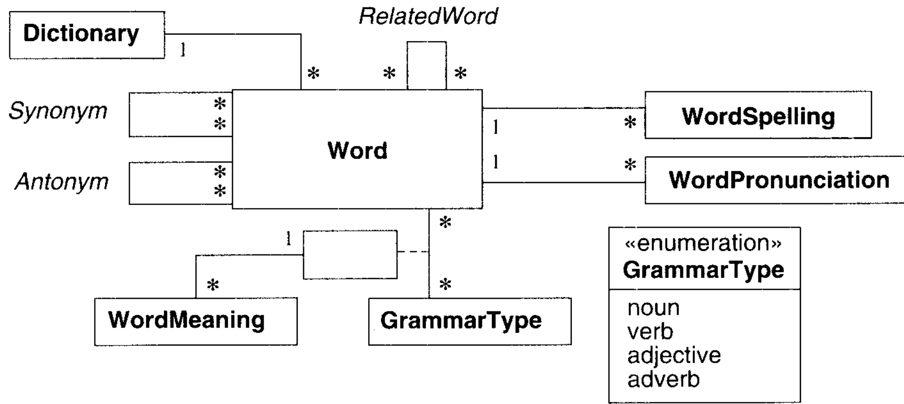


Figure E4.8 Partial model for words in a dictionary

5

State Modeling

You can best understand a system by first examining its static structure—that is, the structure of its objects and their relationships to each other at a single moment in time (the class model). Then you should examine changes to the objects and their relationships over time (the state model). The state model describes the sequences of operations that occur in response to external stimuli, as opposed to what the operations do, what they operate on, or how they are implemented.

The state model consists of multiple state diagrams, one for each class with temporal behavior that is important to an application. The state diagram is a standard computer science concept (a graphical representation of finite state machines) that relates events and states. Events represent external stimuli and states represent values of objects. You should master the material in this chapter before proceeding in the book.

5.1 Events

An ***event*** is an occurrence at a point in time, such as *user depresses left button* or *flight 123 departs from Chicago*. Events often correspond to verbs in the past tense (*power turned on*, *alarm set*) or to the onset of some condition (*paper tray becomes empty*, *temperature becomes lower than freezing*). By definition, an event happens instantaneously with regard to the time scale of an application. Of course, nothing is really instantaneous; an event is simply an occurrence that an application considers atomic and fleeting. The time at which an event occurs is an implicit attribute of the event. Temporal phenomena that occur over an interval of time are properly modeled with a state.

One event may logically precede or follow another, or the two events may be unrelated. Flight 123 must depart Chicago before it can arrive in San Francisco; the two events are causally related. Flight 123 may depart before or after flight 456 departs Rome; the two events are causally unrelated. Two events that are causally unrelated are said to be ***concurrent;*** they

have no effect on each other. If the communications delay between two locations exceeds the difference in event times, then the events must be concurrent because they cannot influence each other. Even if the physical locations of two events are not distant, we consider the events concurrent if they do not affect each other. In modeling a system we do not try to establish an ordering between concurrent events because they can occur in any order.

Events include error conditions as well as normal occurrences. For example, *motor jammed, transaction aborted,* and *timeout* are typical error events. There is nothing different about an error event; only our interpretation makes it an "error."

The term *event* is often used ambiguously. Sometimes it refers to an instance, at other times to a class. In practice, this ambiguity is usually not a problem and the precise meaning is apparent from the context. If necessary, you can say *event occurrence* or *event type* to be precise.

There are several kinds of events. The most common are the signal event, the change event, and the time event.

5.1.1 Signal Event

A ***signal*** is an explicit one-way transmission of information from one object to another. It is different from a subroutine call that returns a value. An object sending a signal to another object may expect a reply, but the reply is a separate signal under the control of the second object, which may or may not choose to send it.

A ***signal event*** is the event of sending or receiving a signal. Usually we are more concerned about the receipt of a signal, because it causes effects in the receiving object. Note the difference between *signal* and *signal event*—a signal is a message between objects while a signal event is an occurrence in time.

Every signal transmission is a unique occurrence, but we group them into *signal classes* and give each signal class a name to indicate common structure and behavior. For example, *UA flight 123 departs from Chicago on January 10, 1991* is an instance of signal class *FlightDeparture*. Some signals are simple occurrences, but most signal classes have attributes indicating the values they convey. For example, as Figure 5.1 shows, *FlightDeparture* has attributes *airline, flightNumber, city,* and *date*. The UML notation is the keyword *signal* in guillemets («») above the signal class name in the top section of a box. The second section lists the signal attributes.

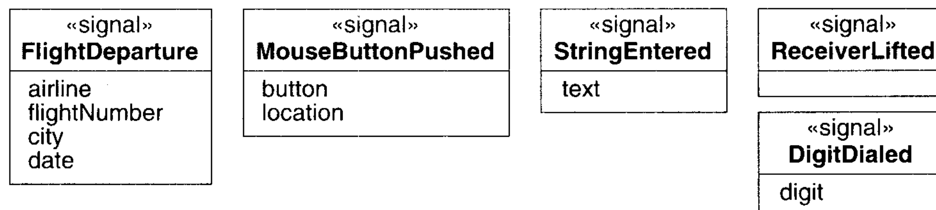


Figure 5.1 Signal classes and attributes. A signal is an explicit one-way transmission of information from one object to another.

5.1.2 Change Event

A ***change event*** is an event that is caused by the satisfaction of a boolean expression. The intent of a change event is that the expression is continually tested—whenever the expression changes from false to true, the event happens. Of course, an implementation would not *continuously* check a change event, but it must check often enough so that it seems continuous from an application perspective.

The UML notation for a change event is the keyword *when* followed by a parenthesized boolean expression. Figure 5.2 shows several examples of change events.

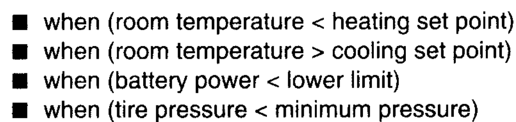


Figure 5.2 **Change events**. A change event is an event that is caused by the satisfaction of a boolean expression.

5.1.3 Time Event

A ***time event*** is an event caused by the occurrence of an absolute time or the elapse of a time interval. As Figure 5.3 shows, the UML notation for an absolute time is the keyword *when* followed by a parenthesized expression involving time. The notation for a time interval is the keyword *after* followed by a parenthesized expression that evaluates to a time duration.

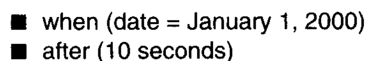


Figure 5.3 **Time events**. A time event is an event caused by the occurrence of an absolute time or the elapse of a time interval.

5.2 States

A ***state*** is an abstraction of the values and links of an object. Sets of values and links are grouped together into a state according to the gross behavior of objects. For example, the state of a bank is either solvent or insolvent, depending on whether its assets exceed its liabilities. States often correspond to verbs with a suffix of "ing" (*Waiting*, *Dialing*) or the duration of some condition (*Powered*, *BelowFreezing*).

Figure 5.4 shows the UML notation for a state—a rounded box containing an optional state name. Our convention is to list the state name in boldface, center the name near the top of the box, and capitalize the first letter.

Figure 5.4 States. A state is an abstraction of the values and links of an object.

In defining states, we ignore attributes that do not affect the behavior of the object, and lump together in a single state all combinations of values and links with the same response to events. Of course, every attribute has some effect on behavior or it would be meaningless, but often some attributes do not affect the sequence of control and you can regard them as simple parameter values within a state. Recall that the purpose of modeling is to focus on qualities that are relevant to the solution of an application problem and abstract away those that are irrelevant. The three UML models (class, state, and interaction) present different views of a system for which the particular choice of attributes and values are not equally important. For example, except for leading 0s and 1s, the exact digits dialed do not affect the control of the phone line, so we can summarize them all with state *Dialing* and track the phone number as a parameter. Sometimes, all possible values of an attribute are important, but usually only when the number of possible values is small.

The objects in a class have a finite number of possible states—one or possibly some larger number. Each object can only be in one state at a time. Objects may parade through one or more states during their lifetime. At a given moment of time, the various objects for a class can exist in a multitude of states.

A state specifies the response of an object to input events. All events are ignored in a state, except those for which behavior is explicitly prescribed. The response may include the invocation of behavior or a change of state. For example, if a digit is dialed in state *Dial tone*, the phone line drops the dial tone and enters state *Dialing*; if the receiver is replaced in state *Dial tone*, the phone line goes dead and enters state *Idle*.

There is a certain symmetry between events and states as Figure 5.5 illustrates. Events represent points in time; states represent intervals of time. A state corresponds to the interval between two events received by an object. For example, after the receiver is lifted and before the first digit is dialed, the phone line is in state *Dial tone*. The state of an object depends on past events, which in most cases are eventually hidden by subsequent events. For example, events that happened before the phone is hung up do not affect future behavior; the *Idle* state "forgets" events received prior to the receipt of the *hang up* signal.

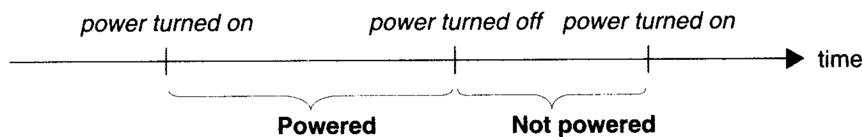


Figure 5.5 Event vs. state. Events represent points in time; states represent intervals of time.

Both events and states depend on the level of abstraction. For example, a travel agent planning an itinerary would treat each segment of a journey as a single event; a flight status

board in an airport would distinguish departures and arrivals; an air traffic control system would break each flight into many geographical legs.

You can characterize a state in various ways, as Figure 5.6 shows for the state *Alarm ringing* on a watch. The state has a suggestive name and a natural-language description of its purpose. The event sequence that leads to the state consists of setting the alarm, doing anything that doesn't clear the alarm, and then having the target time occur. A declarative condition for the state is given in terms of parameters, such as *current* and *target time*; the alarm stops ringing after 20 seconds. Finally, a stimulus-response table shows the effect of events *current time* and *button pushed*, including the response that occurs and the next state. The different descriptions of a state may overlap.

State: *AlarmRinging*

Description: alarm on watch is ringing to indicate target time

Event sequence that produces the state:

setAlarm (*targetTime*)
any sequence not including *clearAlarm*
when (*currentTime* = *targetTime*)

Condition that characterizes the state:

alarm = on, alarm set to *targetTime*, *targetTime* ≤ *currentTime* ≤ *targetTime* + 20 seconds, and no button has been pushed since *targetTime*

Events accepted in the state:

event	response	next state
when (*currentTime* = *targetTime* + 20)	*resetAlarm*	*normal*
buttonPushed (any button)	*resetAlarm*	*normal*

Figure 5.6 Various characterizations of a state. A state specifies the response of an object to input events.

Can links have state? In as much as they can be considered objects, links can have state. As a practical matter, it is generally sufficient to associate state only with objects.

5.3 Transitions and Conditions

A ***transition*** is an instantaneous change from one state to another. For example, when a called phone is answered, the phone line transitions from the *Ringing* state to the *Connected* state. The transition is said to ***fire*** upon the change from the source state to the target state. The origin and target of a transition usually are different states, but may be the same. A transition fires when its event occurs (unless an optional guard condition causes the event to be ignored). The choice of next state depends on both the original state and the event received.

An event may cause multiple objects to transition; from a conceptual point of view such transitions occur concurrently.

A ***guard condition*** is a boolean expression that must be true in order for a transition to occur. For example, a traffic light at an intersection may change only if a road has cars waiting. A guarded transition fires when its event occurs, but only if the guard condition is true. For example, "when you go out in the morning (event), if the temperature is below freezing (condition), then put on your gloves (next state)." A guard condition is checked only once, at the time the event occurs, and the transition fires if the condition is true. If the condition becomes true later, the transition does not then fire. Note that a guard condition is different from a change event—a guard condition is checked only once while a change event is, in effect, checked continuously.

Figure 5.7 shows guarded transitions for traffic lights at an intersection. One pair of electric eyes checks the north-south left turn lanes; another pair checks the east-west turn lanes. If no car is in the north-south and/or east-west turn lanes, then the traffic light control logic is smart enough to skip the left turn portion of the cycle.

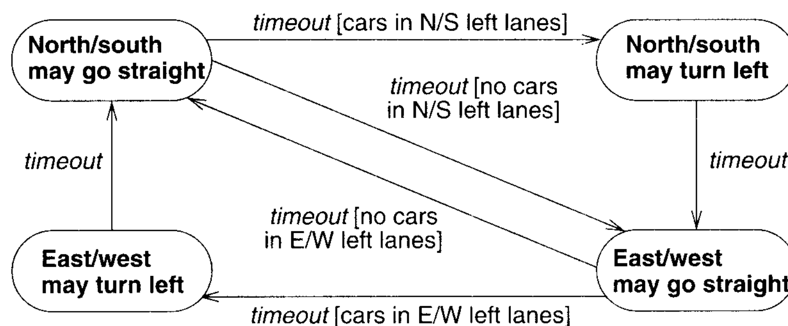


Figure 5.7 Guarded transitions. A transition is an instantaneous change from one state to another. A guard condition is a boolean expression that must be true in order for a transition to occur.

The UML notation for a transition is a line from the origin state to the target state. An arrowhead points to the target state. The line may consist of several line segments. An event may label the transition and be followed by an optional guard condition in square brackets. By convention, we usually confine line segments to a rectilinear grid. We italicize the event name and show the condition in normal font.

5.4 State Diagrams

A ***state diagram*** is a graph whose nodes are states and whose directed arcs are transitions between states. A state diagram specifies the state sequences caused by event sequences. State names must be unique within the scope of a state diagram. All objects in a class execute the state diagram for that class, which models their common behavior. You can implement

state diagrams by direct interpretation or by converting the semantics into equivalent programming code.

The ***state model*** consists of multiple state diagrams, one state diagram for each class with important temporal behavior. The state diagrams must match on their interfaces—events and guard conditions. The individual state diagrams interact by passing events and through the side effects of guard conditions. Some events and guard conditions appear in a single state diagram; others appear in multiple state diagrams for the purpose of coordination. This chapter covers only individual state diagrams; Chapter 6 discusses state models of interacting diagrams.

A class with more than one state has important temporal behavior. Similarly, a class is temporally important if it has a single state with multiple responses to events. You can represent state diagrams with a single state in a simple nongraphical form—a stimulus–response table listing events and guard conditions and the ensuing behavior.

5.4.1 Sample State Diagram

Figure 5.8 shows a state diagram for a telephone line. The diagram concerns a phone line and not the caller nor callee. The diagram contains sequences associated with normal calls as well as some abnormal sequences, such as timing out while dialing or getting busy lines. The UML notation for a state diagram is a rectangle with its name in a small pentagonal tag in the upper left corner. The constituent states and transitions lie within the rectangle.

At the start of a call, the telephone line is idle. When the phone is removed from the hook, it emits a dial tone and can accept the dialing of digits. Upon entry of a valid number, the phone system tries to connect the call and route it to the proper destination. The connection can fail if the number or trunk are busy. If the connection is successful, the called phone begins ringing. If the called party answers the phone, a conversation can occur. When the called party hangs up, the phone disconnects and reverts to idle when put on hook again.

Note that the receipt of the signal *onHook* causes a transition from any state to *Idle* (the bundle of transitions leading to *Idle*). Chapter 6 will show a more general notation that represents events applicable to groups of states with a single transition.

States do not totally define all values of an object. For example, state *Dialing* includes all sequences of incomplete phone numbers. It is not necessary to distinguish between different numbers as separate states, since they all have the same behavior, but the actual number dialed must of course be saved as an attribute.

If more than one transition leaves a state, then the first event to occur causes the corresponding transition to fire. If an event occurs and no transition matches it, then the event is ignored. If more than one transition matches an event, only one transition will fire, but the choice is nondeterministic.

5.4.2 One-shot State Diagrams

State diagrams can represent continuous loops or one-shot life cycles. The diagram for the phone line is a continuous loop. In describing ordinary usage of the phone, we do not know or care how the loop is started. (If we were describing installation of new lines, the initial state would be important.)

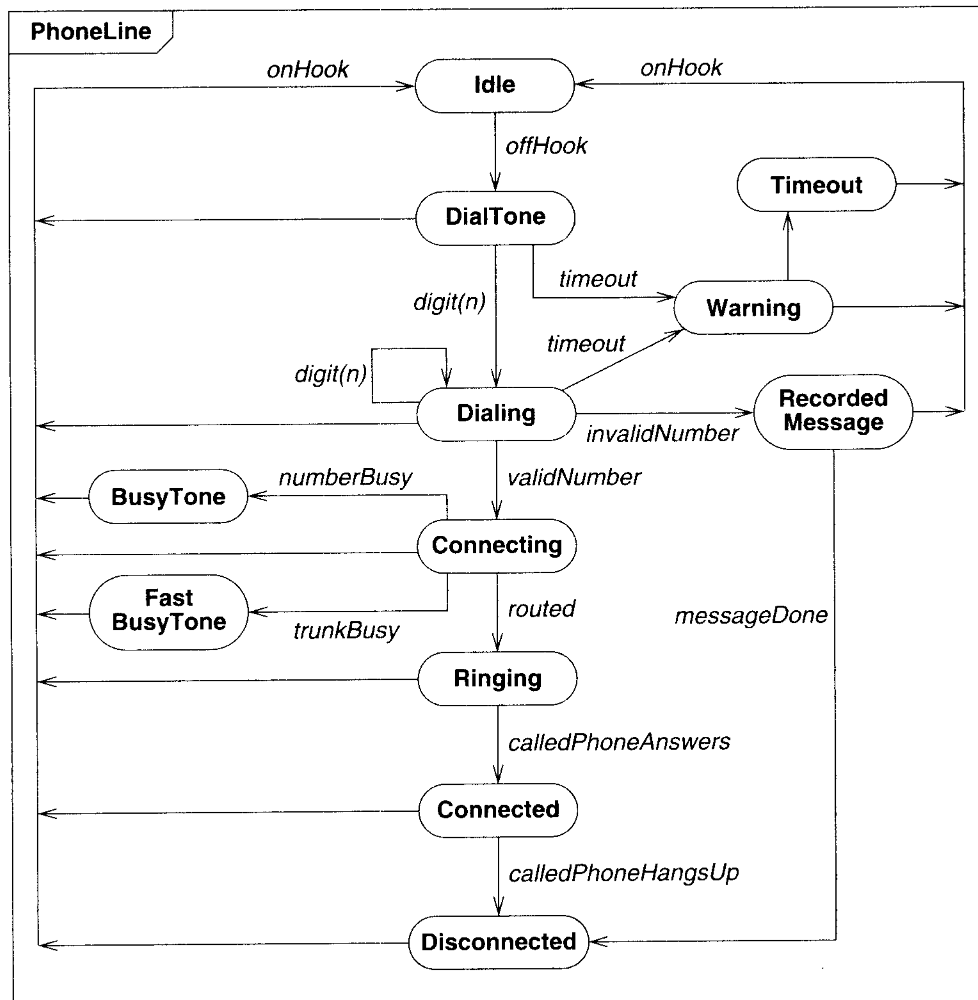


Figure 5.8 State diagram for a telephone line. A state diagram specifies the state sequences caused by event sequences.

One-shot state diagrams represent objects with finite lives and have initial and final states. The initial state is entered on creation of an object; entry of the final state implies destruction of the object. Figure 5.9 shows a simplified life cycle of a chess game with a default initial state (solid circle) and a default final state (bull's eye).

As an alternate notation, you can indicate initial and final states via entry and exit points. In Figure 5.10 the *start* entry point leads to white's first turn, and the chess game eventually ends with one of three possible outcomes. Entry points (hollow circles) and exit points (circles enclosing an "x") appear on the state diagram's perimeter and may be named.

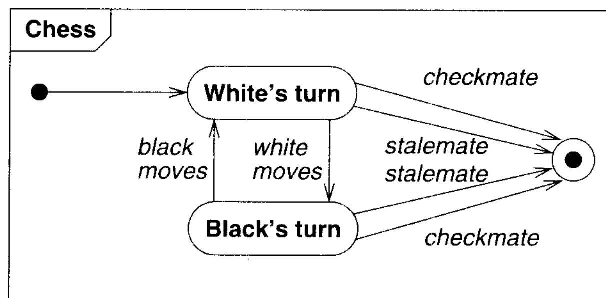


Figure 5.9 State diagram for chess game. One-shot diagrams represent objects with finite lives.

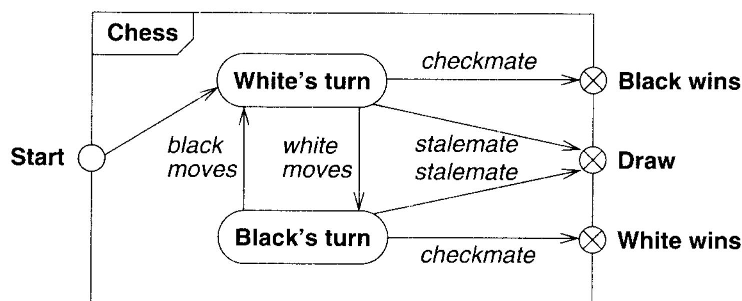


Figure 5.10 State diagram for chess game. You can also show one-shot diagrams by using entry and exit points.

5.4.3 Summary of Basic State Diagram Notation

Figure 5.11 summarizes the basic UML syntax for state diagrams.

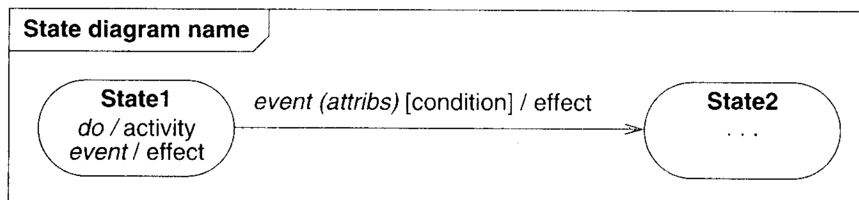


Figure 5.11 Summary of basic notation for state diagrams.

- **State.** Drawn as a rounded box containing an optional name. A special notation is available for initial states (a solid circle) and final states (a bull's-eye or encircled "x").

- **Transition**. Drawn as a line from the origin state to the target state. An arrowhead points to the target state. The line may consist of several line segments.
- **Event**. A signal event is shown as a label on a transition and may be followed by parenthesized attributes. A change event is shown with the keyword *when* followed by a parenthesized boolean expression. A time event is shown with the keyword *when* followed by a parenthesized expression involving time or the keyword *after* followed by a parenthesized expression that evaluates to a time duration.
- **State diagram**. Enclosed in a rectangular frame with the diagram name in a small pentagonal tag in the upper left corner.
- **Guard condition**. Optionally listed in square brackets after an event.
- **Effects** (to be explained in next section). Can be attached to a transition or state and are listed after a slash ("/"). Multiple effects are separated with a comma and are performed concurrently. (You can create intervening states if you want multiple effects to be performed in sequence.)

We also recommend some style conventions. We list the state name in boldface with the first letter capitalized. We italicize event names with the initial letter in lower case. Guard conditions and effects are in normal font and also have the initial letter in lower case. We try to confine transition line segments to a rectilinear grid.

5.5 State Diagram Behavior

State diagrams would be of little use if they just described events. A full description of an object must specify what the object does in response to events.

5.5.1 Activity Effects

An ***effect*** is a reference to a behavior that is executed in response to an event. An ***activity*** is the actual behavior that can be invoked by any number of effects. For example, *disconnectPhoneLine* might be an activity that is executed in response to an *onHook* event for Figure 5.8. An activity may be performed upon a transition, upon the entry to or exit from a state, or upon some other event within a state.

Activities can also represent internal control operations, such as setting attributes or generating other events. Such activities have no real-world counterparts but instead are mechanisms for structuring control within an implementation. For example, a program might increment an internal counter every time a particular event occurs.

The notation for an activity is a slash ("/") and the name (or description) of the activity, following the event that causes it. The keyword *do* is reserved for indicating an ongoing activity (to be explained) and may not be used as an event name. Figure 5.12 shows the state diagram for a pop-up menu on a workstation.When the right button is depressed, the menu is displayed; when the right button is released, the menu is erased. While the menu is visible, the highlighted menu item is updated whenever the cursor moves.

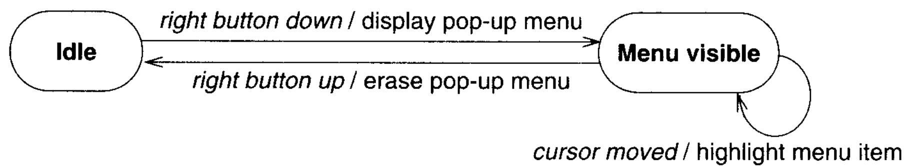


Figure 5.12 Activities for pop-up menu. An activity is behavior that can be executed in response to an event.

5.5.2 Do-Activities

A ***do-activity*** is an activity that continues for an extended time. By definition, a do-activity can only occur within a state and cannot be attached to a transition. For example, the warning light may flash during the *Paper jam* state for a copy machine (Figure 5.13). Do-activities include continuous operations, such as displaying a picture on a television screen, as well as sequential operations that terminate by themselves after an interval of time, such as closing a valve.

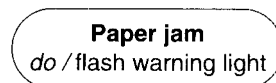


Figure 5.13 Do-activity for a copy machine. A do-activity is an activity that continues for an extended time.

The notation "*do /*" denotes a do-activity that may be performed for all or part of the duration that an object is in a state. A do-activity may be interrupted by an event that is received during its execution; such an event may or may not cause a transition out of the state containing the do-activity. For example, a robot moving a part may encounter resistance, causing it to cease moving.

5.5.3 Entry and Exit Activities

As an alternative to showing activities on transitions, you can bind activities to entry or to exit from a state. There is no difference in expressive power between the two notations, but frequently all transitions into a state perform the same activity, in which case it is more concise to attach the activity to the state.

For example, Figure 5.14 shows the control of a garage door opener. The user generates *depress* events with a pushbutton to open and close the door. Each event reverses the direction of the door, but for safety the door must open fully before it can be closed. The control generates *motor up* and *motor down* activities for the motor. The motor generates *door open* and *door closed* events when the motion has been completed. Both transitions entering state *Opening* cause the door to open.

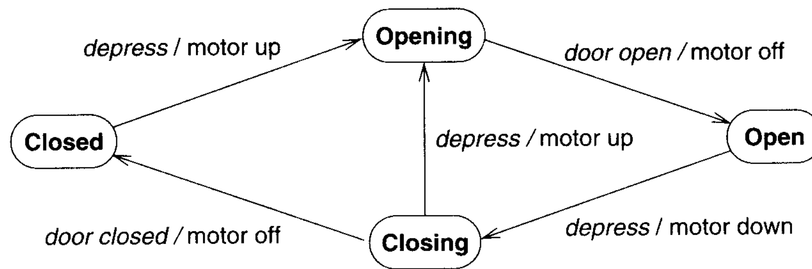


Figure 5.14 Activities on transitions. An activity may be bound to an event that causes a transition.

Figure 5.15 shows the same model using activities on entry to states. An entry activity is shown inside the state box following the keyword *entry* and a "/" character. Whenever the state is entered, by any incoming transition, the entry activity is performed. An entry activity is equivalent to attaching the activity to every incoming transition. If an incoming transition already has an activity, its activity is performed first.

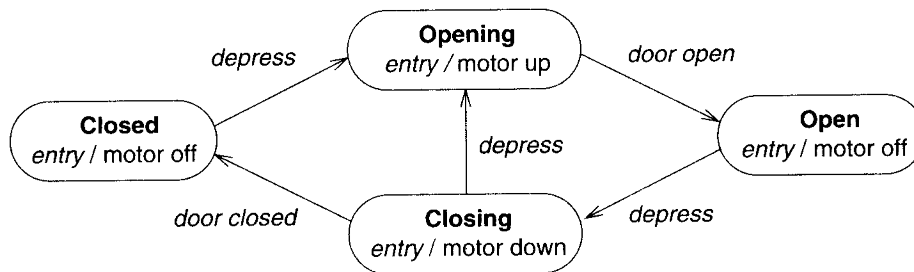


Figure 5.15 Activities on entry to states. An activity may also be bound to an event that occurs within a state.

Exit activities are less common than entry activities, but they are occasionally useful. An exit activity is shown inside the state box following the keyword *exit* and a "/" character. Whenever the state is exited, by any outgoing transition, the exit activity is performed first.

If a state has multiple activities, they are performed in the following order: activities on the incoming transition, entry activities, do-activities, exit activities, activities on the outgoing transition. Events that cause transitions out of the state can interrupt do-activities. If a do-activity is interrupted, the exit activity is still performed.

In general, any event can occur within a state and cause an activity to be performed. *Entry* and *exit* are only two examples of events that can occur. As Figure 5.16 shows, there is a difference between an event within a state and a self-transition; only the self-transition causes the entry and exit activities to be executed.

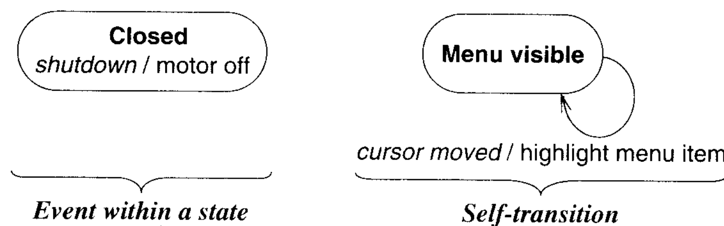


Figure 5.16 Event within a state vs. self-transition. A self-transition causes entry and exit activities to be executed. An event within a state does not.

5.5.4 Completion Transition

Often the sole purpose of a state is to perform a sequential activity. When the activity is completed, a transition to another state fires. An arrow without an event name indicates an automatic transition that fires when the activity associated with the source state is completed. Such unlabeled transitions are called ***completion transitions*** because they are triggered by the completion of activity in the source state.

A guard condition is tested only once, when the event occurs. If a state has one or more completion transitions, but none of the guard conditions are satisfied, then the state remains active and may become "stuck"—the completion event does not occur a second time, therefore no completion transition will fire later to change the state. If a state has completion transitions leaving it, normally the guard conditions should cover every possible outcome. You can use the special condition *else* to apply if all the other conditions are false. Do not use a guard condition on a completion transition to model waiting for a change of value. Instead model the waiting as a change event.

5.5.5 Sending Signals

An object can perform the activity of sending a signal to another object. A system of objects interacts by exchanging signals.

The activity "send *target.S(attributes)*" sends signal *S* with the given attributes to the target object or objects. For example, the phone line sends a *connect(phone number)* signal to the switcher when a complete phone number has been dialed. A signal can be directed at a set of objects or a single object. If the target is a set of objects, each of them receives a separate copy of the signal concurrently, and each of them independently processes the signal and determines whether to fire a transition. If the signal is always directed to the same object, the diagram can omit the target (but it must be supplied eventually in an implementation, of course).

If an object can receive signals from more than one object, the order in which concurrent signals are received may affect the final state; this is called a ***race condition***. For example, in Figure 5.15 the door may or may not remain open if the button is pressed at about the time the door becomes fully open. A race condition is not necessarily a design error, but concur-

rent systems frequently contain unwanted race conditions that must be avoided by careful design. A requirement of two signals being received simultaneously is never a meaningful condition in the real world, as slight variations in transmission speed are inherent in any distributed system.

5.5.6 Sample State Diagram with Activities

Figure 5.17 adds activities to the state diagram from Figure 5.8.

5.6 Practical Tips

The precise content of all models depends on application needs. The chapter has already mentioned the following practical tips, and we summarize them here for your convenience.

- **Abstracting values into states**. Consider only *relevant* attributes when defining a state. State diagrams need not use all attributes shown in a class model. (Section 5.2)
- **Parameters**. Parameterize events for incidental data that do not affect the flow of control. (Section 5.2)
- **Granularity of events and states**. Consider application needs when deciding on the granularity of events and states. (Section 5.2)
- **When to use state diagrams**. Construct state diagrams only for classes with meaningful temporal behavior. A class has important temporal behavior if it responds differently to various events or has more than one state. Not all classes require a state diagram. (Section 5.4)
- **Entry and exit activities**. When a state has multiple incoming transitions, and all transitions cause the same activity to occur, use an *entry* activity within the state rather than repeatedly listing the activity on transition arcs. Do likewise for *exit* activities. (Section 5.5.3)
- **Guard conditions**. Be careful with guard conditions so that an object does not become "stuck" in a state. (Section 5.5.4)
- **Race conditions**. Beware of unwanted race conditions in state diagrams. Race conditions may occur when a state can accept events from more than one object. (Section 5.5.5)

5.7 Chapter Summary

Event and state are the two elementary concepts in state modeling. An event is an occurrence at a point in time. A state is an abstraction of the values and links of an object. Events represent points in time; states represent intervals of time. An object may respond to certain events when it is in certain states. All events are ignored in a state, except those for which behavior is explicitly prescribed. The same event can have different effects (or no effect) in different states.

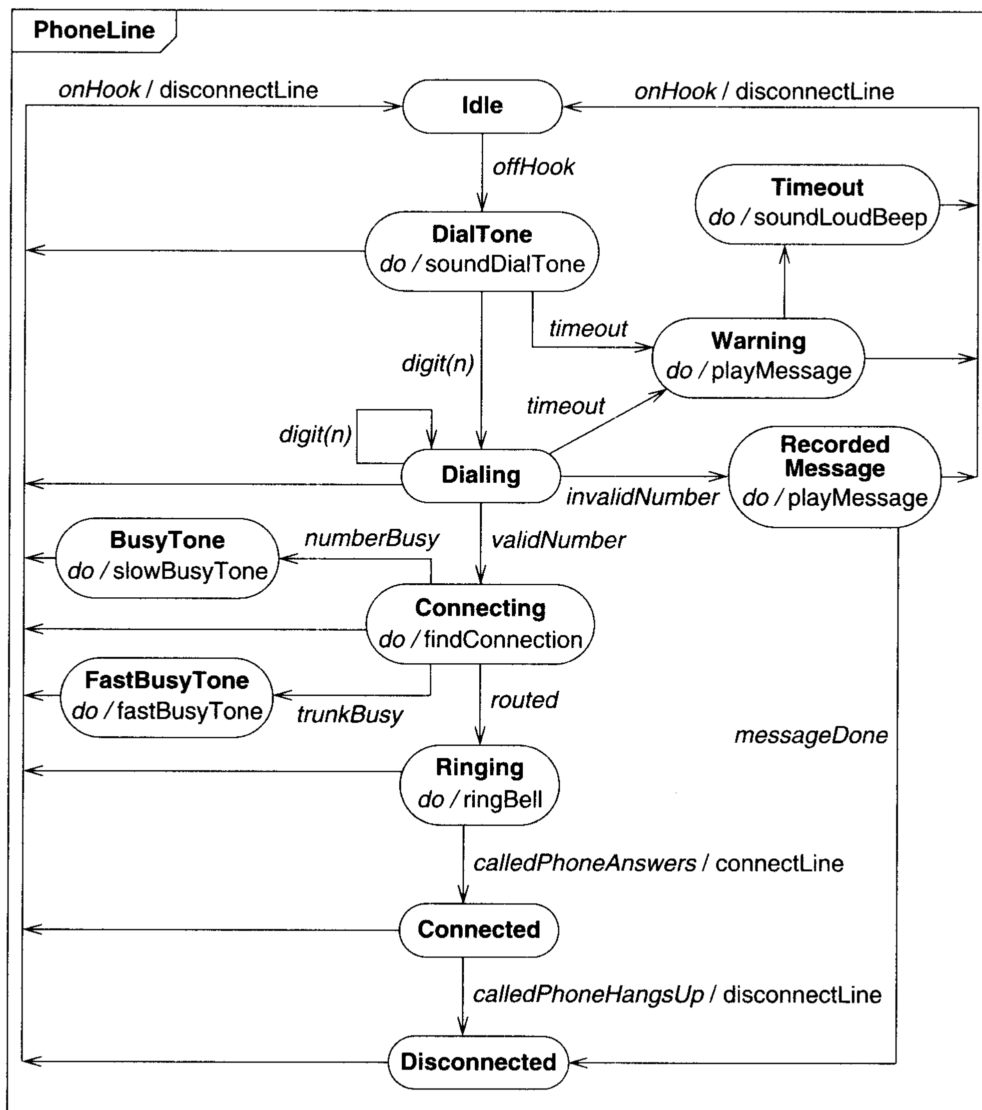


Figure 5.17 State diagram for phone line with activities. State diagrams let you express what objects do in response to events.

There are several kinds of events, such as a signal event, a change event, and a time event. A signal event is the sending or receipt of information communicated among objects. A change event is an event that is caused by the satisfaction of a boolean expression. A time event is an event caused by the occurrence of an absolute time or the elapse of a relative time.

A transition is an instantaneous change from one state to another and is caused by the occurrence of an event. An optional guard condition can cause the event to be ignored. A guard condition is a boolean expression that must be true in order for a transition to occur.

An effect is a reference to a behavior that is executed by objects in response to an event. An activity is the actual behavior that can be invoked by any number of effects. An activity may be performed upon a transition or upon an event within a state. A do-activity is an interruptible behavior that continues for an extended time. Consequently, a do-activity can occur only within a state and cannot be attached to a transition.

A state diagram is a graph whose nodes are states and whose directed arcs are transitions between states. A state diagram specifies the possible states, what transitions are allowed between states, what events cause the transitions to occur, and what behavior is executed in response to events. A state diagram describes the common behavior for the objects in a class; as each object has its own values and links, so too each object has its own state or position in the state diagram. The state model consists of multiple state diagrams, one state diagram for each class with important temporal behavior. The state diagrams must match on their interfaces—events and guard conditions.

activity	do-activity	race condition	state model
change event	effect	signal	time event
completion transition	event	signal event	transition
concurrency	fire (a transition)	state	
control	guard condition	state diagram	

Figure 5.18 Key concepts for Chapter 5

Bibliographic Notes

[Wieringa-98] has a thorough comparison of various ways for specifying software, including specification of the dynamic behavior of systems.

Finite state machines are a basic computer science concept and are described in any standard text on automata theory, such as [Hopcroft-01]. They are often described as recognizers or generators of formal languages. Basic finite state machines have limited expressive power. They have been extended with local variables and recursion as Augmented Transition Networks [Woods-70] and Recursive Transition Networks. These extensions expand the range of formal languages they can express but do little to address the combinatorial explosion that makes them unwieldy for practical control problems. (Chapter 6 addresses this.)

Traditional finite automata have been approached from a synchronous viewpoint. Petri nets [Reisig-92] formalize concurrency and synchronization of systems with distributed activity without resort to any notion of global time. Although they succeed well as an abstract conceptual model, they are too low-level and inexpressive to be useful for specifying large systems.

The need to specify interactive user interfaces has created several techniques for specifying control. This work is directed toward finding notations that clearly express powerful kinds of interactions while also being easily implementable. See [Green-86] for a comparison of some of these techniques.

The first edition of this book distinguished between *actions* (instantaneous behavior) and *activities* (lengthy behavior). UML2 has redefined both of these terms, and we have modified our explanation accordingly. UML2 now defines an activity as a specification of executable behavior and an action as a predefined primitive activity. In effect, the new definition of activity in UML2 subsumes the action and activity of the old book.

References

[Green-86] Mark Green. A survey of three dialogue models. *ACM Transactions on Graphics 5,* 3 (July 1986), 244–275.

[Hopcroft-01] J.E. Hopcroft, Rejeev Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation., Second Edition,* Boston: Addison-Wesley, 2001.

[Reisig-92]. Wolfgang Reisig. *A Primer in Petri Net Design.* New York: Springer-Verlag, 1992.

[Wieringa-98] Roel Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys 30,* 4 (December 1998), 459–527.

[Woods-70] W.A. Woods. Transition network grammars for natural language analysis. *Communications of ACM 13,* 10 (October 1970), 591–606.

Exercises

5.1 (6) An extension ladder has a rope, pulley, and latch for raising, lowering, and locking the extension. When the latch is locked, the extension is mechanically supported and you may safely climb the ladder. To release the latch, you raise the extension slightly with the rope. You may then freely raise or lower the extension. The latch produces a clacking sound as it passes over rungs of the ladder. The latch may be reengaged while raising the extension by reversing direction just as the latch is passing a rung. Prepare a state diagram of an extension ladder.

5.2 (4) A simple digital watch has a display and two buttons to set it, the A button and the B button. The watch has two modes of operation, display time and set time. In the display time mode, the watch displays hours and minutes, separated by a flashing colon.

The set time mode has two submodes, set hours and set minutes. The A button selects modes. Each time it is pressed, the mode advances in the sequence: display, set hours, set minutes, display, etc. Within the submodes, the B button advances the hours or minutes once each time it is pressed. Buttons must be released before they can generate another event. Prepare a state diagram of the watch.

5.3 (4) Figure E5.1 is a partially completed and simplified state diagram for the control of a telephone answering machine. The machine detects an incoming call on the first ring and answers the call with a prerecorded announcement. When the announcement is complete, the machine records the caller's message. When the caller hangs up, the machine hangs up and shuts off. Place the following in the diagram: call detected, answer call, play announcement, record message, caller hangs up, announcement complete.

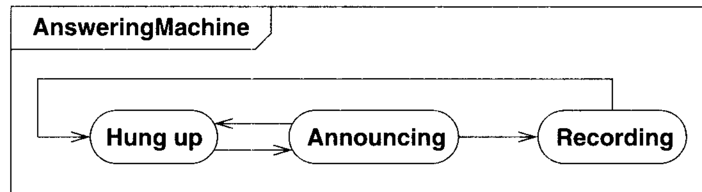


Figure E5.1 Partial state diagram for an answering machine

5.4 (7) The telephone answering machine in the previous exercise activates on the first ring. Revise the state diagram so that the machine answers after five rings. If someone answers the telephone before five rings, the machine should do nothing. Be careful to distinguish between five calls in which the telephone is answered on the first ring and one call that rings five times.

5.5 (3) In a personal computer, a disk controller is typically used to transfer a stream of bytes from a floppy disk drive to a memory buffer with the help of a host such as the central processing unit (CPU) or a direct memory access (DMA) controller. Figure E5.2 shows a partially completed and simplified state diagram for the control of the data transfer.

The controller signals the host each time a new byte is available. The data must then be read and stored before another byte is ready. When the disk controller senses the data has been read, it indicates that data is not available, in preparation for the next byte. If any byte is not read before the next one comes along, the disk controller asserts a data lost error signal until the disk controller is reset. Add the following to the diagram: reset, indicate data not available, indicate data available, data read by host, new data ready, indicate data lost.

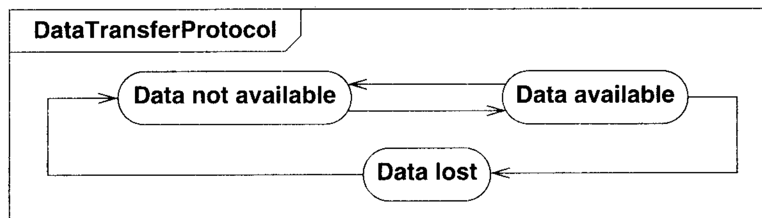


Figure E5.2 Partially completed state diagram of a data transfer protocol

5.6 (5) Figure E5.3 is a partially completed state diagram for one kind of motor control that is commonly used in household appliances. A separate appliance control determines when the motor should be on and continuously asserts *on* as an input to the motor control when the motor should be running.

When *on* is asserted, the motor control should start and run the motor. The motor starts by applying power to both the *start* and the *run* windings. A sensor, called a *starting relay*, determines when the motor has started, at which point the *start* winding is turned off, leaving only the *run* winding powered. Both windings are shut off when *on* is not asserted.

Appliance motors could be damaged by overheating if they are overloaded or fail to start. To protect against thermal damage, the motor control often includes an over-temperature sensor. If

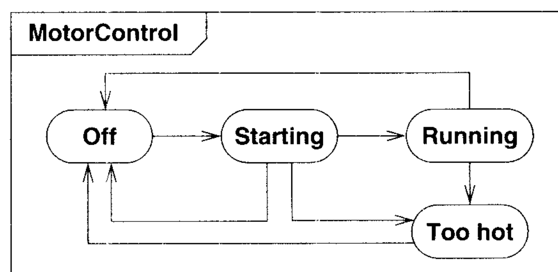


Figure E5.3 Partially completed state diagram for a motor control

the motor becomes too hot, the motor control removes power from both windings and ignores any *on* assertion until a reset button is pressed and the motor has cooled off.

Add the following to the diagram. Activities: apply power to run winding, apply power to start winding. Events: motor is overheated, on is asserted, on is no longer asserted, motor is running, reset. Condition: motor is not overheated.

5.7 (6) There was a single, continuously active input to the control in Exercise 5.6. In another common motor control, there are two pushbuttons, one for *start* and one for *stop*. To start the motor, the user presses the *start* button. The motor continues to run after the *start* button is released.To stop the motor, the user presses the *stop* button. The *stop* button takes precedence over the *start* button, so that the motor does not run while both buttons are pressed.

If both buttons are pressed and released, whether or not the motor starts depends on the order in which the buttons are released. If the *stop* button is released first, the motor starts. Otherwise the motor does not start. Modify the state diagram that you prepared in Exercise 5.6 to accommodate *start* and *stop* buttons.

5.8 (5) Prepare a state diagram for selecting and dragging objects with the diagram editor described in Exercise 4.2.

A cursor on the diagram tracks a two-button mouse. If the left button is pressed with the cursor on an object (a box or a line), the object is selected, replacing any previously selected object. If the left button is pressed with the cursor not on an object, the selection is set to null. Moving the mouse with the left button held down drags any selected object.

5.9 (6) Extend the diagram editor from Exercise 5.8. If the user left clicks on an object and holds the shift key, the object is added to the selection. Moving the mouse with the left button held down drags any selected objects.

5.10 (5) Figure E5.4 shows a state diagram for a copy machine. Initially the copy machine is off. When power is turned on, the machine reverts to a default state—one copy, automatic contrast, and normal size. While the machine is warming, it flashes the ready light. When the machine completes internal testing, the ready light stops flashing and remains on. Then the machine is ready for copying.

The operator may change any of the parameters when the machine is ready. The operator may increment or decrement the number of copies, change the size, toggle between automatic and manual contrast, and change the contrast when auto contrast is disabled. When the parameters are properly set, the operator pushes the start button to begin making copies. Ordinarily, copying proceeds until all copies are made. Occasionally the machine may jam or run out of

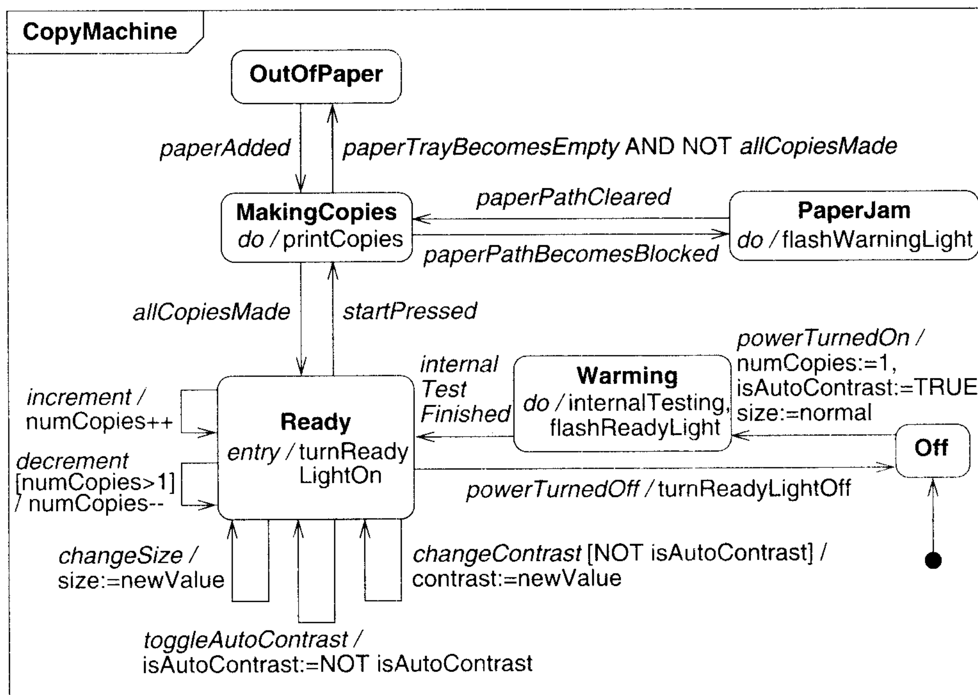


Figure E5.4 State diagram for a copy machine

paper. When the machine jams, the operator may clear the blockage and the machine will resume making copies. Adding paper allows the machine to proceed after running out of paper.

Extend the diagram for the following observations. The copy machine does not work quite right. When it jams, the operator must first remove the jammed paper and then turn the machine off and on before it will operate correctly again. If the machine is turned off and on without first removing the offending paper, the machine stays jammed.

5.11 (7) While exploring an old castle, you and a friend discovered a bookcase that you suspected to be the entrance to a secret passageway. While you examined the bookcase, your friend removed a candle from its holder, only to discover that the candle holder was the entrance control. The bookcase rotated a half turn, pushing you along, separating you from your friend. Your friend put the candle back. This time the bookcase rotated a full turn, still leaving you behind it.

Your friend took the candle out. The bookcase started to rotate a full turn again, but this time you stopped it just shy of a full turn by blocking it with your body. Your friend handed you the candle and together you managed to force the bookcase back a half turn, but this left your friend behind it and you in front of it. You put the candle back. As the bookcase began to rotate, you took out the candle, and the bookcase stopped after a quarter turn. You and your friend then entered to explore further.

Prepare a state diagram for the control of the bookcase that is consistent with the previous scenario. What should you have done at first to gain entry with the least fuss?

6

Advanced State Modeling

Conventional state diagrams are sufficient for describing simple systems but need additional power to handle large problems. You can more richly model complex systems by using nested state diagrams, nested states, signal generalization, and concurrency.

This is an advanced chapter and you can skip it upon a first reading of the book.

6.1 Nested State Diagrams

6.1.1 Problems with Flat State Diagrams

State diagrams have often been criticized because they allegedly are impractical for large problems. This problem is true of flat, unstructured state diagrams. Consider an object with n independent Boolean attributes that affect control. Representing such an object with a single flat state diagram would require 2^n states. By partitioning the state into n independent state diagrams, however, only $2n$ states are required.

Or consider the state diagram in Figure 6.1 in which n^2 transitions are needed to connect every state to every other state. If this model can be reformulated using structure, the number of transitions could be reduced as low as n. Complex systems typically contain much redundancy that structuring mechanisms can simplify.

6.1.2 Expanding States

One way to organize a model is by having a high-level diagram with subdiagrams expanding certain states. This is like a macro substitution in a programming language. Figure 6.2 shows such a state diagram for a vending machine. Initially, the vending machine is idle. When a person inserts coins, the machine adds the amount to the cumulative balance. After adding some coins, a person can select an item. If the item is empty or the balance is insufficient, the machine waits for another selection. Otherwise, the machine dispenses the item and returns the appropriate change.

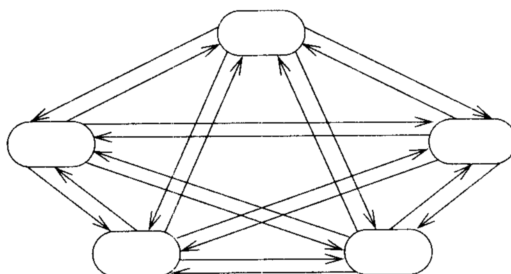


Figure 6.1 Combinatorial explosion of transitions in flat state diagrams. Flat state diagrams are impractical for large problems.

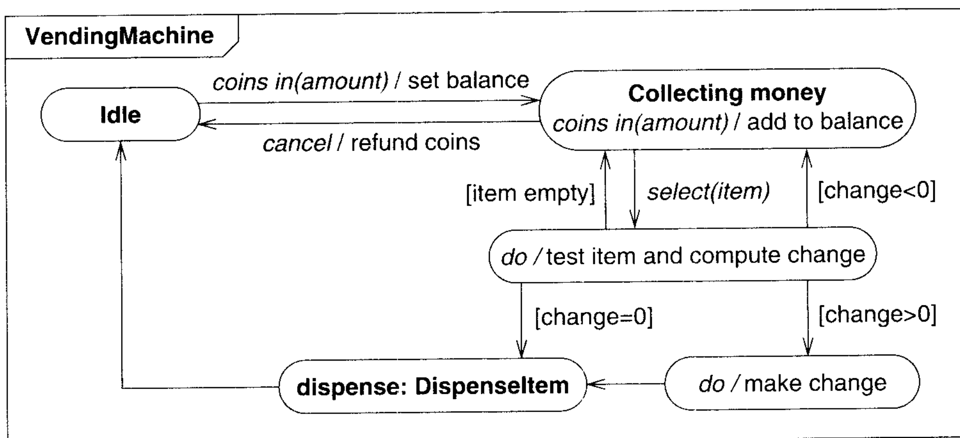


Figure 6.2 Vending machine state diagram. You can simplify state diagrams by using subdiagrams.

Figure 6.3 elaborates the *dispense* state with a lower-level state diagram called a submachine. A ***submachine*** is a state diagram that may be invoked as part of another state diagram. The UML notation for invoking a submachine is to list a local state name followed by a colon and the submachine name. Conceptually, the submachine state diagram replaces the local state. Effectively, a submachine is a state diagram "subroutine."

6.2 Nested States

You can structure states more deeply than just replacing a state with a submachine. As a deeper alternative, you can nest states to show their commonality and share behavior. (In accordance with UML2 we avoid using *generalization* in conjunction with states. See the *Bibliographic Notes* for an explanation.)

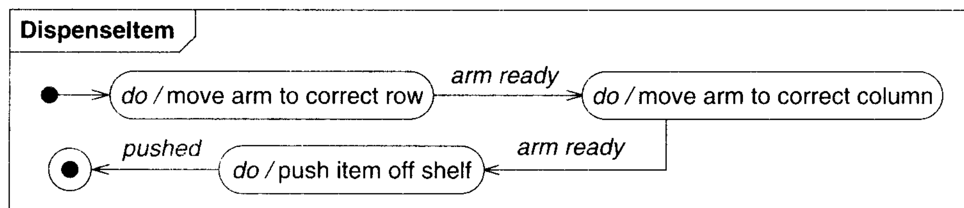


Figure 6.3 ***Dispense item*** **submachine of vending machine**. A lower-level state diagram can elaborate a state.

Figure 6.4 simplifies the phone line model from Chapter 5; a single transition from *Active* to *Idle* replaces the transitions from each state to *Idle*. All the original states except *Idle* are nested states of *Active*. The occurrence of event *onHook* in any nested state causes a transition to state *Idle*.

The ***composite state*** name labels the outer contour that entirely encloses the nested states. Thus *Active* is a composite state with regard to nested states *DialTone*, *Timeout*, *Dialing*, and so forth. You may nest states to an arbitrary depth. A nested state receives the outgoing transitions of its composite state. (By necessity, only ingoing transitions with a specified nested state can be shared, or there would be ambiguity.)

Figure 6.5 shows a state diagram for an automobile automatic transmission. The transmission can be in reverse, neutral, or forward; if it is in forward, it can be in first, second, or third gear. States *First*, *Second*, and *Third* are nested states of state *Forward*.

Each of the nested states receives the outgoing transitions of its composite state. Selecting "N" in any forward gear shifts a transition to neutral. The transition from *Forward* to *Neutral* implies three transitions, one from each forward gear to neutral. Selecting "F" in neutral causes a transition to forward. Within state *Forward*, nested state *First* is the default initial state, shown by the unlabeled transition from the solid circle within the *Forward* contour. *Forward* is just an abstract state; control must be in a real state, such as *First*.

All three nested states share the transition on event *stop* from the *Forward* contour to state *First*. In any forward gear, stopping the car causes a transition to *First*.

It is possible to represent more complicated situations, such as an explicit transition from a nested state to a state outside the contour, or an explicit transition into the contour. In such cases, all the states must appear on one diagram. In simpler cases where there is no interaction except for initiation and termination, you can draw the nested states as separate diagrams and reference them by including a submachine, as in the vending machine example of Figure 6.2.

For simple problems you can implement nested states by degradation into "flat" state diagrams. Another option is to promote each state to a class, but then you must take special care to avoid loss of object identity. The *becomes* operation of Smalltalk lets an object change class without a loss of identity, facilitating promotion of a state to a class. However, the performance overhead of the *becomes* operation may become an issue with many state changes. Promotion of a state to a class is impractical with C++, unless you use advanced techniques, such as those discussed in [Coplien-92]. Java is similar to C++ in this regard.

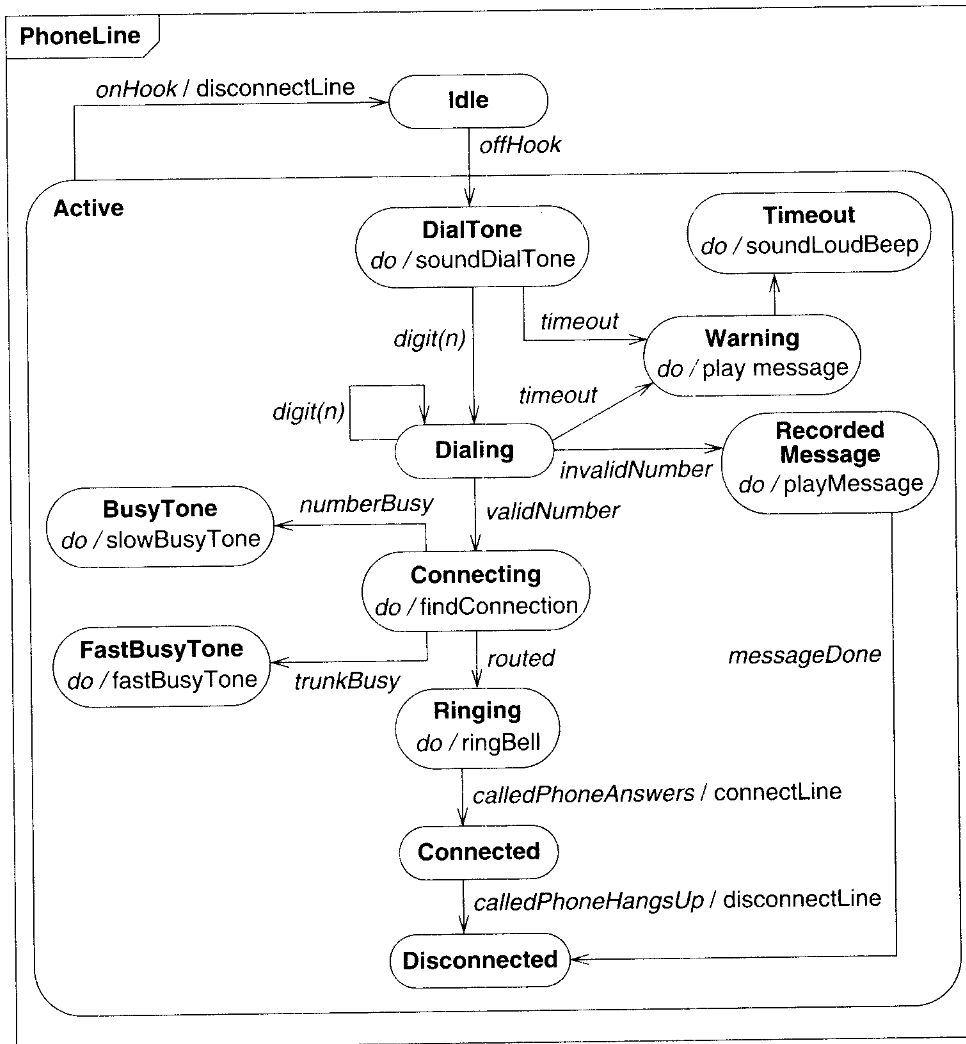


Figure 6.4 Nested states for a phone line. A nested state receives the outgoing transitions of its enclosing state.

Entry and exit activities are particularly useful in nested state diagrams because they permit a state (possibly an entire subdiagram) to be expressed in terms of matched entry-exit activities without regard for what happens before or after the state is active. Transitioning into or out of a nested state can cause execution of several entry or exit activities, if the transition reaches across several levels of nesting. The entry activities are executed from the outside in and the exit activities from the inside out. This permits behavior similar to nested subroutine calls.

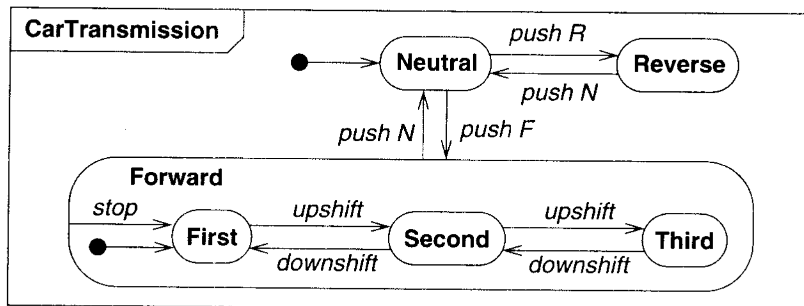


Figure 6.5 Nested states. You can nest states to an arbitrary depth.

6.3 Signal Generalization

You can organize signals into a generalization hierarchy with inheritance of signal attributes. Figure 6.6 shows part of a tree of input signals for a workstation. Signals *MouseButton* and *KeyboardCharacter* are two kinds of user input. Both signals inherit attribute *device* from signal *UserInput* (the root of the hierarchy). *MouseButtonDown* and *MouseButtonUp* inherit *location* from *MouseButton*. *KeyboardCharacters* can be divided into *Control* and *Graphic* characters. Ultimately you can view every actual signal as a leaf on a generalization tree of signals. In a state diagram, a received signal triggers transitions that are defined for any ancestor signal type. For example, typing an 'a' would trigger a transition on signal *Alphanumeric* as well as signal *KeyboardCharacter*. Analogous to generalization of classes, we recommend that all supersignals be abstract.

A signal hierarchy permits different levels of abstraction to be used in a model. For example, some states might handle all input characters the same; other states might treat control characters differently from printing characters; still others might have different activities on individual characters.

6.4 Concurrency

The state model implicitly supports concurrency among objects. In general, objects are autonomous entities that can act and change state independent of one another. However, objects need not be completely independent and may be subject to shared constraints that cause some correspondence among their state changes.

6.4.1 Aggregation Concurrency

A state diagram for an assembly is a collection of state diagrams, one for each part. The aggregate state corresponds to the combined states of all the parts. Aggregation is the "and-relationship." The aggregate state is one state from the first diagram, *and* a state from the second diagram, *and* a state from each other diagram. In the more interesting cases, the part

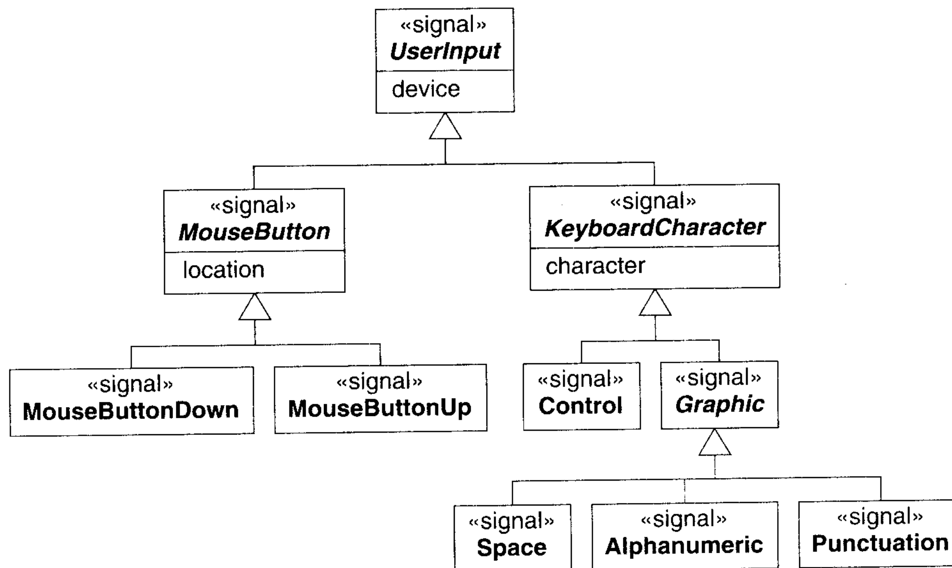


Figure 6.6 Partial hierarchy for keyboard signals. You can organize signals using generalization.

states interact. Transitions for one object can depend on another object being in a given state. This allows interaction between the state diagrams, while preserving modularity.

Figure 6.7 shows the state of a *Car* as an aggregation of part states: *Ignition*, *Transmission*, *Accelerator*, and *Brake* (plus other unmentioned objects). The state of the car includes one state from each part. Each part undergoes transitions in parallel with all the others. The state diagrams of the parts are almost, but not quite, independent—the car will not start unless the transmission is in neutral. This is shown by the guard expression *Transmission in Neutral* on the transition from *Ignition-Off* to *Ignition-Starting*.

6.4.2 Concurrency within an Object

You can partition some objects into subsets of attributes or links, each of which has its own subdiagram. The state of the object comprises one state from each subdiagram. The subdiagrams need not be independent; the same event can cause transitions in more than one subdiagram. The UML shows concurrency within an object by partitioning the composite state into regions with dotted lines. You should place the name of the composite state in a separate tab so that it does not become confused with the concurrent regions.

Figure 6.8 shows the state diagram for the play of a bridge rubber. When a side wins a game, it becomes "vulnerable"; the first side to win two games wins the rubber. During the play of the rubber, the state of the rubber consists of one state from each subdiagram. When the *Playing rubber* composite state is entered, both regions are initially in their respective default states *Not vulnerable*. Each region can independently advance to state *Vulnerable*

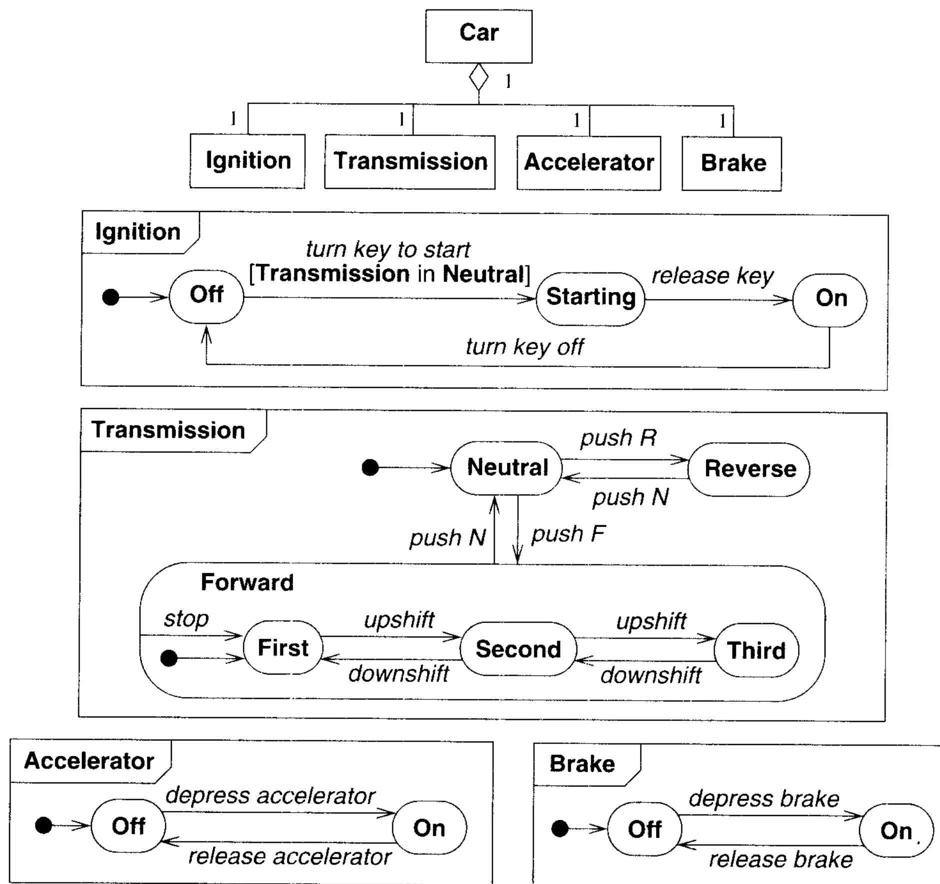


Figure 6.7 An aggregation and its concurrent state diagrams. The state diagram for an assembly is a collection of state diagrams, one for each part.

when its side wins a game. When one side wins a second game, a transition occurs to the corresponding *Wins rubber* state. This transition terminates both concurrent regions, because they are part of the same composite state *Playing rubber* and are active only when the top-level state diagram is in that state.

Most programming languages lack intrinsic support for concurrency. You can use a library, operating system primitives, or a DBMS to provide concurrency. During analysis you should regard all objects as concurrent. During design you devise the best accommodation; many implementations do not require concurrency, and a single thread of control suffices.

6.4.3 Synchronization of Concurrent Activities

Sometimes one object must perform two (or more) activities concurrently. The object does not synchronize the internal steps of the activities but must complete both activities before it can

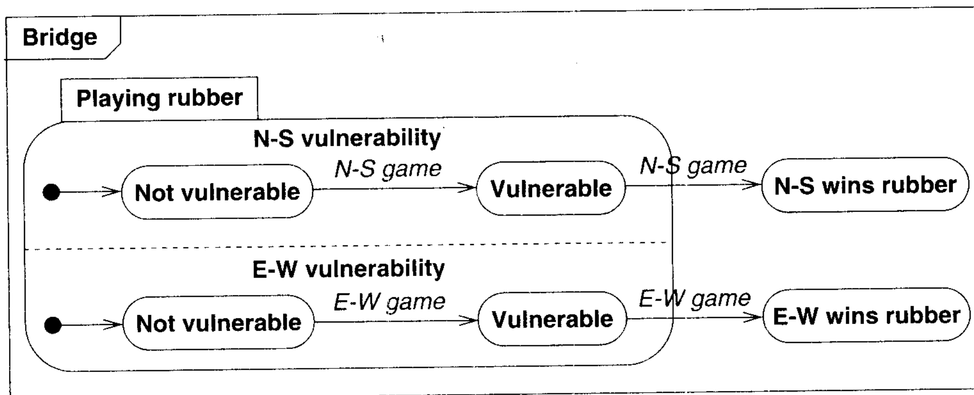


Figure 6.8 Bridge game with concurrent states. You can partition some objects into subsets of attributes or links, each of which has its own subdiagram.

progress to its next state. For example, a cash dispensing machine dispenses cash and returns the user's card at the end of a transaction. The machine must not reset itself until the user takes both the cash and the card, but the user may take them in either order or even simultaneously. The order in which they are taken is irrelevant, only the fact that both of them have been taken. This is an example of *splitting control* into concurrent activities and later *merging control.*

Figure 6.9 shows a concurrent state diagram for the emitting activity. The number of concurrently active states varies during execution from one to two and back to one again. The UML shows concurrent activities within a single composite activity by partitioning a state into regions with dotted lines, as explained previously. Each region is a subdiagram that represents a concurrent activity within the composite activity. The composite activity consists of exactly one state from each subdiagram.

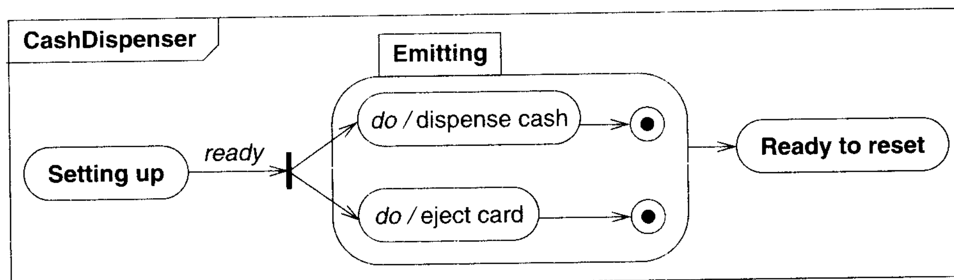


Figure 6.9 Synchronization of control. Control can split into concurrent activities that subsequently merge.

A transition that forks indicates splitting of control into concurrent parts. A small heavy bar with one input arrow and two or more output arrows denotes the fork. The event and an optional guard condition label the input arrow. The output arrows have no labels. Each output

arrow selects a state from a different concurrent subdiagram. In the example, the transition on event *ready* splits into two concurrent parts, one to each concurrent subdiagram. When this transition fires, two concurrent ***substates*** become active and execute independently.

Any transition into a state with concurrent subdiagrams activates each of the subdiagrams. If the transition omits any subdiagrams, the subdiagrams start in their default initial states. In this example, a forked arrow is not actually necessary. You could draw a transition to the *Emitting* state, with each subdiagram having a default initial state.

The UML shows explicit merging of concurrent control by a transition with two or more input arrows and one output arrow, all connected to a small heavy bar (not shown in Figure 6.9). The trigger event and optional guard condition are placed near the bar. The target state becomes active when all of the source states are active and the trigger event occurs. Note that the transition involves a single event, not one event per input arrow. If any subdiagrams in the composite state are not part of the merge, they automatically terminate when the merge transition fires. As a consequence, a transition from a single concurrent substate to a state outside the composite state causes the other concurrent substates to terminate. You can regard this as a degenerate merge involving a single state.

An unlabeled (completion) transition from the outer composite state to another state indicates implicit merging of concurrent control (Figure 6.9). A completion transition fires when activity in the source state is complete. A composite concurrent state is complete when each of its concurrent substates is complete—that is, when each of them has reached its final state. All substates must complete before the completion transition fires and the composite state terminates. In the example, when both activities have been performed, both substates are in their final states, the merge transition fires, and state *Ready to reset* becomes active. Drawing a separate transition from each substate to the target state would have a different meaning; either transition would terminate the other subdiagram without waiting for the other. The firing of a merge transition causes a state diagram to perform the exit activities (if any) of all subdiagrams, in the case of both explicit and implicit merges.

6.5 A Sample State Model

We present a sample state model of a real device (a Sears "Weekender" Programmable Thermostat) to show how the various modeling constructs fit together. We constructed this model by reading the instruction manual and experimenting with the actual device. The device controls a furnace and air conditioner according to time-dependent attributes that the owner enters using a pad of buttons.

While running, the thermostat operates the furnace or air conditioner to keep the current temperature equal to the target temperature. The target temperature is taken from a table of values at the beginning of each program period. The table specifies the target temperature and start time for eight different time periods, four on weekdays and four on weekends. The user can override the target temperature.

The user programs the thermostat using a pad of ten pushbuttons and three switches and sees parameters on an alphanumeric display. Each pushbutton generates an event every time it is pushed. We assign one input event per button:

TEMP UP	raises target temperature or program temperature
TEMP DOWN	lowers target temperature or program temperature
TIME FWD	advances clock time or program time
TIME BACK	retards clock time or program time
SET CLOCK	sets current time of day
SET DAY	sets current day of the week
RUN PRGM	leaves setup or program mode and runs the program
VIEW PRGM	enters program mode to examine and modify eight program time and program temperature settings
HOLD TEMP	holds current target temperature in spite of the program
F-C BUTTON	alternates temperature display between Fahrenheit and Celsius

Each switch supplies a parameter value chosen from two or three possibilities. We model each switch as an independent concurrent subdiagram with one state per switch setting. Although we assign event names to a change in state, it is the state of each switch that is of interest. The switches and their settings are:

NIGHT LIGHT	Lights the alphanumeric display. Values: light off, light on.
SEASON	Specifies which device the thermostat controls. Values: heat (furnace), cool (air conditioner), off (none).
FAN	Specifies when the ventilation fan operates. Values: fan on (fan runs continuously), fan auto (fan runs only when furnace or air conditioner is operating).

The thermostat controls the furnace, air conditioner, and fan power relays. We model this control by activities *run furnace*, *run air conditioner*, and *run fan*.

The thermostat has a sensor for air temperature that it reads continuously, which we model by an external parameter *temp*. The thermostat also has an internal clock that it reads and displays continuously. We model the clock as another external parameter *time*, since we are not interested in building a state model of the clock. In building a state model, it is important to include only states that affect the flow of control and to model other information as parameters or variables. We introduce an internal state variable *target temp* to represent the current temperature that the thermostat is trying to maintain. Some activities read this state variable and others set it; the state variable permits communication among parts of the state model.

Figure 6.10 shows the top-level state diagram of the programmable thermostat. It contains seven concurrent subdiagrams. The user interface is too large to show and is expanded separately (Figure 6.11). The diagram includes trivial subdiagrams for the season switch and the fan switch. The other four subdiagrams show the output of the thermostat: the furnace, air conditioner, the run indicator light, and fan relays. Each of these subdiagrams contains an *Off* and an *On* substate. The state of each subdiagram is totally determined by input parameters and the state of other subdiagrams, such as the season switch or the fan switch. The state of the four subdiagrams on the right is totally derived and contains no additional information.

Figure 6.11 shows the subdiagram for the user interface. The diagram contains three concurrent subdiagrams, one for the interactive display, one for the temperature mode, and

one for the night light. The night light is controlled by a physical switch, so the default initial state is irrelevant; its value can be determined directly. The temperature display mode is controlled by a single pushbutton that toggles the temperature units between Fahrenheit and Celsius. The default initial state is necessary; when the device is powered on, the initial temperature mode is Fahrenheit.

The subdiagram for the interactive display is more interesting. The device is either operating or being set up. State *Operate* has three concurrent substates—one includes *Run* and *Hold*, another controls the target temperature display, and the third controls the time and temperature display. Every two seconds the display alternates between the current time and current temperature. The target temperature is displayed continuously and is modified by the *temp up* and *temp down* buttons, as well as the *set target* event that is generated only in the *Run* state. Note that the *target temp* parameter set by this subdiagram is the same parameter that controls the output relays.

After every second in the *Run* state, the current time is compared to the stored program times in the program table; if they are equal, then the program advances to the next program period, and the *Run* state is reentered. The run state is also entered whenever the *run program* button is pressed in any state, as shown by the transition from the contour to the *Operate* state and the default initial transition to *Run*. Whenever the *Run* state is entered, the entry activity on the state resets the target temperature from the program table.

While the program is in the *Hold* state, the program temperature cannot be advanced automatically, but the temperature can still be modified directly by the *temp up* and *temp down* buttons. If the interface is in one of the setup states for 90 seconds without any input, the system enters the *Hold* state. Entering the *Hold* substate also forces entry to the default initial states of the other two concurrent subdiagrams of *Operate*. The *Setup* state was included in the model just to group the three setup nested states for the 90-second timeout transition. Note a small anomaly of the device: The *hold* button has no effect within the *Setup* state, although the *Hold* state can be entered by waiting for 90 seconds.

The three setup subdiagrams are shown in Figure 6.12. Pressing *set clock* enters the *Set minutes* nested state as initial default. Subsequent *set clock* presses toggle between the *Set hours* and the *Set minutes* nested states. The *time fwd* and *time back* buttons modify the program time. Pressing *set day* enters the *Set day* nested state and shows the day of the week. Subsequent presses increment the day directly.

Pressing *view program* enters the *Set program* nested state, which has three concurrent subdiagrams, one each controlling the display of the program time, program temperature, and program period. The *Set program* state always starts with the first program period, while subsequent *view program* events cycle through the 8 program periods. The *view program* event is shown on all three subdiagrams, each diagram advancing the setting that it controls. Note that the *time fwd* and *time back* events modify time in 15-minute increments, unlike the same events in the *set clock* state. Note also that the *temp up* and *temp down* transitions have guard conditions to keep the temperature in a fixed range.

None of the *Interactive display* nested states has an explicit exit transition. Each nested state is implicitly terminated by a transition into another nested state from the main *Interactive display* contour.

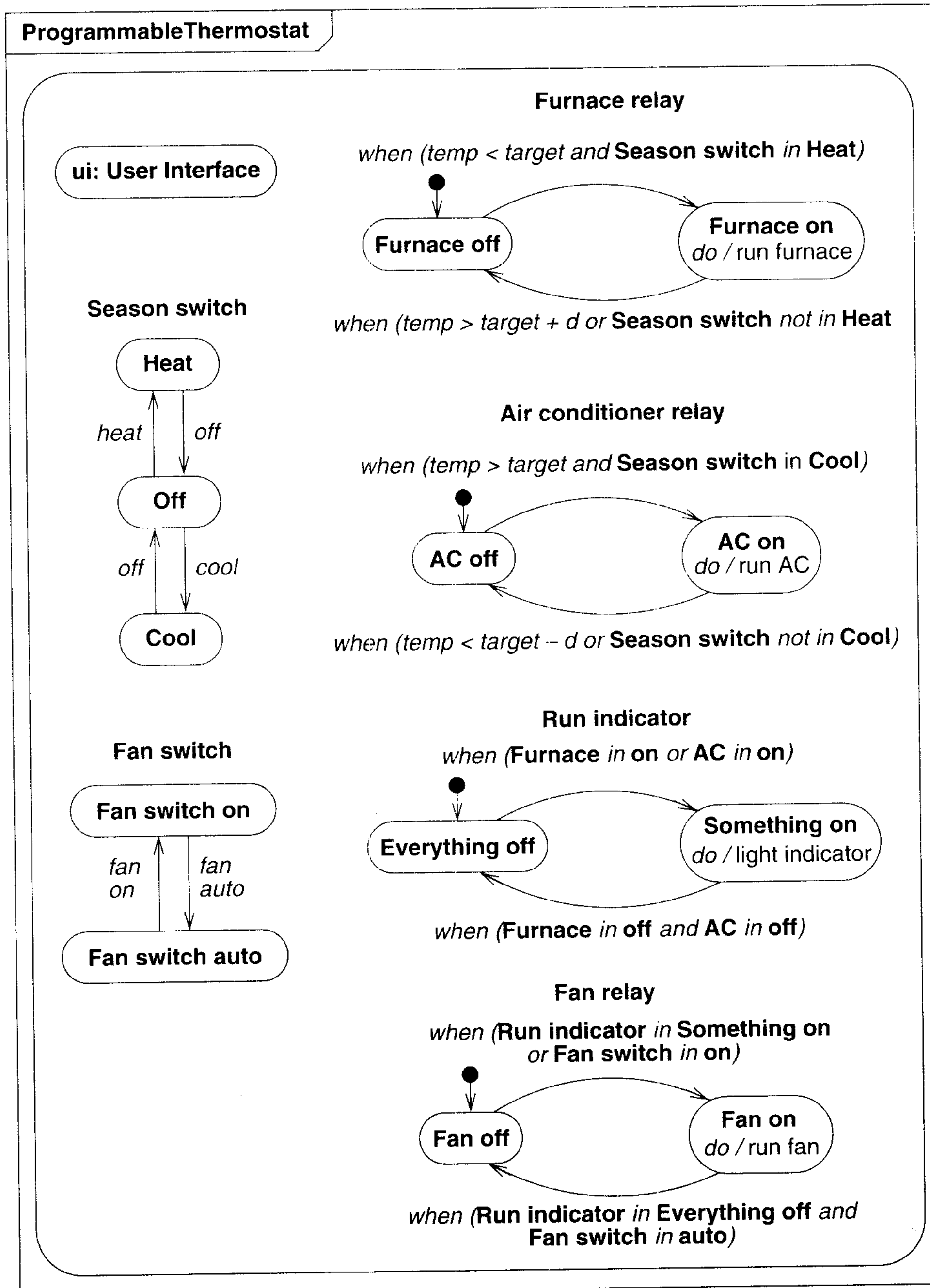


Figure 6.10 State diagram for programmable thermostat

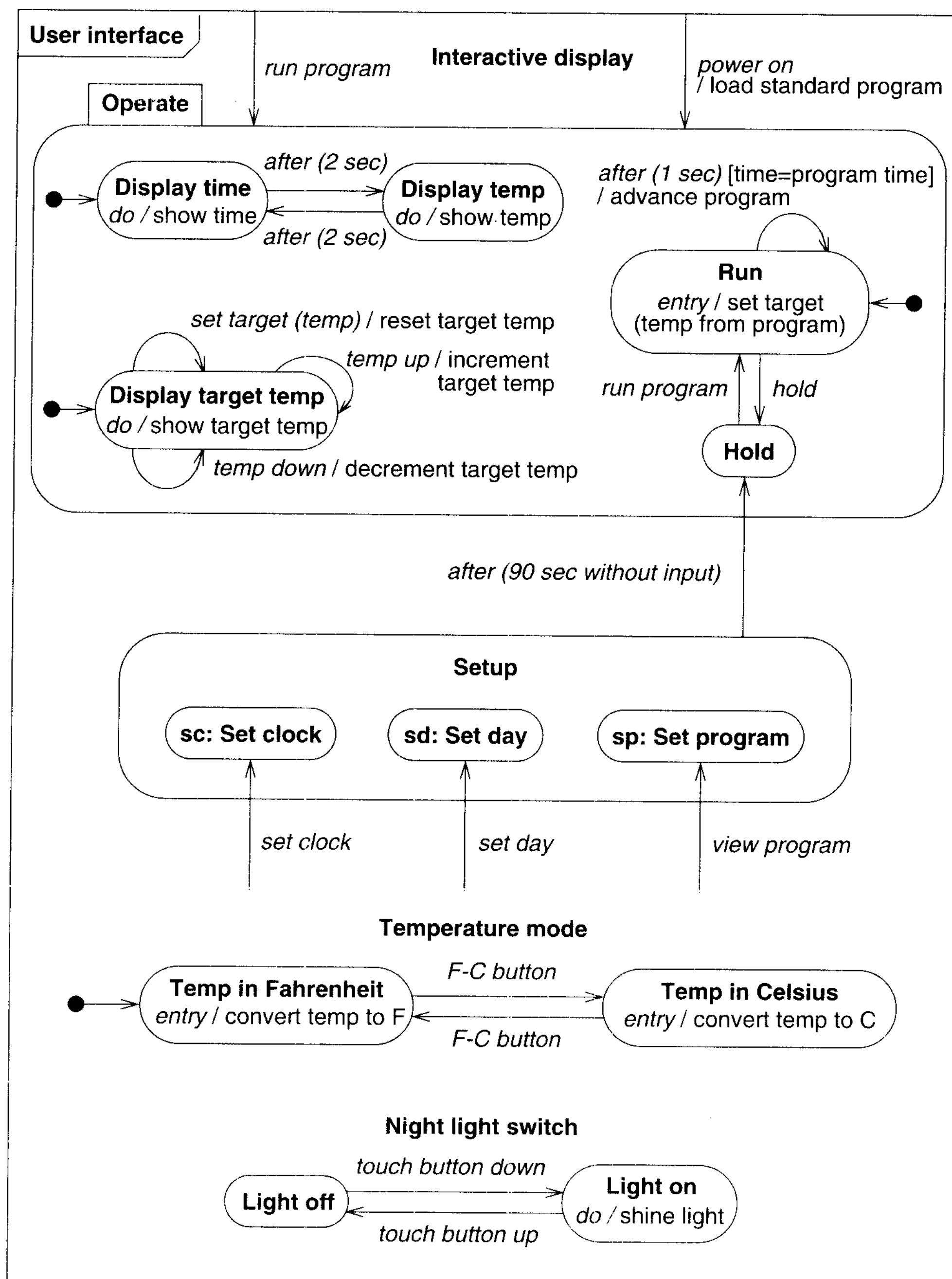


Figure 6.11 Subdiagram for thermostat user interface

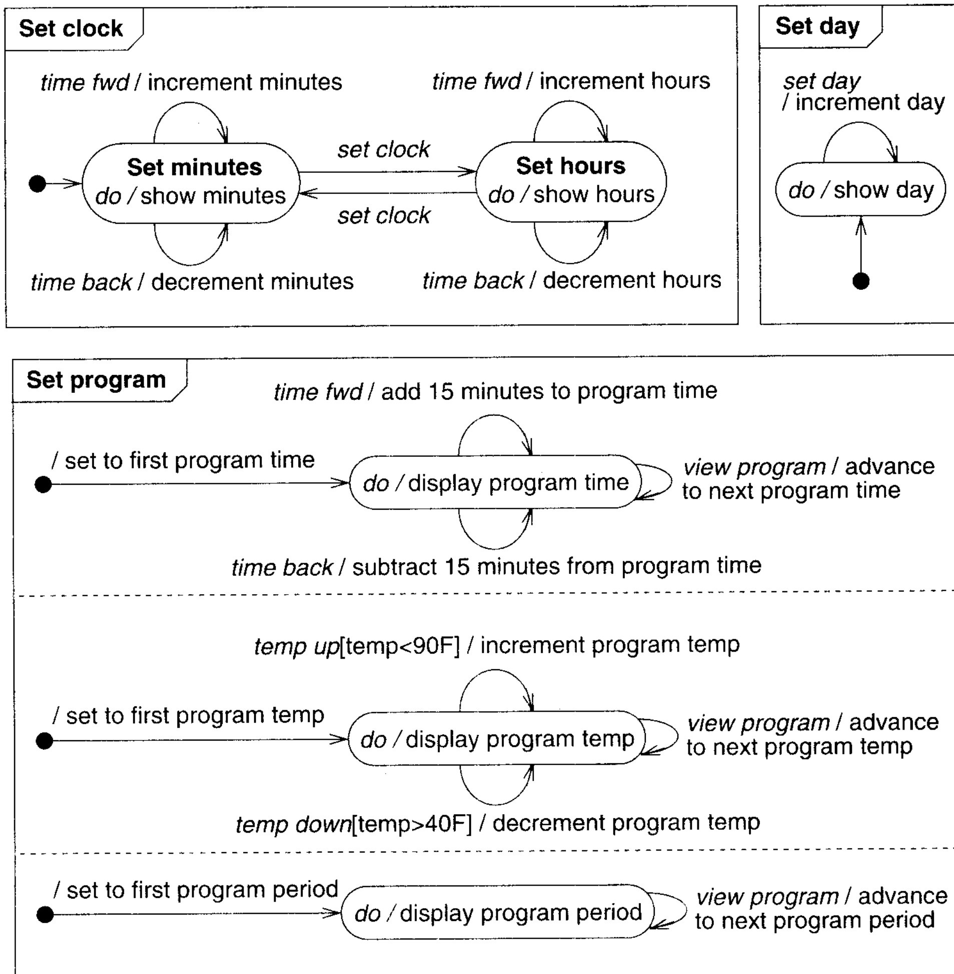


Figure 6.12 Subdiagrams for thermostat user interface setup

6.6 Relation of Class and State Models

The state model specifies allowable sequences of changes to objects from the class model. A state diagram describes all or part of the behavior of the objects of a given class. States are equivalence classes of values and links for an object.

State structure is related to and constrained by class structure. A nested state refines the values and links that an object can have. Both generalization of classes and nesting of states partition the set of possible object values. A single object can have different states over time—the object preserves its identity—but it cannot have different classes. Inherent differ-

ences among objects are therefore properly modeled as different classes, while temporary differences are properly modeled as different states of the same class.

A composite state is the aggregation of more than one concurrent substate. There are three sources of concurrency within the class model. The first is aggregation of objects: Each part of an aggregation has its own independent state, so the assembly can be considered to have a state that is the combination of the states of all its parts. The second source is aggregation within an object: The values and links of an object are its parts, and groups of them taken together define concurrent substates of the composite object state. The third source is concurrent behavior of an object, such as found in Figure 6.9. The three sources of concurrency are usually interchangeable. For example, an object could contain an attribute to indicate that it was performing a certain activity.

The state model of a class is inherited by its subclasses. The subclasses inherit both the states of the ancestor and the transitions. The subclasses can have their own state diagrams. But how do the state diagrams of the superclass and the subclass interact? If the superclass state diagrams and the subclass state diagrams deal with disjoint sets of attributes, there is no problem—the subclass has a composite state composed of concurrent state diagrams.

If, however, the state diagram of the subclass involves some of the same attributes as the state diagram of the superclass, a potential conflict exists. The state diagram of the subclass must be a refinement of the state diagram of the superclass. Any state from the parent state diagram can be elaborated with nesting or split into concurrent parts, but new states or transitions cannot be introduced into the parent diagram directly, because the parent diagram must be a projection of the child diagram. Although refinement of inherited state diagrams is possible, usually the state diagram of a subclass should be an independent, orthogonal, concurrent addition to the state diagram inherited from a superclass, defined on a different set of attributes (usually the ones added in the subclass).

The signal hierarchy is independent of the class hierarchy for the classes exchanging signals, in practice if not in theory. Signals can be defined across different classes of objects. Signals are more fundamental than states and more parallel to classes. States are defined by the interaction of objects and events. Transitions can often be implemented as operations on objects, with the operation name corresponding to the signal name. Signals are more expressive than operations, however, because the effect of a signal depends not only on the class of an object but also on its state.

6.7 Practical Tips

The following practical tips have been mentioned throughout the chapter but are summarized here for convenience.

- **Structured state diagrams**. Use structure to organize models with more than 10–15 states. (Section 6.1)
- **State nesting**. Use nesting when the same transition applies to many states. (Section 6.2)

- **Concrete supersignals**. Analogous to generalization of classes, it is best to avoid concrete supersignals. Then, abstract and concrete signals are readily apparent at a glance—all supersignals are abstract and all leaf subsignals are concrete. You can always eliminate concrete supersignals by introducing an *Other* subsignal. (Section 6.3)
- **Concurrency**. Most concurrency arises from object aggregation and need not be shown explicitly in the state diagram. Use composite states to show independent facets of the behavior of a single object. (Section 6.4)
- **Consistency of diagrams**. Check the various state diagrams for consistency on shared events so that the full state model will be correct. (Section 6.5)
- **State modeling and class inheritance**. Try to make the state diagrams of subclasses independent of the state diagrams of their superclasses. Subclass state diagrams should concentrate on attributes unique to the subclasses. (Section 6.6)

6.8 Chapter Summary

A class model describes the objects, values, and links that can exist in a system. The values and links held by an object are called its state. Over time, the objects stimulate each other, resulting in a series of changes to their states. Objects respond to events, which are occurrences at a point in time. The response to an event depends on the state of the object receiving it, and can include a change of state or the sending of a signal to the original sender or to a third object.

The combinations of events, states, and state transitions for a given class can be abstracted and represented as a state diagram. A state diagram is a network of states and events, just as a class diagram is a network of classes and relationships. The state model consists of multiple state diagrams, one state diagram for each class with important dynamic behavior, and shows the possible behavior for an entire system. Each object independently executes the state diagram for its class. The state diagrams for the various classes communicate via shared events.

States and events can both be expanded to show greater detail. Nested states share the transitions of their composite states. Signals can be organized into inheritance hierarchies. Subsignals trigger the same transitions as their supersignals.

Objects are inherently concurrent, and each object has its own state. State diagrams show concurrency as an aggregation of concurrent states, each operating independently. Concurrent objects interact by exchanging events and by testing conditions of other objects, including states. Transitions can split or merge flow of control.

Entry and exit activities permit activities to cover all the transitions entering or exiting the state. They make self-contained state diagrams possible for use in multiple contexts. Internal activities represent transitions that do not leave the state.

A subclass inherits the state diagrams of its ancestors, to be concurrent with any state diagram that it defines. It is also possible to refine an inherited state diagram by expanding states into nested states or concurrent subdiagrams.

A realistic model of a programmable thermostat takes three pages and illustrates subtleties of behavior that are not apparent from the instruction manual or from everyday operation.

composite state	nested state diagram	state model
concurrency	region	synchronization
control	signal generalization	submachine
nested state	state aggregation	

Figure 6.13 Key concepts for Chapter 6

Bibliographic Notes

Much of this chapter follows the work of David Harel, who has formalized his concepts in a notation called state charts [Harel-87]. Harel's treatment is the most successful attempt to date to structure finite state diagrams and avoid the combinatorial explosion that has plagued them. Harel describes a contour-based notation for state diagrams as a special case of a general diagram notation that he calls *higraphs* [Harel-88].

The first edition of this book included *state generalization*, but the second edition omits the concept in accordance with its omission in UML2. The UML2 metamodel restricts generalization to classifiers and a state is not a classifier. There are similarities between generalization of classes and nesting of states, but strictly speaking, in UML2 there is no state generalization.

There are many fine points of state modeling with UML2. See [Rumbaugh-05] for more information.

We thank Mikael Berndtsson for suggesting Exercise 6.12.

References

[Coplien-92] James O. Coplien. *Advanced C++: Programming Styles and Idioms*. Boston: Addison-Wesley, 1992.

[Harel-87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming 8* (1987), 231–274.

[Harel-88] David Harel. On visual formalisms. *Communications of ACM 31*, 5 (May 1988), 514–530.

[Rumbaugh-05] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Boston: Addison-Wesley, 2005.

Exercises

6.1 (3) The direction control for some of the first toy electric trains was accomplished by interrupting the power to the train. Prepare state diagrams for the headlight and wheels of the train, corresponding to the following sequence of events:

Power is off, train is not moving.
Power is turned on, train runs forward with its headlight shining.
Power is turned off, train stops and headlight goes out.
Power is turned on, headlight shines and train does not move.
Power is turned off, headlight goes out.
Power is turned on, train runs backward with its headlight shining.
Power is turned off, train stops and headlight goes out.
Power is turned on, headlight shines and train does not move.
Power is turned off, headlight goes out.
Power is turned on, train runs forward with its headlight shining.

6.2 (6) Revise the state diagram from Exercise 5.2 to provide for more rapid setting of the time by pressing and holding the B button. If the B button is pressed and held for more than 5 seconds in set time mode, the hours or minutes (depending on the submode) increment once every 1/2 second. (Instructor's note: You may want to give the students a copy of our answer to Exercise 5.2 as the basis for this exercise.)

6.3 (5) Revise the state diagram from your answer to Exercise 5.6 by noting the commonality of the starting and running states. There is a transition from either the starting or the running state to the off state when "on" is not wanted. (Instructor's note: You may want to give the students a copy of our answer to Exercise 5.6 as the basis for this exercise.)

6.4 (6) Three-phase induction motors will spin either clockwise or counterclockwise, depending on the connection to the power lines. In applications requiring motor operation in both directions, two separate contactors (power relays) might be used to make the connections, one for each direction. Also, in some applications of large motors, the motor starts through a transformer that reduces the impact on the power supply. The transformer is bypassed by a third contactor after the motor has been given enough time to come up to speed. There are three momentary control inputs: requests for forward, reverse, or off. When the motor is off, forward or reverse requests cause the motor to start up and run in the requested direction. A reverse request is ignored if the motor is starting or running in the forward direction. and vice versa. An off request at any time shuts the motor off.

Figure E6.1 is a state diagram for one possible motor control. Convert it from a single state diagram into two concurrent state diagrams, one to control the direction of the motor and one for starting control.

6.5 (3) The control in the previous exercise does not provide for thermal protection.

a. Modify the state diagram in Figure E6.1 to shut the motor off if an overheating condition is detected at any time.
b. Modify the concurrent state diagrams that you produced in Exercise 6.4 to shut the motor off if an overheating condition is detected at any time.

6.6 (2) Place the following signal classes into a generalization hierarchy: pick, character input, line pick, circle pick, box pick, text pick, input signal.

6.7 (7) A gas-fired, forced hot-air, home heating system maintains room temperature and humidity in the winter using distributed controls. The comfort of separate rooms may be controlled somewhat independently. Heat is requested from the furnace for each room based on its measured temperature and the desired temperature for that room. When one or more rooms require heat, the furnace is turned on. When the temperature in the furnace is high enough, a blower on the

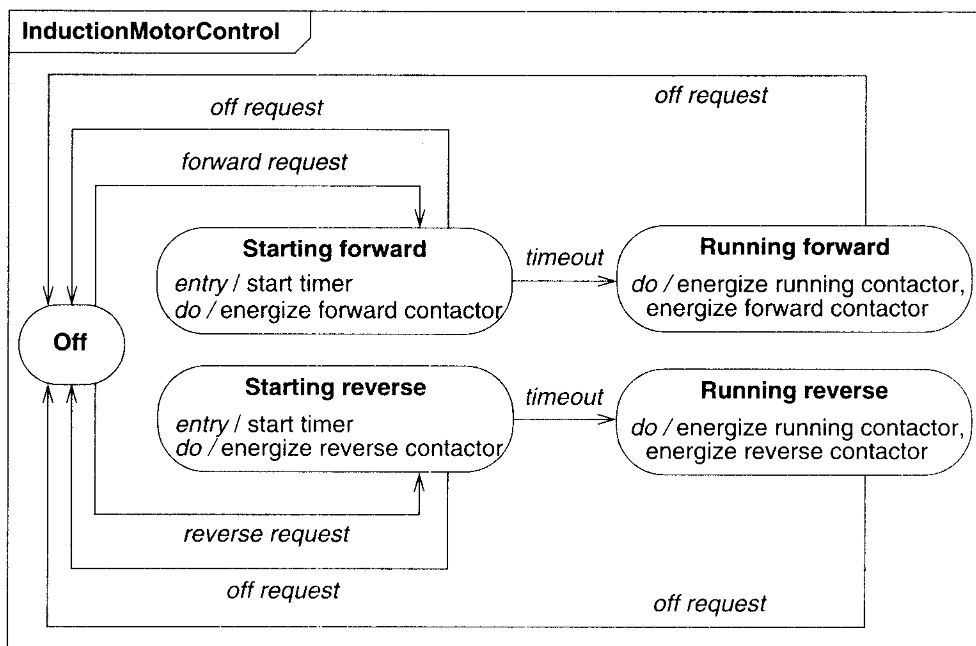


Figure E6.1 State diagram for an induction motor control

furnace is turned on to send hot air through heating ducts. If the temperature in the furnace exceeds a safety limit, the furnace is shut off and the blower continues to run. Flappers in the ducts are controlled by the system to deliver heat only to those rooms that need it. When the room(s) no longer require heat, the furnace is shut off, but the blower continues to deliver hot air until the furnace has cooled off.

Humidity is also maintained based on a strategy involving desired humidity, measured humidity, and outside temperature. The desired humidity is set by the user for the entire home. Humidity of the cool air returning to the blower is measured. When the system determines that the humidity is too low, a humidifier in the furnace is turned on, whenever the blower is on, to inject moisture into the air leaving the blower.

Partition the control of this system into concurrent state diagrams. Describe the functioning of each state diagram without actually going into the details of states or activities.

6.8 Figure E6.2 is a portion of the state diagram for the control of a video cassette recorder (VCR). The VCR has several buttons, including *select*, *on/off*, and *set* for setting the clock and automatic start–stop timers, *auto* for enabling automatic recording, *vcr* for bypassing the VCR, and *timed* for recording for 15-minute increments. Many of the events in Figure E6.2 correspond to pressing the button with the same name. Several buttons have a toggling behavior. For example, pressing *vcr* toggles between VCR and TV mode. Several buttons used for manual control of the VCR are not accounted for in Figure E6.2, such as *play*, *record*, *fast forward*, *rewind*, *pause*, and *eject*. These buttons are enabled only in the *Manual* state. Do the following:

a. (2) Prepare lists of events and activities along with a brief definition.
b. (7) Prepare a user's manual explaining how to operate the VCR.

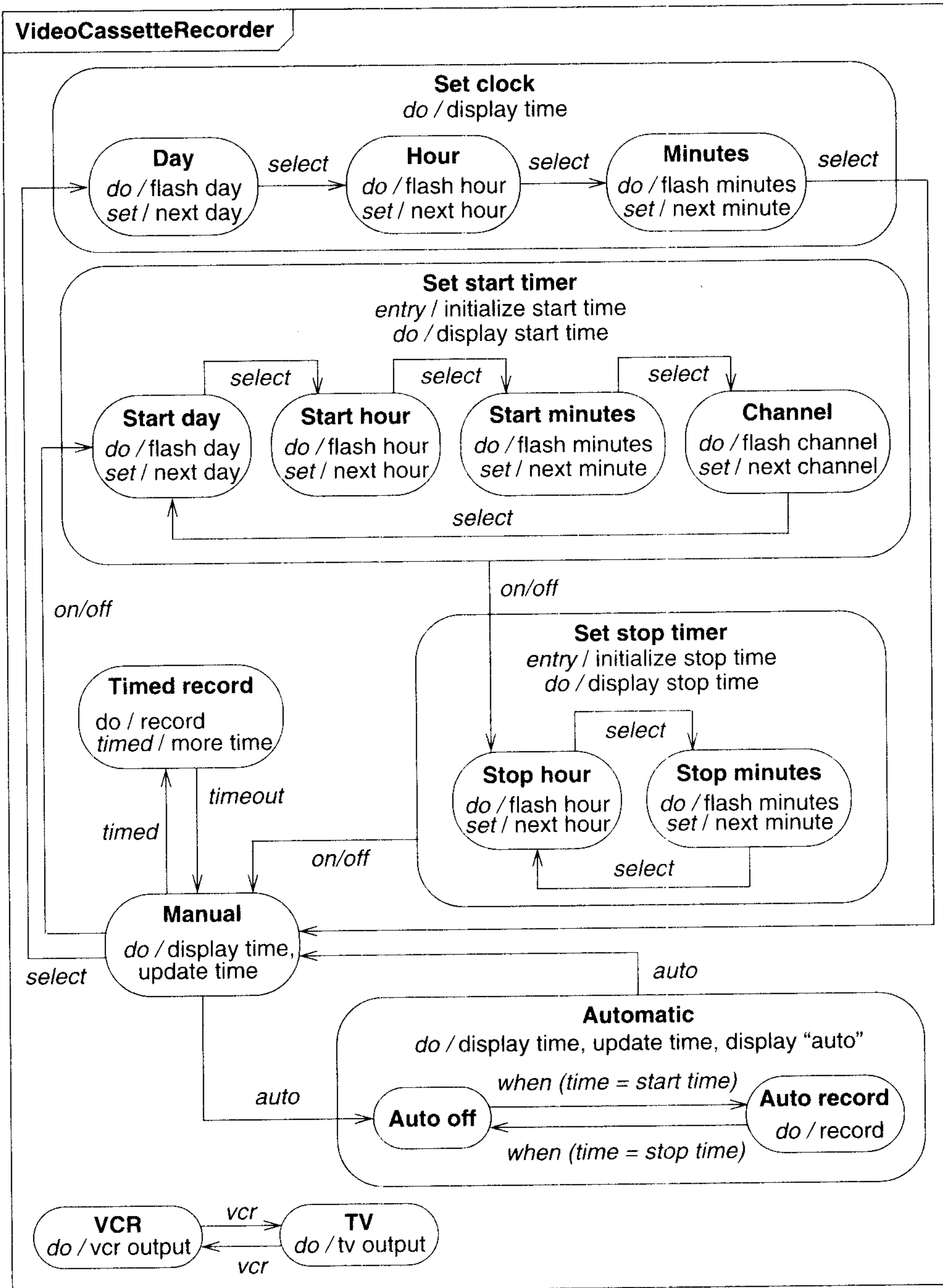


Figure E6.2 Portion of a state diagram for a video cassette recorder

c. (7) By adding states, extend the state diagram to accommodate another start–stop timer for a second channel.
d. (7) There is a great deal of commonality in your answer to the previous part. For example, setting the hour may be done in several contexts with similar results. Discuss how duplication of effort could be reduced.

6.9 (6) The diagram in Figure E5.4 has a major omission. The power can be turned off at any time, and the machine should transition to the off state. We could add a transition from each state to the off state, but this would clutter the diagram. Remedy this defect by using nested states.

6.10 (6) Figure E6.3 contains a class diagram for two persons playing a game of table tennis. Construct a state model corresponding to the class model.

The rules of table tennis are as follows. At the beginning of a game, the two players 'ping' for serve—that is, they bounce the ball over the net and hit it back and forth several times. The winner of the 'ping' serves first.

The winner of the 'ping' serves five times. Then the other player serves five times. Then the winner of the 'ping' serves five times again. This alternation of serve continues until either player wins the game.

A game may be won upon shutout (11-0) or when a player reaches 21 with at least a 2-point margin. If the score becomes tied at 20-20, the players then begin alternating individual serves until a player has a 2-point margin of victory.

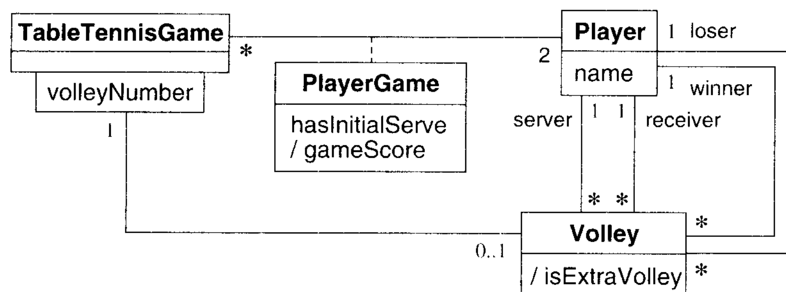


Figure E6.3 Class model for game of table tennis

6.11 (10) Sometimes it is helpful to use reification—to promote something that is not an object into an object. Reification is a helpful technique for meta applications because it lets you shift the level of abstraction. On occasion it is useful to promote attributes, methods, constraints, and control information into objects so you can describe and manipulate them as data.

Construct a class model that reifies and supports the following state modeling concepts: event, state, transition, condition, activity, signal event, change event, and signal attribute.

6.12 (7) Take the model in Figure 6.5 and remove state nesting. That is, construct a flat state diagram with equivalent semantics.

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.

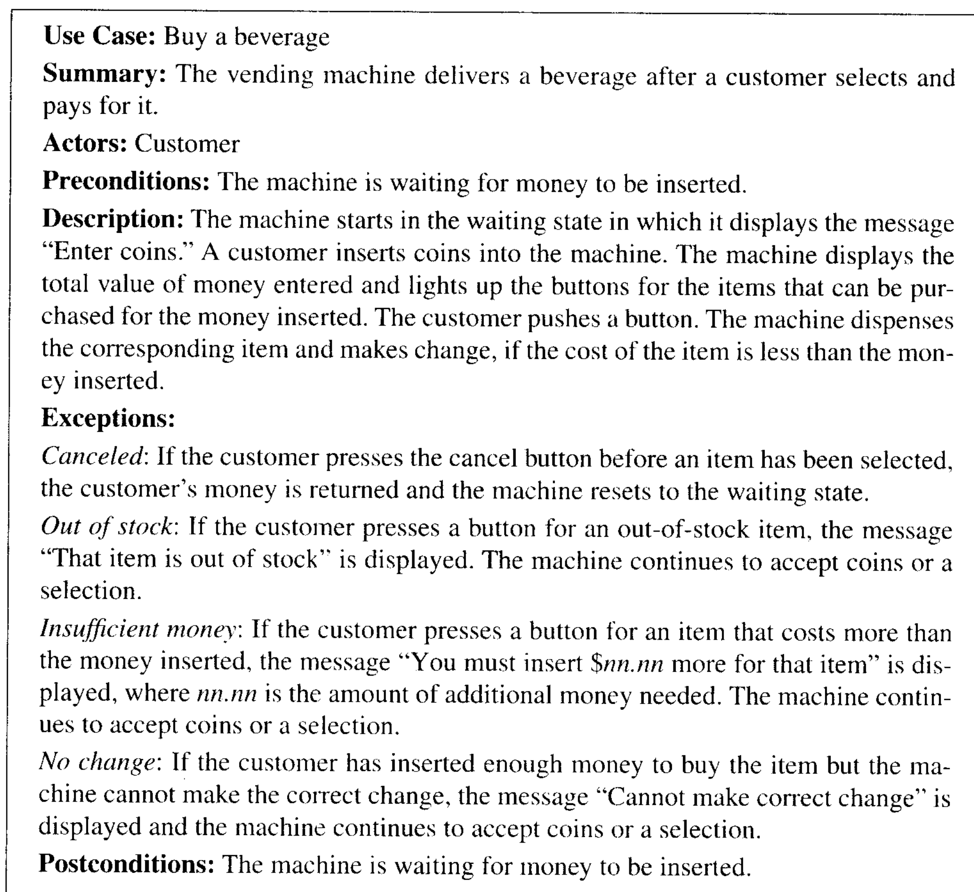
a button to indicate a selection; the vending machine dispenses the beverage and issues change, if necessary.

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

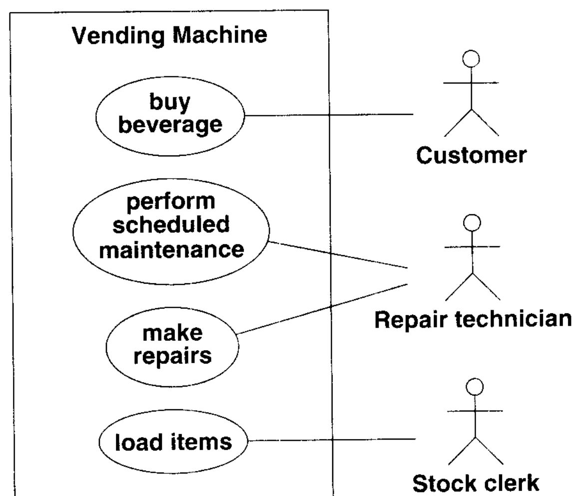


Figure 7.3 Use case diagram for a vending machine. A system involves a set of use cases and a set of actors.

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

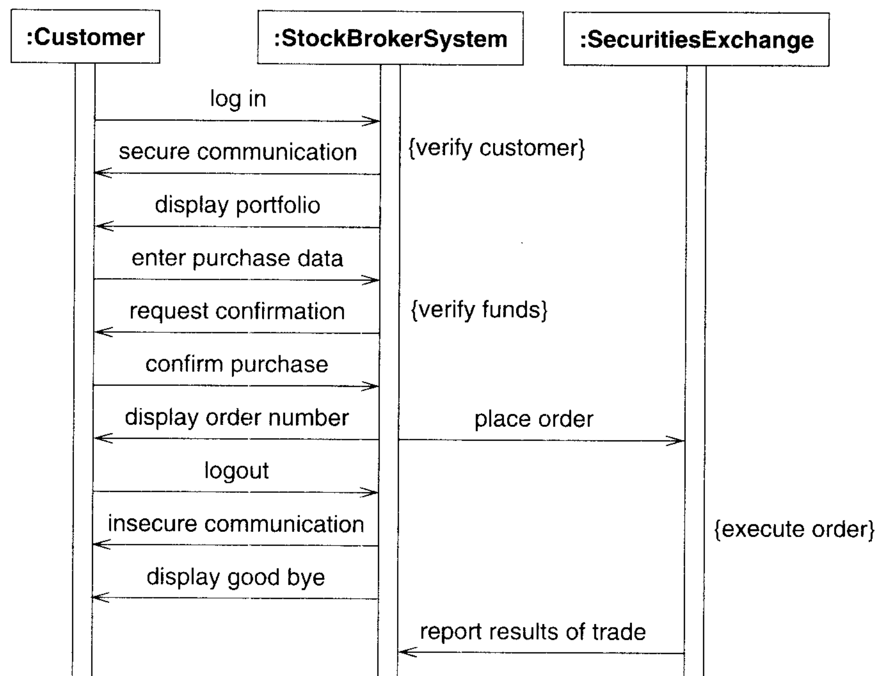


Figure 7.5 Sequence diagram for a session with an online stock broker. A sequence diagram shows the participants in an interaction and the sequence of messages among them.

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.

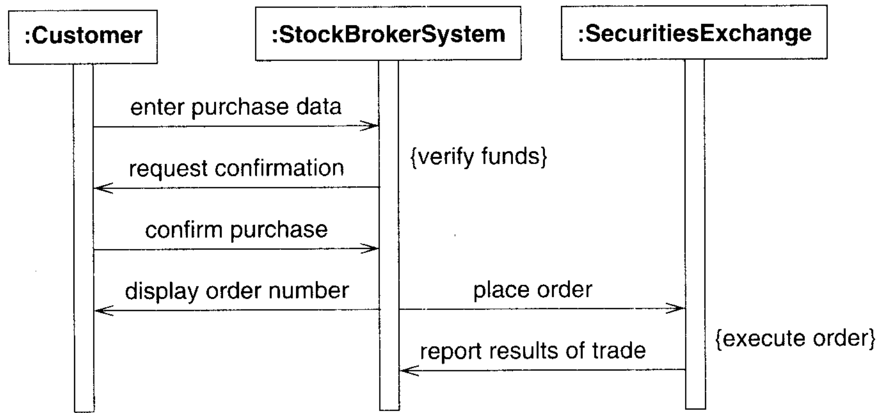


Figure 7.6 Sequence diagram for a stock purchase. Sequence diagrams can show large-scale interactions as well as smaller, constituent tasks.

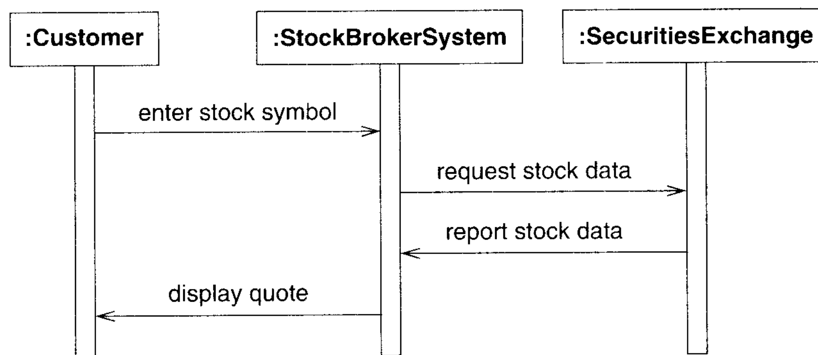


Figure 7.7 Sequence diagram for a stock quote

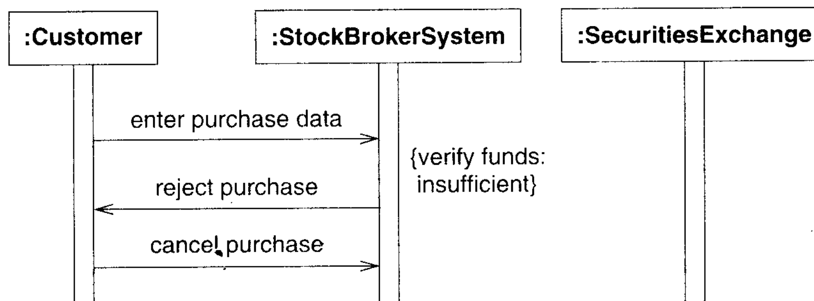


Figure 7.8 Sequence diagram for a stock purchase that fails

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.

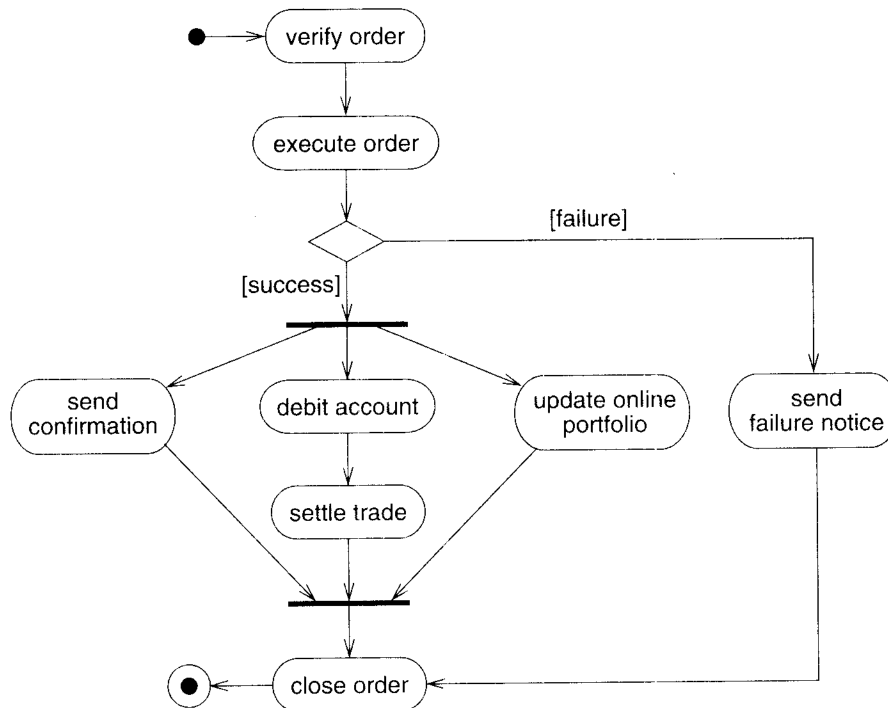


Figure 7.9 Activity diagram for stock trade processing. An activity diagram shows the sequence of steps that make up a complex process.

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.

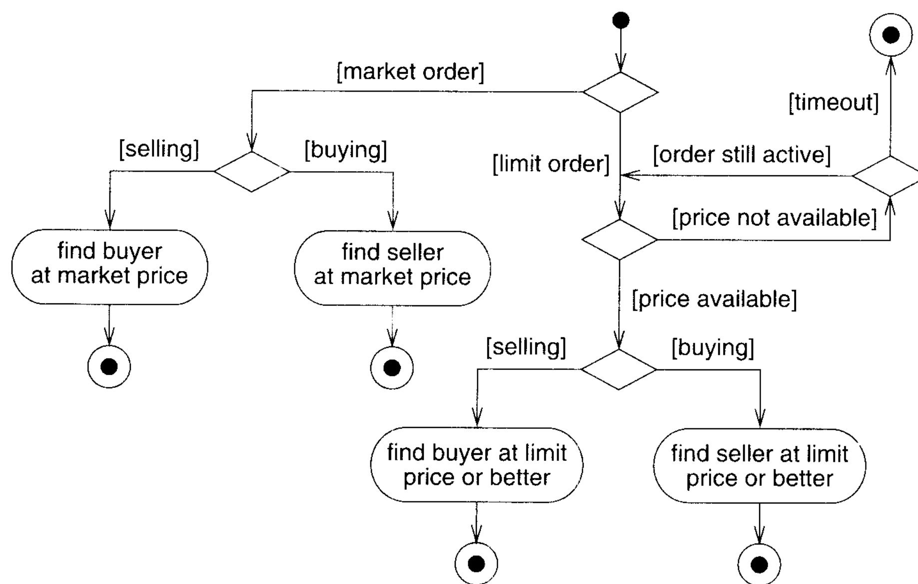


Figure 7.10 Activity diagram for *execute order*. An activity may be decomposed into finer activities.

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.

References

[Jacobson-92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Wokingham, England: Addison-Wesley, 1992.

Exercises

7.1 Consider a physical bookstore, such as in a shopping mall.
  a. (2) List three actors that are involved in the design of a checkout system. Explain the relevance of each actor.
  b. (2) One use case is the purchase of items. Take the perspective of a customer and list another use case at a comparable level of abstraction. Summarize the purpose of each use case with a sentence.
  c. (4) Prepare a use case diagram for a physical bookstore checkout system.
  d. (3) Prepare a normal scenario for each use case. Remember that a scenario is an example, and need not exercise all functionality of the use case.
  e. (3) Prepare an exception scenario for each use case.
  f. (5) Prepare a sequence diagram corresponding to each scenario in (d).

7.2 Consider a computer email system.
  a. (2) List three actors. Explain the relevance of each actor.
  b. (2) One use case is to get email. List four additional use cases at a comparable level of abstraction. Summarize the purpose of each use case with a sentence.
  c. (4) Prepare a use case diagram for a computer email system.
  d. (3) Prepare a normal scenario for each use case. Remember that a scenario is an example, and need not exercise all functionality of the use case.
  e. (3) Prepare an exception scenario for each use case.
  f. (5) Prepare a sequence diagram corresponding to each scenario in (d).

7.3 Consider an online airline reservation system. You may want to check airline Web sites to give you ideas.
  a. (2) List two actors. Explain the relevance of each actor.
  b. (2) One use case is to make a flight reservation. List four additional use cases at a comparable level of abstraction. Summarize the purpose of each use case with a sentence.
  c. (4) Prepare a use case diagram for an online airline reservation system.

7.4 Consider a software system for supporting checkout of materials at a public library.
  a. (2) List four actors. Explain the relevance of each actor.
  b. (2) One use case is to borrow a library item. List three additional use cases at a comparable level of abstraction. Summarize the purpose of each use case with a sentence.
  c. (4) Prepare a use case diagram for a library checkout system.

7.5 (3) Identify at least 10 use cases for the Windows Explorer. Just list them textually and summarize the purpose of each use case in one or two sentences.

7.6 (3) Write scenarios for the following situations:
  a. Moving a bag of corn, a goose, and a fox across a river in a boat. Only one thing may be carried in the boat at a time. If the goose is left alone with the corn, the corn will be eaten. If

the goose is left alone with the fox, the goose will be eaten. Prepare two scenarios, one in which something gets eaten and one in which everything is safely transported across the river.

b. Getting ready to take a trip in your car. Assume an automatic transmission. Don't forget your seat belt and emergency brake.
c. An elevator ride to the top floor.
d. Operation of a car cruise control. Include an encounter with slow-moving traffic that requires you to disengage and then resume control.

7.7 (4) Some combined bath–showers have two faucets and a lever for controlling the flow of the water. The lever controls whether the water flows from the shower head or directly into the tub. When the water is first turned on, it flows directly into the tub. When the lever is pulled, a valve closes and latches, diverting the flow of water to the shower head. To switch from shower to bath with the water running, one must push the lever. Shutting off the water releases the lever, so that the next time the water is turned on, it flows directly into the tub. Write a scenario for a shower that is interrupted by a telephone call.

7.8 (4) Prepare an activity diagram for computing a restaurant bill. There should be a charge for each delivered item. The total amount should be subject to tax and a service charge of 18% for groups of six of more. For smaller groups, there should be a blank entry for a gratuity according to the customer's discretion. Any coupons or gift certificates submitted by the customer should be subtracted.

7.9 (4) Prepare an activity diagram for awarding frequent flyer credits. In the past, TWA awarded a minimum of 750 miles for each flight. Gold and red card holders received a minimum of 1000 miles per flight. Gold card holders received a 25% bonus for any flight. Red card holders received a 50% bonus for any flight.

7.10 (5) Prepare an activity diagram that elaborates the details of logging into an email system. Note that entry of the user name and the password can occur in any order.

8

Advanced Interaction Modeling

The interaction model has several advanced features that can be helpful. You can skip this chapter on a first reading of the book.

8.1 Use Case Relationships

Independent use cases suffice for simple applications. However, it can be helpful to structure use cases for large applications. Complex use cases can be built from smaller pieces with the *include, extend,* and *generalization* relationships.

8.1.1 Include Relationship

The ***include*** relationship incorporates one use case within the behavior sequence of another use case. An included use case is like a subroutine—it represents behavior that would otherwise have to be described repeatedly. Often the fragment is a meaningful unit of behavior for the actors, although this is not required. The included use case may or may not be usable on its own.

The UML notation for an include relationship is a dashed arrow from the source (including) use case to the target (included) use case. The keyword *«include»* annotates the arrow. Figure 8.1 shows an example from an online stock brokerage system. Part of establishing a secure session is validating the user password. In addition, the stock brokerage system validates the password for each stock trade. Use cases *secure session* and *make trade* both include use case *validate password.*

A use case can also be inserted within a textual description with the notation *include use-case-name*. An included use case is inserted at a specific location within the behavior sequence of the larger use case, just as a subroutine is called from a specific location within another subroutine.

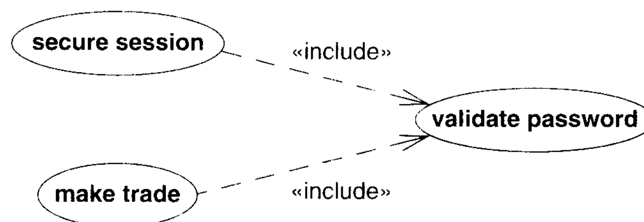


Figure 8.1 Use case inclusion. The *include* relationship lets a base use case incorporate behavior from another use case.

You should not use include relationships to structure fine details of behavior. The purpose of use case modeling is to identify the functionality of the system and the general flow of control among actors and the system. Factoring a use case into pieces is appropriate when the pieces represent significant behavior units.

8.1.2 Extend Relationship

The ***extend*** relationship adds incremental behavior to a use case. It is like an include relationship looked at from the opposite direction, in which the extension adds itself to the base, rather than the base explicitly incorporating the extension. It represents the frequent situation in which some initial capability is defined, and later features are added modularly. The include and extend relationships both add behavior to a base use case.

For example, a stock brokerage system might have the base use case *trade stocks*, which permits a customer to purchase stocks for cash on hand in the account. The extension use case *margin trading* would add the ability to make a loan to purchase stocks when the account does not contain enough cash. It is still possible to buy stocks for cash, but if there is insufficient cash, then the system offers to proceed with the transaction after verifying that the customer is willing to make a margin purchase. The additional behavior is inserted at the point where the purchase cost is checked against the account balance.

Figure 8.2 shows the base use case *trade stocks* for a stock brokerage system. The UML notation for an extend relationship is a dashed arrow from the extension use case to the base use case. The keyword *«extend»* annotates the arrow. The base use case permits simple purchases and sales of a stock at the market price. The brokerage system adds three capabilities: buying a stock on margin, selling a stock short, and placing a limit on the transaction price. The use case *trade options* also has an extension for placing a limit on the transaction price.

The extend relationship connects an extension use case to a base use case. The extension use case often is a fragment—that is, it cannot appear alone as a behavior sequence. The base use case, however, must be a valid use case in the absence of any extensions. The extend relationship can specify an insert location within the behavior sequence of the base use case; the location can be a single step in the base sequence or a range of steps. The behavior sequence of the extension use case occurs at the given point in the sequence. In most cases, an extend relationship has a condition attached. The extension behavior occurs only if the condition is true when control reaches the insert location.

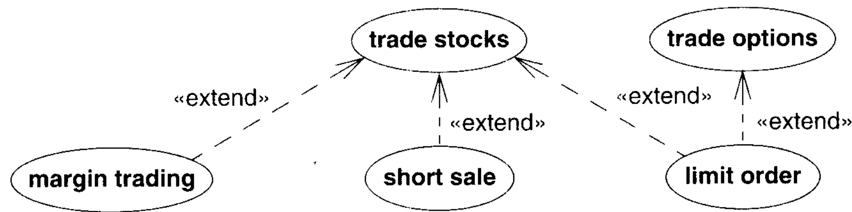


Figure 8.2 Use case extension. The *extend* relationship is like an *include* relationship looked at from the opposite direction. The extension adds itself to the base.

8.1.3 Generalization

Generalization can show specific variations on a general use case, analogous to generalization among classes. A parent use case represents a general behavior sequence. Child use cases specialize the parent by inserting additional steps or by refining steps. The UML indicates generalization by an arrow with its tail on the child use case and a triangular arrowhead on the parent use case, the same notation that is used for classes.

For example, an online stock brokerage system (Figure 8.3) might specialize the general use case *make trade* into the child use cases *trade bonds, trade stocks,* and *trade options*. The parent use case contains steps that are performed for any kind of trade, such as entering the trading password. Each child use case contains the additional steps particular to a specific kind of trade, such as entering the expiration date of an option.

Figure 8.3 Use case generalization. A parent use case has common behavior and child use cases add variations, analogous to generalization among classes.

A parent use case may be abstract or concrete—an abstract use case cannot be used directly. As with the class model, we recommend that you consider only abstract parents and forego concrete ones. Then a model is more symmetric and a parent use case is not cluttered with the handling of special cases. Use cases also exhibit polymorphism—a child use case can freely substitute for a parent use case, for example, as an inclusion in another use case. In all these respects, generalization is the same for use cases and for classes.

In one respect, use case generalization is more complicated than class generalization. A subclass adds attributes to the parent class, but their order is unimportant. A child use case adds behavior steps, but they must appear in the proper locations within the behavior se-

quence of the parent. This is similar to overriding a method that is inherited by a subclass, in which new statements may be inserted at various locations in the parent's method. The simplest approach is to simply list the entire behavior sequence of the child use case, including the steps inherited from the parent. A more general approach is to assign symbolic locations within the parent's sequence and to indicate where additions and replacements go. In general, a child may revise behavior subsequences at several different points in the parent's sequence.

With classes there can be multiple inheritance, but we do not allow such complexity with use cases. In practice, the include and extend relationships obviate the need for multiple inheritance with use cases.

8.1.4 Combinations of Use Case Relationships

A single diagram may combine several kinds of use case relationships. Figure 8.4 shows a use case diagram from a stock brokerage system. The *secure session* use case includes the behavior of the *validate password*, *make trade*, and *manage account* use cases. *Make trade* is an abstract parent with the children—*trade bonds*, *trade stocks*, and *trade options*. Use case *make trade* also includes the behavior of *validate password*. The brokerage system validates the password once per session and additionally for every trade.

The use case *margin trading* extends both *trade bonds* and *trade stocks*—a customer may purchase stocks and bonds on margin, but not options. Use case *limit order* extends abstract use case *make trade*—limit orders apply to trading bonds, stocks, and options. We assume that a *short sale* is only permitted for stocks and not for bonds or options.

Note that the *Customer* actor connects only to the *secure session* use case. The brokerage system invokes all the other use cases indirectly by inclusion, specialization, or extension. The *Securities exchange* actor connects to the *make trade* use case. This actor does not initiate a use case but it is invoked during execution.

8.1.5 Guidelines for Use Case Relationships

Don't carry use case relationships to extremes and lapse into programming. Use cases are intended to clarify requirements. There can be many ways to implement requirements and you should not commit to an approach before you fully understand a problem. Here are some additional guidelines.

- **Use case generalization**. If a use case comes in several variations, model the common behavior with an abstract use case and then specialize each of the variations. Do not use generalization simply to share a behavior fragment; use the include relationship for that purpose.
- **Use case inclusion**. If a use case includes a well-defined behavior fragment that is likely to be useful in other situations, define a use case for the behavior fragment and include it in the original use case. In most cases, you should think of the included use case as a meaningful activity but not as an end in itself. For example, validating a password is meaningful to users but has a purpose only within a broader context.

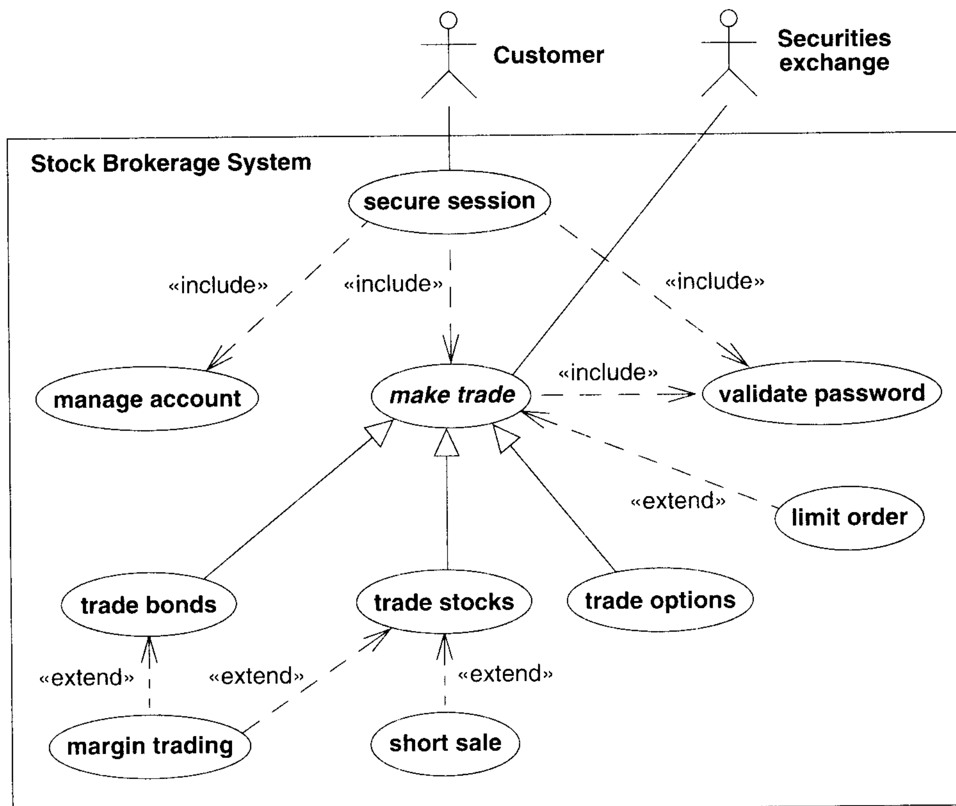


Figure 8.4 Use case relationships. A single use case diagram may combine several kinds of relationships.

- **Use case extension.** If you can define a meaningful use case with optional features, then model the base behavior as a use case and add features with the extend relationship. This permits the system to be tested and debugged without the extensions, which can be added later. Use the extend relationship when a system might be deployed in different configurations, some with the additional features and some without them.
- **Include relationship vs. extend relationship.** The include relationship and the extend relationship can both factor behavior into smaller pieces. The include relationship, however, implies that the included behavior is a necessary part of a configured system (even if the behavior is not executed every time), whereas the extend relationship implies that a system without the added behavior would be meaningful (even if there is no intention to configure it that way).

8.2 Procedural Sequence Models

In Chapter 7, we saw sequence diagrams containing independent objects, all of which are active concurrently. An object remains active after sending a message and can respond to other messages without waiting for a response. This is appropriate for high-level models. However, most implementations are procedural and limit the number of objects that can execute at a time. The UML has elaborations for sequence diagrams to show procedure calls.

8.2.1 Sequence Diagrams with Passive Objects

With procedural code all objects are not constantly active. Most objects are passive and do not have their own threads of control. A passive object is not activated until it has been called. Once the execution of an operation completes and control returns to the caller, the passive object becomes inactive.

Figure 8.5 computes the commission for a stock brokerage transaction. The transaction object receives a request to compute its commission. It obtains the customer's service level from the customer table, then asks the rate table to compute the commission based on the service level, after which it returns the commission value to the caller.

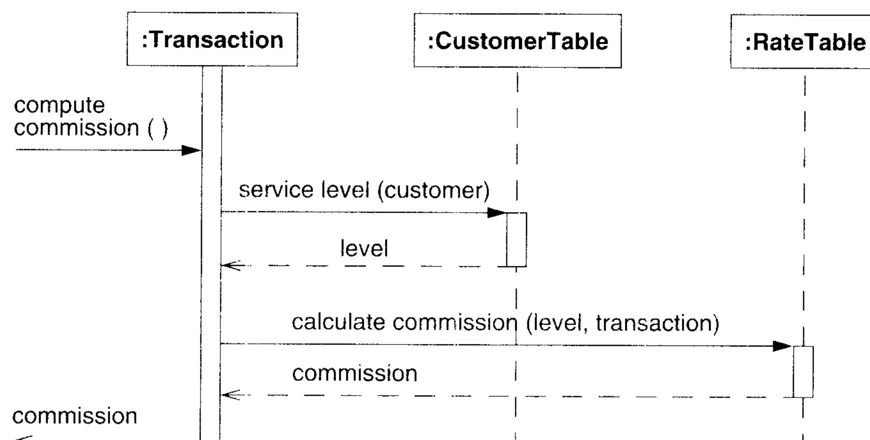


Figure 8.5 Sequence diagram with passive objects. Sequence diagrams can show the implementation of operations.

The UML shows the period of time for an object's execution as a thin rectangle. This is called the ***activation*** or ***focus of control***. An activation shows the time period during which a call of a method is being processed, including the time when the called method has invoked another operation. The period of time when an object exists but is not active is shown as a dashed line. The entire period during which the object exists is called the ***lifeline***, as it shows the lifetime of the object.

8.2.2 Sequence Diagrams with Transient Objects

Figure 8.6 shows further notation. *ObjectA* is an active object that initiates an operation. Because it is active, its activation rectangle spans the entire time shown in the diagram. *ObjectB* is a passive object that exists during the entire time shown in the diagram, but it is not active for the whole time. The UML shows its existence by the dashed line (the lifeline) that covers the entire time period. *ObjectB's* lifeline broadens into an activation rectangle when it is processing a call. During part of the time, it performs a recursive operation, as shown by the doubled activation rectangle between the call by *objectC* on *operationE* and the return of the result value. *ObjectC* is created and destroyed during the time shown on the diagram, so its lifeline does not span the whole diagram.

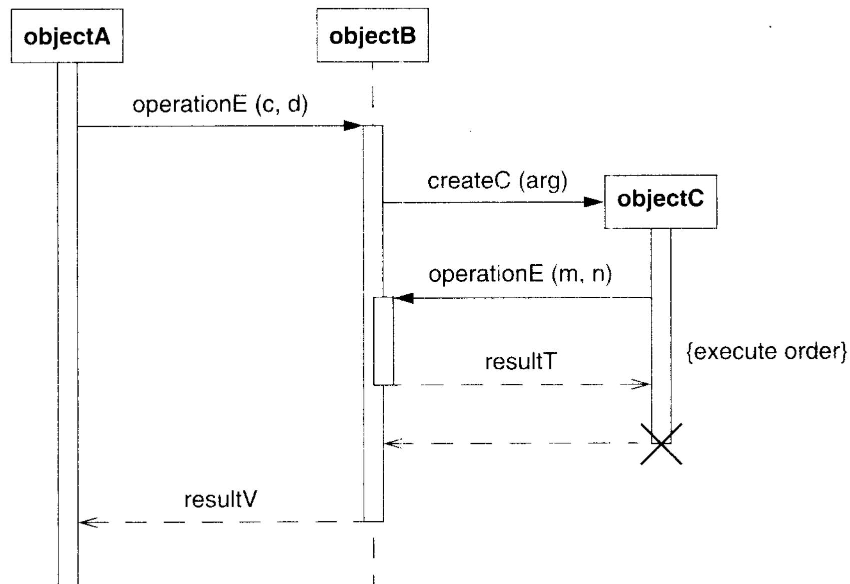


Figure 8.6 Sequence diagram with a transient object. Many applications have a mix of active and passive objects. They create and destroy objects.

The notation for a call is an arrow from the calling activation to the activation created by the call. The tail of the arrow is somewhere along the rectangle of the calling activation. The arrowhead aligns with the top of the rectangle of the newly created activation, because the call creates the activation. The filled arrowhead indicates a call (as opposed to the stick arrowhead for an asynchronous signal in Chapter 7).

The UML shows a return by a dashed arrow from the bottom of the called activation to the calling activation. Not all return arrows have result values—for example, the return from *objectC* to *objectB*. An activation, therefore, has a call arrow coming into its top and a return arrow leaving its bottom. In between, it may have arrows to and from subordinate activations

that it calls. You can suppress return arrows, because their location is implicit at the bottom of the activation, but for clarity it is better to show them.

If an object does not exist at the beginning of a sequence diagram, then it must be created during the sequence diagram. The UML shows creation by placing the object symbol at the head of the arrow for the call that creates the object. For example, the *createC* call creates *objectC*. The new object may or may not retain control after it is created. In the example, *objectC* does retain control, as shown by the activation rectangle that begins immediately below the object rectangle.

Similarly, a large 'X' marks the end of the life of an object that is destroyed during the sequence diagram. The 'X' is placed at the head of the call arrow that destroys the object. If the object destroys itself and returns control to another object, the 'X' is placed at the tail of the return arrow. In the example, *objectC* destroys itself and returns control to *objectB*. The lifeline of the object does not extend before its creation or after its destruction.

The UML shows a call to a second method on the same object (including a recursive call to the same method) with an arrow from the activation rectangle to the top of an additional rectangle superimposed on the first. For example, the second call to *operationE* on *objectB* is a recursive call nested within the first call to *operationE*. The second rectangle is shifted horizontally slightly so that both rectangles can be seen. The number of superimposed rectangles shows the number of activations of the same object.

You can also show conditionals on a sequence diagram, but this is more complex than we wish to include in this book. For further information, see [Rumbaugh-05].

8.2.3 Guidelines for Procedural Sequence Models

There are additional guidelines that apply to procedural sequence models beyond those mentioned in Chapter 7.

- **Active vs. passive objects**. Differentiate between active and passive objects. Most objects are passive and lack their own thread of control. By definition, active objects are always activated and have their own focus of control.
- **Advanced features**. Advanced features can show the implementation of sequence diagrams. Be selective in using these advanced features. Only show implementation details for difficult or especially important sequence diagrams.

8.3 Special Constructs for Activity Models

Activity diagrams have additional notation that is useful for large and complex applications.

8.3.1 Sending and Receiving Signals

Consider a workstation that is turned on. It goes through a boot sequence and then requests that the user log in. After entry of a name and password, the workstation queries the network to validate the user. Upon validation, the workstation then finishes its startup process. Figure 8.7 shows the corresponding activity diagram.

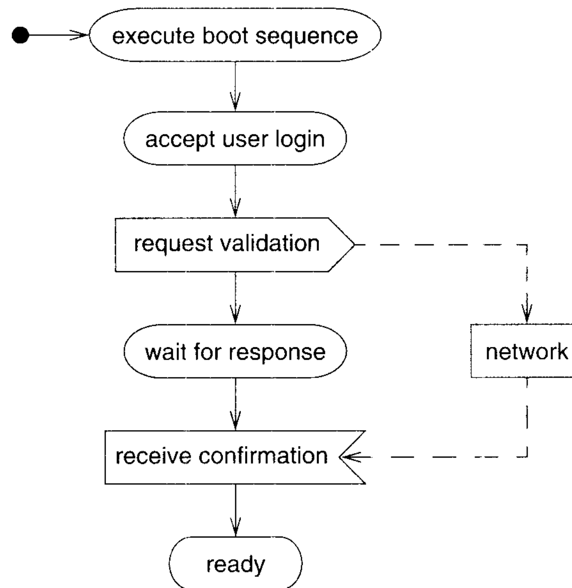


Figure 8.7 Activity diagram with signals. Activity diagrams can show fine control via sending and receiving events.

The UML shows the sending of a signal as a convex pentagon. When the preceding activity completes, the signal is sent, then the next activity is started. The UML shows the receiving of a signal as a concave pentagon. When the preceding activity completes, the receipt construct waits until the signal is received, then the next activity starts.

8.3.2 Swimlanes

In a business model, it is often useful to know which human organization is responsible for an activity. Sales, finance, marketing, and purchasing are examples of organizations. When the design of the system is complete, the activity will be assigned to a person, but at a high level it is sufficient to partition the activities among organizations.

You can show such a partitioning with an activity diagram by dividing it into columns and lines. Each column is called a ***swimlane*** by analogy to a swimming pool. Placing an activity within a particular swimlane indicates that it is performed by a person or persons within the organization. Lines across swimlane boundaries indicate interactions among different organizations, which must usually be treated with more care than interactions within an organization. The horizontal arrangement of swimlanes has no inherent meaning, although there may be situations in which the order has meaning.

Figure 8.8 shows a simple example for servicing an airplane. The flight attendants must clean the trash, the ground crew must add fuel, and catering must load food and drink before a plane is serviced and ready for its next flight.

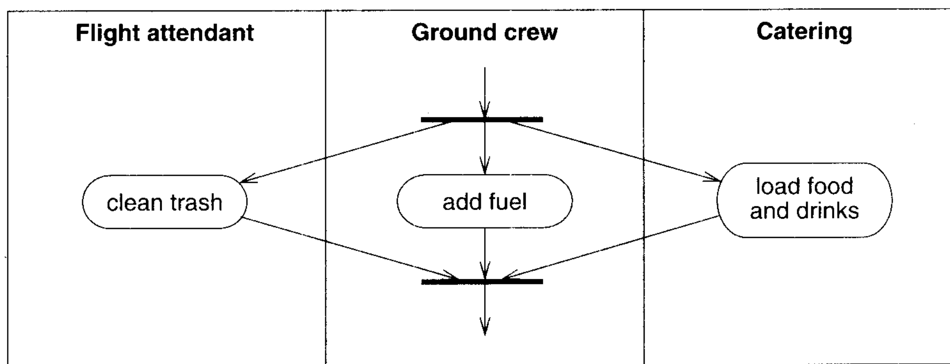


Figure 8.8 Activity diagram with swimlanes. Swimlanes can show organizational responsibility for activities.

8.3.3 Object Flows

Sometimes it is helpful to see the relationships between an operation and the objects that are its argument values or results. An activity diagram can show objects that are inputs to or outputs from the activities. An input or output arrow implies a control flow, therefore it is unnecessary to draw a control flow arrow where there is an object flow.

Frequently the same object goes through several states during the execution of an activity diagram. The same object may be an input to or an output from several activities, but on closer examination an activity usually produces or uses an object in a particular state. The UML shows an object value in a particular state by placing the state name in square brackets following the object name. If the objects have state names, the activity diagram shows both the flow of control and the progression of an object from state to state as activities act on it. In Figure 8.9 an airplane goes through several states as it leaves the gate, flies, and then lands again.

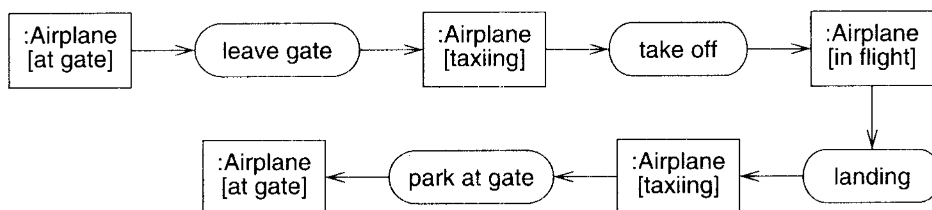


Figure 8.9 Activity diagram with object flows. An activity diagram can show the objects that are inputs or outputs of activities.

An activity diagram showing object flows among different object states has most of the advantages of a data flow diagram without most of their disadvantages. In particular, it unifies data flow and control flow, whereas data flow diagrams often separate them.

8.4 Chapter Summary

Independent use cases suffice for simple applications. However, it can be helpful to structure use cases for large applications using the include, extend, and generalization relationships. The include relationship incorporates one use case within the behavior sequence of another use case, like a subroutine call. The extend relationship adds incremental behavior to a base use case. Generalization can show specific variations on a general use case, analogous to generalization among classes. Don't use these relationships to excess. Remember that use cases are intended to be informal—use case relationships should only be used to structure major behavior units.

Sequence models are not only useful for fleshing out the interactions behind use cases, but they are also helpful for showing details of implementation. Not all objects in a sequence model need be active and exist for the entire computation. Some objects are passive and lack their own flow of control. Other objects are transient and may exist for only part of the duration of an operation.

Activity models also have additional notation that is helpful for large and complex applications. You can show fine controls via the sending and receiving of events that may interact with other objects that are not the focus of an activity diagram. You can augment activity diagrams with swimlanes to show the organizations that are responsible for different activities. And you can show the evolution of states of an object and how the states interleave with the flow of activities.

activation	passive object	use case extension
activity diagram	sequence diagram	use case generalization
focus of control	swimlane	use case inclusion
interaction model	transient object	
lifeline	use case	

Figure 8.10 Key concepts for Chapter 8

References

[Rumbaugh-05] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Boston: Addison-Wesley, 2005.

Exercises

8.1 Consider the purchase of gasoline from an electronic gasoline pump.

a. (4) Prepare a use case diagram. Normally the customer pays cash for a gas purchase. Add extend relationships to handle the incremental behavior of paying by credit card outside or paying by credit card inside. Add an include relationship to represent the optional purchase of a car wash.

b. (2) List and explain the relevance of each actor.

c. (2) Summarize the purpose of each use case with a sentence.

8.2 (5) You are interacting with an online travel agent and encounter the following use cases. Prepare a use case diagram, using the generalization and include relationships.

- **Purchase a flight**. Reserve a flight and provide payment and address information.
- **Provide payment information**. Provide a credit card to pay for the incurred charges.
- **Provide address**. Provide mailing and residence addresses.
- **Purchase car rental**. Reserve a rental car and provide payment and address information.
- **Purchase a hotel stay**. Reserve a hotel room and provide payment and address information.
- **Make a purchase**. Make a travel purchase and provide payment and address information.

8.3 (7) Consider an online frequent flyer program. Some use cases are listed below. Prepare a use case diagram and include the appropriate relationships for the use cases. You can add an abstract parent for each use case generalization.

- **View credits**. View the frequent flyer points currently available in the account.
- **Submit missing credit**. Request credit for an activity that was not credited.
- **Change address**. Submit a new mailing address.
- **Change user name**. Change the user name for the account.
- **Change password**. Change the password for the account.
- **Book a free flight**. Use frequent flyer credits to obtain a free flight.
- **Book a free hotel**. Use frequent flyer credits to obtain a free hotel.
- **Book a free rental car**. Use frequent flyer credits to obtain a free rental car.
- **Request a frequent flyer credit card**. Fill out an application for a credit card that gives frequent flyer points as a bonus for purchases.
- **Check prices and routes**. Find possible routings and corresponding prices for a paid flight.
- **Check availability for a free flight**. Check availability of free travel for a specified flight.

8.4 (8) Consider software that manages electronic music files. Some use cases are listed below. Prepare a use case diagram and include the appropriate relationships for the use cases. You can add an abstract parent for each use case generalization.

- **Play a song**. Add the song to the end of the play queue.
- **Play a library**. Add the songs in the library to the play queue.
- **Randomize order**. Randomly reorder the songs in the play queue.
- **Delete a song**. Delete a song from a music library.
- **Destroy a song**. Delete a song from all music libraries and delete the underlying file.
- **Add a song**. Add a music file to a music library.

- **Create a music library**. Create a new music library that contains no songs.
- **Delete a music library**. Delete the music library.
- **Destroy a music library**. Destroy all songs in the music library and then delete the music library.
- **Rip a CD**. Digitize the music on an analog CD.
- **Create a CD**. Burn an analog CD from a list of digital songs.
- **View songs by title**. Display the songs in a music library sorted by title.
- **View songs by artist**. Display the songs in a music library sorted by artist.
- **View songs by album**. Display the songs in a music library sorted by album.
- **View songs by genre**. Display the songs in a music library sorted by genre.
- **Start play**. Start playing songs from the queue. If previously stopped, resume playing from the last position, otherwise start playing at the start of the queue.
- **Stop play**. Suspend playing of music.

8.5 (8) Consider a simple payroll system. Prepare a use case diagram and include the appropriate relationships for the following use cases. You can add an abstract parent for each use case generalization.

- **Add deduction**. Add another deduction type for the employee and incorporate the deduction in subsequent paychecks.
- **Drop deduction**. Remove the deduction type for the employee.
- **Sum income**. Total all income for a paycheck.
- **Sum deductions**. Total all deductions for a paycheck.
- **Compute net take-home pay**. Compute the total income less the total deductions for a paycheck.
- **Compute charitable contributions**. Total all contributions to charity for a paycheck.
- **Compute taxes**. Compute all taxes paid for a paycheck.
- **Compute retirement savings**. Compute all contributions to retirement funds for a paycheck.
- **Compute other deductions**. Compute the total of all deductions, other than charity, taxes, and retirement for a paycheck.
- **Change employee name**. Change the name of the employee that is on record.
- **Change employee address**. Change the mailing address of the employee that is on record.
- **Compute base pay**. Compute the base pay of the employee for the paycheck.
- **Compute overtime pay**. Compute the overtime pay of the employee for the paycheck.
- **Compute other pay**. Compute all other income (other than base pay and overtime) of the employee for the paycheck.
- **Change method of payment**. Change the method of disbursing the paycheck, such as cash, direct deposit, and check.

8.6 (4) Consider stock management software that records all transactions that occur for a portfolio. For example, stocks may be purchased and sold. Dividend payments may be received. Complex situations can occur, such as stock splits.

The current contents of a portfolio can be determined by replaying the transaction log. The portfolio has some initial contents, and all subsequent changes are captured via the transaction

log. The changes in the transaction log are then applied through the target date to determine the current contents.

Construct a procedural sequence diagram to show the calculation of the contents of a portfolio as of some date. Limit the detail in your diagram to four message flows.

8.7 (5) Compute the value of a stock portfolio as of a specified date. First compute the contents of the portfolio (the previous exercise) and then multiply the quantity of each stock by its value on the specified date to determine the overall value of the portfolio.

8.8 (7) Once again compute the value of a stock portfolio as of a specified date. However, for this exercise a portfolio may contain stock and lesser portfolios. For simplicity, assume that a portfolio is at most three levels deep.

For example, portfolio *net worth* may contain portfolios *retirement funds* and *taxable account*. Portfolios *retirement funds* and *taxable account* contain only stocks.

8.9 (6) A customer decides to upgrade her PC and purchase a DVD player. She begins by calling the sales department of the PC vendor and they tell her to talk to customer support. She then calls customer support and they put her on hold while talking to engineering. Finally, customer support tells the customer about several supported DVD options. The customer chooses a DVD and it is shipped by the mail department. The customer receives the DVD, installs it satisfactorily, and then mails her payment to accounting.

Construct an activity diagram for this process. Use swimlanes to show the various interactions.

8.10 (6) A company is manufacturing a new product and must coordinate several departments. The product starts out as a raw marketing idea that goes to engineering. Engineering simulates the function of the product and prepares a design. Manufacturing reviews the design and adjusts it to conform to existing machinery. Engineering approves the revisions and customer service then looks at the design—a good design must enable ready repair. Engineering approves the customer service proposals and ensures that the resulting design still meets the target functionality.

Construct an activity diagram for this process. Use swimlanes to show the various interactions. Show the changes in the state of the design as the activity diagram proceeds.

9

Concepts Summary

We find it useful to model a system from three related but different viewpoints: the *class model*, describing the objects in the system and their relationships; the *state model*, describing the life history of objects; and the *interaction model*, describing the interactions among objects. A complete description requires all three models, but different problems place different emphasis. Each model applies during all stages of development and acquires detail as development progresses.

9.1 Class Model

The *class model* describes the static structure of objects in a system—their identity, their relationships to other objects, their attributes, and their operations. The class model provides the essential framework into which the state and interaction models can be placed. Changes and interactions are meaningless unless there is something to be changed or with which to interact. Objects are the units into which we divide the world, the molecules of our models.

The most important concepts in class models are classes, associations, and generalizations. A class describes a group of similar objects. An association describes a group of similar connections between objects. Generalization structures the description of objects by organizing classes by their similarities and differences. Attributes and operations are secondary and serve to elaborate the fundamental structure provided by classes, associations, and generalizations.

9.2 State Model

The *state model* describes those aspects of an object concerned with time—events that mark changes and states that define the context for events. Events represent external stimuli and states represent values of objects. Over time, the objects stimulate each other, resulting in a

series of changes to their states. The state model consists of multiple state diagrams, one state diagram for each class with important temporal behavior. The state diagrams must match on their interfaces—events and guard conditions. Each state diagram shows the state and event sequences permitted for one class of objects.

A state diagram specifies the possible states, which transitions are allowed between states, what stimuli cause the transitions to occur, and what operations are executed in response to stimuli. A state diagram describes the collective behavior for the objects in a class. As each object has its own values and links, so too each object has its own state or position in the state diagram.

9.3 Interaction Model

The *interaction model* describes how objects collaborate to achieve 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. Use cases represent pieces of functionality and are helpful for capturing informal requirements. Sequence diagrams provide more detail and show the objects that interact and the time sequence of their interactions. Activity diagrams provide the finest detail and show the flow of control among the processing steps of a computation.

9.4 Relationship Among the Models

The class, state, and interaction models all involve the same concepts—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 and calls).

Each model describes one aspect of the system but contains references to the other models. The class model describes data structure on which the state and interaction models operate. The operations in the class model correspond to events, conditions, and activities. The state model describes the control structure of objects. It shows decisions that depend on object values; the decisions cause changes in object values and subsequent states. The interaction model focuses on the exchanges between objects and provides a holistic overview of the operation of a system.

Generalization and aggregation are relationships that cut across the models, and we will now examine their usage.

9.4.1 Generalization

Generalization appears in all three models. Generalization is the "or-relationship" and can show specific variations on a general situation. In UML 2.0, inheritance applies to classifiers, and classes, signals, and use cases are all classifiers.

- **Class generalization**. Generalization organizes classes by their similarities and differences. A subclass inherits the attributes, operations, associations, and state diagrams of its superclasses. Subclasses can reuse inherited properties from a superclass or override them; subclasses can add new properties.

  A subclass inherits the state diagrams of its ancestors, to be concurrent with any state diagram that it defines. A subclass inherits both the states of its ancestors and the transitions. To avoid confusion, subclass state diagrams should normally be an orthogonal addition to the state diagram from the superclass.

  The class model supports multiple inheritance—a class may inherit from more than one superclass. For simplicity, we normally disallow multiple inheritance for signals and use cases.

- **Signal generalization**. A generalization hierarchy can also organize signals with inheritance of signal attributes. Ultimately you can regard every actual signal as a leaf on a generalization tree of signals. An input signal triggers transitions on any ancestor signal type.

- **Use case generalization**. Generalization also applies to use cases. A parent use case represents a general behavior sequence. Child use cases specialize the parent by inserting additional steps or by refining steps. In one respect, use case generalization is more complicated than class generalization. A subclass adds attributes to the parent class, but their order is unimportant. A child use case adds behavior steps, but they must appear in the proper locations within the behavior sequence of the parent.

With inheritance a parent classifier may be abstract or concrete. However, we recommend that you consider only abstract parents and forego concrete ones. Then abstract and concrete classifiers are readily apparent at a glance; all superclassifiers are abstract and all leaf subclassifiers are concrete. Classifiers also exhibit polymorphism—a child classifier can freely substitute for a parent classifier.

The first edition of this book also supported inheritance of states, but this has been disallowed in UML2 because a state is not a classifier. There are similarities between generalization of classifiers and nesting of states, but strictly speaking, in UML2 there is no state generalization.

9.4.24 Aggregation

Aggregation is the "and-relationship" and breaks an assembly into orthogonal parts that have limited interaction.

- **Object aggregation**. Aggregation is a special form of association with additional properties, most notably transitivity and antisymmetry. The UML has two forms of object aggregation: a general form called aggregation (a constituent part is reusable and may

exist apart from an assembly) and a more restrictive form called composition (the constituent part can belong to at most one assembly and has a coincident lifetime).

A state diagram for an assembly is a collection of state diagrams, one for each part. The aggregate state corresponds to the combined states of all the parts. The aggregate state is one state from the first diagram, *and* a state from the second diagram, *and* a state from each other diagram. In the more interesting cases, the part states interact.

- **State aggregation**. Some states can be partitioned into lesser states, each operating independently and each having its own subdiagram. The state of the object comprises one state from each subdiagram.

This completes our treatment of concepts and notation for object-oriented modeling.

Part 2

Analysis and Design

Chapter 10	Process Overview	167
Chapter 11	System Conception	173
Chapter 12	Domain Analysis	181
Chapter 13	Application Analysis	216
Chapter 14	System Design	240
Chapter 15	Class Design	270
Chapter 16	Process Summary	298

Part 1 covers *concepts*, specifically the concepts and notation for the class, state, and interaction models. We now shift our focus in Parts 2 and 3 and present a *process* for devising the models. Part 1 discusses *what* constitutes a model; Parts 2 and 3 explain *how to* formulate a model. Our treatment of process is language independent and applies equally well to OO languages, non-OO languages, and databases.

Chapter 10 provides an overview of the process for building models and emphasizes that development is normally iterative and seldom a rigid sequence of steps.

Chapter 11 presents the first stage of development—system conception—during which a visionary conceives an application and sells the idea to an organization.

Once you have a concept for an application, you elaborate and refine the concept by building models as Chapters 12 and 13 explain. First build a domain model that focuses on the real-world things that carry the semantics of the application. Then build an application model that addresses the computer aspects of the application that are visible to users.

The analysis models give you a thorough understanding of an application. The next stage is to address the practicalities of realizing the models. Chapter 14 covers system design, in which you devise a high-level strategy for building a solution. Chapter 15 covers class design, in which you flesh out the details for classes, associations, and operations.

Chapter 16 concludes Part 2 by summarizing the analysis and design portion of the development process.

After reading Part 2, you will understand the basics of how to prepare OO models. You will not be an expert, but you will have a good start on learning a valuable software development skill. You will be ready to study implementation and software engineering in the final two parts.

10

Process Overview

A ***software development process*** provides a basis for the organized production of software, using a collection of predefined techniques and notations. The process in this book starts with formulation of the problem, then continues through analysis, design, and implementation. The presentation of the stages is linear, but the actual process is seldom linear.

10.1 Development Stages

Software development has a sequence of well-defined stages, each with a distinct purpose, input, and output.

- **System conception**. Conceive an application and formulate tentative requirements.
- **Analysis**. Deeply understand the requirements by constructing models. The goal of analysis is to specify *what* needs to be done, not *how* it is done. You must understand a problem before attempting a solution.
- **System design**. Devise a high-level strategy—the architecture—for solving the application problem. Establish policies to guide the subsequent class design.
- **Class design**. Augment and adjust the real-world models from analysis so that they are amenable to computer implementation. Determine algorithms for realizing the operations.
- **Implementation**. Translate the design into programming code and database structures.
- **Testing**. Ensure that the application is suitable for actual use and that it truly satisfies the requirements.
- **Training**. Help users master the new application.
- **Deployment**. Place the application in the field and gracefully cut over from legacy applications.
- **Maintenance**. Preserve the long-term viability of the application.

The entire process is seamless. You continually elaborate and optimize models as your focus shifts from analysis to design to implementation. Throughout development the same concepts and notation apply; the only difference is the shift in perspective from the initial emphasis on business needs to the later emphasis on computer resources.

An OO approach moves much of the software development effort up to analysis and design. It is sometimes disconcerting to spend more time during analysis and design, but this extra effort is more than compensated by faster and simpler implementation. Because the resulting design is cleaner and more adaptable, future changes are much easier.

Part 2 covers the first four topics and Part 3 covers implementation. In this book we emphasize development and only briefly consider testing, training, deployment, and maintenance. These last four topics are important, but are not the focus of this book.

10.1.1 System Conception

System conception deals with the genesis of an application. Initially somebody thinks of an idea for an application, prepares a business case, and sells the idea to the organization. The innovator must understand both business needs and technological capabilities.

10.1.2 Analysis

Analysis focuses on creation of models. Analysts capture and scrutinize requirements by constructing models. They specify *what* must be done, not *how* it should be done. Analysis is a difficult task in its own right, and developers must fully understand the problem before addressing the additional complexities of design. Sound models are a prerequisite for an extensible, efficient, reliable, and correct application. No amount of implementation patches can repair an incoherent application and compensate for a lack of forethought.

During analysis, developers consider the available sources of information (documents, business interviews, related applications) and resolve ambiguities. Often business experts are not sure of the precise requirements and must refine them in tandem with software development. Modeling quickens the convergence between developers and business experts, because it is much faster to work with multiple iterations of models than with multiple implementations of code. Models highlight omissions and inconsistencies so that they can be resolved. As developers elaborate and refine a model, it gradually becomes coherent.

There are two substages of analysis: domain analysis and application analysis. ***Domain analysis*** focuses on real-world things whose semantics the application captures. For example, an airplane flight is a real-world object that a flight reservation system must represent. Domain objects exist independently of any application and are meaningful to business experts. You find them during domain analysis or by prior knowledge. Domain objects carry information about real-world objects and are generally passive—domain analysis emphasizes concepts and relationships, with much of the functionality being implicit in the class model. The job of constructing a domain model is mainly to decide which information to capture and how to represent it.

Domain analysis is then followed by ***application analysis*** that addresses the computer aspects of the application that are visible to users. For example, a flight reservation screen is

part of a flight reservation system. Application objects do not exist in the problem domain and are meaningful only in the context of an application. Application objects, however, are not merely internal design decisions, because the users see them and must agree with them. The application model does not prescribe the implementation of the application. It describes how the application appears from the outside—the black-box view of it. You cannot find application classes with domain analysis, but you can often reuse them from previous applications. Otherwise, you must devise application objects during analysis as you think about interfaces with other systems and how your application interacts with users.

10.1.3 System Design

During ***system design***, the developer makes strategic decisions with broad consequences. You must formulate an architecture and choose global strategies and policies to guide the subsequent, more detailed portion of design. The ***architecture*** is the high-level plan or strategy for solving the application problem. The choice of architecture is based on the requirements as well as past experience. If possible, the architecture should include an executable skeleton that can be tested. The system designer must understand how a new system interacts with other systems. The architecture must also support future modification of the application.

For straightforward problems, preparation of the architecture follows analysis. However, for large and complex problems their preparation must be interleaved. The architecture helps to establish a model's scope. In turn, modeling reveals important issues of strategy to resolve. For large and complex problems, there is much interplay between the construction of a model and the model's architecture, and they must be built together.

10.1.4 Class Design

During ***class design***, the developer expands and optimizes analysis models; there is a shift in emphasis from application concepts toward computer concepts. Developers choose algorithms to implement major system functions, but they should continue to defer the idiosyncrasies of particular programming languages.

10.1.5 Implementation

Implementation is the stage for writing the actual code. Developers map design elements to programming language and database code. Often, tools can generate some of the code from the design model.

10.1.6 Testing

After implementation, the system is complete, but it must be carefully tested before being commissioned for actual use. The ideas that inspired the original project should have been nurtured through the previous stages by the use of models. Testers once again revisit the original business requirements and verify that the system delivers the proper functionality. Testing can also uncover accidental errors (bugs) that have been introduced. If an application runs on multiple hardware and operating system platforms, it should be tested on all of them.

Developers should check a program at several levels. Unit tests exercise small portions of code, such as methods or possibly entire classes. Unit tests discover local problems and often require that extra instrumentation be built into the code. System tests exercise a major subsystem or the entire application. In contrast to unit tests, system tests can discover broad failures to meet specifications. Both unit and system tests are necessary. Testing should not wait until the entire application is coded. It must be planned from the beginning, and many tests can be performed during implementation.

10.1.7 Training

An organization must train users so that they can fully benefit from an application. Training accelerates users on the software learning curve. A separate team should prepare user documentation in parallel to the development effort. Quality control can then check the software against the user documentation to ensure that the software meets its original goals.

10.1.8 Deployment

The eventual system must work in the field, on various platforms and in various configurations. Unexpected interactions can occur when a system is deployed in a customer environment. Developers must tune the system under various loads and write scripts and install procedures. Some customers will require software customizations. Staff must also localize the product to different spoken languages and locales. The result is a usable product release.

10.1.9 Maintenance

Once development is complete and a system has been deployed, it must be maintained for continued success. There are several kinds of maintenance. Bugs that remain in the original system will gradually appear during use and must be fixed. A successful application will also lead to enhancement requests and a long-lived application will occasionally have to be restructured.

Models ease maintenance and transitions across staff changes. A model expresses the business intent for an application that has been driven into the programming code, user interface, and database structure.

10.2 Development Life Cycle

An OO approach to software development supports multiple life-cycle styles. You can use a waterfall approach performing the phases of analysis, design, and implementation in strict sequence for the entire system. However, we typically recommend an iterative development strategy. We summarize the distinction here and elaborate in Chapter 21.

10.2.1 Waterfall Development

The waterfall approach dictates that developers perform the software development stages in a rigid linear sequence with no backtracking. Developers first capture requirements, then construct an analysis model, then perform a system design, then prepare a class design, fol-

lowed by implementation, testing, and deployment. Each stage is completed in its entirety before the next stage is begun.

The waterfall approach is suitable for well-understood applications with predictable outputs from analysis and design, but such applications seldom occur. Too many organizations attempt to follow a waterfall when requirements are fluid. This leads to the familiar situation where developers complain about changing requirements, and the business complains about inflexible software development. A waterfall approach also does not deliver a useful system until completion. This makes it difficult to assess progress and correct a project that has gone awry.

10.2.2 Iterative Development

Iterative development is more flexible. First you develop the nucleus of a system—analyzing, designing, implementing, and delivering working code. Then you grow the scope of the system, adding properties and behavior to existing objects, as well as adding new kinds of objects. There are multiple iterations as the system evolves to the final deliverable.

Each iteration includes a full complement of stages: analysis, design, implementation, and testing. Unlike the strict sequence of the waterfall method, iterative development can interleave the different stages and need not construct the entire system in lock step. Some parts may be completed early, while other, less crucial parts are completed later. Each iteration ultimately yields an executable system that can be integrated and tested. You can accurately gauge progress and make adjustments to your plans based on feedback from the early iterations. If there is a problem, you can move backward to an earlier stage for rework.

Iterative development is the best choice for most applications because it gracefully responds to change and minimizes risk of failure. Management and business users get early feedback about progress.

10.3 Chapter Summary

A software engineering process provides a basis for the organized production of software. There is a sequence of well-defined stages that you can apply to each of the pieces of a system. For example, parallel development teams might develop a database design, key algorithms, and a user interface. An iterative development of software is flexible and responsive to evolving requirements. First you prepare a nucleus of a system, and then you successively grow its scope until you realize the final desired software.

analysis	domain analysis	system conception
application analysis	implementation	system design
architecture	iterative development	testing
class design	life cycle	training
deployment	maintenance	waterfall development

Figure 10.1 Key concepts for Chapter 10

Bibliographic Notes

The *class design* stage is renamed from *object design* in the first edition of this book.

Exercises

10.1 (2) It seems there is never enough time to do a job right the first time, but there is always time to do it over. Discuss how the approach presented in this chapter overcomes this tendency of human behavior. What kinds of errors do you make if you rush into the implementation phase of a software project? Compare the effort required to prevent errors with that needed to detect and correct them.

10.2 (4) This book explains how to use OO techniques to implement programs and databases. Discuss how OO techniques could be applied in other areas, such as language design, knowledge representation, and hardware design.

11

System Conception

System conception deals with the genesis of an application. Initially some person, who understands both business needs and technology, thinks of an idea for an application. Developers must then explore the idea to understand the needs and devise possible solutions. The purpose of system conception is to defer details and understand the big picture—what need does the proposed system meet, can it be developed at a reasonable cost, and will the demand for the result justify the cost of building it?

This chapter introduces the automated teller machine (ATM) case study that threads throughout the remainder of the book.

11.1 Devising a System Concept

Most ideas for new systems are extensions of existing ideas. For example, a human relations department may have a database of employee benefit choices and require that a clerk enter changes. An obvious extension is to allow employees to view and enter their own changes. There are many issues to resolve (security, reliability, privacy, and so on), but the new idea is a straightforward extension of an existing concept.

Occasionally a new system is a radical departure from the past. For example, an online auction automates the ancient idea of buyers bidding against each other for products, but the first online auction systems were brand new software. The concept became feasible when several enabling technologies came into place: the Internet, widespread personal computer access, and reliable servers. The large customer base and low unit cost due to automation changed the nature of auctions—an online auction can sell inexpensive items and still make a profit. In addition, online systems have made the auction process concurrent and distributed.

Here are some ways to find new system concepts.

- **New functionality**. Add functionality to an existing system.

- **Streamlining**. Remove restrictions or generalize the way a system works.
- **Simplification**. Let ordinary persons perform tasks previously assigned to specialists.
- **Automation**. Automate manual processes.
- **Integration**. Combine functionality from different systems.
- **Analogies**. Look for analogies in other problem domains and see if they have useful ideas.
- **Globalization**. Travel to other countries and observe their cultural and business practic-es.

11.2 Elaborating a Concept

Most systems start as vague ideas that need more substance. A good system concept must answer the following questions.

- **Who is the application for?** You should clearly understand which persons and organizations are stakeholders of the new system. Two of the most important kinds of stakeholders are the financial sponsors and the end users.

  The financial sponsor are important because they are paying for the new system. They expect the project to be on schedule and within budget. You should get the financial sponsors to agree to some measure of success. You need to know when the system is complete and meets their expectations.

  The users are also stakeholders, but in another sense. The users will ultimately determine the success of the new system by an increase (or decrease) in their productivity or effectiveness. Users can help you if they are receptive and provide critical comments. They can improve your system by telling you what is missing and what could be improved. In general, users will not consider new software unless they have a compelling interest—either personal or business. You should try to help them find a vested interest in your project so that you can obtain their buy-in. If you cannot get their buy-in, you should question the need for the project and reconsider doing it.
- **What problems will it solve?** You must clearly bound the size of the effort and establish its scope. You should determine which features will be in the new system and which will not. You must reach various kinds of users in different organizations with their own viewpoints and political motivations. You must not only decide which features are appropriate, but you must also obtain the agreement of influential persons.
- **Where will it be used?** At this early stage, it is helpful to get a general idea of where the new system might be used. You should determine if the new system is mission-critical software for the organization, experimental software, or a new capability that you can deploy without disrupting the workflow. You should have a rough idea about how the new system will complement the existing systems. It is important to know if the software will be used locally or will be distributed via a network. For a commercial product, you should characterize the customer base.

- **When is it needed?** Two aspects of time are important. The first is the *feasible* time, the time in which the system can be developed within the constraints of cost and available resources. The other is the *required* time, when the system is needed to meet business goals. You must make sure that the timing expectations driven by technical feasibility are consistent with the timing the business requires. If there is a disconnect, you must initiate a dialogue between technologists and business experts to reach a solution.
- **Why is it needed?** You may need to prepare a business case for the new system if someone has not already done so. The business case contains the financial justification for the new system, including the cost, tangible benefits, intangible benefits, risk, and alternatives. You must be sure that you clearly understand the motivation for the new system. The business case will give you insight into what stakeholders expect, roughly indicate the scope, and may even provide information for seeding your models. For a commercial product, you should estimate the number of units that can be sold and determine a reasonable selling price; the revenue must cover costs and a profit.
- **How will it work?** You should brainstorm about the feasibility of the problem. For large systems you should consider the merits of different architectures. The purpose of this speculation is not to choose a solution, but to increase confidence that the problem can be solved reasonably. You might need some prototyping and experimentation.

11.2.1 The ATM Case Study

Figure 11.1 lists our original system concept for an Automated Teller Machine (ATM). We ask high-level questions to elaborate the initial concept.

Develop software so that customers can access a bank's computers and carry out their own financial transactions without the mediation of a bank employee.

Figure 11.1 System concept for an automated teller machine

- **Who is the application for?** A number of companies provide ATM products. Consequently, only a vendor or a large financial company could possibly justify the cost and effort of building ATM software.

  A vendor would be competing for customers in an established market. A large vendor could certainly enter such a market, but might find it advantageous to partner with or acquire an existing supplier. A small vendor would need some special feature to differentiate itself from the crowd and attract attention.

  It is unlikely that a financial company could justify developing ATM software just for its own use, because it would probably be more expensive than purchasing a product. If a financial company wanted special features, it could partner with a vendor. Or it might decide to create a separate organization that would build the software, sell it to the sponsoring company, and then market it to others.

For the ATM case study, we will assume that we are a vendor building the software. We will assume that we are developing an ordinary product, since deep complexities of the ATM problem domain are beyond the scope of this book.

- **What problems will it solve?** The ATM software is intended to serve both the bank and the customer. For the bank, ATM software increases automation and reduces manual handling of routine paperwork. For the customer, the ATM is ubiquitous and always available, handling routine transactions whenever and wherever the customer desires. ATM software must be easy to use and convenient so that customers will use it in preference to bank tellers. It must be reliable and secure since it will be handling money.
- **Where will it be used?** ATM software has become essential to financial institutions. Customers take it for granted that a bank will have an ATM machine. ATM machines are available at many stores, sporting events, and other locations throughout the world.
- **When is it needed?** Any software development effort is a financial proposition. The investment in development ultimately leads to a revenue stream. From an economic perspective, it is desirable to minimize the investment, maximize the revenue, and realize revenue as soon as possible. Thoughtful modeling and OO techniques are conducive to this goal.
- **Why is it needed?** There are many reasons why a vendor might decide to build a software product. If other companies are making money with similar products, there is an economic incentive to participate. A novel product could outflank competitors and lead to premium pricing. Businesses commission internal efforts for technology that is difficult to buy and critical to them. We have no real motivation to develop ATM software, other than to demonstrate the techniques in this book.
- **How will it work?** We will adopt a three-tier architecture to separate the user interface from programming logic, and programming logic from the database. In reality, the architecture is *n*-tier, because there can be any number of intermediate programming levels communicating with each other. We will discuss architecture further in the *System Design* chapter.

11.3 Preparing a Problem Statement

Once you have fleshed out the raw idea by answering the high-level questions, you are ready to write a requirements statement that outlines the goals and general approach of the desired system.

Throughout development, you should distinguish among requirements, design, and implementation. Requirements describe how a system behaves from the user's point of view. The system is considered as a black box—all we care about is its external behavior. For example, some requirements for a car are that when you press on the accelerator pedal, the car goes faster, and when you step on the brake, the car slows down. Design decisions are engineering choices that provide the behavior specified by the requirements. For example, some design decisions are how the internal linkages are routed, how the engine is controlled, and

what kinds of brake pads are on the wheels. Implementation deals with the ultimate realization in programming code.

Frequently customers mix true requirements with design decisions. Usually this is a bad idea. If you separate requirements from design decisions, you preserve the freedom to change a design. Typically there are many possible ways to design a system, and you should defer a solution until you fully understand a problem.

A system concept document may include an example implementation. The purpose of the example is to show how the system could be implemented using current technology at a reasonable cost. It is a "proof of existence" statement. However, make it clear that the sample implementation could be done differently in the final system. The sample implementation is merely proposed as a possibility.

For example, when the Apollo program to put a man on the moon in the 1960s was first proposed, the plan was to place a rocket in earth orbit, then launch a landing vehicle directly to the moon's surface. In the final successful program, the rocket was launched directly into a lunar orbit, from which the lander was launched to the moon's surface. It was not a bad thing to make the first proposal, however, as this gave confidence that there was a feasible approach.

As Figure 11.2 shows, the problem statement should state what is to be done and not how it is to be implemented. It should be a statement of needs, not a proposal for a system architecture. The requestor should avoid describing system internals, as this restricts development flexibility. Performance specifications and protocols for interaction with external systems are legitimate requirements. Software engineering standards, such as modular construction, design for testability, and provision for future extensions, are also proper.

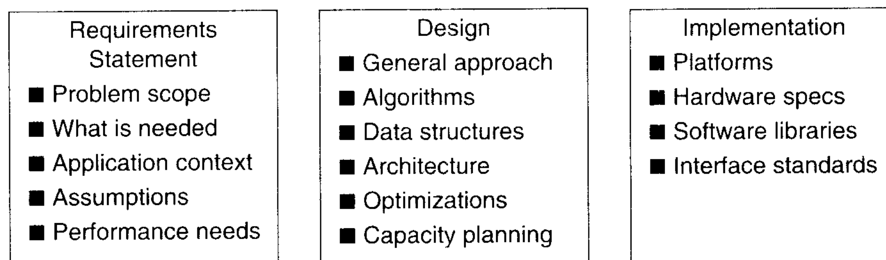


Figure 11.2 Kinds of requirements. Do not make early design and implementation decisions or you will compromise development.

A problem statement may have more or less detail. A requirement for a conventional product, such as a payroll program or a billing system, may have considerable detail. A requirement for a research effort in a new area may lack details, but presumably the research has some objective that should be clearly stated.

Most problem statements are ambiguous, incomplete, or even inconsistent. Some requirements are just plain wrong. Some requirements, although precisely stated, have unpleasant consequences on the system behavior or impose unreasonable implementation costs. Some requirements do not work out as well as the requestor thought. The problem

statement is just a starting point for understanding the problem, not an immutable document. The purpose of the subsequent analysis (next chapter) is to fully understand the problem and its implications. There is no reason to expect that a problem statement prepared without a full analysis will be correct.

11.3.1 The ATM Case Study

Figure 11.3 shows a problem statement for an automated teller machine (ATM) network.

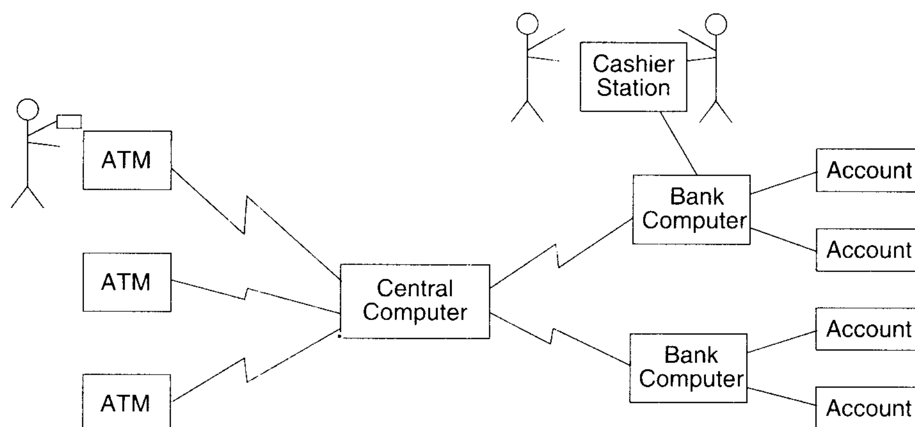


Figure 11.3 ATM network. The ATM case study threads throughout the remainder of this book.

> Design the software to support a computerized banking network including both human cashiers and automatic teller machines (ATMs) to be shared by a consortium of banks. Each bank provides its own computer to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computers. Human cashiers enter account and transaction data.
>
> Automatic teller machines communicate with a central computer that clears transactions with the appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The system requires appropriate recordkeeping and security provisions. The system must handle concurrent accesses to the same account correctly.
>
> The banks will provide their own software for their own computers; you are to design the software for the ATMs and the network. The cost of the shared system will be apportioned to the banks according to the number of customers with cash cards.

11.4 Chapter Summary

The first stage of a project is to devise a new idea. The idea can involve a new system or an improvement to an existing system. Before investing time and money into development, it is

necessary to evaluate the feasibility of the system, the difficulty and risk of developing it, the demand for the system, and the cost-benefit ratio. This process should consider the viewpoints of all the stakeholders of the system and should make the trade-offs necessary to provide a good chance of success, not just technical success, but also business success. This process usually results in some adjustments to the original idea. When the system conception stage is complete, write a problem statement that serves as the starting point for analysis. The problem statement need not be complete, and it will change during development, but the writing of the statement helps to focus the attention of the project.

business case	problem statement
cost-benefit trade-off	requirement
design decision	stakeholder
implementation constraint	system conception

Figure 11.4 Key concepts for Chapter 11

Exercises

11.1 (3) Consider a new antilock braking system for crash avoidance in an automobile. Elaborate the following high-level questions and explain your answers.
  a. Who is the application for? Who are the stakeholders? Estimate how many persons in your country are potential customers.
  b. Identify three features that should be included and three features that should be omitted.
  c. Identify three systems with which it must work.
  d. What are two of the largest risks?

11.2 (3) Repeat Exercise 11.1 for software that supports Internet selling of books.

11.3 (3) Repeat Exercise 11.1 for software that supports the remodeling of kitchens.

11.4 (3) Repeat Exercise 11.1 for an online auction system.

11.5 (4) Prepare a problem statement, similar to that for the ATM system, for each of the following systems. You may limit the scope of the system, but be precise and avoid making implementation decisions. Use 75–150 words per specification.
  a. bridge player
  b. change-making machine
  c. car cruise control
  d. electronic typewriter
  e. spelling checker
  f. telephone answering machine

11.6 (3) Rephrase the following requirements to make them more precise. Remove any design decisions posing as requirements:
  a. A system to transfer data from one computer to another over a telecommunication line. The system should transmit data reliably over noisy channels. Data must not be lost if the receiv-

ing end cannot keep up or if the line drops out. Data should be transmitted in packets, using a master–slave protocol in which the receiving end acknowledges or negatively acknowledges all exchanges.

b. A system for automating the production of complex machined parts. The parts will be designed using a three–dimensional drafting editor that is part of the system. The system will produce tapes that can be used by numerical control (N/C) machines to actually produce the parts.

c. A desktop publishing system, based on a what-you-see-is-what-you-get philosophy. The system will support text and graphics. Graphics include lines, squares, rectangles, polygons, circles, and ellipses. Internally, a circle is represented as a special case of an ellipse and a square as a special case of a rectangle. The system should support interactive, graphical editing of documents.

d. A system for generating nonsense. The input is a sample document. The output is random text that mimics the input text by imitating the frequencies of combinations of letters of the input. The user specifies the order of the imitation and the length of the desired output. For order N, every output sequence of N characters is found in the input and at approximately the same frequency. As the order increases, the style of the output more closely matches the input.

   The system should generate its output with the following method: Select a position at random in the document being imitated. Scan forward in the input text until a sequence of characters is found that exactly matches the last $N - 1$ characters of the output. If you reach the end of the input, continue scanning from the beginning. When a match is found, copy the letter that follows the matched sequence from the input to the output. Repeat until the desired amount of text is generated.

e. A system for distributing electronic mail over a network. Each user of the system should be able to send mail from any computer account and receive mail on one designated account. There should be provisions for answering or forwarding mail, as well as saving messages in files or printing them. Also, users should be able to send messages to several other users at once through distribution lists. Each computer on the net should hold any messages destined for computers that are down.