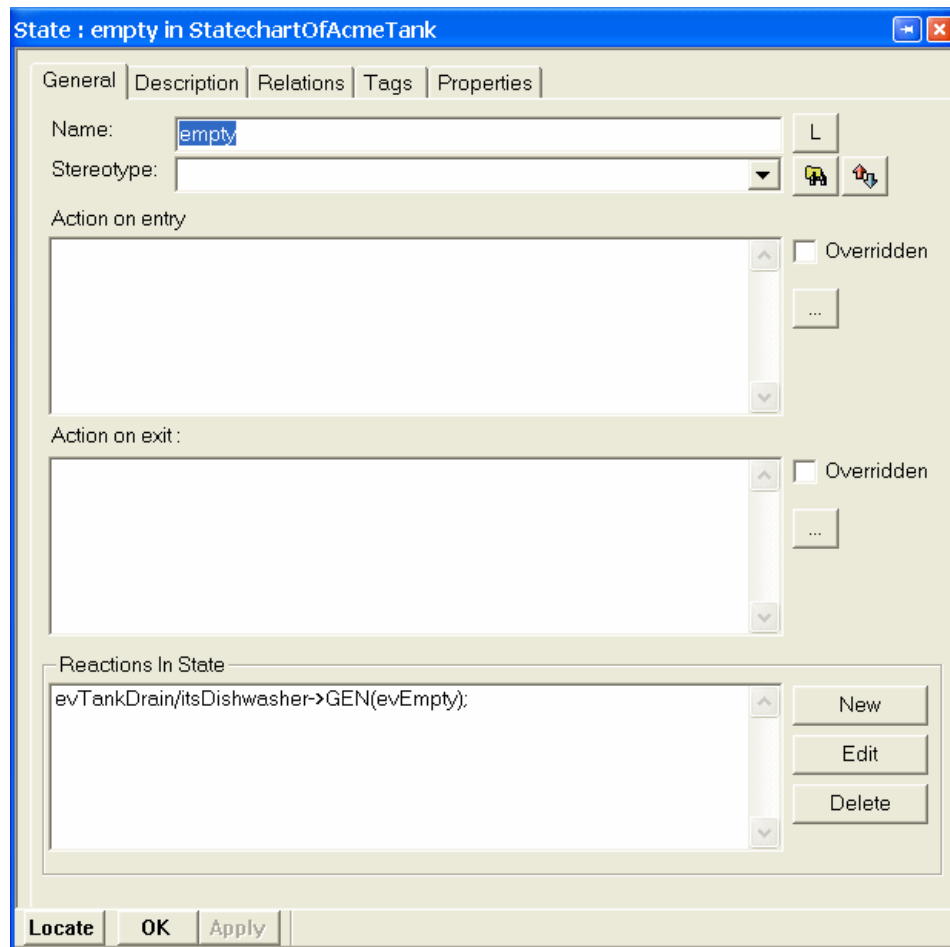
State Name Guidelines

When naming states, follow these guidelines:

- ◆ Must be identifiers.
- ◆ Do not include spaces, “My House” is not valid. Use “MyHouse.”
- ◆ Must be unique among sibling states.
- ◆ Should not be the same as the names of any events or classes in the model.

Modifying the Features of a State

The **Features** dialog box allows you to add and change a state's features, as shown below.



A state has the following General tab features:

- ◆ **Name**—Specifies the name of the state.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the state, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.

- ◆ **Action on entry**—Specifies the action that should be executed whenever the system enters this state, regardless of how the system arrived here.

If the action on entry value is overridden, the **Overridden** check box is checked. The **Overridden** check box is available in the Features dialog boxes for textual information in statecharts (state entry and exit actions, and guards and actions for transitions and static reactions). By enabling or disabling this check box, you can easily override and unoverride statechart inheritance without actually changing the model. As you toggle the check box on and off, you can view the inherited information in each of the dialog box fields, and can decide whether to apply the information or revert back to the currently overridden information. See [Overriding Textual Information](#) for more information.

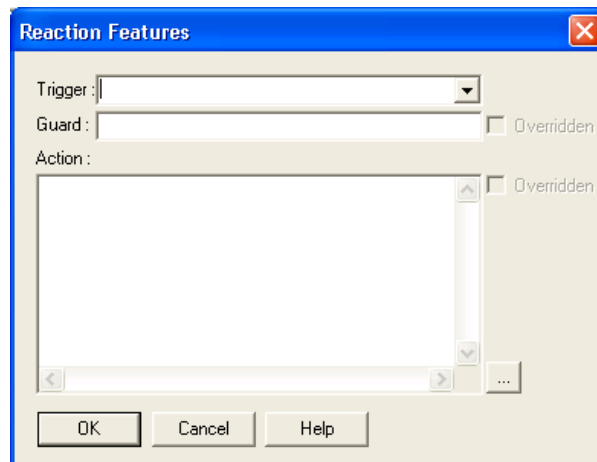
- ◆ **Action on exit**—Specifies the action that should be executed whenever the system exits this state, regardless of how the system exits.

If the action on entry value is overridden, the **Overridden** check box is checked.

- ◆ **Reactions In State**—Specifies the trigger, guard, and actions identified in a transition label. If the trigger occurs and the guard is true, the action is executed. See [Transition Labels](#) for more information.

Use the appropriate dialog box button:

- **New**—Creates a new reaction in state. If you select this option, the Reaction Features dialog box opens so you can specify the new reaction.



- **Edit**—Modifies an existing reaction.
- **Delete**—Deletes a reaction.

Note

If you specify action on entry or exit behavior for a state, the icon shown in the following example is added to the state display.

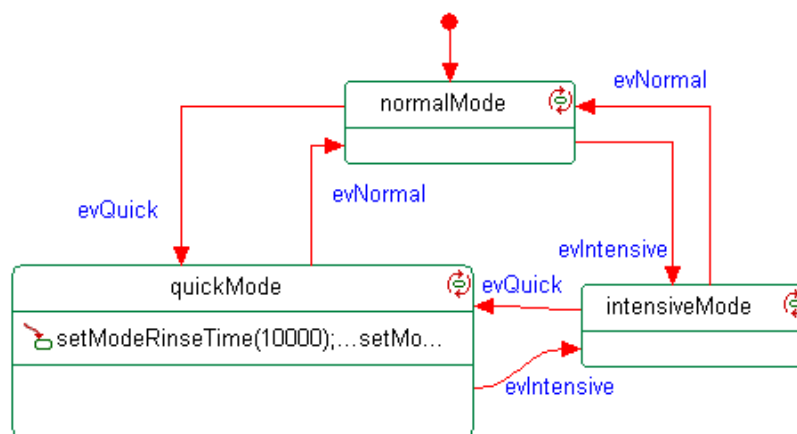


Display Options for States

Using the **Display Options** option in the pop-up menu, you can specify whether:

- ◆ States are displayed with a name or label.
- ◆ Stereotypes are displayed with a name, label, or icon.
- ◆ The exit and entry actions are displayed.

In the following figure, the display for the quickMode state includes its entry and exit actions.



Termination States

A *termination state* provides local termination semantics. Local termination implies the completion of a composite state without the destruction of the context instance.

There are two different modes for local termination:

- ◆ **Statechart mode**—Local termination applies only to composite states with termination states inside them.
- ◆ **Activity diagram mode**—Local termination applies to every block and composite state, even those that do not have internal termination states. (This is the UML statechart/activity diagram-supported mode.)

The `CG::Statechart::LocalTerminationSemantics` property specifies whether activity diagram mode of local termination is enabled. The default value of `Cleared` means that the activity diagram mode of local termination is not enabled (statechart mode is enabled).

Local Termination Code with the Reusable Statechart Implementation

The following sections describe how code is generated to support local termination when you use the reusable statechart implementation.

Or States

Code is generated for local termination of Or states with the reusable statechart implementation as follows:

- ◆ For each termination state, a `Concept` class data member of type `FinalState` is generated. A new class is not created as for other state types.
- ◆ The following local termination guard is added to each outgoing null transition from an Or state that needs local termination semantics:


```
&& IS_COMPLETED(state)
```
- ◆ If an Or state has several termination states, an outgoing null transition is activated when any one of them is reached. However, the specific connector is instrumented.
- ◆ The `isCompleted()` function is overridden for an Or state that has a termination state, returning `True` when the termination state is reached. The function is also overridden for an Or state without a termination state in activity diagram mode, always returning `False`.
- ◆ An instance of a `FinalState` is created by a line similar to the following:

```
FinalA = new FinalState(this, OrState, rootState,
    "ROOT.OrState.FinalA");
```

And States

Code is generated for local termination of And states with the reusable statechart implementation as follows:

- ◆ The following local termination guard is added to each outgoing null transition from an And state if one of the components has a termination state:

```
&& IS_COMPLETED(AndState)
```

In this case, the `isCompleted()` function of the `AndState` framework class is called.

- ◆ The following local termination guard is added to a join transition for each Or state that is a source of the transition:

```
&& IS_COMPLETED(state)
```

- ◆ If a source state of a join transition is a simple state (leaf state), its guard is as follows:

```
(IS_IN(state))
```

The following is an example of the code generated for a join transition with a real guard and local termination guards, where `C1` and `C2` are Or states with termination states and `C3` is a leaf state:

```
if(RealGuard() && IS_COMPLETED(C1) && IS_COMPLETED(C2) && IS_IN(C3))
```

Local Termination Code with Flat Statechart Implementation

The following sections describe how code is generated to support local termination when the flat statechart implementation is used.

Or States

Code is generated for local termination of Or states with the flat statechart implementation as follows:

- ◆ For each termination state, the new state enumeration value is generated (as for a regular state).
- ◆ For each Or state with a termination state, a `<StateName>_isCompleted()` operation is generated. This operation returns an `OMBoolean` value of `True` when the state is completed. If the `CG::Class::IsCompletedForAllStates` property is `Checked`, the operation is generated for all states.

The following is an example of the code generated for a `<StateName>_isCompleted()` operation where `FinalA` and `FinalB` are termination states in the Or state:

```
inline OMBoolean
class_0::OrState_isCompleted() {
    return (FinalA_IN() || FinalB_IN());
}
```

- ◆ The following local termination guard is added to each outgoing null transition from an Or state that needs local termination semantics:

```
&& IS_COMPLETED(state)
```

- ◆ Instrumentation information for `FinalState` is generated in the transition code (as for normal states).

And States

Code is generated for local termination of And states with the flat statechart implementation as follows:

- ◆ The following local termination guard is added to each outgoing null transition from an And state if one of the components has a termination state.

```
&& IS_COMPLETED(AndState)
```

In this case, the `isCompleted()` function of the `AndState` framework class is called:

- ◆ The `isCompleted()` operation of `AndState` calls the `IS_COMPLETED()` macro for all components that have a termination state. This operation returns `TRUE` only when all components are completed. If an And state does not have components with a termination state, the operation returns `TRUE` in statechart mode and `FALSE` in activity diagram mode.

The following is an example of the `<StateName>_isCompleted()` function generated for an And state named `AndState`, with two components, `Component1` and `Component2`, each of which has a termination state:

```
OMBoolean class_0::AndState_isCompleted()
{
    if(IS_COMPLETED(Component1) == FALSE)
        return FALSE;
    if(IS_COMPLETED(Component2) == FALSE)
        return FALSE;
    return TRUE;
}
```

- ◆ Implementation of join transitions with the flat statechart implementation is the same as for the reusable statechart implementation (see [And States](#)).

Transitions

A *basic transition* is composed of a single arrow between a source and a destination. Transitions represent the response to a message in a given state. They show what the next state will be, given a certain trigger. A transition can have a trigger, guard, and actions.

The transition context is the scope in which the message data (parameters) are visible. Any guard and action inherit the context of a transition determining the parameters that can be referenced within it.

The source of a transition can be one of the following:


- ◆ State
- ◆ Default transition
- ◆ History connector

The destination of a transition can be one of the following:

- ◆ State
- ◆ Termination connector
- ◆ History connector

Creating a Statechart Transition

To draw a statechart transition, follow these steps:

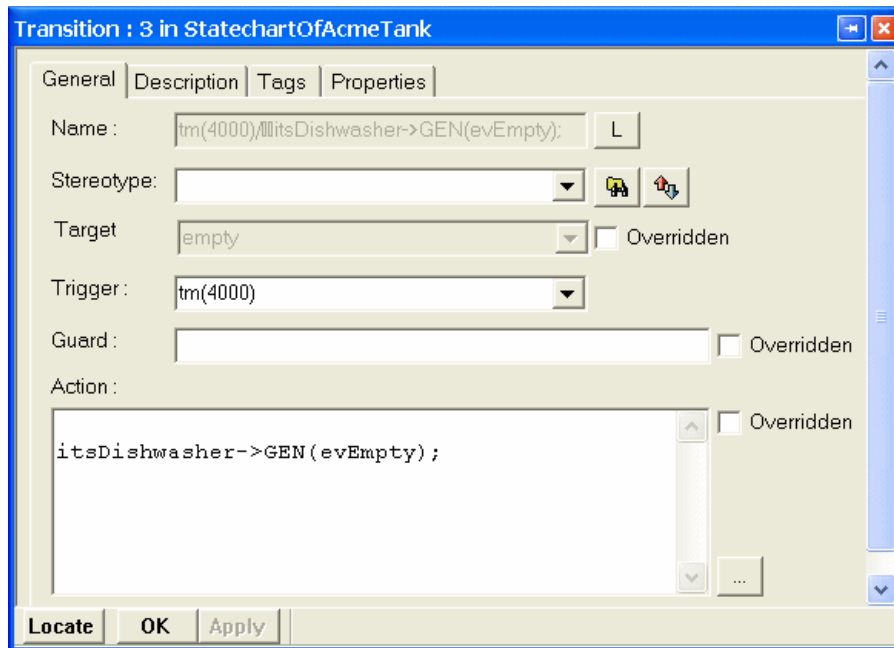
1. Click the **Transition**  icon in the **Drawing** toolbar.
2. Click the bottom edge of the state to anchor the start of the transition.
3. Move the cursor to the top edge of the state and click to anchor the transition line.
4. In the label box, type the name of the event. Press **Ctrl+Enter** to terminate.

Note

Pressing Enter in a transition name, without simultaneously pressing Ctrl, simply adds a new line.

Modifying the Features of a Transition

The Features dialog box allows you to add and change a transition's features, as shown in this example.



A transition has the following General tab features:

- ◆ **Name**—Specifies the name of the transition.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the element, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM Development Guide* for more information about these components.

- ◆ **Target**—Specifies the target of the transition. This field is read-only.
- ◆ **Trigger**—Specifies the trigger for the transition. See [Triggers](#).
- ◆ **Guard**—Specifies the guard for a transition.

The **Overridden** check box is available in the Features dialog boxes for textual information in statecharts (state entry and exit actions, and guards and actions for transitions and static reactions). By enabling or disabling this check box, you can easily override and unoverride statechart inheritance without actually changing the model. As

you toggle the check box on and off, you can view the inherited information in each of the dialog box fields, and can decide whether to apply the information or revert back to the currently overridden information. See [Overriding Textual Information](#) for more information.

- ◆ **Action**—Specifies the transition action.

Types of Transitions

You can create the following types of transitions:

- ◆ Compound transitions
- ◆ Forks
- ◆ Joins

Compound Transitions

A *compound transition* is composed of several transition segments connected by intermediate nodes. A transition segment is a subpart of a transition between any source, destination, or intermediate nodes.

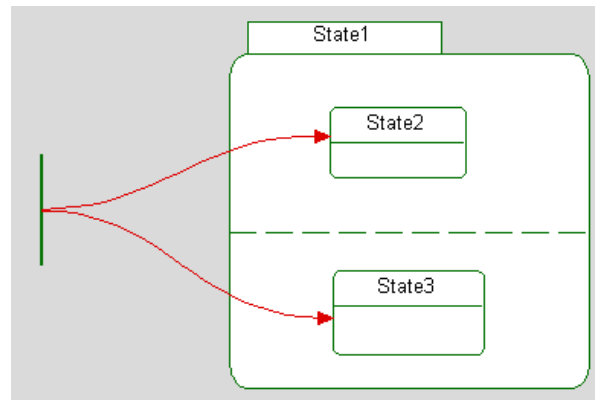
The intermediate nodes can be any of the following:

- ◆ [Forks](#)
- ◆ [Joins](#)
- ◆ [Junction Connectors](#)
- ◆ [Condition Connectors](#)
- ◆ [Diagram Connectors](#)

In semantic terms, forks and joins can both be nodes.

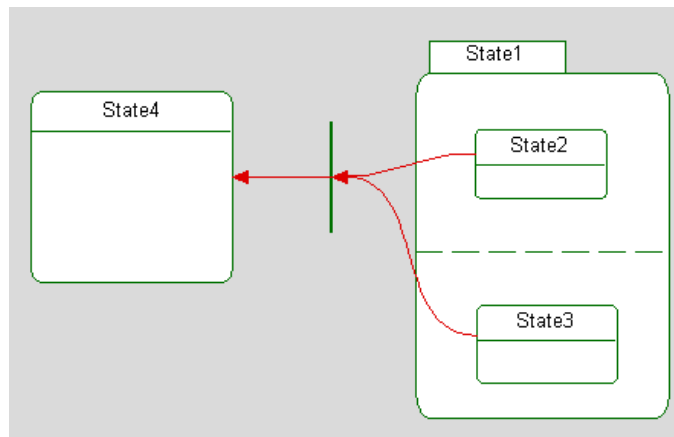
Forks

A *fork* is a compound transition that connects a transition to more than one orthogonal destination. The destination of a fork segment can be a state, termination state, or connector. However, it cannot have a label, as shown in the this example.



Joins

A *join* is a compound transition that merges segments originating from states, termination states, or connectors in different orthogonal components. Rhapsody automatically generates a join when you combine source segments. The segments entering a join connector should not have labels, as shown in the following figure.



Note

If the model contains joins from more than two concurrent states, this generates an error. This is a Rhapsody limitation in that the error is a violation of MISRA rule 33.

Selecting a Trigger Transition

For a list of transitions defined in the diagram, right-click a transition line in the diagram and select **Select Trigger**. The pop-up menu that appears lets you select one of the available triggers, including inherited triggers, that have already been defined for the class.

Notice that if there are more triggers that can appear on the pop-up menu, a **Browse** command appears. Click **Browse** to open the Select Trigger dialog box that shows you all the triggers that are available.

See also [Selecting a Message or Trigger](#).

Transition Labels

Transition labels can contain the following parts:

- ◆ [Triggers](#)
- ◆ [Guards](#)
- ◆ [Actions](#)

The syntax for transitions is as follows:

```
trigger [guard] /action
```

The following is an example of a transition label consisting of a timeout trigger (see [Timeouts](#)), a guard, and an action:

```
tm(500) [isOk()]/printf("a 0.5 second timeout occurred\n")
```

In this example, the trigger is the timeout `tm(500)` and the guard is `[isOk()]`. The action to be performed if the trigger occurs and the guard is true is `printf("a 0.5 second timeout occurred\n")`.

All three parts of the transition are optional. For example, you can have a transition with only a trigger and an action, or only a guard. The following is an example of a transition label consisting of only a trigger and an action:

```
clockw /itsEngine->GEN(start)
```

When typing a multiline transition label—for example, one that has several actions separated by semicolons—you can press **Ctrl+Enter** to advance the cursor to the next line and continue the label.

Triggers

Every transition is associated with a designated message, which is the *trigger* of the transition. In other words, the trigger of the transition is waiting for its event. A transition cannot be associated with more than one message. Triggers can be events, triggered operations, or timeouts.

Events are asynchronous; time can pass between the sending of the event and when it actually affects the statechart of the destination. Triggered operations are synchronous; their effect on the statechart of the destination is immediate.

The following is *not* a valid transition label:

```
e1 or e2
```

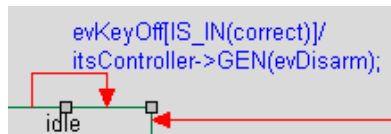
The trigger part of a transition label cannot use conditional expressions; however, guards can.

Events

Events originate in the analysis process as “happenings” within the system to which objects respond. Their concrete realizations are information entities carrying data about the occurrence they describe.

For designers, an event is a one-way, asynchronous communication between objects or between some external interface and the system (see [Events and Operations](#)).

The following figure shows an event.



You can specify an event using the following methods:

- ◆ An event name
- ◆ The name of a triggered operation
- ◆ `tm(time expression)`

For timeout events, the time expression must be an integer expression, in milliseconds.

Event Usage

Classes and their statecharts can receive events defined in any package in the model. Cross-package inheritance of statecharts for reactive classes is not allowed.

Event Class Hierarchy

An event is an instance of a particular event-class. Events can be subclassed to add attributes (event parameters). The base class for all events is `OMEvent`.

For example, windowing systems define several event classes: `MouseEvent` is a subclass of `InputEvent`, and `MouseClickedEvent` and `MouseMotionEvent` are subclasses of `MouseEvent`.

To make an event a subclass of another event:

1. Open the Features dialog for the event.
2. From the *Inherits:* drop-down list, select the event that is to serve as the base class.
3. Click **Apply** or **OK**.

Generating Events

You generate an event by applying a `gen` method on the destination object, as follows:

```
client->gen(new event(p1,p2,...,pn))
```

The generate method queues the event on the proper event queue.

The framework provides a `GEN` macro, which saves you from having to use the `new` operator to generate an event. For example:

```
client->GEN(event(p1, p2, pN))
```

Event Semantics

An event is created when it is sent by one object to another, then queued on the queue of the target object thread (thread partitioning is not covered in this guide). An event that gets to the head of the queue is dispatched to the target object. Once dispatched to an object, it is processed by the object according to the semantics of event propagation in statecharts and the run-to-completion semantics. After the event is processed, it no longer exists and the execution framework implicitly deletes it.

Internal Events

An internal event occurs when an object sends a message to itself. To create an internal event, omit the destination object from the send operation, as follows:

```
GEN(warmEngine(95))
```

Private Events

You can control which classes can generate events to which classes using friendship. In this way, you can ensure that events come from trusted classes only. The event request and queueing function is controlled by the `gen()` methods, which are public by default in the framework base class `OMReactive`. If you want to control the generation of events using friendship, make the first `gen()` method in `Share\oxf\OMReactive.h` protected. This is a one-time effort. Do not change the second `gen()` method, which is used for instrumentation.

Inside each application class, grant friendship to the classes that need to generate events for it. If you do not grant friendship, your program will no longer compile.

Adding Operations to an Event

Rhapsody allows you can add operations to events you have defined. This allows you to add additional behavior to your events.

To add an operation to an event:

1. Select the event in the browser.

2. Right-click to open the context menu, and select the option **Add New > Operation**.

The new operation will appear below the event in the browser, and when code is generated, the operation will appear in the class that represents the event.

Note

Roundtripping will not bring into the model new operations that you have added to event classes in your code, nor will it bring into the model changes that were made directly to the body of operations that were previously created for events.

Events as Attribute Types

Events can be used as types for attributes.

Triggered Operations

Triggered operations are services provided by a class to serve other classes. They provide synchronous communication between a client and a server object. Because its activation is synchronous, a triggered operation can return a value to the client object.

Unlike events, operations are not independent entities; they are part of a class definition. This means that operations are not organized in hierarchies.

The usage of operations corresponds to invocation of class methods in C++. There are three reasons why operations have been integrated with the statechart framework:

- ◆ They allow use of statecharts in architectures that are not event-driven to specify behaviors of objects in the programming sense of operations and object state.
- ◆ They provide for late design decisions to optimize execution time and sequencing by converting event communication into direct operation invocations.
- ◆ They allow the description of behaviors of (primitive) “passive” classes using statecharts.

Applying a Triggered Operation

A triggered operation is invoked in the same way as a primitive operation:

```
server->operation(p1, p2, ..., pn)
```

Or:

```
result = server->operation(p1, p2, ..., pn)
```

Operation Replies

Operations can return a value. The return value for an operation *m* must be determined within the transition whose context is the message *m*, using the *reply* operation:

```
m/reply(7);
```

Note

A triggered operation might not result in another triggered operation on the same instance.

Making Sure a Triggered Operation is Called

There might be a problem with the reply from triggered operations if the receiver object is not in a state in which it is able to react to a triggered operation. If a triggered operation is called when not expected, incorrect return values might result.

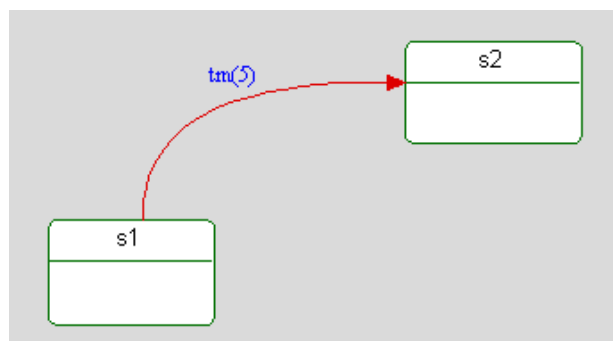
Rather than use the `IS_IN` macro to determine what state the receiver is in, you can design your statechart so the triggered operation is never ignored. To do this, create a superstate encompassing the substates in the object, and in the superstate create a static reaction with a trigger to return the proper value. For example, to make sure that a sensor is always read regardless of what state an object is in, create a static reaction in the superstate with the following transition:

```
opRead/reply(getStatus())
```

This way, no matter what substate the object is in, it will always return the proper value. Although both the trigger to the superstate and that to a substate are active when in a substate, the trigger on the transition to the superstate is taken because it is higher priority. See [Transition Selection](#).

Timeouts

Timeout triggers have the syntax `tm(<expression>)`, where *<expression>* is the number of time units. The default time unit is milliseconds. The time units are set based on the operating system adapter implementation of the tick timer. A timeout is scheduled when the origin state (`s1`) is entered. If the origin state has exited before the timeout was consumed, the timeout is canceled.



You can use the timeouts mechanism (`tm()`) when the quality of service (QOS) accuracy requirement conforms with the following timeout accuracy. When a timeout occurs, it is inserted to the event queue related to the reactive instance. The time on which the timeout is consumed depends on the actual system state. The timeout occurrence depends on three factors:

1. The timeout request time (T)

2. The tick-timer resolution (R)

The *resolution* specifies how often the system checks if there are expired timeouts.

3. Timeout latency (L)

The tick timer implementation for some operating system adapters is synchronous (using a call to `sleep(interval)`). This means that there is a built-in *latency* (the time spent processing the expired timeouts). This latency can be significant when the timeout is very long (involving many timer ticks).

The following formula determines when a timeout will expire:

$$[(T+L) - R, (T+L) + R]$$

Note: If you use triggered operations and events (including timeouts) in the same statechart and the triggered operation can be called from an object running in a thread other than the event consumption thread, it might lead to a race situation. To prevent the race, make the triggered operations guarded (which will also prevent the race with timeouts).

Null Transitions

In some cases, it is useful to use a transition to leave a state without using a trigger. The following are three examples of such cases:

- ◆ When a state tries to allocate a resource that might not be available
- ◆ When you want to branch according to some entry action
- ◆ When you have a join transition

You can accomplish this with a null transition. A *null transition* is any transition without a trigger (event or timeout). Null transitions can have guards (for example, `[x == 5]`). The run-to-completion semantics of the Rhapsody framework checks for an infinite (run-time) loop of null transitions, which might otherwise be difficult to detect.

You can modify the `maxNullSteps` number and recompile the framework if you need to change the number. Refer to the *C++ Execution Framework Reference Guide* for more information.

Guards

A *guard* is a conditional expression that is evaluated based on object attributes and event data. Rhapsody does not interpret guards. They are host-language expressions, or simply code chunks, that must resolve to either a Boolean or an integer value that can be tested. Otherwise, the statechart code generated for guards will not compile.

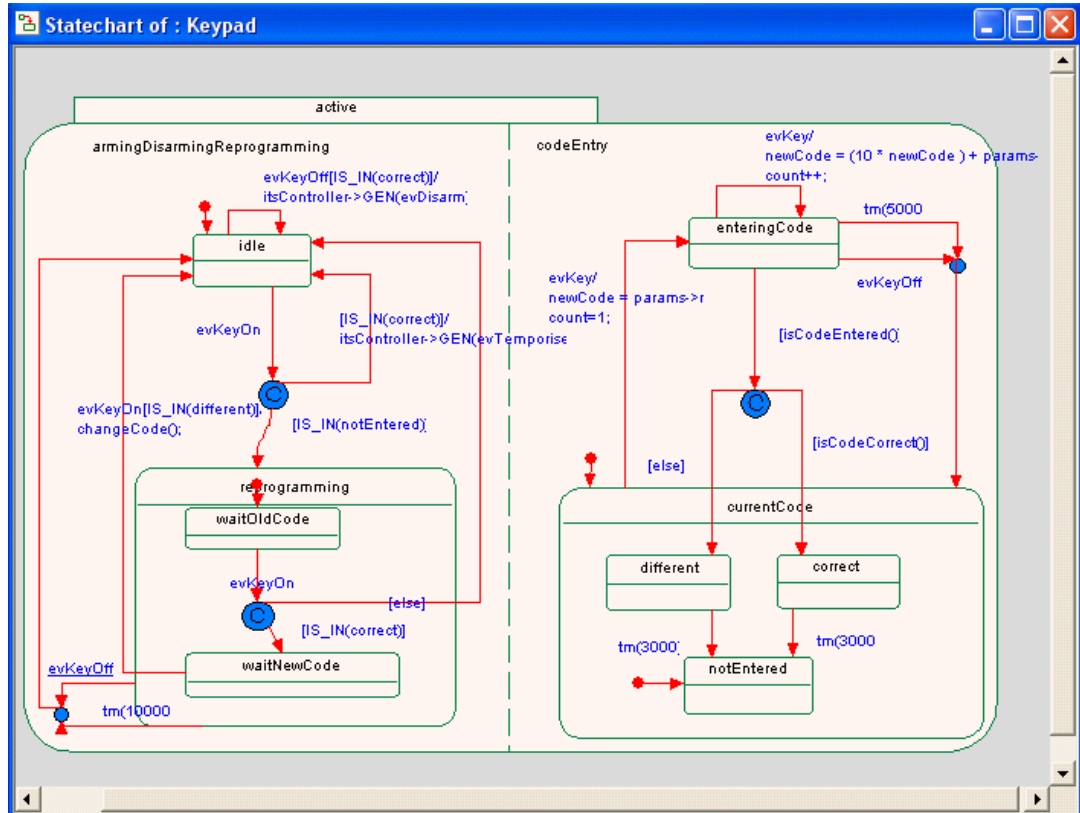
The following is an example of a transition label that consists of a guard and an action that uses the GEN macro to generate an event:

```
[x > 7]/controller->GEN(A7Failures)
```

A transition can consist of only a guard. The low-to-high transition of the condition (or Boolean value) is considered to be the triggering event. For example, the following guard is a valid transition label:

```
[x > 7]
```

During animation, all guards without triggers are tested every time an event happens. The following statechart uses several guards without transitions.



This statechart is for the keypad of a home alarm system. When the keypad of the alarm system is in the idle state, you can enter a code to arm the alarm before leaving the house. After entering the code, you press the On button to turn the alarm on. Pressing the On button issues an `evKeyOn` event. Each time this event occurs, the state machine evaluates the two guards that come after the condition connector—`[IS_IN(correct)]` or `[IS_IN(notEntered)]`—and follows the path of the one that evaluates to true.

By using an animated sequence diagram, you can see when a guard is tested. If you want to test a condition more frequently or at a more regular interval than whenever an event occurs, you can create a polling mechanism. To do this, create a short timeout transition from the state to itself so the guard is evaluated on at least these occasions. Alternatively, you can poll using another object and replace the guard in the current statechart with an event signaled from the polling object.

Note

Using guards that cause side-effects is not recommended, because it might cause problems in the application.

Actions

An *action expression* is a sequence of statements. Like guards, actions are uninterpreted code chunks based on object attributes and event data.

There is no need to add a semicolon to the last statement; Rhapsody adds one for you. Therefore, there is no need for any semicolon if there is only one statement.

The following is an example of a transition label consisting of a trigger and an action sequence with more than one step:

```
e1/x=1;y=2 // comments are allowed
```

Action expressions must be syntactically legal in the programming language. This means they must be written within the context of the owner class (the one that owns the statechart being described). Therefore, all identifiers must be one of the following:

- ◆ Class (or superclass) attributes or operations
- ◆ Role names
- ◆ Global variables known to the class

Any other identifier will cause failures at compile time.

Actions can reference the parameters of an event or operation as defined by the transition context, using the pseudo-variable `params->`. See also [Message Parameters](#).

Events and Operations

Events inherit from the `OMEvent` abstract class defined in the Rhapsody framework. They are abstract entities that do not exist in C++ or other object-oriented programming languages. They are framework-based, and you can implement them in various ways.

In Rhapsody, both events and messages create operations for a class. You can edit the operations created as a result of messages, but you cannot modify any event handlers.

Events and operations relate statecharts to the rest of the model by triggering transitions. Operations specified by a statechart are called triggered operations (as opposed to operations specified in OMDs, called primitive operations).

Events facilitate asynchronous collaborations and operations facilitate synchronous collaborations. Triggered operations have a return type and reply. Triggered operations have a higher priority than events.

In the rest of this guide, the term *message* means either an event or an operation.

Statecharts can react to operations and events that are part of the interface of a reactive class. Using a message as a trigger in a statechart to transition from state *S1* to state *S2* means that if the object is in *S1* when it receives the message, it transitions to *S2*.

Events that do not trigger an enabled transition are ignored and discarded. If the object happens to be in state *S3* when it receives the message and *S3* does not reference the message, it ignores the message.

See [Events](#) for more information.

Sending Events Across Address Spaces

Rhapsody allows you to send events to reactive instances in different address spaces.

This feature applies to multiple address spaces on the same computer. It is not possible to send events to reactive instances on a different computer.

This feature can be used with the following target environments:

- ◆ INTEGRITY5
- ◆ VxWorks6.2diab_RTP
- ◆ VxWorks6.2gnu_RTP

Note

Currently, the multiple address space feature applies only to Rhapsody in C.

Use of this feature requires:

- ◆ Setting a number of properties
- ◆ Calling a different function than that used for sending events within the same address space

Setting Properties

To allow use of the multiple address space feature, different code generation settings are required. These settings are controlled by the following property:

- ◆ `C.CG::Configuration::MultipleAddressSpaces`
When this boolean property is set to `Checked`, Rhapsody uses the code generation settings required for use of the multiple address space feature. *The default value of this property is `Cleared`, so you must change the value to enable this feature.*

In order to be able to receive events from other address spaces, the reactive object must publish the name by which it will be identified. The following two properties, set at the class level, are used for this purpose:

- ◆ `C.CG::Class::PublishedName`
This is the name that will be used to identify the reactive object in order to send a distributed event to it.
If there is only one reactive instance of the class, the value of this property is used to identify the object.
If there is more than one reactive instance of the class, each named explicitly, the name used to identify the reactive object will be the name that you have given to the object, and not the property value.
In the case of multiplicity, where the objects are not named explicitly, the name used to

identify the reactive object will be the published name + the index of the object, for example, if the value of the property `PublishedName` is `truck`, then the objects would be identified by `truck[0]`, `truck[1]`...

- ◆ `C.CG::Class::PublishInstance`
This boolean property indicates whether or not the object should be published as a reactive instance that is capable of receiving distributed events.

In addition, the following property, which is set at the configuration level, allows you to specify a specific target address space when sending events, as described in [API for Sending Events across Address Spaces](#):

- ◆ `C.CG::Configuration::AddressSpaceName`
When you want to send an event to a reactive object in a specific address space, you specify the address space by using the value of this property as a prefix, using the format *addressSpaceName::publishedNameOfReactiveObject*. The default value of this property is the name of the relevant component.

If the events to be sent across address spaces have no arguments or only primitive types as arguments, such as integers or chars, it is sufficient to just set the above properties. However, if the events to be sent include objects as arguments, you must also set the following properties at the event level:

- ◆ `C.CG::Event::SerializationFunction`
Name of user-provided serialization function to use
- ◆ `C.CG::Event::UnserializationFunction`
Name of user-provided unserialization function to use

For details regarding the required structure for these two user-provided functions, see [Functions for Serialization/Unserialization](#).

API for Sending Events across Address Spaces

When sending events to reactive objects in different address spaces, the function `RidSendRemoteEvent` must be used (and not the standard event generation macro `RiCGEN`):

```
RiCBoolean RidSendRemoteEvent (const RhpString strReactiveName, struct RiCEvent* const ev, const RhpPositive eventSize);
```

`strReactiveName` - the published name of the destination reactive object

`ev` - pointer to the event to send

`eventSize` - the size of the event to send

Note

When providing the `strReactiveName` parameter for the function `RidSendRemoteEvent`, you have the option of indicating which address space contains

the target object, using the format *addressSpaceName::publishedNameOfReactiveObject*. This allows you to have objects with the same name in multiple address spaces and still have the event sent to the appropriate object.

When using this option, the name you use for the address space is the value of the property `C_CG::Configuration::AddressSpaceName`, described in [Setting Properties](#).

For convenience, Rhapsody includes a macro named `RiCGENREMOTE`, which calls the function `RidSendRemoteEvent`:

```
RiCGENREMOTE ([string - the published name of the destination reactive object], [type of event with parameters in parentheses])
```

For example:

```
RiCGENREMOTE("destinationObject", Fstarted());
```

Functions for Serialization/Unserialization

If the events to be sent across address spaces have no arguments or only primitive types as arguments, such as integers or chars, you just have to call the function `RidSendRemoteEvent`. However, if the events to be sent include objects as arguments, you must also provide two functions - one for serializing and one for unserializing the event arguments:

Serialization Function

```
RhpAddress evStartSerialize(struct RiCEvent* const ev, const RhpAddress buffer, RhpPositive bufferSize, RhpPositive* resultBufferSize);
```

return value - pointer to the serialized event

`ev` - pointer of the event to be serialized

`buffer` - a local buffer that can be used for storing the serialized event (the user can allocate their own buffer instead)

`bufferSize` - the size in bytes of the parameter `buffer`

`resultBufferSize` - pointer for storing the size of the returned serialized event

Unserialization Function

```
RiCEvent* evStartUnserialize(RhpAddress const serializedBuffer, RhpPositive serializedBufferSize);
```

return value - pointer to the unserialized event

`serializedBuffer` - pointer to the serialized buffer

`serializedBufferSize` - the size of the parameter `serializedBuffer`

Example of Serialization/Unserialization Functions

The example refers to the event `evStart`, which is defined as follows:


```
struct evStart {
    RiCEvent ric_event;
    /** User explicit entries */
    char* msg;
};

RhpAddress evStartSerialize(struct RiCEvent* const ev, const RhpAddress
buffer, RhpPositive bufferSize, RhpPositive* resultBufferSize)
{
    evStart* castedEv = (evStart*)ev;
    RhpPositive msgLength = strlen(castedEv->msg);
    /* Testing the size of the message parameter against the size of local
buffer */
    if (bufferSize <= msgLength)
    {
        /* buffer too small - serialization is aborted */
        return NULL;
    }
    /* copy the message string + the null terminating */
    memcpy(buffer, castedEv->msg, msgLength + 1);
    *resultBufferSize = msgLength + 1;
    return buffer;
}
```

The function below uses a local buffer called `receivedBuffer` to store the string of the event `evStart` which was passed as a parameter.

```
RiCEvent* evStartUnserialize(RhpAddress const serializedBuffer,
RhpPositive serializedBufferSize) {
    /* copy the message to a local buffer */
    memcpy(receivedBuffer, serializedBuffer, serializedBufferSize);
    return (RiCEvent*)RiC_Create_evStart(receivedBuffer);
}
```


Send Action State Elements

The Send Action State icon  can be used in statecharts, activity diagrams, and flow charts to represent the sending events to external entities.

The Send Action State element can be used to specify the following:

- ◆ Event to send
- ◆ Event target
- ◆ Values for event arguments

This is a language-independent element, which is translated into the relevant implementation language during code generation.

Note

Code can be generated for Send Action State elements in C, C++, and Java.

Defining Send Action State Elements

To define the element, provide the following information in the Features dialog box:

- ◆ Using the *Target* drop-down, select the object that is to receive the event.
- ◆ Using the *Event* drop-down, select the event that should be sent.
- ◆ Provide values for the event arguments by selecting the argument in the argument list and clicking the *Value* column.

Note

In cases where there are a number of objects based on the same class, you need to provide additional information after selecting the target from the drop-down list. For cases of simple multiplicity, you must provide the array index to specify the object that receives the event. In the case of qualified associations, you will have to provide the qualifier value for the object that is to receive the event.

The *Preview* text box displays the text that will be displayed on the element if you select full notation as the display option to use.

The target list includes all objects known to the statechart's class. You can choose the name of the target object, or the name of a port on the target object.

You can click the button next to the *Target* drop-down list to open the features dialog of the relationship with the target object. Similarly, you can click the button next to the *Event* drop-down list to open the features dialog for the selected event.

Display Options

The display options for the Send Action State element allow you to display a full notation, such as “Reset (false) to p1” or a short notation, such as “Reset.” Full notation includes the event name, the values for the event arguments, and the name of the target. Short notation includes only the event name.

Graphical Behavior

In terms of its behavior in the graphic editors, Send Action State elements are connected to states in the statechart with transitions.

While the graphical behavior of Send Action State elements is similar to that of states, it should be remembered that semantically these elements are not states. For example, you cannot put a condition on the transition out of a Send Action State element (it is an automatic transition).

Code Generation

Code can be generated for Send Action State elements in C, C++, and Java.

For each language, code generation for this element is determined by the following properties

- ◆ `CG::Framework::EventGenerationPattern` - general format
- ◆ `CG::Framework::EventToPortGenerationPattern` - used when sending event to a port

Note

Rhapsody does not support roundtripping for Send Action State elements.

Default Transitions

A *default transition* leads into the state (or substate of an Or state, or component of an And state) that should be entered by default. Each Or state must designate one of its substates as the default for that state. The default state is indicated using a default transition, of which there can be only one per Or state. The default transition target should be a substate of the Or state to which it belongs.

The default transition cannot have a trigger or a guard, although it can have an action. It might connect to a condition connector following which there might be guards.

Each state can have the following properties:

- ◆ **Entry action**—An expression executed upon entrance to the state (uninterpreted by Rhapsody). Note that an uninterpreted expression is resolved by the compiler, not by the tool.
- ◆ **Exit action**—An expression executed upon exit from the state (uninterpreted by Rhapsody).
- ◆ **Static reactions**—Actions performed inside a state in response to a triggering event/operation. The reaction can be guarded by a condition. A static reaction does not change the active state configuration.

The state executes static reactions if:

- The state is part of the active configuration.
 - The trigger and guard are satisfied.
 - A lower-level state has not already responded to the trigger.
 - There is no enabled transition that causes the state to be exited.
- ◆ **Default entry**
 - ◆ **History**

Note

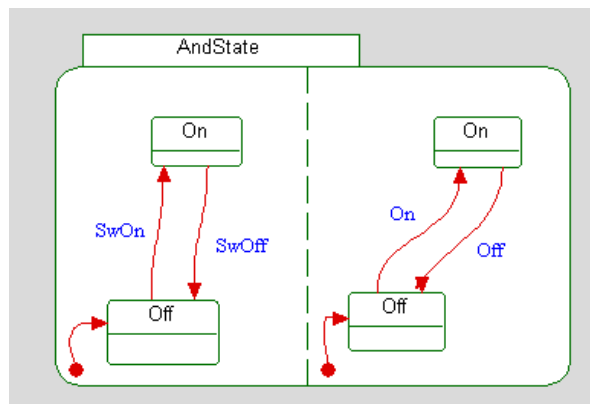


When a state contains an entry action, exit action, or static reaction, an icon appears in the top-right corner of the state. This icon can be used to toggle the display of these actions in the state.


And Lines

An And line is a dotted line that separates the orthogonal components of an And state. There can be two or more orthogonal components in a given And state and each behaves independently of the others. If the system is in an And state, it is also simultaneously in a substate of each orthogonal component.

The following figure shows an And line.



To draw an And line to divide a state into substates, follow these steps:

1. Click the **And line**  icon in the **Drawing** toolbar.
2. Click in the middle of the upper edge of the state to anchor the start of the And line.
3. Move the cursor down to the bottom edge of the state and click to anchor the end of the And line. Rhapsody draws a dotted line that divides the state into two halves (orthogonal states), as shown in the following figure.

Note that the state label, which used to be inside the state, has moved outside into a tab-like rectangle.

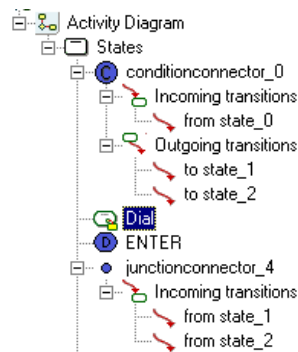
Connectors

Rhapsody supports the following connectors:

- ◆ [Condition Connectors](#)
- ◆ [History Connectors](#)
- ◆ [Junction Connectors](#)
- ◆ [Diagram Connectors](#)
- ◆ [Termination Connectors](#)
- ◆ [EnterExit Points](#)

Rhapsody includes connector information for diagram, condition, and EnterExit points in its repository (core). This means that:

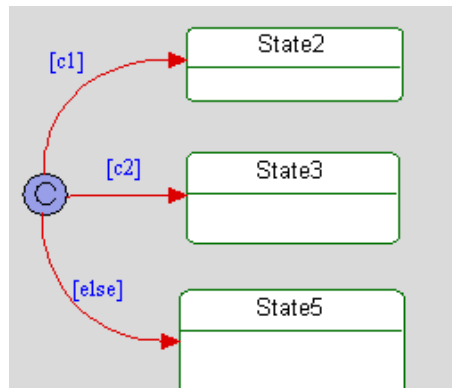
- ◆ Semantic checks are done by the standard core functions.
- ◆ Statechart inheritance is core-oriented, not graphics-oriented.
- ◆ Code is generated for these connectors.
- ◆ The Undo operation supports all connector actions.
- ◆ Rhapsody EnterExit points are now UML-compliant.
- ◆ Reports include information on diagram connectors, condition connectors, and EnterExit points.
- ◆ Diagram connectors, condition connectors, and EnterExit points are displayed in the browser, as shown in the following figure.



Condition Connectors

Condition connectors split a single segment into several branches. Branches are labeled with guards that determine which branch is active.

The following figure shows a condition connector.



The following rules apply to condition connectors and branches:

- ◆ Branches cannot contain triggers.
- ◆ You can nest branching segments. This means that a branching segment can enter another condition connector.
- ◆ A condition connector can have only one entering transition.
- ◆ The branching tree should not have cycles.

Else Branches

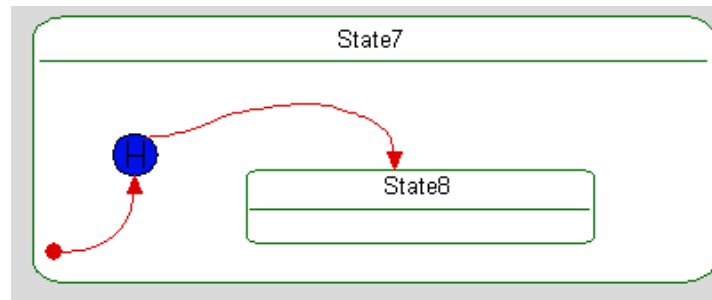
A guard called `[else]` is active if all the guards on the other branches are false. Each condition connector can have only one else branch.

The semantics of an else branch are similar to a structured if-then-else statement.

History Connectors

History connectors store the most recent active configuration of a state and its substates. Once an object is created, it is associated with an active state's configuration, starting in the initial configuration, and evolving as the statechart responds to messages.

The following figure shows a history connector.



When a transition is attached to a history connector and that transition is triggered, the state containing the history connector recalls its last active configuration. A state can have a single history connector.

Transitions from a history connector are constrained to a destination on the same level as the history connector.

Note

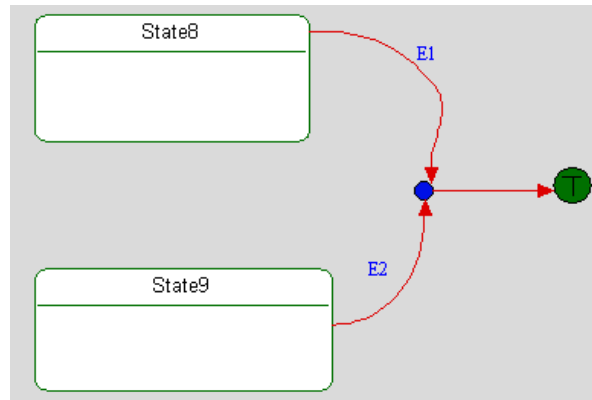
Do not put more than one history connector in a state. Rhapsody allows you to draw more than one history connector in a state; however, the code generator does not support this.

A state might have a history property used for recalling the recent active configuration of the state and its substates. Transitioning into a history connector associated with the state recalls the last active configuration.

A transition originating from the history connector designates the history default state. The default history state is taken if no history existed prior to the history enter.

Junction Connectors

A *junction connector* combines several segments into one outgoing segment, as shown in the following figure.



This means that segments share the same line and a common transition suffix. The segments end up sharing the same transition line.

Diagram Connectors

A *diagram connector* functions similarly to a junction connector in that it joins several segments in the same statechart. Diagram connectors enable you to jump to different parts of a diagram without drawing spaghetti transitions. This helps avoid cluttering the statechart. The jump is defined by matching names on the source and target diagram connectors.

Note

You can rename diagram connectors, and the checks are performed during code generation.

Diagram connectors should either have input transitions or a single outgoing transition. A statechart can have at most one target diagram connector of each label, but it can have several source diagram connectors with the same label.

During code generation, Rhapsody flattens all junctions and diagram connectors by merging the common suffix to each segment entering the connector.

In both diagram and junction connectors, a label that belongs to an incoming segment is shared and duplicated during code generation among outgoing segments of that connector. Rhapsody merges the guards (conjunction), then concatenates the actions.

Note

Both incoming and outgoing transitions cannot have labels. If you label the incoming transitions, do *not* label the outgoing transition because its label will override the label of the incoming transition and negate any action or trigger associated with the incoming transition.

Diagram connectors connect different sections of the same statechart, whereas EnterExit points connect different statecharts. See [EnterExit Points](#).

Termination Connectors

The *termination connector* is the suicide or self-destruct connector. If a transition to a termination connector is taken, the instance deletes itself. A termination connector cannot have an outgoing transition.

EnterExit Points

EnterExit points are used to represent the entry to / exit from sub-statecharts.

At the level of the parent state, these points represent entry to / exit from the various contained substates without revealing any information about the specific substate that the transition connects to.

At the level of the sub-statechart, these points represent the entries to / exits from the parent state vis-a-vis the other elements in the statechart.

When you create a sub-statechart from a parent that contains deep transitions (that is, transitions entering one of the substates), EnterExit points are automatically created on the borders of the parent state in both the sub-statechart and the original statechart.

Once the sub-statechart has been created, you can add additional deep transitions as follows:

1. Add an EnterExit point to the parent state.
2. Add a corresponding EnterExit point in the sub-statechart (manually or using the Update feature - see [Updating EnterExit Points](#)).
3. Draw a transition from the EnterExit point to the relevant substate.

Updating EnterExit Points

If you would like to add additional EnterExit points after creating a sub-statechart, you can add them manually to both the parent state in the original statechart and the parent state in the sub-statechart.

Alternatively, you can have Rhapsody automatically update the EnterExit points:

1. Add one or more EnterExit points to the parent state in either the original statechart or the sub-statechart.
2. Go to the second statechart, and right-click to open the context menu.
3. From the context menu, select **Update EnterExit Points**.

Submachines

Submachines enable you to manage the complexity of large statecharts by decomposition. The original statechart is called the *parent*, whereas the decomposed part is called the *submachine*.

Creating a Submachine

You can create a submachine from a complex state using either the Edit menu or the pop-up menu for the state.

To create a submachine, follow these steps:

1. In a statechart, right-click a state.
2. From the pop-up menu, select **Create Sub-Statechart**. Rhapsody creates a submachine called `<class>.<state>`, which is a new statechart consisting of the submachine state and its contents.

If you decompose the `doorClosed` state into a submachine, Rhapsody creates a new submachine.

Actions and reactions move into the top state of the submachine if the transition goes to the submachine state, and inside the submachine if the transition goes into the nested part.

Note

You cannot create submachines of inherited states. The workaround is to add a dummy state as a child of the inherited state and make that the submachine state.

Opening a Submachine or Parent Statechart

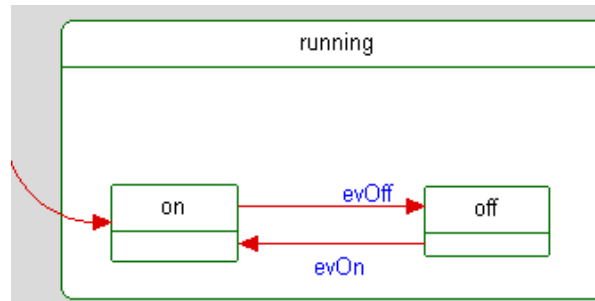
To open a submachine, right-click the submachine state in the parent statechart and select **Open Sub-Statechart** from the pop-up menu.

Similarly, to open a parent statechart from a submachine, right-click the top state and select **Open Parent Statechart** from the pop-up menu.

Creating a Deep Transition

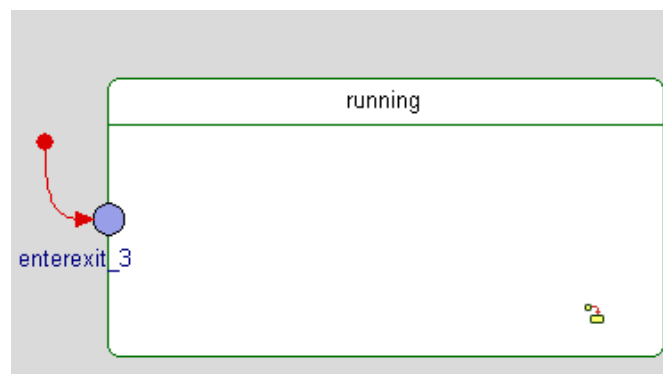
A *deep transition* is a cross-chart transition—for example, from a parent statechart into a submachine, or vice versa. When you create a submachine, deep transitions are automatically split via substates.

Consider the following example:



This statechart has a deep transition that crosses the edge of a parent state (running) and leads into a nested state (on).

If you make a submachine of the running state, the deep transition is automatically split via matching EnterExit points created in the parent statechart and submachine, as shown in this example.



Merging a Sub-Statechart into its Parent Statechart

To merge the contents of a sub-statechart into its parent statechart:

1. Select the parent state in the original statechart.
2. Right-click the state and select **Merge Back Sub-Statechart**.

Statechart Semantics

The following sections describe the object-oriented interpretation of statecharts.

Single Message Run-to-Completion Processing

Rhapsody assumes that statecharts react to a single message applied by some external actor to the statechart. The external actor can be either the system event queue or another object.

Message processing by a statechart is partitioned into steps. In each step, a message is dispatched to the statechart for processing.

Once a message is dispatched, it might enable transitions triggered by the message. Each orthogonal component can fire one transition at most as a result of the message dispatch. Conflicting transitions will not fire in the same step.

The order in which selected transitions fire is not defined. It is based on an arbitrary traversal that is not explicitly defined by the statechart.

Each component can execute one transition as a result of the message. Once all components complete executing the transition, the message is said to be consumed, and the step terminates.

After reacting to a message, the statechart might reach a state configuration in which some of the states have outgoing, enabled null transitions (transient configurations). In this case, further steps need to be taken until the statechart reaches a stable state configuration (no more transitions are enabled). Null transitions are triggered by null events, which are dispatched to the statechart whenever a transient-configuration is encountered. Null events are dispatched in a series of steps until a stable configuration is reached. Once a stable configuration is reached, the reaction to the message is completed, control returns to the dispatcher, and new messages can be dispatched.

Note

Theoretically, it is possible that the statechart will never reach a stable configuration. The practical solution is to set a limit to the maximum number of steps allowed for a statechart to reach a stable configuration. In the current implementation, reaching the maximum number of steps is treated as if the message processing has been completed.

Enabled Transitions

A transition is enabled if:

- ◆ The trigger matches the message posted to the statechart. (Null triggers match the null event.)
- ◆ There is a path from the source to the target states where all the guards are satisfied (evaluate to true).

Note

Guards are evaluated *before* invoking any action related to the transition.

Because guards are not interpreted, their evaluation might include expressions that cause side effects. Avoid creating guards that might cause side effects. Guard evaluation strategy is intentionally undefined as to when guards are evaluated and in which order.

Transition Selection

Transition selection specifies which subset of enabled transitions should fire. Two factors are considered:

- ◆ Conflicts
- ◆ Priorities

Conflicts

Two transitions are said to *conflict* if both cause the same state to exit. Only orthogonal or independent transitions fire simultaneously. This means that interleaved execution causes equivalent results. Disjoint exit states are a satisfactory condition for equivalent results.

Note: With regard to conflicts, static reactions are treated as transitions that exit and enter the state on which they are defined.

Priorities

Priorities resolve some, but not all, transition conflicts. Rhapsody uses state hierarchies to define priorities among conflicting transitions. However, lower-level (nested) states can override behaviors, thus implying higher priority.

A transition's priority is based on its source state. Priorities are assigned to join transitions based on their lower source state.

For example, if transition t_1 has a source state of s_1 and transition t_2 has a source state of s_2 ,

- ◆ If state s_1 is a descendant of state s_2 , t_1 has a higher priority than t_2 .

- ◆ If states s_1 and s_2 are not hierarchically related, relative priorities between t_1 and t_2 are undefined.

Rhapsody does not define a priority with regard to events and transitions other than arrival order. If two transitions within the same orthogonal component are both enabled (ready to fire), as can happen with non-orthogonal guards, only one of them will actually fire, but statecharts do not specify which one it will be.

Transition Selection Algorithm

The set of transitions to fire satisfies the following conditions:

- ◆ All transitions must be enabled.
- ◆ Any transition without conflicts will fire.
- ◆ If a priority is defined between transitions, the transitions with lower priority will not fire.
- ◆ In any set of conflicting transitions, one transition is selected to fire. In cases where conflicts are not resolved by priorities, the selected transition is arbitrary.

The above definition of the selection set is not imperative, but implementing a selection algorithm is done by a straightforward traversal of the active state configuration.

Active states are traversed bottom-up where transitions related to each are evaluated. This traversal guarantees that the priority principle is not violated. The only issue is resolving transition conflicts across orthogonal states. This is solved by “locking” each And state once a transition is fired inside one of its components. The bottom-up traversal and the And state locking together guarantee a proper selection set.

Transition Execution

Once a transition is enabled and selected to fire, there is an implied action sequencing:

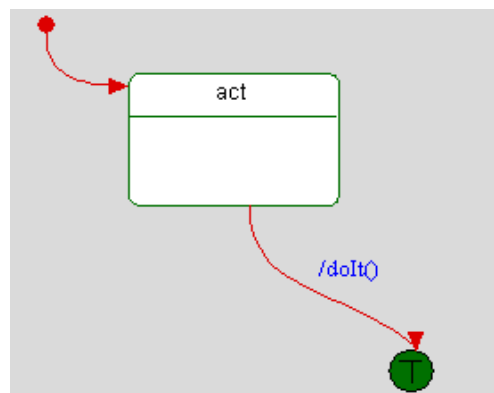
- ◆ States are exited and their exit actions are executed, where deeper states are exited before their parent states. In case of orthogonal components, the order among orthogonal siblings is undetermined.
- ◆ Actions sequencing follow the direction of the transition. The closer the action to the source state, the earlier it is evaluated.
- ◆ Target states are entered and their entry actions are executed, where parent states are entered before substates. In the case of orthogonal components, the entry order is undetermined.

Active Classes without Statecharts

Normally, active classes (threads) must also be reactive (have statecharts). However, you might have tasks that have no state memory. The workaround of defining a dummy (empty) statechart is not entirely acceptable because such an active object uses statechart behavior to process events. It is, however, possible to achieve the same effect by setting the class to active, defining an empty statechart, then overriding the default behavior by defining an operation named `takeEvent()` for the class and adding the desired behavior to this operation. This method allows you to benefit from visual debugging, using the event queue, and so on.

Single-Action Statecharts

Rhapsody cannot interpret simple statecharts that execute a single action and then terminate. For example, if you represent a task as an active class with a simple statechart that essentially executes a single action and terminates, you might be tempted to draw your statechart, as shown in this example.



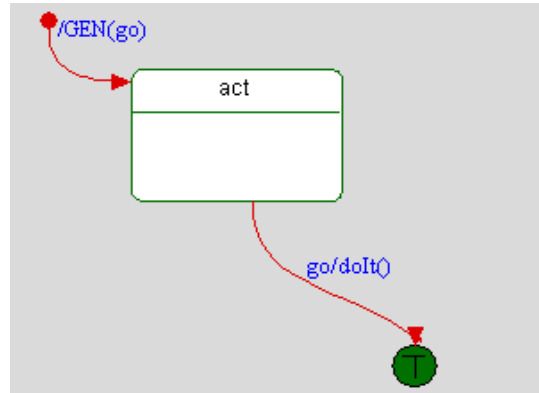
In this diagram, `doIt()` represents the action that needs to be spawned.

This statechart has two problems:

- ◆ The Rhapsody framework does not allow an active instance to terminate on the initial run-to-completion (RTC) step. In other words, the `startBehavior()` call cannot end with a destroyed object.
- ◆ The `startBehavior()` call executes the initial step on the creator thread, not as part of the instance thread. The instance thread processes events following the initial step. In this statechart, the `doIt()` operation is executed on the creator thread, which is probably not what was expected.

The workaround is to create a dummy action on the default transition that leads into a transition. This action can run on the instance thread and thus terminate normally.

For example, the following statechart postpones the execution of the action until the thread is ready to process it.



Inherited Statecharts

Statechart inheritance begins when a class derives from a superclass that has a statechart. The statechart of the subclass, the inherited statechart, is initially a clone of that of the superclass. With the exception of the items listed below, you can add things to an inherited statechart to override behavior in the inherited class.

You *cannot* make the following changes to items in the statechart of a subclass:

- ◆ Change the source of a transition.
- ◆ Change the triggers (events or triggered operations).
- ◆ Delete or rename a state.
- ◆ Draw a state around an existing state.

You *can* make the following changes to items in the statechart of a subclass:

- ◆ Change anything that does not affect the model, such as moving things in the diagram without actually editing.
- ◆ Add objects to a state.
- ◆ Add more states, but not re-parent states.
- ◆ Attach a transition to a different target.

An inherited statechart consists of all the items inherited from the superclass, as well as modified and added elements.

Note

It is possible to inherit statecharts across packages.

If you edit a base statechart, the derived statechart is redrawn only on demand at checks, code generation, report generation, or the opening of a derived statechart.

Types of Inheritance

Each item in the derived statechart can be:

- ◆ **Inherited**—Any modifications to an item in the superclass is applied to the item in the subclass.
- ◆ **Overridden**—Any modifications to an item in the superclass do not apply to the subclass. However, deleting an item from the superclass also deletes the item from the subclass. This is different from C++, for example, where deleting an overridden behavior in the superclass causes the overridden behavior to become a regular item.
- ◆ **Regular**—Regular items are owned by the subclass. The item is not related to the superclass and is not affected by the superclass.

Noting the status of items as inherited, overridden, or regular is crucial both for Rhapsody and the user.

Note

The current implementation of statechart inheritance is restricted to single inheritance. A reactive class can have at most one reactive superclass.

Inheritance Color Coding

Inheritance status is indicated by the following color coding:

- ◆ Inherited items are gray.
- ◆ Regular and overridden items are colored in the usual drawing colors.

Inheritance Rules

Classes with inherited statecharts can reside in different packages. A class with a statechart (reactive class) can inherit from a class without a statechart. Multiple inheritance of reactive classes (with statecharts) is not supported. Derived classes can inherit from multiple primitive classes. Rearranging inheritance hierarchies of reactive classes is not supported.

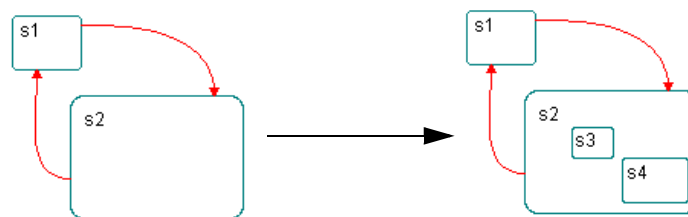
There are different inheritance rules for states, transitions, triggers, guards, and actions.

Rules for States

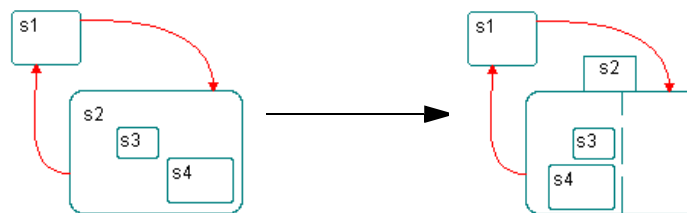
The structure of a state in a subclass should be a refinement of the same state of the superclass. Because of this, state inheritance is strict. All states and their hierarchy are inherited by the statechart of the subclass.

You can add states to the derived statechart, as long as they do not violate the hierarchy in the statechart of the superclass. In practice, this means that a regular state cannot contain inherited substates.

In the following example, the leaf state s_2 was refined and became an Or state. The states s_1 and s_2 on the right are the inherited states.



You can add And lines to inherited states (adding components). If you convert an inherited Or state into an And state, the Or state becomes an And state, and one of the components contains its substates. This is an exception to the previous rule, where the state hierarchy is modified by introducing an orthogonal component. The component that re-parents the substates is designated as “main.” In the following example, s_2 becomes an And state. The component containing s_3 and s_4 is the main component. The component’s name is the same as the And state’s name.



Note the following:

- ◆ You cannot rename inherited states in the derived statechart.
- ◆ You cannot delete inherited states from a derived statechart.
- ◆ A state is either inherited or regular—it cannot be overridden.
- ◆ You cannot change state topology by re-parenting.

Rules for Transition Labels

You can modify the labels of derived segments according to the following rules:

- ◆ You cannot modify triggers—they are inherited from the superclass.
- ◆ You can modify actions and guards.
- ◆ You can override a guard, but still get changes on the action.

Modifications to the label of the corresponding segment in the superclass no longer affect the subclass.

Note

The inheritance color coding of the label and the segment are independent. The label can be overridden while the segment is still inherited, and vice versa.

Rules for Entry and Exit Actions

Both entry and exit actions can be overridden by an inherited state. Once they have been modified (modifying the text of the action) by a derived state, they reach an overridden state.

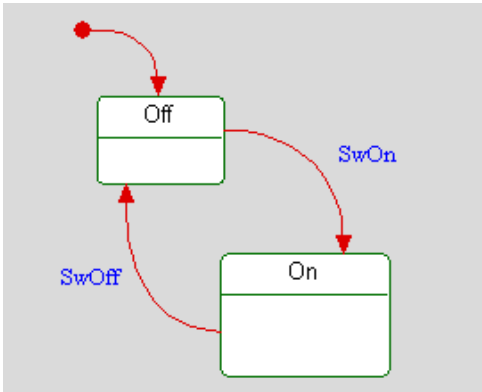
Rules for Static Reactions

Static reactions can be overridden by a derived statechart. Static reactions are designated by their triggers, which cannot be modified in a derived statechart. Therefore, only the guard and action can be modified. A static reaction cannot be deleted by a subclass.

Currently, there is no inheritance color coding for static reactions. In addition, tracing inherited actions between the superclass and the derived statechart is done implicitly by Rhapsody and is not visible.

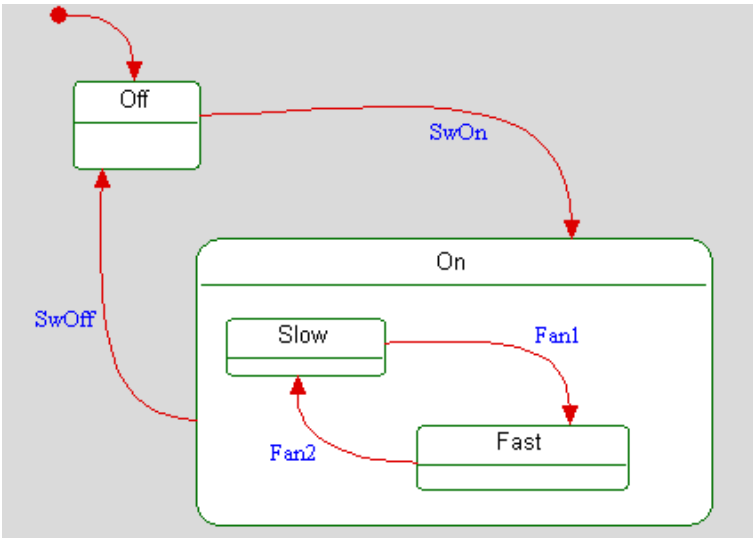
Rules for Connectors

Connectors are always inherited. You cannot modify them or delete them. The following example illustrates statechart inheritance.

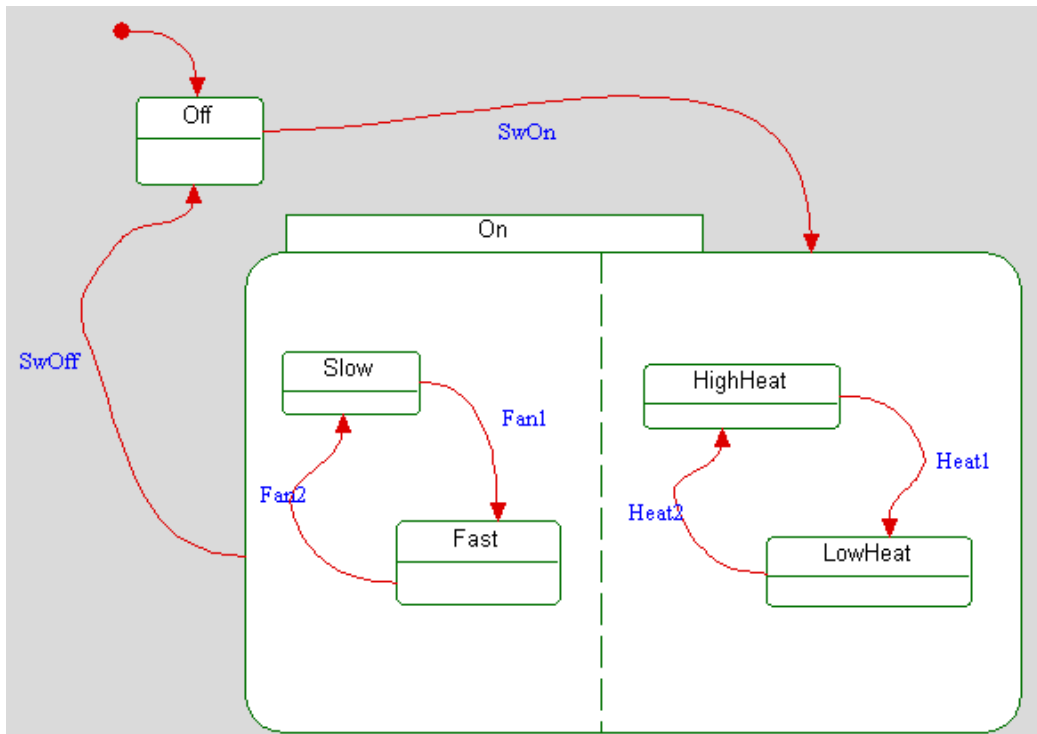


As shown, a basic blower has only On and Off modes.

In a dual-speed blower, the On state is refined to include Fast and Slow modes, as shown in the following figure.



If you make the On state into an And state, you can add a heat mode, as shown.



Overriding Inheritance Rules

To override the inheritance rules of statecharts, right-click in the statechart and select **Override Inheritance** from the pop-up menu.

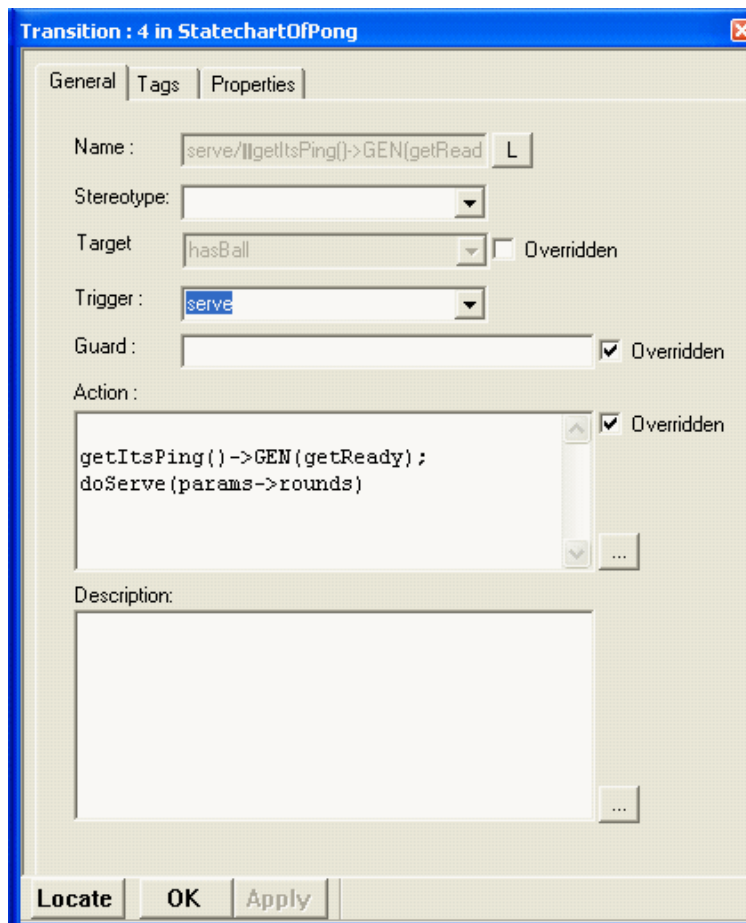
Once you have overridden inheritance, the derived statechart becomes independent from its parent and you can modify it without constraint. In addition, colors are no longer gray—they are the usual statechart colors.

To undo the inheritance override, right-click in the statechart and select **Unoverride Inheritance** from the pop-up menu.

Overriding Textual Information

The Features dialog boxes for textual information in statecharts (state entry and exit actions, and guards and actions for transitions and static reactions) include the **Overridden** check box. By enabling or disabling this check box, you can easily override and unoverride statechart inheritance without actually changing the model. As you toggle the check box on and off, you can view the inherited information in each of the dialog box fields, and can decide whether to apply the information or revert back to the currently overridden information.

For example, the following figure shows the Features dialog box for a transition.



As shown in the figure, the overridden action is as follows:

```
getItsPing()->GEN(getReady());  
doServe(params->rounds)
```

However, if you clear the **Overridden** check box, the revised action is as follows:


```
doServe(params->rounds)
```

To apply the change, click **OK**. The transition changes to `doServe(params->rounds)` and it is displayed in blue text instead of gray because it is no longer overridden.

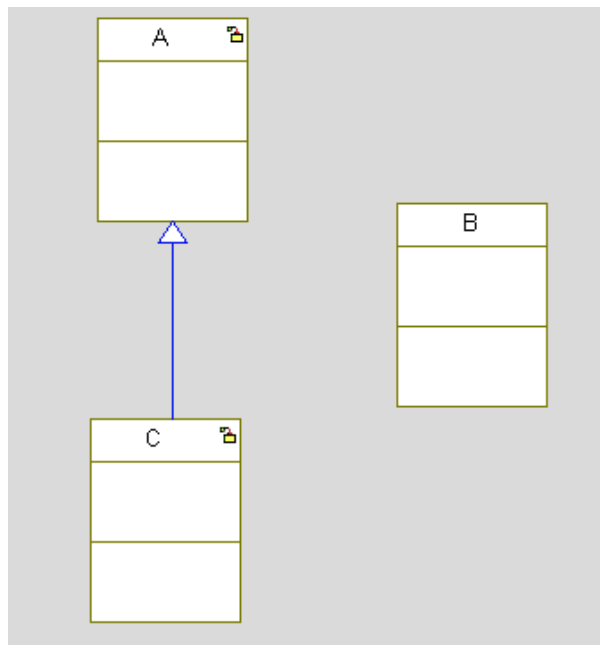
Note that if you override the textual information, the display colors and statechart change as follows:

- ◆ If you unoverride the textual information of a transition or state, the label color reverts to gray.
- ◆ If you unoverride a transition target, the transition color reverts to gray and the graphics are synchronized to the new target.

Refining the Hierarchy of Reactive Classes

You can refine the hierarchy of reactive classes without using overrides and unoverrides—without losing any information.

For example, suppose you have class C inheriting from class A, as shown in the following OMD.



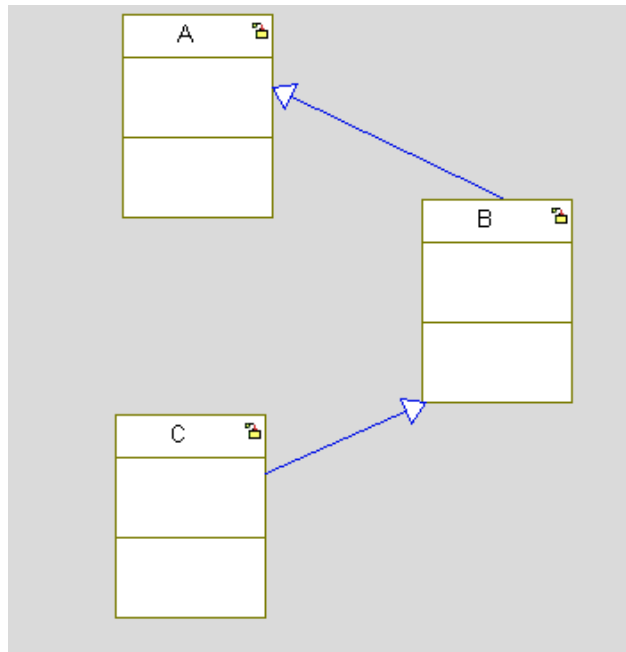
Suppose you want to change the hierarchy so C inherits from B, which in turn inherits from A. Therefore:

- ◆ The statechart that C inherited from A will now be inherited from B; B will inherit its statechart from A.
- ◆ The inheritance between A and C will be deleted.
- ◆ C will not lose any information, because its inherited elements will reference new GUIs.

To make these changes, follow these steps:

1. Using either the browser or the **Drawing** toolbar, create inheritance between B and A.
2. Create inheritance between C and B.
3. Rhapsody displays a dialog box that informs you that you are adding a level of inheritance, and asks you to confirm the deletion of the inheritance between C and A. Click **Yes**.

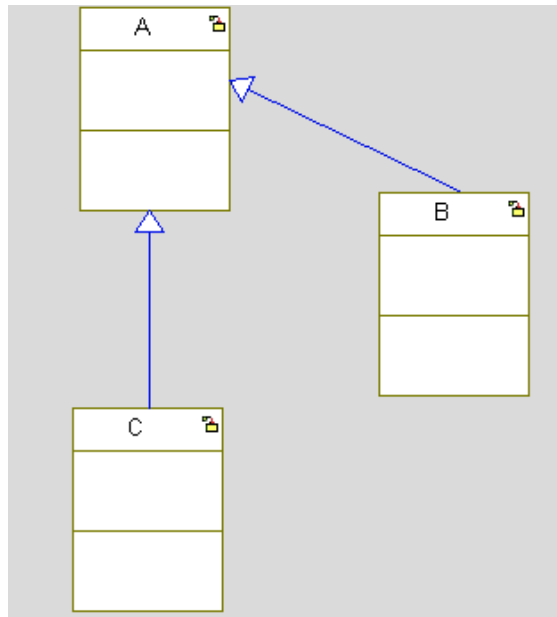
The following figure shows the revised OMD.



Removing a Level of Inheritance

Suppose you now want C to inherit from A instead of B, thereby removing a level of inheritance. When you draw the inheritance between C and A, Rhapsody notifies you that a level of inheritance will be removed and asks for confirmation.

Click **Yes**. The following figure shows the revised OMD.



Inheritance Between Two Reactive Classes

If you try to establish inheritance between two distinct reactive classes (for example, B inherits from A), Rhapsody displays a message stating that the action will result in overriding statechart inheritance. Click **Yes** to override the statechart inheritance.

IS_IN Query

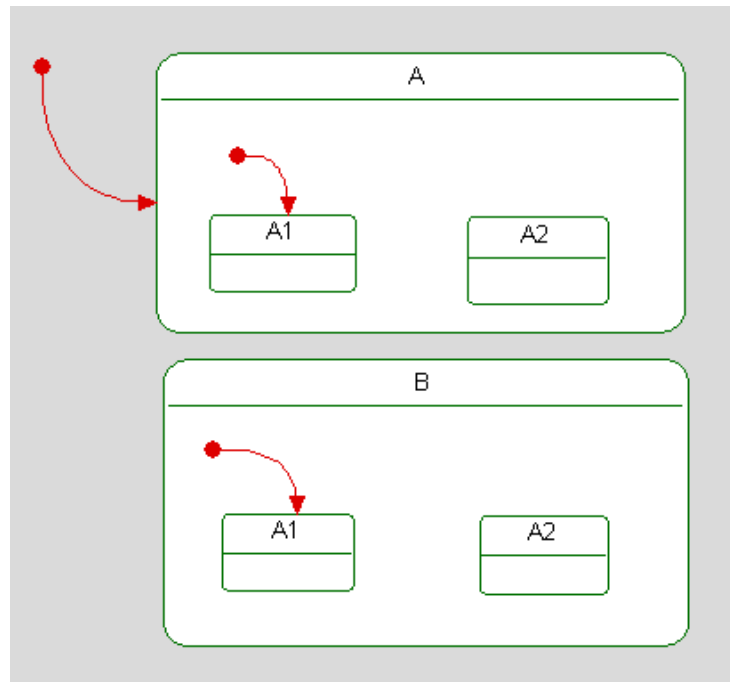
The Rhapsody framework provides the query function `IS_IN(state)`, which returns a true value if the state is in the current active configuration.

Note the following:

- ◆ `IS_IN(state)` returns true if the state is in the active configuration at the beginning of the step. For states entered in a step, `IS_IN(state)` returns false, unless the states are being re-entered.
- ◆ The state name passed to `IS_IN()` is the implementation name, which might be different from the state name if it is not unique.

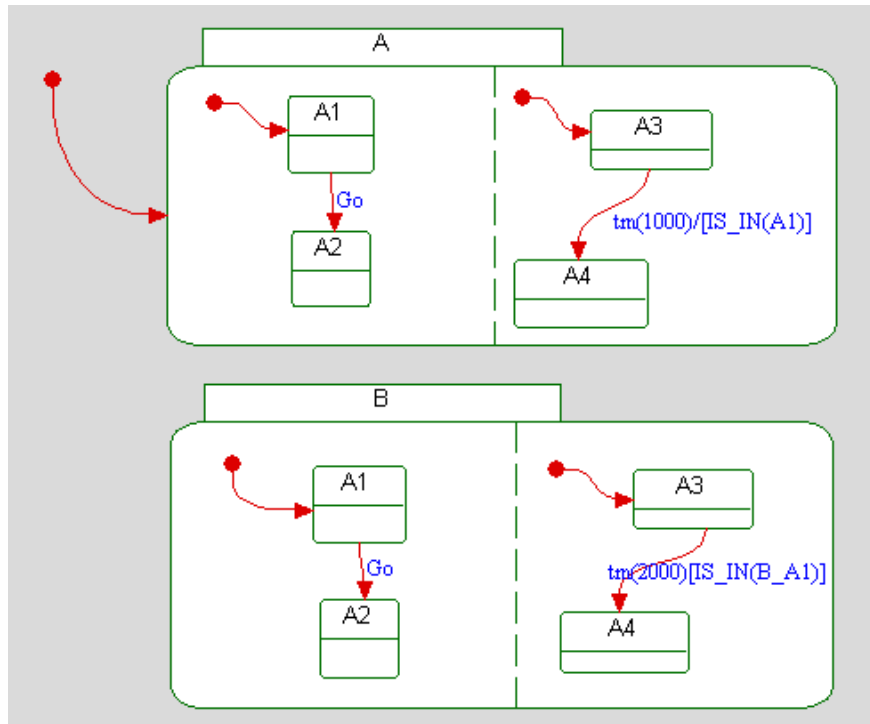
For example, the following state names are generated for the statechart shown in the figure:

```
State* B; State* B_A2; State* B_A1;  
State* A; State* A2; State* A1;
```



The implementation name of A1 in state A is simply A1 because it was drawn first. The implementation name of A1 in state B is B_A1, because it is a duplicate.

In the following statechart, the transition to substate A4 in state A is taken only if the object is still in substate A1 in A after one second. The transition to substate A4 in state B is taken only if the object is still in substate A1 in B after two seconds.



In these two cases, the `IS_IN()` query requires the use of the implementation names `A1` and `B_A1` to differentiate between like-named substates of two different states.

The `IS_IN` macro is called as if it were a member operation of a class. If you want to test the state of another class (for example, a relation), you must use the relation name. For example, if you have a relation to a class `A` called `itsA` and you want to see if `A` is in the idle state, you would use `itsA->IS_IN(idle)` rather than `A->IS_IN(idle)`.

Message Parameters

Message data are formal parameters used within the transition context. By default, if the message is an event, the names of message parameters are the same as the arguments (data members) of the event class.

You reference event arguments in a statechart using the pseudo-variable `params->` with the following syntax:

```
event/params->event_arg1, params->event_arg2
```

Consider a class `Firecracker` that processes an event `discharge`, which has an argument `color`. The argument `color` is of an enumerated type `Colors`, with the possible values `red` (0), `green` (1), or `blue` (2). In the statechart, you would indicate that you want to pass a color to the event when it occurs using the following label on the transition:

```
discharge/params->color
```

When you run the application with animation, you can generate a `discharge` event and pass it the value `red` by typing the following command in the animation command field:

```
Firecracker[0] ->GEN(discharge(red))
```

The application understands `red` as a value being passed to the argument `color` of event `discharge` because of the notation `params->color`. The color `red` is translated to its integer value (0), and the event is entered on the event queue of the main thread as follows:

```
Firecracker[0] ->discharge((int)color = 0)
```

Finally, the event queue processes the event `discharge` with the value `red` passed via `params->`. The `Firecracker` explodes in red and transitions from the `ready` to the `discharged` state.

The way the `params->` mechanism works is as follows: When you create an event and give it arguments, Rhapsody generates an event class (derived from `OMEvent`) with the arguments as its attributes. Code for events is generated in the package file.

The following is sample code for the event `discharge`, which has one argument called `color`. The code was generated in the header file for the `Default` package:

```
//-----
// Default.h
//-----
class discharge;
class Firecracker;
enum Colors {red, green, blue};
class discharge : public OMEvent {
    DECLARE_META_EVENT
    /// User explicit entries    ///
public :
    Colors color;
    /// User implicit entries    ///
public :
    // Constructors and destructors:
    discharge();
    /// Framework entries    ///
public :
    discharge(Colors p_color);
    // This constructor is need in code instrumentation
    discharge(int p_color);
};
```

When the `Firecracker` event queue is ready to take the event `discharge`, it calls `SETPARAMS(discharge)`. `SETPARAMS` is a macro defined in `oxf\state.h` as follows:

```
#define SETPARAMS(type) type *params; params=(type*)event
```

Calling `SETPARAMS(discharge)` allocates a variable `params` of type pointer to an event of type `discharge`. This enables you to use `params->color` in the action part of the transition as a short-hand notation for `discharge->color`.

Modeling Continuous Time Behavior

There are three types of behaviors typical of embedded systems:

- ◆ **Simple**—Implemented in functions and operations
- ◆ **Continuous**—Expressed by items such as PID control loops or digital filters
- ◆ **Reactive**—State-based behavior

Although the Rhapsody GUI directly supports only simple and reactive behaviors, you can implement all three types. To address the continuous time aspects, you can reference or include any code you are currently using to express continuous behavior in any of the operations defined within Rhapsody.

This means that if you are defining your continuous behavior elements manually, you can continue to do so. If you are using another tool to define your continuous behavior and that tool generates code, you can include that code.

Interrupt Handlers

The ability to install an interrupt handler depends on operating system support. Typically, a static function without parameters is installed by passing its address to an operating system operation like `InstallIntHdlr` (operating system-dependent). The static function can be either a special singleton object or a function defined within a package. This operation must use compiler-specific utilities to get to the registers. Eventually, it must return and execute a return from the interrupt instruction.

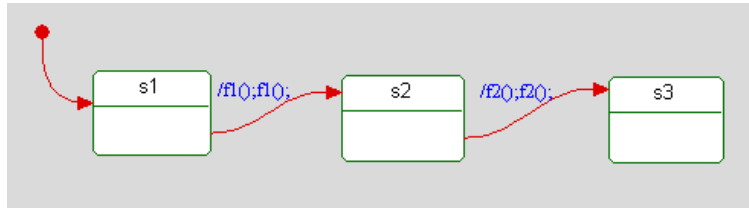
You can pass the data from the interrupt handler to the CPU (assuming that the interrupt handler needs to), in the following ways:

- ◆ Generate an event (using the `GEN()` macro), which then goes via the operating system to the reactive object (which should be in a different thread).
- ◆ Use a rendezvous object with a read/write toggle lock. The interrupt handler checks whether the lock is in the write state, then updates the data and puts the lock in the read state. The reader (in another thread) periodically checks the lock and only reads when it is in the read state. If it is in that state, the reader reads the data and updates the lock to its write state. This can easily be extended to a queue structure.
- ◆ Write the interrupt handler manually outside Rhapsody and send it to the operating system message queue for the target thread. Typically, if the operating system does not support interrupt handlers directly, you store the current handler in your function and write the address of the function in the appropriate place in the interrupt vector table. When the interrupt goes off, either the new function replaces the old interrupt handler (meaning when it is done it simply returns) or it chains to it (calls the original). In any event, when

the interrupt handler is decommissioned, you replace the vector in the vector table with the original address.

Inlining Statechart Code

Consider the following statechart:



When you implement this statechart using the flat scheme, you should expect the following methods to be added to the class and called through a series of a few function calls from `rootState_dispatchEvent()`:

```

int MyClass::s2TakeNull() {
    int res = eventNotConsumed;
    s2_exit();
    //#[ transition 2
    f2();
    f2();
    //#]
    s3_entDef();
    res = eventConsumed;
    return res;
};

int MyClass::s1TakeNull() {
    int res = eventNotConsumed;
    s1_exit();
    //#[ transition 1
    f1();
    f1();
    //#]
    s2_entDef();
    res = eventConsumed;
    return res;
};
  
```

In fact, what happens is that the transition code (between the `//#[` and `//#]` brackets) is immediately inlined (embedded) inside `rootState_dispatchEvent()`, as follows:

```

int MyClass::rootState_dispatchEvent(short id) {
    int res = eventNotConsumed;
    switch(rootState_active) {
        case s1:
            {
                if(id == Null_id)
                {
                    popNullConfig();
                }
            }
    }
}
  
```

```
        //#[ transition 1
        f1();
        f1();
        //#]
        pushNullConfig();
        rootState_subState = s2;
        rootState_active = s2;
        res = eventConsumed;
    }
    break;
};
case s2:
{
    if(id == Null_id)
    {
        popNullConfig();
        //#[ transition 2
        f2();
        f2();
        //#]
        rootState_subState = s3;
        rootState_active = s3;
        res = eventConsumed;
    }
    break;
};
};
return res;
};
```

This code is more efficient because it saves a few function calls via inlining of code. Inlining is enabled using the `CG::Class::ComplexityForInlining` property of the class, which is set to 3 by default. This means that if the user code (the action part of the transition) is shorter than three lines, it is inlined (or embedded) where the function call used to be instead of the function call. To get the “expected” result (not inlined), set this property to 0.

Panel Diagrams

A Rhapsody panel diagram provides you with a number of graphical control elements that you can use to create a graphical user interface (GUI) to monitor and regulate an application. Each control element can be bound to a model element (attribute/event/state). During animation, you can use the animated panel diagram to monitor (read) and regulate (change values/send events) your user application. For more information about animation, see [Animation](#).

This feature provides a convenient way to demonstrate the design and, additionally, provides you with an easy way to create a debug interface for the design.

Note

The panel diagram feature is only available for Rhapsody in C and Rhapsody in C++.

You can use a panel diagram to create the following types of panels for design and testing purposes:

- ◆ Hardware control panel designed for operating and monitoring machinery or instruments.
- ◆ Software graphical user interface (GUI) for display on a computer screen allowing the computer application user easier access to the application function than would be required if the user entered commands or other direct operational techniques.

Panel Diagram Features

You can create a panel diagram to design a graphical interface in Rhapsody in C and Rhapsody C++ projects. Developers may use the diagrams to:

- ◆ Simulate and prototype a panel
- ◆ Imitate hardware devices for users
- ◆ Activate and monitor a user application
- ◆ Test applications by triggering events and change attributes values

Note

Panel diagrams are intended only for simulating and prototyping, and not for use as a production interface for the user application. In addition, panel diagrams can only be “used” on the host and can be “used” only from within Rhapsody.

Panel Diagrams


The following illustration shows an animated panel diagram for a hypothetical coffee maker application. During animation, the developer of the application can test it by doing such things as, for example:

- ◆ Turn on the coffee maker application by clicking the **power** On/Off Switch control.
- ◆ Use the **coffeeContainer** and **milkContainer** Bubble Knob controls to increase/decrease the amount of coffee and milk that is available.
- ◆ Order a coffee by clicking the **evCoin** Push Button control. The following could happen:
 - Messages appear on the Matrix Display control, such as `Filling Coffee` or `Filling Cup`.
 - The **coffeeContainer** and **milkContainer** Level Indicator controls go down as these items are dispensed.
 - The **cup** Level Indicator control rises as coffee fills a cup, until the `Please Take Your Cup` message appears on the Matrix Display control.
 - The **Take cup** LED control turns red.
 - The **cupCounter** Digital Display control keeps a count of each cup of coffee made.
- ◆ Indicate that a cup of coffee has been taken by clicking the **evTakeCup** Push Button control, which could reset the coffee machine.



General Procedure for Using the Panel Diagram

The following procedure lists the general steps to create and use a panel diagram. References to the more specific procedures are noted when necessary.














1. Open your Rhapsody model with model elements ready for use.
2. Create a panel diagram; click the Panel Diagram button  on the **Diagrams** toolbar or choose **Tools > Panel Diagram**. See [Graphic Editors](#) for basic information on diagrams, including how to create, open, and delete them.
3. Create a control element in your panel diagram; click any of the tools on the panel diagram **Drawing** toolbar. See [Panel Diagram Drawing Tools](#).
4. Bind the control to a model element; right-click a control and select **Features**. Use the **Element Binding** tab on the Control Properties tab. See [Binding a Control Element to a Model Element](#).
5. Make whatever changes you may want for the control; see:
 - ◆ [Changing the Settings for a Control Element](#)
 - ◆ [Changing the Properties for a Control Element](#), when applicable
 - ◆ [Setting the Value Bindings for a Button Array Control](#), when applicable
 - ◆ [Changing the Display Name for a Control Element](#)
6. Set your model for animation. See [Animation](#).
7. Generate and make your model. Run your application and the animation for the panel diagram. When animation starts, the control on your panel diagram is initiated with its bound model element value. See [Basic Code Generation Concepts](#).
8. Use your control element(s) on the animated panel diagram. Note that when animation is running, the Control Properties dialog box and the Display Options dialog box are not available.
9. Terminate animation to terminate the use of the control element and exit animation.

Creating Panel Diagram Elements

The following sections describe the drawing tools available for a panel diagram. See [Graphic Editors](#) for basic information on diagrams, including how to create, open, and delete them.

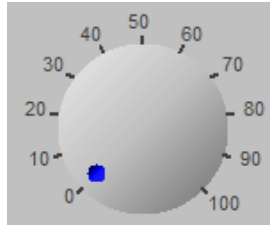
Panel Diagram Drawing Tools

The **Drawing** toolbar for a panel diagram contains the following tools.

Drawing Icon	Description
	Select lets you select a control on a panel diagram.
	Knob represents a Bubble Knob control. See Bubble Knob Control .
	Gauge represents a Gauge control. See Gauge Control .
	Meter represents a Meter control. See Meter Control .
	Level Indicator represents a Level Indicator control. See Level Indicator Control .
	Matrix Display represents Matrix Display control that shows a text string. See Matrix Display Control .
	Digital Display represents a Digital Display control that shows numbers. See Digital Display Control .
	LED represents a light-emitting diode control. See LED Control .
	On/Off Switch represents an On/Off Switch control. See On/Off Switch Control .
	Push Button represents a Push Button control. See Push Button Control .
	Button Array represents a Button Array control. See Button Array Control .
	Text Box represents an editable Text Box control. See Text Box Control .
	Slider represents a Slider control. See Slider Control .

Bubble Knob Control


The Bubble Knob control is an input/output control, as shown in the following figure in its default non-animated appearance:



Note the following:

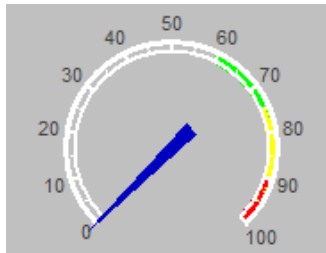
- ◆ You can bind (map) it to an attribute.
- ◆ Its attribute type is a Number.
- ◆ By default its control direction is set to In/Out, though you can change it to In or Out.

To draw a Bubble Knob control on a panel diagram, follow these steps:

1. Click the Knob button  on the **Drawing** toolbar.
2. Click the drawing area to create a Bubble Knob control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Binding a Control Element to a Model Element](#).
4. Make whatever changes you may want for the control:
 - ◆ To change the settings for the control, see [Changing the Settings for a Control Element](#)
 - ◆ To change the properties for the control, see [Changing the Properties for a Control Element](#) and [Properties for a Bubble Knob Control](#)
 - ◆ To change the display name for the control, see [Changing the Display Name for a Control Element](#)

Gauge Control


The Gauge control is an output control that appears as an analog round dial, as shown in the following figure in its default non-animated appearance:



Note the following:

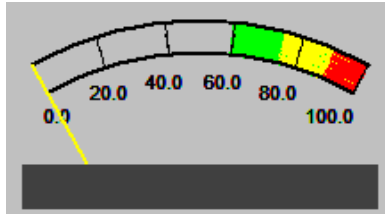
- ◆ You can bind it to an attribute.
- ◆ Its attribute type is a Number.

To draw a Gauge control on a panel diagram, follow these steps:

1. Click the Gauge button  on the **Drawing** toolbar.
2. Click the drawing area to create a Gauge control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Binding a Control Element to a Model Element](#).
4. Make whatever changes you may want for the control:
 - ◆ To change the settings for the control, see [Changing the Settings for a Control Element](#)
 - ◆ To change the properties for the control, see [Changing the Properties for a Control Element](#) and [Properties for a Gauge Control](#)
 - ◆ To change the display name for the control, see [Changing the Display Name for a Control Element](#)

Meter Control


The Meter control is an output control that appears as an analog meter, as shown in the following figure in its default non-animated appearance:



Note the following:

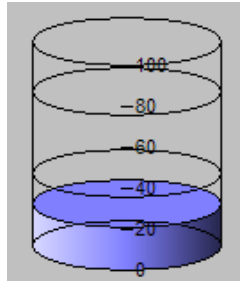
- ◆ You can bind it to an attribute.
- ◆ Its attribute type is a Number.

To draw a Meter control on a panel diagram, follow these steps:

1. Click the Meter button  on the **Drawing** toolbar.
2. Click the drawing area to create a Meter control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Binding a Control Element to a Model Element](#).
4. Make whatever changes you may want for the control:
 - ◆ To change the settings for the control, see [Changing the Settings for a Control Element](#)
 - ◆ To change the properties for the control, see [Changing the Properties for a Control Element](#) and [Properties for a Meter Control](#)
 - ◆ To change the display name for the control, see [Changing the Display Name for a Control Element](#)

Level Indicator Control


The Level Indicator control is an output control. By default it appears as a vertical 3-dimensional cylindrical level indicator, as shown in the following figure in its default non-animated appearance. However, you can change its appearance (for example to a 3-dimensional square shape) through the **Properties** tab of the Control Properties dialog box.



Note the following:

- ◆ You can bind it to an attribute.
- ◆ Its attribute type is a Number.

To draw a Level Indicator control on a panel diagram, follow these steps:

1. Click the Level Indicator button  on the **Drawing** toolbar.
2. Click the drawing area to create a Level Indicator control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Binding a Control Element to a Model Element](#).
4. Make whatever changes you may want for the control:
 - ◆ To change the settings for the control, see [Changing the Settings for a Control Element](#)
 - ◆ To change the properties for the control, see [Changing the Properties for a Control Element](#) and [Properties for a Level Indicator Control](#)
 - ◆ To change the display name for the control, see [Changing the Display Name for a Control Element](#)

Matrix Display Control


The Matrix Display control is an output control, as shown in the following figure in an example of an animated appearance:



Note the following:

- ◆ You can bind it to an attribute.
- ◆ Its attribute types are Number and String.

To draw a Matrix Display control on a panel diagram, follow these steps:

1. Click the Matrix Display button  on the **Drawing** toolbar.
2. Click the drawing area to create a Matrix Display control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Binding a Control Element to a Model Element](#).
4. Make whatever changes you may want for the control:
 - ◆ To change the settings for the control, see [Changing the Settings for a Control Element](#)
 - ◆ To change the properties for the control, see [Changing the Properties for a Control Element](#) and [Properties for a Matrix Display Control](#)
 - ◆ To change the display name for the control, see [Changing the Display Name for a Control Element](#)

Digital Display Control


The Digital Display control is an output control, as shown in the following figure in an example of an animated appearance:



Note the following:

- ◆ You can bind it to an attribute.
- ◆ Its attribute types are a Number and a String.

To draw a Digital Display control on a panel diagram, follow these steps:

1. Click the Digital Display button  on the **Drawing** toolbar.
2. Click the drawing area to create a Digital Display control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Binding a Control Element to a Model Element](#).
4. Make whatever changes you may want for the control:
 - ◆ To change the settings for the control, see [Changing the Settings for a Control Element](#)
 - ◆ To change the properties for the control, see [Changing the Properties for a Control Element](#) and [Properties for a Digital Display Control](#)
 - ◆ To change the display name for the control, see [Changing the Display Name for a Control Element](#)

LED Control


The LED control is an output control, as shown in the following figure in an example of an animated appearance:



Note the following:

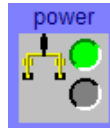
- ◆ You can bind it to a state or an attribute.
- ◆ Its attribute type is a Boolean.

To draw a LED control, follow these steps:

1. Click the LED button  on the **Drawing** toolbar.
2. Click the drawing area to create a LED control.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Binding a Control Element to a Model Element](#).
4. Make whatever changes you may want for the control:
 - ◆ To change the settings for the control, see [Changing the Settings for a Control Element](#)
 - ◆ To change the properties for the control, see [Changing the Properties for a Control Element](#) and [Properties for a LED Control](#)
 - ◆ To change the display name for the control, see [Changing the Display Name for a Control Element](#)

On/Off Switch Control


The On/Off Switch control is an input/output control, as shown in the following figure in one of its many possible shape styles in an example of an animated appearance:



Note the following:

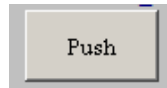
- ◆ You can bind it to a state or an attribute.
- ◆ Its attribute type is a Boolean.
- ◆ By default its control direction is set to In/Out, though you can change it to In or Out.

To draw a On/Off Switch control on a panel diagram, follow these steps:

1. Click the On/Off Switch button  on the **Drawing** toolbar.
2. Click the drawing area to create a On/Off Switch control.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Binding a Control Element to a Model Element](#).
4. Make whatever changes you may want for the control:
 - ◆ To change the settings for the control, see [Changing the Settings for a Control Element](#)
 - ◆ To change the properties for the control, see [Changing the Properties for a Control Element](#) and [Properties for a On/Off Switch Control](#)
 - ◆ To change the display name for the control, see [Changing the Display Name for a Control Element](#)

Push Button Control


The Push Button control is an input control, as shown in the following figure in its default non-animated appearance:



Note the following:

- ◆ You can bind it to an event.
- ◆ By default, this control injects a none parameter event.
- ◆ You can set a fix parameter for the event.

To draw a Push Button control on a panel diagram, follow these steps:

1. Click the Push Button button  on the **Drawing** toolbar.
2. Click the drawing area to create a Push Button control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Binding a Control Element to a Model Element](#).
4. Make whatever changes you may want for the control:
 - ◆ To change the settings for the control, see [Changing the Settings for a Control Element](#)
 - ◆ To change the display name for the control, see [Changing the Display Name for a Control Element](#)

Button Array Control


The Button Array control is an input/output control, as shown in the following figure in its default non-animated appearance:



Note the following:

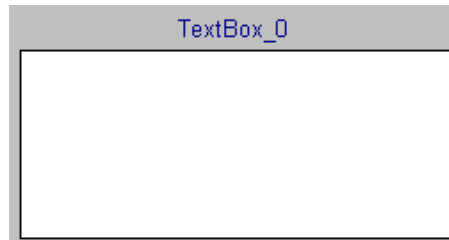
- ◆ You can bind it to an attribute. In addition, you can set a value for each switch to be set on the attribute.
- ◆ Its attribute types are a Number and a String.
- ◆ By default its control direction In/Out, though you can change it to In or Out.

To draw a Button Array control on a panel diagram, follow these steps:

1. Click the Button Array button  on the **Drawing** toolbar.
2. Click the drawing area to create a Button Array control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Binding a Control Element to a Model Element](#).
4. Make whatever changes you may want for the control:
 - ◆ To change the settings for the control, see [Changing the Settings for a Control Element](#)
 - ◆ To change the value binding for the control; see [Setting the Value Bindings for a Button Array Control](#).
 - ◆ To change the display name for the control, see [Changing the Display Name for a Control Element](#)

Text Box Control


The editable Text Box control is an input/output control, as shown in the following figure in its default non-animated appearance:



Note the following:

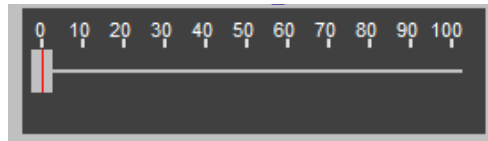
- ◆ You can bind it to an attribute.
- ◆ Its attributes types are a Number and a String.
- ◆ By default its control direction In/Out, though you can change it to In or Out.

To draw a Text Box control on a panel diagram, follow these steps:

1. Click the Text Box button  on the **Drawing** toolbar.
2. Click the drawing area to create a Text Box control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Binding a Control Element to a Model Element](#).
4. Make whatever changes you may want for the control:
 - ◆ To change the settings for the control, see [Changing the Settings for a Control Element](#)
 - ◆ To change the display name for the control and/or its data flow direction, see [Changing the Display Name for a Control Element](#).
 - ◆ To change the format settings (for example, line, fill, and font) for a Text Box control, see [Changing the Format of a Single Element](#).

Slider Control


The Slider control is an input/output control, as shown in the following figure in its default non-animated appearance:



Note the following:

- ◆ You can bind it to an attribute.
- ◆ Its attribute type is a Number.
- ◆ By default its control direction In/Out, though you can change it to In or Out.

To draw a Slider control on a panel diagram, follow these steps:

1. Click the Slider button  on the **Drawing** toolbar.
2. Click the drawing area to create a Slider control, or click and drag so that you can create the control to a certain size.
3. To bind the control element to the model element that the control is to regulate or monitor, see [Binding a Control Element to a Model Element](#).
4. Make whatever changes you may want for the control:
 - ◆ To change the settings for the control, see [Changing the Settings for a Control Element](#)
 - ◆ To change the properties for the control, see [Changing the Properties for a Control Element](#) and [Properties for a Slider Control](#)
 - ◆ To change the display name for the control, see [Changing the Display Name for a Control Element](#)

Binding a Control Element to a Model Element

In a panel diagram, the control element is a GUI that has to be connected to some “real” source/target element. In Rhapsody, binding (mapping) ties the operation between the control element to the model element it is to regulate or monitor.

A binding definition for a control element defines the following binding settings of each control element.

- ◆ Element type:
 - Control element - input sets data to bound element
 - Monitor element - output gets data from bound element
- ◆ Valid model elements for binding (attribute, event, and state)
- ◆ Value attribute types that can be set/get by the control element (Number, String, or Boolean)

A binding definition for a control element is predefined in Rhapsody. It cannot be changed. For example, the Bubble Knob has the following binding definition:

- ◆ Element role: input, output, or both
- ◆ Valid model elements: attributes
- ◆ Value types: numbers

Be sure that the bounded element type (for example, an `int`) is being supported by the control.

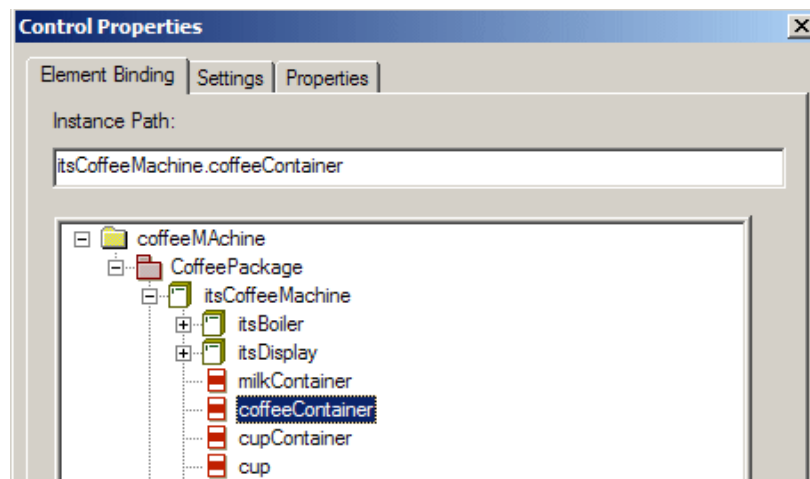
In the binding operation, you have to set the model element for binding. You may also set the instance path.

To bind a control element to a model element in a panel diagram, follow these steps:

1. Right-click the control and select **Features** to open the Control Properties dialog box.
2. On the **Element Binding** tab, depending on your situation:
 - ◆ If the control has no binding, the Control Binding Browser opens with the project container as the selected item. Use the browser to navigate to and select the element(s) for which you want to bind to the control element, or you can enter the element path in the **Instance Path** box.

Note: The browser root is the project and the end nodes are the meta classes that can be bound for the particular control element. If no relevant end node is found, Rhapsody notifies you that no relevant item was found and the Control Properties dialog box does not open.

- ◆ If the control has a bound element, the browser opens with the bound model element selected, as show in the following figure.



- ◆ If no relevant element for binding is found in the model, the **Element Binding** tab appears blank with a note to that effect.

Note: For more information about binding, see [More about Binding a Control Element](#).

3. Click **OK**.

More about Binding a Control Element

Note the following about binding a control element:

- ◆ **Binding an item of a modeled root Instance**

In the case where the element for binding is owned by a modeled root Instance, the object is displayed by the browser on the **Element Binding** tab on the Control Properties dialog box containing all relevant items for binding. You can select the element from the browser or alternatively type in the element path (stating at modeled root object) in the **Instance Path** box.

- ◆ **Binding an item of a dynamic root Instance**

In the case where the element for binding is owned by a dynamic root Instance (modeled class that will be instantiated at run time), the element root class is displayed by the browser on the **Element Binding** tab containing all relevant items for binding. You can select the element from the browser or alternatively type in the element path (stating at modeled root class) in the **Instance Path** box.

In both ways, if the dynamic Instance name is different from the default class instance name, the name should be entered following item selection.

- ◆ **Binding an item with multiplicity on its root object and parts**

In the case where bound element owner parts has multiplicity: Selecting the element through the browser on the **Element Binding** tab creates the path in the **Instance Path** box with “0” multiplicity on the relevant parts. You can then set the multiplicity as needed. If you enter a path, it is your responsibility to add multiplicity where needed.

Panel Diagrams

The following table summarizes the binding (mapping) characteristics for each panel diagram control element:

Control Name	Direction		Bound Element			Attribute Type		
	In	Out	Event	State	Attribute	Number	String	Boolean
Knob	✓	✓			✓	✓		
Gauge		✓			✓	✓		
Meter		✓			✓	✓		
Level Indicator		✓			✓	✓		
Matrix Display		✓			✓	✓	✓	
Digital Display		✓			✓	✓	✓	
LED		✓		✓	✓			✓
On/Off Switch	✓	✓		✓	✓			✓
Push Button	✓		✓					
Button Array	✓	✓			✓	✓	✓	
Text Box	✓	✓			✓	✓	✓	
Slider	✓	✓			✓	✓		

Attribute Types

Only attributes that hold predefined primitive (or enumeration) types could be bound. The supported predefined types are:

- ◆ **Number:** int, unsigned int, short, unsigned short, long, double, float, RhpInteger, RhpPositive, RhpReal
- ◆ **String:** char, char*, RhpString, OMString, CString
- ◆ **Boolean:** Bool, OMBoolean, RhpBoolean, RiCBoolean

Changing the Settings for a Control Element

You can change the settings for the control elements available for a panel diagram. For example, when applicable, its:

- ◆ Minimum and maximum values
- ◆ Shape style (in the case of a On/Off switch)
- ◆ Caption (in the case of the Push Button control)
- ◆ Color scheme (in the case of the Matrix Display and Digital Display controls)

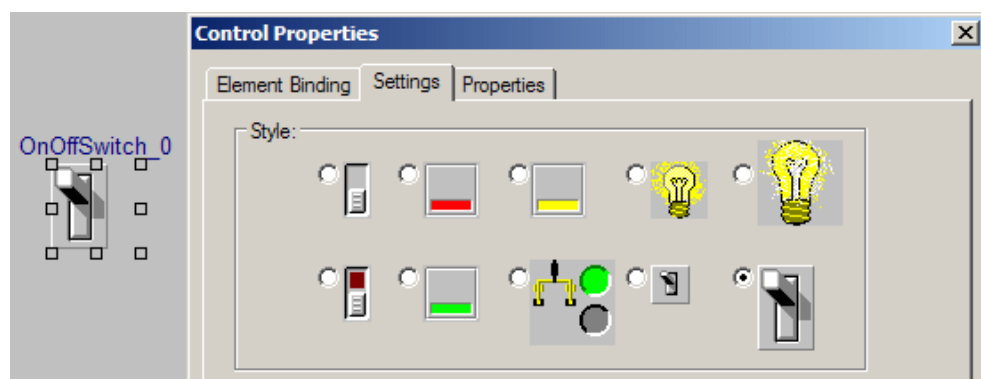
You do this through the **Settings** tab on the Control Properties dialog box for a control element.

Where possible (for example, for the Bubble Knob control), you can also change their control direction (to input, output, or both). By default, they are set to **InOut**. You can use the **Control Direction** area of the **Settings** tab on the Control Properties dialog box to change the control direction when possible.

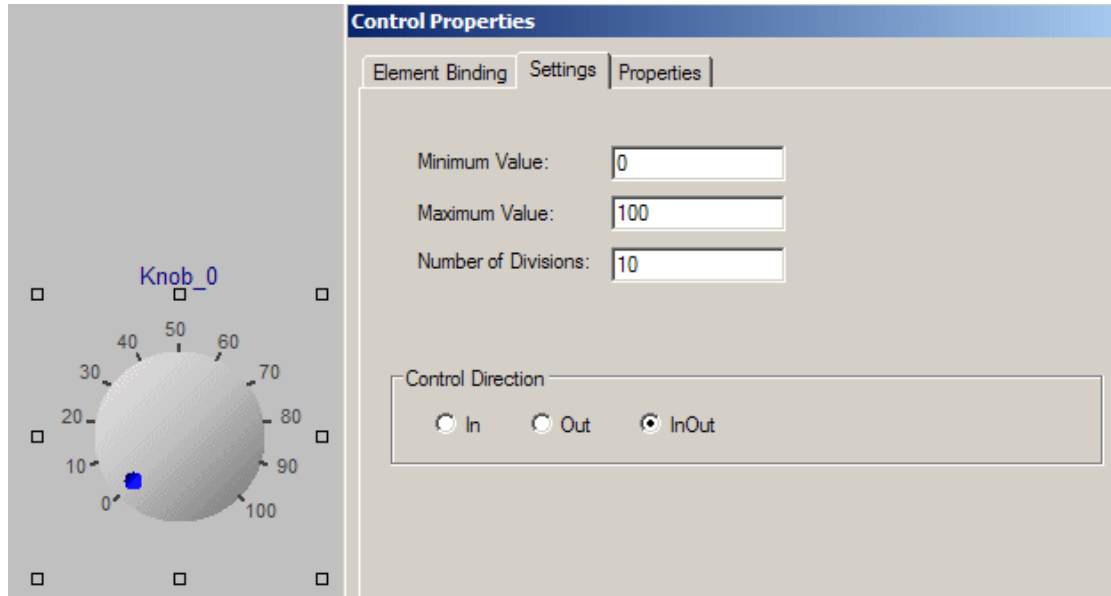
To change the settings for a control, follow these steps:

1. Right-click a control and select **Features** to open the Control Properties dialog box.
2. On the **Settings** tab, as shown in the following figure, make your changes:

Note: The settings that are available depend on the type of control element you have selected.



3. If applicable, select an option button in the **Control Direction** area, as shown in the following figure:



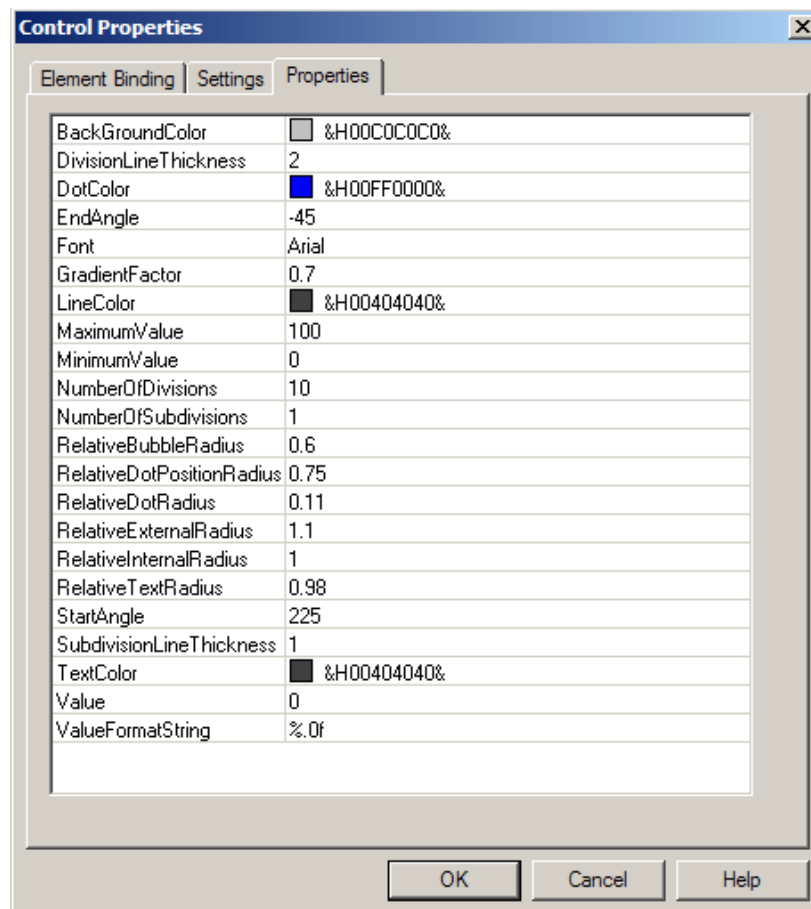
- ◆ **In** for input flow
 - ◆ **Out** for output flow
 - ◆ **InOut** for input/output flow
4. Click **OK**.

Changing the Properties for a Control Element

You can change the properties for many of the control elements available for a panel diagram (for example, its background color, range values, caption, and so forth). You can make changes through the **Properties** tab of the Control Properties dialog box for a control. These properties are ActiveX controls.

Note: The **Properties** tab appears only when appropriate for the selected control element. In addition, the tab shows only those settings that are applicable to that control element.

The following figure shows the **Properties** tab for a Bubble Knob control.



Properties for a Bubble Knob Control

The following table lists the properties that appear on the **Properties** tab of the Control Properties dialog box for a Bubble Knob control. You can change the properties as follows:

Device Settings	Explanation
BackgroundColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
DivisionLineThickness	To change the thickness of the major division lines (also referred to tick markers), change the value in the right column. The default value is 2.
DotColor	To change the color for the dot (indicator mark) on the Bubble Knob control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
EndAngle	To change the distance around the dial (between the minimum and maximum values), change the value in the right column. The default value is -45.
Font	To change the font for the text (for example, the numbers) on the control, click the Ellipses button in the right column and select a font from the Font dialog box that appears. The default value is Arial.
GradientFactor	To change the gradient factor for the control, change the value in the right column. The higher the number the more pronounced the gradient for the appearance of the knob, which appears as light to dark. The default value is 0.7.
LineColor	To change the color of all the tick markers, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
MaximumValue	To change the maximum value for the control, change the value in the right column. The default value is 100.
MinimumValue	To change the minimum value for the control, change the value in the right column. The default value is 0.
NumberOfDivisions	To change the number of major division lines (tick markers) for the control, change the value in the right column. The default value is 10. For example, with the maximum value set at 100, minimum value at 0, and division value at 5, your Bubble Knob control would show major tick markers at 0, 20, 40, 60, 80, and 100.
NumberOfSubdivisions	To change the number of minor division lines (tick markers) between two major ones, change the value in the right column. The default value is 1, which means no minor tick marker appears between two major tick markers. For example, to make three subdivision areas appear between two major markers, enter a value of 3.

Device Settings	Explanation
RelativeBubbleRadius	To change the relative bubble radius for the control, change the value in the right column. This regulates the relative size of the Bubble Knob control, which includes its number scale. The default value is 0.6.
RelativeDotPositionRadius	To change the placement (closer or farther away) of the dot indicator relative to the 0 value marker, change the value in the right column. The default value is 0.75.
RelativeDotRadius	To change the size of the dot indicator, change the value in the right column. The default value is 0.11.
RelativeExternalRadius	To change the relative external radius for the control, change the value in the right column. This setting changes the length of the major and minor tick markers while still touching the control. The default value is 1.1.
RelativeInternalRadius	To change the relative internal radius for the control, change the value in the right column. This setting changes the length of the major and minor tick markers and how far away they are from the the bubble. The default value is 1.
RelativeTextRadius	To change the relative text radius for the control, change the value in the right column. This setting changes the distance between the scale numbers and their associate tick makers. The default value is 0.98.
StartAngle	To change the position of the minimum value marker and the dot indicator, change the value in the right column. The default value is 225.
SubdivisionLineThickness	To change the thickness of the minor division lines (tick markers), change the value in the right column. The default value is 1.
TextColor	To change the color for the text (scale numbers) on the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
Value	To change the default placement of the dot indicator, change the value in the right column. The default value is 0.
ValueFormatString	To change the value format of the numbers on the control, change the value in the right column. The default value is % . 0 f , which shows numbers, for example, as 0, 10, 20, and so forth. For the value % . 1 f , the control would show number as 0.0, 10.0, 20.0, and so forth.

Properties for a Gauge Control

The following table lists the properties that appear on the **Properties** tab of the Control Properties dialog box for a Gauge control. You can change the properties as follows:

Device Settings	Explanation
BackColor	To change the back color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
BackgroundPicture	To add/change an image that appears as the background for the control, click the Ellipses button in the right column and open a bitmap file.
Caption	To insert a caption to show in the center of the Gauge control, type your text in the right column. You can use <code>RelativeCaptionY</code> , <code>RelativeCaptionX</code> , <code>RelativeCenterY</code> , and <code>RelativeCenterX</code> to position the caption somewhere on the gauge other than its center.
CaptionColor	To change the color for your caption text, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
CaptionFont	To change the font for your caption text, click the Ellipses button in the right column and select a font from the Font dialog box that appears. The default value is <code>Arial</code> .
DivisionLineThickness	To change the thickness of the major division lines (also known as tick markers), change the value in the right column. The default value is 2.
EnclosingCircleColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
EndAngle	To change the angle of the scale on the high end of the control, change the value in the right column. The value widens (lower value) or shrinks (higher value) the empty space between the low and high ends of the control. The default is -45.
ExternalCircleThickness	To change the thickness of the external circle of the control, change the value in the right column. The default value is 3.
ForeColor	To change the foreground color, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
GreenColor	To change the color (typically green) of the OK area of the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
GreenStartValue	To change the beginning value for the OK area on the control, change the value in the right column. The default value is 60.
IndexColor	To change the color of the needle indicator for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.

Device Settings	Explanation
IndexLineThickness	To change the thickness of the needle indicator for the control, change the value in the right column. The default value is 0.
InternalCircleThickness	To change the thickness of the internal circle of the control, change the value in the right column. The default value is 2.
MaximumValue	To change the maximum value for the control, change the value in the right column. The default value is 100.
MinimumValue	To change the minimum value for the control, change the value in the right column. The default value is 0.
NumberColor	To change the color of the numbers on the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
NumberFont	To change the font for the numbers on the control, click the Ellipses button in the right column and select a font from the Font dialog box that appears. The default value is Arial.
NumberOfDivisions	To change the number of major division lines (tick markers) for the control, change the value in the right column. The default value is 10. For example, with the maximum value set at 100, minimum value at 0, and division at 5, your control would show major tick markers at 0, 20, 40, 60, 80, and 100.
NumberOfSubdivisions	To change the number of minor division lines (tick markers) between two major ones, change the value in the right column. The default value is 2, which means one minor tick marker appears between two major tick markers. For example, to make three subdivision areas appear between two major markers, enter a value of 3.
RedColor	To change the color (typically red) of the Warning area of the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
RedStartValue	To change the beginning value for the Warning area on the gauge, change the value in the right column. The default value is 90.
RelativeCaptionX	To change the relative X coordinate (east/west) of the text entered for Caption, change the value in the right column. The default value is 0.5.
RelativeCaptionY	To change the relative Y (north/south) coordinate of the text entered for Caption, change the value in the right column. The default value is 0.5.
RelativeCenterX	To change the relative center Y coordinate of the control, change the value in the right column. The default value is 0.5.
RelativeCenterY	To change the relative center X coordinate of the control, change the value in the right column. The default value is 0.55.
RelativeEnclosingCircleRadius	To change the relative enclosing circle radius of the control, change the value in the right column. The default value is .98.

Device Settings	Explanation
RelativeExternalRadius	To change the relative external radius of the control, change the value in the right column. The default value is 1.1.
RelativeIndexBackLength	To change the relative length the needle indicator from the back (not pointy) end, change the value in the right column. The default value is 0.3.
RelativeIndexLength	To change the relative length of the needle indicator from the front (pointy) end, change the value in the right column. This lengthens the needle indicator at its back (wider) end. The default value is 1.2.
RelativeInternalRadius	To change the relative internal radius of the control, change the value in the right column. The default value is 0.35.
RelativeTextRadius	To change the relative position of the text (scale numbers), change the value in the right column. The default value is 1.1.
ScaleCircleColor	To change the color of the scale circle, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
StartAngle	To change the position of the minimum value marker and the needle indicator, change the value in the right column. The default value is 225.
StepValue	To change the step value, change the value in the right column. The default is 1.
SubdivisionLineThickness	To change the thickness of the minor division lines (tick markers), change the value in the right column. The default value is 1.
TailAngle	To change the thickness of the needle indicator, change the value in the right column. The default is 165.
Value	To change the default value for the needle indicator, change the value in the right column. The default value is 0.
ValueFormatString	To change the value format of the scale number, change the value in the right column. The default value is %.0f, which shows numbers, for example, as 0, 10, 20, and so forth. For the value %.1f, the control would show number as 0.0, 10.0, 20.0, and so forth.
YellowColor	To change the color (typically yellow) of the Caution area of the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
YellowStartValue	To change the beginning value for the Caution area on the gauge, change the value in the right column. The default value is 75.

Properties for a Meter Control

The following table lists the properties that appear on the **Properties** tab of the Control Properties dialog box for a Meter control. You can change the properties as follows:

Device Settings	Explanation
BackgroundImage	To add an image so that it appears as the background for the control, click the Ellipses button in the right column and open a bitmap file.
BottomCoverColor	To change the color of the bottom cover for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears. You can use this setting in conjunction with BottomCoverRelativeHeight.
BottomCoverLabelFront	To change the font for the text on the bottom cover for the control, click the Ellipses button in the right column and select a font from the Font dialog box that appears. The default value is Arial. You use this setting in conjunction with Caption and BottomCoverTextColor.
BottomCoverRelativeHeight	To change the height of the bottom cover area for the control, change the value in the right column. The default is 0.2. You can use this setting in conjunction with BottomCoverColor.
BottomCoverTextColor	To change the color for the text that appears in the bottom cover area, click the drop-down arrow in the right column and select a color from the Palette tab that appears. You use this setting in conjunction with Caption and BottomCoverLabelFront.
Caption	To insert a caption to appear in the bottom cover area of the control, type your text in the right column. You can use this setting in conjunction with BottomCoverLabelFront and BottomCoverTextColor.
GreenColor	To change the color (typically green) of the OK area of the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
GreenStartValue	To change the beginning value for the OK area on the control, change the value in the right column. The default value is 60.
IndexColor	To change the color of the needle indicator for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
IndexThickness	To change the thickness of the needle indicator, change the value in the right column. The default value is 2.
InstrumentBackgroundColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.

Device Settings	Explanation
MaximumValue	To change the maximum value for the control, change the value in the right column. The default value is 100.
MinimumValue	To change the minimum value for the control, change the value in the right column. The default value is 0.
NumberOfDivisions	To change the number of major division lines (tick markers) for the control, change the value in the right column. The default value is 5. For example, with the maximum value set at 100, minimum value at 0, and division at 5, your control would show major tick markers at 0, 20, 40, 60, 80, and 100.
NumberOfSubdivisions	To change the number of minor division lines (tick markers) between two major ones, change the value in the right column. The default value is 1, which means no minor tick marker appears between two major tick markers. For example, to make three subdivision areas appear between two major markers, enter a value of 3.
RedColor	To change the color (typically red) of the Warning area of the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
RedStartValue	To change the beginning value for the Warning area on the control, change the value in the right column. The default value is 90.
RelativeIndexLength	To change the relative length of the needle indicator, change the value in the right column. The default value is 1.1.
RelativeTextRadius	To change the relative position of the text (scale numbers), change the value in the right column. The default value is 0.9.
ScaleColor	To change the color of the scale outline and numbers, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
ScaledCircleRelativeDiameter	To change the relative diameter of the control, change the value in the right column. The default is 1.75.
ScaledCircleRelativeShifting	To change the relative height of the control, change the value in the right column. The default is 0.3.
ScaleRelativeWidth	To change the relative wide of the scale, change the value in the right column. The default is 0.1.
ScaleThickLineWidth	To change the top and bottom lines of the scale, change the value in the right column. The default is 2.
ScaleThinLineWidth	To change the thickness of the vertical lines of the scale, change the value in the right column. The default is 1.
ShadedAreaRelativeSize	The default is 0.
SmallScaleFont	To change the font for the scale numbers, click the Ellipses button in the right column and select a font from the Font dialog box that appears. The default value is <i>Arial</i> .

Device Settings	Explanation
StepValue	To change the step value, change the value in the right column. The default is 1.
Value	To change the default value for the needle indicator, change the value in the right column. The default value is 0.
ValueFormatString	To change the value format of the scale number, change the value in the right column. The default value is <code>% .1f</code> , which shows numbers, for example, as 0.0, 10.0, 20.0, and so forth. For the value <code>% .0f</code> , the control would show number as 0, 10, 20, and so forth.
VisibleScaleRelativeSize	To change the relative size of the visible scale, change the value in the right column. The default value is 1.2.
YellowColor	To change the color (typically yellow) of the Caution area of the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
YellowStartValue	To change the beginning value for the Caution area on the control, change the value in the right column. The default value is 75.

Properties for a Level Indicator Control

The following table lists the properties that appear on the **Properties** tab of the Control Properties dialog box for a Level Indicator control. You can change the properties as follows:

Device Settings	Explanation
BackColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
CurrentValue	To change the current (default) value for the control, change the value in the right column. The default is 20.
DrawTextLabels	The default is True.
EnableThreshold1	To enable/disable the appearance of the top middle threshold level line, change the value in the right column. True enables the appearance; False disables it. The default is True.
EnableThreshold2	To enable/disable the appearance of the bottom middle threshold level line, change the value in the right column. True enables the appearance; False disables it. The default is True.
Font	To change the font for the numbers on the control, click the Ellipses button in the right column and select a font from the Font dialog box that appears. The default value is Arial.
FormatString	To change the value format of the scale numbers, change the value in the right column. The default value is <code>%.0f</code> , which shows numbers, for example, as 0, 10, 20, and so forth. For the value <code>%.1f</code> , the control would show number as 0.0, 10.0, 20.0, and so forth.
GradientFactor	To change the gradient factor for the control, change the value in the right column. The higher the number the more pronounced the gradient for the appearance of the knob, which appears as light to dark. The default value is 0.7.
HorizontalLayout	To change the layout of the level indicator to be horizontal, change the value in the right column to True. The default is False.
LiquidColor	To change the color of the level indicator, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
MaximumValue	To change the maximum value for the control, change the value in the right column. The default value is 100.
MinimumValue	To change the minimum value for the control, change the value in the right column. The default value is 0.

Device Settings	Explanation
NumberOfDivisions	<p>To change the number of major division lines (tick markers) for the control, change the value in the right column. The default value is 5.</p> <p>For example, with the maximum value set at 100, minimum value at 0, and division at 5, your control would show major tick markers at 0, 20, 40, 60, 80, and 100.</p>
NumberOfSubdivisions	<p>To change the number of minor division lines (tick markers) between two major ones, change the value in the right column. The default value is 1, which means no minor tick marker appears between two major tick markers. For example, to make three subdivision areas appear between two major markers, enter a value of 3.</p>
RelativeDepth	<p>To change the appearance of the relative depth of the control, change the value in the right column. The default is 0.2.</p> <p>For example, a value of 0 gives the control a flat tall rectangular appearance. While a value of 0.2 gives it a 3-dimensional cylindrical appearance.</p> <p>Note that the appearance of the level indicator control is also affected by SquareShape, RelativeHeight and RelativeWidth.</p> <p>Note also that the appearance of the orientation of the level indicator is affected by HorizontalLayout.</p>
RelativeHeight	<p>To change the relative height of the control, change the value in the right column. The default is 0.9.</p>
RelativeWidth	<p>To change the relative width of the control, change the value in the right column. The default is 0.9.</p>
ShadedAreaRelativeSize	<p>The default is 0.</p>
SquareShape	<p>To change the appearance of the Level Indicator control to look like a 3-dimensional square, change the value in the right column to True. The default is False.</p>
Threshold1Color	<p>To change the color of the top middle threshold level line, click the drop-down arrow in the right column and select a color from the Palette tab that appears.</p>
Threshold1Thickness	<p>To change the thickness of the top middle threshold level line, change the value in the right column. The default is 1.</p>
Threshold1Value	<p>To change the position of the bottom middle threshold level line, change the value in the right column. The default is 75.</p>
Threshold2Color	<p>To change the color of the bottom middle threshold level line, click the drop-down arrow in the right column and select a color from the Palette tab that appears.</p>
Threshold2Thickness	<p>To change the thickness of the bottom middle threshold level line, change the value in the right column. The default is 1.</p>
Threshold2Value	<p>To change the position of the bottom threshold level line, change the value in the right column. The default is 35.</p>

Device Settings	Explanation
UseColorGradients	The default is True.

Properties for a Matrix Display Control

The following table lists the properties that appear on the **Properties** tab of the Control Properties dialog box for a Matrix Display control. You can change the properties as follows:

Device Settings	Explanation
BackColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
Caption	To insert a caption to appear on the control, type your text in the right column.
Style	To change the appearance of the background of the control, change the value in the right column. The default is 0.

Properties for a Digital Display Control

The following table lists the properties that appear on the **Properties** tab of the Control Properties dialog box for a Digital Display control. You can change the properties as follows:

Device Settings	Explanation
BackColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
DisplayedString	To insert text to display on the control, type your text in the right column.
Style	To change the appearance of the background for the control, change the value in the right column. The default is 4.











Properties for a LED Control

The following table lists the properties that appear on the **Properties** tab of the Control Properties dialog box for an LED control. You can change the properties as follows:

Device Settings	Explanation
BackColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears
BlackWhenOff	To change the appearance of the control to black when the application is off, change the value in the right column to <code>True</code> . The default is <code>False</code> .
Blinking	To change it so that the LED is blinking, change the value in the right column to <code>True</code> . The default is <code>False</code> .
BlinkingTimeMillesec	To change the blinking rate, change the value in the right column. The default is <code>300</code> .
Color	To change the color for the LED, change the value in the right column. The default is <code>2</code> .
State	To change the state for the LED, change the value in the right column to <code>True</code> . The default is <code>False</code> .
Style	To change the appearance of the background of the control, change the value in the right column. The default is <code>0</code> .

Properties for a On/Off Switch Control

The following table lists the properties that appear on the **Properties** tab of the Control Properties dialog box for an On/Off Switch control. You can change the properties as follows:

Device Settings	Explanation
BackColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
IconSet	<p>To change the appearance of the On/Off switch icon, change the value in the right column.</p> <p>0 = </p> <p>1 = </p> <p>2 = </p> <p>3 = </p> <p>4 = </p> <p>5 = </p> <p>6 = </p> <p>7 = </p> <p>8 = </p> <p>9 =  (default)</p>
State	To change the state of the control, change the value in the right column. The default is <code>True</code> .
UserInteractionEnabled	To change is user interaction is enable, change the value in the right column. The default is <code>True</code> .

Properties for a Slider Control

The following table lists the properties that appear on the **Properties** tab of the Control Properties dialog box for a Slider control. You can change the properties as follows:

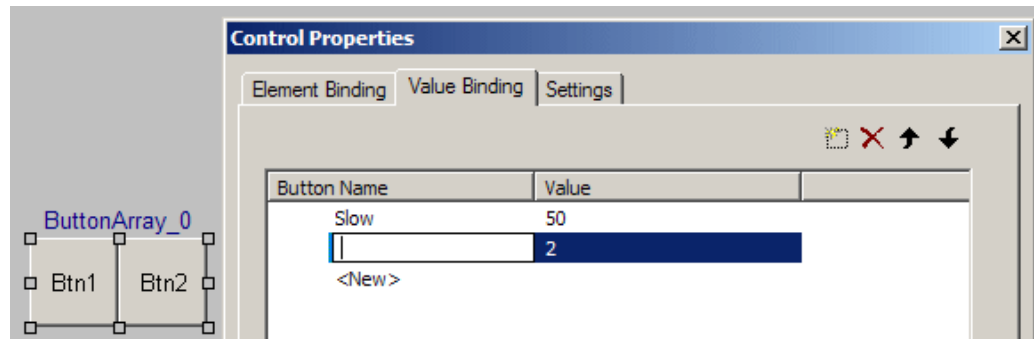
Device Settings	Explanation
BackgroundColor	To change the background color for the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
DivisionLineThickness	To change the thickness of the major division lines (also referred to tick markers), change the value in the right column. The default value is 2.
DotColor	To change the color for the indicator mark on the Slider control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
EndAngle	To change the distance on the Slider rule (between the minimum and maximum values), change the value in the right column. The default value is -45.
Font	To change the font for the text (for example, the numbers) on the control, click the Ellipses button in the right column and select a font from the Font dialog box that appears. The default value is Arial.
GradientFactor	To change the gradient factor for the control, change the value in the right column. The higher the number the more pronounced the gradient for the appearance of the Slider control, which appears as light to dark. The default value is 0.7.
LineColor	To change the color of all the tick markers, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
MaximumValue	To change the maximum value for the control, change the value in the right column. The default value is 100.
MinimumValue	To change the minimum value for the control, change the value in the right column. The default value is 0.
NumberOfDivisions	To change the number of major division lines (tick markers) for the control, change the value in the right column. The default value is 10. For example, with the maximum value set at 100, minimum value at 0, and division value at 5, your Slider control would show major tick markers at 0, 20, 40, 60, 80, and 100.
NumberOfSubdivisions	To change the number of minor division lines (tick markers) between two major ones, change the value in the right column. The default value is 1, which means no minor tick marker appears between two major tick markers. For example, to make three subdivision areas appear between two major markers, enter a value of 3.

Device Settings	Explanation
RelativeBubbleRadius	To change the relative bubble radius for the control, change the value in the right column. This regulates the relative size of the Bubble Knob control, which includes its number scale. The default value is 0.1.
RelativeDotPositionRadius	To change the placement (closer or farther away) of the dot indicator relative to the 0 value marker, change the value in the right column. The default value is 0.75.
RelativeDotRadius	To change the size of the indicator mark, change the value in the right column. The default value is 2.
RelativeExternalRadius	To change the relative external radius for the control, change the value in the right column. This setting changes the length of the major and minor tick markers and how far away they are from the indicator mark line. The default value is 6.25.
RelativeInternalRadius	To change the relative internal radius for the control, change the value in the right column. This setting changes the length of the major and minor tick markers and how far away they are from the indicator mark line. The default value is 4.75.
RelativeTextRadius	To change the relative text radius for the control, change the value in the right column. This setting changes the distance between the scale numbers and their associate tick makers. The default value is 7.5.
StartAngle	To change the position of the minimum value marker and the indicator mark, change the value in the right column. The default value is 225.
SubdivisionLineThickness	To change the thickness of the minor division lines (tick markers), change the value in the right column. The default value is 1.
TextColor	To change the color for the text (scale numbers) on the control, click the drop-down arrow in the right column and select a color from the Palette tab that appears.
Value	To change the default placement of the indicator mark, change the value in the right column. The default value is 0.
ValueFormatString	To change the value format of the numbers on the control, change the value in the right column. The default value is % . 0f , which shows numbers, for example, as 0, 10, 20, and so forth. For the value % . 1f, the control would show number as 0.0, 10.0, 20.0, and so forth.

Setting the Value Bindings for a Button Array Control

To set the value bindings for a Button Array control for a panel diagram, follow these steps:

1. Right-click a Button Array control and select **Features**.
2. On the **Value Binding** tab, change the name of a button and its value, as shown in the following figure.



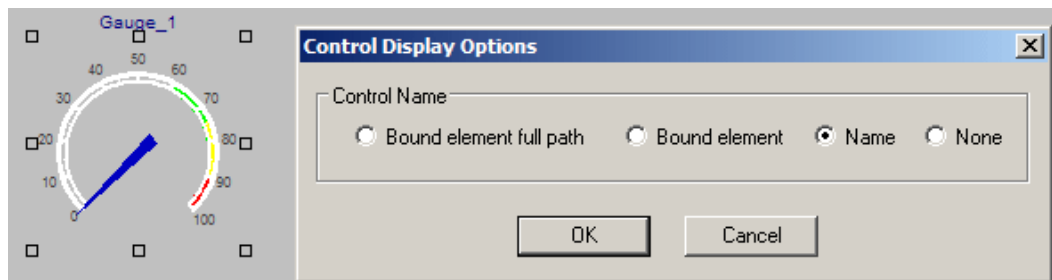
3. Optionally, you can click **<New>** to create another button in your array.
4. Click **OK**.

Changing the Display Name for a Control Element

You can change the display option for the name for any of the control elements available for a panel diagram. When you create a control, by default Rhapsody displays the name of the element (for example, Gauge_1 and Meter_5). You can use the Control Display Options dialog box to change the display option for the name of your control element.

To change the display name and/or data flow options for a control element, follow these steps:

1. Right-click a control on the panel diagram and select **Display Options** to open the Control Display Options dialog box, as shown in the following figure.



2. In the **Control Name** area, select an option button:
 - ◆ **Bound element full path** displays the full path of the bound element (for example, `Default.itsClass_0.speed`).
 - ◆ **Bound element** displays the name of the bound element (for example, `speed`).
 - ◆ **Name** displays the name of the control (for example, **Gauge_1**). This is the default.
 - ◆ **None** does not display any name.
3. Click **OK**.

Limitations

The panel diagram feature has the following limitations:

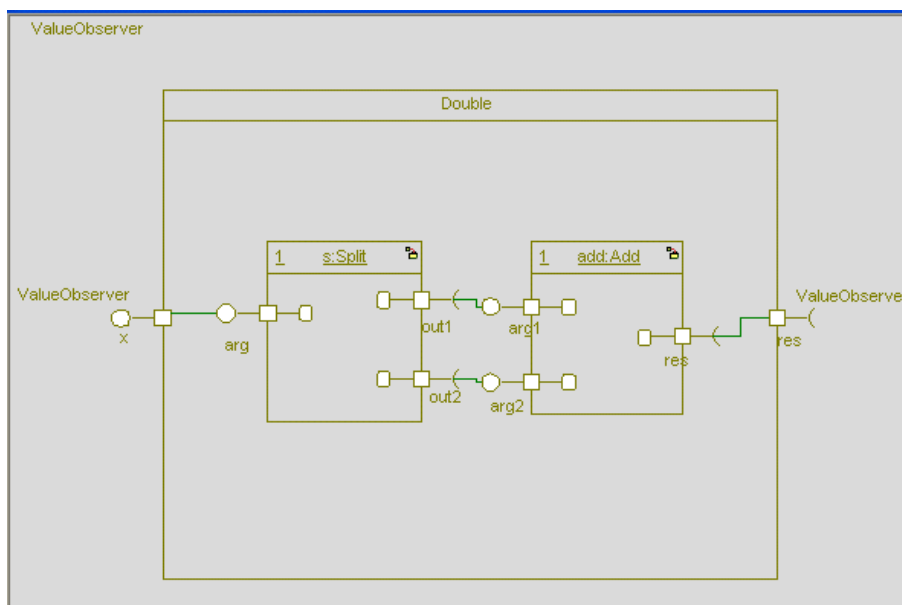
- ◆ Does not support composition made by relations (supports composition made by part only)
- ◆ Cannot launch an event with arguments
- ◆ Does not have support for graphical DiffMerge
- ◆ Is not supported in the Eclipse plug-in
- ◆ RiCString typed attribute (RiC) cannot be bound

Structure Diagrams

Structure diagrams model the structure of a composite class; any class or object that has an OMD can have a structure diagram. In addition, a structure diagram supports some the features supported by an OMD and uses the properties defined in the `ObjectModelGE` subject.








Object model diagrams focus more on the specification of classes, whereas structure diagrams focus on the instances used in the model. Although you can put classes in structure diagrams and objects in the OMD, the toolbars for the diagrams are different to allow a distinction between the specification of the system and its structure.

The following figure shows a structure diagram.



Structure Diagram Drawing Icons

The **Drawing** toolbar for a structure diagram includes the following tools:

Drawing Icon	Description
	Composite class is a container class. You can create objects and relations inside a composite class. See Composite Classes for more information.
	Object is the structural building block of a system. Objects form a cohesive unit of state (data) and services (behavior). Every object has a public part and an private part. See Objects for more information.
	File icon is only available for Rhapsody in C. It allows you to create file model elements. A file is a graphical representation of a specification (.h) or implementation (.c) source file. See External Files for more information.
	Create Port draws connection points among objects and their environments. See Ports for more information.
	Link creates an association between the base classes of two different objects. See Links for more information.
	Dependency creates a relationship in which the proper functioning of one element requires information provided by another element. See Dependencies for more information.
	Flow specifies the flow of data and commands within a system. See Flows for more information.

The following sections describe how to use these tools to draw the parts of a structure diagram. See [Graphic Editors](#) for basic information on diagrams, including how to create, open, and delete them.

Composite Classes

For detailed information about composite classes, see [Composite Classes](#) in [Object Model Diagrams](#).

Objects

Objects are the structural building blocks of a system. They form a cohesive unit of state (data) and services (behavior). Every object has a specification part (public) and an implementation part (private). See [Objects](#) for detailed information about objects.

Ports

A *port* is a distinct interaction point between a class and its environment, or between (the behavior of) a class and its internal parts. A port enables you to specify classes independently of the environment in which they will be embedded; the internal parts of the class can be completely isolated from the environment, and vice versa. See [Ports](#) for detailed information about ports.

Links

Rhapsody separates links from associations so you can have unambiguous model elements for links with a distinct notation in the diagrams. This allows specification of the following:

- ◆ Links without having to specify the association being instantiated by the link.
- ◆ Features of links that are not mapped to an association.

See [Links](#) for detailed information about links.

Dependencies

A *dependency* exists when the implementation or functioning of one element (class or package) requires the presence of another element. For example, if class C has an attribute a that is of class D, there is a dependency from C to D. See [Dependencies](#) for detailed information about dependencies.

Flows

Flows and FlowItems provide a mechanism for specifying exchange of information between system elements at a high level of abstraction. This functionality enables you to describe the flow of data and commands within a system at a very early stage, before committing to a specific design. As the system specification evolves, you can refine the abstraction to relate to the concrete implementation.

See [Flows and FlowItems](#) for detailed information about flows and FlowItems.

Creating an Object

To create an object, follow these steps:

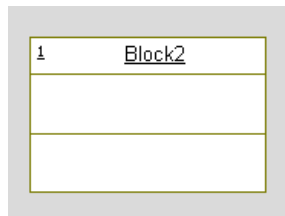
1. Click the **Object** icon in the **Drawing** toolbar.
2. Double-click, or click-and-drag, in the drawing area.
3. Edit the default name, then press **Enter**.

If you specify the name in the format `<ObjectName:ClassName>` (for an object with explicit type) and the class `<ClassName>` exists in the model, the new object will reference it. If it does not exist, Rhapsody prompts you to create it.

Alternatively, you can select **Edit > Add New > Object**. If you want to add a object to an OMD, you must use this method because the **Drawing** toolbar does not include a **Object** tool.

In the OMD, an object is shown like a class box, with the following differences:

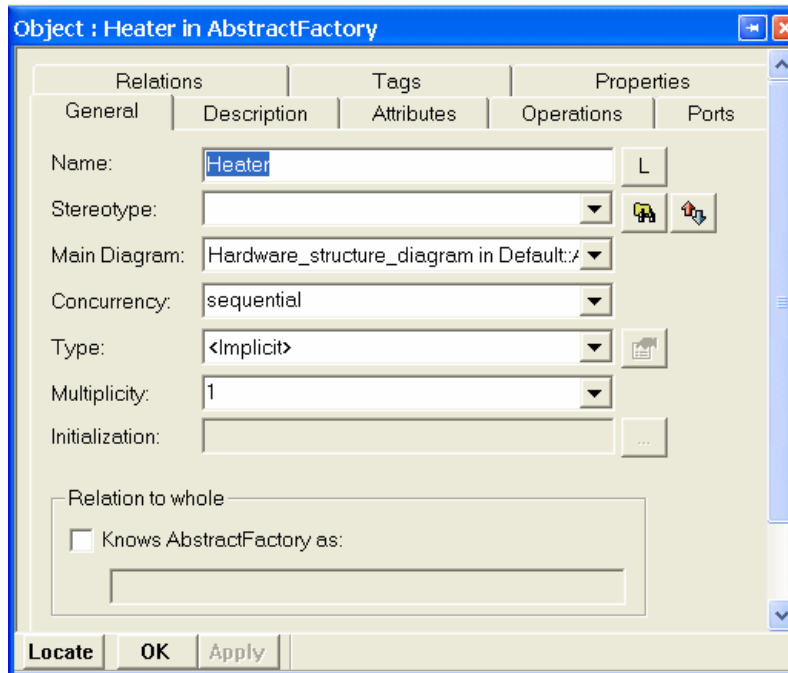
- ◆ The name of the object is underlined.
- ◆ The multiplicity is displayed in the upper, left-hand corner.



As with classes, you can display the attributes and operations in the object. See [Setting Display Options](#) for detailed information.

Modifying the Features of an Object

The Features dialog box enables you to change the features of an object, including its concurrency and multiplicity.



An object has the following features:

- ◆ **Name**—Specifies the name of the element. The default name is `object_n`, where *n* is an incremental integer starting with 0.
- ◆ **L**—Specifies the label for the element, if any. See [Labeling Elements](#) for information on creating labels.
- ◆ **Stereotype**—Specifies the stereotype of the object, if any. They are enclosed in guillemets, for example `«s1»` and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM Development Guide* for more information about these components.

- ◆ **Main Diagram**—Specifies the main diagram for the object. This field is enabled only for objects with implicit type.
- ◆ **Concurrency**—Specifies the concurrency of the object. This field is enabled only for objects with implicit type. The possible values are as follows:

- **Sequential**—The element will run with other classes on a single system thread. This means you can access this element only from one active class.
 - **Active**—The element will start its own thread and run concurrently with other active classes.
- ◆ **Type**—Specifies the class of which the object is an instance. To view that class’s features, click the **Invoke Features Dialog** icon next to the **Type** field.

In addition to the names of all the instantiated classes in the model, this drop-down list includes the following:

- **<Implicit>**—Specifies an implicit object
 - **<Explicit>**—Specifies an explicit object
 - **<New>**—Enables you to specify a new class
 - **<Select>**—Enables you to browse for a class using the selection tree
- ◆ **Multiplicity**—Specifies the number of occurrences of this instance in the project. Common values are one (1), zero or one (0,1), or one or more (1..*).
- ◆ **Initialization**—Specifies the constructor being called when the object is created. If you click the Ellipsis button, the Actual Call dialog box opens so you can see the details of the call.

If the part does not have a constructor, with parameters, this field is dimmed.

- ◆ **Relation to whole**—Enables you to name the relation for a part. If the object is part of a composite class, enable the **Knows its whole as** check box and type a name for the relation in the text box. This relation is displayed in the browser under the **Association Ends** category under the instantiated class or implicit object.

If the **Relation to whole** field is specified on the **General** tab, the Features dialog box includes tabs to define that relation and its properties. However, on the tab that specifies the features of its whole (in the illustration of the `itsController` tab), only the fields **Name**, **Label**, **Stereotype**, and **Description** can be modified. See [Modifying the Features of an Association](#) for more information on relation features.

Changing the Order of Objects

1. In the browser, right-click the **Object** category icon, then select **Edit Objects Order** from the pop-up menu.
2. Unselect the **Use default order** check box.
3. Select the object you want to move.
4. Click **Up** to generate the object earlier or **Down** to generate it later.
5. Click **OK** to apply your changes and close the dialog box.

Supported Rhapsody Functionality in Objects

The following table lists the Rhapsody functionality supported by objects.

Functionality	Description
Roundtrip	Full support.
Diff/Merge	Full support.
ReporterPLUS	ReporterPLUS shows objects separately from objects.
Internal reporter	Objects are printed in the group "Objects."
Search and replace	You can search for objects in the model and select their type from the list of possible types. See Searching in the Model for more information.
Check Model	The same checks for objects are used for objects. See Checks for more information.
XMI	Controls export of objects.
DOORS	Objects can be exported to and checked by DOORS.

External Files

Rhapsody in C enables you to create model elements that represent external files. An *external file* (or simply *file*) is a graphical representation of a specification (.h) or implementation (.c) source file. This new model element enables you to use functional modeling and take advantage of the capabilities of Rhapsody (modeling, execution, code generation, and reverse engineering) without radically changing the existing external files.

See [Files](#) for more information.

Collaboration Diagrams

Collaboration diagrams, like sequence diagrams, display objects, their messages, and their relationships in a particular scenario or use case. Sequence diagrams emphasize message flow and can indicate the time sequence of messages sent or received, whereas collaboration diagrams emphasize relationships between objects.

Collaboration Diagrams Overview

Collaboration diagrams depict classifier roles and their interactions, or messages, via their association roles. A *classifier role* is an instance of a class (or classifier), that is defined only in the context of the collaboration. A classifier can be an object, a multi-object, or an actor. Similarly, an *association role* can be an instance of an association between the two classes and is the link that carries messages between the two classifier roles. This link is also limited to its purpose in the collaboration. In other words, the classifier and association roles are relevant only for that collaboration. An object can have different classifier roles in different collaborations; classifiers can exchange different sets of messages across different association roles.

In addition, collaboration diagrams display the messages passed across association roles. Messages are generally instances of class operations. They are numbered to indicate sequence order; they can be subnumbered (for example, 1a., 1b., 1.1.2, 1.1.3, 2.3a.1., 2.3a.2., and so on) either to indicate tasks that occur simultaneously or that are subtasks that achieve a larger task.

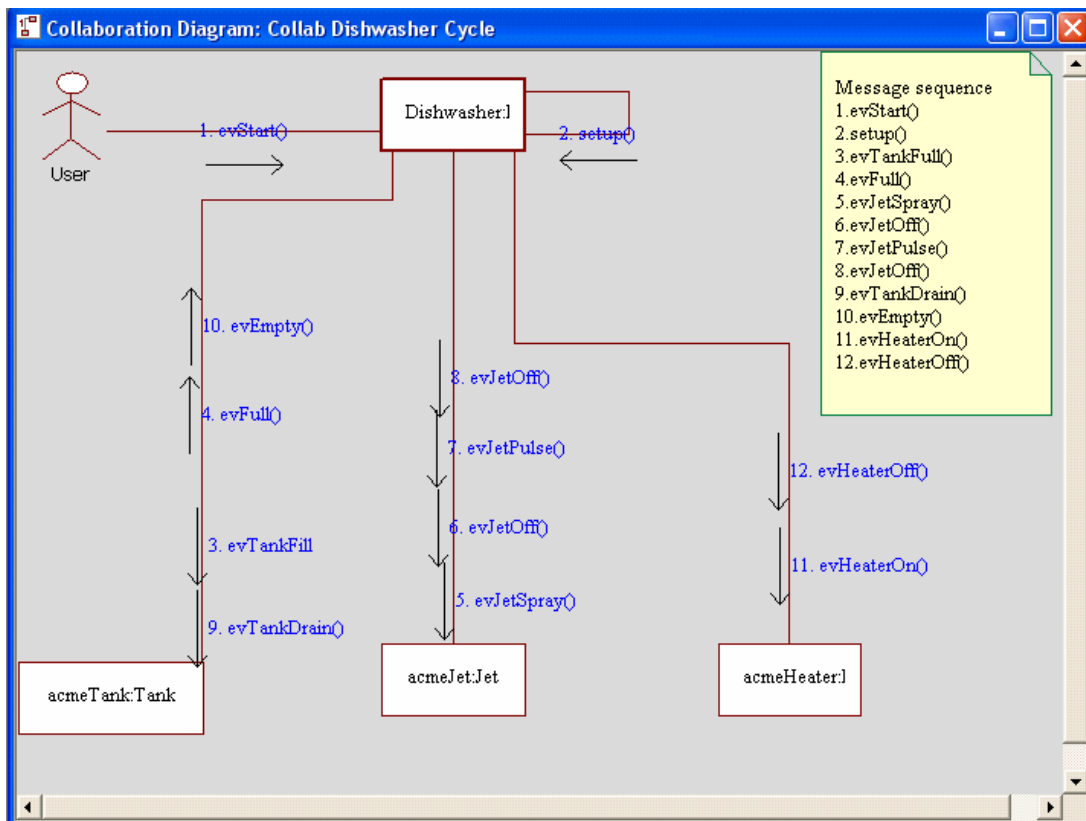
A numbering system that indicates parallelism might look like the following:

1. Make sandwich.
 - 1a. Get jam.
 - 1b. Cut bread.

A numbering system that indicates subtasking might look like the following:








1. Make sandwich.
 - 1.1 Get jam.
 - 1.2 Cut bread.
 - 1.3 Spread jam on bread slices.

Classifier roles, association roles, and messages are not displayed in the browser; however, the underlying classes and operations that they realize are displayed. The following figure shows a collaboration diagram.



Collaboration Diagram Toolbar

The **Drawing** toolbar for a collaboration diagram contains the following tools:


Drawing Icon	Description
	Object creates a new classifier role. A classifier role can be an instance of an existing class, a new class that you create in the collaboration diagram, or <Unspecified>, meaning that it is not a realization of a class. This could be useful if you create collaboration diagrams at the high-level analysis stage of system design. For more information, see Classifier Roles .
	Multi Object creates a classifier role for a set of objects, which means that the classifier role has a set of operations and signals that addresses the <i>entire</i> set of objects, and not just a single object. In other words, the classifier role represents a set of objects that share a common purpose in the scenario or use case described by the collaboration diagram. For more information, see Multiple Objects .
	Actor creates an actor, which represents an element that is external to the system. A classifier role based on an actor represents a coherent set of operations and messages of an external element when it interacts with system elements during a scenario. For more information, see Actors .
	Link draws a message link, or association role, between two classifier roles. Optionally, you can give the association role a name, perhaps to indicate the type of communication that occurs over this link. The association role can be an instance of an existing association between the two classes (from which the classifier roles are realized). For more information, see Link Messages and Reverse Link Messages .
	Link Message adds a message to the link between two classifier roles. The Message tool creates a message pointing toward the second classifier role in the link. For more information, see Link Messages and Reverse Link Messages .
	Reverse Link Message adds a reverse message to the link between two classifier roles. The Reverse Link Message icon creates a message pointing toward the first classifier role in the link. For more information, see Link Messages and Reverse Link Messages .
	Dependency creates a dependency between classifier roles.

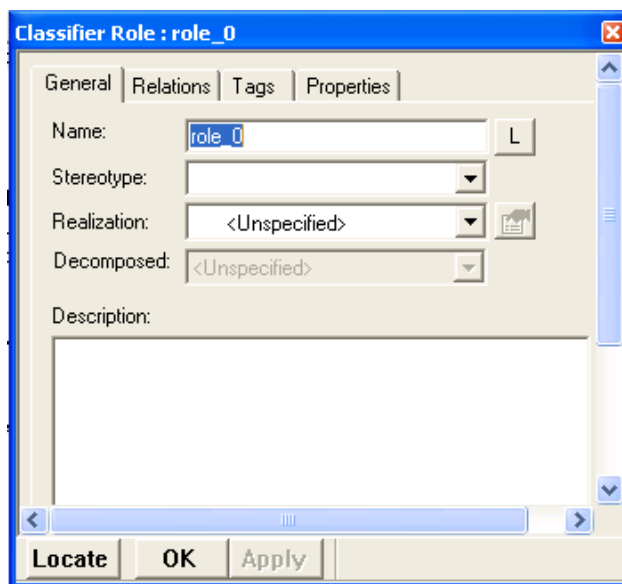
The following sections describe how to use these tools to draw the parts of a collaboration diagram. See [Graphic Editors](#) for basic information on diagrams, including how to create, open, and delete them.

Classifier Roles

A classifier role can be an instance of an existing class, a new class that you create in the collaboration diagram, meaning that it is not a realization of a class. This could be useful if you create collaboration diagrams at the high-level analysis stage of system design.

To create a classifier role, follow these steps:

1. Click the **Object** icon  .
2. Click-and-drag to create the new classifier role.
3. You may type the classifier role name in the drawing or right-click the object to display the features dialog (shown here) and type the name and additional information.



4. Click OK to save the information and close the dialog box.


By default, a new classifier role has an **<Unspecified> Realization** value. To make it an instance of a **<New >** or existing package, use the pull-down menu in the Features dialog box. See [Modifying the Features of a Classifier Role](#) more information.

Multiple Objects

A multiple object signifies that the classifier role has a set of operations and signals that addresses the *entire* set of objects, not just a single object. In other words, the classifier role represents a set of objects that share a common purpose in the scenario or use case described by the collaboration diagram. A multiple object can represent multiple instances of one or more classes or object types that share a common purpose in the scenario. As with an object, a multiple object can be `<Unspecified>` in this representation.

Creating a classifier role for a multiple object involves the same steps as creating a classifier role for a single object.

To create multiple objects, follow these steps:


1. Click the **Multiple Object** icon  on the **Drawing** toolbar for the collaboration diagrams.
2. In the drawing area, click (or click-and-drag) to create the multiple object.
3. Use the features dialog or edit the default name in the drawing.
4. Press **Enter** to save the name change in the drawing.

Actors

An actor represents an element that is external to the system. A classifier role based on an actor represents a coherent set of operations and messages of an external element when it interacts with system elements during a scenario. You can choose an existing actor, create a new one, or set the classifier role to `<Unspecified>`, meaning that the actor is not a realization of an existing actor. This could be useful in diagrams created at the high-level analysis stage of system design. Actors are a stereotype of a class and are defined through a Features dialog box that is, for the most part, the same as that for classes.

If you create a new actor, the Features dialog box opens so you can define the actor.

To create an actor, follow these steps:

1. Click the **Actor** icon .
2. In the drawing area, click (or click-and-drag) to create the actor.
3. Use the features dialog or edit the default name in the drawing.
4. Press **Enter** to save the name change in the drawing.

By default, a new actor is an <Unspecified> **Realization** value. To make it an instance of a <New > or existing package, use the pull-down menu in the Features dialog box. See [Modifying the Features of an Actor](#) for more information.

Links

The **Link** tool draws a message link, or *association role*, between two classifier roles. Optionally, you can give the association role a name, perhaps to indicate the type of communication that occurs over this link. The association role can be an instance of an existing association between the two classes (from which the classifier roles are realized).


The association role of a link can be <Unspecified>, meaning that it is an unspecified association. This could be useful in design- and even detailed design-phase collaboration diagrams, because you can portray messages that are not passed through relations, such as communication with local or global variables (objects) or communication with variables passed as parameters of a method.

Association roles are themselves not directional, even if they are assigned a directional association. This is in keeping with the emphasis on message traffic, regardless of which class initiated the flow. Once you have created an association role, you can draw the messages that go across it.

Note

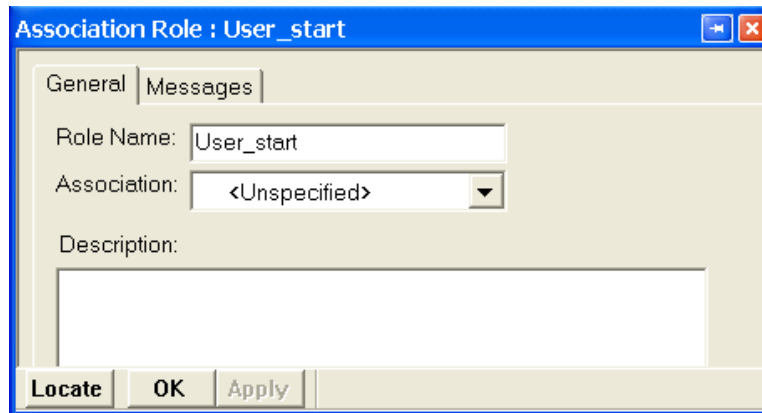
You can physically move an association role from one set of classifier roles to another, but this is not recommended because the connection of the association role to the association is lost. The association of an association role must be between the classes matched to the end classifier roles.

To create a link, follow these steps:

1. Click the **Link** icon .
2. Click in a classifier role.
3. Click in another classifier role. The link is drawn between the two classifier roles, and the cursor automatically opens the association role name text box.
4. If desired, type a name for the association role, then press **Enter**.

Modifying the Features of a Link

The Features dialog box enables you to change the features of a link, including the role name and association. The following figure shows the Features dialog box for a link.



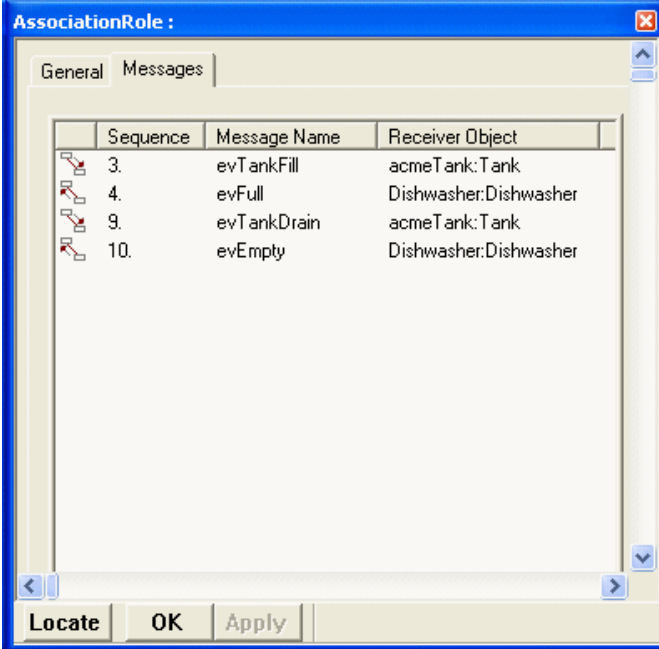
The **General** tab includes the following fields:

- ◆ **Role Name** specifies the name by which one class recognizes the other class.
- ◆ **Association** specifies the association being instantiated by the link.

Rhapsody allows you to specify a link *without* having to specify the association being instantiated by the link. Until you specify the association with the pull-down menu, this field is set to <Unspecified>.


- ◆ **Description** allows the user to add more detailed information about the association role.

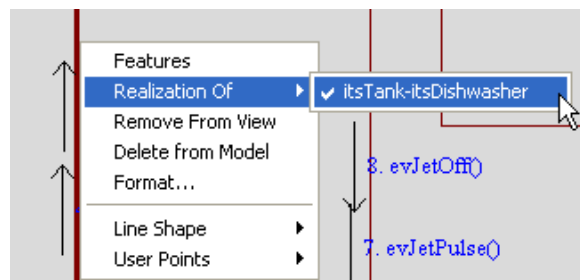
The **Messages** tab of the Features dialog box for the link lists the messages sent across the link, as shown in the following figure.



Changing the Underlying Association

You can set the association on which the association role is based using the Features dialog box. You can also change the association via the pop-up menu, as follows:

1. Click the **Select**  icon.
2. Highlight the association role and right-click it.
3. From the pop-up menu, select **Realization Of**. A drop-down menu of available associations is displayed, as shown in the following figure.



4. Highlight one of the associations, then click.

Link Messages and Reverse Link Messages

Once a link is created between two classifier roles, you can add messages to it. Link messages are numbered automatically, but can be edited and renumbered, for example, using a subnumbering system.

You need to use two icons to create the link messages in collaboration diagrams:



The **Link Message** icon creates a message pointing toward the second classifier role in the link.



The **Reverse Link Message** icon creates a link message pointing in the other direction.

Like classifier and association roles, messages can be `<Unspecified>`, meaning that they are abstract and not realizations of class operations. Link messages can be instances of existing operations of a class or instances of new operations. However, for a link message to realize some operation, the operation must be a method of the class associated with the target of the message.

Messages, whether abstract or instances of operations, have the notation `ReturnValue = MessageName(Arg, Arg, Arg...)`. You can use this notation in the message name when you first create it, or you can fill in these boxes explicitly in the Features dialog box.

Note that a message that is an instance of an operation does not necessarily show the form of the actual call. You can specify just the items of interest in the collaboration. The `ReturnValue` is optional; the function might not return a value, or you might not want to specify the local variable to which the return value applies.

To create a message, follow these steps:

1. Click either the **Link Message** or **Reverse Link Message** icon. The cursor changes to a small arrow pointing upwards.
2. Move the point of the arrow onto the association role, then click with the left mouse button. A text box opens, containing an automatically generated number.
3. Type the name of the message. If desired, you can change the numbering; the autonumbering will continue from whatever number you specify.

Note: If you edit the number, make sure the numbering sequence ends with a period (.) to clearly delineate it. If the period is missing, Rhapsody will not autonumber the messages correctly.

4. Press **Enter** to complete the name.

By default, the new message is `<Unspecified>`. To make it an instance of a new or existing operation of the target class or actor, open its Features dialog box (see [Modifying the Features of a Message](#)).

Component Diagrams

You use *component diagrams* to create new or existing components, specify the files and folders they contain, and define the relations between these elements. These relations include the following:

- ◆ **Dependency**—A relationship in which the proper functioning of one element requires information provided by another element. In a component diagram, a dependency can exist between any component, file, or folder.
- ◆ **Interface**—A set of operations that publicly define a behavior or way of handling something so knowledge of the internals is not needed. Component diagrams define interfaces between components only.

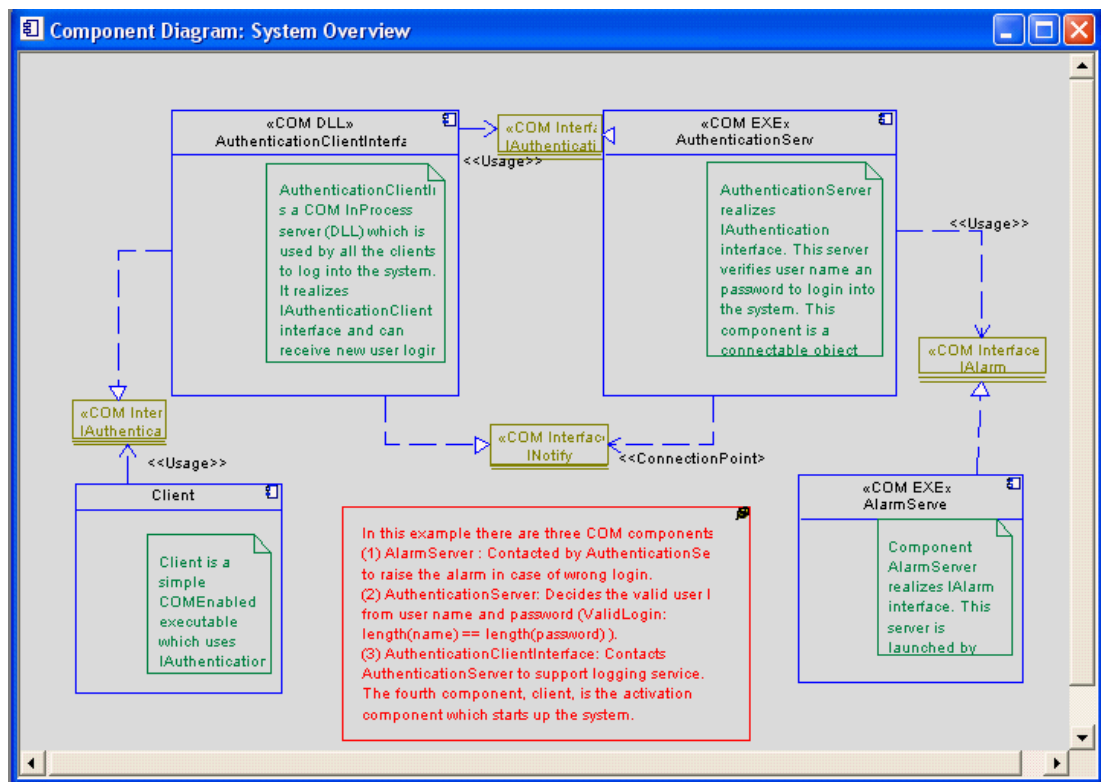
A *component* is a physical subsystem in the form of a library or executable program or other software components such as scripts, command files, documents, or databases. Its role is important in the modeling of large systems that comprise several libraries and executables. For example, the Rhapsody application itself is made up of many components, including the graphic editors, browser, code generator, and animator, all provided in the form of a library.

Component Diagram Uses

Component diagrams are helpful in defining and organizing the physical file hierarchy of your model. You can assign model elements to be contained in certain files, instead of using the default Rhapsody designations; you can organize files into folders or into components directly and organize folders into components.








One aspect of a component that is *not* included in a component diagram, but is included in this section, is how to create the configurations that are part of a component. Configurations specify how the component should be built, such as the target environment, initialization needed, and checks to perform on the model before code is generated. See [Configurations](#) for information.

The following figure shows a component diagram.



Component Diagram Drawing Icons

The **Drawing** toolbar for a component diagram contains the following tools.

Drawing Icon	Description
	Component specifies all the software code that it comprises: libraries, header files, and any other source files. See Components for more information.
	File specifies which model elements are generated in each file, the file names, and the directory paths. See Files for more information.
	Folder Component physically organizes files or other folders. See Folders for more information.
	Dependency shows when one element depends on the existence of another element. See Dependencies for more information.
	Interface creates an interface between components is a set of operations performed by a hardware or software element in the system. See Component Interfaces and Realizations for more information.
	Realization indicates that a component realizes an interface if it supports the interface. Then another component uses that interface.
	Flow provides a mechanism for specifying exchange of information between components. See Flows for more information.

The following sections describe how to use these tools to draw the parts of a component diagram. See [Graphic Editors](#) for basic information on diagrams, including how to create, open, and delete them.

Elements of a Component Diagram

Component diagrams contain the following elements:

- ◆ [Components](#)
- ◆ [Files](#)
- ◆ [Folders](#)
- ◆ [Dependencies](#)
- ◆ [Component Interfaces and Realizations](#)
- ◆ [Flows](#)


The following sections describe these elements in detail.

Components

When you create a component, you specify all the software code that it comprises: libraries, header files, and any other source files. Component diagrams generate code for components that are labeled with the «Executable» or «Library» stereotype.

Creating a Component

To create a component, follow these steps:

1. Click the **Component** icon  on the **Drawing** toolbar.
2. Do one of the following:
 - a. Click once in the diagram to create a component with the default dimensions.
 - b. Click to begin the upper, left corner of the component, drag to the lower right corner, and release.
3. Edit the default name, then press **Enter**.

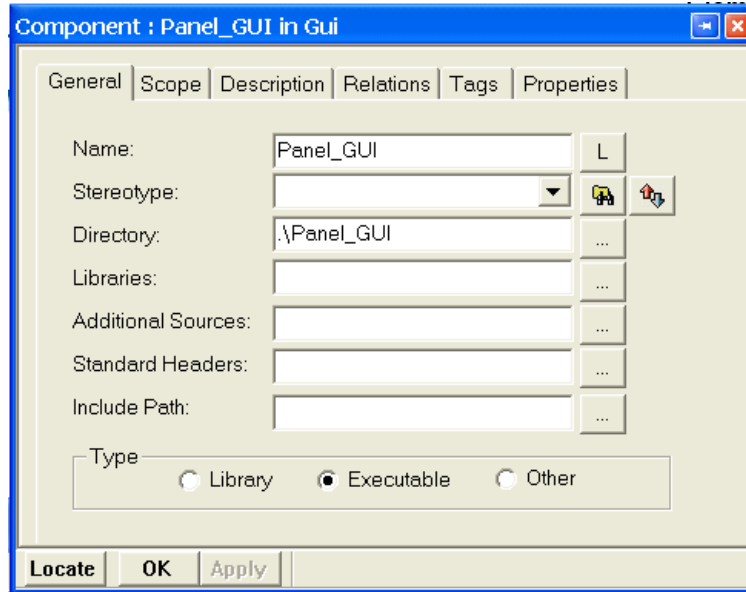
The new component is displayed in the diagram and in the browser, either in the `Components` folder under the main project node, or nested under another component associated with this diagram.

Note

You can draw a component within another component; the browser will display the nested component accordingly.

Modifying the Features of a Component

The Features dialog box enables you to define a component, as shown in this example.



General Features

- ◆ **Name**—Specifies the name of the component.
- ◆ **L**—Specifies the label for the element, if any.
- ◆ **Stereotype**—Specifies the stereotype of the component, if any. They are enclosed in guillemets, for example «s1» and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Note that the COM stereotypes are constructive; that is, they affect code generation. Refer to the *COM API Development Guide* for more information about these components.

- ◆ **Directory**—Specifies the root directory for all configurations of the component. This can be the project directory or another directory.
- ◆ **Libraries**—Specifies any additional libraries to be added to the link. This field is relevant for executables only. Libraries can be off-the-shelf libraries, legacy code, or Rhapsody-generated libraries.
- ◆ **Additional Sources**—Specifies external source files to be compiled with the generated source files. Rhapsody adds the files to the project makefile.
- ◆ **Standard Headers**—Specifies header files to be added to `include` statements in every file generated for the project. Specify either a full path or, if only a filename, add a path in the **Include Path** field.

- ◆ **Include Path**—Specifies the directory in which the include files are located.

Note that this field supports environment variables, such as `$ROOT\Project\ExternalCode`.

- ◆ **Type**—Specifies the build type. Select **Library**, **Executable**, or **Other**. Rhapsody does not generate code for a component that has build type “other,” nor can such a component be set as the active component. “Other” could be used to designate script files or other non-code files.

Component Scope

The **Scope** tab of the Component Features dialog box allows you to specify which model elements should be included in the component.

If you select the **Selected Elements** radio button, you can use the check boxes next to each element to indicate which model elements should be included in the component.

If you select a check box for an element, all of the elements that it contains are included in the component scope (for example, all of the classes in a package). If you would like to be able to select sub-elements individually, right-click the check box of the parent element.

Files

Files owned by a component are compiled together to build the component. You can specify which model elements are generated in each file, the file names, and the directory paths. You can also create a file in the browser and drag-and-drop it into a component diagram.

A file must be nested within a component or a folder.

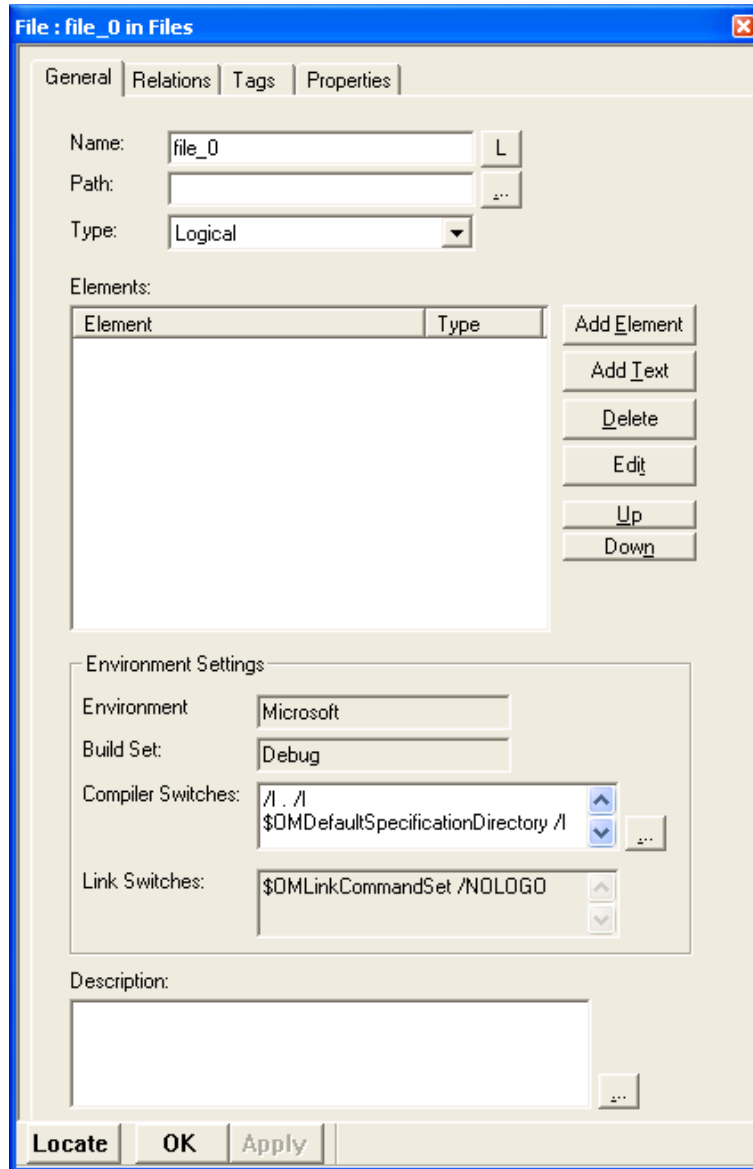
Note

However, a file cannot contain another element.

Creating a File

To create a file, follow these steps:

1. Right-click a Component in the browser and select **Add New > File**. The File dialog displays, as shown here.



2. Begin to define this new file by typing the Name you want to replace the system generated name. Click **Apply** to save the name and keep the dialog open. The new file name displays in the diagram and in the browser, under the component with which it is associated.

Note

A file must be an element of a component or a folder. If the component diagram is under the project node, it is not yet associated with a component. First create a component, then nest the file by drawing it inside the component. If the diagram is already nested under an existing component, you can draw the file in the “free space” of the diagram editor.

3. Change the remaining fields to define the file as you wish.
 - ♦ **Path**—Specifies where a file should be generated in relation to the configuration directory. If this field is blank, the file is generated in the configuration directory.
 - ♦ **File Type**—Specifies the type of file that should be generated. A specification file contains the specifications of all elements; an implementation file contains their implementations. They are specified by their suffixes, as follows:

File Type	C++	C	Java
Implementation	.cpp	.c	.java
Specification	.h	.h	N/A

Choose one of the following values:

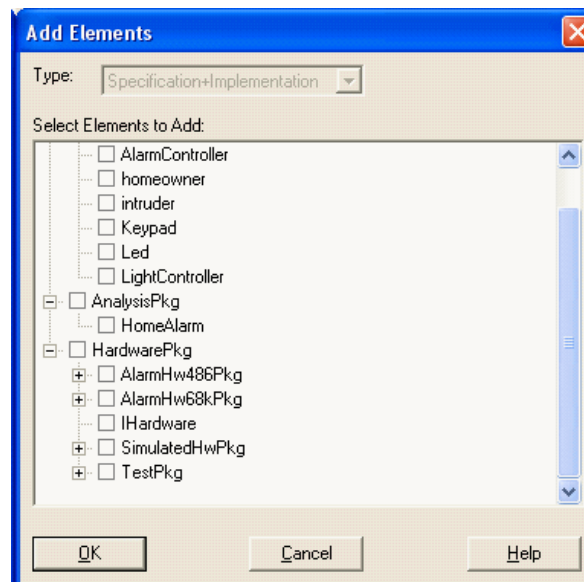
- **Logical**—Create both implementation and specification files.
- **Specification**—Generate only a specification file. Both specifications and implementations of all elements assigned to this file are generated into this file.
- **Implementation**—Generate only an implementation file. Both specifications and implementations of all elements assigned to this file are generated into this file.
- **Other**—One file is generated with the name and extension specified in the **Name** field. Both specifications and implementations of all elements assigned to this file are generated into the file.
- ♦ **Elements**—Lists the elements mapped to a file. Elements that are not explicitly mapped to files are generated in the default files that Rhapsody would normally generate for these elements in the configuration directory.
- ♦ **Environment Settings**—Rhapsody fills in the settings from your environment. The fields are as follows:
 - **Environment**—This read-only field specifies which environment (Microsoft, Solaris2, and so on) is selected for the active configuration. You cannot change the environment for an individual file.

- **Build Set**—This read-only field specifies the build setting (Debug or Release mode) for the active configuration. You cannot change the build setting for an individual file.
 - **Compiler Switches**—This field specifies the compiler switches for the configuration. Compiler switches default to those used for the configuration, but you can override them for an individual file.
 - **Link Switches**—This field specifies the link switches used to link the active configuration. You cannot change link switches for an individual file.
 - ◆ **Description**—Describes the element. This field can include a hyperlink. See [Hyperlinks](#) for more information.
4. Click **OK** to save your selections.

Adding an Element to a File

To add a package or class to a file, follow these steps:

1. In the dialog box, click **Add Element**. The Add Elements dialog box opens, as shown in the following figure.



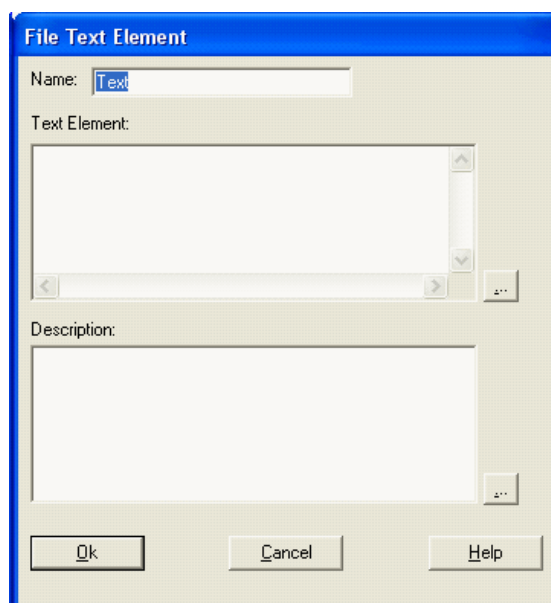
The **Type** box lists the file type you selected in the file Features dialog box. You can change the file type here if it is not logical. Logical files can contain only logical elements, but other types of files can contain whatever you want. See [Using the Features Dialog Box](#) for more information.

2. Select the elements that you want to map to the file. Note that selecting a package automatically maps all its classes to the file.
3. Click **OK**.

Adding Text to a File

To add a text element to a file, follow these steps:

1. In the file Features dialog box, click **Add Text**. The File Text Element dialog box opens, as shown in the following figure.



2. If desired, edit the default name of the text element in the **Name** field.
3. In the **Text Element** field, type the text you want to add to the file. For example, you can add `#ifdef` statements or `#define` statements, headers and footers, or additional comments.
4. In the **Description** field, enter a description of the text element.
5. Click **OK** to apply your changes and close the dialog box.

Deleting an Element from a File

To delete an element from a file, follow these steps:

1. In the file Features dialog box, select the element to delete.
2. Click **Delete**. Rhapsody asks you to confirm that you want to remove the element from the file.
3. Click **Yes** to delete the element.

Editing an Element

To edit an element in a file, follow these steps:

1. In the file Features dialog box, select the element you want to edit.
2. Click **Edit**. The File Text Element dialog box opens.
3. Make the appropriate changes.
4. Click **OK** to apply your changes and close the dialog box.

Rearranging Elements in a File

You have explicit control over the order in which elements are generated in files. Moving an element up or down in the list means that it will be generated earlier or later in the file.

To rearrange elements in a file, follow these steps:

1. In the file Features dialog box, select the element you want to move.
2. Click **Up** to generate the element earlier in the file, or **Down** to generate the element later in the file.

Files Pop-Up Menu

- ◆ **Add New**—Enables you to add a new dependency, constraint, comment, requirement, hyperlink, or tag to the file.
- ◆ **Generate File**—Generates the file from the elements assigned to it. The number and type of files generated depends on the file type you have selected. See [Using the Features Dialog Box](#).
- ◆ **Edit File**—Opens the generated files in a text editor.

You can select an external text editor using the `EditorCommandLine` property under `General::Model`. See [Using an External Editor](#) for more information.

Folders

Folders, or directories, help physically organize files. A folder must be nested within a component or another folder. A folder can contain files or folders.

Creating a Folder

1. Click the **Folder** tool.
2. Do one of the following:
 - a. Click once in the diagram to create a folder with the default dimensions.
 - b. Click to begin the upper, left corner of the folder, drag to the lower, right corner, and release.
3. Edit the default name, then press **Enter**.

The new folder is displayed in the diagram and in the browser under the component or folder with which it is associated.

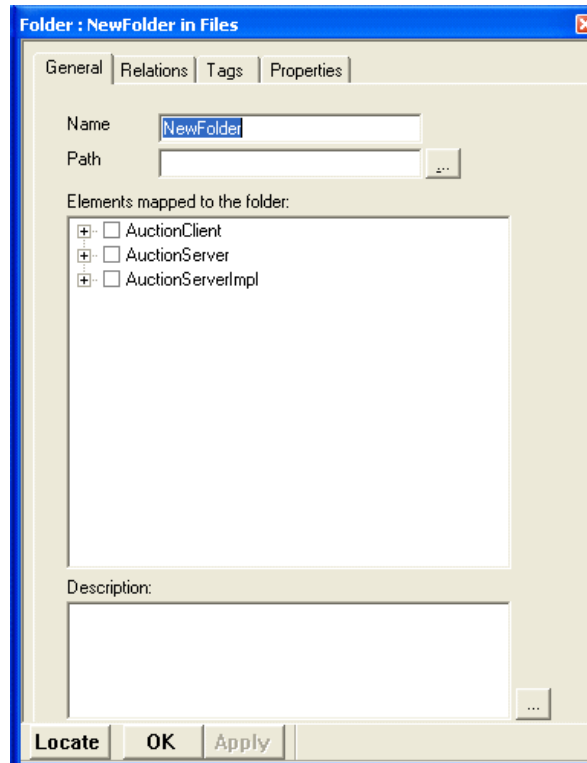
Note

A folder must be nested within a component or another folder. If the component diagram is under the project node, it is not yet associated with a component. You must first create a component, inside of which you can then draw a folder. If the diagram is already nested under an existing component, you can draw the folder in the “free space” of the diagram editor; it is nested within the component associated with the diagram and is displayed accordingly in the browser.

In the browser, folders are located under the components of which they are part.

Modifying the Features of a Folder

The Features dialog box enables you to change the features of a folder, including its name and path. The following figure shows the Features dialog box for a folder.



A folder has the following features:

- ◆ **Name**—Specifies the name of the folder. The default name is `folder_n`, where *n* is an incremental integer starting with 0.
- ◆ **Path**—Specifies where the folder should be generated in relation to the configuration directory. Folders are generated as subdirectories under the configuration directory.
- ◆ **Elements mapped to the folder**—Specifies the elements you want to map to a folder. Rhapsody generates the default types of files it normally generates for these elements in the folder if the elements are not specifically mapped to other files.
- ◆ **Description**—Describes the folder. This field can include a hyperlink. See [Hyperlinks](#) for more information.

Note that the Features dialog box for folders is available only for folders that you add to the configuration—not for the top folder under the component.

Folders Pop-Up Menu

- ◆ **Add New**—Invokes a cascading menu that allows you to add elements to the folder. You can add the following items:
 - **Folder**—Adds a subdirectory to the current directory.
 - **File**—Adds a file to the current directory.

Dependencies

A dependency exists when the functioning of one element depends on the existence of another element. A dependency between two components in a component diagram results in an `#include` statement in the make file for the dependent (or client) component.

Dependencies in component diagrams have the same stereotype values as dependencies created in OMDs. See [Dependencies](#).

In a component diagram, a dependency relation is also used in the definition of a component interface. See [Creating a Component Interface](#) for more information.

Dependencies appear in the browser under the dependent, or client component.


Component Interfaces and Realizations

An *interface* between components is a set of operations performed by a hardware or software element in the system. A component realizes an interface if it supports the interface; another component then uses that interface. Interfaces promote design modularity; components are more easily replaceable when they use interfaces instead of directly depending on components.

Component interfaces can be seen only in a component diagram—they cannot be viewed in the browser. A component diagram supports only interfaces between components.

Creating a Component Interface

To create an interface, follow these steps:

1. Click the **Interface** icon  on the **Drawing** toolbar.
2. Click once in the diagram, or click-and-drag to create the component interface.
3. By default, Rhapsody creates an interface named `Interface_n`, where `n` is an integer value starting with 0. If desired, rename the interface. The following example shows a component interface. Rhapsody adds the `<<Interface>>` stereotype automatically.



Flows

Flows provide a mechanism for specifying exchange of information between system elements at a high level of abstraction. This functionality enables you to describe the flow of data and commands within a system at a very early stage, before committing to a specific design. As the system specification evolves, you can refine the abstraction to relate to the concrete implementation.

Setting Component Configurations in the Browser

In the browser, components are displayed under the project main node or other components. There are a number of component features that you can set via the component pop-up menu in the browser that you cannot do in the component diagram. For example, in the browser you can set a component as the active configuration or define the configuration settings for the component.

When you highlight a component in the browser, the Features dialog box opens. See [Modifying the Features of a Component](#) for more information.

Modifying and Working with Components

When you select a component in the browser, the pop-up menu includes the following component-specific options:

- ◆ **Add New**—Invokes a cascading menu that allows you to add the following items to the component:
 - **Dependency** specifies the element on which the component depends, such as a package that is not part of the component.
 - **Derivation** adds a requirement that was derived from another
 - **Constraint** defines a semantic condition or restriction.
 - **Comment** allows you to add a textual annotation that does not add semantics, but contains information that might be useful to the reader and is displayed in the browser.
 - **Requirement** allows you to add a textual annotation that describes the intent of the component. Requirements are part of the model and are therefore displayed in the browser.
 - **Controlled File** allows you to add a file or reference purposes that was produced in other programs, such as Word or Excel. These files become part of the Rhapsody project and are controlled by it.
 - **Hyperlink** adds internal links to Rhapsody elements or external links to a URL.
 - Tag
- ◆ **Set as Active Component** makes the selected component the active one. See [Active Component](#) for more information.
- ◆ **Generate Component** makes the selected component the active one, then generates the active configuration.
- ◆ **Build Component** builds the active configuration.

See [Editing Elements](#) for information on the common operations available through the pop-up menu.

Active Component

The active component is the one built when you make the code. The icon for the active component includes a red checkmark, as shown in the figure.



To make a component active, do one of the following:

- ◆ Select the component from the drop-down list on the toolbar.
- ◆ In the browser, right-click the component and select **Set as Active Component**.

When you change the active component, the most recent active configuration within the component becomes the active configuration and is listed in the Current Configuration list in the **Code** toolbar.

Note

To become the active component, a component must be set to either the **Executable** or **Library** build type.

Configurations

If a component is a physical subsystem, as in a communications subsystem, a configuration (module) specifies how the component is to be produced. For example, the configuration determines whether to compile a debug or non-debug version of the subsystem, whether it should be in the environment of the host or the target (for example, Windows NT versus VxWorks), and so on.

A component can consist of several configurations. For example, if you want to build a VxWorks version and a pSOSystem version of the same component, you would create two configurations under the component, one for each operating system. The decision as to whether these should be two separate components or two configurations within the same component depends on whether you want to compile them differently, or whether there is some logical variation between them. Creating two separate components would require maintaining two separate implementation views, whereas using separate configurations would not.

For information on setting configuration parameters, see [Modifying the Features of a Configuration](#).

Configuration Pop-Up Menu

If you right-click a configuration in the browser, the pop-up menu contains the following configuration-specific options:

- ◆ **Set as Active Configuration**—Makes the configuration active.
- ◆ **Edit Makefile**—Enables you to edit the makefile generated for the configuration in a text editor.
- ◆ **Edit Configuration Main File**—Edits the main file generated for the component. This option is available for an executable or library component that has at least one package with a global instance in its scope. See [Making Permanent Changes to the Main File](#) for more information.
- ◆ **Generate Configuration Main and Make Files**—Sets the configuration as the active configuration and generates the main file and the makefile.
- ◆ **Generate Configuration**—Makes the configuration active and generates it.
- ◆ **Build Configuration**—Builds the active configuration.

Setting the Active Configuration

The active configuration is the one generated when you generate code, unless you are generating selected classes. The active configuration, or the current configuration, appears in a drop-down list in the **Code** toolbar. You can change the active configuration using either the toolbar or the pop-up menu for the configuration.

To set the active configuration, do one of the following:

- ◆ Select the configuration from the Current Configuration drop-down list in the **Code** toolbar.
- ◆ Select the configuration in the browser. Right-click the configuration, and select **Set as Active Configuration** from the pop-up menu.

The name of the new active configuration is displayed in the Current Configuration list in the **Code** toolbar. In addition, the component that owns this configuration becomes active and is displayed in the Current Component list (see [Active Component](#)).

Modifying the Features of a Configuration

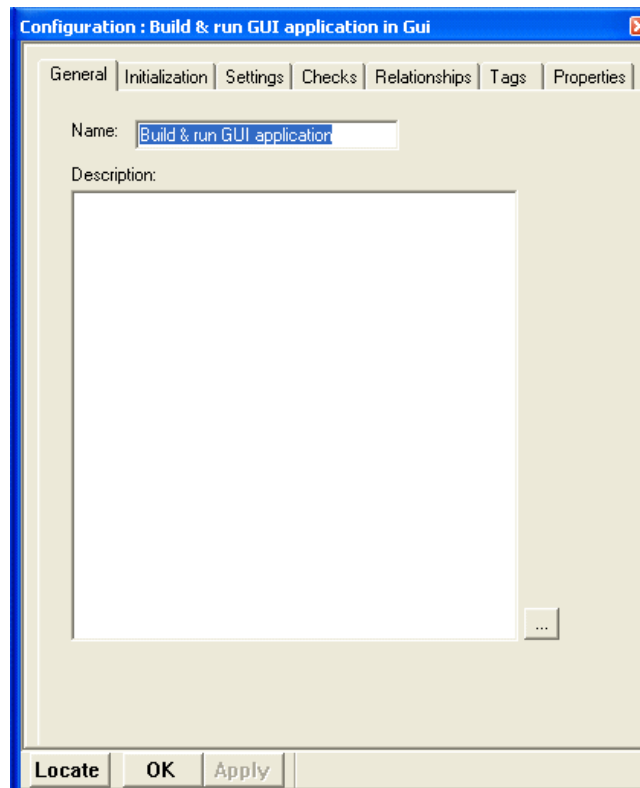
The Features dialog box for a configuration contains the following tabs:

- ◆ [General Tab](#)
- ◆ [Initialization Tab](#)
- ◆ [Settings Tab](#)
- ◆ [Checks Tab](#)
- ◆ [Relations Tab](#)
- ◆ [Tags Tab](#)
- ◆ [Properties Tab](#)

These tabs display configurable features and are described in detail in the following sections.

General Tab

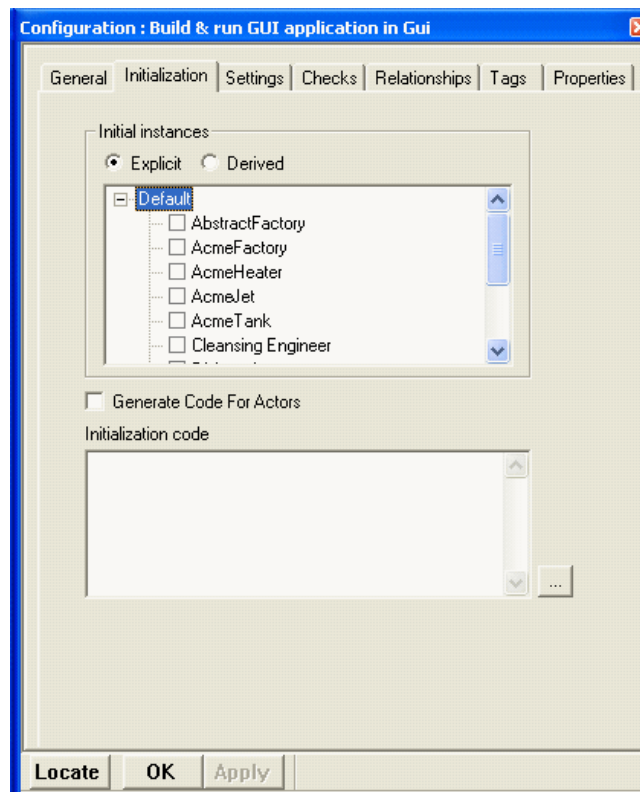
The **General** tab, shown in the following figure, enables you to define general information for the configuration.



- ◆ **Name**—Specifies the name of the configuration. The default name for configurations is `configuration_n`, where *n* is an incremental integer starting with 0.
- ◆ **Description**—Describes the configuration (for example, the target environment).

Initialization Tab

The **Initialization** tab, shown in the following figure, enables you to specify which instances to initialize and whether to generate code for actors.



- ◆ **Initial instances**—Selecting initial instances adds code to the main program to instantiate only those packages, classes, and actors that you specify. The possible values are as follows:
 - **Explicit**—Instantiate only the selected elements.
 - **Derived**—Instantiate the selected elements and any others to which these are related, either directly or indirectly.

For example, if class A is selected, and class B is related to A, B is added to the derived scope. If C is related to B, C is also added to the derived scope, and so on.

Two elements are related if there is a dependency or relation between them, use types of the other element, or use events defined in the other element.

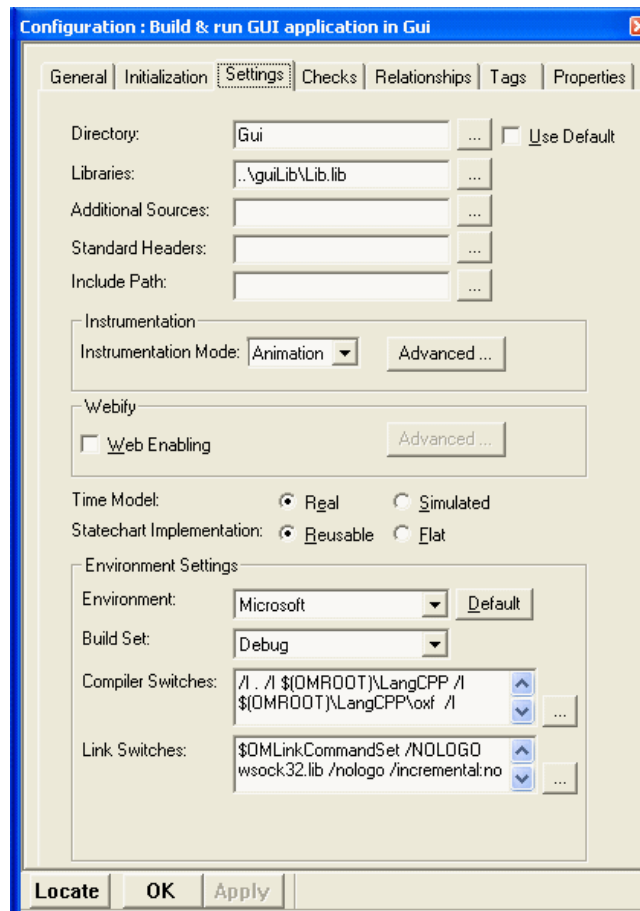
- ◆ **Generate Code For Actors**—If this box is checked (the default), code is generated for the actors specified in the **Initial Instances** box. See [Generating Code for Actors](#) for more information about this feature.
- ◆ **Initialization code**—Type any user code you want to use to instantiate initial instances or to initialize any other elements. This code is added to the main program after any automatically generated initialization code and before the main program loop.

Settings Tab

The **Settings** tab, shown in the following figure, enables you to specify numerous settings for the configuration.

Note

The values of each of these fields are appended to the settings fields of the component that owns the configuration.



The **Settings** tab contains the following fields:

- ◆ **Directory**—Specifies the root directory for files generated for the configuration. This box is enabled only if the **Use Default** option is not checked. You can specify either a full path or a partial path that uses the current directory as the starting point.
- ◆ **Use Default**—Check this box to use the default directory for the configuration. The default location is named after the configuration and is a subdirectory of the project directory.
- ◆ **Libraries**—Specifies additional off-the-shelf, legacy code, or Rhapsody-generated libraries to be added to the link. This box is relevant only for executables.
- ◆ **Additional Sources**—Specifies the external source files to be compiled with the generated sources. Rhapsody adds these files to the project makefile.
- ◆ **Standard Headers**—Specifies the header files to be added to `#include` statements in every file generated for the project. Specify a full path or the file name. If you specify only the file name in this field, specify the directory in the **Include Path** field.
- ◆ **Include Path**—Specifies the directory in which the include files for a configuration are located. The include path is added to the makefile generated for a configuration. For example, if you set the include path to `d:\Rhapsody\MMM`, the following code is generated in the makefile:

```
INCLUDE_PATH= \  
$(INCLUDE_QUALIFIER)d:\Rhapsody\MMM
```

- ◆ **Instrumentation**—Specifies whether the executable will have animation or tracing capabilities, or neither. Select the appropriate value from the **Instrumentation Mode** drop-down list.

Click the **Advanced** button to specify the *instrumentation scope*, which determines the set of Rhapsody classes, packages, and actors that are instrumented in the associated configuration. This functionality enables you to enable or disable animation of classes (or entire packages) without changing the model elements themselves.

See [Using Selective Instrumentation](#) for more information.

- ◆ **Webify**—Specifies whether to Web-enable the configuration. See [Managing Web-enabled Devices](#) for more information.

Note: You cannot webify a file-based C model.

- ◆ **Time Model**—Specifies real or simulated time. With real-time emulation, timeouts and delays are computed based on the system clock. With simulated time, a virtual timer orders timeouts and delays, which are posted whenever the system completes a computation.
- ◆ **Statechart Implementation**—Specifies whether the statechart implementation is Reusable or Flat (the default). The reusable model implements states as classes, whereas the flat model implements states as simple enumerated types. Reusable is preferable for

models with deep class inheritance hierarchies, whereas flat is preferable for models with shallow or no inheritance.

- ◆ **Environment Settings**—Specifies the following information about the target environment:
 - **Environment**—Specifies the target environment
 - **Build Set**—Specifies whether to generate a debug or non-debug (Release) version of the executable
 - **Compiler Switches**—Specifies the compiler switches applied by default when compiling each file
 - **Link Switches**—Specifies the link switches applied by default when linking the compiled code
 - **Additional Settings**—This button allows you to integrate CodeTEST[®] with Rhapsody in C and C++, if you have CodeTEST installed on your system. Otherwise, this button is unavailable (grayed out).

When you click this button, Rhapsody displays the Additional Settings dialog box. This dialog box contains the following fields:

- **With Applied Microsystems CodeTEST**—Click this check box to enable the integration with CodeTEST.
- **CodeTEST settings**—Enables you to edit the compilation switches that will be added to the CodeTEST instrumentation line in the generated makefile. The value of the CodeTEST settings field corresponds to the value of the `<lang>_CG::VxWorks::CodeTestSettings` property.

Checks Tab

The **Checks** tab enables you to specify which checks to be performed on the model before generating code. See [Checks](#) for detailed information on checks performed by Rhapsody.

Relations Tab

The **Relations** tab lists all the relationships (dependencies, associations, and so on) the configuration is engaged with. See [Defining Relations](#) for more information on this tab.

Tags Tab

The **Tags** tab lists the available tags for this configuration. See [Using Profiles](#) for detailed information on tags.

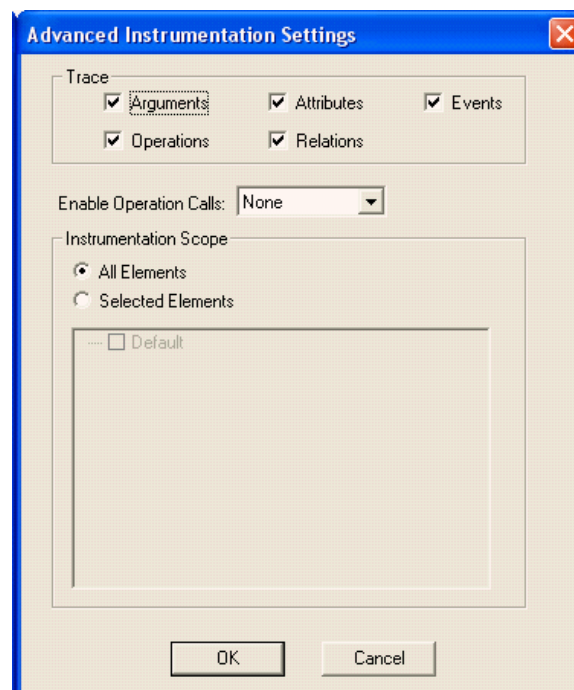
Properties Tab

The **Properties** tab enables you to set properties that affect the configuration. Refer to the *Rhapsody Properties Reference Manual* for detailed information on the available properties.

Using Selective Instrumentation

The dialog box for selective instrumentation enables you to select the specific model elements and element types to be instrumented for animation or tracing in the given configuration. Using this functionality, you can use partial animation or tracing without changing the properties of the specific model elements.

When you click the **Advanced** button in the **Instrumentation** group, the Advanced Instrumentation Settings dialog box opens, as shown in the following figure.



Using this dialog box, you can easily control whether instrumentation is enabled for model elements, operations, and classes and packages for each configuration. The base model (stored in the development tree) could be non-instrumented: to validate parts of the model, you would simply change the animation settings in this dialog box to enable or disable instrumentation.

The dialog box contains the following fields:

- ◆ **Trace**—Determines whether tracing is enabled for the different model element types—arguments, operations, attributes, relations, and events. This is equivalent to setting the `Animate` properties for the metaclass for the configuration.

For example, clearing the **Operations** check box sets the `CG::Operation::Animate` property for the configuration to `Cleared`, so animation/tracing will not monitor operations. You can override this behavior for a specific element by overriding the

property at the element level. For example, to monitor operations in a specific package after clearing the **Operations** check box, set `CG::Operation::Animate` to `Checked` for the specific package.

By default, all model types are selected for instrumentation.

- ◆ **Enable Operation Calls**—Specifies whether you can invoke operation calls from the **Animation** toolbar. The possible values are as follows:
 - **None**—Operation calls cannot be invoked.
 - **Public**—Only public methods can be invoked.
 - **Protected**—Only protected methods can be invoked.
 - **All**—All operation calls can be invoked, regardless of visibility.
- ◆ **Instrumentation Scope**—Specifies which model elements (classes, packages, and actors) to animate. By default, all model elements are selected.

When the **All Elements** radial button is selected, the behavior is as follows:

- Tree control is disabled (grayed out). Tree control is enabled when you click **Selected Elements**.

The tree view contains all the classes, actors, and packages in the component's scope whose `<lang>_CG::<Metaclass>::Animate` property is set to `Checked`. Note that external elements (`UseAsExternal` is `Checked`) cannot be in the scope of the component. When you select a package in this tree, you also select all its aggregated classes and actors.

- All the elements in the code generation scope are instrumented, unless their `Animate` property is set to `Cleared`.

The following table shows how the instrumentation scope and the `Animate` property determine whether an element is instrumented.

Value of the Animate Property	Set in the Instrumentation Scope?	Will the Element be Instrumented?
Checked	Yes	Yes
Checked	No	No
Cleared	Yes	No
Cleared	No	No

Note the following behavior:

- ◆ If the `Animate` property is set to `Cleared`, it applies to *all* configurations, regardless of the instrumentation scope.

- ◆ If you change the instrumentation scope, *all* the source files of the component are regenerated.
- ◆ When you select a class in the package, it is implied that the entire package is instrumented—including all the events, types, and so on—even if the class does not use them.

Making Permanent Changes to the Main File

1. In the browser, right-click the component whose main file you want to edit.
2. Select **Add New > File** to create a new file.
3. Name the new file (for example, `myMain`).
4. Invoke the Features dialog box for `myMain` and set the **File Type** field to Logical or Implementation.
5. Select the **Properties** tab.
6. Click **OK** to apply your changes and close the dialog box.
7. Right-click the component's active configuration (used to build the application), then select **Edit Configuration Main File** from the pop-up menu.
8. Copy the contents of this file, including the header files.
9. Invoke the Features dialog box for `myMain` and click **Add Text**.
10. Paste the code from the Rhapsody-generated main file into the **Text Element** field.
11. Customize the code as desired.
12. Click **OK** to apply your changes and close the dialog box.
13. Set the following properties for the configuration:
 - a. Set `CG::Configuration::MainGenerationScheme` to `UserInitializationOnly`. This property controls how the main is generated.
 - b. Set `<lang>_CG::<Environment>::EntryPoint` to `myMain`. This property specifies the name of the main program.

Now when you compile the application, Rhapsody will compile your customized `main` instead of generating a new one.

Creating Components under a Package

When you create a Rhapsody project, a component called *DefaultComponent* is created directly beneath the project level. You can also create additional components at this hierarchical level.

It is also possible to create a component as part of a package in the model. One of the advantages of this approach is that if you only want to generate code for a specific package, you only have to check out that package.

You can add a component to a package using any of the following methods:

- ◆ Create a new component in the package by right-clicking the package in the browser and selecting **Add New > Component** from the context menu.
- ◆ Move an existing component in the browser to the package.
- ◆ Draw a component in a component diagram that is located under a package.
- ◆ Create a new component in a package using the Rhapsody API.

When a component is created under a package, its default scope is the package to which it belongs.

Like other components, components that belong to packages can be assigned to be the active component for the project. When you create a new component in a package, it automatically becomes the active component for the project.

Note

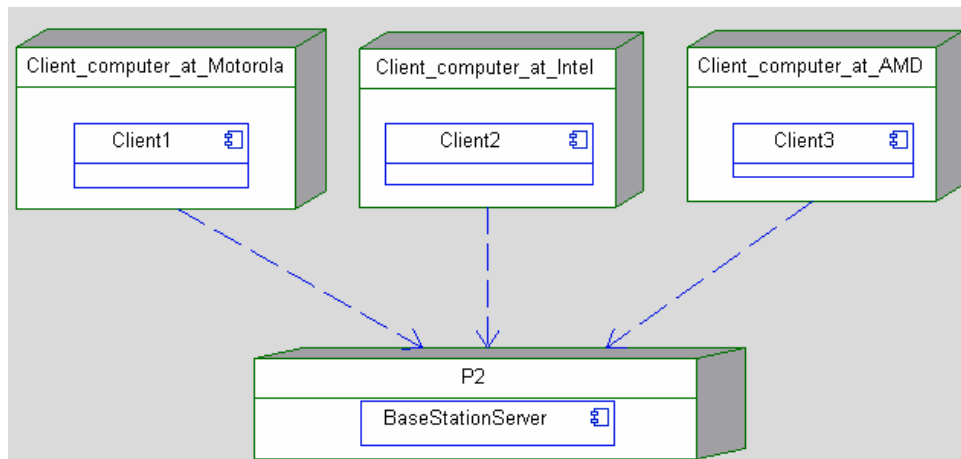
If you draw a component diagram under a package (rather than under the project), keep in mind that then it is not possible to draw a new folder on the diagram. This is because the folder element cannot be contained under a package.

Deployment Diagrams

Deployment diagrams show the configuration of run-time processing elements and the software component instances that reside on them. Use deployment diagrams to specify the run-time physical architecture of a system.





Deployment diagrams are graphs of nodes connected by communication associations. Component instances are assigned to run on specific nodes during program execution. Relation lines represent communication paths between nodes.

The following figure shows a deployment diagram.



Deployment Diagram Drawing Toolbar

The **Drawing** toolbar for a deployment diagram contains the following tools:

Drawing Icon	Description
	Node devices or other resources that store and process instances during run time. See Nodes for more information.
	Component represent executable processes, objects, or libraries that run or reside on processing resources (nodes) during program execution. See Component Instances for more information.
	Dependency represents a requirement by one component instance of information or services provided by another. See Dependencies for more information.
	Flow describes the movement of data and commands within a system. See Flows for more information.

The following topics describes how to use these tools to draw the parts of a deployment diagram. See [Graphic Editors](#) for basic information on diagrams, including how to create, open, and delete them.

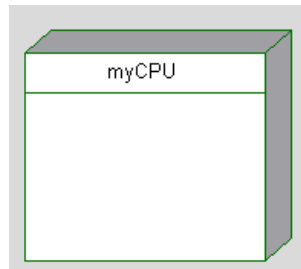
Nodes

Nodes represent devices or other resources that store and process instances during run time. For example, a node can represent a type of CPU. A node can be owned only by a package. In addition, nodes cannot be nested inside other nodes. Nodes can contain component instances.

Note

In Rhapsody, nodes represent UML node instances.

The graphical symbol for a node in the UML is a three-dimensional cube with a name, such as the name of a processor.




Creating a Node

You can create a node using the **Node** tool, Edit menu, or browser.

Using the Toolbar to Create a Node

To use the **Drawing** toolbar to create a node, follow these steps:

1. Click the **Node** icon  on the **Drawing** toolbar.
2. Click or click-and-drag with the mouse to place the node on the diagram. Rhapsody creates a node symbol with the default name of `node_n`, where *n* is an incremental integer starting with 0.

Using the Browser to Create a Node

To use the Rhapsody browser to create a node, follow these steps:

1. Depending on the method you want to use:
 - Right-click a package in the browser and select **Add New > Node**, or
 - Right-click a node category and select **Add New Node**.
2. Edit the default name of the new node.

3. With both the browser and the deployment diagram editor in view, click-drag-and-drop the node onto the diagram.

You can click-and-drag any node that exists in the browser to add it to a deployment diagram.

Changing the Owner of a Node

To change the package that owns a node, follow these steps:

1. In the Rhapsody browser, select the node.
2. Drag the node from its current package to a new package.

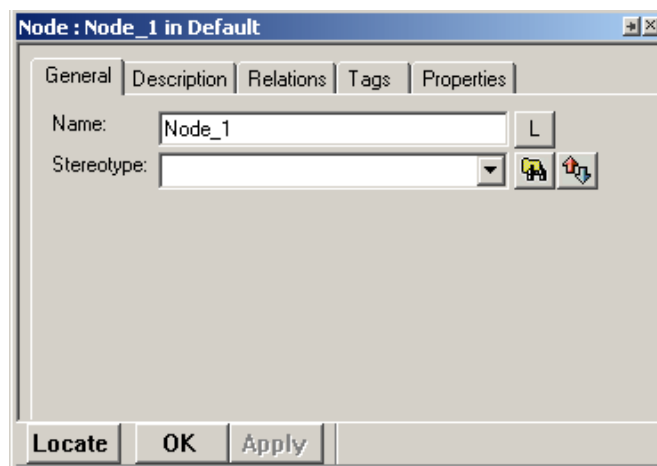
Designating a CPU Type

Nodes drawn in deployment diagrams represent specific node instances, rather than general node types. As such, a node should be given a specific name such as `myPersonalIntelPentium`, which describes a specific processor, rather than a general name such as `IntelPentium`, which can apply to many different processors of the same type.

1. Edit the name of the node, replacing the default name of `instance_n` with the name of a type of CPU (for example, `AMD Duron`).
2. Press **Enter**, or click outside the edit box, to terminate editing of the node name.

Modifying the Features of a Node

The Features dialog box for a node, as shown in the following figure, enables you to change the features of a node, including its type and the event to which the reception reacts.



A node has the following features:

- ◆ **Name** specifies the name of the node. The default name is `node_n`, where *n* is an incremental integer starting with 0.
- ◆ **Stereotype** specifies the stereotype of the element, if any. They are enclosed in guillemets, for example `«s1»` and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.

Component Instances

Component instances represent executable processes, objects, or libraries that run or reside on processing resources (nodes) during program execution. They are represented by the UML component symbol: a box with two small rectangles on the left side.


A component instance is an instance of a component type. Unlike components, there is no special naming convention for component instances. Drawing a component instance inside a node indicates that the component instance lives or runs on that node during run time.

Adding a Component Instance

You can add a component instance to a deployment diagram using the **Component** icon or the Rhapsody browser.

Using the Toolbar to Add a Component Instance

To use the **Drawing** toolbar to add a component instance, follow these steps:

1. With a node already drawn on your deployment diagram, click the **Component** icon  on the **Drawing** toolbar.
2. Draw the component instance inside the node. Rhapsody creates the component instance inside the selected node.
3. Edit the default name of the component instance, then press **Enter**.

Note that the component type is not part of the component instance's name. The full name of a component instance is the same as it appears in the browser.

4. Assign the component instance a type by opening its Features dialog box and setting an existing component in the **Component Type** box.

Note

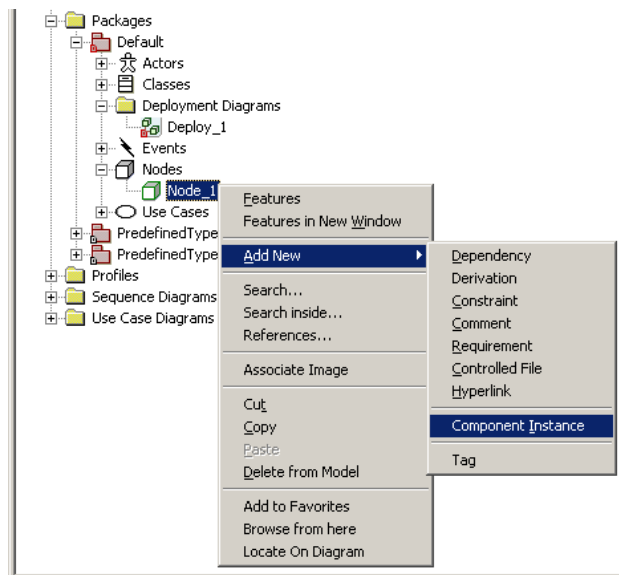
You cannot draw a component instance outside of a node in a deployment diagram.

Using the Browser to Add a Component Instance

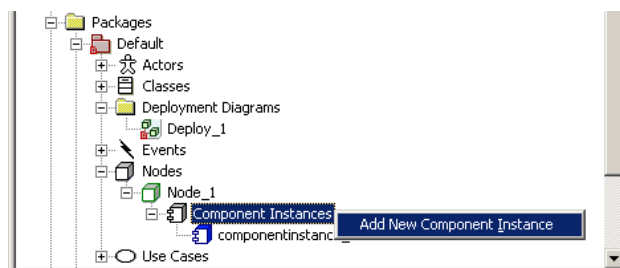
You can populate a deployment diagram with component instances by dragging them from the Rhapsody browser onto the diagram. Rhapsody creates a component instance based on the selected node.

1. Depending on the method you want to use:

- Right-click a node in the browser and select **Add New > Component Instance**, as shown in the following figure, or



- Right-click a component instance category, and select **Add New Component Instance**, as shown in the following figure.



2. Edit the default name of the component instance.
3. With both the browser and the deployment diagram editor in view, click-drag-and-drop the component instance from the browser onto the diagram.

Moving a Component Instance

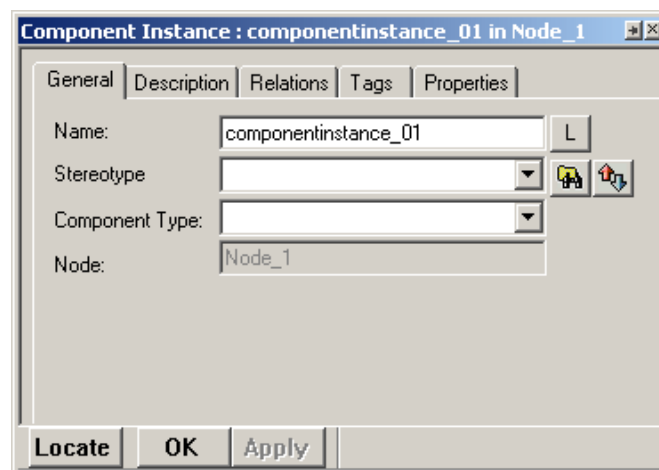
During the course of development, you may decide that you want a component instance to run on a different node.

To move a component instance from one node to another, follow these steps:

1. In the Rhapsody browser, select the component instance you want to move.
2. Use your mouse to drag the component instance to the new node.

Modifying the Features of a Component Instance

The Features dialog box, as shown in the following figure, enables you to change the features of a component instance, including its name and type.




- ◆ **Name** specifies the name of the component instance. The default name is `componentinstance_n`, where *n* is an incremental integer starting with 0.
- ◆ **Stereotype** specifies the stereotype of the element, if any. They are enclosed in guillemets, for example `«s1»` and enable you to tag classes for documentation purposes. See [Defining Stereotypes](#) for information on creating stereotypes.
- ◆ **Component Type** specifies the component type. This drop-down list includes all the components that exist in the model.
- ◆ **Node** specifies the name of the owning node. This box is read-only.

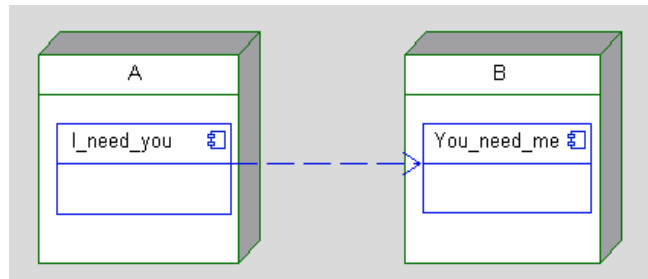
Dependencies

A dependency represents a requirement by one component instance of information or services provided by another. Dependencies can also be drawn between nodes. You can add a dependency using the **Dependency** tool or the Rhapsody browser.

Adding a Dependency Using the Toolbar

To add a dependency using the **Drawing** toolbar, follow these steps:

1. Click the **Dependency** icon  on the **Drawing** toolbar.
2. Click the dependent node or component instance.
3. Click the node or component instance that is being depended on. The arrow at the end of the dependency points to the selected instance, as shown in the following figure.



Adding a Dependency Using the Browser

To add a dependency using the Rhapsody browser, follow these steps:

1. Right-click the dependent node or component instance in the browser.
2. Select **Add New > Dependency** from the pop-up menu.
3. On the Add Dependency dialog box, select the node or component instance that is being depended on from the **Depends on** list. The dependency is added as a relation under the component instance. Click **OK**.
4. Click the **Dependency** icon.
5. Draw the dependency in the deployment diagram.

Note

You cannot add dependencies to a diagram by dragging.

Flows

Flows and FlowItems provide a mechanism for specifying exchange of information between system elements at a high level of abstraction. This functionality enables you to describe the flow of data and commands within a system at a very early stage, before committing to a specific design. As the system specification evolves, you can refine the abstraction to relate to the concrete implementation.

See [Flows and FlowItems](#) for detailed information about flows and FlowItems.

Assigning a Package to a Deployment Diagram

Like other diagrams, deployment diagrams belong to a package. When you create a deployment diagram in the browser at the project level, Rhapsody assigns the diagram to the default package and displays it in the Deployment Diagrams folder located at the project level. To create a deployment diagram in a particular package, first select the package, then create the new diagram. The nodes and component instances in the deployment diagram belong to the package of that diagram.

To assign a deployment diagram to a different package, follow these steps:

1. In the browser, right-click the deployment diagram and select **Features** to open the Features dialog box.
2. From the **Default Package** drop-down list, select the package to which you want to assign the deployment diagram.
3. Click **OK** to apply your changes and close the dialog box.

Diagrams located in the project-level **Deployment Diagrams** category remain in that category, but any nodes and component instances that the diagram contains are listed under the selected package.

Checks

Before generating code, Rhapsody automatically performs certain checks for the correctness and completeness of the model. You can also perform selected checks at any time during the design process. These predefined checks, also known as internal checks, are provided with the Rhapsody product. For a list of these predefined internal checks, see [List of Rhapsody Checks](#).

In addition, you can create checks that you code and customize to meet your needs. These user-defined checks are also known as external checks because they are not part of the set of predefined internal checks.

Both types of checks are displayed in the Rhapsody GUI, as described in this section.

For more specific information about external checks, see [User-defined Checks](#).

Checker Features

The checker works on either the active configuration or selected classes. It generates a list of the errors and warnings found in the model, with errors listed first. Errors prevent code generation from proceeding, while warnings draw your attention to unusual conditions in the model that do not prevent code generation. If there are no errors or warnings, the checker generates a message stating that all checks were completed successfully.

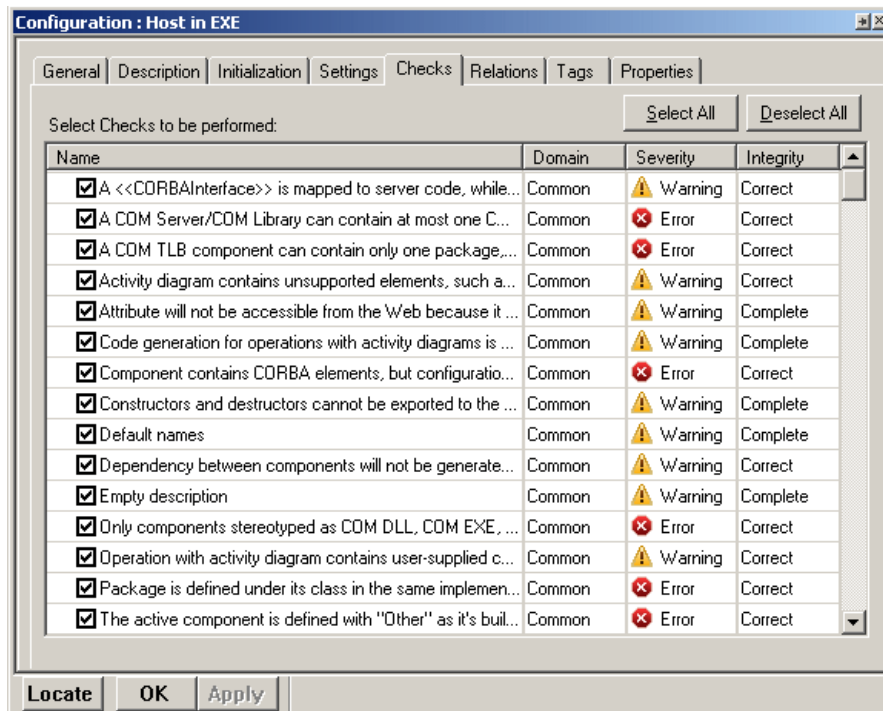
When you double-click a message, the checker opens the location in the model where the offending element or statement can be found, with the source of the error highlighted.

Note

The checker verifies the structural model by checking OMDs, and the behavioral model by checking statecharts. These are the main constructive diagrams in the model.

Using the Checks Tab




The **Checks** tab of the Features dialog box for a configuration, as shown in the following figure, lists all the available checks.



The **Checks** tab contains the following columns:

- ◆ **Name** describes the check to be performed. For example, **Attribute named the same as a state** checks whether an attribute and a state have the same name. By default, all possible checks are selected. To not include a check, clear the applicable check box. If all the checks are not selected and you want to do so, click the **Select All** button. To clear all of the checks (to make it easier to select only certain checks), click the **Deselect All** button.

Note the following that when a name is very long, you can move your mouse pointer over the name to see its full name in a tooltip.

- ◆ **Domain** specifies the area of the model that is searched. You can select checks that belong to one domain or another to limit the scope of the checks. The possible values are as follows:
 - **Class Model** searches the structural part of the model.
 - **Statechart** searches the behavioral part of the model.
 - **Common** searches both the structural and behavioral parts of the model. For example, **Default names** checks for default names in either classes or states.
 - There may be other domains that are from user-defined external checks.
- ◆ **Severity** specifies whether the condition being checked for constitutes an error , a warning , or is informational .

The following table lists the errors that cause code generation to abort.

Name conflicts	<ul style="list-style-type: none"> • An attribute is named the same as a state. • A class is named the same in a different subsystem. • An event and a generated state class have conflicting names. • An event is named the same as a class.
Other errors	<ul style="list-style-type: none"> • An OR state exists with no default state. • A fork to non-orthogonal states. • A join from non-orthogonal states. • A reference to an unresolved event. • A reference to an unresolved relational class. • A reference to an unresolved superclass. • A precondition for symmetric links failed.

- ◆ **Integrity** specifies whether the check has to do with the correctness or completeness of the model.

To sort by a column, click the column header.

Specifying Which Checks to Perform

You can control which checks are done. Note that Rhapsody automatically performs the predefined code generation checks when you do a check model.

To specify which checks to perform, follow these steps:

1. Open your model.
2. Set the configuration for the model whose code you want to check to be the active configuration. (See [Setting the Active Configuration](#).)
3. Open the Features dialog box for the active configuration and select the **Checks** tab. Do either of the following:
 - ♦ Choose **Tools > Check Model > Configure**.
The Features dialog box opens with the **Checks** tab selected.
 - ♦ From the main Rhapsody browser, double-click the active configuration and select the **Checks** tab.
4. Depending on what you want to do:
 - ♦ To select all the checks, click the **Select All** button.
 - ♦ To unselect all the checks so that you can more easily select the checks that you do want, click the **Deselect All** button and then select the checks you do want to perform.
 - ♦ Select and clear the check boxes next to the checks as you want.
 - ♦ Right-click one or more checks and select **Select**, **Deselect**, **Invert Selection**, as applicable.
5. Click **OK**.

Checking the Model

Before checking the model, be sure you have done the steps in [Specifying Which Checks to Perform](#).

To start checking the model, follow these steps:

1. If you want to perform checks only on selected packages or classes, select those elements on the Rhapsody browser.

Note: You can use **Shift+Click** to select contiguous elements and **Ctrl+Click** to select non-contiguous elements.

2. Select **Tools > Check Model** and select one of the following options, when available:
 - ◆ The active configuration
 - ◆ **Selected Elements** to perform the checks on the selected elements only
3. Review the results on the **Check Model** tab of the Output window. See [Check Model Tab](#).
4. For errors and warnings, you can double-click a message on the **Check Model** tab and Rhapsody will open to the relevant model element (for example, the Features dialog box for an association) or to the code on which you can make corrections or view the item more closely.

Checks Tab Limitations

There must be at least one check selected. Even if you clear all the check boxes and click **Apply** and **OK**, the next time you open the Features dialog box for the active configuration, you will see that all checks on the **Checks** tab will be selected.

User-defined Checks

You can create user-defined checks, which are also known as external checks, which are customized checks that you code yourself. System profiles, for example, often require domain-specific checks.

Just as the predefined internal checks provided by Rhapsody, you can define if external checks are called from code generation or not. In addition, you can define on which metaclasses external checks will be executed.

You can implement user-defined external checks through the Rhapsody API and use the GUI already in place in Rhapsody to run them (the **Checks** tab as described in [Using the Checks Tab](#)). Whether it is an internal check or an external check, the checks are performed and their results displayed through the same GUI in Rhapsody.

How to Create User-defined Checks

Rhapsody provides an API for registering, enumerating, and removing user-defined external checks through the use of the COM API for C++ and VB users, and the Java API for Java users. COM callbacks (connection points) allow you to invoke user-defined code when checks are executed. This capability is available for those users who use the COM API or Java API so that you can add, execute, or remove user-defined checks.

You decide which metaclasses (or new terms) you want to check, and your checks are called to check elements of whichever metaclasses you decided upon.

For more information about the COM API, refer to the *Rhapsody API Reference Manual*.

For example, for COM API users to create a user-defined check, follow these steps:

1. Implement a class defined from the interface `IRPEExternalCheck` in the COM API.
2. Register this class using the `IRPEExternalCheckRegister` Add method. You get this singleton via a method on `IRPApplication`.
3. You must implement `IRPEExternalCheck` on your client machine.

The following table lists the methods in the interface that you must implement for a user-defined check.

Method	Explanation
<code>GetName()</code>	Returns the name attribute as a string.
<code>GetDomain()</code>	Returns the domain attribute as a string.
<code>GetSeverity()</code>	Returns one of the predefined severity strings: Error, Warning, or Info.
<code>IsCompleteness()</code>	Returns TRUE if the check is for completeness, otherwise FALSE (the check is for correctness)
<code>ShouldCallFromCG()</code>	Returns TRUE if this check should be called when the user generates code
<code>GetRelevantMetaclasses()</code>	Returns a list of relevant metaclasses and/or new terms. The check will be invoked by Rhapsody for any element in the scope of the current configuration whose metaclass is returned.
<code>Execute()</code>	Called by Rhapsody in order to run the check. This routine returns TRUE if the check passes or FALSE if the check failed. It has two parameters: <ul style="list-style-type: none"> The first parameter provided by Rhapsody is the <code>IRPModelElements</code> that the check should be run on (its metaclass is in the checks <code>GetRelevantMetaclass()</code> list). The second parameter, returned by the check when relevant, is a collection of <code>IRPModelElements</code> that Rhapsody will highlight should the check fail.

How to Remove User-defined Checks

To remove a user-defined check, remove your class using the `IRPExternalCheckRegister` Remove method. You get this singleton via a method on `IRPApplication`.

How to Deploy User-defined Checks

This capability is available for those C++ and VB users who use the COM API, and Java users who use the Java API, so that you can add, execute, or remove user-defined checks. You provide the code in a COM client.

- ◆ If using VB, the client is an EXE file.
- ◆ If using C++, the client is an EXE or DLL file.
- ◆ If using Java, the client is a CLASS or JAR file.

Rhapsody uses the plug-in mechanism to load your code. The recommended way to do this is with a HEP file (recommended) or INI file. The HEP file is typically installed by you next to the relevant project or file. For example, a user wanting to write a Java plug-in would write the plug-in using the Rhapsody Java API and provide a HEP file similar to the one in the following example.

```
[Helpers]
numberOfElements=1
name1=ExternalChecks
JavaMainClass1=JavaPlugin.ExternalChecks
JavaClassPath1=$OMROOT\..\DoDAF
```

Sample check projects are provided for Java and VB in the **ExternalChecksSample** subfolder of your Rhapsody installation path (for example, <Rhapsody installation>\Samples\ExtensibilitySamples\ExternalChecksSample).

External Checks Limitations

RhapsodyCL is not supported because it does not support the COM API.

List of Rhapsody Checks

The following table lists all the predefined internal checks that can be performed by Rhapsody. All other checks that you may see on your system are user-defined external checks. For more information about them, see [User-defined Checks](#).

For ease-of-use, the table lists the checks by name in alphabetical order.

- ◆ The **Correct** column marks the checks for correctness, whereas checks for completeness are marked in the **Complete** column. In these columns, an **E** stands for Error, **I** stands for Informational, and **W** stands for Warning. For example, the **Default names** check has a **W** in the Complete column, which means it is a warning for completeness.
- ◆ The **Domain** column denotes the domain of the check and has these possible values:
 - **C** for Common error
 - **M** for error in the class Model
 - **S** for error in the Statechart

Check	Correct	Complete	Domain	Notes
A <<CORBAInterface>> is mapped to server code, while the configuration is not a CORBA server (property Configuration::CORBA::CORBAEnable is not set to CORBAServer)	W		C	In this case, the server mainline code, as specified by the ServerMainLineTemplate property (under Configuration::ORBname), is ignored.
A COM Interface can inherit only from a single COM Interface	E		M	
A COM Interface cannot have a 1-n relationship	E		M	
A COM Server/COM Library can contain at most one COM Library package	E		C	
A COM TLB component can contain only one package, which should be a COM Library.	E		C	
A Java class can inherit only from a single non-interface class	E		M	

Checks

Check	Correct	Complete	Domain	Notes
A Java interface can inherit only from other interfaces	E		M	
A package stereotyped as <<CORBAModule>> cannot contain functions or variables	E		M	
A circular composition relation was detected	W		M	
A singleton object cannot have a multiplicity other than one	W		M	
Activity diagram contains unsupported elements, such as events or triggered operations. The operation will not be generated!	W		C	
Attempt to create a global instance of an uninstantiable element	E		M	Instances of any kind can be created only from instantiable elements.
Attempt to create an initial instance of an uninstantiable element	E		M	Instances of any kind can be created only from instantiable elements.
Attribute modifiers are not supported in COM/ CORBA	W		M	
Attribute named the same as a state	E		M	
Attribute will not be accessible from the Web because it is missing both its accessor and mutator		W	C	
Attribute/Type references a template class as its type.	E		M	
Bad nesting	E			Similar to "Package defined under a class"
Behavioral port with empty contract owned by a non-reactive class/object. Port will not relay messages.		W	M	
CG::Package::EventsBaseID property value is out of legal event ID range	E		M	

Check	Correct	Complete	Domain	Notes
COM ATL class cannot be an active class	E		M	
COM ATL class cannot inherit from more than one COM Coclass	E		M	
COM Coclass can inherit only from COM Interface	E		M	
COM Interface can inherit only from COM Interface	E		M	
COM Library can contain only COM elements	E		M	
CORBAException has an operation	E		M	
CORBAException has an outgoing relation	E		M	
CORBAException involved in inheritance	E		M	
CORBAInterface inherits a non-CORBAInterface		E	M	
Cannot generate code for multiple inheritance from reactive classes	E		M	
Class does not realize all the interfaces provided by its behavioral ports.		W	M	
Class does not use all its reactive interface's receptions and triggered operations		W	M	
Class named the same as its package	E		M	A class and package cannot have the same name, because this would interfere with proper code generation.
Class with empty statechart		W	M	
Code generation for operations with activity diagrams is not supported for operations with variable-length argument lists. The operation will not be generated!		W	C	

Checks

Check	Correct	Complete	Domain	Notes
Component contains CORBA elements, but configuration is neither a CORBA client nor a CORBA server. (property <code>Configuration::CORBA::CORBAEnable</code>) is set to <code>No</code>	E		C	
Composite class without a statechart composite		W	M	
Composite with single component		W	M	
Constructors and destructors cannot be exported to the Web. Web Instrumentation code will not be generated for them		W	M	
Cross package link requires component based initialization scheme (<code>CG.ComponentInitializationScheme</code>)	W		M	
Dangling transition	E		S	
Default names		W	C	Some elements in the model use the default names assigned by Rhapsody.
Default transition not targeted to its state's substate	W		S	Every Or state with more than one substate must have a default transition. This error occurs when the default transition leads to something other than one of the Or state's substates.
Dependency on unresolved element	E		M	
Duplicated link between the same ends and over the same relation. Some of the links might be ignored by code generation	W		M	
Dynamic allocation should be allowed for a non-embeddable object	E		M	

Check	Correct	Complete	Domain	Notes
ESTL does not support multiple/virtual inheritance	W		M	
Element with no relations and no ports		W	M	
Empty body of primitive operations or global functions		W	M	The implementations of these operations/functions are not defined.
Empty description		W	C	
Event ID is not unique		W	M	Event IDs should be unique to avoid conflicts.
Event and generated state in a class have conflicting names	E		M	The implementation names of events and state cannot be the same.
Event is defined in the package but is not referenced in the interface of this package's class		W	M	You have defined an event in a package but there is no class that actually uses this event.
Event named the same as a class	E		M	
File name has to be in F8.3 format	E		M	If the <code>Filename</code> property (under <code>CG::Package/Class</code>) is defined as a file name longer than eight characters and the property <code><lang>_CG::Environment::IsFileNameShort</code> is set to <code>Checked</code> , the checker reports an error.
For dual interfaces, operations and attributes must have unique IDs (IDs cannot be blank). Rhapsody has generated unique IDs for one or more operations or attributes		W	M	
Fork to non-orthogonal states	E		S	
Global functions and variables are illegal in Java	E		M	
Ill-formed link across composite boundaries, code will not be generated	W		M	
Illegal for COM Coclax or COM Interface to have nested classes	E		M	

Checks

Check	Correct	Complete	Domain	Notes
Illegal for COM Coclass to have attributes	E		M	
Illegal for COM Coclass to have operations	E		M	
Illegal for COM Library to have nested packages	E		M	
Illegal inheritance from a template	E		M	
Illegal initialization of internal objects (Configuration dialog box, Initialization tab)	E		M	
Illegal outgoing relation for COM Coclass	E		M	
Illegal relation to a template	E		M	
Implement statechart property differs for derived and base classes	E		M	The <code>ImplementStatechart</code> property (under <code>CG::Class</code>) must be the same for base and derived classes.
Implementation not supported in the generated language.	E		M	
Inconsistent multiplicity in symmetric relation: instances won't be connected	W		M	
Invalid number of parameters in template instantiation	W		M	A template instantiation must have the same number of parameters as the template definition.
Isolated states		W	S	A state exists in a statechart that is not connected to any other state.
Join from non-orthogonal states	E		S	There is a join connector that is coming from a non-orthogonal state. The transition segments entering a join connector must originate from states residing in different orthogonal components.

Check	Correct	Complete	Domain	Notes
Link doesn't instantiate an association. Link is ignored.		W	M	
Link is based on unresolved relation.	E		M	
Link via ports with no matching interfaces. Link is ignored.		W	M	
Methods of dual and custom interfaces must return HRESULT		W	M	
Mismatch between implementation and multiplicity	E		M	The Implementation property setting is not appropriate for the multiplicity of the relation.
Misuse of embedded implementation in a relation		E	M	Embedded <Fixed/Scalar> properties are not correctly set for a relation.
Modeling of composite types (Enumeration/ Typedef) is not supported in COM/CORBA	W		M	
Multiple timeouts and duplicate triggers from the same state	E		M	Each state should only have a single timeout or trigger.
Name already in use by the component	E		M	
Non-behavioral port not connected to internal part.		W	M	
Non-behavioral port with explicit interfaces is not connected to an internal part. Messages might not be relayed.		W	M	
Non-interface classes are being specified as provided or required by the port. Please revise contract.		E	M	
Not enough values for initializer arguments	W		M	

Checks

Check	Correct	Complete	Domain	Notes
Number of events in the package exceed the event ID range (defined in the property <code>CG::Component::PackageEventIdRange</code>)	E		M	
Only a COM Library can contain COM elements	E		M	
Only components stereotyped as COM DLL, COM EXE, and COM TLB can contain a COM Library.	E		C	
Operation with activity diagram contains user-supplied code, which will be ignored.	W		C	
Or state with no default state		E	S	You have created an Or state without determining which is the default state. Use a default transition in the statechart to determine the default state (error of completeness).
Out of event IDs. There are more packages with events than possible event IDs. Modify the <code>CG::Component::PackageEventIdRange</code> property or reduce the number of packages with events	E		M	
Out of triggered operation IDs	E		M	
Outgoing interface must be a COM Interface	E		M	
Outgoing relation from a COM Interface must be stereotyped as connection point	E		M	
Outgoing relation from a CORBAInterface to a non-CORBAInterface	E		M	CORBAInterfaces can only have outgoing relations to other CORBAInterfaces.

Check	Correct	Complete	Domain	Notes
Package is defined under its class in the same implementation file	E		C	
Port connected to more than one end that provides the same interface.		W	M	
Port has an empty contract—no provided or required interfaces were specified.		W	M	
Port provides and requires same interface(s)—please revise contract details.	E		M	
Primitive, triggered operation or event is named the same as a state	E		M	You created a state with the same name as an event.
Qualifier for qualified relation not found	E		M	No qualifier was defined for a qualified relation.
Reactive interface with a reactive superclass; code cannot be generated	E		M	
Reactive interface with a statechart or an activity diagram; code cannot be generated	E		M	
Reactive interface without receptions or triggered operations		I	M	
Reactive template and Reusable statechart generation scheme	E		M	The Flat implementation of statecharts must be used with reactive template classes.
Reference to unresolved element in the scope of the active component	E		M	
Reference to unresolved event		E	M	A reference to an event exists in one view, but that event does not appear in at least one other view. This can occur when you are collaborating with other developers.
Reference to unresolved relational class		E	M	Same as Reference to unresolved event, except that the unresolved element is a relational class.

Checks

Check	Correct	Complete	Domain	Notes
Reference to unresolved statechart		E	M	Same as Reference to unresolved event, except that the unresolved element is a statechart.
Reference to unresolved stereotype	E		M	Same as Reference to unresolved event, except that the unresolved element is a stereotype
Reference to unresolved superclass		E	M	Same as Reference to unresolved event, except that the unresolved element is a superclass.
Reference to unresolved type		E	M	At least one element is defined to be of a type that is not defined in the model.
Relation should be implemented as static array when static architecture is used	W		M	The Implementation property of the relation should be set to StaticArray when using static architecture.
Relation to a CORBAException	E		M	Relations to CORBAExceptions are not allowed.
Relation without a multiplicity		E	M	
Relations from Java interfaces cannot be generated	E		M	
Singleton stereotype ignored: instance located in a different package		W	M	
Singleton stereotype ignored: instance multiplicity is not 1		W	M	
Singleton stereotype ignored: matching instance is not owned by a package		W	M	
Singleton stereotype ignored: multiple instances found		W	M	
Singleton stereotype ignored: no matching instance found		W	M	

Check	Correct	Complete	Domain	Notes
Singleton stereotype ignored: the <code>C_CG::Class::Object</code> type as singleton property is set to <code>Cleared</code>		W	M	
State named the same as its own class, superclass, or related class	E		M	
Static memory class with non-flat statechart	E		M	Flat implementation of statecharts must be used with static architectures.
Static memory class with override of operator <code>new</code> or <code>delete</code>	E		M	
Static memory element cannot be initialized	E		M	
Static reaction without action		W	S	You have defined a static reaction for a state but have not defined an action for it in the Features dialog box.
Static reaction without guard or trigger		E	S	
Stereotype exception is ignored when inheriting from a non-exception class	W		M	
Symmetric relation is not supported in Ada	E		M	
Symmetric relation with bi-directional inline in specification causes a dependency loop that is not supported by <language>; the inline is ignored during code generation	W		M	
Template instantiation of unresolved template	E		M	
The active component is defined with "Other" as its build type	E		C	
The body of an abstract method is ignored	W		M	

Checks

Check	Correct	Complete	Domain	Notes
The contract of the port is not an interface. Please replace contract or convert it to an interface.	E		M	
The events base ID set by the <code>CG::Package::EventsBaseID</code> property collides with the generated base events ID	E		M	
The name of the actor is not a legal code name	E		M	
Typedef with Constant modifier is based on a type with a Constant modifier.	E		M	
Unable to generate code for link—the multiplicities of the parts and ports being connected do not match.	W		M	
Unresolved type referenced by a template parameter.		E	M	
Unspecified AssociationRole—the AssociationRole is not connected to a formal link	W		M	
Unspecified ClassifierRole—the object is not connected to a formal classifier	W		M	
Unspecified message	W		M	
Web support is not available for a language variable of this type. Web Instrumentation code will not be generated for it.		W	C	
Web support is not available for global functions and global variables. Web Instrumentation code will not be generated for them.		W	C	

Check	Correct	Complete	Domain	Notes
Web support is not available for operations or events with more than one argument. Web Instrumentation code will not be generated for them.		W	C	
Web support is not available for templates and template instantiations. Web Instrumentation code will not be generated for them.		W	C	
When the property <code>CG::Configuration::GenerateDirectoryPerModelComponent</code> is set to Checked, <code>CG::Configuration::MakeFileGenerationScheme</code> should be set to Default.	E		C	
When the property <code>C_CG::Class::EnableDynamicAllocation</code> is set to Cleared, <code>C_CG::Configuration::InitializeEmbeddableObjectsByValue</code> should be set to Checked.	E		C	

Basic Code Generation Concepts

This section provides you with basic code generation concepts in Rhapsody. While this section focuses mostly on C++, information about other languages (C, Java, and Ada) may also appear. For more technical information about code generation, refer to the *Rhapsody Code Generation Guide*.

Rhapsody generates implementation code from your UML model. You can generate code either for an entire configuration or for selected classes. Inputs to the code generator are the model and the code generation (`<lang>_CG` and `CG`) properties. Outputs from the code generator are source files in the target language: specification files, implementation files, and makefiles.

Note that you can set up roundtripping and reverse engineering in Rhapsody in C and C++ so that they respect the structure of the code and preserve this structure when code is roundtripped/regenerated from the Rhapsody model. For details on how to activate the code respect ability, see [Reverse Engineering](#).

C code generation in Rhapsody is compliant with MISRA-C:1998. Note that there are justified violations, which are noted where appropriate.

Code Generation Overview

Between the code generator and the real-time Object Execution Framework (OXF), which is provided as a set of libraries, Rhapsody can implement most low-level design decisions for you. These decisions include how to implement design elements such as associations, multiplicities of related objects, threads, and state machines.

- ◆ **Elaborative.** You can use Rhapsody simply as a code browser, with all relations, state machine generation, and so on disabled. No instance initializations or links are generated—you do everything manually.
- ◆ **Translative.** You can draw a composite with components, links, and state machines, click a button, and get your application running—having to write only a minimum of code (you would have to write at least some actions in a statechart).

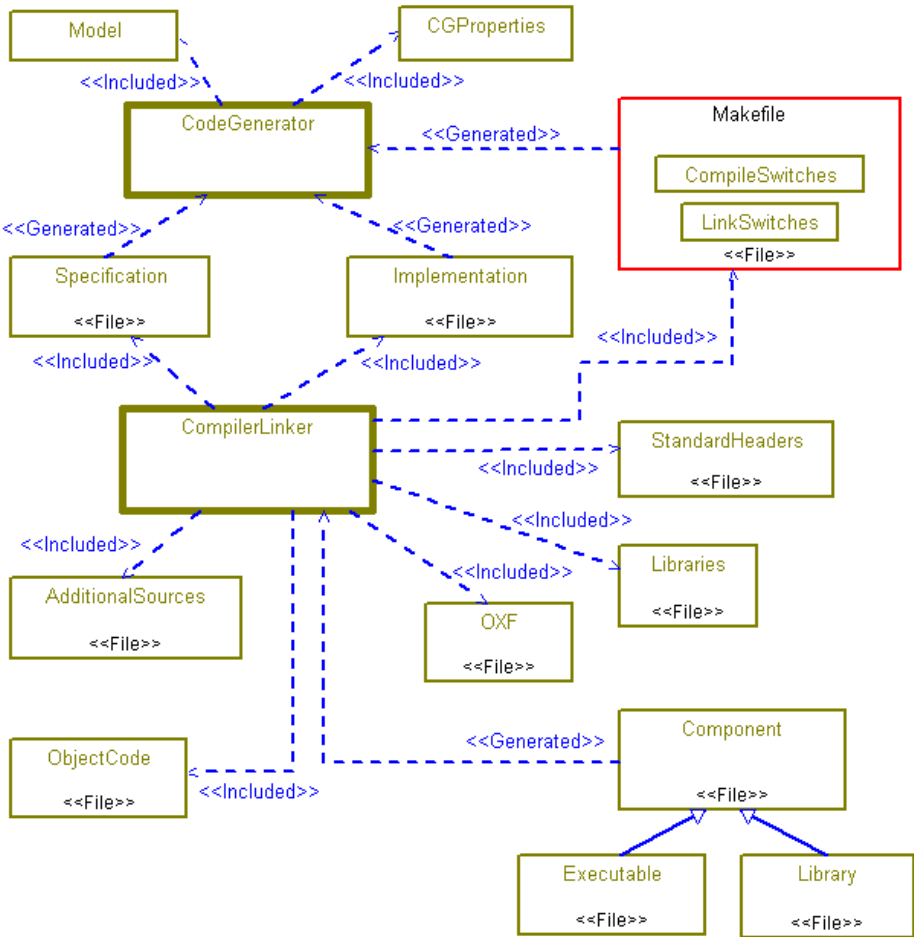
The Rhapsody code generator can be both elaborative and translative to varying degrees. Rhapsody does not force translation, but allows you to refine the code generation process to the desired level. Rhapsody can run in either mode, or anywhere between these two extremes.

Note

Microsoft is the default working environment for Rhapsody. You can specify other “out-of-the-box” environments in the environment settings for the configuration. See [Setting Component Configurations in the Browser](#) for more information.

The following figure shows the elements involved in generating code and building a component in Rhapsody. The dependency arrows indicate which files are generated and which files are included by the code generator and compiler, respectively. The thick borders around the code generator and compiler indicate that these are active classes.






Object Model of Generate, Make, Run



The Code Toolbar

When you are ready to build and execute your program, you can use the Code menu or the **Code** toolbar. If the **Code** toolbar is not displayed, choose **View > Toolbars > Code**.

The code generation process has the following operations and icons:

- ◆  **Make** builds the executable. You must have already generated the code for the model.
- ◆  **GMR** (Generate, Make, Run) generates the code, builds the executable, and runs the executable.
- ◆  **Stop Make/Execution** stops the build, or stops the program execution.
- ◆  **Run Executable** launches the executable portion of the model.
- ◆  **Disable dynamic model code associativity** disables automatic changes to the code whenever you make changes to the model. By default, dynamic model-code associativity is enabled.

Generating Code

Before you generate code, you must set the active configuration, as described in [Setting the Active Configuration](#).

The code generator automatically runs the checker to check for inconsistencies that may cause problems in generating or compiling the code. Some of the checks performed by the checker detect potentially fatal conditions that may cause code generation to abort if not corrected before generating code. See [Checks](#) for information about selecting checks to perform.

- ◆ Select **Code > Generate > <active configuration>**, or
- ◆ Press **Ctrl + F7**

Note that it is possible to generate code without intertask communication and event dispatching from Rhapsody, but this will disable the animation and visual debugging features. You can mitigate this effect by wrapping your in-house intertask communication and event dispatching routines inside an operation that is defined inside the model. In this case, the visualization is of the operation as a representative of your “real” intertask communication and event dispatching.

Incremental Code Generation

After initial code generation for a configuration, Rhapsody generates code only for elements that were modified since the last generation. This provides a significant performance improvement by reducing the time required for generating code.

In support of incremental code generation, Rhapsody creates a file in the configuration directory named `<configuration>.cg_info`. This file is internal to Rhapsody and does not need to be placed under configuration management.

Forcing Complete Code Generation

In some instances, you may want to generate the entire model, rather than just the modified elements.

To force a complete regeneration of the model, select **Code > Re Generate > <configuration>**.

Note

Forcing a regeneration of code using **Code > Re Generate > <configuration>** does not always regenerate the source files. The model is being regenerated (as is shown in the Output window) but only to temporary files. These files are then compared with their original counterparts and then overwritten if changes have been made. If you want to ensure that the code is completely regenerated each time, then you need to delete the active configuration folder. You can automate this with a macro (choose **Tools > Customize** to create your macro (helper application) and set the helper trigger to **Before Code Generation**).

Regenerating Configuration Files

When you delete classes or packages from the model, or add the first global instance to a package, you need to regenerate the configuration files (main and makefile).

- ◆ Select **Code > Re Generate > Configuration Files**, or
- ◆ Right-click the active configuration in the browser and select **Generate Configuration Main and Makefiles**

Smart Generation of Packages

Rhapsody generates code for a package only if it contains meaningful information, such as instances, types, and functions. This streamlines the code generation process by preventing generation of unnecessary package files.

To modify this behavior, use the `CG::Package::GeneratePackageCode` property. See the definition provided for the property on the applicable **Properties** tab of the Features dialog box.

To remove existing package files that do not contain meaningful information, select **Code > Clean Redundant Files**.

Generating Code Guidelines

When you generate code, consider the following:

- ◆ A reactive class that inherits from a non-reactive class may cause a compilation warning in Instrumentation mode. You can ignore this warning.
- ◆ If a class has a dependency on another class that is outside the scope of the component, Rhapsody does not automatically generate an `#include` statement for the external class. You must set the dependent class's `<lang>_CG::Class::SpecInclude` property.

Dynamic Model-Code Associativity

If you choose **Code > Dynamic Model Code Associativity** and then either the **Bidirectional** or **Code Generation** command, Rhapsody generates code automatically when an element changes and is in need of regeneration. These changes include the following:

- ◆ Changes to the element itself, such as its name or properties or by adding or removing parts of the element
- ◆ Changes in its owner, an associated element, a project, a selected component or configuration
- ◆ Adding an element to the model through the **Add to Model** feature
- ◆ Checking out the element with a CM tool
- ◆ Using the **Undo** command while editing the element

Code for an element is automatically generated if one of the following occurs:

- ◆ You open a Code View window for the element, either with the internal text editor or an external editor. See [Viewing and Editing the Generated Code](#).
- ◆ You set focus on an existing Code View window for that element.

If you set dynamic model-code associativity to `None` or to `Roundtrip`, you must use **Code > Generate** to generate code for the modified elements.

Note

Dynamic model-code associativity is applicable to all versions of Rhapsody (meaning, Rhapsody in C, C++, Java, and Ada).

Generating Makefiles

Rhapsody generates plain makefiles that include the list of files to be compiled and linked, their dependencies, and the compilation and link command.

Rhapsody generates makefiles when it generates the target file. The `<lang>_CG::<Environment>::InvokeMake` property defines how to execute the makefile. You can customize the content of the makefile by modifying the `<lang>_CG::<Environment>::MakeFileContent` property. See the definition provided for these properties on the applicable **Properties** tab of the Features dialog box.

Aborting Code Generation

To stop a long code generation session, select **Code > Stop X**. For example, if you are in the middle of building the code, the name of the option is **Stop Build**.

Another way to abort code generation is to create a file named `abort_code_gen` (no extension) in the root generation directory of the configuration. Rhapsody checks for the file while generating code; if the file is found, Rhapsody deletes the file and aborts code generation.

Note

If you invoke code generation from the Active Code View, you can abort only by using the `abort_code_gen` file—you *cannot* abort using the UI.

Targets

Once the code has been generated correctly, you can build the target component and delete old objects.

Building the Target

To build the target, use one of the following methods:

- ◆ Select **Code > Build <targetname>.exe**
- ◆ Click the **Build Target** button on the **Code** toolbar

As Rhapsody compiles the code, compiler messages are displayed in the Output window. When compilation is completed, the message `Build Done` is displayed.

Note

If you want to build applications for 64-bit targets, you must first rebuild the Rhapsody framework libraries. If you are running Rhapsody on a 64-bit system, then if you rebuild the libraries using the menu option **Code > Build Framework**, the Rhapsody libraries will be rebuilt such that you will be able to build applications for 64-bit targets. However, if you are running Rhapsody on a 32-bit system, you will have to rebuild the Rhapsody framework libraries manually.

Deleting Old Objects Before Building Applications

In some cases, it is possible for the linker to link an application using old objects. To clean out old objects, use one of the following methods:

- ◆ Use the **Code > Clean** command to delete all compiled object files for the current configuration. This is the recommended method.
- ◆ Use the **Code > Rebuild** command every time, which may be time-consuming for complex systems.
- ◆ Modify the start of make within `etc/msmake.bat` and remove the `/I` option. This will stop the build after the first source with errors.
- ◆ Within the **site.prp** file, change the `CPPCompileCommand` property from `String` to `MultiLine`, and change the content so that before a file is compiled, the existing `.obj` file is deleted.

For example:

```
CPPCompileCommand Property
MetaClass Microsoft:
Property CPPCompileCommand MultiLine
" if exist $OMFileObjPath erase $OMFileObjPath
    $(CPP) $OMFileCPPCompileSwitches
    /Fo\"$OMFileObjPath\" \"$OMFileImpPath\" "
MetaClass VxWorks:
Property CPPCompileCommand MultiLine
" @echo Compiling $OMFileImpPath
    @$(RM) $OMFileObjPath
    @$(CXX) $(C++FLAGS) $OMFileCPPCompileSwitches -o
    $OMFileObjPath $OMFileImpPath"
```

Running the Executable

Once you have built the target file, you can run it by using one of the following methods:

- ◆ Select **Code > Run <config>.exe**.
- ◆ Click the **Run** tool in the **Code** toolbar.

The default executable application is a console application. Once the application is started, regardless of its instrumentation level, a console window is displayed. If the application sends text messages to the standard output device, they are displayed in the console window.

Shortcut for Creating an Executable

- ◆ Select **Code > Generate/Make/Run**.
- ◆ Click the **Generate/Make/Run** tool in the **Code** toolbar.

Instrumentation

If you included animation instrumentation in your code, running the code displays the **Animation** toolbar. If you included trace instrumentation in your code, running the code displays the Trace window. You can choose animation or trace instrumentation when you set the parameters of your configuration. See [Configurations](#).

For more information about instrumentation, see [Animation](#) and [Tracing](#).

Stopping Model Execution

- ◆ Select **Code > Stop**.
- ◆ Click the **Stop Make/Execution** tool in the **Code** toolbar.

Generating Code for Individual Elements

You can generate code for a package and all its classes, or generate code for selected classes within a package. You can generate code from the Code menu, the browser, or an OMD.

Before you generate code for a class or package, you must set the active configuration, just as if you were generating the executable for the entire model. For more information, see [Setting the Active Configuration](#).

Using the Code Menu

1. Select a class from the browser, or select one or more classes in an OMD.
2. Select **Code > Generate > Selected Classes**.

If you have generated code for the entire model at least once, the **Generate** command generates code only for elements that have been modified since the last code generation.

1. Select a class from the browser, or select one or more classes in an OMD.
2. Select **Code > Re Generate > Selected Classes**.

Using the Browser

1. In the browser, right-click a package or class.
2. For a package, select **Generate Package** from the pop-up menu to generate code for all classes in the package.

For a class, select **Generate Class** from the pop-up menu to generate code only for that class.

Using an Object Model Diagram

To generate code for a class in an OMD, right-click the class and select **Generate Class** from the pop-up menu.

Results of Code Generation

When code generation completes, Rhapsody informs you of the results. Code generation messages are displayed in the Output window. Click a message to view the relevant location in the model.

Output Messages

When you generate code and build the target, messages are displayed in the Output window that either confirm the success of the operation or inform you of errors.

Some modeling constructs can cause code generation errors. Checking the model before generating the code enables you to discover and correct most of these errors before they can cause serious problems.

Locating and Fixing Compilation Errors

Rhapsody displays compilation errors that occur. To find the source of the compilation error in the code, double-click the relevant error message. Rhapsody tries to get you as close as possible to the source of the compilation error:

- ◆ If the source of the problem is in the implementation of an operation, the Features dialog box for the operation is opened with the **Implementation** tab displayed and the problematic line of code highlighted.
- ◆ If the source of the problem is in the initialization for a configuration, the Features dialog box for the configuration is opened with the **Initialization** tab displayed and the problematic line in the initialization code highlighted.
- ◆ If the source of the problem is in the actions defined for a state, the Features dialog box for the state is opened with the **General** tab displayed and the problematic line of code highlighted.
- ◆ If the source of the problem is in a reaction defined for a state, the Features dialog box for the relevant reaction is opened and the problematic line of code is highlighted.
- ◆ If the source of the problem is in the action code defined for a transition, the Features dialog box for the transition is opened with the **General** tab displayed and the problematic line of code is highlighted.
- ◆ For other compilation errors, Rhapsody opens up the relevant file in the code editor and highlights the problematic line of code. If you know where to correct this code within the model, you should do so. If not, you can correct the code manually and roundtrip the corrected code back into the model.

Note

If there is no apparent problem in the model and you have been generating code on selected classes, clean the whole configuration using the **Code/Clean** menu command. Compilation errors can occur if files were generated with different versions of the model.

Viewing and Editing the Generated Code

The Code View feature enables you to edit code for classes or a selected package (but not configurations). You can select one or more classes and bring up a text editor of your choice to edit the code files directly.

Setting the Scope of the Code View Editor

Before you can view or edit code, you must set the scope of the code view editor. follow these steps:

1. Right-click the component that includes the packages or classes whose code you want to view.
2. Select **Set as Active Component** from the pop-up menu.
3. In the **Current Configuration** field, select the configuration you want to use.

Adding Line Numbers

To display line numbers for the generated code, follow these steps:

1. Select **View > Active Code View** from the menu bar. Rhapsody displays the generated code in the results window.
2. To display line numbers, click the tab for the generated code and right-click in the code window.
3. Select **Properties** from the menu and click the **Misc** tab.
4. In the Line Numbering area, select `Decimal` from the **Style** drop-down menu and enter `1` in the **Start at** field as shown in the following figure.



5. Click **OK**.

Editing Code

There are two basic methods that can be used to edit code:

- ◆ Highlight the element, then select **Code > Edit > Selected classes**.
- ◆ Right-click the element, then select **Edit <element>**.

The specification and implementation files for the selected classes or package open in separate windows. You can use the internal text editor or an external editor to edit the files. See [Using an External Editor](#) for more information.

Depending on the dynamic model-code associativity (DMCA) setting, the changes you make to the code can be automatically roundtripped, or incorporated into the model. See [Automatic and Forced Roundtripping](#) for more information.

In addition, the DMCA setting determines whether open code views are updated if the code is modified by an external editor or the code generator. In the case of code generation, the code may have changed significantly. If this happens, the following message may be displayed:

```
filename: This file has been modified outside the source editor. Do you want  
to reload?
```

If you click **Yes**, the code view is refreshed with the new file contents and does not replace implementation files with obsolete information.

Locating Model Elements

When viewing code, Rhapsody provides a rapid method to locate in the browser the model element represented by the current position in the code.

To locate the model element, do one of the following:

- ◆ Right-click in the code window and then select **Locate in Model** from the context menu.
- ◆ Press **Ctrl+L**.

The element represented by the code is highlighted in the browser and the Features dialog box is opened with the relevant line of code highlighted in the dialog box.

If the model element is part of a statechart, the statechart is opened, with the relevant element highlighted in the diagram, and the Features dialog box is opened with the relevant line of code highlighted.

Note

This feature is available both in the normal code view and in the active code view.

Regenerating Code in the Editor

To re-generate code for files that are currently open in the editor, select **Code > Re Generate > Focused Views**.

Associating Files with an Editor

You can associate a specific editor with the extension of a source file. follow these steps:

1. Set the `General::Model::ClassCodeEditor` property to **Associate**.
2. Click **OK** to apply your changes and close the dialog box.

Note: Microsoft Visual C++ 6.0 includes an **Open with MSDEV** operation in the list of operations associated with *.cpp and *.h files. If you have MVC++ 6.0 installed, Rhapsody can erroneously associate the source files with the **Open with MSDEV** operation instead of with the **Open** operation.

To make sure that Rhapsody associates the MVC++ **Open** operation with the source files, follow these steps:

1. From Windows Explorer, select **View > Options > File Types**.
2. Select a file type (for example, **C Header File** associated with the file extension H), then select **Edit**.

The Edit File Type dialog box opens. Generally, the first action listed is **Open with MSDEV**.

3. Click **New**, add an “Open” action, and associate the **Open** action with your preferred editor application.
4. Remove the **Open with MSDEV** action.

Using an External Editor

Note that if you use an external editor to edit code for a Rhapsody model, you can set up Rhapsody in C++ and C so that it respects the structure of the code and preserves this structure when code is regenerated from the Rhapsody model. For details on how to activate the code respect ability, see [Code Respect](#).

To specify an external editor when you edit code, follow these steps:

1. Select **File > Project Properties**.
2. Select the **Properties** tab.
3. Navigate to the `General::Model::EditorCommandLine` property.
4. Click in the property value cell in the right column to activate the field, then click the ellipsis (...) to open the Browse for File dialog box.
5. Browse to the location of the editor you want to use (for example, Notepad) and select the editor. Click **OK** to close the dialog box. Rhapsody fills in the path in the property value field.
6. Click **OK** to close the dialog box.

Viewing Generated Operations

For each model, Rhapsody provides setter and getter operations for attributes and relations, initializer and cleanup operations, and one called `startBehavior()`. However, Rhapsody displays these automatically generated operations in the browser only if you set the `CG::CGGeneral::GeneratedCodeInBrowser` property to `Checked`. After you modify this property and regenerate the code, you will be able to view the automatically generated operations in the browser.

Deleting Redundant Code Files

A file can become redundant for the following reasons:

- ◆ An element mapped to a file is deleted or renamed.
- ◆ A change is made in the component scope.
- ◆ Changes are made in the mapping of elements to files.

Changes are made in component-level elements (components, configurations, folders, and files).

To delete redundant source files from the active configuration, select **Code > Clean Redundant Source Files**.

Generating Code for Actors

When you create a configuration, you can choose to generate code for actors (see [Modifying the Features of a Configuration](#)). If you set the active configuration to generate code for actors, Rhapsody generates, compiles, and links the actor code into the same library or executable as the rest of the system. This enables you to use actors to simulate inputs and responses to the system during system tests.

There are limits on the code generated for an actor, as described in [Limitations on Actors' Characteristics](#).

Selecting Actors Within a Component

Classes have code generation properties that determine whether Rhapsody should generate code for relations and dependencies between actors and classes.

To select code generation for actors within a component, follow these steps:

1. Right-click the component in the browser, then select **Features** from the pop-up menu.
2. Select the **Initialization** tab.
3. Under **Initial Instances**, select the actors to participate in the component.
4. Make sure **Generate Code for Actors** is selected.

Generate Code for Actors is selected by default, but is cleared when loading pre-version 3.0 models to maintain backward compatibility.

By default, relations between actors and classes are generated only if the actor is generated. You can fine-tune this within the class by using the `CG::Relation::GenerateRelationWithActors` property. See the definition provided for the property on the applicable **Properties** tab of the Features dialog box.

Limitations on Actors' Characteristics

There are limitations on how actors can be created, which affect code generation. These limitations are as follows:

- ◆ An actor cannot be specified as a composite in an OMD. Therefore, Rhapsody does not generate code that initializes relationships among its components.
- ◆ The base of an actor must be another actor. The base of a class must be another class.
- ◆ An actor can embed only nested actors. A class can embed only nested classes.
- ◆ The name of an actor does not have to conform to code standards for a class name. But if an actor name is illegal—that is, it does not have a name that can be compiled—an error is generated during code generation.

Generating Code for Component Diagrams

The following code generation semantics apply to component diagrams:

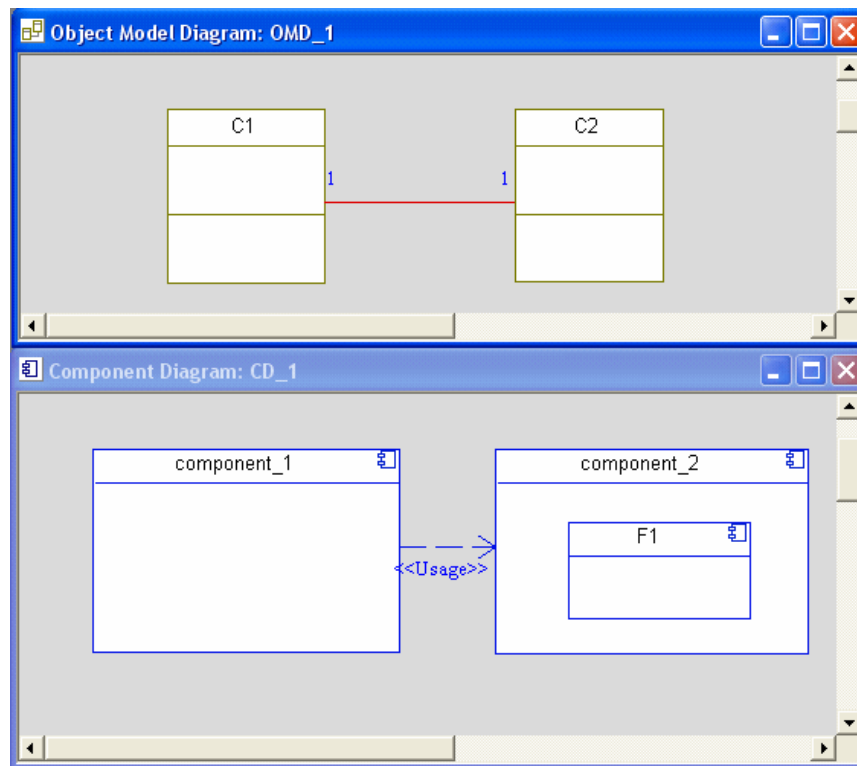
- ◆ Code generation is provided for library and executable build type components.
- ◆ Code generation is provided for folder and file component-related metatypes.
- ◆ Code is generated only with regard to relations between the binary components (library and executable).
- ◆ The following parts of a component diagram provide no code generation:
 - Relations involving files and folders
 - Interfaces realized by a component
 - All other component types that are not stereotyped «Library» or «Executable»
- ◆ A dependency between components generates code only if it has the «Usage» stereotype, with the following restrictions and limitations:
 - Related components must have a configuration with the same name.
 - If the two related components map the same element, code generation uses the first related component that was checked.
 - If an element is uniquely mapped (in the scope of a single binary component), the element file name is taken from that component.
 - If the element is not found in the scope of the generated component, related component, or uniquely mapped to some other component, the file name of the element is synthesized.

To disable the synthesis of the name, set the

`CG::Component::UseDefaultNameForUnmappedElements` property to `Cleared`.

The `CG::Component::ComponentsSearchPath` property specifies the name of related components, although the dependencies are checked before the property. For example, a dependency from component A to component B is equivalent to putting B in the `ComponentsSearchPath` property (with the obvious advantage on name change recovery).

Consider the diagrams shown in the following figure.



Classes `C1` and `C2` have a relation to each other (an association). The model has two components, `component_1` and `component_2`, that each have a configuration with the same name. `Component_1` has a dependency with stereotype `<<Usage>>` to `component_2`. Class `C1` is in the scope of `component_1`. Class `C2` is not in the scope of `component_1`, but is mapped to a file `F1` in `component_2`.

- ◆ Look for an element file name in related components (when the element is not in the scope of the current component).

For example, when generating `component_1`, when Rhapsody needs to include `C2`, it will include `F1` (the file in `component_2`).

- ◆ Add related components to the makefile include path. For example, in the `component_1` makefile, a new line is added to the include path with the location of `component_2`.
- ◆ If the current component build type is executable, and a related component build type is library, add the library to the build of the current component. For example, if the build type of `component_1` is executable, and the build type of `component_2` is library, the `component_1` makefile will include the library of `component_2` in its build.

Cross-Package Initialization

Rhapsody has properties that enable you to specify code that initializes package relations after package instances are initialized, but before these instances react to events. More generally, these properties enable you to specify any other initialization code for each package in the model. They enable you to initialize cross-package relations from any of the packages that participate in the relation or from a package that does not participate in the relation.

The properties that govern package initialization code are as follows:

- ◆ `CG::Package::AdditionalInitialization` specifies additional initialization code to run after the execution of the `package initRelations()` method.
- ◆ `CG::Component::InitializationScheme` specifies at which level initialization occurs. The possible values are as follows:
 - `ByPackage`—Each package is responsible for its own initialization; the component needs only to declare an attribute for each package class. This is the default option and maintains backward compatibility with pre-V3.0 Rhapsody models.
 - `ByComponent`—The component must initialize all global relations declared in all of its packages. This must be done by explicit calls in the component class constructor for each package's `initRelations()`, additional initialization, and `startBehavior()`.

The following is an example of C++ code generated from the model in the diagram above when the `InitializationScheme` property is set to `ByPackage`.

The component code is as follows:

```
class DefaultComponent {
private :
    P1_OMInitializer initializer_P1;
    P2_OMInitializer initializer_P2;
};
```

The P1 package code is as follows:

```
P1_OMInitializer::P1_OMInitializer() {
    P1_initRelations();
    < P1 AdditionalInitializationCode value>
    P1_startBehavior();
}
```

The following is an example of C++ component code generated when the `InitializationScheme` property is set to `ByComponent`:

```
DefaultComponent::DefaultComponent() {  
    P1_initRelations();  
    P2_initRelations();  
        < P1 AdditionalInitializationCode value>  
        < P2 AdditionalInitializationCode value>  
    P1_startBehavior();  
    P2_startBehavior();  
}
```

Class Code Structure

This section describes the structure of the generated code, including information on how the model maps to code and how you can control model mapping. In addition to the model elements, the generated source code includes annotations and, if instrumented, instrumentation macros.

Note

Annotations map code constructs to design constructs. They play an important role in tracing between the two. Do not touch or remove annotations. If you do, you hinder tracing between the model and the code. Annotations are comment lines starting with two slashes and a pound sign (`// #` for C and C++) or two dashes and a plus sign (`--+` for Ada).

Instrumentation macros become hooks and utility functions to serve the animation/trace framework. They are implemented as macros to minimize the impact on the source code. If you remove instrumentation macros, animation cannot work correctly.

The code structures shown are generic, using generic names such as *class* and *state*. Your code will contain the actual names.

Class Header File

The class header file contains the following sections:

- ◆ Prolog
- ◆ Forward declarations
- ◆ Instrumentation
- ◆ Virtual and private inheritance
- ◆ User-defined attributes and operations
- ◆ Variable-length argument lists
- ◆ Relation information
- ◆ Statechart implementation
- ◆ Events interface
- ◆ Serialization instrumentation
- ◆ State classes

Prolog

The prolog section includes the framework file, and a header or footer file (if applicable).

You can add additional include files using the `<lang>_CG::Class/Package::SpecIncludes` property. You may have to do this if you create dependencies to modules that are not part of the Rhapsody design.

For example:

```
#ifndef class_H
#define class_H
#include <oxf.h> // The framework header file
//-----
// class.h
//-----
```

Relationships to Other Classes

This section of the header file includes forward declarations for all the related classes. These are based on the relationships of the class with other classes specified in the model.

For example:

```
class forwarded-class1;
class forwarded-class2;

class class : public OMReactive {
// Class definition, and inheritance
// from framework class (if needed!).
```

Instrumentation

If you compiled with instrumentation, instrumentation macros provide information to animation about the run-time state of the system.

For example:

```
DECLARE_META // instrumentation for the class
```

User-Defined Attributes and Operations

In this section, all operations and attribute data members are defined. The code in this section is a direct translation of class operations and attributes. You control it by adding or removing operations and attributes from the model.

For example:

```
//// User explicit entries ////
public :
  /// operation op1() // Annotation for the operation
  void op1(); // Definition of an operation
protected :
  // Attributes:
```

```

    //## attribute attr1 // Annotation for an attribute
    attrType1 attr1; // Definition of an attribute
    /// User implicit entries ///
public :
    // Constructors and destructors:
    class(OMThread* thread = theMainThread);
    virtual ~class(); // Generated destructor

```

Generating Code for Static Attributes

When you generate code for a static attribute, the initial value entered is generated into a statement that initializes the static attribute outside the class constructor. Consider the following initial values for three static attributes:

Attribute	Type	Value
attr1	int	5
attr2	OMBoolean	true
attr3	OMString	"Shalom"

When you generate code, these values cause the following statements to be generated in the specification file A.h:

```

//-----
// A.h
//-----
class A {
    /// User explicit entries ///
protected:
    //## attribute attr3
    static OMString attr3;

    //## attribute attr1
    static int attr1;

    //## attribute attr2
    static OMBoolean attr2;
    ...
};

```

In the implementation file, A.cpp, the following initialization code is generated:

```

#include "A.h"

//-----
// A.cpp
//-----

// Static class member attribute
OMString A::attr3 = "Shalom";

// Static class member attribute
int A::attr1 = 5;

// Static class member attribute
OMBoolean A::attr2 = true;
A::A() {
};

```

Variable-Length Argument Lists

To add a variable-length argument list (...) as the last argument of an operation, open the Properties dialog box for the operation, and set the

CG::Operation::VariableLengthArgumentList property to Checked.

For example, if the VariableLengthArgumentList property is set to Checked for an operation void foo(int i), the following declaration generated for foo():

```
void foo(int i, ...);
```

Synthesized Methods and Data Members for Relations

This section of the header file includes every relation of the class defined in the model. If appropriate, it includes a data member, as well as a set of accessor and mutator functions to manipulate the relation.

You can control this section by changing the relation properties and container properties. Refer to the *Rhapsody Properties Reference Manual* for more information about the CG properties.

For example:

```
OMIterator<rclass1*> getrelation1() const;
void addrelation1(rclass* p);
void removerelation1(rclass* p);
void clearrelation1();
protected :
//
OMCollection<rclass*> relation1;
attrType1 getattr1() const;
void setattrType1(attrType1 p);
```

Statechart Implementation

- ◆ Change the statechart implementation strategy from reusable to flat.
- ◆ Disable code generation for statecharts.

For example:

```
//// Framework entries ////
public :
    State* state1;           // State variables
    State* state2;
    void rootStateEntDef(); // The initial transition
    // method
    int state1Takeevent1(); // event takers
    int state2Takeevent2();

private :
    void initRelations(); //InitRelations or
    //InitStatechart is
    //generated to initialize
    //framework members
    void cleanUpRelations();//CleanupRelations or
    //CleanupStatechart is
```



```
//generated to clean up
//framework members in
//destruction.
```

Note

The `initRelations()` and `cleanUpRelations()` operations are generated only if the `CG::Class::InitCleanUpRelations` property is set to `Checked`. See the definition provided for the property on the applicable **Properties** tab of the Features dialog box.

Events Interface

The section of the code illustrates events consumed by the class. It is for documentation purposes only because all events are handled through a common interface found in `OMReactive`. This section is useful if you want to implement the event processing yourself.

For example:

```
////      Events consumed      ////
public :
    // Events:
    // event1
    // event2
```

Note that code generation supports array types in events. Is it possible to create the following `GEN()`:

```
Client->GEN(E("Hello world!"));
```

In this call, `E` is defined as follows:

```
class E : public OMEvent {
    Char message[13];
};
```

Serialization Instrumentation

If you included instrumentation, the following instrumentation macro is included in the header file:

```
DECLARE_META_EVENT
};
```

It expands to method definitions that implement serialization services for animation.

State Classes

The state defines state classes to construct the statechart of the class. State classes are generated in the reusable state implementation strategy.

For example:

```
class state1 : public ComponentState {
public :
    // State class implementation
```

```
};  
class state2 : public Orstate {  
public :  
// State class implementation  
};  
#endif
```

You can eliminate the state classes by choosing the flat implementation strategy, where states are manifested as enumeration types.

Implementation Files

The class implementation file contains implementation of methods defined in the specification file. In addition, it contains instrumentation macros and annotations. As noted previously, eliminating or modifying either instrumentation macros and annotations can hurt the tracing and functionality between your model and code.

Headers and Footers

You can define your own headers and footers for generated files. Refer to the property definitions displayed in the **Properties** tab of the Features dialog box or examine the complete list of property definitions in the *Rhapsody Property Definitions* PDF file available from **Help > List of Books**. That list can be searched along with the other PDF versions of the Rhapsody documentation to locate specific property definitions and the procedures that use them.

Prolog

The prolog section of the implementation file includes header files for all the related classes. You can add additional `#include` files using the `C_` and `CPP::Class/Package::ImpIncludes` property. See the definition provided for the property on the applicable **Properties** tab of the Features dialog box.

You can wrap sections of code with `#ifdef` `#endif` directives, add compiler-specific keywords, and add `#pragma` directives using prolog and epilog properties. For more information, see [Wrapping Code with #ifdef-#endif](#).

The following code shows a sample prolog section:

```
/// package package
/// class class
#include "class.h"
#include <package.h>
#include <rclass1.h>
#include <rclass2.h>
//-----
// class.cpp
//-----
DECLARE_META_PACKAGE(Default)
#define op1_SERIALIZE
```

Constructors and Destructors

A default constructor and destructor are generated for each class. You can explicitly specify additional constructors, or override the default constructor or destructor by explicitly adding them to the model.

For example:

```
class::class(OMThread* thread) {
    NOTIFY_CONSTRUCTOR(class, class(), 0,
        class_SERIALIZE);
    setThread(thread);
    initRelations();
};
```

If you are defining states, use `initStatechart` instead of `initRelations`:

```
class::~class() {
    NOTIFY_DESTRUCTOR(~class);
    cleanUpRelations();
};
```

Similarly, if you are defining states, use `cleanUpStatechart` instead of `cleanUpRelations`.

Operations

The following code pattern is created for every primitive (user-defined) operation:

```
void class::op1() {
    NOTIFY_OPERATION(op1, op1(), 0, op1_SERIALIZE);
    // Instrumentation
    //#[ operation op1() // Annotation
    // body of the operation as you entered
    //#]
};
```

Accessors and Mutators for Attributes and Relations

Accessors and mutators are automatically generated for each attribute and relation of the class. Their contents and generation can be controlled by setting relation and attribute properties.

For example:

```
attrlType class::getattr1() const {
    return attr1;
};

void class::setattr1(attrltype p) {
    attr1 = p;
};

OMIteratorrrclass* class::getItsRclass() const {
    OMIteratorrrclass* iter(itsRclass);
    return iter;
};

void Referee::_addItsRclass(Class* p_Class) {
    NOTIFY_RELATION_ITEM_ADDED("itsRClass", p_Class,
```

```
        FALSE, FALSE);
    itsRclass->add(p_Class);
};

void class::removeItsrclass(rclass* p) {
    NOTIFY_RELATION_ITEM_REMOVED();
    rclass.remove(p);
};

void class::clearItsPing() {
};
```

Instrumentation

This section includes instrumentation macros and serialization routines used by animation.

For example:

```
void class::serializeAttributes() const {
    // Serializing the attributes
};

void class::serializeRelations() const {
    // Serializing the relation
};

IMPLEMENT_META(class, FALSE)
IMPLEMENT_GET_CONCEPT(state)
```

State Event Takers

These methods implement state, event, and transition behavior. They are synthesized based on the statechart. If you set the CG::Attribute/Event/File/Generalization/Operation/Relation::Generate property of the class to Cleared, they are not generated.

For example:

```
int class::state1Takeevent1() {
    int res = eventNotConsumed;
    SETPARAMS(hungry);
    NOTIFY_TRANSITION_STARTED("2");
    Transition code
    NOTIFY_TRANSITION_TERMINATED("2");
    res = eventConsumed;
    return res;
};
```

Initialization and Cleanup

These methods implement framework-related initialization and cleanups.

For example:

```
void class::initRelations() {
    state1 = new state1Class(); // creating the states
};

void class::cleanUpRelations() {
};
```

Implementation of State Classes

This section implements a dispatch method and a constructor for each state object. You can change this by choosing the flat implementation strategy, which does not generate state classes.

For example:

```
class state1::class_state1(class* c, State* p,
    State* cmp): LeafState(p, cmp) {
    // State constructor
};

int class_state1::takeEvent(short id) {
    int res = eventNotConsumed;
    switch(id) {
        case event1_id: {
            res = concept->state1Takeevent1();
            // Dispatching the transition method
            break;
        };
    };
};
```

Changing the Order of Operations/Functions in Generated Code

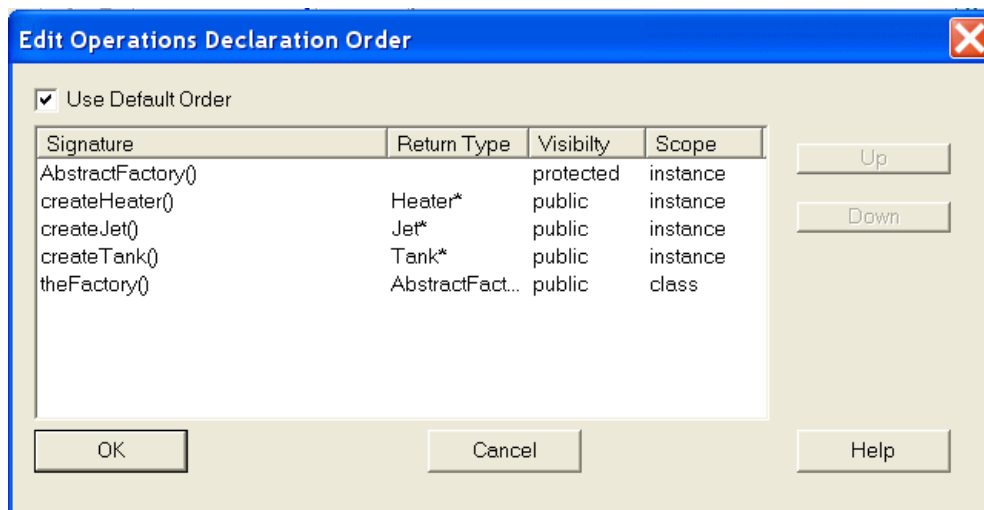
By default, operations appear in the following order in generated code:

1. Constructors and destructors
2. User-defined operations
3. Triggered operations

Within each of these categories, the operations are displayed in the following order: public, protected, private. Within these access subcategories, operations are listed alphabetically.

In some cases, you may want to define a specific order for the operations when the code is generated. To modify the order of appearance in the generated code, follow these steps:

1. Highlight **Operations** or **Functions** in the browser.
2. Right-click and select **Edit Operation Order** (or **Edit Function Order**).
3. In the dialog box that is displayed, highlight the **Signature** that you want to move and use the **Up** and **Down** buttons to modify the order that will be used in code generation.



4. Click **OK**.

Note

If the **Up** and **Down** buttons are disabled (as shown in this example), clear the **Use Default Order** option box at the top of the dialog box.

If you would like to restore the default order used by Rhapsody for code generation, make these changes:

1. Check the **Use Default Order** option box.
2. Click **OK**.

Using Code-Based Documentation Systems

Rhapsody enables you to standardize the way element comments are generated. Using a template, the code generator can take the element description from the model, along with its tag values, and format it as the generated comment inside the code. These generated comments can then be processed by code-based documentation tools such as Doxygen™.

Template Properties

The following table lists the properties (under `<lang>_CG`) that enable you to standardize the way comments are generated in the code.

Metaclass	Property	Description
File	ImplementationFooter	Specifies the multiline footer to be generated at the end of implementation files.
	ImplementationHeader	Specifies the multiline header that is generated at the beginning of implementation files.
	SpecificationFooter	Specifies the multiline footer to be generated at the end of specification files.
	SpecificationHeader	Specifies the multiline header to be generated at the beginning of specification files.
Configuration	DescriptionBeginLine	Enables you to specify the prefix for the beginning of comment lines in the generated code. This functionality enables you to use a documentation system (such as Doxygen), which looks for a certain prefix to produce the documentation. Note: This property affects only the code generated for descriptions of model elements; other auto-generated comments are not affected.
	DescriptionEndLine	Enables you to specify the prefix for the end of comment lines in the generated code.

Metaclass	Property	Description
Argument Attribute Class Event Operation Package Relation Type	DescriptionTemplate	Specifies how to generate the element description in the code. An empty MultiLine (the default value) tells Rhapsody to use the default description generation rules.

Sample Usage

This section documents a simple model configured to generate Doxygen-compatible descriptions, including the appropriate property settings.

The Model Profile

The DoxygenDescription profile defines the description tags and sets the default property values. The following table lists the property values for the DoxygenDescription profile.

Property	Value
C_ and CPP_CG::Configuration	
	"*"
DescriptionEndLine	" "
C_ and CPP_CG::File	
SpecificationHeader	<pre> /** * (C) copyright 2004 * * @file \$FullCodeGeneratedFileName * @author \$FileAuthor * @date \$CodeGeneratedDate * @brief \$FileBrief * * Rhapsody: \$RhapsodyVersion * Component: \$ComponentName * Configuration: \$ConfigurationName * Element: \$ModelElementName * */ </pre>

Property	Value
C_ and CPP_CG::Class	
DescriptionTemplate	"/** * Class \$(Name) * @brief \$Description * @see \$See * @warning \$Warning */"
C_ and CPP_CG::Operation	
DescriptionTemplate	"/** * Operation \$Signature * @brief \$Description * \$Arguments * @return \$Return * @see \$See * @warning \$Warning */"
C_ and CPP_CG::Argument	
DescriptionTemplate	"@param \$(Name): [\$Direction] \$Description"

Setting Up the Point Class

The Point class may contain these elements:

- ◆ Point—A 2D point representation
- ◆ Point.create()—A static method to create a new Point
- ◆ Point.create().x—The x coordinate for the new Point
- ◆ Point.create().y—The y coordinate for the new Point

The following table shows the tag values set by the model elements.

Element	Tag Name	Tag Value
Point	FileAuthor	Ford Perfect
Point	FileBrief	The Point class definition
Point	See	Point3D
Point	Warning	NA
Point.create()	Return	a new Point with the specified coordinates
Point.create()	See	NA
Point.create()	Warning	NA

The Generated Code

The file header for the `Point` specification file is as follows:

```
/**
 * (C) copyright 2004
 *
 * @file      DefaultComponent\DefaultConfig\Point.h
 * @author    Ford Perfect
 * @date      //! Tue, 16, Mar 2004
 * @brief     The Point class definition
 *
 * Rhapsody:      5.0.1
 * Component:     DefaultComponent
 * Configuration: DefaultConfig
 * Element:       Point
 **/
```

The generated description of the `Point` class is as follows:

```
/**
 * Class Point
 * @brief A 2D point representation
 * @see Point3D
 * @warning NA
 */
///class Point
class Point {...
```

The generated description of the `create()` operation is as follows:

```
/**
 * Operation create(int,int)
 * @brief A static method to create a new Point
 * $Arguments
 * @param x: [in] The new Point x coordinate
 * @param y: [in] The new Point y coordinate
 * @return a new Point with the specified coordinates
 * @see NA
 * @warning NA
 */
///operation create(int,int)
static Point* create(int x, int y);
```

Roundtripping Behavior

Advanced/Full roundtrip does not update element descriptions when the relevant `DescriptionTemplate` properties are not empty.

The following table lists the properties for which roundtripping does *not* update the element description.

Element	Element Descriptions are <i>Not</i> Updated when these Properties are Not Empty
Argument	The argument description is not updated when the owner (operation or event) description is not updated.
Association end	<code><lang>_CG::Relation::DescriptionTemplate</code>
Attribute	<code><lang>_CG::Attribute::DescriptionTemplate</code>
Class	<code><lang>_CG::Class::DescriptionTemplate</code>
Event	<code><lang>_CG::Argument::DescriptionTemplate</code> <code><lang>_CG::Event::DescriptionTemplate</code> (both at the operation or argument level)
Operation	<code><lang>_CG::Argument::DescriptionTemplate</code> <code><lang>_CG::Operation::DescriptionTemplate</code> (both at the operation or argument level)
Package	<code><lang>_CG::Package::DescriptionTemplate</code>
Type	<code><lang>_CG::Type::DescriptionTemplate</code>

Wrapping Code with #ifdef-#endif

If you need to wrap an operation with an `#ifdef #endif` pair, add a compiler-specific keyword, or add a `#pragma` directive, you can set the `SpecificationProlog`, `SpecificationEpilog`, `ImplementationProlog`, and `ImplementationEpilog` properties for the operation.

For example, to specify that an operation is available only when the code is compiled with `_DEBUG`, follow these steps:

- ◆ Set `SpecificationProlog` to `#ifdef _DEBUG <CR>`.
- ◆ Set `SpecificationEpilog` to `#endif`.
- ◆ Set `ImplementationProlog` to `#ifdef _DEBUG <CR>`.
- ◆ Set `ImplementationEpilog` to `#endif`.

The same properties are available for configurations, packages, classes, and attributes. For detailed definitions of these properties, refer to the property definitions displayed in the Features dialog box.

Overloading Operators

You can overload operators for classes created in Rhapsody. For example, for a `Stack` class, you can overload the “+” operator to automatically perform a `push()` operation, and the “-” operator to automatically perform a `pop()` operation.

All of the overloaded operators (such as `operator+` and `operator-`) can be modeled as member functions, except for the stream output `operator<<`, which is a global function rather than a member function and must be declared a friend function. The overloaded operators that are class members are all defined as primitive operations.

To illustrate operator overloading, consider two classes, `Complex` and `MainClass`, defined as follows:

Class Complex

Attributes:

```
double imag;  
double real;
```

Operations:

```
Complex() // Simple constructor  
Body:  
{  
    real = imag = 0.0;  
}
```

```

Complex(const Complex& c) //Copy constructor
Arguments: const Complex& c
Body:
{
    real = c.real;
    imag = c.imag;
}
Complex(double r, double i) // Convert constructor
Arguments: double r
           double i = 0.0
Body:
{
    real = r;
    imag = i;
}

operator-(Complex c) // Subtraction
Return type: Complex
Arguments: Complex c
Body:
{
    return Complex(real - c.real, imag - c.imag);
}
operator[](int index) // Array subscript
Return type: Complex&
Arguments: int index // dummy operator - only
           // for instrumentation
           // check
Body:
{
    return *this;
}
operator+(Complex& c) // Addition by value
Return type: Complex

Arguments: Complex& c
Body:
{
    return Complex(real + c.real, imag + c.imag);
}
operator+(Complex* c) // Addition by reference
Return type: Complex*
Arguments: Complex *c
Body:
{
    cGlobal = new Complex (real + c->real,
                           imag + c->imag);
    return cGlobal;
}
operator++() // Prefix increment
Return type: Complex&
Body:
{
    real += 1.0;
    imag += 1.0;
    return *this;
}
operator=(Complex& c) // Assignment by value
Return type: Complex&

```

```
Arguments: Complex& c
Body:
{
    real = c.real,
    imag = c.imag;
    return *this;
}

operator=(Complex* c) // Assignment by reference
Return type: Complex*
Arguments: Complex *c
Body:
{
    real = c->real;
    imag = c->imag;
    return this;
}
```

The following are some examples of code generated for these overloaded operators.

This is the code generated for the overloaded prefix increment operator:

```
Complex& Complex::operator++() {
    NOTIFY_OPERATION(operator++, operator++(), 0,
        operator_SERIALIZE);
    //#[ operation operator++()
    real += 1.0;
    imag += 1.0;
    return *this;
    //#]
};
```

This is the code generated for the overloaded + operator:

```
Complex Complex::operator+(Complex& c) {
    NOTIFY_OPERATION(operator+, operator+(Complex&), 1,
        OM_operator_1_SERIALIZE);
    //#[ operation operator+(Complex&
    return Complex(real + c.real, imag + c.imag);
    //#]
};
```

This is the code generated for the first overloaded assignment operator:

```
Complex& Complex::operator=(Complex& c) {
    NOTIFY_OPERATION(operator=, operator=(Complex&), 1,
        OM_operator_2_SERIALIZE);
    //#[ operation operator=(Complex&)
    real = c.real;
    imag = c.imag;
    return *this;
    //#]
};
```

The browser lists the `MainClass`, which is a composite that instantiates three `Complex` classes.

Its attributes are as follows:


```
Complex* c1
Complex* c2
Complex* c3

Body~MainClass()           //Destructor
Body
{
    delete c1;
    delete c2;
    delete c3;
}
e()                         // Event
```

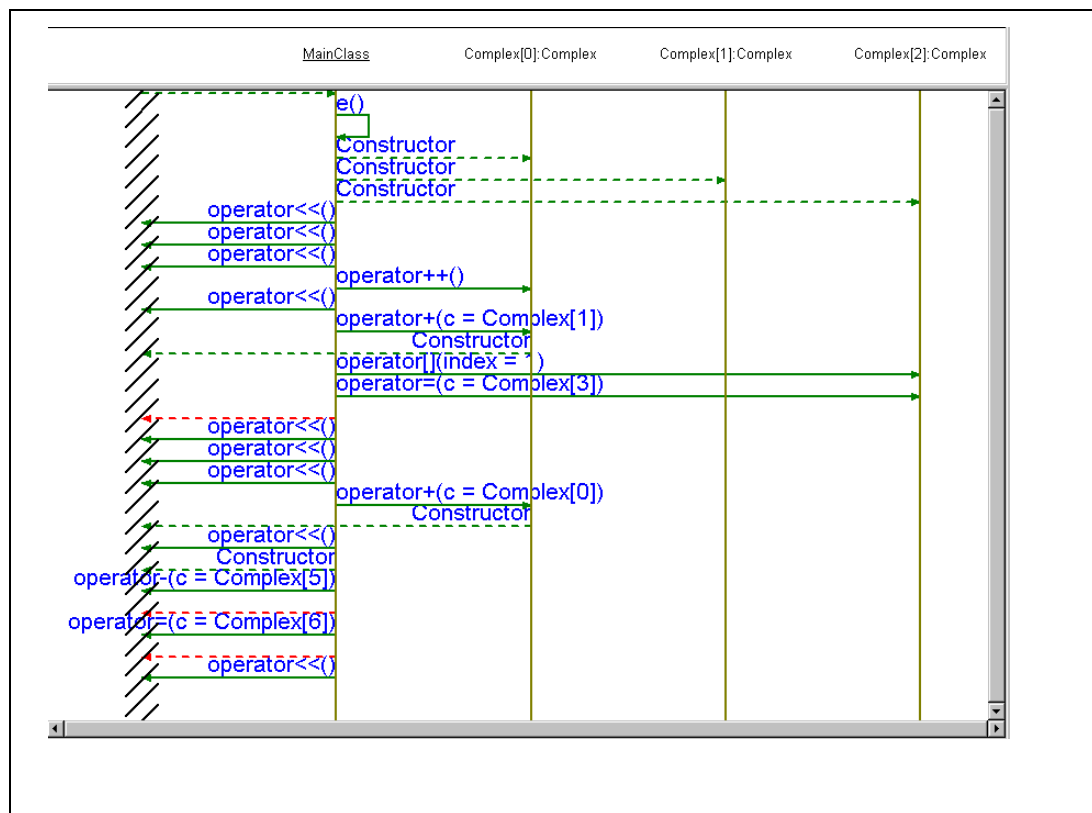
The stream output operator << is a global function that must be declared a friend to classes that want to use it. It is defined as follows:

```
operator<<
Return type: ostream&
Arguments:  ostream& s
           Complex& c
Body:
{
    s << "real part = " << c.real<<
        "imagine part = " << c.imag << "\n" << flush;
    return s;
}
```

To watch the various constructors and overloaded operators as they are being called, follow these steps:

1. Assign animation instrumentation to the project by selecting **Code > Set Configuration > Edit > Setting tab**.
2. Make and run `DefaultConfig.exe`.
3. Using the animator, create an animated sequence diagram (ASD) that includes the `MainClass` and instances `Complex[0]:Complex`, `Complex[1]:Complex`, and `Complex[2]:Complex`.
4. Click **Go** on the Animation bar to watch the constructor and overloaded operator messages being passed between the `MainClass` and its part instances.

The ASD will be similar to the following figure.



Using Anonymous Instances

In Rhapsody, you can create anonymous instances that you manage yourself, or create instances as components of composite instances that are managed by the composite framework.

Creating Anonymous Instances

You can create anonymous instances, apply the C++ `new` operator as in any conventional C++ program. Rhapsody generates a default constructor (`ctor`) for every class in the system. You can add additional constructors using the browser.

For example, to create a new instance of class `A` using the default constructor, enter the following code:

```
A *a = new A();
```

For reactive and composite classes, the default constructor takes a thread parameter that, by default, is set to the system's main thread. To associate the instance with a specific thread rather than the system's main thread, you must explicitly pass this parameter to the constructor.

The following example creates an instance of class `A` and assigns it to a thread `T`.

```
A *a = new A(T);
```

After creating a new instance, you would probably call its relation mutators to connect it to its peers (see [Using Relations](#)). If the class is reactive, you would probably call its `startBehavior()` method next.

Composite instances manage the creation of components by providing dedicated operations for the creation of new components. For each component, there is an operation `phil` of type `Philosopher`. The new `phil` operation creates a new instance, adds it to the component relation, and passes the thread to the new component.

The following code shows how the composite `sys` can create a new component `phil`.

```
Philosopher *pPhil = sys->newPhil();
```

After creating the new instance, you would probably call its relation mutators to connect it to its peers. If the class is reactive, you would probably call its `startBehavior()` method next.

Deleting Anonymous Instances

In Rhapsody, you manage anonymous instances yourself. As a result, it is your responsibility to delete them. Components of composite instances are managed by the composite. To delete them, you must make an explicit request of the composite object.

Apply the C++ `delete` operator as in any conventional C++ program. Deletion of instances involves applying the destructor of that instance. Rhapsody generates a default destructor for every class in the system. You can add code to the destructor through the browser.

For example, to delete an instance pointed to by `aB`, use the following call:

```
delete aB;
```

Deleting Components of a Composite

For each component of a composite, there is a dedicated operation that deletes an instance of that component.

For example, deleting an instance pointed to `aB` from a component named `compB` in a composite `C`, use the following call:

```
C->deleteCompB(aB);
```

Using Relations

Relations are the basic means through which you reference other objects. Relations are implemented in the code using container classes. The Rhapsody framework contains a set of container classes used by default. You can change the default, for example to use Standard Template Library (STL)TM containers, using properties. Refer to the *Rhapsody Properties Reference Manual* for more information.

The collections and relation accessor and mutator functions documented in this section refer to the default set provided with Rhapsody.

Note

To quickly see the relations for a class, object, and package, right-click the element in the Rhapsody browser and select **Show Relations in New Diagram** from the pop-up menu. For more information about this pop-up menu command, see [Showing All Relations for a Class, Object, or Package in a Diagram](#).

To-One Relations

To-one relations are implemented as simple pointers. Their treatment is very similar to that of attributes; that is, they also have accessor and mutator functions.

If B is a class related to A by the role name `role`, A contains the following data member:

```
B* role;
```

It contains the following methods:

```
B* getRole();  
void setRole(B* p_B);
```

These defaults are modifiable through the properties of the role. Refer to the *Rhapsody Properties Reference Manual* for more information.

To-Many Relations

To-many relations are implemented by collections of pointers using the `OMCollection` template.

If E is a class name multiply related to F by role name `role`, E contains the following data member:

```
OMCollection<F*> role;
```

The following methods are generated in E to manipulate this relation:

- ◆ To iterate through the relation, use the following accessor:

```
OMIterator<F*> getRole() const;
```

For example, if you want to send event x to each of the related F objects, use the following code:

```
OMIterator<F*> iter(anE->getRole());
while(*iter)
{
    *iter->GEN();
    iter++;
}
```

In this code, anE is an instance of E .

- ◆ To add an instance to the collection, use the following call:
`void addRole(F* p_F);`
- ◆ To remove an instance from the collection, use the following call:
`void removeRole(F* p_F);`
- ◆ To clear the collection, use the following call:
`void clearRole();`

These defaults are modifiable through the properties of the role. Refer to the *Rhapsody Properties Reference Manual* for more information.

Ordered To-Many Relations

Ordered to-many relations are implemented by collections of pointers using the `OMList` template.

A to-many relation is made ordered by making the `Ordered` property for the relation to `Checked`.

All the manipulation methods are the same as described in [To-Many Relations](#).

Qualified To-Many Relations

A qualified to-many relation is a to-many relation qualified by an attribute in the related class. The qualified to-many is implemented by a collection of pointers using the `OMMap` template.

All the manipulation methods are the same as described in [To-Many Relations](#). In addition, the following key qualified methods are provided.

```
F* getRole(type key) const;
void addRole(type key, F* p_F);
void removeRole(type key);
```

Random Access To-Many Relations

A random access to-many relation is a to-many relation that has been enhanced to provide random access to the items in the collection.

A to-many relation is made random access by making the `GetAtGenerate` property for the relation to `Checked`. This causes a new accessor to be generated:

```
F* getRole(int i) const;
```

Support for Static Architectures

Static architectures are often used in hard real-time and safety-critical applications with memory constraints. Rhapsody provides support for applications without memory management and those in which non-determinism and memory fragmentation would create problems by completely avoiding the use of the general memory management (or heap) facility during execution (after initialization). This is a typical requirement of safety-critical systems.

Rhapsody can avoid the use of the general heap facility by creating special allocators, or local heaps, for designated classes. The local heap is a preallocated, continuous, bounded chunk of memory that has the capacity to hold a user-defined number of objects. Allocation of local heaps is done via a safe and simple algorithm. Use of local heaps is particularly important for events and triggered operations.

Rhapsody applications implicitly and explicitly result in dynamic memory operations in the following cases:

- ◆ **Event generation (implicit)**—Optionally resolved via local heap
- ◆ **Addition of relations**—Resolved by implementing with static arrays (dynamic containers remain dynamic)
- ◆ **Explicit creation of application objects via the `new` operator**—Resolved via local heap if the application indeed dynamically creates objects

You can specify whether local heaps apply to all or only some classes, triggered operations, events, and thread event queues.

Properties for Static Memory Allocation

The following table lists some of the properties that enable you to configure static allocations for the generated code. Setting any of these properties at a level higher than an individual instance sets the default for all instances. For example, setting a class property at the component level sets the default for all class instances. In all cases, behavior is undefined if the actual number of instances exceeds the maximum declared number.

Property	Subject and Metaclass	Description
BaseNumberOfInstances	<lang>_CG::Class CG::Event	<p>Specifies the size of the local heap memory pool allocated for either:</p> <ul style="list-style-type: none"> • Instances of the class (<lang>_CG::Class) • Instances of the event (<lang>_CG::Event) <p>This property provides support for static architectures found in hard real-time and safety-critical systems without memory management capabilities during run time. All instances of events are dynamically allocated during initialization.</p> <p>Once allocated, a thread's event queue remains static in size.</p> <p>Triggered operations use the properties defined for events.</p> <p>When the memory pool is exhausted, an additional amount, specified by the <code>AdditionalNumberOfInstances</code> property, is allocated.</p> <p>Memory pools for classes can be used only with the Flat statechart implementation scheme.</p>
AdditionalNumberOfInstances	<lang>_CG::Class CG::Event	<p>Specifies the number of instances or events for which memory is allocated when the current pool is empty.</p>
ProtectStaticMemoryPool	CG::Class/Event	<p>Determines whether to protect access to the allocated memory pool using an operating system mutex to provide thread safety.</p>

Property	Subject and Metaclass	Description
MaximumPendingEvents	CG::Class	Specifies the maximum number of events that can be simultaneously pending in the event queue of the active class.

See the definition provided for each property on the applicable **Properties** tab of the Features dialog box.

Events generated in an interrupt handler should not rely on rescheduling and therefore should not cause the use of an operating system mutex. In other words, `ProtectStaticMemoryPool` should be `False`. It is up to you to make sure that the memory pool is not thread-safe. The animation framework is not subject to this restriction.

The framework files include reachable code for both dynamic and static allocation. Actual usage is based on the generated code. Use of the `GEN` and `gen` macros is the same for both modes.

Implementation of fixed and bounded relations with static arrays via the `<lang>_CG:Relation::ImplementWithStaticArray` property implicitly uses static allocation apart from initialization.

Static Memory Allocation Algorithm

Rhapsody implements static memory allocation by redefining the `new` and `delete` operators for each event and class that use local heaps. The memory allocator allocates enough memory to hold n instances of the specific element, where n is the value of the `BaseNumberOfInstances` property. The memory allocation is performed during system construction and uses dynamic memory. The memory allocator uses a LIFO (stack) algorithm, as follows:

- ◆ Claimed memory is popped off the top of the stack, and the top pointer is moved to point to the next item.
- ◆ Returned memory is pushed onto the top of the stack, and the top pointer is moved to point to the returned item.

The generated class—whether a class, event, or triggered operation—is instrumented to introduce additional code needed for use by the memory allocator (specifically the `next` pointer).

Attempts to instantiate a class whose memory pool is exhausted result in a call to the `OMMemoryPoolIsEmpty()` operation (in which you can set a breakpoint) and in a tracer message. Failure to instantiate results in a tracer message.

Containment by Value via Static Architecture

Rhapsody normally generates containment by reference rather than containment by value for `1-to-MAX` relationships, regardless of the relationship's type (set in the diagram). However, the static architecture feature enables you achieve the effect of containment by value by defining the maximum number of class instances and event instances via the static architecture properties, and thus avoiding the use of the default, non-deterministic `new()` operator.

Static Memory Allocation Conditions

If the application uses static memory allocation, the checker verifies that the following conditions are met:

- ◆ The maximum number of class instances is non-zero.
- ◆ The statechart implementation is flat.
- ◆ The `new` and `delete` operators are explicitly specified for each event and class using local heaps.

Reports include limits set for the number of instances.

All properties are loaded and saved as part of the element definition in the repository.

Static Memory Allocation Limitations

The following limitations apply to static memory allocation:

- ◆ Support is not yet provided for allocation of arrays. Instances must be allocated one-by-one. This is because of the (unknown) memory overhead associated with allocation of arrays.
- ◆ The `tm()` timeout function is not yet supported.

Using Standard Operations

Standard operations are class operations that are standardized by a certain framework or programming style. Notable examples are serialization methods that are part of distribution frameworks (see [Example](#)), and copy or convert constructors. Such operations generally apply aggregate functions to class features (such as attributes, operations, and superclasses). The following sections describe how to add standard operations to the code generated for classes and events.

Applications for Standard Operations

Standard operations can be used to follow these steps:

- ◆ Create event serialization methods.
- ◆ Support canonical forms of classes.
- ◆ Make a class persistent.
- ◆ Support a particular framework.

Event Serialization

1. An `Event` class is instantiated, resulting in a pointer to the event.
2. The event is queued by adding the new event pointer to the receiver's event queue.

Once the event has been instantiated and added to the event queue of the receiver, the event is ready to be "sent." The success of the send operation relies upon the assumption that the memory address space of the sender and receiver are the same. However, this is not always the case.

For example, the following are some examples of scenarios in which the sender and receiver memory address spaces are most likely different:

- ◆ The event is sent between different processes in the same host.
- ◆ The event is sent between distributed applications.
- ◆ The sender and receiver are mapped to different memory partitions.

One common way to solve this problem is to marshall the information. *Marshalling* means to convert the event into raw data (or serialize it), send it using frameworks such as `publish/subscribe`, and then convert the raw data back to its original form at the receiving end. High-level solutions, such as CORBA, automatically generate the necessary code, but with low-level solutions, you must take explicit care. Standard operations enable you to specify to a fine level of detail how to marshall, and unmarshall, events and instances.

Canonical Forms of Classes

Many style guidelines and writing conventions specify a list of operations to include in every class. Common examples are copy and convert constructors, and the assignment operator.

Rhapsody enables you to define standard operations such as these for every class.

Persistence

Making classes persistent is usually achieved by adding an operation with a predefined signature to every class. This can be done with a standard operation.

Support for Frameworks

Many object-oriented frameworks are used by deriving subclasses from a base class and then overriding the virtual operations. For example, MFC constructs inherit from the `CObject` class and then override operations such as the following:

```
virtual void Dump(CDumpContext& dc) const;
```

This is another example of something that can be done with standard operations.

You can add framework base classes using the `<lang>_CG::Class::AdditionalBaseClasses` property.

Creating Standard Operations

For every standard operation defined, you must specify an operation declaration and definition. This is done by adding the following two properties to the `site.prp` file to specify the necessary function templates:

- ◆ `<LogicalName>Declaration`—Specifies a template for the operation declaration
- ◆ `<LogicalName>Definition`—Specifies a template for the operation implementation

For example, with a logical name of `myOp`, you would add the following property (using the `site.prp` file or the COM API (VBA)):

```
Subject CG
  Metaclass Class
    Property myOpDeclaration MultiLine ""
    Property myOpDefinition MultiLine ""
  end
```

You add all of the properties to be associated with a standard operation to the `site.prp` file under their respective CG subject and metaclasses. All of these properties should have a type of `MultiLine`.

Keywords

The template can contain the following keywords:

- ◆ `$Attributes`—For every attribute in the class, this keyword is replaced with the contents of the template specified in the attribute's property `CG::Attribute::<LogicalName>`. For example, `<lang>_CG::Attribute::myOp`.
- ◆ `$Relations`—For every relation in the class, this keyword is replaced with the contents of the template specified in the relation's property `CG::Relation::<LogicalName>`. For example, `<lang>_CG::Relation::myOp`.
- ◆ `$Base (visibility)`—For every superclass/event, this keyword is replaced with the content of the template specified in the superclass/event property `CG::Class/Event::<logicalName>Base`. For example, `CG::Class::myOpBase`.
- ◆ `$Arguments`—For every argument in the class, this keyword is replaced with the contents of the template specified in the argument's property `CG::Argument::<LogicalName>`. For example, `<lang>_CG::Argument::myOp`.

Note

If you do not set the new location (`CG.Argument.<standardOpName>`), the old location (`CG.Event.<standardOpName>Args`) is still valid.

When expanding the properties, you can use the following keywords:

- ◆ `$(Name)`—Specifies the model name of the class (attribute, relation, operation, or base class), depending on the context of the keyword. If applicable, `$Type` is the type associated with this model element.
- ◆ `$ImplName`—Specifies the name of the (generated) code. This is usually the data member associated with this model element.

This is useful in RiC, where the logical name can be replaced in the generated code to a more complex name.

In addition, you can create custom keywords that relate to any property in the context using the convention `$(property name)`. For example, if you added a property `CG::Class::Foo`, you can relate to the property content in `CG::Class::myOpDefinition` by `$Foo`.

Note that this feature, combined with stereotype-based code generation, lets a power user define a set of standard operations, associate them to stereotypes, and lets other members of the team apply the stereotypes—getting the standard operations without worry over the properties.

Example

If you need to implement a serialization of events for all the events in a certain package, the package's `CG::Event::StandardOperations` property will contain “serialize, unserialize.” If the serialization is done by placing the event argument values inside an `stringstream`, the corresponding properties and their values are as follows:

CG::Event::SerializeDeclaration

```
public:
    stringstream & serializer(stringstream & theStrStream)
    const;
```

CG::Event::SerializeDefinition

```
stringstream & $Name::serialize(stringstream & theStrStream) {
    $Base
    $Arguments
    return theStrStream;
}
```

CG::Event::SerializeArgs

```
theStrStream << $Name;
```


CG::Event::SerializeBase

```
$Name::serialize(theStrStream);
```

If you defined an event `VoiceAlarm` with two arguments, `severity` and `priority`, and `VoiceAlarm` inherits from `Alarm`, the resulting code for the template would be as follows:

```
ostream & VoiceAlarm::serialize(ostream & theStrStream) {  
    Alarm::serialize(theStrStream);  
    theStrStream << severity;  
    theStrStream << priority;  
    return theStrStream;  
}
```

The same process is done for the `unserialize` part.

Providing Support for Java Initialization Blocks

Java initialization blocks are used to perform initializations before instantiating a class, such as loading a native library used by the class. You can use standard operations to provide a convenient entry point for this language construct, as follows:

1. Open the `site.prp` file and add the following property:

```
Subject CG
  Metaclass Class
    Property StandardOperations String "InitializationBlock"
  end
end
```

2. Open the `siteJava.prp` file and add the following property:

```
Subject JAVA_CG
  Metaclass Class
    Property InitializationBlockDeclaration MultiLine ""
  end
end
```

3. For every class that needs an initialization block, enter the text in the class's `InitializationBlockDeclaration` property. For example:

```
static {
  System.loadLibrary("MyNativeLib");
}
```

The text will be entered below the class declaration.

Statechart Serialization

Rhapsody provides a mechanism for serialization of reactive instances. By setting a number of Rhapsody properties, you can have methods added to the generated code, which you can then use to implement serialization.

Note

This feature is available only for C and C++ code.

Generating Methods for Serialization

The property `[C][CPP]_CG::Statechart::StatechartStateOperations` determines whether the code is generated for this feature. The possible values for this property are:

- ◆ **None** (default value)—code is not generated for the feature
- ◆ **WithoutReactive**—Rhapsody does not generate calls to `OMReactive`
- ◆ **WithReactive**—Rhapsody generates calls to `OMReactive`

Note

In C++, when using inheritance, a hierarchy of classes must use the same value for the property `CPP_CG::Statechart::StatechartStateOperations`.

Serialization Properties

The following properties, listed for C++ and C respectively, are related to the use of the serialization methods:

- ◆ `CPP_CG::Framework::ReactiveGetStateCall`
Default value is `OMReactive::getState()`;
Defines the prototype of the `getState` framework method.
- ◆ `CPP_CG::Framework::ReactiveSetStateCall`
Default value is `OMReactive::setState(p_state)`;
Defines the prototype of the `setState` framework method.
- ◆ `CPP_CG::Framework::ReactiveStateType`
Default value is `unsigned long`
Defines the `oxfstate` type.

- ◆ `C.CG::Framework::ReactiveGetStateCall`
Default value is `RiCReactive_getState(ric_reactive);`
Defines the prototype of the `getState` framework method.
- ◆ `C.CG::Framework::ReactiveSetStateCall`
Default value is `RiCReactive_setState(ric_reactive, p_state);`
Defines the prototype of the `setState` framework method.
- ◆ `C.CG::Framework::ReactiveStateType`
Default value is `long`
Defines the `oxfstate` type.

Methods Provided for Implementing Serialization

When the properties are set to generate the relevant code, the following public virtual member functions are generated for reactive classes:

Note

For the C functions, the prefix `<class name>` refers to the “class” in the model, for which the statechart was created.

- ◆ `virtual int getStatechartSize()—C++`
`int <class name>_getStatechartSize(<class name>* me)—C`
Returns the number of variables that the statechart uses. This function should be used to allocate the state vector that is passed to the function `getStatechartStates`.
- ◆ `virtual void getStatechartStates(int stateVector[], unsigned long& oxfReactiveState) const—C++`
`void <class name>_getStatechartStates(const <class name>* const me, int stateVector[], unsigned long* oxfReactiveState)—C`
Fills `stateVector` with the current statechart state, and sets `oxfReactiveState` based on the OMReactive internal state.

The type of `oxfReactiveState` is taken from the property `[C][CPP].CG::Framework::ReactiveStateType`.

The type of `stateVector` is taken from the property `CG::Statechart::FlatStateType` (default value is `int`).

In WithoutReactive mode, the last argument is eliminated and the function prototypes become:

```
- virtual void getStatechartStates(int stateVector[])  
  —C++  
- void <class name>_getStatechartStates(const <class name>*  
  const me, int stateVector[])—C
```

- ◆ virtual void setStatechartStates(int stateVector[], unsigned long* oxfReactiveState)—C++

```
void <class name>_setStatechartStates(<class name>* const me,  
int stateVector[], unsigned long* oxfReactiveState)—C
```

Set the reactive instance states as well as the OMReactive internal state.

The type of oxfReactiveState is taken from the property
[C] [CPP]_CG::Framework::ReactiveStateType.

The type of stateVector is taken from the property
CG::Statechart::FlatStateType (default value is int).

Note

If setStatechartState is used during animation, then the instance statechart will not display new states. In order to “refresh” the statechart, you can switch to Silent animation mode, run the animation, and then switch back to Watch animation mode.

Components-based Development in RiC

We enable component-based development in Rhapsody in C (RiC) by introducing code generation support for interfaces and ports.

A class may realize an interface, that is, provide an implementation for the set of services it specifies (that is, operations and event receptions). As in Rhapsody in C++ and Rhapsody in Java, you use a realization relationship to indicate that a class is realizing an interface. In addition, an interface may inherit another interface, meaning that it augments the set of interfaces the superinterface specifies. You can specify interfaces, realize them, and connect to objects via the interfaces.

RiC users can take advantage of service ports that allows the passing of operations and functions via ports, in addition to passing events. Just like in C++, you can specify ports with provided and required interfaces. In addition, Rhapsody 7.1 provides code generation support for standard UML ports in RiC and code generation of ports supports the initialization of links via ports. For more information about ports, see [Ports](#).

In this type of development in RiC, interfaces are treated as a specification of services (that is, operations) and **not** as inheritance of data (attributes). Also, in this type of development in RiC, realization (as opposed to inheritance) is used to distinguish between realizing an interface and inheriting an interface/class.

As of Rhapsody 7.1, code generation supports realizing interfaces in C. This means interfaces and ports specified in a C model will be implemented by the code generator. This means code generation generates:

- ◆ Code for a C interface (a class with “pure virtual operations”)
 - Virtual tables with function pointers
 - Relay code from the interface to the realizing class according to the virtual table
- ◆ The “realization code” for the realizing class
 - Aggregating the interface
 - Initializing the virtual table
- ◆ Links between objects that instantiate associations to the interface

Action Language for Code Generation

The following syntax is used for C code generation support for interfaces and ports. In these examples, we have an interface `x`, an operation `f`, a port `p`, and a class `A`.

Calling an operation via a C interface

```
[Interface]_[Operation] ([object realizing the interface]
[, argList])
```

Example: To call operation `x_f` (object realizing the interface, port number), where the port number is 5, do:

```
x_f(me->itsl, 5);
```

Sending an event via a C interface

```
RiCGEN_[Interface] ([object realizing the interface], [event([argList]])
```

Example: To send event `RiCGEN_l` (object realizing the interface, port number), do:

```
RiCGEN_l(me->itsl, evt());
```

Calling an operation via a C port

```
[Interface]_[Operation] (OUT_PORT([class], [port], [interface])
[, argList])
```

Example: To call operation `x_f` (object realizing the port, port number), where the port number is 5, do:

```
x_f(OUT_PORT(A, p, x), 5);
```

Sending an event via a C rapid port

```
RiCGEN_PORT([pointer to port], [event])
```

Example: To send event `RiCGEN_PORT` (object realizing the port, event), do:

```
RiCGEN_PORT(me->p, evt());
```

Sending an event via a C rapid port using ISR

```
RiCGEN_PORT_ISR([pointer to port], [event])
```

Example: To send event `RiCGEN_PORT_ISR`, do:

```
RiCGEN_PORT_ISR(me->p, evt());
```

Querying the port through which the event was received

```
RiCIS_PORT([object], [pointer to port])
```

Example: To query port `RiCIS`, do:

```
RiCIS_PORT(me, me->p);
```

Sending an event via a C non-rapid port

```
RiCGEN_PORT_I([class], [port], [interface], [event([argList]])])
```

Example: To send event `RiCGEN_PORT_I`(object realizing the port, event), do:

```
RiCGEN_PORT_I(A, p, x, evt());
```

C Optimization

Note the following:

- ◆ If a port has only provided interfaces, or just required interfaces, code generation generates a single additional inner part.
- ◆ Ports that are purely reactive are implemented as rapid ports. For more information about rapid ports, see [Using Rapid Ports](#).

Backward Compatibility

To enable interface realization for models made before Rhapsody 7.1, you must set the `C_CG::Class::InterfaceGenerationSupport` property to `Checked`. The choices are as follows:

- ◆ `Checked` means interfaces are generated with virtual tables and can be realized.
- ◆ `Cleared` means the `Interface` stereotype is ignored. Meaning you can turn off the interface generation ability.

See the definition provided for the property on the applicable **Properties** tab of the Features dialog box.

Note the following:

- ◆ If the `C_CG::Class::InterfaceGenerationSupport` property is set to `Cleared` for at least one of the interfaces in the port's contract, the port is treated as a rapid port.
- ◆ `C_CG::Class::InterfaceGenerationSupport` is a backward compatibility property only and is not listed in new models.

Limitations

Note the following code generation limitations:

- ◆ There is no general inheritance support.
- ◆ Ports code is generated into separate files (meaning that there are no nested classes in Rhapsody in C).
- ◆ An inheritance from a class to an interface is interpreted as a realization.
- ◆ No code is generated for a `<<friend>>` dependency to the template class.

Customizing C Code Generation

One of the key features of Rhapsody is the ability to generate code based on a Rhapsody model. There are two primary methods you can use to customize code generation:

- ◆ **Modifying the values of various properties in Rhapsody.** This method is available for all Rhapsody versions (C, C++, Java, and Ada). These properties are found under the `CG` and `<lang>_CG` subjects (for example, `CG::Package::UseAsExternal` and `JAVA_CG::Dependency::SpecificationEpilog`). For more information about properties, refer to the *Rhapsody Properties Manual* and the *Rhapsody Property Definitions*.
- ◆ **Using rules.** You may want to use this method if you have significant changes in the generated code where it is not enough to use properties and you want to have a starting point that is the out-of-the-box code generation. This method is available only for Rhapsody in C.

Note: You can also write your own code generator with the use of the RulesComposer tool. Note that you must have a valid license to be able to use this tool.

The using-rules method is described in detail in this section, including the conceptual basis for this customization mechanism, as well as specific instructions for customizing code generation.

Note that both methods let you control the content and appearance of the generated code. These two mechanisms co-exist and can be referred to as basic (using properties) and advanced (using rules) customization, respectively.

Code Customization Concepts

The process of converting a generic UML model into code can be divided into the following phases:

1. **Transformation.** The transformation phase of the original model into a refined model takes into account the elements of the specific programming language in which the code will be generated. In Rhapsody, this is called “Simplification” and the properties related to it begin with the word “Simplify” (for example, `C_CG::ModelElement::SimplifyAnnotations`).
2. **Writing.** This is the conversion of the refined model into valid code in the chosen target language.

Customizing Code Generation

Note

The customization feature is available only for Rhapsody in C.

When you instruct Rhapsody to generate code, Rhapsody can take a number of different paths, depending on the value of the property `C.CG::Configuration::CodeGeneratorTool`.

If `CodeGeneratorTool` is set to a value other than `Customizable`, Rhapsody will invoke its standard internal code generation mechanism.

If `CodeGeneratorTool` is set to `Customizable`, Rhapsody carries out the following steps:

1. Creates a refined model from the original model. This model is referred to as the simplified model. (This step represents the *transformation* phase described in [Code Customization Concepts](#).)
2. Invokes the external RulesPlayer code writer to create the code itself. (This step represents the *writing* phase described in [Code Customization Concepts](#).)

Note: You must have a valid license to be able to use the RulesPlayer code writer.

When code generation is running in Rhapsody, you will see the following messages to show that the RulesPlayer is at work:

```
Loading external generator...
Invoking RulesPlayer
Evaluation of RicWriter.
```

Both of these steps—creation of the simplified model and generation of code from the simplified model—can be customized, as described in [Customizing the Generation of the Simplified Model](#) and [Customizing the Code Writer](#).

Viewing the Simplified Model

When the property `C_CG::Configuration::CodeGeneratorTool` is set to `Customizable`, a simplified model is created automatically as the first step of the code generation process.

By default, the simplified model is not displayed in Rhapsody. To have the simplified model displayed in the browser, set the property `C_CG::Configuration::ShowCGSimplifiedModelPackage` property to `Checked`. Once you have done so, the next time you generate code, the simplified model will be added automatically at the top of the project tree in the Rhapsody browser.

Customizing the Generation of the Simplified Model

Rhapsody contains a default mapping that determines how model elements are treated when generating a simplified model.

Properties Used for Simplification

To change the way specific types of elements are handled, you modify the properties that control simplification. For each type of model element, there is a property that determines how it will be handled during the transformation of the model, for example, `SimplifyConstructors` and `SimplifyDestructors`.

Each of these properties can take any of the following values. Note that these values may not appear for every Simplify property.

- ◆ **None.** The element is ignored in the simplified model.
- ◆ **Copy.** The element is just copied from the original to the simplified model. It is not modified in any way.
- ◆ **Default.** Uses the standard simplification for this item, as defined in Rhapsody.
- ◆ **ByUser.** Uses the customized simplification provided by the user. For details, see [User-Provided Simplification \(ByUser option\)](#).
- ◆ **ByUserPostDefault.** Uses the customized simplification provided by the user, but only after Rhapsody's standard simplification for the element has been applied.

User-Provided Simplification (ByUser option)

If you have indicated in the property settings that a user-provided simplification is to be used for a given type of element, then the code generation process invokes the user-provided code for transforming the model. The basics of this process are as follows:

- ◆ The user-provided transformation code is provided as a Rhapsody plug-in.

- ◆ You add this plug-in information to the `rhapsody.ini` file, or provide the information necessary to have the plug-in run only for a certain profile.
- ◆ During the code generation process, Rhapsody checks whether the user-provided code has implemented the relevant “simplify” interface for the element in question. (These interfaces are defined in the Rhapsody API.)

Note: Rhapsody provides you with sample projects that show the “simplify” interface. Look in the `<Rhapsody installation path>\Samples\CustomCG Samples` path. For example, see the sample projects provided in the `Statechart_Simplifier_Writer` subfolder. You should review the `Readme.txt` file that accompanies each sample project for details about that project.

- ◆ If the user-provided code implements the “simplify” interface, your implementation is called.
- ◆ The user-provided transformation code uses the Rhapsody API to directly modify the way model elements are transformed.

Customizing the Code Writer

The rules for rules-based code generation are contained in the RiCWriter project. You customize the rules with the use of the RulesComposer tool. The RulesPlayer code writer generates code from the simplified model using these rules.

The following topic describes the steps that are required if you want to customize the rules used for generating code from the simplified model.

Note

For detailed information on how to use the RulesComposer, refer to the tutorial PDF provided with the installation of the product (look in `<Rhapsody installation path>\Rhapsody\Sodius\RulesComposer\help\tutorial`). There is also a changes document for the RulesComposer that you may find useful (look in `<Rhapsody installation path>\Rhapsody\Sodius\RulesComposer\help`).

Customizing the RiC Rules

To customize the rules used to generate code for a simplified model, follow these steps:

1. Open RulesComposer from Rhapsody, choose **Tools > RulesComposer**.
2. When RulesComposer opens, the RiCWriter project should already be open. If not, open the project manually by selecting **File > Import** in RulesComposer and selecting the `<rhapsody>\Share\CodeGenerator\GenerationRules\LangC\RuleSet\RiCWriter` folder. When you choose this directory, Eclipse will automatically load the RiCWriter project that it contains.

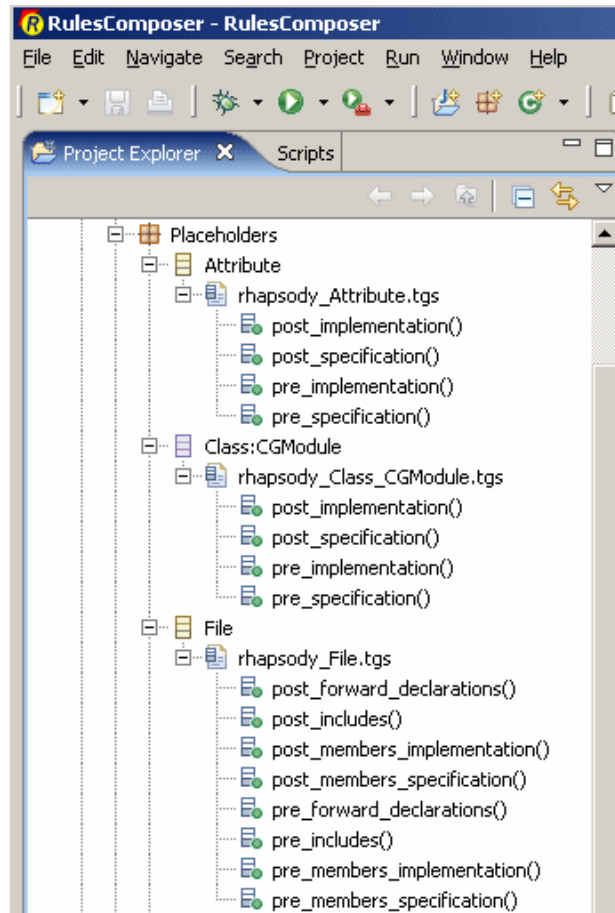
Note: The project is read-only by default. In order to modify the rules, you will need to change the relevant files to read-write.
3. Once the project is open, make your changes to the rules and script files (`.java`, `.tgs`). These are located in the `src` subfolder. Notice the **Placeholders** package. It contains hooks provided in the default rules for user customization. These hooks are empty scripts where you can enter code. These scripts are run from the existing rules at the appropriate time during code generation. For more information, see [Placeholders Package](#).
4. Save your changes to the RiCWriter project.
5. After saving your changes, you can test them by selecting **Run** in Eclipse. This will take your updated rules and apply them to the model that is currently open in Rhapsody. You can then look at the generated code to verify that the new rules had the desired effect.

Note

The updated rules can only be used to generate code if there is an existing simplified model to which they can be applied. This means that you must have already generated code with Rhapsody at least once for the model with the property `CodeGeneratorTool` set to `Customizable` and the property `ShowCGSimplifiedModel` set to `Checked`. (When the property `ShowCGSimplifiedModel` is set to `Cleared`, the simplified model is deleted after code generation has been completed. So in such a case, you would not have a simplified model to which the updated rules could be applied.)\

Placeholders Package

The **Placeholders** package, as *partially* shown in the following figure, contains hooks provided in the default rules for user customization. These hooks are empty scripts where you can enter code. These scripts are run from the existing rules at the appropriate time during code generation. For example, you can insert code in the **post_implementation()** script in the **Attribute** placeholder group. This script runs after implementation of the attribute rules.

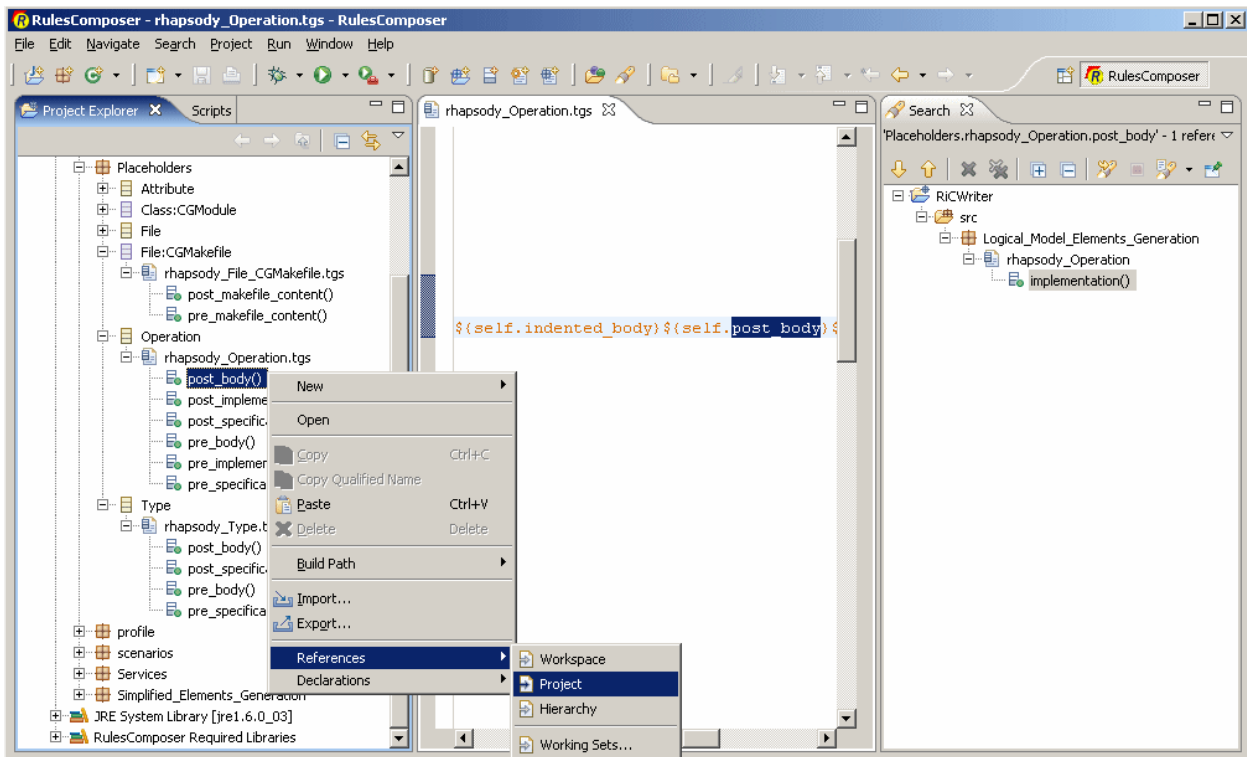


You can go to the reference for a script to find where the script is called.

To find a reference for a particular script, follow these steps:

1. Right-click the script on the RulesComposer Project Explorer, select **References**, and then select **Workspace**, **Project**, or **Hierarchy** (see left third of the following figure), which opens a Search tab in the RulesComposer.
2. Double-click the found script on the **Search** tab (see right third of the figure), which opens the applicable file in the middle of the RulesComposer window (see middle third of the figure).

Note: For illustrative purposes, the following figure shows the results *after* the above steps have been completed once.



Deploying the Changed Rules

Once you are satisfied that the changes to the rules have the desired effect on code generation, the last step is to deploy the rules to the `.jar` file that Rhapsody uses when it performs customized code generation.

To deploy the changed rules, follow these steps:

1. In RulesComposer, select **File > Export**.
2. For the export destination, select **RulesComposer > Deployable RulesComposer configuration**.
3. Click **Next**.
4. For the export directory, select `<rhapsody installation path>\Share\CodeGenerator\GenerationRules\LangC\CompiledRules`

Note: By default, the `.jar` file containing the existing rules (`RiCWriter.jar`) is read-only. Make sure to change this attribute before attempting to export the updated rules.

5. Under Export Options, select **Deploy JAR file**.
6. Click **Finish**. The new rules will be saved as `RiCWriter.jar`.

Note: Rhapsody looks for the compiled rules in the filename given for the property `C.CG::Configuration::GeneratorRulesSet`. The default value for this property is `$OMROOT\CodeGenerator\GenerationRules\LangC\CompiledRules\RiCWriter.classpath`.

7. To have Rhapsody implement the rules in the updated `.jar` file, you must close and then re-open Rhapsody after you have exported the `.jar` file.

Reverse Engineering

Reverse engineering lets you import legacy C, C++, and Java code into a Rhapsody model. It extracts design information from the code constructs found in the source file and builds a model corresponding to the intended design, as much as possible.

Note

You can also reverse engineer Ada code. In Rhapsody in Ada, choose **Tools > Reverse Engineer Ada Source Files**. For documentation about reverse engineering in Ada, refer to the documentation provided in `<Rhapsody installation path>\Sodius\RIA_CG\AdaRevEng\help`.

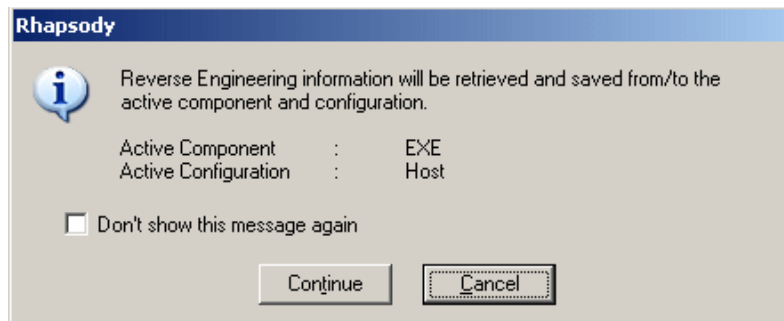
Once you generate a Rhapsody model from legacy code in the reverse engineering process, further edits to the model or to the code become synchronized in the roundtripping process thereafter. See [Roundtripping](#).

Using the Reverse Engineering Tool

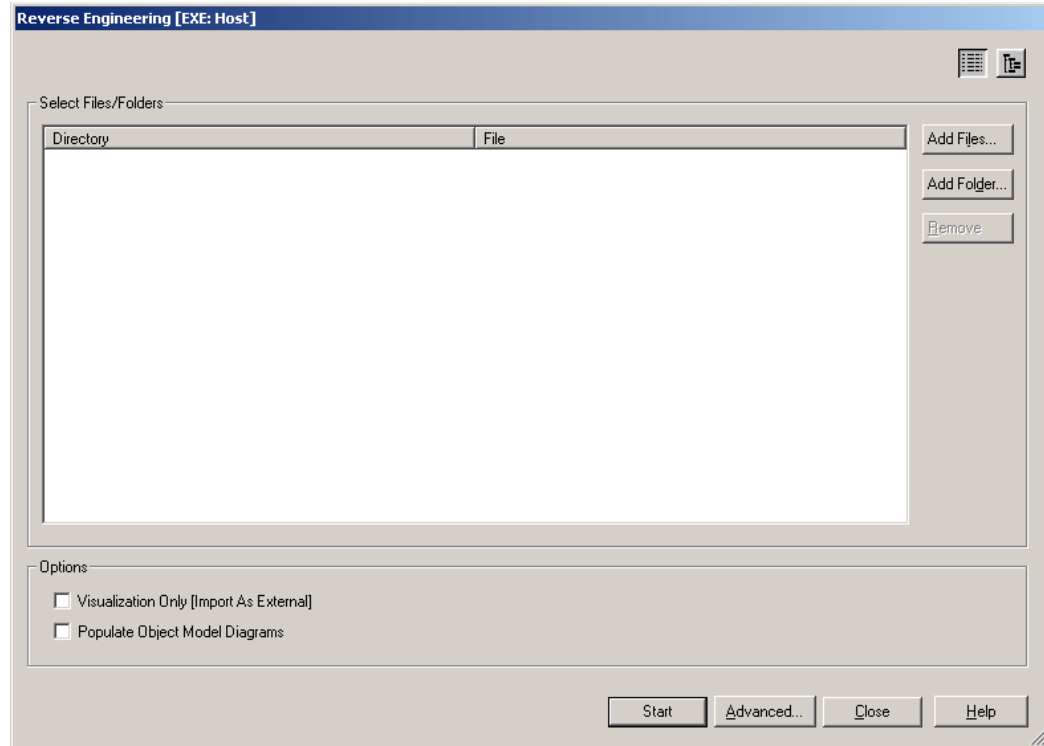
To use the reverse engineering tool to import legacy code, follow these steps:

1. Open the model into which you want to import legacy code and set which configuration you want to be the active configuration. See [Setting the Active Configuration](#).
2. Select **Tools > Reverse Engineering**. A display message appears to confirm that reverse engineering settings will be retrieved from/saved to and for which active component/configuration, as shown in the following figure.



Notice that simultaneously the Output window opens in the Rhapsody main window.



3. Click **Continue**. The Reverse Engineering dialog box opens, which defaults to show a flat view, as shown in the following figure.




In the flat view, the Reverse Engineering dialog box contains the following controls:

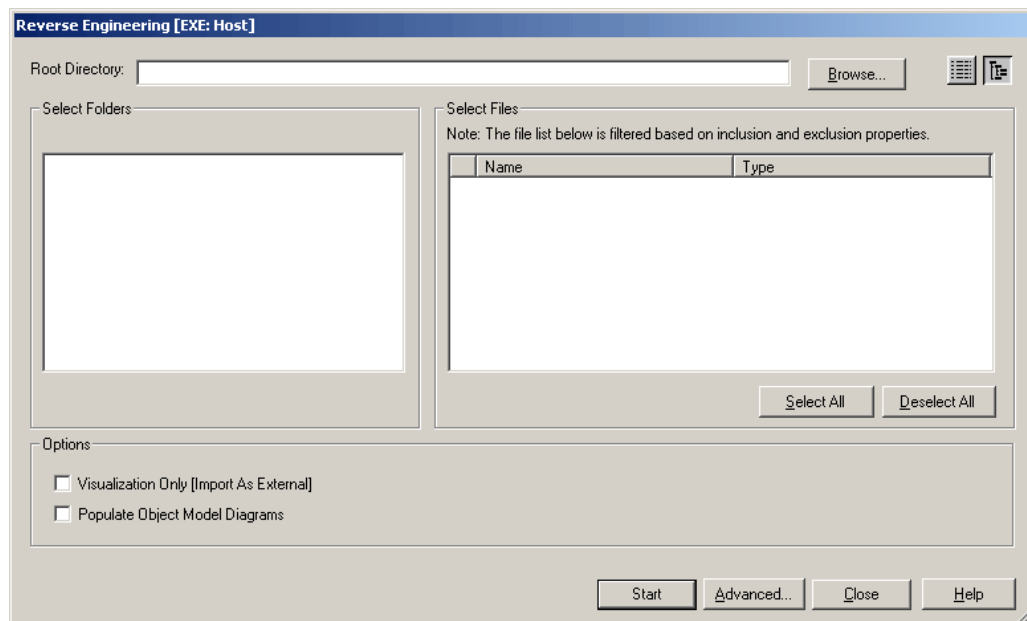
- ◆ Flat View button  and Tree View button , which you can use to toggle between the two views. A flat view shows folders and files in a list structure while a tree view shows a tree structure.
- ◆ **Add Files** button lets you add to the import file individual files (such as .h files for C and C++ projects) that you want to reverse engineer.
- ◆ **Add Folder** button lets you add to the import file a folder that contains files that you want to reverse engineer.
- ◆ **Remove** button, which is enabled only when applicable, lets you remove one or more highlighted items on the import list.
- ◆ **Options** group with **Visualization Only (Import as External)** and **Populate Object Model Diagrams** check boxes.

Select **Visualization Only (Import as External)**, when you want Rhapsody to add as external elements all the elements created in Rhapsody during reverse engineering (the `CG::Class/Package/Type::UseAsExternal` property is set to



Checked.) This means that while you can see the code using pictures and you can relate to it, the code is still maintained outside Rhapsody and is not generated by Rhapsody. This property is overridden for all packages (but not their contained elements, such as classes, files, types, unless the contained elements are also packages; and in that case they will also have their `CG::Package::UseAsExternal` property set to `Checked`). Note that when you select or clear the **Visualization Only (Import as External)** check box, this same check box is automatically set the same way on the **Mapping** tab of the Reverse Engineering Options dialog box. See [Mapping Classes to Types and Packages](#).

Select **Populate Object Model Diagrams** when you want imported object model diagrams to be automatically created in your project. Note that when you select or clear the **Populate Object Model Diagrams** check box, this same check box is automatically set the same way on the **Model Updating** tab of the Reverse Engineering Options dialog box. See [Updating Existing Packages](#).

- ◆ **Start** button lets you start the reverse engineering process. This button changes to **Stop** during the processing.
 - ◆ **Advanced** button gives you access to the Reverse Engineering Options dialog box.
 - ◆ **Close** button lets you close a dialog box without invoking whatever process is associated with the dialog box.
4. If you prefer, you can click the Tree View button  to show the dialog box in a tree view, as shown in the following figure.



In the tree view, the Reverse Engineering dialog box contains the following controls:

- ◆ Flat View button  and Tree View button , which you can use to toggle between the two views. A flat view shows folders and files in a list structure while a tree view shows a tree structure.
- ◆ **Browse** button lets you browse to where you have the files you want to reverse engineer. The folder you choose (which should contain files that can be reversed engineered, such as .h files for C and C++ projects) appears in the **Select Folders** box and **Select Files** boxes, as appropriate.
- ◆ **Select All** and **Deselect All** buttons, which work as labeled on the files in the **Select Files** box. You can use these buttons and then manually select or clear the check boxes next to the files that you do or do not want to include.

Note: You can also select the check boxes next to a folder name in the **Select Folders** box to quickly select or deselect everything (files and subfolders) in that folder.

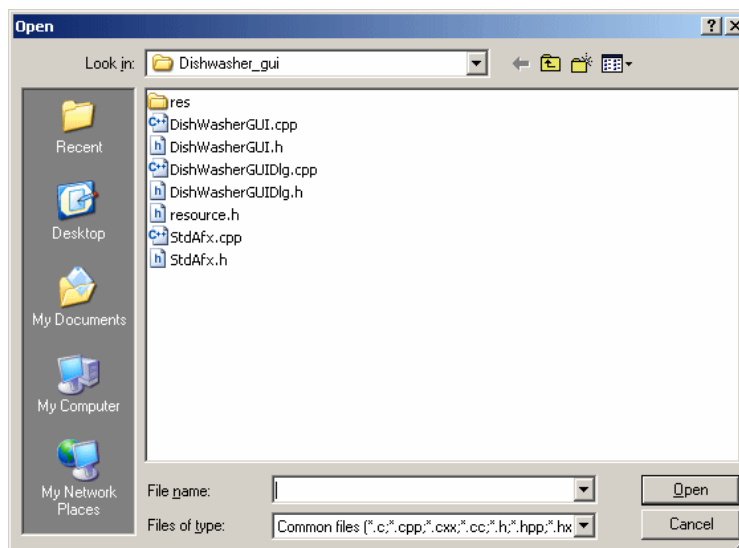
- ◆ **Options** group with **Visualization Only (Import as External)** and **Populate Object Model Diagrams** check boxes.

Select **Visualization Only (Import as External)**, when you want to set Rhapsody to add as external elements all the elements created in Rhapsody during reverse engineering (their `CG::Class/Package/Type::UseAsExternal` property is set to Checked.) This property is overridden for all packages (but not their contained elements, such as classes, files, types, unless the contained elements are also packages; and in that case they will also have their `CG::Package::UseAsExternal` property set to Checked). Note that when you select or clear the **Visualization Only (Import as External)** check box, this same check box is automatically set the same way on the **Mapping** tab of the Reverse Engineering Options dialog box. See [Mapping Classes to Types and Packages](#).

Select **Populate Object Model Diagrams** when you want imported object model diagrams to be created in your project. Note that when you select or clear the **Populate Object Model Diagrams** check box, this same check box is automatically set the same way on the **Model Updating** tab of the Reverse Engineering Options dialog box. See [Updating Existing Packages](#).

- ◆ **Start** button lets you start the reverse engineering process. This button changes to **Stop** during the processing.
 - ◆ **Advanced** button gives you access to the Reverse Engineering Options dialog box.
 - ◆ **Close** button lets you close the Reverse Engineering dialog box without invoking the reverse engineering process.
5. If you are using the flat view on the Reverse Engineering dialog box, go to step 6. If you are using the tree view, go to step 8:

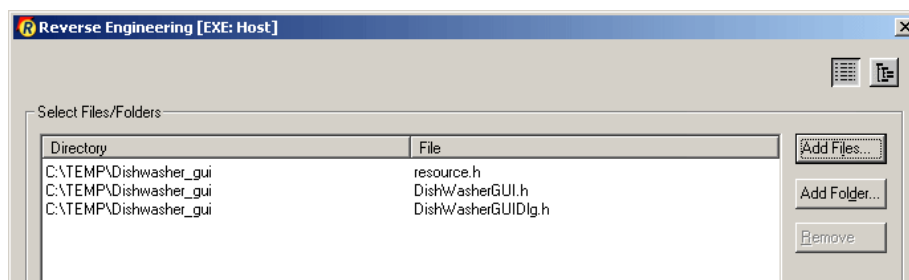
6. If you are using the flat view of the Reverse Engineering dialog box, do the following:
 - a. To select individual files, click **Add Files** to open the Open dialog box open and browse to the file(s) you want to import, as shown in the following figure. Then click **Open**.



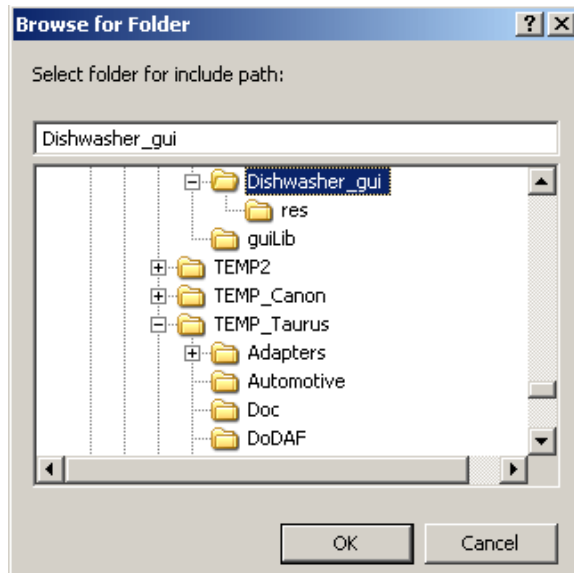
Note the following:

- ◆ Rhapsody does not import makefiles. Even if a configuration already has an existing makefile, Rhapsody generates another makefile and does not use the original.
- ◆ Files to be imported must contain compilable code with no syntax errors.

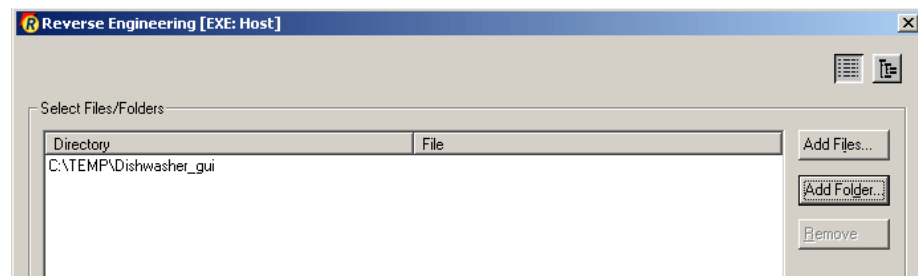
Rhapsody adds the selected file to the Reverse Engineering dialog box, as shown in the following figure.



- b. To include all the files in a folder for reverse engineering, click **Add Folder** to open the Browse for Folder dialog box, as shown in the following figure. This means you want to reverse engineer all the files (that is possible) in the folder. Click **OK**.

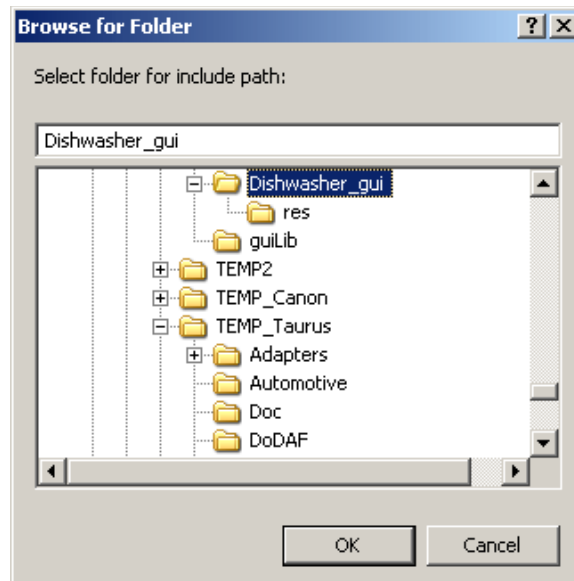


The folder is added to the Reverse Engineering dialog box, as shown in the following figure. Note that when you add files or folders to this dialog box, the list maintains itself from session to session so you can maintain the history of the file list.



7. Continue with step 9.

8. If you are using the tree view on the Reverse Engineering dialog box, do the following:
 - a. Click the **Browse** button to open the Browse to Folder dialog box and browse to the folder that has the legacy code you want to reverse engineer, as shown in the following figure, and click **OK**.

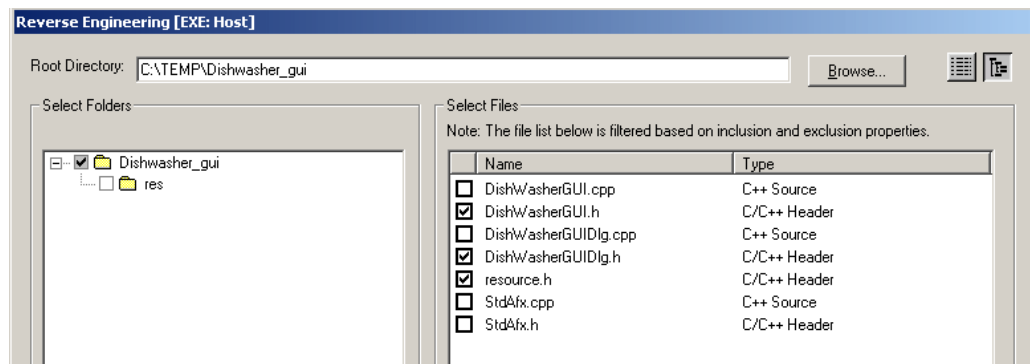


- b. On the Reverse Engineering dialog box, select the files you want to reverse engineer, as shown in the following figure.

In the **Select Files** box, you can select a check box next to one or more individual files. You can also use the **Select All** and **Deselect All** buttons.

In the **Select Folders** box, you can select the check box next to a folder to select all the files in that folder. Note that the appearance of the check boxes in the **Select Folders** box means the following:

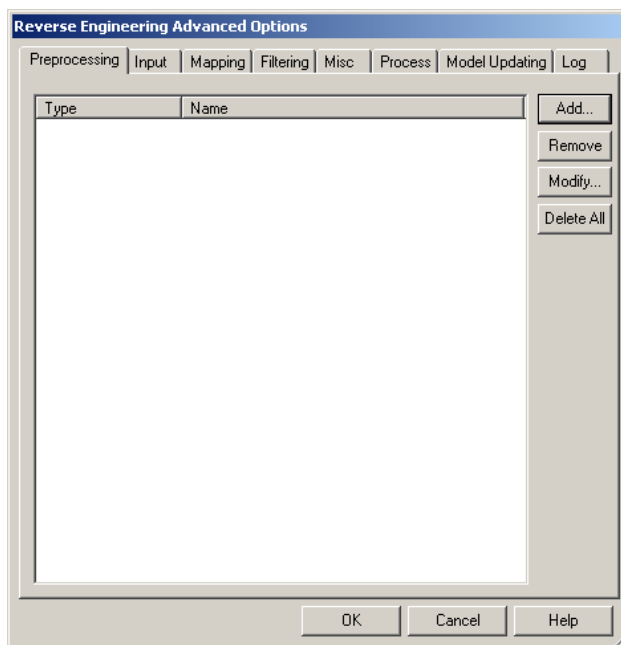
- All items (files and subfolders) in the folder has been selected.
- Some items have been selected (as illustrated in the following figure).
- No items have been selected.



Note: To specify the filename extensions that should be used to filter files in the Reverse Engineering dialog box, use the `<lang>_ReverseEngineering::Main::ImplementationExtension` and `<lang>_ReverseEngineering::Main::SpecificationExtension`. In addition, note that files that are matching the `ReverseEngineering::Main::ExcludeFilesMatching` property will be filtered out in contrast to the other two properties mentioned previously. For more about the `ReverseEngineering::Main::ExcludeFilesMatching` property, see [Excluding Particular Files](#). Also, see the definition provided for each property on the applicable **Properties** tab of the Features dialog box.

Note the following:

- ◆ Rhapsody does not import makefiles. Even if a configuration already has an existing makefile, Rhapsody generates another makefile and does not use the original.
 - ◆ Files to be imported must contain compilable code with no syntax errors.
9. Select any of the check boxes in the **Options** group, as applicable.
 10. To specify options for this reverse engineering session, click the **Advanced** button to open the Reverse Engineering Options dialog box, as shown in the following figure, which contains the following tabs:
 - ◆ **Preprocessing** (see [Defining Preprocessor Symbols](#))
 - ◆ **Input** (see [Analyzing #include Files](#))
 - ◆ **Mapping** (see [Mapping Classes to Types and Packages](#))
 - ◆ **Filtering** (see [Specifying Reference Classes](#))
 - ◆ **Misc** (see [Miscellaneous Options](#))
 - ◆ **Process** (see [Error Handling](#))
 - ◆ **Model Updating** (see [Updating Existing Packages](#))
 - ◆ **Log** (see [Message Reporting](#))



11. Click **OK** to close the Reverse Engineering Options dialog box.

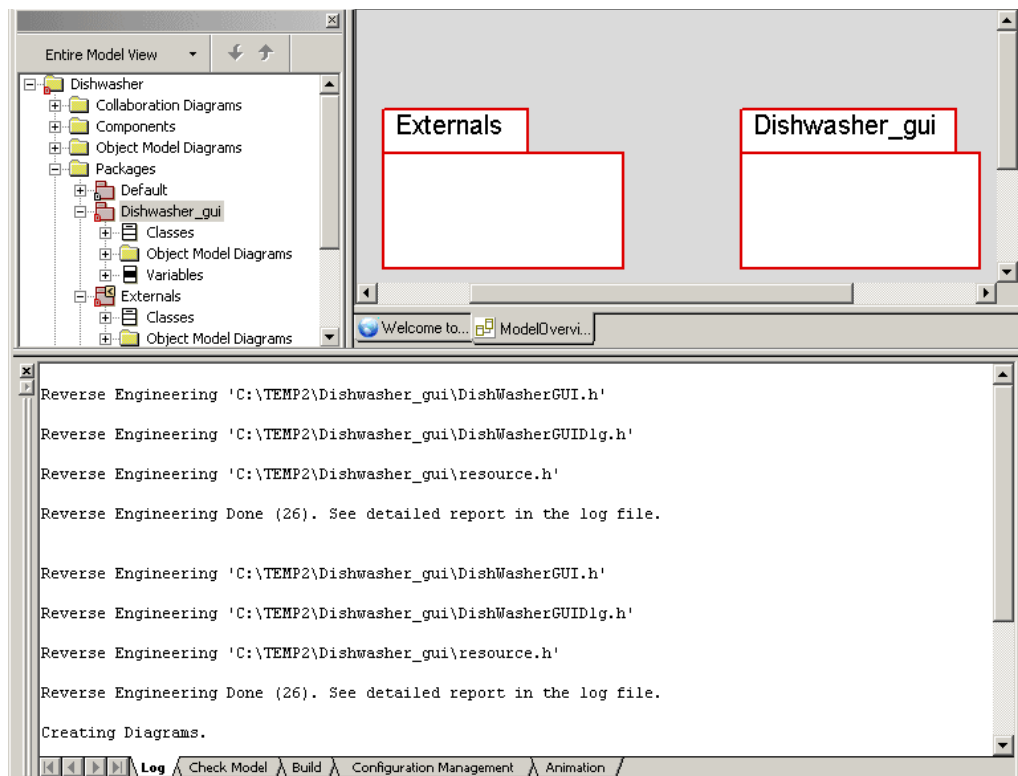
12. On the Reverse Engineering dialog box, click **Start** to begin the import.

Notice that the label for this button changes to **Stop**.

13. Click **OK** to confirm your requested action.
14. Wait while the reverse engineering process completes. When the **Stop** button changes to **Start** again, you can click **Close** to close the Reverse Engineering dialog box.

Note: Depending on the number of files in your folder(s) and the size of your files, the reverse engineering process may take some time. You may want to do a few files first to experience the process before you do whole folders or larger or multiple files at once.

15. You should see information about the reverse engineering in the Output window and in your Rhapsody browser, as shown in the following figure.



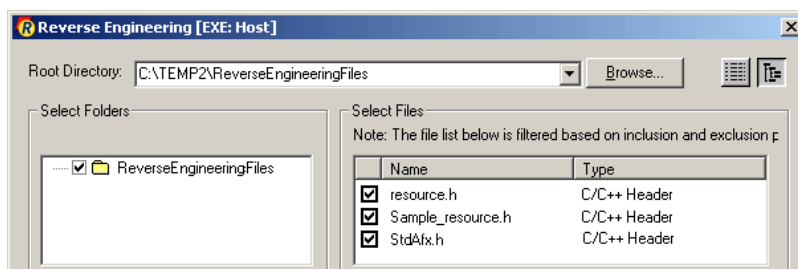
Messages are displayed in the Output window that indicate which files and constructs are being analyzed and which design elements are being added to the model. Note that the reverse engineering tool does not report on every item being parsed, nor does it report on ignored constructs. You can specify how or whether certain events are reported using the Log and Process options. See [Message Reporting](#) and [Error Handling](#) for more information.

Note that once a file has been successfully imported, Rhapsody does not provide a means to delete information associated with an imported file. If you need to delete imported elements, do it manually from the Rhapsody browser. See [Deleting Elements](#).

Initializing the Reverse Engineering Dialog Box

If the active configuration's properties already contain information about a reverse engineering configuration (list of files and other options), the Reverse Engineering dialog box will be initialized with this information by default. This initialization includes the following:

- ◆ The directory tree will be created and the selection status marked properly using the list of files saved with the `ReverseEngineering::Main::Files` property. Be sure to include the backslash (for Windows systems) at the end of a path (for example: `C:\TEMP2\ReverseEngineeringFiles\`).
- ◆ When you open the Reverse Engineering tool, the file list will be created and selection status marked accordingly for each item on the directory tree, as shown in the following figure. Notice the three files in the **Select Files** box.

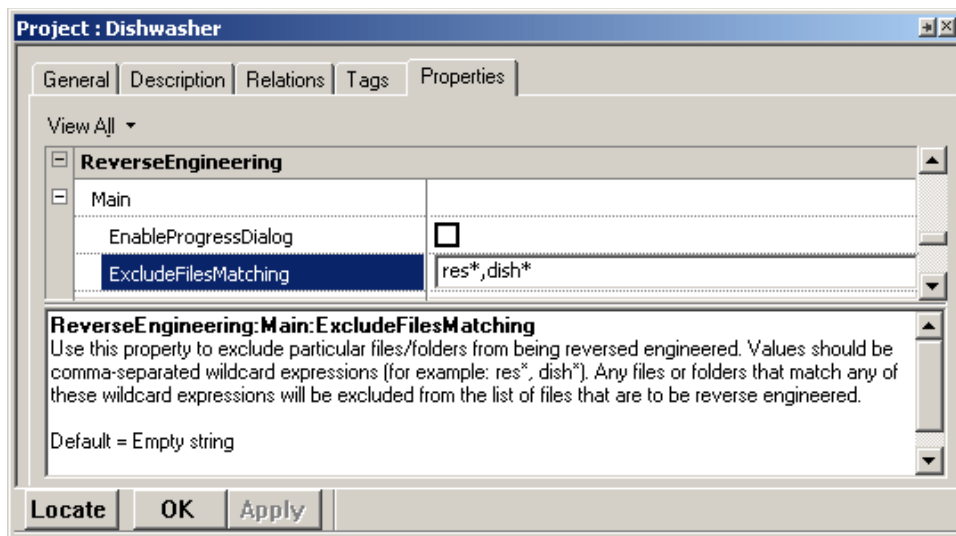


- ◆ The options in the Reverse Engineering dialog box and the Reverse Engineering Options dialog box will be initialized accordingly.

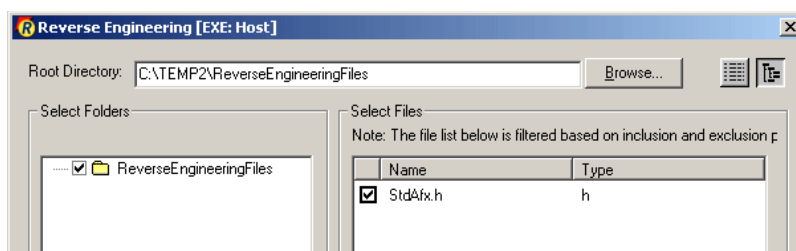
Excluding Particular Files

If you want to exclude particular files or folder from being reverse engineered, you can use the `ReverseEngineering::Main::ExcludeFilesMatching` property. You can assign one or more comma-separated wildcard expressions to this property, as shown in the following figure. The files or folders that are matched using these wildcard expressions will not be reverse engineered.

Note that this property affects only the tree view of the Reverse Engineering dialog box.



With the `ExcludeFilesMatching` property set to `res*`, when you open the Reverse Engineering tool (**Tools > Reverse Engineering**), the Reverse Engineering dialog box (tree view) looks like the following figure, which shows only one file in the **Select Files** box to be reverse engineered. Compare this with the figure in [Initializing the Reverse Engineering Dialog Box](#) that shows the same dialog box but without the `ExcludeFilesMatching` property set.



Reverse Engineering Tool Limitations

Note the following limitations for the Reverse Engineering tool:

- ◆ The only way to change the location of where reverse engineering information will be retrieved and saved is to close the Reverse Engineering dialog box and change the active configuration for the model in Rhapsody.
- ◆ The tree view does not support displaying files selected from multiple drives. In such cases, Rhapsody will not let you switch from tree view to the flat view.

Visualization of External Elements

Rhapsody enables you to visualize legacy code or edit external code as *external elements*. This external code is code that is developed and maintained outside of Rhapsody. This code will not be regenerated by Rhapsody, but will participate in code generation of Rhapsody models that interact or interface with this external code so, for example, the appropriate `#include` statement is generated. This functionality provides easy modeling with code written outside of Rhapsody, and a better understanding of a proven system.

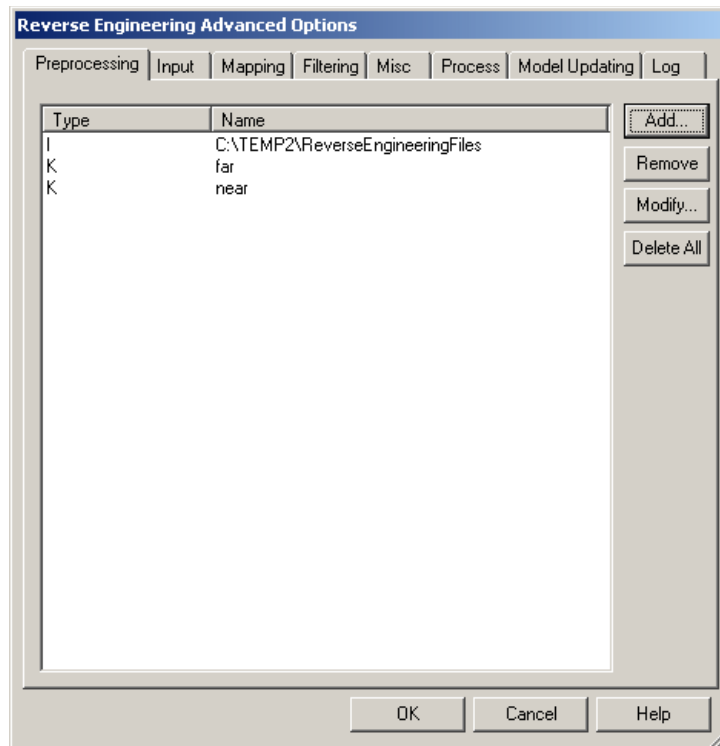
Rhapsody supports the following reverse engineering functionality using external elements:

- ◆ Reverse engineering can import elements as external.
- ◆ Reverse engineering populates the model with enough information to:
 - Model external elements in the model.
 - Enable you to open the source of the external elements, even if the element is not included in the scope of the active component.

For more information, see [External Elements](#).

Defining Preprocessor Symbols

The **Preprocessing** tab, as shown in the following figure, lets you define preprocessor symbols to be uniformly applied to all imported files.



You can define the following types of preprocessor symbols:

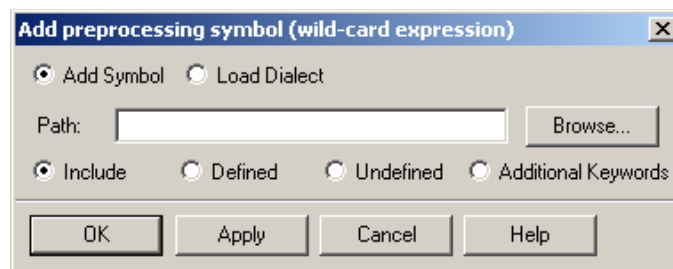
- ◆ **D** means symbol defined with `#define`. See [Defined Symbols \(C and C++\)](#).
- ◆ **I** means directory added to `#include` search path. See [Include/CLASSPATH Paths](#).
- ◆ **K** means additional keywords. See [Additional Keywords \(C and C++\)](#).
- ◆ **U** means symbol undefined with `#undef`. See [Undefined Symbols \(C and C++\)](#).

The **Preprocessing** tab has the following controls:

- ◆ **Add** button lets you add a preprocessing symbol. See [Adding a Preprocessing Symbol](#).
- ◆ **Remove** button, which is enabled only when applicable, lets you remove one or more selected symbols from the list.
- ◆ **Modify** button lets you modify the selected symbol.
- ◆ **Delete All** button lets you delete all preprocessing symbols from the list.

Adding a Preprocessing Symbol

To add a preprocessing symbol, click **Add** on the **Preprocessing** tab of the Reverse Engineering Options dialog box to open the Add Preprocessing Symbol dialog box, as shown in the following figure.

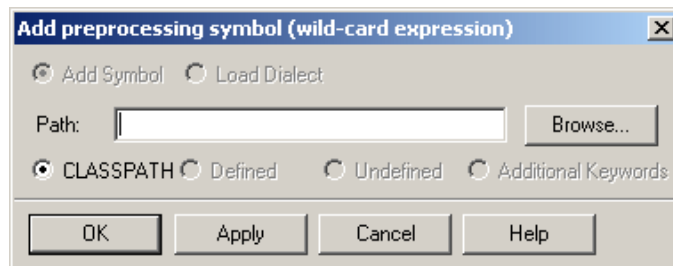


In C and C++, you can add the following symbols:

- ◆ Include symbols. See [Include/CLASSPATH Paths](#).
- ◆ Defined symbols. See [Defined Symbols \(C and C++\)](#).
- ◆ Undefined symbols. See [Undefined Symbols \(C and C++\)](#).
- ◆ Additional keywords. See [Additional Keywords \(C and C++\)](#).

In addition, in C++ you can load dialects. See [Dialects \(C++\)](#).

For Rhapsody in Java, the Add Preprocessing Symbol dialog box enables you to specify only the **CLASSPATH** option, as shown in the following figure.



Include/CLASSPATH Paths

Include paths tell Rhapsody where to look for header files to be included using `#include` directives found in the source file. This directive has the same effect as the following preprocessing switch:

```
/I<dir>
```

The reverse engineering tool adds constructs declared in header files with `#include` statements to the model, as long as either condition is true:

- ◆ The file is in the same directory as the one being imported.
- ◆ An include path is specified on the **Preprocessing** tab.

To include path, follow these steps:

1. In the Add Preprocessing Symbol dialog box, select the **Add Symbol** and **Include/CLASSPATH** radio buttons.
2. Use the **Browse** button to locate the appropriate folder.
3. Depending on what you want to do:
 - ◆ Click **Apply** if you want to enter another Include path.
 - ◆ Click **OK** if you are done and to return to the **Preprocessing** tab. Notice that a symbol with type I is added to the list of preprocessing symbols.
4. Click **OK** again to close the Reverse Engineering Options dialog box.
5. Click **Start** to re-import the file.

Defined Symbols (C and C++)

The `#define` symbol is a constant and macro that the preprocessor expands before compilation. No storage is allocated for these symbols—they have no type, and the debugger cannot reference them. This definition has the same effect as the following preprocessing switch:

```
/D<name>{=|#}<text>
```

- ◆ Solving parser problems with unknown macros or statements
- ◆ For `#ifdef` inclusion or exclusion of code parts

To define symbols, follow these steps:

1. On the Add Preprocessing Symbol dialog box, select the **Add Symbol** and **Defined** radio buttons.
2. In the **Symbol** box, type the name of the symbol and the value, if it has one, using the following format:

```
<symbol> = <value>
```

For example, to define a preprocessing symbol `ev_H` with the value

```
"$Id: event.h 1.22 1999/02/03 11:12:36 amy Exp $", type:
```

```
ev_H = "$Id: event.h 1.22 1999/02/03 11:12:36 amy Exp $"
```

3. Depending on what you want to do:
 - ◆ Click **Apply** if you want to enter another `#define` symbol.
 - ◆ Click **OK** if you are done and to return to the **Preprocessing** tab. Notice that a symbol with type **D** is added to the list of preprocessing symbols.
4. Click **OK** to close the Reverse Engineering Options dialog box.

Using #define

Rhapsody supports #define preprocessor directives that use the following format:

```
#define <identifier> <replacement list>
```

The recommended way to declare a #define declarative in a C model is as follows:

- ◆ If it contains parameters, model it as a function with the `C_CG::Operation::Header` property set to `in_header`. For example:

```
#define MAX(X,Y)  
(X)>(Y)?(X):(Y)
```

- ◆ If it does not contain parameters (for example, it defines a constant), model it as a constant variable. For example:

```
#define SIZE 1024
```

- ◆ Otherwise, model it as a type (for example, multi-line #defines).

The recommended way to declare a #define declarative in a C++ model is as follows:

- ◆ If the #define declarative in C++ does not contain parameters (for example, it defines a constant), model it as a constant variable and set the

`CPP_CG::Attribute::ConstantVariableAsDefine` property to `Checked`. For example:

```
#define SIZE 1024
```

- ◆ Otherwise, model it as a type.

The reverse engineering tool imports the #defines according to the way they are modeled.

However, if the comment for the #define is a multi-line, even though the #define itself is one line, the reverse engineering tool imports it as a type. For example:

```
#define SIZE 1024 /* my buffer  
size */
```

To import all #define as a type, set the

`<lang>_ReverseEngineering::ImplementationTrait::ImportDefineAsType` property to `True`. See the definition provided for the property on the applicable **Properties** tab of the Features dialog box.

Using #if...#ifdef...#else...#endif

The reverse engineering tool reacts to preprocessor conditions (#if...#ifdef...#else...#endif) just as a compiler does. The preprocessor condition structure is not read by the reverse engineering parser, and the only data received is the data inside valid preprocessor conditions.

Consider the following code in a source file:

```
#ifdef _STDC
#define _A
#else
#define _B
#endif
```

- ◆ If `_STDC` is known by the preprocessor, the result of importing this file will be the creation of a user type named `_A` with the following declaration:

```
#define %s
```

- ◆ If `_STDC` is **not** known by the preprocessor, the result of importing this file will be the creation of a user type named `_B` with the following declaration:

```
#define %s
```

Undefined Symbols (C and C++)

The `#undef` preprocessing directive undefines symbols previously defined using the `#define` directive. This has the same effect as the following preprocessing switch:

```
/U<name>
```

To undefine a symbol with `#undef`, follow these steps:

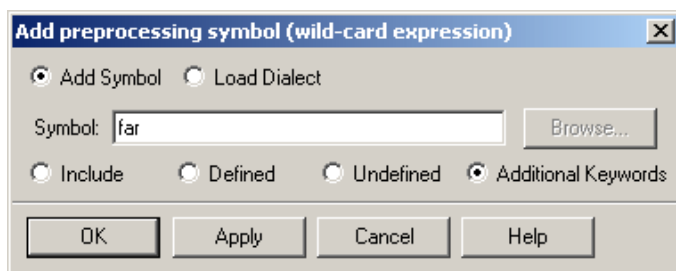
1. In the Add Preprocessing Symbol dialog box, select the **Add Symbol** and **Undefined** radio buttons.
2. In the **Symbol** box, type the name of the symbol you want to undefine.
3. Depending on what you want to do:
 - ◆ Click **Apply** if you want to enter another symbol with `#undef`.
 - ◆ Click **OK** if you are done and to return to the **Preprocessing** tab.
4. Click **OK** to close the Reverse Engineering Options dialog box.

Additional Keywords (C and C++)

To improve keyword support for reverse engineering and roundtripping so that Rhapsody can correctly import and roundtrip declarations that use non-standard or unknown keywords, you can add a list of additional user-defined keywords to the **Preprocessing** tab of the Reverse Engineering Options dialog box.

To add additional keywords, follow these steps:

1. On the Add Preprocessing Symbol dialog box, select the **Add Symbol** and **Additional Keywords** radio buttons.
2. Enter a keyword in the **Symbol** box, as shown in the following figure.



3. Depending on what you want to do:
 - ◆ Click **Apply** if you want to enter more additional keywords.
 - ◆ Click **OK** if you are done and to return to the **Preprocessing** tab.
4. Notice on the **Preprocessing** tab that your keywords with type K are added to the list of preprocessing items.

Note

You can use the `<lang>_ReverseEngineering:Parser:AdditionalKeywords` property to add a list of comma-delimited additional keywords (for example: `far, near`). Note that this property may already have keywords included in it that is provided with Rhapsody.

Additional Keywords Limitations

Note the following limitations for additional keywords:

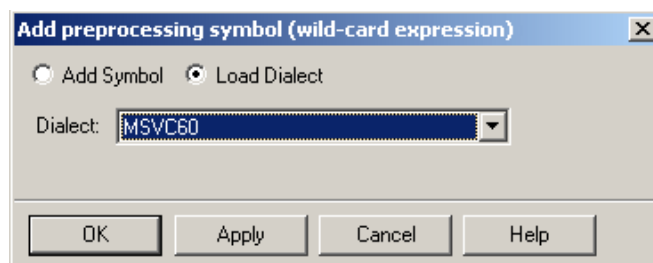
- ◆ Keywords with parameters are not supported.
- ◆ Keywords with more than one word in them are not supported.
- ◆ Keywords cannot be seen in the element's signature.
- ◆ If the same keyword is used in more than one place (for example, before the type and after the type), the parser will encounter an ambiguity and will fail to indicate the keyword use correctly.

Dialects (C++)

Imported source files must contain C++ code that complies with the ANSI/ISO C++ standard. In practice, there are many dialects of C++ in use, some reflecting various stages in the evolution of the language. The dialect setting determines the default dialect. In addition, it can add a list of dialect-specific preprocessing switches to the user-defined switches. The Reverse Engineering tool understands the Microsoft Visual C++ 6.0 (MSVC60) dialect.

To select this dialect in Rhapsody in C++, follow these steps:

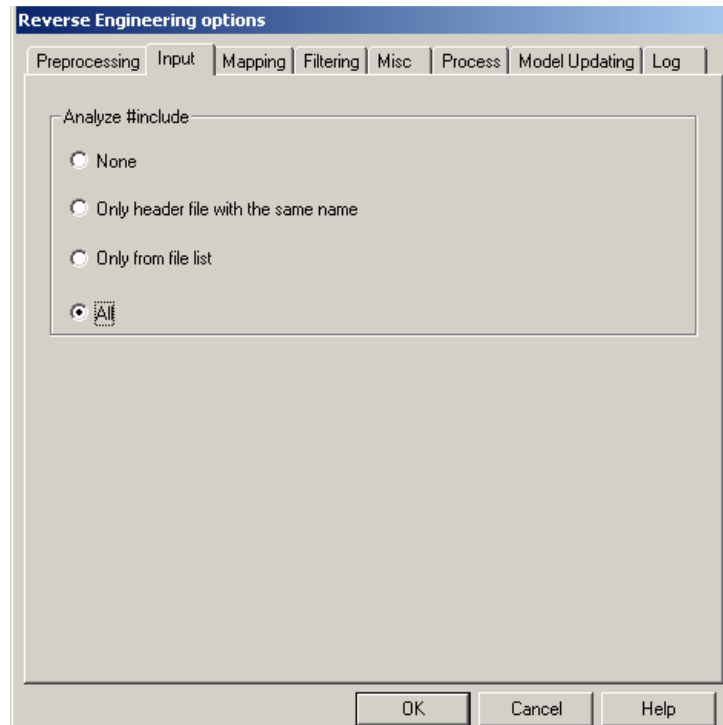
1. In the Add Preprocessing Symbol dialog box, select the **Load Dialect** radio button.
2. From the **Dialect** drop-down list, select **MSVC60**, as shown in the following figure.



3. Click **OK**.

Analyzing #include Files

The **Input** tab, shown in the following figure, lets you specify which include files should be analyzed in the reverse engineering process.



- ◆ **None** means to analyze only the files or folders specified in the main Reverse Engineering dialog box; all #include statements are ignored.

This mode contributes the least performance drain to the reverse engineering process. Reverse engineering in this most limited mode should only be used when appropriate. This mode imports no implementation information (such as operation bodies) or initialization of static variables, and loses needed information such as dependencies. Keep in mind that implementation files cannot be analyzed without first analyzing their corresponding specification files.

Note: Using **None** will not import the content of .cpp or .c files.

- ◆ **Only header file with the same name** means to analyze only the matched included files—in other words, the corresponding specification file for the analyzed implementation file of the same name. This is known as *logical files mode*.

For example, for the implementation file named `MyClass.cpp`, the reverse engineering utility will analyze only the include file named `MyClass.h`. All other included files in the

list of files you selected are not analyzed. This high-performance mode imports full information about analyzed classes, but might lose dependencies through separated parts of a project. This mode is designed for reverse engineering of large projects (about 1000 files).

Note: If you select this option, nested `#include` statements are not analyzed.

See [Analyzing Header Files with the Same Name](#) for an example.

- ◆ **Only from file list** means to analyze only the include files you specified in the main Reverse Engineering dialog box.

For example, suppose you have four files—`one.h`, `two.h`, `three.h` and `one.cpp`—and you select the files named `one.h`, `two.h`, and `one.cpp`. The reverse engineering utility will analyze `one.cpp` and its include files `one.h` and `two.h`. It **will not** analyze `three.h` because you did not select it.

This mode gives you the most control and strategic conservation of performance, eliminating the needless analysis of irrelevant files and redundant information. In addition, it enables you to select files containing important declarations to the analysis without having to add every file within the directory. This mode of reverse engineering imports all the information needed and creates dependencies through whole project. It is designed for middle-sized projects (about 100 files).

See [Analyzing a List of Files](#) for an example.

- ◆ **All** means to analyze all included files on all levels. This mode is called *recursive analysis*, and consumes the most performance because it imports all information, even redundant information such as MFC and STL. This is the default value.

The `<lang>_ReverseEngineering::ImplementationTrait::AnalyzeIncludeFiles` property has enumerated values to indicate how the reverse engineering process analyzes include files. See the definition provided for the property on the applicable **Properties** tab of the Features dialog box.

For C++ projects, you may use the `CreateDependencies` property to specify how the reverse engineering feature should handle the creation of dependency elements in the model from code constructs such as `#includes`, `forward` declarations, `friends`, and `namespace` usage. For more details on how to use this property, see the definition provided for the property on the applicable **Properties** tab of the Features dialog box.

Analyzing Header Files with the Same Name

Suppose you want to reverse engineer the file `omreactive.cpp`. It has the following included files:

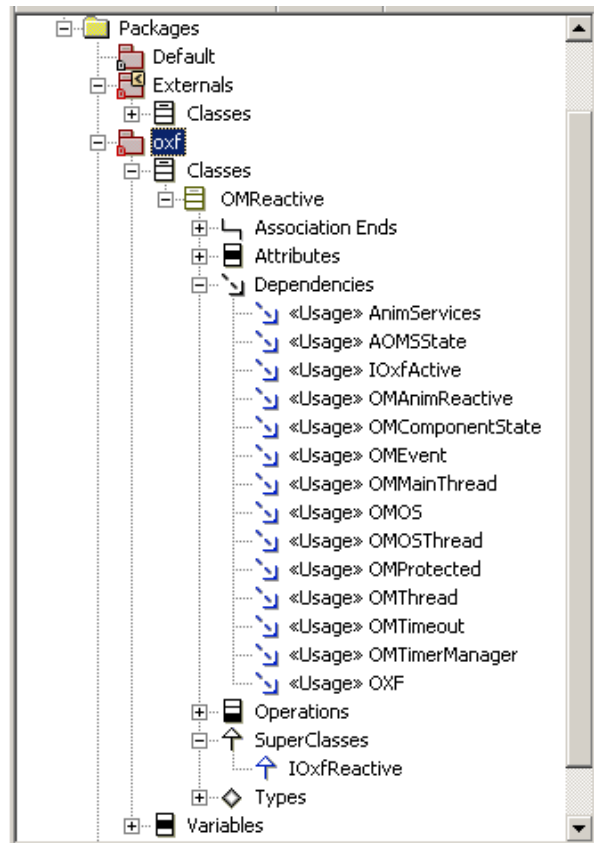
```
#include <oxf.h>
#include <omoutput.h>
#include <omreactive.h>
#include <state.h>
#include <omthread.h>
#include <aommacro.h>
```

To reverse engineer only `omreactive.cpp` and `omreactive.h`, follow these steps:

1. Add the file `omreactive.cpp` to the main Reverse Engineering dialog box.
2. Click **Advanced** to open the Reverse Engineering Options dialog box.
3. On the **Input** tab, select the **Only header file with the same name** radio button.
4. On the **Preprocessing** tab, add the path to the `oxf` folder (for example, `<rhapsody installation path>\Share\LangCpp`). You need to set this value because the directive `#include <omreactive.h>` says to look for the specification file in `omreactive`, so you need to specify where that is.
5. Click **OK** to apply your changes and close the Reverse Engineering Options dialog box.

6. Click **Start** on the Reverse Engineering dialog box.

The tool will analyze `omreactive.h` and ignore the other files included in the `omreactive.cpp` file. As you can see from the following figure, the Rhapsody browser shows the **oxf** package with the **OMReactive** class and some of its data members. The **OMReactive** class has Usage dependencies to externally referenced classes and inherits from the **IOxfReactive** superclass.



Analyzing a List of Files

Suppose you want to reverse engineer the file `omreactive.cpp`. It has the following included files:

```
#include <oxf.h>
#include <omoutput.h>
#include <omreactive.h>
#include <state.h>
#include <omthread.h>
#include <aommacro.h>
```

If you specified all of these files except for `state.h`, all the include files except for `state.h` will be analyzed by the reverse engineering tool.

Mapping Classes to Types and Packages

The **Mapping** tab, as shown in the following figure, enables you to:

- ◆ Create external files in a given package or component.
- ◆ Set the modeling policy based on a file or a class.
- ◆ Allow files to be imported into one package, or to import the files while emulating the existing directory structure.
- ◆ Set standard directories.
- ◆ Create UML dependencies from include statements.



The **Mapping** tab has the following controls:

- ◆ **Visualization Only (Import as External)** check box, when selected, sets Rhapsody to add as external elements all the elements created in Rhapsody during reverse engineering (their `CG::Class/Package/Type::UseAsExternal` property is set to `Checked`). This property is overridden for all the packages (but not their contained elements, such as classes, files, types, unless the contained elements are also packages; and in that case they will also have their `CG::Package::UseAsExternal` property set to `Checked`). In C++, this control has an effect on the controls available in the **Modeling Policy** group.

Note: When you select or clear the **Visualization Only (Import as External)** check box, this same check box is automatically set the same way on the Reverse Engineering dialog box.

- ◆ **Modeling Policy** group lets you save the original file mapping after reverse engineering. This group has the following controls:
 - **File** radio button lets you save files to be imported into separate packages. This is the default value for Rhapsody in C. For C, this means that a model is a file-based (Functional C) model, which is a code-centric model.

This option is only available in C++ when importing the package as “external” (select the **Visualization Only (Import as External)** check box). Importing C++ external files as packages is not advisable if the imported code needs to be reused because code generation for package files is not supported.

This option is not available in Java.

- **Class** radio button means that no files are created.

Note: If you want the mapping file saved to a component (so that you can import files as class-based information and replicates the file structure), set the

```
<lang>_ReverseEngineering::ImplementationTrait::  
RespectCodeLayout property to Mapping.
```

The root path of the top-most folder created during reverse engineering is added to the **Include Path** field of the active component; it is later used to compile `#include` statements for the imported files and to open the source code for external elements. If the imported files have several, non-dependent roots, multiple paths are written to the field, separated by commas.

For example, for files `C:\Project1\Subsystem\A.h` and `D:\Project2\Subsystem2\B.h`, the **Include Path** field would contain the following string:

```
C:\Project1;D:\Project2
```

If this option is used with an empty component (one whose scope is empty or includes only external elements), the first root path is added to the **Directory** field of the component; any additional paths are written to the **Include Path** field.

- ◆ **Map to Package** group lets you specify how to organize imported elements to packages. This group has the following controls:

- **All to** radio button lets you map all imported elements to the package you specify in the text field. For example, you can select the **All to** radio button and the Reverse Engineering tool fills in `ReverseEngineering` in corresponding text box so that all imported elements are mapped to a package called **ReverseEngineering**.

Note: It is possible to set a nested package in this field. The syntax is `Package1::Package2`. Non-existent packages will be created during reverse engineering.

- **Directory is a Package** radio button lets you reverse engineer to create a package for each imported directory. All the files in the directory are imported to the corresponding package. This is the default value.

This option sets the `CG::Configuration::`

`GenerateDirectoryPerModelComponent` property to `Checked` for the active configuration. For elements, the original hierarchy of directories is restored on code generation for the imported elements, and `#include` statements are generated for references to them.

See [Specifying Directory Structures](#) for an example.

- ◆ **Standard Directories** group lets you separate the header and source files to different directories for the reverse engineering of your selected project. This group has the following controls:
 - **Specification** box. Enter the name of the directory for your header files (for example, `inc`).
 - **Implementation** box. Enter the name of the directory for your source files (for example, `src`).

Using the **Specification** and **Implementation** boxes provides you with better modeling of your code if you want to separate the header and source files to different directories. Code generation (after reverse engineering) will also generate each header file to a directory called `inc` and each source file to a directory called `src` (or whatever names you designed in the **Specification** and **Implementation** boxes). The following table illustrates if you use these boxes (left column) versus if you do not (right column).

Specification = <code>inc</code> Implementation = <code>src</code>	Specification = [blank] Implementation = [blank]
client "a elements (from a.h and a.c)" "b elements (from b.h and b.c)" "c elements (from c.h and c.c)" server "e elements (from e.h and e.c)" "f elements (from f.h and f.c)" "g elements (from g.h and g.c)"	client inc "a elements from a.h" "b elements from b.h" "c elements from c.h" src "a elements from a.c" "b elements from b.c" "c elements from c.c" server inc "e elements from e.h" "f elements from f.h" "g elements from g.h" src "e elements from e.c" "f elements from f.c" "g elements from g.c"

Note: You cannot view code using the Active Code View window nor can you edit code if you generate to an external directory.

- ◆ **Dependencies** drop-down list lets you set Rhapsody to create dependencies from the include files during reverse engineering. This means that `#include` between files may create dependencies between the component files and/or between classes. In addition, forward declarations of elements (variables, functions, classes, and so forth) will create dependencies from the component file to the element. The possible values are:
 - **ComponentOnly** means to create dependencies between component files, but not between model classes
 - **None** means do not create dependencies on reverse engineering
 - **PackageAndComponent** means to create dependencies between model classes and component files
 - **PackageOnly** means to create dependencies between model classes, but not between component files
 - **SmartPackageAndComponent** means to create only necessary dependencies to reflect the code

Note: The above values for **Dependencies** are all that are available in the current Rhapsody version. Note the following:

Not all these values will appear for every language of Rhapsody. For example, all of the above values may be available for Rhapsody in C++, but only two may be available for Rhapsody in Java.

You can set the default for **Dependencies** in the `<lang>_ReverseEngineering::ImplementationTrait::CreateDependencies` property. You may notice in the property definition that there is a `DependenciesOnly` value, but it does not appear in the **Dependencies** drop-down list. This value is only for backward compatibility purposes and you cannot set this value directly. In a case where you may have an old model that uses `DependenciesOnly`, the current Rhapsody automatically sets the value to `PackageOnly`.

The list of values and backward compatibility behavior are different among the languages.

This operation is successful if the reverse engineering utility analyzes both the included file and the source—and the source and included files contain class declarations for creating the dependencies between them. If there is not enough information, the includes are not converted into dependencies. This can happen in the following cases:

- The include file was not found, or is not in the scope of the settings on the **Input** tab.
- A class is not defined in the include file or source file, so the dependency could not be created.

If the dependency is not created successfully, the `include` files that were not converted to dependencies are imported to the `<lang>_CG::Class::SpecIncludes` or `<lang>_CG::Class::ImpIncludes` properties so you do not have to re-create them manually. If the include file is in the specification file, the information is imported to the `SpecIncludes` property; if it is in the implementation file, the information is imported to the `ImpIncludes` property.

If a file contains several classes, include information is imported for all the classes in the file.

Specifying Directory Structures

The **Directory is a Package** radio button let you import files into one package, or to import the files while emulating the existing directory structure

For example, instead of using the default value `ReverseEngineering` that the Reverse Engineering tool fills in next to the **All to** control in the **Map to Package** group on the **Mapping** tab of the Reverse Engineering Options dialog box, you could type (without spaces between words), for example, `MyOtherPackage`. If you select the **Directory is a Package** radio button, the file hierarchy is preserved during import, and the tool will create nested packages from the folders.

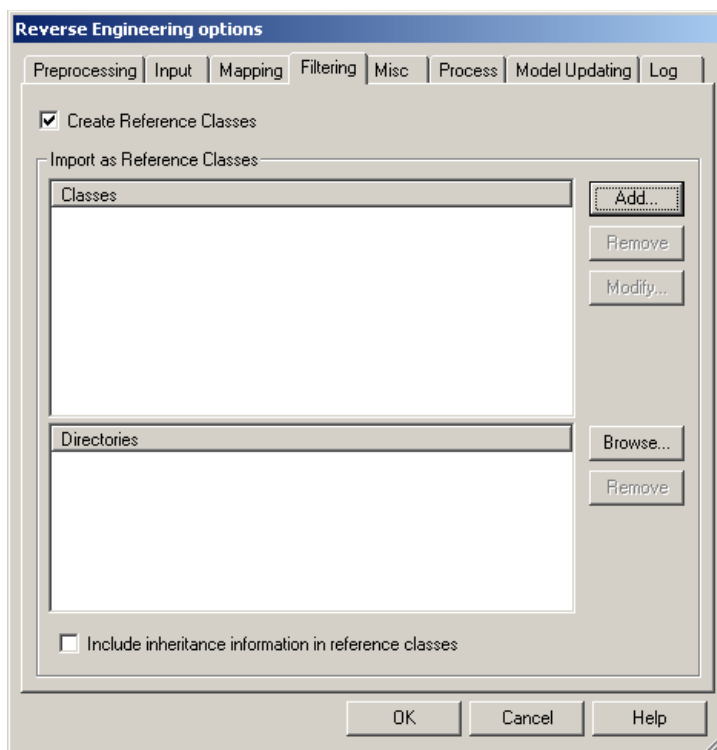
The reverse engineering tool fully supports C++ namespaces. Namespaces are always converted to packages.

For example, the following code will be imported correctly:

```
namespace XXX {
    class CCC {
        int i;
    };
}
```

Specifying Reference Classes

The **Filtering** tab, shown in the following figure, enables you to specify classes that should be imported as reference classes (the equivalent of `CG::Class::UseAsExternal` set to `Checked`). You can select individual classes to model as reference classes, or specify an entire directory of reference classes. Reference classes are imported without attributes and operations—they are used for referencing only.



The tab contains the following controls:

- ◆ **Create Reference Classes** check box lets you specify whether to create external classes for undefined classes that result from forward declarations and inheritance. By default, reference classes are created (as in previous versions of Rhapsody).

If the incomplete class cannot be resolved, the tool deletes the incomplete class if the `<lang>_ReverseEngineering::Filtering::CreateReferenceClasses` property is set to `Cleared`.

Note: In some cases, the class cannot be deleted (for example, a class referenced by a typedef type).

- ◆ **Import as Reference Classes** group has the following controls:

- **Classes** box lists the classes that should be imported as reference classes and their directories. The following buttons control this list:

Add lets you add a reference class.

Remove lets you remove one or more selected reference classes from the list.

Modify lets you modify the specified reference class.

- **Directories** box lists the directories for the imported classes. The following buttons control this list:

Browse button lets you to browse to the correct directory.

Remove lets you remove one or more selected directories from the list.

To analyze these elements during reverse engineering, set the following properties (under `<lang>_ReverseEngineering::Filtering`) to Checked:

- ◆ `AnalyzeGlobalFunctions`
- ◆ `AnalyzeGlobalVariables`
- ◆ `AnalyzeGlobalTypes`

See the definition provided for a property on the applicable **Properties** tab of the Features dialog box.

Reference Classes

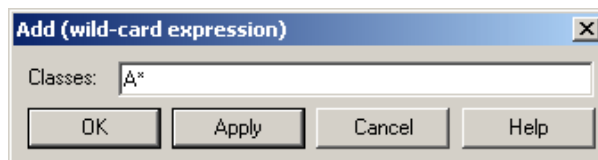
Reference classes are imported into the model as placeholders without members or relations. A typical example is the MFC classes, which are not of interest for elaboration but can be listed simply to show that they are acting as superclasses or peer classes to other classes in the model. Wildcard expressions are permissible for reference class names.

Adding a Reference Class

To add a reference class, follow these steps:

1. In the **Filtering** tab on the Reverse Engineering Options dialog box, click **Add** to open the Add dialog box.
2. Type a wildcard expression in the **Classes** box to match the reference class names, as shown in the following figure. For example, to match all class names that begin with the letter A, enter the following wildcard expression:

A*



3. Click **OK**.

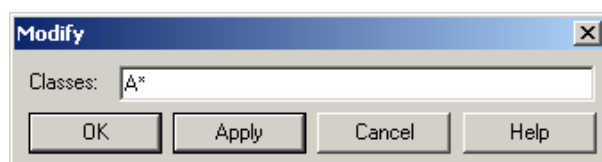
Deleting a Reference Class

To remove a reference class, select the class in the **Classes** list and click **Remove**.

Modifying a Reference Class

To modify a reference class, follow these steps:

1. In the **Classes** list, select the reference class and click **Modify**. The Modify dialog box opens, as shown in the following figure.



2. Modify the wildcard expression.
3. Click **OK**.

Locating a Directory that Contains Reference Classes

The **Directories** list under the **Classes** list on the **Filtering** tab specifies the directories that the reverse engineering tool should search for reference classes, including their subdirectories.

To locate a directory that contains reference classes, follow these steps:

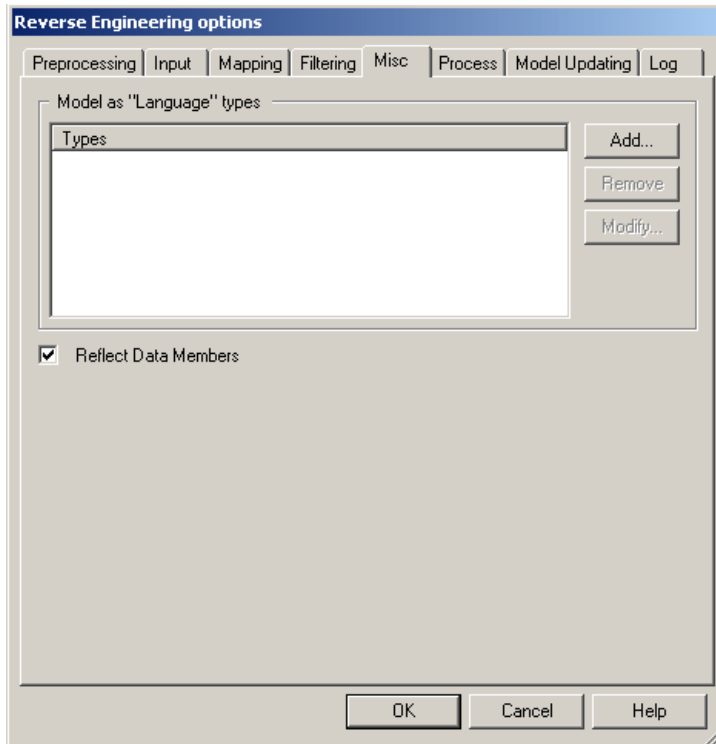
1. On the **Filtering** tab, click **Browse** to open the Browse for Folder dialog box.
2. Select the appropriate directory and click **OK**.

To remove a directory, select the reference class directory and click **Remove**.

Miscellaneous Options

The **Misc** tab, shown in the following figure, enables you to:

- ◆ Import specific classes as Rhapsody types
- ◆ Reflect data members



The **Misc** tab has the following controls:

- ◆ **Model as “Language” types** box lets you specify which classes should be modeled as types. This box has the following controls:
 - **Add** button lets you add a class to be modeled as a type.
 - **Remove** button, which is enabled when applicable, lets you remove the selected type from the list.
 - **Modify** button, which is enabled when applicable, lets you modify the selected type.

See [Modeling Classes as Rhapsody Types](#).

- ◆ **Reflect Data Members** means if this check box is not selected (the check box is cleared), the access level of data members is imported to the `Visibility` property for attributes and the `DataMemberVisibility` property for relations. The visibility in the Features dialog box is always **Public**. This is the behavior of previous versions of Rhapsody.

If this check box is selected, the access level of data members is displayed in the Features dialog box. The `Visibility` property is always set to `fromAttribute` (as `VisibilityOnly`). For relations, the access level is imported to the `DataMemberVisibility` property. In addition, generation of helper functions is disabled on the class properties level.

By default, this checked box is selected.

The following table lists the property values that will be set if the **Reflect Data Members** check box is selected.

Subject and Metaclass	Property	Value
<i>For attributes</i>		
<lang>_CG::Attribute	AccessorGenerate	Cleared
	MutatorGenerate	Never
<i>For relations</i>		
<lang>_CG::Relation	GetAtGenerate	Cleared
	GetKeyGenerate	Cleared
	RemoveKeyGenerate	Cleared
CG::Relation	AddComponentHelpersGenerate	Cleared
	AddGenerate	Cleared
	AddHelpersGenerate	False
	ClearGenerate	Cleared
	ClearHelpersGenerate	Cleared
	CreateComponentGenerate	Cleared
	DeleteComponentGenerate	Cleared
	FindGenerate	Cleared
	GetEndGenerate	Cleared
	GetGenerate	Cleared
	RemoveComponentHelpersGenerate	Cleared
	RemoveGenerate	Cleared
	RemoveHelpersGenerate	False
	SetComponentHelpersGenerate	Cleared
	SetGenerate	Cleared
	SetHelpersGenerate	From Modifier
<i>For classes</i>		
CG::Class	InitCleanUpRelations	Cleared

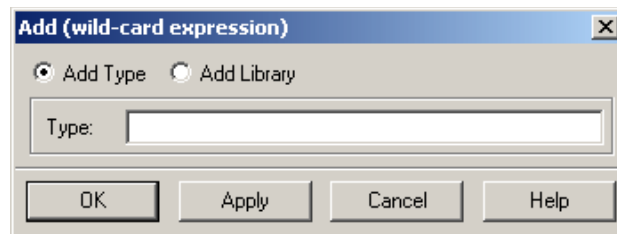
Note: The setting of this check box is equivalent to the <lang>_ReverseEngineering::ImplementationTrait::ReflectDataMembers property. See the definition provided for the property on the applicable **Properties** tab of the Features dialog box.

See [Reflect Data Members](#) for an example.

Modeling Classes as Rhapsody Types

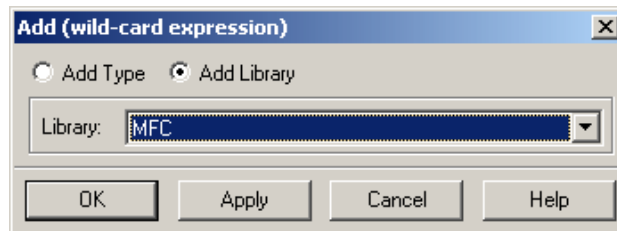
You can tell Rhapsody either to model certain classes as types or to use the MFC type library.

1. On the **Misc** tab of the Reverse Engineering Options dialog box, click **Add** to open the Add (wild card expression) dialog box for types, as shown in the following figure.



2. Select the appropriate radio button:
 - ◆ **Add Type** to add the class as a type
 - ◆ **Add Library** to add the class as an MFC type definition
3. If you selected the **Add Type** radio button, type the name of a single class or a wildcard expression to match the names of several classes to be imported in the **Type** box. For example, `OM*` specifies that all classes that start with “OM” should be types.

Or, if you selected the **Add Library** radio button (applicable to Rhapsody in C++ only), select **MFC** from the **Library** drop-down list, as shown in the following figure.



Note: Currently, Rhapsody in C++ recognizes only one predefined type library (MFC) with only one class (CString). You can add more classes in the `CPP_ReverseEngineering::MFC::DataTypes` property of the current configuration. Alternatively, you can create new library metaclasses, like MFC, in the `factory.prp` and `site.prp` files.

Modeling Typedefs as User-Defined Types

Typedefs are read in as user-defined types under the corresponding package or class in the model.

Example 1

Consider a source file that contains the following typedef:

```
typedef unsigned char CHAR;
```

The resultant type will have the name CHAR and have the following form:

```
typedef unsigned char %s
```

Example 2

Consider a source file that contains the following enumerated type:

```
typedef enum {GOOD, INVALID, SWAPPED}image_file_status;
```

The resultant type will have the name image_file_status and have the following form:

```
typedef enum {GOOD, INVALID, SWAPPED} %s
```

Modeling Structures as Types Instead of Classes

To have Rhapsody model structures as types instead of classes, click the **Add** button on the **Misc** tab of the Reverse Engineering Options dialog box so that you can enter the names of the structures or use a wildcard to apply the mapping to all structures.

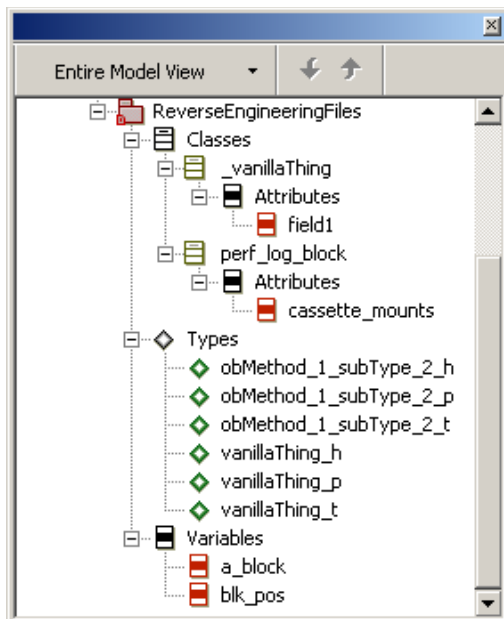
For example, consider the following source file:

```
struct perf_log_block
{
    int cassette_mounts;
};
struct perf_log_block blk_pos[ FIVE ];
another_block a_block[ FIVE ];

typedef struct _vanillaThing
{
    char    field1;
} vanillaThing_t, *vanillaThing_p, **vanillaThing_h;

typedef struct
{
    int    field4;
} obMethod_1_subType_2_t, *obMethod_1_subType_2_p,
**obMethod_1_subType_2_h;
```

If you do not specify anything on the **Misc** tab, the structures are not modeled as types, as shown in the following figure.



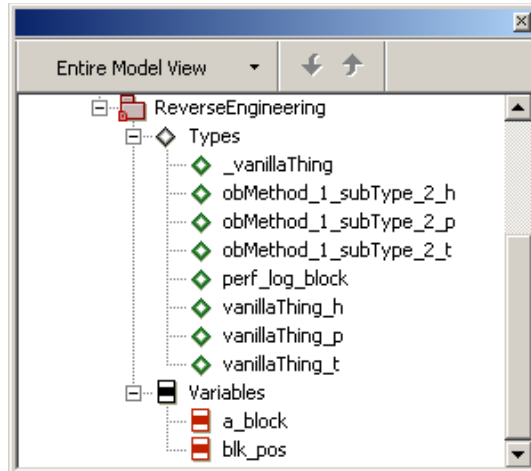
Note the following:

- ◆ The first structure type is modeled as a class named `perf_log_block` with an attribute named `cassette_mounts`.
- ◆ The array of structures is modeled as an instance of type `perf_log_block` with a multiplicity of FIVE.
- ◆ The array is modeled as a variable named `a_block`, with the following form:

```
another_block %s[ FIVE ]
```
- ◆ The structure typedef is modeled as a class named `_vanillaThing` with an attribute named `field1`.
- ◆ The types `vanillaThing_t`, `*vanillaThing_p`, and `**vanillaThing_h` are modeled as types, as follows:

```
- typedef struct _vanillaThing %s
- typedef struct _vanillaThing * %s
- typedef struct _vanillaThing * * %s
```
- ◆ The types `obMethod_1_subType_2_t`, `*obMethod_1_subType_2_p`, `**obMethod_1_subType_2_h` are modeled types.

If you add the wildcard symbol (*) to the **Types** list so all structures are mapped to types, the results are as follows:



Note the following:

- ◆ The first structure type is modeled as a type named `perf_log_block` with the following declaration:


```
struct %s
{
    int cassette_mounts;
};
```
- ◆ The array of structures is modeled as a variable named `blk_pos`, of type `perf_log_block` with a multiplicity of FIVE, as follows:


```
perf_log_block %s[ FIVE]
```
- ◆ The array is modeled as a variable named `a_block`. Its declaration is as follows:


```
another_block %s[ FIVE ]
```
- ◆ The structure typedef is modeled as a type named `_vanillaThing`. Its declaration is as follows:


```
struct %s
{
    char field1;
};
```
- ◆ The types `vanillaThing_t`, `*vanillaThing_p`, and `**vanillaThing_h` are modeled as types, as follows:


```
- typedef struct _vanillaThing %s
- typedef struct _vanillaThing * %s
- typedef struct _vanillaThing * * %s
```

Reflect Data Members

When the **Reflect Data Members** check box is selected on the **Misc** tab of the Reverse Engineering Options dialog box, the Reverse Engineering tool imports all code data members as private. The access level of data members in the code is imported into the `visibility` property of attributes.

When this option is not selected:

- ◆ Reverse engineering imports code data members as attributes with public visibility—all attributes are listed as **Public** in the Features dialog box. In generated code, they have the correct visibility.
- ◆ Accessors and mutators are generated, as are the original, user operations.

When this option is selected:

- ◆ Attributes in the features dialog have the “real” visibility, matching the imported code.
- ◆ Accessors and mutators are not generated.

For example, consider the following file, `clock.h`:

```
#ifndef CLOCK_H
#define CLOCK_H

#include <stdio.h>
class clock
{
    int second;
    int minute;

    public:
        clock();
        void incTime(void);
    protected:
        int present_second(void) {return second;}
        int present_minute(void) {return minute;}
};

#endif
```


The file `clock.cpp` contains the following code:

```

clock.cpp
#include "clock.h"

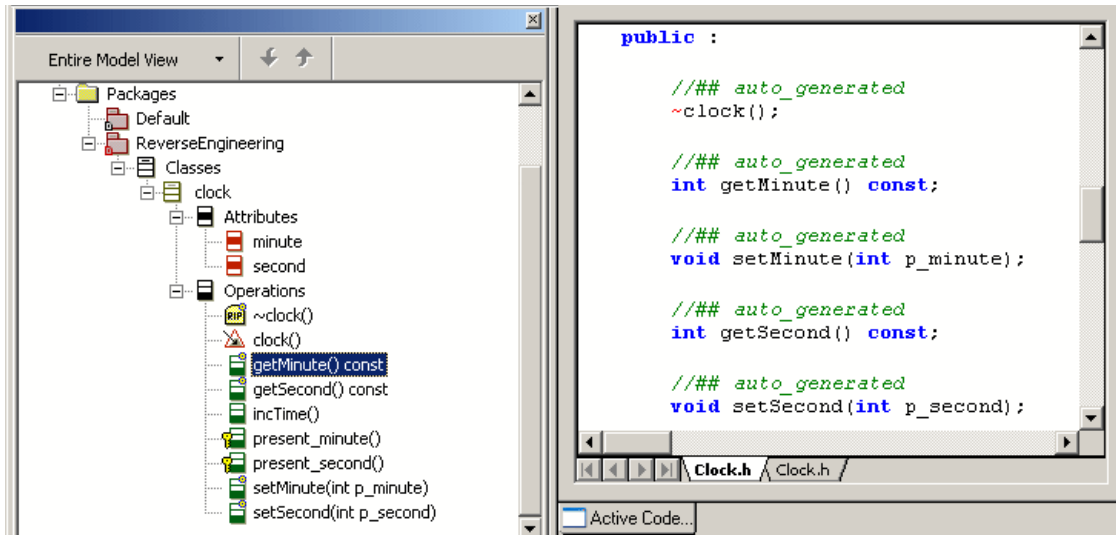
clock::clock() : minute(0),second(0)
{
}

void clock::incTime(void)
{
    if (second == 59)
    {
        second = 0;
        minute ++;
    }
    else
    {
        second++;
    }
    cout << minute << ":" << second << endl;
}

```

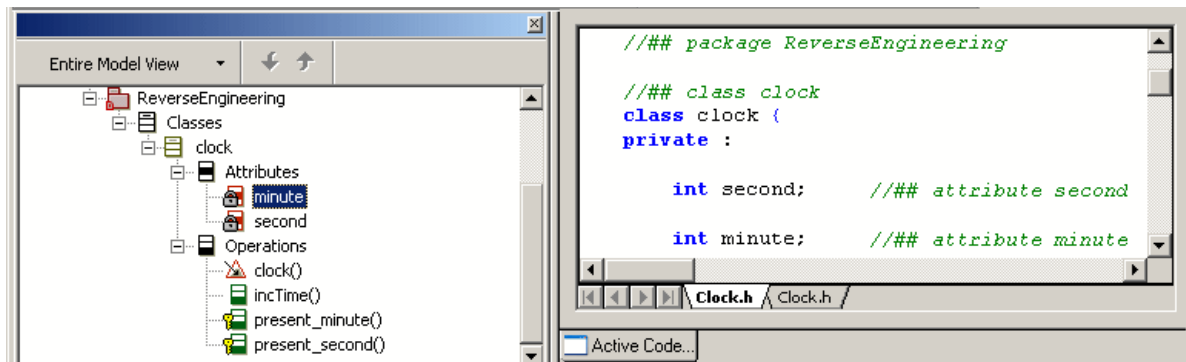
If you reverse engineer these files with the **Reflect Data Members** check box cleared (the equivalent of setting the

`<lang>_ReverseEngineering::ImplementationTrait::ReflectDataMembers` property to None) and the input option **Only from file list** on the **Input** tab of the Reverse Engineering Options dialog box, the results are as shown in the following figure.



Note that the accessors and mutators are shown as public in the browser, but the actual visibility of the attributes is private.

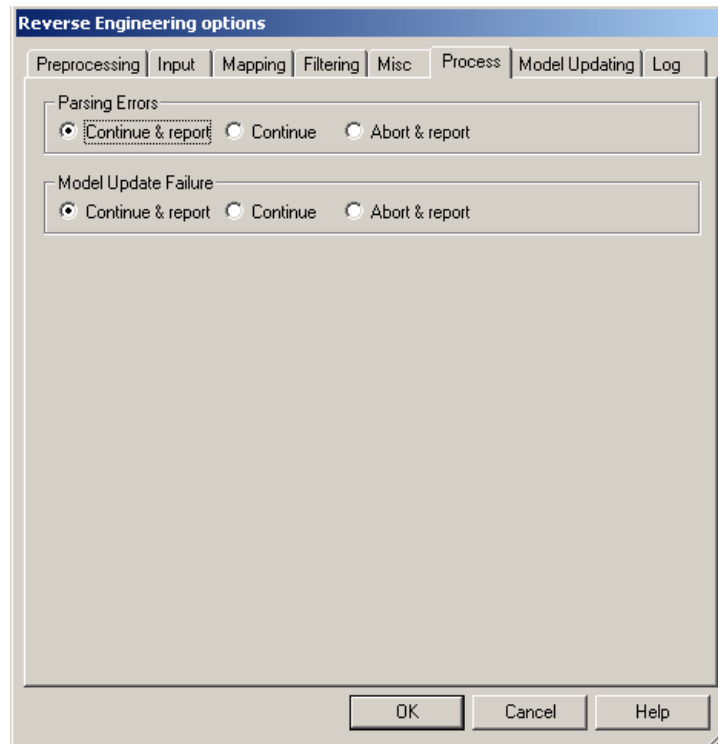
If you select the **Reflect Data Members** check box and repeat the reverse engineering process, the attributes are private and the accessors and mutators are not generated, as shown in the following figure.



In this case, legacy code that already has these operations uses them instead of the Rhapsody default ones.

Error Handling

The **Process** tab, shown in the following figure, enables you to specify what to do when the reverse engineering process encounters certain error conditions.



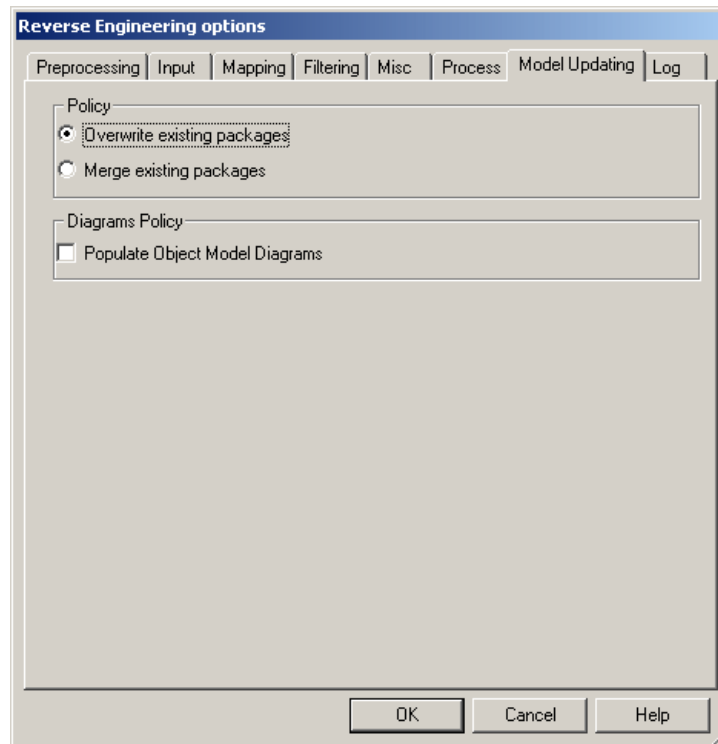
- ◆ **Parsing Errors** are errors encountered while parsing of the source file.
- ◆ **Model Update Failure** are failures that occur when Rhapsody cannot update a model.

For each error condition, there are possible actions:

- ◆ **Continue & report** means to continue importing the next construct and report the error condition.
- ◆ **Continue** means to continue importing but do not report the condition.
- ◆ **Abort & report** means to stop importing and report the condition.

Updating Existing Packages

The **Model Updating** tab, shown in the following figure, enables you to specify how to update existing packages with imported constructs.



The tab contains the following controls:

- ◆ **Overwrite existing packages** means to replace existing packages (including diagrams) with imported ones. For example, if an imported package contains one class and the existing package contains three different classes, the single class being imported replaces the three existing classes.
- ◆ **Merge existing packages** means to merge imported constructs into existing packages. All existing diagrams will be overridden and new diagrams will be created according to the imported packages. For example, if an imported package has a class that contains different attributes than the same class in the existing package, the two classes are merged. The existing package also retains any other classes it had prior to the import.

- ◆ **Populate Object Model Diagrams** means to create object model diagrams (if there are any) when importing. Note that you use **Populate Object Model Diagrams** in conjunction with **Overwrite existing packages** and **Merge existing packages**. Note also that when you select or clear the **Populate Object Model Diagrams** check box, this same check box is automatically set the same way on the Reverse Engineering dialog box.

Note: Your selection is stored in the `ReverseEngineering::Update::CreateDiagramsAfterRE` property.

Command-Line Interface for Populate Object Model Diagrams

If you have set the `ReverseEngineering::Update::CreateDiagramsAfterRE` property to `Checked`, when you run reverse engineering through the Rhapsody command-line interface, any object model diagrams being imported will be created in your Rhapsody model.

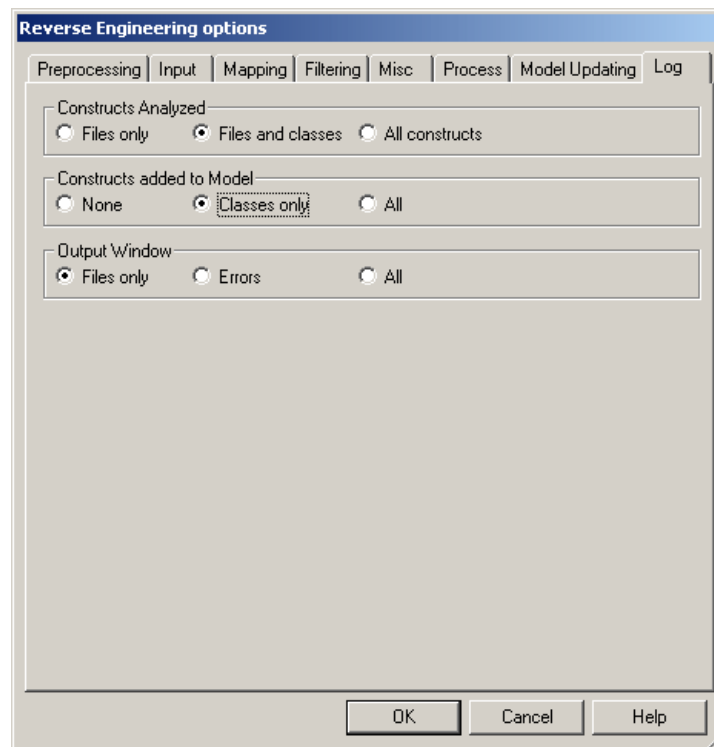
Populate Object Model Diagrams Limitations

Note the following limitations:

- ◆ You cannot merge an existing diagram with a new one.
- ◆ Diagrams can include a maximum of 256 elements.

Message Reporting

The **Log** tab, shown in the following figure, enables you to specify which kinds of constructs the reverse engineering utility should report on during the import. These options directly impact the performance of the reverse engineering process—the more options you select, the slower the process.



The tab contains the controls:

- ◆ **Constructs Analyzed** specifies which constructs to report. The possible choices are as follows:
 - **Files only** means to report only the files being analyzed.
 - **Files and classes** means to report only the files and classes being analyzed.
 - **All constructs** means to report all constructs being analyzed.
- ◆ **Constructs added to Model** specifies which constructs added to the model are reported. The possible choices are as follows:
 - **None** means no constructs.
 - **Classes only** means to report only the classes being added.
 - **All** means to report all the constructs being added.

- ◆ **Output Window** specifies which output to display in the Output window. The possible choices are as follows:
 - **Files only** means to display file information only.
 - **Errors** means to display errors only.
 - **All** means to display all information.

To improve performance when you are reverse engineering large amounts of legacy code, you can hide the Output window. To do this, set the following environment variable in your `rhapsody.ini` file:

```
NO_OUTPUT_WINDOW=TRUE
```

Code Respect and Reverse Engineering for Rhapsody in C and C++

For Rhapsody in C and C++ you can reverse engineer code into the Rhapsody model in a manner that “respects” the structure of the code and preserves this structure when code is regenerated from the Rhapsody model. Meaning that code generated in Rhapsody resembles the original. This means you have complete flexibility for using manually written code or auto-generated code while receiving all the benefits of modeling. You can reverse engineer code into a model in a manner that the model respects the order, location, and dependencies of the global elements in the original code.

For more details about code respect and on how to activate it, see [Code Respect](#).

Reverse Engineering for C++

You can reverse engineer C++ templates.

Reverse Engineering for Rhapsody in Java

There is JDK 1.5 support for generics, enumerations, and type-safe containers.

For more specific information about reverse engineering for Rhapsody in Java, see the following topics:

- ◆ [Reverse Engineering and Java 5 Annotations](#)
- ◆ [Javadoc Handling in Reverse Engineering and Roundtripping](#)
- ◆ [Reverse Engineering / Roundtripping and Static Import Statements](#)
- ◆ [Reverse Engineering / Roundtripping and Static Blocks](#)

Reverse Engineering Other Constructs

This section describes how to reverse engineer these constructs: [Unions](#), [Enumerated Types](#), and [Comments](#)

Unions

Unions are read in as user types under the corresponding package in the model.

Consider the following source file:

```
union bb32
{
    int x;
    char y;
}
```

The resultant type will be named `bb32` and have the following declaration:

```
union %s
{
    int x;
    char y;
};
```

Enumerated Types

Enumerated types are read in as user types under the corresponding package in the model.

Consider the following enum:

```
enum cc_buffer_modes
{
    WRITE_MODE,
    READ_MODE_FORWARD,
    READ_MODE_BACKWARD
};
```

The resultant type will be named `cc_buffer_modes` and have the following declaration:

```
enum %s
{
    WRITE_MODE,
    READ_MODE_FORWARD,
    READ_MODE_BACKWARD
};
```

Comments

During reverse engineering, a comment that comes immediately before the code for an element is considered a comment for that element, and the comment text will be brought into Rhapsody as the description for that element.

You can use the `<lang>_ReverseEngineering::ImplementationTrait::PreCommentSensibility` property to specify the maximum number of lines by which a comment can precede the code for an element and still be considered a comment for that element. Any comment that precedes an element by more than the number of lines specified will be considered a floating comment. For example, a value of 1 means that a comment must appear on the line prior to the code for an element to be considered a comment for that element. The default is 2.

If a C or C++ project has been reverse engineered, the comments are imported as text elements in the relevant SourceArtifacts and are read in as whole blocks. (The comments that are not become a description of some element that is imported.) Then when the code is generated or roundtripped, the comment/text element is placed in its correct place based on its original location.

The following properties are set by default for this feature:

- ◆ `<lang>_ReverseEngineering::ImplementationTrait::RespectCodeLayout` property is set to `Ordering`.
- ◆ `<lang>_CG::Configuration:CodeGeneratorTool` property is set to `Advanced`.
- ◆ `<lang>_Roundtrip::General::RoundtripScheme` property is set to `Respect`.

For information about respect and SourceArtifacts, see [Code Respect](#).

Note the following:

- ◆ If a function has one comment in a .h file and another comment in a .cpp file then the comment in the .cpp file is imported as a floating comment.
- ◆ When any reversed engineered file has a comment as its first element, then any file header comment is turned off. The same is true for any file footer comment.
- ◆ Reverse engineering imports the first/last comment of the file as a regular comment (as a text fragment). Reverse engineering disables generation of the auto-generated header/footer by setting these properties to empty string values.
 - `<lang>_CG::File::ImplementationHeader`
 - `<lang>_CG::File::SpecificationHeader`
 - `<lang>_CG::File::ImplementationFooter`
 - `<lang>_CG::File::SpecificationFooter`

Limitations for Comments

Note the following limitations for comments:

- ◆ The following situation may cause comments to not get imported from open `#IFDEF` branches: When an `.h` file is processed more than once and the `#IFDEF` is at one time “open” and at another time “closed,” some comments inside the `#IFDEF` may be lost.
- ◆ Some comments, usually inside an element declaration (like comments on arguments), change their place after code generation. They are generated below the element.

Macro Collection

Note

The macro collection feature applies to C and C++.

Macro collecting allows Rhapsody to automatically understand macros in code that will be reverse engineered. This enhances the process for re-using legacy C and C++ code within Rhapsody, providing an easier adoption of *Model-driven Development (MDD)* while enabling a more code-centric workflow.

During reverse engineering, Rhapsody imports “include” files according to the options selected on the **Input** tab of the Reverse Engineering Options dialog box. “Include” files that do not satisfy the specified criteria are not imported into the model.

This can lead to problems if there are files that use macros from “include” files that not will not be imported into the model according to the reverse engineering options selected. To prevent any such problems, Rhapsody goes through all “include” files and collects any macros defined in them.

Note

During reverse engineering, macro collection takes place before import of the files so the macros are taken into account when the model is built.

Collected Macro File

The collected macros become part of the model. They are stored in a controlled file called `CollectedMacros.h` which appears in the browser under the configuration used.

Within this controlled file, macros are grouped by their file of origin.

Macros can be modified in or deleted from this file.

Code Generation

Generated code is similar to the original code.

- ◆ Imported elements are generated into the original files.
- ◆ Order of elements is preserved.

When code is generated from the model, the content of the collected macros will be reflected in the generated code (meaning the generated code will not contain references to the macros).

Controlling Macro Collection

The way that Rhapsody collects these macros can be controlled using the `<lang>_ReverseEngineering::ImplementationTrait::CollectMode` property, which is set at the Configuration level.

This property can take the following values:

- ◆ **None** means that macros will not be collected from include files that are not on the reverse engineering list. This is the default.
- ◆ **Once** means that macros will be collected only if the model does not yet include a controlled file of collected macros.
- ◆ **Always** means that macros will be collected each time reverse engineering is carried out. The controlled file that stores the macros will be replaced each time.

Code Generation of Imported Macros

Note

This feature applies to C and C++.

Rhapsody (through reverse engineering) imports macros unexpanded so that imported macros can behave as calls to macros by default. This means that in subsequent code generation a macro will be generated as is.

The following properties are set by default for this feature:

- ◆ `<lang>_ReverseEngineering::ImplementationTrait::RespectCodeLayout` property is set to `Ordering`.
- ◆ `<lang>_ReverseEngineering::ImplementationTrait::MacroExpansion` property is set to `Cleared`.
- ◆ `<lang>_Roundtrip::General::RoundtripScheme` property is set to `Respect`.

Note that the contents of a macro are not be shown in the model (meaning you will not see its contents in the Rhapsody browser).

For information about respect, see [Code Respect](#).

Limitations

Note the following limitations:

- ◆ The `#define` of a macro needs to be known (as if the file was being compiled).
- ◆ The macro call must occupy a line by itself, meaning that it does not have any other text before or after it, as shown in the following example:

```
// a macro to declare a list of int's  
DECLARE_LIST_OF_TYPE(listName, int)
```

If there is text before or after a macro, as shown in the following example, the macro call will be expanded, meaning that it will not be imported as a macro call:

```
DECLARE_LIST_OF_TYPE(listName, int) // a macro to declare a list of int's
```

Backward Compatibility Issues

When you open a model created before Rhapsody 7.2, Rhapsody, by default, does not import a macro as a call to the macro itself. Instead Rhapsody imports the expanded elements of a defined macro.

Results of Reverse Engineering

The results of reverse engineering are as follows:

- ◆ Recognized and supported constructs are added to the model.
- ◆ Existing features in a model are updated from the source file to match the source file definition. For example, if the type of an attribute differs in an existing model and a source file being imported, it is changed in the model to match the source file.
- ◆ With the code respect ability, the reverse engineered code in the Rhapsody model respects the structure of the original code and preserved this structure when code is regenerated from the Rhapsody model. The reverse engineered C++ code respects the order, location, and dependencies of the global elements in the original code.
- ◆ Macro collecting allows Rhapsody to automatically understand macros in code that will be reverse engineered.
- ◆ Unresolved elements that are not resolved by the import process remain unresolved.
- ◆ New diagrams or statecharts are not synthesized using imported elements.
- ◆ New model elements found in a source file are added to the browser, but not to existing diagrams.
- ◆ If you selected the **Overwrite existing packages** option on the **Model Updating** tab of the Reverse Engineering Options dialog box existing model elements not found in the file being imported are deleted from the model.

Lost Constructs

Some design information might be lost during import if it cannot be represented internally by Rhapsody. Rhapsody can approximate some information, such as non-public inheritance, in which case the construct can be saved. However, if approximation is turned off for a particular construct or if Rhapsody cannot approximate it, the construct will be lost. Subsequent code generation may cause compilation errors.

The following table lists the constructs that are lost on import.

C++ Construct	Description
Anonymous types with members	Enum, class.
Unions	Mapped to an uninterpreted type rather than a special kind of class.
Namespaces	Will be lost if they are not mapped to packages.
Anonymous types with no instances	
Comments that cannot be mapped to code constructs	The last comment, where comments are specified as above the construct; the first comment, where comments are specified as below the construct.
Vendor-specific language extensions	MS DevStudio's PASCAL.
Qualifiers	<code>const</code> is shown in the browser as a C++ declaration (volatile).
Storage classes	Auto, register, static, extern, mutable.
Function specifiers	Inline definitions that are part of a function declaration are marked as such, but definitions that are separate from the declaration (even within the same file) are not explicit.
Ellipses in function declarations	

Roundtripping

Roundtripping is an on-the-fly method used to update the model quickly with small changes entered to previously generated UML code. You can activate a roundtrip in batch mode by updating the code in the file system and then explicitly synchronizing the model. You can also activate a roundtrip in an online, on-the-fly, mode by changing a model within one of the Rhapsody designated views. However, roundtripping should not be used for major changes in the model that would require the model to be rebuilt.

With Rhapsody in C and C++ you can roundtrip code into the Rhapsody model in a manner that “respects” the structure of the code and preserves this structure when code is roundtripped in the Rhapsody model. This means the order of elements in the original code can be preserved during code generation and you can freely change the order of class members and globals and Rhapsody respects the change.

When you have changed the order of elements in C and C++, roundtripping in “respect” mode preserves the order of the following elements for the next code generation:

- ◆ Global elements
- ◆ Class elements
- ◆ #includes and forward declarations
- ◆ Auto-generated operations (excluding statechart and instrumentation code)

For more details about code respect and on how to activate it, see [Code Respect](#).

Note

Rhapsody has properties that restrict or control how roundtripping changes the model. See [Roundtripping Properties](#).

Supported Elements

In general, you can roundtrip the following model elements:

- ◆ Classes and class aggregations (template class, types, nested classes)
- ◆ Class elements (attributes, bodies of primitive operations, arguments, types)
- ◆ Class member functions (name, return type, arguments)
- ◆ Operations (comments, name, arguments type/name, arguments addition/removal, return type, visibility)
- ◆ Global elements (functions, variables, types, template functions)
- ◆ Relations (comments, name, other class association, multiplicity, visibility, static-property)
- ◆ Events (comments, name, default argument value, arguments type, arguments addition/removal, return type)
- ◆ Actions placed on transition labels
- ◆ Actions placed inside a state: on entry, on exit, or as a static reaction.
- ◆ Code within annotations
- ◆ `#define-s`.

Roundtripping Limitations

Roundtripping is not supported for changes that are made to the following nor to data that are not supported by Rhapsody modifiers (for example, mutable or volatile):

- ◆ Stereotypes
- ◆ States
- ◆ Transitions
- ◆ Precompiled directives (`#pragma`, macros)
- ◆ File's header/footer
- ◆ File's text element, not supported by Rhapsody modifiers
- ◆ Component / Configuration information (file mapping)
- ◆ Inline operations generated as `#define`

Dynamic Model-code Associativity (DMCA)

Dynamic model-code associativity (DMCA) changes the code of a model to correspond to the changes you make to a model in the Rhapsody UI. Conversely, when you edit the code of a model directly, DMCA redraws the model to correspond to your edits. In this way, Rhapsody maintains a tight relationship and traceability between the model and the code; the code represents another view of the model. The following **Code > Dynamic Model Code Associativity** menu commands control the DMCA in the model:

- ◆ **Bidirectional** changes are automatically put through in both directions. That is, changes that are made to the model cause new, updated code to be generated, and edits made directly to the code are automatically added to the Rhapsody model.
- ◆ **Roundtrip** changes made directly to the code are brought into the Rhapsody model, but changes made to the model do not automatically generate new code. You can roundtrip only code that has been previously generated by Rhapsody; that is, you can only edit code that has been previously generated by Rhapsody and incorporate those changes back to the model. See [The Roundtripping Process](#) for more information.
- ◆ **Code Generation** changes made to the model automatically generate new code, but editing code does not automatically change the model.
- ◆ **None** means DMCA is disabled. The online code view windows become simple text editors

You may also use the following two settings to control DMCA:

- ◆ `ModelCodeAssociativityMode` in the `rhapsody.ini` file
- ◆ The `General::Model::ModelCodeAssociativityFineTune` property allows you to change the default DMCA mode. However, this is usually set through the menu commands listed previously. (Default = `Bidirectional`)

Note

Dynamic model-code associativity is applicable to all versions of Rhapsody (meaning, Rhapsody in C, C++, Java, and Ada).

The Roundtripping Process

If you modify source files directly, then select **Code > Generate Code**, Rhapsody prompts you to roundtrip the code. This section specifies your options within the roundtripping procedure.

Automatic and Forced Roundtripping

If you select automatic roundtripping (the **Bidirectional** or **Roundtrip** setting), the model is automatically updated with code changes if one of the following occurs:

- ◆ You change the window in focus, away from the Code View window.
- ◆ You save the file that you are editing.
- ◆ You close the Code View window.

To force roundtripping, do one of the following:

- ◆ Select **Code > Roundtrip**. Code is roundtripped for modified elements.
- ◆ Select **Code > Force Roundtrip**. Code is roundtripped for all elements.

You can force roundtripping at any time. If you have set DMCA to **None** or **Code Generation**, the only way to roundtrip modified code back into the model is by a forced roundtrip.

Roundtripping Classes

1. In an OMD, left-click the classes.
2. Select **Code > Roundtrip > Selected classes**, or right-click the classes and select **Roundtrip** from the pop-up menu.

Rhapsody responds by listing messages of the form “Roundtripping class x” for every class that is roundtripped. Note that if you attempt to roundtrip a class for which code was not generated (or for which the implementation file was erased), Rhapsody responds with an appropriate error message and that class remains unchanged.

Modifying Code Segments for Roundtripping

You can modify code segments for roundtripping that are procedural behaviors entered as operations and actions in statecharts.

Every segment in the implementation code has the following format:

```
... generated code
//[ segment-type segment-identifier
C++ code you entered
//]
... generated code continues
```

All these code segments are in the implementation files.

The only segments you can modify without losing the ability to roundtrip are as follows:

- ◆ Static reactions

```
//[ reaction reaction-id
// you can modify this code
someReactionCode();
// do not modify beyond the //[ sign
//]
```

- ◆ Exit action

```
//[ exitAction ROOT.idle.(Entry)
someExitAction();
//]
```

- ◆ Entry action

```
//[ entryAction ROOT.idle.(Entry)
someEntryAction();
//]
```

- ◆ Transition actions

```
//[ transition transition-id
someTransitionCode();
//]
```

- ◆ Primitive operations (procedural behaviors)

```
//[ operation doit()
someOperationCode();
someMoreOperationCode();
//]
```

Recovering Lost Roundtrip Annotations

To roundtrip your code, Rhapsody uses special annotations inserted by the code generator into the implementation file. The symbols are as follows:

Language	Annotation Symbols
Ada	Element: --++ <ElementType> <ElementName> Body: --+[<ElementType> <ElementName> --+]
C	Element: /*## <ElementType> <ElementName> */ Body: /*#[<ElementType> <ElementName> */ / *#]*/
C++ and Java	Element: //## <ElementType> <ElementName> Body: //#[<ElementType> <ElementName> //#]

Note

If you edit or delete these annotations, Rhapsody cannot trace your code back to the model.

To recover corrupted roundtrip annotations, follow these steps:

1. Rename the damaged file.
2. Regenerate code for this class. This should produce a new file with the correct annotations.
3. Copy your changes from the damaged file into the newly generated file.
4. Try to roundtrip again.

If you have modified any of the files, the following message is displayed:

```
File <filename> has been modified externally. Do you want to  
roundtrip?
```

If you modified the file contents, you must roundtrip to add the modifications to the model. Choose **Yes** to confirm the roundtrip. Rhapsody updates the model and the generated code reflects your manual modifications.

If you choose **No**, Rhapsody overwrites the modified files and your changes will be lost.

Roundtripping Classes

The following table lists the modifications that can be roundtripped in a class implementation file.

Element	Change
Constructors and operations	<ul style="list-style-type: none"> • Change the name of an argument. • When the name is changed, the argument description is lost. • You <i>cannot</i> change the argument type. • Modify bodies between the //#[and //#[or --+[and --+[delimiters.
State actions	Modify state actions (transition, entry, exit, and reactions in state) between the delimiters.
State-action actions	Modify the state-action actions (in activity diagrams) between the delimiters.
Static attributes	Add or modify (but <i>not</i> remove) the initial value.

Note

Actions may appear in the code multiple times.

Rhapsody roundtrips the first occurrence of the action code. If two or more occurrences are modified, the first modified occurrence is roundtripped. One technique is to call an operation in state, state action, and transition actions, thereby eliminating duplication of the action code and possible roundtrip ambiguity.

The following table lists the modifications that can be roundtripped in a class specification file.

Element	Change
Arguments	<p>Add, remove, and change the type of constructors, operations, and triggered operation arguments.</p> <p>Note that changes to argument descriptions in the class specification file are <i>not</i> roundtripped.</p>
Association	<p>Add or remove association, directed association, or aggregation.</p> <p>You must set the property <code>CPP_ or JAVA_Roundtrip::Update::AcceptChanges</code> property value to <code>All</code>.</p>
Attributes	<ul style="list-style-type: none"> • Modify the descriptions. <p>If there is a blank line at the end of the description, the description is lost.</p> <ul style="list-style-type: none"> • Add or remove attributes. <p>You must set the property <code>CPP_ or JAVA_Roundtrip::Update::AcceptChanges</code> property value to <code>All</code>.</p> <ul style="list-style-type: none"> • Modify the name, type, or access of existing attributes.
Classes	<ul style="list-style-type: none"> • Modify the descriptions. <p>If there is a blank line at the end of the description, the description is lost.</p> <ul style="list-style-type: none"> • Modify the class name. <p>In the next code generation, the modified class will be generated to new files, such as <code><new name>.h</code> and <code><new name>.cpp</code>. When using DMCA, you must close and reopen the class file to reassociate the class text with the proper class in the model.</p> <ul style="list-style-type: none"> • Add a new class. <p>The addition will be reflected under the associated package in the model.</p>
Constructors and operations	<ul style="list-style-type: none"> • Modify the descriptions. <p>If there is a blank line at the end of the description, the description is lost.</p> <ul style="list-style-type: none"> • Add or remove constructors or operations. <p>You must set the property <code>CPP_ or JAVA_Roundtrip::Update::AcceptChanges</code> property value to <code>All</code>.</p> <ul style="list-style-type: none"> • For the specification file, modify types for existing operation or constructor arguments, but not their names. For specification and implementation files, you can modify both the type and the name. However, if the change is only in the implementation file, you can only change the name and not the type. • Modify return types for existing operations.
Destructors	<p>Modify the descriptions.</p> <p>If there is a blank line at the end of the description, the description is lost.</p>
Nested classes	<p>Add, remove, or modify a nested class.</p>

Element	Change
Relations	<ul style="list-style-type: none"> Modify the descriptions. <p>If there is a blank line at the end of the description, the description is lost.</p> <ul style="list-style-type: none"> Modify the role name for an existing relation. <p>Given a relation "Class_1* itsClass_1", you can modify the role name <code>itsClass_1</code>. For directed associations, you can also modify the related class <code>Class_1</code> (for bidirectional association and aggregation, you cannot modify the related class).</p>
Standard operations	<p>Modify standard operations to inline, by adding "inline" to the declaration. Note that the definition is generated automatically—the property <code><lang>CG::Operation::Inline</code> is set to <code>in_source</code>. As a result, the implementation of the function stays in the implementation file. (The "inline" keyword is added to both, specification and implementation files.)</p>
Triggered operations	<p>Modify the descriptions.</p> <p>If there is a blank line at the end of the description, the description is lost.</p>
User-defined types	<p>Add, remove, or modify user-defined types.</p>

Roundtripping Packages

Rhapsody roundtrips function argument name changes in the package implementation file; however, changes to argument types are not roundtripped. When the name is changed, the argument description is lost.

Rhapsody does not roundtrip changes to the initial values of variables.

The following table lists the modifications can be roundtripped in a package specification file.

Element	Change
Event	<ul style="list-style-type: none"> Modify the description. <p>Changes to argument descriptions are not roundtripped.</p> <ul style="list-style-type: none"> Add or remove event "arguments." <p>Event arguments are actually attributes of the corresponding event class.</p> <ul style="list-style-type: none"> Modify an event "argument" type and name. <p>When the name is changed, the argument description is lost.</p>
Function	<ul style="list-style-type: none"> Modify the description. <p>Changes to argument descriptions are <i>not</i> roundtripped.</p> <ul style="list-style-type: none"> Add or remove a function. Modify an existing function's return type.

Element	Change
Function argument	<ul style="list-style-type: none">• Add or remove a function argument.• Modify an existing function's argument type. Changes to argument names are <i>not</i> roundtripped.
Instance	<ul style="list-style-type: none">• Add or remove an instance.• Modify an instance's name or class type.
Variable	<ul style="list-style-type: none">• Modify description. Changes to argument descriptions are not roundtripped. <ul style="list-style-type: none">• Modify a variable type or name.• Add or remove a variable.

To remove a function, variable, or instance with DMCA enabled, follow these steps:

1. Remove the element from the `.h` or `.cpp` file.
2. Switch focus to the `.cpp` or `.h` file while pressing the **Shift** key.
3. Remove the element from the second file.

The **Shift** key prevents DMCA from firing before you have made the changes to the second file.

To remove a function, variable, or instance with DMCA set to **None**, follow these steps:

1. Remove the element from the `.h` or `.cpp` file, then save the file.
2. Remove the element from the `.cpp` or `.h` file, then save the file.

Roundtripping Deletion of Elements from the Code

Note

This feature applies to C and C++ in Respect mode and Java in Advanced mode.

You can manually delete elements in code and use roundtripping to update your Rhapsody model. You can delete variables, functions, types (Struct, Union, Enum, typedef), types' members, attributes, operations, #define-s, #include-s, forward declarations, and associations. Note that you cannot delete auto-generated #include statements to the Rhapsody framework files.

The roundtripping deletion of elements from the code feature involves the following properties:

1. Depending on whether you have Rhapsody in C, C++, or Java:
 - ◆ For Rhapsody in C and C++: Set the `<lang>_Roundtrip::General::RoundtripScheme` property (for example, `CPP_Roundtrip::General::RoundtripScheme`) to Respect to turn on code respect, which is required for this feature. See [Activating the Code Respect Feature](#).
 - ◆ For Rhapsody in Java: Set the `Java_Roundtrip::General::RoundtripScheme` property to Advanced.
2. For C, C++, and Java: Because the `<lang>_Roundtrip::Update::AcceptChanges` property is, by default, set to `Default`, the feature to roundtrip hand-edited deletion of elements is enabled.

Note: Be aware of the following when the

`<lang>_Roundtrip::Update::AcceptChanges` property is set to `Default`:

- ◆ Deletion of the elements Classes, Actors, and Objects is disabled. In addition, deletion of elements is disabled when Rhapsody finds parser errors in the roundtripped code.

Note: You can enable deletion of all elements (no exceptions) and even if there are parser errors during roundtripping. To do so, set the `<lang>_Roundtrip::Update::AcceptChanges` property to `All`. You should consider the consequences of using the `All` value. For more information about this property, see [Update::AcceptChanges](#).

- ◆ Deletion of an element that has a prolog and/or epilog is disabled. (You enter values for the prolog and/or epilog in the following properties: `ImplementationProlog`, `SpecificationProlog`, `ImplementationEpilog`, `SpecificationEpilog`.)

Roundtripping for C++

The following details apply to roundtripping in C++ only:

- ◆ You can perform a Advanced (Full) roundtrip for language types (except language type inside class) in Rhapsody in C++.
- ◆ There is support of #includes and forward declarations.
- ◆ Roundtripping can convert auto-generated operations to user operations on modifying in code through either of the following methods:
 - By setting the `CG::CGGeneral::GeneratedCodeInBrowser` property to `Checked`.

This works for all auto-generated operations shown in the browser except constructors and destructors.
 - If the above property is not used because there are no auto-generated operations in the browser, then you can remove the “`///auto_generated” annotation of the operation so that user operations will be added to the model.`
- ◆ Roundtripping takes into account code changes made for all user-defined types, except a language type that is nested in a class.
- ◆ If you change the order of elements, the “code respect” option preserves the order of the following elements for the next code generation:
 - Global elements
 - Class elements
 - #includes and forward declarations
 - Auto-generated operations (excluding statechart and instrumentation code)
- ◆ Position of `<<friend>>` dependency will be preserved by roundtripping in code respect mode.
- ◆ You can roundtrip C++ templates.

Roundtripping for Java

The following details apply to roundtripping in Java only:

- ◆ Advanced (Full) roundtrip is supported for Java.
- ◆ You can add “import” statement in the code, which creates a dependency in the model.
- ◆ There is JDK 1.5 support for generics, enumerations, and type-safe containers.

For more information about roundtripping and Java, see the following topics:

- ◆ [Javadoc Handling in Reverse Engineering and Roundtripping](#)
- ◆ [Reverse Engineering / Roundtripping and Static Import Statements](#)
- ◆ [Reverse Engineering / Roundtripping and Static Blocks](#)

Roundtripping Properties

Rhapsody includes many properties to control roundtripping. They are specified in `<lang>_Roundtrip`, where `<lang>` is the programming language. For example, in Rhapsody in C, these properties are in `C_Roundtrip`; in Rhapsody in C++, they are in `CPP_Roundtrip`.

A definition for each property is provided on the applicable **Properties** tab of the Features dialog box. The following table lists the properties that control roundtripping.

Property	Description
<code>General::NotifyOnInvalidatedModel</code>	Determines whether a warning dialog box is displayed during roundtrip. This warning is displayed when information might get lost because the model was changed between the last code generation and the roundtrip operation. This property is available only in Rhapsody in C and C++.
<code>General::ParserErrors</code>	Specifies the behavior of roundtrip when a parser error is encountered.
<code>General::PredefineIncludes</code>	Specifies the predefined include path for roundtripping. This property is available only in Rhapsody in C, C++, and Java.
<code>General::PredefineMacros</code>	Specifies the predefined macros for roundtripping. This property is available only in Rhapsody in C and C++.
<code>General::ReportChanges</code>	Defines which changes are reported (and displayed) by the roundtrip operation. This property is available only in Rhapsody in C, C++, and Java.

Property	Description
<code>General::RestrictedMode</code>	<p>The <code>RestrictedMode</code> property is a Boolean value (<code>Checked</code> or <code>Cleared</code>) that specifies whether restricted-mode roundtripping is enabled. This property can be modified on the configuration level. (Default = <code>Cleared</code>)</p> <p>Restricted mode of Advanced (Full) roundtrip enables you to roundtrip unusual usage of Rhapsody elements, such as a class declaration in a user-defined type. Restricted mode has more limitations, but preserves the model from unexpected changes. The additional <i>limitations</i> for restricted mode are as follows:</p> <ul style="list-style-type: none"> • User-defined types cannot be removed or changed on roundtrip because Rhapsody code generation adds the “Ignore” annotation for a user-defined type declaration. • Relations cannot be removed or changed on roundtrip. • New classes are not added to the model. <p>This property is available only in Rhapsody in C and C++.</p>
<code>General::RoundtripScheme</code>	<p>Specifies whether to perform a <code>Basic</code>, <code>Advanced</code> (for C, C++, and Java only), or <code>Respect</code> (for C and C++ only) roundtrip. <code>Basic</code> is the default for Ada, <code>Advanced</code> for Java, and <code>Respect</code> for C and C++.</p>
<code>Update::AcceptChanges</code>	<p>The <code>AcceptChanges</code> property is an enumerated type that specifies which changes are applied to each code generation element (attribute, operation, type, class, or package). You can apply separate properties to each type of code generation element.</p> <p>The possible values are as follows:</p> <ul style="list-style-type: none"> • <code>Default</code> means that all the changes can be applied to the model element, including deletion. However, note that deletion is disabled for classes, actors, and objects. In addition, deletion is disabled if Rhapsody finds parser errors in the roundtripped code. This is the default value. • <code>All</code> means all of the changes can be applied to the model element. There are no exceptions (as there are for the <code>Default</code> value). • <code>NoDelete</code> means all the changes except deletion can be applied to the model element. This setting prevents accidental removal of operations, constructors, attributes, relations, variables, instances, and functions. • <code>AddOnly</code> means to apply only the addition of an aggregate to the model element. You cannot delete or change elements. • <code>NoChanges</code> means do not apply any changes to the model element. <p>Note that the value of the property is propagated to all the aggregates of an element. Therefore, if a package has the property value <code>NoChanges</code>, no elements in that package will be changed.</p> <p>This property is available only in Rhapsody in C, C++, and Java.</p>

Code Respect

Note

The code respect feature applies to Rhapsody in C and Rhapsody in C++, and the reverse engineering and roundtripping features in these products. As of Rhapsody 7.2, any new project you create has the code respect featured activated by default. To activate code respect for an old project, see [Activating the Code Respect Feature](#).

In Rhapsody, code respect means that the order of elements in the original code is preserved during code generation. This means that you can freely change the order of class members and globals and Rhapsody “respects” those changes. Code respect has these additional features:

- ◆ Code generation regenerates text fragments to the correct place in file.
- ◆ Reverse engineering imports `#ifdef`-s to the model as a verbatim text.
 - The branches of `#ifdef`-s that are seen by the compiler are modeled as logical elements.
 - The branches of `#ifdef`-s that **are not** seen by the compiler are modeled as a verbatim text.

This means that code generated in Rhapsody is similar to the original. This gives you complete flexibility for using manually written code or auto-generated code while receiving all the benefits of modeling. You can reverse engineer C++ and C code into a model in a manner that the model respects the order, location, and dependencies of the global elements in the original code. See [Reverse Engineering](#).

In addition, you can set up Rhapsody in C++ and C so that you can roundtrip code into the Rhapsody model that respects the structure of the code and preserves this structure when code is roundtripped in the Rhapsody model.

When you have changed the order of elements in C++ and C, roundtripping in respect mode preserves the order of the following elements for the next code generation:

- ◆ Global elements
- ◆ Class elements
- ◆ `#includes` and forward declarations
- ◆ Auto-generated operations (excluding statechart and instrumentation code)

See [Roundtripping](#).

Activating the Code Respect Feature

To activate the code respect feature for Rhapsody in C++ and Rhapsody in C, follow these steps:

Note: As of Rhapsody 7.2, any new project you create has the code respect featured activated by default. Any old projects opened in Rhapsody 7.2 or higher retain their original roundtrip scheme.

1. Open the Features dialog box.
2. On the **Properties** tab, select the **View** drop-down arrow and select **All**.
3. Expand `<lang>_Roundtrip` and then expand **General**.
4. For the **RoundtripScheme** property, select **Respect**.
5. Click **OK**.

Note that the code respect function is based on elaborating SourceArtifact files (previously known as component files).

Where Code Respect Information is Defined

Note

This feature applies to Rhapsody C++ and Rhapsody C.

Code respect information (such as mapping, ordering, and code snippets) of an element is defined in a *SourceArtifact* element, which is typically created by reverse engineering or roundtripping.

Note that previous to Rhapsody version 7.2, a *SourceArtifact* was referred to as a component file. While component files still exist, they now refer to elements under the **Components** category in Rhapsody. When component files are located under packages or classes, and so forth, they are referred to as *SourceArtifacts*.

Note

For existing component files under a component (for example, created by a user or in old models), their locations are not changed.

Because a *SourceArtifact* (for example, a .h file) is located under its applicable class/package/object/block, a configuration management operation of the element includes any *SourceArtifact*.

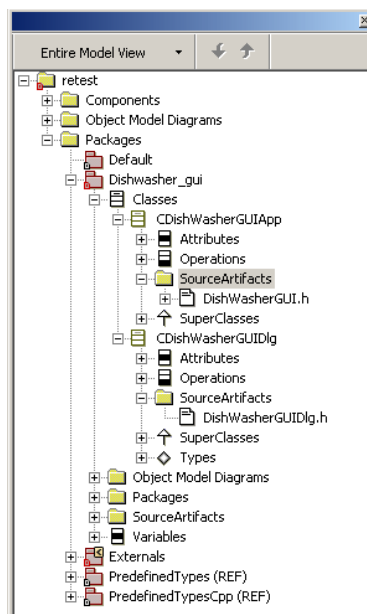
By default the code respect feature is enabled. When enabled, the following properties have the following values:

- ◆ `<lang>_ReverseEngineering::ImplementationTrait::LocalizeRespectInformation` property set to `Checked`.
- ◆ `<lang>_ReverseEngineering::ImplementationTrait::RespectCodeLayout` property set to `Ordering`.
- ◆ `<lang>_Roundtrip::General::RoundtripScheme` property (for example, `CPP_Roundtrip::General::RoundtripScheme`) set to `Respect`. See [Activating the Code Respect Feature](#).

Making SourceArtifacts Appear on the Rhapsody Browser

By default, SourceArtifacts do not appear on the Rhapsody browser as only advanced Rhapsody users may want to view or work with them in Rhapsody. Letting Rhapsody and the reverse engineering/roundtripping process manipulate (that is, create and edit) SourceArtifacts is recommended.

To show SourceArtifacts on the Rhapsody browser (after you have reverse engineered code into Rhapsody), choose **View > Browser Display Options > Show Source Artifacts**. SourceArtifacts are filed in a **SourceArtifacts** folder, as shown in the following figure where the folder appears under its class.



Manually Adding a SourceArtifact

A SourceArtifact is created when you reverse engineer or roundtrip code in Rhapsody, which is the typical and recommended way to do so. Advanced users may also want to manually add a SourceArtifact, which you can do through the Rhapsody browser.

To manually add a SourceArtifact, follow these steps:

1. To enable the display of SourceArtifacts on the Rhapsody browser, choose **View > Browser Display Options > Show Source Artifacts**. See [Making SourceArtifacts Appear on the Rhapsody Browser](#).
2. To add a SourceArtifact, right-click a class/package/object/block and select **Add New > SourceArtifact**.

Reverse Engineering and SourceArtifacts

Note the following rules for reverse engineering and locating a SourceArtifact under a class:

- ◆ If only one class is mapped to the artifact, the artifact is located under the class it is mapped to.
- ◆ If more than one class is mapped, the artifact is located under the first class. Priority is given to a class with the same of the file.
- ◆ If no class is mapped to the artifact, the artifact is located under the package.
- ◆ External files are imported under the component.

Roundtripping and SourceArtifacts

Note the following rules for roundtripping and locating a SourceArtifact, which are similar as the ones for reverse engineering:

- ◆ New SourceArtifacts are added under classes.
- ◆ The order of elements is updated.
- ◆ For existing component files under a component (for example, created by a user or in old models), their locations are not changed.

Code Generation and SourceArtifacts

Note the following rules for code generation and locating a SourceArtifact:

- ◆ Only advanced code generation supports SourceArtifacts.
- ◆ If a class/package belongs to a scope of component, the artifacts under the class are generated according to their code respect information, the same as other component files.
- ◆ If more than one class is mapped to a SourceArtifact, all classes are generated.
- ◆ Other elements mapped to the SourceArtifact are also generated.

Location of the Generated Files

Note the following rules in relation to the location of the generated files:

- ◆ According to folder hierarchy under component. If a class appears in the scope of the component as an element of the folder, the class and its SourceArtifacts aggregates will be generated according to the path of the folder.
- ◆ According to the “package as directory” policy, the package hierarchy determines the folder hierarchy.

Configuration Management and SourceArtifacts

Note the following configuration management considerations:

- ◆ A SourceArtifact under a class/object/block cannot be saved as a unit. Checking in/out the class leads to automatically checking in/out its code respect information.
- ◆ The **Corresponding Component File** check box (on the Add to Archive Options dialog box for configuration management) will not appear unless the configuration management operation is done for a class that is mapped to other’s class SourceArtifact. The same is true for other the other configuration management operations (check in, check out, and so forth).

Animation

Rhapsody animation is a key technology that enables model validation—you can validate the analysis and design model by tracing and stimulate the executable model. In addition, the Rhapsody animator helps you debug your system at the design level rather than the source code level by actually executing the model and animating the various UML design diagrams. Rather than merely simulating the application and viewing values of variables and pointers, you see actual values of instances of states and relations.

The animator enables you to juxtapose different views of an application while it is running. You can watch the animated model executing in any of the following views:

- ◆ Sequence diagrams
- ◆ Statecharts
- ◆ Activity diagrams
- ◆ Browser
- ◆ Event Queue window
- ◆ Call Stack window
- ◆ Output window

Simultaneously viewing animated sequence diagrams, animated statecharts, animated activity diagrams, and the animated browser in adjacent windows as the model is executing enables you to verify that the design behaves as intended. Highlighting in the animated diagrams helps you to pinpoint the current state of execution.

While the model is running, you can use the **Animation** toolbar to step through the program, set and clear breakpoints, and inject events to observe how the system reacts in quasi-real time. You can observe the system's operation either in the animated views or by generating an output trace.

Animation Overview

The animator is a *design-level* debugger, as well as a model validator. In other words, the animator supports the standard functionality of a programming language debugger at the design level. The objects you follow are design-level objects; that is, objects that are modeled in Rhapsody.

Animation Features

During an animation session, you can perform the following activities:

- ◆ Inspect and modify the current status of the model:
 - View current instances and the relationships between them.
 - View the current state for reactive objects.
 - View animated sequence diagrams depicting events and operations actually sent or called.
 - Generate events.
- ◆ Open or close animated views.
- ◆ Set breakpoints.
- ◆ Advance execution using the Go buttons on the **Animation** toolbar.

General Procedure to Prepare for Animation

The following procedure lists the general steps to prepare for and run animation, with references to the more specific procedures.

1. If necessary, create a component. See [Creating a Component](#).
2. If necessary, create a configuration. See [Creating a Configuration](#).
3. Set the Instrumentation mode for the configuration to **Animation**. See [Setting the Instrumentation Mode](#).
4. Set the active configuration. The active configuration is the one generated when you generate code. See [Setting the Active Configuration](#).
5. Generate code for the configuration. See [Generating Code](#).
6. Build the animated component. See [Building the Target](#).
7. Run the animated component. See [Running the Animated Model](#).

Creating a Component

A *component* is a physical subsystem in the form of a library or executable program. It plays an important role in the modeling of large systems that contain several libraries and executables. Each component contains configuration and file specification categories, which are used to generate, build, and run the executable model.

Each project contains a default component, named `DefaultComponent`. You can use the default component (you can rename it) or create a new component.

To rename the default component, follow these steps:

1. In the Rhapsody browser, expand the **Components** category.
2. Double-click **DefaultComponent** to open the Features dialog box.
3. In the **Name** box, replace the name `DefaultComponent` with another name.
4. Click **Apply**.
5. To set the features for this component, see [Setting the Component Features](#)

To create a new component, follow these steps:

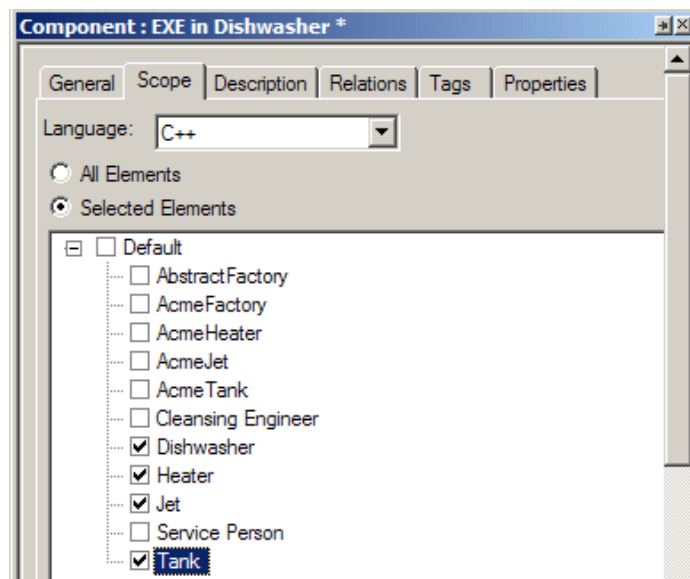
1. In the Rhapsody browser, right-click the **Components** category and select **Add New Component**.
2. Type a name for your new component and press **Enter**.
3. To set the features for this component, double-click the new component to open its Features dialog box and see [Setting the Component Features](#).

Setting the Component Features

Once you have created the component, you must set its features.

To set the component features, follow these steps:

1. With the Features dialog open for your component, on the **General** tab, in the **Type** group, select the **Executable** option button if it is not already selected.
2. On the **Scope** tab, specify which model elements to include in the component.
 - ♦ **All Elements.** Select this option button if you want to select all available elements.
 - ♦ **Selected Elements.** Select this option button if you want to select only certain elements and then use the check boxes next to each element to indicate which model elements to include in the component. Notice that if you select check box next to the parent element, all sub-elements are selected. If you only want certain sub-elements, select the check boxes next to those sub-elements instead of the parent element, as shown in the following figure:



3. Click **OK**.

Creating a Configuration

A component can contain many configurations. A *configuration* specifies how the component is to be produced.

Each component contains a default configuration, named `DefaultConfig`. You can use the default configuration (you can rename it) or create a new one.

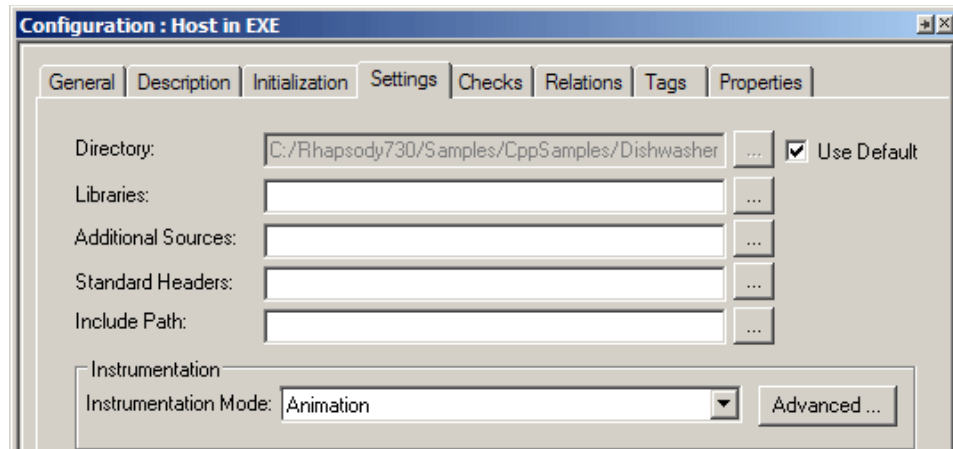
To rename the default configuration, follow these steps:

1. In the Rhapsody browser, expand the applicable component and its **Configurations** category.
2. Double-click **DefaultConfig** to open the Features dialog box.
3. In the **Name** box, replace `DefaultConfig` with another name.
4. Click **Apply**.
5. To enable animation, you must set the instrumentation mode. See [Setting the Instrumentation Mode](#).

Setting the Instrumentation Mode

To set the Instrumentation mode for a configuration, follow these steps:

1. With the Features dialog box open for your configuration, on the **Settings** tab, set the **Instrumentation Mode** box to **Animation**, as shown in the following figure. This adds instrumentation code, which makes it possible to animate the model.



2. Optionally, to instrument operations and set a finer scope on the instrumentation, click the **Advanced** button to open the Advanced Instrumentation Settings dialog box. For more information, see [Using Selective Instrumentation](#).
3. Click **OK**.

Running the Animated Model

When running the animated model, you may use any of these animation methods:

- ◆ Run an executable application on the same machine as Rhapsody (see [Running on the Host](#)).
- ◆ Run an executable application on a different machine than Rhapsody (see [Running on a Remote Target](#) and [Testing an Application on a Remote Target](#)).
- ◆ Test a library built with Rhapsody with a GUI built outside of Rhapsody (see [Testing a Library](#)).

For all of these animation methods, the process follows these steps:

1. The application auto-connects to Rhapsody to run the animation.
2. The message “Initializing animation...” is displayed.
3. An operating system window opens to display console output from the application. You can minimize this window, if desired.
4. The **Animation** toolbar appears (see [Animation Toolbar](#)).

Running on the Host

To run the application on the host, follow these steps:

1. Open a command prompt window.
2. Type the DOS command `ipconfig`.
3. Copy the IP address of the host into the command and press **Enter**.
4. Open your Rhapsody project.
5. Highlight your EXE for the host in the browser and right-click to display the Features dialog box.
6. Set the `<Language>_CG::<Compiler>::UseRemoteHost` property value to **Checked**.
7. Also in the properties list, locate the `<Language>_CG::<Compiler>::RemoteHost` property and paste in the IP address of the host for its value. Click **OK**.
8. Generate and make the executable.

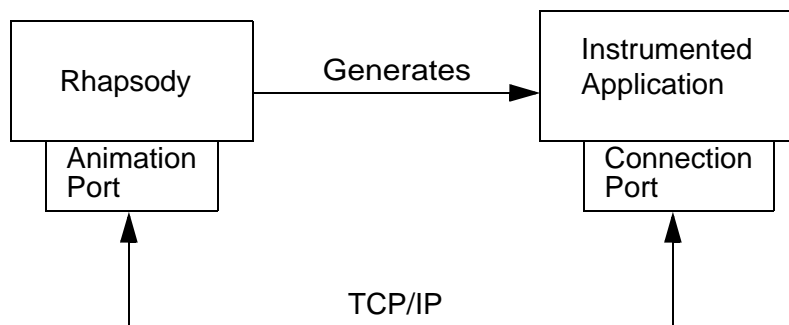
Running on a Remote Target

You can inspect or debug animated code running on a remote target through these steps:

1. Copy the executable from the host to the target.
2. Run the executable.
3. Check to be certain that the animation is running on the host.

The animation views, shown on the host, are at the design level. The animation server communicates with the target via a TCP/IP connection. Each animation connection requires its own unique port number, which is set in the `rhapsody.ini` file. The framework inserts the same port number into the connection port of the instrumented application.

The instrumented application can run on either the same machine as Rhapsody (the host) or a remote target. The following illustration shows the relationship of these components to each other.



Opening a Port Automatically

If you want Rhapsody to locate an open port automatically for animation, change the values of the following variables in the `[General]` section of the `rhapsody.ini` file:

- ◆ **AnimationPortNumber** defines the port to try first
- ◆ **AnimationPortRange** specifies the number of ports to test, in addition to `AnimationPortNumber`, before giving up

These changes set up the search to start with the `AnimationPortNumber` and increment by one until either an open port is found or until the number of tries equals the value of `AnimationPortRange + 1`.

For example, if `AnimationPortNumber = 5000` and `AnimationPortRange = 100`, Rhapsody first attempts to establish a connection on port 5000. If that fails, then it tries port 5001. If that fails, then it tries port 5002. This search continues until either an open port is found or until it finally tests port 5100. If port 5100 fails, then no animation can be performed for that instance of Rhapsody.

In addition to the animation ports, another `rhapsody.ini` file value needs to be manually set. In the `[General]` section, change the `EnableMultipleAnimation` from `FALSE` (default) to `TRUE`.

Testing an Application on a Remote Target

To test an application running on a remote target, follow these steps:

1. Open a command window and change to the directory where the application is located.
2. At the command prompt, enter the command to run the application with the `-hostname` option indicating the machine running Rhapsody. For example, if your application is named `myapp.exe` and Rhapsody is running on a machine named Julius, run your application using the following command:

```
myapp -hostname Julius
```

Testing a Library

To test a Rhapsody library, follow these steps:

1. Build the library inside Rhapsody.
2. Build the application outside Rhapsody and include the library built in Rhapsody.
3. Open a command window and change to the directory where the application is located.
4. At the command prompt, enter the command to run the application. For example, if your application is named `myapp.exe`, use the following command:

```
myapp
```

Partially Animating a Model (C/C++)

Rhapsody enables you to partially animate a model, to test selected elements without the overhead of animating the entire project. This feature also enables you to animate projects that contain non-animated components.

All elements in the model are generated and built, but only the animated elements are displayed in the animation environment during run time. Animated and nonanimated elements, listed below, are able to pass instances and events to one another, but only animated events can be generated using the animation environment.

- ◆ Package
- ◆ Class (including nested classes)
- ◆ Statechart
- ◆ Activity diagram
- ◆ Relation
- ◆ Attribute
- ◆ Actor
- ◆ Operation
- ◆ Event

Note

Partial animation also applies to trace operations.

Setting Elements for Partial Animation

1. Set the instrumentation of the active component to **Animation**.
2. By default, all elements are animated. Therefore, partial animation is a process of elimination.

For each element that you do *not* want animated, set the `<lang>_CG::<type_of_element>::Animate` property to `False`.

The `<type_of_element>` metaclass can be `Package, Class, Statechart, Relation, Attribute, Operation, Event, Or Actor`.

3. Save the model.
4. Generate, make, and run the project.

Note: You can specify partial animation per configuration using the Advanced Instrumentation Settings dialog box. See [Using Selective Instrumentation](#) for more information.

Partial Animation Considerations

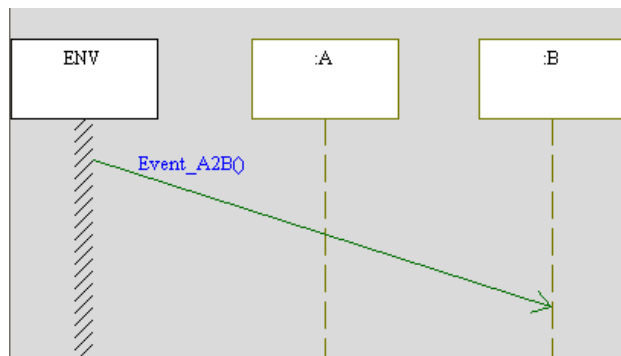
- ◆ Animation for components should be set at the configuration level.
- ◆ If any element in the executable is animated, the instrumentation should be set to **Animation** in the Configuration setting for the executable component. The component, which creates the executable, is linked with the instrumented OXF libraries if at least one part of the model (aggregate, library, and so on) is animated.
- ◆ If an operation is animated, its arguments are also animated. When setting animation for arguments, it is enabled or disabled for all arguments. Arguments cannot be animated individually.
- ◆ The animation setting of a class affects all of its instances, even if the package they belong to is not animated.
- ◆ Only animated events can be injected using the animation environment.
- ◆ Only animated events appear in the event queue.
- ◆ Elements that are not animated are not shown in any animated view.
- ◆ Only animated attributes or relations are shown for an animated instance.
- ◆ Only animated operations, timeouts, and events are displayed in the animated sequence diagram and the call stack.
- ◆ If a model contains an active class that is not animated, its thread is considered a foreign thread. This thread appears in the thread list, but is named by its operating system handle (not by the name of the active object).

- ◆ When using Rhapsody-generated DLLs, the OXF libraries should also be a DLL. This DLL should be animated if any of the Rhapsody-generated DLLs are animated.

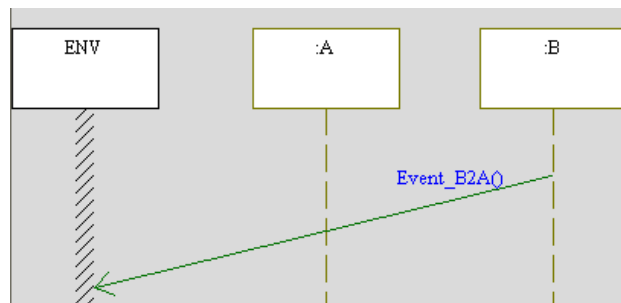
Partially Animated Sequence Diagrams

Instances that are not animated are not displayed in animated views. However, Rhapsody does display messages passed between animated and nonanimated instances.

If a nonanimated instance (A) sends a message, it is shown originating from the system border.



If an animated instance (B) sends a message to a nonanimated instance (A), it is shown being sent to the system border.



A message sent from a nonanimated class is shown coming from its call stack predecessor, if present. Messages sent from a nonanimated class whose call stack predecessor is absent are shown originating from the system.

Messages sent from one nonanimated class to another are not shown.

Ending an Animation Session

To end an animation session, use any of the following methods:

- ◆ In the **Animation** toolbar, click the **Quit Animation** tool.
- ◆ Select **Code > Stop**.
- ◆ Allow the application to terminate.

Animation Toolbar

When you run an executable model with instrumentation set to **Animation**, the Animation toolbar is displayed. Like all other Rhapsody toolbars, the Animation toolbar is dockable so you can dock it to one side of the Rhapsody window, if desired.

The **Animation** toolbar contains the following tools.



Go Step. Advances the application a single step—until the next Rhapsody-level occurrence, such as the calling of an operation.



Go. Advances the application until it terminates or reaches a breakpoint.



Go Idle. Advances the application until the next timeout or event for the focus thread. If there are no timeouts or events waiting in the event queue, nothing happens.



Go Event. Advances the application until the next event is dispatched or the executable reaches an idle state.



Break. Interrupts a model that is executing.



Command Prompt. Issues an animation command, for example, to manually inject an event into the model.



Quit Animation. Ends an animation session.



Thread. Invokes the thread view.



Breakpoints. When a breakpoint is encountered in any thread, the animator switches control from the application to you. The last thread to reach a breakpoint becomes the focus thread.



Event Generator. Generates events to inject into the model.



Call operations. Enables you to invoke operation calls during animation and tracing to validate parts of the design model.



Silent/Watch. Toggles between Silent and Watch mode at any break or when the executable is idle.

Silent mode enables you to perform design-level debugging and display animation information only at breakpoints. In contrast, Watch mode enables you to continually update animation information in normal step-by-step operation via the **Go** buttons.

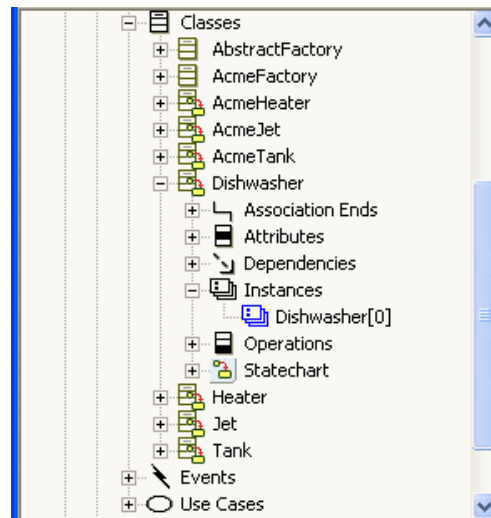
Note that all the **Go** commands cause the tracer to execute immediately, even if it is in the middle of reading commands from a file. The subsequent (unread) lines are used as commands the next

time the tracer stops execution and searches for commands. If the model reaches a breakpoint, control can return to you before a **Go** command is complete.

Creating Initial Instances

It is a good idea to issue a **Go Idle** command immediately after starting an executable model so all initial instances are created.

To create instances, click **Go Idle** after starting the model. The initial instances are created (as well as any instances created by those instances) and are listed under the Instances category for the class in the browser.



The instance name is in the format `class[n]`, where n is the number of instances, beginning with 0, that have been created since the model began executing. It is not possible to change this number.

Break Command

To interrupt a model that is executing, click the **Break** tool.

The **Break** command enables you to regain control immediately (or as soon as possible). Issuing a **Break** command also suspends the clock, which resumes with the next **Go** command.

Note

For simple applications, there might be a backlog of notifications. Although the model stops executing immediately, the animator can accept further input only after it has cleared this backlog and displayed any pending notifications.

The **Break** command cannot stop an infinite loop that resides within a single operation. For example, issuing a **Break** cannot stop the following `while()` loop:

```
while (TRUE) j++;
```

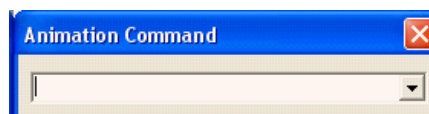
However, it can stop the following code if `increaseJ()` is an operation defined within Rhapsody:

```
while (TRUE) increaseJ();
```

Command Prompt

To issue an animation command, for example, to manually inject an event into the model, click the **Command Prompt** tool.

The Animation Command bar appears, as shown in the following figure:



You can also generate events using the **Event Generator** tool. See [Event Generator](#).

Generating Events Using the Animation Command Bar

Before you can inject an event into the model, the instance to which you are sending the event must already exist. See [Event Generator](#).

To generate an event, follow these steps:

1. In the Animation Command bar, type a `GEN()` command using either of the following formats:

- ◆ `instance->GEN(event)`
- ◆ `instance->GEN(event())`

2. Press Enter.

If the event is one that the instance is able to receive, the event is entered into the call stack (see [Call Stack](#)) in the following format:

```
instance->event()
```

If the instance is not able to receive the event, or no such event is defined in the package, the message "Invalid event name or non-existing event class <event>" is displayed.

3. To process the event, click one of the **Go** tools.

Events with Arguments

If the event has arguments, the GEN command is as follows:

```
instance->GEN(event(parameter[, parameter]*))
```

In this command:

- ◆ `instance`—Specifies the name of the instance to which you are sending the event (using the format `class[n]`)
- ◆ `event`—Specifies the name of the event you want to generate
- ◆ `parameter`—Specifies the values of the actual arguments to be passed to the event

When the event is generated, the actual argument names and their values appear in the call stack in the following format:

```
instance->event(argument = parameter[,  
argument = parameter]*))
```

If an event has arguments, you should provide the GEN command with the correct number of parameters and the correct types. For example, if event `X` is defined as `X(int, B*, char*)` where `B` is a class defined in Rhapsody, to generate an event you can enter either of the following commands:

```
A[1]->GEN(X(3,B[5],"now"))  
A[1]->GEN(X(1,NULL,"later"))
```

Event arguments must be either pointers to classes defined in Rhapsody, or of a type that can be read from a string, such as `int` or `char*`. If you want to generate an event of a user-defined type that you have defined either inside or outside of Rhapsody, you must either overload its I/O stream operator `>>(istream&)`, or instantiate the template `string2X(T& t)` so Rhapsody can interpret the characters entered.

The command `A[1]->GEN(Y(1))` works because the `>>` operator automatically converts the character "1" to the integer 1. On the other hand, the command `A[1]->GEN(Y(one))` would not work because the `>>` operator cannot convert the characters "one" to an integer.

Note

If you pass complex parameters (such as `structs`) and use animation, you must override the `>>` operator. Otherwise, Rhapsody generates compilation errors.

Generating Events Using the Command History List

The Animation Command bar has a drop-down list of commands previously issued in the same animation session. You can select a previous command from this list, rather than retyping it.

To select a previous command from the history list:

1. Click the down arrow to the right of the command-entry box.
2. From the drop-down list, select the command you want to reissue.

Threads

Classes that are both active and reactive run on their own threads. In animated applications with multiple threads, the thread view enables you to control the execution of threads. The animator always maintains one thread in focus. All thread-related operations are performed with respect to the focus thread:

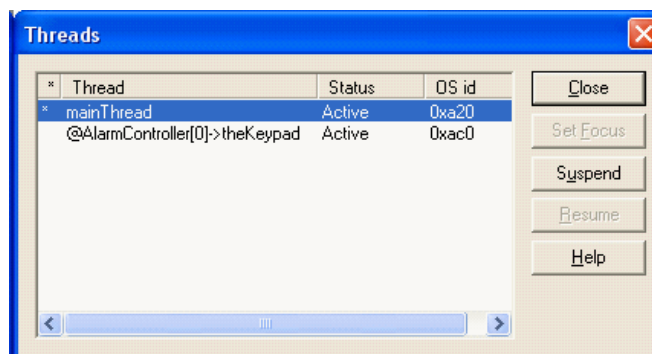
- ◆ The call stack view displays the call stack of the focus thread (see [Call Stack](#)).
- ◆ The event queue view displays the event queue of the focus thread (see [Event Queue](#)).
- ◆ The **Go** buttons on the **Animation** toolbar affect the execution only of the focus thread.

A **Go Step** advances the focus thread one step. While the focus thread is executing, all other threads execute as much as they can until the moment the focus thread finishes executing this step. It is impossible to predict how far non-focus threads will advance, because their behavior depends on how the operating system scheduler behaves during run time.

Thread View

To invoke the thread view, click the **Thread** tool.

The Threads dialog box opens, listing all threads that currently exist in the model, their status (Active or Suspended), and their operating system IDs (in hex).



Setting the Thread Focus

The focus thread has an asterisk in the first column. In the figure, the main thread has focus. This means that the call stack and event queue views will display information for this thread only.

1. Select a thread that does not currently have focus. The **Set Focus** button becomes active.
2. Click **Set Focus**.

To suspend an active thread, select an active thread and click **Suspend**.

Note

The Animator cannot advance the application if the focus thread is suspended.

To resume a suspended thread, select a suspended thread and click **Resume**.

Names of Threads

The first thread of your application is called the `mainThread`. The `mainThread` is also the system thread, and objects that have sequential concurrency run on this thread.

Threads associated with active objects are named according to their object names:

- ◆ Threads associated with active objects are denoted by `@objectName`. For example, if `A` is an active class, `@A[0]` is the name of a thread on which an instance of `A` might run.
- ◆ Threads associated with active objects that are targets of relations are denoted by `@objectName->roleName`. For example, if `A` is an active class that is related to `B` as `itsA`, `@B[3]->itsA` is the name of a thread on which an instance of `A` might run.

You can register external threads that you have manually created (using an operating system API call rather than the Rhapsody `OMThread` wrapper) by assigning your own names to them. The registering of an external thread introduces it to Rhapsody and adds it to the list of threads associated with active objects displayed in the Threads dialog box.

During the course of execution, additional external threads can appear that interact with Rhapsody-defined objects. These external threads are denoted by an ID that the operating system automatically assigns to them.

Notes on Multiple Threads

Go commands speak explicitly to the focus thread and send an implicit **Go** to all other threads. For example, in a multithreaded environment with three threads named `@T1`, `@T2`, `@T3`, and `@T2` having focus, a **Go Step** command would advance `@T2` a single step. During this time, threads `@T1` and `@T3` might advance one or more steps depending on the scheduling policy of the underlying operating system. In any case, when control returns to you, all three threads have executed a whole number of steps (execution does not stop in the middle of a step).

Only *active* (not suspended) threads are advanced in a **Go** command. If the focus thread is suspended, the execution does not advance and you are prompted to either set the focus to another thread or resume the focus thread. If the focus thread dies during a **Go Step**, **Go Event**, or **Go Idle** command, the application immediately stops.

Active Thread Properties

For the VxWorks and pSOSystem environments, you can assign a name to an active thread by modifying the `CG::Class::ActiveThreadName` property for the class. The default value of this property is the empty string. The active thread name is used only by the external source debugger (for example, the Tornado debugger), and is not the same as the thread name that is used for Rhapsody animation.

For all environments, you can set the thread priority by modifying the `ActiveThreadPriority` property for the class. The default priority for all threads is taken from the `DefaultThreadPriority` static class member of the `OMOSThread` framework class (defined in `Share\oxf\os.h`).

For all environments, you can set the initial size of the thread stack by modifying the `ActiveStackSize` property for the class. This property helps provide support for static memory architectures. The default size for thread stacks is taken from the `DefaultStackSize` static class member of the `OMOSThread` framework class.

Breakpoints

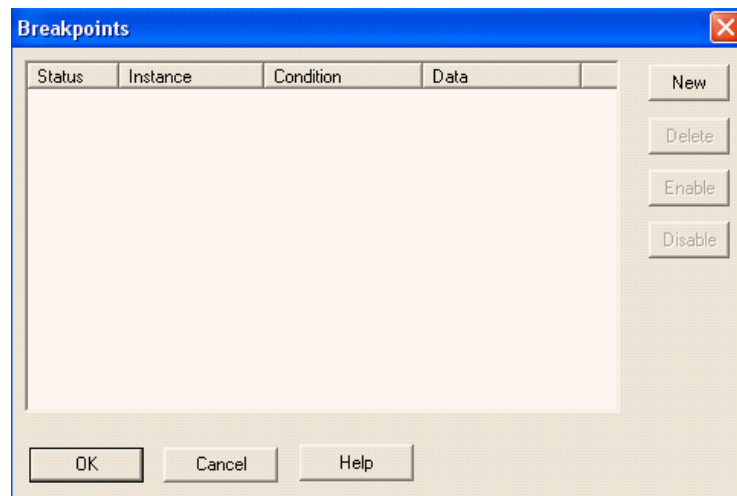
When a breakpoint is encountered in any thread, the animator switches control from the application to you. The last thread to reach a breakpoint becomes the focus thread.

You can control breakpoints by either issuing breakpoint commands in the Animation Command bar or using the Breakpoints tool.

1. Click the **Command Prompt** tool in the **Animation** toolbar.
2. In the Animation Command bar, enter **Show #Breakpoints** (not case-sensitive).

Note: Alternatively, you can control breakpoints using the Breakpoints dialog box. To activate this dialog box, click the **Breakpoints** tool in the **Animation** toolbar.

The Breakpoints dialog box opens, as shown in the following figure:

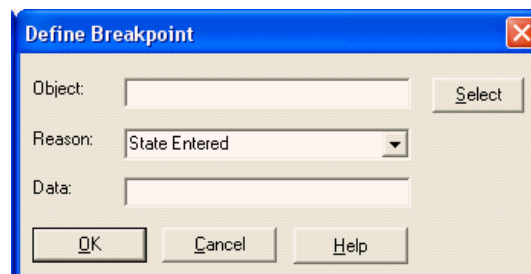


Using this dialog box, you can define, delete, enable, and disable breakpoints.

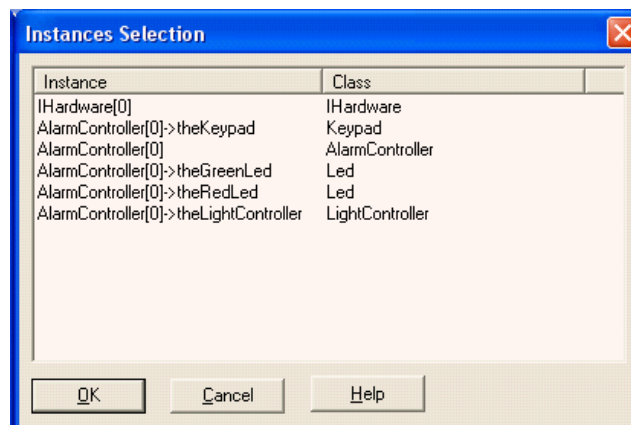
Defining Breakpoints

To define a new breakpoint, follow these steps:

1. In the Breakpoints dialog box, click **New**. The Define Breakpoint dialog box opens.



2. Click **Select** to select the object for which you want to define the breakpoint, or type the name of the instance directly into the **Object** field. The Instances Selection dialog box opens.



3. Select the instance for which you want to define the breakpoint from the list, then click **OK**.

The Instances Selection dialog is dismissed and the selected object appears in the Object box of the Define Breakpoint dialog box.

Note: In general, entering a class name for the object causes the breakpoint to operate on any instance of the class, whereas entering an instance name causes the breakpoint to operate on a particular instance.

4. Click the down arrow to the right of the **Reason** box to view a list of possible reasons for the breakpoint. Select the appropriate reason.

Some reasons might require additional data. For example, if you want to regain control when an object enters a particular state, you must provide the state name. If a state name is not provided, the break occurs when the object enters any state.

The following table shows the possible reasons for breakpoints and what, if any, optional data you can provide for each breakpoint.

Reason for Break	Object	Data	Description
Instance Created	Class	None	Break when any instance of the class is created.
Instance Deleted	Class or instance	None	Break when an instance of the class is deleted.
Termination	Class or instance	None	Break when an instance reaches a termination connector in its statechart.
State Entered	Class or instance	State name	Break when an instance enters a state.
State Exited	Class or instance	State name	Break when an instance exits a state.
State	Class or instance	State name	Break when an instance: <ul style="list-style-type: none"> • Enters a state • Exits a state
Relation Connected	Class or instance	Relation name	Break when a new instance is connected to a relation.
Relation Disconnected	Class or instance	Relation name	Break when an instance is removed from a relation.
Relation Cleared	Class or instance	Relation name	Break when a relation is cleared for an instance.
Relation	Class or instance	Relation name	Break when: <ul style="list-style-type: none"> • A new instance is connected to a relation. • An instance is deleted from a relation. • A relation is cleared for an instance.

Reason for Break	Object	Data	Description
Attribute	Instance	None	Break when any attribute of the instance changes value. A copy of the attribute values is stored and current values are compared to this copy. When a break occurs, the copy is updated with the latest values.
Got Control	Class or instance	None	Break when an instance gets control by: <ul style="list-style-type: none"> Starting to execute one of its user-defined operations Responding to an event Regaining control after an operation that the instance has called on another object finishes executing
Lost Control	Class or instance	None	Break when an instance loses control by: <ul style="list-style-type: none"> Finishing execution of one of its operations Finishing a reaction to an event Calling an operation of another object
Operation	Class or instance	Operation name	Break when an instance starts executing a user-defined operation.
Operation Returned	Class or instance	Operation name	Break when an instance returns from executing a user-defined operation.
Event Sent	Class or instance	Event name	Break when an instance sends an event.
Event Received	Class or instance	Event name	Break when an instance receives an event.

Enabling and Disabling Breakpoints

The first column of the breakpoints list shows the status of all breakpoints. By default, new breakpoints are enabled. The enabled breakpoints are at the top of the list, whereas disabled breakpoints are at the bottom.

To disable a breakpoint, select an enabled breakpoint and click **Disable**. The breakpoint is disabled and moved to the bottom of the list.

To enable a breakpoint, select a disabled breakpoint and click **Enable**. The breakpoint is enabled and moved below the last enabled breakpoint and above the first disabled breakpoint in the list.

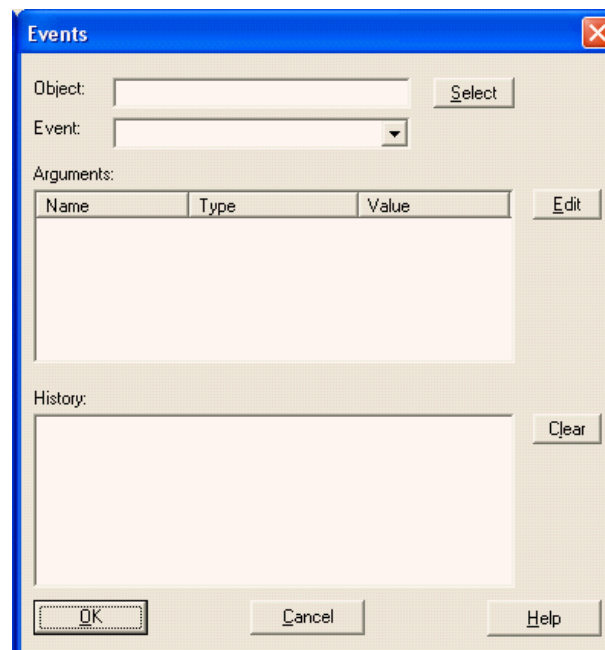
Deleting Breakpoints

To delete a breakpoint, select the breakpoint and click **Delete**. The breakpoint is removed from the list.

Event Generator

You can generate events to inject into the model using either the **Event Generator** tool or the Animation Command bar (see [Generating Events Using the Animation Command Bar](#)).

To generate events using the Event Generator, click the **Event Generator** tool in the **Animation** toolbar. The Events dialog box opens, as shown in the following figure:



This dialog box enables you to select an object as the target of the event and define the event you want to send to that object. If events have previously been generated during the same animation session, those events appear in the Events History list and you can simply select an event from the history list.

1. In the Events dialog box, click **Select**. The Instances Selection dialog box opens (see [Defining Breakpoints](#)).
2. Select an instance from the list, then click **OK**.
3. Click the down-arrow to right of the **Event** box to display a list of events that the object is capable of receiving and select an event from the list.

If the event takes arguments, they are displayed in the arguments list.

4. You must assign actual values to arguments to successfully generate events with arguments. To assign a value to an argument, select the argument from the **Arguments** list, then click **Edit**. The Argument Value dialog box opens.
5. Enter a value for the argument, then click **OK**. The value is displayed next to the argument in the Events dialog box.
6. Click **OK** to apply your changes and close the dialog box.

Events History List

Successfully generated events are stored in an Events History list that is displayed in the Events dialog box.

Every time you save a project, events are stored in the history list, along with active and inactive breakpoints, to a file named <projectname>.ehl in the project directory. The following is an example of an .ehl file:

```
[Events]
A[0] ->GEN(evCount())`Default::A
A[0] ->GEN(evOn())`A
A[0] ->GEN(evDeep())`Default::A
A[0] ->GEN(evOn())`Default::A
[Active Breakpoints]
[Inactive Breakpoints]
```

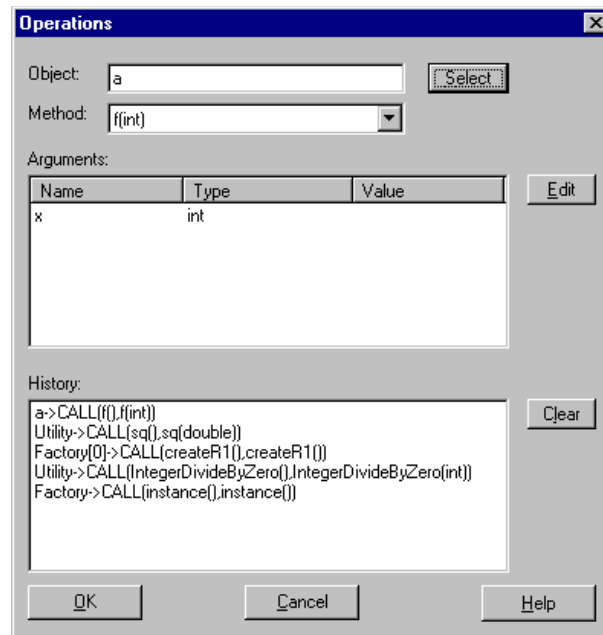
To resend an event from the events history list, follow these steps:

1. In the Events dialog box, select an event from the Events History list.
2. Click **OK** The event is entered into the Event Queue.

To clear the events history list, click **Clear**.

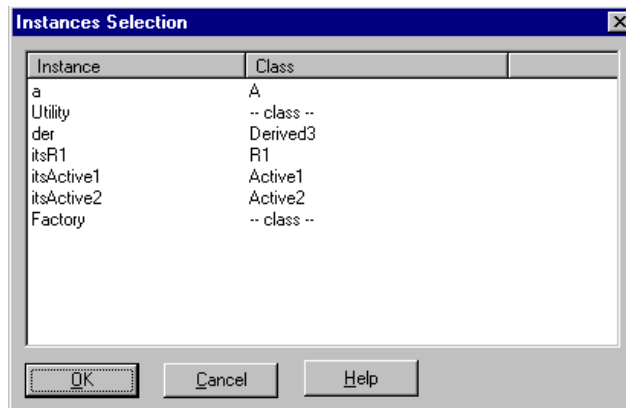
Calling Animation Operations

Using Rhapsody in C++ and Rhapsody in C, you can invoke operation calls during animation and tracing to validate parts of the design model. To call an operation, it must be instrumented; by default, instrumentation is set to None (you cannot call operations). To enable operation calls, either set the property `<lang>_CG::Operation::AnimAllowInvocation` or use the Advanced Instrumentation Settings dialog box (opened by clicking the **Advanced** button on the **Settings** tab for the configuration). To call an operation, click the **Call operations** tool in the **Animation** toolbar. The Operations dialog box opens, as shown in the following figure:



This dialog box contains the following fields:

- ◆ **Object**—Specifies the object (or class) that contains the method you want to invoke. Click **Select** to open the Instances Selection dialog box, shown in the following figure:



This dialog box displays the classes and instances that have operations that you can call.

Select the appropriate instance, then click **OK**. You return to the Operations dialog box.

- ◆ **Method**—Specifies the operation to call.

If you selected a class in the Instances Selection dialog box, only static methods instrumented for operation calls are listed in the Operation dialog box. The Operation dialog box displays all the available operations for invocation, including the ones specified in the base classes.

- ◆ **Arguments**—Displays the arguments used by the specified operation. If necessary, click **Edit** to modify the operation's arguments.

To invoke an operation, all the arguments must be animated. If the data type is complex, you must specify the unserialization routine for the type and force its instrumentation.

- ◆ **History**—Shows the history of all the operations called in the animation session. As with events, the history of operation calls is stored in a log file.

Click **Clear** to delete the history.

Once you have set the appropriate values, click **OK** to invoke the method call. The results are displayed in the output window (animation) or console (tracing).

Scheduling and Threading Issues

The method is invoked by the application thread currently in focus. The workflow is as follows:

1. If needed, set the focus to be the thread that should execute the operation.
2. Send the command to the animator. If several commands are sent to the same thread, the application will execute them one after the other in the calling order. In addition, you can switch focus threads and send invoke requests to different threads to simulate concurrent calls.
3. Continue execution of the application (by one of the “go” commands).
4. The operation is displayed in the callstack of the thread and relevant sequence diagrams, and is carried out.
5. Once the operation returns, its return value is displayed in the output window (animation) or in the console (tracing).

Note

The operation call is *synchronous*—the thread executes the operation, then returns to its last previous position in the interrupted control flow.

Using Partial Animation

By default, operations are not instrumented for calls. To enable operation calls, set the property `<lang>_CG::Operation::AnimAllowInvocation`. Refer to the property definitions in the Properties tab or the List of Books for more information.

Note that if an operation is not animated (either the operation’s `Animate` property is `FALSE`, or the class of the operation is not in the configuration’s instrumentation scope), instrumentation for the invocation is disabled.

Restrictions

Note the following restrictions and limitations:

- ◆ Only an animated Rhapsody (non-foreign) thread can call an operation.
- ◆ You can invoke an operation only if *all* its argument types are animated and have unserialization routines (as with events).
- ◆ The following aspects are not supported:
 - Constructor and destructor invocations
 - Global functions

- Overloaded operators (such as ++, <<, and so on)
 - Templates
- ◆ You must specify all arguments (default arguments are not supported).
- ◆ Once the operation actually starts, the sent request is shown (not the message) and the operation is shown in the callstack.

Animation Modes

Silent mode enables you to perform design-level debugging and display animation information only at breakpoints. In contrast, Watch mode enables you to continually update animation information in normal step-by-step operation via the **Go** buttons. You can toggle between the two modes at any break, or when the executable is idle, using the **Silent/Watch** tool in the **Animation** toolbar.

Note

Watch mode is the default animation mode. If you are in Watch mode, the text “Watch - Display Continuous Update” is displayed in the tooltip when you move the cursor over the Silent/Watch tool. If you are in Silent mode, the text “Display on Breakpoint Only” is displayed.

Silent Mode

In Silent mode (also known as Update-on-Break mode), the model runs at near production speeds and the animated views are not updated until you hit a breakpoint. Execution speeds can be up to 100 times faster in Silent mode than in Watch mode.

To activate Silent mode, follow these steps:

1. When the model is idle and in Watch mode, click the **Silent** tool. The animated views are immediately updated, but the event queue is not.
2. To update the event queue after a break, click the **Command Prompt** tool.
3. In the Animation Command bar, type refresh and press **Enter**.

Alternatively, you can use **View > Refresh** (or press F5). The next time the model reaches a breakpoint, the event queue immediately updates.

Watch Mode

Watch mode is the normal mode of operation in which all animated views and the event queue are continually updated with each **Go**. Watch mode is preferable for unit testing or pinpoint debugging.

To activate Watch mode, click the **Watch** tool when the model is idle and in Silent mode.

Viewing the Model

This section describes how to view and inspect components of a model during animation. The ability to inspect an executable model is a key feature of animation and a major debugging aid during model design.

You can watch an application's active execution in any of the following views:

- ◆ Output window
- ◆ Call Stack window
- ◆ Event Queue window
- ◆ Animated browser
- ◆ Animated sequence diagrams
- ◆ Animated statecharts
- ◆ Animated activity diagrams
- ◆ Thread view (multithreaded applications only)

During animation, you can access:

- ◆ The model as a whole
- ◆ Objects that make up the model and relations between them
- ◆ Internal information about specific objects

Accordingly, you are provided with three types of views:

- ◆ Application-wide status views, available only in animation:
 - Call stack view
 - Event queue view
 - Threads view
- ◆ Multiobject views provided by animated versions of multiobject design tools:
 - Animated sequence diagrams, which depict messages actually passed between instances during the execution of the application.
 - The animated browser, which enables you to inspect instances currently alive in the application
- ◆ Object-specific view provided by the animated version of single-object design tools:
 - Animated statecharts, which describe the current states and latest transitions of the object
 - Animated activity diagrams

Call Stack

The call stack view describes the current stack of calls for the focus thread. To invoke the call stack view, select **View > Call Stack**.

- ◆ `startBehavior`—Initiates the behavior of a reactive object
- ◆ `takeEvent`—Initiates the response of a reactive object to the reception of events

For C++ and Java, each line in the call stack view depicts a single function using the following member pointer notation:

```
<instance>-><operation>
```

If the operation does not belong to any particular instance (for example, top-level function calls) or is a constructor, only the operation name is displayed. Operations are added and removed from the stack in LIFO (last-in, first-out) order. The most recent operation is always pushed onto and popped off the top of the stack.

Event Queue

The event queue view describes the current state of the event queue for the focus thread. To invoke the event queue view, select **View > Event Queue**.

Each line in the display depicts a single event in the format:

```
<name of event destination> -> <event name> (<parameters>=  
value{,<parameter>=value})
```

For example:

```
A[1]->Start(priority=3)  
B[3]->NewGame(score = (5,0), time = 3)
```

The top-most event is the next to be dispatched.

Animated Browser

During animation, the browser displays *instances*—objects instantiated—for each class participating in the execution. Typically, instances are deep blue in color. However, an instance that currently has control (is currently executing) is light blue. Selecting an instance in the animated browser displays a dialog that shows:

- ◆ A list of its attributes for the instance and their current values.
- ◆ A list of its relations. Selecting a relation displays a list of the items in this relation.

From the browser, you can invoke an animated statechart for an instance (see [Animated Statecharts](#)).

Animated Sequence Diagrams

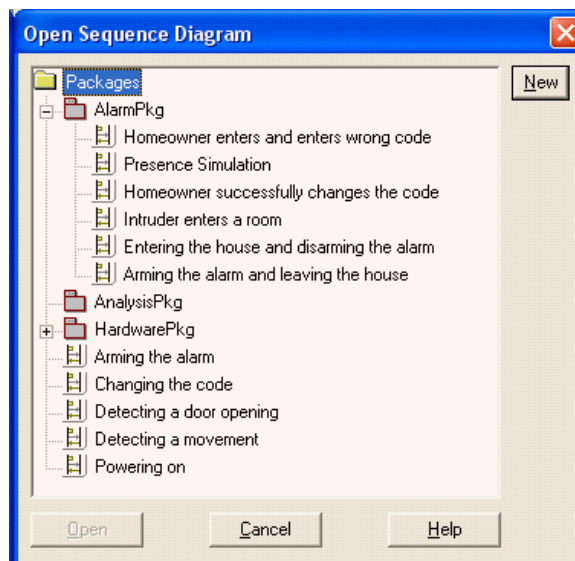
A sequence diagram displays the passing of messages between instances that participate in the chart. Sequence diagrams are a concise and popular way to represent the communication between interacting objects.

Opening Animated Sequence Diagrams

To open an animated sequence diagram (ASD), use either of the following methods.

- ◆ With a nonanimated sequence diagram already open in Rhapsody, start the model running. When the model starts executing, an animated version of the currently open sequence diagram opens along with the original, nonanimated version.
- ◆ Select **Tools > Animated Sequence Diagram**.

The Open Sequence Diagram dialog box opens (as shown in the following figure), listing the nonanimated sequence diagrams that currently exist in the model. Select a nonanimated sequence diagram from the list. The animated version opens in a new window.



Adding and Deleting Instance Lines

To add an instance line to an ASD, create an instance line using the **Instance Line** tool and name the instance, or drag an instance from the browser.

Each line has a label. If the label refers to an instance that currently exists in the executing model, the line is connected to that object.

To delete an instance line, do one of the following:

- ◆ Click the **Delete** button.
- ◆ Press **Ctrl+Delete**.

Auto-creating Animated Instances

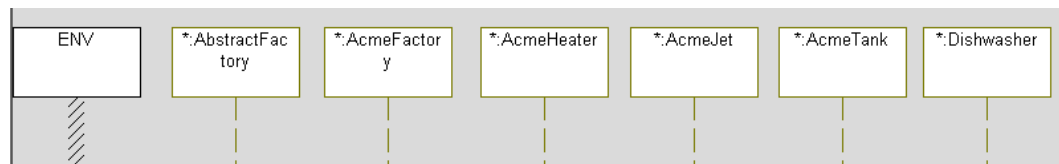
You can make it so that animated instance lines on sequence diagrams are auto-created so that you can see the run-time instance appear in the animated sequence diagram when they are actually created. (Typically during an animation session, you have to drag the created instance from the Rhapsody browser onto the animated sequence diagram to see the operation of that instance getting called.) However, with added notation to an instance line name on a sequence diagram, you can have Rhapsody auto-create the animated instances when you run animation. This capability means that you can mark a specific class to auto-create any sequence diagram instances at run time on the animated sequence diagram.

To auto-create animated instances for a sequence diagram, follow these steps:

1. Make sure you have the active configuration set for Animation. See [Setting the Instrumentation Mode](#).
2. Create a sequence diagram and make sure it is open (or open a current sequence diagram for which you want to auto-create animated instances).

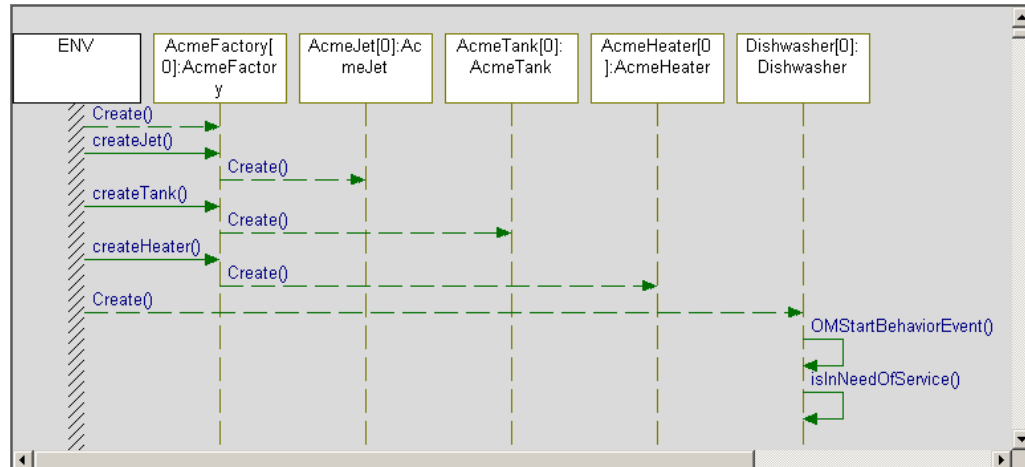
Note: You can set the `SequenceDiagram::General::AutoLaunchAnimation` property to `Always` to make the diagram open automatically when animation starts.

3. Depending on what you did in the previous step:
 - From the Rhapsody browser, drag a class that you want to auto-create instances for on your sequence diagram and add an asterisk (*) to the beginning of the name. For example: `*:Dishwasher`. Or,
 - On the diagram, change the name of an instance by adding an asterisk (*) to the beginning of the name. To do this, click the name to focus the pointer on it. Once the name is highlighted, use your keyboard arrow keys or the mouse to position your mouse pointer to the beginning of the name and add * to it. For example: `*:Dishwasher`, as shown in the following figure:



4. On the **Code** toolbar, click the GMR button  generate, make, and run your model.
5. On the **Animation** toolbar, click the Go button  to start the animation session.

6. Notice that Rhapsody creates an animated sequence diagram that has all auto-generated instances of type `<class_name>`, as shown in the following figure:



Limitations

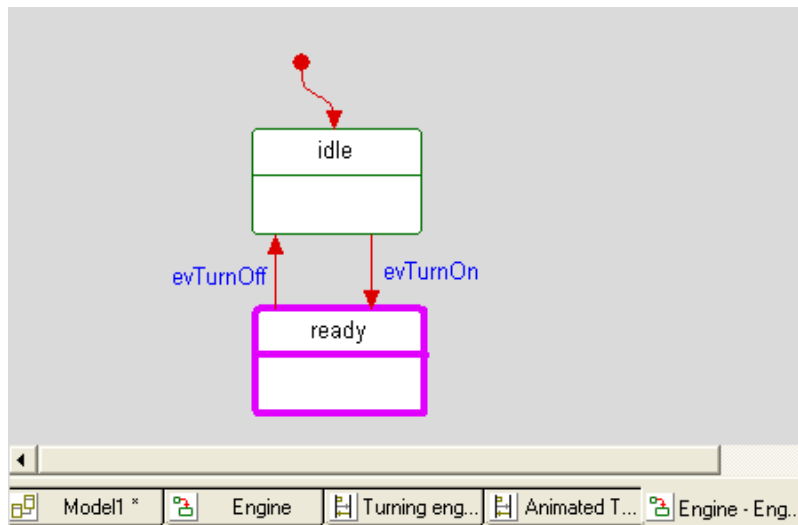
Note the following limitations:

- ◆ There is no auto-creation of derived classes.
- ◆ This feature is not available for Rhapsody in Ada.

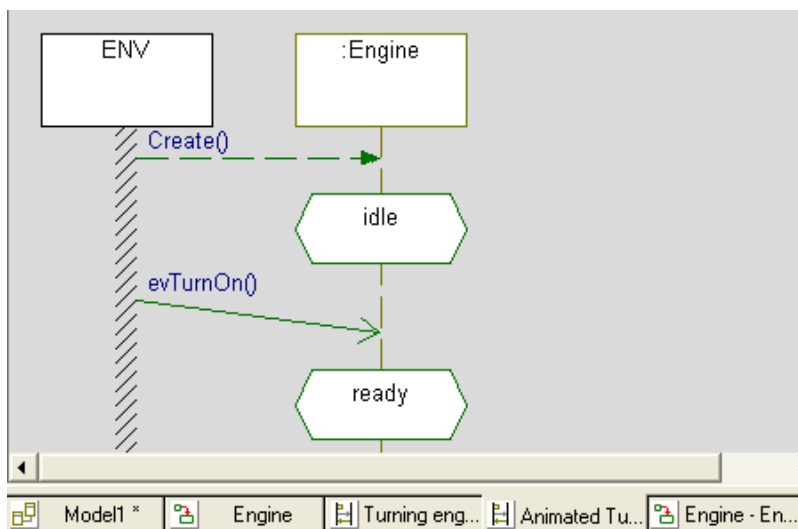
Showing State Transitions in Animated Sequence Diagrams

An animated sequence diagram can cross reference an animated statechart by showing the event of an object entering a state. In Rhapsody, the Condition Mark indicates that an object is in a certain condition or state at a particular point in a sequence. For more information about condition marks, see [Creating a Condition Mark](#).

The following figure shows an animated statechart:



The following figure shows the animated sequence diagram. Notice the transition states (for example, **idle**).



Display Considerations

The following display considerations apply for showing state transitions in animated sequence diagrams:

- ◆ States are displayed in the order of their notifications.
- ◆ When an inner state changes, all of its and-states are listed again, even if unchanged.
- ◆ When re-entering the same state in a self loop, the entry displays again.
- ◆ And-states are listed with a pipeline character (|) in-between.
- ◆ In the case of sub-states, only the most inner current state is listed.
- ◆ The `SequenceDiagram::General::ShowAnimStateMark` property determines whether or not state transitions are displayed in an animated sequence diagram. Its default is `Checked`.

Limitations

The following limitations apply for showing state transitions in animated sequence diagrams:

- ◆ Condition marks are not realized with model states.
- ◆ The DiffMerge tool and Sequence Diagram Compare tool do not support condition marks. For more information about the DiffMerge tool, see [Parallel Project Development](#) and the *Rhapsody Team Collaboration Guide*. For more information about the Sequence Diagram Compare tool, see [Sequence Comparison](#).

The System Border

The system border, if present, is connected to all instances participating in the execution that do not have a special line of their own. In other words, a message is drawn between two instance lines if both the sending and receiving instances have instance lines in the ASD.

If either the sending or receiving instance does not have a line, messages can be displayed between the instance that has a line and the system border:

- ◆ If the ASD includes a system border, the message is drawn between the instance that has a line and the system border.
- ◆ If the ASD does not include a system border, the message is not displayed in the diagram. In other words, removing the system border prevents the display of messages to or from instances that are not explicitly part of the execution sequence.

Messages

ASDs automatically create the following message arrows while the model is executing:

- ◆ Operation calls
- ◆ Self-operation calls (from an instance to itself)
- ◆ Events sent, but not yet received
- ◆ Events sent and received
- ◆ Timeouts sent (matured), but not yet received
- ◆ Timeouts sent and received
- ◆ Constructors
- ◆ Destructors

The y-axis of a sequence diagram indicates both flow of time and steps. No scale is given on this axis, but synchronization is maintained.

When you remove an instance line, all messages sent to and from the instance are also removed. Future arrows relating to the removed instance will be associated with the system border, if the diagram has one.

When you add an instance line, messages to or from the newly added instance are displayed only from the moment the line is added. Messages sent or received by an instance before it was added to the diagram are associated with the system border.

Note

Do not create more than one line referring to the same instance. Otherwise, message arrows will connect to only one of these instance lines in an arbitrary manner.

When you quit animation, you can either save or delete animated sequence diagrams that have been generated by the execution. Saving them creates a record of the execution.

Limiting Message Display in Animated Sequence Diagrams

When running an animation of a sequence diagram containing a very large number of messages, your system may run low on virtual memory. The properties

`SequenceDiagram::General::MaxNumberOfAnimMessages` and

`SequenceDiagram::General::OnReachedMaxAnimMessages` can be used to prevent such problems by limiting the number of messages displayed at any one time.

The property `MaxNumberOfAnimMessages` is used to specify the maximum number of messages that should be displayed in the sequence diagram at any one time during animation.

The property `OnReachedMaxAnimMessages` determines how Rhapsody should behave when the maximum number of messages has been reached. The property can take the following values:

- ◆ `Stop`—Rhapsody stops displaying animated messages in the diagram after the maximum number has been reached.
- ◆ `KeepLast`—After the maximum number of messages specified has been reached, Rhapsody erases the first messages displayed. It will continue erasing displayed messages in this manner so that the number of messages displayed on the diagram at any one time does not exceed the maximum specified.

Suppressing Animated Sequence Diagram Messages

To control the appearance of Create, Destroy, Timeout, and Cancel Timeout messages in an animated sequence diagram, use the following properties in `SequenceDiagram::General`:

- ◆ `ShowAnimCreateArrow`. This property specifies whether to show Create messages in animated sequence diagrams.
- ◆ `ShowAnimDataFlowArrow`. This property specifies whether to show dataflow messages in animated sequence diagrams.
- ◆ `ShowAnimDestroyArrow`. This property specifies whether to show Destroy messages in animated sequence diagrams.
- ◆ `ShowAnimTimeoutArrow`. This property specifies whether to show Timeout messages in animated sequence diagrams.
- ◆ `ShowAnimCancelTimeoutArrow`. This property specifies whether to show Cancel Timeout messages in animated sequence diagrams.

By default these properties are set to `Checked` (so that these messages appear in animated sequence diagrams). To suppress these messages, set the above properties to `Cleared`.

Note that the sequence diagram cloned during animation will use the above properties also.

Dataflows

Rhapsody animation also uses dataflow arrow notation to represent data flow between flowports. For more information about dataflows and flowports, see [Creating a Dataflow](#) and [Flow Ports](#).

Animating Return Values

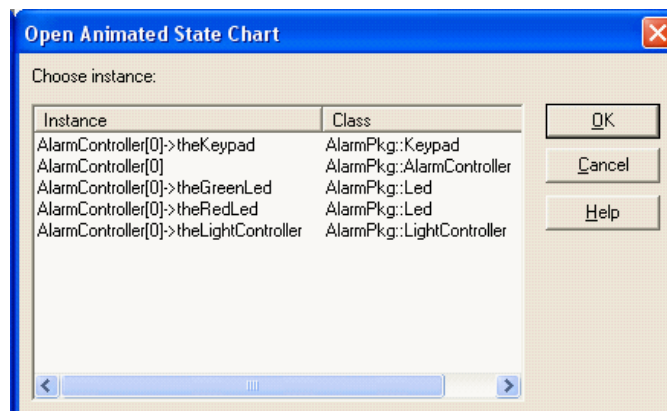
To learn how to instrument return types so that you can see them as reply messages on animated sequence diagrams, see [Animation of an Operation's Return Value](#).

Animated Statecharts

To invoke an animated statechart, right-click an instance in the browser and select **Open Instance Statechart** from the pop-up menu.

Alternatively, follow these steps:

1. Select **Tools > Animated Statechart**. The Open Animated Statechart dialog box opens, as shown in the following figure:



2. Select an existing instance from the list, then click **OK**.

Note that animated statecharts operate in *full behavioral steps*. This means that you see all of the final effects of a behavioral step at once—not one-by-one as they occur.

Animation Highlighting

You can change how animation is displayed by right-clicking the project in the browser and selecting **Format** from the pop-up menu.

The types used for animation highlighting are as follows:

- ◆ `AnimatedTransition`—Specifies how animated transitions are displayed. By default, an animated transition is displayed using olive, 3-point, solid lines.
- ◆ `AnimatedInState`—Specifies how an In state is displayed. By default, an In state is displayed using magenta, 3-point, solid lines.
- ◆ `AnimatedPrevState`—Specifies how the previous state is displayed. By default, the previous state is displayed using olive, 3-point, solid lines.

See [Changing the Format of a Metaclass](#) for detailed instructions on using the Format dialog box.

Instance Names

The building blocks of a model are its classes and relations. The building blocks of the execution of a model are instances of these classes and relations, which are created during execution. This section describes how the animator names these instances.

Names of Class Instances

During execution, the instances of a class *A* are referred to as *A*[0], *A*[1], *A*[2], and so on. The name *A*[0] is given to the first instance of class *A*, the name *A*[1] to the second, and so on.

An instance gets a name only after its construction is completed. An item whose chain of constructors has started but not yet completed is referred to as *in construction*.

An instance retains its name only until its destruction. An item whose chain of destructors has started but not yet completed is referred to as *in destruction*.

An instance that no longer exists is referred to as *non-existent*. This can happen if an instance was deleted, but some other instance still points to it via an attribute or relation.

An instance retains its name, unchanged, during its entire lifetime. For example, an instance assigned the name *A*[5] at its creation continues to be called *A*[5] even if instances *A*[0] to *A*[4] no longer exist.

Names of Component Instances

Instances of a component use the following naming scheme, where the composite is the whole and the component is the part:

`whole[n]->part[m]`

For example, for a component class *A* inside a composite class *B*, the fifth instance of the component *A* is referred to as follows:

`B[3]->itsA[4]`

The name `B[3]->itsA[0]` is assigned to the first instance of class *A* as a component of *B*[3], `itsA[1]` to the second, and so on.

A similar naming scheme is used for relations. In this case, the composite (whole) is considered the source of the relation, and the component (part) is the target.

Because of the chain of construction, a component is typically created first as a class instance and only then as a component. This is reflected in the name of the instance. First, it is created as *A*[*x*] (for example, *A*[4]), and then as *B*[*y*]->itsA[*z*] (for example, *B*[2]->itsA[3]).

Navigation Expressions

Instances can be referred to by the following navigation expressions:

- ♦ If A is a class, $A[\#j]$ denotes the $(j+1)$ th instance of the class currently in existence. For example, $A[\#4]$ can denote the fifth instance of the class. This is consistent with the C/C++ convention of calling the first element $A[0]$.
- ♦ If A is a class, A can denote the first instance of the class. This is the same as $A[\#0]$.

You can use a class name instead of an instance name only in places where there is no ambiguity as to whether it refers to the class or its first instance. For example, $A->GEN(E)$ generates an event E for an instance $A[\#0]$. However, the animation command “Show A relations” displays relation information about class A and *all* of its instances.

- ♦ If B is a name or navigation expression that refers to an instance and that instance has a relation $itsA, B->itsA$ denotes the first element in B 's relation with A and $B->itsA[\#i]$ denotes the $(i+1)$ th element.

The same navigation expression can refer to different instances during the course of the execution. For example, if instances $A[0]$ to $A[5]$ have been created and then $A[3]$ is deleted, the expression $A[\#5]$ refers to $A[4]$ before the deletion and to $A[5]$ after the deletion.

Names of Special Objects

Besides classes and instances, the animator keeps track of a few other special objects such as breakpoints, call stacks, and event queues. You can reference all these from the command prompt using tracer commands. See [Tracing](#) for detailed information about tracer commands.

Animation Scripts

You can create scripts to automate animation sequences using tracer commands. The syntax of these commands is described in detail in [Tracer Commands](#).

Note

To get a list of available scripting commands, type `help` or `?` in the Animation Command bar.

Sample Script

Command Type	Command
Breakpoint	break <object> <op> <breakPointType> <data>
Call	[Object name]->CALL([operation call] [signature])
Comments	// the comment goes here
Display	<ul style="list-style-type: none"> • display • watch
Generate event	<instanceName>- >GEN(<eventName>(<parameterName> [, <parameterName>]*) <instanceName>->GEN(<eventName>()) <instanceName>->GEN(<eventName>)
Go	<ul style="list-style-type: none"> • go • go event • go idle • go step
Help	<ul style="list-style-type: none"> • help • ?
I/O	<ul style="list-style-type: none"> • input [+] <destination> • output <+/-> <destination>
Quit	quit
Resume	<ul style="list-style-type: none"> • resume threadName • resume #Thread threadName
Set focus	<ul style="list-style-type: none"> • set focus <threadName> • set focus #Thread <threadName>
Show	show <object> <interest-list>
Suspend	<ul style="list-style-type: none"> • suspend threadName • suspend #Thread threadName
Time stamp	timestamp <option>
Trace	trace <object> <interest-list>

The following is an example of a script that tests the behavior of a chamber unit in the pacemaker demo:

```
//*****
// file: utChamber.txt
// description: chamber unit test script
//*****// run until we
enter the sensing state
break ut2Chamber->theChamber stateEntered sensing
go
```

```

break ut2Chamber->theChamber -stateEntered sensing

// Trace ... and capture to file utChamber.log
trace #CallStack method
trace #CallStack +timeout
output +test.log

// give several heart beats
ut2Chamber->theChamber->GEN(evHeartBeat)
go idle
ut2Chamber->theChamber->GEN(evHeartBeat)
go idle
ut2Chamber->theChamber->GEN(evHeartBeat)
go idle

// absence of a heartbeat should cause a pace
go idle
break ut2Chamber->theChamber stateEntered sensing
go
break ut2Chamber->theChamber -stateEntered sensing

// regenerate heartbeats
ut2Chamber->theChamber->GEN(evHeartBeat)
go idle
ut2Chamber->theChamber->GEN(evHeartBeat)
go idle
ut2Chamber->theChamber->GEN(evHeartBeat)
go idle

// stop logging to file
output -test.log

```

Running Scripts Automatically

To run a script automatically when an executable starts, follow these steps:

1. Within a component, create a file named `omanipator.cfg`.
2. Give the file the following parameters:
 - ◆ **Path**—Type “..” so the file is generated to a directory one level higher.
 - ◆ **File Type**—Select **Other**.

When animation starts, if Rhapsody finds a file named `omanipator.cfg`, it executes the file. This can be useful for large projects (or those with a GUI), because a script similar to the following could be written and automatically executed:

```

//=====
// Switch off the animation before starting
watch
// Run with animation off to create all objects
go idle
// Switch animation back on
display

// Run continuously
go
//=====

```

Black-Box Animation

Rhapsody includes extended animation for sequence diagrams. Animated instance lines (classifier roles) can represent run-time objects and their internal structure (their parts) as opposed to a single, run-time object (as in previous versions). This enables you to validate higher-level sequence diagrams specified prior to the elaboration of the internal structure of the classes.

Note

By default, animated instance lines in Rhapsody are mapped to single objects. To activate this feature, you must modify the property settings of the instance lines, as described in [Animation Properties](#).

You can map any instance line on a sequence diagram to one of the following:

- ◆ A single object
- ◆ An object and its parts
- ◆ An object and all the objects derived from its reference sequence diagram

In addition, you can specify that during animation, messages sent from the instance line to itself are not displayed.

Animation Properties

Two properties support this functionality:

- ◆ `Animation::ClassifierRole::DisplayMessagesToSelf`—Determines whether messages-to-self are displayed during animation. The possible values are as follows:
 - `None`—Do not display any messages-to-self.
 - `All`—Display all messages-to-self.
- ◆ `Animation::ClassifierRole::MappingPolicy`—Specifies how to map instance lines during animation. The possible values are as follows:
 - `Smart`—Rhapsody decides the mapping policy.

If the instance line has a reference sequence diagram, the mapping will be equivalent to `ObjectAndDerivedFromRefSD`; otherwise, the mapping is equivalent to `ObjectAndItsParts` (which, for an object without any parts, is the same as `SingleObject`).

- `ObjectAndItsParts`—The instance line is mapped to an object and all its parts (recursively), excluding parts that are explicitly shown in the diagram.
- `SingleObject`—The instance line is mapped to a single, run-time object.

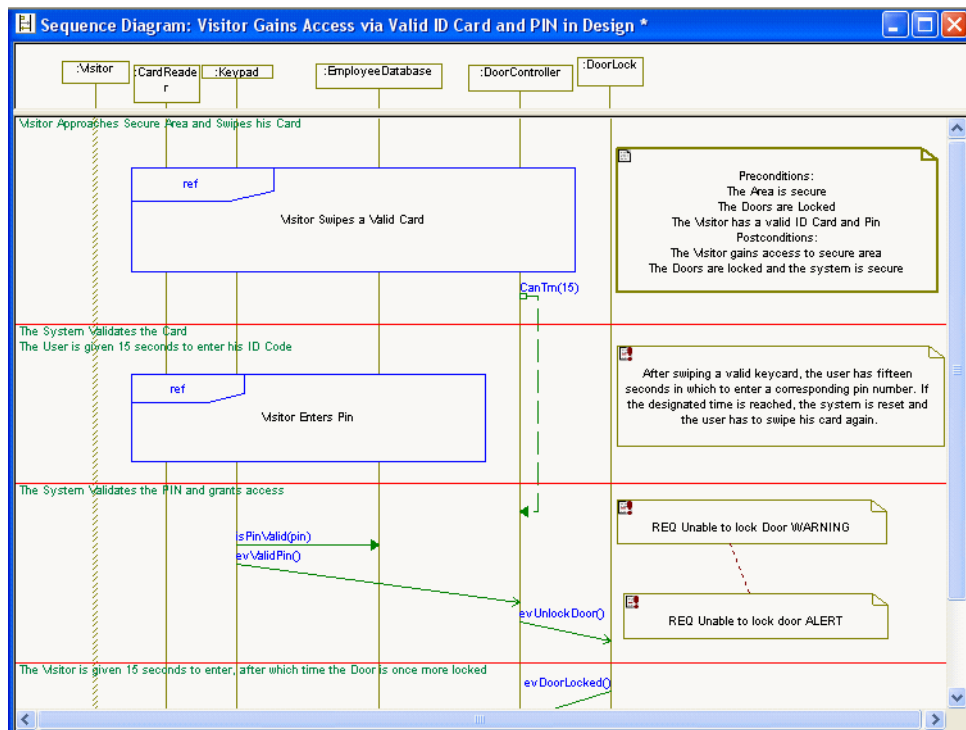
- ObjectAndDerivedFromRefSD—The instance line is mapped to an object by its role name (if it exists) and to all derived objects from the reference SDs (according to their mapping rules).

Note

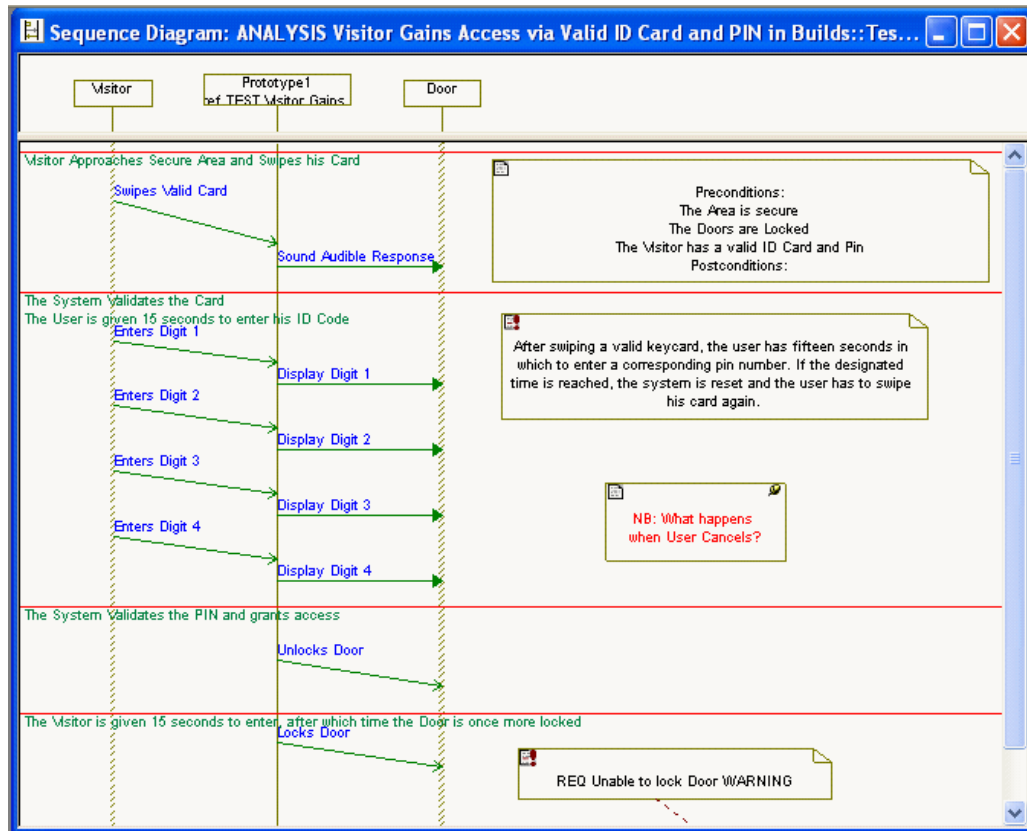
If you change the values of these properties after the sequence diagram has been initialized, the changes will not take effect until you close and reopen the animated SD.

Example

The following figure shows a design sequence diagram that describes a door with keycard access:

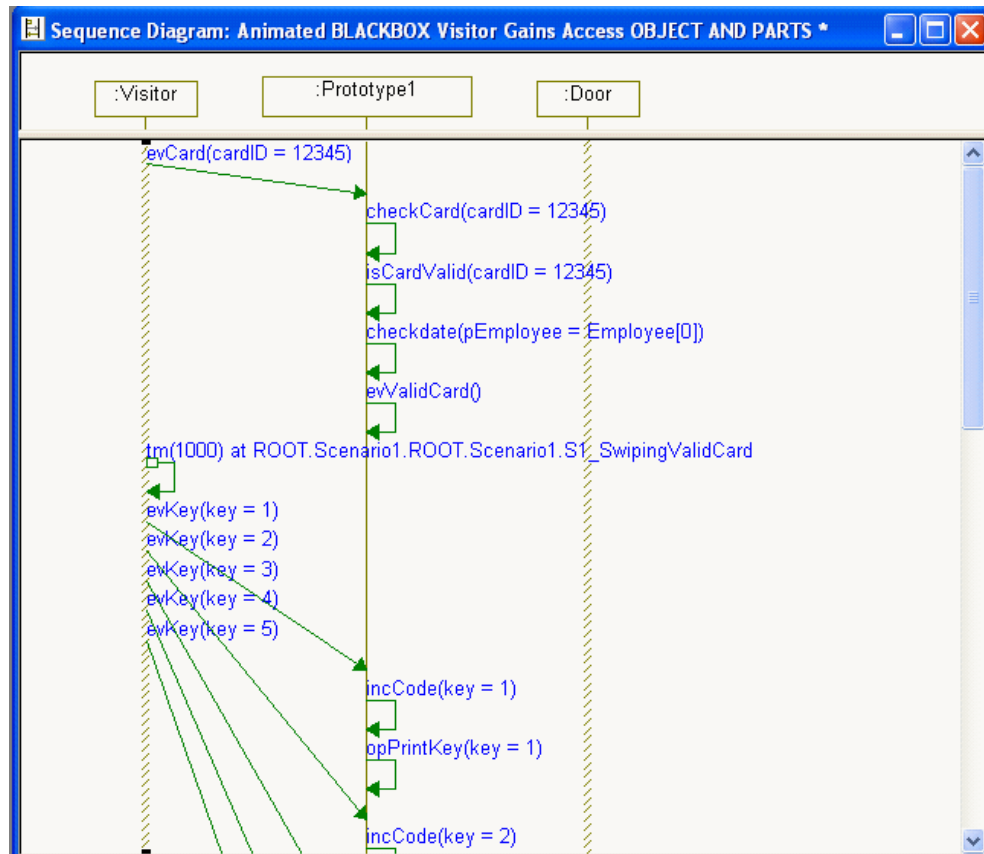


The following figure shows an SD for model analysis:

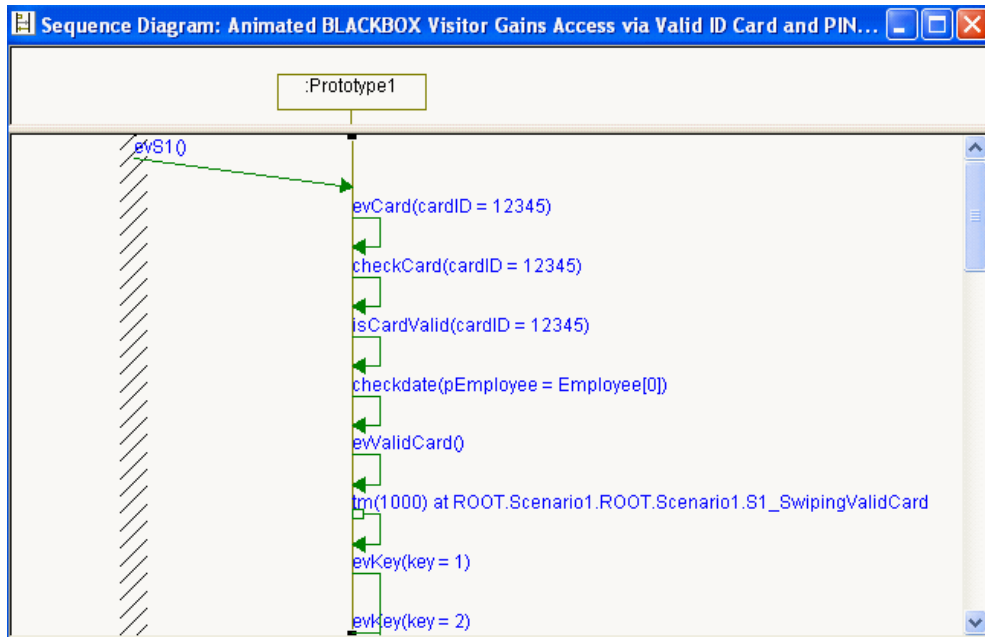


The following figures show some of the ASDs used for black-box testing of the model, and the effects of setting the animation properties to different values.

The following figure shows the resultant ASD when the `Prototype1` instance line uses a mapping policy `ObjectAndItsParts`; the other two instance lines are set to `SingleObject`. Note that the `Prototype1` instance line reflects the messages for the `CardReader`, `Keypad`, `EmployeeDatabase`, and `DoorController` classes.



The following figure shows another view of the model (the instance line uses `ObjectAndItsParts`). Note that messages for other parts of the model are reflected on this instance line because the `Prototype1` instance line can “see” them (because of the policy setting).



Using the Properties for Black-Box Testing

The following scenarios show the effects of the new animation properties on the ASD:

- If you are using an animated sequence diagram (ASD), which is a clone of the existing sequence diagram, the mapping relies on the property settings of the original diagram.
- While animating the model, if you create a new ASD and drag animated objects on it, the mapping is done as if the `MappingPolicy` property is set to `ObjectAndItsParts`. This mapping cannot be changed.
- While still animating the model, if you create a new, non-animated SD and set the `DisplayMessagesToSelf` and `MappingPolicy` properties, then start animating the new SD; the ASD will map its instance lines according to the property settings.

Instance Line Pop-Up Menu

When you right-click a instance line during animation, the pop-up menu contains the following new options:

- ◆ **Open Animated Statechart** or **Open Animated Activity Diagram**—Displays the appropriate animated diagram for selected object.
- ◆ **Generate Event**—Opens the Event dialog box so you can generate an event. See [Event Generator](#) for more information.
- ◆ **Add Breakpoint**—Opens the Breakpoint dialog box, described in [Breakpoints](#).

If the instance line represents more than a single object, these commands are applied to the root object (the object that would have been mapped to the instance line in `SingleObject` mode).

Behavior and Restrictions

Note the following:

- ◆ If any object is added explicitly to a sequence diagram, messages it receives will be shown as being received by it (rather than shown as going to the “owner”).
- ◆ A instance line that is mapped to an object and its parts will not represent parts that have their own instance lines in the same diagram.
- ◆ A single runtime occurrence such as sending a message will be viewed only once in the case of a message relayed to a part via its owner’s ports. For example, if a message is sent to a part via the port of its owner, the message to a instance line that represents the owner and its parts is shown once. In other words, the fact that the message was relayed via the owner’s port is not displayed because logically this is a single occurrence.

Animation Hints

The following sections provide some hints on some of the fine points of using the animator.

Exception Handling

Exceptions can be thrown within operations and caught within operations. However, if you want to affect some set of objects' state machines, you should catch the exception and GEN events for the appropriate objects and rethrow the exception, if necessary (for example, if you want the exception to be resolved by an operation higher in the call tree).

If Animation and Application are Out of Sync

Rhapsody assumes that you are following a certain programming style, outlined in the following notes. You are not forced to follow this style, but if you choose not to, be aware that the animation may get out of sync with the model. For example, at times, it may be convenient to define a static attribute and use it directly for all class instances. Although it is not the most effective programming approach, it is a quick way to solve a number of problems. You can work this way if you prefer. However, be aware that the value of that attribute might not be updated properly during animation.

The following are standard style guidelines:

- ◆ All internals of an object are private or protected; that is, other objects cannot change an object's attributes, relations, or states directly. They must use some operation of the object.
- ◆ States and relations can be changed only through a predefined set of mutators.
- ◆ Invoking self-triggered operations is allowed only between and-state components.

Note

On recovery, if you do not follow these recommendations and suspect that a view of a given object is inconsistent with its actual state, try closing the suspect view and reopening it. This should refresh the view and synchronize it with the actual state of the object.

Passing Complex Parameters

If you pass complex parameters (such as `structs`) and use animation, you must override the `>>` operator. Otherwise, Rhapsody generates compilation errors.

Combining Animation Settings in the Same Model

It is possible to build libraries with animation on for part of an application, with animation off for another part, and then link both parts (the different libraries) into a single executable.

In Rhapsody in C, the architecture was changed from a user object *being* an animation object to a user object being *associated* with an animation object. As a result, the memory layout of animated and nonanimated objects is the same so, in principle, they can mix. Each class or object type is either completely instrumented or completely noninstrumented.

To create a combined application, you can link:

- ◆ Some instrumented user code
- ◆ Some noninstrumented user code
- ◆ Instrumented framework libraries (`oxfinst.lib`, `aomanim.lib`, and so on)

When some user object calls a user-defined method, the animation recognizes this, as the framework and the call stack are animated. The animation looks in a table for the animation associate of the user object (the `me` parameter in the method call). If it finds one, an animation message is sent to Rhapsody with respect to this action. Otherwise, it ignores this action (the action is taken, but not animated).

Limitations

The animation feature cannot provide animation for classes that are nested in template classes.

Guidelines for Writing Serialization Functions

When you define a complex type, it is not understood by the Rhapsody animator and therefore cannot be animated. To write serialization functions in order to animate such types, you can use the following properties:

- ◆ `<lang>_CG::Type::AnimSerializeOperation`
- ◆ `<lang>_CG::Type::AnimUnserializeOperation`

AnimSerializeOperation

The `AnimSerializeOperation` property enables you to specify the name of an external function used to animate all attributes and arguments that are of that type. Compare with [AnimUnserializeOperation](#).

Rhapsody can animate (display) the values of simple types and one-dimensional arrays without any problem. To display the current values of such attributes during an animation session, invoke the Features dialog box for the instance.

However, if you want to animate a more complex type, such as a date, the type must be converted to a string (`char *`) for Rhapsody to display it. This is generally done by writing a global function, an *instrumentation function*, that takes one argument of the type you want to display, and returns a `char *`. You must disable animation of the instrumentation function itself (using the `Animate` and `AnimateArguments` properties for the function).

For example, you can have a type `tDate`, defined as follows:

```
typedef struct date {
    int day;
    int month;
    int year; } %s;
```

You can have an object with an attribute count of type `int`, and an attribute date of type `tDate`. The object can have an initializer with the following body:

```
me->date.month = 5;
me->date.day = 12;
me->date.year = 2000;
```

If you want to animate the date attribute, the `AnimSerializeOperation` property for `date` must be set to the name of a function that will convert the type `tDate` to `char *`. For example, you can set the property to a function named `showDate`. This function name must be entered without any parentheses. It must take an attribute of type `tDate` and return a `char *`. The `Animate` and `AnimateArguments` properties for the `showDate` function must be set to `Cleared`.

The implementation of the `showDate` function might be as follows:

```
showDate(tDate aDate) {
    char* buff;
    buff = (char*) malloc(sizeof(char) * 20);
    sprintf(buff, "%d %d %d",
            aDate.month, aDate.day, aDate.year);
    return buff;
}
```

When you run this model with animation, instances of this object will display a value of 5 12 2000 for the date attribute in the browser.

If the `showDate` function is defined in the same class that the attribute belongs to and the function is not static, the `AnimSerializeOperation` property value should be similar to the following:

```
myReal->showDate
```

This value shows that the function is called from the `serializeAttributes` function, located in the class `OMAnimated<classname>`.

Note

The `showDate` function *must* allocate memory for the returned string via the `malloc/alloc/calloc` function in C, or the `new` operator in C++. Otherwise, the system will crash.

The default for this property is an empty string.

AnimUnserializeOperation

The AnimUnserializeOperation property converts a string to the value of an element (the opposite of the [AnimSerializeOperation](#) property). Unserialize functions are used for event generation or operation invocation using the **Animation** toolbar to convert the string (received from the user) to the value of the event or operation before the event generation or operation invocation.

For example, your serialization operation might look similar to the following:

```
char* myX2String(const Rec &f)
{
    char* cS = new char[OutputStringLength];
    /* conversion from the Rec type to string */
    return (cS);
}
```

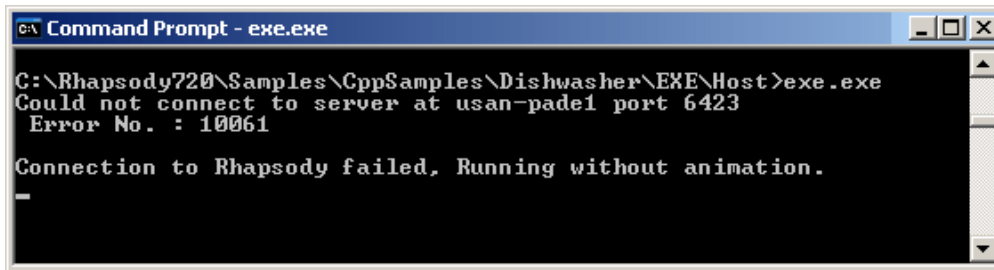
The unserialization operation would be:

```
Rec myString2X (char* C, Rec& T)
{
    T = new Trc;
    /* conversion of the string C to the Rec type */
    delete C;
    return (T);
}
```

The default for this property is an empty string.

Running an Animated Application Without Rhapsody

An animated Rhapsody application will always try to connect to Rhapsody. However, if the application cannot connect to Rhapsody, you can still run it outside of Rhapsody via the command line, as shown in the following figure:



```
C:\Rhapsody720\Samples\CppSamples\Dishwasher\EXE\Host>exe.exe
Could not connect to server at usan-pade1 port 6423
Error No. : 10061

Connection to Rhapsody failed. Running without animation.
-
```

In addition, you can use the `-noanim` flag, as shown in the following figure, to run the application without animation even though Rhapsody is on.



```
C:\Rhapsody720\Samples\CppSamples\Dishwasher\EXE\Host>exe.exe -noanim
-
```


Tracing

The tracer is a stand-alone text version of the animator. In the tracer, you type commands at a command prompt and receive messages detailing the status of the model as it executes.

Note

Tracing is not supported in code generated for Windows CE™.

You can use the tracer from within Rhapsody, or as a stand-alone application outside of Rhapsody.

Tracer Capabilities

Tracing enables you to monitor and control the application without having the Rhapsody GUI in the loop by providing a text-based, console-like application. However, all tracing commands are also available in animation.

Using tracing, you can:

- ◆ Inspect and trace the status of the executing application:
 - Inspect the application via show commands.
 - Identify items to trace with trace commands.
- ◆ Set and remove breakpoints with break commands.
- ◆ Generate events or call operations with the GEN macro or CALL command (CALL applies to Rhapsody in C++ only).
- ◆ Return control to the Tracer for one or more steps using the go commands.

The tracer:

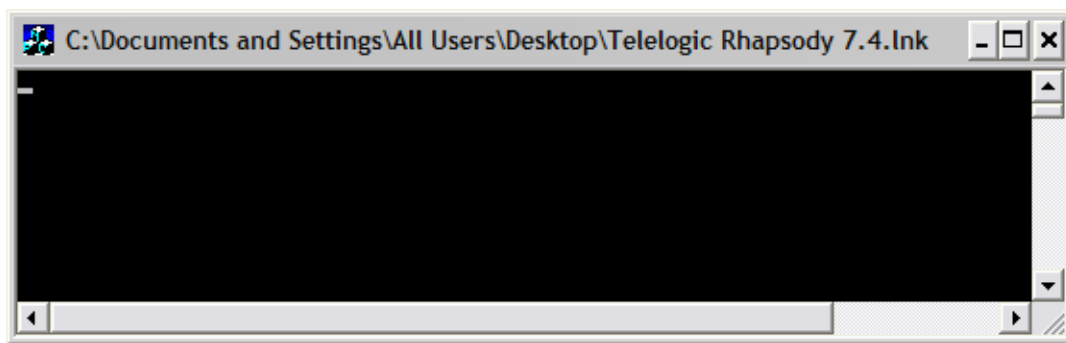
- ◆ Advances execution according to the go commands or until a breakpoint occurs.
- ◆ Displays messages describing what happens to traced objects as the model executes.

Starting a Trace Session

To prepare your application for tracing, follow these steps:

1. Select a configuration, right-click, and select **Features**.
2. In the Settings tab, set the Instrumentation Mode to **Tracing** and click **OK**.
3. In the browser, right-click the configuration and select **Set as Active Configuration**.
4. To generate code for the configuration, select **Code > Generate > <configuration>**. The generation messages are displayed in the Log tab of the Output window.
5. To build the component, select **Code > Build <configuration>**. Build messages are displayed in the Output window.
6. To run the component, select **Code > Run <configuration>**.

A command prompt window opens ready for you to enter a tracer command.



Controlling Tracer Operation

Use tracer commands to control the tracer's operation. The tracer reports on the status of instances as the application executes. Enter tracer commands at the command prompt when using the tracer as a stand-alone application or in the Animation Command Bar during animation. The tracer displays a variety of messages depending on the commands that you enter.

Accessing Tracer Commands

To see a brief description of tracer commands, type `help` at the command-line. See also [Tracer Commands](#).

To place comments in a tracer command, precede the comment text with two forward slashes. The portion of the line from the slashes to the end of the line is considered a comment. For example:

```
trace B[5] relations // Displays a message whenever a
                    // relation of B[5] is modified.
```

Using a File to Enter Tracer Commands

Rather than typing commands at the command prompt, you can give the tracer commands from an input file.

By default, the tracer looks for commands in a file named `OMTracer.cfg` in the configuration directory. If the file does not exist, or the tracer has reached the end of the file, the tracer looks for commands entered at the prompt. You can specify a different input file using the `input` command (see [input](#)).

By default, the animator looks for tracing commands in the `OMAnimator.cfg` file. It is stored in the same directory as the project file.

Note

Because the tracer generates numerous messages, you might want to use the `output` command to send results to a file (see [output](#)).

To send commands to the tracer using the default input file, follow these steps:

1. Create a text file that contains each tracer command as you would type it at the command prompt, in the order that you want the commands to execute.
2. Save the file in the configuration directory under the name `OMTracer.cfg`.
3. Generate, make, and run your application using the **Tracing** instrumentation mode.

To send tracer commands to the animator using an input file, follow these steps:

1. Create a text file that contains each tracer command as you would type it at the command prompt, in the order that you want the commands to execute.
2. Save the file in the project directory under the name `OMAnimator.cfg`.
3. Generate, make, and run your application with **Animation** instrumentation.

Using Threads

During a tracing session, you can suspend, resume, or set focus on a thread.

Each application starts with a single thread named `mainThread`.

Whenever an instance of an active class is created, a new thread is created with it. During construction of the instance, the name of the thread is `@<in construction>`; from then on, the name is `@<instanceName>`.

When an instance of an active class is deleted, its thread dies with it and cannot be referenced anymore.

When you register an external thread, you give it a name that is used to identify it in tracer or animation. This is an advanced feature needed in special cases. Refer to the *Rhapsody C++ Execution Framework Reference Guide (OXF)* for information on when and how to register an external thread.

When an unregistered external thread calls on an instrumented operation, that thread is identified by the handle the operating system gives it. This happens typically if you connect your instrumented code with a GUI engine that does not use events. (This is an advanced feature needed in special cases.)

For more information, see [Notes on Multiple Threads](#).

Tracer Commands

The following sections document the tracer commands. For ease-of-use, the commands are presented in alphabetical order.

The commands are as follows:

- ◆ break
- ◆ CALL
- ◆ display
- ◆ GEN
- ◆ go
- ◆ help
- ◆ input
- ◆ LogCmd
- ◆ output
- ◆ quit
- ◆ resume
- ◆ set focus
- ◆ show
- ◆ suspend
- ◆ timestamp
- ◆ trace
- ◆ watch

break

Description

The `break` command enables you to add or remove a breakpoint on a given occurrence.

Syntax

```
break <object> <op> <breakPointType> <data>
```

Arguments

object

Specifies the object. This must be #All, a valid class name, or a valid instance name.

Setting a breakpoint on a class implies setting a breakpoint on all its instances and subclasses.

op

Specifies the operation. The possible values are +, -, add, or remove. The default value is add.

breakPointType

Specifies the type of breakpoint. The possible values are as follows:

- ◆ `instanceCreated`—With class only. Breaks when a new instance of this class (or a subclass of it) is created.
- ◆ `instanceDeleted`—Breaks when the instance (an instance of the class) is deleted.
- ◆ `termination`—Breaks when the instance (an instance of the class) reaches a termination connector. Does not break if the instance is deleted in a way other than by entering a termination connector.
- ◆ `stateEntered <state name>`—If a state name is specified, it breaks when the instance (an instance of the class) enters that state. If the state name is omitted, it breaks when instance enters any state.
- ◆ `stateExited <state name>`—If a state name is specified, it breaks when the instance (an instance of the class) exits the given state. If the state name is omitted, it breaks when instance exits any state.
- ◆ `state <state name>`—If a state name is specified, it breaks when the instance (an instance of the class) enters or exits the given state. If the state name is omitted, it breaks when instance exits or enters any state.
- ◆ `relationConnected <relation name>`—If a relation name is specified, it breaks when a new instance is connected to the given relation for this instance (an instance of this class). If the relation name is omitted, it breaks when a new instance is connected to any relation.
- ◆ `relationDisconnected <relation name>`—If a relation name is specified, it breaks when an instance is removed from the given relation for this instance (an instance of this class). If the relation name is omitted, it breaks when an instance is removed from any relation.
- ◆ `relationCleared <relation name>`—If a relation name is specified, it breaks when the given relation for this instance (an instance of this class) is cleared. If the relation name is omitted, it breaks when any relation is cleared.
- ◆ `relation <relation name>`—If a relation name is specified, it breaks when a new instance is connected to the relation, an instance is deleted from the relation, or the

relation is cleared for this instance (an instance of this class). If the relation name is omitted, it breaks when a new instance is connected to any relation, an instance is deleted from any relation, or any relation is cleared

- ◆ `attribute`—Instance only. Breaks when any of the attributes of the given instance changes. When the breakpoint is set, a copy of the attribute values of the instance is stored. When any of the attribute values change with respect to this copy a break occurs. After the break, a copy of the new (modified) values is kept as the reference.
- ◆ `gotControl`—Breaks when the instance (an instance of the class) gets control. This happens when the instance starts executing one of its user-defined operations, the instance responds to an event, or an operation the instance called from another object has finished and now it resumes executing.
- ◆ `lostControl`—Breaks when the instance (an instance of the class) loses control; that is, either it has finished executing an operation and it now returns, it finished responding to an event, or it calls an operation of another object.
- ◆ `operation <operation name>`—If an operation name is specified, it breaks when the instance (an instance of the class) starts executing the named operation. If the operation name is omitted, it breaks when the instance starts executing any of its user-defined operations.
- ◆ `operationReturned <operation name>`—If an operation name is specified, it breaks when the instance (an instance of the class) returns from executing the named operation. If the operation name is omitted, it breaks when the instance returns from executing any of its user-defined operations.
- ◆ `eventSent <event name>`—If an event name is specified, it breaks when the instance (an instance of the class) sends the named event. If the event name is omitted, it breaks when the instance sends any event.
- ◆ `eventReceived <event name>`—If an event name is specified, it breaks when the instance (an instance of the class) receives the named event. If the event name is omitted, it breaks when the instance receives any event.
- ◆ `all`—Indicates all break points. This keyword can be used only to remove all breakpoints.

For example, the following command removes all breakpoints on `B[5]`:

```
break B[5] - all
```

The following command removes all breakpoints from the animation:

```
break #all - all  
data
```

Is context-dependent. See [breakPointType](#). Breakpoints that take data are shown with the data parameter in angle brackets.

Setting a breakpoint on some occurrence causes execution to stop when that occurrence happens. For example, the following command causes execution to stop when `B[2]` enters state `ROOT.S1`:

```
break B[2] stateEntered ROOT.S1
```

Saving Breakpoints

To save breakpoints, write them to a file (for example, `myBreakPoints.cfg`). After you have saved them, you can reinsert them next time you run the application by typing the following command:

```
input myBreakPoints.CFG
```

CALL

Description

The `CALL` command invokes an operation call in animation or tracing. `TestConductor` can use this command to invoke operations.

See [Calling Animation Operations](#) for more information on calling operations during animation.

Syntax

```
[Object name]->CALL([operation call]
[, signature (optional)])
```

Arguments

```
operation call
```

If the operation is static, this is the class name. Otherwise, it is the name of the object that performs the call. The format is as follows:

```
[method name] ([list of argument values])
signature
```

Specifies the signature of the operation. This optional argument is used to distinguish between overloaded functions. For example:

```
a->CALL(f(5, "Hello"), f(int, char*))
```

Examples

```
A[0]->CALL(f(5))
```

Invokes `f(5)` on the object `A[0]`

```
A->CALL(g("Hello, World!"))
```

Invokes `g(char*)` on the object `A`

Notifications

The following table lists sample notifications.

Action	Message Format
The operation will be called when the application resumes.	Message: <code><CALL command></code> sent. For example: Message: <code>Utility->CALL(sq(2))</code> sent.
The operation returned, and there is a return value.	<code><CALL command></code> returned <code><return value></code> For example: <code>Utility->CALL(sq2)</code> returned 1.41421.
No matching operation is found.	Unable to perform <code><CALL command></code> , no matching operation found. For example: Unable to perform <code>a->CALL(f())</code> , no matching operation found.
More than one matching operation is found.	Unable to perform <code><CALL command></code> , more than a single matching operation found. For example: Unable to perform <code>a->CALL(f(5))</code> , more than a single matching operation found. Note that this can happen only if you use the command-line interface and do not specify the signature. If you use the dialog box, this message will never be displayed because the dialog box always fills in the signature.

display

Description

The `display` command switches the animation mode to “silent.”

Syntax

```
display
```

GEN

Description

The `GEN` command enables you to generate an event to an object in the executable. The command can be executed with or without parameters.

Syntax

```
<instanceName>->GEN(<eventName>(<parameterName>  
  [, <parameterName>]*)  
<instanceName>->GEN(<eventName>())  
<instanceName>->GEN(<eventName>)
```

Arguments

```
instanceName
```

Specifies the canonical name of an instance or a navigation expression.

A canonical name can be:

- ◆ The name of the class, if the class is a singleton class (for example, `A`)
- ◆ The name of the class followed by a subscript, if the class has multiple instances (for example, `B[2]`)
- ◆ `instanceName->CompositeRelation` (for example, `B[2]->itsC[3]`)

A navigation expression can be:

- ◆ `ClassName[#multiplicity]` for a non-singleton class (for example, `B[#2]`)
- ◆ `instanceName->Relation[#multiplicity]` (for example, `B[2]->itsC[#3]`)

A canonical name always refers to the same instance. A navigation expression can refer to different instances at different times. For example, `B[#0]` might refer to instance `B[4]` if instances `B[0]` to `B[3]` are deleted.

eventName

Specifies the name of the event to be generated. If the event requires parameters, include them in the `GEN` command.

If an event has parameters, the `GEN` command provides the event with the correct number of parameters and the correct types. For example, to generate an event, `X` where `X` is defined as `X(int, B*, char*)`, and `B` is a class defined in Rhapsody, enter:

```
A[1] ->GEN(X(3, B[5], "now"))  
or  
A[1] ->GEN(X(1, NULL, "later"))
```

When the parameters of an event are not pointers to classes defined in Rhapsody (for example `int`, `char*`, or `userType` (where `userType` is a user-defined type defined outside Rhapsody)), the tracer relies on the C++ operator `>>` (`istream&`) or the template `string2X(T& t)` to interpret the characters you type in correctly. `A[1]->GEN(Y(1))` works because the operator `>>` converts the character `1` to the integer `1`, but `A[1]->GEN(Y(one))` does not work because the operator `>>` does not convert the characters “one” to an integer. Similarly, if you use a type you defined outside Rhapsody, you should provide an operator `>>` operation for it if you want to generate events to it via the tracer.

go

Description

In Rhapsody, `go` commands advance the application one or more steps. In the tracer, use `go` commands at the command-line. From the animator, you can type `go` commands at the command-line or use the **Animation** toolbar.

A `go` command causes the tracer to execute immediately—even in the middle of reading commands from a file. The remaining (unread) lines are used as commands the next time the tracer stops the execution and searches for commands.

If the application reaches a breakpoint, control might return to you before a `go` command is complete.

There are four `go` commands:

- ◆ `go`—Executes your application until it terminates or reaches a breakpoint
- ◆ `go step`—Executes your application for a single step (that is, until the next Rhapsody-level occurrence)
- ◆ `go next`—Executes your application until the next Rhapsody-level occurrence
- ◆ `go event`—Executes your code until the next dispatching of an event, or until the executable is idle

- ◆ `go idle`—Executes your application until it reaches an idle state where it is waiting for timeouts or events from GUI threads

See also [Notes on Multiple Threads](#).

Syntax

```
go
go step
go next
go event
go idle
```

help

Description

The `help` command displays a brief description of tracer commands.

Syntax

```
help
```

input

Description

The `input` command causes the tracer to read its next command from the specified destination. Once the tracer reaches the end of an input file, it looks to standard input for commands. However, specifying a “+” in the command causes the tracer to resume taking commands from the destination file after it reaches the end-of-file.

Syntax

```
input [+] <destination>
```

Arguments

```
destination
```

Specifies the destination—either standard input (for example, `cin`) or a file name.

Only standard input or the name of the current file can appear without the plus sign (+).

LogCmd

Description

Traces an animation session and saves the sequence of actions for reuse. Note that this command is not case-sensitive.

Syntax

```
LogCmd [+/-] <filename>
```

Arguments

filename

The name of the file to which to write the commands

Example

To write the commands to the file `myScript.txt`, use the following command:

```
LogCmd + myScript.txt
```

To stop writing commands to the file, use the following command:

```
LogCmd -
```

output

Description

The `output` command directs tracer output to the specified destination.

Syntax

```
output <+/-> <destination>
```

Arguments

+

A plus sign (+) adds the specific destination to a list of destinations. Output goes to all items specified on this list.

-

A minus sign (-) removes the destination from the list of destinations. Messages will no longer be sent to this destination.

```
destination
```

Specifies the destination—either standard output (for example, `cout`) or a file name.

If the specified file cannot be opened, the tracer displays an error message and does not add the file to the list of destinations.

The default destination list contains standard output only.

quit

Description

Ends the tracer session.

Syntax

```
quit
```

resume

Description

The `resume` command resumes the specified thread, if it has been suspended. It has no effect if the thread is already active.

Syntax

```
resume threadName
resume #Thread threadName
```

Arguments

`threadName`

Specifies the thread to resume

set focus

Description

The `set focus` command sets the specified thread to be the focus thread.

By default, tracing starts with `mainThread` in focus. The focus changes in the following cases:

- ◆ You enter a `set focus` command.
- ◆ A thread encounters a breakpoint and stops the application, in which case it becomes the current thread.
- ◆ The focus thread died (the active instance to which it belonged was deleted). In this case, the tracer set focus on one of the remaining threads. The selection is random, but if there are nonsuspended threads, one of these is selected.

Syntax

```
set focus <threadName>
set focus #Thread <threadName>
```

Arguments

`threadName`

Specifies the new focus thread

show

Description

The `show` command enables you to view the current status of an object. It enables you to view the status of the object by subject. Subjects include existence, attributes, methods and events.

For example, the following command displays a list of all `B[5]` attributes and their current values:

```
show B[5] attributes
```

Syntax

```
show <object> <interest-list>
```

Arguments

`object`

- ◆ Specifies the object to be traced. It can be one of the following:
- ◆ The name of a class appearing in code, such as `A`. The trace command is applied to all instances of class `A`.
- ◆ The name of an instance that currently exists in the execution, such as `A[3]`. You cannot refer to instances before their construction or after their destruction.
- ◆ A navigation expression, such as `A[3]->itsB[2]`. See [Navigation Expressions](#) for more information.
- ◆ The name of a package appearing in the code. The tracer will report on all classes in the package.
- ◆ A keyword understood by the tracer. These keywords are not case sensitive. The possible keywords are as follows:
 - `#All`—All classes appearing in code.
 - `#Breakpoints`—The list of all breakpoints.
 - `#CallStack`—Operations currently on stack; that is, operations started but not yet terminated, including behavior operations defined on transitions.
 - `#EventQueue`—A queue of all pending events; that is, events sent but not yet received.
 - `#Thread threadName->#CallStack`—The call stack of the thread `threadName`. (All operations that started on this thread but have not yet terminated, including behavior operations defined on transitions).
 - `#Thread threadName->#EventQueue`—The queue of all pending events of the thread name `threadName`. (Events sent but not yet received.)

`interest-list`

Specifies the list of subjects, separated by a commas. The interest list determines what information about the object is reported to you.

This list is optional; if you do not enter any subjects, the tracer reports on the existence of the object only, as if you had executed the following command:

```
show <object> existence
```

The possible subjects are as follows:

existence	constructors
relations	destructors
attributes	timeouts
states	parameters
controls	subclasses
methods	threads
events	

The subject `existence` reports on the existence of the object.

The subject `subclasses` applies the trace commands to *all* of a class's subclasses. It is relevant only to class objects.

The following keywords can be used to define which objects to show (they are not case-sensitive):

- ◆ `#All`—Shows the subjects in the interest list for all classes in code.
- ◆ `#All events`—Displays all the events recognized by the system. This is useful if you forget the exact name of an event.
- ◆ `#Thread threadName->#CallStack / #CallStack`—Shows all operations currently on stack on the thread `threadName` on the focus thread.
- ◆ `#Thread threadName-># EventQueue/# EventQueue`—Shows all events currently in queue on the thread `threadName` on the focus thread.
- ◆ `#Threads`—Shows the status of all threads. Each live thread is displayed as either reactive or suspended. One of the threads has an asterisk by its name, indicating it is the active thread.
- ◆ `#Breakpoints`—Shows a list of currently active breakpoints.

Examples

```
show A[0] states
```

Shows the current states of A[0].

```
show #all all
```

Displays all information about all instances.

```
show #Breakpoints
```

Shows all breakpoints.

```
show #Threads
```

Shows all threads.

```
Show MyClass relations
```

Shows all relations of all instances for every instance of MyClass.

Special Cases

Consider the following special cases when using the `show` command:

- ◆ **Instance objects**—Only the relation, attribute, and state subjects are relevant.
- ◆ **Class objects**—Showing a class displays a list of all instances belonging to that class. If the interest list includes subclasses, the tracer also displays instances of subclasses. If the interest list contains subjects relevant for instance objects, each displayed object also displays itself with respect to these subjects.

For example, the command `show A states` results in the following:

```
A[1]
A[2]
A[3]
A[1] currently in states
  ROOT
  ROOT.S1
  ROOT.S1.S2
A[2] currently in states
  ROOT
  ROOT.S7
  ROOT.S8
A[3] currently in states
  ROOT
  ROOT.S1
  ROOT.S1.S2
```

suspend

Description

The `suspend` command suspends the specified thread. It has no effect if the thread is already suspended.

Syntax

```
suspend threadName
suspend #Thread threadName
```

Arguments

```
threadName
```

Specifies the thread to suspend

timestamp

Description

The `timestamp` command adds a timestamp to the output of a tracer session.

Syntax

```
timestamp <option>
```

Arguments

```
no <option>
```

A timestamp with no option formats the timestamp as `HH:MM:SS`.

```
-
```

The minus sign (`-`) option disables the timestamp.

```
raw
```

The `raw` option enables the timestamp as a cumulative counter, in milliseconds.

trace

Description

The `trace` command enables you to specify which subjects to trace for a given object (class, instance, or keyword). Subjects include existence, attributes, methods, and events. You can choose to trace all objects by one subject, one object by all subjects, or anything in between.

By default, the tracer traces all classes and instances, and does not trace system items (such as call stacks and event queues), as though you had executed the following command:

```
trace #all all
```

In animation, by default, no items are traced, as though you had executed the following command:

```
trace object nothing
```

Syntax

```
trace <object> <interest-list>
```

Arguments

```
object
```

Specifies the object to be traced. It can be one of the following:

- ◆ The name of a class appearing in code, such as `A`. The trace command is applied to all instances of class `A`.
- ◆ The name of an instance that currently exists in the execution, such as `A[3]`. You cannot refer to instances before their construction or after their destruction.
- ◆ A navigation expression, such as `A[3]->itsB[2]`. See [Navigation Expressions](#) for more information.
- ◆ The name of a package appearing in the code. The tracer will report on all classes in the package.
- ◆ A keyword understood by the tracer. These keywords are not case sensitive. The possible keywords are as follows:
 - `#All`—All classes appearing in code.
 - `#CallStack`—Operations currently on stack; that is, operations started but not yet terminated, including behavior operations defined on transitions.
 - `#Thread threadName->#CallStack`—The call stack of the thread `threadName`. (All operations that started on this thread but have not yet terminated, including behavior operations defined on transitions).
 - `#EventQueue`—A queue of all pending events; that is, events sent but not yet received.
 - `#Thread threadName->#EventQueue`—The queue of all pending events of the thread name `threadName`. (Events sent but not yet received.)
 - `#Breakpoints`—The list of all breakpoints.

```
interest-list
```

Specifies the list of subjects, separated by a commas. The interest list determines what information about the object is reported to you.

The possible subjects are as follows:

existence	constructors
relations	destructors
attributes	timeouts
states	parameters
controls	subclasses
methods	threads
events	

The keywords `all` and `nothing` indicate all or none of these subjects.

Precede a subject with a plus (+) or minus (-) sign to add or subtract subjects from the current interest list. If there is neither a + nor -, the subjects typed become the current interest list, replacing any subjects previously selected.

The subject `existence` reports on the existence of the object.

The subject `subclasses` applies the trace commands to *all* of a class's subclasses. It is relevant only to class objects.

Command Semantics

You set or modify the interest list for a given object with the subjects that you list following the name of the object.

Example 1

The following command sets the interest list of `B[5]` to `relations`:

```
trace B[5] relations
```

The tracer will now display a message every time a relation of object `B[5]` is modified. For example:

```
OMTracer B[5] item A[7] added to relation itsA
```

Note that only messages regarding `relations` are displayed for `B[5]`.

Example 2

The following `trace` command adds `relations` to the interest list of `B[5]`:

```
trace B[5] +relations
```

Other messages regarding `B[5]` are displayed or not depending on the value of the interest list before the subject addition command was given.

Example 3

The following `trace` command removes relations from the interest list of `B[5]`:

```
trace B[5] -relations
```

The effect of this command is that *no* message is displayed about the relations of object `B[5]`. Other messages regarding `B[5]` are displayed or not depending on the value of the interest list before the command was given.

For the full list of messages by subject, see [Tracer Messages by Subject](#).

Example 4

If all subjects in the interest list appear with a `+` or `-` sign, the interest list of the object is modified. For example, the following command adds the subject relations to the interest list of `B[5]` and removes the subject states:

```
trace B[5] +relations, -states
```

In contrast, the following command sets the interest list of `B[5]` to include exactly the subject's relations and states:

```
trace B[5] relations, states
```

Special Cases

Consider the following special cases when using the `trace` command:

- ◆ **Class objects**—Class objects propagate their interest lists to all their current and future instances. For example, the following command adds the relation subject to all instances of class `A`:

```
trace A +relations
```

New instances of class `A` created after you execute this command also have relations added to their interest list.

The subject `subclasses` is relevant only for class objects. It propagates the interest list in the given command to all subclasses (recursively).

- ◆ **#All**—Setting the interest list for the keyword `#All` sets the interest lists for all classes in the executable. Modifying the interest list of `#All` modifies the interest lists of all classes in the executable.
- ◆ **#Thread <threadName>-># CallStack / #CallStack**—For call stack objects, only the parameter, method, constructor, and destructor subjects are relevant. Adding these to the interest list will display method (operation), constructor, and destructor calls, with or without parameters, even if the called item is not set to be traced.
- ◆ **#Thread threadName->#EventQueue / #EventQueue**—For event queue objects, only the parameter, timeouts, and method subjects are relevant. Adding these to the interest list will display events sent and received, with or without parameters, even if the sending or receiving item is not set to be traced.

watch

Description

The `watch` command displays all information as changes occur.

Syntax

```
watch
```

Tracer Messages by Subject

The following table lists the possible tracer messages for each subject.


Subject	Messages
Attributes	<pre> XXX[1] Attribute Values: J = 1 k = 3.4 b = 0x55f00b myFoo = Foo[12] XXX[1] Attribute Values changed - new Values J = 1 k = 3.5 b = 0x55f00b myFoo = Foo[12] </pre> <p>The following message can appear anywhere a piece of code has changed the values of the attributes:</p> <pre> State XXX[1] Entered state Foo XXX[1] Exited state Kuku </pre> <p>All attribute values are displayed.</p> <p>The sending and receiving of events by XXX[1] is determined by the interest list of state or operations.</p>
Constructors	<pre> XXX[1] invoked YY() main() invoked YY() YY() returned </pre>
Destructors	<pre> XXX[1] invoked ~YY() main() invoked ~YY() ~YY() returned </pre>
Existence	<pre> class XXX new instance XXX[1] created instance XXX[1] deleted </pre> <p>Instances get their names from their class (X[1], X[2], and so on). Names are unique. Once a name is used, it is not used by a new instance even if the original instance no longer exists.</p> <p>Instances with a No for existence are not traced for any other subject, and appear in the tracer messages as untraced.</p> <pre> Instance Kuku[4] renamed Foo[2].myKuku </pre> <p>This message appears only when Kuku[4] is connected via the composite relation myKuku to Foo[2].</p>

Subject	Messages
Methods	<pre> XXX[1] invoked YY[2]->doIt(j=3, k = 4.3) XXX[1] invoked YY[2]->doIt(int, float) main() invoked Kuku[1]->foo() YY[2]->doIt(j=3, k=4.3) returned YY[2]->doIt(int, float) returned XXX[2] sent YY[8] event start(starter = Kuku[8], times = 2) XXX[2] sent YY[8] event start(Kuku *, int) Kuku[8] sent to itself Event wakeup(time=10.5) Kuku[8] sent to itself event tm(200) at ROOT.Foo YY[8] received from XXX[2] event start(starter = Kuku[8], times = 2) YY[8] sent XXX[2] event start(Kuku *, int) Kuku[8] itself Event wakeup(time=10.5) Kuku[8] received from itself event tm(200) at ROOT.Foo </pre>
Parameters	<p>These messages indicate whether methods and events are displayed via their:</p> <ul style="list-style-type: none"> • Parameters For example: <code>(doIt(j=3, k = 4.3), start(starter = Kuku[8], times = 2))</code> • Signature For example: <code>(doIt(int, float), start(Kuku *, int))</code> <p>When A sends something to B, the parameters in the send message depend on A and the parameters in the receive message depend on B.</p>

Subject	Messages
Relations	A report on the status of all relations appears together with the notification message on the creation of the new instance: Relation itsFoo - Empty Relation itsKuku - Kuku[1], Kuku[4], Kuku[2] ... XXX[1] instance Kuku[7] added to relation itsKuku XXX[1] relation itsFoo set to Foo[2] XXX[1] instance Kuku[7] removed from relation itsKuku XXX[1] relation itsKuku cleared
Timeouts	XXX[1] set tm(tttt) at ROOT.sss XXX[1] cancelled tm(tttt) at ROOT.sss

Ending a Trace Session

End a tracing session by doing one of the following:

- ◆ Type `quit` and press the **Enter** key at the command-line.
- ◆ Click the **Stop Make/Execution** button  on the **Code** toolbar.
- ◆ Select **Code > Stop Execution**.

Otherwise, the trace session ends automatically when the application terminates.

Managing Web-enabled Devices

This section describes the process of managing Rhapsody-built embedded software via the Internet. It examines how, in the development process, to set components as Web-enabled, and how, in the maintenance process, to monitor, control, and make changes to embedded software. This section also includes how to specify preferences in the automatically generated Web pages, which serve as an interface for updating or changing Rhapsody-built software.

Note

The Webify Toolkit is supported only on Internet Explorer Version 5.0 or higher. Refer to the *Rhapsody Readme* file for the list of currently supported environments.

Using Web-enabled Devices

Web-enabled devices contain Rhapsody-built embedded software that you can monitor and control remotely, in real-time, from the Internet. After assigning Web manageability to a model's elements and running the code for that model, Rhapsody automatically generates and hosts a Web site that accesses the application's components through a built-in Web server. The Web pages created by running Web-enabled Rhapsody code serve as a graphical interface for the management of embedded applications. By using the interactive functionality of this interface, you can remotely control the performance of devices.

Although Web-enabling requires no knowledge of Web hosting, design, or development, development teams that want to refine the capability or appearance of their Web interface can do so using their favorite authoring tools.

Besides its ability to manage devices remotely, Web-enabling a device offers the following benefits to the development process:

- ◆ Web browser serves as a window into the device, through which you look into the model to see how a device will perform, eliminating the time-consuming development of writing protocol and attaching hooks that report performance information.
- ◆ Graphical interface provides additional testing on-the-fly and debugging through visual verification of the state of a device before shipping to a manufacturer.

- ◆ Easily created interfaces, exposed via the Internet, serve as visual aids in collaborative planning and engineering and provide a vehicle for rapid prototyping of an application during development.
- ◆ Filtered views can focus customer-specific aspects of a model.
- ◆ Capability to refresh continuously only the changed values and statuses does not overtax device resources.

To Web-enable software, you must perform several tasks from both the server side (in Rhapsody) and the client side (from the Web GUI). In Rhapsody, you select which elements of a model to control and manage over a network, and assign Web-enabling properties to those elements, then generate the code for the model.

To manage the model from the Web GUI, navigate to the URL of the model. Pages viewed in a Web browser act as the GUI to remotely manage the Rhapsody-built device. You can control and manage devices through the Internet, remotely invoking real-time events within the device. Teams can use the Web-enabled access as part of the development process—to prototype, test on-the-fly, and collaborate—or to demonstrate a model's behavior.

Setting Model Elements as Web-Manageable

The first step in Web-enabling a working Rhapsody model is to set elements of the model for Internet exposure. Within a working Rhapsody model, select which elements of the device's application that you want to control or manage remotely through a Web browser, and assign Web-managed properties to those elements.

Note

You cannot webify a file-based C model.

Limitations on Web-Enabling Elements

At the design level, you might find workarounds for building Web-managed models around elements currently not supported by Web-enabling. The following table lists Rhapsody elements and whether they can be Web-enabled.

Element Type	Support and Restrictions
Attributes	Supported types include <code>char</code> , <code>bool</code> , <code>int</code> , <code>long</code> , <code>double</code> , <code>char *</code> , <code>OMBoolean</code> , <code>OMString</code> , <code>RicBoolean</code> , and <code>RicString</code> . Arrays are not supported. Attributes must have either an accessor or mutator, or both.
Classes	Selected objects within classes must be set as Web-enabled. Web-enabling a class does not Web-enable all the child objects of the class.
Native types	Short and all unsigned types (<code>unsigned char</code> , <code>unsigned int</code> , <code>unsigned short</code> , and <code>unsigned long</code>) are supported.
User-defined types	Not supported.
Global variables	Not supported.
Global functions	Not supported.
Packages, components and diagrams	Not supported.

Note the following restrictions:

- ◆ You are limited to 100 Web-enabled instances per model. Depending on the element type, enabling one element might enable multiple instances.
- ◆ Code generation for Web management is not available in Rhapsody in Ada.
- ◆ C-style strings (`char*`) and `RicString` types might cause memory leaks. Setting new values for this type of string from the Web interface assigns newly allocated strings to the attribute. Be sure to properly deallocate this memory. Note that `OMString` will not cause memory leaks.
- ◆ C unions, bit fields, and enumerations are not supported.
- ◆ C++ templates and template instantiations are not supported.

Selecting Elements to Expose to the Internet

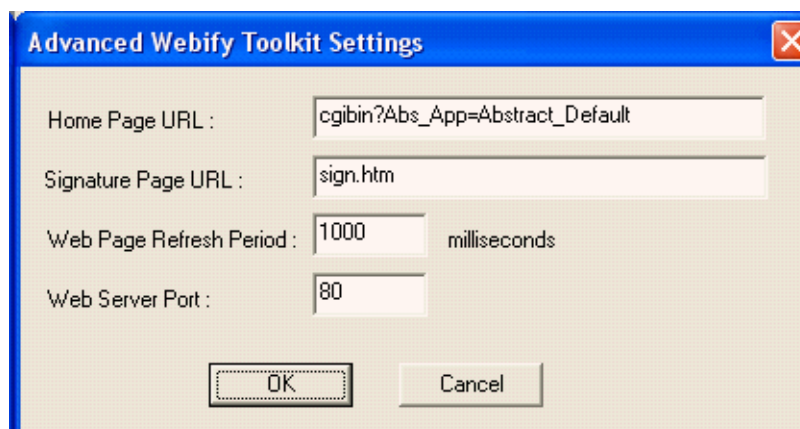
The first step in Web-enabling a working Rhapsody model is to set its components and elements as Web-manageable. When considering which elements to Web-enable within a model, keep in mind the current restrictions and limitations (see [Limitations on Web-Enabling Elements](#)).

To expose your model to the Web, follow these steps:

1. In the browser, navigate to **Components** > <**Component Name**> > **Configurations**. Choose an active configuration belonging to an executable component.
2. In the Features dialog box, click the **Settings** tab.
3. Select the **Web Enabling** check box. This enables Web-enabled code generation for the configuration.

Note: You cannot webify a file-based C model.

4. Optionally, click the **Advanced Settings** button to set the Webify parameters. Rhapsody opens the Advanced Settings dialog box, as shown in the following figure.



This dialog box contains the following controls:

- ◆ **Home Page URL** specifies the URL of the home page. The default value is as follows:
cgibin?Abs_App=Abstract_Default
- ◆ **Signature Page URL** specifies the URL of the signature page. The default value is sign.htm.
- ◆ **Web Page Refresh Period** specifies the refresh rate for the Web page, in milliseconds. The default value is 1000 milliseconds.

- ◆ **Web Server Port** specifies the port number of the Web server. The default value is 80.

Each of these parameters corresponds to a property under `WebComponents::Configuration`. This enables you to save your updated settings with every model, or change them by editing the property values.

5. Repeat Steps 1 through 4 for all the library components that contain Web-enabled elements.
6. Navigate to the elements within a package that you have decided to Web-enable.
7. Double-click the element to open its Features dialog box.
8. Set the stereotype to «Web Managed». Do this for each element you want to view or control from the Internet.

If the element already has an assigned stereotype, you need to Web-enable it through a property, as follows:

- a. Right-click the element and select **Properties**.
- b. Select `WebComponents` as the subject, then set the value of the `WebManaged` property within the appropriate metaclasses to `Checked`.

Currently, the supported metaclasses for the `WebComponents` subject are `Attribute`, `Class`, `Configuration`, `Event`, `File`, `Operation`, and `WebFramework`.

9. Generate the code for the model, build the application, and run it.

Connecting to the Web Site from the Internet

Rhapsody comes with a collection of default pages that serve as a client-side GUI to the remote model. When you run a Web-enabled model, the Rhapsody Web server automatically generates a Web site, complete with a file structure and interactive capability. This site contains a default collection of generated on-the-fly pages that refreshes each element when it changes.

Navigating to the Model through a Web Browser

After generating, making, and running a Rhapsody model with Web-enabled objects, open Internet Explorer (version 5.0 or higher).

For Applications Running on Your Local Machine

In the address box, type the following URL:

```
http://<local host name>
```

In this URL, <local host name> is the “localhost”, machine name, or IP address of your machine.

If you changed the Web server port using the Advanced Settings dialog box, type the following URL:

```
http://<host name>:<port number>
```

By default, the Objects Navigation page will load in your Web browser.

For Applications Running on a Remote Machine or Server

In the address box, type the following URL:

```
http://<remote host>
```

In this URL, <remote host> is the machine name or IP address of the machine running the Rhapsody model.

If you changed the Web server port using the Advanced Settings dialog box, type the following URL:

```
http://<remote host name>:<port number>
```

By default, the Objects Navigation page loads in your Web browser.

Troubleshooting Problems

If you have trouble connecting to a model through the Internet, verify the following:

- ◆ You have IP access to the server you are trying to access.
- ◆ You have Web-instrumented the active component.
- ◆ You have Web-enabled the individual elements.
- ◆ By default, Web-enabled Rhapsody models listen on port 80. Make sure you are currently not running another HTTP protocol application listening on the same port, including another Web-enabled Rhapsody model, or personal Web server. Note that several operating systems (for example, Solaris) do not allow access to port 80. Assign a different port to the Rhapsody model using either the Advanced Settings dialog box or the `WebComponents::Configuration::Port` property.
- ◆ A specific setting of Internet Explorer might cause the following behavior:

The right frame of the “Objects Navigation” page is empty, when it should show the selected object.

To resolve this, in Internet Explorer, go to **Tools > Internet Options > Advanced** and clear the **Use Java2 for <applet> (Requires Restart)** check box.

Connecting to Filtered Views of a Model

If you know the HTTP address of a filtered view of a model, you can initially connect to a model through that view. If you want to monitor or control only certain behaviors of a device from one tailored Web GUI page, you go directly to that page using this method. Rhapsody does not yet support access controls, so providing access to filtered views should be done for the sake of convenience only, because it is not secure.

For information on filtering Web GUI views, see [The Define Views Page](#).

The Web GUI Pages

The default Rhapsody Web server comes with a collection of pages, served up on-the-fly. These pages populate dynamically and contain the status of model elements and present different capabilities and navigation schemes.

The GUI includes the following pages:

- ◆ The Objects Navigation Page
- ◆ The Define Views Page
- ◆ The Personalized Navigation Page
- ◆ The Upload a File to Server Page
- ◆ The Statistics Page
- ◆ The List of Files Page

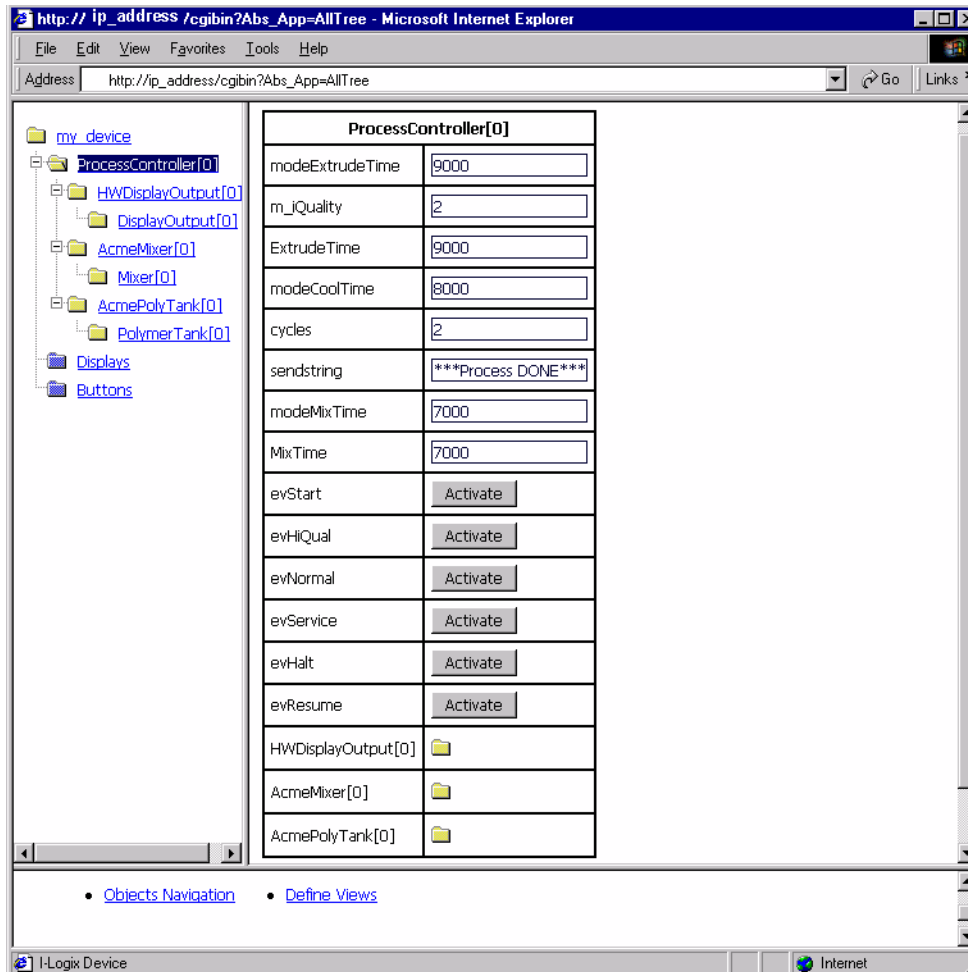
The Objects Navigation Page

The Objects Navigation page provides easy navigation to Web-exposed objects in a model by displaying a hierarchical view of model elements, starting from top-level aggregates. By navigating to, and selecting, an aggregate in the left frame of this page, you can monitor and control your remote device in the aggregate table displayed in the right frame.

This page serves as an explorer-like GUI wherein aggregates appear as folders and correspond to the organization of objects in the model. For information on reading and using aggregate tables, see [Viewing and Controlling a Model via the Internet](#).

When you select an object in the browser frame, it is displayed in the right frame. The status and input fields of objects in the selected aggregate populate the table dynamically. The left column of the table lists the name of the Web-enabled object; the right column lists the corresponding real-time status, input field, activation button, or folder of child aggregates. For information on controlling a model through an aggregate table, see [Customizing the Web Interface](#).

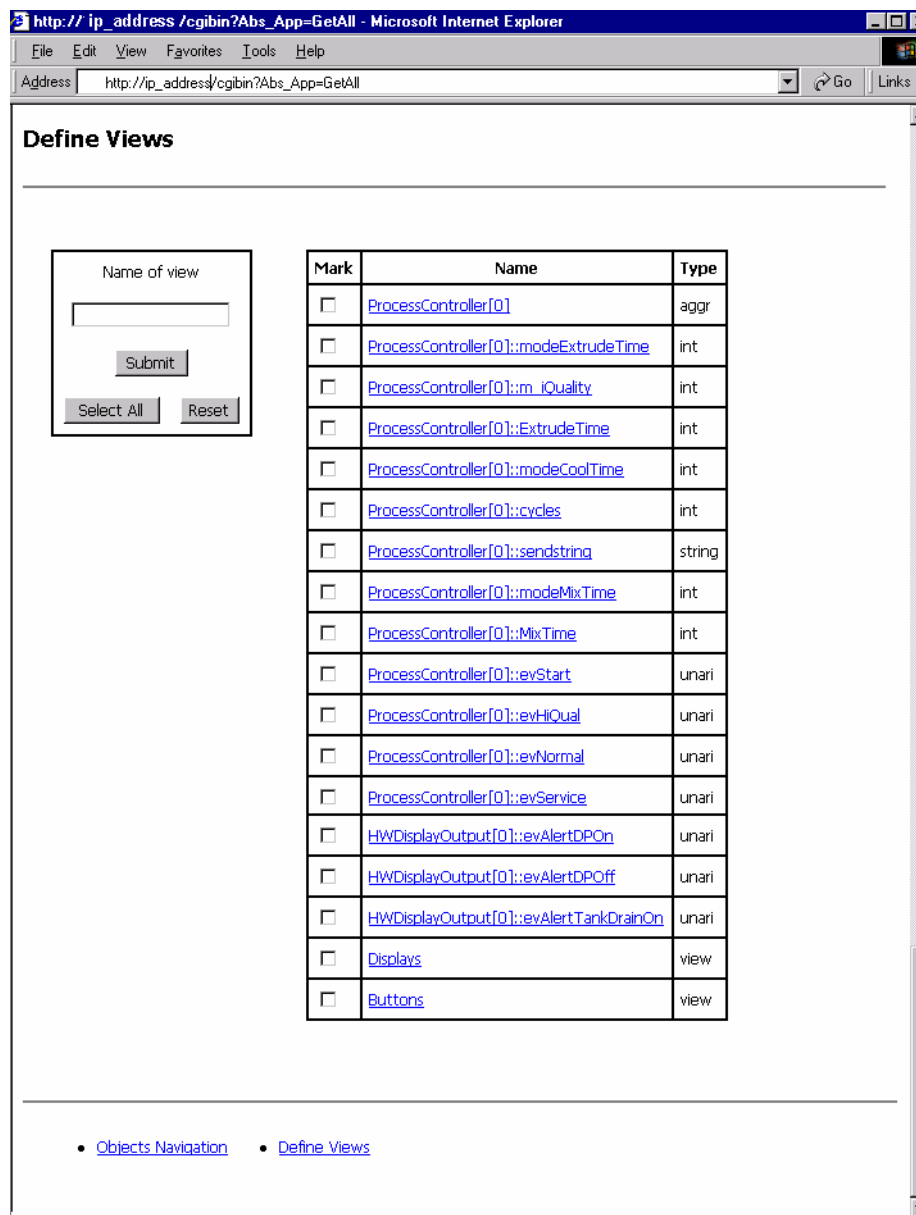
The following figure shows the Objects Navigation page.



The bottom frame in this page contains links to the other automatically generated pages of the Web GUI to manage embedded devices. These links appear at the bottom of all subsequent pages of the default Web GUI and can be customized with your corporate logo and links to your own Web pages.

The Define Views Page

The Define Views page, shown in the following figure, provides a GUI for filtering views into the Rhapsody model. These views can simplify monitoring and controlling maintenance and help you focus on key elements of your models.



The Define Views page enables you to create and name a view of selected elements. The page includes a list of check boxes beside all the available elements that can be included in the view,

and their corresponding element types. Click each element or aggregate to drill down to a graphical representation of the element value, or its aggregate collection of name-value pairs.

To design a view, follow these steps:

1. Check the boxes beside each element you want to include in your view. To include all elements, click **Select All**.
2. Type the name of your view in the **Name of view** box.

The **Reset** button clears the name you entered and all of your selections.

3. Click **Submit**.

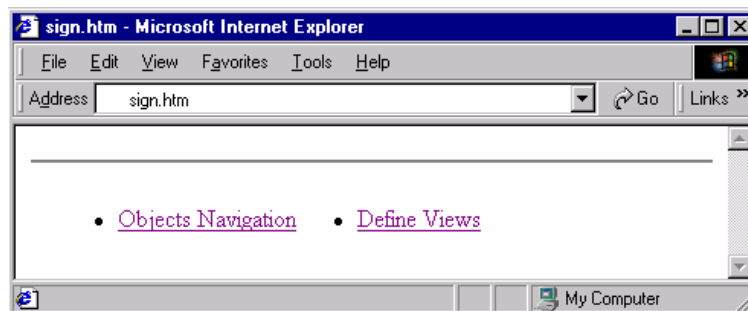
The new view name appears at the bottom of the element table, next to its check box, with “view” as its type.

Note

All views created from the Design View page store in memory and will *not* be saved when you close Rhapsody. Refer to the example and readme file provided in <Rhapsody installation>\MakeTpl\Web\BackupViews.

The Personalized Navigation Page

The Personalized Navigation page, shown in the following figure, enables you to customize your Web interface and add links to more information about your model. This page, conveniently provided as an HTML page, can be opened and viewed for design testing.



Using this page, you could add hyperlinks to the bottom navigation of your Rhapsody GUI, such as a link to e-mail comments to a customer support mailbox. By default, the file includes two links (to the Objects Navigation and Define Views pages) whose addresses are displayed within anchor tags.

To customize the bottom frame of the Web GUI, follow these steps:

1. If you need a general template to edit, right-click the bottom of the Rhapsody Web GUI, and grab the source code.
2. Design the page to your liking, adding your logo and links and adjusting the table structure in the file to accommodate your changes.
3. Rewrite the default file with your own by uploading it to the Rhapsody Web server, either through the Rhapsody interface or through the Upload a File to the Server page. For more on uploading files to the Web server, see [The Upload a File to Server Page](#).

If you do not want to overwrite the default `sign.htm` file, do one of the following:

- ◆ Change the signature page using the Advanced Settings dialog box or the property `WebComponents::Configuration::SignaturePageURL`. This is the preferred method.
- ◆ Upload your own `.html` file to the Rhapsody Web server and call your new file from within the Rhapsody Web configuration file by adding the following function call:

```
SetSignaturePageUrl(<name of your new file>)
```

Viewing and Controlling a Model via the Internet

Controllers manage Rhapsody-built devices through the Internet, remotely invoking real-time events within the device. After Web-enabling, running a Rhapsody model, and connecting to the device in a Web browser, controllers can view and control a device through the Rhapsody Web GUI, using aggregate tables.

Aggregate tables contain name-value pairs of Rhapsody Web-enabled elements that are visible and controllable through Internet access to the machine hosting the Rhapsody model. Navigate to aggregate tables in the Rhapsody Web GUI by browsing to classes selected on the Objects Navigation page.

The following figure shows an example of the name-value pairs as they appear in an aggregate table. This table shows the name of each element within the aggregates, and whether the values are readable or writable.

ProcessController[0]	
modeExtrudeTime	<input type="text" value="7000"/>
modeCoolTime	<input type="text" value="6000"/>
cycles	<input type="text" value="3"/>
sendstring	<input type="text" value="***Process HALT***"/>
modeMixTime	<input type="text" value="100"/>
evStart	<input type="button" value="Activate"/>

You can monitor a device by reading the values in the dynamically populated text boxes and combo-boxes. When a string value extends beyond the width of the text field, position the mouse arrow over the text field to display a tooltip, as shown for the `sendstring` element.

You can control a device in real-time by clicking the **Activate** button, which initializes an event, or by editing writable text fields (the text boxes that do not have a gray background). Note that an input box displays a red border while being edited, indicating that its value will not refresh until you exit the field.

The Web GUI uses different fields and folders to indicate types of values and their read and write permissions. The following table lists the way name-value pairs are displayed.

Name-Value	Read/Write Indication
Numeric values	Readable values dynamically populate the text box. Write-protected values display in a text box with a gray background; otherwise, the value is writable. To send a changed value, type the value in the text box and either press Enter or exit the box by clicking outside of it.
String values	Readable values dynamically populate the text box. Write-protected values display in a text box with a gray background,; otherwise, the value is writable. To send a changed value, type the value in the text box and either press Enter or exit the box by clicking outside of it.
Boolean variable	Writable values display in a combo-box containing True and False values.
Activation buttons	Clicking these buttons initializes the corresponding event in the model.
Tan folders	Denote child aggregates; clicking a folder displays the selected aggregate within.
Blue folders	Denote user-created views nested within aggregates; clicking a folder displays the selected contents within.

Customizing the Web Interface

You can customize the Web interface for a model by creating your own pages to add to their Rhapsody Web GUI, or by referencing the collection of on-the-fly pages that come with Rhapsody. In addition, you can configure the Rhapsody Web server, giving it custom settings appropriate for your management of a Web-enabled device.

Adding Web Files to a Rhapsody Model

You can add your own HTML, image, multimedia or script files to the Rhapsody Web server file system from within the Rhapsody application.

To upload the files to the Web server, follow these steps:

1. In the browser of a working Rhapsody model, navigate to **Components > <Component Name> > Files**.
2. Right-click the **Files** category.
3. Select **Add New File**. Type the name of the new file.
4. In the browser, right-click the file and select **Features**. Set the following controls:

- ◆ **Name**—Type in the name of your HTML file.
 - ◆ **Path**—Type in, or browse to, the file you want to upload to the Web server.
 - ◆ **File Type**—Set to **Other**.
5. Select the **Properties** tab.
 6. Set the `WebComponents::File::WebManaged File` property to **Checked**.
 7. Click **OK** to apply your changes and close the dialog box.
 8. Generate the code. Code generation converts the file to a binary format, and embeds it into the executable file generated by Rhapsody.

You can view Web files you have added to your model by going to the following URL:

```
http://<ip_address>/<filename>
```

Accessing Web Services Provided with Rhapsody

Rhapsody comes with a collection of generated on-the-fly Web pages for you to add to your Web interface to help in the development process. At run time, these pages provide useful functionality in the development process; however, easy access to them in deployment of devices is not appropriate or recommended.

Note

If you need to generate the Web Services libraries for your target, refer to the *Rhapsody Readme* file for your version of Rhapsody to see the list of currently supported environments for the Webify Toolkit. If the libraries you need do not exist in your `$OMROOT\lib` directory, contact Rhapsody Technical Support (see [Contacting Telelogic Rhapsody Support](#)).

The following sections describe how to use, access, and link to each of these pages.

The Upload a File to Server Page

You can upload your own HTML, image, multimedia or script files to the Rhapsody Web server through the Internet from the Upload a File to Server page.

Note

To enable this functionality, you must first add the `RegisterUpload()` call to the `webconfig.c` file.

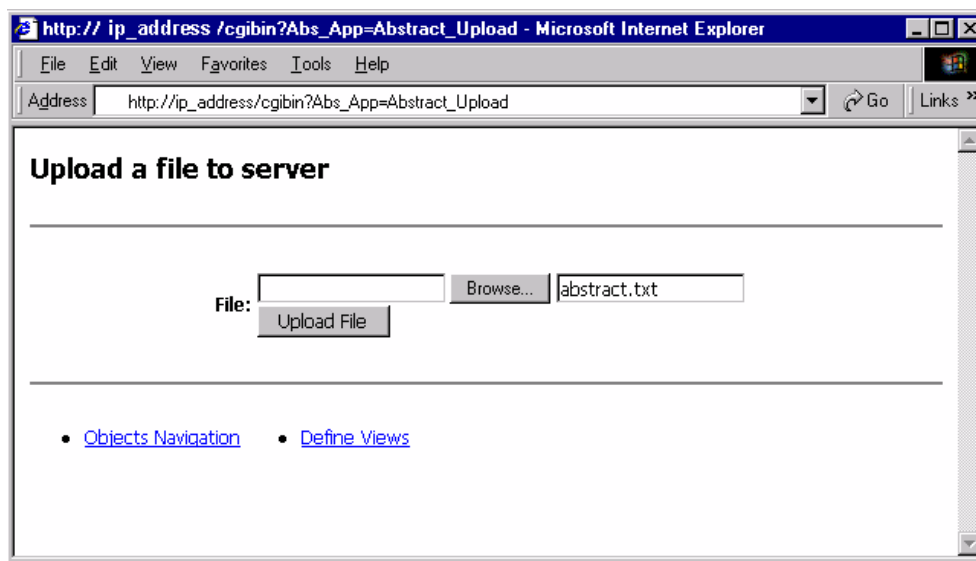
To navigate directly to the page, use the following URL:

```
http://<ip_address>/cgibin?Abs_App=Abstract_Upload
```

To add this page into your Personal Navigation page, its open anchor tag should read as follows:

```
<a href="cgibin?Abs_App=Abstract_Upload">
```

The following figure shows the Upload a File to Server page.



To upload a new file—or overwrite a file—to the Rhapsody Web server using this page, follow these steps:

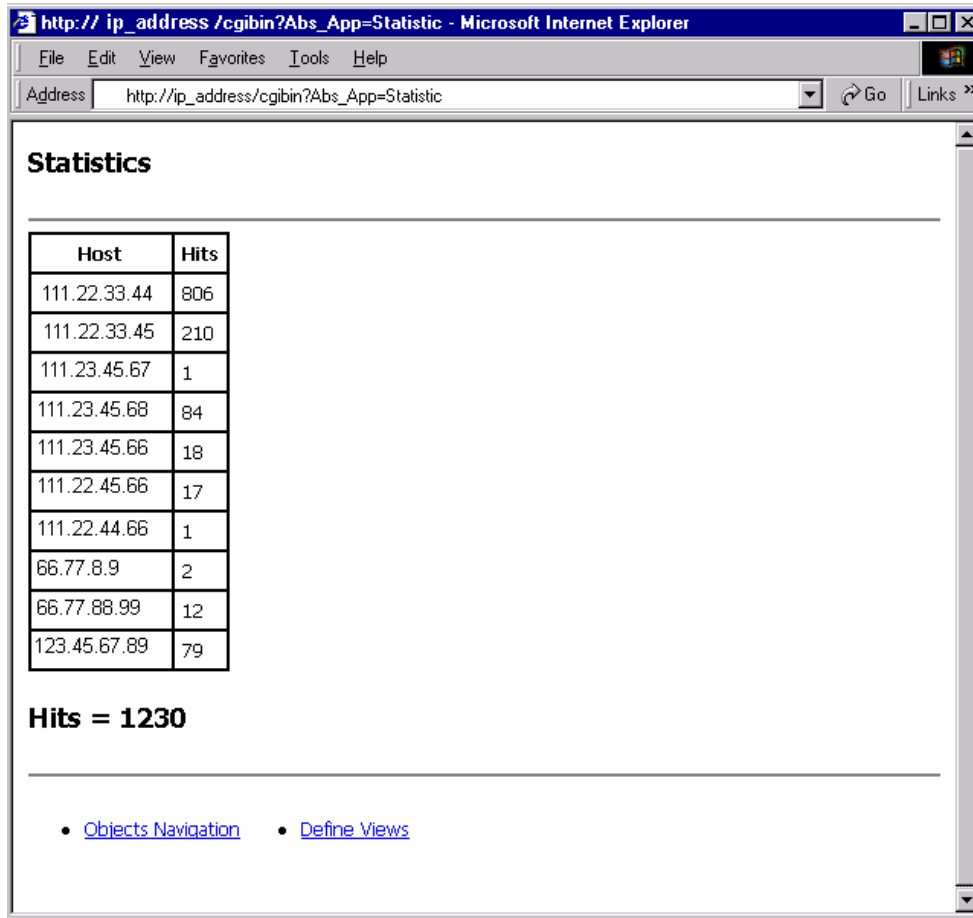
1. Type in the path, or browse, to the file.
2. Click **Upload**.

Note: This page should only be used by developers who understand the impact of their uploads. The collection of on-the-fly services provided with the Rhapsody Web server are intended to assist as a development tool; easy access to them in deployment of devices is not appropriate or recommended.

For an example of the server settings for using this page, see <Rhapsody installation>\Share\MakeTpl\web\WebServerConfig\AddUploadFunctionality.

The Statistics Page

The Statistics page, shown in the following figure, tabulates page and file requests (“hits”) from each machine accessing the model.



Host	Hits
111.22.33.44	806
111.22.33.45	210
111.23.45.67	1
111.23.45.68	84
111.23.45.66	18
111.22.45.66	17
111.22.44.66	1
66.77.8.9	2
66.77.88.99	12
123.45.67.89	79

Hits = 1230

- [Objects Navigation](#)
- [Define Views](#)

You can navigate directly to the page by going to the following URL:

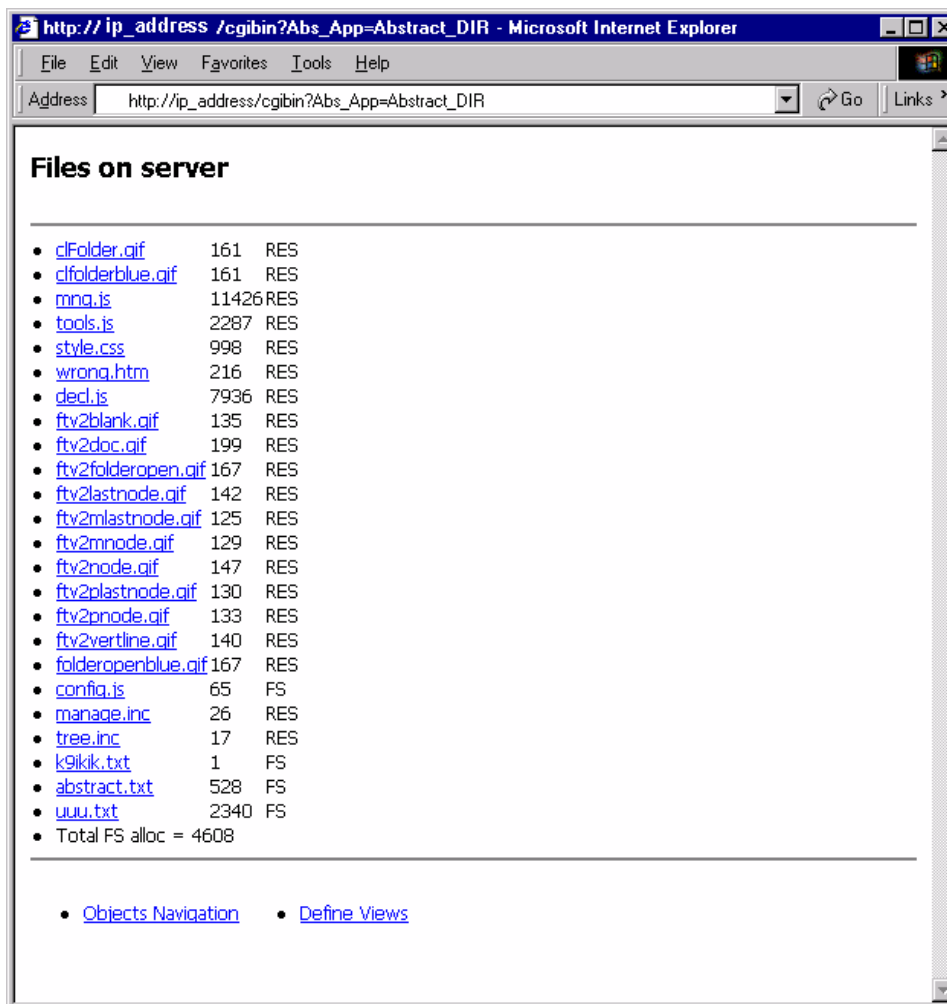
```
http://<ip_address>/cgibin?Abs_App=Statistic
```

To add this page into your Personal Navigation page, its open anchor tag should read as follows:

```
<a href="cgibin?Abs_App=Statistic">
```


The List of Files Page

This List of Files page, shown in the following figure, lists all the text and graphic files that come with the default Web GUI that generates automatically when running a Web-enabled Rhapsody model.



You can navigate directly to the page by going to the following URL:

```
http://<ip_address>/cgibin?Abs_App=Abstract_DIR
```

To add this page into your Personal Navigation page, its open anchor tag should read as follows:

```
<a href="cgibin?Abs_App=Abstract_DIR">
```

The list includes the size, in bytes, and type of each file, and displays the total size of all included Web GUI files.

Files labeled as “RES” denote code segments; files labeled “FS” denote those stored in the file system.

To use the `signEnhanced.htm` navigation page, rename the page `sign.htm`.

Adding Rhapsody Functionality to Your Web Design

You can completely change the layout and design of the Web interface to your Rhapsody model and still have all the functionality provided, by default, when running a Web-enabled Rhapsody model. You can create an interface that refreshes changed element values in real time, and provide the same interactive capabilities to remotely control and monitor a device.

Calling Element Values

After designing a static version of your HTML page, using placeholders for element values, you can overwrite each placeholder text with script that will call the values of each element dynamically from your uploaded page.

1. If you want to design the layout of your page first, design your Web page, leaving static text as a placeholder for a dynamic value.
2. Edit your HTML file, including the `manage.inc` file in the header of the HTML file in script tags before the `</head>` tag:

```
<head>
  <title>Your Title Here</title>
  <script src='manage.inc'></script>
</head>
```

The `manage.inc` file includes a collection of JavaScript files that control client-side behavior of the Rhapsody Web interface.

3. Edit your HTML file, substituting function-calling script for each static value placeholder.

For example:

```
<body>value of evStart</body>
```

This becomes:

```
<body><script>show('nameOfElement')</script></body>
```

In this sample code, `nameOfElement` is the element name assigned to the element by Rhapsody, visible in the default Web interface. Be sure to use the full path name of the element in the `show` function, as in the following example:

```
show('ProcessController[0]::OMBoolean_attribute');
```

4. Save the file and upload it to the Rhapsody Web server (see [Adding Web Files to a Rhapsody Model](#)).

You can link to this file in your Web interface from your new design scheme. If you want to make a page the front page of your Web GUI, see [Setting a Home Page](#). Keep in mind that Rhapsody does not yet support a hierarchical file structure, so HTML and image files are at the Web server's root directory.

Binding Embedded Objects to Your Model

Using this design method, you embed graphical elements in your Web page as JavaScript objects, then bind those objects to the Rhapsody model's real-time values. By binding embedded graphics and mapping them to the values of Web-enabled elements, you can create a page where an image is displayed in the Web interface when a process has stopped (for example, a stop sign), whereas another indicates that the process is running (such as a green light).

One approach to making such pages is to design a page with all your static elements, then add script to your HTML. follow these steps:

1. Edit your HTML file, including the `manage.inc` file in the header of the HTML file in script tags before the `</head>` tag:

```
<head>
  <title>Your Title Here</title>
  <script src='manage.inc'></script>
</head>
```

The `manage.inc` file includes a collection of JavaScript files that control client-side behavior of the Rhapsody Web interface.

2. To embed model elements in the page, create JavaScript objects of the `WebObject` type and bind each object with the elements of the device you are managing and controlling through the Internet, using the `bind` function.

The following sample code displays a yellow lamp when a Boolean element's value is true, and a red lamp when the value is false:

```
<html><head><title>Page Title</title>
<script src='manage.inc'></script>
<script>
function updateMyLamp(val)
{
    if (val == 'On')
    {
        document.getElementById('myImage').src = 'redLamp.gif';
    }
    else
    {
        document.getElementById('myImage').src = 'yellowLamp.gif';
    }
}
</script>
</head>
<body>
<script>
var lamp = new WebObject;
bind(window.lamp,
```

```
        'ProcessController[0]::OMBoolean_attribute');
lamp.update = updateMyLamp;
</script>
<img id=myImage border=0>
<hr noshade>
<i>Rotate the bool values here<i>
<script>show('ProcessController[0]::rotate');</script>
</body>
</html>
```

The declaration of the object and binding takes place in the header of the document, between the second pair of script tags.

The `bind` function serves as a bridge between the element values in the model and the Web interface. It takes two arguments—the variable name of the JavaScript object (`lamp` in the example) and the name of the model element in Rhapsody (`ProcessController[0]::rotate` in the example).

In the example, the following line refreshes updated model values in the Web GUI:

```
lamp.update = updateMyLamp;
```

This update function accepts one argument, which is the new value of the element.

If a GUI control in the page needs to pass information to the device, call the `set` method of the corresponding object. The `set` method accepts one argument, the newly set value.

Calling Model Functions

In addition to the ability to display attribute values on your custom Web pages, you can add to Web pages the ability to call functions in your model.

To allow a Web page to call a function from your model, make the following changes to the page:

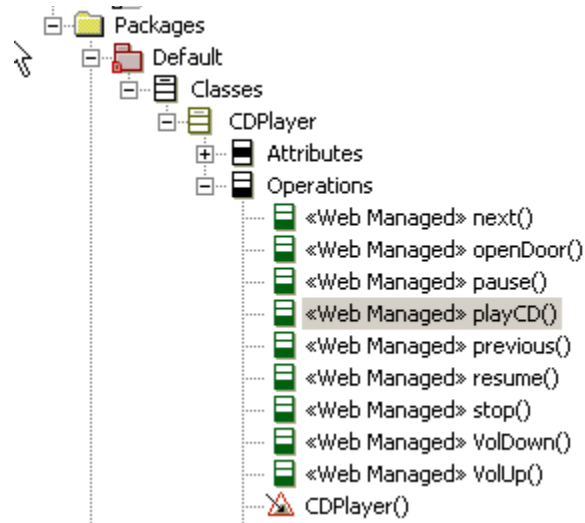
1. Declare a new variable as a `WebObject`.
2. Call the `bind` function, providing the variable name and the name of the function from your model as parameters.
3. Add a link or other control to call the `set()` function for the new variable.

The following code snippets reflect these steps:

```
<script>
var play = new WebObject;
bind(window.play, 'CDPlayer[0]::playCD');
</script>

<p><a href="javascript:play.set();"></a></p>
```

In this example, `CDPlayer[0]` represents the relevant object in the model, and `playCD` is the name of the operation that is to be called.



Customizing the Rhapsody Web Server

You can customize the Rhapsody Web server in two ways:

- ◆ Using the Advanced Settings dialog box
- ◆ Editing the `webconfig.c` file

To customize the Rhapsody Web server, you can modify the `webconfig.c` file and add that file to your Web-enabled Rhapsody model. The `webconfig.c` file is located within your Rhapsody installation directory in `Share\MakeTmp1\Web`. The file is clearly commented before each function. To change the Web server settings, edit the argument of the function to the appropriate setting.

Because other models will use this file to configure Web server settings, copy the file to another directory within your Web-enabled project, for example, and edit that copy of the `webconfig.c` file. In the process of customizing your Web server, be sure to add it to the configuration of your active component from within the Rhapsody interface.

To add the modified `webconfig.c` file to your Web-enabled model, follow these steps:

1. In the browser of a working Rhapsody model, navigate to **Components** > **<Component Name>** > **Configurations**. Select the active configuration.
2. On the Features dialog box, on the **Settings** tab, in the **Additional Sources** box, type the location of the modified `webconfig.c` file.

The kit includes examples in the `<Rhapsody installation>\Share\MakeTmp1\web\WebServerConfig` directory.

Note

If you customize your Web server using the `webconfig.c` file, your changes will overwrite the properties set in the Advanced Settings dialog box.

Setting a Device Name

To change the name of a device, modify the argument to the `SetDeviceName` method.

For example, to change the setting of the device name to “Excellent Device”, modify the call as follows:

```
SetDeviceName("Excellent Device");
```

See `<Rhapsody installation>\Share\MakeTmp1\web\WebServerConfig\ChangeDeviceName` for an example.

Setting a Home Page

To change the setting of the home page to `index.htm`, the function and argument should read as follows:

```
SetHomePageUrl ("index.htm");
```

Setting a Personalized Bottom Navigation

There are two ways to personalize the bottom navigation page:

- ◆ Overwrite the `sign.htm` file with your changed design and upload it to the Rhapsody Web server (see [The Personalized Navigation Page](#)).
- ◆ Use a function in the `webconfig.c` files to use another file for personalized bottom navigation.

For example, to change the page name of the bottom navigation page to `navigation.htm`, the function and argument should read as follows:

```
SetSignaturePageUrl ("navigation.htm");
```

Setting a Port Number

By default, the Rhapsody Web server listens on port 80. To change the port number, change the argument to the `SetPropPortNumber` method.

For example, to change the port number to 8000, use the following call:

```
SetPropPortNumber (8000);
```

Setting an Automatic Refresh Rate

To change how frequently the Rhapsody Web server refreshes changed values, use the `SetRefreshTimeout` function. The argument to this function holds the refresh rate, in milliseconds.

For example, to change the refresh interval to every 5 seconds, use the following call:

```
SetRefreshTimeout (5000);
```

See `<Rhapsody installation>\Share\MakeTpl\web\WebServerConfig\ChangeRefreshRate` for an example.

Enabling File Upload

To enable the file upload capability, add the `RegisterUpload` function to the `webconfig.c` file. This function takes no arguments.

See `<Rhapsody installation>\Share\MakeTmpl\web\WebServerConfig\AddUploadFunctionality` for an example.

Reports

Rhapsody offers two ways to generate reports from the models, charts, the generated code, and other items:

- ◆ A simple and quick internal RTF report generator, described in [Using the Internal Reporting Facility](#).
- ◆ A more powerful reporting tool, Rhapsody ReporterPLUS

ReporterPLUS

ReporterPLUS produces reports that are suitable for formal presentations and can be output in any of these formats:

- ◆ HTML page
- ◆ Microsoft Word
- ◆ Microsoft PowerPoint
- ◆ RTF (.rtf)
- ◆ text (.txt)

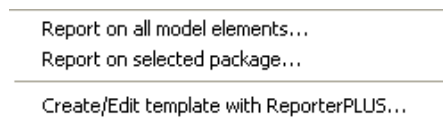
You can save the file and view it in any program designed to display the report's format. See [Viewing Reports Online](#) for more information.

ReporterPLUS creates documents using these techniques:

- ◆ Extracting text and diagrams from a model created in Rhapsody.
- ◆ Adding text and diagrams from the model and images to the document. (Note that text files do not include diagrams.)
- ◆ Adding boilerplate text specified in the ReporterPLUS template to the document.
- ◆ Formatting the document according to the formatting commands in the ReporterPLUS template, as well as the specifications in a *Word* template (.dot file), a PowerPoint template (.pot file), or an HTML style sheet (.css file). Using a .dot, .pot, or .css file is optional. You can also use HTML tags to format HTML documents.

Launching ReporterPLUS

To start ReporterPLUS from inside of Rhapsody, select **Tools > ReporterPLUS**. This menu displays options for printing the model currently displayed in Rhapsody.



The **Report on selected package** option is not available from this menu unless a package in the model is highlighted in the Rhapsody browser. If one of the first two items is selected, a report can be generated using a predefined template without displaying the main ReporterPLUS GUI. If the last option is selected, ReporterPLUS starts with no model elements imported.

To start ReporterPLUS from outside of Rhapsody, select **Start > All Programs > Telelogic > Telelogic Rhapsody (version number) > Rhapsody ReporterPLUS (version number) > Rhapsody ReporterPLUS (version number)** option from the final menu that displays.

Note

ReporterPLUS can also be run from the command line, as described in the *ReporterPLUS Guide*.

ReporterPLUS Templates

Rhapsody includes numerous pre-fabricated report templates that you may want to use as they are or customize to meet your needs.

Note

These files are stored in the Rhapsody\reporterplus\Templates directory.

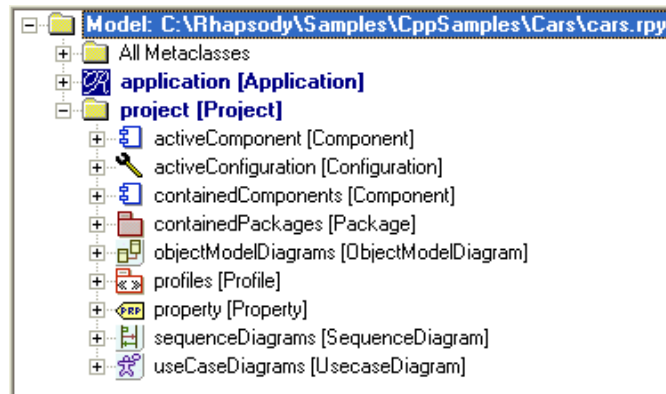
Using the ReporterPLUS Interface

Rhapsody models can be loaded into the ReporterPLUS interface and used to create generic or model-specific templates. This interface allows you to create and modify templates graphically using a drag-and-drop method.

Follow these steps to use the ReporterPlus templates with your model:

1. Start Rhapsody if it is not already started and display the project for which you need a report.
2. Select **Tools > ReporterPLUS > Create/Edit template with ReporterPLUS**.
3. The ReporterPLUS interface opens with a **Tip of the Day** dialog box. Close it.

4. Then the ReporterPLUS Wizard dialog displays. Click **Cancel**.
5. The project model's details are listed in ReporterPLUS' upper left corner as shown in this example.



6. Highlight an item in the model and the detailed description of that item appears to the right, as shown in this example.

Name	Type	Value
description	String	
descriptionHTML	String	
descriptionRTF	String	
displayName	String	
fullPathName	String	
fullPathNameIn	String	
isOfMetaclass	Boolean	
isShowDisplayName	Boolean	
metaClass	String	
name	String	
requirementTraceabilityHandle	Long	
userDefinedMetaClass	String	

Examining Pre-fabricated Templates

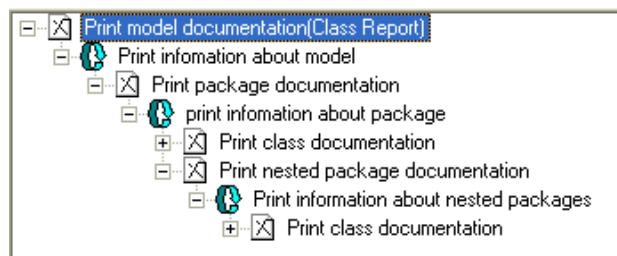
To examine the pre-fabricated Rhapsody templates, follow these steps:

1. From the ReporterPLUS menu, select **File > Open Template**. This displays all of the existing Rhapsody report templates.

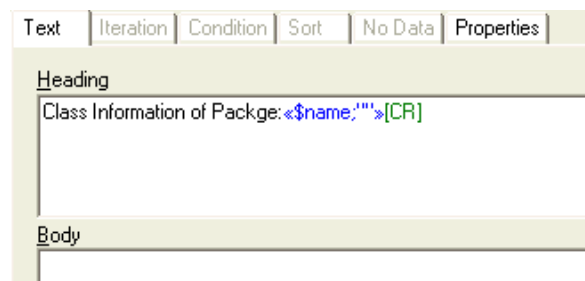
Note

The GetStarted.tpl is a simple template to use when you are becoming familiar with this reporting tool.

2. Select a template from the list. The structure of the selected template displays in the lower left corner. This structure uses a standard Windows tree design as shown in this example.



3. Highlight an item in the tree structure and view the details of that item in the window to the right as in this example.



Customizing Templates

To customize an existing template for your Rhapsody project, follow these steps:

1. Display your Rhapsody model in ReporterPLUS.
2. From the *ReporterPLUS* menu, select **File > Open Template**.
3. Select the template from the list that most closely resembles the type of report you want to generate. The structure of the selected template displays in the lower left corner.
4. An easy method for customizing the generic template for your project is to drag an item from your model down to the template area and drop it in the desired location.
5. You may also want to add standard headings and text to an existing template. To add this “boilerplate” material, highlight a section of the template in the lower left window and click the **Text** tab in the lower right window. Type text in the Heading and Body sections as desired.
6. For more complex changes, study the Q Language that ReporterPLUS uses to define report expressions. This language is defined in a PDF file accessed from ReporterPLUS Help Topics.

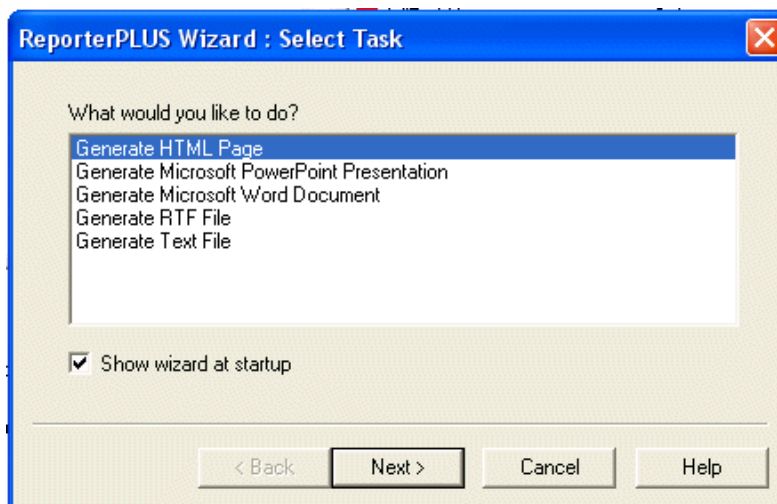
Generating Reports Using Existing Templates

Note

If you are going to generate a report in *Microsoft Word* or *PowerPoint*, be certain that *Word* and *PowerPoint* are closed before you start this procedure to avoid a conflict between *ReporterPLUS* and the Microsoft program.

To generate reports quickly from the Rhapsody interface using templates you have examined or created previously in ReporterPLUS, follow these steps:

1. With your project open in Rhapsody, select **Tools > ReporterPLUS** from the menu.
2. From the next menu, select one of these two options:
 - ◆ Report on all model elements
 - ◆ Report on selected package
3. Rhapsody displays the ReporterPLUS Wizard (shown below) that allows you to select the desired output format and on subsequent dialog boxes, the template, and directory location and name for the finished report.



Note

However, if your project model is very large, you should generate the report from ReporterPLUS interface for a more rapid generation.

Viewing Reports Online

After creating reports using ReporterPLUS templates and facilities, follow these guidelines for viewing the reports online:

- ◆ For reports generated in Linux, view the HTML reports in Mozilla Firefox and the RTF reports in Open Office 2.0 or higher.
- ◆ For reports generated in Windows, view HTML reports in any standard browser available on the PC and for the other report formats, the appropriate programs for viewing these reports launch when the report files are clicked to launch.

Generating a List of Specific Items

If during development you want to generate a list of items, such as all of the ports using an interface, you can focus the generated report on that section of the model. To generate a list of specific items in a model, follow these steps:

1. Display the model in Rhapsody.
2. In the browser, select the section of the model containing the specific items that you need in a list.
3. Select **Tools > ReporterPLUS > Report on selected package**.
4. Select the template you want to use for the report and generate and save the report.

Using the System Model Template

ReporterPLUS includes a template designed for systems engineering called `SysMLreport.tpl`. To use this template for your report, follow these steps:

1. With your model displayed in the Rhapsody interface, select **Tools > ReporterPLUS** from the menu.
2. From the next menu, select **Create/Edit template with ReporterPLUS**.
3. Your model displays in the upper left corner of the ReporterPLUS interface.
4. Select **File > Open Template**.
5. Select the `SysMLreport.tpl` and use it to produce a report for your model. You may wish to change the template to meet your specific needs.

Report Layout

The main elements of each section are shown along with their page locations and are hyperlinked. The generated report contains the following sections covering the complete SysML profile:

- ◆ Requirements diagrams
- ◆ Use case diagrams
- ◆ Sequence diagrams

- ◆ Structure diagrams
- ◆ Object model diagrams
- ◆ Internal and External block diagrams
- ◆ Parametric diagrams
- ◆ Data dictionary
- ◆ Model configuration

The report template uses the standard SysML features built into your model when you select the SysML project **Type** when you first created your project. This selects the SysML profile with the predefined diagrams needed for systems designers.

Requirements Diagrams

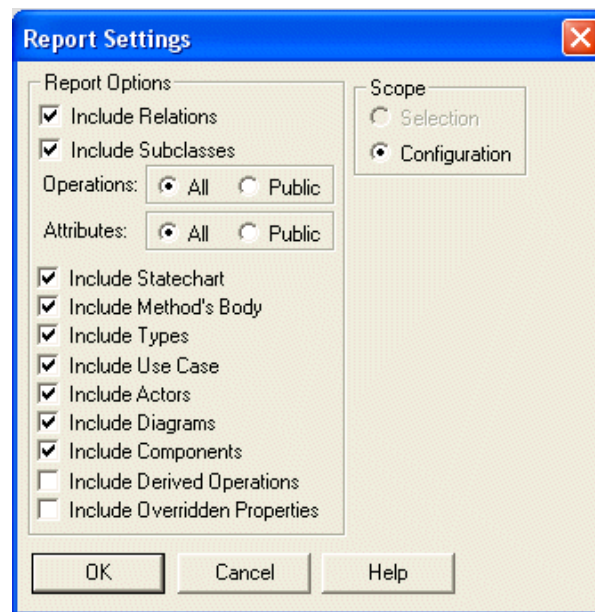
For any requirements diagrams, the `SysMLreport.tpl` supplies hyperlinks to the location of the definition of any use cases, actors, packages, classes or blocks shown in the diagram. Each requirement must have a **Stereotype** setting so that the reporting feature can extract the requirements data.

Using the Internal Reporting Facility

The internal reporting facility is particularly useful for quick print-outs that the developer needs to use for debugging the model. The reports are not formatted for formal presentations.

Producing an Internal Report

To create a report using the simple, internal reporter, select **Tools > Report on mode**. The Report Settings dialog box opens, as shown in the following figure.



The dialog box contains the following fields:

- ◆ **Report Options**—Specifies which elements to include in the report. The possible values are as follows:
 - **Include Relations**—Include all relationships (associations, aggregations, and compositions). By default, this option is checked.
 - **Include Subclasses**—List the subclasses for each class in the report. By default, this option is checked.
- ◆ **Scope**—Specifies the scope of the report. The possible values are as follows:
 - **Selection**—Include information only for the selected elements.
 - **Configuration**—Include information for all elements in the active component scope. This is the default value.

- ◆ **Operations**—Specifies which operations to include in the report. The possible values are as follows:
 - **All**—Include all operations. This is the default value.
 - **Public**—Include only the public operations.
- ◆ **Attributes**—Specifies which attributes to include in the report. The possible values are as follows:
 - **All**—Include all attributes. This is the default value.
 - **Public**—Include only the public attributes.
- ◆ **Include Statechart**—If the project has a statechart, this option specifies whether to list the states and transitions in the report. By default, this option is checked.
- ◆ **Include Method's Body**—Specifies whether to include the code for all method bodies in the report. By default, this option is checked.
- ◆ **Include Types**—Specifies whether to list the types in the report. By default, this option is checked.
- ◆ **Include Use Case**—Specifies whether to list the use cases in the report. By default, this option is checked.
- ◆ **Include Actors**—Specifies whether to list the actors in the report. By default, this option is checked.
- ◆ **Include Diagrams**—Specifies whether to include diagram pictures in the report. By default, this option is not checked.
- ◆ **Include Components**—Specifies whether to include component information (configurations, folders, files, and their settings) in the report. By default, this option is not checked.
- ◆ **Include Derived Operations**—Specifies whether to include generated operations (when the property `CG::CGGeneral::GeneratedCodeInBrowser` is set to `Checked`). By default, this option is `Cleared`.
- ◆ **Include Overridden Properties**—Specifies whether to include properties whose default values have been overridden. By default, this option is not checked.

To generate the report, select the appropriate values, then click **OK** to generate the report. The report is displayed in the drawing area with the current file name in the title bar.

Setting the RTF Character Set

For RTF output from the Rhapsody internal reporter, you may define the necessary character set using the `General::Report::RTFCharacterSet` property.

This character set is used in the RTF multi-language and description styles for the **Name Label** and **Description** fields of the report. The RTF file created by the Rhapsody internal reporter must be included as a specific character set for each language. For example, set this property can be set to `\fcharset128` for Japanese.

The default value (an empty string) preserves the current behavior.

Using the Internal Report Output

When you generate a report in Rhapsody using **Tools > Report on model**, the initial result uses the internal RTF viewer. To facilitate the developer's research, this output may be used in the following ways:

- ◆ To locate specific items in the report online, select **Edit > Find** from the menu and type in the search criteria.
- ◆ To print the initially generated report, select the **File > Print** menu option.
- ◆ The initially generated report is only a view of the RTF file that the facility created. This file is located in the project directory (parallel to the `.rpy` file) and is named `RhapsodyRep<num>.rtf`. If you wish, open the RTF file using a word processor that handles RTF format, such as Microsoft Word.

Java-specific Issues

This section covers Java-specific issues.

Generation of Javadoc Comments

Rhapsody provides a mechanism for including Javadoc comments when code is generated for models developed in RiJ.

In general, the Javadoc generation mechanism is based on the following:

- ◆ Rhapsody properties called `DescriptionTemplate` for elements such as classes and operations. The content of these properties determines the appearance of the generated Javadoc comments.
- ◆ Automatic retrieval of Rhapsody element fields that correspond to Javadoc tags, such as descriptions and operation arguments
- ◆ Special tags and corresponding keywords that can be used for standard Javadoc tags that do not have corresponding Rhapsody elements, for example, `@version`.

Including Javadoc Comments in Rhapsody-generated Code

Javadoc comment generation is enabled, by default, for RiJ projects.

To have Javadoc comments included, just generate code as you normally would.

In the generated code, you should see Javadoc comments based on the descriptions you have provided for model elements. Comments for operations will also include any descriptions you have provided for operation arguments.

If you do not see such comments in your code, open the Features dialog box for the configuration you are using, and verify that the **Generate JavaDoc Comments** check box on the **Settings** tab is selected. If this option is selected, and you still do not see Javadoc comments in your generated code, read the suggestions listed in [Javadoc Troubleshooting](#).

In addition to generating these basic Javadoc comments, you can have Rhapsody include the following standard Javadoc tags: author, deprecated, return, see, since, version. To include these Javadoc tags in your generated code:

1. Open the Features dialog box for a model element that you want to document.
2. Add Javadoc content by providing values for the various tags displayed on the **Tags** tab.
3. Repeat this process for each model element you want to document.
4. Generate the code.

Once your code includes Javadoc comments, you can generate a Javadoc report using the standard Javadoc process (see [Javadoc Tool home page](#)).

Changing the Appearance of Javadoc Comments in Generated Code

The appearance of Javadoc comments in the generated code is determined by the documentation templates defined using the various `DescriptionTemplate` properties.

To change these templates:

1. Examine the content that *JavaDocProfile* provides for the various `DescriptionTemplate` properties.
2. Modify the values of these properties to match the appearance you would like.

When making changes to these properties, keep in mind the following:

- ◆ Use new lines to indicate where you would like Rhapsody to begin a new line. You can only see such line breaks by using the `..` button to open the text editor for the property.
- ◆ As you will notice in the default content that *JavaDocProfile* provides for the `DescriptionTemplate` properties, you can use the characters `[[]]` in your template definitions. If you enclose part of the definition in these brackets, Rhapsody will only generate the relevant Javadoc tag, for example, `@version`, if content has been provided for that tag for the element in question.

Enabling/Disabling Javadoc Comment Generation

For new Java projects, Javadoc comments are generated automatically.

To disable Javadoc generation:

1. Open the Features dialog box for the relevant Configuration in your model.
2. On the **Settings** tab, clear the **Generate JavaDoc Comments** check box.

Note

When you clear/select the **Generate JavaDoc Comments** check box, it changes the value of the boolean property `CG::Configuration::UseDescriptionTemplates`.

To enable the generation of Javadoc comments for existing projects:

1. Open the project in Rhapsody.
2. Add the *JavaDocProfile* profile to your project.

"Built-in" Keywords

The content that *JavaDocProfile* provides for the `DescriptionTemplate` properties contains the following keywords that do not have corresponding Rhapsody tags in the profile.

- ◆ `$.Description` represents the description of the corresponding model element.
- ◆ `$.Name` represents the name of the corresponding model element.
- ◆ `$.Arguments` represents the Javadoc content generated for operation arguments on the basis of the property `JAVA_CG::Argument::DescriptionTemplate`.

Description Templates in JavaDocProfile

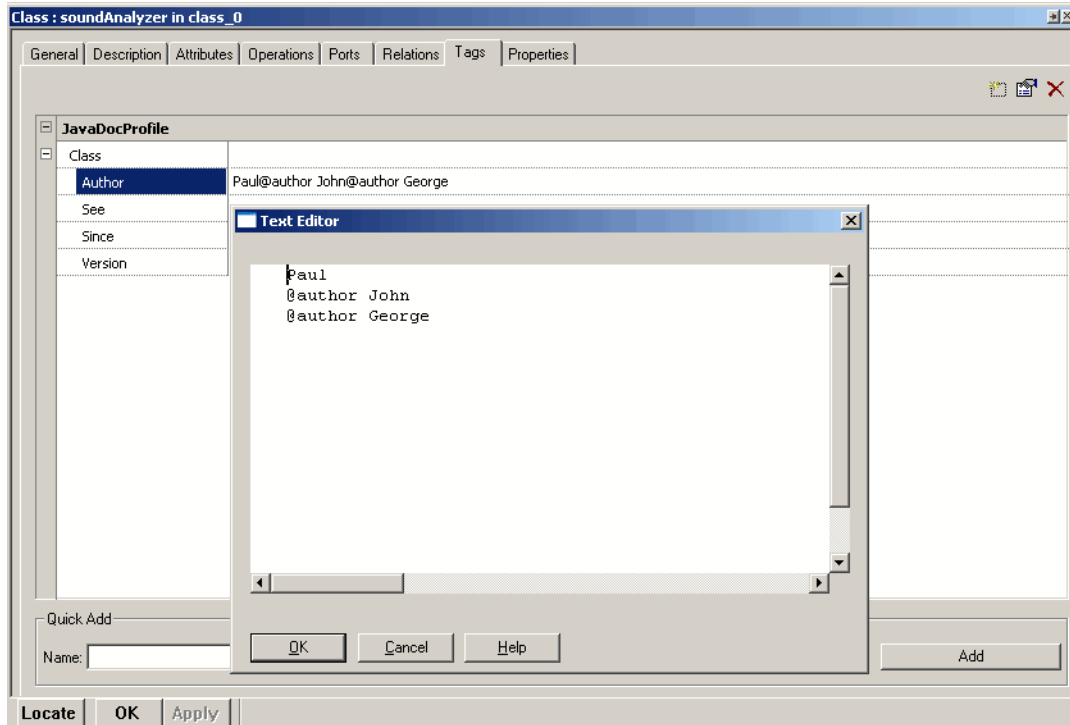
JavaDocProfile provides Javadoc templates for the following properties:

- ◆ `JAVA_CG::File::Header`
- ◆ `JAVA_CG::Package::DescriptionTemplate`
- ◆ `JAVA_CG::Class::DescriptionTemplate`
- ◆ `JAVA_CG::Event::DescriptionTemplate`
- ◆ `JAVA_CG::Attribute::DescriptionTemplate`
- ◆ `JAVA_CG::Relation::DescriptionTemplate`
- ◆ `JAVA_CG::Operation::DescriptionTemplate`
- ◆ `JAVA_CG::Argument::DescriptionTemplate`

Multiple Appearance of Javadoc Tags

You may want to have certain Javadoc tags appear a number of times for a single element. For example, you may want to have *@author* appear a number of times for a single class that was written by a number of individuals.

In such a case, when assigning a value to the relevant Rhapsody tag, add `@author` before each name except for the first.



Adding New Javadoc Tags

The Javadoc generation mechanism allows you to define new Javadoc tags that you would like to use. To define a new Javadoc tag:

1. Create a writable copy of the *JavaDocProfile* profile by selecting **File > Add to Model**, and then selecting the file `<Rhapsody installation path>\Share\Profiles\JavaDoc\JavaDocProfile.sbs` (When Rhapsody indicates that the profile already exists in the model, select the **Replace Existing Unit** option.)
2. Create a new Rhapsody tag in your profile. When you create the new tag, select the appropriate item from the **Applicable To** drop-down list.
3. Modify the value of the `DescriptionTemplate` property for the relevant type of element. Use `${tagname}` to have Rhapsody include the tag text in the Javadoc comment.
4. Open the Features dialog box for specific elements of the relevant type, and on the **Tags** tab provide a value for the new Rhapsody tag you have added.

Example:

1. Create under your profile a new tag called `codeReviewer`. From the `Applicable to` drop-down, select **Class**.

2. For the `JAVA_CG::Class::DescriptionTemplate` property, add the following to the property value: `[[* @codeReviewer $codeReviewer]]`
3. For one or more of the classes in your model, open the Features dialog box, and on the **Tags** tab, enter **Steve** for the value of the tag `codeReviewer`.

When you generate code for your model after these changes, the Javadoc comments for these classes will include `@codeReviewer Steve`.

Javadoc Handling in Reverse Engineering and Roundtripping

When code with Javadoc comments is reverse engineered, all of the Javadoc comments will be made part of the description of the element.

Rhapsody's roundtripping feature does not support changes to Javadoc comments.

Javadoc Troubleshooting

If the **Generate Javadoc Comments** check box on the configuration **Settings** tab is selected and you still do not see Javadoc comments in your generated code:

1. Verify that the *JavaDocProfile* profile is loaded in your model.
2. Confirm that the relevant properties are not overridden at some level (`JAVA_CG::File::Header`, the `DescriptionTemplate` properties, and `CG::Configuration::UseDescriptionTemplates`).

Static Import

The static import construct was introduced in J2SE 5.0 in order to allow unqualified access to static members of a class. Beginning with version 7.2, Rhapsody is capable of modeling static imports and generating appropriate code. In addition, the reverse engineering feature can handle static imports in Java code, and the roundtripping feature can handle changes to static import statements.

Rhapsody allows you to model both static import of individual class members (`import static java.lang.Math.PI`) and static import of all static members of a class (`import static java.lang.Math.*`).

Modeling of static imports is based on use of the **StaticImport** stereotype in the `PredefinedTypesJava` package. The **StaticImport** stereotype inherits from the **Usage** stereotype.

Adding Static Imports to a Model

To add a static import to your model:

1. Create a dependency in the browser or by drawing a dependency in an object model diagram. The dependency can be from a class to a class or from a class to an individual static attribute or operation.
2. Open the Features dialog box for the dependency you created, and apply the *StaticImport* stereotype to it.

When you next generate code, the code for the dependant class will contain the appropriate static import statement.

Reverse Engineering / Roundtripping and Static Import Statements

If you reverse engineer code that contains static import statements, Rhapsody will create dependencies that have the **StaticImport** stereotype applied to them.

The roundtripping feature can handle the addition of static import statements to your code, as well as changes to static import statements, including switching regular import statements to static import statements, and vice versa.

If you delete static import statements from your code, the roundtripping behavior will depend upon the value of the property `JAVA_Roundtrip::Update::AcceptChanges`.

Code Generation Checks

If you create a **StaticImport** dependency between a class and an attribute/method that is not static, Rhapsody will generate the corresponding static import statement but it will issue a warning that you have specified a static import for a non-static class member.

Static Blocks

Java allows you to define blocks of code as *static*. Code within static blocks is executed only once, when the class is first loaded.

Rhapsody allows you to add static blocks to classes in your model, and generates appropriate code for such blocks.

Adding Static Blocks to Classes in a Model

To add a static block to a class:

1. Select the class in the browser and from the context menu, select **Add New > StaticBlock**. (Alternatively, select the class in an object model diagram, and from the context menu select **New StaticBlock**.)
2. Open the Features dialog box for the newly created static block, and on the **Implementation** tab enter the code for the body of the block.

Changing a Static Block to an Operation

Rhapsody makes it easy to switch a static block to an operation, and vice versa.

To change a static block to an operation:

1. Select the static block in the browser.
2. Open the context menu, and select **Change To > Primitive Operation**.

Note

When you change a static block to an operation, the operation created will be a static operation.

To change a primitive operation to a static block:

1. Select the operation in the browser.
2. Open the context menu, and select **Change To > Static Block**.

Reverse Engineering / Roundtripping and Static Blocks

If you reverse engineer code that contains static blocks, Rhapsody recognizes these blocks and adds them to the class in the model.

The roundtripping feature can handle the addition of new static blocks to your code, as well as changes to the body of a static block.

When making changes directly to code within a static block, keep in mind that when adding code to the body of a static block, the new code will be roundtripped into the model only if you placed the code between the Rhapsody annotations inside the block.

If you delete static blocks from your code, the roundtripping behavior will depend upon the value of the property `JAVA_Roundtrip::Update::AcceptChanges`.

Generating JAR Files

In RiJ, you have the option of specifying that Rhapsody should generate a JAR file when you build your project.

To specify that a JAR file should be created as part of the build process:

1. Open the Features dialog box for the relevant configuration.
2. On the **Settings** tab, select the **Generate JAR File** option.

The JAR file generation mechanism is controlled by the following properties (under `JAVA_CG::Configuration`):

- ◆ `JarFileGenerate` is a Boolean property that determines whether or not a JAR file will be generated as part of the build process. The value of this property is controlled by the **Generate JAR File** option on the **Settings** tab of the Features dialog box for configurations.
- ◆ `JarFileGeneratorCommand` specifies the jar command that should be carried out if the property `JarFileGenerate` has been set to `Checked`.

Java 5 Annotations

Rhapsody in Java (RiJ) supports the concept of the Java 5 annotation through modeling and code generation. Java users can use Java annotations to model and generate code for all key Java 5 concepts. You can create annotations within the Rhapsody environment and then generate the annotations within the generated code.

Note the following about Java annotations:

- ◆ They provide data about the program but do not affect the program itself.
- ◆ They can be used by:
 - compilers
 - documentation tools
 - code analysis tools
 - deployment tools
 - run-time analysis tools
- ◆ They can be applied on any kind of program element (for example, class, field, method, enum, and so forth).

To add a JavaAnnotation to a model element, you must do the following general steps:

1. Create a JavaAnnotation type; see [Creating a JavaAnnotation type](#).
2. Add the JavaAnnotation and assign values to the annotation's elements; see [Using a JavaAnnotation type](#).
3. Add the annotation to one or more Java model elements using a dependency with a AnnotationUsage relationships; see [Using a JavaAnnotation within a model](#).

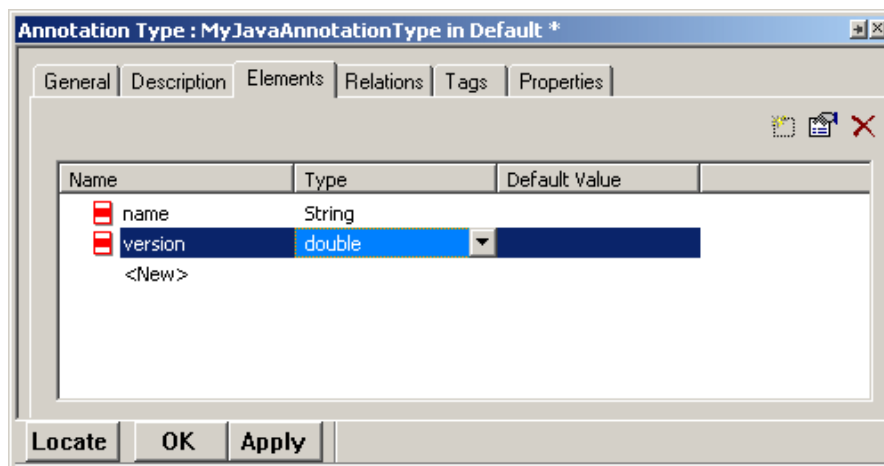
Creating a JavaAnnotation type

Java 5 annotations are modeled similar to the way classes and objects are modeled. This means you must define this type of annotation before you can use it.

To create a Java annotation type, follow these steps:

1. Open your project in RiJ, right-click a package or class on the Rhapsody browser, and select **Add New>Annotation Type**.
2. Type a name for your new annotation type.
3. Double-click the annotation type. The Features dialog box opens.
4. On the **Elements** tab, add any legal JDK 5 data type, as shown in the following figure.
 - a. Click <<New>> and type a name for the element.
 - b. Select a type from the **Type** drop-down list.
 - c. Enter a default value if necessary.

Note: The rows on this tab relate to annotation elements.

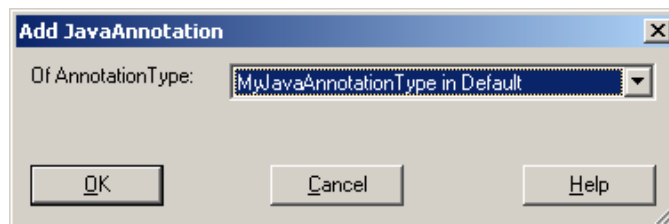


5. Click **OK**.

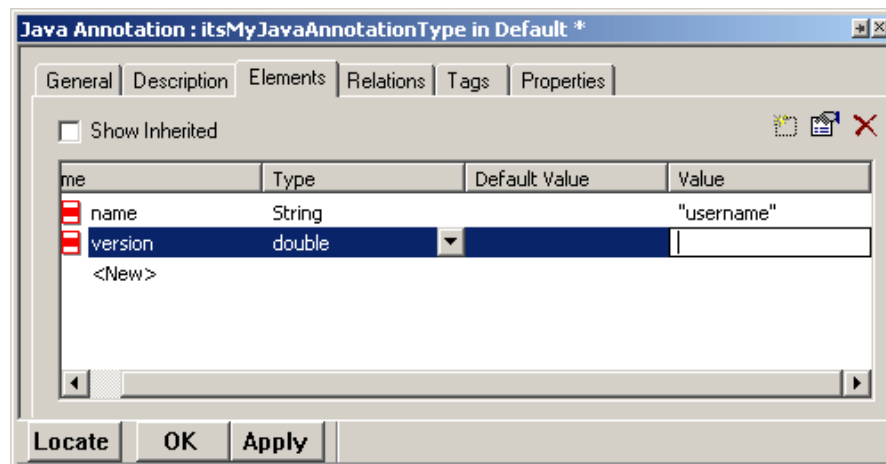
Using a JavaAnnotation type

To use a JavaAnnotation type, you must create a JavaAnnotation that is of the type you want. To do so, follow these steps:

1. Open your project in RiJ, right-click a package on the Rhapsody browser, and select **Add New>JavaAnnotation**. The Add JavaAnnotation dialog box opens.
2. Select the JavaAnnotation type you want from the drop-down list, as shown in the following figure:



3. Click **OK**.
4. Double-click the JavaAnnotation on your Rhapsody browser. The Features dialog box opens.
5. On the **Elements** tab, enter specific values for the JavaAnnotation, as shown in the following figure:



6. Click **OK**.

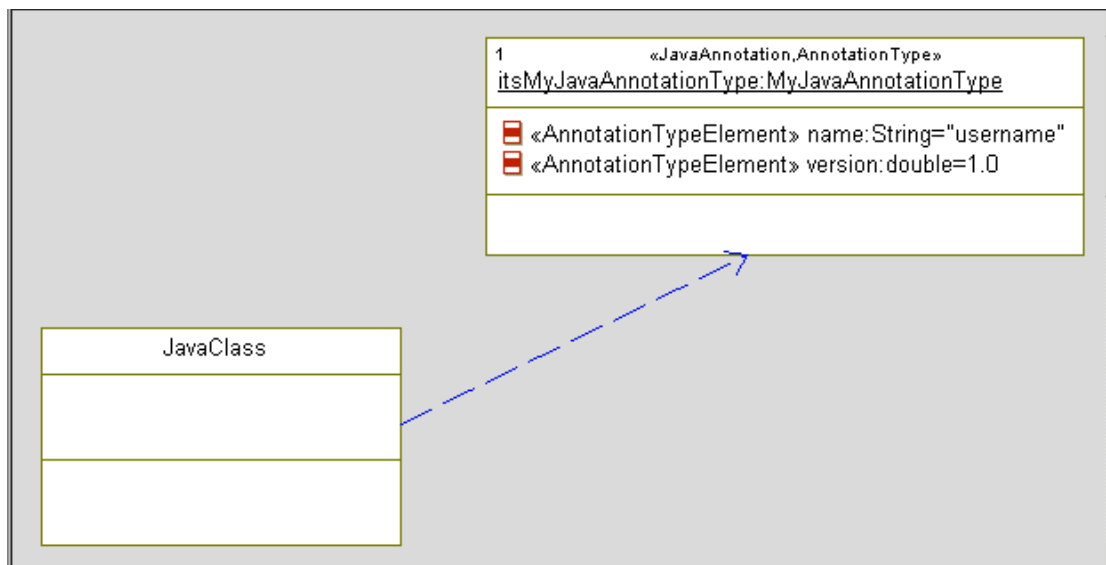
Using a JavaAnnotation within a model

To have classes in your Java design utilize a JavaAnnotation, follow these steps:

1. Open your project in RiJ.
2. On a diagram (for example, an object model diagram), drag your JavaAnnotation from the Rhapsody browser onto your diagram.

Notice that from a model perspective, the JavaAnnotation is shown with two stereotypes.

3. Create a new class on your diagram that represents a user-defined Java class in the system that wants to make use of the JavaAnnotation.
4. Draw a dependency from your class to your JavaAnnotation. Your diagram should resemble something like the following figure:



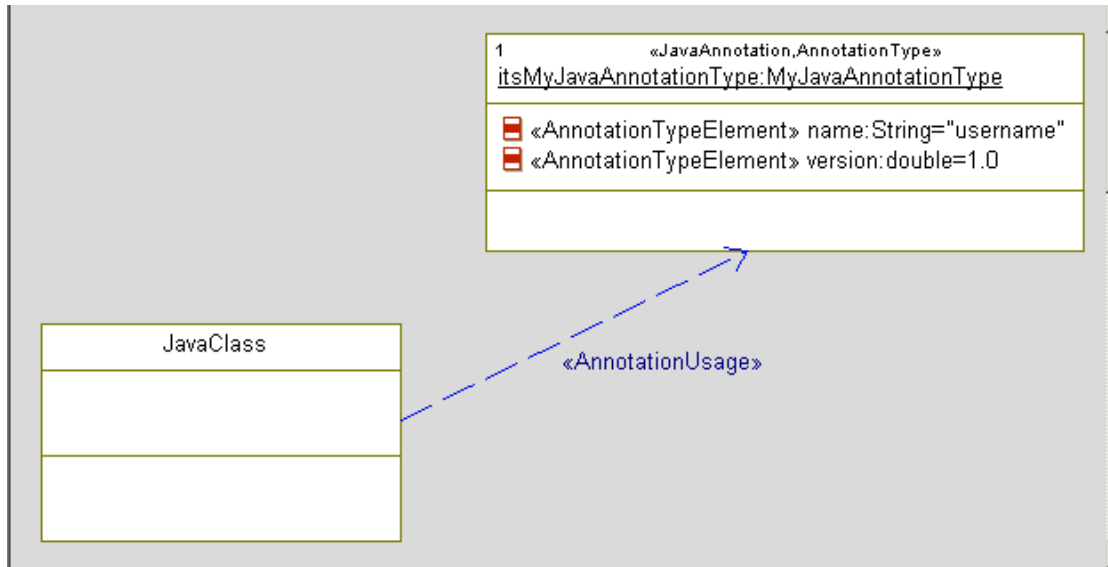
5. Double-click the Dependency link. The Features dialog box opens.
6. On the **General** tab, from the **Stereotype** drop-down list, select **AnnotationUsage in PredefinedTypesJava**.

Using this stereotype ensures that the code is generated correctly.

Note: **AnnotationUsage** appears in the **Stereotype** box.

7. Click **OK**.

Your diagram should resemble something like the following figure:



The following figure shows how the annotation is utilized within the Java class shown above:

```

@MyJavaAnnotationType (
    name = "username",
    version = 1.0
)
/// class JavaClass
public class JavaClass {

    // Constructors

    /// auto_generated
    public JavaClass() {
    }

}
File Path : DefaultComponent/DefaultConfig/Default/JavaClass.java

```

Code Generation and Java 5 Annotations

The code generator interprets the terms `AnnotationType`, `JavaAnnotation` and `AnnotationUsage` to print the corresponding Java code.

In addition, Java annotations of some element can be added as text to the `JAVA_CG::<ElementType>::JavaAnnotation` property. The code generator will print the property content before element declaration.

Reverse Engineering and Java 5 Annotations

To assure reverse engineering works correctly for Java annotations, note the following:

- ◆ To control the behavior of reverse engineering, use the `JAVA_ReverseEngineering::ImplementationTrait::ImportJavaAnnotation` property. The following values are available for this property:
 - **None.** All code parts related to Java Annotation are ignored
 - **Model.** Java annotations are imported as model elements (`AnnotationType`, `JavaAnnotation` and `AnnotationUsage`)
 - **Verbatim.** Java annotation Usage is imported as a verbatim text to `JavaAnnotation` property of the corresponded element. `AnnotationTypes` are imported as model elements. This is the default value.
 - **Mixed.** Java annotation are imported as model elements; if this fails, usage will still be imported as a verbatim text to `JavaAnnotation`.
- ◆ Specify a correct `CLASSPATH` for reverse engineering is important for the correct result of Reverse Engineering of Java Annotations.
- ◆ When you use both schemes of Reverse Engineering of Java Annotations (Model and Verbatim) the code generated for the model created by RE should be the same as in the original code (semantically).
- ◆ Roundtrip completely ignores all code parts related to Java annotations. Nothing should be changed.

Limitations for Java 5 Annotations

Note the following limitations:

- ◆ There is no roundtripping of Java annotations.
- ◆ Java annotations of events are not supported
- ◆ Predefined Java annotations are not supported.
- ◆ There is no **Default Value** field on the Features dialog box for the annotation element (double-click the annotation element on the Rhapsody browser to open this Features dialog box). Instead, you can add it on the **Elements** tab of the Features dialog box for the Annotation type.
- ◆ Java annotations of constructor, destructor (finalize), and associations are not generated by the code generator. Instead, use the `JAVA_CG:Operation::JavaAnnotation` and `JAVA_CG:Relation::JavaAnnotation` properties.
- ◆ Unnamed types in Java annotations are not supported (without “element = ...”), for example: “@Retention(RetentionPolicy.RUNTIME)”. Instead, use the `JAVA_CG:Operation::JavaAnnotation` property.

Java Reference Model

Rhapsody includes a reference model for the classes contained in Java SE 6.

When you create a new project in RiJ, you can add this model to your project as a reference.

The Java reference model can be found in the directory `<Rhapsody installation path>\Share\LangJava\JDKRefModel`.

Systems Engineering with Rhapsody

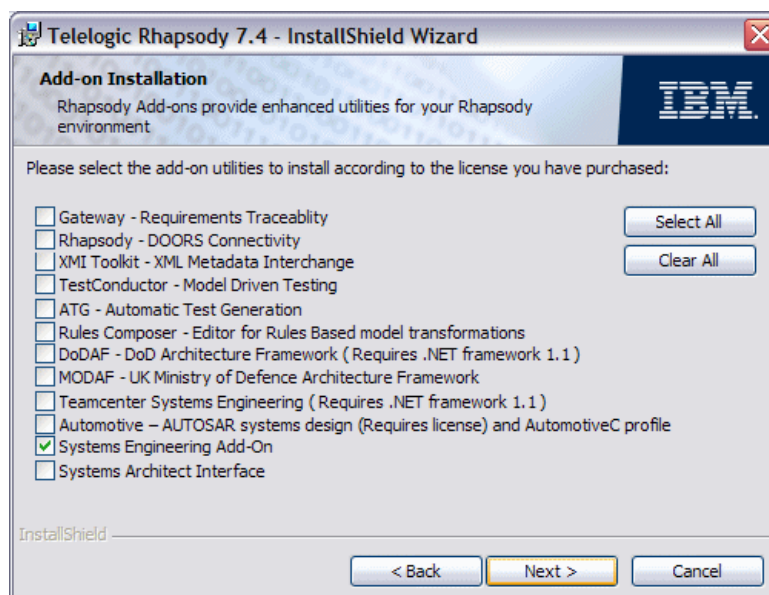
Rhapsody allows systems engineers to capture and analyze *requirements* quickly and then design and validate system behaviors. A Rhapsody systems engineering project includes the UML and SysML diagrams, packages, and simulation configurations that define the model. Systems engineers may use the [SysML Profile Features](#) or the [Harmony Process and Toolkit](#) to guide system development through its iterative development process.

For more information the Rhapsody Harmony process, refer to the *Harmony Deskbook* by Hans-Peter Hoffmann, Ph.D., 2008 on the [Telelogic Rhapsody Support Site](#).

Installing and Launching Systems Engineering

Follow the instructions in the *Rhapsody Installation Guide* to install the development environments you need. Rhapsody's Systems Engineering add-on requires these extra installation steps and start-up steps:

1. When the **Add-on Installation** window displays. Check the **System Engineering Add-On** option, as shown in the example below.




2. Click **Next** and complete the installation as instructed.
3. When you want to use the systems version of Rhapsody, select the Windows **Start > Programs > Telelogic > Telelogic Rhapsody** (version) > (a developer's edition or Rhapsody System Designer edition) > **Rhapsody**.

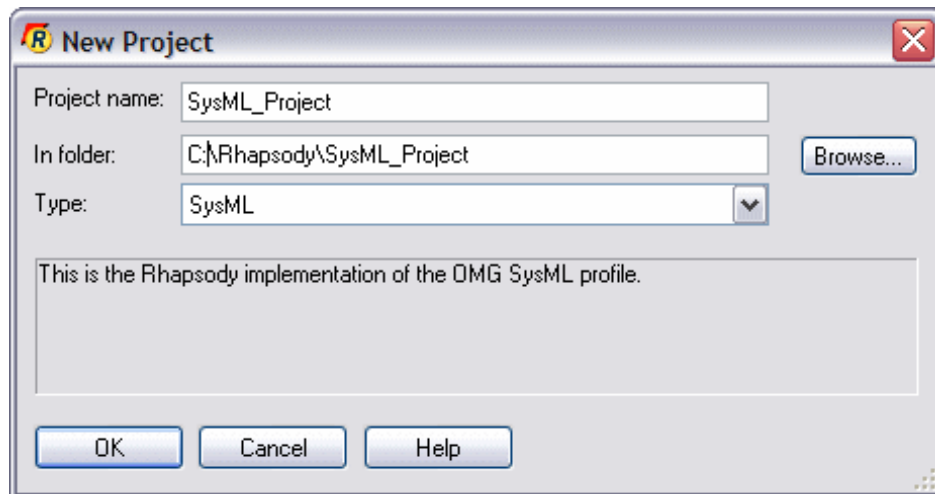
This installation makes the systems engineering features available to support the UML and SysML standards in these specifications:

- ◆ Check the URL for the [UML specification](#)
- ◆ Check the URL for the [SysML specification](#)

Creating a SysML Profile Project

To create a new systems engineering project using the [SysML Profile Features](#), follow these steps:

1. Start Rhapsody.
2. Click the **New** icon  in the main toolbar or select **File > New**.
3. In the **Project name** box, type your project name (`SysML_Project` in the example below).
4. In the **In folder** box, enter the directory in which the new project will be located, or click the **Browse** button to select the directory.
5. In the **Type** box, select the `SysML` profile (shown below) so that you can use the SysML modeling language and the systems engineering diagrams.



6. Click **OK**. Rhapsody verifies that the specified location exists. If it does not, Rhapsody asks whether you want to create it.
7. Click **Yes**. In this example, Rhapsody creates a new project in the selected subdirectory, opens the project, and displays the browser in the left pane.

Note

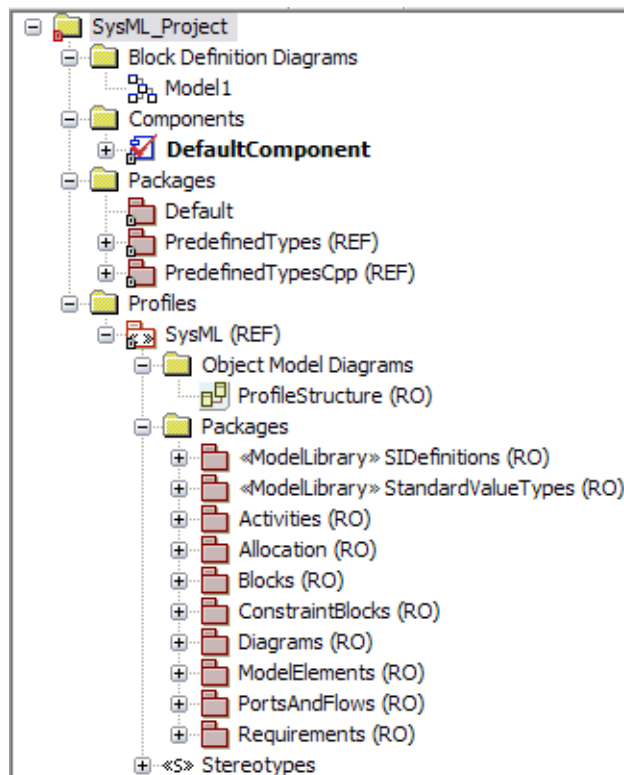
If the browser does not display, select **View > Browser**.

SysML Profile Features

When you select the SysML profile for your project, Rhapsody provides a starting point with a blank Block Definition Diagram (named `Model1`), packages, and predefined types, as shown in [SysML Profile Elements](#). This profile is Rhapsody's implementation of the [OMG SysML Specification](#). The Rhapsody SysML profile provides this additional functionality for your model:

- ◆ SysML enhancements to standard UML diagrams including the Use Case, Requirements, Activity, Sequence diagrams and Statecharts
- ◆ SysML's Block Definition, Internal Block, and Parametric diagrams
- ◆ [Architectural Design Wizard](#) and [Link Wizard](#)
- ◆ XMI 2.1 support

SysML Profile Elements



The SysML profile also contains default and predefined packages and a read-only ProfileStructure object model diagram for you to use as reference of the available Rhapsody SysML features in the profile.

Note

The items listed under `Profiles` in the browser are not intended to be used as part of a working model. They are for information purposes only.

When you create a new project, Rhapsody creates a directory containing the project files in the specified location. The name you choose for your new project is used to name project files and directories, and appears at the top level of the project hierarchy in the Rhapsody browser. Rhapsody provides several default elements in the new project, including a default package, component, and configuration.

SysML Profile Packages

The following Rhapsody packages (shown in [SysML Profile Elements](#)) are available when you select the SysML profile for a new project:

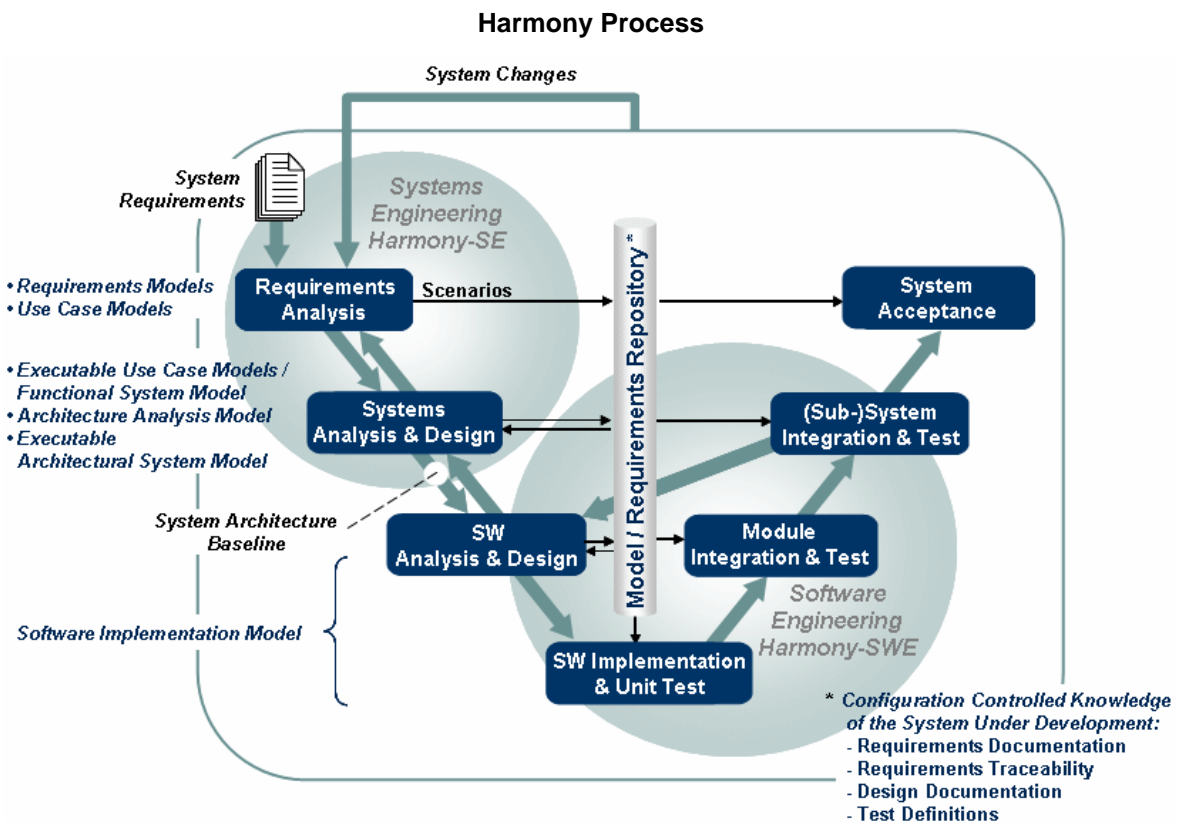
- ◆ <<ModelLibrary>> **SIDefinitions** contains these read-only packages: BaseSIUnits and DerivedSIUnits.
- ◆ <<ModelLibrary>> **StandardValueTypes** contains read-only Complex and Real value types. See [Block Definition Diagram Drawing Tools](#) for more information about valueTypes.
- ◆ **Activities** stereotypes support the SysML expansion of the activity diagram behaviors and links to the blocks that contain the behaviors.
- ◆ **Allocation** contains read-only stereotypes and table layouts.
- ◆ **Blocks** include stereotypes that represent the system capabilities in the model.
- ◆ **ConstraintBlocks** contains the stereotypes that control the relationships of blocks: ConstraintBlock, ConstraintParameter, ConstraintProperty, and valueBinding.
- ◆ **Diagrams** lists the stereotypes needed to support these SysML diagrams: Block Definition, Internal Block, Parametric, and Requirements.
- ◆ **ModelElements** lists the stereotypes needed to support these diagram elements:
 - conform
 - problem
 - rationale
 - refine
 - view
 - viewpoint with its tags
- ◆ **PortsandFlows** show how items flow between blocks and parts. Ports are the connection points between blocks or parts and their environments. Ports are often reused and have clearly defined interfaces. These are most often used in [Activity Modeling in SysML](#).
- ◆ **Requirements** contains the stereotypes to display model conditions that must be met with the finished product. This profile element also includes read-only requirement table layouts.

Harmony Process and Toolkit

The Harmony process facilitates a seamless transition from systems engineering to software engineering. It uses SysML exclusively for system representation and specification. Harmony, a scenario-driven process, is iterative and promotes reuse of test scenarios throughout system development, as shown in the [Harmony Process](#) diagram.

Harmony Process Summary

The Harmony process models allow systems engineers to find design errors early in the development when the cost of correcting them is lower. Customer requests can be more efficiently assessed, incorporated, and given timely feedback. However, the greatest benefit of a model-driven process is improved communication, not only between the engineering disciplines, but also among the technical and non-technical parties involved in the system development process. This is possible because models can represent different levels of abstraction and, therefore, avoid the information overload that often occurs when data is passed among the participating groups.



The Harmony process can be used in any systems engineering project. The key objectives of these projects are the following:


- ◆ Derive required system functionality
- ◆ Identify system states and modes
- ◆ Allocate requirements and functionality to identified subsystems

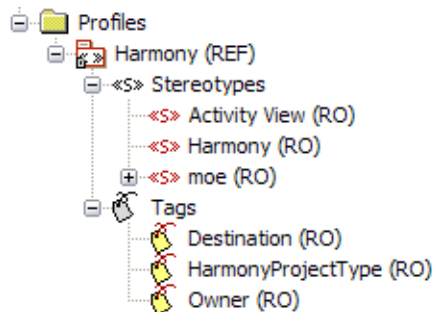
These key objectives can be met with UML's structure diagrams and the following SysML diagrams:

- ◆ Use Case diagrams
- ◆ Sequence diagrams
- ◆ Activity diagrams
- ◆ Statecharts
- ◆ Block Definition diagrams
- ◆ Internal Block diagrams
- ◆ Parametric diagrams

Creating a Harmony Project

Though the Harmony process can be used in any systems engineering project, Rhapsody provides a special profile to make it easier to use this process in a project. To create a Harmony project for systems engineering, follow these steps:

1. Start Rhapsody.
2. Click the **New** icon  in the main toolbar or select **File > New**.
3. In the **Project name** box, type your project name.
4. In the **In folder** box, enter the directory in which the new project will be located, or click the **Browse** button to select the directory.
5. In the **Type** box, select the Harmony profile so that you can use the Harmony wizards and other systems engineering features.
6. Click **OK**. Rhapsody verifies that the specified location exists. If it does not, Rhapsody asks whether you want to create it and generates a starting point for your Harmony project.
7. Check the Profiles folder in the browser to be certain that Harmony is listed with the stereotypes and tags, as shown here.



The <<moe>> or measures of effectiveness stereotype supports the trade analysis feature. See [Special Harmony Menu Options](#) and [Performing a Trade Analysis](#) for more information.

Harmony Toolkit

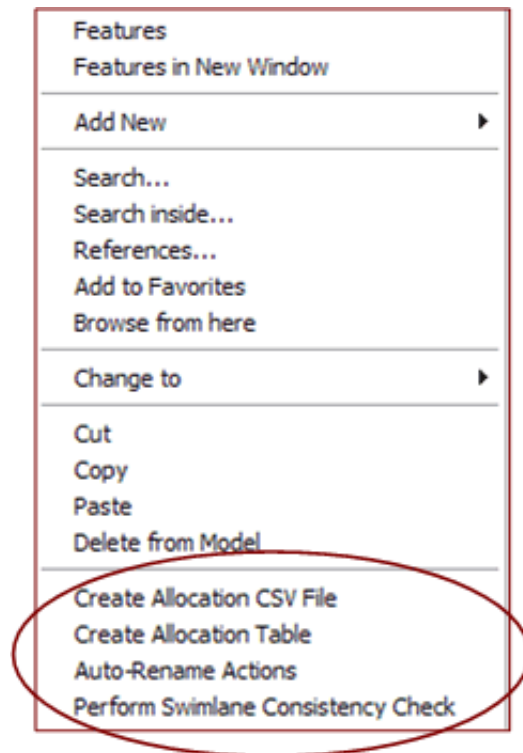
The Rhapsody systems engineering features includes the following automated tools for *Harmony profile* projects only:

- ◆ Right-click menu options to perform common tasks quickly
- ◆ Wizards that perform repetitive tasks automatically or reduce the number of steps required to perform a systems engineering operation

Special Harmony Menu Options

Follow these steps to access the special systems engineering menu options:

1. Click an item in the browser that has a special option available. (See the [Special Harmony Menu Options Chart](#).)
2. Right-click to display the menu with the special systems engineering menu options at the bottom of the menu (circled in the example below). This example shows the right-click menu displayed when an *activity diagram* is selected.



3. Click the menu option to perform the task. The following table lists the special menu options by the browser items used to access them and describes the operations that the menu options perform.

Special Harmony Menu Options Chart

Accessible from Browser Item	Menu Option	Systems Engineering Operation Description
Activity Diagram	Auto-Rename Actions	Renames all of the actions in an activity diagram to be the actual action body text used
Activity Diagram	Create New Scenario from Activity Diagram	Creates a new sequence diagram from an existing activity diagram with the swimlanes converted to life lines
Activity Diagram	Perform Swimlane Consistency Check	For every Swimlane that is represented by an object/part, it ensures the following: <ul style="list-style-type: none"> • Each action has a corresponding operation • Each operation has a corresponding action
Activity View	Duplicate Activity View	Existing activity view is copied using the name of the view appended with “_copy”
Activity View	Perform Activity View Consistency Check	Displays a dialog box that lists the following: <ul style="list-style-type: none"> • Operations in the activity diagram that do not appear in any of the project's sequence diagrams • Errors found in the referenced scenarios <p>The results displayed in this dialog box can be copied into the Clipboard. There is also a Recheck button.</p>
Activity View and Activity Diagram	Create Allocation CSV File	<ul style="list-style-type: none"> • Creates a CSV file based on the Swimlane Allocation • CVS file is added to Rhapsody as a controlled file and can be viewed in Rhapsody
Activity View and Activity Diagram	Create Allocation Table	Creates an Excel Spreadsheet based on the Swimlane Allocation (requires <i>Microsoft Excel</i> to be installed)
Activity View and Use Case	Create Simulation	Create an animated version of the highlighted activity view
Block Definition Diagram	Perform Trade Analysis	<i>Microsoft Excel</i> must be installed for this feature. The selected diagram must contain blocks/classes that represent the “solution” classes, i.e., those classes that aggregate the potential solutions with contained measures of effectiveness--<<moe>> stereotype. For more information, see Performing a Trade Analysis .

Accessible from Browser Item	Menu Option	Systems Engineering Operation Description
Class/Block	Copy MOEs from Base	Copies attributes stereotyped <<moe>> or measure of effectiveness from any parent classes/blocks to all classes/blocks that inherit from it. For each attribute, this option copies the "weight" tag value.
Class/Block	Copy MOEs TO Children	Exhibits the same behavior as the Copy MOEs from Base option but in the other direction (i.e., from the Base Class/block to any that inherit from it). For each attribute, this option copies the "weight" tag value.
Internal Block Diagram	Generate N2 Matrix	Generates an N2 Matrix from a Block Diagram and its associated Ports/ Interfaces/Links
Object or Class	Create Test Architecture	Uses the <i>TestingProfile</i> to generate a <i>Test Context Diagram</i> for the selected block or class and displays the test messages in the Log output window
Object or Class	Create Test Bench	Creates a Test Bench style Statechart for the current Actor
Sequence Diagram	Create Ports And Interfaces	This option may create any of the following depending on what was selected in the browser and available in the model at this point: <ul style="list-style-type: none"> • Creates new interfaces in the InterfacesPkg to hold them • Creates ports on structural blocks involved • Populates Ports with Interfaces • Makes Ports behavioral • Makes any "internal" operations private • Moves Event Declarations to the InterfacesPkg • Errors are flagged and highlighted in red on the sequences
Sequence Diagram	Report Unrealised Messages	Provides the user with a dialog showing a list of messages on a sequence diagram which are not yet realized
Use Case	Create Use Case Scenario	Creates a sequence diagram based on a selected use case diagram

Performing a Trade Analysis

For block definition diagrams, you may generate a weighted methods *Microsoft Excel* spreadsheet from the selected diagram, as shown in this example.

A	B	C	D	E	F
		NikonD70		NikonD300	
	<i>weight</i>	value	WV	value	WV
Battery.cost	0.3	5	1.5	4	1.2
Battery.life	0.2	3	0.6	7	1.4
Battery.capacity	0.5	6	3	4	2
Body.cost	0.4	5	2	6	2.4
Body.weight	0.6	4	2.4	7	4.2
			9.5		11.2

To generate a trade analysis, follow these steps:

1. In the browser, highlight a block definition diagram.

Note: The selected diagram must contain blocks/classes that represent the “solution” classes (i.e., those that aggregate the potential solutions with contained MOEs (measures of effectiveness)).

2. Right-click and select the **Perform Trade Analysis** option. The system generates the spreadsheet or displays a message indicating that the selected diagram is not appropriate for a trade analysis.

If the selected diagram is appropriate, the list of potential solutions is converted into a spreadsheet in a standard weighted methods format. The cells are populated with values, weights, and formulas to calculate totals.

Architectural Design Wizard

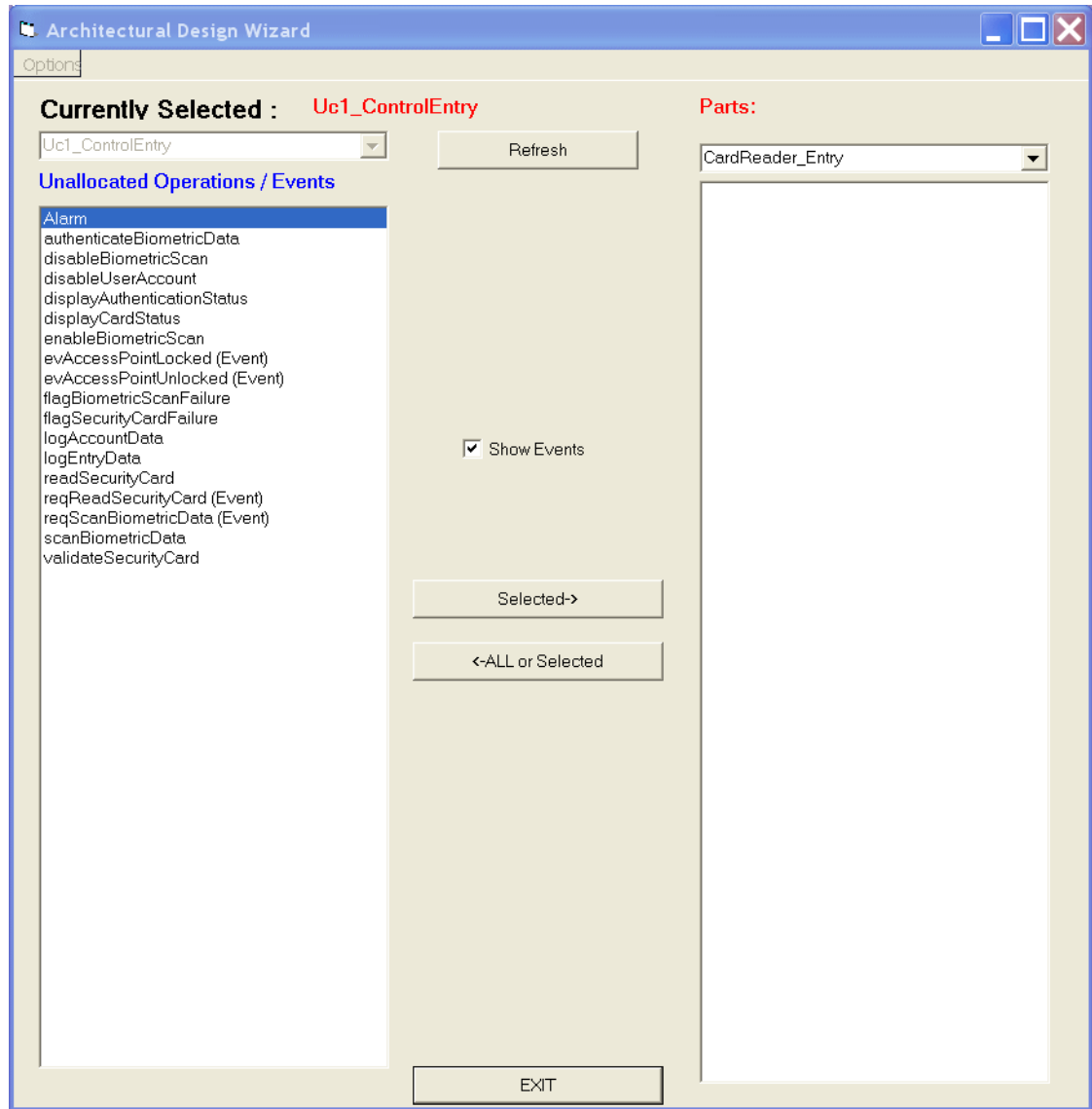
For both the Harmony and SysML profiles, the Architectural Design Wizard is available to copy operations from one architectural layer to another. This allows allocation of operations / events from a parent block to its children by copying and tagging them and provides these features:

- ◆ Allocate the operations (including the documentation and requirement relationships) to their respective systems
- ◆ Track when operations are allocated
- ◆ Allow multiple allocations

To copy unallocated operations/events to specific subsystems, follow these steps:

1. Highlight the block you want in the browser, and select **Tools > Architectural Design Wizard**.
2. If the selected system element does not immediately display, click the **Refresh** button at the top of the dialog box. If you want the events in the selected operation to be available for selection, check the **Show Events** box.
3. Use the pull-down menu above the **Parts** column to select the desired subsystem.

4. Highlight the **Unallocated Operation(s) / Event(s)** from the list on the left that you want to allocate to a **Part** listed on the right..



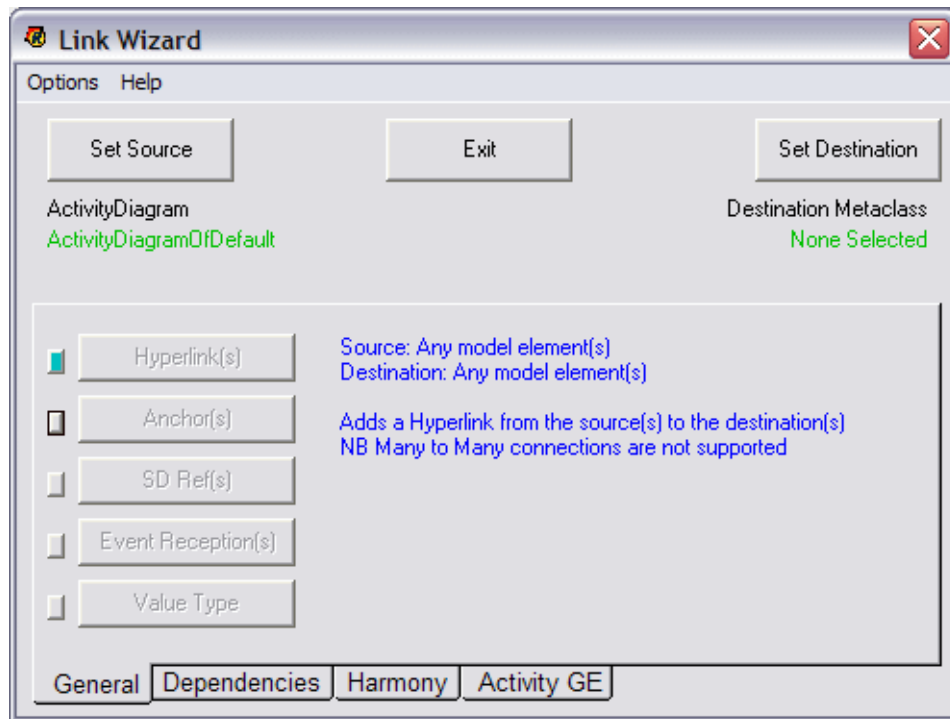
5. Click the **Selected** button between the two columns to move the selected item or items to the right-hand column.
6. If you change your mind and want to return items you put into the **Parts** column back to the **Unallocated** column, highlight individual items in the right column and click the **All or Selected** button. To return all items to the left side quickly, do not select any items and click the **All or Selected** button.
7. Click **Exit** to close the wizard's dialog box.

Link Wizard

For both the Harmony and SysML profiles, the Link Wizard can be used with sequence and activity diagrams to set links between a source and a destination. It allows you to select multiple sources or multiple links and create any type of dependency between them.

To use the automatic Link Wizard, follow these steps:

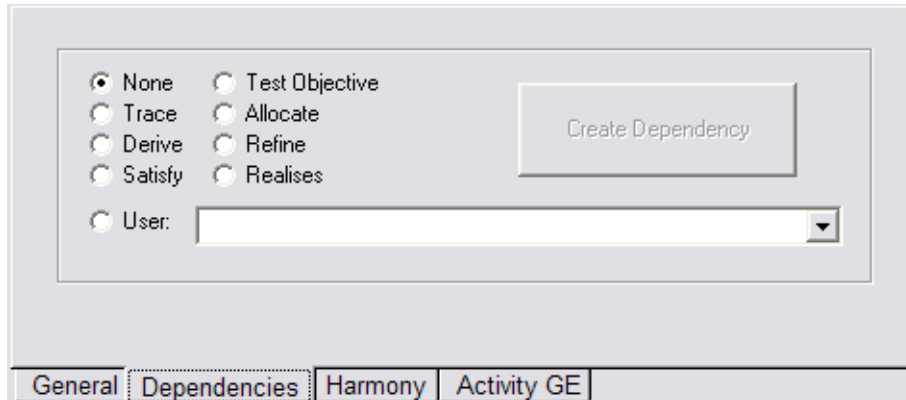
1. Display the sequence or activity diagram to which you need to add links.
2. Select an element in the diagram that is the starting point for the link.
3. Select the **Tools > Link Wizard** menu option.
4. Click the **Set Source** button and the name of the selected element displays.



5. In the diagram, select the destination elements and then click the **Set Destination** button in the Link Wizard. The wizard allows these types of links:
 - ◆ **Hyperlink(s)**
 - ◆ **Anchor(s)**
 - ◆ **SD Ref(s)**
 - ◆ **Event Reception(s)**
 - ◆ **Value Type**

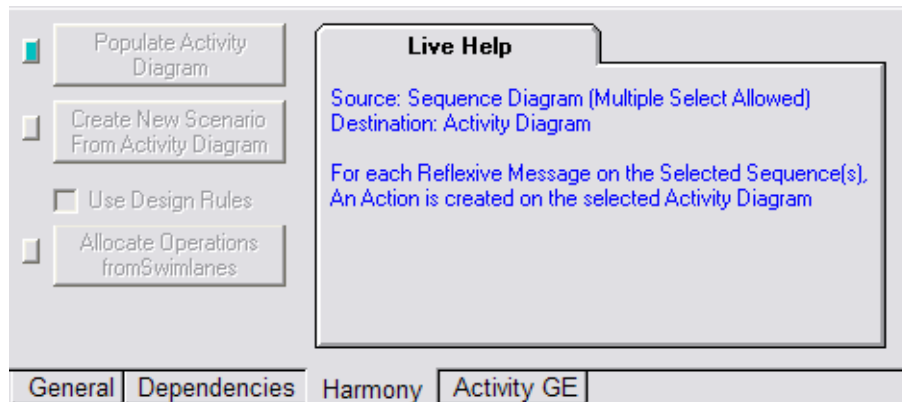
A detailed description of each link type displays when you position the cursor over the small button to the left of the Command button, as shown in the example above. If a button is greyed out, that type of link is not appropriate for the selected Source and Destination. Click the button for the type of link you want to add.

- Click the **Dependencies** tab to create a stereotype of the selected type automatically. For a **User** defined stereotype, click **User** and type the name. Click **Create Dependency** when you have made your stereotyping selections.



- Click the **Harmony** tab to select one of these options:
 - ◆ **Populate Activity Diagram**
 - ◆ **Create New Scenario from Activity Diagram** (determine whether or not to use the Design Rules)
 - ◆ **Allocate Operations from Swimlanes**

A detailed description of each Harmony command displays in the **Live Help** area when you position the cursor over the small button to the left of the Command button.



8. Click the **Activity GE** tab to select either a Reference Activity or a Swimlane Reference.
9. Click **Exit** at the top of the Link Wizard to perform the selected operations from the source to the destination.

Note

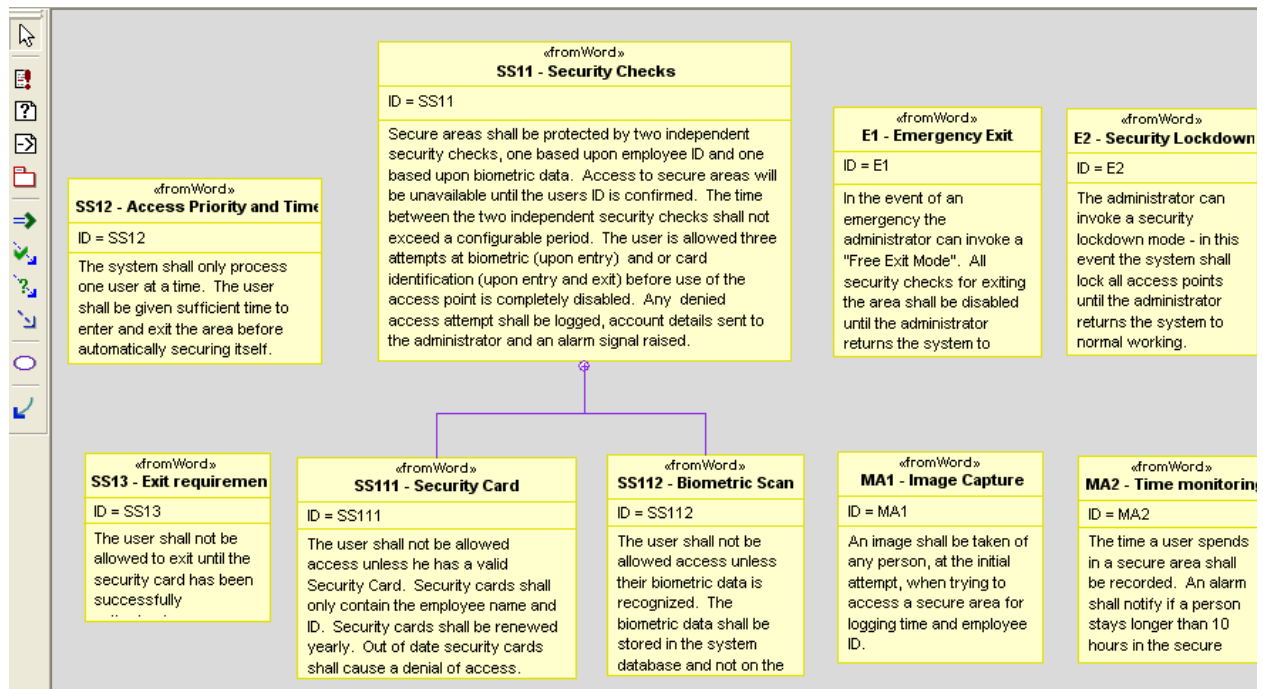
To perform this task manually, see the instructions in the [Systems Engineering Requirements in Rhapsody](#) section.

Systems Engineering Requirements in Rhapsody

The Gateway product is often used to define specific requirements to support your analysis. This add-on product allows Rhapsody to hook up seamlessly with third-party requirements and authoring tools for requirements traceability. In addition, it describes importing requirements using the Rhapsody Gateway and using *use case diagrams* (UCDs) to show the main system functions and the entities that are outside the system (actors). The use case diagrams specify the requirements for the system and demonstrate the interactions between the system and external actors. See the [Link Wizard](#) to use the automatic linking feature.

You may also use other manual methods and software to create requirements diagrams in Rhapsody such as importing them from *DOORS* or *Microsoft Word*, as in the example below.

Requirements Diagram with Requirements Imported from Word



Analysis and Requirements using Gateway

When Rhapsody Gateway analyzes your project information including requirements, documents, and database modules, the software provides the following analysis results:

- ◆ Navigation features between Rhapsody Gateway and interfaced tools
- ◆ Requirements captured at high level accessible in an authoring tool
- ◆ Filter capabilities for more targeted display and results for reports
- ◆ Requirements traceability graph
- ◆ A list of elements violating default rules and customized rules
- ◆ Additional information within the Rhapsody Gateway environment including attributes, links, and text

Importing Gateway Requirements into Rhapsody

You can use the Rhapsody Gateway product to define specific requirements to support your analysis.

This add-on product allows Rhapsody to hook up seamlessly with third-party requirements and authoring tools for complete requirements traceability. The Rhapsody Gateway includes the following features:

- ◆ Traceability of requirements workflow on all levels, in real-time
- ◆ Automatic management of complex requirements scenarios for intuitive and understandable views of upstream and downstream impacts
- ◆ Creates impact reports and requirements traceability matrices to meet industry safety standards
- ◆ Connects to common requirements management/authoring tools including DOORS, Requisite Pro®, Word®, Excel®, Powerpoint PDF®, ASCII, FrameMaker, Code and Test files
- ◆ A bidirectional interface with the third-party requirements management and authoring tools
- ◆ Monitoring of all levels of the workflow, for better project management and efficiency

Gateway Documentation

The following Gateway documentation is accessible from the Rhapsody List of Books page (choose **Help > List of Books**).

- ◆ *Gateway User's Manual*
- ◆ *Gateway Customization Guide*

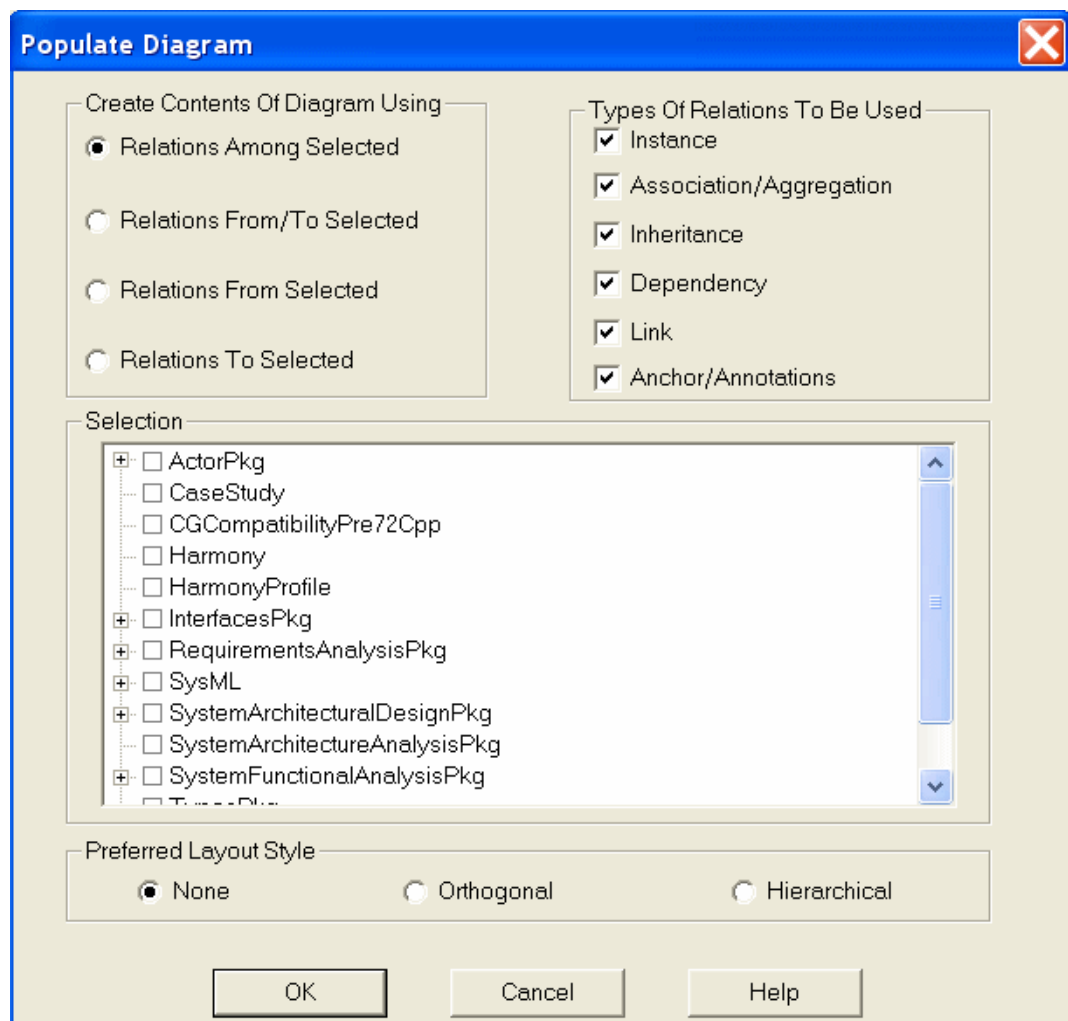
Limitations

This is a limitation in Rhapsody Gateway. If you have multiple Rhapsody projects configured for Gateway, when you right-click and select **Reload** in Gateway, it synchronizes with the active opened Rhapsody project, not the selected project. This creates incorrect data. For example, if you have a Radio and a Handset project configured in Gateway but Radio is set as the active project, you work on the Handset model and select the **Reload** command in Gateway. This would synchronize the Radio project, but not the Handset project because it is not your active project.

Creating Rhapsody Requirements Diagrams

In projects created with the SysML or Harmony profiles, you may use requirements diagrams, with requirements imported from other software products or created in Rhapsody, to communicate complex details to team members. To create a requirements diagram, follow these steps:











1. Highlight the **Requirements** package in the browser.
2. Right-click and select **Add New > Diagrams > Requirements Diagram**.
3. Enter the **Name** of the new diagram. Click **OK** if you want to add items individually to the requirements diagram. However, if you want the system to put existing model items into the new diagram, check the **Populate Diagram** box and select those items.
4. Select the characteristics and content for the new requirements diagram from the Populate Diagram dialog box. Click **OK**.



Requirements Diagram Drawing Tools

Use the drawing icons to show the relationships and functions to create a detailed requirements diagram, as shown in the [Requirements Diagram with Requirements Imported from Word](#).

Requirements Diagram Drawing Tools


Icon	Tool Name	Purpose
	Requirement	Definition of a system requirement
	Problem	Unfavorable environment situation that the requirement is meant to address or may encounter as a blocking condition
	Rationale	Statement of the reason for a specific requirement
	Create Package	Used to show the relationship between the package artifact and requirements
	Derivation	Indicates that a requirement was derived from another
	Satisfaction	Indicates an artifact or condition that is necessary to fulfill a specific requirement
	Verification	Indicates how a specific requirement is verified as supplied in the design
	Dependency	Indicates a dependent relationship between two items in the diagram
	Create Use Case	Used to show the relationship between a use case and a requirement
	Allocation	Indicates that an artifact or requirement is exclusively reserved for use of another

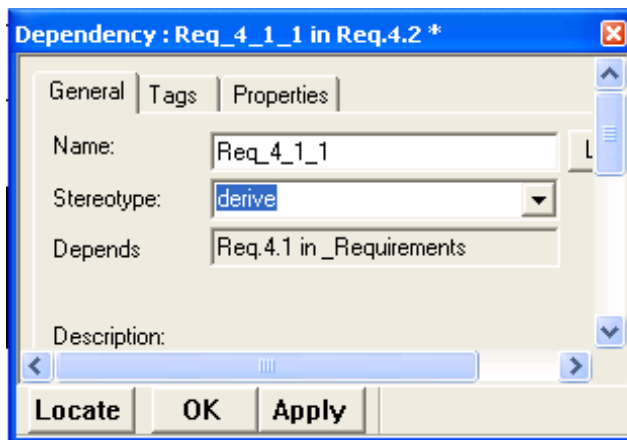
Drawing and Defining the Dependencies

With elements drawn in the requirements diagram, use a *dependency* to show a direct relationship in which the function of an element requires the presence of and may change another element. You can show the relationship between requirements, and between requirements and model elements using dependencies. You may set the following types of dependency stereotypes:

- ◆ **Derive** indicates a requirement that is a consequence of another requirement.
- ◆ **Trace** shows the dependency from the model element to its requirement with the dependency arrow head on the requirement.

To define the relationships between requirements with dependencies, follow these steps:

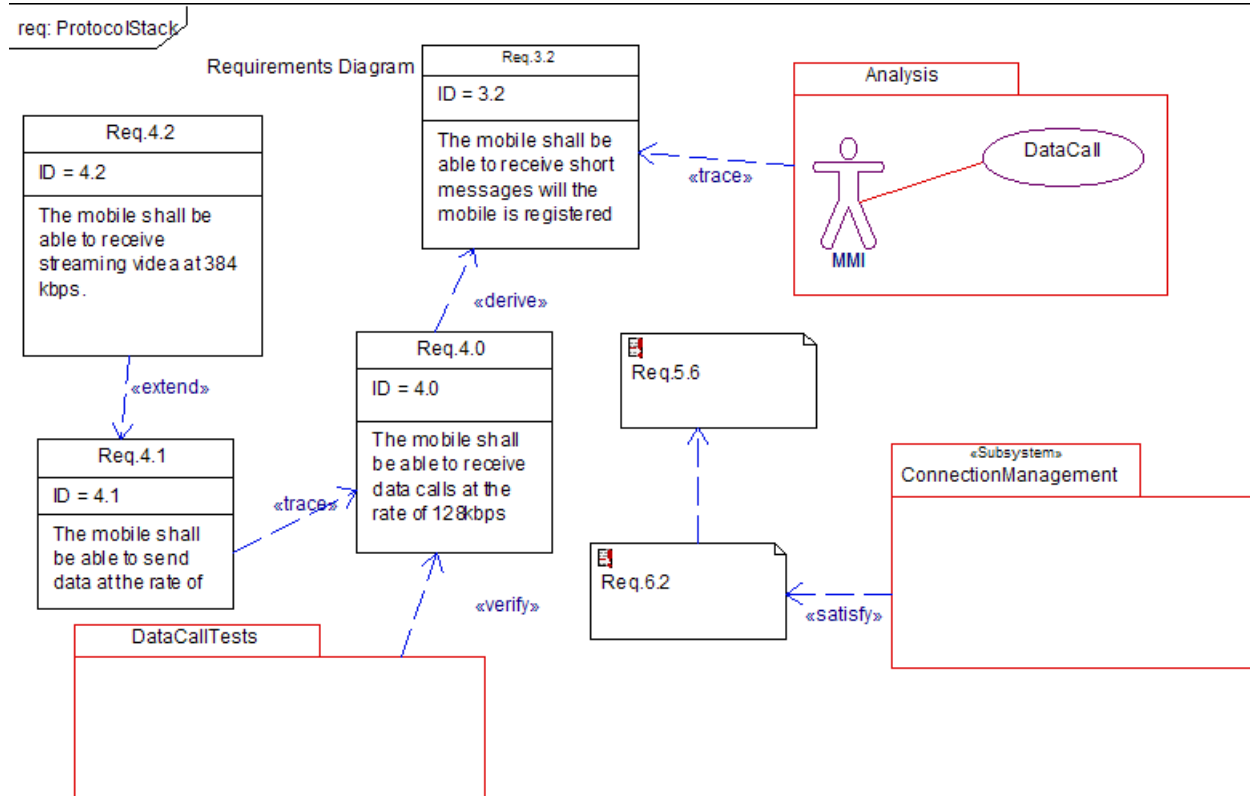
1. Click the **Dependency** icon  on the **Drawing** toolbar.
2. Draw a dependency line from one requirement to another. Right-click on the dependency line and select **Features** from the menu. At this point you may select `derive` as the **Stereotype**, as shown in this example, or another possible stereotype including trace, extend, refine, allocate, conform, decompose, satisfy, verify, valueBinding, Send, Usage, Friend, or <<New>>.



3. Click **OK** to save the change and close the dialog box. Rhapsody automatically adds the dependency relationships to the browser.

Creating Specialized Requirement Types

To specify specialized requirements types, use the sub-typing features of stereotypes (listed below). The following example of these stereotypes can be examined in the Rhapsody Samples directory in the SysMLHandset project.



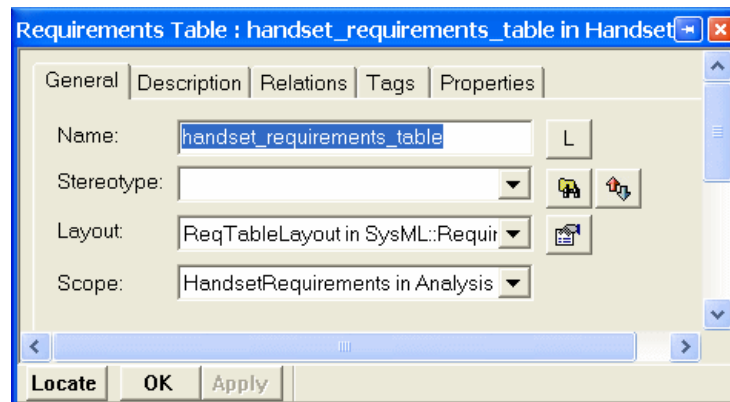
- ◆ **<<extend>>** shows that a requirement expands or provides more detailed view of another requirement. (See Req 4.2 and 4.1 in the example above.)
- ◆ **<<derive>>** shows a relationship between two requirements and supplies additional details. A derive requirement often reflects assumptions about the implementation of the system. (In the diagram, the arrow direction is from the derived to the original requirement.)
- ◆ **<<composite>>** requirements are the sub-requirements within the overall requirements hierarchy. This structure allows a complex requirement to be decomposed into its containing child requirements.
- ◆ **<<satisfy>>** relationship identifies the system or other model element intended to satisfy or fulfill the requirement. (In the diagram, the arrow direction is from the satisfying to the satisfied.)

- ◆ <<verify>> shows the relationship between a requirement and its test case. A test case is usually expressed as an activity or interaction diagram.
- ◆ <<refine>> relationship shows how a model element or set of elements further explains a requirement.
- ◆ <<trace>> requirement relationship provides a general purpose relationship between a requirement and any other model element. The semantics of <<trace>> do not include real constraints and, as a result, should not be used with any of the other requirements relationships listed previously.

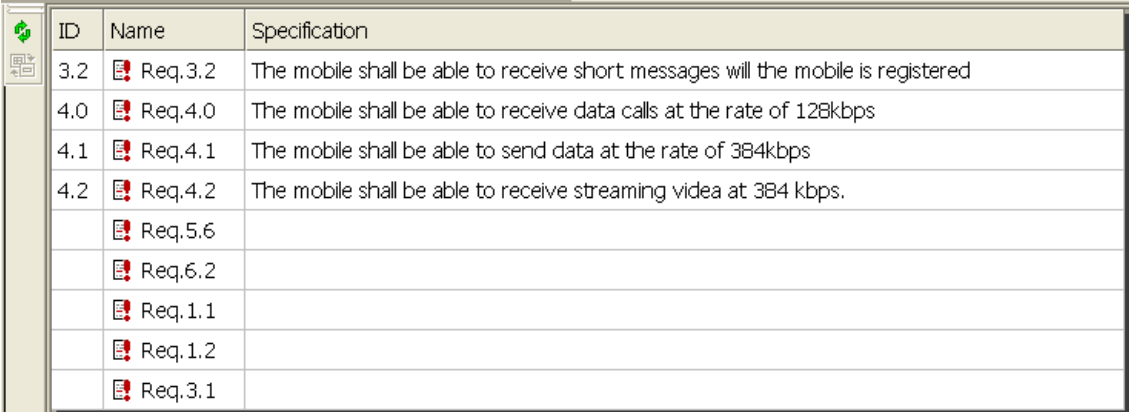
Requirements Tabular View

You may use the [Creating Table and Matrix Views](#) feature to create different layouts to view the requirements information in the project. However, you may also use the preformatted SysML requirements table by following these steps:

1. Right-click the `Requirements` package in the browser.
2. Select the **Add New > Requirements > RequirementsTable** option to access the predefined SysML requirements layout.
3. Right-click the generated name of the new table in the browser and select **Features**.
4. Enter the **Name** of the requirements table to be displayed in the browser list. The preformatted **Layout** is automatically listed, but you need to select the package to be analyzed in the **Scope**, as shown in this example from Rhapsody's `SysMLHandset` sample project. Click **OK**.



5. In the browser double-click the requirements table name to generate the table in the drawing area, as shown in this example.

A screenshot of a software interface showing a table of requirements. The table has three columns: ID, Name, and Specification. The rows contain various requirement IDs and their corresponding specifications. The table is displayed within a window that has a green refresh icon and a small icon in the top-left corner.

ID	Name	Specification
3.2	Req.3.2	The mobile shall be able to receive short messages will the mobile is registered
4.0	Req.4.0	The mobile shall be able to receive data calls at the rate of 128kbps
4.1	Req.4.1	The mobile shall be able to send data at the rate of 384kbps
4.2	Req.4.2	The mobile shall be able to receive streaming videa at 384 kbps.
	Req.5.6	
	Req.6.2	
	Req.1.1	
	Req.1.2	
	Req.3.1	

Creating Use Case Diagrams

Use case diagrams show the system main functions (use cases) and the entities (actors) outside the system. To start a use case diagram containing the actors and basic use cases, follow these steps:

1. In the browser, right-click the package containing your analysis, and select **Add New > Diagrams > UseCaseDiagram** from the menu. The New Diagram dialog box opens.
2. In the **Name** box replace the generated name with the name you want.
3. Check the **Populate Diagram** option if you want to select items in the project to place in the new diagram automatically.
4. Click **OK** to create the use case diagram as defined.


Rhapsody adds the `Use Case Diagrams` category in the browser and opens the new diagram. The [Use Case Diagram Drawing Toolbar](#) provides the drawing icons for these diagrams (shown in the following sections).

Note

For systems engineering purposes, the **Extend** and **Generalization** features should not be used in the use cases.

Boundary Box and the Environment

The boundary box delineates the system under design from the external actors. It shows what is within the system environment. Use cases are inside the system (boundary box); actors are outside the system. To draw the boundary box in a use case diagram, follow these steps:

1. Click the **Create Boundary box** icon  on the **Drawing** toolbar.
2. Click in the upper, left corner of the drawing area and drag to the lower right. Rhapsody creates a boundary box.
3. Rename the boundary box to be represent your project.


Actors and Systems Design in Use Cases

The Rhapsody actors represent the following in systems design models:

- ◆ Entities that are outside the system
- ◆ External interfaces
- ◆ Parts

- ◆ Flows through service ports

To draw the actors, follow these steps:

1. Click the **Create Actor** icon  on the **Drawing** toolbar.
2. Click the location where you want to position the actor symbol in the use case diagram. Rhapsody creates an actor with a default name of `actor_n`, where n is greater than or equal to 0.
3. Type an appropriate name for the actor to represent the function it serves. Rhapsody adds the actor to the browser.

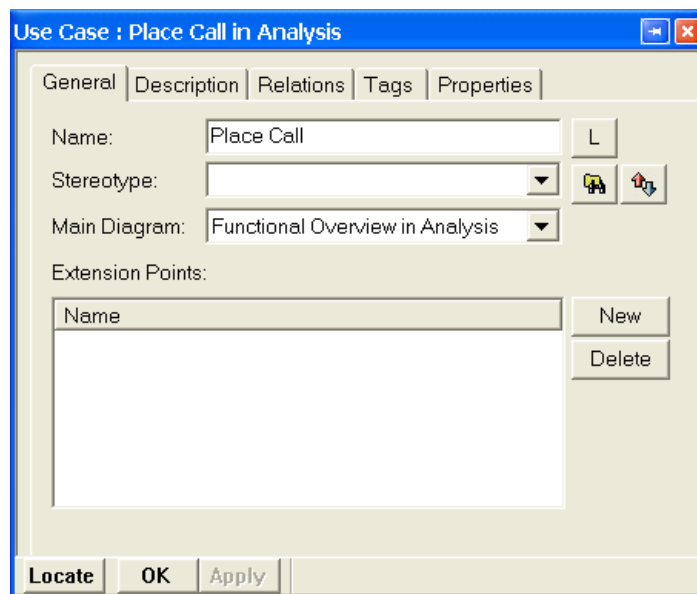
Note

Because code can be generated using the specified names, do not include spaces in the names of actors.

Use Case Features for Systems Engineering

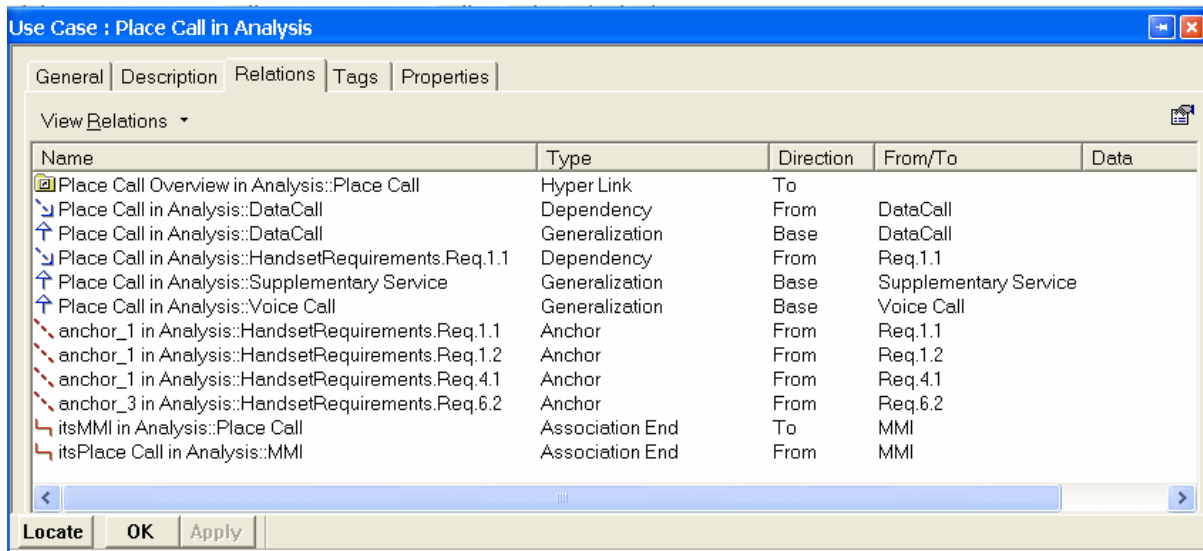
You can define the features of each use case and associate the use case with a different main diagram using the Features dialog box. To define use case features, follow these steps:

1. In the browser, expand a package and the *Use Cases* category. Double-click the use case, or right-click and select **Features** from the pop-up menu. The Features dialog box opens.

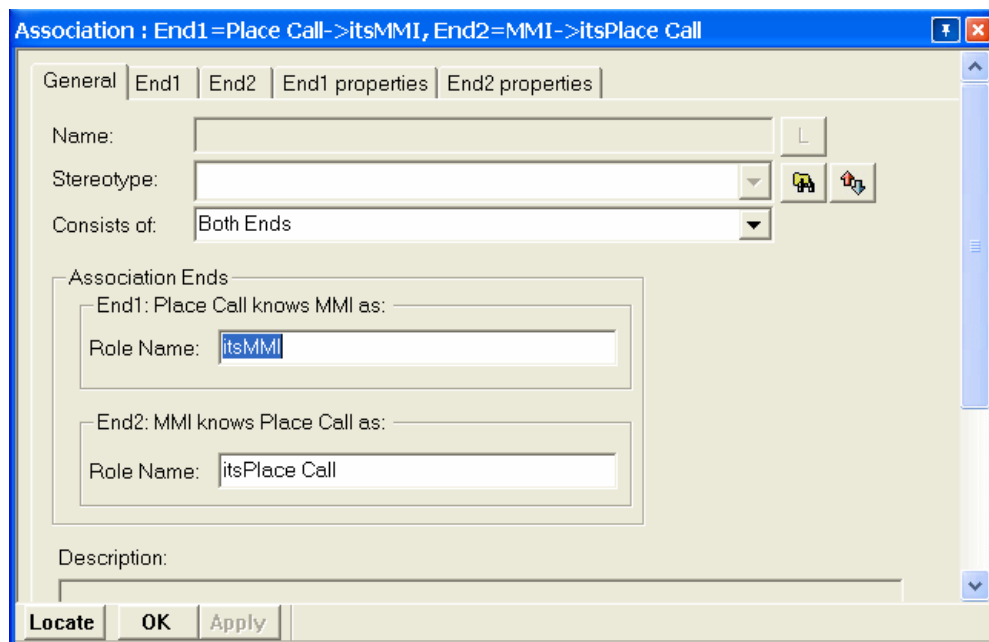


2. Select the **Description** tab, and type the text to describe the purpose of the use case using the internal editor.

3. Select the **Relations** tab to examine the dependencies, flows, generalizations, and associations, shown below, and described in the following sections.




4. Double-click any item in the View Relations list to examine the details and make any required changes, as shown in the AssociationEnd example below.



5. Click **OK** to apply the changes and close dialog box.

Associating Actors with Use Cases

Actors initiate actions or receive information from the system. To create these necessary connections for interaction, draw association lines using these steps:

1. Click the **Create Association** icon  on the **Drawing** toolbar.
2. Click the edge of the actor, then click the edge of a use case. Rhapsody creates an association line with the name label highlighted. You do not need to name this association, so you may press **Enter**, or you may type label text in the highlighted area.
3. In the browser, expand the **Actors** category to view any relationships you have created between actors and use cases.

Defining Requirements in Use Case Diagrams

Systems engineers often employ use case diagrams to define requirements. This technique provides the following advantages:

- ◆ Naming system capabilities to add specificity to design work
- ◆ Showing important user interactions with the system to consider in the design
- ◆ Returning a result visible to one or more actors
- ◆ Organizing requirements by the use cases to recognize possible design flaws early in the design process
- ◆ Assisting project planning by revealing important relationships in the use cases


Tracing Requirements in Use Case Diagrams

You can add requirement elements to use case diagrams to show how the requirements trace to the use cases. To add the requirements to the use case diagram, follow these steps:


1. Select a requirement from the browser and drag it into or beside a use case.
2. To be certain that the requirement is visible in the diagram, right-click the requirement you placed in the diagram
3. Select **Display Options** from the pop-up menu. The Requirement Display Options dialog box opens.
4. The **Show** group box specifies the information to display for the requirement. Select the **Name** radio button to display the name of the requirement.
5. Click **OK**.

Dependencies between Requirements and Use Cases

You may also use dependencies to link the requirements with the use cases as follows:

1. Click the **Dependency** icon  on the **Drawing** toolbar.
2. Click on the element in the use case diagram and draw the dependency line to the associated requirement. (The dependency arrow head rests on the requirement.)
3. In the browser, expand the `Requirements` category to check that the dependency relationship is listed there.

Defining Flow in a Use Case Diagram

To specify the exchange of information between system elements, use the **Flow**  icon to indicate the flow of data and commands within a system without specifying the details of this communication. As the engineer works on the system specification, these abstractions can be tied to the concrete implementations.

Defining the Stereotype of a Dependency

You can specify the ways in which requirements relate to other requirements and model elements using stereotypes. A *stereotype* is a modeling element that extends the semantics of the UML metamodel by typing UML entities.

Rhapsody includes predefined stereotypes, and you can also define your own stereotypes. Stereotypes are enclosed in guillemets on diagrams, for example, «derive». To define the stereotype of a dependency, follow these steps:

1. Double-click the dependency between a requirement and a use case, or right-click and select **Features** from the pop-up menu.
2. Select `trace` from the **Stereotype** pull-down list.
3. Click **OK** to apply the changes and close the Features dialog box.

Activity Modeling in SysML

Activity modeling in SysML emphasizes the inputs and outputs, sequence, and conditions for coordinating other behaviors, instead of who owns those behaviors. Therefore, the SysML activities specify the following:

- ◆ Coordination of executions of lower level behaviors
- ◆ Flow of control
- ◆ Flow of data

Activities are modelled as “Classifiers” or “Types” with the keyword `<<activity>>`.

Action Types in SysML

The following are the five basic action types in SysML:


- ◆ **Atomic action** controls flows into the action, the action is performed, and then control flows out of the action.
- ◆ **Call behavior action** invokes other activities.
- ◆ **Call operation** invokes an operation call on a target block or part.
- ◆ **Accept event action** handles processing of events during the execution of a behavior.
- ◆ **Send signal action** graphically shows an event sent to interact with another block.

SysML Activity Diagrams

Activity diagrams show the essential interactions between the system and the environment and the *interconnections of behaviors* for which the subsystems or components are responsible. These diagrams also illustrate the *flow of control* from activity to activity with sequences and conditions. Activity diagrams can model an operation or the details of a computation and be animated to verify the functional flow.

Creating an Activity Diagram

To create an activity diagram, follow these steps:

1. Start Rhapsody if it is not already running and open the model if it is not already open.
2. In the browser, expand the package containing your subsystems, locate the desired block, and the `Parts` category.
3. Right-click on the item for which you want to describe its activities and select **Add New > Diagrams > Activity Diagram** from the pop-up menu or click the **Activity Diagram** icon  at the top of the window.

The blank diagram opens in the drawing area. You may want to add a title to the diagram to help quickly identify the diagram.

Setting Activity Diagram Properties

Action states represent function invocations with a single exit transition when the function completes. In this example, you will draw the action states that represent the functional processes, and then add names to the action states.

The default settings are used when you add an Action and type a name in the action state on the diagram. That name becomes the action text, not the name of the action. Before adding actions, set the *properties* for the diagram, following these steps:
















1. Right-click outside the Swimlanes frame and select **Diagram Properties**.
2. Select the **Properties** tab and click the **All** radio button for the Filter.
3. Open the **Action** category and change the **showName** and **ShowAction** properties to use these values:









```
Activity_diagram::Action::showName = Name
Activity_diagram :: Action :: ShowAction = Description
```

This second property allows informal text to be displayed on the diagram, while the actual action is described formally using an *executable language*. Click **OK** to save the changes and close the dialog box.

Activity Diagram Drawing Icons for Systems Engineering


A systems engineering activity diagram has the following drawing tools

Drawing Icons	Definitions
	Action icon creates a single, top-level action state. See Drawing Action States for more information.
	Action Block icon draws compound actions that can be decomposed into actions. Action blocks can show more detail than might be possible in a single, top-level action.
	Subactivity icon adds a new subchart to an existing action. This subchart defines a secondary action to the main action. This helps to simplify the diagram. See Drawing a Subactivity for more information.
	Object node icon identifies where an object is passed from the output of one state's actions to the input of another state's actions.
	Call Behavior icon creates a call to a behavior in another activity diagram or to the entire activity diagram. You may add calls to both activity diagrams and subactivity diagrams.
	Activity Flow icon creates a transition between action states.
	Default Flow icon identifies the starting point for the actions in the activity diagram. See Drawing a Default Flow for more information.
	Loop Activity Flow icon creates a looping behavior in a program. Loop activity flows are often used on action blocks to indicate that the block should loop until some exit condition becomes true.
	Condition Connector icon combines different flows into a common target.
	Termination State icon identifies either local or global termination, depending on where they are placed in the activity diagram.
	Junction Connector icon combines different flows to a common target and is often a decision point.
	Diagram Connector icon links one part of an activity diagram to another part on the same diagram. They can also be used to show looping behavior.
	Draw Join Sync Bar icon allows the merging of two or more concurrent flows into a single outgoing flow.
	Draw Fork Sync Bar icon allows the splitting of one in-going flow into two or more outgoing concurrent flows.
	Transition Label icon defines a transition as containing one or more of three possible parts: a timeout trigger, a guard, or an action.

Drawing Icons	Definitions
	Swimlanes Frame icon organizes activity diagrams into sections of responsibility for actions and subactions.
	Swimlanes Divider icon divides the swimlane frame using vertical, solid lines to separate each swimlane (actions and subactions) from adjacent swimlanes.
	Dependency icon indicates a dependent relationship between two items in the diagram.
	Send Action State icon is represents sending actions to external entities. The Send Action State is a language-independent element, which is translated into the relevant implementation language during code generation.
	Call Operation icon represents a call to an operation of a classifier.
	Action Pin icon adds an element to represent the inputs and outputs for the relevant action or action block. An action pin can be used on a Call Operation (derived from the arguments). This icon is displayed by default in a SysML profile project. See Adding Action Pins / Activity Parameters to Diagrams for more information.
	Activity Parameter icon defines a characteristic of an action block. This icon is displayed by default in a SysML profile project.
	Accept Event Action icon lets you add this element to a systems engineering activity diagram so that you can connect it to an action to show the resulting action for an event. This element can specify the following: <ul style="list-style-type: none"> • Event to send • Event target • Values for event arguments This icon is displayed by default in a new SysML profile project.


Drawing Action States

To draw action states in the diagram, follow these steps:

1. Click the **Action** icon  on the **Drawing** toolbar and create action states in the diagram.
2. Name each new action carefully to describe its function.
3. Click the action state, or right-click and select **Features** from the pop-up menu.
4. In the **Description** box, type the desired information.
5. Click **OK** to apply the changes and close the **Features** dialog box.


Drawing a Default Flow

One of the Action States must be the *default* flow. This is the initial state of the Activity. To identify the default flow state, follow these steps:

1. Click the **Default Flow** icon  on the **Drawing** toolbar.
2. Click near the default action state and then click its edge. Press **Ctrl+Enter** stop drawing the connector and not label it.

Drawing a Subactivity

A *subactivity* represents the execution of a non-atomic sequence of steps nested within another activity.

1. Click the **Subactivity** icon  on the **Drawing** toolbar.
2. In the swimlane, click or click-and-drag to draw the subactivity state.
3. Name the subactivity state.
4. To display the subactivity icon in the lower right corner of the state drawing, right-click the subactivity box and select **Display Options** from the menu.
5. Click the **Icon** radio button for the **Show Stereotype** selections and click **OK**.

Note

Limitation: Subactivities do not support swimlanes.

Drawing Transitions

Transitions represent the response to a message in a given state. They show what the next state will be. In this example, you will draw the following transitions:


- ◆ Transitions between states
- ◆ Fork and join transitions
- ◆ Timeout transition

Note

To change the line shape of a transition, right-click the line, select **Line Shape** from the pop-up menu, and then **Straight**, **Spline**, **Rectilinear**, or **Reroute**.

Drawing Transitions Between States

To draw transitions between states, follow these steps:

1. Click the **Activity Flow** icon  on the **Drawing** toolbar.
2. Click the subactivity state, and then click the state.
3. Name the transition and then press **Ctrl+Enter**.



Rhapsody allows you to assign a descriptive label to an element. A labeled element does not have any meaning in terms of an executable action, but the label helps you to reference and locate elements in diagrams and dialog boxes. A label can have any value and does not need to be unique.

Note

When drawing activity flows, it is a good practice to not cross the flow lines. This makes the diagram easier to read.

Drawing Swimlanes

Swimlanes organize activity diagrams into sections of responsibility for actions and subactions. Vertical, solid lines separate each swimlane from adjacent swimlanes. To draw swimlanes, create a swimlane frame and then a swimlane divide using these steps:

1. Click the **Swimlanes Frame** icon  on the **Drawing** toolbar.
2. Click to place one corner, then drag diagonally to draw the swimlane frame.
3. Click the **Swimlanes Divider** icon  on the **Drawing** toolbar.
4. Click the middle of the swimlane frame. Rhapsody creates two swimlanes, named `swimlane_n` and `swimlane_n+1`, where n is an incremental integer starting at 0. Rename the swimlanes as desired.

If you drag the swimlane left or right, it also resizes the swimlane frame.

Drawing a Fork Synchronization

A *fork synchronization* represents the splitting of a single flow into two or more outgoing flows. It is shown as a bar with one incoming transition and two or more outgoing transitions.

To draw a fork synchronization bar, follow these steps:


1. Click the **Draw Fork Synch Bar** icon  on the **Drawing** toolbar.

2. Click or click-and-drag between two action states. Rhapsody adds the fork synchronization bar.
3. Click the **Activity Flow** icon, and draw a single incoming transition from one state to the synchronization bar. Type the name and then press **Ctrl+Enter**. This transition indicates that a call request has been initiated.

Drawing a Join Synchronization

A *join synchronization* represents the merging of two or more concurrent flows into a single outgoing flow. It is shown as a bar with two or more incoming transitions and one outgoing transition.

To draw a join synchronization bar, follow these steps:

1. Click the **Draw Join Synch Bar** icon  on the **Drawing** toolbar.
2. Click or click-and-drag between an action state and a subactivity. Rhapsody adds the join synchronization bar.
3. Click the **Activity Flow** icon and draw the incoming transitions to the synchronization bar.
4. Draw one outgoing transition from the synchronization bar to subactivity. Type name, and then press **Ctrl+Enter**.

Creating a Sequence Diagram from an Activity Diagram

To generate a new sequence diagram from an existing activity diagram, follow these steps:

1. Select the activity diagram in the browser.
2. Right-click and select **Create New Scenario from Activity Diagram** from the menu.

For more information about using this feature, see [Harmony Process and Toolkit](#).

Creating a Design Structure

Internal Block diagrams and Block Definition diagrams define the system structure and identify the large-scale organizational pieces of the system. They can show the flow of information between system components, and the interface definition through ports. In large systems, the components are often decomposed into functions or subsystems.

Block diagrams define the components of a system and the flow of information between components. *Structure diagrams* can have the following parts:

- ◆ *Block* contains parts and may also include links inside a block.
- ◆ *Actors* are the external interfaces to the system.
- ◆ *Service Port* is a distinct interaction point between a class, part, or block and its environment.
- ◆ *Dependency* shows dependency relationships, such as when changes to the definition of one element affect another element
- ◆ *Flow* specifies the exchange of information between system elements at a high level of abstraction.

Creating a Block Definition Diagram
















A *Block Definition diagram* (External block diagram) shows the system structure and identifies the system components (blocks) and describes the flow of data between the components from a black-box perspective. To create a [Block Definition Diagram](#), follow these steps:










1. In the browser, right-click the `Architecture` package, then select **Add New > Diagrams > Block Definition Diagram** from the pop-up menu. The New Diagram dialog box opens.
2. Type a name for your architecture diagram.
3. Click **OK** to close the dialog box.

Rhapsody automatically creates the `Block Definition Diagrams` category in the browser and adds the name of the new block definition diagram. In addition, Rhapsody opens the new diagram in the drawing area. Now you can add blocks and show the relationships of the blocks by linking them from standard or flow ports using connection features as listed in [Block Definition Diagram Drawing Tools](#). You may also add [Block Definition Diagram Graphics](#).

Block Definition Diagram Drawing Tools

The **Drawing** toolbar for a block definition diagram includes the following tools available:

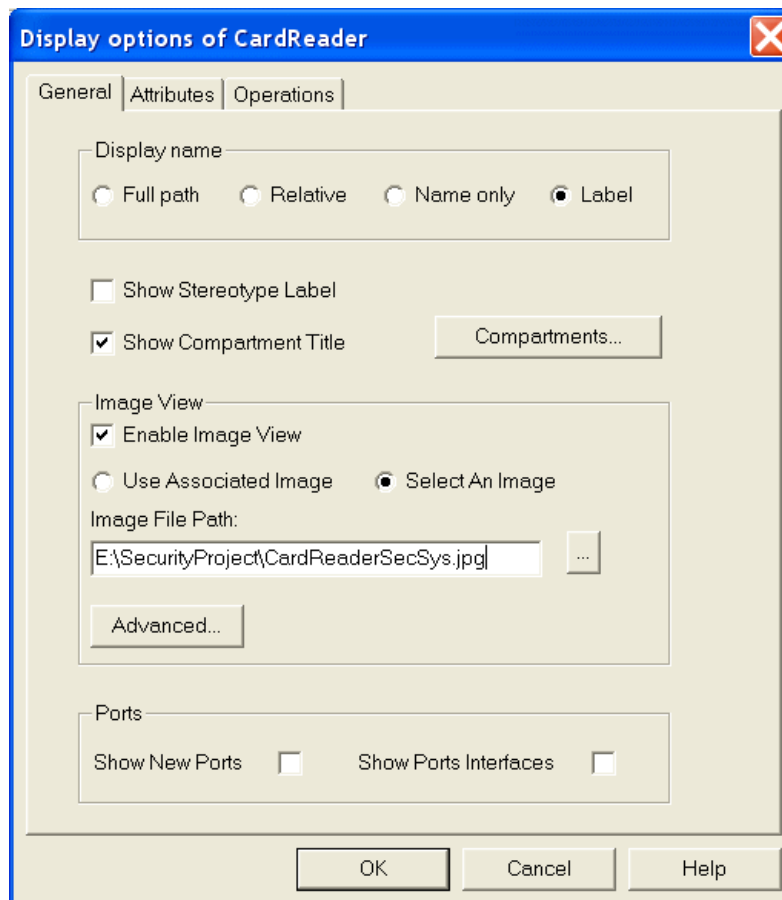
Drawing Icons	Definitions
	Block icon draws an instance of a class that can belong to packages and parts.
	Part icon draws a major component of the model.
	Interface icon creates a connection between components as a set of operations performed by a hardware or software element in the system. A component realizes an interface if it supports the interface; another component then uses that interface.
	FlowSpecification icon creates a non-atomic flow port typed by an interface.
	Create Package icon draws a group of parts or blocks to form a single element of the model.
	FlowPort icon shows how the data flows between blocks.
	StandardPort icon draws the connection points among blocks or parts and their environments.
	Association icon creates connections that are necessary for interaction.
	Directed association icon indicates the only object that can send messages to another object.
	Aggregation icon shows an association specifying a whole-part relationship between the aggregate (whole) and a component part.
	Directed Composition icon shows the instance of a class that cannot be contained by other instances.
	Connector icon shows the relationship among blocks or parts.
	Dependency icon indicates a dependent relationship between two items in the diagram.
	Inheritance icon indicates the relationship between the derived class and its parent. The derived class has the same data members and behaviors as the parent class.
	Flow icon indicates the flow of data and commands within a system.
	ConstraintBlock icon defines restrictive properties controlling the relationships of blocks.

Drawing Icons	Definitions
	Constraint icon defines a semantic condition or restriction.
	BindingConnector icon specifies the properties at the ends of the connector (link).
	Satisfaction icon explains how problem will be overcome. It usually includes the key element or design feature that solves the problem.
	Allocation icon allows a systems engineer to denote the mapping of elements within the structure of a model.
	Problem icon records an unresolved issue, limitation, or failure related to one of the model elements.
	Rationale icon permits you to record the reason for decisions, requirements, and other design issues. A rationale may reference a more detailed document or report.
	ValueType icon creates an extension of the UML dataType. It is used where one would use a dataType (as a type for a value property for a block, for example). A UML dataType usually expresses a quantity in a software implementation type, such as a float or double. However, in SysML, valueType expresses a quantity in a standard unit, such as milliAmpere. The valueType also includes a placeholder for the quantity plus the unit. It should be noted that SysML allows a valueType to be defined without a unit. For example, when the valueType expresses a ratio, the valueType expresses a quantity only.
	Dimension is used in SysML to separate the concept of "Unit" (see definition below) from "Dimension." A SysML Dimension represents a standard physical concept for a set of Units. For example "Length" is a Dimension in SysML. The Units meter, inch, kilometer, mile, light_year refer to the concept of Length; therefore, they all have Length as the value of their Dimension tag. Dimension helps to organize Units into comprehensive sets based on the physical domain. Like Units, a Dimension should be used only to set the Dimension tag for a Unit or for a valueType and is never used as a type. Dimensions are usually selected from a standard library model (part of the Rhapsody's SysML profile).
	Unit is used in conjunction with a SysML valueType to express a given quantity in a standard way so that other quantities having the same Unit can be compared. The Unit is usually taken from a standard library of Units, such as SI or NIST (part of Rhapsody's SysML profile). Units may be expressed in terms of other units or "derived units." Units are only used to set the Unit tag of a valueType. They should not be used as types for value properties since the Unit concept in SysML does not include the concept of quantity. Instead, a valueType should be used.

Block Definition Diagram Graphics

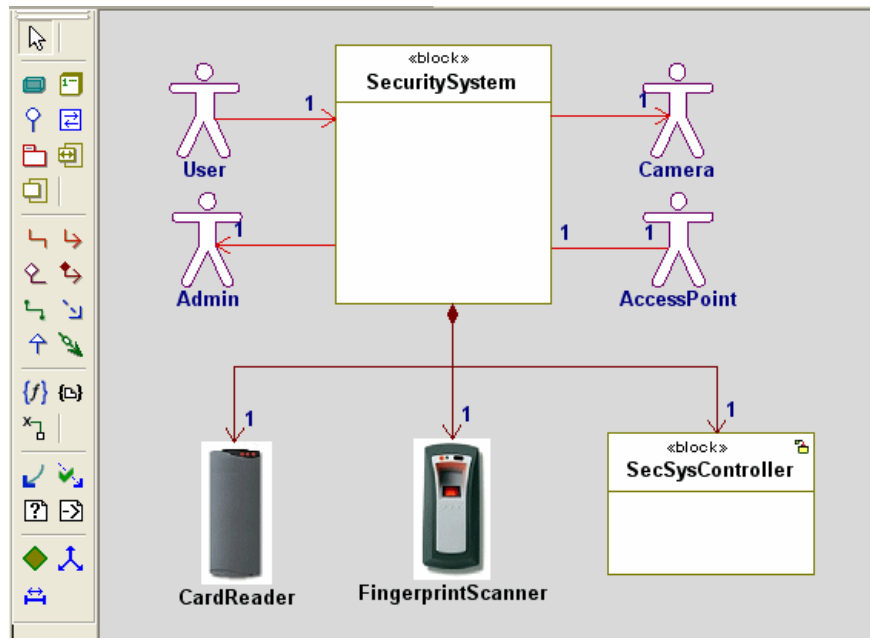
To add the graphics to the block definition diagram, follow these steps:

1. Highlight the item in the diagram for which you want to add a image, such as a photograph or drawing.
2. Right-click and select **Display Options** from the menu.
3. In the **General** tab of the dialog box, select the **Enable Image View** check box. This automatically enables the Select An Image option.
4. Browse to find the **Image File Path** and click **OK** to insert the selected image, as shown in this example.



The following is an example of a completed block definition diagram with two inserted images, the CardReader and FingerprintScanner. Using images to represent system components helps the project team members to identify individual parts quickly.

Block Definition Diagram



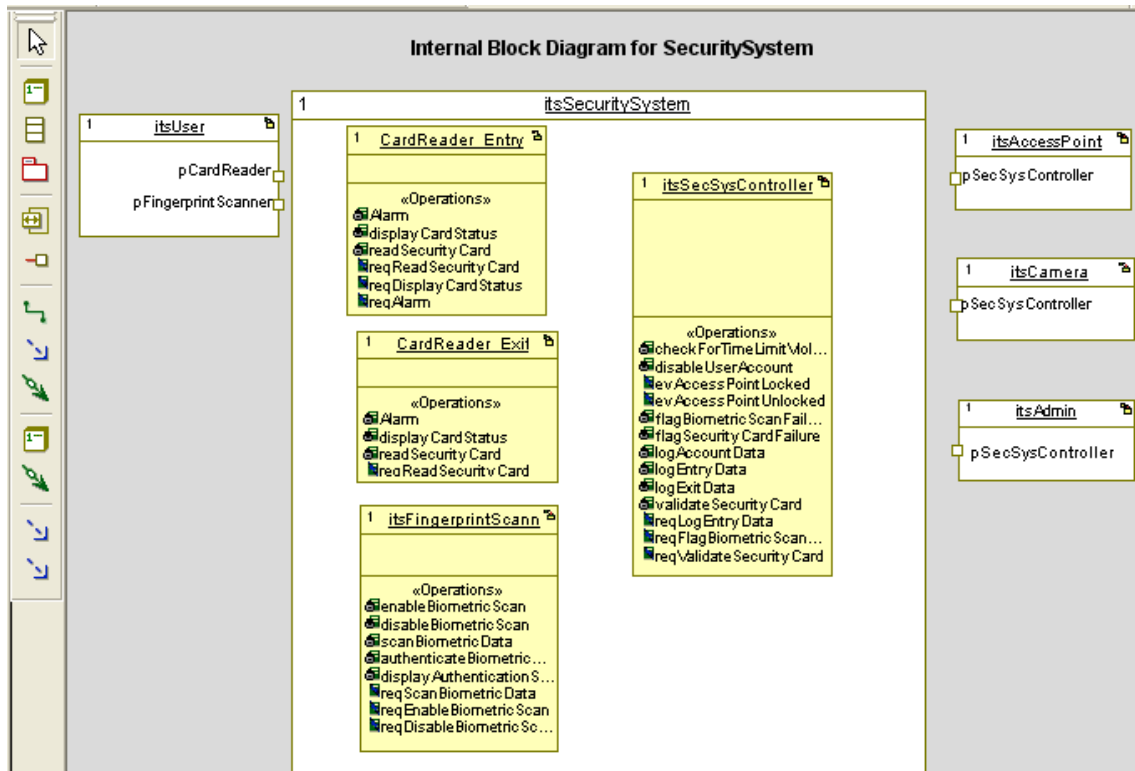
Creating an Internal Block Diagram

An *internal block diagram* shows the internal structure or decomposition of a block into its parts or subsystems. To create an internal block diagram, follow these steps:

1. In the browser, expand a package.
2. Right-click block in the package and select **Add New > Diagrams > Internal Block Diagram**.
3. Type the name diagram and click **OK**. Rhapsody creates the diagram in drawing area.

The following is an example of a completed internal block diagram for a security system.

Internal Block Diagram Example















If the ports are not visible, follow these steps:

1. Right-click the block.
2. From the context menu, select **Ports > Show All Ports**.


Internal Block Diagram Drawing Icons

An internal block diagram has the following drawing icons available:

Drawing Icons	Definitions
	Part icon draws a major component of the model.
	Block icon draws an instance of a class that can belong to packages and parts.
	Create Package icon draws a group of parts or blocks to form a single element of the model.
	FlowPort icon shows how the data is bound to each constraint.
	StandardPort icon draws the connection points among blocks or parts and their environments.
	Connector icon shows the relationship among blocks or parts.
	Dependency icon shows the relationships among the packages in the diagram.
	Flow icon shows how the data is bound to each constraint.
	ConstraintProperty icon defines a characteristic of a semantic condition or restriction.
	BindingConnector icon specifies the properties at the ends of the connector.
	Satisfaction icon explains how problem will be overcome. It usually includes the key element or design feature that solves the problem.
	Allocation icon allows a systems engineer to denote the mapping of elements within the structure of a model.


Drawing the Parts

The internal block diagram uses parts to represent the activities. To draw the parts, follow these steps:


1. Click the **Part** icon  on the **Drawing** toolbar.
2. In the upper, left corner of the block click or click-and-drag.
3. Type the name you want, and then press **Enter**.

Drawing Standard Ports and Links

You need to link the parts to show the interactions of parts. To accomplish this, you need to draw service ports as follows:

1. Click the **Standard Port**  icon on the **Drawing** toolbar.
2. Click on the edge of the block, name the service port.
3. Click on the edge of another block, name the service port.

To draw link the two parts through their service ports, follow these steps:

1. Click the **Connector**  icon on the **Drawing** toolbar.
2. Click the service port of the sending part, and then click the service port of the receiving part.

Specifying the Port Contract and Attributes

Now you may specify the port contract and attributes for a port *interface* as follows:

1. Double-click the port to display the Features dialog box.
2. In the **General** tab, click the **Behavior** and/or **Reversed** radio buttons to set the desired **Attributes**. Click **Apply** to save the changes and keep the dialog box open.
3. Select the **Contract** tab.
4. Select the **Provided** folder icon and click the **Add** button. The Add new interface dialog box opens.
5. Select **In** or another option from the pull-down list, then click **Apply** to save the changes and leave the dialog box open.
6. Select the **Required** folder icon and click the **Add** button. Select **Out** or another option from the pull-down list.
7. Click **OK** to apply the changes and close the dialog box. Rhapsody automatically adds the provided and required interfaces.
8. Click **OK** to apply the changes and close the dialog box.

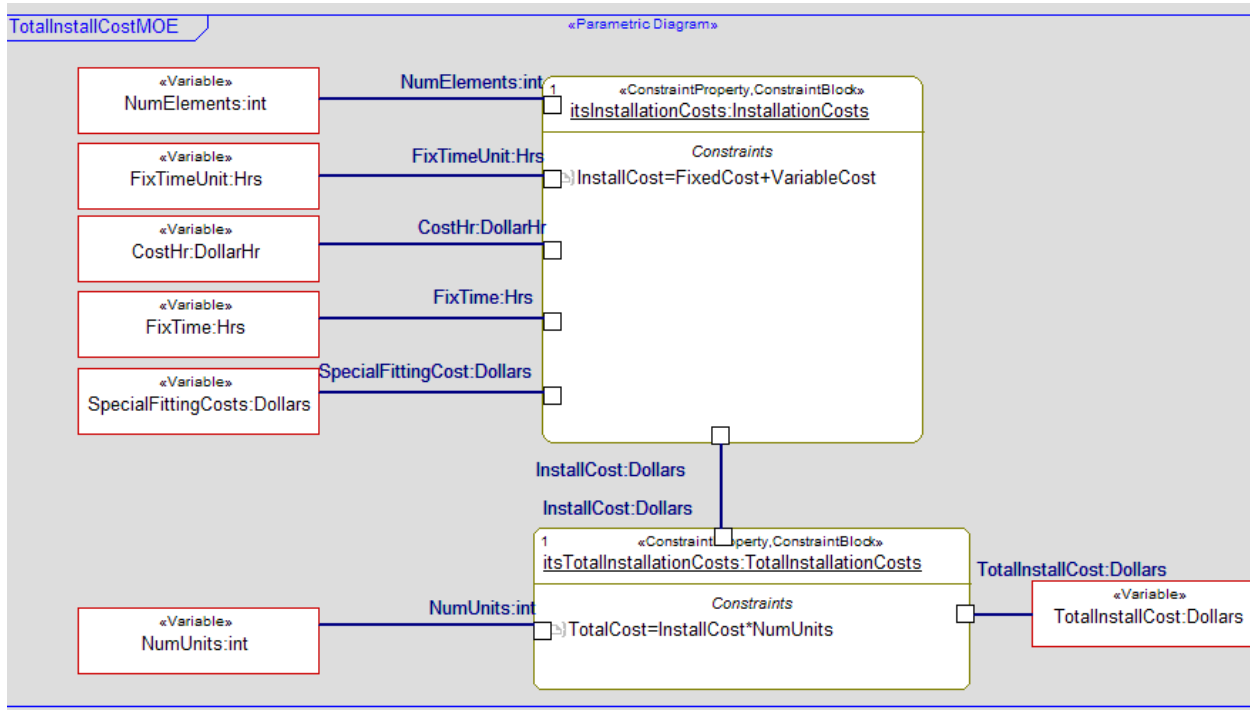
Parametric Diagrams

Parametric diagrams show mathematical relationships (such as performance constraints) among the pieces of the system being designed. These diagrams are only available if you are using the SysML profile for your project. Parametric diagrams have the following general uses:

- ◆ Indicate the relationships for the objective analysis of functions
- ◆ Measure effectiveness
- ◆ Clarify the relationship between one variable and another

Parametric Diagrams cannot exist in isolation. They are created using model elements called *constraint blocks* that define generic or basic mathematical formulas. See the example of a completed parametric diagram in [Security System Total Installation Costs for X number of units](#).











Security System Total Installation Costs for X number of units



To illustrate another possible use for a parametric diagram, a set of constraint blocks could define the volume of a tube, a disc, and the formula for an objects mass (Volume*Density). The parametric diagram would show how these constraint blocks combine, in a particular usage as a set of constraint properties, to give the mass of, perhaps, a tin can or a hollow cylindrical container based upon a set of input parameters. Constraint blocks are created within a block definition diagram, as described in [Creating a Block Definition Diagram](#).


Parametric Diagram Drawing Icons

A parametric diagram has the following drawing icons available:

Drawing Icons	Definitions
	ConstraintBlock icon defines restrictive properties controlling the relationships of blocks.
	ConstraintProperty icon defines the usages of a Parametric Constraint Block so that the blocks can be reused with changes to the usage of the property, but without any changes to the underlying equations.
	Create Package icon draws a group of parts or blocks to form a single element of the model.
	Block icon draws an instance of a class that can belong to packages and parts.
	Part icon draws a major component of the model.
	ConstraintParameter icon defines a characteristic of a semantic condition or restriction.
	Binding Connector icon binds the data to the constraint.
	Satisfaction icon explains how problem will be overcome. It usually includes the key element or design feature that solves the problem.
	Allocation icon allows a systems engineer to denote the mapping of elements within the structure of a model.
	Dependency icon indicates a dependent relationship between two items in the diagram.

Creating the Constraint Block

Parametric diagrams are based upon *constraint properties*. Constraint properties can only be created from *constraint blocks*. To create a constraint block in a block definition diagram, follow these steps:

1. Right-click the package in the browser where you want the diagram to be created.
2. Select **Add New > Diagrams > Block Definition Diagram** from the pop-up menu.
3. Click the **Constraint Block** icon  above the window and place the Constraint Block on the block definition diagram.
4. Rename the new block using the Features dialog box.
5. Since constraints can only be added to an element in the browser, right-click the constraint block and select **Locate**. This navigates to that block in the browser.
6. Right-click the block and select **Add New > General Elements > Constraint**. This specifies the relationship between the constraint parameter and the block.
7. Open the **Constraint Features** dialog box and rename the constraint. Click **Apply**.
8. In the **Specification** of the constraint, add the appropriate mathematical relationship, i.e. $\text{Volume} = B * D * H$. Click **Apply** and the constraint features appear in the constraint block.
9. Add attributes to the constraint block if there are any constants that the constraint formula may use, for example g which is 9.81 M/s^2 . Click **OK** to close the dialog box and save the Features you entered.
10. Add constraint parameters for the variables in the constraint formula. This is also accomplished from the browser. Right-click the constraint block and select **Add New > Constraint Blocks > ConstraintParameter**. Rename the parameter in the Features dialog box.
11. The constraint parameter may be typed with an SI unit by opening its Features dialog box and then selecting **Type**. From the pull-down menu, scroll to the top and select **<<Select>>**, navigate through the package tree to the SysML profile, and locate the ModelLibrary unit definitions. Select the correct unit definition.
12. The constraint parameter with its type then displays on the constraint block. New constraint parameters can be added to the constraint block directly from the constraint parameters section of the browser hierarchy. Repeat the constraint parameter definition steps (10 through 12) for each variable element in the constraint.


Note

Typically, constraint parameters are not shown on constraint blocks. To hide the parameters, right-click the constraint block and select **Ports > Hide All Ports**.

Creating the Parametric Diagram



Constraint properties show how a constraint block is used, and the parametric diagram illustrates this usage. To define a parametric diagram, first create a constraint property and then “type” it with the constraint block.

To create a parametric diagram for a constraint block, follow these steps:

1. From the appropriate package in the browser, right-click the package and select **Add New > Diagrams > Parametric**.
2. From the **Drawing** toolbar, select **ConstraintProperty** .
3. Drag and drop the Constraint Property onto the diagram.
4. Open the Features dialog box and set its Type to the correct Constraint Property.
5. Rename the Property to its usage.
6. Repeat for other relevant blocks for the calculation. Each block has a set of Constraint Parameters which show relationships between the blocks when they are joined together with Binding Connectors.
7. Next you need to add new pieces of data needed in the parametric diagram to bind the parametrics.
8. Double-click a part and select **Features** and then the **Attributes** tab.
9. Click the <<New>> item and add mathematical attributes with the correct name and appropriate type.
10. Click **OK** to save the information and close the dialog box.
11. Drag the data attributes from the browser onto the parametric diagram and connect it to the appropriate constraint parameter with a value binding.

Binding Constraint Properties Together

To show how the data is bound to each constraint, you need to add constraint parameters and binding connections to the parametric diagram with these steps:

1. Click the **ConstraintParameter** icon  on the **Drawing** toolbar and draw this connection point on a constraint.
2. Click the **BindingConnector** icon  and draw the connection between the data source and the constraint to bind a value to its constraint.

Adding Equations

To add the required equations to the constraints, follow these steps:

1. Right-click on a constraint to display the Features dialog box.
2. Type the equation in the **Description** area and click **OK** to save the equation and close the dialog box.
3. To display the equation in the constraint, right-click it and select **Display Options** from the menu. Click the **Compartment** button.
4. Select **Description** from the Available list and click the **Display** button to move it to the Displayed column. Click **OK** to save this change. You may wish to perform this action on each of the boxes containing an equation.

Implementation Using the Action Language

In order to show actions in a model, the designer needs an implementation language. Rhapsody includes an Action Language, a subset of C++ that uses a C++ compiler to allow you to simulate the model. This language provides the following:

- ◆ Message passing
- ◆ Data checking
- ◆ Actions on transitions
- ◆ General model execution

The basics of the Action Language are presented below. A more complete description of the syntax can be found in the [Action Language Reference](#).

Basic Syntax Rules

This streamlined version of C++ has these basic syntax rules:

- ◆ It is case-sensitive, so “evGo” is different from “evgo.”
- ◆ Names must follow these rules:
 - No spaces (“Start motor” is not correct.)
 - No special characters other than underscore (“_”) (“StartMotor@3” is not correct.)
 - Must start with a letter, can’t start with an underscore (“2ToBegin” is not correct.)
- ◆ All statements must end in a semicolon
- ◆ Do not to use [Reserved Words](#) such as *id*, *for*, *next*.

Frequently Used Statements

To add some simple operations to your model, you may use the following:

- ◆ These increment/decrement operators provide standard functions:
 - `X++`; (Increment X)
 - `X--`; (Decrement X)
 - `X=X+5`; (Add 5 to X)
- ◆ To print out on the screen, use one of these:
 - `cout << "hello" << endl;`
 - `cout << attribute_name << endl;`
 - `cout << "hello : " << attribute_name << endl;`

Reserved Words

The Action Language reserved words are listed below. All reserved words for built-in functions are lower case, for example, *if*.

asm	continue	float	int	params	sizeof	typedef
auto	default	for	IS_IN	private	static	union
break	delete	friend	IS_PORT	protected	struct	unsigned
case	do	GEN	long	public	switch	virtual
catch	double	goto	new	register	template	void
char	else	id	operator	return	this	volatile
class	enum	if	OPORT	short	throw	while
const	extern	inline	OUT_PO RT	signed	try	

Assignment and Arithmetic Operations

The following are the the assignment and arithmetic operations available in the action language:

X=1;	Sets X equal to 1
X=Y;	Sets X equal to Y
X=X+5;	Adds 5 to X
X=X-3;	Subtracts 3 from X
X=X*4;	Multiplies X by 4
X=X/2;	Divides X by 2
X=X%5;	Set X to remainder of X divided by 5
X++;	Adds 1 to X
X--;	Subtracts 1 from X

Defining an Action using the Action Language

To define action states, Rhapsody provides an action language that is a subset of C++. To define an action, follow these steps:

1. Double-click an action state, or right-click and select **Features** from the pop-up menu.
2. Type action language into the **Action** box, as shown in this example of an action language instruction:

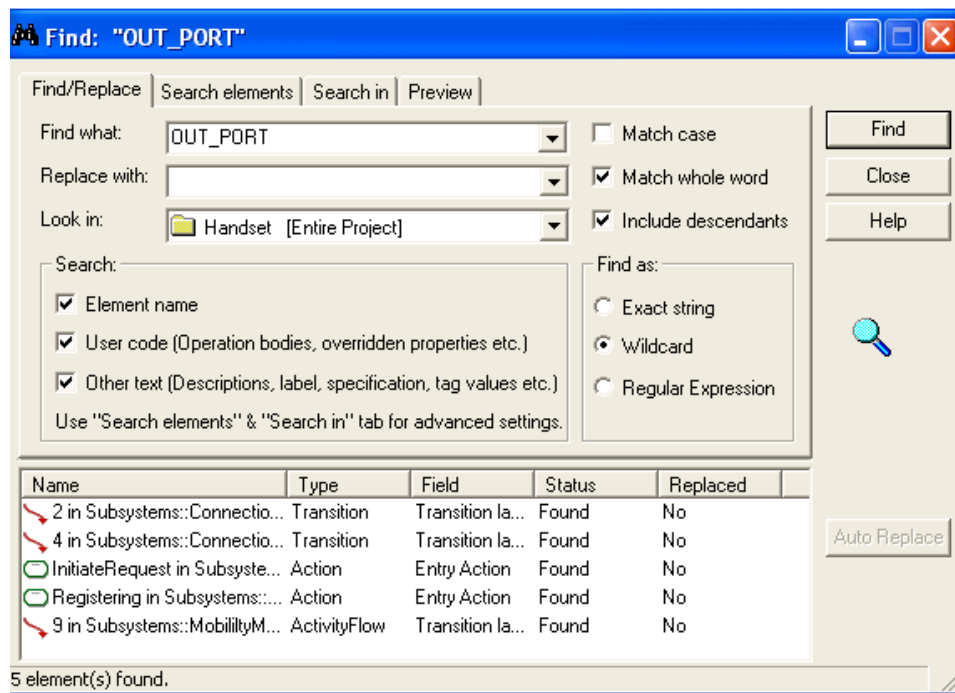
```
OUT_PORT(mm_cc)->GEN(RegistrationReq);
```

3. Click **OK** to apply the changes and close the dialog box.

Checking Action Language Entries

After entering action language into several models, it is useful to check those entries using the Rhapsody search facility. Follow these steps to check your action language entries:

1. Select the **Edit > Search in Model** menu options.
2. Type a portion of the instruction that you want to use for the search in the **Find What** box.
3. Click **Find**. Rhapsody lists all of the locations where it found that search item.
4. Click on the entries in the list of elements found, and the system displays the diagram containing that entry and the dialog box with the full action language content. Make any corrections that are needed.



Action Language Reference

This section lists each of the action language commands with its definition and syntax.

Printing

Use the following action language commands to control print operations within an application.

Using printf

```
printf(format, arg1,...,arg_n)
```

Prints arguments utilizing format specified.

Format is %type, where type is: c character s string d decimals f float

Examples:

Syntax	Output
<pre>printf ("Characters: %c %c \n", 'a', 65);</pre>	Characters: a A
<pre>printf ("Decimals: %d %ld\n", 1977, 650000);</pre>	Decimals: 1977 650000
<pre>printf ("floats: %f \n", 3.1416, 4.67);</pre>	Floats: 3.1416 4.67
<pre>printf ("%s \n", "A string");</pre>	A string

Using cout

```
cout << "Str_1" << Var_1 <<...<< endl;
```

(Prints items listed between cout and endl)

Example:

```
cout << "the value of X is" << X << endl;
```

In this example, if X equals 5, then the output will be: *the value of X is 5*

Comparison Operators

`X==5;`
(Is X equal to 5)

`X!=Y;`
(Is X not equal to Y)

`X<3;`
(Is X less than 3)

`X<=12;`
(Is X<=12)

`X>Z;`
(Is X greater than Z)

`X>=34;`
(Is X greater than or equal to 34)

`X>Y && X<7`
(Is X greater than Y and X less than 7)

`X>Y || X<7`
(Is X greater than Y or X less than 7)

Conditional Statements

`if (comparison expression) statement;else statement;`

Single Statement Example:

```
if (X<=10)    X++; else    X=0;
```

(If X is less than or equal to 10 then add 1 to the value of X, otherwise set X to 0)

Multi Statement Example:

```
If (X<=10) {  
    X++;  
    printf ("%s \n", "X is less than 10");  
} else {  
    X=0;  
    cout << "Finished" << endl;  
}
```

(If X is less than or equal to 10 then add 1 to the value of X, and print the statement "X is less than 10". Otherwise set X=0 and print the statement "Finished.")

Incremental Looping

for (Variable=Start Value; Comparison Statement; increment/decrement Variable)

```
{ Statement; Statement;... }
```

(Execute the statement(s) as long as the variable is true in the comparison statement, then increment or decrement the variable. Variable starts at the defined Start Value.)

Example:

```
For (X=0; X<=10; X++) cout << X << endl;
```

(If X is less than or equal to 10, then print the value of X at that time, then increment X.)

Conditional Looping

While (Conditional Statement)

```
{ Statement; Statement;... }
```

(Execute the statement(s) as long as the conditional statement is true)

Example:

```
X=0;
while (X<=10) {
    cout << X << endl;
    X++;
}
```

(X starts with the value of 0. While X is less than or equal to 10, print the value of X, and then add 1 to the value of X.)

Invoking Block Operations

Operation_Name(parm_1, ...,parm_n);

(Invoke the block operation Operation_Name with/without parameters.)

Examples:

`go ();`

(Invoke operation `go` without parameters)

`min(x,y)`

(Invoke operation `min` with parameters X and Y)

Generating Events

GEN(evName);

(Generate event evName and send to yourself.)

Examples:

`GEN(evStart);`

(Send event `evStart` to your own statechart)

`GEN(evMove(10,X));`

(Send event `evMove` with parameters 10 and X to have statechart respond.)

Generating Port Events

OUT_PORT(pName)->GEN(evName);

(Generate event evName and send it to the port pName)

Examples:

`OUT_PORT(p2)->GEN(evStart);`

(Send event `evStart` to port p2)

`OUT_PORT(p2)->GEN(evMove(10,X));`

(Send event `evMove` with parameters 10 and X to port p2)

Referencing Event Parameters

`params->event_parameter;`

(references value of event_parameter)

Examples:

```
if (params->velocity <= 5)...
```

(Test value of parameter velocity for an event to see if less than or equal to 5.)

Testing Port for an Event

`IS_PORT(port_name);`

(Returns TRUE if event that caused current transition arrived through port port_name)

Examples:

```
if (IS_PORT(port_2))...
```

(Test to see if event that caused current transition arrived through port_2, result is TRUE if yes, FALSE if no.)

Test to See if Currently in a State

`IS_IN(state_name);`

(Returns TRUE if in state state_name)

Examples:

```
if (IS_IN(Accelerating))...
```

(Test to see if currently in state Accelerating. Result is TRUE if yes, FALSE if no.)

System Validation

Rhapsody enables you to visualize the model through simulation. *Simulation* is the execution of behaviors and associated definitions in the model. Rhapsody simulates the behavior of your model by executing its behaviors captured in statecharts, activity diagrams and textual behavior specifications. Structural definitions like blocks, ports, parts and links are used to create a simulation hierarchy of subsystems.

Once you simulate the model, you can open simulated diagrams, which let you observe the model as it is running and perform design-level debugging. You can perform the following tasks:

- ◆ Step through the model
- ◆ Set and clear breakpoints
- ◆ Inject events
- ◆ Create an output trace

It is good practice to test the model incrementally using model execution. You can simulate pieces of the model as it is developed. This allows you to determine whether the model meets the requirements and find defects early in the design process. Then you can test the entire model. In this way, you iteratively build the model, and then with each iteration perform an entire model validation.

Creating a Component

A *component* is a level of organization that names and defines a simulatable component. Each component contains configuration and file specification categories, which are used to build and simulate model.

Each project contains a default component, named `DefaultComponent`. You can use the default component or create a new component. In this example, you can rename the default component `Simulation`, and then use the `Simulate` component to simulate the model.

To use the default component, follow these steps:

1. In the browser, expand the `Components` category.
2. Select `DefaultComponent` and rename it `Simulation`.

Setting the Component Features

Once you have created the component, you must set its features.

To set the component features, follow these steps:

1. In the browser, double-click `Simulation` or right-click and select **Features** from the pop-up menu. The Component dialog box opens.
2. The **Executable** radio button to set the Type.
3. If you used the common design package names, select `Analysis`, `Architecture`, and `Subsystems` as the Selected Elements.

These are the packages for which you create a simulatable component. Do not select the `Requirements` package because you do not simulate it.

4. Click **OK** to apply the changes and close the dialog box.

Creating a Configuration

A component can contain many configurations. A *configuration* includes the description of the classes to include in code generation, and settings for building and Simulating the model.

Each component contains a default configuration, named `DefaultConfig`. In this example, rename the default configuration to `Debug`, and then use the `Debug` configuration to simulate the model.

To use the default configuration, follow these steps:

1. In the browser, expand the `Simulate` component and the Configurations category.
2. Select `DefaultConfig` and rename it `Debug`.

Preparing to Web-enable the Model

The first step in Web-enabling a working Rhapsody model is to set its configuration and elements as Web-manageable and then to simulate, build, and run the model.

Note

You cannot webify a file-based C model.

Creating a Web-Enabled Configuration

In this example, create a new configuration and then set its features as follows:

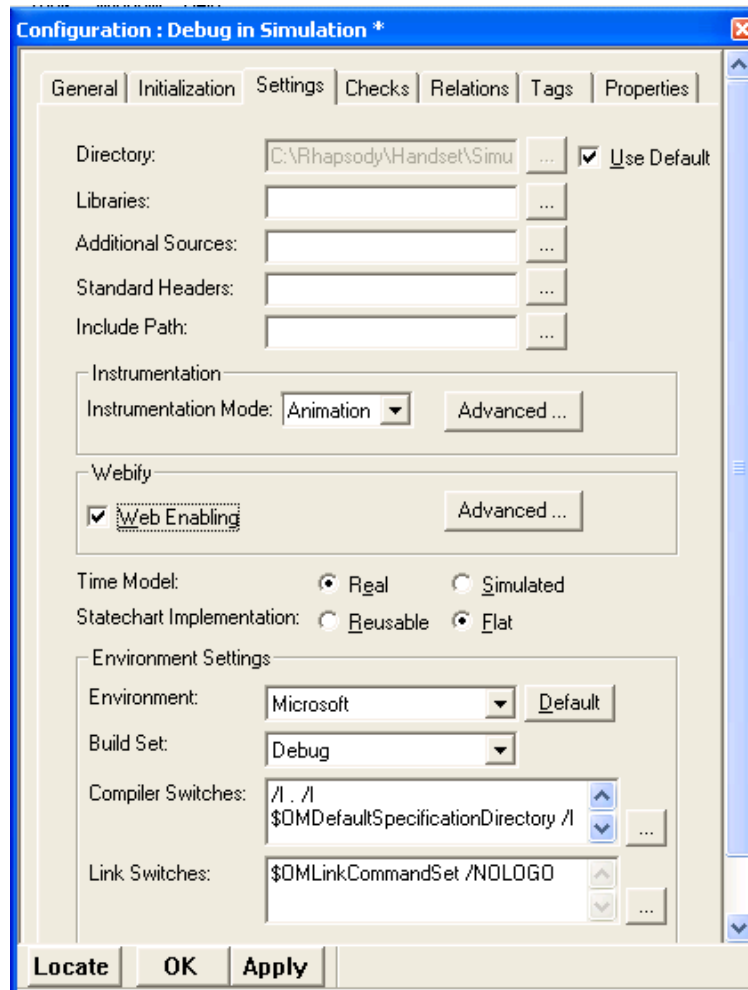
1. Right-click the **Configurations** category and select **Add New Configuration** from the pop-up menu.
2. Type **Panel**.
3. Double-click **Panel** or right-click and select **Features** from the pop-up menu. The Features dialog box opens.
4. Select the **Initialization** tab and set the following values:
 - ◆ For the **Initial instances** box, select **Explicit** to include the classes which have relations to the selected elements.
 - ◆ Select **Generate Code for Actors**.
5. Click **Apply** to save these selections and keep the dialog box open.
6. Select the **Settings** tab, and set the following values:
 - ◆ Select **Animation** from the **Instrumentation Mode** pull-down list. Rhapsody adds instrumentation code to the simulated application, which enables you to simulate the model.
 - ◆ Select **Web Enabling** for **Webify**.
 - ◆ If desired, click the **Advanced** button to change the default values for the Webify parameters. Rhapsody opens the Advanced Webify iconkit Settings dialog box.

This dialog box contains the following boxes, which you can modify:

- Home Page URL—The URL of the home page
- Signature Page URL—The URL of the signature page
- Web Page Refresh Period—The refresh rate in milliseconds
- Web Server Port—The port number of the Web server
- ◆ Select **Real** (for real time) as the **Time model**.

- ◆ Select **Flat** as the **Statechart Implementation**. Rhapsody implements states as simple, enumerated-type variables.

Rhapsody fills in the **Environment Settings** section, based on the compiler settings you configured during installation. At this point the dialog box should resemble this example.



7. Click **OK** to apply the changes and close the dialog box.

Selecting Elements to Web-enable

To Web-enable the model, set the elements that you want to control or manage remotely over the Internet using either the Rhapsody **Web Managed** stereotype or the `WebManaged` property. To select elements to Web-enable, follow these steps:

1. To locate the items you wish to change, select the **Edit > Search in Model** option. Type the name into the **Find What** box and click **Find**. The search shows all instances of that text and the browser path for each.
2. Double-click the item located under the `Subsystems` browser category.
3. In the Features dialog box, select **Web Managed** from the **Stereotype** pull-down list.
4. Click **OK** to apply the changes and close the dialog box.
5. Make the same change to the remaining three events to make them Web Managed.

Note

If the element already has an assigned stereotype, set the element as Web-managed using a property. In the **Properties** tab, select `WebComponents` as the subject, then set the value of the `WebManaged` property within the appropriate metaclasses to `Checked`.

Connecting to the Web-enabled Model

Rhapsody includes a collection of default pages that serve as a client-side user interface for the remote model. When you run a Web-enabled model, the Rhapsody Web server automatically simulates a Web site including the file structure and interactive capability. This site contains a default collection of simulated on-the-fly pages that refreshes each element when it changes.

Note

You can also customize the Web interface by creating your own pages or by referencing the collection of pages that come with Rhapsody.

Navigating to the Model through a Web Browser

You can access a Web-enabled model running on your local machine or on a remote machine. In this example, you will connect to the model on your local machine.

To connect to the Web-enabled model on your local machine, follow these steps:

1. Open Internet Explorer.
2. In the address box, type the following URL:

```
http://localhost
```

Other users on the same network can connect to your local model using the IP address or machine name in place of `localhost`.

If you changed the Web server port using the Advanced Webify iconkit Settings dialog box, type the following:

```
http://<localhost>:<port number>
```

In this URL, `<localhost>` is `localhost` (or the machine name or IP address of the local machine running the MyProject model), `<port number>` is the port specified in the Advanced Webify icon kit Settings dialog box.

By default, the Parts Navigation page of the Rhapsody Web user interface opens.

Note

If you cannot view the right-hand frame in Internet Explorer, go to **icons > Internet Options > Advanced** and clear the **Use Java xx for <applet>** option.

Viewing and Controlling a Model

The Parts Navigation page provides easy navigation to the Web Managed elements in the model by displaying a hierarchical view of model elements, starting from the top level aggregate. By navigating to and selecting an aggregate in the left frame of this page, you can monitor and control your model in the *aggregate table* displayed in the right frame.

Aggregate tables contain name-value pairs of Rhapsody Web-enabled elements that are visible and controllable through Internet access to the machine hosting the Rhapsody model. They can contain text boxes, combo-boxes, and **Activate** buttons. You can monitor the model by reading the values in the dynamically populated text boxes and combo-boxes. You can control the model by pressing the **Activate** button, which initializes an event, or by editing writable text boxes.

Sending Events to Your Model

You can simulate events in the Rhapsody Web user interface and monitor the resulting behavior in the simulated diagrams.

1. If the simulated sequence diagram is not already open, simulate it and click the **Go** button.
2. If the simulated statechart is not already open, simulate it.
3. If the simulated activity diagram is not already open, simulate it.
4. Resize the Rhapsody Web user interface browser window so that you can view the simulated diagrams while sending events to the model.
5. In the navigation frame on the left side of the browser, expand **ConnectionManagement_C[0]**, and click **ConnectionManagement_C::CallControl_C[0]**
6. In the Rhapsody Web user interface, click **Activate** next to the starting point in the sequence diagram.
7. Open the simulated sequence diagram. Rhapsody displays how the instances pass messages.
8. The simulated activity diagram shows transitions from the active state to the inactive state.

You can continue generating events and viewing the resulting behavior in the simulated diagrams.

Importing System Architect's DoDAF Diagrams

System Architect (SA) customers can import SA DoDAF (non-ABM) to use in a Rhapsody project. The SA Importer is a Rhapsody add-on requiring a special license.

The SA Importer requires type mapping between Rhapsody and SA elements using an external mapping file. This allows the users to modify the map to extend the import scope as desired. The out-of-the-box scope of this SA Importer imports information into Rhapsody as SysML elements.

Mapping the Import Scope

The external map file ties SA elements to Rhapsody elements and allows the users to analyze and adjust the imported data. The SA Importer map file is located in Rhapsody installation folder in the `AddOn\ProductIntegrator` directory. You may use any XML editor to modify the map file, `RhpSAMap.xml`. This file has two element types to map SA elements to Rhapsody elements:

- ◆ **Relation** type maps only relation elements such as flows, links or dependencies.
- ◆ **Element** type handles all other elements.

This chart shows the supported System Architect type mappings to the Rhapsody metaclasses:

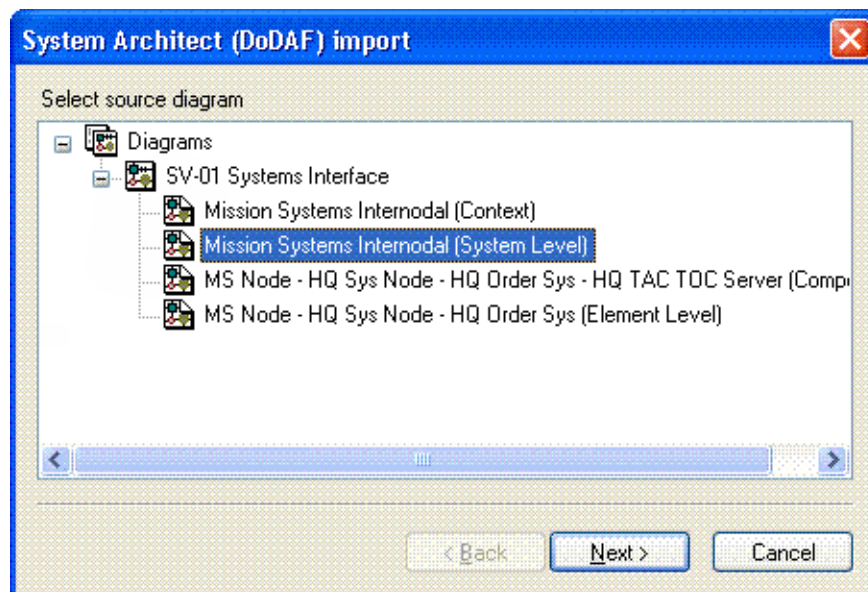
System Architect Type	Rhapsody Metaclass	Comments
SV-1 diagram	Package	Direct mapping through a map file
System Node	Block (SysML element)	Direct mapping through a map file
System Entity	Part (SysML element)	Direct mapping through a map file
System Function	Operation	Final conversion is made by a post processing script
System Interface	Flow	Direct mapping through a map file
System Data exchange	Flow item	Final conversion is made by a post processing script
Data Element	Attribute of a Part (a Part that stands for System entity)	Final conversion is made by a post processing script

Note: All elements that are not created through a direct mapping are mapped to Comments and the related attributes are kept as Tags.

Importing the SA Elements

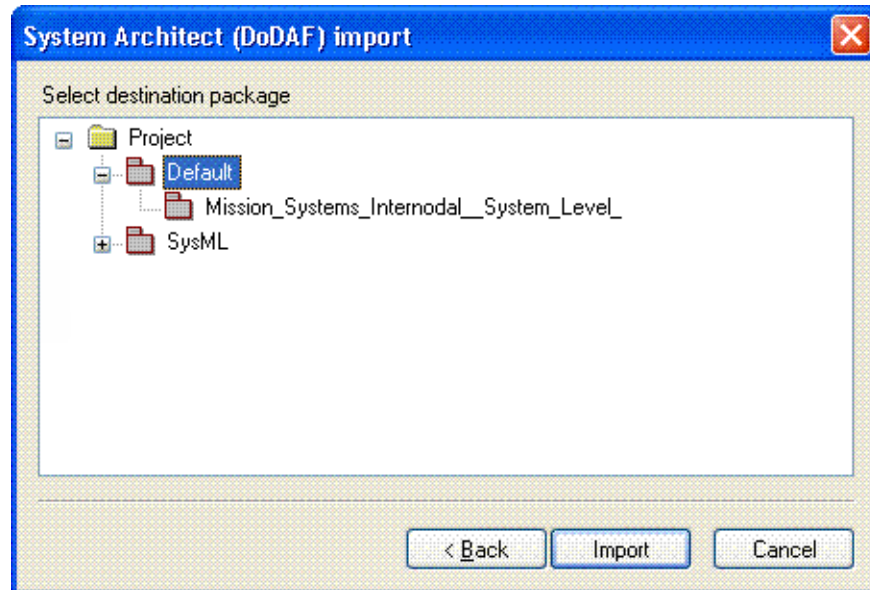
To run the SA Importer, follow these steps:

1. Launch System Architect and load a DoDAF (non-ABM) encyclopedia.
2. Launch Rhapsody.
3. Select the **Tools > Import from System Architect** option. Rhapsody launches the System Architect selection dialog box, shown below.



4. Highlight the SA diagram of interest and click **Next**.

5. In the next dialog box, select the Rhapsody package where the diagram should be placed in your Rhapsody project. Click **Import**.



Only valid Rhapsody elements are imported. The SA Importer puts the selected diagram data into Rhapsody under the selected package and maintains the element hierarchy as it is in the SA encyclopedia. The import operation also adds SysML profile characteristics to the project, unless it was already a SysML profile project.

Converting Imported Data into a Rhapsody Diagram

To convert the imported data into a Rhapsody diagram, follow these steps:

1. Select the package that contains the imported data from SA.
2. From its context menu select **Add New** and the SysML diagram type you want from the menu.
3. In the New Diagram dialog, enter the name of the diagram and check the **Populate diagram** option and press **OK**.
4. The Populate diagram dialog box displays. Select the elements to be added to the new diagram.
5. Press **OK**.

Post Processing Mechanism for SA Users

You can use the post processing mechanism, `SAIntegratorListenerPlug-in`, to perform analysis on the imported data. This Java plug-in is stored in the `AddOn\ProductIntegrator\PostProcessing` directory. Refer to the `readme.txt` in that directory for more information.

Generating a Imported Elements Report

To create an out-of-the-box SysML data flow report on the imported data, follow these steps:

1. Highlight the package holding the imported System Architect elements.
2. Select the **Tools > ReporterPLUS > Report on selected package** option.
3. In ReporterPLUS, select to generate a Word or HTML report and use the `SysMLDataFlowInPackage.tpl` report template.
4. Follow the instructions in the ReporterPLUS wizard to generate the report.

Integration with Teamcenter Systems Engineering

Rhapsody allows you to utilize its modeling abilities in conjunction with Teamcenter Systems Engineering from UGS.

The integration between the two tools allows you to work on the elements common to both Teamcenter and Rhapsody models, such as requirements, use cases, and actors, from within either of the tools. Specifically, you can:

- ◆ Create a new Teamcenter design by importing an existing Rhapsody model
- ◆ Generate Rhapsody models from existing Teamcenter designs.
- ◆ Work on a project in Teamcenter, and then have the common elements updated automatically the next time you open the corresponding model in Rhapsody.
- ◆ Work on common elements in the framework of a Rhapsody model and then save the changes to the Teamcenter repository.

UML or SysML

Out of the box, you can use UML or SysML with the Teamcenter Interface and Rhapsody. For Systems Engineers, this means you can interactively exchange information between Rhapsody models using SysML and the Teamcenter Systems Engineering/Requirements Management environment. For example, you can create and modify SysML elements (such as block and activity) defined in the Rhapsody SysML profile.

In addition to new term SysML elements, you can modify the provided `ElementsMap.xml` file to map a Teamcenter element with a more domain-specific Rhapsody element with one stereotype. This stereotype can be one of the predefined types in Rhapsody or a user-defined stereotype.

To specify UML or SysML, you have to set the default in the `ElementsMap.xml` file, which handles the mapping between Teamcenter and Rhapsody elements.

1. Open the `ElementsMap.xml` file (found in the Rhapsody installation path, for example, `<Rhapsody installation path>\AddOn\TcSE`) in a text editor.

2. Set *either* the **RhapsodyUMLTcSEUML** portion of the `ElementsMap.xml` file to be the default *or* set the **RhapsodySysMLTcSESysML** portion of the file to be the default. Do not set both as the default.
 - For UML, the following figure shows RhapsodyUMLTcSEUML set as the default (Default="Yes"):

```
<RhapsodyTcSEMaps>
  <Map
    Name="RhapsodyUMLTcSEUML"
    Default="Yes"
    Rhapsody_domain="UML"
    TcSE_domain="UML"
    Description="">
    <element
      Rhapsody_MetaClass="Action"
      .
    </Map>
```

- For SysML, the following figure shows RhapsodyUMLTcSEUML set as the default (Default="Yes"):

```
<Map
  Name="RhapsodySysMLTcSESysML"
  Default="Yes"
  Rhapsody_domain="SysML"
  TcSE_domain="SysML"
  Description="">
  <element
    Rhapsody_MetaClass="Class"
    .
  </Map>
</RhapsodyTcSEMaps>
```

Note: If SysML is specified as the default in the `ElementsMap.xml` file and the Teamcenter design contains Rhapsody SysML elements (meaning the map file has elements that contain `Rhapsody_Profile="SysML"`), then when you select **Open Model** or **New Model** in Teamcenter for Rhapsody, Rhapsody SysML will be used. However, if your Teamcenter design does not contain any Rhapsody SysML elements, then Rhapsody UML will be used even if SysML is set as the default in the map file.

Prerequisites for Working with Rhapsody

The prerequisites for working with Rhapsody are as follows:

- ◆ For each Teamcenter project where you would like to use Rhapsody integration, you must first import the provided Rhapsody XML schema.
 - a. Right-click a Teamcenter project and select **Import > Import Schema**.
 - b. Navigate to the Rhapsody installation folder path (for example, <Rhapsody installation path>\AddOn\TcSE\Server).
 - c. Select the appropriate schema:
 - For UML, select **Rhp_Integration_Schema.xml**
 - For SysML, select **RhpSysML_Integration_Schema.xml**
- ◆ Only Teamcenter users with **Architect** permission can use Rhapsody integration from within Teamcenter.

Importing a Rhapsody Model into Teamcenter

To create a new Teamcenter design by importing a Rhapsody model, follow these steps:

1. In Teamcenter, right-click a folder and select **Rhapsody > Import Model**.
2. Select the appropriate Rhapsody file.

A new Teamcenter design will be created, based on the relevant elements in the Rhapsody model. The Rhapsody model will also be attached to the Teamcenter design.

Note

If a Rhapsody model element does not have a corresponding type in Teamcenter, but has children elements that do, these children elements will be ignored and will not be added to the Teamcenter design.

Creating a Rhapsody Model from Existing Teamcenter Project

To create a new Rhapsody model from an existing Teamcenter project, follow these steps:

1. In your Teamcenter project, add the Rhapsody elements that you would like to use. (These elements are available after you have imported the provided Rhapsody XML schema; see [Prerequisites for Working with Rhapsody](#).)
2. Right-click the relevant Teamcenter project folder and select **Rhapsody > New Model**.
3. Browse to where you want to save the Rhapsody model.

A new Rhapsody model is created, and all applicable elements in the folder are added to the Rhapsody model. The name of the new Rhapsody project will be the same as the name of the selected folder.

Note

If an element does not have a corresponding type in Rhapsody, but has children elements that do, these children elements will be ignored and will not be added to the model.

Modifying Shared Elements from within Teamcenter

To modify elements shared with Rhapsody from within Teamcenter:

1. In Teamcenter, right-click the relevant folder and select **Rhapsody > Open Model**.

Note: When the Rhapsody model is opened, it is updated with any changes that other users may have made to the Teamcenter database, or that you may have made in Teamcenter before selecting **Open Model**.

2. Browse to where you want to save the Rhapsody model.
3. Make your changes to the model.
4. Save your changes.

All changes made to shared elements will be applied to the corresponding Rhapsody project as well.

View Corresponding Rhapsody Element

To view the corresponding Rhapsody element for a given Teamcenter element, follow these steps:

1. Right-click the element in Teamcenter and select **Rhapsody > Open Model**.
2. Browse to where you want to save the Rhapsody model.

Rhapsody is launched and the relevant Rhapsody model element is displayed.

Note

This feature works only for the following model elements: object model diagrams, use case diagrams, collaboration diagrams, component diagrams, and sequence diagrams.

Modifying Shared Elements from within Rhapsody

To modify shared elements from within Rhapsody, follow these steps:

1. Make changes to the Rhapsody model.
2. From Rhapsody menu bar, select **File > Synchronize with TcSE**.

This saves any changes that were made to the Rhapsody model and synchronizes the Teamcenter project with the Rhapsody model.

Because changes made through Rhapsody may conflict with changes made by other users to the Teamcenter database, any changes made from within Rhapsody must be merged with changes that may have been made by other users. This is done by using the Rhapsody Base DiffMerge feature. Each time the user opens the corresponding Rhapsody model, a “base” version is also copied to the client machine. When the user saves their changes, Rhapsody compares these changes and any changes contained in the current version from the server against the “base” version. In a rare case, where conflicts are found between the versions, these conflicts can be resolved using the Rhapsody DiffMerge tool. For more information about this tool, refer to the *Rhapsody Team Collaboration Guide*.

Limitations

Note the following limitations:

- ◆ Transition, Flow, and Link relationships are not supported, as well as SysML new terms applied to these types.
- ◆ Attribute and Operation types are not supported, as well as SysML new terms applied to these types.
- ◆ The mapping between a Teamcenter element and a Rhapsody element can only contain one stereotype.

Rhapsody DoDAF Add-on

The Rhapsody *Department of Defense Architectural Framework (DoDAF)* Add-on provides industry standard diagrams and notations for developing DoDAF-compliant architecture models. These diagrams and notations are easily communicated and understood by a wide audience, greatly improving the comparability and communicability of architectures while ensuring the interoperability of systems.

DoDAF is a semantic framework for developing, representing, and integrating architectures in a consistent way for applications for the Department of Defense (DoD). For information on the Department of Defense Architectural Framework (DoDAF) Specification, refer to the documents at www.defenselink.mil/cio-nii/cio/earch.shtml.

Rhapsody DoDAF is part of the System Engineering Add-on component that may have been installed during the Rhapsody installation process (according to your Rhapsody license). Or, if you purchased the Add-on after your initial installation of the Rhapsody product, you must install the Add-on separately with a license key. Refer to the *Rhapsody Installation Guide* for installation instructions and any system requirements.

Note

If you want to import System Architect (SA) DoDAF diagrams as Rhapsody SysML diagrams, see [Importing System Architect's DoDAF Diagrams](#) instructions.

Rhapsody DoDAF Add-on and Profile

The Rhapsody DoDAF Add-on includes a DoDAF Profile, a number of DoDAF helper utilities, a DoDAF Reporter Template, a Microsoft Word Document Template file, a Rhapsody ReporterPLUS License, an image library with a set of public domain graphics for military applications, and a tutorial.

To provide an effective Model Driven Development Solution for creating DoDAF-compliant architectural models, use the Rhapsody DoDAF Add-on together with Rhapsody in conjunction with a sound Systems Engineering Process and Methodology.

The Rhapsody DoDAF Add-on is an independent process, but it also supports a variation of the Harmony development process targeted at the development of DoDAF-compliant architecture

models. The DoDAF Add-on is a template-driven solution that can be customized and extended to meet specific customer requirements and development processes.

By simulating the Rhapsody model, the ability of an architecture to meet its operational goals can be measured, and its effectiveness in comparison with other architecture models can be observed. The operational scenarios captured as event traces can be executed against the model, and the response of the architecture model can be recorded. In addition, using the automated testing capabilities in Rhapsody, robustness of an architecture model can be analyzed, and a suite of functional verification tests can be generated from the model.

To learn more about Rhapsody DoDAF, note these resources:

- ◆ ***Rhapsody DoDAF Tutorial.*** Access the tutorial through the Windows Start menu (Programs > Telelogic > Telelogic Rhapsody Version # > Telelogic Rhapsody DoDAF Add-on > Rhapsody DoDAF Add-on Tutorial) or from within Rhapsody (Help > List of Books).
- ◆ **Instructor-led training for Rhapsody DoDAF.** Contact your Rhapsody sales representative about taking training classes.
- ◆ **Rhapsody DoDAF eLearning course.** Contact your Rhapsody sales representative about taking eLearning courses.

DoDAF Views

DoDAF defines the following views:

- ◆ [Operational View](#), which identifies what needs to be accomplished and who does it
- ◆ [Systems View](#), which relates systems and characteristics to operational needs
- ◆ [Technical View](#), which prescribes standards and conventions
- ◆ All View, which encompasses all of the other views as there are overarching aspects of architecture that relate to the Operation, Systems, and Technical views

Operational View

The Operational view is a description of the tasks and activities, operational elements, and information exchanges required to accomplish DoD missions. DoD missions can include both military missions and business processes.

The Operational view contains graphical and textual elements that comprise an identification of the operational nodes, assigned tasks and activities, and information flows required between nodes. It defines the following aspects of communication:

- ◆ Types of information exchanged
- ◆ Frequency of exchange
- ◆ Tasks and activities supported by the information exchange

Operational views can describe activities and information exchanges at any level of detail and to any breadth of scope that is appropriate. The detail level is driven by the information required to perform the desired analyses. The kind of analysis you want to do determines what kind of information and the level of detail you must put into the Operational view.

Systems View

According to the Department of Defense, a “system” may be partially or fully automated and is defined as “any organized assembly of resources and procedures united and regulated by interaction or interdependence to accomplish a set of specific functions.”

The Systems view relates the system resources to the operational capabilities described in the Operational view. Further detail of the information exchanges described in the Operational view is provided in order to:

- ◆ Translate node-to-node exchanges into system-to-system transactions
- ◆ Communicate capacity requirements
- ◆ Show security protection needs

The Systems view describes systems and interconnections providing for, or supporting, DoD functions. DoD functions include both warfighting and business functions.

The systems, shown in the Systems view, can be existing, emerging, planned, or conceptual, depending on the purpose of the architecture effort. This view may be a reflection of the current state, transition to a target state, or analysis of future investment strategies.

Technical View

The Technical view is the minimal set of rules governing system parts and elements. It governs the following aspects of the parts:

- ◆ Arrangement
- ◆ Interaction
- ◆ Interdependence

The purpose of the Technical view is to ensure a system satisfies a specified set of requirements.

The Technical view provides the basis for the engineering specification of the systems in the Systems view and includes technical standards. The Technical view is the engineering infrastructure that supports the Systems view.

All Views

All Views encompasses all of the other views as there are overarching aspects of architecture that relate to the Operation, Systems, and Technical views.

Products Included in the Rhapsody DoDAF Add-on

The following table lists the products that the Rhapsody DoDAF add-on includes.

Architecture Product	View	Product Name	Product Description
All Views	Package	AllViews	This optional stereotyped package allows you to add in AV products and other views and packages, if desired.
AV-1	All	Overview and Summary Information	This product is typically a text (Word, FrameMaker, HTML) document. You can add AV-1 documents and launch them by clicking on them.
AV-2	All	Integrated Dictionary	This is a DoDAF-generated text product (report).
Operational View	Package		This optional stereotyped package is similar to the All View product. It supports all the operational products.
OV-1	Operational	High-Level Operational Concept Graphic	This high-level graphical/ textual description of the operational concept allows you to import pictures and other operational elements, such as Operational Nodes, Human Operational Nodes, Operational Activities and the relations among them.
OV-2	Operational	Operational Node Connectivity Description	This product shows the connections and flows among operational nodes and operational activities. If desired, the behavior of operational nodes and operational activities can be shown by adding OV-5, OV-6a, OV-6b, and OV-6c diagrams. These diagrams are the primary source of information used by the DoDAF add-on to create the OV-3 diagram.

Architecture Product	View	Product Name	Product Description
OV-3	Operational	Operational Information Exchange Matrix	This product shows information exchanged between nodes, and the relevant attributes of that exchange. OV-3 is generated from the information shown in OV-2 and other operational diagrams. This information is stored as a CSV file and can be added to any product.
OV-5	Operational	Operational Activity Model	This product details the behavior of operational nodes or more commonly, operational activities.
OV-6a	Operational	Operational Rules Model	This product is a textual description of "business rules" for the operation. It is a controlled file. One of three products used to describe the mission objective.
OV-6b	Operational	Operational State Transition Description	This product is a statechart that can be used to depict the behavior of an operational element (node or activity). One of three products used to describe the mission objective.
OV-6c	Operational	Operational Event Trace Description	This product is a sequence diagram that captures the behavioral interactions among and between operational elements and (in the Harmony process) captures the operational contracts among them. One of the three products used to describe the mission objective.
OV-7	Operational	Logical Data Model	This product is a class diagram that shows the relations among Informational Elements (data classes). This is similar to entity relationship diagrams, but is more powerful.
System View	Package		This optional stereotyped package is similar to other views, but contains system elements.
SV-1	Systems	Systems Interface Description	This product is a diagram that contains System nodes, systems, system parts and the connections between them (links). These can be used with or without ports.

Products Included in the Rhapsody DoDAF Add-on

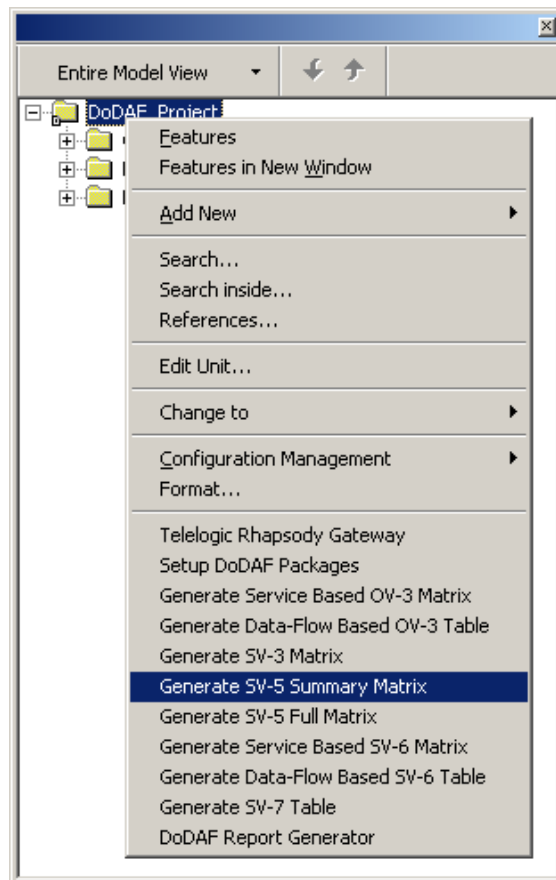
Architecture Product	View	Product Name	Product Description
SV-2	Systems	Systems Communications Description	This product is a diagram that shows the connections among systems via the communications systems and networks.
SV-3	Systems	Systems-Systems Matrix	This product is generated from the information in the other system views. SV-3 assumes that there are links between items stereotyped SystemNode, System, or System Part and represents these in an N ² diagram (system-system matrix).
SV-4	Systems	Systems Functionality Description	This product represents the connection between System Functions and Operational Activities. The connection is made by drawing a Realize dependency line from the System Function to the Operational activity on the diagram. System Functions are mapped onto the system elements that support them by making System Functions parts of the system elements (that is, System Functions are drawn within the other system elements). System elements can also realize system functions. Note that here, as in almost all the other views, you can use Performance Parameters (bound to their constrained elements via <i>anchors</i>) to add performance data. This is summarized in SV-7.
SV-5	Systems	Operational Activity to Systems Function Traceability Matrix	This product is a spreadsheet-like generated view summarizing the relations among system elements (system nodes, systems and system parts), system functions that they support, and the mapping to operational activities.
SV-6	Systems	Systems Data Exchange Matrix	This product shows the information in the flows (information exchanges) between system elements. They may be embedded flows (bound to the links) or they may be flows independent of links. This is a spreadsheet-like generated product.

Architecture Product	View	Product Name	Product Description
SV-7	Systems	Systems Performance Parameters Matrix	This is a generated spreadsheet-like product, showing all the performance parameters and the elements that they constrain.
SV-8	Systems	Systems Evolution Description	This product is the system evolution description. This is an activity diagram (there is a SystemProject element stereotype to serve as the “base” for this activity diagram). SV-8 depicts the workflow for system development, object nodes for products released, and performance parameters for things like start and end dates, slack time, etc.
SV-9	Systems	Systems Technology Forecast	This product is a text document - a stereotype of a Controlled File.
SV-10a, SV-10b, SV-10c	Systems	Systems Rules Model, Systems State Transition Description, Systems Event Trace Description	These products are similar to the OV-6a, OV-6b, and OV-6c products, but they are separately identified, even though they are structurally identical.
SV-11	Systems	Physical Schema	This product is similar to the OV-7 class diagram. This product uses a class diagram to show physical schema (data representation).

DoDAF Add-on Helper Utilities

There are a number of helper utilities provided with the Rhapsody DoDAF Pack. These precompiled helpers assist with common functions. They include helpers to generate the derived products OV-3, SV-3, SV-5, SV-6, and SV-7.

To activate a helper, right-click the applicable model element and select a helper from the pop-up menu. In the following figure, the model element selected is the top-level project folder:



The following table summarizes the helpers and the model elements for which they are available. The **Applicable To** column indicates which model element you must right-click to make the helper appear on the pop-up menu. Refer to the *Rhapsody DoDAF Tutorial* for examples on the use of these helpers.

Helper Name	Applicable To
Setup DoDAF Packages	DoDAF Project
Create OV-2 from Mission Objective	Mission Objective
Create OV-6c from Mission Objective	Mission Objective
Update OV-2 from OV-6c	OV-6c Event Trace
Generate Service Based OV-3 Matrix	DoDAF Project
Generate Data-Flow Based OV-3 Table	DoDAF Project
Generate SV-3 Matrix	DoDAF Project
Generate SV-5 Summary Matrix	DoDAF Project
Generate SV-5 Full Matrix	DoDAF Project
Generate Service Based SV-6 Matrix	DoDAF Project
Generate Data-Flow Based SV-6 Table	DoDAF Project
Generate SV-7 Table	DoDAF Project
DoDAF Report Generator	DoDAF Project

Setup DoDAF Packages

The Setup DoDAF Packages helper is used to configure a new DoDAF project with helpers that are useful in building a DoDAF-compliant architecture model. In addition to doing basic configuration of the project, the helper creates a framework for the DoDAF project. This helper also creates two (empty) OV-1s; one is meant to be “semantic free” and the other “semantic rich.”

Create OV-2 from Mission Objective

The Create OV-2 from Mission Objective helper is used to create an OV-2 Operational Node Connectivity Description product, and associates the OV-2 with the selected mission objective. Operational nodes can be dragged onto the diagram to represent the operational nodes in the OV-2.

Create OV-6c from Mission Objective

The Create OV-6c from Mission Objective helper is used to create an initial OV-6c Operational Event Trace Description product, and associates the OV-6c with the selected mission objective. The OV-6c includes the operational nodes associated with the mission objective as lifelines.

Update OV-2 from OV-6c

The Update OV-2 from OV-6c helper is used to add operational activity allocations, along with needline and information exchanges after realizing the messages in an OV-6c diagram. Ports are added to the operational activities, and interfaces created and attached to the ports. These interfaces form the required and provided interface contracts between operational nodes.

Generate Service Based OV-3 Matrix

For information on the Generate Service Based OV-3 Matrix, see [Generating the OV-3 Operational Information Exchange Matrix](#).

Generate SV-3 Matrix

The Generate SV-3 Matrix is used to create a report that is based on the links between system elements in SV-1s and SV-4s, whether or not ports are used or whether interfaces for those ports are formally defined.

Generate SV-5 Summary Matrix

The Generate SV-5 Summary Matrix helper is used to create a summary to show a system function or system element row if and only if there is a dependency to an operational element. The most common dependency is Realization.

Generate SV-5 Full Matrix

The Generate SV-5 Full Matrix helper is used to create a report to show all system functions and element.

DoDAF Report Generator

The DoDAF Report Generator helper generates a Microsoft Word Document (.doc) file containing DoDAF architecture products from a Rhapsody model. The process of generating a document is largely push button, and requires minimal user interaction. The content of the document is extracted from the Rhapsody model created by the user. The resulting document is organized as follows:

- ◆ AV-1 Overview and Summary Information
- ◆ AV-2 Integrated Dictionary
- ◆ OV-1 High Level Operational Concept Graphic
- ◆ OV-2 Operational Node Connectivity Description
- ◆ OV-3 Operational Information Exchange Matrix
- ◆ OV-5 Operational Activity Model
- ◆ OV-6a Operational Rules Model
- ◆ OV-6b Operational State Transition Description
- ◆ OV-6c Operational Event-Trace Description
- ◆ OV-7 Logical Data Model
- ◆ SV-1 Systems Interface Description
- ◆ SV-2 Systems Communications Description
- ◆ SV-3 Systems-Systems Matrix
- ◆ SV-4 Systems Functionality Description
- ◆ SV-5 Operational Activity to Systems Function Traceability Matrix
- ◆ SV-6 Systems Data Exchange Matrix
- ◆ SV-7 Systems Performance Parameters Matrix
- ◆ SV-8 Systems Evolution Description
- ◆ SV-9 Systems Technology Forecast
- ◆ SV-10a Systems Rules Model
- ◆ SV-10b Systems State Transition Description
- ◆ SV-10c Systems Event-Trace Description

- ◆ SV-11 Physical Schema
- ◆ TV-1 Technical Standards Profile

See [Generating the DoDAF Report from the Architecture Model](#) for more information on the DoDAF Report Generator helper.

Configuring a Rhapsody Project for DoDAF

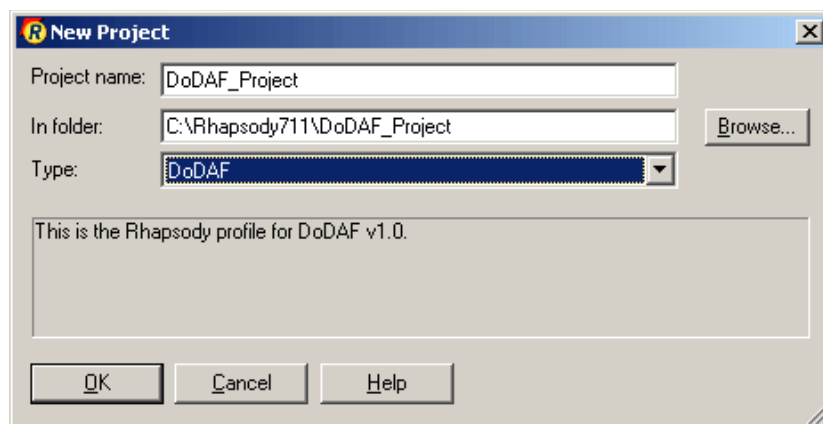
You specify the Rhapsody model elements that are to form the core views in the generated DoDAF documentation. From these, other DoDAF products are derived. The Rhapsody model elements included in the DoDAF generated report are specified using stereotypes provided in the DoDAF profile. The DoDAF profile also provides tags that can be used to specify the location of external data and graphics that should appear in the DoDAF products. The Rhapsody DoDAF Add-on includes a helper utility to create a Rhapsody project with the DoDAF profile preloaded so you.

Creating a Rhapsody DoDAF Project

To create a Rhapsody DoDAF project, follows these steps:

1. Launch Rhapsody and select **File > New**.
 - ◆ On the New Project dialog box:
 - ◆ Type in your project name.
 - ◆ Specify a location.
 - ◆ Select **DoDAF** as the type from the drop-down list, as shown in the following figure.

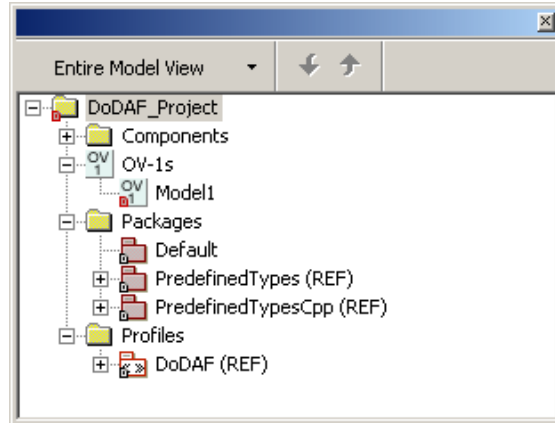
Note: This **DoDAF** type (or profile) is provided by the Rhapsody DoDAF Add-on in order to help you customize and extend the Rhapsody product to support a Domain Specific Language (DSL), which lets you work with DoDAF terms, diagrams, and artifacts rather than UML terms, diagrams, and artifacts.



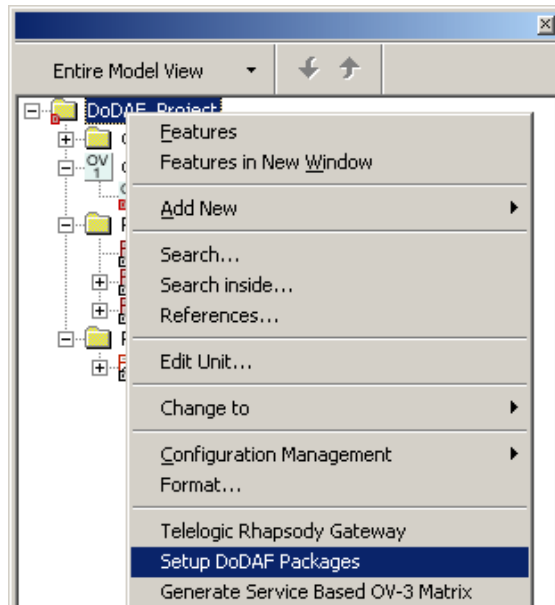
2. Click **OK**.

3. If the folder you specified does not exist, you are asked if you want to create it. Click **Yes**.
4. Expand the **Packages** and **Profiles** folders in the Rhapsody browser, as shown in the following figure.

Note: Profiles shown in your browser may vary depending on your site properties and product licensing.

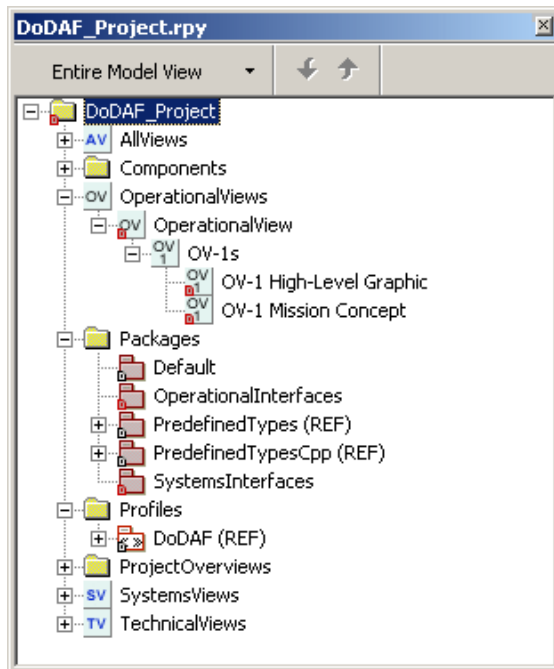


5. To initialize the DoDAF project, right-click the top-level project name (**DoDAF_Project** in the example) and select **Setup DoDAF Packages**, as shown in the following figure:



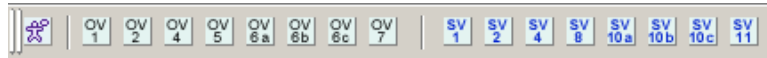
Note: If you do not see **Setup DoDAF Packages**, see [Manually Adding the DoDAF Helpers](#).

6. Click **OK** to dismiss the confirmation dialog box.
7. Look at your Rhapsody browser and notice what **Setup DoDAF Packages** did for your project. For example, expand the new **Operational Views** category until you see the **OV1-High-Level Graphic** and **OV-1 Mission Concept** mission object diagrams, as shown in the following figure:



Diagrams Toolbar for a Rhapsody DoDAF Project

The **Diagrams** toolbar, as shown in the following figure, provides quick access to the graphic editors, where diagrams are created and edited. The **DoDAF** profile that you used to create your DoDAF project displays a **Diagrams** toolbar that is unique to this profile. The available diagrams are represented as icons on the toolbar across the top of the Rhapsody window. To hide or display this toolbar, select **View > Toolbars > Diagrams**.



The following are all of the diagram types with their icons, as displayed on the **Diagrams** toolbar for a project created with the **DoDAF** profile. Note that there are no icons on the **Diagrams** toolbar for the following:

- ◆ OV-3 is a derived product and is generated from the model.
- ◆ SV-3, SV-5, SV-6, and SV-7 are derived products and are generated from the model.
- ◆ AVs and TVs are controlled files and are added to the model.



Project Overview Diagram



OV-1: Hi Level Operational Graphic



OV-2: Operational Node Connectivity Diagram



OV-4: Organizational Relationships Diagram



OV-5: Operational Activity Diagram



OV-6a: Operational Rules Model



OV-6b: Operational State Transition Description Diagram



OV-6c: Operational Event-Trace Description Diagram



OV-7: Logical Data Model



SV-1: System Interface Description



SV-2: System Communication Description Diagram



SV-4: System Functionality Description Diagram



SV-8: System Evolution Description Diagram



SV-10a: Systems Rules Model



SV-10b: System State Transition Description Diagram



SV-10c: System Event-Trace Description Diagram



SV-11: Physical Schema Diagram

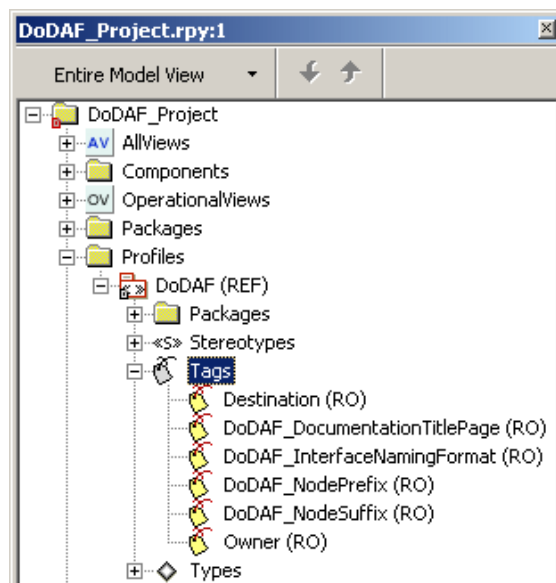
DoDAF Tags

The DoDAF profile allows for the application of tags to diagrams, elements, and relations. These tags can be accessed from the Rhapsody Browser or from the diagram, element, or relation itself.

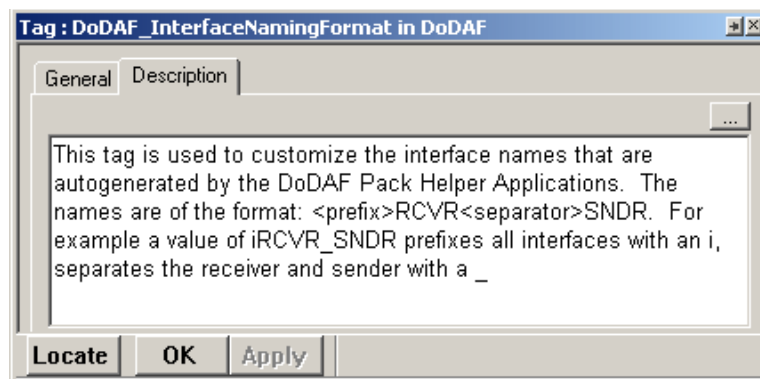
Accessing tags through the Rhapsody browser

To access the tags from the browser, follow these steps:

1. Navigate to the applicable package stereotype by expanding the folders, as shown in the following figure:



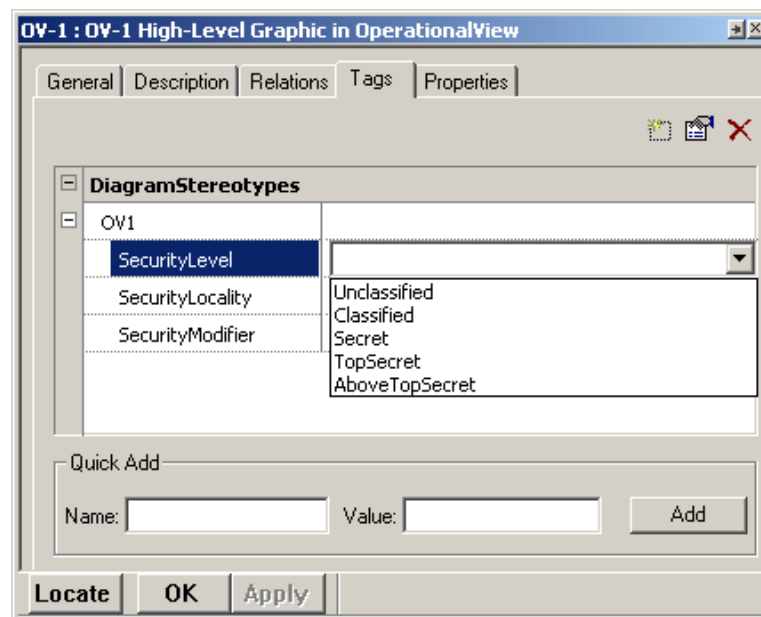
2. Double-click a tag to its Features dialog box where you can view information applicable to it, such as its description, as shown in the following figure:



Accessing tags from a diagram, element, or relation

To access a tag from a diagram, element, or relation itself, follow these steps:

1. Right-click the item you want in the Rhapsody browser and select **Features** to open its Features dialog box.
2. To assign a value to a tag, on the **Tags** tab, click in the cell to the right of the tag select a value from a drop-down list, as shown in the following figure:



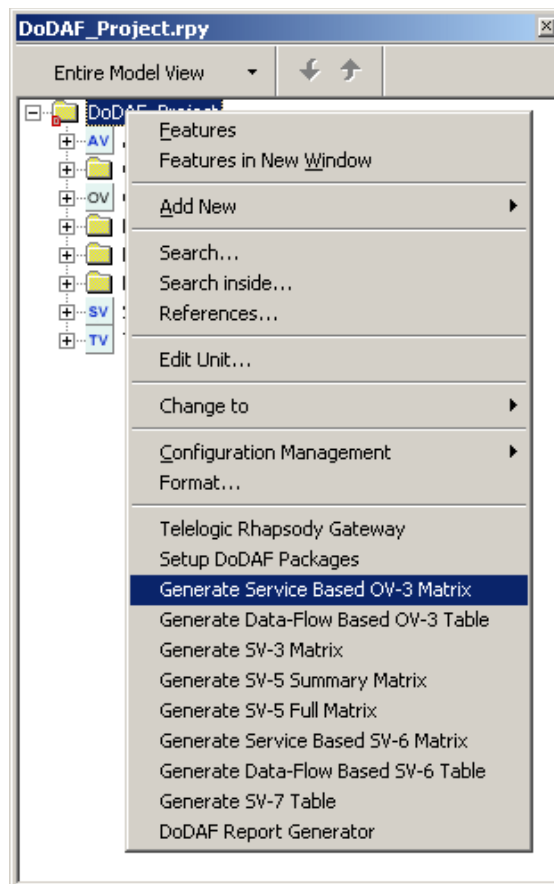
Note: To add a tag locally (meaning for use only by the current element, use the **Quick Add** group: Enter a name for the tag and a value, then click the **Add** button.

Generating the OV-3 Operational Information Exchange Matrix

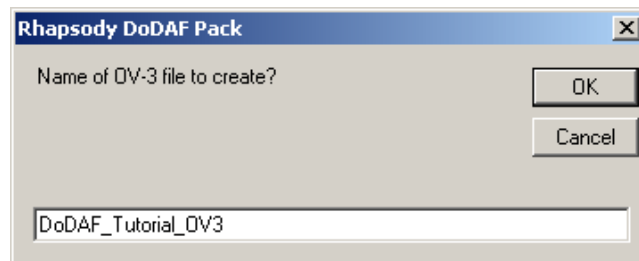
The OV-3 Operational Information Exchange Matrix provides a detailed report of the information exchange between operational nodes.

To automatically generate the OV-3 Matrix, use the following steps:

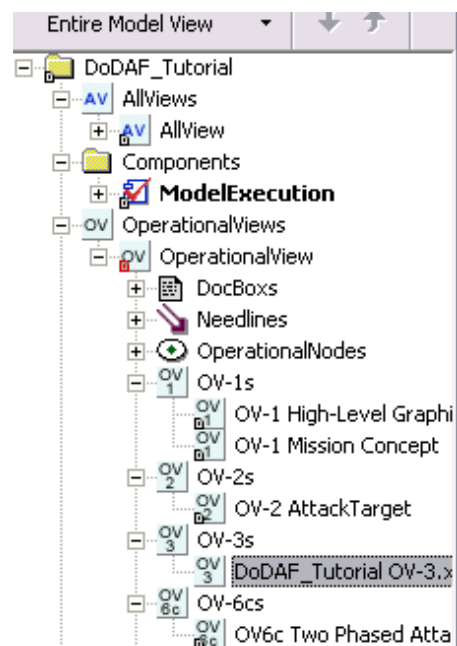
1. Right-click the top-level project folder (**DoDAF_Project** in the example) and select **Generate Service Based OV-3 Matrix**, as shown in the following figure:



2. Enter the name for the OV-3 file, as shown in the following figure, and click **OK**.



3. Wait while the matrix file is generated. Click **OK** to dismiss the confirmation message.
4. The browser now reflects the changes made.



Generating the DoDAF Report from the Architecture Model

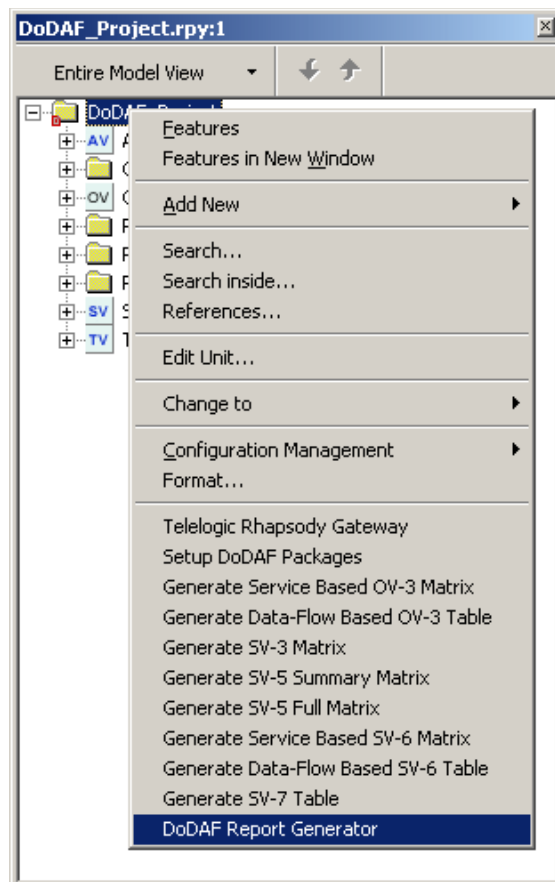
Using another helper, you can generate a DoDAF report from the architecture model that includes the following architecture products: AV-1, AV-2, OV-1, OV-2, OV-3, OV-5, OV-6b, and OV-6c. The AV-2 is generated for you automatically from the architecture model data. Keep in mind that the Rhapsody model itself is a dynamic and searchable AV-2 including all elements of the architecture model.

Note

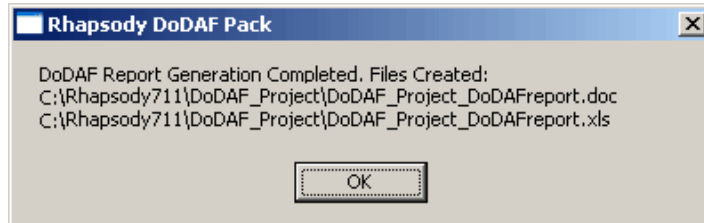
You must already have the Microsoft products mentioned in this section. They are not provide by the Rhapsody DoDAF Add-on or the Rhapsody product.

To generate the DoDAF report, follow these steps:

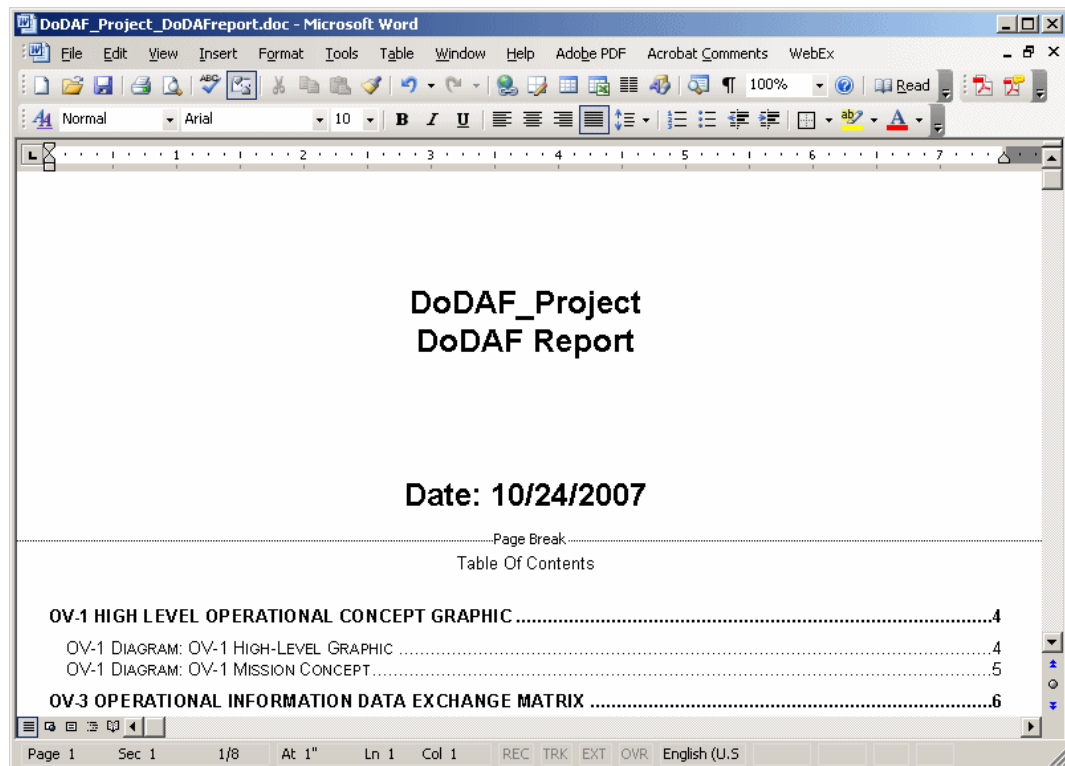
1. Right-click your top-level project folder on the Rhapsody browser and select **DoDAF Report Generator**, as shown in the following figure:



2. Wait while your files are generated.
The DoDAF Report Generator window appears and ReporterPLUS starts to load the model and generate the files. This process may take a few minutes. When completed, Rhapsody display where the files are stored, which will be in the Rhapsody project directory, as shown in the following figure:



3. Click **OK** to dismiss the dialog box.
4. Look at the files generated.
 - The document is formatted and displayed in Microsoft Word, as shown in the following figure:



- The OV-3 spreadsheet is saved as a worksheet in an Excel file

It is possible to navigate to the definition of any interfaces displayed in the OV-3 Matrix. Double-click an interface name in the OV-3 Excel file or in the OV-3 Word document will bring you to the corresponding definition of the interface in the AV-2 Data Dictionary. Use the Back button in Word's Web toolbar to return to the OV-3.

Note

The Word document can be converted to other formats including HTML and PDF using third party software (not provided in Rhapsody).

Limitations

The Rhapsody DoDAF add-on has the following limitation. To use a mapping between Rhapsody Diagrams (artifacts) and DoDAF Architecture products different than the one described here, the Helper Program and ReporterPLUS Template must be modified.

Troubleshooting

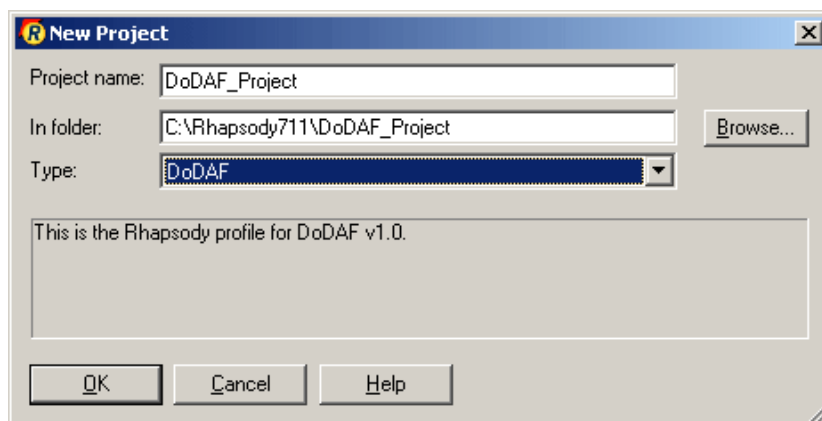
This section provides you with information that you can use for troubleshooting purposes.

Verifying the Rhapsody DoDAF Add-on Installation

If you are having a problem with the Rhapsody DoDAF Add-on or if it does not appear in the Rhapsody product as mentioned in this section, you should verify that it has been installed.

Use any of the following methods to verify that Rhapsody DoDAF Add-on has been installed:

- ◆ See if there is a path of the Add-on from the Windows Start menu. Choose **Programs > Telelogic > Telelogic Rhapsody *Version #* > Telelogic Rhapsody DoDAF Add-on**.
- ◆ See if the Add-on has been installed with the Rhapsody product. Typically that path would be <Rhapsody installation path>\Telelogic Rhapsody DoDAF Add-on.
- ◆ Create a project in Rhapsody and see if the **DoDAF** type (profile) is available, as shown in the following figure.



If you find that the Rhapsody DoDAF Add-on is not installed on your system, then you must install it. You need a license key for the Rhapsody DoDAF Add-on if it is not already a part of your Rhapsody license. Refer to the *Rhapsody Installation Guide* for more information about installing the Rhapsody DoDAF Pack.

If the Rhapsody DoDAF Add-on is installed on your system but you are having problems with it, you may have to un-install and then re-install the Add-on to repair your installation if it has been damaged.

Manually Adding the DoDAF Helpers

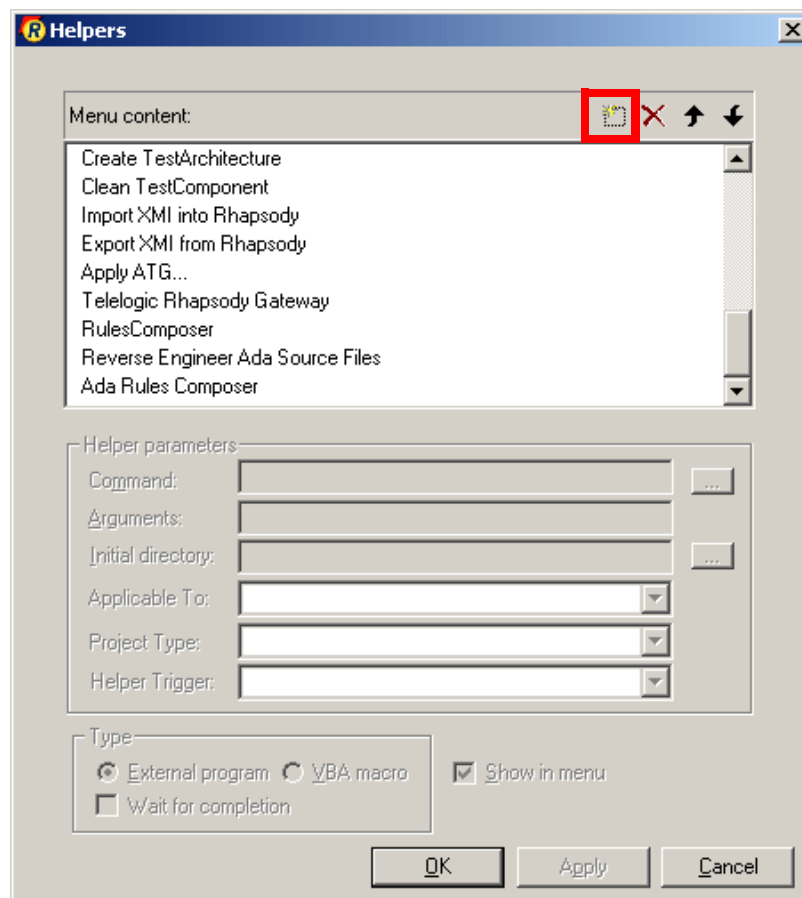
Typically, when you install the Rhapsody DoDAF Pack, the installer will automatically configure the Rhapsody DoDAF Add-on helpers for you. There are a number of helpers provided with the Rhapsody DoDAF Pack. The following table summarizes the helpers and their settings:


Helper Name	Command	Arguments	Applicable To
Setup DoDAF Packages	<Path>\DoDAFPack.exe	-dodaf -i	DoDAF
Create OV-2 from Mission Objective	<Path>\DoDAFPack.exe	-dodaf -uc	MissionObjective
Create OV-6c from Mission Objective	<Path>\DoDAFPack.exe	-dodaf -sd	MissionObjective
Update OV-2 from OV-6c	<Path>\DoDAFPack.exe	-dodaf -d	OV-6c
Generate Service Based OV-3 Matrix	<Path>\DoDAFPack.exe	-dodaf -ov3m	DoDAF
Generate Data-Flow Based OV-3 Table	<Path>\DoDAFPack.exe	-dodaf -ov3t	DoDAF
Generate SV-3 Matrix	<Path>\DoDAFPack.exe	-dodaf -sv3	DoDAF
Generate SV-5 Summary Matrix	<Path>\DoDAFPack.exe	-dodaf -sv5short	DoDAF
Generate SV-5 Full Matrix	<Path>\DoDAFPack.exe	-dodaf -sv5long	DoDAF
Generate Service Based SV-6 Matrix	<Path>\DoDAFPack.exe	-dodaf -sv6m	DoDAF
Generate Data-Flow Based SV-6 Table	<Path>\DoDAFPack.exe	-dodaf -sv6t	DoDAF
Generate SV-7 Table	<Path>\DoDAFPack.exe	-dodaf -sv7	DoDAF
DoDAF Report Generator	<Path>\DoDAFPack.exe	-dodaf -report	Project

If you find that any of the DoDAF helpers are missing from the submenus in Rhapsody, first make sure you have verified the Rhapsody DoDAF Add-on installation as described in [Verifying the Rhapsody DoDAF Add-on Installation](#). Once you have verified that it has been installed, you can manually configure the helpers using the instructions that follow. While verifying the installation, make sure you have noted the path where the Rhapsody DoDAF Add-on is installed.

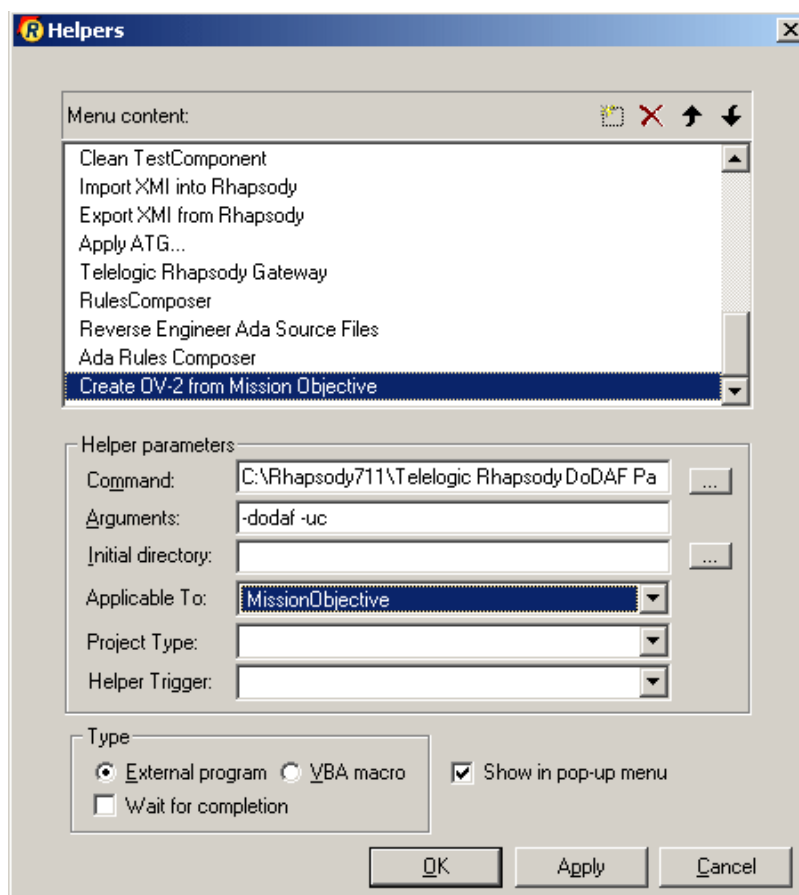
To install the Rhapsody DoDAF Add-on helpers, follow these steps:

1. Select **Tools > Customize** to open the Helpers dialog box, as shown in the following figure:





2. Use the New icon  to create a new helper, and enter the appropriate helper name from the table above.
 - a. Set the **Command**, **Arguments**, and **Applicable To** boxes as listed for a helper on the table above. For the command, replace <Path> with the path where your Rhapsody DoDAF Add-on is installed.
 - b. Select the **External program** option button and select the **Show in pop-up menu** check box.

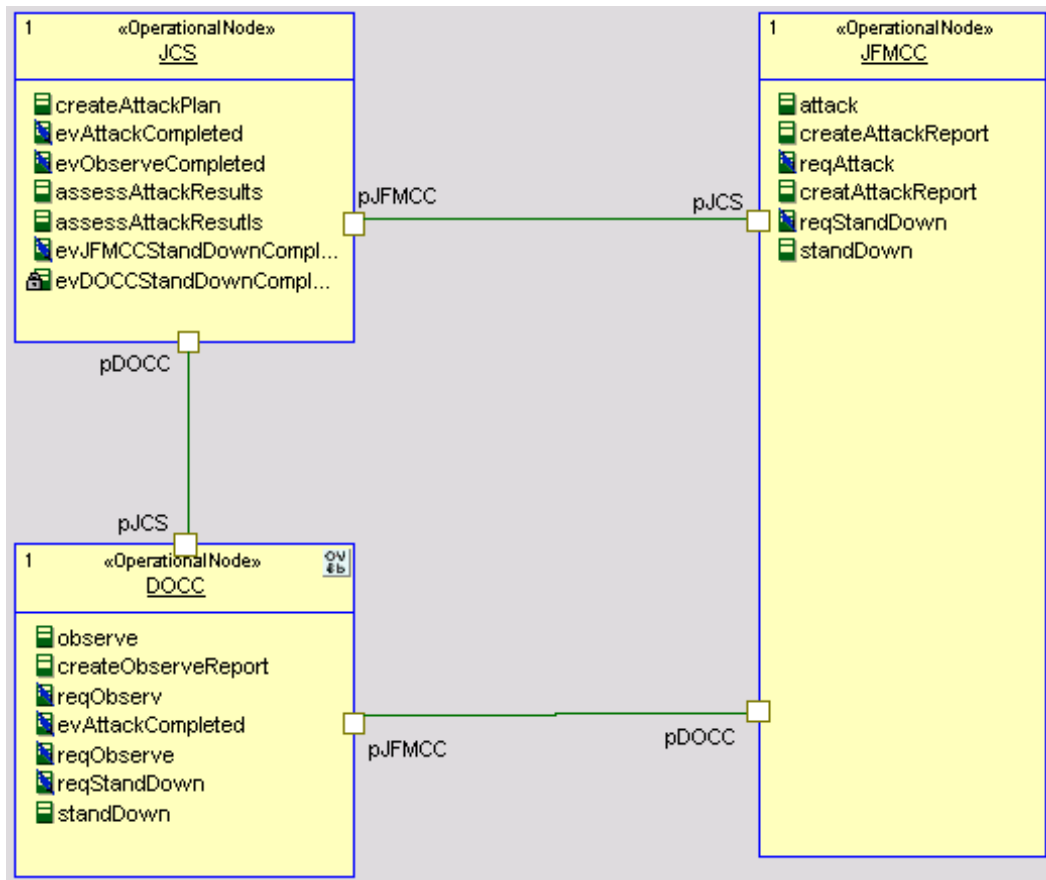
The following figure shows an added helper:



3. Click **Apply**.
4. If needed, configure another helper by using the above steps, or click **OK** to close the Helpers dialog box.



Correcting Messages that Appear as Mission Objectives

After updating the OV-2 from an OV-6c, you should check the OV-2 and make sure the messages between nodes are correctly appearing as events. If you find a message going between operational nodes appears in the OV-2 as an operation rather than an event, this indicates the message type was not changed to Event in the OV-6c diagram. The following figure shows an example of `evJFMCCStandDownCompleted` appearing with the private operation symbol , but should have the event symbol .

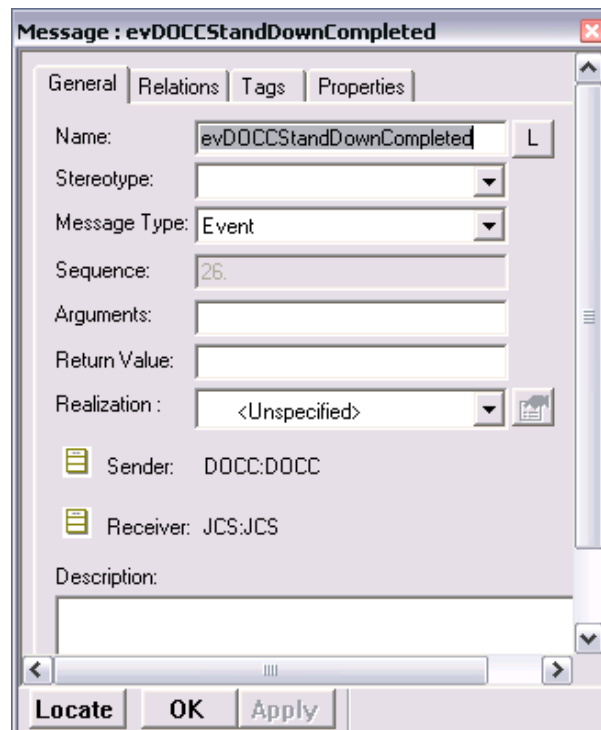


To fix this problem, follow these steps:

1. Expand the Rhapsody browser to see the operation under the appropriate operational node. For example, expand the folder **Project_1 > OperationalViews > Operational View > Operational Nodes > JCS >Operations >evDOCCStandDownCompleted**.
2. Delete the operation by right-clicking the incorrect operation and selecting **Delete from Model**.
3. Click **Yes** to confirm your action.
4. Open the OV-6c diagram and check all the messages with the message name in question.

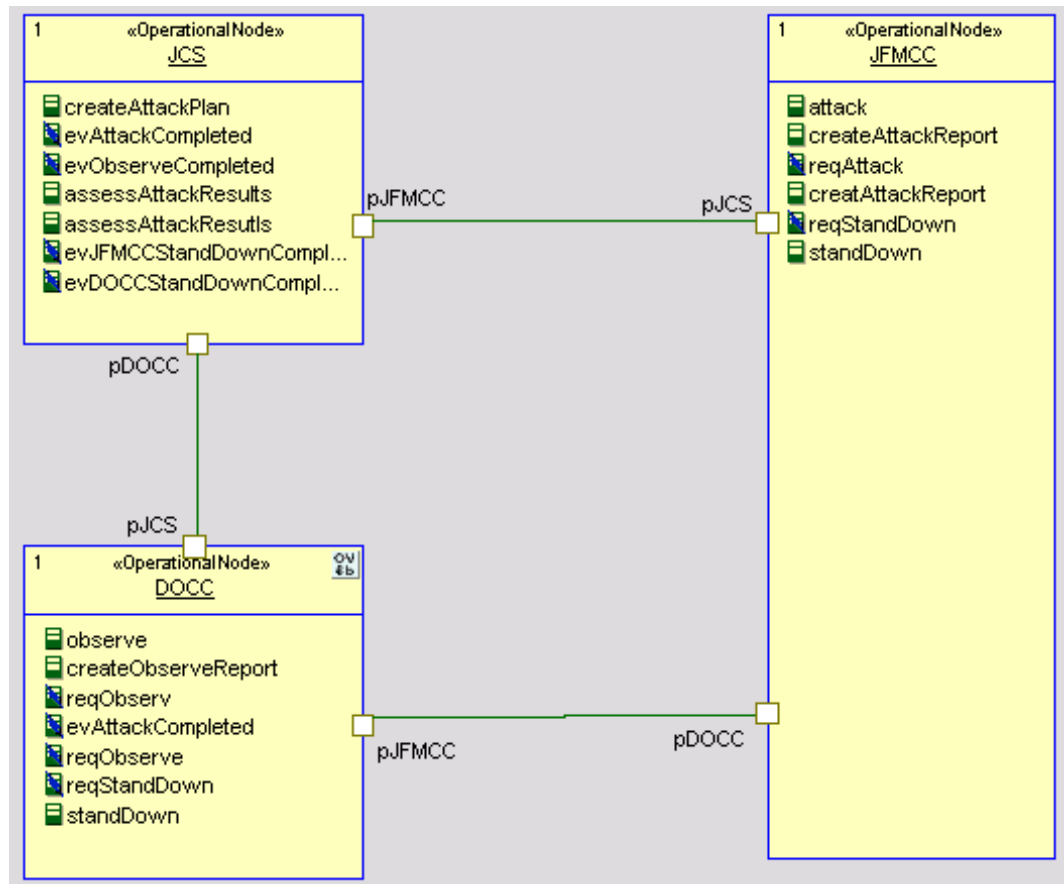
Note: To spot the problematic message, make sure messages that connect operational nodes have a hollow arrowhead , and mission objective messages (messages that start and end on the same node) have a solid arrowhead .

5. To change the incorrect message, change the message type:
 - a. Double-click the message to open the Features dialog box.
 - b. On the **General** tab, in **Message Type** box, select **Event**, as shown in the following figure:



6. With the message selected, choose **Edit > Auto Realize** to realize the message.

7. Right-click **OV-6c Two Phased Attack** on the Rhapsody browser and select **Update OV-2s from OV-6c**. You have repaired the OV-6c and OV-2 diagrams, as shown in the following.



View, Caption, or Table of Figures is Missing from Document

If the final document does not include an OV-3 matrix, figure captions, or table of figures, the macros in the DoDAFReportRTF.dot Word document template file are not being executed. Make sure the security settings for Word are set to medium security to allow macros to be run. In Word, you can change the security setting using **Tools > Macro > Security**.

Microsoft Word may prompt you to enable macros in the generated DoDAF report when it is opened. You will need to select **Enable Macros** button on this dialog box when you open the DoDAF report in order for the macros to run.

Rhapsody MODAF Add-on

The United Kingdom's Ministry of Defence Architecture Framework (known as MODAF) provides standards for enterprise architecture.

Enterprise architecture is the practice of applying a comprehensive and rigorous method for describing a current and/or future structure and behavior for an organization's processes, information systems, personnel, and organization sub-units, so that they align with the organization's core goals and strategic direction. It is effectively a structured approach to describing how a business works or is intended to work so that it can reach its primary objectives. Enterprise architecture is used typically by the military for capability procurement, by governments, and by large businesses.

An *architecture framework* is a specification of how to organize and present an enterprise architecture. It provides a means to present and analyze the enterprises problems. It does not generally tell you how to do something. Architecture frameworks tend to consist of a standard set of viewpoints that represent different aspects of an organization's business as it relates to a particular objective. In the context of MODAF, this implies a systems of systems approach as the analysis is complex and wide-ranging.

Large organizations use MODAF because it enables and facilitates the management of complex enterprise-wide implementations and promotes collaborative architecture development. While MODAF provides an enterprise architecture framework, it is not an architecture. The architecture is the result of using an architecture framework. Therefore, you must use MODAF in conjunction with a knowledge management approach and process that can regulate both the framework and the data.

For all organizations that use MODAF, a key feature of this framework is its goal to help estimate and reduce costs across all projects involved with the enterprise. In the context of military enterprises, the Ministry Of Defence (MOD) sees MODAF as key to the success of its Network Enabled Capability (NEC) goal.

Large frameworks have collections called viewpoints that contain views (also known as products). The viewpoints are inter-related and use elements of each others views. Modeling helps to manage the complexity and retain consistency between the views.

MODAF incorporates aspects of the widely used United States Department of Defense Architectural Framework (known as DoDAF). However, MODAF extends the scope of DoDAF to include viewpoints that reflect the interests of planners and procurement organizations. While DoDAF has four viewpoints (Operational, Systems, Technical, and All Views), MODAF has six with the addition of the Strategic and Acquisition viewpoints. MODAF uses the same views that it shares with DoDAF, though some may work differently in MODAF. For more information on DoDAF, see [Rhapsody DoDAF Add-on](#).

MODAF is defined as a UML profile. See <http://www.modaf.org.uk/> for details of the MODAF MetaModel (or M3, as it is known). Of the 35 views in MODAF, approximately 22 are expressible in UML. The remaining views (except AcV-1 and AcV-2, which are not supported in the Rhapsody MODAF Pack) are either information that can be extracted from the model using the Tables and Matrices functionality in Rhapsody or are text documents that can be added to the model.

For more information about MODAF, see the following Web sites:

- ◆ For technical and introductory material, go to <http://www.modaf.org.uk>. This Web site contains the official online documentation for MODAF.
- ◆ For the history of MODAF, go to <http://www.modaf.com>.

Rhapsody MODAF Pack

The Rhapsody MODAF Add-on includes a MODAF profile, MODAF helper utilities, a model library to enable customization of certain table/matrix views so that you can define your own table/matrix view layouts and add your own custom table/matrix views, a MODAF ReporterPLUS template, a Rhapsody ReporterPLUS license, a set of icons, and an image library with a set of public domain graphics for military applications. In addition, a sample project is available in `<Rhapsody installation path>\Telelogic Rhapsody MODAF Pack\MODAF_ExampleModel`.

The Rhapsody MODAF Add-on may have been installed during the Rhapsody installation process (according to your Rhapsody license). Or, if you purchased the MODAF Add-on after your initial installation of the Rhapsody product, you must install it separately with a license key. You must use the Rhapsody installation wizard's Modify option to install the MODAF Add-on after the initial Rhapsody installation. Refer to the *Rhapsody Installation Guide* for installation instructions and any system requirements.

When you create a new project in Rhapsody, you identify a project type by selecting a profile, in this case, the MODAF profile. This means that you create your project so that it contains model elements that are customized for your specific domain or purpose. See [Configuring a Rhapsody Project for MODAF](#).

To provide an effective Model Driven Development Solution for creating MODAF-compliant architectural models, use the Rhapsody MODAF Add-on together with Rhapsody in conjunction with a sound Systems Engineering Process and Methodology.

The Rhapsody MODAF Add-on is process independent, but it can support a variation of the Harmony development process targeted at the development of MODAF-compliant architecture models. The MODAF Add-on is a template-driven solution that can be customized and extended to meet specific customer requirements and development processes. For information about Harmony, see [Harmony Process and Toolkit](#).

MODAF Viewpoints

Large frameworks have collections of common views called viewpoints (for example, the Systems viewpoint). Viewpoints are heavily interrelated as they use elements of each others views (for example, the SV-1 view has an element called a System that can be used or referred to in many other views).

Note that a view is also called a product.

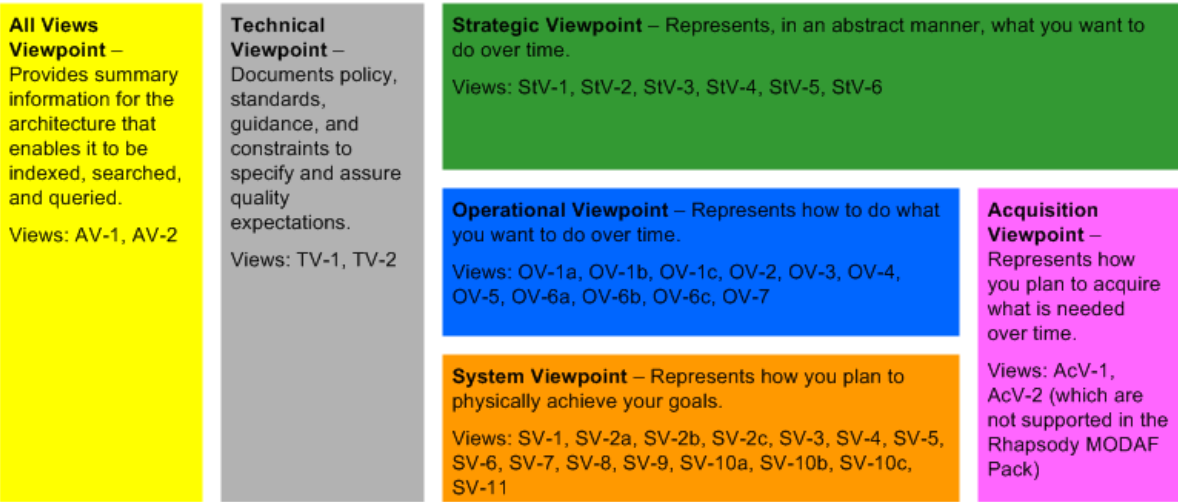
The MODAF viewpoints are:

- ◆ [All Views Viewpoint](#). In both DoDAF and MODAF, and along with the Technical viewpoint, All Views (AV) provides summary information for the architecture that enables it to be indexed, searched, and queried. All Views encompasses all of the other views as there are overarching aspects of architecture that relate to the Strategic, Operation, Systems, Acquisition, and Technical viewpoints.
- ◆ [Strategic Viewpoint](#). Specific to MODAF, the Strategic viewpoint (StV) represents, in an abstract manner, what you want to do over time. It documents the strategic picture of how a capability (for example, a military capability) is evolving in order to support capability deployment and equipment planning. The Strategic, Operational, and Systems viewpoints have a layered relationship.
- ◆ [Operational Viewpoint](#). In both DoDAF and MODAF, the Operational viewpoint (OV) documents the operational processes, relationships, and context to support operational analyses and requirements development. The Operational, Strategic, and Systems viewpoints have a layered relationship.
- ◆ [Systems Viewpoint](#). In both DoDAF and MODAF, the Systems viewpoint (SV) represents how you plan to physically achieve your goals. It documents system functionality and interconnectivity to support system analysis and through-life management (meaning it relates systems and characteristics to operational needs). The Systems, Strategic, and Operational viewpoints have a layered relationship.

- ◆ **Acquisition Viewpoint.** Specific to MODAF, the Acquisition viewpoint (AcV) is partly derived from elements of the Strategic viewpoint and provides information for the Operational and Systems viewpoints. The Acquisition viewpoint represents acquisition program dependencies, timelines, and the status of MOD Defence Lines of Development (DLOD, equivalent to U.S. Department of Defense DOTMLFPs) status so that the various MOD programs are managed and synchronized correctly. Note that the AcV-1 and AcV-2 views are not supported in the Rhapsody MODAF profile.
- ◆ **Technical Viewpoint.** In both DoDAF and MODAF, and along with the All View viewpoint, this viewpoint documents policy, standards, guidance, and constraints to specify and assure quality expectations. It also covers all the other viewpoints.

The following illustration shows the MODAF viewpoints and how they relate to each other. In addition, each viewpoint includes a listing of their views.

MODAF Viewpoints and Their Views



You create viewpoints and views as needed. Which viewpoint you start with is up to you. Note that the viewpoints and their views do not have to be and are not expected to be created all at the same time.

All Views Viewpoint

The All Views viewpoint encompasses all of the other viewpoints as there are overarching aspects of architecture that relate to the Strategic, Operational, Systems, Acquisition, and Technical viewpoints. All Views records what has happened and what should happen going forward. It provides a dictionary (through the AV-2 Integrated Dictionary view) for the architecture that guides the current developers of the architecture and helps future developers understand the framework going forward. This viewpoint is critical to the future success of the current MODAF architecture and any future architecture. Therefore, you should always use the views in this viewpoint for MODAF.

Typical stakeholders of the All Views viewpoint are enterprise planners.

Strategic Viewpoint

The Strategic viewpoint is a description of the strategic picture of how capability (for example, military capability) is evolving in order to support capability management and equipment planning. It contains capability management and shows how capabilities map to operational concepts over time. Its main intent is to analyze the areas of capability gaps, overlaps, and redundancies, and to map capabilities to organizations and platforms. There is no reference to implementation in this viewpoint.

Typical stakeholders of the Strategic viewpoint are high-level planners, policy makers, and analysts.

Operational Viewpoint

The views in the Operational viewpoint can describe activities and information exchanges at any level of detail and to any breadth of scope that is appropriate in logical terms. The detail level is driven by the information required to perform the desired analyses. The kind of analysis you want to do determines what kind of information and the level of detail you must put into the Operational viewpoint. The OV views re-use the capabilities defined in the StV views within content of an operation or scenario. The OV views are used during the various point of an enterprise's lifecycle, including the creation of current and future requirements, and during the planning phase for the operation.

Typical stakeholders of the Operational viewpoint are operation planners.

Systems Viewpoint

The Systems viewpoint relates the system resources to the operational capabilities described in the Operational viewpoint. The views in the viewpoint describe the resources that help you archive the desired capability. They describe the system resources available and their interactions with each other. This includes the matter of human involvement in the operation of systems.

The systems shown in the Systems viewpoint can be existing, emerging, planned, or conceptual, depending on the purpose of the architecture effort. This viewpoint may be a reflection of the current state, transition to a target state, or analysis of future investment strategies.

A primary use of the views in the Systems viewpoint is to develop system solutions that address user requirements and therefore system requirements.

Typical stakeholders of the Systems viewpoint are members of an organization's acquisition group and its associated suppliers.

Acquisition Viewpoint

The Acquisition viewpoint details how the various identified systems will be acquired over time as part of programs. This includes identifying dependencies among projects and capability integration across DLODs (in military endeavors). Note that although you can create views for this viewpoint, the Acquisition viewpoint's AcV-1 and AcV-2 views are not supported in the Rhapsody MODAF Pack.

Typical stakeholders of the Acquisition viewpoint are those involved in capability management and acquisition.

Technical Viewpoint

The purpose of the Technical viewpoint is to ensure a system satisfies a specified set of requirements. The views in this viewpoint cover governance (standards, rules, policy, and so forth) for all aspects of the architecture.

The Technical viewpoint provides the basis for the engineering specification of the systems in the Systems view and includes technical standards (though they do not have to be "technical"). The Technical viewpoint is the engineering infrastructure that supports the Systems viewpoint.

Typical stakeholders of the Technical viewpoint are policy makers and those charged with maintaining core interoperability standards.

Views Included in the Rhapsody MODAF Pack

The following table lists the views (within their viewpoints) included in the Rhapsody MODAF Pack.

For more information about these views, go to the official online documentation Web site for MODAF at <http://www.modaf.org.uk>.

Architecture Viewpoint/View	View	View Name	Description
All Views Viewpoint	Package		The All Views viewpoint contains views that provide overview and nomenclature.
AV-1	All	Overview and Summary Information	This view is typically a text document (for example, Word, FrameMaker, HTML) that provides overview and summary information for the operations and capabilities being considered for the enterprise. It scopes the architecture and gives it context. You can add AV-1 documents and launch them by clicking on them.
AV-2	All	Integrated Dictionary	This view presents all the elements used in an architecture as a standalone structure, generally using a specialization hierarchy. It should provide a text definition for each one and references the source of the element (for example, MODAF Ontology, IDEAS Model, local, and so forth).
Strategic Viewpoint	Package		The Strategic Viewpoint contains views that detail military capabilities and how they evolve to be used by various organizations.
StV-1	Strategic	Enterprise Vision	This view defines the strategic context for a group of enterprise-level capabilities. It takes the overall enterprise vision and goals of the architects and enables them to relate these to realizable capabilities. StV-1 used to be a textual document but is now better represented in a more structured format as UML structure or class diagrams.

Architecture Viewpoint/View	View	View Name	Description
StV-2	Strategic	Capability Taxonomy	<p>This view models capability hierarchies. It enables users to organize capabilities in the context of an enterprise phase, showing required capabilities for current and future enterprises. It specifies all the capabilities that are referenced throughout the current architecture and possibly other reference architectures.</p> <p>StV-2 is realized using UML structure or class diagrams.</p>
StV-3	Strategic	Capability Phasing	<p>This view shows when capabilities are expected to be used. It maps capabilities to time periods. It is also used to perform gap/overlap and redundancy analysis.</p> <p>StV-3 shows a customized table view, with the user defining the time periods.</p>
StV-4	Strategic	Capability Dependencies	<p>Similar to StV-2, this view shows the capability dependencies and logical groupings of capabilities (capability clusters) of the capabilities described in the StV-2.</p> <p>StV-4 is realized using UML structure or class diagrams.</p>
StV-5	Strategic	Capability to Organization Deployment Mapping	<p>This view details what capabilities are mapped to what systems at any particular time and the fulfilment of capability requirements, in particular by network-enabled capabilities. Use this view to perform gap/overlap analysis and interoperability analysis, validate that capabilities have been realized in Systems, and help provide system requirements documents (SRDs) for Systems views.</p> <p>StV-5 is realized using UML class or structure diagrams.</p>

Architecture Viewpoint/View	View	View Name	Description
StV-6	Strategic	Operational Activity to Capability Mapping	<p>This view describes the mapping between the capabilities required by an enterprise and the operational activities that those capabilities support. Use this view to map capabilities to Operations and ensure that all capabilities are fulfilled and can be traced to Operational Activities.</p> <p>StV-6 is derived from UML class diagrams with dependencies. It can be completed once you have done some of the Operational views.</p> <p>StV-6 is a major reason to use tables/matrix layouts and views in Rhapsody.</p>
Operational Viewpoint	Package		The Operational viewpoint contains views that provide a logical view of how an operation is carried out.
OV-1	Operational		OV-1 consists of three parts: OV-1a, OV-1b, and OV-1c. The OV-1 views are realized using UML use case diagrams.
OV-1a	Operational	High-Level Operational Concept Graphic	This high-level graphical presentation of the operational concept allows you to import pictures and other operational elements, such as Operational Nodes, Human Operational Nodes, Operational Activities, and the relations among them. It is intended to be an informal representation of the Concept of Operations (CONOPS).
OV-1b	Operational	Operational Concept Description	This view presents a textual description for OV-1a and it is produced with the associated OV-1a.
OV-1c	Operational	Operational Performance Attributes	<p>This view presents in tabular form the details for the operational performance attributes associated with the scenarios represented in OV-1a and their evolution over time.</p> <p>OV1-1a contains performance parameters that define quality of service requirements.</p>
OV-2	Operational	Operational Node Relationships Description	<p>This view shows the detailed relationships and flows among operational nodes and operational activities. It also may be used to express a capability boundary, that is, the problem domain.</p> <p>OV-2 is realized using UML class or structure diagrams.</p>

Architecture Viewpoint/View	View	View Name	Description
OV-3	Operational	Operational Information Exchange Matrix	This view, presented as a matrix, shows information exchanged between nodes, and the relevant attributes of that exchange. OV-3 can be derived from OV-2 and is generated using the table and matrix functionality.
OV-4	Operational	Organizational Relationships Chart	This view shows an organizational chart for the enterprise. It is divided into two types, a typical chart and an actual chart. OV-4 is realized for using UML class or structure diagrams.
OV-5	Operational	Operational Activity Model	This view shows the flow and ordering of activities required to achieve the operational capability. OV-5 is a lower-down version of OV-2 and is realized using UML activity diagrams.
OV-6a	Operational	Operational Rules Model	The OV-6 views are used to describe the textual rules that control and constrain the mission (for example, doctrine, rules of engagement, and so forth). They are represented as operational constraints placed upon operational view model elements. The actual rules and to which elements they are applied are shown in a UML structure or class diagram and then shown in a table.
OV-6b	Operational	Operational State Transition Description	The OV-6 views are used to describe the mission objective. OV-6b view depicts the behavior of an operational element (node or activity). OV-6b is realized using UML statecharts.
OV-6c	Operational	Operational Event-Trace Description	The OV-6 views are used to describe the mission objective. OV-6c shows the messages and ordering of messages passing between operational nodes. OV-6c is realized using UML sequence diagrams.

Architecture Viewpoint/View	View	View Name	Description
OV-7	Operational	Information Model	This view shows the structures of the data that are being passed between elements. OV-7 is realized using UML class diagrams that define the data, its composition and types.
System Viewpoint	Package		The System viewpoint contains views that look at how Operations are physically implemented. They are used as input to SRDs, detail how platforms interact, and are the physical realization of capabilities in StVs.
SV-1	Systems	Resource Interaction Specification	This view shows what your resources are and how they interact with each other. This includes the human elements of your enterprise, such as roles, posts, and organizations; as well as physical elements, such as systems and platforms. This view is generally used for the definition of systems concepts/options, interface definitions, interoperability analysis, and operational planning. SV-1 is realized from using UML structure or class diagrams.
SV-2a	Systems	Systems Port Specification	The SV2 views are all related to communication and can be realized using UML structure and class diagrams. SV-2a specifies the ports (specified points of interaction) that a system has and defines the protocols that a port may use.
SV-2b	Systems	Systems Port Connectivity Description	The SV2 views are all related to communication and can be realized using UML structure and class diagrams. SV-2b shows the interaction of ports between systems. SV-2b is very similar to SV-1 but with protocols and networks included.

Architecture Viewpoint/View	View	View Name	Description
SV-2c	Systems	Systems Connectivity Clusters	<p>The SV2 views are all related to communication and can be realized using UML structure and class diagrams.</p> <p>SV-2c defines how individual connections between systems are grouped into logical connections between parent resources.</p>
SV-3	Systems	Resource Interaction Matrix	<p>This view tells you what communicates with what. It is generated from the information from SV-1. SV-3 shows the lines of communication between systems as an N^2 diagram (system-system matrix). It indicates source (provider of information) and sink (consumer of information) of data flows.</p>
SV-4	Systems	Functionality Description	<p>This view defines the functional decomposition of a system or function. It can be used to show how functions interact to perform a higher-level function. SV-4 provides the functional requirements from the SRDs.</p> <p>SV-4 can be realized by using UML activity diagrams.</p>
SV-5	Systems	Function to Operational Activity Traceability Matrix	<p>This view is a spreadsheet-like generated view that summarizes the mapping of System and Systems functions to Operations, helps identify missing System functions, and provides traceability links between URDs and SRDs.</p> <p>This matrix is derived from UML class diagrams that have dependencies that link systems resources or functions to operations.</p>
SV-6	Systems	Systems Data Exchange Matrix	<p>Similar to SV-3 but for system data, this view details source-sink, protocols, content, and so forth, of all data items. It helps specify interoperability requirements.</p> <p>SV-6 is derived from information captured as attributes in the model. It is customizable depending upon the information required by the user of the architecture.</p>

Architecture Viewpoint/View	View	View Name	Description
SV-7	Systems	Resource Performance Parameters Matrix	This is a generated spreadsheet-like view that defines the quality of service requirements expected of each part of the system. SV-7 can be derived from a UML requirements diagram or attributes associated with model elements. This view is customizable by the user.
SV-8	Systems	Capability Configuration Management	This view is the system evolution description. It describes how the system or architecture is expected to evolve over long periods of time. SV-8 is similar to StV-3. SV-8 can be realized as an UML class or structure diagram.
SV-9	Systems	Technology and Skills Forecast	This view is a customizable table (it depends on user-defined time periods) that details what technology will be available in the near future. It touches upon system evolution, capability phasing, and acquisitions.
SV-10a	Systems	Resource Constraints Specification	This view specifies functional and non-functional constraints on the implementation aspects of the architecture (that is, the structural and behavioral elements of the SV viewpoint). These elements are mapped to each other on a class or structure diagram and shown in a set of table views.
SV-10b	Systems	Resource State Transition Description	This view shows state-based behavior, of the system resources to various events. For consistency, the state-based behavior should map to the aggregate behavior of all the flows shown in SV-10c. SV-10b is realized using UML statecharts.
SV-10C	Systems	Resource Event-Trace Description	This view provides a time-ordered representation of all the message and event interaction that occur between the various systems resources. These diagrams are focussed around specific scenarios. SV-10c is realized using UML sequence diagram.

Architecture Viewpoint/View	View	View Name	Description
SV-11	Systems	Physical Schema	This view is similar to OV-7 and it defines data used at the physical level (data relationships, structure, attributes), optimizes data structures, and specifies interfaces and data types. SV-11 is realized in UML class diagrams.
Acquisition Viewpoint	Package		It contains elements associated with the Acquisition viewpoint that are used by other views in the Rhapsody MODAF profile. Note that the AcV-1 and AcV-2 views are not supported in the Rhapsody MODAF profile.
Technical Viewpoint	Package		The Technical viewpoint contains views that detail technical standards that constrain the system development.
TV-1	Technical	Standards Profile	TV-1 presents the current technical and non-technical standards, guidance, and policy applicable to the architecture. Note that TV-1 can be a very large external document that is brought into the model or it can be created as a customizable table that maps elements representing standards to model elements.
TV-2	Technical	Standards Forecast	This view identifies the standards under development with expectations going forward. TV-2 can be seen in a customizable table that relates standards to time periods.

Configuring a Rhapsody Project for MODAF

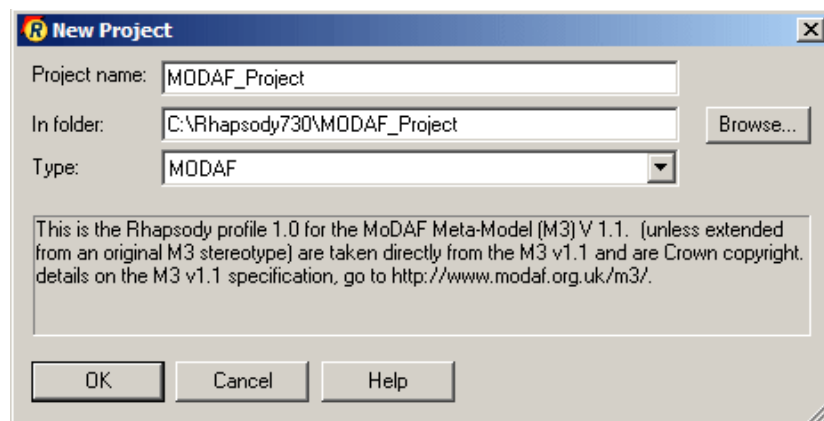
You specify the Rhapsody model elements that are to form the core views in the generated MODAF documentation. From these, other MODAF views are derived. The Rhapsody model elements included in the MODAF generated report are specified using stereotypes provided in the MODAF profile.

Creating a Rhapsody MODAF Project

To create a Rhapsody MODAF project, follows these steps:

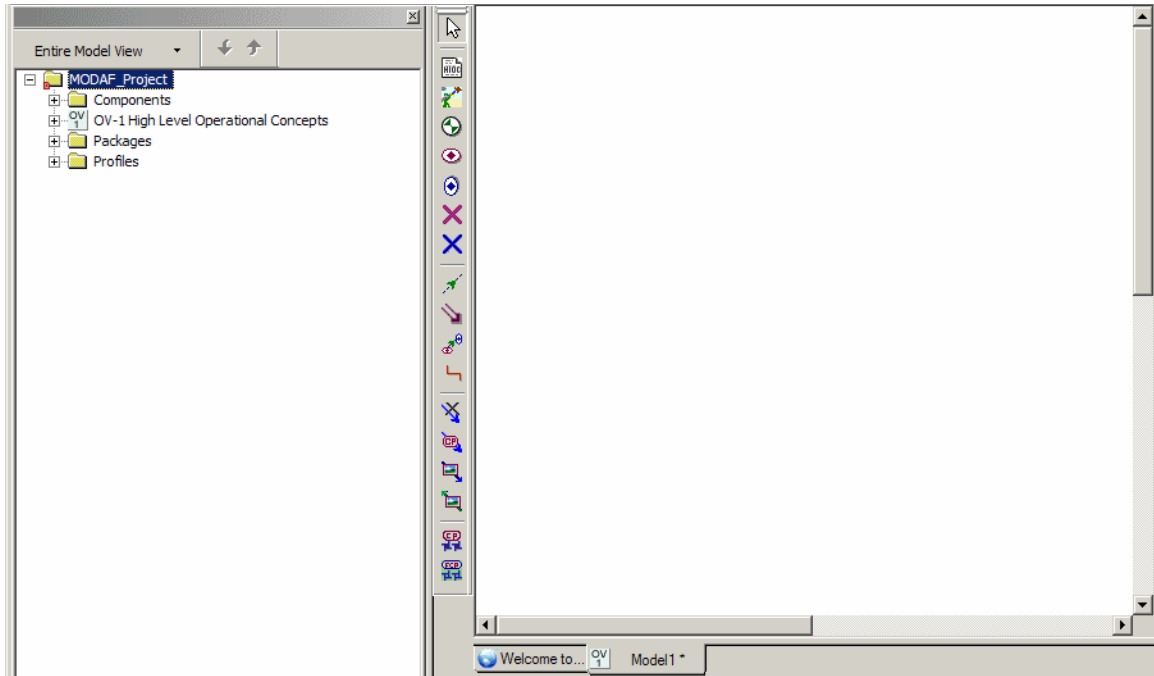
1. Launch Rhapsody and select **File > New** to open the New Project dialog box.
2. On the New Project dialog box:
 - ◆ Enter a project name (for example, `MODAF_Project`, as shown in the following figure).
 - ◆ Specify a folder location.
 - ◆ As the project type, select **MODAF** from the **Type** drop-down list.

Note: This **MODAF** type (or profile) is provided by the Rhapsody MODAF Add-on in order to help you customize and extend the Rhapsody product to support a Domain Specific Language (DSL), which lets you work with MODAF terms, diagrams, and artifacts rather than UML terms, diagrams, and artifacts.



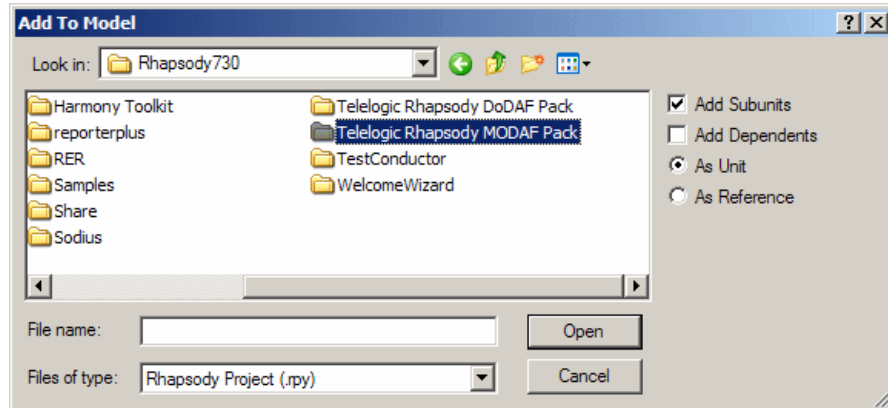
3. Click **OK**.

4. If the folder you specified does not exist, click **Yes** to create it. Rhapsody creates the project and opens with the initial view, as shown in the following figure:

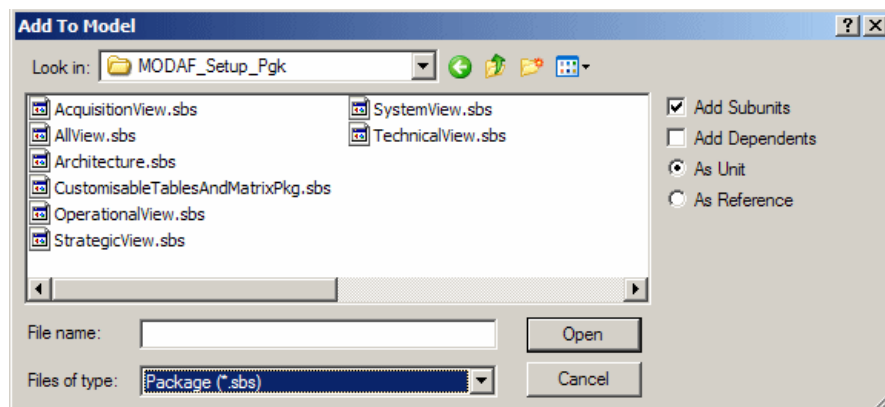


5. Add the primary Architecture Structure. Highlight the top-level project name (**MODAF_Project** in our example) and select **File > Add to Model** to open the Add to Model dialog box.

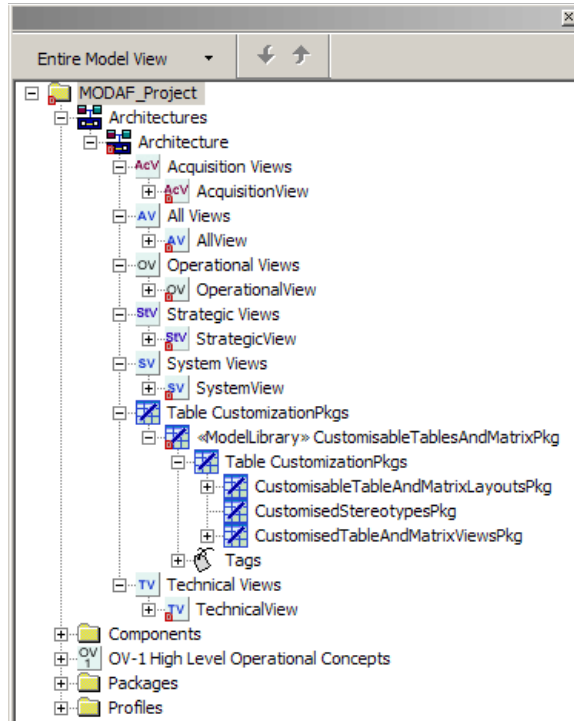
- Browse to the Rhapsody installation folder (for example, C:\Rhapsody730) and locate the **Rhapsody MODAF Pack**, as shown in the following figure:



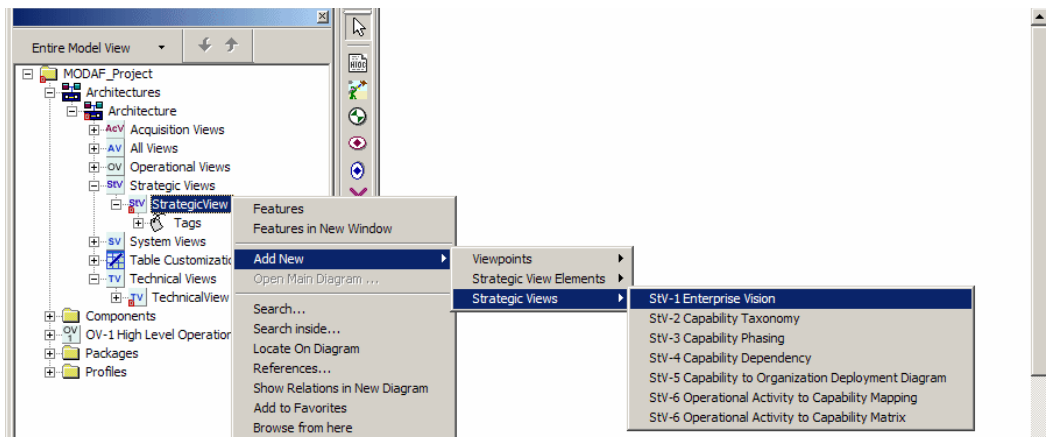
- Accept the defaults and double-click **Telelogic Rhapsody MODAF Add-on** and then locate **MODAF_Setup_Pkg**.
- Accept the defaults and double-click **MODAF_Setup_Pkg** and then change the files of type to **Package (*.sbs)**, as shown in the following figure:



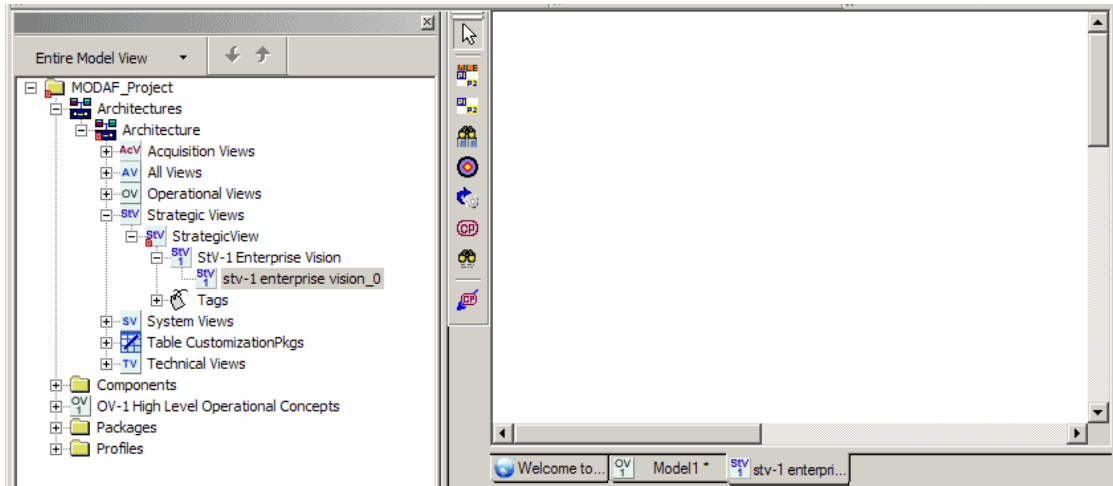
- Accept the defaults and double-click **Architecture.sbs**. This brings in the main viewpoints for your MODAF project. You now have the basic project structure. Your expanded Rhapsody browser should resemble something like the following figure:



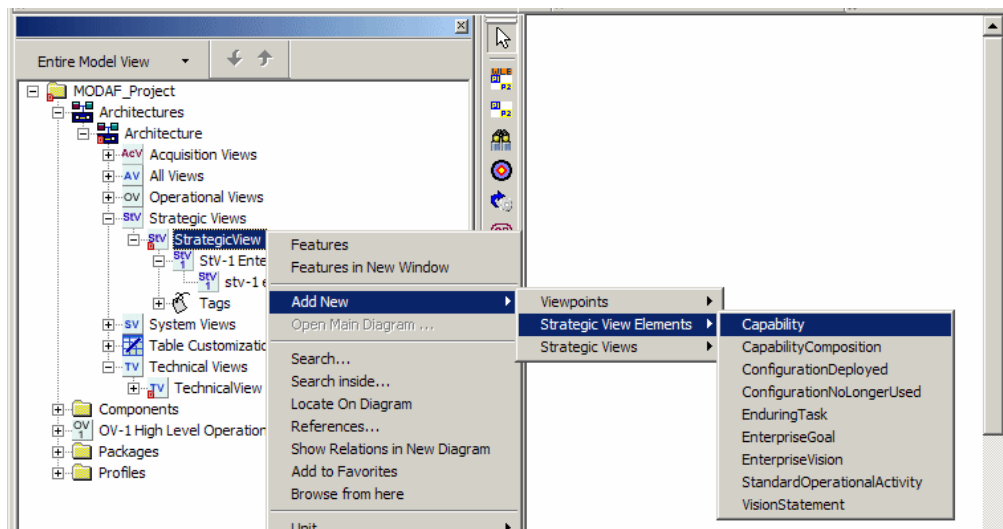
- To add views to a particular viewpoint, right-click the viewpoint and select **Add New > [Name] Views > [Name of View]**.



- Rhapsody opens the drawing area with the applicable **Drawing** toolbar for the view when appropriate, as shown in the following figure:



- To add other elements for a particular viewpoint, right-click the view and select **Add New > [Name] View Elements > [Name of Element]**, as shown in the following figure:

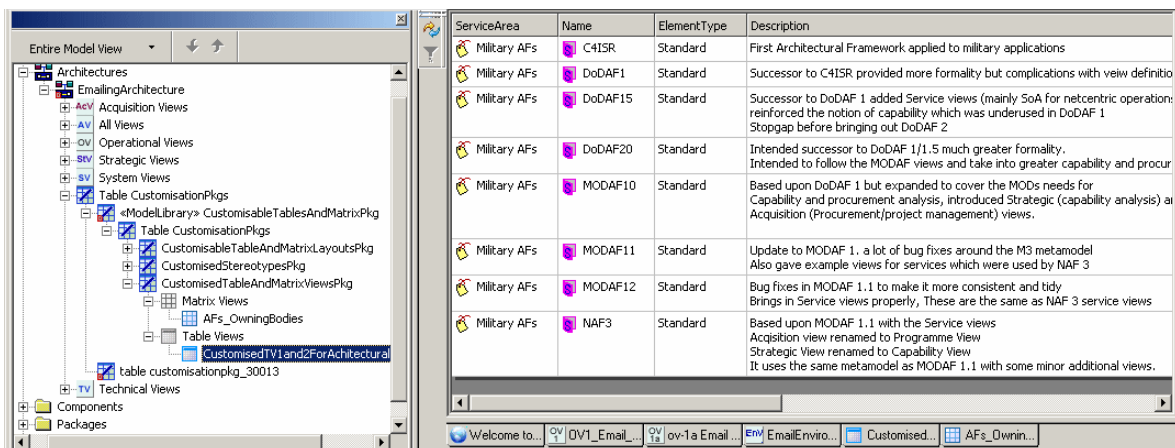


Customizing the Rhapsody Table and Matrix Views for MODAF

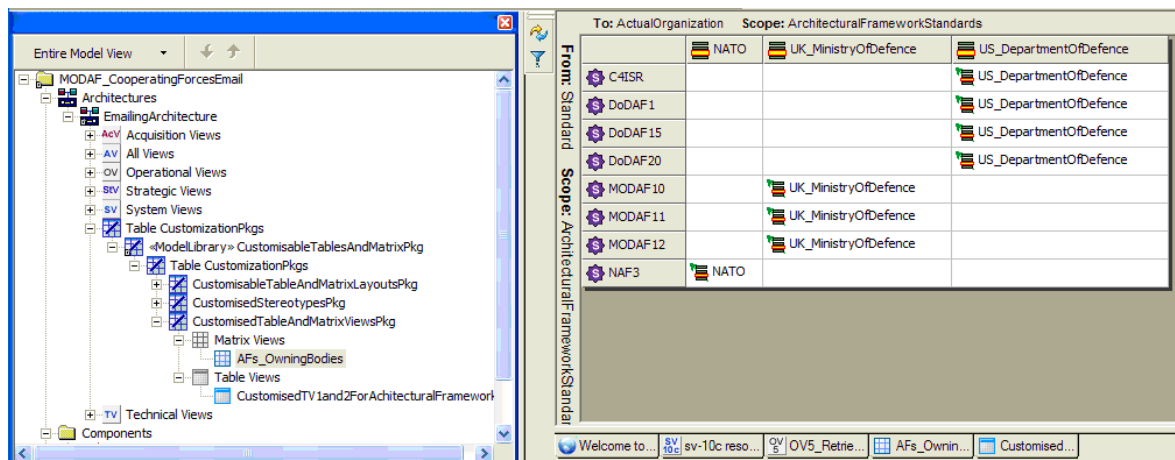
In Rhapsody, you can create your own tables and matrix views to represent your model data, providing you with another option to convey information among your team and stakeholders. For example, you can view your model requirements in a tabular view making it easy to see the details contained in all the requirements. This view is particularly useful for visualizing a large amount of data and their relationships to each other.

Rhapsody provides these additional methods to view model data:

- ◆ **Table view** performs a query on a selected element type and display a detailed list of its various attributes and relations, as shown in the following figure:



- ◆ **Matrix view** displays queries showing the relations among selected model elements, as shown in the following figure:



These views provide the following development capabilities:

- ◆ Define and run dynamic queries of model content
- ◆ Easy display and analysis of requirements
- ◆ Exportable and printable tables and data lists

For details on how to create the table and matrix views in Rhapsody, see [Creating Table and Matrix Views](#). To be able to supply the data for your table and matrix views, you have to create stereotypes and tags. See [Creating Stereotypes and Using Tags](#).

Creating Stereotypes and Using Tags

A profile hosts domain-specific tags and stereotypes. In addition, you can create your own stereotypes and tags for your Rhapsody project. For more information about stereotypes, see [Defining Stereotypes](#). For tags, see [Using Tags to Add Element Information](#).

To be able to create your own custom table and matrix views, you can use the stereotypes and tags provided by the MODAF profile. However, to be able to truly produce table and matrix views of your own design, you will probably find it most useful to create at least one stereotype of your own and then create the tags you want for it. Once set up, you can associate any of your tags with views and operations to produce tables and matrices that are customized for your project needs.

Note

See [Creating Table and Matrix Views](#) before you try to create your own custom table and matrix views. You should also see [About Creating Table/Matrix Views in MODAF](#).

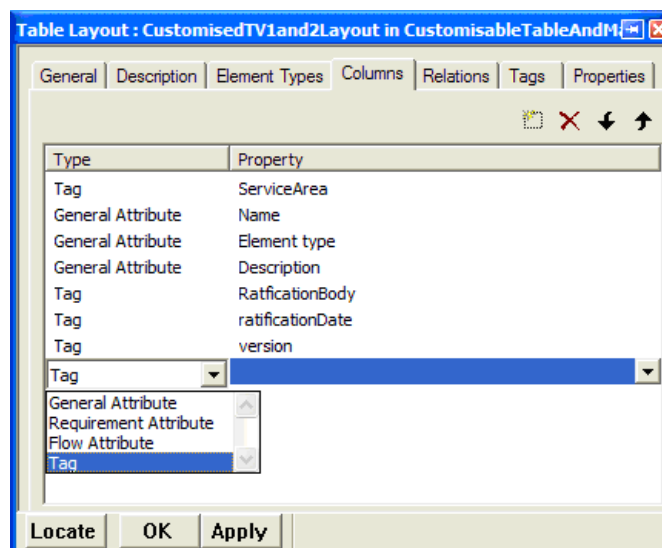
About Creating Table/Matrix Views in MODAF

For the Rhapsody MODAF Pack, the ability to create custom table and matrix views is facilitated by the Table CustomizationPkg, which consists of three subpackages:

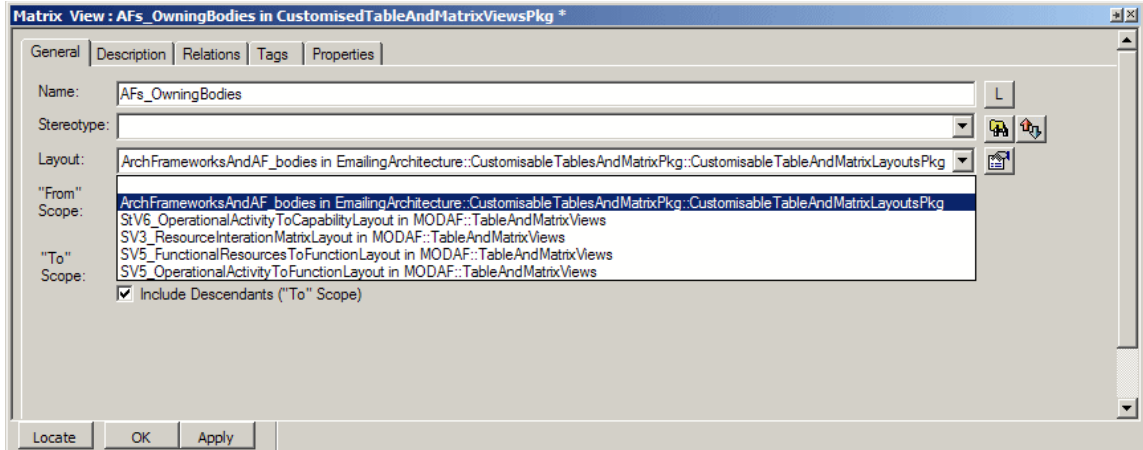
- ◆ [CustomizableTableAndMatrixLayoutsPkg](#)
- ◆ [CustomizedStereotypesPkg](#)
- ◆ [CustomizableTableAndMatrixViewsPkg](#)

CustomizableTableAndMatrixLayoutsPkg

The CustomizableTableAndMatrixLayoutsPkg package contains the table layouts for the MODAF-specific table views that need to be customized by the user. The customization normally consists of setting up the display of the tag elements to be extracted by the user, as shown in the following figure:

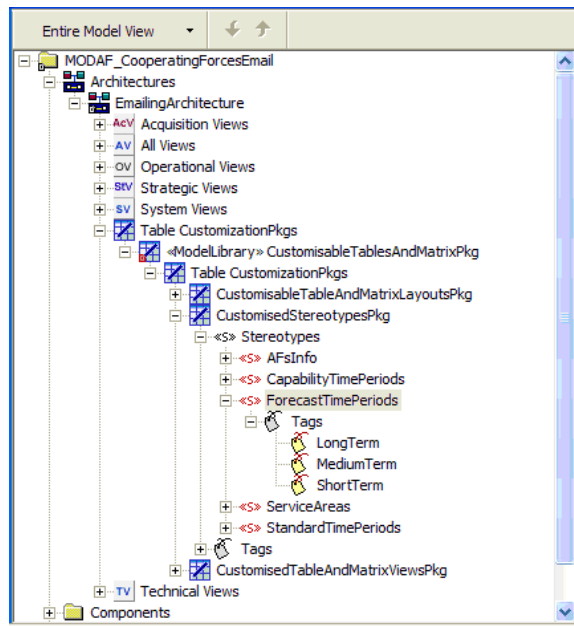


Customized matrix layouts can be added and used by the specific MODAF views by changing the layout and scope to be used by the view, as shown in the following figure:

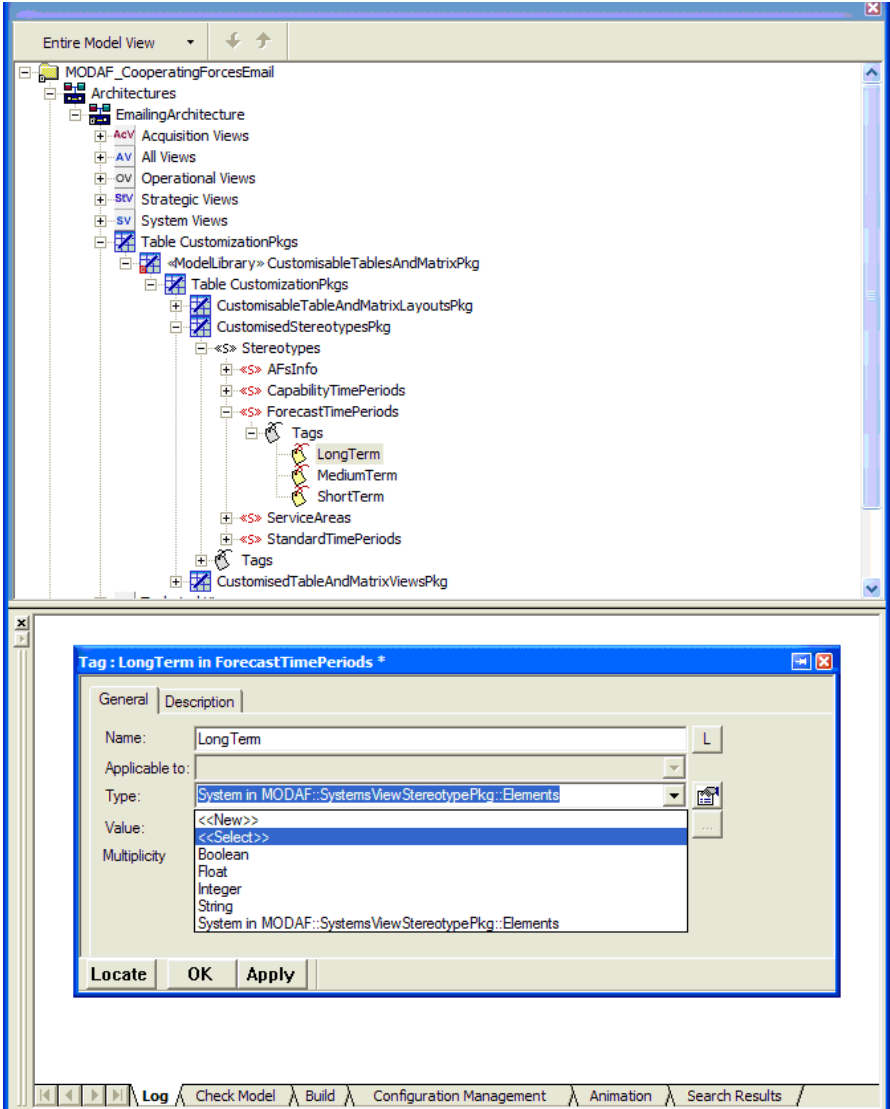


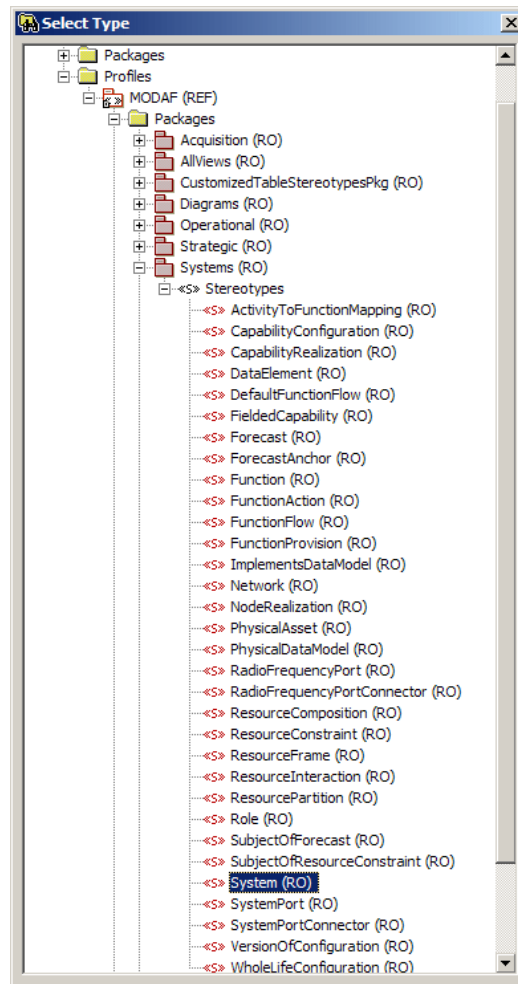
CustomizedStereotypesPkg

The CustomizedStereotypesPkg package provides a location to store the custom stereotypes to be used for custom table layout. Stereotypes can be created with user customizable tags and then applied to specific type of element (typically a class or object).



The tags are typically typed by a specific element (selected from the MODAF profile stereotypes list) type. This ensures that when the tag is populated in the model element that the correct type of information is displayed in the actual table (if defined in the table layout).



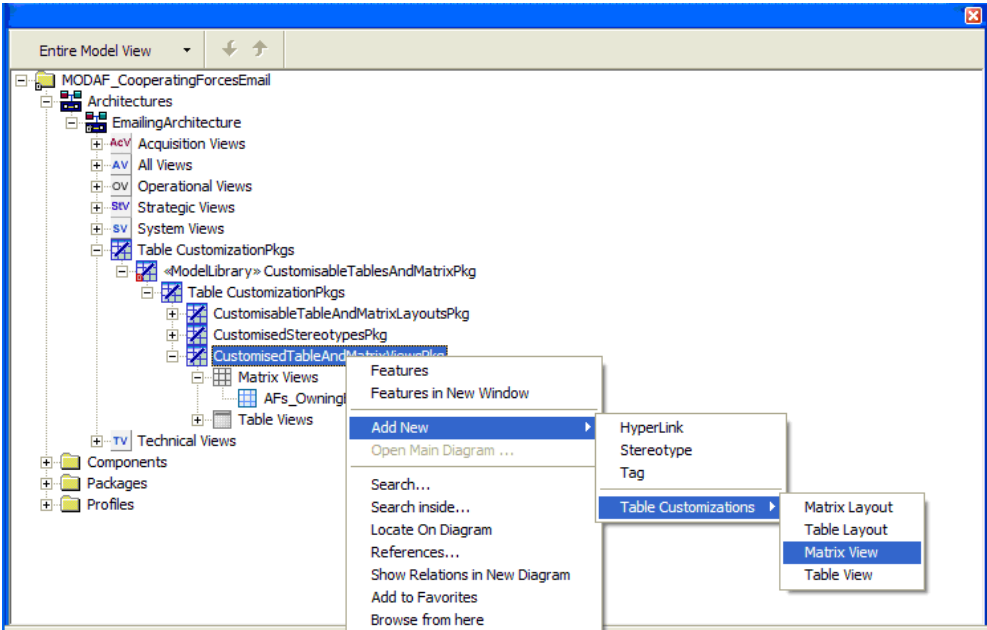


When an element is given a particular stereotype all the tags associated with the stereotype are inherited by the element. Then you have to populate the tag values with the correct information.

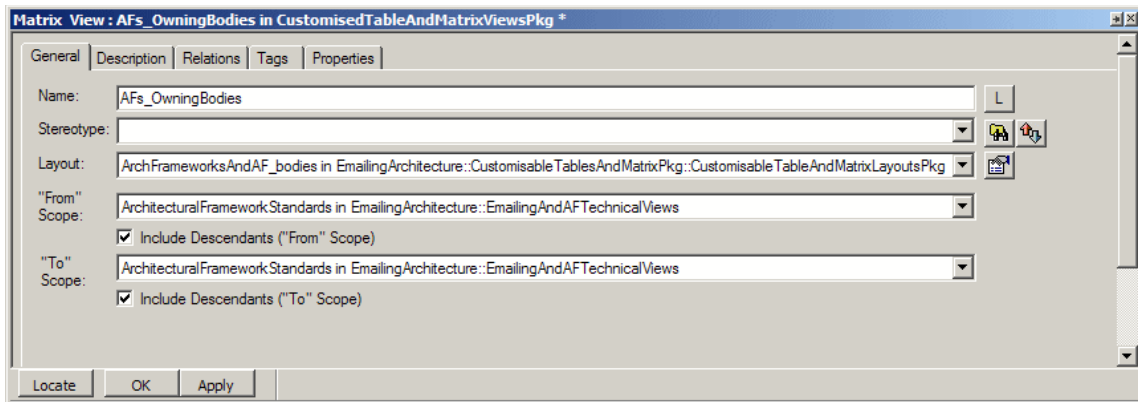
CustomisableTableAndMatrixViewsPkg

The customized MODAF-specific views are still created in their correct places in the Architecture. The CustomisableTableAndMatrixViewsPkg package is intended for custom views outside the scope of the MODAF-specific table and matrix views.

You create a customizable table or matrix by selecting the appropriate package, for example, **Add New > Table Customizations > Matrix View** (as shown in the following figure) or **Add New > Table Customizations > Table View**.



Rhapsody creates a new view. Then you have to open the Features dialog box for the view and select the package scope of the view and the appropriate layout to use (from the CustomizedTableAndLayoutsPkg package), as shown in the following figure:



When the table or matrix view is selected in the browser, it will be brought up in the Graphic Editor area (typically to the right of the browser).

Note: If the view appears empty, incomplete, or has too much information, you may want to review how the scope was set.

For more details about creating the Table and Matrix views in Rhapsody, see [Creating Table and Matrix Views](#).

Note

For the MODAF profile, to create a layout or view for a table or matrix, the menu command path, for example, is **Add New > Table Customization > Table Layout** (instead of **Add New > Table Layout**). Otherwise, the procedures in [Creating Table and Matrix Views](#) are the same.

Creating Documentation for Your MODAF Project With ReporterPLUS

The Rhapsody MODAF Add-on provides you with a ReporterPLUS template that gives you the ability to generate complete electronic documentation with hyperlinks between related and referenced model elements. Output, for example, can be in HTML and Microsoft Word. For information about ReporterPLUS, see the *Rhapsody ReporterPLUS Guide*.

This section describes how to set up ReporterPLUS to use this template. It also discusses the structure of the generated document, how to generate the document, and provides some tips on how to use the model structure to its best advantage when generating a document.

Setting Up ReporterPLUS

As mentioned earlier, the ReporterPLUS template supplied with the Rhapsody MODAF Add-on can produce hyperlinks between referenced elements. To enable this feature of the template, you must modify the `Rhapsody.ini` file located in the Rhapsody installation folder and then move it to the System (or Windows directory). Follow these steps:

1. Make a copy of the `Rhapsody.ini` file that is located in the Rhapsody installation folder (for example, `<Rhapsody installation path>\Rhapsody730`).

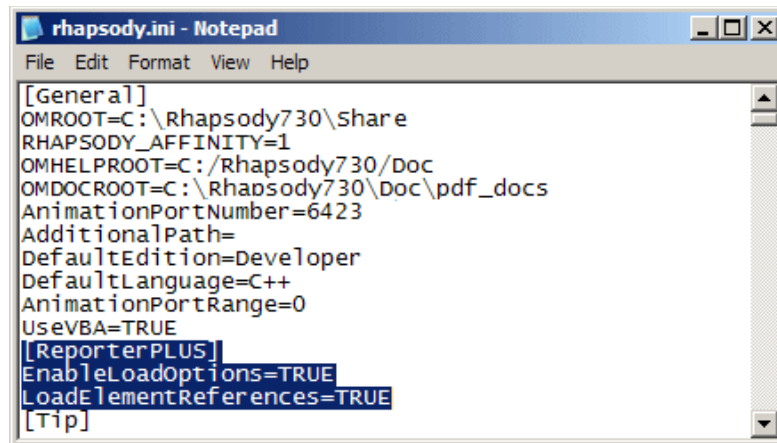
You should now have a `Rhapsody.ini` file and a `Copy of rhapsody.ini` file.

2. Rename `Copy of rhapsody.ini` file to `OriginalRhapsody.ini`.

Having this file ensures that you can return to the original state of the `rhapsody.ini` file when needed.

3. Add the following lines within the [General] section of the Rhapsody.ini file, as shown in the following figure:

```
[ReporterPLUS]
EnableLoadOptions=TRUE
LoadElementReferences=TRUE
```



4. After you save your changes to the Rhapsody.ini file, move it to your System (or Windows) directory).

Note: If you use Copy and Paste, be sure to delete the file from where you copied it from.

5. Restart Rhapsody to ensure the changes take effect.

Document Structure

The document produced by ReporterPLUS consists of these main sections:

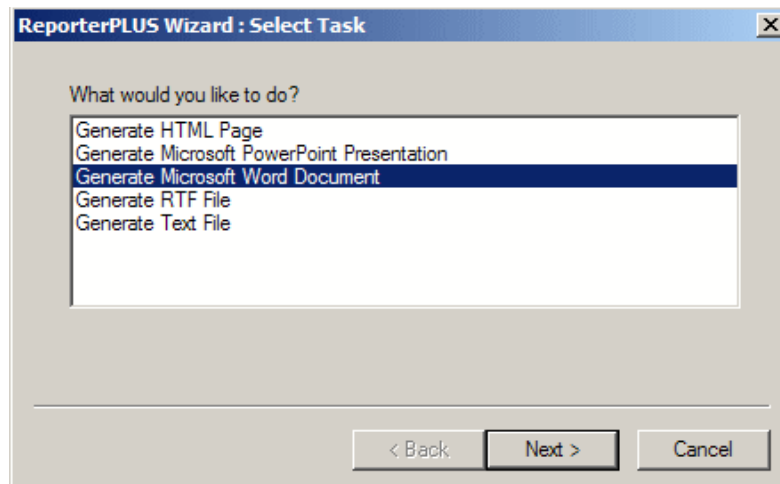
- ◆ A table of contents where all the views and elements are listed and hyperlinked to their location in the documentation.
- ◆ A graphical section that contains all the views, including the embedding of external documents, elements that exist on those views and their descriptions, if they have any. All the elements are hyperlinked to their definition in the Data Dictionary section.
- ◆ A Data Dictionary section that contains the details of all the elements in the model in the viewpoint in which they were created. The model elements from the first section are all hyperlinked to their counterpart in the Data Dictionary as are any connected elements, descriptions and tags for all elements are shown.

Generating a MODAF document

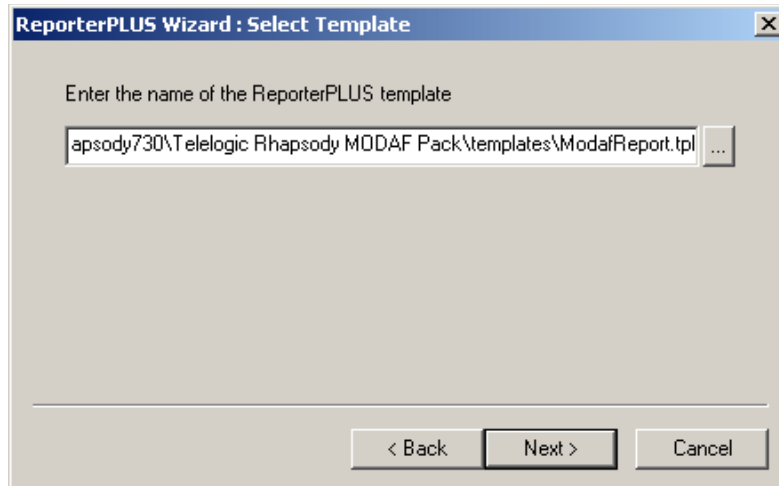
The Rhapsody MODAF Add-on provides you with a ReporterPLUS template called `ModafReport.tpl`. The template is located within the Rhapsody installation path (for example, `<Rhapsody installation path>\Telelogic Rhapsody MODAF Pack\templates`). You can use ReporterPLUS to create, for example, HTML and Microsoft Word documents from any Rhapsody model.

To use the `ModafReport.tpl` template, follow these steps:

1. Be sure that you have set up ReporterPLUS to use the `ModafReport.tpl` template; see [Setting Up ReporterPLUS](#).
2. With your MODAF project open in Rhapsody, select **Tools > ReporterPLUS > Report on all model elements**.
3. On the Select Task dialog box (as shown in the following figure), select the type of output you want, and click **Next**.



4. On the Select Template dialog box, browse to the MODAF ReporterPLUS template location and select it, as shown in the following figure, and click **Next**.



5. On the Confirmation dialog box, click the **Finish** button.
6. On the Generate Document dialog box:
 - ◆ Enter a document name.
 - ◆ Browse to where you want to locate the files that will be produced.
 - ◆ Click the **Generate** button to generate your document.
7. Wait while your document is generated. ReporterPLUS spends some time loading the template and the model. Then it analyzes the model and the model element relationships.
8. When available, click **Yes** to open your report.

Troubleshooting ReporterPLUS and MODAF

If the contents of a package do not appear in the document created by ReporterPLUS, it is possible that you may have created an architecture without adding the architecture.sbs package as described in [Configuring a Rhapsody Project for MODAF](#).

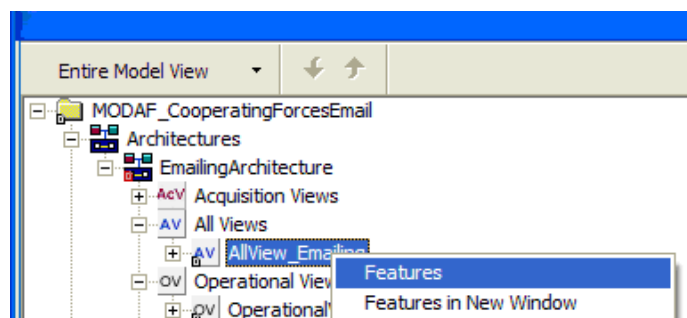
The MODAF profile has two tags that have been added to all the viewpoints, `ContainsViews` and `RootView`. By default, they are set to `Cleared` (meaning False, their check boxes are not selected). However, within the imported architecture.sbs package, these tags have been preset to `Checked` (meaning True, their check boxes are selected).

`ContainsViews` is set to `Checked` when you have established a hierarchy of views within a particular viewpoint that specifically contains views. By default, this is set to `Cleared` because the main reason why subviews are used is to provide containers for sets of model elements so that a table or matrix view can be scoped correctly.

`RootView` indicates that this is the Root Viewpoint for this particular set of views, it should only be set to `Checked` for the viewpoints that sit directly above the Architecture layer.

To override the default settings for a particular viewpoint, follow these steps:

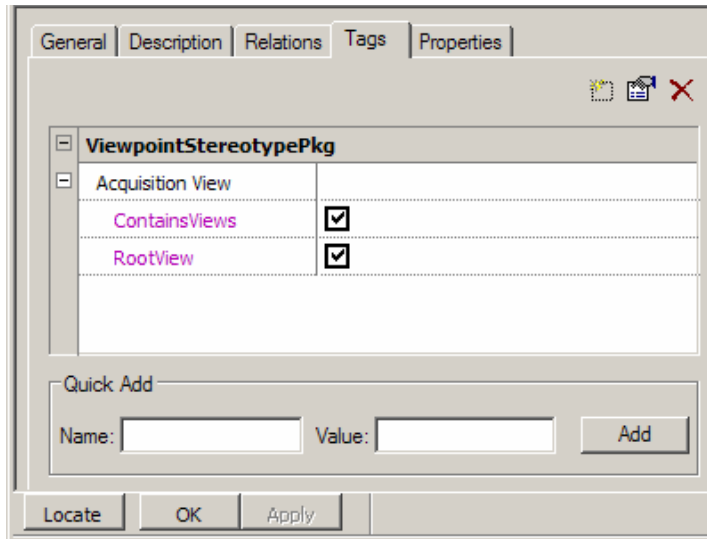
1. Right-click the viewpoint and select **Features**, as shown in the following figure:



2. Select the **Tags** tab on the Features dialog box.
3. Select the **ContainsView** check box (so that it is selected).

4. Select the **RootView** check box if the viewpoint is likely to contain views that you want displayed in the documentation.

The following figure illustrates that both check boxes are selected.



Note: If a viewpoint just contains elements and if those elements are used by other views external to the owning viewpoint, they will still appear in the document generated by ReporterPLUS in the correct places.

Using the Dependencies Linker

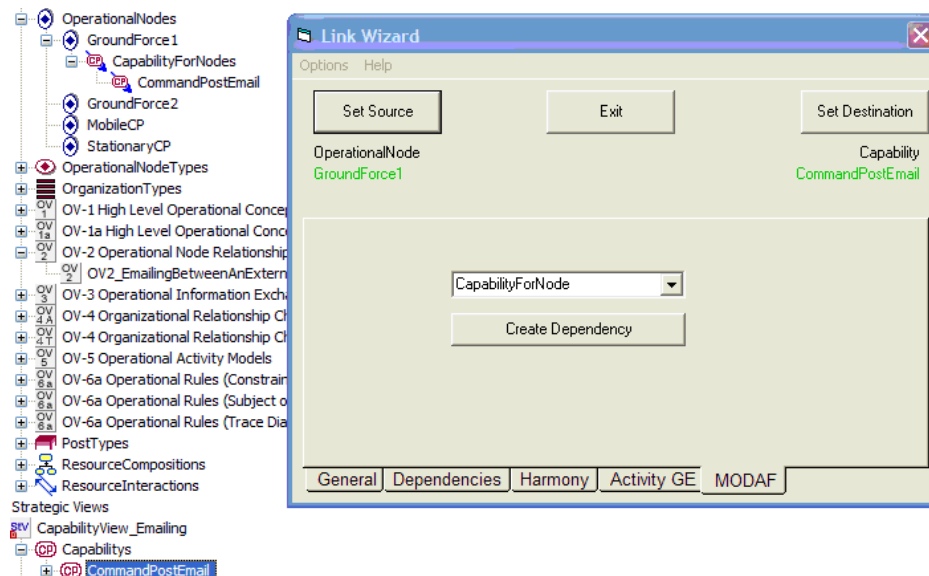
The Dependencies Linker is an extension to the Link wizard documented in the Systems Engineering Toolkit (for more information, see [Harmony Process and Toolkit](#)). The Dependencies Linker allows dependencies to be created between particular model elements. It is located under the **MODAF** tab of the Link Wizard dialog box.

The Dependencies Linker is a tool for advanced users who know which dependency relationships are allowed between which elements and can use the browser effectively.

The `modaf_stereotypes.txt` file contains a list of the possible dependencies. The file is located in the `<Rhapsody installation path>\Share\Profiles\MODAF` path. The profile controls the permissions as to what dependencies can be created by what elements.

To use the Dependencies Linker, follow these steps:

1. Choose **Tools > Dependencies Linker** to open the Link Wizard dialog box.
2. On the Rhapsody browser, select the source element and then click **Set Source** on the Link Wizard dialog box.
3. On the MODAF tab of the Link Wizard, select the dependency from the drop-down list.
4. On the Rhapsody browser, select the destination element and then click **Set Destination** on the Link Wizard dialog box.
5. Click the **Create Dependency** button on the MODAF tab of the Link Wizard, as shown in the following figure:



Note: If the **Create Dependency** button is disabled, this means that you have selected an illegal dependency between the two elements.

Troubleshooting the Dependencies Linker

If the Dependencies Linker tool does not appear, then it is likely that the `MODAF.hep` file is not located in the same folder as the `MODAF.sbs` file. Make sure the `MODAF.hep` file is in the same folder as the `MODAF.sbs` file.

General Troubleshooting

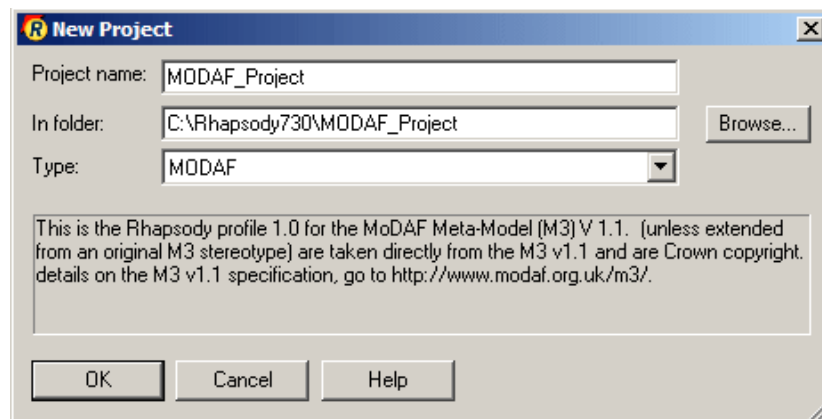
This section provides you with information that you can use for general troubleshooting purposes for Rhapsody MODAF.

Verifying the Rhapsody MODAF Add-on Installation

If you are having a problem with the Rhapsody MODAF Add-on or if it does not appear in the Rhapsody product as mentioned in this section, you should verify that it has been installed.

Use any of the following methods to verify that the Rhapsody MODAF Add-on has been installed:

- ◆ See if there is a path of the Add-on from the Windows Start menu. Choose **Programs > Telelogic > Telelogic Rhapsody Version # > Telelogic Rhapsody MODAF Pack**.
- ◆ See if the Add-on has been installed with the Rhapsody product. Typically that path would be <Rhapsody installation path>\Telelogic Rhapsody MODAF Pack.
- ◆ Create a project in Rhapsody and see if the **MODAF** type (profile) is available, as shown in the following figure:



If you find that the Rhapsody MODAF Add-on is not installed on your system, then you must install it. You need a license key for the Rhapsody MODAF Add-on if it is not already a part of your Rhapsody license. Refer to the *Rhapsody Installation Guide* for more information about installing the Rhapsody MODAF Pack.

If the Rhapsody MODAF Add-on is installed on your system but you are having problems with it, you may have to un-install and then re-install the Add-on to repair your installation if it has been damaged.

Finding Missing Icons on Drawing toolbar

If you notice that there seems to be missing icons on the **Drawing** toolbars, make sure the \$OMROOT is correct for the Rhapsody MODAF Pack.

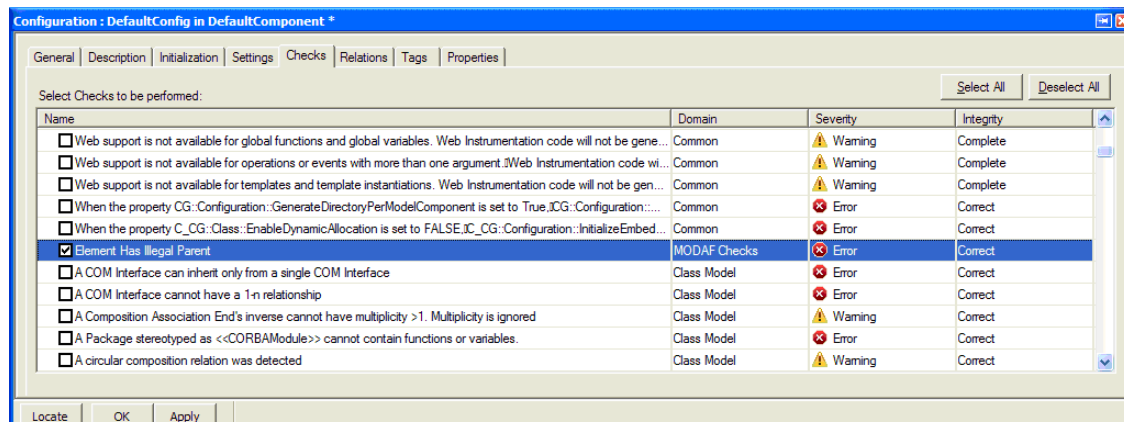
Checking Your MODAF Model

The Rhapsody MODAF Add-on contains a helper (a Java plug-in) that checks the architectural conformance of your model. It indicates where an element is contained within an illegal parent.

Setting Up the Model Checker for a MODAF Project

To set up the model checker so that it checks your MODAF model, follow these steps:

1. Choose **Tools > Check Model > Configure** to open the Configuration dialog box for the Check Model tool.
2. Click the **Deselect All** button to clear the check boxes for all the checks (tests).
3. Select the **Element Has Illegal Parent** test (which is in the MODAF Checks domain), as shown in the following figure:

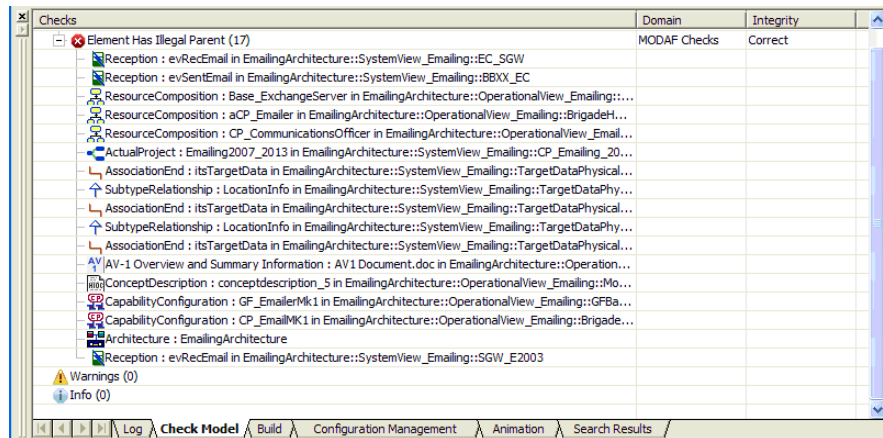


4. Click **OK**.

Running the MODAF Model Checker

To run the MODAF model checker, follow these steps.

1. Choose **Tools > Check Model > DefaultConfig** (the name of the default component).
2. The results appear on the Check Model tab of the Rhapsody Output window, as shown in the following figure:



Note the following about the Check Model tool for a MODAF model:

- ◆ If more than one error is found, the Architecture will always be seen by default. Other results that can be ignored are if Receptions are used on Interfaces and AssociationEnds, and Subtype relationships for data elements.
- ◆ When possible, you can double-click the element highlighted in an error to locate it in the browser to help you understand its parent hierarchy with reference to the M3.
- ◆ You may want to examine the `Model::Sterereotype::Aggregates` property of the stereotype definition in the profile. This contains the elements that should be allowed to be contained by the parent element causing the error.
- ◆ You can examine the structure of an element. Right-click the error element or its parent in the browser and select **Show Relations in New Diagram** to create an object model diagram that is automatically populated with related model elements. For more information on **Show Relations in New Diagram**, see [Showing All Relations for a Class, Object, or Package in a Diagram](#).
- ◆ The Check Model tool only works with JRE 1.5.

Rhapsody's Automotive Industry Tools

Rhapsody's tools for the automotive industry provide for specification and design of automotive systems and software applications. Rhapsody provides the AUTOSAR-specific [Model-driven Development \(MDD\)](#) environment to leverage both UML and SysML. This makes it possible for engineers to reuse specifications for common vehicle features across multiple automotive lines. The AUTOSAR profiles are used for architectural description of an AUTOSAR model using the native AUTOSAR concepts.

The AutomotiveC profile enables code generation for static systems with limited resources.

Rhapsody extends the benefits of MDD to the C developer by allowing designers to work in either a functional or object-oriented environment. Rhapsody includes blocks, flows, graphical files, functions, and data so that C developers can create models using familiar concepts.

AUTOSAR Modeling

Rhapsody provides two AUTOSAR profiles (AUTOSAR_20 and AUTOSAR_21) that can be used for modeling components in accordance with the AUTOSAR development process. AUTOSAR_20 conforms with the AUTOSAR 2.0 standard, and AUTOSAR_21 conforms with the AUTOSAR 2.1 standard.

Note

The AUTOSAR profiles are only visible in the list of profiles if you selected the Automotive option during your installation of Rhapsody.

Information on the AUTOSAR architecture can be found at the AUTOSAR Web site, www.autosar.org

The AUTOSAR Workflow

The workflow for AUTOSAR modeling is as follows:

1. Create an AUTOSAR Rhapsody project.
2. Create diagrams using the AUTOSAR elements provided by Rhapsody.

3. Use Rhapsody's Check Model tool to verify that there are no problems with your AUTOSAR model.
4. Export your model to the AUTOSAR XML format.

Creating an AUTOSAR Project

To create an AUTOSAR project in Rhapsody:

1. Select **File > New** from the main menu.
2. In the New Project dialog:
 - a. Provide a project name.
 - b. Indicate the folder where the project should be saved.
 - c. From the **Type** drop-down list, select **AUTOSAR_20** or **AUTOSAR_21**.
 - d. Click **OK**.

Creating AUTOSAR Diagrams

Rhapsody's AUTOSAR profiles allow you to compose the following types of diagrams:

- ◆ ECU diagram
- ◆ Implementation diagram
- ◆ Internal Behavior diagram
- ◆ SW Component diagram
- ◆ System diagram
- ◆ Topology diagram

Checking an AUTOSAR Model

To verify that there are no problems with your model, select **Tools > Check Model > [configuration name]** from the main menu.

Import/Export from/to AUTOSAR XML Format

Projects based on the AUTOSAR profiles can be exported from Rhapsody to the AUTOSAR XML format.

To export your project, follow these steps:

1. Select **Tools > Generate AUTOSAR XML Document**.
2. Click **Browse** on the When the Export dialog box to select the destination XML file.
3. Click **Export**.

Rhapsody can also import data stored in the AUTOSAR XML format.

To import AUTOSAR data, follow these steps:

1. Select **Tools > Import AUTOSAR XML Document**
2. Click **Browse** on the When the Import dialog box opens to select the source XML file.
3. Click **Import**.

The AutomotiveC Profile

The AutomotiveC profile provides the capabilities that you are likely to require for projects in the automotive industry and developed using the C language only.

The AutomotiveC profile contains the following components:

- ◆ Extended execution model
- ◆ Automotive-specific stereotypes
- ◆ A set of properties
- ◆ Support for fixed-point variables
- ◆ Simulink and StateMate block integration capabilities

Extended Execution Model

The extended execution model is implemented via:

- ◆ An OXF, called ExtendedC_OXF.
- ◆ Profile-specific code generation behavior
- ◆ Changes to Rhapsody's standard Features dialog box that allows you to provide additional information for classes and objects.

ExtendedC_OXF

ExtendedC_OXF is a derivative of the standard Rhapsody OXF. Unused functionality, such as cleanups and malloc'ed data, has been flagged out using compilation flags. The framework is also MISRA98-compliant.

All data and containers are statically defined.

The ExtendedC_OXF framework includes two adaptors, OSEK21 and Mainloop.

You select the adaptor to use by applying one of the configuration-level stereotypes (see [Configuration Stereotypes](#)).

The OSEK21 Adaptor

The OSEK21 adaptor provides a Rhapsody development environment for users of the Metrowerks OSEK 21 operating system and tools.

The OSEK adaptor is a set of Rhapsody components which allow you to easily generate code and build applications for either of the following targets:

- ◆ Metrowerks OSEK21 OS/NT Build 2.1.10 (Build env: MSdev)
- ◆ Metrowerks OSEK21 OS/12 Build 2.1.13 (Build env: Metrowerks)

All of the required files are copied to your disk when you choose the *Automotive* add-on option during installation.

Target Hardware and Software

The OSEK21 adaptor is used to create applications for the following target hardware and software:

Hardware:

- ◆ OSEK21NT target: PC
- ◆ OSEK21HC12 target: Motorola HC12dg128 evaluation board.

Software:

- ◆ OSEK21NT target: Windows XP, Metrowerks OSEK 21 OS for NT, Visual Studio (nmake utility, compiler, linker etc.)
- ◆ OSEK21HC12 target: Windows XP, Metrowerks OSEK 21 OS for HC12, Hiware tools for the Motorola HC12 target (compiler, linker, etc.), Visual Studio (nmake utility)

OSEK21 Tasks

The predefined task OS_TASK is responsible for initialization of the model. Using the relevant properties provided, the following characteristics have been assigned to the task: highest priority, non-preemptive, autostart, basic. You can change these characteristics as required.

The predefined task TIMER_TASK is responsible for timeout-handling and dispatching of timed actions. Using the relevant properties provided, the following characteristics have been assigned to the task: second-highest priority, non-preemptive, autostart, basic. You can change these characteristics as required.

Using the OSEK21 Adaptor - Workflow

To build an application using the OSEK21 Adaptor, follow these steps:

1. Create a new project of type AutomotiveC.
2. Add OSEK tasks to your model: **Add New > AutomotiveC > OSEK21BasicTask** (or **OSEK21ExtendedTask**).
3. For each task, change the **Concurrency** box to **Active**.
4. For each task, use the **Tags** tab of the Features dialog box to set the necessary values for the OIL definition.
5. For each task, add the required attributes and operations.

6. Create a new `OSEK21HC12Configuration` configuration (**Add New > OSEK21 > OSEK21HC12Configuration**).
7. Set the new configuration to be the active configuration
8. In the configuration, set the property `C_CG:OSEK21HC12:OSEKDIR` to the path to OSEK 21 for HC12.
9. In the configuration, set the property `C_CG:OSEK21HC12: HICROSSDIR` to the path to the Hiware tools.
10. If necessary, modify the value of the following properties:
 - `C_CG: :OSEK21HC12::OsekMainFileDefinition` - determines the content of the C source file that contains the main entry of the OSEK application and the definition of the predefined tasks from the framework (`OS_TASK` and `TIMER_TASK`).
 - `C_CG:: OSEK21HC12::OilDefinitionTemplate` - the content of the OIL file `cfg.oil` that contains include statements to include the model's specific OIL definition.
11. Generate and compile the application.

The workflow described above refers to building an application for the OSEK21 OS/12 target.

To build an application for the OSEK21 OS/NT target, use `OSEK21NTConfiguration` when creating a new configuration, and in the configuration, set the property `C_CG::OSEK21NT::OSEKDIR` to the path to OSEK 21 for NT.

Mainloop Adaptor

The Mainloop adaptor is intended for designing applications that will run on hardware without an operating system. The targets are as follows:

MainLoopNT configuration: Target is PC (+ Microsoft)

MainLoopS12 configuration: Target is freescale HCS12X evaluation board

AutomotiveC Code Generation

The profile-specific code generation mechanism is used when the value of the property `General::Model::ExecutionModel` is set to `Extended`. When you create a project based on the AutomotiveC profile, this is the default value for the property.

UI Changes

When the value of the property `General::Model::ExecutionModel` is set to `Extended`, the features dialog for various types of elements contains additional fields that allow you to provide the additional information required for classes and objects when using the extended execution model.

Automotive-specific Stereotypes

The stereotypes are available for the C language and in the Automotive C profile only.

Configuration Stereotypes

The AutomotiveC profile contains the following stereotypes that can be applied to configurations:

- ◆ OSEK21NTConfiguration
- ◆ OSEK21HC12Configuration
- ◆ MainLoopNTConfiguration
- ◆ MainLoopS12Configuration

These stereotypes are “new terms” that are applicable to configurations. They correspond to the following environments:

- ◆ OSEK21NT
- ◆ OSEK21HC12
- ◆ MainLoopNT
- ◆ MainLoopS12.

These stereotypes set the value of the Environment property. This value specifies the set of Rhapsody properties to use for that environment.

You can create a new configuration using either of the following methods:

- ◆ In the browser, from the context menu for components, select **Add New > AutomotiveC > [configuration]**.
- ◆ From the context menu for components, select **Add New > Configuration**. From the context menu for the newly created configuration, select **Change To > [configuration]**.

Note

For AutomotiveC projects, you must use one of the relevant configuration stereotypes. When you create a new project, the browser will contain a configuration called `DefaultConfig`. This is a generic configuration so you must apply one of the automotive configuration stereotypes to it.

Network Port Stereotypes

The AutomotiveC profile contains two “new terms” whose purpose is to allow you to bind data elements to signals on a bus:

- ◆ `inNetworkPort` - connects to an input signal from a bus
- ◆ `outNetworkPort` - connects to an output signal on a bus

Both of these types of network ports can be created in the browser and then be dragged to an object model diagram. (If you have applied the *ArchitectureDiagram* stereotype to the diagram, you can also create network ports by selecting the appropriate icon on the drawing toolbar.)

The following constraints apply to the creation of network ports:

- ◆ Network ports must be connected to a flow port on a Part, using a link.
- ◆ The network port must be of the same type as the flow port to which it is connected.

Adding a Network Port

To add a network port:

1. If you have applied the *ArchitectureDiagram* stereotype to the diagram, select the appropriate icon on the drawing toolbar and drag it to the diagram.

If you are using an object model diagram without the *ArchitectureDiagram* stereotype:

- a. In the browser, select the class that is the parent of the instance to which you will be connecting the port.
 - b. Open the browser context menu, and select **Add New > AutomotiveC > inNetworkPort** (or **outNetworkPort**).
 - c. Drag the network port to the relevant diagram.
2. Connect the network port to the flow port on the Part.
 3. Open the features dialog for the network port, and provide values for the various fields.

Features Dialog for Network Ports

The Features dialog box for network ports contains the following fields:

- ◆ **Name** - the name of the network port
- ◆ **Stereotype** - currently not supported
- ◆ **Type** - the type of the signal on the bus, for example, int.

Note: In terms of type, the two types of network ports support unidirectional atomic types. This means that you cannot use non-atomic types, bi-directional ports, or multiplicity other than 1.

For inNetworkPorts, the Features dialog box also contains the following network access fields:

- ◆ **Get API** - the API call to use to get the signal from the bus
- ◆ **Polling Mode** - allows you to choose synchronous polling or periodic polling (in synchronous, polling is handled by the execution manager that owns the network port in the hierarchy)
- ◆ **Polling Period** (if Periodic mode is selected) - interval at which the port will poll the input - in ticks
- ◆ **Polling Delay** (if Periodic mode is selected) - delay between system startup and the first polling action - in ticks

For outNetworkPorts, the features dialog also contains the following network access fields:

- ◆ **Set API** - the API call to use to set the signal value on the bus

AutomotiveC Properties

The file *AutomotiveC.prp* is loaded when you create a project based on the AutomotiveC profile.

The file contains the specific properties for the various environments, as well as many properties used to provide the unique features of the Extended Execution Model.

Fixed-point Variable Support

The AutomotiveC profile provides support for fixed-point variables by incorporating the various elements that are included in Rhapsody's *FixedPoint* profile:

- ◆ Predefined types representing 8, 16, and 32-bit fixed-point variables
- ◆ Stereotypes
- ◆ Typedefs representing the predefined fixed-point variable types
- ◆ Macros used for carrying out operations on fixed-point variables.

For more information regarding the use of fixed-point variables in Rhapsody, see [Using Fixed-point Variables](#).

Simulink and StateMate Block Integration Capabilities

When you create a new project based on the *AutomotiveC* profile, Rhapsody also loads the *SimulinkInC* profile and the *StateMateBlock* profile.

This allows you to include Simulink and StateMate blocks in your model.

StateMateBlock in Rhapsody

The StateMate system, an IBM product, is a high-level graphical development environment. Many companies use StateMate for their modeling needs. However, some StateMate customers requested the flexibility to provide co-execution of models in StateMate and Rhapsody. The integration of these two modeling products allows the seamless code-to-code merging of a StateMate model into a Rhapsody architecture. This integration has the following prerequisites:

- ◆ Rhapsody in C version 7.1.1 or higher
- ◆ StateMate 4.2 MR2 or higher installed and licensed
- ◆ License for StateMate MicroC code generator

Preparing a StateMateBlock for Rhapsody

To synchronize a StateMate model with Rhapsody, the StateMate model must have the following characteristics:

- ◆ Only one top-level activity
- ◆ MicroC profile with only one module


Perform the following operations in StateMate to prepare the StateMate model for Rhapsody:

1. Open the StateMate model. If you want to show StateMate animation in Rhapsody, be certain to select the GBA option from the StateMate MicroC profile options.
2. To set the required properties in StateMate before generating code, follow these steps:
 - a. Open the StateMate MicroC Code Generator.
 - b. Select **Options > Settings > Application Configuration > Application Files**.
 - c. In the Application Files dialog box, select both of the **Generate Code in a Single File** and **Generate Code as StateMate Block** items.
 - d. Click **OK**.
3. Generate the C code using the StateMate MicroC Code Generator.
4. In the StateMate main interface, select the **Files** tab and follow these steps:

- a. Select a Statemate MicroC code generator profile.
- b. Select the menu item: **Configuration > Create Statemate Block Configuration for Rhapsody > Read mode/Update mode.**

Creating the StatemateBlock in Rhapsody

In order to create a Rhapsody element for the Statemate model, follow these steps:

1. Open Rhapsody.
2. Select **File > New** to create a new Rhapsody project. Fill in the **Project name** and **In folder** boxes.
3. In the **Type** box of the New Project dialog box, select the **StatemateBlock** profile from the pull-down menu.
4. Rename the automatically created diagram to relate to your project.
5. In the diagram, use the Object  icon on the **Drawing** toolbar to create an *object* with a name that is appropriate for your project.
6. Right-click the object in the diagram and select **Features** from the menu.
7. In the **General** tab of the Features dialog box, select the **StatemateBlock** class stereotype for the object and click **OK** to save the change.
8. Right-click the StatemateBlock object and select the **Import/Sync Statemate Model** option from the menu.
9. In the Import/Sync State Block dialog box, fill in the fields in this order:
 - a. The default in the Statemate Installation path should be the path to your Statemate installation in this format `<STM_ROOT>\pm` (pm = project management). If the location of the Statemate pm file is not correctly displayed in the default location, click the **Advanced** button. Then enter the correct path to your Statemate installation in the dialog box. Click **OK** to save the path information and return to the Import/Sync State Block dialog box.


Note: This is an important step because the path entered here resets all Rhapsody path references to your Statemate system to use this newly entered path as the default Statemate path. See [Troubleshooting Statemate with Rhapsody](#) if you receive error messages while performing this operation.
 - b. Select **Statemate Project** from the pull-down list in the next field.
 - c. Select **Statemate Block** from the pull-down list in the next field.
 - d. Select **Statemate Workarea** from the pull-down list in the next field.

- e. In the **Activation Period (msec)** field, enter the time period between calls to the StateMate Block's execution code. This value must be greater than or equal to 50.
- f. Click **Import/Sync** to validate the previous selections and perform the import and synchronization operations if the entries are valid. If one or more of the entries contain errors, the system displays a validation error message. See [Troubleshooting StateMate with Rhapsody](#) for more information.

Connecting and Synchronizing StateMate and Rhapsody

The StateMateBlock, created in the previous procedure, operates as a black-box for StateMate code within the Rhapsody architecture once it has been connected and synchronized. The StateMateBlock interface of the top-level flowing data within the StateMate model is specified in Rhapsody using flowports.

To connect the two models, follow these steps:

1. In the Rhapsody diagram from the previous procedure, use the Object  icon to create a object with the appropriately named flowports.
2. Connect the Rhapsody flowports to the ports on the StateMateBlock object via links.
3. To produce an executable, perform code generation and build the entire Rhapsody model.

The StateMateBlock in Rhapsody automatically synchronizes with the StateMate model and adds or removes flowports from the StateMateBlock to reflect any changes made in the StateMate top-level flowing data. The synchronization operation uses a Rhapsody Block Configuration containing the following StateMate data:

- ◆ StateMate MicroC Profile with a single module
- ◆ StateMate charts that are in the scope of the MicroC profile. The top-level chart must have a single top-level (regular) Activity
- ◆ StateMate Panels that are in the scope

Troubleshooting Statemate with Rhapsody

When entering information into the Import/Sync Statemate Block dialog box, you may receive one of these error messages. The table below shows the possible error messages and their explanations.

Error Message	Explanation
"Cannot load libraries. Please make sure you are using the correct Statemate installation path."	Rhapsody was unable to locate the stmBlockInterfaceDll.dll in the bin directory of the installation path entered into the dialog box. Correct the Statemate Installation Path so that the DLL can be located.
"PM Filepath not found. Please specify a valid PM path."	The Statemate PM file name entered into the Statemate PM Location field must contain "pm.dat" in the name.
"Invalid Statemate Project. Please select a Statemate Project before pushing OK."	The Import/Sync process has checked the Statemate MicroC profile and the Statemate project was not found or the project name did not match the one entered in the Rhapsody dialog box.
"Invalid Rhapsody Block name. Please select a Statemate Block before pushing OK."	The Import/Sync process has checked the Statemate MicroC profile and cannot locate the StatemateBlock that was entered in the Rhapsody dialog box.
"Invalid Statemate Workarea name. Please select a Statemate Workarea before pushing OK."	The Import/Sync process has checked the Statemate MicroC profile and the Statemate Workarea name was not found that matched the name entered in the Rhapsody dialog box.
"Missing Statemate Block's charts. Would you like to perform check-out and generate code now?"	The Import/Sync process checked for the Statemate code that should have been generated as described in Preparing a StatemateBlock for Rhapsody . If it needs to be generated, click Yes in this error message box.
"Missing generated code for StatemateBlock. Would you like to generate code now?"	The Import/Sync process checked for the Statemate code that should have been generated as described in Preparing a StatemateBlock for Rhapsody . If it needs to be generated, click Yes in this error message box.
"Missing required files. Cannot synchronize with Statemate model."	If you selected No when the system offered to generate the Statemate code (see the two previous error messages), this error message indicates that the Statemate model cannot be synchronized with the Rhapsody until the Statemate code has been generated.

DOORS Interface

Software engineers typically need to show traceability of requirements between customer documents and specifications. Moreover, understanding the ramifications of adding or deleting requirements in complex designs, or ascertaining which requirements drove the creation of certain design elements, can be a daunting task.

Rhapsody works with the Dynamic Object Oriented Requirements System (DOORS) to track and manage design requirements throughout the lifetime of a project and to navigate between the design and the requirements, in either direction, online.

The DOORS interface exports design information stored in Rhapsody to the DOORS environment. Design information can include classes, variable and type information, design diagrams, statecharts, and transitions. In DOORS, the information is represented in a logical form as hierarchical requirements inside formal modules reflecting the original hierarchy of the elements in the Rhapsody model. Thus, consistency is maintained between both environments.

The requirements management task is performed within DOORS. Typically, the DOORS tool maintains project documents, user documents, and documentation of changes. System specification and modeling are performed within Rhapsody. The model is built, however, to meet the requirements stored in DOORS, which is the owner of the requirements. Prototyping and analysis done in Rhapsody verify that the model is consistent with your requirements.

The interface works by sharing information between the Rhapsody model and the DOORS database. Requirements are traced by transferring shadow copies of Rhapsody elements into a DOORS formal module, where the shadows are internally linked into the DOORS database.

Note

A “Rhapsody handle” string is attached to each DOORS shadow object to trace the connection from the DOORS shadow to the original Rhapsody element.

Installation Requirements

To use the DOORS interface:

- ◆ Copy the `dxlapi.dll` dynamic link library from the `bin` directory in the DOORS installation to your `winnt\system32` directory.
- ◆ Make sure that the file `pc_server.dxl` is in the `$OMROOT\etc` directory of the Rhapsody installation.
- ◆ DOORS must be installed on the local machine.

Rhapsody reads the locations of your DOORS installation and license from the Windows registry and the `LM_LICENSE_FILE` environment variable. If for some reason DOORS is not registered in the registry in the normal way and your `LM_LICENSE_FILE` variable is not set, you can manually set the `InstallationDir` and `LmLicenseFile` properties under `RTInterface::DOORS`. If these properties are set, they override the registry value or environment variable.

To set the DOORS properties, do the following:

1. Click **File > Project Properties** and select `RTInterface::DOORS`.
2. Set the `InstallationDir` property to the location of your DOORS installation. For example:

```
d:\doors
```

3. Set the `LmLicenseFile` property to the location of your DOORS license. For example:

```
d:\doors\lib\license.dat
```

If you are using a client license for DOORS, you can enter a port number and server using the following format:

```
<port>@<server>
```

Using DOORS Version 7.0

By default, DOORS 7.0 does not show the links module. To show the links module (named “Rhapsody_links” by default), select **View > Show Link Modules** in the main DOORS window.

Solaris-Specific Information

Rhapsody 4.1 on Solaris supports DOORS 4.1.4 only. Use the following settings:

```
setenv DOORSHOME /tools/DOORS4.1.4
setenv SERVERDATA /tools/DOORS4.1.4/data
setenv PORTNUMBER 36677
```

```
setenv DOORSDATA /tools/DOORS4.1.4/data
set path = ( $path $DOORSHOME/bin )
setenv LM_LICENSE_FILE 7192@lily
```

In these settings, `lily` is the name of the DOORS license server.

To use DOORS 6.0 on Solaris systems, use Rhapsody 5.0. The settings are as follows:

```
setenv DOORSHOME /tools/doors6.0
setenv SERVERDATA /tools/doors6.0/data
setenv PORTNUMBER 36677
setenv DOORSDATA $PORTNUMBER@bee
setenv LOCALDATA /tools/doors6.0/data
set path = ( $path $DOORSHOME/bin )
setenv LM_LICENSE_FILE 7192@lily
```

In these settings:

- ◆ `lily` is the name of the DOORS license server.
- ◆ `bee` is the name of the DOORS data server.

Using Rhapsody with DOORS

The general process for using Rhapsody with DOORS is as follows:

1. Set up a project within DOORS.
2. Capture requirements and other design information in DOORS.
 - Note:** DOORS is the owner of the requirements. If you need to make changes to requirements, make them in DOORS.
3. Capture the high-level use cases, structure, sequences, and behavior of the system in Rhapsody.
4. Select elements from the Rhapsody model that you want to trace to elements in DOORS.
5. Export the selected elements as shadow copies to the DOORS database.
6. Navigate to the exported elements in DOORS from the Rhapsody browser.
7. Create links within DOORS between the requirements and shadow copies.
8. Run the Check tool to verify the consistency of shadows in the DOORS database with elements in the Rhapsody repository and the completeness of the links between the two.
9. Navigate from the DOORS database to the respective elements in Rhapsody, as desired.

Navigating from DOORS to Rhapsody

You can select an object in the DOORS exported elements module and navigate to the matching element in the Rhapsody model. The navigation operation highlights the selected element in Rhapsody. When navigating to a browser element, such as a class, the element is highlighted in the Rhapsody browser. In the case of a state or transition, the matching statechart opens and the appropriate state or transition is highlighted. In the case of diagrams, the diagram will open in Rhapsody.

To navigate from DOORS to Rhapsody, do the following:

1. Select any shadow object in the DOORS formal module.
2. Select **Rhapsody > Navigate to Rhapsody**. The Rhapsody application loads, with the matching element highlighted.

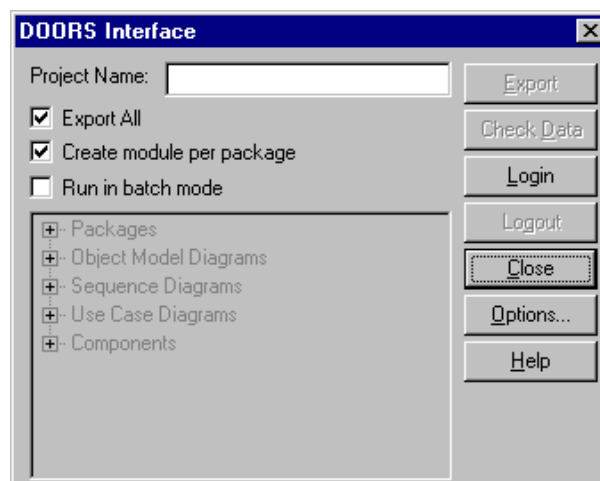
Creating the DOORS Project

The project must already exist in DOORS before you can connect to it from Rhapsody. In DOORS, create the project to which you will export Rhapsody data if the DOORS project does not already exist.

Invoking the DOORS Interface

To invoke the DOORS interface in Rhapsody, do the following:

1. Open the Rhapsody project you want to export.
2. Select **Tools > DOORS Interface**. The DOORS Interface dialog box opens, as shown in the following figure.



3. In the **Project Name** box, type the name of the DOORS project to which you want to connect.
4. Select the mode using the **Run in Batch Mode** option. The default value is interactive mode (unchecked).

To be able to navigate to DOORS objects from the Rhapsody browser, you must use interactive mode.

Note

If you want to launch DOORS, click **Login** and provide your DOORS username and password.

Setting Export Options

Export options enable you to control:

- ◆ Which Rhapsody design elements are exported
- ◆ Whether all elements are exported to one DOORS formal module or to several modules
- ◆ Whether each package is exported to a separate formal module
- ◆ Which metatypes (such as packages or classes) to export

The **Export All** option enables you to export all design elements in the current Rhapsody project as shadows to DOORS.

To export all design elements in the current Rhapsody project as shadows to DOORS, select the **Export All** check box. When you select **Export All**, the browser tree is grayed out, indicating that you can no longer select individual packages or diagrams.

To explicitly select individual packages or diagrams, clear the **All types** check box and select one or more element metatypes to export using the browser tree.

Note that beginning with Version 5.0.1, you can export nested packages to DOORS. When you select a package to export, that package and all the packages nested under it are exported.

Selecting Which Formal Modules to Create

The **Create module per package** option enables you to export design elements from the Rhapsody project to formal modules with the same names as their packages in the DOORS project. If this option is disabled, all elements are exported to a single formal module named `RHAPSODY_MODULE`.

Note the following:

- ◆ When models are exported using the **Create module per package** option, you cannot navigate to DOORS from a class or top-level object model diagram (one that is not in a

package). In both cases, you will receive a message stating that the element's package could not be found.

- ◆ While you are in **Create module per package mode**, a new empty package will not be detected; in this mode, the package has no shadow in DOORS—only a module.

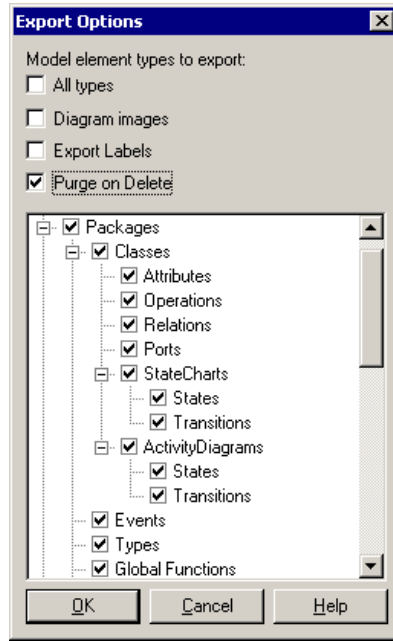
Selecting Export Options

Export options enable you to select the types of elements that can be exported, rather than the individual elements that actually are exported. For example, if you choose to export elements of metatype `Package`, you can still opt to export one particular package but not another. If you choose not to export elements of metatype `Package`, no individual elements of that type will be exported.

Examples of exportable metatypes include packages, classes, attributes, object model diagrams, statecharts, relations, operations, states, transitions, and constraints. Generally, all of the metatypes that you see in the Rhapsody browser are exportable, and more. States and transitions of statecharts are also exportable metatypes.

To select which metatypes to export, do the following:

1. Click **Options**. The Export Options dialog box opens, as shown in the following figure.



2. To export all metatypes in the Rhapsody model to DOORS, select the **All types** check box. The browse tree is grayed out, indicating that you can no longer select individual metatypes.

To explicitly select individual metatypes to export, clear the **All types** check box and select one or more element metatypes to export using the browser tree.

3. To export the graphics of diagrams and statecharts to DOORS, select the **Diagram images** check box, or set the `RTInterface::ExportOptions::ExportPictures` property to Checked. When you have turned on this option, every diagram that you export to DOORS has an OLE object inserted in its shadow. The OLE object holds the diagram graphics as an RTF file stored in the Rhapsody project directory.

Note: If WORD is not installed on the machine, the OLE object will not be created.

4. To export element labels instead of names, select the **Export Labels** check box.

5. There are two kinds of deletion:

- a. **Hard delete**—The element and its link is deleted from the DOORS database.
- b. **Soft delete**—The element is marked as deleted, but remains in the database so it can be recovered. The link is deleted.

The property `RTInterface::ExportOptions::PurgeOnDelete` controls the deletion type used in DOORS. By default, this property is set to `Checked` (hard delete).

To permanently delete the element, select the **Purge on Delete** check box.

Note the following:

- c. When there is an extra element in DOORS that does not exist in Rhapsody, the system asks whether you want to delete it.
- d. If you soft delete an element and later create an element with the same name, a *new* shadow element is created in DOORS—the old one is not used.

The metatypes are shown in hierarchical order, with packages and diagrams at the top of the tree. This is analogous to the way metatypes are shown in the browser. The same information hierarchy is maintained in the DOORS formal module as in the Rhapsody model.

Note the following:

- ◆ You can deselect a subordinate metatype only if its higher-level metatype is selected. For example, you must select packages in order to export classes, events, types, globals, use cases, or actors.
- ◆ When you export only particular types of data, change those elements and then re-export the model without exiting Rhapsody, DOORS “sees” only those elements that were originally exported and updates them in DOORS.
- ◆ If you have exited Rhapsody and re-opened it before re-exporting, the default setting in the Export Options dialog box returns to **All Types**.
- ◆ States are organized according to their position in the state hierarchy in the source statechart. Transitions are organized under their source state. The hierarchy of transitions and states is the same as in the Rhapsody reporter tool.
- ◆ Constraints can only be exported to DOORS with their owners. Constraint shadows become the children of their owner’s shadow.
- ◆ Diagram shadows are created in the shadow of the module for the package to which they belong.
- ◆ Generalization of classes, use cases, and actors are mentioned in the derived shadow’s attributes in DOORS. The short text attribute in DOORS holds the following:

<Super ELEMENT_TYPE> : <object_name>

In this syntax, <object_name> is the name of the parent relating to that shadow. To view the short text attribute in DOORS, right-click the shadow of the element and select **Edit**.

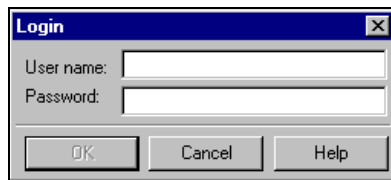
Linking the Data

You link Rhapsody elements to DOORS requirements by first exporting information that identifies the elements to DOORS. The information is inserted into a DOORS module as an object. You can link any exported element that is not undefined or unresolved in Rhapsody.

Once you have set the desired export options, you are ready to export the project.

To export the project, do the following:

1. In the DOORS Interface dialog box, click **Export**. The Login dialog box opens, as shown in the following figure.



2. Type your DOORS user name and password.
3. Click **OK**.

If Rhapsody cannot connect to the DOORS project, the specific DOORS error message appears. If DOORS cannot be run at all, Rhapsody displays an error message.

Once you have successfully logged in, Rhapsody exports design information to the DOORS project. For each Rhapsody element for which the DOORS project does not yet have a shadow, one is created. If the DOORS project already has a shadow for a particular element, the shadow is updated with current information.

After all shadow information is updated in the DOORS project, Rhapsody automatically checks the data to verify that all elements were correctly exported.

Rhapsody remains connected to DOORS for the duration of the Rhapsody session. Therefore, if you close the DOORS Interface dialog box, you do not need to reconnect to DOORS during the same Rhapsody session.

Together with each shadow element, Rhapsody exports structure information that allows DOORS to mirror the hierarchy of information shown in the Rhapsody browser. Rhapsody also maintains internal information on exported elements.

Links in DOORS between Use Case and Sequence Diagrams

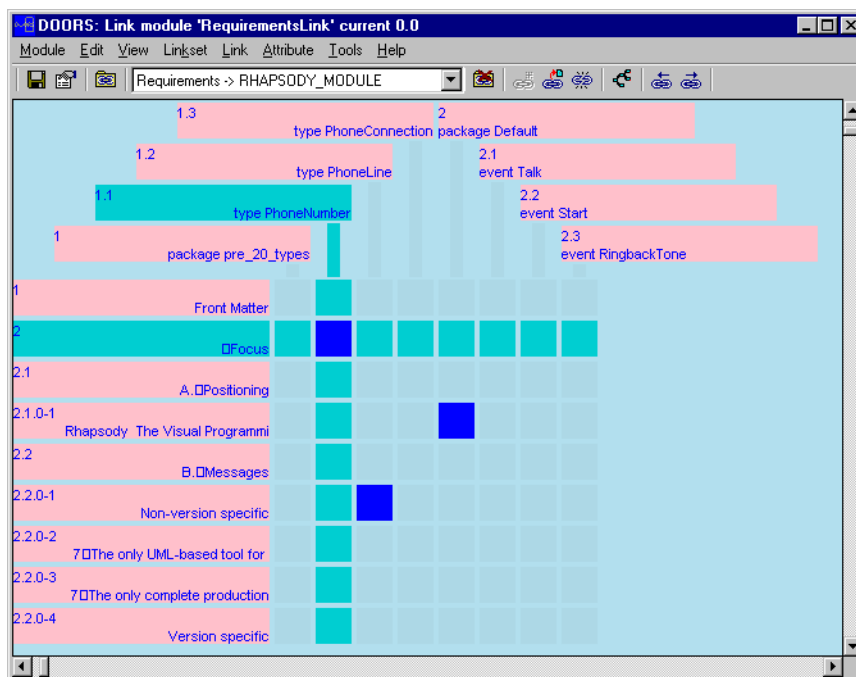
Certain element types in Rhapsody are connected in logical relationships with each other. For example, a use case diagram in Rhapsody can hold references to the sequence diagram that describes it. Rhapsody creates a link between the relational shadows (for example, the use case shadow and its corresponding sequence diagram shadow) during the export operation. These connections are expressed in the DOORS formal module, like any other Rhapsody model information.

Linked elements in Rhapsody include the following:

- ◆ Use cases and their sequence diagrams
- ◆ Use cases and their collaboration diagrams
- ◆ Sequence and collaboration diagrams, and classes that participate in each.

A link module named `Rhapsody_links` (*default name*) is added to the DOORS project to describe the links between the shadows. You can control the name of the link module using the property `RTInterface::DOORS::LinkModuleName`.

The link module describes the links as entry points in a matrix of the shadows. Links can be between shadows in the same module or between shadows from different modules. The blue squares specify a link between the use case “arming and disarming” and sequence diagram “Arming the a...” and between the use case “changing code” to the sequence diagram “Changing the c...”



A link is graphically specified in the DOORS formal module as an arrowhead, red for outgoing links and orange for incoming links.

To navigate from a shadow to its links shadows, do the following:

1. Position the cursor on the arrow specifying the link.
2. Click the right mouse button.
3. Select the desired shadow from the shadows links list.

Information Stored in DOORS

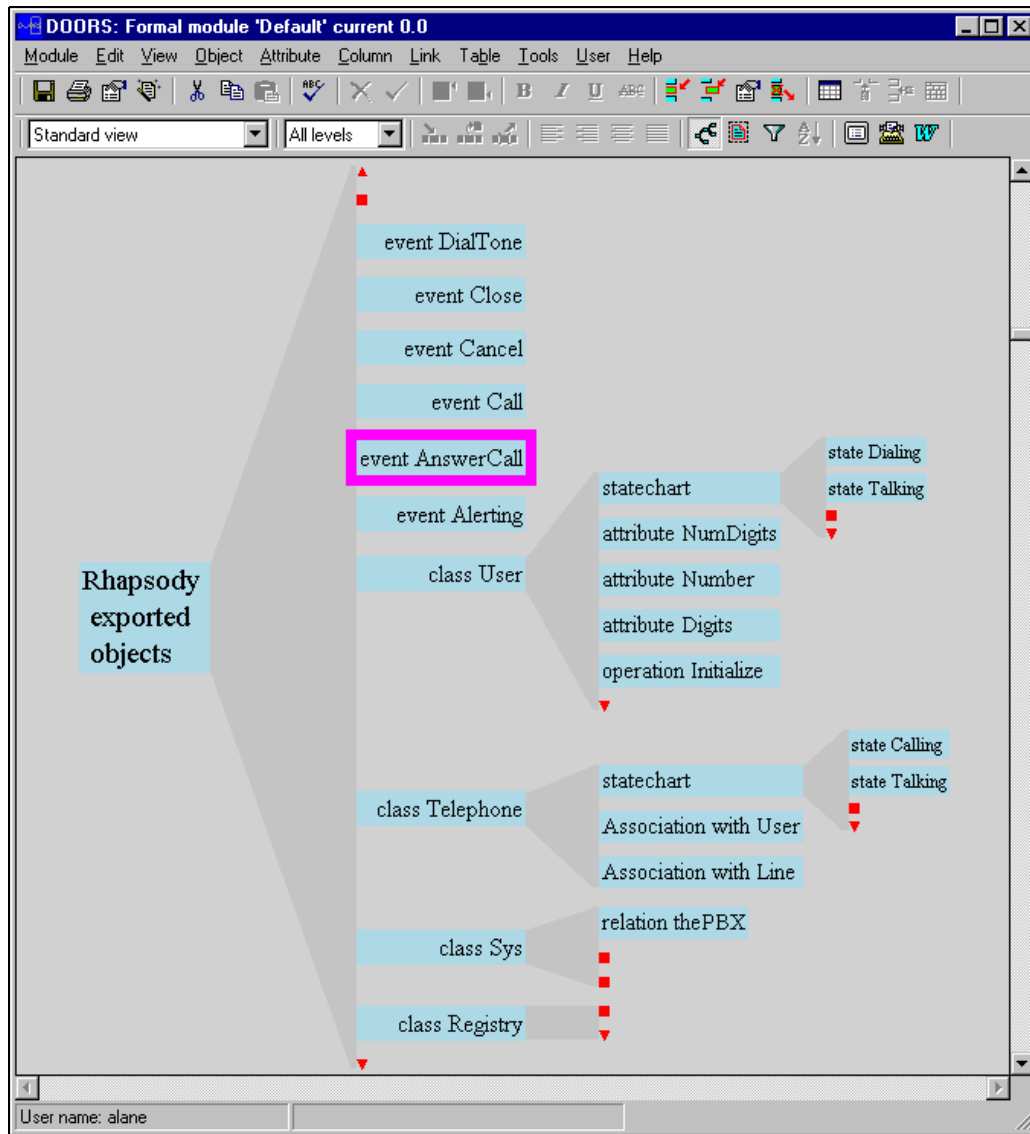
Each exported object is a DOORS shadow element containing the following information:

- ◆ The name of the Rhapsody element.
- ◆ The package or diagram to which the element belongs in Rhapsody.

Note the following:

- If you selected the option **Create module per package**, each exported element belongs to a formal module corresponding to the package in Rhapsody. Otherwise, the DOORS object is hierarchically located under the object representing the package in the `RHAPSODY_MODULE`.
 - If you want the DOORS project name to reflect the Rhapsody project name, set the `RTInterface::DOORS::ModuleNameFromProject` property to `Checked`. If you set the property to `Checked`, the elements are exported to a formal module matching the name of the Rhapsody project, plus a leading “R”. For example, if the project name is `Pbx`, the formal module name will be `RPbx`.
- ◆ The type of the Rhapsody element.
 - ◆ The description of the Rhapsody element.
 - ◆ In special cases, descriptive information is added to the shadow element in DOORS. For relations, for example, information is added similar to that contained in a Rhapsody report.

The following is the graphical view of a Rhapsody project that has been exported to DOORS. It shows the `Default` package, which has been exported to a DOORS formal module of the same name, and the hierarchy of some of the shadow objects, such as events, classes, statecharts, and states, that are part of the package.



Information Stored in Rhapsody

Rhapsody maintains unique identification information for each design element that was successfully exported to DOORS so Rhapsody can retrieve the DOORS internal data for a shadow at any time.

Checking the Data

The Check Data operation checks for inconsistencies between the Rhapsody source elements and their related shadows in the target DOORS project. Rhapsody automatically invokes the Check

Data operation after each successful export operation. You can also check data independently of an export.

To start the Check Data operation, click **Check Data**.

If the Check Data operation finds an inconsistency, the Problem Resolution dialog box opens, which lists any inconsistencies found and offers various ways to deal with the problem.

If you try to re-export the model after closing and then reopening Rhapsody, because the Export Options dialog box reverts back to the **All Types** selection, you might receive errors about elements that have not been exported because they were not among those selected for the earlier export. To ensure that the export does not give you errors of this kind, be sure to reselect all the same metatypes as you selected in previous exports.

Problem Description

The Problem Description dialog box lists inconsistencies discovered by the Check Data operation. To view the next problem, choose one of the operation buttons, such as Ignore or Update, which are active when a problem is selected.

Each problem description includes the following:

- ◆ Definition of the problem
- ◆ Name of the element in Rhapsody, if applicable
- ◆ Type of the element in Rhapsody, if applicable
- ◆ Package to which the element belongs in Rhapsody
- ◆ Name of the shadow element in DOORS, if applicable
- ◆ Type of the shadow element in DOORS, if applicable
- ◆ DOORS formal module to which the shadow was exported

To handle a particular consistency problem, select the problem in the Problem Description list. Each type of problem has an appropriate resolution. Rhapsody offers a default resolution and alternatives, depending on the type of problem. The following table summarizes the types of problems detected by the Check Data operation and their possible resolutions.

Problem	Description	Possible Resolutions
DOORS element is outdated.	Invalid name in DOORS for an element that was renamed or otherwise edited in Rhapsody. The message lists the element's name, type, and package in both DOORS and Rhapsody for comparison.	Default: Update DOORS to match the newer information in Rhapsody. Alternate: Ignore.
DOORS element is not connected to any Rhapsody element.	DOORS shadow element points to a non-existent element in Rhapsody.	Default: Delete shadows. Alternate: Ignore.
Package/diagram missing.	The package, diagram, or statechart that existed in the Rhapsody project when the information was initially exported to DOORS no longer exists in the Rhapsody project.	Default: Ignore Alternate: Delete shadows for all related elements
Missing link in Rhapsody element.	The Rhapsody element is not linked to a shadow in DOORS, although a shadow exists in DOORS that matches the element's unique Rhapsody identification information.	Default: Update both the Rhapsody and DOORS information. Alternate 1: Ignore Alternate 2: Delete the element in DOORS and reexport the Rhapsody project.
Rhapsody element is not connected to any DOORS element.	The Rhapsody element has no link to a DOORS shadow element.	Default: Export the element. Alternate: Ignore the error and export later.

The following buttons are enabled depending on the appropriate solution for a selected problem:

- ◆ **Update**—Updates the DOORS project with the current Rhapsody information so it is consistent in both places. The Update operation also renews the shadow ID on the Rhapsody side to keep the link to the shadow element current.

Updating is the preferred way to deal with stale data or missing links.

- ◆ **Delete Shadow(s)**—Deletes the shadow elements in DOORS that are related to the selected problem.

Deleting shadows is the preferred way to deal with shadows that have invalid names.

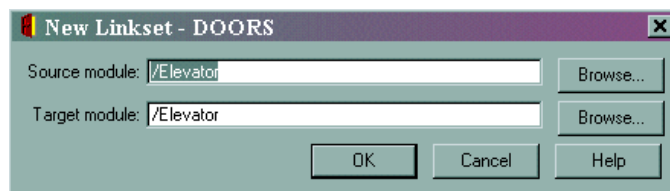
- ◆ **Delete & Create New**—Deletes an existing, erroneous shadow and creates a new, accurate one.
- ◆ **Ignore**—Tells Check Data to proceed with the next problem to be resolved without making any changes to either the Rhapsody or the DOORS project.

Ignoring the inconsistency is the preferred way to deal with missing packages or diagrams.

Mapping Requirements to Imported Elements

You map the imported shadow elements to requirements by creating a link module in DOORS. To create a link module, do the following:

1. In the DOORS Project Manager, select **New > Link Module**.
2. Use the New Link Module dialog box to specify a the name for the link module, its description, and a mapping value.
3. Click **OK** to apply your changes and dismiss the dialog box. The Link Module dialog box opens.
4. Select **File > New > Linkset**. The New Linkset dialog box opens, as shown in the following figure.



5. Specify the source and target modules, then click **OK** to create the link.

DOORS creates a new link module in which requirements from the source module line the left margin of a blank table in the middle of the screen, and shadow objects from the target module line the top margin of the table.

To link requirements to shadow objects, do the following:

1. Right-click a blank square in the table and select **New Link**.
2. In the Edit Link Object dialog box, edit any of the link attributes.
3. Click **Close**.

That square is marked in blue to indicate that a requirement on the Y-axis is linked to a shadow object on the X-axis.

Ending a DOORS Session

To end a DOORS session, do any of the following:

- ◆ Click **Logout** in the DOORS Interface dialog box.
- ◆ Close the Rhapsody project.
- ◆ Exit Rhapsody.

If you are running DOORS in interactive mode, you can exit DOORS from the DOORS window.

Other Information

Note the following:

- ◆ Modeled annotations are supported by DOORS like any other modeled elements.
- ◆ Files are exported to and checked by DOORS.

Summary

The objective of the DOORS interface is to represent a Rhapsody model in a DOORS module. The formal module must always contain the most current information about Rhapsody model elements. Thus, you can treat a Rhapsody project as a special kind of requirements file filled with model elements. This enables you to link requirements to actual Rhapsody model elements that fulfill those requirements. Remember that DOORS is the owner of the requirements. If you need to make changes to requirements, make them in DOORS.

You can transfer information about complete Rhapsody models or subsets of models into DOORS. You select elements to transfer by constructing a list using the Rhapsody browser. In this way, you can only update subsets of the model if it takes too long to transfer the entire model.

If the Rhapsody model is changed, the DOORS module might not contain the correct information. Normally, retransferring the Rhapsody elements would correct the DOORS module. However, it is possible for some inconsistent information to remain. For this reason, the DOORS interface provides a Check Data option, which lists problems and guides you through their correction so the DOORS shadow elements always match the corresponding Rhapsody model elements.

Importing Rose Models

The Rose Importer imports models created in IBM Rational Rose into Rhapsody. Components of the Rose Logical View, such as packages, classes, and relations between classes, are mapped to similar entities in Rhapsody. Class and state diagrams from Rose are imported as object model diagrams and statecharts, respectively, in Rhapsody. In addition, you can import activity, component, sequence, and use case diagrams, along with templates and template instantiations.

Note

The Rose Importer imports the Rose Logical View, Use Case View, and Component View. It does *not* import the Deployment View.

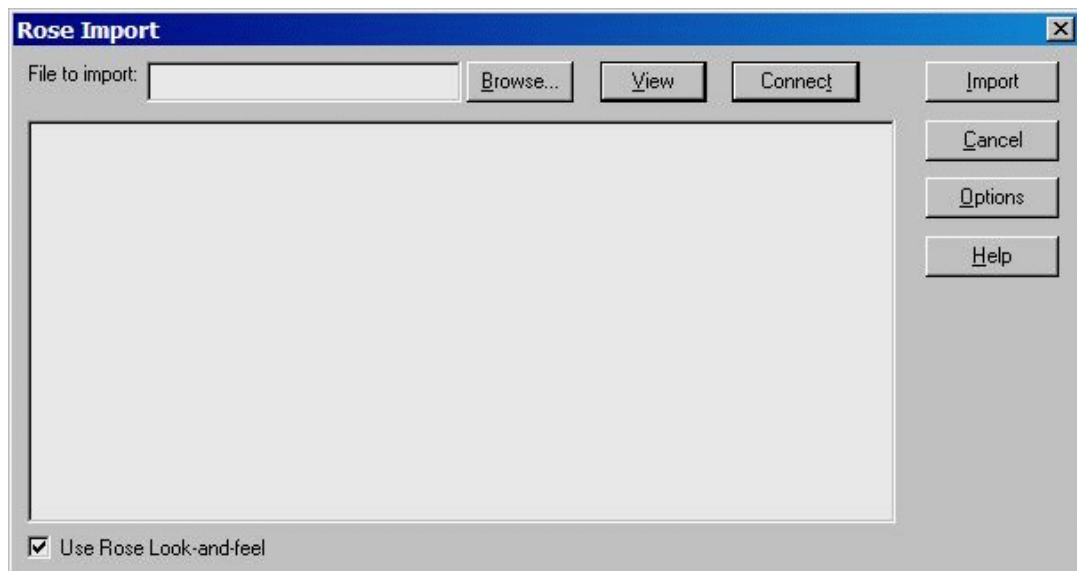
In addition, Rose must be installed on the Rhapsody host machine for import to work. It is not sufficient to simply import a model created in Rose without Rose actually being installed on the Rhapsody machine.

Importing a Rose Model

Before importing a Rose model, verify that the model is correct from the Rose perspective. Use the Rose check model function and clear all reported errors in the model before importing it. Attempting to import a model with errors might result in problems using the Rose Import utility.

A target project must first exist in Rhapsody before you can import a Rose model. To create the Rhapsody project and import a Rose model, follow these steps:

1. With Rhapsody running, create the new project by either selecting **File > New**, or clicking the **New project** icon in the main toolbar.
2. Replace the default project name (Project) with `<your project name>` in the **Project name** box. Enter a new directory name in the **In folder** box or Browse to find an existing directory.
3. In the **Type** box, select the Default profile to use the Rhapsody standard project settings.
4. Click **OK**. If the directory does not exist, Rhapsody asks if you want to create it. Click **Yes** to create the new project directory.
5. Select **Tools > Import from Rose**. The Rose Import dialog box opens, as shown in the following figure.

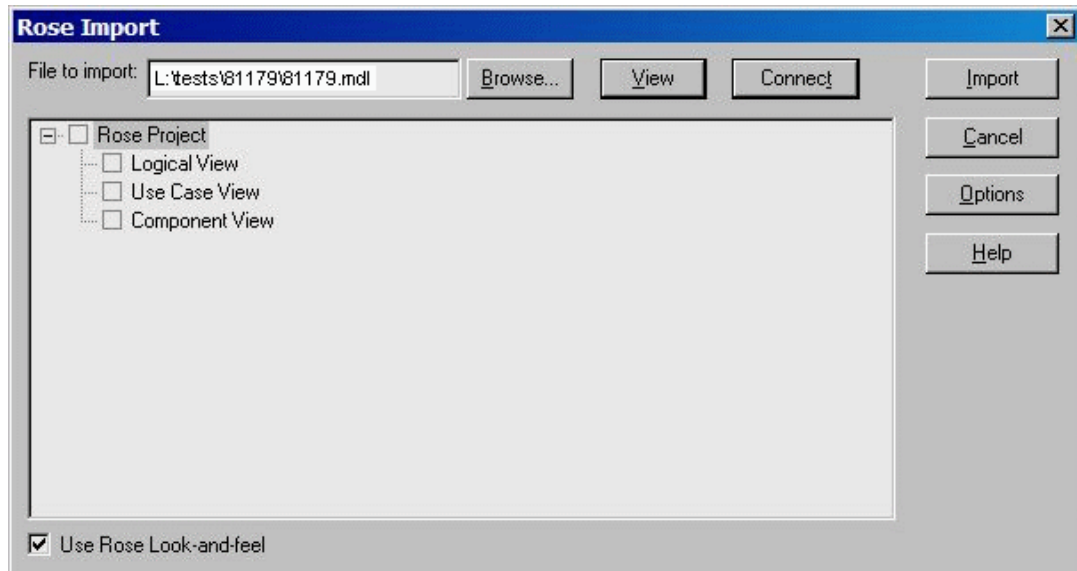


6. Notice that Rhapsody automatically opens the Output window for you. Now you can begin [Selecting Elements to Import](#).

Selecting Elements to Import

To select a Rose model to import, follow these steps:

1. On the Rose Import dialog box, do whichever of the following is applicable to fill in the **File to import** box:
 - ◆ If you have the Rational Rose environment and the Rose model you want to import open, click the **Connect** button to fill in the **File to import** box.
 - ◆ Use the **Browse** button to locate the Rose .mdl file you want to import. Or you can type the name, including the full path, of the Rose model in the **File to import** box.
2. Once the Rose .mdl filename appears in the **File to import** box, the Logical View, Use case View, and Component View branches for the Rose model to be imported are displayed on the Rose Import dialog box, as shown in the following figure.



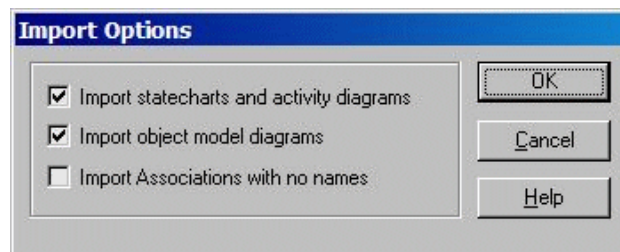
3. Expand the contents of a view choice and select the elements you want to import. Note the following:
 - ◆ Clicking the check box for the main (top) branch selects or clears all sub-branches and their elements.
 - ◆ Clicking the check box for a sub-branch selects or clears that sub-branch and all its elements.
 - ◆ Right-clicking a check box either clears or selects that specific element, depending on its current state.

4. By default, the **Use Rose Look-and-feel** check box is selected. If you do not want the imported Rose project to have the look-and-feel of a Rose project, you can clear this check box.
5. Continue with [Setting Import Options](#).

Setting Import Options

Import options include whether to import object model diagrams and statecharts. To set the import options, follow these steps:

1. In the Rose Import dialog box, click **Options**. The Import Options dialog box opens, as shown in the following figure.



2. If needed, change the settings for the current project.

Note: Rhapsody will automatically use these settings the next time you do an import. For example, if you select the **Import statecharts** check box and clear the **Import object model diagrams** check box, this setting will be used for all subsequent imports until you change the settings again.

3. Click **OK** to confirm your selections
4. Continue with [Starting the Import](#).

Note

Rose allows names with spaces, but Rhapsody does not. Rhapsody approximates spaces in names by replacing them with underscores. For example, a package named “Course roster” in Rose becomes “Course_roster” when imported into Rhapsody. There are other characters not allowed in Rhapsody names (such as &, #, \$, and %). For these characters, Rhapsody will use underscores or truncate the names.

Starting the Import

Once you have selected the elements you want to import and set the import options, you are ready to import the Rose model. Before you import, you may want to be sure of or do the following:

- ◆ If you are re-importing the same package(s) from Rose, remember that the name(s) in Rhapsody and in Rose must be exactly the same.
- ◆ If necessary, move the Rose Import dialog box away from the Output window before you start the import so that you can see any messages as they occur.

To import the model, follow these steps:

1. Click **Import**.
2. If a top-level package with the same name as one you are importing already exists in the Rhapsody model, the following message displays:

```
Packages Logical_View, Use_Case_View, Component_View already exist. Do you want to continue?
```

To continue with the import, click **Yes**. This means that any package that is re-imported will be totally overwritten.

3. The import process begins. Progress meters and possible messages regarding “lost data” are written to the Output window. Here are examples of types of messages:

```
Error: Can't import association itsTerminal from IControlDevice. It has only one role.
```

```
...
```

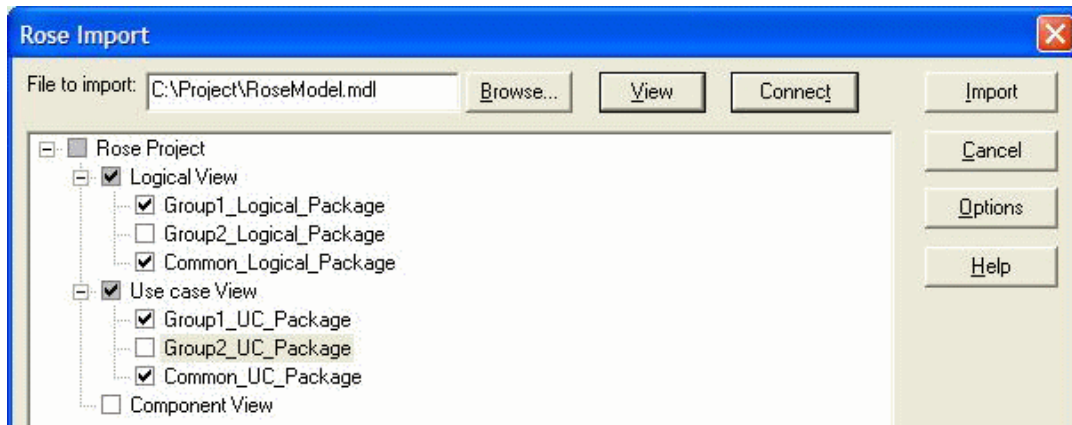
```
Error: Can't add operation GetPropertyValue to class IControlDevice, there is a name or signature clash.
```

```
...
```

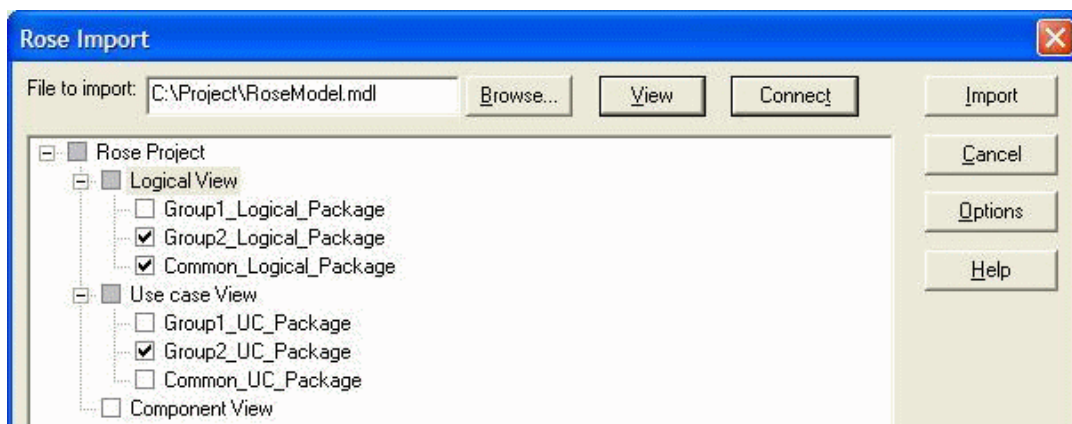
```
Error: Can't override statechart for derived class IAlarm.
```

Incremental Import of Rational Rose Models

Rhapsody allows you to import Rose models incrementally. This makes it easier to import large models according to your workflow processes. For example, for a very large model, you may have more than one team, with each team assigned a specific part of the model. You can import each part of the model in separate import sessions. For example, you may import all of Group1, as shown in the following figure.



Then you could import Group2, as shown in the following figure. Notice also in this example you can re-import a Rose package (in the Logical View, the Common_Logical_Package has been selected again for importing). When you import a Rose package that has already been imported, the incremental import process will completely override the contents of that package in Rhapsody.



Incrementally imported parts of a Rose model will be integrated correctly in Rhapsody when possible. An example of when it is not possible: A class has an association with another class in the original Rose model. During incremental import, only the first class is imported. The other

class is located in another package, which will be imported later. The association between these classes will remain unresolved (that is, incomplete) until you import the second class.

Before the Import Process Starts

Prior to the import process, for performance reasons, Rhapsody will close all diagram windows that may be open (neither saving or unloading diagrams). In this event, the following message appears:

```
All opened diagrams will be closed prior to Rose Import.  
Please click the OK button to proceed or the Cancel button to cancel import.
```

Click **OK** to continue.

About Processing Time and Project Size

The internal steps required for this incremental importing process result in slightly longer processing time as well as slight increases in the size of the project. The size increase is due to the data that must be saved to allow the importing of further increments of a model. You can use **Tools > Clean Project Import Data** to delete the data that was stored in order to allow these incremental imports. By default, this menu item is not visible. To make this menu item visible, add the following line to the [General] section of the `rhapsody.ini` file:

```
ShowCleanImportData=TRUE
```

Note

This data is necessary for importing further increments of the model you have imported. Use this menu option only after you have completely finished importing all of the Rose model. Once import data is destroyed, incremental import of the particular Rose model is no longer possible.

Mapping Rules

The following table shows how various Rose constructs and options map into a Rhapsody model. For ease of use, the Rose elements are listed in alphabetical order.

Rose Element or Option	Rhapsody Element	Notes
Abstract class		Not imported.
Action	Action	
Activity diagram	Activity diagram	
Actor	Actor	
Anchor note to item	Anchor	
Association	Link, linktype = association	See Importing Association Classes .
Cardinality of classes	Part	Class cardinality refers to the number of instances of a class that can be created at run time. A class with exactly one instance has a cardinality of one. In Rhapsody, a class's cardinality is referred to as its <i>multiplicity</i> . The Multiplicity box reflects the cardinality of the class in the original Rose model.
Category	Package	
CategoryDependency	Dependency	
Class	Class	
Class type	Type = class	All types of classes are mapped to classes.
ClassifierRoles	ClassifierRoles	
Collaboration diagram	Collaboration diagram	Not imported.
Component Package	Package	
Component	Component	Since Rhapsody does not allow components to be contained in packages, any imported components will be included under the project level.
Component diagram	Component diagram	
Concurrency—sequential, active, guarded, or synchronous	Concurrency—sequential or active	Operation concurrency is not imported.
Condition	Guard	
Constraints		Not imported.
Containment—by value, reference, unspecified		Not imported.
Dependency (UCD)	Dependency	
Deployment diagram	Deployment diagram	Not imported.

Rose Element or Option	Rhapsody Element	Notes
Derived attributes and relations		Not imported.
End state	Termination connector	
Event	Event	Events trigger transitions from one state to another. Events are imported as classes whose behavior includes triggering state transitions.
Export control		Not imported.
Friend	Property	
Global package		Not imported.
HasRelationship	Link, linktype = aggregation	
Inheritance (use cases)	Inheritance	
InheritRelationship	SuperClass, superevent	
Initial value of attribute		Not imported.
Interface	Class	Interface classes are imported into Rhapsody as classes with virtual operations.
IsConstant (Rose property)		Not imported.
Link Attribute		Not imported.
Link Element		Not imported.
Messages	Messages	
Multiplicity of relations	Multiplicity	
Navigable relation	Feature (from class to class)	In Rhapsody, you cannot add a Navigable feature if there is a navigation (cannot have both Navigable and aggregation).
Nested class		Not imported.
Note	Note	
Operation type—virtual, static, friend, abstract, common	Virtual, static	
OperationIsConst (Rose property)		Not imported.
Parameter	Argument	
Persistence		Not imported.
Private implementation	Private implementation	
Protected implementation	Protected implementation	
Public implementation	Public implementation	

Rose Element or Option	Rhapsody Element	Notes
Qualifier/keys	Qualifier	In Rose, a qualifier might not be a class attribute. In Rhapsody, a qualifier <i>must</i> be a class attribute. Rhapsody approximates qualifiers depending on whether they are also attributes in Rose. If the qualifier is an attribute in Rose, it is mapped to an attribute in Rhapsody. Otherwise, Rhapsody creates an attribute, adds it to the class, and makes it the qualifier. Rose allows multiple qualifiers, whereas Rhapsody allows only one. Therefore, when you import an association with multiple qualifiers, Rhapsody randomly takes the first one it sees.
Qualifier type	Attribute	If the qualifier is not a class attribute, create it.
Relation	MetaLink Relations in UCDs are imported as relations.	Abstract class.
Relation type—by value, by reference, unspecified	All three types map to By reference.	
RealizeRelation	SuperClass	
Role	Role	
Send argument	Action	
Send event	Action	
Send target	Action	The Rose Send event/argument/target are mapped to the Rhapsody action using the following format: <code>Sendtarget->GEN (Sendevent (Sendarguments))</code>
Sequence diagram	Sequence diagram	When Rhapsody imports sequence diagrams from Rose, the Rose ClassifierRoles are converted to Rhapsody ClassifierRoles and Classifiers, and messages are converted to actual operations on the target (receiving) class.
Space of class		Not imported.
Start state	Default transition	Combined with outgoing transition.

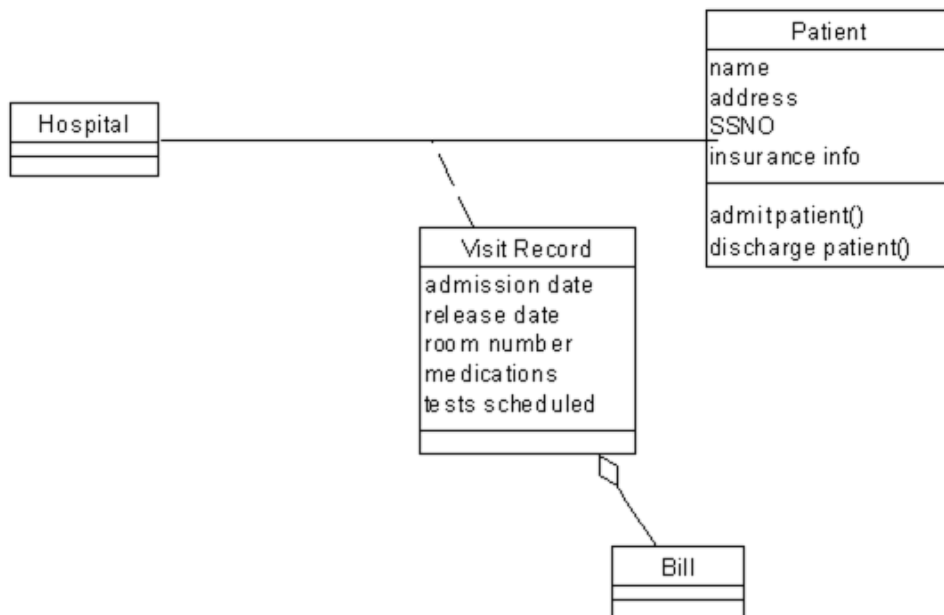
Rose Element or Option	Rhapsody Element	Notes
State	State	If there is more than one view for a single state in Rose, when imported into Rhapsody, the additional views will be converted into new states in the model with the same characteristics (like you would get with the Copy with Model feature). The name will indicate that it is a new state, but the label will be the same.
Static attributes	Static attributes	
Static relation	Static (relation is a static class member)	
StereoType		Not imported.
Substate	State (with parent)	If there is more than one view for a single substate in Rose, when imported into Rhapsody, the additional views will be converted into new substates in the model with the same characteristics (like you would get with the Copy with Model feature). The name will indicate that it is a new substate, but the label will be the same.
Templates and template instantiations	Templates and template instantiations	
Text box	Note	Same as object model.
Transition	Transition	The format for a transition in the diagram is as follows: <code><Event> [<Guard>] / <Action></code> If there is more than one view for a single transition in Rose, when imported into Rhapsody, the additional views will be converted into new transitions in the model with the same characteristics (like you would get with the Copy with Model feature). The name will indicate that it is a new transition, but the label will be the same.

Rose Element or Option	Rhapsody Element	Notes
Types—predefined (such as <code>int</code> or <code>float</code>), user-defined, or class.	Type	<p>When you create a user-defined type in Rose, you can give it a name but no declaration. Rhapsody approximates a user-defined type by adding an on-the-fly type with the new type name as its declaration.</p> <p>In Rose, you can also assign a class type, such as <code>ParameterizedClass</code> or <code>InstantiatedClass</code>. Rhapsody approximates class types by creating an on-the-fly type with the class as its declaration.</p>
Use cases	Use cases	
UseRelation (ClassDependency)	Dependency between packages is saved only in the graphical interface.	

Importing Association Classes

If a class does not have associations or a statechart, it is imported as an association class; otherwise, it is imported as a regular class.

Consider the following hospital model:



In this example, `Visit Record` is a class associated to the `Hospital_Patient` association. Therefore, it could be imported as association class.

If the `Visit Record` class has a statechart or associations with other classes, it will *not* be imported as an association class, but will be imported as a class. As shown in the figure, because `Visit Record` has an association with class `Bill`, it will be imported as a regular class. However, the association `Hospital_Patient` will have a hyperlink to this class.

If `Visit Record` does not have associations or a statechart, it is imported as association class. That means:

- ◆ The name of the association `Hospital_Patient` will be `Visit Record`.
- ◆ The attributes and operations of `Visit Record` will be displayed under the association class.

XMI Exchange Tools

XMI (XML Metadata Interchange) is a format specification produced by the Object Management Group (OMG). The XMI format allows the interchange of objects and models through an XMI formatted file. This is commonly used to exchange UML models between other tools or software.

In addition to XMI, Rhapsody provides additional tools for developers to examine the models, such as the following:

- ◆ ReporterPLUS reports in Word, PowerPoint, and HTML format. Reports are created without any conversion to another format.
- ◆ Model simulation capabilities show how the model components work together.
- ◆ The COM API exports a set of COM interfaces representing the metamodel objects and application operational functions.
- ◆ Write macros provide a means to examine the model within Rhapsody.

Using XMI in Rhapsody Development

Rhapsody's XMI export and import feature facilitates the following development tasks:

- ◆ Export an entire Rhapsody model to XMI to be closely examined as a whole
- ◆ Export the whole model to XMI to be searched in an HTML browser
- ◆ Export the model to XMI in order to parse the entire model with another UML tool or a non-UML tool
- ◆ Imports XMI models or pieces of other XMI models into Rhapsody models
- ◆ Exchange models to or from the Tau system (For more information, refer to the *Rhapsody Tau Integration* manual in <Rhapsody installation path>\Sodius\XMI_Toolkit\doc.)

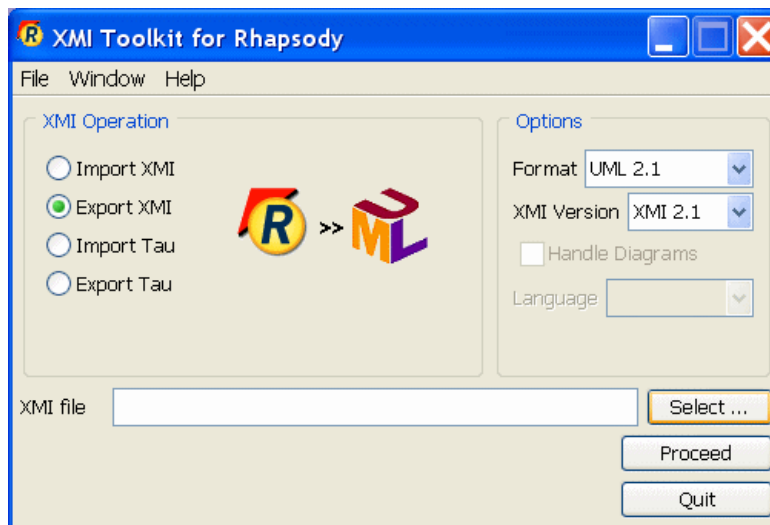
Exporting a Model to XMI

Engineers, designers, or architects may export a Rhapsody project to an XMI file for any of the following reasons:

- ◆ Share a model with another UML tool
- ◆ Create a text file for your model so it is parsable
- ◆ Create a file that can be searched in an HTML browser

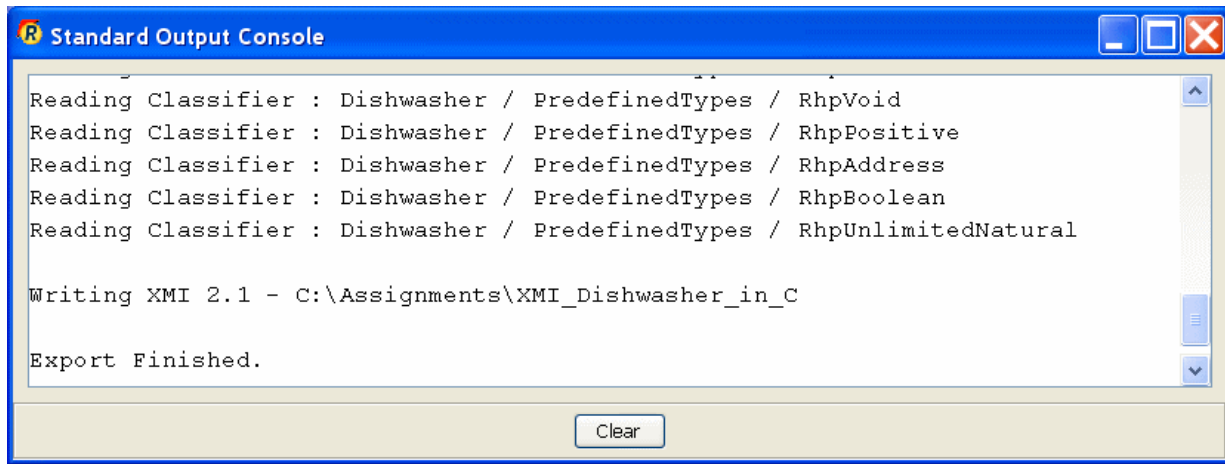
To export a Rhapsody model to an XMI file, follow these steps:

1. Display the model for export in Rhapsody.
2. Select the Rhapsody **Tools > Export XMI** options.
3. In the **XMI Operation** area, select whether you are exporting to XMI or to Tau.
4. Then select the UML **Format** of your project files as either 1.3 or 2.1. If you select UML 2.1, the **XMI Version** is automatically set to 2.1. If you select UML 1.3, you may select 1.0, 1.1, or 1.2 as the XMI Version for your exported file.



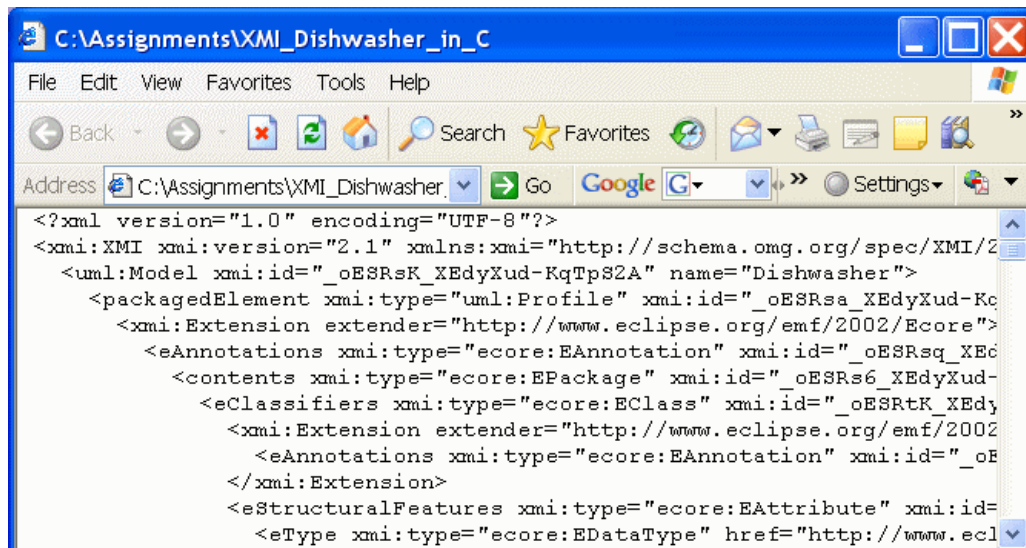
5. If you select the UML 1.3 format, you may decide to **Handle Diagrams** and output the model's diagrams in the UNISYS extensions format during the export operation.
6. In the **XMI file** box, select a directory for the exported file.
7. Click **Proceed** to export the model as defined.

The system displays any messages relating to the export in a dialog box, as shown here.



Examining the Exported File

After exporting a model, you can open the exported file in any standard browser to examine the details of the model. The following example shows an exported Rhapsody model displayed in the *Windows Internet Explorer* browser.



Note

If the **UML 1.3** and **Handle Diagrams** options are selected, the exported file includes the Rhapsody model diagrams in the UNISYS extensions format.

Importing an XMI File to Rhapsody

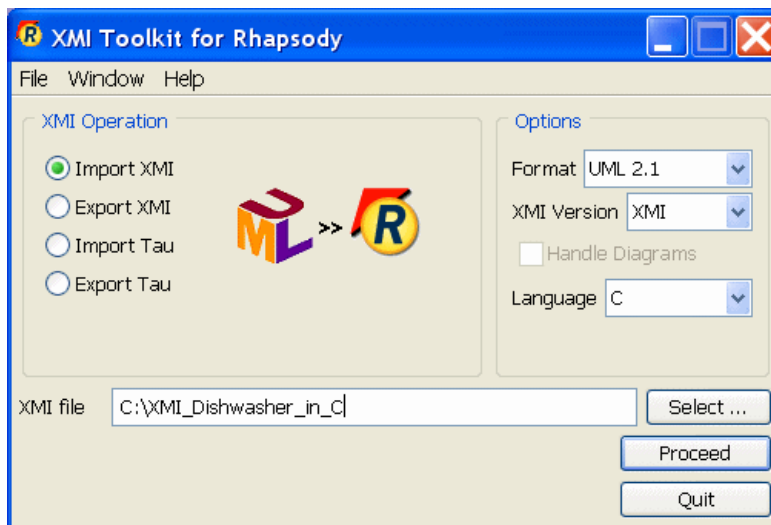
Rhapsody has the ability to import a model from an XMI file into Rhapsody for either of these reasons:

- ◆ A file from another UML tool needs to be brought into Rhapsody. For example, a TAU file could be brought into Rhapsody this way.
- ◆ A file from a non-UML tool needs to be brought into Rhapsody.

Important: Normally you would not export and import XMI files between Rhapsody users. XMI is intended as a means for vendor-neutral data sharing.

To import an XMI file from another source, follow these steps:

1. Create a new Rhapsody project or open an existing Rhapsody project.
2. Select the Rhapsody **Tools > Import XMI** options.
3. In the **XMI Operation** area, select whether you are importing from XMI or from Tau.
4. In this dialog box, select the UML **Format** for the imported file as either 1.3 or 2.1.



5. If you select the UML **Format** for the imported file as UML 1.3, you may also select whether or not the import operations should **Handle Diagrams**.
6. Select the **Language** for your imported file as C, C++, Ada, or Java.
7. In the **XMI file** box, select the directory from which to import the file.
8. Click **Proceed** to import the XMI file as defined and add it to the Rhapsody project.

The system displays messages relating to the import in a dialog box, as shown previously.

More Information

For more information about the XMI toolkit features, refer to the following documentation in `<Rhapsody installation path>\Sodius\XMI_Toolkit\doc`:

- ◆ User Guide
- ◆ Mapping Rules Overview
- ◆ Rhapsody Tau Integration

Integrating Simulink Components

Rhapsody can be used in conjunction with Simulink, the MATLAB extension that allows modeling of continuous processes using block diagrams.

Rhapsody allows you to integrate Simulink models into Rhapsody designs. Simulink models are represented as “Simulink blocks” in the UML model, and these blocks may interact with Rhapsody objects/parts or other Simulink blocks. The integration of Simulink blocks into Rhapsody uses a “black box” approach, in which only the input/output ports of the Simulink blocks are exposed, appearing as flowports in the Rhapsody model. To send/receive data to/from a Simulink block, you use links to connect these flowports to flowports of other Simulink blocks or of other Rhapsody objects. When code is generated for a Rhapsody model containing Simulink blocks, the code generated by Simulink is wrapped into the Rhapsody-generated code.

If changes are made to the Simulink model, you can synchronize the representation of the Simulink model in your Rhapsody project with the updated model.

In general, the process for including such Simulink components in a Rhapsody model is as follows:

1. Build the Simulink model using Real-Time Workshop.
2. Import the model into Rhapsody as a *SimulinkBlock*. The Simulink input and output ports will appear as atomic flowports on the *SimulinkBlock* element. (See [Flow Ports](#).)
3. Connect the flowports of the *SimulinkBlock* element to the flowports of the relevant elements in the Rhapsody model.

The following software is required for integrating Simulink components into a Rhapsody model:

- ◆ Matlab must be fully installed and licensed (Matlab 7), with Simulink (version 6) and the Real-Time Workshop component (which generates C and C++ code from Simulink models).
- ◆ Rhapsody 7.0 or greater

Note

The `..\Samples` directory contains a sample Rhapsody model that includes Simulink integration.

Importing Simulink Components

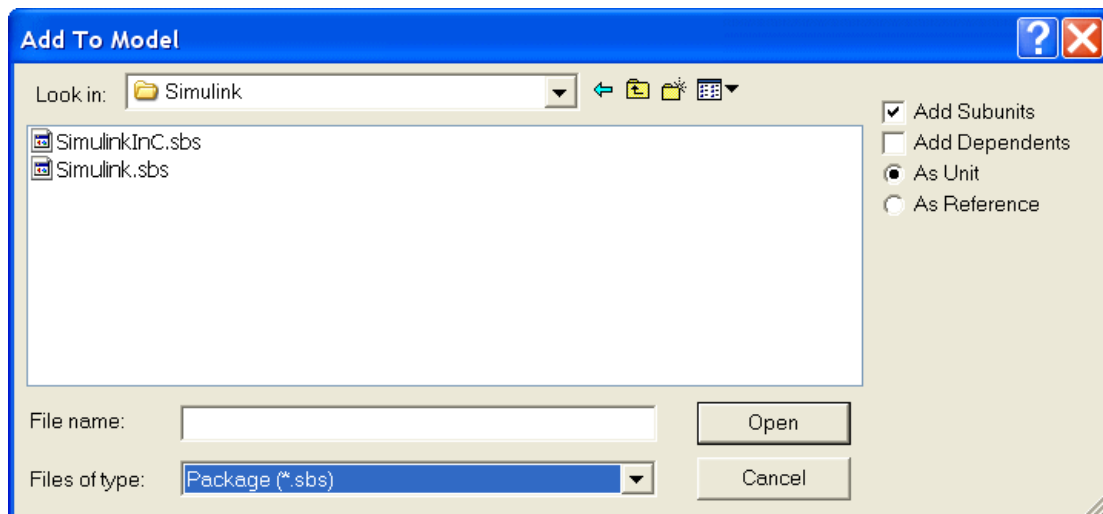
To import a Simulink component, carry out the following steps in Simulink and in Rhapsody:

In Simulink

1. Create a Simulink model, or open an existing one, and save it in your working directory, preferably in the same working directory as your Rhapsody model.
2. For generating code, use the following settings (most are the default settings) in the Real-Time Workshop. You can view the settings by selecting **Tools > Real-Time Workshop > Options**.
 - ◆ Hardware Implementation->Device Type - Unspecified (assume 32bit Generic)
 - ◆ Real-Time Workshop->System target file - ert.tlc
 - ◆ Real-Time Workshop->Language - C or C++ (note that the default setting is C)
 - ◆ Real-Time Workshop->Make command - make_rtw
 - ◆ Real-Time Workshop->Template makefile - ert_default_tmf
3. Generate code for the Simulink model (**Tools > Real-Time Workshop > Build Model**).

In Rhapsody

1. Create a new Rhapsody project.
2. Right-click the project name in the browser and select **Add to Model > Package**.
3. Navigate to your Rhapsody installation's Share/Profiles/Simulink directory. Select the Package (*.sbs) for **Files of type**, as shown in this example.



4. Select the `SimulinkInC.sbs` profile if you are using C and `Simulink.sbs` if you are using C++. Click **Open** to add the selected profile to the project. Check the `Profiles` section in the browser to be certain that the selected Simulink profile is now displayed.
5. Create an object in an object model diagram, and apply the *SimulinkBlock* stereotype to it (in the Features dialog box).
6. Right-click the object and select **Import/Sync Simulink Model**.
7. In the dialog box that is displayed, provide the following information:
 - ◆ **Simulink Model File.** The location of the Simulink model file
 - ◆ **Simulink Generated Source Code.** The location of the *.cpp files generated by the Real-Time Workshop (add all files except `ert_main.cpp`).
 - ◆ **Simulink Model Sample Time.** The interval (in milliseconds) at which Rhapsody should activate the Simulink engine.
8. Click **Import/Sync** and wait until Rhapsody creates flowports on the block representing the input and output of the Simulink model.
9. Once the flowports have been created, you can connect the Simulink block to the flowports on other Rhapsody blocks.

Integration of the Simulink-generated Code

When Simulink components are imported into a Rhapsody model, the .cpp files generated from the Simulink model using Real-Time Workshop are included as source files in the Rhapsody-generated makefile.

In terms of Rhapsody-generated code, *SimulinkBlock* elements in Rhapsody are classes that are based on a framework class called `OMSimulinkBlock`. The superclass periodically calls the method `doStep()`, which is implemented by the derived class. This method initializes the input port, calls the step function in the Simulink-generated .cpp file, and sets the value of the output after the step. (The output is then relayed via the output flow port.)

The `doStep()` function will be generated once you assign the *SimulinkBlock* with a Simulink model and use the **Import/Sync Simulink Model** context menu command. Note that an Embedded Coder License (ERT) is required for this operation.

Troubleshooting Simulink Integration

- ◆ If after importing or synchronizing with your Simulink model, you get an error message about a missing file, *langeng.dll*, verify that MATLAB's `\bin\win32` folder is in your PATH environment variable. After adding it, you will have to restart Rhapsody and try reimporting.
- ◆ If you get compilation errors regarding missing include files, look for them in the MATLAB installation directory. After locating them, you can add them to the include search path for the Rhapsody configuration.

Creating Simulink S-functions with Rhapsody

Using Rhapsody in Conjunction with Simulink

Rhapsody can be used to create Simulink S-functions that can then be plugged into Simulink models.

The specific steps to carry out to create S-functions in Rhapsody are described in [Creating a Simulink S-function](#).

For a broader picture of S-function creation in Rhapsody, see [S-function Creation: Behind the Scenes](#).

Note

This feature is only applicable in Rhapsody in C.

Creating a Simulink S-function

To create a Simulink S-function in Rhapsody and then use it in Simulink:

1. Create a new Rhapsody project.
2. Create a new configuration and apply the stereotype *S-FunctionConfig* to it.
3. Set the newly-created configuration to be the active configuration.
4. Create a new class, and apply the stereotype *S-FunctionBlock* to it.
5. Add incoming flowports to the class to represent incoming data.
6. Add outgoing flowports to the class to represent outgoing data.
7. For each of the flowports you added, add an attribute to the class to represent the flowport. The attribute must have the same name and be of the same type as the corresponding flowport.
8. Implement a statechart for the class.
9. Generate code for the configuration you created.

10. The output directory for the configuration will include the following:
 - ◆ generated source files for the model
 - ◆ Rhapsody framework files (from Rhapsody's IDF framework)
 - ◆ a Simulink C template file called *RhapsSFunc_(the name you gave to the block).c*
 - ◆ a mex options file called *MexOpts.txt*
 - ◆ a Simulink model file, representing the S-function block, called *RhapSFunc_(the name you gave to the block)_Model.mdl*
11. Open MATLAB and go to the output directory containing the Rhapsody code.
12. Run the command `mex @MexOpts.txt`.

S-function Creation: Behind the Scenes

When you generate code for an *S-FunctionConfig* configuration, Rhapsody performs the following actions:

- ◆ Completes the *sfuntmpl_basic.c* template provided by Simulink, and renames it to reflect the name you assigned to your S-function block.
- ◆ Takes the information you have entered for the S-function block in your project and creates a corresponding Simulink model file, using the name you assigned to your S-function block.
- ◆ Generates a mex options file, containing the necessary compiler switches and list of source files to use.

When you run the mex command using the mex options file generated by Rhapsody, MATLAB's MEX compiler creates a binary file that can be used by Simulink.

Timing and S-Functions

For time-related events, Rhapsody uses the timing mechanism of the target operating system. Since Simulink has its own timing mechanism, Rhapsody takes this into account when generating the S-function code. The Simulink clock is added as an input to the S-function. This is not visible to the user in Rhapsody, but when the resulting files are imported into Simulink, you see a clock element in addition to the element representing the defined S-function.

Limitations

When creating Simulink S-functions in Rhapsody, keep the following in mind:

- ◆ You can only have one S-function per configuration.

- ◆ Rhapsody's animation feature does not work with S-function blocks.

Using the Rhapsody Command-line Interface (CLI)

Rhapsody provides command-line options for individual system features, such as for the [DiffMerge Tool Functions](#). You may run the full version of Rhapsody (Rhapsody.exe) from the command line. To assist with command-line operation, Rhapsody includes a lightweight non-GUI version of the program (RhapsodyCL.exe), which allows you to use a subset of the full set of Rhapsody command-line options.

Note

RhapsodyCL.exe is located in the same directory as Rhapsody.exe.

RhapsodyCL

RhapsodyCL allows you to use Rhapsody's code-related functions, such as generate and make, in contexts where you do not require the GUI elements, for example, as part of a nightly build procedure. Since RhapsodyCL is designed for tasks such as code generation, it does not support options relating to diagrams, for example, populating a diagram from the command line. It also does not support the commands relating to configuration management and running macros.

You may send the RhapsodyCL application commands using any of these four methods:

- ◆ Command line
- ◆ Batch file
- ◆ Interactive mode
- ◆ Socket mode

Interactive Mode

In this mode, RhapsodyCL, switches to a “shell mode” using a prompt to enter commands. You may use either of the following techniques to employ the interactive mode for the Rhapsody command-line interface:

- ◆ Adding the `-interactive` switch in the command line
- ◆ Executing RhapsodyCL with no commands

For every command the user enters in interactive mode, RhapsodyCL performs the following:

1. Executes the command.
2. Wait for more commands from the user.
3. Stop when an `exit` command is received.

Note

If any commands exist in the command line when the `-interactive` switch is entered, the existing commands are executed first, and then RhapsodyCL enters interactive mode.

Socket Mode

In this socket mode, RhapsodyCL listens on a socket port (supplied by the user), and any commands that arrive on that socket are executed immediately. RhapsodyCL stops only when it receives an `exit` command. To start the Rhapsody command-line interface in socket mode, enter this command:

```
RhapsodyCL.exe -socket <Socket_Port>
```

The `<Socket_Port>` is the number of the port that the RhapsodyCL listens to for commands.

Note

If any commands exist in the command line when the `-socket <Socket_Port>` switch is entered, the existing commands are executed first, and then RhapsodyCL enters socket mode.

Command-line Syntax

The syntax for using command-line options is the same for both Rhapsody and RhapsodyCL.

The options are in the forms of switches and commands, and the syntax is slightly different for the two groups, as described in the following sections.

Note

Any path names within these commands should not contain spaces. If spaces must be included in a path, enclose the entire path in quotation marks to direct the command to the correct location.

Switches

For switches, the general syntax is as follows:

```
-switchName=parameter
```

Example

```
Rhapsody.exe -lang=cpp ...
```

Commands

For commands, the general syntax is as follows:

```
-cmd=commandName parameter
```

Example

```
Rhapsody.exe -cmd=open modelName.rpy -cmd=generate
```

For both switches and commands, parameters are separated from the command name or previous parameter by a space. No quotation marks are used.

Switches and commands are not case-sensitive but parameters are.

Note

In general, the switches refer to global configuration settings such as language, while the commands represent common actions in Rhapsody such as open or generate.

Order of Commands

All commands must be issued in a logical order. For example, since you must open a project before you can modify and save it, the open command must precede the save command in the command-line.

There is no significance to the order of parameters.

Including Commands in a Script File

The `-f` switch can be used to call a script file consisting of a number of commands. Within a script file, there is no need for the “`-cmd`” before the command name. Comment lines in script files begin with a pound sign (`#`).

Sample Script File

```
# This is a sample script file
setlog d:\log.txt
open d:\rhapsody\samples\Dishwasher dishwasher.rpy
generate EXE gui
save
make
```

Calling a Script File

```
rhapsody -f script.txt
```

Exiting after Use of Command-line Options

For `Rhapsody.exe`, you must close Rhapsody after using the options on the command line in order to close the process, for example:

```
C:\> rhapsody -f script.txt -cmd=exit
```

With `RhapsodyCL`, however, this is not necessary. `RhapsodyCL` exits as soon as it has finished carrying out the specified commands.

Note

For `Rhapsody.exe` (but not `RhapsodyCL.exe`) `make` is an asynchronous command and should be the last command included in a script. You therefore cannot follow a `make` command with an `exit` command to close your project and exit Rhapsody. If you do so, the `make` process will end prematurely.

Return Codes

The following return codes are used for command-line options for both `RhapsodyCL.exe` and `Rhapsody.exe`:

- ◆ 0: success, no errors occurred
- ◆ 100: failed to open the project file

- ◆ 101: license not found
- ◆ 102: code generation failed
- ◆ 103: failed to load the project
- ◆ 104: failed to create or write to the code generation folder
- ◆ 105: errors were found in check model
- ◆ 106: unresolved elements in scope
- ◆ 107: error in the name of the component or configuration specified
- ◆ 108: build failed

Examples

Below are a number of examples of command-line usage with Rhapsody.

```
C:\> rhapsody d:\rhapsody\samples\Dishwasher\Dishwasher.rpy -cmd=setlog
d:\log.txt -cmd=generate EXE gui -cmd=save -cmd=make
```

This sample command line performs the following actions:

1. Invokes Rhapsody.
2. Opens the Dishwasher sample.
3. Directs the output to the file d:\log.txt.
4. Generates code for an executable component using the gui configuration.
5. Saves the project.
6. Builds the component.

The above example specifies the project to open as a parameter immediately following "rhapsody". You can perform the same action using the `-cmd=open` command, as shown below.

```
C:\> rhapsody -cmd=setlog d:\log.txt -cmd=open
d:\rhapsody\samples\Dishwasher\Dishwasher.rpy -cmp EXE -cfg gui -cmd=generate
-cmd=save -cmd=make
```

The following example illustrates the use of RhapsodyCL.

```
RhapsodyCL.exe -lang=cpp -cmd=open
d:\rhapsody\samples\Dishwasher\Dishwasher.rpy -cmd=generate
```

Command-line Switches

The switches that can be used on the command line with Rhapsody are listed below. Unless otherwise noted, switches can be used with both Rhapsody.exe and RhapsodyCL.exe.

-animport=<Number>

(cannot be used with RhapsodyCL.exe)

Instructs Rhapsody to use an animation port other than the one defined in the `rhapsody.ini` file. Using this option allows you to use animation in a number of Rhapsody instances simultaneously. (See [Running on a Remote Target](#) for more details.)

-architect

Runs the architect version of Rhapsody.

-dev_ed (default)

Runs the developer version of Rhapsody (this is also the default value if a specific version is not specified).

-f

Runs the script provided as a parameter.

-hiddenui

(cannot be used with RhapsodyCL.exe)

Hides the Rhapsody user interface. Can be used for tasks such as generating code.

Note: This switch predated RhapsodyCL. For tasks such as code generation, it is recommended that you use RhapsodyCL rather than running the full version of Rhapsody with the `-hiddenui` switch.

-interactive

Switches to a “shell mode” using a prompt to enter commands. See [Interactive Mode](#) for more information.

-lang=<language>

Specifies the code language.

-noanimation

(cannot be used with RhapsodyCL.exe)

Disables animation by not attempting to open the TCP/IP animation port. This is useful for running more than one Rhapsody instance without having to deal with the modal dialog that pops up when the animation port is not available.

-nodiagnostics

(cannot be used with RhapsodyCL.exe)

Loads the specified Rhapsody model without the diagrams it contains.

-profile=<profile name>

Starts Rhapsody with the specified profile.

-root=<pathname>

Specifies the root directory of the Rhapsody installation.

-socket <Socket_Port>

RhapsodyCL listens on the socket port, and commands that arrive on that socket are executed immediately. See [Socket Mode](#) for more information.

-system_architect

Runs the system architect version of Rhapsody.

-system_designer

Run the system designer version of Rhapsody.

-verbose

Use this switch when you want RhapsodyCL to notify a user about a wrong syntax or unsupported commands.

Command-line Commands

Unless otherwise noted, commands can be used both with Rhapsody.exe and RhapsodyCL.exe. In general, the following types of commands cannot be used with RhapsodyCL: diagram commands, configuration management commands, commands for running macros.

Note also that if you try to use a non-supported command with RhapsodyCL, the following will happen depending on if you have set the `-verbose` switch:

- ◆ If you have set the switch, RhapsodyCL will notify the user and ignore the command.
- ◆ If you have not set the switch, RhapsodyCL will simply ignore the command without any notification to the user.

Note: When making changes to projects under source control, check out the project before running RhapsodyCL.

-cmd=addtomodel <file location> <withdescendants|withoutdescendants>

Adds to the current model from the specified file location. The default value is `<withoutdescendants>`.

-cmd=arccheckout <file name> <label/revision> <locked|unlocked> <recursive|nonrecursive>

(cannot be used with RhapsodyCL.exe)

Checks out a file from the archive.

If you do not want to specify a `<label/revision>`, use NULL.

-cmd=call <plugin> <parameters for plug in>

(cannot be used with RhapsodyCL.exe)

Calls one of the Rhapsody plug-ins and forwards the provided parameters to the plug-in.

In contrast to all the other commands, the parameters for this command are provided as a single string enclosed in quotation marks. The first parameter in the string should specify the plug-in that is being called. The remainder of the string contains the parameters that should be sent to the plug-in.

Below are a few examples of using this command to run Test Conductor.

- ◆ `-cmd=call "rtc run <listname>"` will execute every test included in the batchlist with the name `<listname>`
- ◆ `-cmd=call "rtc run all"` will execute all the test defined in the TC.

- ◆ `-cmd=call "rtc run <testpath>"` will execute only the test which is in the path `<testpath>`

For example, Rhapsody `D:\RhapsodyModels\Pbx\PBX.rpy -cmd=call "rtc run all"`

`-cmd=checkin <unit name> <label/revision> <locked|unlocked> <recursive|nonrecursive> <description>` (cannot be used with RhapsodyCL.exe)

Checks in a unit to an archive. If you do not want to specify a `<label/revision>`, use NULL.

For example, `-cmd=checkin p1.sbs NULL locked recursive "my description"`

`-cmd=checkmodel`

Starts a Check Model operation.

Set the current configuration before issuing this command.

`-cmd=checkout <unit name> <label/revision> <locked|unlocked> <recursive|nonrecursive>` (cannot be used with RhapsodyCL.exe)

Checks out a unit from the archive. If you do not want to specify a `<label/revision>`, use NULL.

`-cmd=close <NoSave>`

Closes the open Rhapsody model. By default, Rhapsody will automatically save any changes you have made to the model before closing. If you do not want Rhapsody to save changes upon closing, use the NoSave parameter.

`-cmd=closediagram <diagram type><diagram name>`

(cannot be used with RhapsodyCL.exe)

Closes the specified diagram.

The first parameter specifies the type of diagram. This parameter can take one of the following values: omd, ucd, msc, collaboration, component.

The second parameter is the name of the diagram in the model.

`Connecttoarc <archive location>`

(cannot be used with RhapsodyCL.exe)

Connects to an archive. `<archive location>` includes the full path.

`-cmd=creatediagram <diagram type><diagram name>`

(cannot be used with RhapsodyCL.exe)

Creates a new diagram.

The first parameter specifies the type of diagram. This parameter can take one of the following values: omd, ucd, msc, collaboration, component.

The second parameter is the name of the diagram in the model.

-cmd=exit

Closes the project and exits Rhapsody.

-cmd=forceroundtrip

Performs a roundtrip regardless of the timestamps of the files.

-cmd=generate <component> <configuration>

Generates code for the specified component and configuration.

<component> and <configuration> are optional parameters. If not specified, the active component and configuration are used. Like the generate option in the GUI, this only generates code for modified elements. To regenerate all code, use the `-regenerate` command.

For example, `-cmd=generate EXE Acme`

If you want to generate code for more than one component, or for more than one configuration for a given component, you must repeat the `generate` command for each component/configuration combination, for example:

```
-cmd=generate compA cfg1 -cmd=generate compA cfg2 -cmd=generate compB cfg1
```

If you want to generate code for a nested component, use the syntax `outerComponent::innerComponent`, for example:

```
-cmd=generate def::abc DefaultConfig
```

Note: This command should not be used with RhapsodyCL.exe if you are using “customized code generation” or if you are generating code for the INTEGRITY operating system. Use the command with Rhapsody.exe instead.

-cmd=gmr

Performs generate/make/run.

-cmd=import

Imports classes according to the reverse engineering settings stored in the current configuration. This is equivalent to selecting the Rhapsody command **Tools > Reverse Engineering**.

-cmd=make

Builds the application, using the current configuration.

Make is an asynchronous command and should be the last of all commands in a script.

Because exit is a synchronous command, you cannot follow a make command with an exit (to close your project and exit Rhapsody); doing so will cause the make to cease prematurely.

If you plan to run the application right after the make, use `-syncmake` instead of `-make`. This waits for the make to complete before running any additional commands.

-cmd=new <project location> <project name>

Creates a new project in the specified location and assigns it the specified name.

-cmd=open <project name>

Opens the specified project. (RhapsodyCL.exe can only open projects. Rhapsody.exe can open units as well.)

-cmd=opendiagram <diagram type><diagram name>

(cannot be used with RhapsodyCL.exe)

Opens the specified diagram.

The first parameter specifies the type of diagram. This parameter can take one of the following values: omd, ucd, msc, collaboration, component.

The second parameter is the name of the diagram in the model.

-cmd=populatediagram <diagram type><diagram name>

(cannot be used with RhapsodyCL.exe)

Populates the specified diagram.

The first parameter specifies the type of diagram. This parameter can take one of the following values: omd, ucd, msc, collaboration, component.

The second parameter is the name of the diagram in the model.

-cmd=printcurrentdiagram

(cannot be used with RhapsodyCL.exe)

Prints the open diagram.

-cmd=regenerate <component> <configuration>

Generates code for the specified component and configuration.

<component> and <configuration> are optional parameters. If not specified, the active component and configuration are used. Like the regenerate option in the GUI, this regenerates all the code, not just the code for changed elements.

If you want to generate code for more than one component, or for more than one configuration for a given component, you must repeat the `regenerate` command for each component/configuration combination, for example:

```
-cmd=regenerate compA cfg1 -cmd=regenerate compA cfg2 -cmd=regenerate compB  
cfg1
```

If you want to regenerate code for a nested component, use the syntax `outerComponent::innerComponent`, for example:

```
-cmd=regenerate def::abc DefaultConfig
```

Note: This command should not be used with `RhapsodyCL.exe` if you are using “customized code generation” or if you are generating code for the INTEGRITY operating system. Use the command with `Rhapsody.exe` instead.

-cmd=report <format> <name + location>

Generates a report.

<format> is the report format (RTF or ASCII). The file extension is added automatically (.rtf for RTF and .txt for ASCII).

<name + location> specifies the name and location of the report. These parameters are optional.

If you do not specify a name, the default file name is used (`RhapsodyRep.rtf`).

If you do not specify a location, the default location is used (the project directory).

Set the current configuration before issuing this command.

For example, `-cmd=report RTF myReport`

For `RhapsodyCL`, the `report` command uses Rhapsody's internal reporter and does not extract diagrams.

-cmd=roundtrip

Roundtrips code changes back into the model.

Set the current configuration before issuing this command.

-cmd=runexternalprogram

Runs the specified external program.

(with RhapsodyCL.exe, cannot be used to run COM-based programs.)

-cmd=runvbamacro <module name> <macro_name>

(cannot be used with RhapsodyCL.exe)

Runs the specified VBA macro outside an active project. The VBA script must already exist within a Rhapsody model and be compiled so the file <model>.vba exists.

You can copy a .vba file into your project directory (where the .rpy file is located). For example, if you have the project file abc.rpy, copy your macro .vba file as abc.vba then use the `runvbmacro` command to run the macro.

In VBA, if you do not specify a module name, the default is `module1`. You cannot pass parameters to the module.

For example, `rhapsody -cmd=open d:\rhapsody\models\hhs.rpy -cmd-runvbamacro module1 first`

-cmd=save

Saves the open project. Can be used after making changes like roundtrip, reverse engineering.

-cmd=saveas <project name>

Saves the project to a specified location. <project name> can include the path.

-cmd=setcomponent <active component name>

Sets the active component.

If you want to make a nested component the active component, use the syntax `outerComponent::innerComponent`, for example:

```
-cmd=setcomponent def::abc
```

-cmd=setconfiguration <active configuration name>

Sets the active configuration.

For example, `-cmd=setconfiguration AcmeDebug`

-cmd=setlog <log file>

Redirects the output normally sent to the output window to the specified log file. If the parameter does not specify the path, the log file is put in the "current" Rhapsody directory. If a log file is specified, output is not sent to the standard output.

-cmd=setomroot <alternative OMROOT>

Sets the variable OMROOT to a new location. This variable specifies the root directory of the Rhapsody installation.

For this command to take effect, this must be the first option specified in the command line.

-cmd=syncmake

Builds the application using the current configuration.

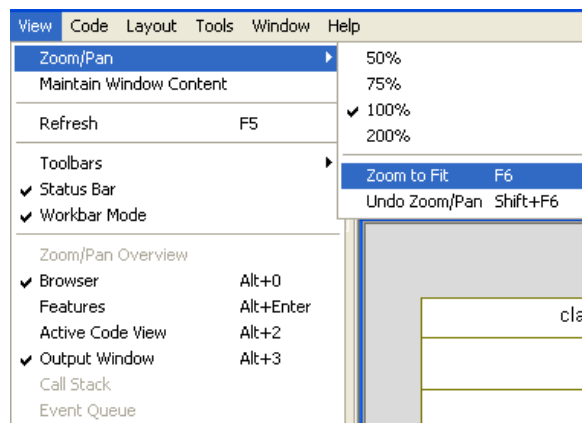
As opposed to the `make` command, the `syncmake` command will wait until the make has completed before running any additional commands. So if you plan to run the application right after building it, use `syncmake` instead of `make`.

Rhapsody Shortcuts

Rhapsody supports the following kinds of keyboard interaction: accelerator keys, mnemonics, modifiers, and standard Windows shortcuts.

Accelerator Keys

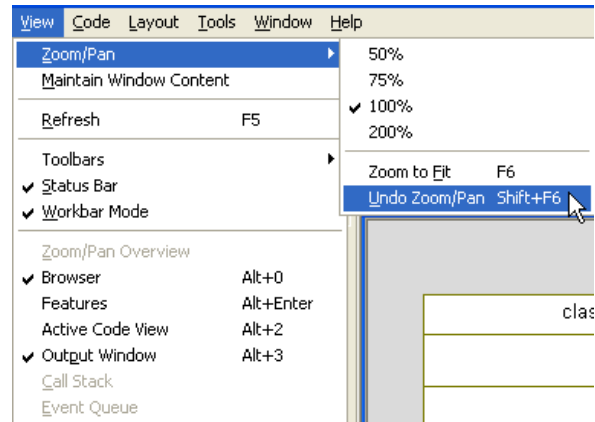
An *accelerator key* is a keyboard key (or combination keys) designated to achieve a specific action. For example, the **F6** keyboard key is an accelerator for **Zoom to fit** of a diagram, whereas **Ctrl+A** is an accelerator key for **Select All**. In most cases, the menu option that serves the same purpose as the accelerator lists the corresponding accelerator, as in this example:



Mnemonics

Mnemonics are small indicators marked as an underline under a letter in a menu name or on a button name. This indicator means that clicking Alt and the designated mnemonic letter will result in activating this menu.

For example, the following figure shows that you can achieve the **Zoom to fit** functionality not only with the accelerator key **F6**, but also using mnemonics by clicking **Alt+{V, Z, F}**: **Alt+V** opens the View menu, **Alt+Z** opens the Zoom menu, and **Alt+F** executes the Zoom to fit command.



This document does not describe the complete set of mnemonics because it is clearly visible on the menus.

Note

Sometimes the underlining is not visible in the menus. If this occurs, try pressing the Alt key until they are displayed. If you still cannot see them, follow the instructions in [Changing Settings to Show the Mnemonic Underlining](#).

Keyboard Modifiers

A *modifier* is a keyboard key applied to a command to slightly modify its behavior or meaning. For example, when using your mouse for resizing a shape in a Rhapsody graphic editor, you can use the Alt key to change the operation behavior from “resize” to “resize without contained.”

Standard Windows Keyboard Interaction

Windows applications have an extensive list of standard keyboard shortcuts to common interactions. For example, **Ctrl+Tab** toggles through the open windows within an application. Using this keyboard shortcut in Rhapsody will navigate through the open diagrams and code editors that are currently open.

Rhapsody Accelerator Keys

Accelerator keyboard keys can be further broken down into three types:

- ◆ **Application accelerators**—Activate menu commands.
- ◆ **Accelerators and modifiers within diagrams**—Assist with drawing activities.
- ◆ **Accelerators in the code editor**—Assist in coding activities.

Application Accelerators

Action	Shortcut
Accelerators for mapping and navigation	
Locate in Browser	Ctrl+L
Search in Model	Ctrl+F
Show References	Ctrl+R
Animation	
Go	F4
Go Event	F10
Quit animation	Shift+F5
Browser navigation	
Expand all	Numeric block *
Navigate	Numeric lock keys
Code	
Build	F7

Rhapsody Shortcuts

Generate	Ctrl+F7
Run	Ctrl+F5
Stop Make Execution	Ctrl+Break
Project	
New Project	Ctrl+N
Open Project	Ctrl+O
Print	Ctrl+P
Redo	Ctrl+Y
Save Project	Ctrl+S Although this usually saves the model, if you are focused on code (for example, using Edit Code), this shortcut saves the <i>file</i> , not the model.
Undo	Ctrl+Z
VBA	
Show Macros	Alt+F8
Visual Basic Editor	Alt+F11
Window management	
Arrange Options	Ctrl+W
Show/Hide Active Code View	Alt+2
Show/Hide Browser	Alt+0 (zero)
Show/Hide Features	Alt+Enter
Show/Hide Output Window	Alt+3
Help	
Help	F1

Accelerators and Modifier Usage in Diagrams

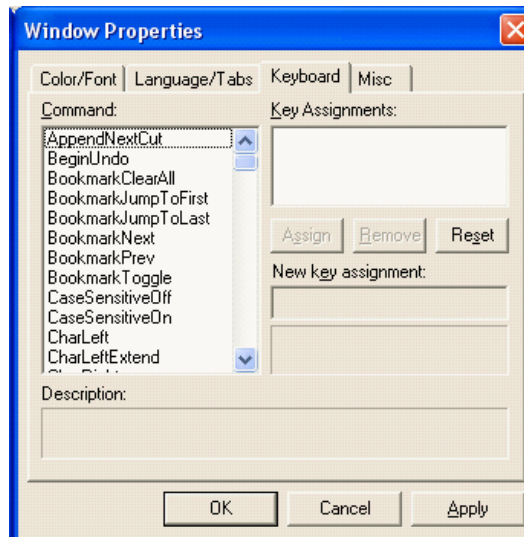
Action	Shortcut
Add a new item to a list compartment (for example, the attributes compartment of a class box).	Insert Note that this shortcut works only if you already have a list—it does not work for the first item in a list.
Change the selection anchor.	Select + Ctrl
Copy.	Ctrl+C; Ctrl+Drag You can also use this shortcut in the Rhapsody browser.
Create more space in an SD (assuming you are using a mouse).	Click+Shift and drag the mouse to display a dashed, horizontal bar. Move the bar down to create more space, or move it up to eliminate unnecessary space.
Create or resize elements (by mouse dragging) with a symmetrical shape.	Shift (while dragging) Note that using the corner anchor and Shift creates a “lock aspect ratio” sizing.
Create straight lines and arrows that are parallel to the axis.	Use the Ctrl key while drawing the lines.
Cut.	Ctrl+X
Delete from Model.	Ctrl+Delete
Insert a new user point in a line or arrow.	Ctrl+Click Note that this does not apply to rectilinear lines or to SD messages).
Paste.	Ctrl+V
IntelliVisor.	Ctrl+Spacebar
Move shape to the <i>ARROW</i> direction (“nudge”).	Ctrl+ARROW (where <i>ARROW</i> could be the up, down, left, or right arrow key)
Move shape to the <i>ARROW</i> direction without its contained elements (“nudge”).	Ctrl+Alt+ARROW (where <i>ARROW</i> could be the up, down, left, or right arrow key)

Refresh.	F5 Although this usually refreshes the model, if you are focused on code (for example, using Edit Code), this shortcut performs a roundtrip.
Remove from View.	Delete Note that sometimes the Delete key deletes the element from the model, depending on the diagram context (for example, statecharts).
Removes the selected (clicked) element from the selection.	Select + Shift
Resize box to fit contained.	Ctrl+E
Resize without contained elements (assuming you are using a mouse).	Use the Alt key while stretching the shape.
Scale from the center of the element (instead of stretching).	Scale + Ctrl
Scale to Fit.	F6
Select All.	Ctrl+A
Select next shape (by proximity).	Ctrl+Alt+N
Undo Zoom.	Shift+F6

Code Editor Accelerators

When you are using the built-in Rhapsody code editor, you can use not only the predefined accelerators, but an extended set of accelerator keys.

Open the Properties dialog for the code editor and click the **Keyboard** tab to view the complete list of commands supported by accelerator keys, and to extend this list.



Useful Rhapsody Windows Shortcuts

The following table lists some of the more common and useful Windows keyboard shortcuts available in Rhapsody.

Action	Shortcut
Close the currently active window.	Ctrl+F4 If you hold down Ctrl+F4 , all the Rhapsody windows close in succession. Note that this method will not work if you have a diagram that needs to be saved (for example, an animated SD)—so using this “close all” method does not put your unsaved work at risk.
Enable or disable check boxes.	Space key
Get to the Windows menu.	Alt+Space
Invoke IntelliVisor when editing code or names of graphic elements.	Ctrl+Space
Navigate between open diagrams.	Ctrl+Shift+Tab
Navigate between boxes when in a dialog box.	Tab To do the same in reverse order, use Shift+Tab .
Navigate between tabs when in a dialog box (for example, the Features dialog box).	Ctrl+Tab To do the same in reverse order, use Ctrl+Shift+Tab .
Navigate to items in lists (such as the list of element types in the Search/Replace dialog box) or in the browser.	Type its name on the keyboard.
Navigate within the browser.	Use the up and down keys to move between nodes. Use the left and right arrow keys to expand or collapse tree nodes.
Navigate within the Properties tab for any item.	Up and down arrow keys
Toggle between all open windows inside the application (diagrams, code windows, and so on).	Ctrl+Tab To do the same in reverse order, use Ctrl+Shift+Tab .

In addition, whenever an item is selected (such as a node in the browser or a class in a graphic editor), you can use the standard Windows keyboard key designated to activate the pop-up menu of the selected item. The following figure highlights this button (but the location may vary, depending on your keyboard).

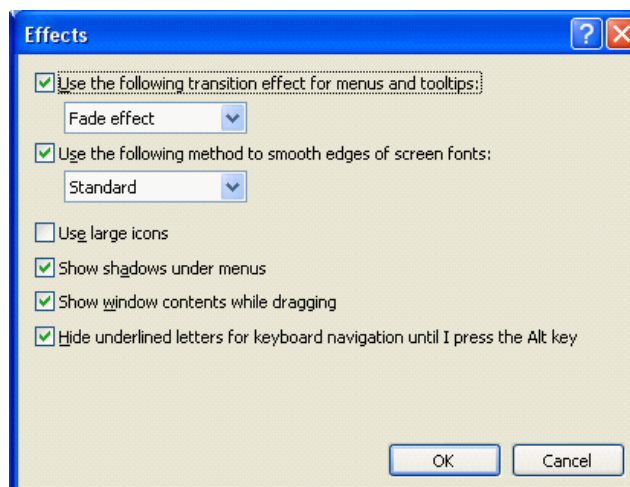


The resultant menu can also be used with mnemonics.

Changing Settings to Show the Mnemonic Underlining

To expose mnemonic underlining in Rhapsody menus, follow these steps:

1. On your desktop, right-click and select **Properties** from the pop-up menu. The Display Properties dialog box opens.
2. Select the **Appearance** tab.
3. Click **Effects**. The Effects dialog box opens, as shown in the following figure.



4. Disable the last option to show underlined letters for keyboard navigation.
5. Click **OK** to dismiss the Effects dialog box.
6. Click **OK** to dismiss the Display Properties dialog box.

Technical Support and Documentation

Rhapsody software continues to be supported by the Telelogic support group for customers who licensed Rhapsody before November 1, 2008. If you are one of these “heritage” customers, you may use the familiar support procedures described in [Contacting Telelogic Rhapsody Support](#).

After November 1, 2008, all Telelogic Rhapsody customers also have access to the IBM technical support team and resources, as described in [Contacting IBM Rational Software Support](#).

Contacting Telelogic Rhapsody Support

The Telelogic Rhapsody technical support group assists heritage Rhapsody customers through the online problem report form, automatically generated problem reports, direct contact by [Calling Telelogic Rhapsody Technical Support](#) or contacting them through IBM’s Support site, as described in [Contacting IBM Rational Software Support](#).

Note

Assistance from the technical support staff for purchased products is only available to companies that have paid for ongoing maintenance.

Accessing the Automated Problem Report Form

Follow these steps to send the automated problem report to the technical support staff:

1. In Rhapsody click **Help** on the menu bar.
2. Select the **Generate Support Request** option from the menu.
3. The following form appears with some of your product information filled in.

Generate Support Request

Problem Details
Please complete below with as much detail as possible to describe your issue:

Impact: My work is not affected. I want an answer to a question.

Summary:

Problem:

Rhapsody Information

Version: 7.3
Build: 987717
Serial No: T10-293399
Language: C++
Edition: Development Edition

Rhapsody Window Snapshot...

System Information

Version: Windows XP
Service pack: Service Pack 2
Build: 2600

Screen Snapshot...

Attachment Information (Double-click Item to View)

Add Video Capture...
Add System Details
Add Product Files
Add File(s)...
Remove...
Remove All...

Item Description:

Help Just Text (No Email)...
Cancel Preview and Send

4. Check the product information to verify it is accurate.
5. From the **Impact** drop-down list box, select the severity of the problem.
6. In the **Summary** box, summarize the problem.

7. In the **Problem** box, type a detailed description of the problem.
8. If available, attach a snapshot. Click the **Rhapsody Window Snapshot** or **Screen Snapshot** button, whichever is applicable, and select the snapshot wherever you have it on your machine.
9. If possible, add the model, active component, files, and/or a video capture by using the buttons in the **Attachment Information** area.
10. Add any additional items or information to help the Technical Support staff resolve the problem.
11. Click **Preview and Send** to submit the report.

The problem report is recorded in the Rhapsody case tracking system and put into a queue to be assigned to a support representative. This representative works with you to be certain that your problem is solved.

Automatically Generated Problem Reports

If your Rhapsody system crashes, it displays a message asking if you want to send a problem report to Rhapsody technical support about this crash.

If you select to send the report, the system displays the same online form that is available from **Help > Generate Support Request**. However, this form contains information about the crash condition in addition the information that is usually filled in describing your system.

Add any more information that you can to help the support staff identify the problem and then click **Preview and Send** to submit the report.

Calling Telelogic Rhapsody Technical Support

If your company has a current Rhapsody maintenance agreement purchased before November 1, 2008, you may call the Telelogic Technical Support staff directly using the appropriate telephone number for your region listed below.

Support Location	Telephone Number	Availability
United States	(800) 577-8449	8:30 a.m. to 8:00 p.m. EST
Europe	00800 57784499	8:00 a.m. to 5:00 p.m. GMT
Europe (alternative number)	+353 1 2090154	8:00 a.m. to 5:00 p.m. GMT

Contacting IBM Rational Software Support

The IBM Rational Software Support team provides the following resources for your assistance:

- ◆ For contact information and guidelines or reference materials that you need for support, read the [IBM Software Support Handbook](#).
- ◆ For FAQs, lists of known problems and fixes, documentation, and other support information, visit the [Telelogic Rhapsody Support Site](#).
- ◆ Voice support is available to all current contract holders by dialing a telephone number in your country (where available). For specific country phone numbers, go to <http://www.ibm.com/planetwide/>.

Before you contact IBM Rational Software Support, gather the background information that you will need to describe your problem. When describing a problem to an IBM software support specialist, be as specific as possible and include all relevant background information so that the specialist can help you solve the problem efficiently. To save time, know the answers to these questions:

- ◆ What software versions were you running when the problem occurred?
- ◆ Do you have logs, traces, or messages that are related to the problem?
- ◆ Can you reproduce the problem? If so, what steps do you take to reproduce it?
- ◆ Is there a work-around for the problem? If so, be prepared to describe the work-around.

For Rational software product news, events, and other information, visit the [IBM Rational Software Web site](#).

Accessing the Rhapsody Documentation

Rhapsody documentation is accessible from three locations:

- ◆ On the [Telelogic Rhapsody Support Site](#)
- ◆ With Rhapsody installed on your computer, select Windows **Start** and the **Programs** menu
- ◆ Open Rhapsody and select the **Help** menu

The documentation is available in two formats: PDF and online. The online documentation displays in a browser from the Help Topics menu option. The PDF files display from the **List of Books** using the Adobe[®] Acrobat Reader[™].

Note

To display these files properly, you must have version 6.0 or greater of this free PDF reader. To download the most up-to-date version of the Acrobat Reader, go to <http://www.adobe.com/products/acrobat/readstep2.html>.

Rhapsody Glossary

abstract class

A class that cannot be directly instantiated, but whose descendants can have instances. Contrast with [concrete class](#).

abstract operation

An operation defined, but not implemented, by an abstract class. The operation must be implemented by all concrete descendant classes.

abstraction

Selecting the essential or common characteristics of an entity that distinguish it from all other kinds of entities. An abstraction defines a boundary relative to the perspective of the viewer.

action

The specification of an executable statement that forms an abstraction of a computational procedure. An action typically results in a change in the state of the system, and can be realized by sending a message to an object or modifying a link or value of an attribute.

action sequence

An expression that resolves to a sequence of actions.

action state

A state that represents the execution of an atomic action, typically the invocation of an operation.

activation

The execution of an action.

active class

A class whose instances are active objects.

active concurrency

The system runs in a distributed environment with many threads. Each active object runs on its own thread. Active objects are also known as tasks.

active object

An instance of an active class. An active object owns its own thread and can initiate control activity.

In Rhapsody, active objects are graphically portrayed with thicker borders.

activity

An operation in dynamic modeling that takes time to complete. Activities are associated with states and represent real-world accomplishments.

activity diagram

A special case of a state machine used to model processes involving one or more classifiers that involve behavior that is not event-driven. Compare to [statechart](#).

actor

A coherent set of roles that users of use cases play when interacting with these use cases. An actor has one role for each use case with which it communicates.

An actor is an external object that interacts with a use case of the system. In use case diagrams (UCDs), actors are drawn as stick figures and can populate both use case diagrams and object model diagrams (OMDs).

actual parameter

A synonym for argument.

aggregate

A class that represents the “whole” in an aggregation relationship.

aggregation

A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part.

The [Unified Modeling Language \(UML\)](#) symbol for an aggregation relationship is a line with a hollow diamond at the end attached to the aggregate class.

analysis

The software development activity for studying and formulating a model of a problem domain. Analysis focuses on what is to be done; design focuses on how to do it.

analysis time

Refers to something that occurs during an analysis phase of the software development process. See also [design time](#) and [modeling time](#).

ancestor class

A class that is a direct or indirect superclass of a given class.

and state

An orthogonal state.

animation

The act of executing an animated model. During an animation session, Rhapsody highlights the current states of execution using animated diagrams and views.

Animation is not the same as simulation. The Rhapsody animator actually runs the real application on the host machine or, if desired, on the target machine. This gives you a better idea of the system's actual behavior than merely simulating it.

API

The Rhapsody Application Program Interface (API) allows you to write applications that access and manipulate Rhapsody model elements. It facilitates reading, changing, adding to, and deleting from all model elements that are available in the Rhapsody [browser](#). Refer to the *Rhapsody API Reference Manual* for more information.

architecture

The organizational structure and associated behavior of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components, and subsystems.

architecture framework

An architecture framework is a specification of how to organize and present an enterprise architecture. It provides a means to present and analyze the enterprises problems. It does not generally tell you how to do something. Architecture frameworks tend to consist of a standard set of viewpoints that represent different aspects of an organization's business as it relates to a

particular objective. In the context of MODAF, this implies a systems of systems approach as the analysis is complex and wide-ranging.

archive

A repository grouping that a configuration management (CM) tool creates to keep track of different versions of a project's configuration items. An archive can be a file or directory, depending on the requirements of the CM system.

argument

A binding for a parameter that resolves to a run-time instance. A synonym for argument is actual parameter. Compare to [parameter](#).

artifact

A piece of information used or produced by a software development process. An artifact can be a model, a description, or software. Synonym: *product*.

association

Defines a semantic relationship between two or more classifiers that specify connections among their instances. It represents a set of connections between the objects (or users).

An association has at least two ends, each of which is connected to an object. The same object can be connected to both ends of an association.

An association can be bidirectional, in which the connected objects know about each other, or directed, in which only one of the objects knows of the other.

An association defines a relationship among instances of two or more classes describing a group of links with common structure and common semantics.

association class

A model element that has both association and class properties. An association class can be seen as an association that also has class properties, or as a class that also has association properties.

association end

The endpoint of an association, which connects the association to a classifier.

attribute

A feature within a classifier that describes a range of values that instances of the classifier can hold.

An attribute consists of three parts:

- ◆ A data member
- ◆ An accessor (`get`) operation for the data member
- ◆ A mutator (`set`) operation for the data member

Attributes are displayed in the object box using the following format:

```
<attribute_name>:<type>
```

Contrast with [part](#).

Attribute breakpoint condition

Interrupts a running application when any of the object's member attributes changes value. Copies of all attribute values are stored as a reference when you set the breakpoint. When any value changes with respect to the reference, a break occurs. After the break, the latest values are again stored as a new reference.

automatic transition

An unlabeled transition in dynamic modeling that automatically fires when the activity associated with the source state is completed.

AUTOSAR

The AUTOSAR (AUTomotive Open System ARchitecture) standard provides development and design guidelines and diagrams to streamline and standardize component modeling in the automotive industry. Rhapsody's AUTOSAR profiles define a new project to use these AUTOSAR standard-compliant diagrams:

- ◆ ECU diagram
- ◆ Internal Behavior diagram
- ◆ SW Component diagram
- ◆ System diagram
- ◆ Topology diagram

base-aware comparison

The Rhapsody [DiffMerge tool](#) compares two versions of a unit with a baseline (common ancestor) version of the unit.

base class

A class from which other classes can inherit data and member functions.

batch transformation

A sequential input-to-output transformation in which inputs are supplied at the start and the goal is to compute an answer. There is no ongoing interaction with the outside world.

behavior

The observable effects of an operation or event, including its results.

behavioral feature

A dynamic feature of a model element, such as an operation or method.

behavioral inheritance

A mechanism by which more specific elements incorporate behavior of more general elements related by behavior.

behavioral model aspect

A model aspect that emphasizes the behavior of the instances in a system, including their methods, collaborations, and state histories.

binary association

An association between two classes. A special case of an [n-ary association](#).

binding

The creation of a model element from a template by supplying arguments for the parameters of the template.

black-box analysis

This type of system analysis defines the system structure and identifies the large-scale organizational pieces of the system. It can show the flow of information between system components and the interface definition through ports. In large systems, the components are often decomposed into functions or subsystems. Basically, it shows the system's interaction with the outside world. Using Rhapsody, you may create activity diagrams, sequence diagrams, and statecharts to communicate this analysis. A [white-box analysis](#) shows a system's internal and external operations and relationships.

block

In [SysML](#), a block is a [class](#). In Rhapsody it is represented as an [object](#). Blocks are most often used in systems engineering designs.

Boolean

An enumeration whose values are true and false.

Boolean expression

An expression that evaluates to a Boolean value.

breakpoint

A useful debugging tool. During animation, breakpoints enable you to inspect data values and the states of various objects in the system at the time of the break.

You can set a breakpoint on object's instance or on the object itself. If you set the breakpoint on an object, it sets the breakpoint on all the object's instances.

browser

Provides an overview of your entire model using an expandable tree structure.

call

An action state that invokes an operation on a classifier.

call operation node

Represents a call to an operation of a classifier.

canceled timeout

Timeouts are set to wait for something to happen. If the something happens, the timeout is canceled. If it does not happen, the object resumes its operation, perhaps with an error recovery process.

For example, a telephone emits a dial tone while waiting for you to dial. If you dial, the dial tone is canceled. If you do not dial, the dial tone changes to a repeating beep. Canceled timeouts are labeled $CanTm(n)$, where n is the length of the time during which the timeout can be canceled.

cardinality

The number of elements in a set. Contrast with [multiplicity](#).

categories mode

Specifies that metatype nodes are displayed in the browser for all metatypes, such as classes, packages, and operations, in hierarchical arrangement by ownership.

For example, under the Objects metatype for a package, all objects belonging to that package are listed. Each object or object_type, in turn, can be expanded into categories for Attributes, Operations, and Relations belonging to that object or object type.

child

In a generalization relationship, the specialization of another element, the parent. See also [subclass](#), [subtype](#). Contrast with [parent](#).

class

In object-oriented languages such as C++ and Java, a class is a template for the creation of instances (objects) that share the same attributes, operations, methods, relationships, and semantics. A class can use a set of interfaces to specify collections of operations it provides to its environment. See also [interface](#).

In the Rhapsody in C implementation, classes are replaced by object types, which are generated into structures. Like classes, object types function as templates in the creation of objects of explicit type. Although object types differ fundamentally from classes, from the perspective of the Rhapsody GUI, they are handled almost identically. Thus, for documentation purposes, any GUI functionality ascribed to classes applies almost equally as well to object types.

In addition to object_types, Rhapsody in C supports objects of implicit type. All objects in C, whether objects of implicit or explicit type, are known as objects. Objects are similar to instances in the Rhapsody C++ implementation, with the exception that they have a separate identity independent of object types.

For example, objects are listed in their own category under a package in the browser. Instances, by contrast, appear in the browser under the class that defines it—and only during execution of the model's application. Because an object is a permanent instance with class-like behavior, in many cases, the GUI functionality ascribed to C++ classes also applies to C objects.

class attribute

An attribute whose value is common to a class of objects, rather than a value peculiar to each instance.

class diagram

A diagram that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.

class interface element

Operations, events, and event receptions.

class name

Identifies the class. If you do not explicitly assign a class to a package, Rhapsody assigns the class to the default package of the diagram. To assign the class to a different package, use the format `<package>::<class>` for the name. If this box is also an instance, use the format `<instance>:<class>` for the class name.

class template

Specifies individual classes constructed using parameterized types. When class templates are instantiated into template classes, types used in the class are provided as arguments.

The following example declares the class template `MyTemplateClass`:

```
template<class T> class MyTemplateClass {
    T* data
    public:
        T& func()
};
```

The class template `MyTemplateClass` can then be used to instantiate a template class and an object as shown by the following example:

```
MyTemplateClass<int> int_object
```

In this example, `MyTemplateClass<int>` is a template class in which the type `int` replaces `T` in the class template definition. It is then used to instantiate the object `int_object`.

Rhapsody allows you to create or change an existing class into a class template or instantiate it into a template class in the Class dialog box.

See also [template class](#).

classification

The assignment of an object to a classifier. See also [dynamic classification](#), [multiple classification](#), and [static classification](#).

classifier

A mechanism that describes behavioral and structural features. Classifiers include interfaces, classes, data types, and components.

cleanup operation

Cleans up an instance that is no longer needed. It replaces the concept of a destructor operation. Instead of cleaning up class instances, a cleanup operation cleans up C objects.

For example, an object can call a cleanup operation to free a dynamically allocated pointer. In sequence diagrams, a cleanup operation is represented as a dotted red arrow from the destroyer object to the object being destroyed. Cleanup lines can either be horizontal or point back to the originating object. Cleanup lines are not labeled.

client

A classifier that requests a service from another classifier. Contrast with [supplier](#).

CM

Acronym for [configuration management](#).

code frame

Refers to code generated from OMDs only, without the behavioral input of statecharts and activity diagrams.

collaboration

The specification of how an operation or classifier, such as a use case, is realized by a set of classifiers and associations playing specific roles used in a specific way. The term collaboration defines this interaction. See also [interaction](#).

collaboration diagram

A diagram that shows interactions organized around the structure of a model, using either classifiers and associations or instances and links. Unlike a sequence diagram, a collaboration diagram shows the relationships among the instances. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways. See also [sequence diagram](#).

comment

An annotation attached to an element or a collection of elements. A note has no semantics. Contrast with [constraint](#).

compile time

Refers to something that occurs during the compilation of a software module. See also [modeling time](#), [run time](#).

component

A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation

of a system, including software code (source, binary, or executable) or equivalents such as scripts or command files.

The role of the component is important in the modeling of large systems that are comprised of several libraries and executables. For example, the Rhapsody application itself is comprised of several dozen components including the graphic editors, browser, code generator, and animator, all provided in the form of a library.

component diagram

A diagram that shows the organizations and dependencies among components.

component file

Contains elements for a [component](#) and can be saved as a [unit](#). In Rhapsody 7.2 or greater, this type of file is now called a [SourceArtifact](#).

composite aggregation

A synonym for composition.

composite class

A class related to one or more classes by a composition relationship.

composite object

An object that contains one or more other objects, typically by storing references to those objects in its instance variables.

composite state

A state that consists of either concurrent (orthogonal) substates or sequential (disjoint) substates. See also [subpackage](#).

composition

A form of aggregation association with strong ownership and coincident lifetime as part of the whole. Put another way, composition is a strong form of aggregation in which the lifetime of the whole determines that of its parts. Parts with non-fixed multiplicity can be created after the composite itself, but once created, they live and die with it (they share lifetimes). Such parts can also be explicitly removed before the death of the composite. Composition can be recursive. A synonym for composition is composite aggregation.

The UML symbol for a composition relationship is a line with a filled diamond at the end attached to the composite class. In Rhapsody, you draw classes as boxes inside the composite class, rather than using relation lines.

If you want the component class to come into being and die with the composite class as a whole, use composition. If, however, you want the parts to have lifetimes of their own separate from that of the whole, use aggregation.

Note the following:

- ◆ By definition, an object that contains another object is a reactive object.
- ◆ Composition in Rhapsody in C is a form of containment of one object by another. An object that contains another is said to be a *composite object*.

compound state

A state that contains other nested states.

concrete class

A class that can be directly instantiated. Contrast with [abstract class](#).

concurrency

The occurrence of two or more activities during the same time interval. The activities are said to be concurrent. Concurrency can be achieved by interleaving or simultaneously executing two or more threads.

concurrent

Two or more tasks, activities, or events are said to be concurrent when their executions overlap in time.

concurrent substate

A substate that can be held simultaneously with other substates contained in the same composite state. Contrast with [disjoint substate](#).

condition

A Boolean function in dynamic modeling of object values valid over an interval of time.

condition connector

A way of visually representing an if-then-else condition. It splits a single transition in a statechart or activity diagram into several branch transitions with guards enclosed in square brackets labeling the branches. Whichever guard is true determines to which element the object will branch. The following apply to condition connectors and branches:

- ◆ A condition connector can have only one entering transition.
- ◆ It can branch to any number of "if" conditions, but only one else condition.

- ◆ Branching segments can be nested. That is, a transition exiting a condition connector can enter another condition connector.
- ◆ Branches entering a condition connector can contain triggers. Labels for transitions into a condition use the following standard format:

```
trigger [guard] /action
```

- ◆ Branches exiting a condition connector cannot contain triggers. Labels for transitions into a condition use the following standard format:

```
[guard] /action
```

condition mark

A hexagon located on an instance line in a sequence diagram. A condition mark indicates that the object is in a certain condition or state. The name of the condition often corresponds to a state name in the object's statechart.

configuration

Defines the construction of a built component. For example, the configuration determines the target environment of the component and whether it is instrumented. It specifies which checks Rhapsody should perform before generating code, which link and compiler switches to use, any additional libraries, sources, and headers to include in the compilation, and which initial instances to create in the main program loop. The configuration also allows you to add custom initialization code to the `main()` function.

configuration item

A unit of collaboration that developers can exchange among themselves. See also [unit](#).

configuration management

The process of managing a set of classes and other resources selected for compilation including version control. This is usually managed by software system that requires "check in" and "check out" procedures for changes to any files managed in the system. Clearcase and MKS Integrity are both configuration management (CM) systems.

constraint

A semantic condition or restriction. Certain constraints are predefined in the UML, whereas others are user-defined. Constraints are one of three extensibility mechanisms in the UML. See also [tagged value](#), [stereotype](#).

In general, UML constraints add semantic information to model elements, which represent requirements, invariants, and so on. Constraints can be specified in a natural language, constraint language (such as OCL), or programming language.

A constraint is a model element owned by some other model element. In Rhapsody, only model elements that have specification dialog boxes can own constraints. These include packages, classifiers, operations, attributes, states, and transitions

Note that the object owning the constraint is *not* necessarily the object to which the constraint applies. By default, if there is no “applies to relation” (via a dependency), the constraint applies to the owner object. This is an optimization, because this would be the common, default case in which there is no need to apply a dependency relation.

A constraint applies to one or more model elements. A model element can have more than a single constraint applied to it.

construct

Previously created items, such as files and classes, that can be added to a model or design.

constructor

Called when an object is instantiated. An object can use a constructor to explicitly initialize object members or dynamically allocate space for member pointers.

In sequence diagrams, a constructor is represented as a dotted green arrow from the creator object or system border to the object being created. Constructor lines are horizontal.

Convert and copy constructors can have arguments.

In Rhapsody in C, the tasks of constructors are performed by initializers for the class-like objects and `object_types`.

container

Can be either of the following:

- ◆ An instance that exists to contain other instances, and provides operations to access or iterate over its contents. (for example, arrays, lists, and sets).
- ◆ A component that exists to contain other components.

containment hierarchy

A namespace hierarchy consisting of model elements and the containment relationships that exist between them. A containment hierarchy forms a graph.

context

A view of a set of related modeling elements for a particular purpose, such as specifying an operation.

continuous transformation

A system in which the output actively depends on changing inputs and must be updated periodically.

contract

A contract is a specification of a set of interfaces between elements, which may be either offered (provided), required, or both. The contract includes the set of operations and events that constitute those interfaces and features. The important features of the interfaces are the parameters and their types, return values, and conditions. A [port](#) is a named connection point for a class and a typed interface (contract).

control

The aspect of a system that describes the sequences of operations that occur in response to stimuli.

control flow

A Boolean value that affects whether a process is executed.

controlled files

Files produced in other programs, such as Word or Excel, that are added to a project for reference purposes and then controlled through Rhapsody.

data type

A descriptor of a set of values that lack identity and whose operations do not have side effects. Data types include primitive, predefined types and user-defined types. Predefined types include numbers, string, and time. User-defined types include enumerations.

deep transition

A transition from a parent statechart into a nested statechart.

default transition

A transition to the default state of the object. A default transition can have neither a trigger nor a guard. It can, however, have an action and load to a condition connector, after which there may be guards.

delegation

The ability of an object to issue a message to another object in response to a message. Delegation can be used as an alternative to inheritance. Contrast with [inheritance](#).

dependency

A relationship between modeling elements in which a change to one modeling element (the independent element) affects the other modeling element (the dependent element).

In a dependency between elements, the implementation or functioning of one element requires the presence of another element. Thus, dependency is a directed relationship. For example, an object might require the definition of another object to compile, similar to when a client element requires the presence and knowledge of a server element. Stereotypes are used to denote different types of dependency.

dependent unit

Configuration items that reference another configuration item.

deployment diagram

A diagram that shows the configuration of run-time processing nodes and the components, processes, and objects that live on them. Components represent run-time manifestations of code units. See also [component diagram](#).

derived association

An association defined in terms of other associations.

derived attribute

An attribute computed from other attributes.

derived class

A class that inherits data and member functions from a base class (subclass).

derived element

A model element that can be computed from another element, but that is shown for clarity or included for design purposes even though it adds no semantic information.

derived scope

Rhapsody decides which instances to include in the configuration based on the objects you select in the **Initial Instances** box. Use the **Derived Scope** option if you are not sure which objects to include in the compilation scope.

descendant class

A class that is a direct or indirect subclass of a given class.

design

The part of the software development process whose primary purpose is to decide how the system will be implemented. It is the software development activity during which strategic and tactical solution decisions necessary for meeting client functionality and quality requirements are made. Analysis focuses on what is to be done; design focuses on how to do it.

design time

Refers to something that occurs during a design phase of the software development process. See [modeling time](#). Contrast with [analysis time](#).

destructor

An operation that cleans up an existing instance of a class (an object) that is no longer needed.

For example, an object can call a destructor to free a dynamically allocated pointer. In sequence diagrams, a destructor is represented as a dotted red arrow from the destroyer object to the object being destroyed. Destructor lines can either be horizontal or point back to the originating object. Destructor lines are not labeled.

In Rhapsody in C, the tasks of destructors are performed by cleanups for the class-like objects and `object_types`.

development process

A set of partially ordered steps performed for a given purpose during software development, such as constructing models or implementing models.

diagram

A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements).

The UML supports the following types of diagrams:

- ◆ Activity diagram
- ◆ Collaboration diagram
- ◆ Component diagram
- ◆ Deployment diagram
- ◆ Object model diagram
- ◆ Sequence diagram
- ◆ Statechart
- ◆ Use case diagram

diagram connector

Joins two segments of a statechart. It allows you to separate different segments of the chart onto different pages to make it easier to read. Matching labels on the source and target diagram connectors define the jump from one diagram to the next. A statechart can have several source diagram connectors with the same label, but only one destination connector of each label. Diagram connectors should have either input transitions or a single outgoing transition.

DiffMerge tool

The Rhapsody DiffMerge tool supports team collaboration by showing how a design has changed between revisions and then merging units as needed. It performs a full comparison including graphical elements, text, and code differences.

This tool can be operated inside and/or outside your CM software to access the units in an archive. The [unit](#) only need to be stored as separate files in directories and accessible from the PC running the DiffMerge tool. For more information about this tool, refer to the *Rhapsody Team Collaboration Guide*.

direct instance

An object that is an instance of a class, but not an instance of any subclass of the class.

directed association

A unidirectional relationship between objects (and users). Only the source of the directed association (client user) knows about the target (the server). The target of the relation can receive messages without knowing their source.

directional relation

A relation that exists in one direction (for example, from source to target), but not in the other.

disjoint substate

A substate that cannot be held simultaneously with other substates contained in the same composite state. Contrast with [concurrent substate](#).

distribution unit

A set of objects or components allocated to a process or a processor as a group. A distribution unit can be represented by a run-time composite or an aggregate.

DoDAF

U.S. Department of Defense Architectural Framework (DoDAF) provides an industry standard for diagrams and notations used for developing DoDAF-compliant architecture models.

domain

An area of knowledge or activity characterized by a set of concepts and terminology used by practitioners in that specific area of knowledge or activity.

dynamic classification

A semantic variation of generalization in which an object might change its classifier. Contrast with [static classification](#).

dynamic model

A description of aspects of a system concerned with control, including time, sequencing of operations, and interaction of objects.

dynamic simulation

A system that models or tracks objects in the real world.

element

An atomic constituent of a model. See also [primary model elements](#).

element name

Specifies the name of the element. As a benefit of hierarchical arrangement of model elements, primary model elements can be entered/identified by a path name in the following format:

```
<ns1>::<ns2>::...::<nsn>::<name>
```

In this format, <ns> can be the name of a package or an object. For example, object *x* in package *P* can be entered for a search or specified in an OMD as *P::X*.

enterprise architecture

Enterprise architecture is the practice of applying a comprehensive and rigorous method for describing a current and/or future structure and behavior for an organization's processes, information systems, personnel, and organization sub-units, so that they align with the organization's core goals and strategic direction. It is effectively a structured approach to describing how a business works or is intended to work so that it can reach its primary

objectives. Enterprise architecture is used typically by the military for capability procurement, by governments, and by large businesses.

entry action

An action executed upon entering a state in a state machine, regardless of the transition taken to reach that state.

enumeration

A list of named values used as the range of a particular attribute type. For example, `RGBColor = {red, green, blue}`. `Boolean` is a predefined enumeration with values from the set `{false, true}`.

event

The specification of a significant occurrence that has a location in time and space.

Statecharts use events to describe the behavior of objects. In that context, an event is an instantaneous occurrence that can trigger a state transition in a class.

The use of events facilitates asynchronous collaborations. In other words, objects that generate events do not need to wait for a response before continuing with their next task.

Because events do not exist as in any programming language, they can be implemented in a variety of ways.

event attribute

Because an event is a type of object, it has its own data elements called event attributes.

Event Received breakpoint condition

Interrupts a running application when the object receives an event.

event reception

Represents an object's ability to react to an event. In other words, the event can be a trigger in that object's statechart.

Event Sent breakpoint condition

Interrupts a running application when the object sends an event to another object.

executable

A file, application, or program that can perform operations when launched. In Rhapsody, executable components have a file extension defined in the

<lang>_CG::<Environment>::ExeExtension property. A common executable extension is “.exe.”

exit action

An action executed upon exiting a state in a state machine, regardless of the transition taken to exit that state.

explicit scope

In **Configuration Settings**, explicit scope means that you explicitly select which packages and objects are used for compiling an application.

export

In the context of packages, to export is to make an element visible outside its enclosing namespace. See also [visibility](#). Contrast with [import](#).

expression

A string that evaluates to a value of a particular type. For example, the expression (7 + 5 * 3) evaluates to a value of type number.

extend relationship

A relationship from an extension use case to a base use case that specifies how the behavior defined for the extension use case augments (subject to conditions specified in the extension) the behavior defined for the base use case. The behavior is inserted at the location defined by the extension point in the base use case. The base use case does not depend on performing the behavior of the extension use case. See also [extension points](#).

extension points

Aspects of a use case that allow it to be extended in the future. To extend a use case means to inherit one use case from another. Use cases are extended by means of «Usage» or «Extends» stereotypes. In a «Usage» relationship, the sub-use case depends on the behaviors provided by the super-use case. In an «Extends» relationship, the subuse case adds its own behaviors to those of the super-use case.

facade

A stereotyped package containing only references to model elements owned by another package. It is used to provide a “public view” of some of the contents of a package.

feature

A property like an operation or attribute that is encapsulated within a classifier, such as an interface, a class, or a data type.

A feature is a modifiable item in a tabbed dialog box. Related features that appear on the same tab in a dialog box are internally stored in tab groups.

file

Placeholders for generated logical units (such as packages, classes, and so on) and verbatim code segments. Files exist only within the context of a component.

file diagram

A file diagram shows how files interact with one another. Typically, a file diagram shows how the `#include` structure is created. A file diagram provides a graphical representation of the system structure. The Rhapsody code generator directly translates the elements and relationships modeled in a file diagram into C source code.

final state

A special kind of state signifying that the enclosing composite state or entire state machine is completed.

fire

Execute a state transition.

fixed-point support

Rhapsody in C supports variables (fixed points) that represent non-integral values. The user can define the word size and precision of the variable.

flat mode

Specifies that category nodes are not displayed in the browser for most metatypes. Instead, they are displayed for all first and second-level metatypes—root node, components, diagrams, and packages. Within these categories, however, all items are displayed alphabetically without being subdivided according to metatype.

flat statechart

Specifies that states are implemented as simple, enumeration-type variables.

When statecharts are inherited, the implementation is duplicated from the base class. This strategy is more effective with shallow statechart inheritance hierarchies.

floating-point support

Rhapsody in C supports constants, variables, or expressions that evaluate to an integer or floating-point number.

flow chart

A flow chart is a schematic representation of an algorithm or a process. In UML and Rhapsody, you can think of a flow chart as a subset of an activity diagram that is defined on methods and functions. For more information, see [Flow Charts](#).

flow ports

Flowports allow you to represent the flow of data between [blocks](#) in an [object model diagram \(OMD\)](#), without defining events and operations. Flowports can be added to blocks and classes in object model diagrams. They allow you to update an [attribute](#) in one object automatically when an attribute of the same type in another object changes its value.

focus of control

A symbol on a sequence diagram that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure.

folder

File subdirectories that appear in the browser under the Files category for a component. They are set to contain files that are compiled together to build the component. When you assign (map) generated model elements to a folder, you are telling Rhapsody to place the generated files for the specified elements in that subdirectory of the project. Elements that are not assigned to any folder are generated in the configuration directory.

formal parameter

A synonym for *parameter*.

framework

There are two definitions:

- ◆ A stereotyped package consisting mainly of patterns
- ◆ An architectural pattern that provides an extensible template for applications within a specific domain

function template

Specifies individual functions constructed using parameterized types that are supplied as input parameters when the function is instantiated into a template function.

In the following C++ example, a function template is defined for the global function swap:

```
template <class T> void swap (T& x, T&y) {  
    T temp = x;  
    x=y;  
    y = temp;  
}
```

Function templates are called the same way ordinary functions are called. This action generates a template function such as the following:

```
int i=22, j=66;  
swap(i,j);
```

Rhapsody allows you to define global function templates through the mechanism for defining global functions.

generalizable element

A model element that can participate in a generalization relationship.

generalization

A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element can be used where the more general element is allowed. See also [inheritance](#).

global object

In C, all top-level objects are global across the project. Nested objects only have local scope relative to the object in which they are nested.

Got Control breakpoint condition

Interrupts a running application when the object gets control. An object gets control when it starts executing a member operation, responds to an event, or when an operation that the object has called on another object finishes and the object resumes its own execution.

Note

Do not enter any information in the **Data** box of the Define Breakpoint dialog box for this condition.

gravity distance

The minimum space, in points, that must always remain between adjacent graphic elements.

guard condition

A condition that must be satisfied in order to enable an associated transition to fire. It is a Boolean expression in dynamic modeling that must be true for a transition to occur.

Harmony process

Systems engineering steps, a standard, iterative workflow, and SysML diagrams used to simplify the communication among all of the participating groups in a design project. See the [Harmony Process](#) diagram for a high-level view of this process.

helper applications

Custom programs that you attach to Rhapsody to extend its functionality. They can be either external programs (executables) or Visual Basic for Applications (VBA) macros that typically use the Rhapsody COM API. They connect to a Rhapsody object via the `GetObject()` COM service.

history connector

Recalls the most recent active configuration of a state and its substates. Each state can have only one history connector. A history connector transitively restores all the subconfigurations that originated in the state.

A transition originating in a history connector denotes the history default. The history default transition is taken if no history existed prior to entry into the history connector.

host machine

For a node-locked license, the host machine is the machine on which Rhapsody is running. For a floating license, the host machine is the machine on which the license server is running.

identity

A distinguishing characteristic of an object that denotes a separate existence of the object, even though the object might have the same data values as another object.

implementation

A definition of how something is constructed or computed. For example, a class is an implementation of a type; a method is an implementation of an operation.

implementation inheritance

The inheritance of the implementation of a more specific element. It includes inheritance of the interface. Contrast with [interface inheritance](#).

implementation method

A style that implements specific computations on fully specified arguments, but does not make context-sensitive decisions.

import

In the context of packages, import is a dependency that shows the packages whose classes can be referenced within a given package (including packages recursively embedded within it). Contrast with [export](#).

include relationship

A relationship from a base use case to an inclusion use case, specifying how the behavior for the base use case contains the behavior of the inclusion use case. The behavior is included at the location defined in the base use case. The base use case depends on performing the behavior of the inclusion use case, but not on its structure (that is, attributes or operations). See also [extend relationship](#).

inherent concurrency

Two objects that can receive events at the same time without interacting have inherent concurrency.

inheritance

The derivation of one class from one or more other classes. The derived class inherits the same data members and behaviors present in the parent class. It is the mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior. See also [generalization](#).

initial instance

An instance of a class that typically bootstraps the system. To configure a system, this class must be instantiated first.

initializer

In Rhapsody in C, a non-object-oriented language, an initializer is called when an object is instantiated much like a constructor is called when a class is instantiated into an object. A Rhapsody in C object or object_type can specify an initializer to explicitly initialize object members or dynamically allocate space for member pointers.

In sequence diagrams, an initializer is represented as a dotted, green arrow from the creator object or system border to the object being created. Constructor lines are horizontal.

instance

An entity to which a set of operations can be applied and that has a state that stores the effects of the operations.

An object described by a class.

Instance Created breakpoint condition

Interrupts a running application when the specified class or a subclass of it is instantiated.

For these breakpoints, specify the instance name without an instance number. For example, you can use the instance name `Heater`, but not the instance name `Heater[0]`.

Instance Deleted breakpoint condition

Interrupts a running application when an object is deleted.

instance diagram

An object diagram that describes how a particular set of object instances relate to each other.

instance line

A vertical timeline in a sequence diagram that shows the sequence of messages that an object processes and states that it enters over its lifetime.

instantiation

For classes, this process denotes of creation of objects (instances) from classes. For templates, this process denotes the creation of a template class from a class template or a template function from a function template.

interaction

A specification of how stimuli are sent between instances to perform a specific task. The interaction is defined in the context of a collaboration. See also [collaboration](#).

interaction diagram

A generic term that applies to several types of diagrams that emphasize object interactions, including collaboration and sequence diagrams.

interaction occurrence

An interaction occurrence (or reference sequence diagram) enables you to refer to another sequence diagram from within a sequence diagram. This allows you to break down complex scenarios into smaller scenarios that can be reused. Each such scenario is an *interaction*.

interface

A named set of operations that characterize the behavior of an object.

interface inheritance

The inheritance of the interface of a more specific element. It does not include inheritance of the implementation. Contrast with [implementation inheritance](#).

internal transition

A transition signifying a response to an event without changing the state of its object.

junction connector

Joins more than one transition into a single, outgoing transition. This enables you to combine several segments into a single, graphical description or use a common transition suffix.

layer

The organization of classifiers or packages at the same level of abstraction. A layer represents a horizontal slice through an architecture, whereas a partition represents a vertical slice.

leaf state

A state without `And` state components and descendants.

library

A library component has a `.lib` extension.

link

A semantic connection among multiple objects. It is an instance of an association.

link attribute

A named data value held by each link in an association.

link end

An instance of an association end.

Lost Control breakpoint condition

Interrupts a running application when the object loses control. An object loses control when it finishes executing an operation, finishes responding to an event, or calls an operation on another object.

Note: Do not enter any information in the **Data** box of the Define Breakpoint dialog box for this condition.

message

A specification of the conveyance of information from one instance to another, with the expectation that activity will ensue. A message can raise a signal or call an operation.

message diagram

Message diagrams, available in the FunctionalC profile, show how the files functionality may interact through messaging (through synchronous function calls or asynchronous communication). Message diagrams can be used at different levels of abstraction. At higher levels of abstractions, message diagrams show the interactions between actors, use cases, and objects. At lower levels of abstraction and for implementation, message diagrams show the communication between classes and objects.

Message diagrams have an executable aspect and are a key animation tool. When you animate a model, Rhapsody dynamically builds message diagrams that record the object-to-object messaging.

Rhapsody message diagrams are based on [Sequence Diagrams](#). For more information about the FunctionalC profile, see [Profiles](#).

message passing

The means by which objects communicate with one another to provide information, send information, and invoke actions. Sending messages is how work gets done in an object-oriented system.

message-to-self

An arrow that bends back onto the originating instance line. The arrow can be on either side of the instance line. If the message is a primitive operation, it immediately folds back (operations are synchronous). If it is an event, it can fold back sometime later (events are asynchronous).

metaclass

A class whose instances are classes—a class of classes. In Rhapsody, metaclasses are used to define properties for whole classes of objects. For example, under the subject of code generation (CG), there are metaclasses for `Class`, `Component`, and `Configuration`.

metamodel

A model that defines the language for expressing a model.

meta-metamodel

A model that defines the language for expressing a metamodel. The relationship between a meta-metamodel and a metamodel is analogous to the relationship between a metamodel and a model.

metaobject

A generic term for all metaentities in a metamodeling language. For example, metatypes, metaclasses, metaattributes, and metaassociations.

method

The implementation of an operation. It specifies the algorithm or procedure associated with an operation.

methodology

A process for the organized production of software using a collection of predefined and notational conventions.

MODAF

United Kingdom's Ministry of Defence Architectural Framework (MODAF) provides an industry standard for diagrams and notations used for developing MODAF-compliant architecture models. This standard builds on the U.S. [DoDAF](#) standard.

mode

The access permission of a unit of collaboration. You can change a read/write (RW) unit, but you cannot change a read-only (RO) unit. An item is "locked" if it is RW for you and RO for others.

model

An abstraction of something for the purpose of understanding it before building it. The model is the overall database of information about your project. Different projects can use the same model. You perform requirements modeling using UCDs, static object modeling using OMDs, and dynamic modeling using sequence diagrams and statecharts.

model aspect

A dimension of modeling that emphasizes particular qualities of the metamodel. For example, the structural model aspect emphasizes the structural qualities of the metamodel.

Model-driven Development (MDD)

This development environment allows designers to visualize a domain, system, product, or process and create that design in diagrams within a model. Then the modeling tool generates implementation artifacts for the design for a selected target, such C++ code.

model elaboration

The process of generating a repository type from a published model. It includes the generation of interfaces and implementations that allows repositories to be instantiated and populated based on, and in compliance with, the model elaborated.

model-view controller (MVC)

An architecture that separates the model and its views by allowing certain model elements to be viewed in different views or not at all.

modeling time

Refers to something that occurs during a modeling phase of the software development process. It includes analysis time and design time. When discussing object systems, it is often important to distinguish between modeling-time and run-time concerns.

module

A software unit of storage and manipulation. Modules include source code modules, binary code modules, and executable code modules. See also [component](#).

multiple classification

A semantic variation of generalization in which an object can belong directly to more than one classifier. See [static classification](#) and [dynamic classification](#).

multiple inheritance

A semantic variation of generalization in which a type can have more than one supertype. Contrast with [single inheritance](#).

multiplicity

Refers to the number of instances of one class that can relate to a single instance of an associated class.

n-ary association

An association among three or more classes. Each instance of the association is an *n*-tuple of values from the respective classes. Contrast with [binary association](#).

namespace

A part of the model in which the names can be defined and used. Within a namespace, each name has a unique meaning.

nested unit

An element that is inside an element of the same [unit](#) type. For example, a package may contain another package (or a nested unit).

node

A classifier that represents a run-time computational resource, which generally has at least a memory, and often processing, capability. Run-time objects and components can reside on nodes.

non-public inheritance

The subclass inherits only the public attributes and operations of the superclass.

null transition

A transition with a guard and an action, but no trigger. Null transitions can be useful when you want to allocate a resource that might not be available. In this case, you might branch based on some entry action or join transition.

object

An entity with a well-defined boundary and identity that encapsulates state and behavior. *State* is represented by attributes and relationships, whereas *behavior* is represented by operations, methods, and state machines.

An object is an instance of a class.

In Rhapsody in C, an object consists of a C `struct` that is mapped, via naming conventions, to operations (functions). This unifies attributes (data) and operations (functions) under one element, a C object, which appears as a primary element in both the browser and OMD. A Rhapsody in C object can be defined as an object of implicit type or an object of explicit type.

object box

In an OMD, a rectangle with three compartments (unless it is a simple object box). The first compartment contains the object name, the second contains the attributes, and the third contains operations.

The symbol for a simple object is simply a rectangle without compartments. An object and simple object are semantically equivalent.

object design

A stage of the development cycle during which the implementation of each class, association, attribute, and operation is determined.

object diagram

A diagram that encompasses objects and their relationships at a point in time. An object diagram can be considered a special case of a class diagram or a collaboration diagram. See [class diagram](#) and [collaboration diagram](#).

object execution framework (OXF)

Rhapsody has one central run-time library, the OXF, that provides all the services required by the generated code in the running application. All the other libraries are related to animation or tracing in one respect or another. See also [framework](#).

object flow state

A state in an activity graph that represents the passing of an object from the output of actions in one state to the input of actions in another state.

object group

Consists of several objects that are examined together as a unit for the purpose of comparing two sequence diagrams. This enables you to compare, for example, a black box diagram showing messages to and from the system as a whole to a gray or white box diagram showing messages to and from the individual parts. For example, given a model of an oven consisting of `Oven`, `Timer`, and `Display` objects, you can create both a black box sequence diagram showing an `Oven` object and a white-box sequence diagram showing objects of all three.

Without object groups, you could compare only the `Oven` objects in the two diagrams, omitting from the comparison any messages to or from the `Timer` and `Display`. Using an object group, you can compare messages to and from the `Oven` in the black box diagram to those to and from the `Oven`, `Timer`, and `Display` in the white box diagram because they are considered to be a single unit. This results in fewer discrepancies and simplifies the comparison.

object lifeline

A line in a sequence diagram that represents the existence of an object over a period of time. See also [sequence diagram](#).

object model

A description of the structure of the objects in a system including their identity, relationships to other objects, attributes, and operations.

object model diagram (OMD)

Show the logical views of a system. OMDs are static, structural diagrams that show the parts of a software system and the relationships that exist between them. These parts can include classes (in C, objects and object types), packages, and actors.

OMDs include objects that appear in related scenarios, which are described using sequence diagrams. OMDs show all relationships, including inheritance, dependencies, compositions, and associations between collaborating objects.

OMDs are constructive in that Rhapsody generates code based on what you draw in them.

object modeling technique (OMT)

An object-oriented development methodology that uses object, dynamic, and functional models throughout the development life cycle.

object of explicit type

In Rhapsody in C, an object of explicit type is defined by a single reference to an object type that defines the object. If an object A is based on an object type B, this is expressed as “object A is of type B.” The name of the object A appears in an OMD as A : B.

Objects of both implicit and explicit types show all or part of their object type-related properties as part of their object appearance. However, objects of explicit type cannot alter or add directly to their object type-related properties. In other words, an object of explicit type is explicitly and completely defined by its object type.

Objects are set as implicit or explicit type through the **Is Of Type** check box in the Object dialog box. If, for a particular object, this box is checked and an object type is specified in the adjacent box, the object is an explicit object. Otherwise, the object is an implicit object.

You can convert objects of explicit type to objects of implicit type through the **Create Type-less Object** feature. This option copies all properties (functions, statechart, and so on) of the object’s object type to a new object of implicit type.

object of implicit type

In Rhapsody in C, objects of implicit type are defined only by the attributes, operations, objects, and object types that they contain. You can display these elements in the box representation of the implicit object base.

Both implicit and explicit objects show all or part of their object type-related properties as part of their object appearance.

You can convert an implicit object to an explicit object and object type through the **Expose Object Type** feature.

object-oriented

A software development strategy that organizes software as a collection of objects that contain both data structure and behavior.

object type

Rhapsody in C uses object types as templates for defining objects. For example, if `A` is an object type, object `B` can be defined as an object of type `A`, also referred to as `B:A`.

operation

A service that can be requested from an object to affect behavior. An operation has a signature, which might restrict the actual parameters that are possible.

Operation breakpoint condition

Interrupts a running application when the object starts executing a member operation.

Operation Returned breakpoint condition

Interrupts a running application when a member operation of the object returns.

origin class

The top-most class that defines an attribute or operation.

orthogonal state

Two or more independent states that an object can be in at the same time. This is also known as an `And` state. For example, a clock radio can be counting the time and playing music at the same time, states that can be completely independent of each other. Dotted lines separate the compartments of orthogonal states in an `And` state.

override

Replace an inherited method for the same operation, or replace the default value of a property with a new value.

package

A general-purpose mechanism for organizing elements into groups. You can think of a system as a single, high-level package, with everything else in the system contained in it. A package is a collection of packages, classes (in C, objects and object types), events, diagrams, globals, types, use cases, and actors.

Because packages can be nested with other packages, they enable you to partition a system into smaller subsystems. Thus, package nesting provides a way to organize large projects into package hierarchies.

You can import packages into your project from other projects using **File > Add to Model**.

panel diagram

A panel diagram provides you with a number of graphical control elements that you can use as a graphical user interface (GUI) to monitor and regulate an application. Each control element can be bound to a model element (variable/attribute/event/state). During animation, you can use the animated panel diagram to monitor (read) and regulate (change values/send events) your user application.

Panel diagrams are intended only for simulating and prototyping, and not for use as a production interface for the user application. In addition, panel diagrams can only be “used” on the host and can be “used” only from within Rhapsody

parameter

The specification of a variable that can be changed, passed, or returned. A parameter can include a name, type, and direction. Parameters are used for operations, messages, and events. See also [formal parameter](#). Contrast with [argument](#).

parameterized element

The descriptor for a class with one or more unbound parameters. Also referred to as a [template](#).

parent

In a generalization relationship, the generalization of another element (the child). See also [subclass](#), [subtype](#). Contrast with [child](#).

part

A part is a role that an instance of that class plays in a context. Meaning it is a stand-in for an object.

Note that attributes, when used by value, will be generated like a part. However, if its type is reactive, then the attribute would be missing the reactive code (startbehavior call, and so forth).

Use a part (rather than an attribute) if it is to be used as an stand-in for an object. The advantage of using a part is that it can be drawn on a diagram, you can connect parts using links, and also view ports, attributes, operations, and so forth. See also [class](#), [package](#), and [component](#). Contrast with [attribute](#).

participation

The connection of a model element to a relationship. For example, a class participates in an association, whereas an actor participates in a use case.

pattern

A template collaboration.

partition

There are two possible definitions:

- ◆ A portion of an activity graph that organizes the responsibilities for actions. See also [swimlane](#).
- ◆ A set of related classifiers or packages at the same level of abstraction or across layers in a layered architecture. A partition represents a vertical slice through an architecture, whereas a layer represents a horizontal slice.

partition line

A red, horizontal line dividing a sequence diagram into vertical segments indicating different phases of the sequence. The text note that accompanies a partition line is initially positioned at the left end of the line. You can move the note anywhere in the diagram, but it always remains associated with the same partition line.

periodic actions

To model periodic actions that repeat as long as a state is active, you can use a transition that loops back to the same state with a timer as a trigger. For example, to define an action that should repeat once every second while an object is in some state, set the trigger on the loop transition to `tm(1000)` and set the action on entry to the statements to be executed at that interval.

persistent object

An object that exists after the process or thread that created it has ceased to exist.

physical system

Either of the following:

- ◆ The subject of a model.
- ◆ A collection of connected physical units, which can include software, hardware and people, that are organized to accomplish a specific purpose. A physical system can be described by one or more models, possibly from different viewpoints. Contrast with [system](#).

pinned dialog box

In “pinned” mode, the information displayed in the “pinned” Features dialog box remains displayed and accessible from all of the dialog box tabs even when a different element is selected. This allows multiple dialog boxes to be displayed and the contents reviewed together. The pin icon in the upper right corner of the dialog box controls this feature.

plug-in

A Rhapsody plug-in is a user Java application that Rhapsody loads into its process. While a plug-in is loaded, Rhapsody supports it with an interface to the hosting Rhapsody application.

Use a plug-in to extend Rhapsody's features with customized functionality by adding menus to Rhapsody's IDE or reacting to Rhapsody's events such as code generation and model check.

polymorphism

A way of simplifying communication between objects by hiding different implementations behind a common interface. For example, defining multiple print methods behind one print command.

port

As in the standard [Unified Modeling Language \(UML\)](#), a port is a named connection point for a [class](#), [object](#), or a [block](#). It is a distinct interaction point between a class and its environment or between (the behavior of) a class and its internal parts. A port allows you to specify classes that are independent of the environment in which they are embedded. The internal parts of the class can be completely isolated from the environment and vice versa. A port can have either of these interfaces: required or provided.

postcondition

A constraint that must be true at the completion of an operation.

precondition

A constraint that must be true when an operation is invoked.

predefined types

The Rhapsody predefined types include `int`, `char`, `double`, `void`, `OMBoolean`, `long`, `OMString`, and `void*`. They do not belong to the `Default` or any other user-defined package. Rather, they belong to the `PredefinedTypes` package, which is part of every model, albeit hidden (in the `PredefinedTypes.sbs` file in the `Share\Properties` directory). Thus, clashes are prevented between user-defined and predefined types with the same name.

primary model elements

Primary model elements within the browser are packages, classes, OMDs, associations, dependencies, operations, variables, events, event receptions, triggered operations, constructors, destructors, and types. Primary model elements in OMDs are packages, classes, associations (links), dependencies, and actors.

Rhapsody in C and C++ classes and their instances are replaced by C equivalent object types and objects, respectively. Similarly, class constructors and destructors are replaced by initializers and cleanup operations.

primitive operation

An operation whose body you write yourself. Rhapsody automatically generates bodies for all other types of operations.

primitive type

A predefined, basic data type without any substructure, such as an integer or a string.

private

When referring to an attribute or operation of a class, private means accessible only by methods of the current class.

process

Has the following meanings:

- ◆ A heavyweight unit of concurrency and execution in an operating system. Contrast this with thread, which includes heavyweight and lightweight processes. If necessary, an implementation distinction can be made using stereotypes.
- ◆ A software development process; the steps and guidelines by which to develop a system.
- ◆ To execute an algorithm or otherwise handle something dynamically.

profile

A special kind of package that is distinguished from other packages in the browser. You specify a profile at the top level of the model. Therefore, a profile is owned by the project and affects the entire model. By default, all profiles apply to all packages available in the workspace, so their tags and stereotypes are available everywhere. A profile is similar to any other package; however, profiles cannot be nested. You may select a specific profile when you create a project and add profiles to existing projects, and Rhapsody automatically adds profiles required for add-on products. See [Using Profiles](#) for more information.

project

Overall set of artifacts describing the system being modeled across all developers. The terms project and [model](#) are interchangeable in Rhapsody.

projection

A mapping from a set to a subset of it.

property

A named value denoting a characteristic of an element. A property has semantic impact. The UML predefines some properties; others are user-defined. The definitions of Rhapsody properties are displayed in the **Properties** tab of the Features dialog box.

See also [tagged value](#).

protected object

In a system with several threads of control, there can be contentions where different threads apply messages to the same passive object. A protected object serves only one message at a time. In other words, an object applying a message to a protected object will be blocked until the protected object processes the message.

provided event

An event to which a class responds. It is defined with the class or its superclass.

pseudo-state

A vertex in a state machine that has the form of a state, but does not behave as a state. Pseudo-states include initial and history vertices.

public

Accessible by methods of any class.

public inheritance

The subclass inherits all attributes and operations of the superclass.

qualifier

An association attribute or tuple of attributes whose values partition the set of objects related to an object across an association.

qualified association

An association that relates two classes and a qualifier; a binary association in which the first part is a composite comprising a class and a qualifier, and the second part is a class.

reactive object

A class (in C, an object or object type) is reactive if:

- ◆ It has a statechart.
- ◆ It is a composite.
- ◆ It has an event reception.

Most reactive objects and object types have statecharts to define their behavior. The presence of a statechart for an object or object type is indicated by the appearance of a small overlaid mini-statechart icon in the browser, OMD, and sequence diagrams.

read-only mode

You can view an item, but cannot modify it.

real-time model

Runs the instrumented application in quasi-real time in which timeouts and delays are computed based on the system clock.

read/write mode

You can modify an item.

receive a message

The handling of a stimulus passed from a sender instance. See also [sender object](#), [receiver object](#).

receiver object

The object handling a stimulus passed from a sender object. Contrast with [sender object](#).

reception

A declaration that a classifier is prepared to react to the receipt of a signal.

reference

Has the following definitions:

- ◆ A denotation of a model element

- ◆ A named slot within a classifier that facilitates navigation to other classifiers

Also called a pointer.

reference class

Classes that are included in the model by reference only, without specifying any of their internal details, such as attributes. For example, you can include the MFC classes for reference, merely to show that they are acting as superclasses or peer classes to other classes in your model, without including any specific information about the MFC classes themselves.

refinement

A relationship that represents a fuller specification of something that has already been specified at a certain level of detail. For example, a design class is a refinement of an analysis class.

relation

Elements can be related to each other in different ways. The relationships that can exist between software entities are modeled on relationships that exist in the real world, as follows:

- ◆ Association
- ◆ Directed association
- ◆ Dependency

Relation breakpoint condition

Interrupts a running application when the object modifies a relation in any way. In other words, the application breaks if the object connects to, disconnects from, or clears a relation.

Relation Cleared breakpoint condition

Interrupts a running application when the object clears a relation. To clear a relation means to disconnect from all instances on the relation and clear the relation itself.

Relation Connected breakpoint condition

Interrupts a running application when the object connects to a relation.

Relation Disconnected breakpoint condition

Interrupts a running application when the object disconnects from a relation. To disconnect from a relation means to disconnect from one individual instance on the relation. Connections to other instances on the same relation remain intact.

relationship

A semantic connection among model elements. Examples of relationships include associations and generalizations.

repository

A facility for storing object models, interfaces, and implementations. Repository files are in ASCII format for easy storage and differentiation by a [CM](#) system.

The repository can also be defined as a subdirectory inside the project directory containing all configuration items in the project, such as diagram, class, and package files. The directory name consists of the project name followed by `_rpy`.

required event

An event that a class knows how to generate. It is not defined with the class, but with the target of the event.

requirement

A desired feature, property, or behavior of a system.

respect mode

In C++ projects, Rhapsody preserves (respects) the changes to the model and other information changes through subsequent code generations.

responsibility

A contract or obligation of a classifier. It defines what the system expects from an object.

reusable statechart

With reusable statechart implementation, states are implemented as class types derived from the abstract state classes defined in the implementation framework. When statecharts are inherited, states are reused by instantiating the same state object from a base class. This strategy is more effective with deep statechart inheritance hierarchies.

reuse

The use of a pre-existing artifact.

role

The named specific behavior of an entity participating in a particular context. A role can be static (for example, an association end) or dynamic (for example, a collaboration role).

In the case of an association, a role specifies how each object relates to the object at the other end of the association.

root state

The default state of a statechart.

roundtrip

The action taken to update a Rhapsody model based on changes made to code previously generated by Rhapsody.

run time

The period of time during which a computer program executes. Contrast with [modeling time](#).

SCC

Abbreviation for the Microsoft Common Source Code Control. Refer to the *Rhapsody Team Collaboration Guide* for more information.

scenario

A specific sequence of actions that illustrates behaviors. A scenario can be used to illustrate an interaction or the execution of a use case instance. See also [interaction](#).

semantic variation point

A point of variation in the semantics of a metamodel. It provides an intentional degree of freedom for the interpretation of the metamodel semantics.

send a message

The passing of a stimulus from a sender instance to a receiver instance. See also [sender object](#), [receiver object](#).

sender object

The object passing a stimulus to a receiver object.

sequence diagram

A diagram that shows object interactions arranged in time sequence. In particular, it shows the objects participating in the interaction and the sequence of messages exchanged between them. Vertical lines represent the objects and arrows are drawn to represent the messages.

Unlike a collaboration diagram, a sequence diagram includes time sequences but does not include object relationships. A sequence diagram can exist in a generic form (describing all

possible scenarios) and in an instance form (describing one actual scenario). Sequence diagrams and collaboration diagrams express similar information, but show it in different ways.

In addition to drawing sequence diagrams, Rhapsody also creates animated sequence diagrams from your running application so you can see that your code is really working the way you want it to.

sequential concurrency

The system runs on a single thread and all operations are executed in sequential order. A sequential object either exists within a composite, or can be a global instance.

Service Oriented Architecture (SOA)

SOA projects are business-centric and facilitate an effective IT infrastructure to help streamline and improve processes across the enterprise. See [Domain-specific Projects and the NetCentric Profile](#) for more information.

signal

The specification of an asynchronous stimulus communicated between instances. Signals can have parameters.

signature

The name and parameters of a behavioral feature. A signature can include an optional, returned parameter.

simulated time model

A virtual timer orders timeouts and delays, which are posted when the system completes a computation.

single inheritance

A semantic variation of generalization in which a type can have only one supertype. Contrast with [multiple inheritance](#).

singleton

An object type that, by definition, has exactly one instance. It is designated by a small “1” in the upper, right corner of the object type box in an OMD.

SourceArtifact

A Rhapsody element that contains code respect information for reverse engineering and roundtripping.

Note that previous to Rhapsody version 7.2, a SourceArtifact was referred to as a component file. While component files still exist, they now refer to elements under the **Components** category in Rhapsody. When component files are located under packages or classes, and so forth, they are referred to as SourceArtifacts.

specification

A declarative description of what something is or does. Contrast with [implementation](#).

spontaneous transition

A transition in dynamic modeling that fires only if a guard condition is true.

state

An abstraction of the mode in which the object finds itself. It is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

Messages, events, timeouts, and the satisfaction of conditions can cause an object to transition from one state to another. Statecharts define an object's behavior by specifying messages, events, timeouts, and conditions that must occur to cause the object to transition from one state to another.

state awareness

Rhapsody shows the configuration management "state" if elements in the [browser](#) with icons for the elements. The icons provide the following configuration management information about the individual items:

- ◆ Under source control
- ◆ Not under source control
- ◆ Checked out
- ◆ Checked in with changes
- ◆ Deleted from source control

State breakpoint condition

Interrupts a running application when the object changes state.

State Exited breakpoint condition

Interrupts a running application when the object exits a state.

state machine

A behavior that specifies the sequences of states that an object or an interaction goes through during its lifetime in response to events, together with its responses and actions. In Rhapsody, the state machine is diagrammed in a [statechart](#).

statechart

Defines the behavior of reactive objects by specifying how they react to events or operations. The reaction can be to perform a transition between states and possibly to execute some actions. When running in animation mode, Rhapsody highlights the transition taken and the entered state to show transitions between states.

statechart diagram

A diagram that shows a state machine.

static classification

A semantic variation of generalization in which an object cannot change classifier. Contrast with [dynamic classification](#).

static structure

Used to describe an OMD. An OMD shows the relationship between instances at any given time during a system's operation. Therefore, the structure displayed by an OMD is timeless, or static.

stereotype

A type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. Stereotypes can extend the semantics, but not the structure of pre-existing types and classes. Certain stereotypes are predefined in the UML, whereas others are user-defined. Stereotypes are one of three extensibility mechanisms in UML. See also [constraint](#) and [tagged value](#).

Stereotypes allow extension of the UML metamodel by “typing” different UML entities. Entities that allow stereotypes in Rhapsody are use cases, packages, classes, (objects and object types in Rhapsody in C) and dependencies. See [Defining Stereotypes](#) for more information.

stimulus

The passing of information from one instance to another, such as raising a signal or invoking an operation. The receipt of a signal is normally considered an event. See also [message](#).

string

A sequence of text characters. The details of string representation depend on implementation, and can include character sets that support international characters and graphics.

structural feature

A static feature of a model element, such as an attribute.

structural model aspect

The model aspect that emphasizes the structure of the objects in a system, including their types, classes, relationships, attributes, and operations.

subactivity state

A state in an activity graph that represents the execution of a non-atomic sequence of steps that has some duration.

subclass

In a generalization relationship, the specialization of another class; the superclass. See also [generalization](#). Contrast with [superclass](#).

submachine

A statechart that is a decomposition of a containing state referred to as a submachine state. Viewing the submachine state in submachine view displays the contents of the submachine state, referred to as a submachine.

submachine state

A state in a state machine that is equivalent to a composite state, but whose contents are described by another state machine.

A submachine state is a composite state that is decomposed to a submachine. The submachine state is in the parent statechart.

subpackage

A package that is contained in another package.

substate

A state that is part of a composite state. See also [concurrent substate](#) and [disjoint substate](#).

subsystem

A grouping of model elements that represents a behavioral unit in a physical system. A subsystem offers interfaces and has operations. In addition, the model elements of a subsystem can be partitioned into specification and realization elements. See also [package](#) and [physical system](#).

subtype

In a generalization relationship, the specialization of another type; the supertype.

subunit

An element that is nested inside another element. For example, a package B that is nested inside a package A is considered a subunit of A .

superclass

In a generalization relationship, the generalization of another class; the subclass. See also [generalization](#). Contrast with [subclass](#).

A superclass is a base class from which another class derives. A superclass is created when you draw an inheritance relation from one class to another in an OMD.

supertype

In a generalization relationship, the generalization of another type, the [subtype](#).

supplier

A classifier that provides services that can be invoked by others. Contrast with [client](#).

swimlane

A partition on an activity diagram for organizing the responsibilities for actions. Swimlanes typically correspond to organizational units in a business model. See also [partition](#).

symmetric relation

Both entities know about and can send messages to each other. Also known as a bidirectional association.

synch state

A vertex in a state machine used for synchronizing the concurrent regions of a state machine.

SysML

Systems Modeling Language (SysML) is a domain specific modeling language for systems engineering applications. It supports the specification, analysis, design, verification and validation of a broad range of systems and systems-of-systems. These systems may include hardware, software, information, processes, personnel, and facilities. SysML's goal is to allow systems engineers to define their designs with precision, conciseness, consistency and understandability and to be able to test the designs for correctness. See [Systems Engineering with Rhapsody](#) for more information.

system

A top-level subsystem in a model. Contrast with [physical system](#).

System Architect (SA)

System Architect® software provides the tools for you to build a Business and Enterprise Architecture—a fully integrated collection of models and documents across five keys domains: Strategy, Business, Information, Systems, and Technology. System Architect software creates a shared workspace for all team members to understand how to improve the company's architecture and overall business. DoDAF data can be imported from SA into Rhapsody as SysML. See [Importing System Architect's DoDAF Diagrams](#) for more information.

system border

Represents the environment outside the system under design. In a sequence diagram, the system border is indicated by a column of short diagonal lines. Messages that originate outside the system or subsystem shown in the sequence come from the system border.

tagged value

The explicit definition of a property as a name-value pair. In a tagged value, the name is referred to as the tag. Certain tags are predefined in the UML; others are user-defined. Tagged values are one of three extensibility mechanisms in UML. See also [constraint](#) and [stereotype](#).

target environment

Rhapsody is used to develop real-time application software to run in different operating system environments that rely on different software compilers.

Rhapsody distinguishes these environments based on a combination of compiler and operating system (some compilers can compile for several operating systems). Refer to the *Rhapsody ReadMe (release notes)* for information on the supported target environments.

template

A parameterized element. Rhapsody provides two types of templates: class and function.

See also [template instantiation](#) and [template parameters](#).

template class

The C++ language enables you to instantiate a class template into a template class using types passed to the class template as arguments. Rhapsody allows you to create or change an existing class into a class template or instantiate it into a template class in the class dialog box.

See also [class template](#).

template function

Generates a template function based on the data types passed to it as arguments.

Consider the following function template:

```
template <class T> void swap (T& x, T&y) {
    T temp = x;
    x=y;
    y = temp;
}
```

When the function template is called (the same way ordinary functions are called), a template function is generated, as follows:

```
int i=22, j=66;
swap(i,j);
```

template instantiation

Rhapsody provides C++ template instantiation in the following two cases:

- ◆ Class templates can be instantiated into template classes using the **Instantiation** radio button.
- ◆ Function templates are instantiated into template functions, using the **Is Template** check box on the global function dialog box.

In Rhapsody, the following features and limitations apply to class template instantiation:

- ◆ A new class is declared as a class template or instantiated template in the browser.
- ◆ An instantiation of a template must be full—all template parameters must have values.

Template instantiation is a named instantiation interpreted as a “typedef” of a real instantiation. The following examples demonstrate the Rhapsody generated code for an instantiated class template:

```
typedef vector<int> vec_int_inst;  
typedef MyTemplateClass<int> MySimpleIntInstance;
```

In the examples, the class `vec_int_inst` is equivalent to the class `vector<Cstring>` and the class `MySimpleIntInstance` is equivalent to the class `MyTemplateClass<int>`.

- ◆ Operations and attributes cannot be added to an instantiation.
- ◆ A model element can be a template (by having template parameters), an instantiation (by having instantiation parameters), or neither one.
- ◆ See also [template parameters](#) and [template specialization](#).

template parameters

When templates are instantiated into template classes or template functions, the types used in the defined body are provided as parameters. In Rhapsody, the following features apply to template parameters and their scope:

- ◆ A template can define some of its parameters with default values.
- ◆ Template parameters can be class or metaclass, as shown by the following examples:

```
template<int> class x;  
template<class T> class x;
```
- ◆ Inside a template object, metaclass template parameters are treated as regular types/classes. Consider the following class template definition:

```
template<class T> class x;
```
- ◆ In this template, `x` can have an attribute of type `T`.
- ◆ Inside a template object, template parameters are treated as expressions for the purpose of instantiation. For example:

```
template<class T> class x {  
    vector<T> a;  
}
```
- ◆ In this example, `T` can be a metaclass or type parameter.
- ◆ The name of a metaclass can be used as a class of another parameter. For example:

```
template <class T, T t> class C {};
```
- ◆ This can be done only in the browser, not in the OMD.
- ◆ When adding a superclass, metaclass template parameters are available. For example:

```
template <class T> class C : public T {...};
```
- ◆ This can be done only in the browser, not in the OMD.

See also [template instantiation](#) and [template specialization](#).

template specialization

Allows specific instantiations to override the content of the general template.

For member operations, set the value of the `Specialization` property for the operation to the text of the instance (for example, `vector<CString> . . .`). For functions, the only difference between the original template function and the specialized function is that the return type, arguments, and so on. of the specialization are instantiations with the type chosen for specialization.

Consider the template global function, f :

1. Define $f<T>$ and in it write the general body.
2. In the same scope, define the specialized function (for example, $f<int>\{ . . . \}$) and in it write the specialized body.

See also [template instantiation](#) and [template parameters](#).

termination breakpoint condition

Interrupts a running application when an object reaches a termination connector in its statechart. The application does not break if the object is deleted in any other way than by entering a termination connector.

termination connector

“Suicide” or “self-destruct” connectors used in statecharts. A transition to a termination connector causes an object to delete itself.

Although termination connectors have the same appearance as termination states in activity diagrams, termination states do not destroy the object they are in.

A termination connector cannot have any outgoing transition.

termination state

Signals an exit from the process specified by the [activity diagram](#) or [statechart](#).

In statecharts, termination states do not destroy the object they are in.

thread of control

A single path of execution through a program, dynamic model, or some other representation of control flow.

In addition, a stereotype for the implementation of an active object as a lightweight process. See also [process](#).

time event

An event that denotes the time elapsed since the current state was entered. See also [event](#).

time expression

An expression that resolves to an absolute or relative value of time.

time interval

A vertical annotation that shows how much (real) time should pass between two points on an instance line in a sequence diagram. The label is free text and is not limited to a number or unit of any kind.

timeout

Used in sequence diagrams when an object should wait a set amount of time before continuing its operation. A timeout is shown as a message-to-self on an instance line with a small box at the start end of the message line. Timeouts are labeled $T_m(n)$, where n is the length of the timeout. Normal timeouts such as these cannot be canceled and must run to completion.

timing mark

A denotation for the time at which an event or message occurs. It is used in sequence diagrams to show how much real time passes between two points in a scenario.

tooltips

Text displayed when the cursor is moved over an interface element. For example, the full path name of an element is displayed in a tooltip when the cursor is over that element.

trace

A dependency that indicates a historical or process relationship between two elements that represent the same concept, without specific rules for deriving one from the other.

tracing

Displays trace messages during program execution.

transient object

An object that exists only during the execution of the process or thread that created it.

transition

A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified

conditions are satisfied. On such a change of state (first state to second state), the transition is said to *fire*.

A transition can have a trigger, a guard, and an action, which is formatted in the transition label as follows:

```
trigger [guard] /action
```

Transitions usually consist of at least a trigger and an action. A trigger can be either an event or a triggered operation. A transition can be qualified by a guard condition, meaning that the guard must be true for the transition to be taken.

transition context

The scope in which message data (parameters) are visible. Any guard and action can inherit the context of a transition determining the parameters that can be referenced within it.

trigger[guard]/action

An expression attached to a transition that determines the following:

- ◆ What event will fire the transition (trigger)?
- ◆ Under what condition the transition will be allowed (guard)?
- ◆ What action code will be executed when the transition takes place (action)?

triggered operation

A synchronous communication between objects (between a client object and a server object). It has a body of operation defined by the statechart actions that it triggers, and can return a value.

A triggered operation is a cross between an operation and an event. It is invoked by another object to trigger a state transition and its body is executed in response to the transition taken. Because it is a synchronous event, the sending object waits for the execution of the triggered operation.

The body of a triggered operation is set in the statechart of the receiving object by the action language associated with a transition. Thus, the body of the same triggered operation can be different based on the state of the object when the triggered operation is invoked.

Because its activation is synchronous, an operation can return a value to the client object. To return a value from a triggered operation, use the `REPLY (VARIABLE)` macro inside the body of the triggered operation.

A statechart consumes one event at a time. Until an event is consumed, no subsequent event can be consumed. Because triggered operations inject events (and “wait” until they are consumed), two triggered operations must run sequentially.

type

A stereotype of class used to specify a domain of instances (objects) together with the operations applicable to the objects. A type cannot contain any methods. See also [class](#) and [instance](#). Contrast with [interface](#).

type expression

An expression that evaluates to a reference to one or more types.

Unified Modeling Language (UML)

The UML is a third-generation language for describing complex systems using models. This language allows system architects and engineers to work at a high level of abstraction and to communicate design intent effectively. Using UML models allows system simulation so that errors to be found and corrected early in the development process.

uninterpreted

A placeholder for types whose implementation is not specified by the UML. Every uninterpreted value has a corresponding string representation.

unit

A composite model element stored in its own file that you can check in and out of a CM system. A model element can be made into a unit as long as it can be saved as a separate file. Some elements that can be saved as units are the entire model, packages, classes (in C, objects and object types), any type of Rhapsody diagram, and components. The project, represented by the root node displayed in the browser, is always a unit.

The primary purpose of units is to support collaboration with other developers. You have explicit control over unit files and modification rights (RO versus RW), and you can check unit files in and out of a CM system. You may also compare units using the Rhapsody [DiffMerge tool](#).

unresolved reference

A reference between two configuration items that have become disconnected, possibly by one having been moved to a different workspace.

usage

A dependency in which one element (the *client*) requires the presence of another element (the *supplier*) for its correct functioning or implementation.

use case

The specification of a sequence of actions, including variants, that a system (or other entity) can perform by interacting with actors of the system.

A use case is one way in which a user, or external actor, might interact with a system. For example, the user of a railcar system might want to call a car. This main use of the system can be represented in a use case named “Call Car.” Each use case belongs to a package, as reflected in the browser.

A use case can be referred to from other packages, but can belong to only one package at a time. The same holds true for types.

use case diagram (UCD)

A diagram that shows the relationships among actors and use cases within a system.

UCDs describe a high-level functional analysis of the system. They enable users to specify major system requirements. For example, user requirements for a VCR system might be to be able to install and set it up, play back and record video cassettes, set the time, and store channels into memory. These main uses of the system can be represented by the use cases Installation&Setup, Playback, Record, SetTime, and AutoProgram. Use cases represent the externally visible behaviors, or functional aspects, of the system.

Actors are added to the UCD to show the external entities interacting with the system.

use case instance

The performance of a sequence of actions being specified in a use case. An instance of a use case.

use case model

A model that describes a system's functional requirements in terms of use cases.

user-defined type

A type defined by a user, not a predefined type. Types allow for meaningful interpretation of fixed-length bit sequences. Common predefined types include integer (int), char (characters), and float (floating-point).

utility

A stereotype that groups global variables and procedures in the form of a class declaration. The utility attributes and operations become global variables and global procedures, respectively. A utility is not a fundamental modeling construct, but a programming convenience.

value

An element of a type domain.

variable

A storage place within an object for a data element. The data element can be a data type such as a date or number, or a reference to another object.

VBA project

A VBA project is a file container for other files and components that you use in Visual Basic to build an application. After all the components have been assembled in a project and code written for it, you can compile the project into an executable file.

Each Rhapsody project is associated with a single VBA project that contains all the VBA artifacts (scripts, forms, and so on) you created within a Rhapsody project. This project file has the name `<project name>.vba`, located in the same directory with the Rhapsody project file (`<project>.rpy`). If present, this binary file is loaded with the Rhapsody project and saved when you press the **Save** button in Rhapsody or the VBA IDE.

vertex

A source or a target for a transition in a state machine. A vertex can be either a state or a pseudo-state. See [pseudo-state](#) and [state](#).

view

A projection of a model seen from a given perspective or vantage point that omits entities that are not relevant to this perspective. This could be a picture of existing model information consisting of a diagram or [statechart](#).

Different views display different sets of information. A class comprises the sum of its appearances in all views. The browser shows a complete view of a class, as does the generated code.

view element

A textual or graphical projection of a collection of model elements.

view projection

A projection of model elements onto view elements. A view projection provides a location and a style for each view element.

virtual base class

An indirect descendant inherits only one set of members from a virtual base class through multiple, intermediate inheritances.

virtual instance

An instance group named like a single instance, but actually containing several instances.

visibility

An enumeration whose value (public (+), protected (#), or private (-)) denotes how the model element it refers to can be seen outside its enclosing namespace.

Visual Basic for Applications

Visual Basic for Applications (VBA) is an OEM version of Microsoft Visual Basic integrated as an automation engine into the Microsoft Office family and ultimately intended for all Microsoft tools.

white-box analysis

This analysis decomposes the system's functions into subsystem components or the internal "logical subsystems." It describes how they relate to each other and to the outside world. A white-box analysis use case is different from a [black-box analysis](#) use case in that the black-box is used only to show the interactions with the outside world. The white box shows interactions both internal and external.

wizard

A small program or macro designed to perform repetitive or simple tasks automatically. Refer to the *Rhapsody Installation Guide* and see [Systems Engineering with Rhapsody](#) for examples of Rhapsody wizards.

workspace

A project in Rhapsody that contains a subset of the project artifacts on which a single user works.

Index

Symbols

- "And" line 766
- #define 1015
- #endif directive 960
- #if...#ifdef...#else...#endif 1016
- #ifndef directive 960
- #include structure 1486
- #pragma directive 960
- \$ImplName keyword 978
- %s character 138
- .clb files 284
- .cls files 284
- .cmp files 284
- .csv files 261
- .ctd files 284
- .dpd files 284
- .ehl file 285
- .hep files 494
- .msc files 284
- .omd files 284
- .rpw file 285
- .rpy file 284
- .sbs files 284
- .sdo file 717
- .ucd files 284
- .vba file 285
- _auto.rpy file 284
- _bak1.rpy file 284
- _bak2.rpy file 284
- _DEBUG, compiling with 960

Numerics

- 136531 813
- 64-bit targets 927

A

- Accelerator keys 1449, 1451, 1453, 1455
 - Ctrl-R for relations 232
- Accelerators
 - application 1451
 - for code editing 1455
 - in diagrams 1453
- AcceptChanges property 1066, 1069, 1072, 1212, 1214

- Access
 - privileges 276
- Accessor implementation file 950
- Accessors 79, 82
- Activity diagrams
 - flow of control 1254
- Acquisition viewpoint (MODAF) 1338
- Acrobat Reader 1463
 - version requirement 1463
- Action blocks 626, 661
 - creating 626, 661
 - creating subactivities from 628
- Action field 746, 1277
- Action flows
 - completion 666
 - default for flow chart 666
 - join for flow charts 668
 - loop for flow chart 667
- Action language 1275, 1277
 - arithmetic operations 1277
 - assignments 1277
 - basic syntax 1275
 - C code generation 985
 - checking 1278
 - example 1277
 - reserved words 1276
- Action on entry field 739
- Action on exit field 739
- Action pins 647, 649
 - adding to diagram 647
 - displayed in search 649
 - graphical characteristics 648
 - modifying features 648
- Action states 1257
 - defining 1277
- Actions 619, 621, 622, 623, 656, 658
 - block 619, 626, 656, 661
 - blocks 622
 - creating 623, 658
 - display options 626, 660
 - drawing 621, 658
 - expression 757
 - flow charts 658
 - showing attributes 626, 660
- Active
 - class 778

- component 874
- configuration 875
- project 5, 262, 263, 264
- state configuration 769
- thread 1099
- Active code view 26
 - code generation 926
 - edit code in 608
 - line numbers 933
 - show/hide 31
 - window 10, 26
- ActiveStackSize property 1099
- ActiveThreadName property 1099
- ActiveThreadPriority property 1099
- Activities 1227
 - subactivities 622
- Activity diagrams 65, 67, 432, 619, 1254
 - accept event action 623
 - action block 626
 - action blocks 622
 - action pins 623, 1257
 - action states 1257
 - action types 1254
 - actions 621, 622
 - activity flow 622
 - activity flows 633
 - advanced features 621
 - algorithm 619
 - browser icon 318
 - call behavior 622
 - call behaviors 633, 645
 - call operation 623, 1257
 - code for operations 651
 - code generation 651
 - code generation limitations 653
 - code generation scope 652
 - code generation values returned 652
 - condition connector 622
 - condition connectors 636
 - connectors 635
 - create new 573
 - creating 34, 622
 - creating in SysML 1255
 - decision points 619
 - default flow 622, 634
 - defining an action 1277
 - dependency 623, 1257
 - diagram connectors 622
 - draw fork sync bar 1256
 - drawing icons 622
 - generating sequence diagram from 1260
 - IntelliVisor information 483
 - join synchronization 1260
 - join transitions 622
 - junction connectors 622
 - limitations 653
 - link wizard 1237
 - loop activity flow 622
 - main behavior 619
 - mode 650, 741
 - modifying called behaviors 646
 - object node 631
 - object nodes 622
 - properties for SysML 1255
 - Send Action State 763
 - send activity state 623, 1257
 - separate transitions 623
 - setting activity parameters 623, 1257
 - setting default flow 1258
 - similarity to flow charts 656
 - states 621
 - subactivities 622
 - subactivity 629, 1258
 - swimlanes 623, 1257, 1259
 - synchronization bars 637
 - SysML behavior interconnections 1254
 - system engineering options 1232
 - systems engineering drawing icons 1256
 - systems engineering options 1231
 - termination state 622
 - termination states 629
 - transition label 1256
 - transition labels 623, 635
 - transitions 633, 1258
- Activity flows 633, 666
 - creating flow chart 666
- Activity parameters 647, 649
 - graphical characteristics 648
 - modifying 648
- Activity view 1232
 - consistency checks 1232
 - copy 1232
- ActivityReferenceToAttributes property 652
- Actors 520, 573, 851, 1249
 - attributes 522
 - browser icon 316
 - collaboration diagram 851, 852
 - concurrency 522
 - creating in OMD 573
 - creating UCD 521
 - drag and drop 345
 - drawing 521
 - elements 339
 - external interfaces 1261
 - generate code for 878
 - generating code for 878, 937
 - generating code for UCDs 523
 - limitations on characteristics 938
 - line creating 698
 - owner 522
 - use cases 520, 1252
- Actual Call dialog box 844
- Ada language 1
 - code generation profile 225

- code generator symbols 1064
- composite types 136
- conditions 141
 - roundtripping 107
 - SPARK profile 226
 - static models 144
 - variant record keys 141
- Additional keywords (reverse engineering) 1017
- Additional Sources
 - configuration option 880
 - field 861
- AdditionalBaseClasses 976
- AdditionalKeywords property 1017
- AdditionalNumberOfInstances property 971
- Address spaces, sending events across 759
- Advanced button 882
- Aggregates property 1370
- Aggregation 557
- Aggregation Kind field 533, 554
- Algorithms 973
 - activity diagram 619
 - flow chart 655
 - sequence comparison 712
- All Include Files option 1020
- All Views view (DoDAF) 1304
- All Views viewpoint (MODAF) 1337
- Allocations
 - block definition diagram 1264, 1271
 - in SysML 1227
- AlternativeDrawingTool property 508
- Analysis 70
 - black-box 70, 722, 1262
 - field 433
 - mode 674
 - object 71
 - recursive mode 1020
 - requirements 70
 - sequence diagram 432
 - trade in Harmony 1234
 - white-box 72, 722
- AnalyzeGlobalFunctions property 1031
- AnalyzeGlobalTypes property 1031
- AnalyzeGlobalVariable property 1031
- AnalyzeIncludeFiles property 1020
- Anchored Elements box 388
- Anchors 389, 390, 391
- And state, code generation 741
- Animated browser 1114
- Animated sequence diagrams 1114, 1118
- Animated statecharts 1118, 1124
- AnimateSDLBlockBehavior property 311
- Animation 209, 1079, 1080
 - automatic for sequence diagrams 706
 - automatic scripts 1129
 - black-box 1130
 - breakpoints 1101, 1102, 1104
 - calling operations 695, 883, 1107
 - command bar 1094
 - configurations 1083
 - creating initial instances 1093
 - debugging applications 211
 - ending 1091
 - exception handling 1136
 - highlighting 1125
 - hints 1136
 - in Eclipse 167
 - injecting events 1094
 - instance lines on sequence diagrams 1116
 - instrumentation mode 1084
 - keyboard shortcuts 1451
 - lifeline 1135
 - limitations 1109
 - local host 1085
 - locating ports automatically 1087
 - mainThread 1098
 - messages 1120
 - modes 983, 1111
 - overriding 1137
 - panel diagrams 797, 799
 - partial 1088, 1089, 1109
 - preparing for 209, 1080
 - properties 1130
 - remote targets 1086
 - return values 694
 - running 210, 1080, 1085
 - running without Rhapsody 1141
 - scripts 1127
 - SDLBlock 311
 - selective instrumentation 882
 - sequence diagrams 1115
 - serialization functions 1138
 - settings 1137
 - stand-alone text version 1143
 - statecharts 1124
 - suppress animated sequence diagram messages 1122
 - synchronization with application 1136
 - testing library 1088
 - threads 1097
 - toolbar 35, 1092
 - viewing 1112
- Animation tab 25
- AnimationPortNumber 1087
- AnimationPortRange 1087
- AnimSerializeOperation property 1138
- AnimUnserializeOperation property 1140
- Annotations 387, 1064, 1215
 - creating 385
 - Java 1215
 - modeled versus graphical 385
 - symbols 108
 - types 384
- Anonymous instance 965
- API
 - annotations supported by COM 395

- exporting COM interfaces 1419
- ports 129
- sending events across address spaces 760
- Applications 211, 316
 - animating steps 35
 - development cycle 1
 - external 131
 - helper 494
 - monitor and control 1143
 - OSEK21 adaptor 1375
- Architectural design
 - UML 72
- Architectural Design Wizard 1235
- Architecture
 - AUTOSAR 1371
 - block definition diagrams 1262
 - high-level diagram 1262
 - multi-threaded 1
 - service oriented (SOA) 300
 - static 970
- Archive 212, 240
- Arguments 92
 - comparing 720
 - creating 92
 - displaying 684
 - field 687
 - roundtripping 1066
 - variable length list 946
- Arrange
 - toolbar 469
- Arrows
 - creating 696
 - dependency 569
 - destroying 696
 - drawing 442
 - naming 444
- Artifacts 1314, 1347
- Associate
 - image file with element 349
 - stereotype with element 407
- Association classes 1417
- Association Ends field 551
- Associations 524
 - aggregation 557
 - bi-directional 549
 - block definition diagram 1263
 - block definition diagrams 1263
 - changing underlying 855
 - Complete Relations 566
 - composite 558
 - composition 558
 - consist of 551
 - container 559
 - creating 549
 - directed 555, 556
 - display options 560
 - displaying in the browser 558
 - features 524
 - icon in browser 317
 - implementing 559
 - in collaboration diagram 852
 - modifying features 550
 - navigability 533, 554
 - pop-up menu options 560
 - qualified in OMDs 533, 554
 - roundtripping 1066
 - selecting in OMDs 561
 - UCD 524
- Asynchronous events 683
- ATG 285
- Attribute types 82
- Attributes 79, 81
 - accessor 950
 - action 626, 660
 - actor 522
 - adding to OMD 528
 - classes 79
 - display options 113
 - initializing 94
 - listed in table views 243
 - mutator 950
 - private 79
 - protected 79
 - protected icon 79
 - Reversed 120
 - roundtripping 1066
 - static 83, 945
 - use case 519
 - value 535
- Auto Enlarge 456
- AutoCopied property 399
- Auto-creating instance lines on sequence diagrams 1116
- Auto-indent text 420
- AutoLaunchAnimation property 706
- Automatic
 - animation of sequence diagrams 706
 - ANSI-compliant code generation 1
 - code generation 8, 925
 - diagram population 33
 - instance lines on sequence diagrams 1116
 - refresh rate 1193
 - requirement sequences comparison 73
 - save 224, 238
- Automatically show this window check box 614
- Automotive development 1371
 - AUTOSAR profiles 225, 1371
 - C profile 225
 - OSEK21 adaptor 1374
- AutomotiveC profile 225, 1371, 1374
 - code generation 1377
 - extended execution model 1374
 - stereotypes 1378
- AutoReferences property 399
- AUTOSAR 2, 1371

- C language only 225
- import/export 1373
- profiles 225
- Autosave 238
 - directory 284
 - file 284
- AutoSaveInterval property 238
- AvailableMetaClasses property 503
- B**
- Back button 31
- Backup 239
 - BackUps property 239
 - directory 284
 - file 284
 - loading 240
 - project 239
- Backward compatibility 938, 987
 - code generation in RiC 987
 - for pre-3.0 Rhapsody models 941
 - issues 1056
 - profiles 398
- BaseNumberOfInstances property 971
- Batch mode 1059
- Behavioral port 132
- Behaviors
 - activity diagrams 619
 - attribute 120
 - classes 106
 - dynamic views 67
 - ports 98
 - time-based 1
- Bidirectional option 1061
- Binding 813
 - connectors 1264, 1271
 - embedded objects 1189
 - layout and view 256
 - SysML value 1227
- Bitmaps 383
 - associating with stereotypes 408
 - transparent background 408
- Black-box
 - analysis 70, 1262
 - animation 1130
 - testing 1134
- Block definition diagrams 1262, 1266
 - aggregations 1263
 - allocation 1264
 - association 1263
 - binding connector 1264
 - block 1263
 - connector 1263
 - constraint block 1263
 - constraints 1264
 - create package 1263
 - dependency 1263
 - dimension 1264
 - directed associations 1263
 - directed composition 1263
 - drawing tools 1263
 - flow 1263
 - flow port 1263
 - flow specification 1263
 - graphics in 1265
 - inheritance 1263
 - interface 1263
 - netcentric 301, 302
 - part 1263, 1271
 - perform trade analysis in Harmony 1234
 - problem satisfaction 1264
 - rationale 1264
 - standard port 1263
 - starting point 1226
 - unit 1264
 - value type 1264
- BlockIsSavedUnit property 280
- Blocks 1227, 1261
 - action 626
 - constraint 1227, 1263, 1270, 1271, 1272
 - create test bench from 1233
 - Java initialization 980
 - SDL 226, 310
 - Statemate 1383
- Bookmark 428
- Border, system 678
- Boundary box 517, 1249
- Bounded relation 972
- Box
 - boundary 517
 - drawing 441
 - naming 444
 - selection handles 453
- Break 1094
- break command 1147
- Breakpoints 307, 1100
 - creating 1101
 - deleting 1104
 - deleting tracer 1147
 - enabling 1104
 - reasons for 1102
- Browse From Here browser 322, 323
- Browser 166
 - displayed in Eclipse 166
 - link to editor 205
- BrowserIcon property 506
- Browsers 12, 313, 342
 - active component icon in 316
 - activity diagram icon in 318
 - actor icons 316
 - animated 313
 - association icon in 317
 - basic icons 316
 - Browse From Here browser 322, 323

- class icons 316
- collaboration diagram icon in 318
- comments icon 318
- component diagram icon in 318
- component icons 316
- constraint icon in 317
- controlled file icon in 317
- copy elements in 345
- copy multiple elements 12
- create objects in 336
- create package in 335
- creating classes 75
- default transition icon in 317
- delete multiple elements 12
- deleting categories 352
- deleting items 324
- dependency icon in 317
- deployment diagrams icon in 318
- display modes 314
- display stereotype or model element 347
- drag and drop elements in 345
- editing code 345
- event icon in 317
- executables icon 316
- file icon 316
- filtered views 320
- filtering by views 28
- filtering views 320
- flow port icon in 317
- folder icon 316
- HTML 1421
- hyperlinks folder icon 316
- keyboard shortcuts 1451
- menu options for Harmony 1231
- object model diagrams icon in 318
- opening 314
- opening multiple instances 12
- part icon in 316
- profile icon in 317
- profiles 234, 329
- profiles displayed in 396
- profiles folder 329
- projects in 262
- removing elements 352
- requirement icon in 317
- requirements diagram icon in 318
- Rhapsody 313
- sequence diagrams icon in 318
- Settings folder 328
- state icon in 317
- statechart icon in 318
- stereotype icon in 317
- structure diagram icon in 318
- superclass icon 317
- table and matrix icons in 318
- tag icon in 317
- type icon in 317

- unit icon 316
- use case diagram icon in 319
- use case icon in 317
- view overridden properties 321
- viewing swimlanes 644

Bubble Knob control 801

Build

- Component option 873
- diagram 34
- failed error 1438
- Set field 865
- tab 22
- target component 927

Button Array control 810

Button Array tool 800

C

C language 1

- add external file 840
- animation 1137
- automotive development 1371
- automotive profile 225
- automotiveC profile 1374
- AUTOSAR 225
- backward compatibility 987
- code generation customization 989
- code generator symbols 1064
- component file type 864
- create file model elements icon 528
- creating a hierarchy of packages 605
- enumeration types property 143
- extended C_OXF for automotive 1374
- FunctionalC profile 33, 34, 225
- FunctionalC profile diagrams 7
- interfaces 985
- interrupt-driven framework 236
- optimization for ports 986
- OXF 236
- packages 223
- panel diagrams 797
- partial animation 1088
- ports 985
- preserving comments 1052
- projects in Eclipse 163, 292
- RespectProfile 226
- reverse engineering 997
- reverse engineering legacy code 600
- roundtripping 107
- Send Action State code generation 764
- serialization properties 981
- simplifier code generation 398
- simplifying code generation 991
- Simulink profile 1427
- SimulinkInC profile 226
- statechart serialization 981
- Statemate blocks 1383

- StateMate code generation 1383
- static models 144
- strings in Web-managed devices 1171
- undefined symbols in reverse engineering 1016
- C++ language 1
 - call stack 1113
 - class implementation 94
 - code generator symbols 1064
 - component file type 864
 - composite types 136
 - constant 140
 - constructs in reverse engineering 1057
 - dialects in reverse engineering 1018
 - enumeration types property 143
 - for action language 1275
 - functor-based code generation 651
 - inheritance 981
 - invoking operation calls during animation 1107
 - library for reverse engineering 1037
 - packages 223
 - panel diagrams 797
 - partial animation 1088
 - preserving comments 1052
 - projects in Eclipse 163, 292
 - RespectProfile 226
 - reverse engineering 997
 - roundtripping 107
 - Send Action State code generation 764
 - serialization properties 981
 - simplifier code generation 398
 - Simulink profile 226, 1427
 - statechart serialization 981
 - static models 144
 - template limitation in Web-managed devices 1171
 - undefined symbols in reverse engineering 1016
 - variables 337
- Call behaviors 633
 - in activity diagram 645
- CALL command 1150
- Call Graph diagram 34
- CALL macro 695
- Call operations 623, 1107, 1257
 - in animation 883
 - nodes 621
- Call stack view 1113
- CALL_INST macro 695
- CALL_SER macro 695
- Called behaviors
 - displaying features 646
 - limitations 646
 - modification 646
- Calls
 - in activity diagrams 622
- Cancel
 - changes in Features dialog box 44
 - timeout 697
- Categories
 - deleting 352
 - mode 314, 1471
- Change
 - applying 44
 - elements 455
 - hyperlink 55
 - language type 1070
 - line shape 443
 - order of types 147
- Change to Package option 399
- Change to Profile option 399
- Check
 - for consistency in activity diagrams 1232
 - for static memory allocation 974
- Check Data 1399
- Check model 191
- Check Model option 903
- Check Model tab 19
- Checklist 907
- Checks 899, 904
 - activity view 1232
 - swimlane consistency 1232
- Checks tab 900
 - component diagrams 881
- Class Type field
 - actor 522
- ClassCentricMode property 674
- ClassCodeEditor 935
- Classes 75, 529, 542, 682, 1233
 - active without statecharts 778
 - aggregated associations 557
 - as containers 341
 - attributes 79
 - base for rapid ports 126
 - behavior 106
 - bi-directional association 549
 - browser 338
 - browser icons 316
 - code structure 943
 - collaboration diagram for multiple objects 851
 - composite 528, 543, 840
 - composite associations 558
 - constructor arguments 92
 - constructors 91, 93
 - converting to objects 534
 - create test bench from 1233
 - creating 75, 542
 - creating in OMD 542
 - default name 75
 - defining features 76
 - deleting 115
 - dependencies between 568
 - derivation 104
 - destructors 95
 - directed association 555
 - display name 110
 - display options 109

- drag and drop 345
- editing code 107
- files 284
- functor 651
- Generic Class 153
- header file 943
- importing as a type 1037
- in collaboration diagram 850
- inheriting from external 546
- instances 1126
- language type in nested 1070
- mapping to types and packages 1024
- modeling for reverse engineering 1037
- naming guidelines 272
- nested 78, 161
- nested roundtripping 1066
- object node association 632
- opening the main diagram 108
- operations 84
- ports 98
- primitive operations 85
- properties 103
- reactive 781
- reactive and refining the hierarchy 787
- receptions 88
- reference 1032
- regular 77
- relations 99
- relations, show all 101
- removing 115
- roundtripping 1062, 1066
- roundtripping supported modifications 1065
- solutions 1234
- structured 476
- superclass 317, 356, 542, 545
- swimlane association 641
- tags 103
- Template Class 153
- template instantiation 77
- templates 77, 153
- triggered operations 91
- Classifier role 850
- Classifier role names 682
- Classifier roles 679
- ClassIsSavedUnit property 280
- CLASSPATH 1013
- Cleaning
 - delete redundant code files 937
 - delete sequence diagram messages property 674
 - old objects 928
 - redundant source files 937
- CleanupRealized property 674
- ClearCase 212
- Clipboard, consistency check results to 1232
- Code 68
 - active view 933
 - associated with element 205
 - clean 928, 932
 - compilation errors 931
 - displaying 26
 - DMCA 199
 - documentation system 955
 - documentation systems 955
 - editing 932
 - editing from a diagram 202
 - editing from browser 345
 - editing in Eclipse 200
 - editing with an external editor 936
 - elaborative generation 922
 - exporting to Eclipse 179
 - external 1010
 - for actors 523
 - for classes 107
 - generate for actors 878
 - generated 932
 - generating 192, 198
 - generating for files 591
 - generating for relations 609
 - generation 106
 - generation for actors 937
 - generation of individual elements 930
 - generation results 931
 - inlining 795
 - keyboard shortcuts 1451
 - legacy 72, 1010
 - line numbers 933
 - return from command-line 1437
 - reverse engineering in Eclipse 185
 - roundtripping 1060
 - roundtripping class code 107
 - source import 185
 - structure of generated 943
 - toolbar 30
 - translative generation 922
 - viewing 932
 - wrapping with `#ifdef #endif` 960
- Code check box 362
- Code editor, external 936
- Code generation 8, 192, 198, 921
 - abort 926
 - activity diagrams 651
 - ANSI-compliant 1
 - automatic 925
 - automotiveC 1377
 - backward compatibility 987
 - change order of operations/functions 953
 - component diagrams 939
 - customizing C 989
 - customizing using properties 989
 - customizing using rules 989
 - dynamic model -code associativity 925
 - flow charts 669
 - for links 567
 - for Send Action State 763, 764

- for templates 161
 - forcing complete 924
 - guidelines 925
 - incremental 923
 - JavaAnnotations 1220
 - limitations 987
 - macros 1055
 - option 1061
 - options 198
 - restrictions in activity diagrams 653
 - restrictions in flow charts 669
 - simplified models 991
 - stop 926
 - transformation phase 989
 - writing phase 989
- Code respect 1073
- activating 1074
 - reverse engineering 921, 1050, 1056
 - roundtripping 921, 936, 1059, 1070, 1073
 - SourceArtifact 1075
 - where code respect information is defined 1075
- Code writer, external 992, 993
- Co-debugging 308
- CodeGeneratorTool property 991, 1052
- CodeTEST 881
- Collaboration diagrams 65, 67, 847
- actors in 851
 - browser icon 318
 - changing underlying association 855
 - classifier role 850
 - creating 33, 433
 - creating links in 852
 - creating messages in 856
 - define for core cases 72
 - editor 6
 - files 284
 - icons 849
 - IntelliVisor information 481
 - link 852
 - message numbering 847
 - messages 855
 - multiple objects 851
 - specifying behavior 70
- CollectMode property 1054
- Color 418
- coding in editor 418
 - default settings 418
 - in sequence diagram comparisons 714
 - set diagram fill 440
- Command line 1435, 1436, 1440, 1442
- Command Prompt tool 1094
- Command-line interface 1433
- commands 1435, 1442
 - exiting 1437
 - interactive mode 1434
 - interactive switch 1434
 - methods of operation 1433
 - order of commands 1436
 - path names 1435
 - quotation marks in commands 1435
 - return codes 1437
 - scripts using commands 1437
 - socket mode 1434
 - switches 1435, 1440
 - syntex 1435
- Commands 1435, 1442
- examples for running Rhapsody 1439
 - order of 1436
 - tracer 1144, 1147
- Comment specification search option 362
- Comments 384, 1052
- added to components 873
 - anchoring 389
 - browser icon 318
 - floating 1052
 - in tracer commands 1145
 - limitations 1052
 - preserving (reverse engineering and roundtripping) 1052
 - reverse engineering 1052
 - specification 362
- Common Drawing toolbar 37
- using 37
- Communicate with ports 130
- Compare
- features 45
 - Names and Values option 721
 - Names Only option 721
 - sequence diagram excluding a message 719
 - sequence diagram instance groups 722
 - sequence diagram's algorithm 712
 - sequence diagrams 717
- Compartments 528
- display stereotype for elements in list 474
 - show label 110
- Compatibility
- automatic settings 397
 - profiles for backward 398
- Compiler
- messages 927
 - specific keywords 949
- Compiler Switches field 865
- Compilers
- adding keywords 960
 - MATLAB MEX 1430
- Complete Relations
- associations 566
- Complete Relations option 479
- Complex parameters
- overriding 1137
- ComplexityForInlining property 796
- Component diagrams 66, 858
- browser icon 318
 - code generation 939

- components 860
- creating 33, 433
- dependency 871
- drawing icons 859
- editor 7
- elements 860
- files 862
- files extension 284
- folder 868
- IntelliVisor information 482
- Component instances
 - deployment diagrams 892
 - modify features 894
- Component interface 871
 - creating 871
- Component Type field
 - component instance 894
- Component View 1405
- ComponentIsSavedUnit property 280
- Components 223, 330, 860
 - active 316, 874
 - adding file to 862
 - browser icons 316
 - building 927
 - building menu option 873
 - comments added to 873
 - configuration 330
 - controlled files added to 873
 - creating 860
 - creating interface 871
 - deleting composite 966
 - directory 285
 - file 284
 - files 330
 - generating 873
 - icon 860
 - importing from Simulink 1426
 - interface 857
 - pop-up menu 873
 - view 320
- Components (subsystems) 1081, 1284
 - creating 1081, 1284
 - creating configuration 1083
 - features 1285
 - setting features 1082
- Components-based development in RiC 984
- Composite association 558
- Composite class 476, 528, 543, 840
- composite stereotype 1246
- Composite type 135
 - properties 142
- Composition association 558
- Composition option 533, 554
- Compound transition 746
- Concurrency field
 - actor 522
 - block 843
- Condition connectors 636, 668, 735
 - creating 636, 668
- Condition mark 696
- Configuration 1083, 1173
- Configuration for Simulink 1430
- Configuration management 374
 - not for project list files 271
- Configuration management (CM) 212
 - DiffMerge tool 220
 - Eclipse and Rhapsody 213
 - performing operations in Eclipse 218
 - preferences 220
- Configuration Management tab 25
- Configurations 209, 330, 874, 1285
 - active 182, 875
 - active Eclipse 193
 - active state 769
 - automotiveC stereotypes 1378
 - checks 881
 - clean 932
 - component 873
 - convert to Eclipse 293
 - creating animation 1083
 - default 1285
 - Eclipse 292
 - features 876
 - generate main option 875
 - generating 875
 - initialization search option 362
 - partial animation 1089
 - pop-up menu 874
 - regenerating 924
 - scope 877
 - Seat as Active option 875
 - settings 879
 - Web components property 1173
 - Web-enable 1286
- Conflict transition 776
- Conform 1227
- Connect
 - objects using ports 131
 - ports 125
 - to filtered views 1175
 - to model from the Web 1174
 - to model from the Web, troubleshooting 1175
- Connectors
 - activity diagram 635, 637
 - binding 1264, 1271
 - block definition diagram 1263
 - condition 622, 636, 668
 - diagram 622
 - flow charts 668
 - history 735
 - junction 622
 - junction for activity diagram 635
 - junction for flow charts 668
 - junction for statecharts 735

- statechart 736
 - termination 735
 - Consists of field 551
 - Constant modifier 82
 - ConstantVariableAsDefine property 1015
 - Constraint specification search option 362
 - Constraints 338, 384
 - anchoring 389
 - binding 1273
 - block 1263, 1271
 - block definition diagram 1264
 - blocks 1270, 1272
 - browser icon 317
 - editing text 385
 - finding references 390
 - parameters 1271, 1273
 - properties 1272, 1273
 - property 1271
 - specification supports Asian languages 362
 - SysML 1227
 - Constructor Arguments dialog box 92
 - Constructors 91
 - Features dialog box 93
 - implementation file 950
 - roundtripping 1066
 - Constructs added to Model option 1048
 - Constructs Analyzed option 1048
 - Contacting technical support 1461
 - Container 559
 - Containment by value 973
 - Content window 14
 - Contract tab 121
 - Contracts 98
 - specifying 122
 - Control
 - model from the Web 1181
 - point 456
 - Control Properties dialog box 814
 - Controlled files 365
 - browse to 368
 - configuration management 374
 - creating 366
 - features 371
 - limitations 376, 378
 - tags 373
 - troubleshooting 375
 - Convert
 - class to object 534
 - external elements 605
 - file 590
 - note to comment 389
 - object types 533
 - package to profile 399
 - profile to package 399
 - Conveyed information 578
 - Copy
 - actor menu option 573
 - Copy with Model 468
 - Copying 263
 - between Features dialog boxes 45
 - element in browser 345
 - elements 466
 - elements to other projects 265
 - format from one element to another 461
 - instance line menu option 455
 - CORBA 3, 907
 - check 907
 - inheritance 611
 - limitation 146
 - CPPCompileCommand property 928
 - CPU 889, 891
 - Create Reference Sequence Diagram option 702
 - CreateDependencies property 1020
 - CreateReferenceClasses property 1030
 - Custom help file 400
 - Customer support 1459
 - CustomHelpMapFile property 402, 403
 - CustomHelpURL property 400, 403
 - CustomizableTableAndMatrixLayoutsPkg 1354
 - CustomizableTableAndMatrixViewsPkg 1358
 - CustomizedStereotypesPkg 1355
 - Customizing
 - C code generation 989
 - code generation rules with RulesComposer 993
 - keyboard mappings 422
 - linking to helper applications 494
 - navigation to your Web GUI 1180
 - new diagrams 506
 - profile 499
 - Rhapsody Web server 1192
 - RiC rules with RulesComposer 993
 - Web interface 1182
 - Cut option
 - actor menu 573
 - instance line menu 455
- ## D
- Data
 - checking 1275
 - export to Excel 261
 - flow 676
 - manage from table or matrix 261
 - query in table 245
 - vendor-neutral sharing 1422
 - Dataflows 699, 1123
 - DataTypes property 1037
 - Debugger 308
 - Debugging 208, 1080
 - animated applications 211
 - perspective 209
 - Decision points 619, 656
 - Decomposed field
 - instance line 680

- Decomposition 703
 - limitations 703
- Default flows
 - activity diagram 634
 - flow chart 666
- Default transition
 - statechart 765
- DefaultDirectoryScheme property 279
- DefaultProvidedInterfaceName property 126
- DefaultReactivePortBase property 126
- DefaultReactivePortIncludeFile property 126
- DefaultRequiredInterfaceName 126
- Define View page 1178
- Defined In field 388
 - actor 522
- Defined symbol 1013
- DEGREES_PER_RADIAN variable 337
- Deleting
 - after search 230
 - anchors 391
 - anonymous instances 966
 - breakpoint 1104
 - breakpoint tracer 1147
 - categories 352
 - classes 115
 - composite components 966
 - elements from file 867
 - elements from model 471
 - from model 324
 - hyperlinks 58
 - instance groups 725
 - message groups 731
 - old objects 928
 - redundant code files 937
 - redundant source files 937
 - reference classes 1032
 - remarks 394
 - sequence diagrams 709
 - stereotypes 409
 - swimlane dividers 644
 - swimlane frames 644
 - tags 416
 - using the Edit menu 352
- Dependencies 338, 525, 568, 569
 - activity diagrams 623, 1257
 - adding stereotype 1253
 - arrow 569
 - between remarks 386
 - block definition diagrams 1263
 - browser icon 317
 - component diagrams 857, 871
 - creating 569
 - deployment diagrams 895
 - friend 987, 1070
 - from Includes option 1028
 - in UCD elements 525
 - Link wizard 1237
 - modifying features of 571
 - new 873
 - parametric diagrams 1271
 - relationships 1245
 - requirement diagrams 1244
 - statecharts 736
 - structure diagram 841
 - system engineering 1261
 - system engineering requirements 1245
- Dependencies Linker (MODAF) 1366
- Depends On field 571
- Deployment diagrams 66, 887
 - assigning a package 897
 - component instances 892
 - creating 433
 - dependencies 895
 - drawing icons 888
 - editor 7
 - elements 889
 - files 284
 - icon in browser 318
 - IntelliVisor information 482
 - node owner 890
 - nodes 889
 - UML 66
- Derivation 104
- Derivation in requirement diagrams 1244
- derive stereotype 1246
- Derived scope 877
- Descendants
 - include in views 245, 247, 251
- Description tab 52
- Description, adding hyperlinks in 52
- Descriptions search option 362
- DescriptionTemplate 957
- DescriptionTemplate property 1209, 1211
- Design
 - architectural 72
 - basic requirements 69
 - details of 72
 - mechanistic 72
 - mode 674
 - option 433
 - requirements for SysML 1246
 - SysML requirements in use cases 1252
- Destination of transition 744
- Destructors 95
 - implementation file 950
 - roundtripping 1066
- Details tab 578
- Development
 - analysis phase 70
 - design phase 72
 - environments 163
 - implementation phase 73
 - methodology 70
 - parallel 212

- phases of 70
- testing phase 73
- Devices
 - managing remotely 1169
 - setting name in Web pages 1192
 - Web-enabled 1169, 1175, 1176
 - Web-enabled, adding files to model 1182
 - Web-enabled, connecting to from the Web 1174
 - Web-enabled, controlling 1181
 - Web-enabled, customizing the GUI 1182
 - Web-enabled, Define View page 1178
 - Web-enabled, name/value pairs 1181
 - Web-enabled, Personalized Navigation page 1179
 - Web-enabled, setting as 1170
 - Web-enabled, using properties 1173
 - Web-enabled, viewing 1181
- Diagram connectors 637
- Diagram editor
 - grid 450
 - properties for 440
- DiagramIsSavedUnit property 275, 280
- Diagrams 68, 431
 - accelerators 1453
 - active resizing 38
 - activity 34, 65, 67, 619, 1254
 - add new customized 506
 - adding 176, 340
 - adding elements 340
 - adding remarks to 384
 - automatically populating 435
 - AUTOSAR 1372
 - block definition 301, 302, 1226, 1262
 - Build 34
 - Call Graph 34
 - collaboration 33, 65, 67, 847
 - comparing 52
 - component 33, 66, 857
 - connector 736
 - creating 33, 431
 - deployment 66, 887
 - drawing area 166
 - editing 68
 - editing code from 202
 - exporting as images 383
 - external block 1261, 1262
 - File 34
 - flow chart 34, 66, 655, 658
 - for FunctionalC profile 34
 - fully constructive 68
 - high-level architecture 1262
 - in reports 221
 - internal block 1233, 1261, 1267
 - locating elements 331
 - Message 34
 - naming 33
 - navigator 477
 - object model 33, 65, 66, 527
 - OpenDiagramsWithLastPlacement property 283
 - opening 434
 - output to UNISYS extensions format 1420, 1421
 - OV-1 High Level Operational Graphic 1317
 - OV-2 Operational Node Connectivity 1317
 - OV-4 Organizational Relationships 1317
 - OV-5 Operational Activity 1317
 - OV-6a Operational Rules Model 1317
 - OV-6c Operational Event-Trace Description 1317
 - OV-6c Operational State Transition Description 1317
 - panel 34, 797
 - parametric 1270, 1273
 - partially constructive 68
 - populate 1243
 - printing 379
 - Project Overview 1317
 - requirements 1243
 - SA DoDAF import 1291
 - saving 277
 - scaling 475
 - sequence 33, 65, 67, 671, 1260
 - set fill color 440
 - standard toolbar 38
 - statecharts 34, 65, 67, 733, 734, 1284
 - structure 33, 66, 839
 - SV-10b System State Transition Description 1318
 - SV-10c System Event-Trace Description 1318
 - SV-11 Physical Schema 1318
 - SV-2 System Communication Description 1318
 - SV-4 System Functionality Description 1318
 - SV-8 System Evolution Description 1318
 - SysML from SA data 1293
 - systems engineering 1223
 - test context 1233
 - UML 65, 223
 - use case 33, 65, 66, 515, 1240, 1249
 - view 320
- DiagramsToolbar property 507, 509
- Dialect 1018
- DiffMerge tool 212, 220, 286
 - annotations 395
 - hyperlinks 52
 - no project list support 271
 - supports action pins 649
 - supports activity parameters 649
 - templates 153
 - version conflicts 1299
- Digital Display control 806
- Digital Display tool 800
- Directed association 555, 556
 - creating 555
- Directed composition 1263
- Direction field
 - flow 578
- Directories
 - autosave 284
 - backup 284

- component 285
- containing reference classes 1033
- DefaultDirectoryScheme property 279
- flat structure 280
- hierarchical structure 280
- project 284
- saving units in separate 279
- structure in RE 1029
- Directory
 - field 861, 880
 - structure 279
- Dismiss
 - IntelliVisor 480
- Display
 - associations 558
 - browser modes 314
 - code 26
 - messages-to-self 1130
 - port interfaces 123
 - stereotype in compartment lists 474
 - stereotype of element in browser 347
- Display options
 - actor menu 573
 - annotations 392
 - attributes 113
 - class name 110
 - default options 466
 - Enable Image View 1265
 - equations 1274
 - file 586
 - flow menu 579
 - general selections 110
 - operations 113
- DisplayMessagesToSelf 1130
- DisplayMode property 314
- Distributed team 212
- Dividers, swimlane 642
- DMCA 199, 1061
 - mode 1061
 - roundtripping 108
- Dock
 - Features dialog box 45
- Documentation 1463
 - Gateway 1242
 - Installation Guide 1223
 - property definitions 47, 103
- Documentation note, converting to comment 389
- Documentation system
 - sample 955
- DoDAF 2, 1301, 1314
 - All Views 1304
 - architectural model 1323
 - artifacts 1314
 - compared to MODAF 1333
 - creating a project 1314
 - helpers 1327
 - importing SA diagrams 1291
 - limitations 1325
 - Operational view 1303
 - OV-1 High Level Operational Graphic 1317
 - OV-2 Operational Node Connectivity 1317
 - OV-3 matrix 1321
 - OV-4 Organizational Relationships 1317
 - OV-5 Operational Activity 1317
 - OV-6a Operational Rules Model 1317
 - OV-6c Operational Event-Trace Description 1317
 - OV-6c Operational State Transition Description 1317
 - OV-7 Logical Data Model 1317
 - profile 225, 1301
 - Project Overview 1317
 - reports 1312, 1323
 - SA encyclopedia 1292
 - setup packages 1311
 - SV-1 System Interface Description 1317
 - SV-10a Systems Rules Model 1318
 - SV-10b System State Transition Description 1318
 - SV-10c System Event-Trace Description 1318
 - SV-11 Physical Schema 1318
 - SV-2 System Communication Description 1318
 - SV-4 System Functionality Description 1318
 - SV-8 System Evolution Description 1318
 - System Architect diagrams 1291
 - Systems view 1303
 - tags 1319
 - Technical view 1304
 - troubleshooting 1326, 1330
 - utilities 1309
 - verifying installation 1326
 - views 1302
- Domain checks 901
- Domain Specific Language 1314
- Domain Specific Language (DSL) 2, 1314, 1347
- DOORS 1240, 1387
 - action pins/activity parameters 649
 - Check Data 1399
 - creating a project for Rhapsody 1390
 - dxlapi.dll 1388
 - exiting 1403
 - export options 1391
 - formal modules 1391
 - Gateway 1241
 - installation requirements 1388
 - Interface dialog box 1390
 - invoking 1390
 - link modules 1388, 1402
 - linking data 1395
 - navigating between Rhapsody and 384
 - navigating to Rhapsody 1390
 - nested packages 1391
 - objects 846
 - on Solaris systems 1388
 - pc_server.dxl 1388
 - requirements 1387
 - shadows 1395

- stored information 1398
 - using with Rhapsody 1389
 - DOS 1141
 - doStep() function 1427
 - Doxygen
 - template-based comments 957
 - using 955
 - Drag and drop in browser 345
 - Drawing
 - area 12
 - arrows 442
 - boxes 441
 - elements 440
 - mode 440
 - state 736
 - Drawing area
 - displayed in Eclipse 166
 - Drawing icons 10
 - DrawingShape property 507
 - DrawingToolBar property 506, 509
 - DrawingToolIcon property 506, 507
 - DrawingToolTip property 507
 - DSL 1314, 1347
 - dxlapi.dll 1388
 - Dynamic behavior views 67
 - Dynamic model-code associativity 925, 1061
 - code generation 925
 - roundtripping 108
 - Dynamic Model-Code Associativity (DMCA) 199
 - Dynamic Object Oriented Requirements System (DOORS) 1387
- E**
- Eclipse 163, 165, 292
 - animation 209
 - build project 206
 - code editor 200
 - configurations 193
 - confirming Rhapsody Platform Integration 164
 - content management operations with Rhapsody 213
 - create IDE project 193
 - debug project 208
 - disassociating project from Rhapsody 299
 - DMCA 199
 - Dynamic Model-Code Associativity (DMCA) 199
 - edit Rhapsody project with 298
 - export Rhapsody code 179
 - importing projects into Rhapsody 293
 - importing Rhapsody units 182
 - locating Rhapsody code in 298
 - Model browser 213
 - open existing configuration 298
 - performing CM operations 218
 - perspectives 168, 197
 - Platform Integration 163
 - projects 170
 - properties 297
 - reports 221
 - repository 163
 - reverse engineering 185
 - reverse engineering source code 185
 - Rhapsody Debug perspective 166
 - Rhapsody Log 198
 - Rhapsody Modeling perspective 165
 - Rhapsody perspectives in 168
 - Rhapsody Platform Integration 163
 - Select with Descendents in Unit View 213
 - sharing a Rhapsody model 215
 - source code import 185
 - Unit View 213
 - viewing code 205
 - work area 171
 - Workflow Integration 163
 - Edit
 - code 932
 - constraint text 385
 - diagrams 68
 - elements 455
 - features 234
 - hyperlink 55
 - implementation code 613
 - remark text 387
 - Rhapsody project 233
 - text 472
 - undo/redo 234
 - Edit Code option 573
 - Edit Configuration Main File option 875
 - Edit Makefile option 875
 - Edit menu
 - Add New option 433
 - Complete Relations option 479
 - Format option 458
 - selecting elements 452
 - Edit Type Order option 147
 - Editable list 80
 - Editing
 - using mouse 427
 - Editor
 - accelerators 1455
 - associating files with 935
 - collaboration diagram 6
 - component diagram 7
 - deployment diagram 7
 - drag and drop to 345
 - object model diagrams 6
 - opening 107
 - Rhapsody's internal 417, 418
 - selecting 867
 - sequence diagram 6
 - set scope 932
 - statechart 6
 - use case diagram 6
 - EditorCommandLine property 867, 936

- Editors
 - Eclipse 200
 - graphic 342
 - show in browser feature 204
 - XML for SA map 1291
- Elaborative code generation 922
- Element types 502
- Elements 223, 233, 328
 - adding 176, 233, 328
 - adding hyperlinks 52
 - adding points to 443
 - adding to file 865
 - arranging 469
 - associated with code 205
 - associating with image file 349
 - associating with stereotype 407
 - based on new term stereotypes 407
 - cell types in matrix views 249
 - changed to units 275
 - changing common operations 455
 - changing the format 458
 - classes 338
 - code for external 608
 - code generation 930
 - component diagram 860
 - constraints 338
 - copy in the browser 345
 - copying 265, 466
 - create for customized diagram 507
 - creating external 598
 - defining tag for 414
 - deleting from model 471
 - deleting using the Edit menu 352
 - dependencies 338
 - deployment diagram 889
 - diagram editing 455
 - diagrams 340
 - display stereotype in browser 347
 - drawing 440
 - editing in component diagram 867
 - events 338
 - exposing on the Web 1172
 - external 598, 605, 1010
 - external and creating by modeling 603
 - external and generating code 609
 - external source path 607
 - files 339, 584
 - finding usage 358
 - functions 336
 - generating code for relations 609
 - graphical 348
 - identification 341
 - in profiles 225
 - labels 342
 - locate from code 934
 - locating in the browser from the code editor 204
 - locating on diagram 331
 - making a unit 276
 - mapped to the folder field 869
 - matrix view of 251
 - moving in browser 345
 - moving in graphic editors 457
 - NewTerm stereotype 49
 - nodes 339
 - non-rectangular 453
 - objects 336
 - operations on a group 12
 - package 223
 - package types 335
 - paths 341
 - pinned features' display 44
 - project 223
 - realizing 615
 - receptions 88
 - receptions, browser icon 338
 - references 358
 - removing from view 471
 - removing points from 443
 - removing with browser 352
 - renaming 346
 - reordering in browser 353
 - resizing 456
 - saved as units 264
 - search 364
 - searching 187, 361
 - selecting 452
 - selecting multiple 454
 - selecting to import (Rose) 1407
 - selection handles 453
 - Send Action State 763
 - set display default options 466
 - set element size as default size 466
 - set formatting 466
 - setting as Web-manageable 1170
 - show in browser from editor 204
 - special characters in names 272
 - stereotypes for SysML 1227
 - table and matrix views 241
 - types 338
 - units 274
 - usage 358
 - use cases 339
 - variables 337
- Elements box 864
- ElementsMap.xml 1295, 1296
- Else branch 768
- Embeddable object 1189
- Embedded Coder License (ERT) 1427
- EMF image format 383
- Enable Docking by Drag option 45
- Enable Operation Calls field 883
- Enabled transition 776
- EnableMultipleAnimation 1087
- End1 and End2 tabs 553

- EnterExit points 771
 - Updating 771
 - EntryPoint property 308
 - Enumerated type
 - creating 137
 - reverse engineering 1051
 - Enums, Java 151
 - Environment
 - field 864
 - variable using with reference units 281
 - Environment Settings group box
 - configuration 881
 - files 864
 - Equations 1274
 - Error
 - checking 931
 - in model 899
 - parsing 1045
 - ERT (Embedded Coder License) 1427
 - Event Generator 1105
 - Events 338, 690
 - accept event action in systems engineering 1257
 - accept event actions 623
 - adding operations to 751
 - allocated to subsystems 1235
 - browser icon 317
 - class hierarchy 750
 - destruction 677
 - Features dialog box 343
 - generating in animation 1105
 - generating, gen method 751
 - generating, tracer 1152
 - generating, using friendship 751
 - History list 1106
 - history list file 285
 - in interrupt handlers 972
 - injecting 1094
 - internal 751
 - name in notation 764
 - naming conventions 273
 - pooling 971
 - private 751
 - queue 751
 - queue view 1113
 - receptions 88, 90
 - roundtripping 1067
 - semantics 751
 - Send Action State 763
 - Sending across address spaces 759
 - statechart 758
 - triggers 750
 - usage 750
 - ExcludeFilesMatching property 1009
 - Executable
 - language 1255
 - running 929
 - ExecutionModel property 1377
 - Exiting command-line interface 1437
 - Explicit
 - object 529
 - scope 877
 - Export
 - files 179
 - Exporting 1419
 - diagrams as images 383
 - labels 1393
 - models 1420
 - projects 1395
 - Rhapsody model files 1421
 - to DOORS 1387, 1391
 - ExportPictures property 1393
 - extend stereotype 1246
 - ExtendedC_OXF 225
 - External
 - checks 899, 904
 - class, inheriting from 546
 - code editor 936
 - code writer 992
 - files 588
 - hyperlink 52
 - External block diagrams 1261, 1262
 - External elements 598
 - accessing the code 608
 - converting 605
 - creating 598
 - creating by modeling 603
 - creating in pre-V5.2 models 603
 - generating code 609
 - limitations 610
 - relations, generating code 609
 - viewing the source path 607
 - visualization 1010
 - External modeling 603
- ## F
- Favorites 5, 325
 - Favorites browser 5, 325
 - creating list 326
 - limitations 327
 - removing items 326
 - showing/hiding 327
 - Favorites toolbar 32
 - Features dialog box 43
 - action in activity diagram 625
 - action in flow chart 659
 - applying changes 44
 - Asian text 342
 - attributes 79, 81
 - buttons 45
 - cancelling changes 44
 - classes 76
 - comparing elements 45
 - component instance 894

- configuration 876
- constructors 93
- copying text from 45
- define Send Action State 763
- Description tab 52
- displaying properties 46
- docking 45
- edit table and matrix data 241
- editing 234
- event 343, 690
- file 588
- flow 577
- FlowItem 581
- Functions tab 589
- General tab 76, 119
- hiding tabs 49
- messages for associations 854
- objects 532
- opening 43
- opening multiple instances 45
- Operations tab 84
- package 544
- pinned mode 44
- port 119
- Ports tab 98
- primitive operations 86
- property search 47
- receptions 89
- Relations tab 99
- tag 411
- toolbar 45
- undocking 45
- Variables tab 589
- Features window 166
 - Instrumentation Mode 209
- Fields, search in 362
- File diagrams 34, 1486
- File extensions 178
- File Type field 864
- FileSavedUnit property 280
- FileName property 546
- Files 584, 862
 - .hep 494
 - .rpy 284
 - .sbs 399
 - .sdo 717
 - adding 176
 - adding elements 865
 - adding text 866
 - adding to component 862
 - associating with an editor 935
 - autosave 284
 - backup 284
 - class 284
 - collaboration diagram 284
 - component 284
 - component diagram extension 284
 - connecting 590
 - controlled 873
 - controlled icon in browser 317
 - converting 590
 - creating 586, 863
 - delete redundant code 937
 - deleting elements 867
 - deployment diagram 284
 - display options 586
 - element 867
 - events history list 285
 - examining exported XMI 1421
 - Excel .csv 261
 - excluding from reverse engineering 1009
 - export 179
 - extensions 284
 - external features 588
 - Features dialog box 588
 - filesTable.dat 284
 - filtering out types 177
 - folders 868
 - FS and RES 1188
 - functions 589
 - generating code for 591
 - header 1021
 - header structure 943
 - implementation 339, 345, 949, 1066
 - imported shown in browser 316
 - in components 330
 - include for rapid ports 126
 - include in reverse engineering 1019
 - list analyzed for reverse engineering 1023
 - load log 285
 - object model diagram 284
 - ownership 587
 - package 284
 - paths 588
 - PDF 1463
 - pop-up menu 867
 - project 284
 - project list 271
 - reverse engineering 1025
 - reverse engineering log 285
 - ReverseEngineering.log 285
 - rhapsody.ini 271, 1087
 - save log 285
 - SD comparison options 717
 - sequence diagram 284
 - server 1184
 - specification 339, 345, 1066
 - store.log 285
 - structure diagram 284
 - support for add-on tools 594
 - target for hyperlinks 53
 - types 865
 - types of controlled 371
 - use case diagram 284

- variables 589
- VBA project 285
- Windows 371
- workspace 285
- WSDL 226, 300, 301
- Files property (for reverse engineering) 1008
- filesTable.dat file 284
- Fill color set 440
- Filtering tab 1030
- Filters
 - browser views 28, 320
 - model view 1175
- Final states 649
- Find element usage 358
- Fixed relation 972
- FixedPoint profile 225
- Fixed-point Variables 148
- Flat mode 279, 280, 1486
 - browser display 314
- Flip Right/Left options 639
- Floating comments 1052
- Flow 574, 575
 - adding information element to 582
 - browser display 576
 - conveyed information 578
 - creating 575
 - displaying the keyword 576
 - features 577
 - menu options 579
 - sample 575
- Flow charts 66, 655
 - action block 661
 - actions 658
 - activity flows 666
 - algorithm 655
 - code generation 669
 - code generation limitations 669
 - connectors 668
 - creating 34, 657, 658
 - decision points 656
 - drawing icons 657
 - limitations 669
 - Send Action State 763
 - similarity to activity diagrams 656
 - termination states 664
- Flow Ends field 578
- FlowItem 580, 581
- flowKeyword 576
- Flowports 596
 - and Simulink 1429
 - atomic 596
 - attributes 597
 - block definition diagram 1263
 - dataflows 699, 1123
 - names 597
 - non-atomic 597
 - StateMate 1385
- StateMateBlock 1385
- Flows 1261
 - activity 622, 666
 - block definition diagram 1263
 - data 676
 - default 622
 - default for activity diagram 634
 - default in activity diagram 1258
 - embedded 583
 - in structure diagrams 841
 - in SysML 1227
 - limitations 584
 - loop activity 622
- Focus
 - thread in animation 1097
 - thread in tracer 1157
- Folders 868
 - in browser 316
- Font 418
- Footers 949
- Force
 - complete code generation 924
 - roundtripping 1062
- Fork 747
 - synchronization bar 637, 638
- Fork synchronization 1259
- Form field 392
- Formal modules 1391
- Format option 458
- Formats
 - copy from one element to another 461
 - default formatting 466
 - exported diagram images 383
 - reports 1195
 - toolbar 39
- Forward button 31
- Frames, swimlane 642
- Framework
 - OMReactive class 751
 - support for 976
- Free shapes 41
- Free text check box 362
- Free text field 53
- Friend dependency 987, 1070
- Friendship 751
- FS file 1188
- FullTypeDefinition property 142
- Fully constructive diagrams 68
- Function 336
- Functional C 2
- Functional C profile 225
- FunctionalC profile 33, 34, 585
- Functions 336
 - changing order in generated code 953
 - roundtripping 1067
 - tab 589
- Functor class 651

G

Gateway 1240
 documentation 1242
 DOORS 1241
 importing requirements into Rhapsody 1241
 limitations 1242
 requirements and analysis 1241
 use case diagrams 1240
Gauge control 802
Gauge tool 800
GEN macro
 events with arguments 1095
 standard operations 975
 statechart 751
 static memory allocation 972
 tracer 1152
General tab 876
 Features dialog box 76
 port 119
Generalizations 524
Generate
 class code 106
 code for actors 937
 code for actors in UCDs 523
 code for component diagrams 939
 code for links 567
 code for objects 534
 code incrementally 923
 code with names 1250
 complete code 924
 event for statechart 751
 event, tracer 1152
 formal reports 1195
 makefiles 925
 package 924
 report 1203
Generate code
 elements 930
Generate Component option 873
Generate option 573
GeneratedCodeInBrowser 1070
GeneratedCodeInBrowser property 936
GenerateDirectoryPerModelComponent property 610,
 1026
GeneratePackageCode 924
Generation
 configuration 875
 main configuration 875
GeneratorRulesSet property 996
Generic Class 153
Global
 functions 336
 tags 413
 variables 337
Glossary 1465
Go Event command 1153
Go Idle command 1153

Go Step command 1153
Graphic editor 6
 collaboration diagram 6
 component diagram 7
 deployment diagram 7
 drawing area 12
 object model diagram 6
 sequence diagram 6
 statechart 6
 use case diagram 6
 using IntelliVisor 479
Graphical annotations 385
Graphical editors
 Send Action State elements 764
Graphics
 block definition diagrams 1265
Grid option 450
Groups
 instance 722
 message 727, 729
Guard 755
 field 745
Guidelines for naming model elements 272

H

Handles, selection 453
Harmony
 Architectural Design Wizard 1235
 auto-rename actions 1232
 Copy MOEs from Base option 1233
 Copy MOEs TO Children option 1233
 generate N2 matrix 1233
 Link Wizard 1237
 measure of effectiveness (MOE) stereotype 1233
 measures of effectiveness (MOE) stereotype 1230,
 1233
 profile 225, 1230, 1243
 project type 1230
 special menu options 1231
 test bench statechart 1233
 toolkit 1231
 trade analysis 1232, 1234
 wizards 1231
Harmony process 1228
HasIDEInterface property 309
Header file 943
Header property 1015
Headers 949
Heaps 970
Help 1452
 custom file 400
 generate support request 1460
help command 1154
Helper applications 488
 .hep files 494
 adding links to VBA macros 497

- deleting links to 489
 - examples of menu command links 492, 493
 - for MODAF 1334, 1366
 - icons on the Helpers dialog box 489
 - linking to external programs 490
 - linking to Rhapsody Tools menu 490
 - modifying 496
 - modifying links to 496
 - moving position of links on Rhapsody Tools
 - menus 489
 - samples 494
 - Helper utilities 1309
 - Helpers dialog box 489
 - HideCellNames property 253
 - HideEmptyRowsCols property 253
 - Hierarchical
 - directory structure 280
 - relation type 435
 - requirements 386
 - Hierarchical mode 279, 280
 - changing to flat 281
 - Hierarchy of reactive classes 787
 - History connector 735
 - Home page
 - changing using the GUI 1172
 - changing using the webconfig.c file 1192
 - Horizontal message 685
 - HTML 1195, 1200
 - browser to examine exported models 1421
 - viewing reports in 1201
 - Hyperlinks 52
 - add new 54
 - changing tag value 57
 - create 52
 - creating in browser 54
 - deleting 58
 - description 52
 - editing 55
 - following 55
 - icons for targets 54
 - limitations 57
 - Link target field 53
 - tag values 56
 - target files 53
 - Text to display field 53
- I**
- IBM Rational Rose 1405
 - IDE menu options 307
 - IDEConnectParameters property 309
 - IDEInterfaceDLL property 309
 - IDF 236
 - IDFProfile 226
 - Image View field 111
 - Images
 - associate with element 349
 - formats 383
 - ImpIncludes property 949, 1029
 - Implement
 - base classes 610
 - relation 559
 - Implement Base Classes option 611
 - ImplementActivityDiagram property 651
 - Implementation 95
 - code, editing 613
 - composite types property 142
 - file structure 949
 - phase 73
 - property 559
 - Implementation files 1066
 - inline keyword 1067
 - ImplementationEpilog property 960
 - ImplementationExtension property 1005
 - ImplementationProlog property
 - #ifdef-#endif 960
 - ImplementBaseClasses environment variable 614
 - ImplementFlowchart property 669
 - ImplementWithStaticArray property 559, 972
 - Implicit
 - contracts 116
 - object 529
 - Import as External 1025
 - ImportDefineAsType property 1015
 - Importing 1419
 - Eclipse projects 293
 - from Rose 1406, 1408, 1409
 - incrementally Rose models 1410
 - language selection 1422
 - models 1422
 - requirements into Rhapsody with Gateway 1241
 - Rhapsody model into Teamcenter 1297
 - Rose association classes 1417
 - SA DoDAF 1291
 - Simulink components 1426
 - ImportJavaAnnotation property 1220
 - In property 142
 - Include file
 - for rapid ports 126
 - Include Path field 862
 - configuration 880
 - Include statements
 - reverse engineering 1020
 - IncludeScheme property 610
 - Increment/decrement operators 1276
 - Incremental code generation 923
 - Information
 - conveyed by flow 578
 - element 582
 - Inheritance 545
 - activity diagrams 619, 653
 - adding a level 787
 - block definition diagram 1263
 - from an external class 546

- from external class 546
- overriding for statechart 785
- removing a level 789
- rules for statechart 781
- statechart 780
- stereotypes 410
- superclass 545
- Initial Instances field 877
- Initial value search option 362
- Initial Values 83
- Initialization
 - block 980
 - field, for blocks 844
 - tab 877
- Initialization code field 878
- InitializationBlockDeclaration 980
- Initialize
 - attribute 94
 - static attribute 945
 - static attributes 83
- Initializer box 94
- InitialLayoutForTables 256, 257
- Inject event 1094
- Inlining 795
- InOut property 143
- input command 1154
- Inserting projects 263
- Installation
 - root directory 449
 - systems engineering 1223, 1224
 - verifying DoDAF 1326
 - verifying MODAF 1368
- Instance group 722
- Instance groups
 - adding instances to 725
 - creating 725
 - deleting 725
 - modifying 724
 - resetting 726
- Instance lines
 - creating 679, 1115
 - pop-up menu 682
- Instances 676
 - adding to instance group 725
 - anonymous 965, 966
 - component 1126
 - component in deployment diagrams 892
 - creating 696
 - names of 1126
 - navigation expressions 1127
 - specifying the value of 535
 - specifying value 535
- Instantiation
 - class templates 77
- Instrumentation
 - field 880
 - header file 944
 - implementation file 951
 - mode 880, 1084
 - selective 882
- Instrumentation Mode 209
- Instrumentation Scope field 883
- Integrate
 - CodeTEST 881
- Integrity checks 901
- IntelliVisor 479
 - dismissing 480
 - information for UCDs 485
 - information in sequence diagrams 482
 - information, activity diagrams 483
 - information, collaboration diagrams 481
 - information, component diagrams 482
 - information, deployment diagrams 482
 - information, OMDs 480
 - information, statecharts 483
 - information, structure diagrams 485
 - invoking 480
- Interaction occurrence 701
 - creating 701
 - menu 702
- Interaction Operators
 - creating 704
- Interactive mode
 - command-line interface 1434
- Interface 857
 - add new 122
 - component 871
 - component, creating 871
 - provided 115
 - provided for rapid ports 126
 - required 115
 - required for rapid ports 126
- InterfaceGenerationSupport property 987
- Interfaces 984
 - automatic creation of 1233
 - block definition diagram 1263
 - C language 985
 - command-line 1433
 - in RiC 984
 - naming conventions 273
 - naming guidelines 272
 - realizing 984
 - service contract 303
 - virtual tables 984
- Internal
 - checks 899
 - event 751
 - hyperlink 52
 - text editor 415
- Internal block diagrams 1233, 1261, 1267
 - drawing tools 1268
- Internal editor
 - auto indenting text 420
 - bookmarks 428

- color-coding 418
- keyboard shortcuts 421
- printing 429
- property 418
- searching 428
- split views 425
- using Undo/Redo 427
- view options 418
- window properties 417
- Interrupt handler
 - static memory 972
 - using 794
- Interrupt-driven framework 236
- Intertask communication
 - generating code without 923
- Invoke IntelliVisor 480
- InvokeMake property 926
- Invoking
 - DOORS 1390
- IS_COMPLETED() macro 650
- IS_IN query 790
- IsCompletedForAllStates property
 - IS_COMPLETED macro 650
 - local termination code 742

J

- JAR files
 - generating 1214
- Java
 - annotations 1215
 - generating JAR files 1214
 - reference model 1221
 - static blocks 1213
 - static import 1212
- Java 5 concepts 1215
- Java language 1, 1207
 - annotations 1215
 - call stack 1113
 - code generator symbols 1064
 - component file type 864
 - composite types 136
 - enums 151
 - inheriting from an external class 546
 - initialization blocks 980
 - interfaces 611
 - modeling constructs 146
 - no flow port support 596
 - packages 223
 - projects in Eclipse 163, 292
 - reverse engineering 997
 - roundtripping 107
 - SA post processing plug-in 1294
 - Send Action State code generation 764
 - specify include classpath 1013
 - static models 144
 - template limitation 161

- JavaAnnotation property 1221
- JavaAnnotations 1215
 - adding 1217
 - code generation 1220
 - creating 1216
 - limitations 1221
 - reverse engineering 1220
 - sample code 1219
 - using 1218
- Javadoc comments 1207
- JavaDocProfile 1209
- Join synchronization bars 637, 638
- Join transitions
 - activity diagram 635
 - statecharts 754
- Joins 748
 - flow chart 668
 - MISRA rule 33 748
- JPEG 383
- Junction connector 735
- Junction connectors
 - activity diagram 635
 - flow charts 668

K

- Keyboard
 - accelerator keys 1449, 1451, 1453
 - application accelerators 1451
 - code editor accelerators 1455
 - mnemonics 1449, 1458
 - modifiers 1451
 - shortcuts 80, 421, 422, 1456
 - shortcuts, standard Windows 1451
- Keywords
 - \$(Name) 978
 - \$Arguments 977
 - \$Attributes 977
 - \$Base 977
 - \$Relations 977
 - activity 1254
 - additional user-defined (reverse engineering) 1017
 - compiler specific 960
 - compiler-specific 949
 - expanding properties 978
 - in standard operations 977
 - inline 1067
- Kind box 135
- Knob tool 800

L

- Labels 342
 - assigning 342
 - display options 314
 - in Asian languages 342
 - mode 344

- removing 344
- search option 362
- show compartment 110
- supported elements 342
- transition 634
- transition, statecharts 749
- Languages 163
 - Asian supported 342, 362, 363
 - independent type 144
 - more than one in projects 288
 - selecting during import 1422
 - type 138, 139
 - type in a nested class 1070
 - WSDL 300
- Layout menu
 - Grid option 450
 - Replicate option 467
- Layouts
 - matrix views 249
 - table view 242
- Leaf state 734
- LED control 807
- LED tool 800
- Legacy code 1010, 1044
 - reverse engineering 599, 998
- Level Indicator control 804
- Level Indicator tool 800
- Libraries 316, 1088
 - add 1037
 - for SysML models 1227
 - model 1264
 - units 1264
- Libraries box
 - in configuration Settings tab 880
 - in Features dialog box 861
- Licensing 163
 - SA Importer 1291
- Lifecycle 535
- Lifeline
 - animation 1135
- Limitations
 - activity diagrams 653
 - actors characteristics 938
 - animation 1109
 - Browse From Here browser 323
 - called behaviors 646
 - CM for project list files 271
 - code generation 987
 - code generation for activity diagrams 653
 - code generation for flow charts 669
 - comments 1052
 - controlled files 376
 - CORBA 146
 - customized diagrams with custom elements 506
 - decomposition 703
 - DiffMerge 271, 378
 - DoDAF 1325
 - Eclipse workflow integration 299
 - external elements 610
 - Favorites browser 327
 - flow diagrams 669
 - flows 584
 - Gateway 1242
 - hyperlink tags 57
 - importing diagrams (reverse engineering) 1047
 - JavaAnnotations 1221
 - joins and MISRA rule 33 748
 - locating elements on a diagram 334
 - no Java flow ports 596
 - no Linux forward and back 31
 - panel diagrams 837
 - project 271
 - properties for only active projects 271
 - Rose import views 1405
 - roundtripping 1060
 - roundtripping for Send Action State 764
 - roundtripping restricted mode 1072
 - search and replace 378
 - show transitions states on animated sequence diagrams 1119
 - Simulink 1430
 - static memory allocation 974
 - subactivities 1258
 - swimlanes 645
 - Teamcenter and Rhapsody 1299
 - templates 161
 - Undo 235
 - Web-enabled devices 1171
- Line numbers in code 933
- Lines
 - anchor 390
 - changing shape 443
 - maintaining shape 457
 - selection handles 453
- Link modules 1396
 - DOORS 1402
- Link Switches field 865
- Link target field 53
- Link with editor 205
- Linking data in DOORS 1395
- LinkModuleName property 1396
- Links 561, 852, 1269
 - creating in OMD 562
 - generating code for 567
 - in collaboration diagrams 852
 - messages 855
 - modify features 853
 - structure diagram 841
- Linux
 - no forward & back navigation 31
 - viewing reports 1201
- List of Books
 - property definitions list 47
- Lists, edit 80

- LmLicenseFile property 1388
 - Load
 - backup 240
 - log 285
 - option settings 717
 - units 282
 - load.log file 285
 - Local
 - heaps 970
 - host 1085
 - tag 414
 - termination code 742
 - Local termination rules 650
 - Local termination semantics 649
 - LocalizeRespectInformation property 1075
 - LocalTerminationSemantics property
 - activity diagrams 630
 - flow charts 664
 - statecharts 741
 - Locate element from code 934
 - Locate On Diagram command 331
 - Log tab 18, 1048
 - LogCmd 1155
 - Logical file type 865
 - Logical files mode 1019
 - Loop activity flows 667
 - Loop transitions 634
 - Lost constructs 1057
- M**
- MacroExpansion property 1055
 - Macros 1055
 - CALL 695
 - CALL_INST 695
 - CALL_SER 695
 - GEN, standard operations 975
 - IS_COMPLETED() 650
 - OM_RETURN 694
 - to examine models 1419
 - VBA 497
 - Main Diagram field
 - actor 522
 - block 843
 - MaintainWindowContent property 14
 - mainThread 1098
 - MakeFileContent property 926
 - Makefiles 924, 925
 - Manage Web-enabled devices 1169
 - Map to Package option 1026
 - Mapping rules for Rose 1412
 - Marshalling 975
 - Mathematical relationships 1270
 - MATLAB 1430
 - MATLAB MEX compiler 1430
 - Matrices 241, 1321
 - Matrix Display control 805
 - Matrix Display tool 800
 - Matrix views 241, 251
 - binding view and layout 256
 - cell element types 249
 - customize for MODAF 1352
 - export data 261
 - from and to values 249
 - layouts 249
 - manage data 261
 - toggle empty rows filter 253
 - MaximumPendingEvents property 972
 - Measures of effectiveness (MOE) 1230, 1233
 - Mechanistic design 72
 - Memory
 - allocation 973
 - allocation algorithm 973
 - fragmentation 970
 - Memory allocation algorithm 973
 - Memory pools 971
 - Merging existing packages 1046
 - Message diagram 34
 - Message diagrams 671, 1493
 - Message groups 727
 - adding messages to 729
 - creating 729
 - deleting 731
 - modifying 730
 - removing messages from 729
 - Message icon 683
 - Message Type field 687
 - Messages 856, 1120
 - arguments, displaying 684
 - arrival times 717
 - code output 931
 - code to pass 1275
 - collaboration diagram 855, 856
 - collaboration diagrams 847
 - copying 689
 - creating 683
 - cutting 689
 - data 792
 - exchanging, port 129
 - excluding from a comparison 719
 - formal parameters 792
 - found 677
 - horizontal 685
 - link 855
 - lost 677
 - moving 689
 - names 684
 - numbering 847
 - pasting 689
 - pop-up menu 686
 - reply 676
 - reverse link 855
 - selecting 688
 - sequence diagrams 676, 683

- slanted 685
- statechart 758
- suppressing in animated sequence diagrams 1122
- tab for associations 854
- to self 685
- to self, displaying 1130
- tracer 1166
- types 689
- Metaclasses 503
- Meter control 803
- Meter tool 800
- Methodology
 - development 70
- Microsoft
 - Excel 1232, 1234
 - PowerPoint 1195, 1200
 - Word 345, 1195, 1200, 1205
- Ministry of Defence Architecture Framework 1333
- MISRA rule 33 748
- MISRA98
 - profile 226
- MISRA-C 1998 748, 921
- MKS Source Integrity 212
- Mnemonics 1449, 1458
- MODAF 2, 1333
 - Acquisition viewpoint 1338
 - All Views viewpoint 1337
 - architectural conformance 1369
 - artifacts 1347
 - checking your model 1369
 - creating a project 1347
 - CustomizableTableAndMatrixLayoutsPkg 1354
 - CustomizableTableAndMatrixViewsPkg 1358
 - CustomizedStereotypesPkg 1355
 - customizing table and matrix views 1352
 - Dependencies Linker 1366
 - Domain Specific Language 1347
 - Drawing toolbar 1369
 - general troubleshooting 1368
 - helper applications 1334, 1366
 - Java plug-ins 1369
 - ModafReport.tpl 1362
 - Network Enabled Capability (NEC) 1333
 - Operational viewpoint 1337
 - pack 1334
 - products 1339
 - profile 226, 1334
 - quality of service requirements 1341, 1345
 - ReporterPLUS template 1360
 - Strategic viewpoint 1337
 - Systems viewpoint 1338
 - Technical viewpoint 1338
 - troubleshooting with Check Model 1369
 - troubleshooting, Dependencies Linker 1367
 - troubleshooting, general 1368
 - troubleshooting, ReporterPLUS 1364
 - UML 1334
 - verifying installation 1368
 - viewpoints 1335
 - views 1339
- ModafReport.tpl 1362
- Mode
 - activity diagram 650, 741
 - analysis 674
 - current 11
 - design 674
 - drawing 440
 - Flat 280
 - flat 279
 - Hierarchical 280
 - hierarchical 279
 - instrumentation 880
 - operation 433
 - pinned 44, 48
 - rapid 125
 - repetitive drawing 441
 - stamp 461
 - statechart 649, 733, 741
 - Workbar 12
- Model as language types option 1034
- Model browser 213
- Model Update Failure option 1045
- Model Updating tab 1046
- ModelCodeAssociativityFineTune property 1061
- ModelCodeAssociativityMode property 1061
- Model-driven Development (MDD) 1053, 1371
- Modeled annotations 385
- Modeled operation 651
- Modeling class type 1037
- Modeling policy 1025
- Modeling toolbar 37
- Models 223
 - actor element 573
 - adding new elements 176
 - adding units to 182
 - animation 1112
 - animation (partial) 1088
 - automotive extended execution 1374
 - checking 191, 902, 903
 - classes 1037
 - connecting to from the Web 1174
 - connecting to from the Web, troubleshooting 1175
 - creating actions 1275
 - data analysis 241
 - define environment 1249
 - deleting elements 471
 - deleting items 324
 - designing 69
 - elements 503
 - elements aggregation 557
 - elements associations 549
 - elements classes 542
 - elements inheritance 545
 - elements, composite classes 543

- errors 899
 - examining 1419
 - examining in HTML browser 1421
 - execution 1275
 - exporting to XMI 1420
 - exposing to the Web 1172
 - find elements 361
 - importing XMI 1422
 - in a Web browser 1289
 - library for SysML 1227
 - locating type of items in 232
 - management views 67
 - minimum requirement 69
 - naming guidelines 272
 - one-way association 555
 - packages 544
 - print to screen 1276
 - properties 239
 - search and replace 187, 227
 - search in 29
 - searching 358
 - searching for text 360
 - simplified 991
 - specifying with Rhapsody 69
 - SysML stereotypes for elements 1227
 - validation 1080
 - warnings 899
 - Web-enable 1286
 - XMI 1419
 - Modes 1494
 - animation 983, 1111
 - batch 1059
 - categories 1471
 - DMCA 1061
 - flat 1486
 - logical files 1019
 - recursive analysis 1020
 - silent 1092, 1111
 - watch 1092, 1111
 - Modifiers 1451
 - attributes 82
 - Constant 82
 - destructors 97
 - primitive operations 87
 - Reference 82
 - Static 82
 - Modifying
 - data types 83
 - instance groups 724
 - message groups 730
 - reference class 1033
 - ModuleNameFromProject property 1398
 - Monitor
 - Web-enabled devices 1169
 - Mouse, select and editing with 427
 - Moving
 - control point 456
 - element in graphic editor 457
 - elements between projects 267
 - elements in browser 345
 - messages 689
 - synchronization bar 639
 - MSVC60 dialect 1018
 - Multiple projects 262
 - Multiple selection 12
 - Multiplicity
 - array index 763
 - attributes 82
 - ports 98, 120
 - Multiplicity field
 - association 553
 - block 844
 - object 533
 - Multi-threaded architecture 1
 - Mutator
 - implementation file 950
 - MutatorGenerate property 142
 - Mutators 79, 82
 - Mutex 972
- ## N
- Name search option 362
 - Name/value pairs
 - for Web GUI 1181
 - Names 272
 - boxes and arrows 444
 - class, default 75
 - guidelines for model elements 272
 - instance 1126
 - message 684
 - project 238
 - roles for classifiers 682
 - special characters in 272
 - thread in animation 1098
 - thread in tracing 1146
 - view field 1179
 - Namespace 341
 - Naming conventions
 - for Rhapsody 272
 - for Rose 1408
 - Navigable field 533, 554
 - Navigate
 - between DOORS and Rhapsody 384
 - from DOORS to Rhapsody 1390
 - hyperlinks 55
 - to reference SD 702
 - Navigation
 - customizing 1180
 - personalized 1193
 - Nested classes 78
 - Nested packages 78
 - exporting to DOORS 1391
 - Nested types 78

- NetCentric 300
 - profile 226, 300
 - WSDL output 303
 - Network Enabled Capability (NEC) 1333
 - New Attribute option 573
 - New element types 502
 - New Operation option 573
 - New Reception dialog box 88
 - New Statechart option 573
 - New Term 406
 - NO_OUTPUT_WINDOW 1049
 - Nodes 339, 889
 - call operation 621
 - deployment diagrams 889
 - features 891
 - owner 890
 - Notes 384
 - converting to comments 389
 - Rational Rose 395
 - NotifyOnInvalidatedModel property 1071
 - Null transitions 754
 - activity diagrams 649
 - statecharts 754
 - Null trigger 754
- O**
- Object analysis 71
 - Object Management Group 65
 - Object Management Group (OMG) 1419
 - Object model diagrams 65, 66
 - adding operations and attributes 528
 - automatically populating 435
 - compared to structure diagram 839
 - creating 33, 433
 - drawing icons 528
 - editor 6
 - files 284
 - flowports in 596
 - icon in browser 318
 - importing by reverse engineering 1000, 1001, 1047
 - IntelliVisor information 480
 - perform trade analysis in Harmony 1234
 - Object nodes 631
 - associated with class 632
 - ObjectIsSavedUnit property 280
 - Objects 1, 336, 529, 1233
 - changing order of 845
 - changing the order of 535
 - concurrency 843
 - converting types 533
 - creating in browser 336
 - creating in OMD 530
 - dependencies between 568
 - destroying 696
 - features 532
 - generating code for 534
 - in activity diagrams 622
 - in structure diagram or OMD 842
 - linking to ports 131
 - modifying 843
 - multiple 851
 - relations, show all 101
 - show status 1158
 - specifying the value of 535
 - StateBlock 1384
 - structure diagrams 840
 - types of 529
 - OM_RETURN macro 694
 - OMG 65
 - SysML profile 226
 - testing profile 226
 - OMMemoryPoolsEmpty() operation 973
 - OMReactive class 751
 - OMSimulinkBlock 1427
 - On/Off Switch control 808
 - On/Off tool 800
 - Only from file list option 1020
 - Only header file with the same name option 1019
 - Open
 - Active Code View 26
 - animated sequence diagram 1115
 - browser window 314
 - editor 107
 - existing diagrams 434
 - main diagram 108
 - multiple Features dialog boxes 45
 - multiple projects 262
 - OpenDiagramsWithLastPlacement property 283
 - OpenWindowsWhenLoadingProject property 283
 - parent statechart 773
 - project 227
 - project with workspace information 283
 - subactivity diagram 629
 - submachine 773
 - workspace 283
 - Open Main Diagram option
 - actor 573
 - Open Reference Sequence Diagram option 682, 702
 - OpenDiagramsWithLastPlacement property 283
 - OpenWindowsWhenLoadingProject property 283
 - Operation bodies search option 362
 - Operation mode 433
 - Operational view (DoDAF) 1303
 - Operational viewpoint (MODAF) 1337
 - Operations 691
 - adding to events 751
 - adding to OMD 528
 - adding to use case 519
 - call 623, 1257
 - calls 883
 - calls, during animation 695, 1107
 - changing order in generated code 953
 - code generation in activity diagram 651

- contracts allocation 1235
 - display options 113
 - generated 936
 - implementation file 950
 - modeled 651
 - modes 432
 - naming conventions 273
 - primitive 85
 - receptions 88
 - roundtripping 1066
 - roundtripping triggered 1067
 - standard keywords in 977
 - statechart 758
 - that cannot be undone 235
 - triggered 91, 690, 752
 - Operations tab 84
 - Operator, overloading 960
 - Or state 734
 - code generation 741
 - local termination code, flat 742
 - Order
 - of types, changing 147
 - of variables 337
 - Ordered to-many relations 968
 - Organize tree 314
 - Orthogonal
 - relation type 435
 - OSEK Adaptor 1374
 - hardware and software 1375
 - Out property 143
 - Output 227
 - command 1156
 - consistency checking results 1232
 - reports 221
 - search results 187
 - Output window 17, 166
 - Animation tab 25
 - Build tab 22
 - Check Model tab 19
 - Configuration Management tab 25
 - display search results 359
 - displayed in Eclipse 166
 - Log tab 18
 - Search Results tab 26
 - Output Window option 1049
 - OV-1 High Level Operational Graphic 1317
 - OV-2 Operational Node Connectivity 1317
 - OV-4 Organizational Relationships Diagram 1317
 - OV-5 Operational Activity Diagram 1317
 - OV-6a Operational Rules Model 1317
 - OV-6c Operational Event-Trace Description
 - diagram 1317
 - OV-6c Operational State Transition Description
 - diagram 1317
 - OV-7 Logical Data Model 1317
 - Overridden properties view 321
 - Overwriting existing packages (during reverse
 - engineering) 1046
 - Owners
 - actor 522
 - node 890
 - OXF 921
 - extended C 1374
 - ExtendedC for automotive environments 225
 - in a C project 236
 - libraries 1089
- ## P
- PackageIsSavedUnit property 280
 - Packages 335, 523, 544
 - adding 176
 - as containers 341
 - assigning to a deployment diagram 897
 - converting to profile 399
 - creating hierarchy in C 605
 - creating in browser 335
 - creating in OMD 544
 - cross-package initialization 941
 - default systems engineering 1225
 - dependencies 568
 - DOORS 1391
 - drag and drop 345
 - elements 223
 - elements types 335
 - files 284
 - in requirements diagrams 1244
 - in UCD 523
 - naming guidelines 272
 - nested 78
 - relations, show all 101
 - roundtripping supported modifications 1067
 - service 302
 - setup DoDAF 1311
 - stereotypes 406
 - storing in directories 280
 - SysML profile 1227
 - units 274
 - WSDL stereotyped 303
 - Pan 38
 - Panel diagrams 797
 - animation 797, 799
 - attribute types 816
 - binding 813
 - binding (mapping) table 816
 - Bubble Knob control 801
 - Button Array control 810
 - Button Array tool 800
 - caption for Push Button control 817
 - changing control flow 817
 - changing display name options 836
 - changing graphical properties of a control
 - element 819
 - changing settings 817

- color schemes for Matrix Display and Digital Display controls 817
- Control Properties dialog box 814
- creating 34, 433, 799, 800
- Digital Display control 806
- Digital Display tool 800
- drawing icons 800
- features 797
- Gauge control 802
- Gauge tool 800
- graphical settings for Button Array control 835
- Knob tool 800
- LED control 807
- LED tool 800
- Level Indicator control 804
- Level Indicator tool 800
- limitations 837
- Matrix Display control 805
- Matrix Display tool 800
- Meter control 803
- Meter tool 800
- minimum and maximum values for controls 817
- On/Off Switch control 808
- On/Off switch shape styles 817
- On/Off tool 800
- properties for Bubble Knob control 820
- properties for Digital Display control 830
- properties for Gauge control 822
- properties for LED control 831
- properties for Level Indicator control 828
- properties for Matrix Display control 830
- properties for Meter control 825
- properties for On/Off Switch control 832
- properties for Slider control 833
- Push Button control 809
- Push Button tool 800
- Select tool 800
- Slider control 812
- Slider tool 800
- Text Box control 811
- Text Box tool 800
- Parallel development 212
- Parameters
 - formal message 792
 - Webify 1172
- Parametric diagrams 1270
 - adding equations 1274
 - allocation 1271
 - binding connector 1271
 - block 1271
 - constraint binding 1273
 - constraint block 1271
 - constraint blocks for 1272
 - constraint parameters 1271
 - constraint properties 1272, 1273
 - constraint property 1271
 - create package 1271
 - dependency 1271
 - drawing icons 1271
 - problem satisfaction 1271
 - showing constraints 1273
- params
 - pseudo-variable 757
- params-> pseudo-variable
 - message parameters 792
- Parent statechart 773
- ParserErrors property 1071
- Parsing Errors option 1045
- Part 476, 529, 543, 1500
 - and composite class 543, 557, 558
 - decomposition 703
 - relation to whole 844
- Partial animation 1088
 - per configuration 1089
- Partially constructive diagram 68
- Partition line 701
- Parts
 - browser icon 316
- Path field 588, 869
 - file 864
 - folder 869
- Paths 341
 - names in commands 1435
 - relative 281
 - specifying physical 399
- Pause Animation command 1094
- pc_server.dxl 1388
- Personalized bottom navigation 1193
- Personalized Navigation page 1179
- Perspectives 168, 197
 - Debug 167, 208, 209
 - Modeling 166
- PI variable 337
- Point
 - adding to line 443
- Pooling events 971
- Populate Diagram 1243
- Populate Diagram feature 435
- Ports 118
 - API for C++ 129
 - atomic flow 596
 - automatic creation of 1233
 - behavior attribute 120
 - C language 985
 - C language optimization 986
 - code generation(C) 133
 - communicating with ports with multiplicity 130
 - connecting 125
 - connecting objects via ports 131
 - creating 118
 - creating programmatically 132
 - definition 115
 - display in the browser 125
 - display options 123

- exchanging messages 129
- features 119
- field 112
- flow 596
- flow for StateBlock 1385
- implicit contracts 116
- in structure diagrams 841
- in SysML 1227
- linking objects via ports with multiplicity 131
- linking to owning instance 132
- links 1269
- listing 232, 1201
- multiplicity 120
- non-atomic flow 597
- properties 124
- rapid 116, 125, 985
- reversed attribute 120
- SDLBlock behavior 311
- service 984
- show all 1267
- show in views 254
- specifying contract 121
- standard 1263, 1269
- tab 98
- Ports tab
 - behaviors 98
 - contracts 98
 - multiplicity 98
 - reversed 98
- PowerPoint 1195, 1200
- PreCommentSensibility property 1052
- Predefined checks 899
- Predefined type 144
- PredefineIncludes property 1071
- PredefineMacros property 1071
- Preprocessing
 - reverse engineering 1011
 - symbol 1014
- Primary model elements 341
- Primary templates 156
- Primitive operation 691
- Primitive operations 85
- Print
 - diagrams 379
 - from internal editor 429
- Print to screen 1276
- Priority
 - thread 1099
 - transition 776
- Private
 - attributes 79
 - event 751
- Problem 1227
- Problems
 - found by Check Data 1400, 1401
 - Simulink 1428
- Process tab 1045
- Profiles 2, 224, 225, 396
 - AdaCodeGeneration 225
 - add to existing project 233
 - adding Rhapsody 329
 - as packages 223
 - as settings 328
 - AutomotiveC 225, 1371, 1374
 - AUTOSAR 225, 1371
 - backward compatibility 398
 - browser icon 317
 - compatibility settings 328
 - converting to package 399
 - creating your own 499
 - default 225
 - diagrams available for 33
 - DoDAF 225, 1301
 - enabling access to custom help file 400
 - FixedPoint 225
 - FunctionalC 7, 33, 34, 225, 585
 - Harmony 225, 1230, 1235, 1237, 1243
 - IDF 226
 - JavaDocProfile 1209
 - MISRA98 226
 - MODAF 226, 1334
 - NetCentric 226, 300
 - new term 406
 - properties 399
 - RespectProfile 226
 - re-using customized 501
 - Rhapsody's predefined 225
 - RoseSkin 226
 - Schedulability, Performance, and Time (SPT) 226, 304
 - SDL 226
 - Simulink 226, 1427
 - SPARK 226
 - SPT 226, 304
 - StateBlock 226, 1384
 - stereotypes 406
 - SysML 226, 1225, 1226, 1235, 1237, 1243, 1264
 - Testing 1233
 - TestingProfile 226
 - when not needed 397
- Programs
 - command-line Rhapsody 1433
 - external 491
 - files from Word or Excel 317
 - ReporterPLUS 1195
- Project Overview diagrams 1317
- Projects 170, 223, 262
 - active 262, 263, 264
 - add elements 233
 - add profile to existing 233
 - adding new elements 176
 - adding project to list 270
 - archiving 240
 - AUTOSAR 1372

- autosave 238
- backing up 239
- build Eclipse 206
- closing 238
- closing all 268
- copying elements 265
- copying in multiple 263
- creating 224
- creating new Rhapsody 224
- creating units 276
- debug Eclipse 208
- directory structure 279
- disassociating Eclipse from Rhapsody 299
- DoDAF 1314
- Eclipse 292
- Eclipse IDE 193
- editing Rhapsody 233
- elements 223
- files and directories 284
- flat directory structure 280
- folder 262
- Harmony 1230
- incremental save 237
- inserting into other project 263
- keyboard shortcuts 1452
- large 342
- lifecycle instance values 535
- limitations 271
- loading backup 240
- managing lists 269
- migration 288
- MODAF 1347
- multi-language 288
- multiple 262
- NetCentric 300
- new 271
- opening 227
- opening project list 269
- organizing 313
- referencing in multiple 263
- removing from project list 270
- renaming 238
- restoring 240
- Rhapsody profiles 225
- saving 237
- saving project in project list file 269
- SOA 300
- systems engineering 1223
- tool icons 29
- tools 28
- types 224
- unit 274
- validation 1284
- without profiles 397
- workspace 271
- Prolog
 - header file 944
 - implementation file 949
- Properties 239, 1393, 1396
 - AcceptChanges 1066, 1069, 1072, 1212, 1214
 - active project's displayed 271
 - ActiveThreadName 1099
 - ActiveThreadPriority 1099
 - activity diagram settings 1255
 - ActivityReferenceToAttributes 652
 - AdditionalBaseClasses 976
 - AdditionalKeywords 1017
 - AdditionalNumberOfInstances 971
 - AddressSpaceName 761
 - affecting diagram editors 440
 - Aggregates 1370
 - AlternativeDrawingTool 508
 - AnalyzeGlobalFunctions 1031
 - AnalyzeGlobalTypes 1031
 - AnalyzeGlobalVariables 1031
 - AnalyzeIncludeFiles 1020
 - AnimateSDLBlockBehavior 311
 - animation 1130
 - AutoCopied 399
 - AutoReferences 399
 - AutoSaveInterval 238
 - AvailableMetaClasses 503
 - backup 239
 - BackUps 239
 - BaseNumberOfInstances 971
 - BlockIsSavedUnit 280
 - BrowserIcon 506
 - change directory scheme 279
 - ClassCentricMode 674
 - ClassCodeEditor 107, 935
 - ClassIsSavedUnit 280
 - CleanupRealized 674
 - CodeGeneratorTool 991, 1052
 - CollectMode 1054
 - ComplexityForInlining 796
 - ComponentIsSavedUnit 280
 - ConstantVariableAsDefine 1015
 - constraint in parametric diagrams 1272
 - ContainerSet 559
 - CPPCompileCommand 928
 - CreateDependencies 1020
 - CreateReferenceClasses 1030
 - CustomHelpMapFile 402, 403
 - CustomHelpURL 400, 403
 - DataTypes 1037
 - DefaultDirectoryScheme 279
 - DefaultProvidedInterfaceName 126
 - DefaultReactivePortBase 126
 - DefaultReactivePortIncludeFile 126
 - definitions 47, 103, 124
 - DescriptionTemplate 957, 1209, 1211
 - DiagramIsSavedUnit 275, 280
 - DiagramsToolbar 507, 509
 - displaying 46

- DisplayMessagesToSelf 1130
- DisplayMode 314
- DrawingShape 507
- DrawingToolBar 506, 509
- DrawingToolIcon 506, 507
- DrawingToolTip 507
- Eclipse Workbench 297
- EditorCommandLine 867, 936
- EventGenerationPattern 764
- EventToPortGenerationPattern 764
- ExcludeFilesMatching 1009
- ExecutionModel 1377
- expanding with keywords 978
- FileIsSavedUnit 280
- FileName 546
- Files (for reverse engineering) 1008
- flowKeyword 576
- for black-box testing 1134
- for composite types 142
- FullTypeDefinition 142
- GeneratedCodeInBrowser 936, 1070
- GenerateDirectoryPerModelComponent 1026
- GeneratePackageCode 924
- GeneratorRulesSet 996
- Header (for reverse engineering) 1015
- HideCellNames 253
- HideEmptyRowsCols 253
- ImpIncludes 1029
- ImplementActivityDiagram 651
- Implementation 559
- Implementation of composite types 142
- ImplementationExtension 1005
- ImplementFlowchart 669
- ImplementWithStaticArray 559, 972
- ImportDefineAsType 1015
- ImportJavaAnnotation 1220
- In 142
- IncludeScheme 610
- InitializationBlockDeclaration 980
- InitialLayoutForTables 256, 257
- InOut 143
- InterfaceGenerationSupport 987
- internal editor 418
- InvokeMake 926
- IsCompletedForAllStates, IS_COMPLETED
macro 650
- IsCompletedForAllStates, local termination code 742
- JavaAnnotation 1221
- LocalizeRespectInformation 1075
- LocalTerminationSemantics 630, 664
- LocalTerminationSemantics, statecharts 741
- MacroExpansion 1055
- MakeFileContent 926
- MaximumPendingEvents 972
- ModelCodeAssociativityFineTune 1061
- ModelCodeAssociativityMode 1061
- ModuleNameFromProject 1398
- MutatorGenerate 142
- NotifyOnInvalidatedModel 1071
- ObjectIsSavedUnit 280
- OpenDiagramsWithLastPlacement 283
- OpenWindowsWhenLoadingProject 283
- Out 143
- overridden view 321
- PackageIsSavedUnit 280
- ParserErrors 1071
- ports 124
- PreCommentSensibility 1052
- PredefineIncludes 1071
- PredefineMacros 1071
- profile 399
- ProtectStaticMemoryPool 971
- RealizeMessages 674
- ReferenceImplementationPattern 142
- ReflectDataMembers 1043
- RemoteHost 1085
- ReportChanges 1071
- RespectCodeLayout 1052, 1055, 1075
- RestrictedMode 1072
- ReturnType 143
- reverse engineering 1036
- roundtripping 1071
- RoundtripScheme 1052, 1055, 1072
- RTFCharacterSet 1205
- SDLSignalPrefix 310
- searching for 47
- serialization 981
- setting a Web-enabled device 1173
- ShowAnimCancelTimeoutArrow 1122
- ShowAnimCreateArrow 1122
- ShowAnimDataFlowArrow 1122
- ShowAnimDestroyArrow 1122
- ShowAnimStateMark 1119
- ShowAnimTimeoutArrow 1122
- ShowArguments 684
- ShowAttributes 438
- ShowCGSimplifiedModelPackage 991
- ShowContainerElementForPorts 254
- ShowLabels 342
- ShowLogViewAfterBuild 24
- ShowOperations 438
- ShowPorts 124
- ShowPortsInterfaces 124
- simplification 991
- simplifying C code generation 991
- SpecificationEpilog 960
- SpecificationExtension 1005
- SpecificationHeader 956
- SpecificationProlog 960
- SpecInclude 572
- SpecIncludes 1029
- StandardOperations 977
- StrictExternalElementsGeneration 603
- SupportExternalElementsInScope 603

- tab 124, 881
- to set sequence numbers 692
- TriggerArgument 143
- UsageType 609
- UseAsExternal 546, 1030
- UseDescriptionTemplates 1208
- UseIncrementalSave 237
- UseRapidPorts 310
- UseRemoteHost 1085
- VariableInitializationFile 337
- VariableLengthArgumentList 946
- viewing overridden 321
- WebComponents 1173
- WebManaged 1173
- Protected attributes 79
- ProtectStaticMemoryPool property 971
- Provided interface 115
 - for rapid ports 126
- PurgeOnDelete property 1394
- Push Button control 809
- Push Button tool 800
- Pushpin note 392

Q

- Qualified association
 - in OMDs 533, 554
- Qualified to-many relations 968
- Qualifier field 533, 554
- Queries 241
- Queue 751
- Quick Add 411
- quit command 1156
 - ending the tracer session 1168

R

- Random access to-many relations 969
- Rapid external modeling 603
- Rapid ports 985
 - include files 126
 - reactive base class 126
 - required interface 126
- Rational Rose 1405
 - notes in 395
- Rationale 1227
 - block definition diagram 1264
 - requirements diagram 1244
- Reactions In State field 739
- Reactive class 781
 - refining the hierarchy 787
- Reactivity 1
- Real time environment 1
- Realization field 687
 - instance line 680
- Realization Of option 855
- Realization relationship 547, 984

- Realization, implementing base classes 611
- Realize elements of base class 615
- RealizeMessages property 674
- Real-Time Workshop 1425, 1426, 1427
- Rearrange
 - elements 469
 - elements in file 867
- Receptions 88
 - browser icon 338
 - Features dialog box 89
- Rectilinear line 443
- Recursive analysis mode 1020
- Recursive composite 341
- Redo 234
- ReferenceImplementationPattern property 142
- References 29, 230, 264
 - class 1032
 - diagram 701
 - dialog box 358
 - finding 358
 - finding element 358
 - limitations 646
 - modifier 82
 - to constraints 390
 - to element 358
 - to locate elements 343
 - unit using environment variables 281
- References dialog box 271
- Referencing 263
- refine stereotype 1247
- Reflect Data Members option 1035, 1042
- ReflectDataMembers property 1043
- Refresh 61, 246, 474
 - new terms 509
 - table and matrix view 241
 - view option 474
 - Web pages, using the GUI 1172
 - Web pages, using webconfig.c file 1193
- Regenerate
 - code 26
 - configuration files 924
- RegisterUpload function 1194
- RegisterUpload() call 1184
- Re-importing a Rose package 1410
- Relations 99, 328, 967
 - accessor 950
 - adding 328
 - bounded 972
 - fixed 972
 - implementing 559
 - mutator 950
 - of instances 567
 - ordered to-many 968
 - qualified to-many 968
 - random access to-many 969
 - roundtripping 1067
 - show all 101

- show in views 254
 - Show Relations in New Diagram menu command 101
 - tab 99
 - to external element 609
 - to whole field 844
 - to-many 967
 - to-one 967
 - type for populating diagrams 435
- Relationships 524
- Remarks 387
- anchors 389
 - converting to comment 389
 - creating 385
 - deleting 394
 - dependencies 386
 - display options 392
 - editing 387
 - types of 384
- Remote
- managed devices 1169
 - target animation 1086
- RemoteHost property 1085
- Removing
- classes 115
 - element from model 471
 - element from view 471
 - hyperlinks 58
 - label 344
 - level of inheritance 789
 - tags 416
 - user-defined points from lines 443
- Rename 1232
- element 346
 - project 238
- Reorder model element in browser 353
- Repetitive Drawing Mode 441
- Replace 187, 227, 231
- Replicate option 467
- Report on imported items 1008
- ReportChanges property 1071
- ReporterPLUS 1195
- generate list 232
 - launching 1196
 - list of ports 1201
 - MODAF 1360
 - reports to examine models 1419
 - requirements diagrams 1202
 - viewing reports online 1200
- Reports 221, 1195
- customize templates 1196
 - DoDAF 1312, 1323
 - for presentations 1195
 - formal 1195
 - generating from templates 1200
 - HTML 1195, 1200
 - internal output 1205
 - layout for systems engineering 1201
 - overridden properties 1204
 - PowerPoint 1195, 1200
 - prepackaged templates 1196
 - ReporterPLUS 221
 - requirements 1202
 - RTF 1195
 - RTF character set 1205
 - System Architect import 1294
 - system model templates 1201
 - templates 222, 1196
 - text format 1195
 - view overridden properties 1204
 - viewing online 1200
 - Word format 1195
- Reposition windows 11
- Repository 163, 220
- Repository file 284
- Represented field 581
- Represents field 643
- Required interface 115
- for rapid ports 126
- Requirements 1, 70, 384, 1223, 1244
- defining 71
 - defining in use cases 1252
 - dependencies 1245, 1253
 - derive 1245
 - diagrams 1243
 - DOORS 1387
 - hierarchical 386
 - icon in browser 317
 - identifying 70
 - in SysML 1227
 - listed in table views 244
 - reports 1202
 - satisfaction of 1244
 - specialized types 1246
 - specification supports Asian languages 363
 - specifications option 363
 - systems engineering 1240
 - tabular view for SysML 1247
 - trace 1245
 - tracing in use cases 1252
 - verification of 1244
 - view 321
- Requirements diagrams
- allocations in 1244
 - create package in 1244
 - dependencies 1244
 - derivations 1244
 - icon in browser 318
 - output in ReporterPLUS 1202
 - problem 1244
 - rationale 1244
 - use cases in 1244
- Re-route line 443
- RES file 1188
- Reset instance groups 726

Index

- Resize element 456
- Respect 1073
- RespectCodeLayout property 1052, 1055, 1075
- RespectProfile 226
- RestrictedMode property 1072
- Return codes 1437
- Return Value box 687
- Return values, animating 694, 1123
- Returns
 - primitive operations 87
- ReturnType property 143
- Reusable statechart
 - local termination code 741
- Reverse engineering 185, 997
 - #define 1015
 - #if...#ifdef...#else...#endif 1016
 - add wild card expression 1037
 - additional user-defined keywords 1017
 - analyzing list of files 1023
 - analyzing same name header files 1021
 - code respect 1050, 1073
 - comments 1052
 - confirm settings 998
 - directing output 1049
 - enumerated types 1051
 - excluding particular files 1009
 - external elements 1010
 - files 1025
 - Filtering tab 1030
 - flat view 999
 - imported items report 1008
 - importing legacy files 998
 - importing object model diagrams 1000, 1001, 1047
 - include files 1019
 - include statements 1020
 - include/CLASSPATH 1013
 - initializing the Reverse Engineering dialog box 1008
 - Input tab 1019
 - Java 1013
 - JavaAnnotations 1220
 - Javadoc comments 1211
 - legacy code 599, 1044
 - log file 285
 - Log tab 1048
 - lost constructs 1057
 - macros 1055
 - mapping classes 1024
 - mapping classes as types 1037
 - Mapping tab 1024
 - Merging existing packages 1046
 - Model Updating tab 1046
 - options 1006
 - overwrite existing packages 1056
 - overwriting existing packages 1046
 - preprocessing 1011
 - Preprocessing tab 1011
 - preserving comments 1052
 - Process tab 1045
 - properties 1036
 - results 1056
 - static blocks 1212, 1214
 - static import statements 1212
 - templates in C++ 1050
 - to create external elements 598
 - tree view 1000
 - unions 1051
- Reverse messages 856
- Reversed
 - attribute 120
 - ports 98
- ReverseEngineering.log file 285
- Rhapsody 1, 163
 - accessing Web services 1183
 - adding to Web design 1188
 - analysis phase 70
 - and DOORS 1389, 1390
 - and Rose 1406
 - and Tornado 307
 - animation 167
 - Animation toolbar 35
 - backing up 239
 - Browse From Here browser 322
 - browser 313
 - building the target 927
 - checks 907
 - code generation, incremental 923
 - command-line interface 1433
 - common drawing tools 37
 - components-based development 984
 - convert configuration to Eclipse 293
 - creating model from existing Teamcenter project 1298
 - customer support 1459
 - customized workspace 282
 - design phase 72
 - development methodology 70
 - diagram icons 33
 - diagrams 68
 - DiffMerge tool 286, 1299
 - directory structures 280
 - documentation 1463
 - DOORS 384
 - DOORS project 1390
 - drawing icons 10
 - dynamic model-code associativity 925
 - Eclipse Debug perspective 166
 - Eclipse Modeling perspective 165
 - exporting 1419
 - exporting code 179
 - exposing elements on the Web 1172
 - Favorites browser 5, 325
 - Favorites toolbar 32
 - features 2
 - file extensions 178

- filtering out file types 177
- formal reports 1195
- generating code 192
- generating reports 1203
- glossary 1465
- graphic editors 6
- help 1452
- implementation phase 73
- importing 1419
- importing legacy files 998
- integrating with CodeTEST 881
- IntelliVisor 480
- keyboard shortcuts 1456
- license for Platform Integration 163
- linked elements 1396
- locating code in Eclipse 298
- log in Eclipse 198
- MODAF 1335
- MODAF Pack 1333, 1334
- model checking 903
- model, connecting to from the Web 1174
- modeling class types 1037
- Modeling toolbar 37
- modifying elements shared with Teamcenter 1299
- multiple projects 262
- perspectives in Eclipse 168
- plug-in for Eclipse 292
- project 223
- project files and directories 284
- project types 224
- properties for DOORS 1393, 1396, 1398
- properties for Tornado 309
- reports 1195
- repository 163
- reverse engineering 998
- roundtripping 1060
- running animated application without Rhapsody 1141
- running the executable 929
- search facility 1278
- standard toolbar 29
- Statemate block in 1383
- stored information about DOORS 1399
- synchronizing with Statemate 1385
- systems engineering version 1223
- Teamcenter 1295
- testing phase 73
- timeouts 753
- units 182, 212
- utilities 8
- VBA toolbar 35
- Web server 1192
- windows 10, 31
- XMI in development 1419
- Rhapsody handle 1387
- Rhapsody Platform Integration (Eclipse and Rhapsody) 163
 - confirming 164
 - Rhapsody.exe 1433
 - Rhapsody.ini 614
 - animation port, other than default 1440
 - control DMCA 1061
 - favorites list 325
 - General section 1411
 - hide Output window 1049
 - placement of GUI elements 271
 - plug-in information 992
 - RhapsodyCL.exe 1433
 - Role names for classifiers 682
 - Role of field 532, 553
 - ROPES process 70
 - analysis phase 70
 - design phase 72
 - implementation phase 73
 - testing phase 73
 - Rose 1405
 - importing 226
 - importing association classes 1417
 - importing from 1406
 - incrementally importing 1410
 - look-and-feel 1408
 - mapping rules 1412
 - naming conventions 1408
 - re-importing a package 1410
 - selecting elements to import 1407
 - Rose Logical View 1405
 - Rotate on synchronization bar 639
 - Roundtripping 107, 1059, 1061
 - annotations 1064
 - arguments 1066
 - associations 1066
 - attributes 1066
 - automatic 1062
 - classes 1066
 - classes, supported modifications 1065
 - code respect 936, 1059, 1073
 - constructor 1066
 - deletion of elements from the code 1069
 - destructors 1066
 - DMCA 1061
 - edited code 108
 - events 1067
 - forcing 1062
 - friend dependency 1070
 - functions 1067
 - limitations 1060
 - menu option 573
 - operations 1066
 - package supported modifications 1067
 - preserving comments 1052
 - properties 1071
 - relations 1067
 - restricted mode 1072
 - static blocks 1212, 1214
 - static import statements 1212

- supported elements 1060
- templates in C++ 1070
- text edits 108
- triggered operations 1067
- variables 1068
- RoundtripScheme property 1052, 1055, 1072, 1074
- rpyRetVal variable 652
- RTF
 - reports 1195
 - storage format for Asian text 342
 - viewing reports in 1201
- RTFCharacterSet property. 1205
- Rules compiler 992, 993
- RulesComposer 989, 992, 993
- RulesPlayer 990, 992
- Running
 - animation 1085
 - animation automatically 1129
 - animation scripts 1129
 - executable 929
 - Rhapsody from command line 1433
- S**
- Samples
 - helper (.hep) applications 494
- satisfy stereotype 1246
- Save As command 237
- Saving
 - diagrams 277
 - log files 285
 - option settings to a file 717
 - projects 237
 - units in separate directory 279
 - units individually 277
 - viewing preferences 282
 - window preferences 283
 - workspaces 282
- Scale 475
- Scale to Fit button 475
- Scenarios 1233, 1241
 - creating reusable 677
- Schedulability, Performance, and Time (SPT)
 - profile 226, 304
- Scope
 - code generation in activity diagram 652
 - code view editor 932
 - configuration 877
 - derived 877
 - explicit 877
- Screen snapshot 1461
- Scripts 1437
 - animation 1127, 1129
 - Java 1188, 1189
 - on Web server 1182
 - overwriting placeholder text 1188
 - tags 1190
 - uploading 1184
 - VBA 281
- SDL 310
 - profile 226
- SDLBlock 310
 - animation 311
 - behavior ports 311
- SDLSignalPrefix property 310
- Search 187, 188, 227
 - customize criteria 188
 - display results 189
 - working with results 190
- Search and replace 29, 231
 - automatic 360
 - element name 359
 - elements 364
 - in model 358
 - limitations 378
 - text 360
 - using internal editor 428
- Search facility 1278
- Search Results tab 26
- Searching
 - delete item 230
 - elements 361
 - launch methods 228
 - references 230
 - results changes 230
 - within fields 362
- Select Association option 560
- Select Information Flow option 580
- Select tool 800
- Selecting
 - association in OMD 561
 - elements 452
 - messages 688
 - multiple elements 454
- Selection handles 453
- Selective instrumentation 882
- Self transitions 634
- Self-directed message 685
- Send Action 763
- Send Action State 623, 763, 1257
 - code generation 763, 764
 - display options 764
 - event 763
 - graphical behavior 764
 - properties 764
 - target 763
- Sequence
 - field 687
 - UCD 525
- Sequence diagrams 65, 67, 671
 - actor line 698
 - analysis 432
 - animated 1114
 - animated, partial 1090

- animated, show transition states 1118
- auto-create instance lines 1116
- automatic animation 706
- cancelled timeout 676
- comparison 711, 717
- comparison excluding a message 719
- comparison of instance groups 722
- comparison of message arrival times 717
- comparison of message groups 727
- comparison, algorithm 712
- comparison, color coding 714
- comparison, saving options to a file 717
- condition mark 677, 696
- create arrow 676
- created from activity diagrams 1232
- creating 33, 432, 676
- creating message in 683
- data flow 676
- dataflows 699
- deleting 709
- design 432
- destroy arrow 676, 696
- destruction event 677
- drawing icons 676
- editor 6
- enable animating return values 694
- execution occurrence 677
- files 284
- found message 677
- generated from activity diagrams 1260
- icon in browser 318
- instance line 676, 679
- IntelliVisor information 482
- interaction occurrence 677, 701
- interaction operator 677
- interaction operator separator 677
- layout 672
- link wizard 1237
- lost message 677
- message 676, 683
- message types 689
- messages 1120
- navigating to reference 702
- operation modes 432
- part decomposition 703
- partition line 677, 701
- pop-up menu in browser 709
- property for sequence numbers 692
- reference diagrams 701
- reply message 676
- reusable scenarios 677
- sequence numbers 692
- shifting elements with mouse 707
- system border 676, 678
- time interval 676, 698
- timeout 676
- timeouts 697
- Serialization
 - generating methods 981
 - implementation methods 982
 - reactive instances 981
- Service Oriented Architecture (SOA) 300
- Service ports 984, 1261
- Set as Active Component option 873
- SetDeviceName function 1192
- SetHomePageUrl function 1193
- SetPropPortNumber function 1193
- SetRefreshTimeout function 1193
- setSeparateSaveUnit 281
- SetSignaturePageUrl function 1193
- Settings 328, 397
 - backward compatibility profiles 398
 - CGSimplifiedModelProfileC 398
 - CGSimplifiedModelProfileCpp 398
- Settings tab 879
- Severity checks 901
- S-functions 1429
- Shadows 1395
- Shortcuts 1456
- Shortcuts, keyboard 421
- show command 1158
- Show Inherited option 536
- Show Labels 314
- Show Relations in New Diagram pop-up menu
 - command 101, 331, 967, 1370
- Show Roles Labels option 560
- ShowAnimCancelTimeoutArrow property 1122
- ShowAnimCreateArrow property 1122
- ShowAnimDataFlowArrow property 1122
- ShowAnimDestroyArrow property 1122
- ShowAnimStateMark property 1119
- ShowAnimTimeoutArrow property 1122
- ShowArguments property 684
- ShowAttributes 438
- ShowCGSimplifiedModelPackage property 991
- ShowCleanImportData 1411
- ShowContainerElementForPorts property 254
- ShowLabels 342
- ShowLogViewAfterBuild property 24
- ShowOperations 438
- ShowPorts property 124
- ShowPortsInterfaces property 124
- Signature page
 - changing using the GUI 1172
 - changing using webconfig.c file 1193
- Silent animation 1111
- Silent mode 1092, 1111
- Simplification 991
- Simplified models 991
- Simulation 1284
 - setting scope 1285
- Simulink 1426
 - configuration 1430
 - flowports 1429

- profiles 1427
- Simulink integration 1425
- SimulinkBlock 1425, 1427
- Slanted message 685
- Slider control 812
- Slider tool 800
- Smart generation 924
- Snapshot 1461
 - viewing system 535
- Socket mode
 - command-line interface 1434
- Software
 - configuration management 212
 - developers 163
 - development in Eclipse 292
 - prerequisites 163
- Solaris, DOORS settings 1388
- Source of transition 744
- SourceArtifacts 1052, 1075
- Special characters 272
- Specialization parameters 156
- Specification field 388
- Specification files 1066, 1067
- SpecificationEpilog property 960
- SpecificationExtension property 1005
- SpecificationHeader 956
- SpecificationProlog property 960
- Specifications
 - behavior 1284
 - comment 362
 - requirements option 363
 - UML 65
 - WSDL 303
- Specify object values 535
- SpecInclude property 572
- SpecIncludes property 1029
- Spline line 443
- SPT profile 226, 304
- Stamp mode 441
- Standard Diagram toolbar 38
- Standard Headers field 861, 880
- Standard operation 977
- Standard toolbar 29
- StandardOperations property 977
- Statechart mode 649, 733, 741
- Statecharts 65, 67, 432, 734, 736, 1284
 - "And" line in 766
 - And states 734
 - animated 1124
 - basic state 734
 - compound transitions 746
 - condition connector 735
 - created from class for Harmony 1233
 - creating 34
 - default transition 765
 - default transitions 765
 - dependency 736
 - description 739
 - diagram connector 736, 770
 - drawing options 735
 - editor 6
 - else branch 768
 - enabled transitions 776
 - EnterExit point 736
 - events 750
 - events and operations 758
 - flat 880
 - for a use case 520
 - forks 747
 - guards 755
 - history connector 735, 769
 - icon in browser 318
 - inheritance 780
 - inlining 795
 - inlining code 795
 - IntelliVisor information 483
 - IS_IN 790
 - join transitions 736
 - junction connector 735, 770
 - leaf state 734
 - local termination code 741, 742
 - local termination rules 650
 - merge sub-statechart 774
 - messages 758
 - operations 758
 - Or states 734
 - overriding inheritance 785
 - parent 773
 - reusable 880
 - SDLBlock 311
 - semantics 775
 - Send Action 763
 - Send Action State 763
 - send action state 736
 - separate transitions 736
 - serialization 981
 - single-action 778
 - states 734
 - static reactions 783
 - sub-statecharts 736
 - termination connector 735, 775
 - termination state 736
 - timeout triggers 753
 - transition labels 736, 749
 - transitions 744
 - transitions in 744, 749
 - triggers 749
 - Updating EnterExit points 771
- Statemate 1383
 - block 1384
 - Block profile 226
 - flowports 1385
 - model requirements 1383
 - preparing for Rhapsody 1383

- synchronizing with Rhapsody 1385
- troubleshooting 1386
- StateMateBlock profile 1384
- States 734
 - action 621, 658, 1257
 - activity diagrams 621
 - And and code generation 741
 - basic 734
 - block 626, 661
 - creating 736
 - default 1258
 - drawing 736
 - final 649
 - leaf 734
 - names 737
 - Or 734
 - Or and code generation 741
 - Or and local termination code 742
 - send action 736
 - subactivity 1258
 - termination 622, 736, 741
 - transitions between 1259
- Static
 - attribute 945
 - attributes 83
 - modifier 82
- Static architecture 970
 - memory allocation algorithm 973
- Static blocks 1213
- Static import 1212
- Static memory allocation 971
 - algorithm 973
 - conditions 974
 - limitations 974
- Static reactions 783
- Statistics page 1186
- Status of an object 1158
- Stereotypes 396, 406, 1245
 - associating a bitmap 408
 - associating with element 407
 - automotiveC profile 1378
 - browser icon 317
 - change order 408
 - composite requirement 1246
 - conform 1227
 - copying MOEs 1233
 - defining a tag 411
 - defining tag 411
 - deleting 409
 - dependency 1253
 - derive 104, 1245
 - derive requirement 1246
 - display for compartment lists 474
 - display with elements in browser 347
 - extend requirement 1246
 - graphical representation 348
 - inheritance 410
 - measure of effectiveness (MOE) 1230
 - new term 406, 407, 410
 - NewTerm 49
 - problem 1227
 - rationale 1227
 - refine requirement 1246
 - satisfy requirement 1246
 - SDLBlock 310
 - serviceContract 302
 - serviceProvide 302
 - show label 110
 - special 410
 - specialized requirement types 1246
 - SysML allocation 1227
 - SysML blocks 1227
 - SysML model element 1227
 - SysML requirements 1227
 - trace requirement 1246
 - verify requirement 1246
 - view 1227
- store.log file 285
- Straight line 443
- Strategic viewpoint (MODAF) 1337
- StrictExternalElementsGeneration 603
- Structural view 66
- Structure diagrams 66, 839, 1261
 - compared to OMD 839
 - composite class 840
 - creating 33
 - dependencies 841
 - drawing icons 840
 - files 284
 - flows 841
 - icon in browser 318
 - IntelliVisor information 485
 - links 841
 - objects 840
 - ports 841
 - toolbar 840
- Structure type 139
- Structure, modeling as a type 1039
- Structured class 476
- Stub unit 283
- Subactivity 628, 629
 - creating 629
 - creating from action blocks 628
- Subactivity diagrams 629
- Submachine 773
 - opening 773
- Sub-statechart, merge into parent statechart 774
- Subsystems
 - operations allocated to 1235
- Superclass 317, 479, 542, 545
 - add 546
 - prevent code generation for 546
- SupportExternalElementsInScope 603
- Suspend 1128

- SV-1 System Interface Description 1317
- SV-10a Systems Rules Model 1318
- SV-10b System State Transition Description diagram 1318
- SV-10c System Event-Trace Description diagram 1318
- SV-11 Physical Schema diagram 1318
- SV-2 System Communication Description diagram 1318
- SV-4 System Functionality Description diagram 1318
- SV-8 System Evolution Description diagram 1318
- Swimlanes 619, 621, 623, 641, 1257, 1259
 - converted to life lines 1232
 - divider 642
 - dividers, deleting 644
 - frame 642
 - frames, deleting 644
 - limitations 645
 - subactivity limitation 1258
 - viewing in browser 644
- Switches, command-line 1440
- Swimlanes
 - consistency checks 1232
- Symbol
 - defined 1013
 - preprocessor 1011
 - undefined 1016
- Synchronization 637
 - bar constraints 633
 - bars 637
 - between client and server 752
 - code changes 602
 - fork bars 638
 - modifying bars 640
 - StateMate and Rhapsody 1385
- Synchronization bar 639
- Synchronization option 717
- Synchronous events 683
- SysML 2
 - action types 1254
 - activity diagrams 1254
 - activity modeling 1254
 - block definition diagrams 1226
 - diagrams 1227
 - dimension 1264
 - flow of control 1254
 - flows 1227
 - model element stereotypes 1227
 - model libraries 1227
 - parametric diagram 1270
 - ports 1227
 - profile 1243
 - project type 1225
 - report template 1201
 - requirements diagrams 1243
 - requirements in 1227
 - SA imported elements 1291
 - specialized requirement types 1246
 - Teamcenter 1295, 1297
 - units 1264
 - valueType 1264
- SysML profile 226, 1225
 - allocation 1227
 - packages 1227
 - requirements tabular view 1247
 - starting point 1226
- System
 - border 678
 - border in ASDs 1120
 - boundary box 517
 - thread 1098
 - viewing snapshot 535
- System Architect 1291
 - creating SysML diagram from 1293
 - encyclopedia 1292
 - importing DoDAF elements 1292
 - mapping elements to Rhapsody 1291
 - post processing 1294
- Systems engineering 1223
 - accept event action 1257
 - action pins 623, 1257
 - activity diagrams 1254
 - activity diagrams drawing icons 1256
 - activity parameters 623, 1257
 - activity view 1232
 - actors 1249
 - architectural design wizard 1235
 - architecture 1262
 - block definition diagram 1262
 - create ports and interfaces 1233
 - creating a project 1223
 - default flow 1258
 - defining dependencies 1245
 - design structure 1261
 - diagrams 1223
 - display options 1274
 - equations 1274
 - for synchronization 1259
 - Harmony process 1228
 - Harmony toolkit 1231
 - link wizard 1237
 - modeling behavior 1284
 - organize activities 1259
 - requirements 1240
 - simulation 1284
 - statecharts 1284
 - stereotypes 1253
 - subactivity 1258
 - SysML profile 1225
 - system boundary 1249
 - tracing requirements 1252
 - transitions 1258
 - use case diagrams 1240, 1249
 - validation 1284
 - version 1223
 - WSDL file generation 301

Systems view (DoDAF) 1303
Systems viewpoint (MODAF) 1338

T

Table views 241, 245
 attributes listed in 243
 binding view and layout 256
 customize for MODAF 1352
 export data 261
 layouts 242
 layouts for SysML 1227
 manage data 261
 requirements listed in 244

Tables
 browser icon 318

Tabs
 Contract 121
 end 553
 hiding on Features dialog box 49
 Implementation 95

Tags 373, 396, 411
 browser icon 317
 creating 411
 deleting 416
 DoDAF 1319
 features 411
 global 413
 graphical representation 348
 individual element 414
 stereotype 411
 tab 103

Target
 building 927
 field 745
 of hyperlink 53

Target label field 53

Target name field 53

Targets 927
 for hyperlinks 55
 icons for hyperlinks 54
 Send Action State 763

Teamcenter 1295
 creating Rhapsody model 1298
 importing Rhapsody model 1297
 integration 1295
 modifying elements shared with Rhapsody 1298
 prerequisites for working with Rhapsody 1297
 SysML 1295, 1297
 UML 1295, 1297
 viewing corresponding Rhapsody elements 1299

Technical support 1459, 1461
 current Rhapsody customers 1459
 new customers 1462

Technical view (DoDAF) 1304

Technical viewpoint (MODAF) 1338

Template check box 87

Template Class 153

Template instantiation 156

Template Instantiation Argument Dialog box 160

Template specialization 153, 156

Templates 153, 1196
 classes 77
 code generation 161
 customizing report 1199
 DiffMerge tool 153
 for code-based documentation systems 955
 instantiation 77
 limitations 161
 ModafReport.tpl 1362
 parameters 161
 reverse engineering (C++ only) 1050
 roundtripping (C++ only) 1070
 system model 1201

Termination
 connector 771
 state 741

Termination states 630, 664
 activity diagram 629
 creating 630, 664
 flow chart 664

Terminology
 in customized profile 406

Test bench 1233

TestConductor 285

Testing
 application on remote target 1087
 phase 73

Testing profile for OMG 226

TestingProfile 1233

Text
 adding to file 866
 editing 472
 format 39
 selecting editor 867

Text Box control 811

Text Box tool 800

Text editor 415

Third-party interfaces 9

this_ variable 652

Threads 1146
 active 1099
 animation 1097
 focus in animation 1097
 focus in tracer 1157
 mainThread 1098
 multiple 337, 1098
 naming in animation 1098
 naming in tracing 1146
 priority 1099
 resuming 1157
 suspending 1161
 system 1098

TIFF 383

- Tile, maintaining window content 14
- Time
 - behavior modeling 794
 - interval 698
 - message arrival 717
 - Model field 880
 - out trigger 753
 - outs 1, 697
 - real setting 880
 - real-time environment 1
 - simulated setting 880
 - stamp 238, 1128
 - stamp command 1161
- Timeout 676
 - cancelled 676, 697
 - creating 697
 - interval 676
 - trigger 753
- timestamp command 1161
- tm() keyword 753
- To-many relations 967
- Tool
 - Break 1094
 - Call operations 695, 1107
 - Command Prompt 1094
- Toolbars
 - Animation 35, 1092
 - Arrange 469
 - Code 30
 - Common Drawing 37
 - Favorites 32
 - Format 39
 - Free Shapes 41
 - Layout 40
 - modeling 37
 - Standard 29
 - structure diagram 840
 - VBA 35
 - Windows 31
 - zoom 38
- Tools menu 492
- To-one relations 967
- Tornado 307, 308, 309
- Trace 1128, 1155
 - calling operations 695, 1107
 - command 1161
 - field 882
 - requirements 1245
 - selective 882
- trace stereotype 1247
- Tracer 1143
 - break command 1147
 - CALL command 1150
 - commands 1127, 1147
 - display commands 1152
 - ending a session 1144
 - GEN commands 1152
 - go command 1153
 - help command 1154
 - input 1154
 - input command 1154
 - messages 1166
 - output command 1156
 - output destination 1156
 - quit command 1156
 - show command 1158
 - starting a session 1144
 - stepping through the application 1153
 - timestamp command 1161
 - trace command 1161
 - using 1143
 - watch command 1165
- Trade analysis option 1232, 1234
- Transition states 1118
- Transitions 744, 1258
 - activity diagrams 633
 - add actions on 1275
 - completion 634
 - compound 746
 - conflicts 776
 - context 744
 - context message parameters 792
 - creating activity diagram 633
 - default 765
 - default in statecharts 765
 - enabled 776
 - execution 778
 - fork 747
 - in statecharts 744
 - join 622, 736, 748, 754
 - join for activity diagrams 635
 - label 623, 736
 - label for activity diagrams 634
 - label for statechart 749
 - labels 635
 - labels in statecharts 749
 - loop for activity diagram 634
 - null activity diagrams 649
 - null in statecharts 754
 - priorities 776
 - separate 623, 736
 - statechart 744
 - to self 634
 - types 746
- Translative code generation 922
- Transparency 408
- Trigger 749
 - field 745
 - null 754
 - selecting 748
 - timeout 753
- TriggerArgument property 143
- Triggered operations 91, 690
 - applying 752

- replies 752
- roundtripping 1067
- statechart 752
- Troubleshooting
 - clean configuration 932
 - code generation 932
 - connecting to models from the Web 1175
 - controlled files 375
 - DoDAF 1326, 1330
 - MODAF 1367, 1368, 1369
 - MODAF and ReporterPLUS 1364
 - ReporterPLUS and MODAF 1364
 - Simulink 1428
 - Statemate with Rhapsody 1386
- Type declarations search option 363
- Type field 862, 1225, 1230
 - block 844
- Typedef 140, 1038
- Types 338
 - block definition diagram value 1264
 - composite 135
 - creating 135
 - creating enumerated 137
 - creating languages 138
 - creating structure 139
 - creating typedef 140
 - creating union 141
 - editing the order of 147
 - field 145
 - language 139
 - language-independent 144
 - logical file 865
 - message 689
 - modifying 83
 - nested 78
 - predefined 144
 - primitive operations 87
 - receptions 90
 - roundtripping user-defined 1067
 - transitions 746
 - user-defined 1153
- U**
- UML 2, 1419
 - diagrams 65, 223
 - dynamic behavior view 67
 - export format to XMI 1420
 - export versions 1420
 - import format from XMI 1422
 - MODAF 1334
 - signal 88
 - structural view 66
 - Teamcenter 1295, 1297
 - views 66
- Undefined symbol 1016
- Undo 234
 - limitations 235
 - operations that cannot be undone 235
 - zoom 475
- Undo/Redo internal editor 427
- Undock Features dialog box 45
- Unified Modeling Language
 - diagrams 65
 - dynamic behavior views 67
 - structural views 66
 - views 66
- Union 141
 - creating 141
 - reverse engineering 1051
- UNISYS format for diagrams 1420, 1421
- Units 212, 274
 - access privileges 276
 - adding to model 182
 - adding to workspace 282
 - browser icon 316
 - characteristics 275
 - ClassIsSavedUnit property 280
 - comparing 286
 - creating 276
 - elements saved as 264
 - icons 264, 277
 - importing 182
 - loading 282
 - loading and unloading 277
 - modifying 277
 - names 275
 - projects 274
 - referencing 264, 275
 - saving in separate directories 279
 - saving individual 277
 - separating project into 276
 - stub 283
 - SysML 1264
 - Unit View 213
 - unloaded 283
 - unresolved 283
 - view 321
- Unloaded unit 283
- Unresolved unit 283
- Update-on-Break mode 1111
- Upload file 1182
- Upload to a File Server page 1184
- Usage of elements 358
- UsageType property 609
- Use case diagrams 65, 66, 515, 1249
 - associations 524
 - automatically populating 435
 - boundary box 1249
 - creating 433, 516
 - defining SysML requirements 1252
 - dependencies 525, 1253
 - drawing icons 516
 - editor 6

- files 284
 - flow of information 1253
 - Gateway 1240
 - generalizations 524
 - icon in browser 319
 - IntelliVisor information 485
 - packages 523
 - system boundary box 517
 - tracing requirements 1252
 - Use cases 339, 517, 1249
 - actors with 1252
 - attributes 519
 - browser icon 317
 - creating 517
 - creating a statechart 520
 - define features 1250
 - drag and drop 345
 - in requirements diagrams 1244
 - operation 519
 - requirements 1252
 - view in browser 320
 - Use Default check box 880
 - Use existing type field 82, 87
 - UseAsExternal property 1030
 - UseDescriptionTemplates property 1208
 - UseIncrementalSave property 237
 - User Points option 443
 - UseRapidPorts property 310
 - User-defined checks 899, 904
 - User-defined keywords (reverse engineering) 1017
 - User-defined type 1153
 - creating 82
 - generate event of 1095
 - Typedefs 1038
 - UseRemoteHost property 1085
 - Utilities 8, 1309
- V**
- Validation 1284
 - Value field 535
 - Value, specifying for instances 535
 - VariableInitializationFile property 337
 - Variable-length argument list 946
 - VariableLengthArgumentList 946
 - Variables 337
 - adding to a diagram 595
 - create 337
 - global 337
 - ordering 337
 - roundtripping 1068
 - rpyRetVal 652
 - tab 589
 - this_ 652
 - VBA
 - adding macros 497
 - keyboard shortcuts 1452
 - project file 285
 - toolbar 35
 - verify stereotype 1247
 - Video capture 1461
 - View
 - components 320
 - designing 1179
 - diagrams in browser 320
 - entire model 320
 - generated code 932
 - internal editor options 418
 - model from the Web 1181
 - overridden properties with browser filter 321
 - removing elements from 471
 - requirements 321
 - scale 475
 - split 425
 - Unit View 213
 - use case 1405
 - use cases in browser 320
 - Viewing preference 282
 - Viewpoint
 - tags 1227
 - Viewport 14
 - Views
 - hide empty cells 253
 - include descendants 245, 247, 251
 - include ports 243, 254
 - matrix 251
 - show relations 243, 254
 - specification 476
 - structured 476
 - SysML requirements 1247
 - table 245
 - Visibility
 - attributes 82
 - destructors 97
 - primitive operations 87
 - receptions 90, 94
 - Visualization Only (Import as External) 999, 1001
 - Visualize eternal elements 1010
- W**
- Warning 899
 - Watch command 1165
 - Watch mode 1092, 1111
 - Web browser, navigating to a model 1174
 - Web GUI 1181, 1182
 - Web pages
 - adding Rhapsody functionality to 1188
 - automatic refresh rate 1193
 - binding embeddable objects 1189
 - changing home page in webconfig.c file 1192
 - Define View 1178
 - device name 1192
 - navigation 1176

- Personalized Navigation page 1179
 - Web server 1169
 - changing port number 1193
 - customizing 1192
 - generating Web site 1174
 - port 1173
 - Web services 1183
 - libraries 1183
 - Statistics page 1186
 - Upload to a File Server page 1184
 - Web site 1463
 - webconfig.c file 1184, 1192, 1193, 1194
 - Web-enable 1286
 - a model 1286
 - configuration 1286
 - interface 1289
 - property 1288
 - sending events to a model 1290
 - setting stereotype 1288
 - Web-enabled devices 1169, 1176
 - adding files to model 1182
 - Advanced Settings 1172
 - connecting to a model 1174
 - controlling 1181
 - customizing the GUI 1182
 - Define View page 1178
 - limitations 1171
 - name/value pairs 1181
 - Personalized Navigation page 1179
 - properties for 1173
 - setting elements as 1170
 - troubleshooting 1175
 - viewing 1181
 - Webify 880, 1170, 1286
 - setting parameters 1172
 - Toolkit 1169, 1183
 - WebManaged property 1173
 - Web-services Definition Language (WSDL) 226, 300
 - creating 303
 - exporting 303
 - Welcome Screen 27
 - Window
 - characteristics 10
 - docking 11
 - drawing area 12
 - keyboard shortcuts 1452
 - maintaining the content 14
 - output 17
 - preference 283
 - properties for internal editor 417
 - repositioning 11
 - undocking 11
 - viewport 14
 - Windows 163, 371
 - browsers 1201
 - Internet Explorer 1421
 - navigation buttons 31
 - toolbar 31
 - viewing reports 1201
 - Wizards
 - Architectural Design 1235
 - Link 1237
 - Workflow 598, 1010
 - Workflow Integration (Rhapsody and Eclipse) 163
 - Workspace 282
 - adding units to 282
 - creating 282
 - file 285
 - opening 283
 - saving 282
 - window preferences 283
- X**
- XMI 1419
 - examining exported file 1421
 - export action pins 649
 - export activity parameters 649
 - export as version 2.1 1420
 - importing 1422
 - in development 1419
 - SysML support for version 2.1 1226
 - XML Metadata Interchange (XMI) 1419
- Z**
- Zoom 38, 474
 - scaling percentage 475
 - undoing 475

