7 Interaction Modeling

The interaction model is the third leg of the modeling tripod and describes interactions within a system. The class model describes the objects in a system and their relationships, the state model describes the life cycles of the objects, and the interaction model describes how the objects interact.

The interaction model describes how objects interact to produce useful results. It is a holistic view of behavior across many objects, whereas the state model is a reductionist view of behavior that examines each object individually. Both the state model and the interaction model are needed to describe behavior fully. They complement each other by viewing behavior from two different perspectives.

Interactions can be modeled at different levels of abstraction. At a high level, use cases describe how a system interacts with outside actors. Each use case represents a piece of functionality that a system provides to its users. Use cases are helpful for capturing informal requirements.

Sequence diagrams provide more detail and show the messages exchanged among a set of objects over time. Messages include both asynchronous signals and procedure calls. Sequence diagrams are good for showing the behavior sequences seen by users of a system.

And finally, activity diagrams provide further detail and show the flow of control among the steps of a computation. Activity diagrams can show data flows as well as control flows. Activity diagrams document the steps necessary to implement an operation or a business process referenced in a sequence diagram.

7.1 Use Case Models

7.1.1 Actors

An ***actor*** is a direct external user of a system—an object or set of objects that communicates directly with the system but that is not part of the system. Each actor represents those objects that behave in a particular way toward the system. For example, *customer* and *repair technician* are different actors of a vending machine. For a travel agency system, actors might include *traveler, agent,* and *airline*. For a computer database system, actors might include *user* and *administrator*. Actors can be persons, devices, and other systems—anything that interacts directly with the system.

An object can be bound to multiple actors if it has different facets to its behavior. For example, the objects Mary, Frank, and Paul may be customers of a vending machine. Paul may also be a repair technician for the vending machine.

An actor has a single well-defined purpose. In contrast, objects and classes often combine many different purposes. An actor represents a particular facet of objects in its interaction with a system. The same actor can represent objects of different classes that interact similarly toward a system. For example, even though many different individual persons use a vending machine, their behavior toward the vending machine can all be summarized by the actors *customer* and *repair technician*. Each actor represents a coherent set of capabilities for its objects.

Modeling the actors helps to define a system by identifying the objects within the system and those on its boundary. An actor is directly connected to the system—an indirectly connected object is not an actor and should not be included as part of the system model. Any interactions with an indirectly connected object must pass through the actors. For example, the dispatcher of repair technicians from a service bureau is not an actor of a vending machine—only the repair technician interacts directly with the machine. If it is necessary to model the interactions among such indirect objects, then a model should be constructed of the environment itself as a larger system. For example, it might be useful to build a model of a repair service that includes dispatchers, repair technicians, and vending machines as actors, but that is a different model from the vending machine model.

7.1.2 Use Cases

The various interactions of actors with a system are quantized into use cases. A ***use case*** is a coherent piece of functionality that a system can provide by interacting with actors. For example, a *customer* actor can *buy a beverage* from a vending machine. The customer inserts money into the machine, makes a selection, and ultimately receives a beverage. Similarly, a *repair technician* can *perform scheduled maintenance* on a vending machine. Figure 7.1 summarizes several use cases for a vending machine.

Each use case involves one or more actors as well as the system itself. The use case *buy a beverage* involves the *customer* actor and the use case *perform scheduled maintenance* involves the *repair technician* actor. In a telephone system, the use case *make a call* involves two actors, a *caller* and a *receiver*. The actors need not all be persons. The use case *make a trade* on an online stock broker involves a *customer* actor and a *stock exchange* actor. The stock broker system needs to communicate with both actors to execute a trade.

A use case involves a sequence of messages among the system and its actors. For example, in the *buy a beverage* use case, the customer first inserts a coin and the vending machine displays the amount deposited. This can be repeated several times. Then the customer pushes

- **Buy a beverage**. The vending machine delivers a beverage after a customer selects and pays for it.
- **Perform scheduled maintenance**. A repair technician performs the periodic service on the vending machine necessary to keep it in good working condition.
- **Make repairs**. A repair technician performs the unexpected service on the vending machine necessary to repair a problem in its operation.
- **Load items**. A stock clerk adds items into the vending machine to replenish its stock of beverages.

Figure 7.1 Use case summaries for a vending machine. A use case is a coherent piece of functionality that a system can provide by interacting with actors.

a button to indicate a selection; the vending machine dispenses the beverage and issues change, if necessary.

Some use cases have a fixed sequence of messages. More often, however, the message sequence may have some variations. For example, a customer can deposit a variable number of coins in the *buy a beverage* use case. Depending on the money inserted and the item selected, the machine may, or may not, return change. You can represent such variability by showing several examples of distinct behavior sequences. Typically you should first define a mainline behavior sequence, then define optional subsequences, repetitions, and other variations.

Error conditions are also part of a use case. For example, if the customer selects a beverage whose supply is exhausted, the vending machine displays a warning message. Similarly, the vending transaction can be cancelled. For example, the customer can push the coin return on the vending machine at any time before a selection has been accepted; the machine returns the coins, and the behavior sequence for the use case is complete. From the user's point of view, some kinds of behavior may be thought of as errors. The designer, however, should plan for all possible behavior sequences. From the system's point of view, user errors or resource failures are just additional kinds of behavior that a robust system can accommodate.

A use case brings together all of the behavior relevant to a slice of system functionality. This includes normal mainline behavior, variations on normal behavior, exception conditions, error conditions, and cancellations of a request. Figure 7.2 explains the *buy a beverage* use case in detail. Grouping normal and abnormal behavior under a single use case helps to ensure that all the consequences of an interaction are considered together.

In a complete model, the use cases partition the functionality of the system. They should preferably all be at a comparable level of abstraction. For example, the use cases *make telephone call* and *record voice mail message* are at comparable levels. The use case *set external speaker volume to high* is too narrow. It would be better as *set speaker volume* (with the volume level selection as part of the use case) or maybe even just *set telephone parameters*, under which we might group setting volume, display pad settings, setting the clock, and so on.

Use Case: Buy a beverage

Summary: The vending machine delivers a beverage after a customer selects and pays for it.

Actors: Customer

Preconditions: The machine is waiting for money to be inserted.

Description: The machine starts in the waiting state in which it displays the message "Enter coins." A customer inserts coins into the machine. The machine displays the total value of money entered and lights up the buttons for the items that can be purchased for the money inserted. The customer pushes a button. The machine dispenses the corresponding item and makes change, if the cost of the item is less than the money inserted.

Exceptions:

Canceled: If the customer presses the cancel button before an item has been selected, the customer's money is returned and the machine resets to the waiting state.

Out of stock: If the customer presses a button for an out-of-stock item, the message "That item is out of stock" is displayed. The machine continues to accept coins or a selection.

Insufficient money: If the customer presses a button for an item that costs more than the money inserted, the message "You must insert $*nn.nn* more for that item" is displayed, where *nn.nn* is the amount of additional money needed. The machine continues to accept coins or a selection.

No change: If the customer has inserted enough money to buy the item but the machine cannot make the correct change, the message "Cannot make correct change" is displayed and the machine continues to accept coins or a selection.

Postconditions: The machine is waiting for money to be inserted.

Figure 7.2 Use case description. A use case brings together all of the behavior relevant to a slice of system functionality.

7.1.3 Use Case Diagrams

A system involves a set of use cases and a set of actors. Each use case represents a slice of the functionality the system provides. The set of use cases shows the complete functionality of the system at some level of detail. Similarly, each actor represents one kind of object for which the system can perform behavior. The set of actors represents the complete set of objects that the system can serve. Objects accumulate behavior from all the systems with which they interact as actors.

The UML has a graphical notation for summarizing use cases and Figure 7.3 shows an example. A rectangle contains the use cases for a system with the actors listed on the outside. The name of the system may be written near a side of the rectangle. A name within an ellipse

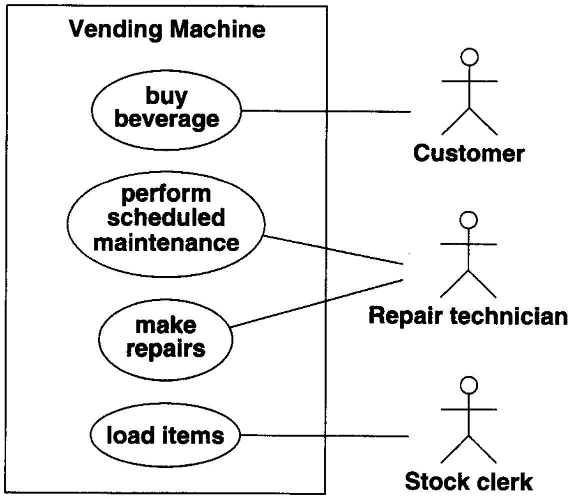

Figure 7.3 Use case diagram for a vending machine. A system involves a set of use cases and a set of actors.

denotes a use case. A "stick man" icon denotes an actor, with the name being placed below or adjacent to the icon. Solid lines connect use cases to participating actors.

In the figure, the actor *Repair technician* participates in two use cases, the others in one each. Multiple actors can participate in a use case, even though the example has only one actor per use case.

7.1.4 Guidelines for Use Case Models

Use cases identify the functionality of a system and organize it according to the perspective of users. In contrast, traditional requirements lists can include functionality that is vague to users, as well as overlook supporting functionality, such as initialization and termination. Use cases describe complete transactions and are therefore less likely to omit necessary steps. There is still a place for traditional requirements lists in describing global constraints and other nonlocalized functionality, such as mean time to failure and overall throughput, but you should capture most user interactions with use cases. The main purpose of a system is almost always found in the use cases, with requirements lists supplying additional implementation constraints. Here are some guidelines for constructing use case models.

- **First determine the system boundary.** It is impossible to identify use cases or actors if the system boundary is unclear.
- **Ensure that actors are focused.** Each actor should have a single, coherent purpose. If a real-world object embodies multiple purposes, capture them with separate actors. For example, the owner of a personal computer may install software, set up a database, and send email. These functions differ greatly in their impact on the computer system and the potential for system damage. They might be broken into three actors: *system admin-*

istrator, database administrator, and *computer user.* Remember that an actor is defined with respect to a system, not as a free-standing concept.

- **Each use case must provide value to users.** A use case should represent a complete transaction that provides value to users and should not be defined too narrowly. For example, *dial a telephone number* is not a good use case for a telephone system. It does not represent a complete transaction of value by itself; it is merely part of the use case *make telephone call.* The latter use case involves placing the call, talking, and terminating the call. By dealing with complete use cases, we focus on the purpose of the functionality provided by the system, rather than jumping into implementation decisions. The details come later. Often there is more than one way to implement desired functionality.
- **Relate use cases and actors.** Every use case should have at least one actor, and every actor should participate in at least one use case. A use case may involve several actors, and an actor may participate in several use cases.
- **Remember that use cases are informal.** It is important not to be obsessed by formalism in specifying use cases. They are not intended as a formal mechanism but as a way to identify and organize system functionality from a user-centered point of view. It is acceptable if use cases are a bit loose at first. Detail can come later as use cases are expanded and mapped into implementations.
- **Use cases can be structured.** For many applications, the individual use cases are completely distinct. For large systems, use cases can be built out of smaller fragments using relationships (see Chapter 8).

7.2 Sequence Models

The sequence model elaborates the themes of use cases. There are two kinds of sequence models: scenarios and a more structured format called sequence diagrams.

7.2.1 Scenarios

A ***scenario*** is a sequence of events that occurs during one particular execution of a system, such as for a use case. The scope of a scenario can vary; it may include all events in the system, or it may include only those events impinging on or generated by certain objects. A scenario can be the historical record of executing an actual system or a thought experiment of executing a proposed system.

A scenario can be displayed as a list of text statements, as Figure 7.4 illustrates. In this example, John Doe logs on with an online stock broker system, places an order for GE stock, and then logs off. Sometime later, after the order is executed, the securities exchange reports the results of the trade to the broker system. John Doe will see the results on the next login, but that is not part of this scenario.

The example expresses interaction at a high level. For example, the step *John Doe logs in* might require several messages between John Doe and the system. The essential purpose of the step, however, is the request to enter the system and providing the necessary identifi-

John Doe logs in.
System establishes secure communications.
System displays portfolio information.
John Doe enters a buy order for 100 shares of GE at the market price.
System verifies sufficient funds for purchase.
System displays confirmation screen with estimated cost.
John Doe confirms purchase.
System places order on securities exchange.
System displays transaction tracking number.
John Doe logs out.
System establishes insecure communication.
System displays good-bye screen.
Securities exchange reports results of trade.

Figure 7.4 Scenario for a session with an online stock broker. A scenario is a sequence of events that occurs during one particular execution of a system.

cation—the details can be shown separately. At early stages of development, you should express scenarios at a high level. At later stages, you can show the exact messages. Determining the detailed messages is part of development.

A scenario contains messages between objects as well as activities performed by objects. Each message transmits information from one object to another. For example, *John Doe logs in* transmits a message from John Doe to the broker system. The first step of writing a scenario is to identify the objects exchanging messages. Then you must determine the sender and receiver of each message, as well as the sequence of the messages. Finally, you can add activities for internal computations as scenarios are reduced to code.

7.2.2 Sequence Diagrams

A text format is convenient for writing, but it does not clearly show the sender and receiver of each message, especially if there are more than two objects. A ***sequence diagram*** shows the participants in an interaction and the sequence of messages among them. A sequence diagram shows the interaction of a system with its actors to perform all or part of a use case.

Figure 7.5 shows a sequence diagram corresponding to the previous stock broker scenario. Each actor as well as the system is represented by a vertical line called a ***lifeline*** and each message by a horizontal arrow from the sender to the receiver. Time proceeds from top to bottom, but the spacing is irrelevant; the diagram shows only the sequence of messages, not their exact timing. (Real-time systems impose time constraints on event sequences, but that requires extra notation.) Note that sequence diagrams can show concurrent signals—*stock broker system* sends messages to *customer* and *securities exchange* concurrently—and signals between participants need not alternate—*stock broker system* sends *secure communication* followed by *display portfolio*.

Each use case requires one or more sequence diagrams to describe its behavior. Each sequence diagram shows a particular behavior sequence of the use case. It is best to show a specific portion of a use case and not attempt to be too general. Although it is possible to

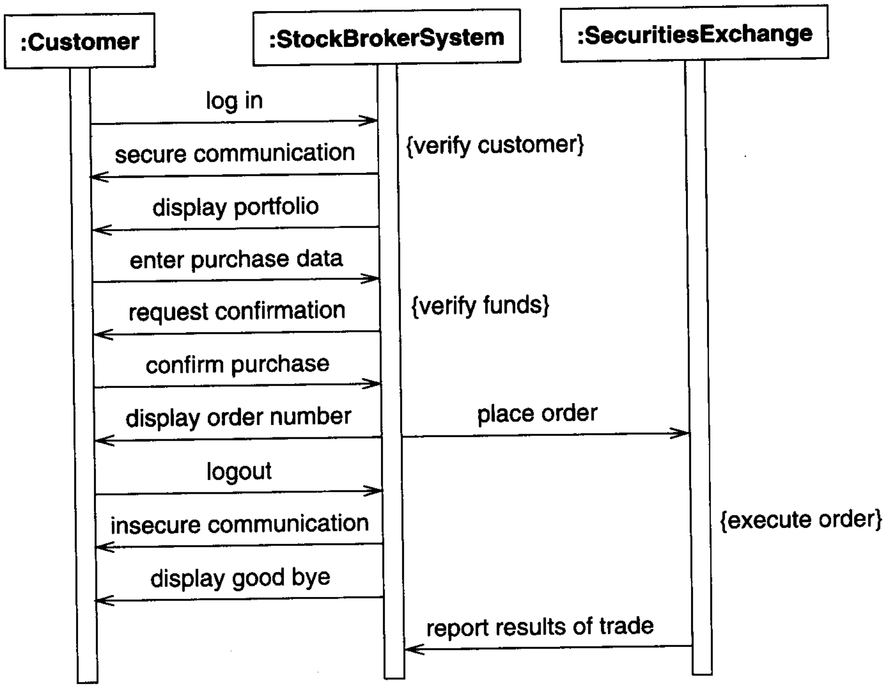

Figure 7.5 Sequence diagram for a session with an online stock broker. A sequence diagram shows the participants in an interaction and the sequence of messages among them.

show conditionals within a sequence diagram, usually it is clearer to prepare one sequence diagram for each major flow of control.

Sequence diagrams can show large-scale interactions, such as an entire session with the stock broker system, but often such interactions contain many independent tasks that can be combined in various ways. Rather than repeating information, you can draw a separate sequence diagram for each task. For example, Figure 7.6 and Figure 7.7 show an order to purchase a stock and a request for a quote on a stock. These and various other tasks (not shown) would fit within an entire stock trading session.

You should also prepare a sequence diagram for each exception condition within the use case. For example, Figure 7.8 shows a variation in which the customer does not have sufficient funds to place the order. In this example, the customer cancels the order. In another variation (not shown), the customer would reduce the number of shares purchased and the order would be accepted.

In most systems, there are an unlimited number of scenarios, so it is not possible to show them all. However, you should try to elaborate all the use cases and cover the basic kinds of behavior with sequence diagrams. For example, a stock broker system can interleave purchases, sales, and inquiries arbitrarily. It is unnecessary to show all combinations of activities, once the basic pattern is established.

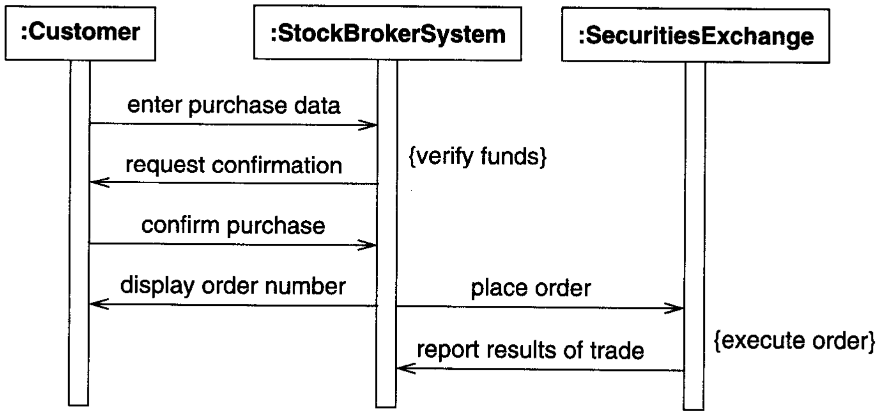

Figure 7.6 Sequence diagram for a stock purchase. Sequence diagrams can show large-scale interactions as well as smaller, constituent tasks.

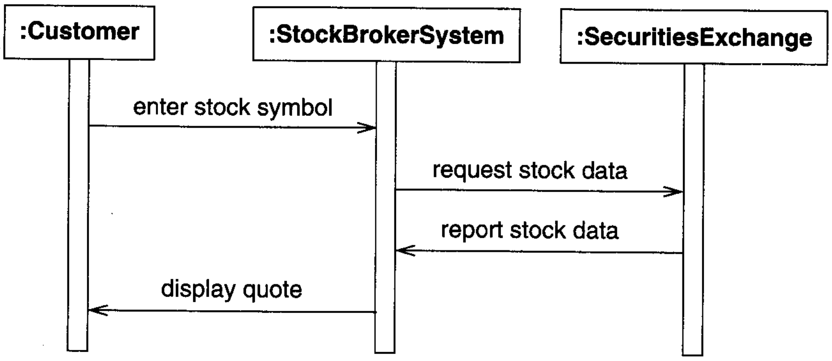

Figure 7.7 Sequence diagram for a stock quote

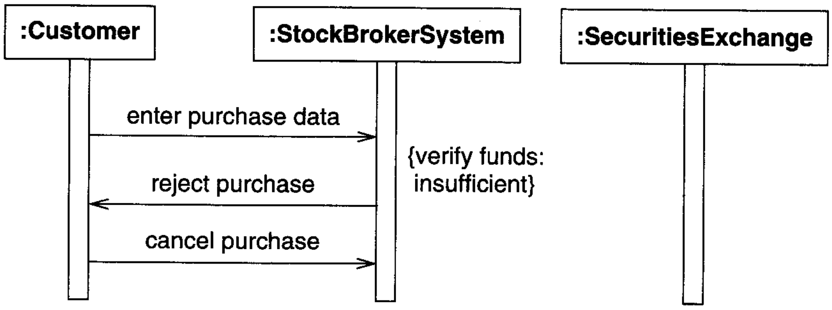

Figure 7.8 Sequence diagram for a stock purchase that fails

7.2.3 Guidelines for Sequence Models

The sequence model adds detail and elaborates the informal themes of use cases. There are two kinds of sequence models. Scenarios document a sequence of events with prose. Sequence diagrams also document the sequence of events but more clearly show the actors involved. The following guidelines will help you with sequence models.

- **Prepare at least one scenario per use case.** The steps in the scenario should be logical commands, not individual button clicks. Later, during implementation, you can specify the exact syntax of input. Start with the simplest mainline interaction—no repetitions, one main activity, and typical values for all parameters. If there are substantially different mainline interactions, write a scenario for each.
- **Abstract the scenarios into sequence diagrams.** The sequence diagrams clearly show the contribution of each actor. It is important to separate the contribution of each actor as a prelude to organizing behavior about objects.
- **Divide complex interactions.** Break large interactions into their constituent tasks and prepare a sequence diagram for each of them.
- **Prepare a sequence diagram for each error condition.** Show the system response to the error condition.

7.3 Activity Models

An ***activity diagram*** shows the sequence of steps that make up a complex process, such as an algorithm or workflow. An activity diagram shows flow of control, similar to a sequence diagram, but focuses on operations rather than on objects. Activity diagrams are most useful during the early stages of designing algorithms and workflows.

Figure 7.9 shows an activity diagram for the processing of a stock trade order that has been received by an online stock broker. The elongated ovals show activities and the arrows show their sequencing. The diamond shows a decision point and the heavy bar shows splitting or merging of concurrent threads.

The online stock broker first verifies the order against the customer's account, then executes it with the stock exchange. If the order executes successfully, the system does three things concurrently: mails trade confirmation to the customer, updates the online portfolio to reflect the results of the trade, and settles the trade with the other party by debiting the account and transferring cash or securities. When all three concurrent threads have been completed, the system merges control into a single thread and closes the order. If the order execution fails, then the system sends a failure notice to the customer and closes the order.

An activity diagram is like a traditional flowchart in that it shows the flow of control from step to step. Unlike a traditional flowchart, however, an activity diagram can show both sequential and concurrent flow of control. This distinction is important for a distributed system. Activity diagrams are often used for modeling human organizations because they involve many objects—persons and organizational units—that perform operations concurrently.

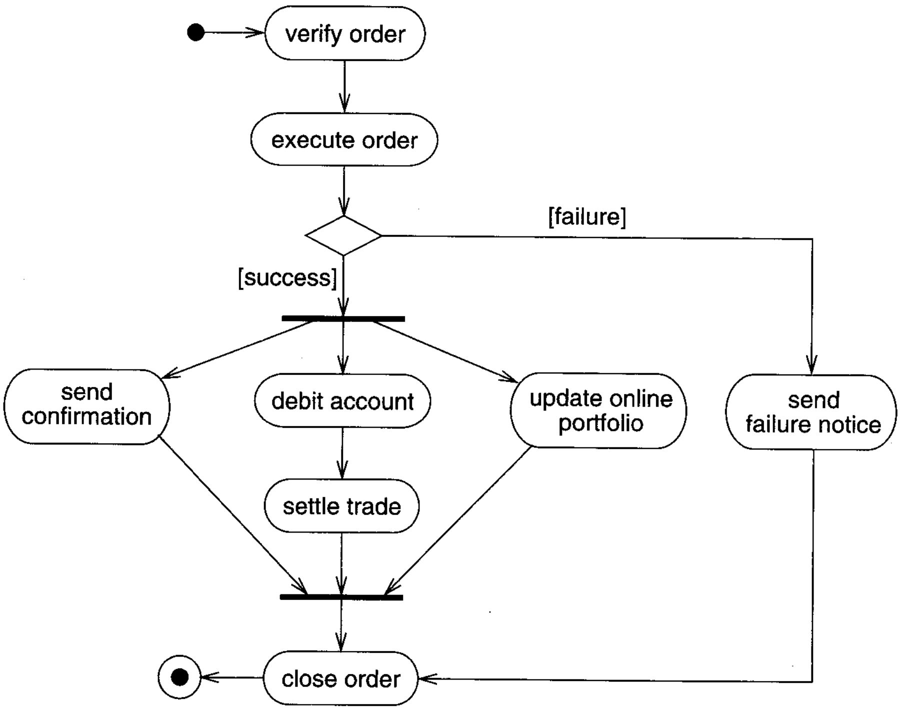

Figure 7.9 Activity diagram for stock trade processing. An activity diagram shows the sequence of steps that make up a complex process.

7.3.1 Activities

The steps of an activity diagram are operations, specifically activities from the state model. The purpose of an activity diagram is to show the steps within a complex process and the sequencing constraints among them.

Some activities run forever until an outside event interrupts them, but most activities eventually complete their work and terminate by themselves. The completion of an activity is a completion event and usually indicates that the next activity can be started. An unlabeled arrow from one activity to another in an activity diagram indicates that the first activity must complete before the second activity can begin.

An activity may be decomposed into finer activities. For example, Figure 7.10 expands the *execute order* activity of Figure 7.9. It is important that the activities on a diagram be at the same level of detail. For example, in Figure 7.9 *execute order* and *settle trade* are similar in detail; they both express a high-level operation without showing the underlying mechanisms. If one of these activities were replaced in the activity diagram by its more detailed steps, the other activities should be replaced as well to maintain balance. Alternatively, balance can be preserved by elaborating the activities in separate diagrams.

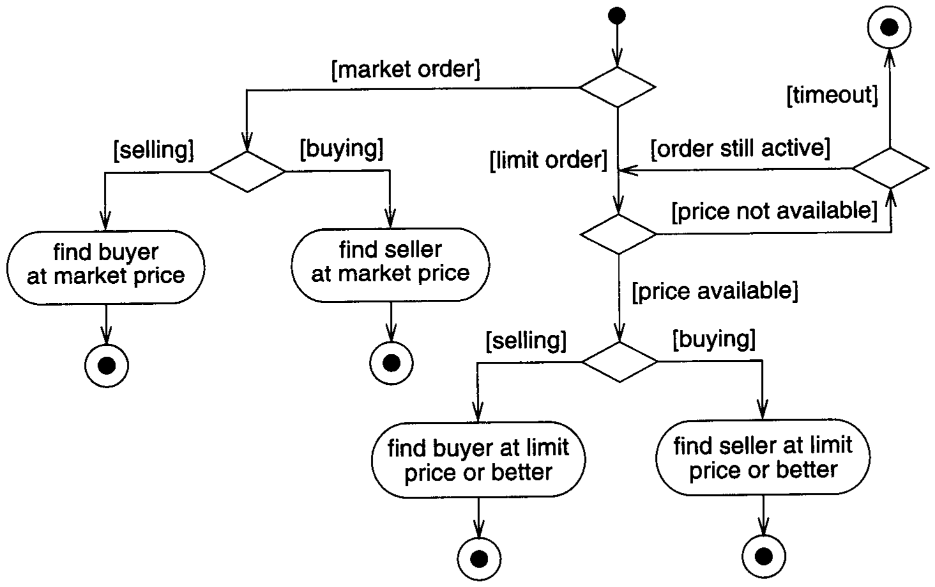

Figure 7.10 Activity diagram for *execute order*. An activity may be decomposed into finer activities.

7.3.2 Branches

If there is more than one successor to an activity, each arrow may be labeled with a condition in square brackets, for example, *[failure]*. All subsequent conditions are tested when an activity completes. If one condition is satisfied, its arrow indicates the next activity to perform. If no condition is satisfied, the diagram is badly formed and the system will hang unless it is interrupted at some higher level. To avoid this danger, you can use the *else* condition; it is satisfied in case no other condition is satisfied. If multiple conditions are satisfied, only one successor activity executes, but there is no guarantee which one it will be. Sometimes this kind of nondeterminism is desirable, but often it indicates an error, so the modeler should determine whether any overlap of conditions can occur and whether it is correct.

As a notational convenience, a diamond shows a branch into multiple successors, but it means the same thing as arrows leaving an activity symbol directly. In Figure 7.9 the diamond has one incoming arrow and two outgoing arrows, each with a condition. A particular execution chooses only one path of control.

If several arrows enter an activity, the alternate execution paths merge. Alternatively, several arrows may enter a diamond and one may exit to indicate a merge.

7.3.3 Initiation and Termination

A solid circle with an outgoing arrow shows the starting point of an activity diagram. When an activity diagram is activated, control starts at the solid circle and proceeds via the outgoing arrow toward the first activities. A bull's-eye (a solid circle surrounded by a hollow circle) shows the termination point—this symbol only has incoming arrows. When control reaches a bull's-eye, the overall activity is complete and execution of the activity diagram ends.

7.3.4 *Concurrent Activities*

Unlike traditional flow charts, organizations and computer systems can perform more than one activity at a time. The pace of activity can also change over time. For example, one activity may be followed by another activity (sequential control), then split into several concurrent activities (a fork of control), and finally be combined into a single activity (a merge of control). A fork or merge is shown by a synchronization bar—a heavy line with one or more input arrows and one or more output arrows. On a synchronization, control must be present on all of the incoming activities, and control passes to all of the outgoing activities.

Figure 7.9 illustrates both a fork and merge of control. Once an order is executed, there is a fork—several tasks need to occur and they can occur in any order. The stock trade system must send confirmation to the customer, debit the customer's account, and update the customer's online portfolio. After the three concurrent tasks complete and the trade is settled, there is a merge, and execution proceeds to the activity of closing the order.

7.3.5 *Executable Activity Diagrams*

Activity diagrams are not only useful for defining the steps in a complex process, but they can also be used to show the progression of control during execution. An ***activity token*** can be placed on an activity symbol to indicate that it is executing. When an activity completes, the token is removed and placed on the outgoing arrow. In the simplest case, the token then moves to the next activity.

If there are multiple outgoing arrows with conditions, the conditions are examined to determine the successor activity. Only one successor activity can receive the token, even if more than one condition is true. If no condition is satisfied, the activity diagram is ill formed.

Multiple tokens can arise through concurrency. If an executing activity is followed by a concurrent split of control, completion causes an increase in the number of tokens—a token is placed on each of the concurrent activities. Similarly, a merge of control causes a decrease in the number of tokens as tokens migrate from the input activities to the output activities. All the input activities must complete before the merge can actually occur.

7.3.6 *Guidelines for Activity Models*

Activity diagrams elaborate the details of computation, thus documenting the steps needed to implement an operation or a business process. In addition, activity diagrams can help developers understand complex computations by graphically displaying the progression through intermediate execution steps. Here is some advice for activity models.

- **Don't misuse activity diagrams.** Activity diagrams are intended to elaborate use case and sequence models so that a developer can study algorithms and workflow. Activity diagrams supplement the object-oriented focus of UML models and should not be used as an excuse to develop software via flowcharts.

- **Level diagrams.** Activities on a diagram should be at a consistent level of detail. Place additional detail for an activity in a separate diagram.
- **Be careful with branches and conditions.** If there are conditions, at least one must be satisfied when an activity completes—consider using an *else* condition. In undeterministic models, it is possible for multiple conditions to be satisfied—otherwise this is an error condition.
- **Be careful with concurrent activities.** Concurrency means that the activities can complete in any order and still yield an acceptable result. Before a merge can happen, all inputs must first complete.
- **Consider executable activity diagrams.** Executable activity diagrams can help developers understand their systems better. Sometimes they can even be helpful for end users who want to follow the progression of a process.

7.4 Chapter Summary

The interaction model provides a holistic view of behavior—how objects interact and exchange messages. At a high level, use cases partition the functionality of a system into discrete pieces meaningful to external actors. You can elaborate the behavior of use cases with scenarios and sequence diagrams. Sequence diagrams clearly show the objects in an interaction and the messages among them. Activity diagrams specify the details of a computation.

The class, state, and interaction models all involve the same concepts, namely data, sequencing, and operations, but each model focuses on a particular aspect and leaves the other aspects uninterpreted. All three models are necessary for a full understanding of a problem, although the balance of importance among the models varies according to the kind of application. The three models come together in the implementation of methods, which involve data (target object, arguments, and variables), control (sequencing constructs), and interactions (messages, calls, and sequences).

activity	concurrency	scenario	use case
activity diagram	interaction model	sequence diagram	use case diagram
activity token	lifeline	system boundary	
actor	message	thread	

Figure 7.11 Key concepts for Chapter 7

Bibliographic Notes

Jacobson first introduced use cases [Jacobson-92]. The first edition of this book included scenarios and event trace diagrams. The latter are equivalent to simple sequence diagrams.