# Modeling and developing embedded Java applications

Presented by the Rational Developer Domain


## Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. Before you start ..................................................................................... 2
2. Rational Rose RealTime basics ............................................................ 4
3. Build a model within RoseRT .............................................................. 7
4. Add code and build the model .......................................................... 24
5. Test and alterations ........................................................................... 28
6. Wrap up ............................................................................................ 30
Section 1. Before you start

About this tutorial

Telecoms, data processing and network management all operate using a real-time operating system (RTOS). Unlike the typical desktop application, this means that they need to be able to respond to activity from the user, external device or an internal trigger immediately. There are no hour-glasses in an RTOS.

The problem is that developing in this environment can also be quite complex. Instead of the usual sequential or event driven nature of your typical desktop application, you need to work with concurrent systems and processes and state-based environments. A typical development and modeling system is difficult to use when designing your application because they are incapable of modeling the various systems operating simultaneously within a typical RTOS and application.

Rational Rose RealTime, on the other hand, provides a complete environment for modeling your application. Using UML as a basis, it is able to produce code and your final application directly from within the modeling environment. In this tutorial you’re going to look at a basic message passing application designed to work within an RTOS. You'll build the model within RationalRose and then use that to produce a Java application.

The application is a simple alarm system, with different sensors and a simple alarm bell. You'll concentrate on one section of the alarm, a fire alarm, using a smoke detector as the trigger.

Key topics covered in this tutorial include:

° Specifying a design
° Translating the design to an application model in Rational Rose RT
° Extending and updating the structure
° Generating code and target specification
° Testing and re-modeling the application

This tutorial is useful for developers interested in using Rational Rose RealTime (RoseRT) to design and model applications using physical diagrams, thus simplifying the functional testing process by avoiding the need to develop the code.
Prerequisites

RoseRT is designed to work on a number of environments and with a range of different platforms. RoseRT is available for Linux, a selection of UNIX variants and Windows NT/2000/XP. You can also use it to develop applications using C, C++, or Java code as the deployment language, and it’s possible to migrate a model from one language to another.

In addition to all of this, RoseRT can also work with and create native code for a number of deployment environments. These include Linux, QNX and LynxOS, among many others. As with the target deployment language, the model can be migrated to work with an alternative target environment, so you can use RoseRT to develop a Java application that can be deployed among a number of target environments.

To complete the steps in this tutorial, you should have the following software installed:
° Rational Rose RealTime. Download a trial version.
° Java compiler, such as the Java SDK from Sun Microsystems.
° A copy of a make tool, such as GNU make or the Microsoft nmake.

See Resources on page 30 for more information

About the author

Martin C. Brown, a Studio B (http://www.studiob.com/) author, is a former IT Director with experience in cross-platform integration. A keen developer he has produced dynamic sites for blue-chip customers, including HP and Oracle and is the Technical Director of Foodware.net. Now a freelance writer and consultant, MC, as he is better known, works closely with Microsoft as an SME, is the LAMP Technologies Editor for LinuxWorld magazine, a core member of the AnswerSquad.com team and has written a number of books on topics as diverse as Microsoft Certification, iMacs and open source programming. Despite his best attempts, he remains a regular and voracious programmer on many platforms and numerous environments. MC can be contacted at questions@mcslp.com, or through his Web site at http://www.mcslp.com.
Section 2. Rational Rose RealTime basics

Visual modeling

The first stage to any application development process should be to build a model -- a non-working example -- of how the system works. You can use this model to examine the operation of the system, follow the necessary execution paths and connected fragments in order to study your application without ever having to write any code.

Many traditional modeling systems rely on a complex organization of documentation and text based descriptions of the process and program flow. This can be complex both to produce and to follow when building an application. With Rational Rose RealTime (RoseRT) you can design and model your application by using physical diagrams. These diagrams demonstrate the components in the system and enable you -- and other programmers -- to see at a glance how an application works and how the components should work together to produce the final application.

Like all good programming practices, the visual modeling system works on the basis of abstraction. In programming, abstraction is used with methods, functions, objects and classes. Depending on the level of the component you are looking at you can get different information about the component in question. Each level of the system is an abstraction of the previous. For example, a class is an abstraction of the sub-classes or objects it defines.

Within RoseRT, different diagrams and views exist within the visual model for different abstractions of the application model, including looking at specifics such as classes, but also at different components and execution sequences.

For the system in this tutorial, you'll create a model for an alarm system. This enables you to look at most of the different diagrams within the system and how they work together to produce an application.

Role of the Unified Modeling Language (UML)

The visual modeling within RoseRT is based on the Unified Modeling Language. UML is now the accepted standard for visual modeling after years of competition between a number of different companies and organizations and their respective solutions.

UML is actually a general purpose modeling language which can be used to
model all sorts of different systems, including business processes, objects, components and entire software applications.

RoseRT enables you to model an application using the UML system, but hides you from the complexities of the system itself. You'll take a closer look at the UML components in the Building a Model within RoseRT section.

Rational Rose RealTime approach

Rational Rose RealTime uses the application modeling approach and enables you to build and visualize your application entirely within the RoseRT environment. More than a simple modeling tool, RoseRT can also simulate the application that you have modeled, simplifying the functional testing process by avoiding the need to develop the code.

But the code isn't foreign to RoseRT. Using model-driven development techniques, RoseRT enables you to convert your application model directly into a code-based implementation of the application. This means that you can use RoseRT to analyze, design, implement, build, execute, debug, and test an entire application. The model you create actually is the application. If you need to change the application, change the model, and let RoseRT do the rest.

This eliminates one of the most annoying and frustrating elements of the modeling process, the translation of the model to the code. The entire process is driven by the model and you no longer have to synchronize between different teams and developers to achieve the results you need.

More significantly, the ability to execute models -- even incomplete ones -- enables you to continuously test and adjust your application directing development, helping you to identify bugs and produce higher quality software while still keeping to a healthy schedule and delivery time.

Real-time and embedded systems

Embedded systems are often based on fairly simple processors which are built into an overall system. You can find embedded systems everywhere, from washing machines to DVD players, from phones to car engine management systems. Unlike more traditional computing platforms such as the one you are probably using right now, embedded systems are not normally exposed directly to the user. Instead, the entire system works on a system of external stimuli.
For example, the embedded system within a washing machine sits and does nothing until the dial or controller on the front of the machine, controlled by the user, tells it to start a particular program. There is no way for the user to do anything with the machine other than get it to wash clothes.

Typically embedded systems are reliant on a real-time environment. They can accept input from a range of systems (sensors, controls, etc.), as well as initiating processes based on these stimuli. Responding to these stimuli, by definition of a real-time system, must happen quicker than in a typical operating system. On a desktop OS, you're used to seeing a spinning beach ball or hourglass when you wait. Now imagine the same operating system in a car when the breaks are applied!

Ultimately, the combination of an embedded and real-time system leads to a relatively high level of complexity and a low-tolerance to errors. You're used to your desktop computer occasionally crashing or an application failing, but when was the last time your washing machine chose to use a boil wash instead of the usual economy? More importantly, if you have a modern vehicle, when was the last time your car decided to turn right when you turned the wheel left?

Both of these cases are examples where the quality of the software that was produced was extremely high, mostly because it had to be. This is exactly why technologies like UML exist to enable designers to model the system, and why tools like RoseRT are so useful, because they enable the designers to build and test the applications.

Now that we've gone over the basics of the RoseRT application and the use of application modeling, let's have a look the basic process of creating a model within RoseRT.
Section 3. Build a model within RoseRT

Target design

For this tutorial, you're going to use an alarm system as an example of the embedded system you are trying to develop. You can see a basic diagram of the alarm system below.

Figure 1. The structure of the alarm system

The stimuli on the left in the figure should be relatively obvious. Basically, you have three different types: a smoke sensor, which will pick up a potential fire, a sensor on a window, which indicates a probable break-in, and a sensor on the entry door.

In each case, the stimuli triggers a particular event, and it's fair to say that there is the potential for one or more stimuli to happen at the same time. For example, you can walk in the door as the smoke detector identifies a fire.

Let's start looking at how to model that within RoseRT.

Modeling components
RoseRT uses the UML model of active objects to model components in the system. These active objects are known as capsules and are described by class diagrams within RoseRT. Capsules enable you to model the individual elements in the system in isolation. For example, the smoke detector has more to it than a simple on/off switch to indicate a potential fire. Capsules communicate with each other through a series of ports. Ports allow communication through a series of discrete messages, and ports on different capsules are connected together in order to provide communication through the system.

A structure diagram describes the capsule's interface and internal composition and how it communicates with other capsules in the system through its ports. RoseRT uses the structure diagram to generate detailed code to implement the communication of information between capsules. A good example here is the communication that occurs between the smoke detector and the alarm. The smoke detector must send a message to the alarm to tell it to go off when smoke is detected.

The final part of the modeling process is the state diagram. State diagrams describe the behavior of a capsule class. The state machine itself is designed to generate the code to use with your model. Changes in states in each capsule are triggered by an incoming message from a given port. For example, the fire alarm capsule is triggered when it receives a message from the smoke detector.

Creating a new model

To create a new model within RoseRT, decide what the target development language is going to be. Because you are working toward an embedded Java solution for this example, use the RTJava framework.

Within RoseRT:

1. Choose New from the File menu. The Create New Model window opens.

Figure 2. Choosing a framework
2. Double-click the RTJava icon. A blank model window opens, shown in Figure 3. The window is basically split into four sections. The browser on the left allows you to navigate through the entire model. The main panel on the right is the diagram window. Underneath it is the output window on the left and a code/documentation window on the right.

**Figure 3. A blank RoseRT model**
Now you're ready to start building the various capsules and structure diagrams to describe your system.

### Modeling a capsule

The first task is to create the capsules that make up the system. There are five main capsules: three defining the different sensors, the alarm bell and the control panel/keypad. There are also four capsules to define the sequence during different events. These events are what happens during entry, exit, a triggered intruder alarm, and a triggered fire alarm.

To create a single capsule:

1. Within the browser, expand Logical View and then double-click the Main component. This is the location of the main code executed by the system.
2. Click on the capsule tool, and then click on the main diagram window.
3. Rename the created capsule **Alarm**. You'll have a diagram like the one shown in Figure 4.
4. Repeat the process to create additional capsules for the following elements:
   - WindowDetector
   - IntruderAlarm
   - DoorDetector
   - Entry
   - Exit
   - Keypad
   - SmokeDetector
   - FireAlarm

You now have a series of capsules that define the main components in your system. Four of the capsules are used to define specific events, such as what happens during an entry procedure, or when a fire alarm needs to be sounded. The others model the actual sensors and the alarm.

Creating a structure diagram for a capsule
It isn't necessary to look at the entire system of your example to illustrate how to use RoseRT to build an application, but let's look at a the structure diagram for one of the components within the sequence of a typical alarm. A fire alarm is raised when the smoke detector identifies smoke and the detector is enabled, so let's look at the specifics of the smoke detector component.

The smoke detector capsule models a typical smoke detector. Here you have an external interface to a physical device that indicates whether there is or isn't smoke detected, and an internal control that determines the state of the smoke detector. Either the smoke detector has been enabled by the alarm system, and it's switched on, or it's disabled and switched off. If it's switched on and smoke has been detected, then it needs to raise the alarm, and obviously if it's off, it won't.

The structure diagram therefore has three ports: one to accept the external smoke detection trigger, one to accept the enable/disable alarm, and another to send messages to the alarm to indicate a fire.

Each port also has a protocol that you create to define the content of the messages sent between components in the model. For example, the alarm enable/disable protocol called EnableDetector, has two predefined messages: EnableDetector and DisableDetector. These protocols and their messages are used by the state machine to help control the execution of the system. For example, when the SmokeDetector capsule receives the EnableDetector message it switches from the disabled to the enabled state, and then listens for messages on the SmokeDetection port to identify whether there is a fire.

Finally, all structure diagrams ultimately create a suitable class for the capsule, and within this capsule you can create attributes (cat class and instance level) to store information about the objects to be used by the different parts of the application. There is an attribute in SmokeDetector called DetectorState, which can be used to determine the state of the detector: zero is disabled and one is enabled. RoseRT enables you to specify the datatype and initial values of these attributes.

The sequence for the structure diagram is:

1. Create protocols that don't already exist.
2. Configure attributes you want to use.
3. Create necessary ports.

This sequence of developing a model within RoseRT is described in more detail in the next few panels.
Creating protocols

Protocols are merely the message contents sent between capsules in this model. The protocol defines the message contents and determines the direction that a message can be sent between two ports. For example, the Enable message within the EnableDetector protocol can only be sent to a detector.

To create a new protocol:

1. Right-click on the Logical View component of your model.
2. Select **New > Protocol**.
3. A new protocol is created within the browser. Name the protocol **EnableDetector**.
4. Click the Signals tab. Right-click in the In Signals panel and select **Insert**.
5. Name the signal **Enable**. You can give the message a data class, for example to send a text string or other value, but you won't be using that here.
6. Repeat the procedure to add a signal called **Disable**.
7. Use the same protocol for sending and receiving messages on this port. Drag and drop each signal from the In to the Out list so that the protocol can be used for bidirectional messaging. A window like the one below appears.

   **Figure 5. The EnableDetector protocol specification**
Now you've got a protocol you can use. Note that protocols are available to all elements within the system and can be employed by any of the ports in the system. The EnableDetector protocol can be used with any of the sensors in our system. In the full model for the fire alarm system, there are additional protocols for SmokeDetection (from the external sensor) and RaiseAlarm, which tells the alarm bell to ring. The EnableDetector protocol you created in this panel enables and disables the detector.

Adding ports

Ports allow communication between capsules. You must add ports to a structure diagram so it can accept messages from other capsules in your model, and then use this in combination with the state diagram to control execution of the system.

To add a port to your capsule:
1. Open the SmokeDetector capsule in the browser.
2. Double-click on the Structure Diagram.
3. Click the Port button on the toolbar.
4. Place the port on the edge of the diagram. When asked to select the protocol to use on the system, select EnableDetector.
5. Name the port EnableDetection. This helps identify the port in the state diagram.
6. Repeat the process to add other ports to the SmokeDetector capsule, including the RaiseAlarm port and the SmokeDetection port and their respective protocols. Your structure diagram should look like Figure 6.
   **Figure 6. The SmokeDetector structure diagram**

![Image of the SmokeDetector structure diagram]

7. Individual ports have a series of different configurable properties. For your system, use End ports which are bidirectional and allow communication between ports on two capsules in both directions. Click the select tool and double-click on each port to open its properties window and select the End port property from the general tab.

You're now ready to build a state diagram to configure how your SmokeDetector operates and how it responds to different messages on the various ports.
State diagrams

Within RoseRT, the state diagram describes the sequence of events during the execution of an individual capsule. This links the ports, capsules, and messages that link and control the sequence together. The result is a coherent sequence that is translated by RoseRT into a sequence of events within the final code. This is in turn used by the message system and protocols to control the sequence of events.

A state diagram works by mapping out the individual states of a system and the state changes. It's probably easiest to think about states as a sequence of events. Each step in the sequence is a state within the state machine, and each transition from one state to another is what triggers the movement from one step to another.

There are two basic states for SmokeDetector. Either it's enabled or disabled. In the disabled state, messages from the SmokeDetection port can be ignored. In the enabled state, there is a RaiseAlarm state that is triggered when the SmokeDetection port sends a message that smoke (and ergo fire) has been detected. This in turn triggers the delivery of a message to the Alarm to ring the bell.

Finally, to cancel the bell, the Enabled state can return to a DoNothing state if a CancelAlarm message is received on the Alarm port from the FireAlarm system.

Each state model has an initial state, which is the status of the system when your model is first started. Again, in this case, the default state for the SmokeDetector is to be disabled until it's specifically enabled because the alarm has been switched on.

Generate a state model for the SmokeDetector capsule

The SmokeDetector capsule models a system that sends a message to the Alarm capsule when smoke has been detected. To create a state diagram for the SmokeDetector capsule:

1. Double-click on the State Diagram for the SmokeDetector capsule. A blank diagram appears as shown in Figure 7.

   Figure 7. A blank state diagram
2. Click the State tool in the toolbar and click inside the state diagram. This creates a new state, which you'll disable.

3. Select the state transition tool from the toolbar. Click the circle in the top left. This is the initial state trigger and is used to tell the capsule what state it should be in when first created.

4. Click the Disabled state you just created. This automatically creates a state transition called Initial.

5. Select the Transition to self tool from the toolbar. Click the Disabled state. Self state transitions are used to indicate when there should be no change to the state. For example, while the SmokeDetector is disabled, whenever a message is received from the SmokeDetection port it's ignored, whatever its value. Name the transition IgnoreSmoke.

---

**Continuing the state diagram for the smoke detector**

You're half way through building the state diagram for your SmokeDetector capsule. To finish configuring the state diagram and set up the additional transitions:
1. Click the Select tool and then double-click the **IgnoreSmoke** transition.
2. Click the Triggers tab and then right-click and choose **Insert**.
3. Select the port, **SmokeDetection**, to accept a message and then select the asterisk (any) value for the incoming signal. This configures the transition to be triggered whenever a message is received from the SmokeDetection port. In this case, if you are in a disabled state, then you transition to the disabled state whenever a message is received on the SmokeDetection port. In other words, nothing is done.
4. The last transition to the disabled state comes from the EnableDetection port. Click the Transition tool.
5. Click the edge of the state diagram. This indicates that the transition is triggered by an external message.
6. Click again on the **Disabled** state.
7. Name this transition **Disable**.
8. Click the Select tool and double-click the **Disable** transition.
9. Click the Triggers tab and create a new trigger from the EnableDetection port for the Disable message.

You have the Enabled state for the SmokeDetector configured. Next, set up the transition information for the Enabled state.

---

### Setting up the enabled state for the SmokeDetector

You're almost finished setting up the state diagram for the SmokeDetector capsule.

You should now be able to create another state within the top level diagram for the Enabled state. The Enabled state is triggered by an Enable message on the EnableDetection port. Once you create it, you should have a diagram that looks similar to Figure 8.

**Figure 8. The top-level SmokeDetector state diagram**
Individual states within the top-level state diagram also have their own sub-states. For example, you know that when the SmokeDetector capsule is in its Enabled state it needs to respond to messages from the SmokeDetection port and send a suitable message to the alarm to indicate a fire. There are in fact two sub-states: RaiseAlarm that was just described, and the DoNothing state that is active when the system is enabled but no alarm signal is received from the SmokeDetection port. The final state diagram for the Enabled state is shown in Figure 9.

**Figure 9. The enabled state of the SmokeDetector**
Now that your model is created, you can see the power of the RoseRT system. You've built an entire class and sequence of events for the execution of an instance of the class just by drawing a diagram. Behind the scenes, RoseRT converts your model into the class and the necessary methods to help control the capsule during execution.

As described earlier in the tutorial, communication between the various capsules is handled through the message system and the ports. The messages sent (and received) through the ports drive the states of the system, and in turn the transition between states can be used to transmit messages.

Within a real-time system, such a message based approach makes the execution of an application much easier, as it relies on components working on individual triggers, rather than the usual method-calling approach you might use when developing a desktop or Web application.

Models for the Alarm capsule

Now that you've modeled the internal states and structure of the
SmokeDetector, you can do the same for the Alarm capsule. This has only one port, RaiseAlarm, that is used to accept alarm messages in the event of a fire. You can see the structure diagram in Figure 10.

**Figure 10. The structure diagram for the alarm**

![Image of structure diagram for the alarm](image)

The state diagram is also fairly simple, and can be seen in Figure 11. The initial state is AlarmOff, and a transition to this state is also triggered when a CancelAlarm message is received on the RaiseAlarm port. An AlarmOn state is triggered when the RaiseAlarm message is received on the RaiseAlarm port. Simple!

**Figure 11. The state diagram for the alarm**
The final part of the system is the FireAlarm capsule. Details are not provided in this tutorial, but its main role is to aggregate the steps and state for the entire FireAlarm sequence. It provides controls, using messages received from the Keypad capsule created earlier in the tutorial, to enable and disable the alarm system and to cancel the alarm in the event that it goes off.

There are also other capsule systems in the overall alarm system. The WindowDetector, for example, works pretty much like the SmokeDetector does; working the BurglarAlarm capsule and the Alarm capsule to ring the bell in the event of an intruder. The DoorDetector works with a state machine that enables and disables the alarm after a wait period, using one of the built-in timer systems provided by the RoseRT patterns.

If you were to continue modeling the full system, all of these different capsules and states could be wrapped up together into a single AlarmSystem capsule. This capsule would have specific states for operating in the Alarm enable state, DoorEntry and DoorExit states (which hopefully turn the alarm off and on respectively) and a simpler AlarmOff state, where nothing happens.

Now let's look at what happens if you want to adjust your model.
Changing the model

In most alarm systems there is more than one sensor, and in this example you have smoke, window, and door sensors. But what if you want to add another sensor to this model?

If you have cats, for example, you're aware that sometimes it would be nice to know when they let themselves in and out of the house through the cat door. You can do this with your alarm system by introducing an additional sensor and alarm into the system that is triggered each time a cat walks through the cat door. Obviously the physical mechanics of this system are beyond the scope of this tutorial, but the technical mechanics of the software that support it are quite simple.

All you have to do is create a new capsule, and a new class, which makes use of the external input, and then generates a different message value during the RaiseMessage protocol to indicate that a cat has gone in or out.

The structure diagram and state machine are pretty much the same as the SmokeDetector capsule, although you're taking input from the physical cat door.

You also need a wrapper around the system, a CatAlarm Capsule, to enable and disable the sensor and cancel the alarm.

The important thing to remember here is that you can use the same protocols to form the communication between the different components. The ultimate communication can be through the existing Alarm system. After all, the RaiseAlarm protocol supports the raising of different alarms based on the value of the message that you send.

Unlike a traditional system, you don't have to worry about adding a new check sequence, or writing any complex additional code into the system to add the new sensor. RoseRT takes care of the code generation, the message parsing, and all the concurrency that you need to support the system.

Ultimately, very little happens when you add a new capsule to the system. You don't have to re-write anything or even change the definition of our existing model, you just extend it and let RoseRT handle the complexities.
Section 4. Add code and build the model

Adding application specific code

In the sample system, there isn't a lot of code beyond that which has been generated by the RoseRT system. Although this simplifies the execution of the model from an architectural and modeling point of view, it doesn't actually turn the model into a full blown application.

To add code to the system, you need to add code fragments to the individual components -- states, transitions, ports and protocols -- that change the default (and usually simplistic) behavior of the model. For example, in the Alarm capsule, you'll need to add the code that tells the device that the application has been deployed to ring the bell, or play the tune.

Custom code is triggered by different events in the model. For example, states within a model have EntryActions and ExitActions, which are code blocks that will be executed when the capsule enters and leaves the state respectively. To refer to the previous example, the AlarmOn state in the Alarm capsule would have the custom code to ring the bell in the host device when the state was entered. It would probably have a similar switch off command to the device as an ExitAction.

Code can be added in two ways: the corresponding tab within the properties for the object, or the Code panel within the main browser (bottom right) to add the code without having to open the component.

Creating component specifications

The two stages to generate code with RoseRT are the deployment specification and the target specification. The component specification defines how to generate code and build individual components within the overall model. Either the entire model can be built to the same specification, or you can use different compilation options for different components. RoseRT gives you the flexibility to build and test the different components you need.

You can also use this to build different parts of your model based on how much of your model is complete, and which part of your model you want to test. During the early stages, you probably want to test only individual fragments of your model (the FireAlarm in this example), but as you finalize each section you'll start to test more of the overall system.
To create a component specification:

1. In the model browser, right-click Component View and choose **New > Component**.
2. Name the new component specification **FireAlarm**.
3. Double-click the component and check the configuration, as shown in Figure 12. Given how you are generating code for an embedded Java application, make sure the RTJava and RTJava Project options are selected for Environment and Type respectively.
   
   **Figure 12. Component specification**

4. Click the References tab and then drag the FireAlarm capsule from the browser on to the References panel. This tells RoseRT that this is the capsule to be used when generating code.
5. The remaining tabs enable you to finely control the tools, files and other components that are used when building the system. For example, if you were using the CLDC and MIDP extensions for J2ME, ensure that the necessary files and packages are referenced in this section.

Your model is now ready to be translated into code and built into an application.
The deployment specification is used to build and deploy an application for your target platform. Details are provided later. First, let's take a brief look at the build process.

---

### Building the application

Once you have created a component specification, build the component by right-clicking the component and choosing **Build**. This generates the necessary code and then builds the application, providing you have the correct tools and environment available to do this.

You can monitor the build process with the Build log in the bottom left of the window, and examine any errors generated during the process by looking at the Build Errors tab. Note that you might have to accept additional references for a component build, as the other capsules, classes and protocols are included in the references for the component view.

The process should be relatively easy. Remember that it is RoseRT generating all of the classes, the message passing system, and all the other code required to actually implement your model.

To view the raw source code that has been generated by the system, open the folder where you saved your model. There is a directory called src that contains all of the raw Java code that was generated.

---

### Building for a specific target

When it comes to deploying an application on a target device, you need to tell RoseRT about the computing hardware and processor that the application will be deployed on. To do this, first define the processor and then the underlying hardware through a deployment diagram.

To create a deployment for the development machine (so that you can run and test your model):

1. In the model browser, right-click the Deployment View and select **New > Processor**. Rename the processor `LocalHost` and press enter.
2. Drag the FireAlarm component from the component view onto the newly created Deployment View. This creates a new instance within the browser that can be used to run the model on the development machine.
You can create additional deployment views that build the application for other platforms, and even allows you to directly execute the application on the target platform.

The final stage of any modeling process is to execute the model and monitor its execution to see if the model works. This stage is covered briefly the final section.
Section 5. Test and alterations

Direct testing

Once you have successfully built and deployed your model into an application, execute your model and monitor its execution. Let's have a look at some of the key features involved. The system is intelligent enough to monitor the execution of different elements and link these elements back to the model that you are executing. In essence, it's identical to any symbolic debugger, but unlike a code-level symbolic debugger, it attaches to the model, not the code.

To run a model, right-click the FireAlarmInstance you created in the last panel and choose Run. You might need to re-build the instance if you've made any changes to your model, and you'll be prompted if this is the case.

Once your model is running, use the Step tool to step through the model. Each click of the Step button allows one message to be transferred and the current transition (if there is one) to complete.

Monitor execution and make alterations

To monitor the progress of your model, add state monitors that report the current state of a capsule. Right-click the Capsule instance on the Runtime View tab and choose Open State Monitor. In each case you get the graphical state diagram for a component, with the hard-line box surrounding the current state within each capsule state diagram.

You can also add watches to monitor attributes in classes, which is useful when monitoring the execution of the system. The final debug system is a capsule instance trace, which provides a blow-by-blow account of the state changes, and messages transferred between capsule components.

All of these systems together enable you to monitor the progress and execution and identify any potential problems. Because you are working directly with a model, you can spot exactly where the problem is, adjust the model, and then try again. This saves having to hunt through the code for the problem. You can see, visually, the precise point of the problem and fix it just by adjusting the diagram. Typical issues are capsules not changing to the correct state, or messages not being received and therefore not triggering an appropriate transition.

Once you've monitored the model while it's working, you will most likely want to
make changes to the sequence of events and the execution model that you have created. Like other parts of the RoseRT system, this is where a modeling system really helps you out. Rather than laboriously changing the code, modify the model to achieve the result you want, then regenerate the code and test again. This makes the entire process of building an application, generating the code, and debugging the system significantly easier.

Integration with other Rational tools

Like the rest of the Rational system, RoseRT is integrated with other tools in the Rational range:

- Rational ClearCase helps you manage code and track different versions and revisions. The integration is automatic and is related and linked to the individual build and code generation steps within RoseRT.
- RequisitePro can be used to build requirements which can be linked directly into the capsules and classes within RoseRT. RoseRT can also be driven through Use-Case statements, which can be taken from RequisitePro.
- Rose RealTime Purify integrates with RoseRT during the build and execute steps, producing build and runtime analysis of your model automatically.
Section 6. Wrap up

Summary

In this tutorial you've seen the basic steps of modeling a system within RoseRT. The basic structure was an entire alarm system, although you actually only modeled the main fire alarm component of the system. Using the principles in this tutorial, however, you can easily model the rest of the alarm system.

Through the process, you learned how to specify an overview model, and then create and further define the individual components of the system. Through the tutorial, you've seen that RoseRT works on the basis of messages passed between capsules (active objects). Each capsule has a number of states, and messages passed between the capsules can be used to force a transition from one state to another. Within a real-time/embedded environment, the message-based system and the state model allow individual capsules to work independently, and also allows events to occur simultaneously and instantly, just as you would expect from a real-time application.

Resources

For more information on Rational Rose RealTime and using it for modeling and developing applications:

- For product information on Rational RoseRT, see the Rational Rose Developer page.
- Download a trial version of Rational RoseRT.
- You can download Java development tools from Sun's Java home page (http://java.sun.com).
- Obtain a version of Microsoft's nmake, which you'll need if you don't already have access to Visual Studio and want to build Java applications with RoseRT.

Feedback

Colophon
This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/.