Rational Software Corporation

Java Reference

RATIONAL ROSE® REALTIME

VERSION: 2002.05.20

PART NUMBER: 800-025105-000

WINDOWS/UNIX

Rational® the software development company

support@rational.com
http://www.rational.com

Contents

Figures

Preface

This manual provides an introduction to Rational Rose RealTime Java. The Java module joins the current C and C++ modules to add the ability to design, generate, build, and debug applications in the Java language to the Rational Rose RealTime product.

Contents

This chapter is organized as follows:

Audience

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

Other Resources

- Online Help is available for Rational Rose RealTime.

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

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

- For more information on training opportunities, see the Rational University Web site: http://www.rational.com/university.

Contacting Rational Technical Publications

To send feedback about documentation for Rational products, please send e-mail to our Technical Documentation Department at techpubs@rational.com.

Contacting Rational Technical Support

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

Your Location	Telephone	Fax	E-mail
North America	(800) 433-5444 (toll free) (408) 863-4000 Cupertino, CA	(781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 (0) 20-4546-200 Netherlands	+31 (0) 20-4546-202 Netherlands	support@europe.rational.com
Asia Pacific	+61-2-9419-0111 Australia	+61-2-9419-0123 Australia	support@apac.rational.com

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

- Your name, telephone number, and company name
- Your computer's make and model
- Your computer's operating system and version number
- Product release number and serial number
- Your case ID number (if you are following up on a previously-reported problem)

Overview 1

Contents

This chapter is organized as follows:

Using this Guide

Use this guide to learn how to use Rational Rose RealTime Java to build, compile and debug Java based Rational Rose RealTime models. Additional information is given on how to deploy the model to a target system, and how to optimize and configure your target to fit your project's needs.

Using Rational Rose RealTime Java, you can produce Java source code, compile it, then run and debug your Java program using the information contained in a Rose RealTime model. The code generated for each selected model element is a function of the element's specification, model properties, and the model's design properties. Model properties provide the language-specific information required to map your model onto Java.

To understand how Rational Rose RealTime Java works, you need to understand the main parts of the language add-in that are discussed in this Chapter. In addition, there are a number of Java example models that demonstrate features of the toolset, the model properties, and the Java UML Services Library.

In this document, a property is identified by the type of the element and the tab where this property is available. For example, in "**Class File Header** property (**Class**, **RTJava**)", the property is available on the **RTJava** tab of the **Specification** dialog box for **Class** elements. In the **Options** dialog box, this property is available on the **RTJava** tab when the **Type** is set to **Class**.

Getting Started with Rational Rose RealTime Java

There is an expected sequence of work activities for taking a model from early prototyping to final production.

During the initial phases of model development, you probably want to run your models primarily on the host workstation. This keeps the modify-compile-debug cycle as short as possible. Also, you can take advantage of workstation-based debug tools, such as Java source-level debugger which may not be available on your target platform. For many projects, this is the final step, if you are using a workstation-based target.

The workflow of Rational Rose RealTime is intended to provide as much up-front verification and debugging as possible in the tool-rich environment of the host workstation. This environment is typically provided by a combination of Rose RealTime host-based tools and workstation-based Java tools. This leaves a minimal amount of debugging to do on the target, where debugging is typically more difficult. The use of target observability to monitor and control models at the model level greatly enhances the ability to debug target applications.

For further information on building and running Rational Rose RealTime Java models, see *Getting Started with Rational Rose RealTime Java* on page 23

Using Java Code in Models

Java is used as a detail-level coding language in Rational Rose RealTime. At a higher level of abstraction, the program is described both structurally and behaviorally as a graphical model using the Unified Modeling Language (UML). Java code can be added to a variety of behavioral elements in a UML model. The abstract behavior of a capsule is described as a graphical state diagram, which shows the allowable sequence of events that the capsule can process. In order to actually carry out less structured activity, detailed code must be added to the states, transitions, and operations in the model. There are no restrictions on the code that you enter into your model. You can also make use of external Java classes (that is, classes defined outside of Rational Rose RealTime) in your model.

Rational Rose RealTime is designed to be the central interface point for developing Java based models, and provides support for all activities in the development process, including requirements capture, high-level design, coding, versioning, loadbuilding, and testing. It does not, however, replace your existing Java tools. Rather, it depends on the existence of other tools to handle language-specific work. It coordinates and controls these activities in the context of your model. For example, the toolset does not include a Java compiler. Rational Rose RealTime requires that you already have a Java compiler installed and accessible in your environment prior to compiling a Java model.

Note: Rational Rose RealTime does not support Rational Rose Java models in this release.

For details on using Java code in models, see *User code segment methods* on page 61.

Code Generation

This section discusses some aspects of how a model is converted to Java code and compiled. This should clarify the output you will see in the **Build Log** window, and help you browse the generated code.

The Java generator uses the specifications and model properties of elements in the current model to produce Java source code. You generate code for a component which in turn references a set of elements from the logical view. The location of the source files that are generated for elements referenced by (or assigned to) a component is determined by the name of the component, the location of your model file (.rtmdl), and the **OutputDirectory** (Component, Java Generation) property.

For details on the generated code pattern, see *Model to Code Correspondence* on page 49.

For details on the code generation process, see *Build Overview* on page 64 and *Build Details* on page 67.

Java UML Services Library

The Java UML Services Library is at the heart of Rational Rose RealTime Java. The Java UML Services Library is a model loaded at Startup. For details, see *Java UML Services Library Framework* on page 75. You need to understand its architecture to start optimizing and configuring the Java UML Services Library for your project.

The behavior of a model is specified using a combination of capsule state diagrams and operations defined on classes and capsules. The relationships in the model are specified with a combination of capsule structure and class diagrams. When a model is built, these abstractions are automatically converted to implementation. The Rational Rose RealTime Java UML Services Library provides a set of built-in services commonly required in real-time systems.

These services include:

- State machine handling
- Message passing
- Timing
- Concurrency control
- Thread management
- Debugging facilities

In summary, the facilities provided by the Rational Rose RealTime Java UML Services Library are:

- The mechanisms that support the implementation of concurrent communicating state machines.
- Thread management and concurrency control.
- Dynamic structure.
- Timing.
- Inter-thread communication.
- Observation and debugging of a running model.

Compilation

Rational Rose RealTime Java converts a model to Java code but does not include the compiler that will build from the generated source code. Before trying to build a generated model, ensure that your compiler tools are correctly installed. For example, try building a simple Java program from the command-line. If that works, then Rational Rose RealTime Java will be able to properly invoke the configured compiler.

For details on the the compilation process, see *Build Overview* on page 64 and *Build Details* on page 67.

Model Properties

The notations supported in Rational Rose RealTime are more abstract than the Java programming language. Model properties enable you to provide language-specific information that is not expressed in the notation, but that is necessary for generating and building source code. Each model property can be assigned a model property value. When a model element is created, each model property is assigned a default value, which you can optionally modify.

In order to build source code, the code generator also generates makefiles which specify how to build the generated source code. Certain properties affect how these makefiles are to be generated and their contents.

Controlling a particular aspect of code generation may require several model properties. For detailed reference to the model properties, see *Model Properties Reference* on page 103.

Note: Not all model components for which code is generated require model properties. For example, there are no model properties for inherits relationships, yet the Java generator produces code from inherits relationships. In such cases, information obtained from specifications is sufficient to control code generation.

Target Observability

Rational Rose RealTime's graphical observation tools are a sophisticated, yet intuitive debugging environment allowing you to use the toolset to execute, monitor and control a model running on the Java UML Services Library, even on a remote target platform. The Java UML Services Library is a high-performance implementation intended for use in a wide-range of real-time products.

For details on the the Java UML Services Library, see *Java UML Services Library* on page 75.

Getting Started with Rational Rose RealTime Java

2

Contents

This chapter is organized as follows:

Building Java Systems in Rational Rose RealTime

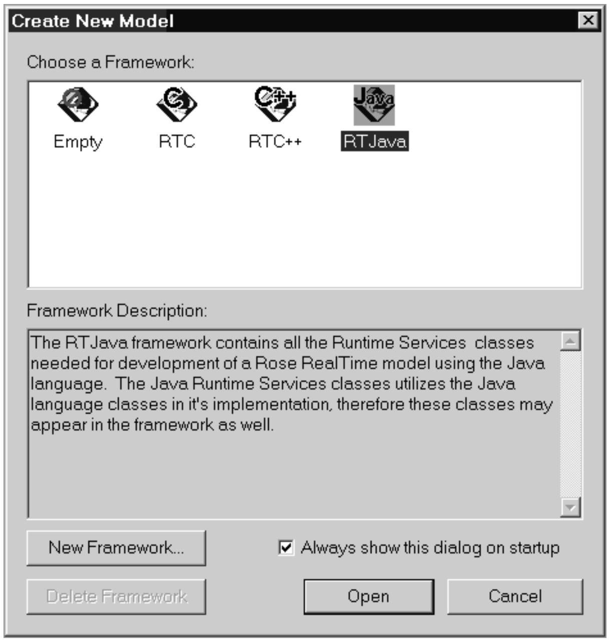
This chapter is intended as a quickstart guide for getting started building Java systems in Rational Rose RealTime. It covers building and running simple data classes, building and running capsule classes, and integrating external classes into your model.

The models described here are available in the Rational Rose RealTime home directory located in the Examples/Models/Java directory.

Creating an Empty Model

The fastest way to get started with Rational Rose RealTime Java is to use the **New Model Wizard** that opens when Rational Rose RealTime is first invoked. Figure 1 on page 24 shows the New Model Wizard startup screen.

Figure 1 New Model Wizard



The RTJava framework contains all the Runtime Services classes needed for development of a Rose RealTime model using the Java language. The Java Runtime Services classes utilizes the Java language classes in it's implementation, therefore these classes may appear in the framework as well.

[✓] Always show this dialog on startup

In the **Create New Model** dialog box, double-click **RTJava.** The Java classes are loaded into the model, including:

Logical View/com/rational/rosert: All the classes used to implement the Java UML Services Library.

Logical View/java/[io,lang,util]: The Java base classes. This package contains only those classes which are needed for the Java UML Services Library.

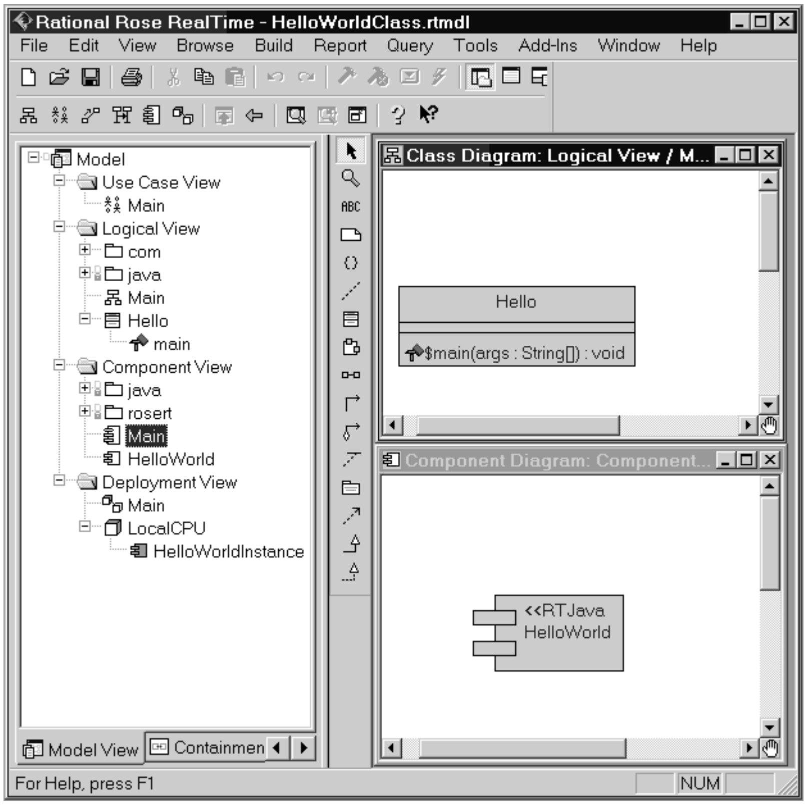
Component View/java: The components needed to create applications that reference the Java base classes.

Component View/rosert: The components needed to create applications that depend upon the Java UML Services Library.

Note: There may be additional frameworks available from $ROSERT_HOME/RTJava/.

Creating a Simple Class

Figure 2 Rational Rose RealTime with HelloWorldClass.rtmdl

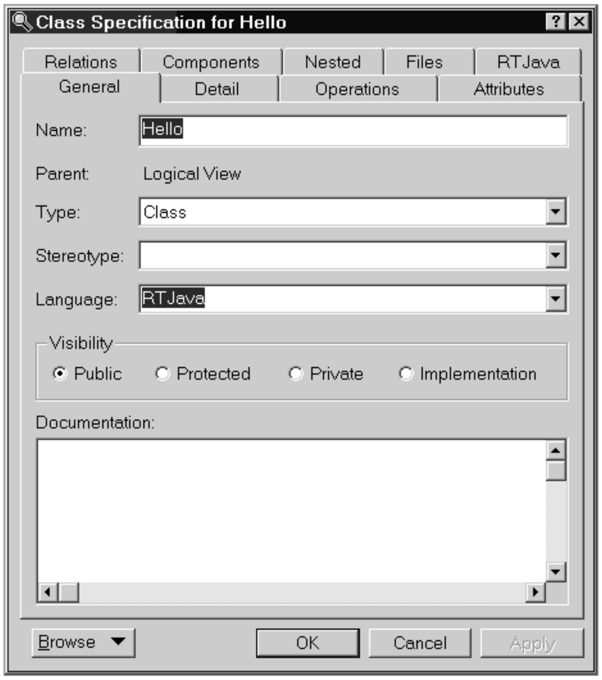

Rational Rose RealTime - HelloWorldClass.rtmdl
File Edit View Browse Build Report Query Tools Add-Ins Window Help
Model
Use Case View
Main
Logical View
com
java
Main
Hello
main
Component View
java
rosert
Main
HelloWorld
Deployment View
Main
LocalCPU
HelloWorldInstance
Model View
Containmen
Class Diagram: Logical View / M...
Hello
$main(args : String[]) : void
Component Diagram: Component...
<<RTJava
HelloWorld
For Help, press F1
NUM

The above diagram shows the results of creating a simple "Hello World" class in Rational Rose RealTime, and opening the **Component Diagram** for the **Main** component.

To create the new class, right-click **Logical View** (in the **Model View**) and click **New > Class.** A new **Class Diagram** appears. In the **Component View**, right-click on **Main**, and click **Open**. The **Component Diagram** appears.

You can now open the **Class Specification** diagram for the new class. In the **Class Diagram**, right-click the **Hello** class, and click **Open Specification**. The **Class Specification for Hello** dialog box appears. This is illustrated in Figure 3.

Figure 3 Specification for Hello Class



Name: Hello

Parent: Logical View

Type: Class

Stereotype:

Language: RTJava

Visibility: [✓ Public] Protected Private Implementation

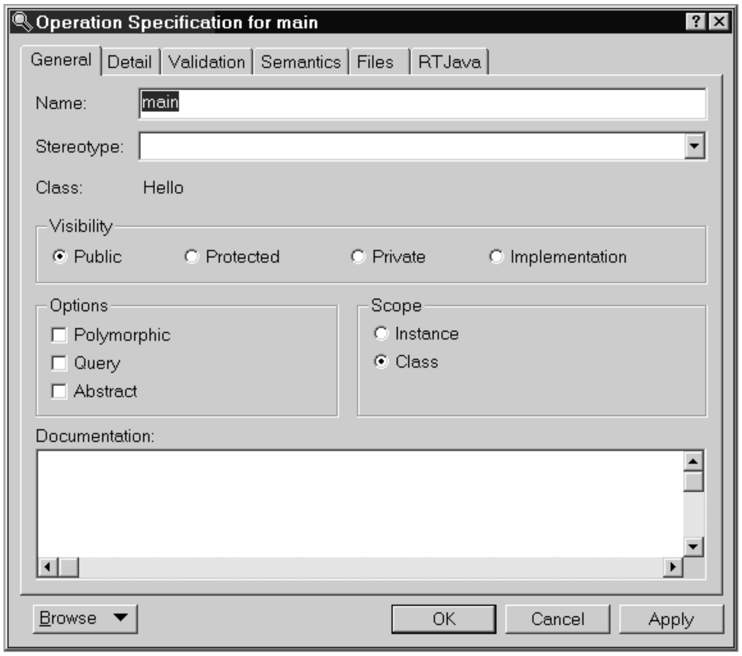
Documentation:

In Java, any class can be run if it has an operation named "main" that has the appropriate scope and signature (that is, class scope, public, one parameter of type String[], and a return type of **void**).

To create the "main" operation, right-click in the window of the **Operations** tab (**Class Specification for Hello** dialog box), and click **Insert**. A new operation appears that you can rename **main**, and make **void** as the **Return type**.

Figure 4, Figure 5, and Figure 6 show different tabs of the **Operation Specification for main** dialog box.

Figure 4 General Specification for Operation "main"


Operation Specification for main

General | Detail | Validation | Semantics | Files | RTJava

Name: main

Stereotype:

Class: Hello

Visibility: [✓ Public] Protected Private Implementation

Options: ☐ Polymorphic ☐ Query ☐ Abstract

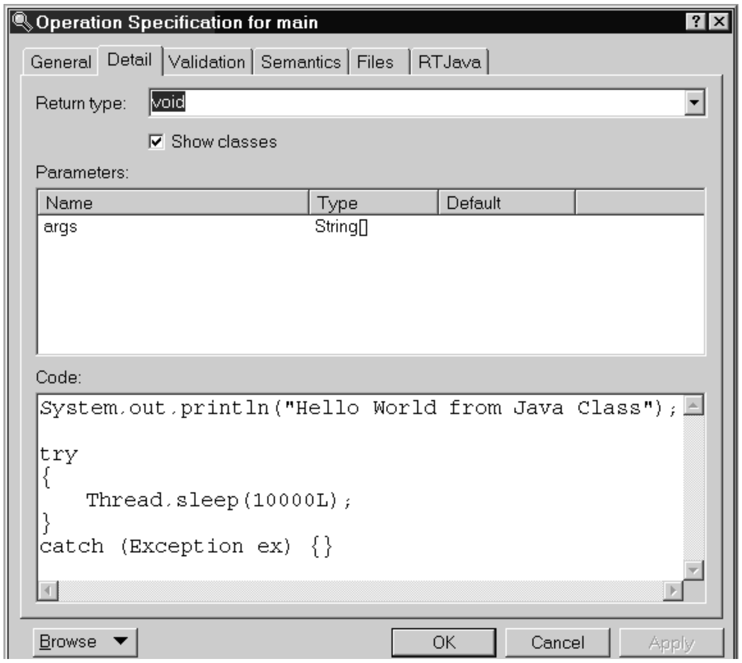
Scope: Instance [✓ Class]

Documentation:

Figure 4 shows how to make this a public class scoped operation.

Note: If additional customization of the operation is required, you can make these customizations in the **RTJava** tab (shown in Figure 6).

Figure 5 Detail Specification for Operation “main”



Return type: void

☑ Show classes

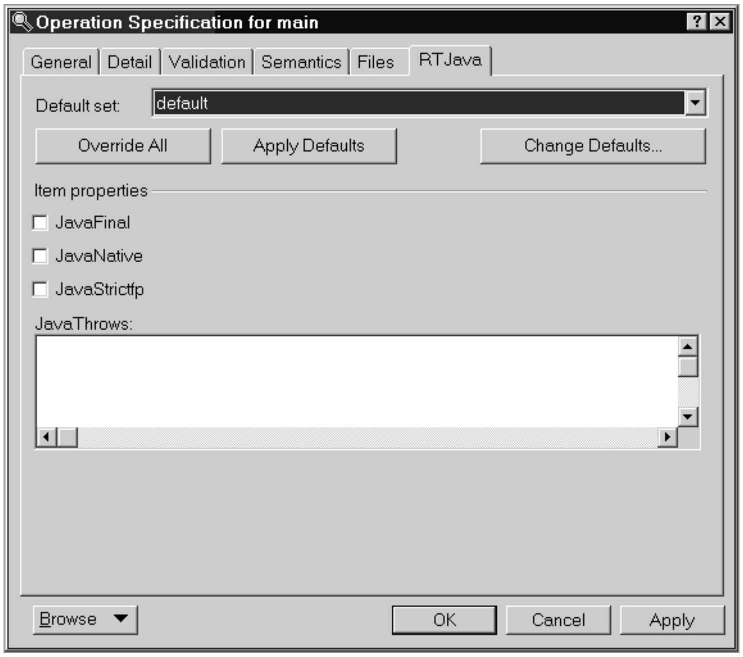
Parameters:

Name	Type	Default
args	String[]	

Code:

```
System.out.println("Hello World from Java Class");

try
{
    Thread.sleep(10000L);
}
catch (Exception ex) {}
```

Figure 6 RTJava Properties for Operation “main”



Default set: default

☐ JavaFinal

☐ JavaNative

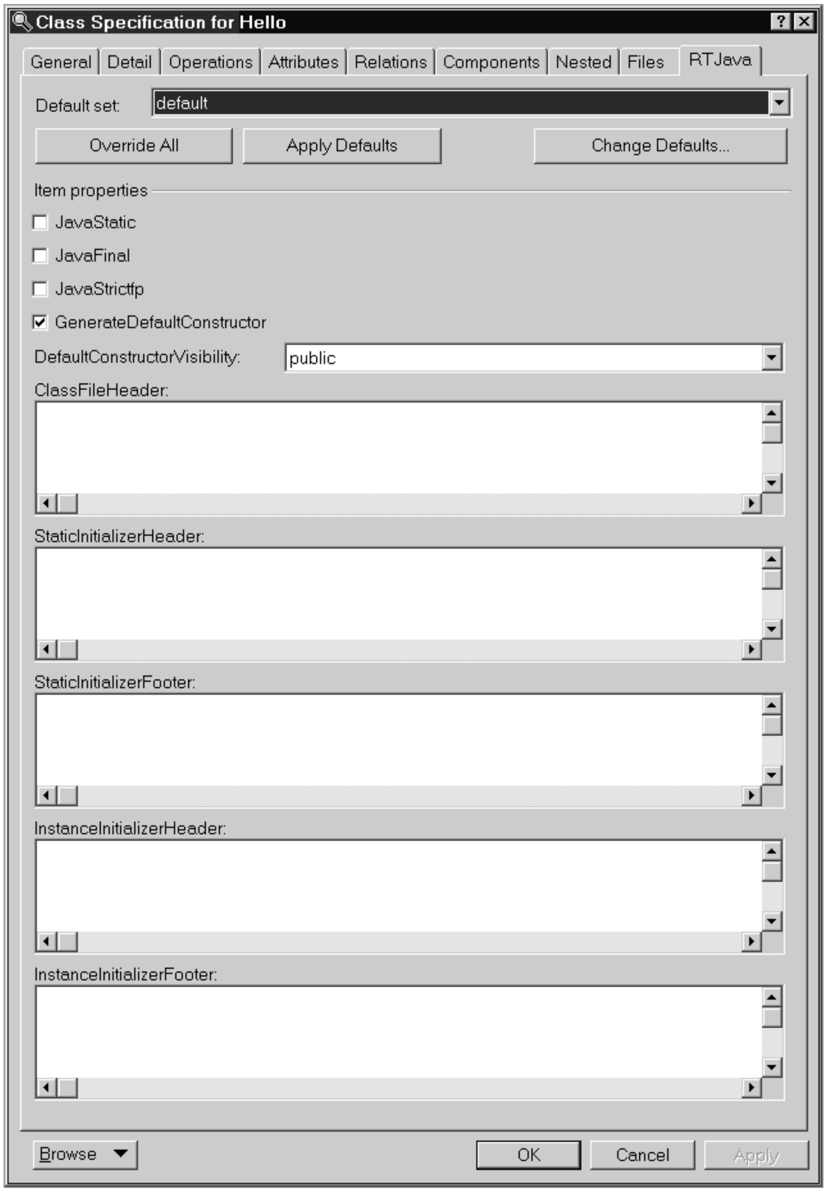
☐ JavaStrictfp

JavaThrows:

The **RTJava** property page shows how to set "final" (**JavaFinal**), "native" (**JavaNative**), and "**strictfp**" (**JavaStrictfp**) modifiers for this operation, and also provides a place to specify the list of exceptions that can be thrown (**JavaThrows**).

The **RTJava** tab is used to set other properties of this class.

Figure 7 Default RTJava Properties for Class "Hello"



Default set: default

- [ ] JavaStatic
- [ ] JavaFinal
- [ ] JavaStrictfp
- [✓] GenerateDefaultConstructor

DefaultConstructorVisibility: public

ClassFileHeader:

StaticInitializerHeader:

StaticInitializerFooter:

InstanceInitializerHeader:

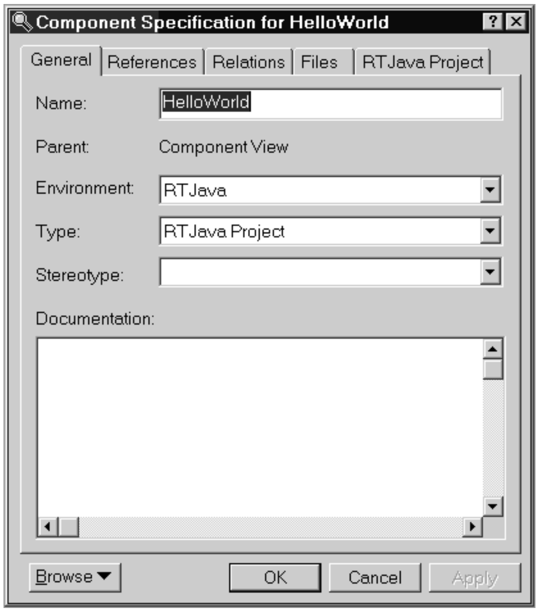
InstanceInitializerFooter:

Figure 7 shows the default **RTJava** tab for the **Hello** class. The **RTJava** tab is used to set other properties of this class. For further information on the contents of the **RTJava** tab, see *RTJava Properties* on page 103.

Building Classes

To generate the code for this class, and then compile it, you need to create a component in the **Component View**. This must be an **RTJava Project** so that the classes will be generated and built by this component. For a definition of **RTJava External Project**, see *Integrating External Classes* on page 42. The diagram containing this component is shown in Figure 2 on page 25. Figure 8, Figure 9 and Figure 10 show the specification of this component.

Figure 8 General Specification for Component "HelloWorld"


Component Specification for HelloWorld

Name: HelloWorld
Parent: Component View
Environment: RTJava
Type: RTJava Project
Stereotype:
Documentation:

Figure 9 shows that this component only has one class ("Hello") that will be included. You can also add packages in this tab. If a package is added, it is equivalent to adding individually all the classes in this package, and all the classes contained in all the subpackages.

Figure 9 References Specification for Component "HelloWorld"

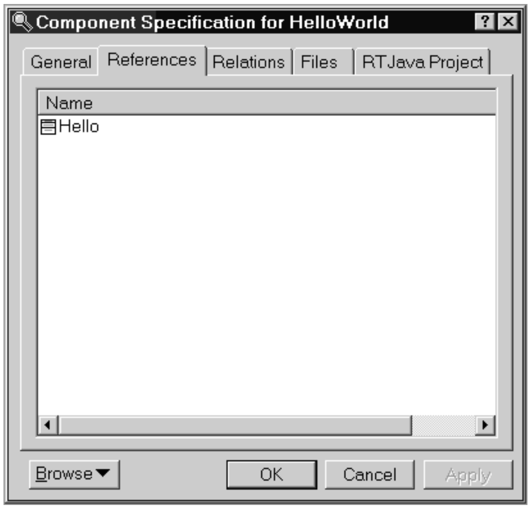
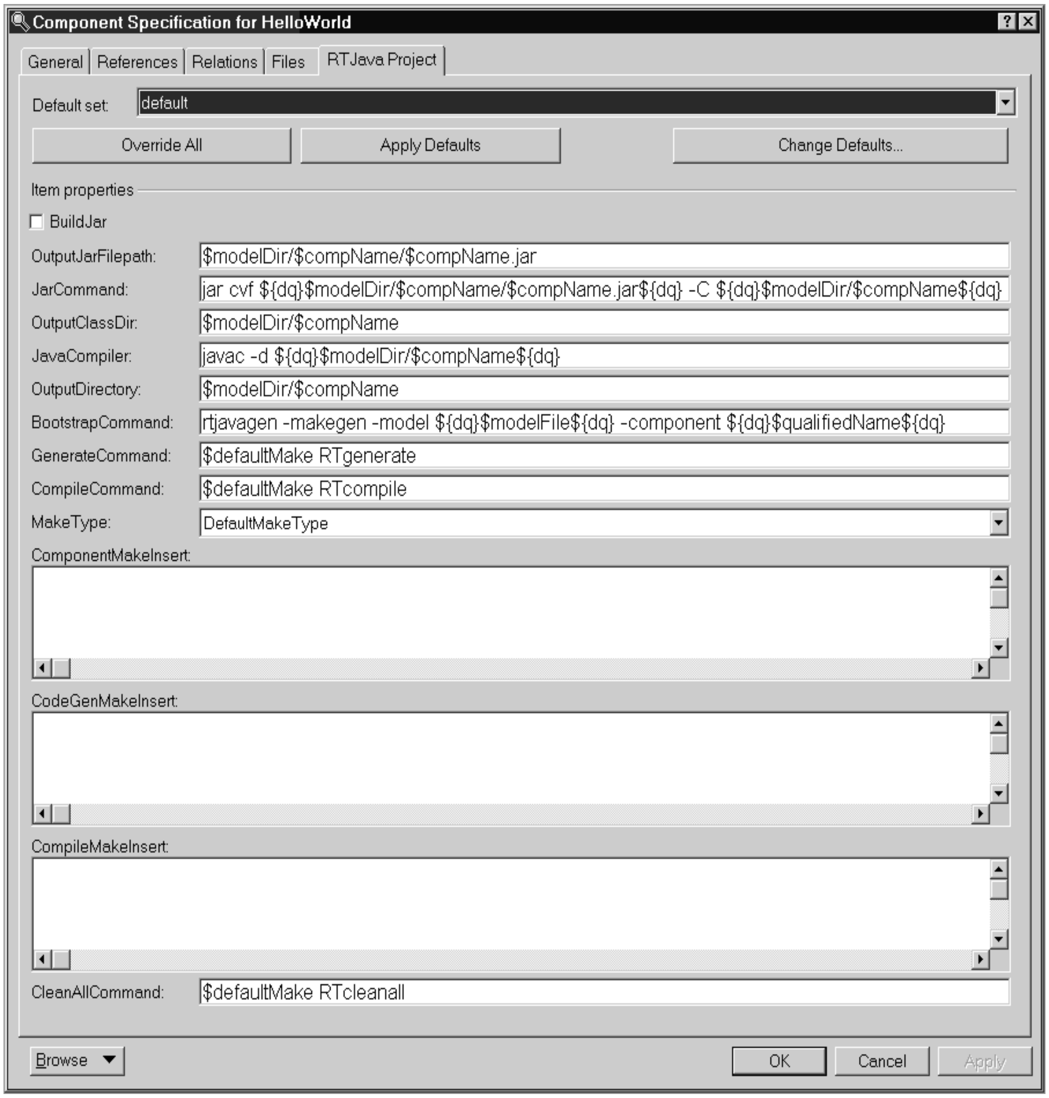


Figure 10 shows all the properties that may be used to configure how this set of classes will be generated and built. Although the **RTJava Project tab** may appear complicated, all the values shown are default values. You can use this tab to change properties, such as, the **MakeType** or the name or arguments of the **JavaCompiler**. For example, you may want to add additional flags to `javac`.

Figure 10 RTJava Project Settings for Component “HelloWorld”



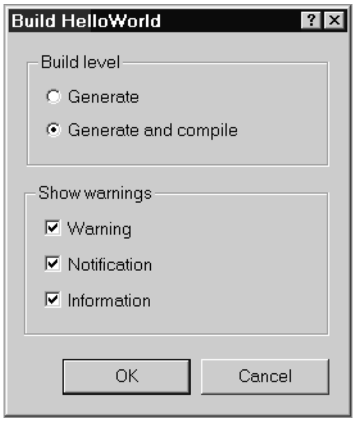
Default set: default

☐ BuildJar

OutputJarFilepath:	$modelDir/$compName/$compName.jar
JarCommand:	jar cvf ${dq}$modelDir/$compName/$compName.jar${dq} -C ${dq}$modelDir/$compName${dq}
OutputClassDir:	$modelDir/$compName
JavaCompiler:	javac -d ${dq}$modelDir/$compName${dq}
OutputDirectory:	$modelDir/$compName
BootstrapCommand:	rtjavagen -makegen -model ${dq}$modelFile${dq} -component ${dq}$qualifiedName${dq}
GenerateCommand:	$defaultMake RTgenerate
CompileCommand:	$defaultMake RTcompile
MakeType:	DefaultMakeType
ComponentMakeInsert:	
CodeGenMakeInsert:	
CompileMakeInsert:	
CleanAllCommand:	$defaultMake RTcleanall

After you have created a valid component, you can build it.

Figure 11 Build Dialog Box for Component "HelloWorld"


Build HelloWorld

Build level

Generate [✓ Generate and compile]

Show warnings

[✓ Warning]

[✓ Notification]

[✓ Information]

To build a component, right-click the **HelloWorld** component (in the **Component View**), and click **Build > Build**. The **Build HelloWorld** dialog box appears. Click **OK**, and the class "Hello" generates and compiles.

The generated code for this class is:

```
public class Hello
{
   public static void main( String[] args )
   {
      System.out.println("Hello World from Java Class");
      try
      {
         Thread.sleep(10000L);
      }
      catch (Exception ex) {}
   }
   public Hello()
   {
   }
}
```

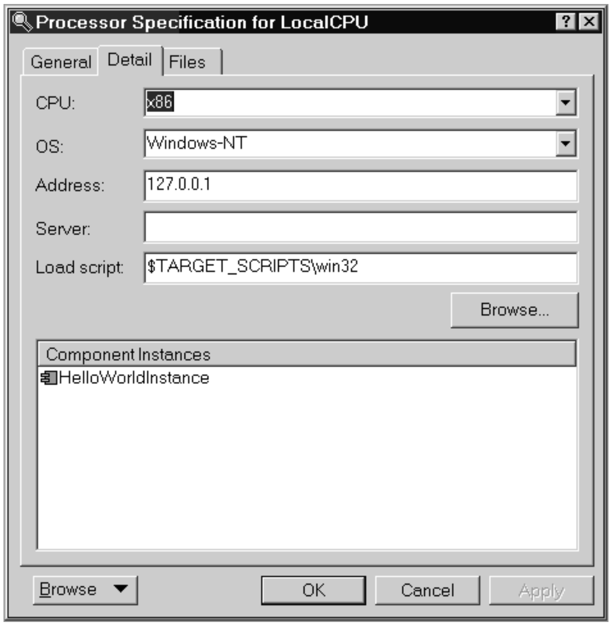
In the previous code, you can see the sleep function that is called after doing the `println()`. This sleep statement is included for illustration purposes, only. If the toolset is used to start the executable, it will close the **Output Window** when the application is finished (for details, see the following section). In this example, the **Output Window** would have closed very quickly without the sleep.

Running a Class Based Application

In order to run the generated application, you need a description of the target environment. Figure 12 shows the default values for a local processor that was created in the **Deployment View**.

Note: It is recommended that you start with the default values, and modify the specification as required.

Figure 12 Default Specification for Processor "LocalCPU"


Processor Specification for LocalCPU

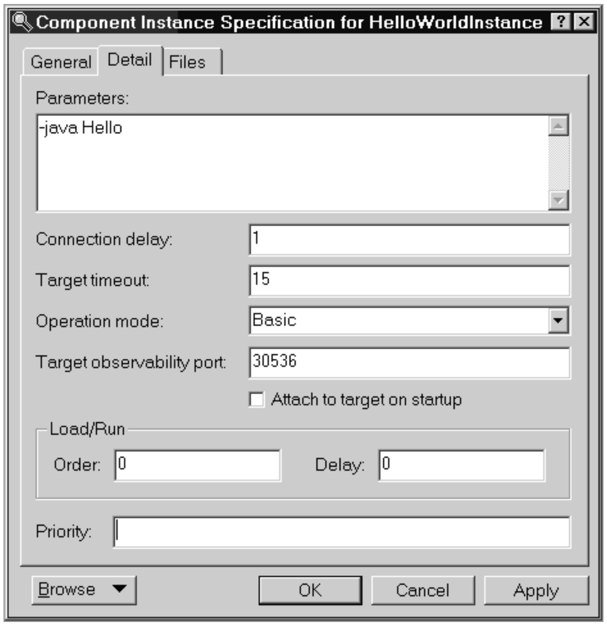
CPU: x86
OS: Windows-NT
Address: 127.0.0.1
Server:
Load script: $TARGET_SCRIPTS\win32

Component Instances
HelloWorldInstance

After you have defined the processor, drag the **HelloWorld** component onto it. A component instance, **HelloWorldInstance**, is created under the **LocalCPU** processor.

Figure 13 shows a specification of the **HelloWorldInstance**.

Figure 13 Component Instance Specification for “HelloWorldInstance”



Parameters: -java Hello

Connection delay: 1

Target timeout: 15

Operation mode: Basic

Target observability port: 30536

☐ Attach to target on startup

Load/Run

Order: 0 Delay: 0

Priority:

In the above diagram, **`-java`** indicates that you will use Java as the virtual machine (VM). `Hello` is the name of the class that you will run. Ensure you clear the **Attach to target on startup** box because only capsule based models can be debugged with this flag.

The **Parameters** box has two sections:

- Commands for Rational Rose RealTime:
- Arguments for the Java Virtual Machine

Valid commands for Rational Rose RealTime are:

```
-java
-vm <vm_name>
-classpath <path>
```

All other commands are passed to the Java Virtual Machine.

If **`-java`** is listed, Rational Rose RealTime invokes java.exe as the virtual machine. If **`-vm<vm_name>`** is listed, Rational Rose RealTime uses **`<vm_name>`** as the virtual machine. (For example, **`-vm kvm.exe`** or **`-vm midp.exe`**.)

Note: Only one occurrence of these commands can be used.

The **`-classpath`** argument constructs a `classpath` argument for the VM. Rational Rose RealTime may include additional elements to the `classpath` that are typically needed to ensure that all the components in the design are used. Unlike most Java VM's, there can be many **-classpath** options. Rational Rose RealTime merges them into a single `classpath` before presenting them to the VM.

Valid commands for the Java VM are many and varied. In this document, we will assume

```
<ClassName> [ <arguments> ]
```

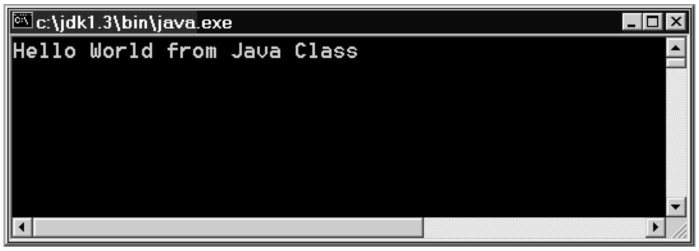
where `<ClassName>` is the entry point for Java. This class has a main routine that will be invoked first. The `<arguments>` will be passed in as the parameter to **main**.

In the **Deployment View**, right-click **HelloWorldInstance**, and click **Run**. A console window opens on the target (using the local host specified in Figure 12), and runs the Java application in it.

Note: Rational Rose RealTime adds `-classpath` to the **Run** command to reflect the directories used by the build process. For example, if the output directory is C:\HelloWorld, the following command would be used:

```
Java -classpath C:\HelloWorld\ Hello
```

Figure 14 Output Console for class "Hello"



```
Hello World from Java Class
```

The console window will disappears in ten seconds. To stop the application when it is running, click **Shutdown** on the **HelloWorldInstance**.

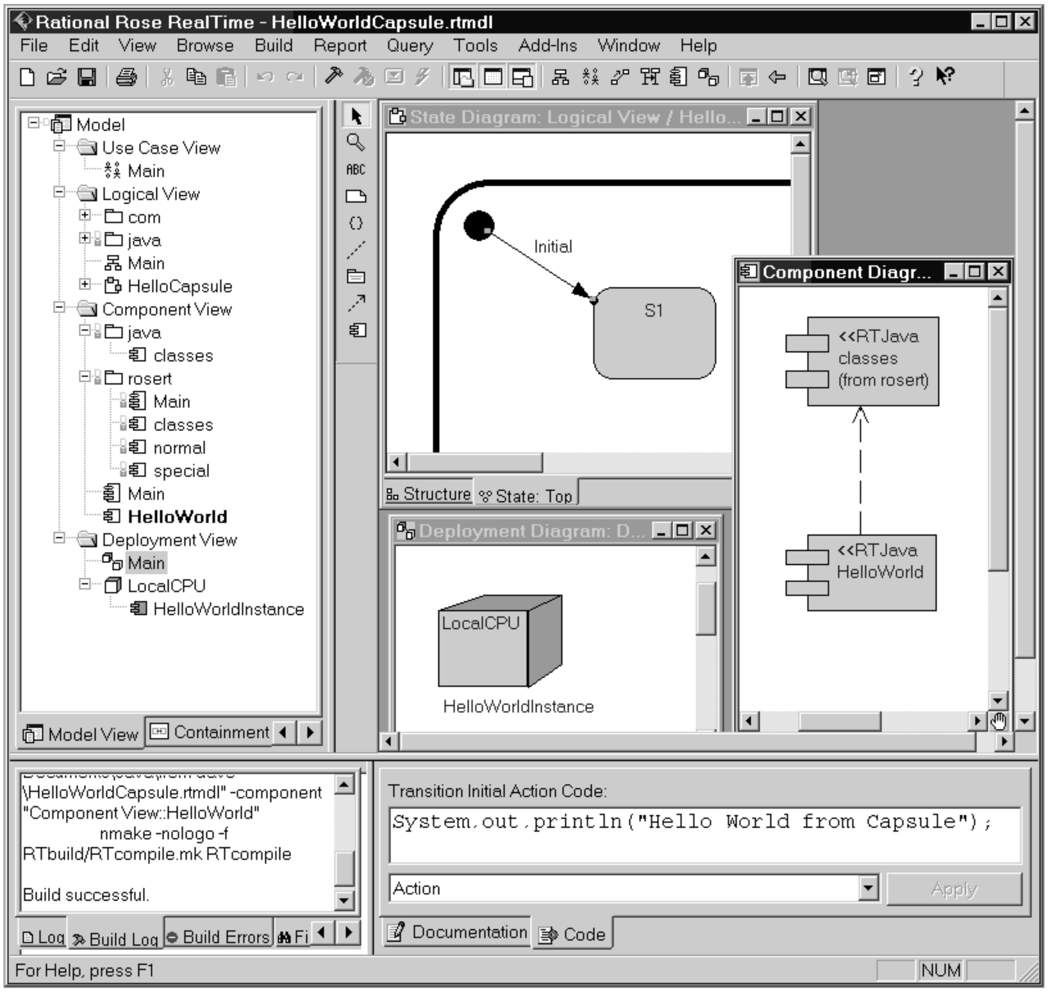
Creating Capsules

A system that contains capsules is similar to one that does not but has a few important differences, including:

- Capsules have state machine behavior.
- Capsules have structure.
- Capsules require the Java UML Services Library to compile and run.
- Systems containing capsules are usually invoked differently from simple class based systems.
- Systems containing capsules can be debugged visually by Rational Rose RealTime at the "message" level, in addition to the more traditional 3GL statement level.

The following diagram shows a capsule based design in Rational Rose RealTime. This model is located in
$ROSERT_HOME/Examples/Models/RTJava/HelloWorldCapsule.rtmdl.

Figure 15 Capsule Based Model



```
\HelloWorldCapsule.rtmdl" -component
"Component View::HelloWorld"
            nmake -nologo -f
RTbuild/RTcompile.mk RTcompile

Build successful.
```

```
System.out.println("Hello World from Capsule");
```

This is a very simple model that has a single capsule with a trivial state machine which outputs "`Hello World From Capsule`" on its initial transition.

Figure 15 also shows a dependency from the **HelloWorld** component to the **classes** component from the **rosert** component package. The **classes** component contains all the classes necessary to implement the services (that is, frame, log, messaging, behavior, threading, and so on) needed by capsules. This dependency must be added by the designer when creating a capsule based system. If the dependency is left out, and capsules are used, the code generator raises an error message "Use of Capsules requires Java UML Services Library" on the offending component.

The **rosert** package also contains a component **normal** that is used to build the classes in the Java UML Services Library.

Note: These classes are all present in the tool in the **Logical View** under **com.rational.rosert**. If there are any questions about the behavior of the Java UML Services Library, this is the definitive answer because it is the actual source code. This behavior differs from the C and C++ versions of Rational Rose RealTime which have external code that provides these services. You can modify the Java UML Services Library by changing this model, generating, and compiling it from the **normal** component (however, this is beyond the scope of this document).

Running a Capsule Based Application

In addition to the dependency on **classes**, a capsule based application also needs a different configuration of its component instance in order to run.

The Java UML Services Library has its own entry points; therefore, the command-line arguments are more complicated than a simple class name.The Java UML Services Library requires that you used a predefined entry point and use the top capsule as its argument. The entry points that should be used are:

- `com.rational.rosert.Application`: An entry point for a minimal Java UML Services Library that does not include event debugging.
- `com.rational.rosert.DebugApplication:` An entry point that includes support for full UML event level debugging. This version will load approximately twice as much code as the simpler `Application`.

In the **Component Specification,** capsule based applications can use the same Rational Rose RealTime commands but require a few additional argument for the entry point. In general, the form is one of the following:

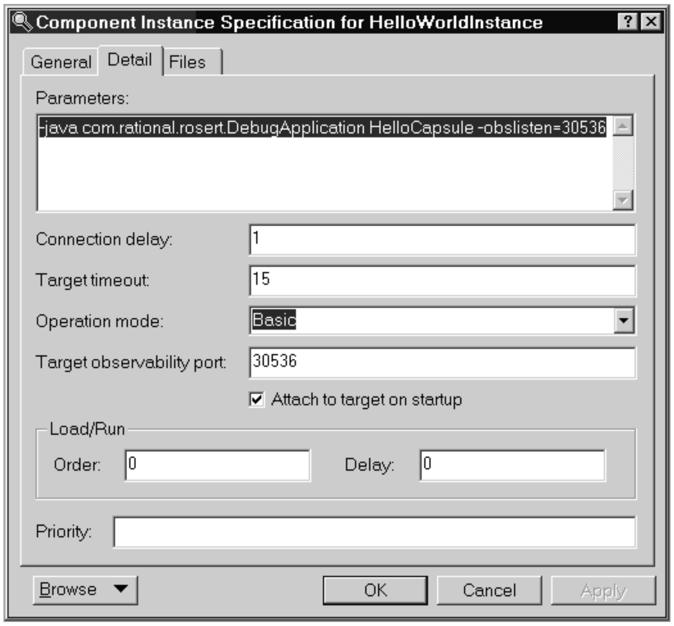
```
com.rational.rosert.Application <Top_Capsule>  [<extra args>]
```

```
com.rational.rosert.DebugApplication <Top_Capsule> [ -obslisten
=<port_num>]
```

where

`-obslisten=<port_num>` enables the debugging of UML events using TCP port number `<port_num>`. The running application will listen on this port for both toolset and telnet connections. The menu item on the component instance **Attach Console** will open a telnet session to the target allowing a "command-line" debugging functionality.

Figure 16 Component Instance Specification for Capsule Based Model


Component Instance Specification for HelloWorldInstance

Parameters: -java com.rational.rosert.DebugApplication HelloCapsule -obslisten=30536

Connection delay: 1

Target timeout: 15

Operation mode: Basic

Target observability port: 30536

☑ Attach to target on startup

Load/Run

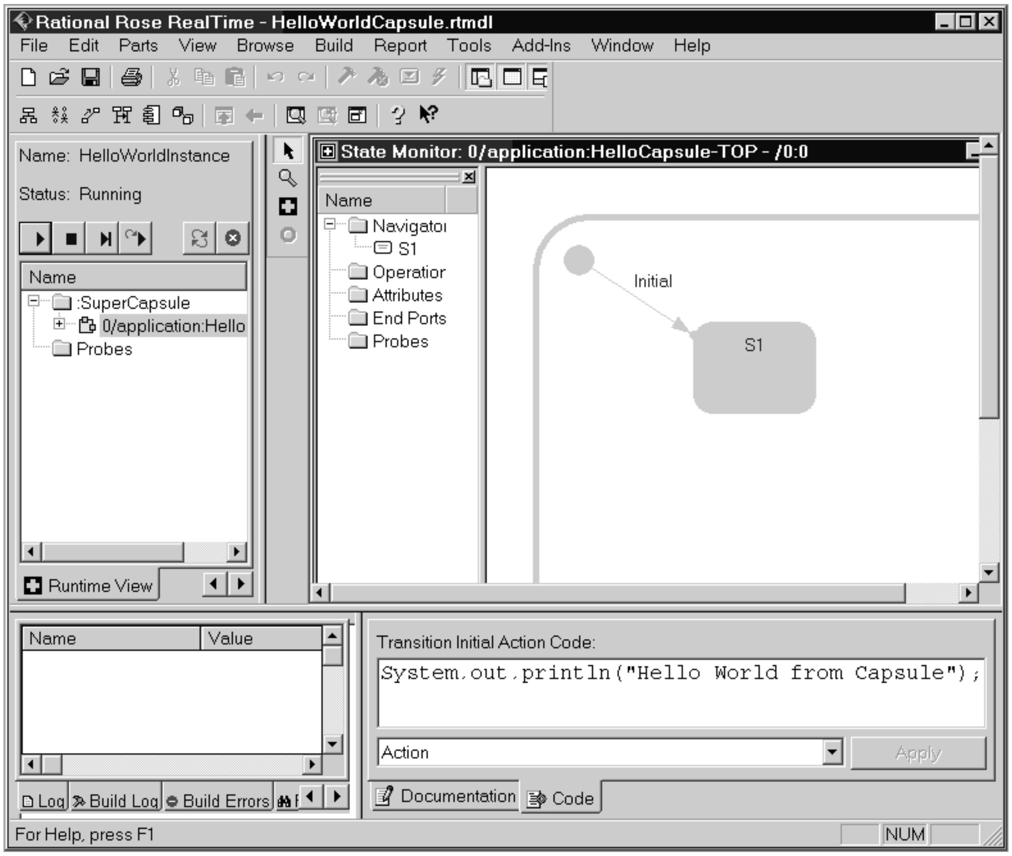
Order: 0 Delay: 0

Priority:

A debug enabled component instance for the Hello World system is shown in the above **Component Instance Specification**. The **Attach to target on startup** checkbox has the tool make the debugging connection immediately on startup of the application.

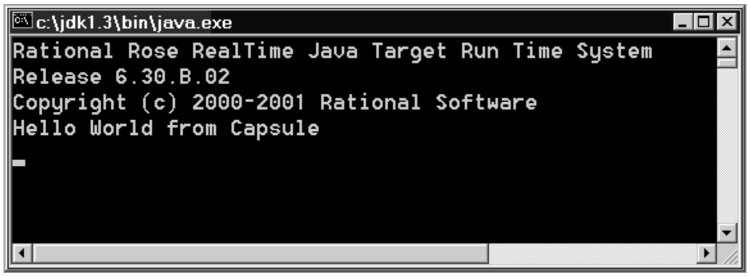
Click **Run** in the **Runtime View** browser, and open the **State Monitor** for the **0/application:HelloCapsule**. The results of these actions are illustrated in Figure 17 and Figure 18.

Figure 17   UML Debugger on a Running “HelloWorldCapsule”



```
System.out.println("Hello World from Capsule");
```

Figure 18   Output Console for “HelloWorldCapsule”



```
Rational Rose RealTime Java Target Run Time System
Release 6.30.B.02
Copyright (c) 2000-2001 Rational Software
Hello World from Capsule
```

Integrating External Classes

There are many cases where you may have classes that are not defined in the toolset, either from a third-party or in code that will be re-used for a new project. These externally defined classes can be integrated within Rational Rose RealTime, and can be used for class modeling. They are then available in the type lists, or are used within detail level code.

Any class or type defined outside the toolset can be used in your model. Before integrating classes into Rational Rose RealTime, you need to first consider how the class will be used within the model. Then, based on how the class or type is needed in your model, you can integrate the class or type within Rational Rose RealTime:

- Will objects of this type only be used to store information within a single capsule or data instance?

  In this case, the only step required for using this class in your model is to either refer to the class always by its fully qualified name. For example, use `java.lang.String` rather than `String`. Or, explicitly import the class or package from the **RTJava** property **ClassFileHeader** (see Figure 7) by typing the exact import command. For example, either `import java.lang.*;` or `import java.lang.String`. After the import statement is added, this external class will be available to the modeled class so you can use the class within any detailed level code (or attribute or operation definition).

- Do objects of this type need to be shown in the UML Diagram? For example, if they must be sub-classed by classes in your model, or if they are architecturally significant and you wish to show them is a diagram.

  In this case, it is necessary to create "stub" classes so that the code generation engine can access properties about these classes. The most commonly needed property is the classes' name.

  Note: Figure 19 shows an sample system that has multiple levels of components, the root of which are the external classes for `java` and `javax`. This model is located in $ROSERT_HOME/Examples/Models/RTJava/PhoneSystem.rtmdl.

Figure 19 Component Diagram of a System Using AWT and Swing

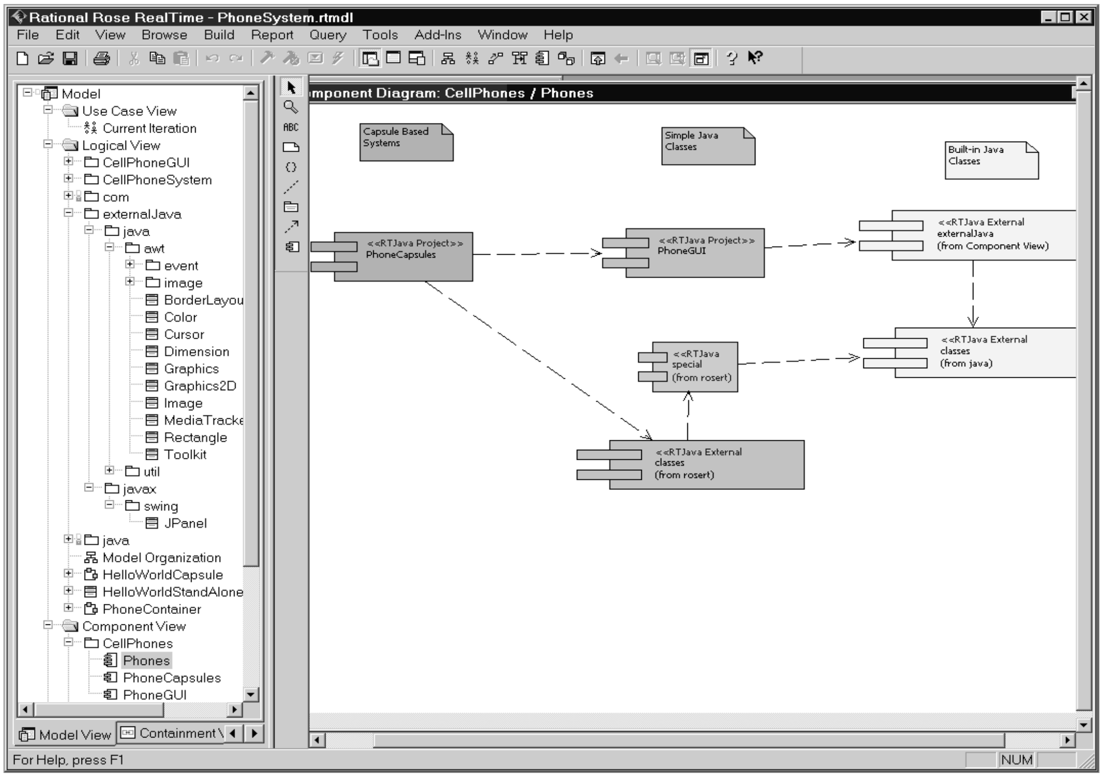
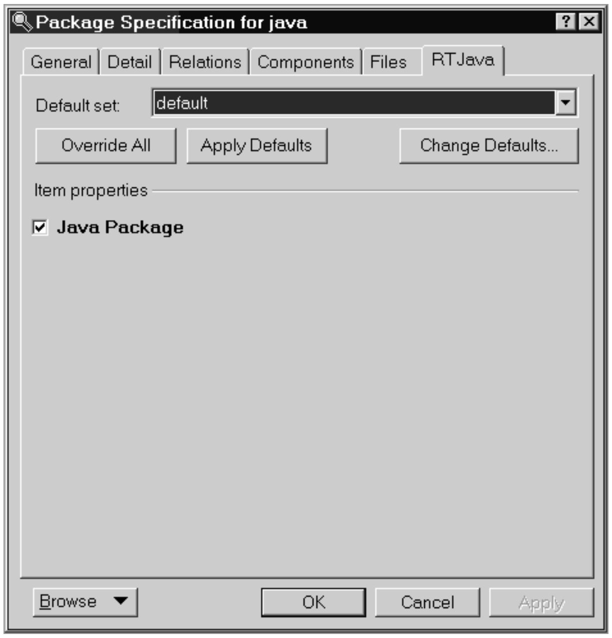


Figure 20 illustrates the specification of the **java** package in the **externalJava** package from Figure 19. The property **Java Package** distinguishes UML packages that have no namespace implications from Java packages that do have namespace implications.

Figure 20 Package Specification with java Namespace Support


Package Specification for java

Default set: default

Item properties

☑ Java Package

With this information we can create the **stub** class.

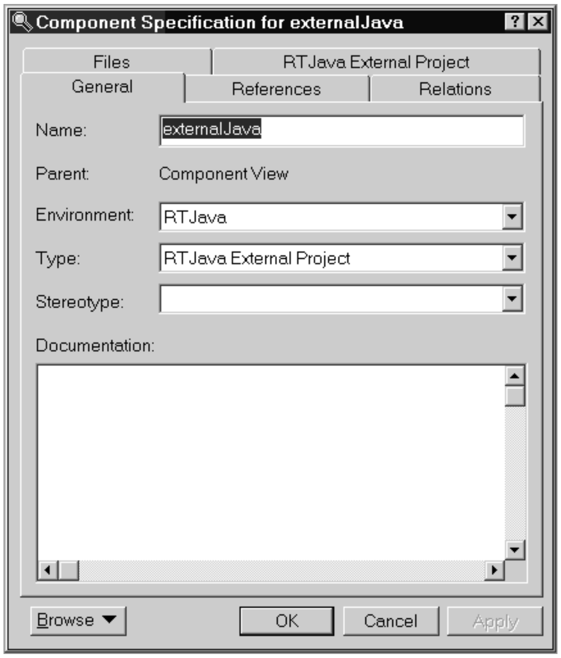
To use `javax.swing.Jpanel` in your model, do the following:

1. Create a top level package, **externalJava**, to contain the new classes. Ensure that **JavaPackage** is cleared.
2. In the **externalJava** package, create a new package, **javax**. Select **JavaPackage**.
3. In the **javax** package, create a New package, **swing**. Select **JavaPackage**.
4. In the **swing** package, create a new class, **JPanel**.

   All properties on this class will be ignored by the code generator, so the user may either ignore them or set them to the real values (for documentation purposes).
5. Create a component, **externalJava**.

6 In the **General** tab, do the following:

- In the **Environment** box, select **RTJava**.
- In the **Type** box, select **RTJava External Project**.



Name: externalJava

Parent: Component View

Environment: RTJava

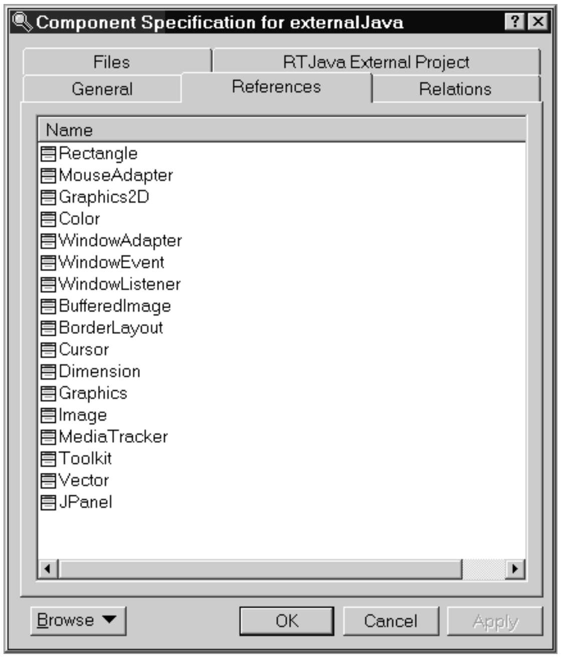
Type: RTJava External Project

Stereotype:

Documentation:

7 In the **References** tab, add **JPanel**.

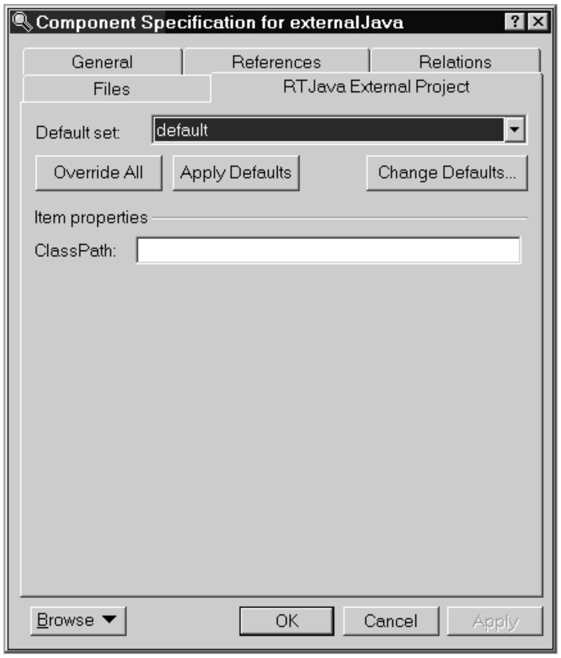
Note: You can add the entire package **externalJava** as a shortcut.



Name
Rectangle
MouseAdapter
Graphics2D
Color
WindowAdapter
WindowEvent
WindowListener
BufferedImage
BorderLayout
Cursor
Dimension
Graphics
Image
MediaTracker
Toolkit
Vector
JPanel

8 In **RTJava External Project** tab, set the **ClassPath** to this class

Note: Since **JPanel** is already on the **ClassPath**, the following diagram shows the **ClassPath** as empty.



Default set: default

ClassPath:

9 You must now create a dependency from your component (for example, **HelloWorld**) to this component (**externalJava**).

Note: This ensures that the code generator and deployment technology understands that you will use these external classes. Figure 19 on page 43 illustrates a more realistic component diagram.

Code Generation 3

Contents

This chapter is organized as follows:

Model to Code Correspondence

The Java code generator uses the specifications and model properties of elements in the current model to produce Java source code. You generate code for a component which in turn references a set of classes from the **Logical View**. The location of the source files that are generated for these classes is determined by the **OutputDirectory** property (**Component, RTJava Project**).

If classes have not been assigned to components, either directly or by means of a dependency on other assigned classes, the Java code generator will not see those classes and they will not be generated to source.

Logical View Packages

Logical packages may be marked as Java packages using the **JavaPackage** property (**Package**, **RTJava**), or by setting the stereotype of the package to **JavaPackage**. The qualified name formed by the hierarchy of Java packages designates the package for the classes contained in that package.

Classes

Each class is generated in its own Java source file. While there are a number of class types available in Rational Rose RealTime, only regular classes are supported for Java.

The layout of the file generated from a class is as follows:

- package declaration based on the containing Java package hierarchy
- import statements automatically derived from the model
- contents of the **ClassFileHeader** property (**Class**, **RTJava**) that can be used to place any additional import statements
- the class definition

Modifiers for the class are determined as follows:

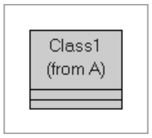
- The **Visibility** setting controls the visibility of the class. The **Implementation** setting specifies default access whereby the class is only visible to other classes in the same package.
- If the **Abstract** setting is selected, the class is declared `abstract`.
- If the **JavaStatic** property (**Class**, **RTJava**) is selected, the class is declared `static`.
- If the **JavaFinal** property (**Class**, **RTJava**) is selected, the class is declared `final`.
- If the **JavaStrictfp** property (**Class**, **RTJava**) is selected, the class is declared `strictfp`.

Properties are also provided to allow the specification of code for initializers. The properties are as follows:

- **StaticInitializerHeader** specifies code in a static initializer at the beginning of the class.
- **StaticInitializerFooter** specifies code in a static initializer at the end of the class.
- **InstanceInitializerHeader** specifies code in an instance initializer at the beginning of the class.
- **InstanceInitializerFooter** specifies code in an instance initializer at the end of the class.

To quickly create compilable classes, select the **GenerateDefaultConstructor** property (**Class**, **RTJava**) which causes an empty default constructor to be generated. The **DefaultConstructorVisibility** property specifies the visibility of this constructor.

Figure 21   A Java Class



The settings for the above class are:

- **Visibility** is set to **pubic.**
- **JavaFinal** is selected.
- **GenerateDefaultConstructor** is selected.
- **DefaultConstructorVisibility** is set to **public**.
- All remaining settings are cleared.
- Code is added to **StaticInitializerHeader**.

The code generated for the class in Figure 21 is:

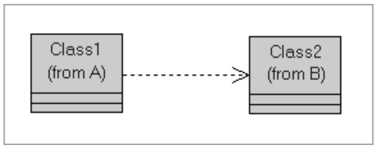
```
package A;

public final class Class1
{
   static {
      // This is static initializer code
   }
   public Class1()
   {
   }
};
```

Note: Nested classes are generated as nested classes in the class definition, and are defined in a similar manner as the containing class.

Dependencies

When a uses relationship exists from a generated class to another class, an imports statement is generated in the generated class.

Figure 22 A Java Class Dependency



If the above dependency is specified, the import statement generated for **Class1** is:

```
import B.Class2;
```

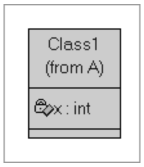
Attributes

An *attribute* is generated in code as a field in the client class. The name of the attribute becomes the name of the field and the type of the attribute becomes the type of the field.

The following settings affect the generation of the attribute:

- The visibility of the attribute becomes the visibility of the field. If the **Visibility** is set to **Implementation**, the default access is used.
- If the **Scope** is set to **Class**, then the field is declared `static`.
- If the **Changeability** is set to **Frozen**, then the field is declared `final`.
- If the **JavaVolatile** property (**Attribute**, **RTJava**) is selected, the field is declared `volatile`.
- If the **JavaTransient** property (**Attribute**, **RTJava**) is selected, the field is declared `transient`.
- The **InitializationCode** property (**Attribute**, **RTJava**) may be used to specify code that is placed in an instance initializer next to the attribute.

Figure 23 A Java Class with an Attribute



The code generated for **Class1** for attribute **x** is:

```
private int x;
```

Associations

An *association* is a relationship among two or more elements. Rational Rose RealTime supports binary associations between classes, including capsule and protocol classes. The ends of each association are called *association ends*. Ends may be labeled with an identifier that describes the role that an associate element plays in the association. An end has both generic and language specific properties that affect the generated code.

The form of code that is generated depends upon the type of classes involved in the association:

- Capsule to protocol is generated as a port on the capsule class. For more information, see *Ports* on page 60.
- Capsule to capsule is generated as a capsule role on the capsule class. For details, see *Capsule Roles* on page 59.
- Any to Capsule is invalid.
- Protocol to Any is invalid.

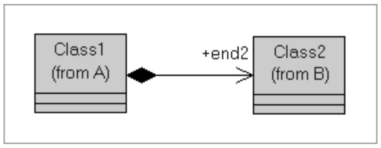
For capsule and regular class to regular class, if the association end is named and is navigable, an attribute is generated in the client class of the end (that is, the class at the other end). The client class may use this attribute to navigate to the supplier objects designed by the class at the association end.

The settings and properties that control the generation of the association end are:

- If **Aggregation** is set to **Composite**, code is generated to initialize the attribute with new objects of the supplier class. Otherwise, the attribute is not initialized.
- The **Multiplicity** of the association end is used when the **Aggregation** is **Composite** to determine the size of the array to allocate.
- If **Target Scope** is set to **Class**, the attribute is declared `static`.

- If the **JavaFinal** property (**AssociationEnd**, **RTJava**) is selected, the attribute is declared `final`.
- If the **JavaVolatile** property (**AssociationEnd**, **RTJava**) is selected, the attribute is declared `volatile`.
- If the **JavaTransient** property (**AssociationEnd**, **RTJava**) is selected, the attribute is declared `transient`.
- The **InitialValue** property (**AssociationEnd**, **RTJava**) contains an initial value that is assigned to the generated attribute.
- The **InitializationCode** property (**AssociationEnd**, **RTJava**) contains code that appears in an instance initializer for the attribute.
- The **NameQualification** property (**AssociationEnd**, **RTJava**) determines whether the supplier class name is to be fully qualified or whether an import statement should be generated for the class.

Figure 24 An Association



If **end2** has **Aggregation** set to **Composite**, and **Multiplicity** set to **One**, the attribute generated in **Class1** is:

```
public B.Class2 end2 = new B.Class2();
```

Operations

Operations in a class is translated by the Java code generator into methods of the generated class. The name and parameters of the operation become the signature of the method. The code associated with the operation becomes the code body for the method.

Other settings and properties that control the generation of operations are:

- The visibility of the operation becomes the visibility of the generated method. If the **Visibility** is set to **Implementation**, the method is declared with default access.
- If the **Abstract** setting is selected, the method is declared `abstract`, and any code in the code setting is ignored.
- If the **Scope** is set to **Class**, the method is declared `static`.

- If the **JavaFinal** property (**Operation**, **RTJava**) is selected, the method is declared final.
- If the **JavaStrictfp** property (**Operation**, **RTJava**) is selected, the method is declared `strictfp`.
- If the **JavaNative** property (**Operation**, **RTJava**) is selected, the method is declared `native`, and any code in the code setting is ignored.
- If the **Concurrency** setting is set to **Guarded**, the method is declared `synchronized`.
- The **JavaThrows** property (**Operation**, **RTJava**) may be used to specify an comma separated list of exceptions that the operation may throw. This list is placed in the throws clause of the method declaration.

For example, an operation `'f'` that takes an `int` parameter and returns an `int` would generate the following code:

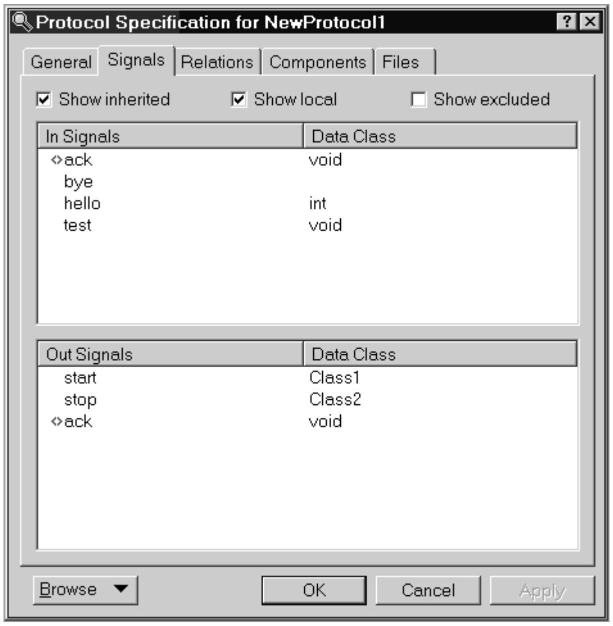
```
public int f( int x )
{
   // Code from the code setting
   return x * 2;
}
```

Protocols

Each protocol is generated as a Java class. This class contains two nested classes for each of the Base and Conjugate protocol roles. Ports on capsules are instances of the protocol role classes. The capsules use the methods on the protocol role classes to access features of the messaging service of the UML Services Library.

Methods and attributes are generated for each signal in a protocol depending on whether the signal is an **In Signal**, an **Out Signal**, or whether the signal is symmetric. The Conjugate protocol role treats the **In Signals** as **Out Signals**, and the **Out Signals** as **In Signals**.

Figure 25 A Protocol Class Definition


Protocol Specification for NewProtocol1

[✓] Show inherited   [✓] Show local   [ ] Show excluded

In Signals	Data Class
<>ack	void
bye	
hello	int
test	void

Out Signals	Data Class
start	Class1
stop	Class2
<>ack	void

The types of signals contained in this **Protocol Specification** are:

- In Signals
- Out Signals
- Symmetric Signals

In Signal

An attribute and an method are defined for each In Signal. The attribute provides a numeric identifier for the signal. The method converts the signal into an object that provides methods to access messaging services for the incoming message.

The code generated for the In Signals (not including methods for the symmetric signals) in Figure 25 on page 56 is:

```
public class NewProtocol1
{
   public static class Base extends ...
   {
      public static final int rti_ack = 1;
      public static final int rti_bye = 2;
      public static final int rti_hello = 3;
      public static final int rti_test = 4;

      public final InSignal bye() ...
      public final InSignal hello() ...
      public final InSignal test() ...
      ...
   }
   public static class Conjugate extends ...
   {
      public static final int rti_start = 1;
      public static final int rti_stop = 2;
      public static final int rti_ack = 3;

      public final InSignal start() ...
      public final InSignal stop() ...
      ...
   }
}
```

Out Signal

A method is generated for each Out Signal. This method has the same name as the signal. The method takes as a parameter any data associated with the signal. The data type may be one of the following:

- If void, then no data is passed as a parameter.
- If a Java class, then an instance of that class or a subclass may be passed as a parameter.

- If a Java primitive type, then a value of that type may be passed as a parameter. In this case, the generated method will create an instance of the appropriate Java wrapper class and pass this instance to the messaging service.
- If blank, then the parameter is optional. If present, then it may be an instance of any Java class.

The method returns an object that provides methods that allow access to the services provided by the messaging service for outgoing messages.

The code generated for the Out Signals (not including the symmetric signals) in Figure 25 on page 56 is:

```
public class NewProtocol1
{
   public static class Base extends ...
   {
      public final OutSignal start( Class1 data ) ...
      public final OutSignal stop( Class1 data ) ...
   }
   public static class Conjugate extends ...
   {
      public final OutSignal bye( java.lang.Object data )
      public final OutSignal hello( int data ) ...
      public final OutSignal test() ...
   }
}
```

Symmetric Signals

If a signal is used as both an In Signal and an Out Signal, and the Out Signal does not take any data, then the methods generated for the signal would only be different by return type. This is not valid in Java. As a result, the two methods that would be generated are combined and return an object that could be used to access services for both incoming and outgoing messages.

The code generated for the symmetric signal in Figure 25 on page 56 is:

```
public class NewProtocol1
{
   public static class Base extends ...
   {
      public final SymmetricSignal ack() ...
   }
   public static class Conjugate extends ...
   {
      public final SymmetricSignal ack() ...
   }
}
```

Capsules

Capsules are extensions of classes that add ports and capsule roles to enhance modeling the structure of the classes. Capsules also add state machines to handling modeling of the behavior of the classes. The generated code for capsules relies on the UML Services Library to provide a framework for these modeling concepts.

Capsule Roles

Capsule roles are generated as attributes of the Services Library provided type **CapsuleRole**. Capsule roles are used by the Frame service for operations that involve the instantiation, destruction, and importation of capsules referenced by the capsule roles.

For example, to incarnate an optional capsule role, the code generated is:

```
import com.rational.rosert.CapsuleRole;
import com.rational.rosert.Frame;

public class MyCapsule ...
{
   protected static final CapsuleRole role;
   public Frame.Base frame;

   public void f()
   {
      frame.incarnate( role );
   }
}
```

Ports

Ports are generated as attributes and are typed based on the protocol class of the port and the role it partakes in that protocol, that is, either Base or Conjugate. The port can then be used to send messages or to access messaging services for incoming messages.

For example, to send an new message (start) and to recall an incoming message (test), the code generated is:

```
public class MyCapsule
{
   public MyProtocol.Base port;

   public void f()
   {
      port.start( new Class1( ... ) ).send();
      port.test().recall();
   }
}
```

State Machine

The state machine depicted in the collection of state diagrams associated with a capsule is translated by the Java code generator into a number of methods on the class generated for the capsule.

There are three main categories of methods:

- User code segment methods
- Chain methods
- rtBehavior methods

User code segment methods

User code segments may be added for the following state machine elements:

- Transitions for both Actions and Guard code
- Choice points
- State Entry and Exit Actions

These methods are given a name by the Java code generator. They should not be directly called from user code.

The return type for these methods is **void** except for:

- Guard code on triggers that return a boolean value which, if true, allows the transition to be executed.
- Choice points that return a boolean to determine whether the outgoing True or False transition is to be taken.

Transitions and choice points have access to two parameters that are passed to the method: `rtdata` and `rtport`.

`rtdata`

The `rtdata` parameter provides access to the data in the message in a type safe manner. The generation of this parameter is enabled or disabled by the **GenerateDataParameter** property (**Transition**, **RTJava**) or (**Choice Point**, **RTJava**). The type of this paramter is calculated by the Java code generator to be the common superclass of the data type on all possible signals that may trigger the transition or choice point.

It is possible for a subclass of a capsule to introduce additional possible signals. The code generator will detect if these additional signals would invalidate the parameter in the superclass, and will raise an error. The `rtdata` parameter must be disabled in the superclass to remove the error.

`rtport`

The second parameter, `rtport`, works the same way. It is controlled by the **GeneratePortParameter** property (**Transition**, **RTJava**) or (**Choice Point**, **RTJava**). The type of this parameter is the common superclass of the protocol role type on all possible signals that may trigger the transition or choice point.

For example, the code generated for a transition is:

```
public void transition1_t1( Integer rtdata, MyProtocol.Base rtport )
{
   // User code here
   System.out.print( "In transition t1 with data = " );
   System.out.println( rtdata );
}
```

Chain methods

These methods control the invocation of user code segments and the entering and exiting of states. They also inform the services layer of the current activity in the state machine for debugging. For transitions and choice points, the data and port parameters are cast to their appropriate types given the collection of triggers that can cause the chain to execute.

For example, the code generated for a chain method is:

```
protected void chain1_t1()
{
   rtChainBegin( 1, "t1" );
   rtExitToChainState( 1 );
   rtTransitionBegin();
   transition1_t1( (Integer)rtGetMsgData(), (MyProtocol.Base)
   rtGetMsgPort() );
   rtTransitionEnd();
   rtProcessHistory();
}
```

rtBehavior methods

This is the main method that determines the handles a message dispatch by calling a specific chain method given the current state, and the signal and port of the incoming message.

Special Overrideable Capsule Class Operations

There is a special operation that is defined as virtual methods in the root capsule class (com.rational.rosert.Capsule), and that can be overridden in your capsule class to handle error conditions.

rtDestroy

Even though java has automatic garbage collection, it is often more efficient to release resources as early as possible. This is the mechanism whereby capsules can ensure that resources are not held longer than necessary.

rtUnexpectedMessage()

This operation is called when the **rtBehavior** method is unable to determine how to handle the current message.

Build Overview

Build Process Flow

Figure 26 Build-Time Process Flow Diagram

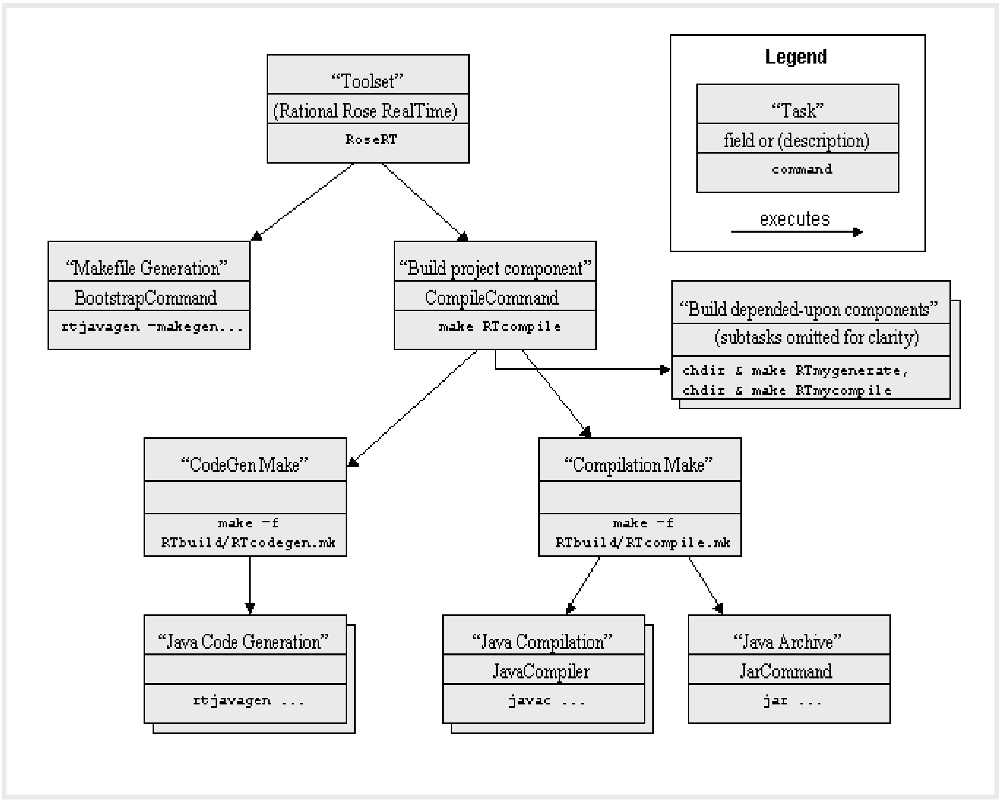


Figure 26 shows the general process flow while executing a **Build > Compile** for an RTJava Project component. Tasks are launched in a top-down, left-to-right traversal. The tasks typically require success to continue traversing the process flow. The subtasks of **"Build depended-upon project components"** are trimmed for diagram clarity, but it would also have its own **"CodeGen Make"** and **"Compilation Make"** subtasks.

Executing a **Build > Generate** would trim the **"Compilation Make"** task. Executing a **Build > Rebuild** would prepend the **"Makefile Generation"** with a task to run the **"CleanAllCommand"** which would clean all RTJava Project components.

For further information, see *RTJava Component Properties* on page 107.

Required Third-Party Tools

In order to build and run Java programs using Rational Rose RealTime, you will need to install and configure the following third-party tools:

- A Make utility, such as Microsoft **nmake**, **Gnu_make**, or the make distributed with any UNIX platform
- A Java compiler, such as **javac** from Sun's Java 2 SDK 1.3 or **jikes** from IBM's Open Source Java project
- Optionally, a Java archiver, such as **JAR** from Sun's Java 2 SDK 1.3.
- A Java Run-Time Environment, or Virtual Machine, such as **java**, **kvm** (for CLDC) or **midp**.

You should verify that these tools are available from a command prompt before starting Rational Rose RealTime.

Components

There are two component types supported in Rational Rose RealTime Java:

- RTJava Project
- RTJava External Project

RTJava Project Component

The RTJava Project component type is intended for specifying how to build a collection of classes. An RTJava Project typically references one or more Java classes and specifies how to generate and compile code from those classes. An RTJava Project produces one or more "deliverable" artifacts, such as a Java Archive (JAR) file or a set of Java Class files.

An RTJava Project component may depend on other RTJava Project components or on RTJava External Project components. When you build an RTJava Project component, you also build all of the RTJava Project components upon which it depends, directly or indirectly.

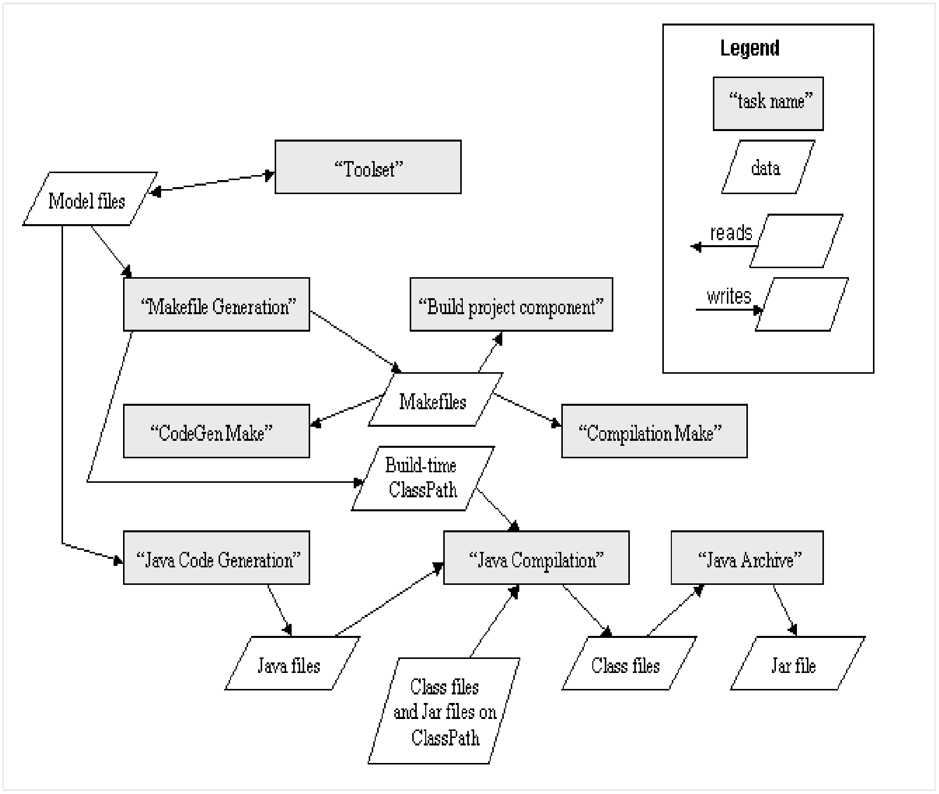
RTJava External Project Component

The RTJava External Project is intended for re-using modeling elements and their previously-generated build artifacts. You cannot perform any build activities on RTJava External Projects because they have been built "externally" for you.

An **RTJava External Project** references zero or more Java classes, and specifies zero or more additions to the **ClassPath**. The referenced classes do not need to be referenced by a depending RTJava Project and will not need to be generated. The specified **ClassPath** additions are passed to the Java compiler when building a depending RTJava Project.

The intended use of the **RTJava External Project** component is to be depended upon, directly or indirectly, by an RTJava Project component. An **RTJava External Project** may depend upon other **RTJava External Projects**; if a RTJava Project directly depends on the former component, it indirectly depends on the latter components. Consequently, the **RTJava External Project** component type provides a mechanism to manage and reuse build artifacts.

Figure 27 Build-Time Data Diagram



The above diagram shows the general data flow between the tasks identified in Figure 26 on page 64. **"Build depended-upon project components"** is omitted for clarity, but the resulting **"Class/Jar files from other project components"** is not.

"Class/Jar files from external sources" are referenced in the model by adding their **ClassPath** to an RTJava External Project. For further information, see *RTJava External Project* on page 115.

Build Details

Generated Makefile Patterns

Java compilers are not typically Makefile-driven. Java adopts a simple strategy of "compile everything always". Unfortunately, this can become a problem as the size of your model and the number of generated Java classes increase.

The use of Makefiles allows projects to generate only where the model has changed, and to compile only where the Java code has changed. Consequently, it is a more scalable solution for large models.

Makefile Generation

The **BootstrapCommand** that generates the Makefiles is always executed before every Build.

Default Directory Layout

The default **OutputDirectory** for a RTJava Project component is **$modelDir/$compName** that is a directory named after the component, and is a subdirectory of the directory where the model is found. After building two components, **MyComponent1** and **MyComponent2,** a simplified model directory may appear as follows:

```
/my_home/my_models/
   hello_world.rtmdl
   hello_world/          # contains Child Units, if applicable
   MyComponent1/         # Output Directory for MyComponent1
   MyComponent2/         # Output Directory for MyComponent2
```

Note: You should specify component Output Directories that do not conflict with the Child Units directories or the Output Directories of other components in this model, or in other models stored in the same directory.

After a build, you can find the generated files under each component's `Output Directory`. By default, the compiled Java class files and the Java archive (if applicable) will also be located in the `Output Directory`. The files included in the `Output Directory` are listed in the following table:

Makefile	The Component Makefile.
*.java	Generated Java class.
*.class	Compiled Java class or nested class.
MyComponent1.jar	The (optional) Java Archive.
RTclasspath.pl	The generated class-path needed for run-time.
RTbuild/RTcodegen.mk	The CodeGen Makefile (includes *.gmk).
RTbuild/*.gmk	Dependency list for each generation rule.
RTbuild/*.gtg	Target for each generation rule.
RTbuild/RTcompile.mk	The Compilation Makefile (includes *.cmk).
RTbuild/*.cmk	Dependency list for each compilation rule.
RTbuild/*.ctg	Target for each target rule.
RTbuild/RTcompile.pl	Lists all java files and runs a command.

The package property **JavaPackage** flag can be used to scope the namespace of the *.java and *.class files (as well as the *.gmk, *.gtg, *.cmk and *.ctg files). For further information, see *JavaPackage* on page 106

Guidelines for Efficient Incremental Builds

In order to generate efficient, incremental builds, the generated Makefile patterns for code generation and compilation accommodate the following guidelines:

1 Note which files were read (and read as few files as possible). A subsequent Make will know the file changes that should trigger the rule again.

2 Only write the output files if the content has changed. This is particularly relevant if there are "downstream" Makefile rules that might be triggered. For example, a single unnecessary regeneration may trigger a number of unnecessary recompilations.

3 If the command is successful, always update the timestamp of the target. A subsequent Make will know when this rule was last successfully executed, and will have a timestamp to compare against the input dependencies in guideline #1. A separate target file is necessary if none of the output files will reliably change due to guideline #2.

4 If the command fails, never update the timestamp of the target. Otherwise, a subsequent Make will assume this rule was successful and skip it.

These guidelines do not apply to the default **BootstrapCommand** because it directly runs the Makefile Generation without evaluating the need to do so first. This is the default for simplicity, since it is error-prone to evaluate the trustworthiness of the previous Makefile Generation without first reading the artifacts of the previous Makefile Generation.

Code Generator Behavior

Command-Line Arguments

The code generator has the following usage synopsis:

```
rtjavagen -model <model_file> -component <component_ name>
   [-debug] [-version] [-makegen | -class <class_name>]
```

The only user serviceable options are the `-debug` and `-version` options. For information on adding these to your code-generation command-line, see *Passing options to the code generator* on page 111. The other arguments are discussed here for informational purposes only:

`-model <model_file>:` specifies the file-path of the `.rtmdl` file.

`-component <component_name>:` specifies the fully qualified UML name (typically enclosed in double-quotes) of an RTJava Project component.

`-debug:` turns on verbose debugging information, such as environment variable expansion and file reads and writes.

`-version:` echos the version number of the code generator to standard output.

`-makegen:` instructs the code generator to generate the Makefiles for the given component, and the RTJava Project components upon which it depends.

`-class <class_name>:` instructs the code generator to generate the Java file for the specified class in the model. The specified `class_name` should be a fully qualified UML name, and should be referenced by the component.

If neither the `-makegen` and `-class` arguments are provided but the `-model` and `-component` arguments are provided, the code-generator will generate all of the Java files for the specified component.

Efficient Incremental Builds During Code Generation

The code generator produces a Generation Makefile dependencies (*GMK)* file when generating Java files to indicate to the next invocation of Make when the generated Java files should be regenerated.

The code generator produces a Generation Target (*GTG)* file upon successfully generating the Java files for all requested classes without error. If a Java file already existed from a previous build, it is only rewritten if the content has changed. If an error occurred, neither the GTG file nor the Java files are written.

Compiler Behavior

Invoking the Compiler

The Java compiler is specified from the **JavaCompiler** property on the RTJava Project component specification. For more information, see *JavaCompiler* on page 108. The implicit usage pattern is such that the property can be followed by one or more Java files (for example `*.java` or `subdir/Main.java`) to compile those files. The Java files are not provided in the **JavaCompiler** property: they are provided by the Makefile Generation.

A convenient feature of most Java compilers is an option to tell the compiler to put Java class files in a specific directory. Typically this is given by `-d <output_directory>`, and this is reflected in the default property value for invoking the compiler. If the component's **OutputClassDir** changes, the `-d` argument must reflect the change.

Class Path

The Java compiler's `-classpath` argument is constructed during Makefile generation by examining the component's dependencies and appending the user's **CLASSPATH** environment variable at build time. Do not modify the **ClassPath** by specifying the `-classpath` argument in the **JavaCompiler** property unless you know your Java compiler will concatenate the results. Some compilers will replace one argument for the other.

Compiler Wrapper Script

The default compiler wrapper script `$ROSERT_HOME/RTJava/scripts/rtcomp.pl` performs the following methods:

- Uses the compiler's `-verbose` output to produce a CMK file to discover compilation dependencies.
- Converts compiler-generated errors and warnings into Generic Error Stream messages. For more information, see *Build Errors* on page 72.
- Uses the compiler's return code to produce a CTG file upon successful compilation.

Compile All Script

The `RTbuild/RTcompile.pl` is a Perl script generated during Makefile Generation. It lists all of the generated Java files and can pass them to a Java compiler while circumventing command-line length limitations.

Efficient Incremental Builds During Compilation

The compiler wrapper script produces a Compilation Makefile dependencies (*CMK)* file when generating Java class files to indicate to the next invocation of Make when the compiled Java class files should be recompiled.

The compiler wrapper script produces a Compilation Target (CTG) file upon successfully compiling the requested Java classes without error. If an error occurred, neither the GTG file nor the Java files are written.

Some Java compilers (including javac in J2ME) do not write class files incrementally. They will write new Java class files even when the new content is the same as the old content. To achieve incremental compilation, you must tell the Compilation Make to compile every Java file individually. For further information, see *Incremental compilation and individual Java file compilation* on page 112.

Note: Individually compiling every file will slow down non-incremental builds (that is, initial builds and rebuilds).

Build Errors

The error handler of the toolset allows you to quickly locate the modeling element to which the error or warning message is attributable. Error and warning messages are summarized on the **Build Errors** window of the **Output Window** after every build.

Click on a message to view and correct the element (such as a transition action code). Double-click on a message to browse the element's specification, and to correct it as required.

Converting Compiler Errors into Build Errors

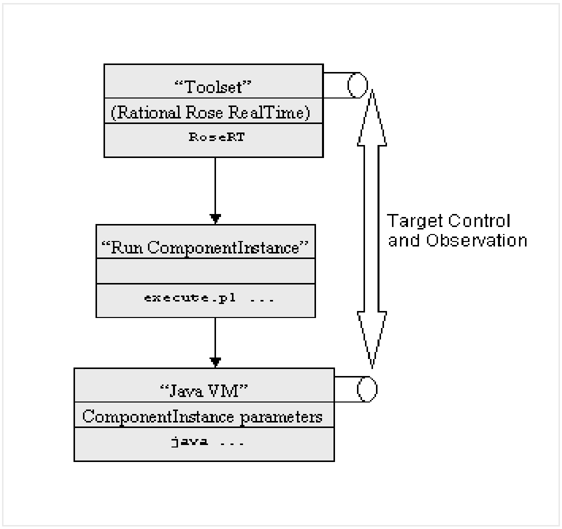
The compiler, or any other third-party tool, is not aware of the model, and produces messages in its own format. The toolset cannot predict the format of the compiler messages, and does not normally read the generated code. To handle compiler-generated messages in a way that is flexible to various compiler vendors, a compiler wrapper script translates compiler warnings and errors from the compiler into a format that can be captured by the toolset. The format of the output of the compiler wrapper script is Generic Error Stream (GES) format, and the GES messages are passed via standard output to the toolset.

The code generator is aware of the model, and already produces messages in the Generic Error Stream format. The code generator also provides tags that help the compiler wrapper script locate the right location and context within the model of a particular compiler message.

The default compiler wrapper script is a Perl script, `$ROSERT_HOME/RTJava/scripts/rtcomp.pl`. Advanced users can test changes to their own compiler wrapper script by overriding the `JAVAC_WRAPPER` Make macro in the **CompilationMakeInsert** field. The compiler wrapper script performs other tasks and should be modified carefully. For further information, see *Compiler Wrapper Script* on page 71.

Run-Time Overview

Figure 28 Run-Time Process Flow Diagram

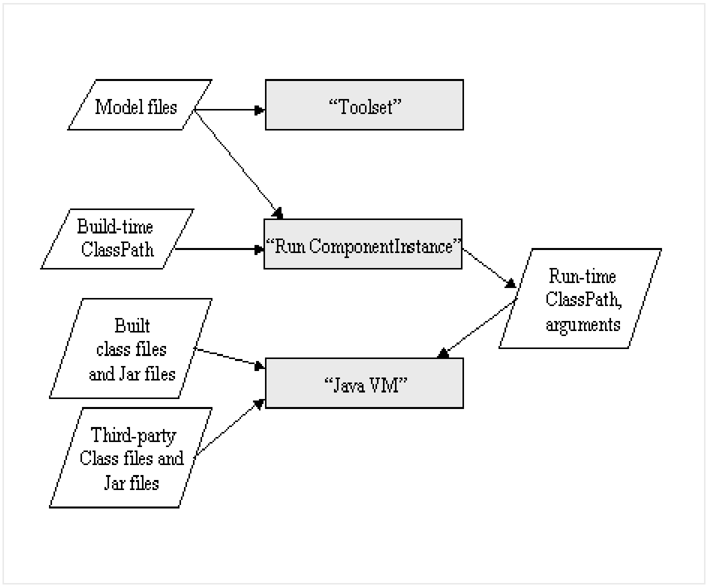


The above diagram shows the general process flow of running a component instance.

Note: Component instances are created under the **Deployment View** of the model.

The component instance can be configured to run various VMs, for example `java.exe`, `kvm.exe` or `midp.exe`.

Figure 29 Run-Time Data Diagram


Model files
"Toolset"
Build-time ClassPath
"Run ComponentInstance"
Run-time ClassPath, arguments
Built class files and Jar files
"Java VM"
Third-party Class files and Jar files

The above diagram shows the data flow diagram of running the component instance from Figure 28. This is particularly relevant to understand how the run-time **ClassPath** is derived since it may require more information than the compile-time **ClassPath**.

Java UML Services Library

4

Contents

This chapter is organized as follows:

Java UML Services Library Framework

The classes and data types defined in the Java UML Services Library provide an application framework in which your Rational Rose RealTime application will run.

The framework defines the skeleton of a real-time application:

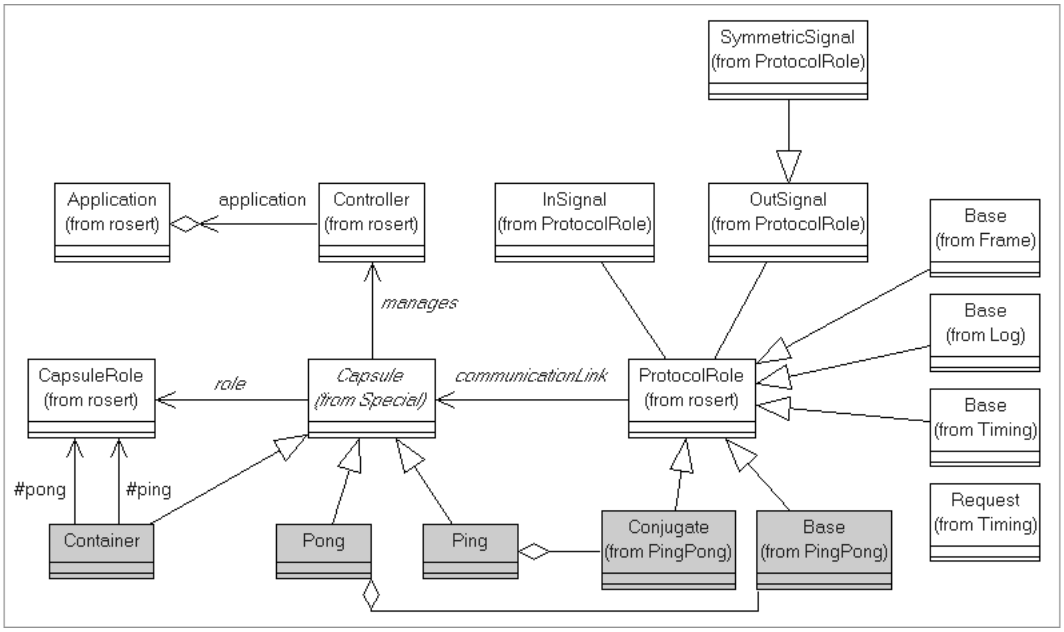
- Messaging
- Timing
- Dynamic structure
- Concurrency
- Event-based processing

As a Rational Rose RealTime developer, your job is to fill in the rest of the skeleton, that is, the classes, capsules, and protocols that are specific to your system.

The capsules, capsule roles, protocols, ports and classes in a Rational Rose RealTime model will be generated to Java code, and integrated into the Java UML Services Library framework.

The following class diagram shows how a set of generated model elements integrate within the framework. The white boxes are predefined classes in the Java UML Services Library and the grey boxes are classes generated from the *Framework Sample Model* on page 76.

Figure 30 Java UML Services Library Framework



This simplified class diagram illustrates:

- The high level view of the Java UML Services Library classes and their relationships.
- How your application level modeling elements (grey boxes) integrate into this framework
- The relationships between your modeling elements and the framework.

Framework Sample Model

This model was used as an example of how elements from a model integrate into the Java UML Services Library Framework. The gray boxes in the above diagram show classes generated from the model illustrated in Figure 31 and Figure 32:

Figure 31 Ping Pong Model Class Diagram

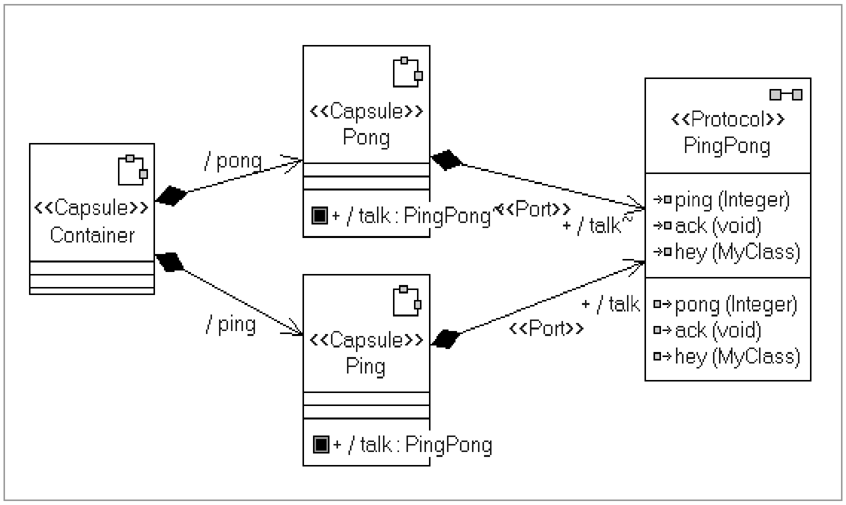
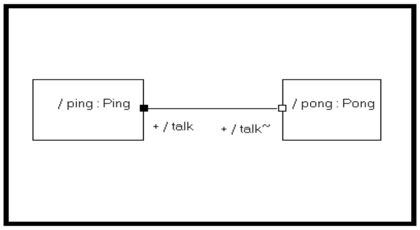


Figure 32 Container Capsule Structure Diagram



Message Processing

Events and Messages

An *event* is a message arriving on a capsule's port. Message-based communication is the basic mechanism for communication between capsules. Both synchronous and asynchronous communication are supported allowing a variety of different interaction semantics to be represented. Messages are also used by the Java UML Services Library to communicate with the capsules in the model.

Information about a message can be accessed through the use of the following methods:

- rtGetMsgSignal
- rtGetMsgPriority
- rtGetMsgData
- rtGetMsgPort
- rtGetMsgPortIndex

Capsule Processing

The heart of the Java UML Services Library is a controller object that dispatches messages to capsules. The controller object takes the next message from the outstanding message queue, and delivers it to the destination capsule for processing. When the message is delivered, the controller object invokes the destination capsule's state machine to process the message.

Control is not returned to the Java UML Services Library until the capsule's transition has completed processing the message. Each capsule processes only one message at a time. It processes the current message to the completion of the transition chain (for example, guard, exit, transition, choice point, exit, and entry), returns control to the Java UML Services Library, and waits for the next message. This is referred to as *run-to-completion* semantics. Typically, transition code segments are short, and result in rapid handling of messages.

Message Processing

The Java UML Services Library runs in a loop executed by a system controller object. This loop waits for messages and delivers them, one at a time, to capsules for processing. Each physical thread in a Rational Rose RealTime model has its own controller object, and its own set of message queues. Messages that cross thread boundaries are placed in special queues and are picked up by the receiving thread in its processing.

The model is first initialized by queueing a special system-level message (the initialization message) for the top-level capsule. After the initialization message is queued, the controller object enters the main processing loop (the `run` function). In `run`, the controller object takes the highest priority message from the message queues, delivers it to the receiver capsule, and invokes that capsule's behavior to process the message. During start-up, the highest priority message on the queue of the main thread will be the initialization message. When a capsule processes the initialization message, the capsule's initial transition segment is executed.

When the capsule has completed processing a message, it returns control to the controller. The controller continues this loop until there are no more messages to be processed. At that point, it waits for a message from another physical thread in the model, or for a timeout to expire.

Threads

A capsule has its own logical thread of control, and can operate independently of other capsules as if each capsule had its own dedicated processor. These independent capsules synchronize to perform higher-level scenarios through message-passing. One capsule sends a message to another capsule allowing the other capsule to update its state based on this outside stimulus. In practice, most Rational Rose RealTime models run on a machine with a single processor, or possibly in a distributed environment, with a few processors. In any case, there are almost always more capsules than processors. Thus, the capsules must share the processor in some manner.

Mapping Capsules to Threads

Rational Rose RealTime allows designers to make use of the underlying Java threads so that the processing of a capsule on one thread does not block the processing of capsules on other threads. Designers can specify the physical Java environment threads onto which the capsules will be mapped at runtime.

In a system with only one thread, there are situations where a single capsule transition can block other capsules from running (such as, if the capsule invokes a blocking system call). By placing some capsules in different threads, the designer can avoid the problems that arise from these situations, and can make better use of the underlying processor. For example, capsules that have excessively long processing times, and those with transitions that may block, should be placed on separate threads. Deciding which capsules need to execute in different threads is a matter for design consideration.

Note: Not every capsule should run on a separate thread. Most capsules can be in one thread, and the Java UML Services Library controller will invoke their behavior as messages arrive.

Capsules can belong to different logical threads. Logical threads are mapped to a set of concurrent physical threads defined by the user. No other capsules in a thread can execute until the currently executing capsule returns control to the main loop of that thread (except in the case of invoke). However, other capsules on other physical threads may be executing simultaneously.

The Java environment is responsible for switching control among active physical threads. The Java environment may preempt one physical thread in the middle of execution to switch to another physical thread. Each thread can be assigned a separate priority so that the designer has some control over the scheduling.

Framework Services

With Rational Rose RealTime, you develop your application in a high level language using state diagrams and structure diagrams. These elements are automatically converted to Java, and are placed in a framework that provides critical real-time system services.

The key to using the services provided by the framework is to understand how your application will integrate into the Java UML Services Library skeleton. The framework provides four main services to our application:

- *Log services:* a general purpose logging service.
- *Communication services:* the basic mechanism for using message-based communication via ports.
- *Timing services:* provide general purpose timing facilities.
- *Frame services:* used to gain control over the dynamic structure of a model.

Services are explained by identifying the classes that are used to implement the service followed by a discussion of the general concepts related to the service.

Log Services

Implementation Classes

`Log.Base`

Concepts

The System Log is a stream of ASCII text in which system or application events can be recorded. Currently, all log output is directed to `java.lang.System.out`.

Execution speed is affected since each write to the log involves an output system call, which is a relatively expensive operation.

Communication Services

Implementation Classes

`ProtocolRole`

`ProtocolRole.InSignal`

`ProtocolRole.OutSignal`

`ProtocolRole.SymmetricSignal`

Concepts

This fundamental service provides most of the standard communication models prevalent in concurrent software system design, including asynchronous messaging, and rendezvous like synchronous inter-capsule communication.

The Communication Service is accessed by referencing a *port name* that will be an instance of a subclass of `ProtocolRole` with the appropriate operations. The port name is the user defined name of the port declared in the model. The named port is generated as a member of the capsule containing the port.

Depending on the multiplicity of the port, every named port may actually have a number of port instances associated with it. Each port instance is capable of sending and receiving messages, and is encapsulated within each `ProtocolRole` object.

A service request results in the creation of instances of `Message`. These messages are delivered by the Java UML Services Library to the ports at the other ends of the connections. They are eventually processed by the behavior of the capsules containing those ports.

Primitives

This service is used for passing messages between capsules in real time. Messages sent via this service are processed whenever the necessary CPU cycles become available.

A capsule instance accesses information in the message that was received through `rtGetMsg` operations.

When processing a message received at a particular port, the `rtGetMsgPortIndex` operation returns an index to the particular port instance that received the message. An `OutSignal.sendAt` to the port instance returned by `rtGetMsgPortIndex` results in a send to only that particular port instance. The communications services also provide a number of functions for dealing with replicated ports.

Asynchronous and Synchronous Communication

If an *asynchronous* send is used, the sending capsule will not block while the message is in transit. This mode is well-suited for high-throughput and fault-tolerant systems.

Conversely, if *synchronous* communication is desired, a blocking send can be used. During the invocation of the method, the sender (invoker) is blocked until a reply is received even if higher-priority messages arrive. At the other end, the receiver does not normally distinguish between synchronous and asynchronous communications but replies to either in the same way. In this way, the receiver is decoupled from the implementation decisions of its clients regarding which communication mode to use (that is, blocking or non-blocking). However, in practice, the receiver must know something about the expectations of the sender, and there are three restrictions that must be observed:

- The receiver must reply to messages that are synchronous with `rtport.signal(data).reply()` within the same transition.
- Circular invokes are not permitted. For example, if capsule A invokes capsule B, and capsule B tries to invoke capsule A, the invoke operation in B will fail with an exception.
- Invokes across thread boundaries are not permitted.

Order Preservation

Messages of equal priority that are sent along the same binding are delivered in the same order. This applies to both messages sent to capsules executing within the same thread, and for messages going to another thread.

Note: Such guarantees may not be available when capsules are in different processes, or when messages are sent on different bindings.

Message Loss

Messages have a high probability of being delivered to the receiving object but it is not guaranteed. For example, messages can be lost if they are sent through unbound ports, or if the destination capsule is destroyed dynamically. In distributed versions of this service, loss of messages can also be due to temporary resource depletion (such as no buffer space) or actual loss in the physical communications medium.

Minimal Overhead in Message Handling

This is due to the relative simplicity of the service, and its lack of any automatic form of acknowledgment or flow-control protocols.

Request-Reply

A special feature of the communications services is support for a *request-reply* communication model. These are message exchanges between a sender capsule and a receiver in which the specified reply is expected, handles the request, and responds within the scope of a single transition.

The communications services also support synchronous messaging (similar to a rendezvous). During a synchronous send or invoke, the sender is blocked until the receiver has processed the message and has sent back a reply. Run-to-completion semantics are enforced, and a synchronous invoke has the same semantics as a procedure call.

The receiver of an invoked message, or an asynchronous message with expected reply, must respond to it prior to the completion of message processing.

Message Priority

A *message priority* is interpreted as the relative importance of an event with respect to all other unprocessed messages on a thread. This is reflected in a bias towards higher-priority messages over lower-priority messages when scheduling CPU time. If two or more messages of different priority are queued and waiting to be processed, messages with a higher priority are usually processed before messages of lower priority. The slight ambiguity of this definition reflects the variability of scheduling policies due to the inherent non-determinism of distributed systems, as well as changing implementations.

In general, good designs should not be critically sensitive to a particular scheduling policy. The current Java UML Services Library scheduler uses simple priority scheduling so that messages at a particular priority level are not processed until all higher-priority messages on that controller have been processed.

Within a given priority level, the Java UML Services Library guarantees that messages will be processed in the order of arrival.

Note: In a distributed system, the order of arrival may not be the same as the order in which the messages were sent.

Message priorities do not imply interruption of the processing of the current event even if a newly-arrived message is of a higher priority. This is due to the run-to-completion semantics of transition (described above).

A user-level message has one of five priority levels associated with it. The following predefined symbols allow the user to specify the priority of a message by name:

- **`Priority.Panic`**: Highest priority available to users. Used only for emergencies.
- **`Priority.High`**: For high-priority processing.
- **`Priority.General`**: For most processing. The default.
- **`Priority.Low`**: For low-priority.
- **`Priority.Background`**: The lowest priority. Used for background-type activities.

Message priorities disrupt the temporal order of events, which often leads to implementation problems. For this reason, it is recommended that applications limit themselves to a single priority level. If priorities are used, avoid the high and low extremes of the range in order to save room for subsequent design changes.

In addition to these user-level message priorities, there are some *system-level priorities*. System-level priorities are higher than the highest user-level priority in order to guarantee the correct operation of Java UML Service Library routines.

Wired and Unwired Ports

Ports can be either *wired* or *unwired*. Wired ports are explicitly connected to other wired ports with connectors. Unwired ports are not connected during design: instead, they are dynamically connected at runtime. Unwired ports are bound to other unwired ports by a registered name.

Layer communication involves support for managing connections between unwired ports.

Published and Unpublished Unwired Ports

In the layered communication paradigm, unwired published (SPP) ports can only connect with unwired unpublished (SAP) ports, or vice versa.

Note: The terms SAP (Service Access Points) and SPP (Service Provision Point) are used to abbreviate `unwired [unpublished|published] port`.

A SAP cannot connect to another SAP, and a SPP cannot connect to another SPP. By convention, a SPP is the server side of a connection, and a SAP is the client. Some of the communication service operations are named with these abbreviations to differentiate SAP and SPP operations.

For any given *service*, there is one server (the SPP), and there may be many clients (the SAPs). A service is some functionality provided by the server capsule to the client capsules. The service is uniquely identified by name. There may exist many different server capsules, each providing a different service. Any given service (name) may have only one server (SPP) registered for it at any given time. Any other providers that attempt to register an SPP of the same name will be declined (that is, the registration will fail).

SPPs are often replicated, with their multiplicity specifying the maximum number of clients that can be bound to the server at runtime; otherwise, no SAPs can be bound. By default, a SAP or SPP is automatically registered under its reference name when the capsule containing that SAP/SPP is initialized.

Multiplicity may be changed dynamically at runtime with the `ProtocolRole.resize` operation. This may destroy bindings if multiplicity is reduced, and allow new bindings if it is increased.

Registration by Name

The basic element of layer communication is a generic name server. SAPs register to the layer service for binding to a SPP under a unique name. SPPs need also register to the layer service in order to publish its unique name for binding with SAPs.

All SAPs are bound to the first SPP that registered for binding under that name. If no SPP exists, the SAP registrations are queued (usually in order) waiting for the SPP to register. SAPs will be bound with the SPP up to the maximum multiplicity of that SPP. SAPs not bound will continue to be queued until an instance of the SPP becomes available due to a SAP deregistering, a SPP with a larger multiplicity registering, or the SPP is resized.

Registration String

A *registration string* is used to identify a unique name and service under which SAPs and SPPs will connect. The string has the following format:

```
[<service_name>:]<registration_name>
```

The first part of the registration string is case sensitive. The interpretation of the remaining registration string depends on the specified communication service.

For example:

```
name

service1:name

service2:name

service3://address/name
```

The default `service_name` is `""`. That is, "`name`" is the same as "`:name`".

Automatic Registration and Application Registration

SAPs and SPPs can be configured to be automatically registered with the layer service, or to be registered by the application using a name to be determined by the application at runtime. If automatic registration is chosen, the registration name must be supplied in the **Port Specification** dialog box, and the Java UML Services Library will register the name at startup. In the case of application registration, the SAP or SPP is registered at runtime by calling a communication service operation (such as, `ProtocolRole.registerSAP` and `ProtocolRole.deregisterSAP`) in the detail level code of a capsule. The same port may be registered under different names at different points in the model execution, and as either a SAP or a SPP.

Deferring and Recalling Messages

The Java UML Services Library enforces the reactive model of behavior by automatically putting a capsule into a receive mode between successive transitions. This means that there is no need for an explicit user-specified receive method. When a message is selected for processing, the Java UML Services Library wakes up the capsule and starts execution of the appropriate transition.

When a message is received, the capsule may decide to postpone the handling of this event for a later time. For example, the behavior may be in the middle of a complex sequence of state transitions when it receives an asynchronous request to handle a new sequence. Instead of trying to execute two sequences in parallel, it may be simpler to serialize them. To do this, the newly-received message must be held until the current event-handling sequence is complete, and it is then resubmitted. The Java UML Services Library allows messages to be deferred, and then recalled later.

Timing Services

Implementation Classes

`Timing, Timing.Request`

Concepts

The timing services provide users with general-purpose timing facilities based on both absolute and relative time. To access the timing services, you reference, by name, a timing port that has been defined on that capsule (that is, by creating a port with the pre-defined Timing protocol). Service requests are made by operation calls to this port while replies from the service are sent as messages that arrive through the same port. If a timeout occurs, the capsule instance that requested the timeout receives a message with the pre-defined message signal `'timeout'`. A transition with a trigger event for the timeout signal must be defined in the behavior in order to receive the timeout message.

Each request to the timer service for a timing event will return a handle to the request. This handle can be used to cancel the request.

Absolute and Relative Time

Two forms can be used to specify a timer request: *absolute time* and *relative time*. This service defines absolute time as elapsed time since some fixed point in the past. Relative time is expressed as a number of time units from the current time instant.

One Shot Timer

The one shot timer expires only once: after the specified time duration (that is, relative time), or at the specified time (that is, absolute time). If subsequent timeouts are required, the timer must be reset after each expiration.

Periodic Timer

If repeated timeouts are required, the extra time added for processing each timeout and requesting a new timer may cause some amount of drift in the timing. For example, requesting a timeout every 10 seconds will result in a timeout occurring every 10 seconds plus the amount of time required to process the timeout, and to reset the timer. Round-off of clock ticks may reduce or exaggerate this drift.

The periodic timer is set to timeout repeatedly after the specified duration until the timer is explicitly cancelled. It does not need to be reset after each expiration. Using the periodic timing service will generally provide more accurate timing than repeatedly resetting a one-shot timer.

Timing Precision and Accuracy

The precision of the timing service depends on the granularity of timing supported by the underlying Java environment. Although you can request timeouts with a granularity down to the millisecond, this does not mean you will get millisecond precision.

The service does not guarantee absolute accuracy. This means that intervals can take slightly longer than specified, and events scheduled for a particular time may in fact happen slightly after the actual time has occurred. The magnitude of the delay depends on many factors. However, unless the system is under severe overload, the discrepancy is usually not significant.

Frame Services

Implementation Classes

`Frame, Capsule, CapsuleRole`

Concepts

Capsule roles can be classified into three categories: fixed, optional, and plug-in. The latter two types of capsule references are used for dynamically changing structures.

The frame service provides the ability to instantiate and destroy optional capsules, to `plugIn` and `unplug` capsule instances to and from plug-in roles, plus a number of other functions. The rules and definitions governing the frame service can become quite involved. See the *Rational Rose RealTime Modeling Language Guide* for more details on the concepts behind dynamic structure.

A capsule role can contain a specific class if either that class is the role classifier, or the role has the substitutable property and the class is a subclass of the role classifier.

All the public ports of the class must have protocol roles that are compatible with the roles to which they are connected.

Optional Capsule Roles

The current number of existing instances of an optional capsule reference at any given time may be less than the cardinality specified for that capsule role. The rules governing the instantiation and destruction of optional capsules are as follows:

- An optional capsule can be instantiated as an instance of a particular class if it follows the compatibility rule (above).
- An optional capsule that is explicitly destroyed by the invocation of a method by the immediate container ceases to exist and does not appear anywhere.

Plug-In Capsule Roles

The following rules must be satisfied at runtime in order for a capsule instance to appear as a plug-in capsule role:

- The capsule instance cannot already be an aspect in the plug-in capsule reference.
- The class of the capsule instance must be compatible with the role.
- Capsules may not be imported across model boundaries. That is, a capsule cannot be imported across a process boundary, although it can be imported across a thread boundary within the same model.

All of the ports of the capsule instance to be bound in the destination plug-in role must have sufficient cardinality unbound.

Multiple Containment

Multiple containment allows you to represent capsule roles that are simultaneously part of two or more capsule collaborations. Specifying that two different capsule roles are actually bound to the same runtime instance can simplify the structure of the system by allowing it to be decomposed into different views.

Using Multiple Containment

In order to understand the need for this, it is necessary to examine the meaning of encapsulation in object-oriented design. When two or more capsule roles are placed together in a common capsule, the intent is to capture some user-defined relationship between these components. The simplest example of a relationship between objects is pure physical containment (such as, a shelf contains a particular card).

The type of relationships that exist in the software domain can be quite diverse. When two terminals are connected to each other in order to exchange information, they are involved in a call relationship. The object-oriented approach encourages us to capture such identifiable relationships as distinct objects.

Note: In physical terms, there is no real entity corresponding to a call; however, it may be useful to think of it in that way.

Once relationships such as these are captured in unique addressable objects, then it is possible to conceive of operations on such objects, such as terminating a particular call or adding another party to it. To the entities invoking the operations, the structure and implementation within such objects are typically of no concern. Following this line of thought leads us to conclude that these objects are in fact like any other software objects: entities with a set of externally accessible operations and an encapsulation shell that hides their internals. Therefore, capsules can be used to represent arbitrary user-defined relationships between their component capsules.

With this explanation of capsules, the need for multiple containment is more apparent. It is required to capture situations where a capsule role is involved in multiple simultaneous relationships with capsule roles in other containments.

Replicated Capsule Roles

Replication semantics are a function of the type of role that is replicated:

- All instances of a *fixed* capsule role are created automatically when the containing capsule is incarnated. The number of instances is equal to the cardinality.
- Instances of an *optional* replicated capsule role are created dynamically by the user at runtime using the frame service. The number of instances can vary from zero to the number specified by the cardinality. Any attempt to increase the number of instances beyond the cardinality will fail.
- *Plug-in* capsule roles are filled dynamically at runtime. The maximum number of instances that can be imported into the plug-in capsule role is limited by the cardinality of the role. Any attempt to increase the number of instances beyond the cardinality will fail.

Command-Line Model Observer 5

Contents

This chapter is organized as follows:

Overview

The Run Time System command-line observer provides a mechanism to allow UML for Real-Time models executing on the Run Time System to be debugged at the UML for Real-Time concept level. The Run Time System command-line observer does not provide source-level debugging. Source code debugging requires an external source-level debugger for Java, such as jdb.

Starting the Run Time Command-Line Observer

To use the command-line observer, first start the model executable from the toolset or from the command-line:

```
java <Java VM options> com.rational.rosert.DebugApplication
<TopCapsule> -obslisten=<portNumber>
```

This command can be followed by additional command-line arguments that the model may use.

You will see the following banner:

```
Rational Rose RealTime Java Target Run Time System

Release 6.30.C.00

Copyright (c) 2000-2001 Rational Software
```

When the model is started, run telnet to the machine that the model is running on, specifying the port that the model is listening on (`portNumber` from the command above). Once the telnet client is connected, press **ENTER.** You should see the Run Time System banner in the command-line observer session followed by the prompt.

You may connect additional command-line and toolset observers to a target that is already running.

Each observer has its own thread of control. If the logging of `stdout` is enabled, this may result in the model output being interleaved with the observer output. When using the observer, the threads related to timing should usually be detached. Other threads can be attached or detached as required.

Run Time Command-Line Observer Summary

taskName

Physical threads in the application are each identified by a `taskName`. Listing the threads in the application using the **tasks** command shows the `taskName` of each task. Use the `taskName` when referring to a particular thread for commands such as **attach**, **detach**, and **printstats**.

capsulePath

Each capsule instance has a unique `capsulePath`. The `capsulePath` indicates the capsule's position in the containment hierarchy. The supercapsule instance always has a path of `/`, and the top capsule `/0`. The instances contained in it are called: `/0/0`, `/0/1`, `/0/2`, and so on.

Replicated references are shown by a single Id. They can be identified individually by suffixing the role number with *n*, where *n* is the particular instance number (for example, `/0/5.1`). The index number of the first instance is `0`.

Note: The default replication factor is always `0.` For example, `/0/5` is exactly the same as `/0.0/5.0`.

The `capsulePath` is used in conjunction with the **info** command, and sometimes with the **system** command. The **system** command shows the `capsulePath` corresponding to each capsule.

portId

Each port is identified by a `portId`. `PortIds` are relative to the capsule where they are defined, and are unique only within this capsule class.

The `portIds` for a capsule class can be listed using the **info** command.

Thread commands

- **attach <taskName>**: Monitors a thread specified by `taskName`. `TaskNames` of the different physical threads in the model can be determined using the **tasks** command.
- **detach <taskName>**: Does not monitor a thread specified by `taskName`. Allows the thread to run freely.
- **newtasks <mode>**: Selects whether the newly created tasks should be started as attached or as detached.
- **tasks**: Prints the list of tasks (threads).

Informational commands

- **info <capsulePath>**: Shows information about the capsule instance specified by the `capsulePath`.
- **printstats <taskName>**: Prints the runtime statistics for thread `taskName`.
- **stats <taskName>**: Alias for **printstats <taskName>**.
- **system [<capsulePath> [depth>]]**: Lists all instantiated capsules in the system, starting with the one specified by `capsulePath` to a specific `depth`.
- **unwired**: Shows all registered unwired ports.

Tracing commands

- **log <category>**: Logs Java UML Services Library primitives.

  The argument `category` is one of the following: `all, communication, exceptions, frame, none, observability, stdout, timer, unwired.`

Control commands

- **continue**: Delivers messages until stopped by the user.
- **exit**: Terminates the application.
- **step [<n>]**: Delivers *n* messages.
- **stop**: Alias for **step 0**.

Other commands

- **close**: Closes the observer session.
- **help**: Prints help information.
- **?**: Alias for **help**.

Thread Commands

attach <taskName>

Allows the debugger to interact with the specified task (thread). `TaskName` must be one of the `taskNames` listed by the **tasks** command. When a thread is attached, messages within that thread are only processed when the **step** (or **continue**) command is given.

```
RTS Debug ->attach main


RTS Debug ->
```

detach <taskName>

Allows the thread specified by `taskName` to run freely. The debugger does not control the specified thread any longer. The thread processes all outstanding messages and then waits for new messages.

```
RTS Debug ->detach main


RTS Debug ->
```

newtasks <mode>

Specifies how newly created tasks should be started. If the `mode` is `attached`, the observers will have control over the new tasks. If the `mode` is `detached`, new tasks will be allowed to run freely.

```
RTS Debug ->newtasks attached


RTS Debug ->
```

tasks

Lists all threads in the model. Each thread is identified with a `taskName`. The main thread always appears in the list of threads. Any additional user-defined physical threads also appear in the list.

```
RTS Debug -> tasks


   0 main     attached

   1 taskA    attached

   2 taskB    detached

   3 taskC    attached

   4 task5    detached


RTS Debug ->
```

Informational Commands

info <capsulePath>

The **info** command displays information about a capsule instance specified by `capsulePath`. The **info** command displays:

- the name of the capsule class for the identified instantiation
- the role name (from the container)
- the current state of the capsule
- a list of ports, components and attributes.

Note: Ports listed are identified by an Id number.

```
RTS Debug ->info /0/3

ClassName: FiveStates
RoleName : fiveStates
State    : hasEntryExit

Relay Ports:
   0: relay(0) (InfoProt$Base)
   1: relay(1) (InfoProt$Base)

End Ports:
   0: timer (protected unregistered)
   1: log (protected unregistered)
   2: frame (protected unregistered)
   3: prot2 (protected wired)
   4: prot1 (protected wired)
   5: info (public wired)

Components:
   0: echo1
   1: echo2

Attributes:
   0: boolean BeenThere = false
   1: boolean DoneThat = true
   2: java.lang.Integer Token = 49
   3: int GotResp = 2

RTS Debug ->
```

printstats <taskName>

Prints information about the number of queued messages and a breakdown of these messages by priority. The alias **stats** is mapped to this command.

RTS Debug ->printstats main

```
main


messages        queued
   Synchronous     0
   System          0
   Panic           0
   High            0
   General         1
   Low             0
   Background      0
   Total           1


RTS Debug ->
```

stats <taskName>

Alias for **printstats <taskName>**.

system [<capsulePath> [<depth>]]

The **system** command lists all the active capsules in the system, starting with capsule, specified by `<capsulePath>` (default: / = the supercapsule) and `depth` `(default: 0 = all)` levels down.

Both the parameters `capsulePath` and `depth` are optional; however, if you also give the `depth` parameter, you must also give the `capsulePath` parameter.

Each capsule is displayed in the following form:

```
roleName : className (type) capsulePath [[more]]
```

Containment is indicated by indentation, and one leading dot for each containment level.

Example 1

The supercapsule is listed first, followed by all the capsule instances in its decomposition:

```
RTS Debug ->system

root : com.rational.rosert.SuperCapsule (system,fixed) /
. application : TopCapsule (optional) /0
. . capsuleA : CapsuleA (optional) /0/0
. . capsuleB : CapsuleB (optional) /0/1
. . capsuleC : CapsuleC (optional) /0/2
. . capsuleC : CapsuleC (optional) /0/2.1
. . fiveStates : FiveStates (optional) /0/3
. . . echo1 : Echo1 (optional) /0/3/0
. . . echo2 : Echo2 (optional) /0/3/1
. . capsuleD : CapsuleD (fixed) /0/4

RTS Debug ->
```

Example 2

We start with a different capsule:

```
RTS Debug ->system /0/3

fiveStates : FiveStates (optional) /0/3
. echo1 : Echo1 (optional) /0/3/0
. echo2 : Echo2 (optional) /0/3/1

RTS Debug ->
```

Example 3

We start with a different capsule, and also limit the depth to `1` level:

```
RTS Debug ->system /0/3 1

fiveStates : FiveStates (optional) /0/3 [...]

RTS Debug ->
```

The `[...]` message after the capsule means that the capsule in question has contained capsules that were not displayed since the supplied `depth` parameter limited the output.

unwired

Lists all registered unwired ports.

```
RTS Debug ->unwired

name:  protUnwired
   SAP: application[0]/protUnwired[0]
   SPP: echo2[0]/protUnwired[0]

RTS Debug ->
```

Tracing Commands

log <category>

The **log** command turns `ON` the logging of system services. UML-RT events are displayed as they happen according to the specified category.

The categories are: `all, communication, exceptions, frame, none, observability, stdout, timer, unwired.`

Events that will be logged are:

- all: everything
- communication: message delivery
- exceptions: exceptions
- frame: incarnate, destroy, plugIn, unplug
- none: stop logging
- observability: run status change, task attaching/detaching
- stdout: all log events that normally go only to stdout
- timer: timeout, cancel timer
- unwired: register/deregister unwired ports

The categories are additive. That is, the log timer followed by log frame will cause the command-line observe to log both timer and frame events.

Each message log shows the direction of the message, the receiving capsule (the 'to' capsule), the sending capsule (the 'from' capsule), and the data. The form of each message log is as follows:

```
message: signal (Priority)
   to      capsule[index](Class)<state>.portName[index]
   from    capsule[index](Class)<state>.portName[index]
   data    (dataType) dataValue
```

An example of message trace is shown below:

```
RTS Debug ->log communication

RTS Debug ->step 2

RTS Debug 2>

message:  sig_out (General)
   to    application[0](TopCapsule)<TOP>.portC[0]
   from  capsuleC[0](CapsuleC)<TOP>.port[0]

message:  sig_out_payload(General)
   to    application[0](TopCapsule)<TOP>.portA[0]
   from  capsuleA[0](CapsuleA)<TOP>.port[0]
   data  (java.lang.Integer) 17

RTS Debug ->
```

Control Commands

continue

Delivers unlimited number of messages.

exit

Exits the process. If you have logs turned ON, you may notice a sequence of cancellation/stop messages before the process is exited.

step [<n>]

Delivers *n* messages in the model. If *n* is omitted, the default is `1`.

stop

Alias for **step 0**.

Other Commands

close

Closes the observer session. The application and other observers continue running.

help

Prints help information.

?

Alias for **help**.

Model Properties Reference 6

Contents

This chapter is organized as follows:

RTJava Specific Properties

This chapter provides a guide to using RTJava specific properties for elements in the Logical View and the Component View.

For an example of class properties, see Figure 8. For an example of operation properties, see Figure 7.

RTJava Properties

These are properties that control the Java code that is generated for the model elements.

Class

A class within the model is generated as a Java class. The model's class and any nested classes within it are typically generated to a single Java file.

JavaStatic

Specifies whether the class is declared with the `static` modifier.

JavaFinal

Specifies whether the class is declared with the `final` modifier.

JavaStrictfp

Specifies whether the class is declared with the `strictfp` modifier.

ClassFileHeader

Specifies text that is inserted at the top of the Java file in which this class is generated.

StaticInitializerHeader

Specifies code that is placed in a static initializer at the top of the class definition.

StaticInitializerFooter

Specifies code that is placed in a static initializer at the bottom of the class definition.

InstanceInitializerHeader

Specifies code that is placed in an instance initializer at the top of the class definition.

InstanceInitializerFooter

Specifies code that is placed in an instance initializer at the bottom of the class definition.

Attribute

An attribute within the model is generated as a Java field.

JavaVolatile

Specifies whether the attribute is declared with the `volatile` modifier.

JavaTransient

Specifies whether the attribute is declared with the `transient` modifier.

InitializationCode

Specifies code that is placed in an (instance) initializer for the attribute.

Association End

An association end within the model is generated as a Java field if the association is navigable to that end.

JavaFinal

Specifies whether the association end is declared with the `final` modifier.

JavaVolatile

Specifies whether the association end is declared with the `volatile` modifier.

JavaTransient

Specifies whether the association end is declared with the `transient` modifier.

InitialValue

Specifies the initial value for the association end. It is assigned to the association end in its definition.

InitializationCode

Specifies code that is placed in an initializer for the association end.

NameQualification

Specifies whether the type of the association end is specified using its fully qualified name or whether an `import` statement should be generated and only the short name of the type is used.

Operation

An operation within the model is generated as a Java method.

JavaFinal

Specifies whether the operation is declared with the `final` modifier.

JavaNative

Specifies whether the operation is declared with the `native` modifier and without a code body.

JavaStrictfp

Specifies whether the operation is declared with the `strictfp` modifier.

JavaThrows

Specifies a comma separated list of exceptions that can be thrown by this operation.

Generalization

A generalization relationship within the model may be viewed as the inverse of a specialization relationship. A specialization within the model is generated as a Java class that inherits from, or `extends`, another Java class.

NameQualification

Specifies whether the parent class of the generalization is specified using its fully qualified name or whether an `import` statement should be generated and only the short name of the type is used.

Package

A package within the model is a logical grouping of model elements within the model. Packages are not explicitly generated, but they can affect how their contents are generated.

JavaPackage

Specifies whether the package is a Java package. This affects the namespace for the classes contained in the package and, transitively, all of the subpackages.

Transition

GenerateDataParameter

Controls whether or not the **`rtdata`** parameter for the transition function is generated. **`rtdata`** is the data parameter of the message cast to the common superclass for all signals that may cause the transition to execute. This should be disabled if the **`rtdata`** parameter is not used by the user code, or if the Java code generator detects a situation and raises an error where the parameter would be invalid.

GeneratePortParameter

Controls whether or not the **`rtport`** parameter for the transition function is generated. **`rtport`** is the port that the message arrived on cast to the common protocol role class for all signals that may cause the transition to execute. This should be disabled if the **`rtport`** parameter is not used by the user code, or if the Java code generator detects a situation and raises an error where the parameter would be invalid.

Choice Point

GenerateDataParameter

Controls whether or not the **`rtdata`** parameter for the choice point function is generated. **`rtdata`** is the data parameter of the message cast to the common superclass for all signals that may cause the choice point to execute. This should be disabled if the **`rtdata`** parameter is not used by the user code, or if the Java code generator detects a situation and raises an error where the parameter would be invalid.

GeneratePortParameter

Controls whether or not the **`rtport`** parameter for the choice point function is generated. **`rtport`** is the port that the message arrived on cast to the common protocol role class for all signals that may cause the choice point to execute. This should be disabled if the **`rtport`** parameter is not used by the user code, or if the Java code generator detects a situation and raises an error where the parameter would be invalid.

RTJava Component Properties

There are two supported component types:

- `RTJava Project`
- `RTJava External Project`

RTJava Project

An `RTJava Project` component allows you to specify build instructions for a set of UML classes. With this component type, you can generate Java files, compile Java class files, or produce a single Java Archive (.jar) file.

BuildJar Flag

If this flag is set, the component exports a .jar file containing a set of Java class files produced by this component. If this flag is not set, the component exports a set of Java class files.

OutputJarFilepath

Specifies the absolute file path of the .jar file produced, including the file name and extension. It has no meaning if the **`BuildJar`** flag is not set.

JarCommand

Specifies the command to create the .jar file from the compiled Java classes. The field has no meaning if the **`BuildJar`** flag is not set.

OutputClassDir

Specifies the absolute file path for the classes created. Java class files are typically produced in the specified directory whether the **`BuildJar`** flag is set or not set.

If the **OutputClassDir** is the same as the **OutputDirectory** for this component, then all Java and Java class files for this component will share the same directory structure.

If the **OutputClassDir** is the same as the **OutputClassDir** of other components, those Java class files will share the same directory structure, and will only require one addition to a client's **ClassPath**.

JavaCompiler

Specifies how the Java compiler will be invoked to compile the generated Java files, and to produce Java class files in the **OutputClassDir**.

OutputDirectory

Specifies the directory where the generated files for this component are placed. The **OutputDirectory** (once expanded) should not conflict with the **OutputDirectory** for other components.

BootstrapCommand

Specifies the command used to generate the Makefiles for this component, and all components upon which it depends.

GenerateCommand

Specifies the command used to generate all Java files for this component, and all components upon which it depends. Since the default **GenerateCommand** is Makefile-driven, it requires successful completion of the **BootstrapCommand** first.

CompileCommand

Specifies the command used to generate and compile all Java files (possibly producing a .jar file) for this component, and all components upon which it depends. Since the default **CompileCommand** is Makefile-driven, it requires successful completion of the **BootstrapCommand** first.

CleanAllCommand

During a rebuild, this field specifies the command to clean this component and all (`RTJava Project`) components upon which it depends.

When you invoke a "**Build > Rebuild**" of a component from the toolset, you effectively invoke the component's **CleanAllCommand**, then the **BootstrapCommand**, `and` then the **CompileCommand** (or **GenerateCommand**). The default **CleanAllCommand** is Makefile-driven. If the Component Makefile does not exist, the **CleanAllCommand** will fail and a warning is issued.

Note: When you **Build > Clean** a component, you do not invoke the **CleanAllCommand**. A "**Build > Clean**" will prompt you to delete the **OutputDirectory** of the component. A **"Build > Clean"** only cleans one component at a time, it does not clean depended-upon components.

MakeType

MakeType may be **DefaultMakeType**, **MS_nmake**, **Gnu_make**, or **Unix_make**. There are slight formatting differences between generated Makefiles that necessitate specifying the (third-party) **MakeType** you are using.

If you are generating on Windows NT, **DefaultMakeType** evaluates to **MS_nmake**. If you are generating on UNIX, **DefaultMakeType** evaluates to **Unix_make**.

Note: Support for **Clearcase_clearmake** and **Clearcase_omake** may be added in a future release.

ComponentMakeInsert

This text block is inserted in the Component Makefile (the Makefile in the component's **OutputDirectory)**.

Use this text block to add extra rules or to modify any of the generated Make macros. Some possible uses are described below.

Adding Make flags

If you want to add flags, such as UNIX make's **`-s`** (silent) flag, to the `RTgenerate` and `RTcompile` make tasks, add the following to your **ComponentMakeInsert**.

```
GMK_OPTS = -s
CMK_OPTS = -s
```

Replacing the CLEAN_CMD used in rebuild

The default **CleanAllCommand** (`$defaultMake RTcleanall`) calls the clean activity (the `RTmyclean` rule) for this component, and every RTJava Project component that it depends upon.

The default clean activity for each component is specified by the `RTmyclean` rule in the Component Makefile:

```
CLEAN_CMD=$(RM) -f -R RTbuild
RTmyclean :
   $(CLEAN_CMD)
```

While you cannot change the `RTmyclean` rule, you can change its order dependencies, and you can redefine **CLEAN_CMD**.

To disable **CLEAN_CMD**, add the following to your **ComponentMakeInsert**:

```
CLEAN_CMD=$(NOP)
```

Note: To disable the **CLEAN_CMD**, you do not want to set the **CLEAN_CMD** to empty because it is required by the `RTmyclean` rule.

To cause **CLEAN_CMD** to do something else instead, add the following to your **ComponentMakeInsert**:

```
CLEAN_CMD=do_this arg1 arg2 arg3
```

To cause **CLEAN_CMD** to do multiple steps, add the following to your **ComponentMakeInsert**:

```
RTmyclean : clean_first
clean_first :
   do_step1
   do_step2
CLEAN_CMD=do_step3
```

CodeGenMakeInsert

This text block is inserted in the Makefile responsible for generating this component.

Use this text block to add extra rules or to modify any of the generated Make macros. For example, to add pre-generation or post-generation steps.

Passing options to the code generator

To include the version number of the code generator in the **Build Log**, add the following to your component's **CodeGenMakeInsert** field:

```
RTGEN_USER_FLAGS = -version
```

To obtain (verbose) debugging information about the code generator's internal activities, add the following to your component's **CodeGenMakeInsert** field:

```
RTGEN_USER_FLAGS = -debug
```

Generating one Java file at a time

To force the code-generator to generate one Java file at a time, add the following to your component's **CodeGenMakeInsert** field:

```
RTGENERATE_TARGET = $(RTGENERATE_EACH_TARGET)
```

Adding a pre-generation activity

To add an extra rule "`myPreRule`" to run before all generation, add the following to your component's **CodeGenMakeInsert**:

```
$(RTGENERATE_TARGET) : myPreRule
myPreRule :
   echo "Starting generation"
```

Adding a post-generation activity

To add an extra rule "`myPostRule`" to run after all generation is successful, add the following to your component's **CodeGenMakeInsert** field:

```
RTGENERATE_TARGET = myrule
myPostRule : $(RTGENERATE_ONE_TARGET)
   echo "Finished generation"
```

CompileMakeInsert

This text block is inserted in the Makefile responsible for compiling this component.

Use this text block to add extra rules or to modify any of the generated Make macros, for example, to add pre-compilation or post-compilation steps.

Incremental compilation and individual Java file compilation

The Java Compiler for Sun JDK does not write class files incrementally (that is, only when a real change is necessary to the output files). To achieve incremental compilation, you must tell Make to compile every Java file individually.

Note: Individually compiling every file will slow down non-incremental builds (that is, initial builds and rebuilds).

To compile every Java file individually, add the following to the **CompileMakeInsert** of your components:

```
COMPILE_TARGET=$(COMPILE_EACH_TARGET)
```

Adding a pre-compilation activity

To add an extra rule "myPreRule" to run before all compilation, add the following to your component's **CompileMakeInsert** field:

```
$(COMPILE_TARGET) : myPreRule
myPreRule :
   echo "Starting compilation"
```

Adding a post-compilation activity

To add an extra rule "myPostRule" to run after all compilation is successful (and before running the JAR command, if applicable), add the following to your component's **CompileMakeInsert** field:

```
COMPILE_TARGET = myPostRule
myPostRule : $(COMPILE_ONE_TARGET)
   echo "Finished compilation"
```

To add an extra rule "myPostJarRule" to run after building the JAR, add the following to your component's **CompileMakeInsert** field:

```
BUILD_TARGET = myPostJarRule
myPostJarRule : $(JAR_TARGET)
   echo "Finished building JAR"
```

Variable Expansion for Fields

Table 1 Variable Expansion for Fields in RTJava Project

Variable	Definition	Sample Expansion
$compName	The unqualified name of this component.	`NewComponent1`

$defaultMake	Depends on the value of MakeType.	`nmake (for MS_nmake)`
		`make (for Unix_make)`
		`gmake (for Gnu_make)`

$dq or *${dq}*	Double quotes. Using a literal double quote will change the interpretation of a field to a text-block the next time the model file is loaded into the tool-set.	`"`
$modelDir	The directory in which the model file (or .rtmdl file) is stored.	`C:/MyModelDir/SmallModels` or `/home/me/models/BigModel1`
$modelName	The name of the model (the root name of the .rtmdl file).	`TestModel1`
$qualifiedName	The fully qualified name of this component.	`Component View::NewComponent1`

Table 1 Variable Expansion for Fields in RTJava Project (continued)

Variable	Definition	Sample Expansion
`$VARIABLE or${VARIABLE}`	Any Path Map variable or Environment variable.	Value of Path Map variable, or value of Environment variable, or empty if no such variable exists.
`$(VARIABLE)`	Any Make macro variable.	Unexpanded by code-generator. Expanded by Make.
`$$`	The escape sequence for dollar-sign, as used in Make macros (for example: `$$@`).	`$ (for example: $@)`

In cases where the variable's name token is followed by an alphanumeric character, it is recommended to use ${VARIABLE} notation instead of $VARIABLE notation.

Path Map Variables, Environment Variables and Make Macro Variables

You may want to use Path Map variables, Environment variables or Make macros for specifying:

- directories, such as `OutputClassDir`
- paths, such as ClassPath
- a file-path for `JavaCompiler`

This avoids hard-coding paths that may differ for another user using the same model.

For example, `'${J2ME_CLDC_HOME}/api/classes'` (versus `'/java/j2me_cldc/api/classes'`)

allows other users to specify their own path for `J2ME_CLDC_HOME`.

Path Map variables and Environment variables look and behave in similar ways, with a few exceptions. You can view and edit Path Map variables from the toolset, however, you cannot directly view or modify Environment variables from the toolset. Also, Path Map variables are exported to the build environment as Environment variables.

Make macro variables can also be used in some component fields to avoid hard-coding paths. In the above example, you could specify

```
'$(J2ME_CLDC_HOME)/api/classes'
```

using parentheses instead of braces. Since Make utilities will typically use the Environment variable (including Path Map variables) as a default when expanding a Make macro variable, the distinction between Make macros and Path Map variables are subtle. The main benefit is that Make macros are unexpanded in the Makefile. Consequently, a runtime environment (which expands Make macros in `ClassPath`) can be different from the generation or compilation environment.

RTJava External Project

An `RTJava External Project` component represents a set of UML classes, and/or a .jar file, or a set of Java class files that are not built by this component. This component type facilitates reuse (without rebuild) by `RTJava Project` components that depend upon it.

ClassPath

Specifies optional additions to the **ClassPath** variable that are necessary to compile or run another component that depends upon this component.

Multiple entries on the **ClassPath** may be separated with **';'** (on Windows NT) or **':'** (on UNIX).

You may want to use `Path Map` Variables or Make macros for directories to avoid hard-coding paths that may differ for another user using the same model.

For example,

```
'$(J2ME_CLDC)/api/classes' (versus 'R:/java/j2me_cldc/api/classes')
```

allows other users to specify their own path for `J2ME_CLDC`.

File Name Conventions

The use of backslashes and spaces are important considerations in the `RTJava Project` and `RTJava External Project` component fields.

Backslashes

In fields where paths are expected, it is recommended that you use forward slashes (even on Windows NT). Most compilation tools will accept both forward and backward slashes. For consistency, the code generator may convert backward slashes in specified paths to forward slashes.

Spaces in Directory Names

Spaces can legitimately occur in some fields (such as, directory names, file paths and UML qualified names). The code generator encloses these tokens in double quotes.

For example, it may seem natural to put double quotes around the directory name's declaration where the space occurred. Instead, the code generator adds double quotes wherever the directory name is used, such as:

- in a file path (as in **`OutputJarFilepath`**)
- in a list of directories (as in the **`-classpath`** argument to the Java compiler)
- a single-token directory name (as in the **`-d`** `argument` to the Java compiler)

Consequently, do not encapsulate the declared directory name with double quotes.

Java UML Services Library Class Reference 7

Contents

This chapter is organized as follows:

Java UML Services Library Class Reference Overview

The Java UML Services Library Class Reference is a reference to the classes that you will need to use within the detailed code of a capsule class to access the services provided by the Java UML Services Library.

The *Java UML Services Library Framework* on page 76 shows the classes in a class diagram. The remainder of the Class Library Reference consists of an alphabetical listing of the classes.

In the alphabetical listing section, each class description includes a member summary by category, followed by alphabetical listings of operations and attributes. Nested classes are then listed in the same fashion. This reference does not describe private or restricted elements of the Services Library. Some features and classes in the Services Library are internal to the library itself (even though they may be accessible according to the Java language rules) and thus are not supported as interfaces available for use in applications.

Application

An instance of this class is the anchor for an application; it is the main controller object. Useful extensions would include creating other controller and thread objects.

Operations

getArgCount	Gets the number of arguments present on the command line.
getArgString	Gets the specified argument from the command line. The first argument is at index zero.
logicalControllerDeregister	Destroy the mapping from a name or to a given controller object.
logicalControllerFind	Find the controller registered with the given name or return null if exists.
logicalControllerRegister	Create a mapping from the given name to the given controller object. Return true if successful. If the name is already in use false is returned.
main	This is the default entry-point to run an application. The first argument names the top-level capsule class. Remaining arguments will be available via getArgString().
run	The code executed by the associated thread. This is the implementation of the java.lang.Runnable interface.

Attributes

There are no attributes accessible to user code.

Nested Classes

There are no nested classes.

Application.getArgCount

public int Application.getArgCount()

Return value

Returns the number of arguments that follow the top capsule class name on the command line.

Exceptions

None.

Parameters

None.

Remarks

Examples

The operation getArgCount would return the value 3 if the following command were used to start an application.

```
java  com.rational.rosert.Application example.TopCapsule one two three
```

Application.getArgString

public java.lang.String Application.getArgString(int index)

Return value

Returns the arguments at position “index” that follows the top capsule class name on the command-line.

Exceptions

None.

Parameters

index	The position on the command-line of the argument.

Remarks

Indexing starts at 0, and ends at Application.getArgCount() -1.

Examples

The operation getArgString(0) would return the string "one" if the following command were used to start an application.

```
java com.rational.rosert.Application example.TopCapsule one two three
```

Application.logicalControllerDeregister

void Application.logicalControllerDeregister(Controller controller)

void Application.logicalControllerDeregister(java.lang.String name)

Return value

None.

Exceptions

None.

Parameters

controller	The controller to be deregistered.
name	The logical name to be removed.

Remarks

The first form removes the mapping from any name to the given controller. The latter destroys the mapping from the given name to any controller.

Examples

The predefined mapping from "main" to the application object may be removed as shown below.

```
rtGetController().getApplication().logicalControllerDeregister
( "main" );
```

Application.logicalControllerFind

Controller Application.logicalControllerFind(java.lang.String name)

Return value

The controller object associated with the given logical name or null if no controller was registered with that logical name.

Exceptions

None.

Parameters

name	The logical controller name.

Remarks

This operation allows designs to be largely independent of the actual number of controller objects (and their associated Java threads).

Examples

The following two expressions normally yield the same object.

```
rtGetController().getApplication();
rtGetController().getApplication().logicalControllerFind( "main" );
```

Application.logicalControllerRegister

boolean Application.logicalControllerRegister(java.lang.String name, Controller controller)

Return value

Returns true if no other controller object is registered with the given name.

Exceptions

None.

Parameters

name	The logical controller name.
controller	The controller object.

Remarks

Capsules may be incarnated in the context of distinct controllers. Sometimes it is beneficial to produce an abstraction of controllers independent of the actual number of controllers. In this abstraction, we refer to 'logical' controllers. This operation permits a design to capture that abstraction in the application object. Controllers are registered with one or more logical names allowing model elements to use the logical name through use of the logicalControllerFind operation.

Examples

```
com.rational.rosert.Application application =
rtGetController().getApplication();

com.rational.rosert.Controller secondary = new
com.rational.rosert.Controller( application, "two" );

application.logicalControllerRegister( "two", secondary );
```

Application.main

static void Application.main(java.lang.String[] args)

Return value

None.

Exceptions

None.

Parameters

args	The command-line arguments.

Remarks

This function can be used as the entry-point to an application. The first element of the array of strings is interpreted as the name of a class that is to be used to construct the top capsule.

Examples

```
Application.main( new java.lang.String[] { "TopCapsule", "foo",
"bar" } );
```

Application.run

void Application.run()

void Application.run(java.lang.String[] args)

Return value

None.

Exceptions

None.

Parameters

args	The command-line arguments.

Remarks

This operation is normally invoked by main but an extension of the Application class might supply its own argument list for operation in environments where there is no concept of 'command-line'.

Examples

```
run( new java.lang.String[] { "TopCapsule" } );
```

Capsule

Every capsule class when generated as Java code is a subclass of Capsule. This common base class for all capsules defines attributes and operations that allow the Services Library to communicate with the running capsule instances.

Since all detail level code added to a capsule class is generated as part of a capsule class, the detail level code has direct access to some useful attributes, operations and nested classes defined in Capsule. You should only use the features described below.

Note: The attributes and operations on Capsule should be considered private. One capsule instance should not manipulate another capsule's attributes nor invoke its operations directly.

Operations

rtDeferMessage	Used to defer processing of the current message.
rtDestroy	The Services Library invokes this method as a capsule instance is destroyed. At the time of the call, all capsule roles have been destroyed or unplugged, all ports are unbound, and any timing requests have been cancelled.
rtForwardMessage	Used to forward the current message through the given port.
rtGetController	Gets the controller for the physical thread on which a capsule instance is executing.
rtGetMsgData	Gets the data associated with the message currently being handled.
rtGetMsgPort	Gets the port on which the current message arrived.
rtGetMsgPortIndex	Gets the index of the port on which the current message arrived.
rtGetMsgPriority	Gets the priority of the current message.
rtGetMsgSignal	Gets the code for the signal of the current message.
rtWasInvoked	Queries whether the current message was the result of an invoke or invokeAt operation.

Attributes

There are no attributes accessible to user code.

Nested Classes

Message	Instances represent replies to synchronous communication operations.

Capsule.rtDeferMessage

void Capsule.rtDeferMessage()

Return value

None.

Exceptions

AlreadyDeferredException	A message may only be deferred once.
DeferredInitializationException	An initialization message may not be deferred.
DeferredInvokeException	An invoked message may not be deferred.

Parameters

None.

Remarks

Only asynchronous messages may be deferred.

Examples

```
rtDeferMessage();
```

Capsule.rtDestroy

void Capsule.rtDestroy()

Return value

None.

Exceptions

None.

Parameters

None.

Remarks

Even though java has automatic garbage collection, it is often more efficient to release resources as early as possible. This is the mechanism whereby capsules can ensure that resources are not held longer than necessary.

Examples

A capsule that dynamically creates an auxillary controller and its associated thread must be prepared to clean up when it is destroyed. If we assume the capsule has the following attributes.

```
com.rational.rosert.Controller auxController;
java.lang.Thread               auxThread;
```

Then the body of its rtDestroy operation might look like this.

```
auxController.abort( rtGetController() );
for(;;)
{
   try
   {
      auxThread.join();
      break;
   }
   catch( java.lang.InterruptedException ex )
   {
      // keep trying to join with the other thread
   }
}
```

Capsule.rtForwardMessage

void Capsule.rtForwardMessage(ProtocolRole port)

Return value

None.

Exceptions

IllegalForwardException	The signal type of the message is not an outgoing signal of the specified port.

Parameters

port	The port via which the message is to be forwarded.

Remarks

A recurring design pattern seems to involve forwarding messages. This operation permits it to be done safely.

Examples

```
//Forward the current message to the port delegate at index “index”
rtForwardMessage( delegate ).sendAt( index );
```

Capsule.rtGetController

Controller Capsule.rtGetController()

Return value

Returns the controller object for the thread on which this capsule instance is running.

Exceptions

None.

Parameters

None.

Remarks

The controller object is required to gain access to the application object and to terminate other controllers.

Examples

```
Application application = rtGetController().getApplication();
```

Capsule.rtGetMsgData

java.lang.Object Capsule.rtGetMsgData()

Return value

The data given to the signal function used to produce the message currently being handled.

Exceptions

None.

Parameters

None.

Remarks

The signal is likely associated with a more specific data type: the value will need to be cast accordingly.

Examples

```
java.lang.Integer value = (java.lang.Integer)rtGetMsgData();
```

Capsule.rtGetMsgPort

ProtocolRole Capsule.rtGetMsgPort()

Return value

Returns the port object on which the current message was received, or null in the case of the initialization message.

Exceptions

None.

Parameters

None.

Remarks

Transitions are often triggered by events on a single port. In such cases, it is more efficient to simply use that port name in the action code and avoid the cost of a cast of the result of this operation.

Examples

```
((Request.Base)rtGetMsgPort()).granted().send();
```

Capsule.rtGetMsgPortIndex

int Capsule.rtGetMsgPortIndex()

Return value

Yields the zero-based index of the port on which the current message arrived.

Exceptions

None.

Parameters

None.

Remarks

The port index may be a convenient distinguishing feature when handling multiple clients.

Examples

```
port.hello().sendAt( rtGetMsgPortIndex() );
```

Capsule.rtGetMsgPriority

Priority Capsule.rtGetMsgPriority()

Return value

Yields the priority of the current message.

Exceptions

None.

Parameters

None.

Remarks

It is recommend that only the General priority be used.

Examples

The priority of a message might dictate whether it gets immediate attention.

```
if( rtGetMsgPriority() == Priority.Panic )
{
   // handle this now
}
else
{
   rtDeferMessage(); // handle it later
}
```

Capsule.rtGetMsgSignal

int Capsule.rtGetMsgSignal()

Return value

Signals within the same protocol role are assigned small distinct numeric codes. This operation returns the signal code of the current message.

Exceptions

None.

Parameters

None.

Remarks

It is recommended that when actions depend on the triggering signal those actions be coded into separate transitions. Sometimes this is not convenient. In such cases, this operation can be used to discriminate the different signals.

Examples

```
// common preface
if( rtGetMsgSignal() == Service.Base.rti_request )
{
   // handle small variation for 'request' signal
}
```

Capsule.rtWasInvoked

boolean Capsule.rtWasInvoked()

Return value

Answers whether the current message was the result of an invoke or invokeAt operation.

Exceptions

None.

Parameters

None.

Remarks

In the forwarding pattern, we must know whether we were invoked so that we can handle a reply properly.

Examples

```
if( rtWasInvoked() )
{
   Capsule.Message reply = rtForwardMessage( server ).invokeAt(
   serverIndex );
   if( reply != null )
      reply.forward( rtGetMsgPort() ).reply();
}
else
{
   rtForwardMessage( server ).sendAt( serverIndex );
}
```

Capsule.Message

Instances of this class are only accessible through use of the invoke and invokeAt operations. They return an array of messages or a single message, respectively, corresponding to each reply.

Operations

forward	To forward the reply.
getData	Gets the data in a reply.
getSignal	Gets the signal code from a reply.

Attributes

There are no attributes accessible to user code.

Nested Classes

There are no nested classes.

Capsule.Message.forward

ProtocolRole.OutSignal Capsule.Message.forward(ProtocolRole port)

Return value

The object returned is an OutSignal.

Exceptions

None.

Parameters

None.

Remarks

A recurring design pattern seems to involve forwarding messages. This operation permits replies to be done safely.

Examples

```
Capsule.Message reply = server.ping().invokeAt( serverIndex );
if( reply == null )
   client.noResponse().sendAt( clientIndex );
else
   reply.forward( client ).sendAt( clientIndex );
```

Capsule.Message.getData

java.lang.Object Capsule.Message.getData()

Return value

This operation returns the data associated with a reply message.

Exceptions

None.

Parameters

None.

Remarks

This operation must be used to access any data present in the reply to an invoke or invokeAt operation.

Examples

```
Capsule.Message reply = server.ping().invokeAt( serverIndex );
if( reply != null )
{
   log.show( "reply data is " );
   log.show( reply.getData() );
   log.cr();
}
```

Capsule.Message.getSignal

int Capsule.Message.getSignal()

Return value

This operation returns the code used to distinguish among signals of the related protocol role.

Exceptions

None.

Parameters

None.

Remarks

It is recommended that separate transitions be used to handle distinct signals in different ways. Unfortunately, this is not possible when handling the reply to synchronous messages.

Examples

The following illustrates how to identify a particular signal. Be careful to ensure that the protcol role class matches the port through which the invoke occurred.

```
Capsule.Message msg = port.request().invokeAt( serviceIndex );
if( msg.getSignal() == Service.Conjugate.rti_Granted )
   // handle grant of request
```

CapsuleRole

This class is used to allow capsules to contain other classes as represented by a structure diagram. Instances of this class are generated by the code generator.

Operations

There are no operations intended for use in user code.

Attributes

There are no attributes accessible to user code.

Nested Classes

MissingClassError	An exception thrown by the constructor if the specified capsule class cannot be found.

Controller

Controllers are used to represent groups of capsules running on the same thread. When controllers are created, they are usually given to a thread to run (controllers implement the run interface). Controllers may be then used as an argument for the Frame operation of incarnate.

Operations

abort	Used to cause the controller to terminate.
getApplication	Gets the application (main controller) object.
run	The implementation of the java.lang.Runnable interface.

Attributes

There are no attributes accessible to user code.

Nested Classes

There are no nested classes accessible to user code.

Controller.abort

void Controller.abort()

void Controller.abort(Controller requestor)

Return value

None.

Exceptions

None.

Parameters

requestor	The controller object associated with the current thread.

Remarks

Calling this operation will cause the controller to terminate. The second form must be used from threads other than the one associated with the controller that is to be terminated.

Examples

```
rtGetController().abort();
```

```
secondController.abort( rtGetController() );
```

Controller.getApplication

Application Controller.getApplication()

Return value

This operation returns the application object (that is, the main controller).

Exceptions

None.

Parameters

None.

Remarks

This operation must be used to access the application object. It is useful for accessing the command-line arguments, for example.

Examples

```
Application application = rtController().getApplication();
```

Controller.run

void Controller.run()

Return value

None.

Exceptions

None.

Parameters

None.

Remarks

Controller instances may be created and destroyed dynamically. Each new controller must have an associated thread context which will execute this operation.

Examples

```
Application application = rtController().getApplication();
Controller secondController = new Controller( application, "second" );
java.lang.Thread secondThread = new java.lang.Thread
( secondController );
secondThread.start();
```

Frame.Base

Operations

cardinalityOf	Gets the cardinality of the given capsule role.
destroy	Destroy one or all capsule instances of the specified capsule role.
incarnate	Create a capsule within the specified optional capsule role.
incarnationAt	Retrieves a particular capsule instance of a capsule role.
plugIn	Import a capsule instance into a plug-in capsule role.
unplug	Remove a capsule instance from a plug-in capsule role.

Attributes

There are no attributes accessible to user code.

Nested Classes

There are no nested classes intended for use in user code.

Frame.Base.cardinalityOf

int Frame.Base.cardinalityOf(CapsuleRole role)

Return value

This operation returns the cardinality of the given capsule role.

Exceptions

None.

Parameters

role	The name of the capsule role contained in the structure of the capsule instance making the call.

Remarks

This operation returns the replication size of the specified capsule role whether there is a capsule instance currently present at any specific location or not.

Examples

To unplug all capsule instances from a plug-in role

```
for( int index = frame.cardinalityOf( plugInRole ); --index >= 0; )
   frame.unplug( plugInRole, index );
```

Frame.Base.destroy

void Frame.Base.destroy(CapsuleRole role)

void Frame.Base.destroy(CapsuleRole role, int index)

void Frame.Base.destroy(Capsule instance)

Return value

None.

Exceptions

java.lang.IllegalArgumentException	The role is not an optional capsule role in the immediate decomposition of the capsule which owns the frame port.
java.lang.IndexOutOfBoundsException	The index is negative or not less than the cardinality of the capsule role.

Parameters

role	The name of the capsule role contained in the structure of the capsule instance making the call.
index	The zero-based index within the capsule role of the capsule instance.
instance	The capsule instance.

Remarks

The first form destroys all instances within the specified capsule role. The other two destroy at most one capsule instance.

The role must be an optional capsule role in the immediate decomposition of the capsule instance which owns the frame port. In the third form, the capsule role is implicitly the one in which the capsule instance was incarnated.

Examples

```
frame.destroy( terminal );
```

Frame.Base.incarnate

void Frame.Base.incarnate(CapsuleRole role)

void Frame.Base.incarnate(CapsuleRole role, java.lang.Class capsuleClass)

void Frame.Base.incarnate(CapsuleRole role, java.lang.Object data, Controller controller)

void Frame.Base.incarnate(CapsuleRole role, java.lang.Class capsuleClass, java.lang.Object data, Controller controller)

void Frame.Base.incarnate(CapsuleRole role, java.lang.Object data, Controller controller, int index)

void Frame.Base.incarnate(CapsuleRole role, java.lang.Class capsuleClass, java.lang.Object data, Controller controller, int index)

Return value

None.

Exceptions

BadIndexException	The index is out of bounds or specifies an occupied location in the role.
FullException	There are no unoccupied locations in the role.
IncompatibleClassException	The capsuleClass was specified and is incompatible with the class used to define the role.
java.lang.IllegalAccessException	The class or its default constructor are inaccessible.
java.lang.IllegalArgumentException	The role is not owned by the same capsule instance as the frame port.
java.lang.InstantiationException	The class is abstract or an interface.

Parameters

Name	Meaning	Default
role	The name of an optional capsule role contained in the structure of the capsule instance making the incarnate call	N/A
capsuleClass	The class of the new capsule instance.	The class used to define the role.
data	Data to be available during the initial transition of the new capsule instance.	null
controller	The context in which the new instance should execute. A null value may be used to have the new capsule instance execute within the same context as the capsule making the incarnate call.	null
index	The zero-based index within the capsule role of the new capsule instance.	N/A

Remarks

To use the frame operations, create a protected end port using the Frame protocol.

Examples

```
try
{
   frame.incarnate( terminal );
}
catch( com.rational.rosert.FullException ex )
{
}
. . .
```

Frame.Base.incarnationAt

Capsule Frame.Base.incarnationAt(CapsuleRole role, int index)

Return value

The capsule instance at the specified capsule role index, or null if no instance exists at that location.

Exceptions

java.lang.IllegalArgumentException	The role is not owned by the capsule instance that owns the frame port.
java.lang.IndexOutOfBoundsException	The index must be non-negative and less than the cardinality of the capsule role.

Parameters

role	The capsule role in the immediate decomposition of the capsule instance that owns the frame port.
index	The zero-based index into the capsule role.

Remarks

This is the only mechanism available for obtaining a reference to a capsule instance which is required for use with plug-in roles, for example.

Examples

The code below sends a capsule instance to be plugged into a role in another context.

```
Capsule capsule = frame.incarnationAt( endpoints, index );
If( capsule != null )
   peer.endpoint( capsule ).send();
```

Frame.Base.plugIn

void Frame.Base.plugIn(CapsuleRole role, Capsule instance)

void Frame.Base.plugIn(CapsuleRole role, Capsule instance, int index)

Return value

None.

Exceptions

BadIndexException	The index is out of bounds or specifies an occupied location in the role.
FullException	All locations within the capsule role are occupied.
IncompatibleClassException	The class of the capsule instances is incompatible with the role.
java.lang.IllegalArgumentException	The role is not owned by the same capsule instance as the frame port.

Parameters

role	The capsule role in the immediate decomposition of the capsule instance that owns the frame port.
instance	The capsule instances which is to be plugged into the capsule role.
index	The zero-based index within the capsule role.

Remarks

A given capsule instance can be plugged into a capsule role at most once.

Examples

```
frame.plugIn( server, serverCapsule );
```

Frame.Base.unplug

void Frame.Base.unplug(CapsuleRole role, Capsule instance)

void Frame.Base.unplug(CapsuleRole role, int index)

Return value

None.

Exceptions

BadIndexException	The index is out of bounds or specifies an unoccupied location in the role.
java.lang.IllegalArgumentException	The role is not owned by the same capsule instance as the frame port.

Parameters

role	The plug-in capsule role from which the capsule instance is to be removed.
instance	The capsule instance to be removed. It must currently be plugged into the capsule role.
index	The zero-based index of the capsule index within the role.

Remarks

This operation is the inverse of the plugIn operation.

Examples

```
frame.unplug( caller, 0 );
```

Log.Base

The System Log is accessed via ports using the Log protocol, which are instances of the class Log.Base. The log port only takes incoming messages and does not pass any information in the reverse direction. The operations available for accessing the system log are listed below.

Operations

close	Disable further output.
commit	Ensure any buffering is flushed.
cr	Begin a new line.
crtab	Begin a new line with a number of tab characters.
log	Write data and begin a new line.
open	Enable output from writing operations.
show	Write data.
space	Write a space.
tab	Write a tab character.

Attributes

There are no attributes accessible to user code.

Nested Classes

There are no nested classes intended for use in user code.

Log.Base.close

void Log.Base.close()

Return value

None.

Exceptions

None.

Parameters

None.

Remarks

Temporarily suspends output via this port.

Examples

```
log.close();
```

Log.Base.commit

void Log.Base.commit()

Return value

None.

Exceptions

None.

Parameters

None.

Remarks

Ensure all output is written.

Examples

```
log.commit();
```

Log.Base.cr

void Log.Base.cr()

Return value

None.

Exceptions

None.

Parameters

None.

Remarks

Begin a new line.

Examples

```
log.cr();
```

Log.Base.crtab

void Log.Base.crtab()

void Log.Base.crtab(int tabCount)

Return value

None.

Exceptions

None.

Parameters

tabCount	The number of tab characters to write. The default is one.

Remarks

Begin a new line with the specified number of tab characters.

Examples

```
log.crtab( 2 );
```

Log.Base.log

void Log.Base.log(boolean value)

void Log.Base.log(byte value)

void Log.Base.log(char value)

void Log.Base.log(int value)

void Log.Base.log(long value)

void Log.Base.log(short value)

void Log.Base.log(java.lang.Object object)

Return value

None.

Exceptions

None.

Parameters

value	A value of a primitive data type.
object	A java object reference.

Remarks

Convert the argument to a human-readable string which is output. A new line is begun

Examples

```
log.log( "The time is " );
log.log( java.lang.System.currentTimeMillis() );
```

Log.Base.open

void Log.Base.open()

Return value

None.

Exceptions

None.

Parameters

None.

Remarks

Resume writing.

Examples

```
log.open();
```

Log.Base.show

void Log.Base.show(boolean value)

void Log.Base. show(byte value)

void Log.Base. show(char value)

void Log.Base. show(int value)

void Log.Base. show(long value)

void Log.Base. show(short value)

void Log.Base. show(java.lang.Object object)

Return value

None.

Exceptions

None.

Parameters

value	A value of a primitive data type.
object	A java object reference.

Remarks

Convert the argument to a human-readable string which is output. Any following output will appear on the same line.

Examples

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

Log.Base.space

void Log.Base.space()

Return value

None.

Exceptions

None.

Parameters

None.

Remarks

Writes one space.

Examples

```
log.space();
```

Log.Base.tab

void Log.Base.tab()

Return value

None.

Exceptions

None.

Parameters

None.

Remarks

Writes a tab character.

Examples

```
log.tab();
```

Priority

The Priority class is used to specify priority values for the messaging system. This class consists of five class scoped Final values.

Operations

There are no operations defined by this class.

Attributes

Panic	The highest priority available to user code.
High	
General	The default priority.
Low	
Background	The lowest priority available to user code.

Nested Classes

There are no nested classes.

ProtocolRole

For each protocol class in your model, two subclasses of the ProtocolRole class are generated (one for each direction of the protocol or protocol roles). Each port defined on a capsule becomes an attribute of the generated Java capsule class. The port attribute has the same name as the port and its type is either the base or conjugate role.

Operations

bindingNotification	Use this operation to request notification of the creation and destruction of bindings to instances of this port.
bindingNotificationRequested	Use this operation to query whether binding notification has been requested for this port.
cardinality	Obtains the cardinality of this port.
deregister	Deregisters an unwired end port.
deregisterSAP	Deregisters an unpublished unwired end port.
deregisterSPP	Deregisters a published unwired end port.
getRegisteredName	Get the name that an unwired end port has registered with the layer service.
isBoundAt	Query whether the given replication index is connected.
isRegistered	Query whether the unwired end port is registered with the layer service.
purge	Empties the queue of deferred messages that are associated with this port.
purgeAt	Empties the queue of deferred messages that are associated with this port and the specified replication index.
recall	Recall a deferred message associated with this port.
recallAll	Recall all deferred messages associated with this port.
recallAllAt	Recall all deferred messages associated with this port and the specified replication index.
recallAt	Recall a deferred message associated with this port and the specified replication index.
registerSAP	Registers an unpublished unwired end port with the service layer.

registerSPP	Registers a published unwired end port with the service layer.
resize	Adjust the cardinality of this port.

Attributes

There are no attributes accessible to user code.

Nested Classes

InSignal	Returned by 'in' signals of protocol roles.
OutSignal	Returned by 'out' signals of protocol roles.
SymmetricSignal	Returned by 'symmetric' signals of protocol roles.

ProtocolRole.bindingNotification

void ProtocolRole.bindingNotification(boolean enable)

Return value

None

Exceptions

None.

Parameters

enable	Specifies whether binding notification messages are desired.

Remarks

Use this operation to request notification of the creation and destruction of bindings to this port. The signals sent to the port by the Services Library are **rtBound** and **rtUnbound**.

This operation will have no effect on binding notification messages which have already been queued nor will it cause any messages to be sent for instances which are already bound.

Examples

```
port.bindingNotification( true );
```

ProtocolRole.bindingNotificationRequested

boolean ProtocolRole.bindingNotificationRequested()

Return value

Answers whether binding notifications messages will be sent to this port.

Exceptions

None.

Parameters

None.

Remarks

The initial state is defined by the 'Notification' property of the port. It can be modified at run-time.

Examples

The action associated with rtBound and rtUnbound signals can be made to ignore those events after disabling the feature as shown below.

```
if( rtport.bindingNotificationRequested() )
{
   // handle the expected notification message
}
```

ProtocolRole.cardinality

int ProtocolRole.cardinality()

Return value

Queries the cardinality of the port.

Exceptions

None.

Parameters

None.

Remarks

The initial cardinality is defined in the model and may be adjusted at run-time with the resize operation.

Examples

A server might prepare to handle more clients by increasing the cardinality of an unwired published port as shown below.

```
port.resize( port.cardinality() + 10 );
```

ProtocolRole.deregister

void ProtocolRole.deregister()

Return value

None.

Exceptions

UnsupportedOperationException	This operation is only supported for unwired ports.

Parameters

None.

Remarks

This operation calls deregisterSAP or deregisterSPP as appropriate.

Examples

A capsule need not remember whether an unwire port registration published or unpublished.

```
port.deregister();
```

ProtocolRole.deregisterSAP

void ProtocolRole.deregisterSAP()

Return value

None.

Exceptions

UnsupportedOperationException	This operation is only supported for unwired ports.

Parameters

None.

Remarks

This operation removes any registration as an unpublished unwired port. If the port was not registered via registerSAP, this operation has no effect.

Examples

```
port.deregisterSAP();
```

ProtocolRole.deregisterSPP

void ProtocolRole.deregisterSPP()

Return value

None.

Exceptions

UnsupportedOperationException	This operation is only supported for unwired ports.

Parameters

None.

Remarks

This operation removes any registration as a published unwired port. If the port was not registered via registerSPP, this operation has no effect.

Examples

```
port.deregisterSPP();
```

ProtocolRole.getRegisteredName

java.lang.String ProtocolRole.getRegisteredName()

Return value

Returns the name that an unwired port has registered with the layer service or null if the port is unregistered.

Exceptions

UnsupportedOperationException	This operation is only supported for unwired ports.

Parameters

None.

Remarks

The value returned is equal to the name supplied to registerSAP or registerSPP but will not necessarily be the same object.

Examples

```
if( ! port.getRegisteredName().equals( desirecService ) )
{
   port.registerSAP( desiredService );
}
```

ProtocolRole.isBoundAt

boolean ProtocolRole.isBoundAt(int index)

Return value

Returns true if the specified replication of this port is connected to another port and false otherwise.

Exceptions

None.

Parameters

index	Specifies the replication index of the port.

Remarks

Bindings can be asynchronous created and destroyed by actions of capsules on other threads. Care must be taken when using this operation not to introduce race conditions into the application.

Examples

The code below is fragile: the port may become unbound between the call to isBoundAt and the call to sendAt. It is preferable to catch the PortUnBoundException of the sendAt operation.

```
if( port.isBoundAt( index ) )
   port.serviceRequest( data ).sendAt( index );
```

ProtocolRole.isRegistered

boolean ProtocolRole.isRegistered()

Return value

Returns true if the port has been registered and false otherwise.

Exceptions

UnsupportedOperationException	This operation is only supported for unwired ports.

Parameters

None.

Remarks

This operation merely queries whether the registerSAP or registerSPP operations have been used without being followed by the corresponding deregisterSAP or deregisterSPP. It conveys no information about whether other port roles are registered with the same or matching name.

Examples

```
if( port.isRegistered() )
   port.deregister();
```

ProtocolRole.purge

int ProtocolRole.purge()

Return value

Returns the number of messages removed from the defer queue.

Exceptions

None.

Parameters

None.

Remarks

This operation selects messages for removal without regard for the signal or the replication index of the port with which they are associated.

Examples

```
port.purge();
```

ProtocolRole.purgeAt

int ProtocolRole.purgeAt(int index)

Return value

Returns the number of messages removed from the defer queue.

Exceptions

None.

Parameters

index	Specifies the replication index of the related port.

Remarks

This operation selects messages for removal without regard for the signal but only if they are associated with the specified replication index of the port.

Examples

```
rtport.purgeAt( rtGetMsgPortIndex() );
```

ProtocolRole.recall

boolean ProtocolRole.recall()

boolean ProtocolRole.recall(boolean front)

Return value

Returns true if a matching message was recalled, false otherwise.

Exceptions

None.

Parameters

front	Specifies whether recalled messages should be queued ahead of (true) or behind (false) other queued messages.

Remarks

This operation recalls the first deferred message without regard to its signal type or port replication index.

The first form is equivalent to calling the second with the argument 'false'.

Examples

```
port.recall( true );
```

ProtocolRole.recallAll

int ProtocolRole.recallAll()

int ProtocolRole.recallAll(boolean front)

Return value

Returns the number of messages taken from the defer queue and requeued for delivery.

Exceptions

None.

Parameters

front	Specifies whether recalled messages should be queued ahead of (true) or behind (false) other queued messages.

Remarks

This operation recalls all deferred messages without regard to their signal type or port replication index.

The first form is equivalent to calling the second with the argument 'false'.

Examples

```
port.recallAll( true );
```

ProtocolRole.recallAllAt

int ProtocolRole.recallAllAt(int index)

int ProtocolRole.recallAllAt(int index, boolean front)

Return value

Returns the number of messages taken from the defer queue and requeued for delivery.

Exceptions

None.

Parameters

index	Specifies the replication index of the related port.
front	Specifies whether recalled messages should be queued ahead of (true) or behind (false) other queued messages.

Remarks

This operation recalls all deferred messages of any signal type at the specified port replication index.

The first form is equivalent to calling the latter with the 'false' as the second argument.

Examples

```
port.serviceRequest().recallAllAt( 0, true );
```

ProtocolRole.recallAt

boolean ProtocolRole.recallAt(int index)

boolean ProtocolRole.recallAt(int index, boolean front)

Return value

Returns true if a matching message was recalled, false otherwise.

Exceptions

None.

Parameters

index	Specifies the replication index of the related port.
front	Specifies whether recalled messages should be queued ahead of (true) or behind (false) other queued messages.

Remarks

This operation recalls the first deferred message of any signal type at the specified port replication index.

The first form is equivalent to calling the latter with the 'false' as the second argument.

Examples

```
port.serviceRequest().recallAt( 0, true );
```

ProtocolRole.registerSAP

void ProtocolRole.registerSAP(java.lang.String name)

Return value

None.

Exceptions

UnsupportedOperationException	This operation is only supported for unwired ports.

Parameters

name	Specifies the service name.

Remarks

This operation is used to request bindings to one or more other ports which have announced or will announce their availability with the registerSPP operation. The number of bindings requested is equal to the cardinality of this port.

If the port was already registered for a different service name, this operation implicitly calls deregister first.

Examples

```
service.registerSAP( "service" );
```

ProtocolRole.registerSPP

void ProtocolRole.registerSPP

Return value

None.

Exceptions

UnsupportedOperationException	This operation is only supported for unwired ports.

Parameters

name	Specifies the service name.

Remarks

This operation is used to provide a place to terminate bindings one or more other ports which have announced or will announce their availability with the registerSAP operation. The number of bindings provided is equal to the cardinality of this port.

If the port was already registered for a different service name, this operation implicitly calls deregister first.

Examples

```
service.registerSPP( "service" );
```

ProtocolRole.resize

void ProtocolRole.resize(int newCardinality)

Return value

None.

Exceptions

UnsupportedOperationException	This operation is only supported for unwired ports.
java.lang.IllegalArgumentException	The specified cardinality must be non-negative.

Parameters

newCardinality	Specifies the desired cardinality.

Remarks

This operation is used to adjust the cardinality of an unwired port. The new cardinality may be larger or smaller than the old cardinality.

If the cardinality is reduced while the port is registered, bindings at replication indices equal to or larger than the new cardinality are severed and no rtUnbound signal will be received on this port even if notification is enabled. If the cardinality is increased while the port is registered, new bindings become possible—perhaps immediately.

Examples

A server might prepare to handle more clients by increasing the cardinality of an unwired published port as shown below.

```
port.resize( port.cardinality() + 10 );
```

ProtocolRole.InSignal

Instances of this class are created by operations generated for incoming signals of protocol roles.

Operations

purge	Delete all of these deferred signals associated with the related port.
purgeAt	Delete all of these deferred signals associated with the related port and the specified replication index.
recall	Recall one of these deferred signals associated with the related port.
recallAll	Recall all of these deferred signals associated with the related port.
recallAllAt	Recall all of these deferred signals associated with the related port and the specified replication index.
recallAt	Recall one of these deferred signals associated with the related port and the specified replication index.

ProtocolRole.InSignal.purge

int ProtocolRole.InSignal.purge()

Return value

Returns the number of messages removed from the defer queue.

Exceptions

None.

Parameters

None.

Remarks

This operation selects messages for removal without regard for the replication index of the port with which they are associated.

Examples

```
port.rtBound().purge();
```

ProtocolRole.InSignal.purgeAt

int ProtocolRole.InSignal.purgeAt(int index)

Return value

Returns the number of messages removed from the defer queue.

Exceptions

None.

Parameters

index	Specifies the replication index of the related port.

Remarks

This operation selects for removal only those messages associated with the specified replication index of the port.

Examples

```
port.rtBound().purgeAt( 0 );
```

ProtocolRole.InSignal.recall

boolean ProtocolRole.InSignal.recall()

boolean ProtocolRole.InSignal.recall(boolean front)

Return value

Returns true if a matching message was recalled, false otherwise.

Exceptions

None.

Parameters

front	Specifies whether recalled messages should be queued ahead of (true) or behind (false) other queued messages.

Remarks

This operation recalls the first deferred message of this signal type at any port replication index. To recall the first message of any signal type, use the ProtocolRole.recall operation.

The first form is equivalent to calling the second with the argument 'false'.

Examples

```
port.serviceRequest().recall( true );
```

ProtocolRole.InSignal.recallAll

int ProtocolRole.InSignal.recallAll()

int ProtocolRole.InSignal.recallAll(boolean front)

Return value

Returns the number of messages taken from the defer queue and requeued for delivery.

Exceptions

None.

Parameters

front	Specifies whether recalled messages should be queued ahead of (true) or behind (false) other queued messages.

Remarks

This operation recalls all deferred messages of this signal type at any port replication index. To recall all messages of any signal type, use the ProtocolRole.recallAll operation.

The first form is equivalent to calling the second with the argument 'false'.

Examples

```
port.serviceRequest().recallAll();
```

ProtocolRole.InSignal.recallAllAt

int ProtocolRole.InSignal.recallAllAt(int index)

int ProtocolRole.InSignal.recallAllAt(int index, boolean front)

Return value

Returns the number of messages taken from the defer queue and requeued for delivery.

Exceptions

None.

Parameters

index	Specifies the replication index of the related port.
front	Specifies whether recalled messages should be queued ahead of (true) or behind (false) other queued messages.

Remarks

This operation recalls all deferred messages of this signal type at the specified port replication index. To recall all messages of any signal type, use the ProtocolRole.recallAllAt operation.

The first form is equivalent to calling the latter with the 'false' as the second argument.

Examples

```
port.serviceRequest().recallAllAt( 0, true );
```

ProtocolRole.InSignal.recallAt

boolean ProtocolRole.InSignal.recallAt(int index)

boolean ProtocolRole.InSignal.recallAt(int index, boolean front)

Return value

Returns true if a matching message was recalled, false otherwise.

Exceptions

None.

Parameters

index	Specifies the replication index of the related port.
front	Specifies whether recalled messages should be queued ahead of (true) or behind (false) other queued messages.

Remarks

This operation recalls the first deferred message of this signal type at the specified port replication index. To recall the first message of any signal type, use the ProtocolRole.recallAt operation.

The first form is equivalent to calling the latter with the 'false' as the second argument.

Examples

```
port.serviceRequest().recallAt( 0, true );
```

ProtocolRole.OutSignal

Instances of this class are created by operations generated for outgoing signals of protocol roles.

Operations

invoke	Synchronous message broadcast via all port instances.
invokeAt	Synchronous message send via a specific port instance.
reply	Used to reply to a synchronous or asynchronous message.
send	Asynchronous message broadcast via all port instances.
sendAt	Asynchronous message send via a specific port instance.

ProtocolRole.OutSignal.invoke

Capsule.Message[ ] ProtocolRole.OutSignal.invoke()

Return value

An array of reply messages. The length of the array will correspond to the cardinality of the associated port. Elements of the array will be null if the message could not be delivered or no reply was provided by the receiver.

Exceptions

CrossThreadInvokeException	Synchronous messages are not supported across thread boundaries.
IllegalSignalException	This is not an out signal of the related protocol role.

Parameters

None.

Remarks

The communications services also support synchronous messaging (similar to rendezvous). During a synchronous send, or invoke, the sender is blocked until the receiver has processed the message and sent back a reply. Run-to-completion semantics are enforced, such that a synchronous invoke has similar semantics to a procedure call.

Examples

```
Capsule.Message[] replies = port.ack().invoke();
for( int index = 0; index < replies.length; ++index )
{
   if( replies[ index ] == null )
   {
      // no reply
   }
   else
   {
      // handle reply
   }
}
```

The receiver of the invoke must use ProtocolRole.OutSignal.reply to respond.

```
rtport.nack().reply();
```

ProtocolRole.OutSignal.invokeAt

Capsule.Message ProtocolRole.OutSignal.invokeAt(int index)

Return value

The reply message or null if none was provided by the receiver.

Exceptions

CrossThreadInvokeException	Synchronous messages are not supported across thread boundaries.
IllegalSignalException	This is not an out signal of the related protocol role.
PortUnboundException	The is no binding at the specified port replication index.
java.lang.IndexOutOfBoundsException	The index is negative or not less than the cardinality of the port.

Parameters

index	The port replication index.

Remarks

The communications services also support synchronous messaging (similar to rendezvous). During a synchronous send, or invoke, the sender is blocked until the receiver has processed the message and sent back a reply. Run-to-completion semantics are enforced, such that a synchronous invoke has similar semantics to a procedure call.

Examples

```
Capsule.Message reply = port.ack().invokeAt( index );
if( reply == null )
{
   // no reply
}
else
{
   // handle reply
}
```

The receiver of the invoke must use ProtocolRole.OutSignal.reply to respond.

```
rtport.nack().reply();
```

ProtocolRole.OutSignal.reply

void ProtocolRole.OutSignal.reply()

Return value

None.

Exceptions

IllegalPortReplyException	The reply must be formed using the same port on which the message arrived.
IllegalSignalException	This is not an out signal of the related protocol role.
PortUnboundException	There is no binding at the implied port replication index.

Parameters

None.

Remarks

The receiver of the invoke must use this operation to response.

Examples

```
rtport.nack().reply();
```

ProtocolRole.OutSignal.send

int ProtocolRole.OutSignal.send()

int ProtocolRole.OutSignal.send(Priority priority)

Return value

Returns the number of messages successfully queued.

Exceptions

IllegalSignalException	This is not an out signal of the related protocol role.

Parameters

priority	Specifies the priority at which the messages should be sent.

Remarks

Since a port can be replicated, this operation effectively broadcasts via all instances of the port. If you want to send only via one instance of a replicated port, use the sendAt operation.

The first form is equivalent to calling the second with the argument Priority.General.

Examples

```
port.nack().send();
port.nack().send(Priority.General);
```

ProtocolRole.OutSignal.sendAt

void ProtocolRole.OutSignal.sendAt(int index)

void ProtocolRole.OutSignal.sendAt(int index, Priority priority)

Return value

None.

Exceptions

IllegalSignalException	This is not an out signal of the related protocol role.
PortUnboundException	There is no binding at the specified port replication index.
java.lang.IndexOutOfBoundsException	The index is negative or not less than the cardinality of the port.

Parameters

index	The port replication index.
priority	Specifies the priority at which the messages should be sent.

Remarks

This operation attempts to send a single message via the specified instance of the port.

Examples

```
port.nack().sendAt( 1 );
```

```
port.nack().sendAt( 1, Priority.General );
```

ProtocolRole.SymmetricSignal

Instances of this class are created by operations generated for symmetric signals of protocol roles. Symmetric signals are those that have no data in the outgoing direction and are present with or without associated data in the incoming direction.

All operations from InSignal and OutSignal are available.

Timing.Base

The timing services provide users with general-purpose timing facilities based on both absolute and relative time. To access the timing services, you reference, by name, a timing port that has been defined on that capsule (that is, by creating a port with the pre-defined Timing protocol). Service requests are made by operation calls to this port while replies from the service are sent as messages that arrive through the same port. If a timeout occurs, the capsule instance that requested the timeout receives a message with the pre-defined message signal `'timeout'`. A transition with a trigger event for the timeout signal must be defined in the behavior in order to receive the timeout message.

Operations

cancelTimer	Used to cancel an outstanding timer.
informAt	Requests a timeout at a particular absolute time.
informEvery	Requests periodic timeouts.
informIn	Requests a timeout at a relative point in time.
timeout	Used to refer to incoming timeout events.
timeouts	Queries how many timeouts an event represents.

Attributes

rti_timeout	The code used to identify the timeout signal in this protocol role.

Nested Classes

There are no nested classes intended for use in user code.

Timing.Base.cancelTimer

boolean Timing.Base.cancelTimer(Timing.Request request)

Return value

Returns true if the request was valid and has been successfully cancelled, or false otherwise.

Exceptions

None.

Parameters

request	The timeout request to be cancelled.

Remarks

This operation guarantees that no timeout message relating to the given request will be received from the cancelled timer, even if it had already expired and the message queued.

Examples

A capsule can request a timeout once a second with the code below.

```
request = timer.informEvery( 1000L );
```

It can be cancelled sometime later with this code.

```
timer.cancelTimer( request );
```

Timing.Base.informAt

Timing.Request Timing.Base.informAt(long absoluteTimeMillis)

Timing.Request Timing.Base.informAt(long absoluteTimeMillis, java.lang.Object data)

Timing.Request Timing.Base.informAt(long absoluteTimeMillis, java.lang.Object data, Priority priority)

Return value

An object that represents this request is returned or null in the case of a failure.

Exceptions

None.

Parameters

absoluteTimeMillis	The absolute time when the timeout should occur.
data	The data to be associated with the timeout message. The default is null.
priority	The priority of the timeout message. The default is Priority.General.

Remarks

The time is expressed in milliseconds with the same time base underlying the function java.lang.System.currentTimeMillis.

Examples

These two statements have roughly the same effect.

```
timer.informAt( 1000L + System.currentTimeMillis() );
timer.informIn( 1000L );
```

Timing.Base.informEvery

Timing.Request Timing.Base.informEvery(long intervalTimeMillis)

Timing.Request Timing.Base. informEvery(long intervalTimeMillis, java.lang.Object data)

Timing.Request Timing.Base. informEvery(long intervalTimeMillis, java.lang.Object data, Priority priority)

Return value

An object that represents this request is returned or null in the case of a failure.

Exceptions

java.lang.IllegalArgumentException	The specified interval must be positive.

Parameters

intervalTimeMillis	The requested interval to the first and between successive timeout events.
data	The data to be associated with the timeout messages. The default is null.
priority	The priority of the timeout messages. The default is Priority.General.

Remarks

The interval between timeouts is expressed in milliseconds. Successor timeout events are not queued until handling of the current timeout begins. The timeouts operation can be used to discover whether other events would have occurred had the resources been available to handle them.

Examples

The code below requests a timeout event every second.

```
timer.informEvery( 1000L );
```

Timing.Base.informIn

Timing.Request Timing.Base.informIn(long relativeTimeMillis)

Timing.Request Timing.Base. informIn(long relativeTimeMillis, java.lang.Object data)

Timing.Request Timing.Base. informIn(long relativeTimeMillis, java.lang.Object data, Priority priority)

Return value

An object that represents this request is returned or null in the case of a failure.

Exceptions

None.

Parameters

relativeTimeMillis	The requested time interval to the timeout.
data	The data to be associated with the timeout message. The default is null.
priority	The priority of the timeout message. The default is Priority.General.

Remarks

The interval is expressed in milliseconds. If the interval is zero or negative, the event will essentially be queued immediately.

Examples

```
timer.informIn( 100L );
timer.informIn( 100L, new String( “Your timeout”) );
```

Timing.Base.timeout

ProtocolRole.InSignal Timing.Base.timeout()

Return value

Returns a ProtocolRole.InSignal object for access to the defer queue management operations.

Exceptions

None.

Parameters

None.

Remarks

If a timeout message is deferred, this operation is used to gain access to the defer queue management operations to purge or recall that message.

Examples

```
timer.timeout().recall();
```

Timing.Base.timeouts

int Timing.Base.timeouts(Timing.Request request)

Return value

Returns the number of timeout events represented by the current timeout message.

Exceptions

None.

Parameters

request	The timeout request to be queried.

Remarks

If the informEvery is used with a small interval or if CPU resources are unavailable, timeout events may be delayed to a time beyond what was requested. This operation returns a value that includes those missed events. When no timeout events have been missed, this operation returns the value one.

This operation always returns -1 when used with timeout requests created via informAt or informIn or with requests that have been cancelled.

Examples

A capsule might request timeout events every millisecond (at a lower priority) and adjust a representation of the time using the code below.

```
advanceTimeDisplay( timer.timeouts( request ) );
```

Timing.Request

Instances of this class are returned by the operations Timing.Base.informAt(), informEvery() and informIn(). They can only be used as the argument to the cancelTimer() and timeouts() operations of Timing.Base.

Operations

There are no operations accessible to user code.

Attributes

There are no attributes accessible to user code.

Nested Classes

There are no nested classes accessible to user code.

Exceptions

The table below lists the exceptions defined by the Java Services Library and identifies the primary usage.

AlreadyDeferredException	Deferring a message more than once.
BadIndexException	The index specified in a call to incarnate or plugIn is out of bounds or is already in use.
CrossThreadInvokeException	A synchronous message send crossed a thread boundary.
DeferralException	The common base class of all deferral exceptions.
DeferredInitializationException	Deferring an initialization message.
DeferredInvokeException	Deferring an invoked message.
FullException	An incarnate or plugIn operation applied to a full capsule role.
IllegalBindingException	An incarnate or plugIn operation would create a binding between incompatible protocol roles.
IllegalForwardException	Forwarding a message via a port which does not have the required out signal type.
IllegalPortReplyException	Replying via a port other than the one on which it arrived.
IllegalSignalException	Sending a signal via a port which does not have the required out signal type.
IncompatibleClassException	An incarnate or plugIn operation involves a class which is incompatible with the class used to define the capsule role.
PortUnboundException	Sending via a port which is unbound.
RecursiveInvokeException	A cycle of synchronous message sends.
UnexpectedException	The services library encountered an exception unexpectedly.
UnrecoverableError	The services library encountered an exception from which it cannot recover.
UnsupportedOperationException	Using an unwired operation intended on a wired port.

Index