

Tutorials

RATIONAL ROSE® REALTIME

VERSION: 2003.06.00

PART NUMBER: 800-026117-000

WINDOWS/UNIX

Legal Notices

©1993-2003, Rational Software Corporation. All rights reserved.

Part Number: 800-026117-000

Version Number: 2003.06.00

This manual (the "Work") is protected under the copyright laws of the United States and/or other jurisdictions, as well as various international treaties. Any reproduction or distribution of the Work is expressly prohibited without the prior written consent of Rational Software Corporation.

Rational, Rational Software Corporation, the Rational logo, Rational Developer Network, AnalystStudio, ClearCase, ClearCase Attache, ClearCase MultiSite, ClearDDTS, ClearGuide, ClearQuest, ClearTrack, Connexis, e-Development Accelerators, DDTS, Object Testing, Object-Oriented Recording, ObjecTime, ObjecTime Design Logo, Objectory, PerformanceStudio, PureCoverage, PureDDTS, PureLink, Purify, Quantify, Rational Apex, Rational CRC, Rational Process Workbench, Rational Rose, Rational Suite, Rational Suite ContentStudio, Rational Summit, Rational Visual Test, Rational Unified Process, RUP, RequisitePro, ScriptAssure, SiteCheck, SiteLoad, SoDA, TestFactory, TestFoundation, TestStudio, TestMate, VADS, and XDE, among others, are trademarks or registered trademarks of Rational Software Corporation in the United States and/or in other countries. All other names are used for identification purposes only, and are trademarks or registered trademarks of their respective companies.

Portions covered by U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,574,898 and 5,649,200 and 5,675,802 and 5,754,760 and 5,835,701 and 6,049,666 and 6,126,329 and 6,167,534 and 6,206,584. Additional U.S. Patents and International Patents pending.

U.S. GOVERNMENT RIGHTS. All Rational software products provided to the U.S. Government are provided and licensed as commercial software, subject to the applicable license agreement. All such products provided to the U.S. Government pursuant to solicitations issued prior to December 1, 1995 are provided with "Restricted Rights" as provided for in FAR, 48 CFR 52.227-14 (JUNE 1987) or DFARS, 48 CFR 252.227-7013 (OCT 1988), as applicable.

WARRANTY DISCLAIMER. This document and its associated software may be used as stated in the underlying license agreement. Except as explicitly stated otherwise in such license agreement, and except to the extent prohibited or limited by law from jurisdiction to jurisdiction, Rational Software Corporation expressly disclaims all other warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability, non-infringement, title or fitness for a particular purpose or arising

from a course of dealing, usage or trade practice, and any warranty against interference with Licensee's quiet enjoyment of the product.

Third Party Notices, Code, Licenses, and Acknowledgements

Portions Copyright ©1992-1999, Summit Software Company. All rights reserved.

Microsoft, the Microsoft logo, Active Accessibility, Active Client, Active Desktop, Active Directory, ActiveMovie, Active Platform, ActiveStore, ActiveSync, ActiveX, Ask Maxwell, Authenticode, AutoSum, BackOffice, the BackOffice logo, bCentral, BizTalk, Bookshelf, ClearType, CodeView, DataTips, Developer Studio, Direct3D, DirectAnimation, DirectDraw, DirectInput, DirectX, DirectXJ, DoubleSpace, DriveSpace, FrontPage, Funstone, Genuine Microsoft Products logo, IntelliEye, the IntelliEye logo, IntelliMirror, IntelliSense, J/Direct, JScript, LineShare, Liquid Motion, Mapbase, MapManager, MapPoint, MapVision, Microsoft Agent logo, the Microsoft eMBEDded Visual Tools logo, the Microsoft Internet Explorer logo, the Microsoft Office Compatible logo, Microsoft Press, the Microsoft Press logo, Microsoft QuickBasic, MS-DOS, MSDN, NetMeeting, NetShow, the Office logo, Outlook, PhotoDraw, PivotChart, PivotTable, PowerPoint, QuickAssembler, QuickShelf, RelayOne, Rushmore, SharePoint, SourceSafe, TipWizard, V-Chat, VideoFlash, Visual Basic, the Visual Basic logo, Visual C++, Visual C#, Visual FoxPro, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, the Visual Studio logo, Vizact, WebBot, WebPIP, Win32, Win32s, Win64, Windows, the Windows CE logo, the Windows logo, Windows NT, the Windows Start logo, and XENIX, are either trademarks or registered trademarks of Microsoft Corporation in the United States and/or in other countries.

Sun, Sun Microsystems, the Sun Logo, Ultra, AnswerBook 2, medialib, OpenBoot, Solaris, Java, Java 3D, ShowMe TV, SunForum, SunVTS, SunFDDI, StarOffice, and SunPCi, among others, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Purify is licensed under Sun Microsystems, Inc., U.S. Patent No. 5,404,499.

Licensee shall not incorporate any GLOBEtrotter software (FLEXIm libraries and utilities) into any product or application the primary purpose of which is software license management.

BasicScript is a registered trademark of Summit Software, Inc.

Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Copyright © 1995 by Addison-Wesley Publishing Company, Inc. All rights reserved.

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

Contents

- Preface xi**
- Audience. xi
- Other Resources xi
- Rational Rose RealTime Integrations With Other Rational Productsxii
- Contacting Rational Customer Support xiii
- 1 Rational Rose RealTime Tutorials15**
- Overview. 15
- Navigating the Tutorials 17
- Printing the Tutorials. 17
- 2 QuickStart Tutorial19**
- Getting Started 19
- Rational Rose RealTime User Interface 20
- Online Help 22
- Sample Model. 22
- Model Description. 23
- Creating a New Model 24
- Creating the Logical View. 25
- Creating a Capsule. 25
- Adding a State to a Capsule. 26
- Drawing an Initial Transition 29
- Adding a Port to a Capsule. 30
- Saving a Model. 34
- Adding the Detail Code to a State Machine 34
- Creating the Component View 36
- Creating a Component 36
- Building the Component 43
- Creating the Deployment View 45
- Creating a Component Instance. 45
- Running the Component Instance 47

Tutorial Summary	48
Viewing the Generated Code	49
What's Next?	49
3 Card Game Tutorial.	51
What You Will Learn?	51
Why a Card Game?	52
Card Game Requirements	52
Before You Begin	53
Tutorial Lessons	53
Adding Detail Code to Operations	54
Build Information.	54
Several Ways of Doing the Same Thing	54
Lesson 1: Creating a New Model and Configuring the Toolset	55
Opening a New Model	55
Configuring Toolset Options	56
Lesson 2: Creating a Use Case and Initial Capsules.	59
Adding the Use Case	60
Documentation Window	67
Are Elements Owned by the Class Diagram?	67
Meaning of the Delete Dey in Class Diagrams	67
Defining the Classes.	67
Classes Versus Capsules	68
Describing the Behavior of the Classes	68
Creating Classes and Capsules	69
RTClasses Package	74
Changing Element Types	74
Creating the HeadsUpPoker Capsule Structure	75
Creating the HeadsUpPoker Capsule	75
Lesson 3: Sequence Diagrams, Protocols, Ports, and Connectors	81
Creating the Protocol	94
Creating Ports and Connectors	100
Documenting the Responsibilities	105
Lesson 4: Building and Running	106
Prototyping	106
Building a Model	107
Creating a Component	108

Creating the Deployment View	114
Starting the Build	115
Where is the Source Code Generated?	118
Running the Component Instance	118
Review	121
Lesson 5: Adding Behavior to the Capsules	122
Opening Capsule State Diagrams	122
Creating the Dealer's Behavior	123
Creating the Player's Behavior	133
Creating the State Diagram	134
Adding Attributes	140
Creating the Actions	142
Review	146
Lesson 6: Navigating and Searching	146
Suggested Reading	147
Lesson 7: Using Traces and Watches to Debug the Design	147
Rebuilding the Model	147
Setting Up the Runtime Windows	148
Problems with the Player Capsule	155
Unexpected Messages	155
Warning Message for No Defined Trigger?	156
Building the Player Capsule	156
Debugging the Player Capsule	157
Verifying the Fix	164
Review	165
Lesson 8: Class Modeling	165
Importing Classes	166
Creating a Package	167
Creating the Initial Class Structure	168
Creating Relationships Between Classes	169
Adding Attributes to the Card Class	175
Adding Details to the CardList Class	176
Generating Code for the Association Ends	186
Encoding and Decoding by the Services Library	186
Encoding and Decoding	187
Adding Details to the Deck Class	188

Adding Details to the Hand Class	192
Adding Details to the PokerHand Class	195
Review	197
Lesson 9: Adding Card Classes to the Capsule Behavior	198
Completing the Dealer Capsule Behavior	198
Adding a Destructor to the Dealer Capsule	203
Completing the Player Capsule Behavior	204
Using Attributes Versus Aggregations	205
Adding Dependencies	205
Dependency Properties	213
Adding Inclusions	213
Building and Running the Card Game	215
Trace Summary	218
Fixing Compilation Errors	218
Lesson 10: Aggregating in a State Diagram	219
Aggregating the Receiving Behavior	219
Tutorial Summary	221
4 Rational Rose RealTime Extensibility Interface Tutorials	223
RRTEI Tutorial Overview	223
Previewing the Tutorials	224
Creating a Summit Basic Script	225
Writing a Script	225
Running and Testing a Script	227
Compiling a Script	227
Creating a Menu File	227
Adding Entries to the Registry	228
Running and Testing the Script From the Menu	229
Creating a Visual Basic Add-in	229
Creating the ActiveX DLL	230
Creating the Add-in Menu File	233
Adding Entries to the Registry	234
Testing the New Add-in	235
Common Problems	235
Creating an Add-in Which Extends the Context Menus	236
How Context Menus Work	236
Menus Associated with Default or Specific Elements	237

Creating the ActiveX DLL	238
Adding Entries to the Registry	242
Testing the New Add-in	243
5 Concept Tutorials	245
Overview	245
Messages and Capsule State Machines	246
Capsule Hierarchical State Machines	247
Capsules and Capsule Roles	247
Ports, Protocols, and Protocol Roles	247
Index	249

Preface

This manual provides introduction and concept tutorials to help you become familiar with Rational Rose RealTime.

This manual is organized as follows:

- *Rational Rose RealTime Tutorials* on page 15
- *QuickStart Tutorial* on page 19
- *Card Game Tutorial* on page 51
- *Rational Rose RealTime Extensibility Interface Tutorials* on page 223
- *Concept Tutorials* on page 245

Audience

This guide is intended for all readers including managers, project leaders, analysts, developers, and testers.

This guide is specifically designed for software development professionals familiar with the target environment they intend to port to.

Other Resources

- Online Help is available for Rational Rose RealTime.

Select an option from the **Help** menu.

All manuals are available online, either in HTML or PDF format. To access the online manuals, click **Rational Rose RealTime Documentation** from the **Start** menu.

- To send feedback about documentation for Rational products, please send e-mail to techpubs@rational.com.
- For more information about Rational Software technical publications, see: <http://www.rational.com/documentation>.

- For more information on training opportunities, see the Rational University Web site: <http://www.rational.com/university>.
- For articles, discussion forums, and Web-based training courses on developing software with Rational Suite products, join the Rational Developer Network by selecting **Start > Programs > Rational Suite > Logon to the Rational Developer Network**.

Rational Rose RealTime Integrations With Other Rational Products

Integration	Description	Where it is Documented
Rose RealTime–ClearCase	You can archive Rose RT components in ClearCase.	<ul style="list-style-type: none"> ▪ <i>Toolset Guide: Rational Rose RealTime</i> ▪ <i>Guide to Team Development: Rational Rose RealTime</i>
Rose RealTime–UCM	Rose RealTime developers can create baselines of Rose RT projects in UCM and create Rose RealTime projects from baselines.	<ul style="list-style-type: none"> ▪ <i>Toolset Guide: Rational Rose RealTime</i> ▪ <i>Guide to Team Development: Rational Rose RealTime</i>
Rose RealTime–Purify	When linking or running a Rose RealTime model with Purify installed on the system, developers can invoke the Purify executable using the Build > Run with Purify command. While the model executes and when it completes, the integration displays a report in a Purify Tab in RoseRealTime.	<ul style="list-style-type: none"> ▪ Rational Rose RealTime Help ▪ <i>Toolset Guide: Rational Rose RealTime</i> ▪ <i>Installation Guide: Rational Rose RealTime</i>
Rose RealTime–RequisitePro	You can associate RequisitePro requirements and documents with Rose RealTime elements.	<ul style="list-style-type: none"> ▪ <i>Addins, Tools, and Wizards Reference: Rational Rose RealTime</i> ▪ <i>Using RequisitePro</i> ▪ <i>Installation Guide: Rational Rose RealTime</i>
Rose RealTime–SoDa	You can create reports that extract information from a Rose RealTime model.	<ul style="list-style-type: none"> ▪ <i>Installation Guide: Rational Rose RealTime</i> ▪ <i>Rational SoDA User's Guide</i> ▪ SoDA Help

Contacting Rational Customer Support

If you have questions about installing, using, or maintaining this product, contact Rational Customer Support.

Your Location	Telephone	Facsimile	E-mail
North America	(800) 433-5444 (toll free) (408) 863-4000 Cupertino, CA	(781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 (0) 20-4546-200 Netherlands	+31 (0) 20-4546-201 Netherlands	support@europe.rational.com
Asia Pacific	+61-2-9419-0111 Australia	+61-2-9419-0123 Australia	support@apac.rational.com

Note: When you contact Rational Customer Support, please be prepared to supply the following information:

- Your name, company name, telephone number, and e-mail address
- Your operating system, version number, and any service packs or patches you have applied
- Product name and release number
- Your Service Request number (SR#) if you are following up on a previously reported problem

When sending email concerning a previously-reported problem, please include in the subject field: "[SR#XXXXX]", where XXXXX is the Service Request number of the issue. For example, "[SR#0176528] - New data on rational rose realtime install issue ".

Rational Rose RealTime Tutorials

1

Contents

This chapter is organized as follows:

- *Overview* on page 15
- *Navigating the Tutorials* on page 17
- *Printing the Tutorials* on page 17

Overview

Rational Rose RealTime provides tutorials to help you learn how to use the main features of the development tool. There are two types of tutorials: *hands-on* and *concept overviews*.

The hands-on tutorials show you how to build models, while demonstrating key concepts and toolset features required when developing your own Rational Rose RealTime models. Both the *Quickstart* and *Card Game* tutorials provide sample models that you can use to review the procedures and concepts introduced in each tutorial. These models are located in the Rational Rose RealTime installation directory \$ROSERT_HOME/Tutorials.

Concept tutorials are meant to provide an introduction to important Rational Rose RealTime concepts. They expand and summarize the explanations and examples provided in the *Rational Rose RealTime Modeling Language Guide*.

If you do not know where to begin, the following table may help you find the tutorial recommended for your individual modeling experience.

Tutorial	New Modeling Tool User	Rose 98 User	ObjecTime Developer User
QuickStart Tutorial (hands-on)	X	X	
Contents (hands-on)	X	X	X
Rational Rose RealTime Extensibility Interface Tutorials (RRTEI Tutorial Overview -hands-on)	X	X	X

Tutorial	New Modeling Tool User	Rose 98 User	ObjecTime Developer User
Messages and Capsule State Machines <i>Tutorial</i> (concept overview)	X	X	
Capsule Hierarchical State Machines <i>Tutorial</i> (concept overview)	X	X	
Capsules and Capsule Roles <i>Tutorial</i> (concept overview)	X	X	X
Ports, Protocols, and Protocol Roles <i>Tutorial</i> (concept overview)	X	X	X

QuickStart: Create a simple "Hello World" model. This is the quickest way to get started without having to read extensively.

Card Game: Learn the most important features of the tool by designing, and developing a fun application in C++.

Rational Rose RealTime Extensibility Interface: Learn how to write a simple script in Summit Basic and Visual Basic that will control the Rational Rose RealTime application. Learn the basics of creating add-ins.

Messages and Capsule State Machines: For users who want an introduction to the basics of message passing between capsules.

Capsule Hierarchical State Machines: For users who want to review the basic elements of state machines and understand some of the complexities involved with hierarchical capsule state machines.

Capsules and Capsule Roles: For users who already understand class modeling, and want to understand the additional concepts involved when modeling with capsules.

Ports, Protocols, and Protocol Roles: For users who want an introduction to the use of protocols and protocol roles in a Rational Rose RealTime model.

Navigating the Tutorials

Depending on the whether you wish to complete the tutorials sequentially or jump to specific lessons, you can navigate the tutorials in two ways:

- You can view a tutorial in order without having to scroll through the Table of Contents using the **Next** and **Previous** buttons at the top and bottom of each Help window.
- You can jump to a particular topic by directly selecting it from the **Contents** window in the **Help** browser.

Printing the Tutorials

If you prefer working from a printed copy of the tutorial, you can find a PDF version of the tutorial chapters located in the Rational Rose RealTime installation directory `$ROSERT_HOME/Help/rosert_tutorials_guide.pdf`.

Contents

This chapter is organized as follows:

- *Getting Started* on page 19.
- *Model Description* on page 23.
- *Creating a New Model* on page 24.
- *Creating the Logical View* on page 25.
- *Creating the Component View* on page 36.
- *Building the Component* on page 43.
- *Creating the Deployment View* on page 45.
- *Running the Component Instance* on page 47.
- *Tutorial Summary* on page 48.

Getting Started

As a new Rational Rose RealTime user, this *QuickStart* tutorial gets you up-and-running in Rational Rose RealTime as quickly as possible. Using a simple example, the tutorial guides you through the basic steps involved in constructing a model in Rational Rose RealTime using the C++ language add-in.

Note: Although this tutorial is based on the C++ language, Rational Rose RealTime also supports the Java and C languages.

In this tutorial, you learn how to:

- Construct elements in the **Logical View**, including:
 - capsules
 - capsule state diagrams
 - ports
- Build and execute a model in the Rational Rose RealTime execution environment.

Note: Ensure that your environment is properly configured for your compiler. For additional information about configuring your environment, see *Installation Guide Rational Rose RealTime*.

The Rational Rose RealTime toolset provides a complete development environment for using UML to create executable models.

The UML provides a wide range of visual modeling constructs. Not all of these constructs are directly applicable to creating a working model. Many exist for purposes of building more complex systems, communicating designs to other team members, capturing design decisions, and for organizing models, but are not strictly required to build a simple model.

Rational Rose RealTime provides additional constructs that are based on UML modeling elements and are specialized for creating executable real-time models. In order to produce a working model, it is important to understand the model elements that must be defined, and the sequence of the elements.

Because this tutorial is geared toward the new user, the focus is only on those elements and tools that are required to create a basic executable model. If you want to construct a more complicated Rational Rose RealTime model, you should familiarize yourself with some of the advanced elements that are outside the scope of this tutorial. For example: use cases and use case diagrams, actors, packages, sequence diagrams, collaboration diagrams, and so on.

Note: To complete this tutorial, you need to have Rational Rose RealTime and a compiler (such as, Visual C++ 6.0) installed on your computer.

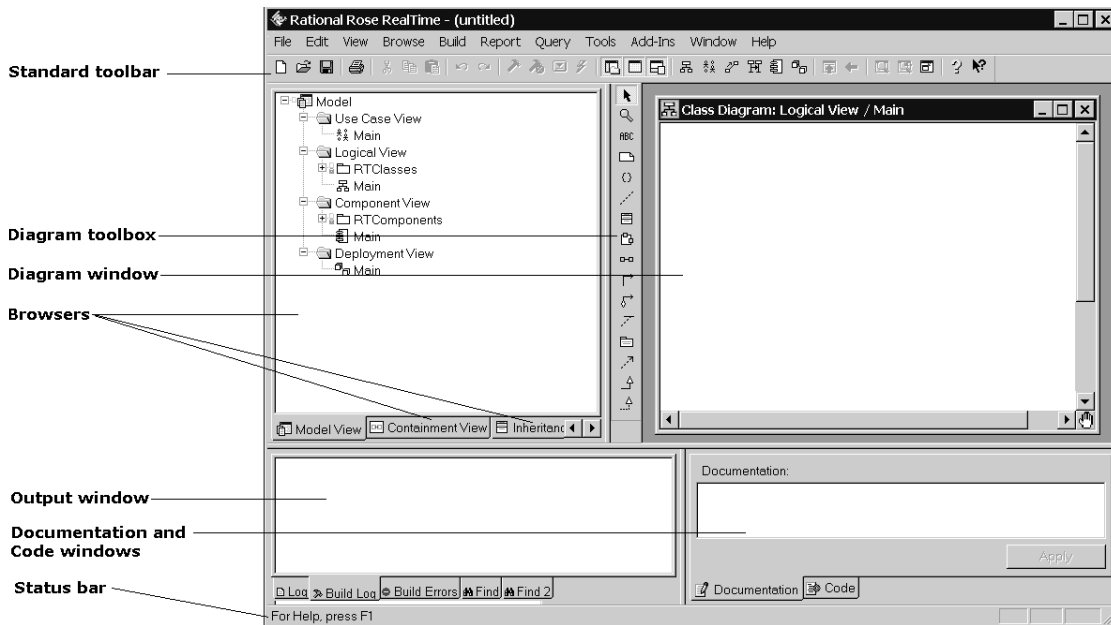
Rational Rose RealTime User Interface

Before proceeding with the tutorial, you need to become familiar with the main features of the Rational Rose Realtime user interface.

Note: When you initially open Rational Rose RealTime, a default set of windows appear (as illustrated below). However, you can customize the user interface to suit your modeling needs. For example, you can change where the windows are docked, or set them to free-floating. You can add or remove buttons from the toolbar, and you can show or hide the browsers.

Figure 1 shows a typical display of the Rational Rose RealTime Graphical User Interface.

Figure 1 Rational Rose RealTime Graphical User Interface



The main features of the Rational Rose RealTime user interface are:

- The **Standard Toolbar** remains the same for all views and diagrams. It contains standard Windows functions as well as those specific to Rational Rose RealTime.
- The **Diagram Toolbox** is used for adding elements to the model by drawing them on a diagram. The toolbox elements change depending on the active diagram. For example, the **Use-Case Diagram** has a tool for adding actors, but the **Component Diagram** does not have this tool.
- Browsers are hierarchical and can be expanded and contracted. When you start Rational Rose RealTime, the **Model View**, the **Containment View**, and the **Inheritance View** browsers are docked on the left side of the interface in a stacked format. They can be set to visible/invisible, docked, or floating. To activate a specific browser, select the appropriate tab located at the bottom of the interface.

There are two additional browsers, also referred to as editors, that can be opened to perform specific tasks: the **Structure/State Diagram Browser/Editor**, and the **Run Time System (RTS) Browser/Editor**. These browsers cannot be moved.

- Rational Rose RealTime offers four main views located on the **Model View** browser. Each view is related to a software lifecycle phase, and the diagrams are artifacts of those phases.

- **Use-Case View** shows what a system (subsystem, class, or interface) does but does not specify how the system internally performs its tasks.
- **Logical View** represents the architectural processes as the model moves from analysis, through design, and into development.
- **Component View** contains concrete representations of the system. Components realize the active and data classes, and provide the components for building an executable model.
- **Deployment View** shows how the system is to be distributed. It defines the processors, and contains a diagram of the nodes in the system.

Online Help

For more information about the Rational Rose RealTime user interface, or how to complete a specific task, see the Rational Rose RealTime online Help.

To open the online Help:

- On the **Help** menu, click **Contents**.

The Table of Contents for the Rational Rose RealTime Online Help appears.

You can select topics from the Table of Contents or search through the index for keywords. The Help is content-sensitive, and automatically opens a **Help** window related to the tool or diagram you use.

Sample Model

In addition to the online Help, Rational Rose RealTime includes a sample model, QuickstartTutorial.rtm dl. You can use this model to explore the various elements of Rational Rose RealTime, and to practice the procedures for model building and execution.

To open the sample model:

- 1 On the **File** menu, click **Open**.
- 2 In the **Open** dialog box, locate the Rational Rose RealTime installation directory, and select `Tutorials/gstarted/QuickstartTutorial.rtm dl`.

3 Click **Open**.

Note: If prompted with "Do you wish to create the default workspace?", click **No**. A workspace maintains information about the current model, open windows and window positions, etc. The workspace information is stored in a separate file (a .rtwks file).

4 Click **File > New** to create a new model.

Model Description

In this tutorial, your goal is to create a new model with one capsule that prints "Hello world!" to the console.

This is a very simple model that could easily be accomplished in a native development environment by writing the code in the C++ programming language (see below), and then compiling the code into an executable file.

```
#include <iostream.h>

main() {

cout << "Hello world!\n";
```

However, you will generate the equivalent executable from within the Rational Rose RealTime modeling environment by following a basic workflow consisting of these simple tasks:

- 1 Creating a new model.
- 2 Creating the **Logical View**.
- 3 Creating the **Component View**.
- 4 Building the component.
- 5 Creating the **Deployment View**.
- 6 Running the component instance.

Creating a New Model

When you start Rational Rose RealTime, the **Create New Model** dialog box appears.

Typically, there are seven frameworks listed: **Empty**, **RTC**, **RTC++**, **RTJava**, **StartupC**, **StartupCPP**, **StartupJ**. However, you may have additional optional frameworks, such as the **Gateway** framework and other frameworks you create.

To open a model containing all the classes required for development using the C, C++, or Java language, click the framework for the specified language. The **Model View** browser appears with the packages and classes populated in the **Logical View** and **Component View**.

Note: Set the language and environment used in your model defaults to the same language that you specified in the framework. You can change the language or environment settings in the **Language/Environment** tab (**Options** dialog box, **Tools** menu).

Note: The **Empty** framework is useful for creating use case designs but should not be used for developing RealTime applications.

If you want to open Rational Rose RealTime without the **Create New Model** dialog box automatically appearing, clear **Always show this dialog on startup**.

In this tutorial, you will create a model using the C++ framework.

To create a model and specify the default language:

- 1 Start Rational Rose RealTime.

The **Create New Model Dialog** appears. If it does not appear, on the **File** menu, click **New**.

- 2 Double-click **RTC++**.

You are now ready to build and execute a Rational Rose RealTime model.

Creating the Logical View

Creating the **Logical View** involves discovering and creating the various classes that make up the design solution for the problem (that is, how to create a model that prints "Hello world!").

Usually, you define several classes, capsules, and protocols to create a working model. You iterate between defining the model elements, compiling and running the model to see how it works, and then fixing problems and add more functionality to the model. However, for this basic *QuickStart* tutorial, you will create only one capsule to print the "Hello world!" message to the console.

The steps involved in creating the **Logical View** are:

- Creating a capsule.
- Adding a state to a capsule.
- Drawing an initial transition.
- Adding a port to a capsule (not required when using the C language).
- Adding the detail code to a state machine.

Creating a Capsule

Capsules are special forms of classes with some supporting mechanisms that enforce a higher degree of encapsulation and abstraction than classes. Capsules are very useful for building concurrent, event-driven systems, and are an important element in Rational Rose RealTime.

The code required to implement this example could easily be defined in an operation on a regular class, but something needs to initiate the activity. Operations do not run by themselves. They must be invoked.

In Rational Rose RealTime, you place the code in a capsule that is automatically invoked by the `main()` program in the Rational Rose RealTime Services Library. The `main()` block in the Services Library creates the capsule in your model, and starts the state machine. You describe the capsule and define the state machine, and they are automatically created and executed by the Services Library.


Note: A model must include at least a top-level capsule that initiates behavior in the system, and results in generated code which forms the executable. In this case, your model will contain only one top-level capsule that prints "Hello world!" from within the body of its state machine.

You will create a capsule class in the **Logical View** of the **Model View** browser, and name it **HelloWorld**. This simple capsule class implements the design model.

To create a new capsule class:

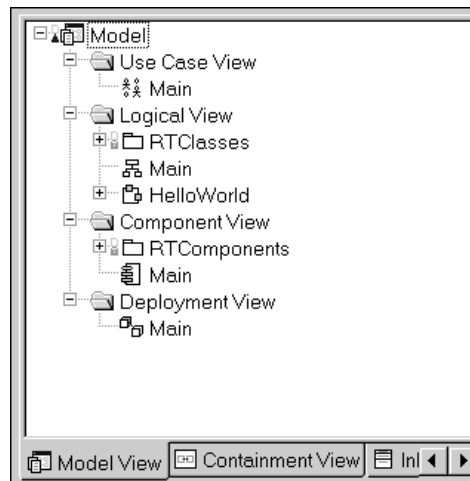
- 1 In the **Model View** browser, right-click **Logical View**, and select **New > Capsule**.

Or . . .

In the toolbox for the **Class Diagram**, select the **Capsule** button , and click on the **Class Diagram: Logical View / Main** window.

- 2 Rename the capsule **HelloWorld** and press **ENTER**.

On the **Model View** tab in the browser, the new capsule **HelloWorld** appears in the **Logical View** folder.



Adding a State to a Capsule

For capsule classes, a state diagram results in a complete code implementation generated for the class. The state diagram defines the majority of a capsule class implementation. The capsule class may also have operations defined on it, but the state diagram gives the capsule its asynchronous message processing capability.

In this tutorial, you will open the **State Diagram** for the **HelloWorld** capsule, and add a state to that capsule.

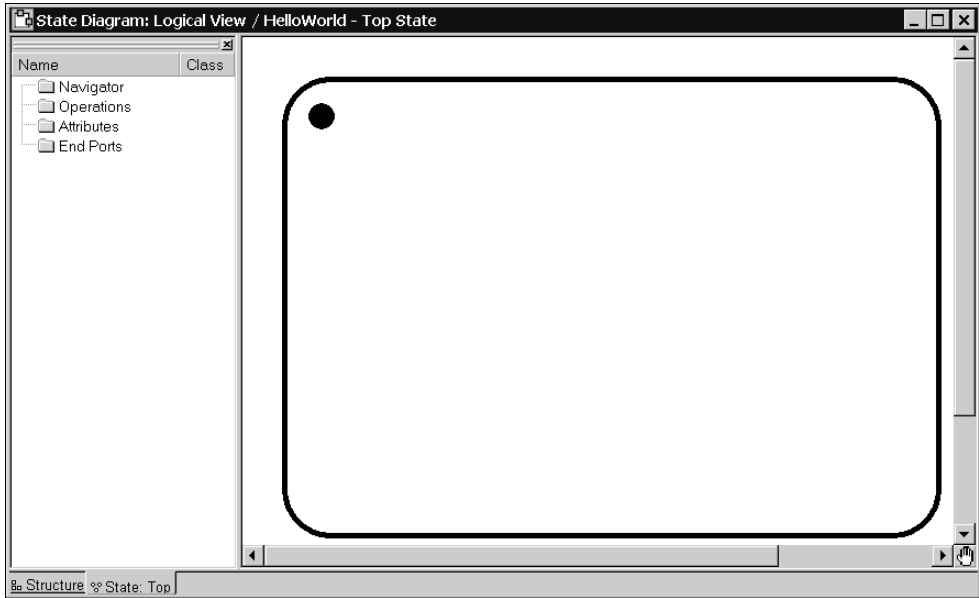
To add a state to a capsule:


- 1 In the **Model View** browser, right-click the **HelloWorld** capsule, and click **Open State Diagram**.

Or . . .

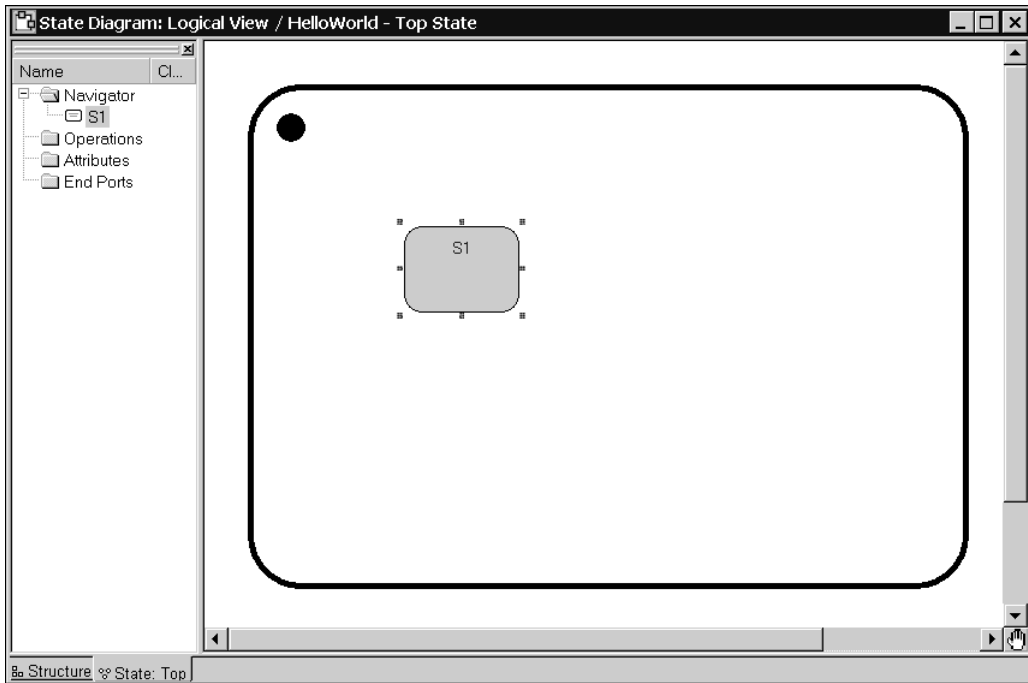
In the **Class Diagram**, right-click on the **HelloWorld** capsule and click **Open State Diagram**.

The **State Diagram** appears.

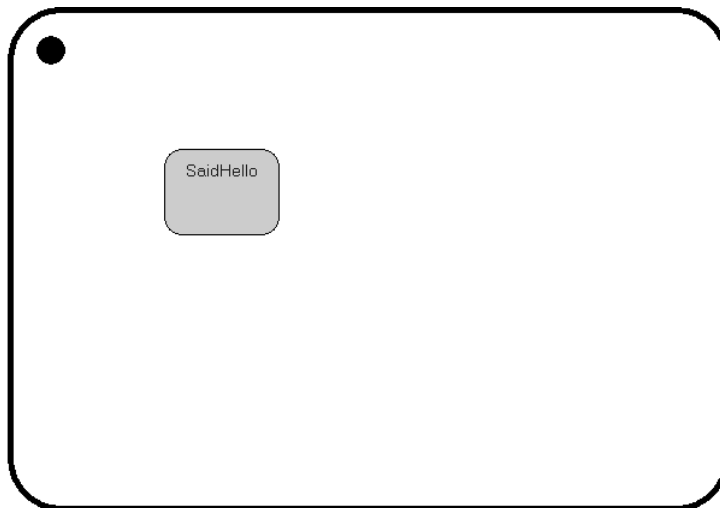



- 2 In the **Diagram** toolbox, click **State** .
- 3 Move the cross hairs into the rounded rectangle in the **State Diagram**, and click to create a new state.

A state appears with the default name **S1**.



- 4 Rename the state **SaidHello** and press ENTER.



The state diagram contains an *initial point*, , and an *initial state* called **SaidHello**. An initial point is a special point which explicitly shows the beginning of the state machine. You connect the initial point to a start state (in this case, SaidHello). Where the start state will be the first active state in the objects state machine. The transition from the initial point to the start state, the initial transition, is the first transition taken before any other transition. Only one initial state is allowed in each state diagram.



Only one outgoing transition can exist from the initial point.

There can be several incoming transitions to the initial state. In this case the initial state acts like a junction point which forces the behavior back through the initial transition. If the initial transition is used to completely initialize an object, then any incoming transition to the initial state will effectively reset the behavior of an object without having to destroy then re-create it.

Drawing an Initial Transition

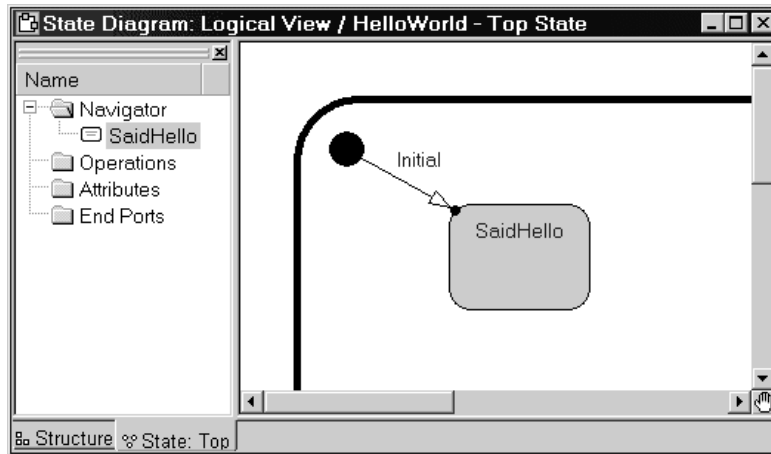
A transition is a relationship between two states, a source state and a destination state. It specifies that when an object in the source state receives a specified event and certain conditions are met, the behavior moves from the source state to the destination state. You create an **Initial Transition** that is automatically invoked at runtime when a capsule instance is created. Any action code associated with the **Initial Transition** runs when the capsule instance is created.

To draw a transition:

- 1 In the toolbox for the **State Diagram** dialog box, click **State Transition** .
 - 2 Click and hold the left mouse button on the **Initial Point** in the state diagram, .
- The **Initial Point** is the black circle that appears in the top-left corner of the **State Diagram**.

- 3 Drag the **Transition** line to the top of the **SaidHello** state.

Note: The **Initial Transition** has a default name of **Initial**.



Now that you created an initial transition - a transition from the initial point to the initial state - you specified that at runtime, the SaidHello state is the first state to receive a specified event and the behavior moves from the initial point to the SaidHello state.

Next, you will create a port to communicate with the HelloWorld capsule instance by sending and receiving messages.

Adding a Port to a Capsule

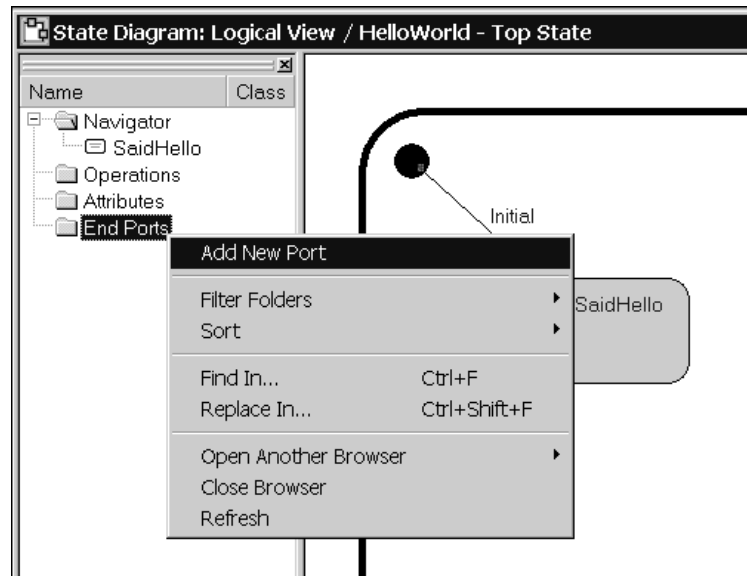
Ports are objects that communicate with capsule instances by sending and receiving messages. A port is owned by the capsule instance, and is created/destroyed with the capsule. Each port has a separate identity that is distinct from the identity and state of the associated capsule instance.

By default, new ports are public. Public ports appear on a capsule's boundary in the **Structure Diagram**, and are visible both from outside and inside the capsule. In this model, you will make a protected port. Protected ports cannot be accessed outside the capsule.

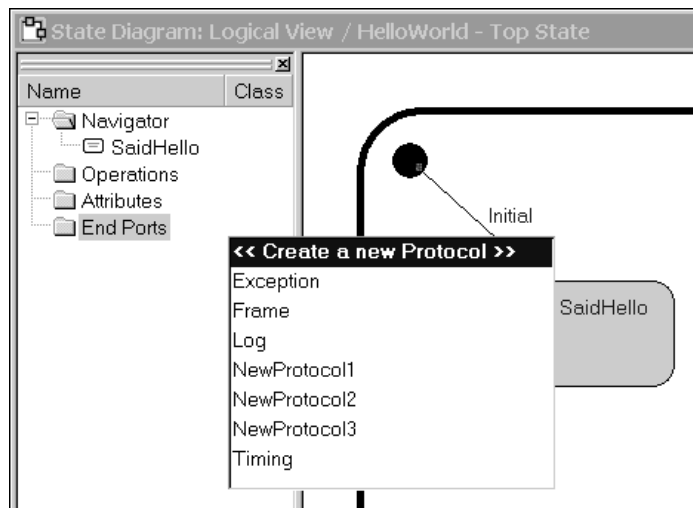
You will add a new port to the "Hello world" capsule using the built-in Log protocol.

To add a protected port to a capsule:

- 1 In the browser to the left of the **HelloWorld Top State diagram**, right-click **End Ports**.

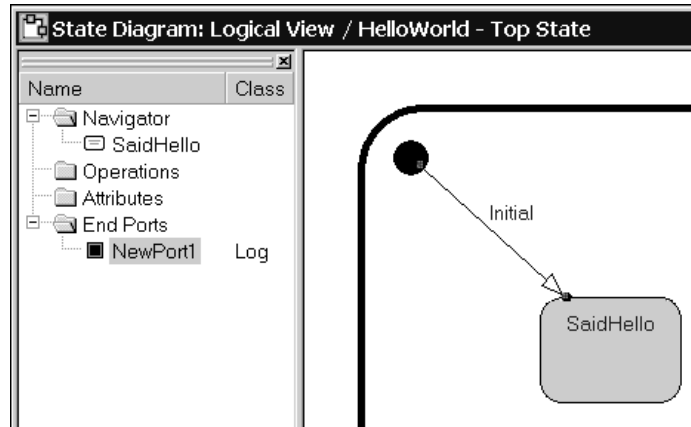


- 2 Click **Add New Port**.

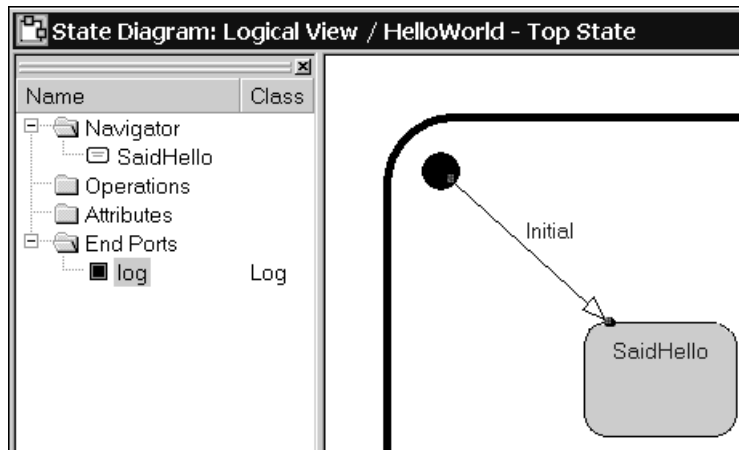


- 3 From the list of protocol classes available for the model, double-click to select **Log**.

A port appears with the default name **NewPort1**.

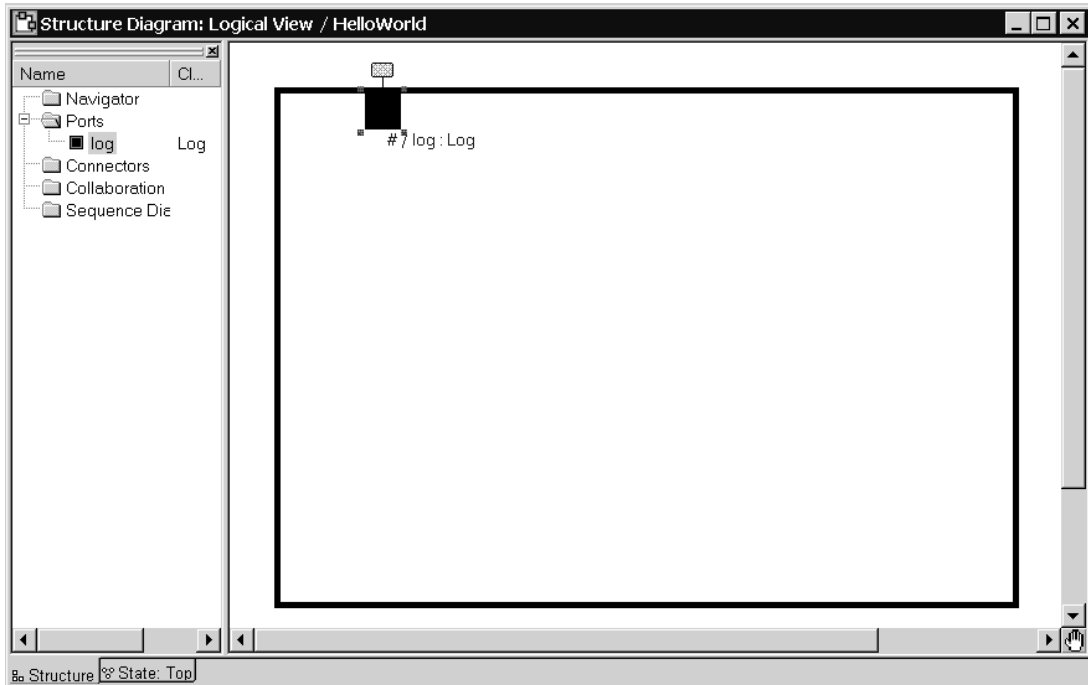



- 4 Rename the port name to **log** and press ENTER.



- 5 Click the **Structure** tab for the **HelloWorld** capsule.

The **Structure Diagram** appears as the active diagram.



6 In the **Structure Diagram**, click the **log** port, .

To send and receive messages, the capsule must have end ports. The end port's protocol defines the set of messages that can be sent and received. Protected ports are not public. This means that these ports are not visible from the outside of a capsule since they are not part of the capsule's interface.

Saving a Model

Before you continue with the tutorial, we recommend that you save your model.

To save a model:

- 1 On the **File** menu, click **Save Model As...**
- 2 In the **File Name** box, type **HelloWorld**.
- 3 In the **Save In** box, select the folder where you want to save the model.
- 4 Click **Save**.
- 5 If prompted to create a default workspace with a HelloWorld.rtwks path name, click **Yes**.

Note: When you create a workspace, all components, specifications and windows in your model are saved as they exist in your model. You can also save a model without saving these specifications by using the file extension `.rtmdl`.

The sample model showing the completed procedures to this point in the tutorial is located in the Rose RealTime installation directory:

```
$ROBERT_HOME/Tutorials/gstarted/QuickstartTutorial.rtmdl
```

Adding the Detail Code to a State Machine

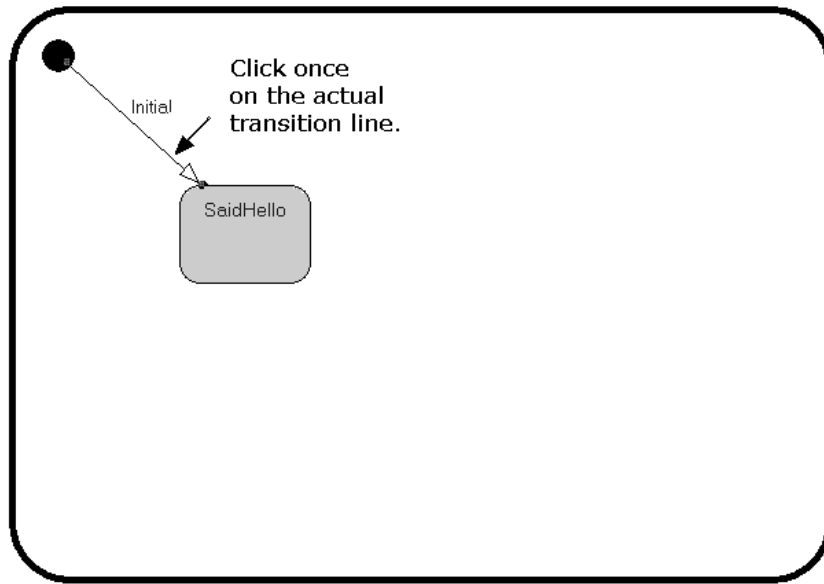
You have all the required elements in place (that is, initial state, initial transition, and log port), and you will now add detail code to the **Initial Transition**. The detail code will be executed when the **Initial Transition** is run at model execution time.

You need to add the C++ code to the state machine to implement the desired behavior. C++ code can be added as actions on transitions, choice points, and state entry or exit on capsule state diagrams that are executed at runtime. Only code added to a capsule state diagram is included in the generated code for the model. Detailed actions on protocol or data class state diagrams are not included in the generated code for those classes.

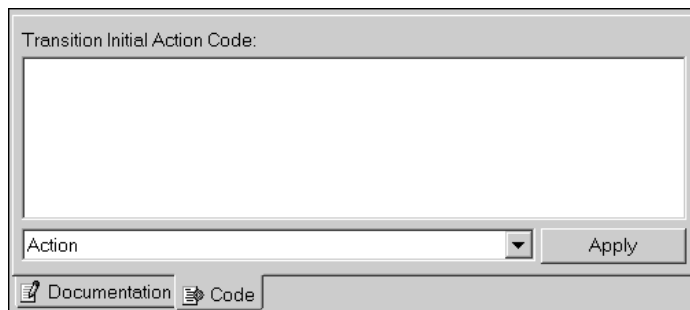
In our model, the detail code will use the Log service to write the "Hello world!" message to the console.

To add code to the Initial Transition:

- 1 Click the **State:Top** tab for the **HelloWorld** capsule
- 2 Click the line for the **Initial Transition**.



Next, you will use the convenience of the **Code** window to add code to the transition. The **Code** window is located in the lower right-hand corner of the Rational Rose RealTime window. For example, the **Code** window for the **Initial** transition is:



Note: If you prefer, you can also double-click on the **Initial** transition line to open the **Transition Specification** dialog.

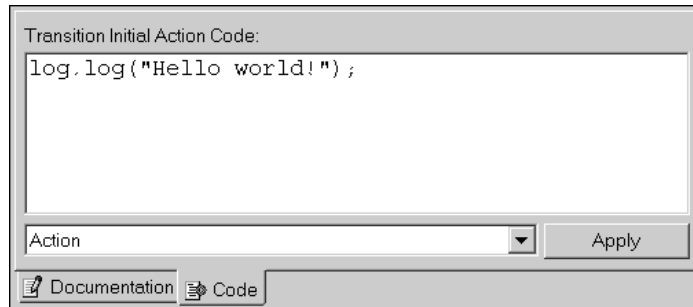
- 3 Click the **Code** tab
- 4 Select **Action** from the drop-down list.

- 5 Type the following C++ code:

```
log.log("Hello world!");
```

Ensure that you include the semi-colon at the end of the line.

Your **Code** window will look like the following:



- 6 To save the changes, click **Apply**.

Creating the Component View

The **Component View** specifies how to compile various parts of the model. The primary element of the **Component View** is a component that you need to create. This component specifies the capsules and classes to compile, how to compile those elements, and the inclusions and libraries to incorporate into the build.

You must create a component for the top-level capsule in order to build and execute your model. You can draw component diagrams for situations where you have many related components or packages of components.

Creating a Component



You will create a component by opening a series of dialog boxes so that you can become familiar with the common features of the Rational Rose RealTime user interface. Step 1 provides with two ways of creating a component. Choose only one method.

Note: You can also use the **Component Wizard** to create a component. The wizard guides you through creating and configuring a component, and running the component instance. To access the wizard, click **Component Wizard** on the **Build** menu.

To create a component:

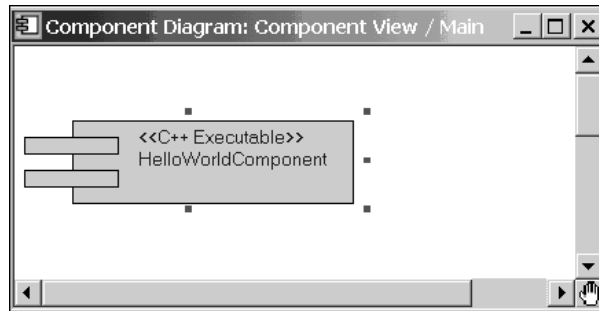
- 1 In the **Model View** tab in the browser, select **Component View**, and right-click **New > Component**.

Or . . .

In the **Toolbar**, select the **Browse Component Diagram** button . For the **Component View** package, select **Component Diagram: Component View / Main** in the **Component diagrams** list, and click **OK**. From the **Toolbox**, select the **Component** tool , then click in the diagram.

- 2 Rename the component **HelloWorldComponent**.

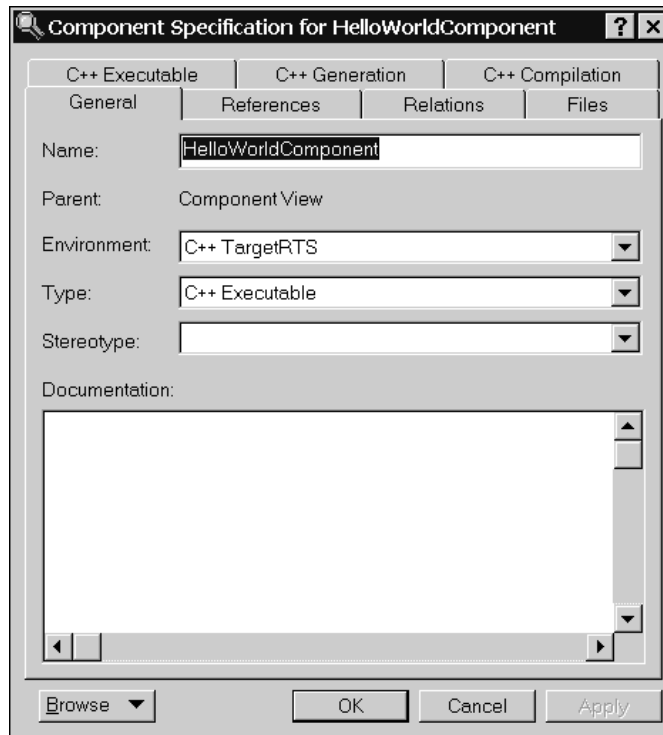
If you used the second method, you will have a visual representation of your component in the **Component Diagram** for **Main**.



The **Model View** tab in the browser is updated to include the new component.

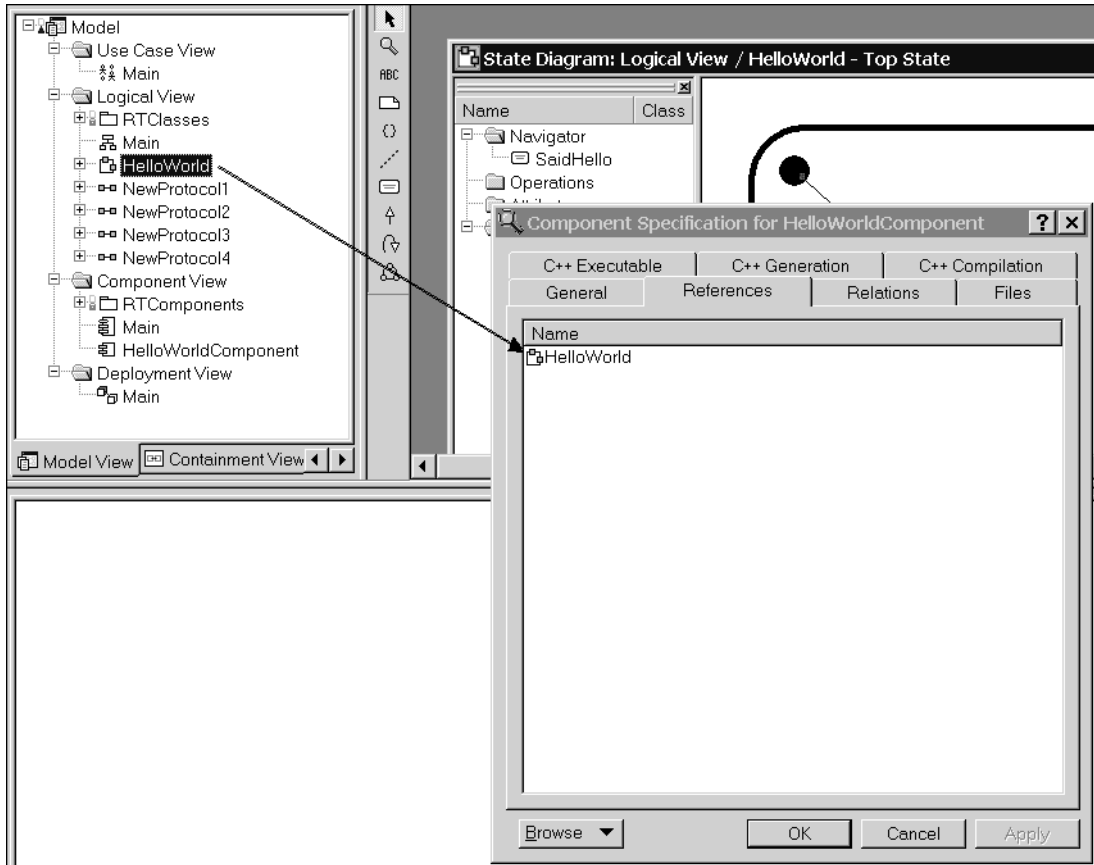


- 3 Double-click **HelloWorldComponent** to open the **Component Specification for HelloWorldComponent** dialog box.



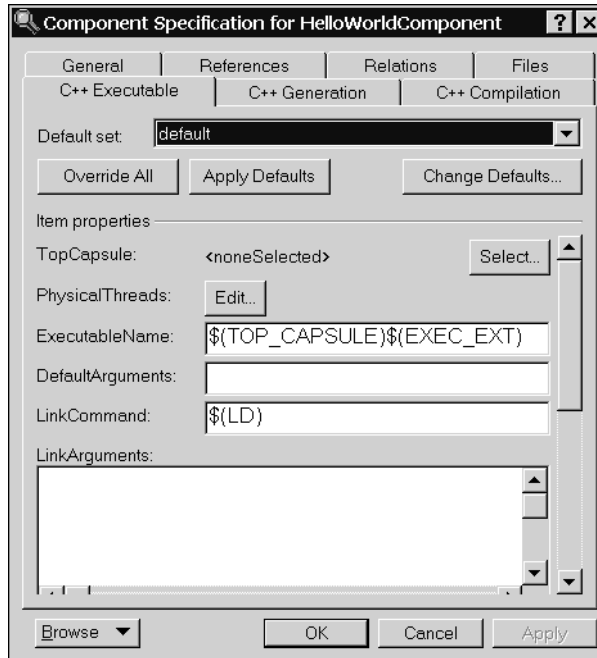
- 4 Click the **General** tab if not already selected.
- 5 In the **Environment** box, select **C++ TargetRTS** if not already selected.
Setting the **Environment** box to C++ TargetRTS specifies that the C++ run-time system and code generation are used in the build process.
- 6 In the **Type** box, select **C++ Executable** if not already selected.
Setting the **Type** box specifies that you want to build a C++ executable version of the model.
- 7 Click the **References** tab.
- 8 In the **Model View** browser, drag the **HelloWorld** capsule in the Logical View folder, onto the **References** tab.

The **HelloWorld** capsule appears in the window.



You drag the **HelloWorld** capsule onto the **References** tab because the items in the **References** tab identify what is compiled with the **HelloWorld** component.

9 Click the **C++ Executable** tab.



10 Click **Select...**

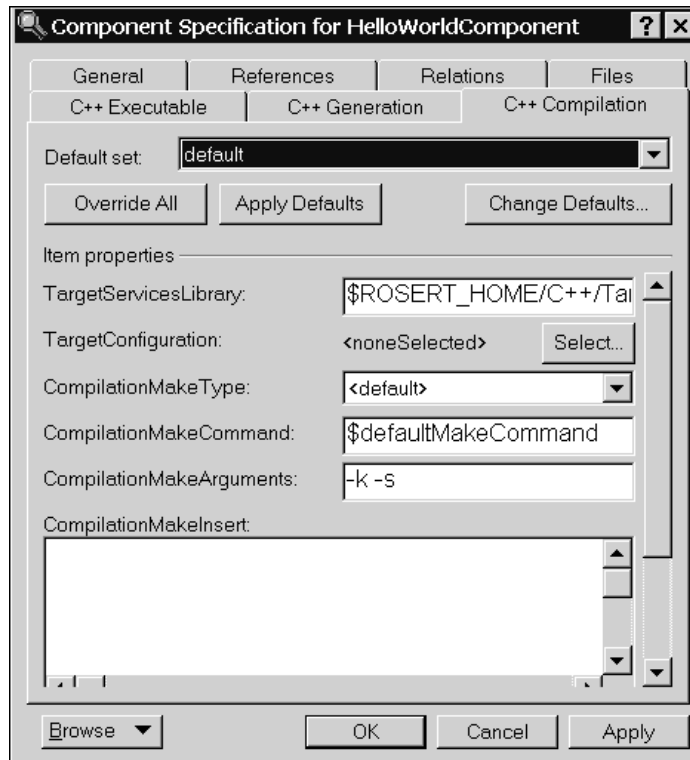
The **Select Top Capsule** dialog box appears. In this dialog box, you select the capsule that will be the top capsule in the model.

11 Click **HelloWorld** to designate it as the top capsule.



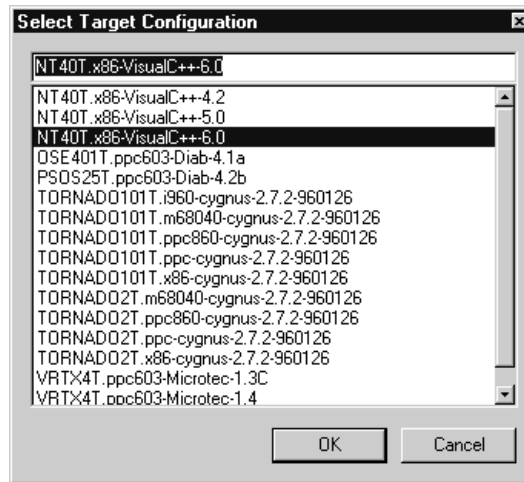
12 Click **OK**.

13 Click the **C++ Compilation** tab.



14 Click **Select...**

The **Select Target Configuration** dialog box appears.



A component is always created with a default configuration for your host machine. This includes a default compiler, compiler flags, linker, and so forth. In many cases, these settings are sufficient for building simple sets of classes and capsules that do not require integration with external source files, or libraries.

In this dialog box, you will specify the operating system, compiler, and processor that you want to use to build and run the model.

The information is listed in the following format:

`<operating system>.<processor and compiler>`

For example:

- If you are running Windows 4.X on a x86 processor, and you have installed the Visual C++ 6.0 build tools, select:

`NT40T.x86-VisualC++6.0`

- If you are running Solaris 5.X on a sparc processor, and you have installed the gnu 2.8.1 build tools, select:

`SUN5T.sparc-gnu-2.8.1`

15 Select the configuration for your computer, and click **OK**.

The sample model showing the completed procedures to this point in the tutorial is located in the Rose RealTime installation directory:

`$ROSERT_HOME/Tutorials/gstarted/QuickstartTutorial.rtmidl`


Building the Component

In order to build an executable of your model, you created a component, called **HelloWorldComponent**, that will be used to manage the build configuration parameters. To build the executable, you need to build (or compile) this **HelloWorldComponent** component.

To build a component:

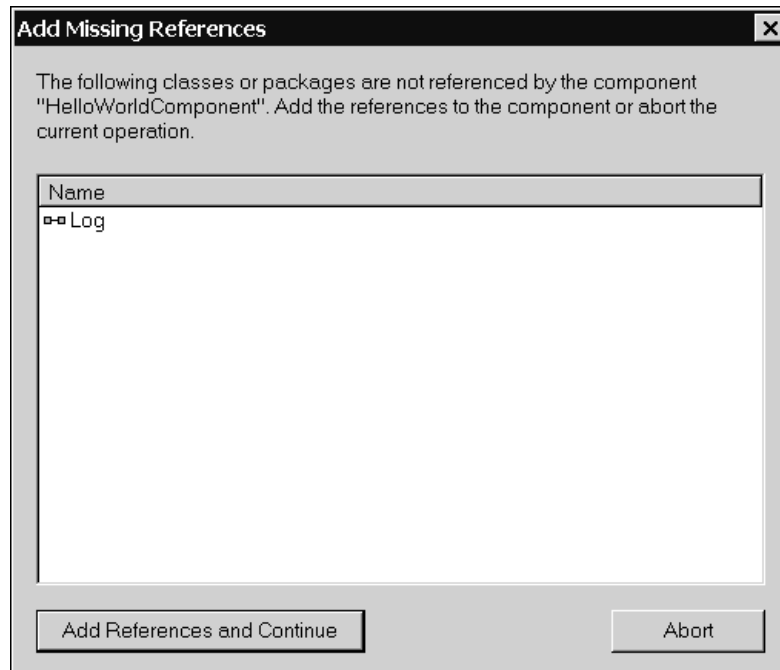
- 1 In the **Model View** browser, right-click **HelloWorldComponent**, and click **Set As Active**.

Because you will be building and running the same component and component instances often, you should configure an active component. Setting the **Set As Active** option ensures that the toolbar build icons and menu items, for the common run and build commands, become available for easy access.

The **Build Component** tool  becomes active.

- 2 Click **Build Component**.

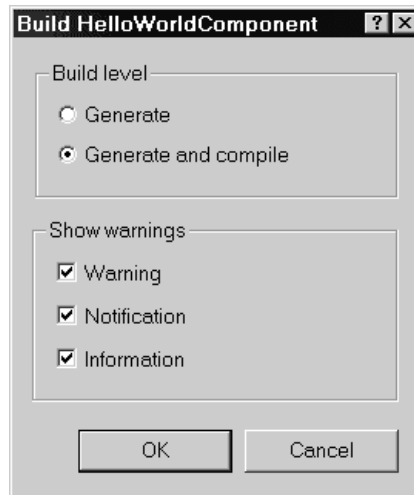
The **Add Missing References** dialog box appears.



The HelloWorldComponent component should contain all referenced classes before it is compiled. Rational Rose RealTime checks the references, and prompts you to add any missing references that it detected.

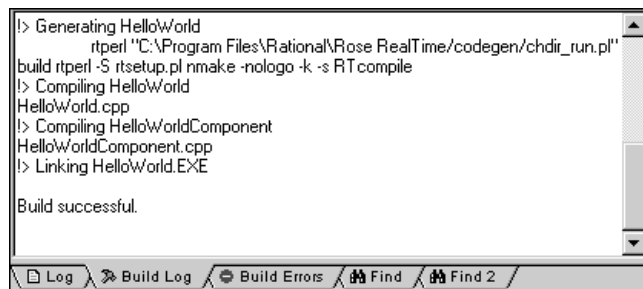
3 Click Add References and Continue.

The **Build HelloWorld Component** dialog box appears.



4 If not currently selected, click Generate and compile, and click OK.

The **Build Log** tab of the **Output Window** shows the results of code generation and compilation. When the build finishes, the **Build Log** should indicate "Build successful".



Note: If there are compile errors, double-click on an error message on the **Build Errors** tab. Rational Rose RealTime opens the appropriate editor where the source of the error appears. You can then resolve any errors.

Creating the Deployment View

The **Deployment View** describes the computing environment in which your model is executed, and specifies how it is deployed within the environment. The most important elements of the **Deployment View** are processors and component instances.

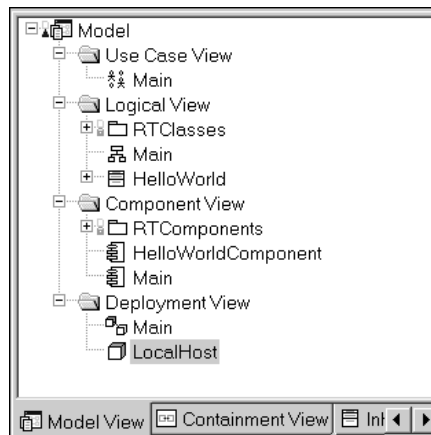
First, you define a processor on the **Deployment Diagram** that describes the computing hardware on which the model will be executed. Next, you map the component onto the processor to create a component instance. Finally, you run the model.

Creating a Component Instance



In this tutorial, there are two ways to create a component instance: in the **Model View** tab in the browser, and using the Toolbar and Toolbox buttons. Step one shows how to use both methods. Use only one of these methods.

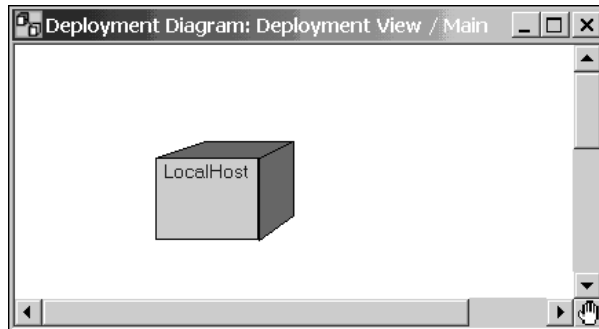
To create a component instance:

- 1 In the **Model View** tab in the browser, right-click **Deployment View**, and click **New > Processor**. Rename the processor **LocalHost** and press **ENTER**.



Or . . .

In the Toolbar, select the **Browse Deployment Diagram** button . For the **Deployment View** package, select **Deployment Diagram: Deployment View / Main** from the Deployment diagrams list, and click **OK**. In the toolbox, click the **Process** tool , and click on the diagram. Rename the processor **LocalHost** and press **ENTER**.

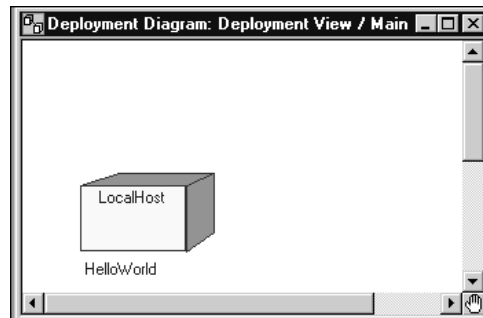


- 2 In the **Model View** tab in the browser, drag the **HelloWorldComponent** from the **Component View** folder onto **LocalHost**.

Or . . .

If you created a processor on the **Deployment Diagram** for **Main**, from the **Model View** tab in the Browser, drag the **HelloWorldComponent** from the **Component View** folder onto **LocalHost** processor in the **Deployment Diagram**.

After dragging the **HelloWorldComponent**, the **Deployment Diagram: Deployment View / Main** dialog looks like the following:



For both methods, a component instance, **HelloWorldComponentInstance** appears under **LocalHost**.



Now that you have associated a component with a processor (creating a component instance), your component instance is complete and you can now run HelloWorldComponentInstance.

Before you run the component instance, save your model (**File > Save Model**)

Running the Component Instance

After you created a component instance by associating a component with a processor, you can run the component instance within Rational Rose RealTime.

To run a component instance:

- 1 In the **Model View** tab in the browser, right-click **HelloWorldComponentInstance** in the **Deployment View** folder, and click **Run**.
- 2 If you are prompted to build the component, click **No**.

Note: Since you previously built the component successfully, you do not need to build it again.

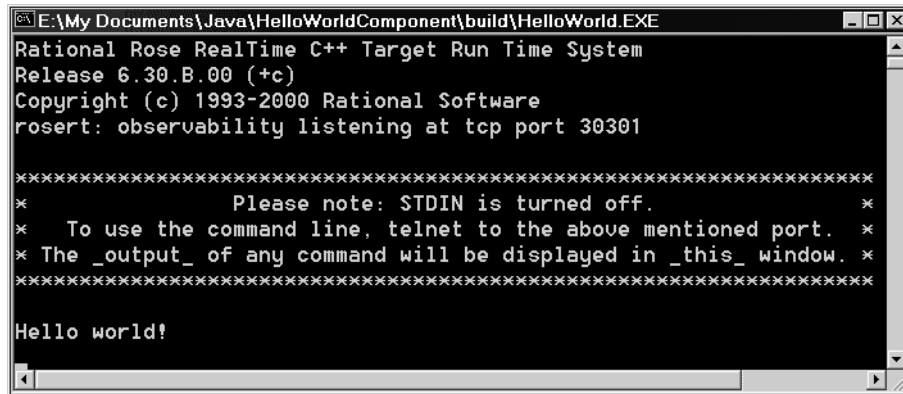
A **Runtime View** browser and a console window appear. The model output displays in the console window.

The model is ready to run.



- 3 Click **Start**  on the **Runtime View** browser.

- 4 Select the console window to make it visible in the Rational Rose RealTime window.



```
E:\My Documents\Java\HelloWorldComponent\build\HelloWorld.EXE
Rational Rose RealTime C++ Target Run Time System
Release 6.30.B.00 (+c)
Copyright (c) 1993-2000 Rational Software
rosert: observability listening at tcp port 30301

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*           Please note: STDIN is turned off.           *
*   To use the command line, telnet to the above mentioned port.   *
* The _output_ of any command will be displayed in _this_ window. *
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Hello world!
```

The "Hello world!" output appears in the console window.

- 5 To terminate the component instance, click **Shutdown**  on the **Runtime View** browser.

The sample model showing the completed procedures to this point in the tutorial is located in the Rose RealTime installation directory:

`$ROSE_HOME/Tutorials/gstarted/QuickstartTutorial.rtmidl`

Tutorial Summary

In the *QuickStart* tutorial, you created a capsule and constructed a system that had one capsule instance. As part of the capsule definition, you defined a log port to access the Log service for printing messages to the console window. You also defined a state machine for the capsule with a single state and an initial transition.

After creating the capsule, you defined a component and deployed the component to enable the generation, compilation, and execution of code for the model.

Finally, you ran the model. At execution time, the Rational Rose RealTime Services Library created a single instance of the top-level capsule class in the model (in this case, the only class in the model), and then called a function to execute the initial transition of the capsule's state machine. The initial transition in turn called a function on Log port to print "Hello world!" in the console window.

After the initial transition executed, the capsule entered the **SaidHello** state where it waited to process any incoming messages. Since there is nothing else in the model, there was no further activity.

Viewing the Generated Code

To examine the generated code, select the folder where you saved your model. In this folder, open the HelloWorldComponent\src folder. The **src** folder contains the generated code for your model.

What's Next?

Try constructing some simple models of your own. Explore the online Help for information on the various modeling tools. You can also use the QuickstartTutorial.rtmidl for further practice on building and compiling a model.

Contents

This chapter is organized as follows:

- *What You Will Learn?* on page 51
- *Before You Begin* on page 53
- *Lesson 1: Creating a New Model and Configuring the Toolset* on page 55
- *Lesson 2: Creating a Use Case and Initial Capsules* on page 59
- *Lesson 3: Sequence Diagrams, Protocols, Ports, and Connectors* on page 81
- *Lesson 4: Building and Running* on page 106
- *Lesson 5: Adding Behavior to the Capsules* on page 122
- *Lesson 6: Navigating and Searching* on page 146
- *Lesson 7: Using Traces and Watches to Debug the Design* on page 147
- *Lesson 8: Class Modeling* on page 165
- *Lesson 9: Adding Card Classes to the Capsule Behavior* on page 198
- *Lesson 10: Aggregating in a State Diagram* on page 219
- *Tutorial Summary* on page 221

What You Will Learn?

The *Card Game* tutorial introduces the basics of creating Rational Rose RealTime applications using the C++ Language Add-in, and demonstrates how to use the main features of Rational Rose RealTime. You will learn the basics of how to:

- Create a new model and configure the toolset.
- Create a use case and initial design elements from the requirements statement: capsules and classes.
- Create sequence diagrams, protocols, ports, and connectors.
- Build and run a model.
- Add behavior (state diagrams) to capsules.
- Navigate and search the model.
- Use traces and watches to debug an error in a model.
- Model classes.
- Use aggregation in a state diagram.

Why a Card Game?

Since most people know how to play cards, you can concentrate on how the tool works without having to spend time understanding the requirements of the tutorial model. Using this model, you will explore the main features of Rose RealTime.



You may also want to refer to the *C++ Reference* to understand the functionality that the language add-in provides.

Card Game Requirements

Card game, the application you build in this tutorial, will be a simulation of a two hand (termed heads-up for a play between only two players), five card stud poker game. There are many variants of poker with many functions having to be implemented in order to simulate all possible poker variations. However, in this tutorial you will only implement a set of simple rules.

The simulation scenario is:

- 1 The player places an ante (minimum ante is \$5).
- 2 The dealer shuffles the cards before each game to reduce the chance of you counting cards.
- 3 The dealer deals the two hands (5 cards each).
- 4 The player shows his hand to the dealer.
- 5 The dealer compares his hand to the players and decides who wins.
- 6 The dealer informs the player if he wins (with the winning bet multiplier) or if he loses.
- 7 The game restarts.

To make things simpler, only the following hands will be checked as possible winning hands:

- 1 pair
- 2 pairs
- 3 of a kind
- full house
- 4 of a kind

In the case of a tie (that is, when the dealer and player have the same hand), the dealer wins.

Before You Begin

Ensure that your environment is properly configured for your compiler. For additional information about configuring your environment, see *Installation Guide, Rational Rose RealTime*.

Tutorial Lessons

The tutorial develops the Card Game application in ten lessons. You will start with a simple design, and add features as you complete each lesson.

Card Game Tutorial Lessons
<i>Lesson 1: Creating a New Model and Configuring the Toolset on page 55</i>
<i>Lesson 2: Creating a Use Case and Initial Capsules on page 59</i>
<i>Lesson 3: Sequence Diagrams, Protocols, Ports, and Connectors on page 81</i>
<i>Lesson 4: Building and Running on page 106</i>
<i>Lesson 5: Adding Behavior to the Capsules on page 122</i>
<i>Lesson 6: Navigating and Searching on page 146</i>
<i>Lesson 7: Using Traces and Watches to Debug the Design on page 147</i>
<i>Lesson 8: Class Modeling on page 165</i>
<i>Lesson 9: Adding Card Classes to the Capsule Behavior on page 198</i>
<i>Lesson 10: Aggregating in a State Diagram on page 219</i>

Each lesson includes a set of procedures and conceptual information that guides you through the Rose RealTime modeling process.

Adding Detail Code to Operations

In this tutorial, you add source code to implement the behavior of the card classes (for example, to shuffle a deck and calculate the value of a hand). We recommend that you take the time to enter the detail code yourself. If you want to complete the tutorial faster, or only partially enter detail code for a few operations, you can import the completed classes from a sample file.

Note: The advantage of adding the code yourself is that you will gain experience working with the tool. Also, you will have an opportunity to practice debugging your model.

Build Information

As you progress through the tutorial, adding code as you read, you should occasionally build the version of the model that you've been developing. After you complete Lesson 5: Adding Behavior to the Capsules, you can build the model or any capsules you need to test. You can also browse the generated source files at any point. For details on where you can find the generated C++ source files, see *Where is the Source Code Generated?* on page 118.

If you encounter problems building the model because there are too many errors reported at build time, you can continue using the finished models that are shipped with the tutorial.

Several Ways of Doing the Same Thing

In Rational Rose RealTime, there are several ways of accomplishing the same task. For example, to open the **Specification** dialog box for an element, you can:

- right-click on the element in a diagram or in the browser, and click **Open Specification** from the context menu

Or . . .

- click or double-click on the element in a diagram.

In Rational Rose RealTime, you can edit models from the **Model View** tab in the browser window, from the **menu**, and from a **diagram**. In this tutorial, the detailed task instructions will only show one way of performing a task. Consult the *Rational Rose RealTime Toolset Guide* for other options. You may want to try different methods as you work through this tutorial.

Also, when specifying code, you can open the appropriate dialog box for an element, or you can use the **Code** pane to edit or view the code associated with the currently selected model element.

Lesson 1: Creating a New Model and Configuring the Toolset

In this lesson, you will learn how to create a new empty model, and configure some basic toolset options.

Suggested Reading

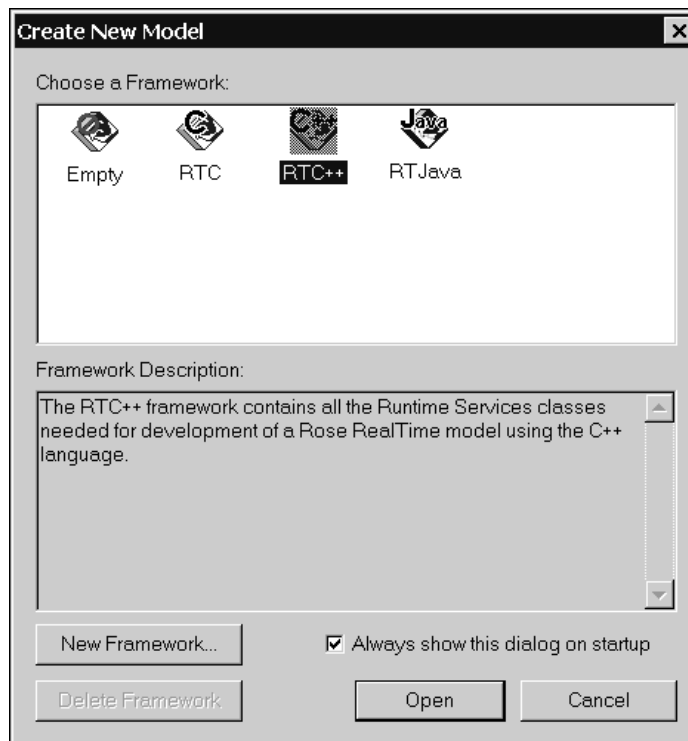
- Opening and Saving Models, (see the *Toolset Guide, Rational Rose RealTime*)
- Model Browser, (see the *Toolset Guide, Rational Rose RealTime*)

Opening a New Model

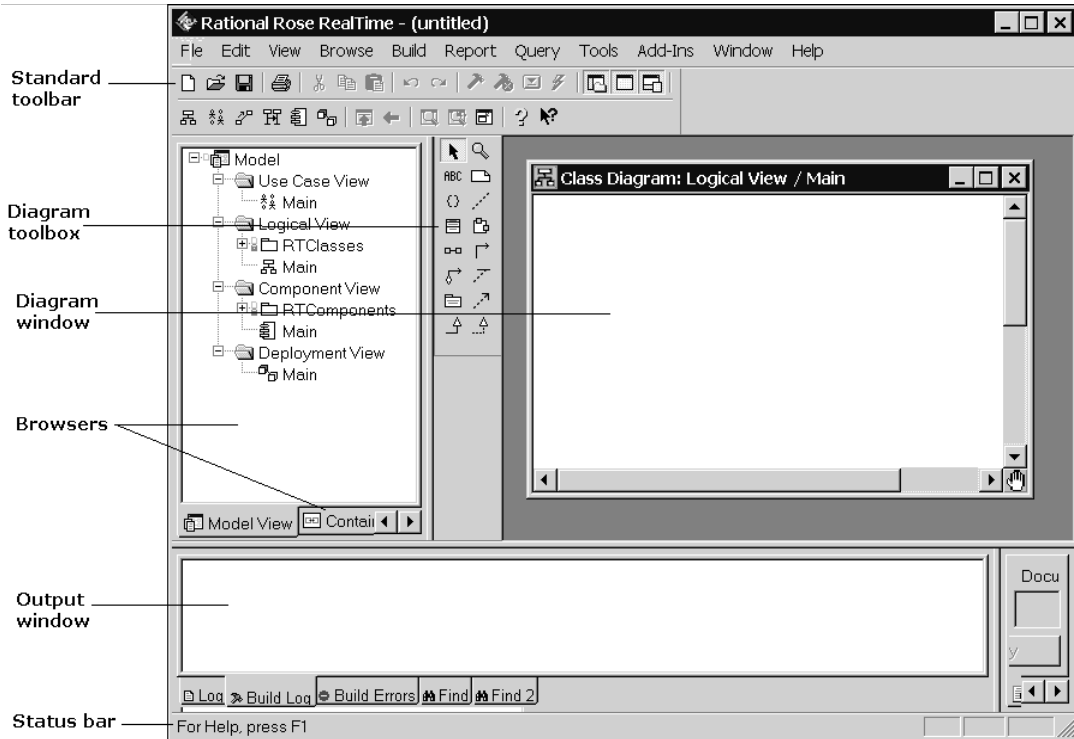
Rational Rose RealTime allows you to work on only one model per Rational Rose RealTime session, therefore if you were working on a previous model, save it before creating the following model for this tutorial.

- 1 On the **File** menu, click **New**.

If presented with the **Create New Model** dialog box, select **RTC++** and click **Open**.



Rational Rose RealTime initializes a new empty model into the toolset. The title bar shows "Rational Rose RealTime - (untitled)" indicating that this is a new model that has not been saved to disk.



Configuring Toolset Options

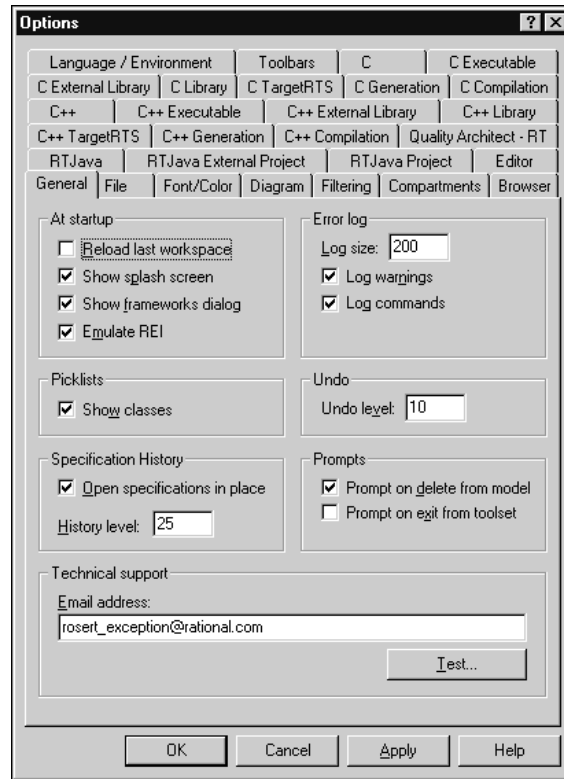
Before you start modeling, there are some user options that you may want to modify.

Suggested Reading

- Options Dialog, (see the *Toolset Guide, Rational Rose RealTime*)

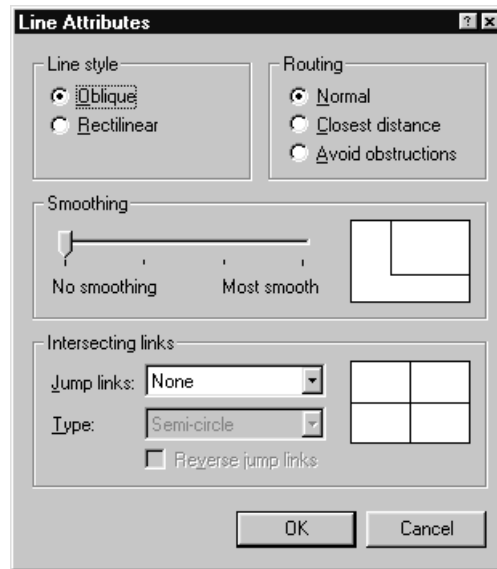
To access the Options dialog box

- 1 On the **Tools** menu, click **Options**.

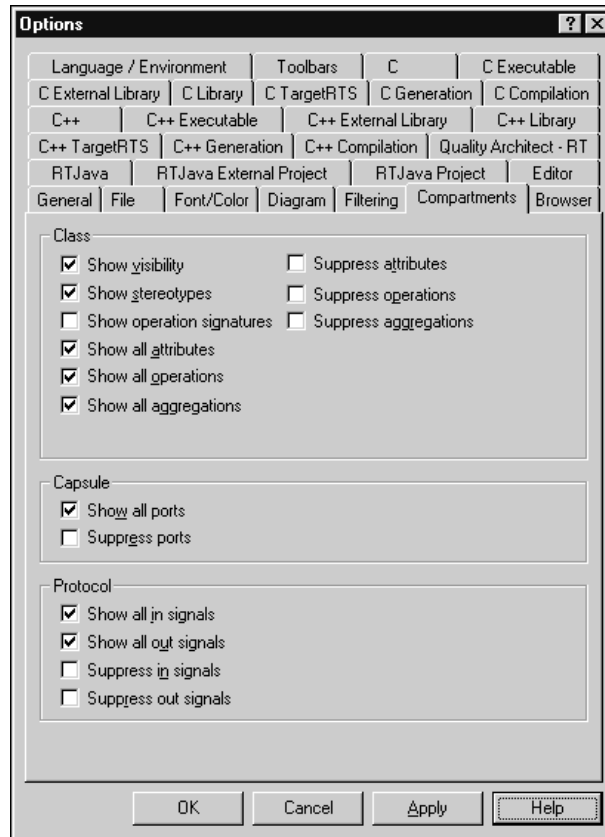


- 2 You may want to consider changing the following attributes to suit your preferences. The options are grouped by the tab on which they can be found:
 - Font/Color: Change the default diagram colors to gray.

- Diagram: Set the default line type (oblique or rectilinear) and grid parameters. The following diagram shows the **Line Attributes** dialog.



- Compartments: For this tutorial, all graphics show the defaults selected in the following diagram.



Lesson 2: Creating a Use Case and Initial Capsules

In this lesson you will add the requirements for the poker simulation and start designing the application. You will:

- Add the textual requirements to a use case diagram
- Create a HeadsUpPoker use case to document the desired functionality of the simulation.
- Create three new capsules: HeadsUpPoker, Dealer, and Player.

You will use the requirements to develop your design. This includes the high-level design, the design elements, and the final implementation.

Modeling is important because it helps the development team visualize, specify, construct, and document the structure and behavior of a system's architecture. Different members of the development team can communicate their decisions unambiguously to one another.

Suggested Reading

- Use Cases, *Rational Rose RealTime Modeling Language Guide*
- Capsules, *Rational Rose RealTime Modeling Language Guide*
- Toolboxes, *Rational Rose RealTime Toolset Guide*
- Specification dialogs, *Rational Rose RealTime Toolset Guide*

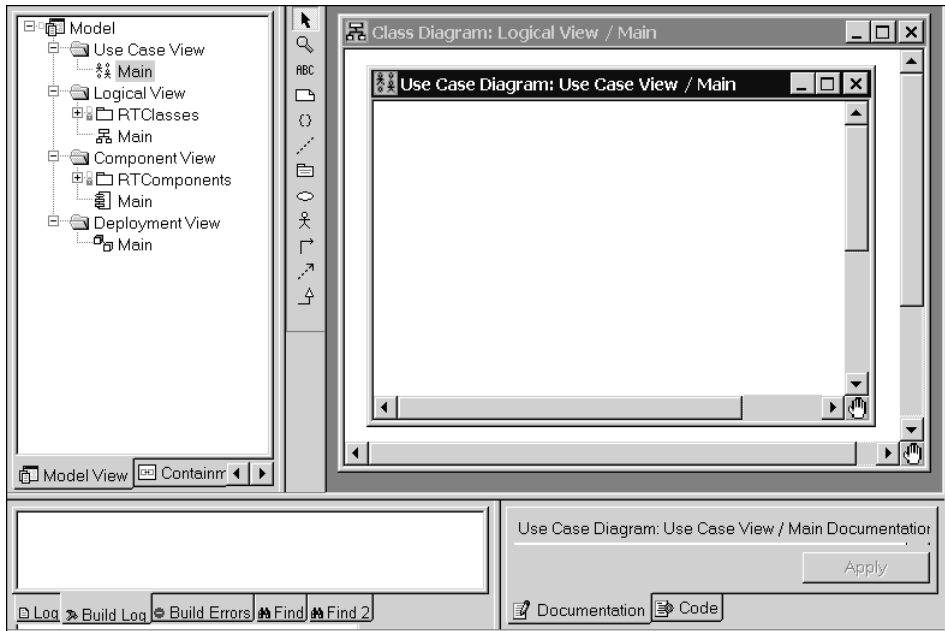
Adding the Use Case


In this topic you will add the textual requirements and a use case. A use case is a description of a set of sequences of actions, called scenarios, that a system performs to yield an observable result of value to an actor. A use case describes what a system (subsystem, class, or interface) does but does not specify how the system internally performs its tasks. For the purposes of this tutorial, you will create a single use case that describes the main flow for the Heads Up Poker.


To enter the textual requirements

- 1 Expand the **Use Case View** folder in the model browser and double-click on the element called **Main**.

A window with the title **Use Case Diagram: Use Case View / Main** appears.

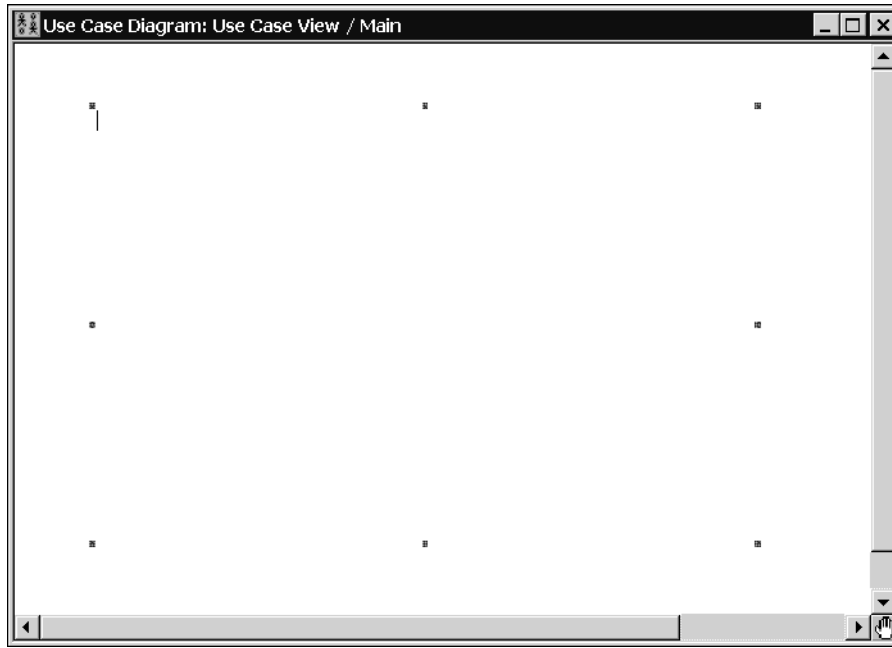


- 2 Ensure that the **Use Case Diagram: Use Case View / Main** window is the active window.
- 3 From the toolbox, select the **Text** tool .

As you move your mouse over the **Use Case Diagram**, your cursor changes to the text tool .

- 4 Left-click in the diagram, and drag (it will show as an outlined rectangle as you drag).

- 5 Release the mouse button after you have a rectangle similar to that in the following diagram (you can resize the rectangle later).



A cursor appears in the top left-hand corner of the text area.

- 6 Type the following text into the text box:

HeadsUpPoker simulation textual requirements.

HeadsUpPoker will be a simulation of a two hand (termed heads-up for a play between only two players), five (5) card stud (in draw poker games you can draw new cards from the deck after the initial cards are dealt; in stud poker games, you're stuck with the cards you get) poker game. There are many variants of poker with many functions having to be implemented in order to simulate all possible poker variations. However, this simulation requires that only the following poker hands be considered:

1 pair, 2 pairs, 3 of a kind, full house, 4 of a kind.


In case of a tie, that is when the dealer and player have the same hand the dealer wins.

The simulation should run continuously.

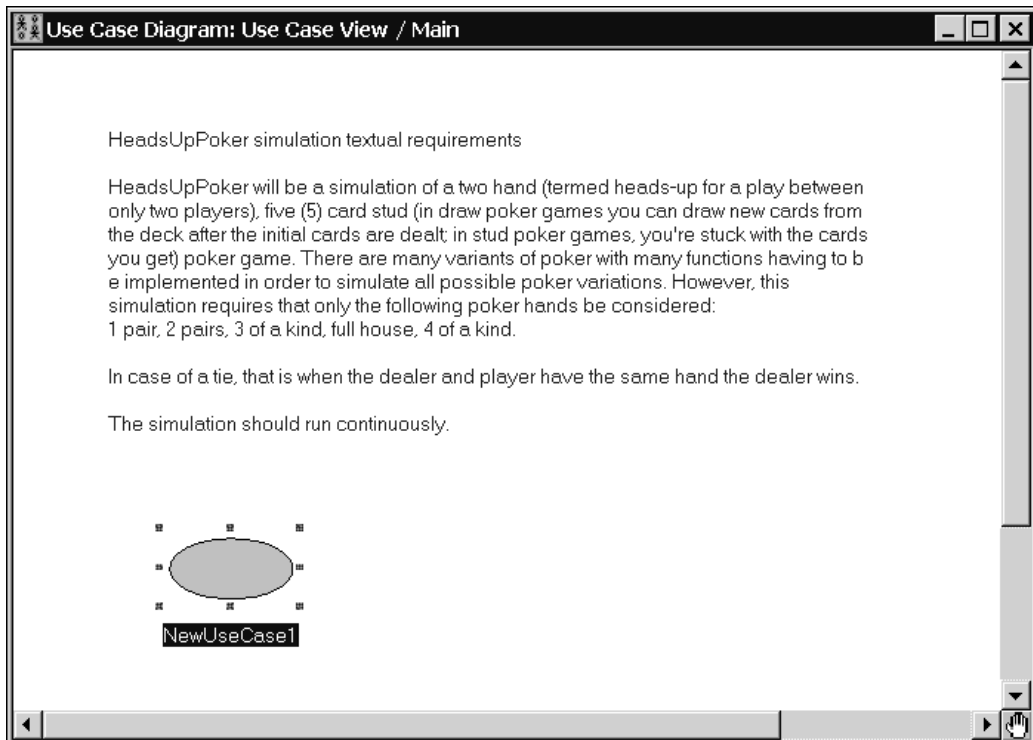
Note: Rational Rose RealTime allows you to also link any kind of file to certain model elements. This can be useful in cases where the textual requirements or use cases already exist in other documents. You can attach these files to the model by using the **Files** tab that appears in most Specification dialogs.

To create the use case

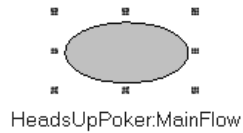
Ensure the **Use Case Diagram** window in which you entered the textual requirements is the active window.

- 1 Select the **Use Case** tool from toolbox .
- 2 Left-click in the diagram.

A use case element appears with a default name.



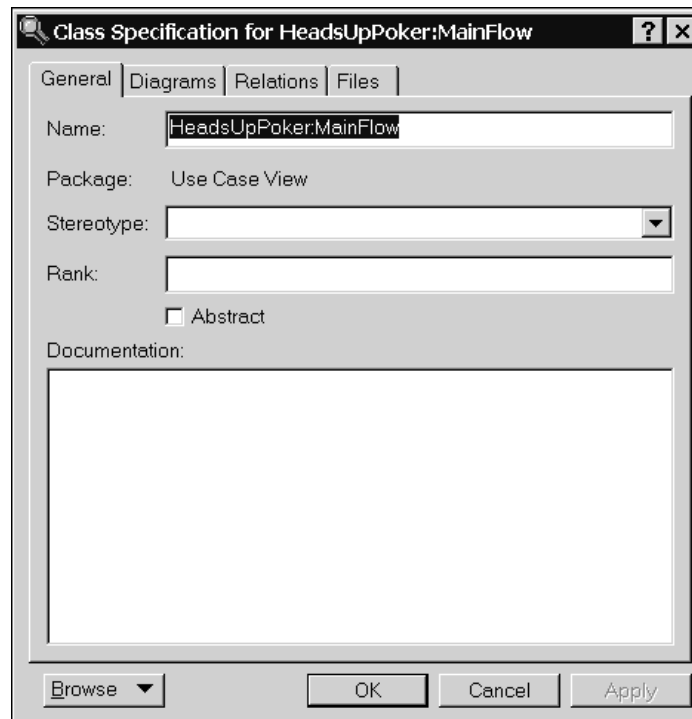
- 3 Rename the use case to **HeadsUpPokerMainFlow**.



Add a flow of events to the use case

- 1 Double-click on the **HeadsUpPokerMainFlow** use case.

The **Class Specification for HeadsUpPokerMainFlow** dialog box appears.



2 In the **Documentation** box, type the following scenario description:

Use case starts: when the application is run

1. The Player places an ante of 5 dollars.
2. The Dealer shuffles the 52 card deck before each game to reduce the chance of players counting cards.
3. The Dealer deals the two hands (5 cards each) distributing cards alternating between the Player and himself. The first card is dealt to the Player.
4. The Player shows his hand to the Dealer.
5. The Dealer compares his hand to the Player's and decides who wins.
6. The Dealer informs the Player if he wins (with the winning bet multiplier) or if he loses.
7. The game restarts.

Use case ends: when the application is stopped.

3 Click **OK**.

This scenario defines **what** the simulation application must do, but not **how** to do it. Determining how to implement this simulation is independent of the requirements. In fact, there can be an unlimited number of ways to implement an application that satisfies the above requirements.

Your case diagram will look like this:

The screenshot shows a software development environment with a window titled "Use Case Diagram: Use Case View / Main". The main area contains the following text:

HeadsUpPoker simulation textual requirements

HeadsUpPoker will be a simulation of a two hand (termed heads-up for a play between only two players), five (5) card stud (in draw poker games you can draw new cards from the deck after the initial cards are dealt; in stud poker games, you're stuck with the card s you get) poker game. There are many variants of poker with many functions having to be implemented in order to simulate all possible poker variations. However, this simulation requires that only the following poker hands be considered:
1 pair, 2 pairs, 3 of a kind, full house, 4 of a kind.

In case of a tie, that is when the dealer and player have the same hand the dealer wins.

The simulation should run continuously.

Below the text is a Use Case Diagram showing a central oval labeled "HeadsUpPoker:MainFlow" surrounded by eight small squares representing use cases.

Overlaid on the diagram is a "Class Specification for HeadsUpPoker:MainFlow" dialog box. The dialog has tabs for "General", "Diagrams", "Relations", and "Files". The "General" tab is active, showing the following fields:

- Name: HeadsUpPoker:MainFlow
- Package: Use Case View
- Stereotype: (empty dropdown)
- Rank: (empty text box)
- Abstract

The "Documentation" section contains the following text:

Use case starts: when the application is run

1. The Player places an ante of 5 dollars.
2. The Dealer shuffles the 52 card deck cards before each game to reduce the chance of players counting cards.
3. The Dealer deals the two hands (5 cards each) distributing cards alternating between the Player and himself. The first card is dealt to the Player.
4. The Player shows his hand to the Dealer.
5. The Dealer compares his hand to the Players and decides who wins.
6. The Dealer informs the Player if he wins (with the winning bet multiplier) or if he loses.
7. The game restarts.

Use case ends: when the application is stopped.

At the bottom of the dialog are "Browse", "OK", "Cancel", and "Apply" buttons.

Documentation Window

Suggested Reading:

- Documentation window, (see the *Toolset Guide, Rational Rose RealTime*)

In the remainder of this tutorial, you will add documentation to almost every model element you create. It is important to document the responsibilities of all elements and any additional information that would make the model more understandable.

Click on the new use case to view the flow of events in the documentation window.

Are Elements Owned by the Class Diagram?

When you place elements on a class diagram do they exist only in the context of that diagram? The answer is yes and no. Most elements you add to a class diagram are actually added to the model (that is, they appear in the Browser window when created in a diagram). This means that a single model element can appear in several class diagrams, even on diagrams in other views. Notes and freeform text are owned by a specific diagram.

Meaning of the Delete Key in Class Diagrams

You can delete elements from the model rather than deleting from the diagram. You can delete elements owned by the diagram (such as, states, transitions, call messages, and capsule roles) using the `DELETE` key or the `CTRL+D` shortcut because they exist only in the context of the diagram in which they appear.

To delete items which are not owned by the diagram (such as, any class on a class diagram), use the `CTRL+D` shortcut. This will delete the class from the model as well as the diagram.

To delete elements which are not owned by the diagram (that is, you can see the elements in the **Model View** tab in the browser) only from the diagram and not from the model, use the `DELETE` key.

Defining the Classes

You will define the classes and capsules that implement the `HeadsUpPoker:MainFlow` use case.

Suggested Reading

- Capsules, (see the *Rational Rose RealTime Modeling Language Guide*)
- Classes, (see the *Rational Rose RealTime Modeling Language Guide*)

After the requirements have been defined, you identify a candidate set of classes capable of performing the behavior described in the use case. There are several methods that exist for finding classes from a mere statement of system behavior. It is out of the scope of this tutorial to describe them all. Within Rose RealTime, the basic transformation from the requirements to a description of how the system will work requires you to identify the domain classes from the requirements, and classify them as either classes or capsules.

Classes Versus Capsules

Generally, classes in a Rose RealTime model represent stores of information in the system. They are typically used to represent the key concepts the system manages. Classes store and manage information in the system, whereas capsules provide coordinating behavior in the system. The system can perform some use cases without capsule objects particularly use cases that involve only the simple manipulation of stored information. More complex use cases generally require one or more capsules to coordinate the behavior of other objects in the system. In addition, capsules are units of concurrency with the model.

Flows of events should be encapsulated in a separate capsule when it is complex and consists of dynamic behavior (that is, when it describes the behavior of several other objects interacting in some fashion). For simple flows of events that primarily enter, retrieve and display, or modify information, a class is usually justified instead of a capsule.

Capsules also give you transparent concurrency, easy thread assignment, state diagram generation, and message passing.

Describing the Behavior of the Classes

You identify the main classes and capsules from the requirements. Then, you describe the flow of events between capsules identified in the use case in a capsule structure and sequence diagrams. These two diagrams:

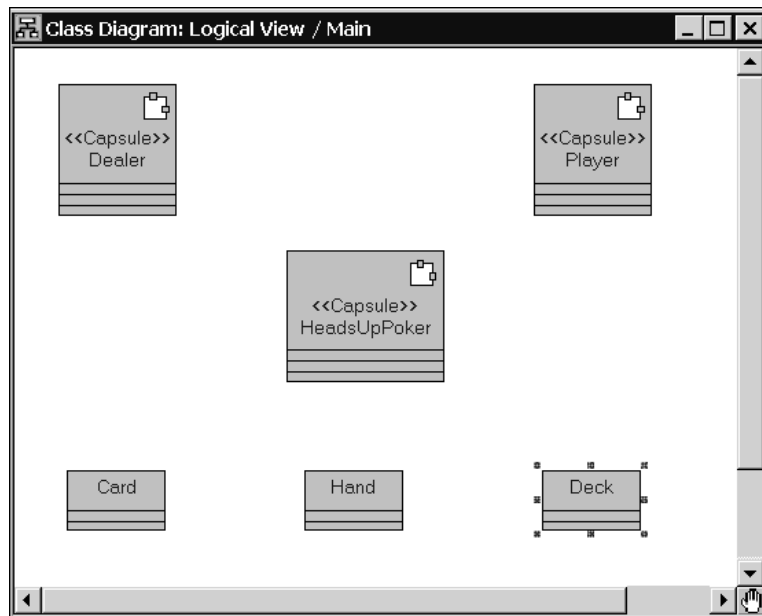
- help you identify the responsibilities of the capsules and how they interact to perform the behavior defined in the use case
- allow you to identify the ports and protocols that will be required to allow the capsules to communicate between each other

Creating Classes and Capsules

For the **HeadsUpForkerMainFlow** use case, you will create the following classes and capsules:

- **HeadsUpPoker** (capsule) - Encapsulates the flow of events described in the HeadsUpForker:MainFlow use case
- **Dealer** (capsule) - Deals the cards and determines who wins
- **Player** (capsule) - Initiates the simulation by placing an ante
- **Card** (class) - An individual card
- **Hand** (class) - Set of 5 cards given to each participant
- **Deck** (class) - Set of 52 cards from which cards are distributed to the game players


When finished adding these capsules and classes, your **Class Diagram: Logical View / Main** diagram will look like this:

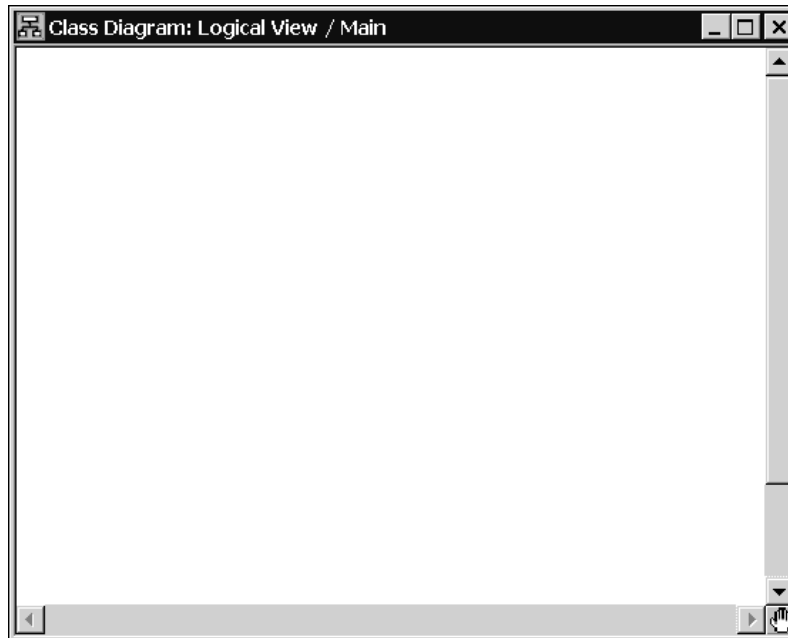



To create capsules and classes

- 1 If the **Logical View** folder in the **Model View** tab in the browser is not expanded, click on the plus sign [+] to expand it, and double-click on the **Main** class diagram.

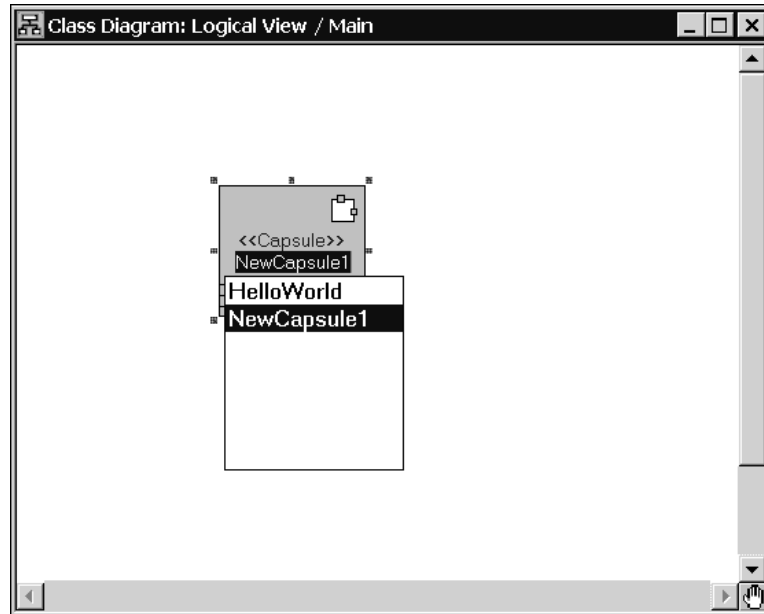
Or . . .

From the toolbar, select the **Browse Class Diagram** button , then from the **Logical View** package, select **Class Diagram: Logical View / Main** and click **OK**.



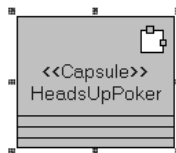
- 2 From the **Main** class diagram toolbox, click the **Capsule** tool .
- 3 Move the mouse over the diagram, and left-click in the window.

A new capsule appears in the diagram.



- 4 Rename the capsule **HeadsUpPoker**.

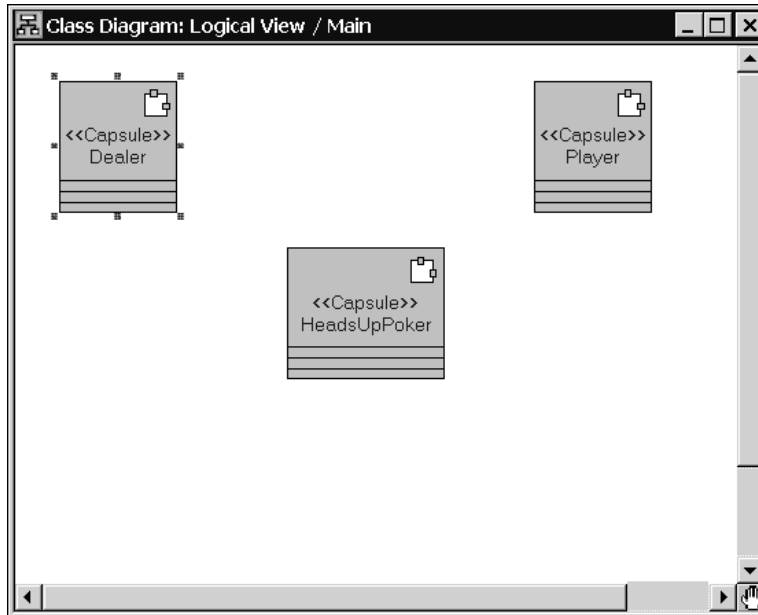
After you type the name, you may have to left-click in the diagram to exit edit mode, and to save the new name.




- 5 Repeat the above steps to add the **Player** and **Dealer** capsules.

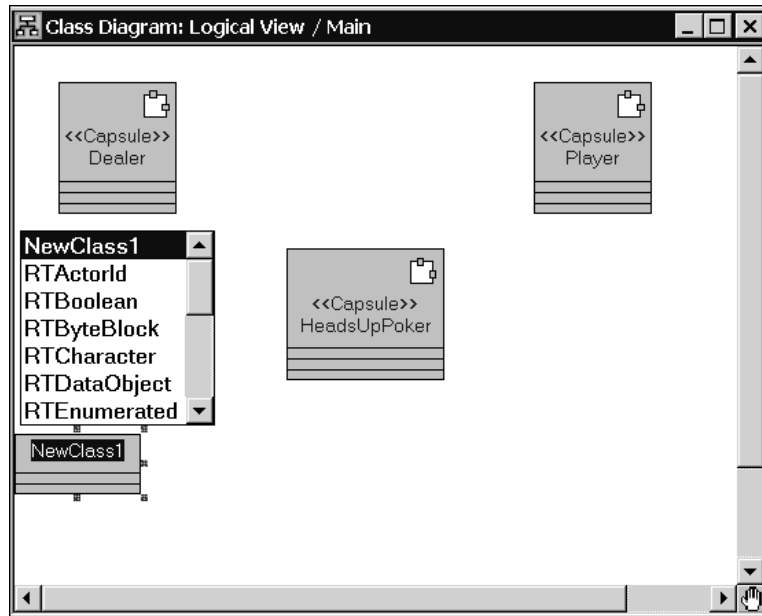
Note: You can change any model element's property (including the name) with the **Specification** dialog box.

When finished adding all three capsules, your **Class Diagram** will look like the following:



- 6 To add the classes, select the **Class** tool  from the **Main** class diagram toolbox.
- 7 Move the mouse over the diagram, and left-click in the window.

A new class appears in the diagram.

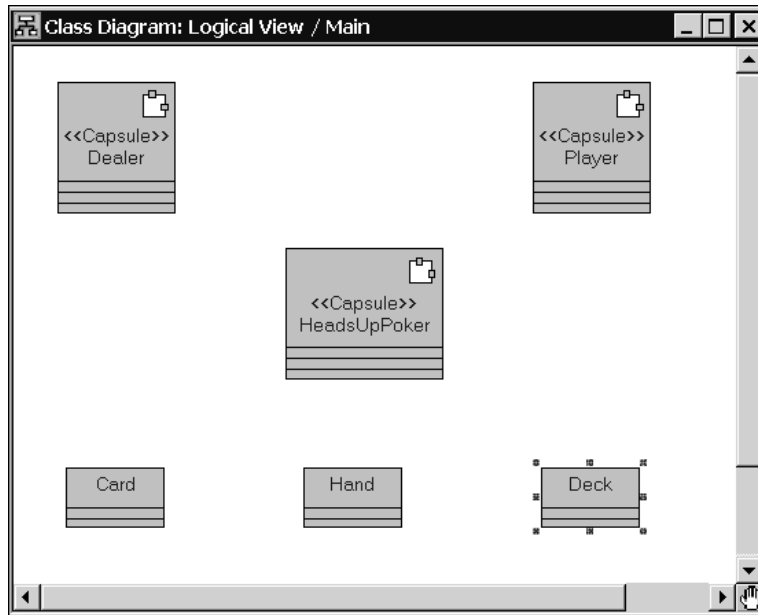


- 8 Rename the class to **Card**.

After you type the name, you may have to left-click in the diagram to exit edit mode, and save the new name.

- 9 Repeat the above steps to add the **Deck** and **Hand** classes.

When finished, your class diagram will look like this:



RTClasses Package

The **RTClasses** package exists in each model and contains a set of predefined classes and protocols implemented by the Services Library. Most models require the use of one or many of these predefined classes. For example, the package contains a Timing protocol used to communicate with the timing service.

Changing Element Types

You can change the element type within a model. For example, you initially designed something as capsule and realize that it should be a class. You can change the element to another type using the menu command **Edit > Change Into**. The menu item is context-sensitive and displays only the valid elements to which you can change the currently selected item.

Note: When transforming elements, information not part of the destination element is lost during the transformation. For example, when transforming a capsule to a class, the capsule roles and state diagram are not part of the converted class element.

Creating the HeadsUpPoker Capsule Structure

In this topic you create a capsule structure diagram of capsules which implement the behavior described in the use case. Then, you create a sequence diagram to show how they will interact to implement the behavior described in the flow of events.

Since the **HeadsUpPoker** capsule encapsulates the flow of events described in the use case that you are implementing, it also encapsulates all the classes and capsules necessary to implement the use case.

Suggested Reading

- Capsules, *Rational Rose RealTime Modeling Language Guide*
- Capsule structure diagrams, *Rational Rose RealTime Modeling Language Guide*
- Sequence Diagrams, *Rational Rose RealTime Modeling Language Guide*
- Interactions, *Rational Rose RealTime Modeling Language Guide*

Creating the HeadsUpPoker Capsule

There are two main ways that you can aggregate capsules with one another:

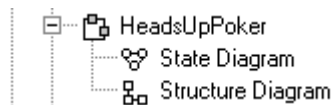
- Use the **Aggregation** tool in the class diagram
- Drag the capsule onto the structure diagram of another capsule

Since both methods are equivalent, you will use one method only. When a capsule is added to the structure of another, you are creating an attribute of the capsule called a capsule role. The capsule role has a name that describes the capsule's role in relation to the containing capsule.

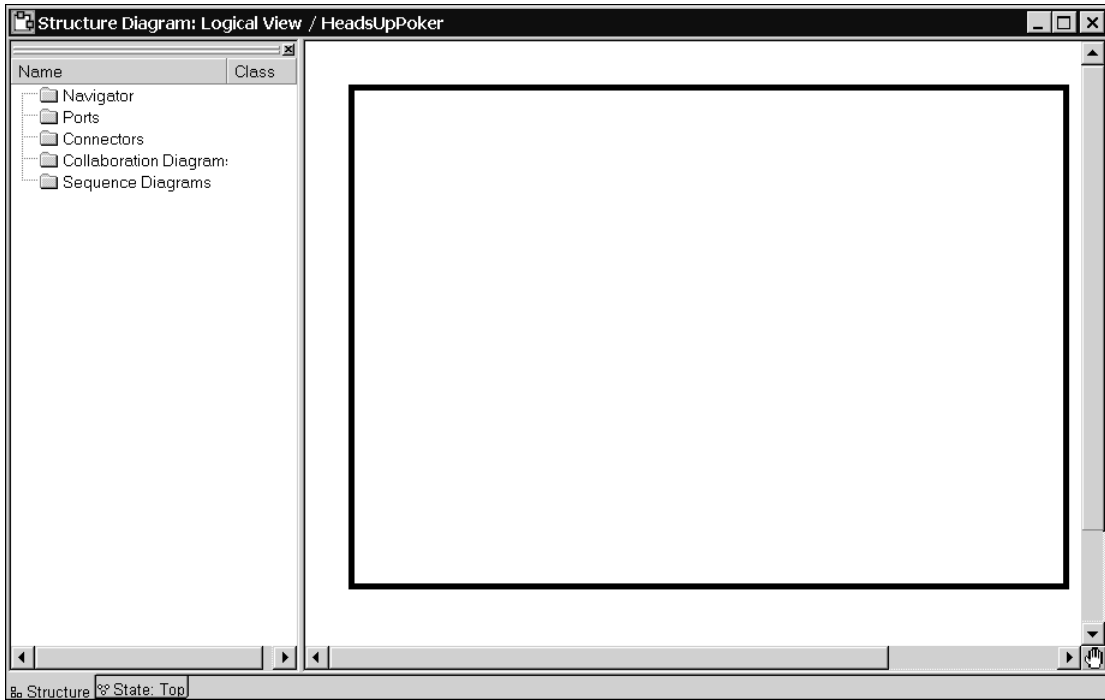
To create the HeadsUpPoker capsule structure

- 1 From the **Model View** tab in the browser, click the plus sign [+] beside the icon to expand the **HeadsUpPoker** capsule.

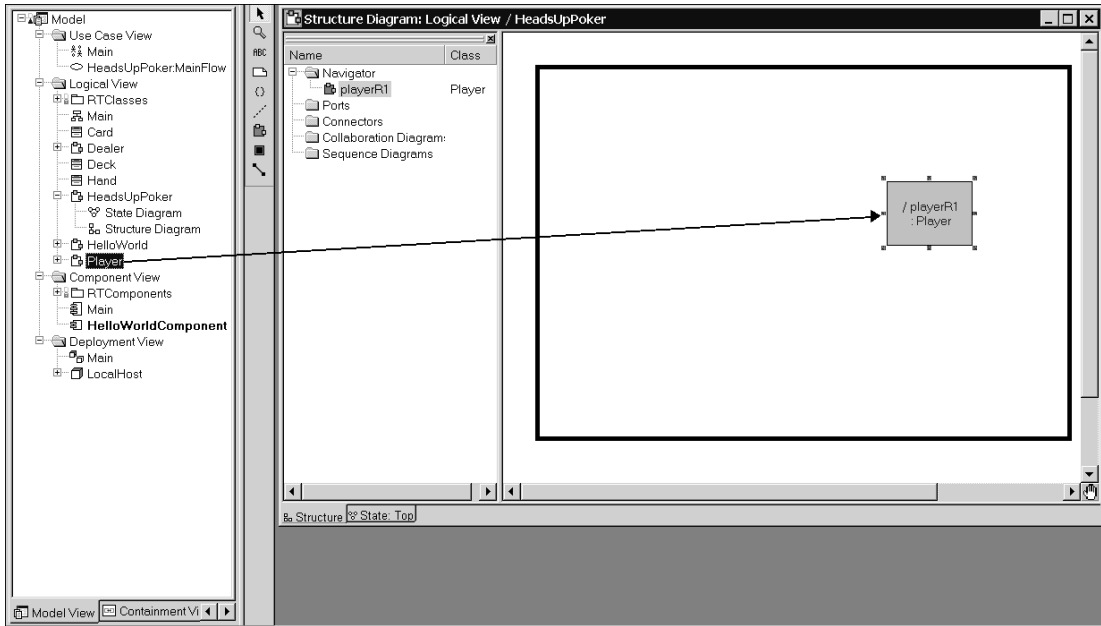
Two items appear: one representing the state diagram, and the other the structure diagram:



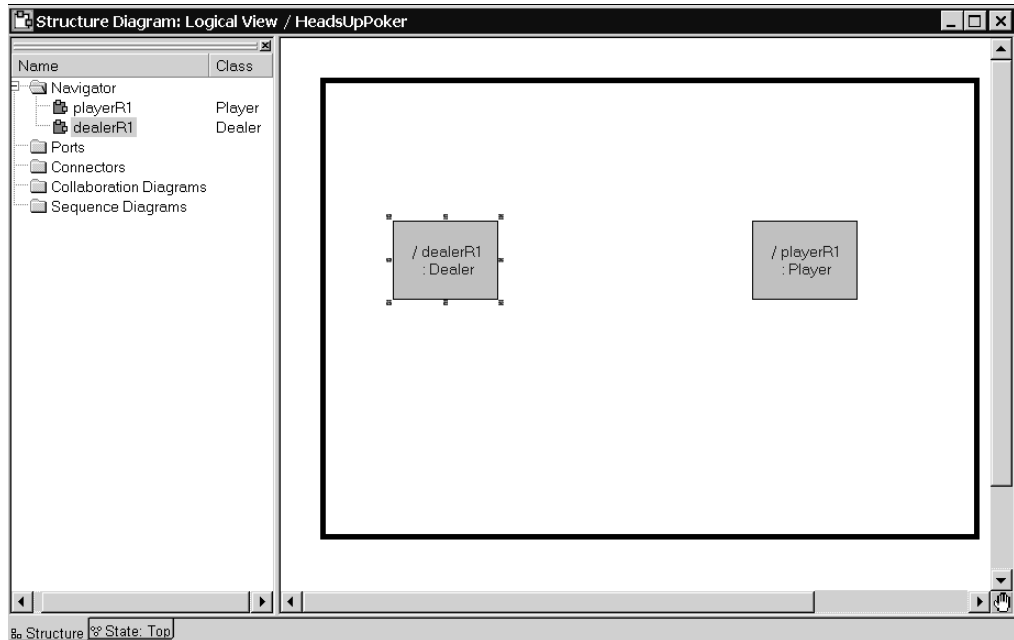
2 Double-click on the **Structure Diagram** icon to open the diagram window.



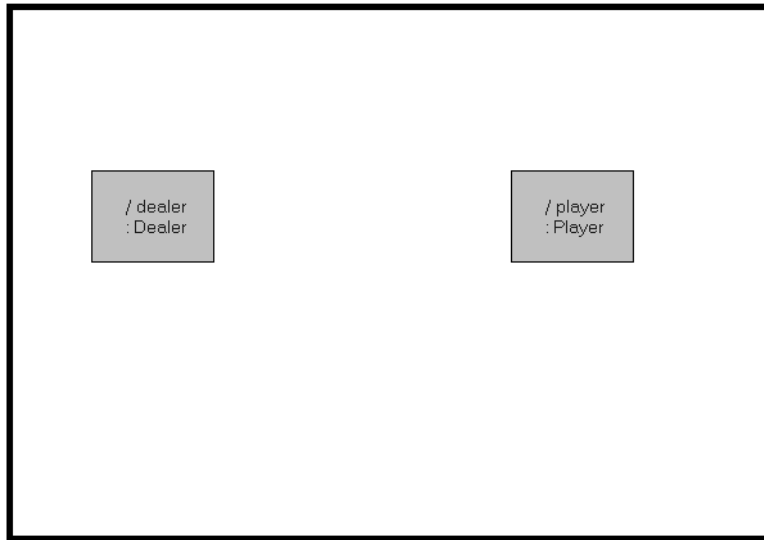
- 3 Drag the **Player** capsule from the **Model View** tab in the browser into the **HeadsUpPoker** structure diagram.



- 4 Repeat the above step for the **Dealer** capsule.

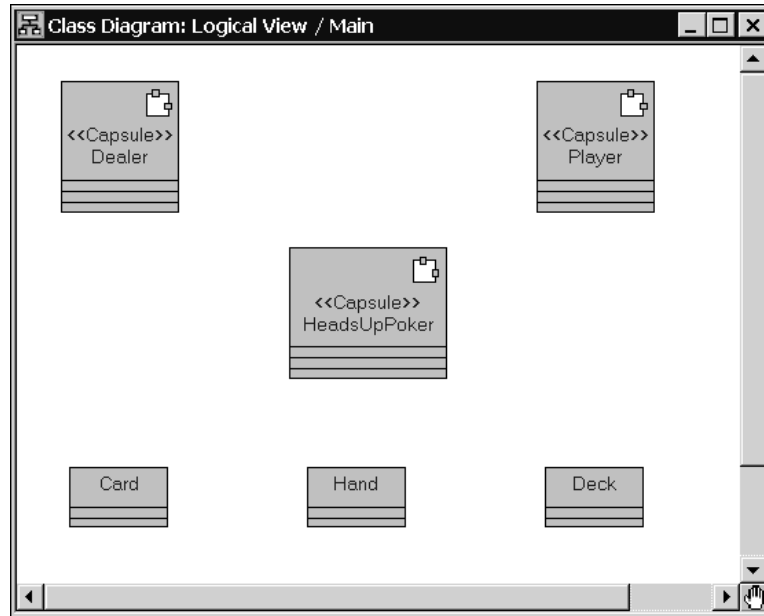


- 5 Double-click the **/playerR1** capsule role and rename it **player**.
- 6 Double-click the **/dealerR1** capsule role and rename it **dealer**.

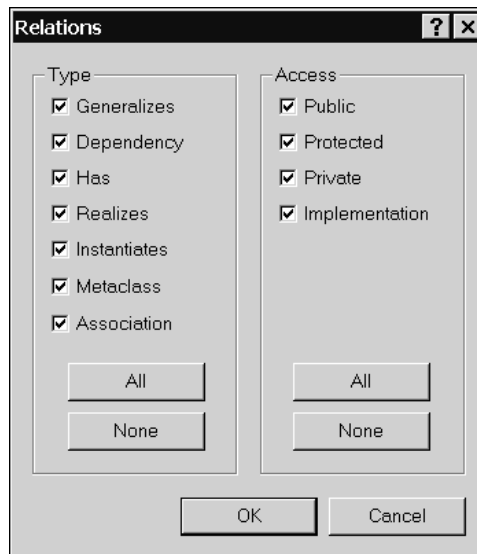


- 7 In the **Logical View** folder, double-click **Main** to open the **Class Diagram: Logical View / Main** dialog.

You should see the three capsules and three classes that you have previously created.



- 8 From the **Query** menu click **Filter Relationships**.



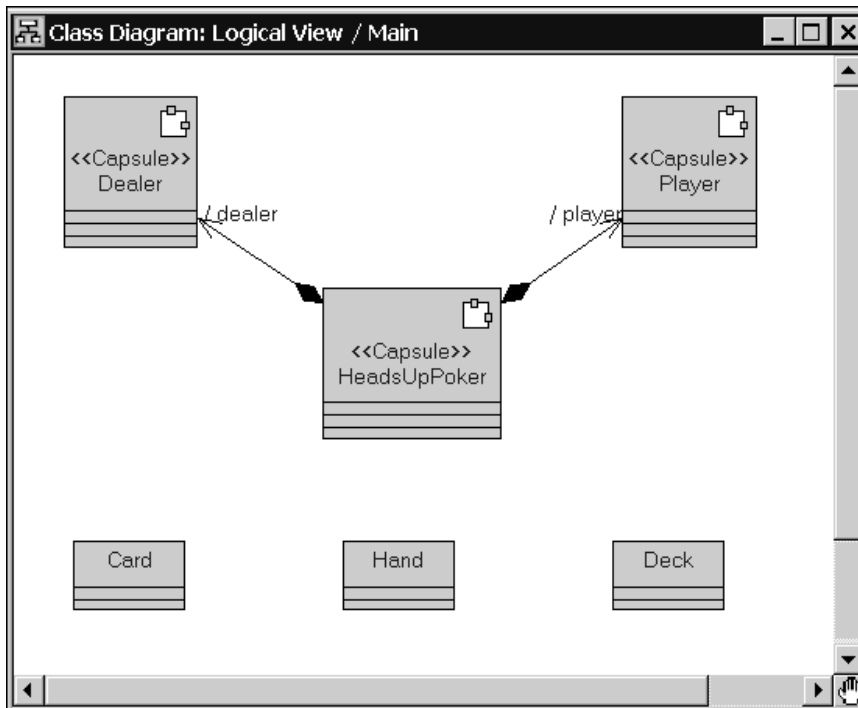
The Relations dialog allows you to control which relationships appear in diagrams. You can also use it to refresh the current diagram.

9 Ensure that all checkboxes in the **Type** and **Access** groups are selected.

10 Click OK.

In the **Class Diagram: Logical View / Main** diagram, you will see aggregation relationships between the **HeadsUpPoker** capsule and the **Dealer** and **Player** capsules. The capsule role names appear at the part-of end of the aggregation relationship.

If necessary, re-arrange the class diagram so that you can properly see the relationships between the capsules.



Both representations are essentially the same, with one exception. As you will see later from the **Structure Diagram**, you can interconnect capsule roles, something that is not possible in the class diagram.

Before proceeding with Lesson 3, we recommend that you save your work.

Note: If you do not want to proceed to Lesson 3 using the file you created at the end of Lesson 2, you can open the file `<ROBERT_HOME>/Help/Tutorials/cardgame/cardgame_step1.rtmidl`, and then continue with the steps in Lesson 3.


Lesson 3: Sequence Diagrams, Protocols, Ports, and Connectors

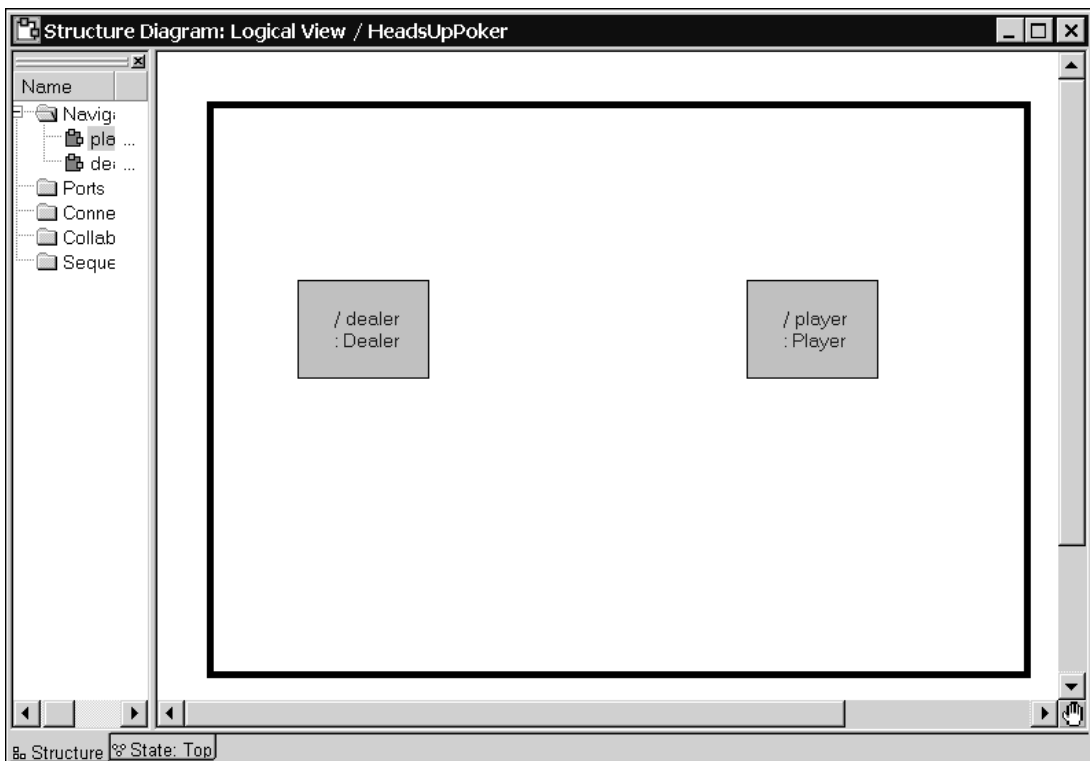
In the following lesson, you create a sequence diagram on the **HeadsUpPoker** capsule to show how its contained capsule roles interact to implement the behavior described in the **HeadsUpPokerMainFlow** use case. This sequence diagram should highlight the important design decisions. You can also add scripts to a sequence diagram to expand and explain the interaction.

To create the Sequence Diagram

- 1 In the **Model View** tab in the browser, expand **Logical View**, expand the **HeadsUpPoker** capsule, then double-click **Structure Diagram**.

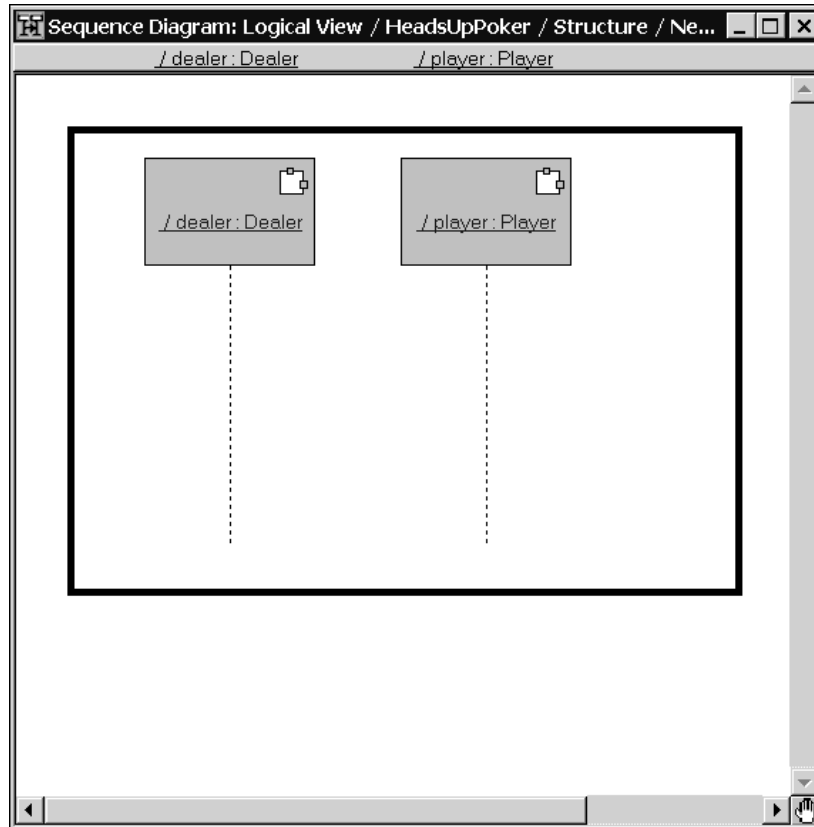
Or . . .

From the toolbar, select the **Browse Collaboration Diagram** button , then select **Structure Diagram: Logical View / HeadsUpPoker** from the **Logical View** package.

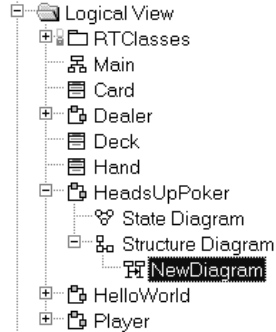


- 2 Using the `SHIFT` key, select both the outer boxes for the player and dealer capsule roles.
- 3 Right-click on an empty area within the border of the **Structure Diagram** (do not right-click on capsule roles or the back border), and click **Create Sequence Diagram**.

A new sequence diagram appears with both capsule roles already in the sequence.



The diagram appears in the **Model View** tab in the browser under the **Structure Diagram** for the **HeadsUpPoker** capsule.

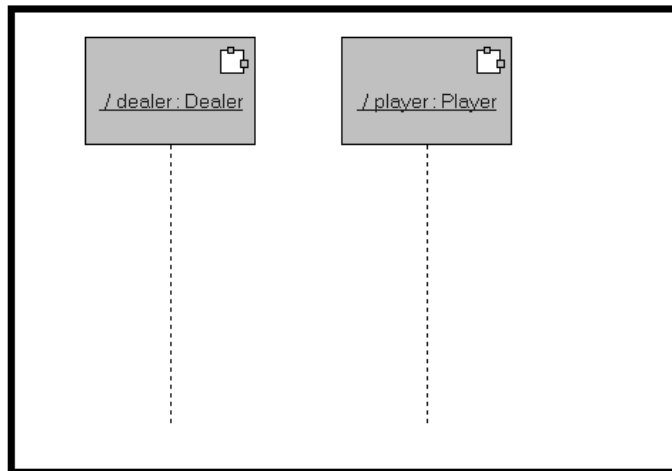


- 4 Right-click **NewDiagram** and click **Rename**.
- 5 Rename **NewDiagram** to **HeadsUpPokerMainFlow**.

To create the interactions in the sequence diagram

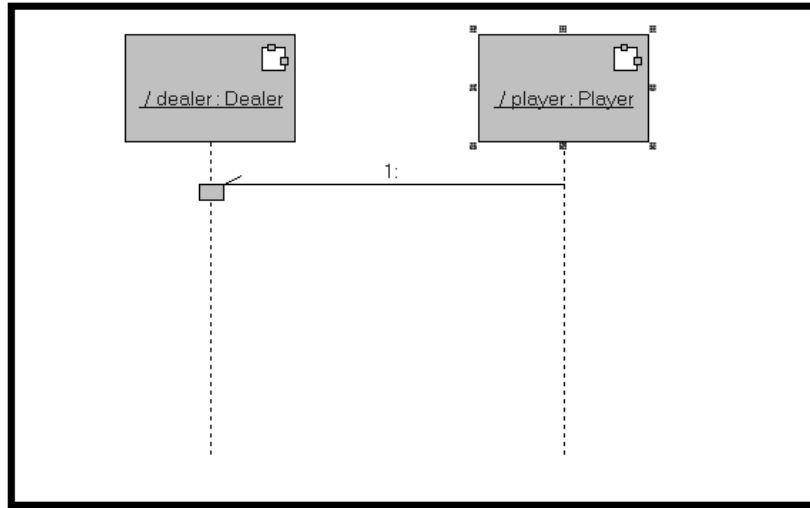
You will describe the interactions between these two capsules by referring to the flow of events described in the **HeadsUpPokerMainFlow** use case.

- 1 Ensure that the sequence diagram window for **HeadsUpPokerMainFlow** is active.

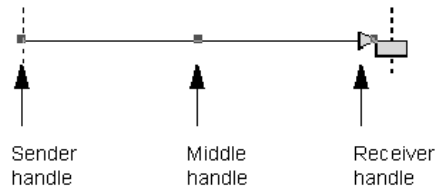


- 2 Select the **Asynchronous Send Message**  tool from the toolbox.

- 3 Left-click on the player lifeline (dotted line beneath capsule role box), and drag the mouse to the dealer's lifeline.



The vertical placement of messages represents time. The direction of the messages represents the flow.



- 4 Double-click on the message line (not the label for the line) to open the **Send Message Specification** dialog box.

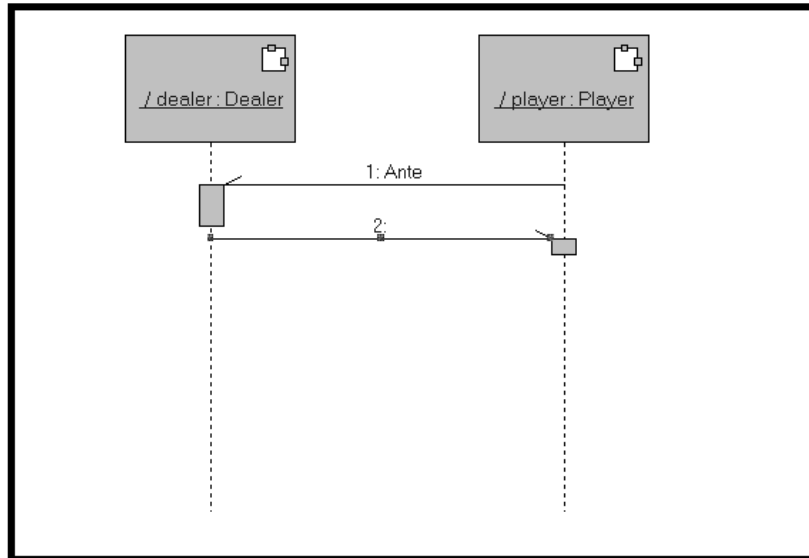


- 5 On the **General** tab, type **Ante** in the **Name** box.
- 6 Click **OK**.

Next, you will add five messages; one message for each of the five cards that make up a poker hand.

- 7 Select the **Asynchronous Send Message**  tool from the toolbox.

- 8 Left-click on the dealer lifeline (dotted line beneath capsule role box), and drag the mouse to the player's lifeline.




- 9 Double-click on the message line (not the label for the line) to open the **Send Message Specification** dialog box.

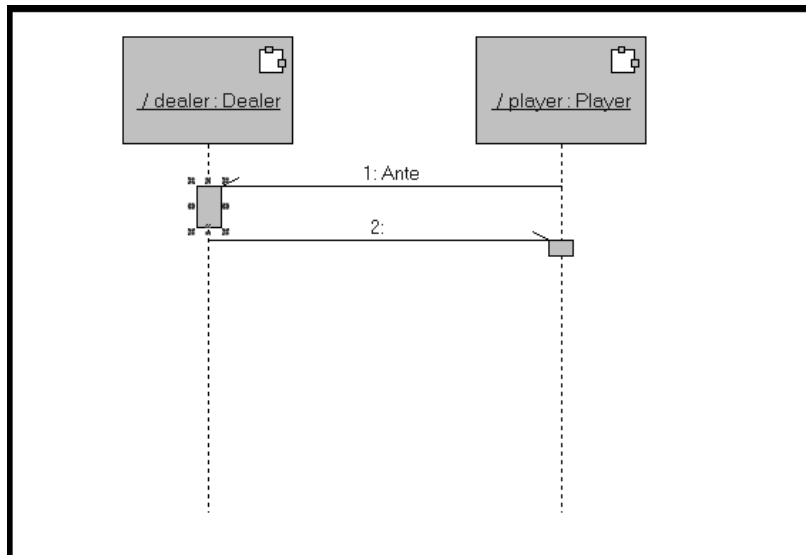


10 On the **General** tab, type **ACard** in the **Name** box.

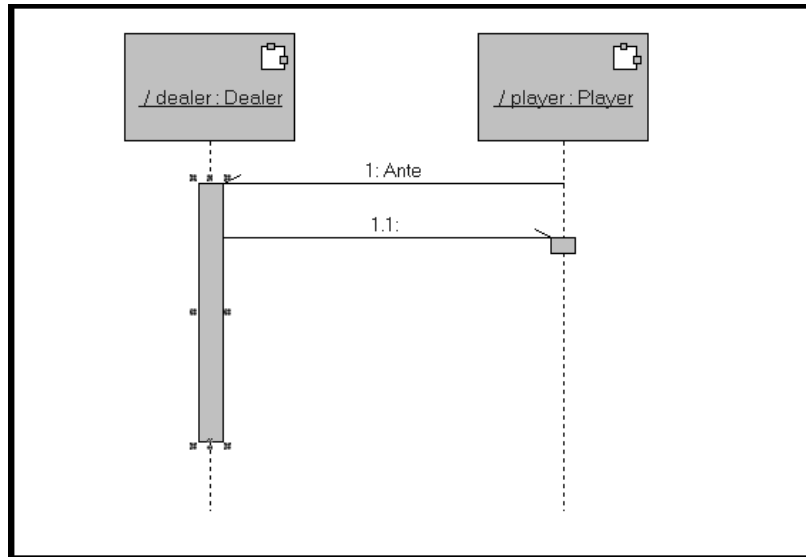
11 Click **OK**.

The Focus of Control (FOC) shows the period of time during which an object is performing an action, either directly or through an underlying procedure. The FOC is portrayed through narrow rectangles  that adorn lifelines (the dashed vertical lines descending from each object). The length of an FOC indicates the amount of time it takes for a message to be performed. When you move a message vertically, each dependent message moves vertically as well. To enlarge the FOC, select the FOC, click on a pic handle surrounding the rectangle, then drag the mouse until you change the rectangle to the desired size.

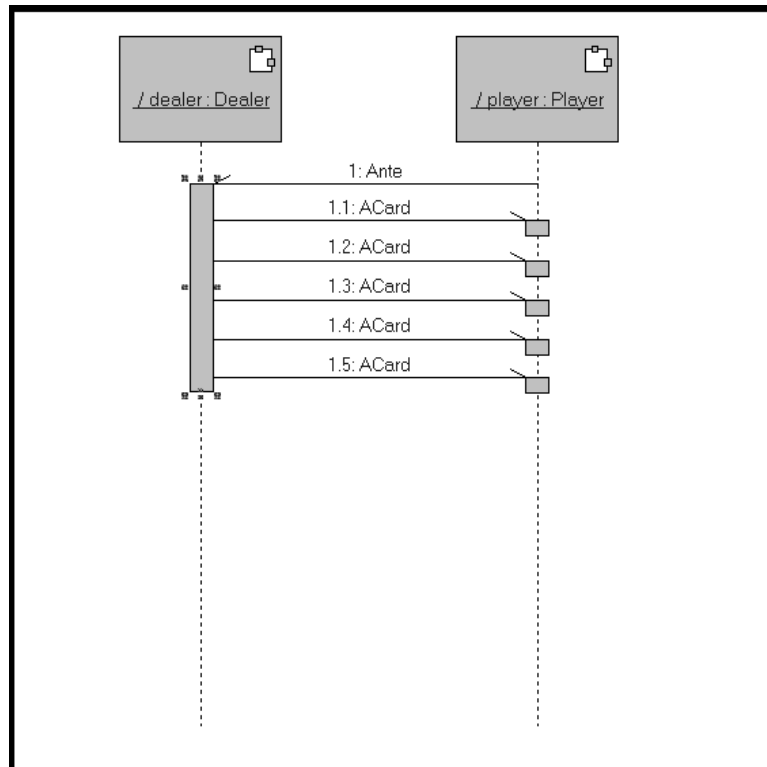
For example, you can select the FOC as in the following diagram:



Then enlarge it as in the following diagram:



12 Repeat steps 7 through 11 to add 4 additional **ACard** messages.



Next, you will create two additional messages; a message to inform the dealer role of the value of the player's hand (HandValue) and another message from the dealer role that informs the player as to whether or not they won the hand (Win).

13 Select the **Asynchronous Send Message**  tool from the toolbox.

14 Left-click on the player lifeline (dotted line beneath capsule role box), and drag the mouse to the dealer's lifeline.

Note: Resize the FOC from the last ACard message to include the sender handle for HandValue message.

15 Double-click on the message line (not the label for the line) to open the **Send Message Specification** dialog box.

16 On the **General** tab, type **HandValue** in the **Name** box.

17 Click **OK**.

18 Select the **Asynchronous Send Message**  tool from the toolbox.

19 Left-click on the dealer lifeline (dotted line beneath capsule role box), and drag the mouse to the player's lifeline.

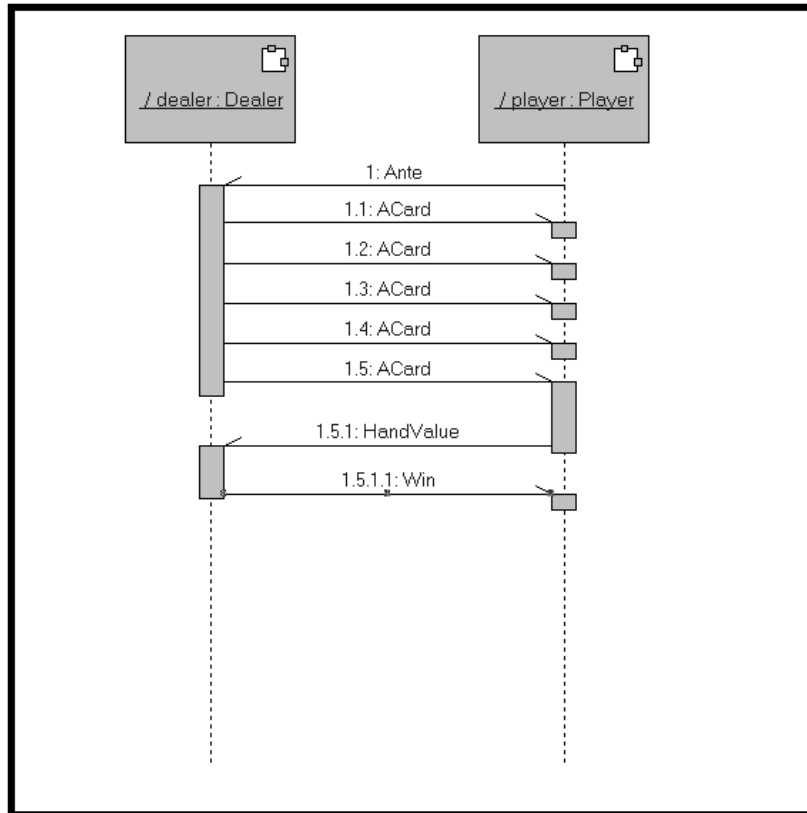
Note: Resize the FOC from the HandValue message to include the sender handle for Win message.

20 Double-click on the message line (not the label for the line) to open the **Send Message Specification** dialog box.

21 On the **General** tab, type **Win** in the **Name** box.

22 Click **OK**.

Your **Sequence Diagram** should look like the following:

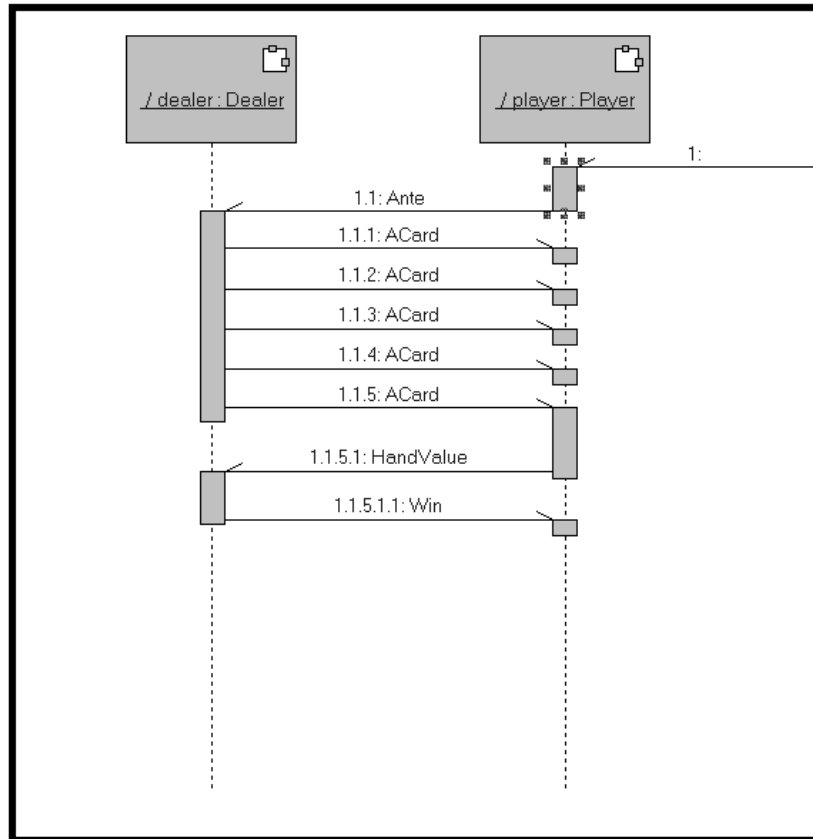


Note: After adding the message exchanged between the Player and Dealer you will realize that the scenario must be started somehow, either when the simulation is started (when the player capsule is instantiated) or after some other specified event occurs. For this tutorial, you will use a timer to start the simulation, this will make it possible to adjust the delay between each game and make it easier to observe the running model.

23 Select the **Asynchronous Send Message**  tool from the toolbox.

- 24 Left-click on the black border of the sequence diagram to the left of the player role, and drag the mouse to the player lifeline (dotted line beneath capsule role box).

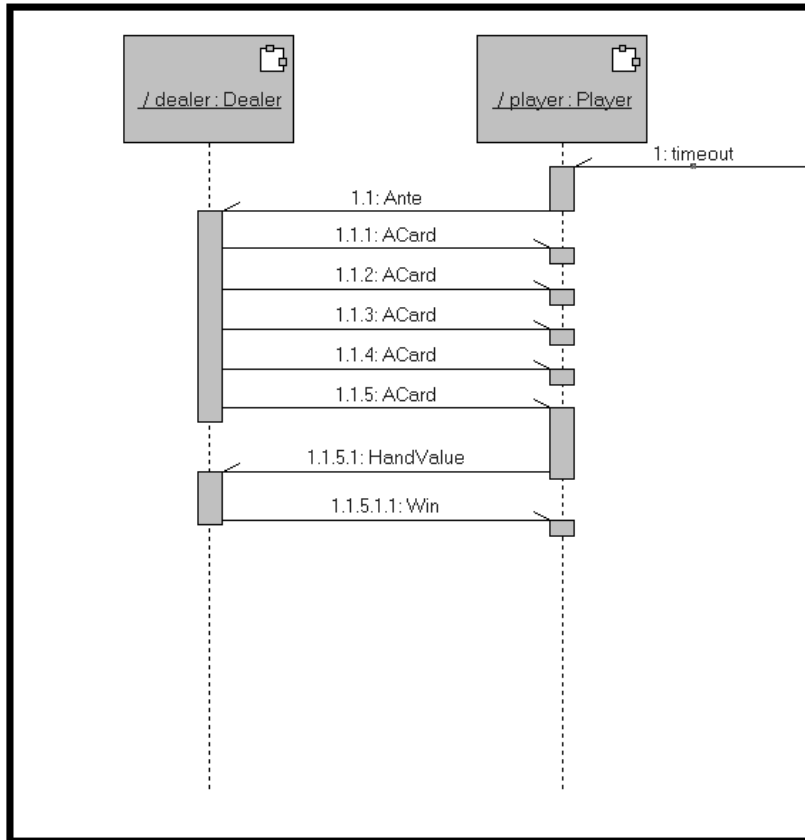
Note: Resize the FOC on the player lifeline to include the Ante message as in the following diagram.



Note: By adding this new message, the number labelling for the messages changed to one level deeper. For example, the message label 1:Ante changed to 1.1:Ante, and 1.5.1:HandValue changed to 1.1.5.1:HandValue.


- 25 Double-click on the message line (not the label for the line) to open the **Send Message Specification** dialog box.
- 26 On the **General** tab, type **timeout** in the **Name** box.
- 27 Click **OK**.


Your **Sequence Diagram** will look like the following.



Now, you may want to include some supporting text.

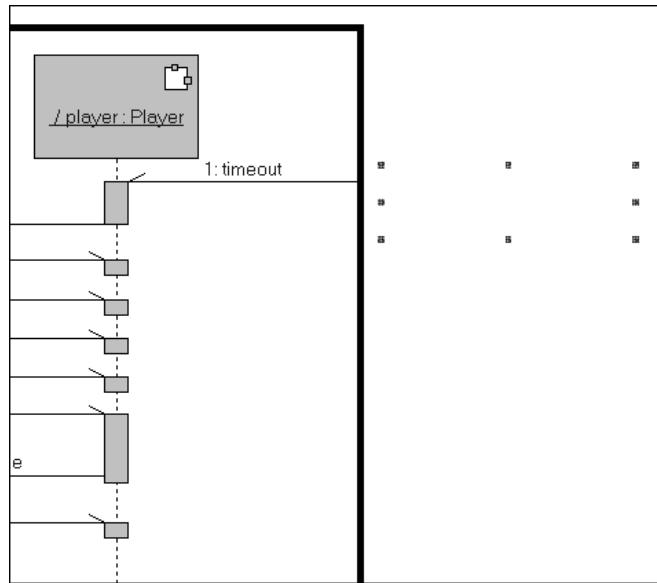
28 Ensure that the **Sequence Diagram: Logical View / HeadsUpPoker** window is the active window.

29 From the toolbox, select the **Text** tool .

As you move your mouse over the **Use Case Diagram**, your cursor changes to the text tool .

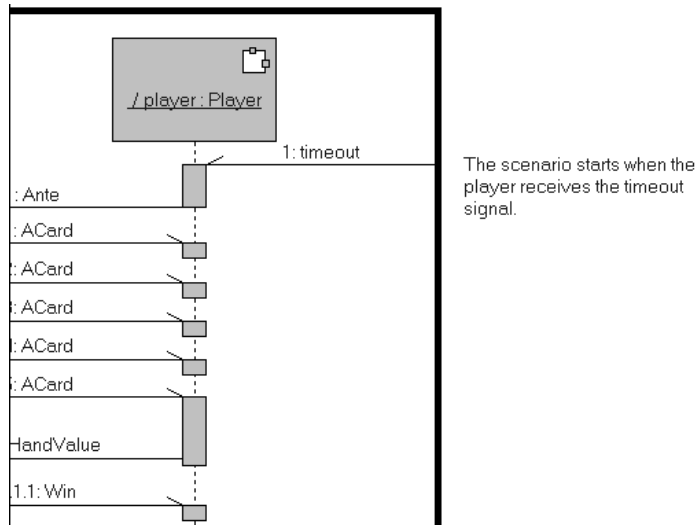
30 Left-click in the diagram, and drag (it will show as an outlined rectangle as you drag).

- 31 Release the mouse button after you have a rectangle similar to that in the following diagram (you can resize the rectangle later).

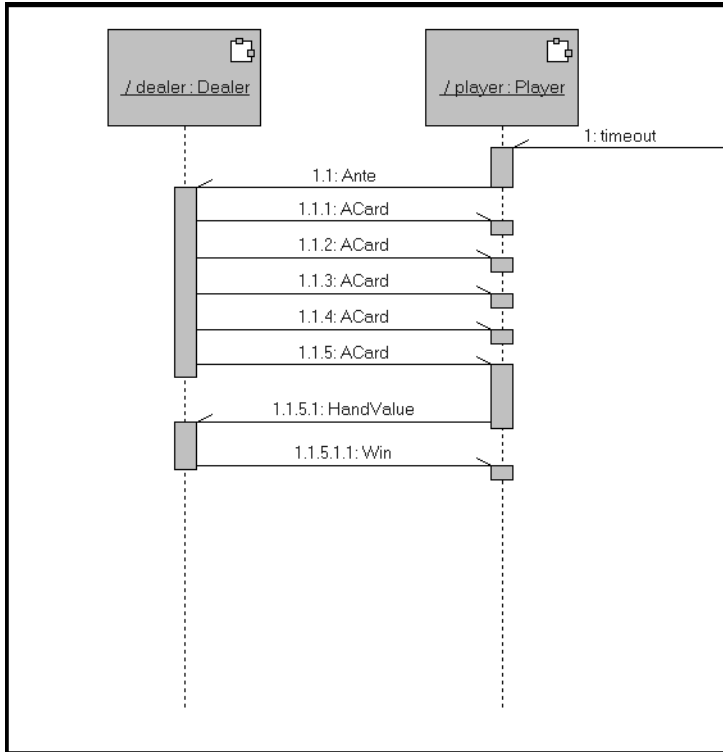


- 32 Type the following text:

The scenario starts when the player receives the timeout signal.



33 Create 3 more text boxes and include the text that appears in the following diagram.



The scenario starts when the player receives the timeout signal.

After the player places an ante, the dealer distributes the cards.

When the player receives all five cards, the player informs the dealer of his hand and the dealer notifies the player if they won.

This scenario repeats itself indefinitely.

Creating the Protocol

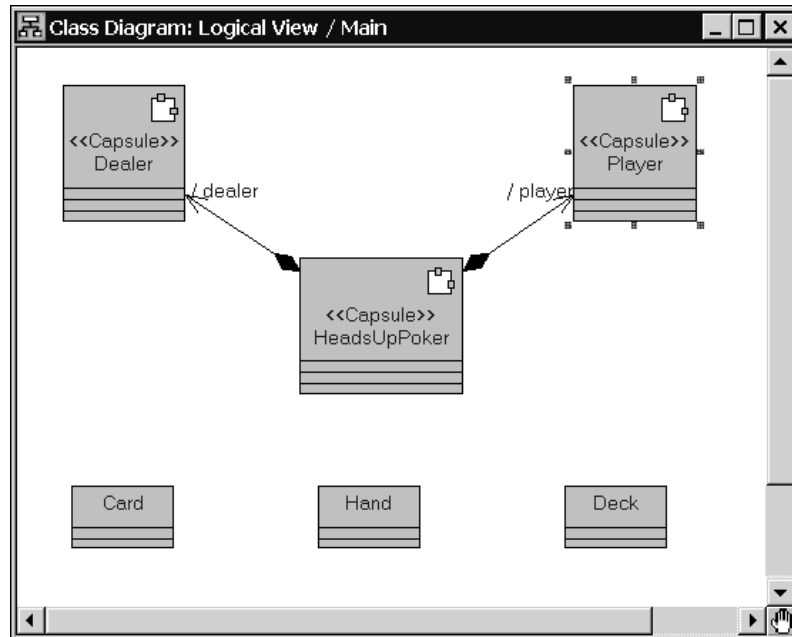
Now that you defined the classes and capsules in your design and specified the interactions, you can define the set of messages to exchange between the Dealer and Player capsules, and create ports through which they can communicate to each other.

Suggested Reading:

- Ports, *Rational Rose RealTime Modeling Language Guide*
- Protocols, *Rational Rose RealTime Modeling Language Guide*

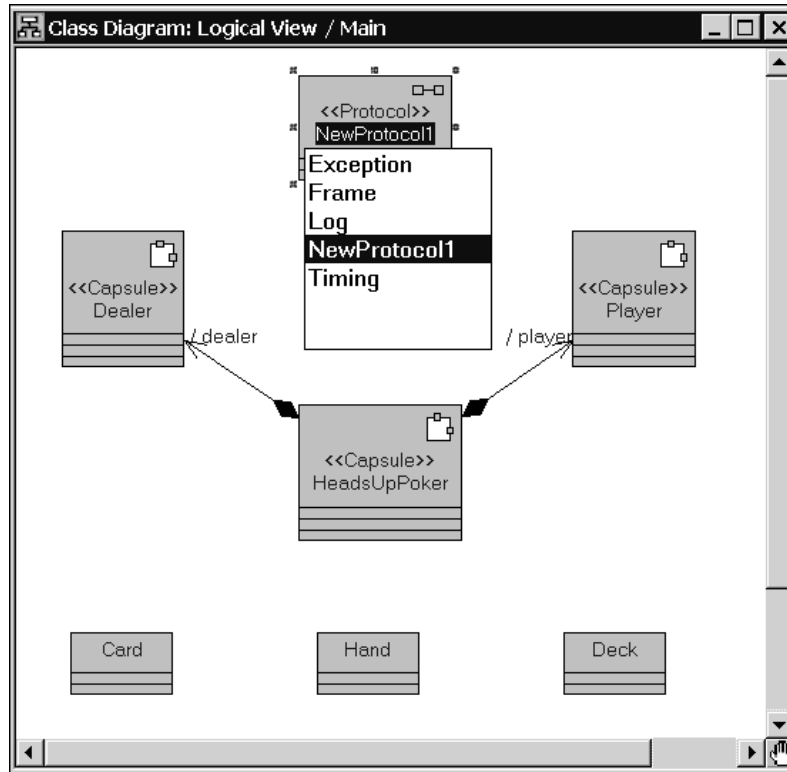
To create the communication protocol

- 1 Open the **Main** class diagram.



- 2 Select the **Protocol** tool from the toolbox  .

- 3 Create the protocol in the diagram in the same way that you created classes and capsules.



Note: The protocol was added to the **Logical View** folder in the **Model View** tab in the browser.

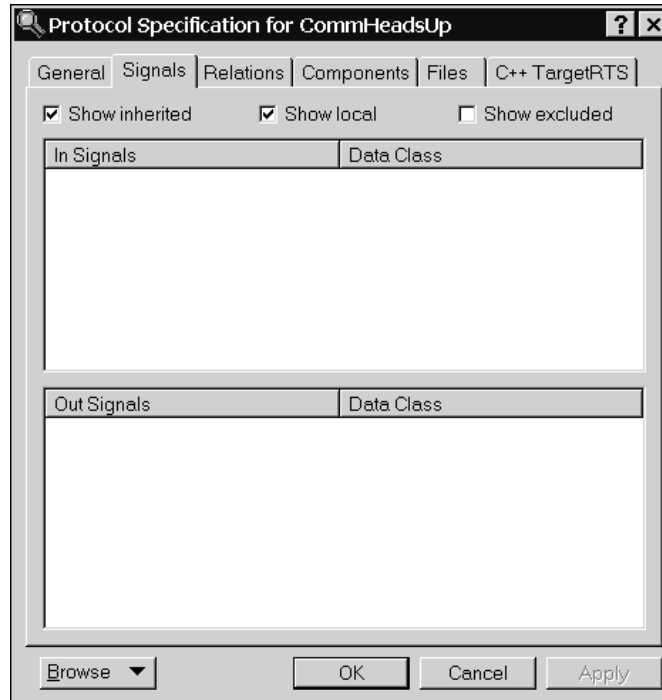
- 4 Type **CommHeadsUp** and press **ENTER**.

To create the set of signals for this protocol

- 1 Right-click on the protocol in the diagram, and click **Open Specification**.

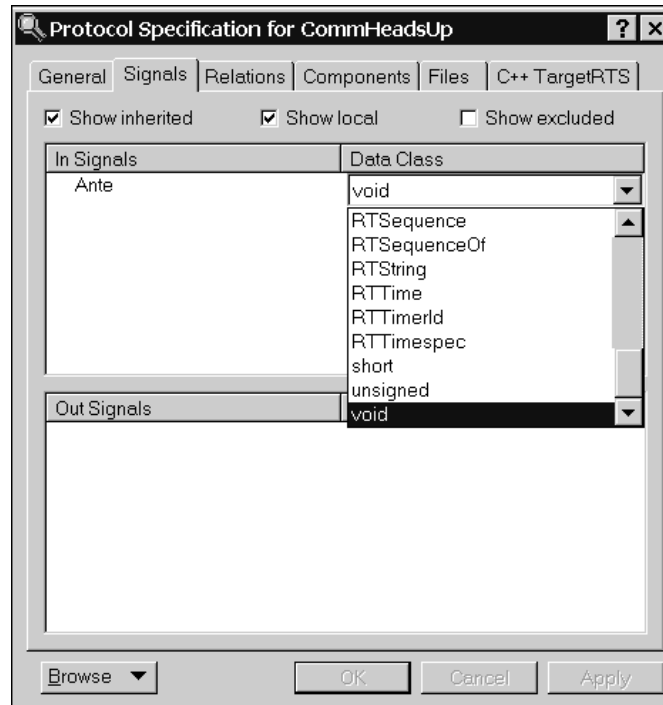
The **Protocol Specification for CommHeadsUp** dialog box appears.

- 2 Click the **Signals** tab.



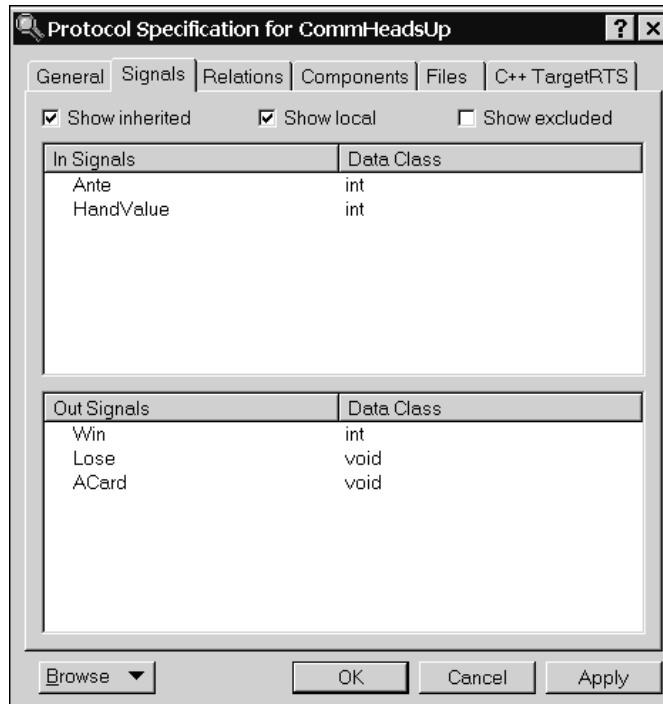
- 3 Right-click in the **In Signals** pane, and click **Insert**.
- 4 Rename the signal **Ante** and press ENTER.
- 5 In the **In Signals** box, click **void** in the **Data Class** column, and press F8.

A drop down list appears with a list of all the basic types and classes in the model.



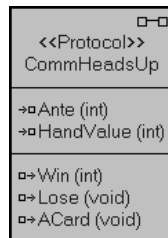
6 Select `int` from the list.

- Repeat the steps above to add the following **In Signals** and **Out Signals** to the protocol:



- Click **OK**.

The protocol on the **Class Diagram: Logical View / Main** changes to include these new **In** and **Out signals**.



Note: Later, you will add more details to this protocol as the model evolves. For example, you have not yet specified which data class to send with the **ACard** signal.

Creating Ports and Connectors

You will create the **player_comm** and **dealer_comm** ports based on the **CommHeadsUp** protocol. You can connect the player and dealer capsule roles to allow them to communicate the signals defined in the protocol.

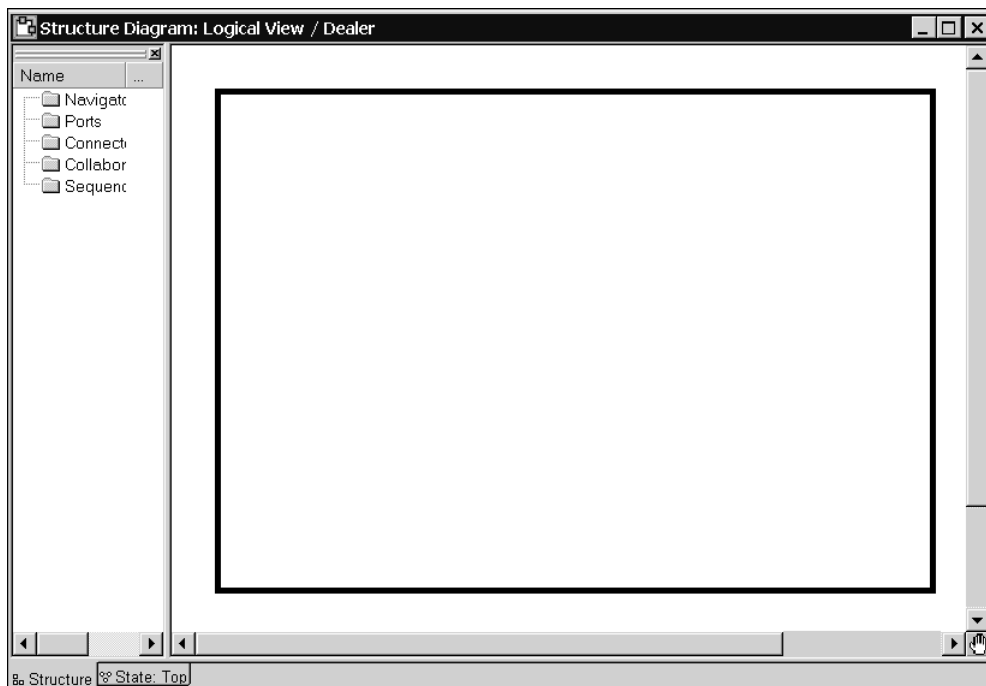
Ports are attributes of a capsule, and as such, all capsule roles of the same capsule have the same ports available. It is a common mistake to think that you should add ports to specific capsule roles.

Suggested Reading:

- Ports, *Rational Rose RealTime Modeling Language Guide*
- Connectors, *Rational Rose RealTime Modeling Language Guide*

To create ports

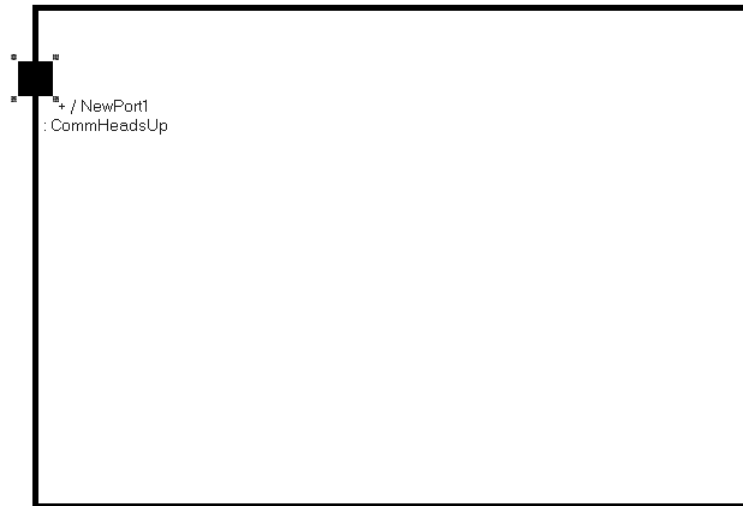
- 1 Open the **Structure Diagram** for the **Dealer** capsule.



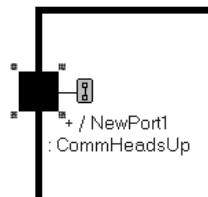
Note: You can double-click on a capsule role to open the capsule structure diagram of the capsule. This capsule role represents in the structure diagram.



- 2 From the **Model View** tab in the browser, drag the **CommHeadsUp** protocol to the black border of the **Dealer's** structure diagram.

When the mouse is near the border, a dotted rectangle snaps to the border automatically. This creates a public port that is visible outside of the capsule. Release the mouse button when the dotted rectangle appears. The **Structure Diagram** will look like the following.

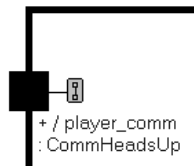


- 3 Right-click the new port, and click **End Port**.



The port graphic changes from  to  to indicate that it is an **End Port**. End ports provide a connection between the behavior of the capsule containing the end port and the outside world. To send and receive messages, a capsule must have end ports. The end port's protocol defines the set of messages that can be sent.

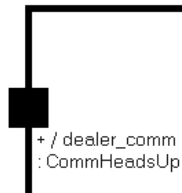
- 4 Rename the port **player_comm**.



Now, you will create a port for the **Player** capsule.

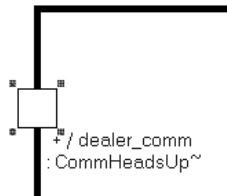
- 5 Open the **Structure Diagram** for the **Player** capsule.

- 6 From the **Model View** tab in the browser, drag the **CommHeadsUp** protocol to the black border of the **Player**'s structure diagram.
- 7 Right-click the new port, and click **End Port**.
- 8 Name the port on the player capsule **dealer_comm**.

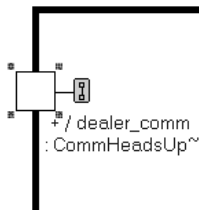


To enable communication between capsules, you must connect together the ports on their interfaces. You can only connect compatible ports together. For a port to be compatible, the out signals on each side must be a subset of the in signals on the other side.

- 9 Right-click on the **dealer_comm** port in the **Structure Diagram** for the **Player** capsule, and click **Conjugate**.



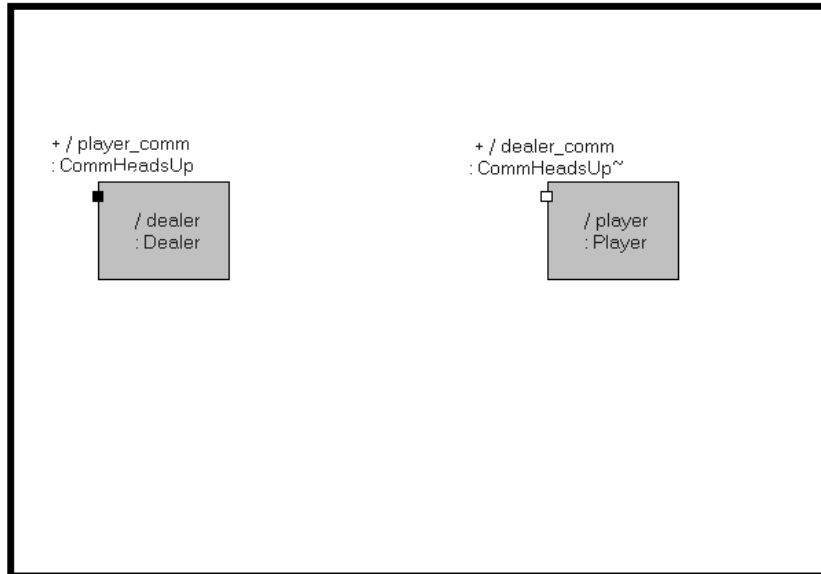
- 10 Right-click on the **dealer_comm** port in the **Structure Diagram** for the **Player** capsule, and click **Wired** if it not currently selected.





To connect the dealer and player capsule roles

- 1 From the **Model View** tab in the browser, open the **Structure Diagram** for the **HeadsUpPoker** capsule.


The new public ports appear on both capsule roles.



Now, you will connect these ports.

- 2 From the **Structure Diagram** toolbox, click the **Connector** tool .
- 3 Position the mouse over the port  on the **Dealer** capsule role.

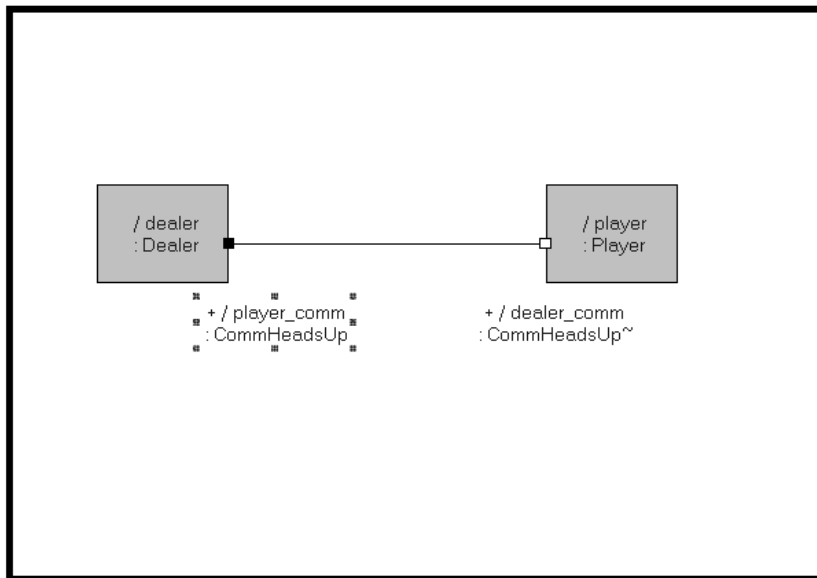
The cursor changes to .

- 4 Drag the mouse over to the conjugated port  on the **Player** capsule role, and release the mouse.

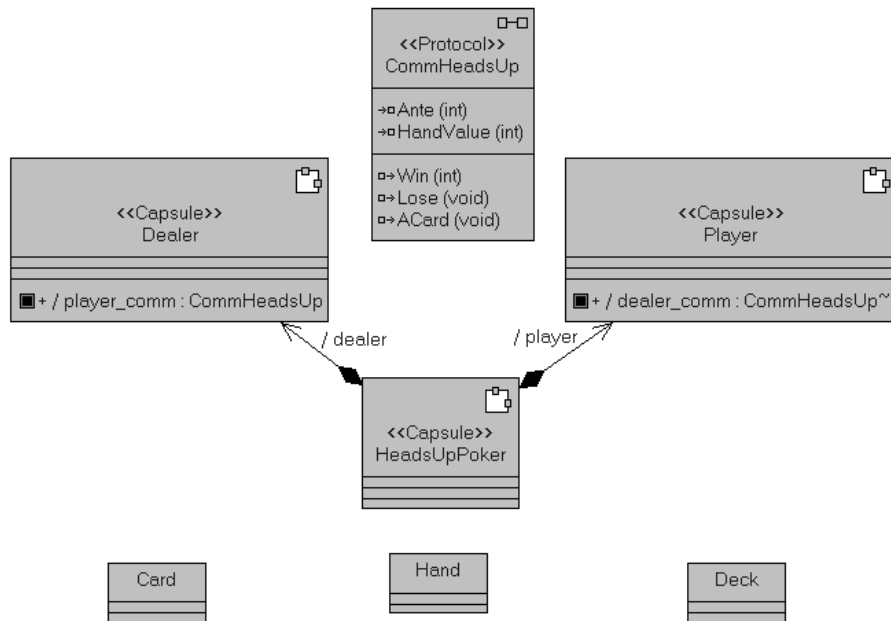
A connector appears between both capsule roles.



To make the graphic easier to view, you can move the ports along the border of the capsule roles to look like the following.



By creating these ports and connectors, the **Class Diagram: Logical View / Main** changes to include the ports on the **Dealer** and **Player** capsules and the **HeadsUpPoker** capsule.



Documenting the Responsibilities

It is good practice to include a brief responsibility statement for each element in the design that describes what is included in the data, and the behavior of each component. Every element in a Rose RealTime model can include documentation.

To add documentation to model elements:

- 1 In the **Model View** tab in the browser, either right-click to open the **Specification** dialog boxes for the following capsule and class elements, or select the element, then begin typing in the **Documentation** box in the **Documentation** window.

The elements and descriptive text is as follows:

Model element	Type	Documentation text
HeadsUpPoker	capsule	Encapsulates the flow of events described in the HeadsUpPoker:MainFlow use case.
Player	capsule	Interfaces with the dealer. Places an ante. Receives cards. Tells the dealer the value of the player hand.
Dealer	capsule	Interfaces with a player. Shuffles the card deck and distributes the cards. Decides who wins, and informs the player if they win or lose.
CommHeadsUp	protocol	Set of signals exchanged between the player and dealer in a HeadsUp poker game.

Before proceeding with Lesson 4, we recommend that you save your work.

Lesson 4: Building and Running

In this lesson, you will learn how to build and run a model.

Prototyping

One of the fundamental impacts Rational Rose RealTime is that it encourages a highly iterative development workflow at the individual software developer level. An iterative approach:

- helps to resolve major risks before making large investments
- enables early user feedback
- makes testing and integration continuous

This iterative workflow allows developers to create prototypes, or executable versions of the system, regularly. Creating executable versions of a system early on has the following advantages:

- Allows you to validate and test the design.
- Gives you something to play with, more tangible than just boxes and diagrams.
- Helps you discover missing requirements.
- Allows you to build a working demonstration for feedback from the user.

Instead of waiting until the model is complete, you can construct a high-level model with sufficient details to validate your design approach. You can execute the model, find and fix problems, and revise the model to add more detail or to implement additional behavior.

By validating the design early, and often, you reduce the risk of not delivering the project on time. Errors at the design level are more costly to fix than implementation errors, so finding and fixing these errors early in your process will save time and effort.

Note: *This highly iterative approach to development is the key to making the most of Rational Rose RealTime.*

Building a Model

You will build and run for the first time an executable version of the card game simulation. Although you have not yet added any detailed behavior, this is a good time to build and run a model.

Suggested Reading

- Components, *Rational Rose RealTime Modeling Language Guide*
- Build basics, *Rational Rose RealTime Toolset Guide*
- Building and running models, *Rational Rose RealTime Toolset Guide*

To run your model, you need to build it, and then execute it on a processor. A component describes how to build a set of capsules and classes. The deployment of a component describes on what processor to execute the built component.

Note: A component can have instances. An instance of a component can be a single executable that can reside on a number of different nodes. To allow for this, specific component instances can be assigned to processors. Component instances are not shown in the component diagram, they appear in the deployment diagram.

You will create a component by opening a series of dialog boxes so that you can become familiar with the common features of the Rose RealTime user interface.

Note: You can also use the **Component Wizard** to quickly create, modify, and deploy a component. The wizard guides you through creating and configuring a component, and running the component instance. To access the wizard, click **Component Wizard** on the **Build** menu.

Creating a Component

The **Component View** specifies how to compile various parts of the model. The primary element of the **Component View** is a component that you need to create. This component specifies the capsules and classes to compile, how to compile those elements, and the inclusions and libraries to incorporate into the build.

You must create a component for the top-level capsule in order to build and execute your model. You can draw component diagrams for situations where you have many related components or packages of components.

To create a component:

- 1 In the **Model View** browser, right-click **Component View**, and click **New > Component**.

Or . . .

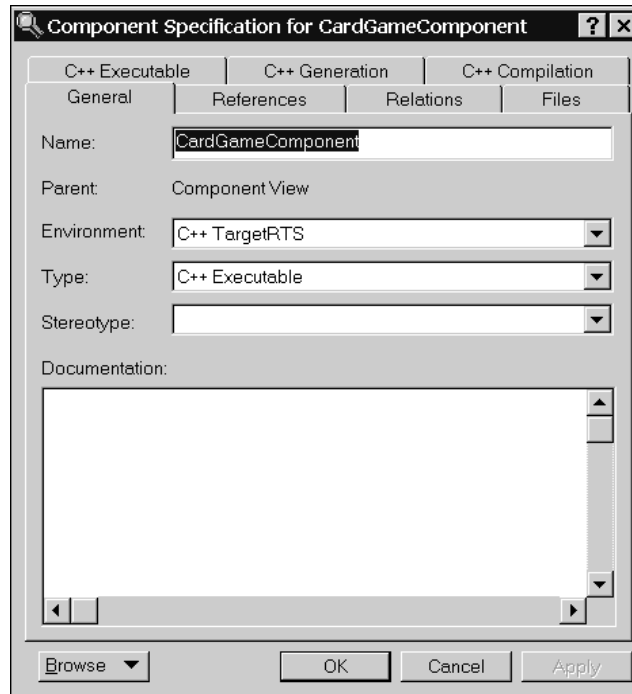
In the **Main** diagram for the **Component View**, from the toolbox, select the **Component** tool , then click on the **Main** diagram.

A component appears with the default name **NewComponent1**.



- 2 Rename the component **CardGameComponent**.

- 3 In the **Model View** tab in the browser, double-click **CardGameComponent** to open the **Component Specification for CardGameComponent** dialog box.
- 4 Click the **General** tab if not already selected.



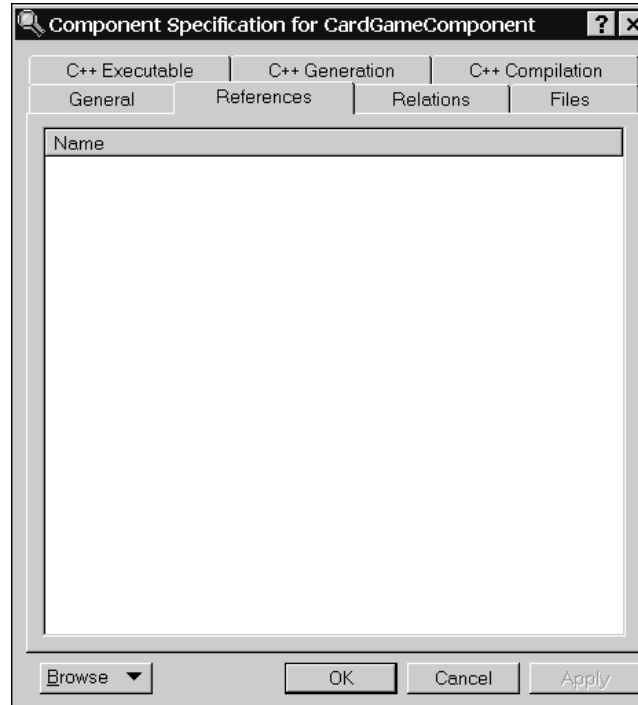
- 5 In the **Environment** box, select **C++ TargetRTS** if not already selected.

Setting the **Environment** box to C++ TargetRTS specifies that the C++ run-time system and code generation are used in the build process.

- 6 In the **Type** box, select **C++ Executable** if not already selected.

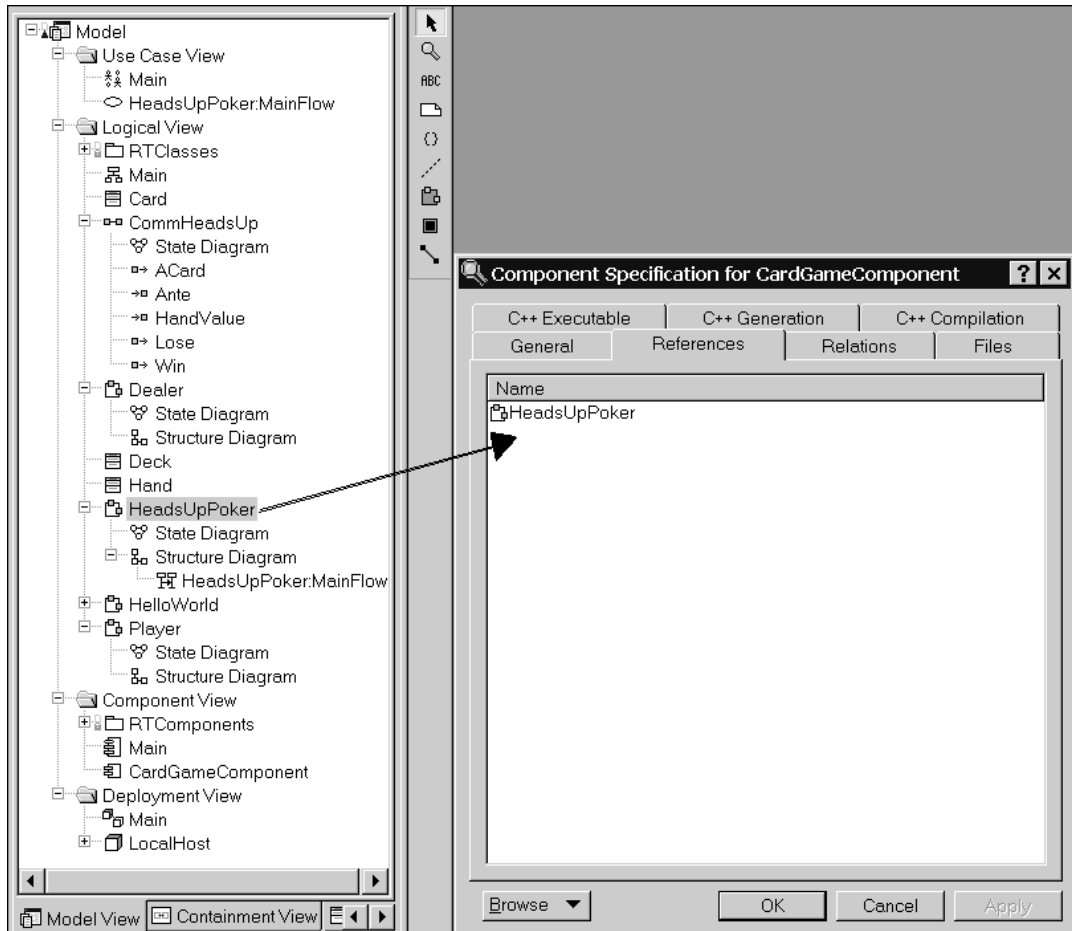
Setting the **Type** box specifies that you want to build a C++ executable version of the model.

- 7 Click the **References** tab.



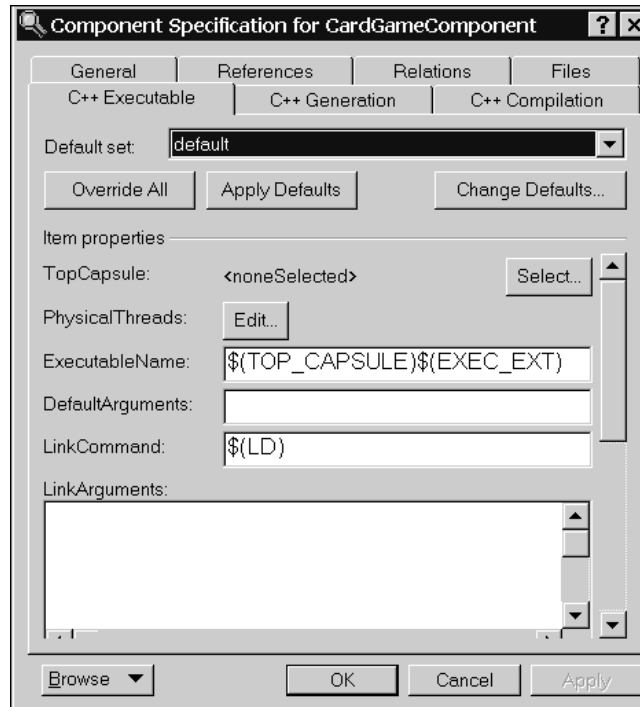
- 8 In the **Model View** tab in the browser, drag the **HeadsUpPoker** capsule onto the **References** tab.

The **HeadsUpPoker** capsule appears in the window.



You drag the **HeadsUpPoker** capsule onto the **References** tab because the items in the **References** tab identify what is compiled with the **HeadsUpPoker** component.

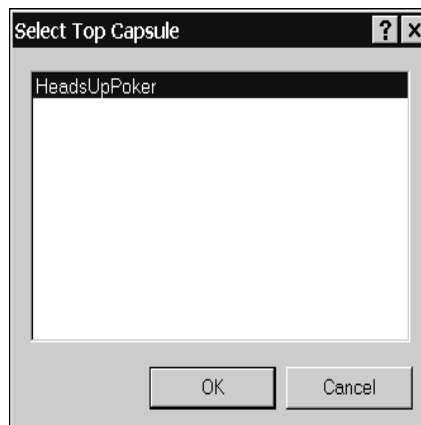
9 Click the **C++ Executable** tab.



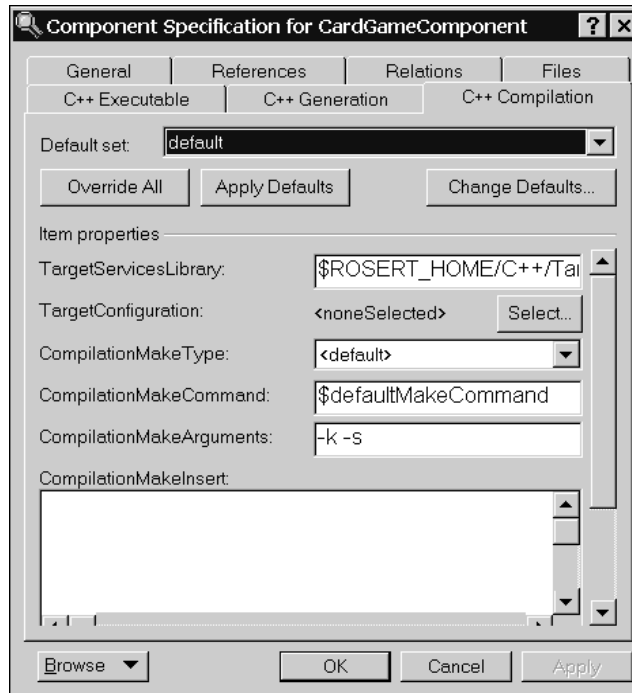
10 Click **Select...**

The **Select Top Capsule** dialog box appears. In this dialog box, you select the capsule that will be the top capsule in the model.

11 Click **HeadsUpPoker** to designate it as the top capsule.

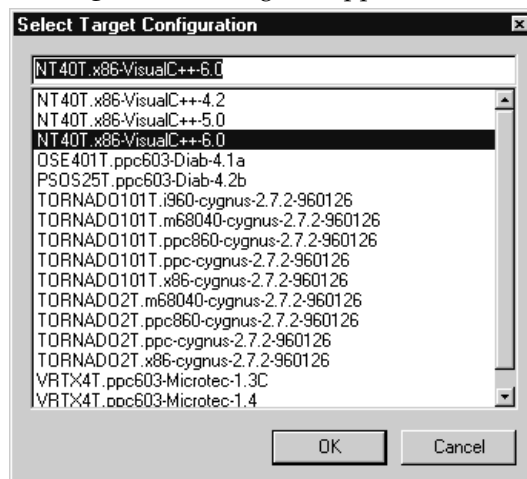


- 12 Click **OK**.
- 13 Click **Apply**.
- 14 Click the **C++ Compilation** tab.



- 15 Click **Select...**

The **Select Target Configuration** dialog box appears.



A component is always created with a default configuration for your host machine. This includes a default compiler, compiler flags, linker, and so forth. In many cases, these settings are sufficient for building simple sets of classes and capsules that do not require integration with external source files, or libraries.

In this dialog box, you will specify the operating system, compiler, and processor that you want to use to build and run the model.

The information is listed in the following format:

```
<operating system>.<processor and compiler>
```

For example:

- If you are running Windows 4.X on a x86 processor, and you have installed the Visual C++ 6.0 build tools, select:

```
NT40T.x86-VisualC++-6.0
```

- If you are running Solaris 5.X on a sparc processor, and you have installed the gnu 2.8.1 build tools, select:

```
SUN5T.sparc-gnu-2.8.1
```

16 Select the configuration for your computer, and click **OK**.

17 To save the changes, click **Apply**.

18 To close the **Component Specification for CardGameComponent** dialog box, click **OK**.

Creating the Deployment View


The **Deployment View** describes the computing environment in which your model is executed, and specifies how it is deployed within the environment. The most important elements of the **Deployment View** are processors and component instances.

First, you define a processor on the **Deployment Diagram** that describes the computing hardware on which the model will be executed. Next, you map the component onto the processor to create a component instance. Finally, you run the model.

To create a processor

- 1 In the **Model View** tab in the browser, right-click **Deployment View**, and click **New > Processor**.

Or . . .

In the **Deployment View**, open the **Main** diagram, from the toolbox, select the **Processor** tool , then click on the diagram to add a processor.

A processor appears with the default name **NewProcessor**.

- 2 Rename the processor **LocalHost** and press ENTER.
- 3 In the **Model View** tab in the browser, drag the **CardGameComponent** from the **Component View** folder onto **LocalHost**.

A component instance, **CardGameComponentInstance**, is created on the processor, and appears under **LocalHost**.



Now that you associated a component with a processor (creating a component instance), your component instance is complete and you can now run **CardGameComponentInstance**.

- 4 Before you run the component instance, save your model (**File > Save Model**).

Starting the Build

Now that you created the component (**CardGameComponent**), processor (**LocalHost**), and component instance, you can build and run the executable version of the model.


Suggested Reading

- Build menu, *Rational Rose RealTime Toolset Guide*
- Build settings dialog, *Rational Rose RealTime Toolset Guide*
- Build log, *Rational Rose RealTime Toolset Guide*
- Starting a build, *Rational Rose RealTime Toolset Guide*
- Common build errors, *Rational Rose RealTime Toolset Guide*

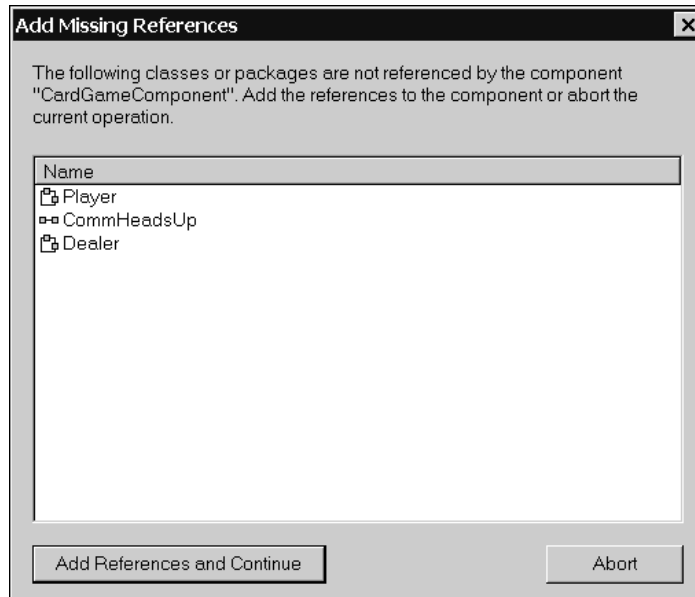
To build the CardGameComponent

- 1 In the **Model View** tab in the browser, right-click **CardGameComponent**, and click **Set As Active**.

Because you will be building and running the same component and component instances often, you should configure an active component. Setting the **Set As Active** option ensures that the toolbar build icons and menu items, for the common run and build commands, become available for easy access.

The **Build Component** tool  becomes active.

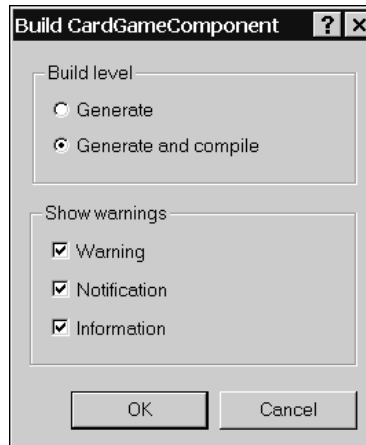
- 2 Click **Build Component**.



Note: The **CardGameComponent** component should contain all referenced classes before it is compiled. Rose RealTime checks the references, and prompts you to add any missing references that it detected.

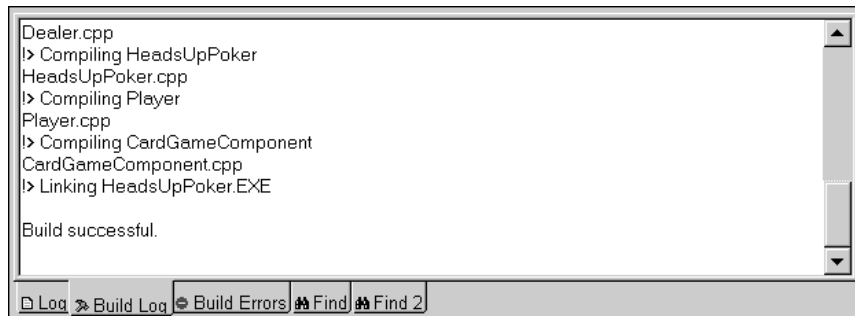
- 3 Click **Add References and Continue**.

The **Build CardGameComponent** dialog box appears.



4 If not currently selected, click **Generate and compile**, and click **OK**.

The **Build Log** tab of the **Output Window** shows the results of code generation and compilation. When the build finishes, the **Build Log** should indicate "Build successful".



Note: If there are compile errors, double-click on an error message on the **Build Errors** tab. Rational Rose RealTime opens the appropriate editor where the source of the error appears. You can then resolve any errors.

If you do not see "Build Successful", review the topic on Common build errors. The most common error is not having one of the supported compilers for your platform installed or accessible in your path. Also, check the Rational Rose RealTime product web site for any known issues, or updates to tutorials and model examples.

Where is the Source Code Generated?

If you wish to examine the generated code, look in the directory where you saved your model. In that directory, you should see a subdirectory labelled **CardGameComponent**. In this directory, you will see another subdirectory called **src**. This directory contains the generated code for your model. You may want to explore and examine the files in this directory.

Running the Component Instance

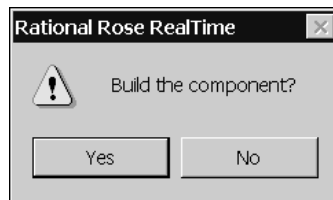
Now, you will run the built component.

Suggested Reading

- Building and running models, *Rational Rose RealTime Toolset Guide*

To run the component instance:

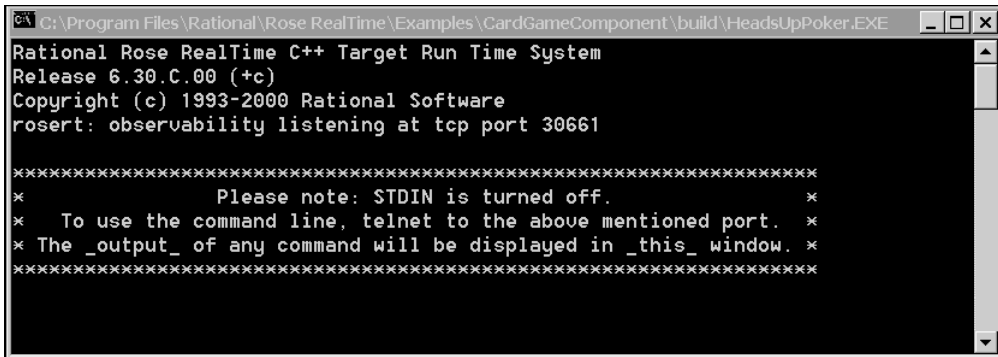
- 1 From the **Model View** tab in the browser, right-click **CardGameComponentInstance**, and click **Run**.



- 2 When prompted to rebuild the component, click **No**. You already built the component in an earlier step.

The **Build Log** appears while dependencies are recalculated. If the toolset determines that model elements changed since the last build the component is rebuilt before the component instance is run.

A console window appears showing output to stderr or stdout.

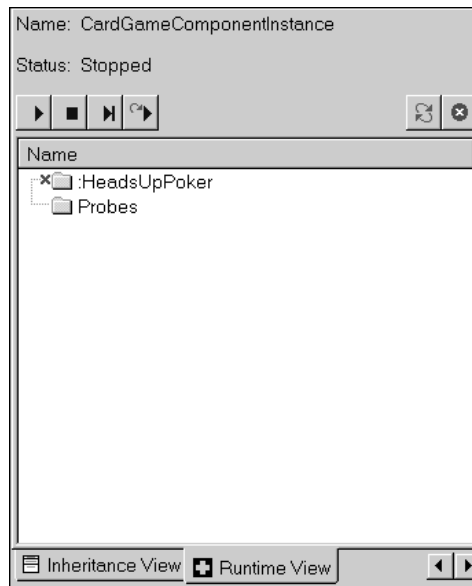


```
C:\Program Files\Rational\Rose RealTime\Examples\CardGameComponent\build\HeadsUpPoker.EXE
Rational Rose RealTime C++ Target Run Time System
Release 6.30.C.00 (+c)
Copyright (c) 1993-2000 Rational Software
rosert: observability listening at tcp port 30661

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*
*           Please note: STDIN is turned off.           *
* To use the command line, telnet to the above mentioned port. *
* The _output_ of any command will be displayed in _this_ window. *
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```


- 3 Click on any part of the toolset to make it active.

A new browser, **RTS Browser (Runtime View)**, appears on your **Model** browser. It controls the execution of a running component instance. You can run and control multiple component instances from within Rose RealTime. There is a separate **RTS Browser (Runtime View)** tab for each running instance.

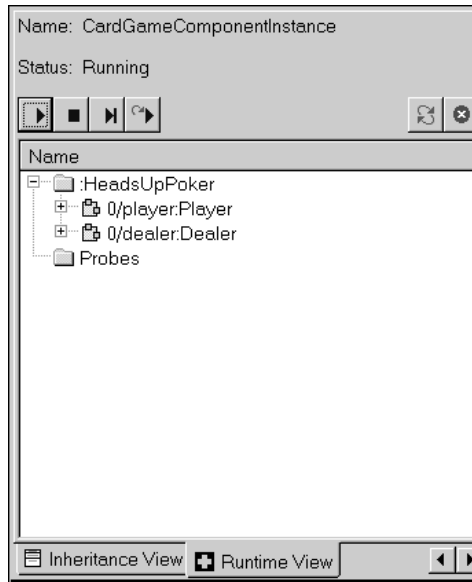


- 4 Click on the **Runtime View** tab.

The **RTS Browser** appears. The top folder shows the name of the capsule, **HeadsUpPoker**, which encapsulates the card game simulation.

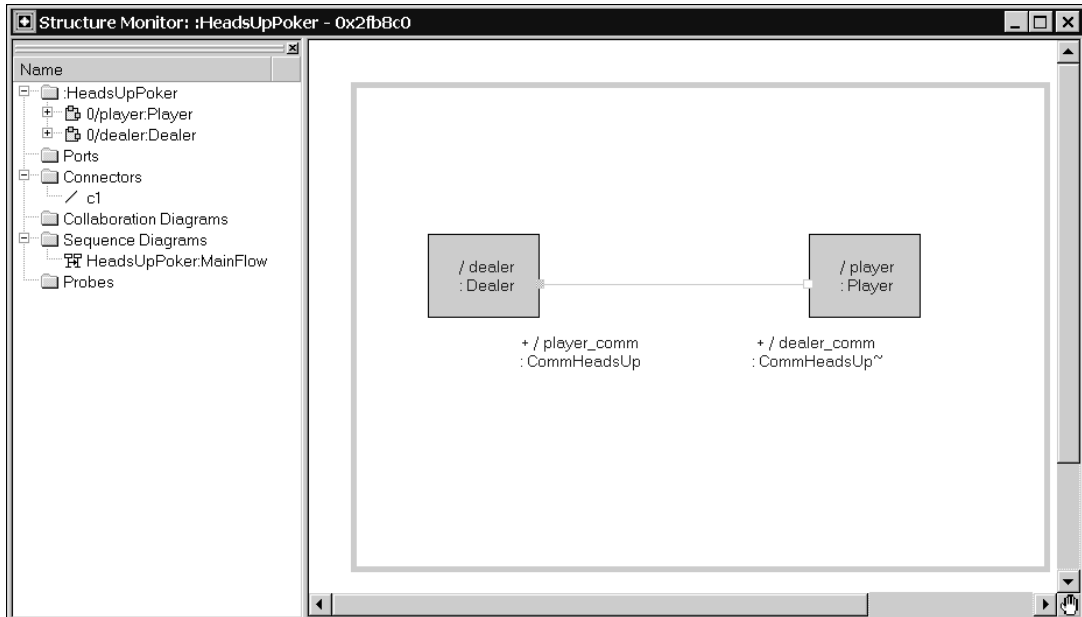
- 5 Click the **Start** button  to start the execution of the loaded component instance.

The player and dealer capsule instances display in the browser.




- 6 Right-click on the **HeadsUpPoker** capsule instance, and click **Open Structure Monitor**.

The following diagram shows a run-time view of the structure of this capsule instance.



In the next lesson, you will add detailed behavior to your card game model. Because there is no behavior in your existing model, you will stop the component instance.

- 7 To stop the component instance from running, click the **Shutdown** button .

Review

You have now completed the following activities:

- Described the requirements of the simulation in a use case.
- Discovered from the requirements the initial design objects needed to implement the simulation: capsules and classes.
- Described the communication paths and scenarios between capsules using sequence and structure diagrams.
- Created the protocols which describe the sets of signals which are exchanged between capsules.
- Build and run the model.

You used Rational Rose RealTime to describe and develop the high-level Design of a card game simulation in a way that allows others to understand the system you are building.

The key parts of a design consist of:

- The name of the key components in the system (viewable from the **Model View** tab in the browser).
- The main responsibility of each component (captured in the **Documentation** box).
- The communication patterns between the components (captured in Structure and Sequence diagrams).

Next, you will incrementally add details to the simulation, first adding behavior to the capsules, and then implementing the required card classes.

Lesson 5: Adding Behavior to the Capsules

You created the major classes and capsules that make up the card game simulation, and determined how they are connected and communicate. However, the capsules and classes do not have any behavior, and do not perform a function. You will add that behavior in this lesson.

First, you will add behavior to the capsules to run the simulation to test the design. Then, you will implement the card classes to complete the simulation.

Suggested Reading

- Creating capsule state diagrams, *Rational Rose RealTime Toolset Guide*
- State diagrams, *Rational Rose RealTime Modeling Language Guide*
- Signal events, *Rational Rose RealTime Modeling Language Guide*

Opening Capsule State Diagrams

Capsule behavior is described in a state diagram. You do not have to create a State Diagram for a capsule, it is automatically created for each capsule.

To open a state diagram:

- 1 In the **Model View** tab in the browser, for the **Dealer** capsule in the **Logical View**, double-click on **State Diagram** to open the state diagram.



In the **State Diagram** editor, the state diagram is in the right pane and the **Navigator** in the left. The **Navigator** allows you to quickly access and browse capsule behavior.

Creating the Dealer's Behavior

In this topic, you will create states, transitions, triggers, and action code to implement the first pass behavior for the **Dealer** capsule.


To determine the behavior of a capsule, review the Responsibility description of the capsule, and the Sequence diagram in which the capsule participates. These two sources indicate the messages that the capsule receives, and sends.

For example, in the **Dealer** capsule's documentation and role in the **HeadsUpPoker:Mainflow** sequence diagram, you can see that the behavior has to handle the receipt of the **Ante** and **HandValue** signals, and send out the **ACard**, **Win**, **Lose** signals.

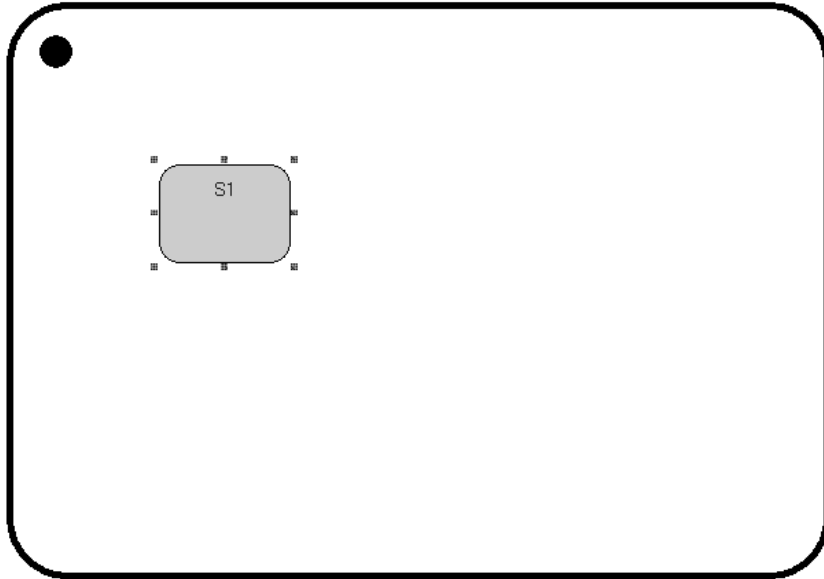
Suggested Reading

- Creating capsule state diagrams, *Rational Rose RealTime Modeling Language Guide*
- TopState, *Rational Rose RealTime Modeling Language Guide*
- Transitions, *Rational Rose RealTime Modeling Language Guide*
- States, *Rational Rose RealTime Modeling Language Guide*

To create the Dealer states:

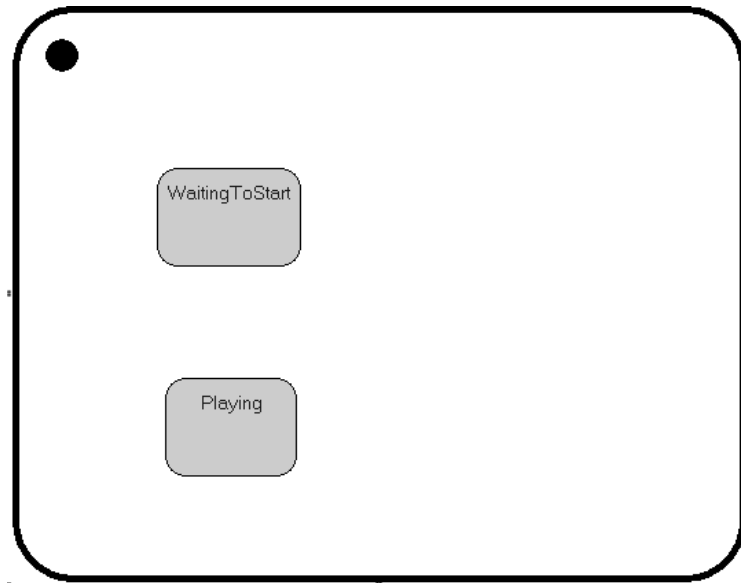
- 1 If not currently open, open the **State Diagram** for the **Dealer** capsule.
- 2 Click the **State** tool  from the **State Diagram** toolbox.
- 3 Move the mouse over the **State Diagram** and within the state diagram border.
- 4 Click to create a state.

The state name is selected.



- 5 Rename the state to **WaitingToStart**.
- 6 Repeat the steps above, and create a state called **Playing**.

Your diagram should look like the following diagram.





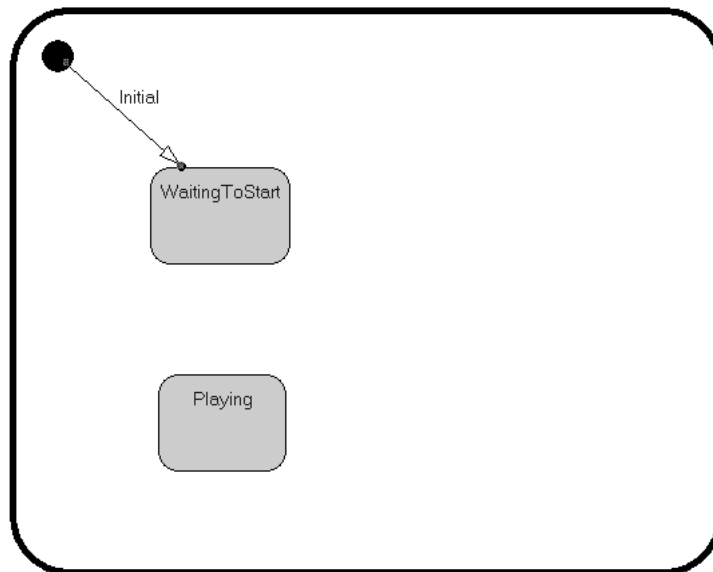
Next, you want to create transitions. The state diagram contains an *initial point*, ●, and an *initial state* called **WaitingToStart**. An initial point is a special point which explicitly shows the beginning of the state machine. You connect the initial point to a start state (in this case, **WaitingToStart**). Where the start state will be the first active state in the Dealer objects state machine. The transition from the initial point to the start state, the initial transition, is the first transition taken before any other transition. Only one initial state is allowed in each state diagram. Only one outgoing transition can exist from the initial point.

There can be several incoming transitions to the initial state. In this case the initial state acts like a junction point which forces the behavior back through the initial transition. If the initial transition is used to completely initialize an object, then any incoming transition to the initial state will effectively reset the behavior of an object without having to destroy then re-create it.

A transition is a relationship between two states, a source state and a destination state. It specifies that when an object in the source state receives a specified event and certain conditions are met, the behavior moves from the source state to the destination state. You create an **Initial Transition** that is automatically invoked at runtime when a capsule instance is created. Any action code associated with the **Initial Transition** runs when the capsule instance is created.

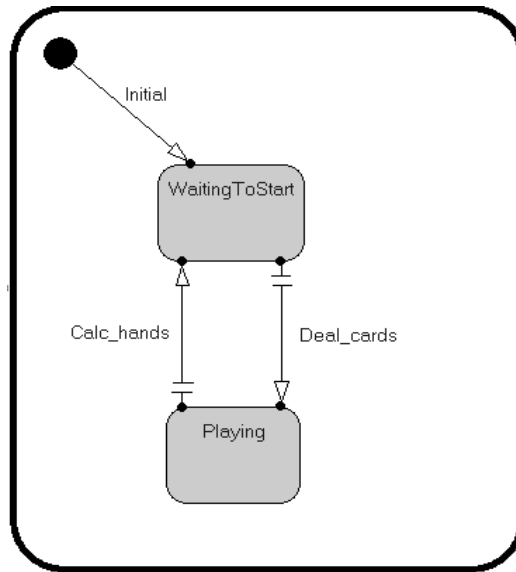
To create the transitions

- 1 From the **State Diagram** toolbox, click the **State Transition** tool .
- 2 Click and hold the left mouse button on the **Initial Point** in the state diagram, .
The **Initial Point** is the black circle that appears in the top-left corner of the **State Diagram**.
- 3 Drag the **State Transition** tool to the top of the **WaitingToStart** state.
The **Initial Transition** has a default name of **Initial**.



Note: You can resize the black border of the State Diagram by selecting it, then click and hold your mouse over a pic handle and change the size.

- 4 Repeat the steps above and create the transitions as shown in the following state diagram.



Note: You can change the position of the transition lines and the labels for the transitions by selecting and moving the object.

You will notice that the transition lines are broken $\frac{\perp}{\perp}$. This broken line means that these transitions do not have a trigger defined. A trigger defines which events from which ports cause the transition to be taken. The trigger is associated with the port on which the triggering event is expected to arrive. Moreover, a transition can have multiple triggers such that an event that satisfies any one of the triggers causes the transition to be taken.

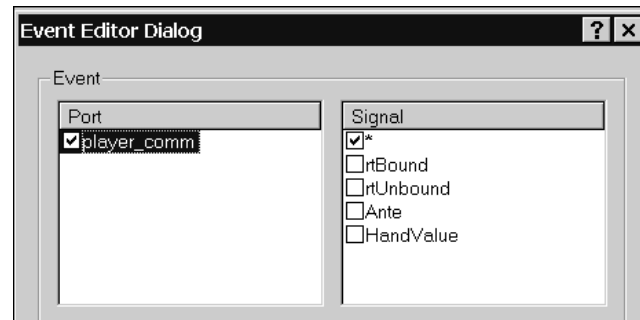
To create the triggers:

- 1 Double click on the **Deal_cards** transition line.
The **Transition Specification** dialog box appears.
- 2 Click the **Triggers** tab.

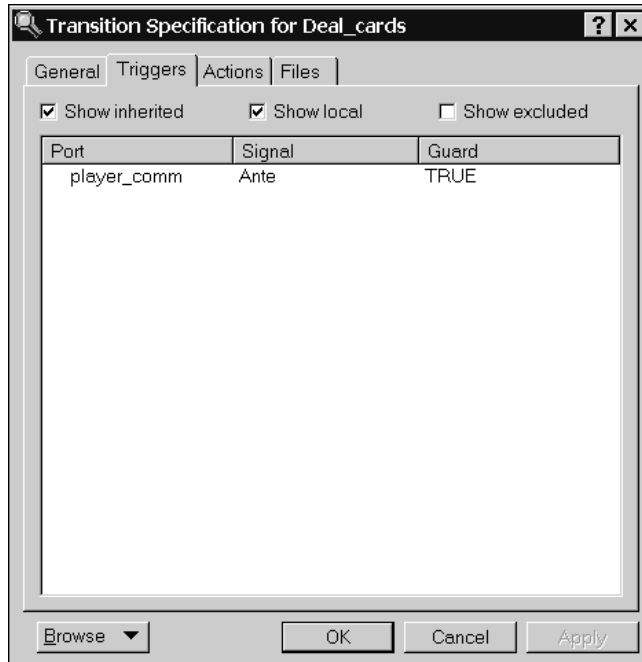


- 3 Right-click in the list area, and click **Insert**.
- 4 Double-click **player_comm**.

The incoming signals defined for this port automatically appear in the **Signal** list. Use this dialog to define the ports and signals that trigger this particular transition to be taken.



- 5 In the **Signal** list, select the **Ante** signal.
- 6 Click **OK**.



The trigger you defined now appears on the **Triggers** tab of the **Transition Specification** dialog box.

- 7 Click **OK**.
- 8 Repeat the steps above to create a trigger for the **Calc_hands** transition using the following:

Port	Signal
player_comm	HandValue

The **Transition Specification** for the Calc_hands transition looks like the following.



- 9 Click **OK** to close the **Transition Specification** for **Calc_hands**.

To create the actions

Actions are the things the behavior does when a transition is taken. They represent executable atomic computations that are written as statements in a detail-level programming language (that is, in C++) and incorporated into a state machine. Actions are atomic, in that they cannot be interrupted by the arrival of a higher priority event. An action runs to completion.

- 1 Double-click on the **Deal_cards** transition line.
The **Transition Specification** dialog box appears.
- 2 Click the **Actions** tab.

3 In the **Code** box, type the following:

```
// distribute hands to player and dealer
for( int i = 0; i < 5; i++ )
{
    player_comm.ACard().send();

    // will add code here later to take
    // a card for the dealers hand
}
```

Note: C++ is case-sensitive. Ensure that you type the code exactly as shown to avoid any errors.



When **Deal_Cards** transition is taken, the dealer sends the **ACard** signal out the **player_comm** port. A total of five cards are sent. This is the first iteration of the behavior. After you develop the card classes, you can send actual cards to the player and take cards for the dealers hand. This level of detail code allows you to run the simulation, and to observe the interactions between the player and dealer capsules.

4 Click **OK**.

5 Repeat the above steps and type the following code for the **Calc_hands** transition:

```
// Receive the hand value from the player
int hand_value = *rtdata;
int bet_multi = 2;
// For now, always let the player win twice
// their bet value.
// Will add code here to compare
// the two hands.
player_comm.Win( bet_multi ).send();
```



6 Click **OK**.

In the **Calc_hands** transition, the dealer uses the **rtdata** argument to extract the data sent with the signal. The **rtdata** parameter is available to all transition code and is a casted version of the data in a message. The `rtdata` parameter is casted to the highest common superclass of the possible data classes for the given code segment. In

this case, the **CommHeadsUp** protocol tells us that the **HandValue** signal is accompanied by an integer representing the value of the players hand. For now, you will let the player win all the games.

Creating the Player's Behavior

The player's behavior is more involved than the dealers behavior because the player starts the simulation at some interval, and counts the value of their cards.

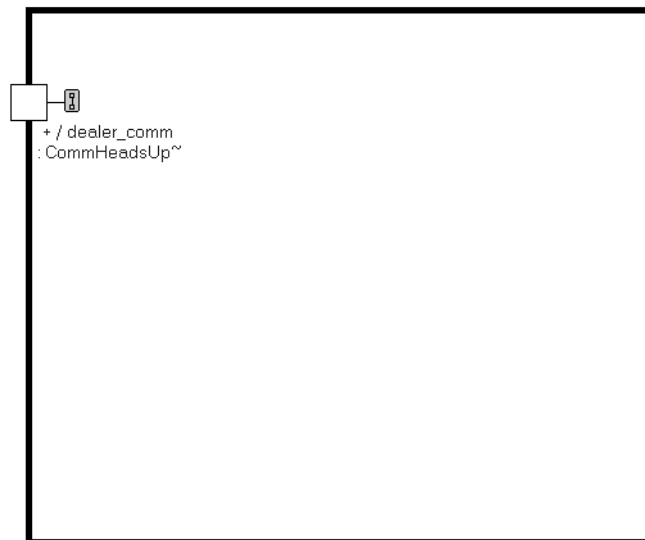
Suggested Reading


- Functionality of the language add-in, *Rational Rose RealTime C++ Reference*.

To create a timing port:

The player will initiate the flow of events. To control the amount of time elapsed between games, you will use a timer to have an interval of two seconds between each game to help you debug and observe the simulation while it runs.

- 1 Open the **Structure Diagram** for the **Player** capsule.



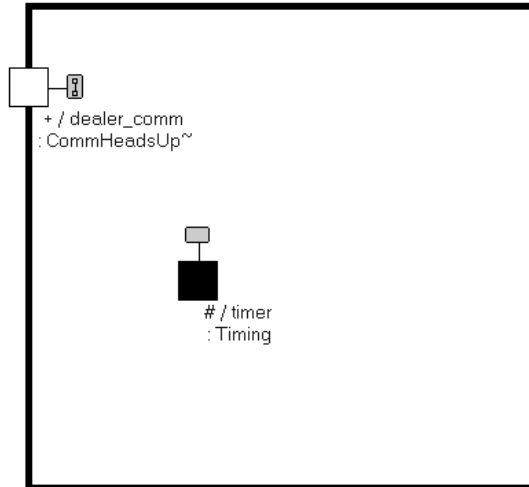
- 2 From the capsule **Structure Diagram** toolbox, select the **Port** tool .

- 3 In the **Structure Diagram**, left-click inside the black border to add the port.

Note: Do not click on the black border. Clicking on the border makes the port *public*, clicking inside the border makes the port *private*.




- 4 Double-click to select the **Timing** protocol from the list of available protocol classes.
- 5 Rename the port to **timer**.

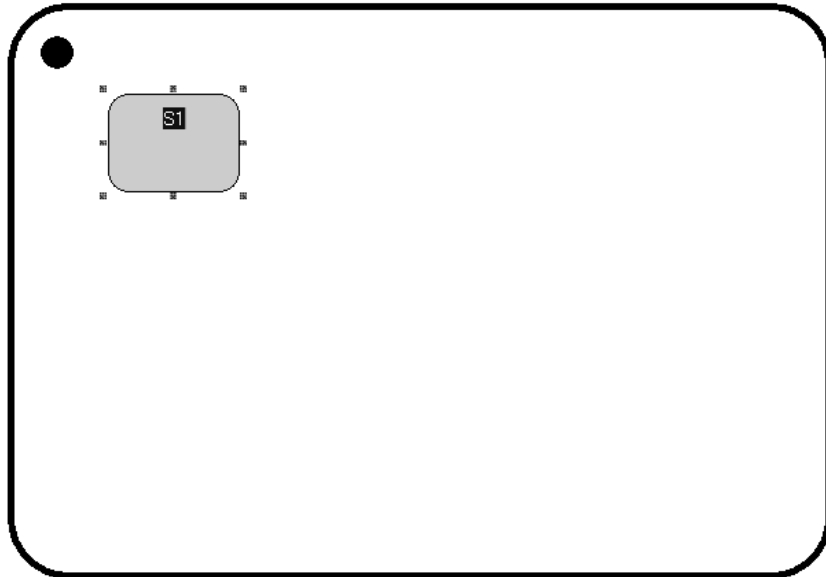


Creating the State Diagram

You will create the behavior for the **Player** capsule. As with the **Dealer** capsule, you will create the appropriate states and transitions and label them. Your completed State Diagram for the Player capsule will look like the following diagram.

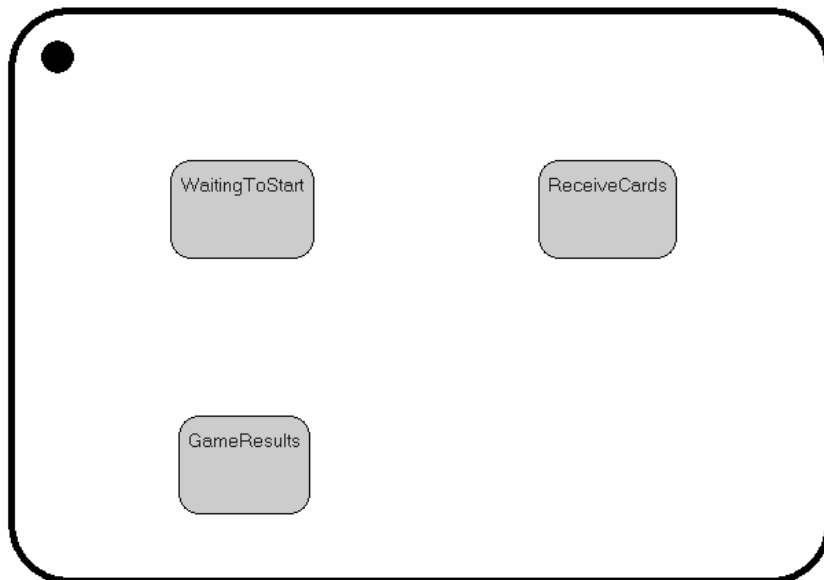
- 1 If not currently open, open the **State Diagram** for the **Player** capsule.
- 2 Click the **State** tool  from the **State Diagram** toolbox.
- 3 Move the mouse over the **State Diagram** and within the state diagram border.

- 4 Click to create a state.



- 5 Rename the state to **WaitingToStart**.
- 6 Repeat the steps above, and create two additional states called **ReceiveCards** and **GameResults**.

Your diagram should look like the following.




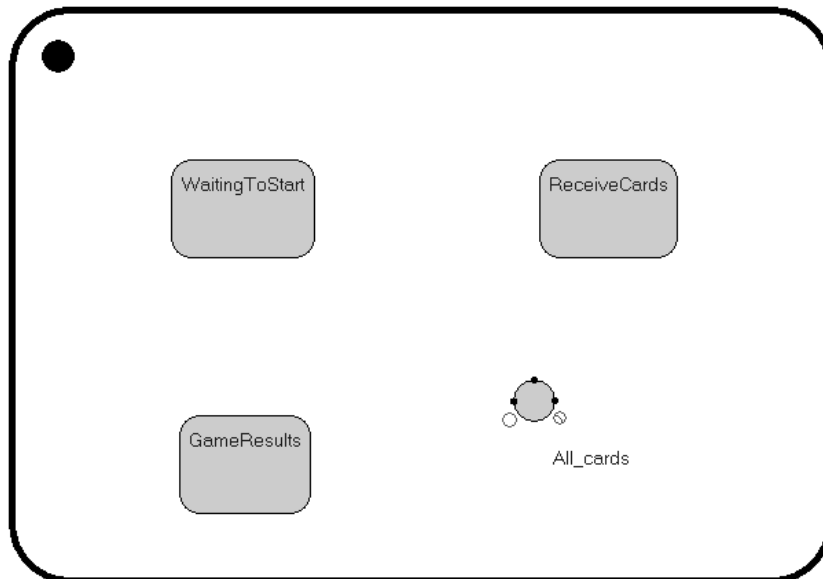
Next, you will create a choice point.

To create a choice point:

Choice points allow a single transition to be split into two outgoing transition segments, each of which can terminate on a different state. The decision of which branch to take is made after the transition is taken.



Each choice point has an associated boolean predicate that is evaluated after the incoming transition action is executed. Depending on the truth value of this predicate, either the True or the False branch is taken.

- 1 From the **State Diagram** toolbox, click the **Choice Point** tool .
- 2 Click in the diagram to add a choice point.
- 3 Rename the choice point to **All_cards**.

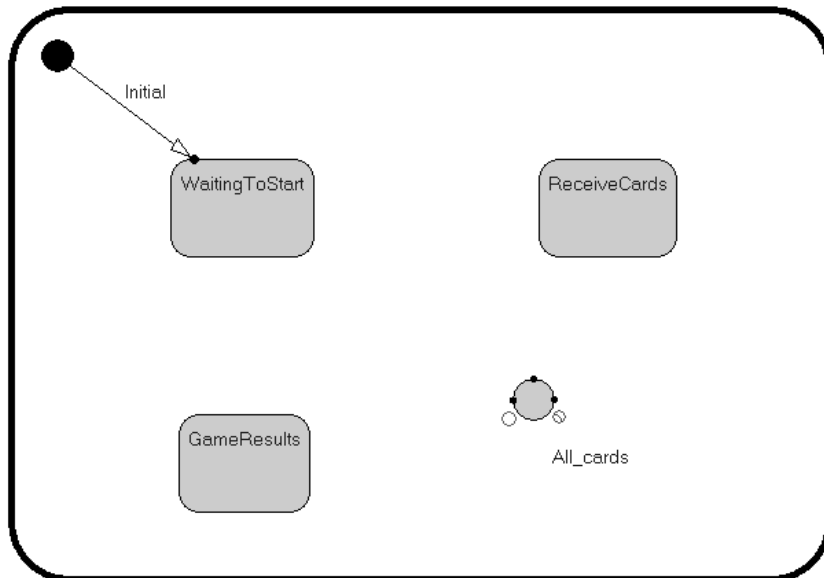


Note: You can rotate the choice point by grabbing one of its pic handles and turning. You can change the True and False branches using the popup menu.

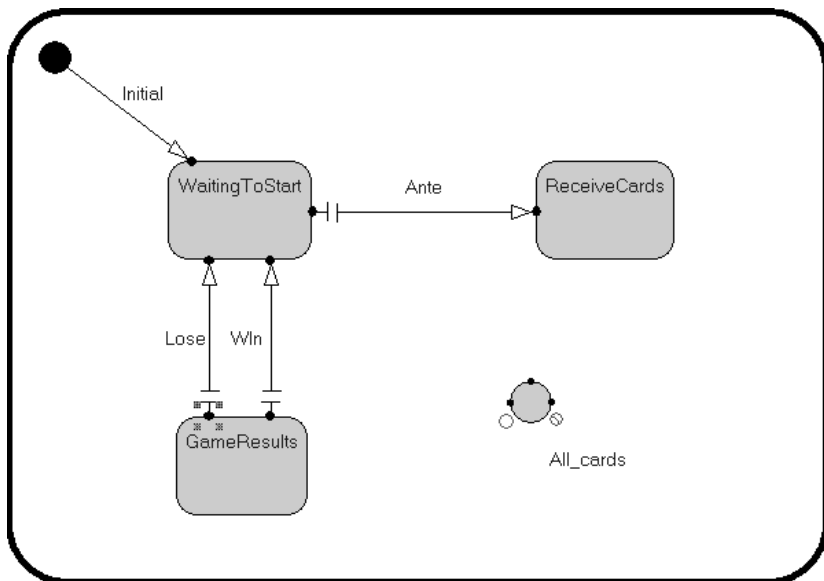
Next, you want to create transitions.

- 1 From the **State Diagram** toolbox, click the **State Transition** tool .
- 2 Click and hold the left mouse button on the **Initial Point** in the state diagram, .
- 3 Drag the **State Transition** tool to the top of the **WaitingToStart** state.

The **Initial Transition** has a default name of **Initial**.

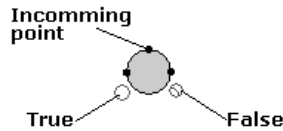


- 4 Repeat the steps above and create the transitions as shown in the following State Diagram.



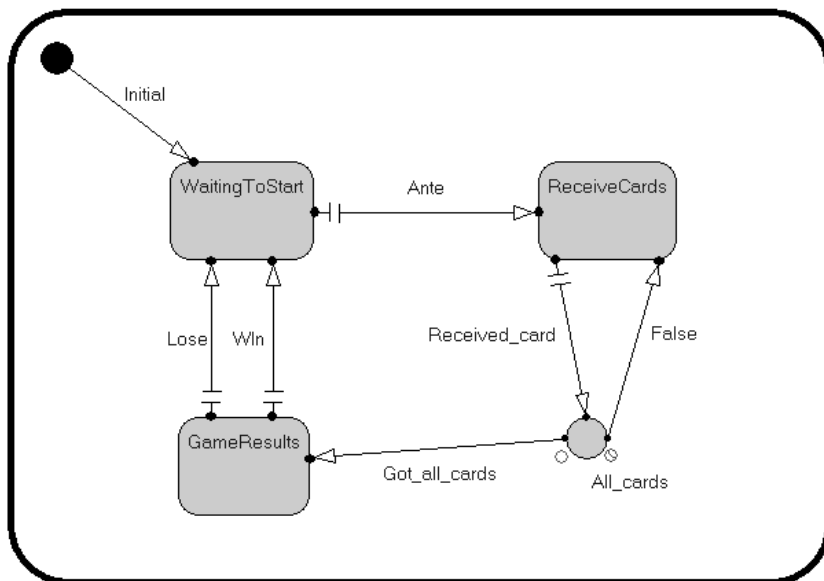
Note: You can change the position of the transition lines and the labels for the transitions by selecting and moving the object.

Now, you will add the transition lines to and from the choice point. A choice point is rendered as a circle with one incoming point, and two outgoing for the True and False transitions. The choice point is shown with a 'C' in the middle if the boolean predicate is defined.



- 5 Add the transitions for the choice point.

Your **State Diagram** for the **Player** capsule looks like the following.



You will notice that some transition lines are broken $\frac{\perp}{\perp}$. This broken line means that these transitions do not have a trigger defined.

To create the triggers:

- 1 Double click on the **Ante** transition line.
The **Transition Specification** dialog box appears.
- 2 Click the **Triggers** tab.

- 3 Right-click in the list area, and click **Insert**.
- 4 Double-click **timer**.

Use this dialog to define the ports and signals that trigger this particular transition to be taken.

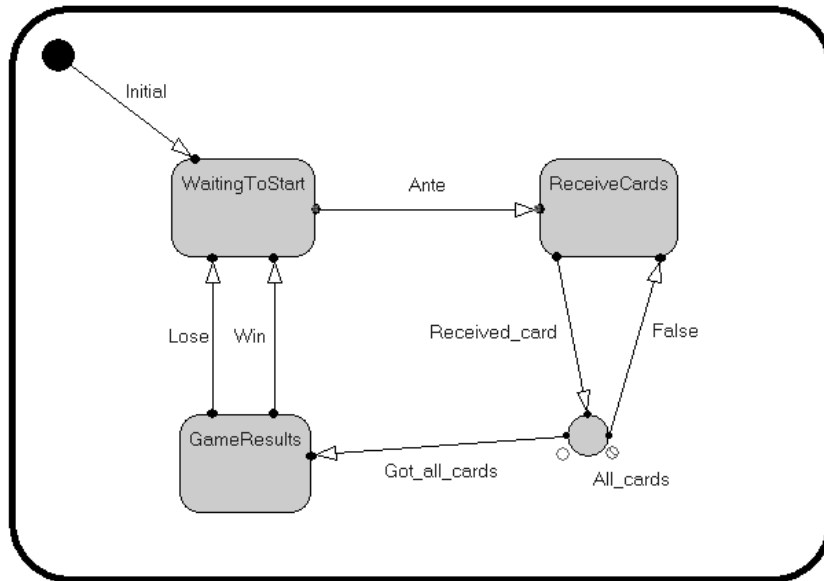
- 5 In the **Port** list, select **timer** port.
The incoming signals defined for this port automatically appear in the **Signal** list.
- 6 In the **Signal** list, select the **timeout** signal.
- 7 Click **OK**.
- 8 Click **OK**.
- 9 Repeat steps 1 through 8 to create triggers for the following transitions:

Transition	Port	Signal
Received_card	dealer_comm	ACard
Win	dealer_comm	Win
Lose	dealer_comm	Lose

Note: Transitions out of the choice point do not require triggers. They belong to the same transition chain as the transition incoming to the choice point.

Verify that there are no broken transition lines. If you have a broken transition, check that you have defined a trigger for a transition.

You **State Diagram** should look like the following:



Adding Attributes

The Player’s attributes manage the data required to track the bet, the winnings, and the number of cards received. These attributes can be accessed in the action code of the capsule state diagram and in its operations (very similar to how classes attributes are available in its operations).

Attribute	Description
_bet	Stores the amount of money (rounded up to the dollars) that the player bets for each hand.
_money	Stores the money for the player. The player starts with \$150. This value is updated after each hand to reflect either a win or a loss.
_ncards	Stores the number of cards the player receives. This attribute helps the player know when they have received all five cards.

To add Attributes:

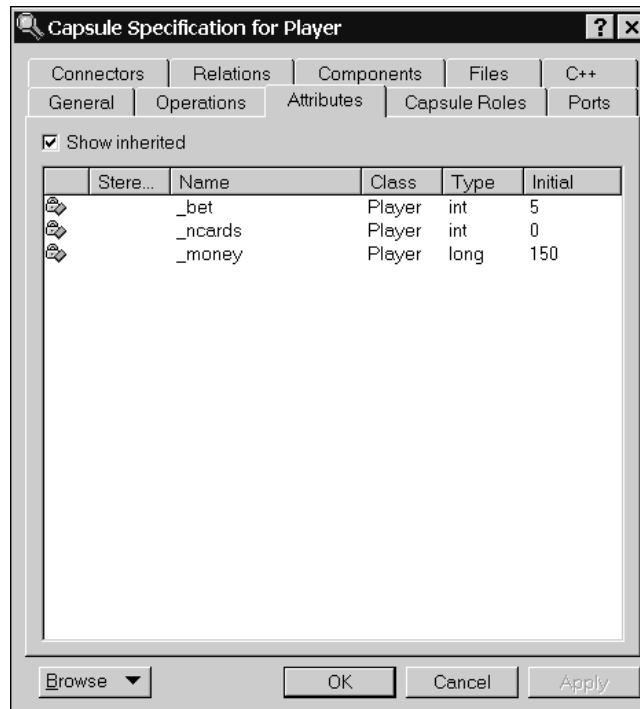
Note: The following steps show you how to add an attribute using the Specification dialogs; however, you can also use the Attribute wizard. Right-click on a capsule and click Attribute Tool.

To create attributes:

- 1 Open the **Specification** dialog box for the **Player** capsule.
- 2 Click the **Attributes** tab.
- 3 Right-click in the **Attributes** list, and click **Insert**.
- 4 Type **_bet** and press **ENTER**.
- 5 Press **TAB** twice to advance to the **Type** column.
- 6 Press **F8** and select **int** from the list.
- 7 Press **TAB** to advance to the **Initial** column.
- 8 Press **F8**, type **5**, and press **ENTER**.
- 9 Repeat steps 3 through 8 to add the following attributes:

Name	Type	Initial Value
_ncards	int	0
_money	long	150

The Capsule Specification dialog box looks like the following:



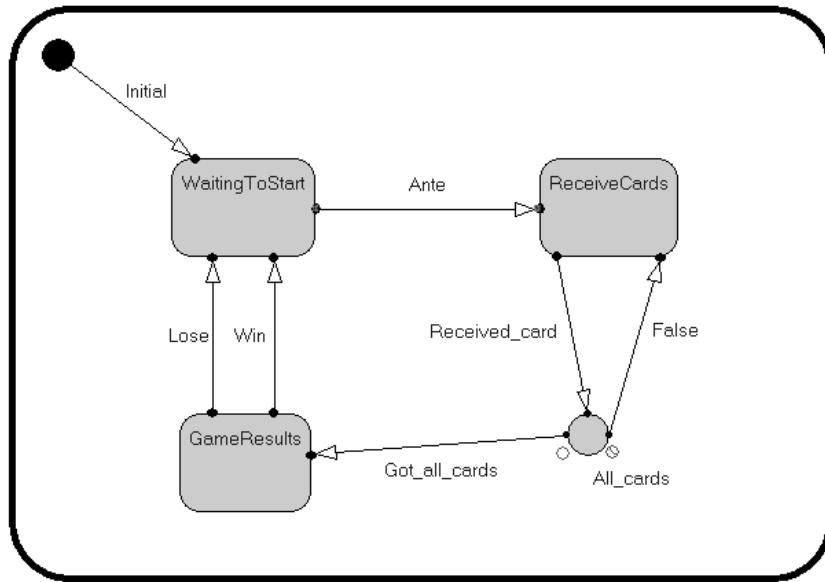
10 Click OK.

Creating the Actions

The player capsule starts the flow of events by sending an Ante signal to the dealer. The player must then receive the five cards from the dealer, and reply with the value of his hand. Depending on the outcome of the game, he will receive a Win or Lose signal. If he wins, the dealer sends the bet multiplier that is used to calculate how much was won.

To add a state entry action:

- 1 Open the **State Diagram** for the **Player** capsule.



- 2 Right-click on the **WaitingToStart** state, click **Open Specification**, then click the **Entry Actions** tab.

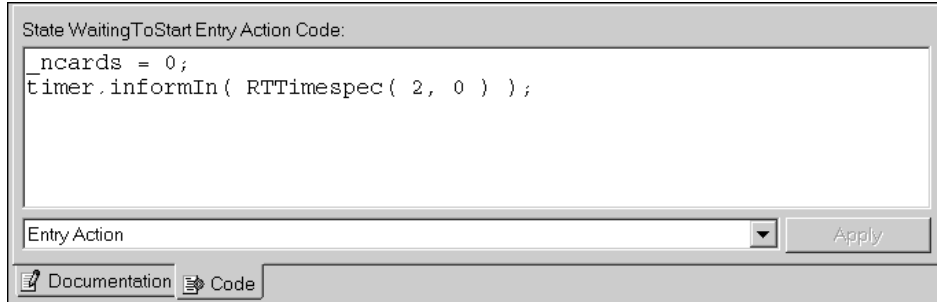
Or...

Click the **WaitingToStart** state, on the **Code** tab in the **Documentation** window, select **Entry Action** from the drop-down list.

- 3 Type the following code in the **Code** box:

```
_ncards = 0;
timer.informIn( RTTimespec( 2, 0 ) );
```

If you used the **Code** tab in the documentation window, your window will look like the following:



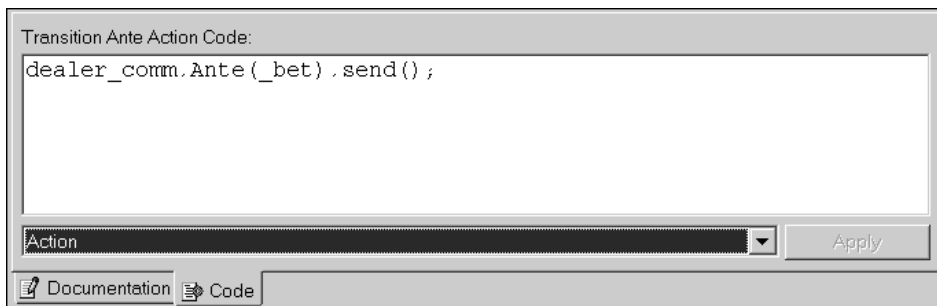
State entry actions are executed every time a transition is taken which terminates on the state. The `_ncards` attribute resets for each game. The `timer` requests that one time-out message be sent to this capsule in two seconds; this enables the simulation to run continuously.

- 4 Click **OK** in the **Transition Specification dialog box** or click **Apply** in the **Code** window.

To add a transition action for Ante:

- 1 Click the **Ante** transition line.
- 2 In the **Code** window, select **Action** from the drop-down list.
- 3 Type the following in the **Code** box:

```
dealer_comm.Ante(_bet).send();
```



By adding this code, you specify that the player starts the game by sending an ante to the dealer.

- 4 Click **Apply**.

To add a transition action for **Got_all_cards**:

- 1 Click on the **Got_all_cards** transition line.
- 2 On the **Code** tab, select **Action** from the drop-down list.
- 3 Type the following code in the **Code** box:

```
// Will have to update to send the real hand value  
dealer_comm.HandValue(2).send();
```

- 4 Click **Apply**.

Note: The card classes are not developed, so you will send a dummy hand value to the dealer. You will have to update this code after you develop the card classes.

To add a transition action for **Win**:

- 1 Click on the **Win** transition line.
- 2 On the **Code** tab, select **Action** from the drop-down list.
- 3 Type the following in the **Code** box:

```
// Update winnings from bet and win multiplier  
// sent from dealer.  
_money += _bet * (*rtdata);
```

- 4 Click **Apply**.

If the player wins, the dealer sends the bet multiplier to calculate the amount of money relative to the bet that the player has won.

To add a transition action for **Lose**:

- 1 Click on the **Lose** transition line.
- 2 On the **Code** tab, select **Action** from the drop-down list.
- 3 Type the following code in the **Code** box:

```
// update for loss  
_money -= _bet;
```

- 4 Click **Apply**.

By adding this code, you specify that if the player loses, subtract the bet from total money for the player.

To add a action for the **All_cards** choice point:

Each choice point has an associated boolean predicate that is evaluated after the incoming transition action is executed. Depending on the truth value of this predicate, one or the other branch is taken. Modify the predicate for the **All_cards** choice point:

- 1 Click on the **All_cards** choice point.
- 2 On the **Code** tab, select **Condition** from the drop-down list.
- 3 Type the following code:

```
return( ++_ncards < 5 );
```

This code forces the branch to take the False transition until the player receives all five cards.

- 4 Click **Apply**.

Before proceeding with the next lesson, we recommend that you save and build your model. For details on building the model, see Lesson 4: Building and Running. If you encounter any errors, review this lesson and fix any errors until you receive the message "Build successful" in the **Log** window.

Note: If you do not want to proceed to Lesson 6 using the file you created at the end of Lesson 5, you can open the file `<ROBERT_HOME>/Help/Tutorials/cardgame/cardgame_step2.rtmidl`, and then continue with Lesson 6.

Review

The **Player** and **Dealer** capsules are not complete. Later, you will add code to complete the Transition actions. The capsules are complete enough to allow you to build, run, and debug the simulation.

Lesson 6: Navigating and Searching

As the size of your model increases, you may need to start navigating and searching. In this lesson, you will do some reading to become familiar with the navigation and search tools available within Rational Rose RealTime.

To obtain a working knowledge of a model, you must find key elements, such as, capsules, attributes, operations, classes, code statements, and so on. You must then navigate through the code and/or structures to find other occurrences, establish the hierarchy, and get a general understanding of the flow of the design.

Suggested Reading

- The Find dialog, *Rational Rose RealTime Toolset Guide*
- The Code window, *Rational Rose RealTime Toolset Guide*
- Model Browser, *Rational Rose RealTime Toolset Guide*
- The Browse menu, *Rational Rose RealTime Toolset Guide*

Lesson 7: Using Traces and Watches to Debug the Design

In this topic, you will learn how to use the monitors, traces, and watches features. You can use these tools to help debug Rose RealTime models.

A very powerful feature of Rational Rose RealTime is the ability to observe a running component instance at the model level. This high-level debugging is not what most developers are familiar with. Typically, developers converted design models to source code, and when it was compiled and run, the only way to trace the execution was at the source code level. The design model representation was not useful.

In Rational Rose RealTime, you can see the triggered transitions, active states in the state diagram monitors, and watch the dynamic structure animate in the structure monitor. In addition, you can use probes to trace the messages being passed in the system.

Suggested Reading

- Summary of the observability options, *Rational Rose RealTime Toolset Guide*

Rebuilding the Model

You can use the same component, component instances, and processor that you had created previously. Now it is time to rebuild the **CardGameComponent**. For details on building the model, see *Lesson 4: Building and Running* on page 106.

When you rebuild the **CardGameComponent**, all changes made to the model are regenerated and recompiled into a new executable.

Errors may be reported during the build. This is not a problem. Read the error as shown in the **Build Errors** pane in the output window, and then double-click on the error to bring you to the element that caused the error. (C++ syntax errors are a common type of error.) Correct any errors and then rebuild the model.


Note: If you do not want to proceed to Lesson 7 using the file you created at the end of Lesson 6, you can open the file `<ROSERT_HOME>/Help/Tutorials/cardgame/cardgame_step1.rtmdl`, and then continue with Lesson 7.

Setting Up the Runtime Windows

First, you want to run your model.

- 1 On the **Model View** tab in the browser, expand **Deployment View**, then expand **LocalHost**.
- 2 Right-click **CardGameComponentInstance** and click **Run**.
- 3 If you are prompted to build the component, click **Yes**.

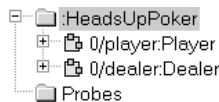
After running the executable (component instance), the **RTS Browser** appears.

- 4 Click the **Step** button  to initialize the model.

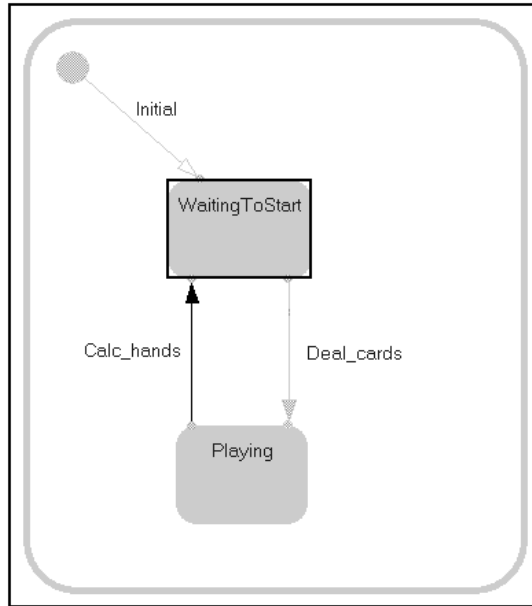
Clicking the **Step** button allows the delivery of one message in the component instance runs allows the current executing transition to finish, the next message is delivered, then it either pauses or stops.

To open the State monitor:

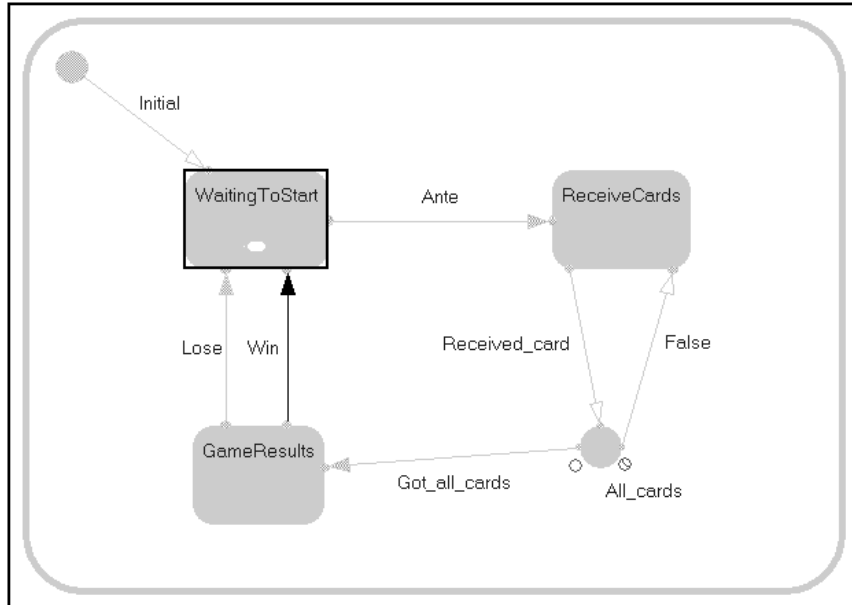
- 1 From the **RTS Browser** pane on the **Runtime View** tab in the **Model** browser, click the **Start** button  , and expand the **HeadsUpPoker** (capsule instance) folder.



- 2 Right-click on the **Dealer** capsule instance, and click **Open State Monitor**.



- 3 Right-click on the **Player** capsule instance, and click **Open State Monitor**.



These windows show you read-only views of the capsule instance behavior, and they animate to show you the current state and last transition taken by the capsule instance while it runs.

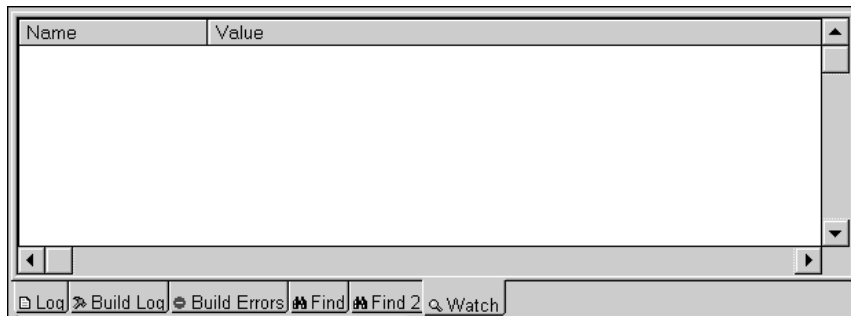
Organize the monitor windows, you can use the zoom tool to resize. Try and organize your desktop so that you can see both **State Monitor** diagrams at the same time.

For both capsules, there is a black rectangular outline around the **WaitingToStart** states; it is the first state.

To open the Watch window:

- 1 From the **View** menu, click **Output** so that it is selected.

The **Watch** window appears in the output window. The window has two columns: **Name** and **Value**.

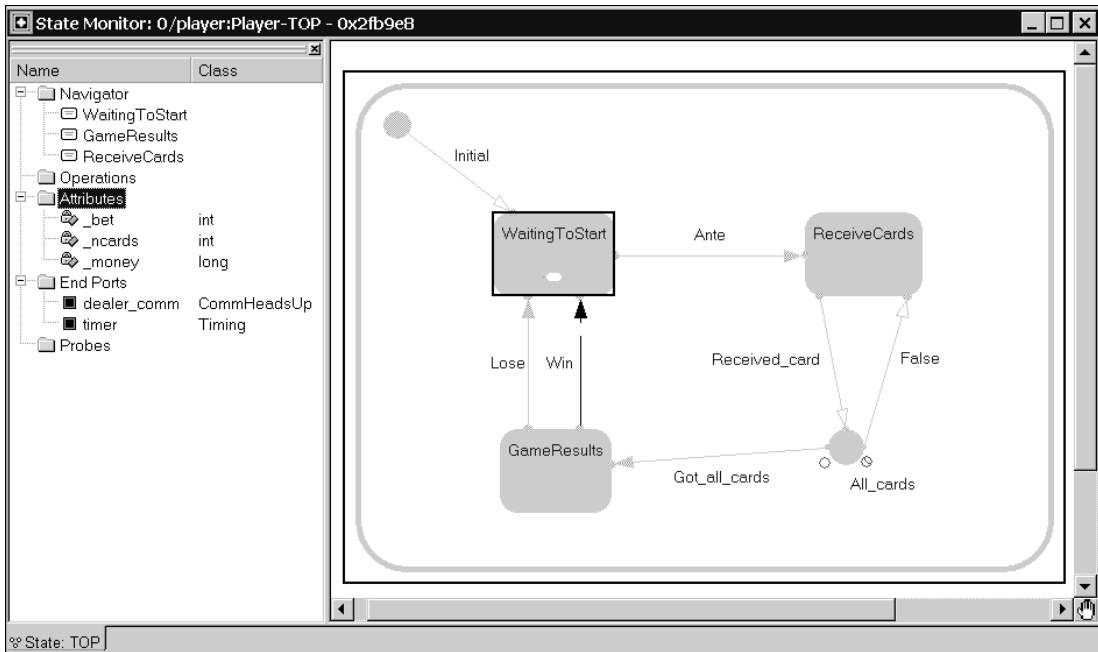


Note: Use the **Watch** window to specify attributes that you want to watch while debugging your running component instance. You can also modify the value of a variable using the **Watch** window.

You will place watches on three attributes for the **Player** capsule instance.

- 2 From the browser in the **Player** State Monitor, expand the **Attributes** folder located in the left hand pane.

There are three attributes listed: `_bet`, `_ncards`, and `_money`.



- 3 Drag each attribute individually from the **Attributes** folder into the **Watch** window.

The attributes appear in the window. The value is not updated until the component instance starts.

Name	Value
<code>_bet</code>	5
<code>_ncards</code>	1
<code>_money</code>	7860

619x226

Log Build Log Build Errors Find Find 2 Watch

To open a capsule instance Trace

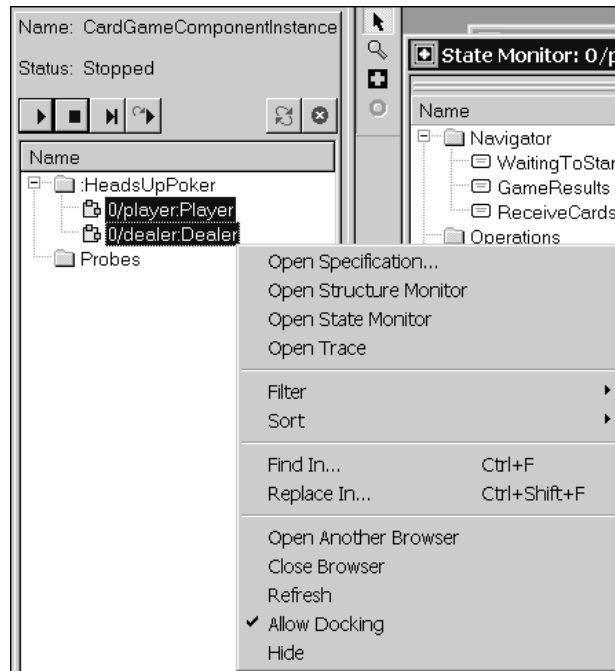
You can trace messages sent between a set of capsule instances. Typically, when a message fails to flow through a set of capsules as expected, it is important to see where the message flow was first in error. To debug these kinds of errors, first use Capsule instance traces to look at the messages originating and terminating from the capsules in the message flow.

For the purposes of this lesson, you want to open a trace on the **Player** and **Dealer** capsule instances.

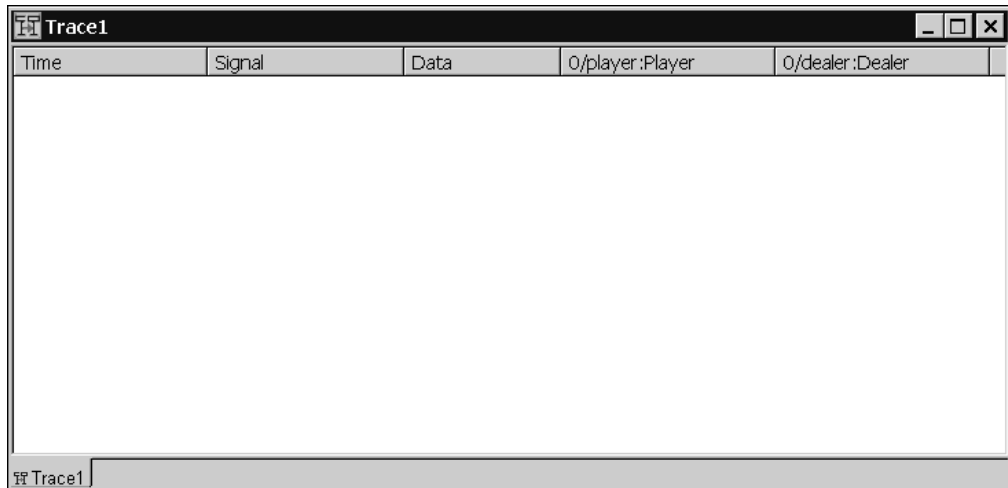
- 1 On the **Runtime View** tab, from the **RTS browser**, select the **Player** and **Dealer** capsule instances.

Note: You may have to click the **Stop** button  to select both capsules.

- 2 Right-click, and select **Open Trace**.



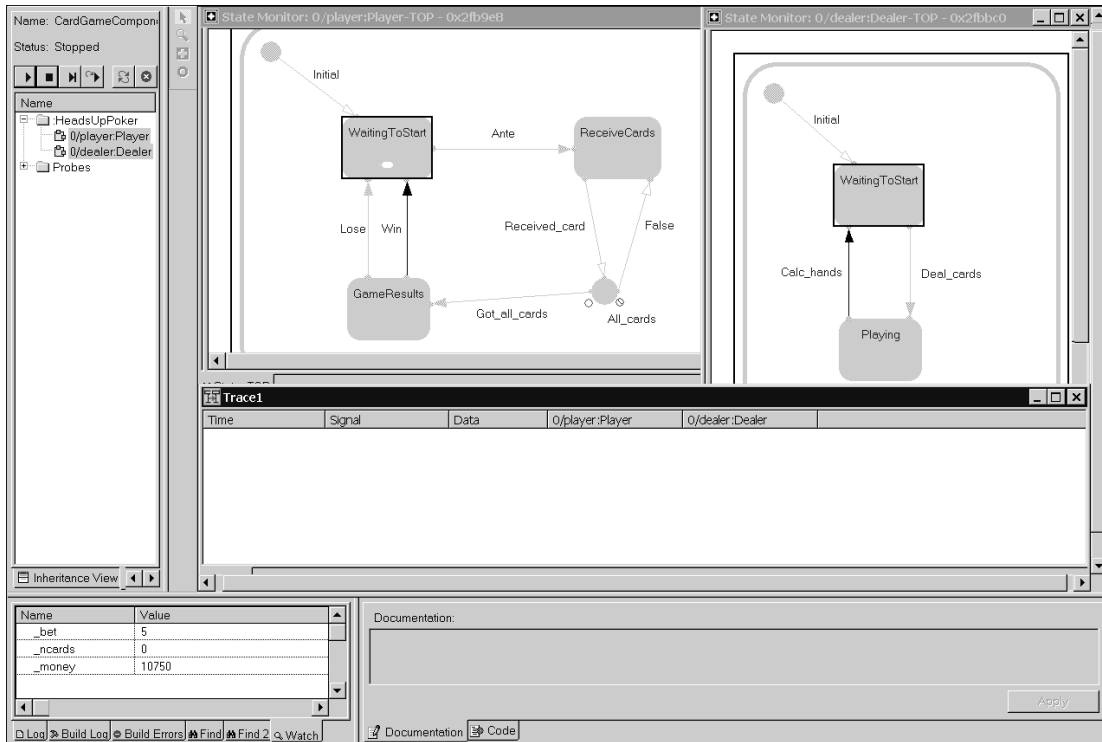
Initially, the **Trace** window looks like the following:




- 3 Resize the **Trace** window so that it fits your screen.
- 4 To resize the columns, right-click on the top column (the one with the titles) and select **Adjust Columns** from the menu.

The columns resize to fit the current window size.

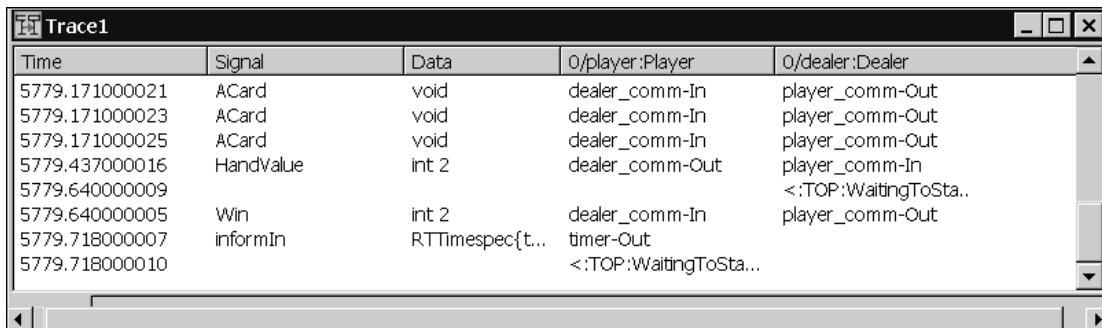
Note: Messages appear in the **Trace** window after the component instance starts.



To start the component instance

- 1 Press the **Start** button  in the **RTS Browser (Runtime View tab in the Model browser)** window to start the component instance.

Note: If data does not begin to appear in the **Trace** window, you may have to stop, then start the trace again.



Messages appear in the **Trace** window, and the **Value** fields for the attributes appear in the **Watch** window. Since you implemented the simulation so that the player always wins, the player's value for **money** will increase.

- 2 Double-click in the **Value** column for the **_bet** attribute in the **Watch** window, and change the value to 10000.

This modifies the value of that attribute in the running instance, and you can see that the player wins more money.

- 3 Click the **Shutdown** button  to terminate the component instance.

Problems with the Player Capsule

An error was purposely introduced in the behavior of the player capsule. In this topic, you will debug and fix this error.

When you ran the component instance, a console window appeared. You will use output from this console window, and use probes and traces to find the error.

If you do not have the **CardGameComponentInstance** running and started, do so now.

Suggested Reading

- Probes, *Rational Rose RealTime Toolset Guide*

This topic highlights a very powerful feature of Rational Rose RealTime. You can build any capsule independently in order to run and debug it. Since the **Player** and **Dealer** capsules do not run individually, they do not encapsulate a flow of events; they are part of one. When they are built and run, they will not do anything unless you simulate messages that it expects to trigger behavior. All behavior in a capsule is triggered by received messages; therefore, when a capsule is run by itself, or out of context of its flow of events, you have to generate these messages to test its behavior.

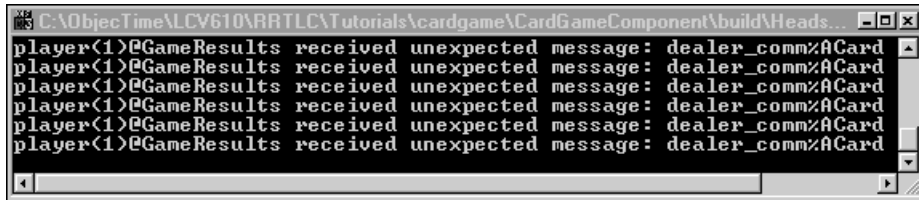
Unexpected Messages

Ensure the component instance is started, then observe the behavior monitors for the player and dealer instances. Is the behavior what you expected? Now look at the **Trace** window. Are all the messages that you expect there?

If you examine the results closely, you will notice that the **Player** instance is not actually receiving all five cards. The **Player** should receive five cards, and then send the hand value to the dealer. However, the **Player** only receives one card, and the four others are being ignored.

Warning Message for No Defined Trigger?

At runtime, if a capsule does not have a trigger for a message it receives, (that is, when a capsule receives a message that cannot be processed in the current state), a warning message is printed to **stdout** in the console windows, as follows:

A screenshot of a Windows console window. The title bar shows the path: C:\ObjectTime\VCV610\BRTLCA\tutorials\cardgame\CardGameComponent\build\Heads... The console contains five lines of text, each representing a warning message: player(1)@GameResults received unexpected message: dealer_comm%ACard. The messages are repeated five times, one for each line.

```
C:\ObjectTime\VCV610\BRTLCA\tutorials\cardgame\CardGameComponent\build\Heads...
player(1)@GameResults received unexpected message: dealer_comm%ACard
player(1)@GameResults received unexpected message: dealer_comm%ACard
player(1)@GameResults received unexpected message: dealer_comm%ACard
player(1)@GameResults received unexpected message: dealer_comm%ACard
player(1)@GameResults received unexpected message: dealer_comm%ACard
```

The Services Library prints this warning when a capsule instance receives a message and there is no transition event defined from the current state which can handle the event.

In this case, messages printed to the console indicate that the **Player** instance is in the **GameResults** state when it received the **ACard** signal on the **dealer_comm** port. This occurred several times.

Building the Player Capsule

In this topic, you create a component and component instance to build and run the **Player** capsule. This allows you to run the **Player** capsule to investigate the unexpected message.

To create the component and component instance

- Create a component named **PlayerComponent** with the **Player** capsule as the top level capsule. Follow the instructions in the topic *To create a component*: on page 108.
- Create a component instance of the **PlayerComponent**. Follow the instructions in the topic *To build the CardGameComponent* on page 116.

To build the PlayerComponentInstance


Build the **PlayerComponent** in the same way that you built the **CardGameComponent**. Right-click the component, and click **Build**.

For additional information on building a component instance, see *To build the CardGameComponent* on page 116.

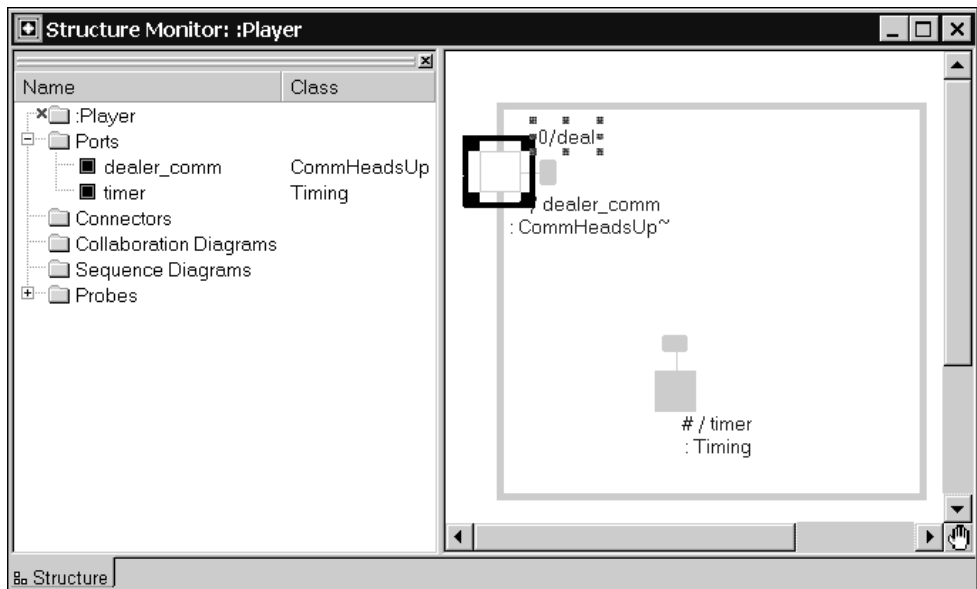
Debugging the Player Capsule

To debug the **Player** capsule, run it then inject messages into the **dealer_comm** port to observe the player capsule instance behavior.

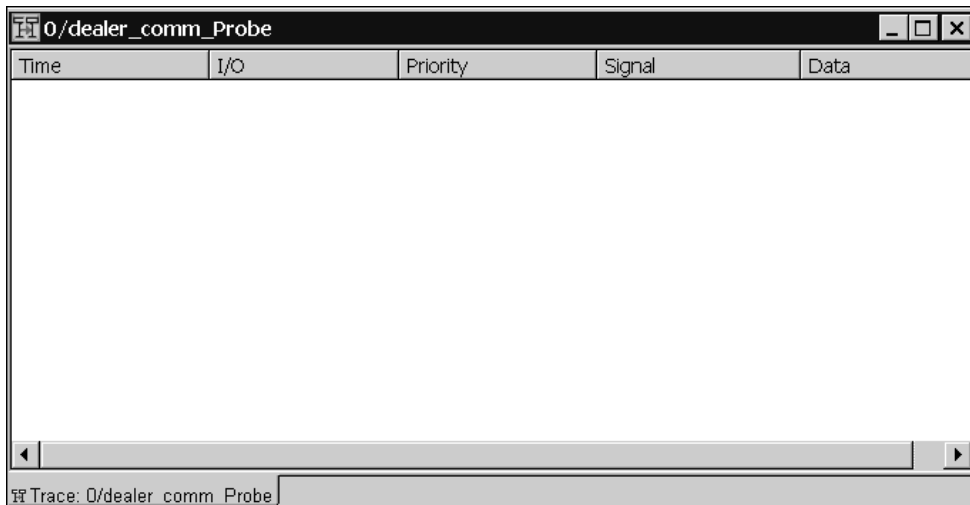
To trace and inject messages:

- 1 Run the **PlayerComponentInstance**. For instructions *To run the component instance:* on page 118.
- 2 Open the **Structure Monitor** diagram for the **Player** capsule.
- 3 Select the **Probe** tool  from the **Structure Monitor** toolbox.
- 4 Position the cursor over the **dealer_comm** port in the **Structure Monitor** diagram, and left-click.

This creates a port on the **dealer_comm** port.

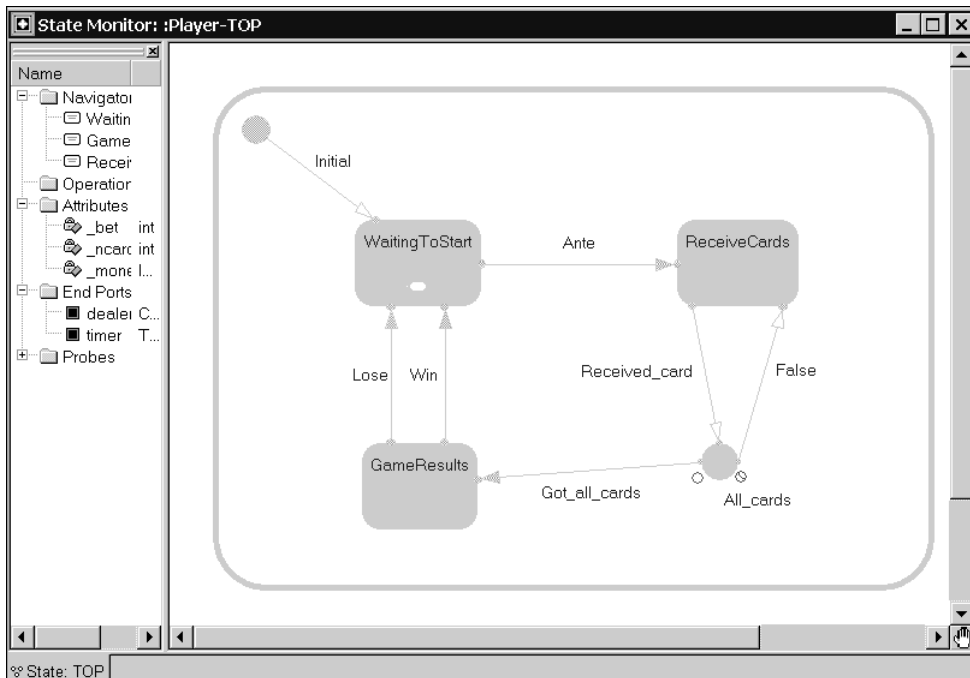


- 5 Select the probe, right-click, and click **Open Trace**.



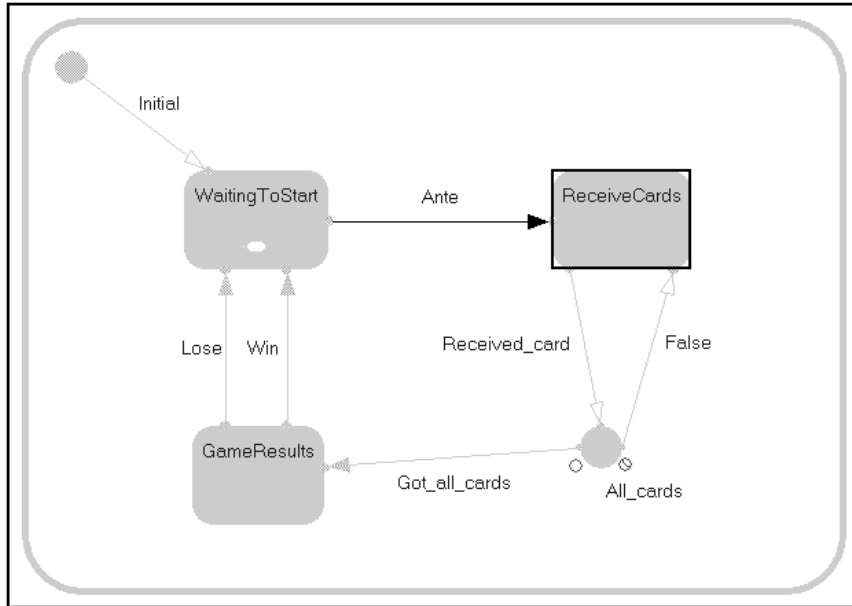
A **Trace** window shows all messages that pass through (in or out) of a port.

- 6 From the **Runtime View** tab, right-click on **Player** and click **Open State Monitor**.

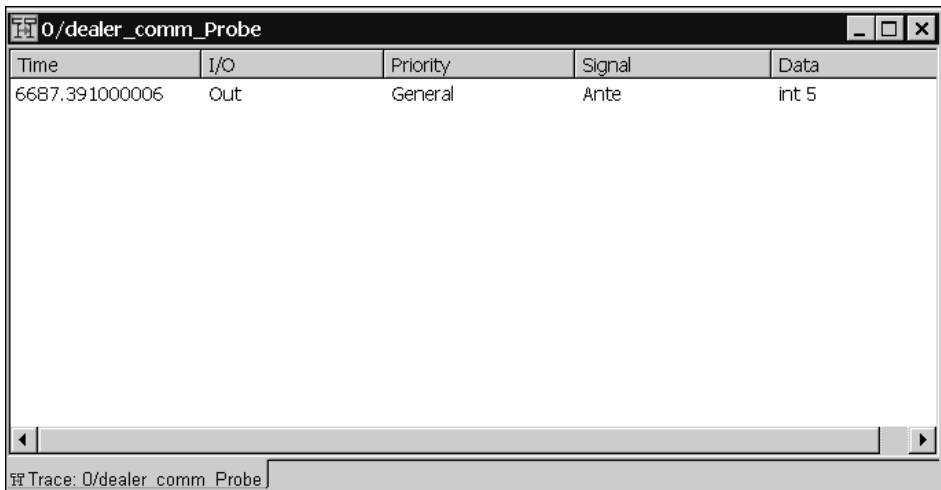


- 7 From the **Runtime View** tab, click the **Start** button  to start the execution of the loaded component instance.

After a two second delay, the state changes from **WaitingToStart** to **ReceiveCards** in the State Monitor diagram for Player.



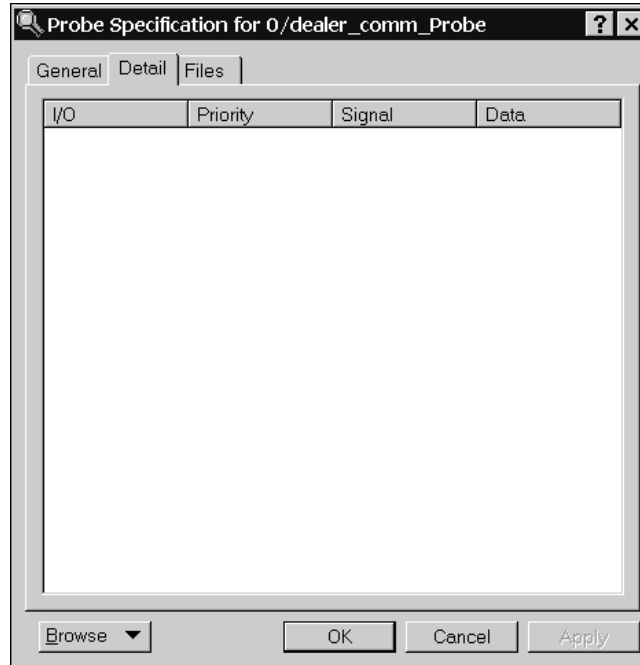
A message appears in the **Trace** window with signal **Ante**.



The Player remains in the **ReceiveCards** state until receiving cards from the **dealer_comm** port. Since a dealer instance is not running, you will inject messages into the player instance.

- 8 In the **Structure Monitor** dialog for the **Player** capsule instance, select the probe, right-click and click **Open Inject**.

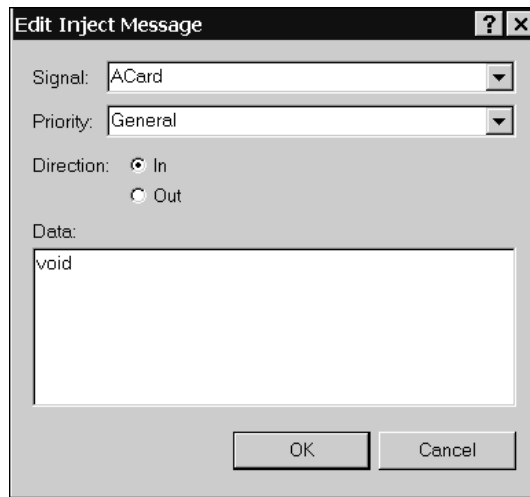
The **Probe Specification** dialog box appears with the **Detail** page active.



- 9 Right-click in the list, and click **Insert**.

The **Edit Inject Message** dialog box appears.

- 10 From the **Signal** box, select the **ACard** signal, and set the **Direction** to **In**.



- 11 Click **OK**.

You created the specification for an inject message that appears in the inject list of the **Probe Specification** dialog box.

- 12 In the **Probe Specification** dialog box, select the inject message, right-click, and click **Inject**.

The state monitor shows the behavior moving from the **ReceiveCards** state to the **GameResults**. This is incorrect, and is a symptom of the problem with the **Player** capsule. The **Player** capsule should remain in the **ReceiveCards** state until it receives five cards. Instead, it only receives one card before it changes state.

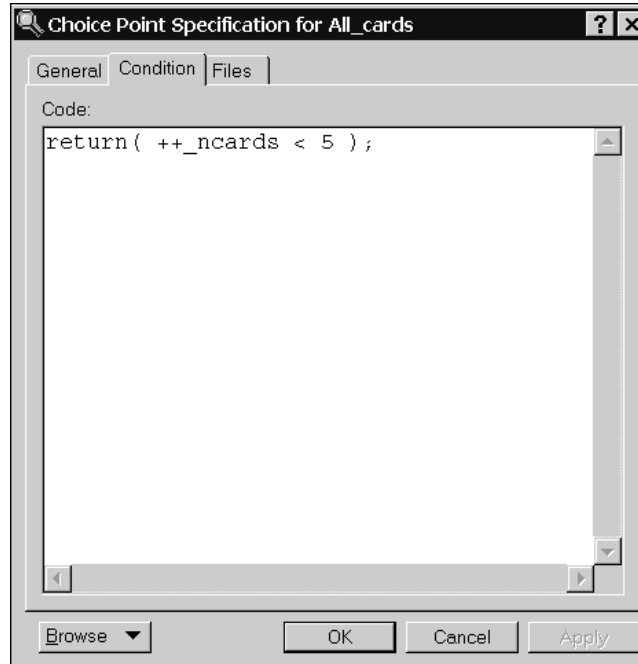
This indicates that there is an error with the logic in the choice point. In the next steps, you will correct the error and test the changes.

Note: Ensure that the **PlayerComponentInstance** component instance is running. You can check the status by looking at the **Status** field at the top of the **RTS Browser** on the **Runtime View** tab: the status should indicate **Running**.

To correct the error in the Player capsule:

- 1 Without stopping or shutting down the **Player** capsule instance, double-click on the **All_cards** choice point in the **State Monitor** dialog for the **Player** capsule.

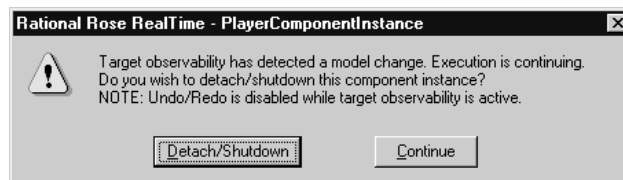
The **Choice Point Specification** dialog box appears.



- 2 If not currently selected, click the **Condition** tab.
- 3 Replace the code with:

```
return( ++_ncards >= 5 );
```
- 4 Click **OK**.

A dialog appears indicating that you have changed the model and the component instance must be shut down.



- 5 Click **Detach/Shutdown**.

The **RTS Browser** closes.

- 6 In the **Model View** tab in the browser, right-click the **PlayerComponentInstance** in **LocalHost** under **Deployment View**, then click **Run**.

The changes you made earlier force the re-compilation of the **Player** capsule.

- 7 Click **Yes** when queried to recompile the component.

- 8 From the **Runtime View** tab, click the **Start** button  to start the execution of the loaded component instance.

Now, you will inject five **ACard** messages into the player instance to ensure that the capsule behaves as expected.

- 9 In the **Runtime View** tab in the browser, right-click **Player** and select **Open Structure Monitor**.

- 10 On the probe you created earlier, right-click and select **Open Inject**.

- 11 On the **Detail** tab, right-click on the **In** message.

- 12 Click **Inject**.

- 13 Repeat steps 11 and 12 four more times, for a total of five inject messages.

After you inject five cards, the **Player** capsule instance changes to the **GameResults** state. You will now inject a **Win** message.

If you look at the State Diagram, the current state is **GameResults**. Now, you will create a **Win** inject message to test the **Win** transition and complete one hand of the **HeadsUp** poker game.

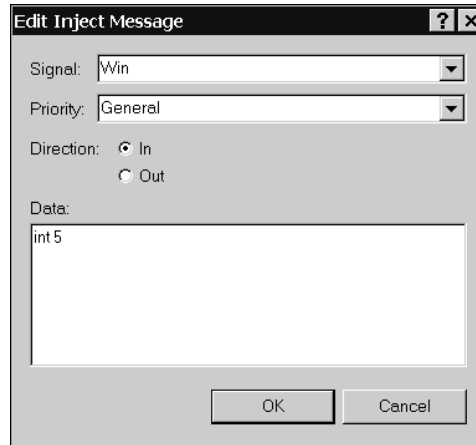
- 14 In the **Structure Diagram** for the **Player** capsule, right-click on the probe you created earlier, then select **Open Inject**.

- 15 On the **Detail** tab of the **Probe Specification** dialog box, right-click and select **Insert**.

- 16 In the **Signal** box, select **Win**.

17 In the **Data** box, delete **void** and type:

```
int 5
```



This inject message requires an integer data value that represents the bet multiplier.


18 Click **OK**.

19 On the **Probe Specification** dialog box, select the **Win** signal, right-click and select **Inject**.

You can see the **Win** transition being taken. If you have a watch on the **_money** attribute, you can see that value is updated to 155 to correctly reflect the fact that the player has won one hand.

Verifying the Fix

You have located and fixed the error with the **Player** capsule. You should now run the simulation and verify that it works as expected. Ensure that unexpected message warnings no longer appear in the console window. Try injecting a **Lose** message to verify that the **_money** attribute is updated correctly when the player loses.

Note: When you finish testing your model, in the **Runtime View** tab of the browser, click the **Shutdown** button . Also, close the **State Monitor**, **Structure Monitor**, **Trace**, and **Probe Specification** dialog boxes.

Review

In this lesson, you learned how to:

- Use the observability features of the toolset that enable you to debug and test a running process built with Rational Rose RealTime at the model level. Instead of stepping through the generated source code, Rational Rose RealTime shows you a view of the structure diagrams and state diagrams of the running capsule instances.
- Build, run, and test individual capsules at any time. This is very useful for unit testing parts of your design, or trying to isolate a problem.
- Expand the initial analysis and design model and added detailed design components to your model. That includes state diagrams and source code.
- Test the design before starting to expand the implementation details.

Next, you will create and integrate the card classes used by the **Player** and **Dealer** capsules, and used to complete the implementation of the **HeadsUp** poker simulation.

Before continuing with the next lesson, we recommend that you save your work.

Lesson 8: Class Modeling

In this lesson, you will create the structure and behavior for the classes identified from the main use case. This is not an UML tutorial. This tutorial assumes that you have a basic understanding of the UML concepts such as classes, relationships, multiplicity, and polymorphism.

In a previous lesson, you created the main classes, **Card**, **Deck**, and **Hand**, but you have not yet added any details to them. You will be implementing the following class structure in this lesson:

Class	Description
Card	Represents a playing card with a suit and rank. Suit values are: 1 - 4 (Heart, Club, Spade, Diamond) Rank values are: 1 - 13 (Ace-King)
CardList	Responsible for providing basic memory and access services for a list of cards of any size.

Class	Description
Hand	A hand refers to the cards a player holds in a card game. A hand is a general concept and alone does not represent a specific game hand. It should be specialized for each particular game. A hand knows how to determine its game value.
Deck	A deck refers to the cards the dealer holds and distributes in a card game. The deck can be shuffled and re-ordered.
PokerHand	Is a specialized hand for poker games. It knows how to evaluate the value of a poker hand. This kind of hand can contain a maximum of five cards.

Importing Classes

To implement the classes, you will add detail code to the classes.

Note: If you are **already familiar** with class modeling (for example, if you are an experienced Rose user), you have the option of importing the completed classes into the model. We recommend that you read the tutorial (without having to create the classes and enter the detail level code) because there are some added properties that are specific to Rose RealTime for associations and classes.

To import classes (only for those familiar with class modeling):

Note: If you are new to Rational Rose RealTime, we recommend that you skip this Import and perform the actual steps.

- 1 On the **Model View** tab in the browser, right-click on the **Logical View** folder.
- 2 Click **File > Import**.
- 3 Locate the directory that contains the tutorial model files (<ROSE_HOME>/Tutorials/cardgame), and select the file card_classes.rtpl.
- 4 Click **Open**.

The classes in the file are merged into the model. Since you already have classes in your model that have the same names as those being imported, the classes you imported into your model are renamed. If you open the log window, you can see which classes were renamed by the import operation. Delete the classes that you had originally created, and rename the imported classes.

Note: The imported classes are contained in their own package, called **CardDefinitions**.

Creating a Package

A package is a collection of classes, relationships, use-case realizations, diagrams, and other packages. A package structures the model by dividing it into smaller parts. Packages are used primarily for model organization, and serve as a unit of configuration management. By grouping design model elements into packages, and showing how those groupings relate to one another, it is easier to understand the overall structure of the model.


You will create a package to organize the card classes that will be created.

To create a package:

Note: If you imported the classes, **do not** continue with the remaining steps in Lesson 8. Continue with *Lesson 9: Adding Card Classes to the Capsule Behavior* on page 198.

- 1 In the **Model View** tab in the browser, right-click on the **Logical View** package, and click **New > Package**

Or...

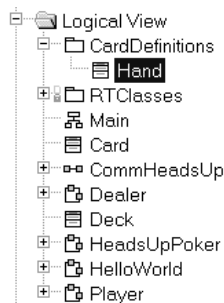
From the toolbox, select the Package button  and click on the diagram.

- 2 Rename the package to **CardDefinitions**.

To move classes to the new package:

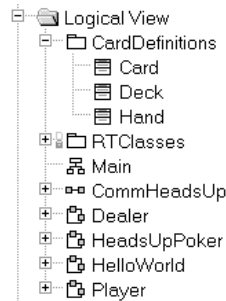
- 1 In the **Model View** tab in the browser, left-click on the **Hand** class and drag the class over top of the **CardDefinitions** package in the **Logical View** folder.

The class element moved to the new package.



- 2 Repeat the previous steps for the **Deck** and **Card** classes.

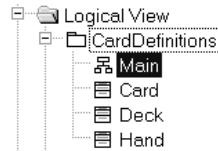
When finished, your **Logical View** folder will look like the following:



To create a class diagram for the new package:

You may want to have at least one class diagram per package to describe the contents of the package.

- 1 Right-click on the **CardDefinitions** package, and click **New > Class Diagram**.
A new class diagram is created, with a default name of **NewDiagram**.
- 2 Rename the Class Diagram to **Main**.



Creating the Initial Class Structure

Each class works together to carry out more behavior than individual classes. From the class descriptions, you can understand the responsibilities of each class. Now, you will specify the relationships between the classes and design how the classes will work together.

Suggested Reading:

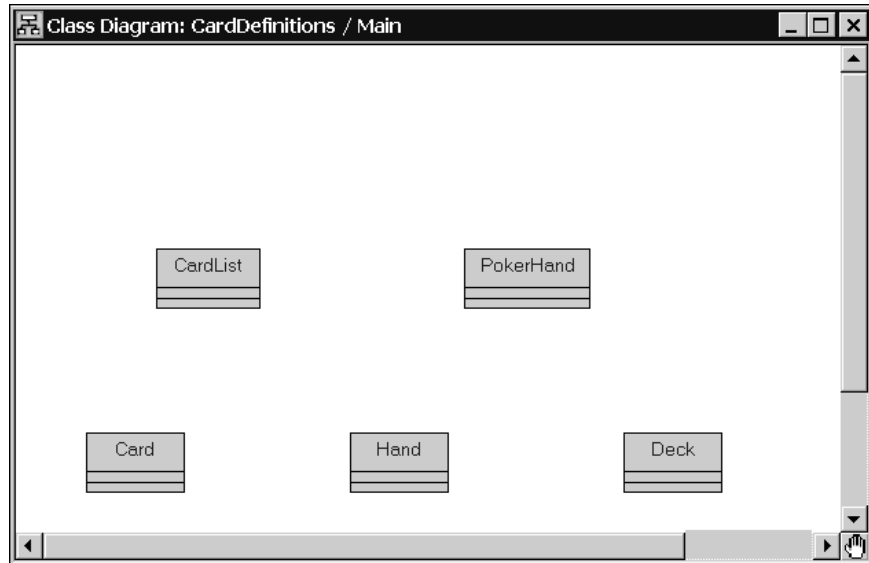
- The Class Diagram, *Rational Rose RealTime Toolset Guide*
- Creating associations, *Rational Rose RealTime Toolset Guide*

To populate the class diagram:

- 1 On the **Model View** tab in the browser, expand the **CardDefinitions** package, then double-click **Main** to open the **Class Diagram** window.
- 2 Drag the **Hand** class from the **Model View** tab on to the **Class Diagram** window.

- 3 Repeat the previous step for the **Deck** and **Card** classes.
- 4 From the **Toolbox**, select the **Class** tool and create two new classes called **CardList** and **PokerHand**.

When finished, your **Class Diagram** for the **CardDefinitions** package will look like the following:




Creating Relationships Between Classes

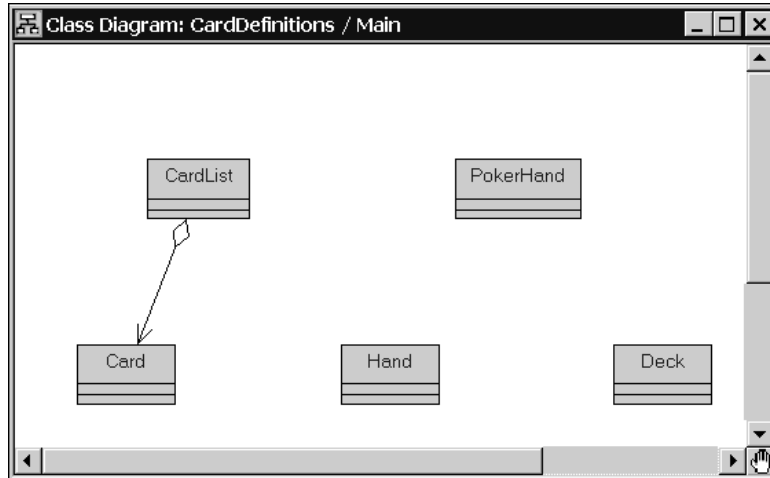
In this topic you will create associations between the card game classes. An association is a structural relationship used to connect one element to another. Associations can be used during analysis to initially identify general relationships between classes. As your model evolves, you will add additional properties to associations to make them more specific.


Aggregation relationships are a form of association relationship that indicate one class (the contained class) is a part-of another class (the container, or aggregate class).

To create an aggregation between **Card** and **CardList**:

- 1 From the **Class Diagram** toolbox, select the **Unidirectional Aggregate Association** tool .
- 2 Click on the **CardList** class, and drag the mouse to the **Card** class.

An aggregation relationship is created between the two classes. The **Card** class is the part and the **CardList** the whole.

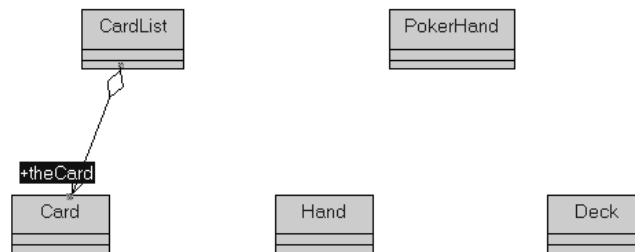


- 3 Right-click on the association end  nearest to the **CardList** class, and clear **Navigable** if it is selected.

The arrow appears that near the **Card** class means that it is not possible to navigate from the **Card** class to the **CardList** class using the association.

- 4 Right-click on the association end near **Card**, and click **End Name**.

A text box near the association end initialized with a default end name.



- 5 Rename to **_contents**.

By default, when an end is named, association, aggregation, and composition relationships are represented in the code as an attribute in the client class. The code generation does not generate attributes for ends which are not named.

- 6 Right-click on the association end near **Card**, and click **Protected**.


Protected means that it is visible to this class, any subclasses of this class, and any designated friend classes.

- 7 Right-click on the association end near **CardList**, and ensure that **Aggregation > Aggregate** is selected.

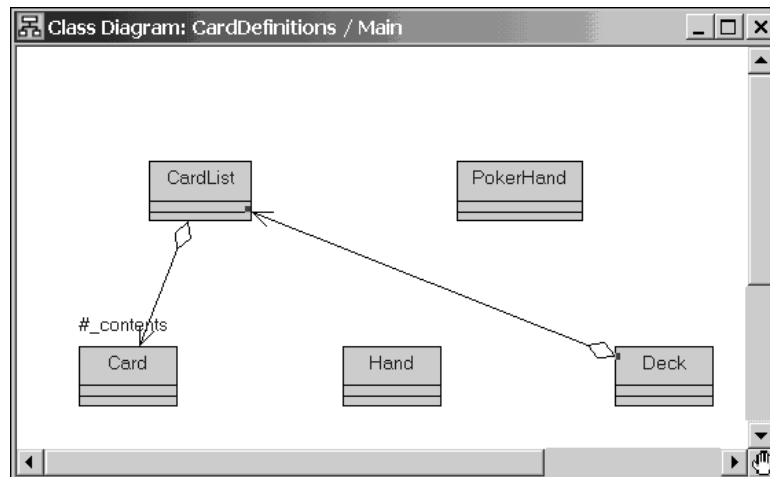
Only use aggregation for a composition relationship between classes, where one class is composed of other classes, where the "parts" are incomplete outside the context of the whole.

Next, you will create an aggregation relationship between the **CardList** class and the **Deck** class. You want to create this relationship because the cards selected for a Player's hand are a subset of the cards from a deck.

To create an aggregation between **CardList** and **Deck**

- 1 Select the **Unidirectional Aggregate Association** tool .
- 2 Click on the **Deck** class, and drag the mouse to the **CardList**.

An aggregation relationship is created where **Deck** is the whole-part and **CardList** is the part-of.



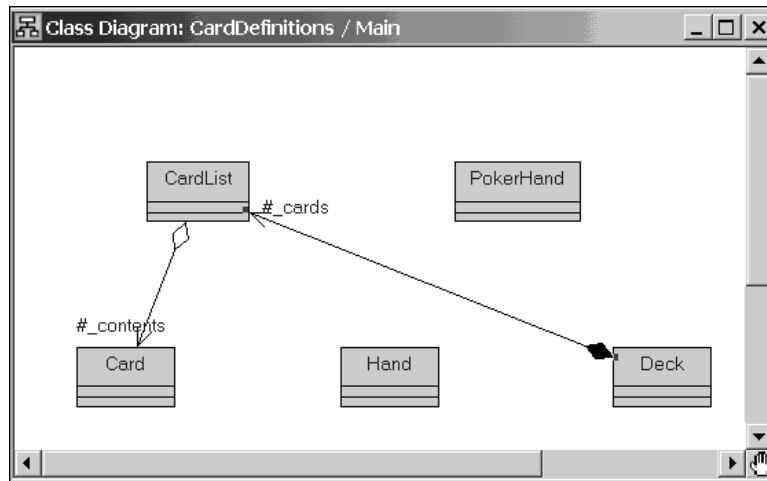
- 3 Right-click on the association end closest to the **CardList** class, and click **End Name**.
- 4 Rename the end name to **_cards**.
- 5 Right-click on the association end near **CardList**, and click **Protected** so that it is checked.

- 6 Right-click on the association end near **Deck**, and click **Aggregation > Composite**.

Composition is a form of aggregation with strong ownership and coincident lifetime of the part with the aggregate. By implication, a composite aggregation forms a "tree" of parts, with the root being the aggregate, and the "branches" the parts.


- 7 Right-click on the association end near **Deck**, and clear **Navigable** if it is currently set.

Your **Class Diagram for CardDefinitions** will look like the following:

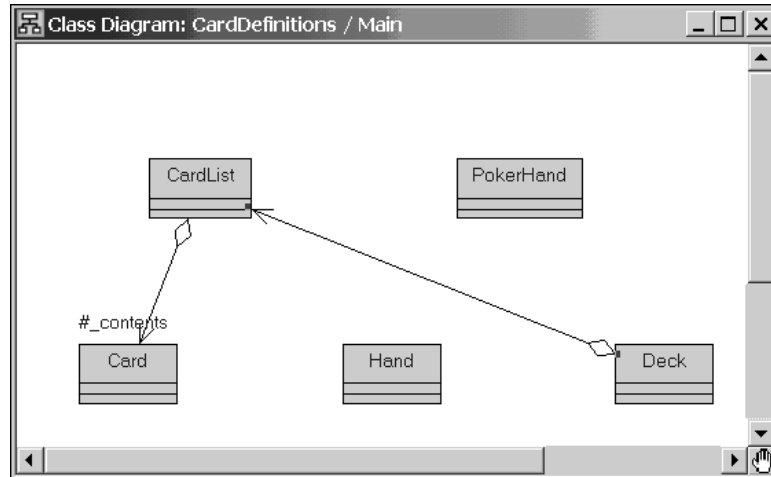


To create an aggregation between **CardList** and **Hand**

Now, you will create an association between **CardList** and **Hand**.

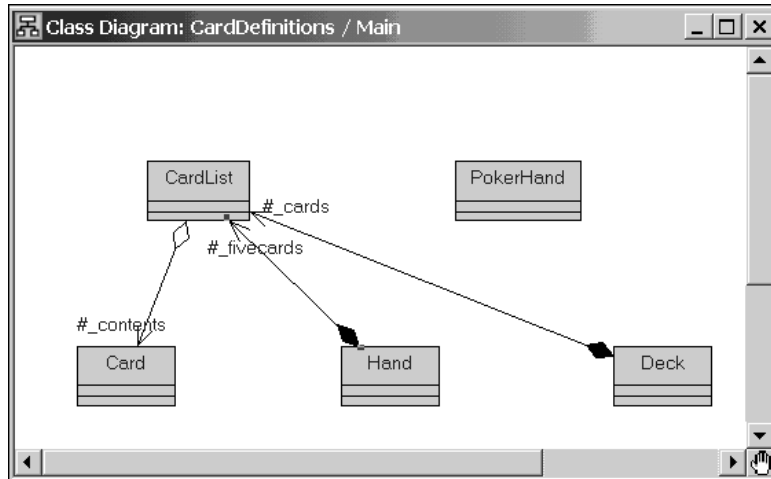
- 1 Select the **Unidirectional Aggregate Association** tool .
- 2 Click on the **Hand** class, and drag the mouse to the **CardList**.

An aggregation relationship is created where **Hand** is the whole-part and **CardList** is the part-of.




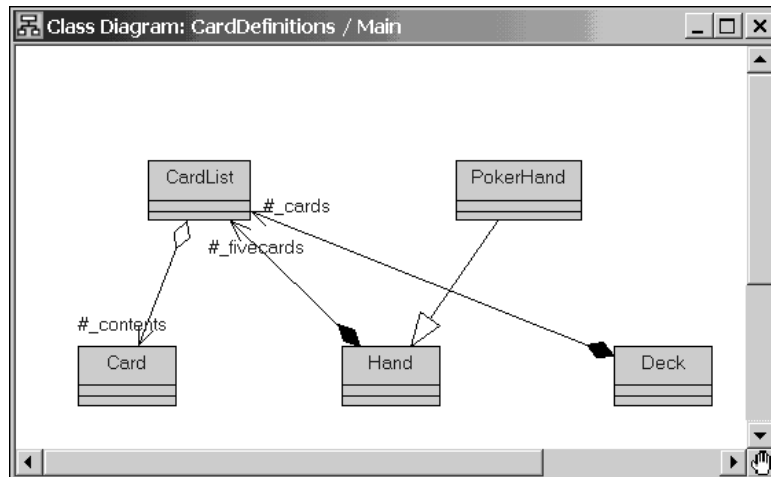
- 3 Right-click on the association end closest to the **CardList** class, and click **End Name**.
- 4 Rename the end name to `_fivecards`.
- 5 Right-click on the association end near **CardList**, and click **Protected** so that it is checked.
- 6 Right-click on the association end near **Hand**, and click **Aggregation > Composite**.
- 7 Right-click on the association end near **Hand**, and clear **Navigable** if it is currently set.

When finished, your class diagram will look like the following:



To create a generalization between Hand and PokerHand

- 1 Select the **Generalization** tool .
- 2 Click on the **PokerHand** class, and drag the mouse to the **Hand** class.



A generalization relationship is created between the two classes. The **Hand** class is the super class and the **PokerHand** the subclass.

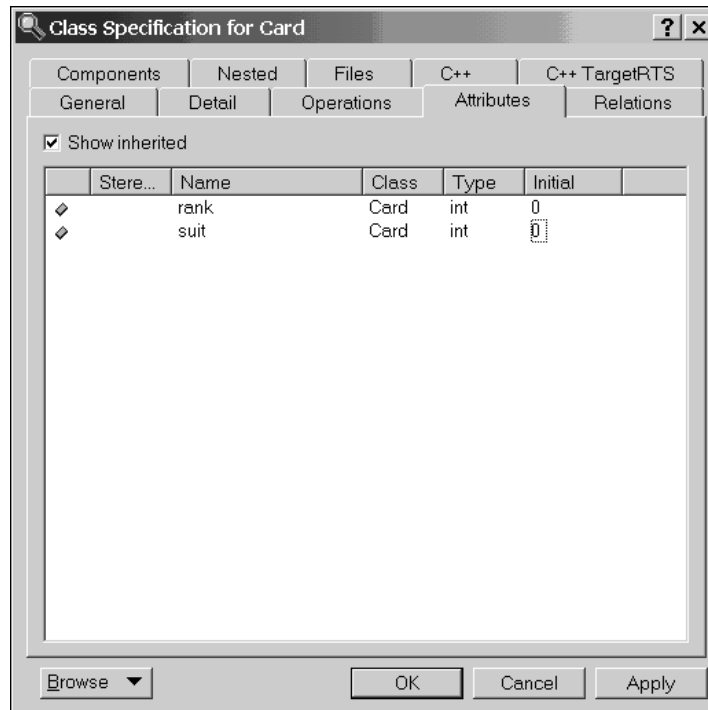
Adding Attributes to the Card Class

The card class represents a playing card, and is identified by a suit (Heart, Diamond, Spade, Club) and rank (Ace through King).

To create the suit and rank attributes

- 1 From the **Main Class Diagram** for **CardDefinitions**, double-click the **Card** class.
The **Class Specification for Card** dialog box appears.
- 2 Click the **Attributes** tab.
- 3 Right-click in the **Attributes** list, and click **Insert**.
A new attribute appears in the list.
- 4 Rename the attribute to **rank**.
- 5 Use the **TAB** key to move to the **Type** column, press **F8**, then select **int**.
- 6 Tab to the **Initial** column, press **F8**, then type **0**.
- 7 Right-click on the **rank** attribute, and click **Open Specification**.
- 8 Click the **General** tab.
- 9 In the **Visibility** area, select **Public**.
- 10 Click **OK**.
- 11 Repeat steps 3 through 10 to create a **suit** attribute of type **int**. Set the initial value to **0**.

When finished, the **Attributes** tab on the **Class Specification for Card** dialog box will look like the following:



12 Click OK.

Adding Details to the CardList Class

The responsibility of the **CardList** class is to manage the memory and access to a list of cards of any size. The **Hand** and **Deck** classes require card lists of different sizes.

The following table describes the attributes and operations that you will create to implement the behavior of the **CardList** class.

Note: Previously, you created a relationship between **CardList** and **Card** with the association end name of **_contents**. When the C++ code generator generates the source code, it creates an attribute in the **CardList** class named **_contents**.

The following table identifies the attributes and operations that you will create.

Attribute/ Operation	Description
_size	This attribute holds the number of cards that the list contains.
CardList	The CardList constructors allocate memory and initialize the list of cards.
~CardList	The CardList destructor returns the memory allocated for the cards.
size()	Returns the value of the private _size attribute.
operator[]	Allows access to the cards in the list.
operator=	Allows assigning a card list to another.

Note: The following steps show you how to add attributes and operations using the Specification dialogs; however, you can also use the Attribute and Operation wizards. Right-click on an element, such as a class or capsule) and click Attribute Tool to add or modify an attribute, or click Operation Tool to add or modify an operation.

To create the attribute **_size**:

- 1 In the **Model View** tab of the browser, expand **Logical View** and double-click **CardList**.

The **CardList Specification** dialog box appears.

- 2 Click the **Attributes** tab.
- 3 Right-click in the **Attributes** list, and click **Insert**.

A new attribute appears in the list.

- 4 Rename the attribute to **_size**.
- 5 Use the **TAB** key to move to the **Type** column, press **F8**, then select **int**.
- 6 Tab to the **Initial** column, press **F8**, then set the initial value to **0**.
- 7 Click **OK**.

To create the **CardList** constructors

The **CardList** class will have two constructors, a default constructor and a copy constructor. By default, Rose RealTime generates default constructors, destructors, and assignment operators for classes (configured in the **C++** tab). You may have to write your own version of these operations because they have specialized behavior

that cannot be automatically generated by the code generator. When you define these operations, the code generator determines that you created your own, and does not generate default versions of these operations.

Since the **CardList** class will contain a pointer to a list of **Card** objects, you will create the copy constructor, destructor, and assignment operator operations, with no memory leaks.

- 1 In the **Main Class Diagram** for **CardDefinitions**, double-click the **CardList** class.

The **CardList Specification** dialog box appears.

- 2 Click the **Operations** tab.
- 3 Right-click in the **Operations** list, and click **Insert**.

A new operation appears in the list.

- 4 Rename the operation to **CardList**.
- 5 Double-click on the **CardList** operation.

The **Operation Specification** dialog box appears.

- 6 Click the **Detail** tab.
- 7 Right-click in the parameters list, and click **Insert**.

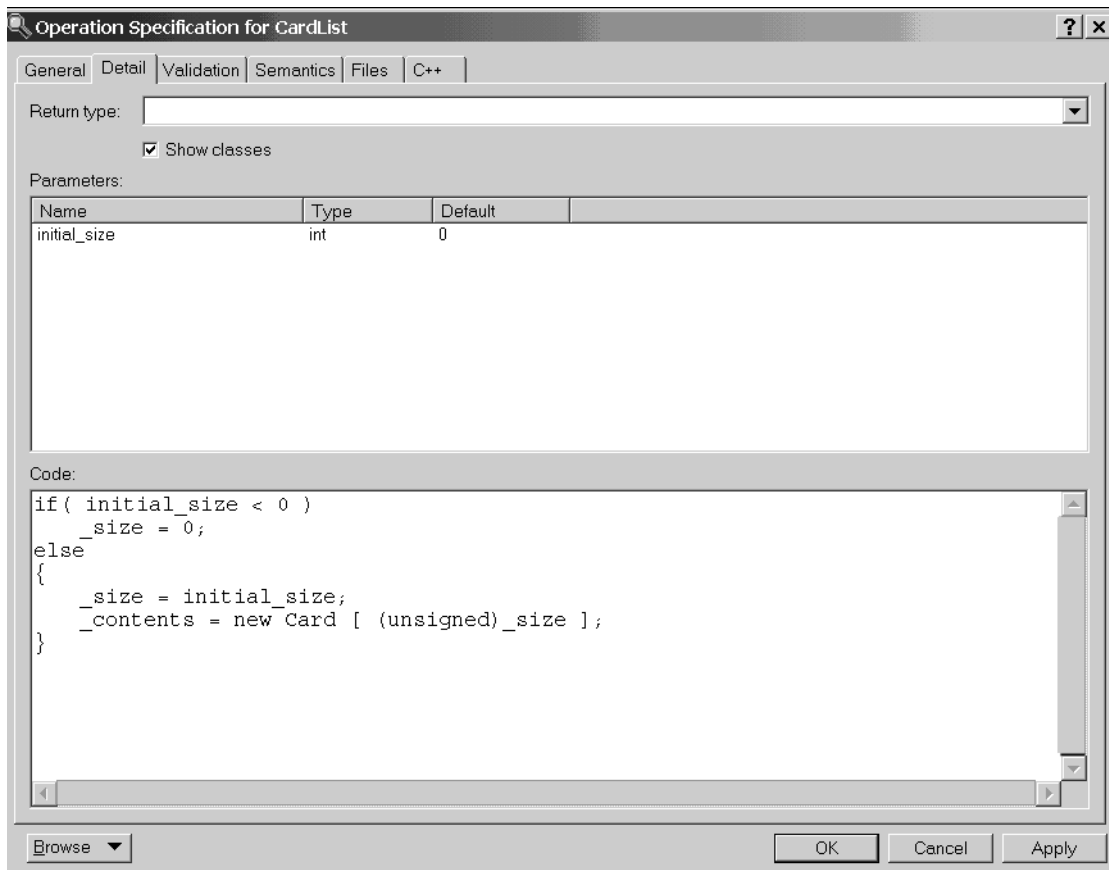
A new parameter appears in the list.

- 8 Rename the parameter to **initial_size**.
- 9 Press **TAB** to advance to the **Type** column, press **F8**, then select **int**.
- 10 Press **TAB** to advance to the **Default** column, press **F8**, then type **0**.

- 11 In the **Code** box, type the following C++ code:

```
if( initial_size < 0 )
    _size = 0;
else
{
    _size = initial_size;
    _contents = new Card [ (unsigned)_size ];
}
```

When finished, your **Detail** tab will look like the following:

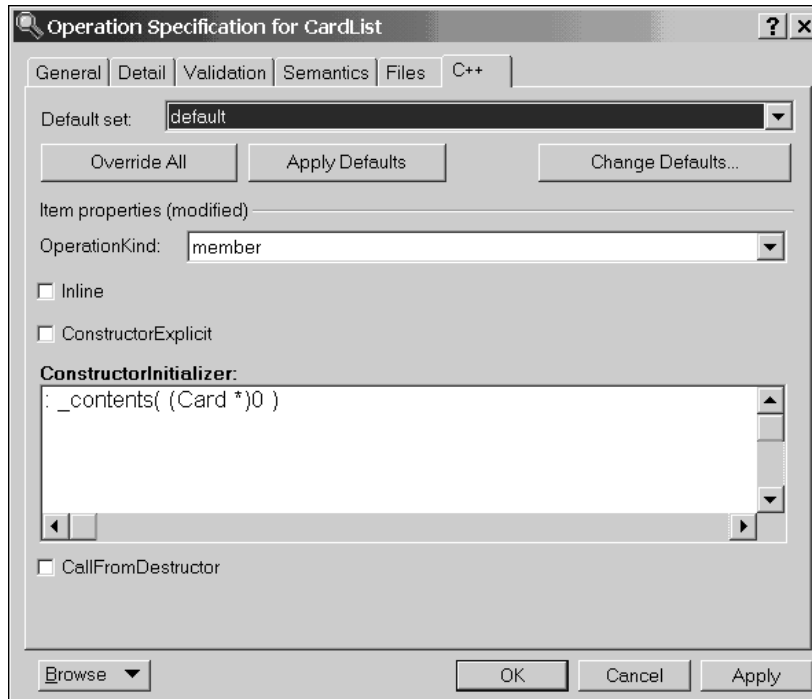


12 Click the C++ tab.

13 Type the following into the **ConstructorInitializer** box:

```
: _contents( (Card *)0 )
```

Note: **ConstructorInitializer** provides the initialization parameters for this operation if it is a constructor. It controls the initialization of parent classes and member variables. Be sure to add the colon to start the initializer list.



14 Click **OK** to save the changes, and to return to the **Operations** tab for **CardList**.

Next, you will create the copy constructor with one parameter called **other** of type **const CardList &**.

15 On the **Operations** tab, right-click and click **Insert**.

A new operation appears in the list.

16 Rename the operation to **CardList**.

17 Double-click on the **CardList** operation.

The **Operation Specification** dialog box appears.

18 Click the **Detail** tab.

19 Right-click in the **Parameters** list, and click **Insert**.

A new parameter appears in the list.

20 Rename the parameter to **other**.

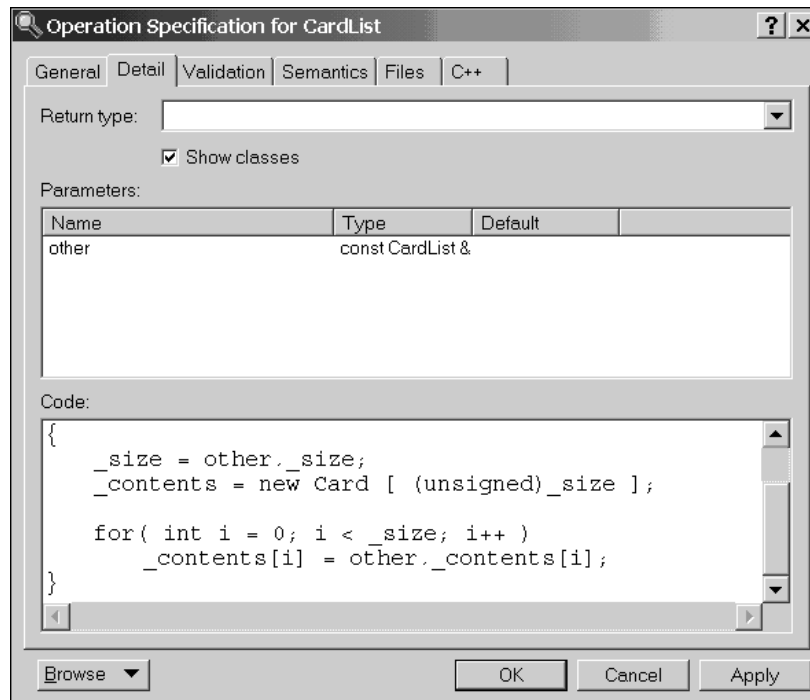
21 Press **TAB** to advance to the **Type** column, press **F8**, then type **const CardList &**.

22 In the **Code** box, type the following C++ code to the copy constructor:

```
if( other._size < 0 )
    _size = 0;
else
{
    _size = other._size;
    _contents = new Card [ (unsigned)_size ];

    for( int i = 0; i < _size; i++ )
        _contents[i] = other._contents[i];
}
```

The **Detail** tab will look like the following:



23 Click the **C++** tab.

24 Type the following into the **ConstructorInitializer** box for this copy constructor:

```
: _contents( (Card *)0 )
```

25 Click **OK** to save the changes, and to return to the **Operations** tab for **CardList**.

To create the destructor:

Because memory is allocated when a **CardList** is created, it is important to free that memory in the destructor.

1 On the **Operations** tab, right-click and click **Insert**.

A new operation appears in the list.

2 Rename the operation to **~CardList**.

3 Double-click on the **~CardList** operation.

The **Operation Specification** dialog box appears.

- 4 Click the **Detail** tab.

The destructor takes no arguments and does not return a value.

- 5 In the **Code** box, type the following C++ code to the copy destructor:

```
delete [] _contents;
```

- 6 Click **OK** to save the changes, and to return to the **Operations** tab for **CardList**.

To create the remaining **CardList** operations:

- 1 On the **Operations** tab, right-click and click **Insert**.

A new operation appears in the list.

- 2 Rename the operation to **size**.

- 3 Double-click on the **size** operation.

The **Operation Specification** dialog box appears.

- 4 Click the **General** tab.

- 5 Click **Query** to make this operation **const**.

Selecting **Query** indicates that the operation is read-only and does not modify the object's state.

- 6 Click the **Detail** tab.

- 7 In the **Return Type** box, select **int**.

- 8 In the **Code** box, type the following C++ code:

```
return _size;
```

- 9 Click **OK** to save the changes, and return to the **Operations** tab for **CardList**.

- 10 On the **Operations** tab, right-click and click **Insert**.

A new operation appears in the list.

- 11 Rename the operation to **operator[]**.

- 12 Double-click on the **operator[]** operation.

The **Operation Specification** dialog box appears.

- 13 Click the **Detail** tab.

- 14 In the **Return Type** box, type **Card &**.

- 15 Right-click in the parameters list, and click **Insert**.

A new parameter appears in the list.

16 Rename the parameter to **index**.

17 Press TAB to advance to the **Type** column, press F8, then type **const int**.

18 In the **Code** box, type the following C++ code:

```
return _contents[ index ];
```

19 Click **OK** to save the changes, and to return to the **Operations** tab for **CardList**.

20 On the **Operations** tab, right-click and click **Insert**.

A new operation appears in the list.

21 Rename the operation to **operator=**.

22 Double-click on the **operator=** operation.

The **Operation Specification** dialog box appears.

23 Click the **Detail** tab.

24 In the **Return Type** box, type **CardList &**.

25 Right-click in the parameters list, and click **Insert**.

A new parameter appears in the list.

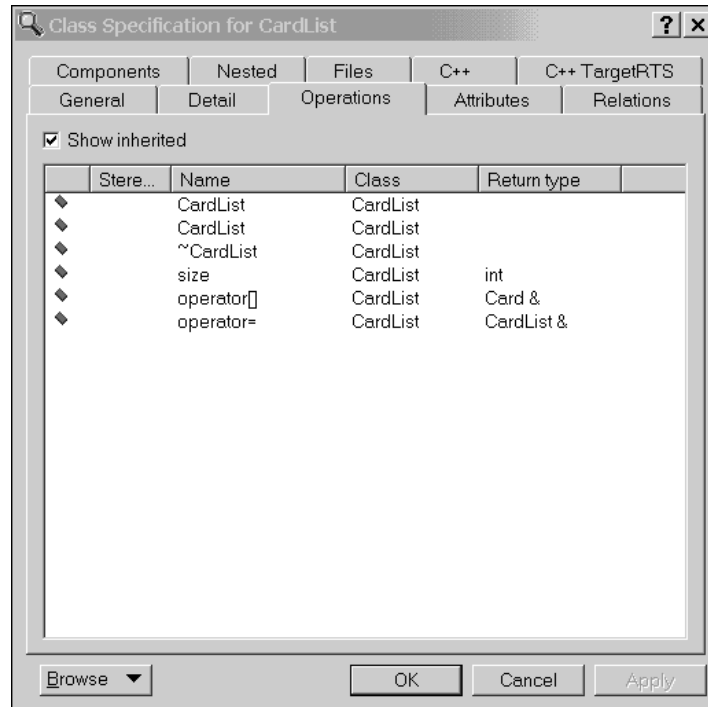
26 Rename the parameter to **rhs**.

27 Press TAB to advance to the **Type** column, press F8, then type **const CardList &**.

28 In the **Code** box, type the following C++ code:

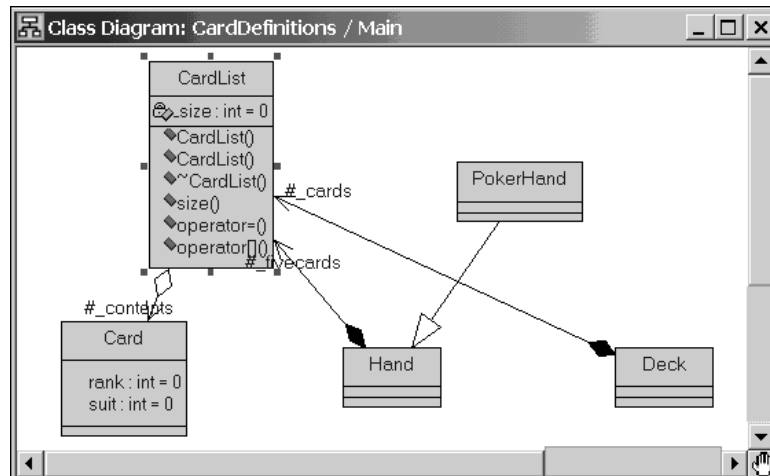
```
if( this != &rhs )
{
    delete [] _contents;
    _size      = rhs._size;
    _contents = new Card [ (unsigned)_size ];
    for( int i = 0; i < _size; ++i )
        _contents[i] = rhs._contents[i];
}
return *this;
```


When finished, your Class Specification for CardList will look like the following:



29 Click OK.

The Class Diagram for CardDefinitions looks like the following:



30 Click OK.

Note: At this point in the lesson, we recommend that you save, then build your model. If there are compile errors, double-click on an error message on the **Build Errors** tab. Rose RealTime opens the appropriate editor where the source of the error appears. You can then resolve any errors.

Generating Code for the Association Ends

An association is a relationship among two or more classes. Code can be produced to efficiently traverse the relationship in neither, one, or both directions.

The end of each association is called an **association end** or an **end**. You can label ends with an identifier that describes the role that an associate class plays in the association. An end has both model and language properties that affect the generated code it receives. For example, marking an end *navigable* means that code traveling from the opposite role's class to this end's class is to be implemented. Depending on the direction in which the generated code travels, the ends may be designated as the client end and the supplier end, respectively.

By default, an association relationship is represented in code as an **attribute**. In the simplest case, when each client class is associated with exactly one supplier class, the default code that is generated consists of an implementation attribute with a type that is the supplier class and the name is based on the value of the association end name property.

A number of factors affect the actual code generated for an association relationship. The type of the data member is affected by:

- The cardinality and **containment** adornments of the association relationship.
- Visibility adornments.
- The containment of the association in the target class can be by value (composition) or by reference (aggregation).

Encoding and Decoding by the Services Library

The **CardList** class implements the correct behavior, and is well-formed (that is, no memory leaks). However, there is one thing missing.

When running a model, it is often useful to observe the value of attributes, and modify them at runtime. To allow this, the Services Library must know how to encode and decode the object. It does this by using the classes type descriptor that describes the contents of the class to the Services Library.

Suggested Reading

- Watch window, *Rational Rose RealTime Toolset Guide*
- Type descriptors, *Rational Rose RealTime C++ Reference*

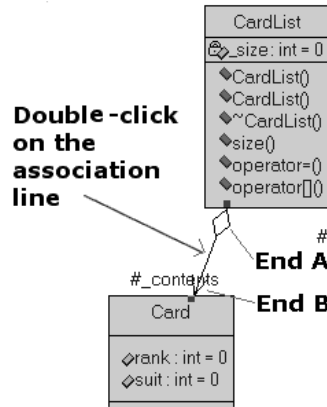
Encoding and Decoding

The Services Library can encode and decode objects that do not contain attributes that are pointers. However, when attributes are pointers, the Services Library can not determine how many objects are pointed to by the pointer. For classes with pointers to encode and decode, you must include a special operation to inform the Services Library at runtime how many objects are referenced by the pointer.

Since the **CardList** class contains an attribute **_contents** that is a pointer to a **Card** object (modeled via the aggregation which was set by reference), you must define this special operation.

To create the special operation on the **_contents association end:**

- 1 Double-click the association line between the **Card** and **CardList** class to open the **Association Specification** dialog for the association.



- 2 Click the C++ TargetRTS B tab.

Effect on generated code: The end name is generated as a member of the class at the other end of the association. That is, if the class at End A is **Class A** and the class at End B is **Class B**, and the name of End A is **foo**, then **Class B** will have a member named **foo** of type **Class A**.

- 3 In the **NumElementsFunctionBody** box, type the following code:

```
return( source->size() );
```

NumElementsFunctionBody provides the body of the function which calculates the number of objects the pointer points to. If the body is empty, the pointer is assumed to point to only one object. This function is required to make attributes which are pointers to arrays observable in the execution monitors.

Note: The code entered into this property is added to an operation created by the code generator:

```
int _rtg_nefb_CardList__contents( const RTTypeModifier * modifier,
const CardList * source )
{
    return( source->size() );
}
```

4 Click OK.

The name of the operation is different for each attribute or association end that has the **NumElementsFunctionBody** defined. However, the signature is always defined the same way. The operation must return an **int** that represents the number of objects referenced by the pointer, and has a parameter called **source** that points to the object of the class that contains the attribute.

For the **CardList** class, you used the **size()** operation to return to the number of card objects the **_contents** pointer references.

Adding Details to the Deck Class

The following table describes the attributes and operations that you will create to implement the behavior of the Deck class.

Attribute/ Operation	Description
_top	Tracks the index of the next card taken from the deck.
Deck	The Deck constructor creates a deck with a specified number of cards.
shuffle	Shuffles the cards in the deck (for any size of deck).
init	Orders all cards in the deck by suit, then by rank.
get	If the deck is not empty, it removes a card from the top of the deck.
size	Returns the initial size of the deck.

To create the Deck attribute and operations

- 1 In the **Model View** tab in the browser, under **Logical View**, expand **CardDefinitions** and double-click on **Main** to open the **Class Diagram**.

Or . . .

From the **Toolbar**, select the **Browse Class Diagram** button , then select **CardDefinitions** from the list.

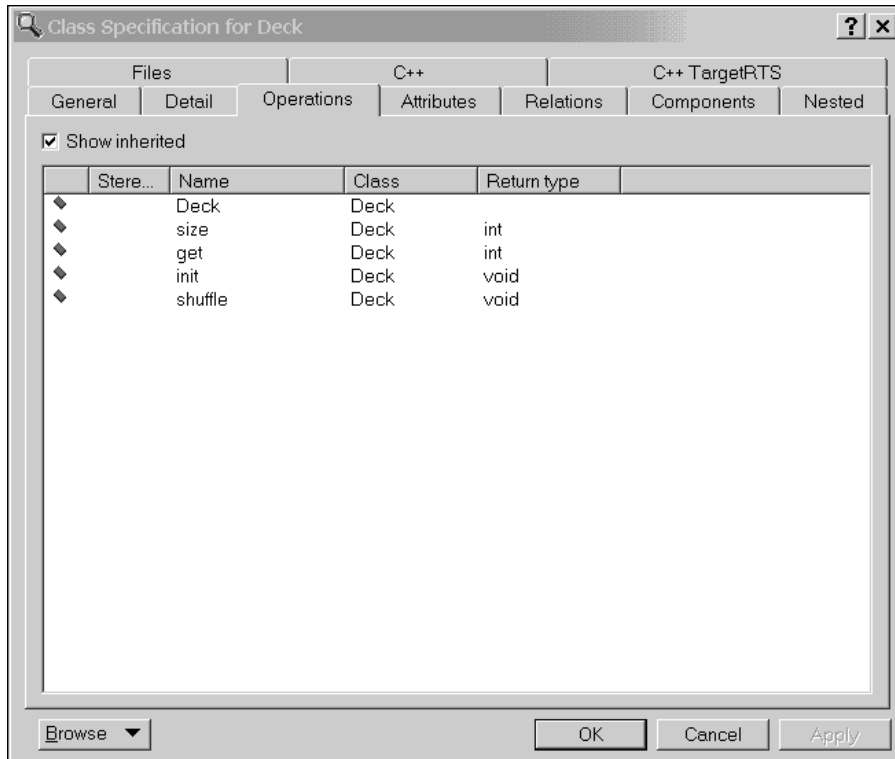
- 2 Double-click the **Deck** class.
- 3 Click the **Attributes** tab.
- 4 Right-click and click **Insert**.
- 5 Rename the attribute to **_top**.
- 6 Press **TAB** to advance to the **Type** column, press **F8**, then select **int**.
- 7 Press **TAB** to advance to the **Initial** column, press **F8**, then type **0**.
- 8 Click the **Operations** tab.
- 9 Right-click and click **Insert**.
- 10 Rename the operation to **Deck**.
- 11 Double-click the **Deck** operation.
- 12 Click the **Detail** tab.
- 13 In the **Parameters** box, right-click and click **Insert**.
- 14 Rename the parameter to **initial_size**.
- 15 Press **TAB** to advance to the **Type** column, press **F8**, then select **int**.
- 16 Click the **C++ tab** for Operations.
- 17 In the **ConstructorInitializer** box, type the following code:

```
: _cards( initial_size ), _top( 0 )
```
- 18 Click **OK**.

19 Follow steps 8 through 18 and add these operations:

Operation name	Return Type	Parameters	Parameter type	Code
size	int			<code>return _cards.size();</code>
get	int	card	Card &	<code>if(_top == size()) return 0; card = _cards[_top++]; return 1;</code>
init	void			<code>int suit = 1; for(int start = 0 ; start < size() ; start++) { _cards[start].rank = (start % 13) + 1; _cards[start].suit = suit; if(_cards[start].rank == 13) suit++; } _top = 0;</code>
shuffle	void			<code>// shuffle any size of card deck int swap = 150; int index1, index2; Card temp; int split = size() / 2; init(); srand((unsigned)time(NULL)); for(int i = 0 ; i < swap ; i++) { // pick two random cards index1 = rand() % split; index2 = rand() % (split * 2); // swap cards temp = _cards[index1]; _cards[index1] = _cards[index2]; _cards[index2] = temp; }</code>

When finished, your **Class Specification for Deck** dialog box will look like the following:



20 Click **OK**.

21 Click **OK**.

Adding Details to the Hand Class

The following table describes the attributes and operations that you will create to implement the behavior of the **Hand** class.

Attribute/ Operation	Description
add	Adds a card to the Hand at a specified offset within the hand.
get	Returns the specified card (at an offset within the hand).
Hand	The hand constructor creates a card hand with a specified number of cards.
size	Returns the number of cards in the hand.
value	It is meant to be overridden by subclasses to return the value of a given hand, which depends on the card game being played.

To create the Hand operations

- 1 In the **Class Diagram** for **CardDefinitions**, double-click the **Hand** class.
- 2 Click the **Operations** tab.
- 3 Right-click and click **Insert**.
- 4 Rename the operation to **add**.
- 5 Double-click the **add** operation.
- 6 Click the **Detail** tab.
- 7 In the **Return Type** box, select **int**.
- 8 In the **Parameters** box, right-click and click **Insert**.
- 9 Rename the parameter to **card**.
- 10 Press **TAB** to advance to the **Type** column, press **F8**, then type **const Card &**.
- 11 In the **Parameters** box, right-click and click **Insert**.
- 12 Rename the parameter to **index**.
- 13 Press **TAB** to advance to the **Type** column, press **F8**, then select **int**.

14 In the **Code** box, type the following code:

```
if( index >= _fivecards.size() )
    return 0;
else
    _fivecards[ index ] = card;
return 1;
```

15 Click the **General** tab.

16 In the **Visibility** area, select **Public**.

17 Click **OK**.

18 Follow the steps above to add the following operations to the **Hand** class:

Name	Return Type	Parameter name	Parameter type	Code	General tab settings
get	Card &	index	int	return _fivecards[index];	Public
Hand		initial_size	int		Public
size	int			return _fivecards.size();	Public Query
value	int			return 0;	Public Polymorphic

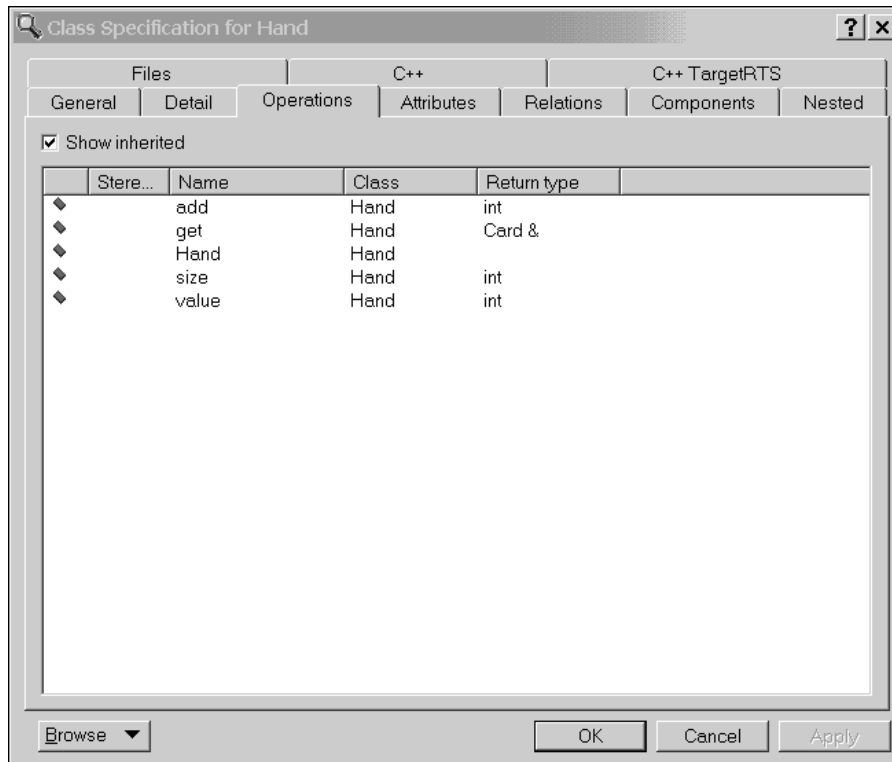
Note: Setting the size operation to **Query** makes the operation **const**. Setting the value operation to **Polymorphic** makes the operation virtual.

19 Double-click the **Hand** operation and click the **C++** tab.

20 Set the **ConstructorInitializer** property to:

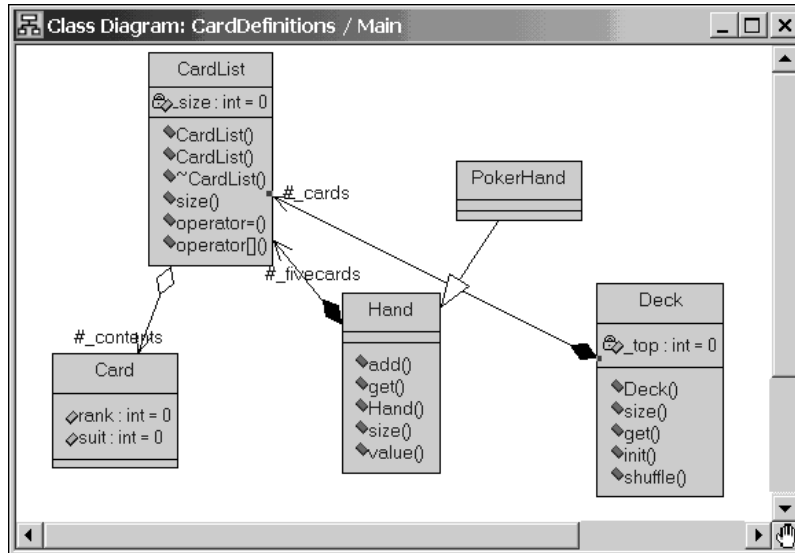
```
: _fivecards( initial_size )
```

When completed the **Class Specification for Hand** dialog box will look like the following:



21 Click **OK**.

The **Class Diagram** for **CardList** looks like the following:



Adding Details to the **PokerHand** Class

The following table describes the attributes and operations that you will create to implement the behavior of the **PokerHand** class.

Attribute/ Operation	Description
PokerHand	The PokerHand constructor creates a hand that contains five cards.
value	Overrides the virtual Hand operation to calculate the value of a five card poker hand with no wild cards.

To create the **PokerHand** operations

- 1 In the **Class Diagram** for **CardDefinitions**, double-click the **PokerHand** class.
- 2 Click the **Operations** tab.
- 3 Right-click and click **Insert**.
- 4 Rename the operation to **PokerHand**.
- 5 Double-click the **PokerHand** operation.
- 6 Click the **C++** tab.
- 7 Set the **ConstructorInitializer** property to:

```
: Hand( 5 )
```

- 8 Click **OK**.
- 9 On the **Operations** tab, right-click and click **Insert**.
- 10 Rename the operation to **value**.

Note: The **Operations** tab already contains an operation called value. The existing value operation is for the **Hand** class. The value operation that you are adding is for the **PokerHand** class.

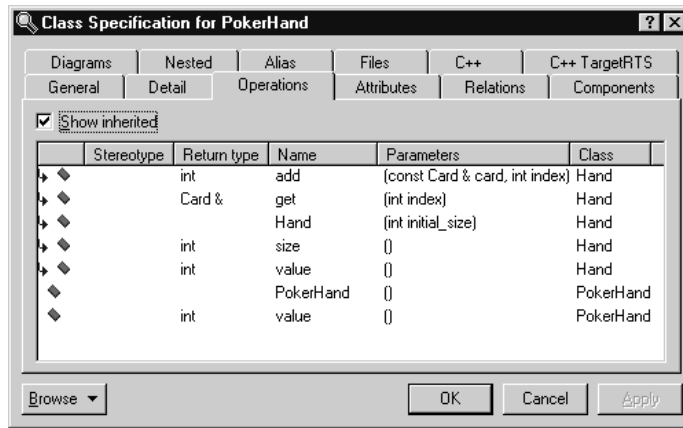
- 11 Double-click the **value** operation.
- 12 Click the **Detail** tab.
- 13 In the **Return Type** box, select **int**.
- 14 In the **Code** box, type the following code:

```
// Evaluates a hand and returns the following results:
// 1 - one pair
// 2 - two pairs
// 3 - three of a kind
// 4 - full house
// 6 - four of a kind

int result = 0;
for( int i = 0 ; i < size() ; i++ )
{
    for( int j = i + 1; j < size() ; j++ )
    {
        if( _fivecards[i].rank == _fivecards[j].rank )
            result++;
    }
}
return result;
```

- 15 Click the **General** tab.
- 16 In the **Options** area, select **Polymorphic** to make this operation virtual.
- 17 Click **OK**.

The Class Diagram for CardList looks like the following:



18 Click **OK**.

Note: We recommend that you save, then build your model. If there are compile errors, double-click on an error message on the **Build Errors** tab. Rose RealTime opens the appropriate editor where the source of the error appears. You can then resolve any errors.

Review

The goal of Lesson 8 was to create the classes used by the **Dealer** and **Player** capsules to implement the card simulation. The classes represent the information used in the system, namely the data that describes the cards used in a card game.

Rose RealTime allowed you to add all implementation details to the classes from within the toolset.

The key points to remember when class modeling are:

- **Classes must be well-formed.** The developer is responsible for ensuring that their classes do not leak memory. Rose RealTime can generate default copy constructors, assignment operators, and destructors for classes. However, these default operations only work with simple classes, that is, classes without pointers (dynamic memory). Most often, you will have to implement the copy constructor, assignment operator, and destructor.
- **NumFunctionElementsBody** must be defined for all pointer (by reference) attributes/association ends observed at runtime. This allows the Services Library to decode the objects references by the pointer.

- **ConstructorInitializer** allows you to add attributes to the initializer list of any constructor.
- **Association ends** that are named are generated as attributes of the target class. If a navigable association end is not named, the code generator does not generate an attribute.

Next, you will use these classes to finish the card game simulation.

Note: If you do not want to proceed to Lesson 9 using the file you created at the end of Lesson 8, you can open the file `<ROBERT_HOME>/Help/Tutorials/cardgame/cardgame_step3.rtmidl`, and then continue with the steps in Lesson 9.

Lesson 9: Adding Card Classes to the Capsule Behavior

In the first iteration of the card game simulation, you did not use any cards. The Dealer and Player simply exchanged signals with no data. You will extend the capsule behavior to use the newly developed card classes. The **ACard** signal was originally sent without data, but now the Player and Dealer capsules will send cards to one another.

To add the Card class to the ACard signal:

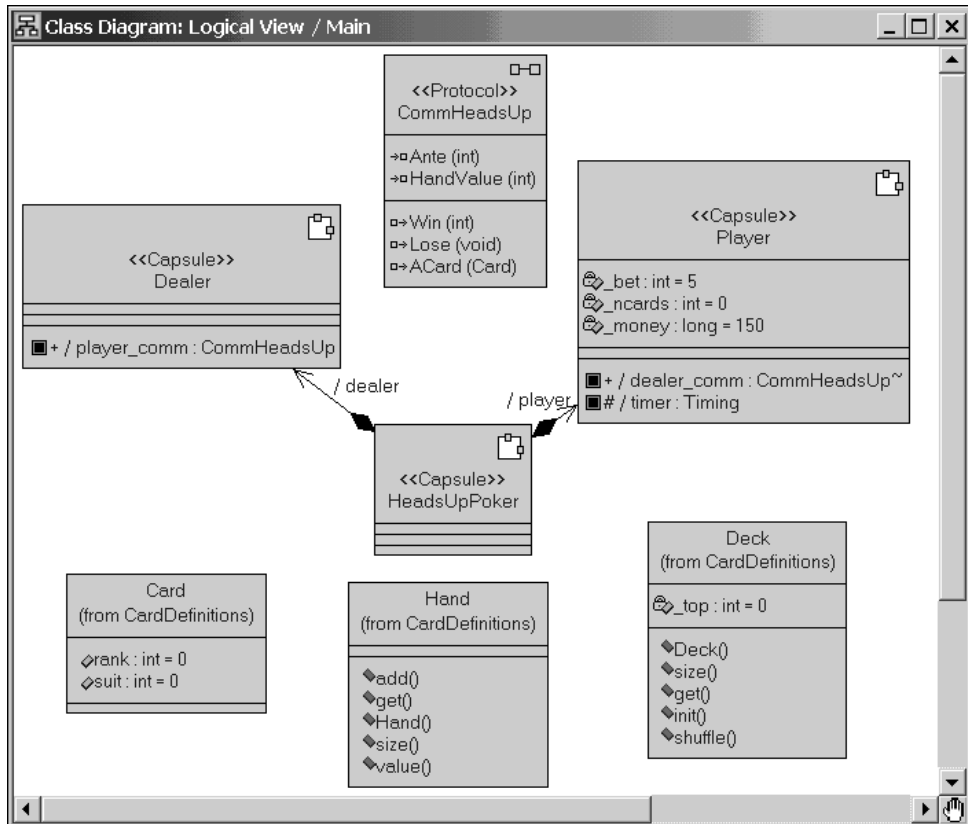
- 1 Open the **CommHeadsUp** protocol (Logical View), and click the **Signals** tab.
- 2 In the **Data Class** column for the **ACard** out signal, press **F8** and change the data class from **void** to **Card**.
- 3 Click **OK**.

Completing the Dealer Capsule Behavior

In the scenario description of the use case defined for the card game simulation, the Dealer possesses the deck from where the cards are dealt, as well as the Dealer hand.

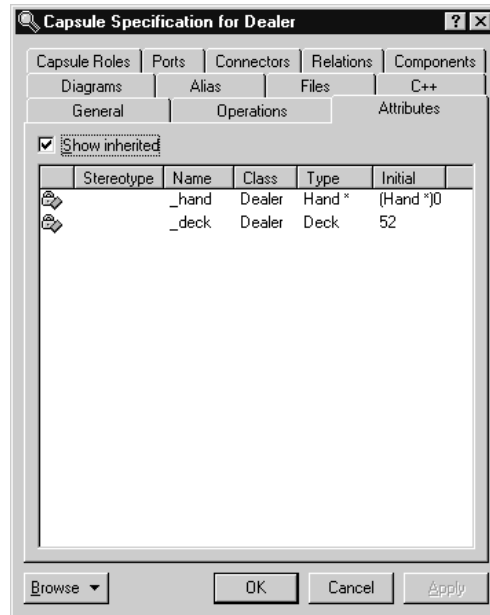
To add the deck and hand attributes to the Dealer:

- 1 Open the **Capsule Specification for Dealer** dialog box.



- 2 Click the **Attributes** tab.
- 3 Insert an attribute called **_hand** of type **Hand *** that has an initial value of **(Hand *)0**.
- 4 Insert an attribute called **_deck** of type **Deck** that has an initial value of **52**.

Adding the initial value to a capsule attribute initializes the attributes when the capsule is created.

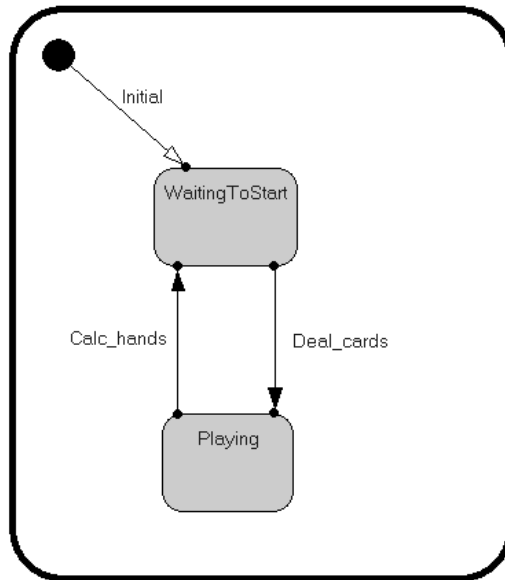


5 Click **OK**.

To modify the state diagram (behavior)

Previously, only empty signals were sent from the Dealer to the Player. You will modify the Dealer's **State Diagram** to use the **Deck** and **Hand** classes to send cards from the deck, and to also maintain the Dealer hand. At the end of each game the Player sends the value of his hand to the Dealer. The Dealer then decides who wins based on the value of both hands.

- 1 Open the **State Diagram** for the **Dealer** capsule.



- 2 Double-click the **Initial** transition line, and click the **Actions** tab.
- 3 In the **Code** window, type the following code to initialize a new **PokerHand** and shuffle the deck:

```
_hand = new PokerHand;  
_deck.shuffle();
```
- 4 Click **OK**.

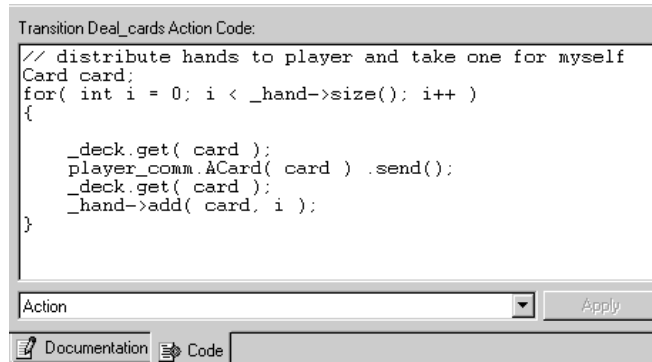
Now, you will use the **Code** window in the lower right-hand corner to perform the same task of adding code to transitions rather than using the **Specification** dialogs.

- 5 Select the **Deal_cards** transition line.
- 6 In the **Code** window, select **Action** from the drop-down list if it is not currently selected.

- 7 In the **Code** window, replace the existing code with the following code that determines who deals the cards:

```
// distribute hands to player and take one for myself
Card card;
for( int i = 0; i < _hand->size(); i++ )
{
    _deck.get( card );
    player_comm.ACard( card ).send();
    _deck.get( card );
    _hand->add( card, i );
}
```

Your **Code** window will look like the following:



- 8 Click **Apply**.

Note: You can either use the Code window or the Specification dialogs to type code for the remaining transitions. The tutorial lesson will continue using the Specification dialogs.

- 9 Double-click the **Calc_hands** transition line, and click the **Actions** tab.

- 10 In the **Code** box, replace the existing code with the following code that determines who won the game:

```
// compare the players hand to ours. If the Player has a better
// hand then the Player wins; otherwise, the Dealer wins.
if( *rtdata > _hand->value() )
    player_comm.Win(2).send();
else
    player_comm.Lose().send();

// shuffle deck for next game
_deck.shuffle();
```

- 11 Click **OK**.

Adding a Destructor to the Dealer Capsule

Because the **Dealer** capsule allocates memory from the heap using the **new** operation, you must return the memory. You will add a destructor operation to the **Dealer** capsule. The destructor on the capsule is called when a capsule instance is destroyed.

To add a destructor operation

- 1 Open the **Capsule Specification for Dealer** dialog, and click the **Operations** tab.
- 2 Right-click, select **Insert**, and rename the operation to **~Dealer**. Do not specify a return type.
- 3 Double-click the **~Dealer** operation.
- 4 In the **Code** box on the **Detail** tab, type the following code:

```
delete _hand;
```

- 5 Click **OK**.
- 6 Click **OK**.

Completing the Player Capsule Behavior

The player receives cards from the Dealer that become the Player's hand. You will modify the Player capsule to have a hand, and to give cards to the hand as they are received from the Dealer.

To add the Hand attribute to the player:

- 1 Open the **Capsule Specification for Player** dialog, and click the **Attributes** tab.
- 2 Insert an attribute called `_hand` of type **Hand *** that has an initial value of **(Hand *)0**.

Adding the initial value to a capsule attribute initializes the attributes when the capsule is created.

- 3 Click **OK**.

To modify the state diagram (behavior)

Only empty signals were sent from the Dealer to the Player. Now, you will modify the Player's **State Diagram** to use the **Hand** classes to hold the cards sent from the Dealer. At the end of each game, the Player sends the value of his hand to the Dealer.

- 1 Open the **State diagram** for the **Player** capsule.
- 2 Double-click the **Initial** transition line, and click the **Actions** tab.
- 3 In the **Code** box, type the following code to initialize a new PokerHand:

```
_hand = new PokerHand;
```
- 4 Click **OK**.
- 5 Double-click the **Received_card** transition line, and click the **Actions** tab.
- 6 In the **Code** box, replace the existing code with the following code that receives a card, and add the value to the hand:

```
_hand->add( *rtdata, _ncards++ );
```
- 7 Click **OK**.
- 8 Double-click the **All_cards** Choice Point, and click the **Condition** tab.
- 9 In the **Code** box, replace the existing code with the following code that determines whether the player received all cards from the Dealer:

```
return( _ncards >= _hand->size() );
```
- 10 Click **OK**.
- 11 Double-click the **Got_all_cards** transition line, and click the **Actions** tab.

- 12 In the **Code** box, replace the existing code with the following code that sends the value of the Player's hand to the Dealer:

```
// Send the value of the hand to the Dealer
dealer_comm.HandValue( _hand->value() ).send();
```

- 13 Click **OK**.

Using Attributes Versus Aggregations

You could have used aggregation between the capsules and the card classes instead of creating attributes. The decision of using associations versus attributes depends on your project guidelines.

Adding Dependencies

Before you can compile the model, you have to explicitly add dependencies between the capsules and classes. A dependency specifies that a class or capsule uses a target class. Dependency relationships are converted to C++ as either forward references or inclusions in header and implementation files.

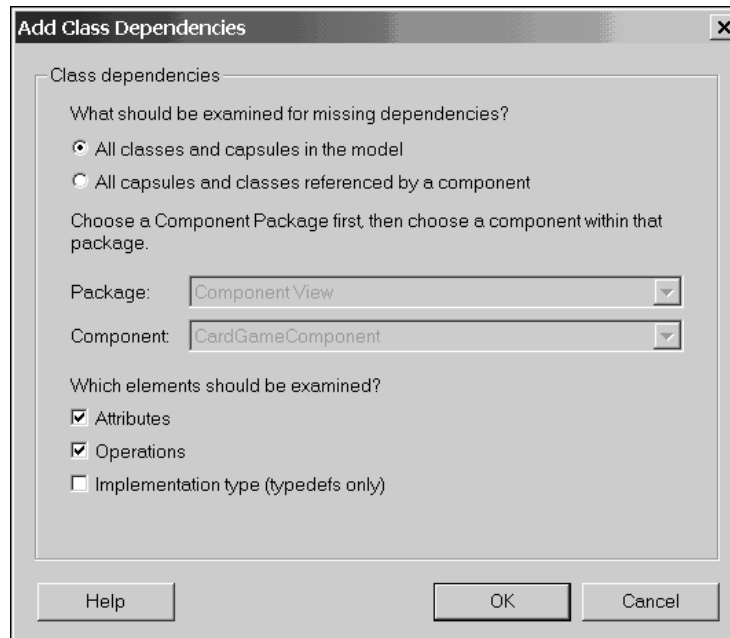
The **Add Dependencies** add-in can search a set of classes and capsules looking at attribute types, operation return values, and parameters and can automatically create dependencies. There are some dependencies that the add-in cannot find (such as, code dependencies).

If you compiled the model now, the compiler would report that certain types are not defined. When you are programming, if an inclusion or forward reference is missing, the compiler cannot resolve certain symbols defined in other files.

To add the dependencies:

- 1 On the **Build** menu, click **Add Class Dependencies** to create the dependencies.
- 2 In the **Add Class Dependencies** dialog box, select **All classes and capsules in the model**.

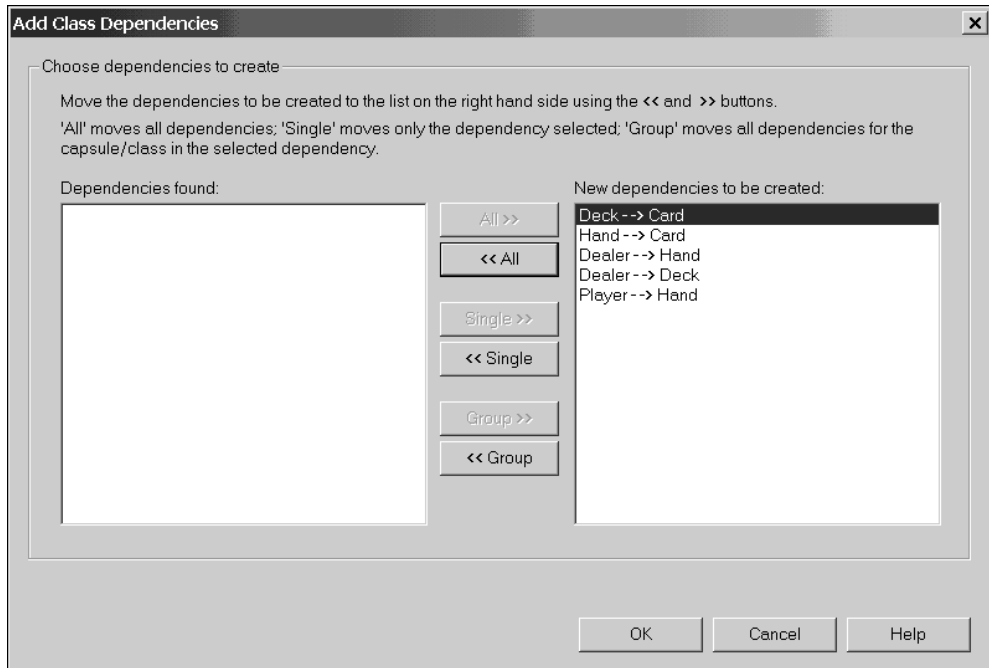
- 3 In the **Which elements should be examined?** box, click **Attributes** and **Operations** only.



- 4 Click **OK**.

A second **Add Class Dependencies** dialog box appears.

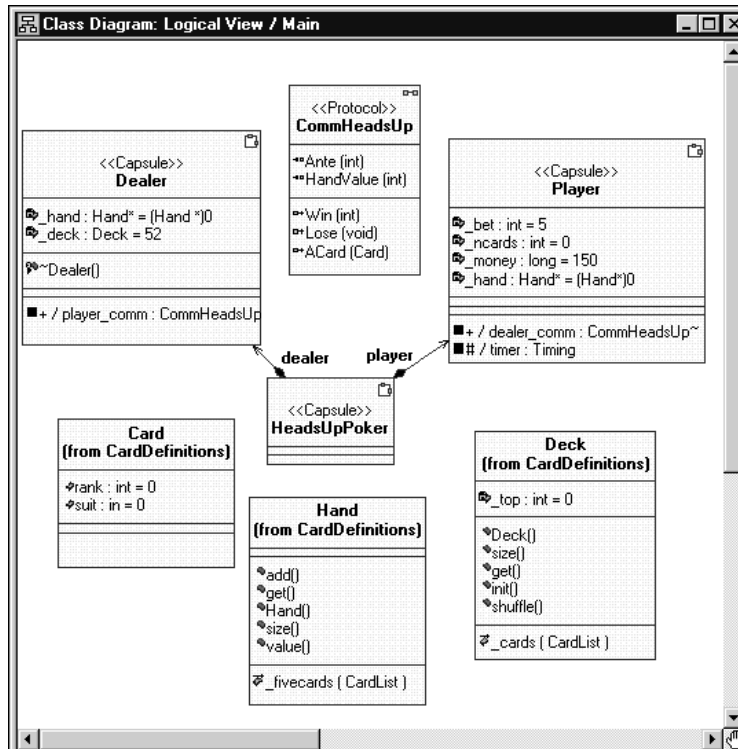
5 Click **ALL >>** to select all dependencies.



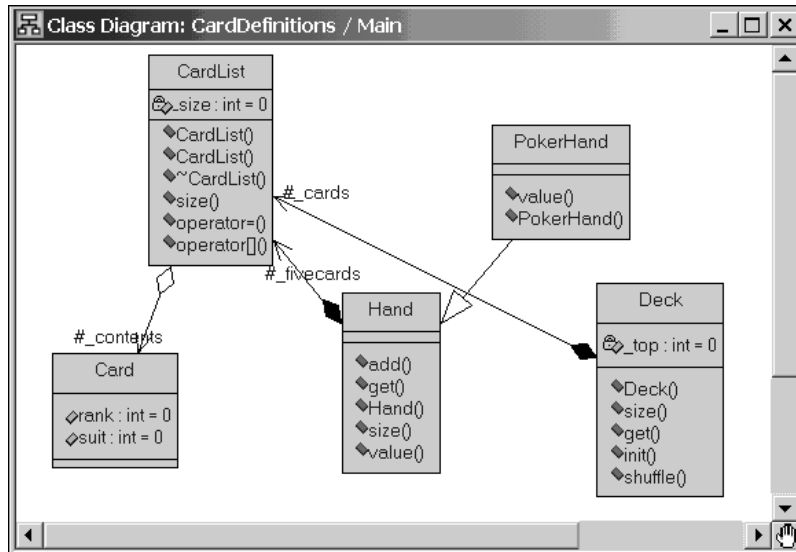
6 Click **OK**.

7 Click OK.

Note: The dependencies are added to the model by the **Add Class Dependencies** add-in. However they are not automatically added to class diagrams. When working with a model in source control or on a large model, you do not necessarily want to check out or modify diagrams because you add dependencies. In this tutorial, you add dependencies to diagrams to show that they have been created.



- 8 Open the **Main** class diagram in the **CardDefinitions** package.



- 9 On the **Query** menu, click **Filter Relationships**.

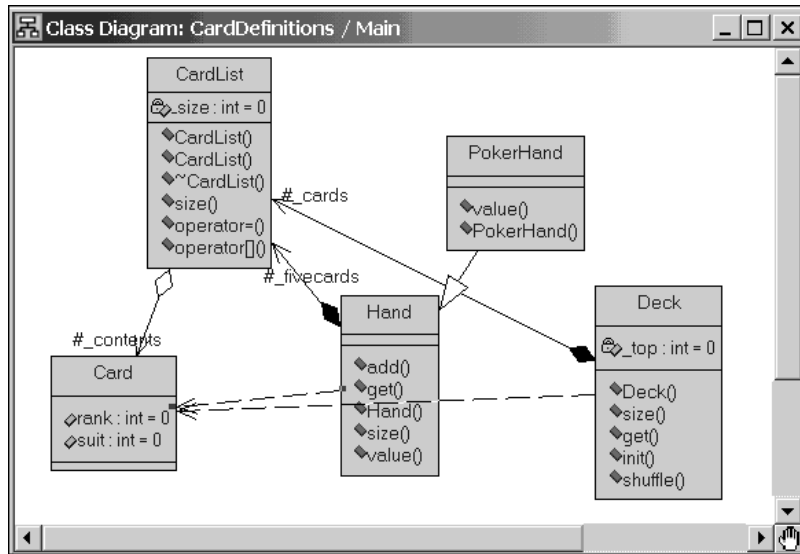
The **Relations** dialog appears.

- 10 In the **Type** box, ensure that **Dependency** is selected.




- 11 Click **OK**.

There were two dependencies created: **Deck to Card**, and **Hand to Card**.

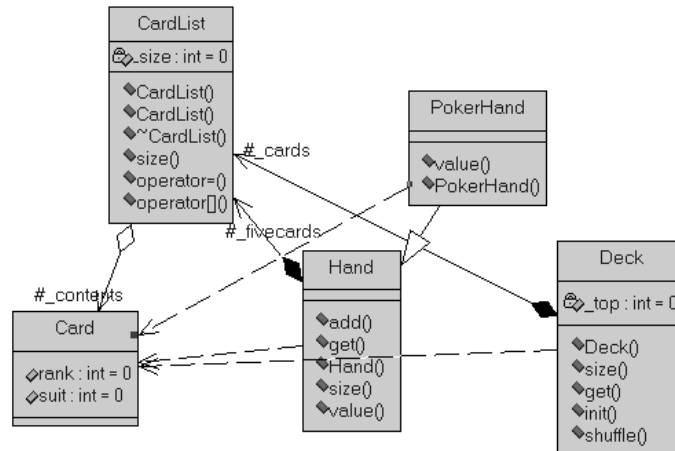


Note: There is a dependency that **was not** created by the **Add Class Dependency** add-in. This dependency is between the **PokerHand** and **Card** class.

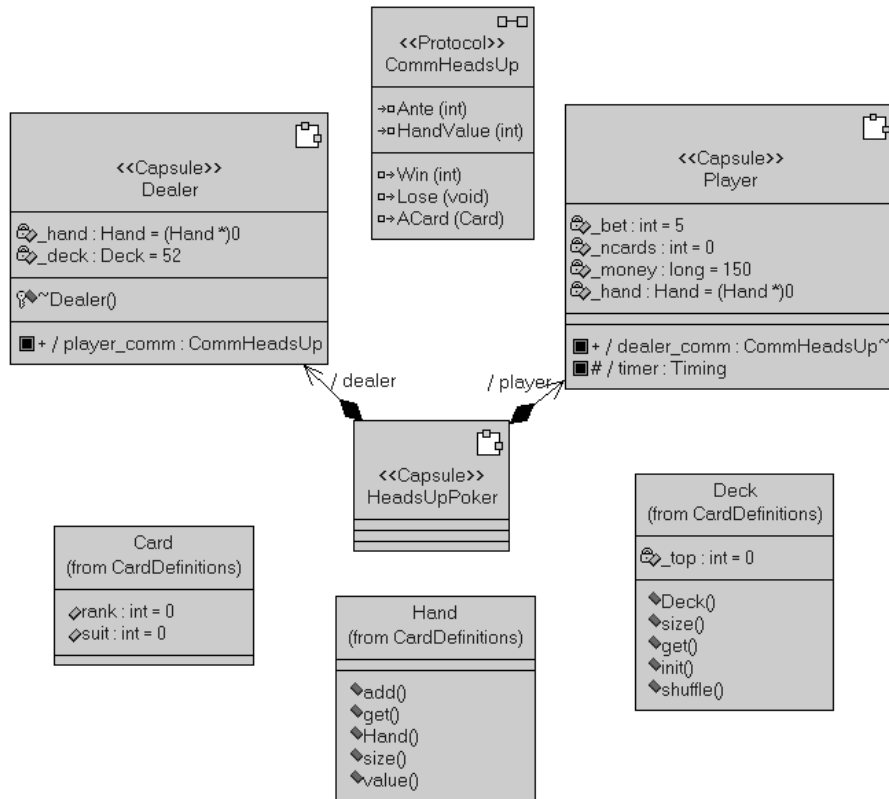
12 In the Toolbox, click the **Dependency** tool .


13 Click the **PokerHand** class and drag the mouse to the **Card** class to create a dependency between the **PokerHand** and **Card** class.


Your **Class Diagram** will look like this:



14 Open the **Main** class diagram directly contained in the **Logical View** package.



15 From the Toolbox, select the **Class** tool , and add a **PokerHand** class to the diagram.

16 From the Toolbox, select the **Dependency** tool , and create a dependency between ALL of the following:

- the **Dealer** capsule and the **Card** class
- the **Dealer** capsule and the **PokerHand** class
- the **Player** capsule and the **PokerHand** class

Now, you can use **Filter Relationships** to create the other dependencies.

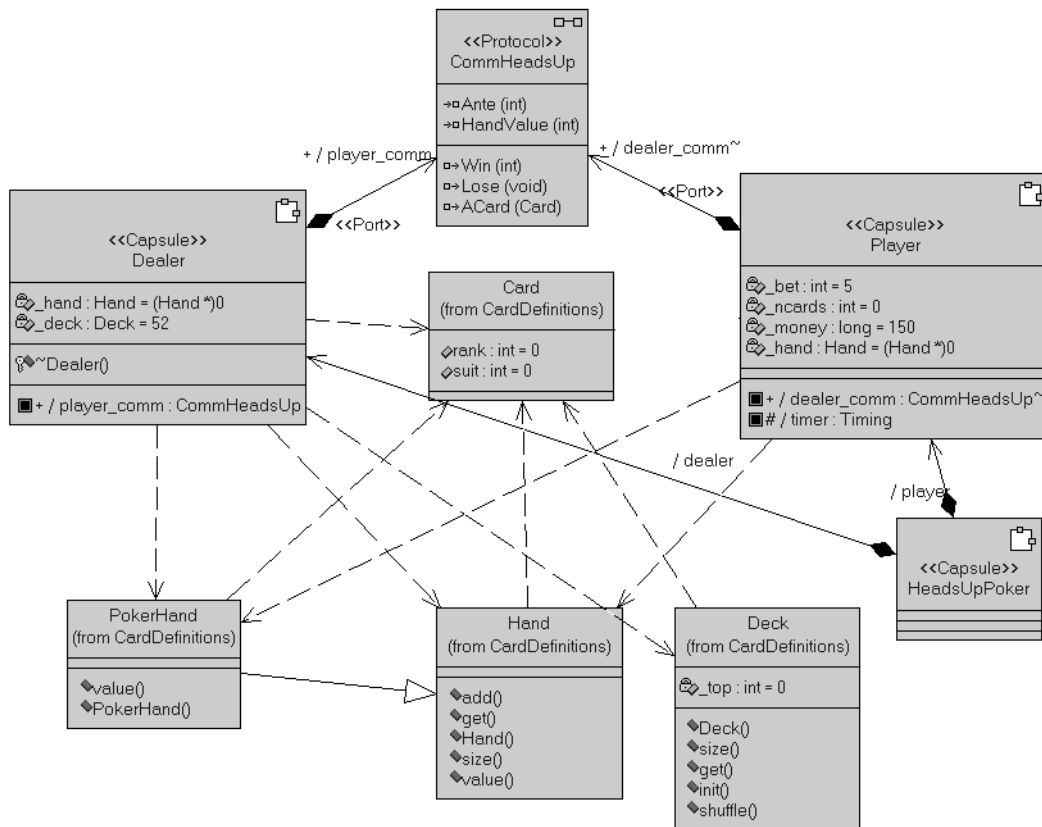
17 On the **Query** menu, select **Filter Relationships**.

18 Click **OK**.

The **Add Class Dependencies** add-in created the dependency between the **Dealer** and the **Deck** using the default C++ properties. Since the `_deck` attribute is declared by value in the **Dealer** capsule, you must specify the dependency between **Dealer** and **Deck** as **Inclusion** in the header, and **None** in the implementation.

Ensure that your diagram contains *all* of the dependencies contained in the following diagram.

Note: The objects in the following diagram were rearranged to make the diagram easier to read.



Note: The C++ compiler needs to know the size of the **Deck** class within the **Dealer** capsule during compilation. Forward references can only be used when attributes are declared as pointers.

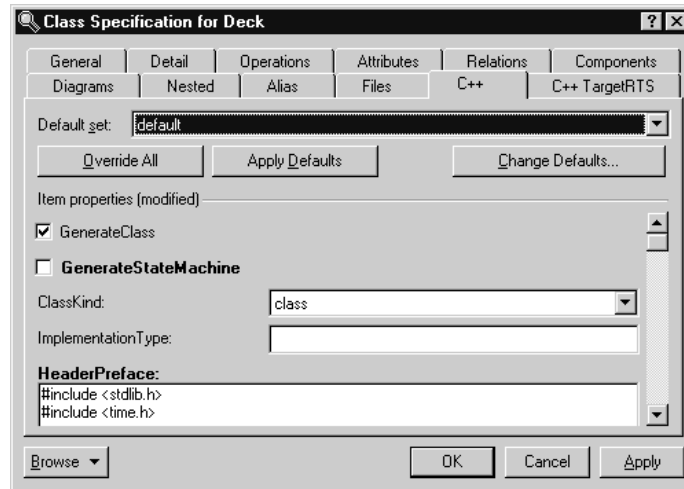
19 Double-click on the dependency between the **Dealer** capsule and the **Deck** class.

The **Dependency Specification** dialog box appears.

20 Click the C++ tab.

21 In the **KindInHeader** box, select **inclusion**.

22 In the **KindInImplementation** box, select **none**.



23 Click OK.

Dependency Properties

A dependency is converted by the code generator as an `#include` or forward reference directive in either the header or implementation files generated for the class or capsule. By default, a dependency generates a forward reference in the header file and an inclusion in the implementation. This kind of dependency is the most conservative in terms of keeping compilation dependencies to a minimum.

You can change how a dependency is generated (either as an include directive or a forward reference) from the dependency relationship properties in the **C++** tab.

Adding Inclusions

For the implementation of the **Deck**, you used services included in the system header files **time.h** and **stdlib.h**. You need to include those in the **Component Specification** so that when the component compiles, the header files are included.

Adding inclusions at the component level adds system-wide dependencies to your model. This may not be desirable because of the compile dependencies that it causes. An alternative to adding include directives to the component is to add them to capsules or classes.

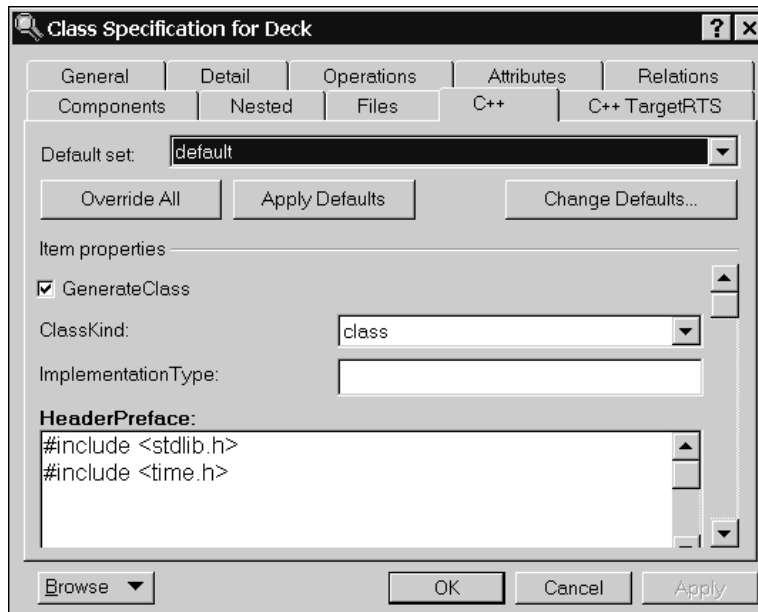
Note: Add include directives to the **C++** tab on both capsules and classes. Enter the `#include` directives in the **HeaderPreface** property.

To add capsule or class scoped inclusions:

- 1 Open the **Class Specification for Deck** dialog box, and click the **C++** tab.
- 2 In the **HeaderPreface** box, type the following `#include` statements.

```
#include <stdlib.h>
#include <time.h>
```

Your **Class Specification for Deck** dialog box will look like the following.

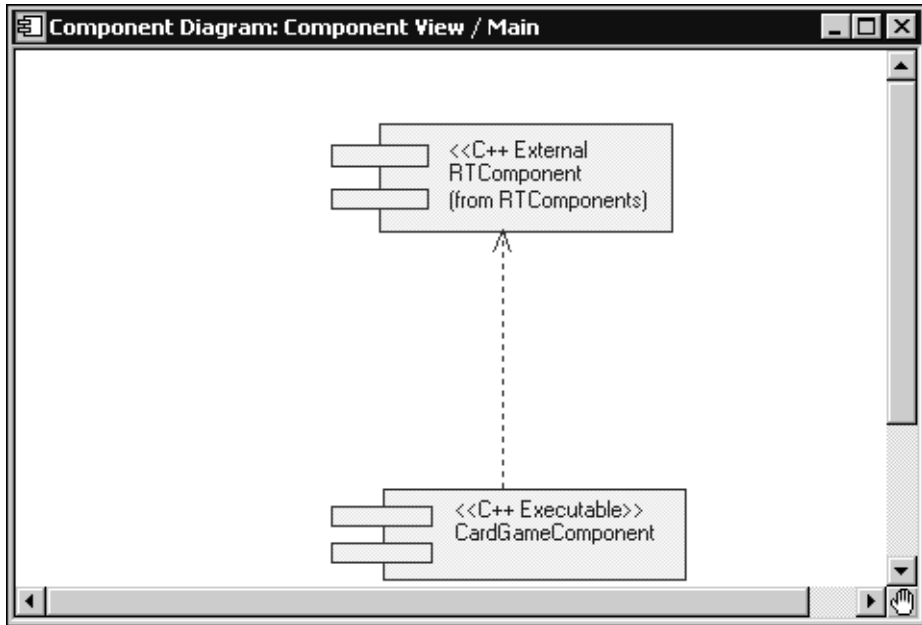


- 3 Click **OK**.

Next, you will next to create a dependency between two components: **CardGameComponent** and **RTComponent**.

- 1 On the **Model View** tab in the browser, open the Main diagram.
- 2 On the **Model View** tab, drag the **RTComponent** on to the diagram.
- 3 On the **Model View** tab, drap the **CardGameComponent** on to the diagram.

- 4 Select the **Dependency** tool.
- 5 Draw a dependency from the **CardGameComponent** to the **RTComponent**.



- 6 Save your model.

Building and Running the Card Game

Following the steps detailed in *Lesson 4: Building and Running* on page 106 and in *Lesson 7: Using Traces and Watches to Debug the Design* on page 147, build the **CardGameComponent**, and run the component instance.

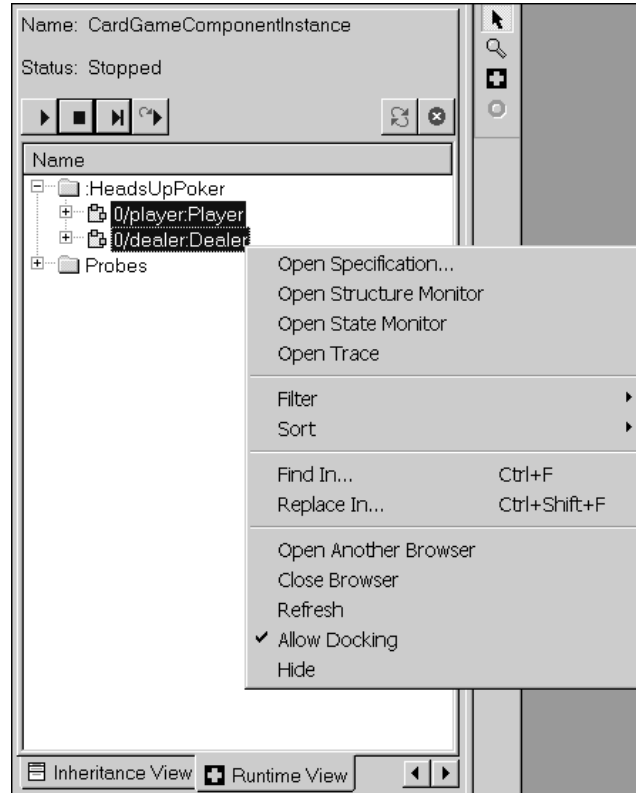
Create watches on the **_hand** attributes to verify that cards are being dealt to both the **Player** and **Dealer**, and trace the messages exchanged between the two capsule instances.

Note: You can make changes to model elements while a component instance is running, however, Rational Rose RealTime will shutdown after you make the change.

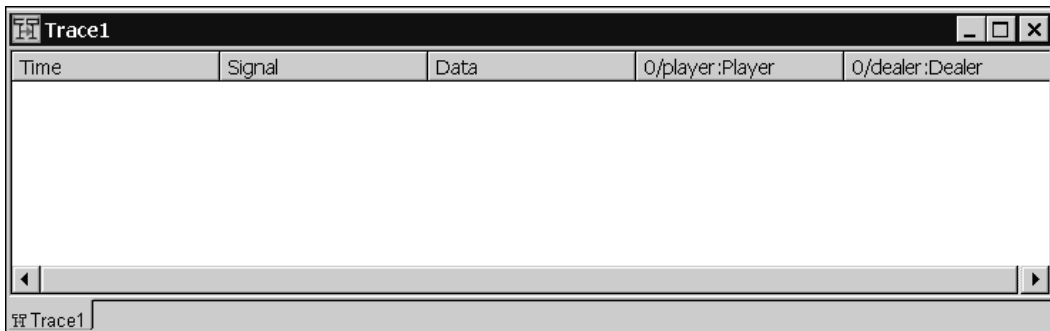
Now, try creating a trace on component instances to show messages being exchanged between the capsule instances.

To observe messages exchanged between capsule instances:

- 1 Save, then build and run the model.
- 2 On the **Runtime View** tab in the browser, select the **Player** and **Dealer** instances, right-click, then click **Open Trace**.




The **Trace** window displays.



A capsule trace window is a type of message trace that shows capsule instances with messages listed in separate columns for recording event flow between instances. The left column displays the time at which the message occurred, the subsequent columns display the source and destination ports, the signal name, optional data, and the capsule instances.

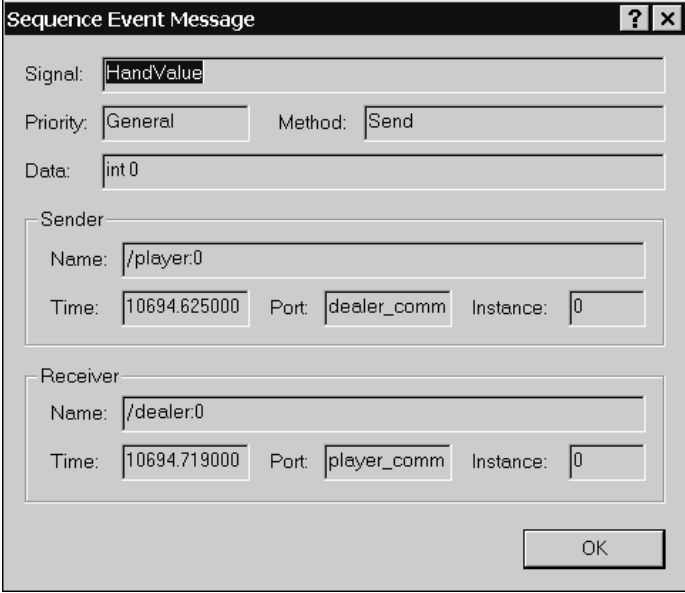
3 Either click the **Start** button, , or the **Step** button, , to begin.

4 In the **Trace** window, select a signal.

Note: If you clicked the **Start** button, you can click the **Stop** button  to pause the execution of the component instance.

5 Right-click and click **Open Specification**.

The **Sequence Event Message** dialog box for the selected signal appears.

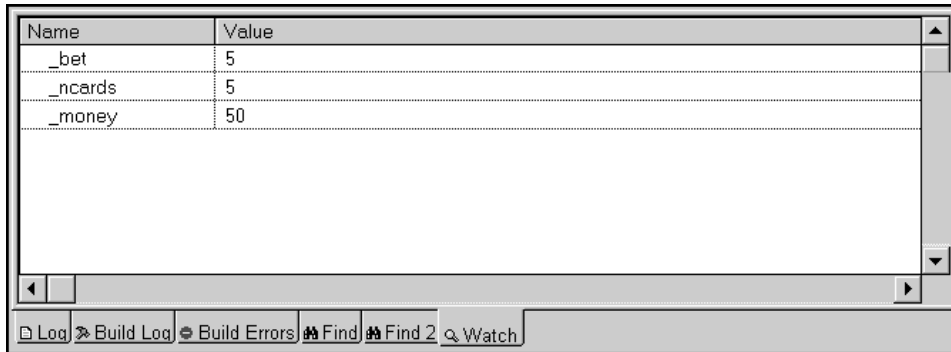


The image shows a dialog box titled "Sequence Event Message" with a question mark and close button in the title bar. The dialog contains several input fields:

- Signal: HandValue
- Priority: General
- Method: Send
- Data: int 0
- Sender section:
 - Name: /player:0
 - Time: 10694.625000
 - Port: dealer_comm
 - Instance: 0
- Receiver section:
 - Name: /dealer:0
 - Time: 10694.719000
 - Port: player_comm
 - Instance: 0
- OK button

You can view any **Sender** and **Receiver** information for your selected signal.

Also, take note of the value of **_money** in the **Watch** window. Depending on whether the **Player** wins or loses, the value increases or decreases for each hand.



Name	Value
_bet	5
_ncards	5
_money	50

We recommend that you save your work at this time.

Note: If you do not want to proceed to Lesson 10 using the file you created at the end of Lesson 9, you can open the file `<ROBERT_HOME>/Help/Tutorials/cardgame/cardgame_step4.rtmidl`, and then continue with the steps in Lesson 10.

Trace Summary

Typically when a message fails to flow through a set of capsules as expected, it is important to see where the message flow was first in error. To debug these kinds of errors, we can first use capsule instance traces to look at the messages originating and terminating from the capsules in the message flow. If the messages are incorrect and the fault origination cannot be identified, you can then place probes on specific ports in a composite capsule. Based on whether the messages are still faulty, you can narrow down the cause of the error by further subdivision. Once the faulty capsule has been identified, it is valuable to place traces and message breakpoints on the state machine.

Fixing Compilation Errors

When building a component instance, a common error is missing dependencies. Ensure that you create all the proper dependencies, and then check for any syntax errors.

Lesson 10: Aggregating in a State Diagram

State diagrams are hierarchical which allows modeling of complex state machines by abstracting detailed behavior into multiple levels. A state that does not contain a substate is a *simple state*. A state that has substates is a *composite state*. States may be nested to any level.

You can create capsule State Diagrams by defining simple behavior that is expanded a little at a time. If you want to hide some of the details of the behavior, Rose RealTime allows you to aggregate elements from the state diagram into composite states.

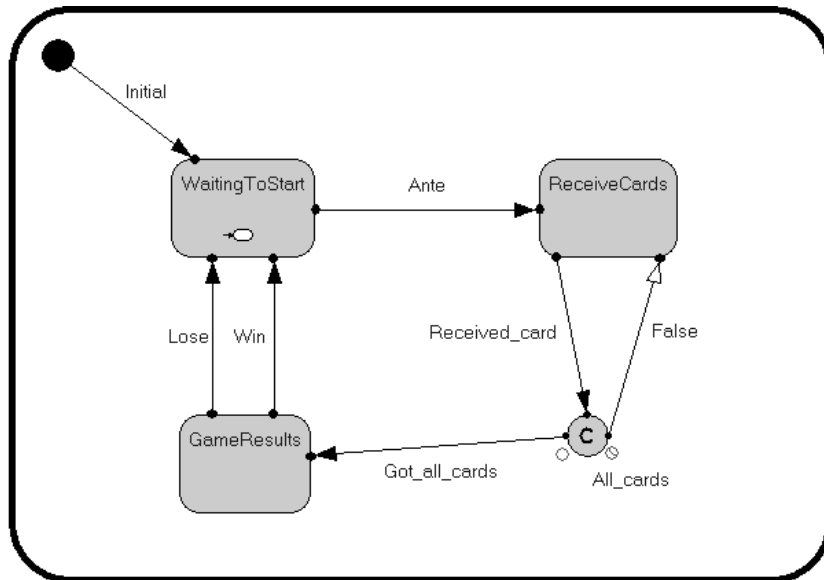
In this lesson, you will simplify the Player's state diagram by hiding the receiving cards logic into a composite state. This will make the top level state diagram easier to understand by hiding details about the behavior of actually receiving cards. Both state diagrams will remain identical in function. If a state diagram shows too many details (states and choice points) at the same level, it can become difficult to understand.

Aggregating the Receiving Behavior

Your goal is to abstract the choice point into one top level state named **ReceivingCards**. The Player's top level behavior is composed of only three states.

To create the composite state

- 1 Open the **State Diagram** for the **Player** capsule.

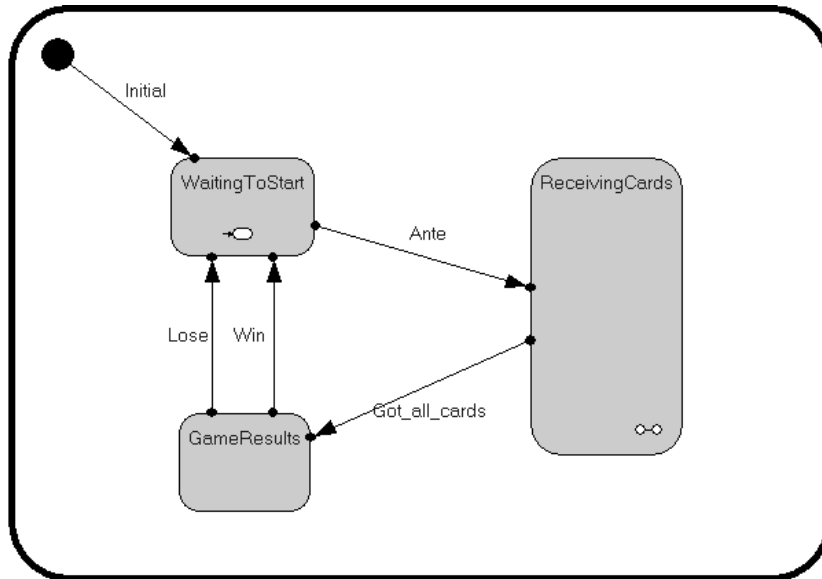


2 Multi-select the elements to aggregate by pressing the CTRL key and, at the same time, select the **ReceiveCards** state and the **All_cards** choice point.

3 On the **Parts** menu, click **Aggregate**.

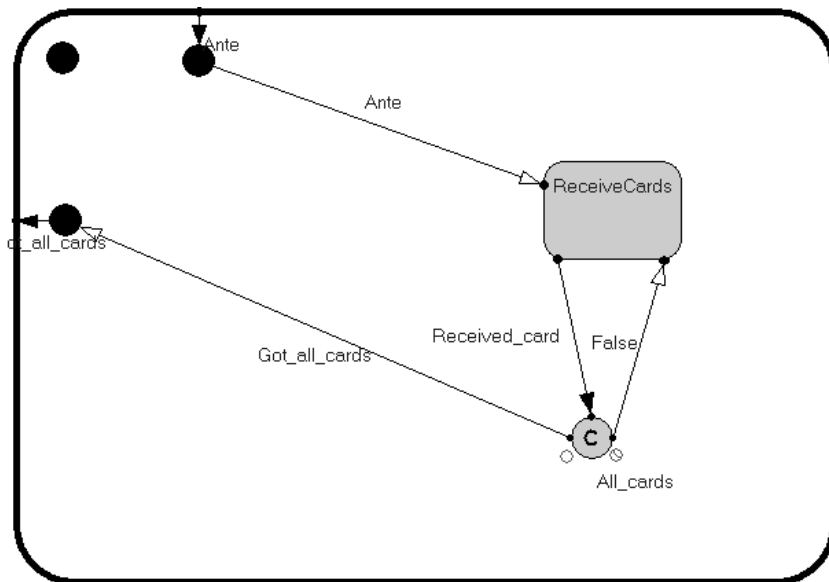
A composite state appears (the selected elements are removed from this state diagram) called **S1**.

4 Rename the state to **ReceivingCards**.



Note: A composite state icon appears in the bottom right-hand corner of the state indicating that the state contains sub-states.

- 5 Double-click on the new **ReceivingCards** state to view the state diagram.



Note: The elements that you chose to aggregate are part of the **ReceivingCards** state. Both state diagrams remain functionally equivalent.

Tutorial Summary

In this tutorial, you learned how to use the main features of Rational Rose RealTime to build a model.

You may want to explore Rational Rose RealTime further by:

- Expanding the card game model, and adding more players so that you can play multiple games at the same time. You may want to add dynamic structure and replication. The possibilities are endless.
- Customizing Rational Rose RealTime. For details, see the tutorials for the Rational Rose RealTime Extensibility Interface in RRTEI Tutorial Overview.
- Looking at the Examples. They show common design patterns used in beginner and intermediate level models.
- Looking at the model file that contains all of the tutorials in `<ROSERT_HOME>/Help/Tutorials/cardgame/cardgame_step5.rtmidl`.

Rational Rose RealTime Extensibility Interface Tutorials

4

Contents

This chapter is organized as follows:

- *RRTEI Tutorial Overview* on page 223
- *Creating a Summit Basic Script* on page 225
- *Creating a Visual Basic Add-in* on page 229
- *Creating an Add-in Which Extends the Context Menus* on page 236

RRTEI Tutorial Overview

This small set of tutorials describe the complete set of steps to follow to use the Rational Rose RealTime Extensibility Interface (RRTEI) to extend and customize the capabilities of the toolset. There are two ways of extending Rational Rose RealTime:

- writing Basic Scripts
- using the Automation object to access the RRTEI from within another application.

Both methods are explained in the following tutorials:

- Scripting language: *Creating a Summit Basic Script* on page 225.
- Automation: *Creating a Visual Basic Add-in* on page 229.
- Automation: *Creating an Add-in Which Extends the Context Menus* on page 236.

Note: The automation interface provided with Rational Rose RealTime is implemented using Microsoft Windows technology and can only be used on Windows platforms. The scripting language can be used on both UNIX and Windows.

These tutorials show Rational Rose RealTime users how to extend the toolset for their individual needs.

The following tutorials assume that you have some knowledge of the RRTEI. The RRTEI is a public interface to the Rational Rose RealTime meta model, or internal representation of a model, which is from where you can extract and manipulate information in a model.

Basic Scripts

The Rational Rose RealTime scripting language is an extension of the Summit BasicScript. The extensions allow you to automate specific functions of Rational Rose RealTime using the RRTEI interface. The script editor runs within the Rational Rose RealTime user interface.

Automation

Rational Rose RealTime automation works in two ways:

- Using Rational Rose RealTime as an automation controller that allows you to call an OLE automation object from within a Basic Script. For example, to execute functions in an application such as Microsoft Word or Microsoft Excel.
- Using Rational Rose RealTime as an automation server that allows other applications to call functions in the RRTEI to control Rational Rose RealTime. This can be done from any OLE-compliant development tool, such as, Visual Basic, Visual C++, and so on.

Note: The tutorials use Rational Rose RealTime as an automation server.

Previewing the Tutorials

The directory \$ROSE_HOME/Tutorials/rrtei contains one subdirectory for each of the RRTEI tutorials. In each folder, you will find examples of what will be developed in the tutorials.

To preview each add-in, do the following:

- 1 In each subdirectory there is a file of type .reg. Double-click on this file to update the registry with the settings for the add-in.

Note: The .reg files included with the tutorials assume that Rational Rose RealTime is installed in the default location, C:\Program Files\Rational\Rose RealTime\. If you installed the tool in another location, open the .reg file with a text editor and replace all path occurrences with the correct installation directory.

- 2 For the Automation examples to work, open and compile the Visual Basic project (.vbp). This will create and register a DLL file. The automation tutorials are located in the **vbaddin** and **contextmenu** directories.

For the BasicScript example, in the **summit** directory, you must first compile the script to .ebx format. See the tutorial for instructions on how to do this.

- 3 Re-start Rational Rose RealTime.
- 4 Read each tutorial to find out what each add-in does.

Creating a Summit Basic Script

The script you will create automates several steps required to create a capsule in Rational Rose RealTime. The script allows users to create capsules which already contain one state and an initial transition.

To create a Summit Basic script:

- 1 Use the Basic Script editor to write the script (you can also optionally use the dialog editor to create dialogs).
- 2 Run and test the script from within the Basic Script environment.
- 3 Compile the script to an .ebx file.
- 4 Create a menu file for the script.
- 5 Add entries to the registry to inform Rational Rose RealTime to add the menu items to its main menu.

Writing a Script

The script you will write will prompt the user for a capsule name and then use the RRTEI to create a new capsule and add an initial transition and a state to the state diagram of this newly created capsule.

- 1 Start Rational Rose RealTime.
- 2 Click **Tools > New Script**.
- 3 Type in the following lines of Basic code, or cut and paste from this document:

```
' This script creates one or more capsules within the  
' root Logical View package. It will only create a capsule  
' if there is no existing capsule with that name. To use  
' this script, run it, and repeatedly enter capsule  
' names into the pop up dialog. When you are finished press  
' the Cancel button. Each capsule created will have one  
' state called "Ready", with an Initial transition.
```

```

' Create capsule details
Sub CreateCapsuleDetails(theCapsule As RoseRT.Capsule)
    Dim theFsm As RoseRT.StateMachine
    Dim theTopState As RoseRT.CompositeState
    Dim theState As RoseRT.CompositeState
    Dim theTransition As RoseRT.Transition
    Set theFsm = theCapsule.StateMachine
    Set theTopState = theFsm.Top
    Set theState = theTopState.AddState(rsNormalState)
    theState.Name = "Ready"
    Set theTransition = theTopState.AddTransition("Initial",
rsTrueSourceRegion, "Ready")
End Sub

' Create a capsule within a given package
Sub CreateACapsule (thePackage As RoseRT.LogicalPackage, capsuleName
As String)
    If thePackage.Capsules.FindFirst(capsuleName) = 0 Then
        Dim theCapsule As RoseRT.Capsule
        Set theCapsule = thePackage.AddCapsule(capsuleName)
        CreateCapsuleDetails theCapsule
    End If
End Sub

' Create one or more capsules within the root
' LogicalView package

Sub CreateCapsules (theModel As RoseRT.Model)
    Dim thePackage As RoseRT.LogicalPackage
    Set thePackage = theModel.RootLogicalPackage
    capsuleName$ = askBox$( "Capsule" )
    While capsuleName$ <> ""
        CreateACapsule thePackage, capsuleName$
        capsuleName$ = askBox$( "Capsule" )
    Wend
End Sub

```

```

' Main
Sub Main
    CreateCapsules RoseRTApp.CurrentModel
End Sub

```

- 4 Select **File > Save Script As** to save the new script. Name it `CreateCapsules.ebs`.

Running and Testing a Script

- 1 Click the **Start** button at the top of the Basic Script edit window.
- 2 Enter one or more capsule names.
- 3 Click **Cancel**.

Verify that each capsule state diagram contains one state with an initial transition.

Compiling a Script

Select **File > Edit Path Map**. Note which directory is mapped to the `$SCRIPT_PATH` symbol, and which is mapped to `$ROSERT_HOME`.

With the script edit window selected, select **Debugger > Compile** to compile the script file. Name the compiled script `CreateCapsules.ebx`, and save it to the Rational Rose RealTime `$SCRIPT_PATH` directory.

Creating a Menu File

You should add a menu item to the Rational Rose RealTime main menu that will invoke your compiled script. To create a menu entry first you have to create a menu file. This is a simple text file containing the following:

```

Menu Tools
{
    Separator
    option "Create C&apsules"
    {
        RoseScript $SCRIPT_PATH/CreateCapsules.ebx
    }
}

```

The **RoseScript** keyword tells Rational Rose RealTime that this item represents a compiled Basic Script. You can add the menu item to any of the Rational Rose RealTime main menus. The **Menu <name>** tag in the menu file indicates under which main menu this new menu item should be added.

Save this file with the name `CreateCapsules` with the extension `.mnu`.

Adding Entries to the Registry

Next you have to update the Rational Rose RealTime registry to inform the toolset about the script. You can use **regedit** to directly edit the registry or create a .reg file, and double-click it to add the entries to the registry. Below is a sample .reg file that you can use as a reference:

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\Rational Software\Rose
RealTime\6.4\AddIns\CreateCapsules]

"Active"="Yes"

"HelpFileName"="CreateCapsules.htm"

"RoseRTAddIn"="Yes"

"Version"="1.0"

"InstallDir"="C:/Program Files/Rational/Rose
RealTime/6.4/Tutorials/rrtei/summit"

"MenuFile"="CreateCapsules.mnu"
```

Key	Description
Active	A Yes indicates that this add-in should be enabled in the Tool menu. (It also indicates that the add-in should receive special event notifications when Rational Rose RealTime starts and stops).
HelpFileName	The name of the HTML help file that corresponds to a specific help topic, if necessary.
RoseRTAddIn	A Yes indicates that add-in was created using the RRTEI. Omitting this entry or an entry with a No value indicates that this is a Rose REI add-in. To run Rose add-ins, Rational Rose RealTime has to be started in emulation mode.
Version	The version to display in the Add-Ins manager dialog.
InstallDir	The directory where Rational Rose RealTime can find the menu file.
MenuFile	The name of the file with the menu commands. This file specifies the menu entry name for the Tools menu, and the method to invoke in the automation server.

Note: When specifying a path in a registry entry, use the UNIX path separator "/", or use double DOS path separator "\\ ". Not doing so will prevent the registry entry from being created.

Running and Testing the Script From the Menu

Update the registry then restart Rational Rose RealTime. Ensure that the menu item was correctly added to the main menu.

Ensure that your script is run when the menu item is selected.

Troubleshooting

If Rational Rose RealTime has problems loading the add-in it will display any errors in the log. The most common errors are typos in the registry entries and menu file. This usually causes the .ebx and .mnu files not to be found. Remember that you have to restart Rational Rose RealTime every time you change the registry settings.

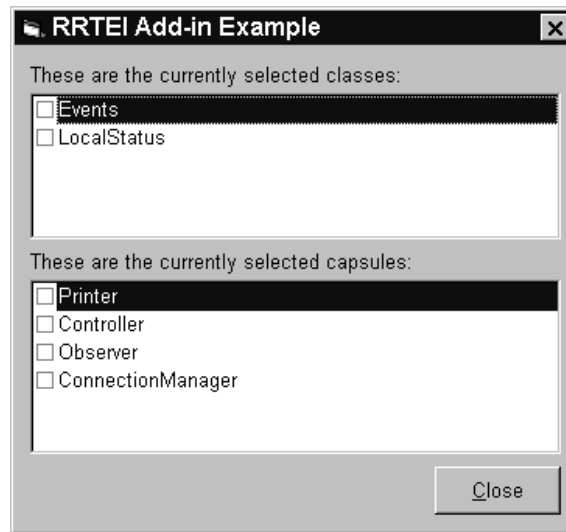
Creating a Visual Basic Add-in

By following a few well defined steps you can easily create a complete Visual Basic add-in. These add-ins are compiled as Active X DLLs and when registered with Rational Rose RealTime are loaded and invoked by the toolset.

To create a Rational Rose RealTime add-in with Visual Basic:

- Create an ActiveX DLL with a public interface and sub that accepts a **RoseRTApplication** object reference as a parameter.
- Add a few entries in the Registry to let Rational Rose RealTime know about the new add-in.
- Create a menu file that calls the add-in from a new menu item you can add to the main menu.

The add-in you create in this tutorial will query the model and find which classes and capsules are selected in a diagram. The add-in will then add the names of the selected items to list boxes that will be displayed in a dialog box.



Note: This is not a tutorial on using Microsoft Visual Basic. It is assumed that you have some basic knowledge of the environment and language.

Creating the ActiveX DLL

Load Microsoft Visual Basic (version 5 or 6) and from the **New Project** dialog chose to create an **Active X DLL** project. This will create a new project named **Project1** with a class module called **Class1**.

- 1 Rename the **Project1** to **rrtei_intro_tutorial**.
- 2 Rename the **Class1** to **clsMain**.

These names are important because they will determine the DLL name and the class name that you will register with the add-in manager.

- 3 Make sure that the **Instancing** property on the **clsMain** class module is set to **MultiUse**.

This value specifies whether you can create instances of a public class outside a project, and if so, how it will behave.

- 4 Select **Project > References**.

A dialog box appears listing all the references (type libraries) that are registered on your machine.

5 Select **RoseRT**.

This will allow you to get drop-down hints when editing code and also to easily browse the RoseRT type library. The type library contains all the classes, properties, and methods that you can access from the RRTEI. To browse the contents of the RoseRT type library (this is very useful) press **F2** or chose **View > Object Browser** then select the RoseRT library from the drop-down list in the top left corner of the dialog box.

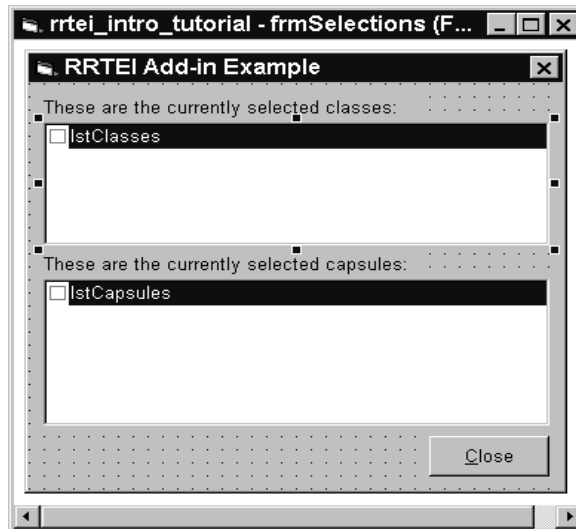
Note: Rational Rose RealTime must be installed on your machine to access the type library.

6 Create a public sub called **RunRRTEITutorial** on the **clsMain** class.

```
Public Sub RunRRTEITutorial(theRTApp As RoseRT.Application)
...
End Sub
```

This **Sub** takes one parameter of type **RoseRT.Application**. This is the operation that will be invoked when your add-in is called from Rational Rose RealTime.

7 Then create a new form called **frmSelections** that has two list boxes named **lstClasses** and **lstCapsules**. You can create something similar to:



- 8 Add code to unload the form when the Close button is pressed:

```
Private Sub CloseCommand2_Click()  
    Unload Me  
End Sub
```

- 9 Add the code to the **RunRRTEITutorial** sub that will populate the list boxes in the **frmSelections** form.

```
Public Sub RunRRTEITutorial(theRTApp As RoseRT.Application)  
    ' create a form object  
    Dim mainForm As New frmSelections  
    ' initialize local vars  
    Dim theModel As Model  
    Dim SelectedClasses As ClassCollection  
    Dim SelectedCapsules As CapsuleCollection  
    Dim aClass As Class  
    Dim aCapsule As Capsule  
  
    ' get the list of selected capsules and classes  
    Set theModel = theRTApp.CurrentModel  
    Set SelectedClasses = theModel.GetSelectedClasses  
    Set SelectedCapsules = theModel.GetSelectedCapsules  
  
    ' populate the lists on the frmSelections form  
    mainForm.lstClasses.Clear  
    For i = 1 To SelectedClasses.Count  
        Set aClass = SelectedClasses.GetAt(i)  
        mainForm.lstClasses.AddItem aClass.Name  
    Next i  
    mainForm.lstCapsules.Clear
```



```

For i = 1 To SelectedCapsules.Count
    Set aCapsule = SelectedCapsules.GetAt(i)
    MainForm.lstCapsules.AddItem aCapsule.Name
Next i

' display the form
MainForm.Show vbModal

End Sub

```

You create an instance of the **frmSelections** form. Then you get a reference to the model instance from the Application object passed to the sub by Rational Rose RealTime when the add-in was called. By calling the **GetSelectedClasses** and **GetSelectedCapsules** operations on the model reference, you can get the collection of classes and capsules which are selected. Then the list boxes are populated and the form is displayed.

10 Build the add-in DLL file by selecting **File > Make rrtei_intro_tutorial.dll**.

This will compile the DLL and register it in the Windows registry.

Creating the Add-in Menu File

You now want to create the menu file. You can extend the menu system with custom items that invoke compiled Basic Scripts or add-ins. To create a menu entry first, you have to create a menu file. This is a simple text file containing the following:

```

Menu Tools
{
    Separator
    option "Run RRTEI &Tutorial Add-in"
    {
        InterfaceEvent rrtei_intro_tutorial RunRRTEITutorial
    }
}

```

The **InterfaceEvent** keyword tells Rational Rose RealTime that this item represents an add-in which is specified in the registry under the **AddIns** key. The second parameter indicates the name of the subkey under **AddIns** where the name of the automation server (OLEServer) for the add-in can be found. The third parameter is the name of the operation to call on the OLEServer described in the registry. You can therefore have several entry points implemented in the same add-in DLL.

The menu file should be placed in the directory specified by the **InstallDIR** registry entry, and must have the name specified by the **MenuFile** entry.

Adding Entries to the Registry

Next you have to update the Rational Rose RealTime registry to let the toolset know about the add-in. You can use **regedit** to directly edit the registry or create a .reg file, and double-click it to add the entries to the registry. Below is a sample .reg file that you can use as a reference:

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\Rational Software\Rose
RealTime\6.4\AddIns\rrtei_intro_tutorial]

"Active"="Yes"

"HelpFileName"="tutorial.htm"

"RoseRTAddIn"="Yes"

"Version"="1.0"

"InstallDir"="C:/Program Files/Rational/Rose
RealTime/6.4/Tutorials/rrtei/vbaddin"

"MenuFile"="rrtei_intro_tutorial.mnu"

"OLEServer"="rrtei_intro_tutorial.clsMain"
```

Key	Description
Active	A Yes indicates that this add-in should be enabled in the Tool menu. (It also indicates that the add-in should receive special event notifications when Rational Rose RealTime starts and stops).
HelpFileName	The name of the HTML help file that corresponds to a specific help topic, if necessary.
RoseRTAddIn	A Yes indicates that add-in was created using the RRTEI. Omitting this entry or an entry with a No value indicates that this is a Rose REI add-in. To run Rose add-ins, Rational Rose RealTime has to be started in emulation mode.
Version	The version to display in the Add-Ins manager dialog.
InstallDir	The directory where Rational Rose RealTime can find the menu file.
MenuFile	The name of the file with the menu commands. This file specifies the menu entry name for the Tools menu, and the method to invoke in the automation server.
OLEServer	The name of the registered automation server (ActiveX DLL) and the name of the interface to associate with this add-in. This is also called the ProgID .

Testing the New Add-in

After you have completed all the previous steps, run Rational Rose RealTime and then select **Add-Ins > Add-In Manager...** You should see your add-in listed with the version number you entered into the registry.

Create a couple of classes and capsules and place them on a class diagram. Then multi-select the elements (using the CTRL key while selecting the elements), and select **Tools > Run RRTEI Tutorial** (or whatever name you gave the menu entry). A dialog box appears with the classes and capsules list updated with the names of the selected elements in the current diagram.

Note: You can test the add-in from within Visual Basic by setting "Start Program" VB project's property on the debugging tab to "RoseRT". Then press the start button in Visual Basic and Rational Rose RealTime will be started automatically.

Common Problems

The most common problem is that your registry entries (**InstallDir**, **OLEServer**) are incorrect. If any of these entries are wrong either the Add-In won't register when Rational Rose RealTime loads or the Add-In will not run when you select the menu item. Ensure that you have the following names correct:

- **Registry key name entry under the Addin subfolder:** used in the .mnu file.
- **DLL name (usually the same as your VB project name):** used as the first parameter in the **OLEServer** registry key.
- **Class name in which your add-in entry sub is declared:** used as the second parameter in the **OLEServer** registry key.
- **Procedure name which should be invoked by Rational Rose RealTime to run the add-in:** used in the .mnu file.

The next common problem is that run-time errors occur while your add-in is executing. Rational Rose RealTime will report the error. Try and trap and report errors from within your add-in.

Creating an Add-in Which Extends the Context Menus

When you right-click in Rational Rose RealTime, the system displays a shortcut menu. The commands displayed on the shortcut menu are determined by where you click the mouse and what items are selected in the diagram or browser. You can take advantage of this feature so that your add-in user sees your shortcut menu items when they right-click.

Benefits

- Provides shortcut access to an add-in.
- Ability to create one shortcut menu item that works for items selected in the browser as well as in a diagram (you do not have to create one menu item for items selected in the browser and another menu item for items selected in the diagram).
- Can add submenus.
- Can control the state in which the context menu appears (disabled, checked ...).

Limitations

The position on the shortcut menu where your menu item displays is controlled by Rational Rose RealTime. If you have more than one item on the shortcut menu, however, you can control the order in which those items display by adding the items (using the **AddContextMenuItem** method) in the order in which you want the menu items displayed.

How Context Menus Work

Using context menus is fairly simple once you understand the events generated by Rational Rose RealTime and the objects and operations available in the RRTEI that support adding and removing context menus.

The main flow of events that occurs when an add-in uses context menus in Rational Rose RealTime are:

- 1 Add-in designer first decides to which model elements shortcuts should be added. Then, it decides on unique names called the **internal name** for each of the menu items. The internal name is what Rational Rose RealTime will pass to your add-in when a menu item is selected.
- 2 When Rational Rose RealTime is started, it issues the **OnActivate** event to your add-in. Your add-in would create all the context menus it requires using the **Addin::AddContextMenuItemForClass** operation at this time.
- 3 When the user right-clicks on an element for which your add-in has added a context menu, Rational Rose RealTime issues the **OnEnableContextMenuItemsForClass** before the context menu is shown. In this operation, your add-in has the option of changing the state of any of the menu items before they are shown.
- 4 When the user selects an add-in menu item from the context menu, Rational Rose RealTime issues the **OnSelectedContextMenuItem** event to your add-in. The **internal name** is passed with the **OnSelectedContextMenuItem** event as is used to determine which menu item has been selected.

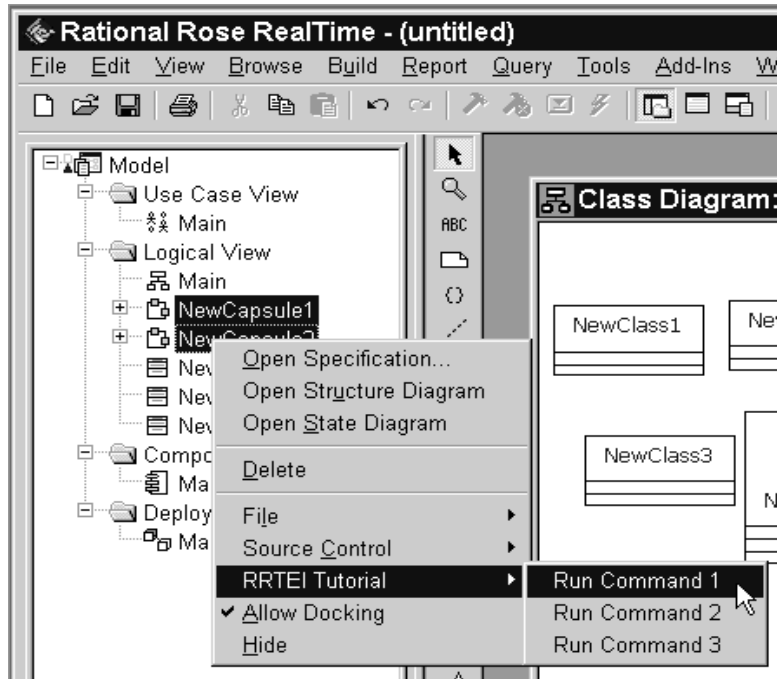
Menus Associated with Default or Specific Elements

The default context menu is the shortcut menu that is displayed when there are multiple selections of different types of items. You either create a context menu for a specific type of element or for the default.

When creating a context menu, you assign it to a model element by referring to the element name. For example, to create a context menu for a capsule, you would pass the "Capsule" string to the **AddContextMenuForClass** function.

Creating the ActiveX DLL

In this tutorial, you will create a simple add-in which adds a submenu called **RRTEI Tutorial** and three menu items to the Capsule context menu. The first menu item will only be enabled if more than one capsule is selected in the model when the context menu appears.



Note: The steps for setting-up and configuring a Visual Basic project for an add-in were covered in the previous tutorial. For information, see *Creating a Visual Basic Add-in*.

Creating a New Visual Basic Project

Open Visual Basic and create a new **ActiveX DLL** project. Then rename the project to **rrtei_contextmenu_tutorial** and the class module to **clsMain**.

Add a reference to the **RoseRT** type library.

Adding Module Variables

Open the **clsMain** module and add the following variables and constants to the module file:

```
' global vars
Dim MyAddin_ As RoseRT.AddIn
Dim MyMenuItems_(3) As RoseRT.ContextMenuItem

' context menu internal names used to identify individual menu items
Const CMID_COMMAND1 As String = "command1"
Const CMID_COMMAND2 As String = "command2"
Const CMID_COMMAND3 As String = "command3"
```

The first variable **MyAddin_** is used to keep a reference to this add-in for quick access in other functions. The **MyMenuItems_(3)** array holds references to all the new menu items that will be added. This again allows quick access to the menu items.

The constants represent the declaration of the **internal names** that will be used for the menu items. These internal names will be used when a menu item is created and when an item is selected. The internal name will be passed to the **OnSelectedContextMenuItem** function to identify which menu item has been pressed.

Adding the OnActivate Function

This function is called when the add-in is initialized. This is the time to create the new context menu items. Add the **OnActivate** function to the **clsMain** module:

```
Sub OnActivate(pRoseApp As RoseRT.Application)
    Dim aContextMenu As RoseRT.ContextMenuItem
    Dim AddIns As RoseRT.AddInCollection
    Set AddIns = pRoseApp.AddInManager.AddIns

    ' find the add-in
    Set MyAddin_ = AddIns.GetFirst("rrtei_contextmenu_tutorial")

    If MyAddin_ Is Nothing Then
        MsgBox ("Error cannot add context menus")
        Exit Sub
    End If
```

```

' Build the context menu

Set aContextMenu = MyAddin_.AddContextMenuForClass("Capsule",
"Submenu RRTEI Tutorial", "")

Set MyMenuItems_(1) =
MyAddin_.AddContextMenuForClass("Capsule", "Run Command 1",
CMID_COMMAND1)

Set MyMenuItems_(2) =
MyAddin_.AddContextMenuForClass("Capsule", "Run Command 2",
CMID_COMMAND2)

Set MyMenuItems_(3) =
MyAddin_.AddContextMenuForClass("Capsule", "Run Command 3",
CMID_COMMAND3)

Set aContextMenu = MyAddin_.AddContextMenuForClass("Capsule",
"EndSubmenu", "")

End Sub

```

The first part of the function simply tries to obtain a reference to the add-in object that represents this add-in within the Rational Rose RealTime application object. The add-in is searched by name.

Next the add-in reference is used to call the **AddContextMenuForClass** operation to add the menu items. The menu item references are saved on the **MyMenuItems_(3)** array for use later on. See the RRTEI reference for a description of the parameters that are passed to the **AddContextMenuForClass** function.

Adding the OnEnableContextMenuItemsForClass Function

This function will be called prior to the context menu for the class is displayed. This is the time to change the status of any of the menu items that have been added. For example, depending on the elements that have been selected at the time the right-click occurs. For this tutorial you will only enable the first menu item if more than one element was selected at the time the context menu is selected.

Add the following code:

```
Function OnEnableContextMenuItemsForClass(pRoseApp As  
RoseRT.Application, items As RoseRT.ControllableElementCollection) As  
Boolean  
  
    Dim cmItem As RoseRT.ContextMenuItem  
    Dim cmCollection As RoseRT.ContextMenuItemCollection  
  
    ' activate this menu item only if two or more model elements are  
    selected  
    If items.Count > 1 Then  
        MyMenuItems_(1).MenuState = RoseRT.rsEnabled  
        OnEnableContextMenuItemsForObjects = True  
    Else  
        MyMenuItems_(1).MenuState = RoseRT.rsDisabled  
        OnEnableContextMenuItemsForObjects = False  
    End If  
End Function
```

The **OnEnableContextMenuItemsForClass** function's second parameter is a collection of elements that are selected when the user requested the context menu. This collection determines if the first menu item is enabled or disabled. Since we had already saved references to the added menu items, we can change the **MenuState** property of the menu item object.

Adding the OnSelectedContextMenuItem Function

This function will be called when one of the add-in menu items is selected. The function is passed an internal name as a parameter which is used to determine which menu item has been selected. In this tutorial you will simply display a message box with the internal name followed by the names of the model elements that are selected when the menu item is selected.

```

Function OnSelectedItemForObjects(pRoseApp As
RoseRT.Application, InternalName As String, items As
RoseRT.ControllableElementCollection) As Boolean

    Dim msg As String
    msg = InternalName + ": "

    Dim i As Integer
    Dim myElement As RoseRT.ControllableElement

    For i = 1 To items.Count Step 1
        Set myElement = items.GetAt(i)
        msg = msg + myElement.Name + " "
    Next
    MsgBox (msg)
    OnSelectedItemForObjects = True
End Function

```

The list of selected elements is passed to the function as a **ControllableElementCollection**. You would typically use this and the **internal name** to decide what action to take.

Building the Add-in

The last step is to build the add-in DLL file by selecting **File > Make rrtei_contextmenu_tutorial.dll**. This will compile the DLL and register it in the Windows registry.

Adding Entries to the Registry

Next you have to update the Rational Rose RealTime registry to let the toolset know about the add-in. You can use **regedit** to directly edit the registry or create a .reg file, and double-click to add the entries to the registry. Below is a sample .reg file that you can use as a reference:

```

REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\Rational Software\Rose
RealTime\6.4\AddIns\rrtei_contextmenu_tutorial]
"Active"="Yes"
"Version"="1.0"
"LanguageAddIn"="No"
"OLEServer"="rrtei_contextmenu_tutorial.clsMain"
"RoseRTAddin"="Yes"

```

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Rational Software\Rose  
RealTime\6.4\Addins\rrtei_contextmenu_tutorial\Events]  
"OnActivate"="Interface"
```

The events subkey is used to tell Rational Rose RealTime that we want to receive the **OnActivate** event when the add-in is registered in Rational Rose RealTime.

Testing the New Add-in

You don't have to create a menu file, but you could, since the menu items for this tutorial are all dynamically created. You could optionally add the same commands to the main menu to allow users several ways of invoking your add-in.

Once you have completed all the previous steps, run Rational Rose RealTime and then select **Add-Ins > Add-In Manager...** You should see your add-in listed with the version number you entered into the registry.

Next, create a couple of capsules. Multi-select the elements (using the CTRL key while selecting the elements) in the model browser or in the diagram and right-click. You should see the **RRTEI Tutorial** submenu on the context menu. Select one of the commands that you have added. Also, test that when the context menu is selected and only one capsule is selected, the first command is disabled. And that it is enabled when more than one is selected.

Contents

This chapter is organized as follows:

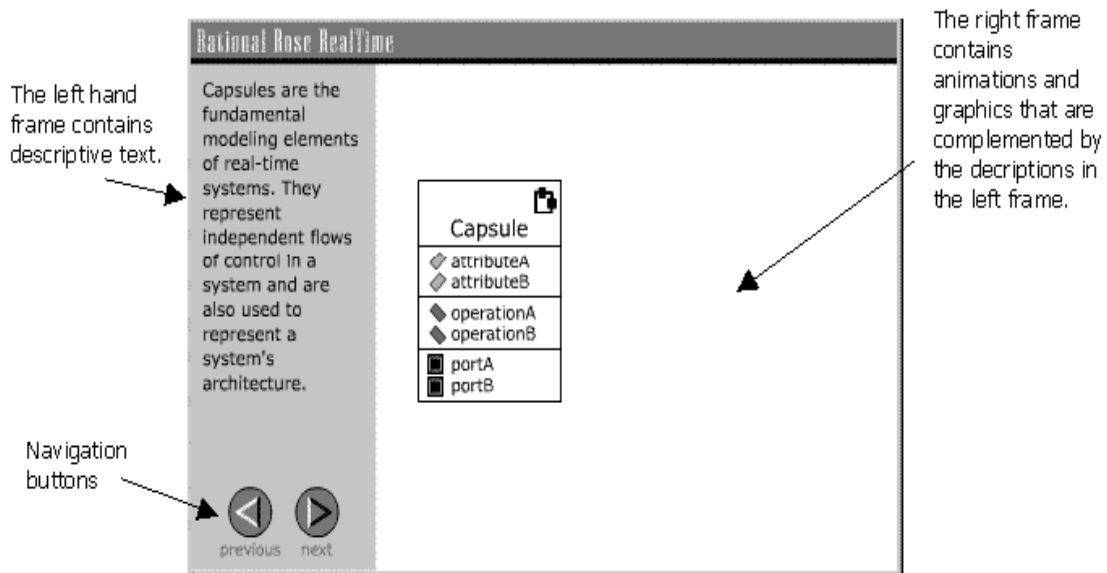
- *Overview* on page 245
- *Messages and Capsule State Machines* on page 246
- *Capsule Hierarchical State Machines* on page 247
- *Capsules and Capsule Roles* on page 247
- *Ports, Protocols, and Protocol Roles* on page 247

Overview

The *Concept* tutorials provide an introduction to the important Rational Rose RealTime concepts. They expand and summarize the explanations and examples given in the *Rational Rose RealTime Modeling Language Guide*.

Note: The concept tutorials are presented as Shockwave animations that you activate from the online help. To view the tutorials, you must install the Shockwave plug-in. The tutorials are displayed in the Help browser with text shown on the left side of the window, with graphics and animations on the right. You control the tutorial by using the navigation buttons at the bottom of the left-hand frame.

Figure 2 Tutorial Window



Note: These tutorials cannot be viewed with the UNIX Help viewer. You can install the UNIX Netscape plug-in from Shockwave, and load the tutorials using a Netscape 4.Xbrowser. For more information, see [\\$ROSSERT_HOME/Tutorials/unix/index.htm](#).

Messages and Capsule State Machines

Note: This tutorial introduces you to the basics of message passing between capsules. If you do not see the tutorial window, install the Shockwave plug-in. These tutorials cannot be viewed with the Unix help viewer. You can download the Unix Netscape plug-in from Shockwave and load the tutorials using a Netscape 4.X browser. See [\\$ROSSERT_HOME/Tutorials/unix/index.htm](#) for more information.

Capsule Hierarchical State Machines

This tutorial reviews the basic elements of state machines and explains some of the complexities involved with hierarchical capsule state machines.

Note: If you do not see the tutorial window, install the Shockwave plug-in. These tutorials cannot be viewed with the Unix help viewer. You can download the Unix Netscape plug-in from Shockwave and load the tutorials using a Netscape 4.X browser. See `$ROBERT_HOME/Tutorials/unix/index.htm` for more information.

Capsules and Capsule Roles

This tutorial is intended for those users who are familiar with class modeling and who want to understand the additional concepts involved when modeling with capsules. The tutorial outlines the differences and similarities between class and structure diagrams.

Note: If you do not see the tutorial window, install the Shockwave plug-in. These tutorials cannot be viewed with the Unix help viewer. You can download the Unix Netscape plug-in from Shockwave and load the tutorials using a Netscape 4.X browser. See `$ROBERT_HOME/Tutorials/unix/index.htm` for more information.

Ports, Protocols, and Protocol Roles

Note: This tutorial is intended for those users who want an introduction to protocols and protocol roles in a Rational Rose RealTime model. If you do not see the tutorial window, install the Shockwave plug-in. These tutorials cannot be viewed with the Unix help viewer. You can download the Unix Netscape plug-in from Shockwave and load the tutorials using a Netscape 4.X browser. See `$ROBERT_HOME/Tutorials/unix/index.htm` for more information.

Index

A

- actions 130
 - creating
 - creating
 - actions 142
- Active X DLL 230, 238
- AddContextMenuForClass 240
- adding
 - attributes 140
 - capsule behavior 122
 - code to state machine 34
 - destructor 203
 - detail code to operations 54
 - entries to the registry 234, 242
 - inclusions 213
 - port to a capsule 30
 - registry entries 228
 - state to a capsule 26
 - use case 60
- add-ins
 - building 242
 - creating menu file 233
 - creating Visual Basic add-in 229
 - extending the context menus 236
 - testing 235, 243
- aggregating state diagrams 219
- aggregations 205
- association ends 186
- Attributes 205
- attributes 140
- automation
 - RRTEI tutorials 224

B

- build
 - errors 44, 117, 186, 197
 - log 44, 117

- results 44, 117
- starting 115
- build information 54
- building
 - component 43
 - model 107
 - specifying platform requirements 42, 114

C

- capsule 68
 - adding a port 30
 - adding a state 26
 - adding behavior 122
 - creating 25, 69
 - structure 75
 - top-level 36, 108
- capsule instance trace 152
- capsules
 - initial 59
- card game
 - simulation scenario 52
- card game requirements 52
- card game tutorial
 - tutorials
 - card game 52
- changing
 - element types 74
- choice point
 - creating 136
- class 68
 - behavior 68
 - creating 69
 - importing 166
- Class Diagrams
 - delete key 67
- class modeling 165
- classes
 - defining 67
- communication protocol 95

- compiling
 - Summit Basic script 227
- component 114
 - building 43
- component instance 45, 115
 - running 47
 - terminating 48
 - trace 216
- component view 22
 - creating 36, 108
- component wizard 36, 108
- concept tutorials
 - capsule hierarchical state machines 247
 - capsules and capsule roles 247
 - messages and capsule state machines 246
 - overview 245
 - ports, protocols, and protocol roles 247
- configuring
 - runtime windows 148
 - toolset 55
 - toolset options 56
- connector 100
- ConstructorInitializer 180, 198
- contacting Rational customer support xiii
- Context Menus
 - how they work 236
- context menus
 - overview 236
- ControllableElementCollection 242
- crating
 - destructor 182
- creating
 - a new model 24
 - actions 130
 - Active X DLL 230, 238
 - add-in menu file 233
 - capsules 25, 69
 - choice point 136
 - classes 69
 - component 36, 108
 - component instance 45, 115
 - component view 36, 108
 - components using Component wizard 36
 - connectors 100
 - deployment view 45, 114
 - initial capsules 59
 - logical view 25
 - menu file 227
 - model 55
 - new model 55
 - packages 167
 - ports 100
 - processor 115
 - protocol signals 96
 - protocols 94
 - relationships between classes 169
 - sequence diagram 81
 - sequence diagram interactions 83
 - signals for a protocol 96
 - components using Component wizard 108
 - state diagram 134
 - states 124
 - summit Basic script 225
 - timing port 133
 - transitions 126
 - triggers 128
 - use case
 - use case
 - creating 63
 - use cases 59
 - Visual Basic Add-in 229

D

- debugging
 - inject messages 157
 - injecting 157
 - trace 157
 - using traces 147
 - using watches 147
- decoding
 - Services Library 186
- defining
 - classes 67
- deleting
 - elements from diagram only 67
 - elements from model 67
- dependency
 - properties 213

deployment view 22
 creating 45, 114
destructor 182

E

element types 74
encoding
 Services Library 186
end ports 33
errors
 troubleshooting Summit Basic scripts 229
extending the context menus using add-ins 236

F

filter relationships 79
FOC 87
focus of control 87

G

generated code 49
getting started
 QuickStart tutorial 19

H

HeaderPreface 214

I

importing
 classes 166
Inclusions 213
initial capsules 59
initial point 29, 125
initial state 29, 125
initial transition 29, 125
inject
 messages 157
injecting 157

L

logical view 22
 creating 25

M

menus
 context 236
messages
 inject 157
 injecting 157
 tracing 157
models
 description of QuickStart 23
 sample 22
MyAddin_ 239
MyMenuItems_ 239

N

NumElementsFunctionBody 188
NumFunctionElementsBody 197

O

OnActivate 239
OnEnableContextMenuItemsForClass 240
online help 22
OnSelectedContextMenuItem 241
opening
 new model 55

P

package
 creating 167
 RTClasses 74
port 100
 adding to capsule 30
 creating 100
 timing 133

- ports
 - end port 33
 - protected 31
- protected port 31
- protocol
 - creating 94
 - signals 96
- prototyping 106

R

- Rational customer support
 - contacting xiii
- rebuilding a model 147
- referenced classes 44, 116
- registry
 - adding entries 228, 234, 242
- relationships 79
- roles
 - connecting 103
- RRTEI tutorials
 - Active X DLL 230, 238
 - add-in menu file, creating 233
 - adding entries to the registry 228, 234, 242
 - adding module variables 239
 - automation 224
 - building add-ins 242
 - common add-in problems 235
 - compiling a Summit Basic Script 227
 - context menu overview 236
 - creating a menu file 227
 - creating a Summit Basic script 225
 - extending the context menus using
 - add-ins 236
 - overview 223
 - previewing 224
 - running a script 227
 - testing a script 227
 - testing add-ins 243
 - testing new add-ins 235
 - Visual Basic add-in 229
 - writing a script 225
- RTClasses package 74
- rtdata 132

- running
 - component instance 47
 - Summit Basic script 227
- runtime windows 148

S

- sample model 22
- saving a model 34
- sequence diagram
 - creating 81
- sequence diagrams
 - interactions 83
- Sequence Event Message dialog 217
- state diagram
 - aggregating 219
 - creating 134
 - drawing an initial transition 29, 125
 - initial point 29, 125
 - initial state 29, 125
 - initial transition 29, 125
- state machine
 - adding detail code 34
- stdlib.h 213
- Summit Basic
 - compiling a script 227
 - creating a script 225
 - running a script 227
 - testing a script 227

T

- testing
 - Summit Basic script 227
- time.h 213
- timing port 133
- top-level capsule 36, 108
- trace
 - capsule instance 152
 - component instance 216
- traces 147
- tracing 157

- transitions 126
 - drawing 29, 125
 - initial 29, 125
- triggers 128
- troubleshooting
 - common new add-in problems 235
 - Summit Basic scripts 229
- tutorials
 - environment configuration 19, 53
 - navigating 17
 - printing 17
 - QuickStart 19

U

- use case
 - adding 60
 - documentation 65
 - flow of events 64
- use cases
 - creating 59
- use-case view 22
- user interface
 - main features 21
 - Rational Rose RealTime 20
 - views 21

V

- viewing
 - generated code 49
- views
 - component 22
 - deployment 22
 - logical 22
 - use-case 22

W

- watch window 150
- watches 147
- writing a Summit Basic script 225

