Model Examples

RATIONAL ROSE® REALTIME

VERSION: 2003.06.00

PART NUMBER: 800-026118-000

WINDOWS/UNIX

Rational®
the software development company

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

Legal Notices

Part Number: 800-026118-000
Version Number: 2003.06.00

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

Third Party Notices, Code, Licenses, and Acknowledgements

Contents

Figures

Preface

The information in this document supersedes all other manuals and documentation included in this release.

This manual is organized as follows:

Audience

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

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

Other Resources

- Online Help is available for Rational Rose RealTime.

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

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

- To send feedback about documentation for Rational products, please send e-mail to `techpubs@rational.com`.

- For more information about Rational Software technical publications, see: `http://www.rational.com/documentation`.

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

Rational Rose RealTime Integrations With Other Rational Products

Integration	Description	Where it is Documented
Rose RealTime–ClearCase	You can archive Rose RT components in ClearCase.	▪ *Toolset Guide: Rational Rose RealTime* ▪ *Guide to Team Development: Rational Rose RealTime*
Rose RealTime–UCM	Rose RealTime developers can create baselines of Rose RT projects in UCM and create Rose RealTime projects from baselines.	▪ *Toolset Guide: Rational Rose RealTime* ▪ *Guide to Team Development: Rational Rose RealTime*
Rose RealTime–Purify	When linking or running a Rose RealTime model with Purify installed on the system, developers can invoke the Purify executable using the **Build > Run with Purify** command. While the model executes and when it completes, the integration displays a report in a Purify Tab in RoseRealTime.	▪ Rational Rose RealTime Help ▪ *Toolset Guide: Rational Rose RealTime* ▪ Installation Guide: Rational Rose RealTime
Rose RealTime–RequisitePro	You can associate RequisitePro requirements and documents with Rose RealTime elements.	▪ *Addins, Tools, and Wizards Reference: Rational Rose RealTime* ▪ *Using RequisitePro* ▪ Installation Guide: Rational Rose RealTime
Rose RealTime–SoDa	You can create reports that extract information from a Rose RealTime model.	▪ Installation Guide: Rational Rose RealTime ▪ *Rational SoDA User's Guide* ▪ SoDA Help

Contacting Rational Customer Support

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

Your Location	Telephone	Facsimile	E-mail
North, Central, and South America	+1 (800) 433-5444 (toll free) +1 (408) 863-4000 Cupertino, CA	+1 (781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 20 4546-200 Netherlands	+31 20 4546-201 Netherlands	support@europe.rational.com
Asia Pacific	+61-2-9419-0111 Australia	+61-2-9419-0123 Australia	support@apac.rational.com

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

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

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

Examples Introduction 1

Contents

This chapter is organized as follows:

Overview

Rational Rose RealTime comes installed with a variety of example models and sample RRTEI scripts. You can access the sample models and scripts in one of two ways:

- Look through the documentation abstracts included in the help. When you find an interesting sample, click on the link at the top of the abstract to view or download sample files.
- Browse the Rational Rose RealTime examples directory structure in the installation directory:

  ```
  $ROSERT_HOME/Examples (on UNIX)
  ```

  ```
  %ROSERT_HOME%/Examples (on Windows)
  ```

Note: On UNIX, the links at the top of the example abstracts do not work. You will have to browse the directory structure to open a sample.

There are four categories of examples:

Tips for Browsing Model Examples

The model examples have been developed using a set of conventions that allow you to easily navigate and understand each model:

- On the main use case diagram in the Use Case View, every model contains a textual explanation of the models domain requirements and what is demonstrated by the model. Large models may even have use cases.
- Browse model elements documentation. The documentation is meant to explain the responsibility of the element, and any special instructions that is relevant. Using the documentation pane is a good way of browsing documentation.
- Each capsule, which was designed as the top level container for the example model, is stereotyped as "Top Level". Generally, these are the capsules you should look at first to understand the structure of the example model. The top level capsule will usually contain sequence diagrams that should be viewed to understand the main interactions between the contained capsule roles.
- Each model contains sequence diagrams that illustrate example communication scenarios in the example model. A quick way of browsing sequence diagrams is via the **Browse > Sequence Diagrams** menu item.
- Before building and running a component review the component diagram named "Tips for building and running the model". The diagram contains text outlining any special requirements for building and running the model.
- Each component view will contain a package for each platform (operating system) on which the model example was tested. Each operating system component package will contain a set of components named using the following convention:

  `<component name>_<libset name>`

  For example:

  `AutoTestMarkI_x86VisualCpp60`

  `(Meaning the AutoTest capsules for the MarkI coffee machine compiled for the x86 processor with the Visual C++ 6.0 compiler)`

  `MarkI_sparcgnu281`

  `(Meaning the MarkI coffee machine capsules compiled for the sparc processor with the gnu 2.8.1 compiler)`

The example models have been tested with each configuration that exists in the Component View.

- The deployment view contains a package for each OS and processor that was used to test the example models. Each processor contains the component instances that can be run on it.

Referenced Configurations

Each model example contains the components on which the example was tested. If you want to build the example for another platform, make a copy of a component and configure appropriately.

Some models will require more work then others to port to other platforms, for this reason it may be best that you are very familiar with the model example, any supporting files, and your target environment before trying to port an example.

C++ Model Examples 2

Contents

This chapter is organized as follows:

Overview

Listed in the following table are the C++ model examples currently available. See the C++ Reference for more information regarding the use of C++ within Rational Rose RealTime models.

Model	Description
Callbacks on page 20	Provides an example of using callbacks within a Rational Rose RealTime application.
CoffeeMachine on page 29	Models a simple coffee machine that includes a complete set of use cases and test harnesses. This is a good intermediate level model.
DynamicForwarding on page 29	Demonstrates how to implement dynamic forwarding.

Model	Description
DynamicStructurePatterns on page 30	Provides examples of the dynamic structure patterns.
GameOfLife on page 31	Is an implementation of the classic Game of Life invented by the mathematician John Conway around 1970.
IntegratingData on page 31	Demonstrates how any kind of well formed data class can be integrated with Rational Rose RealTime. Uses external libraries and template classes.
IsrExample on page 32	Provides and example of using interrupt service routines within a Rational Rose RealTime model.
ObserverPattern on page 43	Demonstrates an implementation of the Observer pattern explained in *Gang of Four Design Patterns* on page 67.
SendReceiveData on page 43	Provides an example of sending data between capsules. Example includes sending by value, sending by reference, and sending subclasses.
SocketInterfaceExample on page 44	Demonstrates how to use the **CustomPeerControler** to integrate sockets into a model.
TrafficLights on page 56	This is a good starter model. It shows simple structure, inheritance, and nested behavior used in the implementation of a simple traffic light simulation.
UserPrompt on page 56	Demonstrates approaches for getting user input into a model. Uses **MFC** on Windows and **stdin** on UNIX.

Callbacks

The Callback example is intended to provide an overview of issues and development approaches/strategies related to using callback/RPC mechanisms with code developed using Rational Rose RealTime. Though this approach is neither recommended, nor demonstrates good OO practice, it has nonetheless become a communications mechanism many of our customers are familiar with and need to make use of.

For this example model, review the following topics:

The example contains the following supporting files:

For the example, there is some external code that accompanies it in the form of .cc and .h files. The external code provides the shared data resources (for this example, global variables, and a callback interface simulation functions). The files are sufficiently documented, so no further explanation is provided here.

- **External_CPP.h, .cpp** - Source files for callback interface simulation stub.
- **External.mk** - Compilation overrides file for compiling the external source code at the same time as the model is built from within the toolset.

Background Information

The callback approach to communication between application components has been used quite successfully in many different environments. Windowing systems often use callbacks to facilitate the processing involved with handling window events. Signal handlers use this approach to allow users to "register" the routine responsible for reacting to the signal. As well, many off the shelf communications packages are based on an RPC mechanism. Application users of the package register their "services" with a controller, which in turn makes these registered interfaces available to all applications needing the services. The user of this mechanism should be familiar with the Services Library and the thread architecture on their target platform. The user should also be familiar with the callback concepts.

Please note that although signal handlers (interrupt service routines) use a callback type approach, the strategy explained by this application note is not suitable for use by signal handlers. The strategy requires use of the Services Library in a way that uses system calls that are normally not allowed to be called from within a signal handler.

The example model provided is intended to be an example only and is only supported as such.

Rational Rose RealTime Constraints

The callback approach violates several key architectural premises of the capsule paradigm. The use of this mechanism with software developed using Rational Rose RealTime will thus require careful consideration as to the extent of this violation and the resultant constraints which must be placed upon the code which implements the callback mechanism. The following discussion outlines many of the areas the user will have to be aware of when using callbacks with Rational Rose RealTime.

Capsule Encapsulation

The developer using callbacks in Rational Rose RealTime must build some knowledge about the implementation of the callback mechanism into the model. Thus, you are creating a dependency between the capsule exporting a function to a callback mechanism, the Rational Rose RealTime capsule ultimately performing the work, and some piece of external code. Changes to any of those components will lead to cascading changes in other areas of your model.

Capsule Concurrency

Rational Rose RealTime maintains the concept of all capsules being separate "units of concurrency" within a model. For the most part, this is accomplished in the use of asynchronous messaging between capsules requiring the services of other capsules. By their nature, callbacks introduce a synchronous restriction into the system, since in many cases, the service being provided by the Rational Rose RealTime capsule handling the callback, will have to ensure that all necessary actions have been performed to satisfy the request and then return data to the caller. As well, all developers involved in this endeavor must also have a fairly sound understanding of how work is to be serialized in both the model and the external process.

Common, Unprotected Data Access During a Send

The Services Library is designed and implemented such that each physical thread used by the application has a single thread of control, implemented by an instance of the **RTPeerController** object. All access, to Services Library data and capsules allocated on that thread, is managed by this instance of the controller, i.e. the controller has the thread of control. Rational Rose RealTime uses "*Run to Completion*" semantics which specify that a capsule, in the process of executing a transition on a thread, must complete the transition, including entry, exit, guard, and choice-point code segments, before any other capsule on that thread may begin handling a message. Thus, there is no need to protect the access of these elements. Interactions between Services Library threads (messaging between capsules on different threads) access globally shared data through mutual exclusion resources (**mutex**). In this case,

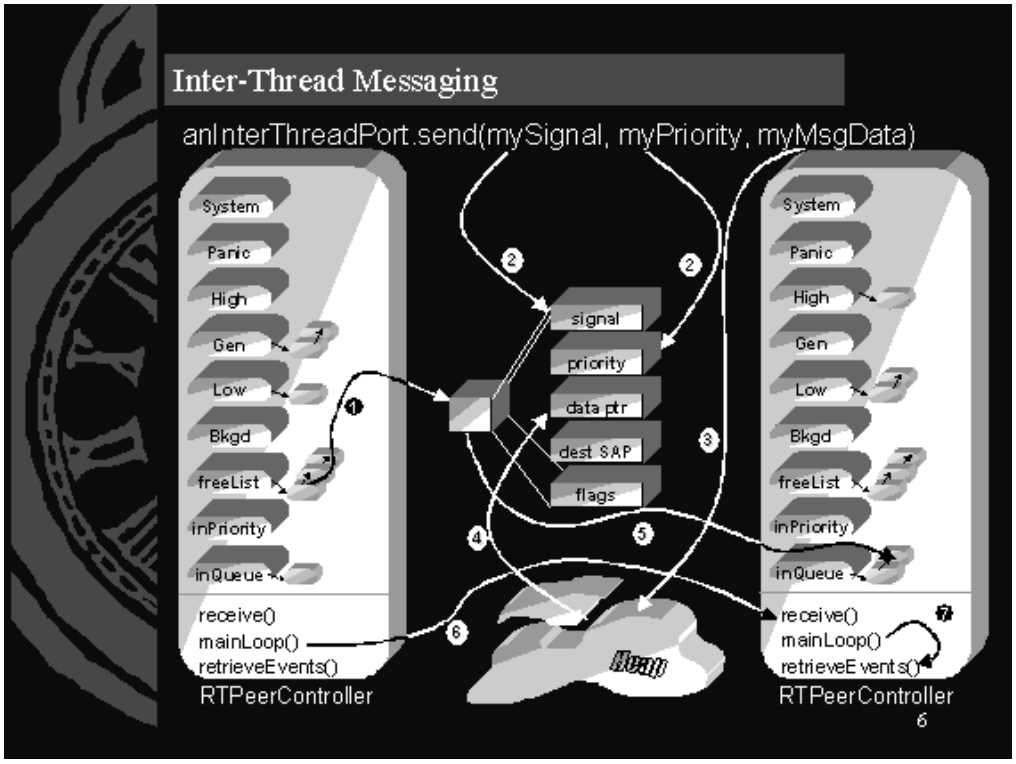
the shared data is an **inbox** (queue) for dropping off messages. During a message send, the Services Library determines if the destination port is on another thread and uses the appropriate **mutex** to protect the drop off of the message.

A callback runs on the external thread of control, which can be defined as some non-Rational Rose RealTime thread of control, such as some other user application or system services providing the callback functionality. In the next section, reference is made to a Callback Capsule. This is defined as the capsule containing the function (callback function) which is exported to the external application and used by the external application to "call" into the model to perform some action. This presents a problem to the Services Library since the assumption, as noted above, is that a component of the Services Library (the **RTPeerController**) is always in control. When the callback runs, and executes a **send()** call, as one would expect it to do, it will be obtaining a message from a list of free messages, populating the message information, including data, and then dropping it in the destination thread's **InQueue**.

If a capsule, running on the same thread on which the Callback capsule is incarnated, is performing a send() at the same time as the callback is executing, the **freeList** message queue for the thread can become corrupted. This would occur if the operating system preempted the running capsule's thread and allowed the thread on which the callback is occurring to become active and perform the callback. The corruption of the message queue can also occur if the simultaneous access were to occur when the message is freed up and returned to the free message list. Beyond just the messaging corruption, a callback executing while a capsule is executing a transition, can corrupt the data of the capsule if the callback function and the transition code both alter the data element, leaving the capsule and the thread in an indeterminate state. As mentioned above, this is a violation of the Capsule *Run-to-completion* semantics. Please refer to Figure Inter-Thread Messaging below for the sequence involved in performing a send() operation from a capsule on one thread to another thread.

Note: A callback function should only perform a send() call to a capsule on another thread. (As explained above, the Services Library uses a **mutex** during a message send to another thread. Most signal handlers (interrupt service routines) do not allow access to a **mutex** and that is why this approach is not suitable when using them.) Capsules incarnated on the callback thread must not process any messages. As well, access to data in a callback function, such as extended state variables of the callback capsule, needs to be carefully controlled and should be protected with a **mutex**, especially if the data needs to be updated.

Figure 1 Figure: Inter-Thread Messaging



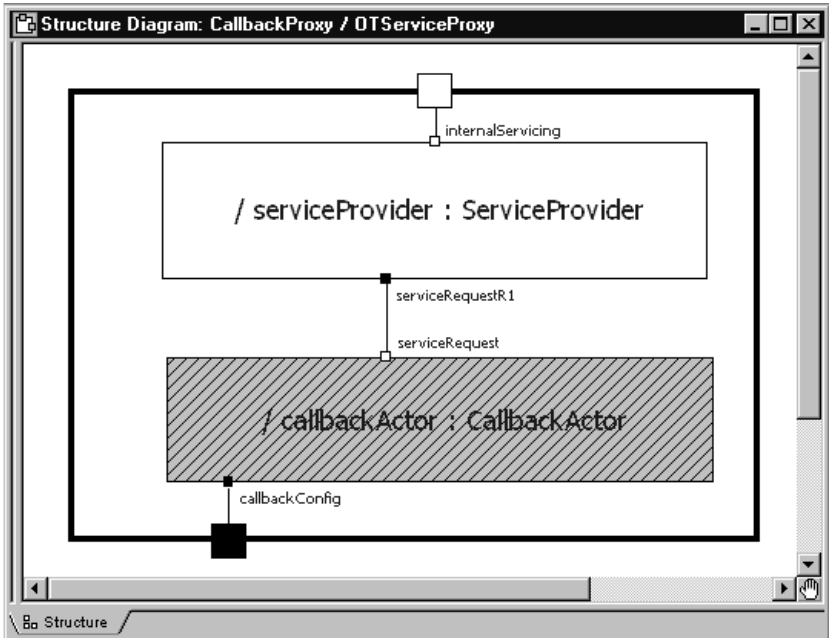
Control Flow of an inter-thread message send:

1. Get next message from the **freeList**.
2. Fill in signal and priority.
3. If sending an **RTDataObject**: make a copy of the data.
4. Fill in data field with pointer to data.
5. Queue the message on the receiving thread's **inQueue** and update **inPriority** if necessary.
6. Call **receive()** on the receiver (destination capsule's **RTPeerController**).
7. Receiver thread moves messages from the **inQueue** to internal message queues by calling **retrieveEvents()**.

Recommended Design Approach

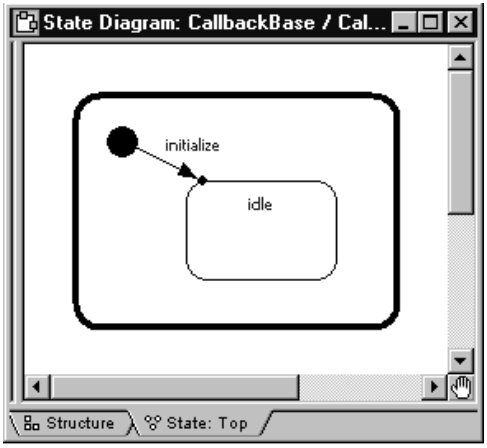
Given the constraints outlined above, the following points need to be taken into consideration when developing the software in Rational Rose RealTime to handle a callback mechanism. Please refer to the diagrams below for clarification of the structure and behavior described.

Figure 2 Figure: OTServiceProxy Capsule



Note: The **callbackConfig** port is used in the sample model to indicate to the **callbackActor** to register the callback function.

Figure 3 Figure: CallbackActor capsule behavior



Simple, Single Callback Approach

This approach assumes that there will be only one possible callback occurring at a time and that no data is returned from the callback function.

- The Callback interface should be encapsulated in a Proxy Capsule (**OTServiceProxy**). This Proxy capsule will contain an optional instance of the capsule which contains the callback function (**callbackActor**) along with an instance of the capsule which will handle the actual servicing of the callback, from a Rational Rose RealTime perspective (**serviceProvider**).
- The **OTServiceProxy** capsule must incarnate the **callbackActor** on it's own thread. This has the effect of indicating to the Services Library that the message queues will have to be protected since messages sent from this **callbackActor** will be cross-thread sends. Inter-thread message sends are thread-safe.
- The Callback Capsule (**callbackActor**) will have no behavior, other than an initial transition (see Figure: Callback Capsule behavior above). On this initial transition, it will register its services, i.e. exported functions, with the external system through whatever mechanism is provided. The **callbackActor** must not have any entry code, exit code, or transitions handling incoming signals. This effectively means that after the initial transition is run, the normal processing of the thread, on which the **callbackActor** is incarnated, is never executed.
- If the callback must be de-registered when the capsule is destroyed, this could be done in a separate function which is designated as a *destructor* function on the **callbackActor** capsule.

- Registered functions of a Callback Capsule may access extended state variables, other functions defined on the **callbackActor**, and perform *port.send()* calls destined to capsules on other threads. No messaging to capsules on the same thread as the **callbackActor** is permitted. The amount of work done in the actual function should be minimal. Some data preparation, followed by a *port.send()* to the **serviceProvider** capsule is the recommended approach.

  Note: Note that if a class (as opposed to instance) scoped operation (static function) is registered with the callback mechanism, the capsule will have to store a pointer to the instance in some global variable so that during the callback you can access the capsule instances attributes (ports …) and operations. This mechanism is used in the example model.

- All work done to actually service the callback must be done by capsules, other than the callback capsule, running on other threads. Specifically, the **serviceProvider** will handle the message sent from the **callbackActor**'s callback function, converting it into the messages and function calls needed to satisfy the service request, within the rest of the Rational Rose RealTime developed application.

Multiple Callback Approach

This approach assumes that there will be multiple callbacks occurring concurrently, but that no data is returned from the callback function.

1. Similar setup to the Simple case.
2. The Scenarios for this are:
   - Callback Capsule (**callbackActor**) may have several functions it registers.
   - Several different Callback capsules and thus Proxy capsules are developed with different exported services for each, and/or
   - Several external application threads may call back using the published functions(s) concurrently.
3. In these situations, the access to these exported callback functions must be serialized to ensure that no two functions are active at the same time. This is accomplished through one of the following methods:

- Each Callback capsule has a single function and is placed on a separate thread from every other capsule. This approach is very heavy on task resources. A separate thread is needed for each of the external threads which can access the callbacks. This approach assumes that no two external threads will access the same

callback thread at the same time. It will thus not address this third situation of concurrent calls to the same service function on the same **callbackActor** from separate external threads.

- All Callback Capsules are incarnated on the same callback thread, but access, to the published service functions, is serialized through the use of a **mutex**. On entry to a service function, the **mutex** is set, restricting all other access, the function processing is completed, and the **mutex** is released, allowing the next caller to run it's function.

The second approach, outlined in (3) is recommended for most situations since it will address all the concurrent access concerns. If there are a large number of callback capsules, service functions, and/or calling threads, placing multiple callback capsules on several callback threads, each protected with a separate **mutex**, would improve concurrency aspects of this approach, allowing several callbacks to be active, on separate threads, at the same time.

Callbacks Returning Data

This approach assumes that there will be multiple callbacks occurring concurrently and that data of some sort is returned from the callback function.

- Extension of either the Simple or Multiple cases. (see *Appendix A: Callback Function*)
- In this case, the **callbackActor** passes a pointer to the data item which needs to be returned, when the message is sent (***port.send()***) to the **serviceProvider**.
- A separate **mutex** (for synchronization purposes as opposed to mutual exclusion) is configured to allow the **callbackActor** to block, after the send, until the **serviceProvider** has finished processing the callback and has filled in the data.
- The **serviceProvider** capsule then releases the **mutex**, freeing the **callbackActor's** callback function to complete, returning the data. The code segment in *Appendix B: Callback* **WaitForData** *Function*, is an example of the code needed to wait for the data to be prepared by the **serviceProvider**.

Sample Model Outline

The sample model for Callbacks is provided in CallbacksDemo.rtmdl, which needs to be loaded into Rational Rose RealTime

The Semaphore class contains macros (in the **HeaderPreface** property of the C++ tab of this class) defines **MUTEX_INIT**, **MUTEX_LOCK**, and **MUTEX_UNLOCK** to handle the serialization. These macros must be tailored to the **mutex**/semaphore support for the target system.

There are 3 top level capsules (can be found in the **TestHarnesses** package) in the sample model. The **SimpleExampleTop** demonstrates the simplest callback scenario, whereby the callback capsule is placed on its own thread and access is made by a single simulated external application thread (actually a Rational Rose RealTime Capsule on it's own Rational Rose RealTime thread for testing). The **SimpleProtectedExampleTop** is the next level of complexity whereby the access to the callback function(s) is serialized by a **mutex**. The **SimpleProtectedDataExampleTop** capsule contains the capsules which demonstrate the last level of complexity in which the callbacks are serialized with a **mutex** and the callback needs to return some data.

For additional information, see the *External Port Service* on page 57 example.

CoffeeMachine

This is a model of a coffee machine that includes a variety of sensors and actuators, including a money box.

This is an introductory to moderate-level model. It is a good example of requirements capture and use cases, traceability from requirements to testing, automated testing, inheritance and layering. It also shows how a simple model, the MarkI coffee machine, can be scaled to include a new feature, the money box.

DynamicForwarding

This is a model that demonstrates how to implement dynamic forwarding. Dynamic forwarding allows the maintenance of a protocol class to be de-coupled from the places where it is used to forward messages. Static forwarding requires you to look at the signal that was received then decide how to forward, this is fine if the protocol is known at design time. However if the protocol is not known, or you don't want to couple your detail level code with the protocol you can use the dynamic forwarding method illustrated in this example model.

DynamicStructurePatterns

This is a large model that includes examples of the Dynamic Structure Pattern, and of the Accessor Mechanism discussed in more detail in *Safe Dynamic Structure Pattern* on page 74.

A common problem in many systems is a resource with limited availability to which a wide variety of other elements require access. There is a need to dynamically coordinate access to the limited resource. When access to the resource is required you want to set up a dynamic connection (binding) to it if the resource is available. When the use of the limited resource is complete, you want to tear down the dynamic connection. This frees up the resource and enables it to participate in a different connection. The relationship between the resource and its user is independent of the problem of managing access to the resource. The relationship could be peer-to-peer, client-server, etc.

Use the dynamic structure pattern when:

- The binding required between 2 elements is temporary in nature.
- You need to dynamically coordinate access between 2 elements.
    - Dynamically arrange the connection.
    - Coordinate the use of the connection.
    - Tear-down the connection.
- The 2 elements that need to be bound together reside in the same physical process. If distributed communication is required, you need to use the layer services (unwired ports). You may still want to use this pattern when controlling client access to the proxies or controlling proxy access to the service.
- A scalable, testable, safe solution for dynamic structure through the use of multiple containment is required.

An Accessor is a general mechanism that can be used to dynamically connect capsules.

GameOfLife

This is the classic Game of Life invented by the mathematician John Conway around 1970.

This model includes a large and variable array of capsule roles, which interact with each other through a mediator. It also shows a class utility that controls how many capsule roles are instantiated at runtime, and includes game observer and game initialization capsules.

IntegratingData

This example shows how any kind of well formed data class can be integrated with Rational Rose RealTime, so that it can be safely sent, received, and optionally observed in a Rational Rose RealTime model.

This example demonstrates integrating the following data types:

- a class with pointers
- a class with smart memory management
- a simple class (no pointers) defined outside the toolset
- a complex class defined outside the toolset
- the STL string class
- an instance based on the STL vector template class

The example also shows how to use externally created libraries as components of type External Library.

The example contains the following supporting files:

- **extclasses.h** - Defines two sample classes called Sample1, and Sample2.
- **extclasses.cpp** - Contains the implementation for the two sample classes.
- **sstream.h** - Contains a class which implements a string stream.

lib/<platform>/extclasses.lib, extclasses.a - there is a library for each platform on which the example was tested. The library contains the compiled classes defined in extclasses.h. This library is referenced from within the External Library Component defined in the toolset.

IsrExample

This model is intended to provide a simple demonstration of a strategy for interfacing a Rational Rose RealTime model and an ISR (Interrupt Service Routine).

Note: This example is intended for Unix only.

It is recommended that you read the rest of this topic before viewing the model. The rest of this topic will provide an explanation of the strategy, an overview of what the example does, and provides some detail on specific portions of the model that are important to the strategy being demonstrated.

In addition, this example discusses and makes use of a Rational Rose RealTime Services Library class named **RTCustomController**.

The example contains the following supporting files:

For the example, there is some external code that accompanies it in the form of .cc and .h files. There is a Solaris version and a Tornado version of the external code. The external code provides the shared data resources (for this example, global variables **ISRFired** and **ISRCounter**) and the ISR operation itself. The files are sufficiently documented, so no further explanation is provided here.

- **ISR_Interface_SUN5T.h, .cc** - Source files for Solaris-specific external code.
- **SUN5T_With_External.mk** - **Makefile** insert file for compiling the Solaris-specific external code. This is called from the Component::C++ Compilation::CompilationMakeInsert property.
- **ISR_Interface_TORNADO101T.h, .cc** - Source files for Tornado-specific external code
- **TORNADO101T_With_External.mk** - **Makefile** insert file for compiling the Tornado-specific external code.

Background Information

Many real-time applications require interaction with interrupts/ISRs and developing such applications with Rational Rose RealTime does not change those requirements. In general, using ISRs with an application requires knowledge of concurrency issues. When attempting to use ISRs with a Rational Rose RealTime model, this is still the case. The underlying Services Library and how a model uses them must be considered. It is expected that readers and users of the example are knowledgeable about concurrency issues and about the Services Library. Of course, it is also assumed that the readers already understand the use of ISRs, especially in their target environment.

This specific strategy and example only applies to a multi-threaded Services Library application.

The example model provided is intended to be an example only and is only supported as such.

The ISR Interfacing Strategy

Overview of the Issues

Applications that interact with devices of any kind often need to be notified of certain events. Common approaches for receiving this notification are: polling, i.e., continuously and explicitly checking for an occurrence of something, and blocking, i.e., using an operating system call to block the application (or the task) until an interrupt/ISR causes the call to unblock. From a Rational Rose RealTime model perspective, both of these approaches are still valid. Both of these approaches force the application to perform some operation to receive the event. It is never safe to attempt to call directly into the model from an ISR (or any external thread of control).

A polling approach is very simple to implement in a Rational Rose RealTime model and no special knowledge about concurrency or the Services Library is required. The simplest idea for this is to declare a Timing port (creating a port with the predefined Timing protocol) on a capsule and use it to trigger self transitions at whatever interval desired, where the self transition would check for event occurrence. However, any polling approach causes unnecessary CPU consumption. Implementing a blocking approach in a Rational Rose RealTime model is possible; however, in the simplest case of blocking in a transition, it would cause the model to block such that inter-capsule messages could not be delivered and/or processed. Placing the blocking capsule on its own Rational Rose RealTime physical thread may appear to be a quick solution. However, there are issues with this approach as well: dedicating a task or thread consumes additional resources (memory, CPU, etc.) and, the Services Library implementation uses internal messages to perform some of its operations, specifically

incarnation and destroy. Blocking the capsule would not allow these messages to be processed. For some applications these approaches and accompanying limitations/restrictions may be suitable. There is a third approach that can be used that addresses some of the issues mentioned above. This document describes the third approach.

The Strategy

The strategy explained and demonstrated by this ISR interfacing example uses a combination of polling and blocking. The strategy provides the polling portion via the normal message dispatching loop of the **RTPeerController** of the Services Library when delivering messages and the blocking portion via the normal waiting mechanism used by the **RTPeerController** when there are no messages to deliver. Using the **RTCustomController** class as the implementation class of a Rational Rose RealTime physical thread and an accompanying "ISR layer" capsule, one can provide capsule operations that change the control flow of the thread such that interaction with an ISR can be accomplished easily and efficiently. (The **RTCustomController** class is derived from the **RTPeerController** class.)

There are several model elements that need to be provided for this approach:

- A **wakeup** operation in the ISR layer capsule
- A **waitForEvents** operation in the ISR layer capsule
- A high priority event processing operation in the ISR layer capsule
- A shared data resource for passing information from the ISR to the model

A description of each of the model elements listed above, followed by a runtime scenario is a suitable way to explain them. A later description of the example model will give more concrete details because the example actually implements the scenario (and strategy) described. Knowledge of the Services Library message passing algorithm will be helpful in understanding these descriptions.

The ISR layer capsule must implement and provide two operations (as part of its interface to the **RTCustomContoller**): **waitForEvents()** and **wakeup ()**. The purpose of these operations is to provide a way for the custom controller to wait and to be woken up. A semaphore is a commonly used resource that can be used for blocking (waiting) on and signaling to unblock, so the ISR layer capsule must provide this resource. The **waitForEvents()** operation simply waits on or takes a token from the semaphore and the **wakeup()** operation posts to or gives a token to the semaphore. These operations are used in the normal processing and control flow of the Services Library during inter-thread message sending. When a controller (controlling delivery of all messages on a Rational Rose RealTime physical thread) has no messages to

deliver it sleeps by calling **waitForEvents()**. When a message from another Rational Rose RealTime thread is delivered to the sleeping controller, the delivering controller calls the sleeping controller's **wakeup()**. The ISR layer capsule also provides a high priority event processing operation that will be called from the custom controller's dispatching loop. This operation can be used to determine if an ISR has been called since the last time it checked. Detection of an ISR being called can be as simple as checking a global or shared resource such as a variable or queue that both the capsule and the ISR can access. This shared resource would preferably be something that is thread-safe or interrupt-safe. The ISR can place information in this shared resource, indicating that the ISR has been called and also providing the data of interest.

At runtime, the ISR layer capsule provides a pointer to its **wakeup()** operation for the ISR to access (as it will later call it). At some time later, the model (could be the ISR layer capsule) sets up an ISR to be called upon occurrence of some interrupt. While there are no interrupts the model, and in particular, the capsules on the Custom Controller thread, can send and receive messages as normal. During this normal processing, the ISR capsule's high priority processing operation is called each time through the Custom Controller's dispatching loop, that is prior to each message delivery. The processing operation checks the shared data resource to determine if the ISR has been called. Assuming that no interrupts have occurred, then the operation returns immediately and the dispatch loop carries on with normal processing.

At some time later the interrupt occurs and the ISR is called. The ISR places some information in the shared data resource and then uses the capsule's **wakeup()** operation to attempt to wake up the custom controller. It is not easy for the ISR to determine whether or not the custom controller is actually sleeping. Some scheme or protocol can be arranged between the ISR and the ISR layer capsule such that ISR only attempts to wake up the custom controller if the shared data resource is empty. For this to work, the ISR layer capsule's processing operation would always empty the resource upon detection that there is new data contained in it. For the ISR and the processing operation to be more efficient, the "detection" mechanism should not be a costly call. (A call to check an RTOS queue for example may be too time consuming. The detection mechanism should just be a global variable.)

When the ISR layer capsule's processing operation is called during execution of the custom controller's dispatch loop, the operation detects that the ISR has been called (the shared data resource has data in it), and then performs whatever processing is required as a result of the interrupt. There are several ways for the processing to occur. If the processing is time critical (although not critical enough that it all had to occur in the ISR), then it can occur right in the ISR layer capsule's processing operation. If the processing is not too time critical, the processing operation can send a message to another capsule in the model for processing.

This can be done with a send or invoke. For a send, being asynchronous, the actual processing of the interrupt notifications would occur later because the message will be queued within the Services Library for delivery to some other capsule, but quick, repeated notifications would be detected/picked up sooner. For an invoke, being synchronous, the processing of individual notifications will be faster, but detection of quick, repeated notifications would not be. (Invokes also require the invoked capsule to be on the same thread as the ISR layer capsule.) It basically comes down to whether queuing is desired in the model (asynchronous sends) or in the data resource (assuming a buffer or queue required in this case) shared by the ISR layer capsule and the ISR. The amount of processing done in the ISR itself must be kept to the absolute minimum. In most cases hopefully, the only processing done in the ISR is to set the shared data resource.

This strategy provides a way for an ISR to indirectly and quickly notify a capsule that an interrupt has occurred, while maintaining the run-to-completion event handling model of the Rational Rose RealTime runtime system. The capsule or model does not have to continually poll or block just for the purpose of interrupt or ISR detection. This strategy could also be used for interfacing with any external thread of control if receiving input from the external thread is considered high in priority. (High in priority refers to higher than message processing because the high priority processing operation is actually called before attempting to deliver a message.)

Please note that capsule operation providing the **wakeup** functionality **must not** do anything more than perform the **wakeup**, whatever that may be, because the thread of control is the ISR, not a Rational Rose RealTime thread.

ISR Interface Example Model

Overview of the Example Model

The example model for demonstrating the ISR Interface strategy is a very simple one. However, it should still provide a good example of the basic strategy that can be built upon for developing and using the strategy in real applications. The basic requirement of the example model (application) is that it must be notified of an interval-based timer interrupt occurring twice per second for a period of time. An ISR layer capsule, working with the **RTCustomController**, receives (detects) the notification from the ISR/interrupt. The ISR layer also interacts with the application layer. The application layer requests with the ISR layer that it be notified of the interrupts. The ISR layer starts the interval-based timer, detects the interrupts (ISR called), and notifies the application layer upon each detection. After a period of time, the application layer requests that the notifications stop.

The current example has a VxWorks ISR layer and a Solaris ISR layer. (Knowledge of VxWorks and/or UNIX would be beneficial in understanding the example description and the example itself.) The VxWorks ISR layer uses a semaphore (VxWorks) for the wait/wakeup mechanism and uses a watchdog timer for the interval-based timer interrupts. (When the watchdog timer expires, a previously user-registered operation (ISR) is called.) The Solaris ISR layer uses a semaphore (POSIX) for the wait/wakeup mechanism and uses an interval timer (**itimer**) for the timer interrupts. (When the **itimer** expires, a SIGALRM signal is sent to the process/thread, and then a previously user-registered operation (signal handler for SIGALRM) is called.) All of the target-specific code (except for ATTRIBUTEs) is located in capsule operations. The shared data resources used for both are simply two global variables, one to provide a "detection" mechanism and one to provide what may be considered as the data of interest. While it may be suitable to use a global variable for the detection mechanism, it would be better to choose a thread/task-safe or interrupt-safe data resource for sharing any interesting data, especially to allow buffering of such data. An example of such a data resource is an operating system message queue, which can often be written/sent to from an ISR. However, for the convenience of developing this example the global variable approach is acceptable.

Example Model Description

Packages and Classes of the Model

The model consists of the following packages and classes:

Package ISRLayer

- Capsule **BaseCustomIPCLayer**: skeleton behavior for set up of a capsule to use with the **RTCustomController**
- Capsule **SolarisISRLayer** (derived from **BaseCustomIPCLayer**): Solaris-specific capsule for providing set up and management of a Solaris **itimer** and for detecting **itimer** ISR calls
- Capsule **TornadoISRLayer** (derived from **BaseCustomIPCLayer**): Tornado-specific capsule for providing set up and management of a VxWorks watchdog timer and for detecting watchdog ISR calls
- Protocol **InterruptControl**: protocol for ISR and application layers to use for interrupt requests and notifications

Package Application

- Capsule **SomeInterruptProcessor**: simple capsule for requesting interrupt notifications and for processing the notifications

Package TestSolarisItimer

- Capsule **TopSolarisItimer**: test capsule (harness) for Solaris

Package TestTornadoWD

- Capsule **TopTornadoWD**: test capsule (harness) for Tornado

The ISR itself and the shared data resources used are actually located in external code (.h and .cc) that must be compiled and linked with model. (For simplicity, the compilation of the external file is accomplished by using a Target override file (see Overrides in the Online Help for a description of override files).

Class Descriptions

It is recommended that the example model be available in the toolset while reading this so that the model can easily be viewed and navigated to provide a better understanding of the descriptions. The detailed code is commented and some of the specification dialogs contain some description.

ISRLayer – BaseCustomIPCLayer:

This capsule has no detailed behavior. It simply provides a skeleton FSM that can be used for setup of a capsule to use the **RTCustomController**. It also provides three empty capsule operations that are to represent the wakeup, wait, and high priority event processing operations. Detailed behavior and additional states can be added to this capsule to provide the ISR layer capsule or a class can be derived from it to provide the specific ISR interfacing (and application layer interfacing) that is desired. The latter approach is used for this example.

ISRLayer – SolarisISRLayer:

This capsule's main purpose is to detect ISR calls and pass notification on to a capsule that previously requested such notification.

External files (source code) contain the ISR itself and the shared data resources (global variables chosen for this example).

This capsule provides the structural interface for a capsule to request and receive interrupt. This interface is provided through the **InterruptControl** protocol.

This capsule provides the behavior for:

- set up of internal wait/wakeup mechanism resource (semaphore)
- set up and management of an **itimer**
- set up of data by the **itimer** ISR

- set up of its capsule operations for use by the **RTCustomController**
- responding to interrupt notification requests
- sending notification requests
- detection of ISR call (via change in global variable)

The capsule contains the following attributes:

- **semaForSync**: semaphore id
- **internalSetupSucceeded**: flag to indicate whether set up of the semaphore succeeded
- **externalSetupSucceeded**: flag to indicate whether set up of the **itimer** ISR succeeded

The capsule has the following states:

- Reset: all set up fails
- Operational: all set up succeeds and the capsule is ready to respond to capsules and to detect ISR calls
- **WaitInterruptRequests**: awaiting request for interrupt notification from other capsule
- **InformingOfInterrupts**: other capsule has requested notifications; notifications are sent

The capsule provides the following capsule operations for registration with **RTCustomController**:

- **wakeup**(): called to wake up the controller the capsule is on
- **waitForEvents**(): called by the controller to have itself wait while there is nothing to do
- **checkInterrupts**(): called by the controller, prior to dispatching a message, for the purposes of detecting and processing "high priority events"

The setup of the semaphore and the ISR both occur during the initial transition chain. The first to occur is the internal set up - creation and initialization of the semaphore. If it fails, then the first choice point will fail, causing the entry into the Reset state. The false chain or the Reset state is where any error handling or recovery would take place. For this example, the handling only provides consumption of all received messages. If the internal set up succeeds, the first choice point passes and the external set up is attempted - set up of the **itimer** ISR (signal handler for SIGALRM). This set up also includes registering its **wakeup**() operation via the external operation

registerWakeupWithISR() such that the ISR can access it. If the external set up fails, then the second choice point will fail causing a change to the Reset state. If it succeeds, then the true chain is executed, at which time the capsule's three "controller" operations **waitForEvents**(), **wakeup**(), and **checkInterrupts**() are registered with the **RTCustomController**. Upon completion of the true chain, the Operation state is entered.

When the Operational state is entered, the **WaitInterruptRequests** state is immediately entered. At this time, the capsule waits for a request to be notified of interrupts. Note that at this time, although the **RTCustomController** is using the capsule's **wakeup**() and **waitForEvents**(), the capsule (and thread) can send and receive inter-thread and intra-thread messages as per normal. Upon reception of a **notifyOfInterrupts** signal, the **notifyOfInterrupts** transition is taken and the **InformingOfInterrupts** state is entered. In the transition the **itimer** is activated and the notification is replied to with an **interruptAccept** signal

While in the **InformingOfInterrupts** state, a **noMoreInterrupts** signal can be received, at which time the **noMoreInterrupts** transition is taken. In the transition, the **itimer** is deactivated. However, the main purpose of the **InformingOfInterrupts** state is for the capsule to detect **itimer** ISR calls. The detection is in the **checkInterrupts**() operation, which is actually called from the **RTCustomController**'s dispatching loop each time around, as opposed to from within a transition upon arrival of a message. The detection of the ISR call is determined by the **ISRFired** global variable being non-zero. If it is non-zero then (post-) processing of the interrupt occurs. First, the **ISRFired** variable is set to zero such that the ISR can set it again on the next call. (This is the detection protocol that has been arranged between the capsule and ISR. It would be practical to disable the interrupt around the section of code that accesses **ISRFired** in the capsule, but this example does no do that.) The processing that this capsule provides is only to invoke, with the **interruptOccurred** signal, the capsule that requested notifications. An invoke was chosen because it is faster than an asynchronous send.

ISRLayer - TornadoISRLayer:

This capsule is very similar to the **SolarisISRLayer**. The difference is the use of Tornado specific types and operating systems calls. The types are used for declaration of the attributes. The operating system calls are all isolated to capsule operations. Given that the two capsules are so similar it probably would not take too much extra effort to wrap the attributes and operating system calls into two external classes that provide a common interface and then just use one capsule. But for this example, the two capsules do contain target specific code. With these two capsules behaving the same, a description of this capsule is not required. It should only be noted that this capsule uses a Tornado specific semaphore (binary) and uses a Tornado watchdog

timer for the interval timer. The watchdog timer is not really an interval timer, but is only triggered once per start/activation. Because of this some special handling is required to pass information to the watchdog timer ISR such that it can start another timer within the ISR. Starting the next watchdog in the ISR instead of the capsule will provide for better accuracy (less drift) in the interrupt occurring at the desired interval.

External files (source code) contain the ISR itself and the shared data resources (global variables chosen for this example).

Application - SomeInterruptProcessor:

This capsule's main operations are to receive interrupt (interval-based timer interrupts twice per second) notification and to perform appropriate processing. This capsule uses the **InterruptControl** protocol to request interrupt notification and to receive interrupt notification.

This capsule provides the behavior for:

- requesting interrupt notification
- processing interrupt notifications for a period of time
- canceling interrupt notification

The capsule contains the following attributes:

- **waitForAcceptTID**: **RTTimerId** used to cancel repeated timeout requests while attempting to request interrupt notifications

The capsule has the following states:

- Idle: awaiting to attempt interrupt notification requests
- **WaitServiceAcceptance**: repeatedly request interrupt notifications until accepted
- **WaitForInterrupts**: receive notifications and process them

While in the **WaitServiceAcceptance** state, the capsule responds to Timing port timeouts. On a timeout transition, an **interruptRequest** signal is sent via the **InterruptControl** protocol and another timeout is set. Also in this state, the capsule responds to an **interruptAccepted** signal, at which time the **interruptAccepted** transition is taken and the **WaitForInterrupts** state is entered. In the transition, the previously requested timeout for retry of interrupt requests is canceled. In addition, a new timeout is set that represents the period of time that the capsule will respond to the interrupt notifications. For this example, 20 seconds was chosen.

In the **WaitForInterrupts** state, the capsule receives **interruptOccurred** signals. For these signals, the **interruptOccurred** self transition is taken and the interrupt notification is processed. For this example, the processing is simply to set a global variable, **ISRCounter**, to zero that was previously incriminated by the ISR. (In a real application, a global variable for data transfer would probably not be appropriate in order to avoid simultaneous access of the data buffer by the user (capsule) and the ISR. Instead, interrupt/thread safe buffers would be used.) After a period of time, the Timing port timeout signal arrives, the **timeToStop** transition is taken, and the Idle state is entered again. In the transition, a **noMoreInterrupts** signal is sent via the **InterruptControl** protocol to cancel the notifications.

In the Idle state, the capsule can receive some late **interruptOccurred** signals. They can occur because in this example it is possible (if the interval used by the ISR layer is small enough) that an interrupt notification can be sent after the cancellation of notifications has been sent.

TestSolarisItimer and TestTornadoWD:

The capsules in these test packages do nothing more than incarnate the capsules under test. There are two important things to note about the test capsules. They each contain a reference to an appropriate **ISRLayer** capsule and a **SomeInterruptProcessor** capsule with a binding connecting the references to the **InterruptControl** port on each. They incarnate the ISR layer capsule onto a Rational Rose RealTime physical thread that uses the **RTCustomController** as the implementation class.

External Code:

For the example, there is some external code that accompanies it in the form of .cc and .h files. There is a Solaris version and a Tornado version of the external code. The external code provides the shared data resources (for this example, global variables **ISRFired** and **ISRCounter**) and the ISR operation itself. The files are sufficiently documented, so no further explanation is provided here.

Expanding on the Example

When the strategy and example are understood, the strategy can be used for much bigger and better things. Here is a list of some of things one may want to use it for:

- Single interrupt notification to a single capsule (basically the example provided)
- Single interrupt notification to multiple capsules
- ISR layer could maintain a list (replicated ports) of those to be notified
- Multiple interrupt notifications to single/multiple capsules

- ISR layer could maintain a list of interrupt data (ISRs, shared data resources, and so on) and can detect calling of any of the ISRs
- ISR layer on multiple threads (each requiring the use of a **RTCustomController** on separate threads)

This ISR interfacing strategy provides a reasonably efficient approach for interfacing a Rational Rose RealTime model with an interrupt/ISR. However, it is still up to the designer/project to determine what the real performance, reliability, and response requirements are for any interrupts that may be used in their application. The strategy should be reviewed for possible incorporation, and if it appears suitable, it should be fully prototyped in the target environment to ensure that it is tested to determine if it can meet expected requirements.

For additional information, see the *External Port Service* on page 57 example.

ObserverPattern

Sometimes in a system, two or more objects need to simultaneously present different views of the same data. Whenever the data changes, often through some action of one of the observing objects, all observers need to be notified of the change. The Observer Pattern is presented in the book: Gamma, E., et al, Design Patterns - Elements of reusable object-oriented software, Addison-Wesley, 1995. This book is often referred to as the Gang of Four (GOF) Patterns Book.

This example is one of many possible implementations of the Observer Pattern. It is capsule rather than class based, and therefore differs in some ways from the class-centric implementation presented in the GOF patterns book.

This example uses dynamically incarnated capsules that are subsequently imported into an observer plug-in role.

The ability to broadcast a message to all capsules that are bound to a port with a multiplicity greater than one.

SendReceiveData

This is a simple model that includes Sender and Receiver capsules that demonstrate how to send and receive a variety of built-in data types.

Have a look at this example for information on message sends, on the correct syntax for sending a variety of data types, how to receive each of these types in another capsule, how to log the received data to the console, and how to observe these messages at run-time.

SocketInterfaceExample

This example is intended to provide a simple example of integrating socket-based IPC into a Rational Rose RealTime application.

Why use IPC?

Socket based communication appears at first glance to provide an ideal mechanism for implementing a distributed application. If connectivity were the only requirement, then sockets might be ideal, however many projects often have other, difficult to realize requirements, like:

- ease of use, including ease of changing the distribution architecture;
- the mapping of object-to-object communication and sockets;
- rerouting messages when transports fail;
- a name service so that the sender can find the receiver;
- fault tolerance so that when something fails there is a back-up;
- dealing with data representation issues in a mixed CPU environment;
- fault reporting so that when things go wrong the application can react;
- optimization for memory and performance

It is important that the reader realize that the IPC example in this note is not intended to serve as a robust IPC implementation guideline, but an illustration of a simple design technique. Building and maintaining a robust distribution infrastructure can require the resources of a dedicated team. An example of a robust, industrial strength IPC implementation designed to meet the application needs for reliability, availability, performance, fault-tolerance, and ease of use, is beyond the scope of this note.

Build Versus Buy

If you require more than a simple IPC mechanism, then you should consider Rational Connexis. Connexis works together with Rational Rose RealTime to let you model and build distributed Rational Rose RealTime applications. Built-in middleware provides an off-the-shelf communication infrastructure that solves many of the challenges common to distributed applications including object-to-object connectivity, fault tolerance, name lookup service, reliability and performance. Capsules continue

to communicate with each other in the same way as with Rational Rose RealTime - by sending messages to ports -however the receiving capsule can be in another process, or even on another processor. For more information on Rational Connexis see **http://www.rational.com/products**.

Pre-requisites

To understand this IPC example the reader should be familiar with sockets, IPC in general, multi-threaded applications and the Rational Rose RealTime C++ Services Library. This example discusses and makes use of a C++ Services Library subclass of **RTPeerController** known as **RTCustomController**. For detailed information on the **RTCustomController** and its usage please consult the Rational Rose RealTime Online Help.

Overview

In this simple example, the main loop of a **RTCustomController** waits on a socket for messages. The key benefit to this approach is that no capsule is required to block on the socket interface. In addition the capsule receiving messages from a socket can also receive messages from other capsules. To keep the example simple, the messages received will be a stream of bytes over a socket at a specific IP and socket address. There is no attempt to decompose the stream into a sequence of messages destined for other ports or capsules, nor is any encoding or decoding of data performed. This is not an example of a robust IPC solution.

Socket Example Description

This section presents an example of how a TCP/IP socket connection can be monitored by a capsule (**TCPClient**) incarnated on a **RTCustomController** thread. The server side of the connection uses the same monitoring mechanism and will not be presented here.

The client consists of three capsules:

- **SenderSameThread** (top level capsule) - Incarnates an **IPCSender** capsule on the **RTCustomController** thread.
- **IPCSender** - Controls the number of messages exchanged with the server through the contained optional capsule '**customIPCLayer**' (Figure 2).
- **TCPClient** - Capsule monitoring the IPC channel when incarnated on the **RTCustomController**. It overrides only the '**waitForEvents**' and '**wakeup**' controller functions.

Figure 4 Structure Diagram: IPCSender

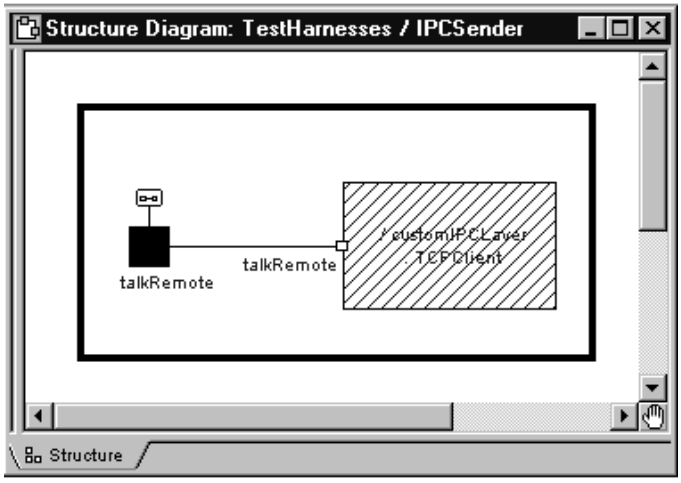
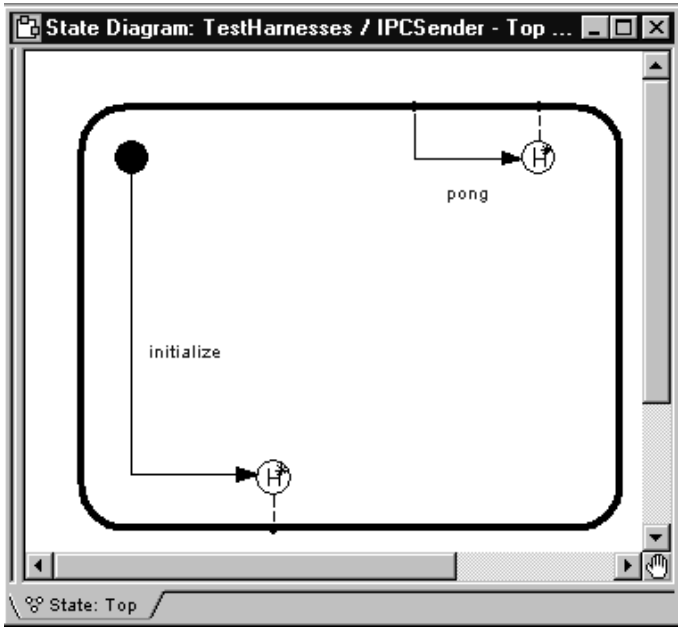
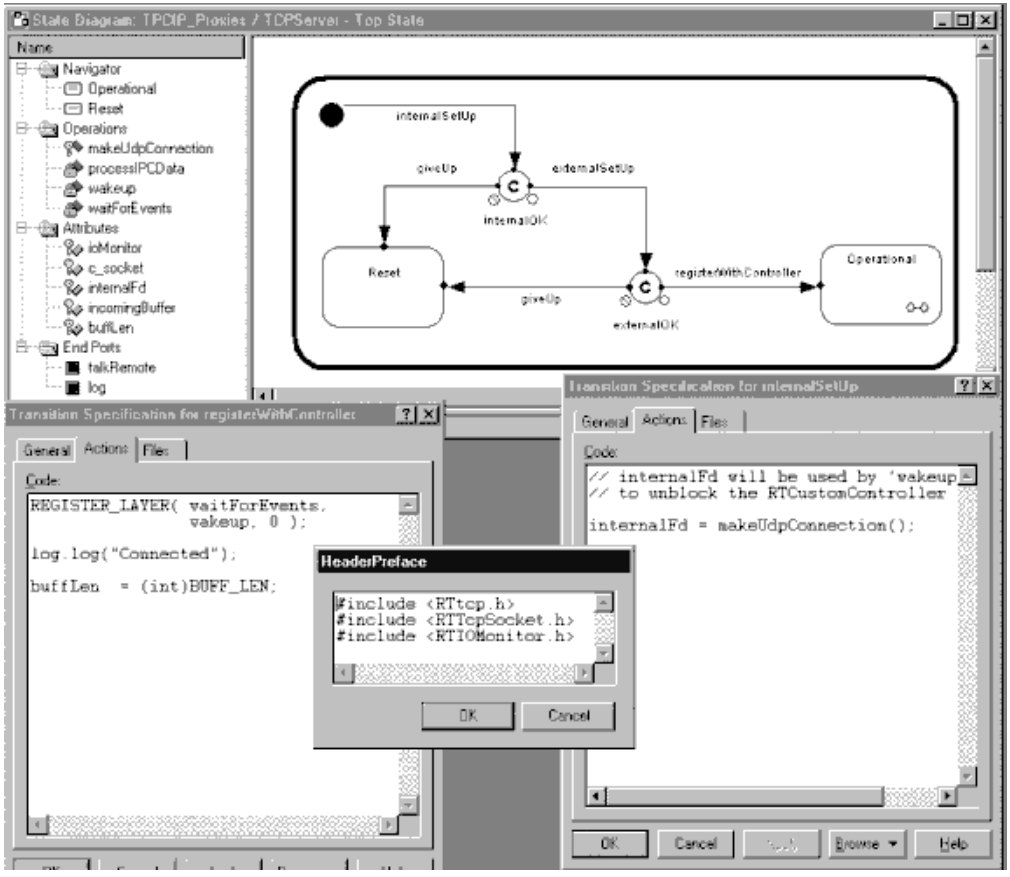


Figure 5 State Diagram: IPC Sender



The **IPCSender** incarnates an IPC monitoring capsule on the RTCustomController thread.

Figure 6 Behavior of a Capsule Monitoring a TCP/IP Socket Connection



This example uses two Services Library implementation classes. **RTTcpSocket** represents the IPC channel used in this example. The following methods are used:

- `int RTTcpSocket::create()` - makes the system call 'socket()' and sets channel attributes.
- `int RTTcpSocket::connect()` - makes the system call 'connect()'.
- `int RTTcpSocket::state()` - returns '::Established' if the socket file descriptor is usable.

- `int RTTcpSocket::read()` - reads a socket file descriptor (using the system call 'read').
- `int RTTcpSocket::write()` - writes on a socket file descriptor (using the system call 'write').
- `int RTTcpSocket::close()` - closes the underlying socket descriptor.

RTIOMonitor is the other Services Library implementation component used and is constructed to contain the parameters required by the 'select' system call.

This example uses these Services Library implementation classes for the convenience of providing a quick and simple example. However, it is recommended that customers provide their own implementation for socket related interfaces.

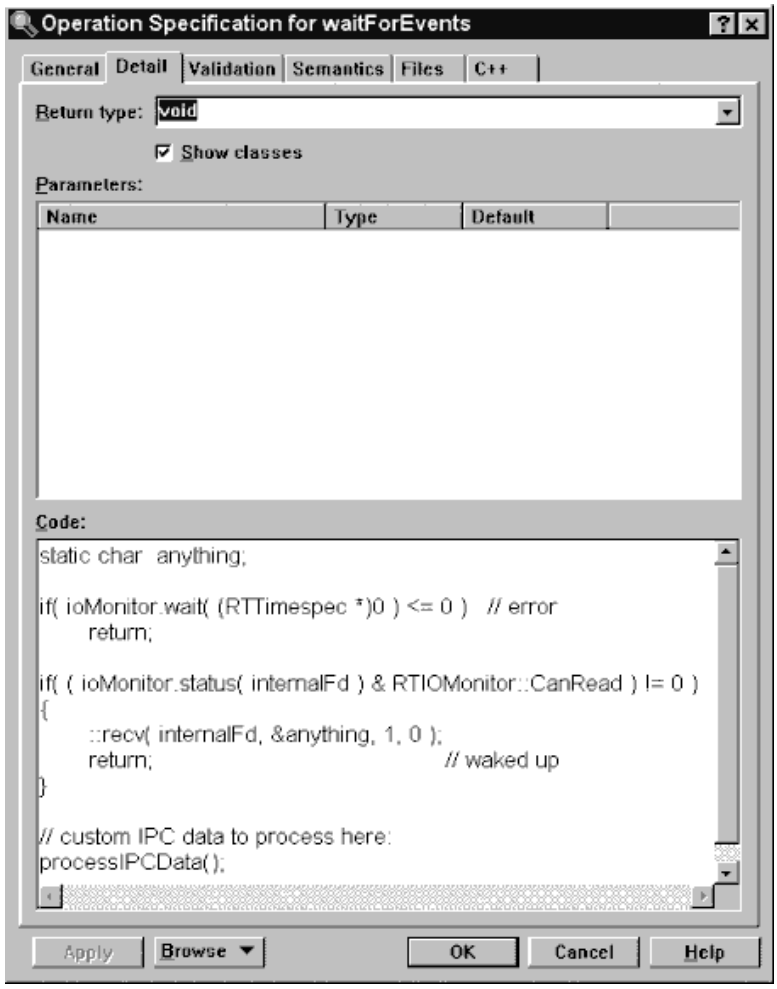
The attribute 'c_socket' (of class **RTTcpSocket**) represents the application-specific IPC channel. 'ioMonitor' (of class **RTIOMonitor**) is used here to monitor the external channel ('c_socket') and the internal socket descriptor, 'internalFd', used by other threads to wake-up the Custom Controller thread. 'internalFd' is a UDP socket, which is connectionless and has no flow control, but is suitable to use for the purpose of local host (board) communication.

Please note that for this example there is no behavior provided in the transitions leading to the Reset state (of **TCPClient**) nor in the Reset state itself. However, this is where one may want to provide error handling and recovery.

The macro **REGISTER_LAYER** takes three arguments - pointers to the 'waitFunc', 'wakeupFunc' and 'processFunc' functions to be called by the **RTCustomController**. If any of these are null pointers, the default values are used.

Figures 4 and 5 show the implementation of the functions 'wakeup' and '**waitForEvents**'. In this case, IPC data is processed at the end of '**waitForEvents**', conferring this channel a lower priority than for Rational Rose RealTime messages.

Figure 7 waitForEvents



An implementation for the **waitForEvents** function called when the **RTCustomController** has no messages left to deliver (TCPClient::waitForEvents).

Figure 8 Operation Specification for wakeup

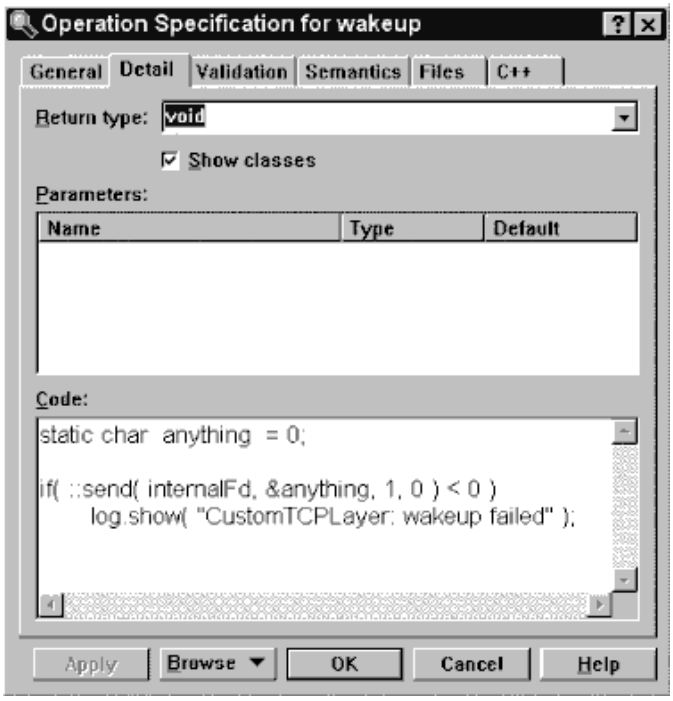
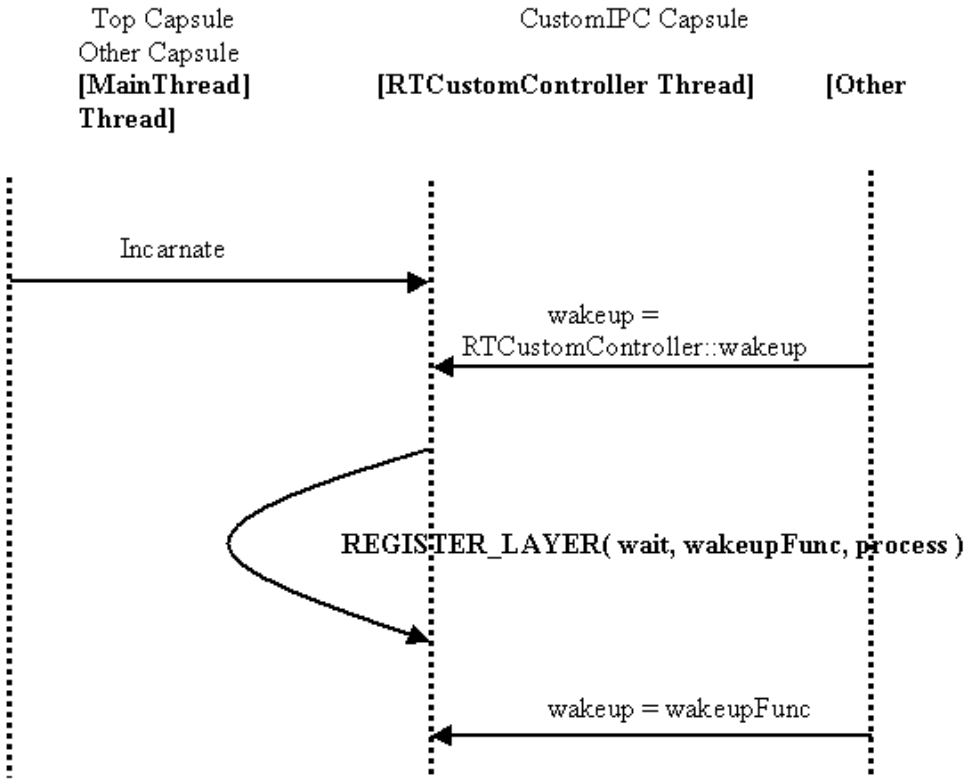


Figure 8 shows an implementation for the **wakeup** function called when the **RTCustomController** receives an inter-thread message (**TCPClient::wakeUp**).

Figure 9 shows an example of the **wakeup** control flow of an inter-thread message send related to the usage of the **RTCustomController**.

Figure 9 Wakeup control flow of inter-thread message send with

RTCustomController



Operational Concept of IOMonitor

To further understand this example, it is important to identify what the **RTIOMonitor** is doing for the IPC example. The example is based on the premise that we need to be able to cause an **RTCustomController** (essentially a thread) to block, and then to "**wake up**" when either an "internal" message (such as a message from a Rational Rose RealTime capsule on another thread/controller), or some communication from an external application (IPC – perhaps from a non-Rational Rose RealTime application) occurs.

We need to block the Customer Controller we are interested in, and then have the ability to re-activate that thread when one of two possible events occurs (internal or external **comms**)

The example uses two sockets to make this happen. One socket is created purely for internal purposes and uses UDP. The other socket is used for the external communication channel, and uses TCP/IP.

We have two sockets – the UDP socket to wake up the thread due to internal capsule communication, while incoming data on the TCP socket must wake up the thread to handle the IPC.

To monitor both of these sockets simultaneously, use an **RTIOMonitor**. This class is essentially a wrapper for the operating system **select** function. The **select** function allows the system to block "waiting for incoming data" on one of multiple sockets. **RTIOMonitor** wraps this function in a portable RRT construct for us. We add sockets to an **IOMonitor** instance, and then invoke the **RTIOMonitor** '**wait**' command. If you use a parameter for wait of NULL, the **IOMonitor** blocks indefinitely until one of the sockets it is monitoring receives some data.

RTIOMonitor is used in the example to monitor a UDP and a TCP socket, and block waiting for either of them to receive some data.

Overall Operational Profile of the Example

The **TCPClient** and **Server** capsules both include a function called **makeUDPConnection**, which is called during the internal set-up stage. Running this function, uses direct operating system calls to create a socket (using the constant **SOCK_DGRAM** to specify this as UDP). After the UDP socket is created, it is added to **IOMonitor**'s list of sockets using the command :

```
ioMonitor.add( internalFd, RTIOMonitor::CanRead );
```

This code appears in the Choice Point at the end of the **internal set-up** transition.

Note: The reason for using a Datagram/UDP socket is that it is connecitonless. This means that anything can write to the UDP socket without having to go through a notification procedure making it ideal for allowing multiple capsules to write to another thread without having to configure the connection.

At this stage, the **IOMonitor** knows about the UDP socket and nothing more.

Then, we proceed to set up the actual IPC communication mechanism. The IPC mechanism is TCP/IP-based. In this example, we use some of the TCP Socket utilities provided in the Rational Rose RealTime Run-Time Service Library. These functions are subject to change by Rational without notice, so they are not recommended for general use and as such, are not documented.

We use the **RTTCPSocket** class and related functions because it provides a portable wrapper for platform native socket implementations.

In the external configuration phase, we create one or more **RTTCPSockets** to communicate - Client and Server variants differ in their implementation here. The Server capsule creates two Sockets – one to listen for incoming communication requests (local variable called Listener), and the other is the socket which will actually

be connected when an incoming request is handled (Capsule Attribute called **c_socket**). The client capsule does not listen for incoming requests, so the client uses just one **RTTCPSocket** instance (also called **c_socket**).

Now, we have a UDP socket to detect internal Capsule messages, and we have some TCP sockets used for IPC. All of these sockets are registered with the **IOMonitor**.

After the listener socket on the server has accepted a connection from the client, the listener socket is destroyed (this example only supports a single socket connection). At this stage Client and Server capsules both use the **REGISTER_LAYER** macro, to register new **waitForEvents** and **wakeup** functions.

Now, we need to consider the operation of the **waitForEvents** and **wakeup** functions.

waitForEvents

When a Controller has consumed all available capsule messages (there are no more messages for capsules on that Thread/Controller to process) the Controller's **mainLoop** function calls the **waitForEvents** function.

In the example, the **waitForEvents** function blocks the thread by waiting for some incoming data on either the UDP or the TCP/IP socket. This blocking functionality is achieved with the following code :

```
if( ioMonitor.wait( (RTTimespec *)0 ) <= 0 )   // error

return;
```

We invoke the **IOMonitor**'s **wait** function with a time value pointer of NULL, which causes the **IOMonitor** to wait indefinitely for incoming data on any of the sockets it monitors.

The Controller/Thread is blocked at this stage, allowing the processor to handle other threads.

When some data arrives at either the UDP or TCP socket, the **IOMonitor** unblocks the thread, and processing will continue.

Two possible outcomes can occur :

1 Data arrives at the **internal** UDP port

The following code is evaluated in the **waitForEvents** function:

```
if( ( ioMonitor.status( internalFd ) & RTIOMonitor::CanRead ) != 0 )
{
   ::recv( internalFd, &anything, 1, 0 );
   return;                                          // waked up
}
```

The **if** statement uses the **IOMonitor** status function to obtain the latest status of the internal UDP socket. In this example, the UDP socket received some data, so the status returned by the **status** command is the equivalent of **RTIOMonitor::CanRead**. The **if** statement performs a bitwise **AND** operation of the status function result against the **CanRead** constant - because these are the same, we know the UDP socket received some data and the **if** statement block is executed.

Execution of the **if** statement block reads whatever data is there (it is discarded) and returns from the **waitForEvents** function. Control is returned to the **mainLoop** function of the controller, which will then obtain the real message (which will have been inserted on the Controller's Message Queue) and processing continues from there.

2 Data arrives across the TCP/IP channel

For this situation, there is no internal capsule message to handle, instead some data has come across the IPC link to our TCP Socket.

The **IOMonitor** detects the incoming data and unblocks the thread. The following code is executed:

```
if( ( ioMonitor.status( internalFd ) & RTIOMonitor::CanRead ) != 0 )
{
   ::recv( internalFd, &anything, 1, 0 );
   return;                                          // waked up
}

// custom IPC data to process here:
processIPCData();
```

When we evaluate the **if** statement, the status of the UDP port is not **CanRead**, since there is no data to be read on that socket. As a result, the **if** fails, and we continue to the **processIPCData** function call.

In the **ProcessIPCData** function, we read the data from the TCP/IP socket into a buffer and proceed to send a message to notify a capsule of the data received. Processing continues as normal from this point.

Notes on Message Passing in Rational Rose RealTime

When one Rational Rose RealTime Thread/Controller, wishes to send a message to another thread, the following short sequence is followed:

1. The sending thread/controller places a message construct on the message queue of the receiving thread/controller.
2. The sending thread/controller calls the target thread/controller's **wakeup** function.

The effect then, is that a thread/controller blocked waiting for a message, is woken up by the sender, and it then checks its message queue, finds the transmitted message and processes it (dispatching it to the correct capsule).

What we have described here, is handled in the IPC example solely by the UDP socket - we use a UDP socket simply because the **IOMonitor** can use a UDP socket as a 'blockable' construct.

To accommodate IPC, we not only need to configure the communication channel, but we need to decide where it will fit into the message handling process. Essentially, there are two options:

- **Option 1**: We can treat IPC as 'lower' priority than internal capsule messages
- **Option 2**: We can treat IPC as higher priority than internal messages.

The IPC example implements Option 1 – IPC is lower priority than internal **comms**. The rationale is that in the **waitForEvents** function, we first check whether the incoming socket data was on the UDP channel. Only after verifying that there was no internal comms via the UDP **wakeup** mechanism, do we then check for TCP/IP data. Additionally, we only check for incoming IPC data when we run the **waitForEvents** function. The **waitForEvents** function is only called when we have exhausted the processing of all internal messages.

For this situation, for ecample, if we were to receive ten internal messages and ten IPC messages, the internal messages would be handled first, and only after the controller had completed the processing of these messages, would the **waitForEvents** function be invoked and the IPC messages detected and handled.

To ensure that IPC messages are dealt with first, we can use the third parameter of the **REGISTER_LAYER** macro. This parameter is a function pointer to a function that will be executed every time the controller runs its **mainLoop** before dispatching internal messages.

For this situation, for example, if we had ten internal and ten external messages arriving simultaneously, we can handle the ten IPC messages first, and then handle the internal capsule messages.

The danger of this method is that it has an impact on performance. In this example, every time we iterate through the **mainLoop** (once for each internal message), we will run the specified function – if that situation uses processor time, the entire **mainLoop** algorithm will be slowed down.

For additional information, see the *External Port Service* on page 57 example.

TrafficLights

Traffic Lights captures the behavior of a set of traffic lights at an intersection in Austria, as observed by a North American visitor. It also includes the North American behavior for comparison.

Traffic Lights is a good starter model. It shows simple structure, inheritance, and nested behavior.

UserPrompt

This example demonstrates:

1. Two approaches for getting user input into a Rational Rose RealTime executable, one generic and the other specific to Windows configurations.
2. Integration of a Rational Rose RealTime executable with a graphical dialog DLL on the Windows configurations.

External Port Service

The External Port service example provides an API that lets non-Rational Rose RealTime threads call a function to raise an event on a port of a Capsule in a Rational Rose RealTime application.

There can be any number of external ports in any number of capsules in a Rational Rose RealTime application, however the external ports may not be replicated. Figure 10 and Figure 11 show a Class diagram and Capsule structure diagram that identify the declaration of an external port.

Figure 10 Capsule structure diagram

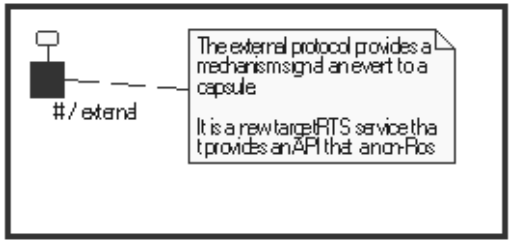
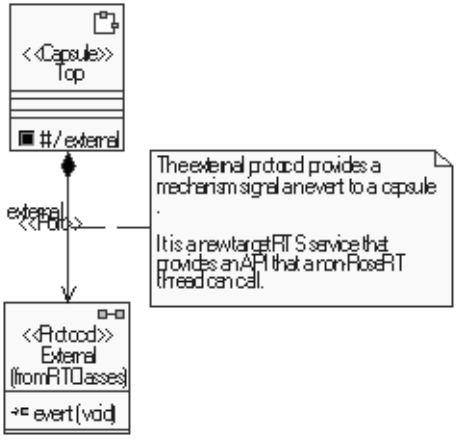


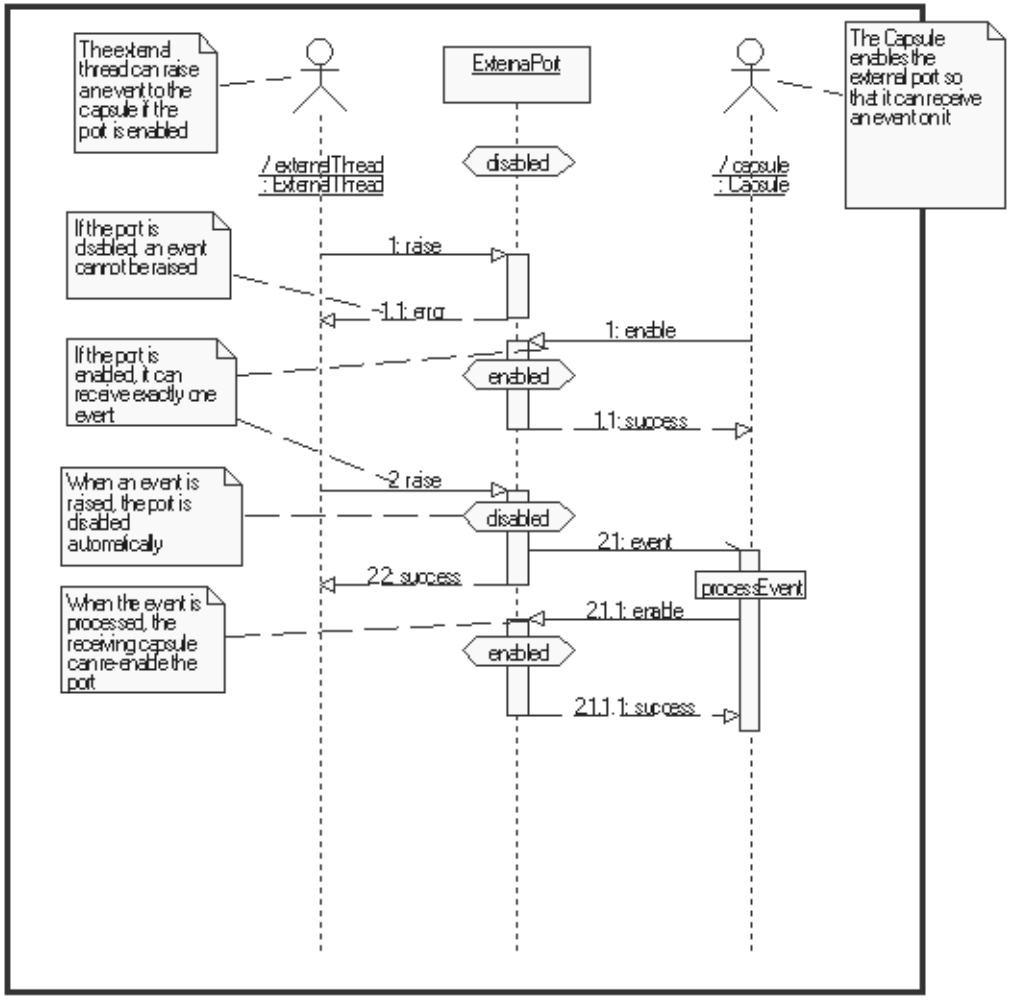
Figure 11 Class diagram



The capsule application must pass an external port pointer to the external thread so that the external thread can raise events on the port. The external thread raises an event by calling the port function `int raise()`, a non zero return value indicates that the event was successfully raised. Before an event can be raised, the capsule must enable the port to receive events. It does this by calling the port function `enable()`.

The port is automatically disabled every time an event is raised, and the port must be re-enabled before another event can be raised. Figure 12 shoes the Sequence diagram that illustrates this functionality.

Figure 12 Sequence diagram



You can find the C++ API in the following location:

$ROSERT_HOME/C++/TargetRTS/include/RTExternal.h

The following code is an extract of the relevant API from the file RTExternal.h:

```
/* For use with CPPExternal ports */
// These two functions may be used only by the thread on which the
// owner capsule executes.
void enable( void );
void disable( void );
// This function may be used only on threads other than the one on
// which the owner capsule executes.
int raise( void );
```

The C API can be found in the following location:

$ROSERT_HOME/C/TargetRTS/include/RTPublic/RTPort.h

The following code is an extract of the relevant API from the file RTPort.h:

```
/* For use with CExternal ports */
// These two functions may be used only by the thread on which the
// owner capsule executes.
void RTPort_disableExternal( RTPort * );
void RTPort_enableExternal( RTPort * );
// This function may be used only on threads other than the one on
// which the owner capsule executes.
int  RTPort_raiseExternal( RTPort * );
```

The default mutual exclusion mechanism used by the External Port mechanism is a **mutex**. If this mechanism is not appropriate, for example for notification events from interrupt handlers, then the mechanism can be changed by overriding the **unload** function for the specific target, and replacing the calls to enter and leave the critical section with code specific to your environment (such as disable/enable interrupts).

For details on how to override a TargetRTS operation, see the book *Adapting Rational Rose RealTime for Target Environments, Rational Rose RealTime*.

The default implementations for the **unload** function are:

- For C++ (using %ROSERT_HOME%\C++\TargetRTS\src\RTExternal\unload.cc):

```
#include <RTExternal.h>
#include <RTMutex.h>

// This mutex prevents a race condition between calls to disable
and raise.

// More efficient implementations may be available in some
configurations.

static RTMutex critical;

RTMessage * External::Base::unload( void )
```

```
{
critical.enter();
RTMessage * oldMsg = msg;
msg = (RTMessage *)0;
critical.leave();
return oldMsg;
}
```

- For C (using %ROSERT_HOME%\C\TargetRTS\src\Port\unload.c):

```
#include <RTPriv/Port.h>
#include <RTPubl/Control.h>
#include <RTPriv/Mutex.h>
RTMessage *
RTPort_unloadExternal( RTExternalEndPort * external,RTController
* controller )
{
RTMutex   * mutex = controller->_mutex;
RTMessage * msg;
RTMutex_enter( mutex );
msg = external->msg;
external->msg = (RTMessage *)0;
RTMutex_leave( mutex );
return msg;
}
```

C Model Examples 3

Contents

This chapter is organized as follows:

Overview

Listed in the following table are the C model examples currently available. See the C Language Guide for more information regarding use of C within Rational Rose RealTime models.

Model	Description
CardGame on page 61	Provides a C version of the Card Game tutorial model. In addition contains extended functionality to show threads, and replication in C models.
SendReceiveData on page 62	Provides an example of sending data between capsules. Example includes sending by value and sending by reference.

CardGame

This is a C version of the model developed in the Card Game tutorial. In addition the model contains extended functionality to demonstrate how to use replication, threads, inheritance in C models. This is a good model to understand before starting your own C models.

SendReceiveData

This is a simple model that includes Sender and Receiver capsules that demonstrate how to send and receive a variety of built-in data types.

Have a look at this example for information on message sends, on the correct syntax for sending a variety of data types, how to receive each of these types in another capsule, how to log the received data to the console, and how to observe these messages at run-time.

External Port Service

Fordetailed information on the External Port Service example, see the *External Port Service* on page 57.

Java Model Examples 4

Contents

This chapter is organized as follows:

Overview

Rational Rose RealTime includes several Java model examples. You can find the example Java model files in the following location:

$ROSERT_HOME/Examples/Models/Java/

You can open and view the following example Java model files:

- HelloWorldCapsule.rtmdl
- HelloWorldClass.rtmdl
- PhoneSystem.rtmdl
- ReliableServiceJava.rtmdl
- RTJavaPingPong.rtmdl

RRTEI Examples

5

Contents

This chapter is organized as follows:

Overview

Listed in the following table are the Rational Rose Extensibility examples currently available. See the RRTEI Guide for more information regarding the RRTEI and the SummitBasic script reference.

See the Tutorials for related step by step instructions for building SummitBasic scripts and using Rational Rose RealTime as an automation server.

The $ROSERT_HOME/Scripts directory contains source files for utility scripts that are used within Rational Rose RealTime. These provide good examples of using SummitBasic and the RRTEI. We recommend that you make a backup copy before modifying any of these scripts.

Script	Description
Various SummitBasic Sample Scripts on page 66	Demonstrates how to use the RRTEI to write SummitBasic scripts.
CreateCapsule1State on page 66	Visual Basic example of using Rational Rose RealTime as an automation server.

Various SummitBasic Sample Scripts

There are a number of simple SummitBasic scripts that demonstrate particular aspects of BasicScript syntax and the RRTEI interface. See the examples in the **/Examples** directory.

CreateCapsule1State

This example uses Visual Basic to use Rational Rose RealTime as an automation server. You can also consult the Tutorials for related step by step instructions for using Rational Rose RealTime as an automation server.

Patterns

6

Contents

This chapter is organized as follows:

Gang of Four Design Patterns

Software Designers have found patterns to be a useful concept. This interest has spawned a considerable patterns literature, including the book Design Patterns - Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, commonly referred to as the Gang of Four (GOF) Patterns book.

Patterns become more visually apparent when you use a visual modeling tool such as Rational Rose RealTime. In the literature patterns are usually presented using a visual notation, such as UML. With Rational Rose RealTime the visual notation is also the model from which executable code is generated.

Every domain or application can have its own characteristic patterns. This section does not claim to provide a set of patterns that will be generally useful across all domains. What we have done is to explore the patterns presented in one popular book, and see how they apply to a variety of models that have been done using Rational Rose RealTime.

The patterns in the GOF Patterns book are presented in the context of a user interface domain, which is quite different from what is typically found in real-time applications. However the abstract aspects of some of these patterns are obviously of general interest to real-time developers.

In this section, we briefly present five concrete examples loosely based on the patterns presented in the GOF Patterns book. These are not necessarily the patterns that will make sense in your applications, and we are not suggesting that you take any of these and blindly apply them to your projects. Instead the intent here is to present the

concept of patterns as they apply to systems built using capsules. We hope that these examples will get you thinking about the patterns that apply within your own applications.

We suggest that you look over this section and the sample models in conjunction with a copy of the GOF Patterns book.

Several GOF patterns are at least partially contained as part of the Rational Rose RealTime paradigm - for example the Façade and State patterns. A capsule has much in common with a Façade. State is a first class concept within Rational Rose RealTime.

The four examples presented here are:

- *Mediator Pattern* on page 69
- *Chain of Responsibility Pattern* on page 71
- *Factory Method Pattern* on page 72
- *Observer Pattern* on page 72

For more information on The Gang of Four design patterns and book, you may want to visit the following site:

http://hillside.net/patterns/DPBook/DPBook.html

Complete information on the book is as follows:

Title: Design Patterns : Elements of Reusable Object-Oriented Software

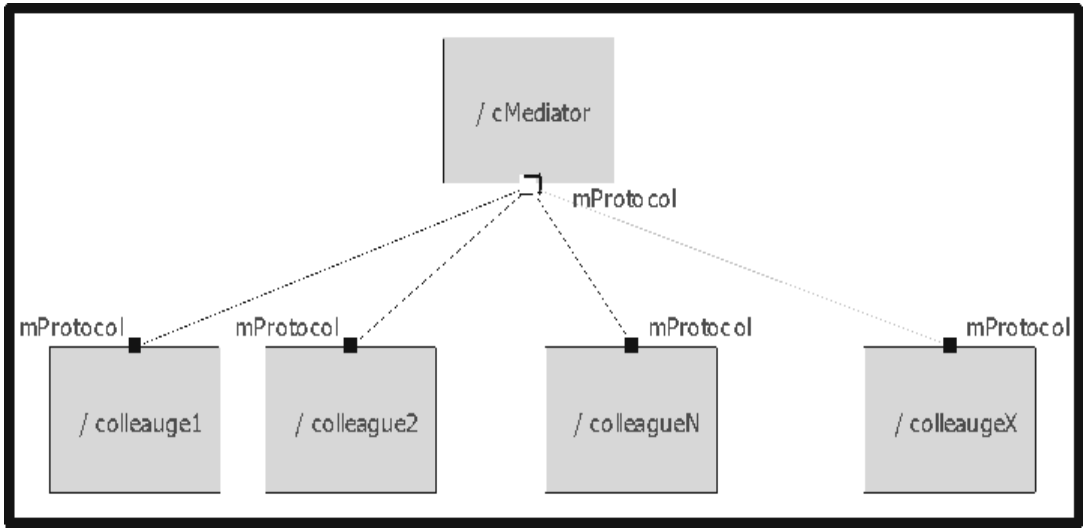
Authors: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

ISBN: 0-201-63361-2

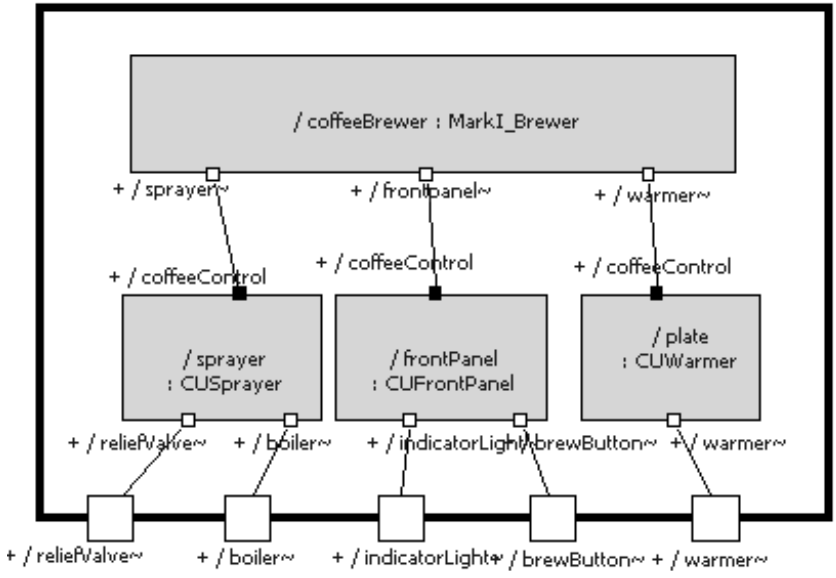
Details: Addison-Wesley, 1994

Mediator Pattern

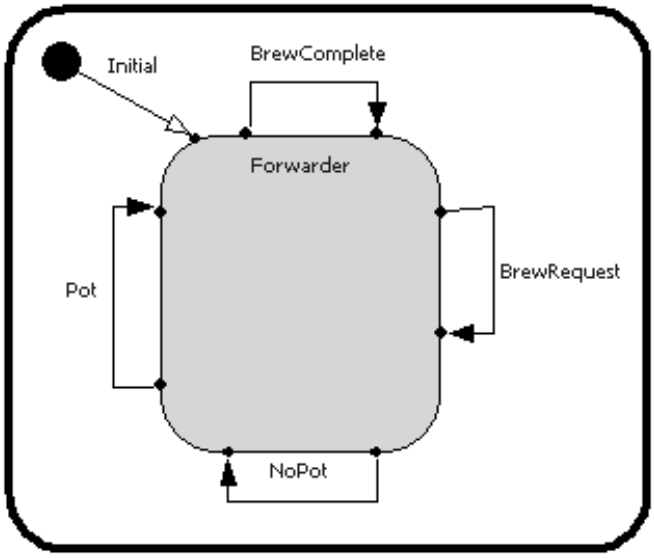
A Mediator capsule mediates or controls the interactions between a set of two or more colleague capsules. It provides for very loose coupling between capsules.



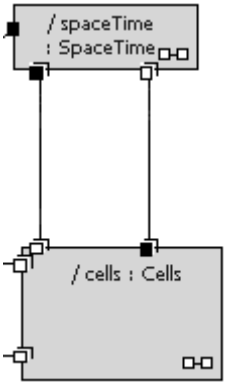
See the **MarkI_Container** or **MarkII_Container** capsules in the **CoffeeMachine_MoneyBox** example model. The **MarkI_Brewer** and **MarkII_Brewer** capsules function as concrete mediators, while **CUSprayer**, **CUFrontPanel**, **CUWarmer** and **CUCashBox** function as concrete colleagues.



Mediator capsules tend to have modeless behavior; they forward messages between ports and do not retain any state history. The state diagram of **MarkI_Brewer** is:



The Game of Life example model provides another example of a Mediator. The **SpaceTime** capsule mediates between a large number of identical Cell capsules. Cells have no direct knowledge of who their neighbors are. **SpaceTime** does know who every Cell's neighbors are, and is able to forward each message from one Cell to all of that Cell's neighbors.



Chain of Responsibility Pattern

The GOF book describes the intent of the Chain of Responsibility pattern as follows:

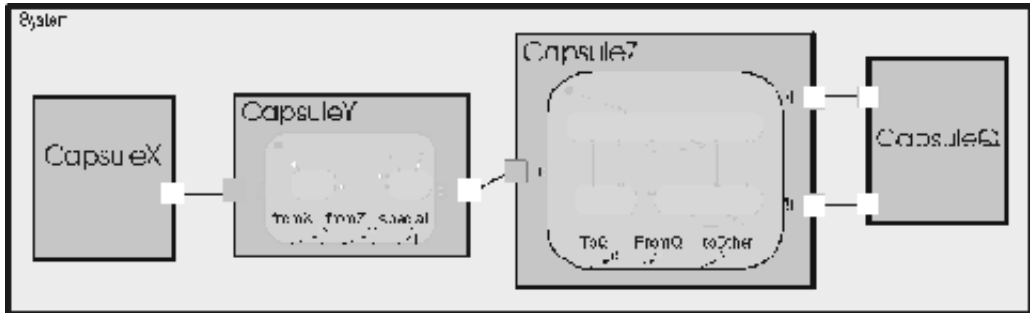
"Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it."

The Message Forwarding example model is a very simple example of this. Capsules are always de-coupled from the receiver of the messages they send, because all that a capsule does is send a message out a port, and the capsule has no knowledge of how or even if the port is bound. In the Message Forwarding example, CapsuleA sends a message to CapsuleB which directly forwards it to CapsuleC.

A somewhat more complex example is found in the **AlarmClock** model used in the Rational Rose RealTime Evaluation Workshop. This model is connected to an external Java application which includes a number of buttons that users can press to update the time and adjust various alarm settings. It also displays the currently set time and various status messages. It allows the user to control the Rational Rose RealTime model, and immediately view the results. All interaction with the Java GUI passes through one capsule, called CapsuleX in the diagram below.

Some of the messages arriving at CapsuleX have to do with setting up a socket connection between the Rational Rose RealTime model and the Java GUI. These messages are processed directly by CapsuleX. Any other messages it forwards out its port which is bound to CapsuleY. CapsuleY is responsible for converting certain combinations of GUI button presses into higher level commands which it eventually passes on to CapsuleZ through its port. CapsuleY immediately forwards all messages that it doesn't process on to CapsuleZ. CapsuleZ in turn processes some messages, but directly forwards everything having to do the setting of alarms on to CapsuleQ.

Thus, there's a chain of responsibility starting with CapsuleX and continuing on to CapsuleQ. Each capsule in the chain has a chance to process each received message if that is part of its responsibility. Otherwise it forwards it on to the next capsule in the chain.

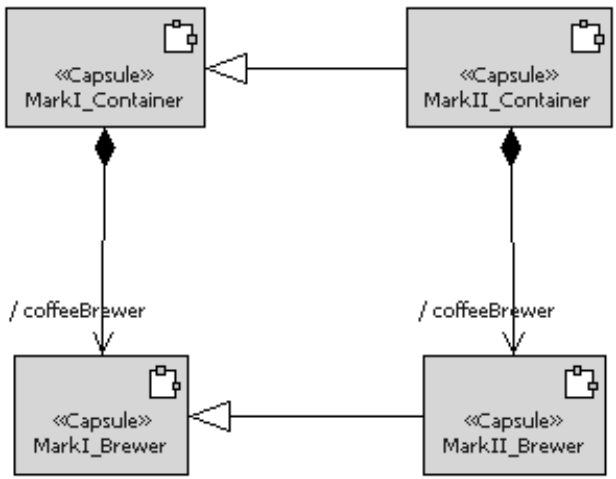


Factory Method Pattern

The GOF Patterns GOF says about the Factory Method pattern:

"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."

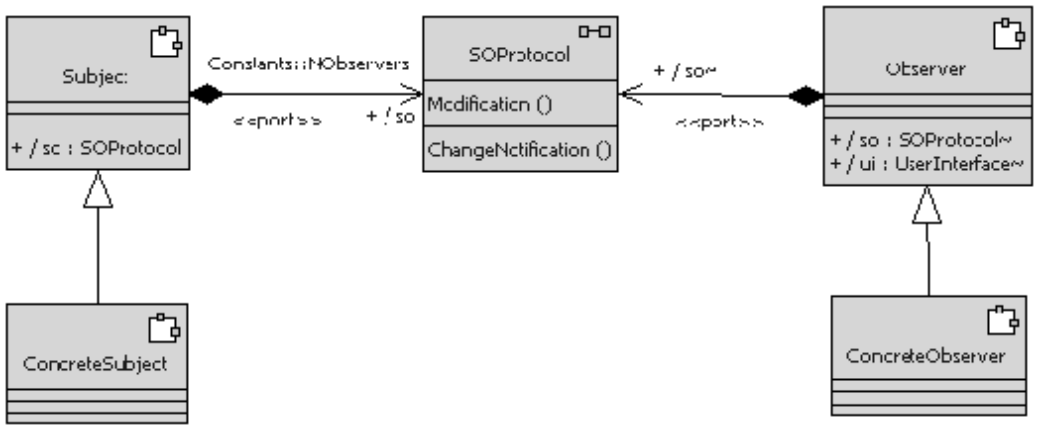
The **CoffeeMachine_MoneyBox** model includes an example of the Factory Method Pattern. **MarkII_Container** is a subclass of **MarkI_Container**, and **MarkII_Brewer** is a subclass of **MarkI_Brewer**. At the same time, **MarkI_Brewer** is part of **MarkI_Container**, and **MarkII_Brewer** is part of **MarkII_Container**. This sets up parallel inheritance hierarchies as shown in the following class diagram:



Observer Pattern

Sometimes in a system, two or more objects need to simultaneously present different views of the same data. Whenever the data changes, often through some action of one of the observing objects, all observers need to be notified of the change.

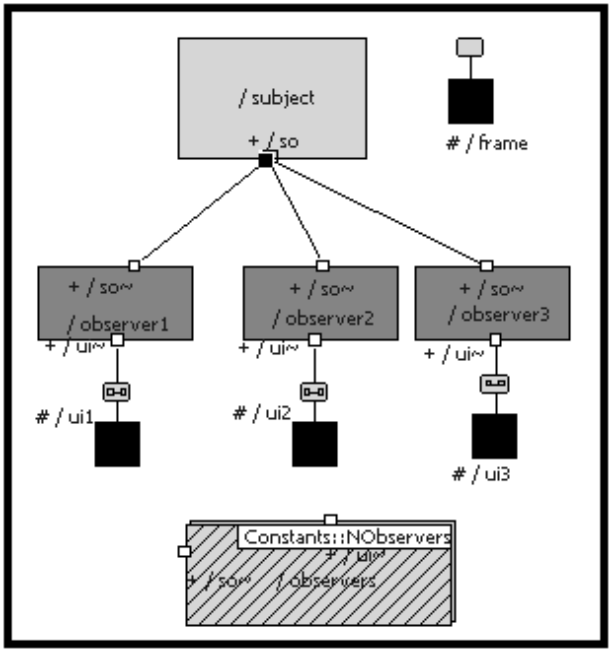
For an example of one quite simplistic implementation of the Observer Pattern using capsules in Rational Rose RealTime, please load and run ObserverPattern.rtmdl. Mirroring the description in the GOF Patterns book, this model includes four capsules - Subject, Observer, **ConcreteSubject** and **ConcreteObserver**, related as shown in the following diagram. The **SOProtocol** protocol class includes signals that allow concrete subjects and observers to communicate with each other.



The following diagram shows the structure diagram for **ObserverPattern**, the top capsule in the system. There is one subject, with some variable number of observers.

The observers are dynamically incarnated at runtime, and are then imported into the **observerP** capsule plug-in role, which has a connector to the subject. The reason for doing this is to simulate a larger system where a capsule might be incarnated at one location in the model, and then imported into a different role for connection to a subject.

In this example, the subject does not keep track of the observers that it is connected to. When it receives a Modification message from any one of the observers, it simply sends a **ChangeNotification** message back out the so port. This acts a broadcast to any capsules that happen to be connected to the so port, because the message send is not to any specific instance of the port, which effectively makes it a broadcast.



To clarify what's happening, look at the transition code for this simple model. Then build and run the model. Follow the instructions contained in the "Tips on running this model" diagram within the Component View.

Safe Dynamic Structure Pattern

A common problem in many systems is a resource with limited availability to which a wide variety of other elements require access. There is a need to dynamically coordinate access to the limited resource. When access to the resource is required you want to set up a dynamic connection (binding) to it if the resource is available. When the use of the limited resource is complete, you want to tear down the dynamic connection. This frees up the resource and enables it to participate in a different connection. The relationship between the resource and its user is independent of the problem of managing access to the resource. The relationship could be peer-to-peer, client-server, etc.

Use the dynamic structure pattern when:

- The binding required between 2 elements is temporary in nature.
- You need to dynamically coordinate access between 2 elements.
    - Dynamically arrange the connection.
    - Coordinate the use of the connection.
    - Tear-down the connection.
- The 2 elements that need to be bound together reside in the same physical process. If distributed communication is required, you need to use the layer services (unwired ports). You may still want to use this pattern when controlling client access to the proxies or controlling proxy access to the service.
- A scalable, testable, safe solution for dynamic structure through the use of multiple containment is required.

An Accessor is a general mechanism that can be used to dynamically connect capsules.

- *Motivation* on page 75 - The forces and types of design problems which led to the development of the safe dynamic structure pattern.
- *Applicability* on page 79 - When you should use safe dynamic structure.
- *Participants* on page 79 - Description of the capsules and their purpose.
- *Consequences* on page 81 - The benefits of using safe dynamic structure.
- *Implementation* on page 83 - Things to consider when applying the pattern to your problem.
- *Accessor Capsules* on page 86 - An Accessor capsule is a general mechanism, that amongst other uses, can be used to help implement safe dynamic structure.

Motivation

Design Problem

A common problem found in many systems is that there is a resource with limited availability to which a wide variety of other elements require access. There is a need to dynamically coordinate access to the limited resource. When access to the resource is required you want to set up a dynamic connection (binding) to it if the resource is available. When the use of the limited resource is complete, you want to tear down the dynamic connection. Thus freeing up the resource and enabling it to participate in a

different connection. The relationship between the resource and its user is independent of the problem of managing access to the resource. The relationship could be peer-to-peer, client-server, etc.

For example, consider a client-service type of system. We have a service that can handle **MaxServiceRequests** from clients at any point in time and a **MaxClients** number of clients that want to make use of the service. **MaxClients** is greater than **MaxServiceRequests**, making the service a limited resource. The service is implemented by a capsule and will respond to requests from other capsules. We want to dynamically arrange a connection (binding) between the Client capsule and the Service capsule only when the client requires the service to perform one of its tasks.

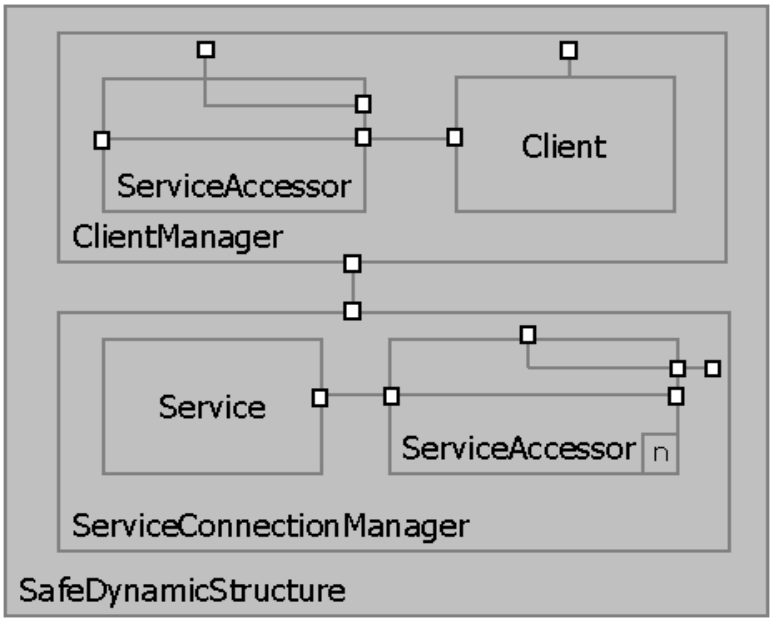
Usually designs which require dynamic structure make use of multiple containment. With multiple containment, a single capsule instance can exist in more than one capsule role at a point in time. This allows you to dynamically configure the structure of the system during execution.

In our example, the client identifies that it needs to perform some work that requires the use of the service. A request for access to the service is then made. If the service is available, it is expected that the bindings between the client and the service be dynamically setup. When the client finished its task, it provides notification that the service is no longer needed. It is expected that the bindings between the client and the service then be dynamically torn down.

Forces

When we make use of dynamic structure, we want to do so in a safe fashion. Therefore, we need to carefully consider how will the connection be arranged. Which capsule should be involved in two roles (that is, be imported to create the dynamic binding)? Where is the dynamic relationship to be managed? Can we take full advantage of structure to set up the dynamic binding and minimize the amount of detail code required? How scalable is the solution? Will it handle increases in the number of clients, number of clients requests, etc.?

Below is a simplified view of a safe dynamic structure framework. The **ClientManager** implements the policy for obtaining access to the service. The **ServiceConnectionManager** manages the access to the service. The **ServiceAccessor** is the same capsule instance contained in both aspects at run-time (i.e. multiple containment). The Client communicates directly with Service through the **ServiceAccessor**.



In a simple scenario, the client identifies that it has a task to perform. The **ClientManager** decides what service is required and makes a request to the **ServiceManager**. The request contains the id of the **ServiceAccessor**. The **ServiceManager** decides if the service was available. If it is, it imports the **ServiceAccessor** so that it is bound with the service. This results in a connection between the client and the service (they are dynamically bound). Messages coming from the client go through the **ServiceAccessor** and arrive at the Service. Similarly, the messages from the Service go through the **ServiceAccessor** and arrive at the client. The **ServiceAccessor** acts as a conduit for the messages. The **ServiceAccessor** also acts as a conduit for future messages (regarding this particular connection) between the **ServiceManager** and **ClientManager**. There is a **ClientManager- ServiceManager** communication path on a per-client-service interaction basis.

We wanted the Client of the service to concentrate on its primary responsibilities. The policy for establishing a connection in the model example is simple, but in a full robust system it can become quite complex. Therefore **ClientManager** is responsible for requesting the establishment of the connection. It knows what services are required for the task (there could be more than 1), where to obtain particular kinds of services, what the retry policy is if the service is initially unavailable, the order in which to obtain and release the services, and so on. This allows the Client to concentrate on its primary responsibilities.

The **ServiceConnectionManager** manages the access to the service. It is responsible for importing/deporting the user of the service into a multiple containment relationship. We don't want the service to be imported/deported elsewhere because this would result in the **ServiceConnectionManager** losing control over it, leaving it vulnerable to errors in the user of the service. For example what if the user of the service never gives it up, what if they destroy the service, what if they import the service again after they release it, etc. These types of errors would not only affect the **ServiceConnectionManager**, but they could potentially cause unexplained behavior for other users of the service.

We also don't want to try and import the client directly into a slot connecting to the service either. In the system, there are likely to be many different types of clients each having different interfaces (ports) for communicating with other areas of the system. It would be difficult for all of them to share the same inheritance hierarchy in order for them all to be compatible with a single slot with which to access the server. Instead the element that is to be imported is a **ServiceAccessor**. The **ServiceAccessor** has only relay ports which are bound to conjugated relay ports on the other side of the capsule. The binding passes straight through the capsule. The **ServiceAccessor** also does not have any behavior. This type of capsule is also sometimes known as a pass-through capsule. It is essentially a "proxy" object that allows two completely independent capsules to band together.

We wanted to use structure (multiple containment) and explicit binding between ports rather than unwired ports. This way the connections are visible, observable and highly controllable. The layer service used by unwired ports does not have these properties.

We want to minimize the amount of detail code necessary and instead take advantage of structure. We want to avoid the use of data structures to keep track of requests issued, which services are in use, etc. These data structures don't usually scale up. As the number of clients and services in the executing model grows, the size of these structures increases along with the time to manage and maintain them. They are a source of hidden complexity and potential error.

The **ServiceAccessor** has relay ports for the communication between the Client and the Service, plus it has relay ports to allow for communication between the **ClientManager** and the **ServiceConnectionManager**. **ClientManager** has a communication path directly to **ServerManager** on a per-client-service interaction basis. Messages regarding the establishment and tearing down of the dynamic connection are sent along this path. Using the **ServiceAcessor** in this fashion provides the "context of the request" to both the **ClientManager** and the **ServiceConnectionManager** allowing us to minimize the amount of detail code they have and the amount of data sent in messages.

Applicability

Use Safe dynamic structure when:

- The binding required between 2 elements is temporary in nature.
- You need to dynamically coordinate access between 2 elements.
    - Dynamically arrange the connection.
    - Coordinate the use of the connection.
    - Tear-down the connection.
- The 2 elements that need to be bound together reside in the same physical process. If distributed communication is required, you need to use the layer services (unwired ports). You may still want to use this pattern when controlling client access to the proxies or controlling proxy access to the service.
- A scalable, testable, safe solution for dynamic structure through the use of multiple containment is required.

An overview of the solution and the forces that led to its development are described in *Motivation* on page 75. The benefits of using this safe dynamic structure pattern are discussed in the Consequences section.

Participants

ClientManager

The **ClientManager** is responsible for requesting the establishment and destruction of a connection(s) needed by a client to do work. It manages the policy for the establishment of a connection(s) between the client and the service(s)

- can determine the best place to obtain a service if offered by more than one **ServiceConnectionManager**.
- can implement the policy for setting up the connections if more than one dynamic connection is required (for example, order of setup, what to do if one of the connections can not be made, etc.). It also ensures all connections have been setup before the client performs its task.
- implement the retry policy when the service connection can not be obtained. (i.e. wait and retry, abort, retry immediately, etc.).
- implement the tear down policy when the client is finished its task. (i.e. release the services in a particular order, delay releasing some of the services in case the client requires them again very soon after, etc.)

Client

The Client capsule represents the element requiring the use of a service in order to perform a task. It does not know which specific service is required, it simply identifies the task it wants to perform and waits for notification to go ahead. In the model example it is a very simple capsule for purposes of showing the dynamic structure pattern. Its complexity does not impact the solution. It might have ports for interfacing with other capsules. It may in turn contain capsule roles. It may be a fixed capsule role (as shown in the model example), but is more likely to be incarnated or imported into the capsule role in the **ClientManager**.

Accessor

The Accessor is an abstract capsule. It contains the relay ports needed for the coordination and tear down of the connection. It does not have any behavior.

ServiceAccessor

The **ServiceAccessor** is a subclass of the Accessor capsule. It additionally contains the relay ports specific to the protocol between the Client and Service capsules. It also does not have any behavior. It simply acts as a conduit for messages. The solution easily allows different accessors to be derived based on the nature of interfaces of the services.

ConnectionManager

The **ConnectionManager** is an abstract capsule. It is responsible for providing the framework for establishing and tearing down a dynamic connection. Its client and coordination capsule roles are based upon the Accessor capsule (also abstract).

ServiceConnectionManager

The **ServiceConnectionManager** specializes **ConnectionManager** to support a connection to the Service Capsule. It is responsible for controlling access to the service. The framework easily allows different service connection managers to be derived based on the nature of the service to be accessed.

The **ServiceConnectionManager**:

- determines if the service is unavailable (busy), and notifies the requestor.
- sets up the dynamic connection between the client and the service and notifies when the connection is complete.
- destroys the dynamic connection between a user and the service when notified the service is no longer required.
- confirms when the dynamic connection is about to be torn down.

Service

The Service is a component in the sense that it is a non-trivial, nearly independent, and reusable part of a system that fulfills a clear function. The Services fulfils this function for numerous and possibly different types of clients. The Service has a restriction that it can interact with only **MaxServiceRequests** clients at any point in time.

Consequences

Safe Dynamic Structure has the following benefits:

1. Safe Dynamic Structure

It is ensured that the connection will be set up before the client attempts to use it. The connection will be torn down only when the user of the connection is finished with the service. There is very little detail code and no connection data is maintained reducing the risk of coding errors. The complexity regarding the state of the connection is not implemented by detail code, but rather by structure. The service library's information is used rather than the application duplicating the information (further reducing the risk the information getting out of sync with the actual system).

2. High Cohesion

Each capsule has a single well defined responsibility. The **ClientManager** manages only the policy for establishing the connections. It does not create or destroy the connections. The **ServiceConnectionManager** is responsible only for the controlling access to the service. The **ServiceAccessor** acts only as a conduit for messages. There is no behavior in this capsule. The capsules being connected require no knowledge of who or how they are being connected. The user of the service must however adhere to the **clientTransaction** protocol ("go" and "done" messages).

3. Low Coupling

Capsules are communicating only with the other capsules in their layer of abstraction or lower. The communication between capsules is specific to the purpose of their functions. None of the capsules are acting as routers of messages. The number of messages to setup and terminate a connection is minimal.

Only the **serviceAccessor** capsule id is sent as data in the initial request for service. The communication between the capsules involved is purely signal based (no detailed data required). The protocols are request - success/failure in nature. The results of the request do not need to be interpreted nor analyzed.

The capsules do not know or depend upon how the other capsules perform their tasks in response to messages sent. The **ClientManager** does not know anything about the service's implementation nor how the connection will be built. The **ServiceConnectionManager** does not need to know all the client's interfaces (only the **serviceAccessor** is imported). The client identifies the task it wants to perform, it does not need to know which particular services are needed, where they are located in the system, how to connect to them, the retry policy, etc. It is completely isolated from the details of the connection policy that is implemented by the **ClientManager**. Both ends of the dynamic connection are unaware that they are part of a multiple containment hierarchy.

The Client, **ClientManager**, **ServiceConnectionManager,** and Service capsules can be developed independently. This is due in part to the low coupling and high cohesiveness of the elements and in part to the logical layering of the model. The elements being developed do not mix low level concerns from the problem domain with the high level concerns of the solution domain. The teams working on the Client and Service can be specialists in the intricacies of the client and service. While the teams working on the **ClientManager** and **ServiceConnectionManager** can be specialists for higher level solution concerns such as connection policies.

4. Testable

You can easily test the capsules which make use of the connection without establishing a connection. Neither the client nor the service participate in the establishment nor tear down of the connection. The interaction protocol between the client and service is independent of the manner in which they are connected. If they could be tested independently before introducing the dynamic relationship, they still can. For unit testing purposes, you can use fixed capsule roles for the connection between the client and the service in a test container. The test container would tell the client to go immediately in response to the **needToDoWork** request.

The establishment of the dynamic relationship can be easily tested. The parties being connected are independent of the dynamic relationship. The example model uses very simple client and service capsules to focus on illustrating the safe dynamic structure pattern.

5. Scalable and flexible

The subclass hierarchy allows you to easily customize the solution for different types of and services. The use of the **ServiceAccessor** allows you to easily customize the solution for many different types of clients.

The high level of cohesiveness results in a flexible solution. The **ClientManager** and **ServiceConnectionManager** capsules can be customized according to your system's particular requirements.

The solution is scalable since it takes advantage of the structure information maintained by the service library. Increasing the number of clients, services, etc. do not cause any internal data structures to grow.

Increasing the number of client results in an increased number of service accessors, there is no affect on the **ServiceConnectionManager**. Increasing the number of **ClientManagers** results in an increase in the replication factor of the **serviceAccess** port on the **ServiceConnectionManager** only. Increasing the number of Service capsules instead of replicating the port on the Service capsule has no affect.

Implementation

Consider the following when implementing Safe Dynamic Structure pattern:

1. ServiceAccessor

When using the **ServiceAccessor** as a conduit for messages, the messages appear to need to pass through an extra pair of ports before they arrive at their destination. While it may look like there is overhead in the use of the **ServiceAccessor**, this is not the case. When the UML model is compiled (i.e. code is generated to represent the model), the relay ports are optimized away in the generated code. Thus messages sent from either side of the **ServiceAccessor** arrive directly at their destination's message queue.

2. Use of the Coordination Capsule Role

The coordination capsule role is used to convey the result of the request to access the service (**serviceReady**, **serviceUnavailable**). In the success situation, the result is the **serviceAccessor** is imported twice (once as a coordinator, once as a client). The benefit of using the coordination role is it makes it simple to ensure the **serviceAvailable**

message is sent to the right requestor. If you wanted to send this message when the **serviceAccessor** is in its client role, you would need to know which slot it was imported into (in order to know the port index on which to send the message). This can still be determined, it requires looking at each client instance and determining if it equals the one which was just imported. Use the frame service's **incarnationAt** method to get each client instance's id. If **MaxServiceRequests** is small you may find this more efficient than doing the second import. Keep in mind though that the improvement would only be noticeable if the **ServiceConnectionManager** handled a high volume of requests and the cardinality of the client capsule role is low.

3. Service Role Cardinality

The example shows a single service with a port with a cardinality greater than one. There are no restrictions in the pattern preventing the Service from having a cardinality greater than one. In the end the cardinality of the client capsule role just needs to match the number of bindings to the Service supported.

4. ServiceConnectionManager

The example shows a **ServiceConnectionManager** which does not interact with the service. When building your model, you will want to determine if the your Service needs to notify the **ServiceConnectionManager** of errors. Whether the **ServiceConnectionManager** needs to reset it on termination of a connection, and so on. As well, the **ServiceConnectionManager** may also implement some usage restrictions, for example limiting the length of the access. The **ServiceConnectionManager** could be extended to support your requirements.

5. Fixed Client and ServiceAccessor Capsule Roles in the ClientManager

The example shows fixed Client and **ServiceAccessor** capsule roles in the **ClientManager**. Depending on your requirements, you can instead import or dynamically incarnate the Client and **ServiceAccessor** in the **ClientManager**. In fact if you are importing the Client, you may prefer to import a pass through representative of the Client. Possibly even reusing this pattern.

Building an Application Using the Safe Dynamic Structure Pattern

When applying the pattern to your application, you may find it easier to start with the sample implementation given in the example model.

- Import the safe dynamic structure package from the model example into your model.
- Determine the capsules which correspond to the "Client" and "Service" components. Determine the protocol(s) which govern their communication.
- Create a subclass of the Accessor capsule. Specialize it by adding the relay ports for the protocol(s) just identified.
- Create a subclass of the **ConnectionManager** capsule. Specialize it for your service by:
    - overriding the client capsule role's class with the Accessor subclass just created.
    - overriding the client capsule role's cardinality with a constant that represents the maximum number of users of your service allowed.
    - creating a capsule role for your capsule representing the service. Connect up the bindings between the client and the service.
    - overriding the choice point in the behavior to check if active is less than the constant representing the maximum number of users allowed.
- Modify your client component to utilize the **ClientTransaction** protocol.
- Create a new capsule representing your **ClientManager**. This capsule will be very specific to your application since it implements your connection and retry policies. The **ClientManager** in the example shows a simple implementation of such a capsule. When creating your **ClientManager**, you may find it easier to create it from a copy of the **ClientManager** capsule. To create a copy of the **ClientManager**, create a subclass of the **ClientManager** capsule and then delete the inheritance relationship. When you delete the inheritance relationship, activate the check box to absorb all superclass properties. This way you will have a copy of the **ClientManager** that you can modify as drastically as desired.

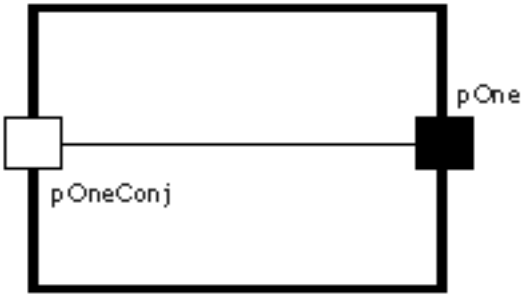
Accessor Capsules

This document briefly introduces the concept of an Accessor capsule. For detailed examples, please look at and run the various systems contained in the Rational Rose RealTime **AccessorExample** model.

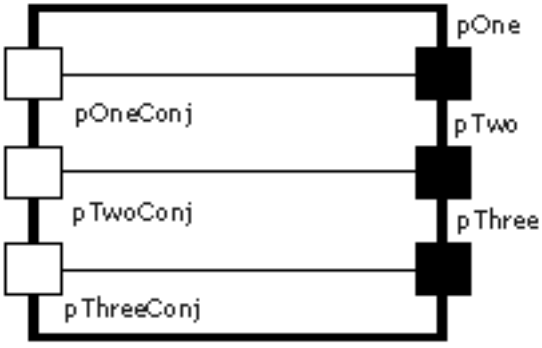
What is an Accessor

An Accessor is a capsule that contains two or more relay ports. It must be possible to legally connect each relay port to at least one other compatible relay port on the Accessor. A pure Accessor contains no end ports and no internal behavior. At least two ports must be internally connected for an Accessor to be usable at run-time.

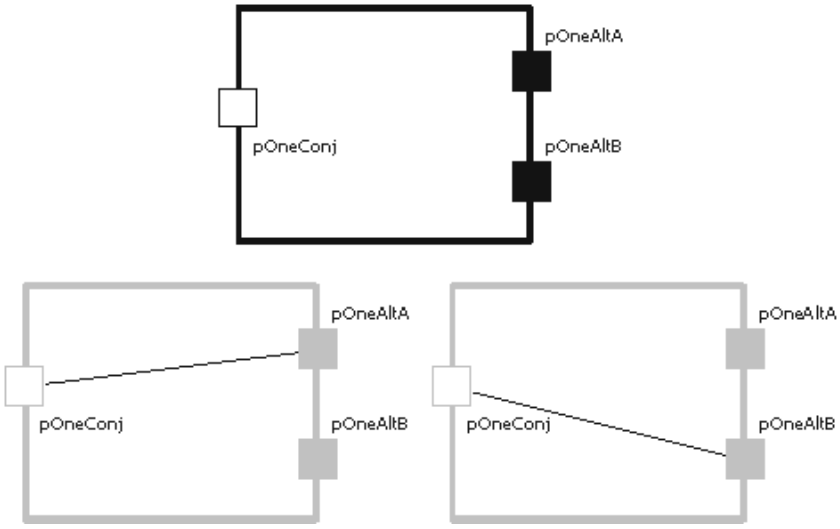
This is the simplest possible Accessor:



The following diagram shows a more complex Accessor:



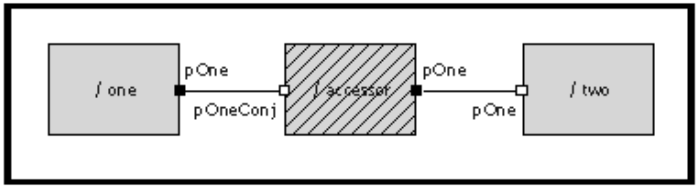
The following diagram shows a family of Accessors that belong to the same inheritance hierarchy:



Some Uses for Accessors

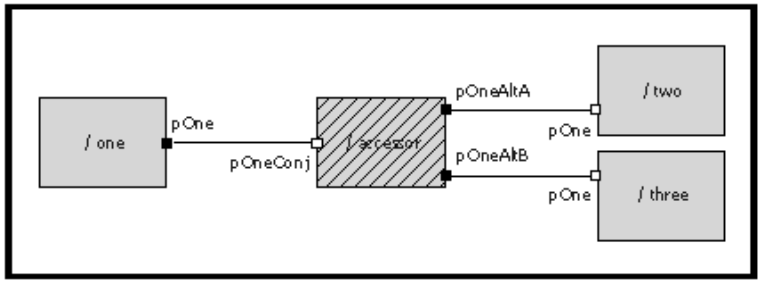
An Accessor can be used to dynamically bind pairs of capsules at run-time.

In the following simple configuration, the optional Accessor functions as an on/off switch. If the Accessor is currently incarnated (if the switch is on), then messages between capsules one and two will arrive at their destination. If the Accessor is not currently incarnated (if the switch is off), then messages will not be able to get through. The containing capsule controls whether or not messages arrive by dynamically incarnating and destroying the Accessor.



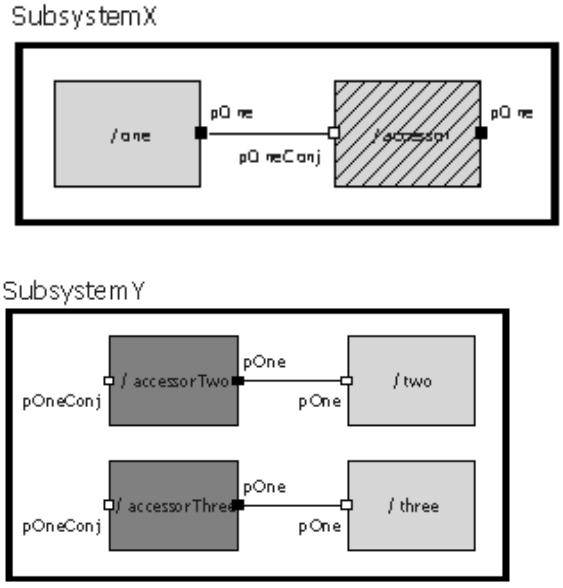
In the next configuration, the Accessor serves as a three-way switch.

- If the Accessor is not incarnated, then the switch is off.
- If the Accessor's **pOneConj** port is internally connected to port **pOneAltA**, then message flow will be switched between capsules one and two.
- If the Accessor's **pOneConj** port is internally connected to port **pOneAltB**, then message flow will instead be between capsules one and three.



An Accessor can also be used to dynamically connect capsules contained within multiple subsystems, using multiple containment. An Accessor is incarnated within Subsystem X and is then passed in a message to Subsystem Y using either a wired or unwired communication path (not shown in diagram). Subsystem Y imports the Accessor into one of several possible plug-in capsule roles, effectively connecting capsules contained within the two subsystems.

In the following diagram, at run-time, capsule one can be connected to either capsule two or three depending on whether the **SubsystemX** accessor is imported into **SubsystemY** is **accessorTwo** or **accessorThree** plug-in capsule role.



Index

A

B

C

D

E

F

G

I

J

M

O

P

R

S

T

U