Unified Modeling Language for Real-Time Systems Design

Introduction

The Unified Modeling Language, or UML, is a third-generation object-oriented modeling language. It adapts and extends the published works of Grady Booch, Jim Rumbaugh, and Ivar Jacobson [Booch94, OMT91, OOSE92] and contains improvements and suggestions made by dozens of others. The UML is being presented to the Object Management Group in the hope that it will become a standard modeling language for object-oriented development. Because the UML is meant to be applicable to the modeling of all types of systems, it applies equally well to real-time systems, client/server, and other kinds of "standard" software applications. It provides a rich set of notations and promises to be supported by all major CASE tool vendors.

The purpose of this white paper is to discuss some of the highlights of the UML, particularly as they apply to the design of real-time systems. This work is based on the latest drafts of the UML documentation available at the time of writing [UML0.8, UML0.91]. Because of the depth of these documents, this paper addresses only the most fundamental elements of the UML but omits many details. For the latest and most complete information, please see the object technology section of Rational Software Corporation's Web site (*http://www.rational.com/ot/uml.html.*

An Object-Oriented Approach to Modeling Systems

Structured methods clearly separate data from functions, decreasing their cohesion. Object-oriented (OO) methods take a different approach. They unify data and the functions that operate on them into software components called *objects*. In the real-time world, objects are models of things such as sensors, motors, and communication interfaces. The following table provides an informal object-oriented description of these kinds of objects:

Object Type	Data	Functions
Temperature Sensor	Temperature	Acquire( )
	Calibration Constant	Set Calibration( )
Stepper Motor	Position	Step Forward( )
		Step Backward( )
		Park( )
		Power( )
RS232 Interface	Data to be Transmitted	Send Message( )
	Data Received	Receive Message( )

Object Type	Data	Functions
	Baud Rate	Set Comm Parameters( )
	Parity	Pause( )
	Stop Bits	Start( )
	Start Bits	Get Error( )
	Last Error	Clear Error ( )

Because some of the objects in a system can be replicates of one another, it would be redundant and tedious to specify the data and functions of each. OO methods use the notion of a *class* to capture, in one place, the *structure* (for example, the common data fields) and the *behavior* (for example, common functions callable) of such objects.

A class is like a C-language *struct* declaration that contains both data and function (pointer) fields. Structs are definitions of groups of fields that are closely related. All variables of the same struct type look alike in that they all have the same fields. Correspondingly, every object has a definition that is prescribed by its class. But unlike C structs, which generally define only data fields, an object's class defines both its data and its functions in one cohesive structure. Using the UML terminology, the data portion of an object is defined by a set of *attributes,* and the functional portion of an object is defined by a set of *operations*.

While every object of a given class has exactly the same structure and behavior, each object is unique in that changes to one do not automatically affect others. For example, every object of the *Temperature Sensor* class has its own copy of the data items *Temperature* and *Calibration Constant*, as well as its own access to the operations *Acquire*( ) and *Set Calibration*( ).

Structuring data and functions into classes is fundamental to modeling with an OO approach. Another key principle of object orientation is *encapsulation.* This principle states that the data portion of an object is accessible only through the functions defined by the object's class. Encapsulation makes objects more reliable and safe in that:

- Users of an object cannot directly affect the state of the object, but rather must go through a more controlled interface, such as a function call.
- Developers of an object's class can make changes to the data portion often with little or no impact on the users.

Learning to appreciate and efficiently use encapsulation is often the most difficult part of making the "paradigm shift" that is required when going from a structured method background to an object-oriented method.

Modeling with UML Diagrams

While tables like those in the previous section provide a convenient way to informally capture class definitions, the UML goes much further in the kinds of information that can be modeled. The easiest way to describe the various modeling aspects of the UML is through the notation defined for its various types of diagrams.

The UML distinguishes between the notions of model and diagram. A *model* contains all of the underlying elements of information about a system under consideration and does so independently of how those elements are visually presented. A *diagram* is a particular visualization of certain kinds elements from a model and generally exposes only a subset of those elements' detailed information. A given model element might exist on multiple diagrams, but there is but one definition of that element in the underlying model.

This paper introduces the notation and semantics for the following kinds of diagrams supported in the UML:

- Class diagram
- Use-case diagram
- Interaction diagram
  - Sequence diagram
  - Collaboration diagram
- State diagram
- Component diagram
- Deployment diagram

Each type of diagram captures a different perspective, or view, of the system's underlying model. For any of the diagram types, multiple diagrams of that type may exist.

Note that not all of these diagram types have features specifically intended for the modeling of real-time systems. In particular, interaction, state, and deployment diagrams have the most to offer in the real-time arena. But for completeness, the following sections describe the notation and semantics for each of these types of diagrams.

Class Diagram

As with other object-oriented methods, the class diagram is core to a UML model. A class diagram shows the important abstractions in a system and how they relate to each other. The primary elements found on class diagrams are class icons and relationship icons.

Classes, Attributes, and Operations

Individual classes are represented in the UML as solid, outline rectangles with one, two, or three compartments. Figure 1 shows how the classes described in the previous tables can be represented using the UML class icons. The first compartment is for the name of the class and is required. The second and third compartments are optional and may be used to list the attributes and operations defined by the class.

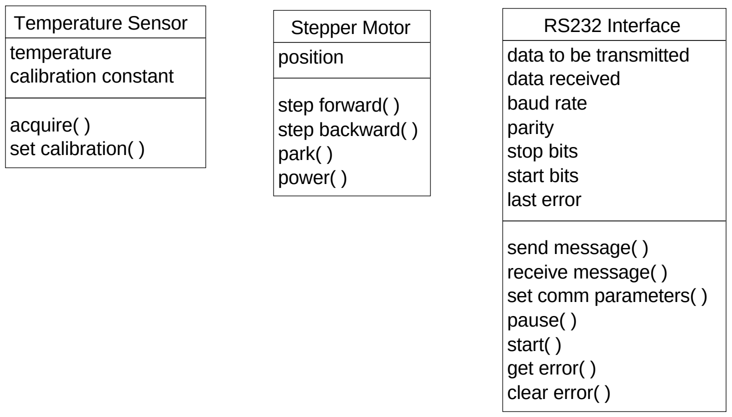

Figure 1: Class icons with attribute and operation compartments

While displaying such details for a few specific classes can be useful, showing all class members for all classes in a model can quickly clutter a class diagram. Such complexity defeats the purpose for using class diagrams, which is to provide some degree of abstraction above the vast underlying details found in any nontrivial model. This is why the UML allows a class icon to suppress displaying any or all of its members. This principle is key to the UML: Diagrams do not generally expose the total contents of a model, but rather only some subset of the model's details.

Thus, all of the icons shown in Figure 2 are valid representations for the class Temperature Sensor—they are merely different views of the underlying model.

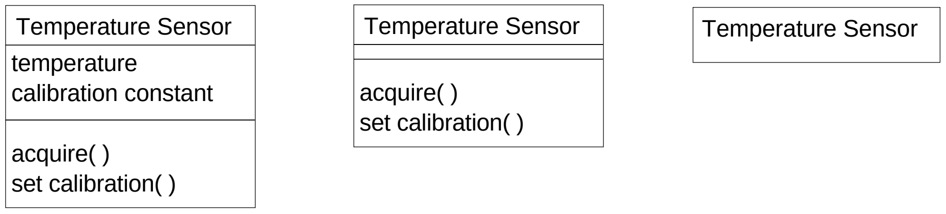

Figure 2: Multiple views of the same underlying class model

Relationships

Stand-alone classes provide only so much value in and of themselves. Most classes in a system will be related to other classes so that their corresponding objects can collaborate to accomplish more complex functionality. So in addition to classes, attributes, and operations, class diagrams also depict various types of *relationships* that exist between dependent classes.

The UML distinguishes between several different kinds of relationships, each of which has its own set of special adornments and associated meanings. Figure 3 illustrates a few of the most basic types and forms of relationships.

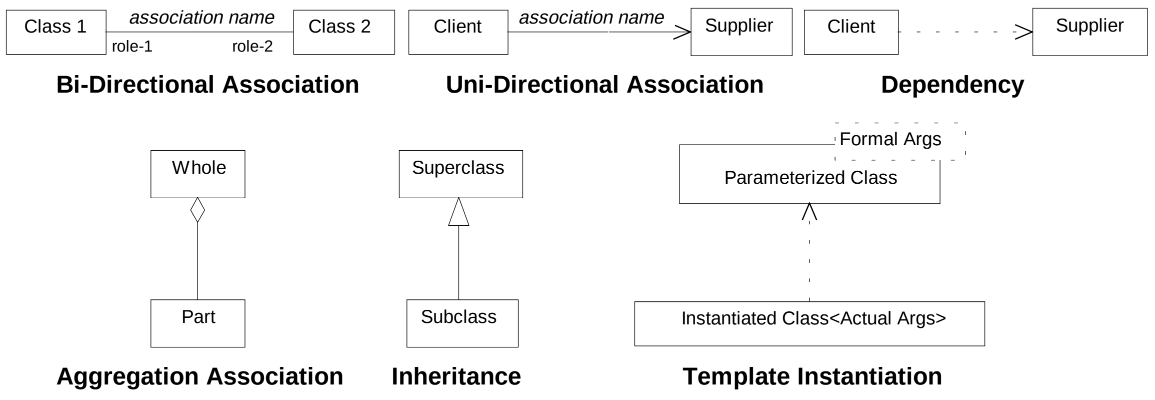

Figure 3: Types of UML relationships

As an example of how some of these relationships might be used, consider a simple system in which a controller uses a sensor with an analog-to-digital converter to acquire information. This same controller then uses an actuator to affect its environment. The system may use a temperature sensor to activate either a heater or a fan for closed-loop control. Additionally, it may use a pressure sensor to provide information about a gas line that it uses to control a valve with a stepper motor. A class diagram for such a system is shown below:

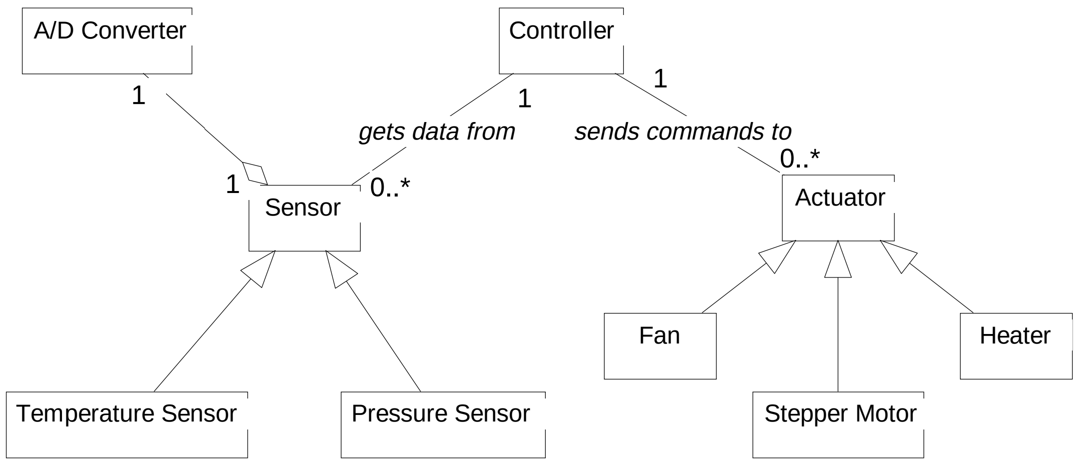

Figure 4: Simple sensor-actuator-controller system

The boxes visually represent the key abstractions of this system as classes. The lines connecting these classes represent various types of relationships between these abstractions. From these classes, software objects will be created to carry out the responsibilities of the system. The following subsections describe the UML notation used for relationships in more detail.

Associations

An association is used to represent a structural dependency between objects, generally of different classes. For example, the line between the Controller and the Sensor in Figure 4 means that there exists some connection between objects of these classes. The textual annotation on the line clarifies the situation. The label “gets data from” indicates that an instance of the Controller class acquires information from possibly a number of instances of the Sensor class. Similarly, an instance of the Controller class “sends commands to” possibly a number of instances of the Actuator class.

To specify how many instances are to participate in an association, the UML defines adornments for *multiplicity*. If you look closely at Figure 4, you will see small numbers and/or asterisks next to the endpoints of the associations. These indicators specify the potential numbers of objects participating in the relationship. The association between the Controller and Sensor classes states that a given instance of the Controller class can be linked with “zero-to-many” Sensor objects, as indicated by the 0..* symbols next to the Sensor class. Similarly, a given instance of the Sensor class can be linked with precisely one Controller object. Multiplicity can be indicated by either a constant (when known) or with a ‘*’ to indicate “many.” An unadorned relationship end is assumed to be “unspecified” in its multiplicity; no default value is assumed in the UML.

Associations are bidirectional by default. This means that an instance of one class can *navigate* to instances of the other class, and vice versa. Navigability is often realized by objects maintaining references of some kind between associated objects. When a association is left in its bidirectional form, there will exist a circular dependency between the corresponding objects, resulting in a *peer-to-peer* relationship between those objects. As a design decision, one may “turn off navigability” in one direction of the association so as to simplify the implementation and establish a more *client-supplier* relationship. In a unidirectional association, the UML introduces an arrowhead at the supplier end of the line segment.

Although bidirectional associations are useful for analysis modeling, they can be expensive and unnecessarily complex to implement directly. During design, it is common to establish a client-supplier relationship so that the client knows about the server, but not vice versa. For example, a Sensor object might act as a server providing monitored data to a number of clients. The Sensor has no idea which objects may ask for the information, but the client objects know how to ask for the monitored value from the Sensor.

Aggregation

An *aggregation* is a special form of an association that is used to show that one kind of object is composed, at least in part, of another. For example, a Sensor may be composed of a number of components, including an A/D converter as is shown on Figure 4. The line between the Sensor and the A/D Converter classes has a small diamond at the Sensor end, suggesting that instances of the A/D Converter class are “part-of” some instance of the Sensor class. The Sensor is thus designated to be the aggregate—that is, the “Whole” in the Whole-Part relationship.

Aggregation indicates that the *lifetime* of the parts are dependent on the lifetime of the whole. This means that part-side objects cannot be created unless and until their associated aggregate-side object is created. Similarly, part objects cannot be destroyed by any object other than the aggregate object that created them in the first place. In the example, a Sensor object contains an A/D Converter object, and thus controls the lifetime of its A/D Converter. This is not true for objects emanating from a regular association relationship. A Controller object has a lifetime independent from that of a Sensor object, and vice versa.

Aggregation can be further refined in the UML to denote how the aggregate’s containment of its parts is implemented. The default is *by reference*, which means that the Whole object maintains a pointer or a reference to its parts. The hollow diamond symbol used earlier indicates by-reference implementation. But when aggregation is *by value*, the Whole object declares an actual instance of the Part object within the body of the Whole object itself, thus making the part object physically contained by the whole. By-value containment is indicated by filling in the diamond symbol on the aggregate side of the relationship (see Figure 5).

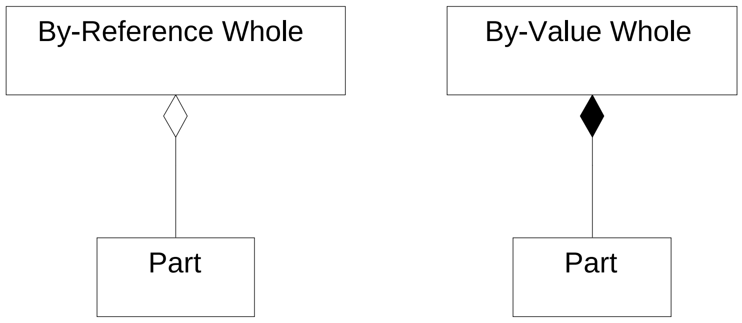

Figure 5: Types of aggregation containment

By-value aggregation has the same semantics as an attribute. While the form used is largely subjective, the aggregation form may be desired when the contained object has complex structure itself.

Inheritance

An *inheritance* relationship is used when one class is to share the structure and behavior defined by another; that is, when one class is a specialization or extension of the other. Referring again to Figure 4, a Temperature Sensor is a more specialized type of Sensor. In the UML, the more generalized class is called the *superclass* and the more specialized class is called the *subclass*. A subclass inherits all the attributes and operations specified in its superclass, as well as any relationship dependencies that the superclass might have against other classes. A subclass may then specialize the implementation of inherited operations, or extend the superclass' structure and behavior by adding brand new data and operations.

Focusing just on the domain of motors, a basic Motor class might have operations, Power( ) and Speed( ). A DC Motor will specialize the base Motor behavior by applying voltage to control the motor speed. A Stepper Motor can specialize the behavior of a DC Motor by adjusting the frequency of stepping. A Stepper Motor can additionally extend the behavior of its superclass by adding a Zero( ) operation. These distinctions can be made by creating a separate class for each particular "kind-of" motor and connecting them with an inheritance relationship. When inheritance is used in this way, it is referred to as *specialization.*

Inheritance may work in the other direction as well. Referring back to Figure 4, one might first identify abstractions for Fan, Stepper Motor, and Heater, and then later realize that these kinds of objects share some common attributes, operations, and relationships. In this case, a more general class called Actuator was created so that these common features could be maintained in one place. When inheritance is used in this way, it is referred to as *generalization.*

Inheritance is probably one of the most intriguing aspects of object modeling. It provides a means for constructing highly reusable components. But one should use inheritance very carefully, for it is often used when aggregation or another type of relationship might be more appropriate. As a general guideline, whenever your domain experiences suggest that one type of object is "kind-of" like another, inheritance is probably a good candidate for the relationship choice. But if it is more intuitive to use the expression "part-of," "contains," or "has," then aggregation is most likely the better choice. And if one object is neither "kind-of" nor "part-of" another, then an appropriately labeled association will often be your best choice.

Dependency

Associations and inheritance relationships generally affect the structure of objects generated from their related classes. But sometimes a class is related to another simply because of the services that are being used. In the UML, a *dependency* relationship means that a client class depends on some service(s) of a supplier class, but does not have an internal structural dependency against the supplier. The most common form of general dependency is found when an operation defined in a client class takes on an argument that is of some other class type.

Instantiates

An *instantiates* relationship is a special type of dependency that exists between a *parameterized class* and the class that is created as a result of instantiation. Parameterized classes are *templates* for regular classes because they are set up to function independently of the type of information with which actual classes will work. *Container classes* are most often implemented as parameterized classes so that they can be written without regard of the type of items that are being contained.

The UML denotes a parameterized class by having a smaller dashed box contain the name of the *formal parameter(s)* to the parameterized class (see Figure 6). These formal parameters are usually generic class or type names. The result of an instantiation relationship is an *instantiated class* that is named with the parameterized class as a prefix, followed by an angle-bracketed name that represents the *actual parameter* of the instantiation (usually some other plain class name). The instantiates relationship often provides an alternative to inheritance for constructing reusable components.

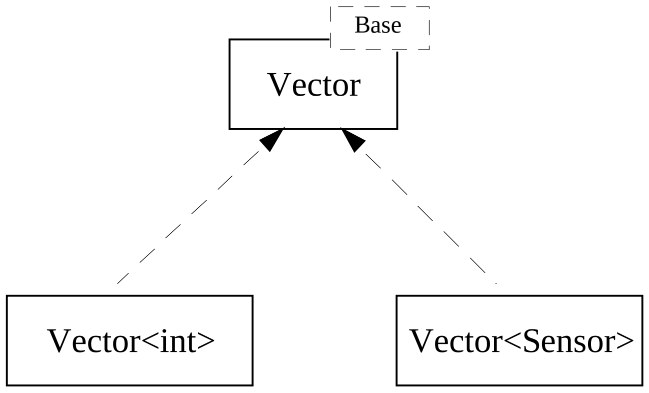

Figure 6: Example of a parameterized class and two instantiations

Use-Case Diagram

Uses cases are broad-stroke descriptions of how a system will be used. They provide a natural high-level view of the intended functionality of the system that is understandable by engineers and nonengineers alike. Use cases, therefore, are invaluable for talking with customers and marketing executives who must specify the system to be implemented. An example use case-diagram is shown in Figure 7.

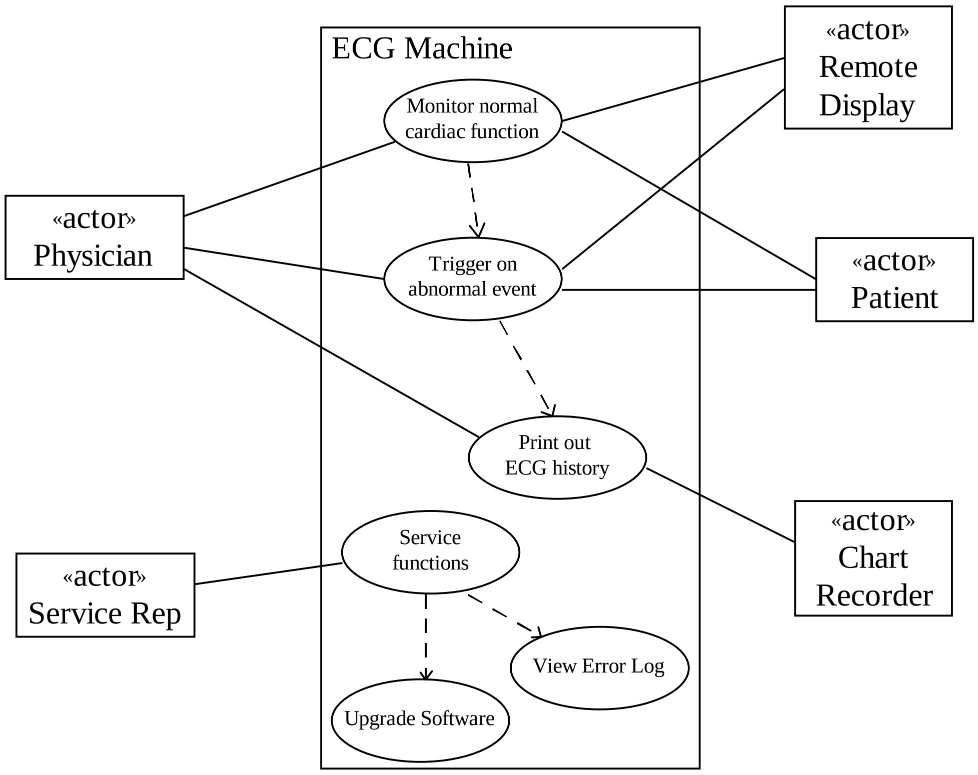

Figure 7: Use case example

The large rectangle shows the boundaries of the system. The rectangles arranged around the system are external entities that interface with the system. These external entities are modeled as classes, but annotated with a special property called a *stereotype* (discussed later) designating them as *actors*. An actor will generally initiate a use case but sometimes may be the recipient of the system usage.

The ellipses inside the system rectangle indicate the use cases themselves. In Figure 7, we see six primary types of system uses, each represented by a single use case ellipse. Some use cases will depend on others, as denoted by the dependency relationships between use case ellipses. For example, one use case is to print out reports containing the ECG case history. The "Trigger on abnormal event" use case will actually use the facilities of the print use case if it automatically prints out a chart recording during a cardiac event.

Interaction Diagrams

Class and use-case diagrams are very static in nature. That is, they are useful at capturing the structural nature of a system design in a very generalized way. Hence, these diagrams are not very useful for specifying real-time requirements or design constraints. *Interaction diagrams*, however, do have applicability to timing and sequencing requirements. This section will introduce interaction diagrams and will point out those features in the UML that have particular applicability to real-time system design.

Scenarios

A use case is a general pattern or strategy of system use. Consider the use case "Monitor normal cardiac function." This broad statement might include the physician setting up the ECG monitor to display four waveforms in one case, or six in another. He may set a bradycardia alarm to annunciate if the heart rate falls below 50 beats per minute in one situation, or 40, or 34. The waveform sweep speeds may be set to 12.5, 25, or 50 mm/sec. In one situation, the patient may not have any abnormal cardiac events; in another, he may have tachycardia or asystole.

The point is that a use case represents many different possible threads of specific interaction. Each specific thread through a use case is called a *scenario*. In other words, a scenario is a specific instance of a use case.

An example of a scenario from the "Monitor normal cardiac function" use case would be:

- The physician powers up the ECG machine with 12 leads in place.
- The physician sets up for four waveforms at 25 mm/sec sweep speed.
- The physician sets the bradycardia alarm at 40 bpm and the tachycardia alarm at 110 bpm.
- The patient undergoes an asystole event.
- The system detects the asystole and raises the bradycardia alarm.
- The physician provides therapy to correct the problem (external to the system boundary).
- The system detects the restarted heart rate to be 45 bpm and lowers the alarm.

A scenario is a particular path through the system functionality, but a single use case represents many related yet distinctly different scenarios.

In system behavioral modeling, it is common to depict dozens of scenarios for each use case. In an attempt to provide some graphical abstraction on top of all the details, the UML provides two notations for modeling scenarios: the *sequence diagram* and the *collaboration diagram*. Objects are shown on both types of diagrams using rectangles just as for classes. To differentiate between objects and classes, the name in an object rectangle is <u>underlined</u>. The class of the object may optionally follow the object name and a colon (':').

Sequence Diagram

Sequence diagrams use object icons with vertical dashed lines projected downward on the diagram. The horizontal directed lines represent the messages passed between objects. Time flows from the top of the page downward. Unless specifically annotated, only the sequence of messages is shown, not the exact time.

The above scenario may be modeled by using a sequence diagram as shown below in Figure 8:

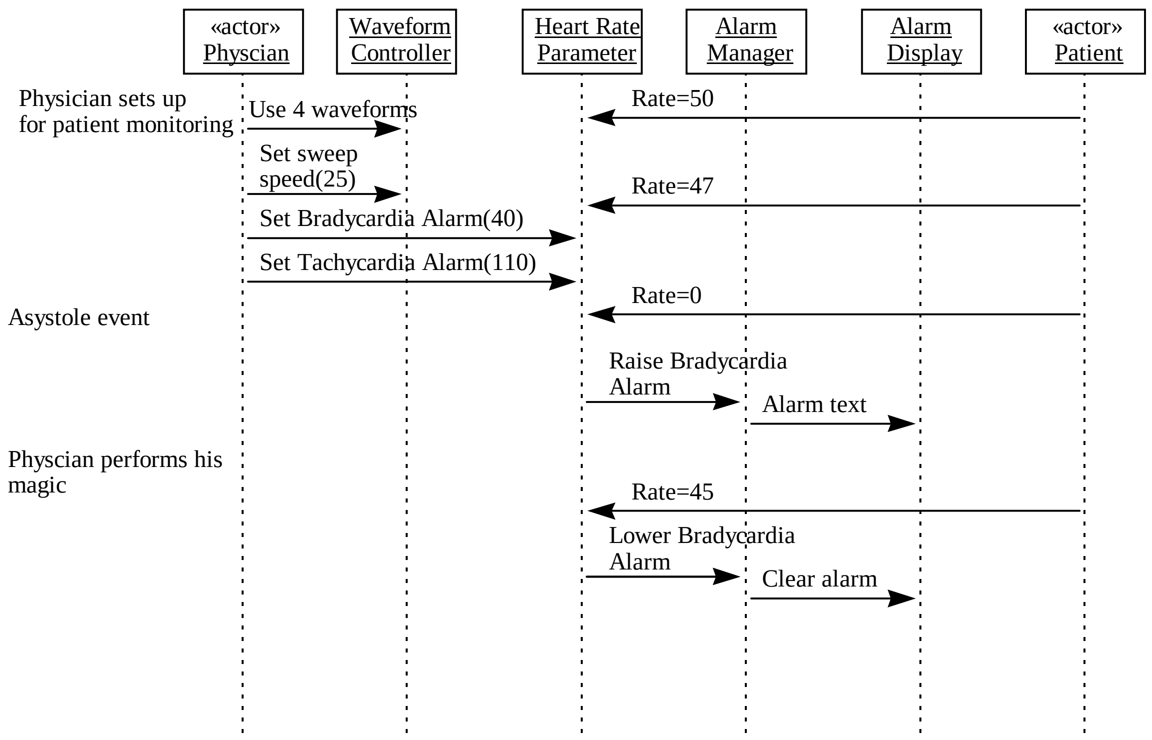

Figure 8: Example of a sequence diagram

The textual annotations along the left edge of the diagram are optional and are referred to as a *script*. Each statement in the script helps explain one or more messages being passed in the diagram. A script may directly correspond to the actual scenario that is being modeled by the sequence diagram.

For real-time designs, exact timing often must be specified. The UML allows textual annotations to be added to sequence diagrams when timing is important. Figure 9 shows a scenario for the real-time display of the waveforms. You can see that two different notations are used to specify timing. The first uses short horizontal lines with a time indication between them. The second labels the messages and specifies a timing expression between curly braces using these message labels.

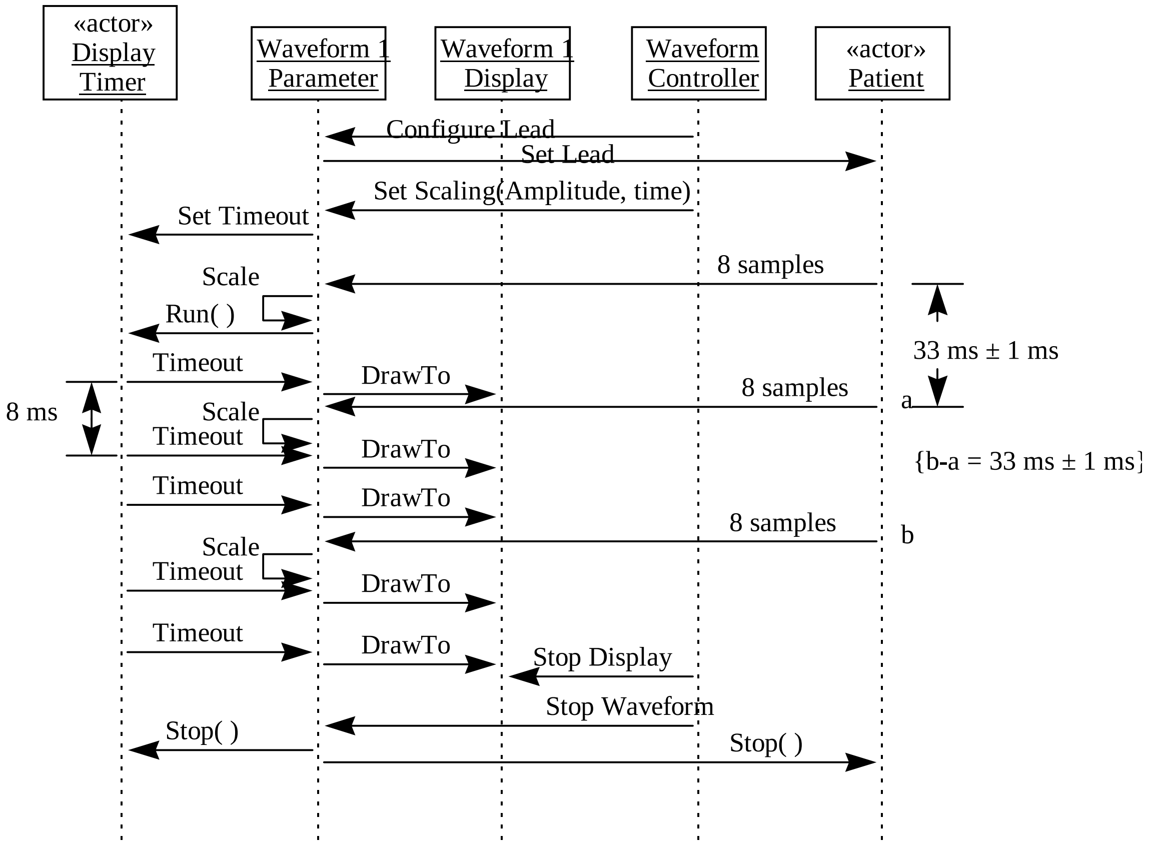

Figure 9: Sequence diagram with timing marks

Collaboration Diagram

The other notation for modeling scenarios is the collaboration diagram. Figure 10 models the same scenario as was done in Figure 8, but in this case as a collaboration diagram. Because there is no “top” to a collaboration diagram, sequence numbers must be attached to the messages to indicate the relative order in which the messages are sent in the scenario. The arrows show the message direction.

You can see that sequence progression is more prominent in the sequence diagram, but structure is more obvious in the collaboration diagram. Timing annotations can be added to collaboration diagrams as well.

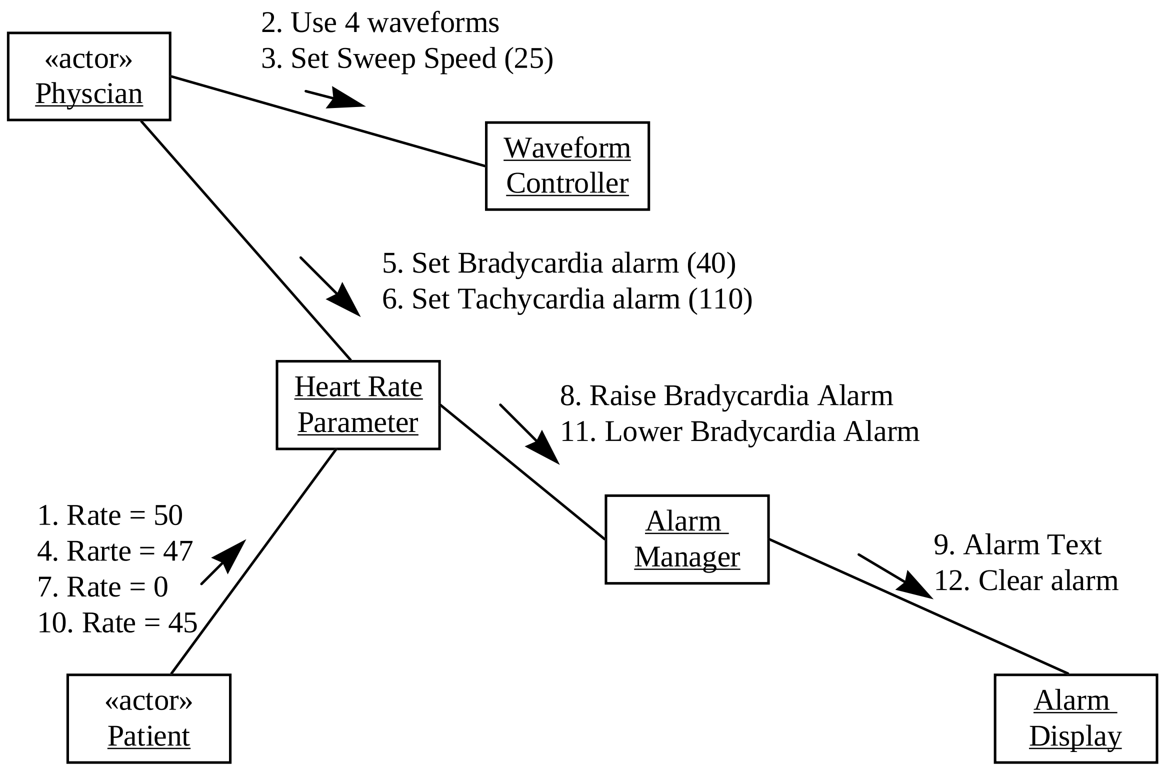

Figure 10: ECG scenario using collaboration diagram

Message Synchronization

Real-time systems often concern themselves with the synchronization of concurrent processes during message passing. The UML provides icons that can be added to any message to indicate its concurrent behavior. The symbols are taken from Booch's earlier work [Booch94].

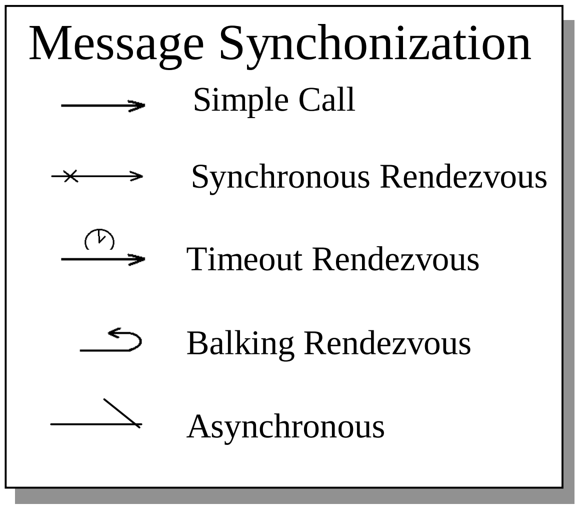

Figure 11: Message synchronization icons

These symbols can be used in conjunction with messages to indicate how the concurrent processes are synchronized during the message transfer.

- *Simple Call*
  Simple messages denote that the synchronization either has not yet been specified or is a sequential message (for example, function call semantics).
- *Synchronous Rendezvous*
  A synchronous rendezvous means that the sender will wait indefinitely for the receiver to accept the message before continuing on with its processing.
- *Timeout Rendezvous*
  A timeout rendezvous indicates that the sender will wait for the receiver to be ready for the message up to some fixed period of time before aborting the message transmission process and continuing on with its processing.
- *Balking Rendezvous*
  A balking rendezvous means that if the receiver of the message is not immediately ready to accept the message, the sender aborts the message and continues.
- *Asynchronous*
  An asynchronous message means that the sender sends the message immediately and continues on with processing without waiting for the receiver to acknowledge its readiness for receiving the message.

State Diagram

The UML state models are based on finite state machines using an extended Harel state chart notation. Harel state charts are more powerful than the more common Mealy-Moore diagrams because state charts support:

- Guards on transitions
- Propagated transitions
- Actions on transitions
- Actions on state entry
- Activities occurring as long as a state is active
- Actions on state exit
- Nesting of states
- Concurrency

Figure 12 shows a simple state diagram for a system with two concurrent processes: the power subsystem and the application subsystem. Note that these processes are independent: the application doesn’t care if it is receiving power from the battery or from the wall outlet. Similarly, the power subsystem is independent of what the application itself is doing. The dashed line separates the two state machines, providing a clear indication that the states are both independent and concurrent.[1]

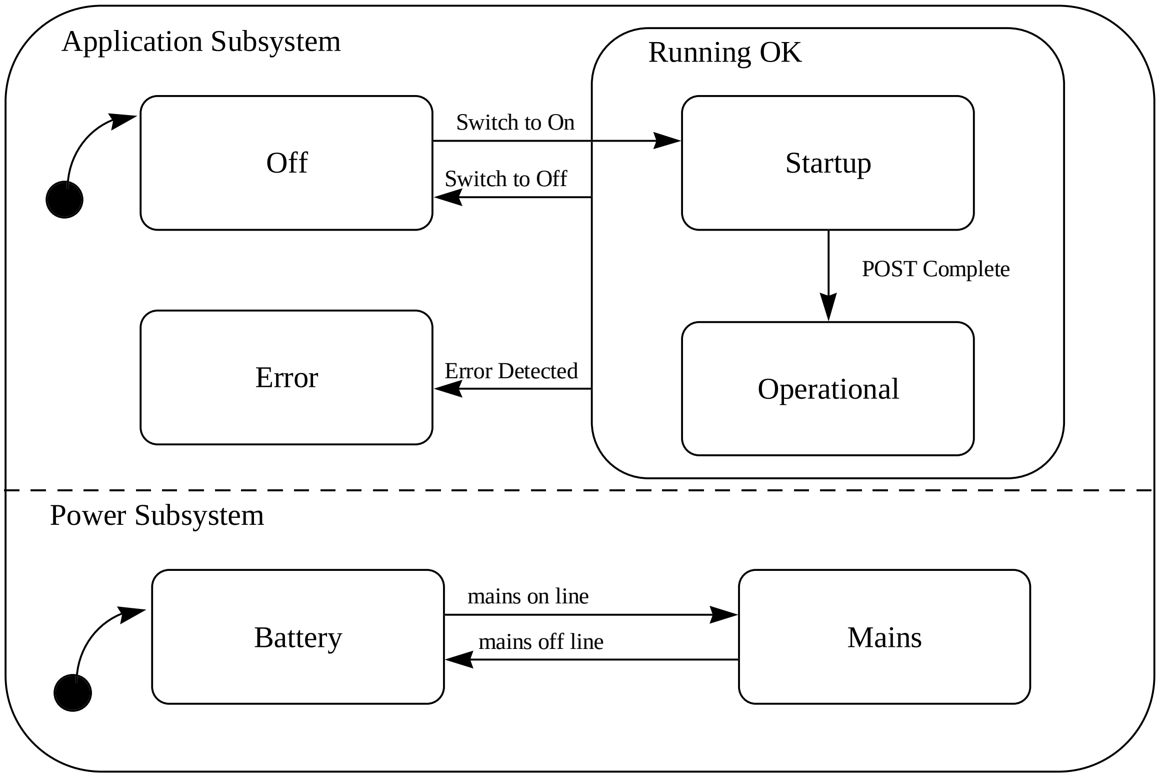

Figure 12: Simple concurrent state machine

Within the application subsystem itself, we see several states. The Off and Error states are clear enough, but both Startup and Operational states are nested within another state called Running OK. What does this mean?

Startup and Operational are *substates* of the *superstate*, Running OK. When the application subsystem is in the state of Running OK, it must also be in one of these substates. Nesting states in this way allows states to be hierarchically decomposed, allowing the developer to break down complex state machines into simpler structures.

One of the benefits of nesting states is that it removes duplicated transitions. Note the Error Detected transition from the Running OK to Error state. Because Startup and Operational are substates of Running OK, this transition applies to both substates. Flat Mealy-Moore state models would require two transitions here, one from each of the substates. In fact, a full Mealy-Moore model of this simple system requires a total of 22 transitions!

[1] Note that this diagram shows the concurrent states from two different classes on the same diagram. Although this is fine, and illustrates a valid point in this context, it is more common to show only the states of a single class on one diagram. Combining the state machine on a single diagram is especially useful when the two state machines interact nontrivially.

Transitions are more elaborate in the UML than even in the Harel notation. The UML syntax for transitions is:

event (arguments)[condition] ^target .sendEvent(arguments)/ operation (arguments)

Each of these fields is optional—even the name may be omitted when it is clear when the transition will be taken. Let's examine each of these fields in turn.

The *event* is the name of the transition. Often this is the only thing specified for the transition. The transition name has an optional *argument* list to indicate when data is present in the transition, such as an error code or a monitored value. This argument list is enclosed within parentheses like a standard function call. A guard *condition* is shown in square brackets. A guard is a condition that must be met before the transition is taken. The *sendEvent* list is a comma-separated list of events, directed toward a given *target* object, each with possible *arguments*. Such events will be propagated outside of the enclosing object as a result of this transition. This is largely how concurrent state machines communicate, allowing a transition in one state machine to affect other concurrent state machines. Lastly, the *operation* list specifies a comma-separated list of functions (each with possible arguments) that will be called as a result of the transition being taken.

Within states, both *entry* and *exit actions*, as well as an ongoing *activity,* may be specified. An entry action is a function that is called when the state is entered (even when the transition is self-directed). An exit action is a function that is executed when the state is exited (even when the transition is self-directed). Activities denote processing that continues until completion, or until interrupted by a transition (even when the transition is self-directed). Figure 13 shows a simple timer behavioral model.

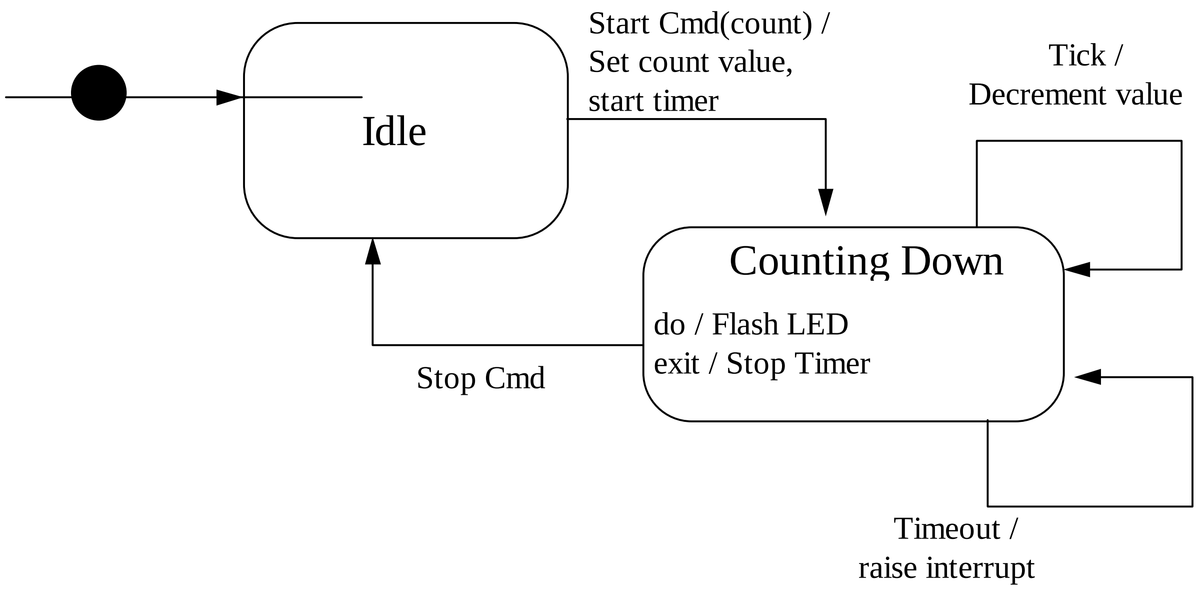

Figure 13: Retriggerable one-shot timer

Component Diagram

All of the previously discussed diagrams address elements of a system's *logical* model. By "logical," we mean that the system is being modeled somewhat independently of exactly how actual software components are named and organized. The purpose of a component diagram is to model the development view of a system's components and their relationships.

For each logical element in the model, there generally exists a default mapping to an implementation artifact. For example, a class from the logical model might map to two files in a C++ implementation: a .h file for the class definition and a .cpp file for the class member function definitions. If this default mapping was inappropriate for a certain class, a component diagram could be used to define a more appropriate representation.

Deployment Diagram

Real-time systems are often delivered on custom platforms, and the engineer must develop not only the software, but the hardware components as well. The hardware devices must be bound together with the portions of software they will run. The UML provides *deployment diagrams* to show the organization of the hardware and the binding of the software to the physical devices. Deployment diagrams show various hardware devices and their physical interfaces. The type of the hardware device is given by its *stereotype* (discussed below), such as Processor, Device, Display, Memory, Disk, and so on.

Advanced UML Features

The UML provides a number of advanced notations and semantics when more complex modeling is required. Although some of these advanced features are intended to cover the more "corner cases" of modeling needs, others are necessary for extending the UML in a controlled way and for supporting system modeling in-the-large. Two such advanced features are stereotypes and packages.

Stereotypes

A *stereotype* is the metaclassification of an element in the UML. It identifies the type of the element in the UML. For example, predefined UML class stereotypes include Event, Exception, Interface, Metaclass, and Utility. Predefined task stereotypes include Process and Thread.

The primary advantages of stereotypes are first that it is possible to refer to the type of the element, as in "That class is an Exception class;" and second that the UML is extensible by the user of the method through the definition additional stereotypes.

Stereotypes are indicated with a name that is enclosed by guillemots (« »), as is shown in Figure 14:

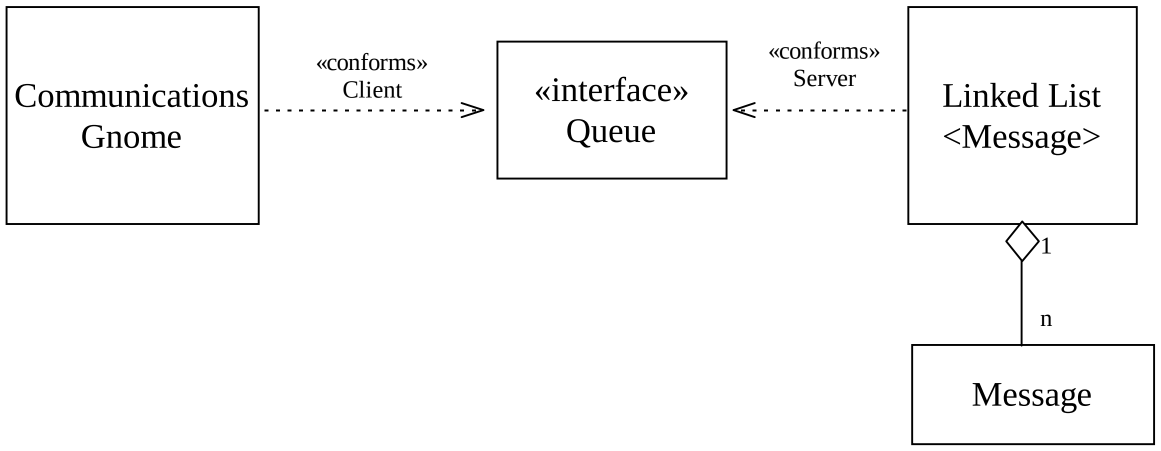

Figure 14: Use of stereotypes in class diagram

As shown in the figure, an instantiated Linked List class provides an underlying implementation model for managing a collection (of bus messages in this case). An interface class Queue provides the clients with queue functionality by providing queue-like access behavior even though the actual implementation uses a linked list. The stereotype of the Queue class is «interface» because it provides an interface for the Linked List class. The dependency relationship also has a stereotype called «conforms», meaning that the relationships conform to the specified interface.

Large-Scale Logical Packaging

For large-scale development, the UML supports the concept of *packages*. A package is a grouping of inherently cohesive entities. All of the classes in a model can be packaged by area of concern, such as user interface, device I/O, and so on. The implemented code can be packaged into subsystems that represented deployed software components. The notation for packages is a tabbed folder. Stereotypes clarify the type of package («category» for the class model or «subsystem» for the code model).

Packages provide a namespace for the items it contains. This becomes important as a system grows larger, and particularly when third-party software is being used. If a name collision occurs between components, they can be placed into separate packages and referenced with their fully qualified naming scheme:

packageName::componentName

Conclusion

The UML is a third-generation object-oriented modeling language that is particularly appropriate for real-time systems. It provides support for modeling classes, objects, and the many kinds of relationships among them, including association, aggregation, inheritance, dependency, and instantiation. Use cases are directly supported with scenarios for detailed descriptions of required system behavior. Interaction diagrams graphically model scenarios and can include both timing and message synchronization annotations. Enhanced finite state machine modeling supports a number of real-time features, including concurrency, event propagation, and nested states. The UML is itself extensible though the definition of additional stereotypes. Real-time developers can use the UML to model either simple or complex systems clearly and succinctly. Many CASE tool vendors have already committed to supporting the UML, and it is likely that the UML will become the standard notation for object-oriented systems development for the future.

Acknowledgments

The initial drafts of this paper were written by Bruce Douglass of A Priori Software. The current release was edited by Gary Cernosek of Rational Software Corporation. Any feedback regarding this paper may be sent to Gary Cernosek at garyc@rational.com.

References

[Booch94] Booch, G.: *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, 1994.

[OMT91] Rumbaugh, J., et al.: *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[OOSE92] Jacobson, I., et al.: *Object-Oriented Software Engineering,* Addison-Wesley, 1992.

[UML0.8] Booch, G. and Rumbaugh, J.: “Unified Method for Object-Oriented Development,” Documentation Set Version 0.8, October 1995.

[UML0.91] Booch, G.; Jacobson, I.; and Rumbaugh, J.: “The Unified Modeling Language for Object-Oriented Development,” Documentation Set Version 0.91 Addendum UML Update, September 1996.