Use Case Realization by Means of Sequence Diagrams

Analysis Model

Activity Diagram

Use Case Model

Sequence Diagram

Class Diagram

State Diagram

User Requirement

Use Case Specification

Storyboard

Functional Requirement

Nonfunctional Requirement

Software Requirement Specification

Interface Requirements Specification
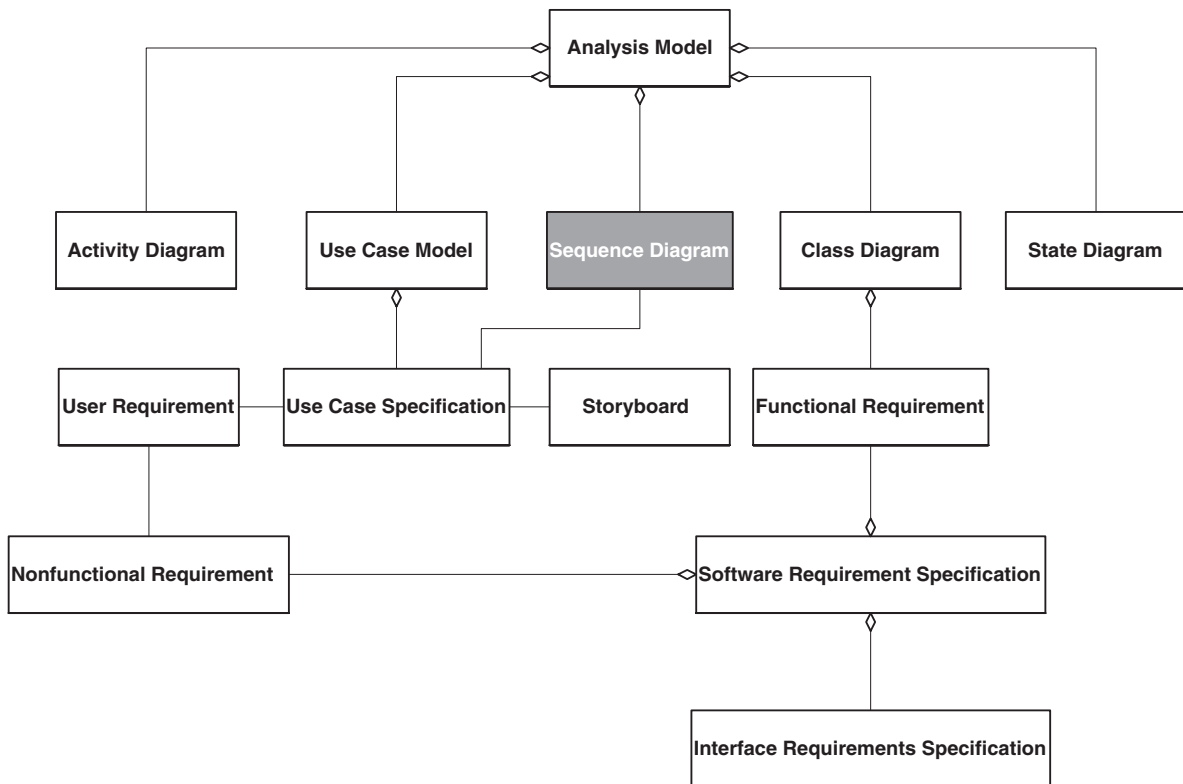
We use sequence diagrams to realize use cases in the analysis model. Before we demonstrate the realization, we need to have a good understanding of sequence diagrams. Sequence diagrams are a type of interaction diagram. Collaboration diagrams are another type of interaction diagram. Although each of these types of interaction diagrams provides the same information, the focus of attention is different. Collaboration diagrams focus on the objects that work together to accomplish a given task or series of tasks. Sequence diagrams focus on the interaction of a given task or series of tasks as observed over time. In fact, some modeling tools automatically convert one diagram to the other. In this chapter we focus on sequence diagrams since use cases model the services the system provides over time.

We discuss the elements that make up the sequence diagram. We then discuss how we map use cases onto a class diagram with sequence diagrams. Finally, we map a use case from our Change Management System onto the class diagram or analysis model of the Change Management System.

Introduction to Sequence Diagrams

A sequence diagram is a collection of objects interacting to accomplish a given task or series of tasks over time. Objects appear across the top of the diagram. A dashed line extends from the object to the bottom of the sequence diagram. Time is represented on the vertical axis. Methods that appear higher on the diagram occur earlier than methods that appear lower on the diagram. Figure 13.1 illustrates a sequence diagram.

Figure 13.1 shows a simple sequence diagram. In the example, an object of type *USER* triggers the occurrence of some event by calling the method *METHODA* in the object *OBJECTA* of type *CLASSA*. *METHODA* then calls the method *METHODB* in *OBJECTB* of type *CLASSB*. *METHODB* then calls method *PRIVATEMETHODB* within the object that contains *METHODB*.

The arrowhead points to the method that was called. Information can flow both ways. If the method is fully specified, as is the case for *METHODB* of *OBJECTB*, the information about the call is included in the method call. The return value is not shown in the sequence diagram.

The thick bar in the sequence diagram indicates the focus of control. Focus of control implies that one method called another; control will return to the first method. This focus can continue through any number of method calls. Focus of control does not always return to the calling method. Sometimes the new method starts a new thread of execution that takes the focus of control.
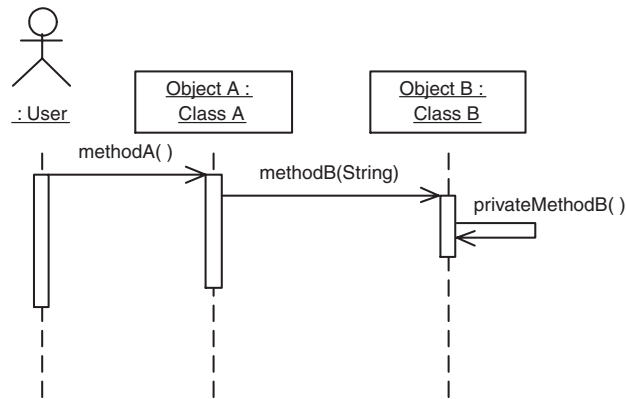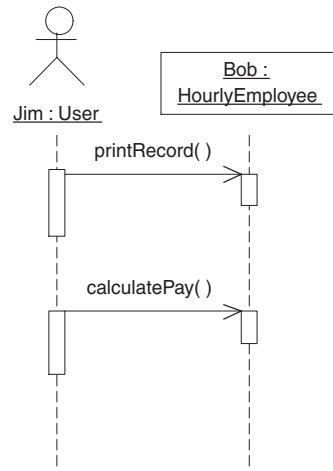
FIGURE 13.1
Sequence diagram

Recall from Chapter 11, that for a method from one object to call a method from another object, that object must have a public access specifier. Private methods can only be called by methods within the same object. Protected methods in a base class can be called by other methods declared in the same class or by methods that are declared in a derived class. If a method in a base class is defined as private, only methods declared in the base class can call that method. In Figure 13.2 we look at another example.

The example in Figure 13.2 shows an inheritance tree for an employee record. In the example, all objects of type EMPLOYEE include a method PRINTRECORD. The EMPLOYEE class also includes method CALCULATEPAY to allow polymorphism. Each of the derived objects—HOURLYEMPLOYEE and
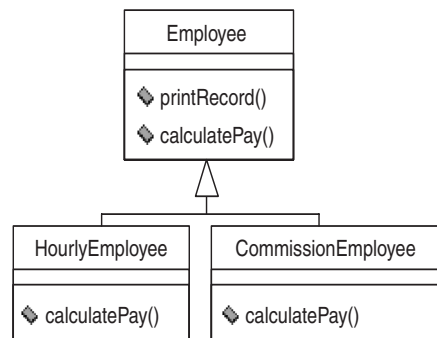
FIGURE 13.2
Employee inheritance tree

FIGURE 13.3
Employee sequence diagram

COMMISSIONEMPLOYEE—has its own version of the method *CALCULATEPAY*. We use objects from this set of classes in our next sequence diagram, Figure 13.3.

Figure 13.3 illustrates a sequence diagram that uses our Employee inheritance tree. We have instantiated an instance of a *USER* class and named the instance *JIM*. *JIM* initiates the method *PRINTRECORD* on the object *BOB*. The object *BOB* is of type *HOURLYEMPLOYEE*. *HOURLYEMPLOYEE* knows about the method *PRINTRECORD* because it inherited the method from *EMPLOYEE*. Next, *JIM* initiates the method *CALCULATEPAY* on our object, *BOB*. We can be assured that the correct *CALCULATEPAY* method will be called without regard to the derived type. This correct behavior appears natural in this example, but imagine if we had an array of objects of type *EMPLOYEE* with random objects in the array being of type *HOURLYEMPLOYEE* and the remainder being *COMMISSIONEMPLOYEE*. In that situation, we would depend entirely on polymorphism to determine the correct *CALCULATEPAY* to execute.

Realizing Use Cases in Sequence Diagrams

Realizing use cases by means of sequence diagrams is an important part of our analysis. It ensures that we have an accurate and complete class diagram. The sequence diagrams increase the completeness and understandability of

our analysis model. Often, analysts use the sequence diagram to assign responsibilities to classes. The behavior is associated with the class the first time it is required, and then the behavior is reused for every other use case that requires the behavior.

When assigning behaviors or responsibilities to a class while mapping a use case to the analysis model, you must take special care to assign the responsibility to the correct class. The responsibility or behavior belongs to the class if it is something you would do to the thing the class represents. For example, if our class represented a table and our application must keep track of the physical location of the table, we would expect to find the method *MOVE* in the class. We also expect the object to maintain all the information associated with an object of a given type. Therefore, the *TABLE* class should include the method *WHERELOCATED.* This simple rule of thumb states that a class will include any methods needed to provide information about an object of the given class, will provide necessary methods to manipulate the objects of the given class, and will be responsible for creating and deleting instances of the class.

Another guideline helpful in building an analysis model is to examine the model from the whole-part perspective. The start of the sequence diagram is normally the most difficult aspect. It is important to have access to the object on which you will call a method to start the sequence. It is often the case that you will have to return to the whole to navigate through the whole-part relationship to arrive at the class you will be working with.

A simple example illustrates the whole-part relationship navigation. Suppose you were asked to read the first paragraph of three chapters of a book. First, you would need to know where to go to get the book. We might state that all books we are referring to are available at the Fourth St. library. You might then know to first go to the library, but the library has thousands of books. You might next have to consult the card catalog to determine where the book is located and then retrieve the book. Next, you might look at the book's table of contents to determine which pages concern you and then turn to those pages. We could consider the library as the whole and the books as the part of the whole-part relationship. The relationship between the book and the pages could also be viewed as a whole-part relationship. When we determined where to look and then proceeded to find that point, we were navigating the whole-part relationship.

Every time you find yourself navigating the whole-part relationship to find the appropriate class, you will need to assign responsibilities to the classes you are navigating to ensure that you can, in fact, find the appropriate class. Said another way, the navigating behavior must be a method on the

class representing the whole. It is not unusual that this requires returning to the class that represents the system itself. Figures 13.4 and 13.5 illustrate this point.

The class diagram in Figure 13.4 shows three classes. Objects of type *POINTOFSALE* have zero to many objects of type *CUSTOMER*. Objects of type *CUSTOMER* have zero to many objects of type *ORDER*. Suppose the system
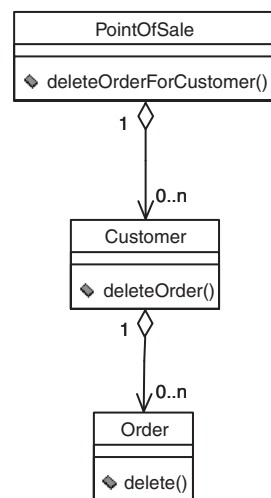
FIGURE 13.4
Whole-part class diagram
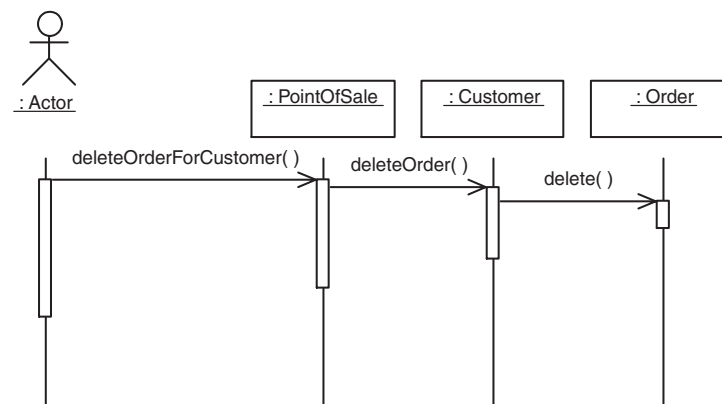
FIGURE 13.5
Whole-part sequence diagram

receives a message from an actor requesting that you delete a given order belonging to a given customer. The sequence diagram might look like the example in Figure 13.5.

The sequence diagram requires the system to navigate the whole-part relationships to delete the order specified by the object of type *ACTOR*. The sequence of events begins when the object of type *ACTOR* requests that a specific order be deleted for a specific customer. There is no way for the object of type *ACTOR* to call the delete method on the object of the type *ORDER* because the object of type *ACTOR* does not have a reference to the specific order. It is appropriate for the object of type *ACTOR* to have a reference to the object of type *POINTOFSALE*. This follows because there is only one object of type *POINTOFSALE* and it can, therefore, be referenced by name. The logical starting point for all interaction with the actor is the object of type *POINTOF-SALE*. The object of type *ACTOR* can then traverse the whole-part relationships to arrive at the specific object of type *ORDER* for which the action is intended. The responsibilities for navigating the whole-part relationship result in assigning behaviors to the object of type *POINTOFSALE* and the object of type *CUSTOMER*.

Example Sequence Diagram for the Change Management System

The Change Management System includes a method to create new versions for a given system. Figure 13.6 illustrates this activity by viewing the collaboration of objects required to accomplish this task with respect to time.

Figure 13.6 shows a sequence diagram from our Change Management System. In this sequence diagram, we assume the user has already logged in to the system. The example shows an instance of the *ACTOR* class, *JIM*. *JIM* begins the use case by requesting to work with versions, effected by clicking the Version Button. This action causes the *VERSIONSCREEN* object to be displayed.

Next, *JIM* indicates the intent to add a new version for the selected system by pressing the Add button. This action calls the *CLICKADD* method on the object of type *VERSIONSCREEN*. This method then triggers the screen's *CLEARFIELDS* method, which prepares the screen for the addition of the version.

The user then provides the appropriate information for the new version. Once *JIM* has provided all the information for the new version and clicks the Save button, the method *CLICKSAVE* is executed. The *CLICKSAVE* method calls
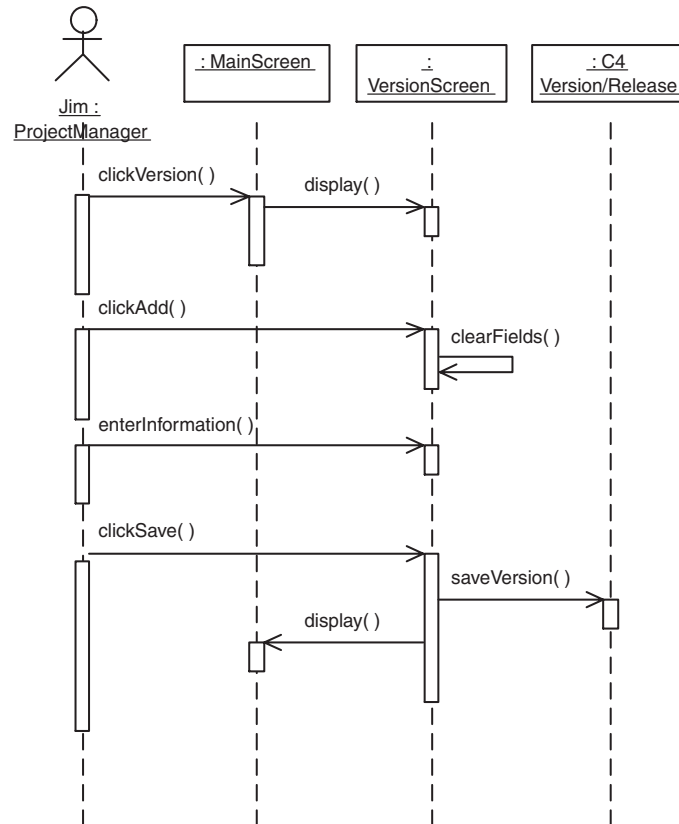
FIGURE 13.6
Create version sequence diagram

the method *SAVEVERSION* on the object of type *VERSION*. Once the new version is saved, the *CLICKSAVE* method of the object of type *VERSIONSCREEN* calls the *DISPLAY* method on the object of type *MAINSCREEN*.

This interaction ensures that our analysis model supports the use case Create Version. Once we have completed this mapping for all use cases in our Change Management System, we are sure that our analysis model fully supports our User Requirements. We are now ready to begin specifying the software requirements.