Copyrights

Copyright Notice

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

Copyright © 2008 by IBM Corporation.

IBM Patents and Licensing

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send written license inquiries to the following:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software|
IBM Corporation
1 Rogers Street
Cambridge, Massachusetts 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Disclaimer of Warranty

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions. Therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Confidential Information

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Additional legal notices are described in the legal_information.html file that is included in your software installation.

Sample Code Copyright

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are

written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

IBM Trademarks

For a list of IBM trademarks, visit this Web site www.ibm.com/legal/copytrade.html. This contains a current listing of United States trademarks owned by IBM. Please note that laws concerning use and marking of trademarks or product names vary by country. Always consult a local attorney for additional guidance. Those trademarks followed by ® are registered trademarks of IBM in the United States; all others are trademarks or common law marks of IBM in the United States.

Not all common law marks used by IBM are listed on this page. Because of the large number of products marketed by IBM, IBM's practice is to list only the most important of its common law marks. Failure of a mark to appear on this page does not mean that IBM does not use the mark nor does it mean that the product is not actively marketed or is not significant within its relevant market.

Third-party Trademarks

Adobe, the Adobe logo, Acrobat, the Acrobat logo, FrameMaker, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.

AIX and Informix are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

HP and HP-UX are registered trademarks of Hewlett-Packard Corporation.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Macrovision and FLEXnet are registered trademarks or trademarks of Macrovision Corporation.

Microsoft, Windows, Windows 2003, Windows XP, Windows Vista and/or other Microsoft products referenced herein are either trademarks or registered trademarks of Microsoft Corporation.

Netscape and Netscape Enterprise Server are registered trademarks of Netscape Communications Corporation in the United States and other countries.

Sun, Sun Microsystems, Solaris, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Pentium is a trademark of Intel Corporation.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product or service names may be trademarks or service marks of others.

Table of Contents

Copyrights

Copyright Notice 1

Introduction

1

Introduction to DOORS Analyst 4.2 **1**

DOORS Analyst commands in DOORS 3
General functionality 3
Restrictions 8
Analyst menu 9
DOORS Analyst diagram view 12
Overview of DOORS Analyst User Interface 12
Desktop 14
Workspace window 14
Views 15
Shortcuts window 17
Output window 17
Working with windows 18
Menu bar and toolbar 22
Status bar 25
Options 25
Model and Diagrams 26

Models 26
Diagrams 27
How to Use Help 31
Navigate in the help file 31
Search syntax in help 34

UML Modeling

2

Working with Models 39

Models and Model Elements 40
Model element and Presentation element 41
Model element 42
Text Highlighting 43
Properties 43
Model checking 43
Models and Diagrams 45
Diagrams 45
Presentation element 45
Properties Editor 47
Opening the Properties Editor 47
The Properties Editor View 47
Different Kinds of Properties 49
Properties Editor Options 50
General Shortcut Menu 52
Control Shortcut Menu 54
Color Codes 55
Customizing the Properties Editor 58
Designing a Stereotype 58

Designing a Metaclass 61
TTDExtensionManagement Profile 63
instancePresentation 63
extensionPresentation 65
filterStereotypes 67
Control model 68
Create Presentation 76
Model Navigator 78
Model navigator tabs 78
Tab categories 79
Navigation 80
Presentation tabs 80
Links 81
Entity tabs 81
Columns 82
Generate Diagram 84
Diagram Generation Parameters 85
Regenerate Diagram 85
Using Diagram Generators in Existing Diagrams 86
Advanced Diagram Generators 86
Customization 88
Queries 89
Query expression 90
Collection Operators 90
The Query Dialog 93
Built-in Queries and Predicates 95
User-defined Queries and Predicates 95
Executing a Query Expression from the APIs 95
Drag and Drop 97
Within the Model View 97
From Model View to a Diagram 98
Within and between Diagrams 99

3

Working with Diagrams 109

Common Diagram Operations 110
Create diagrams 111
Open, save and print diagrams 111
Move diagrams 112
Resize diagrams 112
Find 113
Text parsing 113
Diagram auto layout 114
Organizing the view 115
DOORS Analyst commands 116
Common Symbol Operations 117
Symbol information 118
Add symbols 119
Show elements 120
Select symbols 121
Move symbols 122
Resize symbols 122
Connect symbols 123
Symbol flow editing 124
Edit text fields in symbols 125
Diagram element properties 125
Handling comments 126
Copy, cut, delete or paste symbols 127
Icon 128
Image Selector 129
Undo 130
Model references 130
Nested symbols 132
Symbols with compartments 132
Compartment text fields 134
Common Line Operations 134
Line styles 135

Draw lines 136
Editing vertices 137
Move lines 137
Delete lines 137
Re-direct and bi-direct lines 137

4

UML Language Guide 139

Introduction 140
UML version 140
Diagrams 140
Models and diagrams 141
List of language constructs 144
Scope, model elements, and diagrams 145
General Language Constructs 147
Names 147
Alternative syntax 150
Common element properties 151
Predefined names 155
Use Case Modeling 156
Use case diagram 156
Use cases 157
Actors 159
Subjects 160
Relationships 160
Scenario Modeling 162
Sequence diagram 162
Interaction 164
Interaction reference 165
Lifeline 166
Message 170
Timer event 175
Time specification line 176
State 178

Action . . . 179
Create . . . 179
Destroy . . . 181
Inline Frame . . . 181
Co-region . . . 184
Continuation . . . 185
Method call . . . 186
Appearance and filtered delete . . . 188
Interaction overview diagram . . . 189
Package Modeling . . . 191
Package diagram . . . 191
Package . . . 192
Relationships . . . 193
<<noScope>> Packages . . . 196
<<openNamespace>> Packages . . . 197
Class Modeling . . . 197
Class diagram . . . 199
Class . . . 200
Collaboration . . . 206
Attribute . . . 206
Operation . . . 211
Active class . . . 212
Port . . . 214
Interface . . . 217
Realized interface . . . 219
Required interface . . . 220
Signal . . . 221
Signallist . . . 223
Timer . . . 223
Datatype . . . 224
Choice . . . 227
Syntype . . . 228
State machine . . . 229
Stereotype . . . 229

Relationships . . . 229
Object Modeling . . . 229
Object Diagram . . . 230
Named Instance . . . 232
Slot . . . 234
Architecture Modeling . . . 235
Composite structure diagram . . . 235
Part . . . 236
Connector . . . 239
Behavior port . . . 241
Relationships . . . 242
Component Modeling . . . 243
Component diagram . . . 243
Component . . . 244
Relationships . . . 245
Activity Modeling . . . 245
Activity Diagram . . . 246
Activity . . . 248
Activity implementation . . . 249
Initial Node . . . 250
Action Node . . . 250
Object Node . . . 253
Decision . . . 254
Merge . . . 255
Fork . . . 256
Join . . . 257
Connector . . . 258
Accept Event . . . 258
Send Signal . . . 259
Accept Time Event . . . 259
Activity Final . . . 260
Flow Final . . . 260
Activity Partition . . . 261
Pin . . . 263

Relationships 264
Behavior Modeling 265
State machine diagram 265
State machine 267
State 268
Transition 271
History nextstate 272
Signal Receipt (Input) 274
Start 276
Action 277
Signal sending action (output) 278
Decision 280
Guard 282
Timer set action 284
Timer reset action 284
Action (task) 285
Assignment 285
Compound statement 286
New 286
Save 287
Stop 288
Return 288
Junction 289
Flow 290
Simple transition 290
Expressions 291
Composite state 297
State machine inheritance 299
Operation body 299
State machine implementation 300
Internals 300
Text extension symbol 301
Deployment Modeling 301
Deployment diagram 301
Artifact 303

Node 303
Execution environment 304
Deployment specification 304
Relationships 305
Relationships in UML 306
Dependency 307
Generalization 307
Realization 308
Association 308
Aggregation 311
Composition 311
Containment 312
Extension 313
Association 313
Common Symbols 313
Frame 313
Text symbol 313
Comment 314
Constraint 314
Stereotype instance 315
Annotation line 315
Extensibility 316
Metamodel 316
Metaclass 317
Stereotype 317
Profile 318
Extension 318
Predefined Data 318
Predefined 319
Metamodel Classes 320
Classifier 320
Signature 320
Implementation 322
Method 322

Signature and implementation 323
Profile for Schedulability, Performance, and Time 324
RTresourceModeling 324
RTtimeModeling 324
RTconcurrencyModeling 327
SAprofile 327
PAprofile 330
RSAprofile 332

5

Error and Warning Messages 335

General Application Errors and Warnings 336
DOORS Analyst minidumps (Windows) 336
Errors and Warnings 337
TSX: Syntax Analysis 338
TSX0026: Port should not contain two in or two out parts 338
TSX0047: Tagged values are not allowed here 338
TSC: Semantic Check 339
About semantic checks 339
TSC0123: A cyclic dependency was found in definition of the %n. (via <string>) 339
TSC0134: Incomplete transition. A transition must end with stop, nextstate or join action 339
TSC0092: A corresponding 'virtual' or 'redefined' operation was not found in the parent signatures (or parent signatures does not exist). 339
TSC0196: A finalized operation cannot be redefined. 341
TSC0236: Operation '<name>' cannot be specified as 'Realized' on a port. 341
TSC0237: Operation '<name>' cannot be specified as 'Required' on a port. 342
TSC2300: Expression 'any (type)' cannot be of interface or state machine type 342
TSC2302: An association from a datatype may not have a navigable remote association end 343
TSC2303: At most one association end may be aggregate or composite . . 343
TSC2304: An attribute that is not a part may not have initial count 343
TSC2305: A part cannot have a default value 344

TSC2306: A composite attribute or association end may not be typed by a datatype . . . 344
TSC2307: A composite attribute may not have a type, which owns this attribute (directly or indirectly) . . . 344
TSC2308: The 'via' of a call expression should reference either a port or a connector . . . 345
TSC0269: Generalization between 'Interface I' and 'Class Y' is not allowed . . 345
TSC2325: Cyclic inheritance . . . 345
TSC4001: When generating C code, return values must be handled in left hand side of assignment expression . . . 346
TNR: Name Resolution . . . 347
TNR0023: Failed to locate element referred by: <name> . . . 347

UML Import and Export

6

UML 1.x Import 351
Operation Principles . . . 352
XMI import . . . 352
Import an XMI file . . . 354
Supported XMI and UML . . . 355
Language and version support . . . 355
Supported diagram types . . . 357
Import from UML 1.x tools . . . 358
Restrictions . . . 359
Type and variable definitions . . . 359
Incomplete model . . . 359
Unsupported classes . . . 360
Unsupported attributes . . . 361

Unsupported composition 363
Export restrictions 364
Error Messages 368

7

UML 1.x Export 371

XMI Export 372
Operation principles 372
Supported XMI and tool versions 372
Supported UML entities 372
Model hierarchy 380
Restrictions for XMI export to Rational Rose 384
Error and warning messages 386

Common Reference

8

Printing 391

Printing Diagrams 392
Print settings 392
Select diagrams to be printed 393
Preview of diagrams 393
Print a single diagram 393
Print multiple diagrams 394

9

Internationalization Support 395

Supported environments 395

Font settings 396
Modeling with CJK characters 397
Handling textual files 398
Restrictions 398

10

Useful Shortcut Keys 399

Workspace Operations 399
Project Operations 400
File Operations 400
Navigate in Files 400
Highlight Text 401
Edit Text 402
Editor Shortcuts 403
Window Navigation 405
Properties editor 406
Show/Hide Windows and Dialogs 406
Zoom/Pan 407

11

Dialog Help 409

The New Wizard 410
Files tab 410
Projects tab 410
UML Projects - page 2 410
UML Projects - page 3 410
Workspaces 411
Customize 411
Commands tab 411
Toolbars tab 412
Create New Toolbar 413
Windows layouts 413
Tools tab 413

Add-ins tab 415
Options 415
General 415
Save 417
Workspace 417
Format 418
Font settings 418
Links 419
Editor Shortcut 420
Show Elements 420
Reconfigure ModelView 420
Other 421
Select Stereotypes 421

12

Additional Resources 423

Links 424
Contacting IBM Rational Software Support 424
UML documents 425
Other links 425

Introduction

DOORS Analyst is a visual modeling environment available inside the requirements management tool DOORS. DOORS Analyst enables users to augment and visualize requirements using diagrams, symbols and pictures based on the standardized, visual modeling language UML.

To fully take advantage of DOORS Analyst and be able to start working quickly, it may prove useful to start with one of the DOORS Analyst is a visual modeling environment available inside the requirements management tool DOORS. DOORS Analyst enables users to augment and visualize requirements using diagrams, symbols and pictures based on the standardized, visual modeling language UML.

includes many capabilities for analysis and development of service-oriented architectures, but of specific interest for this application area are the following chapters:

Chapter 65, Web Services Support, that describes the web service modeling support in DOORS Analyst is a visual modeling environment available inside the requirements management tool DOORS. DOORS Analyst enables users to augment and visualize requirements using diagrams, symbols and pictures based on the standardized, visual modeling language UML.

- Chapter 69, Modeling XML Schemas, that describes how to model the XML data used by the web services,

Chapter 63, Using Tau with System Architect, that describes how to use DOORS Analyst is a visual modeling environment available inside the requirements management tool DOORS. DOORS Analyst enables users to augment and visualize requirements using diagrams, symbols and pictures based on the standardized, visual modeling language UML.

- Chapter 67, WSDL/XSD Importer Reference, that describes how to import existing service descriptions in WSDL or XSD and how to generate WSDL/XSD from UML models.

In addition, the following sections can provide useful information:

- Chapter 10, Useful Shortcut Keys will provide a listing of possible shortcuts, this chapter can provide you with information on how to work faster and more efficient once you are familiar with what DOORS Analyst can achieve.

1

Introduction to DOORS Analyst 4.2

UML

DOORS Analyst contains a set of model-driven tools based on UML 2.1 which are backwards compatible with UML 1.x. There is support for the following diagram types:

- Use case diagram
- Sequence diagram
- State machine diagram
- Activity diagram
- Interaction overview diagram
- Class diagram
- Package diagram
- Component diagram
- Deployment diagram
- Composite structure diagram (formerly called Architecture diagram)

To be able to start working quickly with UML, the topics listed below may prove useful to start with:

- Working with Models
  Describes the basics behind model-based development. It provides you with instructions and introductory information.
- UML Language Guide
  This section is a guide to the UML language.
- Java Tutorial
  A basic tutorial that allows you to work with the supported diagrams and to learn how to verify your model.
- UML Quick reference guide
  Examples on common constructs in graphical and textual UML.

DOORS Analyst commands in DOORS

DOORS Analyst is a set of UML tools that allows handling of UML models from a DOORS requirement database.

General functionality

In all formal modules there is an Analyst menu with a set of commands that allows you to work with UML models. **Enable Analyst** should be performed on a formal module to initiate the other commands. Elements in a formal module will also have a set of DOORS Analyst specific commands added to the shortcut menu. This allows you to apply these commands by right-clicking on the object and point to the command of your choice.

UML Kind

When a module is enabled for DOORS Analyst, a column "Analysis Type" (in previous versions: "Object Type") showing the UML kind will be inserted. A UML icon will also appear next to the name column.

If a UML diagram symbol is synchronized, "Analysis Type" will show "Other".

If a non-synchronized object is moved to a location where the value of "Analysis Type" is invalid, a small red exclamation mark is displayed in the top left corner of the UML icon. This marker indicates that the object is out of context.

Displaying DOORS attribute values in diagrams

The attribute values of an object can be shown in diagrams using a comment symbol attached to the symbol representing the object. Attributes are divided in two categories:

- Object Text
- all other attributes, including user defined ones

For instructions on how to display them in diagrams, see:

- Object Text
- Attributes

Note

*Diagrams can not have comments in DOORS Analyst. It's therefore not possible to display Object Text or any other attribute for a diagram object. Comment symbols placed on the diagram canvas not attached to any symbol is **not** associated with the diagram, but it's owner.*

Object Text

The value of the Object Text attribute of an object is by default stored in the model as well as in DOORS. It can be displayed and edited in DOORS Analyst.

To display Object Text in a diagram:

- Select the symbol representing the object whose text you want to display
- Right-click the symbol and select **Show/Hide->Show Comments**

To remove a comment symbol just select the symbol and click **Delete**. The comment will not be deleted from the model, just from the diagram. It can be displayed again repeating the procedure described above.

To edit Object Text in DOORS Analyst:

- Make sure the Object Text is displayed in a comment symbol as described above
- Edit the text, keeping the heading `Object Text` unchanged

The contents of the **Object Text** attribute is by default propagated between DOORS and DOORS Analyst during synchronization. It is stored as a comment in the model, with the heading `Object Text`. The synchronization of the object text is controlled by the UML Comment Symbol attribute and can be turned on and off at will.

To add Object Text to an object in DOORS Analyst when there is none:

- Make sure there's a symbol representing the correct object
- Create a comment symbol and attach it to the other symbol
- Enter `Object Text` as the first line in the comment symbol
- Enter the desired object text on the following lines

During synchronization, the text will be inserted as object text in the corresponding DOORS object.

UML Comment Symbol attribute

By default, the value of the Object Text attribute is propagated between DOORS and DOORS Analyst and stored as a comment in the model.

The value of the object level attribute **UML Comment Symbol** controls whether the object text shall be propagated or not.

If UML Comment Symbol is `True` (the default) the object text is propagated between the tools. If the value is `False`, it isn't. The value can be changed at any time.

Deleting a UML comment representing Object Text from DOORS Analyst will **not** delete the originating DOORS Object Text. Instead the tool will set the attribute to `False` indicating that it will not be propagated anymore.

Attributes

The values of the DOORS attributes of an object can be displayed in a comment symbol in a diagram in DOORS Analyst.

To show attribute values in a diagram:

- In DOORS, select the object whose values you would like to display in a diagram
- Execute the Select Attributes to Show in Analyst command from the Analyst menu
- Select the attributes you would like to see
- Right-click the object and select **Edit in Analyst**
- Select the symbol representing the object
- Right-click the symbol and select **Show/Hide->Show Comments**

A comment symbol is created and attached to the symbol. The comment has a heading named `Attributes`, followed by the attribute names and values in the following syntax:

```
attribute name : attribute value
```

Attributes with empty values are *not* included.

To remove the comment symbol just select the symbol and click **Delete**. The comment will not be deleted from the model, just from the diagram. It can be displayed again repeating the procedure described above.

Diagrams in DOORS Analyst

A formal module stores UML information in its objects. When the DOORS Analyst window opens up, you will be able to access the information through the DOORS Analyst diagram editors.

Storing the model in DOORS

When DOORS Analyst closes its session, this information will be stored as a UML data model (corresponds to a file with `.u2` extension) in the DOORS module. The next time DOORS Analyst is opened, the UML information will be based on this data model together with any changes made in the formal module.

Referencing elements from multiple modules

In an Analyst-enabled module, it is possible to reference elements from other Analyst modules. This is done by doing “Edit in Analyst” on the modules, they will now be opened in the same diagram editor. Then, enable the Model View, and drag-and-drop elements from the Model View to the diagram where the elements should be referenced.

Shareable edit mode

DOORS Analyst supports DOORS shareable edit mode, enabling users to concurrently work on different parts of the same DOORS Analyst module.

To use this feature proceed as follows:

- Before using this feature, the formal module must be enabled for DOORS Analyst (when the module is in exclusive edit mode).
- Set up editable sections using the standard DOORS commands. Please refer to the DOORS User Manual for more information about how this is done.
- Open the module in shareable edit mode.

  When a module is opened in shareable edit mode, the command **Enable Analyst for Section** will be available.

- To create an Analyst section, first create a DOORS object with an arbitrary name. Select this object and choose **Enable Analyst for Section**.

  The "**Analysis Type**" of this object will now be set to "**Model**", indicating that it corresponds to a UML model, and a corresponding UML root package will be created for this object.
- "Insert UML" can now be used to add UML objects and diagrams.
- To open the diagram editor, use "Edit in Analyst".

  Before launching the editor, a check will be made that all objects in this section can be locked. If successful, the objects will be locked and the editor will be opened. Otherwise, an error message will be displayed.

Note

The hierarchy of Analyst models in a formal module must be "flat", which means that there cannot be any Analyst models in the object hierarchy below another Analyst model. This means that care must be taken if Analyst models are moved around in a formal module, so that they do not end up in the same object hierarchy. The tool cannot detect this error situation until "Edit in Analyst" is performed on an Analyst section, i.e. not when the actual error is introduced.

Objects in one Analyst section can reference objects in another Analyst section, see "Referencing elements from multiple modules" for more information.

Analyst shareable sections cannot be used together with the "standard" Analyst-enabled mode.

Links

Links to/from UML elements are shown in DOORS Analyst as red and orange arrows, in the same way as in DOORS. Right-clicking on an arrow in the Model View will display a menu showing an object's link endpoints. By selecting one of these, the DOORS formal module containing this object will be displayed, with the object selected.

Restrictions

Comments

When you edit diagrams in DOORS Analyst which reference elements from other modules/sections and the edit affects elements in sections that are not directly editable this may create presentations that will not be persistent. This typically can happen if you edit a class diagram and add comments to a class in another module, given that this module is not currently editable.

DOORS Analyst attributes

There exists some attributes used for internal purposes, for example UML Kind, UML Location and UML Name. Deletion of these attributes leads to losing DOORS Analyst data. There is furthermore no possibility to recover such a deletion with Undo and unless you have baselined the module the data is permanently lost.

Diagram/Diagram below

The diagram can be opened by double-clicking on the diagram. This does not apply for DOORS 7.1.

Import Partition and Clone

The DOORS functions “Import Partition” and “Clone” are currently not supported by Analyst.

Copying Analyst modules, e.g. through the DOORS “Copy”, “Paste” or “Paste and update references” commands, will lead to conflicts between the original UML elements and the ones in the copied module. This should be avoided, unless the original module and the copied module are not used concurrently.

Multiple DOORS servers

DOORS Analyst does not support DOORS clients concurrently running from different DOORS servers. It is not recommended for example to open (through “Edit in Analyst”) a DOORS Analyst module from one DOORS database, and then open a DOORS Analyst module from another DOORS database.

Analyst menu

Enable Analyst / Disable Analyst

These commands will indicate and control if the DOORS Analyst mode is activated. A module will be able to store UML models after the menu entry **Enable Analyst** is chosen, and a module will lose its ability to store UML models after the menu entry **Disable Analyst** is chosen. It will no longer be possible to open any objects in Analyst after a **Disable Analyst** command.

Enable Analyst for Section

This command is available when working in DOORS Shareable edit mode. To create an Analyst editable section, the DOORS module must first have been enabled for Analyst, in exclusive edit mode.

A model (`.u2`) file will be indicated with a icon in the DOORS module.

Insert UML

This command gives you the possibility to create diagrams and elements in a DOORS Analyst enabled module. From its submenu it is possible to select between creating an UML diagram or an element and whether to create the new entity in the same scope as the current selection or create it below the current selection.

Element/Element below

These commands allow you to create a model element. It is possible to create the following model elements:

- Actor
- Attribute
- Class
- Component
- Node
- Package
- Subject
- System
- Use case

Once created, model elements are visualized by an icon.

Diagram/Diagram below

These commands allow you to create a diagram. The diagram is opened in DOORS Analyst by double-clicking on the diagram. When closing DOORS Analyst the diagram image is saved in the formal module.

All diagram types are displayed as images in the formal module. They are stored as a Windows metafile (`.wmf`).

Note

In a formal module, Diagrams are represented by a DOORS object, and a table (where the table is shown as an image). DOORS Analyst assumes this table to be directly below the Diagram object. However, other objects may also be placed directly below the Diagram object. A general recommendation is to keep the table immediately following its related object.

Edit in Analyst

This command allows you to edit a selected model element in a diagram using the appropriate DOORS Analyst editor. “Edit in Analyst” is also available from the shortcut menu when right-clicking on a model element or diagram image. When an element is present in several diagrams, only one of them will be displayed.

This command also allows for opening and editing diagrams without any element selection in the formal module. In this case, the UML root package that corresponds to the DOORS module is considered 'selected' and used as the base for the command.

For Shareable edit mode it is possible to edit objects that you have applied Enable Analyst for Section for.

If there are references to model elements stored in other DOORS modules, or other shareable sections, these elements will be loaded in read-only mode by the Analyst diagram editor. The mode can be changed to read/write by opening the corresponding DOORS module, and then select “Edit in Analyst” for these elements.

Select Attributes to Show in Analyst

This command opens a dialog from which you can select DOORS Attributes to be shown in a comment symbol when the selected DOORS element is shown in DOORS Analyst.

Update Diagrams

This command updates all diagrams with the changes made to UML elements in the DOORS module.

Convert Module

This command updates a module which was created with a previous version of Analyst to the current version.

Note

Diagrams etc. may need updating as well, this is done when opening a UML object or diagram in Analyst through "Edit in Analyst", and then saving.

Opening DOORS Analyst modules created in an earlier version may require a conversion of the modules, for example this could be due to new attributes being added in the later version. The need for conversion is detected automatically when opening a DOORS Analyst module, by executing a DXL Interaction script that prints a message to user about the need for conversion. Conversion is performed with the command **Convert Module** followed by a save of the modules and diagrams.

If "Shareable Edit Mode" has been used on the DOORS Analyst modules, it is required that the conversion to the new format is performed on all sections in one operation. For this, user must open the module using "Exclusive Edit".

Help

This command opens the DOORS Analyst help file.

About Analyst

This command displays the About dialog for DOORS Analyst, showing tool version and licensing information.

DOORS Analyst diagram view

When DOORS Analyst opens up you will be able to access its information through the diagrams from the formal module.

Any changes done will be synchronized back to the formal module when save is performed from DOORS Analyst.

Important!

When you move between DOORS and DOORS Analyst you should always use Edit in Analyst or double-click on a diagram. When you move from DOORS Analyst to DOORS you should always close the Analyst window or use the shortcut command “Edit in DOORS”. This will ensure that your elements are properly synchronized. It is also possible to perform an explicit save in DOORS Analyst before changing to the DOORS window.

Overview of DOORS Analyst User Interface

Basic layout

The DOORS Analyst user interface contains two different areas: the **Desktop** and the tool bars. In addition there is a status bar in the lower part of the frame and a menu and tool bar area at the top of the interface frame (the menus will be disabled in Basic layout).

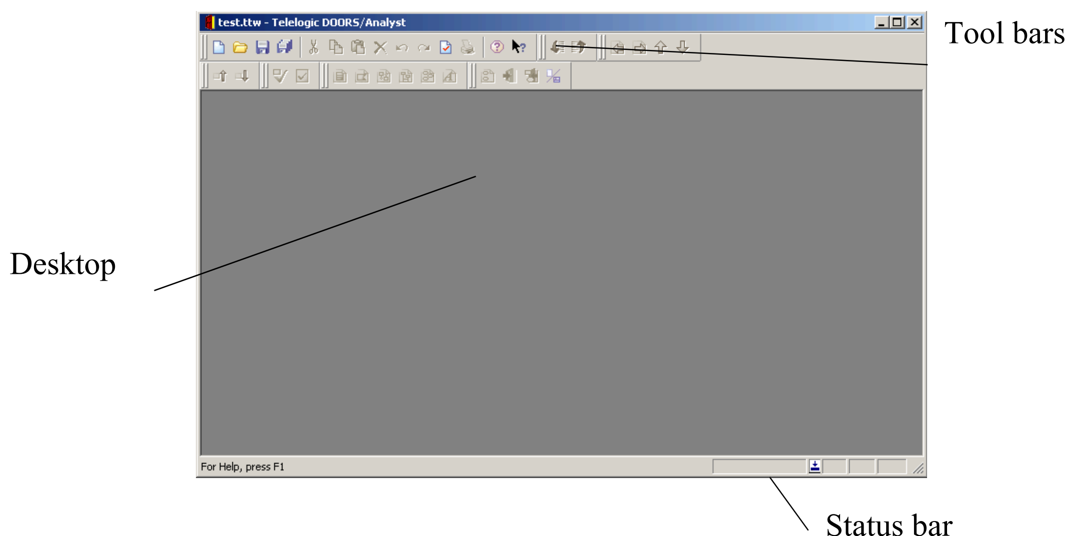


Figure 1: The DOORS Analyst basic layout.

Advanced layout

The complete DOORS Analyst user interface contains several different areas that can be switched on and off at the will of the user: the **Desktop**, the **Workspace** window and the Output window. In addition there is a status bar in the lower part of the frame and a menu and Toolbar area at the top of the interface frame. The windows are all possible to dock according to the user's preferences. It is also possible to drag-and-drop frequently used toolbars to a Shortcuts window.

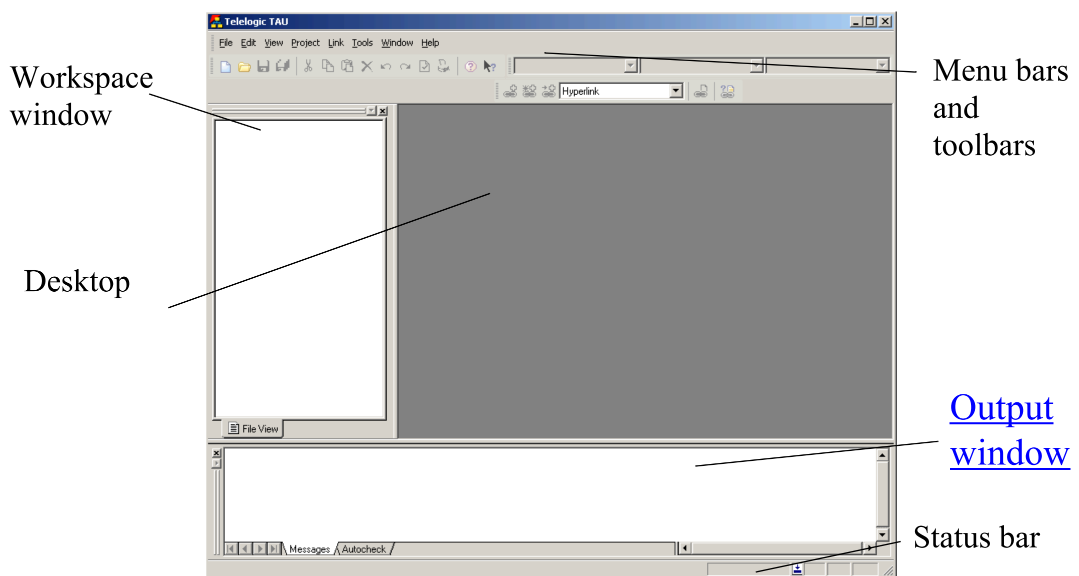


Figure 2: The DOORS Analyst Desktop.

Desktop

The Desktop, or editing area, is the area of your working documents. This is where the actual development takes place. Here you will see diagrams, documents, source files, etc. Once you have opened them for editing or viewing. Which editor or viewer that is displayed depends on the file types that are included in your project.

If you have more than one document opened on the desktop, you can move between with commands on the Window menu, or by pressing CTRL + TAB (forwards) or CTRL + SHIFT + TAB (backwards).

Hint

If you want to have your editor expanded to full screen, select Full Screen in the View menu. To go back, press ESC or ALT + '1'.

See also

"Working with windows" on page 18

Workspace window

The Workspace window is a graphical tool that presents and manages the structure of the workspace information in a number of Views.

The Workspace window shows the information structure with expandable nodes. By collapsing and expanding these nodes, and by using different views, you can focus on different sets of the information in the workspace.

It is possible to move the Workspace window and to make it a floating palette. When not floating, it is docked at this left-most position. When docked, the window can only be resized horizontally. The vertical boundary is determined at the top by the toolbars and at the bottom by the Output window.

Views

In the Workspace window you have access to a number of views. They are each accessible via separate tabs. The views show different aspects of the model.

File View

The File View shows your workspace with all elements that are represented as files.

The File view is not visible when you start DOORS Analyst the first time. To enable the File view you right-click on the frame of the Workspace window (the Model View tab) and select File view.

You select the File View by clicking the **File View** tab in the **Workspace window**. Here you can open, edit and save all files. However, if you delete a file it is only deleted from the File View. The file is still available in the file system of your operating system.

To get a better overview of your files, you can create folders. You can drag and drop files between folders. You can display properties by selecting an item, right-clicking it, and clicking Properties in the shortcut menu.

Model View

The Model View contains all the data that you are working with in an abstract structure. All UML elements are found here. Since the Model View shows all the elements that are not represented as files, this is the view to select when you add elements to the model and create diagrams.

The elements in this view are considered as graphical representations of the model. You may use diagram editors for the design process, but you may as easily fully design a system just working with the nodes in the Model view.

You select the Model View by clicking the **Model View** tab in the **Workspace** window.

The shortcut menu on the project nodes or the Model nodes will contain a sub-menu called **Model View Filters**. This menu will contain check boxes that defines how a set of predefined filters are applied to the Model View.

A Metamodel may include one Metaclass that has its base set to **Resource**. This model element will map to Resource elements in the loaded element at run-time and will thus only be visible in the Model View if the Show Files model view filter is turned on. It is possible to show files and resource elements in the Model View. Use the shortcut sub-menu **Model View Filters** and select **Show Files**.

A metamodel may include metaclasses that have their base set to object model classes that are subclasses of Diagram. These model elements will map to diagrams and will thus only be visible in the Model View if the **Show Diagrams** model view filter is turned on.

For convenience the metaclasses are classified as either structural entities or as detailed entities. This will affect how the contents of the Model View is affected by the **Show Details** filter.

A metamodel may include metaclasses that have their base set to object model classes that are subclasses of Implementation. These model elements will map to implementation-oriented elements and will thus only be visible in the Model View if the **Show Implementations** model view filter is turned on

When the **Sort Definitions** filter is active (indicated by check-mark in the menu item), model elements in a given Model View node are sorted in a lexicographical order. Diagram nodes will not be affected by the sort operation.

The command **Reconfigure Model View** (from the **View** menu) allows you to filter the information in the Model View based on a predefined metamodel. This affects the project which the selected element belongs to.

In some situations it is possible to cause a node in the Model View to "disappear" when you have selected a filter different from the **Standard View**. This happens because some operations in DOORS Analyst (like drag-and-drop) rely on the basic metamodel that is used for storage of UML information, while others (like showing the elements in tree form in the Model View) rely on the currently selected metamodel. When you switch to **Standard View**, this model is the same as the basic metamodel. The **Diagram View** will present the information based on model elements that can own diagrams and push the diagram nodes to directly below their owning elements.

Typically this can happen in the following situations:

- drag-and-drop
- cut and paste
- creating diagrams from the Model Navigator Diagrams tab

To restore a node that have disappeared in this way you have two options:

- Use undo to get the node back at its original place.
- Switch to **Standard View**. In that view, all nodes will be visible. Any nodes that may have been drag-and-dropped to a position where they did not appear in the Model View will now show in their new place.

Shortcuts window

The Shortcuts window may contain toolbars, but it will only do so if you put them there. To put a Toolbar in the Shortcuts window, right-click the toolbar and click **To Shortcut** from the shortcut menu. You can reverse the process by right-clicking on the Shortcuts shown and clicking **As Toolbar** from the shortcut menu that appears.

The **View** menu **Shortcuts** command controls if the Shortcut window is shown.

Note

Not all toolbars will be possible to submit to the shortcuts window.

Output window

The Output window consists of a number of different tabs that records and displays information for the corresponding tool. This information is typically error messages, warnings, result of actions, logging of events, etc. Each tab represents a different tool.

For some of the tabs, it is possible to navigate from the located element (in the subject column) to a presentation in a diagram.

The **View** menu **Output** command controls if the window is shown.

General tabs

Messages

The Messages tab shows general information regarding the project loading and other executed actions.

No navigation is available from this tab to other parts of the tool.

Search result

This tab displays the result of a Find operation.

Presentations

This tab displays the result of a List presentations operation.

References

This tab displays the result of a List references operation.

Script

The Script tab shows the result of scripts.

UML tool tabs

Check

You can initiate a complete check of the model to detect errors and warnings. This tab displays the result of the check. The errors remain in the list even after they are corrected. The list is changed the next time you invoke the check procedure.

Navigate

This tab contains a tabular model navigation tool, the Model Navigator which is used to navigate through any given model.

Working with windows

The Window menu is only available in the Advanced layout.

Arrange windows

Tile all document windows:

- On the **Window** menu, click **Tile Horizontally** or **Tile Vertically**.

Overlap document windows:

- On the **Window** menu, click **Cascade**.

To change the position of document windows:

Note

The docking state can not be changed for a window with tabbed documents.

1. Right-click the title bar of a document window.
2. In the menu select:
   - **Docked** to dock the window within the application window. There are also options for where it should be docked.
   - **Floating** to be able to move it outside the application window.
   - **MDI Child** to make the window float within the editing area. There are also options for maximizing, minimizing and restoring the window.

To view the active document in full screen:

- On the **View** menu, click **Full Screen**

  Or

- Press **ALT + 1**

To view the active document in normal size:

To display the active document in normal size again after you have viewed it in full screen, do one of the following:

- Move the cursor to the top of the screen. When the menu bar appears, on the **View** menu, click **Full Screen**.

  Or
- Press **ALT + 1**

Show and hide windows

Show or hide the workspace window:

- On the **View** menu, click **Workspace**

  Or
- Press **ALT + 0**.

Show or hide the output window:

- On the **View** menu, click **Output**

  Or
- Press **ALT + 2**.

Close windows

To close a document window:

- On the **Window** menu, click **Close**.

To close all document windows:

- On the **Window** menu, click **Close All**.

Create a new window

To create a new document window:

- On the **Window** menu, click **New Window**.

Tabbed documents

If the option “Tabbed documents” is enabled in general options page, documents will be opened as tabs in a single window.

A document can be detached from a tabbed window by right-clicking the tab and selecting the Detach menu item. It will then function as a normal MDI child and the docking state can be changed.

Docking windows

There are three different modes for editor windows in the DOORS Analyst framework. These are applied to each diagram window individually by right-clicking the diagram title bar.

Note

The docking state can not be changed for a window with tabbed documents.

Docked

A docked editor window will align into the DOORS Analyst framework like the workspace window or the Output window. It will be possible to move the windows around to arrange a suitable view.

Floating

A floating window is on top of the DOORS Analyst framework. It will turn into a docked window if it is moved towards the framework frame.

MDI child

An MDI child window is positioned into the desktop area. Adjusting can be done manually inside this area or with commands from the **Window** menu.

Auto-hide docked window (Windows)

A window that has a pin symbol in the gripper bar can set to auto-hide mode. If the pin is pressed, the window will be hidden and a label representing the window will be displayed instead. By hovering with the mouse over the label the hidden window will be displayed. Window can be docked again by clicking on the pin symbol again.

Expand/Contract docked window

When two docked windows share the same side of the main window, a window can be expanded to take the whole side in possession by clicking on the arrow symbol on the gripper bar of the window (if available). By doing this the other windows on the side are minimized. The window can be contracted by clicking on the arrow symbol again.

Stored workspace windows

All windows opened during a session will be reopened when the workspace is loaded again. The information will be saved in a .ttx file with the same path and name as the workspace.

See also

"Organizing the view" on page 115

Menu bar and toolbar

Depending on your workspace preferences, and the size of your screen, you can display as many toolbars as you want, or none at all. You can add a button with a command to a toolbar, change the size of the buttons, and move the toolbars to different locations to suit your needs.

Menu Bar

The menu bar contains well-known menus such as File, Edit, Project, and so on. Depending on the task you are performing the number of menus differ.

Most menu commands have a shortcut assigned. A list of Useful Shortcut Keys is found in Common Reference.

You can add commands to the Tools menu allowing you to easy access to non-Telelogic tools. This is done from the Tools tab.

As an example, the following procedure demonstrates how to add the Windows Notepad accessory to the Tools menu.

To add a command to the Tools menu:

1. From the **Tools** menu select Customize, and then click the Tools tab.
2. Click the New (Insert) button.
3. Type the name of the tool, as you want it to appear on the Tools menu, and press ENTER.

   For example, if you want to add a command for the Windows Notepad accessory, you might type Notepad.
4. In the **Command** box, browse or type the path and name of the program, for example, C:\Windows\notepad.exe.

5. In the **Arguments** text box, browse or type any arguments to be passed to the program. Leave this field empty for the Notepad accessory.

Note

You can use the drop-down arrow next to the Arguments text box to display a menu of arguments. Select an argument from the list to insert argument syntax into the Arguments text box.

6. The **Initial Directory** box is used to specify the file directory where the executable file for the command is located. For the Notepad accessory this field is left empty.

When the command appears on the Tools menu, you may click it to run the program.

You can add arguments to be passed to the program by typing them in the Arguments text box, or set the initial directory for your program by typing it in the Initial Directory text box.

If the program you are adding to the Tools menu has a `.pif` file, the startup directory specified by the `.pif` file overrides the directory specified in the Initial Directory text box.

Toolbar

The toolbar allows you to set up a palette of your most common used tools in order to have quick access to them. Once you have made any changes to the toolbar, these changes are saved and retrieved for your next work session.

The standard toolbar corresponds to the operations available in the menu bar. The standard toolbar can be toggled on and off from the View menu (Standard command) or from the toolbar area’s shortcut menu, the other toolbars only from the shortcut menu.

Note

Not all toolbars and commands can be modified. This feature belongs to the DOORS Analyst framework and is not supported for toolbars related to editors.

To add a toolbar button:

1. Make sure that the toolbar you want to change is displayed.
2. From the **Tools** menu select Customize, and then click the **Commands** tab.

3. Add a button by clicking the name of the category in the Categories box, and then dragging the button or item from the Buttons area to the displayed toolbar.

To delete a toolbar button:

1. Make sure that the toolbar you want to change is displayed.
2. From the **Tools** menu select Customize, and then click the **Commands** tab.
3. To delete a button, drag it off the toolbar.

When you delete a default button from a toolbar, the button is still available in the Customize dialog box. However, when you delete a toolbar button with a custom appearance, its appearance is permanently lost, although the command is still available (Customize dialog box, Commands tab).

Hint

To save a toolbar button with a custom appearance for later use, create a toolbar for storing unused buttons, move the button to this storage toolbar, and then hide the storage toolbar.

Show or hide toolbars:

1. From the **Tools** menu select Customize, and then click the **Toolbars** tab.
2. Select and clear toolbars to show or hide in the **Toolbars** list.
3. Click Close.

alternatively

1. Right-click anywhere in the toolbar area in the user interface.
2. Click the toolbar you want to show or hide. The menu closes automatically.

To change the appearance of toolbar buttons:

1. From the **Tools** menu select Customize, and then click the **Toolbars** tab.
2. Select the following options:
   - **Show Tooltips** to enable tooltips to be displayed when the cursor moves over a button or field in the toolbars.
   - **Large Buttons** to display larger sized buttons in the toolbars.
3. Click **Close**.

Status bar

The status bar presents useful information about status of several different types of tasks, for example it lists errors and tooltips. Here will also be presented information about progress and current actions.

For text files the current line number and column position are shown in the right most corner of the Status bar.

Line navigation

Navigation to a specific line in a text file is done by pressing CTRL + SHIFT + G. Enter the wanted line number in the dialog that opens.

Progress bar

There is one progress bar displayed showing the overall progress when opening a workspace to the right of the status bar.

There can also be one displayed for the progress of the separate parts of the loading process displayed in the message field. In this case there will also be a message explaining the current action in progress.

Options

Tool options affect DOORS Analyst, not just the current project or workspace. There are different ways of changing these options:

In the Options dialog, there are different tabs for different options that you can change. The number of tabs differs depending on what type of project that is active. To see a description of an option in the Options dialog box, click the question mark in the dialog box title bar, and then click the option.

Options file

The option settings can be saved in an options file, `.tot`. This file can later be edited.

In the installation there will be a number of files with a `.tot` extension containing internal framework settings and options. These files should normally not be edited by the user. Editing a file with a `.tot` extension may cause loss of data and incorrect behavior of the tool set. The options controlled should be edited from the Tools menu Options dialog.

Change options

To change options:

1. On the **Tools** menu, click **Options**.
2. In the **Options** dialog box, select and clear options in the different tabs. In the Advanced tab press F2 to access an option value.
3. Click **OK**.

Work with options Files

Save the current options in a new options file:

1. On the **Project** menu, click **Options** and then **Save As**.
2. In the Save As dialog, select a name and location for the options file (.tot).
3. Click **Save**.
4. Click **Yes** when you are asked to include the options file in your active project.

Model and Diagrams

Models

The model comprises all diagrams that describe your system. Different diagrams describe different aspects of the application. When modeling a system in UML, class diagrams describe the entities and the relationships between these entities.

Use case diagrams and use cases in form of sequence diagrams allows you to specify external interaction and an overview of a systems behavior.

Activity diagrams and Interaction overview diagrams can be used to describe parallel behavior in a model.

State machine diagrams describe the behavior of each active class and composite structure diagrams describe the external behavior of an entity and how the entity interacts with other entities.

The application is compiled from the model. Different diagram types show different views of the model. This means that entities that are available in the diagram exist in the model, but not necessarily the other way around.

Model elements

Removing a symbol, a presentation element, from a diagram will normally not result in the deletion of the corresponding entity in the model since an entity might be represented in more than one diagram.

However, deleting an entity in the model results in the deletion of the equivalent symbols in the diagrams since it is the model that represents the application and the diagrams only presents different aspects of the model.

Deletion of the presentation element will also delete the model element when there is a one-to-one relationship between the model element and the presentation element. A one-to-one relationship exists when a model element can only have one editable presentation, for example this is the case with many state machine symbols.

The model elements are shown in the Model View of the Workspace window.

See also

Chapter 2, Working with Models

Chapter 4, UML Language Guide

“Views” on page 15

Diagrams

A diagram is a representation of the model you are working with in UML. Depending on the type of diagram, you will be able to define different properties and actions.

Diagrams in general represent different views of a single model. There are a number of different types of diagrams. Their names are derived from UML concepts. Supported diagram types are:

- Activity Diagram
- Class diagram

- Component diagram
- Composite structure diagram
- Deployment diagram
- Interaction overview diagram
- Package diagram
- Sequence diagram
- State machine diagram
- Use case diagram

Using the diagrams

As there are several different types of diagrams available to describe the model, this section gives some hints how they can be used.

The order to perform activities when building a UML model is optional. A possible workflow is displayed in Figure 3 on page 28.

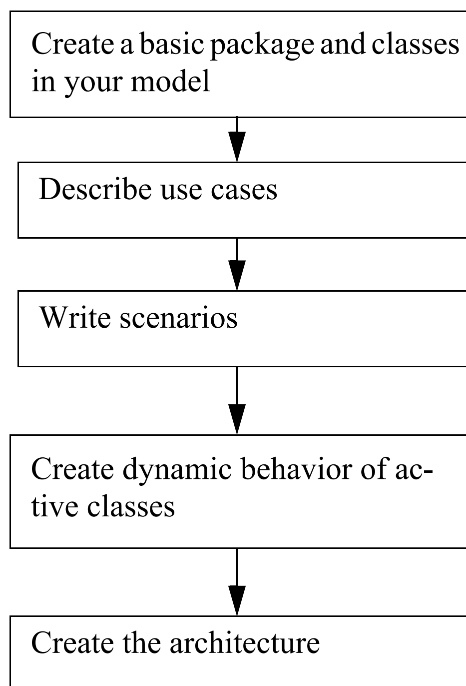


Figure 3: Diagram creation workflow

Create a basic package and classes in your model

It is possible to create new elements directly in the Model View by right-clicking on a package and from the shortcut menu select New and then choose the desired element from the submenu. To add a class diagram you right-click the package in the Model View and from the shortcut menu select **New** and then **Class diagram**.

Create use cases

A use case diagram can exist directly in a package, in a class or be grouped in a collaboration. A use case can be inserted directly in a package, in a class or in a collaboration.

Write scenarios

Scenarios in form of sequence diagrams is very easy-to-understand way of illustrating use cases. Syntax is simple and intuitive to understand, sequence diagrams is also a good basis for a dynamic behavior design.

Create dynamic behavior of the classes

The next step is to define the behavior of classes that you have set to be active. This is done using the state machine diagrams. For each Active class, add a state machine diagram to the model and when opened, build the internal state machine that defines the behavior of the class using the symbols available in the active toolbar.

Create architecture

The next step is to define how objects communicate. Instantiation of objects (parts) and communication between parts can be described with composite structure diagrams. Composite structure diagrams describe the internal structure of classes, attributes of classes and instantiations of classes.

The next step

To continue working, create a test project and get to know how DOORS Analyst lets you work with different diagram types, elements and symbols.

See also

"UML Language Guide" on page 139

"Working with Diagrams" on page 109

How to Use Help

This help file includes basic and advanced topics covering the supported functionality.

Additional product documentation is available in the section "Additional Resources" on page 423. There you can find tutorials, language descriptions, the installation guide and links to external sites, for instance the DOORS Analyst Support site.

Additional documentation in Adobe PDF format is also available on the installation CD.

Navigate in the help file

The help file contains functionality that helps you to easier find the information that you are looking for:

- "Search" on page 31
- "Search highlighting" on page 32
- "Index" on page 32
- "Locate search or index results" on page 33
- "Bookmark topics in the help file" on page 33
- "Print help topics" on page 33

Search

To perform a full-text search:

1. In the help viewer, click the **Search** tab.
2. Type your search string in the **Type in the word(s) to search for** field. You may use regular expressions, operators, and nested expressions when searching.
3. Optionally, you may check some of the following options: **Search previous result**, **Match similar words** and **Search titles only**.
4. Click **List Topics**.
5. To open a topic, double-click the topic in the Select topic list or select a topic and click **Display**.

Example 1:

To search for words beginning with “link”, type the following in the search field:

```
link*
```

Search highlighting

The words that you are searching for are highlighted on all pages where they are found. If you want to, you can turn off this functionality.

Turn off search highlighting:

1. In the help viewer, click the **Options** button and then click **Search Highlight Off**.
2. If you have already performed a search, click the **Display** button in the help viewer and the search highlighting disappear.

The search highlighting functionality is now turned off until you enable it again.

Turn on search highlighting:

1. In the help viewer, click the **Options** button and then click **Search Highlight On**.
2. If you have already performed a search, click the **Display** button in the help viewer and the search highlighting re-appear.

The search highlighting functionality is now turned on until you disable it again.

Index

To see the list of index entries, select the Index tab. To find the entry you are looking for, type the first letters of the word or scroll the list. To view the entry, double-click the entry or select the entry and click **Display**.

Locate search or index results

When you are using the search or the index functionality, the topic you are looking for will be displayed in the right-hand window. To locate where this topic is listed in the table of contents, click the **Locate** button. This allows you to easily find related topics or to learn where this topic is located the next time you are looking for it.

Bookmark topics in the help file

If you know that there are topics that you will refer to often or that there are topics that you consider important for your work, you can bookmark them as you would do in a regular web browser.

Bookmark a topic:

1. Find your topic using the Contents, Index or Search tabs.
2. Click the **Favorites** tab. The name of the topic is listed in the **Current topic** field.
3. Click **Add**. The topic is now displayed in the topics list.

Print help topics

You can print a single topic or you can select to print several topics within the same chapter.

Print an active topic:

- Right-click the displayed topic in the right-hand window and click **Print**. The print dialog opens.

Print a single topic from the table of contents

1. Right-click the topic window in the table of contents and click **Print**. The **Print Topics** dialog opens.
2. Click **Print the selected topic** and click OK. The Print dialog opens.

Print multiple topics:

1. Right-click a book icon in the table of contents and click **Print**. The **Print Topics** dialog opens.
2. Click **Print the selected heading and all sub-topics** and click **OK**. The print dialog opens.

Search syntax in help

The help viewer supports full text search, and you can search for any combination of letters (a-z) and numbers (0-9). Words like “the”, “a”, “and”, “but”, are reserved and cannot be searched for. In addition, you cannot search for punctuation marks such as colon (:), semicolon (;), hyphen (-) and period (.).

You can group search elements by using quotes and parenthesis.

Match similar words

The **Search** tab in the help viewer includes a **Match similar words** option. If you select this, you will be able to find all occurrences of a word, including common suffixes. For example, if you search for “run”, the words “run”, “running”, and “runner” will be found, but not “runtime”.

Regular expressions

The following regular expressions may be used when searching the help:

- * for matching 0 or more characters.
- ? for matching 1 characters.
- A string within quotes (“ab cd”) for matching the string literally.

Search for this:	Type this in the search field
Topics containing “analyze”, “analysis”, “analyses”, “analyzed”, and “analyzing”	analy*
Topics containing “analyzer” and “analyzed”, but not “analyze” or “analyzers”	analyze?
Topics containing the literal phrase “analyze and generate”	“analyze and generate”

Operators

You may use the following operators to refine a search in the help: AND, OR, NOT, and NEAR. The search string is evaluated from left to right. See table below for examples:

Search for this:	Type this in the search field
Topics containing both “work-space” and “file”	workspace AND file **or** workspace & file **or** workspace file
Topics containing either “work-space” or “file”	workspace OR file **or** workspace | file
Topics containing “workspace” but not “file”	workspace NOT file **or** workspace | file
Topics containing “workspace” and “file” close together, that is “work-space” within 8 words of “file”	workspace NEAR file
Topics containing “workspace” but not “file”, or topics containing “workspace” but not “directory”	workspace NOT file OR directory

Nested expressions

By using parentheses, you may nest expressions to perform a complex search in the help. An expression within parentheses will be evaluated first, before the rest of the search expression. Expressions may not be nested more than 5 levels.

Search for this:	Type this in the search field
Topics containing “workspace” without either of “file” or “directory”	workspace NOT (file OR directory)
Topics containing “workspace” with and “file” and “project” close together; or topics containing “workspace” with “directory” and “project” close together	workspace AND ((file OR directory) NEAR project)

UML Modeling

The chapters that are listed under UML Modeling describe functionality that is exclusive to UML projects.

2

Working with Models

This chapter is intended to give an introduction to model-based development. It contains the background to how model bindings are maintained. It explains the syntax color scheme for text information.

See also

“Working with Diagrams” on page 109

“UML Language Guide” on page 139

Models and Model Elements

Model-based development

The model-based nature of the UML tool set offers strong mechanisms to aid you in creating and maintaining complex models.

There are two different ways of working:

- **diagram-centric**, where you create your model as you are creating and editing the diagrams of the model.
- **model-centric**, where you create your model in the Model View browser and afterwards define your diagram views.

It is of course also possible to combine these two paradigms.

Diagram-centric workflow

The diagram-centric workflow is well known for users experienced with graphical languages. An example how this is carried out can look like the following:

- Create a diagram.
- Create the entities in that diagram.
- For the defined entities, create new diagrams that describe these entities in further detail.

A benefit with this approach is that you have a graphical context when you create new entities, which makes it easier to get it right.

Model-centric workflow

The model centric workflow is not dependent on the graphical presentations that may or may not exist for the definitions in a model. Example of workflow:

- Define model elements in the model browser.
- New model elements are placed within this model structure.
- Diagrams are created whenever needed or wanted to visualize relevant parts of the model.

- Visualizing entities in the diagrams is easily done by dragging an element from the Model View browser to the diagram.
- Model elements may be visualized several times, and in different diagram views.

One consequence of model-based development is that it is sometimes optional to describe entities within diagrams. If completeness of the diagram representation of the model is important, the tool can be configured to check for entities that are not represented graphically.

Model element and Presentation element

Model element

When creating a new definition by entering a new name on either a new object or an existing object, the tool will recognize that it does not exist in the model. This will create a new Model element. This model element will be visible in the Model view of the Workspace window.

Presentation element

A symbol in a diagram is a Presentation element, which is based on a model element. There can in many cases be any number of presentation elements to a given model element.

Element properties

Changing properties on a presentation element, like for example the name of an attribute in a class symbol, this change will also take place in other presentations of the class. The change has been done on the model element, and all presentations that show the affected property will be updated.

If you add a new attribute to the class (either in the model browser or in one of the presented class symbols), this will not automatically appear in all presentations. The attribute is of course available in the model so that it can be conveniently added in the class symbols where it is wanted to visualize this property.

Delete

If you delete an attribute in a class symbol, this will only remove the presentation of the attribute in that symbol.

Delete from Model

When you delete the attribute in the Model View, this will delete the model element for the attribute and subsequently all presentation elements of the attribute will disappear (it is also possible to right-click on a presentation element in a diagram and use the shortcut menu command **Delete from Model**).

If you delete a class that is referenced in other places, for example as an attribute type in other classes, these references will become unbound when the class is deleted.

Model element

Automatic naming of new elements

When adding new symbols that can define model elements, a default name is created for the symbol so that the name is unique in the current scope. You can just start typing (without selecting the text) to change the given name to your wanted name.

Copying and moving model elements

A model element can be copied and pasted.

If you copy a symbol that references a model element and paste this symbol, this will just give a new presentation of the existing model element. Changing the name in one of these two symbols will also change the name in the other symbol: they are both presentations of the same element.

If you copy a model element in the browser, you can paste this in another place (here it is a copy of the model element itself). If you paste into the same scope you will have two conflicting definitions with the same name, something the checker will report. If you change the name of one of the model elements, the conflict will be resolved.

Just as model elements can be copied, they can also be moved, typically by dragging from one scope to another.

Text Highlighting

Object Location

In several places in a UML model, it is possible to locate a definition, for example by double-clicking an object in the Model View or in an output pane, or by choosing the Locate command for an object in an output pane. When this is performed, the correct diagram appears with the definition highlighted by yellow text background. Alternatively, if there are several possible presentations of the definition (or none), the Create Presentation is activated.

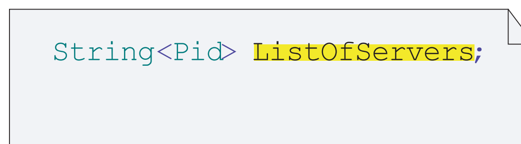


Figure 4: Location marker

Name navigation

It is possible to navigate on names by holding CTRL and clicking on non-gray representations of names with no underline. The tool-tip also indicates this possibility.

Properties

The complete set of properties of a model element is not always possible to edit through its presentation elements in diagrams. For this purpose there is a Properties Editor in which it is possible to view and edit the properties of a model element. The Properties editor is opened from the shortcut menu (right-click on an element and point to **Properties...**), or press ALT + ENTER.

Model checking

Syntax parse

When you edit text in diagram symbols, if the text is parsed correctly, the text is added to the model. After that, the text will be written back to the diagram symbol again based on the model.

This Text parsing is a consequence of the tight model-based approach. In some cases it will be written to one specific (of several possible) syntax alternative, thus not preserving your exact formatting.

Restore model (F8)

It is possible to restore a model during text editing when the changes are not found correct by the syntax parser. This is done with the command F8, while the selection for the syntax error is still active in text edit. This will restore the text from the model information.

This command should be used with care. It will erase any text that is not bound to the model, like comments. When the model is first created and no correct model has been parsed, this command will erase everything.

Name support

There are different ways you can get help from the tool when you want to reference a definition.

- Create Presentation lets you browse and navigate quickly through your complete model.
- **Name completion**

  After typing the first letters of the name, for example `ca`, pressing CTRL + SPACEBAR the tool will try to complete the name to an existing name, for example `card`. If there are multiple matches a Name completion scroll menu will open. Some special cases can be identified

  – Typing after a period ('.'). Name completion will list candidates matching the written characters that are local or inherited members (structural features or event classes) to the type of the left side expression.

  – Typing after a scope qualifier ('::'). Name completion will list candidates that are in the namespace of the left side expression.
- **Reference existing**

  When creating a new symbol (that can define or reference a model element) using the Diagram element creation toolbar and pressing the right mouse button in the diagram, the shortcut menu appears, with a submenu called Reference existing. This submenu contains a list of all visible definitions of the symbol kind, so that the wanted identifier can be chosen.

Checking a part of a model

Select the part in the Model View to be checked. Use **Check selection** quick button (in toolbar Analyzing).

Errors and warnings

If any problems are detected during a check of the model, this will be reported in the Check tab in the Output window. Each problem (warnings and errors) can normally be traced back to its origin, either in a diagram, or in the Model View browser. This is done by double-clicking the message or by selecting the message and right-click, then choose **Locate** in the shortcut menu.

Models and Diagrams

Diagrams

Different views of the model

Diagrams are presentations of a model, typically focusing on one particular aspect and part of the model. One of the powers of UML is the capability to give different views of a model. This means that model elements are referenced in several places. Normally, this could be a problem when maintaining the model, but with the strong model-based tool support, all references are automatically kept up-to-date, that is if properties of a model element change, these changes will be reflected in all places where the element is referenced.

Presentation element

Symbols

Symbols are **presentation elements** that differ from model elements. If a symbol is deleted, the model element is still present in the model. The model element will be deleted when one of the following applies:

- the element is deleted in the Model View browser
- the command Delete from Model is performed on the symbol.

If you change the name of an attribute in a class symbol, this change will also take place in other presentations of the class.

If you add a new attribute to the class (either in the model browser or in one of the presented class symbols), this will not automatically appear in all presentations. The attribute is of course available in the model so that it can be conveniently added in the class symbols where it is wanted to visualize this property.

If you Delete an attribute in a class symbol, this will only remove the presentation of the attribute in that symbol. If you delete the attribute in the Model View browser, all presentations of the attribute in different class symbols will disappear.

If you delete the class itself in the Model View browser, the symbols referring to this class will disappear from all diagrams. If the class is referenced in other places, for example as an attribute type in other classes, these references will become unbound when the class is deleted.

Properties Editor

Opening the Properties Editor

The Properties Editor is opened by selecting an element in the Model View or in a diagram, and selecting “Properties...” in the shortcut menu. The Properties Editor will open as a docked window. Similar to other editors it can be undocked, or docked at a different location by right-clicking in its title bar. The Properties Editor will stay open until you close it.

Multiple windows

It is possible to open more than one Properties Editor. This can for example be useful when comparing the values of properties on different elements. To enable this you must deactivate Track selection for one of the Properties Editor windows.

The Properties Editor View

The view of the Properties Editor consists of the following areas from top to bottom (see Figure 5 on page 48):

- In the top left of the window is shown the selected element, element name and icon.
- An “Options...” button for setting Properties Editor Options for the current window.
- A Filter selection menu that controls which properties of the element that are displayed.
- A “Stereotypes...” button for controlling which stereotypes that are applied to the element. The dialog that is opened when pressing this button is the same as is opened when the **“Stereotypes...”** menu item is chosen in the shortcut menu of an element.
- Controls for viewing and editing properties of the element. This area is dynamically populated with controls based on the edited element and the selected filter.

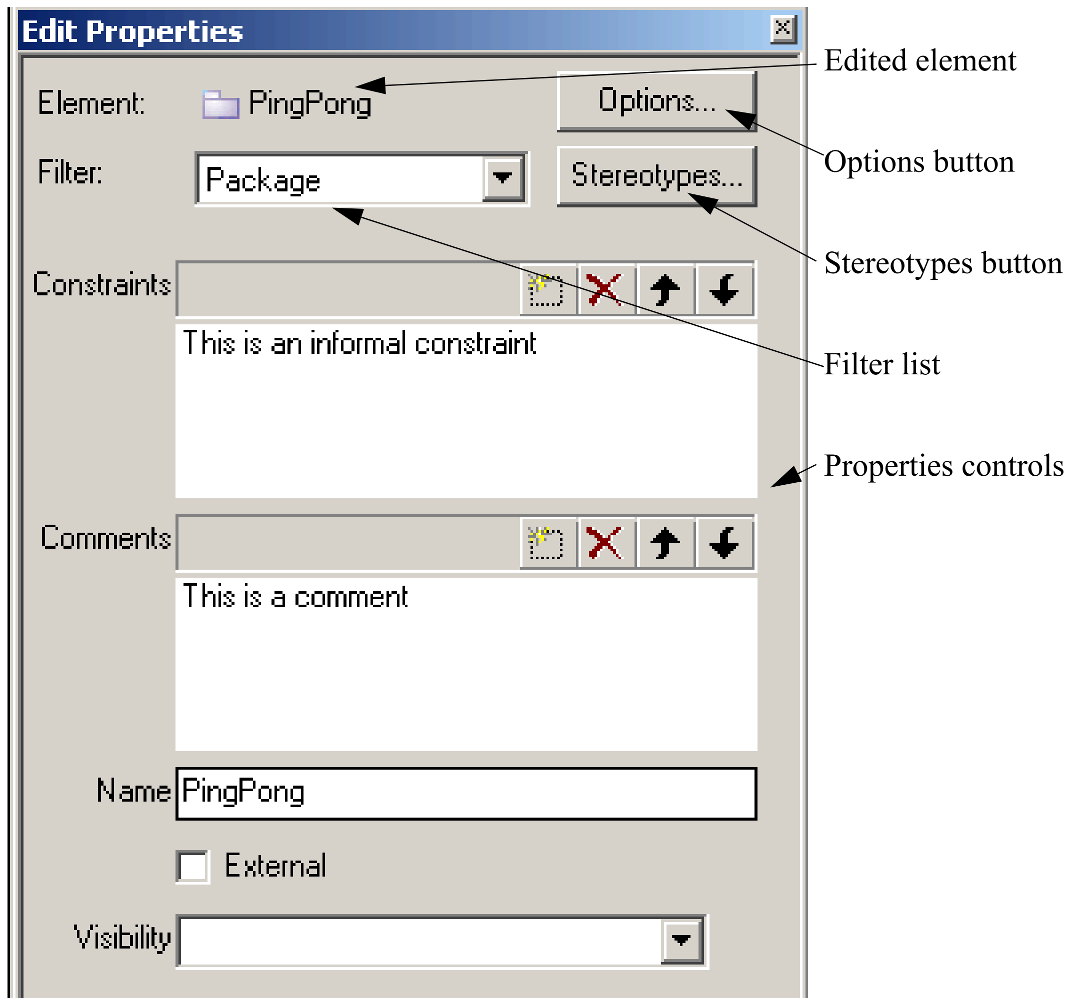


Figure 5: Properties editor.

The Filter list consists of the following items (not necessarily in this order though):

- Name of the Metaclass of the edited element.
  When this item is selected the Properties Editor will display the Metafeature values of the element (see “Different Kinds of Properties” on page 49).
- Name of each stereotype that is applied to the edited element.
  This includes both stereotypes with optional (`0..1`) and non-optional (`1`) extension to their metaclass. See “Extensibility” on page 316 for more information about stereotype extensions. However, hidden stereotypes (a stereotype which has the <<hidden>> stereotype applied) are not listed.

- “Comment”
  When this item is selected the Properties Editor will display the comment that is attached to the edited element. If no comment is attached, a button will appear that lets you create a comment for the element. If multiple comments are attached to the element, the first comment will be displayed.
- “All Properties”
  When this item is selected the set of properties is not filtered, and the Properties Editor will display all properties for the edited element. The order of the property controls will be the same as the order of the corresponding items in the Filter list.

Properties Editor view when selecting an instance

When an instance is selected some of the standard controls of the Properties Editor view described above no longer make sense, and will therefore be removed:

- The Stereotypes button is removed, because it is not possible to apply stereotypes on instances.
- The Options button is removed, because some options are not applicable for instances.
- The Filter list is replaced with information about the signature (e.g. a class) of the instance.

The typical case when you will see this modified Properties Editor view is when an instance is selected in the Model View, for example a stereotype instance. However, an instance can also be selected when editing a tagged value for an attribute typed by a structured type, such as a class.

Different Kinds of Properties

There are in principle two different kinds of properties that can be associated with the selected element, Metafeature values and Tagged values. The Properties Editor can edit both kinds of properties.

Metafeature values

These are values for the metafeatures of the element’s Metaclass.
The set of metafeatures for an element is fixed (and to some extent dictated by the UML standard), so it is not possible to add new metafeatures. The ex-

isting set of metafeatures can however be filtered so that only values for some metafeatures are displayed in the Properties Editor.
An example of a metafeature value is the “Active” property of a class.

Tagged values

These are values for attributes of the stereotypes that are applied to the element. Contrary to Metafeature values the number of tagged values on an element can be arbitrary large since it is possible to apply any number of stereotypes on an element and each of these stereotypes may have any number of attributes.
An example of a tagged value is the “Icon File” property that lets you specify an icon to be displayed for a symbol. Another example is shown in Figure 9 on page 56.

Properties Editor Options

The Properties Editor Options dialog is reached from the “Options” button of the Properties Editor, see Figure 6 on page 51. The options that are set in this dialog only affects the current Properties Editor, and only as long as it stays open. This means that it is possible to have two different Properties Editors open each of which uses a different set of options.

Note

Some of the options are also available in the general Options, allowing you to set and store options for all Properties Editors that are opened. The values of some options can also be modified using the General Shortcut Menu of the Properties Editor.

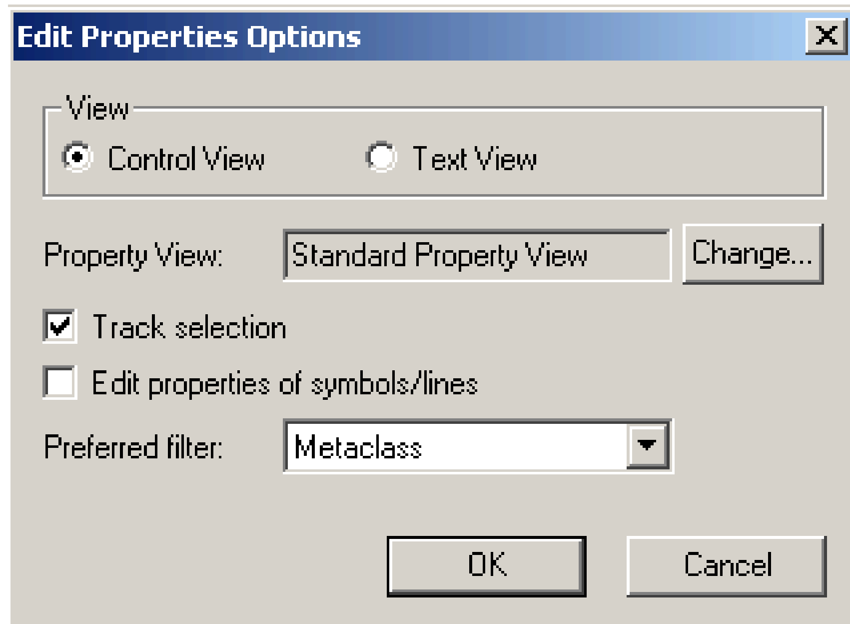


Figure 6: Options dialog for Properties editor.

View

By default the Properties Editor uses the Control View for editing property values. This view contains controls for editing element properties in form of check boxes, pull-down menus etc. For Tagged values textual editing in UML syntax is also possible. This is supported by using the Text View.

Text field values entered in the Control view will be committed to the model when you leave edit mode for the field.

Property view

The Properties Editor can be customized using a Metamodel. Such a metamodel controls for example which metafeatures that are available for each Metaclass, and the Properties Editor will use this information when deciding which properties to display for an element. See "Customizing the Properties Editor" on page 58 for more information on how to use metamodels.

Track selection

By default the Properties Editor will show properties for the element that is selected in the Model View or in the diagrams. Sometimes it is useful to turn off this selection tracking to be able to compare the properties of two dif-

ferent elements. This is done by opening the Properties editor for the element you want to be fixed. Then point to the **Options...** button and in the dialog make sure that **Track selection** is not selected. Now you can open another Properties editor for any other element, this new properties window will then track your selection in the model.

Edit properties of symbols/lines

If a symbol or line is selected the Properties Editor will by default show the properties for the model element that corresponds to that symbol or line. In order to show the properties of the selected symbol or line instead this option should be selected.

For example, if a class symbol is selected, the Properties Editor will normally show the properties for the corresponding class. However, if the **Edit properties of symbols/lines** option is set, the properties for the class symbol will be shown instead.

Preferred filter

This option controls which filter item that is the preferred when a new element is selected. Available items are Metaclass, Stereotype, Comment and All Properties. The option will take effect the next time the edited item is changed.

General Shortcut Menu

The Properties Editor has a shortcut menu that will appear if the right mouse button is clicked in the editor view outside a control. The menu is shown in Figure 7 on page 52.

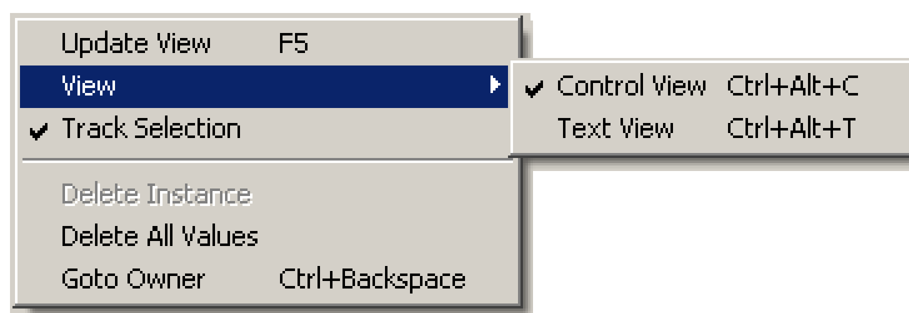


Figure 7: Shortcut menu in Properties editor.

Update View

Refreshes the Properties Editor view. The Properties Editor will normally update its view automatically if values are changed from outside the Properties Editor. However, there are situations when it is necessary to force an update, for example if new attributes are added to an applied stereotype whose attribute values are currently displayed, or if the active property Metamodel is changed when the Properties Editor stays open.

View

This menu item is a shortcut for the corresponding option in the Options dialog, see “Properties Editor Options” on page 50.

Track Selection

This menu item is a shortcut for the corresponding option in the Options dialog, see “Properties Editor Options” on page 50.

Delete Instance

This menu item is available when editing Tagged values for one single applied stereotype. It will delete the entire stereotype instance, effectively removing all tagged values contained in that instance. It can be seen as a shortcut for opening the “Stereotypes” dialog and removing the edited stereotype from the list of applied stereotypes.

Delete All Values

This menu item will delete the values of all displayed properties. Those properties that have a default value will obtain that value, others will be unspecified. In the case of editing Tagged values, this menu item will remove all tagged values, but keep the applied stereotype instance.

Goto Owner

This is a convenient shortcut for going to the property page of the edited element’s owner. For example, if the properties of a class attribute is edited, “Goto Owner” will display the properties for the attribute’s owner, i.e. the class.

Note

Some menu items of the Properties Editor shortcut menu are not available when an instance is selected for editing.

Control Shortcut Menu

There is also a shortcut menu for each property control. The exact contents of this menu depends on the kind of property control. For example the edit controls have the standard Cut/Copy/Paste menu items. The menu items shown in Figure 8 on page 54 are common for all property controls.

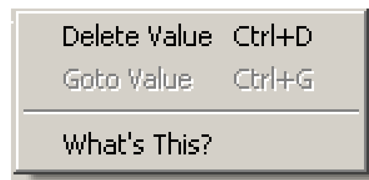


Figure 8: Property control, shortcut menu example.

Delete Value

This menu item will delete the value of the property control. If the property has a default value, it will obtain that default value, otherwise it will get an unspecified value.

Goto Value

The list controls may show a value that is a list of other elements in the model. For such controls the Goto Value menu item will navigate to the selected element of the list.

For example, most elements have a Comments list which display the list of all comments that are attached to the edited element. If one of these comments are selected, the Goto Value menu item will be enabled and if chosen the Properties Editor will display the properties of the selected Comment instead (it typically just has one property - the comment text).

What’s This?

If the attribute that corresponds to the property control (i.e. an attribute in a stereotype or in a Metaclass) has a comment attached, this menu item will be enabled. If chosen, that comment will be displayed in a tool tip. Stereotype

and Metamodel designers should use the possibility to add comments to stereotype and metaclass attributes in order to help the user of the stereotype or metaclass to know which value that should be entered in the property control.

For certain controls (for example those showing Metafeature values) a standard What's This? text may be displayed even if the attribute has no comment attached. Such a text appears when the value to be entered in the control is text that is translated into model elements. The tool tip then displays the kind of text that should be entered into the control. For example, if a UML expression should be entered in the control the tool tip may say “Expression”.

Color Codes

When editing Tagged values (i.e. not Metafeature values) the Properties Editor uses a color coding scheme for showing the status of a tagged value.

A tagged value that has been specified explicitly in the applied stereotype instance is indicated by displaying the property control in a white color.

A tagged value that is unspecified in the stereotype instance, but for which the corresponding stereotype attribute has a default value, is indicated by displaying the property control in a green color.

A tagged value that is unspecified in the stereotype instance, and for which the corresponding stereotype attribute has no default value specified, is indicated by displaying the property control in a yellow color.

These color codes should be used by the designer of a stereotype, to express the intent of the stereotype attributes to the user of the stereotype. A green value signals that it is optional to specify a value for the attribute, since there is an appropriate default value. A yellow value signals that the user should specify a value, since no appropriate default value is available for that particular attribute.

Example 2: Stereotype with colored attribute fields

Consider a stereotype with three attributes. In Figure 9 on page 56 the stereotype `MyStereo` is applied to a class X. The user specifies a value for the second attribute, thus the color for this field will change from yellow to white.

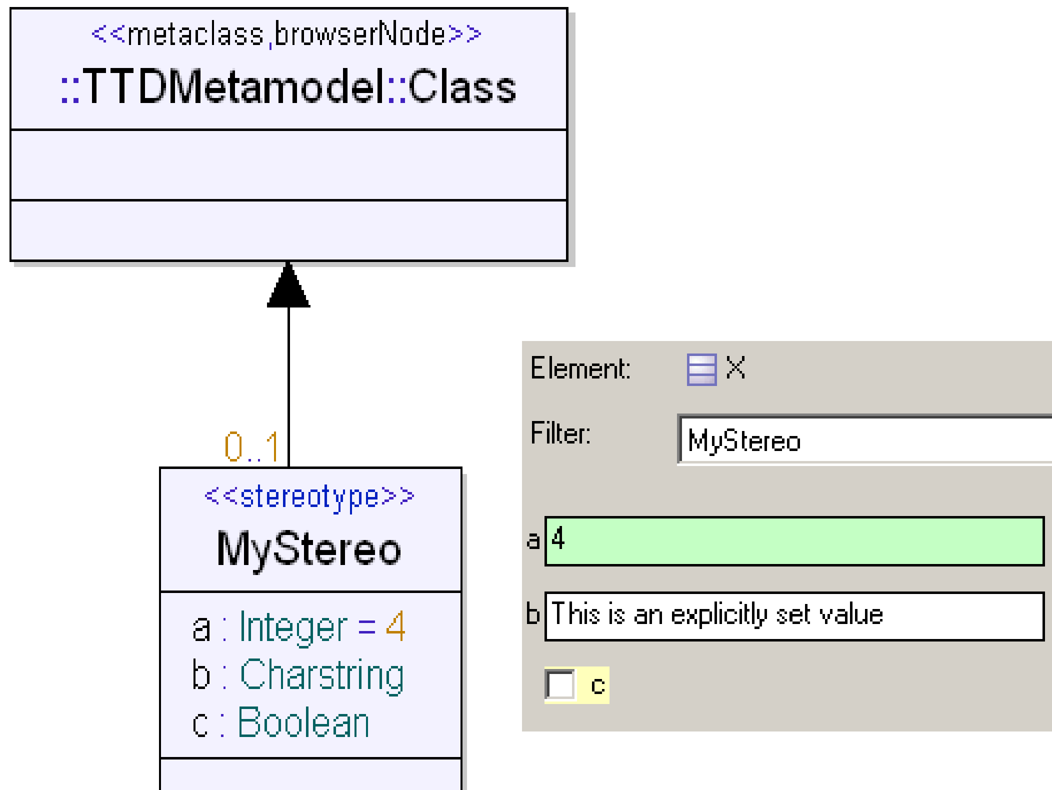


Figure 9: Stereotype with attributes.

Another colorization that is used is to show whether the text of a control contains a syntax error. Such syntax checks are made for all controls whose text must comply with the U2 textual syntax grammar. Text containing a syntax error will be shown in red, while correct text will be black. If you leave editing while the text for such a control is red, the value will go back to its previously correct value. This colorization is thus a help to avoid accidentally loosing information while editing.

Example 3: Syntax error colorization in the Properties Editor

The 'Realizes' metafeature of a Port expects a list of identifiers. The current text (see Figure 10 on page 57) for the metafeature contains a syntax error since 'signal' is a UML keyword. Hence, if leaving edit mode now, that value will go back to the previously correct value (whatever that is).

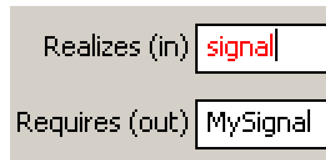


Figure 10: Correct and incorrect metafeature values.

Customizing the Properties Editor

When designing a stereotype to be applied to an element, two user roles can be identified; the designer of the stereotype who decides which attributes the stereotype shall have, and the user of the stereotype who applies it to an element and specifies Tagged values for the stereotype attributes. Although these roles could be possessed by the same individual it is very common that the designer and the user of a stereotype are two different people.

This section will address the designer role, describing how to design a new stereotype or Metaclass. This also includes how to utilize the various possibilities for customizing the Properties Editor to edit instances of these stereotypes and metaclasses in the way the designer wants.

The Properties Editor uses a profile to control most of its customization, called the TTDExtensionManagement Profile, and is available in the Library folder of any model.

Designing a Stereotype

The following steps should be taken in order to design a stereotype for use with the Properties Editor:

- Decide where to place the definition of the new stereotype. If the stereotype is only intended to be used locally within the current project, it could be added in the same file as the elements on which it should be applied. However, it is typical that a stereotype shall be used in multiple projects, and then it should be placed in a package that is stored in a file of its own. Such a reusable package with stereotypes is typically a so called profile package. See “Add-Ins” on page 1987 for more information on how to load such a package as a library in the tool.
- Give the stereotype a good name. The name of the stereotype will appear in the Stereotypes dialog, in the filter list of the Properties Editor and in some symbols in the diagrams. Sometimes it can be useful to use the `TTDExtensionManagement::instancePresentation` stereotype in order to specify a more user-friendly display name for the stereotype. Such a specified display name will be shown in the Stereotypes dialog and in the filter list of the Properties Editor. See “TTDExtensionManagement Profile” on page 63 for more information.

- Make a comment for the stereotype. This comment should describe the purpose of the stereotype, any constraints on elements onto which it can be applied and so on. The comment will be displayed at the bottom of the Stereotypes dialog, when the stereotype is selected. It will also be displayed as a tool tip for presentations of the stereotype.
- Add attributes with appropriate types and multiplicities to the stereotypes. A stereotype attribute may have any type and Multiplicity, but you should be aware of the subset of types and multiplicities that are supported by the Properties Editor when using its Control View for editing. If an attribute has a non-supported type or multiplicity, values for that attribute cannot be edited in the Control View. Instead the Text View has to be used.

The table below specifies the supported combinations of types and multiplicities, and which graphical control that will be used in each case. See also the table in section "Designing a Metaclass" on page 61 in Chapter 2, *Working with Models* for a listing of the supported combinations of types and multiplicities that are applicable for attributes in metaclasses only.

Attribute type and multiplicity	**Property control**
Boolean Single multiplicity	CheckBox
Charstring Single multiplicity	EditControl
Charstring Non-single multiplicity	EditList
Enumeration Single multiplicity	DropDownMenu (one item for each literal)
Enumeration Non-single multiplicity	CheckBoxList (one check box for each literal)
Structured type (e.g. a class) Non-optional, single multiplicity (1) Attribute is a part (composition)	Group (with one subcontrol for each attribute of the structured type)

Attribute type and multiplicity	**Property control**
Structured type (e.g. a class) Optional, single multiplicity (0..1)	InstanceEditControl
Structured type (e.g. a class) Non-single multiplicity	InstanceEditList
Metaclass type Single multiplicity Reference	DropDownMenu (one item for each visible definition of the metaclass)
Metaclass type Non-single multiplicity Reference	EditControl
All other types Single multiplicity	EditControl (expecting a U2 expression)
All other types Non-single multiplicity	EditControl (expecting a comma-separated list of U2 expressions)

Naturally, a syntype of any of the above mentioned types are also supported.

- In case the default control for an attribute is not appropriate you may apply the `TTDExtensionManagement::extensionPresentation` stereotype to an attribute, specifying a custom control as a tagged value. See “TTDExtensionManagement Profile” on page 63 for more information.
- It is possible to add additional “non-value” controls to the property page of the stereotype. For example you could add a static text or a button to the property page. This is done by applying the TTD`ExtensionManagement::instancePresentation` stereotype to the stereotype and specifying the additional controls as Tagged values for the `nonValueControls` attribute.
- Use the possibility to attach a comment to each stereotype attribute. The comment text will be visible to the user of the stereotype in the What’s This shortcut menu item on the control that corresponds to the attribute.
- Consider the possibility of using inheritance between stereotypes. The property page for the derived stereotype will include all base stereotype attributes followed by the attributes of the derived stereotype.

- Specify the kind of elements onto which the stereotype shall be applicable. This is done by establishing an Extension between the stereotype and a Metaclass. The meaning of this is that the stereotype will be applicable to all elements of the specified metaclass. The UML semantics state that if a stereotype lacks extensions, it cannot be applied to any kind of element.
  If you want the stereotype to be automatically available for all instances of the extended metaclass, you should make the extension non-optional (type "1" on the extension line). Thereby the stereotype will be available in the filter list of the Properties Editor without first having to apply it to the edited element.
  If you want the stereotype to be manually applied, you should make the extension optional (type "`0..1`" on the extension line).
  It is allowed to use multiple extensions. The stereotype will be available for all elements that is of any of the specified metaclasses.

Now you are ready to test the new stereotype. Create an element of the correct kind, i.e. an element of a metaclass that is extended by the stereotype. Make sure the stereotype is visible from the location of the created element. Open the Properties Editor on the created element and take a look at the property page for the new stereotype. If you specified an optional extension you should first apply the stereotype, using the "**Stereotypes...**" button.

Designing a Metaclass

The process of designing a Metaclass is almost the same as when designing a stereotype. The main difference is how to specify the elements for which the metaclass shall be available in the Properties Editor. For a metaclass this is done by applying the `<<metaclass>>` stereotype to the class that describes the metaclass. It is in fact this step that makes it a metaclass instead of an ordinary class. The tagged value for the `base` attribute shall specify the name of the built-in UML metaclass on which the new metaclass shall be based.

Note

A good starting point for learning how to design a metaclass, is to study the TTDMetamodel profile that is available as a library in all models. Here you can find information about the names of the built-in metaclasses and metafeatures to be used as base for your own metaclasses and their attributes. You can also see example of use of the TTDExtensionManagement Profile for customizing the Properties Editor for elements of the specified metaclasses.
It is TTDMetamodel that is referred to as "Standard Property View" in the Options dialog of the Properties Editor.

In contrast to a stereotype it is not possible for a metaclass to specify plain new attributes. All attributes of a metaclass must be based on already existing metafeatures of the base metaclass. This is done by applying the `metafeature` stereotype to the metaclass attributes. If the name of the metaclass attribute is the same as the name of the corresponding metafeature, the `base` tagged value can be omitted. Otherwise it has to be specified.

Note

The careful user will find some metaclass attributes in TTDMetamodel which do not correspond to metafeatures of the base metaclass. These are so called query features, and they use the <<queryFeature>> stereotype to specify a query agent that computes entities from the model. Query features are not displayed in the Properties Editor - only in Model View.

The table below specifies the supported combinations of types and multiplicities that are applicable for attributes in metaclasses only, and which graphical control that will be used in each case. Compare with the table in section Designing a Stereotype for a listing of combinations that are valid for all Stereotype attributes.

Attribute type and multiplicity	Property control
Metaclass type Single multiplicity Composition	EditControl
Metaclass type Non-single multiplicity Composition	EntityList

When your new metaclass is ready you will have to place it in a package and store the package in a file of its own. The predefined stereotype `<<propertyModel>>` should be applied on the package. Then you should follow the normal procedure for writing Add-Ins that loads the profile. When the profile has been loaded you can use the "Options..." button of the Properties Editor to specify the profile package as the property view to use with the Properties Editor.

TTDExtensionManagement Profile

The TTDExtensionManagement profile contains stereotypes and classes that allows you to customize the property pages for your own stereotypes and metaclasses. Here are the details of this profile, and also some examples of how to use it.

Stereotypes

The profile contains three stereotypes that are relevant for the Properties Editor: `instancePresentation`, `extensionPresentation` and `filterStereotypes`.

instancePresentation

The `instancePresentation` stereotype may be applied on a stereotype or Metaclass to customize how instances of the stereotype or metaclass will be presented in the Properties Editor.

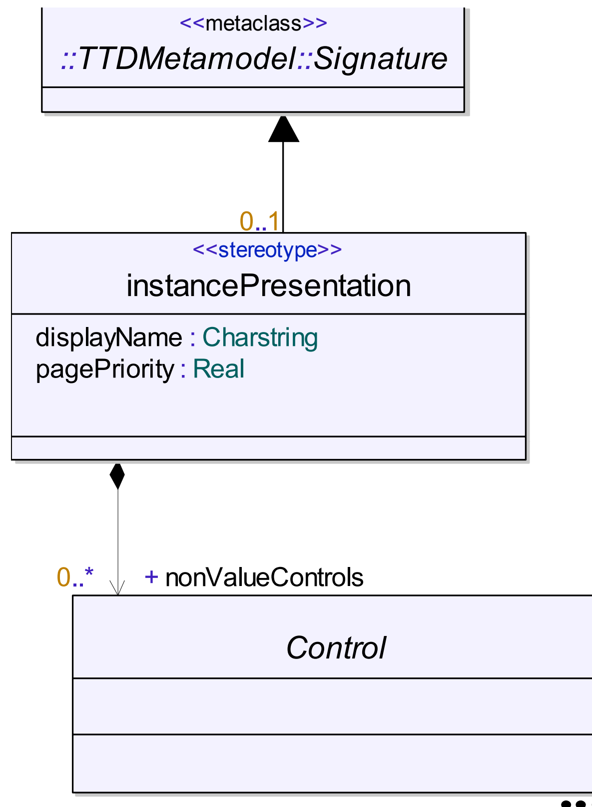


Figure 11: The <<instancePresentation>> stereotype

displayName: Charstring

This attribute specifies the display name of instances of the stereotype or metaclass. It is used in the filter list of the Properties Editor and in the Stereotypes dialog. It is also used in some other places in the tool, such as in tool tips and in the Model View.

If no tagged value is specified for this attribute, the name of the stereotype or metaclass will be used as display name.

pagePriority: Real

This attribute controls the relative order of two stereotype instances in the filter list of the Properties Editor and in the property page (if the All Properties filter is used). An instance of a stereotype with a higher page priority will be placed before an instance of a stereotype with a lower page priority number. Any specified page priority is considered to be a higher priority value than an unspecified page priority.

Note

If you want to specify a page priority you must use a single numeric value. More complex expressions will not be evaluated.

nonValueControls: Control[*]

This attribute specifies a list of “non-value” controls, i.e. controls in a property page that do not correspond to a particular attribute. Examples of such controls are “adornments” such as static texts, but it could also be controls with some behavior, such as a Button.

extensionPresentation

The `extensionPresentation` stereotype (Figure 12 on page 65) may be applied on an attribute of a stereotype or a Metaclass to customize how the Properties Editor will draw the control that corresponds to that attribute.

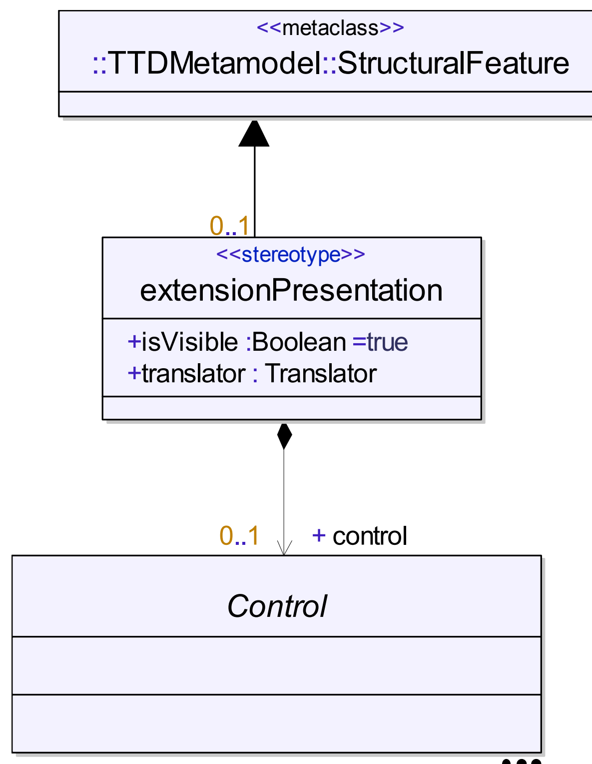


Figure 12: The <<extensionPresentation>> stereotype

isVisible: Boolean

This attribute controls whether the control for the attribute shall be visible on the property page or not. You may set its value to false in order to hide the control for an attribute completely.

translator: Translator

This attribute is used exclusively for parts typed by a metaclass. As mentioned in "Designing a Metaclass" on page 61 the Properties Editor uses an EditControl (in case of single multiplicity) or an EntityList (in case of non-single multiplicity) as the control for such an attribute. Since the text that is entered into these controls in this case is UML textual syntax, a parser (translator) is needed to interpret the text. The Translator enumeration contains one literal for each available entry point of the UML grammar. Although the Translator enumeration resides in a hidden (internal) profile, you can find out the names of its literals with the following procedure:

- Create an enumeration symbol in a class diagram by right-clicking and choosing Reference existing.
- In the list that appears, select the `U2ParserProfile::Translator`.
- Right-click on the enumeration symbol and choose "Show Literals" from the Show/Hide submenu.

Note

Use the List References command (available in the shortcut menu) to find out how the Translator literals are used in the TTDMetamodel profile. For example, listing the references for the literal PEP_Multiplicity shows that it is used as the translator of the StructuralFeature::Multiplicity attribute. Thus, this translator is used for parsing the multiplicity syntax of UML.

control: Control[`0..1`]

As mentioned in "Designing a Stereotype" on page 58 the Properties Editor uses a default control based on the type and multiplicity, and sometimes also the aggregation kind, of an attribute. The `control` attribute makes it possible to specify that a non-default control shall be used for an attribute, or that some properties of the default control should be changed.

Example 4: Specifying a custom control using the Text View

```
extensionPresentation(.
  control = EditControl(.
```

```
        text = "My Control",
        autoLayout = GrowRight
      .)
    .)
```

The Control class is an abstract class with one derived class for each control that is supported by the Properties Editor.

filterStereotypes

The `filterStereotypes` stereotype may be applied on a package to reduce the number of stereotypes that will be shown in the Properties Editor when an element in that package is selected.

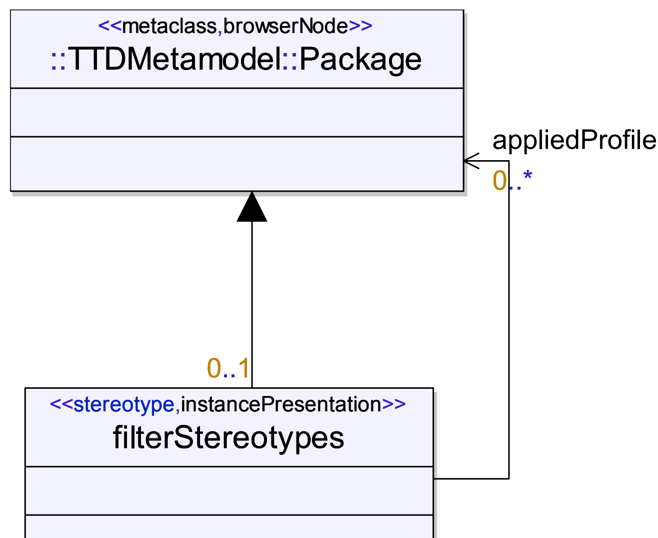


Figure 13: The <<filterStereotypes>> stereotype

appliedProfile: Package[*]

If this list of profile packages is specified, the Properties Editor and the Stereotypes dialog will only show stereotypes defined in these packages for a selected element in the package on which the `filterStereotypes` stereotype is applied.

Control model

The `TTDExtensionManagement` profile contains a number of Control classes representing graphical controls used by the Properties Editor. See the class diagram `Controls` for an overview of all available control classes.

Control

The `Control` class is a common base class for all control classes.

text: Charstring

This attribute controls which caption to use for the control. If it is left unspecified, the caption will be the name of the edited stereotype or metaclass attribute.

isEnabled: Boolean

By default a control will be enabled, meaning that it can be used for editing the displayed value. If this attribute is set to false, the control will instead be disabled. In some situations the Properties Editor will force a control to be disabled, regardless of the value for this attribute. This happens if the file that contains the edited element is read-only, and also for attributes that are derived.

onEnable: Operation

This attribute can be used to give a dynamic condition for when a control shall be enabled. If an agent operation is specified here it will be called each time the Properties Editor needs to decide whether the control shall be enabled or not. The model context of the agent call is the edited element. The call has the following parameters:

- `[out] enable : Boolean`
  The agent should set this out parameter to false if the control shall be disabled. By default the control will be enabled.
- `stereotypeInstance : Entity`
  The stereotype instance that is edited in the property page containing the control. This parameter is only passed when the edited instance is a stereotype instance.

Note

When `isEnabled` *is false the* `onEnable` *agent will not be invoked.*

See also

"Agents" on page 2025 in Chapter 75, *Agents*

Button

The `Button` class represents a button that can be pushed. It is not used for editing a value, but can be used as a non-value control on a property page. `CheckBox` is a special kind of button, a toggle box control, which can be used to edit boolean values.

onClicked: Operation

This attribute can be used to specify some behavior to be executed when the button is clicked. It may specify an agent operation, which will be invoked when the button is clicked. The model context of the agent call is the edited element. There are no parameters in the agent call.

EditControl

An `EditControl` can be used for editing string values. There are two specialized versions of the class that can be used when the edited string is a directory name or a filename. They will add a browse button [...] for opening a directory or file selection dialog, as an alternative for manually typing the name in the control.

There is also a special kind of EditControl called `InstanceEditControl`. It is used for editing instances (for example instances of classes). The instance is shown using textual syntax in the control, but to edit the instance there is a browse button [...] which will bring up another Property Editor for editing the selected instance.

isMultiLine: Boolean

By default an edit control shows exactly one line of text. By setting this attribute to true, the control will enable multiline editing. In order to see more than one line of text at the same time, the vertical size of the control may have to be extended. See PositionedControl for more information on how to do this.

EditList

An `EditList` control can be used to edit lists of strings. The control contains buttons for creating a new string in the list and for deleting a selected string from the list. There are also two buttons for moving a selected string up or down in the list. A string can also be moved by drag and drop in the list directly.

Figure 14: An EditList control with buttons for creating, deleting and moving strings.

There are two specialized versions of EditList that can be used when the edited strings are directory or file names. They are called `DirectoryEditList` and `FileEditList` and will add a browse button [...] for opening a directory or file selection dialog, as an alternative for manually typing the name in the control.

There is also a special kind of EditList known as an `EntityList`, that can be used as the control for metaclass attributes (i.e. metafeatures) that are compositions typed by another metaclass. Each edited item in an EntityList is thus an element in the model. The string displayed in the control for such an element is its textual UML syntax.

Another special kind of EditList is the `InstanceEditList` which is used for editing a list of instances (for example instances of classes). The instances are shown using textual syntax in the control (one instance on each line). To edit one of the instance double-click on it, and press the browse button [...] which appears. Doing so will bring up another Property Editor for editing the selected instance.

StaticText

A `StaticText` is a non-value control that can be used as an adornment in a property page. It can for example be useful to add a static text for giving instructions to the Properties Editor user on how to specify values in the supplied controls.

EnumeratedList

This is an abstract class that is the common base for controls that edit lists of enumerated elements. There are two concrete specializations of this class; DropDownMenu and CheckBoxList.

items: Charstring[*]

If an enumerated list is used as the control for an attribute that is typed by an enumeration, it will contain one item for each literal of the enumeration. The name of each item will by default be the name of the corresponding literal. However, by specifying a list of strings as the value of the `items` attribute the names of the list items can be customized.

DropDownMenu

A `DropDownMenu` is a list of items edited in a drop down menu.

isEditable: Boolean

By default the user can only select one of the existing items from a drop down menu. By setting this attribute to true, the drop down menu will be editable, allowing the user to type the name of the item manually.

CheckBoxList

A `CheckBoxList` is a list of items edited in a list of check boxes. Hence this control allows multiple list items to be selected.

Group

A `Group` is just a container control that can contain other controls. It is typically used as the control for a part attribute of multiplicity 1 typed by a structured type. There is then one contained subcontrol for each attribute of the structured type.

ColorControl

A `ColorControl` can be used for attributes of integral type. The value of such a control is interpreted as a color reference, with three components; Red, Green and Blue (RGB).

Figure 15: A ColorControl with green color as value

The color value can either be edited using a standard color picker dialog (opened by clicking on the arrow button), or the RGB value can be typed directly using the syntax RGB(<red>, <green>, <blue>).

QueryControl

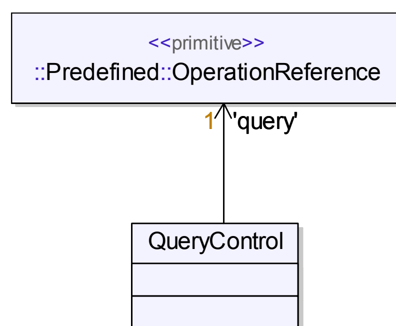


Figure 16: QueryControl definition

A `QueryControl` has a similar appearance as a DropDownMenu, but instead of having a fixed list of entities, the list is dynamically populated by executing a query (see Queries). The value of the control is a reference to the entity that is selected from the query result.

query: Operation

This attribute is a reference to the query agent to execute in order to populate the list.

NavigationButton

A `NavigationButton` can be used as the control for metafeatures of single multiplicity that are typed by a metaclass. This means that the value of the control is a reference to another entity in the model. When the button is pressed the property page for that entity is shown.

Navigation buttons can be used when there is a relationship between two entities in a model to make it easier to reach the property page for one of the entities from the property page of the other entity.

GotoOwnerButton

A `GotoOwnerButton` is a special kind of NavigationButton which always performs navigation to the composition owner of the edited element.

ValueControl

Some control classes inherit the `ValueControl` class, representing controls that can display and edit a value.

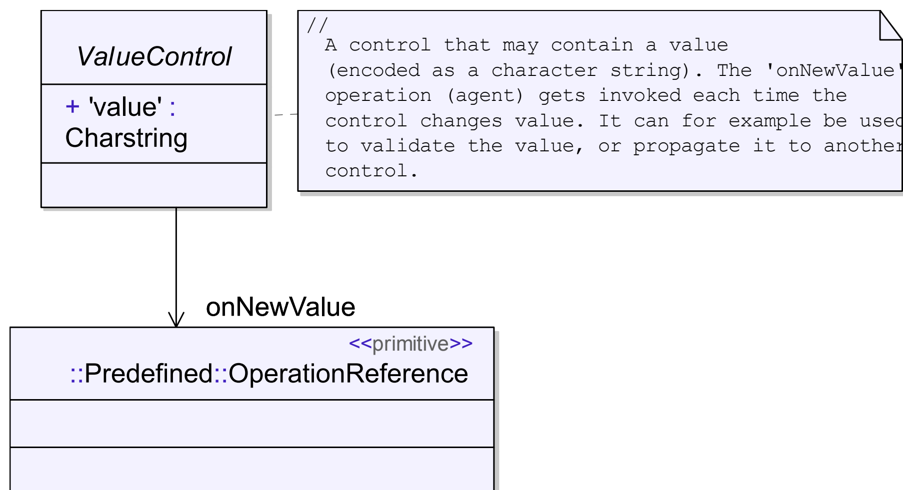


Figure 17: ValueControl classes

value: Charstring

This attribute is used internally by the Properties Editor to hold a representation of the control’s value. However, it can also be explicitly specified to force a control to always show a particular value.

onNewValue: Operation

This attribute specifies an agent operation which will be invoked each time the control gets a new value. It can be used as a means for validating the entered value of a control, or to propagate a value to another control. The agent will be called just before the new value is set, with the edited element as its model context, and with the following parameters:

- `attribute : Entity`
  The edited attribute (stereotype or Metaclass attribute)
- `newValue : Entity`
  The new value to be set to the control.
- `stereotypeInstance : Entity`
  If the edited attribute is a stereotype attribute, this parameter is the stereotype instance that is about to be modified. Otherwise this parameter is not passed.

PositionedControl

The `PositionedControl` class represents those properties of a control that are related to its graphical position and size. By default the Properties Editor applies a simple kind of autolayout for determining where a control shall be positioned. Attributes will be positioned left aligned and top-down, and autolayout position for a control is calculated relative to the preceding control. The attributes of the PositionedControl class makes it possible to customize this layout to some extent.

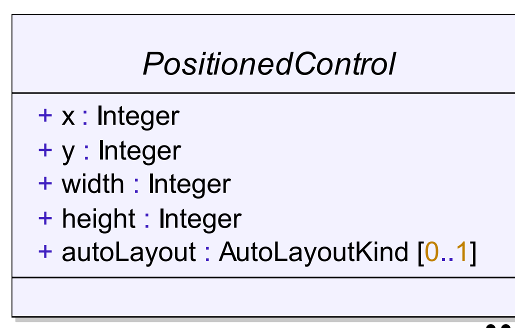


Figure 18: PositionedControl.

x: Integer

Specify a value for this attribute to override the default horizontal position of the control.

y: Integer

Specify a value for this attribute to override the default vertical position of the control.

Note

To override the default placement of a control you must give a value both for the x and y attributes. The position you give to a control will affect a succeeding control that uses default layout.

width: Integer

Specify a value for this attribute to override the default width of the control.

height: Integer

Specify a value for this attribute to override the default height of the control.

autoLayout: AutoLayoutKind

This attribute specifies an option to the autolayout algorithm that decides how a control is affected by resizing the Properties Editor window. The following values can be used for this attribute:

- `GrowRight`
  The size of the control grows at its right side when the size of the Properties Editor window is increased. This is the default behaviour for most controls.
- `GrowBottom`
  The size of the control grows at its bottom side when the size of the Properties Editor window is increased.
- `GrowRightAndBottom`
  This size of the control grows at both its right and bottom sides when the size of the Properties Editor window is increased.

Create Presentation

The **Create Presentation** dialog provides a natural entry point to models, as an alternative to using the **New** command to Create diagrams from the Model View. This dialog is opened from the shortcut menu of any element.

Create Presentation dialog

The **Create Presentation** dialog has a title and a set of tabs. The dialog title shows the type and name of the current entity that the Create Presentation is focused on. The tabs each contain a tab description and a list of alternatives.

A click on an alternative in a tab closes the dialog, creates model elements, symbols, lines or diagrams as needed, and navigates to them.

New Symbol

With the **New Symbol** tab, you can create a symbol for the current entity either in an existing diagram or create a new diagram containing a presentation element for the entity.

New Diagram

The **New Diagram** tab follows the Model View creation rules. From this tab you may create a diagram below the current entity. This is equivalent to using the Model View shortcut menu **New** for creating diagrams.

Location column

The location of the alternative in the model.

Diagram Name column

Name of the alternative.

Item Type column, Diagram Type column

The type of the described entity. For instance: `Class`, `ClassSymbol` or `ClassDiagram`.

See also

“Model navigation/creation” on page 78

“Add symbols” on page 119 in Chapter 3, *Working with Diagrams*

Model Navigator

The Model Navigator is a tab, named Navigate, in the Output window that allows you to browse and navigate through various aspects of any entity in a model.

The key purpose of the Model Navigator is to provide a natural and powerful tool for navigation in the model. While the Model View displays the model based on a hierarchical scope view, the Model Navigator has a number of different views allowing you to cross-examine the model based on the model's internal relations.

The model navigator also allows you to:

- Select and display diagrams.
- Navigate to a symbol or line representing the current entity.
- Take navigation shortcuts to entities related to the current entity.

If you select **Model Navigator** from the Model View shortcut menu or an editor shortcut menu, then the Model Navigator will be opened.

Model navigation/creation

When you double-click an element in a diagram or in the Model view a model navigation/creation will be activated

- If there is any presentation element representing the double-clicked element the diagram with this element will become active and the Navigate tab will be activated.
- If there is no presentation element representing the double-clicked element the Create Presentation dialog will be opened.

Model navigator tabs

The Model Navigator tab itself has a set of tabs. These tabs each contain a tab description and a list of alternatives. The set of tabs depends on the current entity. The start tab in the window will be selected by using the following criteria:

- Latest used tab
- Highest priority of applicable tabs

Column widths may be resized by dragging the vertical bar to the right of each column header.

Sorting

Alternatives in tab lists are initially sorted in ascending order based on the name column. For tabs without a name column, the type or index number column is used instead.

Manual sorting is done by clicking on a column header. Repeated clicks will reverse the sort order.

Tab categories

The Model Navigator tabs can be categorized into two main groups:

- Tabs that show the alternative in the Model View or in a diagram. In this group you find the **Presentation tabs** and the Links tab.
- Tabs that refocuses (on CTRL + click) the Model Navigator on a new model element. This type of tabs are called **Entity tabs**.

Below, you will find more information on the different tab groups.

- **Presentation tabs**

  A click on an alternative in a presentation tab navigates to a symbol or line in a diagram (the Symbols tab), or to a diagram itself (the Diagrams tab).
- **The Links tab**

  A click on an alternative in the Links tab closes the dialog and navigates to the other link endpoint.
- **Entity tabs**

  On CTRL + click on an alternative in an entity tab the Model Navigator refocuses on the clicked alternative, which becomes the new current entity. The new current entity is selected in the Model View, if possible. In this category, you find the **Package, Features, Bookmarks, Definitions, Shortcuts, References, Model Index** and **Recent** tabs.

The Model Navigator tabs are ordered according to the table below.

Priority	Tab name	Category
1	Symbols	Presentation
2	Diagrams	Presentation
3	Links	Link
4	Package	Entity
5	Features	Entity
6	Bookmarks	Entity
7	Definitions	Entity
8	Shortcuts	Entity
9	References	Entity
10	Model Index	Entity
11	Recent	Entity

Navigation

Double-clicking on an alternative will show the alternative in both the Model View and a diagram (if this is possible to do).

Holding down CTRL while you click or double-click will refocus the model navigator on the clicked alternative, as well as show the alternative in both the Model View and a diagram (if this is possible to do).

Holding down SHIFT while you click or double-click will show the alternative only in the Model View, not in a diagram.

The tab and alternative shortcut menus in the Model Navigator contain a list of recent Model Navigator entities. This list allows you to refocus the Model Navigator on an entity that recently has been the current Model Navigator entity.

Presentation tabs

Symbols

The Symbols tab shows symbols and lines related to the current entity.

Diagrams

The Diagrams tab shows diagrams closely related to the current entity.

Links

The Links tab contains a list of incoming and outgoing hyperlinks for the current entity. Click on a link to navigate to the other link endpoint associated with the link.

Entity tabs

Package

The Package tab shows a complete list of definitions visible in the package containing the current entity.

Features

If the current entity is a class or something similar (more precisely: If the current entity is a Classifier or is contained in a Classifier), then the Features tab lists the definitions in that class, together with any inherited definitions.

Definitions

The Definitions tab shows a complete list of local and inherited definitions in the scope of the current entity.

References

The References tab will list Model references to the current Definition, for quick navigation to the places where the Definition is used. This information is similar to the Model View shortcut menu choice List references.

Shortcuts

The Shortcuts tab provides quick navigation through some commonly used relationships of a model. The most common shortcuts are described in the text about the Shortcut column.

Bookmarks

The Bookmarks tab provides a method for setting and navigating through bookmarks, to select places in the model that you anticipate re-visiting. The contents of this tab will only be persistent over the current tool session. Adding and removing items from the list is done by clicking on the **Add/Remove** and **Remove all bookmarks** rows in the list.

From the shortcut menu for any model element in the Model View you can choose **Bookmark** to add the selected element to this list.

Model Index

The Model Index tab contains an alphabetical list of all definitions in the model with the exception of unnamed parameters (return parameters). See also description of the Find dialog.

Recent

The Recent tab keeps track of entities that the Model Navigator has been focused on, allowing you to refocus the Model Navigator on any of your recently visited entities. You can as an alternative to this tab use the shortcut menu, which contains the 5 most recent Model Navigator entities.

Columns

Below is a list of the columns appearing in the Model Navigator and a short description of the listed information.

Index column

This column can be found in the Recent tab and in the Bookmarks tab. It contains numbers indicating the order that entities were visited in. A lower number means a more recently visited entity.

Links column

The number of incoming and outgoing links to and from the current entity.

Location column

The location of the alternative in the model.

Name column, Diagram Name column

Name of the alternative.

Page column

The diagram page number. This column can be found in the Diagrams tab.

Role column

The role the current entity plays in the listed reference. This column can be found in the References tab.

Shortcut column

This column contains a list of shortcuts from the current entity to various related entities. This column can be found in the Shortcuts tab. Here are a couple of examples on shortcuts that may appear in the Shortcut column:

- The **Scope** shortcut: Refocus the Model Navigator on the scope entity that contains the current entity.
- The **Container** shortcut: Refocus on the entity that owns the current entity.
- The **Model Root** shortcut: Refocus on the model root for the current entity. This shortcut is especially useful when having a workspace with more than one model.
- The **Predefined Package** shortcut: Refocus on the internal library of predefined types.

Type column, Item Type column, Diagram Type column

The type of the described entity. For instance: `Class`, `ClassSymbol` or `ClassDiagram`.

Views column

Number of symbols and lines representing the current definition.

Generate Diagram

DOORS Analyst supports automatic generation of diagrams in order to visualize existing model elements. There are a number of built-in diagram generators available, for generating commonly useful diagrams, such as inheritance diagrams, composition diagrams, dependency diagrams etc. It is also possible to add additional custom diagram generators to support specific visualization needs.

To generate a diagram, follow these steps:

1. Select an element in the Model View. The selected element will be the context of the generated diagram. For example, if you want to visualize super- and sub-classes of a certain class, then you should select that class.
2. In the context menu select **Generate Diagram** and choose which diagram generator to use in the sub menu. For example, to generate an inheritance diagram, select “Generate inheritance view”.

The generated diagram is typically placed under the selected context element, but some diagram generators may place it elsewhere in the model, for example as a top-level diagram, or in a separate package. Afterwards you may move it to where you want it to be.

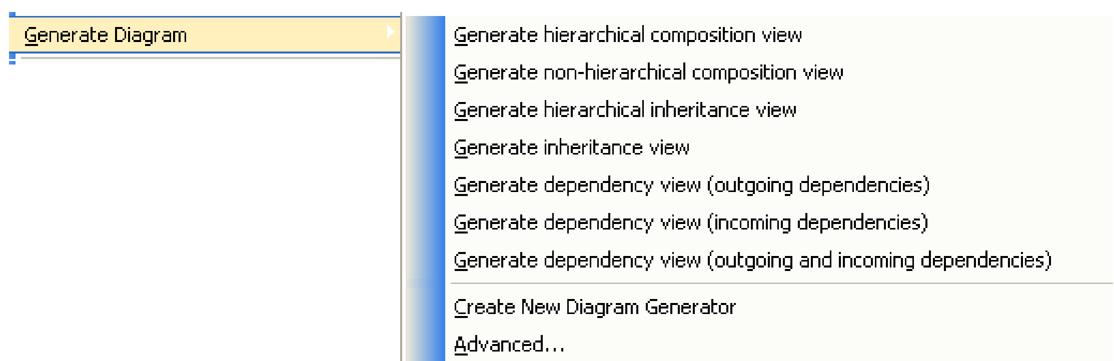


Figure 19: The Generate Diagram context menu

Diagram Generation Parameters

Diagram generators may take actual parameters to control how the diagram shall be generated. For example, when generating an inheritance view for a class, a parameter controls whether the diagram only shall show immediate super- and sub-classes, or if all recursive super- and sub-classes shall be shown.

When a diagram generator is run from the **Generate Diagram** context menu, default values are used for these parameters. To change the actual parameters use the command **Edit Diagram Generation Parameters** available in the context menu of the generated diagram. You can also edit them by opening the Properties Editor on the generated diagram, and selecting the <<generated>> stereotype as filter. Parameters can then either be edited textually in the Parameters field, or you may press the **Edit Parameters** button.

Regenerate Diagram

A generated diagram can be regenerated based on new information in the model. For example, you may want to regenerate an inheritance diagram when new super- or sub-classes have been added. You may also want to regenerate a diagram if you have modified the Diagram Generation Parameters.

To regenerate a generated diagram choose the **Regenerate** command available in the diagram context menu. It is also possible to regenerate all generated diagrams in the model by selecting the **Regenerate All Diagrams** command in the Tools menu. Only generated diagrams are affected by these commands.

Important!

When a diagram is regenerated everything it contains will be deleted and regenerated. This means that if you have made manual modifications to the diagram, such as changing layout, colors etc., these changes will be lost.

Convert a generated diagram into an ordinary diagram

To avoid accidentally regenerating a generated diagram that has been modified, it is recommended to convert the diagram into an ordinary non-generated diagram if you want to maintain it manually. To do so follow these steps:

1. Select the generated diagram in Model View.
2. Select **Stereotypes...** in the context menu.
3. Uncheck the ‘generated’ checkbox and press **OK**.

After this it is no longer possible to regenerate the diagram.

Using Diagram Generators in Existing Diagrams

A diagram generator doesn’t have to always generate a new diagram. It is also possible to use a diagram generator in order to add information to an existing diagram. The steps to do so are:

1. Drag the context element from the Model View into a diagram using the right mouse button.
2. Drop the element on the diagram and select **Visualize in Diagram** in the context menu that appears.
3. In the sub menu select which diagram generator to use.

The symbols and lines generated by the diagram generator will be inserted in the diagram where the entity was dropped.

Advanced Diagram Generators

In addition to the diagram generators you will find in the **Generate Diagram** context menu there are also a few more advanced diagram generators. To use these diagram generators select the **Advanced...** command in the **Generate Diagram** context menu. This will open the Advanced Generate Diagram dialog:

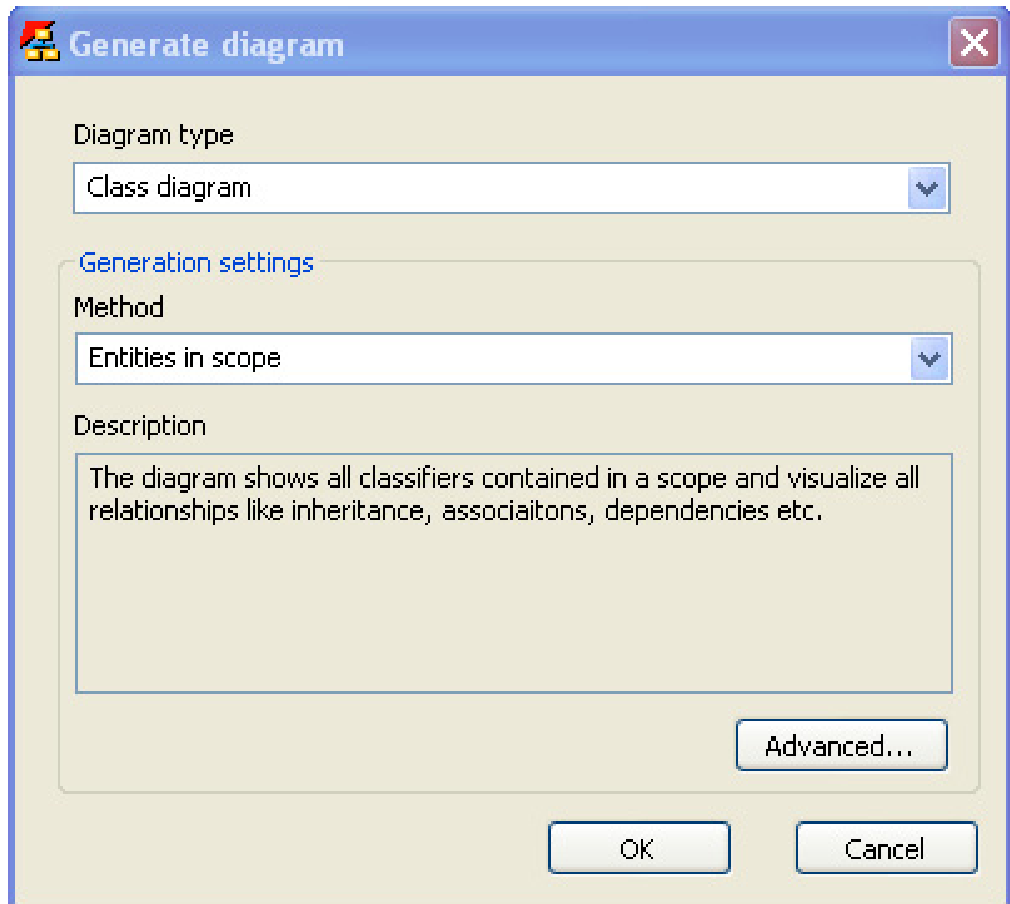


Figure 20: The advanced diagram generator dialog

This dialog only provide a limited set of diagram generators, but each of these diagram generators have several customizable layout options.

Diagram type

First thing to do in the Advanced Generate Diagram dialog is to select the wanted diagram type. Only the diagram types where there is a method to generate a diagram is displayed.

Generation settings

Secondly, the **Method** of generation can be selected. Only methods of generation that can be applied for the above selected diagram type is available.

A description of the selected generation method is displayed below the list of available generation methods.

Settings for the selected generation method are available by pressing the **Advanced** button. These settings will be associated with the generated diagram and can be edited after generation in the Properties Editor.

Customization

It is possible to create your own diagram generators in order to generate custom diagrams. See Adding Diagram Generators for more information on this topic.

It is also possible to invoke diagram generators programmatically. This can be useful for example when implementing add-ins. See Invoking Diagram Generators Programmatically for more details.

Queries

This section describes how to perform a query on a UML model in order to find entities that fulfill certain conditions.

Queries are useful for finding entities in the model that cannot be found by using the more basic search facilities of the Find dialog. Using a query is an alternative to using one of the standard APIs for finding the wanted information. Since a query may contain calls to many of the available API functions (COM, C++, Tcl), the expressive power of a query is equivalent to using the APIs.

Concepts

A **query** is an operation that yields a collection of entities from the model.

A **predicate** is an operation that yields a boolean true or false.

Both a query and a predicate may take any number of input arguments. One input argument that always is implicitly present is the **model context**. This is an entity on which the query or predicate is invoked.

In order to be able to define query and predicate operations in a UML model, there is a built-in library called `TTDQuery`, which defines the stereotypes in Figure 21 on page 89. See also “The Query Dialog” on page 93.

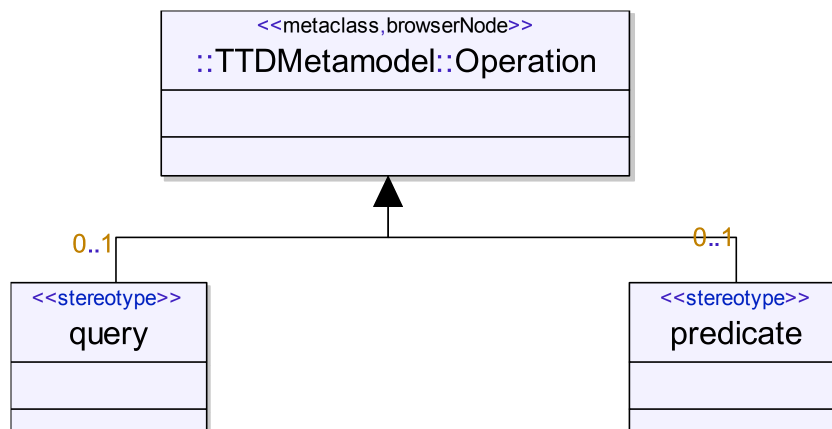


Figure 21: `TTDQuery` *library with stereotypes*

In addition to these stereotypes, the `TTDQuery` profile also contains a number of built-in queries and predicates, that are ready to use.

Calls to queries and predicates can be put together in a Query expression. This is an expression that can be interpreted by DOORS Analyst, and just like when invoking a query operation, the result of the interpretation is a collection of entities from the model. A query expression may use boolean operators and literals, as well as a small subset of the collection operators that are found in OCL, in order to modify the result obtained by calling queries and predicates.

Note

Many operations of the public APIs work as either queries or predicates. These operations are also available for use in query expressions. The UML definition of these API operations can be found in the library called `u2`.

Query expression

A query expression is written in textual UML expression syntax. The type of a query expression must be a collection of entities. This means that when a query expression is interpreted the result should be a collection of entities.

All sub-expressions that are contained in a query expression must be of either boolean type, or the type should be a collection of entities. For expressions of boolean type, the usual boolean operators can be used. The following boolean operators and literals are supported within a query expression:

```
and (&&)
or (||)
not (!)
true
false
```

Parentheses can also be used.

For expressions whose type is a collection of entities, a number of predefined Collection Operators may be used.

Collection Operators

Certain predefined operators may be used on the collection of entities that result from executing an expression within a query expression. The names and semantics of these operations come from OCL (Object Constraint Language). In fact, a query expression is a legal OCL expression, except that periods (.) are used instead of the arrow notation (->) when invoking a predefined collection operation. However, only a subset of OCL is supported. This subset allows powerful queries to be performed.

select

Syntax:

```
select(<boolean expr>)
```

Type: collection of entities

`select` projects one collection of entities into another collection of entities. The resulting collection will contain those entities in the input collection for which the boolean expression evaluates to `true`. Thus, `select` can be used to filter a collection through a predicate.

exists

Syntax:

```
exists(<boolean expr>)
```

Type: boolean

`Exists` is a boolean operator that returns `true` if there exists at least one entity in the input collection for which the boolean expression evaluates to `true`, otherwise it returns `false`.

isEmpty

Syntax:

```
isEmpty()
```

Type: boolean

This operator returns `true` if the input collection is empty. Otherwise it returns `false`.

Examples

Here are some examples of query expressions that uses some of the available Built-in Queries and Predicates, combined with predefined boolean operators and collection operators.

Example 5

Find all active classes defined in a package.

[model context = the package]

```
GetAllEntities().select(IsKindOf("Class") and
HasPropertyWithValue("isActive", "true"))
```

Example 6

Find all attributes in the model that are directly owned by a class.

[model context = the model, i.e. the Session]

```
GetAllEntities().select(IsKindOf("Attribute") &&
GetOwner().exists(IsKindOf("Class")))
```

Example 7

Find all «access» dependencies in the model.

[model context = the model, i.e. the Session]

```
GetAllEntities().select(not
GetTaggedValue("access(..)").isEmpty())
```

This query will obtain the wanted result, but is quite inefficient since it will check for an applied «access» stereotype on each entity in the model. Performance will be greatly improved just by adding a check that the entity must be a dependency. For all entities that are not dependencies, there is no need to invoke the `GetTaggedValue` query.

```
GetAllEntities().select(IsKindOf("Dependency") and not
GetTaggedValue("access(..)").isEmpty())
```

You can rewrite the expression by using the `HasAppliedStereotype` predicate, which is the recommended way to check if a stereotype is applied on an element.

```
GetAllEntities().select(IsKindOf("Dependency") and
HasAppliedStereotype("access"))
```

Finally, it should be mentioned that the most efficient (and also the shortest) query expression for finding the «access» dependencies makes use of the built-in `GetStereotypedEntities` query:

[model context = the «access» stereotype, found in the `TTDPredefinedStereotypes` library]

```
GetStereotypedEntities()
```

As seen in this example there can be alternative query expressions that can be used to obtain the same result. There can be a great difference in execution performance between different semantically equivalent queries, so it is can be worthwhile to consider different alternatives before writing a query expression.

The Query Dialog

The Query dialog allows you to construct a Query expression to execute. The dialog is opened by selecting an entity in the Model View or the diagrams, and selecting the menu item Edit -> Query. The selected entity will be the model context of the query expression.

Note

The model context of the query expression may be a presentation element (e.g. a symbol or a line in a diagram). Thus, if you open the query dialog from a selected entity in a diagram, the selected presentation element will be the model context. Use the context menu "Show in Model View" to find the corresponding element in the Model View, in case you want to run the query on the model element instead.

The Query dialog lists all available Queries and predicates that can be found in the current model. This list consists of all "built-in" queries and predicates that are supplied in the predefined `TTDQuery` and `u2` libraries, together with all queries and predicates that are defined elsewhere (for example user-defined queries and predicates).

The query expression is executed by pressing the **Execute** button. By default the result will be output in the "Search Result" tab, but this can be changed by typing another tab name in the drop down control.

You may construct the query expression either by writing the expression directly in the edit control, or you can double-click on entries in the list of available queries and predicates. If the selected operation (query or predicate) do not have any formal input parameters, a call to the operation will be added directly at the position of the cursor in the query expression text. If, however, the operation has at least one formal input parameter, a dialog (see Figure 22 on page 94) will pop-up which allows you to provide the corresponding actual parameter for the operation call.

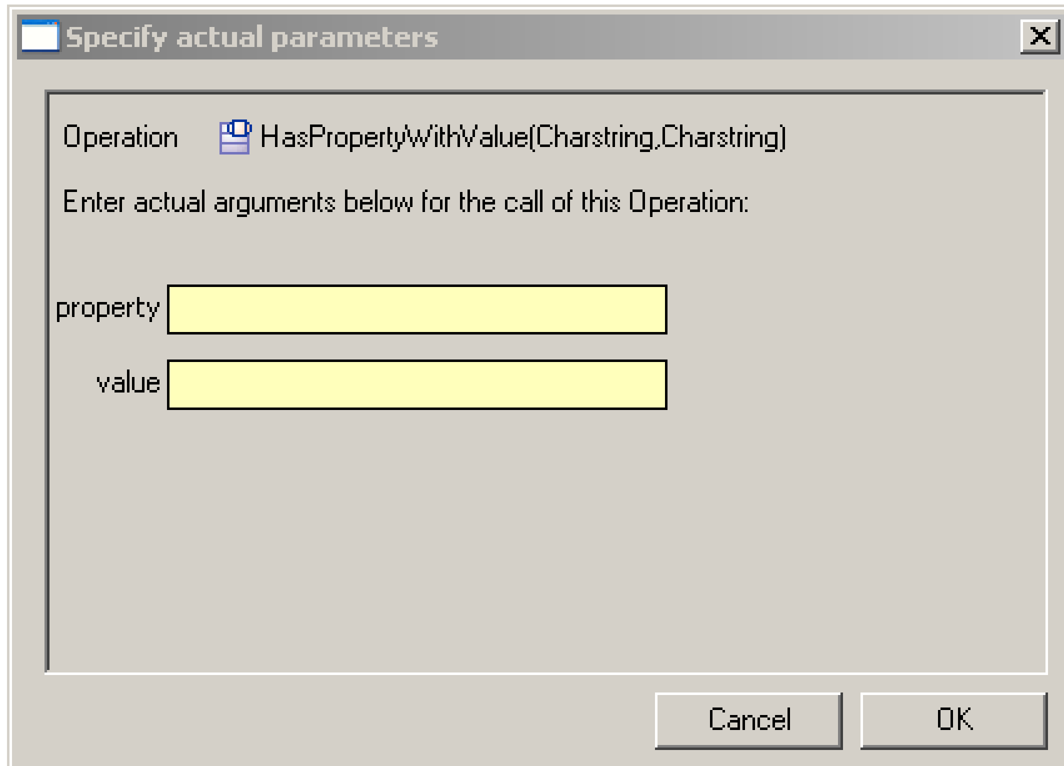


Figure 22 Specify actual parameters

This dialog is in fact a Properties Editor (the parameters are seen as properties of the operation) and edited values follow the same Color Codes as the Properties editor. Other features of the Properties Editor, such as obtaining "What's This?" help on the meaning of the parameters are also available.

Saving a query expression as a new query

The Query dialog has a Save button that allows you to save a query expression as a new query in the model. Use this possibility if you have constructed a query expression that you want to save for the future. You will be prompted to specify a name and description of the new query, as well as a location in the model where it shall be stored. It can be a good idea to put all queries in a common place, for example in a profile package stored in a separate .u2 file. Thereby you can include and use your saved queries in multiple projects.

When you have saved a query expression as a new query, it immediately becomes available in the list of queries and predicates that are ready to use in new query expressions.

Built-in Queries and Predicates

A number of built-in queries and predicates are available for use in query expressions. These are defined and documented in the profile libraries `TTDQuery` and `u2`.

In addition to these, it is possible to add user-defined queries and predicates as described in User-defined Queries and Predicates.

User-defined Queries and Predicates

It is possible to define additional queries and predicates than those that are supplied as “built-in”. This is done by defining an agent which has the <<query>> or <<predicate>> stereotype applied. The implementation of such Agents must fulfill the signature of a query or predicate. Thus a query agent must return a list of entities, and a predicate agent must return a boolean value. This mandatory output parameter is passed as the first parameter to the agent. In addition the agent may take any number of input parameters. These parameters may have any type supported by Agents.

Executing a Query Expression from the APIs

It is possible to execute a Query expression programmatically from all the public APIs, using the agent in Figure 23 on page 95.

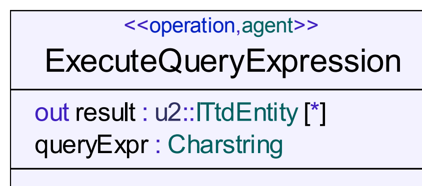


Figure 23 Agent for query execution

This agent is (like any other agent) invoked using the InvokeAgent operation.

Example 8: Executing a query expression from the Tcl API

This small example shows how to execute the query expression “`GetAllEntities()`” from a Tcl script. The script just prints the Tcl ids of the resulting entities.

```
set s [std::GetSelection]
set a [u2::FindByGuid $U2
```

```
"@TTDQuery@ExecuteQueryExpression"]
set p [lappend p {} "GetAllEntities()"]
u2::InvokeAgent $U2 $a $s p
output [lindex $p 0]
```

Drag and Drop

This section describes how drag and drop can be used to work with the model.

A drag and drop operation can be done with three different variations of drag sources and drop targets:

- Within the model view.
- From model view to a diagram.
- Within and between diagrams.

A drag and drop operation can be done either with the left or the right mouse button. If a drag and drop operation is done with the right mouse button, a context menu is opened listing the possible operations to perform as a result of the drag from the source element to the target element. The context menu will always have a highlighted alternative. This is the operation that will be performed when a drag and drop operation is done using the left mouse button. There can also be modifier key within parenthesis next to the operation. If so, this operation can be performed by holding down this modifier key will doing a drag and drop using the left mouse button.

Next follows the different operations available using drag and drop.

Within the Model View

Move

Moves an element within the model view.

This is the default operation for drag and drop within the model view and will be performed if drag and drop is done using the left mouse button.

Copy

Copies an element within the model view.

This operation can be performed by doing a drag and drop operation using the left mouse button while holding down the CTRL button.

Link

Creates a link between the drag source element and the drop target element. The currently active link type will be used.

Copy with Traceability

Copies an element (including subelements) in the model view and creates <<trace>> dependencies from the copy to the original.

The operation is performed by doing a drag and drop operation using the right mouse button and choosing the “Copy with Traceability” command in the pop-up menu.

The dependencies will be created for all definitions, like e.g. packages, classes, attributes and operations.

See also

“Working with links” on page 2409

From Model View to a Diagram

Create Presentation

Creates a symbol representing the drag source element in the context of the drop target element.

This is the default operation for drag and drop from the model view to a diagram and will be performed if drag and drop is done using the left mouse button.

Create Presentation (include lines)

Does the same thing as Create Presentation with the addition that lines representing the drag source element connections to other elements in the drop target diagram will be created.

Visualize in Diagram

This is a sub-menu containing the possible diagram generation methods that are available for the drag source element and the drop target diagram. The drag source elements will be visualized in the diagram without affecting any already existing elements in the diagram.

See also

“Generate Diagram” on page 84

Within and between Diagrams

Drag and drop within and between diagrams has the same operations as within the model view.

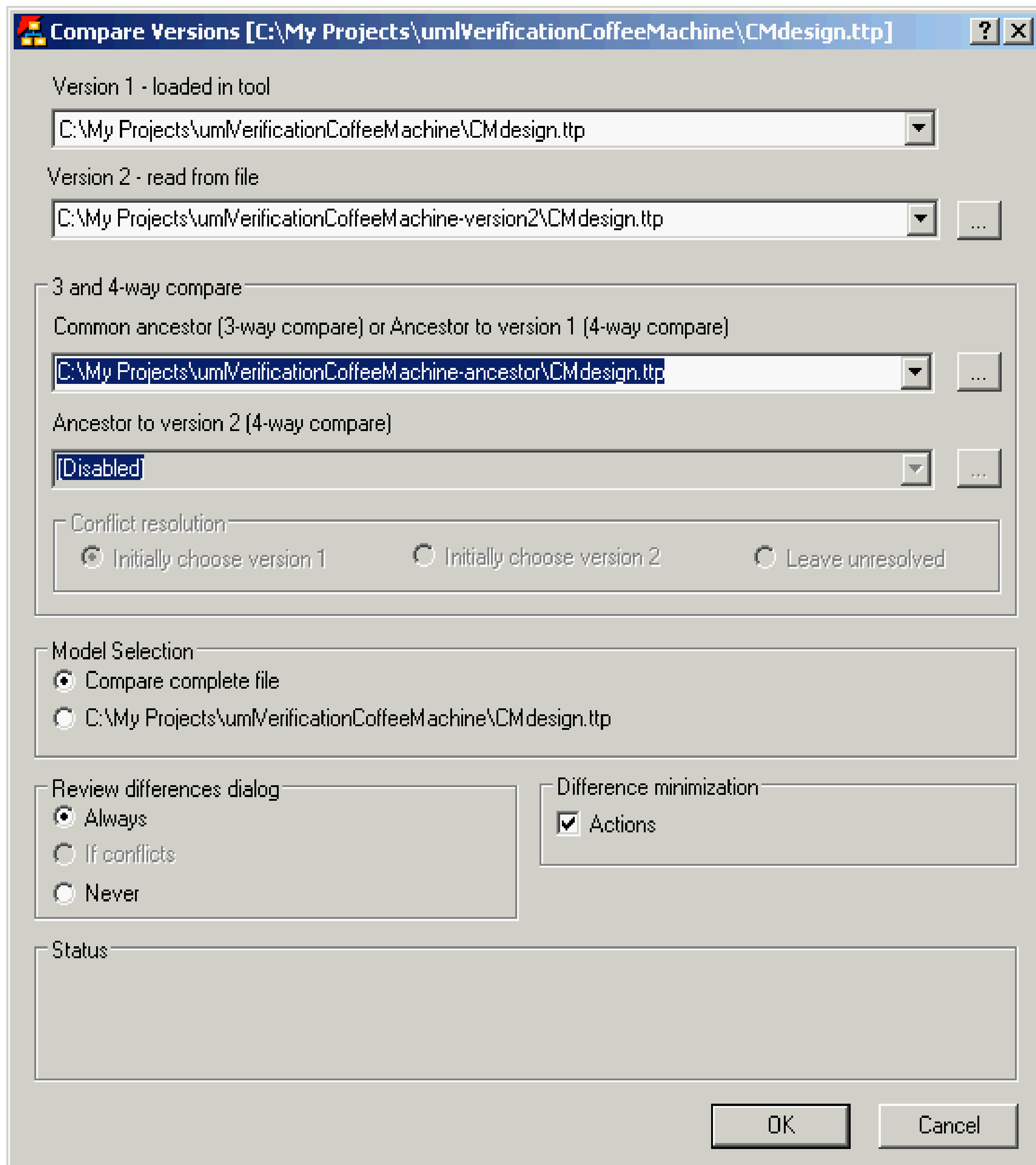


Figure 24: Compare Versions dialog

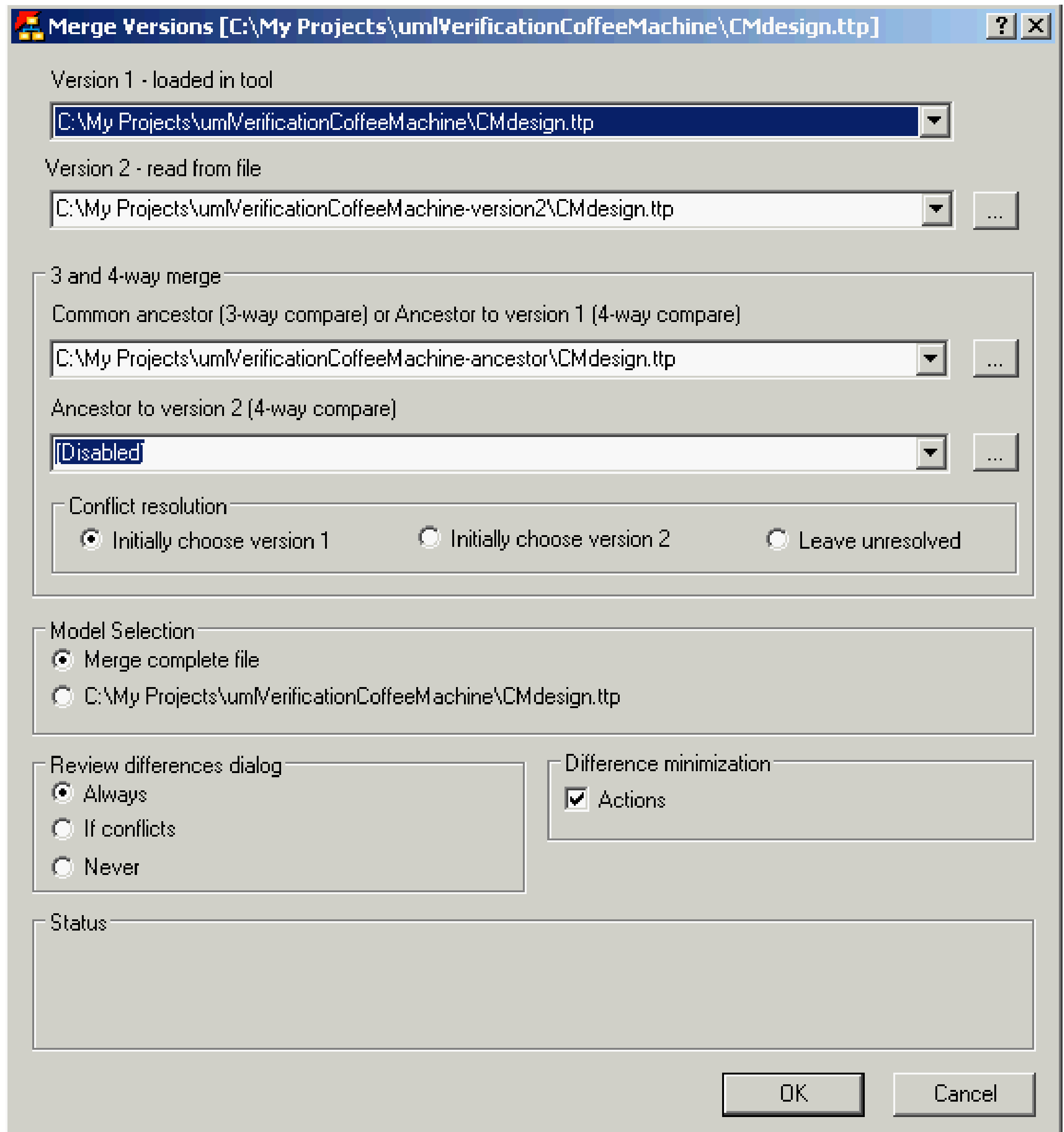


Figure 25: Merge Versions dialog

ExitOnSuccess	Description
`true`	close Tau after successful operation
`false`	do not close Tau after operation

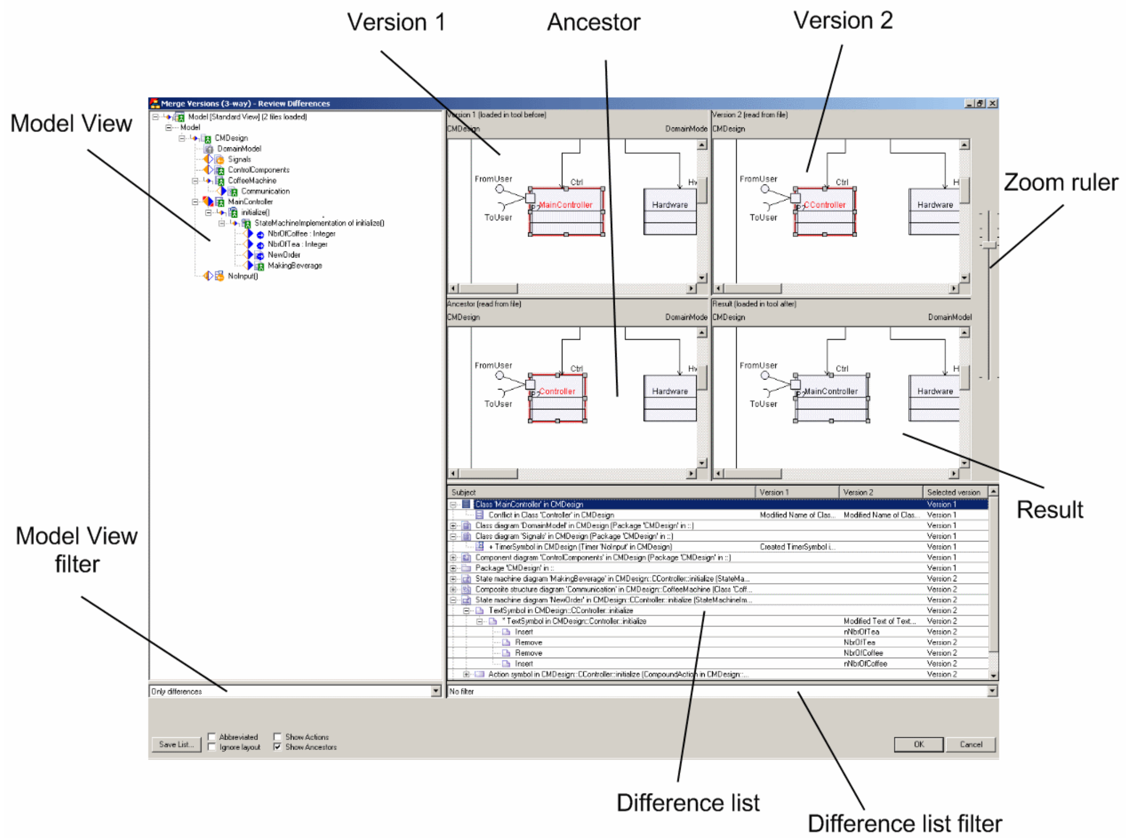


Figure 26: Review Difference dialog

External text compare/External text merge

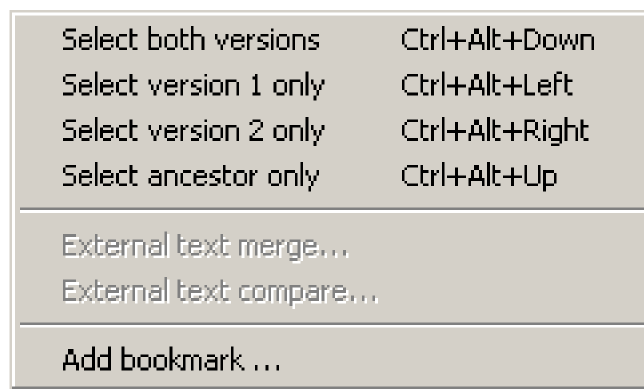


Figure 27: Context Menu

An external textual compare and merge tools can be used for comparing and/or merging comments, text symbols, task symbols and instance expressions. The “External text compare...” and “External text merge...” operations are available where applicable.

If an external textual merge is done, the result will be checked if it can be re-entered into the model. If it cannot be entered into the model, the result file from the external tool is saved and the path is reported together with an error message box.

Path and command line switches for the external text compare/merge tool are available via the Tools menu, Options dialog, under the Compare/Merge tab.

Add bookmark

A bookmark can be added on a selected entity. A comment can be added to the bookmark. The bookmarks can later be listed in the model navigator.

Subject	Version 1	Version 2	Selected version
Class diagram 'Signals' in CMDesign (Package 'CMDesign' in ::)			Both
Class 'Hardware' in CMDesign			Version 1
Class 'MainController' in CMDesign			Version 1
Package 'CMDesign' in ::			Both
--> Timer 'Heater' in CMDesign	Moved (from) Timer 'H...		Version 1
+ Timer 'NoInput' in CMDesign	Created Timer 'NoInpu...		Version 1
- Signal 'InternalSignal' in CMDesign		Deleted Signal 'Intern...	Version 2
Attribute 'nNbrOfTea' in CMDesign::CController::initialize			Version 2
Attribute 'nNbrOfCoffee' in CMDesign::CController::initialize			Version 2
Component diagram 'ControlComponents' in CMDesign (Package 'CMDesign' in ::)			Version 1
Class diagram 'DomainModel' in CMDesign (Package 'CMDesign' in ::)			Version 1
State machine diagram 'NewOrder' in CMDesign::CController::initialize (StateMachineImple...			Version 2
Action symbol in CMDesign::CController::initialize (CompoundAction in CMDesign::CCo...			Version 2
* Action symbol in CMDesign::Controller::initialize (CompoundAction in CMDesign::...		Modified Text of Actio...	Version 2
TextSymbol in CMDesign::CController::initialize			Version 2
* TextSymbol in CMDesign::Controller::initialize		Modified Text of Text...	Version 2

Figure 28: Semantic and presentation model differences grouping

Subject	Version 1	Version 2	Selected version
Class diagram 'Signals' in CMDesign (Package 'CMDesign' in ::)			Both
+ TimerSymbol in CMDesign (Timer 'NoInput' in CMDesign)	Created TimerSymbol i...		Version 1
- SignalSymbol in CMDesign (Signal 'InternalSignal' in CMDesign)		Deleted SignalSymbol ...	Version 2
Class 'Hardware' in CMDesign			Version 1
Class 'MainController' in CMDesign			Version 1
Package 'CMDesign' in ::			Both
--> Timer 'Heater' in CMDesign	Moved (from) Timer 'H...		Version 1
+ Timer 'NoInput' in CMDesign	Created Timer 'NoInpu...		Version 1
- Signal 'InternalSignal' in CMDesign		Deleted Signal 'Intern...	Version 2
Attribute 'nNbrOfTea' in CMDesign::CController::initialize			Version 2
Attribute 'nNbrOfCoffee' in CMDesign::CController::initialize			Version 2
Component diagram 'ControlComponents' in CMDesign (Package 'CMDesign' in ::)			Version 1
Class diagram 'DomainModel' in CMDesign (Package 'CMDesign' in ::)			Version 1
State machine diagram 'NewOrder' in CMDesign::CController::initialize (StateMachineIm...			Version 2
Action symbol in CMDesign::CController::initialize (CompoundAction in CMDesign::...			Version 2
* Action symbol in CMDesign::Controller::initialize (CompoundAction in CMDesi...		Modified Text of Actio...	Version 2
TextSymbol in CMDesign::CController::initialize			Version 2

Figure 29: Grouping of "created entity" and "deleted entity" differences

Subject	Version 1	Version 2	Selected version
Class diagram 'Signals' in CMDesign (Package 'CMDesign' in ::)			Both
Class 'Hardware' in CMDesign			Version 1
<-- Timer 'Heater' in CMDesign::Hardware	Moved (to) Timer 'Hea...		Version 1
Class 'MainController' in CMDesign			Version 1
Package 'CMDesign' in ::			Both
--> Timer 'Heater' in CMDesign	Moved (from) Timer 'H...		Version 1
+ Timer 'NoInput' in CMDesign	Created Timer 'NoInpu...		Version 1
- Signal 'InternalSignal' in CMDesign		Deleted Signal 'Intern...	Version 2
Attribute 'nNbrOfTea' in CMDesign::CController::initialize			Version 2
Attribute 'nNbrOfCoffee' in CMDesign::CController::initialize			Version 2
Component diagram 'ControlComponents' in CMDesign (Package 'CMDesign' in ::)			Version 1
Class diagram 'DomainModel' in CMDesign (Package 'CMDesign' in ::)			Version 1
State machine diagram 'NewOrder' in CMDesign::CController::initialize (StateMachineImple...			Version 2
Action symbol in CMDesign::CController::initialize (CompoundAction in CMDesign::CCo...			Version 2
* Action symbol in CMDesign::Controller::initialize (CompoundAction in CMDesign::...		Modified Text of Actio...	Version 2
TextSymbol in CMDesign::CController::initialize			Version 2
* TextSymbol in CMDesign::Controller::initialize		Modified Text of Text...	Version 2

Figure 30: Grouping of "moved entity" differences

Subject	Version 1	Version 2	Selected version
Class diagram 'Signals' in CMDesign (Package 'CMDesign' in ::)			Both
Class 'Hardware' in CMDesign			Version 1
Class 'MainController' in CMDesign			Version 1
Conflict in Class 'Controller' in CMDesign	Modified Name of Clas...	Modified Name of Clas...	Version 1
Package 'CMDesign' in ::			Both
Attribute 'nNbrOfTea' in CMDesign::CController::initialize			Version 2
* Attribute 'NbrOfTea' in CMDesign::Controller::initialize		Modified Name of Attri...	Version 2
Attribute 'nNbrOfCoffee' in CMDesign::CController::initialize			Version 2
* Attribute 'NbrOfCoffee' in CMDesign::Controller::initialize		Modified Name of Attri...	Version 2
Component diagram 'ControlComponents' in CMDesign (Package 'CMDesign' in ::)			Version 1
Class diagram 'DomainModel' in CMDesign (Package 'CMDesign' in ::)			Version 1
State machine diagram 'NewOrder' in CMDesign::CController::initialize (StateMachineImple...			Version 2
Action symbol in CMDesign::CController::initialize (CompoundAction in CMDesign::CCo...			Version 2
* Action symbol in CMDesign::Controller::initialize (CompoundAction in CMDesign::...		Modified Text of Actio...	Version 2
TextSymbol in CMDesign::CController::initialize			Version 2
* TextSymbol in CMDesign::Controller::initialize		Modified Text of Text...	Version 2

Figure 31: Grouping of "modified attributes" differences

This composite node contains conflicting differences which are owned by different representative elements. For example, if entity has been moved in Version 1 and in Version 2 and the new owners of that entity are different in Version 1 and Version 2, then the Composite Conflict Group will be created. This group can contain Difference Nodes only.

Composite Group
Composite Conflict Group
Conflict Node
Consolidated Node
Difference Node

Subject	Version 1	Version 2	Selected version
Class diagram 'Signals' in CMDesign (Package 'CMDesign' in ::)			Both
+ TimerSymbol in CMDesign (Timer 'NoInput' in CMDesign)	Created TimerSymbol i...		Version 1
- SignalSymbol in CMDesign (Signal 'InternalSignal' in CMDesign)		Deleted SignalSymbol ...	Version 2
Class 'Hardware' in CMDesign			Version 1
Conflict in Timer 'Heater' in CMDesign::Hardware			Version 1
<-- Timer 'Heater' in CMDesign::Hardware	Moved (to) Timer 'Hea...		Version 1
<-- Timer 'Heater' in CMDesign::CoffeeMachine		Moved (to) Timer 'Hea...	Ancestor
Class 'MainController' in CMDesign			Version 1
Conflict in Class 'Controller' in CMDesign	Modified Name of Clas...	Modified Name of Clas...	Version 1
Package 'CMDesign' in ::			Both
Identity in Timer 'Heater' in CMDesign	Moved (from) Timer 'H...	Moved (from) Timer 'H...	Both
+ Timer 'NoInput' in CMDesign	Created Timer 'NoInpu...		Version 1
- Signal 'InternalSignal' in CMDesign		Deleted Signal 'Intern...	Version 2
Attribute 'nNbrOfTea' in CMDesign::CController::initialize			Version 2
Attribute 'nNbrOfCoffee' in CMDesign::CController::initialize			Version 2
Class 'CoffeeMachine' in CMDesign			Version 1
Component diagram 'ControlComponents' in CMDesign (Package 'CMDesign' in ::)			Version 1
Class diagram 'DomainModel' in CMDesign (Package 'CMDesign' in ::)			Version 1
State machine diagram 'NewOrder' in CMDesign::CController::initialize (StateMachineImple...			Version 2
Action symbol in CMDesign::CController::initialize (CompoundAction in CMDesign::CCo...			Version 2
* Action symbol in CMDesign::Controller::initialize (CompoundAction in CMDesign::...		Modified Text of Actio...	Version 2
Remove		NbrOfCoffee	Version 2
Insert		nNbrOfCoffee	Version 2
Remove		NbrOfTea	Version 2
Insert		nNbrOfTea	Version 2
TextSymbol in CMDesign::CController::initialize			Version 2

Figure 32: Nodes in Difference list

Conflict Node

This node corresponds to conflicting differences which are related to the same representative element.

Consolidated Node

This node corresponds to consolidated differences which are related to the same representative element.

Difference Node

This node describes the simple change that has been made in Version 1 or in Version 2.

Composite Textual Difference Node

This group node corresponds to a group of primitive textual differences. This group can contain Textual Difference Nodes only.

Textual Difference Node

This node corresponds to a primitive textual difference. There are two operations that represent modifications in the text: **Remove** and **Insert**. **Remove** means that a part of the text has been deleted (in comparison with the ancestor version). **Insert** means that a new text has been added.

Composite Textual Difference Nodes

Textual Difference Nodes

Subject	Version 1	Version 2	Selected version
Class diagram 'Signals' in CMDesign (Package 'CMDesign' in ::)			Both
Class 'Hardware' in CMDesign			Version 1
Class 'MainController' in CMDesign			Version 1
Package 'CMDesign' in ::			Both
Attribute 'nNbrOfTea' in CMDesign::CController::initialize			Version 2
Attribute 'nNbrOfCoffee' in CMDesign::CController::initialize			Version 2
Class 'CoffeeMachine' in CMDesign			Version 1
Component diagram 'ControlComponents' in CMDesign (Package 'CMDesign' in ::)			Version 1
Class diagram 'DomainModel' in CMDesign (Package 'CMDesign' in ::)			Version 1
ClassSymbol in CMDesign (Class 'MainController' in CMDesign)			Version 1
Identity in ClassSymbol in CMDesign (Class 'Controller' in CMDesign)	Modified Text of Class...	Modified Text of Class...	Version 1
Remove	Controller	Controller	Version 1
Conflict: Insert vs. Insert	MainController	CController	Version 1
State machine diagram 'NewOrder' in CMDesign::CController::initialize (StateMachineImple...			Version 2
Action symbol in CMDesign::CController::initialize (CompoundAction in CMDesign::CCo...			Version 2
* Action symbol in CMDesign::Controller::initialize (CompoundAction in CMDesign::...		Modified Text of Actio...	Version 2
Remove		NbrOfCoffee	Version 2
Insert		nNbrOfCoffee	Version 2
Remove		NbrOfTea	Version 2
Insert		nNbrOfTea	Version 2
TextSymbol in CMDesign::CController::initialize			Version 2
* TextSymbol in CMDesign::Controller::initialize		Modified Text of Text...	Version 2
Remove		NbrOfCoffee	Version 2
Insert		nNbrOfCoffee	Version 2
Remove		NbrOfTea	Version 2
Insert		nNbrOfTea	Version 2

Figure 33: Textual difference nodes

When Textual Difference Node is selected in Differe list, the modified part of code is highlighted in the one or several windows which represent Ancestor, Version 1, Version 2 and Result models, see Figure 34 on page 107.

If the selected node corresponds to **Remove** operation, then the removed part of the text is selected in Ancestor window (and in Result window, if that operation is rejected). If the selected node corresponds to **Insert** operation, then the inserted part of the text is selected in Version 1 (and/or Version 2) window (and in the Result window, if that operation is accepted).

Figure 34: Textual difference highlighting

The colors are used in the Review differences dialog in order to simplify the understanding of changes that have been done in both versions. The **blue** color is used to mark presentation elements that correspond to model elements modified in version 1 only. The **green** color is used to mark presentation elements corresponding to model elements that have been modified in version 2 only. And presentation elements that correspond to model elements modified in both versions are marked by **red** color. The same coloring is applied to modified presentation model elements, i.e. Symbols and Lines as well as to parts of the text.

3

Working with Diagrams

When you have opened a project you are ready to edit your model.

How you should use the diagram editors in combination with the model information is highly dependent on the application that you want to create. The recommendations below are meant to help you getting familiar with the tool and later you can adapt and change the workflow to fit your needs.

Common Diagram Operations

Common diagram operations include:

- Create diagrams
- Open, save and print diagrams
- Move diagrams
- Resize diagrams
- Find
- Text parsing
- Diagram auto layout
- Organizing the view

Grid

The diagram drawing area will always have an active grid, which means that there will be a **snap to grid** which is always set to **on**. The grid spacing is set to 2 millimeters and cannot be changed. It is possible to show or hide the grid from the shortcut menu or by changing in the Options, default is hidden. All symbols, lines and text fields (except those fixed in symbols) will adhere to the grid. When a symbol is resized, it is done in grid steps. This also applies to auto resizing.

Frame

All diagrams will have a frame enclosing all symbols, except port symbols which can be placed on the frame. The upper left corner of the frame symbol will be at (x=10, y=10) millimeters, measured from the canvas top left corner. The frame is sized according to the diagram size and layout. The position can be overruled if extra space is needed for symbols placed on or outside the frame.

The frame may be resized and moved in all directions, canvas space permitting.

Heading

The diagram heading is positioned in the top left corner of the diagram. The text is calculated from the properties of the defining entity.

Diagram Name

The diagram name is positioned in the top right corner of the diagram. The text is right aligned. The information is calculated from the model information, and thus only editable through the Model view.

Create diagrams

A diagram contains a set of presentation elements representing elements from your model. Diagrams are managed through the Workspace window.

1. In the **Workspace** window, click the **Model View** tab.
2. Select or create a suitable package.
3. You are now able to choose **New** from the shortcut menu to create an appropriate diagram (or model element).

See also

"Create Presentation" on page 76 in Chapter 2, *Working with Models*

Open, save and print diagrams

Information in the diagram is saved in a file when you save the project. If you do not specify a file, the information will reside in the data file (.u2 extension). This file will normally reside together with the project (`.ttp`) and workspace (`.ttw`) file you are currently using.

- To open a diagram: double-click the diagram icon in the **Model** View.
- To print an open diagram: on the **File** menu, click **Print...**

The diagram size when creating a new diagram is derived from the printer settings. For example, if the page layout is set to landscape on the printer, diagrams will have a landscape orientation.

A diagram can be exported as an image in a range of formats including JPEG, GIF, BMP and SVG.

A diagram can be saved as an image by opening it and clicking **Save As...** on the **File** menu. A save dialog will open where a file path and wanted image format can be selected.

See also

“Select diagrams to be printed” on page 393 in Chapter 8, *Printing*

“Save” on page 417 in Chapter 11, *Dialog Help*

Move diagrams

You can move diagrams within projects and between projects in the same workspace.

- Click the icon corresponding to the diagram in the Model View and drag the diagram to its desired location.

Resize diagrams

When a diagram is created the default is that the diagram is placed in an auto size mode. This mode can be turned off or on from the diagram shortcut menu or the diagram element properties toolbar. When the diagram is in auto size mode diagram elements can be dropped, inserted, pasted, moved etc. anywhere on the editor canvas. If the element is placed outside the frame the diagram will automatically be resized to fit the element. The auto size mode will always keep the minimum full pages to contain all diagram elements.

The initial diagram size and layout is determined by the Print settings (size and orientation). If no printer is installed the size is determined from the current Options set. Resize of the diagram is done in two steps. If the diagram is to be enlarged, it can be done in one of the following ways:

- Change the size of the diagram from the shortcut menu. Right-click on the diagram in the Model view and select **Diagram size...**, this will open a dialog that allows you to control the size manually or return to auto sizing.
- When auto size is off, you can increase the diagram size in steps of whole papers by holding down CTRL and clicking on the drag handles of the frame symbol. If CTRL + SHIFT keys are pressed when clicking on the handles the diagram will be decreased in size. The mouse cursor will change appearance according to which keys are pressed.

The frame symbol is automatically resized to match the resize of the diagram. When resizing the diagram to make it smaller the frame symbol may not be made smaller than the extent of all symbols (or lines) in the diagram. The frame must be at least 1 grid point apart from any symbol (line) inside. The

diagram name symbol will be moved automatically when the diagram is resized. Ports and lines connected to the frame move with the frame. The size of the canvas may not be made smaller than the frame + margins. Resize of the frame should follow the grid.

Find

Open the **Find Diagrams and Definitions** dialog by choosing **Find** from the Edit menu. Through the dialog it is possible to locate definitions. To find the usage of a definition it is possible to **Find text in diagrams too**. Results are shown in tab Search result in the Output window. To list where an entity is used, right-click the entity in the Model view and from the shortcut menu click List references.

See also

"Model Index" on page 82.

Text parsing

In general, C++ style syntax is used in text symbols (and external files). UML style syntax is used in all other symbols. However, the parsers accept some minor deviation from UML ("public" instead of "+", "output" instead of "^", etc.) but will unparse everything in UML style. In text symbols, UML style deviations are accepted, but will be converted to C++ style (i.e. "+" as visibility operator is converted to "public").

These changes fall into this category:

- Visibility: `+` converts to `public`
- Visibility: `#` converts to `protected`
- Visibility: `-` converts to `private`
- Signal Sending: `^` converts to `output`
- Decision alternative: `else` converts to `default`

There are some other properties of the unparse phase:

- "`in`" parameter direction kind is not unparsed
- quotation marks are removed from names that do not need it (for example `'Name1'` becomes `Name1` after unparse)
- "`in / out`" parameter direction kind is unparsed as "`inout`"

- data types only with literals become enumerated data types, i.e. `datatype colors { literals red, green; }` un-parsed as `enum colors { red, green }`

The unparser will expand shortcut notations. Several attributes, remote variables, signals, timers, exceptions or synonyms defined at once (for example, `Integer i, j, k;`) will be unparsed as several separate definitions (`Integer i; Integer j; Integer k;`).

The unparsed adds omitted parenthesis in signal and timer definitions (i.e. “`timer T`” becomes “`timer T()`”).

Open ranges (if possible) are converted to UML style: “`>= n`” is converted to “`n..*`” and “`>=0`” is converted to “0..*”.

Auto-quote

The purpose of auto-quote is to assist you with typing quotation marks. If you add a whitespace in a name there should be quotation marks added wrapping the text. Only a limited set of symbols (labels) are auto-quoted. Typically labels that contains names are auto-quoted.

Word wrapping

Word wrapping allows words to be broken up into several lines. This is applied to multiline labels and to non-autosized symbols. The algorithm for determining where to break a word looks for any of the following: ‘`:`’, “`::`”, whitespace, capital letter, comma (‘`,`’), period (‘`.`’), underscore.

Diagram auto layout

The shortcut menu for diagrams (canvas background) includes an **Automatic layout** menu item for diagram types that has an auto layout algorithm associated with it. If this menu item is selected the diagram elements will be placed in a layout suitable for that specific diagram type. For example will Class diagrams and state machine diagrams have a hierarchical layout.

The auto layout algorithms will be used when placing diagram elements with the Show elements dialog.

When using the auto layout:

- It is only Generalization lines that are included in the layout algorithm in class diagrams.

- Flow lines and transition lines are included in the layout algorithm in state machine diagrams.

Organizing the view

The editors have several features allowing you to organize your view. These include shortcuts for scrolling and zoom in/out.

When a diagram is closed, the current scroll and zoom settings can be saved to a separate file with the extension `.u2x` and the name `<project>_DiagramSettings`. This file is not added to the project (ttp file). Instead when a project is loaded there is a step that loads files with the extension `.u2s` and a corresponding name. This feature is controlled by the option **Remember scroll and zoom**.

Scroll

When your diagram is sized so it is not entirely visible in the desktop area it is possible to scroll the view. Scrolling your diagram view can be done with the window scroll bars.

(**Windows**) It is possible to scroll with the IntelliMouse pointing device.

- Vertical scrolling is done by use of the scroll wheel.
- Horizontal scrolling is done by pressing CTRL and at the same time use the scroll wheel.

Zoom

It is possible to zoom in fixed steps via the Zoom command on the View menu.

It is possible to do a continuous zoom via the shortcut menu. Right-click in a diagram, point to zoom and select the desired enlargement level. When not in text edit mode it is possible to use “-” (minus) for zoom out and “+” (plus) for zoom in.

(**Windows**) It is possible to zoom with the IntelliMouse pointing device.

- Press SHIFT and use the middle mouse button, the diagram will zoom.
- Double-click on the scroll wheel will zoom to 100%.
- SHIFT + Double-click on the scroll wheel will zoom to fit the current diagram width in the desktop area.

See also

“Docking windows” on page 21

“Workspace Operations” on page 399

DOORS Analyst commands

In this section, a set of commands which are specific for the synchronization between the DOORS Analyst window and DOORS are described. These commands are represented in three different ways:

- by buttons in a specific tool bar for DOORS Analyst layouts
- in a menu named Analyst
- by right-clicking on an object in a diagram and selecting the command from the shortcut menu

Switch Layout

The Basic layout is by default set to display the diagrams and the most common tool bars for editing. The Switch Layout button allows switching to and from a view containing the Workspace window (with the Model View) and more tool bars.

When starting DOORS Analyst initially the set of symbols available for editing is set by a configuration (Metamodel) named “Analyst view”. This allows for a high visibility of the most common symbols.

Show elements

When creating a diagram in DOORS Analyst it is possible to use the Show elements button (this is also available from the shortcut menu when you right-click in a diagram) and get a dialog where it is possible to select which elements to insert in the current diagram. The elements will be placed with an auto-layout function.

Get DOORS Changes

This button allows you to synchronize the DOORS Analyst contents with the DOORS objects in the formal module. This is useful when you work with both the DOORS formal module and in the DOORS Analyst UML view in parallel and want to propagate the DOORS changes into the UML view.

Check DOORS Changes

This command allows you to check for changes in the DOORS objects in the formal module. This is useful when you work with both the DOORS formal module and in the DOORS Analyst UML view in parallel.

Edit in DOORS

This button will switch focus from the UML element to the corresponding element in the DOORS module.

Enable synchronization with DOORS

When one or several symbols in a diagram are selected, this command will enable the synchronization of these symbols with DOORS. This means that corresponding objects will be created in the DOORS formal module the next time a synchronization is performed. The “Object Type” column in the DOORS formal module will contain the text “Other”, meaning that this object is not one of the standard predefined model elements in DOORS Analyst.

Note that the synchronization for these objects is one-way, which means that changes to these objects must be done in the diagrams themselves. It is not possible to propagate changes from the DOORS formal module to the diagrams for these objects.

Disable synchronization with DOORS

When one or several symbols in a diagram are selected, this command will disable the synchronization of these symbols with DOORS. However, the pre-defined model elements in DOORS Analyst will still be synchronized with DOORS.

See also

“DOORS Analyst commands in DOORS” on page 3

Common Symbol Operations

- “Symbol information” on page 118
- “Add symbols” on page 119
- “Show elements” on page 120
- “Select symbols” on page 121

- “Move symbols” on page 122
- “Resize symbols” on page 122
- “Connect symbols” on page 123
- “Edit text fields in symbols” on page 125
- “Diagram element properties” on page 125
- “Handling comments” on page 126
- “Copy, cut, delete or paste symbols” on page 127
- “Icon” on page 128
- “Image Selector” on page 129
- “Undo” on page 130
- “Model references” on page 130
- “Nested symbols” on page 132

Symbol information

The selection of **Show symbol and line tooltips** will allow you to see contextual model information, for example stereotype and model bind information.

The selection of **Show edit mode tooltips** will allow you to see information from syntax parsing while in text edit mode.

Show/hide model element details toolbar

The **Show/hide model element details** toolbar lets you toggle showing and hiding of certain features of symbols in diagrams. These settings are remembered per diagram element. If the setting is applied on the diagram, all diagram elements without a setting of their own, will inherit the diagram setting.

- Show/Hide qualifiers

  Toggles the qualifying parts of a label text. For example: `Package1::Package2::Class1` will be toggled to `Class1`.
- Show/Hide stereotypes

  Toggles the stereotype label. When the label is hidden the space occupied by it is considered minimal and can thus affect the symbol size.

- Show/Hide quotation marks

  Toggles the automatic quotation marks, thus a limited set of symbols are auto-quoted and those symbols are affected by this button. The text will still be considered quoted when the quotation marks are hidden. Auto-quoted text is typically names containing a whitespace.

Add symbols

To add a symbol, you click its corresponding icon in the Diagram element toolbar, then click or right-click in the diagram to position the symbol.

It is also possible to generate symbols from the Model View. You can in many cases drag a Model element into the desired diagram.

In state machine diagrams it is possible to Insert a symbol in the flow by pressing CTRL and then click in the diagram element toolbar on the new symbol. The symbol will be positioned after the currently selected symbol.

Reference existing

When you right-click to position a symbol a shortcut menu will appear for most symbols. Symbols that do not have this menu are:

- Transition line (used in state machine diagrams when designing in State-oriented view).
- State machine transition symbols not related to states, signals and operations.

The shortcut menu contains the following choices: **Create New <element>**, **Leave Unbound**, **Reference existing**.

- **Create New <element>**: a new symbol will be created and a corresponding model element will be created in the model.
- **Leave Unbound**: a new symbol will be created but no corresponding model element will be created.
- **Reference existing**: will display a drop-down box with existing model elements matching type and scope.

Auto placement

In many cases it is desired to position a symbol in connection with the previous, for example a port on a class. For this purpose auto placement of symbols is supported.

- Hold down SHIFT and click the symbol toolbar. The symbol clicked will be connected to the currently selected symbol.
- Hold down CTRL and click the symbol toolbar. The symbol clicked will be inserted between the currently selected symbol and the next one.

The symbols in the toolbar will be dimmed if they cannot be connected in a syntactically correct flow to the currently selected symbol.

It is also possible to use the SHIFT + SPACEBAR and CTRL + SPACEBAR shortcuts to get a list of the symbols that can be auto placed.

See also

"Name support" on page 44 in Chapter 2, *Working with Models*

"Create Presentation" on page 76 in Chapter 2, *Working with Models*

"Model navigation/creation" on page 78 in Chapter 2, *Working with Models*

"Diagram auto layout" on page 114

"Show elements" on page 120

"Creating a message" on page 171

Show elements

The **Show Elements** dialog makes it possible to decide what model elements to show as symbols in the current diagram.

Show Elements is available:

- From the Tools menu
- In diagram shortcut menus.

When **Show Elements** is invoked, a dialog appears, with a list of model elements that can be shown as symbols in the current diagram. By adding or removing check marks for model elements, you can add or remove symbols for their corresponding presentation from the diagram.

The **Show Elements** dialog has the following features:

- The **All** button allows you to check all model elements in the list with one click.
- The **None** button allows you to remove all existing check marks with one click.
- The **Short list** check box makes it possible to toggle between a short and a long list of model elements.
  - The **short list** contains model elements that are natural to show in the current diagram. (For instance, a class in a class diagram.) Model elements that are already shown as symbols in the current diagram, are also included in the short list, even if they are not natural to show in the current diagram.
  - The **long list** contains all model elements from the selected scope that can be shown as symbols in the current diagram. In addition to the natural model elements for the current diagram, this list also includes more uncommon conversions. (For instance, the alternative to show a class as an actor in a use case diagram is included in the long list.)
- The **Select scope** button brings up a dialog, in which it is possible to select the scope or scopes to pick model elements from for the list in the main dialog. As default, only model elements from the local scope where the diagram resides is included in the list.

A model element is automatically deleted when the last presentation element (symbol or line) in a diagram has been deleted.

Select symbols

Pressing CTRL while double clicking outside a text field (but within the symbol boundaries) will select the symbol, outgoing lines and all connected symbols.

Click and drag (when not clicking on a symbol or line) will create a selection rectangle. Everything that is placed within the rectangle is selected.

Click and drag while pressing CTRL (when not clicking on a symbol or line) will create a selection rectangle. Everything that is intersecting the rectangle is selected.

In a state machine flow if you do CTRL + double-click on a symbol in a flow this will select the symbol and all symbols after. This also works when the flow is branched.

Move symbols

To move a symbol, click it and drag it to the desired location within the diagram. It is also possible to drag symbols to other diagrams.

Symbols with text fields should be selected as not to enter the text edit mode. The cursor will change appearance to indicate this.

Moving text fields

It is possible to move some text fields (labels). This is true for labels that belongs to lines and for labels that belongs to symbols with the label outside of the symbol boundaries (port, pin, etc.).

This is done by first selecting a label, which then can be dragged in one of its handles. The new position will be saved as an offset to the default position. If you move a line or symbol with a label, then the label will also move and preserve the offset.

The offset can be removed by clicking on the shortcut command **Reset all label positions**. That will reset all labels belonging to the currently selected symbol to their default position.

Labels can be dragged to anywhere on the desktop, even outside the frame symbol and diagram area. Labels outside the diagram area will not be printed.

Resize symbols

To resize symbols manually

1. Select the symbol in question.
2. Place the mouse over one of the eight gray squares.
3. Drag the mouse until the symbol is the size you want it to be.

Autosize symbols

All symbols offer the choice of **Autosize**, which adapts the symbol to the size of the text entered within it. Right-click the symbol and choose **Autosize** from the shortcut menu.

Collapse symbol

A symbol with compartments (for example a class symbol) can be collapsed by checking the “Collapsed” menu item in the shortcut menu for that symbol. Compartments and any labels inside the compartments will not be visible when in a collapsed state.

Resized symbol indicators

When three dots appear outside a symbol’s lower right corner, the size of the symbol is too small to show all text that is available in the symbol’s text fields. To resize the symbol to adapt to the text, select the symbol and double-click the dots.

Connect symbols

To connect symbols manually

1. Click a symbol and find its line handles.
2. Drag the line to the other symbol.
3. The end of the line will appear as a circle with crossbars when you reach the second symbol. Complete the connection by clicking inside the symbol close to the border position where you would like to attach the line.

Some connections will result in model elements. If you for example drag the generalization handle to another class in a class diagram you will end up with a line between the two classes as well as new icons in the Model View, informing you that a generalization has been added.

Symbols in a State machine diagram can be connected automatically to a flow with Auto placement.

See also

“Draw lines” on page 136

Symbol flow editing

In diagrams which have a concept of flows, such as state chart diagrams and activity diagrams, there are certain operations possible to manage these flows.

Select a flow or a branch of a flow

If you do CTRL + double-click on a symbol in a flow this will select the symbol and all symbols after the selected symbol. This also works when the flow is branched.

Append symbols to the flow

When adding symbols in an diagram you can create a connected flow of symbols. Hold down SHIFT and click the toolbar. The symbol clicked will be connected to the currently selected symbol. The symbols in the toolbar will be dimmed if they cannot be connected in a syntactically correct flow to the currently selected symbol.

See also

“Auto placement” on page 120

Insert a symbol in the flow

Insert operation is possible when CTRL is pressed and a button is clicked while the selection corresponds to one of the following cases:

- A single selected symbol (insert operation after this symbol)
- A single selected line (insert operation on this line)
- Two selected symbols with one line between them (insert operation between the symbols)

Note

- *Using **CTRL** + **decision symbol** is not possible. There are several possible flows out from a decision symbol and because of this it is not supported to insert a decision symbol this way.*

See also

“Auto placement” on page 120

Remove a symbol from the flow

When a symbol is cut or deleted from a flow an auto-created line replaces the removed symbol and its connected lines if possible.

Edit text fields in symbols

To be able to edit a text field in a symbol, the symbol must first be selected.

- To edit a text field in a symbol, select the symbol and click in the text field at the position where you would like to add or change. You are now able to enter your text changes. Text within guillemets, «» (for example stereotype information), cannot be edited.
- If a symbol is selected, and a double-click is done in a text field the closest text word will be selected. If the symbol is not selected, the double-click will be done on the symbol (normally navigation).
- If a symbol is selected, click and drag in a text field in the symbol will enter edit mode and select the text. If the symbol is not selected, the symbol will be moved.
- Pressing F2 will enter edit mode for the main text of the selected symbol. For a single line text all text will be selected, while for a multi line text, there will be no selection and the text insertion marker will be placed at the end of the text.

Note

Double clicking outside a text field (but within the symbol boundaries) will always perform the double click operation for the symbol (normally navigation).

Diagram element properties

A specific toolbar called Diagram Element Properties is available. This will contain drop-down boxes that control various properties of the selected symbol(s)/line(s):

- font
- font size
- symbol / line background color

The toolbar contains a button that will remove the set properties and revert to default styles.

If no symbol is selected, the toolbar commands will apply to all symbols in the current diagram, except symbols with individually applied properties.

Handling comments

Comment symbols can be added to all symbols.

1. Click on the comment symbol on the toolbar.
2. Position the symbol in the diagram.
3. Connect the annotation line from the comment symbol to the symbol you want the comment to belong to.

Comments and constraints

The shortcut command **Show Comments** for Signature symbols (in the submenu to **Show/Hide**) will create and attach a comment symbol for each comment model element owned by the signature symbol that does not already have a comment symbol in the current diagram.

The shortcut command **Show Constraints as Symbols** for Signature symbols will create and attach one constraint symbol for each constraint model element owned by the signature symbol that does not already have a symbol in the current diagram.

Column of Remarks

Two or more comment symbols form a **Column of Remarks** when they are positioned close and aligned or almost aligned in vertical position, see Figure 35 on page 127. The vertical positions will be auto-adjusted to form a left-aligned column when a column of remarks is detected.

The column can be moved in the horizontal direction by pressing SHIFT and moving the top comment symbol a small vertical distance (less than the total width of the column). If another comment symbol in the column is moved a small distance (with SHIFT pressed), it will be repositioned back into its place in the column. Moving any comment symbol a larger distance will remove it from the column.

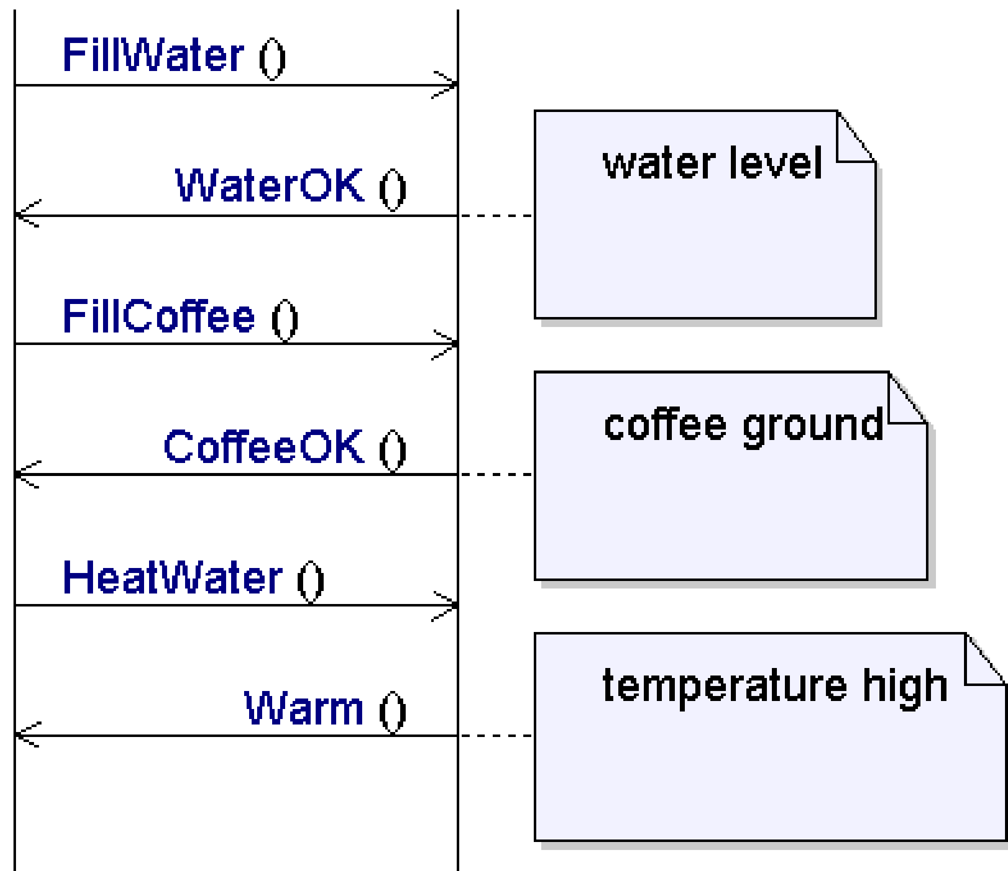


Figure 35:Column of remarks

Note

The top comment symbol in the column of remarks must be positioned below the lifeline headers to be included in the column.

Copy, cut, delete or paste symbols

All symbols have a shortcut menu that you may access by right-clicking on the symbol. From this menu, choose **Cut**, **Copy** or **Paste** according to your needs.

You can also drag symbols to other tools, for example MS Word.

To delete a symbol, select it and press the **Delete** key.

Note

A Delete operation may or may not affect your model depending on the type of symbol and its relation to the model. When you add symbols to your diagrams this will in most cases add information to your model. When you delete a symbol in a diagram it will only delete information in the model if there is a one-to-one relation with the symbol and the model. This is for example the case with State machine flow symbols. To delete a symbol and its corresponding model element use Delete from Model.

Icon

User-specified icons

It is possible to use an image file and replace selected symbol icons with a user-specified icon. The icon can be specified on the following levels:

- for a specific symbol
- for a specific semantic model element, implying that all symbols that are associated with the model element will use the icon
- for a specific stereotype, implying that all symbols that are associated with model elements stereotyped by the symbol will use the icon
- for a specific type (for example class or datatype), implying that all symbols that are associated with instances of the type will use the icon

Add stereotype

This feature is controlled by a stereotype. To activate this you right-click the model element that is to have the icon, and from the shortcut menu select Apply Stereotypes. In the dialog apply the stereotype `TTDStereotypeDetails::`Icon. You can also through the Properties editor open this dialog using the **Stereotypes** button.

The entities that this stereotype can be applied to is controlled by the Metamodel properties. To view this information go to the `Library` section in the Model View and open the package for `TTDStereotypeDetails`. In the class diagrams you can view the relations between the supported entities (metaclasses) and the icon stereotype.

Ordering

If more than one user-specified icon is specified among the alternatives above, then the order is according to the list above. Thus, if an icon is specified for a specific symbol, this will be used, otherwise the icon for the model element is used and so on.

Icon mode

For symbols identified with a user-specified icon there will be a shortcut menu choice called **Icon mode**. When selected this menu choice will cause the symbol to be visualized instead of the usual symbol.

Image file

The icon is defined by a property of the symbol, model element or applied stereotype and can be changed using the Properties dialog for the entity. In this dialog select **Icon** in the Filter drop-down menu to display a text field called Icon File. The text in this field is a relative path from the model file (`.u2`) to where the image file can be found.

The formats that are supported for the icon image files are

- bitmap (file extension "`.bmp`")
- JPEG compressed images (file extension "`.jpeg`" or "`.jpg`")
- Enhanced Meta File (file extension "`.emf`")
- GIF (file extension "`.gif`")
- TIFF (file extension "`.tif`", "`.tiff`")
- Targa (file extension "`.tga`", "`.targa`")
- PCX (file extension "`.pcx`")

Note

Using a white and transparent background for an icon image may result in a black background when Printing Diagrams.

Image Selector

The symbols in diagrams can also be displayed with a user-defined image using the Image Selector. Activate the add-in `ImageSelector`, then the commands **Load image** and **Remove image** are available in the **Tools** menu.

Undo

Multiple level of Undo and Redo is available. The whole tool (workspace window and editors) has a common undo stack. When an operation is undone, it is put first in the redo stack to make it possible to redo the undone operation.

Note

You can undo operations in diagrams that are not currently visible.

Some special considerations have to be taken when using Undo and Redo when in text-editing mode. Undo steps will be available for each update. An update is made according to the following scheme: a series of character additions will not cause an update until you do something else, for example back space, delete, arrow keys or mouse selection. Similarly will a series of deletes not cause an update until you do something different.

When doing an explicit unload (including revert) of a file/resource in a project the Undo stack is emptied.

Undo is not possible for file system operations.

The Undo stack is not emptied when a Save is performed.

Model references

To find references to model definitions and their usage there is a group of shortcut commands with similar features. These commands have a contextual nature meaning that they will be dependant on the element that they are applied on.

List references

List References is a shortcut command in the Model View, which applies to all model elements. The command calls up a dialog and returns a reference list in the References tab of the Output window. Possible settings:

- **References made to...**
  The listing contains all references to the model element, for example usage of a specific class as a type for an attribute is a reference to the class.

- **References made from...**
  The listing contains references from a selection, for example references from an attribute to the class used as its type.
- **Include Contained Hierarchy in Report**
  When selected, this alternative will cause the tool to recursively include any references to or from the elements contained under the selected element. An example is to find for all definitions outside a package, used by the package and its contained definitions.
- **Include Internal references in Report**
  When selected, this alternative will cause the tool to report references originating in the object or its contained hierarchy to itself, or to its contained hierarchy. An example of this is to find all uses of a package and of its contents without finding references made within the package.

List presentations

List Presentations is a shortcut command in the Model View, which applies to all model elements. Returns a list of all presentation elements in the Presentations tab of the Output window.

Reference existing

This is a shortcut command when placing a new symbol.

Navigate

Shortcut command in the Model View, opens the Model Navigator or if there is no existing presentation for the element the Create Presentation dialog.

When you have a selection of multiple nodes in the Model view the commands **List References** and **List Presentations** will be applied to all selected elements.

See also

"Add symbols" on page 119 in Chapter 3, *Working with Diagrams*

Nested symbols

Some symbols can be placed inside other symbols. When a symbol is created inside another symbol, the model element of the parent symbol is used as context for creation.

If the parent symbol is auto-sized the size of the parent will change to fit the created symbol. Otherwise the new symbol will be resized to fit within the boundaries of the parent symbol. A nested symbol can not be dragged outside the parent symbol boundaries.

Symbols with compartments

Symbols with compartments have some special functionality related to the compartments and the contained text fields.

Compartments can contain text fields which in turn is associated with model elements. Compartment text fields are left-aligned.

Certain symbols, like for example the class symbol, are created with a default set of compartments. The class symbol for example will have an attribute and an operation compartment.

Compartments can be selected, and there are a set of operations possible to perform on them.

When hovering over a compartment for a moment, a tool tip will display the compartment type.

Resizing

When a symbol with compartments is resized, any compartment that does not fit in the symbol will be hidden. If some content in the compartment can be hidden instead of the entire symbol, this will be done instead.

When a symbol is larger than necessary to display all the compartment contents, then the extra space will be evenly distributed among the compartments.

Creating compartments

If a symbol can contain compartments there is a **Compartments** sub-menu available on the symbol’s shortcut menu. The sub-menu contains a set of operations for creating compartments in the form **Create <Element> compart-**

ment. Executing one of these operations will create a compartment that can be used to create and display elements of the element type. New compartments will be added in the bottom of the symbol.

Deleting compartments

A compartment can be deleted by selecting it and using the normal **Delete** command. Certain compartments can also be directly associated with the model elements that the compartment is showing, and in that case the **Delete Model** command can also be used.

Moving compartments

The order of compartments can be changed by using the **Move Up** and **Move Down** commands available on the **Move** toolbar.

Show/Hide on compartments

When a specific compartment is selected, the shortcut menu will give the possibility to show and hide elements of the type that the compartment is used for. Show and hide operation will only be displayed in the shortcut menu if they are applicable.

When the symbol is selected, the shortcut menu will give the possibility to show and hide elements in any of the existing compartments, or create a new compartment to display a certain type of model element. These operations will only be displayed if applicable. If there are several compartments showing the same type of model element, show and hide operations will only be done on the one first in order. To show and hide elements on a specific compartment use the shortcut menu of the compartment instead.

It should be noted that the owned model elements will not be shown by default if an already created element is dragged into a diagram.

See also

"Default Class symbol appearance" on page 643

For the elements to become visible it is also possible to drag-and-drop them into the compartment or symbol or type them in manually.

Hint

Using Name completion is a good way of avoiding to create new features by mistake. Start typing the name, press CTRL + SPACEBAR or SHIFT + SPACEBAR, if there are multiple possibilities a list will be displayed.

Compartment text fields

Delete element

A text field in a compartment is a separate presentational element that is associated with a model element. Due to this it is not possible to delete a text field connected to a model element by deleting all text on the line. To delete the element associated with the label, enter text mode and use the **Delete <Element>** operation from the shortcut menu.

Note

It is not possible to delete a text field connected to a model element by deleting the text on the line, this will only delete the characters on that line. The row will still be connected to the model element.

A text field **not** connected to a model element can be deleted by deleting all the text on the line and then pressing backspace or delete. It is also possible to delete an empty text field not connected to a model element by pressing backspace when the text cursor is first on the text line below or by pressing delete when the text cursor is last on the text line above.

Hide element

To hide a specific element displayed in a compartment text field enter text edit mode and use the **Hide <Element>** operation from the shortcut menu.

Move text fields

It is possible to move feature text fields up and down with the **Move Up** and **Move Down** toolbar buttons (found on the **Move** toolbar).

Common Line Operations

- Line styles
- Draw lines
- Editing vertices

- Move lines
- Delete lines
- Re-direct and bi-direct lines

Line styles

There are five different line styles that can be applied to a line. After a line is created these styles are available in the context menu on the line.

Auto-routed (assign endpoints)

Line is routed automatically so that obstacles are avoided. Line is orthogonal as long as there is a possible route for the line. In other cases the line is drawn as a straight line. Endpoints are automatically reassigned to make the shortest route as possible. When an endpoint is moved the line style of that line will automatically be changed to Auto-routed (keep endpoints).

Note that there is no difference in behavior from the Auto-routed (keep endpoints) line style if the line can only be connected at the center of a symbol.

Auto-routed (keep endpoints)

Line is routed automatically so that obstacles are avoided. Line is orthogonal as long as there is a possible route for the line. In other cases the line is drawn as a straight line.

If the source endpoint is shared with another line of the same type, a tree structure will be routed as far as it is possible. This is only possible for certain type of lines that are common to draw as tree structures, such as the generalization line.

Orthogonal

Line is always kept orthogonal and line vertices and segments can be moved. Vertices can be added and removed from the line.

Non-orthogonal

Line vertices can be moved, added and removed without restrictions. If a non-orthogonal line is rearranged into an orthogonal line, the line style is automatically changed to Orthogonal.

Bezier

Will give the line a curved layout. When the line is selected two control points are displayed which can be used to shape the curve.

See also

"UML Editing Line Styles" on page 645

Draw lines

A line can be created either by using the toolbar button or the line handle representing the line.

Creating a line with a line handle

1. Select the source symbol.
2. Click the line handle.
3. Add vertices and/or lock endpoint (optional).
4. Click the target symbol or line.

Creating a line with a toolbar button

1. Click the toolbar button.
2. Click the source symbol.
3. Add vertices and/or lock endpoint (optional).
4. Click the target symbol or line.

Vertices can be added while creating the line, with the exception of auto-routed lines. When it is allowed to place a vertex the cursor will have the shape of a plus sign.

For all line styles it is possible to lock the starting point position to the symbol edge, if the starting point is selectable. When creating a line with the line style Auto-routed (assign endpoints) or Auto-routed (keep endpoints) a cursor in the shape of a padlock is displayed. When clicking at this state the starting point of the line will be locked to its current position. If the line have the line style Auto-routed (assign endpoints) as default line style and the endpoint is locked in this way, the line style will automatically be changed to Auto-routed (keep endpoints). When creating a line with a line style different from Auto-routed (assign endpoints) and Auto-routed (keep endpoints) the starting point can be locked by holding down SHIFT and clicking.

Editing vertices

To add a vertex for an existing line hold down CTRL and click on the segment where the vertex should be created.

To remove a vertex hold down CTRL and click on the vertex that should be removed.

This can only be done for lines with the line style Orthogonal or Non-orthogonal applied. The mouse cursor will change to indicate that the operation is possible.

See also

"Connect symbols" on page 123

Move lines

To move a line, click one of the endpoints and drag it to the desired location.

Delete lines

Lines are in many cases representing a model element which will remain in the model even if they are deleted from a diagram. If you want to completely remove a line, for example an association, then make sure to use Delete from Model.

Re-direct and bi-direct lines

- To re-direct a line (when applicable), right-click the line and choose **Re-direct** from the shortcut menu.
- To bi-direct a line (when applicable), right-click the line with the cursor close to the side without direction or signal list. Select **Enabled Direction** from the shortcut menu.
- If a line is bi-directed, and you want to allow it only one direction, locate the cursor over the line, close to the side you want to disable. Then deselect **Enabled Direction** from the shortcut menu.

  You can also re-direct the line before or after this operation to make it point in the desired direction.

4

UML Language Guide

This chapter describes the UML language as implemented and supported in DOORS Analyst 4.2.

- For more information on the supported version of UML, see “UML version” on page 140.

See also

Chapter 2, Working with Models

Chapter 3, Working with Diagrams

Introduction

UML is a modeling language that allows you to specify, visualize, document, and construct software and systems. In subsequent sections you find information about the different diagrams and constructs that can be used to describe the structure and behavior of systems at different levels of abstraction. Some constructs are more useful in early development phases, such as requirements and analysis, while others are more useful in later development phases, such as design, implementation, and test. This ability to tie together the different development phases is one of the primary strengths of UML.

UML version

The language used in DOORS Analyst is based on the latest OMG UML 2.1 Superstructure submission. In some cases the implementation of DOORS Analyst differs from the language specification; this is primarily due either to tool optimizations or the fact that some design decisions were founded on earlier versions of the submission.

DOORS Analyst also includes some extensions to the language, for example the possibility to use a textual syntax in conjunction with the graphical notation defined for UML.

Diagrams

UML consists of a set of diagrams that are used to express different viewpoints of a system. Some diagrams focus on the structure of the system, while others are dedicated to describing behavioral aspects of the system, such as how an entity interacts with another entity or the set of actions to be performed under specific conditions. Typically, these diagrams are the primary means through which you specify systems.

The diagrams that are supported in DOORS Analyst are the following:

Diagram	Purpose
Use case diagram	Describes how a set of actors interacts in terms of use cases, usually in the context of a subject (the described system).
Sequence diagram	Describes the event sequence for a use case or an operation.
Package diagram	Describes packages and dependencies between them.
Class diagram	Declares classes and their relations to each other, typically in the scope of a package or another (container) class.
Composite structure diagram	Describes how parts of a (container) class are connected to each other to form an internal structure of the container.
Activity Diagram	Used to show parallel and intertwined behavior. This may allow a simplified view of a complex structure where it is possible to focus on a specific flow of control.
Interaction overview diagram	Describes some form of parallel behavior. It is often used to describe a use case.
Component diagram	Focused on the design of components and shows relations and structure of components
Deployment diagram	Used to show how the physical implementation is structured and the relations between software and hardware
State machine diagram	Defines the behavior of classes, state machines and operations.

Models and diagrams

A model is a representation of a physical system, and is typically defined by the entities contained in one or more packages.

Since the model is a representation of a system, it should only be as detailed as necessary. If, for example, it should be used as the source for automatic application generation, it needs to contain quite a lot more detail at an algorithmic level than if it is used to visualize requirements.

The model contains all entities that are necessary to describe a system; this includes diagrams and model elements. The model, or rather the model elements it contains, are typically shown in different diagrams using symbols (sometimes called presentation elements, as opposed to model elements).

Model elements

The primary contents of a model are model elements such as classes, attributes, operations, actions, and constraints. A model element is used to store all characteristics of an entity. It is then possible to show different aspects of a model element in diagrams. For example, one class diagram may show the attributes and operations of the class, while another class diagram may shown the class hierarchy in which it is defined. These diagrams give partial views of the same model element, but many more are possible.

Symbols

Symbols are used to graphically visualize (parts of) model elements. Each symbol is a two-dimensional object that is shown in a diagram. It has a size that specifies its dimensions and a position that is given in terms of the coordinate system of its diagram.

Most symbols are direct visualizations of a corresponding model element, such as the class symbol but there are a few that have no underlying model element, such as the text symbol. These are then only associated with a specific diagram.

The distinction between model element and symbol is important, but in daily speak the distinction between the two is often blurred. A class model element or a class symbol is commonly referred to simply as class.

Different views of a model element

In Figure 36 on page 143, there is an example of the model element a, which is shown using three different views. First, it is shown as an attribute of the class C. Second, it is shown as an association end between the classes C and D. Third, it is shown as a part of the internal structure of class C.

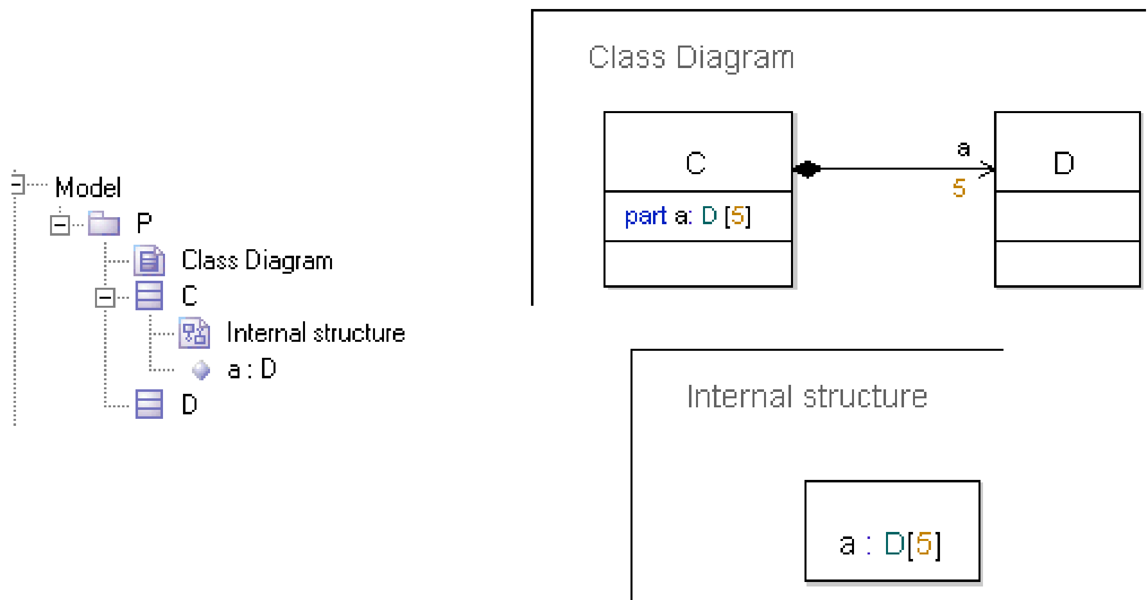


Figure 36: Example of different views of an attribute

Here it is also possible to appreciate the difference between a model deletion (Delete from Model) and an ordinary Delete from a diagram. The browser view to the left shows the model, and when deleting elements from the browser view you delete them from the model *and* the diagrams in which they are shown. The two diagram views to the right, however, show the same attribute `a` in three different ways: in the attribute compartment of the class C, as an association end between the classes C and D, and as a part of the internal structure of the class C.

Deleting symbols and model elements

Elements can be deleted in two different ways from a diagram. An ordinary Delete removes the symbol, but the model element is retained in the model. A Delete from Model operation deletes the element from the model and from *all other* diagrams in which it is shown.

In some diagrams, model elements and symbols are tightly connected with each other. This includes sequence diagrams, and state machine diagrams. Here, there is a one-to-one mapping between symbols and model elements, and if one is deleted then the other is also deleted. (In other words, a delete is the same as a Delete from Model in these diagrams.) In particular, this is applies to for example actions and transitions, but not for states.

See also

“Add symbols” on page 119

“Move symbols” on page 122

“Resize symbols” on page 122

“Connect symbols” on page 123

“Edit text fields in symbols” on page 125

“Copy, cut, delete or paste symbols” on page 127

List of language constructs

The following table lists all the concrete model elements as well as the most significant other language constructs in UML.

UML model element
Accept Event, Accept Time Event, Access, Action (in operation body, state machine and state machine diagram), Action (in interaction and sequence diagrams), Action Node (in activity diagrams), Active class, Activity, Activity Final, Actors, Aggregation, Arbitrary value (any) expression, Artifact, Assignment, Association, Attribute
Behavior port
Choice, Class, Classifier, Comment, Component, Composite state, Composition, Compound statement, Conditional expression, Constant, Connector (in composite structure diagrams), Connector (in activity diagrams), Continuation, Co-region, Create
Datatype, Decision (in state machine diagrams), Decision (in activity diagrams), Dependency, Deployment, Deployment specification, Diagrams, Destroy
Entry connection point, Execution environment, Exit connection point, Expressions, Extension
Field expression, Flow Final, Fork
Generalization, Guard
History nextstate

UML model element
Imperative expressions, Realization, Import, Index expression, Initial Node, Inline Frame, Interaction, Interaction reference, Interface, Internals
Join, Junction
Lifeline, Literal
Manifestation, Merge, Message, Method, Method call
New, Nextstate, Node, Now expression
Object Node, Offspring, Operation, Operation body, Signal sending action (output)
Package, Parent, Part, Activity Partition, Pid expressions, Pin, Port, Pre-defined, Profile
Range check expression, Realized interface, Required interface, Return
Save, Self, Send Signal, Sender, Signal, Signallist, Signature, Initial transition, State, State machine, State machine implementation, State expression, Stereotype, Stop, Subjects, Syntype
Tag definition, Tagged value, Target code expression, Action (task), This expression, Timer, Timer active expression, Timer reset, Timer reset action, Timer set, Timer set action, Timer timeout, Transition
Use cases

Scope, model elements, and diagrams

Some model elements, like packages and classes, represent name scopes. This means that they are allowed to contain definitions of other model elements. All definitions within a name scope must be uniquely named, or the semantic checker will complain. You can think of a scope as a container or grouping of model elements that belong together.

Most scopes may not only contain model elements, but also diagrams in which those model elements are shown. The table below shows which diagrams are available for each scope.

Scope unit	Allowed model elements	Diagrams
Package	Package, Class, Use cases, Artifact, Stereotype, Association, Datatype, Interface, Syntype, Choice, Operation, Attribute, Signal, Signallist, Timer, State machine	Class diagram Sequence diagram Use case diagram
Class	Class, Artifact, Stereotype, Datatype, Interface, Syntype, Choice, Signal, Signallist, Timer, Attribute, Operation, Use cases, State machine,	Class diagram Composite structure diagram
Use cases	Interaction, State machine implementation, Operation body	Sequence diagram State machine diagram
Interaction	Lifeline	Sequence diagram Use case diagram
Stereotype	Attribute	
Datatype	Literal, Operation	
Choice	Attribute, Operation,	
Interface	Signal, Timer, Attribute, Operation	
Operation	Operation body, State machine implementation, Interaction	
Operation body	State machine, Class, Artifact, Stereotype, Datatype, Interface, Syntype, Signal, Signallist, Timer, Operation, Attribute	State machine diagram

Scope unit	Allowed model elements	Diagrams
State machine implementation	Class, Artifact, Stereotype, Datatype, Interface, Syntype, Signal, Signallist, Timer, Operation, State, Action, Attribute	State machine diagram Class diagram Use case diagram
Activity implementation	Initial Node, Action Node, Object Node, Decision, Merge, Fork, Join, Connector, Accept Event, Send Signal, Accept Time Event, Activity Final, Flow Final, Activity Partition	Activity Diagram
Compound statement	Action, Attribute	

Overloaded Definitions

For certain kinds of definitions it is allowed to have many definitions with the same name in a scope. This is true for behavioral features, such as Operation, Signal, Timer and State machine. These definitions are identified not only by their names, but also by the types of their parameters. The name and list of parameter types is called the signature of the behavioral feature. All behavioral features in the same scope must have unique signatures. Two behavioral features in the same scope which have the same name, but different signatures, are said to be overloaded.

General Language Constructs

There are some language constructs in UML that are common to several diagrams.

Names

All definitions in a UML model should have a name—an identifier. There are certain rules to which these have to adhere.

Naming rules

The characters that are allowed in a name are letters, digits, and '_' (underscore).

The first character of a name cannot be a digit, but should be either a letter or an underscore. There is furthermore a special case for destructor names, which always start with an initial '~' (tilde).

Using spaces and special characters in identifiers

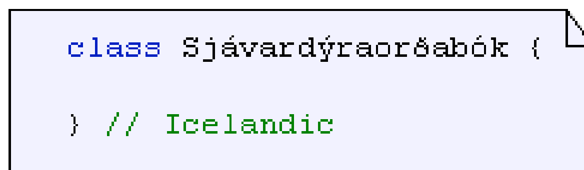


Figure 37: Using special characters in identifiers

By enclosing a name in single quotes, it is possible to get rid of the above mentioned restrictions, so that (almost) any character can be part of a name, see Figure 37 on page 148. For example, it is possible to use spaces in a name as long as the name is enclosed by single quotes, see Example 9 on page 148.

There exist a number of escape characters for string handling. They are \n, \t, \b, \r and \f and can be placed inside charstring (inside "") or used as a character (e.g. '\n').

The "`\"`" is used in charstring, "`\'`" respectively as character, and "`\\`" is used in both to represent backslash. Any other escaped character between quotes ('\+', '\s') is interpreted as identifier (+ and s respectively). Inside a quoted string any other character can follow the slash, representing just itself (e.g. "`a\qa`" = "`aqa`").

```
\n: new line
\t: tab
\b: backspace
\r: carriage return
\f: form feed
\": quotation mark, e.g. "my \"quoted\" word"
\': apostrophe character, '\''
\\: backslash
```

Example 9: Spaces in identifiers

```
Boolean 'has finished'=false;
```

Case sensitivity

Identifiers are case sensitive. This means that names that differ only in the way they use lower and upper case characters are distinct.

Example 10: Case sensitivity

```
Integer MyInt, myint; // Two distinct attributes
```

References

Named definitions may be referenced from other places in a model. In simple cases a reference just consists of the name of the definition (enclosed in single quotes if necessary). However, in the general case a reference can be more complex.

- A reference may contain a qualifier.

In some cases it is necessary to qualify a name in order to be able to distinguish a definition in one scope from another definition with the same name but in another scope. This is done by prefixing the identifier with the scope path and using the special scope resolution operator “::”. Global names have no path, and are simply preceded by “::”. Qualifiers that start with “::” are called absolute qualifiers, while those that start with a name are called relative qualifiers.

- A reference may contain actual template arguments.

If the referenced definition is a template (i.e. has Template parameters) the reference must contain actual values for its template parameters. The actual template arguments are given as a comma separated list after the name within ‘<‘ ‘>’ brackets.

- A reference may contain a list of parameter types.

When referring to a behavioral feature you must add the names of the parameter types, since not only the name but also the parameter types are part of the signature of the behavioral feature. The parameter type names are given enclosed in parenthesis after the name.

Example 11: Different kinds of references

In this example two attributes refer to their types using a qualified name.

```
::Predefined::Integer i;
UtilityTypes::Sorts::ClientIdx j;
```

If the type is a template class actual template arguments must be specified:

```
MyClass<Integer, 4> k;
```

When referring to a behavioral feature such as an operation, the parameter types must be specified. Note also the keyword ‘operation’ which must be used to syntactically disambiguate such a reference from an ordinary call of the operation.

```
OperationReference r = operation foo(Integer, Boolean);
```

Reserved words

Some names are reserved words in DOORS Analyst and cannot be used to name model elements directly. For a complete list of reserved words, see .

Although it is possible to use reserved words as names of definitions by enclosing them with single quotes, risk for confusion is apparent and this should be done only when absolutely necessary.

Example 12: Using single quotes for names that are otherwise reserved

```
Integer ‘class’; // confusing attribute name, but valid
```

Alternative syntax

In addition to the graphical notation defined by UML, a complementary textual syntax is defined for describing the model in plain text. This can be used in lieu of the graphical symbols, or in conjunction with them.

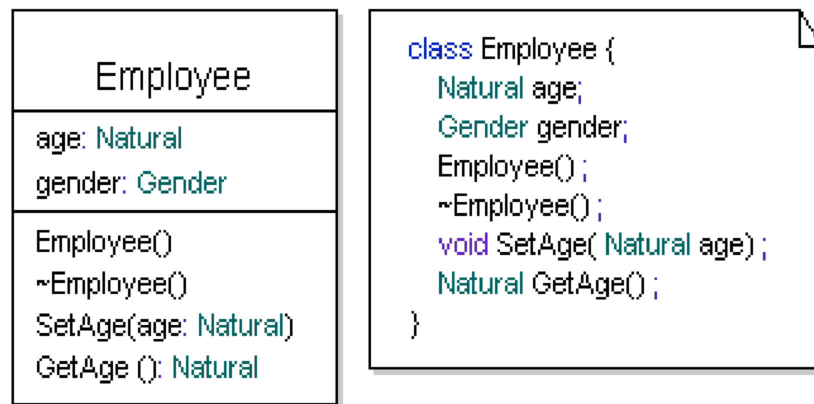


Figure 38: Example of differences between the syntax in class symbols and the textual syntax

In Figure 38 on page 151, the same model element is shown twice in a single diagram. To the left, a graphical notation is used, and to the right, a textual syntax within a text symbol is used. Changes in either of these views are automatically propagated to the other.

Common element properties

The following properties exist for many different kinds of model elements. They can be inspected and controlled through the Properties Editor.

Visibility

Many model elements have a visibility, which is used to determine access rights for elements outside the scope in which the element is defined. Within a scope, all elements can be accessed regardless of visibility. There are different levels of visibility:

- **Public**
  All elements that can see (access) the container of the element with public visibility can also access the element.
- **Protected**
  All elements in the same scope as the element with protected visibility and the subclasses of its container can access the element
- **Private**
  Only elements within the same scope as the element with private visibility can access the element.

- **Package**
  An element with package visibility is accessible by all elements enclosed within the same package.
- **None**
  If visibility is not specified, the element gets a default visibility according to the table below.

The default visibility of a definition is decided by its scope and type.

Scope	Visibility
`Class, Choice, Stereotype, Collaboration, Artifact`	Private
`Package`	Public
`Interface`	Public
`DataType`	Public

Note

Literals always have public visibility.
All literals and public static members of a datatype are visible outside of a datatype without a qualifier. Qualifiers are only required to resolve ambiguities, for example when two datatypes in the same scope have literals with the same name.

Virtuality

Virtuality comes into play when you have generalization between classifiers such as classes, and determines whether contained model elements of a specialized class can be redefined or not.

Virtuality only applies to elements that are contained in types (classifiers that can be specialized). If the container is specialized, the individual virtuality of each contained element controls if that element may be changed.

- **Virtual**
  If a contained element is virtual, it is allowed to redefine (change) this element when its container is specialized.

- **Redefined**
  If an element in a specialized container is redefined, it is changing the definition of the original element from the base container. The original element in the base container must be virtual.

  A redefined element is still virtual, that is if the container is specialized once more, the element may be redefined further.
- **Finalized**
  If an element in a specialized container is finalized, it is changing the definition of the original element from the base container. The original element in the base container must be virtual. Finalizing also implies prohibiting further redefinition of this element if the container is specialized once more. In this sense, finalized means “redefined but not virtual“.
- **None**
  If a contained element has no virtuality, it is not allowed to redefine (change) this element when the container is specialized.

Derived

If an element is derived, it means that its value can be calculated by means of other elements. Exactly how to specify how to perform the calculation of the value is context dependent.

A common case of derived elements are derived attributes. For these the derivation rules used when accessing the attribute can be specified using accessor operations called ‘get’ and ‘set’.

Example 13: Specifying derivation rules for a derived attribute

```
Integer y;
Integer / x
  get { return 5; }
  set { y = value; };
```

Other properties

- **External**
  If a definition is external, it means that it resides outside this model. The code generators supplied will not generate code for external elements. External elements can thus be seen as model representations of externally available definitions.

- **Abstract**
  If a classifier is abstract, it is not allowed to directly instantiate the classifier. If an abstract classifier is specialized, which it typically is, it is allowed to instantiate the specializing classifier (unless it too is marked as abstract).
- **Static**
  If a definition is static, all instances of the containing classifier shares the implementation for this element, that is uses the same piece of data. Hence a static definition can be used without having an instance of the classifier in which it is defined.

Parameters

Definitions that are behavioral features, such as Operation, Signal or State machine, may have parameters. The general format (used in classifier symbols and in the Properties Editor) is:

```
name:type, name2: type2
```

A parameter has a direction which specifies the direction in which data "flows" in a call to the behavior:

- **In** (default)
  Data is passed from the caller to the invoked behavior.
- **In/Out**
  Data is passed from the caller to the invoked behavior and also from the invoked behavior back to the caller.
- **Out**
  Data is passed from the invoked behavior back to the caller.
- **Return**
  Data is passed from the invoked behavior back to the caller as the return value of the call. At most one parameter may be a return parameter.

Template parameters

A template parameter is a concept for allowing flexible, context-free classifiers. Another name for template parameters is context parameters.

Elements that can be specialized or that can be instantiated (called) may have template parameters, for example classes and operations.

Template parameters are bound with actual parameter “values” either at instantiation or when the containing classifier is specialized or redefined. It is allowed to bind a subset of the template parameters at specialization. On instantiation, all template parameters must be bound.

As a general rule, whenever a template definition is referenced actual values for all its template parameters must be specified. There are two exceptions to this rule

1. If a template parameter has a default value, it is not needed to give an actual value for it. The default value will then be used.
2. In calls to a behavioral feature with template parameters it is not necessary to specify the actual template arguments if these can be deduced from the actual call arguments used in the call.

The operators `reinterpret_cast<T>` and `cast<T>` cannot be used as actual template parameters. Any template instantiation containing `reinterpret_cast<T>` or `cast<T>` operator cannot be resolved by name resolution.

Example 14: Template instantiation containing casting operator

The following example illustrates this restriction:

```
template<const Integer x>
class MyTemplate { }
enum E { L }

/* These template instantiations cannot be resolved */
MyTemplate<cast<Integer>(L)> myVar1;
MyTemplate<reinterpret_cast<Integer>(L)> myVar1;
```

Predefined names

In the provided utility package Predefined are found a number of useful datatypes, literal values and operations. The names of these entities are not reserved, but it is recommended to avoid using these names for other entities as that is likely to cause human misinterpretation.

See also

“Predefined” on page 319

Use Case Modeling

Use case modeling focuses on determining the context of a system or parts of it, often in terms of the actors that interact with it, but also on modeling the requirements of the behavior of these elements.

Use case diagram

A use case diagram illustrates a usage situation by showing the relationships between use cases and actors. A use case diagram gives a static view of dynamic aspects of systems.

Example

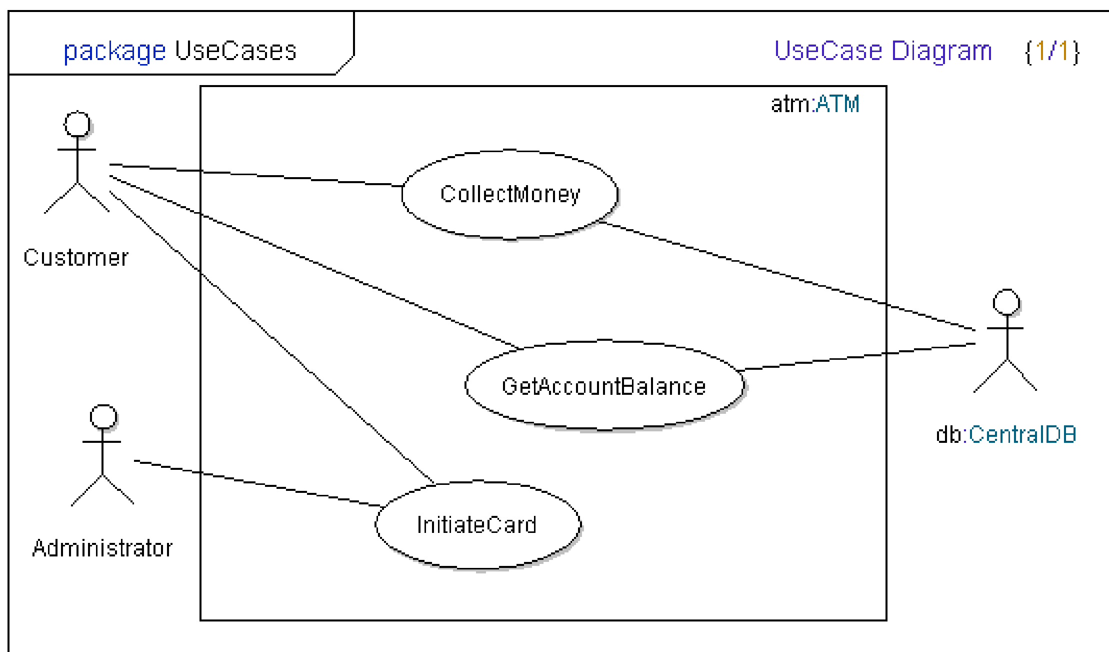


Figure 39: Use case diagram with Actors, Use Cases, a Subject and Association relationship between the Actors and the Use Cases

Model elements in use case diagrams

The following elements are found in use case diagrams

- Use cases
- Actors

- Subjects
- Dependencies
- Includes
- Extends
- Generalizations
- Association

Create a use case diagram

Use case diagrams can be included in packages, classes and collaborations.

1. Select the package (class, collaboration) in the Model View.
2. From the shortcut menu select **New** and then **Use Case diagram**.

   Use cases can then be drawn using the toolbar or you can drag use cases from your model into a use case diagram.

- To use the toolbar, first click on the use case symbol and then click in your diagram where you want to position the use case symbol.

Use cases

A use case represents a coherent unit of functionality provided by a system or parts of a system. Usually, the system is represented by a class. The functionality is often manifested in terms of communications between the system and one or more outside actors, including the behavior performed by the system.

A use case is in many ways similar to an operation, and is in fact modeled as an operation with the stereotype «use case».

Symbol

Figure 40: Use case symbol

A use case is visualized through the use case symbol in a use case diagram. It can be specified within the scope of:

- a package
- a class
- a collaboration
- an implementation

The description of a use case

The behavior of a use case can be defined by:

- an interaction
- a state machine
- an operation body
- an activity

It is also possible to describe the behavior of a use case textually. In this case, there is often some structure to the text, where the name of the use case is given, followed by its goals, preconditions and post conditions, exceptional cases, and the actual functionality in the form of actions that should be performed by the use case.

Example 15: A textual use case

```
Use case: CloseAccount
Goal: Close a user account and make sure the balance of
the account is settled
Preconditions: Customer has an open account
Postconditions: Customer has closed the account and has
paid outstanding dues
Description:
1. Check balance of account
2.a If balance is positive, pay customer
2.b If balance is negative, collect payment from
customer
3. Terminate card associated with account
4. Close account
```

Naming use cases

When naming Use Cases, it is common to use some kind of verbal description, typically a phrase which contains a verb and an object, for example “do something”. It is possible to use this name convention, in spite of the fact that names may not contain spaces, by using a quoted name:

Example 16: Quoted use case name

```
<<usecase>> void 'Open Account' ();
```

More often, the verb and the noun are written together without the white space.

Actors

An actor represents an entity that takes part in a use case, for example to initiate the functionality or as a resource for information needed by the use case.

Symbol

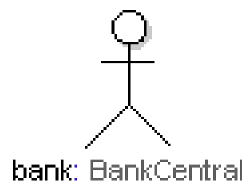


Figure 41: Actor symbol

An actor is visualized using a stick figure symbol in a use case diagram. Actors are connected to use cases using Association.

The role of an actor

In a use case diagram, the focus is on showing the relationships between actors and use cases. An actor is an entity that is involved in use cases, most often in the context of one or more Subjects. An actor is external to the subject for which the use case is defined, and can be human users, external hardware devices, or other subjects. An actor is not necessarily one single physical entity, but can for example be an entire computer network.

In different use cases, there can be different actors representing the same physical entity, but with a different role. An actor may also represent different physical entities in different use cases.

The actor is either a reference to a part or an instance of a class.

In a use case diagram the focus is on showing the relationships between actors and use cases. However, sometimes it is also beneficial to focus on the type-like aspects of an actor. For example to show how actors relate to each other using inheritance or show some properties of the actor. This is shown in class diagrams where actors are visualized as a class symbol with the «actor» stereotype.

The Actor symbol visualizes stereotypes applied to the actor and the class that the actor references (only if no stereotype is applied to the actor)

Subjects

A subject defines the system boundary for a set of use cases. A subject can represent a system, subsystem or class. The subject is either a reference to a part or an instance of a class.

Subject corresponds to the System Boundary of use cases in UML 1.X.

Symbol

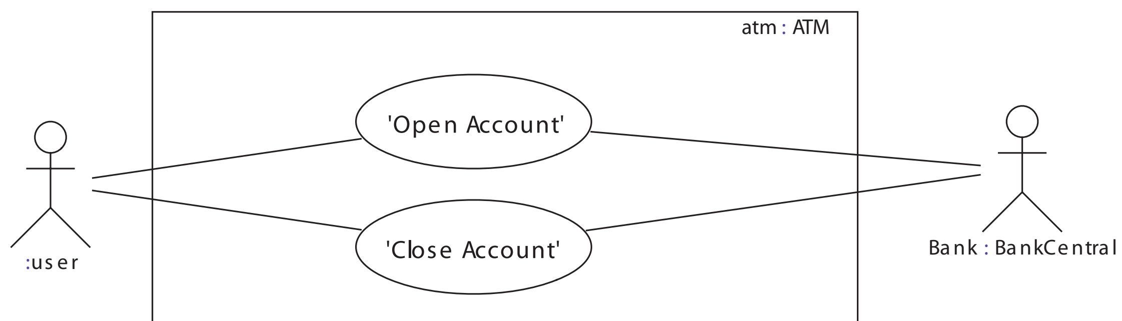


Figure 42: Subject symbol (ATM)

Use cases can be enclosed inside a subject symbol. The subject symbol is drawn around a set of use cases that represent the behavior of for example an active class. A name and the class type can be written in the upper right corner of a subject symbol.

A hatched background color can be assigned to the subject symbol.

Relationships

The following relationships can be used within a collaboration or a use case diagram:

Association

The association relationship is used between an actor and a use case and indicates that an actor participates in that use case. Reversely, the use case is performed by the actor. One actor may participate in several use cases and one use case may have several participating actors. Association text is informal.

Includes

The include relationship is used between different use cases to indicate that one use case is part of another use case. This provides a mechanism to split large use cases into smaller ones. The behavior of the including use case is typically not meaningful by itself, but is dependent on the included use cases.

Extends

The extend relationship is used between different use cases to indicate how and when a use case should be inserted into an extended use case. The extended use case should be complete by itself; the extensions typically describe supplementary functionality to be addressed under certain conditions.

Dependencies

Dependencies may be specified between use cases or between actors. A dependency does not give any indication about how the entities are related.

When a dependency is created between two use cases it will implicitly become an include relationship.

Generalizations

A generalization can be specified between use cases; one use case may specialize a more general use case. For actors that are associated with classes a generalization can be specified. Generalization text is informal.

See also

"Relationships in UML" on page 306

Scenario Modeling

Scenario modeling focuses on describing scenarios of system or subsystem usage. These scenarios are described as sequences of events that occur on lifelines.

When describing message interactions in increasing detail during this modeling activity, a clearer view emerges of how the responsibilities are divided between components of the system, but also of the borderline between the system and the external actors that interact with it.

The scenario modeling activity often takes place rather early in the analysis activity, but can of course also continue, with greater precision, in the design activity. The scenarios that are produced are specifications of the dynamic interfaces of the system and system components. They often have a twofold purpose:

- as a basis for the behavior modeling of components
- as a basis for test cases.

In UML scenarios are modeled using Interactions and the events are shown in Sequence diagrams as described in this section. Interaction overview diagrams are used to control and coordinate individual interactions.

Scenario modeling is very often done as part of a use case analysis. For each use case an interaction is created describing the behavior associated with the use case and a sequence diagram is used to visualize the interaction.

Sequence diagram

Description

A Sequence diagram describes an Interaction, visualizing the message interchange between lifelines, but also other event occurrences.

Example

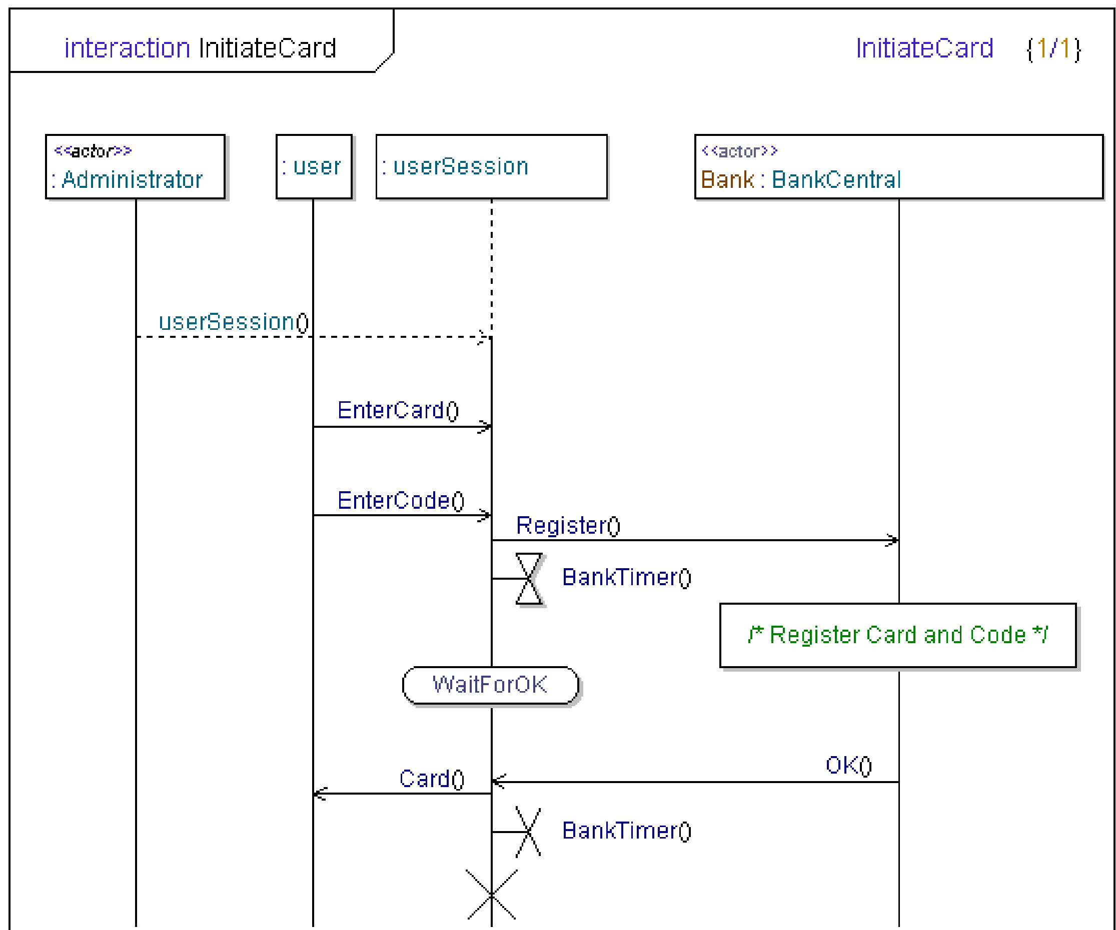


Figure 43: Sequence diagram

Model elements in sequence diagrams

The following model elements can be found in sequence diagrams:

- Lifeline
- Message
- Action
- State
- Interaction reference
- Timer event
- Time specification line
- Create

- Destroy
- Inline Frame
- Co-region
- Continuation
- Method call

Create a sequence diagram

A sequence diagram is a graphical description of the implementation of an Interaction. When creating a sequence diagram for example in a package it will automatically be encapsulated in an Interaction with its implementation.

It is however also possible to give a sequence diagram as implementation of other behaviors, such as operations and use cases. To accomplish this the sequence diagram can be created directly inside the behavior itself.

There are also options related to sequence diagrams:

- Message separation
- Lifeline separation

The lifeline ruler section

When the header is not visible on screen at its normal position in the diagram because the header is scrolled out of sight in the vertical direction, then the header is instead visible in the lifeline ruler section.

Interaction

An interaction is a description of the behavior of a use case, operation or other entity that can have a behavior. In an interaction the focus is on information exchange between parts. It is typically described by a Sequence diagram.

The semantics of an interaction is defined by the set of traces that can be derived from the interaction. A trace is a sequence of event occurrences. This sequence is not necessarily totally ordered. The traces may describe both possible and impossible scenarios.

Interactions can be referenced from within other interactions, thus allowing reuse. This is normally done by the Interaction reference symbol that reference another use case or operation that contains an interaction as its behavior definition. It is also possible to refer to an interaction by a Lifeline decomposition.

Interactions are typically used in two different ways:

- to specify the externally visible behavior of a system and its components
- to describe a trace of an execution of a system

See also

"Sequence diagram" on page 162

"Use cases" on page 157

Interaction reference

An Interaction Reference is used to represent references of interactions in sequence diagrams. The referenced interaction is usually described in a sequence diagram of its own.The name used in the Interaction Reference is the name of the use case or operation that contains the interaction, not the name of the interaction itself.

The interaction reference is useful in two ways:

- It can be used as an encapsulation mechanism to hide detailed interactions while focusing on the important message interchange
- It enables reuse of interaction descriptions.

Symbol

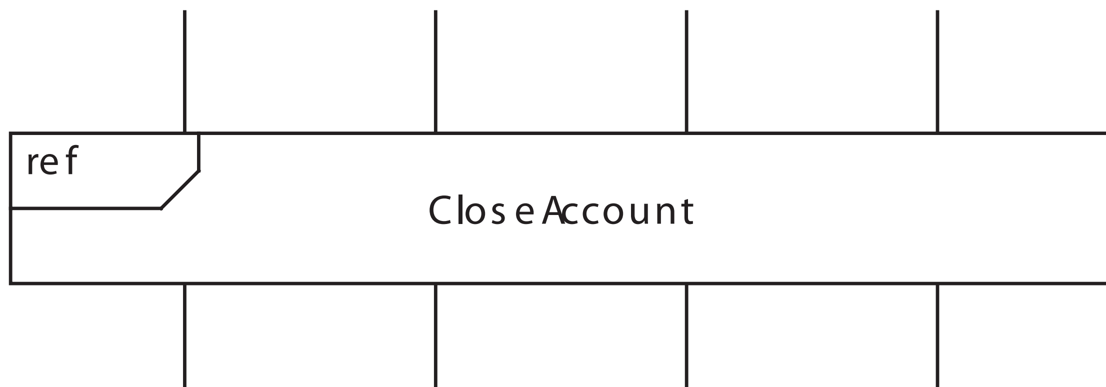


Figure 44: Interaction reference

Syntax

The Interaction Reference symbol contains a name, referring to a use case, operation or other entity that can contain an interaction.

See also

"Interaction" on page 164

"Use cases" on page 157

"Sequence diagram" on page 162

"Attach/Detach from lifeline" on page 167

Lifeline

A Lifeline represents an individual participant in an interaction. While Parts and structural features may have Multiplicity greater than 1, lifelines represent only one interacting entity. If a lifeline represents a part that has greater multiplicity than 1, a specific instance must be chosen through indexing.

Symbol

The lifeline symbol consists of a head and an axis. If the lifeline has not been created yet, the axis is drawn by a dashed line. When a lifeline is destroyed (the instance is terminated), the axis is again drawn with a dashed line.

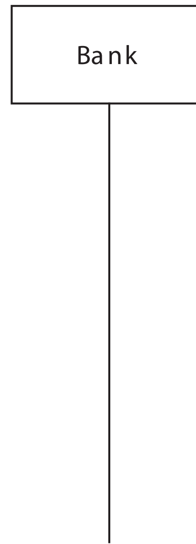


Figure 45: Lifeline symbol

Create a lifeline

To create a lifeline you can either:

- Use the Diagram element toolbar and select a lifeline symbol. Place it in your diagram. Type in the appropriate information in the heading, for example **Partname** or **Classname**.
- Drag a class symbol from your model into the sequence diagram to create a lifeline representing this class. The lifeline will represent any instance of the class, text in heading symbol reading **Classname**.
- Drag a part from your model to create a lifeline representing this part. The lifeline will represent the instance of the part, text in head reading **Partname** (or **Qualifier::Partname** if scope qualifier is necessary).

Attach/Detach from lifeline

When a symbol that can span over several life lines (such as the inline frame symbol) is selected there is a button next to each lifeline covered by the symbol. This button attaches or detaches the symbol from the lifeline the button is next to, depending on its current state. If the symbol is currently attached, the button display a minus sign (-). Otherwise the button display a plus sign (+).

Ordering of events

The order of event occurrences along a Lifeline is significant, denoting the order in which these event occurrences will occur. The absolute distances between the event occurrences on the Lifeline are, however, irrelevant for the semantics.

Although the order of events is strictly specified on one lifeline, there is generally no ordering between events on different lifelines. It is possible to describe a distributed system using an interaction or a sequence diagram, so that each asynchronous component is described by its own lifeline.

The only mechanism to order events on different lifelines in the general case is to synchronize them by message sending. The ordering mechanism of sequence diagrams is often called **partial ordering**; they do not describe a total order, nor a complete disorder.

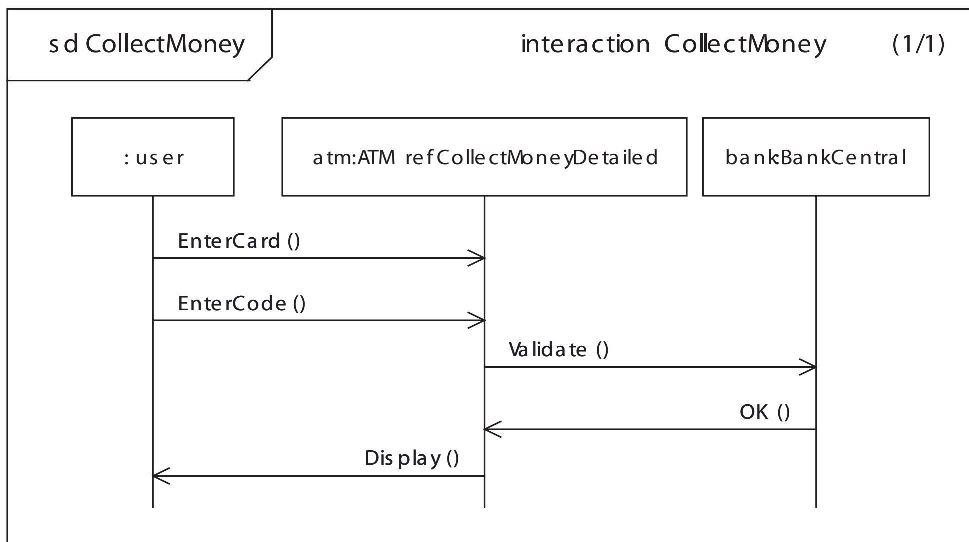
For systems that by nature are not asynchronous or distributed (normal programs, without threading), it is of course possible to have a stricter order interpretation than the general, asynchronous case.

Lifeline decomposition

A lifeline can refer to a composite, that is to an object with parts. This is a way to reduce complexity of interactions and focus on the most important message interchange.

In some situations, though, you also want to see the internal communication, that is to say the detailed message interactions between the parts of a composite object. The decomposition mechanism offers this duality: it is possible to have two descriptions of the same behavior: one high-level description and one detailed. The detailed interaction is referenced in the lifeline heading and is defined in a separate use case or operation, as the example in Figure 46 on page 169 shows.

Decomposition example



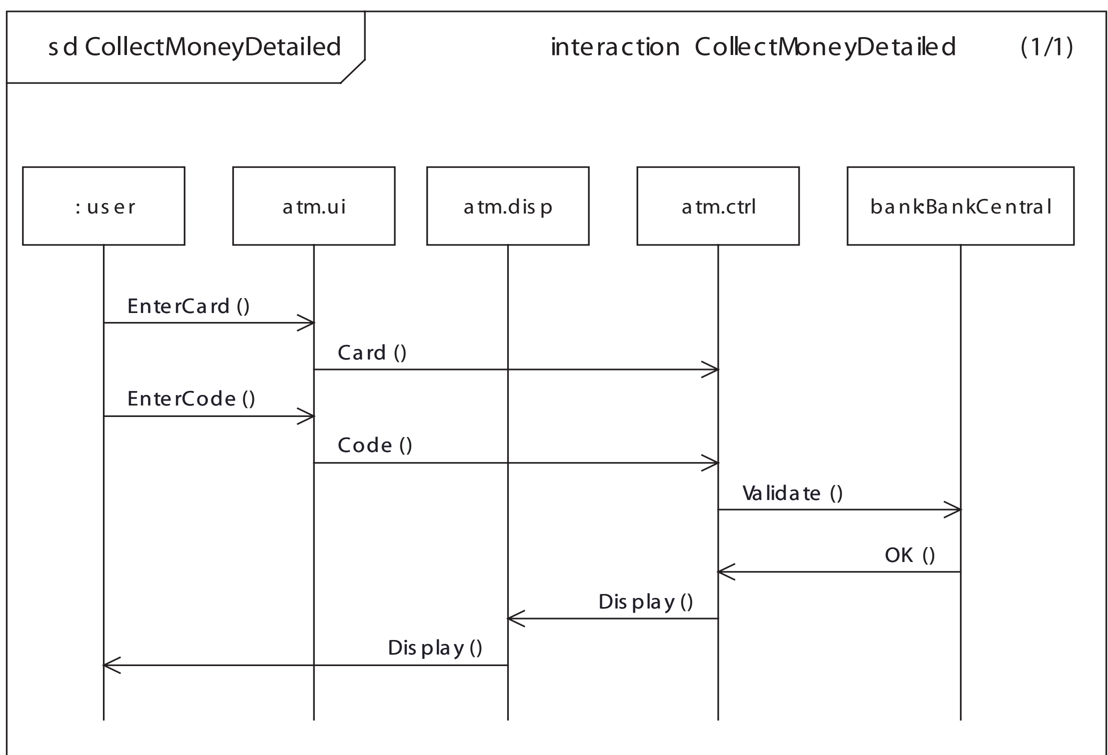


Figure 46: Example of lifeline decomposition

Syntax

The following syntax is accepted in a Lifeline:

`Bank`
An instance name, referring to a Part, Port, Attribute or Subject.

`Bank: BankCentral`
An instance name and a type name, referring to a Class.

`:BankCentral`
A type name, referring to a Class.

`atm[3]`
An instance name with a selector expression to reduce the Multiplicity to 1 instance.

`atm.Display`
An instance name with an attribute referring to a part.

`atm ref OpenAccountDetailed`
An instance name and a lifeline decomposition, referring to a Use Case or an Operation, described in a separate interaction and Sequence diagram.

`atm[2].Display:ATM ref CloseAccountDetailed`
An instance name with selector, part, type and lifeline decomposition.

Message

A message is an occurrence of a Signal, a method call, or a method reply. It normally has two events; one send event (out) on the sending lifeline and one receive event (in) on the receiving lifeline. A message can be horizontal or it can have a slope, but the receive event should not appear above the send event in the diagram.

Symbol

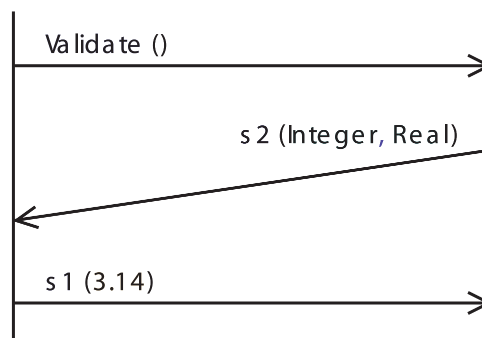


Figure 47: Messages

It may take time to send a message and pass it to the receiving side, but a slope does not have that interpretation. Correspondingly, a horizontal message is not necessarily directly delivered at the receiver.

Because of the relationship between signals and messages, the name of a message must always refer to a visible Signal in the model. If the signal has parameters, the message should have actual parameter expressions.

Creating a message

Messages have three associated text fields, one for signal name and parameters and two for Gate names.

There are two different methods for placing messages allowing you to create messages in a simple and unrestricted way.

Traditional: Click on **Message line** in the Diagram element toolbar. Click on the sender lifeline, then on the receiver lifeline.

Single-click: Click on **Message line** in the Diagram element toolbar. When you **click and hold** between lifelines the message will attach to the lifeline to the left. When you **release** the lifeline will attach to the lifeline to the right. You can now do several things to create the message you aim for:

- Release to attach the receive point to the lifeline to the right
- Drag to cross any intersecting lifelines, release close to the left of the lifeline you want to receive the message.
- SHIFT + click to send the message from right-to-left.

Summary of how different line types can be created:

- **Normal message**: Select message in the toolbar. Click between lifelines for left-to-right direction. SHIFT + click for right-to-left direction. Click and hold, then drag to cross intersecting lifelines, release to attach next lifeline in message direction.
- **Message to self**: Select **Message line** in the element toolbar. Then click twice on the same lifeline.

Reference existing signals when you draw a message

1. Click on **Message line** in the Diagram element toolbar.
2. Point and click on the lifeline that the message should go from.

3. Point and right-click close to the lifeline that the message should go to. Point to Reference existing on the shortcut menu and select the signal from the list.

Reference existing will display the signals visible in the scope.The signals that are shown are computed as follows:

- If the target lifeline has a type, then the signals/operations shown in the list are all signals that can be received by this type taking into account signals in realized interfaces, signals defined in the class itself etc.
- If the source lifeline has a type, then the signals/operations shown in the list are all signals that can be sent by this type, taking into account all required interfaces.
- If the source and target lifelines do not have types, but the target lifeline has a selector then the signals/operations shown in the list are all signals that can be received by the type of the selector taking into account signals in realized interfaces, signals defined in the class itself etc.
- If the source and target lifelines do not have types, but the source lifeline has a selector then the signals/operations shown in the list are all signals that can be sent by this type, taking into account all required interfaces that exist.
- If none of the above conditions apply, the signals/operations shown in the list are all signals visible from the lifeline itself.
- If none of the above mentions conditions apply, the signals/operations shown in the list are all signals visible from the lifeline itself.

A message can in some cases be drawn so it is only connected to one lifeline. This is particularly useful when using sequence diagrams for tracing. There are four message types that can be identified:

- **New**, the message is sent but not yet received. The message is connected to its sender.
- **Lost**, the message is sent but will not be received. The message is connected to its sender and a small circle is drawn at the message arrowhead.
- **Old**, the message is received but the sender is so far unspecified. The message is connected to its receiver.
- **Found**, the message is received but the sender is unknown. The message is connected to its receiver and origins from a small circle.

Use the property editor to mark a message as Lost or Found.

A message line can also be auto created in the following ways:

- SHIFT + click on the message in the symbol element toolbar when a lifeline is selected creates a new message. The new message is placed last on the lifeline, but before any destroy lifeline symbol.
- SHIFT + click on the message in the symbol element toolbar when two lifelines are selected creates a normal message between the lifelines. The normal message is horizontal, placed last on the lifelines (before any destroy lifeline symbol), and has a left to right direction.
- SHIFT + click on the message in the symbol element toolbar when a message is selected creates a normal message immediately below the selected message. The normal message is connected to the same lifelines as the selected message and has the same direction.

Note

When you edit a message, you will see all parameters for that message, independently of whether parameters are shown or not. When you leave editing mode, the message text will go back to showing parameters or not in the same way as other messages do

Toggle parameters

Hides or shows all message parameters in the diagram. As default, parameters are shown. When you enter edit mode for a message text, the parameters will be shown for all messages.

Incomplete message

A message may be incomplete in the sense that only one of its events is specified. If the receive event (in) is missing, it is a Lost message. If the send event (out) is missing, it is a Found message.

Lost message

A lost message is a message where the send event is known, but there is no receive event. This can be used to describe the case when a message never reaches its destination.

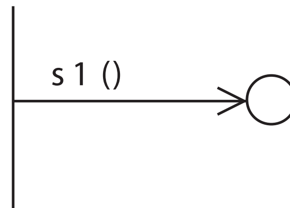


Figure 48: Lost Message

Found message

A found message is a message where the receive event is known, but there is no (known) send event. This can be used to model the case when the origin of the message is outside the scope of the description. It can also be used to avoid over-specification: when several lifelines can be the sender, but which one is not relevant to the scenario.

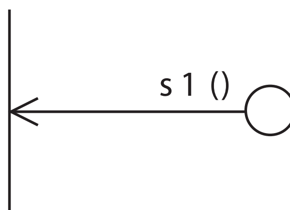


Figure 49: Found Message

Copying a message

There are two different methods of copying messages. The first method always keeps message sender and receiver.

CTRL + Drag: Press CTRL key and hold it. Then click and hold a message you want to be copied. Drag the message and drop it to the new position. Release CTRL key.

The other method allows setting different sender and receiver.

Copy and Paste commands: Open the shortcut menu for a message you want to be copied by right-clicking on this line. Choose Copy from the menu. Open the shortcut menu by right-clicking in a place in the diagram where the new message should be inserted. Choose Paste from the menu. You can also

use CTRL + C and CTRL + V shortcuts for performing Copy and Paste commands, but note that the position of the new message is defined by the point in the diagram where you clicked last before pasting.

There are the following options.

- If the point you clicked is between two lifelines, then the new message will be inserted between these lifelines.
- If the point you clicked is either before the first lifeline or after the last lifeline, then the sender and receiver will be kept as in source message.

Timer event

A timer is normally described by two distinct events in an interaction. The first event is the timer set, the second event is either a time-out or a reset.

A timer needs to be declared, before it can be used (just as messages need the corresponding signals or operations to be declared). Timers are declared with the Timer symbol in class diagrams.

The timer event symbols have one text field, for name and parameters.

Timer set

The set event creates a timer instance, which now is active. The timer set event maps to the Timer set action.

Timer reset

The reset event cancels an active timer. The timer reset event maps to the Timer reset action.

Timer timeout

The timeout event occurs when the timer duration has passed and the timer signal has been received and consumed by a state machine. The timeout event maps to a timer signal consumption.

Symbols

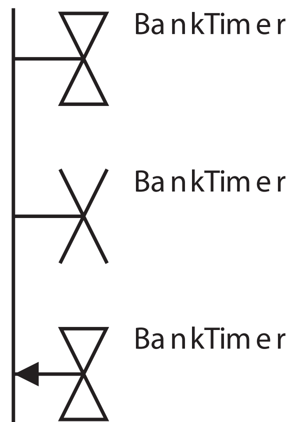


Figure 50: Timer set, reset and timeout symbols

See also

"Timer" on page 223

"Timer set action" on page 284

"Timer reset action" on page 284

Time specification line

The Time specification line is used to create an **Absolute time** line, a **Relative time** line and a **General ordering line**.

Absolute time line

An absolute time line can be added to the left or right of a lifeline, specifying an absolute time or a range, "`{<Time>}`". The line can be moved up or down along the lifeline. An absolute time line is created by clicking in the symbol palette on **Time specification line**, and by drawing a line connected to a lifeline in only one end.

Relative time line

A relative time line is created by clicking in the symbol palette on **Time specification line**, and by drawing a line connected to the same lifeline in both ends.

A specific time duration observation, "{<Duration>}", or a time duration constraint, "{<Duration>..<Duration>}", can be specified in the text field.

A relative time line has an upper border, a lower border and a duration. The line is always drawn on the right side of a lifeline, but can be moved to the left side. The borders can be moved up or down along the lifeline.

In most cases, the start and stop events of a relative time line are connected to other events of the lifeline. For instance:

- The arrival of a message
- The sending of a message
- The start/top of a reference symbol
- The end/bottom of a reference symbol

It is allowed to place a Relative time line start or stop at a place where the event is not connected to other events.

General ordering line

The general ordering line is a time specification line going between two lifelines. Create a general ordering line by clicking on **Time specification line** in the symbol palette, and by drawing the line between two lifelines.

The general ordering line is used to specify the order of events on different lifelines without using message lines. It is visualized as a dashed line, with a filled arrow in the middle. No text is normally associated with the line, but it is possible to associate a specific duration, "{<Duration>}", or a range, "{<Duration>..<Duration>}" with the line.

Symbols

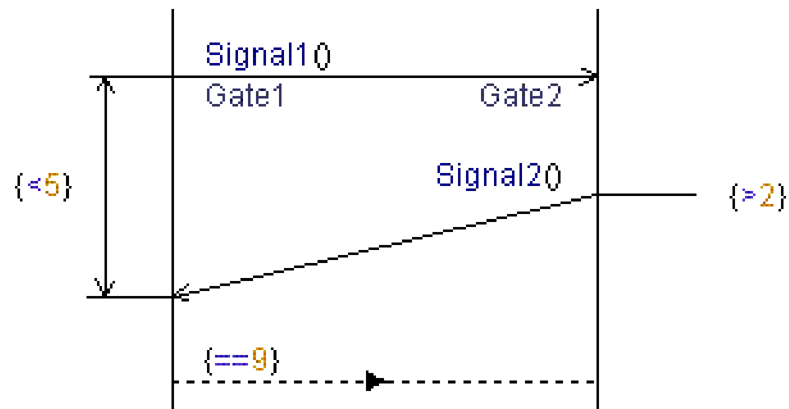


Figure 51: Absolute time line, Relative time line and General ordering line

State

The state symbol is used to indicate that the instance described by the lifeline is in a specific state.

Symbol

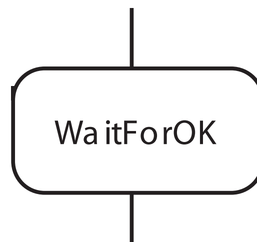


Figure 52: State

In scenario specifications, the use of the state is mostly done to highlight a certain state. Normally, you do not indicate all passed states along the life-line.

The State will bind to a model element if the state machine of the active class that the lifeline references has a state with the same name.

For traces, though, each state symbol maps to a specific Nextstate occurrence in a state machine transition. This is true if the lifeline object only has one main state machine; for active objects with parts, that is active objects that have several state machines, a simple mapping is not feasible.

Action

The action symbol is used to express events that occur in a lifeline. It corresponds to an action symbol in a State machine. Informal statements must be written as comments.

Symbol

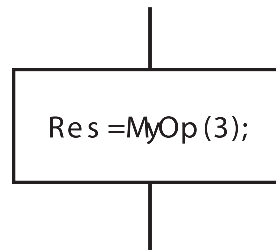


Figure 53: Action

The allowed textual syntax in the Action is the same as for the Action (task) symbol in state machine diagrams.

Create

The Create event corresponds to the New operation applied on active classes.

The lifeline that is created is dashed before the reception of the create event, meaning that it has yet to be created. The name on the create line is the name of the class corresponding to the lifeline.

Symbol

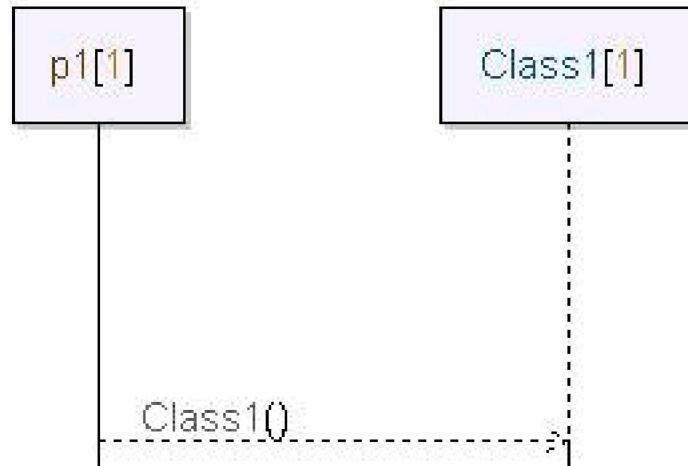


Figure 54: Create line

Creating a Create line

When drawing a lifeline representing a dynamic instance of a class it is possible to draw the create event. This is done with the Create line button in the diagram element toolbar and is handled much like a message. The name of a create line is the name of the class corresponding to the lifeline. It refers to a constructor operation for the class. A create line have three associated text fields, one for the constructor operation name and parameters and two for Gate names. Formal parameters can be added similar to adding of operation parameters to a method call line.

Binding of a constructor

Binding of a constructor initializer reference to a base class constructor fails if the base class constructor is called initialize. The recommendation is to name it to the same name as the class.

Example 17: Constructor initialize that does not bind

```
class AutoDispatchableClass : tor::DispatchableClass {
  initialize(tor::DispatchableClass d) {
    d.addToCurrentDispatcher(this);
    init();
    'start'();
  }
}
```

```
class MyClass : AutoDispatchableClass {
  initialize(tor::DispatchableClass d):
AutoDispatchableClass(d) {  }
}
```

The `AutoDispatchableClass` reference does not bind.

Destroy

The Destroy event represents a termination of the instance. It corresponds to the Stop action in a State machine. Events can not occur on a lifeline after the destroy event.

Symbol

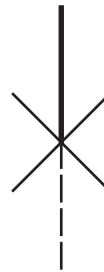


Figure 55: Destroy

Inline Frame

The inline frame symbol provides a way to group messages that should be treated similarly within an interaction. This means that it is possible to express different kinds of variations in a single diagram rather than having to create a new diagram for each possible variation.

Symbol

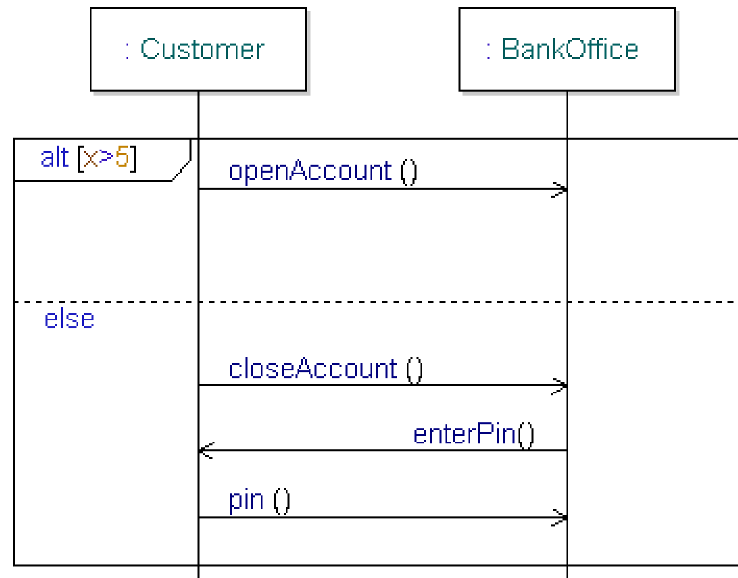


Figure 56: Inline frame

It is possible to have inline frame symbols inside other inline frame symbols. When a second inline frame symbol is added at the same height as an existing inline frame symbol, this will place a new inline frame symbol inside the existing inline frame symbol.

An inline frame symbol can have one or several inline frame sections. The default inline frame symbol has one inline frame section. Inline frame separator lines divide the inline frame symbol into several inline frame sections. Each inline frame separator line has a constraint text.

An inline frame separator line is created with a line handle that appears when an inline frame symbol is single-selected.

You can drag the inline frame separator line up or down within the symbol, but it is not possible to cross another separator line connected to the same inline frame symbol. An inline frame separator line can be deleted by selecting the separator line and pressing the delete key.

When a section is removed, objects in that section will also be removed as they are a part of the removed separator.

If the inline frame symbol is deleted, the contained objects are deleted with it.

The inline frame symbol has one text, which is a combination of:

- Operator keywords: Examples: `seq` (default keyword), `alt`, `else`, `loop`, `assert`.
- Constraint text. Examples: “[a<3]”, “`else`”

It is possible to assign a background color to the inline frame symbol. The color will be shown as diagonal colored lines in the background of the symbol.

Variations

There are several different possible variations, where the frame is sometimes split to express alternative groups of messages. The available variations are the following:

- `alt`**:** Expresses one branch of an alternative, or a decision. The frame can be split into multiple operands, and each operand can be associated with a condition. Only the alternative branch whose condition is evaluated to true will be chosen. Exactly one of the branches may be an else branch.
- `opt`**:** Expresses that the grouped messages are optional, meaning they do not have to happen. An optional frame cannot be split. It is possible to associate the optional frame with a condition, in which case it behaves just like an alternative, where the second choice is empty.
- `loop`**:** Expresses that a set of messages should be repeated a number of times. A loop frame cannot be split. The number of iterations is given using a minimum value and a maximum value of the format “`loop(min, max)`”. It is possible to give “`max`” the value “*” which then denotes an infinite loop.
- `par`**:** Expresses that the messages of multiple operands can be interleaved with each other, or occur in parallel, but the ordering constraint within each operand must still be preserved. To be meaningful, a parallel frame must be split.
- `seq`**:** This represents the normal semantics of sequence diagrams, where each lifeline is independent of other lifelines. Weak sequencing is primarily used to override strict sequencing.

- `strict`: Expresses that the messages enclosed either in the sequence diagram or the combined fragment should have strict sequencing, that is to say that the vertical position in the diagram is equivalent to the order in which things will happen. Compare this with weak sequencing, which is the default for a sequence diagram, where each lifeline has its own timeline. When using strict sequencing, you can think of this as having a common global time for the involved lifelines.
- `neg`: Expresses that the set of messages represented are invalid.
- `critical`: Expresses that the enclosed messages cannot be interleaved by other inline frames. This can for example be used within a parallel frame to override the implied interleaving for a set of messages.
- `break`: Expresses an exceptional occurrence that interrupts the rest of the sequence diagram, and instead performs the set of messages enclosed by the break frame. A break frame cannot be split.
- `assert`: Expresses that the sequences expressed by the assert frame are the only valid ones, and that all other sequences are invalid. An assert frame cannot be split.
- `ignore`: Expresses that a given set of messages are insignificant and should not be shown within the frame. This gives a way to only show the most important messages of an interaction. The format is ignore {<list_of_messages>}. The converse operation is consider. An ignore frame cannot be split.
- `consider`: Expresses that a given set of messages are significant within the frame, and that messages not shown are thus insignificant. The format is consider {<list_of_messages>}. The converse operation is ignore. A consider frame cannot be split.

See also

"Attach/Detach from lifeline" on page 167

Co-region

Symbol and lines can be connected to the lifeline in the normal way also inside the co-region symbol. When symbols are connected inside the co-region symbol, they are always covering the co-region symbol.

A co-region is used to indicate that the order in which elements on a single lifeline is insignificant.

Symbol

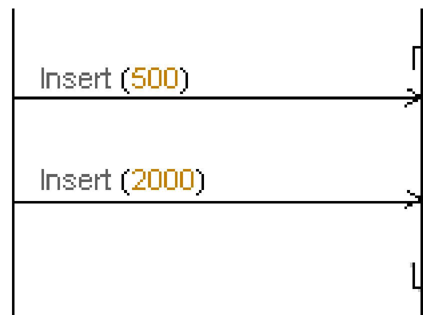


Figure 57: Co-region

Continuation

Continuations are only used in alternative inline frames, and acts as labels that decide how to continue from one part of a sequence to another. An alternative or interaction that ends with a continuation can only be continued in an interaction or alternative that starts with the same continuation.

Symbol

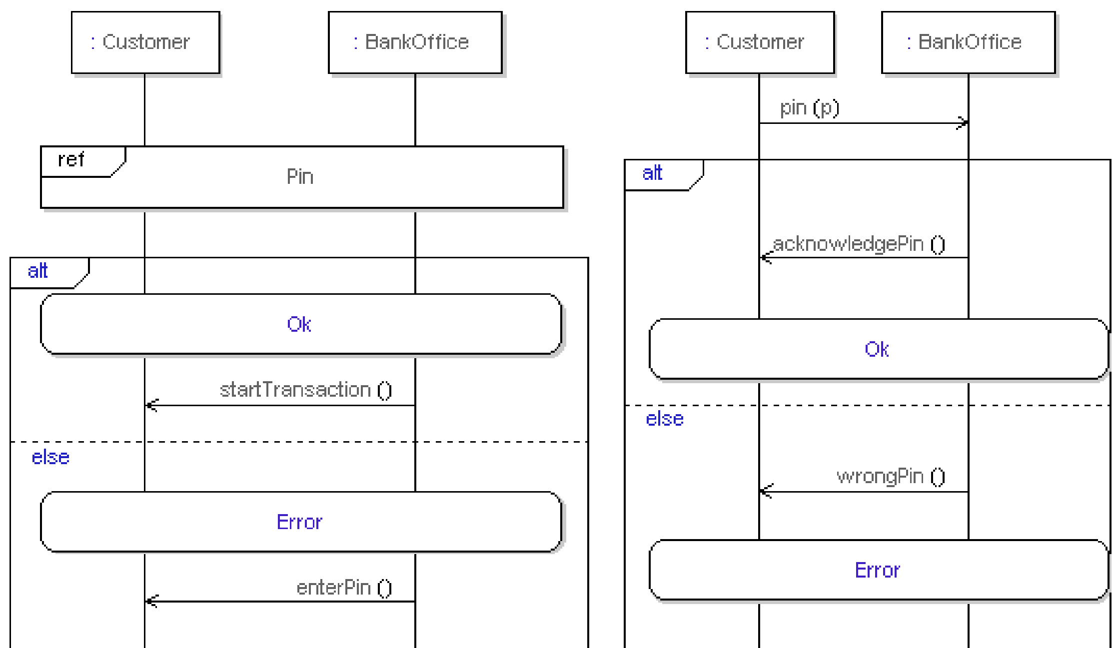


Figure 58: Continuations

The continuation symbol looks like the state symbol, but may span multiple lifelines.

The symbol contains a text field, located in the center of the symbol. The entered text is not parsed, just saved in the symbol.

It is not possible to place symbols and lines inside the continuation symbol.

See also

“Attach/Detach from lifeline” on page 167

Method call

A method call is similar to a message, but is always synchronous. This means that it will always be associated with a method reply. Method calls are used to model for example how operations are invoked between different classes.

Symbol

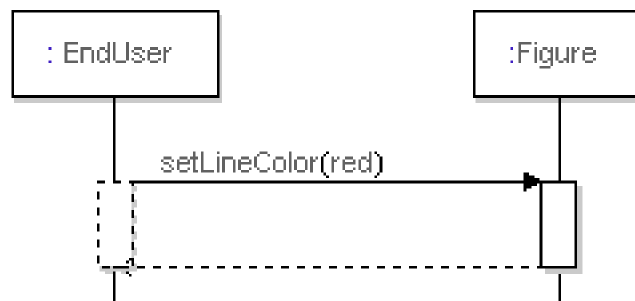


Figure 59: Method call and reply

A method call results in four graphical elements: a call, which is a solid arrow, a reply, which is a dashed arrow, an activation area, and a suspension area. The suspension area is a dashed rectangle on the caller lifeline, while the activation is a solid rectangle on the called lifeline.

The call message and reply message have three associated text fields each, one for operation name and parameters and two for Gate names.

To draw a complete method call:

1. Click on **Method call** in the Diagram element toolbar.
2. Place the **call message** start event on the lifeline that the method call should origin from, drag it to the receiver.
3. Type in operation name and parameter information or drag an operation from the model onto the message.
4. Edit the operation parameter type information in the call message and reply message name fields.

The main text of the reply message should normally refer to the same method as in the call message. The parameters may be different, for instance when the method has assigned values to out parameters, and there may also be a return value. It is allowed to only give a return value for the reply line: ": <value>".

Deleting a method call or reply line, connected suspension and activation area symbols are also deleted.

Deleting a suspension or activation area symbol, only the symbol is deleted, not any connected method call or reply lines.

When you drag a call or reply message the Method call symbol will be re-sized in the corresponding direction.

Gate names

With the shortcut menu choice **Add/Remove Gate text**, gate names can be added to a message, method call or create event. The two gate name texts are placed below the line. When a gate text is activated, the gate gets a default name, which can be edited.

Activation and suspension

The lifeline from which the method call originates is suspended while the re-ceiver is busy executing. This means that it is not doing anything but waiting for a reply. The lifeline that receives the method call becomes activated while it is executing the method that is invoked. Once the reply has been sent back to the caller, both the activation and suspension areas are closed

Appearance and filtered delete

Compress Layout

The Compress Layout button will compress the distance between messages and lifelines to be as defined in the tool **Options** for sequence diagrams.

When the Compress Layout button is pressed the lifelines are compressed and lined up by moving lifelines in the horizontal direction.

When the Compress Layout button is pressed together with the SHIFT button the lifelines will be compressed as described above, and objects on lifelines are also compressed, by moving these objects up or down along the lifelines.

When you press and hold CTRL and press **Compress layout** the lifelines are reordered to have the lifeline with the first event (for example a signal sending) to the left in the diagram.

When you press and hold SHIFT + CTRL and press **Compress layout** the lifelines and objects on lifelines are compressed as described above, and lifelines are reordered to have the lifeline with the first event (for example a signal sending) to the left in the diagram.

Delete selected signals

Deletes the selected messages. This command will also delete messages using the same signals as the selected messages. Can also be used to delete other objects:

This command will delete all <X>, when <X> is selected.

<X> is one of:

- create line
- state symbol
- timer symbol (set, reset and time-out)
- time specification line (absolute time, relative time, general ordering line)
- method call (call line, activation symbol, reply line, suspension symbol)
- action symbol
- destroy symbol
- reference symbol

- inline frame symbol
- continuation symbol
- text symbol
- comment symbol

Keep selected signals

If you press SHIFT and at point to Delete selected signals on the toolbar, the command will reverse the filtering effect: Only those messages that are selected, and those messages using the same signals as those messages that are selected, will remain in the sequence diagram. For other objects, there are the following rules:

This command will delete all <X>, if there is no selected <X> (<X> is defined in Delete selected signals).

Make space

This command will make space below the selected symbol or line. Press SHIFT and point to **Make space** in the toolbar to remove space below the selected symbol or line.

Interaction overview diagram

Interaction overview diagram is a form of Activity Diagram that focuses on the control flow between Interactions.

Interaction references in interaction overview diagrams can both define and reference operations/activities. Interaction reference is used instead of Action Node node and Object Node. An Activity edge and control constructs such as Decision, Fork and Activity Final nodes are the same as in activity diagrams.

The table below lists how you can represent the most common interaction operands listed in Variations in an interaction overview diagram.

Operand	Interaction overview construct
`alt`	A Decision node matched with a corresponding Merge node.
`par`	A Fork node matched with a corresponding Join node.
`loop`	Decisions and graph cycles in the diagram.

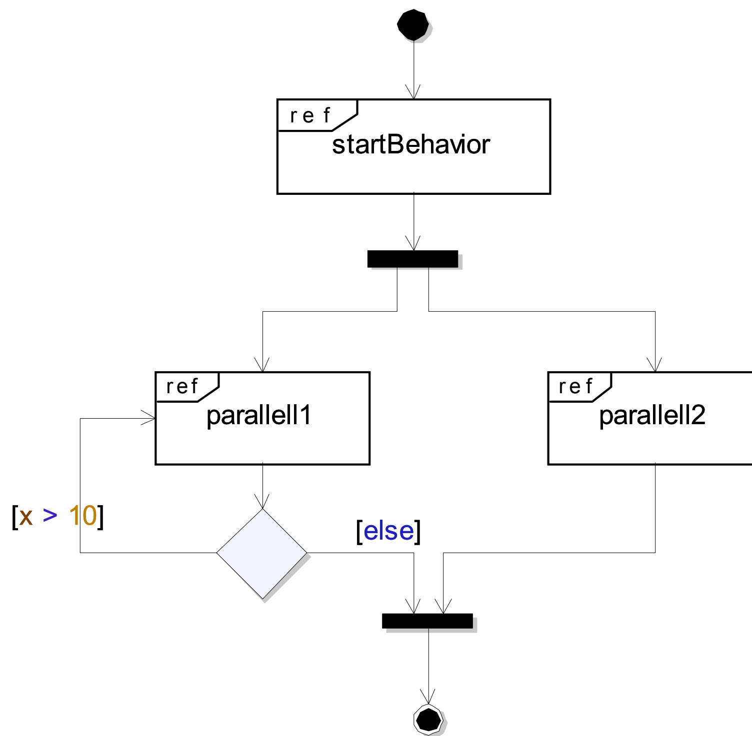


Figure 60: Interaction overview diagram

Create an interaction overview diagram

Interaction overview diagrams can be included in classes and use cases.

1. Select the class (use case) in the Model View.
2. From the shortcut menu select **New** and then **Interaction overview diagram**.

Model elements in interaction overview diagrams

The following model elements can be found in interaction overview diagrams:

- Decision
- Flow Final
- Fork
- Initial Node
- Join
- Merge
- Interaction reference, see Action Node
- Relationships

See also

"Sequence diagram" on page 162

"Activity Diagram" on page 246

Package Modeling

When larger systems are to be modeled, the Package construct is vital for organizing all the different definitions into logical and manageable groups. A good principle for the organization is to group semantically close elements that are likely to change together.

Package diagram

Package diagrams are used to visualize collections of Packages and the Relationships between them. It is used to model the breakdown of a system into logical packages and dependencies between these packages.

The package diagram contains packages and dependencies between these packages (for example Import and Access dependencies).

A Class diagram can be used for the same purpose.

Example

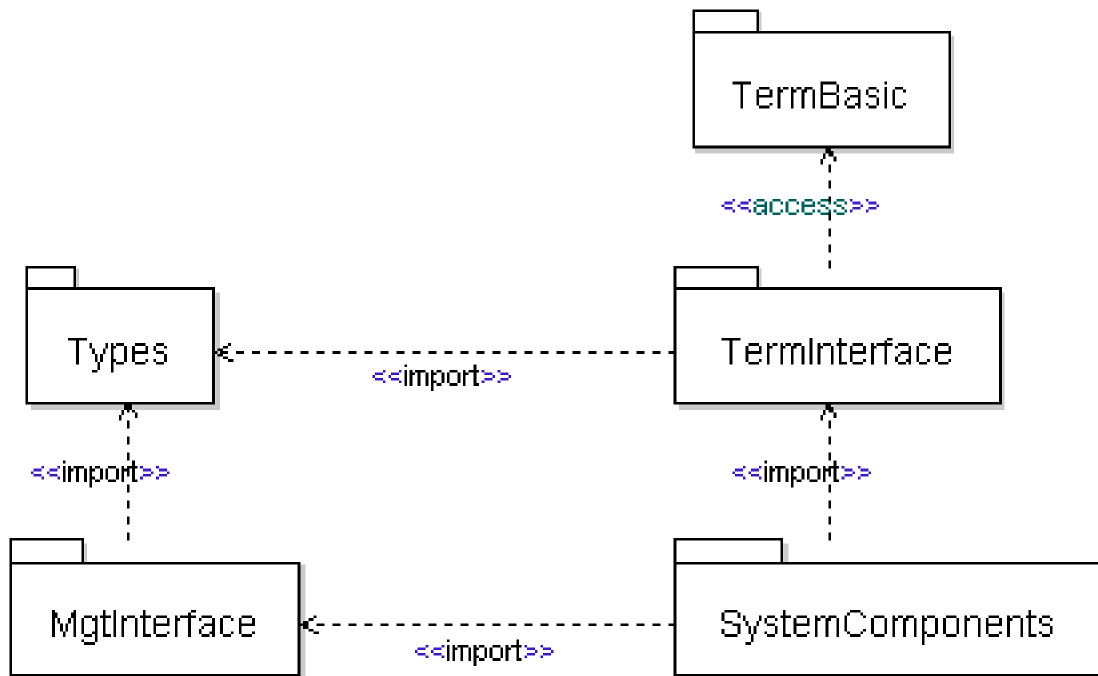


Figure 61: Packages and their relationships

Model elements in package diagrams

The following model elements can be found in package diagrams:

- Package
- Relationships

See also

"Class diagram" on page 199

Package

A Package is a mechanism for organizing elements into groups. A package provides a namespace for the grouped elements. Within the package, those elements can be referred to directly using their names, but from outside the package it is often necessary to qualify the names of the model elements.

A model normally consists of several packages that depend on each other. Understanding how packages relate to each other is critical when modeling systems of any complexity, but the larger the system becomes, the more important this activity becomes since it is often a reflection of the system architecture.

Symbol

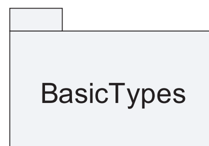


Figure 62: Package

Packages also let you control the visibility and access rights to the individual elements defined in the packages.

- Definitions (such as classes and other packages) may be collected in a Package.
- A Package may be imported or accessed by another Package.

It is possible to nest other symbols hierarchically inside a package symbol. An element created inside a package symbol will have the package as owner.

Syntax

The package symbol contains a text field with the name of the package. When the referenced package is defined in another namespace the package name is preceded by a qualifier, like in “OuterPackage::MyPackage”.

See also

“Relationships” on page 193

Relationships

The following Relationships can be used in package diagrams. These are described further in the section Relationships in UML.

- Dependency
- Containment

A dependency is often stereotyped to give a more precise meaning to the dependency. Two common stereotypes used for that purpose are the <<import>> and <<access>> stereotypes described below.

Import

Import is a special kind of Dependency that is valid in particular between Packages, but also from for example Classes or State machines to packages. Its role is to import the names of definitions from a package into the current namespace, which is usually also a package. This provides a means to avoid having to use qualifiers. Names of definitions in a package P that has been imported by another package Q are automatically included in packages that in turn import or access package Q.

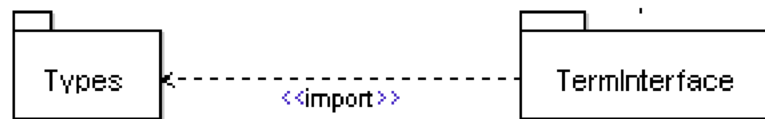


Figure 63: Import

Note

Be restrictive with using import dependencies, as the set of names that become accessible without qualifier in the importing scope can become very large. It is often better to use access dependencies. If only a small subset of definitions shall be used the use of qualifiers should also be considered. Although qualified names mean more typing, it becomes very clear for all readers of a model which definition that is used.

Access

Access is a special kind of Dependency that is valid in particular between Packages, but also from for example Classes or State machines to packages. Its role is to import the names of definitions from a package into the current namespace, which is usually also a package. This provides a means to avoid having to use qualifiers. Names of definitions in a package P that has been accessed by another package Q are not included in packages that in turn import or access package Q.

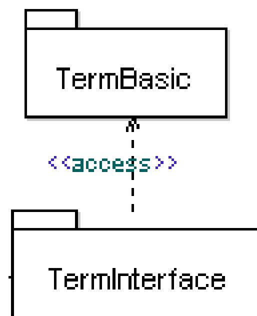


Figure 64: Access

An import is very closely related to an access; the distinction is primarily that an import is transitive, meaning that if a package is accessed or imported, you automatically also get the names of the definitions that are in turn imported by that package, but not the names of the definitions that are in turn accessed. Looking at Figure 61 on page 192, the package **TermBasic** is accessed by the package **TermInterface**, meaning that it is possible to refer to the names of definitions in **TermBasic** directly in **TermInterface**. However, these names are not directly available in package **SystemComponents**, which imports package **TermInterface**. In **SystemComponents**, it is therefore necessary to either explicitly access or import package **TermBasic** to refer to those names or explicitly qualify the names.

From an architectural point of view, accesses are preferred over imports since they force you to consider all the packages that you need, and will not bring in excess baggage by accident.

Note

It is not necessary to import or access a package to be able to reference definitions within it. As long as the definitions are public, they can be referenced using qualification, for example "`TermBasic::Xterm`*" can be used to reference the element* `Xterm` *in package* `TermBasic`*. For understandability, however, it is usually a good idea to produce a description of how packages depend on each other.*

See also

"Relationships in UML" on page 306

<<noScope>> Packages

A «noScope» package is typically used when there is a need to divide the elements of a package into more than one file. However, it can also be used as soon as there is a need to structure the contents of a package into different parts but when the package from a UML name scope point still should be viewed as one entity.

Semantically a package stereotyped by the «noScope» stereotype will be as visible as any other package in the model view. It will also work as other packages with respect to storing it in a separate file. From a semantic point of view all of the elements in the «noScope» package are considered to be part of the containing package. When referring to an element in a «noScope» package using a qualifier, the name of the «noScope» package should normally not be used as part of the qualifier. The «noScope» stereotype makes all definitions visible outside of the package without a qualifier. It is possible to use an explicit qualifier to resolve ambiguous cases.

Example 18: «noScope» package

```
package A {
    <<noScope>> package B {
        class C {

        }
    }
    C c; // <<noScope>> makes C visible
}

package A {
    <<noScope>> package B {
        class C {

        }
    }
    class C { }
    C c; // class A::C hides class B::C
}

package A {
    <<noScope>> package B1 {
        class C {

        }
    }
    <<noScope>> package B2 {
        class C {

        }
    }
```

```
    B1::C c;
/* 'C' is an ambiguous name. B1::C or B2::C must be
used. If C is used without qualifier there will be name
resolution errors. */
}
```

<<openNamespace>> Packages

In some situations it is useful to be able to incrementally define a package as the sum of a set of packages. Depending on what packages are loaded in a specific session the package will from a logical point of view have different contents.

This can in DOORS Analyst be accomplished using «openNamespace» packages. In practise it works as follows: Define two packages in the same scope (for example as model roots). Give the package the same name and stereotype both of them with the «openNamespace» stereotype. From a semantics point of view the contents of the packages will now be merged. This implies that elements from one of the packages can directly be used in the other package without qualifier and also that the used names must be unique within all of the merged packages.

It is possible to have a hierarchy of nested «openNamespace» packages. So for example if you have an «openNamespace» Top containing an «openNamespace» Sub stored in one file you can have another file that also contains an «openNamespace» Top with an «openNamespace» Sub. If you load both of these files into the same project both the contents of Top and the contents of Sub will be merged.

The most important scenario when «openNamespace» packages are used is when you have a base version of a package hierarchy that is maintained separately but want to extend this, for example with a sub-package, when using it in a specific application.

Class Modeling

Class modeling is the process of identifying the kind of objects that are part of the system being designed. This activity often takes place early in the design phase, or even in the analysis phase, typically after the objects that are part of the designed system have been identified (through use case and/or

scenario modeling). Objects that appear to share the same properties, behavior, and relationships with other objects are then grouped together and modeled as a class of objects.

Apart from identifying classes, the class modeling activity also involves the definition of these classes. This is typically done in a Class diagram. For each identified class, the following typical questions are answered:

Does the class have structure?
What parts does an instance of the class contain?

The structure of a class is described in a class diagram by means of attributes, and relationships such as generalizations and associations. A composite structure diagram can also be used to show how a class is composed.

Does the class have behavior?
Which operations are available?

The behavior of a class is perceived as operations on the class, and the signature of these operations are described in a class diagram. The same goes for other behavioral features of the class such as signals, timers or state machines.

Which relationships exist between the class and other elements?

A class may have relationships not only to other classes, but also to interfaces, datatypes, choices, etc. In the section “Relationships in UML” on page 306 you will find information on how to use them in class modeling.

Is the class active or passive?

Simply put you may say that an Active class defines dynamic event-triggered behavior and a passive class handle information. An instance of an active class has the ability to dispatch events.

Which communication ports does the class expose to its environment?

The ports of a class may be visualized in a class diagram.

Class diagram

A Class diagram gives a static view of the model and is used to describe the types of the objects in a model. These types are typically Classes, but could also be other classifiers such as primitive, enumeration, interface, choice or syntype. A class diagram may also show relationships between the types, and their structural and behavioral features.

The definitions that are shown in a class diagram will by default be contained in the scope (for example a class or package) that owns the diagram, but it is also possible to show definitions from another scope.

In “Package Modeling” on page 191, information is provided on how to use package diagrams as a means for describing the packages of a system and how they depend on each other, but the same information can alternatively be described in a class diagram.

Example of class diagram

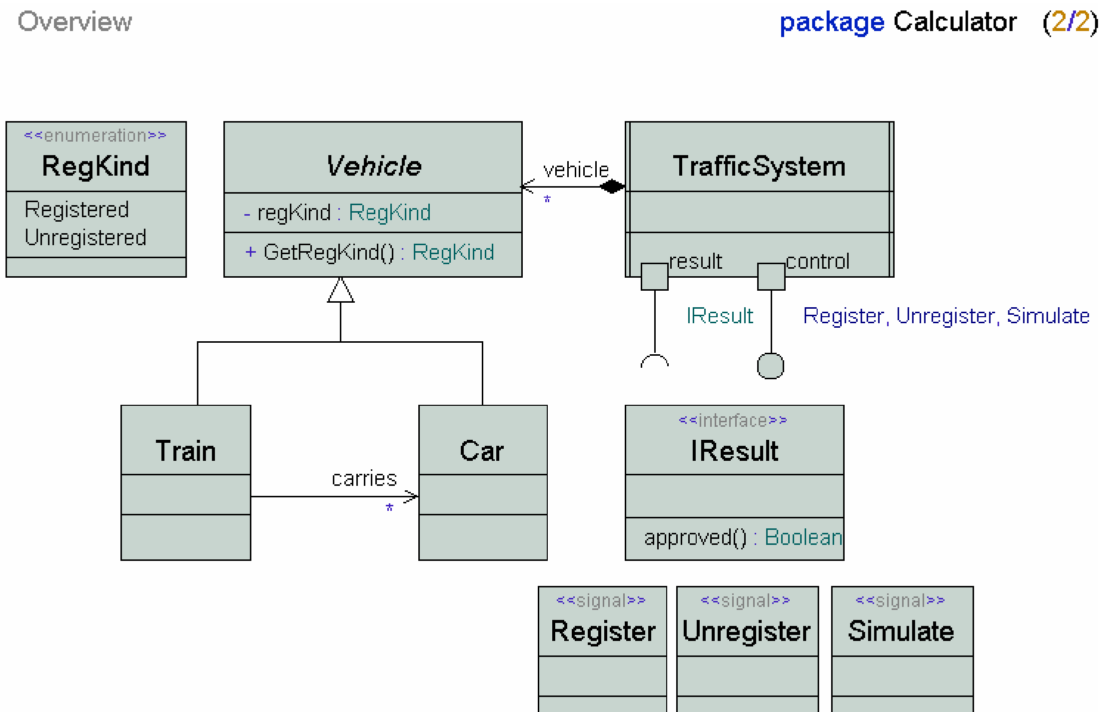


Figure 65: Class diagram

Model elements in class diagrams

The following model elements can be visualized in class diagrams:

- Artifact
- Collaboration
- Class
  - Active class
- Attribute
- Operation
- Port
- Interface
  - Realized interface
  - Required interface
- Signal
- Signallist
- Timer
- Datatype
- Choice
- Syntype
- State machine
- Relationships

See also

"Package diagram" on page 191.

Class

A Class is an abstraction of a group of objects that share the same properties (attributes), behavior (operations), structure, and relationships. A class may be instantiated (as long as it is not defined to be abstract) into a number of instances, all of which share the same properties.

Symbol

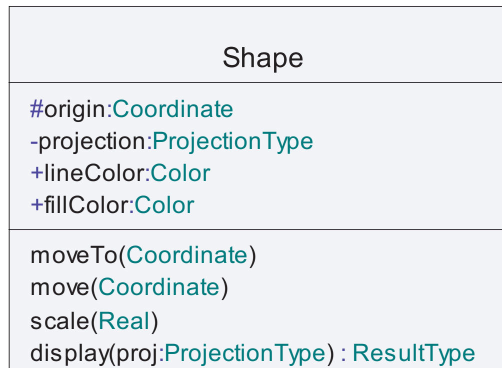


Figure 66: Class with Attributes and Operations

If instances of a class will maintain their own thread of execution (run concurrently with other instances), the class is said to be an Active class. If not, the instances will execute in the thread of another active instance, and the class is then said to be Passive.

To make a class active either:

- In the diagram (or the Model View), right-click the class you want to set as active, then on the shortcut menu click **Active**.
- Open the Properties Editor for the class and select **Active**.

The active class is displayed in the diagram with double vertical border lines.

A class may also have an internal structure, visualized in a composite structure (former architecture) diagram with parts and connectors, that describe how it is structured from an internal communication point-of-view. It may also have a state machine (called Initialize or the same name as the class) that describes it from a run-time execution point of view. This state machine is the “main” behavior that will be scheduled for execution when the instance of an active class is created.

Furthermore, a class may have a set of Ports, which specify how instances of the class may be connected to other instances in the architectural description of the class. The ports may also be used to group sets of interfaces that are exposed to different stakeholders.

There are several ways to add classes to a model.

- A class can be added directly in the Model View of the workspace window. Select the scope where the class should reside and from the shortcut menu select New/Class.
- Draw a class in a class diagram. Create and open a class diagram, select a class symbol from the toolbar and place it in the diagram.
- From a composite structure diagram: When double-clicking on an unbound part with no type name. This will allow you to create a new diagram describing the part in question, and this diagram will belong to an inline class created to the part. Possible diagrams are: class diagrams, composite structure diagrams, state machine diagrams, use case diagrams.

Multiple state machines in an active class

You can insert any number of state machines in an active class. However the following applies:

- If one of the state machines is named **initialize** or has the same name as the class, this state machine is considered to be the main state machine of the class. This state machine is executed when an instance of the class is created. If you omit this state machine, the Start and Stop symbols will automatically be inserted in the state machine diagram during the build process.
- Other state machines in the active class must explicitly be called in order to be executed. For example, it is possible to use such statemachines as the behavior when defining composite states.

Syntax

The class symbol contains compartments with editable text fields:

- Class Heading (required)
- Attribute (optional)
- Operation (optional).
- Constraint compartment (optional).
- Stereotype instance compartment (optional).

Class heading

The following example shows different class headings.

Example 19: Class heading

A simple class:

```
myClass
```

A class including virtuality:

```
redefined myC
```

A class using template parameters:

```
MyParamClass < type T, Integer c >
```

Attribute

Example 20: Classes and attributes

A class with an Attribute:

```
public A : Integer = 4
```

Attributes with multiplicity:

```
A: Integer [10]
B: Integer [3,>15]
C: Integer [*]
```

Operation

Example 21: Signal

```
signal s (Integer, Real)
```

Example 22: A method example

```
private m( x: Integer) : Integer
```

Abstract class

A class can be *abstract*. This means that it is not possible to create instances of this class. The class thus needs to be specialized before it can be instantiated.

If a class is abstract then the name of the class is shown using *italics* in the class symbol.

To make a class abstract either:

- In the diagram (or the Model View), right-click the class you want to set as abstract, then on the shortcut menu click **Abstract**.
- Open the Properties Editor for the class and select **Abstract**.

Virtuality

Virtuality defines whether a class can be redefined or not. This is only applicable if the class is contained in another class.

Visibility

The visibility of a feature of a class, typically an attribute or operation, defines if it can be accessed from outside the class where it is defined.

- **None**

  When no visibility is defined for a feature.
- **Public**

  This feature can be referenced from any place where its contained class is visible.
- **Protected**

  This feature can be referenced from any descendant (by specialization) of the class that defines the feature.
- **Private**

  Only the class that defines a private feature can use the feature.
- **Package**

  This feature can be referenced from any place within the nearest enclosing package from which its contained class is visible.

For more information about visibility, see Visibility.

External class

To define a class as external:

Open the Properties Editor for the class and select **External**. The external property is only shown in the Properties editor.

Classes and components

There is no specific concept for components representing abstractions, but it can be modeled in other ways.

Classes and components are very similar in UML. A Component is a subclass of Class in the Metamodel. They can both have attributes, operations, composite structure (what is drawn in composite structure diagrams), ports, interfaces, etc. The primary purpose of the component is to provide terminology, and to highlight those features that are most important in component-based modeling. This includes the ability to represent how the component is realized, and also to specify the required and provided interfaces of the component. Typically, the provided interfaces are realized by one of the realizing classifiers.

Constraint compartment

It is possible to attach one or several constraint compartments to a class symbol, with the **Add Constraint Compartment** shortcut menu choice. A constraint compartment can also be attached to other class-like symbols, such as interface or stereotype symbols.

A constraint compartment is placed below the last visible ordinary compartment of the class symbol.

A constraint compartment is similar to a Constraint symbol, with one read-only "{}" text label and a main text label that is editable.

The shortcut command **Show Constraints as Compartments** will create and attach one Constraint compartment for each constraint owned by the model element corresponding to the class symbol that does not already have a constraint compartment below the class symbol. The shortcut command **Show Constraints as Symbols** will create and attach one Constraint symbol for each constraint owned by the model element corresponding to the class symbol.

Stereotype instance compartment

It is possible to attach one or several stereotype instance compartments to a class symbol, with the **Add Stereotype Instance Compartment** shortcut menu choice. A stereotype instance compartment can also be attached to other class-like symbols, such as interface or stereotype symbols.

A stereotype instance compartment is placed below the last visible ordinary compartment of the class symbol.

A stereotype instance compartment is similar to a Stereotype instance symbol, with one read-only "«»" text label and a main text label that is editable.

The shortcut command **Show Stereotypes as Compartments** for class symbols will create and attach one Stereotype instance compartment for each stereotype instance applied to the model element associated with the class symbol that does not already have a stereotype instance compartment below the class symbol.The shortcut command **Show Stereotypes as Symbols** will create and attach one Stereotype instance symbol for each stereotype instance owned by the model element corresponding to the class symbol.

See also

"Datatype" on page 224

"Choice" on page 227

Collaboration

The collaboration symbol behaves like a class symbol, including support for Icon Mode, but the collaboration symbol is not showing attributes and operations in the symbol.

Attribute

An attribute is a structural feature that may hold one or several values at runtime.

Attributes are used for modeling several different, but related, constructs of the UML language:

- **Attributes**

  An attribute of a Structured Classifier is modeled as an Attribute. The instance of such an attribute is often called a *field*, and it may be referenced by using a Field Expression. There are also so called class-scoped attributes (also called static attributes). All instances of a class share the same value for a class-scoped attribute.

  In composite structure diagrams, attributes with composition aggregation are often referred to as parts, which is due to the particular nature of that view to show the hierarchical structure of a class.

  Attributes are also used to represent the ends of an association.

- **Local variables**

  A local variable of a state machine, operation or compound statement is modeled as an Attribute. Such an attribute may be referenced directly by its name, with a scope qualifier if necessary.

- **Constants**

  A constant is modeled as a read-only Attribute. The value of the constant is the Default value of the attribute. Typically constants are defined on package level, but it is possible to define a constant wherever an attribute can be defined. Constants may be referenced directly by its name, with a scope qualifier if necessary. As the name indicates, the value of the constant may not be changed once it has been set.

An attribute always has exactly one static type. This type is determined at the point of defining the attribute, and can be either:

- a class,
- an interface,
- a primitive or enumeration,
- a syntype,
- a delegate,
- or a choice.

Attributes are closely related to associations. A navigable association end and an attribute is in practice the same thing. This implies that it is possible to first define an attribute and then visualize this attribute in a class diagram as the role name of a navigable end of an association. The opposite is of course also possible: Start by defining an association with one navigable association end. Then visualize the association end in the attribute compartment of a class symbol as an attribute.

The navigability is necessary if you want to use a specific association end/attribute that it is associated with to make a call.

Example 23: Navigability

Given the classes A and B. You want to invoke an operation B.op() from the class A.

With an association with an association end name ("role name") 'b' in the direction from A to B you can make a call 'b.op();' only if the association is navigable.

Attributes can also be visualized as symbols in composite structure diagrams. Although this is allowed for all attributes of the containing class, this possibility is often only used for parts.

Aggregation kind

If an attribute is typed by a class this implies that the values for this attribute will be objects, that is instances of the class. In this case the attribute can have different aggregation kinds that determine the lifetime relationship between instances of the class containing the attribute and the value instances:

- **None**

  There is no lifetime dependency between the instances of the two classes. This implies that the attribute will contain one or more references to instances of the value class.
- **Shared aggregation**

  There is no lifetime dependency between the instances of the two classes. However, informally one is considered to be "owned" by the other. In the attribute compartment a shared aggregation is indicated by the keyword "shared" before the attribute name as in "shared a:myclass". Some code generators may attach a specific semantics to shared but in practice it is rarely used due to its weak semantics, and it is normally better to use an association with no aggregation instead.

- **Composition**

  There is a strong part/whole relationship between instances of the containing class and instances of the value class. In practice this implies that there is a lifetime dependency between the two instances. If the containing instance is terminated then the contained instance will also terminate. Composition is indicated by the keyword “part” before the attribute name as in “part a:myclass”

Note

A non-static attribute may hold a value only when its defining context has been instantiated. The possible defining contexts listed above for an attribute are instantiated differently. For example, a package is instantiated when it is used, and an event class is instantiated when it is invoked.

Default value

An attribute may have a default value specified as an Expression. If it does not have a default value, its value remains undefined when the defining context is instantiated until it is explicitly assigned.

Port

An attribute that is typed by a Class may have communication ports to which connectors can be connected. These connectors describe communication paths in a system that convey signals to and from the attribute. This is mainly used when the attribute represents a part.

Multiplicity

An attribute may have a multiplicity, modeled as a collection of ranges. It specifies a restriction on how many instances the attribute may hold at runtime.

Depending on whether the multiplicity of the attribute is >1 or not the actual type of the attribute is different. If the multiplicity is >1 then the attribute will have a container type that can hold a list of values. If the multiplicity is exactly 1 (or `0..1`) then this is not the case.

Depending on what datatype libraries are available the container type may be different. Typically different code generators will supply different container types to provide a suitable integration with the target language. If no specific datatype library is loaded the String type will be used in the built-in pre-

defined package as the type of attributes with multiplicity > 1. (The String type is a predefined collection type that represent an ordered list, or a sequence. The values in the list must adhere to the type of the attribute.)

In the attribute compartment of a class symbol the multiplicity is shown within brackets after the type of the attribute as in:

```
a : myClass [*]
```

In the above example, the multiplicity is unbound (represented by the asterisk), meaning that it can have any number of values.

If no multiplicity is given it is considered to be 1 by default.

Initial cardinality

For a composite attribute with a multiplicity > 1 there is a shorthand that allows specifying the initial number of instances using an Expression. That number specifies how many instances that will be automatically created when the owning Class is instantiated. If an initial number of instances is omitted, exactly one instance will be created.

Note

If and how the cardinality is interpreted is code generator dependent. Some code generators may ignore the cardinality of an attribute.

The initial cardinality can be given if an attribute is shown using a part symbol in a composite structure diagram. However, in this kind of symbol the syntax is as in:

```
a : myClass [*] / 2
```

where the initial number of instances for ‘a’ would be 2.

Visibility

It is possible to specify Visibility for attributes. This can be one of **public**, **private**, **protected** or **package**.

Derived

An attribute can be declared to be `derived`. This indicates that the value of the attribute is not stored in the corresponding object, but instead is computed from for example the values of other attributes. The syntax for a `derived` attribute is a ‘/’ preceding the attribute name as in

```
/a:myClass
```

For more information on how to specify the derivation rule for a derived attribute, see Derived.

Static

A static attribute is an attribute that is owned by the class scope rather than the instance scope. This means that there is only one Attribute instance that is shared by all the instances of a particular class.

Constant

A Constant attribute is an attribute which value cannot be modified dynamically. The value of the constant is the default value of the attribute.

An external Constant attribute means that the value is defined outside of the model or at a later time (build time for example).

Example 24: Textual constant declaration

```
const Integer a = 10;
const Integer extern ext_const;
```

Operation

An operation is a declaration that instances of a class will be able to handle calls that match the signature of the operation. An operation can be implemented either by an operation body or a state machine. This implementation (often called a method) will be executed when the operation has been invoked. This means that if the receiver is a passive instance, the implementation will be executed immediately after the operation has been invoked, while if the receiver is an active instance the execution of the implementation may be delayed and executed some time in the future when the instance is in a state where the operation call is accepted.

Operations can be declared textually in the operations compartment of a class symbol, and using a special operation symbol.

DOORS Analyst supports a `derived` property for operations as an extension to standard UML. This can be used to indicate that the operation has no implementation but is implicitly computed. This property is for analysis only and will not affect any generated code.

Symbol

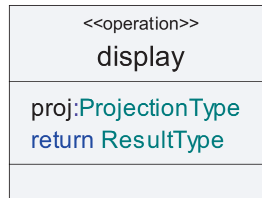


Figure 67: Operation

Syntax

The symbol contains two editable text fields: Operation Heading and Parameters. The bottom field is always empty.

Active class

An Active Class is a class with its own thread of control. It is distinguished from the normal (passive) class by the property Active. Graphically, this is indicated by the special Active Class symbol, as in Figure 68 on page 212.

Symbol

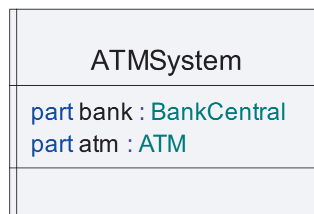


Figure 68: Active Class

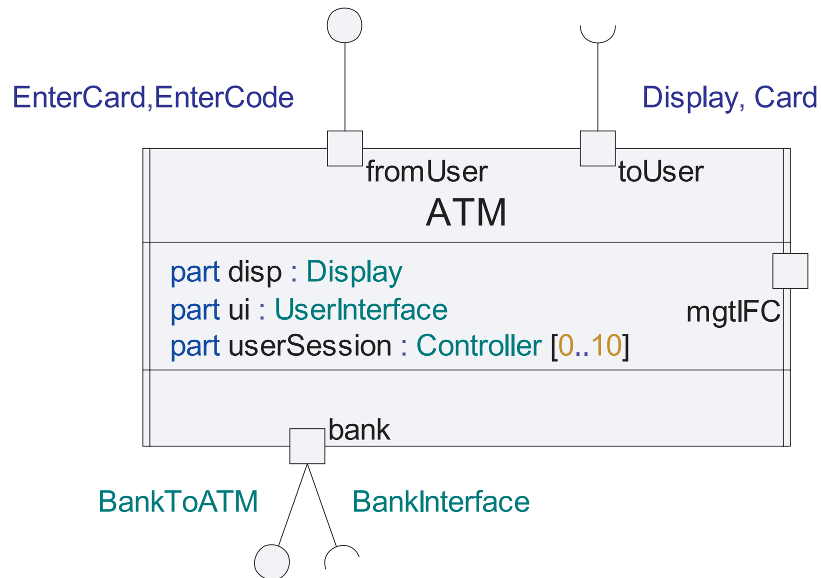


Figure 69: Active Class with Ports and Realized and Required Interfaces

You can make a class active by:

- Selecting it and in the shortcut menu choose **Active**.
- Selecting it and in the Properties Editor select **Active**.

The active class is the fundamental building block for modeling real-time behavior in UML. Active classes define both the structure (architecture) and behavior of a model. This duality of the active class concept in UML offers strong and flexible design capabilities.

Structure

The structure of an active class is defined in one or several Composite structure diagram, which defines the active class as a set of instances of other active classes. These active classes can also have structure, thus enabling descriptions of complex architectures.

Behavior

The behavior of an active class is defined by a State machine in one or several State machine diagram. This state machine should be named `initialize()`, or given the same name as the class.

In order for an active class to be completely specified, it must have either a structure definition, a state machine definition, or both.

An active class has its own flow of control and can both initiate behavior and passively react to behavior as observed on its interfaces. Traditionalists prefer the name *reactive* class instead of active class, since such classes are typically event-driven. The initiation of behavior is often done through the use of timers; at the expiration of a timer some behavior is kicked into gear.

When an active class has several contained parts defined in its composite structure diagram(s), each part executes asynchronously and concurrently with other parts in the system. This semantic ensures that the model can be deployed in a distributed physical environment and is not dependent on being run on a single processor with shared memory access.

An active class can realize and require interfaces via a Port. Ports together with their required and realized interfaces define the static contract between the active class and its environment.

Attributes and operations

In the active Class symbol, it is possible to specify or show attributes of the class in the second compartment of the symbol and operations in the third compartment.

See also

"Attribute" on page 206

"Operation" on page 203.

Port

Ports are named interaction points of an active class. They specify the implemented interface (realized) and the needed interfaces from other classes (required).

Ports are typically used only on active classes. To visualize an already created port on an active class symbol or a part symbol, use the **Show/Hide** command on the shortcut menu and point to **Show Ports**.

Symbol

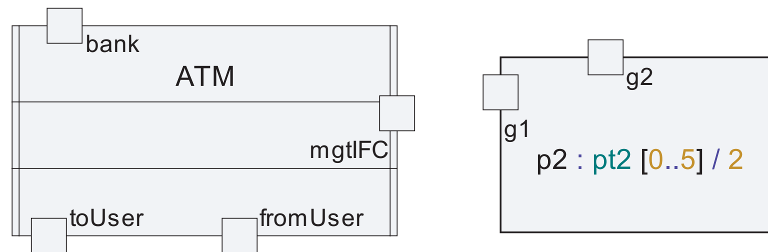


Figure 70: Ports on a Class and Ports on a Part

Hint

The easiest way to attach a port symbol is to first select the frame of the symbol where the port symbol should be placed and then click the port symbol in the toolbar.

Port type

The symbol has one text field that should contain a name and that optionally may contain a **type**. The type of the port is mainly intended to be used in an analysis phase.

Note

The code generators in DOORS Analyst do not take the port type into account. Instead code is generated based on the information given for realized and required interfaces of the port.

Behavior ports

There are two different kinds of ports: behavior ports and non-behavior ports. The difference between these two different kinds of ports is that a behavior port is directly associated with the state machine of the class, whereas a non-behavior port needs to be connected using connectors and are typically only relaying the communication from outside the class to some of the internal parts of the class.

A Behavior port is a port that is directly connected to the state machine of the class. All signals sent to this port are consumed by the behavior of the class itself.

Ports and interfaces

For each port, the realized and required interfaces may be specified. The realized interface of a port defines the incoming requests that can be handled via the port. The required interface defines the outgoing requests that must be handled by a class connected to the port from the outside via one or more connectors. In Figure 69 on page 213 you will find an example of ports with realized and required interfaces.

When defining the structure or behavior of an active class, ports can be declared on the border of a diagram used for this purpose (a composite structure diagram or a State machine diagram). Ports can also be referenced from parts, where they are shown on the border of the part symbol.

It is also possible to send messages through a port (without knowledge about possible receivers at the other end of the attached connector) from a state machine as an addressing mechanism.

The realized (or required) interface of a port may typically contain references to interfaces, but also to a signal list, signal or attribute.

The realized and required interfaces of a port are visualized by attaching the Realized interface symbol and the Required interface symbol to the port. On these symbols, the supported or needed Interface names (signal list, signal or attribute) can be specified.

Another way to specify the Realized and Required Interface of a Port is by the Properties Dialog.

Ports represent:

- Connection points for Interfaces to classes
- Connection points for Connector lines in Composite structure diagram, connecting instances of these classes with other instances or with the enclosing frame symbol.

The port symbol can be placed

- On Class symbols
- On Part symbols
- On Behavior symbols
- On the frame of a State machine that is owned by an active class
- On the frame of a composite structure diagrams

- Within Architecture and State machine diagrams (which has the same semantics as when the port is placed on the frame of these diagrams).

A port can have both explicit and implicit connectors. Each Port symbol can have zero, one or two interface symbols attached to it.

When you have two interface symbols, one of them should be defined as a Realized Interface symbol specifying the incoming interfaces (or signals) to the port and the other should be defined as a Required Interface symbol specifying the outgoing interfaces from the port.

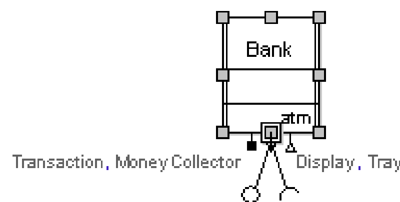


Figure 71: A port with realized and required interfaces

Ports with or without interfaces can be drawn directly on a class. Either:

- Select the class and hold down SHIFT while you click on the toolbar port symbol. Type the port name. The port will be positioned on the class' left border segment as close as possible to the upper left corner.
- Click on the toolbar port symbol, click on the class where you want to position the port. Edit the name text field.

Inheritance

In case of a generalization between classes where there are ports belonging to the supertype these ports will also be inherited.

Ports can be declared public and private to distinguish if a port is externally exposed or if it is only used internally. It is possible to add more signals to ports in subclasses.

Interface

An interface is a structured classifier that may not be instantiated. Instead, it is used for grouping a set of attributes, operations, and signals that must be implemented by the class that implements the interface. A class that imple-

ments an interface is said to **realize** the interface, thus supporting the operations declared in the interfaces. A class can also **require** interfaces, it is then dependent on other active class(es) in order to perform its operations.

Symbol

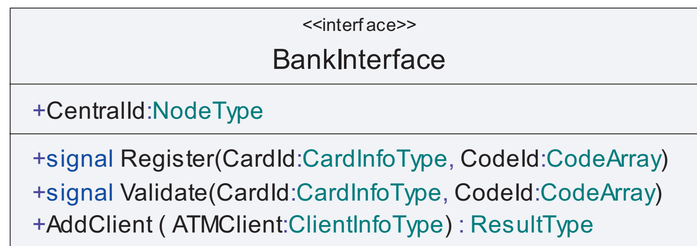


Figure 72: Interface symbol

The operations of an interface typically describe services that are offered by the class(es) that realizes the interface. Naturally, a class may realize more than one interface.

Apart from operations, an interface may contain signals and attributes. It may also contain other definitions, such as types.

An interface can be specialized and may have Template parameters. Multiple inheritance of interfaces is a useful mechanism to define the communication interfaces of active classes.

Interfaces can also be associated to each other to provide a definition of protocols or contracts between classes that realize the involved interfaces. An example is given in Figure 73 on page 219 that defines the `MgmI` and `MgmReplyI` interfaces. The association between the two interfaces establishes a relationship between them. This means that wherever one of the interfaces is referenced, for example on a port or associated with a connector, the other interface will automatically be inserted in the other direction. So, for example if a class realizes the `MgmI` interface via a port then the `MgmReplyI` interface will automatically be a required interface of the same port

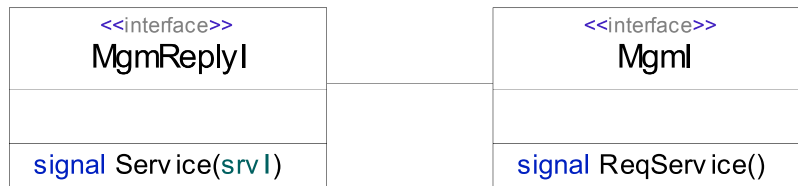


Figure 73A contract defined using two associated interfaces

Syntax

The symbol contains three editable text fields:

- Heading,
- Attribute, and
- Operation.

The heading field is used to define the name of the interface.

The attribute field contains definitions of attributes that must be implemented by classes realizing the interface. Typically, this is a shorthand for a getter operation and a setter operation to a protected attribute of the realizing class.

The operations field contains definitions of operations and signals that must be handled by classes realizing the interface.

See also

“Realized interface” on page 219

“Required interface” on page 220

Realized interface

A realized interface attached to a port on a Class visualizes what interfaces the Class realizes through that port. Interfaces, signals, signal lists and attributes may be specified in the text field.

Symbol

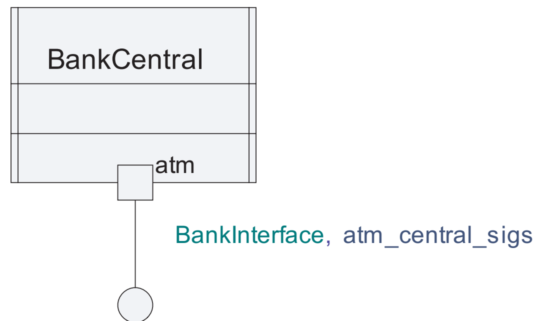


Figure 74: Realized Interface

Syntax

The symbol contains a text field.

Example 25: Realized interface

```
S, p, SigList
```

See also

"Interface" on page 217

"Required interface" on page 220

Required interface

A required interface attached to a port on a class visualizes what requests the class expects to be handled through the port. Interfaces, signals, signal lists, and attributes may be specified in the text field.

Symbol

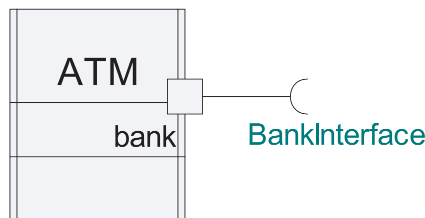


Figure 75: Required Interface

Syntax

The symbol contains a text field.

Example 26: Required interface

```
S, p, SigList
```

See also

"Interface" on page 217

"Realized interface" on page 219

Signal

A Signal is one of the primary means for communication in UML. A signal represents an asynchronous message that is sent between active classes. The signal can carry data, which must conform to the declared parameter types of the signal.

A signal is most conveniently declared together with other signals, operations and attributes in an Interface that represents the capabilities of the classes that realize the interface.

However, a standalone signal declaration can also be made using a special signal symbol similar to a class symbol as shown in Figure 76 on page 222.

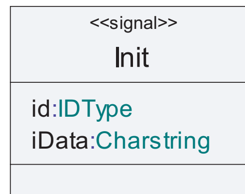


Figure 76Signal

If numerous distinct signals will be used, it is often more practical to declare the signals textually in a text symbol:

Example 27: Textual signal declaration

```
signal Init (IDType id, Charstring iData);
signal SetupReq, SetupInd, AbortReq, AbortInd;
signal ForwardedMsg (IDType, MsgData);
```

Syntax

The signal symbol contains two editable text fields:

- Heading
- Parameters

The heading field declares the name of the signal and the parameters field declares the parameters of the signals. The name of the parameters may be omitted, but the parameter types are required.

The third compartment that exists for many class like symbols is always empty for signal symbols.

See also

“Message” on page 170

“Signallist” on page 223

“Interface” on page 217

“Timer” on page 223

Signallist

The keyword `signallist` is used to denote a group of related signals in order to make the description easier to comprehend. It is typically used in ports and connectors.

Example 28: signallist declaration

```
signallist MgtSignals = MOGetStatus, MOSet, MOReset;
```

Note

Using an Interface to group signals together is a more structured approach, compared to signal lists, since the Interface also encapsulates the signal declarations.

See also

"Signal" on page 221

"Interface" on page 217

Timer

A Timer is an event that, in the same fashion as a signal, can trigger transitions. A timer is set by an implementation executed by an active class and at timeout, a timer event can be received by the state machine of that same active class instance. A time value is associated with an active timer, which is the time of the timeout.

Symbol

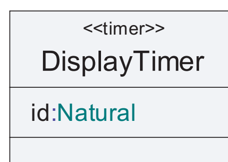


Figure 77: Timer

Timers can, like signals, have parameters. This can be used to allow to set more than one timer of the same kind without resetting the already active timer; that is several timers with different parameter values may be active at the same time.

Syntax

Timers can also be declared textually in a text symbol:

Example 29: Textual timer declaration

```
timer DisplayTimer (Natural id) = 2;
timer BankTimer () = BankTimeout;
timer UserTimer ();
```

When declaring a timer textually, it is also possible to give the timer a default duration, that is a duration before timeout that allows to set the timer without specifying the duration.

See also

“Timer set action” on page 284

“Timer reset action” on page 284

“Timer set” on page 175

“Timer reset” on page 175

“Timer timeout” on page 175

“Timer active expression” on page 296

Datatype

Datatypes are used for two different purposes:

- To describe primitive types that are available
- To describe user-defined enumeration types

Primitive types are most often defined in model libraries that accompany specific UML profiles, either standalone profiles or profiles defined to be used together with specific code generators. In the latter case the datatypes typically define the target language primitive types and makes them available in UML models.

It is however also possible to define primitive datatypes in user models, but this may cause code generation problems.

An enumeration defines a set of values simply by enumerating them as a list of enumeration literals.

In any case the datatype may also optionally contain behavior that is defined by **operations**.

Symbol

Figure 78: Enumeration Datatype

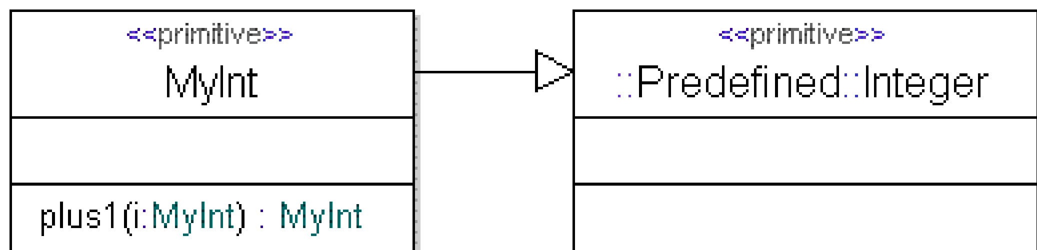


Figure 79: Datatype with operator

Enumerated datatype

An enumerated datatype is a datatype where the literal values are logical names. The logical names can optionally be attached to an integral value specified by a simple expression.

The available default operations are:

- Equality (`==`, `!=`)
- Relational operations (`<`, `>`, `<=`, `>=`)
- Assignment (`=`)

Example 30: Enumerated datatype

```
enum UKColors { blue, red, white }
```

```
enum LinePrinterState {
  outOfService = 1,
  inServiceFree =2,
  inServiceBusy = 6
}

void op() {
  LinePrinterState e;
  Integer i;
  e = cast<LinePrinterState>(1);
  e = inServiceFree;
        i = cast<Integer>(e);
}
```

Note

It is possible to convert between Integer and enumeration types using the cast operation as in the operation `op` *in* *Example 30 on page 225*.

Primitive datatypes

Primitive datatypes are usually defined in model libraries in profiles but can also be user defined. However, a user-defined primitive type will not have a literal syntax, which makes them less useful in practice.

There are however two ways to relate the datatype to another already existing datatype:

- use copy constructors
- use inheritance

In both of these cases the literal syntax of the existing datatype will be used. The copy constructor mechanism is the recommended mechanism to introduce new primitive datatypes in UML and this is what is used in most model libraries.

Note

Primitive datatypes usually need special treatment in code generators. A user-defined primitive datatype is not likely to work in a code generator unless specifically stated in the code generator documentation.

Example 31: Datatype with operators

```
datatype simpleInt {
  simpleInt(Integer) {}
}
datatype myInt : Integer
{
```

```
  myInt plus1 ( myInt i) { return i+1;}
}
```

Literal

A Literal is a definition of an element of the type defined by an enumerated datatype. The literal is owned by that datatype. The visibility of a literal is always public.

Aside from having a name (which all definitions have), a literal may also have an integral value which allows it to be used in arithmetic expressions.

Choice

A Choice is a datatype that can hold one value. This value can be of different datatypes during the execution. A choice of which type is made when assigning a value to a variable. For each potential type field, there is a boolean operator `IsPresent()` that can check if the field is present or not.

Example 32: choice

```
choice IntOrBool {
    Integer a;
    Boolean b;
}

IntorBool ib;
Integer i;
Boolean b=true;

ib.a=5;
i=ib.IsPresent("a")?ib.a:0; /* check if ib is Integer;
       if Integer, return ib,
       if not, return 0 */
ib.b=b;
```

Example 33: choice

```
choice IntOrBool {
  Integer a;
  Real r;
  Integer GetInt() {
    if (IsPresent("r")) {
      return 0;
    } else {
      return a;
    }
  }
}
```

Using the `IsPresent()` operator:

```
IntOrBool MyVar;
Real num_real;
Integer num_int;
MyVar.a=1;
if(IsPresent(MyVar,"a"))
  {
    num_int =MyVar.a;
    MyVar.r=3.14;
  }
  else
  {
    num_real=MyVar.r;
  }
if (MyVar.IsPresent("r")) {
switch (MyVar.r) {
  case 3.14 :
  {
    nextstate idle;
  }
  default :
  {
    nextstate idle;
  }
}
}
```

A choice instance value can be specified by an instance expression having only one assignment where `choice_field = value`.

Example 34: Choice instance value

```
choice choice_type
{
  public Integer ifield;
  public Boolean bfield;
}
choice_type an_int = choice_type (. ifield = 1.);
```

Syntype

A syntype is a datatype that is based on another datatype, the parent type. The two types are not distinct in terms of type compatibility and literals. The literals of a syntype are either identical with or a subset of the literals of the parent. A syntype can be regarded as an alias of another type; an alias that may be constrained.

Example 35: Syntype

```
syntype myInt = Integer constants (> -10, != 0, <10);
syntype smallPrime = Natural constants (1,2,3,5,7);
```

```
Integer [1..10] myvar; /* inline syntype definition */
```

Note

Constraints attached to a syntype are treated informally, that is they are not checked by the Semantic Checker, or considered by the code generators.

State machine

The State machine concept is explained in detail in the Behavior Modeling section.

Stereotype

The Stereotype concept is explained in detail in the Extensibility section.

Relationships

The following Relationships can be used in class diagrams. These are described further in the section Relationships in UML.

- Association
- Aggregation
- Composition
- Dependency
- Extension
- Generalization
- Realization
- Manifestation.
- Containment

Object Modeling

While class modeling focuses on finding the kinds of objects in the designed application, object modeling is concerned with describing how these objects may appear at run-time. Typical questions for this analysis activity may be:

- **Which objects exist in the application at different points in time?**
- **What does the objects look like in terms of attribute values etc.?**

- **How are the objects linked to each other? Which objects have knowledge of which other objects?**

Objects are also knows as instances, and instance modeling is thus also used as a term for describing this analysis activity.

It is common to perform object modeling in parallel with class modeling. As objects of the application are identified they can be defined in the model. This can be done even before the type of the object is known.

In most real-world applications the number of objects at run-time is very large. It is therefore common to only describe those objects that are of special interest for the design. For example, it may be particularly interesting to identify objects that get created at application start-up time to get an understanding of the initialization phase of the application.

Object modeling uses mainly Object Diagrams for defining objects and their relationships, although Class diagrams are sometimes also used.

Object Diagram

An object diagram gives a static view of objects that exist in an application at a specific point in time (a "snapshot" view). The objects shown in an object diagram can be named, and it is possible to specify the type of objects. The objects' attribute values, called **slots**, can also be specified. Links between objects can be visualized using link lines.

A named object in DOORS Analyst is called a **named instance** to distinguish it from unnamed instances, such as applied stereotype instances. A Named Instance is a definition which by default is placed in the scope containing the object diagram. It is, however, also possible to show named instances from other scopes by dragging them from the Model View onto an object diagram.

An object diagram may contain multiple instance symbols showing the same named instance.

Example of object diagram

The object diagram below shows a snapshot view of objects available in the application described by the class diagram shown in Figure 65 on page 199.

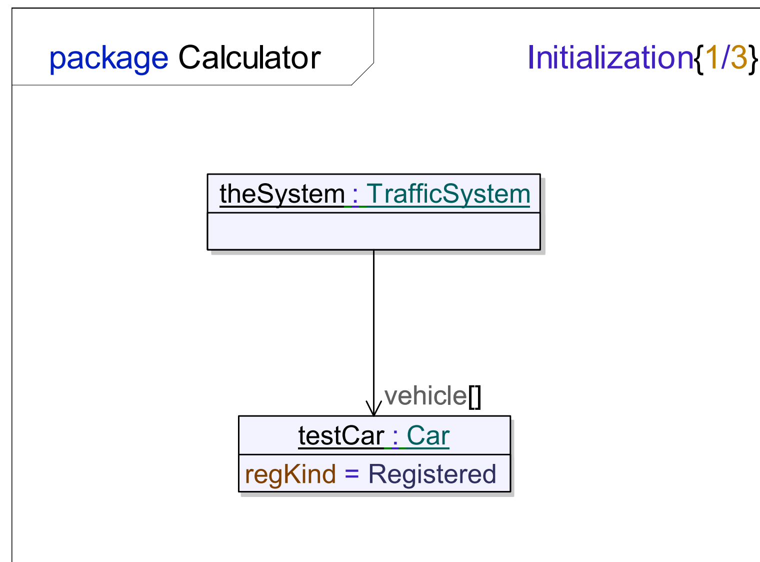


Figure 80: Object diagram

This object diagram tells us that at some point in time (supposedly at initialization time judging from the name of the diagram) this application contains one instance of the `TrafficSystem` class. It has one instance of `Car`, called `testCar`, in its `vehicle` list attribute. The `testCar` object has the value `Registered` for its `regKind` attribute.

Model elements in object diagrams

The following model elements can be visualized in object diagrams:

- Named Instance
- Slot
- Dependency

See also

"Class diagram" on page 199.

Named Instance

A named instance represents an object (instance) in a modeled system and describes this object completely or partially. Since objects may change over time, a named instance only provides information about the object at a specific point in time, or for a specific time period. Note that UML object diagrams do not provide means for formally specifying

- the point in time, or time period, where the object complies with the named instance specification
- whether or not the named instance is a complete or partial specification of the object

A named instance may have a name. Often this name is to be interpreted informally, and does not correspond to any property at the run-time object described by the named instance. However, the usual rules for definitions apply to named instances. For example, the names of named instances in the same scope must be unique (see Scope, model elements, and diagrams).

A named instance may have a type. If the specified type is a class, the named instance describes an object of that class. If it is a datatype, the named instance describes a value of that datatype. It is also possible to specify a behavioral feature, such as an operation or a signal, as the type. In that case the named instance describes an event in the system. For example, if the type is an operation the named instance describes an operation call, and if the type is a signal it describes an event of that signal.

The type of a named instance can also be an association. In that case the named instance represents a Link.

It is allowed to specify an abstract type for a named instance. This does not mean that the described object is of abstract type, but merely that all shown properties for the object belong to the abstract type only. The described runtime object would have a type that is a concrete subtype of the abstract type.

If the named instance type contains structural features, such as class attributes or signal parameters, the named instance may specify values for those structural features. Such a value specification is called a Slot.

A named instance is shown in an object diagram using an **InstanceSymbol**.

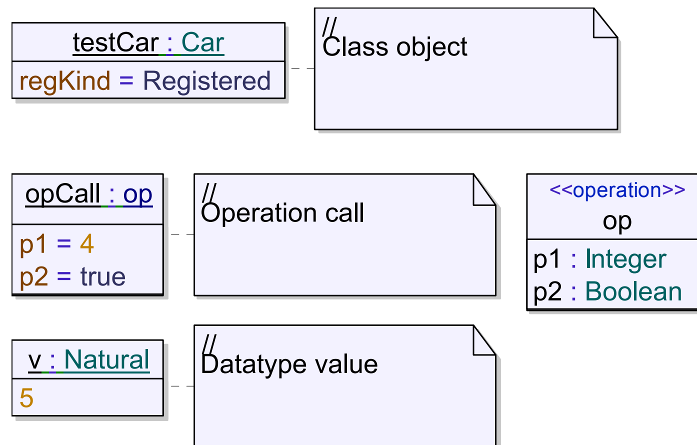


Figure 81: Instance symbols defining named instances

As can be seen an instance symbol contains two basic compartments. The upper compartment holds the name and type of the named instance, while the lower compartment contains the slots. Note that the syntax for defining a slot is the usual assignment syntax (the structural feature is assigned a value). For data type values a plain value can also be used.

Note

Currently the semantic checker will not check type compatibility between a datatype value and the datatype. Hence, datatype values in object diagrams are for informal modeling only.

Link

A link is a named instance whose type is an association. It describes a run-time relationship between two objects. In programming language terms, a link could correspond to a pointer or a reference.

Links can be visualized in object diagrams in two ways:

1. As a link line, connecting two instance symbols.
2. As an ordinary slot in an instance symbol, where the right hand side of the slot refers to the target named instance.

Figure 82: Two ways to specify a link

The text that is entered on the target end of a link line is an expression (see Expressions). It becomes the left hand side of a Slot expression.

The name of a link can be specified by typing it in a label in the center of the link line.

Slot

A slot is a value specification for a structural feature belonging to the type of a named instance.

Slots are used for showing those values of an object that are of interest. The fact that a named instance has no slots defined does not mean that the corresponding object has no structural feature values, but merely that those values are not of interest in the model.

Slots may reference all kinds of structural features of a type, including inherited features, and features with non-public visibility.

A slot is an assignment of a value (the right hand side) to a structural feature (the left hand side). The right hand side is often just a plain identifier, but more advanced expressions can also be used. Refer to the following model:

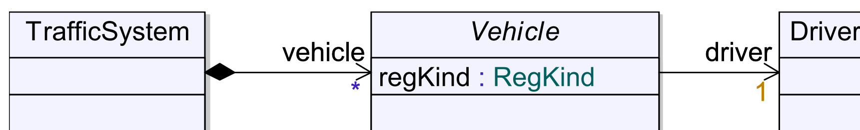


Figure 83: Three classes with relationships

Slots defined for an instance of TrafficSystem can for example have the left hand sides listed in the table below:

Slot left hand side	Meaning
`vehicle[]`	One instance in the `vehicle` collection. The index of the instance in the collection is not specified.
`vehicle[4]`	One instance in the `vehicle` collection, located at index 4.
`vehicle[].driver`	The `driver` instance of an instance in the `vehicle` collection.

Note that if the latter example is visualized with a link line we can show these kinds of indirect links between objects in a compact way:

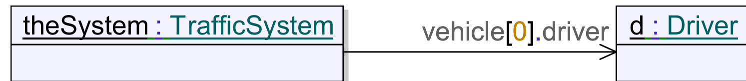


Figure 84: Visualizing the link from the traffic system to the driver of its first vehicle

Self reference

There are two equivalent ways to specify self references for objects. The right hand side of such a slot can either be a reference to the containing named instance, or the keyword `this` can be used.

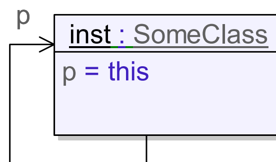


Figure 85: Self reference shown in slot label and with link line

Architecture Modeling

During architecture modeling, the internal structure of active classes is described from a communication point of view. This is done by connecting the attributes of the class (in this context referred to as parts) with connectors, and to specify which signals that may be sent along these connectors. This structure of parts and connectors is called the architecture, or composite structure, of the class.

Architecture modeling typically takes place after, or in parallel with, class modeling during the design phase.

Composite structure diagram

A composite structure (former architecture) diagram defines the internal runtime structure of an active class, in terms of other active classes. These building blocks are referred to as parts when they are composite parts of the containing class. Furthermore the parts are also restricted to be instantiations

of active classes. Composite structure diagrams may also express the communication within the active class by visualizing connectors between the communication ports of the parts.

Example

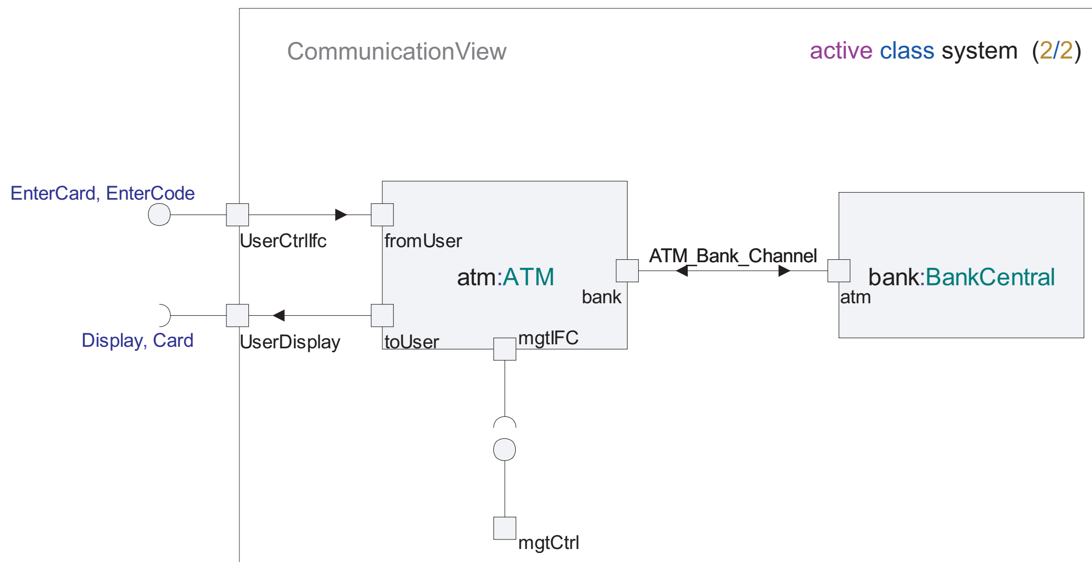


Figure 86: Composite structure diagram with parts, ports and connectors

Part

A part represents one or more instances that is owned by a containing class instance.

As for all attributes a part can have a Multiplicity that constrains the number of run-time instances that may exist. If the part has a multiplicity > 1, then a container type is assumed for the parts. The specific container type can differ depending on the loaded profiles and Add-Ins, but by default the String type is used.

When an instance of the containing class is created, a set of instances corresponding to these parts may be created either immediately or at some later time as described by the initial cardinality and the multiplicity for the part.

Symbol

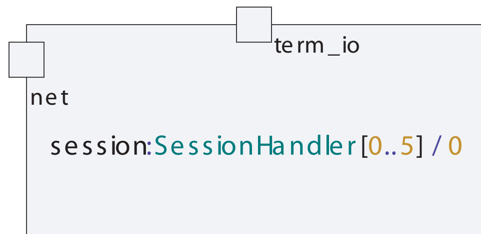


Figure 87: Part

- If the part symbol has only a name, the implicit class is constructed automatically when the part symbol is created.
- More than one part symbol with the same name can be present in the same composite structure diagram.

If the referenced class is omitted, this corresponds to a part definition with an inline class definition. Specifying a part in this way means that the class definition is not separated from the usage of the class which makes the description more compact, but on the other hand less suitable for reuse.

A part of an active class may be shown in the attribute compartment of the active class symbol, since a part is an Attribute of the containing class. When an attribute is of the kind part, it describes a composition relationship between the container class and the part class.

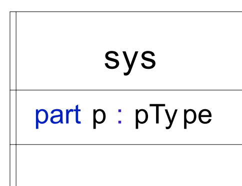


Figure 88: A part visualized in the attribute compartment of a class symbol.

It is also possible to give an overview of a hierarchy of parts using composition relations in a class diagram as in Figure 89 on page 238

The initial cardinality determines the number of initial instances that will be created automatically when the containing entity is created. If no initial cardinality is given, the number of initially created instances will be equal to the lower bound of the Multiplicity of the part. If no multiplicity is given, one

instance will be created automatically and there will be no upper bound for the number of simultaneous instances. These instances are instances of the classifier typing the part.

Parts may be joined by connectors attached to ports. Parts are used to describe both static and dynamically created and terminated active instances.

A part specifies that a set of instances may exist; this set of instances is a subset of the total set of instances specified by the classifier typing the part. When an instance of the containing class is terminated, the contained instances will also terminate.

A part symbol refers to an attribute in the model. The appearance of a part symbol in a composite structure diagram varies with the aggregation kind of the corresponding attribute. If the Aggregation kind is composite, the outline of the part symbol is a solid line. If the aggregation kind is reference or shared, the outline of the part symbol is dashed.

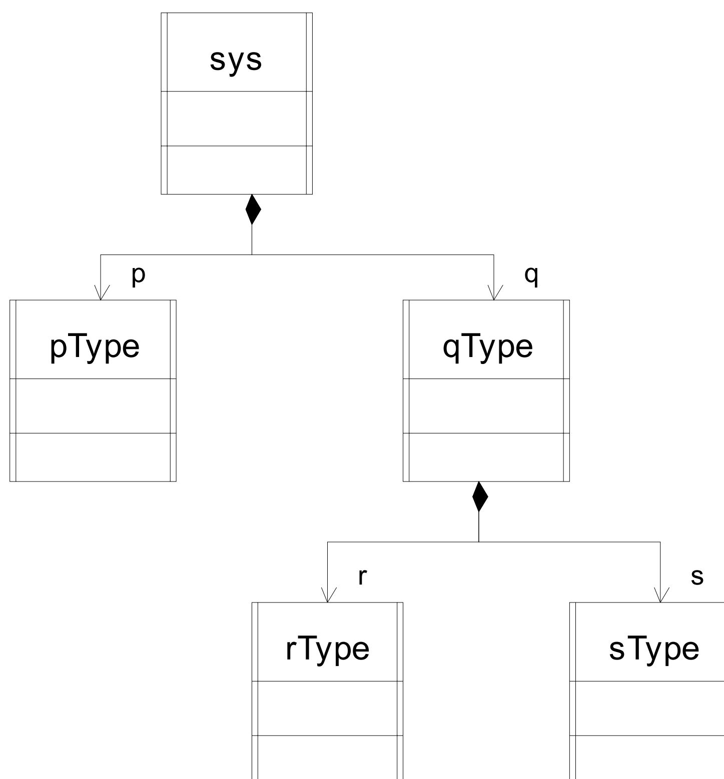


Figure 89: A part hierarchy visualized using composition in a class diagram

Example 36: Simple part

```
myP
```

Example 37: Type-based part

```
myP : PT
```

Example 38: Part with initial and maximum number of instances specified

```
myP : PT [0..10] / 1
```

Connector

A Connector specifies a medium that enables communication between parts of an active class or between the environment of an active class and one of its parts. Connectors can visualize communication paths in an intuitive fashion.

A connector may be unidirectional or bi-directional and specifies for each direction the allowed information. Information that can be sent or conveyed on a connector can be described by: signal, attribute, signal list and interface. When the number of signals is large, it is more convenient to define an interface or a signal list to use for each direction of the connector.

By default a connector has no name, it is non-delayed and it is bi-directional. It is possible to control the properties for a connector line from the shortcut menu on the connector line.

Symbol

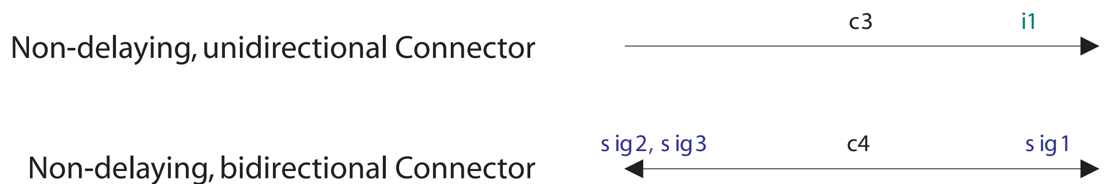


Figure 90: Connector types

A connector line specifies the communication path between two end points, for example ports attached to part symbols, to behavior symbols or to the frame of the diagram.

- If necessary connectors are omitted, some code generators may be able to create them implicitly.

- You can re-direct and bi-direct a connector from the shortcut menu.
- When you re-direct a bi-directional connector the signal list areas change places.
- The name of the connector is optional.
- The lists of interfaces, signals etc. associated with a connector are optional.

The structure of an active class can contain either explicit or implicit connector lines or both. Explicit connectors are visible while implicit connectors are invisible and cannot be referenced.

Implicit connectors are calculated from all matching realized and required interfaces on:

- Ports on parts contained in the containing class,
- Ports of the containing class,
- Behavior ports of the containing class.

Note

If a port has explicit connectors no implicit connectors will be connected to the port.

Syntax

The line contains two (uni-directional connector) or three (bidirectional connector) editable text fields.

The center field specifies the name of the connector and the field placed at the end of the line specifies the signal list area. There is one signal list area for each arrowhead in the line. The signal list areas may be empty.

Stereotypes applied to the connector line are visible in a non-editable text field, positioned above the name field.

Example 39: Connector signal list

```
i1,i2,sl1
```

Signal lists and interfaces

It is possible to draw a connector with signal lists to a port. In this situation the following applies:

- When there are no signals or interface given on any of the signal lists the information on the connected ports is used to deduce the signals and interfaces.
- When there are any signals or interfaces given on the signal lists associated with a connector then all transported signals and interfaces must be mentioned.

A shortcut menu choice for connector lines in composite structure diagrams **Show All Signals** is available. This fills the signal list text fields with signals and interfaces taken from the attached ports.

- Existing signals and interfaces will not be removed from affected signal lists.
- Only signals and interfaces not already existing are added to the signal list.
- The union of signals and interfaces found in the two attached ports is used. It is thus enough for a signal or interface to appear in one port, for the signal to show up in a signal list.
- If a signal is realized or required determines which signal list the signal will be put in.

Part communication

Normally communication between parts is explicitly modeled with ports and connector lines between ports.

It is not necessary to explicitly model communication if it is unambiguous, that is if the classes for the parts in the diagram have defined ports that can be connected in only one possible way.

It is allowed to connect a connector directly to a part symbol. The behavior is that an unnamed port is created, attached to the part and the connector is connected to this part. This is allowed both when creating a connector and when reconnecting an existing connector. This port is not deleted if the connector line is deleted, the port must be manually deleted if it is not needed in the model anymore.

Behavior port

Even if an active class has structure, that is has parts, it may also have its own behavior, expressed as a state machine. This behavior can be referenced in the Composite structure diagram using a behavior port.

The main purpose of the behavior port is when defining Connectors between a Part of the Active Class and the Behavior of the Active class; in this case it must have a Behavior Port.

It is possible to attach connectors to a behavior port in the same way as for Ports on Parts in order to define the communication interface of the state machine. It is allowed to have several behavior port in one diagram; in this case, they all refer to the same underlying behavior.

Symbol

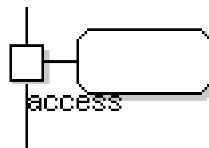


Figure 91: A behavior port

The Behavior port symbol specifies a reference to the unique state machine of the defined class.

- There can be several behavior port symbols present in one diagram.
- There is no text field in the symbol.

Behavior ports looks like ordinary ports in class diagrams. The appended behavior information is only shown in architecture and state machine diagrams.

Hint

You can add the behavior symbol to a composite structure diagram in two ways. Either by adding a port symbol to the diagram or by dragging an existing port from the Model View browser to the composite structure diagram. In both cases you also have to choose the command behavior port from the shortcut menu, or set this property using the Properties Editor.

Relationships

Dependency

The Dependency relationship in composite structure diagrams is used between parts, to show that one part is dependent of another. One common use is to indicate a create dependency between parts, that is that instances of one part can create new instances of another part.

Component Modeling

Component modeling is about identifying key Component of a system and model their Interfaces and Relationships.

The key focus when modeling components is to enforce strong encapsulation by hiding the implementation details inside a component and only expose a small set of well defined interfaces.

Weak coupling, i.e. minimized dependencies between different components, is another design principle often applied in component modeling.

Component diagram

A component diagram describes the static structure of a system through a set of Components, their Relationships and their Realized interfaces and Required interfaces. Other model elements like Classes and Artifacts can also be shown in a component diagram to illustrate their relationships with the components.

Example

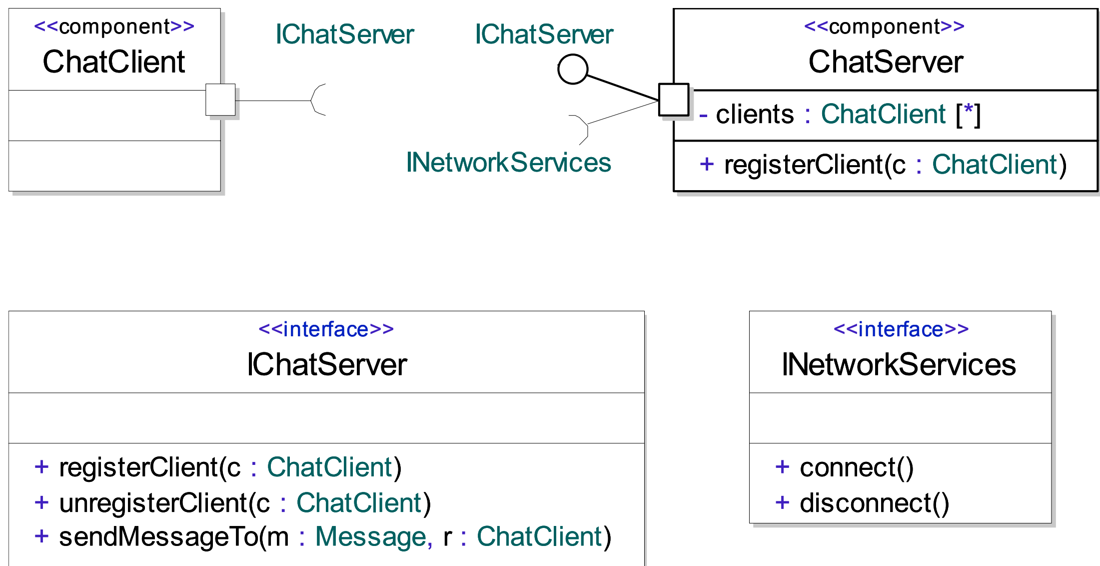


Figure 92: Component diagram

Model elements in component diagrams

The following elements are found in component diagrams

- Component
- Artifact
- Class
- Interface
- Port
- Realized interface
- Required interface
- Relationships

See also

"Class diagram" on page 199

Component

A component is a small part of a system that is well encapsulated and provides a well specified service.

The service provided by a component is specified through its Realized interfaces. A component should only be accessed through them. The component may also be dependent on other services; this is specified through its Required interfaces.

The implementation of the component, i.e. its behavior and architecture should not be exposed to clients. When only the Interfaces are exposed, one component can easily be substituted with another one, with a completely different implementation, without affecting the client.

The differences between a Class and a component in UML is minimal, and they can be used interchangeably. Anything that can be done with a class can also be done with a component. When using components though, the design principles outlined above should be adhered to.

Symbol

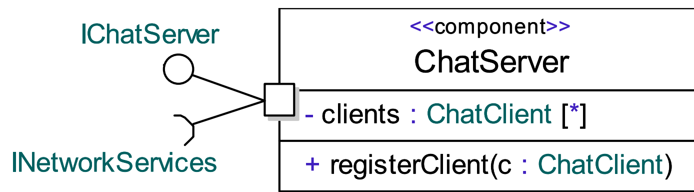


Figure 93: Component with a port and realized/required interfaces

The component symbol is identical to the Class Symbol, with the `keyword` «`component`» added to the top.

See also

"Class" on page 200.

Relationships

The following relationships can be used in Component diagrams:

- Association
- Aggregation
- Composition
- Dependency
- Generalization
- Realization
- Manifestation
- Containment

Activity Modeling

Activity modeling is about using Activity Diagrams to model behavior by organizing it into small behavioral units and to describe the control and data flow between these units. It can also describe how these units are distributed across a system.

Activity modeling can be used at an abstract level for business modeling or at a very low level to model behavior at action code level. It is particularly useful for the modeling of asynchronous and distributed systems.

See also

“Scenario Modeling” on page 162

“Behavior Modeling” on page 265

Activity Diagram

An activity diagram describes how a behavior is divided into small behavioral units, Action Nodes, and controls the execution sequence between them using Activity edges and control constructs such as Decision, Fork and Activity Final nodes.

Object Nodes and Pins are used to describe how objects and data are passed between the different actions.

Activity Partitions are used to group related actions into groups, for example by function or by owner.

Activity diagrams are similar to flowcharts.

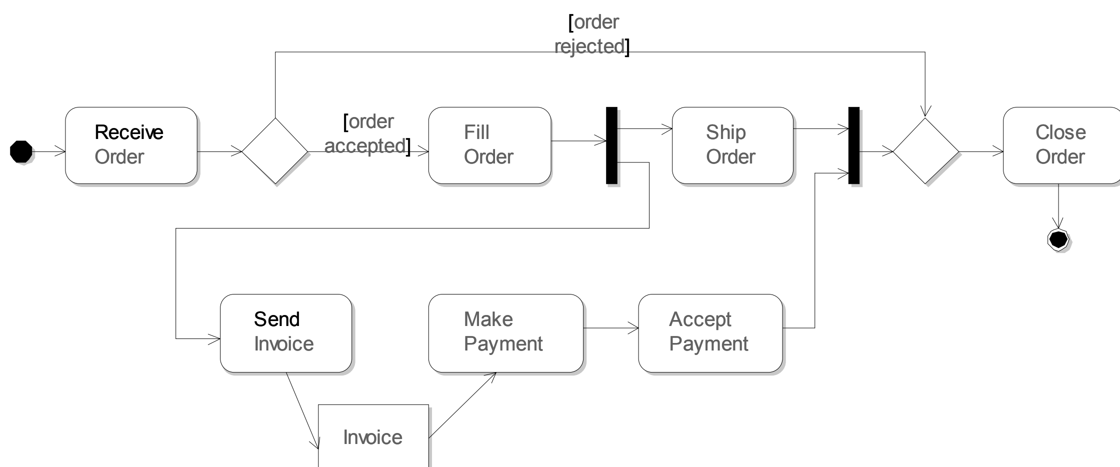


Figure 94: Activity diagram

Create an activity diagram

Activity diagrams can be included in packages, classes, use cases, operations and activities.

1. Select the entity where to create the activity diagram in the Model View.
2. From the shortcut menu select **New** and then **Activity diagram**.

Flow orientation

Horizontal is default.

When horizontal orientation is chosen for activity diagrams, it is easy to create horizontal activity flows:

- Line handles are placed on the middle of the right symbol border.
- New fork/join symbols have a vertical orientation as default. (Already existing fork/join symbols are not changed when the default orientation is changed.)
- New partition symbols have as default a header size where the height is larger than the width. (Already existing partition symbols are not changed when the default orientation is changed.)

It is possible to change the default flow orientation while appending symbols in a flow by pressing SHIFT + CTRL together.

Activity symbols from model elements

Copying information from the Model View to an activity diagram using drag-and-drop is possible. For example it is possible to drag-and-drop an operation node to create an activity symbol which references this operation. The same can be done with interaction nodes, state machine nodes and use case nodes.

Note

Actions must be selected (via the shortcut menu) for an existing activity symbol in order for the reference to be visible before dragging an activity node to the activity symbol.

Model elements in activity diagrams

The following elements are found in activity diagrams:

- Initial Node
- Action Node
- Object Node
- Decision
- Merge
- Fork

- Join
- Connector
- Accept Event
- Send Signal
- Accept Time Event
- Activity Final
- Flow Final
- Activity Partition
- Pin
- Relationships.

Activity

An activity is a Signature representing the behavior of a use case, operation or any other entity that can have a behavior. An activity focuses on breaking down the behavior into small behavioral units, Action Nodes, and control the execution of these units based on a token flow model. The implementation of an activity is typically described by an Activity Diagram.

Symbol

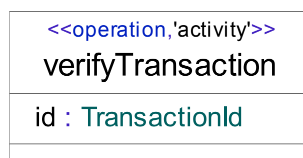


Figure 95: Activity

Syntax

An activity symbol is based on the Operation symbol. It has an editable field for the name of the activity, and a compartment for Parameters of the activity.

Stereotypes applied to the activity are visible in a non-editable text field, positioned above the name field.

Activity implementation

An activity implementation is the Implementation of an Activity signature. It contains the Activity Diagrams and a set of activity nodes connected by Activity edges. An activity implementation is normally created implicitly when creating an Activity Diagram.

Token flows

The execution semantics of an activity implementation is based on a token flow model.Tokens flow from one activity node to other activity nodes through connected Activity edges. There are two kinds of token:

- Control token
- Data token (also known as object token)

An activity edge can transport both kinds of tokens. When a control token is transported across the edge it represents a control flow, and when a data token is transported across the edge it represents a data flow. A control flow is an activity edge with any activity nodes linked to its ends, except object nodes. A data flow is an activity edge with an object node linked to at least one of the edge's ends.

Control tokens constitute a state of logic control of a modeled system, whereas data tokens are needed to represent a state of data units which are flowing through a modeled system.

An activity edge is a directed edge which is linked to action nodes, control nodes, object nodes, pins or connectors. The direction of an edge represents the direction of the flow. The semantics of an activity edge depends on its target and source nodes.

When an activity is invoked (called) its activity implementation starts its execution by placing a control token on each Initial Node it contains. These tokens then flow downstream across outgoing activity edges and collect on the incoming activity edge ends of those activity nodes to which these edges are connected. An activity node is allowed to start executing as soon as its **input condition** is fulfilled. Different kinds of activity nodes have different input conditions, but a typical condition is that there must be a token available on each incoming activity edge end before execution can start. When the activity node has completed its execution it delivers a token (of some kind) on all outgoing activity edge ends. These tokens eventually reach other activity nodes, and the procedure is repeated.

The activity implementation continues to execute as long as there are tokens flowing in it. If none of the activity nodes in the activity implementation has its input condition fulfilled, no tokens will flow, but the activity implementation is still in an executing mode, that is control will not be returned to the caller of the activity. Only when a special activity node, the Activity Final node, is executed will the entire activity implementation finish its execution and control is returned to the caller of the activity.

Initial Node

An initial node specifies a starting point for the control flow in an activity implementation. When an activity is invoked and its implementation begins executing each initial node of its implementation receives a control token.

Note that an activity implementation can have any number of initial nodes, meaning that multiple control flows can be started. Note also the it is not required to have any initial nodes at all. Flows can also start from a Pin, an Accept Event and an Accept Time Event.

An initial node may not have any incoming activity edges, and therefore has no input condition. It executes as soon as it receives a control token and then offers this token to outgoing edges.

Symbol

Figure 96: Initial node

Action Node

An action node is a piece of executable functionality in an activity. The behavior of an action node can be specified in many ways, for example using an Activity, Operation, or a State machine. But it is also allowed not to associate a behavior to an action node. This can be useful at early stages of development, when the details of the behavior is not known.

The behavior of an action node, if any, can either be defined inline in the action node, or it can be referenced from the action node. Inline defined behaviors are appropriate in order to specify composite hierarchical activity implementations. Compare with Composite state. Referenced behaviors are appropriate in order to reuse the same behavior for multiple action nodes in

a model. When using a referenced behavior it should normally be an activity, but in general it is possible to refer to any operation. The referenced behavior may in turn have an implementation. For example, a referenced activity may have an activity implementation.

The input condition for an action node is fulfilled when a token is available on all incoming activity edges. It then consumes these tokens and starts its execution. When it has finished its execution, control tokens are offered on all outgoing edges.

Avoid execution deadlocks

As long as the input condition for an action node is not fulfilled it cannot execute. To avoid deadlocks in the execution it is therefore very important to understand the semantics of Token flows in an activity implementation. As an example of a common misunderstanding, consider the activity implementation in Figure 97 on page 251 below.

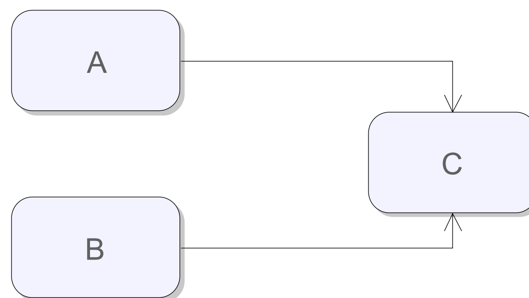


Figure 97: Control flow between action nodes

In this example we have three action nodes A, B, C and two control flow edges from A to C and from B to C respectively. Here C can be executed only when *both of these* edges have a token. If only the edge from A to C has a token, then the C node will wait for a token on the edge from B to C node. It is important to understand that nodes are collected on edges not on nodes.

If we instead would want the C node to execute when *at least one* of these edges have a token we should insert a Merge node between them as shown below.

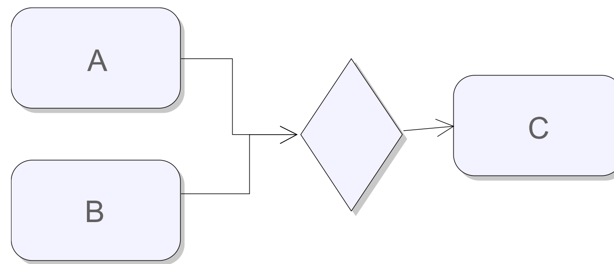


Figure 98: Using a merge node

Pins

If an action node has a behavior it can have Pins representing Parameters to its behavior. It is significant if a token reaches an action node directly via an incoming activity edge, or indirectly via an attached pin. In the former case the action node will be executed when its input condition is fulfilled. In the latter case, however, the action node itself does not execute. Instead the token flows into the behavior implementation in a “streaming” way, so that execution of the behavior implementation starts with a data token on a pin, rather than with a control token on an initial node. It is possible to combine these two mechanisms, by letting both a control token flow into the action node and data tokens flow into its pins. That is a common way of designing when the behavior needs data for its execution. It then obtains input data on the input pins, and a control token to control when execution shall begin. Before finishing the execution by executing an activity final node, output data is typically offered as data tokens placed on the output pins.

For more information about pins see Pin.

Symbol

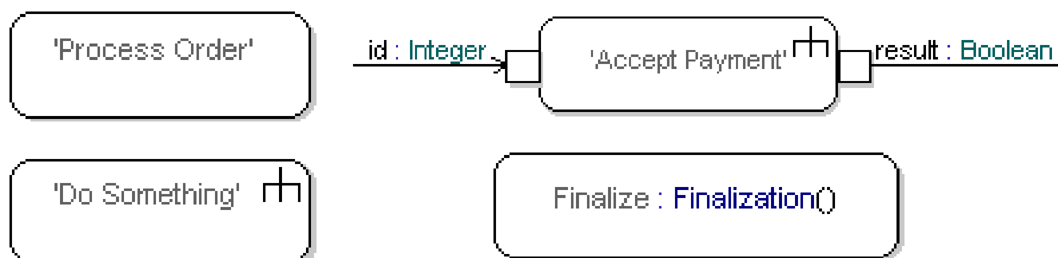


Figure 99: Action nodes with and without pins, and with and without a behavior (inline to the left and referenced to the right)

A shortcut menu choice **Actions** is available. When checked the a text field is added for action code. The default is to not display the Actions text field.

A shortcut menu choice **Partition Reference** is available. This command will display the text field for Partition Reference above the symbol name field. The default is to not display the Partition Reference text field.

A shortcut menu choice **Show All Parameters** is available. The Show All Parameters command will make all Pin/Parameter symbols visible for the current selection.

Syntax

The action node symbol may have an informal name. If it references a behavior the signature of the behavior appears after a colon. If it has an inline behavior a "rake" symbol is shown in the upper right corner of the symbol.

Activity partitions in which an action node is explicitly contained, may be specified in a separate text field above the name field. The syntax is a comma-separated list of references to activity partitions enclosed in parenthesis.

Stereotypes applied to the action node are visible in a non-editable text field, positioned above the activity partition reference field.

Object Node

An object node represents an instance of a classifier, for example a Class, participating in the flow. The instance and its values is available for use by the activity.

The input condition for an object node is that there must be a token on each incoming activity edge before it may execute. Execution of an object node simply means that a data token is placed on each outgoing edge. The type of the data token is the type of the object node, that is the classifier.

An object node does not specify how the output data is obtained. To do that an Action Node node with an output pin can be used instead. The behavior of the action node then specifies how to compute the data.

Symbol

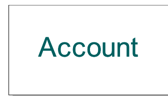


Figure 100: Object node

Syntax

The object node symbol has one text label containing the name of the classifier it represents. It is also possible to give an informal name for the object node. The syntax is then `<name> : <type>`.

Stereotypes applied to the object node are visible in a non-editable text field, positioned above the name field.

Decision

A decision node is a control node used in a flow to select one out of several outgoing flows based on guard conditions. A decision node has one incoming edge and multiple outgoing edges, each with a guard.

When a token arrives at the incoming edge of a decision node, the guards of the outgoing edges are evaluated. The order in which the guards are evaluated is not defined by UML, except that any ‘else’ guard is evaluated last. It is therefore recommended to specify guard conditions that are mutually exclusive. At most one of the guards may be an “else” guard. This guard condition is fulfilled if no other guard condition is fulfilled.

The input token will be placed on the first edge that is encountered for which its guard condition is fulfilled. If no such edge is found the token is consumed by the decision node. This is typically an exceptional situation which is best avoided by using an ‘else’ guard on one of the edges

Note

The current implementation of the activity execution semantics in the Activity Simulator only supports informal decisions and decision answers. When such a decision node is executed the Model Verifier will prompt interactively for which outgoing edge to select. This is a useful feature at early stages of development, since it allows activities to be simulated before the exact guard conditions are known.

Symbol

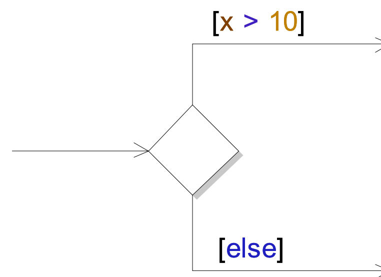


Figure 101: Decision node

When formally defined the guard conditions shall evaluate to boolean expressions. Any visible variables, e.g. local variables of an activity implementation, may be used in the guard condition. The keyword `else` is used in a guard to indicate that the edge is selected if none of the other guards evaluates to `true`.

To merge back multiple outgoing decision flows into a single flow, use a Merge node.

Note

The Decision and Merge nodes share the same symbol in the symbol palette of the activity diagram editor.

Merge

A merge node is a control node used to bring together multiple flows into one. Whenever a token arrives at one of the incoming edges, it is relayed onto the outgoing edge. Unlike Join, it is not a synchronization of the incoming flows.

Symbol

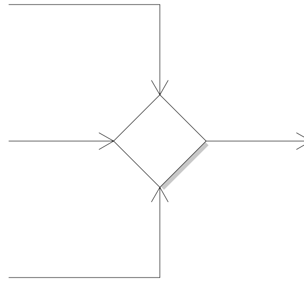


Figure 102: Merge node

Note

The Merge and Decision nodes share the same symbol in the symbol palette of the activity diagram editor.

Fork

A fork node is a control node that splits one flow into multiple concurrent flows. Whenever a token arrives at the input edge it will be copied, and one copy will be placed on each outgoing edge. A fork node is thus a means for introducing parallelism in an activity model.

To join multiple concurrent flows back into one single flow, use the Join node.

Symbol

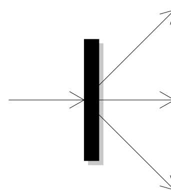


Figure 103: Fork node

Note

The Fork and Join nodes share the same symbol in the symbol palette of the activity diagram editor.

Join

A join node is a control node used to join, or synchronize, multiple concurrent flows back into one single flow.

The input condition for a join node is that there must be a token available on all incoming edges. When that condition is fulfilled tokens are placed on the outgoing edge according to the following rules:

- If all input tokens are control tokens, then one single control token is placed on the outgoing edge.
- If some of the input tokens are data tokens, then all these tokens, but only these, are placed on the outgoing edge.

Note

The current implementation of the activity execution semantics in the Activity Simulator does not follow this rule. Instead it is the token that arrives last to the join that will decide which kind of token that is placed on the outgoing edge.

To fork a single flow into multiple concurrent flows, use the Fork node.

Symbol

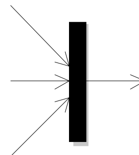


Figure 104: Join

Note

The Join and Fork nodes share the same symbol in the symbol palette of the activity diagram editor.

Connector

Connector nodes are used as a graphical short hand to simplify drawing of complex flows. An Activity edge can end in a connector node and be continued at another connector node with the same name. This can be used to split an activity implementation specification over multiple activity diagrams.

A connector node may have many incoming edges, but at most one outgoing edge. Semantically a connector node is equivalent with a Merge node. A token that arrives at an incoming edge of a connector node is relayed onto its outgoing edge. If a connector node does not have an outgoing edge it is semantically equivalent with a Flow Final node.

Symbol

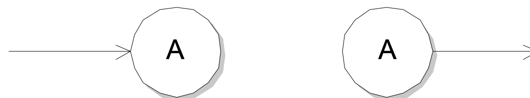


Figure 105: Connector node

Syntax

The connector node symbol has one text label containing the name of the connector node.

Accept Event

An accept event node is used to indicate waiting for a specific event, typically a Signal. When the specific event is received, the flow continues by placing control tokens on all outgoing edges.

Semantically an accept event node is equivalent with an Action Node node, with a behavior that waits for the event to be received.

Data passed on with the event can be used later in the flow by using output Pins from the accept event node. An accept event node may not have any input Pins.

The accept event action is similar to an Signal Receipt (Input) in a State machine.

Symbol

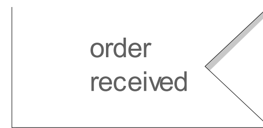


Figure 106: Accept event node

Send Signal

The send signal node is used to create an instance of a Signal and send it. It is similar to the Signal sending action (output) in a State machine.

Semantically a send signal node is equivalent with an Action Node node, with a behavior that sends the signal.

A send signal node may have input Pins providing actual arguments for the formal parameters of the signal to send. It may not have any output Pins.

Symbol

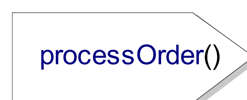


Figure 107: Send signal symbol

Accept Time Event

An accept time event is a special version of the Accept Event node. It is used to indicate waiting for a specific time event, typically a Timer timeout or an absolute time value. When the specific time event is received, the flow continues by placing control tokens on all outgoing edges.

Contrary to an Accept Event node, an accept time event node may not have any Pins. In order to wait for a timer with parameters, use an Accept Event node instead.

Symbol

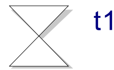


Figure 108: Accept time event

Activity Final

The activity final node indicates the termination of an activity. When a token reaches an activity final node, all flows of the activity are terminated, and the execution of the activity is completed. Control is returned to the caller of the activity.

An activity final node may have an arbitrary number of input edges, but no output edges.

To terminate a single flow of an activity, use Flow Final nodes.

Symbol

Figure 109: Activity final

Flow Final

The flow final indicates the termination of a single flow in an activity. Only that particular flow is terminated, not the entire activity. There might still be other ongoing flows (compare Fork) in the activity.

Tokens received by a flow final node will be consumed by it. A flow final node may have an arbitrary number of input edges, but no output edges.

To terminate the entire activity, use Activity Final nodes.

Symbol

Figure 110: Flow final

Activity Partition

An activity partition, sometimes called a swimlane, is a grouping mechanism used to group related Action Nodes to each other. They provide a way of splitting an activity diagram into different sections to make it easy to see which section that performs a certain activity, and how data flows between the different sections.

For example, in business modeling, the different subdivisions of a company can each be represented by a partition. Another example is to let each partition represent a thread in a real-time operating system. The diagram would then show how the actions of a system are distributed among threads.

An activity partition may have a type, which typically is a Class. This expresses a constraint that those instances that perform the actions of the activity partition, must be instances of that type. The activity partition may further constrain performed actions by specifying one particular instance, which then must perform the actions. It may also specify an Attribute, which then must contain the instances that perform the actions.

Symbol

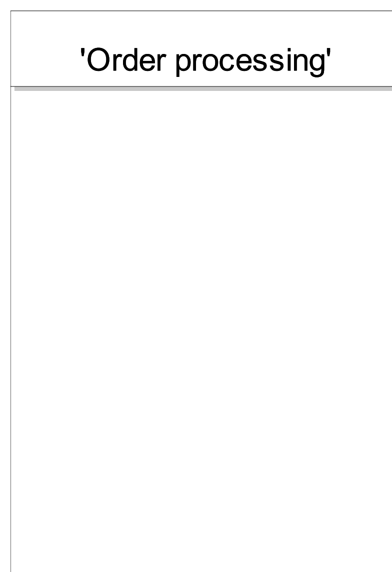


Figure 111: Activity partition

A constraint concerning type, instance or attribute for the activity partition is specified in a label just below the name label. The syntax is the same as is used for a Lifeline.

Action Node node symbols that are graphically contained in an activity partition symbol represent actions that belong to that activity partition. It is possible for an action node to belong to more than one activity partition. This can happen when using activity partition symbols that are rotated, so that the intersection of two activity partition symbols contains the same action node symbol. However, it is not possible to accomplish involvement in more than two activity partitions this way, because an activity diagram only has two dimensions. In order to specify that an action node belongs to more than two activity partitions an explicit list of included partitions may be specified for the action node. If an action node has an explicit list of activity partition references it overrides the implicit reference that can be deduced from the graphical position.

Example 40: Implicit and explicit activity partition references

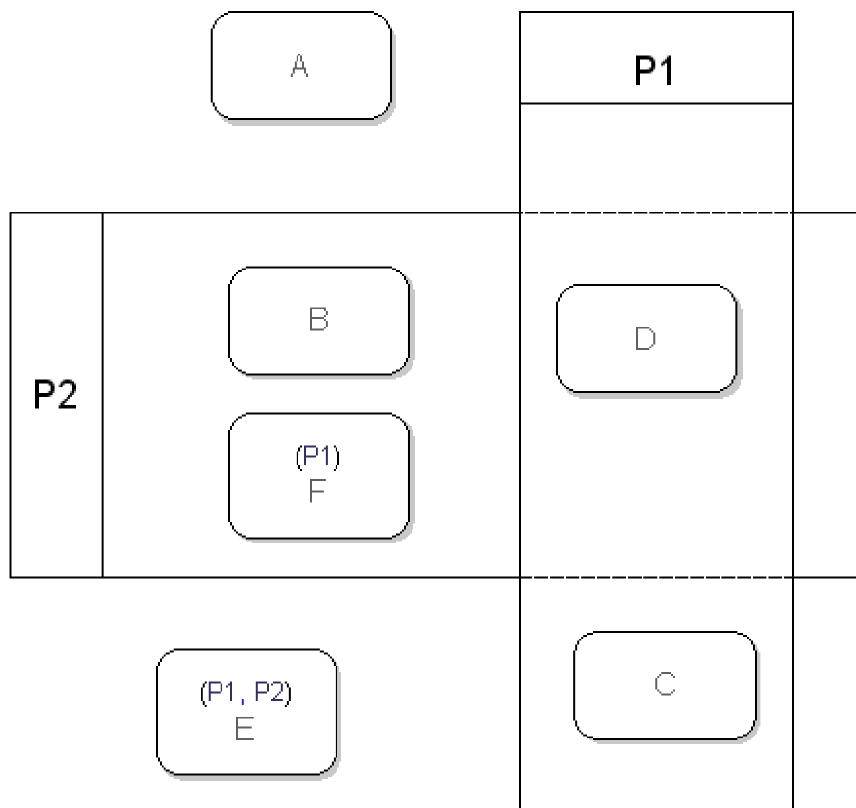


Figure 112: Action nodes referring to activity partitions

The action nodes above use both implicit activity partition reference (deduced from the graphical position of the action node symbol) and explicit activity partition references.

A does not belong to any partition.

B belongs to partition P2 (implicit reference)

C belongs to partition P1 (implicit reference)

D belongs to partition P1 and partition P2 (implicit reference)

E belongs to partition P1 and partition P2 (explicit reference)

F belongs to partition P1 (explicit reference)

Partition symbol as Dimension Specification symbol

When there are several rows of partition symbols, the partition symbol in the top row may be used as a dimension specification symbol. The partition symbol has a shortcut menu choice for Dimension. A partition has a plain text in the main label while a dimension has an italic font in the main label.

It is possible to use both horizontal and vertical dimensions at the same time.

Pin

A pin represents a parameter of the behavior of an Action Node node, and are used for passing data to and from that behavior. They can be seen as Object Nodes for inputs and outputs to actions.

Pins that have incoming edges input data to the behavior, and are therefore called input pins. Pins that have outgoing edges output data from the behavior, and are consequently called output pins. The direction of the Parameters represented by the pin should match how edges are connected to the pin. For example, an input pin should only have incoming edges, and the corresponding parameter should have “in” direction.

The semantics of executing a pin is the same as for an Object Node. Hence, execution places a data token on each outgoing edge, and the type of these data tokens is the type of the parameter represented by the pin.

A pin may be streaming or non-streaming. In the streaming case the pin can execute to produce output data tokens even when the behavior of the Action Node node is executing. In fact there is no connection between the presence of tokens on streaming input pins and the condition for when the behavior of the Action Node node is invoked. In the non-streaming case, however, the behavior will not execute until tokens are available on all input pins.

Note

The current implementation of the activity execution semantics in the Activity Simulator only supports streaming pins. However, a non-streaming pin can be emulated by combining a streaming pin with a Join node in the activity implementation of the Action Node node behavior. The join node then has two incoming edges; one from the pin on which the data token will arrive, and one from the Initial Node on which the control token will arrive when the behavior is executed.

Symbol

Figure 113: Pin symbol

Syntax

The pin text has the same syntax as Parameters, i.e. `name : Type`.

Relationships

Activity edge

An activity edge is used to connect nodes in an activity implementation. It enables the flow of control and data tokens between the two connected nodes.

An activity edge is always directed, meaning that a token only can flow in one direction over an activity edge. The direction of an edge represents the direction of the flow. An activity edge can transport both kinds of tokens. When a control token is transported across the edge it represents a control flow, and when a data token is transported across the edge it represents a data flow.

An activity edge can have an informal name describing the flow it represents.

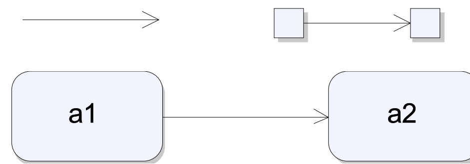


Figure 114: Activity edge

Behavior Modeling

In order to obtain an executable model, the detailed behavior of operations and active classes must be specified. This is done during behavior modeling, an activity that usually takes place at the end of the design phase.

A behavior specification may contain states (that is a State machine implementation), or it may be stateless (that is an Operation body). In either case there are two ways to describe the behavior:

- As a state machine in a State machine diagram

For implementations that contain states, the graphical form (State machine diagram) is often to be preferred, while for simple implementations of operations it could be enough with a textual description of the actions that constitute the Operation body.

State machine diagram

A State machine diagram visualizes a State machine. There are two different styles of drawing state machine diagrams supported. They are described and exemplified below. It is possible to combine the two styles.

State-oriented view

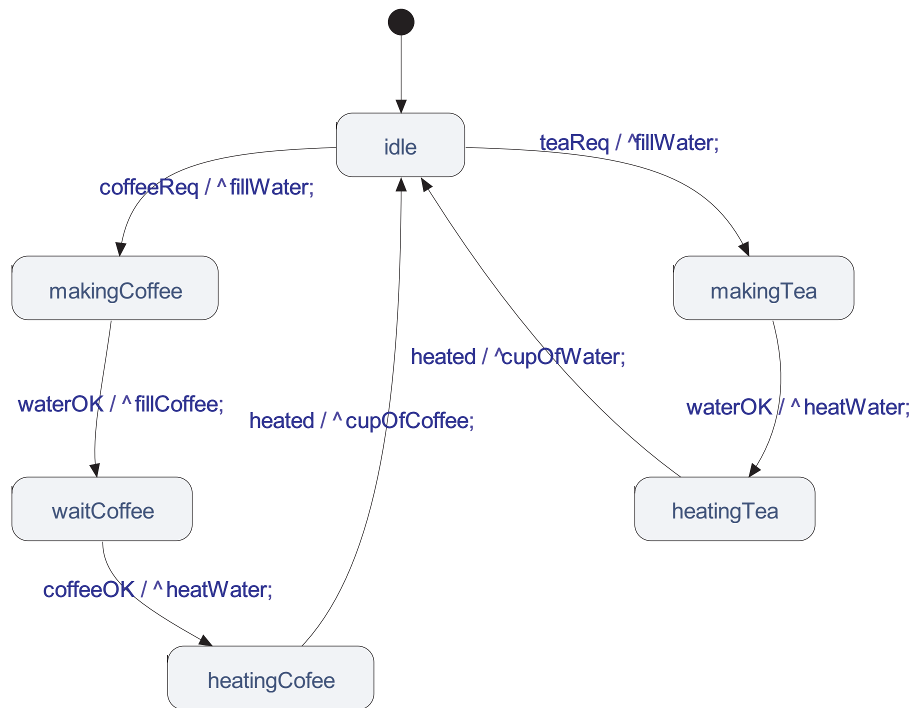


Figure 115: State-oriented view of a state machine

The state-oriented view of a state machine gives good overview of a complex state machine but is less practical when focusing on the control flow and communication aspects of a specific set of transitions. For this reason, it is also possible to describe the state machine in a transition-oriented way, with explicit symbols for different actions that can be performed during the transition.

Transition-oriented view

Figure 116: Transition-oriented view of a state machine

Create a state machine diagram

State machine diagrams can be included in classes and operations (including use cases).

1. Select the entity where to create the statemachine diagram in the Model View.
2. From the shortcut menu select **New** and then **State machine diagram**.

State machine

A UML state machine is a finite state machine extended with data and signal handling. The basic elements of a state machine is the state and the transition. In a model based on the state machine paradigm, execution is carried out with a certain state as the starting point and a triggering event that causes a transition to be executed. In the transition, actions can be carried out. At the end of the transition, a new state is entered. The state machine will be idle in this state until a new triggering event that may start a transition occurs. An alternative way to end a transition is to stop the entire state machine (active class).

Hint

State machines are most simply created either by right-click of a class in the Model View and choosing New->State machine diagram in the shortcut menu or by opening the Create Presentation dialog.

Symbol

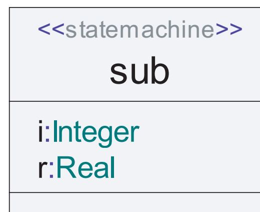


Figure 117: State machine

Syntax

The symbol contains two editable text fields:

- Class Heading
- Parameters

(The Operation field is empty.)

The Parameters field contains the formal parameters of the state machine. These are used for:

- Passing values to an active class instance upon creation.
- Passing values to a composite state when entering it.

State

A State represents a situation in a State machine where the containing object is waiting for an event that will trigger a transition to another State. This situation may have a static condition (if the state does not have substates); in this case the state machine is inactive while in the state. The situation can also be dynamic in the sense that there can be state machine behavior hidden in substates of the state.

Symbol

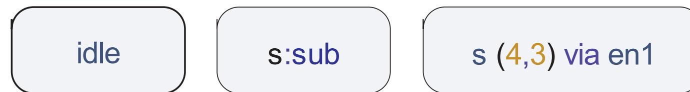


Figure 118: State

The State symbol references one or more states and acts as source and/or target for transitions leading from or to this state (or set of states).

Syntax

- Simple state:

```
State1
```

- State with state list:

```
St1,st2
```

- State with asterisk state, including list of not included states:

```
*(st1,st2)
```

An asterisk state is a shortcut that refers to all states defined in the current state machine except the states mentioned in the list following the ‘*’ symbol.

Since state machines are hierarchical a state may contain a sub-state machine. This is indicated in the syntax by giving the name of the state machine after a colon following the name of the state as in Figure 119 on page 269

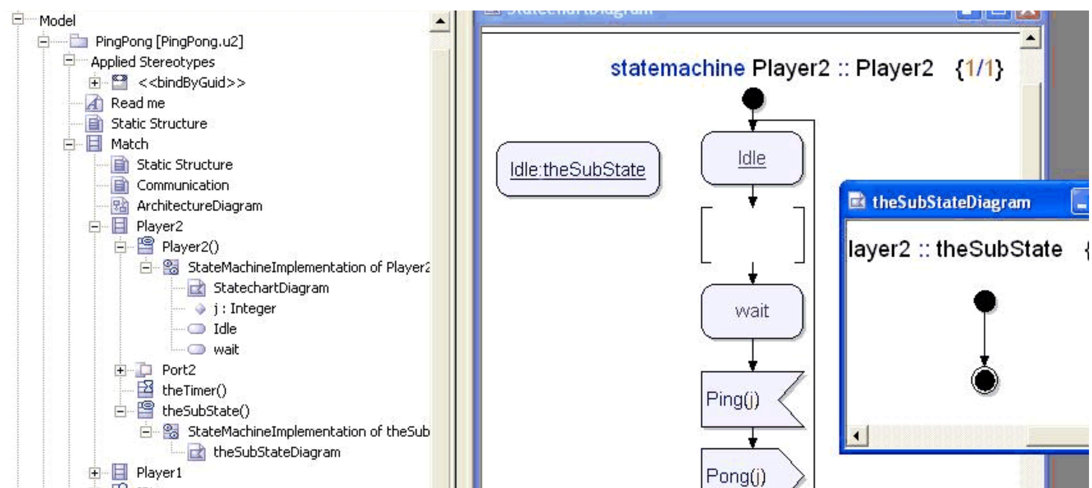


Figure 119: Sub-state reference

The `<state>:<state machine>` syntax may only be used for state symbols without incoming lines (i.e. the state symbol should not be a “nextstate”). It is a syntax error if a state symbol with the label `s:myStatemachine has` incoming lines.

Note

The state symbol may not contain a list of states or an asterisk state definition if there are transitions that has this particular state symbol as target. State lists and asterisk states may only specify the source of transitions, not the target of transitions.

If a state has a substate state machine and this state machine has an entry point then the entry point may be indicated in the state symbol. This may only be used if there only exist one transition that has the state symbol as its target state.

Example 41: State with via clause

State `St1` containing a ‘`via`’ clause that determines the entry point in a substate state machine:

```
St1 via entry1
```

If a state has a substate this is indicated in the state symbol by a “rake” in the upper right corner, symbolizing a split flow, see Figure 120 on page 270.

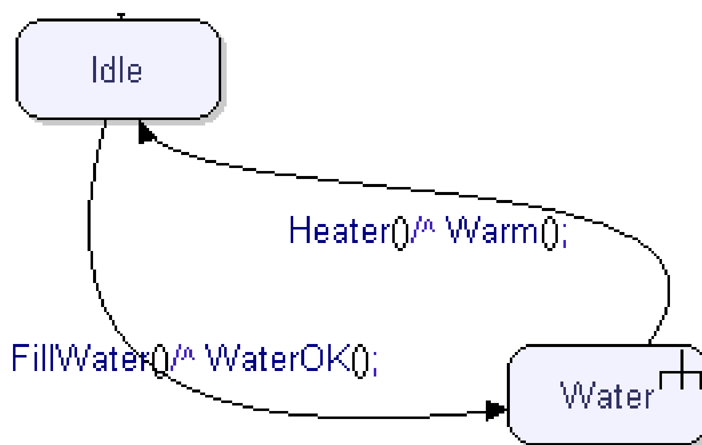


Figure 120: State with substate

Since the State symbol can be used for both defining a state and referencing a state (the target state of a transition), it is possible to let the symbol act as both an end point for a transition and the starting point for a new transition, thus chaining the transitions. This makes sense if using a state-oriented layout of the state machine. When using a transition-oriented layout it is however good practice to avoid this, and always separate the transitions, as in the example above, for the sake of readability.

If a state acts as source for many transitions, it is allowed to specify these in different diagrams in order to improve readability. Thus the State symbol is a partial definition of the state.

If the same transition is valid in several states, it is possible to refer to several states from the State symbol.

See also

"Composite state" on page 297

Transition

A Transition is a sequence of actions that are executed when a State machine changes the active state.

The syntax used for transitions falls into two different categories depending on if a state-oriented or transition oriented syntax is used. The state-oriented transition syntax is described in "Simple transition" on page 290, the transition-oriented syntax is described by a set of trigger symbols for starting the transitions and then a set of action symbols that describe the transition details.

The different trigger symbols correspond to what event that causes the transition to be initiated. Based on this, different kinds of transitions can be distinguished:

- triggered transitions
- guarded transitions
- labelled transitions
- initial transitions

A triggered transition is characterized by the trigger that is associated with the transitions. Typically this trigger is defined by the specific signal, but it may also be defined by for example a timer or by an operation. Triggered transitions are described in more detail in section "Signal Receipt (Input)" on page 274

A guarded transition is characterized by the fact that it is *not* triggered by a specific event. Instead it is triggered either by a certain condition (Guard) that can be true or false.

Labelled transitions are not real transitions in terms of describing a state-to-state behavior. Instead they are used to decompose a transition into two (or more) parts that can be described on two different pages in a diagram. Junction is also a related construct to labelled transitions used to divide flows.

The Initial Transition (Start) is the transition that will be executed directly when the state machine is created.

A transition always ends with the state machine entering a state, with a stop, with a return or with the transfer of control to another transition.

Guarded transition

A Guarded Transition may or may not have a trigger.

If the guarded transition has a trigger, the evaluation of the expression will be done after the triggering event has happened. If the expression evaluates to true, the transition is fired. If the expression evaluates to false, the state machine will remain in the state and the signal that caused the triggering event will be kept in the signal queue.

See also

"Save" on page 287

History nextstate

The History nextstate is used at the end of a transition to return to the last visited state.

The symbol can be used to end both simple transitions and flow line (detailed) transitions.

Shallow history

By default, the History nextstate is **shallow**. This means that when a nextstate with History is interpreted at the end of a transition, the next state will be the one in which the current transition is activated.

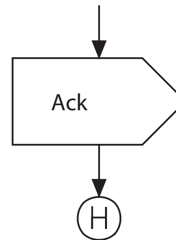


Figure 121: Shallow History nextstate

History nextstate can also be expressed with a normal nextstate, using a hyphen instead of a name in the symbol.

```
nextstate -;
```

Deep history

It is possible to make the history nextstate **deep**. This means that similar to the shallow history, the next state will be the one in which the current transition is activated. This will apply recursively to all levels of substates of the entered state.

Hint

You can make the history nextstate deep by selecting it and choosing the command ***Deep History*** *from the shortcut menu.*

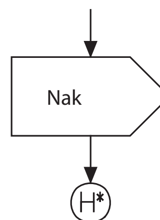


Figure 122: Deep History Nextstate

Deep history nextstate can also be expressed with a normal nextstate, using the following syntax:

```
nextstate ^-;
```

Examples

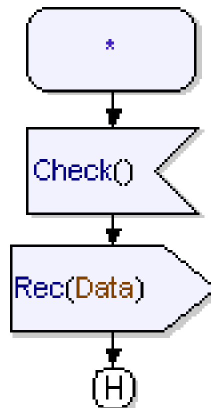


Figure 123: Shallow History Nextstate with an asterisk state transition

In the above example, the transition will end up in the state that was active when the transition was triggered.

Signal Receipt (Input)

The signal receipt symbol defines which signals that should trigger a particular transition.

The transition can be guarded by a guard expression that also is shown in the symbol.

Symbol

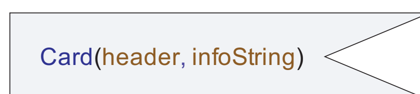


Figure 124:Signal Receipt

The signal receipt symbol receives a signal and must always be preceded by a State symbol. Together they define a transition.

Hint

You can flip the symbol horizontally from the shortcut menu.
When you delete the signal receipt symbol, the succeeding subtree is deleted as well.

If the same transition behavior should be invoked for several triggers in one state, it is possible to have a list of identifiers in the signal receipt symbol. This mechanism does not allow handling the parameters of each signal and all the signals will trigger the same transition that ends in one nextstate.

When receiving a signal, its parameters are normally stored in local variables. It is also allowed to ignore parameters.

The optional guard expression is defined after the trigger and is surrounded by square brackets.

Signal queue

A State machine is associated with a **signal queue** that stores the signals that are sent to the state machine in the order they arrive.

It is not necessary to specify a transition for every possible trigger in each state. Often it is possible to predict which signals that may arrive, from your knowledge about the application or domain you are modeling. If the next signal to consume from the signal queue is not handled in the current state, that signal will be thrown away. It is also possible to Save a signal temporarily.

Syntax

The following kinds may be referenced as triggers in a signal receipt symbol:

- Signal
- Timer
- Operation.

Example 42: Simple signal receipt

```
s1( i )
```

Example 43: Signal Receipt with several triggers

```
s1(i), myTimer, s3
```

Example 44: Signal Receipt with virtuality

```
redefined input s1( i )
```

Example 45: Asterisk signal receipt

It is allowed to specify that all triggers may invoke the transition. This is done by using an asterisk to denote all visible triggers.

```
*
```

Example 46: Guarded signal receipt

```
s1 [ x>10 ]
```

Start

The Start symbol defines the starting point of a state machine or one starting point of a composite state. The start symbol thus defines the initial transition.

Symbol

Figure 125: Start

Syntax

The start symbol has one text field that can be used for:

- Referencing an entry point in a composite state

  ```
  Entry1
  ```

- Defining virtuality for the transition

  ```
  virtual
  virtual Entry2
  ```

Action

Actions are typically done in the Action symbol using a textual syntax. The available actions are:

- Local variable definition statement
- Empty statement
- Compound statement
- Assignment
- Action
  - Signal Sending (output)
  - New
  - Set
  - Reset
- Expression statement
- If statement
- Decision statement
- Target code statement
- While statement
- For statement
- Delete statement
- Try statement
- Terminating statement
  - Return
  - Break
  - Continue
  - Stop
  - Nextstate
  - Goto (join)
  - Throw

A few of these statements also have a graphical syntax, that is a dedicated symbol. The stop, return, decision and signal sending statements have distinct symbols that allow highlighting of important operations on the transition. It is of course allowed to use the textual syntax for these statements as well. The most important actions are described below.

Signal sending action (output)

The signal sending action in a transition allows to send signals to other State machines, the environment or within the same state machine. If the signal has parameters, expressions matching the parameter types should be provided. It is allowed to ignore parameters when sending a signal.

It is allowed to specify more than one signal at a signal sending, which will be handled as sending separate, consecutive signals.

Symbol

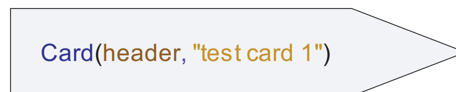


Figure 126: Signal Sending

The signal sending symbol sends a signal from a transition.

Hint

You can flip the symbol horizontally from the shortcut menu.

Signal addressing

There are several ways to direct a signal to a receiver or routing the signal, including:

- Omitting addressing
- Directing the signal as a method application on the receiver
- Signal Sending via port or interface

Each of these addressing mechanisms are described below. Direct addressing of a signal is expressed using period (“<receiver>.<signal>”) for the method application on a receiver.

Signal sending

No address or path is specified. The signal will be sent on one of the possible paths (that is a port / connector).

Receiver is this

If the context is a state machine or an operation of an active class, **this** means the state machine of the current active instance, that is the same as self.

If the context is an operation of a passive class, **self** should be used instead, to reference to the state machine of the current instance. In this context, **this** refers to the instance of the passive class

Signal sending via port or interface

A port identifier is given. The signal will be sent via this port

If an anonymous port that realizes exactly one interface is defined for the class the identifier can also be an Interface name. In this case it refers to the anonymous port.

Receiver is an attribute

Either a variable or attribute is given as destination. The type of the variable or attribute must either be an interface (signal sending via) or an active class (or the special type Pid defined in the RTUtilities package).

The attribute may also refer to one of the implicit attributes **self**, **sender**, **parent** or **offspring**.

Receiver is an expression

The expression must be typed by an interface or an active class (or the special type Pid defined in the RTUtilities package). This is a similar situation to when Receiver is an attribute. The difference is that more complex expressions can be given within the parenthesis, for example field or string extraction.

Examples

Example 47: Addressing mechanisms

No address or path is specified:

```
SuspendInd
```

Receiver is an implicit attribute:

```
sender.Ack(id)
```

Receiver is an attribute, signal carries parameters:

```
Bank.Card(carddata)
```

Receiver is a Pid expression (indexed array with Pid elements):

```
(myList[10].addr).Sig1
```

An interface (referring to a port):

```
Ack(id) via myInterface
```

All these addressing mechanisms have the following in common:

- If there is no alive instance of a state machine at the end of the communication path, the signal will be lost.
- If the destination references a state machine instance that has terminated, the signal will be lost.
- If the receiving state machine is in a state were the signal is not handled, the signal will be lost.

Decision

The decision construct is used to perform alternative actions in a transition dependent on the value of an expression. It is a mechanism similar to a switch. A decision has one **question** part, which contains a dynamic expression that is evaluated when the decision is executed. Furthermore, a decision has multiple **answer** parts, each containing a range expression (or just a simple expression containing a value or a constant) and leading to different partial transitions.

Symbol

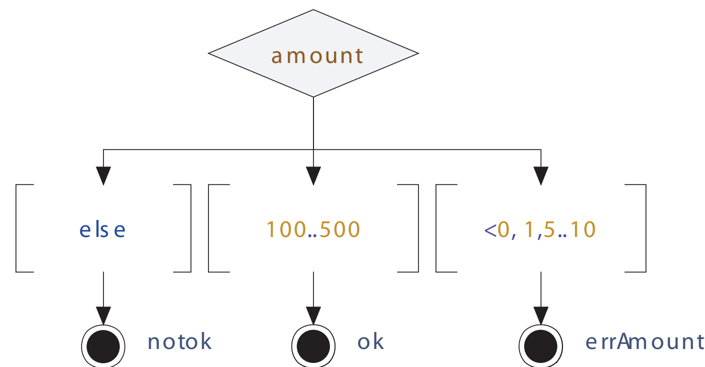


Figure 127: Use of decision

The Decision symbol specifies alternative paths in the behavior part of a transition.

- An expression must be defined. Each path is labeled with an answer that should match the expression for the path to be taken.
- When you delete the decision symbol, the succeeding subtree is deleted as well.

Decision answer

The Decision Answer symbol specifies one alternative path in the behavior part of a transition and contains a range condition which is an answer to a decision question.

A range condition is given either as

- a specific value (for example “`10`” or “`true`”)
- an open range (for example “`>10`”)
- an closed range (for example “`2..10`”)
- a comma separated list of the above mentioned alternatives.

Informal decisions

To facilitate early verification of models it is possible to specify informal decisions. These are characterized by having an expression that is a character string and answers that also are character strings.

Nondeterministic decisions

It is also possible to describe a nondeterministic decision. This is done by giving the expression “any” (without quotes) and leaving the decision answers empty.

Syntax

Example 48: Decision expression text example

```
v+4
```

Example 49: Decision alternative text example

Simple example:

```
True
```

Open range:

```
>10
```

Closed range:

```
0..3
```

Several ranges:

```
<-5, 0..2, >10
```

Guard

A guard symbol can be used for either:

- Triggering a transition based on a certain condition evaluating to “true”.
- A connect transition, that is leaving a substate via an exit point.

Symbol

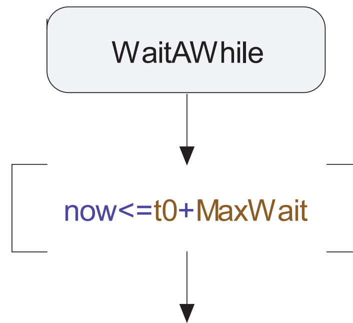


Figure 128: Guarded transition

If a guarded transition is based on a condition, the invocation of the transition occurs when the provided expression defining the condition is evaluated to true. The provided expression must be a simple expression and it may not cause any side effects.

If the transition is defined by referencing an exit point, then the source state of the transition must have a substate state machine. The transition is executed whenever this substate state machine exits via the specified exit point.

Syntax

Example 50: Guarded transition

```
[ x>10 ]
```

Example 51: Connect transition

A transition triggered by leaving a composite state via a named exit point called ‘a’

```
[ a ]
```

Example 52: Connect transition

A transition triggered by leaving a composite state via an unnamed exit point.

```
[ ]
```

Timer set action

The Timer Set Action creates a timer instance, which now is active. Performing set once more on an active timer instance, implicitly resets the first timer instance and creates a new timer instance.

A timer with parameters may have several timer instances active at the same time, as long as the parameter values are distinct.

Syntax

Example 53: Absolute time

```
set (MyTimer, aTime);
```

Example 54: Relative time

```
set (MyTimer, now+10);
```

Example 55: Timer with default duration

```
timer MyTimer () = 5;
...
set(MyTimer);
```

Example 56: Timer with parameter

```
timer MyTimer (Integer id);
Integer i = 1;
...
set (MyTimer (i), now+5);
```

See also

“Timer active expression” on page 296

Timer reset action

The Timer Reset Action resets an active timer instance, if such an instance exists.

Syntax

Example 57: Reset of normal timer

```
reset(MyTimer);
```

Example 58: Reset of timer with parameter

```
reset(MyTimer(i));
```

Action (task)

The Action symbol is used for writing textual code in the behavior part of a transition, for example variable assignments, for-loops and calls of value returning procedures.

Symbol

```
set(t, now+10);
for(Integer i=1;i<=5;i=i+1){
    output Ack(i) to ListOfServers[i];
}
```

Figure 129: Action symbol

Syntax

Example 59: Simple example

```
Integer v1;
v1 = 4;
output s(v1);
```

Assignment

Assignments are done according to the syntax in the example below. The left hand side of the assignment can contain a variable identifier, an element of an indexed variable or a struct field of a struct or class. The right hand side contains an expression of the same type as the left hand side.

Example 60: Various assignments

```
Integer i = 0;
myObject = new (theType);
person.age = person.age+1;
arrival[currentDate,person] = now;
```

The assignment can also be used as an expression in itself. The returned value of an assignment expression is the right hand side expression, if the assignment is successful.

Example 61: Assignment expression

```
if ((a=10)==10) { output s; };
```

Compound statement

A compound statement contains a number of statements enclosed within braces {}. It also defines a namespace which makes it possible to declare local variables within a compound statement.

New

The **new** statement is used to create instances of both active and passive classes. To create an instance of the same class as the current class, the keyword **this** can be used. The new construct returns a reference to the created object.

It is of course always possible to communicate with the created instance using a reference to the object, so by assigning the result of **new** to a reference attribute it is possible to for example send signals directly to the created instance or to call an operation on the instance.

However, to make it possible to communicate with the instance using the ports and connectors that exists in a model, the created instance must be added to the architecture (the structure of connectors and ports) that exists in the application.

Save

It is often wanted to deal with arriving signals in a certain order. However, signals arriving from the outside world may not always arrive in the order that is expected. To temporarily save a signal in the signal queue, while looking for other signals to consume, the Save symbol should be used.

Several signals may be saved in each state, but if a saved signal is not handled in the next state, it again risks being discarded.

Symbol

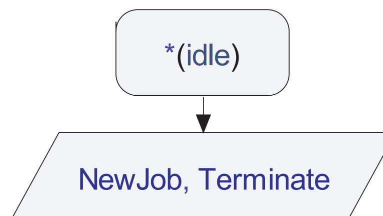


Figure 130Using the save symbol

The Save symbol saves signals from being discarded when being next to consumed in the state that does not handle the signal.

- This symbol should always be preceded by a State symbol.
- You cannot insert any symbols after the Save symbol.

Syntax

Example 62: Save

A simple example:

```
save s;
```

Asterisk save:

```
save *;
```

Stop

The Stop symbol stops the execution of the current instance. It is possible to delete an instance of an active class only from within the state machine of the class, by performing the stop action.

Symbol

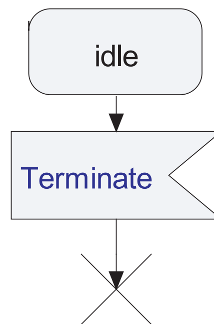


Figure 131: Stop

The stop action is handled in the following way:

1. If the instance is a simple state machine without any parts, the state machine will be immediately stopped.
2. If the instance contains parts, each of the part instances will be handled according to 1) above, as well as this instance.

Return

The Return symbol finishes the execution of operations or substates and transfers the control to the calling context.

Symbol

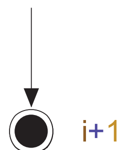


Figure 132: Return in Operation

Syntax

Example 63: Return simple example

```
4+r
```

Example 64: Return with exit point name in composite state

```
exP2
```

If the operation has no return type or if the composite state exit is through the default exit point, the text field should be empty.

Junction

Normally, the state and nextstate are sufficient mechanisms to split up a complex state machine into several diagrams. However, if a transition is very long, it might be necessary to split the description of the transition into several parts. This can be done by the Junction symbol, which is used both as a label and as a jump statement. Another reason for using the junction might be to avoid having crossing flowlines in a complex flow.

Symbol

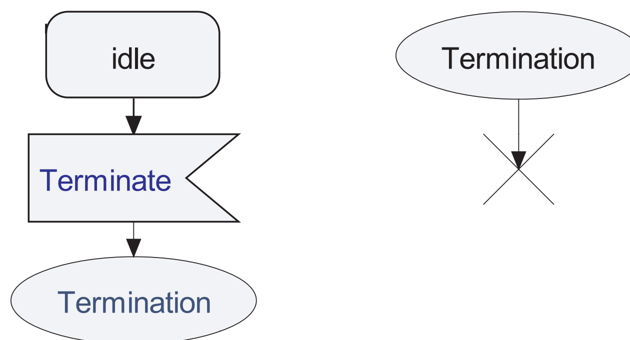


Figure 133: Using the junction as a label and corresponding goto

The Junction symbol corresponds to the label and join symbols but is also used in all cases where there is a need to merge flow lines.

- The Junction symbol can have more than one incoming flow line.

Syntax

The symbol contains a text field.

Flow

The Flow line connects two symbols in a transition.

- If you have a symbol selected in the drawing area, and then add another symbol from the toolbar while holding the <SHIFT> key down, then a flow line will automatically be created between the symbols.
- You can also create the line by drawing it from the line handle and connecting it to the next symbol.
- If you delete a symbol the line connected to the symbol will also be deleted.

Simple transition

The Simple Transition line is used to define a transition when using a state-oriented style.

- You can draw a Simple Transition line from the State symbol only.
- You can create the line by drawing it from the line handle and connecting it to the next symbol.
- If you delete a symbol the line connected to the symbol will also be deleted.

Syntax

There is one text field associated with a simple transition line. This text field describes both the trigger of the transition, the guard and the actions on the transition.

The trigger and guard follow the same syntax as is used in an Signal Receipt (Input) symbol. The actions follow the same syntax as in a Action (task) symbol with the exception that a short-hand is used to denote signal sending to save diagrams space: ‘^s’ means the same as ‘output s’.

Simple example

```
s1(x) / ^s;
```

Simple transition with guard

```
[ x>10 ] / myproc(x);
```

Both guard and signal receipt

```
s1 [ x>10 ] / myproc(x);
```

Expressions

Expressions in UML are similar to expressions in most programming languages. As expected, expressions may contain references to variables (attributes), literals, constants and operations (calls).

Many expressions may be used as actions by appending a semicolon (;) after the expression. For example, the following expressions are commonly used as actions:

- Assignment Expression
- Call expression
- New expression
- Conditional expression

There are a couple of expressions for special variable access or creation of complex values:

- Field expression
- Index expression
- Instance expression
- This expression

There is also a group of expressions, that similar to variable access, depend on the underlying dynamic state of the system, they are often referred to as Imperative expressions:

- Arbitrary value (any) expression
- Now expression
- Pid expressions
  - Self
  - Sender
  - Parent
  - Offspring
- Timer active expression

Other expressions available are:

- Range check expression
- Target code expression

Call expression

A call expression is used for calling operations. It may contain actual parameters for the operation call.

Example 65: Call expression

```
foo(3, true, "mmo")
```

The value of a call expression is the actual value of the operation's return parameter after the call. If the called operation has no return parameter, the call expression has no value, and must then only be used as a stand-alone expression in an expression action.

Before the operation call takes place the expressions provided as actual arguments will be evaluated. Note, however, that UML does not define the order in which the expressions will be evaluated. The actual evaluation order depends on which code generator that is used, and sometimes even on which compiler that is used for compiling generated code. Therefore, it is recommended that models do not depend on the evaluation order of actual arguments in call expressions.

Example 66: Argument evaluation order is undefined

```
foo(f1(), f2())
```

The operations in this example can either be called in the order 'f2', 'f1', 'foo' (right-to-left evaluation order) or 'f1', 'f2', 'foo' (left-to-right evaluation order).

Note

When using the C code generator (including Model Verifier and Model Validator, but excluding AgileC) call arguments will always be evaluated from left to right, regardless of which target compiler that is used. Still it is not recommended to exploit this behavior since the evaluation order is not defined in the UML standard.

New expression

The new expression contains the new() construct as described in “New” on page 286.

Conditional expression

A Conditional Expression has the form

```
expr_1 ? expr_2 : expr_3
```

where the first expression is of the boolean type and the second and third expressions are of the same type.

The expression `expr_1` is evaluated first. If it is true, then the expression `expr_2` is evaluated and provided as the resulting value of the conditional expression, otherwise `expr_3` is evaluated and given as result.

Example 67: Conditional expression

```
imax = ( i > j ) ? i : j; /* imax = max (i, j) */
```

Field expression

The Field Expression is used to access a field of a structured datatype, that is an attribute of a class.

Example 68: Field expression

```
a.b = true;
test = a.b;
```

Index expression

The Index Expression is used to access an element of an indexed datatype, typically an array or a string.

Example 69: Index expression

```
iarr[i, j] = 1;
i = iarr[k,l];
```

Instance expression

The Instance Expression is used to create complex values in one operation. By this, it is possible to initialize a structured type in one operation, instead of initializing each field separately. Note, however, that constructors are recommended for initializing structured types.

Example 70: Instance expression

```
class sType {
    Integer Age;
    Charstring Name;
    Boolean MaleGender;
}
s = sType(. 'John', 44, true .);
```

Instance expressions are also used to describe stereotype instances containing tagged values.

This expression

This refers to the current instance. If this is used in an operation of a passive class, this refers to the instance of the passive class. If this is used in an operation of an active class or in a state machine, this refers to the instance of the active class.

Imperative expressions

Imperative Expressions include:

- Arbitrary value (any) expression
- Now expression
- Pid expressions
- State expression
- Timer active expression

Arbitrary value (any) expression

The any Expression yields an arbitrary value of the provided type.

Example 71: any expression

```
anInt = any(Integer);
```

```
output resultSig(any(Boolean));
```

Now expression

The Now Expression returns the current time value.

Example 72: Now expression

```
Time time_0 = now;
set(delayTimer, now + 10);
```

Pid expressions

Pid expressions are expressions of the datatype Pid. A Pid Expression is either of **self**, **parent**, **offspring** or **sender**.

Example 73: Pid expressions

```
currentClientId = sender;
new serverAgent;
if (offspring != NULL)
  output sender.serverId(offspring)
  else output sender.AllServersBusy;
```

State expression

The State Expression can be used to check the most recently visited state in the current state machine. If the state machine contains composite states, the expression returns the most recently visited state of the nearest enclosing scope. The returned expression will be of the Charstring datatype. If no state has been visited, an empty string is returned.

Example 74: State expression

```
if (state  == "idle") return ;
```

Timer active expression

The Timer Active expression is used to check if a named timer is active or not. A boolean value will be returned. A timer is active either if the timer has not expired yet or if the timer has expired but the timer signal has not been consumed yet (or discarded).

Example 75: Timer active expression

```
if (active(userTimeout)) reset(userTimeout);
```

Range check expression

A Range Check Expression is used to check if an expression meets a value range condition at run-time. It has the form:

```
expr_1 in type type_ident
```

Where type_ident may be further restricted by a constraint. The range check expression will return a Boolean value depending on if the expression matches the provided type.

Example 76: Range check expressions

```
sender in type clientType;
intVar in type Integer constants (1..9, -9..-1);
age in type ageSyntype;
```

Target code expression

A Target Code Expression is dependent of the selected implementation language and contains implementation language code that is not parsed by the UML parser, but instead added directly to the generated code.

Target code has the format

```
[[ target_code_details ]]
```

The target code (for example Inline C/C++) can contain any expression in the implementation language that matches the type that the UML context specifies.

If the target code contains the text

```
]]
```

this must be escaped by a # as

```
#]]
```

If the target code contains

```
#
```

this must be escaped by a # as

```
##
```

If it is needed to reference model entities from the target code, this has the form #(name) where name is an identifier in the model.

Example 77: Target code expression

```
Real side_a, side_b;
...
Real hypotenuse = [[ sqrt( pow(#(side_a),2) + pow(#(side_b),2) )]];
```

See also

“C Application” on page 978

Composite state

A composite state is a state which is composed by other states and transitions. While in any of the substates of the composite state, a trigger with a transition defined for the composite state will cause an exit of the composite state (and substates) for a new state.

A composite state can be created in two ways: either by an inline state machine definition or by referring to a state machine defined elsewhere.

A composite state can implicitly be created when a state machine diagram is created on a state.

A composite state is marked with a “rake” symbol in the upper right corner of the state symbol.

The composite state may have several entry and exit points, which are labelled.

Transitions in a substate has higher priority than transitions in an outer state. This applies both to transitions triggered by signals and to transitions triggered by timers.

This is in UML referred to as **transition overriding**.

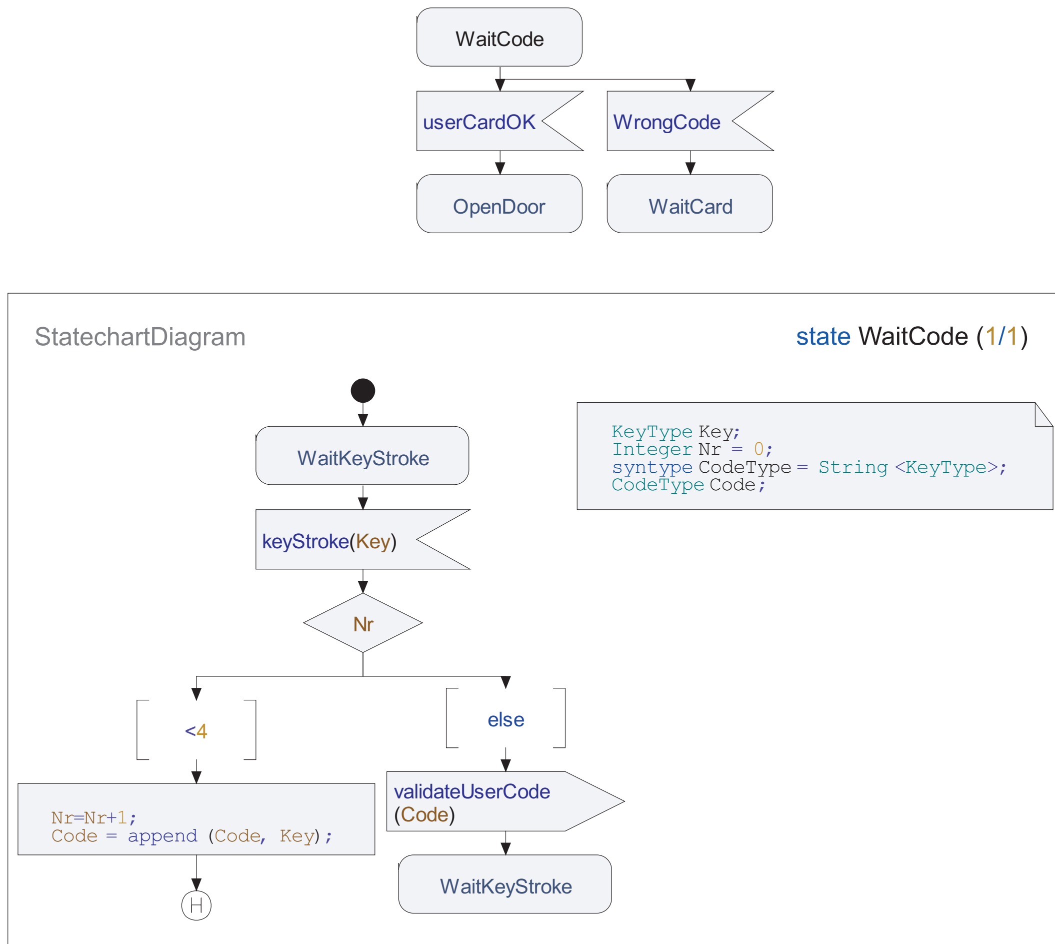


Figure 134: Use of a composite state

Entry connection point

An Entry Connection Point is a named starting point for entering a composite state. Entry connection points are referenced at a start symbol inside a composite state and in the nextstate symbol when entering a composite state.

There must be at least one named or unnamed start symbol in a composite state. Not more than one start symbol can be named in a composite state.

Hint

An Entry connection point is defined in the Model View by selecting a state machine and choosing the ***New / Entry Connection Point*** *command in the shortcut menu.*

Exit connection point

An Exit Connection Point is a named exit for leaving a composite state. Exit connection points are referenced at a return symbol in a composite state and at connect transitions leading out of composite states.

If there are more than one connect transition from a composite state, at most one of these connect transitions can be unnamed.

Hint

An Exit connection point is defined in the Model View by selecting a state machine and choosing the ***New / Exit Connection Point*** *command in the shortcut menu.*

State machine inheritance

A State machine can be specialized, either directly by inheritance between state machines or by specialization of the active class that owns the state machine. A specialized state machine may add features or change features of the original state machine. Features that may be added include states, transitions, variables and other entities that can be declared in a state machine. In order for allowing a feature to be changed by specialization, it must be declared as **virtual** in the original state machine. A virtual definition may be redefined in the specialized state machine. The following concepts can be virtual (and thus redefined) in a state machine:

- Transitions
  - Start
  - Signal Receipt
  - Guard
  - Save
- Operation

Operation body

An Operation Body is a method without states. The action is often a compound action, which contains a list of other actions.

An Operation Body may be informal, meaning that the specification of how to execute it is not formally expressed in the UML language, but maybe in some other language. In that case the operation body will contain an informal expression containing the informal description.

See also

"State machine implementation" on page 300

"Internals" on page 300

"Implementation" on page 322

State machine implementation

A State machine Implementation is a method containing states and everything else needed to realize the State machine signature. A State machine Implementation is typically implicitly defined when defining a State machine.

See also

"State machine" on page 267

"Internals" on page 300

"Implementation" on page 322

Internals

Internals are used to be able to divide a class definition into one signature-oriented part and one implementation-oriented part and then store the signature for a class in a different file than the implementation of the class. The purpose of this is to facilitate component-based modeling by allowing separate version handling and delivery for the signature and the implementation.

See also

"State machine implementation" on page 300

"Operation body" on page 299

"Implementation" on page 322

Text extension symbol

The text extension symbol can be connected to the action symbol to display the content of the action symbol. This is particularly useful when drawing transition oriented flows where an action with a large amount of text can disturb the overview of the diagram. The action code can be edited either in the action symbol or in the text extension symbol.

Deployment Modeling

In deployment modeling, the run-time architecture of the system is modeled. It describes how deployable pieces of the software, Artifacts, are deployed onto Nodes representing physical computation resources. Deployment specifications are used to describe how artifacts are deployed onto nodes. Associations are used to model connections between nodes.

Deployment diagram

A deployment diagram specifies a set of Artifacts deployed onto a set of interconnected Nodes. A Deployment specification is used to specify execution parameters used when deploying an artifact onto a node. An Execution environment can be used to model a node providing a set of services to the artifacts deployed onto it.

Example

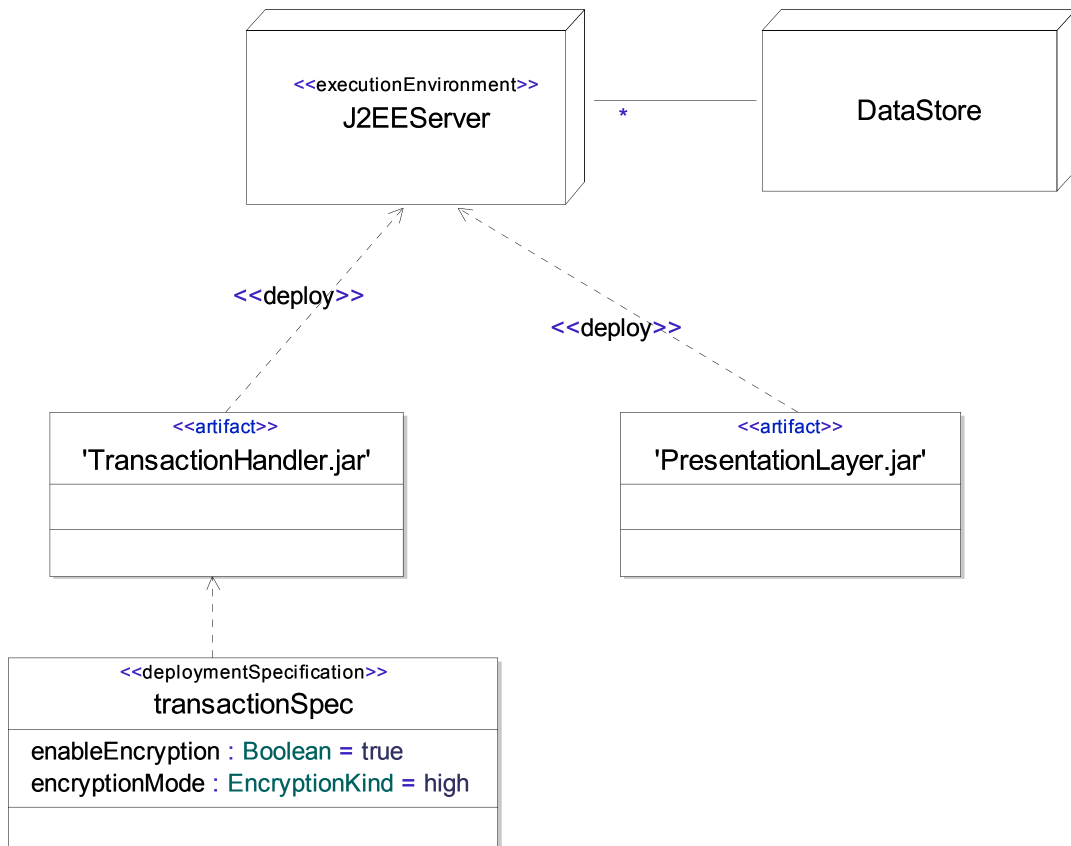


Figure 135: Deployment diagram

Model elements in deployment diagrams

The following elements are found in deployment diagrams

- Artifact
- Node
- Execution environment
- Deployment specification
- Artifact
- Class
- Relationships

See also

"Class diagram" on page 199

"Component diagram" on page 243

Artifact

An Artifact represents a physical piece of information that is used or produced by a software development process. Examples of artifacts include source files, scripts, libraries and executable programs.

An artifact manifests a number of elements through Manifestation relations, meaning that the artifact is built up, or constructed from, these elements. For example, an artifact representing a header file in C++ can have a manifestation relation to the class declared in the header file. This information can then be used by a code generator when generating the physical header file from the model.

During deployment modeling, artifacts are deployed on nodes using the Deployment relationship.

Artifacts are similar to Classes and can have Attributes and Operations. Artifacts can also participate in the following relations: Dependency (of any element), Generalization (between artifacts), Composition (typically to other artifacts). In addition, an artifact is a namespace and can therefore own other model elements.

Symbol

Figure 136: Artifact symbol

The artifact symbol is identical to the Class Symbol, with the keyword «`artifact`» added to the top.

Node

A node is a named computational resource, typically a specific computer. Nodes can be connected using Associations to model network topologies.

Symbol

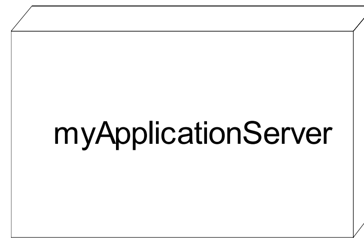


Figure 137: Node symbol

Syntax

A node is depicted as a 3-dimensional cube with the name inside.

Execution environment

A special kind of Node offering an execution environment for the artifacts deployed onto it. The execution environment typically consists of a set of services required by the artifacts during execution.

A typical example is a J2EE server prepared for deployment of J2EE beans.

Symbol

Figure 138: Execution environment symbol

Syntax

Same as node with the stereotype «`executionEnvironment`» applied.

Deployment specification

A deployment specification is used to specify a set of properties acting as execution parameters for an artifact when deployed onto a Node.

A deployment specification is applied to an artifact by drawing a Dependency from the specification to the artifact.

Symbol

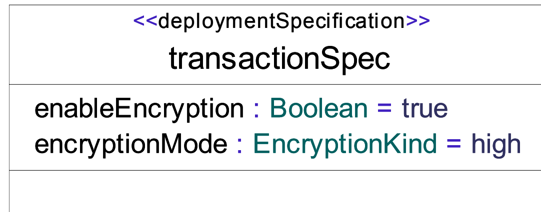


Figure 139: Deployment specification symbol

Syntax

Same as class with the «`deploymentSpecification`» stereotype applied.

Relationships

The following relationships can be used in Deployment diagrams:

- Deployment
- Manifestation
- Association
- Aggregation
- Composition
- Generalization
- Dependency

Deployment

A special kind of Dependency used to deploy an artifact onto a deployment target, typically a Node. An artifact deployed onto a node will perform its execution in the context of that node.

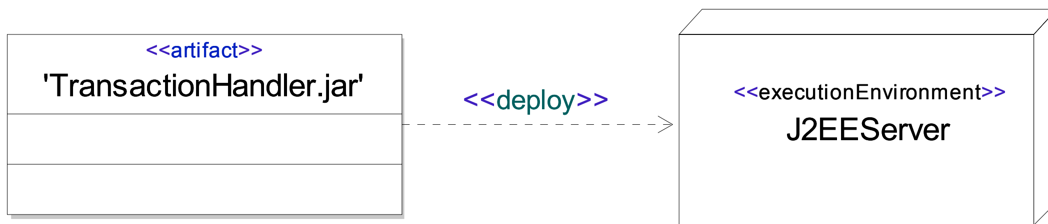


Figure 140: Deployment dependency

Manifestation

Manifestation is a special kind of Dependency used from an Artifact to a set of other elements to describe that the artifact is built up, or constructed from, these elements.

For example, an artifact representing a header file in C++ can have a manifestation relation to the Class declared in the header file. This information can then be used by a code generator when generating the physical header file from the model.

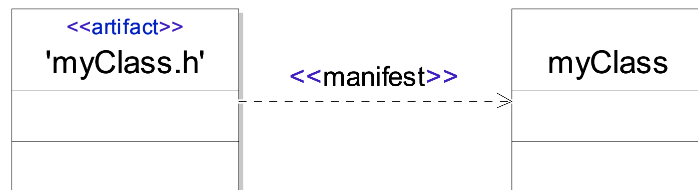


Figure 141: Manifestation dependency

Relationships in UML

For general help on editing lines, please see:

“Draw lines” on page 136

“Move lines” on page 137

“Delete lines” on page 137

“Re-direct and bi-direct lines” on page 137

Dependency

A Dependency is a relationship between two definitions, saying that one of these definitions (the *client*) is dependent on the other definition (the *supplier*) for some reason. The somewhat loose semantics of a Dependency makes it usable when the other relationship classes are inappropriate and cannot model a certain relationship.

There is one case when the dependency is used in a more specific way: indicating a creation relationship between instances of active classes, that is when an instance uses the New statement to create a new instance of a class. In this case, the dependency can be used between parts or between a part and the behavior symbol that refers to the state machine of the enclosing active class.

It is common to give dependencies a more detailed semantics by means of applying stereotypes on them. For example, see Import and Access dependencies.

Generalization

A Generalization is a relationship between two Signatures (for example classes or operations), saying that one of these is a more general signature, and the other is a more specific one. The more specific signature inherits member definitions from the more general signature, and may also contain additional members. Because of this, the generalization relationship is also known as inheritance.

If a generalization is established between two types (for example two classes) the more specific type defines a subtype of the more general type (which is sometimes called a supertype). This means that an instance of the more general type may be substituted by an instance of the more specific type. In other words, a specialized type is assignment compatible with the more general type.

Syntax

The generalization line has a text field, which may contain the discriminator.

Realization

The Realization relationship is a special kind of the Generalization relationship. A Realization is used between a class and an interface to express that the realizing class conforms to (implements) the interface.

Association

An Association is a semantic relationship between two or many Classifiers, indicating that instances of these classifiers will be related.

Symbol

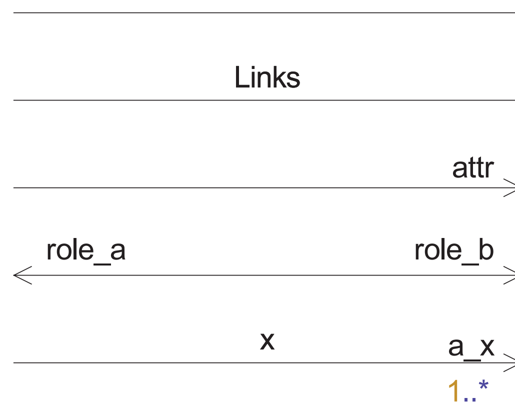


Figure 142: Association

The line contains one name field, two role name fields and two Multiplicity fields.

An Association has two association ends, represented as attributes. These attributes could either both be owned by the association (reflecting the situation when none of the associated classifiers are affected by the association), or one attribute could be owned by the association and one by the connected-to classifier C (reflecting the situation when the association is navigable only in the direction from C), or the attributes could be owned by one connected-to classifier each (reflecting the situation when the association is navigable in both directions). In the case when the association is unidirectional, the second (remote) Attribute will only exist if it is needed (for example if it carries a role name or a multiplicity).

An Association may also have properties that belong to the Association itself, and not to any particular association end.

An association can be navigable in both directions.

Multiplicity

Multiplicity at an association end defines how many instances of the class that can be related by the association.

Aggregation kind

An Association is either a normal association, an Aggregation or a Composition.

You can change aggregation type on the shortcut menu that is displayed when you click the ending parts of the line. The alternatives are Association, Aggregation and Composition. You must first add role names before you can select aggregation type.

- An Aggregation line specifies that an instance of the aggregate class is an informally considered owned by the instance of the component class.
- A Composition line specifies a stronger form of aggregation where the instance of the aggregate class exists only as long as the component class exists. The lifetime of the contained instance is thus strongly tied to the lifetime of the containing instance.

Navigable end

A navigable end is an association end that is also an attribute of the classifier that is the type of the other end.

Symbol

The line contains one name field, two role name fields and two Multiplicity fields.

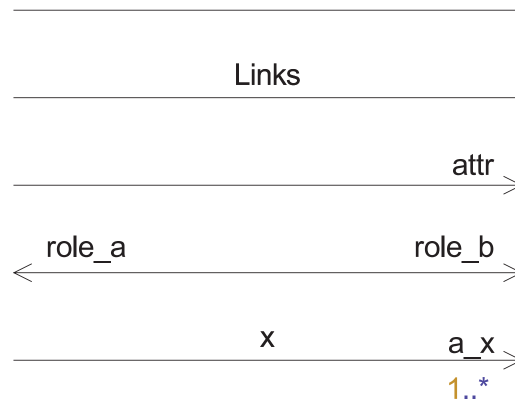


Figure 143: Association

Examples

Example 78: Role text

```
+ myrole
```

Example 79: Multiplicity text

Infinite range:

```
*
```

Range condition:

```
0..3
```

Multiple range conditions:

```
1..7,>10
```

See also

“Attribute” on page 206

“Aggregation” on page 311

“Composition” on page 311

Aggregation

Aggregation is a special kind of Association. It is a binary association that specifies an aggregation relationship (a whole/part relationship).

An aggregation has two ends, an aggregate end and a part end. An aggregation specifies that an instance of a classifier on the aggregate end aggregates an instance of the classifier at the part end. The aggregate instance may in turn be part of another aggregate.

An aggregation part may be part of more than one aggregate.

Symbol

Figure 144: Aggregation

See also

"Attribute" on page 206

"Association" on page 308

"Composition" on page 311

Composition

Composition is a special kind of Aggregation. The composite part is strongly owned by the composite and may thus only be part of one composition.

Composite parts that are typed by active classes can also be used as parts of the internal structure of a class as described by Composite structure diagrams.

Symbol

Figure 145: Composition and a corresponding attribute

See also

"Attribute" on page 206

"Association" on page 308

"Aggregation" on page 311

"Part" on page 236

"Composite structure diagram" on page 235

Containment

The Containment relationship shows that one definition contains another definition. The contained definition appears in the scope of the container definition. When used between namespaces, such as packages, the containment relationship is sometimes also called namespace nesting.

Symbol

The Containment line is drawn from the container definition to the contained definition, and shows a plus sign at the container side.

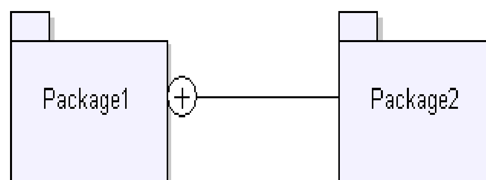


Figure 146: Containment

Extension

Extension is used between a stereotype and a metaclass (a Metamodel class) to indicate that the stereotype extends the metamodel class.

Association

Description

Association is described in detail in the section Use Case Modeling.

Common Symbols

Frame

The symbols in a diagram are enclosed by the Frame symbol placed on the canvas.

- The frame has margins on all sides.
- You may resize and move the frame in all directions on the canvas, including the margins.

Text symbol

The Text Symbol is used for defining variables, interfaces, datatypes etc.

It is not possible to connect lines to the symbol.

Syntax

Example 80: Including the definition of an interface and a syntype

```
interface i {
  signal s;
}
syntype s = Integer;
```

Example 81: Including the definition of a stereotyped class

```
<<struct>> class X {
  private Integer I;
```

```
    void inc ( Integer incr ) {
      I = I + incr;
    }
  }
```

Comment

You use the Comment symbol to define comment text related to graphical symbols in a diagram.

Comments can also be made in the textual syntax.

Comment symbol

The comment symbol is drawn similar to a Text symbol, but has a read-only text label in the upper left corner of the symbol. The text is set to “//”, to distinguish the constraint symbol from for example a constraint symbol. It is possible to connect the symbol to another symbol with an Annotation line.

The comment symbol is connected on the left side but you can flip the symbol horizontally from the shortcut menu to connect it from the right side instead. When a Comment symbol in a diagram is not connected to any other symbol then the comment model element belongs to the element owning the diagram. If a Comment symbol is connected to two or more symbols in a diagram, then the comment model element belongs to the element owning the diagram

Syntax

The text is informal and will not be syntactically checked.

See also

“Handling comments” on page 126

Constraint

You use the Constraint symbol to define constraint text related to graphical symbols in a diagram.

Constraints can also be made in the textual syntax.

Constraint symbol

The constraint symbol is drawn similar to a Comment symbol, but has a read-only text label in the upper left corner of the symbol. The text is set to “{}”, to distinguish the constraint symbol from a comment symbol. It is possible to connect the symbol to another symbol with an Annotation line.

Syntax

The text is informal and will not be syntactically checked.

Stereotype instance

You use the Stereotype instance symbol to define stereotype instance text related to a model element.

Stereotype instance symbols can also be made in the textual syntax.

Stereotype instance symbol

The stereotype instance symbol is drawn similar to a Comment symbol, but has a read-only text label in the upper left corner of the symbol. The text is set to “«»”, to distinguish the constraint symbol from a comment symbol. It is possible to connect the symbol to another symbol with an Annotation line.

Syntax

The text is informal and will not be syntactically checked.

Annotation line

The Annotation line connects the Comment, Constraint and Stereotype instance symbol to another element.

You can draw an Annotation line from the line handle on the symbol and attach it to any symbol, inside the diagram frame, other than other Comment, Constraint and Stereotype instance symbols. It is also not allowed to attach it to a Text symbol.

Extensibility

UML is a language that you can customize - in a controlled way. There are predefined mechanisms to extend UML constructs and to specialize them to a use for a specific purpose.

The extensibility mechanism of UML is based on the concept of Profile and Metamodel.

A metamodel is simply a special kind of UML package class model that is used to describe the information stored in a repository in a tool. A package is a metamodel if the package name is preceded by the keyword «metamodel». A metamodel typically contains a set of classes stereotyped by the keyword «metaclass» that define the metaclasses.

It is possible to define different metamodels and use the build-in repository to store user-level models based on these metamodels. The only requirement is that the metamodel must be possible to map to the object model used to define the run-time repository and storage.

A profile is a special kind of package, identified by the keyword «profile» before the package name in the heading. A profile contains a set of *stereotypes*, that have attributes (called *tagged value definitions*) and that extend one or more *metaclasses*.

In a user model the stereotype can be applied to an object that is an instance of the extended Metaclass. This will automatically make it possible to add values

Metamodel

A metamodel is a set of metaclasses, metaattributes etc. that defines a conceptual view of the information stored in the model repository. The main practical usage of a metamodel is to form a basis for profile definitions.

A user profile can define stereotypes that extend the metaclasses in order to associate more information to model elements. The extra information is from a user's point of view editable using the Properties Editor and is stored in the model repository.

The UML tool set is able to represent different metamodels each giving a different view of a specific model.

Hint

An example of a metamodel is given in the installation. Simply check the TTDMetamodel package in the Library node in the Model View. This package is a simple metamodel that describes the information stored.The purpose of the TTDMetamodel is to give a view that is very close to the underlying repository structure and each of the classes found in this metamodel corresponds directly to a core class found in the repository definition. However, the TTDMetamodel is a simplification of the core repository in the sense that only the classes that are useful to stereotype are included. Another simplification is that almost all of the associations and attributes found in the core repository model are omitted.

Metaclass

A metaclass is used to categorize a set of elements stored in a UML repository. It can be defined in metamodels using class symbols where the class name is stereotyped by «metaclass».

Stereotype

A Stereotype is used to extend the information that can be stored in the model for a given entity. The extra information is described by the attributes of the stereotype.

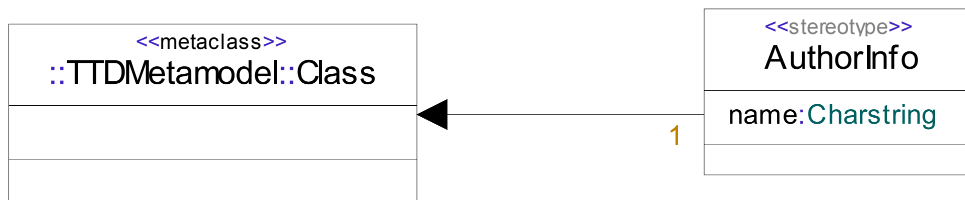


Figure 147: Stereotype Example

Figure 147 on page 317 shows an example of how to extend all classes with information about the author of the class definition. This is accomplished by defining a stereotype `AuthorInfo` with an attribute `name` that extends the Metaclass `TTDMetamodel::Class`.

Tag definition

Tag Definitions are the attributes of a stereotype. When the stereotype is applied, the tag definitions are used by giving them specific values.

Tagged value

Tagged values are the values that can be given tag definitions. These values are set using the Properties Editor.

See also

"Extension" on page 318

Profile

A profile is a special kind of package that is identified by the stereotype «profile». Profiles are used to extend the information that can be stored in a UML repository by defining stereotypes that extend metaclasses. Figure 147 on page 317 shows an example of a simple profile.

A profile is applied by using the package Import or Access constructs, for example if a model should adapt a certain profile, the top package of the model should have an import or access that references the package that defines the wanted profile.

Extension

Extension is used between a Stereotype and a Metamodel class to indicate that the stereotype extends the metamodel class.

There is one text field associated with the extension line. This can have the text '1' or '`0..1`'. If the text is '1' then all elements that are instances of the extended Metaclass will automatically have the stereotype applied.

If the text is '`0..1`' then you will have to manually apply the stereotype. In Figure 147 on page 317 is an example of an extension line.

When the stereotype is manually applied, some symbols (class symbol, signal symbol etc.) will show the applied stereotype in the symbol.

Predefined Data

The data modeling constructs in UML are powerful and allows for modeling and defining data in numerous ways. However, UML does not contain many built-in datatypes. Instead UML can be extended with different sets of datatypes depending on the application area. This is done by defining datatypes in model libraries (also often referred to as predefined packages).

Predefined

This package contains generic datatypes with operations that always can be used.

See also

"Datatype" on page 224

Predefined

The package Predefined is a proprietary extension to UML, which is always available in a project. This package is automatically used by the model defined in a project. The package defines a number of datatypes, but also a few other utilities.

Some of the datatypes exist in OMG UML (e.g. Integer or Boolean), but the Predefined package provides operations for these datatypes which is not done in the standard.

For each datatype, there is a set of operations to be applied on expressions of the type.

The package contains the following definitions:

Kind	Definitions
Datatypes	Boolean, Character, String, Charstring, Integer, Natural, Real, Array, Any
Constants	PLUS_INFINITY, MINUS_INFINITY

The Package Predefined is also available for inspection or browsing directly in the Model view. Each project has a node called predefined package. Expanding this node lets you browse the available datatypes, operators and other definitions.

PLUS_INFINITY

PLUS_INFINITY is a constant of the datatype Real. It can be used as a reference to the largest Real number that can be used on host, or on a specific target.

MINUS_INFINITY

PLUS_INFINITY is a constant of the datatype Real. It can be used as a reference to the largest negative Real number that can be used on host, or on a specific target.

Metamodel Classes

A few of the more important metaclasses are described below.

Metamodel profile

The TTDMetamodel is available for inspection and browsing directly in the Model view. When adding a project, there is always a node with the applied profiles, TTDMetamodel is one of these profiles. Expand Library node and the TTDMetamodel profile package to see the language model elements, abstract metaclasses and the relationships between these.

Classifier

Classifier is a Metaclass in the UML language.

A Classifier is a description of data and is the Signature of a set of instances or Instance Sets. A Classifier defines a type, which for example may be the type of a `StructuralFeature`. A Classifier may be associated to other Classifiers by means of Associations.

Most class-like model elements are classifiers, including:

- Class
- Datatype, Syntype, Choice
- Stereotype
- Interface
- Collaboration

Signature

Signature is a Metaclass in the UML language.

A Signature is an entity that can be the basis for the definition of another Signature. There are two main mechanisms that enable this:

- Specialization, or inheritance
- Parameterization

Specialization means that a super-signature may be specialized into a set of sub-signatures. Each sub-signature shares all the properties of the super-signature and may have some additional ones too. In the Metamodel the specialization mechanism is modeled by the Generalization class which is owned by Signature.

Parameterization means that a Signature may have a list of formal context parameters. Such a Signature is known as a *template*. Formal context parameters of a template may be substituted by actual context parameters when the template is instantiated (for example in a `TemplateTypeInstantiation`). Parameterization could make a Signature more flexible for use in different contexts. In the metamodel the parameterization mechanism is modeled by the `ContextParameter` class which is owned by Signature.

In addition to these two mechanisms for defining new Signatures based on another Signature, there is a third such mechanism that only one Signature has; the Syntype. This mechanism defines a new Signature by possibly constraining another one.

Some Signatures may have an Implementation. In that case the Signature acts as a façade for the Implementation, hiding all details which users of the Signature do not need to know. A façade allows for separating of a definition from its implementation and is what enables separate compilation of parts of a system. Compare for example with the use of header files in C programming. The following statements are true for a façade:

- A façade does not depend on its implementation.
- A façade does not depend on its uses. (This is in fact true for all Definitions.)

An implementation may only depend on façades.

The following model elements are signatures:

- Classifier
- Operation, signal, timer

Implementation

An Implementation describes details about a Signature which users of the signature do not need to know about, but that are necessary from an execution point of view. While a Signature typically describes static properties of an entity, the corresponding Implementation is more concerned with the dynamic properties.

There are two main kinds of Implementations; Internals and Method. An Internals describes how a Class is structured, both physically and from a communication point of view, while a Method describes an Operation, a StateType, or a Class from a run-time execution point of view.

An Implementation only depends on Signatures (also referred to as façades), not on the usage of these Signatures. This is important in order to enable separate analysis of parts of a system.

Method

A Method is the implementation of an Operation. It describes how it is executed at run-time. There are three kinds of methods, each of which has its own semantics of execution:

- Operation body – a stateless method which is executed by executing the Action of the OperationBody.
- State machine implementation – a method with states and transitions which is executed by executing the Action associated with a Transition that can be initiated in the active state.
- Interaction – a method which describes the interaction and information exchange between a set of attributes. Contrary to other methods, an interaction may not only provide a complete specification of how the operation shall be executed, but it may also be used to describe how it actually is executed (that is describing a trace), or provide a partial description of how it must execute (thereby putting semantic requirements on its other Methods).
- Activity implementation - a method executing a controlled set of small behavioral units.

Signature and implementation

Signature and Implementation are two metaclasses in the UML language. A signature declares an entity and an implementation defines the same entity. The idea is that these concepts make it possible to separate the signature physically from the implementation (compare header files for C and C++).

The concepts for which it is possible to do this are:

Operation

Operation signature and Operation body, Activity implementation, State machine implementation or Interaction.

Activity

Activity signature and Activity implementation.

State machine

State machine signature and State machine implementation.

Class

Class signature and Internals.

•

Profile for Schedulability, Performance, and Time

This section lists all stereotypes, tagged values and enumerations of the **UML Profile for Schedulability, Performance, and Time** also commonly referred to as the UML Real-time profile.

Note

Some tagged values can only be edited using the textual syntax. These are marked as italic in this document.

RTresourceModeling

GRMacquire

GRMblocking : Boolean

GRMcode

GRMrealize

GRMmapping : GRMmappingString

GRMdeploys

GRMrelease

GRMrequires

RTtimeModeling

RTaction

RTstart : RTtimeValue

RTend : RTtimeValue

RTduration : RTtimeValue

RTclkInterrupt

RTstimulus

RTstart : RTtimeValue

RTend : RTtimeValue

RTclock

RTclockId : Charstring

RTdelay

RTevent

RTat : RTtimeValue

RTinterval

RTintState : RTtimeValue

RTintEnd : RTtimeValue

RTintDuration : RTtimeValue

RTnewClock

RTnewTimer

RTtimerPar : RTtimeValue

RTpause

RTreset

RTset

RTtimePar : RTtimeValue

RTstart

RTtime

RTkind : RTkindEnum

RTtimeout

RTtimer

RTduration : RTtimeValue

RTperiodic : Boolean

RTtimeService

RTtimingMechanism

RTstability : Real

RTdrift : Real

RTskew : Real

RTmaxValue : RTtimeValue

RTorigin : Charstring

RTresolution : RTtimeValue

RToffset : RTtimeValue

RTaccuracy : RTtimeValue

RTcurrentVal : RTtimeValue

RTkindEnum

Literals:

- dense
- discrete

RTconcurrencyModeling

CRaction

CRatomic : Boolean

CRasynch

CRconcurrent

CRcontains

CRdeferred

CRimmediate

CRthreading : CRthreadingEnum

CRmsgQ

CRsynch

CRthreadingEnum

Literals:

- local
- remote

SAprofile

SAaction

SApriority : Integer

SAblocking : RTtimeValue

SAdelay : RTtimeValue

SApreempted : RTtimeValue

SAready : RTtimeValue

SArelease : RTtimeValue

SAworstCase : RTtimeValue

SAabsDeadline : RTtimeValue

SAlaxity : SAlaxityEnum

SArelDeadline : RTtimeValue

SAengine

SAaccessPolicy : SAaccessControlPolicyEnum

SAcontextSwitch : TimeFunction

SAschedulable : Boolean

SApreemptible : Boolean

SApriorityRange : Range

SArate : Real

SAschedulingPolicy : SAschedulingPolicyEnum

SAutilization : Real

SAaccessPolParam : Real

SAowns

SAprecedes

SAresource

SAacquisition : RTtimeValue

SAcapacity : Integer

SAdeacquisition : RTtimeValue

SAconsumable : Boolean

SAaccessControl : SAaccessControlPolicyEnum

SAptyCeiling : Integer

SApreemptible : Boolean

SAaccessCtrlParam : Real

SAresponse

SAutilization : Real

SAspare : RTtimeValue

SAslack : RTtimeValue

SAoverlaps : Integer

SAschedRes

SAscheduler

SAschedulingPolicy : SAschedulingPolicyEnum

SAsituation

SAtrigger

SAschedulable : Boolean

SAoccurrence : RTarrivalPattern

SAendToEnd : Charstring

SAusedHost

SAuses

SAlaxityEnum

Literals:

- hard
- soft

SAschedulingPolicyEnum

Literals:

- rateMonotonic
- deadlineMonotonic
- HKL
- fixedPriority
- minimumLaxityFirst
- maximizeAccruedUtility
- MinimumSlackTime

SAaccessControlPolicyEnum

Literals:

- FIFO
- priorityInheritance
- noPreemption
- highestLockers
- priorityCeiling

PAprofile

PAclosedLoad

PArespTime : PAperfValue

PApriority : Integer

PApopulation : Integer

PAextDelay : PAperfValue

PAcontext

PAhost

PAutilization : Real

PAschdPolicy : PAschdPolicyEnum

PArate : Real

PActxtSwT : PAperfValue

PAprioRange : Range

PApreemptable : Boolean

PAthroughput : Real

PAopenLoad

PArespTime : PAperfValue

PApriority : Integer

PAoccurrence : RTarrivalPattern

PAresource

PAutilization : Real

PAschdPolicy : PAschdPolicyEnum

PAcapacity : Integer

PAaxTime : PAperfValue

PArespTime : PAperfValue

PAwaitTime : PAperfValue

PAthroughput : Real

PAstep

PAdemand : PAperfValue

PArespTime : PAperfValue

PAprob : Real

PArep : Integer

PAdelay : PAperfValue

PAextOp : PAextOpValue

PAinterval : PAperfValue

PAschdPolicyEnum

Literals:

- FIFO
- priority

RSAprofile

RSAclient

RSAtimeout : RTtimeValue

RSAclPrio : Integer

RSAprivate : Integer

RSAconnection

RSAshared : Boolean

RSAhiPrio : Integer

RSAloPrio : Integer

RSAmutex

RSAorb

RSAserver

RSAsrvPrio : Integer

RSAchannel

RSAschedulingPolicy : RSAschedulingPolicyEnum

RSAaverageLatency : RTtimeValue

RSAschedulingPolicyEnum

Literals:

- FIFO
- RateMonotonic
- DeadlineMonotonic
- HKL
- FixedPriority
- MinimumLaxityFirst
- MaximizeAccruedUtility
- MinimumSlackTime

5

Error and Warning Messages

This document is a reference guide to error and warning messages from the UML tool set.

General Application Errors and Warnings

DOORS Analyst minidumps (Windows)

DOORS Analyst has built in debug information capturing capabilities on the Windows platform. If at anytime during running the tool you receive a window saying DOORS Analyst has crashed and a minidump has been created please contact your local DOORS Analyst support. The minidump contains the current call stack and can help identify which calls have been made. This can help to identify if an error occurred due to internal tool calls and in these cases may make it possible to resolve problems not already identified. With consideration to dependencies on operating systems and third party calls this information can also be of help to improve integrations to the environment and publish clearer requirements for third party software which DOORS Analyst is dependent on.

Figure 148: Using special characters in identifiers

Minidump location

The minidumps are by default created in a local settings directory but can be relocated using an environment variable.

Default location:

```
C:\Documents and Settings\<user>\Local Settings\Temp
```

Environment variable example:

```
TAU_DUMP_PATH=c:\DevTools\Telelogic\minidumps\
```

Minidump contents

The minidumps only contain the call stack and registers, and no memory. this means that there is no information about the model that the minidump originated from.

Errors and Warnings

Phases and identifiers

There are several phases involved when transforming a UML model to another language, or format. During the processing of the model, error and warning messages may be presented from each phase to help you identify where the problems occur. The prefixes identifying the phases are:

- TSX: Syntax Analysis
- TSC: Semantic Check
- TNR: Name Resolution

TSX: Syntax Analysis

The syntax analysis checks how language elements are constructed and put together in order to form correct UML constructions.

TSC: Semantic Check

The semantic check verifies that the UML model is complete and that the relations between language constructs are meaningful.

TNR: Name Resolution

The name resolution identifies names of the UML entities and attempts to bind them to the correct definition in the model.

TSX: Syntax Analysis

The syntax analysis checks how language elements are constructed and put together in order to form correct UML constructions.

The direct cause of syntax errors will in most cases be possible to locate in the UML model.

Errors and warnings from this phase are prefixed with TSX.

```
Internal error: <string>
```

These kinds of errors should not appear. If they do, please contact DOORS Analyst Support.

TSX0026: Port should not contain two in or two out parts

This error does not occur by normal usage of the tool.

TSX0047: Tagged values are not allowed here

In some places, for example inside a class symbol, you are prohibited to edit properties (Tagged values). Only the stereotype itself can be added.

The preferred way to edit properties is by using the Properties Editor.

TSC: Semantic Check

About semantic checks

The semantic check verifies that the UML model is complete and that the relations between language constructs are meaningful.

Semantic errors occur when there are incomplete constructs in your model. The UML Language Guide can be useful to identify supported constructs.

Errors and warnings from this phase are prefixed with TSC.

TSC0123: A cyclic dependency was found in definition of the %n. (via <string>)

This is a cyclic dependency error. Since two classes cannot be containers for the other one at the same time, this is illegal.

The following is an example of this error:

Example 82

```
class X {
    part Y y;
}

class Y {
    part X x;
}
```

TSC0134: Incomplete transition. A transition must end with stop, nextstate or join action

A decision must cover all answer possibilities, including 'else'.

TSC0092: A corresponding 'virtual' or 'redefined' operation was not found in the parent signatures (or parent signatures does not exist).

There are a number of situations that may be the cause of this error. The following examples shows the situations which can occur.

Using a redefined operation in an active class that does not have generalizations:

Example 83: Class without generalizations.

```
active class P {
  redefined void Op() { }
}
```

Using a redefined operation in a generalization of an active class can cause this error:

Example 84: No matching operation in the parent class.

```
active class P {
}
active class C : P {
     redefined void Op() { }
}
```

When the operation (`Op`) in the parent class has a different signature there can be the following situations:

Example 85: Virtuality must be "virtual" or "redefined".

It is not possible to redefine non-virtual operations.

```
active class P {
     void Op () {   }
}
active class C : P {
    redefined void Op() { }
}
```

Example 86: Different return type.

```
active class P {
    virtual Integer Op () { return 1; }
}
active class C : P {
    redefined void Op() { }
}
```

Example 87: Different count of formal parameters.

```
active class P {
    virtual void Op (Integer x) {  }
}
active class C : P {
    redefined void Op() { }
}
```

Example 88: Different type of formal parameters.

```
active class P {
    virtual void Op (Integer x) {  }
}
active class C : P {
    redefined void Op(Real x) { }
}
```

TSC0196: A finalized operation cannot be redefined.

Operation in the parent class is finalized, but it has the same signature as in the child.

Example 89: Finalized operation

```
active class P {
    finalized void Op () { }
}
active class C : P {
    redefined void Op() { }
}
```

TSC0236: Operation '<name>' cannot be specified as 'Realized' on a port.

The check will detect the following case:

```
active class <class name>
{
    port <port name> in with <in_name>;
}
```

where `<in_name>` is bound to some operation with the same name.

Example 90

```
active class a {
   void foo()  {}
   port p in with foo;
}
```

This will be reported as an error. To remedy this, `foo()` must be defined in an interface to the active class a.

TSC0237: Operation '<name>' cannot be specified as 'Required' on a port.

The check will detect the following case:

```
active class <class name>
{
    port <port name> out with <out_name>;
}
```

where `<out_name>` is bound to some operation with the same name.

Example 91

```
active class a {
   void foo()  {}
   port p out with foo;
}
```

This will be reported as an error. To remedy this, `foo()` must be defined in an interface to the active class a.

TSC2300: Expression 'any (type)' cannot be of interface or state machine type

The following is an example of this error:

Example 92

```
interface I {
}

active class X {
    Integer Op () {
        switch (any (I)) {
```

```
                        case 5   : { return 1; }
                        default  : { return 0; }
                    }
                }
            }
```

TSC2302: An association from a datatype may not have a navigable remote association end

Since datatypes cannot have attributes, it is illegal to have an association from a datatype. The navigability must always be to the datatype.

This error does not occur by normal usage of the tool.

TSC2303: At most one association end may be aggregate or composite

Since aggregation and composition are different kind of "part-of" constructs, two classes cannot be containers for each other.

Example 93

This situation could occur in the situation shown in Figure 149 on page 343.

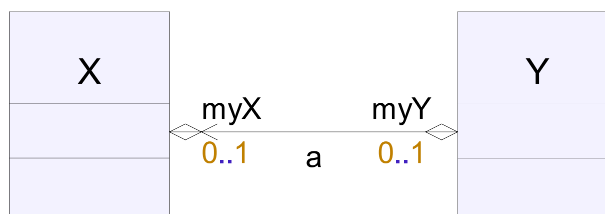


Figure 149: Classes with circular references.

TSC2304: An attribute that is not a part may not have initial count

In UML it is not possible to specify the initial count for regular attributes. That is something that is only possible for parts.

The following is an example of this error:

Example 94

```
class Z {
    Integer [1..*] a / 1;
}
```

TSC2305: A part cannot have a default value

Parts are instances of active classes and they cannot have default values. The following is an example of this error:

Example 95

```
active class X {
    part Y a = 10;
}
```

TSC2306: A composite attribute or association end may not be typed by a datatype

Composite attributes also known as parts in UML must not be instances of datatypes.

The following is an example of this error:

Example 96

```
class X {
    part Integer d;
}
```

TSC2307: A composite attribute may not have a type, which owns this attribute (directly or indirectly)

This is a cyclic dependency error. Since a class cannot be a container for itself this is illegal.

The following is an example of this error:

Example 97

```
class X {
```

```
    part X y;
}
```

TSC2308: The 'via' of a call expression should reference either a port or a connector

The following is an example of this error:

Example 98

```
class Y {}
signal sig ();
active class X {
    port p out with sig;
    void Op () {
        output sig via Y;
    }
}
```

TSC0269: Generalization between 'Interface I' and 'Class Y' is not allowed

The following is an example of this error:

Example 99

```
class Y {
}
interface I : Y {
}
```

TSC2325: Cyclic inheritance

This error is caused if a Signature is based on itself, directly or indirectly.

The following is an example of this error:

Example 100

```
class X : Y {
}
class Y : X {
```

```
}
```

TSC4001: When generating C code, return values must be handled in left hand side of assignment expression

Return values from for example value returning operations must not be ignored. Such return values must be saved in for example an attribute.

Example 101

Consider an Operation `Op`, returning an Integer:

```
Op ():Integer
```

Call to `Op`:

```
...
Integer i;
...
i=Op(); // Correct way of calling Op
Op(); // Error is reported
...
```

This check is performed only when the semantic checker is run in the context of a build which involves any of the C code generators and build types. (Model Verifier, C Code Generator and AgileC Code Generator).

TNR: Name Resolution

The name resolution identifies names of the UML entities and binds them to the correct definition in the model. Name resolution errors are caused by inconsistencies in your model. This may happen when you change names on entities in such a way that ambiguities occur and can not be resolved in a deterministic way.

Errors and warnings from this phase are prefixed with TNR.

TNR errors where the **Subject** of the error refers to the project file (`.ttp` file) rather than to a UML entity, should be reported to DOORS Analyst Support.

TNR0023: Failed to locate element referred by: <name>

Name binding uses the name to refer to an entity in the current scope. GUID binding means that an entity is referred by its unique id (GUID). That means that this error occurs if an entity for some reason is removed and it is referred somewhere in the model by its GUID.

Solutions are to load the entity with the correct GUID, remove the reference or change the reference so it uses name binding.

UML Import and Export

The chapters in this section describe DOORS Analyst's capabilities for importing and exporting data in external formats from and to a UML model. This includes features for information exchange with other modeling tools.

See also

Adding Importers to learn how to add custom importers to DOORS Analyst.

6

UML 1.x Import

This chapter describes the import of UML 1.x models and diagrams created by other UML tools than Telelogic DOORS Analyst.

Operation Principles

XMI

XMI - XML Metadata Interchange - is a UML metadata representation standard based on XML that allows to interchange UML models between different (separate) tools. XMI DTDs (XML Document Type Definitions) serve as syntax specifications for XMI documents, and allow generic XML tools to be used to compose and validate XMI documents.

A UML meta model class is represented in the XMI DTD by an XML element whose name is the class name. The element definition describes the attributes of the class; references to association ends relating to the class; and nested classes, either explicitly or through composition associations.

An attribute of a Metamodel class is represented in the DTD by an XML element whose name is the attribute name.

An association (both with and without containment) between metamodel classes is represented by two XML elements that represent the roles of the association ends.

XMI import

During UML import a file that complies to the XMI standard is read, and after interpreting the contents of the XMI file a UML model is created. After the import has been done, presentation elements (diagrams and symbols) are created in order to visualize the imported contents. Furthermore, if the imported XMI file contains diagram and symbol information that, such information will be use to preserve the appearance of the resulting UML model.

XMI files without any diagram information will be imported, but only UML model elements will be created.

XMI import add-in

The XMI import is provided among the Add-Ins and named **XMIImport**.

XMI import architecture

The architecture of this feature is outlined in Figure 150 on page 353. The XMI Reader module reads a file with XMI specification. XMI Reader transforms information from each tag and passes it to the UML API.

All elements of the UML model are created in the UML API. The core of UML API is a set of C++ classes with the same class hierarchy as in the UML meta model. The UML API is the module builder, which (together with XMI Reader) creates a skeleton of UML model on the fly (tag by tag).

Some kinds of information can not be added to UML model in this phase. This is collected and passed to UML Resolver module.

The U2 Resolver performs a set of transformations to the skeleton of UML model.

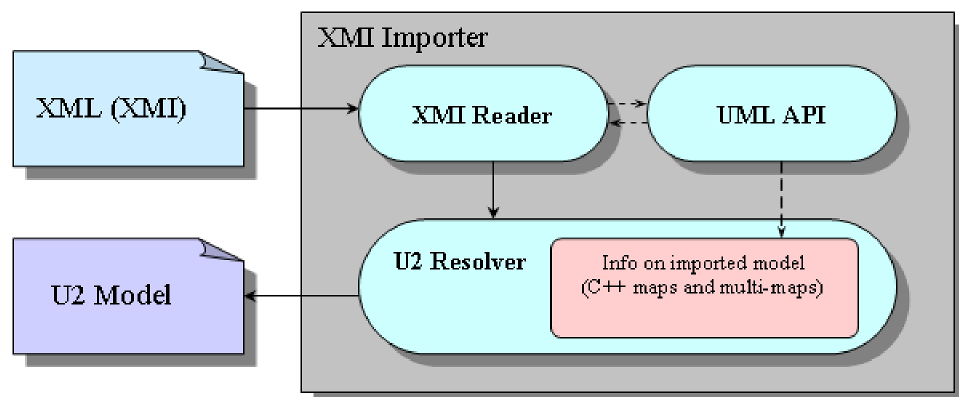


Figure 150: XMI import architecture.

Example 102: UML resolver

An example of information passed to the U2 resolver is import of an "enumeration" data type. For example Rational Rose will export "enumeration" as a class stereotyped by «enumeration», however in DOORS Analyst "enumeration" is a `DataType`. Information about applied stereotypes is not available during Class import, thus this Class must be transformed later. Information about required transformation is passed to the U2 Resolver during stereotype import.

Import an XMI file

The XMI import is called from DOORS Analyst graphical user interface. In order to activate the XMI import, a workspace with a project must be open.

- Select a **Package** in the Model View. (Use the Advanced layout, view tabs are located in the Workspace window)
- Open the Import Wizard (**File** menu, **Import...** command).
- Select **Import XMI** in the dialog window and press **OK**.
- Specify the XMI file to import in the dialog window that appears.

The following should be the result when the second dialog closes:

- A package **ImportedXMIDefinitions** is created in the model
- A stereotype **xmiImportSpecification** is applied to the package.
- The XMI file to import is stored as a value in the stereotype instance for the package.
- The import operation is performed, and the result is added to the created package.

Importing XMI specification with the same settings once again

In the Model View select a package with the **xmiImportSpecification** stereotype applied.

Right-click on the package and in the pop up menu, select **Import XMI**.

The import operation is performed, and the result is placed into the package.

To change settings (select file to import), the properties in the stereotype instance for the package can be edited, before doing an **Import XMI** command.

Note

When the Import wizard dialog is used, a new package is created. When Import XMI from the pop up menu is reused, the existing package is reused.

Supported XMI and UML

Language and version support

The following languages and versions are supported by the XMI import:

- XMI 1.0/1.1
- UML 1.4

Listed below are the UML 1.4 entities that are supported by the XMI import. Relations and attributes to entities are also supported unless specified otherwise.

Foundation / core

```
Association
AssociationEnd
Attribute
Class
Comment
Component
Constraint
DataType
Dependency
ElementResidence
Enumeration
EnumerationLiteral
Generalization
Interface
Method
Operation
Parameter
Permission
StructuralFeature
```

Foundation / extension mechanisms

```
Stereotype
TaggedValue
TagDefinition
```

Foundation / data types

```
Boolean
BooleanExpression
Expression
Integer
Multiplicity
MultiplicityRange
Name
```

```
ProcedureExpression
String
Uninterpreted
```

Model management

```
Model
Package
Subsystem
```

Behavioral elements / common behavior

```
ActionSequence
Argument
CallAction
CreateAction
DestroyAction
Exception
ReturnAction
SendAction
Signal
TerminateAction
UninterpretedAction
```

Behavioral elements / collaborations

```
ClassifierRole
Collaboration
Interaction
Message
```

Behavioral elements / use cases

```
Actor
Extend
Include
UseCase
```

Behavioral elements / state machines

```
CompositeState
CallEvent
FinalState
Guard
Pseudostate
 Initial
 Choice
 Junction
 DeepHistory
 ShallowHistory
SignalEvent
State
SimpleState
StateMachine
```

Supported diagram types

Provided that the XMI file contains the required diagram information, the XMI import supports the following UML diagram types:

- Class diagram
- Component diagram
- Deployment diagram
- Package diagram
- Activity diagram
- Sequence diagram
- Use case diagram
- State machine diagram

Importing with preserved layout

Diagrams that belong to this category are diagrams in which the graphical layout is present in the XMI file.

- Class diagram
- Component diagram
- Deployment diagram
- Package diagram
- Activity diagram
- Use case diagram
- Sequence diagram
- State machine diagram

Import of nested states

Although the layout is preserved some special considerations apply for nested states.

- For each state with nested states a set of diagrams will be created (one for each nested level).
- The positions of states on these diagrams will as far as possible be the same as on original.

- Start and Return symbols will be created on each new diagram when necessary. The positions of these symbols will as far as possible be the same as the position of corresponding symbols on the higher nested level.
- New entry and exit connection points will be created when necessary.
- Transition events and actions containing large amounts of text may overlap.

Import from UML 1.x tools

In general terms, the XMI import tool supports XMI files from the following UML 1.x tools that comply to the supported XMI version(s).

- Rational Rose/Unisys (JCR.2 v.1.3.x)
- DOORS Analyst UML Suite
- Borland Together
- IBM XMI Toolkit.

Rhapsody

Rhapsody exports XMI, but without any diagram information The information in the XMI files originating from Rhapsody is used to create model elements. This will result in a UML structure in the workspace window (but no diagrams).

Rational Rose

- Rational Rose with Unisys extensions exports XMI with diagram information. This information is used during the XMI import when creating the diagrams (provided that the diagrams are among the Supported diagram types).
- Diagram layouts are preserved for class diagrams, use case diagrams and sequence diagrams.
- Rational Rose names are supported.

Preserve DOORS links

It is possible to preserve DOORS links during import of XMI from Rational Rose.

- Export the UML model. Make sure that the “Generate UUIDs” check button is selected.

- Import the generated XMI into DOORS Analyst.
- Export the new UML from DOORS Analyst to DOORS, using the existing DOORS integration commands.
- Open the DOORS Analyst surrogate module in DOORS, and select the menu choice Import Links from Rational Rose and follow the instructions.

When these actions will be completed, all links to or from the surrogate module in DOORS (created by the DOORS Rose Link integration) will then be copied for the DOORS Analyst surrogate module.

DOORS Analyst UML Suite

- DOORS Analyst UML Suite with Unisys extensions exports XMI with diagram information. This information is used during the XMI import when creating the diagrams (provided that the diagrams are among the Supported diagram types).
- Diagram layouts are preserved for class diagrams, use case diagrams and sequence diagrams.

See also

"Language and version support" on page 355

Restrictions

In addition to the level of XMI/UML support stated elsewhere in this chapter, the following sections describe other known restrictions.

Type and variable definitions

- Local datatype definitions are not visible in state machine diagrams.
- Local variable definitions are not visible in state machine diagrams.

Incomplete model

XMI specification must be a complete, semantically correct UML model in order to be imported. In general, incomplete, or incorrect, specifications cannot be imported to DOORS Analyst, however in some cases such specifications can be imported as a complete specification or with losing some model information.

Example 103: Import of incomplete model

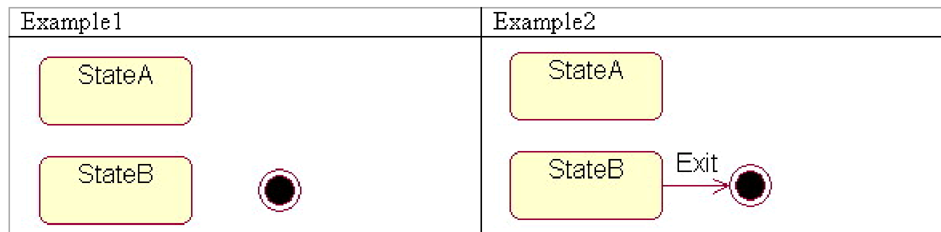


Figure 151: Incomplete models.

In Example1 FinalState will not be imported because this state will be transformed to a ReturnAction. This action should be owned by the incoming (to FinalState) transition, and as such a transition does not exist in the example FinalState will not be imported.

In Example2 all diagram elements will be imported, although this diagram is also incomplete (there is no InitialState in this diagram).

Unsupported classes

Some UML constructs will not be processed during XMI import.

The error message TUI0004 (unsupported classes) will be printed for the following constructs:

Foundation: Core

- Artifact
- Association (between Use Cases)
- Binding
- Flow
- Generalization (between Actors)

Behavioral Elements: Common Behavior

- AttributeLink
- ComponentInstance
- DataValue

- Instance
- Link
- LinkEnd
- NodeInstance
- Object
- Reception
- Stimulus
- SubsystemInstance

Behavioral Elements: ActivityGraphs

- ClassifierInState
- ObjectFlowState
- Pseudostate (Shallow history and Deep history)

Behavioral Elements: Collaborations

- AssociationEndRole
- AssociationRole
- CollaborationInstanceSet
- InteractionInstanceSet

Behavioral Elements: State Machines

- ChangeEvent
- StubState
- TimeEvent

Behavioral Elements: Use Cases

- UseCaseInstance

Unsupported attributes

An error message for unsupported attributes (TUI0006) will be printed for the following attributes:

Foundation: Core

- AssociationEnd
  - Specification
- Attribute
  - AssociationEnd
- BehavioralFeature
  - RaisedSignal
- Component
  - Deployment
- Constraint
  - ConstrainedStereotype
- Feature
  - Owner
- Method
  - Body
  - OwnerScope
- ModelElement
  - Presentation
  - Template
- Operation
  - Concurrency
  - Occurrence
  - Specification

Foundation: Data Types

- Expression
  - Language

Foundation: Extension Mechanisms

- Stereotype
  - Icon
  - StereotypeConstraint

Behavioral Elements: Collaborations

- Collaboration
  - RepresentedClassifier
  - RepresentedOperation
- Interaction
  - Context
- Message
  - Activator

Behavioral Elements: State Machines

- CompositeState
  - Concurrent property

Behavioral Elements: Use Cases

- Actor
  - Abstract property
- Use Case
  - ExtensionPoint

Model Management

- Subsystem
  - Instantiable property

Unsupported composition

An error message for unsupported composition (TUI0008) will be printed for the following constructions:

Foundation: Core

- `AssociationEnd`
  - `Qualifier`
- `Component`
  - `Implementation`

Behavioral Elements: ActivityGraphs

- `State`
  - `InternalTransitions` (actions of an activity)
  - `State` (from `StateMachine`)
  - `Pseudostate`: History (Shallow history and Deep History)

Behavioral Elements: Collaborations

- `Collaboration`
  - `ConstrainingElement`

Behavioral Elements: State Machines

- `StateMachine`
  - `SynchState` (Synchronization bar)
- `State`
  - `InternalTransition` (actions of a `state`)
  - `Pseudostate`: Junction

Collaboration diagram is not supported at all.

Export restrictions

In some cases Rational Rose provides incomplete export. This may result in that some information will be lost after import. The known problems (not exported features) of Rational Rose exporter (Unisys 1.3.6) are listed below.

Class diagram

- Class
  - Type (ParameterizedClass, ClassUtility, InstantiatedClass etc.)
  - Multiplicity
  - Space
  - Concurrency
  - Format (show visibility)
- Attribute
  - Containment

- Operation
  - Protocol
  - Qualification
  - Size
  - Time
- Binary Association
  - Constraints
  - Containment
  - Derived
  - Friend
  - LinkElement
  - Name Direction
- Inheritance
  - Documentation
  - Virtual inheritance
  - Friendship Required
- Realization
  - Documentation
- Dependency/Instantiates
  - Multiplicity from
  - Multiplicity to
  - Friendship Required

State diagram

- Transition
  - Stereotype
  - Documentation

Sequence diagram

- Message
  - Frequency (periodic, aperiodic)
- Destruction Marker

Use Case diagram

- Actor
  - Type
  - Multiplicity
- Use Case
  - Stereotype
  - Rank
- Binary Association
  - Derived
  - Link Element
  - Name Direction
  - Constraints
  - Friend
  - Containment
- Dependency

Package diagram

- Dependency
  - Documentation

Component diagram

- Package
  - Global
- Component
  - Declarations

Deployment diagram

- Processor
  - Scheduling
- Process
  - Priority
- Device
  - Stereotype

- Connection

Activity diagram

- Swimlane
  - Documentation
- Object
- Object Flow

Error Messages

General

Messages during XMI import are printed in the Output window.

Messages from XMI import

Code	Text	Comment
TUI0004	Attribute '<name>' (<name>) of class '<name>' is unsupported	Error occurs when XMI specification contains attribute that is not specified in XMI standard or it cannot be applied to current class in DOORS Analyst. For example, attribute 'isAbstract' of class 'Actor' cannot be applied in DOORS Analyst.
TUI0006	Composition '<name>' from class '<name>' to class '<name>' is unsupported	This error message will be printed in case when composition from one class to other class is unsupported. For example, the 'qualifier' composition is unsupported in DOORS Analyst.
TUI0008	Class '<name>' is unsupported	Error occurs when imported XMI specification contains unsupported class, for example 'Instance'.
TUI0009	Graphical element '<name>' of class '<name>' was not drawn	This error message will be printed in case when a corresponding ModelElement cannot be found for a PresentationElement. For example, the corresponding ModelElement is unsupported.
TUI0010	Diagram representation of '<name>', with value '<name>', is unsupported	Error occurs when presentation element is not supported by DOORS Analyst. For example, a PresentationElement for stereotype.

Code	Text	Comment
TUI0016	Failed to open file '<name>'	File passed to XMI Importer cannot be open
TUI0017	Parse error occurred during parsing of XMI file	This error message will be printed in case when XML parser cannot read information from XMI specification. For example, XML parser cannot find end tag.
TUI0022	Internal error	Internal error occurs during importing.

7

UML 1.x Export

This chapter describes how DOORS Analyst supports export of model data in XMI format to Tools using UML 1.x.

XMI Export

Operation principles

The UML exporter generates a file format that complies with the XMI standard. During export, both model elements and presentation elements are written out to the file. Diagram and symbol layout information is included in order to preserve the appearance of the UML model according to Unisys XML plug-in.

XMI export add-in

The XMI export is provided among the Add-Ins and is named **XMIExport**.

Export to an XMI file

The XMI exporter is called from the DOORS Analyst graphical user interface.

- Initiate the XMI Export (**Tools** menu, **Export Model to XMI...** command).
- Specify the XMI file to export to in the next dialog window that appears.

When more than one project exists in the workspace, the XMI export is done on the selected project. Otherwise, i.e. when zero or more than one projects are selected, the menu choice is dimmed.

Supported XMI and tool versions

The XMI exporter supports the following:

- XMI 1.1

The XMI exporter is tested for the following target tool environment:

- Rational Rose Enterprise Edition 2003
- Rose XML Tools (UniSys XML plug-in) 1.3.6 for Rational Rose

Supported UML entities

Following is a list of tables covering DOORS Analyst UML entities that are supported by the XMI exporter and shows:

- **UML Entity**
  The UML entity - in DOORS Analyst
- **Export**
  The resulting entity in Rose if exported from DOORS Analyst and imported into Rose.
- **Roundtrip**
  The resulting entity in DOORS Analyst if doing an XMI roundtrip.

All other entities not mentioned in this list are not exported.

UML Diagram	Export	Roundtrip
Activity diagram	[Same]	[Same]
Class diagram	[Same]	[Same]
Component diagram	Class diagram	Class diagram
Deployment diagram	Class diagram	Class diagram
Package diagram	Class diagram	Class diagram
Sequence diagram	[Same]	[Same]
State machine diagram	[Same]	[Same]
Text diagram	Class diagram with a note	Class diagram with a note
Use case diagram	Class diagram	Class diagram

General	Export	Roundtrip
Comment symbol	Note	[Same]
Annotation line	Anchor	[Same]
Text symbol	Note	Comment symbol
<Any>		
Comments	Documentation	Nothing
Stereotype	[Same]	[Same]

General	Export	Roundtrip
Links	Files	Nothing
Color	[Same]	[Same]
Font	[Same]	[Same]

Activity diagram	Export	Roundtrip
Activity symbol	[Same]	[Same]
• Name	[Same]	[Same]
Actions	'Entry' action	Nothing
Activity	Class stereo-typed as «ac-tivity»	Class stereotyped as «ac-tivity»
Initial Node	[Same]	[Same]
Activity Final	End State	Activity Final
Flow Final	End State	Activity Final
Activity line	Transition	[Same]
Text	Transition Label	[Same]
Fork/Join	Synchroniza-tion	[Same]
Decision	[Same]	[Same]
• Name	[Same]	[Same]

Activity diagram	Export	Roundtrip
SendSignalSymbol	Unnamed activity with a 'Do/send' action	Auto-renamed activity without action
AcceptEventSymbol	Unnamed activity with a 'Do/receive' action	Auto-renamed activity without action
Accept-TimeEventSymbol	Unnamed activity with a 'Do/receive' action	Auto-renamed activity without action

Class diagram	Export	Roundtrip
Class	[Same]	[Same]
• Name	[Same]	[Same]
• Abstract	[Same]	[Same]
• Template parameters	Formal arguments	[Same]
• Visibility	Export control	[Same]
Class Attribute	[Same]	[Same]
• Name	[Same]	[Same]
• Type	[Same]	[Same]
• Visibility	Export control	[Same]
• Default value	Initial value	[Same]
• Derived	[Same]	[Same]
Class Operation	[Same]	[Same]
• Name	[Same]	[Same]
• Return type	[Same]	[Same]

Class diagram	Export	Roundtrip
• Visibility	Export control	[Same]
• Raised exceptions	Exceptions	[Same]
Class Operation Parameter	[Same]	[Same]
• Name	[Same]	[Same]
• Type	[Same]	[Same]
• Default value	[Same]	[Same]
Required Interface	Interface	Class stereotyped as «interface»
Realized Interface	Interface	Class stereotyped as «interface»
Interface	[Same]	Class stereotyped as «interface»
Timer	Class stereotyped as «timer»	Class stereotyped as «timer»
Signal	Class stereotyped as «signal»	[Same]
Stereotype	Class stereotyped as «stereotype»	Class stereotyped as «stereotype»
Operation	Class stereotyped as «operation»	Class stereotyped as «operation»
State machine	Class stereotyped as «`statemachine`»	Class stereotyped as «`statemachine`»
Primitive/Enumeration	Class stereotyped as «primitive»/«enumeration»	Class stereotyped as «primitive»/DataType
Artifact	Class stereotyped as «artifact»	Class stereotyped as «artifact»
Collaboration	Class stereotyped as «collaboration»	Class stereotyped as «collaboration»

Class diagram	Export	Roundtrip
Choice	Class stereotyped as «choice»	Class stereotyped as «choice»
Association line	[Same]	[Same]
• Name	[Same]	[Same]
Association role	[Same]	[Same]
• Name	[Same]	[Same]
• Visibility	Export control	[Same]
Constraints	[Same]	[Same]
Multiplicity	[Same]	[Same]
Aggregation	Aggregate, Containment	[Same]
Owner scope	Static	Nothing
Generalization/Realization line	[Same]	[Same]
Dependency line	[Same]	[Same]
Extension line	Dependency stereotyped as «extend»	Dependency stereotyped as «extend»

Component diagram	Export	Roundtrip
Component symbol	Class stereotyped as «component»	[Same]

Deployment diagram	Export	Roundtrip
DeploymentSpecificationSymbol	Class stereotyped as «deploymentSpecification»	Class stereotyped as «deploymentSpecification»
ExecutionEnvironmentSymbol	Class stereotyped as «executionEnvironment»	Class stereotyped as «executionEnvironment»
NodeSymbol	Class stereotyped as «node»	Class stereotyped as «node»

Package diagram	Export	Roundtrip
Package	[Same]	[Same]
• Name	[Same]	[Same]
Dependency line	[Same]	[Same]

Sequence diagram	Export	Roundtrip
Lifeline	[Same]	[Same]
• Name	[Same]	[Same]
• Type	Class	[Same]
Message	Simple message	[Same]
• Name	[Same]	[Same]
Method call	Procedure Call message	Message
• Name	[Same]	[Same]
Method reply	Return message	[Same]
• Name	[Same]	[Same]
Timeout	Timeout message	Message

Sequence diagram	Export	Roundtrip
• Name	[Same]	[Same]
Create line	Message where the name is suffixed by ‘:{Create}’	Message where the name is suffixed by ‘:{Create}’
Interaction	Class stereotyped as «interaction»	Class stereotyped as «interaction»

State machine diagram	Export	Roundtrip
State	[Same]	[Same]
• Name	[Same]	[Same]
Multi-state (state with a state list or asterisk state)	State with state name set to the original state text	State with state name set to the original state text
Transition line	[Same]	[Same]
Label	[Same]	[Same]
Decision	[Same]	[Same]
Decision question	Name	[Same]
Decision answer symbol	Transition Guard Condition	Transition Guard Condition
Start	[Same]	[Same]
Stop	End State	Return
Return	End State	[Same]
Flow line	Transition	Transition
Signal Receipt	Transition Event	Transition Event

State machine diagram	Export	Roundtrip
Guard symbol	Transition Guard Condition	Transition Guard Condition
Action symbol	Transition Action	Nothing
Signal sending	Transition Send Event	Nothing

Use Case diagram	Export	Roundtrip
Actor	[Same]	[Same]
• Name	[Same]	[Same]
• Visibility	Export control	[Same]
Use Case	[Same]	[Same]
• Name	[Same]	[Same]
Performance line	Association stereotyped as «performance»	[Same]
Dependency line	[Same]	Nothing
• Name	[Same]	Nothing
Generalization line	[Same]	[Same]

Model hierarchy

The containment hierarchy in Rational Rose is structured as shown in Figure 152 on page 381.

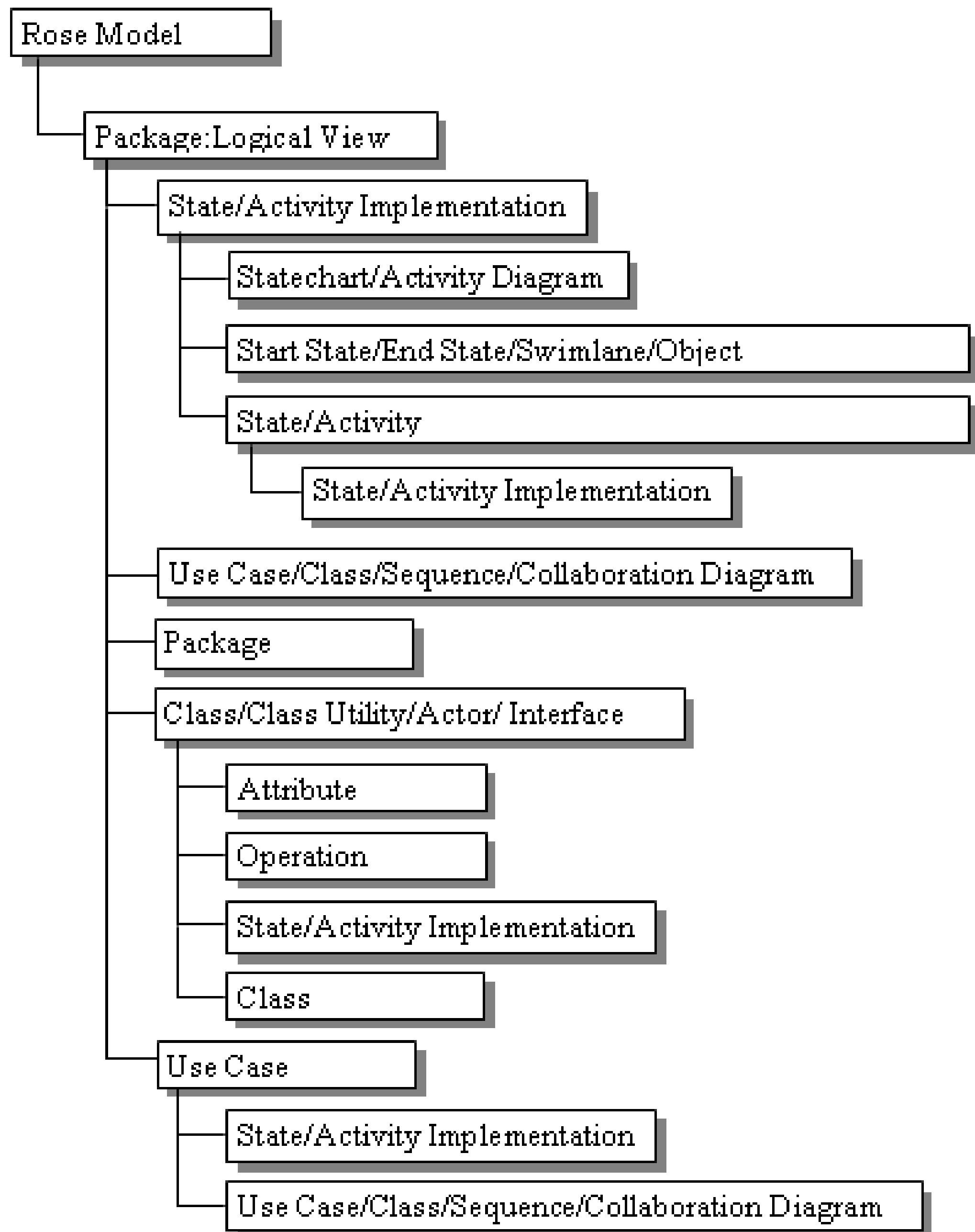


Figure 152: The containment hierarchy

Rational Rose views are defined as packages. Logical View is a hard coded predefined package, which is the default place where the Rational Rose XMI module imports model elements and diagrams.

State/Activity Implementations represent state machine specifications and are placed directly beneath the element for which they apply. The hierarchy can be infinitely deep since Packages and Classes (via Classes, Class Utilities, Actors and Interfaces) can be nested. All elements can have Files and URLs beneath them.

As a general rule, containments in an XMI file that are not supported will be lost on XMI import.

Model transformations

Some transformations take place in order to preserve as much model information as possible.

The table below shows:

- DOORS Analyst entity.
- A description of the reason to move entities in exported XMI.
- Entities that are moved up in the hierarchy if contained by the DOORS Analyst entity.

DOORS Analyst	Description	Moved Entities
Internals	This has no counterpart.	`Class diagram, Package diagram, Text diagram, UseCase diagram, Activity, Actor, Artifact, Association, Attribute, Choice, Class, Collaboration, DataType, Interaction, Interface, Operation, Signal, StateMachine, Stereotype, Timer, UseCase`
State Machine Implementation	This level is very restricted regarding the types of entities allowed.	`Activity, Actor, Artifact, Association, Attribute, Choice, Class, Collaboration, DataType, Interface, Operation, Signal, Stereotype, Timer, UseCase, Class diagram, Package diagram, Text diagram, Use Case diagram`
Activity Implementation	This level is very restricted regarding the types of entities allowed.	`Actor, Artifact, Choice, Class, Collaboration, DataType, Interface, Signal, Stereotype, Timer, Use Case diagram`
Interaction Implementation	This has no counterpart.	`Activity, Actor, Artifact, Attribute, Choice, Class, Collaboration, DataType, Interface, Operation, Signal, StateMachine, Stereotype, Timer, UseCase, Sequence diagram, UseCase diagram`
Nested classes	State machine diagrams beneath nested classes are not imported.	`State machine diagram`

DOORS Analyst	Description	Moved Entities
Class	Interface beneath classes are not imported.	`Interface`
Attribute	Do not contain anything.	`Artifact, Choice, Class, Collaboration, DataType, Interface, Stereotype`
Choice	Transformed into a class.	`UseCase`

Restrictions for XMI export to Rational Rose

There are a number of limitations in the Rational Rose XMI Import on DOORS Analyst exported XMI data. The following is a list of known issues.

General Features	Description
Visibility options	These settings cannot be transferred through XMI and therefore sometimes diagram elements overlap if they do not have the same visibility options set as in DOORS Analyst. Examples of this are Class attributes, operations and operation signatures. If they are switched off in DOORS Analyst, but switched on when importing the XMI data, this might cause symbol overlap because of the resulting difference in size of Class symbols.
Diagram types	Use Case diagrams are imported as Class diagrams in Logical View.
Lines	Lines lose their vertices on import.
	Line color is not imported.
Elements	Cannot import more than one instance of a symbol (e.g class) in the same diagram.
Notes	Size is not imported.
	A note gets duplicated once for each of its anchors.

Activity diagram	Description
Activity	When an activity has both a stereotype and a sub-activity beneath it, it does not import/display properly.
	Fill color, Font and Font size are not imported.
Decision	Fill color, Font and Font size are not imported.
Object	Not imported.

Class diagram	Description
Class	Multiplicity is not imported.
	A Class beneath an Interface is not imported.
	Attributes and Operations are not imported for nested classes.
Class Attribute	Static is not imported.
Interface	Attributes are not imported.
Package	Font, Font size and Fill color are not imported.
Association	Derived is not imported.
	Constraints is not imported.

Package diagram	Description
Package	Fill color, Font and Font size are not imported.

Sequence diagram	Description
Lifeline	The horizontal spacing may not show correctly in exported XMI, especially if the associated text is long.
	Fill color, Font and Font size are not imported.

Sequence diagram	Description
Message	Space is added vertically between messages on import.
	Messages on the same vertical coordinates get into different levels on the lifeline.
	Line color, Font and Font size are not imported.
Destruction Marker	Not imported.
Note	Does not connect Anchors to Messages on import.

State machine diagram	Description
State	If a state exists more than once in the same diagram, only one symbol is imported.
	Fill color, Font and Font size are not imported.
Decision	Fill color, Font and Font size are not imported.
Transition line	Line color, Font and Font size are not imported.

Use Case diagram	Description
Actor	Size is not imported.
	Multiplicity is not imported
Use Case	Stereotype is not imported.
	Size is not imported.
Dependency	Not imported if drawn between Use Cases.

Error and warning messages

Error and warning messages are given in the Output window in a tab called **XMIExport** and these messages are all navigable.

UML entities that cannot be represented in XMI generates an error message.

There is a warning message given when UML entities are transformed or moved in the containment hierarchy. This is due to incapabilities of Rational Rose to handle such constructs,.

Common Reference

The reference chapters listed in this section describe functionality that is valid for all types of DOORS Analyst projects.

8

Printing

This chapter describes different ways of printing a diagram and how to change print settings.

Printing Diagrams

There are several ways of printing diagrams. You can print single diagrams from:

- The diagram itself.
- The Model View.
- The Print Manager.
- The diagram preview window.

You can print multiple diagrams from:

- The Model View.
- The Print Manager.

Note

Using a white/transparent background for an Icon image may result in a black background when printing. This is related to a Windows postscript driver PS level 2. Changing to PS level 1 may remove the situation. Using a colored background or frame will also prevent this.

Print settings

To change print settings:

1. On the **File** menu, select **Print Setup**.
2. In the Print Setup dialog, select printer, paper size and other properties allowed for the selected printer. The paper size and orientation will be used to determine the default diagram size in the editors.
3. Click **OK**.

1. Print files

To print a file:

1. Open the file that you want to print, and place the cursor somewhere in the text.
2. On the **File** menu, click **Print** or click the print icon in the toolbar.
3. In the Print dialog, change settings according to your preferences.
4. Click **OK**.

Select diagrams to be printed

All diagrams in your model are available in the Model View. The Print Manager allows you to select which diagrams to print. To open the Print Manager, click **Print Manager**, on the **File** menu.

The diagrams that are included in the container that is active in the Model View, are listed in the Print Manager. Use the **Track Selection** button if you want to change container in the Model View. If the button is not pressed in, the contents in the Print Manager is locked to the first selection you made.

Figure 153: Track selection button, when not selected

You can also decide which type of diagrams you want to print by checking or clearing the diagram type check boxes in the **Filter** area.

You can calculate the number of pages to print by clicking **Pages** in the **Print** window.

Preview of diagrams

To get a preview of a diagram:

1. Select the diagram in the Model View.
2. Select **Print Preview** on the **File** menu. A preview of the diagram is displayed.
   – You can scroll to other diagrams by using the **Next Page** and **Previous Page** buttons.

Print a single diagram

To print a single diagram from the diagram itself:

1. Open the diagram.
2. Select **Print** on the **File** menu. The standard print dialog is displayed.

To print a single diagram from the Model View:

1. Select the diagram in the Model View.
2. Right-click the diagram and select **Print**. The standard print dialog is displayed.

To print a single diagram from the Print window in the Print Manager:

1. Select the diagram in the Model View. The diagram icon is displayed in the **Selection** area.
2. Click the **Print** button. The standard print dialog is displayed.

To print a single diagram from the preview window:

- Select **Print**. The standard print dialog is displayed

Print multiple diagrams

To print multiple diagrams from the Model View:

1. Select the diagrams in the Model View.
2. Select **Print Manager** on the **File** menu. The **Print** window is displayed.
3. Click the **Print** button or select **Print Preview** on the **File** menu and then **Print**. The standard print dialog is displayed.

You can print diagrams of the same type(s) at the same time if you use the Print window and the Filter functionality.

To print multiple diagrams from the Print window:

1. Select **Print Manager** on the **File** menu. The **Print** window is displayed.
2. In the Model View, select the diagram(s) you want to print. The diagrams and page numbers for the diagram type(s) you selected are displayed in the **Selection** area.
3. Click the **Print** button or select **Print Preview** on the **File** menu and then **Print**. The standard print dialog is displayed.

9

Internationalization Support

This section describes the internationalization support in Telelogic DOORS Analyst/Developer and DOORS Analyst/Architect. The main focus of this document is Chinese, Japanese and Korean (CJK) language handling.

Supported environments

This section describes specific information for Internationalization support of system environments. The information not described in this section is common through all languages. Please refer to the installation guide for general information.

Supported platforms

The internationalization support in DOORS Analyst is available for Windows 2000 and XP. It is assumed that you use a local version of Windows and set the locale to use your local language.

Configuration Management

DOORS Analyst does not support CJK environments beyond limitations of each configuration management tool for CJK support.

IME (Input Method Editor)

Default IMEs bundled in Windows are supported. Using supported IME, you can enter your local characters inline.

Font settings

By selecting the correct font for your language, your language is displayed correctly.

1. Select **Tools** and then **Options** from the DOORS Analyst menu bar**.**
2. Select **Format** tab.
3. Choose **Category** and specify font type
   - **Dialog fixed** : the font type setting for dialogs using a fixed width font.
   - **Developer diagram symbol font**: the font type setting for other symbols and diagrams.
   - **Report Windows**: the font type setting for tabs in the Output window.
   - **Output Windows**: the font type setting for Message and Script tabs in the Output window.
   - **Tcl Files**: the font type setting for Tcl and text files opened in DOORS Analyst.
   - **C/C++ Header/Source**: the font type setting for C/C++ header and source files opened in DOORS Analyst.

Note

The instructions presented below should be performed before you start to create elements in your diagrams.

There is also fonts settings for diagrams elements.

1. Select **Tools** and then **Options** from the Tau menu bar.
2. Select **Font settings** tab.
3. Specify font types. See “Font settings” on page 418.

Note

You can also change the font style and size for each element from the Diagram element properties toolbar.

Modeling with CJK characters

DOORS Analyst supports modeling with CJK characters. You can use CJK characters for

- names of all elements
- comments
- Charstring literals.

You can type CJK characters in the same way as English characters. No special operation is needed to draw models with CJK characters.

Preconditions for using CJK characters

In order for CJK characters to display properly in diagrams inside DOORS formal modules, the following language settings must be properly set:

- Language for non-Unicode programs
- Code page conversion tables

To check or change these settings:

- In the control panel, the setting **Regional and Language Options->Advanced->Language for non-Unicode programs** shall be set to the desired language
- The **Code page conversion tables** setting should include code pages for the desired language

In addition a font with the correct national characters must be used in the diagrams.

To change the default font for new diagrams:

- In **Tools->Options...->Format** change the following fonts:

```
Developer diagram symbol font (for normal symbols)
Developer diagram code font (for fixed text symbols)
```

Note 1

The font options are only used for newly created diagrams, old diagrams have to be changed manually with the diagram element properties toolbar.

The `Developer diagram symbol font` property is set on each diagram and `Developer diagram code font` is set in each fixed text symbol such as `Text symbol`, `Task symbol`, `Comment symbol`, etc.

Note 2

If all settings aren't properly set up as described above, the diagrams will look fine in DOORS Analyst, but bad inside a formal module in DOORS.

This is because DOORS Analyst uses font substitution, so it will always display characters even if they aren't present in the used font.

Handling textual files

Textual files can be opened inside DOORS Analyst. DOORS Analyst supports local ANSI encoding and UTF-8 for the textual file. When existing textual files are opened in DOORS Analyst, DOORS Analyst saves the files in the original encoding. When the textual file is created in DOORS Analyst, the file will be saved in UTF-8 by default. You can select encode type from the **Save as** dialog.

Restrictions

- Single byte Japanese Katakana and Japanese characters defined between 0x80 and 0xFF in Shift-JIS are not supported.
- CJK characters are not supported for Project names.

10

Useful Shortcut Keys

This section lists useful shortcut keys that you can use. Access keys can be used in the same way as other standard applications.

Workspace Operations

Keyboard shortcut	Description
CTRL + N Then CTRL + TAB to Workspaces tab	Create a new workspace
CTRL + O	Open an existing workspace.
MINUS SIGN (-) on the numeric keypad	Contracts the tree of a selected entity.
MULTIPLICATION SIGN (*) on the numeric keypad	Expands the model tree one level below the selection. Can be used repeatedly to expand deeper.
PLUS SIGN (+) on the numeric keypad	Expands the selection.
ALT + 4	Reconfigure Model View, selection of model filter

Project Operations

Keyboard shortcut	Description
CTRL + N Then CTRL + TAB to Project tab	Create a new project
CTRL + O	Open project.

File Operations

Keyboard shortcut	Description
CTRL + N	Create a new file
CTRL + O	Open a file
CTRL + P	Print the active document
CTRL + S	Save active document

Navigate in Files

Keyboard shortcut	Description
CTRL + DOWN ARROW	Scroll down a few rows at a time, without moving the insertion point
CTRL + END	Move insertion point to end of file
CTRL + SHIFT + G	Opens the **Go to line number** dialog
CTRL + HOME	Move insertion point to beginning of file
CTRL + LEFT ARROW	Step left one word at a time
CTRL + M	Open Navigator tab in Output window
CTRL + RIGHT ARROW	Step right one word at a time

Keyboard shortcut	Description
CTRL + UP ARROW	Scroll up a few rows at a time, without moving the insertion point
END	Move insertion point to end of line
HOME	Move insertion point to beginning of line

Highlight Text

Keyboard shortcut	Description
CTRL + SHIFT + END	Highlight text to the end of the file
CTRL + SHIFT + HOME	Highlight text to the beginning of the file
CTRL + SHIFT + LEFT ARROW	Highlight one word at a time to the left
CTRL + SHIFT + RIGHT ARROW	Highlight one word at a time to the right
SHIFT + DOWN ARROW	Highlight one row downwards
SHIFT + END	Highlight to the end of the line
SHIFT + HOME	Highlight to the beginning of the line
SHIFT + LEFT ARROW	Highlight one character at a time to the left
SHIFT + RIGHT ARROW	Highlight one character at a time to the right
SHIFT + UP ARROW	Highlight one row upwards

Edit Text

Keyboard shortcut	Description
CTRL + A	Select all
CTRL + C	Copy
CTRL + F	Find in active file
CTRL + H	Replace
CTRL + SPACEBAR SHIFT + SPACEBAR	Name completion, if a definition is found that matches the current name up to the cursor position. If there are multiple matches a Name completion scroll menu will open.
CTRL + V	Paste
CTRL + X	Cut
CTRL + Y	Redo
CTRL + Z	Undo
F1	Help with textual syntax on current selection.
SHIFT + F8	Restores text from model, discarding comments or user added formatting.
SHIFT + arrow keys	Extends the current text selection. Requires that text is selected
SHIFT + END	Selects text from cursor position to end of text row.
SHIFT + HOME	Selects text from start of text row to cursor position.

Editor Shortcuts

Keyboard shortcut	**Description**
Arrow key	Selects the symbol in the direction of the arrow, requires current selection
CTRL + <click when placing symbols in diagram>	Allows you to place a number of symbols of the same type. Requires that you first select a symbol from the symbol toolbar.
CTRL + <click when placing symbols in state machine flow>	Allows you to insert a symbol in the flow. Requires that you first select the preceding symbol or flowline in the flow.
CTRL + <click word in diagram>	Navigates to definition. If no diagrams contain the definition, the Model Navigator opens.
CTRL + <double-click with a symbol selected in state machine flow>	Selects the entire flow from the selected symbol and downward. Selection will be done on branched flows (multiple signals, decision etc.).
CTRL + <Rotate the wheel button>	Scroll the diagram horizontally (requires an IntelliMouse pointing device)
CTRL + ALT + END	Diagram navigation, go down in diagram scope
CTRL + ALT + Page Down CTRL + ALT + TAB	Diagram navigation, navigate to next diagram in diagram scope
CTRL + Arrow key	Moves the selected symbol 5 grid steps in the direction of the arrow.
CTRL + DELETE	Delete from Model, deletes the presentation element and its corresponding model element. If other presentation elements are connected to this model element, they will also be deleted.
CTRL + DIVISION SIGN (/) on the numeric keypad	Hide all operations. (Valid for signature symbols: class, timer, signal, interface, operation, state machine, datatype, enumeration...)
CTRL + F3	Jumps to the next presentation element of the same model element

Keyboard shortcut	**Description**
CTRL + SHIFT + F3	Jumps to the previous presentation element of the same model element
CTRL + MINUS SIGN (-) on the numeric keypad	Hide all attributes, parameters. (Valid for signature symbols: class, timer, signal, interface, operation, state machine, datatype, enumeration...)
CTRL + MULTIPLICATION SIGN (*) on the numeric keypad	Show operations. (Valid for signature symbols: class, timer, signal, interface, operation, state machine, datatype, enumeration...)
CTRL + PLUS SIGN (+) on the numeric keypad	Show attributes, parameters. (Valid for signature symbols: class, timer, signal, interface, operation, state machine, datatype, enumeration...)
CTRL + SHIFT + <click symbol in toolbar>	Interaction overview and Activity diagram: Append a symbol and toggle orientation. Symbol position will be in the currently not selected orientation. (Append requires that a symbol is selected.)
CTRL + SHIFT + Arrow key	Moves the selected symbol 1 grid step in the direction of the arrow.
CTRL + SHIFT + M	Open Create Presentation dialog
CTRL + TAB	Switches to the next open diagrams
CTRL+ALT + HOME	Diagram navigation, go up in diagram scope
CTRL + ALT + Page Up CTRL + ALT + SHIFT + TAB	Diagram navigation, navigate to previous diagram in diagram scope
ESC, DELETE <Right-click canvas>	Aborts line creation
F2	Enters the edit mode on a selected symbol
F4	Moves to the next selection in the Output window
SHIFT + <click symbol in toolbar>	Create and append a symbol in the diagram. Symbols that cannot be auto-created appear dimmed. (Append requires that a symbol is selected.)

Keyboard shortcut	Description
SHIFT + Arrow key	Selects the symbol in the direction of the arrow and adds it to the selection. (requires current selection)
SHIFT + F4	Moves to the previous selection in the Output window
ALT + UP ARROW	Moves a selected node up in the Model View.
ALT + DOWN ARROW	Moves a selected node down in the Model View.
SHIFT + ENTER	Shows the model element of the selected diagram element in the Model View.
F8	Check the current selection.
CTRL + F8	Do a check of the entire model.
SHIFT + SPACEBAR	Auto creation. All elements that can be auto created on the current selection will be displayed. See Auto placement.
CTRL + SPACEBAR	Auto insertion. All elements that can be auto inserted after the current selection will be displayed. See Auto placement.
CTRL + R	Route selected lines and assign new endpoints.

Window Navigation

Keyboard shortcut	Description
ALT + 1	Toggle full screen mode
CTRL + F2	Toggles definition at cursor position as Bookmark in the Model Navigator
CTRL + F4	Close the active window

Keyboard shortcut	Description
CTRL + SHIFT + TAB CTRL + SHIFT + F6	Navigate to the previous window
CTRL + TAB CTRL + F6	Navigate to the next window
SHIFT + F2	Displays Model Navigator with context of definition at cursor position.

Properties editor

Keyboard shortcut	Description
ALT + ENTER	Display Properties editor
CTRL + BACKSPACE	Go to owner, change scope in model tree to the owner of the current selection
CTRL + ALT + C	Switch to Control view
CTRL + ALT + T	Switch to Text view

Show/Hide Windows and Dialogs

Keyboard shortcut	Description
ALT + 0	Show/ hide workspace window
ALT + 2	Show/ hide Output window
ALT + ENTER	Display Properties editor
CTRL + Q	Open Query dialog on selection
F1	Display Help

Zoom/Pan

Keyboard shortcut	Description
<Rotate the wheel button>	Scroll the diagram vertically (requires an IntelliMouse pointing device)
<Double-click middle mouse button>	Zoom to 100%
SHIFT + <double-click middle mouse button>	Zoom to fit editor window
SHIFT + <rotate wheel button>	Zoom in or zoom out depending on the rotate direction.The zoom in point will be where the mouse pointer is located
CTRL + SHIFT + <Rotate the wheel button>	When a single line is selected the diagram will be scrolled along the line until one of the endpoints are centered in view (requires an IntelliMouse pointing device)
MINUS SIGN (-) on the numeric keypad	Zoom out 25% (This works when a diagram is active and not in text edit mode for any element)
PLUS SIGN (+) on the numeric keypad	Zoom in 25% (This works when a diagram is active and not in text edit mode for any element)
LESS-THAN SIGN (<)	When a single line is selected the diagram will be scrolled to the source endpoint of the line.
GREATER-THAN SIGN (>)	When a single line is selected the diagram will be scrolled to the destination endpoint of the line.

11

Dialog Help

This section lists the help texts that are displayed when you click the help button in dialogs.

The New Wizard

Files tab

This dialog provides the possibility to add new files to your design.

- When adding a file you must specify a file name and a location.
- The file can be added to an existing project. The project must be opened in the File View in order for you to add the file to it.
- The new file is opened in the Desktop.

Projects tab

This dialog provides the possibility to add a new project.

When you add a project, you specify how the project will be used. Depending on your choice, different add-ins will be loaded at start-up, for example:

UML for Modeling
No add-ins are loaded.

- When adding a project you must specify a project name and a location.
- The project can be included in the current workspace, or a new workspace can be created for the project.

See also

UML Projects - page 2

This dialog displays a suggested file directory and a suggested name for the file holding the model.

- You can change or confirm the suggestions.
- As an option, an empty package can be added.

UML Projects - page 3

This dialog displays the name of the project and the name of the related file.

- You can confirm the names by clicking the Finish button or enable changes by clicking the **Back** button.

- The new project appears in the Workspace window.

Workspaces

This dialog provides the possibility to add a new workspace.

- When adding a workspace you must specify a workspace name and a location.
- The new workspace is loaded in the Workspace window.

Customize

Commands tab

This tab lists the default menus with toolbar buttons, commands and menus that you can add to a toolbar or menu. It allows you to move, delete or add buttons to your toolbars.

1. In the **Categories** box, click the toolbar name that you want to customize.
2. In the **Buttons** area, drag the item from the dialog on to the toolbar. Click the item first to receive information about the specific item.
3. To remove an item from a toolbar, drag the item from the toolbar on to the dialog.

To add a button to a toolbar:

1. Make sure that the toolbar you want to change is displayed.
2. In the **Categories** box, the available toolbar buttons or items are grouped. Select the category where the toolbar button or item you want to add is located.
3. Click a button or item to receive information about its functionality.
4. Drag the button or item from the **Buttons** area to the toolbar in the user interface.

To delete a button from a toolbar:

1. Make sure that the toolbar you want to change is displayed.
2. Drag the button or item off the toolbar.

When you delete a default button from a toolbar, the button is still available in the Customize dialog box. However, when you delete a toolbar button with a custom appearance, its appearance is permanently lost, although the command is still available (Customize dialog box, Commands tab).

Hint

To save a toolbar button with a custom appearance for later use, create a toolbar for storing unused buttons, move the button to this storage toolbar, and then hide the storage toolbar.

Toolbars tab

This tab lists standard and custom toolbars.

Select or clear the check boxes to display or hide the toolbars. Each toolbar appears either in the default location or in the last location that it is moved to. The menu bar cannot be hidden.

Show Tooltips
Click the check box to enable tooltips to be displayed when the cursor moves over a button or field in the toolbars.

Large Buttons
Click the check box to display larger sized buttons in the toolbars.

Create a new toolbar:

1. Click **New**.
2. In the dialog that opens, type the name of the toolbar. The new toolbar appears in the toolbar area of the interface.
3. From the **Commands** tab, select the items that you want to add to the toolbar.

Restore the default toolbar settings:

1. Click the toolbar in the list.
2. Click **Reset**.

A user-created toolbar cannot be restored.

Delete a user-created toolbar:

1. Click the toolbar in the list.
2. Click **Delete**.

A default toolbar cannot be deleted.

Rename a user-created toolbar:

1. Click the toolbar in the list.
2. In the **Toolbar Name** field, type a new name for the toolbar.
3. Click the toolbar again to save the change.

Create New Toolbar

Type the name of the new custom toolbar. You can use upper or lower case letters, but each name must be unique regardless of case. The name must be unique from other toolbars. If you want to change this name later, you can edit the name in the Toolbar Name box on the Toolbars tab.

Windows layouts

This tab allows you to customize the appearance of the Windows layout. You can save toolbar positions, visibility and location of docked windows.

Save a new layout:

1. Click the New button.
2. Type a name for your layout.
3. Close the window.

To restore a new layout:

1. Click the layout you want to restore.
2. Click Restore.

To delete a layout:

1. Click the layout you want to delete.
2. Click the Delete button.

Tools tab

This tab allows you to add commands in the Tools menu. These commands can be associated with any program that runs on your operating system. The information is saved in a file named Tools.dat in the directory:

```
C:\Documents and Settings\<user>\Application Data\Telelogic\Shared
```

Add a command to the Tools menu:

1. Click the **New (Insert)** button. A blank line, indicated by an empty rectangle, appears in the **Menu Contents** box.
2. Type the name of the command as it will appear in the Tools menu. Press **ENTER** to save the name.
3. In the **Command** field, type the path to the program. You can also use the browse button to locate the program.
4. In the **Arguments** text box, browse or type any arguments to be passed to the program. Use the drop-down arrow next to the Arguments text box to display a menu of arguments.
5. In the **Initial directory** box, browse or type the file directory where the command will be located.
6. If the program is a console program, for instance the Windows command prompt, you can select to have it run in the Output window. Just select the **Use Output Window** check box.
7. Select the **Prompt for Arguments** check box, if you want to be able to change argument each time you want to use the command.
8. Select the **Use OEM format** check box, if you want to the application's output to be in OEM format.
9. Click **OK**. The command appears in the Tools menu.

Additional tasks

- To insert the command in a submenu, separate the menu name and the name with a backslash '\'. For instance, the command Notepad in an editor menu should be typed `editor\Notepad`.
- To insert an access key, type an ampersand '&' before the selected letter in the name.
- Move commands up and down in the menu by using the **Move Up** and **Move Down** buttons.
- To change the name of the command, double-click it and type a new name.

Delete a command in the Tools menu:

1. Click the command in the list.
2. Click the **Delete** button.

Add-ins tab

Add-ins are used to extend the tool functionality. From the Add-ins tab you can load a selection of predefined add-ins.

- To load or unload add-ins, select or clear the check boxes. Close the dialog.

Note

It is not recommended to write add-ins in the scope of DOORS Analyst.

See also

“Contents and structure of an add-in” on page 1988 in Chapter 73, *Customizing Telelogic Tau*

Options

General

This tab allows you to set general options:

Display status bar
Allows you to show or hide the status bar that is available at the bottom of the DOORS Analyst user interface.

Show output window when receiving content
When the Output window is closed, information that is normally listed in the different tabs is not displayed. However, when selecting this option, the output window will open automatically when new information is listed, for instance after a manual check.

Track selection in the Print Manager
The Print Manager by default tracks the active selection in the Model View. This option can be turned off to disable this tracking.

Show advanced option page
Select this option to display an additional tab with all options listed in a tree structure. Some advanced option can only be set from this additional tab.

Tabbed documents
Select this option to open documents in a single window as tabs.

Show welcome page at startup
This option controls whether or not the welcome page should be opened when starting the tool. This option can also be set from the welcome page itself. If you turn this option off you can open the welcome page manually from the Help menu.

Source control provider
If you have a source control system installed, you can use this option to enable using it from DOORS Analyst. Doing so will enable a source control menu and toolbar for interaction with your source control system. For more information see Configuration Management.

Automatically update files
This option can be used if Generic Source Control is selected as source control provider. If the option is enabled files will be automatically updated from the CM system before attempting to do a check out.

Disable external program launch for these types of file
In this field you can specify the extension of files, that DOORS Analyst should attempt to open instead of the external application which otherwise is associated with that file extension. For instance, if you add the `*.txt` extension, text files will be opened in the DOORS Analyst text editor instead of in your external text editor application.

Select the default help context
If there are many Telelogic tools installed, you can choose which help file to use as default by selecting the file in this list.

URN Map
Use the **URN Map** (Universal Resource Name) to define shorthand names for file storage locations. For example:

```
home:C:\MyHomeDir;work:C:\MyWorkDir
```

Here “`home`” is shorthand for `C:\MyHomeDir` and “`work`” is shorthand for `C:\MyWorkDir`. Each user may define URNs for his/her environment. These are used by some components for referring to files, bitmaps and other resources.

•

Save

This tab allows you to set save options in DOORS Analyst.

Save before running tools
Select this option to automatically save any unsaved work before an external tool is launched.

Prompt before saving files and projects
Select this option to be prompted for saving when modified files and projects exist when an editor is closed.

Automatic reload of externally modified files
You will by default receive an information message and be prompted to reload an externally modified file. Select this option to avoid this prompting, in order to automatically reload a file that has been modified in another tool than DOORS Analyst.

Save project's add-in state in all the loaded projects
This option will let any loaded add-ins become activated for all projects currently loaded.

Auto-backup
Select the **Activate** check box to allow automatic saves of your model in predetermined intervals. Enter the desired number of minutes between the saves, either by typing the number or by clicking the up and down buttons.

Workspace

This tab allows you to set general options for the workspace that you have opened.

Reload last workspace at startup
Select this option to open the workspace that you were working in the last time DOORS Analyst was running.

Warn on project file status change
Select this option to receive a warning if the status of the project file you are working in has been changed to read-only. This will protect you from potentially loosing unsaved work.

Projects default location
When you create new projects, you will receive a suggestion where the project file will be stored. In this text field, type a path, or browse to a folder, where the new projects will be stored.

Format

This tab allows you to format the appearance of text and colors in windows and files.

When you have selected a category, you can select:

- **Font** and **Size** of the text in the file or window.
- Background color and text color for the selected Category. By default, system colors defined in the control panel are used. Clear the **Automatic** check boxes to select text and background colors.

Font settings

This tab allows you to customize the default fonts used when creating a new diagram.

Diagram font settings
These font settings determine the default text appearance of the generic diagram element in a created diagram.

Fixed font settings
These font settings determine the default text appearance of symbols that have texts which are better displayed using a fixed width font. An example of a symbol that uses this font setting is the Text symbol. If the **Enabled** check box is checked, any created symbol of this kind will have the fixed font settings applied on it.

Label font settings
These font settings determine the default text appearance of text labels that are not the main label of a diagram element. An example of this is the Attribute and Operation labels in a Class symbol. If the **Enabled** check box is checked, any created label element will have the label font settings applied on it.

Links

This tab allows you to customize link creation behavior.

Active link end is an active target, not an active source
If this option is off, then when you use automatic creation of links, you will create links from your active link end to the other models. If this option is on, then you will create links to your active link end from the other models.

Automatically create links between modified objects and active link end
If this option is on, then when you select an active link end, all your modifications will be linked to this link end.

Show link indicators
If this option is on, DOORS Analyst will show the link markers.

Use requirement as target when creating links by drag and drop
Links can be created using drag and drop. If this option is on when doing this on a requirement the target of the link will be the requirement. If the option is off the requirement will instead be the source of the link.

Some of the advanced options are documented below.

Web server

`Studio - Settings - WebServer`

The options **PortRangeBegin** and **PortRangeEnd** define the range of TCP/IP ports used by the Tau Web Server. You may need to change these options if the default port numbers are not available for use on your machine.

Proxy settings

`U2 - Options - ProxySettings`

The options **Host**, **Password** and **User** can be set if you access the web through an HTTP proxy server. They will be used whenever DOORS Analyst accesses information from an URL, for example when importing a WSDL file from an URL. The syntax of the host option is `<address>:<port>`.

Editor Shortcut

Show Elements

This dialog provides the possibility to add multiple elements to a diagram with a selection of symbols from your existing model.

- Elements are selected by checking the check box in the element list.
- The element list contains the elements of the current set scope.
- The Set Scope button is used to add elements from any scope in your model to the element list.

When this dialog is entered as a result of an operation where an element is initially selected this element will be pre-checked in the list.

- The new diagram is opened in the Desktop.

Reconfigure ModelView

Select the browser model that you want to use. There are two predefined browser views. The browser view “Standard View”, gives a comprehensive view of the loaded model including design detail. This view is intended for design-oriented users.

The other browser view “Diagram View” gives a simplified view of the loaded model. This view is intended for analysis-oriented users.

See also

“Metamodel” on page 316 in Chapter 4, *UML Language Guide*

Other

Select Stereotypes

Select the stereotypes that you want to apply to the element. Click each line to see a description of the each stereotype. The number of applicable stereotypes varies depending on the selected element.

See also

“Stereotype” on page 317 in Chapter 4, *UML Language Guide*

12

Additional Resources

This section list documents that are not part of the help file, but that may help you to extend your knowledge about DOORS Analyst. Links to useful web resources are also provided.

Links

Contacting IBM Rational Software Support

Support and information for Telelogic products is currently being transitioned from the Telelogic Support site to the IBM Rational Software Support site. During this transition phase, your product support location depends on your customer history.

Product support

- If you are a heritage customer, meaning you were a Telelogic customer prior to November 1, 2008, please visit the DOORS Analyst Web site. Telelogic customers will be redirected automatically to the IBM Rational Software Support site after the product information has been migrated.
- If you are a new Rational customer, meaning you did not have Telelogic-licensed products prior to November 1, 2008, please visit the IBM Rational Software Support site.

Before you contact 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 workaround for the problem? If so, be prepared to describe the workaround.

Other information

For Rational software product news, events, and other information, visit the IBM Rational Software Web site.

UML documents

- **Java Tutorial**

  The tutorial is available in your installation in:

- **UML Tutorial**

  The purpose of this tutorial is to make you familiar with DOORS Analyst and the UML language. The tutorial addresses persons with knowledge of the basic concepts of how to work with requirements modules in DOORS and also have a basic UML knowledge.

  The tutorial is available in your installation in:

  locale/en/radar_ctrl.pdf

- **UML Quick reference guide**

  This document contains common graphical and textual constructs in UML.

  The guide is available in your installation in:

  locale/en/quickref.pdf

Other links

Cygwin

For information about the contents of various Cygwin versions, see:
http://www.cygwin.com

GNU C/C++

C/C++ dialect supported by the GNU Compiler Collection.
http://www.gnu.org/software/gcc

ITU-T

Formerly CCITT
http://www.itu.int/

Macrovision

For more information about FLEXnet or Macrovision, please see:
http://www.macrovision.com

MISRA

The code generated by the AgileC Code Generator is to a large extent compliant with the MISRA coding rules described in the document “MISRA-C:2004 Guidelines for the use of the C language in critical systems” from October 2004. Please see:

http://www.misra.org.uk

OCL

For more information about OCL (Object Constraint Language), see:
http://www.omg.org

OMG

For more information about Object Management Group (OMG), see:
http://www.omg.org

PDF

PDF files are opened and read with Adobe Acrobat Reader:
www.adobe.com

Tcl

For detailed information refer to the Tcl Developer Site
http://tcl.activestate.com/

TTCN-3

The TTCN-3 standard can be downloaded from
http://www.etsi.org

XML

For information about Extensible Markup Language (XML), see:
http://www.w3.org/XML

Index

Symbols

#, inline code 296
#, private 113
«» 125

A

absolute
  time line 176
abstract
  class, UML 203
access 194
Acrobat Reader 426
action 285
action, UML sequence diagram 179
action, UML state machine 277
Actions 253
activation
  method call 187
active class 212
active timer 296
activity diagram
  operations 243
actor 159
Add
  Stereotype Instance Compartment .206
add
  class in diagram 201
  printers (UNIX) 392
  stereotype 128
  symbol 119
  symbol in activity flow 124
  toolbar button 23
add-ins
  CApplication 410
  ModelVerifier 410
  tab, Customize 415
  XMIExport 372
  XMIImport 352
advanced layout 13
aggregation 311
  association 308
  kind, association 309
  kind, attribute 208
All
  Show Elements 121
All Properties, Properties Editor 49
alt
  inline frame 183
alternative syntax 150
any, UML 294
  expression 294
appearance 188
append
  symbol in activity flow 124
architecture 29
architecture diagram. See composite structure diagram
architecture modeling 235
arrange windows 19
artifact
  UML 303
assert
  inline frame 184
assignment 285
association
  navigable 207
  relationships 308
  use case modeling 161
attribute 206
  class 203
  compartment 214
  DOORS 3
  Object Text 4
  UML Comment Symbol 5
  UML Kind 3
  UML Location 8
  UML Name 8
auto
  placement, in diagram 120

Automatic layout 114
Auto-routed (keep endpoints) 135
Autosize 123
autosize
    diagram 112
    symbols 123

B

behavior
    modeling 265
behavior port 241
    port 215
behavioral elements
    collaborations 356
    common behavior 356
    state machines 356
    use cases 356
bi-direct 239
    line edit 137
bmp 129
bookmark
    help file 33
break
    inline frame 184

C

capture
    minidump 336
cardinality 210
cascade 19
case sensitivity
    UML 149
change
    options 26
Check
    Output window 18
check
    part of a model 45
choice
    UML 227
CJK characters 397
class 200
    abstract 204
    components 205
    external 204
    heading examples 202
    hide attributes 133
    modeling 197
    new 286
    show attributes 133
    signature 323
    this 286
    UML 200
class diagram 199
classifier
    metaclass 320
close
    window 20
Collapsed
    symbol command 123
color
    Properties Editor values 55
column
    Create Presentation 76
    diagram type 76, 83
    item 76, 83
    Model Navigator 82
    type 76, 83
Column of Remarks 126
commands
    customize 411
commands tab 411
comment
    Column of Remarks 126
comment symbol 314, 315
    reference in diagram 126
Comment, Properties Editor 49
Compartment text fields 134
completion 44
component 205
composite state 297
composite structure diagram
    UML 235
composition
    association 308
    relationships 311
compound statement 286
compress layout 188
conditional expression 293
configuration management
    internationalization 395
connect
    symbols 123
connector 239

consider, inline frame 184
constant
UML 211
Constraint compartment 205
constraint symbol 314
continuation
UML 185
convert
UML to C++ style 113
Convert Module .11
copy 127
model elements 42
co-region 184
create
activity diagram 246
compartments 132
diagram 111
interaction overview diagram 190
sequence diagram 164
state machine diagram 267
use case diagram 157
window 20
Create Presentation 76
create symbol
UML 179
critical
inline frame 184
Customize
dialog 411
toolbars 24
customize
commands 411
new toolbar 413
toolbars 412
tools 413
windows layouts 413
cut 127
Cygwin 425

D

dat, tools file extension 413
datatype
UML 224
debug 336
decision 280
answer 280
decomposition .169
deep history
UML 273
default
converts from else 113
default value
UML 209
definitions tab 81
delete 127
attribute 134
element 41
line 137
model elements 143
operation 134
parameter 134
selected signals 188
symbols 143
Delete All Values, Properties Editor 53
Delete Instance, Properties Editor 53
Delete Model 42
Delete Value, Properties Editor 54
dependency
architecture modeling 242
relationship 307
use case modeling 161
derived 153
attribute 210
operation 211
desktop 14
destroy
UML symbol 181
diagram 27
autosize 112
create 111
element properties 125
frame 110
general 27
grid 110
heading 110
move 112
name 111
open 111
operations 110
print 111
save 111
scope 145
size 111
size, print 392

UML 140
zoom 115
Diagram Element Properties 125
Diagram Name column 76, 83
Diagram size 112
Diagram View 17
diagram-centric workflow 40
Diagrams, Model Navigator tab 81
dialog help 409
direct addressing
method application 278
dock
window 21
dock, window 21
docked window 19
Document Type Definition 352
Drag and Drop 97
create presentation 98
from model view to a diagram 98
link 98
within and between diagrams 99
within the model view 97
dynamic behavior
classes 29

E

edit
symbols 127
text 402
text in symbols 125
Edit properties of symbols/lines, Properties Editor 52
Editing vertices 137
editor
shortcuts 403
elements
navigation 80
properties 41, 151
else
converts to default 113
emf 129
Enable Analyst for Section 6
enabled direction 137
entity tabs 81
entry connection point 298
enumerated data type
converts from datatype 114
enumerated type 225
error
XMI export 386
error messages 335
exclamation mark
UML Kind 3
exit
connection point 299
explicit connector 239
export
XMI 372
expression
UML 291
extends 161
extensibility 316
Extensible Markup Language 426
extension 318
relationships 313
external
class 204

F

Favorites
tab 82
Features
tab 81
field
expression 293
file
operations 400
options 25
show in Model view 16
file extension
.bmp 129
.dat 413
.emf 129
.gif 129
.jpeg 129
.jpg 129
.pcx 129
.pdf 426
.targa 129
.tga 129
.tif 129
.tiff 129
.tot 25
.u2 111

.u2x 115
external program launch 416
File View 15
files
tab 410
filter
delete in sequence diagram 188
Model view 16
find 113, 113
Find text in diagrams too 113
float
window 21
floating window 19
flow 290
append symbol 124
insert symbol 124
orientation 247
remove symbol 125
flow line 290
font settings 396
Font settings, options tab 418
Format, options tab 418
found message 174
foundation
core 355
data types 355
extension mechanisms 355
frame 110, 313
full screen 19

G

gate
names 187
text, add/remove 187
general ordering line 177
generalization 307
use case modeling 161
Generate Diagram dialog 84
gif 129
Globetrotter, see Macrovision 425
GNU
C/C++ 425
go to line 25
Goto Owner, Properties Editor 53
Goto Value, Properties Editor 54
grid 110
guard 282, 282
guarded transition 272
guillemets, «» 125

H

heading 110
help
on-screen 31
hide
windows 406
hide windows 20
highlight
text edit 401
history nextstate 272

I

IBM Customer Support 424
icon 128
IconFile 129
Icon mode 129
ignore
inline frame 184
image file 129
IME (Input Method Editor) 396
imperative expressions 294
implementation
activity 249
metaclass 323
signature 322
implicit connector 239
import 194
preserved layout 357
UML 351
UML 1.4 355
UML Suite 359
XMI 352
XMI/UML, restrictions 359
incoming signal 219
incomplete
message 173
index 32
column 82
expression 293
results 33
informal
decisions 281
initialize
state machine 202

inline
- **class** ... 202
- **frame** ... 181

inout
- **converts from in/out** ... 113

input. See signal receipt

insert
- **symbol** ... 119
- **symbol in activity flow** ... 124
- **symbol in flow** ... 124

Interaction ... 164

interaction reference ... 165

interface ... 217
- **port** ... 216

interface symbol ... 217

internals ... 300

internationalization ... 395
- **support** ... 395

J

jpeg ... 129

jpg ... 129

junction ... 289

K

Keep selected signals ... 189

keywords. See reserved words

kind ... 319

L

layout
- **advanced** ... 13

lifeline ... 166

lifeline decomposition ... 168

line
- **aggregation** ... 308
- **association** ... 308
- **bi-direct** ... 137, 239
- **composition** ... 308
- **connector** ... 239
- **delete** ... 137
- **dependency** ... 307
- **flow** ... 290
- **generalization** ... 307
- **go to** ... 25
- **move** ... 137
- **number** ... 25
- **operations** ... 134
- **realization** ... 308
- **re-direct** ... 137, 239
- **simple transition** ... 290

link
- **drag and drop** ... 98

links
- **column** ... 82
- **external** ... 423
- **tab** ... 81, 419
- **web sites** ... 424

List Presentations ... 131

List References ... 130

literal ... 227

Load image ... 129

locate
- **search** ... 33

Location
- **column** ... 76, 82

loop
- **inline frame** ... 183

lost message ... 173

M

Macrovision ... 425

Match Similar Word ... 34

MDI child ... 21

MDI Child window ... 19

menu bar ... 22

message ... 170
- **Output window** ... 18

metaclass ... 317
- **classifier** ... 320
- **implementation** ... 323
- **signature** ... 320, 323

metafeature values, Properties Editor ... 49

metamodel ... 316
- **classes** ... 320
- **model view filter** ... 16
- **profile** ... 320

method ... 322

method call ... 186

minidump ... 336

MINUS_INFINITY ... 320

model checking ... 43

model element ... 142

activity diagrams . . . 247
class diagrams . . . 199, 231
component diagrams . . . 244
deployment diagrams . . . 302
handling . . . 42
modeling workflow . . . 40
name scope . . . 145
presentation element . . . 41
sequence diagrams . . . 163, 192
use case diagrams . . . 156
Model Index tab . . . 82
model management, XMI import . . . 356
Model Navigator . . . 78
tabs . . . 78
model references . . . 130
Model View . . . 15
Filters . . . 16
model-based development . . . 26, 40
model-centric workflow . . . 40
modes
entity . . . 81
link . . . 81
presentation . . . 80
move
diagram . . . 112
line . . . 137
model elements . . . 42
symbols . . . 122
multiple
state machines in class . . . 202
multiplicity
association . . . 309
attribute . . . 209

N

name . . . 147
column . . . 76, 83
completion . . . 44
navigation . . . 43
referencing . . . 44
naming
new elements . . . 42
rules . . . 148
use cases . . . 158
navigable
association . . . 207
end . . . 309
Navigate . . . 78
navigation . . . 80
files . . . 400
help file . . . 31
Model View . . . 131
name . . . 43
Navigator . . . 76
neg
inline frame . . . 184
nested
expressions . . . 35
nested states
import . . . 357
new
create diagram . . . 111
expression . . . 293
instance of class . . . 286
toolbar . . . 413
window . . . 20
New diagram
tab . . . 76
New symbol
tab . . . 76
new toolbar
customize . . . 413
New Wizard . . . 410
nextstate
history . . . 272
nondeterministic decisions . . . 282
None
Show Elements . . . 121
noScope
stereotype . . . 196
now . . . 295

O

object
locate in UML . . . 43
Object Management Group . . . 426
Object Text . . . 4
OCL . . . 426
OMG . . . 426
open
diagram . . . 111
operation
compartment . . . 214
operation, UML . . . 211

body . . . 299
class . . . 203
compartment . . . 214
signature . . . 323
operators . . . 34
opt
inline frame . . . 183
options . . . 25
dialog . . . 415
file . . . 25
format . . . 418
general . . . 415
link . . . 419
save . . . 417
Save As . . . 26
workspace . . . 417
ordering . . . 129
events . . . 168
organizing
view . . . 115
orientation . . . 247
outgoing signal . . . 220
output . . . 279
symbol, converts from ^ . . . 113
window . . . 17
Output window
Check . . . 18
message . . . 18
Presentations . . . 18
References . . . 18
Script . . . 18
search result . . . 18
output. See signal sending . . . 278

P

package . . . 192
modeling . . . 191
Predefined . . . 319
tab . . . 81
page column . . . 83
par
inline frame . . . 183
parameter . . . 154
parse text . . . 113
part . . . 236
communication . . . 241
Partition Reference . . . 253
paste
symbols . . . 127
pcx . . . 129
pdf . . . 426
pdf, Acrobat file extension . . . 426
Pid
expressions . . . 295
placement . . . 120
PLUS_INFINITY . . . 319
port . . . 214
attribute . . . 209
interface . . . 216
type . . . 215
Predefined . . . 319
predefined
data . . . 318
names . . . 155
predicate . . . 89
agent . . . 95
Preferred filter, Properties Editor . . . 52
presentation element . . . 45
in diagram . . . 41
navigation . . . 79
presentation tabs . . . 80
preview diagram . . . 393
primitive datatypes . . . 226
print . . . 111
add printer . . . 392
diagram . . . 111, 111
help topics . . . 33
multiple diagrams . . . 394
settings . . . 392
single diagram . . . 393
priority . . . 80
transition in composite state . . . 297
private, converts from # . . . 113
profile . . . 318
metamodel . . . 320
project
new . . . 410
operations . . . 400
Projects tab . . . 410
properties . . . 43
Properties editor
shortcuts . . . 406
Property View, Properties Editor . . . 51
protected, converts from - . . . 113

public, converts from + ...113
purpose 141

Q

Query
dialog 93
query 89
agent 95
expression 90
quotation marks, automatic 114
quote
automatic typing 114

R

range check 296
Rational Rose 358
realization, UML 308
realized interface 219
Real-time profile 324
receiver
attribute 279
expression 279
this 279
Recent tab 82
Reconfigure Model View 16
re-direct 137, 239
lines 137
redo 130
shortcut 402
reference
definition 44
existing definitions 119
to model 130
Reference existing 119
messages 171
name support 44
references tab 81
regular expressions 34
relationships
class 229
collaboration, use case 160
composite structure 242
UML 306
relative time line 176
Remember scroll and zoom 115
remove
printers (UNIX) 392
symbol from activity flow 125
Remove image 129
required interface 220
reserved words 150
resize
diagrams 112
symbol indicators 123
symbols 122
resource
meta class base set 16
restore model (f8) 44
restrictions 8
Clone 8
Diagram below 8
Import Partition 8
internationalization 398
multiple servers 8
XMI export 384
XMI import 359
return 288
return value, method call 187
Rhapsody 358
role
actor 159
column 83

S

save 287
Auto-backup 417
diagram 111
dialog 417
options tab 417
scenario
as sequence diagram 29
modeling 162
scope 145
scope unit
UML 146
script
Output window 18
scroll, window 115
search 35
help file 31
help file, examples 35
help file, highlighting 32
syntax in help 34
search. See find.

select
diagrams for print 393
flow 124
metamodel 420
stereotypes 421
symbols 121
Select Attribute to Show in Analyst 11
Select scope
Show Elements 121
selector
expression 170
semantic
check 337, 339
errors 339
seq
inline frame 183
sequence diagram 162
shallow history 273
shareable edit 6
Short list
Show Elements 121
shortcut
column 83
shortcut keys 399
shortcuts
as toolbar 17
tab 81
window 17
Show
Comments 126
Constraints as Compartments 205
Constraints as Symbols 126
Stereotypes as Symbols 206
show
diagram 16
dialogs 406
element 420
file 16
implementation 16
windows 20, 406
Show all
Constraints as Symbols 205
Show All Parameters 253
Show All Signals 241
Show edit mode tooltips 118
Show Elements 120
Show symbol and line tooltips 118
Show/hide model element details
toolbar 118
Show/Hide qualifiers 118
Show/Hide quotation marks 119
Show/Hide stereotypes 118
signal 221
addressing 278
incoming 219
outgoing 220
queue 275
signal list 223
signal receipt
symbol 274
UML definition 274
signal sending 279
symbol 278
via port or interface 279
signal sending action 278
signallist, UML 223
signature 320
metaclass 323
TTCN-3 426
simple transition 290
sort
ordering 79
Sort definitions
Model View filter 16
space
in identifier 148
special characters, in identifiers 148
standard toolbar 23
Standard View 17
start 276
state 178, 268
state expression 295
state machine 229, 267
implementation 300
inheritance 299
initialize 202
signature 323
state machine diagram 265
statement
compound 286
state-oriented view 266
static 211
status bar 25
stereotype 317

activity symbol 248
connector line 240
noScope 196
object node 253, 254
openNamespace 197
xmiImportSpecification 354
Stereotype instance compartment 206
stop, UML 288
strict
inline frame 184
subject 160
substate
indicator 270
suspension area 187
symbol 313
action 179, 285
autosize 123
behavior 241
class 200
create 179
decision 280
decision answer 282
destroy 181
edit 127
frame 313
insert 119
interface 217
junction 289
multiple selection 121
operation 211
operations .117
package 192
part 236
port 214
realized interface 219
required interface 220
return 288
save 287
signal 221
signal receipt (input) 274
signal sending 278
start 276
state 268
state machine 267
stereotype 317
stop 288
text 313
timer 223
symbol flow editing 124
Symbols with compartments 132
syntax
parse 43
syntype 228

T

TAB
Application Build 347
tab
categories 79
name 80
tabbed documents 20
tag definition 317
tagged value 316, 318
tagged values, Properties Editor 50
targa 129
target code expression 296
task. See action 285
TCL 426
template
parameters 154
TTCN-3 426
text
file 398
highlighting 43
parse 113
Text extension symbol 301
text symbol 313
tga 129
this 279
instance of class 286
this expression 294
tif 129
tiff 129
Tile Horizontally 19
Tile Vertically 19
time specification line 176
timer 223
active expression 296
event 175
timeout 175
timer reset 175
action 284
timer set 175
action 284

TNR
  error prefix . . . 347
  Name Resolution . . . 347
Toggle parameters . . . 173
toolbar . . . 23
  customize . . . 24
toolbar button
  add . . . 23
toolbars
  customize . . . 412
Toolbars, tab . . . 412
tools
  customize . . . 413
Tools, tab . . . 413
tot . . . 25
Track selection, Properties Editor . . . 51
transition . . . 271
transition line . . . 290
transition oriented
  view . . . 267
transition overriding . . . 298
TSC
  error prefix . . . 339
  Semantic Check . . . 339
TSX
  error prefix . . . 338
  errors . . . 338
  Syntax Analysis . . . 338
TTCN-3 . . . 426
TTDQuery . . . 89

U

u2, file extension . . . 111
u2x
  file extension . . . 115
UML . . . 140
  1.4, import . . . 355
  import . . . 355
  import, restrictions . . . 359
UML Comment Symbol . . . 5
UML Kind . . . 3, 8
UML Location . . . 8
UML Name . . . 8
UML Profile for Schedulability, Performance, and Time . . . 324
UML Suite
  import . . . 359
underlined name . . . 43
undo . . . 130
  shortcut . . . 402
Update Diagrams . . . 11
Update View, Properties Editor . . . 53
URN Map . . . 416
use case . . . 157
  modeling . . . 156
use case diagram . . . 156
user defined
  icons . . . 128
user interface . . . 12

V

vertical orientation . . . 247
via . . . 279
View, Properties Editor . . . 51, 53
views
  File View . . . 15
  model . . . 45
  Model View . . . 15
Views column . . . 83
Visualize in Diagram . . . 99

W

warning
  messages . . . 335
  XMI export . . . 386
What's This?, Properties Editor . . . 54
window
  auto-hide . . . 21
  Cascade . . . 19
  close . . . 20
  dock . . . 21
  expand/contract . . . 21
  layout . . . 18
  layout, Help dialog . . . 413
  Navigation . . . 405
  new . . . 20
  scroll . . . 115
  stored workspace windows . . . 22
  zoom . . . 115
windows layouts
  customize . . . 413
workspace
  Help dialog . . . 411
  Operations . . . 399

Options dialog 417
Workspace window 14
views 15

X

XMI 352
DTD 352
export 372
import 352
import, restrictions 359
version 355
xmiImportSpecification 354
XML 426

Z

zoom 115
shortcut 407