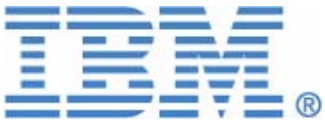
Rational. Rhapsody

IBM

C++ Tutorial

C++ Tutorial for Rational Rhapsody



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

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

© Copyright IBM Corporation 1997, 2009.

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

Contents

Getting Started 1

Audience for the C++ Tutorial 1

Before You Begin 1

C++ Tutorial Overview. 2

C++ Tutorial Objectives 3

Documentation Conventions 4

About the Rational Rhapsody Product 5

UML Diagrams 5
Starting the Rational Rhapsody Product 6
Closing the Rational Rhapsody Product 6

Setting Up the C++ Tutorial 7

Creating a Project 7
About a Rational Rhapsody Project 9
Saving a Project 12
Organizing the Model Using Packages 13

Opening the Handset Model 18

Using Naming Conventions 19

Prefixes 19
Model Element Names 19

Rational Rhapsody User Interface 20

Toolbars 21
Browser 22
Drawing Area 23
Output Window 23
Drawing Toolbars 23
Features Dialog Box 24

Summary 29

Lesson 1: Creating Use Case Diagrams 31

Goals of this Lesson 31

Exercise 1: Creating the Functional Overview UCD. 32
Task 1a: Creating the Functional Overview Use Case Diagram 33
Task 1b: Drawing the Boundary Box and Actors. 35
Task 1c: Drawing the Use Cases 37
Task 1d: Defining Use Case Features 39
Task 1e: Associating Actors with Use Cases 40
Task 1f: Drawing Generalizations 42
Task 1g: Adding Remarks to Model Elements and Diagrams. 43
Exercise 2: Creating the Place Call Overview UCD 45
Task 2a: Creating the Place Call Overview Use Case Diagram 46
Task 2b: Drawing the Use Cases 46
Task 2c: Defining Use Case Features. 48
Task 2d: Drawing Generalizations 49
Task 2e: Modeling Requirements in Rational Rhapsody 50
Exercise 3: Creating the Data Call Requirements UCD 58
Task 3a: Creating the Data Call Requirements Use Case Diagram 59
Task 3b: Adding Requirements. 59
Task 3c: Drawing and Defining the Dependencies 61
Summary 63

Lesson 2: Creating Structure Diagrams 65

Goals of this Lesson 65
Exercise 1: Creating the Handset System Structure Diagrams 65
Task 1a: Creating the Handset System Structure Diagram 67
Task 1b: Drawing Objects 69
Task 1c: Drawing More Objects 72
Task 1d: Drawing Ports. 75
Task 1e: Drawing Flows 77
Task 1f: Specifying the Port Contract 82
Task 1g: Allocating the Functions Among Subsystems 88
Exercise 2: Creating the Connection Management Structure Diagram 91
Task 2a: Creating the Connection Management Structure Diagram. 92
Task 2b: Drawing Objects 92
Task 2c: Drawing Ports 93
Task 2d: Drawing Links. 94
Exercise 3: Creating the Data Link Structure Diagram 96
Task 3a: Creating the Data Link Structure Diagram 97
Task 3b: Drawing Objects 97
Task 3c: Drawing Ports 98
Task 3d: Drawing Links. 98
Task 3e: Specifying the Port Contract and Attributes 99

Exercise 4: Creating the MM Architecture Structure Diagram 100
Task 4a: Creating the MM Architecture Diagram 101
Task 4b: Drawing Objects 101
Task 4c: Drawing Ports 102
Task 4d: Drawing Links 102
Task 4e: Specifying the Port Contract and Attributes 103
Summary 104

Lesson 3: Creating Object Model Diagrams 105

Goals for this Lesson 105
Exercise 1: Creating the Subsystem Architecture OMD 106
Task 1a: Creating the Subsystem Architecture Object Model Diagram 107
Task 1b: Drawing Objects 108
Task 1c: Drawing More Objects 109
Task 1d: Drawing Links 110
Summary 110

Lesson 4: Generating Code and Building Your Model 111

Goals for this Lesson 111
Exercise 1: Preparing for Generating Code 112
Task 1a: Creating a Component 112
Task 1b: Setting the Component Features 113
Task 1c: Creating a Configuration 114
Task 1d: Generating Code 115
Task 1e: Building the Model 117
Summary 118

Lesson 5: Creating Sequence Diagrams 119

Goals for this Lesson 119
Exercise 1: Creating the Place Call Request Successful SD 120
Task 1a: Creating the Place Call Request Sequence Diagram 122
Task 1b: Drawing Actor Lines 123
Task 1c: Drawing Classifier Roles 124
Task 1d: Drawing Messages 125
Task 1e: Drawing an Interaction Occurrence 128
Exercise 2: Creating the NetworkConnect SD 129
Task 2a: Creating the NetworkConnect Sequence Diagram 130
Task 2b: Drawing Messages 130
Task 2c: Drawing Time Intervals 131
Task 2d: Moving Events 132

Exercise 3: Creating the Connection Management Place Call Request Success SD 133
Task 3a: Creating the Connection Management Place Call Request Success Sequence Diagram 134
Task 3b: Drawing the System Border 134
Task 3c: Drawing Classifier Roles 135
Task 3d: Drawing Messages 136
Task 3e: Setting the Features of locationUpdate 137
Task 3f: Moving ConfirmIndication 138
Exercise 4: Animating a Sequence Diagram 139
Task 4a: Changing the Settings for the Debug Configuration 139
Task 4b: Regenerating Code and Rebuilding Your Model 141
Task 4c: Starting Animation 141
Task 4d: Animating a Sequence Diagram 143
Task 4e: Viewing the Browser 145
Task 4f: Quitting Animation 147
Summary 147

Lesson 6: Creating Activity Diagrams 149

Goals for this Lesson 149
Exercise 1: Creating the MMCallControl Activity Diagram 150
Task 1a: Creating an Activity Diagram 151
Task 1b: Drawing Swimlanes 153
Task 1c: Drawing Action Elements 155
Task 1d: Drawing a Default Flow 157
Task 1e: Drawing a Subactivity 157
Task 1f: Drawing Send Action States 158
Task 1g: Drawing Transitions 160
Task 1h: Drawing a Fork Synchronization 162
Task 1i: Drawing a Join Synchronization 163
Task 1j: Drawing a Timeout Transition 165
Task 1k: Specifying an Action on a Transition 165
Exercise 2: Creating the InCall Subactivity Diagram 166
Task 2a: Creating the InCall Subactivity Diagram 167
Task 2b: Drawing Action Elements 167
Task 2c: Drawing a Default Flow 167
Task 2d: Drawing Transitions 168
Task 2e: Drawing a Timeout Transition 169
Exercise 3: Creating the RegistrationMonitor Activity Diagram 170
Task 3a: Creating the RegistrationMonitor Activity Diagram 171
Task 3b: Drawing Action Elements 171
Task 3c: Drawing a Send Action State 172
Task 3d: Drawing a Default Flow 172
Task 3e: Drawing Transitions 173

Task 3f: Drawing a Timeout Transition 173

Exercise 4: Animating the MMCall Control Activity Diagram 174

Task 4a: Regenerating Code and Rebuilding Your Model 174

Task 4b: Animating the MMCall Control Activity Diagram. 175

Summary 177

Lesson 7: Creating Statecharts 179

Goals for this Lesson 179

Exercise 1: Creating the CallControl Statechart. 180

Task 1a: Creating the CallControl Statechart 181

Task 1b: Drawing States 182

Task 1c: Drawing Nested States 182

Task 1d: Drawing Default Connectors. 183

Task 1e: Drawing Send Action States. 184

Task 1f: Drawing Transitions. 186

Task 1g: Drawing a Timeout Transition. 187

Exercise 2: Animating the CallControl Statechart 188

Task 2a: Regenerating Code and Rebuilding the Model. 188

Task 2b: Animating the CallControl Statechart 189

Summary 190

Lesson 8: More Animation 191

Goals for this Lesson 191

Exercise 1: Animating Your Diagrams 192

Task 1a: Preparing for Animation 192

Task 1b: Animating Your Diagrams. 192

Exercise 2: Sending Events to Your Model. 193

Task 2a: Sending an Event to Your Model 193

Task 2b: Sending Another Event 197

Task 2c: Quitting Animation 200

Summary 200

Index 201

Getting Started

Welcome to the *C++ Tutorial for IBM Rational Rhapsody*! IBM® Rational® Rhapsody® is the Model-Driven Development environment of choice for systems engineers and software developers of either embedded or real-time systems. Rational Rhapsody in C++ generates full production C++ code for a variety of target platforms based on UML 2.0 behavioral and structural diagrams. The Rational Rhapsody product also provides for the reverse engineering of C++ code for re-use of your intellectual property within a Model-Driven environment.

Audience for the C++ Tutorial

The intended audience for this tutorial is system engineers and software engineers who are familiar with the C++ language. The tutorial assumes that you are familiar with UML (Unified Modeling Language) and Object Oriented concepts.

Before You Begin

Before you work through this tutorial, you might find it helpful to review the *Getting Started Guide* for the Rational Rhapsody product. It provides a functional overview for the Rational Rhapsody product for system designers, system engineers, and software developers with more functions (meaning how to do something), explanations, and details than this tutorial provides. In addition, throughout the tutorial, references are made to other Rational Rhapsody documentation where appropriate. Note also that the *IBM Rational Rhapsody User Guide* has a Glossary section that you might find useful. Note the following:

- You must have installed the compiler necessary to generate code.
- Before you can work through any of the lessons in this tutorial, you must create the Handset project, which is detailed in **Setting Up the C++ Tutorial**.
- You should work through the tutorial in the order of the lessons. During the course of working through this tutorial, you generate code as well as build your model at various stages. For example, in the lesson where you first learn how to generate code, you will get warning messages. Once you work through the next lesson, you will no longer get those warning messages. In addition, in the later lessons, you set up for animation and work through some initial animation as you go along. Near the end of the tutorial, you

culminate the animation lessons by sending events to your model to see more involved animation.

C++ Tutorial Overview

This tutorial helps you become familiar with the Rational Rhapsody product. You should consider it part of the Rational Rhapsody learning process, in addition, for example, to the *Rhapsody Essential Tool Training* class and the Rational Rhapsody eLearning courses, both of which are available at an additional cost.

This tutorial shows you how to use the Rational Rhapsody product to analyze, design, and build a model of a wireless telephone using a file-based modeling approach. Before you begin creating this model, you need to consider the functions of the wireless telephone. Wireless telephony provides voice and data services to users placing and receiving calls. To deliver services, the wireless network must receive, set up, and direct incoming and outgoing call requests, track and maintain the location of users, and facilitate uninterrupted service when users move within and outside the network.

When the wireless user initiates a call, the network receives the request, and validates and registers the user. Once registered, the network monitors the user's location. In order for the network to receive the call, the wireless telephone must send the minimum acceptable signal strength to the network. When the network receives a call, it directs it to the appropriate registered user.

For this tutorial, you are going to create a project called Handset. The Rational Rhapsody product contains a sample handset model that you can use to compare with the lessons in this tutorial. The sample model is located in the `<Rational Rhapsody installation>\Samples\CppSamples` subfolder.

Note that the official sample Handset model in the Samples subfolder might be different from the model you create when you follow the instructions in this tutorial. While the model you create with the tutorial might have the same name as the official product sample, the tutorial might demonstrate different techniques and features for instructional purposes.

Note

To minimize the complexity of the tutorial, the operations have been simplified to focus on the function of placing a call.

C++ Tutorial Objectives

When you have completed this tutorial, you will have performed the following standard tasks:

- Created a project
- Created use case diagrams, which show the main functions of the system (use cases) and the entities that are outside the system
- Created structure diagrams, which define the system structure and identify the large-scale organizational pieces of the system
- Created object model diagrams, which specify the structure of the classes, objects, and interfaces in the system and the static relationships that exist between them
- Created sequence diagrams, which describe how structural elements communicate with one another over time, and identify the required relationships and messages
- Created activity diagrams, which show the dynamic aspects of a system and the flow of control from activity to activity
- Created statecharts, which define the behavior of classifiers (actors, use cases, or classes), objects, including the states that they can enter over their lifetime and the messages, events, or operations that cause them to transition from state to state
- Generated code
- Built a model
- Animated a model

Documentation Conventions

This document uses the following conventions:

- **Boldtype** for names of GUI objects and controls, including selection choices; and emphasis. Examples:
  - From the **Interface** drop-down list box, select **Out** and click **OK**.
  - Click the **<<Subsystem>> ConnectManagement** object and drag it into the **CM_Subsystem** package.
  - Click the Create Port button on the **Drawing** toolbar click the left edge of the **CallControl** object.
  - If the Rational Rhapsody browser does not display, select **View > Browser**.
  - A project file, called **<project_name>.rpy**.
- `Courier font in 10 point` for pathnames, system messages, and items that you have to type. Examples:
  - To avoid overwriting the Handset sample project provided with the Rational Rhapsody product, do not create your project in `<Rational Rhapsody installation>\Samples\CppSamples`.
  - The Output window displays the message `Animation session terminated`.
  - In the **Project name** box, replace the default project name with `Handset`.
  - Type `cc_in` press **Enter**.
- *Italics* for the first mention of a concept with an explanation.

About the Rational Rhapsody Product

The Rational Rhapsody product is a visual design tool for developing object-oriented embedded software, and performing structural and systems modeling. It enables you to perform these tasks:

- **Analyze**, during which you can define, analyze, and validate the system requirements.
- **Design**, during which you can specify and design the architecture.
- **Implement**, during which you can automatically generate code build and run it within the Rational Rhapsody product.
- **Model Execution**, during which you can animate the model on the local host or a remote target to perform design-level debugging within animated views.

UML Diagrams

The following are the UML diagrams in Rational Rhapsody:

- **Use Case Diagrams** show the main functions of the system (use cases) and the entities (actors) outside the system.
- **Structure Diagrams** show the system structure and identify the organizational pieces of the system.
- **Object Model Diagrams** show the structure of the system in terms of classes, objects, and the relationships between these structural elements.
- **Sequence Diagrams** show sequences of steps and messages passed between structural elements when executing a particular instance of a use case.
- **Activity Diagrams** specify a flow for classifiers (classes, actors, use cases), objects, and operations.
- **Statecharts** show the behavior of a particular classifier (class, actor, use case) or object over its entire life cycle.
- **Collaboration Diagrams** provide the same information as sequence diagrams, emphasizing structure rather than time.
- **Component Diagrams** describe the organization of the software units and the dependencies among units.
- **Deployment Diagrams** show the nodes in the final system architecture and the connections between them.

In addition, **Flow Charts** are available in the Rational Rhapsody product. Flow charts are not in UML. They are a subset of activity diagrams with parts (of the functionality for activity diagrams) excluded. Flow charts have specifically event-driven behavior. You can use a flow chart to describe a function or class operation and for code generation.

Starting the Rational Rhapsody Product

Windows

To start the Rational Rhapsody product in **Windows**: Select **Start** > **Programs** > **IBM Rational** > **IBM Rational Rhapsody Version#** > **Rhapsody Developer Edition** > **Rhapsody in C++**.

Linux

To start the Rational Rhapsody product in **Linux**, use these steps:

1. From the Terminal, browse to the Rational Rhapsody home directory.
2. Execute the `RhapsodyInCPP` script. For example:

   ```
   [RhapsodyUser@MyHostMachine]# cd /home/Rhapsody
   [RhapsodyUser@MyHostMachine Rhapsody]# ./RhapsodyInCpp
   ```

   In this example, “RhapsodyUser” is the username, “`MyHostMachine`” is the host machine and “/home/Rhapsody” is the installation directory.

Closing the Rational Rhapsody Product

To close the Rational Rhapsody product, follow these steps:

1. Save your changes. See **Saving a Project**.
2. Choose **File** > **Exit** or click the Close button .

Setting Up the C++ Tutorial

Before you can work through this tutorial, you must create and set up the Handset project, which you do in this section. The following tasks show you how to:

- Create the Handset project
- Save a project
- Create the packages needed for the Handset project

Creating a Project

To create a new project, follow these steps:

1. Start the Rational Rhapsody product if it is not already running. If necessary, see **Starting the Rational Rhapsody Product**.

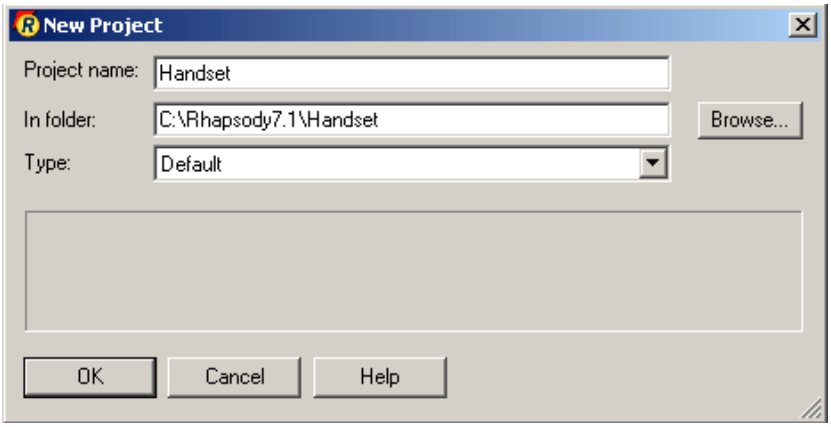
2. Click the New button on the main toolbar or select **File > New**. The New Project dialog box opens.

3. In the **Project name** box, replace the default project name with `Handset`.

4. In the **In folder** box, browse to find an existing folder or enter a new folder name.

   Note: To avoid overwriting the sample Handset project provided with the Rational Rhapsody product, do not create your project in `<Rational Rhapsody installation>\Samples\CppSamples`. Also, to avoid potentially long pathnames, do not create the project on the desktop.

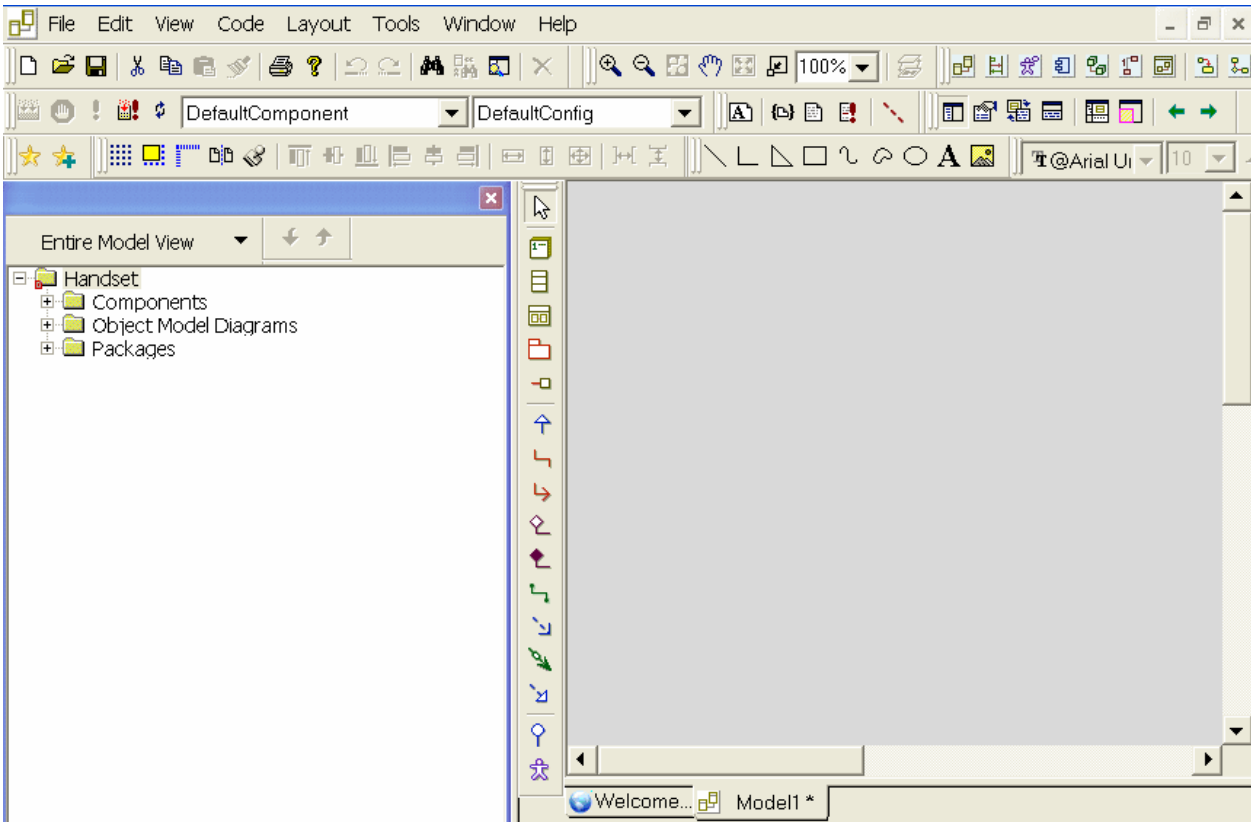
5. In the **Type** box, accept **Default**, which provides all of the basic UML structures. It is useful for most Rational Rhapsody projects. Your dialog box should resemble the following figure:



Note: For a description of the available project profile types that you can select from the **Type** drop-down list, refer to the *IBM Rational Rhapsody User Guide*. (Do a search of the user guide PDF file for "specialized profile.")

6. Click **OK**. The Rational Rhapsody product verifies that the specified location exists. If it does not exist, Rational Rhapsody asks whether you want to create it. Click **Yes**.

   Rational Rhapsody creates your project in the new **Handset** subfolder, opens the project, and displays the Rational Rhapsody browser in the left pane and the drawing area for an object model diagram (by default because of your **Type** [project profile] choice on the **New Project** dialog box), as shown in the following figure:



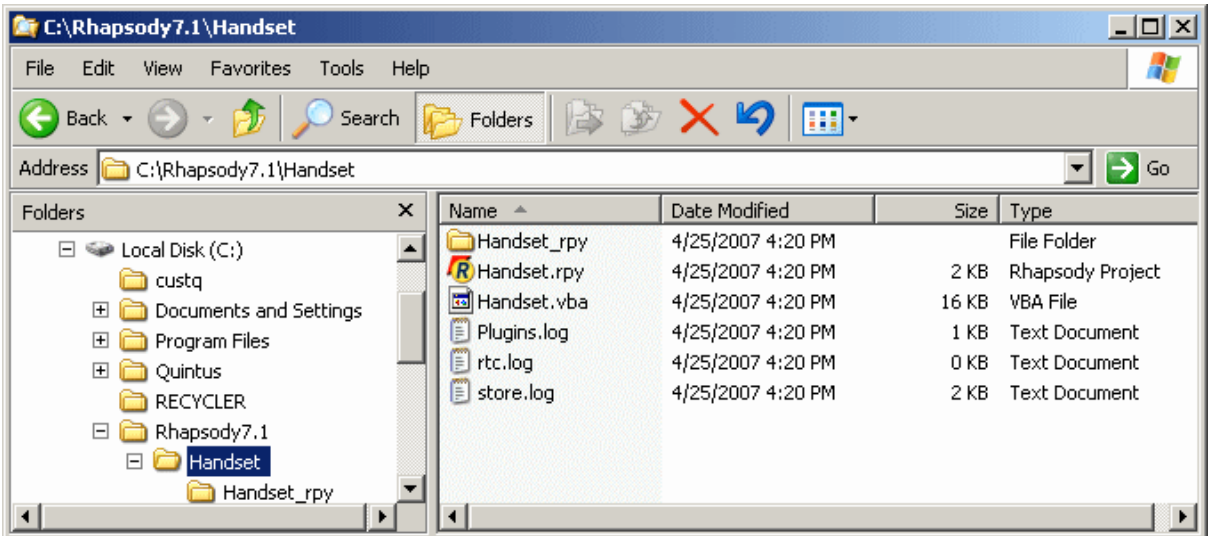
Note: An asterisk (*) in a title bar for the Rational Rhapsody window and any dialog box means that data has been modified and a save has not been done yet.

If the Rational Rhapsody browser does not display, select **View** > **Browser**.

7. Save your project. If necessary, see **Saving a Project**.

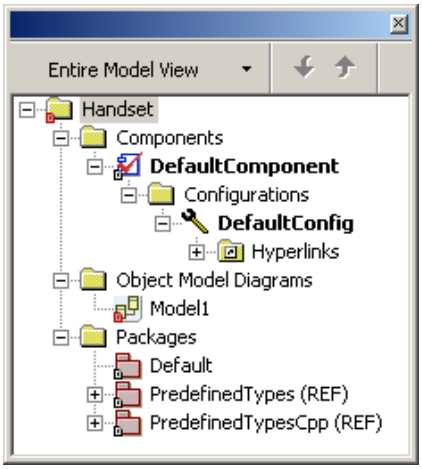
About a Rational Rhapsody Project

A Rational Rhapsody project includes the UML diagrams, packages, and code generation configurations that define the model and the code generated from it. When you create a new project, Rational Rhapsody creates a project folder that contains the *project files* in the specified location. The name you choose for your new project is used to name project files and folders, as shown in the following figure.



For more information about the folders and files that are part of a Rational Rhapsody model, see **About Project Files and Folders**.

In addition, the name appears at the top level of the project hierarchy in the Rational Rhapsody browser. Rational Rhapsody provides several default elements in the new project: a object model diagram, package, component, and configuration, as shown in the following figure:



An *element* is an atomic constituent of a model. In the Rational Rhapsody product, primary model elements within the browser are packages, classes, object model diagrams, associations, dependencies, operations, variables, events, event receptions, triggered operations, constructors, destructors, and types. Primary model elements in object model diagrams are packages, classes, associations (links), dependencies, and actors.

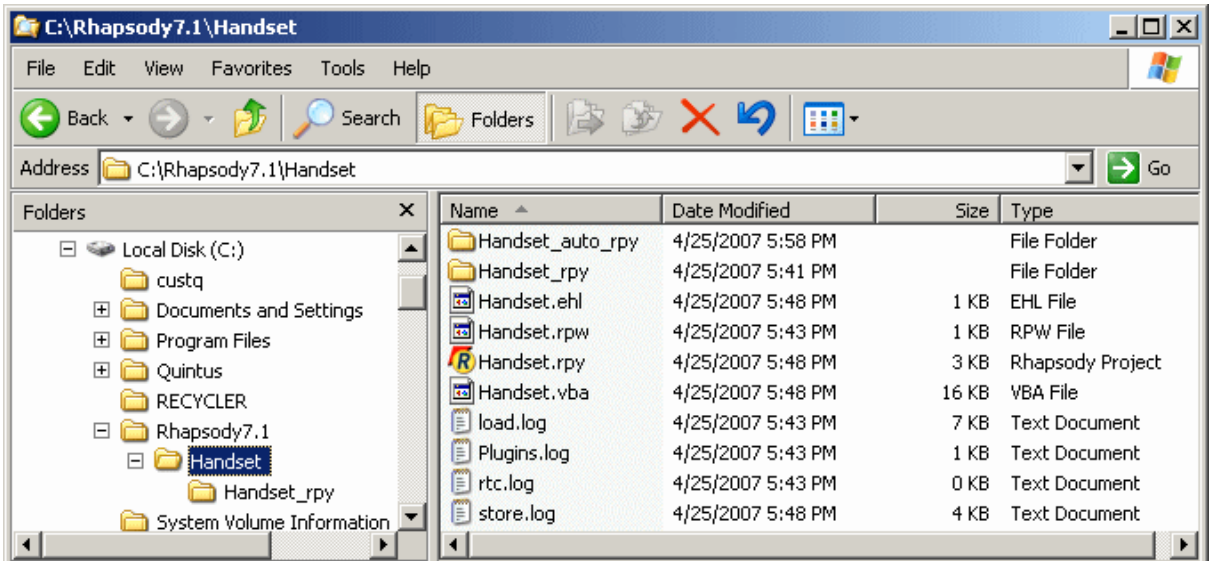
About Project Files and Folders

The Rational Rhapsody product creates the following files and subfolders in the project folder. Some folders and files are created when you initially create a project, others only when applicable.

- A project folder, called **<project_name>_rpy**, which contains the unit files for the project, including UML diagrams, packages, and code generation configurations.
- A project file, called **<project_name>.rpy**.
- A subfolder, called **<project_name>_auto_rpy**, which appears only when necessary (after ten minutes if a save has not been made) and disappears after you save.
- An event history file, called **<project_name>.ehl**, which contains a record of events injected during animation, and active and nonactive breakpoints. This file appears after your first save of a project.
- Log files, which record when projects were loaded and saved in the product; for example, **load.log** and **store.log**.
- A .vba file, called **<project_name_>.vba**, which contains macros or wizards.
- Backup project files and folders (**<project_name>_bak1_rpy**, **<project_name>_bak2_rpy**), which are optional, depending on project settings.
- An **_RTC** subfolder, when applicable, which holds any tests created using the Rational Rhapsody TestConductor™ add-on.

The **<project_name>.rpy** file and the **<project_name>_rpy** folder are necessary for the generation of source code.

The following figure shows the project folder for the Handset project and some of its files and subfolders.



Saving a Project

To save a project in the current location, use one of the following methods:

- Click the Save button on the main toolbar
- Select **File > Save**.

To save the project to a new location, select **File > Save As**.

Note that the **Save** command saves only the modified units, reducing the time required to save large projects.

A *unit* is a composite model element stored in its own file that you can check in and out of a Content Management system. A model element can be made into a unit as long as it can be saved as a separate file. Some elements that can be saved as units are the entire model, packages, classes (in C, objects and object types), any type of Rational Rhapsody diagram, and components. The project, represented by the root node displayed in the browser, is always a unit. The primary purpose of units is to support collaboration with other developers.

About Autosave

The Rational Rhapsody product automatically performs an autosave every ten minutes to back up changes made between saves. Modified units are saved in an autosave folder (**<project_name>_auto_rpy**), along with any units that have a time stamp older than the project file. Note that the autosave folder appears only when necessary (after ten minutes if a save has not been made) and disappears after you save.

About Backups

You can set a property to create backups of your model every time you save your project. This gives you the opportunity to revert to a previously saved version if you encounter a problem. By default, Rational Rhapsody does not create backups. Refer to the *IBM Rational Rhapsody User Guide* for more information about creating backups. (Do a search of the user guide PDF file for “backups.”)

Organizing the Model Using Packages

Packages can be used to divide the model into functional domains or subsystems, which consist of objects, object types, functions, variables, and other logical artifacts. They can be organized into hierarchies to provide a high level of partitioning.

The handset model will have the following main packages:

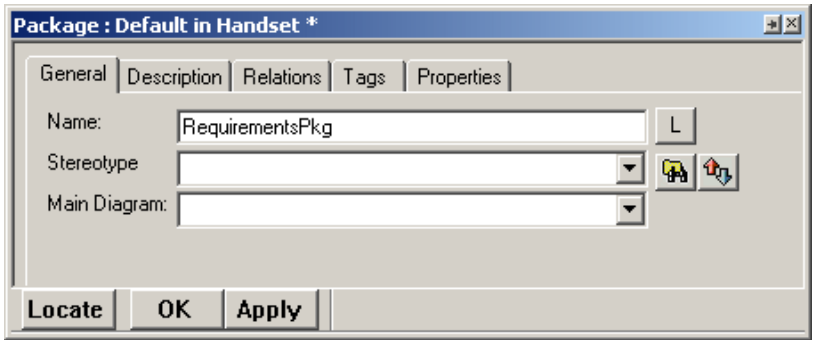
- **RequirementsPkg** to contain the system's functional requirements.
- **AnalysisPkg** to contain the use case diagrams, which identify the requirements of the system.
- **ArchitecturePkg** to contain the structure diagram, which details the design of the system model and the flow of information.
- **SubsystemsPkg** to contain the components of the system.

Note

To establish traceability between analysis and implementation, the **RequirementsPkg**, **AnalysisPkg**, and **ArchitecturePkg** packages can be referenced from the software application model (even if it is a different Rational Rhapsody project) to establish traceability from design to analysis.

To organize the model into packages, follow these steps:

1. In the Rational Rhapsody browser, expand the **Packages** category.
2. Rename the **Default** package:
   a. Double-click the **Default** package, or right-click it and select **Features**. The Features dialog box opens.
   b. On the **General** tab, in the **Name** box, replace `Default` with `RequirementsPkg`, as shown in the following figure:



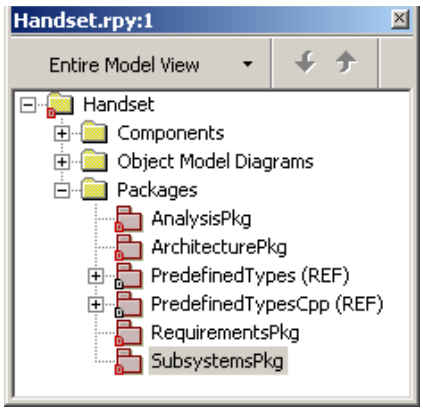
   c. Click **OK**.

3. Create another package:

   a. Right-click **Packages** in the Rational Rhapsody browser and select **Add New Package**. Rational Rhapsody creates a package with the default name **package_n**, where ***n*** is greater or equal to 0.

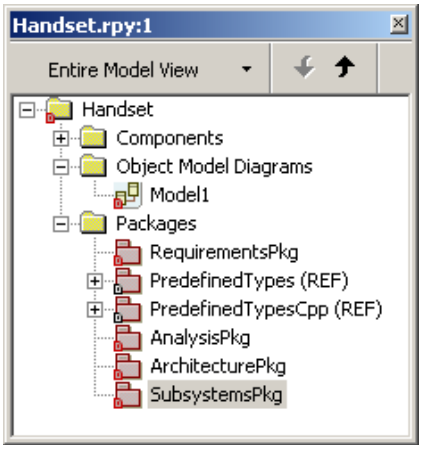
   b. Rename the package `AnalysisPkg` and press **Enter**.

4. Repeat the previous step but create a package named `ArchitecturePkg` and then a package named `SubsystemsPkg`. Your browser should resemble the following figure:



5. To re-order the packages so that the **RequirementsPkg** package is first, do the following:

   a. With focus in the browser, choose **View > Browser Display Options > Enable Ordering**. This activates the Up and Down buttons for the browser. Because the first package you created was **RequirementsPkg**, Rational Rhapsody makes it the first package on the list of packages, as shown in the following figure. If not, go to the next substep.



   Note: By default **Enable Ordering** is not enabled. This means that all elements in the Rational Rhapsody browser appear in alphabetical order. Once you enable the **Enable Ordering** capability, elements are listed on the browser in the order entered. Where allowed, you can re-order the elements in the Rational Rhapsody browser.

   b. If needed, select **RequirementsPkg** in the browser and then click the Up button and move the package to the top of the list.

Hiding Predefined Packages

To unclutter the browser for this tutorial, you can hide the predefined packages (seen in the previous figure showing the browser). To do this, you must modify one of the many properties in the Rational Rhapsody product.

You can modify a Rational Rhapsody property through the **Properties** tab of the Features dialog box. The **Properties** tab lists the properties associated with the selected Rational Rhapsody element.

- The top left column on this tab shows the metaclass and property (for example, **Settings** and **ShowPredefinedPackage**).
- The top right column shows the default for the selected property, if there is one (for example, **Cleared**).
- The box at the bottom portion of the **Properties** tab shows the definition for the property selected in the upper left column of the tab. The definition display shows the names of the subject, metaclass, property, and the definition for the property.

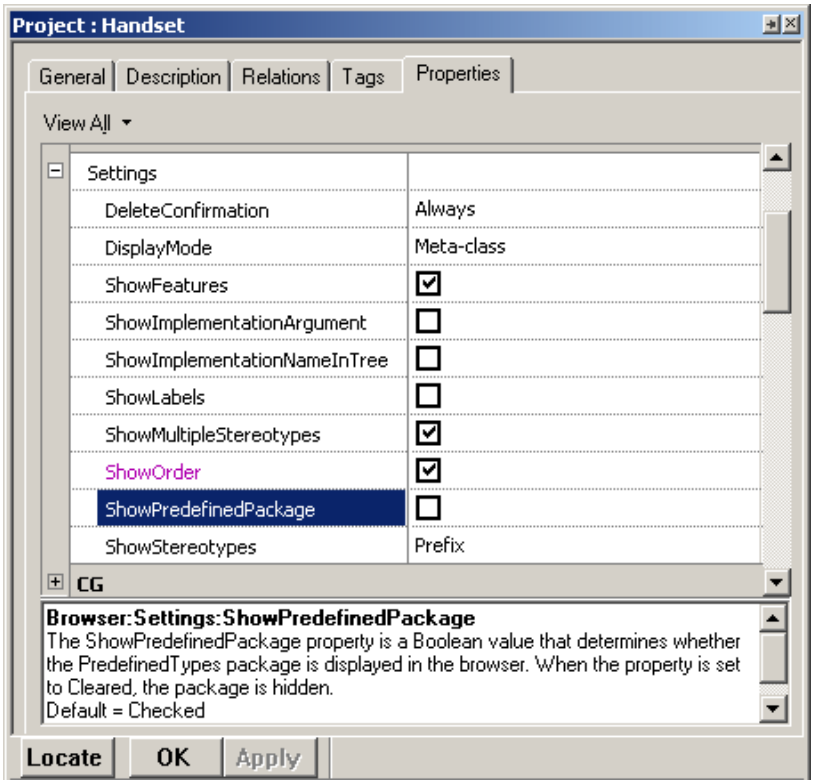
Note that Rational Rhapsody documentation uses a notation method with double colons to identify the location of a specific property, for example, `Browser::Settings::ShowPredefinedPackage`. In this example, **Browser** is the name of the subject, **Settings** is the name of the metaclass, and **ShowPredefinedPackage** is the name of the property.

Refer to the *IBM Rational Rhapsody User Guide* for more information on setting properties. (Do a search of the user guide PDF file for “properties tab.”)

To hide the predefined packages, follow these steps:

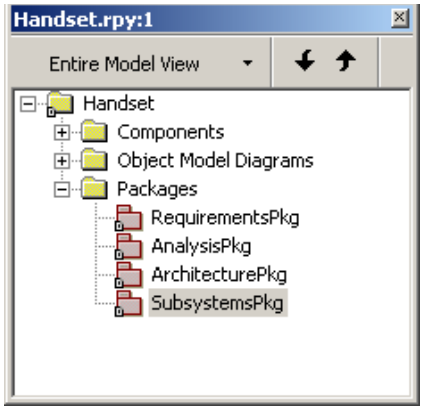
1. Choose **File** > **Project Properties**.
2. On the **Properties** tab, select **All** from the drop-down menu. (The label appears as **View All** after you make the selection.)
3. Scroll down to and expand the **Browser** subject expand the **Settings** metaclass.

4. Clear the check box for the **ShowPredefinedPackage** property, as shown in the following figure:



5. Click **OK**.

6. Click the Save button to save your project. Your Rational Rhapsody browser should resemble the following figure:



Opening the Handset Model

Once you have created, saved, and closed the handset model, you can open and work on it at any time.

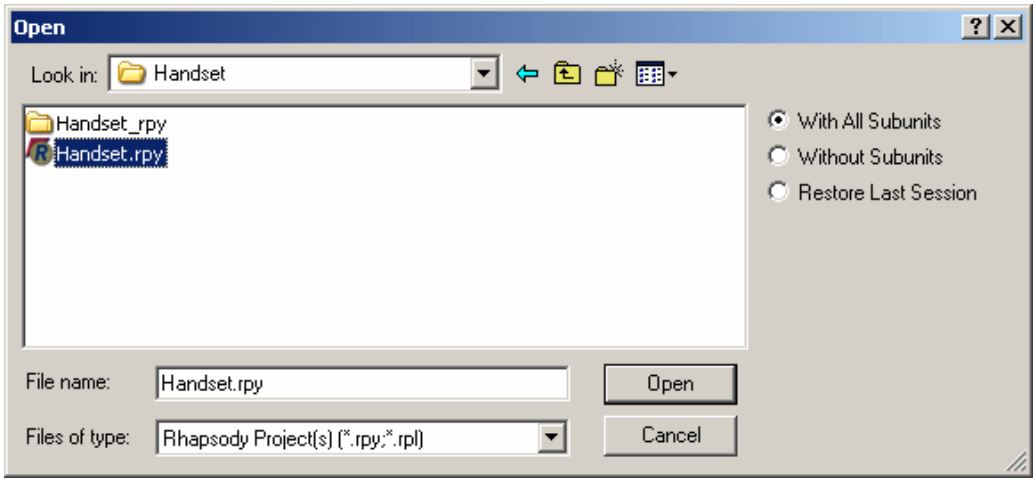
To open the handset model, follow these steps:

1. Start Rational Rhapsody if it is not already running. If necessary, see **Starting the Rational Rhapsody Product**.

2. Click the Open button on the main toolbar or select **File > Open**. The Open dialog box opens.

3. Navigate to the location in which you saved the Handset project.

4. Select **Handset.rpy**, or type the name of the project file in the **File name** box, as shown in the following figure:



5. Accept the default **With All Subunits** option.

   This choice means that the Rational Rhapsody product will load all units in the project. Refer to the *IBM Rational Rhapsody User Guide* for information about the options. (Do a search of the user guide PDF file by the option names.)

6. Click **Open**. Rational Rhapsody opens the handset model.

Using Naming Conventions

To assist all members of your team in understanding the purpose of individual items in the model, it is a good idea to define naming conventions. These conventions help team members to read a diagram quickly and to remember model element names easily.

Note

Remember that the names used in the Rational Rhapsody models are going to be automatically written into the generated code. Therefore, the names should be simple and clearly label all of the elements. Note also that since the C++ language is CASE sensitive, typing errors can prevent the model from building.

Prefixes

Lower and upper case prefixes are useful for model elements. The following is a list of common prefixes with examples of each:

- Event names = “ev” (evStart)
- Trigger operations = “op” (opPress)
- Condition operations = “is” (isPressed)
- Interface classes = “I” (IHardware)

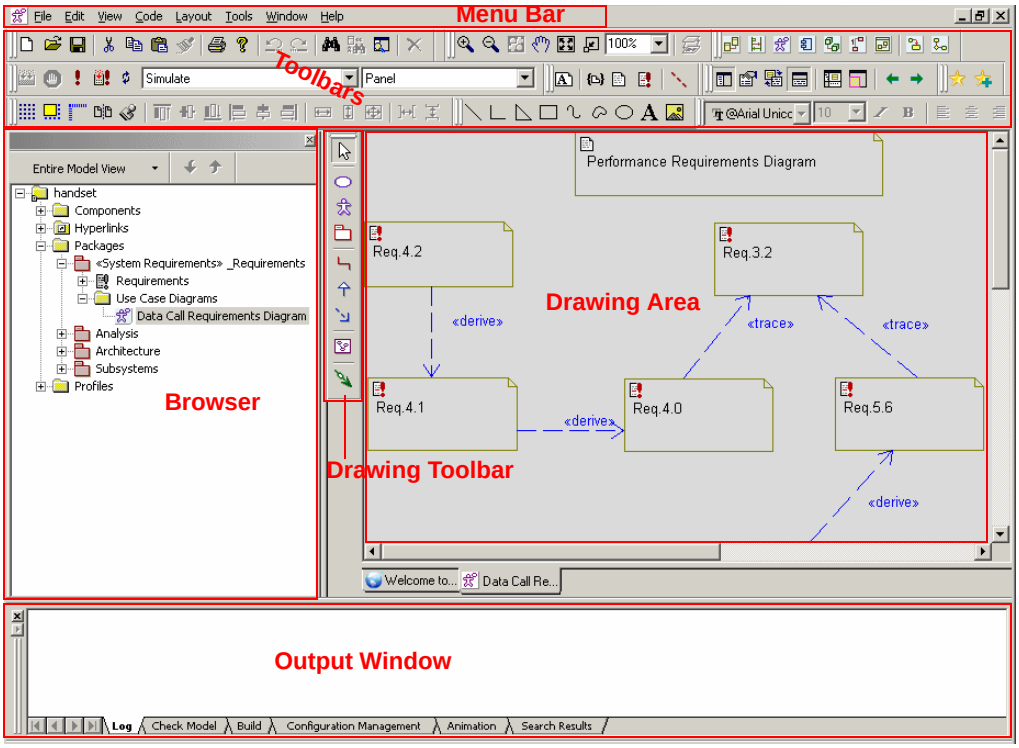
Model Element Names

The names of the elements themselves should follow conventions, such as these:

- Block and class names begin with an upper case letter, such as “System.”
- Operations and attributes begin with lower case letters, such as “restartSystem.”
- Upper case letters to separate concatenated words, such as “checkStatus.”

Rational Rhapsody User Interface

Before proceeding with this tutorial, you should become familiar with the main features of the Rational Rhapsody graphical user interface (GUI). The Rational Rhapsody GUI is made up of three key windows and different toolbars for each of the UML diagram types. The following figure shows the Rational Rhapsody GUI.



Toolbars

The Rational Rhapsody toolbars provide quick access to the commonly used commands. These commands are also available from the menus. The Rational Rhapsody product has the following toolbars:

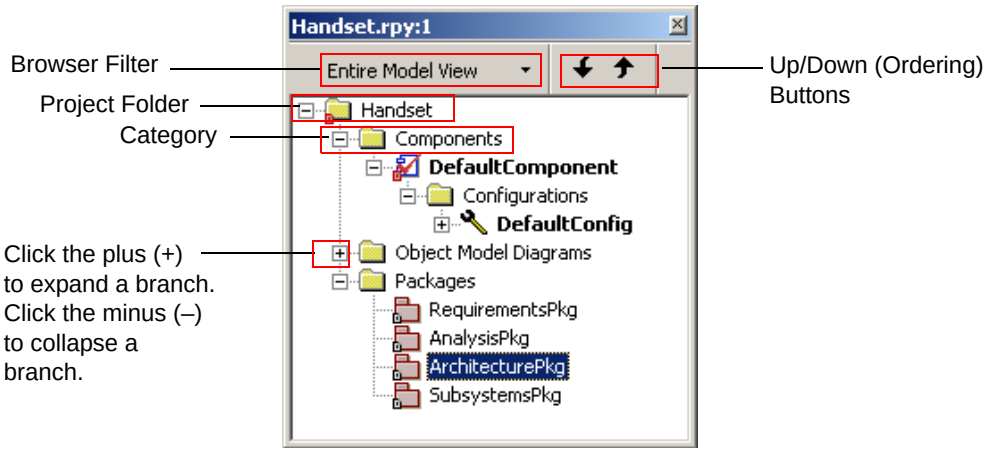
- **Standard** has buttons for the frequently used options on the File, Edit, and Help menus. Examples: New, Open, Save; Copy, Paste, Locate in Browser; About.
- **Code** has buttons for the frequently used options on the Code menu, such as Make, Run Executable and G/M/R (for Generate/Make/Run).
- **Windows** has buttons for the frequently used options on the View menu, such as Show/ Hide Browser and Show/Hide output window.
- **Diagrams** has buttons for the part of the Tools menu that give you quick access to the diagrams in the project, such as Sequence Diagrams and Open Statechart.
- **VBA** provides access to the VBA options, such as **VBA Editor** and **Show Macros Dialog**. Note that VBA is for Windows only.
- **Animation** has buttons for the animation options during an animation session, such as Go, Animation Break, and Quit Animation.
- **Layout** has buttons that help you with the layout of elements in your diagram, such as Snap to Grid, Align Top, and Align Left.
- **Drawing** has buttons for the graphics editor used to create and edit diagrams. Each **Drawing** toolbar is unique to its particular diagram type. For example, the **Drawing** toolbar for a sequence diagram is different from that for a statechart.
- **Common Drawing** has buttons to add requirements, comments, and other annotations to any diagram, such as Note and Requirement.
- **Free Shapes** has buttons for basic drawing shapes, such as Polyline and Polycurve.
- **Zoom** has buttons to zoom options, such as Zoom In, Zoom Out, and Pan.
- **Format** has buttons for various text formatting options and line/fill options, such as Italic and Font Color.

Refer to the *IBM Rational Rhapsody User Guide* for detailed information about the toolbars.

Browser

The Rational Rhapsody browser shows the contents of the project in an expandable tree structure. By default, it is the upper, left-hand part of the Rational Rhapsody GUI. The top-level folder, which contains the name of the project, is the *project folder* or *project node*. Although this folder contains no elements, the folders that reside under it contain elements that have similar characteristics. These folders are referred to as *categories*.

A project consists of at least one package in the **Packages** category. A package contains UML elements, such as classes, files, and diagrams. Rational Rhapsody automatically creates a default package called **Default**, which it uses to save model parts unless you specify a different package. The following figure shows an example of the browser.



Filtering the Browser

The browser filter lets you display only the elements relevant to your current task.

To filter the Rational Rhapsody browser, click the drop-down arrow at the top of the browser window, and select the view you want to see from the menu. Refer to the *IBM Rational Rhapsody User Guide* for information on the view options.

Repositioning the Browser

To make more room for you to work on diagrams, you can move the browser outside of the Rational Rhapsody GUI and reposition it as a separate window on the desktop. To reposition the Rational Rhapsody browser, click the bar at the top of the browser and drag it to another desktop location.

Drawing Area

The drawing area displays the graphic editors and code editors, and it is the region for drawing diagrams. By default, it is the upper, right-hand section of the Rational Rhapsody GUI. Rational Rhapsody displays each diagram with a tab that includes the name of the diagram and an icon that denotes the diagram type. When you make changes to a diagram, Rational Rhapsody displays an asterisk after the name of the diagram in the title bar to indicate that you must save your changes.

Output Window

The Output window displays Rational Rhapsody messages. By default, it is the lower section of the Rational Rhapsody GUI. It includes tabs that display the following types of messages:

- Log
- Check Model
- Build
- Configuration Management
- Animation
- Search results

If the Output window does not appear, choose **View > Output Window**.

Drawing Toolbars

The Rational Rhapsody product displays a separate **Drawing** toolbar for each UML diagram type. By default, it places the **Drawing** toolbar to the left of the diagram.

To move the toolbar, click and drag it to another location.

Features Dialog Box

The Features dialog box lets you view and edit the features of an element in the Rational Rhapsody product.

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

- Double-click an element (for example, **Out** [an interface])
- Right-click an element (for example, **Subsystem Architecture** [a diagram]) select **Features**
- Select an element and press **Alt + Enter**
- Select an element and select **View > Features**

You can resize the Features dialog box and hide the tabs on it if you want. For more information about the Features dialog box, refer to the section on it in the *IBM Rational Rhapsody User Guide*.

Keeping Open the Features Dialog Box

Once you open the Features dialog box, you can leave it open and select other elements to view their features. This means that after you make changes to the Features dialog box for an element in your drawing or on the Rational Rhapsody browser, you can click **Apply**. Then, without closing the dialog box, you can select another element to view its features. Once you are done with the Features dialog box, you click **OK** to close it.

Note

Even though you clicked **Apply** or **OK** for your changes in the Features dialog box, you must still save your model to save all the changes you made. Clicking **Apply** or **OK** applies any changes to the currently opened model. However, to save the changes for the model so that they are in effect the next time you open it, you must save your model.

Note the following about the **Apply** and **OK** buttons on the Features dialog box:

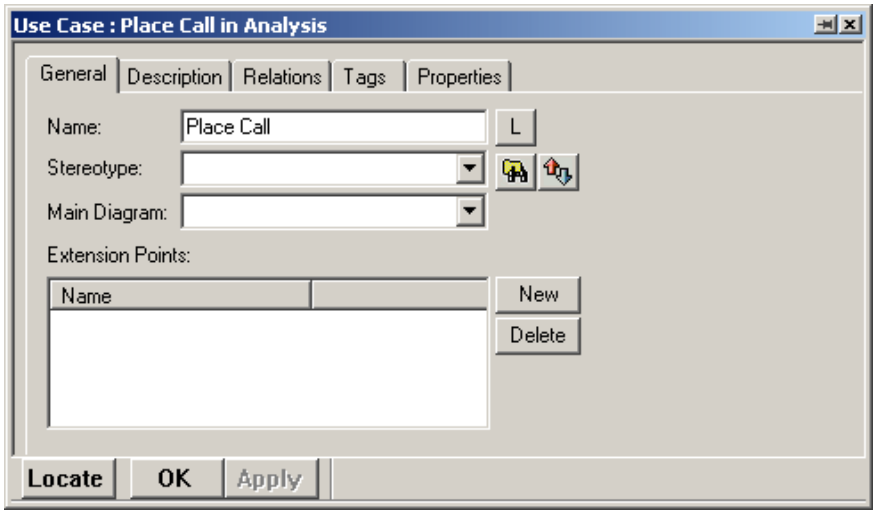
- Click **Apply** when you want to apply any changes you made to the Features dialog box but want keep it open. For example, you might need to apply a change before you can continue with using the Features dialog box, or you want to apply a change and see its effect before continuing making any more changes on the dialog box.
- Click **OK** when you want to apply your changes and close the Features dialog box at the same time.

Tabs for the Features Dialog Box

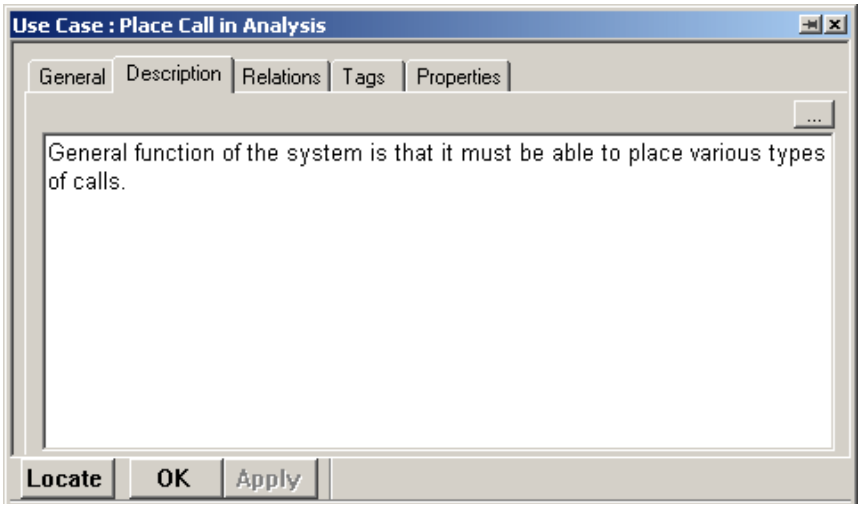
The Features dialog box has different tabs at the top of the dialog box and different boxes on the tabs depending on the element type.

The following tabs are common to all types of elements. For more information about these tabs, as well as the other tabs that you might see in the Features dialog box, refer to the section on it in the *IBM Rational Rhapsody User Guide.*

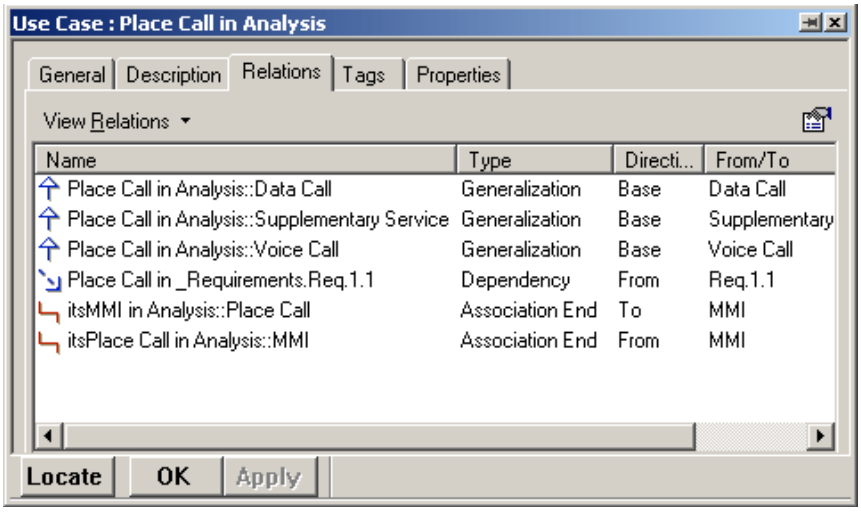
- **General** typically contains the name of the element and other general options, as shown in the following figure:



- **Description**, as its title implies and as shown in the following figure, contains the description of the element, if it has been included.

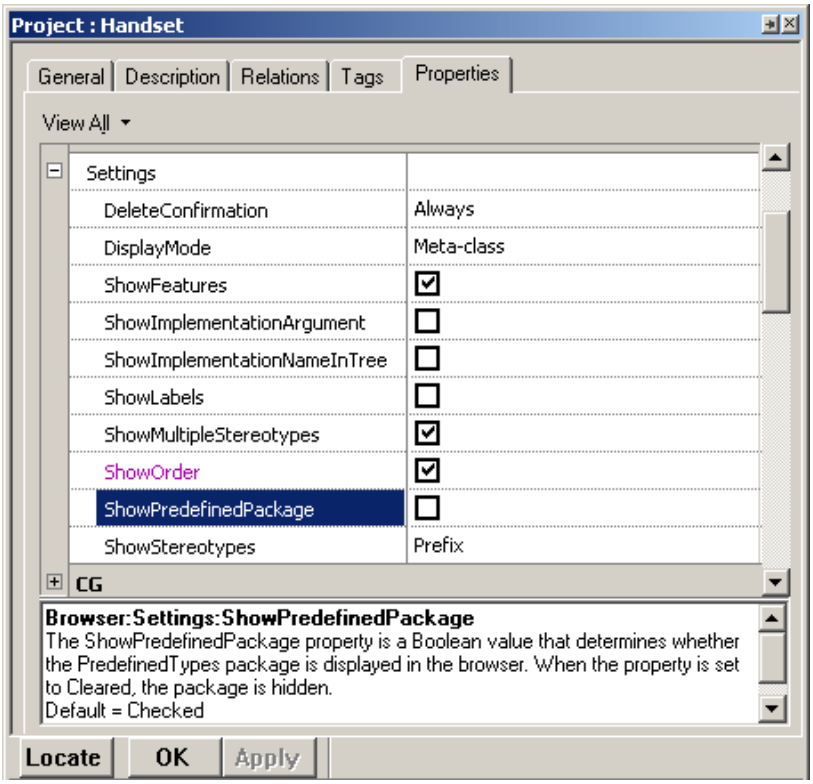


- **Relations** lists all the relationships (dependencies, associations, and so on) an element is engaged with, as shown in the following figure:



- **Tags** lists any tags available for an element. *Tags* enable you to add information to certain kinds of elements to reflect characteristics of the specific domain or platform for the modeled system. Refer to the *IBM Rational Rhapsody User Guide* for more information about tags.

- **Properties** lists the properties associated with the Rational Rhapsody element.
  - The top left column on this tab shows the metaclass and property (for example, **Settings** and **ShowPredefinedPackage**).
  - The top right column shows the default for the selected property, if there is one (for example, **Cleared**).
  - The box at the bottom portion of the **Properties** tab shows the definition for the property selected in the upper left column of the tab. The definition display shows the names of the subject, metaclass, property, and the definition for the property, as shown in the following figure:



Note: Rational Rhapsody documentation uses a notation method with double colons to identify the location of a specific property. For example, for the property in the above figure, the location is `Browser::Settings::ShowPredefinedPackage` where `Browser` is the subject, `Settings` is the metaclass, and `ShowPredefinedPackage` is the property.

Moving the Features Dialog Box

The Features dialog box is a floating window that can be positioned anywhere on the screen, or docked to the Rational Rhapsody GUI.

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

- Double-click the title bar. The dialog box docks. You can now drag it to another location if you want.
- Right-click the title bar and select **Docking by Drag**. Then drag the dialog box to another location.

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

- Double-click the title bar to undock it.
- Right-click the title bar and clear **Docking by Drag** drag the dialog box to another location.

Summary

In this section, you became familiar with the Rational Rhapsody product and its features. You performed the following:

- Created the Handset project
- Saved the project
- Created and organized packages needed for the project

You are now ready to proceed to the next sections where you are going to create the handset model. In the next section, you are going to model the requirements of the wireless telephone and the functions of placing a call using use case diagrams.

For ease of presentation, this tutorial organizes the sections by diagram type and general workflow. However, when modeling systems, diagrams are often created in parallel or might require elements in one diagram to be planned or designed before another diagram can be finalized. For example, you might identify the communication scenarios using sequence diagrams before defining the flows, flow items, and port contracts in the structure diagrams. In addition, you might perform black-box analysis using activity diagrams, sequence diagrams, and statecharts; and white-box analysis using sequence diagrams before decomposing the system's functions into subsystem components.

When you do *black-box analysis*, such as when you do a black-box sequence diagram, you are showing the sequence of messages between external actors and the system as a whole. When you do *white-box analysis*, such as when you do a white-box sequence diagram, you are showing messages to and from the internal individual parts.

Lesson 1: Creating Use Case Diagrams

Use case diagrams (UCDs) show the main functions of the system (use cases) and the entities that are outside the system (actors). Use case diagrams allow you to specify the requirements for the system and show the interactions between the system and external actors.

Note

You must complete all the tasks in **Setting Up the C++ Tutorial** in the **Getting Started** section before you start this lesson.

Goals for this Lesson

In this lesson, you are going to create the following use case diagrams:

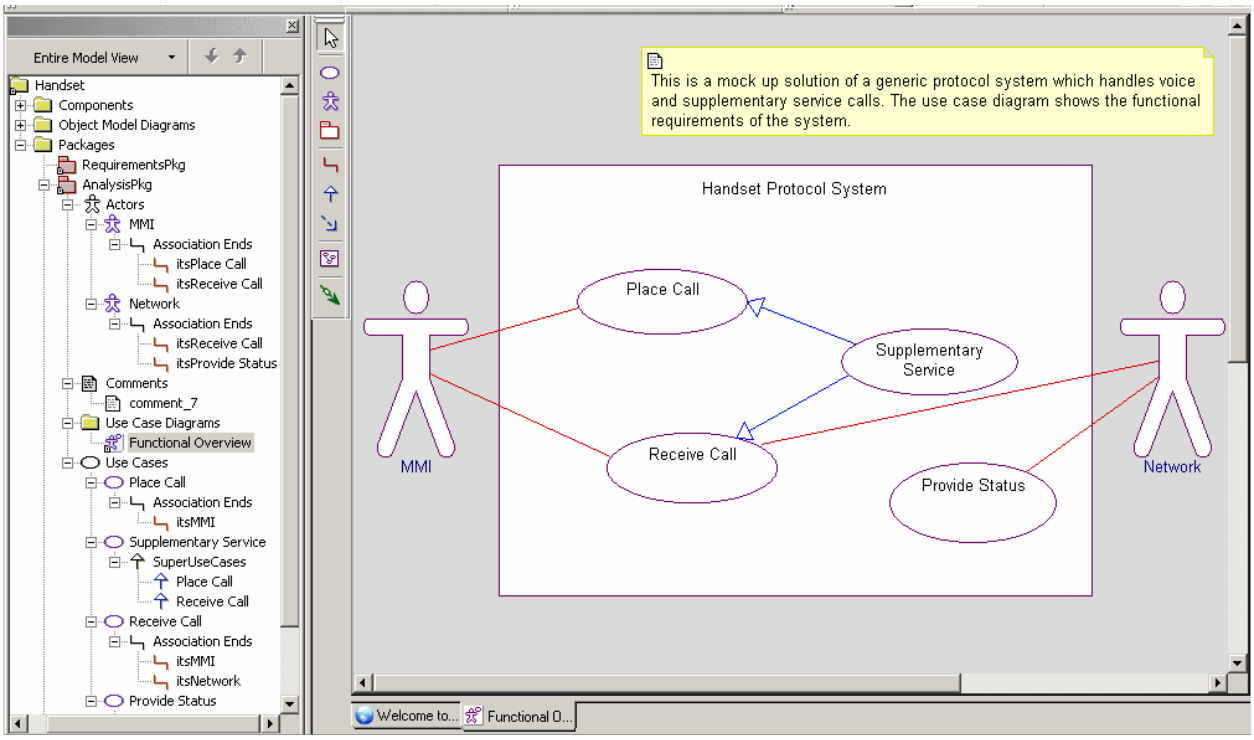
- **Functional Overview** to show the requirements and functions of the handset.
- **Place Call Overview** to show the functions of placing a call.
- **Data Call Requirements** to show the relations among requirement elements.

Exercise 1: Creating the Functional Overview UCD

In this exercise you are going to create the Functional Overview use case diagram. This UCD shows the system requirements, including the actors, the major use cases of the system, and the relationships between them.

The following figure shows the Functional Overview use case diagram that you are going to create in this exercise.

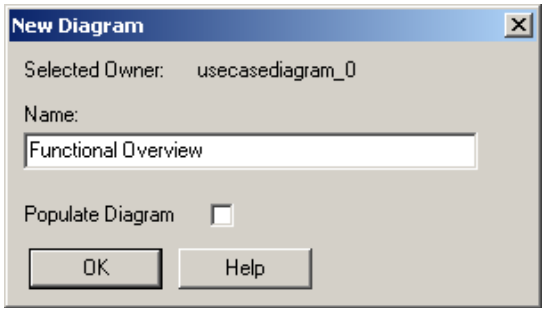
Functional Overview Use Case Diagram



Task 1a: Creating the Functional Overview Use Case Diagram

To create the Functional Overview use case diagram, follow these steps:

1. Start Rational Rhapsody and open the handset model if they are not already open.
2. In the Rational Rhapsody browser, expand the **Packages** category, then right-click the **AnalysisPkg** package, and then select **Add New > Use Case Diagram**. The New Diagram dialog box opens.
3. Type `Functional Overview`, as shown in the following figure, and then click **OK**.



Rational Rhapsody automatically adds the **Use Case Diagrams** category and the name of the new diagram to the browser, as shown in the **Functional Overview Use Case Diagram** figure, and opens the new diagram in the drawing area.

Note

You can also create a diagram by using the Tools menu or the **Diagrams** toolbar. Also, once you create a diagram you can open it using the **Diagrams** toolbar. Refer to the *IBM Rational Rhapsody User Guide* for more information.

Preparing to Draw the Functional Overview UCD

Before drawing the Functional Overview use case diagram, you must identify the system requirements including the actors, the major use cases of the system, and the relationships between them.

For this Functional Overview use case diagram, these actors interact with the system:

- **MMI** represents the handset user interface, including the keypad and display
- **Network** represents the system network or infrastructure of the signalling technology

The major use cases of the system are:

- The handset enables users to place and receive calls.
- The network receives incoming and outgoing call requests, and tracks users.

The actors and the system relate to each other in the following ways:

- **MMI** places and receives calls.
- **Network** tracks users, monitors signal strength, and provides network status and location registration.

You draw a use case diagram using the following general steps:

1. Draw the boundary box.
2. Draw the actors outside of the boundary box.
3. Draw the use cases inside the boundary box.
4. Associate the use cases with the actors.

Task 1b: Drawing the Boundary Box and Actors

The boundary box delineates the system under design from the external actors. Use cases are inside the boundary box; actors are outside the boundary box. In this task, you are going to draw the boundary box and actors using the **Functional Overview Use Case Diagram** figure as a reference.

To draw the boundary box and actors, follow these steps:

1. Click the Create Boundary Box button on the **Drawing** toolbar.
2. Click in the upper, left corner of the drawing area and drag to the lower right. Rational Rhapsody creates a boundary box, named `System Boundary Box`.
3. Rename the boundary box `Handset Protocol System` and then press **Enter**.
4. Click the Create Actor button on the **Drawing** toolbar.
5. Click the left side of the drawing area. Rational Rhapsody creates an actor with a default name of **actor_n**, where **n** is greater than or equal to 0.
6. Rename the actor `MMI` and then press **Enter**.

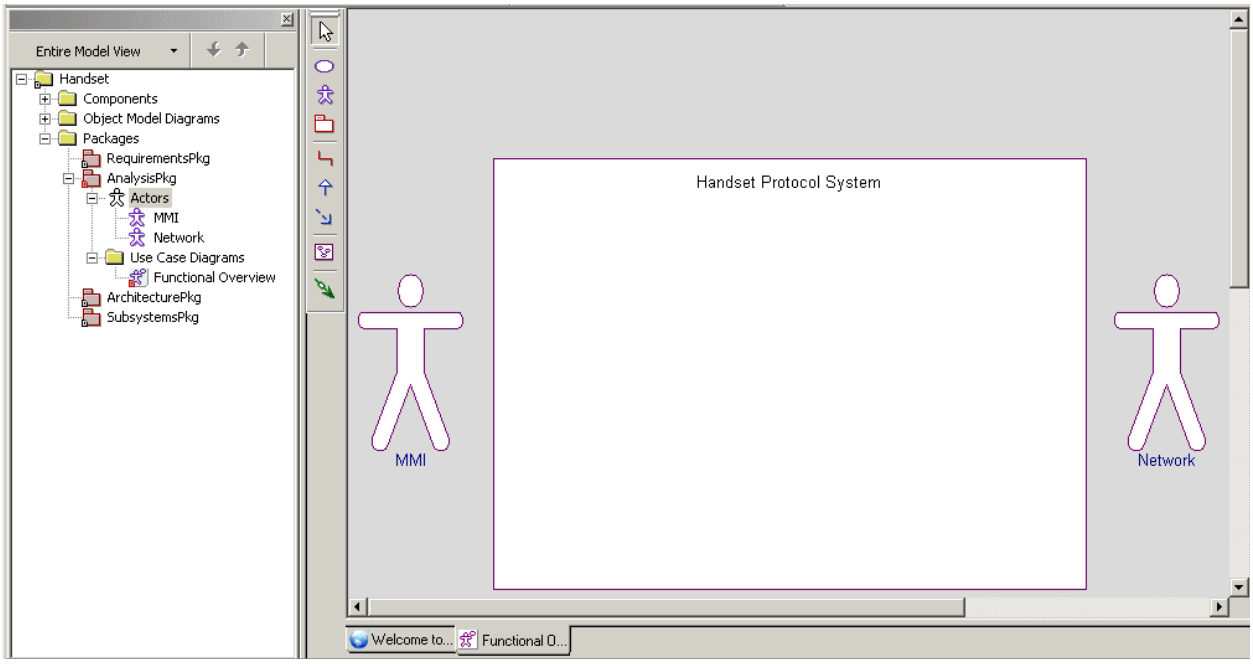
> **Note:** Because code can be generated using the specified names, do not include spaces in the names of actors.

7. Draw an actor on the right side of the drawing area named `Network`.

   Note: You can use the tools on the **Layout** toolbar to help you with the layout of selected elements (including labels) in your diagram. For example, you can select **MMI** and **Network** and use Align Bottom to align them to be on the same bottom edge or Same Size to resize them so that they are the same size. Keep in mind that the last element selected is used as the default. Refer to the *IBM Rational Rhapsody User Guide* for more information about the **Layout** toolbar.

   In addition if you want to move a drawn element on a drawing more precisely than clicking it and dragging it, click one or more elements, press the Ctrl key and use the standalone directional arrow keys to move your element(s). You can also use the directional arrows on the numeric keypad with NumLock not active.

8. In the browser, expand the AnalysisPkg package and the Actors category to view your newly created actors, as shown in the following figure:



Note: To quickly find the **Actors** category in the Rational Rhapsody browser, right-click an actor on the use case diagram and click **Locate** or press **Ctrl+L**. You can use this technique with other objects on a diagram.

Task 1c: Drawing the Use Cases

A *use case* represents a particular function of the system. The Functional Overview use case diagram has the following use cases:

- **Place Call** to show that the user can place various types of calls.
- **Supplementary Service** to show that the system can provide services, such as messaging, call forwarding, call holding, call barring, and conference calling.
- **Receive Call** to show that the system can receive various types of calls.
- **Provide Status** to show that the system can provide network status, user location, and signal strength.

In this task, you are going to draw use cases using the **Functional Overview Use Case Diagram** figure as a reference.

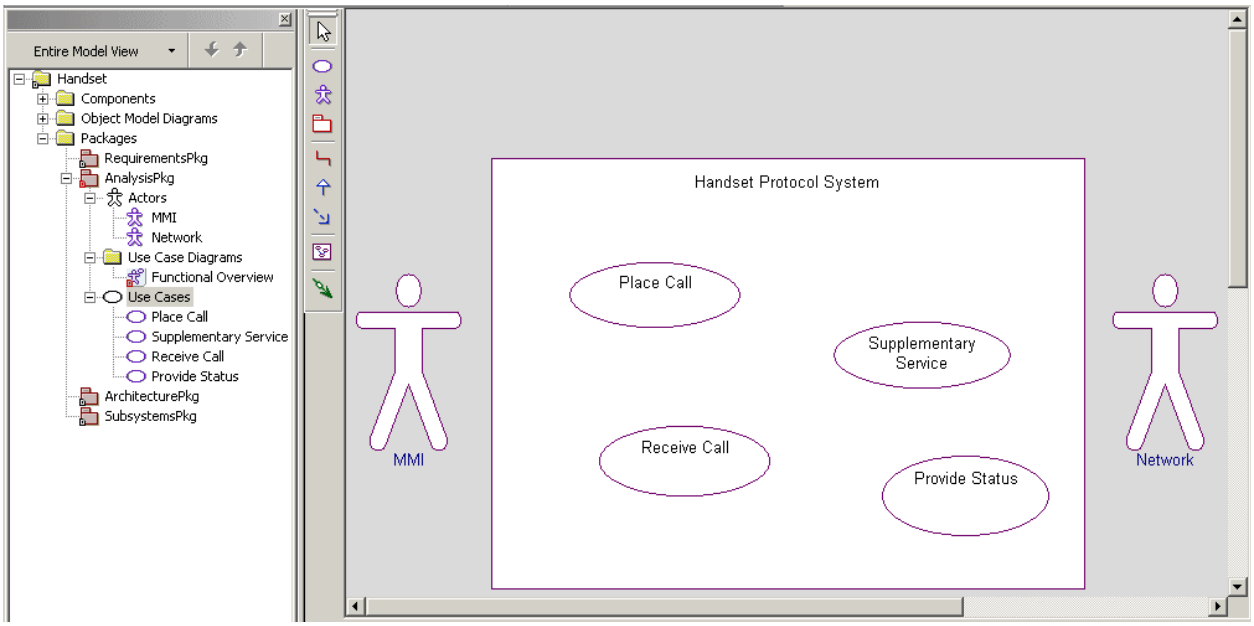
To draw the use cases, follow these steps:

1. Click the Create Use Case button on the **Drawing** toolbar.
2. Click inside the upper left of the boundary box. Rational Rhapsody creates a use case with a default name of **usecase_n**, where **n** is equal to or greater than 0.
3. Rename the use case `Place Call` and then press **Enter**.

> **Note:** For use case names, you can use spaces because use case names do not correspond to actual generated code. In the previous task where you drew actors, you did not use spaces in actor names because code can be generated using the specified actor names.

4. Create three more use cases inside the boundary box named `Supplementary Service`, `Receive Call`, and `Provide Status`.

   In the browser, expand the **AnalysisPkg** package and the **Use Cases** category to view the use cases you created, as shown in the following figure:



Task 1d: Defining Use Case Features

In this task, you define use case features. You can define the features of a use case, enter a description, and do other things through the use of the Features dialog box. You can access the Features dialog box from the browser or the diagram.

To define use case features, follow these steps:

1. In the browser, if necessary, expand the **AnalysisPkg** package and **Use Cases** category.
2. Double-click the **Place Call** use case, or right-click and select **Features**. The Features dialog box opens.

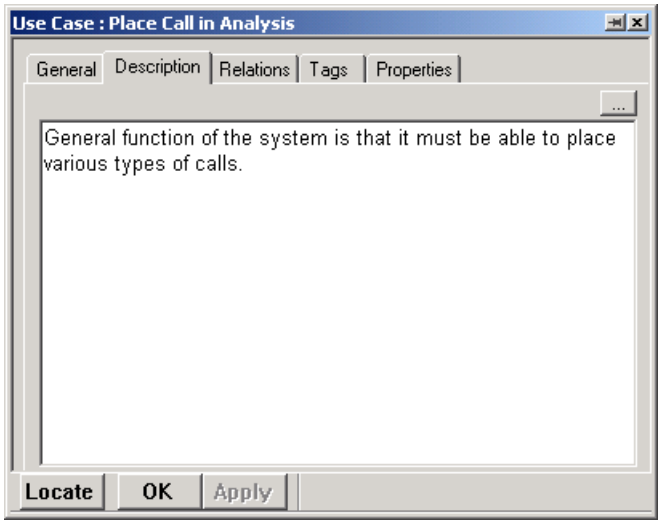
   Note: You can also open the use case through the Functional Overview use case diagram: Double-click the **Place Call** use case, or right-click the use case and select **Features**.

3. On the **Description** tab, type the following text to describe the purpose of this use case:

   ```
   General function of the system is that it must be able to place various
   types of calls.
   ```

   Note: You can also click the Ellipsis button just above the **Description** box on the **Description** tab to expand the internal text editor. When you have entered the description, click **OK** to close the Text Editor dialog box and return to the Features dialog box.

   Your **Description** tab should resemble the following figure:



4. Click **Apply**.

5. With the Features dialog box still open (if necessary, see **Keeping Open the Features Dialog Box**), for the other use cases, type a description for each as follows:

   - For the **Supplementary Service** use case:

     ```
     A supplementary service is a short message, call forwarding, call
     holding, call barring, or conference calling.
     ```

   - For the **Receive Call** use case:

     ```
     General function of the system is that it must be able to receive
     and terminate calls.
     ```

   - For the **Provide Status** use case:

     ```
     The system must be able to communicate with the network in order to
     show the user the visual status such as signal strength and current
     registered network. It must also handle user requests for network
     status and location registration.
     ```

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

7. Click the Save button to save your model.

Task 1e: Associating Actors with Use Cases

The **MMI** actor places calls and receives calls. The **Network** actor notifies the system of incoming calls and provides status. You want to show the associations between actors and the relevant use cases using association lines. An *association* represents a connection between objects or users. In this task, you associate actors with use cases using the **Functional Overview Use Case Diagram** figure as a reference.

To draw association lines, follow these steps:

1. Click the Create Association button on the **Drawing** toolbar.

   Once you move your cursor over the drawing area, notice that the mouse pointer turns into a crosshairs pointer to signify that it is enabled and that it changes into a circled crosshairs pointer when drawing is possible.

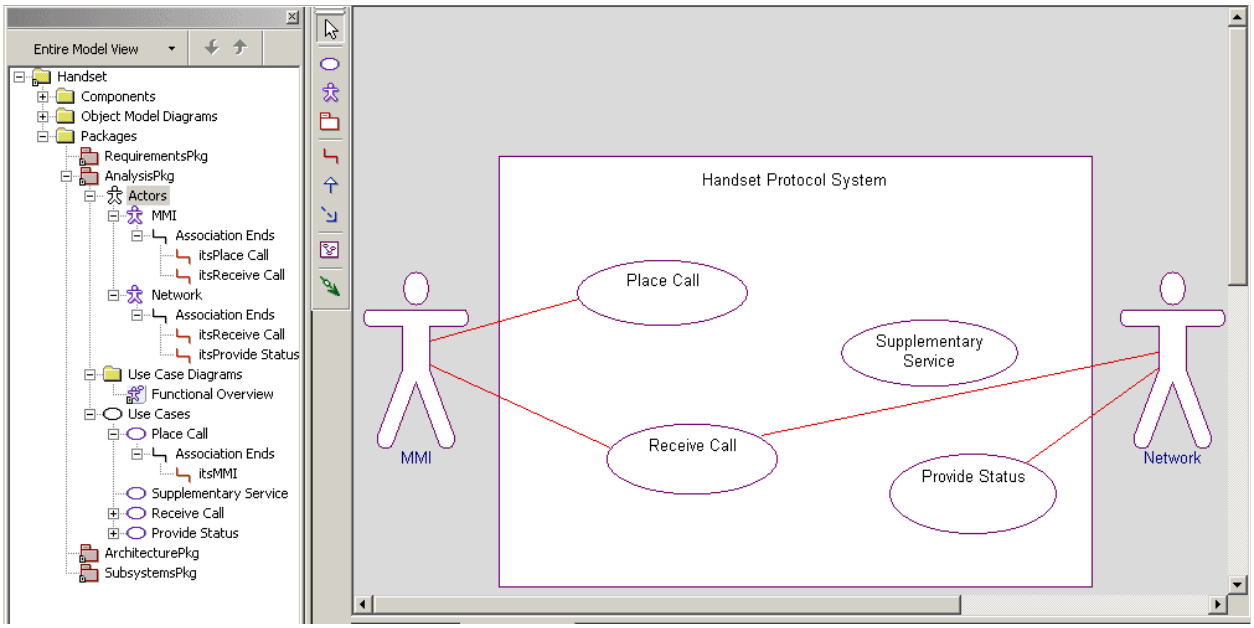
2. Click the edge of the **MMI** actor and then click the edge of the **Place Call** use case. Rational Rhapsody creates an association line with the name label highlighted. You do not need to name this association, so click the mouse button again (this is the same as pressing **Enter**).

3. Create an association between the **MMI** actor and the **Receive Call** use case and then click the mouse button again or press **Enter**.

4. Create an association between the **Network** actor and the **Receive Call** use case.

5. Create an association between the **Network** actor and the **Provide Status** use case.

6. In the Rational Rhapsody browser, expand the **Actors** category to view the relations for the actors and use cases, as shown in the following figure:



The **MMI** actor has two new relations:

- **itsPlace Call** is the role played by the **Place Call** use case in relation to this actor.
- **itsReceive Call** is the role played by the **Receive Call** use case in relation to this actor.

The **Network** actor also has two new relations:

- **itsProvide Status** is the role played by the **Provide Status** use case in relation to this actor.
- **itsReceive Call** is the role played by the **Receive Call** use case in relation to this actor.

Task 1f: Drawing Generalizations

A *generalization* is a relationship between a general element and a more specific element. The more specific element inherits the properties of the general element and is substitutable for the general element. A generalization lets you derive one use case from another.

The **Supplementary Service** use case is a more specific case of placing a call, and it is a more specific case of receiving a call. In this task, you are going to draw generalizations indicating that **Supplementary Service** is derived from the **Place Call** use case and the **Receive Call** use case. Use the figure in this section as a reference.

To draw generalizations, follow these steps:

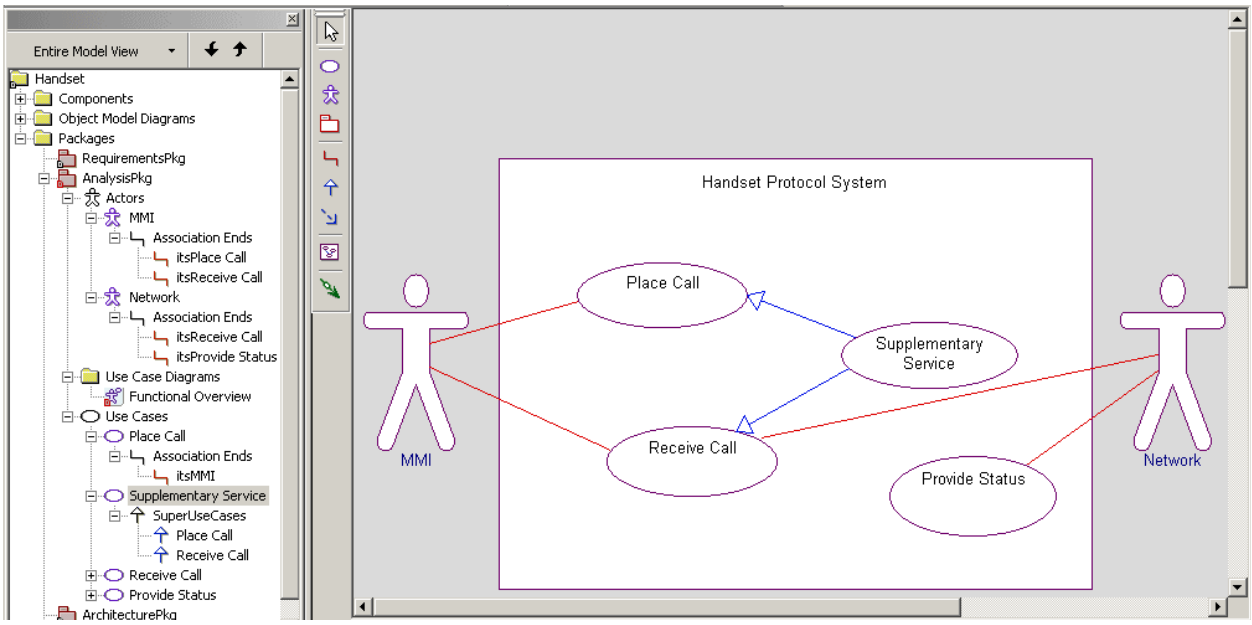
1. Click the Create Generalization button on the **Drawing** toolbar, and then click the **Supplementary Service** use case and draw a line to the **Place Call** use case.

2. Click the Create Generalization button, and then click the **Supplementary Service** use case and draw a line to the **Receive Call** use case.

3. In the browser, expand the **Supplementary Service** use case. Notice that **Place Call** and **Receive Call** are **SuperUseCases** for this use case.

   Note: To quickly find the **Supplementary Service** use case in the Rational Rhapsody browser, right-click it on the Functional Overview use case diagram and click **Locate** or press **Ctrl+L**.

Your Rational Rhapsody browser should resemble the following figure:



Task 1g: Adding Remarks to Model Elements and Diagrams

In this task, you are going to add a comment to the Functional Overview use case diagram. You can add remarks to specify additional information about a model element or diagram. Rational Rhapsody supports the following types of remarks in diagrams, which can be accessed from the **Common Drawing** toolbar:

- **Note** is a textual annotation that contains information that might be useful to the reader, but it does not add semantics. A note is not stored in the model repository and is not visible in the Rational Rhapsody browser.
- **Constraint** is a condition or restriction expressed in text. Constraints might have semantics in terms of the application, but the Rational Rhapsody product does not do anything with them nor does it enforce those semantics. Constraints are part of the model and are, therefore, visible in the browser.

  Note: Most constraints are declarative and not imperative, and therefore do not affect code generation. For example, if you add a constraint that says the worst case execution time of an operation is `< 12ms`, it does not change how code is generated (which it would if it were imperative), but it does change whether or not the generated code is correct.

- **Comment** is a textual annotation that contains information that might be useful to the reader, but it does not add semantics. Comments are visible in the browser.
- **Requirement** is a textual annotation that describes the intent of the element. Requirements might have semantics in terms of the application, but the Rational Rhapsody product does not do anything with them nor does it enforce those semantics. Requirements are part of the model and are, therefore, visible in the browser.
- **Anchor** attaches a constraint, comment, requirement, or note to one or more elements.

To add remarks to model elements and diagrams, follow these steps:

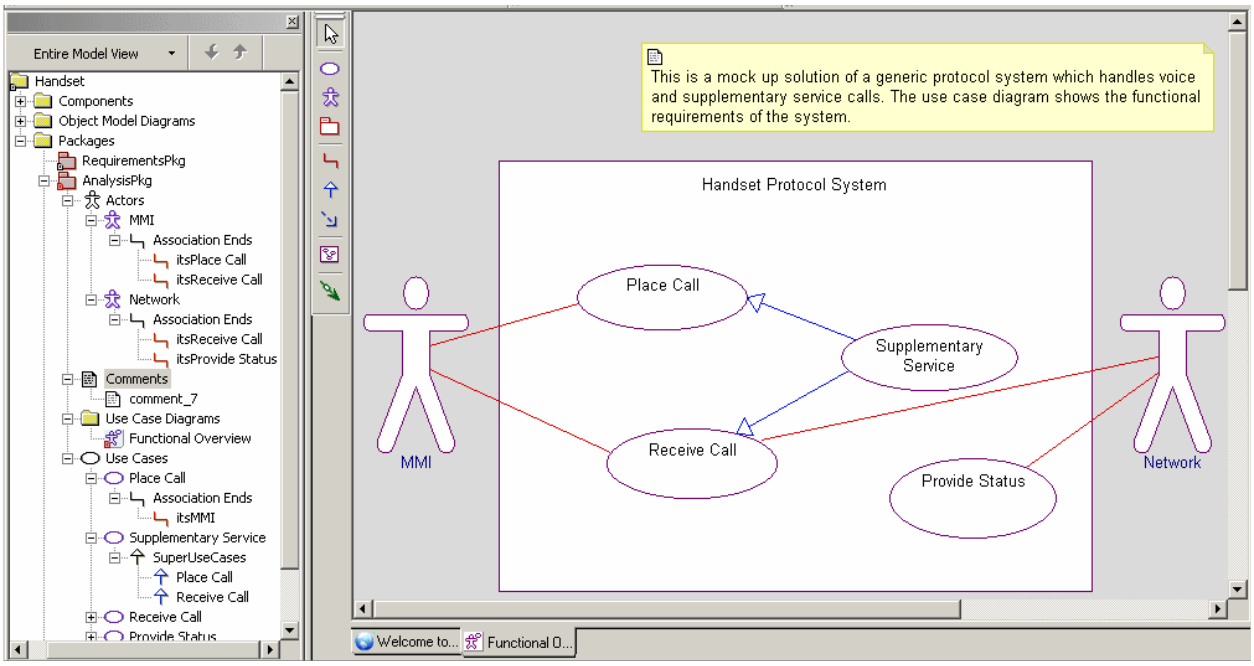
1. Click the Comment button on the **Common Drawing** toolbar.

   Note: If the toolbar is not open, select **View > Toolbars > Common Drawing**.

2. Click the top section of the diagram (outside of the boundary box).
3. Type the following description:

   ```
   This is a mock up solution of a generic protocol system which handles
   voice and supplementary service calls. The use case diagram shows the
   functional requirements of the system.
   ```

Rational Rhapsody adds the comment to the **Comments** category in the **AnalysisPkg** package, as shown in the following figure:



4. Click the Save button to save your model.

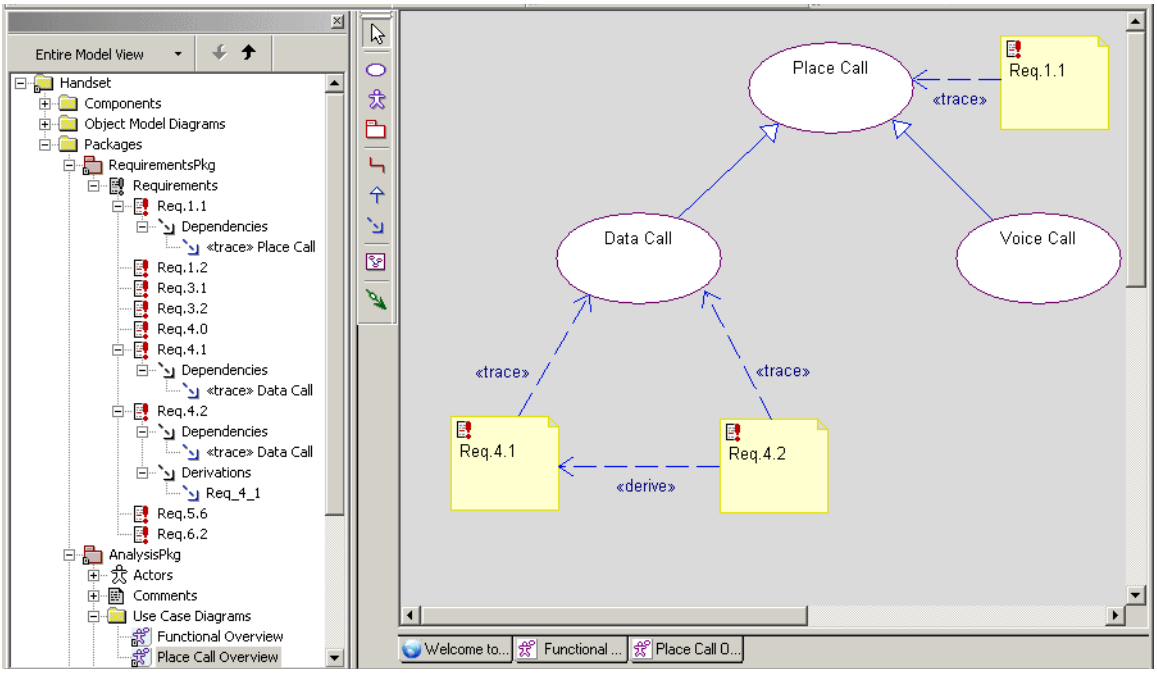
You have completed drawing the Functional Overview use case diagram. It should resemble the figure shown above.

Exercise 2: Creating the Place Call Overview UCD

The Place Call Overview use case diagram breaks down the **Place Call** use case and identifies the different types of calls that can be placed as use cases.

The following figure shows the Place Call Overview use case diagram that you are going to create in this exercise.

Place Call Overview Use Case Diagram



Task 2a: Creating the Place Call Overview Use Case Diagram

To create the Place Call Overview use case diagram, follow these steps:

1. In the browser, in the **AnalysisPkg** package, right-click the **Use Case Diagrams** category and select **Add New Use Case Diagram**. The New Diagram dialog box opens.
2. Type `Place Call Overview` and then click **OK**.

Rational Rhapsody automatically adds the name of the new use case diagram to the browser and opens the new diagram in the drawing area.

Task 2b: Drawing the Use Cases

In this task, you are going to draw the following use cases:

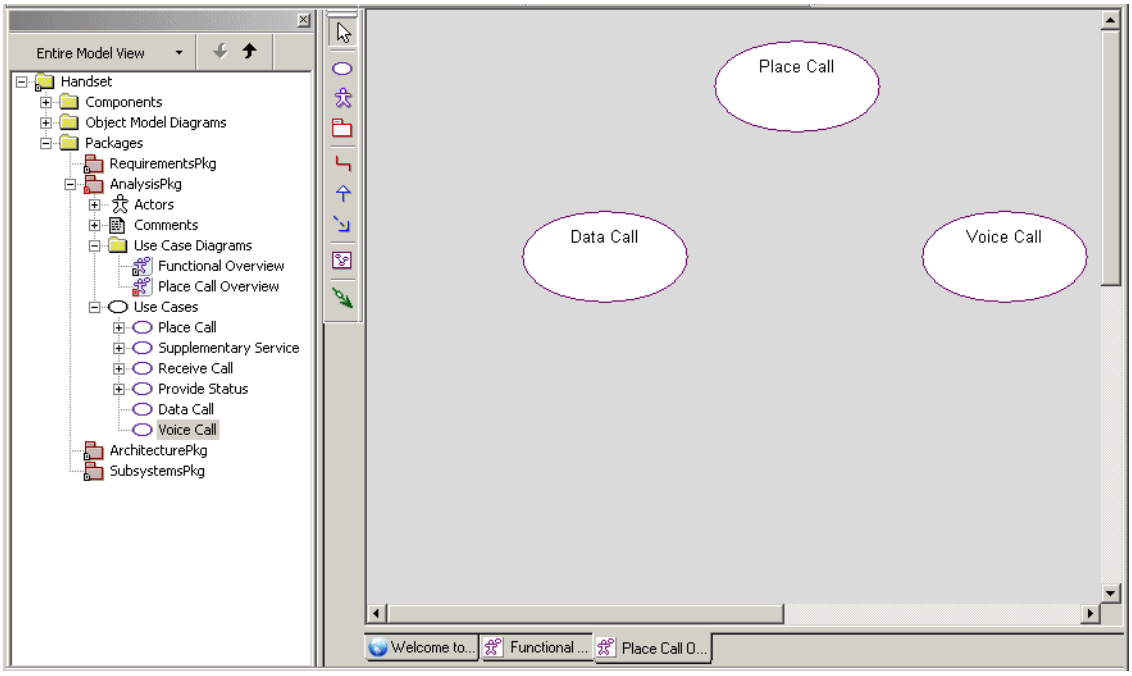
- **Place Call** to show that the user can place various types of calls. You defined the **Place Call** use case in the Functional Overview use case diagram.
- **Data Call** to show that the user can originate and receive data requests. It is a more specific case of placing a call.
- **Voice Call** to show that the user can place and receive voice calls, either while transmitting or receiving data, or standalone. It is a more specific case of placing a call.

Use the **Place Call Overview Use Case Diagram** figure as a reference.

To draw the use cases, follow these steps:

1. Continuing from the previous task, in the Rational Rhapsody browser, if it is not already, expand the **Use Cases** category.
2. Select the **Place Call** use case and drag it to the top center of the drawing area for the Place Call Overview use case diagram.
3. Click the Create Use Case button on the **Drawing** toolbar.
4. Create a use case in the lower left of the drawing area, named `Data Call`, and press **Enter**.

5. Create a use case in the lower right of the drawing area, named `Voice Call`, and press **Enter**. Your browser and diagram should resemble the following figure:



Task 2c: Defining Use Case Features

To add descriptions to the **Data Call** and **Voice Call** use cases, follow these steps:

1. In the Place Call Overview use case diagram or the browser, double-click the **Data Call** use case, or right-click and select **Features**. The Features dialog box opens.

2. On the **Description** tab, type the following text to describe its purpose:

   ```
   The system must be able to originate and receive data requests of up to
   384 kbps. Data calls can be originated or terminated while active voice
   calls are in progress.
   ```

   Note: You can also click the Ellipsis button just above the **Description** box on the **Description** tab to expand the internal text editor. When you have entered the description, click **OK** to close the Text Editor dialog box and return to the Features dialog box.

3. Click **Apply**.

4. With the Features dialog box still opened (if necessary, see **Keeping Open the Features Dialog Box**), for the **Voice Call** use case, type the following description:

   ```
   The user must be able to place or receive voice calls, either while
   transmitting or receiving data, or standalone. The limit of the voice
   calls a user can engage in at once is dictated by the conference call
   supplementary service.
   ```

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

6. Click the Save button to save your model.

Task 2d: Drawing Generalizations

In this task, you are going to draw generalizations to show that the **Data Call** use case and the **Voice Call** use case derive from the **Place Call** use case. Use the **Place Call Overview Use Case Diagram** figure as a reference.

To draw generalizations, follow these steps:

1. Click the Create Generalization button on the **Drawing** toolbar to activate the tool.
2. Click the edge of the **Data Call** use case and draw the line to the edge of the **Place Call** use case.
3. Click the Create Generalization button .
4. Click the edge of the **Voice Call** use case and draw the line to the edge of the **Place Call** use case.

Task 2e: Modeling Requirements in Rational Rhapsody

Modeling requirement elements in Rational Rhapsody enables you to provide requirements traceability without a Requirements Management (RM) tool. Modeling requirement elements also supplements the Rational Rhapsody to DOORs interface.

Requirements traceability is the ability to describe and follow the life of a requirement, in both a forward and backward direction. It supports requirements verification and validation, prevents the introduction of unspecified features, and provides visibility to derived requirements that need to be specified and tested.

For more information on the Rational Rhapsody interface to DOORS, refer to the *IBM Rational IBM Rational Rhapsody User Guide*.

Adding Requirement Elements to the Model

You can represent requirements in the browser and diagrams as requirement elements. Requirement elements are textual annotations, which describe the intent of the element.

In this task, you are going to add the handset model requirements to the **RequirementsPkg** package in the browser. You can also add requirements directly to the diagram using the Requirement tool from the **Common Drawing** toolbar. Refer to the *IBM Rational Rhapsody User Guide* for more information.

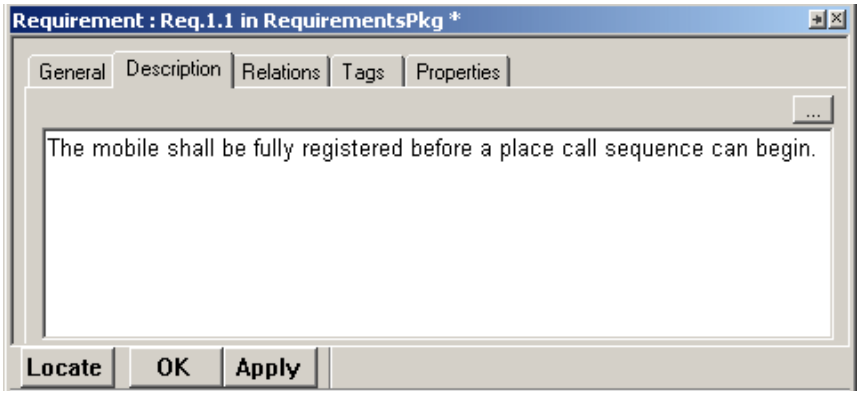
Use the **Place Call Overview Use Case Diagram** figure as a reference.

To add requirements elements, follow these steps:

1. In the Rational Rhapsody browser, right-click the **RequirementsPkg** package, and select **Add New > Requirement**. Rational Rhapsody creates the **Requirements** category and a requirement with a default name of **requirement_*n***, where ***n*** is greater than or equal to 0.
2. Rename the requirement `Req.1.1` and then press **Enter**.
3. Double-click **Req.1.1** or right-click and select **Features**. The Features dialog box opens.
4. On the **Description** tab, type the following:

```
The mobile shall be fully registered before a place call sequence can
begin.
```

Your Features dialog box should resemble the following figure:



5. Right-click the **Requirements** category and select **Add New Requirement** for each of the remaining requirements and their specifications as follows:

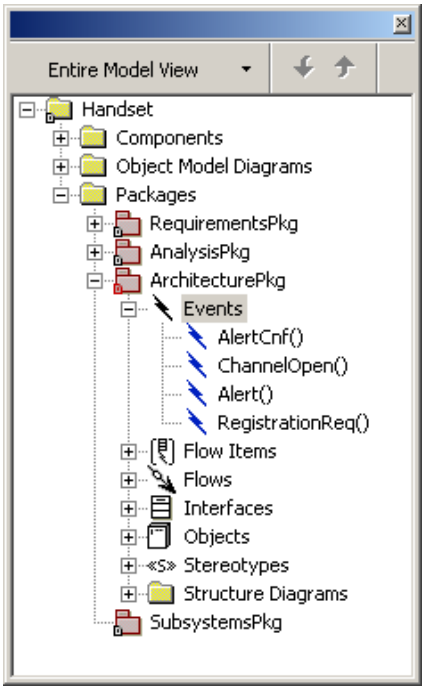
   - **Req.1.2** – `The mobile shall have a signal strength within +/- 1 of the minimum acceptable signal.`
   - **Req.3.1** – `The mobile shall be able to place short messages while registered.`
   - **Req.3.2** – `The mobile shall be able to receive short messages while registered.`
   - **Req.4.0** – `The mobile shall be able to receive data calls at the rate of 128 kbps.`
   - **Req.4.1** – `The mobile shall be able to send data at the rate of 384 kbps.`
   - **Req.4.2** – `The mobile shall be able to receive streaming video at 384 kbps.`
   - **Req.5.6** – `The mobile shall be able to receive a maximum of 356 characters in a short message.`
   - **Req.6.2** – `The optimal size of messages the mobile can send in a text message is 356 characters.`

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

7. Click the Save button to save your model. Your diagram and browser should resemble the following figure:

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

    Rational Rhapsody adds the provided and required interfaces to the **mm_dl** port in the Object Diagram. Rational Rhapsody also adds the receptions to the **Events** category in **ArchitecturePkg** package, as shown in the following figure:



16. To specify the port interfaces for **dl_in**, double-click the **dl_in** port, or right-click and select **Features**. The Features dialog box opens.

17. On the **General** tab, from the **Contract** drop-down list box, select **In**.

18. Select the **Contract** tab. Notice that Rational Rhapsody automatically added the provided interfaces previously defined as **In**.

19. Select the **Required** folder icon click the **Add** button. The Add New Interface dialog box opens.

20. From the **Interface** drop-down list box, select **Out** click **OK**. Rational Rhapsody automatically adds the required interfaces previously defined as **Out**.

21. Click **OK** to apply the changes and close the Features dialog box.

Reversing a Port

In this task, you are going to reverse a port.You can reverse ports so that the provided interfaces become the required interfaces, and the required interfaces become the provided interfaces.

To reverse the **dl_in** port, follow these steps:

1. Open the Features dialog box for the **dl_in** port.
2. On the **General** tab, in the **Attributes** group, select the **Reversed** check box.
3. Click **OK**.
4. Save your model.

You have completed drawing the Handset System structure diagram, which should resemble the **Handset System Structure Diagram** figure.

Task 1g: Allocating the Functions Among Subsystems

Now that you have captured the architectural design in the Object Diagram, you need to divide the operations of the system into its functional subsystems and allocate the activities among the subsystems.

Note

For ease of presentation, this section includes both the system and subsystem structure diagrams. Depending on your workflow, you might perform further black-box analysis with activity diagrams, sequence diagrams, and statecharts, and white-box analysis using sequence diagrams before decomposing the system's functions into subsystem components.

Organizing the SubsystemsPkg Package

Packages let you divide the system into functional domains or subsystems, which consist of objects, object types, functions, variables, and other logical artifacts. They can be organized into hierarchies to provide a high level of partitioning.

In this task, you are going to create the following subpackages, which represent the functional subsystems:

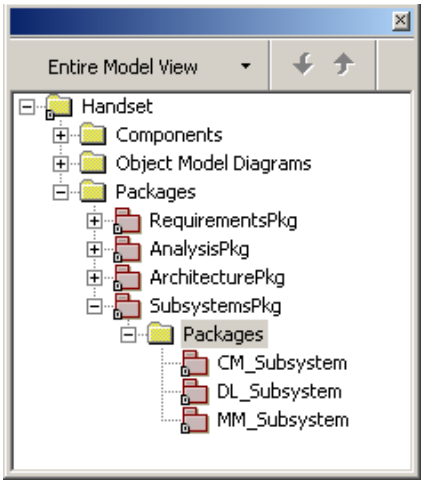
- **CM_Subsystem** for **ConnectionManagement**
- **DL_Subsystem** for **DataLink**
- **MM_Subsystem** for **MobilityManagement**.

To create packages within the **Subsystems** package, follow these steps:

1. In the browser, right-click **SubsystemsPkg** and select **Add New > Package**. Rational Rhapsody creates a new **Packages** category within **SubsystemsPkg** and a package with the default name **package_*n***, where ***n*** is greater or equal to 0.

2. Rename the package `CM_Subsystem` press **Enter**.

3. Right-click Packages (under SubsystemsPkg), select **Add New Package**, and create two additional packages named `DL_Subsystem` and `MM_Subsystem`, as shown below:



Organizing Elements

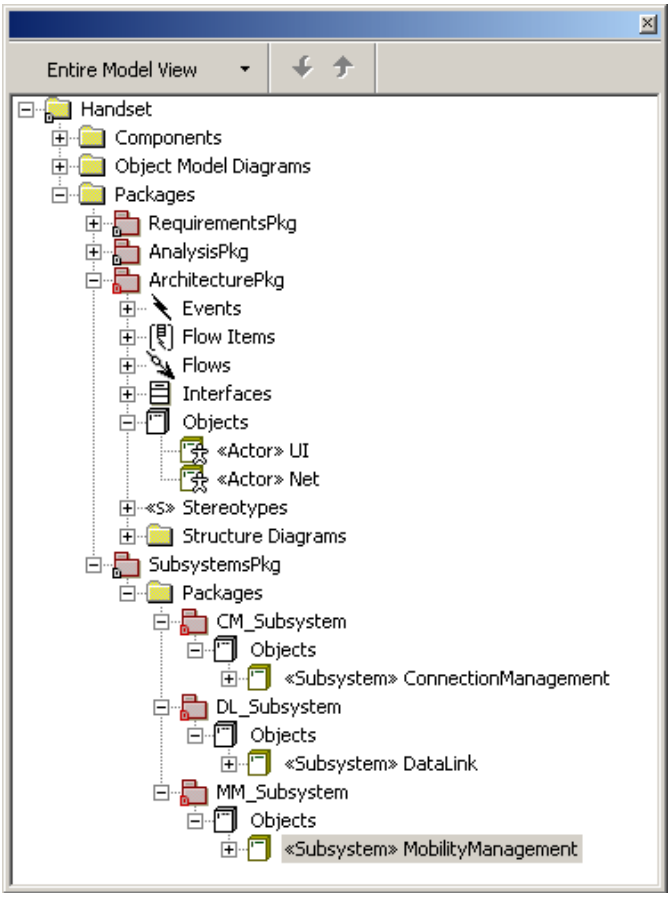
In this task, you are going to allocate the subsystem objects from the Object Diagram in the **ArchitecturePkg** package to their respective packages in the **SubsystemsPkg** package by moving them.

To organize elements, follow these steps:

1. In the browser, expand the **ArchitecturePkg** package and the **Objects** category.
2. Select the **<<Subsystem>> ConnectionManagement** object and drag it into the **CM_Subsystem** package.
3. Select the **<<Subsystem>> DataLink** object and drag it into the **DL_Subsystem** package.

4. Select the **<<Subsystem>> MobilityManagement** object and drag it into the **MM_Subsystem** package.

   The objects are removed from the **ArchitecturePkg** package and added to the **SubsystemsPkg** packages, as shown in the following figure:



5. Save your model.

You can decompose the system-level objects in the Handset System structure diagram into sub-objects and corresponding structure diagrams to show their decomposition. In the later sections, you are going to create the following subsystem structure diagrams:

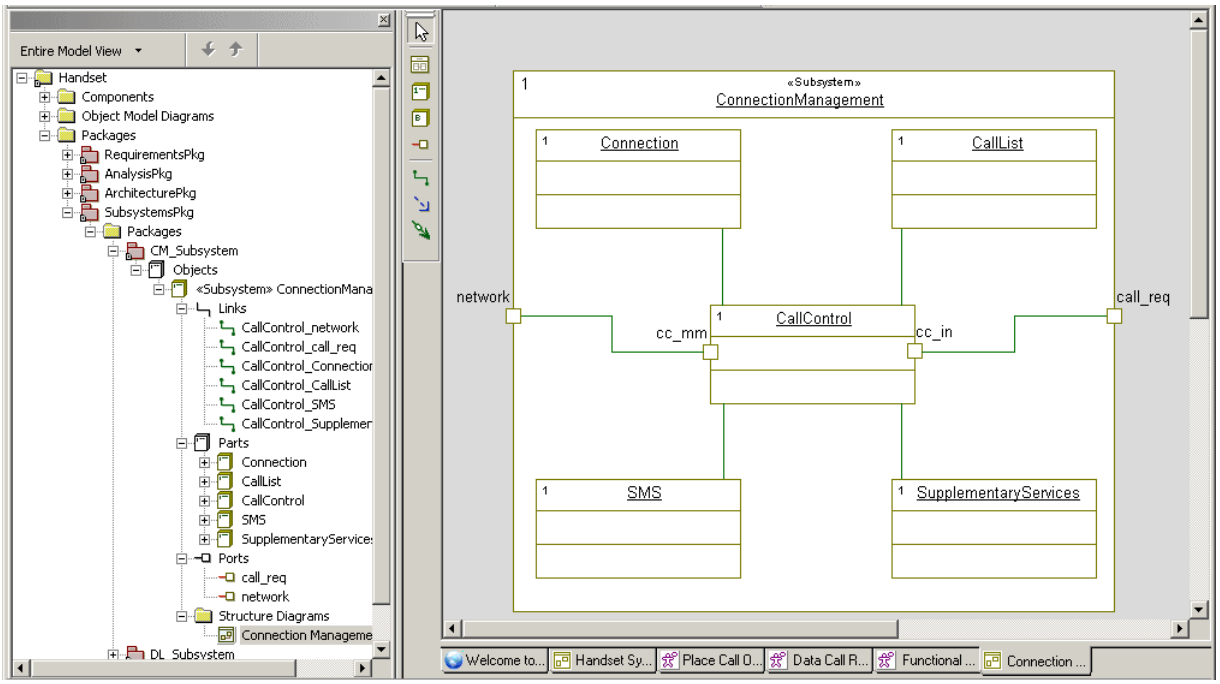
- Connection Management from the **ConnectionManagement** object
- Data Link from the **DataLink** object
- MM Architecture from the **Mobility Management** object

Exercise 2: Creating the Connection Management Structure Diagram

The Connection Management structure diagram decomposes the **ConnectionManagement** object into its subsystems. Connection Management identifies how calls are set up, including the establishment and clearing of calls, short message services, and supplementary services.

The following figure shows the Connection Management structure diagram that you are going to create in this exercise.

Connection Management Structure Diagram



Task 2a: Creating the Connection Management Structure Diagram

To create the Connection Management structure diagram, follow these steps:

1. In the browser, if not already expanded, expand the **Packages** category, the **SubsystemsPkg** package, the **Packages** package, the **CM_Subsystem** package and the **Objects** category. Right-click **<<Subsystem>> ConnectionManagement** and select **Add New > Structure Diagram**. The New Diagram dialog box opens.

   or

   In the Handset System structure diagram, right-click **ConnectionManagement** and select **New Structure Diagram**. The New Diagram dialog box opens.

2. Type `Connection Management Structure` click **OK**.

Rational Rhapsody automatically creates the **Structure Diagrams** category in the **CM_Subsystem** object, and adds the name of the new structure diagram. In addition, Rational Rhapsody opens the new diagram in the drawing area

Task 2b: Drawing Objects

In this task, you are going to draw the following objects, which represent the activities performed by Connection Management:

- **Connection** to track the number of valid connections
- **CallList** to maintain the list of currently active calls
- **CallControl** to manage incoming and outgoing calls
- **SMS** to manage the short message services
- **SupplementaryServices** to manage the supplementary services, including call waiting, holding, and barring

To draw objects, follow these steps:

1. Click the Object button on the **Drawing** toolbar click or click-and-drag in the upper, left corner of **ConnectionManagement**. Rational Rhapsody creates an object with a default name of **object_*n***, where ***n*** is equal to or greater than 0.

2. Rename the object `Connection` press **Enter**.

3. Draw the **CallList**, **CallControl**, **SMS**, and **SupplementaryServices** objects using the **Connection Management Structure Diagram** figure as a reference.

Task 2c: Drawing Ports

In this task, you are going to draw ports using the **Connection Management Structure Diagram** figure as a reference.

To draw ports, follow these steps:

1. Click the Create Port button on the **Drawing** toolbar click the left edge of the **CallControl** object.
2. Type `cc_mm` press **Enter**. This port relays messages to and from **MobilityManagement**.
3. Click the Create Port button on the **Drawing** toolbar click the right edge of the **CallControl** object.
4. Type `cc_in` press **Enter**. This port relays messages from the user interface.

Changing the Placement of Ports

When Rational Rhapsody adds the **ConnectionManagement** object to the diagram, it places the ports defined in the Handset System structure diagram (created in exercise 1 of this lesson) on the boundary. You can change the port placement by selecting the port and dragging it to another location on the object.

To change port placement, follow these steps:

1. Typically, the ports from the **ConnectionManagement** object are not visible. To make them visible, right-click the object and select **Ports > Show All Ports**.
2. Use the **Connection Management Structure Diagram** figure as a reference and change the placement of the **call_req** and **network** ports, if necessary.
3. Save your model.

Task 2d: Drawing Links

In this task, you are going to draw links between objects and ports. A *link* is an instance of an association. You can specify links without having to specify the association being instantiated by the link; you can specify features of links that are not mapped to an association. Use the **Connection Management Structure Diagram** figure as a reference.

To draw links, follow these steps:

1. Click the Link button on the **Drawing** toolbar click the **cc_mm** port click the **network** port, and then click the mouse button again (this is the same as pressing **Enter**).

2. Click the Link button click the **cc_in** port click the **call_req** port, and then click the mouse button again or press **Enter**.

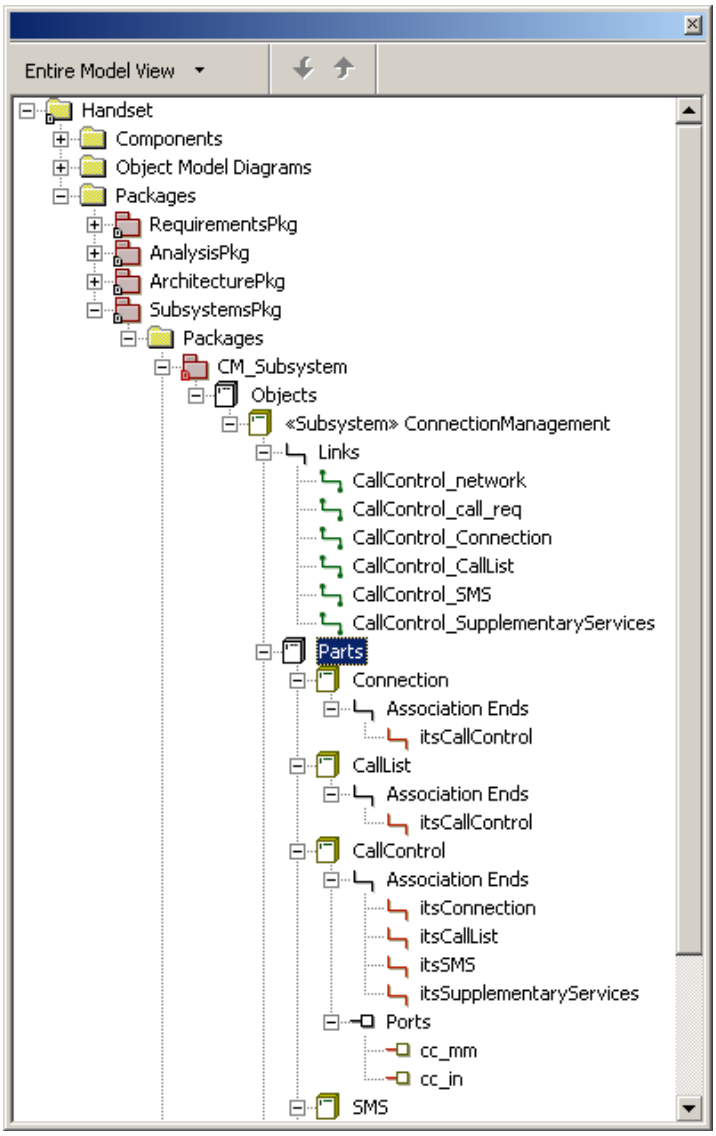
3. Click the Link button click the **CallControl** object click the **Connection** object, and then click the mouse button again or press **Enter**.

4. Click the Link button click the **CallControl** object click the **CallList** object, and then click the mouse button again or press **Enter**.

5. Click the Link button click the **CallControl** object click the **SMS** object, and then click the mouse button again or press **Enter**.

6. Click the Link button click the **CallControl** object click the **SupplementaryServices** object, and then click the mouse button again or press **Enter**.

7. Save your model. In the browser, expand the **<<Subsystem>> ConnectionManagement** category to view the newly created links and parts, as shown in the following figure:



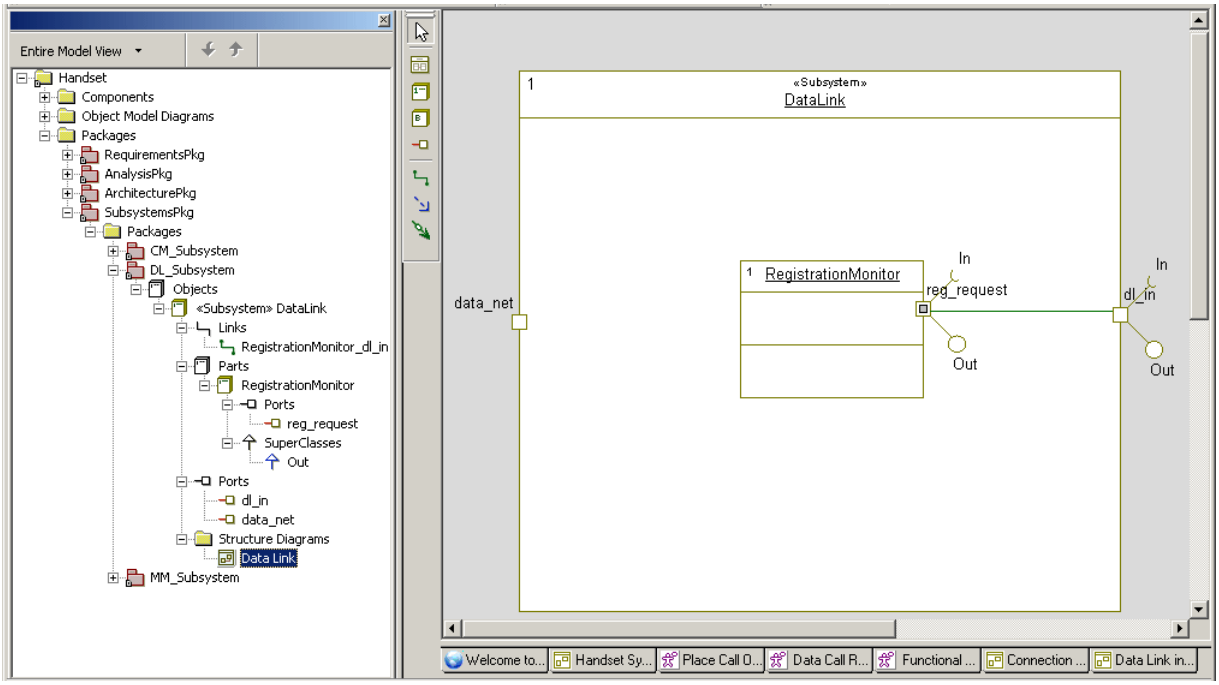
You have completed drawing the Connection Management diagram. It should resemble the **Connection Management Structure Diagram** figure.

Exercise 3: Creating the Data Link Structure Diagram

The Data Link structure diagram decomposes the **DataLink** object into its subsystems. It identifies how the system monitors registration.

The following figure shows the Data Link structure diagram that you are going to create in this exercise.

Data Link Structure Diagram



Task 3a: Creating the Data Link Structure Diagram

To create the Data Link diagram, follow these steps:

1. In the browser, if not already expanded, expand the **Packages** category, the **SubsystemsPkg** package, the **Packages** package, the **DL_Subsystem** package, and the **Objects** category. Right-click **<<Subsystem>> DataLink** and select **Add New > Structure Diagram**. The New Diagram dialog box opens.

   or

   Right-click **DataLink** in the Handset System structure diagram and select **New Structure Diagram**. The New Diagram dialog box opens.

2. Type `Data Link` click **OK**.

Rational Rhapsody automatically creates the **Structure Diagrams** category in the **<<Subsystem>> DL_Subsystem** object, and adds the name of the new structure diagram. In addition, Rational Rhapsody opens the new diagram in the drawing area.

Task 3b: Drawing Objects

In this task, you are going to draw the **RegistrationMonitor** object, which represents the activity performed by the **DataLink** object. Use the **Data Link Structure Diagram** figure as a reference.

To draw the **RegistrationMonitor** object, follow these steps:

1. Click the Object button on the **Drawing** toolbar click or click-and-drag in the center of **DataLink**.

2. Type `RegistrationMonitor` press **Enter**.

Task 3c: Drawing Ports

In this task, you are going to draw ports using the **Data Link Structure Diagram** figure as a reference.

To draw ports, follow these steps:

1. Click the Create Port button on the **Drawing** toolbar.
2. Click the right edge of **RegistrationMonitor** and create a port named `reg_request` press **Enter**. This port relays registration requests and results.
3. If the ports for the **DataLink** object are not visible, right-click the object and select **Ports > Show All Ports**.
4. If the ports for the **DataLink** object are not located as shown on the **Data Link Structure Diagram** figure, change these port placements by selecting a port and dragging it to another location on the object. For example, you might have to drag **dl_in** to the right edge of the **DataLink** object. This makes the Data Link structure diagram easier to work with.

Task 3d: Drawing Links

To draw links, follow these steps:

1. Click the Link button on the **Drawing** toolbar.
2. Click the **reg_request** port click the **dl_in** port click the mouse button again or press **Enter**.

Task 3e: Specifying the Port Contract and Attributes

In this task you are going to specify the port contract and features for **reg_request** using the **Data Link Structure Diagram** figure as a reference.

To specify the port contract and attributes, follow these steps:

1. Double-click the **reg_request** port, or right-click and select **Features**. The Features dialog box opens.
2. On the **General** tab, in the **Attributes** group, select the **Behavior** and **Reversed** check boxes.
3. On the **Contract** tab, select the **Provided** folder icon click the **Add** button. The Add New Interface dialog box opens.
4. From the **Interface** drop-down list box, select **In** click **OK**.

   Rational Rhapsody automatically adds the **Provided** and **Required** interfaces.
5. On the **General** tab, from the **Contract** drop-down list box, select **In**.
6. Click **Apply**. Rational Rhapsody displays a message that the port is not realized. Click **Yes** to add the realization.
7. Click **OK** to close the Features dialog box.
8. Save your model.

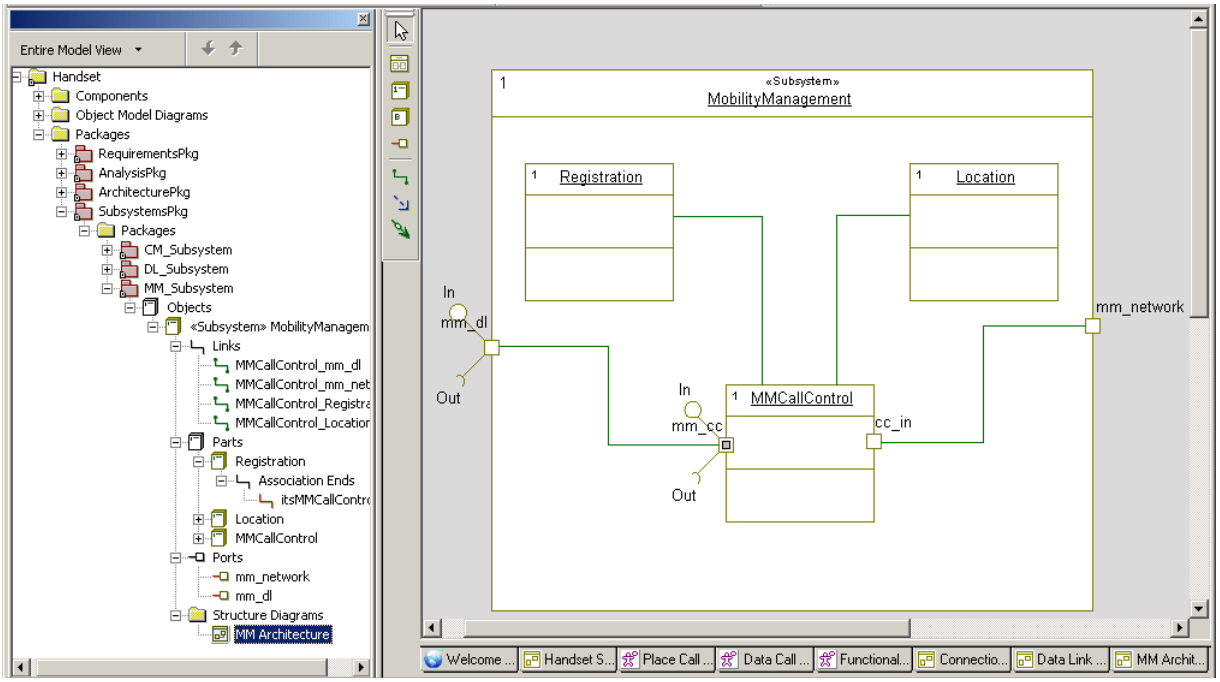
You have completed drawing the Data Link structure diagram. It should resemble the **Data Link Structure Diagram** figure. Rational Rhapsody automatically adds the newly created objects, links, and ports to the **DataLink** object in the browser.

Exercise 4: Creating the MM Architecture Structure Diagram

The MM Architecture diagram decomposes the **MobilityManagement** object into its subsystems. Mobility Management supports the mobility of users, including registering users on the network and providing their current location.

The following figure shows the MM Architecture diagram that you are going to create in this exercise.

MM Architecture Structure Diagram



Task 4a: Creating the MM Architecture Diagram

To create the MM Architecture diagram, follow these steps:

1. In the browser, if not already expanded, expand the **Packages** category, the **SubsystemsPkg** package, the **Packages** package, the MM**_Subsystem** package, and the **Objects** category. Right-click **<<Subsystem>> MobilityManagement** and select **Add New > Structure Diagram**. The New Diagram dialog box opens.

   or

   Right-click **MobilityManagement** in the Handset System structure diagram, and select **New Structure Diagram**. The New Diagram dialog box opens.

2. Type `MM Architecture` click **OK**.

Rational Rhapsody automatically creates the **Structure Diagrams** category in the **MM_Subsystem** object, and adds the name of the new structure diagram. In addition, Rational Rhapsody opens the new diagram in the drawing area.

Task 4b: Drawing Objects

In this task, you are going to draw the following objects, which represent the activities performed by MobilityManagement:

- **Registration** to maintain the registration status
- **Location** to track the location of users
- **MMCallControl** to maintain the logic for **MobilityManagement**

To draw objects, follow these steps:

1. Click the Object button on the **Drawing** toolbar click (or click-and-drag) in the upper, left corner of **MobilityManagement**.
2. Type `Registration`, and then press **Enter**.
3. Draw two objects and name them `Location` and `MMCallControl`. Use the **MM Architecture Structure Diagram** figure as a reference.

Task 4c: Drawing Ports

In this task, you are going to draw ports using the **MM Architecture Structure Diagram** figure as a reference.

To draw ports, follow these steps:

1. Click the Create Port button on the **Drawing** toolbar click the left edge of the **MMCallControl** object and name the port `mm_cc` press **Enter**. This port relays information to **ConnectionManagement**.

2. Click the Create Port button on the **Drawing** toolbar click the right edge of the **MMCallControl** object and name the port `cc_in` press **Enter**. This port sends and receives information from the **DataLink**.

3. If the ports for the **MobilityManagement** object are not visible, right-click the object and select **Ports > Show All Ports**.

4. If the ports for the **MobilityManagement** object are not located as shown on the **MM Architecture Structure Diagram** figure, change a port placement by selecting it and dragging it to another location on the object. This makes the MM Architecture structure diagram easier to work with.

Task 4d: Drawing Links

In this task, you are going to draw links using the **MM Architecture Structure Diagram** figure as a reference.

To draw links, follow these steps:

1. Click the Link button on the **Drawing** toolbar click the **mm_cc** port click the **mm_dl** port, and then click the mouse button again (this is the same as pressing **Enter**).

2. Click the Link button click the **cc_in** port click the **mm_network** port, and then click the mouse button again or press **Enter**.

3. Click the Link button click the **MMCallControl** object click the **Registration** object, and then click the mouse button again or press **Enter**.

4. Click the Link button click the **MMCallControl** object click the **Location** object, and then click the mouse button again or press **Enter**.

Task 4e: Specifying the Port Contract and Attributes

In this task, you are going to specify the port contract and attributes for the **mm_cc** port.

To specify the port contract and attributes, follow these steps:

1. Double-click the **mm_cc** port, or right-click and select **Features**. The Features dialog box opens.
2. On the **General** tab, specify the following settings:
   a. From the **Contract** drop-down list box, select **In**.
   b. In the **Attributes** group, select the **Behavior** check box.
3. Click **Apply**. Rational Rhapsody displays a message that the port is not realized. Click **Yes** to add the realization.

   Rational Rhapsody automatically adds the provided and required interfaces to the **Contract** tab.
4. Click **OK** to close the dialog box.
5. Save your model.

You have completed drawing the MM Architecture diagram. It should resemble the **MM Architecture Structure Diagram** figure. Rational Rhapsody automatically adds the newly created objects, links, and ports to the **MobilityManagement** object in the browser.

Summary

In this lesson, you created a system-level structure diagram, and then decomposed that diagram into functions to show how the software systems trace to the system functional objects. You became familiar with the parts of a structure diagram and created the following:

- Objects
- Ports
- Flows
- Links
- Dependencies

As mentioned earlier, for ease of presentation, this section included both the system and subsystem structure diagrams. Depending on your workflow, you might identify the communication scenarios using sequence diagrams (which are covered in **Lesson 5: Creating Sequence Diagrams**) before defining the flows, flowitems, and port contracts.

In addition, you might perform black-box analysis using activity diagrams (which are covered in **Lesson 6: Creating Activity Diagrams**), sequence diagrams, and statecharts (which are covered in **Lesson 7: Creating Statecharts**). You might perform white-box analysis using sequence diagrams before decomposing the system's functions into subsystem components.

You are now ready to proceed to the next lesson, where you are going to define how the system components are interconnected using object model diagrams.

Lesson 3: Creating Object Model Diagrams

Object model diagrams (OMDs) specify the structure of the classes, objects, and interfaces in the system and the static relationships that exist between them. Object model diagrams provide a graphical representation of the system structure. The Rational Rhapsody code generator directly translates the elements and relationships modeled in OMDs into C++ source code.

Goals for this Lesson

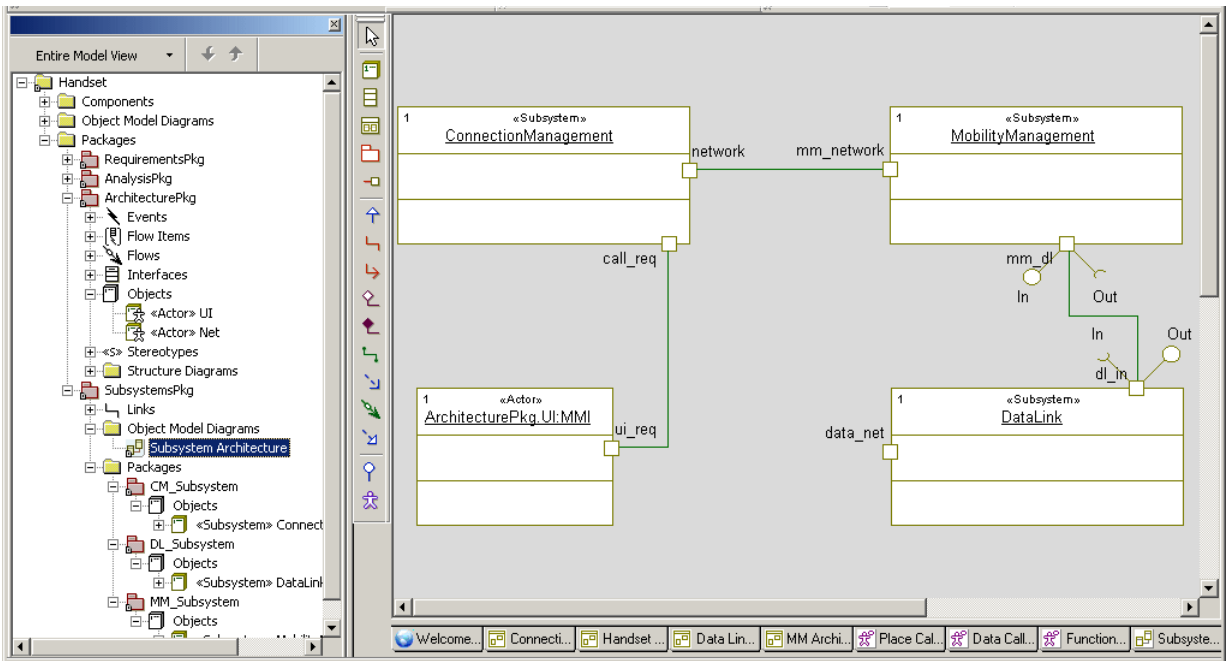
In this lesson, you are going to create the Subsystem Architecture object model diagram, which shows how the system components are interconnected at the subsystem level, and identifies the port connections and the flow of information between components as links.

Exercise 1: Creating the Subsystem Architecture OMD

Object model diagrams show the types of objects in the system, the attributes and operations that belong to those objects, and the static relationships that can exist between classes (types).

The following figure shows the Subsystem Architecture object model diagram that you are going to create in this exercise.

Subsystem Architecture Object Model Diagram



Task 1a: Creating the Subsystem Architecture Object Model Diagram

The Subsystem Architecture object model diagram identifies how the system components are interconnected at the subsystem level. It shows the realization of flows between objects through links and ports. Flows are used for high-level analysis, and links are used for executability (realization of flows).

You draw an object model diagram using the following general steps:

1. Draw objects.
2. Draw links.

The following tasks describe each of these steps in detail.

To create an object model diagram, follow these steps:

1. Start Rational Rhapsody and open the handset model if they are not already open.
2. In the browser, expand the **Packages** category, right-click the **SubsystemsPkg** package select **Add New > Object Model Diagram**. The New Diagram dialog box opens.
3. Type `Subsystem Architecture` click **OK**.

Rational Rhapsody adds the **Object Model Diagrams** category and the name of the new object model diagram to the browser. Rational Rhapsody also opens the new object model diagram in the drawing area.

Task 1b: Drawing Objects

In this task, you are going to draw objects for the Subsystem Architecture object model diagram. Use the **Subsystem Architecture Object Model Diagram** figure as a reference.

To draw objects for the object model diagram, follow these steps:

1. In the Rational Rhapsody browser, expand **Packages** package, the **CM_Subsystem** package, and the **Objects** category.
2. Click the **<<Subsystem>> ConnectionManagement** object and drag it to the upper, left side of the drawing area. The **ConnectionManagement** object and its ports are added to the diagram.
3. In the browser, expand the **MM_Subsystem** package and the **Objects** category.
4. Click the **<<Subsystem>> MobilityManagement** object and drag it to the upper, right side of the drawing area.
5. In the browser, expand the **DL_Subsystem** package and the **Objects** category.
6. Click the **<<Subsystem>> DataLink** object and drag it to the lower, right side of the drawing area.
7. To make the drawing a little easier to work with, make the following settings for all the objects in your diagram:
   a. Right-click an object in the drawing area (for example, **ConnectionManagement**) and select **Display Options**. The Display Options dialog box opens.
   b. On the **General** tab, in the **Display Name** group, select the **Name only** option button.
   c. Click **OK**.
8. Save your model.

Task 1c: Drawing More Objects

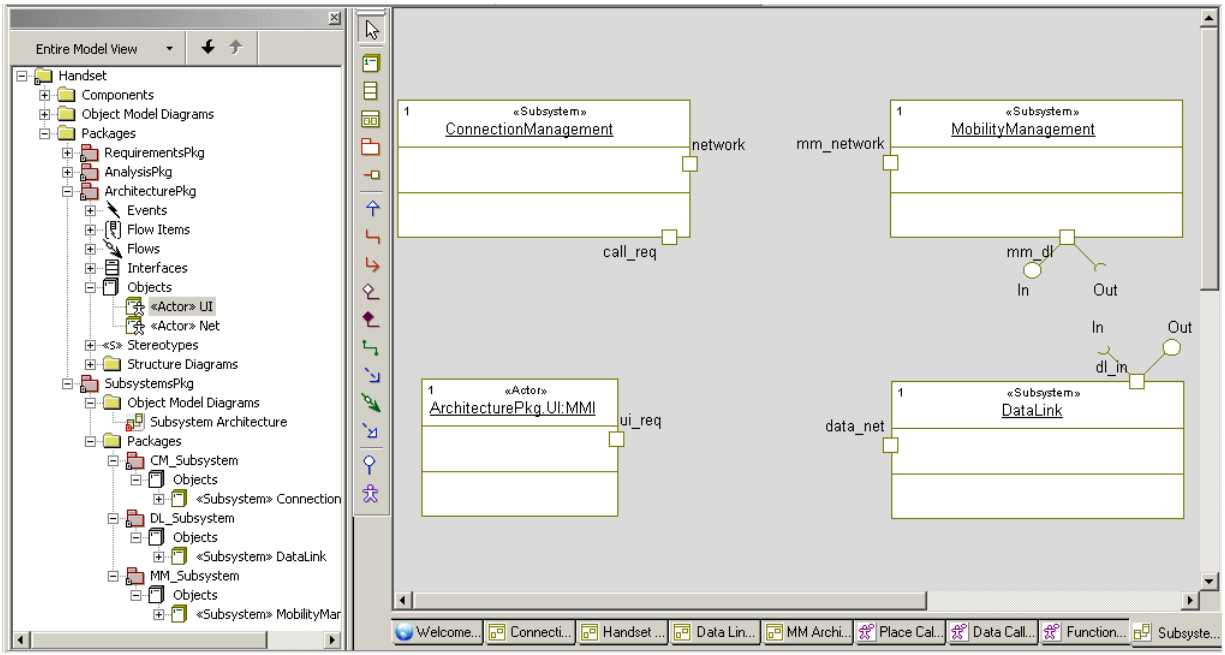
The Subsystem Architecture object model diagram contains the **UI** object defined in the Handset System structure diagram you created in **Exercise 1: Creating the Handset System Structure Diagrams** in the previous lesson. The **UI** object interacts with **ConnectionManagement** to establish and clear calls, and request and receive data services.

Use the **Subsystem Architecture Object Model Diagram** figure as a reference to do this task.

To draw the **UI** object in your object model diagram, follow these steps:

1. In the browser, expand the **ArchitecturePkg** package and the **Objects** category.
2. Click the **<<Actors>> UI** object and drag it to the bottom, left side of the object model diagram. The **UI** object and its port are added to the diagram.

Note: When Rational Rhapsody add an object to a diagram, it places the object's ports on the object's boundary. Using the **Subsystem Architecture Object Model Diagram** figure as a reference, change the placement of ports by selecting a port and dragging it to the location shown in the following figure. Doing this makes the diagram easier to work with.



Task 1d: Drawing Links

In this task, you are going to draw links that show the flow of information for the Subsystem Architecture object model diagram. A link is an instance of an association.

To draw a link, follow these steps:

1. Click the Link button on the **Drawing** toolbar.
2. Click the **network** port and then click the **mm_network** port click the mouse button again (this is the same as pressing **Enter**).
3. Draw the following links:
   - From the **call_req** port to the **ui_req** port
   - From the **mm_dl** port to the **dl_in** port
4. Save your model.

You have completed drawing the Subsystem Architecture diagram. It should resemble the **Subsystem Architecture Object Model Diagram** figure.

Summary

In this lesson, you created an object model diagram, which shows how the system components are interconnected. You became familiar with the parts of an object model diagram and added the following elements:

- Objects
- Links

In **Lesson 2: Creating Structure Diagrams**, you specified some ports and links between them on structure diagrams. In this lesson, you specified some ports and links at the overall system level.

You are now ready to proceed to the next lesson, where you are going to generate code and build your handset model in its current state. This lets you determine whether the model meets the requirements and identify defects early on in the design process.

Lesson 4: Generating Code and Building Your Model

It is good practice to test the model incrementally using model execution. You can animate pieces of the model as it is developed. This gives you the opportunity to determine whether the model meets the requirements and find defects early on. Then you can test the entire model. In this way, you iteratively build the model, and then with each iteration perform an entire model validation.

Goals for this Lesson

In this lesson, you are going to generate code for the handset model and build your model at its current point by doing the following:

- Create a component
- Set the component features
- Create a configuration
- Generate code in Rational Rhapsody
- Build your handset model

Exercise 1: Preparing for Generating Code

Before you generate code, you must do the following general steps:

1. Create a component and set its features.
2. Create a configuration.
3. Generate component code.
4. Build the component application.
5. Run the component application.

The following tasks describe these steps in detail.

Task 1a: Creating a Component

A *component* is a physical subsystem in the form of a library or executable program. It plays an important role in the modeling of large systems that contain several libraries and executables. Each component contains configuration and file specification categories, which are used to generate, build, and run the executable model.

Each project contains a default component, named `DefaultComponent`. You can use the default component or create a new component. In this task, you are going to rename the default component `Simulate`, and then use the **Simulate** component to animate the model.

To use the default component, follow these steps:

1. In the Rational Rhapsody browser, expand the **Components** category.
2. Double-click **DefaultComponent** to open the Features dialog box.
3. In the **Name** box, replace the name `DefaultComponent` with `Simulate`.
4. Click **Apply**. Do not close the dialog box.

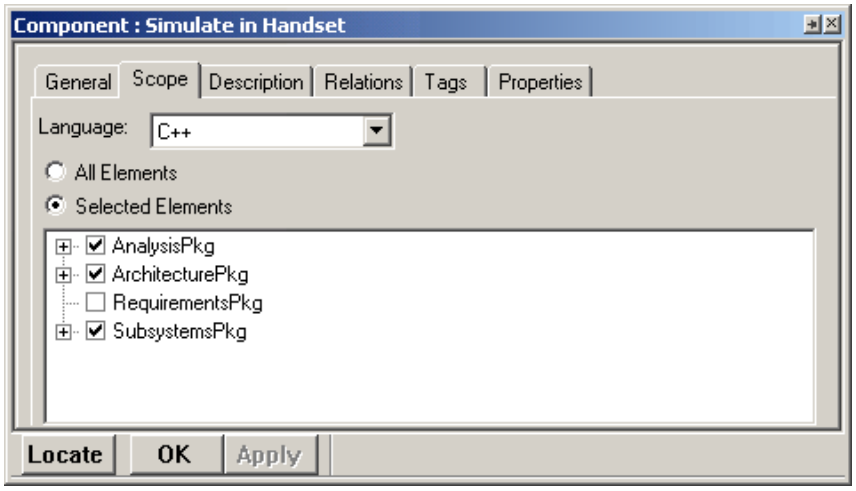
Task 1b: Setting the Component Features

Once you have created the component, you must set its features.

To set the component features, follow these steps:

1. If you closed the Features dialog box from the previous task, open it. In the Rational Rhapsody browser, double-click **Simulate** or right-click and select **Features**.
2. On the **General** tab, in the **Type** group, select the **Executable** option button if it is not already selected.
3. On the **Scope** tab:
   a. Select the **Selected Elements** option button.
   b. Select the **AnalysisPkg**, **ArchitecturePkg**, and **SubsystemsPkg** check boxes. These are the packages for which you are going to generate code. Do not select **RequirementsPkg** because you are not going to generate code for it.

      Your **Scope** tab should look like the following figure.



4. Click **OK**.

Task 1c: Creating a Configuration

A component can contain many configurations. A *configuration* specifies how the component is to be produced.

Each component contains a default configuration, named `DefaultConfig`. In this task, you are going to rename the default configuration to `Debug`, and then use the **Debug** configuration to animate the model.

To use the default configuration, follow these steps:

1. In the Rational Rhapsody browser, expand the **Simulate** component and the **Configurations** category.
2. Double-click **DefaultConfig** to open the Features dialog box.
3. In the **Name** box, replace `DefaultConfig` with `Debug`.
4. Click **OK**.

Task 1d: Generating Code

In this task you generate code in Rational Rhapsody. Before you generate code, you must first set the *active configuration*. The active configuration is the configuration for which you generate code. The active configuration appears in the drop-down list on the **Code** toolbar.

To set the active configuration and generate code for the `Debug` configuration, follow these steps:

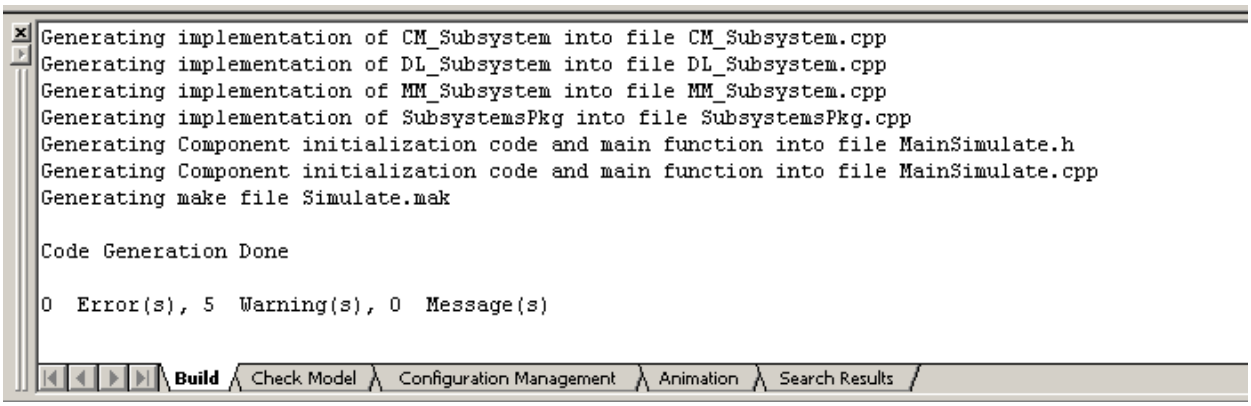
1. In the Rational Rhapsody browser, right-click the **Debug** configuration select **Set as Active Configuration**.

   Note: You can also select the active configuration from the drop-down list on the **Code** toolbar.

2. Select **Code > Generate > Debug**. Rational Rhapsody displays a message that the **Debug** directory does not yet exist and asks you to confirm its creation.

3. Click **Yes**. Rational Rhapsody places the files generated for the active configuration in the new `Debug` directory.

Rational Rhapsody generates the code and displays output messages in the **Build** tab of the Output window, as shown in the following figure:



Note

As you can see, when you generated code you received warnings; see **About Code Generation Warnings**.

The messages inform you of the code generation status, including:

- Success or failure of internal checks for the correctness and completeness of your model. These checks are performed before code generation begins.
- Names of files generated for classes and packages in the configuration.
- Names of files into which the **main()** function is generated.
- Location of the generated make file.
- Completion of code generation.

Fixing Code Generation Errors

If you receive code generation errors, double-click the error in the Output window to go to the source of the error. The source of the error appears as a highlighted element. Once you fix the problem, regenerate the code, and rebuild the application until there are no error messages.

About Code Generation Warnings

If you receive code generation warnings, double-click the warning in the Output window to go to the source of the warning. The source of the warning appears as a highlighted element. You might be able to fix the warning. Or you might leave the warning as is because your model is not yet fully formed.

In this case, the five warnings you received is because your model is not yet fully formed so that all your port connections are not yet in place. For now, you will ignore the warnings. They will go away as you continue to build the handset model.

In other cases, if you do have warnings that are valid for the current state of your mode, fix them regenerate the code, and rebuild the application until those warnings are no longer appearing.

Examining Generated Source Files

To examine any of the generated source files, go to the `\Simulate\Debug` subfolder of the handset project.

Using External Elements

The Rational Rhapsody product enables you to visualize frozen legacy code or edit external code as external elements. This external code is code that is developed and maintained outside of the Rational Rhapsody product. This code will not be regenerated by the Rational Rhapsody product, but will participate in the code generation and build process of Rational Rhapsody models that interact or interface with this external code. You can create external elements by reverse engineering the files or by modeling. Refer to the *IBM Rational Rhapsody User Guide* for more information on using external elements.

Task 1e: Building the Model

Once you generate code without any errors, you are ready to build the model.

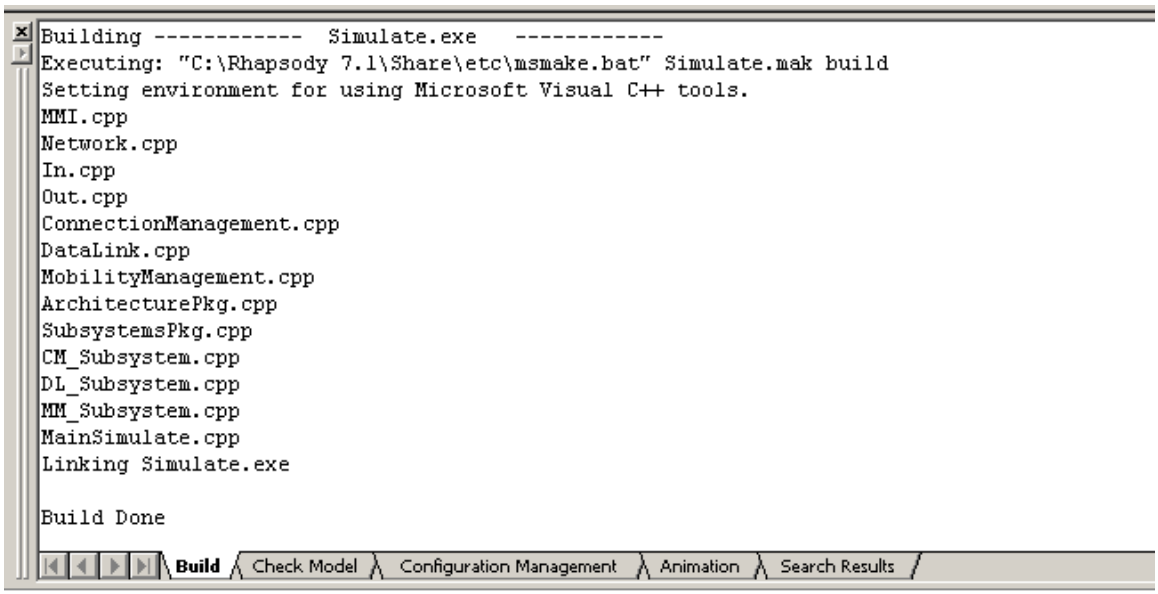
To build the model, do one of the following:

- Select **Code > Build Simulate.exe**, or
- Click the Make button on the **Code** toolbar.

Rational Rhapsody builds the model by performing the following tasks:

- Executes the makefile that it generated for the configuration.
- Sets up the environment for the compiler.
- Starts the compiler and linker, which run on the generated code. Once the code is compiled and linked, the Rational Rhapsody product displays the message `Build Done` in the Output window.

For a successful model build, the Build tab of the Output window should resemble the following figure:



Fixing Build Errors

If you receive build errors, double-click the error in the Output window to go to the source of the error. The source of the error appears as a highlighted element. Once you fix the problem, regenerate the code and rebuild the application until there are no error messages.

Any time you make changes to the model, you need to regenerate and rebuild the model before animating it. For more information about full code generation and an incremental code generation, refer to the *IBM Rational Rhapsody User Guide*. (Search the user guide PDF for “incremental code generation.”) You might also find it useful to use the Clean function. Do a search of the user guide PDF for “deleting old objects.”

Summary

In this lesson, you created generated code and built your model at its current point. You performed the following:

- Created a component and set its features
- Created a configuration
- Set a configuration as the active configuration
- Generated code in Rhapsody
- Built the handset model at its current point

You are now ready to proceed to the next lesson, where you continue to creating your handset model. You are going to define the message exchange between subsystems and subsystem modules when placing a call using sequence diagrams. You also get to regenerate code and rebuild your model, plus you learn a little about animation.

Lesson 5: Creating Sequence Diagrams

Sequence diagrams (SDs) describe how structural elements communicate with one another over time, and identify the required relationships and messages. Sequence diagrams can be used at different levels of abstraction. At higher levels of abstractions, sequence diagrams show the interactions between actors, use cases, and objects. At lower levels of abstraction and for implementation, sequence diagrams show the communication between classes and objects.

Sequence diagrams have an executable aspect and are a key animation tool. When you animate a model, Rational Rhapsody dynamically builds sequence diagrams that record the object-to-object messaging.

Goals for this Lesson

In this lesson, you are going to create the following sequence diagrams:

- **Place Call Request Successful** to identify the message exchange when placing a call
- **NetworkConnect** to identify the scenario of connecting to the network
- **Connection Management Place Call Request Success** to identify the message exchange between functions when placing a call

For ease of presentation, this section includes all sequence diagrams. Depending on your workflow, you might first identify the high-level communication scenario of placing a call and then refine the high-level structure diagram and object model diagrams, before defining the communication scenarios of the modules.

In addition, in this lesson you are going to set up for animation, as well as do a little by animating one of the sequence diagrams you create in this section.

Exercise 1: Creating the Place Call Request Successful SD

The Place Call Request Successful sequence diagram shows how subsystems interact during the scenario of successfully requesting to place a call. It identifies the order and exchange of messages between the objects as represented in the Handset System structure diagram, which you created in Lesson 2. By describing the flows through scenarios, you create the logical interfaces of the objects. For example, if a message is shown going into the **DataLink** object, you can see that the message belongs to the object as an event or operation.

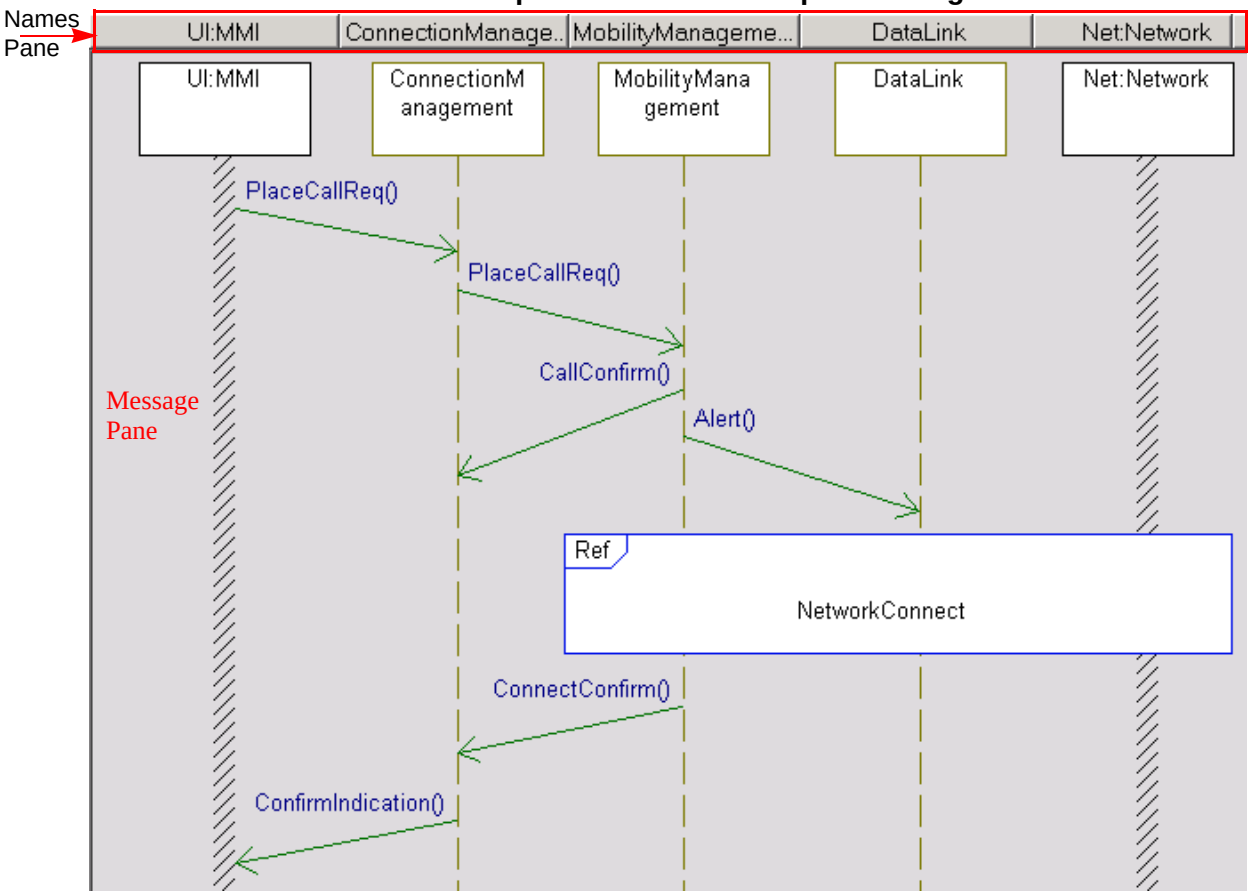
You draw a sequence diagram using the following general steps:

1. Draw the actor lines.
2. Draw classifier roles.
3. Draw messages.
4. Draw interaction occurrences.

This exercise describes each of these steps in detail.

The following figure shows the Place Call Request Successful sequence diagram that you are going to create in this exercise.

Place Call Request Successful Sequence Diagram

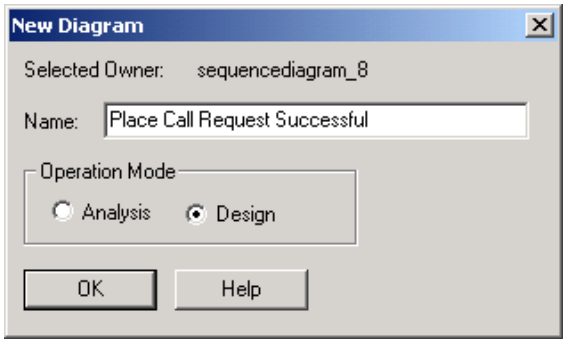


Rational Rhapsody separates sequence diagrams into a Names pane and a Message pane. The Names pane contains the name of each instance line or classifier role. The Message pane contains the elements that make up the interaction.

Task 1a: Creating the Place Call Request Sequence Diagram

To create a new sequence diagram, follow these steps:

1. Start Rational Rhapsody and open the handset model if they are not already open.
2. In the Rational Rhapsody browser, right-click the **SubsystemsPkg** package, and select **Add New > Sequence Diagram**. The New Diagram dialog box opens.
3. Type `Place Call Request Successful`, as shown in the following figure.



4. In the Operation Mode group, select the **Design** option button.

   Rational Rhapsody lets you create sequence diagrams in two modes:

   a. In *analysis mode*, you draw message sequences without adding elements to the model. This means you can brainstorm your analysis and design without affecting the generated source code. Once the design is finalized, you can realize the instance lines and messages so that they display in the Rational Rhapsody browser, and can have code generated for them.

   b. In *design mode*, every instance line and message you create or rename can be realized as an element (class, object, operation, or event) that appears in the Rational Rhapsody browser, and for which code can be generated. When you draw a message, Rational Rhapsody asks if you want to realize it. Click **Yes** to realize the message.

5. Click **OK** to close the dialog box.

Rational Rhapsody automatically creates the **Sequence Diagrams** category in the **SubsystemsPkg** package, and adds the name of the new sequence diagram. In addition, Rational Rhapsody opens the new diagram in the drawing area.

Note

You can also create a sequence diagram using the Tools menu. Refer to the *IBM Rational Rhapsody User Guide* for more information.

Task 1b: Drawing Actor Lines

In this task, you are going to draw the actor lines that represent the two objects, **MMI** and **Network**, as defined in the Handset System structure diagram by dragging them from the Rational Rhapsody browser to the diagram. *Actor lines* show how actors participate in the scenario. Actors are represented as instance lines with a column of diagonal lines. In use case diagrams and sequence diagrams, actors describe the external elements with which the system context interacts. For placement of the actor lines, use the **Place Call Request Successful Sequence Diagram** figure as a reference.

To draw actor lines, follow these steps:

1. In the Rational Rhapsody browser, expand the **ArchitecturePkg** package and the **Objects** category.
2. Click the **UI** object and drag-and-drop it at the beginning of the sequence diagram. Rational Rhapsody creates the actor line.
3. Click the **Net** object and drag-and-drop it at the end of the sequence diagram.

Task 1c: Drawing Classifier Roles

In this task, you are going to draw the classifier roles that represent the system components, **ConnectionManagement**, **MobilityManagement**, and **DataLink**.*Classifier roles* or instance lines are vertical timelines labeled with the name of an instance, which indicate the lifecycle of classifiers or objects that participate in the scenario. They represent a typical instance in the scenario being described. Classifier roles can receive messages from or send messages to other instance lines. Time proceeds downward on the vertical axis. For placement of the classifier roles, use the **Place Call Request Successful Sequence Diagram** figure as a reference.

To draw classifier roles, follow these steps:

1. In the Rational Rhapsody browser, expand the **SubsystemsPkg** package, the **Packages** category, the **CM_Subsystem** package, and the **Objects** category.
2. Click **<<Subsystem>> ConnectionManagement** and drag-and-drop it next to the **UI** object. Rational Rhapsody creates the classifier role with the name of the role in the Names pane.
3. In the browser, expand the **MM_Subsystem** package and the **Objects** category. Click **<<Subsystem>> MobilityManagement** and drag-and-drop it next to **ConnectionManagement**.
4. In the browser, expand the **DL_Subsystem** package and the **Objects** category. Click **<<Subsystem>> DataLink** and drag-and-drop it next to **MobilityManagement**.
5. Save your model.

Note

To add white space to (or remove it from) a sequence diagram (such as between actors lines and classifier roles), press the **Shift** key and drag the actor line or classifier role to its new location.

Task 1d: Drawing Messages

A *message* represents an interaction between objects, or between an object and the environment. A message can be an event, a triggered operation, or a primitive operation. Depending on the shape of the line, Rational Rhapsody interprets the message as follows:

- If the message line is horizontal, the message is interpreted as a triggered operation if the target is a reactive class, or a primitive operation if the target is a nonreactive class. A message line that is horizontal indicates that the operations are synchronous.
- If the message line is slanted, the message is interpreted as an event if the target is a reactive class, or as a primitive operation if the target is a nonreactive class. A message line that is slanted emphasizes that time passes between the sending and receiving of messages. Message lines that are slanted can cross each other.
- If the message line returns to itself, the message is interpreted as a primitive operation if the arrow folds back to a nonreactive class or if the arrow folds back immediately; or it is interpreted as an event if the arrow folds back sometime later. The arrow can be on either side of the instance line.

Note

Reactive classes can receive events, triggered operations, and primitive operations. Non-reactive classes can receive only messages that are calls to primitive operations.

In this task, you are going to draw events that represent the exchange of information when placing a call. The **UI** actor issues a request to connect when placing a call. Call and connect confirmations occur between **MobilityManagement** and **ConnectionManagement**. Alerts occur between **MobilityManagement** and **DataLink**. The user receives confirmation from **ConnectionManagement**. Use the **Place Call Request Successful Sequence Diagram** figure as a reference.

To draw messages, follow these steps:

1. Click the Message button on the **Drawing** toolbar.
2. Click the **UI** actor line to show that the first message comes from the **UI** actor when the user issues the command to place a call request.
3. Click the **ConnectionManagement** line to create a downward-slanted diagonal line. Rational Rhapsody creates a message with the default name **event_n()**, where ***n*** is an incremental integer starting with 0.

4. Rename the message `PlaceCallReq` press **Enter**.

   Note: Because you are creating the sequence diagram in design mode, each time you draw a new message, Rational Rhapsody asks if you want to realize the message. Click **Yes** to realize each new message.

5. Draw the following messages using the **Place Call Request Successful Sequence Diagram** figure as a reference:

   a. From **ConnectionManagement** to **MobilityManagement**, named `PlaceCallReq`

   b. From **MobilityManagement** to **ConnectionManagement**, named `CallConfirm`

   c. From **MobilityManagement** to **DataLink**, named `Alert`

6. Leave a space for the interaction occurrence (reference sequence diagram) you are going to create in **Task 1e: Drawing an Interaction Occurrence**.

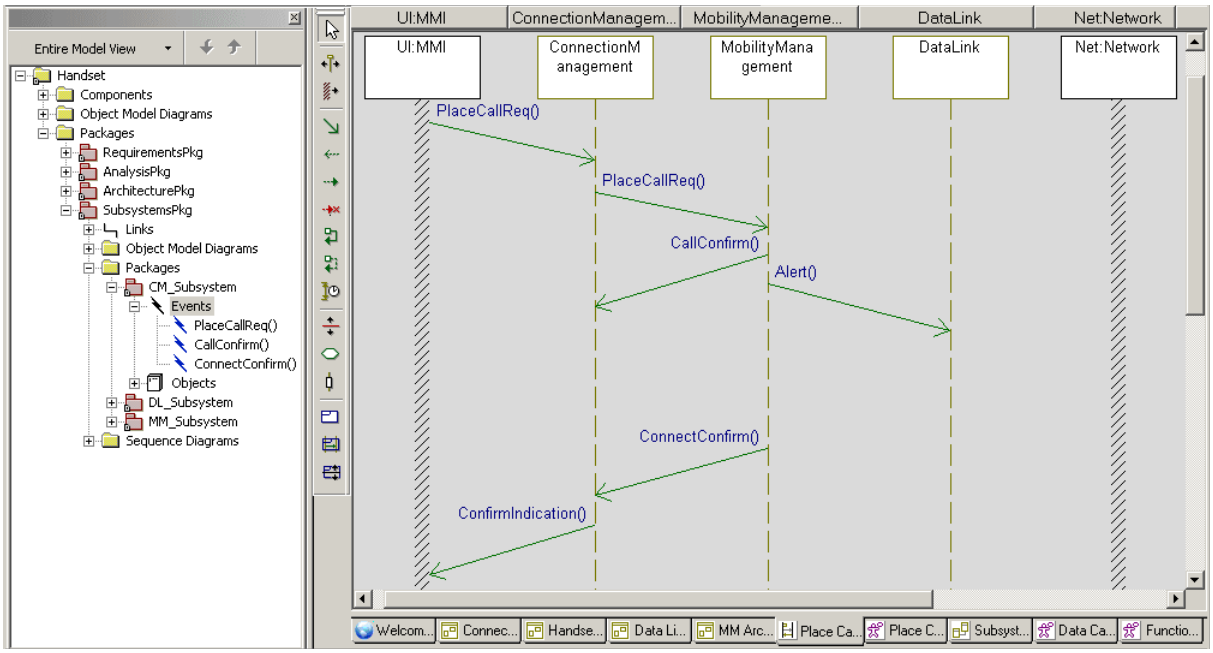
7. Draw the following messages using the **Place Call Request Successful Sequence Diagram** figure as a reference:

   – From **MobilityManagement** to **ConnectionManagement**, named `ConnectConfirm`

   – From **ConnectionManagement** to the **UI** actor, named `ConfirmIndication`.

8. Save your model.

9. In the Rational Rhapsody browser, view the realized events. Rational Rhapsody adds the new realized events to the package in which the message is passed. Rational Rhapsody adds **CallConfirm**, **ConnectConfirm**, and **PlaceCallReq** to the **Events** category in the **CM_Subsystem** package, as shown in the following figure.



Note

To locate an event in the Rational Rhapsody browser, select the element in the sequence diagram and click the Locate in Browser button on the standard toolbar or select **Edit > Locate in Browser**.

Task 1e: Drawing an Interaction Occurrence

In this task, you are going to draw an interaction occurrence. An *interaction occurrence* (or reference sequence diagram) allows you to refer to another sequence diagram from within a sequence diagram. It lets you break down complex scenarios into smaller scenarios that can be reused.

To draw an interaction occurrence, follow these steps:

1. Click the Interaction Occurrence button on the **Drawing** toolbar.
2. Using the **Place Call Request Successful Sequence Diagram** figure as a reference, draw the interaction occurrence below the **Alert** message and across the **MobilityManagement** instance line and the **Net** actor line. The interaction occurrence appears as a box with the **Ref** label in the top corner.
3. Type `NetworkConnect`. You are going to draw the NetworkConnect diagram in the next section, **Exercise 2: Creating the NetworkConnect SD**.
4. Save your model.

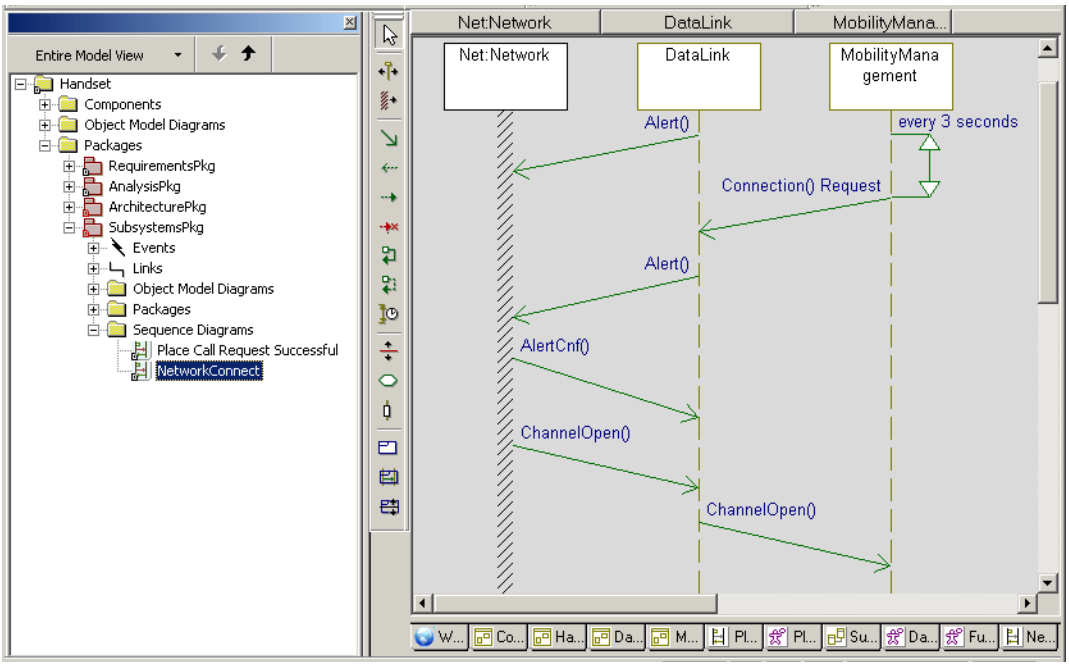
You have completed drawing the Place Call Request Successful sequence diagram. It should resemble the **Place Call Request Successful Sequence Diagram** figure.

Exercise 2: Creating the NetworkConnect SD

As mentioned in the previous exercise, NetworkConnect is a reference sequence diagram. In this exercise, you create this sequence diagram, which shows the scenario of connecting to the network when placing a call. It is a generic interaction that can be reused within voice, data, supplementary services, and short message services.

The following figure shows the NetworkConnect sequence diagram that you are going to create in this exercise.

NetworkConnect Sequence Diagram



Task 2a: Creating the NetworkConnect Sequence Diagram

To create the NetworkConnect sequence diagram, right-click the interaction occurrence, which you named **NetworkConnect** previously in the Place Call Request Successful sequence diagram, and select **Create Reference Sequence Diagram**. Rational Rhapsody opens the new diagram in the drawing area containing the three functions the interaction occurrence crosses on the Place Call Request Successful sequence diagram, and adds the NetworkConnect sequence diagram to the Rational Rhapsody browser.

Opening a Reference Sequence Diagram

When needed, once you have created a reference sequence diagram, you can open it using the following methods:

- Double-click the name of the diagram in the Rational Rhapsody browser.
- Right-click the interaction occurrence in the Place Call Request Successful sequence diagram and select **Create Reference Sequence Diagram**.

Task 2b: Drawing Messages

In this task, you are going to draw events using the **NetworkConnect Sequence Diagram** figure as a reference.

To draw messages, follow these steps.

Note

For a cleaner presentation of the task, re-order the classifier roles (**DataLink** and **MobilityManagement**) and actor line (**Net:Network**) as shown in the **NetworkConnect Sequence Diagram** figure. (Use click-and-drag in the drawing area.)

1. Click the Message button on the **Drawing** toolbar.
2. Draw the following messages:
   - From **DataLink** to **Net**, named `Alert`
   - From **MobilityManagement** to **DataLink**, named `ConnectionRequest`
   - From **DataLink** to **Net**, named `Alert`
   - From **Net** to **DataLink**, named `AlertCnf`
   - From **Net** to **DataLink**, named `ChannelOpen`
   - From **DataLink** to **MobilityManagement**, named `ChannelOpen`

   Note: When prompted, click **Yes** to realize each new message.

Rational Rhapsody adds the new realized events to the browser.

Task 2c: Drawing Time Intervals

In this task, you are going to set a time interval of 3 seconds in which **MobilityManagement** checks for a connection request. Sequence diagrams can specify the maximum amount of time that can elapse between two points. A *time interval* is a vertical annotation that shows how much (real) time should pass between two points in the scenario. The name of the time interval is free text; it is not constrained to be a number or unit.

To draw a time interval, follow these steps:

1. Click the Time Interval button on the **Drawing** toolbar.
2. Click near the top of the **MobilityManagement** line click the origin of the **ConnectionRequest** event. Rational Rhapsody draws two horizontal lines at the start and end points of the time interval, and a two-headed vertical arrow in the middle, indicating the time lapse between the two points.
3. Edit the label on the time interval (`<n sec>`) as follows:

   ```
   every 3 seconds
   ```

4. Save your model.

You have completed drawing the NetworkConnect diagram. It should resemble the **NetworkConnect Sequence Diagram** figure.

Task 2d: Moving Events

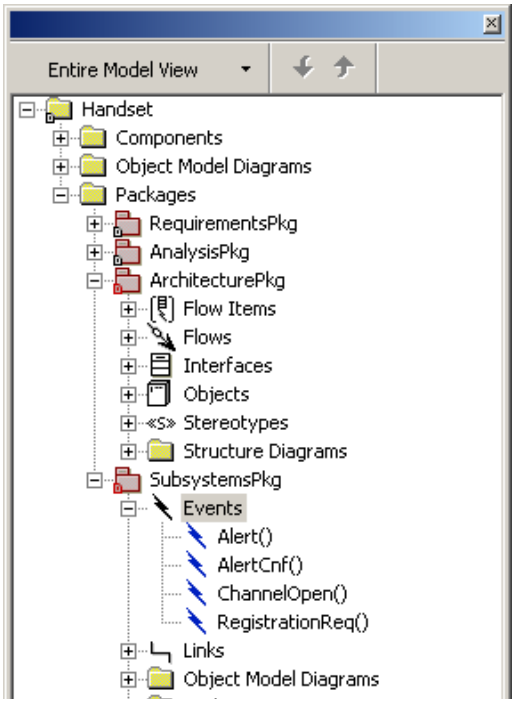
The **SubsystemsPkg** package contains the sequence diagrams that detail the design of the system and the flow of information. When you draw messages on the sequence diagrams, several messages are added to the **Events** category in the **ArchitecturePkg** package. To make these events available for model execution, you need to move them from the **ArchitecturePkg** package to the **SubsystemsPkg** package.

To move events, follow these steps:

1. Expand the **ArchitecturePkg** package and the **Events** category.
2. Select **Alert** and drag-and-drop it in the **SubsystemsPkg** package. Rational Rhapsody automatically creates the **Events** category in the **SubsystemsPkg** package and adds the **Alert** event.
3. Drag-and-drop the remaining events from the **ArchitecturePkg** package to the **SubsystemsPkg** package.

   Note: You can select multiple events to move by using **Shift+Click**.

4. Expand the **SubsystemsPkg** package and **Events** category to view the events you moved, as shown in the following figure. Also notice that there is no longer an **Events** category in the **ArchitecturePkg** package.

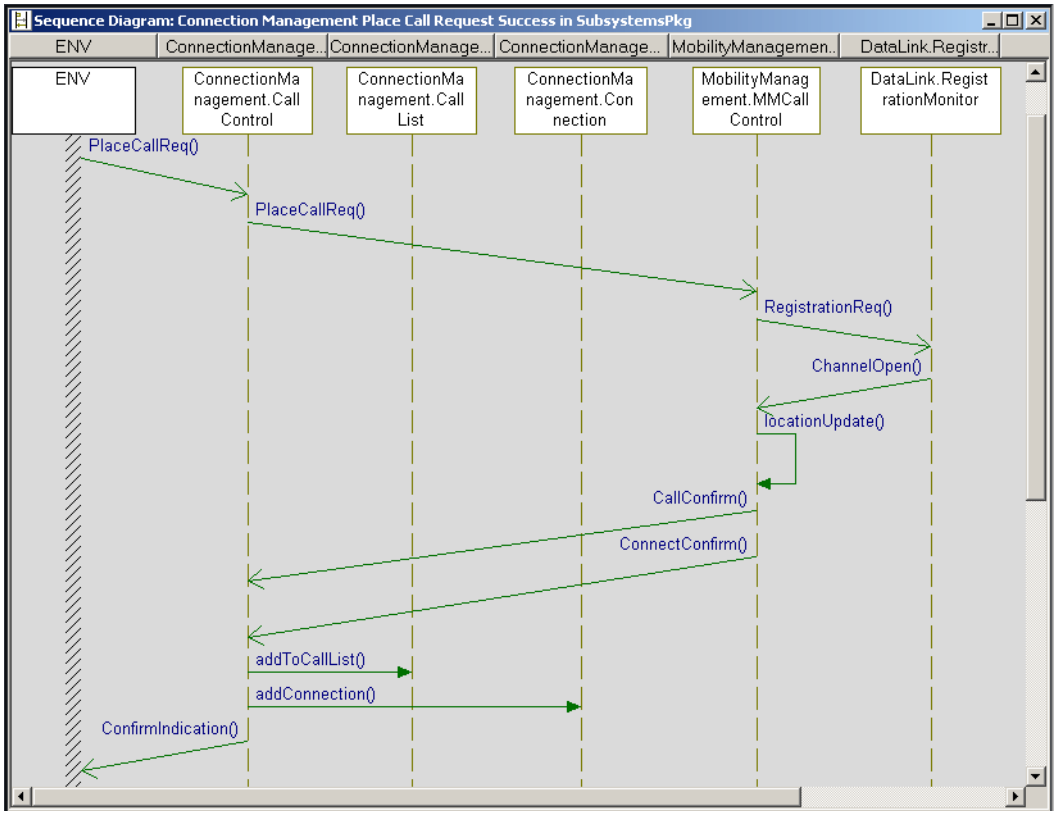


Exercise 3: Creating the Connection Management Place Call Request Success SD

The Connection Management Place Call Request Success sequence diagram shows the interaction of the subsystem modules. It identifies the part decomposition interaction when placing a successful call.

The following figure shows the Connection Management Place Call Request Success sequence diagram that you are going to create in this lesson.

Connection Management Place Call Request Success Sequence Diagram



Task 3a: Creating the Connection Management Place Call Request Success Sequence Diagram

Because the Connection Management Place Call Request Success sequence diagram identifies how the modules communicate, you are going to create it in the **SubsystemsPkg** package.

To create the Connection Management Place Call Request Success sequence diagram, follow these steps:

1. In the Rational Rhapsody browser, expand the **SubsystemsPkg** package right-click **Sequence Diagrams** and select **Add New Sequence Diagram**. The New Diagram dialog box opens.
2. Type `Connection Management Place Call Request Success`.
3. In the **Operation Mode** group, select the **Design** option button.
4. Click **OK** to close the dialog box.

Rational Rhapsody adds the name of the new sequence diagram to the **Sequence Diagrams** category in the browser. In addition, Rational Rhapsody opens the new diagram in the drawing area.

Task 3b: Drawing the System Border

In this task, you are going to draw the system border. The *system border* represents the environment and is shown as a column of diagonal lines. Events or operations that do not come from instance lines are drawn from the system border. You can place a system border anywhere an instance line can be placed; the most usual locations are the left or right side of the sequence diagram. Use the **Connection Management Place Call Request Success Sequence Diagram** figure as a reference.

To draw the system border, follow these steps:

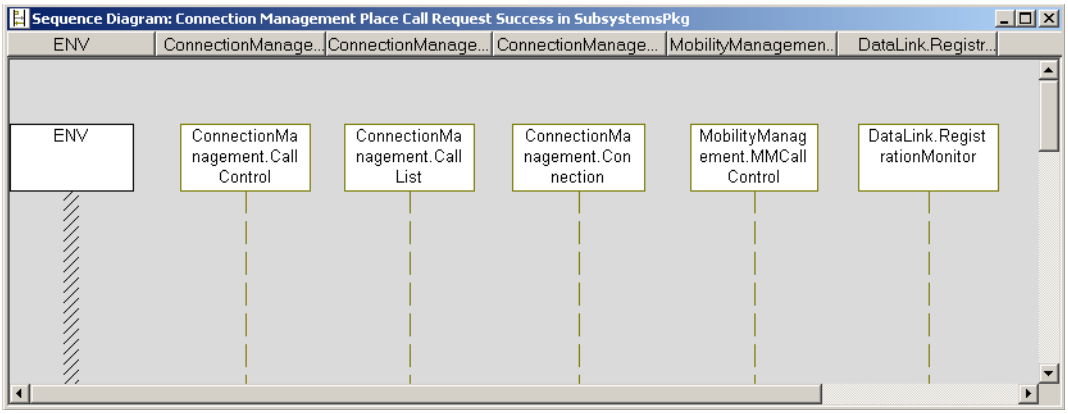
1. Click the System Border button on the **Drawing** toolbar.
2. Click on the left side of the diagram to place the border.

Task 3c: Drawing Classifier Roles

In this task, you are going to draw the classifier roles that represent the internal functions of the subsystems by dragging elements from the Rational Rhapsody browser to the sequence diagram. Use the **Connection Management Place Call Request Success Sequence Diagram** figure as a reference.

To draw classifier roles, follow these steps:

1. In the Rational Rhapsody browser, expand the **ConnectionManagement** object and the **Parts** category (found under the **CM_Subsystem** package).
2. Click **CallControl** and drag-and-drop it next to the system border. Rational Rhapsody creates the classifier role with the name of the function in the Names pane.
3. Click **CallList** and drag-and-drop it next to **CallControl**.
4. Click **Connection** and drag-and-drop it next to **CallList**.
5. In the browser, expand the **MobilityManagement** object and the **Parts** category (found under the **MM_Subsystem** package).
6. Click **MMCallControl** and drag-and-drop it next to **Connection**.
7. In the browser, expand the **DataLink** object and the **Parts** category (found under the **DL_Subsystem** package).
8. Click **RegistrationMonitor** and drag-and-drop it next to **MMCallControl**.
9. Save your model. Your Connection Management Place Request Success sequence diagram should resemble the following figure:



Task 3d: Drawing Messages

When the system receives a request to place a call, it validates and registers the user; once registered, it monitors the user's location. The call and connection are confirmed, the connection is set up, and confirmation is provided.

In this task, you are going to draw events using slanted lines, primitive operations using horizontal lines, and messages-to-self. Use the **Connection Management Place Call Request Success Sequence Diagram** figure as a reference. When prompted, click **Yes** to realize each new message.

To draw messages, follow these steps:

1. Click the Message button on the **Drawing** toolbar.
2. Draw the following events using slanted lines:
   a. From the system border to the **CallControl** line, named `PlaceCallReq`
   b. From **CallControl** to **MMCallControl**, named `PlaceCallReq`
   c. From **MMCallControl** to **RegistrationMonitor**, named `RegistrationReq`
   d. From **RegistrationMonitor** to **MMCallControl**, named `ChannelOpen`
3. Draw a message-to-self on the **MMCallControl** instance line, named `locationUpdate`.

   Note: Message names are case-sensitive.
4. Draw the following events:
   a. From **MMCallControl** to **CallControl**, named `CallConfirm`
   b. From **MMCallControl** to **CallControl**, named `ConnectConfirm`
5. Draw the following primitive operations using horizontal lines:
   a. From **CallControl** to **CallList**, named `addToCallList`
   b. From **CallControl** to **Connection**, named `addConnection`
6. Draw an event from **CallControl** to the system border, named `ConfirmIndication`.

Rational Rhapsody adds the new realized events and primitive operations to the part to which the message is passed. For example, Rational Rhapsody adds **locationUpdate** to the **Operations** category in the **MMCallControl** part (found under the **MM_Subsystem** package).

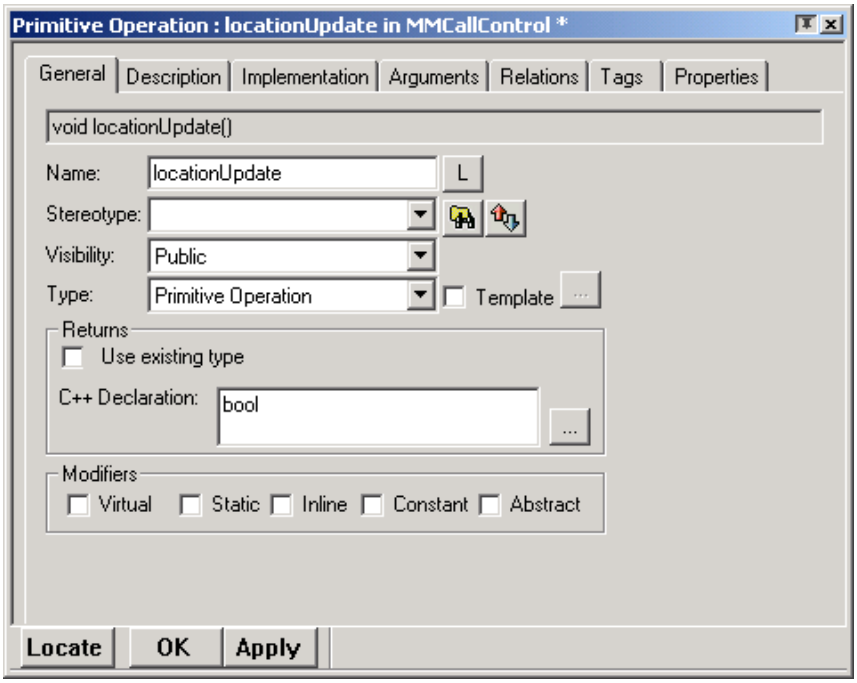
Task 3e: Setting the Features of locationUpdate

In this task, you are going to set the return type and implementation for **locationUpdate**.

To set the features, follow these steps:

1. On the Rational Rhapsody browser, double-click **locationUpdate** (expand the **MM_Subsystem** package, the **MobilityManagement** object, and then **MMCallControl** part), or right-click and select **Features**. The Features dialog box opens.

2. On the General tab, in the **Returns** group, clear the **Use existing type** check box and in the **C++ Declaration** box, type `bool`, as shown in the following figure.



3. On the **Implementation** tab, enter the following:

```
return TRUE;
```

4. Click **OK**.

Task 3f: Moving ConfirmIndication

When you drew messages on the sequence diagrams, the **ConfirmIndication** message was added to the **Events** category in the **AnalysisPkg** package. To make this event available for model execution, you need to move it from the **AnalysisPkg** package to the **CM_Subsystem** package.

To move **ConfirmIndication**, follow these steps:

1. Expand the **AnalysisPkg** package and the **Events** category.
2. Select **ConfirmIndication** and drag-and-drop it in the **CM_Subsystem** package. Rational Rhapsody adds **ConfirmIndication** to the **Events** category.
3. Save your model.

You have completed drawing the Connection Management Place Request Success sequence diagram. It should resemble the **Connection Management Place Call Request Success Sequence Diagram** figure.

Exercise 4: Animating a Sequence Diagram

As the model gets more and more complicated, it is a good practice to stop and validate the model periodically and provide design-level debugging. One of the primary methods the Rational Rhapsody product uses to simulate a model is *animation*.

In this exercise, you are going to animate the Connection Management Place Call Request Success sequence diagram. Note that this exercise introduces you to animation. You will learn more about and do more animation in subsequent lessons.

Animation is the observable execution of behaviors and associated definitions in the model. Rational Rhapsody animates the model by executing the code generated, with instrumentation, for classes, operations, and associations. Once you start model animation, you can open animated diagrams, which let you observe the model as it is running and perform design-level debugging. You can step through the model, set and clear breakpoints, inject events, and generate an output trace.

Task 4a: Changing the Settings for the Debug Configuration

In **Lesson 4: Generating Code and Building Your Model** you created a component and configuration so that you could generate code and build the handset model at that point in time. To be able to animate a model, you have to change the settings for the Debug configuration.

To change the settings for the Debug configuration for animation, follow these steps:

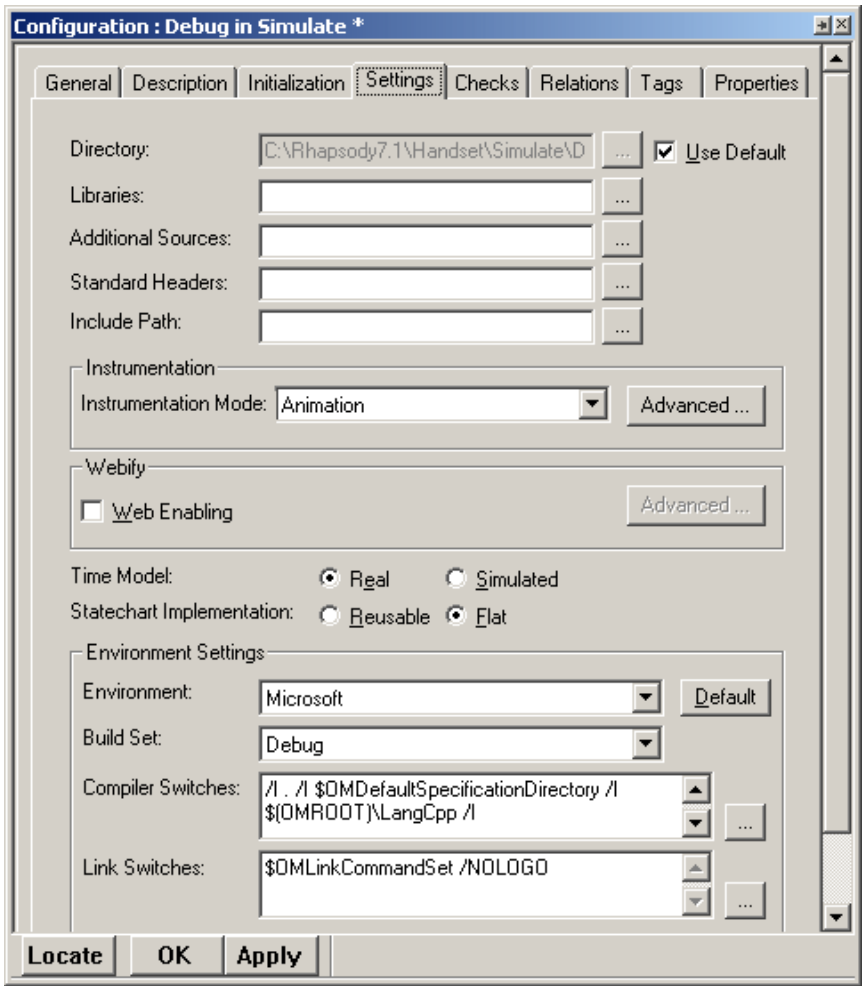
1. In the Rational Rhapsody browser, double-click **Debug** or right-click and select **Features**. The Features dialog box opens.
2. On the **Initialization** tab, make sure the following values are already set (they should be):
   a. In the **Initial Instances** group, select the **Explicit** option button.
   b. Select the **Generate Code for Actors** check box.

   Note: For more information about these options, refer to the *IBM Rational Rhapsody User Guide*. (Do a search of the user guide PDF for “initialization tab” and go to the section on this topic.)
3. Define the environment so that Rational Rhapsody knows how to create an appropriate makefile. On the **Settings** tab, set the following values:
   a. In the **Instrumentation** group, from the **Instrumentation Mode** drop-down list box, select **Animation**. This adds instrumentation code, which makes it possible to animate the model.
   b. In the **Time Model** group, if not already selected, select the **Real** (for real time) option button.

c. In the **Statechart Implementation** group, if not already selected, select the **Flat** option button. Rational Rhapsody implements states as simple, enumerated-type variables.

d. Rational Rhapsody sets the values in the **Environment Settings** group based on the compiler settings you configured during installation. If you want to use a different compiler, select a system compiler from the drop-down menu in the **Environment** box.

This example uses a system with the Microsoft compiler, as shown in the following figure. Your environment might use a different compiler.



4. Click **OK**.

Task 4b: Regenerating Code and Rebuilding Your Model

Before you can run animation for any of the sequence diagrams you created in this lesson, you have to regenerate the code and rebuild your model.

To regenerate code and rebuild your model, follow these steps:

1. Make sure **Debug** is your active configuration. It should appear in boldtype in the Rational Rhapsody browser when it is set as the active configuration. If needed, in the Rational Rhapsody browser, right-click the **Debug** configuration select **Set as Active Configuration**.

   Note: If you have more than one configuration, you can also select the active configuration from the drop-down list on the **Code** toolbar.

2. If the Output window is already open and there is information on the Build tab, to ensure that you will only be looking at information for the latest code generation/build, right-click on the tab select **Clear**. You might want to do this if information from a previous generation/build is still there.

3. Select **Code > Re Generate > Debug**. If applicable, fix any errors noted on the Build tab of the Output window.

4. Select **Code > Rebuild Simulate.exe**. If applicable, fix any errors noted on the Build tab.

Task 4c: Starting Animation

Note

If you have many diagrams opened in the drawing area, you might find it less confusing if you close them before you do this task. To close a diagram, click the Windows Close button for the diagram. Save a diagram if necessary.

If you have any sequence diagrams already open, Rational Rhapsody automatically creates an animated sequence diagram for each one that is open. Closing all sequence diagrams before you do this task will be less confusing, as this task deals with only one sequence diagram.

To start animation, do one of the following:

- Select **Code > Run Simulate.exe**, or
- Click the Run Executable button .

Rational Rhapsody starts animation and performs the following tasks:

- Runs the application to **main()**.
- Displays the **Animation** toolbar, which lets you control the animation process.
- Displays a log window, which provides input to and output from the model. You can position and resize the log and Rational Rhapsody windows so both are visible.
- Displays the following two output panes:
  - **Call Stack** to show the logical call stack of the executing model at the design level, rather than the code level.
  - **Event Queue** to show the events waiting on the event queue of the executing process.

Note

If the output panes are not displayed, select **View > Call Stack** or **View > Event Queue**. The output panes are dockable, so you can move them out of the Rational Rhapsody GUI to increase the viewable area for animations. To move a window, click-and-drag it to another location.

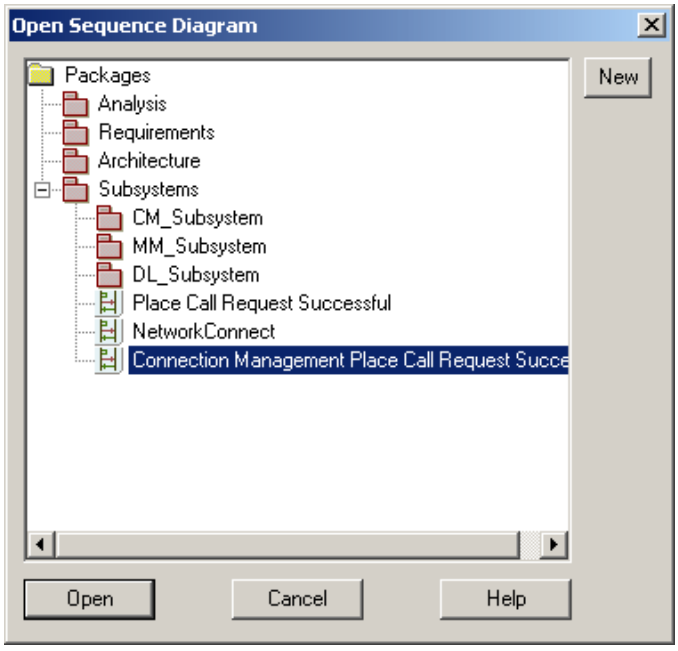
Task 4d: Animating a Sequence Diagram

Animated sequence diagrams show how objects pass messages while the model is executing. You do not manually add messages to an animated sequence diagram—the animation process adds them for you while the model is running. This means you can observe the communication taking place in the system. You can then compare the message sequence to the non-animated sequence diagrams to see whether the model is behaving correctly.

In this task, you are going to animate the Connection Management Place Call Request Success sequence diagram you created in **Exercise 3: Creating the Connection Management Place Call Request Success SD**.

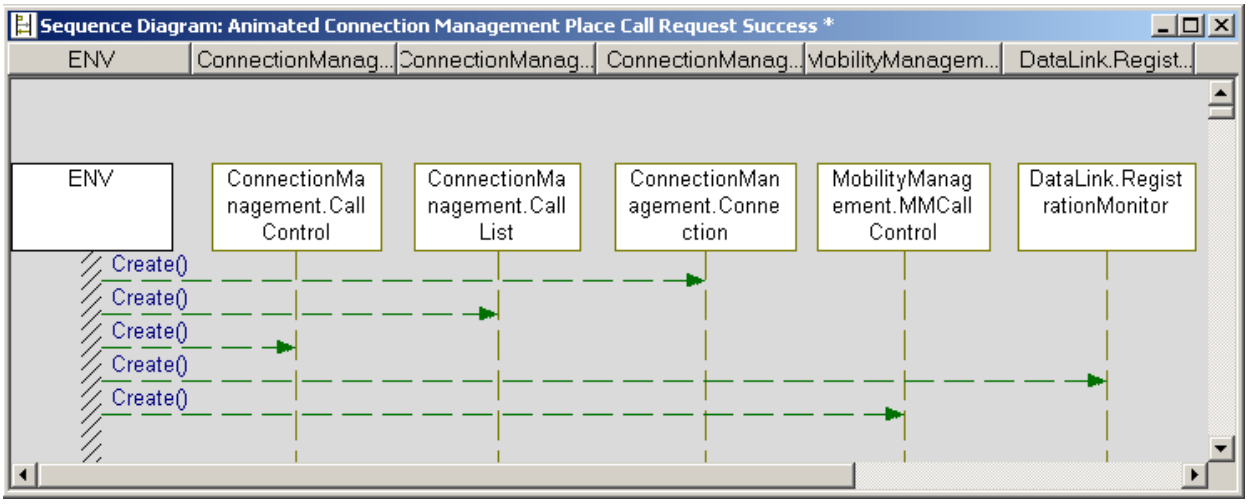
To animate the sequence diagram, follow these steps:

1. Select **Tools > Animated Sequence Diagram**. The Open Sequence Diagram dialog box opens.
2. Expand **SubsystemsPkg** and select **Connection Management Place Call Request Success**, as show in the following figure.



3. Click **Open**. This creates an animated version of the SD with the same instance lines as the original, but without the messages.

4. Click the Go button on the **Animation** toolbar. Rational Rhapsody creates the constructors for the objects, as shown in the following figure.



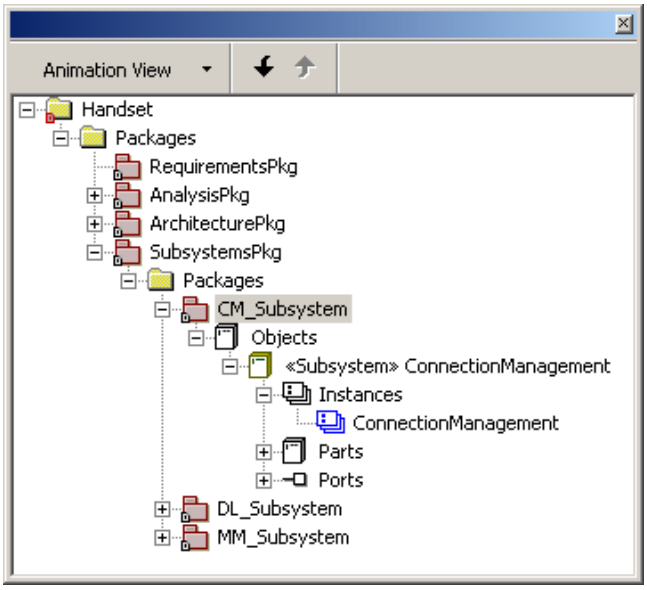
Task 4e: Viewing the Browser

During animation, Rational Rhapsody adds the **Instances** category to the Rational Rhapsody browser, which provides information on the status of instances, and their attributes and relations.

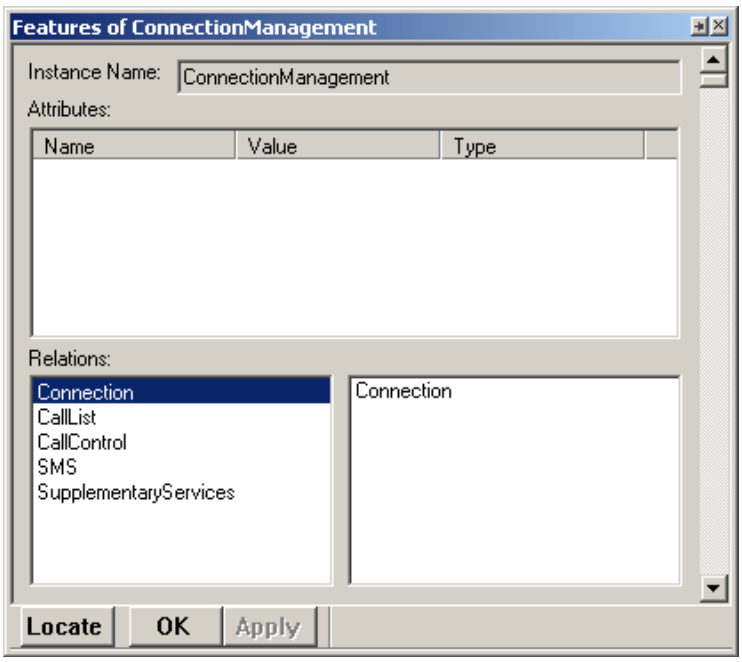
To view the **Instances** category on the Rational Rhapsody browser, follow these steps:

1. In the browser, expand the **SubsystemsPkg** package, the **CM_Subsystem** package, the **ConnectionManagement** object, and then **Instances** category.

2. From the browser filter drop-down list, select **Animation View** so that the browser displays only the elements relevant to your current task.



3. Double-click **ConnectionManagement** or right-click and select **Features**. The Features dialog box opens with the current values of all the initialized attributes and relations for **ConnectionManagement**, as shown in the following figure.



4. Click **OK**.

5. Save your model. Answer **Yes** when asked if you want to save your animated Connection Management Place Call Request Success sequence diagram.

Task 4f: Quitting Animation

To end the animation session, follow these steps:

1. Click the Animation Break button on the **Animation** toolbar click the Quit Animation button .
2. Click **Yes** to confirm ending the animation session.

The Animation tab on the Output window displays the message `Animation session terminated`.

Note

When you close the project or an animated diagram, Rational Rhapsody prompts whether or not you want to save the animated diagram. Saving an animated diagram is useful in order to compare the results of the current session to those of different execution scenarios.

Summary

In this lesson, you created sequence diagrams, which identify the message exchange between subsystems and subsystem modules when placing a call. You became familiar with the parts of a sequence diagram and created the following:

- System border
- Classifier roles and actor lines
- Interaction occurrences
- Events and primitive operations
- Time intervals
- Timeouts

In this lesson, you also set up for animation. To test that you can do animation, you created an animated sequence diagram. You will learn more about and do more animation in subsequent lessons.

You are now ready to proceed to the next lesson, where you are going to identify the functional flow of users placing a call and registering users on the network using activity diagrams.

Lesson 6: Creating Activity Diagrams

Activity diagrams show the dynamic aspects of a system and the flow of control from activity to activity. They describe the essential interactions between the system and the environment, and the interconnections of behaviors for which the subsystems or components are responsible. They can also be used to model an operation or the details of a computation. In addition, you can animate activity diagrams to verify the functional flow.

Goals for this Lesson

In this lesson, you are going to create the following activity diagrams:

- **MMCallControl** to identify the functional flow of users placing a call, which includes registering users on the network, providing their current location, and obtaining an acceptable signal strength.
- **InCall** to identify the flow of information once the system connects the call.
- **RegistrationMonitor** to identify the functional flow of registering users on the network, which includes monitoring registration requests and sending received requests to the network.

In addition, in this lesson, you are going to animate an activity diagram.

Exercise 1: Creating the MMCallControl Activity Diagram

The MMCallControl activity diagram shows the functional flow that supports the mobility of users when placing a call, which includes registering users on the network, providing their current location, and obtaining an acceptable signal strength. When the user places a call, the system leaves the **Idle** action element, checks for an acceptable signal strength and whether the wireless telephone is registered. It then waits for the call to connect and enters a connection action element.

An *action element* represents function invocations with a single exit transition when the function completes.

Note

The activity diagrams in this section use labels to provide descriptions of the actions, rather than language.

You are going to draw an activity diagram using the following general steps:

1. Draw swimlanes.
2. Draw action elements.
3. Draw a subactivity.
4. Draw send action states.
5. Draw a default flow.
6. Draw transitions.

This exercise describes these steps in detail.

Task 1a: Creating an Activity Diagram

To create an activity diagram, follow these steps:

1. Start Rational Rhapsody and open the handset model if they are not already open.
2. In the Rational Rhapsody browser, expand the **SubsystemsPkg** package, the **MM_Subsystem** package, the **MobilityManagement** object, and the **Parts** category. Right-click **MMCallControl** and select **Add New > Activity Diagram**.

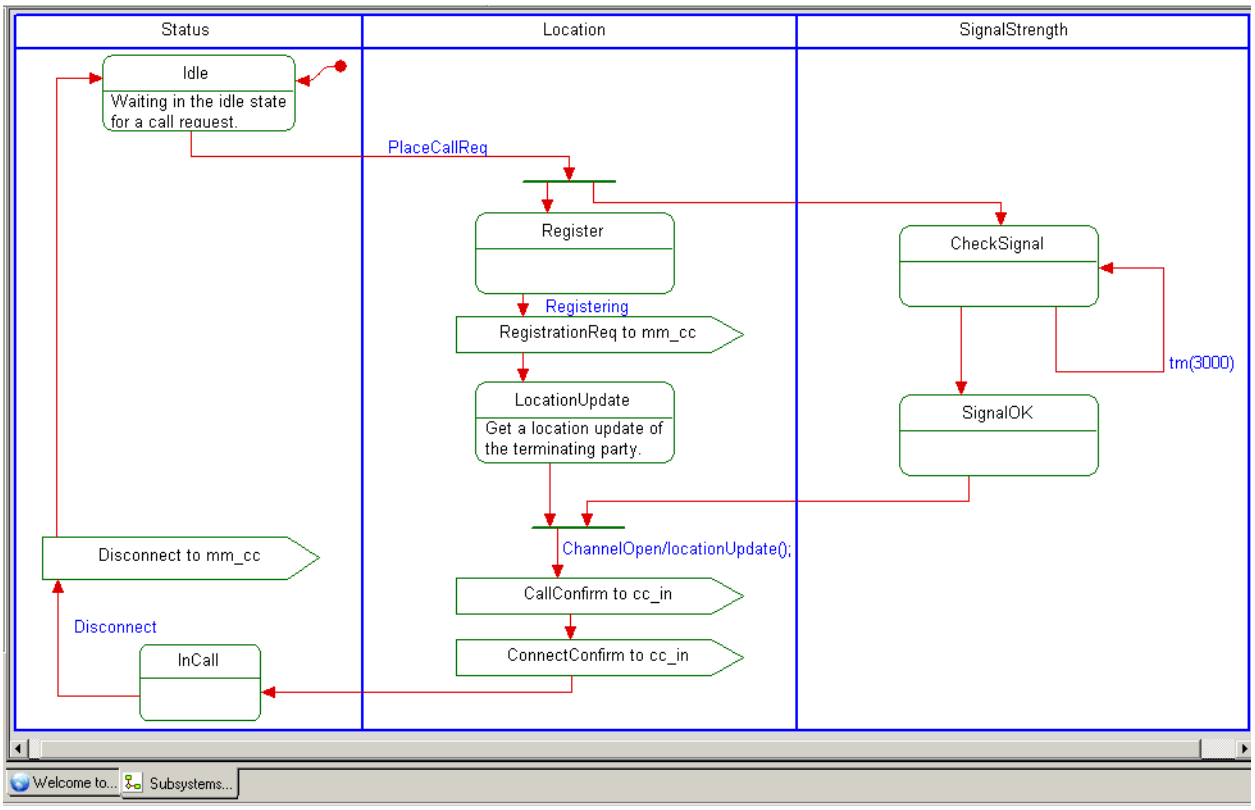
   or

   Open the MM Architecture structure diagram. Right-click **MMCallControl** and select **New Activity Diagram**.

Rational Rhapsody automatically adds the **Activity Diagram** category and the new activity diagram for the **MMCallControl** part in the Rational Rhapsody browser, and opens the new activity diagram in the drawing area.

The following figure shows the MMCallControl activity diagram that you are going to create in this exercise.

MMCallControl Activity Diagram



Task 1b: Drawing Swimlanes

In this task, you are going to draw swimlanes. *Swimlanes* organize activity diagrams into sections of responsibility for actions and subactions. Vertical, solid lines separate each swimlane from adjacent swimlanes. To draw swimlanes, you first need to create a swimlane frame and then a swimlane divider. Use the **MMCallControl Activity Diagram** figure as a reference.

To draw swimlanes, follow these steps:

1. You might find it helpful to expand the drawing area by closing the Rational Rhapsody browser (click the browser's Close button or select **View > Browser**). This gives you more space for the drawing area.

2. Click the Swimlanes Frame button on the **Drawing** toolbar.

3. Click to place one corner drag diagonally to draw the swimlane frame.

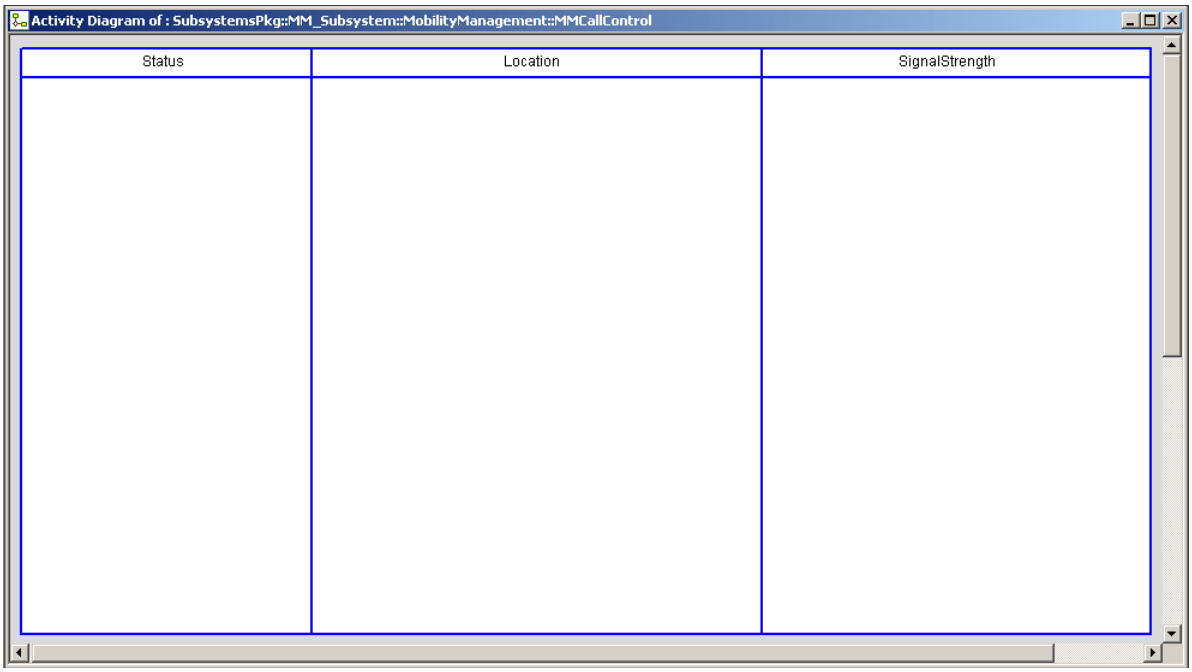
4. Click the Swimlanes Divider button on the **Drawing** toolbar for each swimlane:

   - Click about a third of the way in from the left edge of the swimlane frame. You have created two swimlines. The one on the left is named **swimlane_0** and the one on the right is named **swimlane_1**.
   - Click about the middle of swimlane_1 to create a third swimlane. You have three swimlanes: **swimlane_0**, **swimlane_2**, and **swimlane_1**. Your swimlane numbers might be different.

5. Starting from the leftmost swimlane, change its name to `Status`, the middle swimlane to `Location`, and the rightmost swimlane to `SignalStrength`. (Double-click the swimlane to open its Features dialog box.)

   - The **Status** swimlane tracks the status of calls.
   - The **Location** swimlane tracks the location of users.
   - The **SignalStrength** swimlane tracks the signal strength of users.

6. Save your model. Your activity diagram should resemble the following figure:



Task 1c: Drawing Action Elements

In this task, you are going to draw the action elements that represent the functional processes, and then add names to the action elements. Use the **MMCallControl Activity Diagram** figure as a reference.

Note

You add names to action elements using the Features dialog box. When you draw an action element and type a name in the action element on the diagram, that name becomes the action, not the name of the action.

To draw action elements, follow these steps:

1. Click the Action button on the **Drawing** toolbar.
2. At the top of the **Status** swimlane, click to create an action element press **Ctrl+Enter**.

   Note: If you press **Enter**, you move your cursor to a new line. In this case, you have to press **Ctrl+Enter** to end your action.
3. Double-click the action element, or right-click the action element and select **Features**. The Features dialog box opens.
4. On the General tab, in the **Name** box, type `Idle`. This indicates that no call is in progress.
5. On the Description tab, type the following:

   ```
   Waiting in the Idle state for a call request.
   ```
6. Click **OK** to apply the changes and close the Features dialog box.
7. Set the display options. Right-click the action element, select **Display Options**:
   a. From the **Show Name** group, select the **Name** option button.
   b. From the **Show Action, Description or Label** group, select the **Description** option button.
   c. Click **OK** to close the Display Options dialog box.

   Note: You can widen the action element if necessary to make the name and description appear better on your diagram.

8. Repeat the previous steps but create the following action elements where noted:

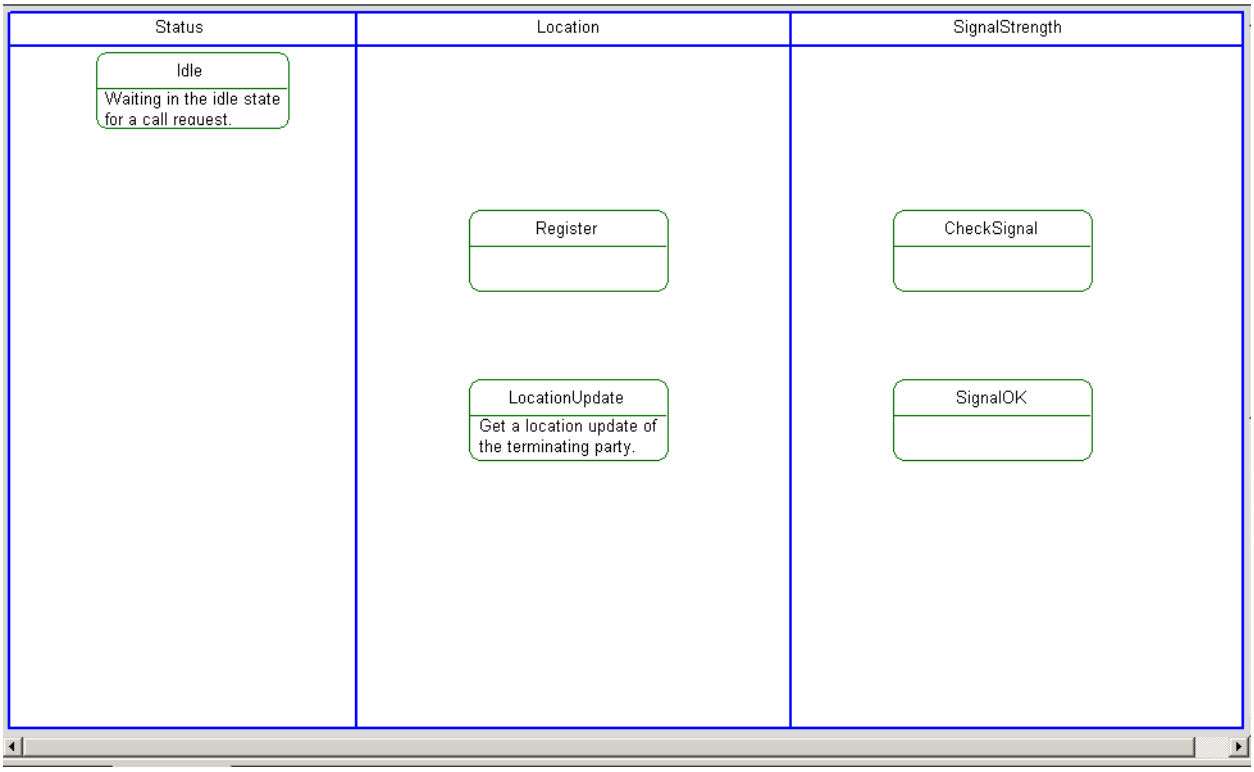
   a. In the lower half of the **Location** swimlane, draw an action element and name it `LocationUpdate` and include the following description:

      Get a location update of the terminating party.

   b. In the upper half of the **SignalStrength** swimlane, draw an action element and name it `CheckSignal`.

   c. Above the **LocationUpdate** action element, draw an action element and name it `Register`.

   d. Below the **CheckSignal** action element, draw an action element and name it `SignalOK`. Your diagram should resemble the following figure:



Task 1d: Drawing a Default Flow

In this task, you are going to draw a default flow. The action element with the default flow is the *default* action element. It is the initial action element of the object. **Idle** is in the default action element as it waits for call requests. Once the default action is activated, other actions in the MMCallControl activity diagram can happen. Use the **MMCallControl Activity Diagram** figure as a reference.

To draw a default flow, follow these steps:

1. Click the Default Flow button on **Drawing** toolbar.
2. Click to the right of the **Idle** action element click its edge click the mouse button again (this is the same as pressing **Enter**).
3. Save your model.

Task 1e: Drawing a Subactivity

In this task, you are going to draw the **InCall** subactivity, which indicates that the call has been established. A *subactivity* represents the execution of a non-atomic sequence of steps nested within another activity. It looks like an action element with a subactivity icon in its lower, right corner, depicting a nested activity diagram. Use the **MMCallControl Activity Diagram** figure as a reference.

To draw a subactivity, follow these steps:

1. Click the Subactivity button on the **Drawing** toolbar.
2. In the bottom section of the **Status** swimlane, click to create a subactivity.
3. Double-click the subactivity element you just created, or right-click it and select **Features**. The Features dialog box opens.
4. On the **General** tab, in the **Name** box, type `InCall`.
5. Click **OK**.

In the subsequent section, **Exercise 2: Creating the InCall Subactivity Diagram**, you are going to open and draw the **InCall** subactivity diagram.

Task 1f: Drawing Send Action States

The Send Action State element can be used to represent the sending of events to external entities.

Send Action State elements allow you to specify the event to send, the event target, and values for event arguments. This is a language-independent element that is translated into the relevant implementation language during code generation.

To define the element, provide the following information in the Features dialog box:

- From the **Target** drop-down list, select the object that is to receive the event.
- From the **Event** drop-down list, select the event that should be sent.
- When necessary, provide values for the event arguments by selecting the argument in the argument list and clicking the **Value** column.

For more information about send action states, refer to the *IBM Rational Rhapsody User Guide*. (Do a search of the user guide PDF file for “send action state elements” and go to the section on this topic.)

To draw a send action state, follow these steps:

1. Click the Send Action State button on the **Drawing** toolbar.
2. Click in the Status swimlane between **Idle** and **InCall**.
3. Double-click the Send Action element on the diagram. The Features dialog box opens.
4. On the **General** tab, in the **Target** drop-down list, select **mm_cc in SubsystemsPkg**.
5. In the **Event** drop-down list, select <<**New**>>.
6. In the dialog box that opens, on the **General** tab, in the **Name** box, type `Disconnect`.
7. Click **OK** to close that dialog box.
8. Click **OK** to close the Features dialog box for the send action.

   The **Disconnect to mm_cc** send action state element you just created sends an asynchronous message out the **mm_cc** port when disconnecting.

9. Draw the following send action state elements:

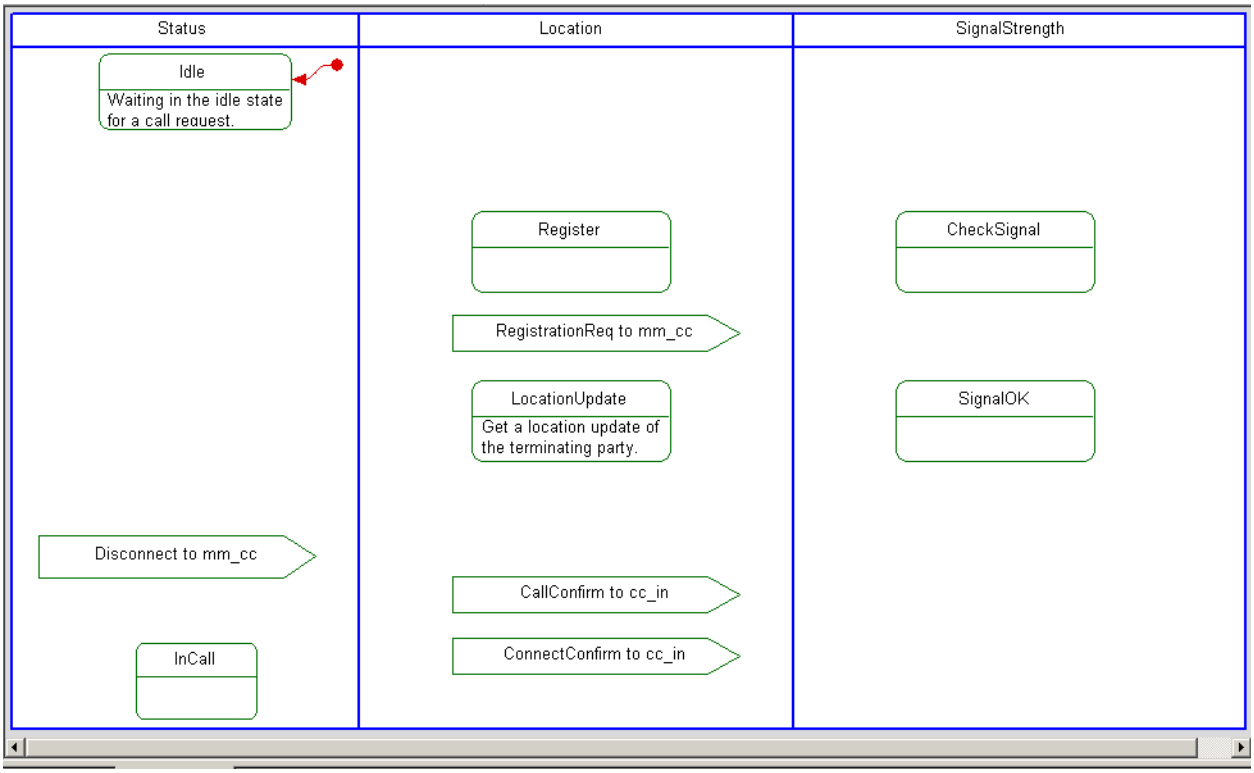
   - Between the **Register** and **LocationUpdate** action elements: For the **Target**, select **mm_cc in SubsystemsPkg**; for the **Event**, select **RegistrationReq in SubsystemsPkg**.

     This send action state sends an asynchronous message out the **mm_cc** port for registration requests.
   - After the **LocationUpdate** element: For the **Target**, select **cc_in in SubsystemsPkg**; for the **Event**, select **CallConfirm in SubsystemsPkg**.
   - After the **CallConfirm in SubsystemsPkg** send action state element: For the **Target**, select **cc_in in SubsystemsPkg**; for the **Event**, select **ConnectConfirm in SubsystemsPkg**.

   The last two send action states send asynchronous messages out the **cc_in** port.

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

11. Save your model. Your model should resemble the following figure:



Task 1g: Drawing Transitions

A *transition* represents a relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied. In this task, you are going to draw the following transitions:

- Transitions between actions
- Fork and join transitions
- Timeout transition

Drawing Transitions Between Actions

In this task, you are going to draw two transitions: one named `Disconnect`, and one with the label `Registering`. Use the **MMCallControl Activity Diagram** figure as a reference.

To draw transitions between actions, follow these steps:

1. Click the Activity Flow button on the **Drawing** toolbar.
2. Click the **InCall** subactivity action click the **Disconnect to mm_cc** send action state element.
3. Type the name `Disconnect` press **Ctrl+Enter**.

   Note: To change the line shape of a transition, right-click the line, select **Line Shape**, and then **Straight**, **Spline**, **Rectilinear**, or **Re-Route**.
4. Draw a transition from the **Disconnect to mm_cc** send action state element to the **Idle** action element.
5. Draw a transition from **Register** to the **RegistrationReq to mm_cc** send action state element, and then from the send action state element to the **LocationUpdate** action element.

   You are going to label this element in the next set of instructions.
6. Draw a transition from **CheckSignal** to **SignalOK**.

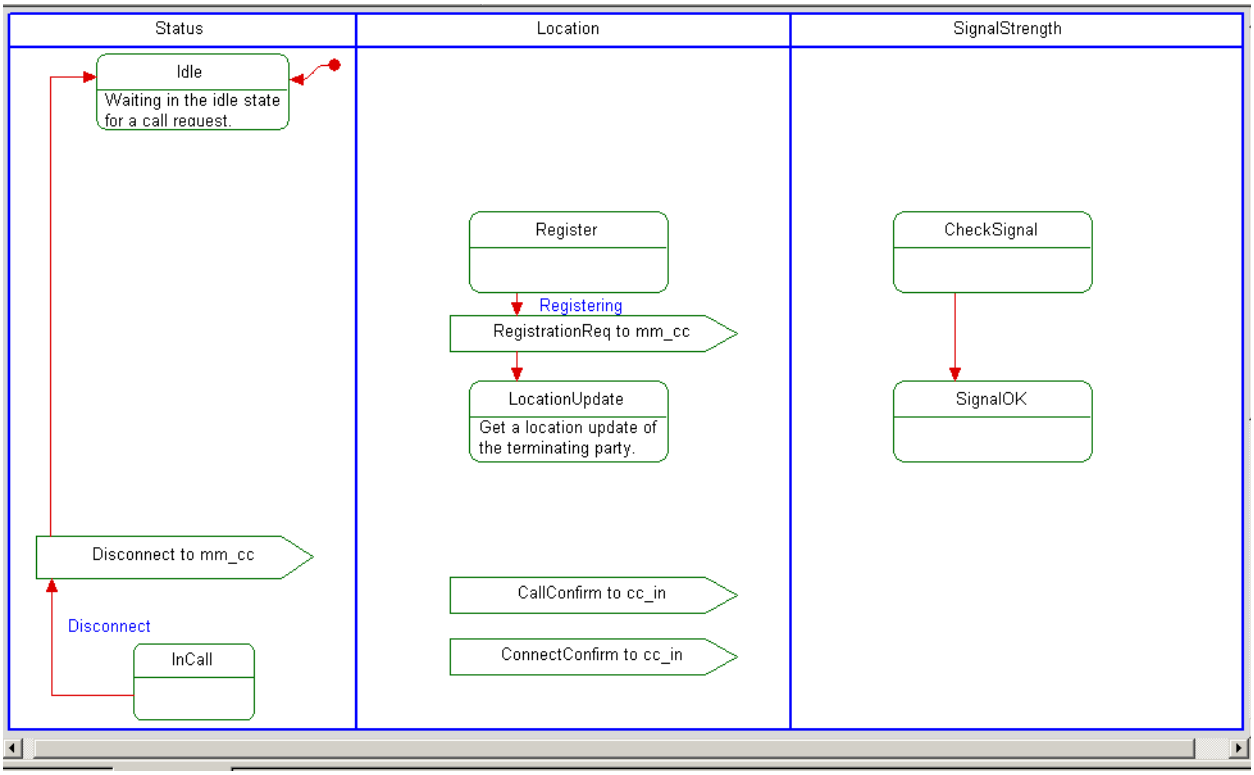
Labeling Elements

In this task, you are going to label the transition between **Register** and **RegistrationReq to mm_cc**.

You can assign a descriptive label to an element. The label of an element does not have any meaning in terms of generated code, but it lets you to easily reference and locate elements in diagrams and dialog boxes. A label can have any value and does not need to be unique.

To label elements, follow these steps:

1. Double-click the transition between **Register** and **RegistrationReq to mm_cc** or right-click and select **Features**. The Features dialog box opens.
2. Click the **L** button next to the **Name** box. The Name and Label dialog box opens.
3. Type `Registering` in the **Label** box.
4. Click **OK** to close the Name and Label dialog box.
5. Click **OK** to close the Features dialog box.
6. To display the label, right-click the transition and select **Display Options** to open the Display Options dialog box; from the **Display Name** group, select **Label**; then click **OK**.
7. Save your model. Your diagram should resemble the following figure:



Task 1h: Drawing a Fork Synchronization

In this task, you are going to draw a fork synchronization bar. A *fork synchronization* represents the splitting of a single flow into two or more outgoing flows. It is shown as a bar with one incoming transition and two or more outgoing transitions. Use the **MMCallControl Activity Diagram** figure as a reference.

To draw a fork synchronization bar, follow these steps:

1. Click the Draw Fork Sync Bar button on the **Drawing** toolbar.
2. In the **Location** swimlane, click above **Register**. Rational Rhapsody adds the fork synchronization bar.
3. Click the Activity Flow button, and draw a single incoming transition from **Idle** to the synchronization bar. Type `PlaceCallReq` press **Ctrl+Enter**. This transition indicates that the interface has initiated a call request. **PlaceCallReq** corresponds to the trigger of the transition.
4. Use the Activity Flow button to draw the following outgoing transitions from the synchronization bar:
   a. To the **Register** action.
   b. To the **CheckSignal** action.
5. To change the line shape of a transition, right-click the line, select **Line Shape**, and then **Straight**, **Spline**, **Rectilinear**, or **Re-Route**.

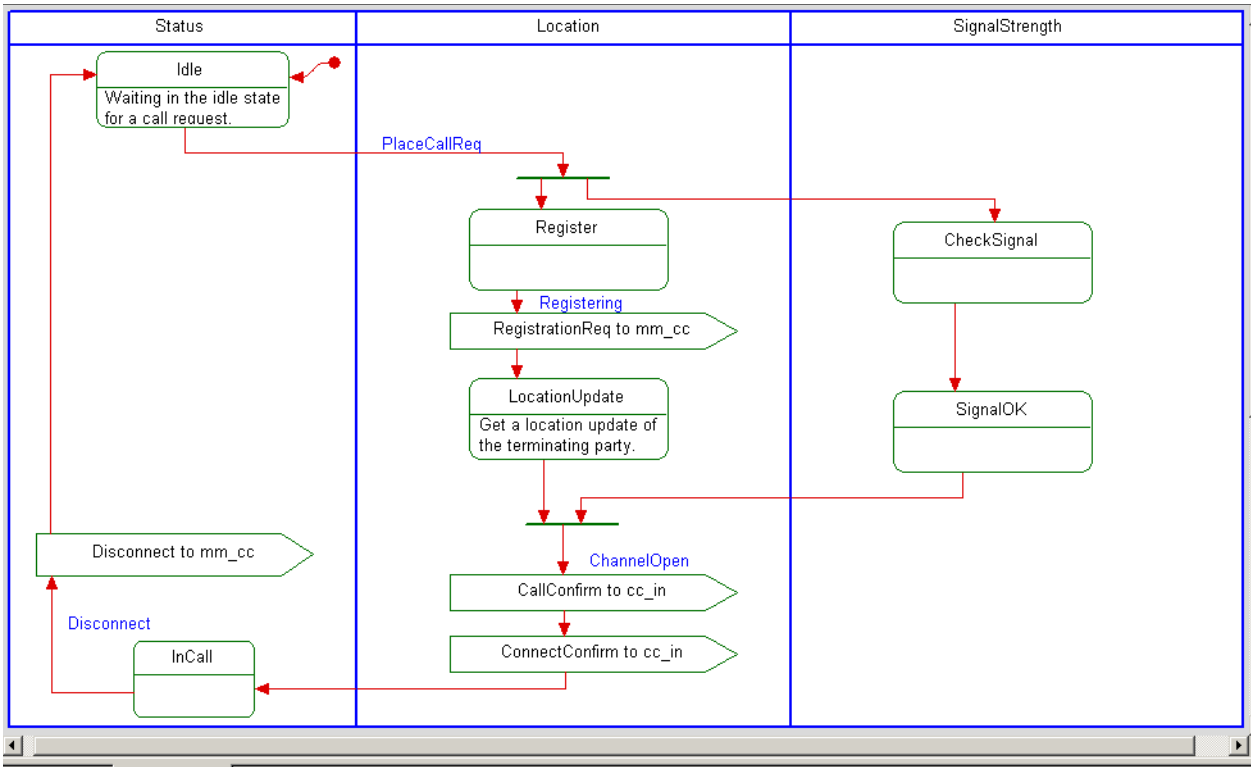
Task 1i: Drawing a Join Synchronization

In this task, you are going to draw a join synchronization bar. A *join synchronization* represents the merging of two or more concurrent flows into a single outgoing flow. It is shown as a bar with two or more incoming transitions and one outgoing transition. Use the **MMCallControl Activity Diagram** figure as a reference.

To draw a join synchronization bar, follow these steps:

1. Click the Draw Join Sync Bar button on the **Drawing** toolbar.
2. Click below the **LocationUpdate** action element and above the **CallConfirm to cc_in** send action state element. Rational Rhapsody adds the join synchronization bar.
3. Click the Activity Flow button, and draw the following incoming transitions to the synchronization bar:
   a. From **LocationUpdate** press **Ctrl+Enter**.
   b. From **SignalOK**.
4. To change the line shape of a transition, right-click the line, select **Line Shape**, and then **Straight**, **Spline**, **Rectilinear**, or **Re-Route**.
5. Draw one outgoing transition from the synchronization bar to **CallConfirm to cc_in**, type `ChannelOpen` press **Ctrl+Enter**. This transition indicates that the channel is open and the call can be established. **ChannelOpen** corresponds to the trigger of the transition.
6. Draw a transition from **CallConfirm to cc_in** to **ConnectConfirm to cc_in**.
7. Draw a transition from **ConnectConfirm to cc_in** to **InCall**.
8. Change the line shape if you want.

9. Save your model. Your diagram should resemble the following figure:



Task 1j: Drawing a Timeout Transition

In this task, you are going to draw a timeout transition that monitors the signal strength of transmissions every three seconds. A *timeout transition* causes an object to transition after a specified amount of time has passed. It is an event with the form `tm(n)`, where *n* is the number of milliseconds the object should wait before making the transition. Use the **MMCallControl Activity Diagram** figure as a reference.

To draw a timeout transition, follow these steps:

1. Click the Activity Flow button on the **Drawing** toolbar.
2. Draw a transition originating and ending with **CheckSignal**.
3. Type `tm(3000)` press **Ctrl+Enter**.
4. Save your model.

Task 1k: Specifying an Action on a Transition

In this task, you are going to specify actions for **ChannelOpen**.

To specify an action, follow these steps:

1. Double-click the **ChannelOpen** transition, or right-click and select **Features**. The Features dialog box opens.
2. On the **General** tab, in the **Action** box, type the following code:

   ```
   locationUpdate();
   ```

3. Click **OK** to apply the changes and close the Features dialog box. Rational Rhapsody displays the transition name with the action command.
4. Save your model.

Note

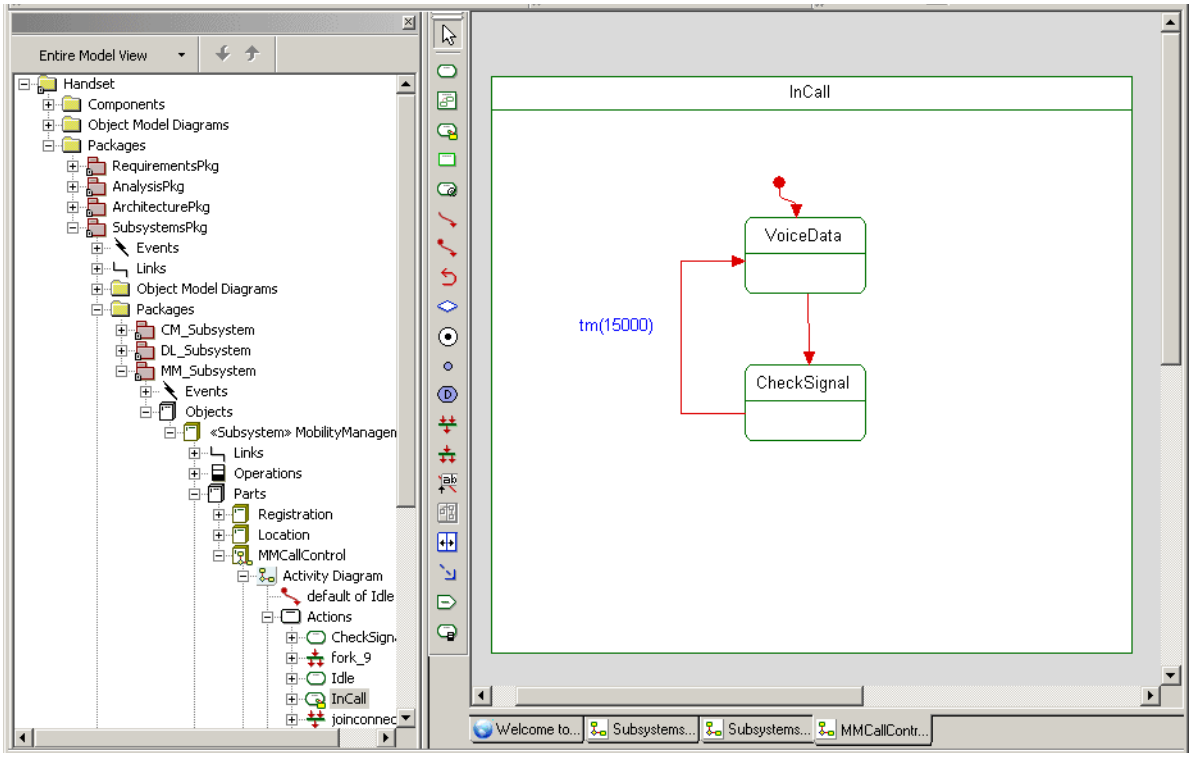
To display the transition name without the action, you can type the transition name as the label using the Features dialog box. Then right-click the transition and select **Display Options** to open the Display Options dialog box, and from the **Display Name** group, select **Label** click **OK**.

You have completed drawing the MMCall Control diagram. It should resemble the **MMCallControl Activity Diagram** figure. Rational Rhapsody automatically adds the action elements and transitions to the **MMCallControl** part in the browser.

Exercise 2: Creating the InCall Subactivity Diagram

Subactivities represent nested activity diagrams. The InCall subactivity diagram shows the flow of information once the system connects the call. The system monitors the signal strength for voice data every 15 seconds. The following figure shows the InCall subactivity diagram that you are going to create in this exercise.

InCall Subactivity Diagram



Task 2a: Creating the InCall Subactivity Diagram

To create the InCall subactivity diagram, follow these steps:

1. Right-click **InCall** in the MMCallControl activity diagram.
2. Select **Open Sub Activity Diagram**.

Rational Rhapsody displays the subactivity diagram with the **InCall** activity in the drawing area.

Note

After you have created the subactivity diagram, you can open it through the Rational Rhapsody browser too: Expand the **MM_Subsystem** package, the **<<Subsystem>> MobilityManagement** object, the **MMCallControl** part, the **Activity Diagram**, and the **Actions** category right-click **InCall** and select **Open nested Activity Diagram**.

Task 2b: Drawing Action Elements

In this task, you are going to draw the following action elements, and then add names to them. Use the **InCall Subactivity Diagram** figure as a reference.

- **VoiceData** to process voice data
- **CheckSignal** to check the signal strength on the network

To draw the action elements, follow these steps:

1. Click the Action button on the **Drawing** toolbar and click in the top half of the **InCall** action element press **Ctrl+Enter**.
2. Double-click the action element to open the Features dialog box, and on the **General** tab, in the **Name** box, type `VoiceData` click **OK**.
3. Add an action element to the bottom section of the **InCall** action element.
4. Open the Features dialog box, and on the **General** tab, in the **Name** box, type `CheckSignal` click **OK**.
5. For each action element, set the display options to **Name** to show the name on the diagram. Select both action elements, right-click, select **Display Options** to open the Display Options dialog box select **Name** from the **Display Name** group, and click **OK**.

Task 2c: Drawing a Default Flow

In this task, you are going to draw a default flow. The subactivity diagram must have an initial action element. Execution begins with the initial action element when an input transition to the subactivity action element is triggered. Use the **InCall Subactivity Diagram** figure as a reference.

To draw the default flow, follow these steps:

1. Click the Default Flow button on the **Drawing** toolbar.
2. Click above **VoiceData**, then click **VoiceData**. Press Ctrl+Enter.

Task 2d: Drawing Transitions

In this task, you are going to draw a transition between **VoiceData** and **CheckSignal**. Use the **InCall Subactivity Diagram** figure as a reference.

To draw transitions, follow these steps:

1. Click the Activity Flow button on the **Drawing** toolbar.
2. Draw a transition from **VoiceData** to **CheckSignal**. Press **Ctrl+Enter**.

Task 2e: Drawing a Timeout Transition

In this task, you are going to draw a timeout transition to check for voice data every 15 seconds. Use the **InCall Subactivity Diagram** figure as a reference.

To draw a timeout transition, follow these steps:

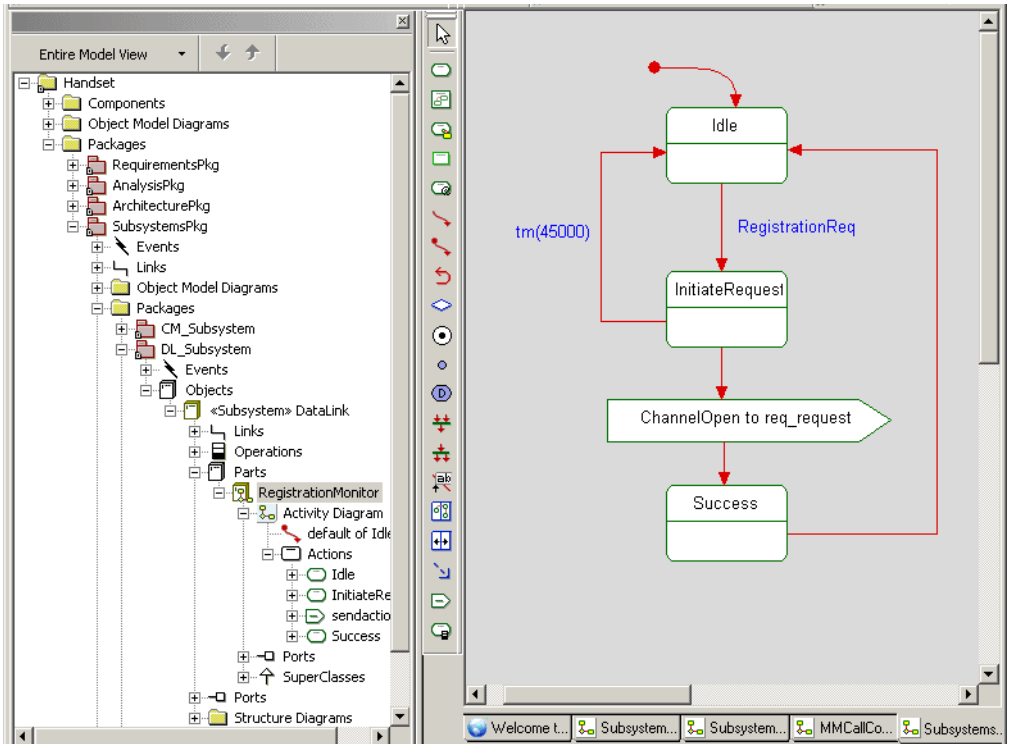
1. Click the Activity Flow button on the **Drawing** toolbar.
2. Draw a transition from **CheckSignal** to **VoiceData**.
3. Type `tm(15000)` press **Ctrl+Enter**.
4. To change the line shape of a transition, right-click the line, select **Line Shape**, and then **Straight**, **Spline**, **Rectilinear**, or **Re-Route**.
5. Save your model.

You have completed drawing the **InCall** subactivity diagram. It should resemble the **InCall Subactivity Diagram** figure. Rational Rhapsody automatically adds the newly created action elements and transitions to the browser.

Exercise 3: Creating the RegistrationMonitor Activity Diagram

The RegistrationMonitor activity diagram shows the functional flow of network registration requests. The system checks for registration requests and then sends received requests to the network. The following figure shows the RegistrationMonitor activity diagram that you are going to create in this exercise.

RegistrationMonitor Activity Diagram



Task 3a: Creating the RegistrationMonitor Activity Diagram

To create the RegistrationMonitor activity diagram, do either of the following:

- In the Rational Rhapsody browser, expand the **DL_Subsystem** package, the **<<Subsystem>> DataLink** object, and the **Parts** category. Right-click **RegistrationMonitor** and select **Add New > Activity Diagram**.

  or

- Open the Data Link structure diagram. Right-click **RegistrationMonitor** and select **New Activity Diagram**.

Rational Rhapsody adds the **Activity Diagram** category and the new activity diagram to the **RegistrationMonitor** part in the Rational Rhapsody browser, and opens the new activity diagram in the drawing area.

Task 3b: Drawing Action Elements

In this task, you are going to draw three action elements and then add names to the action elements. Use the **RegistrationMonitor Activity Diagram** figure as a reference.

To draw action elements, follow these steps:

1. Click the Action button on the **Drawing** toolbar.
2. In the upper section of the drawing window, create an action element press **Ctrl+Enter**.
3. Open the Features dialog box for this action element, in the **Name** box, type `Idle`. Click **OK**.
4. Repeat the previous steps but create these additional action elements where noted:
   a. Below **Idle**, with a name of `InitiateRequest`.
   b. Below **InitiateRequest**, with a name of `Success`.
5. For each action element, set the display options to **Name** to show the name on the diagram. Select all the action elements, right-click, select **Display Options** to open the Display Options dialog box, from the **Display Name** group, select **Name** click **OK**.

Task 3c: Drawing a Send Action State

As you did for the MMCallControl activity diagram in **Exercise 1: Creating the MMCallControl Activity Diagram**, you are going to create a send action state to represent the sending of an event. Use the **RegistrationMonitor Activity Diagram** figure as a reference.

To draw a send action state, follow these steps:

1. Click the Send Action State button on the **Drawing** toolbar.
2. Click between **InitiateRequest** and **Success**.
3. Double-click the Send Action State element on the diagram. The Features dialog box opens.
4. On the General tab, in the Target drop-down list, select **reg_request in SubsystemsPkg**.
5. In the **Event** drop-down list, select **ChannelOpen in SubsystemsPkg**.

   This command sends an asynchronous message out the **reg_request** port when the channel is open.
6. Click **OK**.

Task 3d: Drawing a Default Flow

In this task, you are going to draw a default flow. Use the **RegistrationMonitor Activity Diagram** figure as a reference.

To draw a default flow, follow these steps:

1. Click the Default Flow button on the Drawing toolbar.
2. Click above **Idle** click **Idle**. Press **Ctrl+Enter**.

Task 3e: Drawing Transitions

In this task, you are going to draw transitions between action elements. Use the **RegistrationMonitor Activity Diagram** figure as a reference.

To draw transitions, follow these steps:

1. Click the Activity Flow button on the **Drawing** toolbar.
2. Draw a transition from **Idle** to **InitiateRequest**. Type `RegistrationReq` press **Ctrl+Enter**.
3. Click the Activity Flow button for each of these transitions:
   - From **InitiateRequest** to **ChannelOpen to reg_request** press **Ctrl+Enter**.
   - From **ChannelOpen to reg_request** to **Success**.
   - From **Success** to **Idle**.
4. To change the line shape of a transition, right-click the line, select **Line Shape**, and then **Straight**, **Spline**, **Rectilinear**, or **Re-Route**.

Task 3f: Drawing a Timeout Transition

In this task, you are going to draw a timeout transition to return to the **Idle** action element after 45 seconds if no response is received from the network. Use the **RegistrationMonitor Activity Diagram** figure as a reference.

To draw a timeout transition, follow these steps:

1. Click the Activity Flow button on the **Drawing** toolbar.
2. Draw a transition from **InitiateRequest** to **Idle**.
3. Type the transition label `tm(45000)` press **Ctrl+Enter**.
4. Change the line shape if you want.
5. Save your model.

You have completed drawing the RegistrationMonitor diagram. It should resemble the **RegistrationMonitor Activity Diagram** figure. Rational Rhapsody automatically adds the newly created action elements and transitions to the **RegistrationMonitor** part in the Rational Rhapsody browser.

Exercise 4: Animating the MMCall Control Activity Diagram

As mentioned in the previous lesson, as a model gets more and more complicated, it is a good practice to stop and validate the model periodically and provide design-level debugging. In this task, you are going to regenerate the code, rebuild the model, and animate the MMCall Control activity diagram.

Animated activity diagrams show how states transition to other states while the model is executing.

Note

You must have completed **Lesson 4: Generating Code and Building Your Model** and **Lesson 5: Creating Sequence Diagrams** before you perform this task. In working through tasks in these previous lessons, you set up the **Simulate** component and the **Debug** configuration, and you made settings necessary for animation.

Task 4a: Regenerating Code and Rebuilding Your Model

Because you created activity diagrams in this lesson, you must regenerate code and rebuild your model before you do anything.

To regenerated code and rebuild your model, follow these steps:

1. Make sure **Debug** is your active configuration. It should appear in boldtype in the browser when it is set as the active configuration. If needed, in the Rational Rhapsody browser, right-click the **Debug** configuration select **Set as Active Configuration**.

   Note: If you have more than one configuration, you can also select the active configuration from the drop-down list on the **Code** toolbar.

2. If you have many diagrams open, you might find it less confusing to close them.

3. If the Output window is already open and there is information on the Build tab, to ensure that you will only be looking at information for the latest code generation/build, right-click on the tab select **Clear**. You might want to do this if information from a previous generation/build is still there.

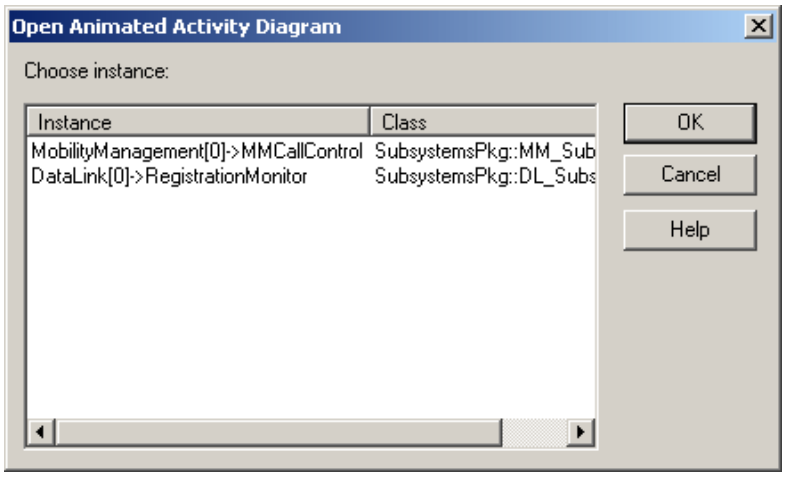
4. Select **Code > Re Generate > Debug**. If applicable, fix any errors noted on the Build tab of the Output window.

5. Select **Code > Rebuild Simulate.exe**. If applicable, fix any errors noted on the Build tab.

Task 4b: Animating the MMCall Control Activity Diagram

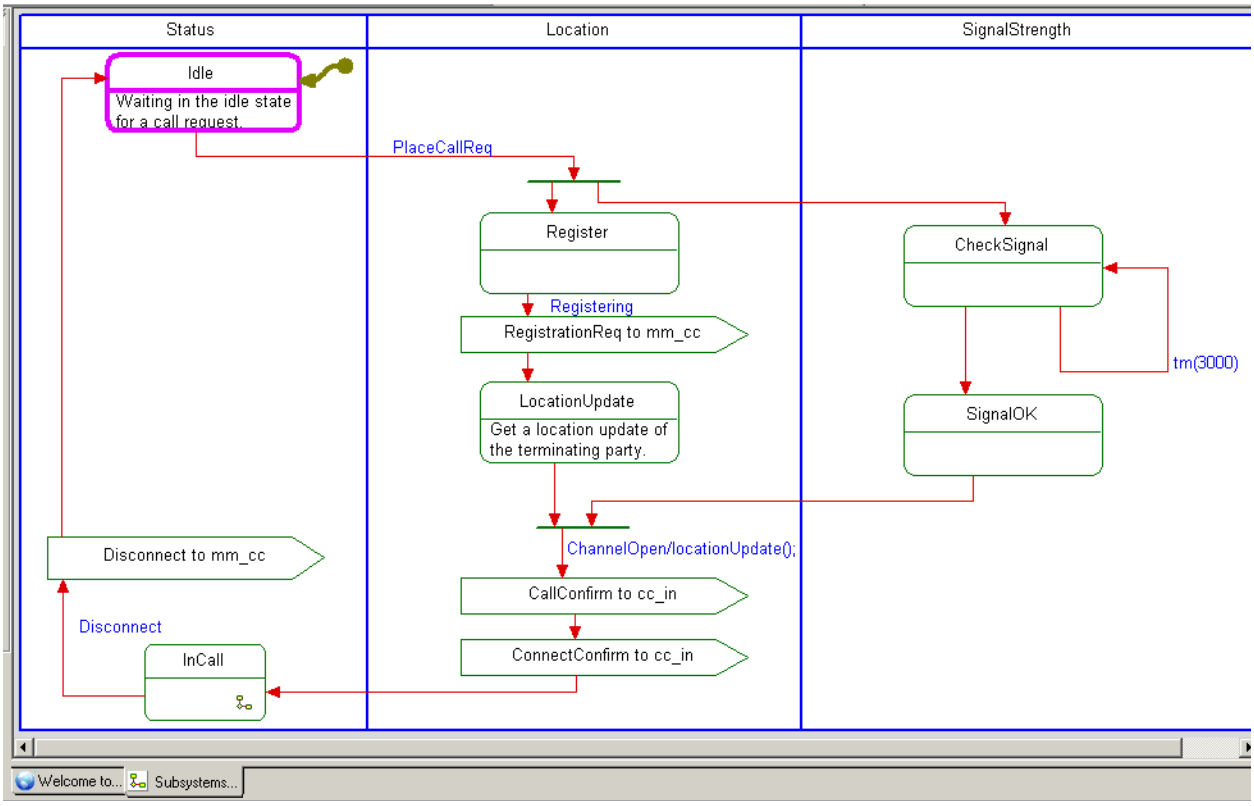
To animate the MMCall Control activity diagram, follow these steps:

1. Start animation:
   - Select **Code > Run Simulate.exe**, or
   - Click the Run Executable button .
2. Select **Tools > Animated Activity diagram**. The Open Animated Activity Diagram dialog box displays, as shown in the following figure.



3. Select **MobilityManagement[0]>MMCallControl** click **OK**.

4. Click the Go button on the **Animation** toolbar. Rational Rhapsody displays an animated version of your activity diagram, as shown in the following figure. Rational Rhapsody highlights **Idle** in magenta because it is active, while olive green shows what is inactive.



5. End the animation when you are done. If necessary, see **Task 4f: Quitting Animation**.

Summary

In this lesson, you created activity diagrams and a subactivity diagram, which show the functional flow of placing a call and registering users. You became familiar with the parts of an activity diagram and created the following:

- Swimlanes
- Action elements
- Send action states
- Subactivity diagram
- Default flows
- Transitions and timeout transitions
- Fork synchronization bar and join synchronization bar

You also regenerated code and rebuilt your model, and then you animated an activity diagram.

You are now ready to proceed to the next lesson, where you are going to identify the action element-based behavior when the system receives call requests and connects calls using a statechart.

Lesson 7: Creating Statecharts

Statecharts (SCs) define the behavior of classifiers (actors, use cases, or classes), objects, including the states that they can enter over their lifetime and the messages, events, or operations that cause them to transition from state to state.

Statecharts are a key animation tool used to verify the functional flow and moding. Statecharts can be animated to view the design level of abstraction and graphically show dynamic behavior.

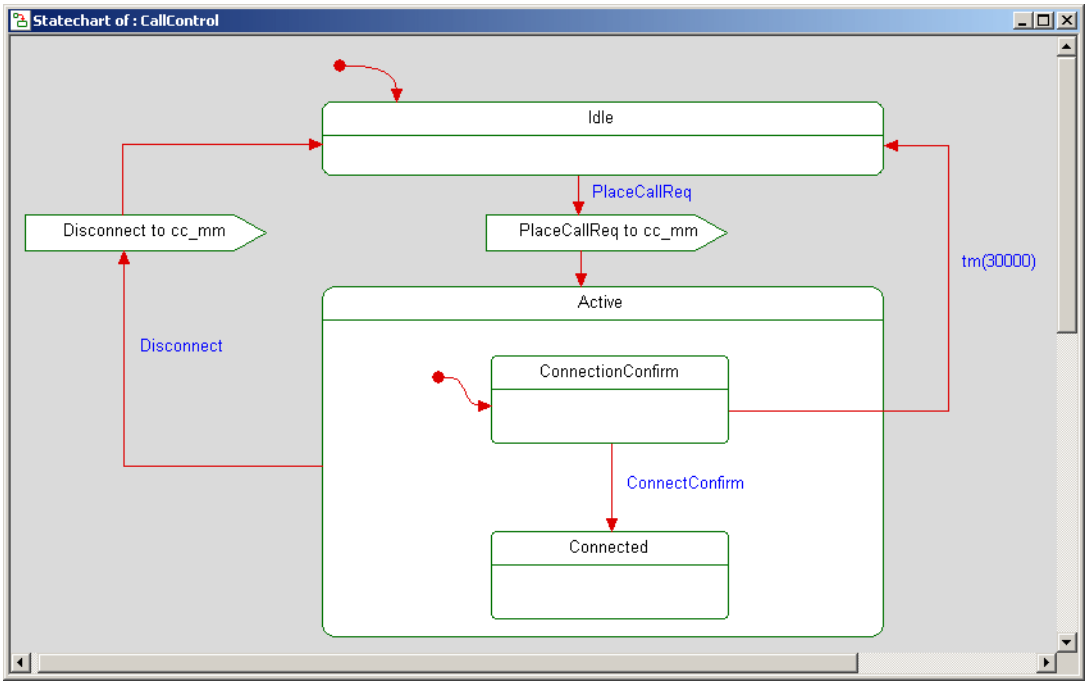
Goals for this Lesson

In this lesson, you are going to create the CallControl statechart to identify the state-based behavior when the system receives call requests and connects calls.

Exercise 1: Creating the CallControl Statechart

Statecharts define state-based behavior. The following figure shows the CallControl statechart that you are going to create in this exercise.

CallControl Statechart



Task 1a: Creating the CallControl Statechart

The CallControl statechart identifies the state-based behavior of instances of CallControl when the system receives call requests from users and connects calls. **CallControl** waits for an incoming call in the **Idle** state. When an incoming call is received, it forwards the message. If it does not receive a confirmation from the network in thirty seconds, it returns to the **Idle** state. If it receives a confirmation, the call connects, and remains connected until it receives a message to disconnect.

You draw statecharts using the following general steps:

1. Draw states and nested states.
2. Draw default connectors.
3. Draw send action states.
4. Draw transitions and specify actions on transitions.
5. Draw timeout transitions.

The following tasks describe these steps in detail.

To create a statechart, follow these steps:

1. Start Rational Rhapsody and open the handset model if they are not already open.
2. In the Rational Rhapsody browser, expand the **SubsystemsPkg** package, the **CM_Subsystem** package, the **ConnectionManagement** object, and the **Parts** category. Right-click **CallControl** and select **Add New > Statechart**.

   or

   Open the Connection Management structure diagram. Right-click **CallControl** and select **New Statechart**.

Rational Rhapsody adds the **Statechart** category and the new statechart to the **CallControl** part in the browser, and opens the new statechart in the drawing area.

Note

Once you create a statechart, you can open it using the **Diagrams** toolbar.

Task 1b: Drawing States

In this task, you are going to draw two states, **Idle** and **Active**. A *state* is a graphical representation of the status of an object. It typically reflects a certain set of its internal data (attributes) and relations. Use the **CallControl Statechart** figure as a reference.

To draw a state, follow these steps:

1. Click the State button on the **Drawing** toolbar click on the top section of the drawing area. (You can also use click-and-drag.) Rational Rhapsody create a state with a default name of **state_*n***, where ***n*** is equal to or greater than 0.
2. Type `Idle` press **Enter**. This state indicates that no call is in progress.
3. Draw a larger state named `Active` in the center of the drawing area. This state indicates that the call is being set up or is in progress.

Task 1c: Drawing Nested States

In this task, you are going to draw the following states nested inside the **Active** state.

- **ConnectionConfirm** to wait for a connection and then confirms the connection
- **Connected** to connect as a voice or data call

Use the **CallControl Statechart** figure as reference.

To draw nested states, follow these steps:

1. Click the State button on the **Drawing** toolbar.
2. In the top half of the **Active** state, draw a state named `ConnectionConfirm`.
3. In the bottom half of the **Active** state, draw a state named `Connected`.

Task 1d: Drawing Default Connectors

One of an object's states must be the default state, that is, the state in which the object finds itself when it is first instantiated. **Idle** is in the default state as it waits for call requests, and **Active** is in the default state before it confirms the connection. Use the **CallControl Statechart** figure as a reference.

To draw default connectors, follow these steps:

1. Click the Default Connector button on the **Drawing** toolbar.
2. Click to the upper left of the **Idle** state, click **Idle** press **Ctrl+Enter**.
3. Draw a default connector to **ConnectionConfirm** press **Ctrl+Enter**.

Task 1e: Drawing Send Action States

As mentioned in **Task 1f: Drawing Send Action States** in the previous section on activity diagrams, the Send Action State element can be used to represent the sending of events to external entities.

To draw a send action state, follow these steps:

1. Click the Send Action State button on the **Drawing** toolbar.
2. Using the **CallControl Statechart** figure as a reference, click to the left of the **Idle** and **Active** states.
3. Double-click the Send Action element on the diagram. The Features dialog box opens.
4. On the **General** tab, in the **Target** drop-down list, select **cc_mm in SubsystemsPkg**.
5. In the **Event** drop-down list, select **Disconnect in SubsystemsPkg**.

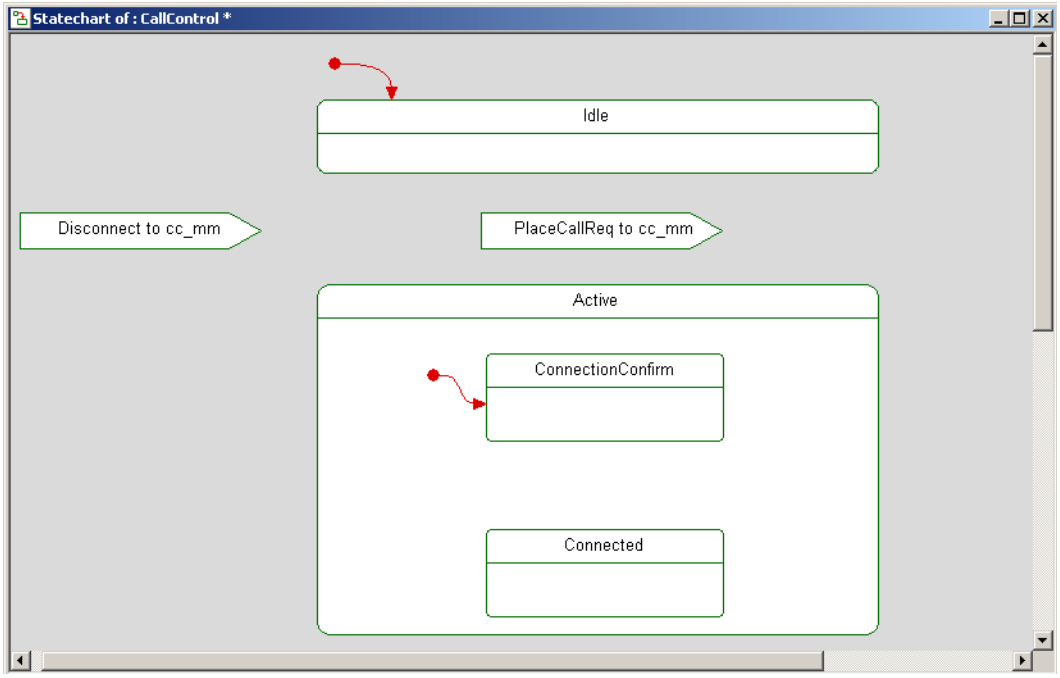
   This command sends an asynchronous message out the **cc_mm** port when disconnecting.
6. Click **OK** to close the Features dialog box for the send action.
7. Draw another Send Action State between the **Idle** and **Active** states: For the **Target**, select **cc_mm in SubsystemsPkg**; for the **Event**, select **PlaceCallReq in SubsystemsPkg**.

   This command sends an asynchronous message out the **cc_mm** port when placing a call.

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

9. Save your statechart.

Your statechart should resemble the following figure:



Task 1f: Drawing Transitions

In this task, you are going to draw transitions with triggers. *Transitions* represent the response to a message in a given state. They show what the next state is going to be. A transition can have an optional trigger, guard, or action. Use the **CallControl Statechart** figure as a reference.

To draw transitions, follow these steps:

1. Click the Transition button on the **Drawing** toolbar.
2. Click the **Idle** state and then click the **PlaceCallReq to cc_mm** send action state.
3. In the label box, type `PlaceCallReq` press **Ctrl+Enter**.
4. Create a transition from **PlaceCallReq to cc_mm** to **Active**.
5. Create a transition from **ConnectionConfirm** to **Connected** named `ConnectConfirm`.
6. Create a transition from the **Active** state to the **Disconnect to cc_mm** send action state named `Disconnect`. This transition indicates that the user has disconnected or the network has terminated the call.
7. Create a transition from the **Disconnect to cc_mm** send action state to **Idle**.

Note

To change the line shape, right-click the line, select **Line Shape**, and then **Straight**, **Spline**, **Rectilinear**, or **Re-Route**.

Task 1g: Drawing a Timeout Transition

In this task, you are going to draw a timeout transition in which **ConnectionConfirm** waits thirty seconds before returning to the **Idle** state if a connect confirmation is not made. A *timeout transition* causes an object to transition to the next state after a specified amount of time has passed. It is an event with the form `tm(n)`, where *n* is the number of milliseconds the object should wait before making the transition. Use the **CallControl Statechart** figure as a reference.

To draw a timeout transition, follow these steps:

1. Click the Transition button on the **Drawing** toolbar.
2. Draw a transition from **ConnectionConfirm** to **Idle**.
3. Type `tm(30000)` press **Ctrl+Enter**.
4. Save your statechart.

You have completed drawing the `CallControl` statechart. It should resemble the **CallControl Statechart** figure. Rational Rhapsody automatically adds the newly created states and transitions to the **CallControl** part in the browser.

Exercise 2: Animating the CallControl Statechart

As mentioned in earlier lessons, as a model gets more and more complicated, it is a good practice to stop and validate the model periodically and provide design-level debugging. In this task, you are going to regenerate the code, rebuild the model, and animate the CallControl statechart.

Animated statecharts show how states transition to other states while the model is executing.

Note

You must have completed **Lesson 4: Generating Code and Building Your Model** and **Lesson 5: Creating Sequence Diagrams** before you perform this task. In working through tasks in these previous lessons, you set up the **Simulate** component and the **Debug** configuration, and you made settings necessary for animation.

Task 2a: Regenerating Code and Rebuilding the Model

To do this task, follow these steps:

1. Make sure **Debug** is your active configuration. It should appear in boldtype in the browser when it is set as the active configuration. If needed, in the Rational Rhapsody browser, right-click the **Debug** configuration select **Set as Active Configuration**.

   Note: If you have more than one configuration, you can also select the active configuration from the drop-down list on the **Code** toolbar.

2. If you have many diagrams open, you might find it less confusing to close them.

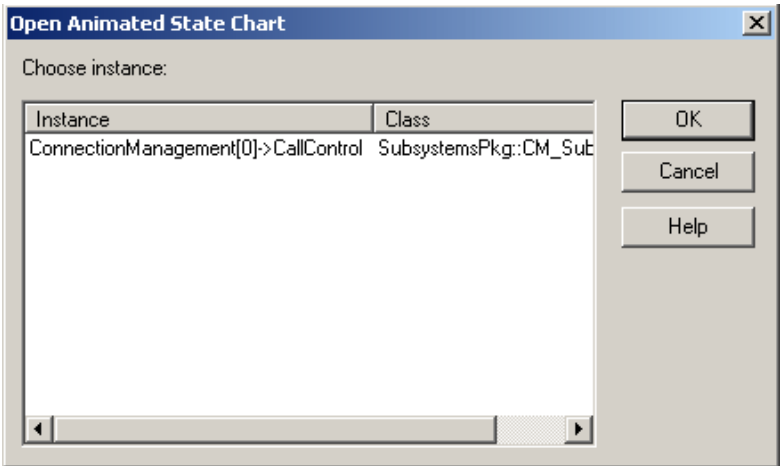
3. If the Output window is already open and there is information on the Build tab, to ensure that you will only be looking at information for the latest code generation/build, right-click on the tab select **Clear**. You might want to do this if information from a previous generation/build is still there.

4. Select **Code > Re Generate > Debug**. If applicable, fix any errors noted on the Build tab of the Output window.

5. Select **Code > Rebuild Simulate.exe**. If applicable, fix any errors noted on the Build tab.

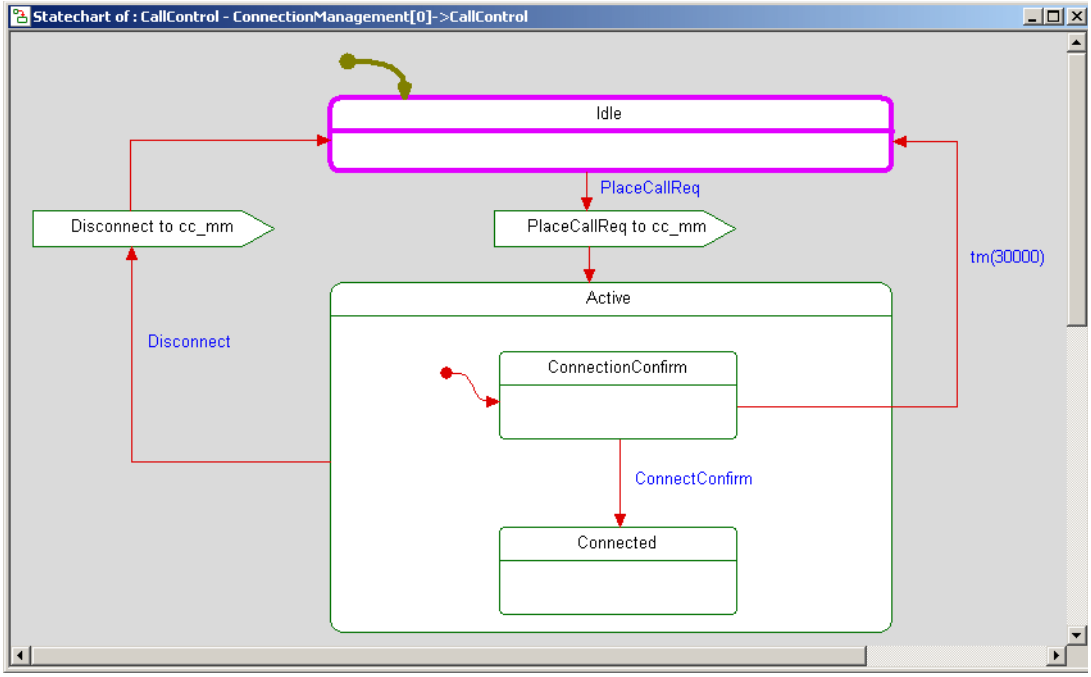
Task 2b: Animating the CallControl Statechart

1. Start animation:
   - Select Code > Run Simulate.exe, or
   - Click the Run Executable button .
2. Select **Tools > Animated Statechart**. The Open Animated State Chart dialog box displays, as shown in the following figure.



3. Select **ConnectionManagement[0]->CallControl** click **OK**. Rational Rhapsody displays an animated version of your statechart.

4. Click the Go button on the **Animation** toolbar. Rational Rhapsody displays an animated version of your activity diagram, as shown in the following figure. Rational Rhapsody highlights **Idle** in magenta because it is active, while olive green shows what is inactive.



5. End the animation when you are done. If necessary, see **Task 4f: Quitting Animation**.

Summary

In this lesson, you created a statechart, which identifies the state-based behavior when the system receives call requests and connects calls. You became familiar with the parts of a statechart and created the following:

- States and nested states
- Default connectors
- Send action states
- Transitions and timeout transitions

You also regenerated code and rebuilt your model, and then you animated your statechart.

You have completed the handset model. You are now ready to proceed to the next lesson, where you learn more about animation, including sending events to your model.

Lesson 8: More Animation

Animation is the observable execution of behaviors and associated definitions in the model. Rational Rhapsody animates the model by executing the code generated, with instrumentation, for classes, operations, and associations. Once you start model animation, you can open animated diagrams, which let you observe the model as it is running and perform design-level debugging. You can step through the model, set and clear breakpoints, inject events, and generate an output trace.

It is good practice to test the model incrementally using model execution, which you have practiced in earlier lessons. You can animate pieces of the model as it is developed. This gives you the opportunity to determine whether the model meets the requirements and find defects early on. Then you can test the entire model. In this way, you iteratively build the model, and then with each iteration perform an entire model validation.

In the previous lessons you animated a sequence diagram, activity diagram, and a statechart individually. Now that you have completed designing your model so that all your model elements are in place, you can view a fuller animation sequence for your handset model.

Goals for this Lesson

In the previous lessons on sequence diagrams, activity diagrams, and statecharts you learned about animation and you animated these diagrams. In this lesson, you are going to send events to your model and view this in animation.

Exercise 1: Animating Your Diagrams

This tutorial assumes that you have done the lessons in order in this tutorial.

Before you can animate your model, you have to generate code and build your model, which you learned how to do in **Lesson 4: Generating Code and Building Your Model**. You also learned how to set up for animation and run animation as part of **Lesson 5: Creating Sequence Diagrams** (specifically in **Exercise 4: Animating a Sequence Diagram**).

Task 1a: Preparing for Animation

Before you do animation, regenerate your code and rebuild your model so that you know that you are working with the latest code and model. If necessary, see **Exercise 2: Animating the CallControl Statechart** from **Lesson 7: Creating Statecharts**.

Task 1b: Animating Your Diagrams

Open the following diagrams and animate them:

1. Connection Management Place Call Request Success sequence diagram
   (select **Tools > Animated Sequence Diagram**). If necessary, see **Exercise 4: Animating a Sequence Diagram**. Once you animate this diagram, the other diagrams will be animated once you open them.

2. MMCallControl activity diagram
   (select **Tools > Animated Activity Diagram**). If necessary, see **Exercise 4: Animating the MMCall Control Activity Diagram**.

3. ConnectionManagement>CallControl statechart
   (select **Tools > Animated Statechart**). If necessary, see **Exercise 2: Animating the CallControl Statechart**.

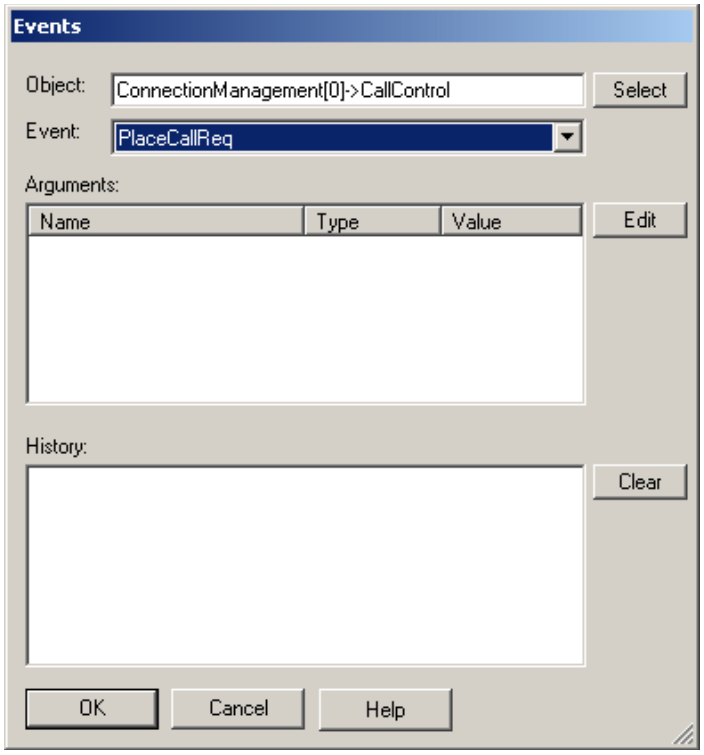
Exercise 2: Sending Events to Your Model

You can inject events in an animated diagram to see how the model reacts. In this exercise, you are going to generate an event in the animated statechart and view the resulting behavior in the animated statechart, animated sequence diagram, and animated activity diagram. You also get to send the Disconnect event to your model

Task 2a: Sending an Event to Your Model

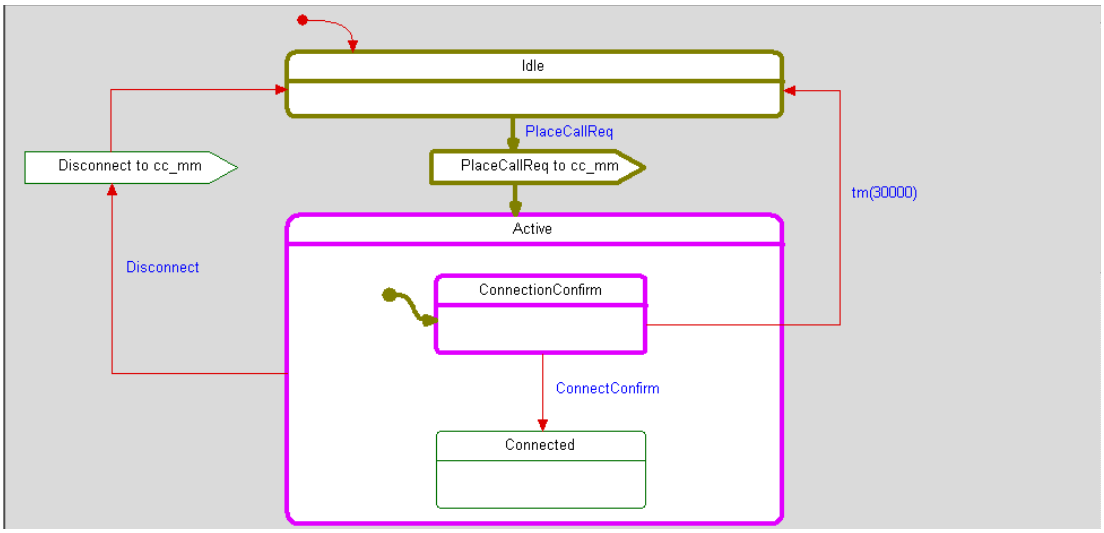
To send an event to your model, follow these steps:

1. In the animated CallControl statechart, right-click Idle and select **Generate Event**. The Events dialog box opens.
2. From the **Event** drop-down list box, select **PlaceCallReq**, as shown in the following figure.

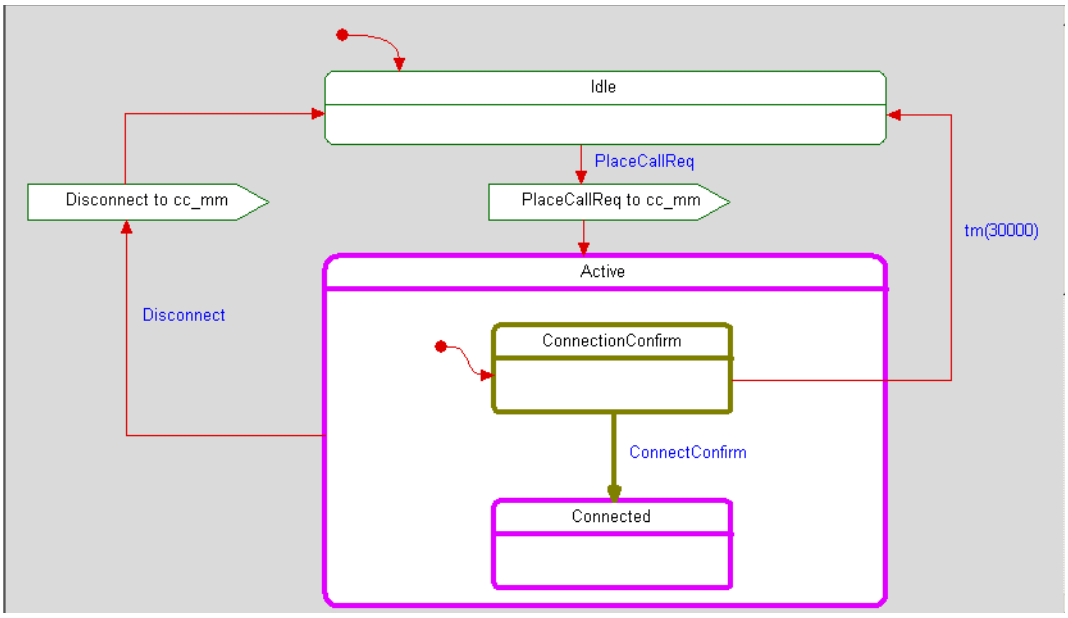


3. Click **OK** to close the dialog box.

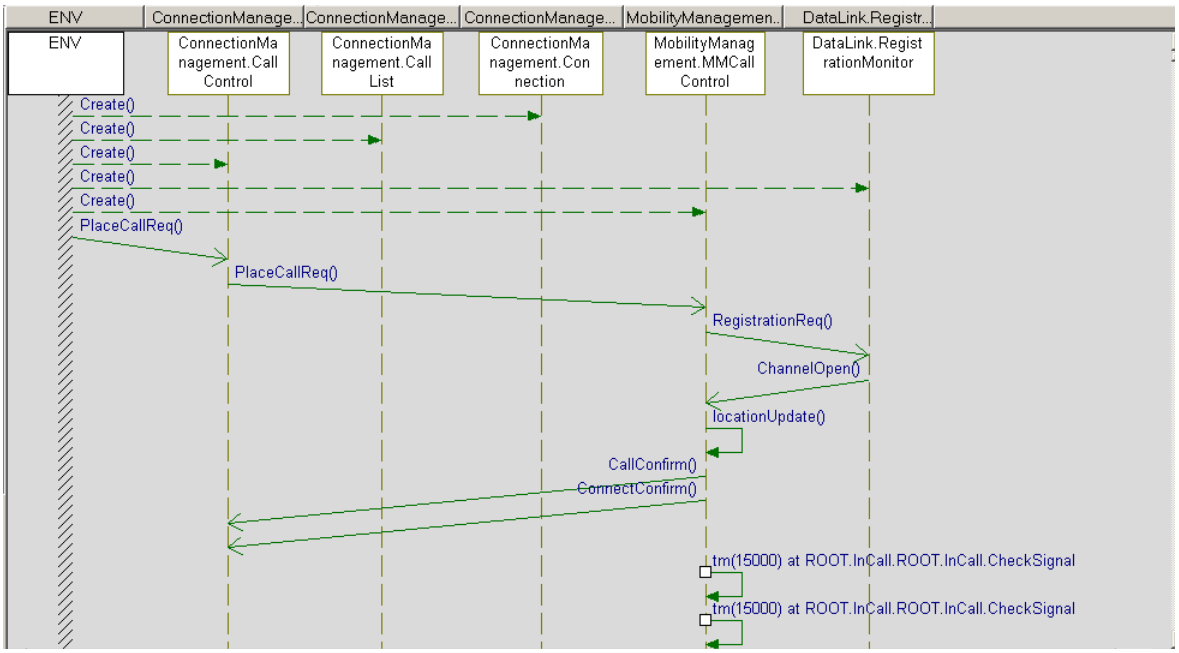
In the animated statechart, **Idle** and **PlaceCallReq** becomes inactive (olive), and **Active** and **ConnectionConfirm** become active (magenta), as shown in the following figure.



Then **ConnectionConfirm** and **ConnectConfirm** become inactive, and **Connected** becomes active, as shown in the following figure:

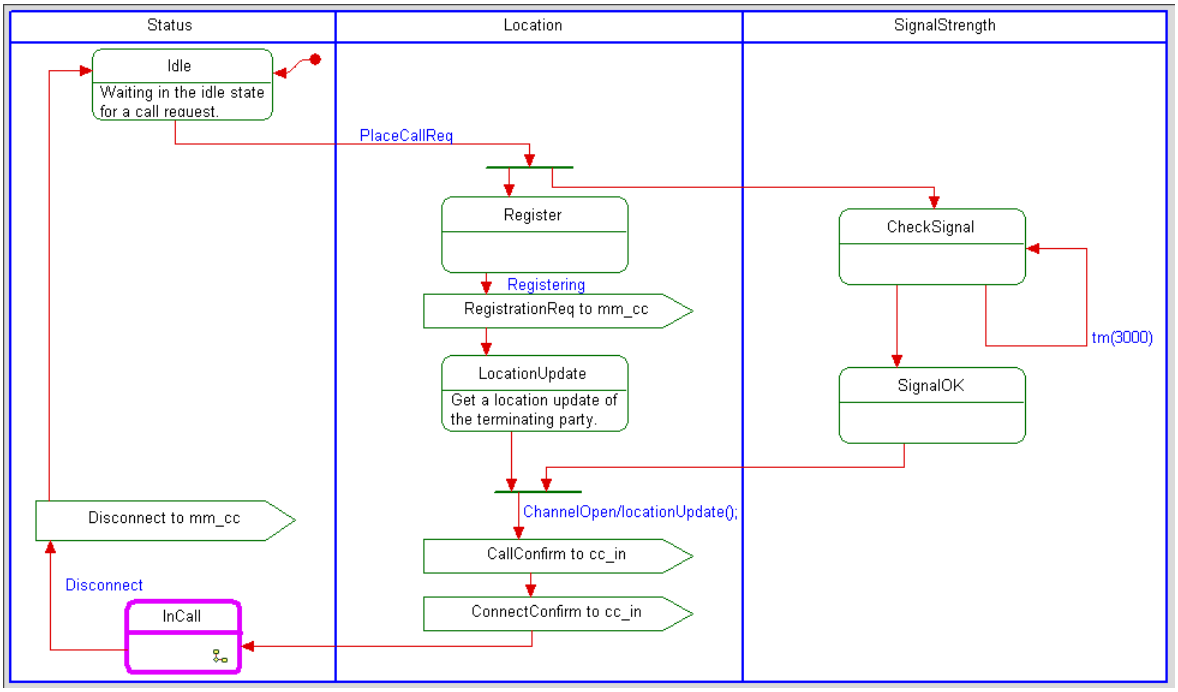


4. Switch to the animated ConnectionManagement Place Call Request Success sequence diagram. Rational Rhapsody dynamically displays how the instances pass messages, as shown in the following figure.



5. Switch to the animated MMCallControl activity diagram. **Idle** becomes inactive (olive). **Register** and **CheckSignal** become active (magenta), and then **LocationUpdate** becomes active.

   Then **LocationUpdate** becomes inactive, and **InCall** becomes active, as shown in the following figure:

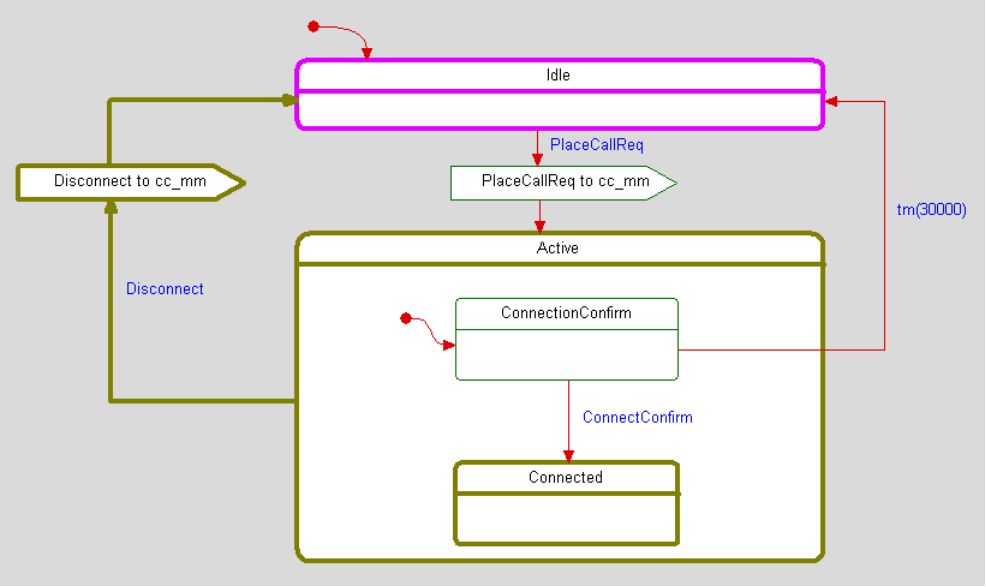


You can continue generating events and viewing the resulting behavior in the animated diagrams.

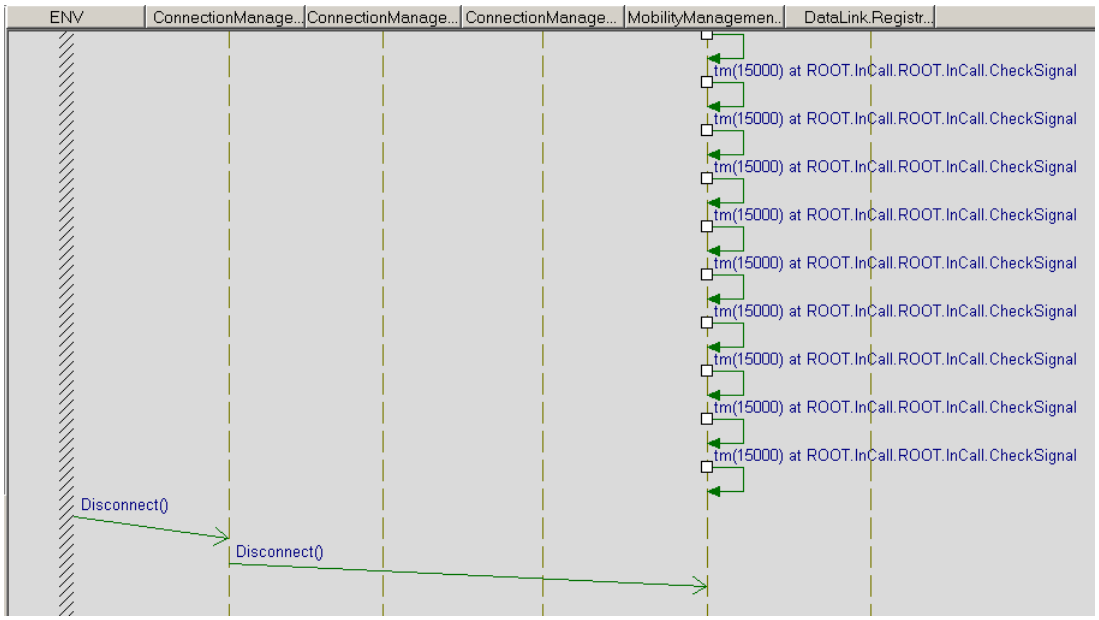
Task 2b: Sending Another Event

To send another event to your model, follow these steps:

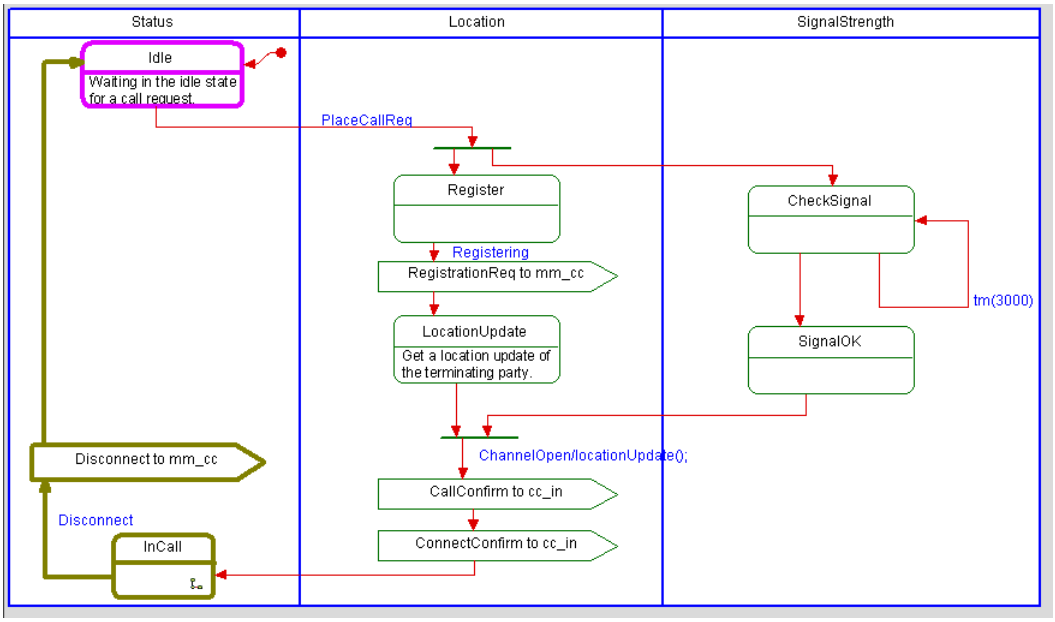
1. In the animated CallControl statechart, right-click **Idle** and select **Generate Event**.
2. Select **Disconnect** in the **Event** drop-down list box, and click **OK**.
3. View your statechart, which should show a transition to the **Idle** state (magenta) and **Active** and **Disconnect** become inactive (olive green), as shown in the following figure:



4. View your animated ConnectionManagement Place Call Request Success sequence diagram. Rational Rhapsody displays how the Disconnect message, as shown in the following figure:



5. Switch to the animated MMCallControl activity diagram, **Idle** transitions to the active state (magenta) and **InCall** becomes inactive (olive green).



Task 2c: Quitting Animation

To end the animation session, follow these steps:

1. Click the Animation Break button on the **Animation** toolbar click the Quit Animation button .
2. Click **Yes** to confirm ending the animation session.

The Output window displays the message `Animation session terminated`.

Note

When you close the project or an animated sequence diagram, Rational Rhapsody prompts whether or not you want to save the diagram. Saving an animated sequence diagram is useful in order to compare the results of the current session to those of different execution scenarios.

Summary

In this lesson, you animated the model and sent events to the model and saw it progress through states and pass messages.

In the next section, you learn about Technical Support and documentation plus other useful information.

Index

Symbols

_rpy file 11
_RTC directory 11

A

Action element 155
Active configuration 115
Activity diagram 5, 149
  action element 155
  animating 174
  creating 149
  default flow 157
  fork sync bar 162
  InCall 166
  join sync bar 163
  MMCallControl 152
  opening subactivity diagram 167
  RegistrationMonitor 170
  subactivity 157
  swimlanes 153
  timeout transition 165
  transition 160
Actor 31
  associating with use cases 40
  line 123
Actor lines 123
Analysis mode 122
Anchor 43
Animation 139, 191
  activity diagram 174
  browser 145
  Call Stack window 142
  configuration 114
  Event Queue window 142
  generating code 115
  injecting events 193
  output panes 142
  quitting 147, 200
  sending events 193
  sequence diagram 143
  starting 141
  statechart 188
Association 40
Autosave 12

B

Backup 12
Behavioral port 76
Black-box analysis 29
Boundary box 35
Browser 22
Building the model 117

C

C++ language 1
  case-sensitivity 19
Call stack 142
Case-sensitivity 19
Categories 22
Classifier roles 124, 135
Code 1
Code generation 115
  debugging 116, 118
  source files 116
Collaboration diagram 5
Comment 43
Compilers 140
Component 112
  creating 112
  creating configuration 114
  default description 112
  features 113
Component diagram 5
Configuration 114
  creating animation 114
  Debug 115
  default 114
Connection Management Place Call Request Success
  sequence diagram 133
Connection Management structure diagram 91
Constraint 43
Contract-based port 82
Creating
  activity diagram 149
  animation configuration 114
  component 112
  handset project 7
  object model diagram 107
  sequence diagram 119

statechart 179
structure diagram 65
use case diagram 31

D

Data Call Requirements use case diagram 58
Data Link structure diagram 96
Debug configuration 115
Debugging 116, 118
Default component 112
Default configuration 114
Default connector 183
Default flow 157
Dependency 54
adding stereotype 56
Deployment diagram 5
Description tab 25
Design mode 122
Diagrams 5
activity 149
Connection Management Place Call Request Success 133
Connection Management structure diagram 91
Data Call Requirements 58
Data Link structure diagram 96
Functional Overview 32
Handset System structure diagram 66
InCall subactivity diagram 166
MM Architecture structure diagram 100
MMCallControl 152
NetworkConnect 129
object model 105
Place Call Overview 45
Place Call Request Successful 121
RegistrationMonitor 170
sequence 119
statechart 179
Subsystem Architecture object model diagram 106
UML 5
use case 31
Display options 53
Docking the Features dialog box 28
Domains 13
Drawing area 23
Drawing toolbar 23

E

ehl file 11
Elements
adding remarks 43
display options 53
external 116
labeling 160
organizing 88
Event 83, 125, 127, 132, 165, 187, 193
Event history file 11
Event Queue 142
Events, naming conventions 19
External elements 116

F

Features dialog box 24
Apply and OK buttons 24
Description tab 25
docking 28
General tab 25
keeping open 24
moving 28
Properties tab 27
Relations tab 26
requirement description 51
save all changes 48
tabs 25
Tags tab 26
Files 11
code generation 116
project 9
source 116
Flow 77
changing the direction 78
drawing 77
specifying flow items 79
Flow charts 5
Flow item 79
Folders 11
Fork sync bar 162
Fork synchronization 162
Functional Overview use case diagram 32

G

General tab 25
Generalization 42
Generated source files 116
Generating C++ code 1
Generating code for animation 115
Graphical user interface 20

H

Handset 2
activity diagram 149
animating 139, 191
creating 7
object model diagram 105
opening 18
sequence diagram 119
statechart 179
use case diagram 31
Handset System structure diagram 66, 67
Help pane for property 16

I

Implementation 13, 119
InCall subactivity diagram 166
Instance area 121
Instance line 124, 135
Interaction occurrence 128
Interface naming conventions 19
Interfaces 82

J

Join sync bar 163
Join synchronization 163

L

Labeling elements 160
Legacy code 116
Line shapes 81
Link 94, 110
Linux 6
Locate in Browser 127
Log files 11

M

Makefile 139
Message pane 121
Messages 125
MM Architecture structure diagram 100
MMCallControl activity diagram 152
Model building 117
Models
  naming conventions 19
Moving the Features dialog box 28

N

Names pane 121
Naming conventions 19
Nested state 182
NetworkConnect sequence diagram 129
Non-behavioral port 76
Noncontract-based port 82
Note 43

O

Object model diagram 5
  link 110
  Subsystem Architecture 106
Objects 69
  adding stereotype 70
  diagram 67
  drawing 72
Occurrence 128
Opening
  project 18
  Rational Rhapsody 6
Operations
  names 19
  naming conventions 19
Output window 23, 116, 118
  Call Stack 142
  Event Queue 142

P

Packages 13, 16, 22
  AnalysisPkg 13, 33, 113, 138
  ArchitecturePkg 13, 67, 89, 109, 113, 132
  RequirementsPkg 13, 50
  SubsystemsPkg 13, 88, 89, 107, 113, 122, 132, 151, 181
Place Call Overview use case diagram 45
Place Call Request Successful sequence diagram 121
Port 75
  behavioral 76
  changing the placement 93
  contract-based 82
  drawing 75
  non-behavioral 76
  noncontract-based 82
  rapid 82
  reversing 87
  specifying port contract 82
Port contract 82
Predefined types of packages 16
Profiles 8
Project
  creating 7
  files 11
  opening 18
  saving 12
Project files 9, 11
Project folder 22
Project node 22
Project profiles 7
Project subfolders 11
Project types 7
Properties 42
Properties tab 16, 27
Provided interfaces 82

Q

Quitting animation 147, 200

R

Rapid port 82
Rational Rhapsody
  autosave 12

backup 12
browser 22
closing 6
drawing area 23
Drawing toolbar 23
exiting 6
Features dialog box 24
GUI 20
interface 20
naming conventions 19
Output window 23
project profiles 7
project types 7
specialized profiles 7
starting 6
toolbars 21
UML diagrams 5
Rebuilding the application 116, 118
Rectilinear line 81
Regenerating code 116, 118
RegistrationMonitor activity diagram 170
Relations tab 26
Remarks 43
Repository directory 11
Required interfaces 82
Requirement 43
Requirements elements 50
Requirements traceability 50
Reverse engineering 1
rpy file 11

S

Send Action State 158, 172
Sending events
animation 193
Sequence diagram 5, 119
actor line 123
animating 143
classifier role 124, 135
Connection Management Place Call Request Success 133
creating 119
instance area 121
instance line 124
interaction occurrence 128
message 125
Message pane 121
Names pane 121
NetworkConnect 129
operation mode 122
Place Call Request Successful 121
system border 134
time interval 131
types of messages 125
Source files 116
Specialized profiles 8
Spline line 81
Stamp mode 75
State 182
Statechart 5, 179
animating 188
creating 179
default connector 183
nested state 182
state 182
timeout transition 187
transition 186
Stereotype 56
dependency 56
subsystem 70
Straight line 81
Structure diagram 5
Connection Management 91
creating 65
Data Link 96
flow 77
Handset Structure 67
link 94
MM Architecture 100
objects 69, 72
port 75
specifying flow items 79
Structure diagrams 65
Subactivity 157
Subactivity diagram 166, 167
Subfolders 11
Subsystem Architecture object model diagram 106
Subsystems 13
Swimlanes 153
System border 134

T

Tags tab 26
Time interval 131
Timeout transition
activity diagram 165
statechart 187
Toolbars 21, 23
Traceability 13, 50
Transition
activity diagram 160
specifying action 165
statechart 186
timeout 187
Troubleshooting
case-sensitivity 19
Types of profiles 8

U

UML (Unified Modeling Language) 1
Unit 12

Use case
associating with actors 40
drawing 37
features 39
Use case diagram 5, 31
boundary box 35
Data Call Requirements 58
dependencies 54
drawing 34
Functional Overview 32
Place Call Overview 45
requirements 52
use cases 37

V

vba file 11

W

White-box analysis 29
Windows 6