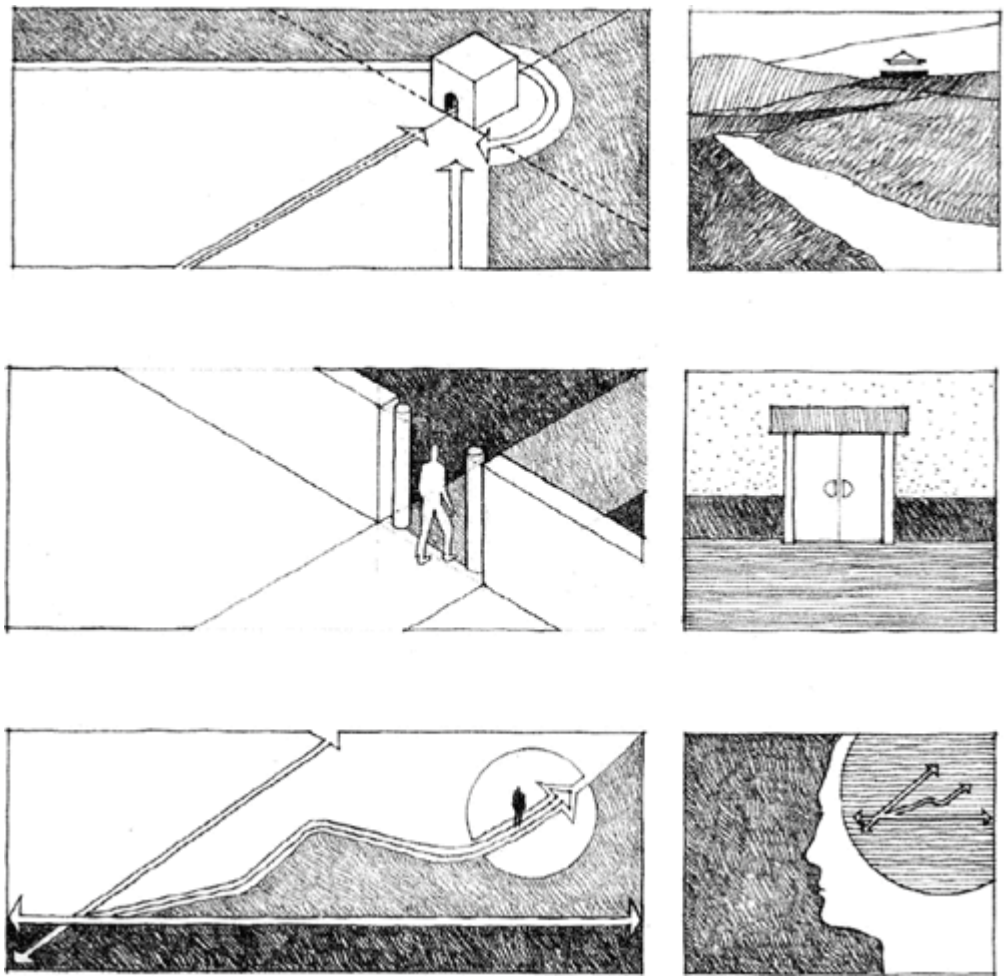
Part 5: Advanced Behavioral Modeling



Chapter 21. Events and Signals

In this chapter

- Signal events, call events, time events, and change events
- Modeling a family of signals
- Modeling exceptions
- Handling events in active and passive objects

In the real world, things happen. Not only do things happen, but lots of things may happen at the same time, and at the most unexpected times. "Things that happen" are called events, and each one represents the specification of a significant occurrence that has a location in time and space.

In the context of state machines, you use events to model the occurrence of a stimulus that can trigger a state transition. Events may include signals, calls, the passing of time, or a change in state.

Events may be synchronous or asynchronous, so modeling events is wrapped up in the modeling of processes and threads.

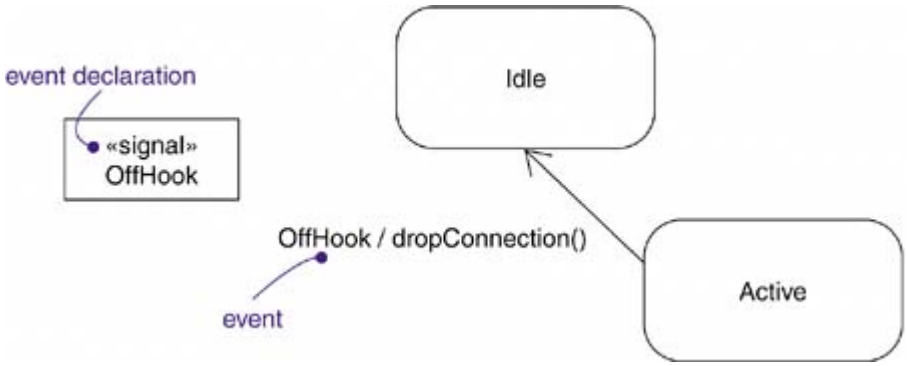
Getting Started

A perfectly static system is intensely uninteresting because nothing ever happens. All real systems have some dynamic dimension to them, and these dynamics are triggered by things that happen externally or internally. At an ATM machine, action is initiated by a user pressing a button to start a transaction. In an autonomous robot, action is initiated by the robot bumping into an object. In a network router, action is initiated by the detection of an overflow of message buffers. In a chemical plant, action is initiated by the passage of time sufficient for a chemical reaction.

In the UML, each thing that happens is modeled as an event. An event is the specification of a significant occurrence that has a location in time and space. A signal, the passing of time, and a change of state are asynchronous events, representing events that can happen at arbitrary times. Calls are generally synchronous events, representing the invocation of an operation.

The UML provides a graphical representation of an event, as Figure 21-1 shows. This notation permits you to visualize the declaration of events (such as the signal `OffHook`) as well as the use of events to trigger a state transition (such as the signal `OffHook`, which causes a transition from the `Active` to the `Idle` state as well as the execution of the `dropConnection` action).

Figure 21-1. Events



Terms and Concepts

An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. A *signal* is a kind of event that represents the specification of an asynchronous message communicated between instances.

Kinds of Events

Events may be external or internal. External events are those that pass between the system and its actors. For example, the pushing of a button and an interrupt from a collision sensor are both examples of external events. Internal events are those that pass among the objects that live inside the system. An overflow exception is an example of an internal event.

> *Actors are discussed in Chapter 17; systems are discussed in Chapter 32.*

In the UML, you can model four kinds of events: signals, calls, the passing of time, and a change in state.

> *The creation and destruction of objects are also kinds of signals, as discussed in Chapter 16.*

Signals

A message is a named object that is sent asynchronously by one object and then received by another. A signal is a classifier for messages; it is a message type.

> *Classes are discussed in Chapters 4 and 9; generalization is discussed in Chapters 5 and 10.*

Signals have a lot in common with plain classes. For example, signals may have instances, although you don't generally need to model them explicitly. Signals may also be involved in generalization relationships, permitting you to model hierarchies of events, some of which are general (for example, the signal `NetworkFailure`) and some of which are specific (for example, a specialization of `NetworkFailure` called `WarehouseServerFailure`). Also as for classes, signals may have attributes and operations. Before it has been sent by one object or after it is received by another, a signal is just an ordinary data object.

Note

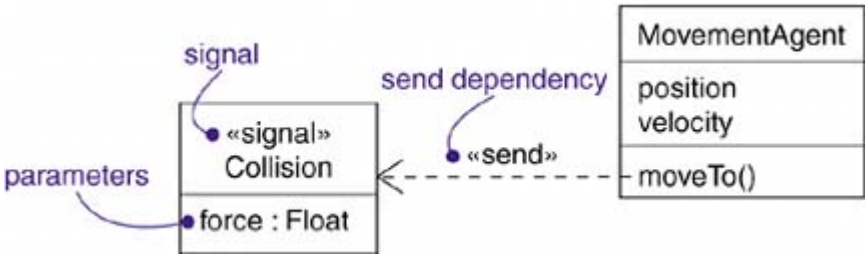
The attributes of a signal serve as its parameters. For example, when you send a signal such as `Collision`, you can also specify a value for its attributes as parameters, such as `Collision(5.3)`.

A signal may be sent by the action of a transition in a state machine. It may be modeled as a message between two roles in an interaction. The execution of a method can also send signals. In fact, when you model a class or an interface, an important part of specifying the behavior of that element is specifying the signals that its operations can send.

State machines are discussed in Chapter 22; interactions are discussed in Chapter 16; interfaces are discussed in Chapter 11; dependencies are discussed in Chapter 5; stereotypes are discussed in Chapter 6.

In the UML, as Figure 21-2 shows, you model signals as stereotyped classes. You can use a dependency, stereotyped as `send`, to indicate that an operation sends a particular signal.

Figure 21-2. Signals



Call Events

Just as a signal event represents the occurrence of a signal, a call event represents the receipt by an object of a call request for an operation on the object. A call event may trigger a state transition in a state machine or it may invoke a method on the target object. The choice is specified in the class definition for the operation.

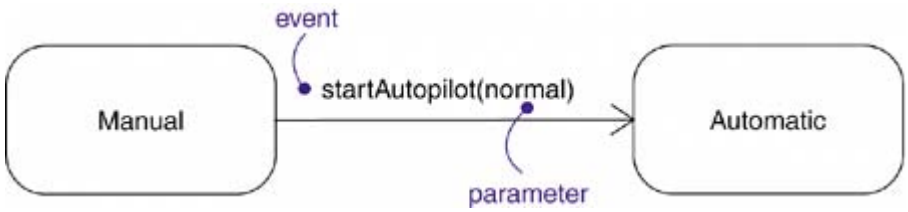
State machines are discussed in Chapter 22.

Whereas a signal is an asynchronous event, a call event is usually synchronous. This means that when an object invokes an operation on another object that has a state machine, control passes from the

sender to the receiver, the transition is triggered by the event, the operation is completed, the receiver transitions to a new state, and control returns to the sender. In those cases where the caller does not need to wait for a response, a call can be specified as asynchronous.

As Figure 21-3 shows, modeling a call event is indistinguishable from modeling a signal event. In both cases, you show the event, along with its parameters, as the trigger for a state transition.

Figure 21-3. Call Events



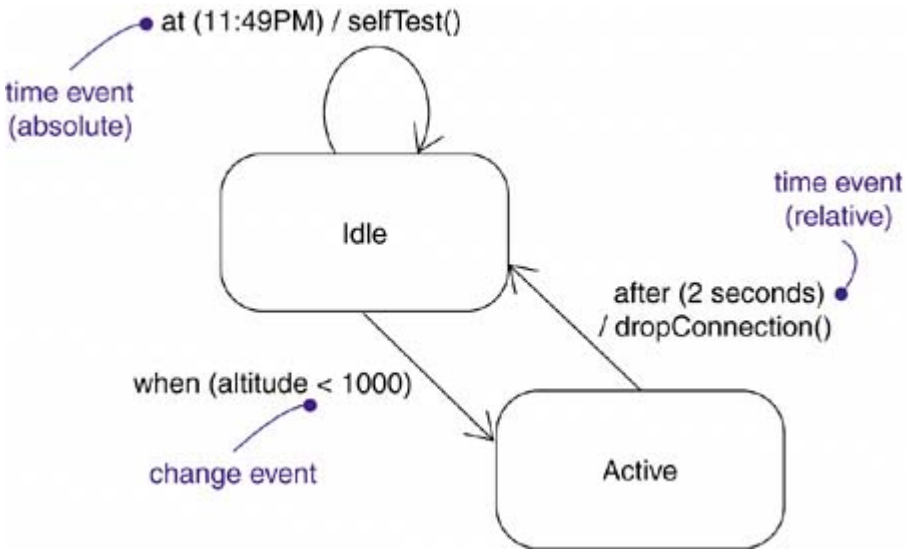
Note

Although there are no visual cues to distinguish a signal event from a call event, the difference is clear in the backplane of your model. The receiver of an event will know the difference, of course (by declaring the operation in its operation list). Typically, a signal will be handled by its state machine, and a call event will be handled by a method. You can use your tools to navigate from the event to the signal or the operation.

Time and Change Events

A time event is an event that represents the passage of time. As Figure 21-4 shows, in the UML you model a time event by using the keyword `after` followed by some expression that evaluates to a period of time. Such expressions can be simple (for example, `after 2 seconds`) or complex (for example, `after 1 ms since exiting Idle`). Unless you specify it explicitly, the starting time of such an expression is the time since entering the current state. To indicate a time event that occurs at an absolute time, use the keyword `at`. For example, the time event `at (1 Jan 2005, 1200 UT)` specifies an event that occurs on noon Universal Time on New Year's Day 2005.

Figure 21-4. Time and Change Events



A change event is an event that represents a change in state or the satisfaction of some condition. As Figure 21-4 shows, in the UML you model a change event by using the keyword `when` followed by some Boolean expression. You can use such expressions for the continuous test of an expression (for example, `when altitude < 1000`).

A change event occurs once when the value of the condition changes from false to true. It does not occur when the value of the condition changes from true to false. The event does not recur while the event remains true.

Note

Although a change event models a condition that is tested continuously, you can typically analyze the situation to see when to test the condition at discrete points in time.

Sending and Receiving Events

Signal events and call events involve at least two objects: the object that sends the signal or invokes the operation and the object to which the event is directed. Because signals are asynchronous, and because asynchronous calls are themselves signals, the semantics of events interact with the semantics of active objects and passive objects.

Processes and threads are discussed in Chapter 23.

Any instance of any class can send a signal to or invoke an operation of a receiving object. When an object sends a signal, the sender dispatches the signal and then continues along its flow of control, not waiting for any return from the receiver. For example, if an actor interacting with an ATM system sends the signal `pushButton`, the actor may continue along its way independent of the system to which the signal was sent. In contrast, when an object calls an operation, the sender dispatches the operation and then waits for the receiver to reply. For example, in a trading system, an instance of the class `trader` might invoke the operation `confirmTransaction` on some instance of the class `TRade`,

thereby affecting the state of the `TRade` object. If this is a synchronous call, the `trader` object will wait until the operation is finished.

Instances are discussed in Chapter 13.

Note

In some situations, you may want to show one object sending a signal to a set of objects (multicasting) or to any object in the system that might be listening (broadcasting). To model multicasting, you'd show an object sending a signal to a collection containing a set of receivers. To model broadcasting, you'd show an object sending a signal to another object that represents the system as a whole.

Any instance of any class can receive a call event or a signal. If this is a synchronous call event, then the sender and the receiver are in a rendezvous for the duration of the operation. This means that the flow of control of the sender suspends until the execution of the operation completes. If this is a signal, then the sender and receiver do not rendezvous: The sender dispatches the signal but does not wait for a response from the receiver. In either case, this event may be lost (if no response to the event is specified), it may trigger the receiver's state machine (if there is one), or it may just invoke a normal method call.

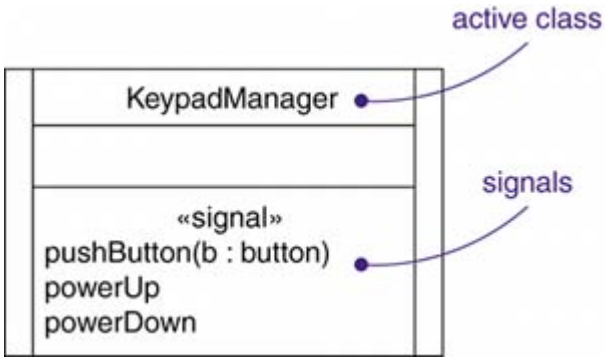
State machines are discussed in Chapter 22; active objects are discussed in Chapter 23.

Note

A call may be asynchronous. In this case, the caller continues immediately after issuing the call. The transmission of the message to the receiver and its execution by the receiver occur concurrently with the subsequent execution of the caller. When the execution of the method is complete, it just ends. If the method attempts to return values, they are ignored.

In the UML, you model the call events that an object may receive as operations on the class of the object. In the UML, you model the named signals that an object may receive by naming them in an extra compartment of the class, as shown in Figure 21-5.

Figure 21-5. Signals and Active Classes



Operations are discussed in Chapter 4; extra class compartments are discussed in Chapter 4.

Interfaces are discussed in Chapter 11; asynchronous operations are discussed in Chapter 23.

Note

You can also attach named signals to an interface in this same manner. In either case, the signals you list in this extra compartment are not the declarations of a signal, but only the use of a signal.

PREV NEXT

Common Modeling Techniques

Modeling a Family of Signals

In most event-driven systems, signal events are hierarchical. For example, an autonomous robot might distinguish between external signals, such as a `Collision`, and internal ones, such as a `HardwareFault`. External and internal signals need not be disjoint, however. Even within these two broad classifications, you might find specializations. For example, `HardwareFault` signals might be further specialized as `BatteryFault` and `MovementFault`. Even these might be further specialized, such as `MotorStall`, a kind of `MovementFault`.

Generalization is discussed in Chapters 5 and 10.

By modeling hierarchies of signals in this manner, you can specify polymorphic events. For example, consider a state machine with a transition triggered only by the receipt of a `MotorStall`. As a leaf signal in this hierarchy, the transition can be triggered only by that signal, so it is not polymorphic. In contrast, suppose you modeled the state machine with a transition triggered by the receipt of a `HardwareFault`. In this case, the transition is polymorphic and can be triggered by a `HardwareFault` or any of its specializations, including `BatteryFault`, `MovementFault`, and `MotorStall`.

State machines are discussed in Chapter 22.

To model a family of signals,

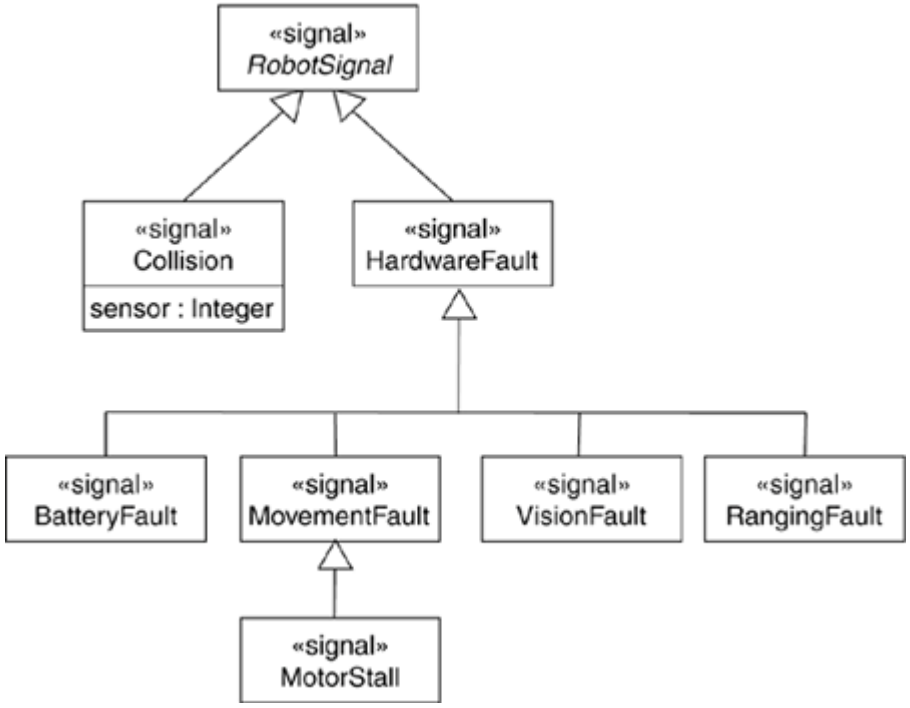
- Consider all the different kinds of signals to which a given set of active objects may respond.
- Look for the common kinds of signals and place them in a generalization/specialization hierarchy using inheritance. Elevate more general ones and lower more specialized ones.
- Look for the opportunity for polymorphism in the state machines of these active objects. Where you find polymorphism, adjust the hierarchy as necessary by introducing intermediate abstract signals.

Abstract classes are discussed in Chapters 5 and 9.

Figure 21-6 models a family of signals that may be handled by an autonomous robot. Note that the root signal (`RobotSignal`) is abstract, which means that there may be no direct instances. This signal has two immediate concrete specializations (`Collision` and `HardwareFault`), one of which

(`HardwareFault`) is further specialized. Note that the `Collision` signal has one parameter.

Figure 21-6. Modeling Families of Signals



Modeling Abnormal Occurrences

An important part of visualizing, specifying, and documenting the behavior of a class or an interface is specifying the abnormal occurrences that its operations can produce. If you are handed a class or an interface, the operations you can invoke will be clear, but the abnormal occurrences that each operation may raise will not be clear unless you model them explicitly.

> *Classes are discussed in Chapters 4 and 9; interfaces are discussed in Chapter 11; stereotypes are discussed in Chapter 6.*

In the UML, abnormal occurrences are just additional kinds of events that can be modeled as signals. Error events may be attached to specification operations. Modeling exceptions is somewhat the inverse of modeling a general family of signals. You model a family of signals primarily to specify the kinds of signals an active object may receive; you model abnormal occurrences primarily to specify the kinds of abnormal occurrences that an object may produce.

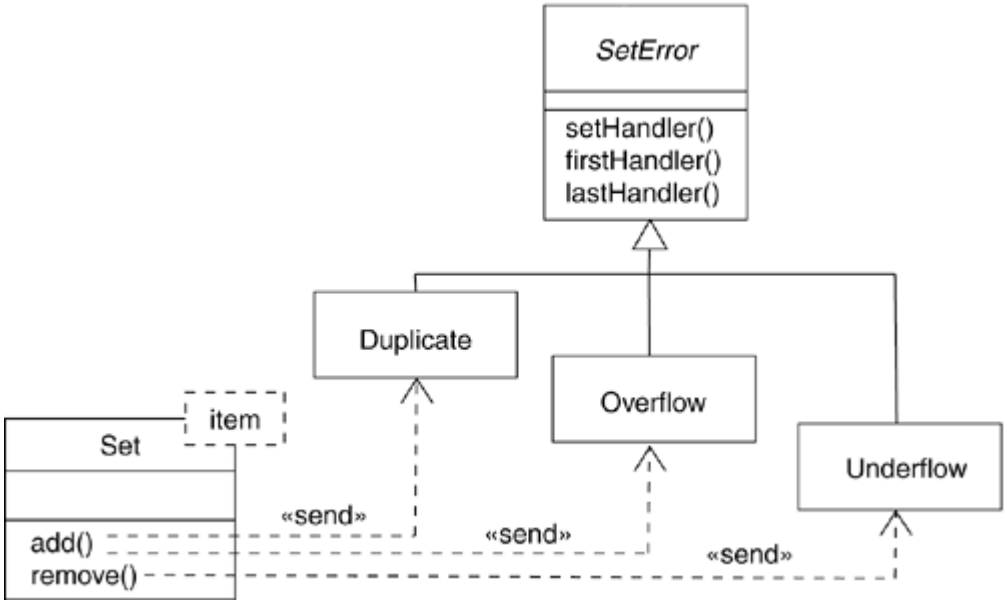
To model abnormal occurrences

- For each class and interface, and for each operation of such elements, consider the normal things that happen. Then think of things that can go wrong and model them as signals among objects.
- Arrange the signals in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions as necessary.

- For each operation, specify the abnormal occurrence signals that it may raise. You can do so explicitly (by showing `send` dependencies from an operation to its signals) or you can use sequence diagrams illustrating various scenarios.

Figure 21-7 models a hierarchy of abnormal occurrences that may be produced by a standard library of container classes, such as the template class `Set`. This hierarchy is headed by the abstract signal `Error` and includes three specialized kinds of errors: `Duplicate`, `Overflow`, and `Underflow`. As shown, the `add` operation may produce `Duplicate` and `Overflow` signals, and the `remove` operation produces only the `Underflow` signal. Alternatively, you could have put these dependencies in the background by naming them in each operation's specification. Either way, by knowing which signals each operation may send, you can create clients that use the `Set` class correctly.

Figure 21-7. Modeling Error Conditions



Template classes are discussed in Chapter 9.

Note

Signals, including abnormal occurrence signals, are asynchronous events between objects. UML also includes exceptions such as those found in Ada or C++. Exceptions are conditions that cause the mainline execution path to be abandoned and a secondary execution path executed instead. Exceptions are not signals; instead, they are a convenient mechanism for specifying an alternate flow of control within a single synchronous thread of execution.

Hints and Tips

When you model an event,

- Build hierarchies of signals so that you exploit the common properties of related signals.
- Be sure you have a suitable state machine behind each element that may receive the event.
- Be sure to model not only those elements that may receive events, but also those elements that may send them.

When you draw an event in the UML,

- In general, model hierarchies of events explicitly, but model their use in the backplane of each class that sends or receives such an event.

Chapter 22. State Machines

In this chapter

- States, transitions, and activities
- Modeling the lifetime of an object
- Creating well-structured algorithms

Using an interaction, you can model the behavior of a society of objects that work together. Using a state machine, you can model the behavior of an individual object. A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.

> *Interactions are discussed in Chapter 16; objects are discussed in Chapter 13.*

You use state machines to model the dynamic aspects of a system. For the most part, this involves specifying the lifetime of the instances of a class, a use case, or an entire system. These instances may respond to such events as signals, operations, or the passing of time. When an event occurs, some effect will take place, depending on the current state of the object. An *effect* is the specification of a behavior execution within a state machine. Effects ultimately resolve into in the execution of actions that change the state of an object or return values. A *state* of an object is a period of time during which it satisfies some condition, performs some activity, or waits for some event.

> *Classes are discussed in Chapters 4 and 9; use cases are discussed in Chapter 17; systems are discussed in Chapter 32; activity diagrams are discussed in Chapter 20; state diagrams are discussed in Chapter 25.*

You can visualize the dynamics of execution in two ways: by emphasizing the flow of control from activity to activity (using activity diagrams) or by emphasizing the potential states of the objects and the transitions among those states (using state diagrams).

Well-structured state machines are like well-structured algorithms: They are efficient, simple, adaptable, and understandable.

Getting Started

Consider the life of your home's thermostat on one crisp fall day. In the wee hours of the morning, things are pretty quiet for the humble thermostat. The temperature of the house is stable and, save for a rogue gust of wind or a passing storm, the temperature outside the house is stable, too. Toward dawn, however, things get more interesting. The sun starts to peek over the horizon, raising the ambient temperature slightly. Family members start to wake; someone might tumble out of bed and twist the thermostat's dial. Both of these events are significant to the home's heating and cooling system. The thermostat starts behaving like all good thermostats should, by commanding the home's heater to raise the inside temperature or the air conditioner to lower the inside temperature.

Once everyone has left for work or school, things get quiet, and the temperature of the house stabilizes once again. However, an automatic program might then cut in, commanding the thermostat to lower the temperature to save on electricity and gas. The thermostat goes back to work. Later in the day, the program comes alive again, this time commanding the thermostat to raise the temperature so that the family can come home to a cozy house.

In the evening, with the home filled with warm bodies and heat from cooking, the thermostat has a lot of work to do to keep the temperature even while it runs the heater and cooler efficiently. Finally, at night, things return to a quiet state.

A number of software-intensive systems behave just like that thermostat. A pacemaker runs continuously but adapts to changes in activity or heartbeat pattern. A network router runs continuously as well, silently guiding asynchronous streams of bits, sometimes adapting its behavior in response to commands from the network administrator. A cell phone works on demand, responding to input from the user and to messages from the local cells.

In the UML, you model the static aspects of a system by using such elements as class diagrams and object diagrams. These diagrams let you visualize, specify, construct, and document the things that live in your system, including classes, interfaces, components, nodes, and use cases and their instances, together with the way those things sit in relationship to one another.

> *Modeling the structural aspects of a system is discussed in Parts 2 and 3.*

In the UML, you model the dynamic aspects of a system by using state machines. Whereas an interaction models a society of objects that work together to carry out some action, a state machine models the lifetime of a single object, whether it is an instance of a class, a use case, or even an entire system. In the life of an object, it may be exposed to a variety of events, such as a signal, the invocation of an operation, the creation or destruction of the object, the passing of time, or the change in some condition. In response to these events, the object performs some action, which is a computation, and then it changes its state to a new value. The behavior of such an object is therefore affected by the past, at least as the past is reflected in the current state. An object may receive an event, respond with an action, then change its state. An object may receive another event and its response may be different, depending on its current state in response to the previous event.

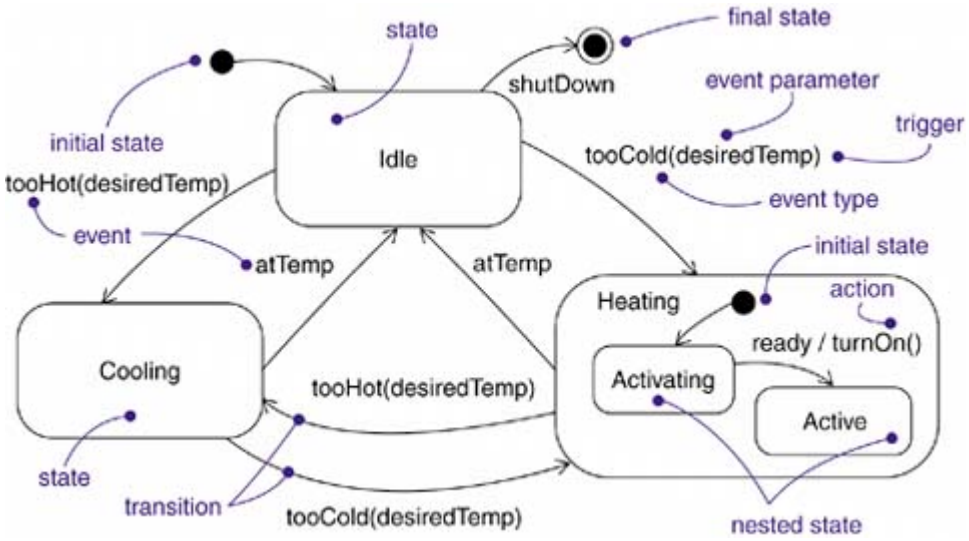
You can also model the dynamic aspects of a system by using interactions, as discussed in Chapter 16; events are discussed in Chapter 21.

You use state machines to model the behavior of any modeling element, most commonly a class, a use case, or an entire system. State machines may be visualized using state diagrams. You can focus on the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

Activity diagrams are discussed in Chapter 20; state diagrams are discussed in Chapter 25.

The UML provides a graphical representation of states, transitions, events, and effects, as Figure 22-1 shows. This notation permits you to visualize the behavior of an object in a way that lets you emphasize the important elements in the life of that object.

Figure 22-1. State Machines



PREV NEXT

Terms and Concepts

A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events. A *state* is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for events. An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. A *transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. An *activity* is ongoing nonatomic execution within a state machine. An *action* is an executable computation that results in a change in state of the model or the return of a value. Graphically, a state is rendered as a rectangle with rounded corners. A transition is rendered as a solid directed line or path from the original state to the new state.

Context

Every object has a lifetime. On creation, an object is born; on destruction, an object ceases to exist. In between, an object may act on other objects (by sending them messages) as well as be acted on (by being the target of a message). In many cases, these messages will be simple, synchronous operation calls. For example, an instance of the class `Customer` might invoke the operation `getAccountBalance` on an instance of the class `BankAccount`. Objects such as these don't need a state machine to specify their behavior because their current behavior does not depend on their past.

Objects are discussed in Chapter 13; messages are discussed in Chapter 16.

In other kinds of systems, you'll encounter objects that must respond to signals, which are asynchronous messages communicated between instances. For example, a cellular phone must respond to random phone calls (from other phones), keypad events (from the customer initiating a phone call), and to events from the network (when the phone moves from one call to another). Similarly, you'll encounter objects whose current behavior depends on their past behavior. For example, the behavior of an air-to-air missile guidance system will depend on its current state, such as `NotFlying` (it's not a good idea to launch a missile while it's attached to an aircraft that's still sitting on the ground) or `Searching` (you shouldn't arm the missile until you have a good idea what it's going to hit).

Signals are discussed in Chapter 21.

The behavior of an object that must respond to asynchronous messages or whose current behavior depends on its past is best specified by using a state machine. This encompasses instances of classes that can receive signals, including many active objects. In fact, an object that receives a signal but has no transition for that signal in its current state and does not defer the signal in that state will simply ignore that signal. In other words, the absence of a transition for a signal is not an error; it means that the signal is not of interest at that point. You'll also use state machines to model the behavior of

entire systems, especially reactive systems, which must respond to signals from actors outside the system.

Active objects are discussed in Chapter 23; modeling reactive systems is discussed in Chapter 25; use cases and actors are discussed in Chapter 17; interactions are discussed in Chapter 16; interfaces are discussed in Chapter 11.

Note

Most of the time, you'll use interactions to model the behavior of a use case, but you can also use state machines for the same purpose. Similarly, you can use state machines to model the behavior of an interface. Although an interface may not have any direct instances, a class that realizes such an interface may. Such a class must conform to the behavior specified by the state machine of this interface.

States

A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An object remains in a state for a finite amount of time. For example, a `Heater` in a home might be in any of four states: `Idle` (waiting for a command to start heating the house), `Activating` (its gas is on, but it's waiting to come up to temperature), `Active` (its gas and blower are both on), and `ShuttingDown` (its gas is off but its blower is on, flushing residual heat from the system).

When an object's state machine is in a given state, the object is said to be in that state. For example, an instance of `Heater` might be `Idle` or perhaps `ShuttingDown`.

You can visualize the state of an object in an interaction, as discussed in Chapter 13; the last four parts of a state are discussed in later sections of this chapter.

A state has several parts:

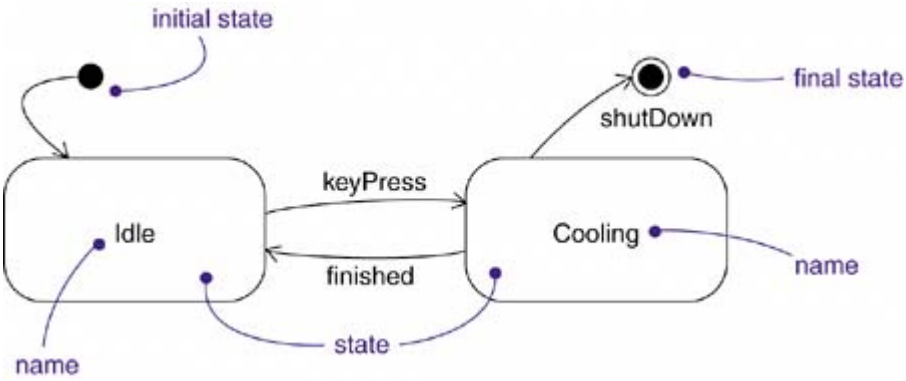
1. Name	A textual string that distinguishes the state from other states; a state may be anonymous, meaning that it has no name
2. Entry/exit effects	Actions executed on entering and exiting the state, respectively
3. Internal transitions	Transitions that are handled without causing a change in state
4. Substates	The nested structure of a state, involving nonorthogonal (sequentially active) or orthogonal (concurrently active) substates
5. Deferred events	A list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state

Note

A state name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the colon) and may continue over several lines. In practice, state names are short nouns or noun phrases drawn from the vocabulary of the system you are modeling. Typically, you capitalize the first letter of every word in a state name, as in `Idle` or `ShuttingDown`.

As Figure 22-2 shows, you represent a state as a rectangle with rounded corners.

Figure 22-2. States



Initial and Final States

As the figure shows, there are two special states that may be defined for an object's state machine. First, there's the initial state, which indicates the default starting place for the state machine or substate. An initial state is represented as a filled black circle. Second, there's the final state, which indicates that the execution of the state machine or the enclosing state has been completed. A final state is represented as a filled black circle surrounded by an unfilled circle (a bull's eye).

Note

Initial and final states are really pseudostates. Neither may have the usual parts of a normal state, except for a name. A transition from an initial state to an ordinary state may have the full complement of features, including a guard condition and action (but not a trigger event).

Transitions

A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to fire. Until the transition fires, the object is said to be in the source state; after it fires, it is said to be in the target state. For example, a `Heater`

might transition from the `Idle` to the `Activating` state when an event such as `tooCold` (with the parameter `desiredTemp`) occurs.

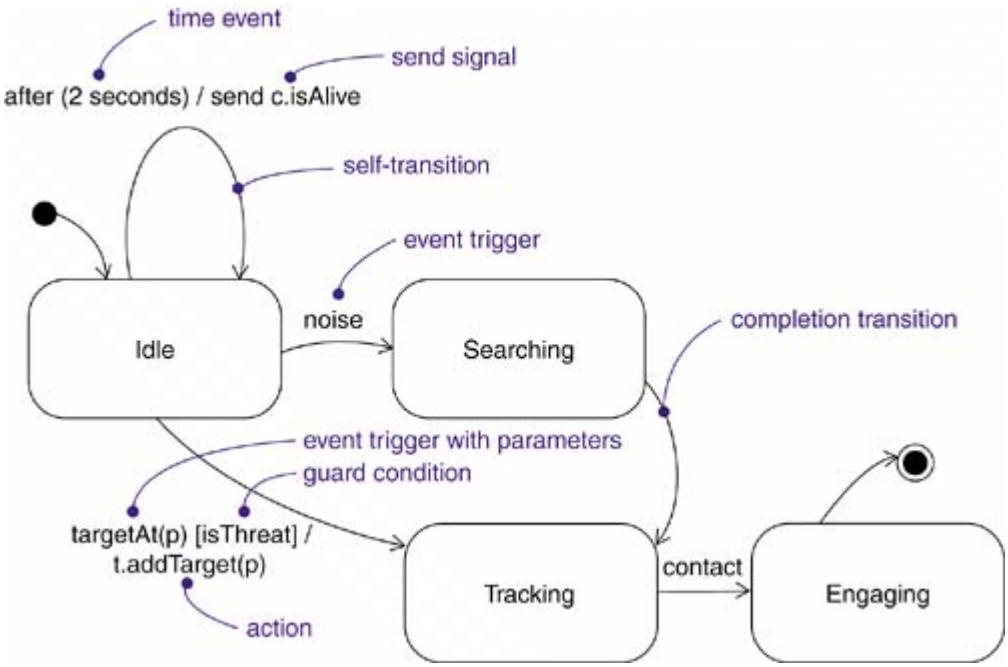
A transition has five parts.

1. Source state	The state affected by the transition; if an object is in the source state, an outgoing transition may fire when the object receives the trigger event of the transition and if the guard condition, if any, is satisfied
2. Event trigger	The event whose recognition by the object in the source state makes the transition eligible to fire, providing its guard condition is satisfied
3. Guard condition	A Boolean expression that is evaluated when the transition is triggered by the reception of the event trigger; if the expression evaluates true, the transition is eligible to fire; if the expression evaluates false, the transition does not fire, and if there is no other transition that could be triggered by that same event, the event is lost
4. Effect	An executable behavior, such as an action, that may act on the object that owns the state machine and indirectly on other objects that are visible to the object
5. Target state	The state that is active after the completion of the transition

Events are discussed in Chapter 21.

As Figure 22-3 shows, a transition is rendered as a solid directed line from the source to the target state. A self-transition is a transition whose source and target states are the same.

Figure 22-3. Transitions



Note

A transition may have multiple sources (in which case, it represents a join from multiple concurrent states) as well as multiple targets (in which case, it represents a fork to multiple concurrent states). See later discussion under orthogonal substates.

Event Trigger

An event is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. As shown in the previous figure, events may include signals, calls, the passing of time, or a change in state. A signal or a call may have parameters whose values are available to the transition, including expressions for the guard condition and action.

Events are discussed in Chapter 21.

Specifying a family of signals is discussed in Chapter 21; multiple, nonoverlapping guard conditions form a branch, as discussed in Chapter 20.

It is also possible to have a completion transition, represented by a transition with no event trigger. A completion transition is triggered implicitly when its source state has completed its behavior, if any.

Note

An event trigger may be polymorphic. For example, if you've specified a family of signals, then a transition whose trigger event is `S` can be triggered by `S`, as well as by any children of `S`.

Guard Condition

As the previous figure shows, a guard condition is rendered as a Boolean expression enclosed in square brackets and placed after the trigger event. A guard condition is evaluated only after the trigger event for its transition occurs. Therefore, it's possible to have multiple transitions from the same source state and with the same event trigger, as long as those conditions don't overlap.

A guard condition is evaluated just once for each transition at the time the event occurs, but it may be evaluated again if the transition is retriggered. Within the Boolean expression, you can include conditions about the state of an object (for example, the expression `aHeater in Idle`, which evaluates true if the `Heater` object is currently in the `Idle` state). If the condition is not true when it is tested, the event does not occur later when the condition becomes true. Use a change event to model that kind of behavior.

Change events are discussed in Chapter 21.

Note

Although a guard condition is evaluated only once each time its transition triggers, a change event is potentially evaluated continuously.

Effect

An effect is a behavior that is executed when a transition fires. Effects may include inline computation, operation calls (to the object that owns the state machine as well as to other visible objects), the creation or destruction of another object, or the sending of a signal to an object. To indicate sending a signal you can prefix the signal name with the keyword `send` as a visual cue.

Transitions only occur when the state machine is quiescent, that is, when it is not executing an effect from a previous transition. The execution of the effect of a transition and any associated entry and exit effects run to completion before any additional events are allowed to cause additional transitions. This is in contrast to a do-activity (described later in this chapter), which may be interrupted by events.

Activities are discussed in a later section of this chapter; dependencies are discussed in Chapters 5 and 10.

Note

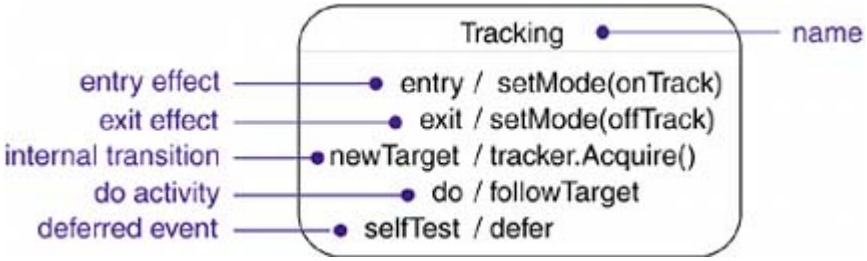
You can explicitly show the object to which a signal is sent by using a dependency stereotyped as `send`, whose source is the state and whose target is the object.

Advanced States and Transitions

You can model a wide variety of behavior using only the basic features of states and transitions in the UML. Using these features, you'll end up with flat state machines, which means that your behavioral models will consist of nothing more than arcs (transitions) and vertices (states).

UML state machines have a number of features that help you to manage complex behavioral models. These features often reduce the number of states and transitions you'll need, and they codify a number of common and somewhat complex idioms you'd otherwise encounter using flat state machines. Some of these advanced features include entry and exit effects, internal transitions, do-activities, and deferred events. These features are shown as text strings within a text compartment of the state symbol, as shown in Figure 22-4.

Figure 22-4. Advanced States and Transitions



Entry and Exit Effects

In a number of modeling situations, you'll want to perform some setup action whenever you enter a state, no matter which transition led you there. Similarly, when you leave a state, you'll want to perform some cleanup action no matter which transition led you away. For example, in a missile guidance system, you might want to explicitly announce the system is `onTrack` whenever it's in the `tracking` state, and `offTrack` whenever it's out of the state. Using flat state machines, you can achieve this effect by putting those actions on every entering and exiting transition, as appropriate. However, that's somewhat error prone; you have to remember to add these actions every time you add a new transition. Furthermore, modifying this action means that you have to touch every neighboring transition.

As Figure 22-4 shows, the UML provides a shorthand for this idiom. In the symbol for the state, you can include an entry effect (marked by the keyword `entry`) and an exit effect (marked by the keyword `exit`), each with its appropriate action. Whenever you enter the state, its entry action is dispatched; whenever you leave the state, its exit action is dispatched.

Entry and exit effects may not have arguments or guard conditions, but the entry effect at the top level of a state machine for a class may have parameters for the arguments that the machine receives

when the object is created.

Internal Transitions

Once inside a state, you'll encounter events you'll want to handle without leaving the state. These are called internal transitions, and they are subtly different from self-transitions. In a self-transition, such as you see in Figure 21-3, an event triggers the transition, you leave the state, an action (if any) is performed, and then you reenter the same state. Because this transition exits and then enters the state, a self-transition executes the state's exit action, then it executes the action of the self-transition, and finally, it executes the state's entry action.

However, suppose you want to handle the event but don't want to execute the state's entry and exit actions. The UML provides a shorthand for this idiom using an internal transition. An *internal transition* is a transition that responds to an event by performing an effect but does not change state. In Figure 21-4, the event `newTarget` labels an internal transition; if this event occurs while the object is in the `tracking` state, action `tracker.acquire` is executed but the state remains the same, and no entry or exit actions are executed. You indicate an internal transition by including a transition string (including an event name, optional guard condition, and effect) inside the symbol for a state instead of on a transition arrow. Note that the keywords `entry`, `exit`, and `do` are reserved words that may not be used as event names. Whenever you are in the state and an event labeling an internal transition occurs, the corresponding effect is performed without leaving and then reentering the state. Therefore, the event is handled without invoking the state's exit and then entry actions.

Note

Internal transitions may have events with parameters and guard conditions.

Do-Activities

When an object is in a state, it generally sits idle, waiting for an event to occur. Sometimes, however, you may wish to model an ongoing activity. While in a state, the object does some work that will continue until it is interrupted by an event. For example, if an object is in the `tracking` state, it might `followTarget` as long as it is in that state. As Figure 21-4 shows, in the UML you use the special `do` transition to specify the work that's to be done inside a state after the entry action is dispatched. You can also specify a behavior, such as a sequence of actionsfor example, do / op1(a); op2(b); op3(c). If the occurrence of an event causes a transition that forces an exit from the state, any ongoing do-activity of the state is immediately terminated.

Note

A do-activity is equivalent to an entry effect that starts the activity when the state is entered and an exit effect that stops the activity when the state is exited.

Deferred Events

Consider a state such as `tracking`. As illustrated in Figure 21-3, suppose there's only one transition leading out of this state, triggered by the event `contact`. While in the state `TRacking`, any events other than `contact` and other than those handled by its substates will be lost. That means that the event may occur, but it will be ignored and no action will result because of the presence of that event.

Events are discussed in Chapter 21.

In every modeling situation, you'll want to recognize some events and ignore others. You include those you want to recognize as the event triggers of transitions; those you want to ignore you just leave out. However, in some modeling situations, you'll want to accept some events but postpone a response to them until later. For example, while in the `TRacking` state, you may want to postpone a response to signals such as `selfTest`, perhaps sent by some maintenance agent in the system.

In the UML, you can specify this behavior by using deferred events. A deferred event is an event whose processing in the state is postponed until another state becomes active; if the event is not deferred in that state, the event is handled and may trigger transitions as if it had just occurred. If the state machine passes through a sequence of states in which the event is deferred, it is preserved until a state is finally encountered in which the event is not deferred. Other nondeferred events may occur during the interval. As you can see in Figure 21-4, you can specify a deferred event by listing the event with the special action `defer`. In this example, `selfTest` events may happen while in the `TRacking` state, but they are held until the object is in the `Engaging` state, at which time it appears as if they just occurred.

> **Note**
>
> The implementation of deferred events requires the presence of an internal queue of events. If an event happens but is listed as deferred, it is queued. Events are taken off this queue as soon as the object enters a state that does not defer these events.

Submachines

A state machine may be referenced within another state machine. Such a referenced state machine is called a *submachine*. They are useful in building large state models in a structured manner. See the *UML Reference Manual* for details.

Substates

These advanced features of states and transitions solve a number of common state machine modeling problems. However, there's one more feature of the UML's state machinessubstatesthat does even more to help you simplify the modeling of complex behaviors. A substate is a state that's nested inside another one. For example, a `Heater` might be in the `Heating` state, but also while in the `Heating` state, there might be a nested state called `Activating`. In this case, it's proper to say that the object is both `Heating` and `Activating`.

A simple state is a state that has no substructure. A state that has substatesthat is, nested statesis called a composite state. A composite state may contain either concurrent (orthogonal) or sequential (nonorthogonal) substates. In the UML, you render a composite state just as you do a simple state,

but with an optional graphic compartment that shows a nested state machine. Substates may be nested to any level.

Composite states have a nested structure similar to composition, as discussed in Chapters 5 and 10.

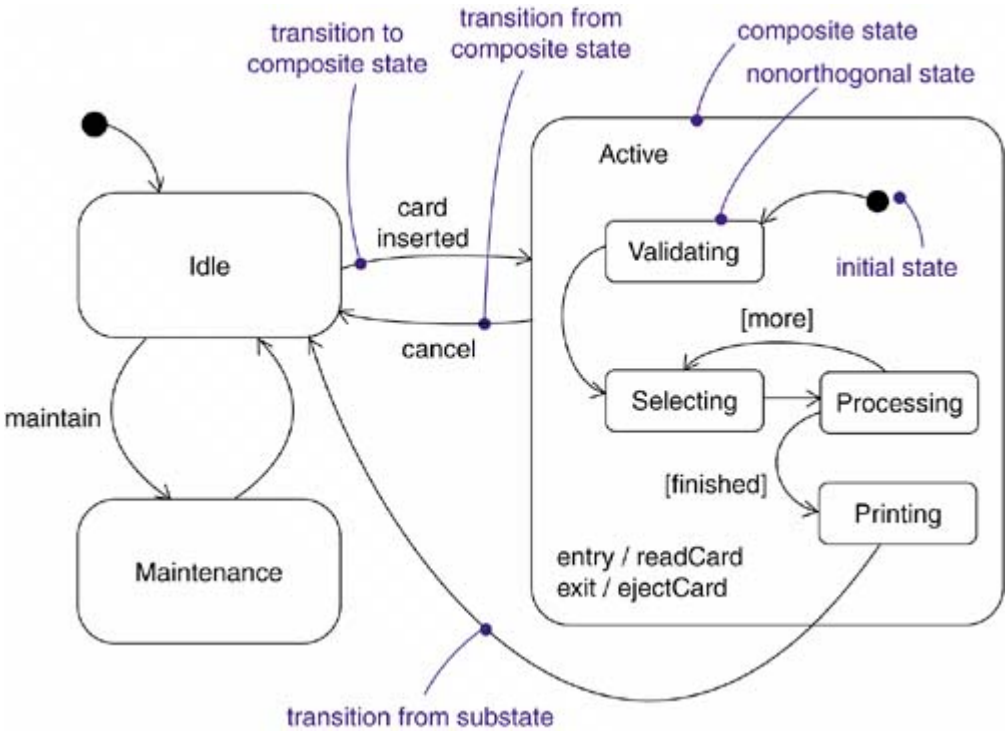
Nonorthogonal Substates

Consider the problem of modeling the behavior of an ATM. This system might be in one of three basic states: `Idle` (waiting for customer interaction), `Active` (handling a customer's transaction), and `Maintenance` (perhaps having its cash store replenished). While `Active`, the behavior of the ATM follows a simple path: Validate the customer, select a transaction, process the transaction, and then print a receipt. After printing, the ATM returns to the `Idle` state. You might represent these stages of behavior as the states `Validating`, `Selecting`, `Processing`, and `Printing`. It would even be desirable to let the customer select and process multiple transactions after `Validating` the account and before `Printing` a final receipt.

The problem here is that, at any stage in this behavior, the customer might decide to cancel the transaction, returning the ATM to its `Idle` state. Using flat state machines, you can achieve that effect, but it's quite messy. Because the customer might cancel the transaction at any point, you'd have to include a suitable transition from every state in the `Active` sequence. That's messy because it's easy to forget to include these transitions in all the right places, and many such interrupting events means you end up with a multitude of transitions zeroing in on the same target state from various sources, but with the same event trigger, guard condition, and action.

Using nested substates, there's a simpler way to model this problem, as Figure 22-5 shows. Here, the `Active` state has a substructure, containing the substates `Validating`, `Selecting`, `Processing`, and `Printing`. The state of the ATM changes from `Idle` to `Active` when the customer enters a credit card in the machine. On entering the `Active` state, the entry action `readCard` is performed. Starting with the initial state of the substructure, control passes to the `Validating` state, then to the `Selecting` state, and then to the `Processing` state. After `Processing`, control may return to `Selecting` (if the customer has selected another transaction) or it may move on to `Printing`. After `Printing`, there's a completion transition back to the `Idle` state. Notice that the `Active` state has an exit action, which ejects the customer's credit card.

Figure 22-5. Sequential Substates



Notice also the transition from the `Active` state to the `Idle` state, triggered by the event `cancel`. In any substate of `Active`, the customer might cancel the transaction, and that returns the ATM to the `Idle` state (but only after ejecting the customer's credit card, which is the exit action dispatched on leaving the `Active` state, no matter what caused a transition out of that state). Without substates, you'd need a transition triggered by `cancel` on every substructure state.

Substates such as `Validating` and `Processing` are called nonorthogonal, or disjoint, substates. Given a set of nonorthogonal substates in the context of an enclosing composite state, the object is said to be in the composite state and in only one of those substates (or the final state) at a time. Therefore, nonorthogonal substates partition the state space of the composite state into disjoint states.

From a source outside an enclosing composite state, a transition may target the composite state or it may target a substate. If its target is the composite state, the nested state machine must include an initial state, to which control passes after entering the composite state and after performing its entry action, if any. If its target is the nested state, control passes to the nested state after performing the entry action (if any) of the composite state and then the entry action (if any) of the substate.

A transition leading out of a composite state may have as its source the composite state or a substate. In either case, control first leaves the nested state (and its exit action, if any, is executed), then it leaves the composite state (and its exit action, if any, is executed). A transition whose source is the composite state essentially cuts short (interrupts) the activity of the nested state machine. The completion transition of a composite state is taken when control reaches the final substate within the composite state.

Note

A nested nonorthogonal state machine may have at most one initial substate and one final substate.

History States

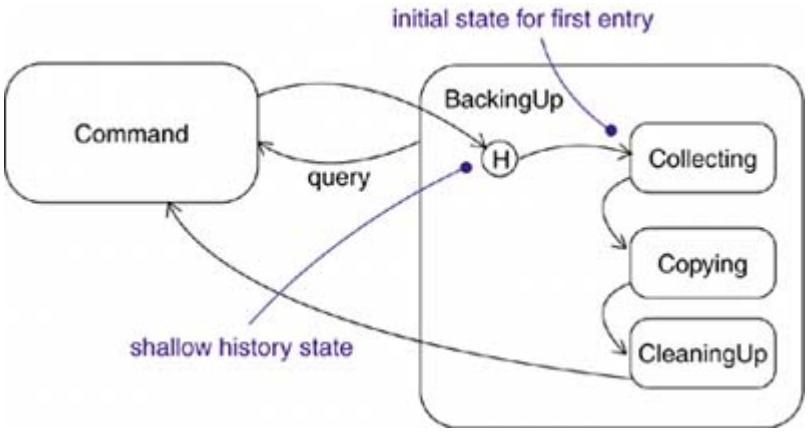
A state machine describes the dynamic aspects of an object whose current behavior depends on its past. A state machine in effect specifies the legal ordering of states an object may go through during its lifetime.

Unless otherwise specified, when a transition enters a composite state, the action of the nested state machine starts over again at its initial state (unless, of course, the transition targets a substate directly). However, there are times you'd like to model an object so that it remembers the last substate that was active prior to leaving the composite state. For example, in modeling the behavior of an agent that does an unattended backup of computers across a network, you'd like it to remember where it was in the process if it ever gets interrupted by, for example, a query from the operator.

Using flat state machines, you can model this, but it's messy. For each sequential substate, you'd need to have its exit action post a value to some variable local to the composite state. Then the initial state to this composite state would need a transition to every substate with a guard condition, querying the variable. In this way, leaving the composite state would cause the last substate to be remembered; entering the composite state would transition to the proper substate. That's messy because it requires you to remember to touch every substate and to set an appropriate exit action. It leaves you with a multitude of transitions fanning out from the same initial state to different target substates with very similar (but different) guard conditions.

In the UML, a simpler way to model this idiom is by using history states. A history state allows a composite state that contains nonorthogonal substates to remember the last substate that was active in it prior to the transition from the composite state. As Figure 22-6 shows, you represent a shallow history state as a small circle containing the symbol `H`.

Figure 22-6. History State



If you want a transition to activate the last substate, you show a transition from outside the composite state directly to the history state. The first time you enter a composite state, it has no history. This is the meaning of the single transition from the history state to a sequential substate such as `Collecting`. The target of this transition specifies the initial state of the nested state machine the first time it is entered. Continuing, suppose that while in the `BackingUp` state and the `Copying` state, the `query` event is posted. Control leaves `Copying` and `BackingUp` (dispatching their exit actions as necessary) and returns to the `Command` state. When the action of `Command` completes, the completion transition returns to the history state of the composite state `BackingUp`. This time, because there is a history to the nested state machine, control passes back to the `Copying` statethus bypassing the `Collecting` statebecause `Copying` was the last substate active prior to the transition from the state `BackingUp`.

Note

The symbol `H` designates a shallow history, which remembers only the history of the immediate nested state machine. You can also specify deep history, shown as a small circle containing the symbol `H*`. Deep history remembers down to the innermost nested state at any depth. If you have only one level of nesting, shallow and deep history states are semantically equivalent. If you have more than one level of nesting, shallow history remembers only the outermost nested state; deep history remembers the innermost nested state at any depth.

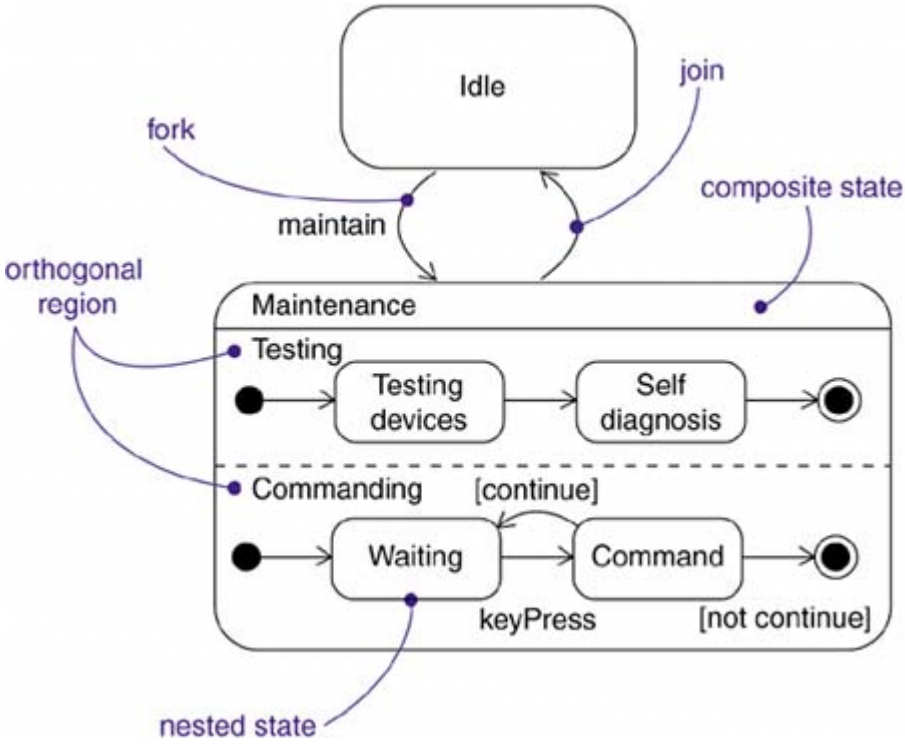
In either case, if a nested state machine reaches a final state, it loses its stored history and behaves as if it had not yet been entered for the first time.

Orthogonal Substates

Nonorthogonal substates are the most common kind of nested state machine you'll encounter. In certain modeling situations, however, you'll want to specify orthogonal regions. These regions let you specify two or more state machines that execute in parallel in the context of the enclosing object.

For example, Figure 22-7 shows an expansion of the `Maintenance` state from Figure 21-5. `Maintenance` is decomposed into two orthogonal regions, `Testing` and `Commanding`, shown by nesting them in the `Maintenance` state but separating them from one another with a dashed line. Each of these orthogonal regions is further decomposed into substates. When control passes from the `Idle` to the `Maintenance` state, control then forks to two concurrent flowsthe enclosing object will be in both the `Testing` region and the `Commanding` region. Furthermore, while in the `Commanding` region, the enclosing object will be in the `Waiting` or the `Command` state.

Figure 22-7. Concurrent Substates



Note

This is what distinguishes nonorthogonal substates and orthogonal substates. Given two or more nonorthogonal substates at the same level, an object will be in one of those substates or the other. Given two or more orthogonal regions at the same level, an object will be in a state from each of the orthogonal regions.

Execution of these two orthogonal regions continues in parallel. Eventually, each nested state machine reaches its final state. If one orthogonal region reaches its final state before the other, control in that region waits at its final state. When both nested state machines reach their final states, control from the two orthogonal regions joins back into one flow.

Whenever there's a transition to a composite state decomposed into orthogonal regions, control forks into as many concurrent flows as there are orthogonal regions. Similarly, whenever there's a transition from a composite substate decomposed into orthogonal regions, control joins back into one flow. This holds true in all cases. If all orthogonal regions reach their final states, or if there is an explicit transition out of the enclosing composite state, control joins back into one flow.

Note

Each orthogonal region may have an initial, final, and history state.

Fork and Join

Usually, entry to a composite state with orthogonal regions goes to the initial state of each orthogonal region. It is also possible to transition from an external state directly to one or more orthogonal states. This is called a fork, because control passes from a single state to several orthogonal states. It is shown as a heavy black line with one incoming arrow and several outgoing arrows, each to one of the orthogonal states. There must be at most one target state in each orthogonal region. If one or more orthogonal regions have no target states, then the initial state of those regions is implicitly chosen. A transition to a single orthogonal state within a composite state is also an implicit fork; the initial states of all the other orthogonal regions are implicitly part of the fork.

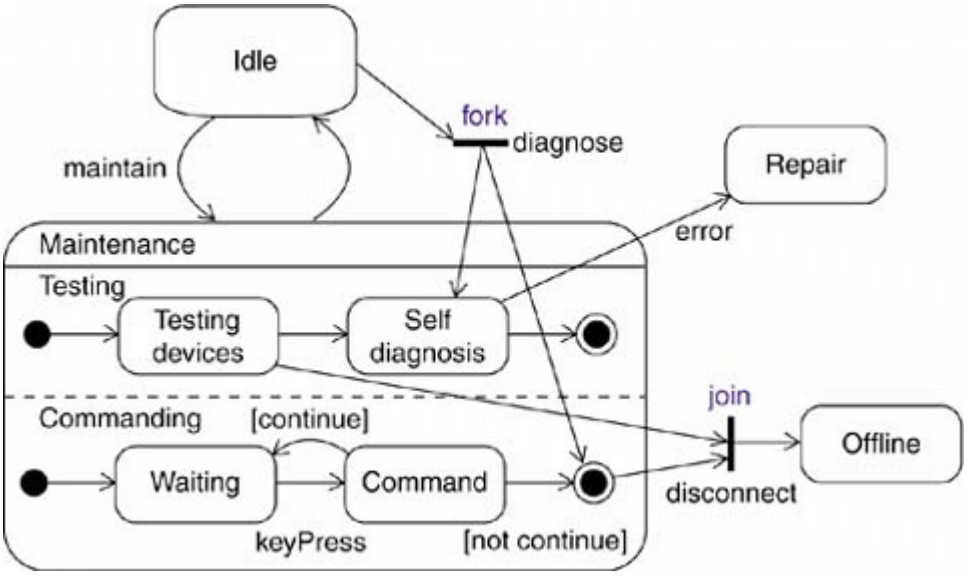
Similarly, a transition from any state within a composite state with orthogonal regions forces an exit from all the orthogonal regions. Such a transition often represents an error condition that forces termination of parallel computations.

A join is a transition with two or more incoming arrows and one outgoing arrow. Each incoming arrow must come from a state in a different orthogonal region of the same composite state. The join may have a trigger event. The join transition is effective only if all of the source states are active; the status of other orthogonal regions in the composite state is irrelevant. If the event occurs, control leaves all of the orthogonal regions in the composite state, not just the ones with arrows from them.

Figure 22-8 shows a variation on the previous example with explicit fork and join transitions. The transition `maintain` to the composite state `Maintenance` is still an implicit fork into the default initial states of the orthogonal regions. In this example, however, there is also an explicit fork from `Idle` into the two nested states `Self diagnose` and the final state of the `Commanding` region. (A final state is a real state and can be the target of a transition.) If an error event occurs while the `Self diagnose` state is active, the implicit join transition to `Repair` fires: Both the `Self diagnose` state and whatever state

is active in the `Commanding` region are exited. There is also an explicit join transition to the `Offline` state. This transition fires only if the `disconnect` event occurs while the `Testing devices` state and the final state of the `Commanding` region are both active; if both states are not active, the event has no effect.

Figure 22-8. Fork and join transitions



Active Objects

Another way to model concurrency is by using active objects. Thus, rather than partitioning one object's state machine into two (or more) concurrent regions, you could define two active objects, each of which is responsible for the behavior of one of the concurrent regions. If the behavior of one of these concurrent flows is affected by the state of the other, you'll want to model this using orthogonal regions. If the behavior of one of these concurrent flows is affected by messages sent to and from the other, you'll want to model this using active objects. If there's little or no communication between the concurrent flows, then the approach you choose is a matter of taste, although most of the time, using active objects makes your design decisions more obvious.

Active objects are discussed in Chapter 23.

PREV NEXT

Common Modeling Techniques

Modeling the Lifetime of an Object

The most common purpose for which you'll use state machines is to model the lifetime of an object, especially instances of classes, use cases, and the system as a whole. Whereas interactions model the behavior of a society of objects working together, a state machine models the behavior of a single object over its lifetime, such as you'll find with user interfaces, controllers, and devices.

When you model the lifetime of an object, you essentially specify three things: the events to which the object can respond, the response to those events, and the impact of the past on current behavior. Modeling the lifetime of an object also involves deciding on the order in which the object can meaningfully respond to events, starting at the time of the object's creation and continuing until its destruction.

To model the lifetime of an object,

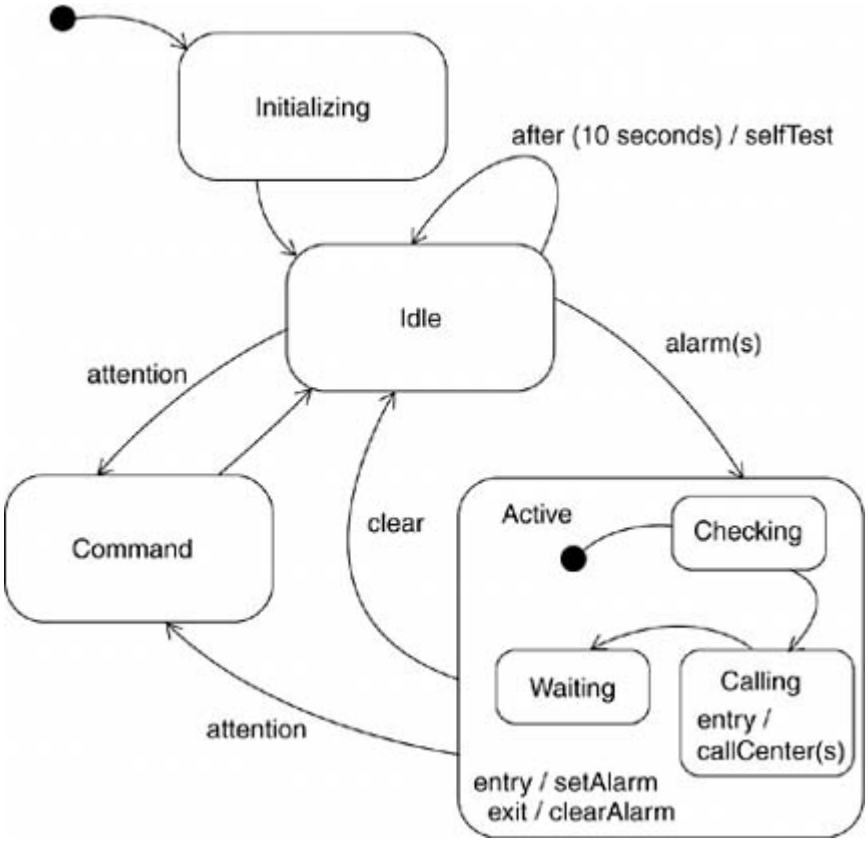
Objects are discussed in Chapter 13; classes are discussed in Chapters 4 and 9; use cases are discussed in Chapter 17; systems are discussed in Chapter 32; interactions are discussed in Chapter 16; collaborations are discussed in Chapter 28; pre- and postconditions are discussed in Chapter 10; interfaces are discussed in Chapter 11.

- Set the context for the state machine, whether it is a class, a use case, or the system as a whole.
    - If the context is a class or a use case, find the neighboring classes, including any parents of the class and any classes reachable by associations or dependences. These neighbors are candidate targets for actions and are candidates for including in guard conditions.
    - If the context is the system as a whole, narrow your focus to one behavior of the system. Theoretically, every object in the system may be a participant in a model of the system's lifetime, and except for the most trivial systems, a complete model would be intractable.
- Establish the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.
- Decide on the events to which this object may respond. If already specified, you'll find these in the object's interfaces; if not already specified, you'll have to consider which objects may interact with the object in your context, and then which events they may possibly dispatch.
- Starting from the initial state to the final state, lay out the top-level states the object may be in. Connect these states with transitions triggered by the appropriate events. Continue by adding actions to these transitions.
- Identify any entry or exit actions (especially if you find that the idiom they cover is used in the state machine).

- Expand these states as necessary by using substates.
- Check that all events mentioned in the state machine match events expected by the interface of the object. Similarly, check that all events expected by the interface of the object are handled by the state machine. Finally, look to places where you explicitly want to ignore events.
- Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.
- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses. Be especially diligent in looking for unreachable states and states in which the machine may get stuck.
- After rearranging your state machine, check it against expected sequences again to ensure that you have not changed the object's semantics.

For example, Figure 22-9 shows the state machine for the controller in a home security system, which is responsible for monitoring various sensors around the perimeter of the house.

Figure 22-9. Modeling the Lifetime of An Object



In the lifetime of this controller class, there are four main states: `Initializing` (the controller is starting up), `Idle` (the controller is ready and waiting for alarms or commands from the user), `Command` (the controller is processing commands from the user), and `Active` (the controller is processing an alarm condition). When the controller object is first created, it moves first to the `Initializing` state and then unconditionally to the `Idle` state. The details of these two states are not shown, other than the self-transition with the time event in the `Idle` state. This kind of time event is commonly found in embedded systems, which often have a heartbeat timer that causes a periodic check of the system's health.

Control passes from the `Idle` state to the `Active` state on receipt of an `alarm` event (which includes the parameter `s`, identifying the sensor that was tripped). On entering the `Active` state, `setAlarm` is performed as the entry action, and control then passes first to the `Checking` state (validating the alarm), then to the `Calling` state (calling the alarm company to register the alarm), and finally to the `Waiting` state. The `Active` and `Waiting` states are exited only upon `clear`ing the alarm or by the user signaling the controller for `attention`, presumably to issue a command.

Notice that there is no final state. That, too, is common in embedded systems, which are intended to run indefinitely.

PREV NEXT

Hints and Tips

When you model state machines in the UML, remember that every state machine represents the dynamic aspects of an individual object, typically representing an instance of a class, a use case, or the system as a whole. A well-structured state machine

- Is simple and therefore should not contain any superfluous states or transitions.
- Has a clear context and therefore may have access to all the objects visible to its enclosing object (these neighbors should be used only as necessary to carry out the behavior specified by the state machine).
- Is efficient and therefore should carry out its behavior with an optimal balance of time and resources as required by the actions it dispatches.

> *Modeling the vocabulary of a system is discussed in Chapter 4.*

- Is understandable and therefore should name its states and transitions from the vocabulary of the system.
- Is not nested too deeply (nesting substates at one or two levels will handle most complex behaviors).
- Uses orthogonal regions sparingly because using active classes is often a better alternative.

When you draw a state machine in the UML,

- Avoid transitions that cross.
- Expand composite states in place only as necessary to make the diagram understandable.

Chapter 23. Processes and Threads

In this chapter

- Active objects, processes, and threads
- Modeling multiple flows of control
- Modeling interprocess communication
- Building thread-safe abstractions

Not only is the real world a harsh and unforgiving place, but it is a very busy place as well. Events happen and things take place all at the same time. Therefore, when you model a system of the real world, you must take into account its process view, which encompasses the threads and processes that form the system's concurrency and synchronization mechanisms.

> *Interaction views in the context of software architecture are discussed in Chapter 2.*

In the UML, you model each independent flow of control as an active object that represents a process or thread that can initiate control activity. A process is a heavyweight flow that can execute concurrently with other processes; a thread is a lightweight flow that can execute concurrently with other threads within the same process.

Building abstractions so that they work safely in the presence of multiple flows of control is hard. In particular, you have to consider approaches to communication and synchronization that are more complex than for sequential systems. You also have to be careful to neither over-engineer your process view (too many concurrent flows and your system ends up thrashing) nor under-engineer it (insufficient concurrency does not optimize the system's throughput.

Getting Started

In the life of a dog and his doghouse, the world is a pretty simple and sequential place. Eat. Sleep. Chase a cat. Eat some more. Dream about chasing cats. Using the doghouse to sleep in or for shelter from the rain is never a problem because the dog, and only the dog, needs to go in and out through the doghouse door. There's never any contention for resources.

Modeling doghouses and high rises is discussed in Chapter 1.

In the life of a family and its house, the world is not so simple. Without getting too metaphysical, each family member lives his or her own life, yet still interacts with other members of the family (for dinner, watching television, playing games, cleaning). Family members will share certain resources. Children might share a bedroom; the whole family might share one phone or one computer. Family members will also share chores. Dad does the laundry and the grocery shopping; mom does the bills and the yard work; the children help with the cleaning and cooking. Contention among these shared resources and coordination among these independent chores can be challenging. Sharing one bathroom when everyone is getting ready to go to school or to work can be problematic; dinner won't be served if dad didn't first get the groceries.

In the life of a high rise and its tenants, the world is really complex. Hundreds, if not thousands, of people might work in the same building, each following his or her own agenda. All must pass through a limited set of entrances. All must jockey for the same bank of elevators. All must share the same heating, cooling, water, electrical, sanitation, and parking facilities. If they are to work together optimally, they have to communicate and synchronize their interactions properly.

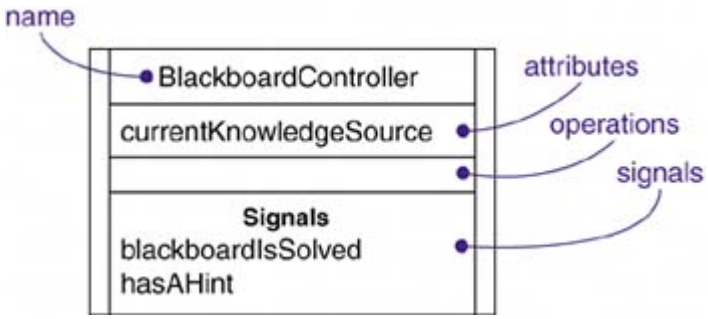
In the UML, each independent flow of control is modeled as an active object. An active object is a process or thread that can initiate control activity. As for every kind of object, an active object is an instance of a class. In this case, an active object is an instance of an active class. Also, as for every kind of object, active objects can communicate with one another by passing messages, although here, message passing must be extended with certain concurrency semantics to help you to synchronize the interactions among independent flows.

Objects are discussed in Chapter 13.

In software, many programming languages directly support the concept of an active object. Java, Smalltalk, and Ada all have concurrency built in. C++ supports concurrency through various libraries that build on a host operating system's concurrency mechanisms. Using the UML to visualize, specify, construct, and document these abstractions is important because without doing so, it's nearly impossible to reason about issues of concurrency, communication, and synchronization.

The UML provides a graphical representation of an active class, as Figure 23-1 shows. Active classes are kinds of classes, so they have all the usual compartments for class name, attributes, and operations. Active classes often receive signals, which you typically enumerate in an extra compartment.

Figure 23-1. Active Class



Classes are discussed in Chapters 4 and 9; signals are discussed in Chapter 21.

PREV NEXT

Terms and Concepts

An *active object* is an object that owns a process or thread and can initiate control activity. An *active class* is a class whose instances are active objects. A *process* is a heavyweight flow that can execute concurrently with other processes. A *thread* is a lightweight flow that can execute concurrently with other threads within the same process. Graphically, an active class is rendered as a rectangle with double lines for left and right sides. Processes and threads are rendered as stereotyped active classes (and also appear as sequences in interaction diagrams).

> *Interaction diagrams are discussed in Chapter 19.*

Flow of Control

In a purely sequential system, there is one flow of control. This means that one thing, and one thing only, can take place at a time. When a sequential program starts, control is rooted at the beginning of the program and operations are dispatched one after another. Even if there are concurrent things happening among the actors outside the system, a sequential program will process only one event at a time, queuing or discarding any concurrent external events.

> *Actors are discussed in Chapter 17.*

This is why it's called a flow of control. If you trace the execution of a sequential program, you'll see the locus of execution flow from one statement to another, in sequential order. You might see actions that branch, loop, and jump about, and if there is any recursion or iteration, you see the flow circle back on itself. Nonetheless, in a sequential system, there would be a single flow of execution.

> *Actions are discussed in Chapter 16.*

In a concurrent system, there is more than one flow of controlthat is, more than one thing can take place at a time. In a concurrent system, there are multiple simultaneous flows of control, each rooted at the head of an independent process or a thread. If you take a snapshot of a concurrent system while it's running, you'll logically see multiple loci of execution.

In the UML, you use an active class to represent a process or thread that is the root of an independent flow of control and that is concurrent with all peer flows of control.

Nodes are discussed in Chapter 27.

Note

You can achieve true concurrency in one of three ways: first, by distributing active objects across multiple nodes; second, by placing active objects on nodes with multiple processors; and third, by a combination of both methods.

Classes and Events

Active classes are just classes, albeit ones with a very special property. An active class represents an independent flow of control, whereas a plain class embodies no such flow. In contrast to active classes, plain classes are implicitly called passive because they cannot independently initiate control activity.

Classes are discussed in Chapters 4 and 9.

You use active classes to model common families of processes or threads. In technical terms, this means that an active objectan instance of an active classreifies (is a manifestation of) a process or thread. By modeling concurrent systems with active objects, you give a name to each independent flow of control. When an active object is created, the associated flow of control is started; when the active object is destroyed, the associated flow of control is terminated.

Objects are discussed in Chapter 13; attributes and operations are discussed in Chapter 4; relationships are discussed in Chapters 4 and 10; extensibility mechanisms are discussed in Chapter 6; interfaces are discussed in Chapter 11.

Active classes share the same properties as all other classes. Active classes may have instances. Active classes may have attributes and operations. Active classes may participate in dependency, generalization, and association (including aggregation) relationships. Active classes may use any of the UML's extensibility mechanisms, including stereotypes, tagged values, and constraints. Active classes may be the realization of interfaces. Active classes may be realized by collaborations, and the behavior of an active class may be specified by using state machines. Active classes may participate in collaborations.

In your diagrams, active objects may appear wherever passive objects appear. You can model the collaboration of active and passive objects by using interaction diagrams (including sequence and collaboration diagrams). An active object may appear as the target of an event in a state machine.

State machines are discussed in Chapter 22; events are discussed in Chapter 21.

Speaking of state machines, both passive and active objects may send and receive signal events and call events.

Note

The use of active classes is optional. They don't actually add much to the semantics.

Communication

When objects collaborate with one another, they interact by passing messages from one to the other. In a system with both active and passive objects, there are four possible combinations of interaction that you must consider.

Interactions are discussed in Chapter 16.

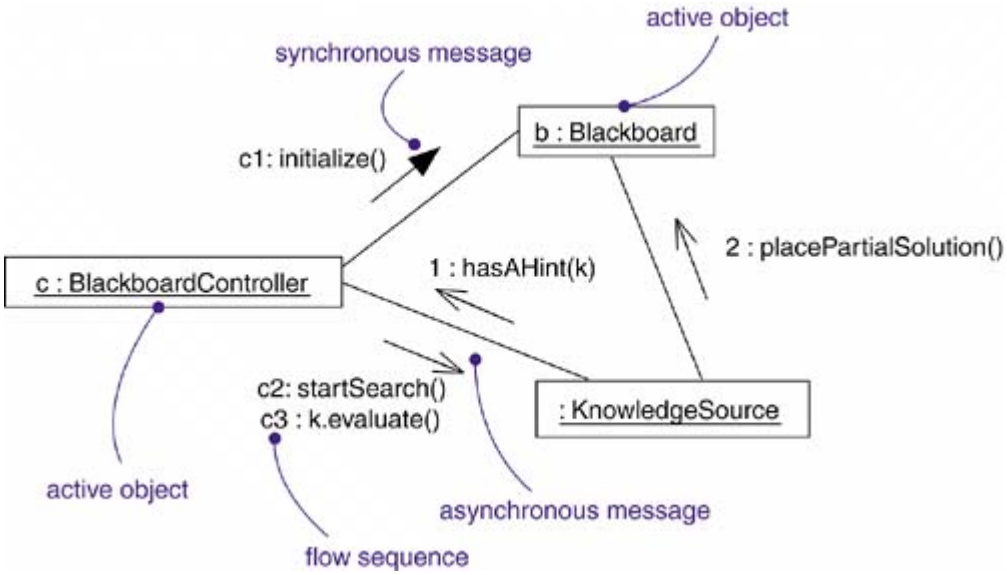
First, a message may be passed from one passive object to another. Assuming there is only one flow of control passing through these objects at a time, such an interaction is nothing more than the simple invocation of an operation.

Second, a message may be passed from one active object to another. When that happens, you have interprocess communication, and there are two possible styles of communication. First, one active object might synchronously call an operation of another. That kind of communication has rendezvous semantics, which means that the caller calls the operation; the caller waits for the receiver to accept the call; the operation is invoked; a method is chosen for execution based on the operation and the class of the receiver object; the method is executed; a return object (if any) is passed back to the caller; and then the two objects continue on their independent paths. For the duration of the call, the two flows of control are in lock step. Second, one active object might asynchronously send a signal or call an operation of another object. That kind of communication has mailbox semantics, which means that the caller sends the signal or calls the operation and then continues on its independent way. In the meantime, the receiver accepts the signal or call whenever it is ready (with intervening events or calls queued) and continues on its way after it is done. This is called a mailbox because the two objects are not synchronized; rather, one object drops off a message for the other.

Signal events and call events are discussed in Chapter 21.

In the UML, you render a synchronous message with a solid (filled) arrowhead and an asynchronous message as a stick arrowhead, as in Figure 23-2.

Figure 23-2. Communication



Third, a message may be passed from an active object to a passive object. A potential conflict arises if more than one active object at a time passes its flow of control through one passive object. It is an actual conflict if more than one object writes or reads and writes the same attributes. In that situation, you have to model the synchronization of these two flows very carefully, as discussed in the next section.

Fourth, a message may be passed from a passive object to an active one. At first glance, this may seem illegal, but if you remember that every flow of control is rooted in some active object, you'll understand that a passive object passing a message to an active object has the same semantics as an active object passing a message to an active object.

Constraints are discussed in Chapter 6.

Note

It is possible to model variations of synchronous and asynchronous message passing by using constraints. For example, to model a balking rendezvous as found in Ada, you'd use a synchronous message with a constraint such as `{wait = 0}`, saying that the caller will not wait for the receiver. Similarly, you can model a time out by using a constraint such as `{wait = 1 ms}`, saying that the caller will wait no more than one millisecond for the receiver to accept the message.

Synchronization

Visualize for a moment the multiple flows of control that weave through a concurrent system. When a

flow passes through an operation, we say that at a given moment, the locus of control is in the operation. If that operation is defined for some class, we can also say that at a given moment, the locus of control is in a specific instance of that class. You can have multiple flows of control in one operation (and therefore in one object), and you can have different flows of control in different operations (but still result in multiple flows of control in the one object).

The problem arises when more than one flow of control is in one object at the same time. If you are not careful, more than one flow might modify the same attribute, corrupting the state of the object or losing information. This is the classical problem of mutual exclusion. A failure to deal with it properly yields all sorts of race conditions and interference that cause concurrent systems to fail in mysterious and unrepeatable ways.

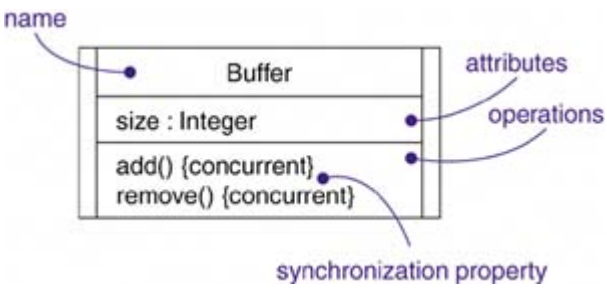
The key to this problem is serialization of access to the critical object. There are three approaches, each of which involves attaching certain synchronization properties to the operations defined in a class. In the UML, you can model all three approaches.

1. Sequential — Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semantics and integrity of the object cannot be guaranteed.
2. Guarded — The semantics and integrity of the object are guaranteed in the presence of multiple flows of control by sequentializing all calls to all of the object's guarded operations. In effect, exactly one operation at a time can execute on the object, reducing this to sequential semantics. There is a danger of deadlock if care is not taken.
3. Concurrent — The semantics and integrity of the object are guaranteed in the presence of multiple flows of control because multiple flows of control access disjoint sets of data or only read data. This situation can be arranged by careful design rules.

Some programming languages support these constructs directly. Java, for example, has the `synchronized` property, which is equivalent to the UML's `concurrent` property. In every language that supports concurrency, you can build support for all these properties by constructing them out of semaphores.

As Figure 23-3 shows, you can attach these properties to an operation, which you can render in the UML by using constraint notation. Note that concurrency must be asserted separately for each operation and for the entire object. Asserting concurrency for an operation means that multiple invocations of that operation can execute concurrently without danger. Asserting concurrency for an object means that invocations of different operations can execute concurrently without danger; this is a more stringent condition.

Figure 23-3. Synchronization



Constraints are discussed in Chapter 6.

Note

It is possible to model variations of these synchronization primitives by using constraints. For example, you might modify the `concurrent` property by allowing multiple simultaneous readers but only a single writer.

PREV NEXT

Common Modeling Techniques

Modeling Multiple Flows of Control

Building a system that encompasses multiple flows of control is hard. Not only do you have to decide how best to divide work across concurrent active objects, but once you've done that, you also have to devise the right mechanisms for communication and synchronization among your system's active and passive objects to ensure that they behave properly in the presence of these multiple flows. For that reason, it helps to visualize the way these flows interact with one another. You can do that in the UML by applying class diagrams (to capture their static semantics) and interaction diagrams (to capture their dynamic semantics) containing active classes and objects.

> *Mechanisms are discussed in Chapter 29; class diagrams are discussed in Chapter 8; interaction diagrams are discussed in Chapter 19.*

To model multiple flows of control,

- Identify the opportunities for concurrent execution and reify each flow as an active class. Generalize common sets of active objects into an active class. Be careful not to over-engineer your system by introducing unnecessary concurrency.

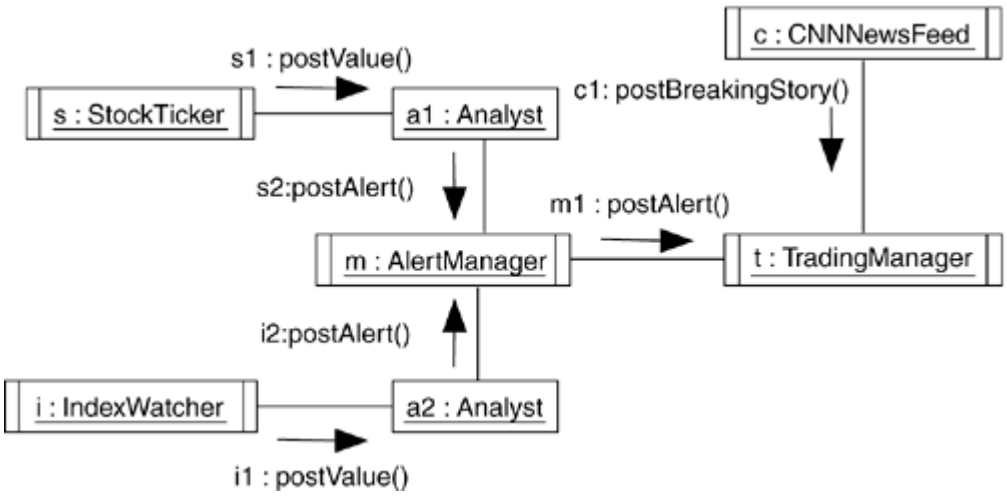
  > *Process views are discussed in Chapter 19; classes are discussed in Chapters 4 and 9; relationships are discussed in Chapters 5 and 10.*

- Consider a balanced distribution of responsibilities among these active classes, then examine the other active and passive classes with which each collaborates statically. Ensure that each active class is both tightly cohesive and loosely coupled relative to these neighboring classes and that each has the right set of attributes, operations, and signals.
- Capture these static decisions in class diagrams, explicitly highlighting each active class.
- Consider how each group of classes collaborates with one another dynamically. Capture those decisions in interaction diagrams. Explicitly show active objects as the root of such flows. Identify each related sequence by identifying it with the name of the active object.
- Pay close attention to communication among active objects. Apply synchronous and asynchronous messaging, as appropriate.
- Pay close attention to synchronization among these active objects and the passive objects with which they collaborate. Apply sequential, guarded, or concurrent operation semantics, as appropriate.

For example, Figure 23-4 shows part of the process view of a trading system. You'll find three objects

that push information into the system concurrently: a `StockTicker`, an `IndexWatcher`, and a `CNNNewsFeed` (named `s`, `i`, and `c`, respectively). Two of these objects (`s` and `i`) communicate with their own `Analyst` instances (`a1` and `a2`). At least as far as this model goes, the `Analyst` can be designed under the simplifying assumption that only one flow of control will be active in its instances at a time. Both `Analyst` instances, however, communicate simultaneously with an `AlertManager` (named `m`). Therefore, `m` must be designed to preserve its semantics in the presence of multiple flows. Both `m` and `c` communicate simultaneously with `t`, a `tradingManager`. Each flow is given a sequence number that is distinguished by the flow of control that owns it.

Figure 23-4. Modeling Flows of Control



Note

Interaction diagrams such as these are useful in helping you to visualize where two flows of control might cross paths and, therefore, where you must pay particular attention to the problems of communication and synchronization. Tools are permitted to offer even more distinct visual cues, such as by coloring each flow in a distinct way.

In diagrams such as this, it's also common to attach corresponding state machines, with orthogonal states showing the detailed behavior of each active object.

State machines are discussed in Chapter 22.

Modeling Interprocess Communication

As part of incorporating multiple flows of control in your system, you also have to consider the mechanisms by which objects that live in separate flows communicate with one another. Across threads (which live in the same address space), objects may communicate via signals or call events, the latter of which may exhibit either asynchronous or synchronous semantics. Across processes

(which live in separate address spaces), you usually have to use different mechanisms.

> *Signals and call events are discussed in Chapter 21.*

The problem of interprocess communication is compounded by the fact that, in distributed systems, processes may live on separate nodes. Classically, there are two approaches to interprocess communication: message passing and remote procedure calls. In the UML, you still model these as asynchronous or synchronous events, respectively. But because these are no longer simple in-process calls, you need to adorn your designs with further information.

> *Modeling location is discussed in Chapter 24.*

To model interprocess communication,

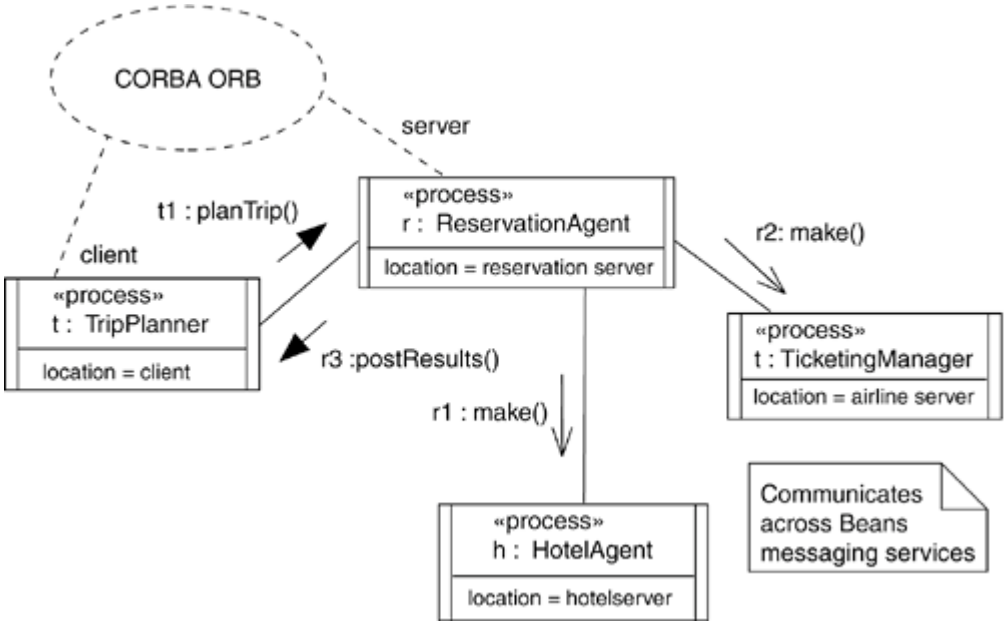
- Model the multiple flows of control.

  > *Stereotypes are discussed in Chapter 6; notes are discussed in Chapter 6; collaborations are discussed in Chapter 28; nodes are discussed in Chapter 27.*

- Model messaging using asynchronous communication; model remote procedure calls using synchronous communication.
- Informally specify the underlying mechanism for communication by using notes, or more formally by using collaborations.

Figure 23-5 shows a distributed reservation system with processes spread across four nodes. Each object is marked using the `process` stereotype. Each object is marked with a `location` attribute, specifying its physical location. Communication among the `ReservationAgent`, `TicketingManager`, and `HotelAgent` is asynchronous. Communication is described in a note as building on a Java Beans messaging service. Communication between the `tripPlanner` and the `ReservationSystem` is synchronous. The semantics of their interaction is found in the collaboration named `CORBA ORB`. The `TRipPlanner` acts as a `client`, and the `ReservationAgent` acts as a `server`. By zooming into the collaboration, you'll find the details of how this server and client collaborate.

Figure 23-5. Modeling Interprocess Communication


CORBA ORB
server
client
t1 : planTrip()
«process»
r : ReservationAgent
location = reservation server
r2: make()
«process»
t : TripPlanner
location = client
r3 :postResults()
«process»
t : TicketingManager
location = airline server
r1 : make()
«process»
h : HotelAgent
location = hotelserver
Communicates
across Beans
messaging services

PREV NEXT

Hints and Tips

A well-structured active class and active object

- Represents an independent flow of control that maximizes the potential for true concurrency in the system.
- Is not so fine-grained that it requires a multitude of other active elements that might result in an over-engineered and fragile process architecture.
- Carefully manages communication among peer active elements, choosing between asynchronous and synchronous messaging.
- Carefully treats each object as a critical region, using suitable synchronization properties to preserve its semantics in the presence of multiple flows of control.

When you draw an active class or an active object in the UML,

- Mark those attributes, operations, and signals that are important in understanding the abstraction in its context and hide the others using a filtering capability, if your modeling tool allows.
- Explicitly show all operation synchronization properties.

Chapter 24. Time and Space

In this chapter

The real world is a harsh and unforgiving place. Events may happen at unpredictable times yet demand specific responses at specific times. A system's resources may have to be distributed around the worldsome of those resources might even move aboutraising issues of latency, synchronization, security, and quality of service.

Modeling time and space is an essential element of any real time and/or distributed system. You use a number of the UML's features, including timing marks, time expressions, constraints, and tagged values, to visualize, specify, construct, and document these systems.

Dealing with real time and distributed systems is hard. Good models reveal the properties of a system's time and space characteristics.

Getting Started

When you start to model most software systems, you can usually assume a frictionless environmentmessages are sent in zero time, networks never go down, workstations never fail, the load across your network is always evenly balanced. Unfortunately, the real world does not work that waymessages do take time to deliver (and, sometimes, never get delivered), networks do go down, workstations do fail, and a network's load is often unbalanced. Therefore, when you encounter systems that must operate in the real world, you have to take into account the issues of time and space.

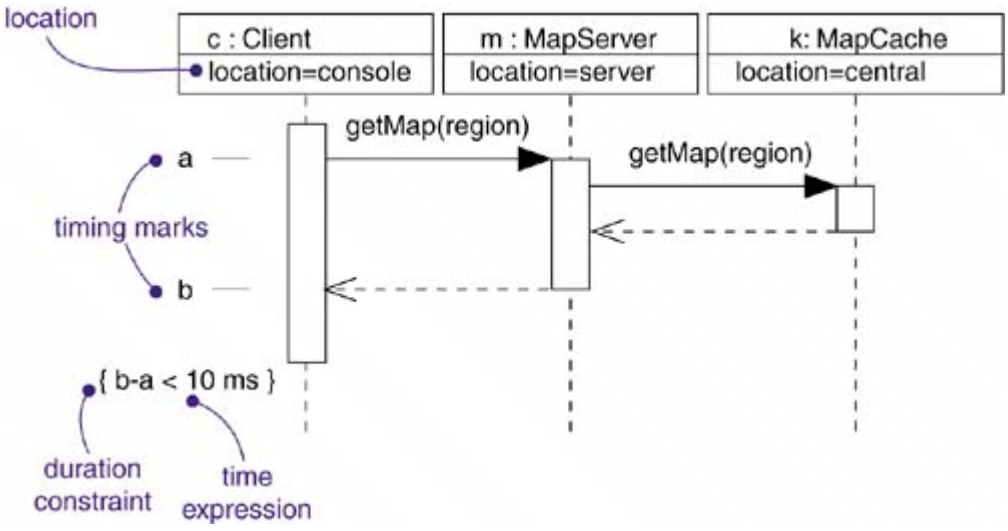
A real-time system is one in which certain behavior must be carried out at a precise absolute or relative time and within a predictable, often constrained, duration. At one extreme, such systems may be hard real time and require complete and repeatable behavior within nanoseconds or milliseconds. At the other extreme, models may be near real time and also require predictable behavior, but on the order of seconds or longer.

A distributed system is one in which components may be physically distributed across nodes. These nodes may represent different processors physically located in the same box, or they may even represent computers that are located half a world away from one another.

> *Components are discussed in Chapter 26; nodes are discussed in Chapter 27.*

To represent the modeling needs of real time and distributed systems, the UML provides a graphic representation for timing marks, time expressions, timing constraints, and location, as Figure 24-1 shows.

Figure 24-1. Timing Constraints and Location



Terms and Concepts

A *timing mark* is a denotation for the time at which an event occurs. Graphically, a timing mark is depicted as a small hash mark (horizontal line) on the border of a sequence diagram. A *time expression* is an expression that evaluates to an absolute or relative value of time. A time expression can also be formed using the name of a message and an indication of a stage in its processing, for example, `request.sendTime` or `request.receiveTime`. A *timing constraint* is a semantic statement about the relative or absolute value of time. Graphically, a timing constraint is rendered as for any constraint-that is, a string enclosed by brackets and generally connected to an element by a dependency relationship. *Location* is the placement of a component on a node. Location is an attribute of an object.

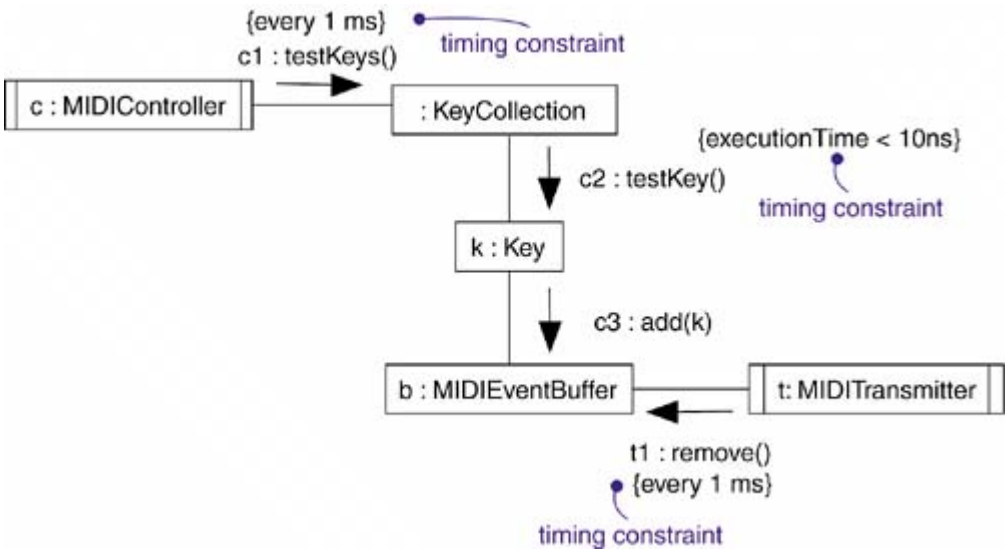
Time

Real time systems are, by their very name, time-critical systems. Events may happen at regular or irregular times; the response to an event must happen at predictable absolute times or at predictable times relative to the event itself.

> *Events, including time events, are discussed in Chapter 21; messages and interactions are discussed in Chapter 16; constraints are discussed in Chapter 6.*

The passing of messages represents the dynamic aspect of any system, so when you model the time-critical nature of a system with the UML, you can give a name to each message in an interaction to be used in time expressions. Messages in an interaction are usually not given names. They are mainly rendered with the name of an event, such as a signal or a call. However, you can give them names to write a time expression because the same event may trigger different messages. If the designated message is ambiguous, use the explicit name of the message in an expression to designate the message you want to mention in a time expression. Given a message name, you can refer to any of three functions of that messagethat is, `sendTime`, `receiveTime`, and `transmissionTime`. (These are our suggested functions, not official UML functions. A real-time system might have even more functions.) For synchronous calls, you can also reference the round-trip message time with `executionTime` (again our suggestion). You can then use these functions to specify arbitrarily complex time expressions, perhaps even using weights or offsets that are either constants or variables (as long as those variables can be bound at execution time). Finally, as shown in Figure 24-2, you can place these time expressions in a timing constraint to specify the timing behavior of the system. As constraints, you can render them by placing them adjacent to the appropriate message, or you can explicitly attach them using dependency relationships.

Figure 24-2. Time



Note

Especially for complex systems, it's a good idea to write expressions with named constants instead of writing explicit times. You can define those constants in one part of your model and then refer to those constants in multiple places. In that way, it's easier to update your model if the timing requirements of your system change.

Location

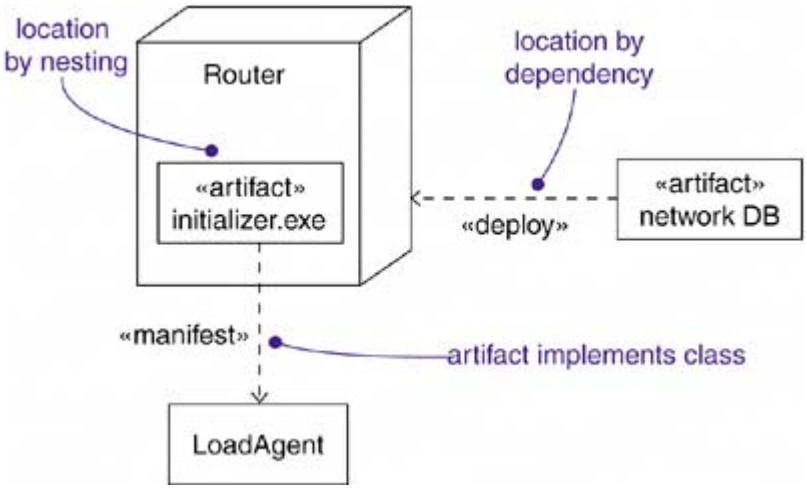
Distributed systems, by their nature, encompass components that are physically scattered among the nodes of a system. For many systems, components are fixed in place at the time they are loaded on the system; in other systems, components may migrate from node to node.

Components are discussed in Chapter 15; nodes are discussed in Chapter 27; deployment diagrams are discussed in Chapter 31; the class/object dichotomy is discussed in Chapters 2 and 13; classes are discussed in Chapters 4 and 9.

In the UML, you model the deployment view of a system by using deployment diagrams that represent the topology of the processors and devices on which your system executes. Artifacts such as executables, libraries, and tables reside on these nodes. Each instance of a node will own instances of certain artifacts, and each instance of an artifact will be owned by exactly one instance of a node (although instances of the same kind of artifact may be spread across different nodes).

Components and classes may be manifested as artifacts. For example, as Figure 24-3 shows, class `LoadAgent` is manifested by the artifact `initializer.exe` that lives on the node of type `Router`.

Figure 24-3. Location



As the figure illustrates, you can model the location of an artifact in two ways in the UML. First, as shown for the `Router`, you can physically nest the element (textually or graphically) in a extra compartment in its enclosing node. Second, you can use a dependency with the keyword `«deploy»` from the artifact to the node that contains it.

PREV NEXT

Common Modeling Techniques

Modeling Timing Constraints

Modeling the absolute time of an event and modeling the relative time between events are the primary time-critical properties of real time systems for which you'll use timing constraints.

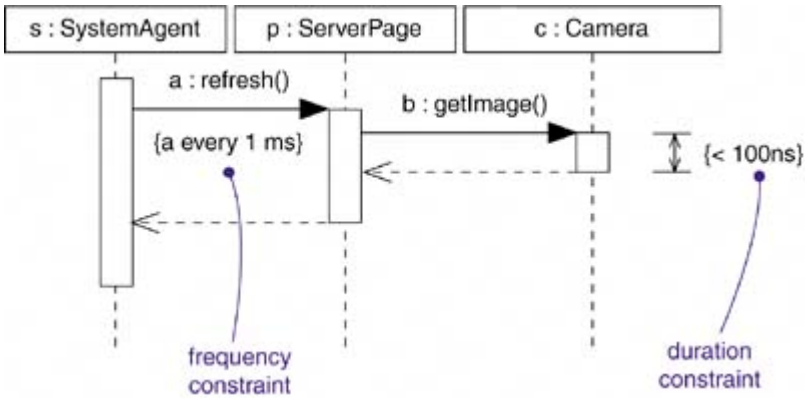
> *Constraints, one of the UML's extensibility mechanisms, are discussed in Chapter 6.*

To model timing constraints,

- For each event in an interaction, consider whether it must start at some absolute time. Model that real time property as a timing constraint on the message.
- For each interesting sequence of messages in an interaction, consider whether there is an associated maximum relative time for that sequence. Model that real time property as a timing constraint on the sequence.

For example, as shown in Figure 24-4, the left-most constraint specifies the repeating start time for the call event `refresh`. Similarly, the right timing constraint specifies the maximum duration for calls to `getImage`.

Figure 24-4. Modeling Timing Constraint



Often, you'll choose short names for messages so that you don't confuse them with operation names.

Modeling the Distribution of Objects

When you model the topology of a distributed system, you'll want to consider the physical placement of both nodes and artifacts. If your focus is the configuration management of the deployed system, modeling the distribution of nodes is especially important in order to visualize, specify, construct, and document the placement of physical things such as executables, libraries, and tables. If your focus is the functionality, scalability, and throughput of the system, modeling the distribution of objects is what's important.

> *Modeling the distribution of a component is discussed in Chapter 15.*

Deciding how to distribute the objects in a system is a difficult problem, and not just because the problems of distribution interact with the problems of concurrency. Naive solutions tend to yield profoundly poor performance, and over-engineered solutions aren't much better. In fact, they are probably worse because they usually end up being brittle.

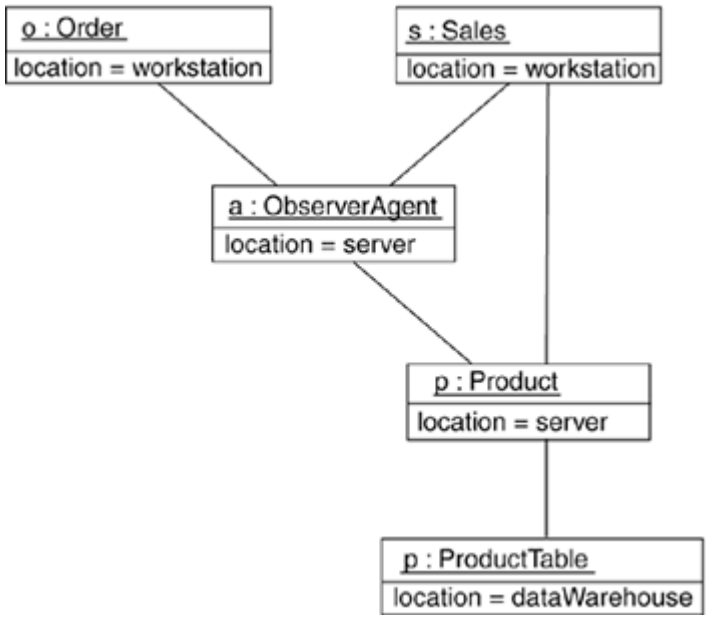
> *Modeling processes and threads is discussed in Chapter 23.*

To model the distribution of objects,

- For each interesting class of objects in your system, consider its locality of reference. In other words, consider all its neighbors and their locations. A tightly coupled locality will have neighboring objects close by; a loosely coupled one will have distant objects (and thus there will be latency in communicating with them). Tentatively allocate objects closest to the actors that manipulate them.
- Next consider patterns of interaction among related sets of objects. Colocate sets of objects that have high degrees of interaction, to reduce the cost of communication. Partition sets of objects that have low degrees of interaction.
- Next consider the distribution of responsibilities across the system. Redistribute your objects to balance the load of each node.
- Consider also issues of security, volatility, and quality of service, and redistribute your objects as appropriate.
- Assign objects to artifacts so that tightly coupled objects are on the same artifact.
- Assign artifacts to nodes so that the computation needs of each node are within capacity. Add additional nodes if necessary.
- Balance performance and communication costs by assigning tightly coupled artifacts to the same node.

Figure 24-5 provides an object diagram that models the distribution of certain objects in a retail system. The value of this diagram is that it lets you visualize the physical distribution of certain key objects. As the diagram shows, two objects reside on a `Workstation` (the `Order` and `Sales` objects), two objects reside on a `Server` (the `ObserverAgent` and the `Product` objects), and one object resides on a `DataWarehouse` (the `ProductTable` object).

Figure 24-5. Modeling the Distribution of Objects



Object diagrams are discussed in Chapter 14.

PREV NEXT

Hints and Tips

A well-structured model with time and space properties

- Exposes only those time and space properties that are necessary and sufficient to capture the desired behavior of the system.
- Centralizes the use of those properties so that they are easy to find and easy to modify.

When you draw a time or space property in the UML,

- Give your timing marks (the names of messages) meaningful names.
- Clearly distinguish between relative and absolute time expressions.
- Show space properties only when it's important to visualize the placement of elements across a deployed system.
- For more advanced needs, consider the *UML Profile for Schedulability, Performance, and Time*. This OMG specification addresses the needs of real-time and high-performance reactive systems.

Chapter 25. State Diagrams

In this chapter

Sequence diagrams, communication diagrams, activity diagrams, and use case diagrams also model the dynamic aspects of systems. Sequence diagrams and communication diagrams are discussed in Chapter 19; activity diagrams are discussed in Chapter 20; use case diagrams are discussed in Chapter 18.

State diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. A state diagram shows a state machine. Both activity and state diagrams are useful in modeling the lifetime of an object. However, whereas an activity diagram shows flow of control from activity to activity across various objects, a state diagram shows flow of control from state to state within a single object.

You use state diagrams to model the dynamic aspects of a system. For the most part, this involves modeling the behavior of reactive objects. A reactive object is one whose behavior is best characterized by its response to events dispatched from outside its context. A reactive object has a clear lifetime whose current behavior is affected by its past. State diagrams may be attached to classes, use cases, or entire systems in order to visualize, specify, construct, and document the dynamics of an individual object.

State diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.

Getting Started

Consider the investor who finances the building of a new high rise. She is unlikely to be interested in the details of the building process. The selection of materials, the scheduling of the trades, and the many meetings about engineering details are activities that are important to the builder, but far less so to the person bankrolling the project.

> *The differences between building a dog house and building a high rise are discussed in Chapter 1.*

The investor is interested in getting a good return on the investment, and that also means protecting the investment against risk. A very trusting investor will give a builder a pile of money, walk away for a while, and return only when the builder is ready to hand over the keys to the building. Such an investor is really interested in the final state of the building.

A more pragmatic investor will still trust the builder, but will also want to verify that the project is on track before releasing money. So, rather than give the builder an unattended pile of money to dip into, the prudent investor will set up clear milestones for the project, each of which is tied to the completion of certain activities, and only after meeting each one will money be released to the builder for the next phase of the project. For example, a modest amount of funds might be released at the project's inception to fund the architectural work. After the architectural vision has been approved, then more funds may be released to pay for the engineering work. After that work is completed to the project stakeholders satisfaction, a larger pile of money may be released so that the builder can proceed with breaking ground.

Along the way, from ground breaking to issuance of the certificate of occupancy, there are other milestones. Each of these milestones names a stable state of the project: architecture complete, engineering done, ground broken, infrastructure completed, building sealed, and so on. For the investor, following the changing state of the building is more important than following the flow of activities, which is what the builder might be doing by using Pert charts to model the workflow of the project.

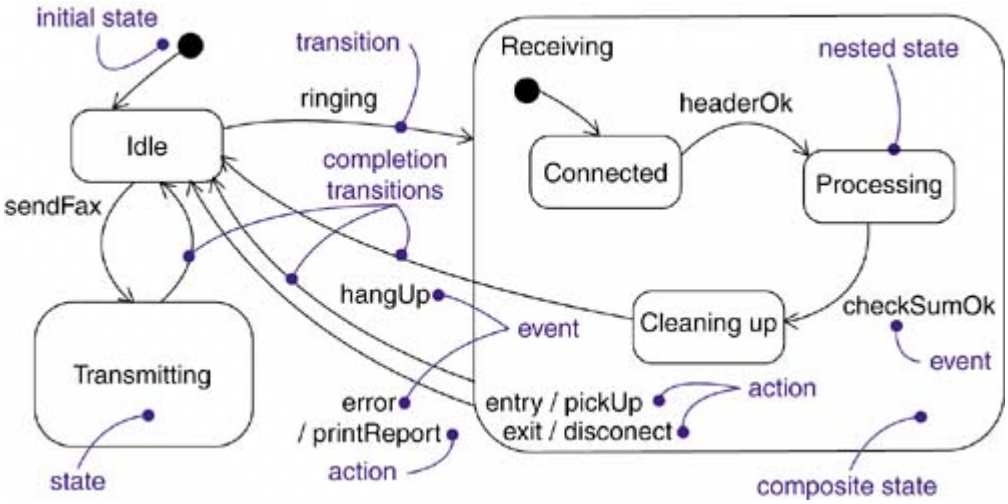
> *Gantt charts and Pert charts are discussed in Chapter 20.*

In modeling software-intensive systems as well, you'll find that the most natural way to visualize, specify, construct, and document the behavior of certain kinds of objects is by focusing on the flow of control from state to state rather than from activity to activity. You would do the latter with a flowchart (and in the UML, with an activity diagram). Imagine, for a moment, modeling the behavior of an embedded home security system. Such a system runs continuously, reacting to events from the outside, such as the breaking of a window. In addition, the order of events changes the way the system behaves. For example, the detection of a broken window will only trigger an alarm if the system is first armed. The behavior of such a system is best specified by modeling its stable states (for example, `Idle`, `Armed`, `Active`, `Checking`, and so on), the events that trigger a change from state to state, and the actions that occur on each state change.

Activity diagrams as flowcharts are discussed in Chapter 20; state machines are discussed in Chapter 22.

In the UML, you model the event-ordered behavior of an object by using state diagrams. As Figure 25-1 shows, a state diagram is simply a presentation of a state machine, emphasizing the flow of control from state to state.

Figure 25-1. State Diagram



PREV NEXT

Terms and Concepts

A *state diagram* shows a state machine, emphasizing the flow of control from state to state. A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events. A *state* is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. A *transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. An *activity* specifies an ongoing execution within a state machine. An *action* specifies a primitive executable computation that results in a change in state of the model or the return of a value. Graphically, a state diagram is a collection of nodes and arcs.

> **Note**
>
> The state diagrams used in UML are based on the statechart notation invented by David Harel. In particular, the concepts of nested states and orthogonal states were developed by Harel into a precise, formal system. The concepts in UML are somewhat less formal than Harel's notation and differ in some details, in particular, by being focused on object-oriented systems.

Common Properties

A state diagram is just a special kind of diagram and shares the same common properties as do all other diagramsthat is, a name and graphical contents that are a projection into a model. What distinguishes a state diagram from all other kinds of diagrams is its content.

The general properties of diagrams are discussed in Chapter 7.

Contents

Simple states, composite states, transitions, events, and actions are discussed in Chapter 22; activity diagrams are discussed in Chapter 20; notes and constraints are discussed in Chapter 6.

State diagrams commonly contain

- Simple states and composite states
- Transitions, events, and actions

Like all other diagrams, state diagrams may contain notes and constraints.

Note

A state diagram is basically a projection of the elements found in a state machine. This means that state diagrams may contain branches, forks, joins, action states, activity states, objects, initial states, final states, history states, and so on. Indeed, a state diagram may contain any and all features of a state machine.

Common Uses

The five views of an architecture are discussed in Chapter 2; instances are discussed in Chapter 13; classes are discussed in Chapters 4 and 9.

You use state diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the event-ordered behavior of any kind of object in any view of a system's architecture, including classes (which includes active classes), interfaces, components, and nodes.

When you use a state diagram to model some dynamic aspect of a system, you do so in the context of virtually any modeling element. Typically, however, you'll use state diagrams in the context of the system as a whole, a subsystem, or a class. You can also attach state diagrams to use cases (to model a scenario).

When you model the dynamic aspects of a system, a class, or a use case, you'll typically use state diagrams to model reactive objects.

Active classes are discussed in Chapter 23; interfaces are discussed in Chapter 11; components are discussed in Chapter 15; nodes are discussed in Chapter 27; use cases are discussed in Chapter 17; systems are discussed in Chapter 32.

A reactiveor event-drivenobject is one whose behavior is best characterized by its response to events dispatched from outside its context. A reactive object is typically idle until it receives an event. When it receives an event, its response usually depends on previous events. After the object responds to an event, it becomes idle again, waiting for the next event. For these kinds of objects, you'll focus on the stable states of that object, the events that trigger a transition from state to state, and the actions that occur on each state change.

Note

In contrast, you'll use activity diagrams to model a workflow or to model an operation. Activity diagrams are better suited to modeling the flow of activities over time, such as you would represent in a flowchart.

PREV NEXT

Common Modeling Techniques

Modeling Reactive Objects

The most common purpose for which you'll use state diagrams is to model the behavior of reactive objects, especially instances of classes, use cases, and the system as a whole. Whereas interactions model the behavior of a society of objects working together, a state diagram models the behavior of a single object over its lifetime. Whereas an activity diagram models the flow of control from activity to activity, a state diagram models the flow of control from event to event.

Interactions are discussed in Chapter 16; activity diagrams are discussed in Chapter 20.

When you model the behavior of a reactive object, you essentially specify three things: the stable states in which that object may live, the events that trigger a transition from state to state, and the actions that occur on each state change. Modeling the behavior of a reactive object also involves modeling the lifetime of an object, starting at the time of the object's creation and continuing until its destruction, highlighting the stable states in which the object may be found.

Modeling the lifetime of an object is discussed in Chapter 22.

A stable state represents a condition in which an object may exist for some identifiable period of time. When an event occurs, the object may transition from state to state. These events may also trigger self- and internal transitions, in which the source and the target of the transition are the same state. In reaction to an event or a state change, the object may respond by dispatching an action.

Time and space are discussed in Chapter 24.

Note

When you model the behavior of a reactive object, you can specify its action by tying it to a transition or to a state change. In technical terms, a state machine whose actions are all attached to transitions is called a Mealy machine; a state machine whose actions are all attached to states is called a Moore machine. Mathematically, the two styles have equivalent power. In practice, you'll typically develop state diagrams that use a combination of Mealy and Moore machines.

Pre- and postconditions are discussed in Chapter 10; interfaces are discussed in Chapter 11.

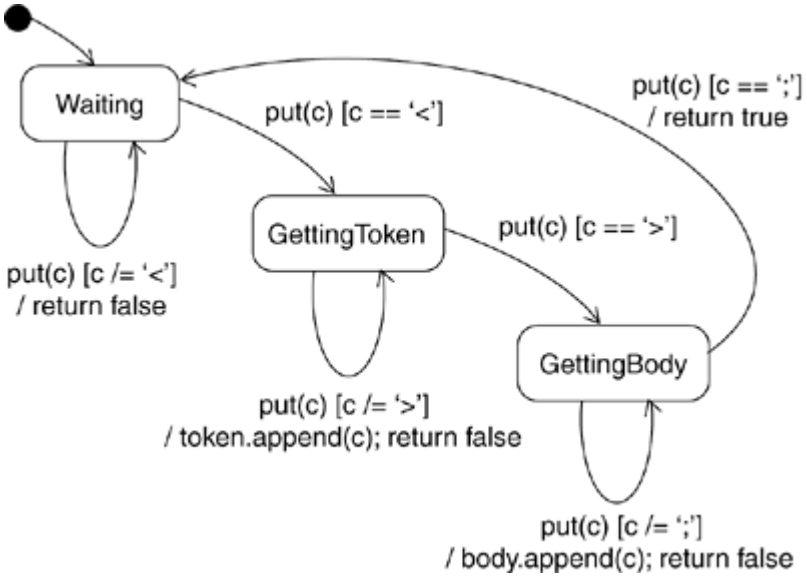
To model a reactive object,

- Choose the context for the state machine, whether it is a class, a use case, or the system as a whole.
- Choose the initial and final states for the object. To guide the rest of your model, possibly state the pre-and postconditions of the initial and final states, respectively.
- Decide on the stable states of the object by considering the conditions in which the object may exist for some identifiable period of time. Start with the high-level states of the object and only then consider its possible substates.
- Decide on the meaningful partial ordering of stable states over the lifetime of the object.
- Decide on the events that may trigger a transition from state to state. Model these events as triggers to transitions that move from one legal ordering of states to another.
- Attach actions to these transitions (as in a Mealy machine) and/or to these states (as in a Moore machine).
- Consider ways to simplify your machine by using substates, branches, forks, joins, and history states.
- Check that all states are reachable under some combination of events.
- Check that no state is a dead end from which no combination of events will transition the object out of that state.
- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses.

For example, Figure 25-2 shows the state diagram for parsing a simple context-free language, such as you might find in systems that stream in or stream out messages to XML. In this case, the machine is designed to parse a stream of characters that match the syntax

```
message : '<' string '>' string ';'
```

Figure 25-2. Modeling Reactive Objects



The first string represents a tag; the second string represents the body of the message. Given a stream of characters, only well-formed messages that follow this syntax may be accepted.

As the figure shows, there are only three stable states for this state machine: `Waiting`, `GettingToken`, and `GettingBody`. This state is designed as a Mealy machine, with actions tied to transitions. In fact, there is only one event of interest in this state machine, the invocation of `put` with the actual parameter `c` (a character). While `Waiting`, this machine throws away any character that does not designate the start of a token (as specified by the guard condition). When the start of a token is received, the state of the object changes to `GettingToken`. While in that state, the machine saves any character that does not designate the end of a token (as specified by the guard condition). When the end of a token is received, the state of the object changes to `GettingBody`. While in that state, the machine saves any character that does not designate the end of a message body (as specified by the guard condition). When the end of a message is received, the state of the object changes to `Waiting`, and a value is returned indicating that the message has been parsed (and the machine is ready to receive another message).

> *Events are discussed in Chapter 21.*

Note that this state specifies a machine that runs continuously; there is no final state.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for state diagrams, especially if the context of the diagram is a class. For example, using the previous state diagram, a forward engineering tool could generate the following Java code for the class `MessageParser`.

```
class MessageParser {
public
 boolean put(char c) {
  switch (state) {
   case Waiting:
    if (c == '<') {
```

```
      state = GettingToken;
      token = new StringBuffer();
      body = new StringBuffer();
     }
     break;
    case GettingToken :
     if (c == '>')
      state = GettingBody;
     else
      token.append(c);
     break;
    case GettingBody :
     if (c == ';')
      state = Waiting;
     else
      body.append(c);
     return true;
   }
   return false;
 }
 StringBuffer getToken() {
  return token;
 }
 StringBuffer getBody() {
  return body;
 }
private
 final static int Waiting = 0;
 final static int GettingToken = 1;
 final static int GettingBody = 2;
 int state = Waiting;
 StringBuffer token, body;
}
```

This requires a little cleverness. The forward engineering tool must generate the necessary private attributes and final static constants.

Reverse engineering (the creation of a model from code) is theoretically possible but practically not very useful. The choice of what constitutes a meaningful state is in the eye of the designer. Reverse engineering tools have no capacity for abstraction and therefore cannot automatically produce meaningful state diagrams. More interesting than the reverse engineering of a model from code is the animation of a model against the execution of a deployed system. For example, given the previous diagram, a tool could animate the states in the diagram as they were reached in the running system. Similarly, the firing of transitions could be animated, showing the receipt of events and the resulting dispatch of actions. Under the control of a debugger, you could control the speed of execution, setting breakpoints to stop the action at interesting states to examine the attribute values of individual objects.

Hints and Tips

When you create state diagrams in the UML, remember that every state diagram is just a projection on the same model of a system's dynamic aspects. A single state diagram can capture the semantics of a single reactive object, but no one state diagram can capture the semantics of an entire nontrivial system.

A well-structured state diagram

- Is focused on communicating one aspect of a system's dynamics.
- Contains only those elements essential to understanding that aspect.
- Provides detail consistent with its level of abstraction (expose only those features that are essential to understanding).
- Uses a balance between the styles of Mealy and Moore machines.

When you draw a state diagram,

- Give it a name that communicates its purpose.
- Start with modeling the stable states of the object, then follow with modeling the legal transitions from state to state. Address branching, concurrency, and object flow as secondary considerations, possibly in separate diagrams.
- Lay out its elements to minimize lines that cross.
- For large state diagrams, consider advanced features such as submachines that are included in the full UML specification.