# 5 Use Case View

## Overview

The use case view captures the behavior of a system, subsystem, or class as it appears to an outside user. It partitions the system functionality into transactions meaningful to actors—idealized users of a system. The pieces of interactive functionality are called use cases. A use case describes an interaction with actors as a sequence of messages between the system and one or more actors. The term *actor* includes humans, as well as other computer systems and processes. Figure 5-1 shows a use case diagram for a telephone catalog sales application. The model has been simplified as an example.

## Actor

An actor is an idealization of an external person, process, or thing interacting with a system, subsystem, or class. An actor characterizes the interactions that outside users may have with the system. At run time, one physical user may be bound to multiple actors within the system. Different users may be bound to the same actor and therefore represent multiple instances of the same actor definition.

Each actor participates in one or more use cases. It interacts with the use case (and therefore with the system or class that owns the use case) by exchanging messages. The internal implementation of an actor is not relevant in the use case; an actor may be characterized sufficiently by a set of attributes that define its state.

Actors may be defined in generalization hierarchies, in which an abstract actor description is shared and augmented by one or more specific actor descriptions.

An actor may be a human, another computer system, or some executable process.

An actor is drawn as a small stick person with the name below it.

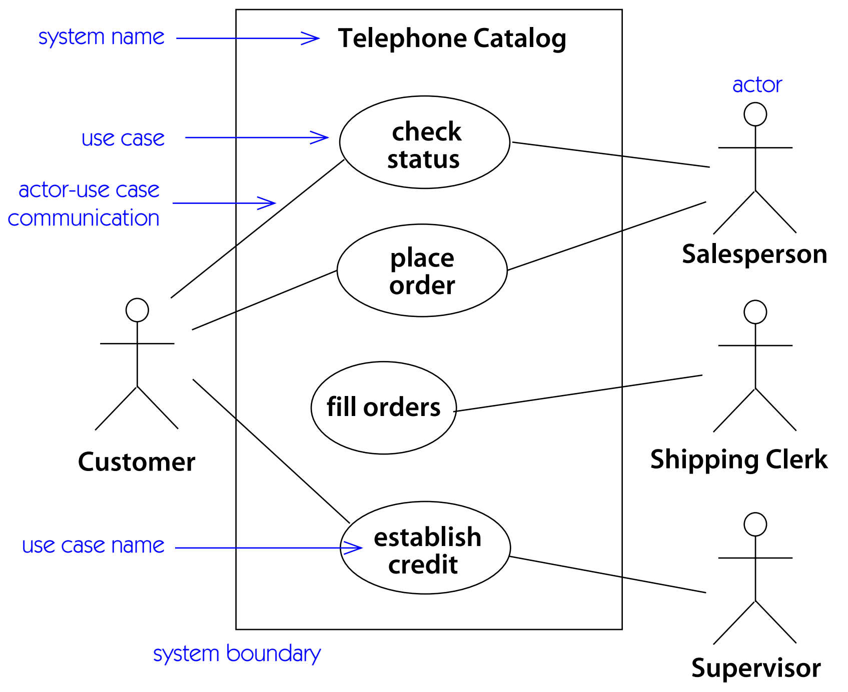


**Figure 5-1.** *Use case diagram*

## Use Case

A use case is a coherent unit of externally visible functionality provided by a system unit and expressed by sequences of messages exchanged by the system unit and one or more actors of the system unit. The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system. The definition of a use case includes all the behavior it entails—the mainline sequences, different variations on normal behavior, and all the exceptional conditions that can occur with such behavior, together with the desired response. From the user's point of view, these may be abnormal situations. From the system's point of view, they are additional variations that must be described and handled.

In the model, the execution of each use case is independent of the others, although an implementation of the use cases may create implicit dependencies among them due to shared objects. Each use case represents an orthogonal piece of functionality whose execution can be mixed with the execution of other use cases.

The dynamics of a use case may be specified by UML interactions, shown as statechart diagrams, sequence diagrams, collaboration diagrams, or informal text descriptions. When use cases are implemented, they are realized by collaborations among classes in the system. One class may participate in multiple collaborations and therefore in multiple use cases.

At the system level, use cases represent external behavior of the entire system as visible to outside users. A use case is like a system operation, an operation invocable by an outside user. Unlike an operation, however, a use case can continue to receive input from its actors during its execution. Use cases can also be applied internally to smaller units of a system, such as subsystems and individual classes. An internal use case represents behavior that a part of the system presents to the rest of the system. For example, a use case for a class represents a coherent chunk of functionality that a class provides to other classes that play certain roles within the system. A class can have more than one use case.

A use case is a logical description of a slice of system functionality. It is not a manifest construct in the implementation of a system. Instead, each use case must be mapped onto the classes that implement a system. The behavior of the use case is mapped onto the transitions and operations of the classes. Inasmuch as a class can play multiple roles in the implementation of a system, it may therefore realize portions of multiple use cases. Part of the design task is to find implementation classes that cleanly combine the proper roles to implement all the use cases, without introducing unnecessary complications. The implementation of a use case can be modeled as a set of one or more collaborations. A collaboration is a realization of a use case.

A use case can participate in several relationships, in addition to association with actors (Table 5-1).

**Table 5-1:** *Kinds of Use Case Relationships*

| *Relationship* | *Function* | *Notation* |
|---|---|---|
| association | The communication path between an actor and a use case that it participates in | ——— |
| extend | The insertion of additional behavior into a base use case that does not know about it | «extend» - - - - -> |
| use case generalization | A relationship between a general use case and a more specific use case that inherits and adds features to it | ———▷ |
| include | The insertion of additional behavior into a base use case that explicitly describes the insertion | «include» - - - - -> |

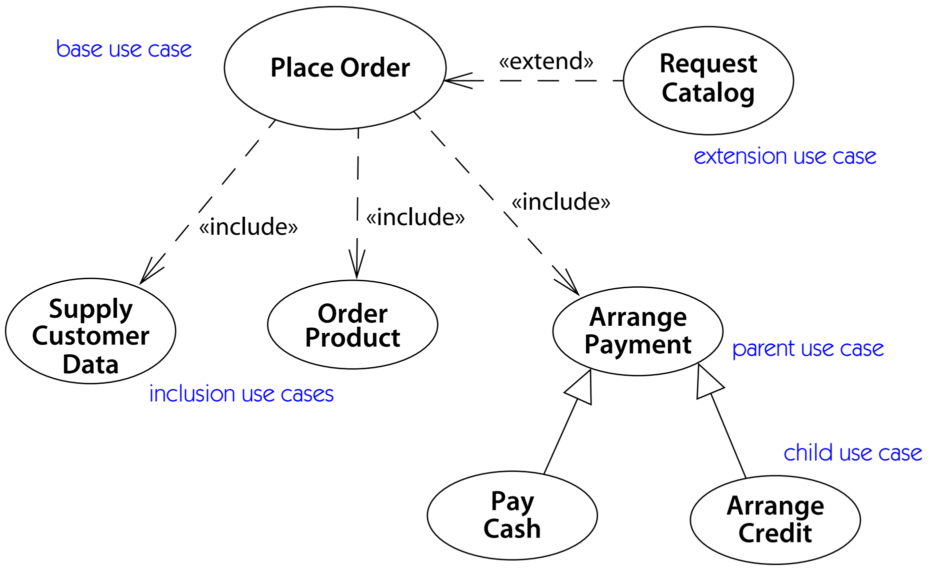


**Figure 5-2.** *Use case relationships*

A use case is drawn as an ellipse with its name inside or below it. It is connected by solid lines to actors that communicate with it.

Although each use case instance is independent, the description of a use case can be factored into other, simpler use cases. This is similar to the way the description of a class can be defined incrementally from the description of a superclass. A use case can simply incorporate the behavior of other use cases as fragments of its own behavior. This is called an include relationship. In this case, the new use case is not a special case of the original use case and cannot be substituted for it.

A use case can also be defined as an incremental extension to a base use case. This is called an extend relationship. There may be several extensions of the same base use case that may all be applied together. The extensions to a base use case add to its semantics; it is the base use case that is instantiated, not the extension use cases.

The include and extend relationships are drawn as dashed arrows with the keyword «include» or «extend». The include relationship points at the use case to be included; the extend relationship points at the use case to be extended.

A use case can also be specialized into one or more child use cases. This is use case generalization. Any child use case may be used in a situation in which the parent use case is expected.

Use case generalization is drawn the same as any generalization, as a line from the child use case to the parent use case with a large triangular arrowhead on the parent end. Figure 5-2 shows use case relationships in the catalog sales application.

# 6 State Machine View

## Overview

The state machine view describes the dynamic behavior of objects over time by modeling the lifecycles of objects of each class. Each object is treated as an isolated entity that communicates with the rest of the world by detecting events and responding to them. Events represent the kinds of changes that an object can detect—the receipt of calls or explicit signals from one object to another, a change in certain values, or the passage of time. Anything that can affect an object can be characterized as an event. Real-world happenings are modeled as signals from the outside world to the system.

A state is a set of object values for a given class that have the same qualitative response to events that occur. In other words, all objects with the same state react in the same general way to an event, so all objects in a given state execute the same action when they receive the same event. Objects in different states, however, may react differently to the same event, by performing different actions. For example, an automatic teller machine reacts to the cancel button one way when it is processing a transaction and another way when it is idle.

State machines describe the behavior of classes, but they also describe the dynamic behavior of use cases, collaborations, and methods. For one of these objects, a state represents a step in its execution. We talk mostly in terms of classes and objects in describing state machines, but they can be applied to other elements in a straightforward way.

## State Machine

A state machine is a graph of states and transitions. Usually a state machine is attached to a class and describes the response of an instance of the class to events that it receives. State machines may also be attached to operations, use cases, and collaborations to describe their execution.

A state machine is a model of all possible life histories of an object of a class. The object is examined in isolation. Any external influence from the rest of the world is summarized as an event. When the object detects an event, it responds in a way that depends on its current state. The response may include the execution of an action and a change to a new state. State machines can be structured to inherit transitions, and they can model concurrency.

A state machine is a localized view of an object, a view that separates it from the rest of the world and examines its behavior in isolation. It is a reductionist view of a system. This is a good way to specify behavior precisely, but often it is not a good way to understand the overall operation of a system. For a better idea of the system-wide effects of behavior, interaction views are often more useful. State machines are useful for understanding control mechanisms, however, such as user interfaces and device controllers.

## Event

An event is a noteworthy occurrence that has a location in time and space. It occurs at a point in time; it does not have duration. Model something as an event if its occurrence has consequences. When we use the word *event* by itself, we usually mean an event descriptor—that is, a description of all the individual event occurrences that have the same general form, just as the word *class* means all the individual objects that have the same structure. A specific occurrence of an event is called an event instance. Events may have parameters that characterize each individual event instance, just as classes have attributes that characterize each object. As with classes, signals can be arranged in generalization hierarchies to share common structure. Events can be divided into various explicit and implicit kinds: signal events, call events, change events, and time events. Table 6-1 is a list of event types and their descriptions.

**Table 6-1:** *Kinds of Events*

| *Event Type* | *Description* | *Syntax* |
|---|---|---|
| call event | Receipt of an explicit synchronous request among objects that waits for a response | op (a:T) |
| change event | A change in value of a Boolean expression | **when** (exp) |
| signal event | Receipt of an explicit, named, asynchronous communication among objects | sname (a:T) |
| time event | The arrival of an absolute time or the passage of a relative amount of time | **after** (time) |

***Signal event***. A signal is a named entity that is explicitly intended as a communication vehicle between two objects; the reception of a signal is an event for the receiving object. The sending object explicitly creates and initializes a signal instance and sends it to one or a set of explicit objects. Signals embody asynchronous one-way communication, the most fundamental kind. The sender does not wait for the receiver to deal with the signal but continues with its own work independently. To model two-way communication, multiple signals can be used, at least one in each direction. The sender and the receiver can be the same object.

Signals may be declared in class diagrams as classifiers, using the keyword «signal»; the parameters of the signal are declared as attributes. As classifiers, signals can have generalization relationships. Signals may be children of other signals; they inherit the parameters of their parents, and they trigger transitions that depend on the parent signal (Figure 6-1).

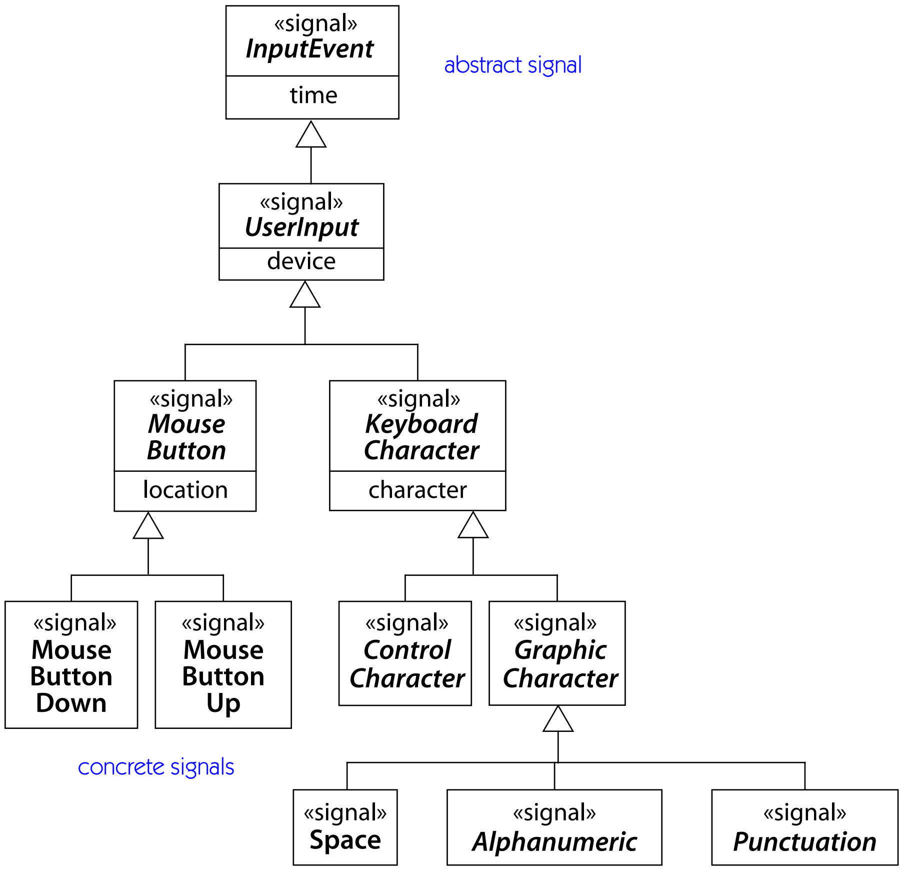


**Figure 6-1.** *Signal hierarchy*

***Call event.*** A call event is the reception of a call by an object that chooses to implement an operation as a state machine transition rather than as a fixed procedure. To the caller, an ordinary call (implemented by a method) is indistinguishable from a call event. The receiver chooses whether an operation will be implemented as a method or a call event trigger in a state machine. The parameters of the operation are the parameters of the event. Once the receiving object processes the call event by taking a transition triggered by the event or failing to take any transition, control returns to the calling object. Unlike an ordinary call, however, the receiver of a call event may continue its own execution in parallel with the caller.

***Change event.*** A change event is the satisfaction of a Boolean expression that depends on certain attribute values. This is a declarative way to wait until a condition is satisfied, but it must be used with care, because it represents a continuous and potentially nonlocal computation (action at a distance, because the value or values tested may be distant). This is both good and bad. It is good because it focuses the model on the true dependency—an effect that occurs when a given condition is satisfied—rather than on the mechanics of testing the condition. It is bad because it obscures the cause-and-effect relationship between the action that changes an underlying value and the eventual effect. The cost of testing a change event is potentially large, because in principle it is continuous. In practice, however, there are ways to avoid unnecessary computation. Change events should be used only when a more explicit form of communication is unnatural.

Note the difference between a guard condition and a change event. A guard condition is evaluated once when the trigger event on the transition occurs and the receiver handles the event. If it is false, the transition does not fire and the condition is not reevaluated. A change event is evaluated continuously until it becomes true, at which time the transition fires.

***Time event.*** Time events represent the passage of time. A time event can be specified either in absolute mode (time of day) or relative mode (time elapsed since a given event). In a high-level model, time events can be thought of as events from the universe; in an implementation model, they are caused by signals from some specific object, either the operating system or an object in the application.

## State

A state describes a period of time during the life of an object of a class. It can be characterized in three complementary ways: as a set of object values that are qualitatively similar in some respect; as a period of time during which an object waits for some event or events to occur; or as a period of time during which an object performs some ongoing activity. A state may have a name, although often it is anonymous and is described simply by its actions.

In a state machine, a set of states is connected by transitions. Although transitions connect two states (or more, if there is a fork or join of control), transitions are processed by the state that they leave. When an object is in a state, it is sensitive to the trigger events on transitions leaving the state.

A state is shown as a rectangle with rounded corners (Figure 6-2).

Confirm Credit

**Figure 6-2.** *State*

## Transition

A transition leaving a state defines the response of an object in the state to the occurrence of an event. In general, a transition has an event trigger, a guard condition, an action, and a target state. Table 6-2 shows kinds of transitions and implicit actions invoked by transitions.

***External transition***. An *external transition* is a transition that changes the active state. This is the most common kind of transition. It is drawn as an arrow from the

**Table 6-2:** *Kinds of Transitions and Implicit Actions*

| *Transition Kind* | *Description* | *Syntax* |
|---|---|---|
| entry action | An action that is executed when a state is entered | entry/ action |
| exit action | An action that is executed when a state is exited | exit/ action |
| external transition | A response to an event that causes a change of state or a self-transition, together with a specified action. It may also cause the execution of exit and/or entry actions for states that are exited or entered. | e(a:T)[exp]/action |
| internal transition | A response to an event that causes the execution of an action but does not cause a change of state or execution of exit or entry actions | e(a:T)[exp]/action |

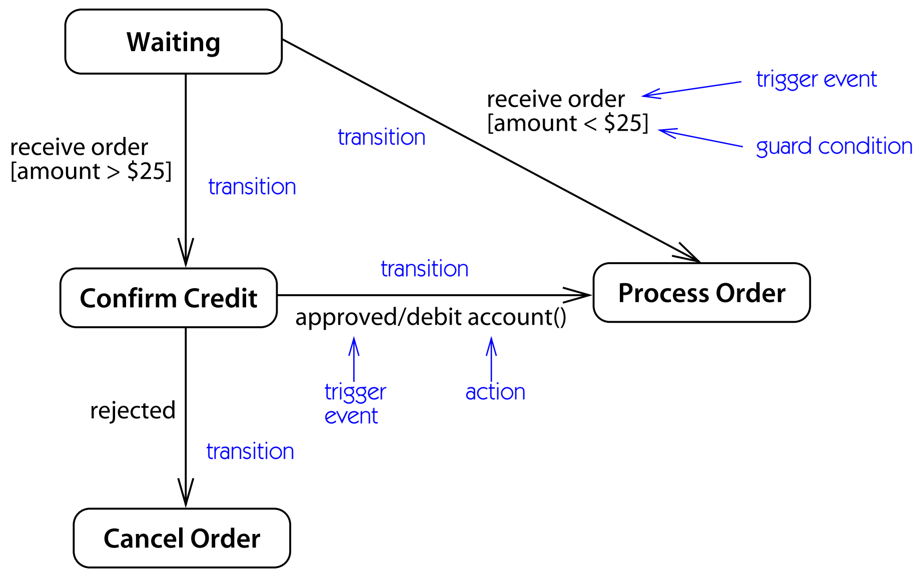


**Figure 6-3.** *External transitions*

source state to the target state, with other properties shown as a text string attached to the arrow (Figure 6-3).

***Trigger event.*** The trigger is an event the occurrence of which enables the transition. The event may have parameters, which are available to an action on the transition. If a signal has descendants, any descendant of the signal enables the transition. For example, if a transition has MouseButton as a trigger (see Figure 6-1), then MouseButtonDown will trigger the transition.

An event is not a continuous thing; it occurs at a point in time. When an object receives an event, it saves the event if it is not free to handle the event. An object handles one event at a time. A transition must fire at the time the object handles the event; the event is not “remembered” until later (except in the special case of deferred events, which are saved until they trigger a transition or until the object is in a state where they are not deferred). If two events occur simultaneously, they are handled one at a time. An event that does not trigger any transition is simply ignored and lost. This is not an error. It is much easier to ignore unwanted events than to try to specify all of them.

***Guard condition.*** A transition may have a guard condition, which is a Boolean expression. It may reference attributes of the object that owns the state machine, as well as parameters of the trigger event. The guard condition is evaluated when a trigger event occurs. If the expression evaluates as true, then the transitions fires—that is, its effects occur. If the expression evaluates as false, then the transition does not fire. The guard condition is evaluated only once, at the time the trigger event occurs. If the condition is false and later becomes true, it is too late to fire the transition.

The same event can be a trigger for more than one transition leaving a single state. Each transition with the same event must have a different guard condition. If the event occurs, a transition triggered by the event may fire if its condition is true. Often, the set of guard conditions covers all possibilities so that the occurrence of the event is guaranteed to fire some transition. If all possibilities are not covered and no transition is enabled, then an event is simply ignored. Only one transition may fire (within one thread of control) in response to one event occurrence. If an event enables more than one transition, only one of them fires. A transition on a nested state takes precedence over a transition on one of its enclosing states. If two conflicting transitions are enabled at the same time, one of them fires nondeterministically. The choice may be random or it may depend on implementation details, but the modeler should not count on a predicable result.

***Completion transition***. A transition that lacks an explicit trigger event is triggered by the completion of activity in the state that it leaves (this is a completion transition). A completion transition may have a guard condition, which is evaluated at the time the activity in the state completes (and not thereafter).

***Action***. When a transition fires, its action (if any) is executed. An action is an atomic and normally brief computation, often an assignment statement or simple arithmetic computation. Other actions include sending a signal to another object, calling an operation, setting return values, creating or destroying an object, and undefined control actions specified in an external language. An action may also be an action sequence—that is, a list of simpler actions. An action or action sequence cannot be terminated or affected by simultaneous actions. Conceptually, its duration is negligible compared to outside event timing; therefore, a second event cannot occur during its execution. In practice, however, actions take some time, and incoming events must be placed on a queue.

The overall system can perform multiple actions simultaneously. When we call actions atomic, we do not imply that the entire system is atomic. The system can process hardware interrupts and time share between several actions. An action is atomic within its own thread of control. Once started, it must complete and it must not interact with other simultaneously active actions. But actions should not be used as a long transaction mechanism. Their duration should be brief compared to the response time needed for external events. Otherwise, the system might be unable to respond in a timely manner.

An action may use parameters of the trigger event and attributes of the owning object as part of its expression.

Table 6-3 lists the kinds of actions and their descriptions.

**Table 6-3:** *Kinds of Actions*

| *Action Kind* | *Description* | *Syntax* |
|---|---|---|
| assignment | Sets the value of a variable | target := expression |
| call | Calls an operation on a target object; waits for completion of the operation execution; may return a value | opname (arg, arg) |
| create | Creates a new object | new Cname (arg, arg) |
| destroy | Destroys an object | object . destroy () |
| return | Specifies return values for the caller | return value |
| send | Creates a signal instance and sends it to a target object or set of objects | sname (arg, arg) |
| terminate | Self-destruction of the owning object | terminate |
| uninterpreted | Language-specific action, such as conditional or iteration | [language specific] |

***Change of state.*** When the execution of the action is complete, the target state of the transition becomes active. This may trigger exit actions or entry actions.

***Nested states.*** States may be nested inside other composite states (see following entry). A transition leaving an outer state is applicable to all states nested within it. The transition is eligible to fire whenever any nested state is active. If it fires, the target state of the transition becomes active. Composite states are useful for expressing exception and error conditions, because transitions on them apply to all nested states without the need for each nested state to handle the exception explicitly.

***Entry and exit actions.*** A transition across one or more levels of nesting may exit and enter states. A state may have actions that are performed whenever the state is entered or exited. Entering the target state executes an entry action attached to the state. If the transition leaves the original state, then its exit action is executed before the action on the transition and the entry action on the new state.

Entry actions are often used to perform setup needed within a state. Because an entry action cannot be evaded, any actions that occur inside the state can assume that the setup has occurred, regardless of how the state is entered. Similarly, an exit action is an action that occurs whenever the state is exited, an opportunity to perform clean up. It is particularly useful when there are high-level transitions that represent error conditions that abort nested states. The exit action can clean up such cases so that the state of the object remains consistent. Entry and exit actions could in principle be attached to incoming and outgoing transitions, but declaring them as special actions of the state permits the state to be defined independently of its transitions and therefore encapsulated.

***Internal transition***. An internal transition has a source state but no target state. The firing rules for an internal transition are the same as for a transition that changes state. An internal transition has no target state, so the active state does not change as a result of its firing. If an internal transition has an action, it is executed, but no change of state occurs, and therefore no exit or entry actions are executed. Internal transitions are useful for modeling interrupt actions that do not change the state (such as counting occurrences of an event or putting up a help screen).

Entry and exit actions use the same notation as internal transitions, except they use the reserved words **entry** and **exit** in place of the event trigger name, although these actions are triggered by external transitions that enter or leave the state.

A self-transition invokes exit and entry actions on its state (conceptually, it exits and then reenters the state); therefore, it is not equivalent to an internal transition. Figure 6-4 shows entry and exit actions as well as internal transitions.

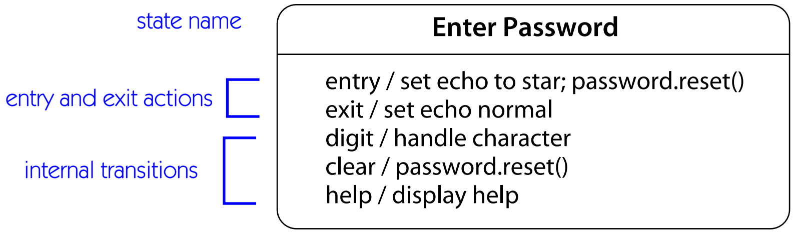


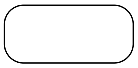
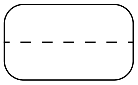
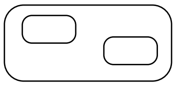
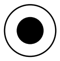
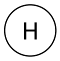
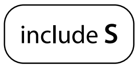
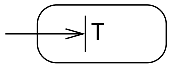
**Figure 6-4.** *Internal transitions, and entry and exit actions*

## Composite States

A simple state has no substructure, just a set of transitions and possible entry and exit actions. A composite state is one that has been decomposed into sequential substates or concurrent substates. Table 6-4 lists the various kinds of states.

A decomposition into disjoint substates is a kind of specialization of a state. An outer state is refined into several inner states, each of which inherits the transitions of the outer state. Only one sequential substate can be active at one time. The outer state represents the condition of being in any one of the inner states.

**Table 6-4:** *Kinds of States*

| *State Kind* | *Description* | *Notation* |
|---|---|---|
| simple state | A state with no substructure |  |
| concurrent composite state | A state that is divided into two or more concurrent substates, all of which are concurrently active when the composite state is active |  |
| sequential composite state | A state that contains one or more disjoint substates, exactly one of which is active at one time when the composite state is active |  |
| initial state | A pseudostate that indicates the starting state when the enclosing state in invoked | |
| final state | A special state whose activation indicates the enclosing state has completed activity |  |
| junction state | A pseudostate that chains transition segments into a single run-to-completion transition | |
| history state | A pseudostate whose activation restores the previously active state within a composite state |  |
| submachine reference state | A state that references a submachine, which is implicitly inserted in place of the submachine reference state |  |
| stub state | A pseudostate within a submachine reference state that identifies a state in the referenced state machine |  |

Transitions into or out of a composite state invoke the entry actions or exit actions of the state. If there are several composite states, a transition across several levels may invoke multiple entry actions (outermost first) or several exit actions (innermost first). If there is an action on the transition itself, the action is executed after any exit actions and before any entry actions are executed.

A composite state may also have an initial state within it. A transition to the composite state boundary is implicitly a transition to the initial state. A new object starts at its initial state of its outermost state. Similarly, a composite state can have a final state. A transition to the final state triggers a completion transition (triggerless transition) on the composite state. If an object reaches the final state of its outermost state, it is destroyed. Initial states, final states, entry actions, and exit actions permit the definition of a state to be encapsulated independent of transitions to and from it.

Figure 6-5 shows a sequential decomposition of a state, including an initial state. This is the control for a ticket-selling machine.

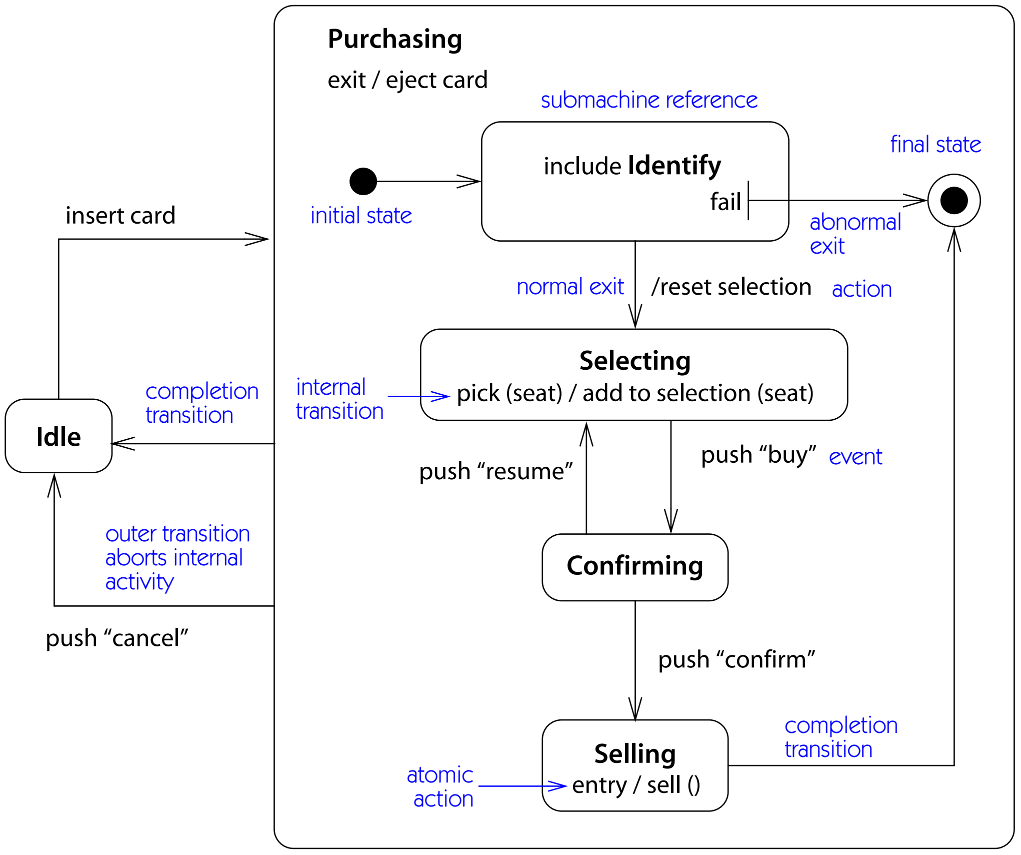


**Figure 6-5.** *State machine*

A decomposition into concurrent substates represents independent computation. When a concurrent superstate is entered, the number of control threads increases. When it is exited, the number of control threads decreases. Often, concurrency is implemented by a distinct object for each substate, but concurrent substates can also represent logical concurrency within a single object. Figure 6-6 shows the concurrent decomposition of taking a university class.

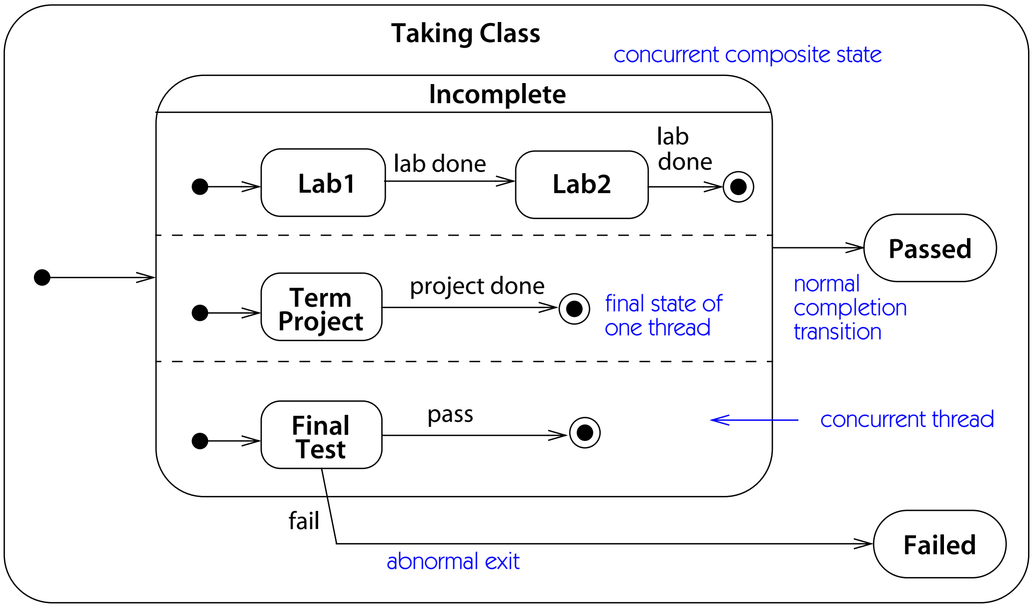


**Figure 6-6.** *State machine with concurrent composite state*

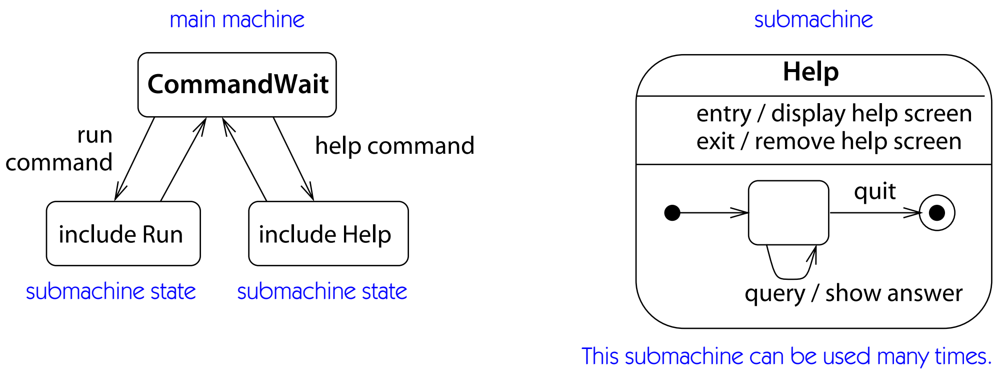


**Figure 6-7.** *Submachine state*

It is often convenient to reuse a fragment of a state machine in other state machines. A state machine can be given a name and referenced from a state of one or more other machines. The target state machine is a submachine, and the state referencing it is called a submachine reference state. It implies the (conceptual) substitution of a copy of the referenced state machine at the place of reference, a kind of state machine subroutine. Instead of a submachine, a state can contain an activity—that is, a computation or continuous occurrence that takes time to complete and that may be interrupted by events. Figure 6-7 shows a submachine reference.

A transition to a submachine reference state causes activation of the initial state of the target submachine. To enter a submachine at other states, place one or more stub states in the submachine reference state. A stub state identifies a state in the submachine.

# 7
# *Activity View*

## Overview

An activity graph is a special form of state machine intended to model computations and workflows. The states of the activity graph represent the states of executing the computation, not the states of an ordinary object. Normally, an activity graph assumes that computations proceed without external event-based interruptions (otherwise, an ordinary state machine may be preferable).

An activity graph contains activity states. An activity state represents the execution of a statement in a procedure or the performance of an activity in a workflow. Instead of waiting for an event, as in a normal wait state, an activity state waits for the completion of its computation. When the activity completes, then execution proceeds to the next activity state within the graph. A completion transition in an activity diagram fires when the preceding activity is complete. Activity states usually do not have transitions with explicit events, but they may be aborted by transitions on enclosing states.

An activity graph may also contain action states, which are similar to activity states, except that they are atomic and do not permit transitions while they are active. Action states should usually be used for short bookkeeping operations.

An activity diagram may contain branches, as well as forking of control into concurrent threads. Concurrent threads represent activities that can be performed concurrently by different objects or persons in an organization. Frequently concurrency arises from aggregation, in which each object has its own concurrent thread. Concurrent activities can be performed simultaneously or in any order. An activity graph is like a traditional flow chart except it permits concurrent control in addition to sequential control—a big difference.

## Activity Diagram

An activity diagram is the notation for an activity graph (Figure 7-1). It includes some special shorthand symbols for convenience. These symbols can actually be

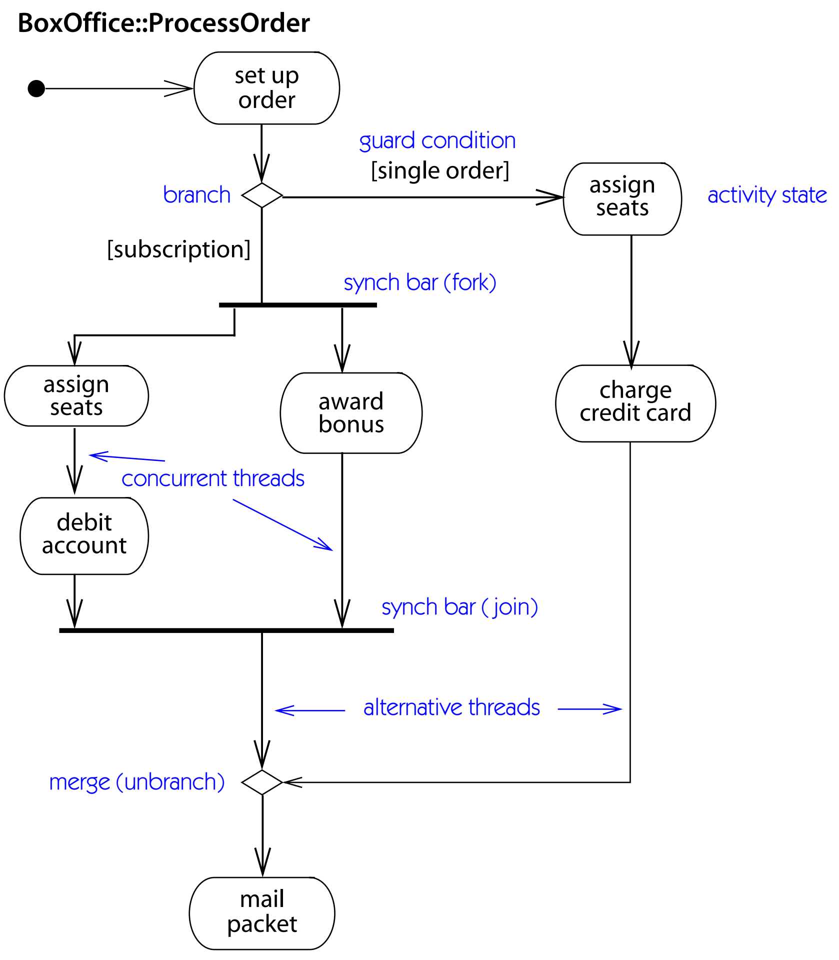


**Figure 7-1.** *Activity diagram*

used on any statechart diagram, although mixing notation may be ugly much of the time.

An activity state is shown as a box with rounded ends containing a description of the activity. (Normal state boxes have straight sides and rounded corners.) Simple completion transitions are shown as arrows. Branches are shown as guard conditions on transitions or as diamonds with multiple labeled exit arrows. A fork or join of control is shown the same way as on a statechart, by multiple arrows entering or leaving a heavy synchronization bar. Figure 7-1 shows an activity diagram for processing an order by the box office.

For those situations in which external events must be included, the receipt of an event can be shown as a trigger on a transition or as a special inline symbol that denotes waiting for a signal. A similar notation shows sending a signal. If there are many event-driven transitions, however, an ordinary statechart diagram is probably preferable.

***Swimlanes***. It is often useful to organize the activities in a model according to responsibility—for example, by grouping together all the activities handled by one business organization. This kind of assignment can be shown by organizing the activities into distinct regions separated by lines in the diagram. Because of their appearance, each region is called a swimlane. Figure 7-2 shows swimlanes.

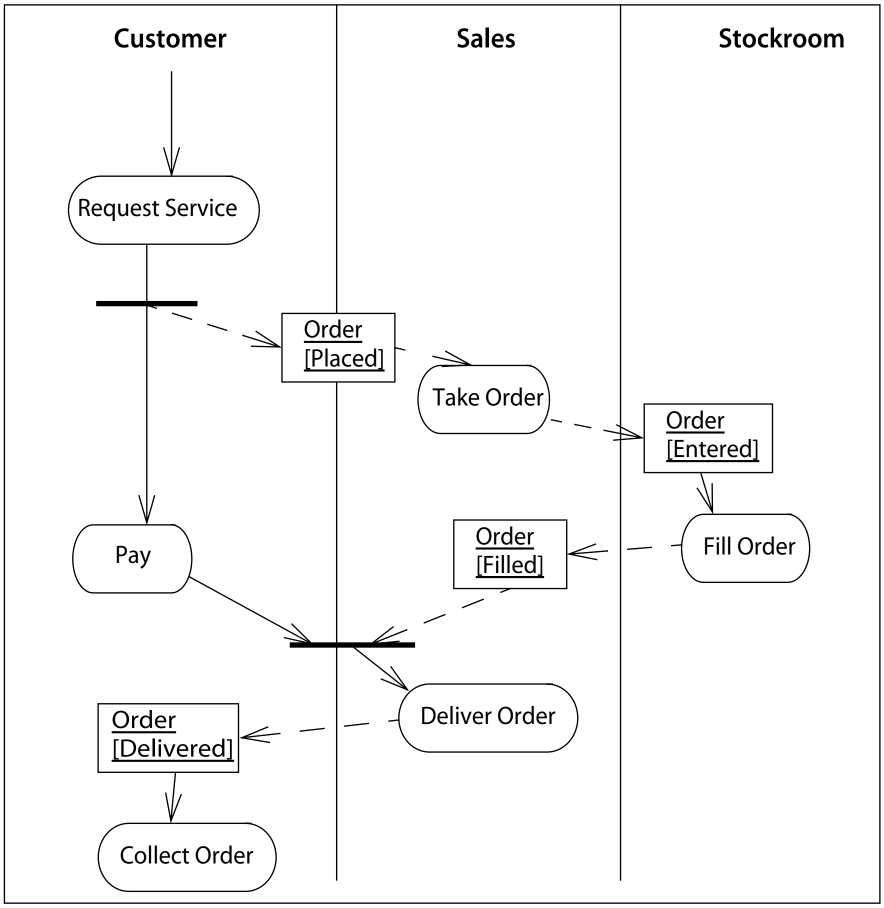


**Figure 7-2.** *Swimlanes and object flows*

***Object flows***. An activity diagram can show the flow of object values, as well as the flow of control. An object flow state represents an object that is the input or output of an activity. For an output value, a dashed arrow is drawn from an activity to an object flow state. For an input value, a dashed arrow is drawn from an object flow state to an activity. If an activity has more than one output value or successor control flow, the arrows are drawn from a fork symbol. Similarly, multiple inputs are drawn to a join symbol.

Figure 7-2 shows an activity diagram in which both activities and object flow states have been assigned to swimlanes.

## Activities and Other Views

Activity graphs do not show the full detail of a computation. They show the flow of activities but not the objects that perform the activities. Activity graphs are a starting point for design. To complete a design, each activity must be expanded as one or more operations, each of which is assigned to a specific class to implement. Such an assignment results in the design of a collaboration that implements the activity graph.

# 8 Interaction View

## Overview

Objects interact to implement behavior. This interaction can be described in two complementary ways, one of them centered on individual objects and the other on a collection of cooperating objects.

A state machine is a narrow, deep view of behavior, a reductionist view that looks at each object individually. A state machine specification is precise and leads immediately to code. It can be difficult to understand the overall functioning of a system, however, because a state machine focuses on a single object at a time, and the effects of many state machines must be combined to determine the behavior of an entire system. The interaction view provides a more holistic view of the behavior of a set of objects. This view is modeled by collaborations.

## Collaboration

A collaboration is a description of a collection of objects that interact to implement some behavior within a context. It describes a society of cooperating objects assembled to carry out some purpose. A collaboration contains slots that are filled by objects and links at run time. A collaboration slot is called a role because it describes the purpose of an object or link within the collaboration. A classifier role represents a description of the objects that can participate in an execution of the collaboration; an association role represents a description of the links that can participate in an execution of the collaboration. A classifier role is a classifier that is constrained by its part in the collaboration; an association role is an association that is constrained by its part in the collaboration. Relationships among classifier roles and association roles inside a collaboration are only meaningful in that context. In general, the same relationships do not apply to the underlying classifiers and associations apart from the collaboration.

The static view describes the inherent properties of a class. For example, a Vehicle has an owner. A collaboration describes the properties that an instance of a class has because it plays a particular role in a collaboration. For example, a rentalVehicle in a RentalCar collaboration has a rentalDriver, something that is not relevant to a Vehicle in general but is an essential part of the collaboration.

An object in a system may participate in more than one collaboration. Collaborations in which it appears need not be directly related, although their execution is connected through the shared object. For example, one person may be both a rentalDriver and a hotelGuest as part of a Vacation model. Somewhat less often, an object may play more than one role in the same collaboration.

A collaboration has both a structural aspect and a behavioral aspect. The structural aspect is similar to a static view—it contains a set of roles and their relationships that define the context for its behavioral aspect. The behavioral aspect is the set of messages exchanged by the objects bound to the roles. Such a set of messages on a collaboration is called an interaction. A collaboration can include one or more interactions, each of which describes a series of messages exchanged among the objects in the collaboration to perform a goal.

Whereas a state machine is narrow and deep, a collaboration is broad but more shallow. It captures a more holistic view of behavior in the exchange of messages within a network of objects. Collaborations show the unity of the three major structures underlying computation: data structure, control flow, and data flow.

## Interaction

An interaction is a set of messages within a collaboration that are exchanged by classifier roles across association roles. When a collaboration exists at run time, objects bound to classifier roles exchange message instances across links bound to association roles. An interaction models the execution of an operation, use case, or other behavioral entity.

A message is a one-way communication between two objects, a flow of control with information from a sender to a receiver. A message may have parameters that convey values between the objects. A message can be a signal (an explicit, named, asynchronous interobject communication) or a call (the synchronous invocation of an operation with a mechanism for later returning control to the sender).

The creation of a new object is modeled as an event caused by the creator object and received by the class itself. The creation event is available to the new instance as the current event on the transition from the top-level initial state.

Messages can be arranged into sequential threads of control. Separate threads represent sets of messages that are concurrent. Synchronization among threads is modeled by constraints among messages in different threads. One synchronization construct can model forks of control, joins of control, and branches.

Sequencing of messages can be shown in two kinds of diagrams: a sequence diagram (focusing on the time sequences of the messages) and a collaboration diagram (focusing on the relationships among the objects that exchange the messages).

## Sequence Diagram

A sequence diagram displays an interaction as a two-dimensional chart. The vertical dimension is the time axis; time proceeds down the page. The horizontal dimension shows the classifier roles that represent individual objects in the collaboration. Each classifier role is represented by a vertical column—the lifeline. During the time an object exists, the role is shown by a dashed line. During the time an activation of a procedure on the object is active, the lifeline is drawn as a double line.

A message is shown as an arrow from the lifeline of one object to that of another. The arrows are arranged in time sequence down the diagram.

Figure 8-1 shows a typical sequence diagram with asynchronous messages.

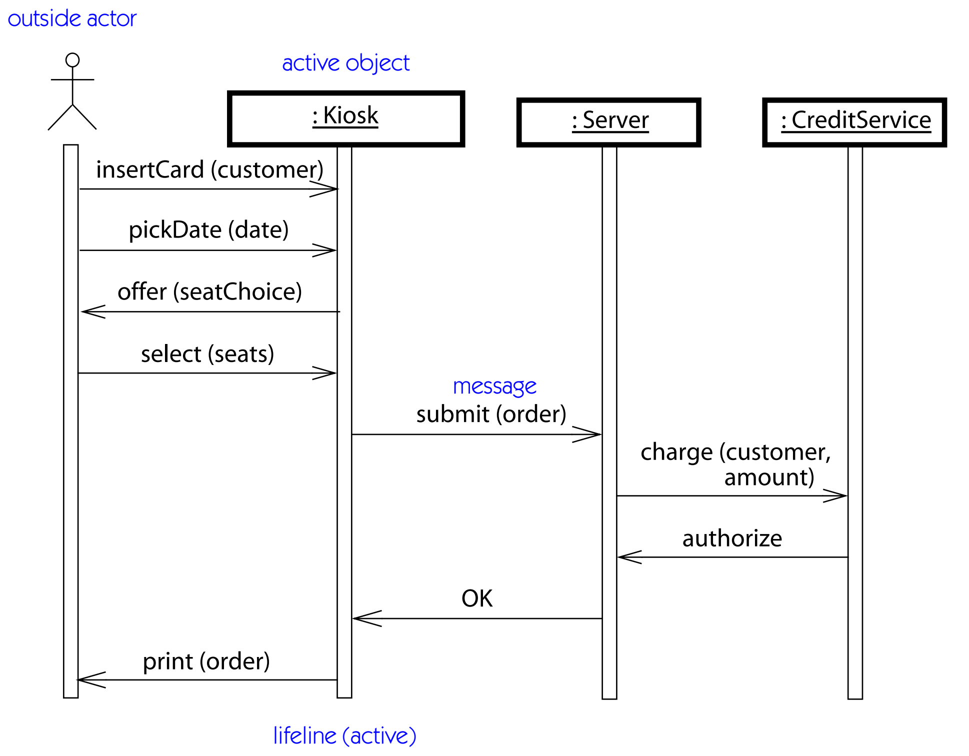


**Figure 8-1.** *Sequence diagram*

## Activation

An activation is the execution of a procedure, including the time it waits for nested procedures to execute. It is shown by a double line replacing part of the lifeline in a sequence diagram. A call is shown by an arrow leading to the top of the activation the call initiates. A recursive call occurs when control reenters an operation on an object, but the second call is a separate activation from the first. Recursion or a nested call to another operation on the same object is shown in a sequence diagram by stacking the activation lines. Figure 8-2 shows a sequence diagram with procedural flow of control, including a recursive call and the creation of an object during the computation.

An active object is one that holds the root of a stack of activations. Each active object has its own event-driven thread of control that executes in parallel with other active objects. The objects that are called by an active object are passive objects; they receive control only when called, and they yield it up when they return.

If several concurrent threads of control have their own procedural flows of control using nested calls, the different threads must be distinguished using thread names, colors, or other means to avoid confusion when two threads come together

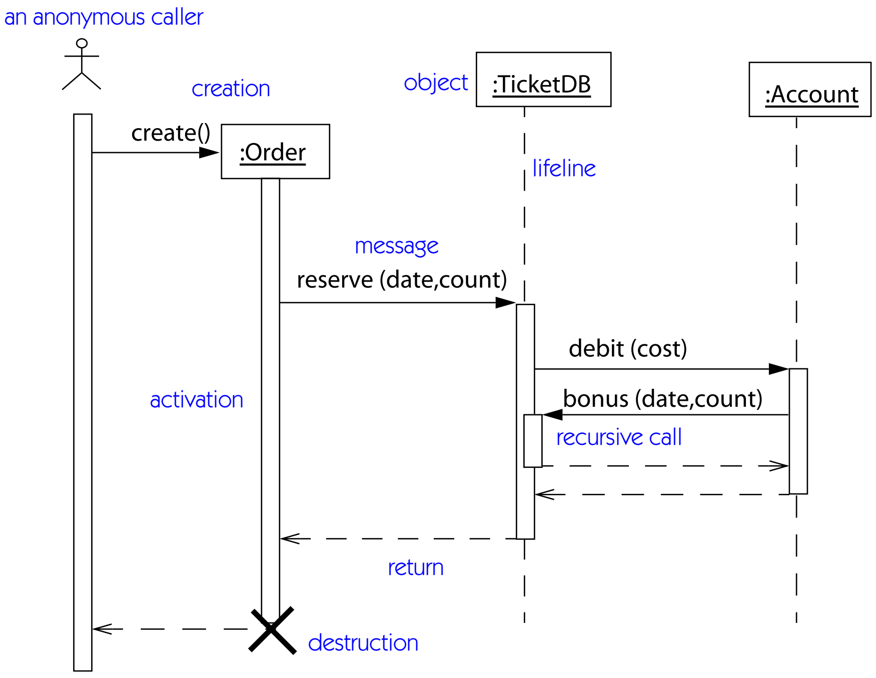


**Figure 8-2.** *Sequence diagram with activations*

on a single object (by a rendezvous, for example). Usually, it is best not to mix procedure calls with signals on a single diagram.

## Collaboration Diagram

A collaboration diagram is a class diagram that contains classifier roles and association roles rather than just classifiers and associations. Classifier roles and association roles describe the configuration of objects and links that may occur when an instance of the collaboration is executed. When the collaboration is instantiated, objects are bound to the classifier roles and links are bound to the association roles. Association roles may also be played by various kinds of temporary links, such as procedure arguments or local procedure variables. Link symbols may carry stereotypes to indicate temporary links («parameter» or «local») or calls to the same object («self»). Only objects that are involved in the collaboration are represented, although there may be others in the entire system. In other words, a collaboration diagram models the objects and links involved in the implementation of an interaction and ignores the others. Figure 8-3 shows a collaboration diagram.

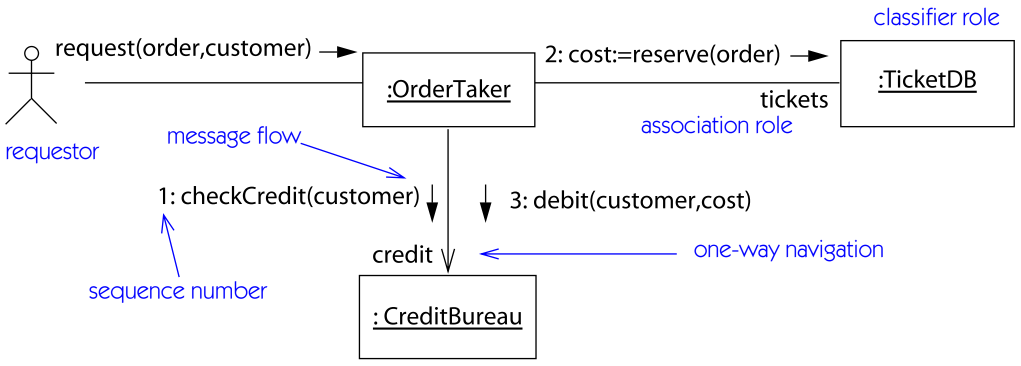


**Figure 8-3.** *Collaboration diagram*

It is useful to mark the objects in four groups: those that exist through the entire interaction; those created during the interaction (constraint {new}); those destroyed during the interaction (constraint {destroyed}); and those that are created and destroyed during the interaction (constraint {transient}). During design, you can start by showing the objects and links available at the start of an operation and then decide how control can flow to the correct objects within the graph to implement the operation.

Although collaborations directly show the implementation of an operation, they may also show the realization of an entire class. In this usage, they show the context needed to implement *all* of the operations of a class. This permits the modeler to see the multiple roles that objects may play in various operations. This view can be constructed by taking the union of all the collaborations needed to describe all the operations of the object.

***Messages***. Messages are shown as labeled arrows attached to links. Each message has a sequence number, an optional list of predecessor messages, an optional guard condition, a name and argument list, and an optional return value name. The sequence number includes the (optional) name of a thread. All messages in the same thread are sequentially ordered. Messages in different threads are concurrent unless there is an explicit sequencing dependency. Various implementation details may be added, such as a distinction between asynchronous and synchronous messages.

***Flows***. Usually, a collaboration diagram contains a symbol for an object during an entire operation. Sometimes, however, an object has different states that must be made explicit. For example, an object might change location, or its associations might differ significantly at different times. An object can be shown with both its class and its state—an object with a class-in-state. The same object can be shown multiple times, each with a different location or state.

The various object symbols that represent one object may be connected using become flows. A become flow is a transition from one object state to another. It is drawn as a dashed arrow with the stereotype «become» and may be labeled with a sequence number to show when it occurs (Figure 8-4). A become flow is also used to show migration of an object from one location to another.

Less commonly, the stereotype «copy» shows an object value produced by copying another object value.

Table 8-1 shows the kinds of object flow relationships.

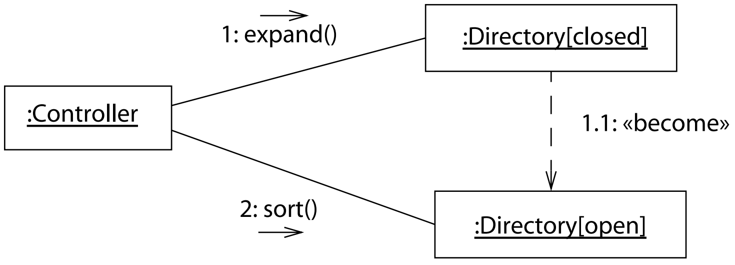


**Figure 8-4.** *Become flow*

**Table 8-1:** *Kinds of Flow Relationships*

| *Flow* | *Function* | *Notation* |
|---|---|---|
| become | Transformation from one value of an object to another value | «become» - - - - - - - -> |
| copy | Copy of an object that is thereafter independent | «copy» - - - - - - - -> |

***Collaboration and sequence diagrams.*** Collaboration diagrams and sequence diagrams both show interactions, but they emphasize different aspects. Sequence diagrams show time sequences clearly but do not show object relationships explicitly. Collaboration diagrams show object relationships clearly, but time sequences must be obtained from sequence numbers. Sequence diagrams are often most useful for showing scenarios; collaboration diagrams are often more useful for showing detailed design of procedures.

## Patterns

A pattern is a parameterized collaboration, together with guidelines about when to use it. A parameter can be replaced by different values to produce different collaborations. The parameters usually designate slots for classes. When a pattern is instantiated, its parameters are bound to actual classes within a class diagram or to roles within a larger collaboration.

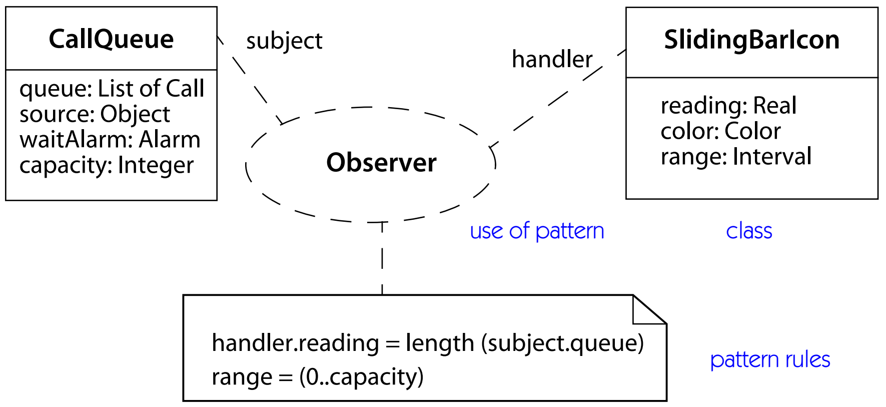


**Figure 8-5.** *Pattern usage*

The use of a pattern is shown as a dashed ellipse connected to each of its classes by a dashed line that is labeled with the name of the role. For example, Figure 8-5 shows the use of the **Observer** pattern from [Gamma-95]. In this use of the pattern, **CallQueue** replaces the **subject** role and **SlidingBarIcon** replaces the **handler** role.

Patterns may appear at the analysis, architecture, detailed design, and implementation levels. They are a way to capture frequently occurring structures for reuse. Figure 8-5 shows a use of the Observer pattern .