APPENDIX

1 AN INTRODUCTION TO UML[1]

KEY CONCEPTS

The *Unified Modeling Language* (UML) is "a standard language for writing software blueprints. UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system" [Boo05]. In other words, just as building architects create blueprints to be used by a construction company, software architects create UML diagrams to help software developers build the software. If you understand the vocabulary of UML (the diagrams' pictorial elements and their meanings), you can much more easily understand and specify a system and explain the design of that system to others.

Grady Booch, Jim Rumbaugh, and Ivar Jacobson developed UML in the mid-1990s with much feedback from the software development community. UML merged a number of competing modeling notations that were in use by the software industry at the time. In 1997, UML 1.0 was submitted to the Object Management Group, a nonprofit consortium involved in maintaining specifications for use by the computer industry. UML 1.0 was revised to UML 1.1 and adopted later that year. The current standard is UML 2.0 and is now an ISO standard. Because this standard is so new, many older references, such as [Gam95] do not use UML notation.

UML 2.0 provides 13 different diagrams for use in software modeling. In this appendix, I will discuss only *class, deployment, use case, sequence, communication, activity,* and *state* diagrams. These diagrams are used in this edition of *Software Engineering: A Practitioner's Approach.*

You should note that there are many optional features in UML diagrams. The UML language provides these (sometimes arcane) options so that you can express all the important aspects of a system. At the same time, you have the flexibility to suppress those parts of the diagram that are not relevant to the aspect being modeled in order to avoid cluttering the diagram with irrelevant details. Therefore, the omission of a particular feature does not mean that the feature is absent; it may mean that the feature was suppressed. In this appendix, exhaustive coverage of all the features of the UML diagrams is *not* presented. Instead, I will focus on the standard options, especially those options that have been used in this book.

1 This appendix has been contributed by Dale Skrien and has been adapted from his book, *An Introduction to Object-Oriented Design and Design Patterns in Java* (McGraw-Hill, 2008). All content is used with permission.

Class Diagrams

To model classes, including their attributes, operations, and their relationships and associations with other classes,[2] UML provides a *class diagram.* A class diagram provides a static or structural view of a system. It does not show the dynamic nature of the communications between the objects of the classes in the diagram.

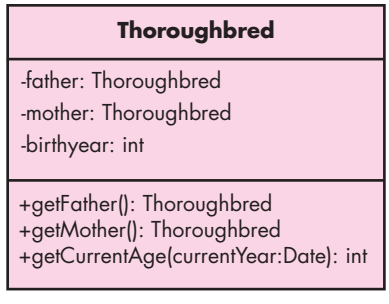
The main elements of a class diagram are boxes, which are the icons used to represent classes and interfaces. Each box is divided into horizontal parts. The top part contains the name of the class. The middle section lists the attributes of the class. An *attribute* refers to something that an object of that class knows or can provide all the time. Attributes are usually implemented as fields of the class, but they need not be. They could be values that the class can compute from its instance variables or values that the class can get from other objects of which it is composed. For example, an object might always know the current time and be able to return it to you whenever you ask. Therefore, it would be appropriate to list the current time as an attribute of that class of objects. However, the object would most likely not have that time stored in one of its instance variables, because it would need to continually update that field. Instead, the object would likely compute the current time (e.g., through consultation with objects of other classes) at the moment when the time is requested. The third section of the class diagram contains the operations or behaviors of the class. An *operation* refers to what objects of the class can do. It is usually implemented as a method of the class.

Figure A1.1 presents a simple example of a **Thoroughbred** class that models thoroughbred horses. It has three attributes displayed—**mother**, **father**, and **birthyear**. The diagram also shows three operations: *getCurrentAge(), getFather(),* and *getMother()*. There may be other suppressed attributes and operations not shown in the diagram.

Each attribute can have a name, a type, and a level of visibility. The type and visibility are optional. The type follows the name and is separated from the name by a

Figure A1.1

A class diagram for a Thoroughbred class



2 If you are unfamiliar with object-oriented concepts, a brief introduction is presented in Appendix 2.

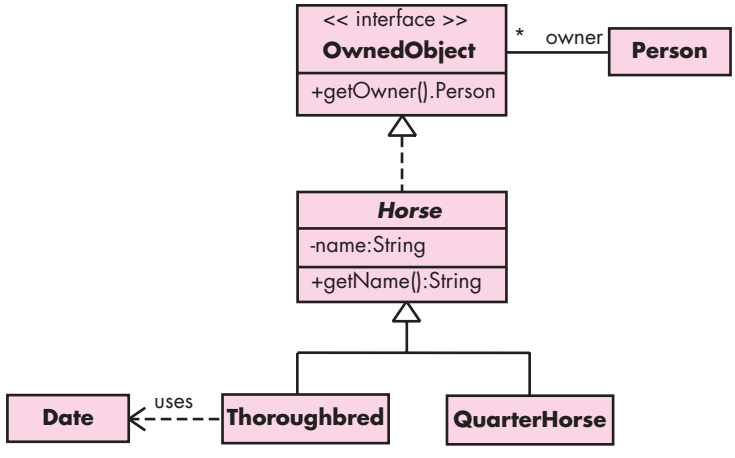
colon. The visibility is indicated by a preceding –, #, ~, or +, indicating, respectively, *private, protected, package,* or *public* visibility. In Figure A1.1, all attributes have private visibility, as indicated by the leading minus sign (–). You can also specify that an attribute is a static or class attribute by underlining it. Each operation can also be displayed with a level of visibility, parameters with names and types, and a return type.

An abstract class or abstract method is indicated by the use of italics for the name in the class diagram. See the **Horse** class in Figure A1.2 for an example. An interface is indicated by adding the phrase "«interface»" (called a *stereotype*) above the name. See the **OwnedObject** interface in Figure A1.2. An interface can also be represented graphically by a hollow circle.

It is worth mentioning that the icon representing a class can have other optional parts. For example, a fourth section at the bottom of the class box can be used to list the responsibilities of the class. This section is particularly useful when transitioning from CRC cards (Chapter 6) to class diagrams in that the responsibilities listed on the CRC cards can be added to this fourth section in the class box in the UML diagram before creating the attributes and operations that carry out these responsibilities. This fourth section is not shown in any of the figures in this appendix.

Class diagrams can also show relationships between classes. A class that is a subclass of another class is connected to it by an arrow with a solid line for its shaft and with a triangular hollow arrowhead. The arrow points from the subclass to the superclass. In UML, such a relationship is called a *generalization*. For example, in Figure A1.2, the **Thoroughbred** and **QuarterHorse** classes are shown to be subclasses of the **Horse** abstract class. An arrow with a dashed line for the arrow shaft indicates implementation of an interface. In UML, such a relationship is called a *realization*. For example, in Figure A1.2, the **Horse** class implements or realizes the **OwnedObject** interface.

FIGURE A1.2
A class diagram regarding horses



An *association* between two classes means that there is a structural relationship between them. Associations are represented by solid lines. An association has many optional parts. It can be labeled, as can each of its ends, to indicate the role of each class in the association. For example, in Figure A1.2, there is an association between **OwnedObject** and **Person** in which the **Person** plays the role of owner. Arrows on either or both ends of an association line indicate navigability. Also, each end of the association line can have a multiplicity value displayed. Navigability and multiplicity are explained in more detail later in this section. An association might also connect a class with itself, using a loop. Such an association indicates the connection of an object of the class with other objects of the same class.

An association with an arrow at one end indicates one-way navigability. The arrow means that from one class you can easily access the second associated class to which the association points, but from the second class, you cannot necessarily easily access the first class. Another way to think about this is that the first class is aware of the second class, but the second class object is not necessarily directly aware of the first class. An association with no arrows usually indicates a two-way association, which is what was intended in Figure A1.2, but it could also just mean that the navigability is not important and so was left off.

It should be noted that an attribute of a class is very much the same thing as an association of the class with the class type of the attribute. That is, to indicate that a class has a property called "name" of type String, one could display that property as an attribute, as in the **Horse** class in Figure A1.2. Alternatively, one could create a one-way association from the **Horse** class to the **String** class with the role of the **String** class being "name." The attribute approach is better for primitive data types, whereas the association approach is often better if the property's class plays a major role in the design, in which case it is valuable to have a class box for that type.

A *dependency* relationship represents another connection between classes and is indicated by a dashed line (with optional arrows at the ends and with optional labels). One class depends on another if changes to the second class might require changes to the first class. An association from one class to another automatically indicates a dependency. No dashed line is needed between classes if there is already an association between them. However, for a transient relationship (i.e., a class that does not maintain any long-term connection to another class but does use that class occasionally) we should draw a dashed line from the first class to the second. For example, in Figure A1.2, the **Thoroughbred** class uses the **Date** class whenever its *getCurrentAge()* method is invoked, and so the dependency is labeled "uses."

The *multiplicity* of one end of an association means the number of objects of that class associated with the other class. A multiplicity is specified by a nonnegative integer or by a range of integers. A multiplicity specified by "0..1" means that there are 0 or 1 objects on that end of the association. For example, each person in the world has either a Social Security number or no such number (especially if they are not U.S.

citizens), and so a multiplicity of 0..1 could be used in an association between a **Person** class and a **SocialSecurityNumber** class in a class diagram. A multiplicity specified by "1..*" means one or more, and a multiplicity specified by "0..*" or just "*" means zero or more. An * was used as the multiplicity on the **OwnedObject** end of the association with class **Person** in Figure A1.2 because a **Person** can own zero or more objects.

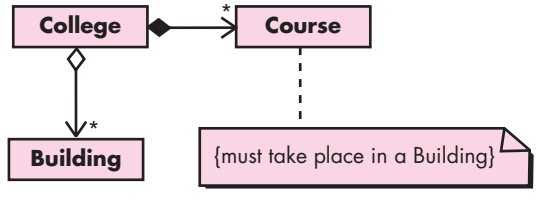
If one end of an association has multiplicity greater than 1, then the objects of the class referred to at that end of the association are probably stored in a collection, such as a set or ordered list. One could also include that collection class itself in the UML diagram, but such a class is usually left out and is implicitly assumed to be there due to the multiplicity of the association.

An *aggregation* is a special kind of association indicated by a hollow diamond on one end of the icon. It indicates a "whole/part" relationship, in that the class to which the arrow points is considered a "part" of the class at the diamond end of the association. A *composition* is an aggregation indicating strong ownership of the parts. In a composition, the parts live and die with the owner because they have no role in the software system independent of the owner. See Figure A1.3 for examples of aggregation and composition.

A **College** has an aggregation of **Building** objects, which represent the buildings making up the campus. The college also has a collection of courses. If the college were to fold, the buildings would still exist (assuming the college wasn't physically destroyed) and could be used for other things, but a **Course** object has no use outside of the college at which it is being offered. If the college were to cease to exist as a business entity, the **Course** object would no longer be useful and so it would also cease to exist.

Another common element of a class diagram is a *note*, which is represented by a box with a dog-eared corner and is connected to other icons by a dashed line. It can have arbitrary content (text and graphics) and is similar to comments in programming languages. It might contain comments about the role of a class or constraints that all objects of that class must satisfy. If the contents are a constraint, braces surround the contents. Note the constraint attached to the **Course** class in Figure A1.3.

FIGURE A1.3
The relationship between Colleges, Courses, and Buildings



Deployment Diagrams

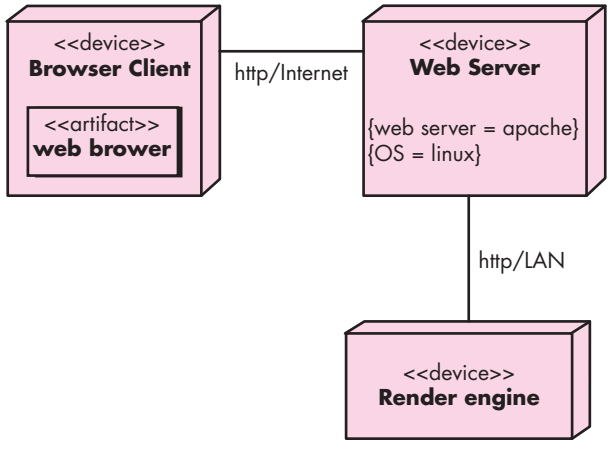
A UML *deployment diagram* focuses on the structure of a software system and is useful for showing the physical distribution of a software system among hardware platforms and execution environments. Suppose, for example, you are developing a Web-based graphics-rendering package. Users of your package will use their Web browser to go to your website and enter rendering information. Your website would render a graphical image according to the user's specification and send it back to the user. Because graphics rendering can be computationally expensive, you decide to move the rendering itself off the Web server and onto a separate platform. Therefore, there will be three hardware devices involved in your system: the Web client (the users' computer running a browser), the computer hosting the Web server, and the computer hosting the rendering engine.

Figure A1.4 shows the deployment diagram for such a package. In such a diagram, hardware components are drawn in boxes labeled with "«device»". Communication paths between hardware components are drawn with lines with optional labels. In Figure A1.4, the paths are labeled with the communication protocol and the type of network used to connect the devices.

Each node in a deployment diagram can also be annotated with details about the device. For example, in Figure A1.4, the browser client is depicted to show that it contains an artifact consisting of the Web browser software. An artifact is typically a file containing software running on a device. You can also specify tagged values, as is shown in Figure A1.4 in the Web server node. These values define the vendor of the Web server and the operating system used by the server.

Deployment diagrams can also display execution environment nodes, which are drawn as boxes containing the label "«execution environment»". These nodes represent systems, such as operating systems, that can host other software.

Figure A1.4
A deployment diagram



Use-Case Diagrams

Use cases (Chapters 5 and 6) and the UML *use-case diagram* help you determine the functionality and features of the software from the user's perspective. To give you a feeling for how use cases and use-case diagrams work, I'll create some for a software application for managing digital music files, similar to Apple's iTunes software. Some of the things the software might do include:

- Download an MP3 music file and store it in the application's library.
- Capture streaming music and store it in the application's library.
- Manage the application's library (e.g., delete songs or organize them in playlists).
- Burn a list of the songs in the library onto a CD.
- Load a list of the songs in the library onto an iPod or MP3 player.
- Convert a song from MP3 format to AAC format and vice versa.

This is not an exhaustive list, but it is sufficient to understand the role of use cases and use-case diagrams.

A *use case* describes how a user interacts with the system by defining the steps required to accomplish a specific goal (e.g., burning a list of songs onto a CD). Variations in the sequence of steps describe various scenarios (e.g., what if all the songs in the list don't fit on one CD?).

A UML use-case diagram is an overview of all the use cases and how they are related. It provides a big picture of the functionality of the system. A use-case diagram for the digital music application is shown in Figure A1.5.

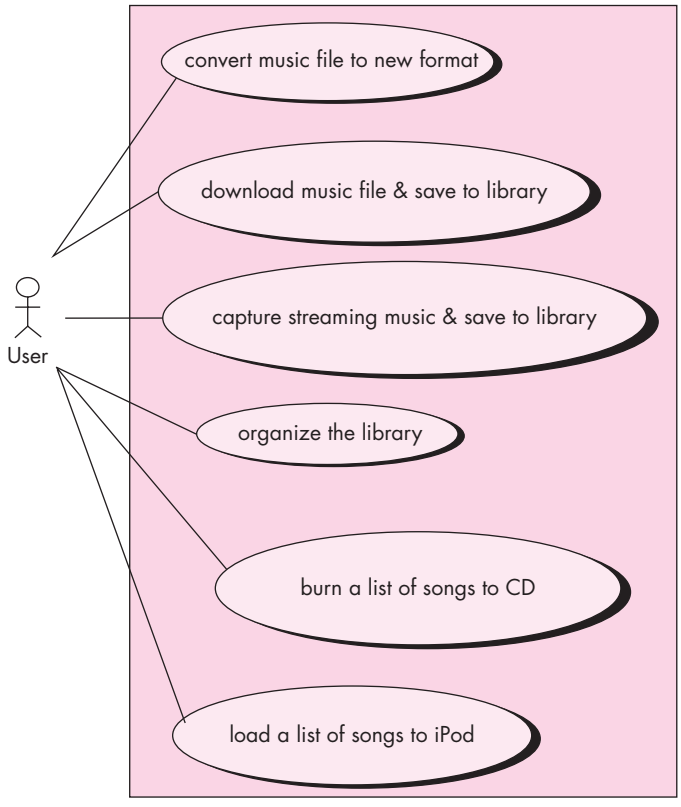
In this diagram, the stick figure represents an *actor* (Chapter 5) that is associated with one category of user (or other interaction element). Complex systems typically have more than one actor. For example, a vending machine application might have three actors representing customers, repair personnel, and vendors who refill the machine.

In the use-case diagram, the use cases are displayed as ovals. The actors are connected by lines to the use cases that they carry out. Note that none of the details of the use cases are included in the diagram and instead need to be stored separately. Note also that the use cases are placed in a rectangle but the actors are not. This rectangle is a visual reminder of the system boundaries and that the actors are outside the system.

Some use cases in a system might be related to each other. For example, there are similar steps in burning a list of songs to a CD and in loading a list of songs to an iPod. In both cases, the user first creates an empty list and then adds songs from the library to the list. To avoid duplication in use cases, it is usually better to create a new use case representing the duplicated activity, and then let the other uses cases include this new use case as one of their steps. Such inclusion is indicated in

FIGURE A1.5

A use-case diagram for the music system



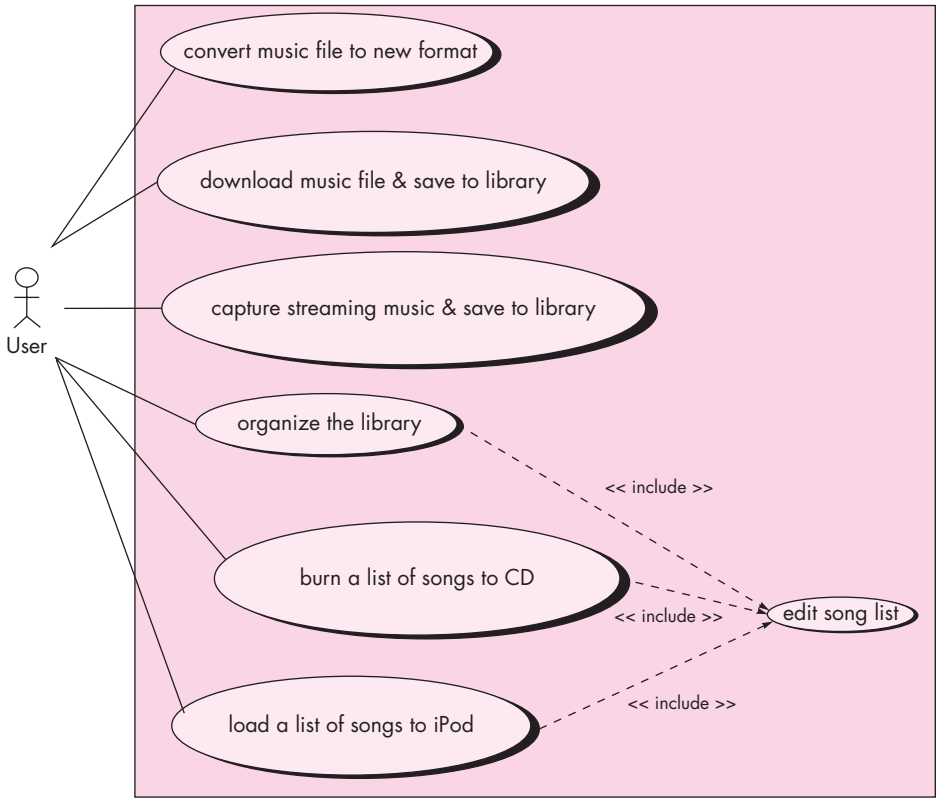
use-case diagrams, as in Figure A1.6, by means of a dashed arrow labeled «include» connecting a use case with an included use case.

A use-case diagram, because it displays all use cases, is a helpful aid for ensuring that you have covered all the functionality of the system. In the digital music organizer, you would surely want more use cases, such as a use case for playing a song in the library. But keep in mind that the most valuable contribution of use cases to the software development process is the textual description of each use case, not the overall use-case diagram. [Fow04b]. It is through the descriptions that you are able to form a clear understanding of the goals of the system you are developing.

SEQUENCE DIAGRAMS

In contrast to class diagrams and deployment diagrams, which show the static structure of a software component, a *sequence diagram* is used to show the dynamic communications between objects during execution of a task. It shows the temporal order in which messages are sent between the objects to accomplish that task. One might use a sequence diagram to show the interactions in one use case or in one scenario of a software system.

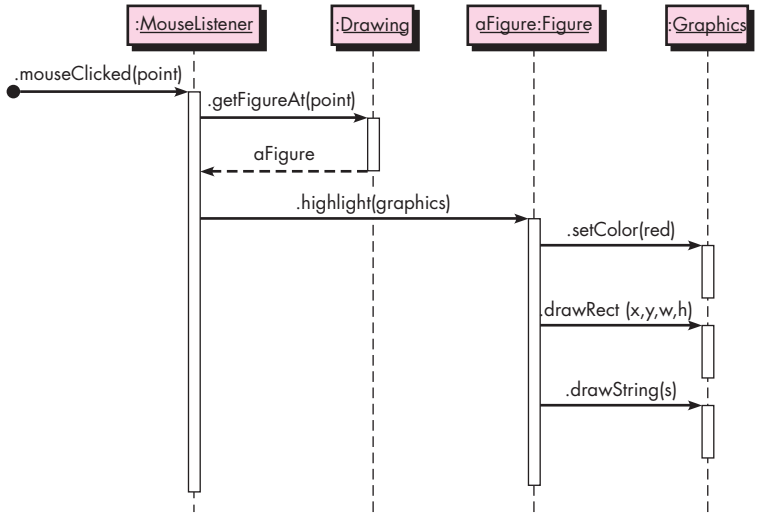
FIGURE A1.6
A use-case diagram with included use cases



In Figure A1.7, you see a sequence diagram for a drawing program. The diagram shows the steps involved in highlighting a figure in a drawing when it has been clicked. Each box in the row at the top of the diagram usually corresponds to an object, although it is possible to have the boxes model other things, such as classes. If the box represents an object (as is the case in all our examples), then inside the box you can optionally state the type of the object preceded by the colon. You can also precede the colon and type by a name for the object, as shown in the third box in Figure A1.7. Below each box there is a dashed line called the *lifeline* of the object. The vertical axis in the sequence diagram corresponds to time, with time increasing as you move downward.

A sequence diagram shows method calls using horizontal arrows from the *caller* to the *callee*, labeled with the method name and optionally including its parameters, their types, and the return type. For example, in Figure A1.7, the **MouseListener** calls the **Drawing**'s *getFigureAt()* method. When an object is executing a method (that is, when it has an activation frame on the stack), you can optionally display a white bar, called an *activation bar*, down the object's lifeline. In Figure A1.7, activation bars are drawn for all method calls. The diagram can also optionally show the return from a method call with a dashed arrow and an optional label. In Figure A1.7,

FIGURE A1.7
A sample sequence diagram



the *getFigureAt()* method call's return is shown labeled with the name of the object that was returned. A common practice, as we have done in Figure A1.7, is to leave off the return arrow when a void method has been called, since it clutters up the diagram while providing little information of importance. A black circle with an arrow coming from it indicates a *found message* whose source is unknown or irrelevant.

You should now be able to understand the task that Figure A1.7 is displaying. An unknown source calls the *mouseClicked()* method of a **MouseListener**, passing in the point where the click occurred as the argument. The **MouseListener** in turn calls the *getFigureAt()* method of a **Drawing**, which returns a **Figure.** The **MouseListener** then calls the highlight method of **Figure**, passing in a **Graphics** object as an argument. In response, **Figure** calls three methods of the **Graphics** object to draw the figure in red.

The diagram in Figure A1.7 is very straightforward and contains no conditionals or loops. If logical control structures are required, it is probably best to draw a separate sequence diagram for each case. That is, if the message flow can take two different paths depending on a condition, then draw two separate sequence diagrams, one for each possibility.

If you insist on including loops, conditionals, and other control structures in a sequence diagram, you can use *interaction frames*, which are rectangles that surround parts of the diagram and that are labeled with the type of control structures they represent. Figure A1.8 illustrates this, showing the process involved in highlighting all figures inside a given rectangle. The **MouseListener** is sent the **rectDragged** message. The **MouseListener** then tells the drawing to highlight all figures in the rectangle by called the method *highlightFigures()*, passing the rectangle as the argument. The method loops through all **Figure** objects in the **Drawing** object and, if the

FIGURE A1.8

A sequence diagram with two interaction frames

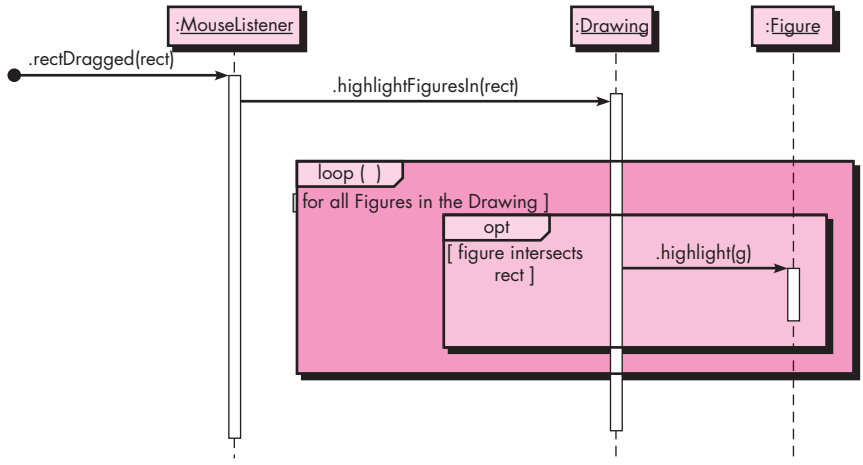


Figure intersects the rectangle, the **Figure** is asked to highlight itself. The phrases in square brackets are called *guards*, which are Boolean conditions that must be true if the action inside the interaction frame is to continue.

There are many other special features that can be included in a sequence diagram. For example:

1. You can distinguish between synchronous and asynchronous messages. Synchronous messages are shown with solid arrowheads while asynchronous messages are shown with stick arrowheads.
2. You can show an object sending itself a message with an arrow going out from the object, turning downward, and then pointing back to the same object.
3. You can show object creation by drawing an arrow appropriately labeled (for example, with a «create» label) to an object's box. In this case, the box will appear lower in the diagram than the boxes corresponding to objects already in existence when the action begins.
4. You can show object destruction by a big X at the end of the object's lifeline. Other objects can destroy an object, in which case an arrow points from the other object to the X. An X is also useful for indicating that an object is no longer usable and so is ready for garbage collection.

The last three features are all shown in the sequence diagram in Figure A1.9.

COMMUNICATION DIAGRAMS

The UML *communication diagram* (called a "collaboration diagram" in UML 1.X) provides another indication of the temporal order of the communications but emphasizes the relationships among the objects and classes instead of the temporal order.

FIGURE A1.9

Creation, destruction, and loops in sequence diagrams

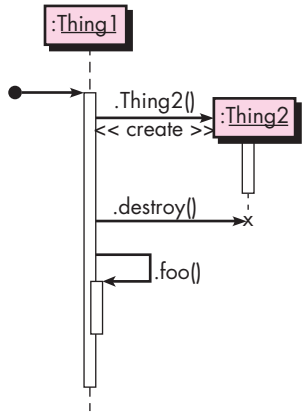
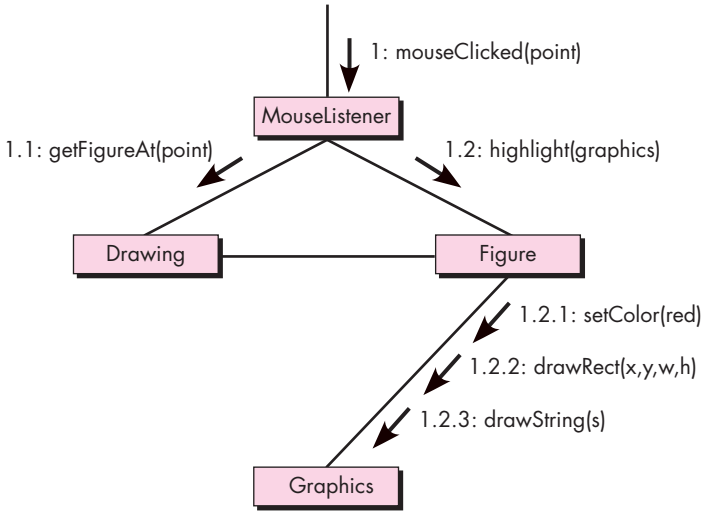


FIGURE A1.10

A UML communication diagram



A communication diagram, illustrated in Figure A1.10, displays the same actions shown in the sequence diagram in Figure A1.7.

In a communication diagram the interacting objects are represented by rectangles. Associations between objects are represented by lines connecting the rectangles. There is typically an incoming arrow to one object in the diagram that starts the sequence of message passing. That arrow is labeled with a number and a message name. If the incoming message is labeled with the number 1 and if it causes the receiving object to invoke other messages on other objects, then those messages are represented by arrows from the sender to the receiver along an association line and are given numbers 1.1, 1.2, and so forth, in the order they are called. If those

messages in turn invoke other messages, another decimal point and number are added to the number labeling these messages, to indicate further nesting of the message passing.

In Figure A1.10, you see that the **mouseClicked** message invokes the methods *getFigureAt()* and then *highlight().* The *highlight()* message invokes three other messages: *setColor(), drawRect(),* and *drawstring().* The numbering in each label shows the nesting as well as the sequential nature of each message.

There are many optional features that can be added to the arrow labels. For example, you can precede the number with a letter. An incoming arrow could be labeled **A1: mouseClicked(point)**. indicating an execution thread, A. If other messages are executed in other threads, their label would be preceded by a different letter. For example, if the *mouseClicked()* method is executed in thread A but it creates a new thread B and invokes *highlight()* in that thread, then the arrow from **MouseListener** to **Figure** would be labeled **1.B2: highlight(graphics)**.

If you are interested in showing the relationships among the objects in addition to the messages being sent between them, the communication diagram is probably a better option than the sequence diagram. If you are more interested in the temporal order of the message passing, then a sequence diagram is probably better.

Activity Diagrams

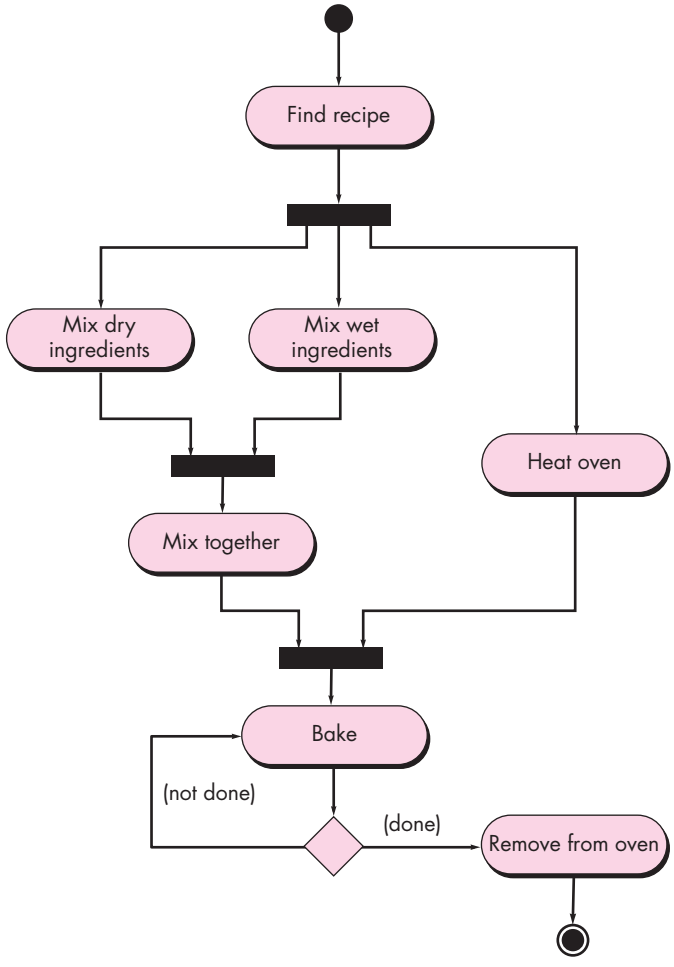
A UML *activity diagram* depicts the dynamic behavior of a system or part of a system through the flow of control between actions that the system performs. It is similar to a flowchart except that an activity diagram can show concurrent flows.

The main component of an activity diagram is an *action* node, represented by a rounded rectangle, which corresponds to a task performed by the software system. Arrows from one action node to another indicate the flow of control. That is, an arrow between two action nodes means that after the first action is complete the second action begins. A solid black dot forms the *initial node* that indicates the starting point of the activity. A black dot surrounded by a black circle is the *final node* indicating the end of the activity.

A *fork* represents the separation of activities into two or more concurrent activities. It is drawn as a horizontal black bar with one arrow pointing to it and two or more arrows pointing out from it. Each outgoing arrow represents a flow of control that can be executed concurrently with the flows corresponding to the other outgoing arrows. These concurrent activities can be performed on a computer using different threads or even using different computers.

Figure A1.11 shows a sample activity diagram involving baking a cake. The first step is finding the recipe. Once the recipe has been found, the dry ingredients and wet ingredients can be measured and mixed and the oven can be preheated. The mixing of the dry ingredients can be done in parallel with the mixing of the wet ingredients and the preheating of the oven.

FIGURE A1.11
A UML activity diagram showing how to bake a cake



A *join* is a way of synchronizing concurrent flows of control. It is represented by a horizontal black bar with two or more incoming arrows and one outgoing arrow. The flow of control represented by the outgoing arrow cannot begin execution until all flows represented by incoming arrows have been completed. In Figure A1.11, we have a join before the action of mixing together the wet and dry ingredients. This join indicates that all dry ingredients must be mixed and all wet ingredients must be mixed before the two mixtures can be combined. The second join in the figure indicates that, before the baking of the cake can begin, all ingredients must be mixed together and the oven must be at the right temperature.

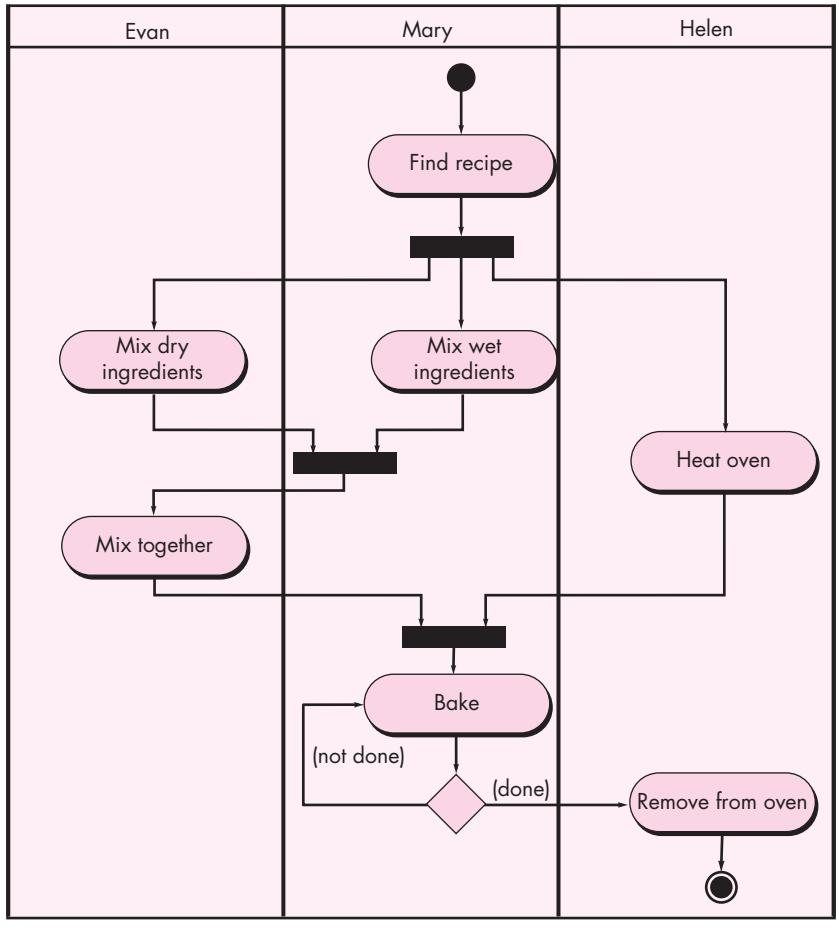
A *decision* node corresponds to a branch in the flow of control based on a condition. Such a node is displayed as a white triangle with an incoming arrow and two

or more outgoing arrows. Each outgoing arrow is labeled with a guard (a condition inside square brackets). The flow of control follows the outgoing arrow whose guard is true. It is advisable to make sure that the conditions cover all possibilities so that exactly one of them is true every time a decision node is reached. Figure A1.11 shows a decision node following the baking of the cake. If the cake is done, then it is removed from the oven. Otherwise, it is baked for a while longer.

One of the things the activity diagram in Figure A1.11 does not tell you is who or what does each of the actions. Often, the exact division of labor does not matter. But if you do want to indicate how the actions are divided among the participants, you can decorate the activity diagram with swimlanes, as shown in Figure A1.12. *Swimlanes*, as the name implies, are formed by dividing the diagram into strips or "lanes," each of which corresponds to one of the participants. All actions in one lane are done by the corresponding participant. In Figure A1.12, Evan is responsible for mixing the dry

FIGURE A1.12

The cake-baking activity diagram with swimlanes added



ingredients and then mixing the dry and wet ingredients together, Helen is responsible for heating the oven and taking the cake out, and Mary is responsible for everything else.

State Diagrams

The behavior of an object at a particular point in time often depends on the state of the object, that is, the values of its variables at that time. As a trivial example, consider an object with a Boolean instance variable. When asked to perform an operation, the object might do one thing if that variable is *true* and do something else if it is *false*.

A UML *state diagram* models an object's states, the actions that are performed depending on those states, and the transitions between the states of the object.

As an example, consider the state diagram for a part of a Java compiler. The input to the compiler is a text file, which can be thought of as a long string of characters. The compiler reads characters one at a time and from them determines the structure of the program. One small part of this process of reading the characters involves ignoring "white-space" characters (e.g., the *space, tab, newline,* and *return* characters) and characters inside a comment.

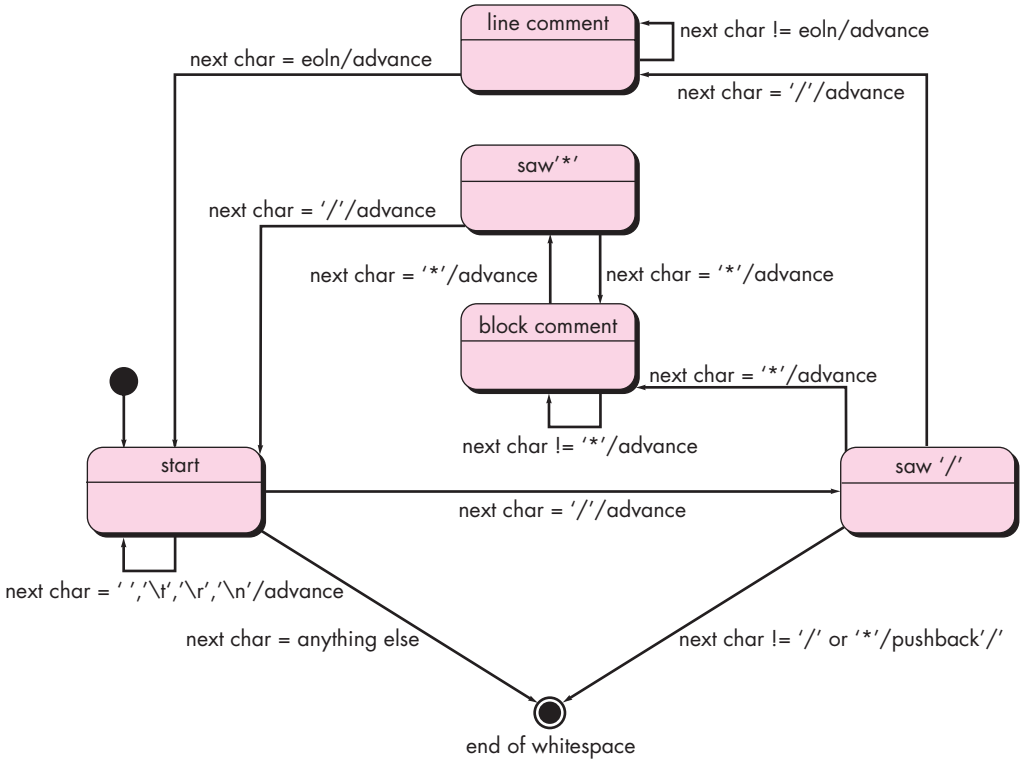
Suppose that the compiler delegates to a **WhiteSpaceAndCommentEliminator** the job of advancing over white-space characters and characters in comments. That is, this object's job is to read input characters until all white-space and comment characters have been read, at which point it returns control to the compiler to read and process non-white-space and noncomment characters. Think about how the **WhiteSpaceAndCommentEliminator** object reads in characters and determines whether the next character is white space or part of a comment. The object can check for white space by testing the next character against " ", "\t", "\n", and "\r". But how does it determine whether the next character is part of a comment? For example, when it sees a"/"for the first time, it doesn't yet know whether that character represents a division operator, part of the /= operator, or the beginning of a line or block comment. To make this determination, **WhiteSpaceAndCommentEliminator** needs to make a note of the fact that it saw a "/" and then move on to the next character. If the character following the "/" is another "/" or an "*", then **WhiteSpaceAndCommentEliminator** knows that it is now reading a comment and can advance to the end of the comment without processing or saving any characters. If the character following the first "/" is anything other than a "/" or an "*", then **WhiteSpaceAndCommentEliminator** knows that the "/" represents the division operator or part of the /= operator and so it stops advancing over characters.

In summary, as **WhiteSpaceAndCommentEliminator** reads in characters, it needs to keep track of several things, including whether the current character is white space, whether the previous character it read was a "/", whether it is currently reading characters in a comment, whether it has reached the end of comment, and so forth.

These all correspond to different states of the **WhiteSpaceAndCommentEliminator** object. In each of these states, **WhiteSpaceAndCommentEliminator** behaves differently with regard to the next character read in.

To help you visualize all the states of this object and how it changes state, you can use a UML state diagram as shown in Figure A1.13. A state diagram displays states using rounded rectangles, each of which has a name in its upper half. There is also a black circle called the "initial pseudostate," which isn't really a state and instead just points to the initial state. In Figure A1.13, the **start** state is the initial state. Arrows from one state to another state indicate transitions or changes in the state of the object. Each transition is labeled with a trigger event, a slash (/), and an activity. All parts of the transition labels are optional in state diagrams. If the object is in one state and the trigger event for one of its transitions occurs, then that transition's activity is performed and the object takes on the new state indicated by the transition. For example, in Figure A1.13, if the **WhiteSpaceAndCommentEliminator** object is in the **start** state and the next character is "/", then **WhiteSpaceAndCommentEliminator** advances past that character and changes to the **saw '/'** state. If the character after the "/" is another "/", then the object advances to the **line comment** state and stays there until it reads

FIGURE A1.13 **A state diagram for advancing past white space and comments in Java**



an end-of-line character. If instead the next character after the "/" is a "*", then the object advances to the **block comment** state and stays there until it sees another "*" followed by a "/", which indicates the end of the block comment. Study the diagram to make sure you understand it. Note that, after advancing past white space or a comment, **WhiteSpaceAndCommentEliminator** goes back to the **start** state and starts over. That behavior is necessary since there might be several successive comments or white-space characters before any other characters in the Java source code.

An object may transition to a final state, indicated by a black circle with a white circle around it, which indicates there are no more transitions. In Figure A1.13, the **WhiteSpaceAndCommentEliminator** object is finished when the next character is not white space or part of a comment. Note that all transitions except the two transitions leading to the final state have activities consisting of advancing to the next character. The two transitions to the final state do not advance over the next character because the next character is part of a word or symbol of interest to the compiler. Note that if the object is in the **saw '/'** state but the next character is not "/" or "*", then the "/" is a division operator or part of the /= operator and so we don't want to advance. In fact, we want to back up one character to make the "/" into the next character, so that the "/" can be used by the compiler. In Figure A1.13, this activity of backing up is labeled as pushback '/'.

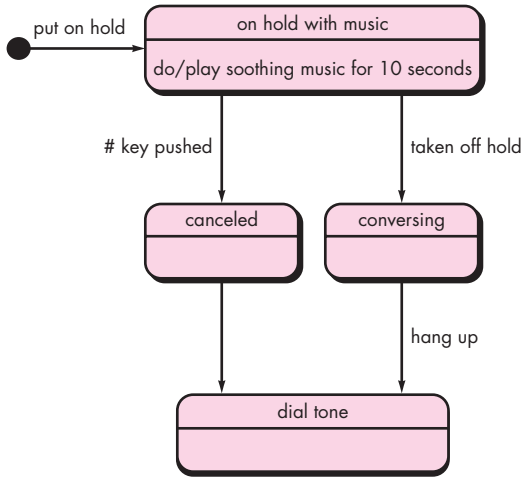
A state diagram will help you to uncover missed or unexpected situations. That is, with a state diagram, it is relatively easy to ensure that all possible trigger events for all possible states have been accounted for. For example, in Figure A1.13, you can easily verify that every state has included transitions for all possible characters.

UML state diagrams can contain many other features not included in Figure A1.13. For example, when an object is in a state, it usually does nothing but sit and wait for a trigger event to occur. However, there is a special kind of state, called an *activity state*, in which the object performs some activity, called a *do-activity*, while it is in that state. To indicate that a state is an activity state in the state diagram, you include in the bottom half of the state's rounded rectangle the phrase "do/" followed by the activity that is to be done while in that state. The do-activity may finish before any state transitions occur, after which the activity state behaves like a normal waiting state. If a transition out of the activity state occurs before the do-activity is finished, then the do-activity is interrupted.

Because a trigger event is optional when a transition occurs, it is possible that no trigger event may be listed as part of a transition's label. In such cases for normal waiting states, the object will immediately transition from that state to the new state. For activity states, such a transition is taken as soon as the do-activity finishes.

Figure A1.14 illustrates this situation using the states for a business telephone. When a caller is placed on hold, the call goes into the **on hold with music** state (soothing music is played for 10 seconds). After 10 seconds, the do-activity of the state is completed and the state behaves like a normal nonactivity state. If the caller pushes the # key when the call is in the **on hold with music** state, the call

FIGURE A1.14
A state diagram with an activity state and a triggerless transition



transitions to the **canceled** state and then transitions immediately to the **dial tone** state. If the # key is pushed before the 10 seconds of soothing music has completed, the do-activity is interrupted and the music stops immediately.

OBJECT CONSTRAINT LANGUAGE—AN OVERVIEW

The wide variety of diagrams available as part of UML provide you with a rich set of representational forms for the design model. However, graphical representations are often not enough. You may need a mechanism for explicitly and formally representing information that constrains some element of the design model. It is possible, of course, to describe constraints in a natural language such as English, but this approach invariably leads to inconsistency and ambiguity. For this reason, a more formal language—one that draws on set theory and formal specification languages (see Chapter 21) but has the somewhat less mathematical syntax of a programming language—seems appropriate.

The *Object Constraint Language* (OCL) complements UML by allowing you to use a formal grammar and syntax to construct unambiguous statements about various design model elements (e.g., classes and objects, events, messages, interfaces). The simplest OCL statements are constructed in four parts: (1) a *context* that defines the limited situation in which the statement is valid, (2) a *property* that represents some characteristics of the context (e.g., if the context is a class, a property might be an attribute), (3) an *operation* (e.g., arithmetic, set-oriented) that manipulates or qualifies a property, and (4) keywords (e.g., **if, then, else, and, or, not, implies**) that are used to specify conditional expressions.

As a simple example of an OCL expression, consider the printing system discussed in Chapter 10. The guard condition placed on the **jobCostAccepted** event that

causes a transition between the states *computingJobCost* and *formingJob* within the statechart diagram for the **PrintJob** object (Figure 10.9). In the diagram (Figure 10.9), the guard condition is expressed in natural language and implies that authorization can only occur if the customer is authorized to approve the cost of the job. In OCL, the expression may take the form:

```
customer
   self.authorizationAuthority = 'yes'
```

where a Boolean attribute, **authorizationAuthority,** of the class (actually a specific instance of the class) named **Customer** must be set to "yes" for the guard condition to be satisfied.

As the design model is created, there are often instances in which pre- or post-conditions must be satisfied prior to completion of some action specified by the design. OCL provides a powerful tool for specifying pre- and postconditions in a formal manner. As an example, consider an extension to the print shop system (discussed as an example in Chapter 10) in which the customer provides an upper cost bound for the print job and a "drop-dead" delivery date at the same time as other print job characteristics are specified. If cost and delivery estimates exceed these bounds, the job is not submitted and the customer must be notified. In OCL, a set of pre- and postconditions may be specified in the following manner:

```
context PrintJob::validate(upperCostBound : Integer, custDeliveryReq :
   Integer)
 pre: upperCostBound > 0
   and custDeliveryReq > 0
   and self.jobAuthorization = 'no'
 post: if self.totalJobCost <= upperCostBound
      and self.deliveryDate <= custDeliveryReq
    then
     self.jobAuthorization = 'yes'
    endif
```

This OCL statement defines an invariant **(inv)**—conditions that must exist prior to (pre) and after (post) some behavior. Initially, a precondition establishes that bounding cost and delivery date must be specified by the customer, and authorization must be set to "no." After costs and delivery are determined, the postcondition specified is applied. It should also be noted that the expression:

```
self.jobAuthorization = 'yes'
```

is not assigning the value "yes" but is declaring that the **jobAuthorization** must have been set to "yes" by the time the operation finishes. A complete description of OCL is beyond the scope of this appendix. The complete OCL specification can be obtained at **www.omg.org/technology/documents/formal/ocl.htm**.