UNIT – 5 : Advanced Behavioral Modeling

Syllabus :Events and signals,statemachines,processes and Threads ,time and space chart diagrams, Component, Deployment, Component Diagrams and Deployment diagrams

Events and Signals

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 stimulus 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.

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

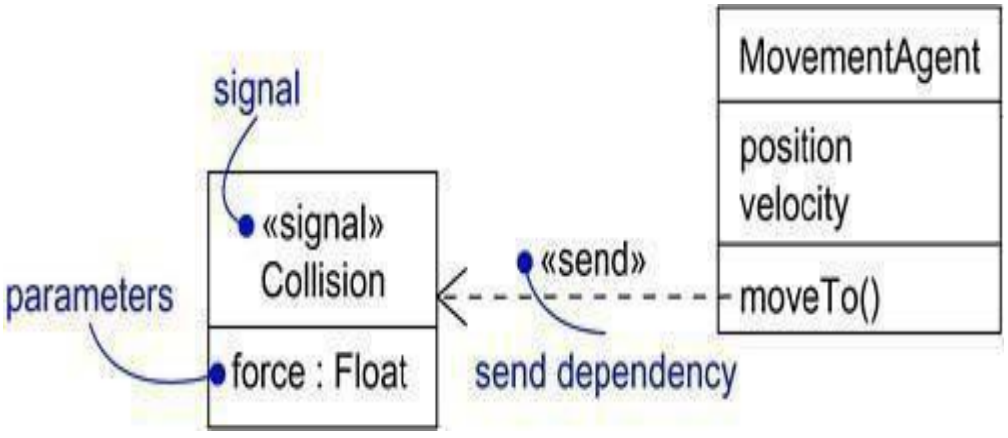
Signals

A signal represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another. Exceptions are supported by most contemporary programming languages and are the most common kind of internal signal that you will need to model.

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.

A signal may be sent as the action of a state transition in a state machine or the sending of a message in an interaction. The execution of an operation 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. In the UML, you model the relationship between an operation and the events that it can send by using a dependency relationship, stereotyped as **send.**

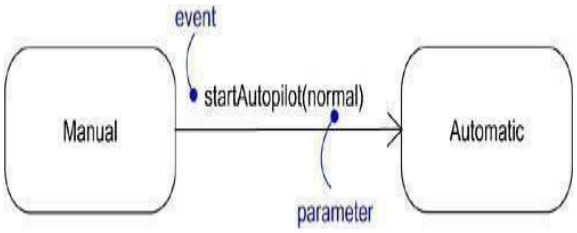
Figure Signals



Call Events

Just as a signal event represents the occurrence of a signal, a call event represents the dispatch of an operation. In both cases, the event may trigger a state transition in a state machine.Whereas a signal is an asynchronous event, a call event is, in general, 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.

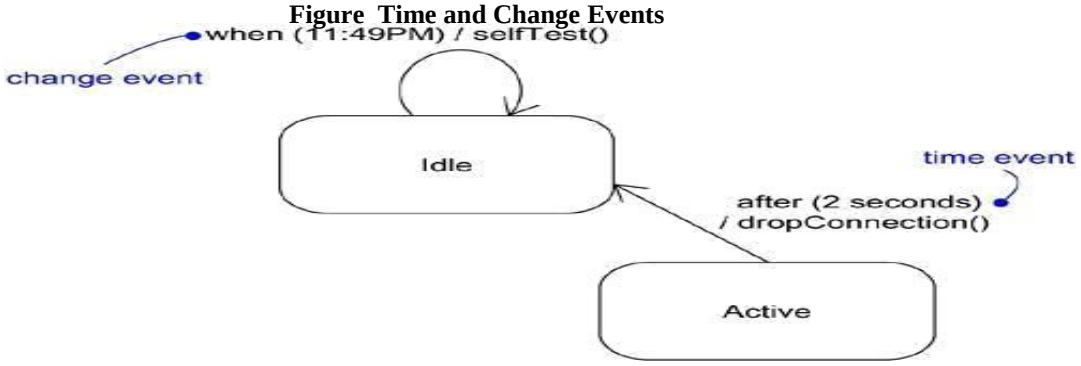
Figure Call Events



Time and Change Events

A time event is an event that represents the passage of time. As Figure 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.

Figure Time and Change Events



A change event is an event that represents a change in state or the satisfaction of some condition. As Figure shows, in the UML you model a change event by using the keyword**when** followed by some Boolean expression. You can use such expressions to mark an absolutetime (such as **when time = 11:59**) or for the continuous test of an expression (for example, **when altitude < 1000**).

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

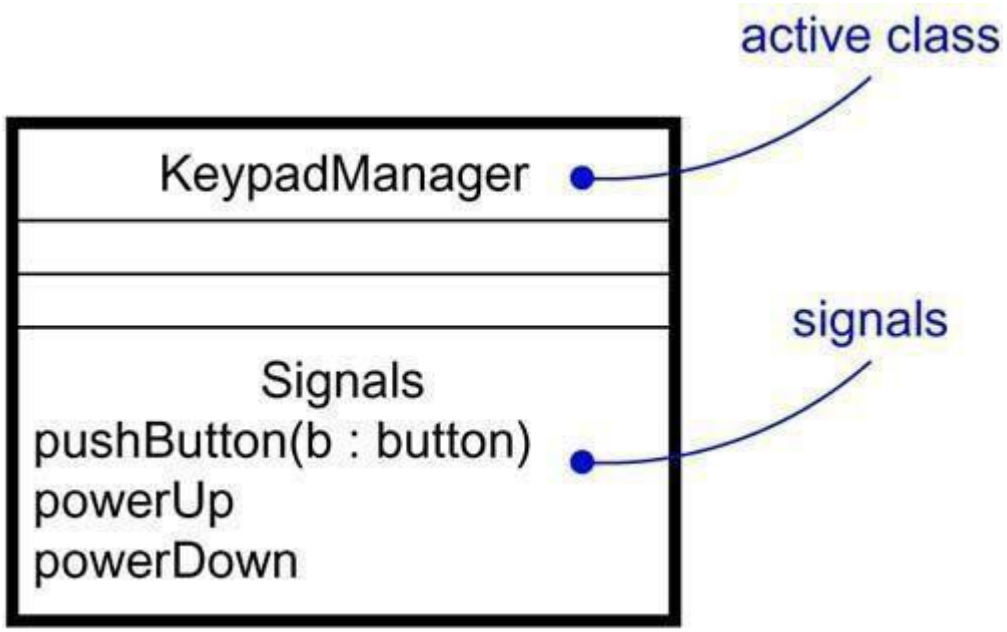
objects and passive objects.

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. 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.

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 is put in lock step with the flow of control of the receiver until the activity of the operation is carried out. 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.

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 .

Figure Signals and Active Classes.



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 thesetwo 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.**

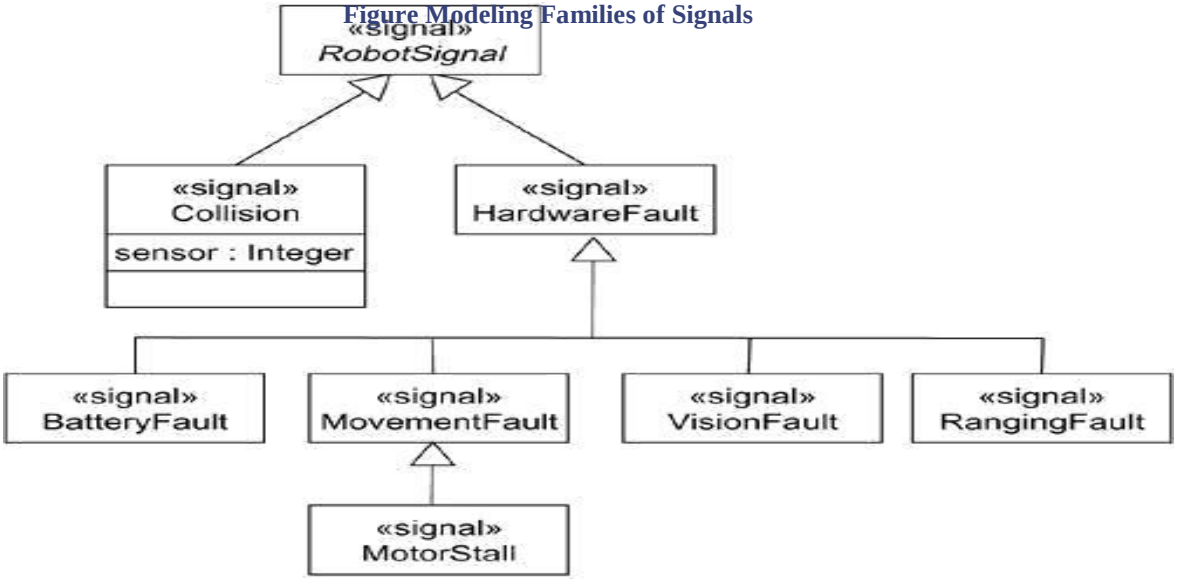
By modeling hierarchies of signals in this manner, you can specify polymorphic events. For example, consider a state machine with a transition triggered only by the receipt of a **MotorStall.** As a leaf signal in this hierarchy, the transition can be triggered only by thatsignal, so it is not polymorphic. 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.**

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.

Figure models a family of signals that may be handled by an autonomous robot. Note thatthe 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 Modeling Families of Signals



Modeling Exceptions

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

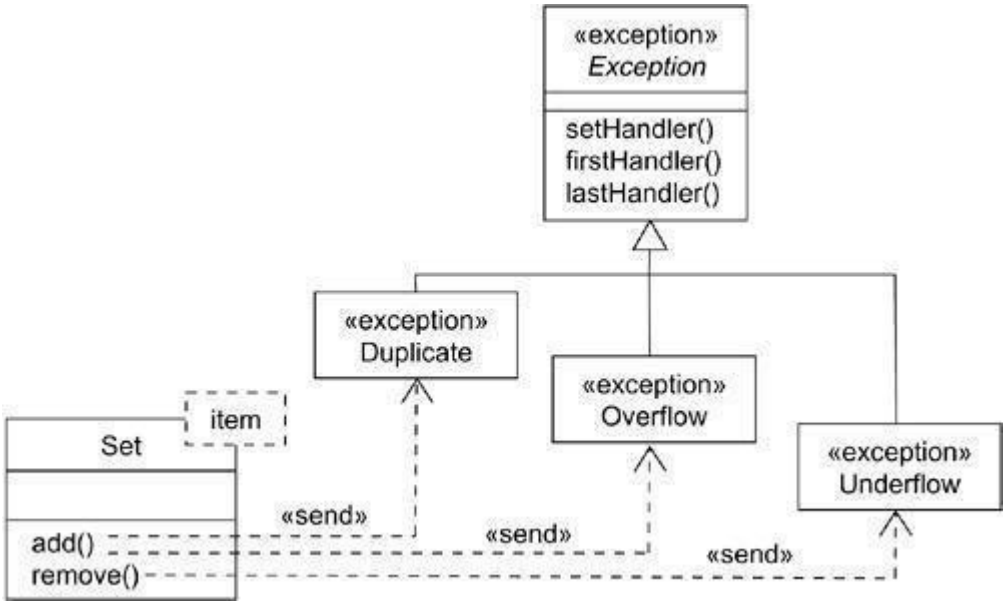
In the UML, exceptions are kinds of signals, which you model as stereotyped classes. Exceptions may be attached to specification operations. Modeling exceptions is somewhat the inverse of modeling a general family of signals. You model a family of signals primarily to specify the kinds of signals an active object may receive; you model exceptions primarily to specify the kinds of exceptions that an object may throw through its operations.

To model exceptions,

- For each class and interface, and for each operation of such elements, consider the exceptional conditions that may be raised.
- Arrange these exceptions in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions, as necessary.
- For each operation, specify the exceptions that it may raise. You can do so explicitly (by showing **send** dependencies from an operation to its exceptions) or you can put this in the operation's specification.

Figure models a hierarchy of exceptions that may be raised by a standard library ofcontainer classes, such as the template class **Set.** This hierarchy is headed by the abstract signal **Exception** and includes three specialized exceptions: **Duplicate, Overflow,** and **Underflow.** As shown, the **add** operation raises **Duplicate** and **Overflow** exceptions, andthe **remove** operation raises only the **Underflow** exception. Alternatively, you could have put these dependencies in the background by naming them in each operation's specification. Either way, by knowing which exceptions each operation may send, you can create clients that use the **Set** class correctly.

Figure Modeling Exceptions



State Machines

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 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* is ongoing nonatomic execution within a state machine. An *action* is an executable atomic 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.

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.

In other kinds of systems, you'll encounter objects that must respond to signals, which are asynchronous stimuli 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).

The behavior of objects that must respond to asynchronous stimulus or whose current behavior depends on their 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 state machine will simply ignore that signal. 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.

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).

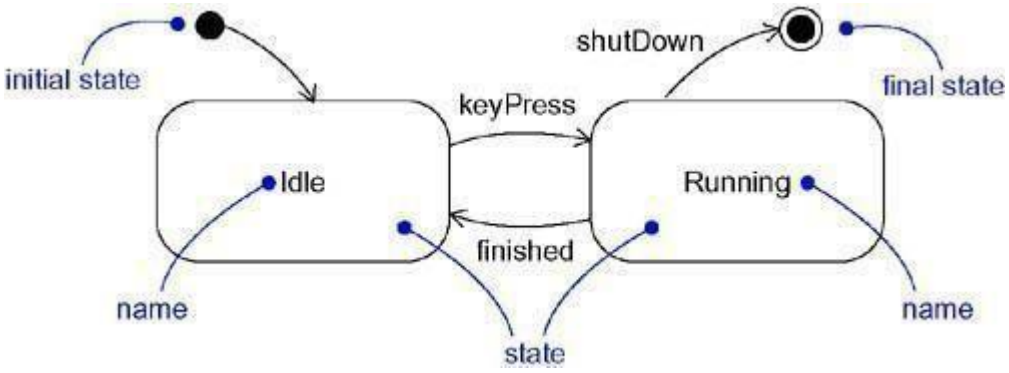
When an object's state machine is in a given state, the object is said to be in that state.
For example, an instance of **Heater** might be **Idle** or perhaps **ShuttingDown.**

Astate 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 actions	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 disjoint (sequentially active) or concurrent (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

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

Figure 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.

> **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 a final state may have the full complement of features, including a guard condition and action (but not a trigger event).

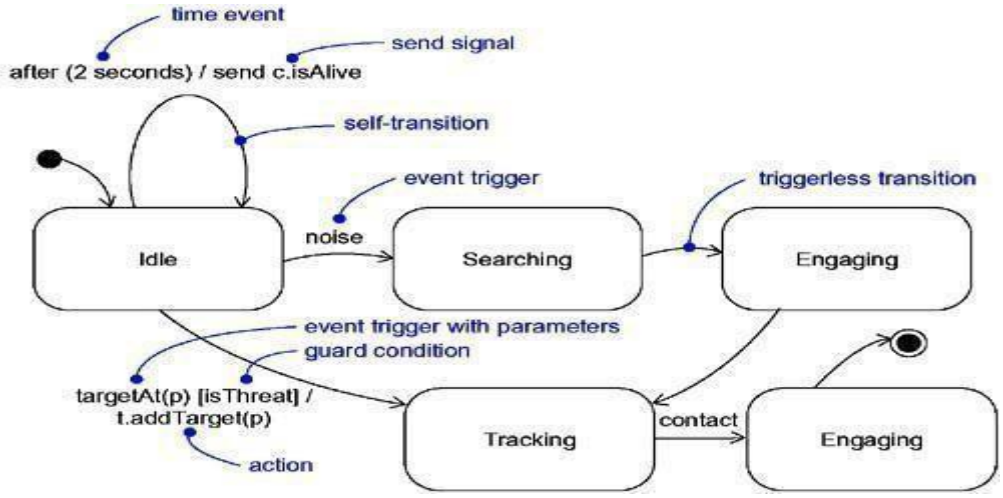
Transitions

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.

As Figure 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 Transitions



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.

It is also possible to have a triggerless transition, represented by a transition with no event trigger. A **triggerless transition• also called a completion transition• is triggered implicitly when its source state has** completed its activity.

Guard

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).

Action

An action is an executable atomic computation. Actions may include 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. As the previous figure shows, there's a special notation for sending **a signal• the signal name is prefixed with the keyword send** as a visual cue.

Activities are discussed in a later section of this chapter; dependencies are discussed in An action is atomic, meaning that it cannot be interrupted by an event and therefore runs to completion. This is in contrast to an activity, which may be interrupted by other events.

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).

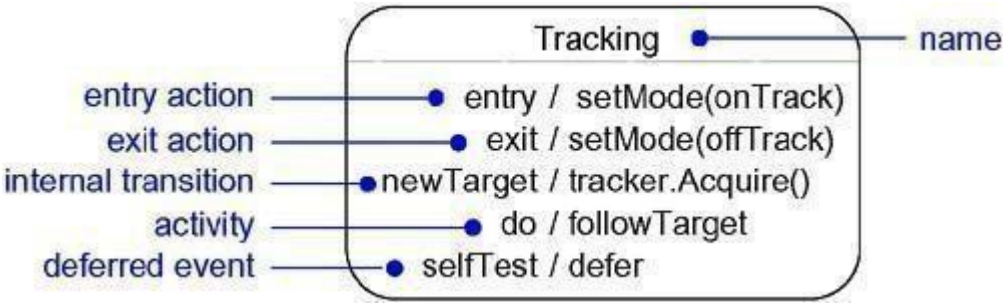
However, the UML's state machines have a number of advanced features that help you to manage complex behavioral models. These features often reduce the number of states and transitions you'll need, and they codify a number of common and somewhat complex idioms you'd otherwise encounter using flat state machines. Some of these advanced features include entry and exit actions, internal transitions, activities, and deferred events.

Entry and Exit Actions

In a number of modeling situations, you'll want to dispatch the same action whenever you enter a state, no matter which transition led you there. Similarly, when you leave a state, you'll want to dispatch the same 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~~ shows, the UML provides a shorthand for this idiom. In the symbol for the state, you can include an entry action (marked by the keyword event **entry**) and an exit action (marked by the keyword event **exit**), together with an appropriate action. Whenever you enter the state, its entry action is dispatched; whenever you leave the state, its exit action is dispatched.

Figure Advanced States and Transitions



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 , an event triggers the transition, you leave the state,an action (if any) is dispatched, and then you reenter the same state. Because this transition exits and then enters the state, a self-transition dispatches the state's exit action, then it dispatches the action of the self-transition, and finally, it

dispatches the state's entry action. However, suppose you want to handle the event but don't want to fire the state's entry and exit actions. Using flat state machines, you can achieve that effect, but you have to be diligent about remembering which of a state's transitions have these entry and exit actions and which do not.

As Figure shows, the UML provides a shorthand for this idiom, as well (for example, for the event **newTarget**). In the symbol for the state, you can include an internal transition (marked by an event). Whenever you are in the state and that event is triggered, the corresponding action is dispatched without leaving and then reentering the state. Therefore, the event is handled without dispatching the state's exit and then entry actions.

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 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. The activity of a **do** transition might name another state machine (such as **followTarget**). You **can also specify a sequence of actions•for** example, do / op1(a); op2(b);op3(c). Actions are never interruptible, but sequences of actions are. In between each action (separated by the semicolon), events may be handled by the enclosing state, which results in transitioning out of the state.

Deferred Events

Consider a state such as **Tracking.** As illustrated in Figure, 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 willbe lost. That means that the event may occur, but it will be postponed and no action will result because of the presence of that event. 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 recognize 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 maintenanceagent in the system.

In the UML, you can specify this behavior by using deferred events. A deferred event is a list of events whose occurrence in the state is postponed until a state in which the listed events are not deferred becomes active, at which time they occur and may trigger transitions as if they had just occurred. As you can see in the previous figure, 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 itappears as if they just occurred.

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 machines• substates• that 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 substates• that is, nested states• is called a composite state. A composite state may contain either concurrent (orthogonal) or sequential (disjoint) substates. In the UML, you render a composite state just as you do a simple state, but with an option al graphic compartment that shows a nested state machine. Substates may be nested to any level.

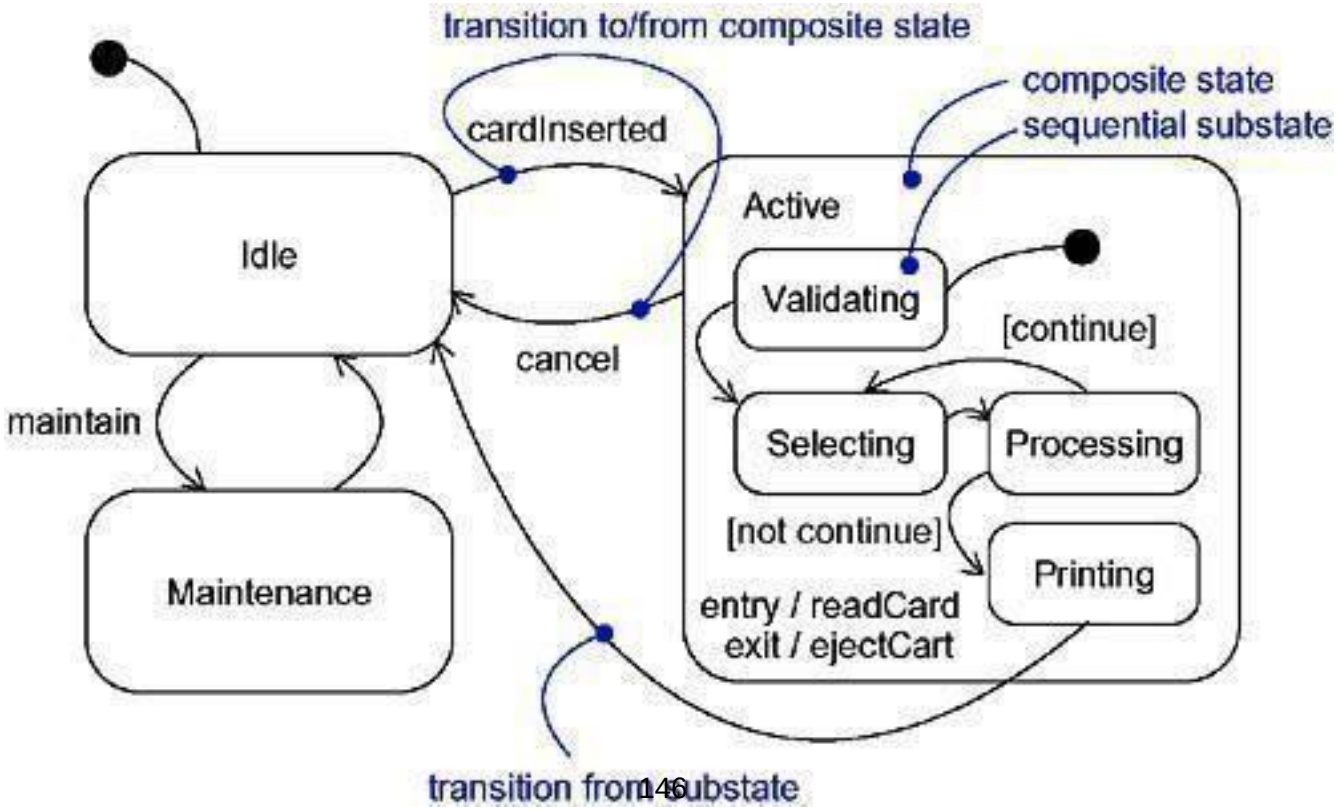
Sequential Substates

Consider the problem of modeling the behavior of an ATM. There are three basic states in which this system might be: **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 andprocess 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 suchinterrupting 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 sequential substates, there's a simpler way to model this problem, as Figure 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 triggerless transition back to the **Idle** state. Notice that the **Active** state has an exit action, which ejects the customer's credit card.

Figure SequentialSubstates



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 returnsthe 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 sequential, or disjoint, substates. Given a set of disjoint 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, sequential 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 dispatching its entry action (if any). If its target is the nested state, control passes to the nested state, after dispatching 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 dispatched), then it leaves the composite state (and its exit action, if any, is dispatched). A transition whose source is the composite state essentially cuts short (interrupts) the activity of the nested state machine.

History States

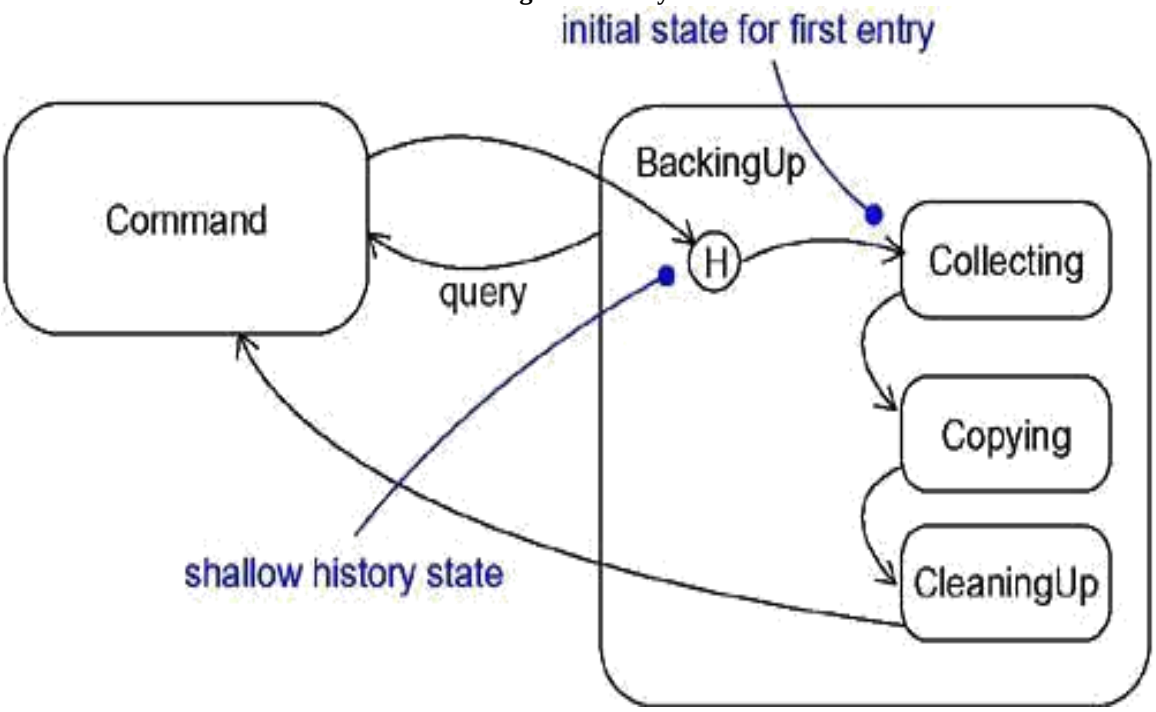
A state machine describes the dynamic aspects of an object whose current behavior depends on its past. A state machine in effect specifies the legal ordering of states an object may go through during its lifetime.

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 sequential substates to remember the last substate that was active in it prior to the transition from the composite state. As Figure shows, you represent ashallow history state as a small circle containing the symbol **H.**

Figure 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 statemachine 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 triggerless 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 state• thus bypassing the Collecting state• because Copying** was the last substate active prior to the transition from **BackingUp.**

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.

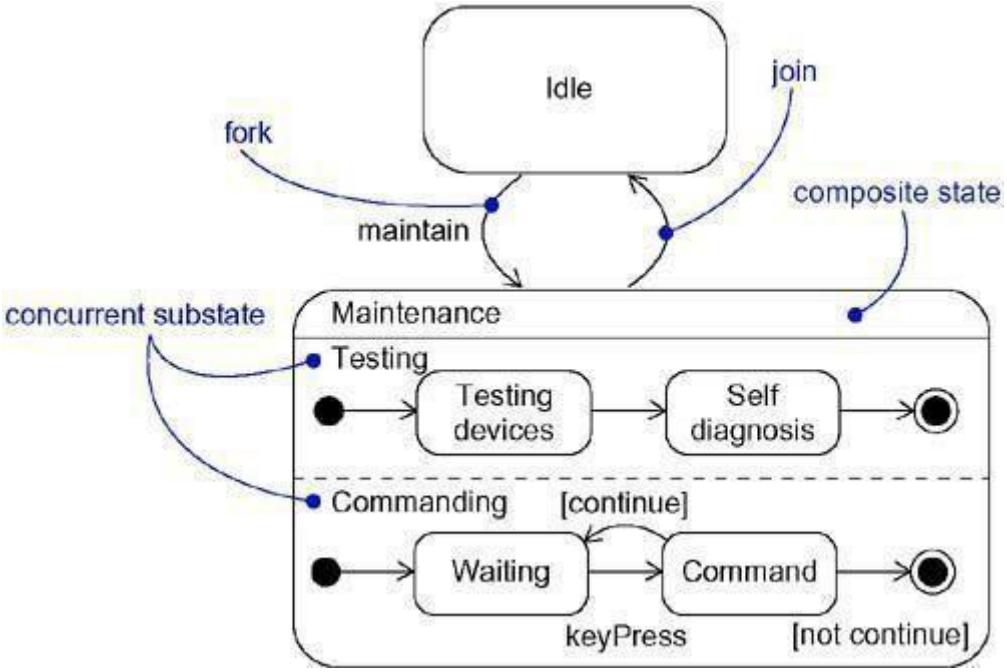
Concurrent Substates

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

For example, Figure shows an expansion of the **Maintenance** state from Figure .**Maintenance** is decomposed into two concurrent substates, **Testing** and **Commanding,** shown by nesting them in the **Maintenance** state but separating them from one another with a dashed line. Each of these concurrent

substates is further decomposed into sequential substates. When control passes from the **Idle** to the **Maintenance state, control then forks to two concurrent flows• the enclosing object will be in the Testing** state and the **Commanding** state. Furthermore, while in the **Commanding** state, the enclosing object will be in the **Waiting** or the **Command** state.

Figure Concurrent Substates



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

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

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.

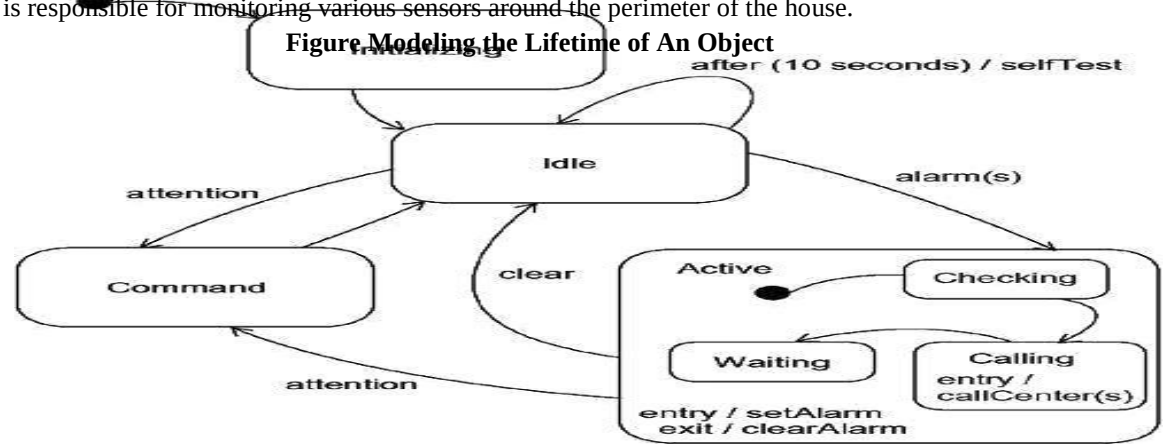
To model the lifetime of an object,

- Set the context for the state machine, whether it is a class, a use case, or the system as a whole.
    1. If the context is a class or a use case, collect 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.

    2. 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 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 Modeling the Lifetime of An Object



In the lifetime of this controller class, there are four main states: **Initializing**(the controller isstarting 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 isprocessing 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 statesare not shown, other than the self-transition with the time event in the **Idle** state. This kind of time event is common 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 dispatched 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 issuea command.

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

Processes and thread

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 thatcan 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 thick lines. Processes and threads are rendered as stereotyped active classes (and also appear as sequences in interaction diagrams).

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.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.

In a concurrent system, there is more than one flow of control• that 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.

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.

You use active classes to model common families of processes or threads. In technical terms, this means **that an active object• an instance of an active class• reifies (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.

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.

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.

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

Standard Elements

All of the UML's extensibility mechanisms apply to active classes. Most often, you'll use tagged values to extend active class properties, such as specifying the scheduling policy of the active class.

The UML defines two standard stereotypes that apply to active classes.

1. Specifies a h	eavyweight flow that can execute concurrently with other processes **process**
2. **thread**	Specifies a lightweight flow that can execute concurrently with other threads withinthe same process

The distinction between a process and a thread arises from the two different ways a flow of control may be managed by the operating system of the node on which the object resides.

A process is heavyweight, which means that it is a thing known to the operating system itself and runs in an independent address space. Under most operating systems, such as Windows and Unix, each program runs as a process in its own address space. In general, all processes on a node are peers of one another, contending for all the same resources accessible on the node. Processes are never nested inside one another. If the node has multiple processors, then true concurrency on that node is possible. If the node has only one processor, there is only the illusion of true concurrency, carried out by the underlying operating system.

A thread is lightweight. It may be known to the operating system itself. More often, it is hidden inside a heavier-weight process and runs inside the address space of the enclosing process. In Java, for example, a thread is a child of the class **Thread.** All the threads that live in the context of a process are peers of one another, contending for the same resources accessible inside the process. Threads are never nested inside one another. In general, there is only the illusion of true concurrency among threads because it is processes, not threads, that are scheduled by a node's operating system.

Communication

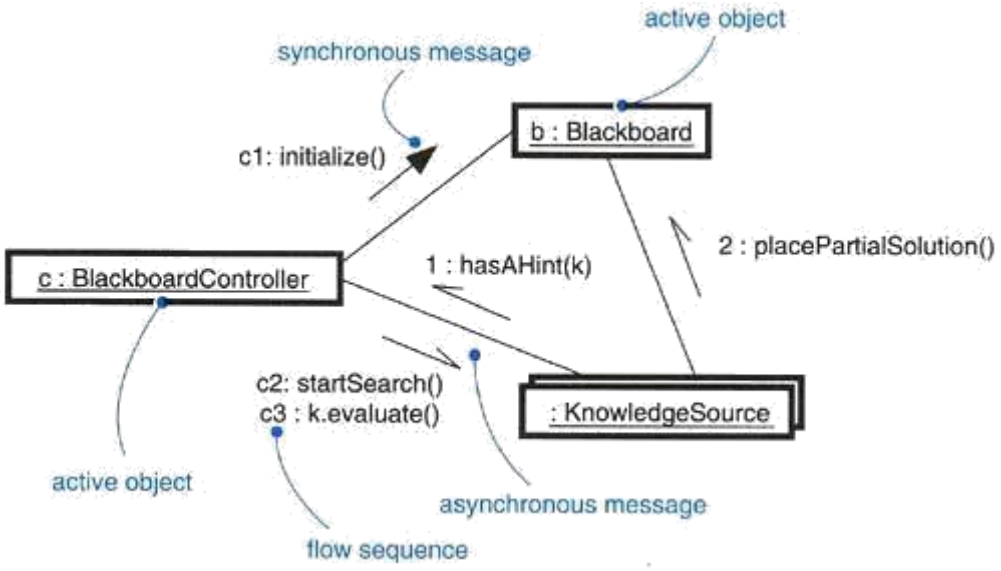
When objects collaborate with one another, they interact by passing messages from one to the other. In a system with both active and passive objects, there are four possible combinations of interaction that you must consider.

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 return object (if any) is passed back to the caller; and then the two continue on their independent paths. For the duration of the call, the two flows of controls 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.

In the UML, you render a synchronous message as a full arrow and an asynchronous message as a half arrow, as in Figure .

Figure Communication



Third, a message may be passed from an active object to a passive object. A difficulty arises if more than one active object at a time passes their flow of control through one passive object. 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.

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, anything more than one flow will interfere with another, corrupting the state of the object. 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.

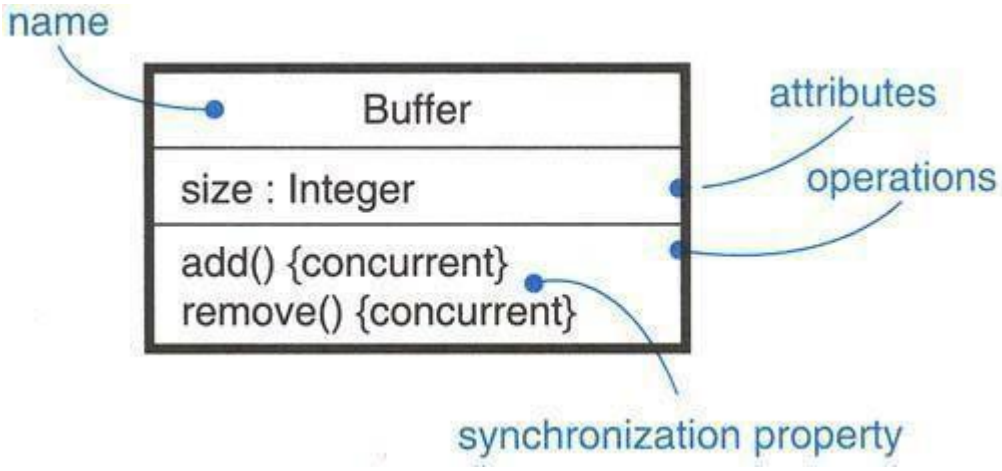
The key to solving this problem in object-oriented systems is by treating an object as a critical region. There are three alternatives to this approach, each of which involves attaching certain synchronization properties to the operations defined in a class. In the UML, you can model all three approaches.

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 is 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 be invoked on the object, reducing this to sequential semantics.
3. Concurrent	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic.

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

As Figure shows, you can attach these properties to an operation, which you can render in the UML by using constraint notation.

Figure Synchronization



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.

Process Views

Active objects play an important role in visualizing, specifying, constructing, and documenting a system's process view. The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability, and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as **for the design view• that is, class diagrams, interaction diagrams, activity diagrams, and statechart diagrams, but** with a focus on the active classes that represent these threads and processes.

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.

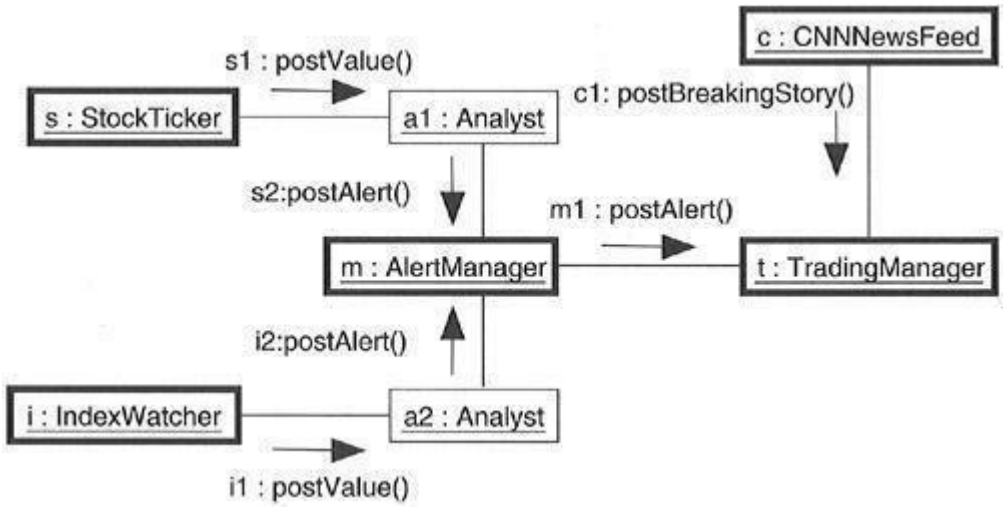
To model multiple flows of control,

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

- 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~~ 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 thismodel 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 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.

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.

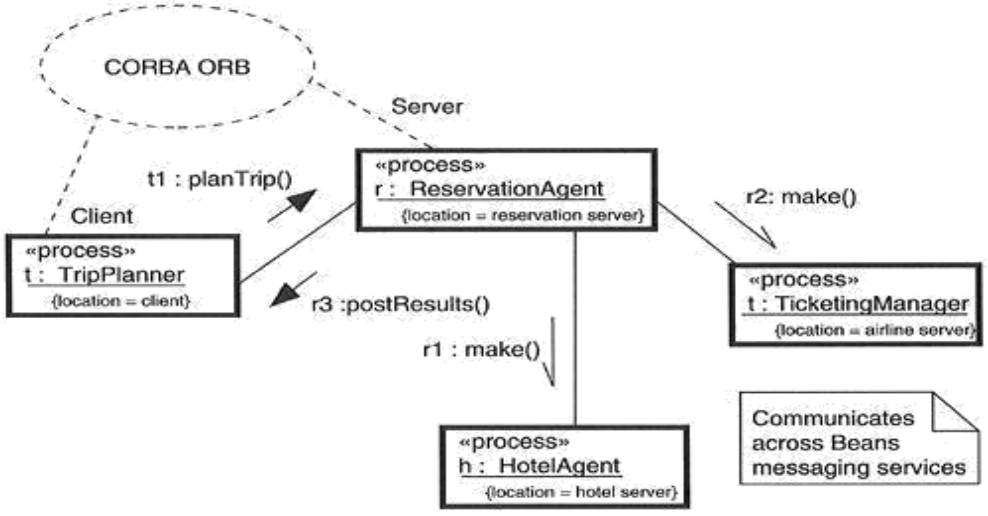
The problem of interprocess communication is compounded by the fact that, in distributed systems, processes may live on separate nodes. Classically, there are two approaches tointerprocess communication: message passing and remote procedure calls. In the UML, you still model these as asynchronous or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

To model interprocess communication,

- Model the multiple flows of control.
- Consider which of these active objects represent processes and which represent threads. Distinguish them using the appropriate stereotype.
- 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 shows a distributed reservation system with processes spread across four nodes.Each object is marked using the **process** stereotype. Each object is also marked with a **location** tagged value, specifying its physical location. Communication among the**ReservationAgent, TicketingManager,** and**HotelAgent**is asynchronous. Modeled witha note, communication is described 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
TripPlanneracts as a **client,** and the**ReservationAgent**acts as a **server.** By zoominginto the collaboration, you'll find the details of how this server and client collaborate.

Figure Modeling Interprocess Communication



Time and Space

Terms and Concepts

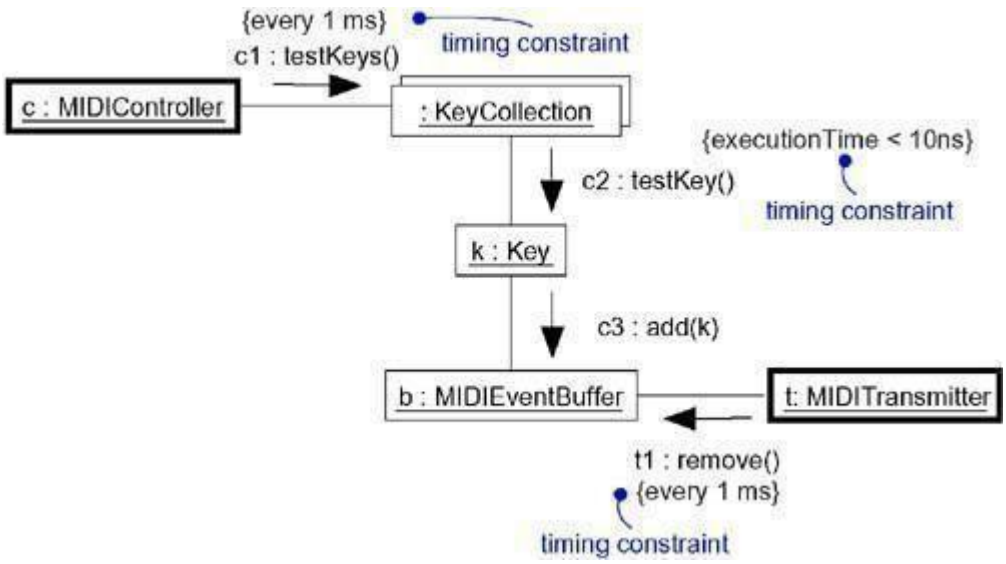
A *timing mark* is a denotation for the time at which an event occurs. Graphically, a timing mark is formed as an expression from the name given to the message (which is typically different from the name of the action dispatched by the message). A *time expression* is an expression that evaluates to an absolute or relative value of time. 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 bracke**ts and generally connected to an element by a dependency relationship. *Location* is the placement of a component on a node. Graphically, location is rendered as a tagged **value•** that is, a string enclosed by brackets and placed below an element's name, or as the nesting of components inside nodes.

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.

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 as a timing mark. 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. As a result, you can't use the event name to write an expression because the same event may trigger different messages. If the designated message is ambiguous, use the explicit name of the message in a timing mark to designate the message you want to mention in a time expression. A timing mark is nothing more than an expression formed from the name of a message in an interaction. Given a message name, you can refer to any of three functions **of that message• that is, startTime**,**stopTime**, and**executionTime.** You can then use these functions to specify arbitrarilycomplex 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 , you can place these time expressions in a timing constraint to specify the timingbehavior 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 Time

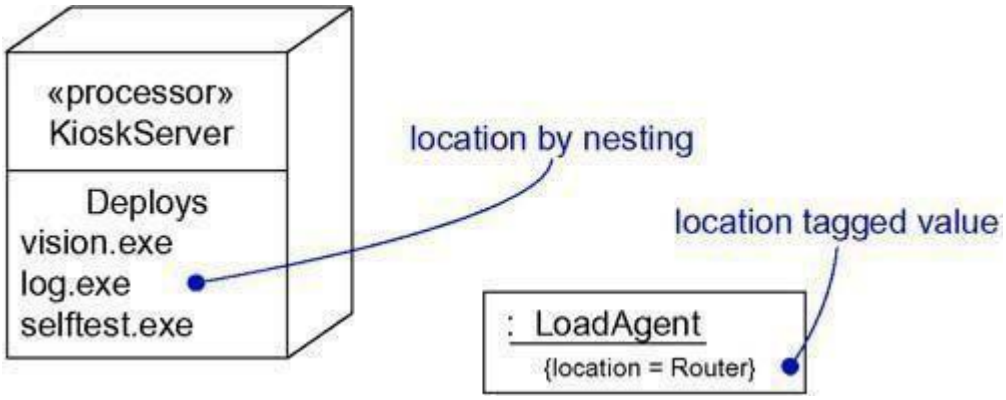


Location

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.

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. Components such as executables, libraries, and tables reside on these nodes. Each instance of a node will own instances of certain components, and each instance of a component will be owned by exactly one instance of a node (although instances of the same kind of component may be spread across different nodes). For example, as Figure shows, the executable component **vision.exe** may reside on the node named**KioskServe**r.

Figure Location



Instances of plain classes may reside on a node, as well. For example, as Figure shows, an instance of the class **LoadAgent** lives on the node named **Router.**

As the figure illustrates, you can model the location of an element in two ways in the UML. First, as shown for the **KioskServer**, you can physically nest the element (textually or graphically) in a extra compartment in its enclosing node. Second, as shown for the **LoadAgent**, you can use the defined tagged value **location** to designate the node on which the class instance resides.

You'll typically use the first form when it's important for you to give a visual cue in your diagrams about the spatial separation and grouping of components. Similarly, you'll use the second form when modeling the location of an element is important, but secondary, to the diagram at hand, such as when you want to visualize the passing of messages among instances.

Common Modeling Techniques

Modeling Timing Constraints

Modeling the absolute time of an event, modeling the relative time between events, and modeling the time it takes to carry out an action are the three primary time-critical properties of real time systems for which you'll use timing constraints.

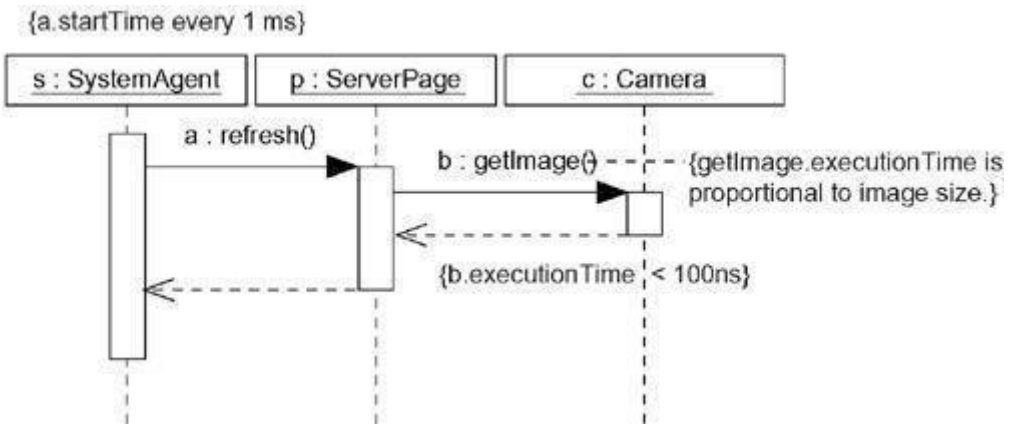
To model timing constraints,

- 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 each time critical operation in each class, consider its time complexity. Model those semantics as timing constraints on the operation.

For example, as shown in Figure the left-most constraint specifies the repeating start time the call event **refresh.** Similarly, the center timing constraint specifies the maximum duration for calls to **getImage.** Finally, the right-most constraint specifies the time complexity of the call event **getImage.**

Figure Modeling Timing Constraint



Note

Observe that **executionTime**may be applied to actions such as**getImage**, as wellas to timing marks such as **a** and**b**. Also, timing constraints such as these may be written as free- form text. If you want to specify your semantics more precisely, you can use the UML's Object Constraint Language (OCL), described further in *The UnifiedModeling Language Reference Manual*.

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 components and class instances. If your focus is the configuration management of the deployed system, modeling the distribution of components 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.

Deciding how to distribute the objects in a system is a wicked 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-engineering solutions aren't much better. In fact, they are probably worse because they usually end up being brittle.

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. Co-locate 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.

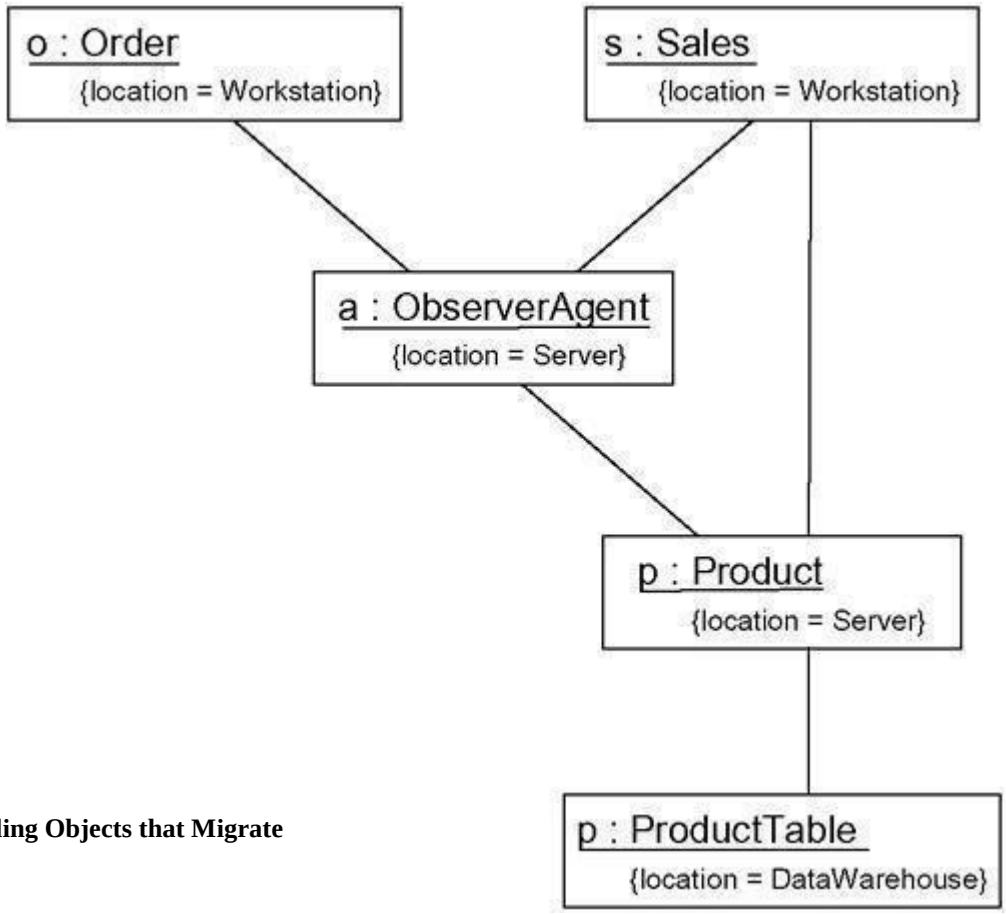
- Render this allocation in one of two ways:

    1. By nesting objects in the nodes of a deployment diagram

    2. By explicitly indicating the location of the object as a tagged value

Figure provides an object diagram that models the distribution of certain objects in a retailsystem. 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 Modeling the Distribution of Objects



Modeling Objects that Migrate

For many distributed systems, components and objects, once loaded on the system, stay put. For their lifetime, from creation to destruction, they never leave the node on which they were born. There are certain classes of distributed systems, however, for which things move about, usually for one of two reasons.

First, you'll find objects migrating in order to move closer to actors and other objects they need to work with to do their job better. For example, in a global shipping system, you'd see objects that represent ships, containers, and manifests moving from node to node to track their physical counterpart. If you have a ship in Hong Kong, it makes for better locality of reference to put the object representing the ship, its containers, and its manifest on a node in Hong Kong. When that ship sails to San Diego, you'd want to move the associated objects, as well.

Second, you'll find objects migrating in response to the failure of a node or connection or to balance the load across multiple nodes. For example, in an air traffic control system, the failure of one node cannot be allowed to stall a nation's entire operations. Rather, a failure-tolerant system such as this will migrate elements to other nodes. Performance and throughput may be reduced,

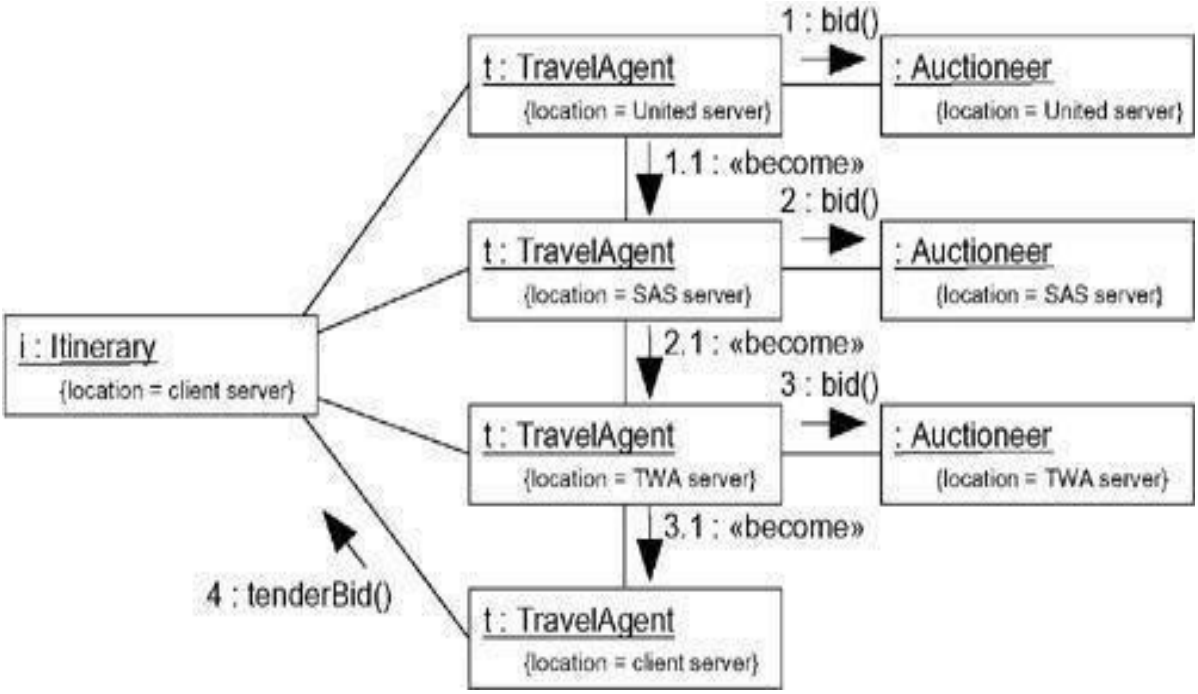
but safe functionality will be preserved. Similarly, and especially in Web-based systems that must deal with unpredictable peaks in demand, you'll often want to build mechanisms to automatically balance the processing load, perhaps by migrating components and objects to underused nodes.

Deciding how to migrate the objects in a system is an even more wicked problem than simple static distribution because migration raises difficult problems of synchronization and preservation of identity.

To model the migration of objects,

Figure provides a collaboration diagram that models the migration of a Web agent thatmoves from node to node, collecting information and bidding on resources in order to automatically deliver a lowest-cost travel ticket. Specifically, this diagram shows an instance (named **t**) of the class **TravelAgent** migrating from one server to another. Along the way, the object interacts with anonymous **Auctioneer** instances at each node, eventually delivering a bid for the **Itinerary** object, located on the **client server.**

Figure Modeling Objects that Migrate



Statechart Diagrams

Terms and Concepts

A *statechart 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* is ongoing nonatomic execution within a state machine. An *action* is an executable atomic computation that results in a change in state of the model or the return of a value. Graphically, a statechart diagram is a collection of vertices and arcs.

Common Properties

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

Contents

Statechart diagrams commonly contain

- Simple states and composite states
- Transitions, including events and actions distinguishes an activity diagram from a statechart diagram is that an activity diagram is basically a projection of the elements found in an activity graph, a special case of a state machine in which all or most states are activity states and in which all or most transitions are triggered by completion of activities in the source state.

Common Uses

You use statechart 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 statechart 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 statechart diagrams in the context of the system as a whole, a subsystem, or a class. You can also attach statechart 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 statechart diagrams in one way.

- To model reactive objects

A reactive — or event-driven — object 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.

Common Modeling Technique

Modeling Reactive Objects

The most common purpose for which you'll use statechart diagrams is to model the behavior of reactive objects, especially instances of classes, use cases, and the system as a whole. Whereasinteractions model the behavior of a society of objects working together, a statechart 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 statechart diagram models the flow of control from event to event.

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.

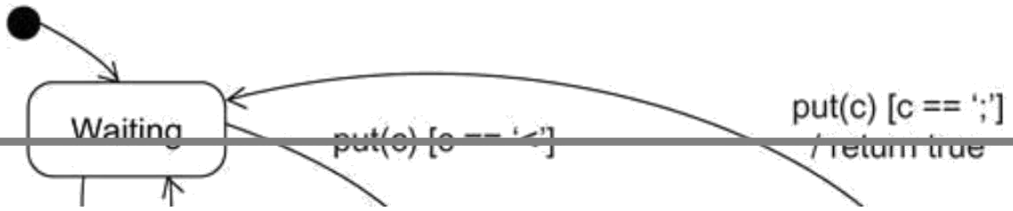
A stable state represents a condition in which an object may exist for some identifiable period of time. When an event occurs, the object may transition from state to state. These events may also trigger self- and internal transitions, in which the source and the target of the transition are the same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

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 shows the statechart diagram for parsing a simple context- freelanguage, 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

Figure 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 statechart is designed as a Mealy machine, withactions 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 designatethe 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).Note that this statechart 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 statechart diagrams,especially if the context of the diagram is a class. For example, using the previous statechart diagram, a forward engineering tool could generate the following Java code for the class
MessageParser.

```
classMessageParser { public
  boolean put(char c) { switch (state)
     { case Waiting:
       if (c == '<') {
         state = GettingToken;
         token = new StringBuffer(); body =
         new StringBuffer();
       }
       break;
```

```
            caseGettingToken : if (c == '>')
                state = GettingBody; else
                token.append(c);
              break;
            caseGettingBody : if (c == ';')
               state = Waiting; else
                    body.append(c); return true;
        }
        return false;
      }
      StringBuffergetToken() { return token;
      }
      StringBuffergetBody() { return body;
      }
  private
      final static int Waiting = 0; final static intGettingToken =
      1; final static intGettingBody = 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 practicallynot 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 statechart 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.

Components

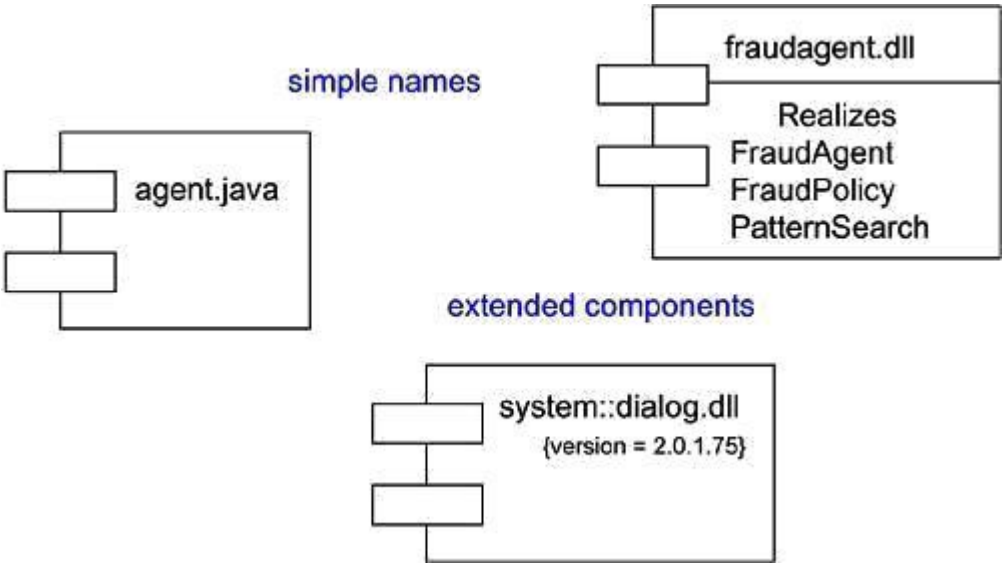
Terms and Concepts

A *component* is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Graphically, a component is rendered as a rectangle with tabs.

Names

Every component must have a name that distinguishes it from other components. A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the component name prefixed by the name of the package in which that component lives. A component is typically drawn showing only its name, as in Figure . Just as with classes, you may draw components adorned with tagged values or with additional compartments to expose their details, as you see in the figure.

Figure Simple and Extended Component



Components and Classes

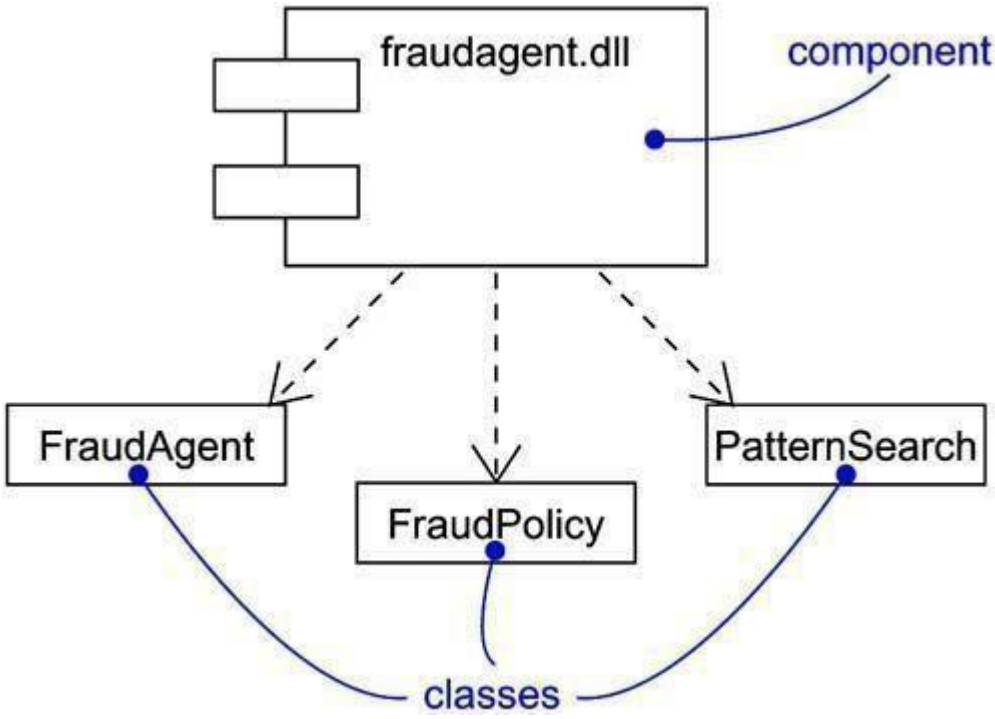
In many ways, components are like classes: Both have names; both may realize a set of interfaces; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions. However, there are some significant differences between components and classes.

- Classes represent logical abstractions; components represent physical things that live in the world of bits. In short, components may live on nodes, classes may not.
- Components represent the physical packaging of otherwise logical components and are at a different level of abstraction.
- Classes may have attributes and operations directly. In general, components only have operations that are reachable only through their interfaces.

The first difference is the most important. When modeling a system, deciding if you should use a class or a component involves **a simple decision• if the thing** you are modeling lives directly on a node, use a component; otherwise, use a class.

The second difference suggests a relationship between classes and components. In particular, a component is the physical implementation of a set of other logical elements, such as classes and collaborations. As Figure shows, the relationship between a component and the classes it implements can be shown explicitly by using a dependency relationship. Most of the time, you'll never need to visualize these relationships graphically. Rather, you will keep them as a part of the component's specification.

Figure Components and Classes



The third difference points out how interfaces bridge components and classes. As described in more detail in the next section, components and classes may both realize an interface, but a component's services are usually available only through its interfaces.

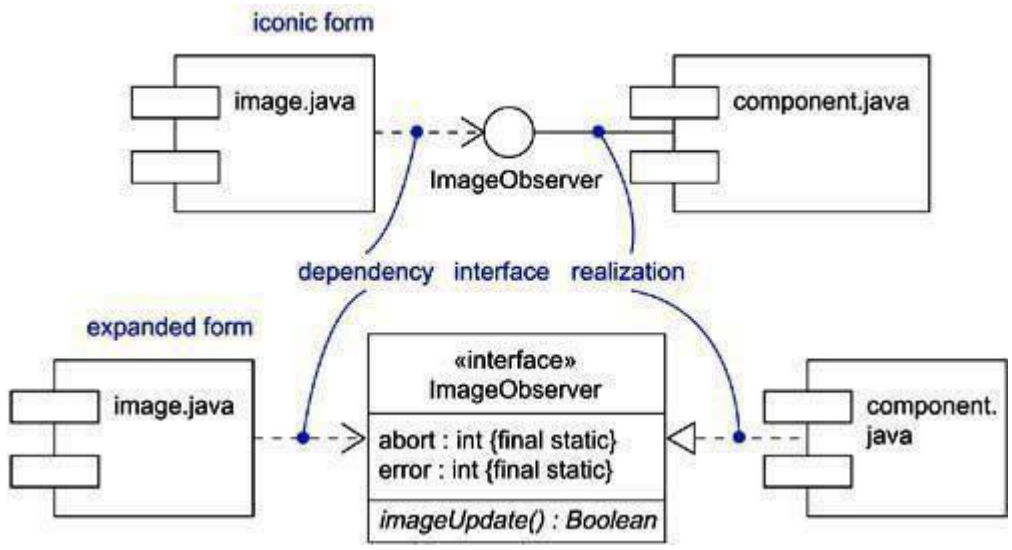
Components and Interfaces

An interface is a collection of operations that are used to specify a service of a class or a component. The relationship between component and interface is important. All the most common component-based operating system facilities (such as COM+, CORBA, and Enterprise Java Beans) use interfaces as the glue that binds components together.

Using one of these facilities, you decompose your physical implementation by specifying interfaces that represent the major seams in the system. You then provide components that realize the interfaces, along with other components that access the services through their interfaces. This mechanism permits you to deploy a system whose services are somewhat location-independent and, as discussed in the next section, replaceable.

As Figure indicates, you can show the relationship between a component and its interfaces in one of two ways. The first (and most common) style renders the interface in its elided, iconic form. The component that realizes the interface is connected to the interface using an elided realization relationship. The second style renders the interface in its expanded form, perhaps revealing its operations. The component that realizes the interface is connected to the interface using a full realization relationship. In both cases, the component that accesses the services of the other component through the interface is connected to the interface using a dependency relationship.

Figure Components and Interfaces



An interface that a component realizes is called an *export interface,* meaning an interface that the component provides as a service to other components. A component may provide many export interfaces. The interface that a component uses is called an *import interface,* meaning an interface that the component conforms to and so builds on. A component may conform to many import interfaces. Also, a component may both import and export interfaces.

A given interface may be exported by one component and imported by another. The fact that this interface lies between the two components breaks the direct dependency between the components. A component that uses a given interface will function properly no matter what component realizes that interface. Of course, a component can be used in a context if and only if all its import interfaces are provided by the export interfaces of other components.

Binary Replaceability

The basic intent of every component-based operating system facility is to permit the assembly of systems from binary replaceable parts. This means that you can create a system out of components and then evolve that system by adding new components and replacing old ones, without rebuilding the system. Interfaces are the key to making this happen. When you specify an interface, you can drop into the executable system any component that conforms to or provides that interface. You can extend the system by making the components provide new services through other interfaces, which, in turn, other components can discover and use. These semantics explain the intent behind the definition of components in the UML. A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.

First, a component is *physical.* It lives in the world of bits, not concepts.

Second, a component is *replaceable.* **A component is substitutable• it is possible to replace a component** with another that conforms to the same interfaces. Typically, the mechanism of inserting or replacing a component to form a run time system is transparent to the component user and is enabled by object models (such as COM+ and Enterprise Java Beans) that require little or no intervening transformation or by tools that automate the mechanism.

Third, a component is *part of a system.* A component rarely stands alone. Rather, a given component collaborates with other components and in so doing exists in the architectural or technology context in which it is intended to be used. A component is logically and physically cohesive and thus denotes a meaningful structural and/or behavioral

chunk of a larger system. A component may be reused across many systems. Therefore, a component represents a fundamental building block on which systems can be designed and composed. **This definition is recursive• a system** at one level of abstraction may simply be a component at a higher level of abstraction.

Fourth, as discussed in the previous section, a component *conforms to and provides the realization of a set of interfaces.*

Kinds of Components

Three kinds of components may be distinguished.

First, there are *deployment components.* These are the components necessary and sufficient to form an executable system, such as dynamic libraries (DLLs) and executables (EXEs). The UML's definition of component is broad enough to address classic object models, such as COM+, CORBA, and Enterprise Java Beans, as well as alternative object models, perhaps involving dynamic Web pages, database tables, and executables using proprietary communication mechanisms.

Second, there are *work product components.* These components are essentially the residue of the development process, consisting of things such as source code files and data files from which deployment components are created. These components do not directly participate in an executable system but are the work products of development that are used to create the executable system.

Third are *execution components.* These components are created as a consequence of an executing system, such as a COM+ object, which is instantiated from a DLL.

Organizing Components

You can organize components by grouping them in packages in the same manner in which you organize classes.

You can also organize components by specifying dependency, generalization, association (including aggregation), and realization relationships among them.

Standard Elements

All the UML's extensibility mechanisms apply to components. Most often, you'll use tagged values to extend component properties (such as specifying the version of a development component) and stereotypes to specify new kinds of components (such as operating system-specific components).

The UML defines five standard stereotypes that apply to components:

1. **executable**	Specifies a component that may be executed on a node
2. **library**	Specifies a static or dynamic object library
3. **table**	Specifies a component that represents a database table
4. **file**	Specifies a component that represents a document containing source code or Data
5. **document**	Specifies a component that represents a document

Common Modeling Techniques

Modeling Executables and Libraries

The most common purpose for which you'll use components is to model the deployment components that make up your implementation. If you are deploying a trivial system whose implementation consists of exactly one executable file, you will not need to do any component modeling. If, on the other hand, the system you are deploying is made up of several executables and associated object libraries, doing component modeling will help you to visualize, specify, construct, and document the decisions you've made about the physical system. Component modeling is even more important if you want to control the versioning and configuration management of these parts as your system evolves.

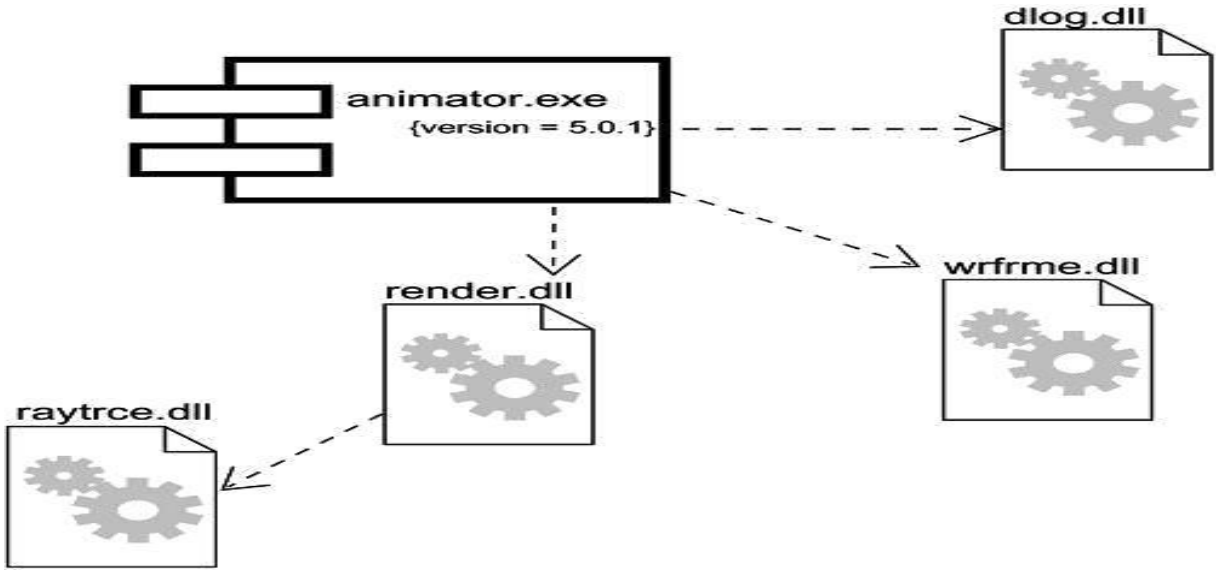
For most systems, these deployment components are drawn from the decisions you make about how to segment the physical implementation of your system. These decisions will be affected by a number of technical issues (such as your choice of component-based operating system facilities), configuration management issues (such as your decisions about which parts will likely change over time), and reuse issues (that is, deciding which components you can reuse in or from other systems).

To model executables and libraries,

- Identify the partitioning of your physical system. Consider the impact of technical, configuration management, and reuse issues.
- Model any executables and libraries as components, using the appropriate standard elements. If your implementation introduces new kinds of components, introduce a new appropriate stereotype.
- If it's important for you to manage the seams in your system, model the significant interfaces that some components use and others realize.
- As necessary to communicate your intent, model the relationships among these executables, libraries, and interfaces. Most often, you'll want to model the dependencies among these parts in order to visualize the impact of change.

For example, Figure shows a set of components drawn from a personal productivity tool that runs on a single personal computer. This figure includes one executable (**animator.exe**, with a tagged value noting its version number) and four libraries (**dlog.dll**, **wrfrme.dll**, **render.dll**, and **raytrce.dll**), all of which use the UML's standard elements for executables and libraries, respectively. This diagram also presents the dependencies among these components.

Figure Modeling Executables and Libraries



As your models get bigger, you will find that many components tend to cluster together in groups that are conceptually and semantically related. In the UML, you can use packages to model these clusters of components.For larger systems that are deployed across several computers, you'll want to model the way your components are distributed by asserting the nodes on which they are located.

Modeling Tables, Files, and Documents

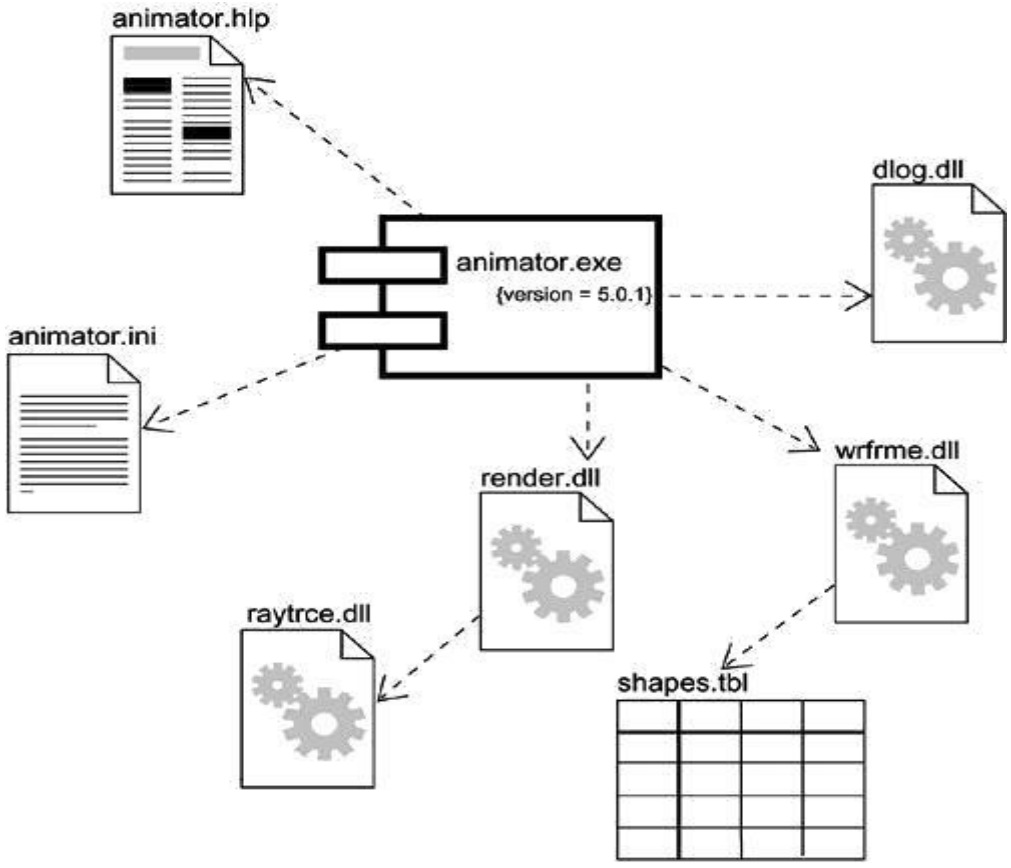
Modeling the executables and libraries that make up the physical implementation of your system is useful, but often you'll find there are a host of ancillary deployment components that are neither executables nor libraries and yet are critical to the physical deployment of your system. For example, your implementation might include data files, help documents, scripts, log files, initialization files, and installation/removal files. Modeling these components is an important part of controlling the configuration of your system. Fortunately, you can use UML components to model all of these artifacts.

To model tables, files, and documents,

- Identify the ancillary components that are part of the physical implementation of your system.
- Model these things as components. If your implementation introduces new kinds of artifacts, introduce a new appropriate stereotype.
- As necessary to communicate your intent, model the relationships among these ancillary components and the other executables, libraries, and interfaces in your system. Most often, you'll want to model the dependencies among these parts in order to visualize the impact of change.

For example, Figure builds on the previous figure and shows the tables, files, and documents that are part of the deployed system surrounding the executable **animator.exe.** This figure includes one document (**animator.hlp**), one simple file (**animator.ini**), and one database table (**shapes.tbl**), all of which use the UML's standard elements for cuments, files, and tables, respectively.

Figure Modeling Tables, Files, and Documents



Modeling databases can get complicated when you start dealing with multiple tables, triggers, and stored procedures. To visualize, specify, construct, and document these features, you'll need to model the logical schema, as well as the physical databases.

Modeling an API

If you are a developer who's assembling a system from component parts, you'll often want to see the application programming interfaces (APIs) that you can use to glue these parts together. APIs represent the programmatic seams in your system, which you can model using interfaces and components.

An API is essentially an interface that is realized by one or more components. As a developer, you'll really care only about the interface itself; which component realizes an interface's operations is not relevant as long as *some* component realizes it. From a system configuration management perspective, though, these realizations are important because you need to ensure that, when you publish an API, there's some realization available that carries out the API's obligations. Fortunately, with the UML, you can model both perspectives.

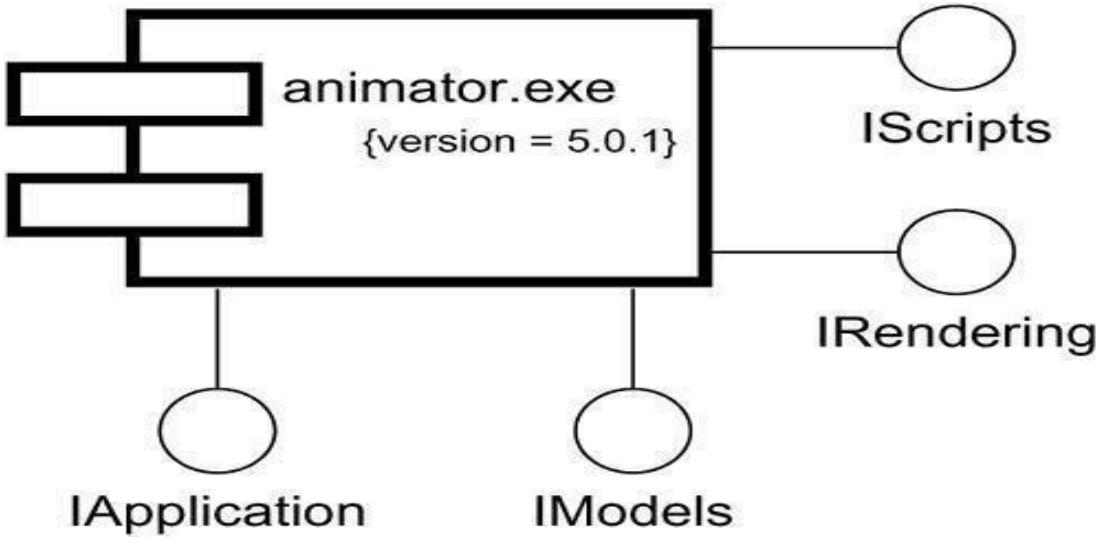
The operations associated with any semantically rich API will be fairly extensive, so most of the time you won't need to visualize all these operations at once. Instead, you'll tend to keep theoperations in the backplane of your models and use interfaces as handles with which you can find these sets of operations. If you want to construct executable systems against these APIs, you will need to add enough detail so that your development tools can compile against the properties of your interfaces. Along with the signatures of each operation, you'll probably also want to include uses cases that explain how to use each interface.

To model an API,

- Identify the programmatic seams in your system and model each seam as an interface, collecting the attributes and operations that form this edge.
- Expose only those properties of the interface that are important to visualize in the given context; otherwise, hide these properties, keeping them in the interface's specification for reference, as necessary.
- Model the realization of each API only insofar as it is important to show the configuration of a specific implementation.

Figure exposes the APIs of the executable in the previous two figures. You'll see four interfaces that form the API of the executable: **IApplication**, **IModels**, **IRendering**, and **IScripts.**

Figure Modeling an API



Modeling Source Code

The most common purpose for which you'll use components is to model the physical parts that make up your implementation. This also includes the modeling of all the ancillary parts of your deployed system, including tables, files, documents, and APIs. The second most common purpose for which you'll use components is to model the configuration of all the source code files that your development tools use to create these components. These represent the work product components of your development process.

Modeling source code graphically is particularly useful for visualizing the compilation dependencies among your source code files and for managing the splitting and merging ofgroups of these files when you fork and join development paths. In this manner, UML components can be the graphical interface to your configuration management and version control tools.

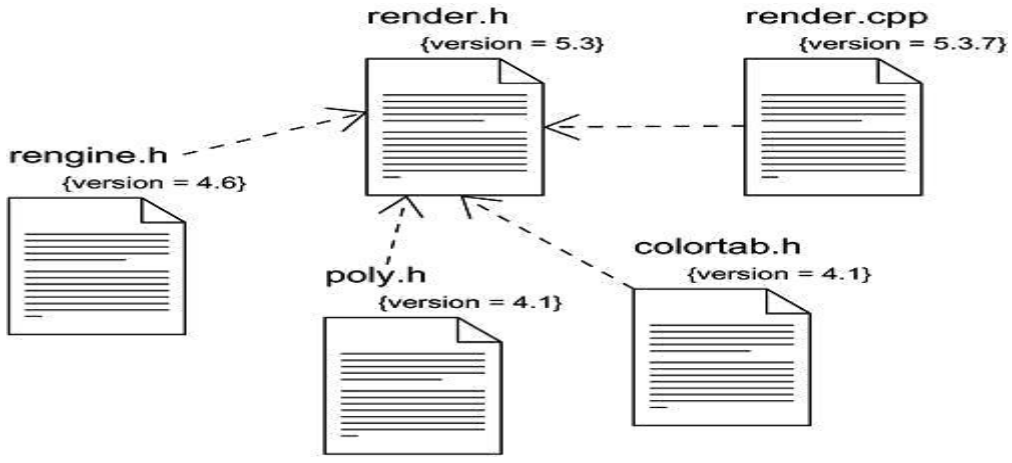
For most systems, source code files are drawn from the decisions you make about how to segment the files your development environment needs. These files are used to store the details of your classes, interfaces, collaborations, and other logical elements as an intermediate step to creating the physical, binary components that are derived from these elements by your tools. Most of the time, these tools will impose a style of organization (one or two files per class is common), but you'll still want to visualize the relationships among these files. How you organize groups of these files using packages and how you manage versions of these files is driven by your decisions about how to manage change.

To model source code,

- Depending on the constraints imposed by your development tools, model the files used to store the details of all your logical elements, along with their compilation dependencies.
- If it's important for you to bolt these models to your configuration management and version control tools, you'll want to include tagged values, such as version, author, and check in/check out information, for each file that's under configuration management.
- As far as possible, let your development tools manage the relationships among these files, and use the UML only to visualize and document these relationships.

For example, Figure shows some source code files that are used to build the library r**ender.dll** from the previous examples. This figure includes four header files (**render.h**, **rengine.h**, **poly.h**, and **colortab.h**) that represent the source code for the specification of certain classes. There is also one implementation file (**render.cpp**) that represents the implementation of one of these headers.

Figure Modeling Source Code



As your models get bigger, you will find that many source code files tend to cluster together in groups that are conceptually and semantically related. Most of the time, your development tools will place these groups in separate directories. In the UML, you can use packages to model these clusters of source code files.

In the UML, it is possible to visualize the relationship of a class to its source code file and, in turn, the relationship of a source code file to its executable or library by using trace relationships. However, you'll rarely need to go to this detail of modeling.

Deployment

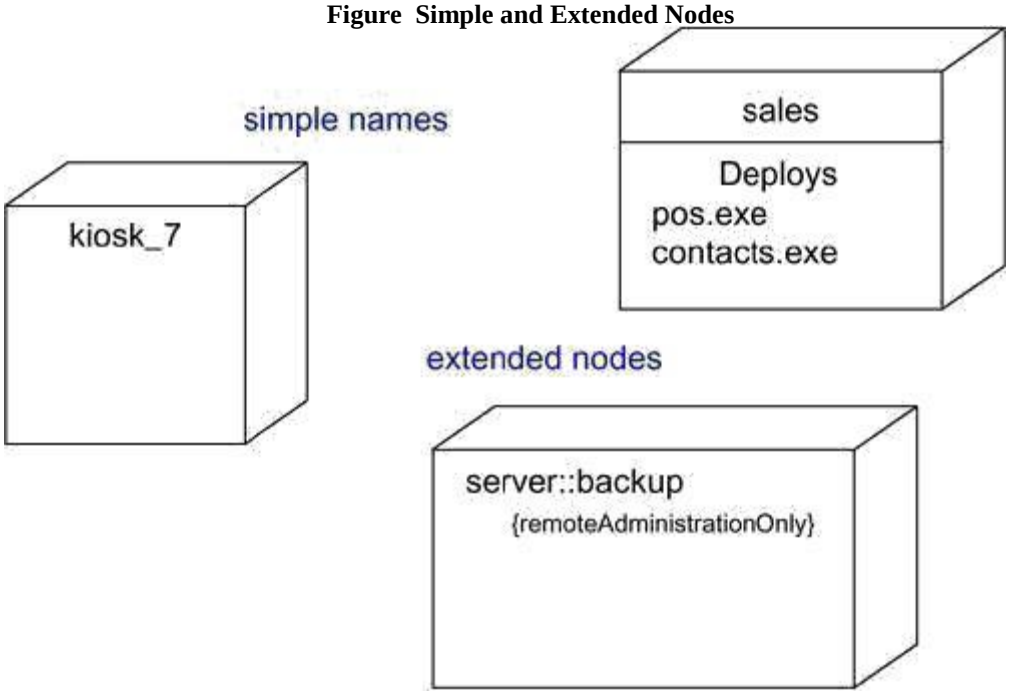
Terms and Concepts

A *node* is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. Graphically, a node is rendered as a cube.

Names

Every node must have a name that distinguishes it from other nodes. A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the node name prefixed by the name of the package in which that node lives. A node is typically drawn showing only its name, as in Figure . Just as with classes, you may draw nodes adorned with tagged values or with additional compartments to expose their details.

Figure Simple and Extended Nodes



Note

A node name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the colon, which is used to separate a node name and the name of its enclosing package) and may continue over several lines. In practice, node names are short nouns or noun phrases drawn from the vocabulary of the implementation.

Nodes and Components

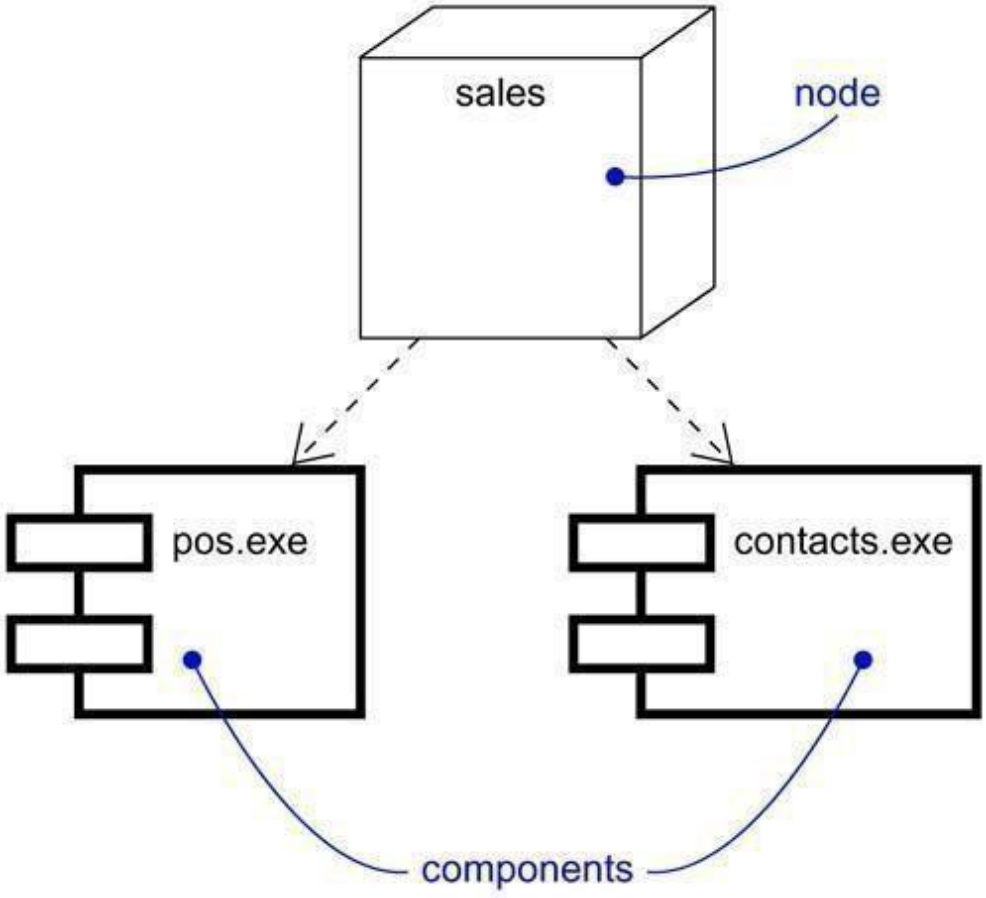
In many ways, nodes are a lot like components: Both have names; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions. However, there are some significant differences between nodes and components.

- Components are things that participate in the execution of a system; nodes are things that execute components.
- Components represent the physical packaging of otherwise logical elements; nodes represent the physical deployment of components.

This first difference is the most important. Simply put, nodes execute components; components are things that are executed by nodes.

The second difference suggests a relationship among classes, components, and nodes. In particular, a component is the materialization of a set of other logical elements, such as classes and collaborations, and a node is the location upon which components are deployed. A class may be implemented by one or more components, and, in turn, a component may be deployed on one or more nodes. As Figure shows, the relationship between a node and the components it deploys can be shown explicitly by using a dependency relationship. Most of the time, you won't need to visualize these relationships graphically but will keep them as a part of the node's specification.

Figure Nodes and Components



A set of objects or components that are allocated to a node as a group is called a *distribution unit.*

Note

Nodes are also class-like in that you can specify attributes and operations for them. For example, you might specify that a node provides the attributes **processorSpeed** and **memory,** as well as the operations **turnOn, turnOff,** and **suspend.**

Organizing Nodes

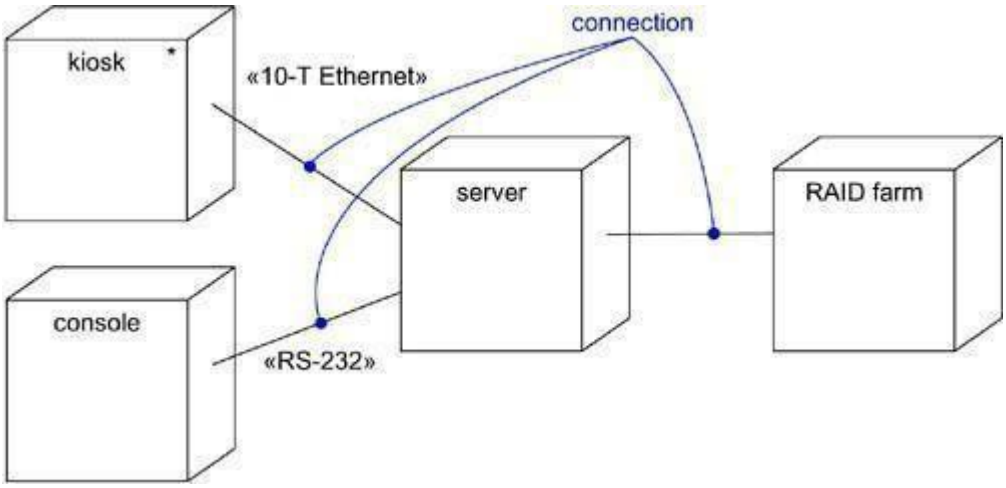
You can organize nodes by grouping them in packages in the same manner in which you can organize classes and components.

You can also organize nodes by specifying dependency, generalization, and association (including aggregation) relationships among them.

Connections

The most common kind of relationship you'll use among nodes is an association. In this context, an association represents a physical connection among nodes, such as an Ethernet connection, a serial line, or a shared bus, as Figure shows. You can even use associations to model indirect connections, such as a satellite link between distant processors.

Figure Connections



Because nodes are class-like, you have the full power of associations at your disposal. This means that you can include roles, multiplicity, and constraints. As in the previous figure, you should stereotype these associations if you want to model new kinds of **connections•** for example, to distinguish between a 10-T Ethernet connection and an RS-232 serial connection.

Common Modeling Techniques

Modeling Processors and Devices

Modeling the processors and devices that form the topology of a stand-alone, embedded, client/server, or distributed system is the most common use of nodes.Because all of the UML's extensibility mechanisms apply to nodes, you will often use stereotypes to specify new kinds of nodes that you can use to represent specific kinds of processors and devices. A *processor* is a

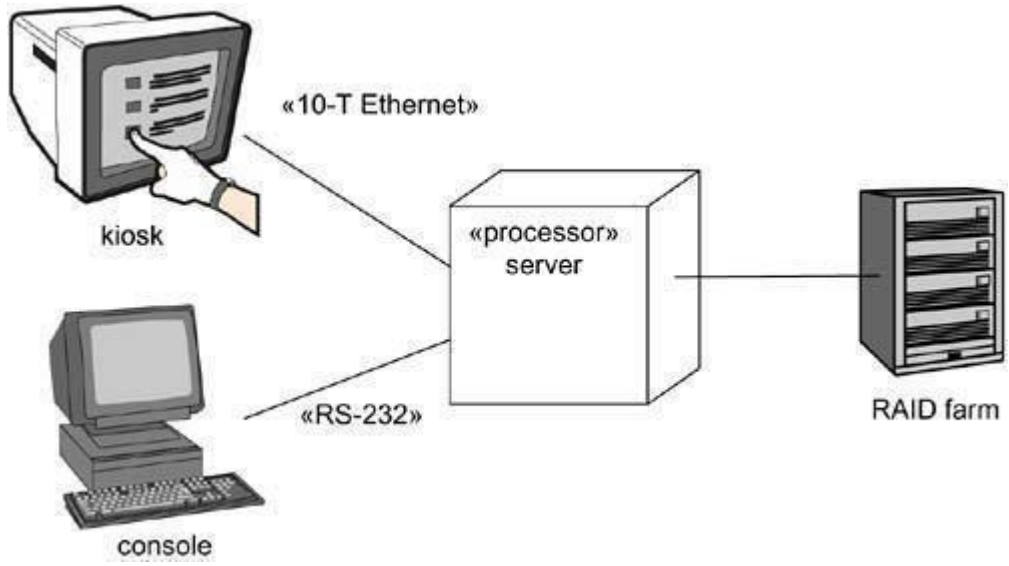
node that has processing capability, meaning that it can execute a component. A *device* is a node that has no processing capability (at least, none that are modeled at this level of abstraction) and, in general, represents something that interfaces to the real world.

To model processors and devices,

- Identify the computational elements of your system's deployment view and model each as a node.If these elements represent generic processors and devices, then stereotype them as such. If they are kinds of processors and devices that are part of the vocabulary of your domain, then specify an appropriate stereotype with an icon for each.
- As with class modeling, consider the attributes and operations that might apply to each node.

For example, Figure takes the previous diagram and stereotypes each node. The **server** is a node stereotyped as a generic processor; the **kiosk** and the **console** are nodes stereotyped as special kinds of processors; and the **RAID farm** is a node stereotyped as a special kind of device.

Figure Processors and Devices



Modeling the Distribution of Components

When you model the topology of a system, it's often useful to visualize or specify the physical distribution of its components across the processors and devices that make up the system.

To model the distribution of components,

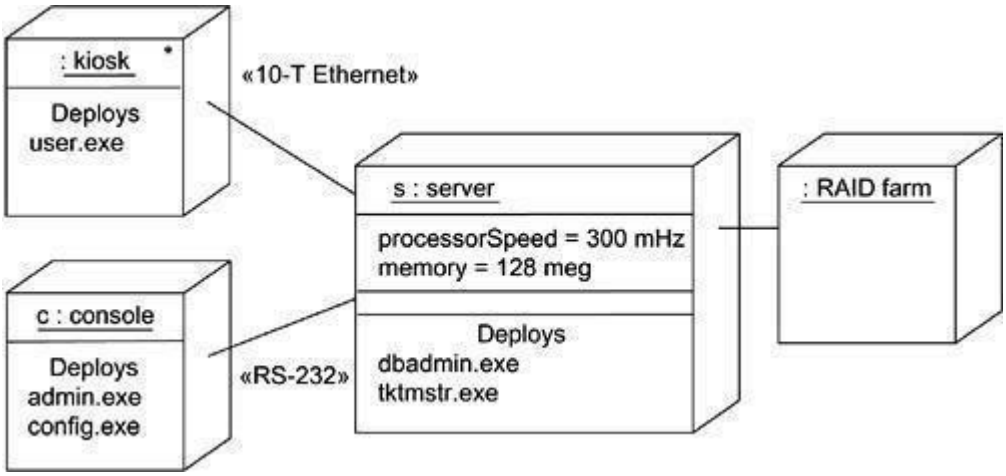
- For each significant component in your system, allocate it to a given node.

- Consider duplicate locations for components. It's not uncommon for the same kind of component (such as specific executables and libraries) to reside on multiple nodes simultaneously.
- Render this allocation in one of three ways.

    1. Don't make the allocation visible, but leave it as part of the backplane of your **model• that is, in each node's** specification.
    2. Using dependency relationships, connect each node with the components it deploys.
    3. List the components deployed on a node in an additional compartment.

Using the third approach, Figure takes the earlier diagrams and specifies the executable components that reside on each node. This diagram is a bit different from the previous ones in that it is an object diagram, visualizing specific instances of each node. In this case, the **RAID farm** and **kiosk** instances are both anonymous and the other two instances are named (c for the **console** and **s** for the **server**). Each processor in this figure is rendered with an additional compartment showing the component it deploys. The **server** object is also rendered with its attributes (**processorSpeed** and **memory**) and their values visible.

Figure Modeling the Distribution of Components.



Components need not be statically distributed across the nodes in a system. In the UML, it is possible to model the dynamic migration of components from node to node, as in an agent-based system or a high-reliability system that involves clustered servers and replicated databases.

Collaborations

Component Diagrams

Terms and Concepts

A *component diagram* shows a set of components and their relationships. Graphically, a component diagram is a collection of vertices and arcs.

Common Properties

A component diagram is just a special kind of diagram and shares the same common properties as do all **other diagrams• a name and graphical c**ontents that are a projection into a model. What distinguishes a component diagram from all other kinds of diagrams is its particular content.

Contents

Component diagrams commonly contain

- Components
- Interfaces
- Dependency, generalization, association, and realization relationships Like

all other diagrams, component diagrams may contain notes and constraints.

Component diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your component diagrams, as well, especially when you want to visualize one instance of a family of component-based systems.

Common Uses

You use component diagrams to model the static implementation view of a system. This view primarily supports the configuration management of a system's parts, made up of components that can be assembled in various ways to produce a running system.

When you model the static implementation view of a system, you'll typically use component diagrams in one of four ways.

1. To model source code

With most contemporary object- oriented programming languages, you'll cut code using integrated development environments that store your source code in files. You can use component diagrams to model the configuration management of these files, which represent work-product components.

2. To model executable releases

A release is a relatively complete and consistent set of artifacts delivered to an internal or external user. In the context of components, a release focuses on the parts necessary to deliver a running system. When you model a release using component diagrams, you are visualizing, specifying, and documenting the decisions about the physical parts **that constitute your software• that is, its deployment components.**

3. To model physical databases

Think of a physical database as the concrete realization of a schema, living in the world of bits. Schemas, in effect, offer an API to persistent information; the model of a physical database represents the storage of that information in the tables of a relational database or the pages of an object-oriented database. You use component diagrams to represent these and other kinds of physical databases.

4. To model adaptable systems

Some systems are quite static; their components enter the scene, participate in an execution, and then depart. Other systems are more dynamic, involving mobile agents or components that migrate for purposes of load balancing and failure recovery. You use component diagrams in conjunction with some of the UML's diagrams for modeling behavior to represent these kinds of systems.

Common Modeling Techniques

Modeling Source Code

If you develop software in Java, you'll usually save your source code in **.java** files. If you develop software using C++, you'll typically store your source code in header files (**.h** files) and bodies (**.cpp** files). If you use IDL to develop COM+ or CORBA applications, one interface from your design view will often expand into four source code files: the interface itself, the client proxy, the server stub, and a bridge class. As your application grows, no matter which language you use, you'll find yourself organizing these files into larger groups. Furthermore, during the construction phase of development, you'll probably end up creating new versions of some of these files for each new incremental release you produce, and you'll want to place these versions under the control of a configuration management system.

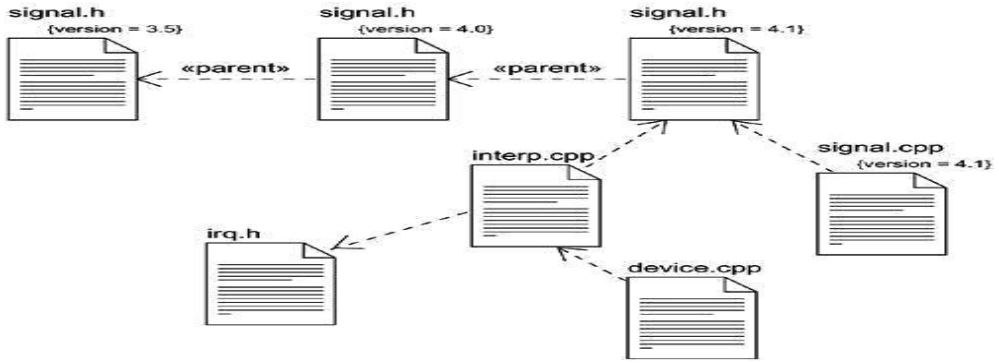
Much of the time, you will not need to model this aspect of a system directly. Instead, you'll let your development environment keep track of these files and their relationships. Sometimes, however, it's helpful to visualize these source code files and their relationships using component diagrams. Component diagrams used in this way typically contain only work-product components stereotyped as files, together with dependency relationships. For example, you might reverse engineer a set of source code files to visualize their web of compilation dependencies. You can go in the other direction by specifying the relationships among your source code files and then using those models as input to compilation tools, such as **make** on Unix. Similarly, you might want to use component diagrams to visualize the history of a set of source code files that are under configuration management. By extracting information from your configuration management system, such as the number of times a source code file has been checked out over a period of time, you can use that information to color component diagrams, showing "hot spots" of change among your source code files and areas of architectural churn.

To model a system's source code,

- Either by forward or reverse engineering, identify the set of source code files of interest and model them as components stereotyped as files.
- For larger systems, use packages to show groups of source code files.
- Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.
- Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies.

For example, Figure shows five source code files. **signal.h** is a header file. Three of its versions are shown, tracing from new versions back to their older ancestors. Each variant of this source code file is rendered with a tagged value exposing its version number.

Figure Modeling Source Code



This header file (**signal.h**) is used by two other files (**interp.cpp** and **.signal.cpp**), both of which are bodies. One of these files (**interp.cpp**) has a compilation dependency to another header (**irq.h**); in turn, **device.cpp** has a compilation dependency to **interp.cpp.** Given this component diagram, it's easy to trace the impact of changes. For example, changing the source code file **signal.h** will require the recompilation of three other files: **signal.cpp**, **interp.cpp**, and transitively, **device.cpp.** As this diagram also shows, the file **irq.h** is not affected.

Diagrams such as this can easily be generated by reverse engineering from the information held by your development environment's configuration management tools.

Modeling an Executable Release

Releasing a simple application is easy: You throw the bits of a single executable file on a disk, and your users just run that executable. For these kinds of applications, you don't need component diagrams because there's nothing difficult to visualize, specify, construct, or document.

Releasing anything other than a simple application is not so easy. You need the main executable (usually, a **.exe** file), but you also need all its ancillary parts, such as libraries (commonly **.dll** files if you are working in the context of COM+, or **.class** and **.jar** files if you are working in the context of Java), databases, help files, and resource files. For distributed systems, you'll likely have multiple executables and other parts scattered across various nodes. If you are working with a system of applications, you'll find that some of these components are unique to each application but that many are shared among applications. As you evolve your system, controlling the configuration of these many components **becomes an important activity• and a more d**ifficult one because changes in the components associated with one application may affect the operation of other applications.

For this reason, you use component diagrams to visualize, specify, construct, and document the configuration of your executable releases, encompassing the deployment components that form each release and the relationships among those components. You can use component diagrams to forward engineer a new system and to reverse engineer an existing one.

When you create component diagrams such as these, you actually just model a part of the things and relationships that make up your system's implementation view. For this reason, each component diagram should focus on one set of components at a time.

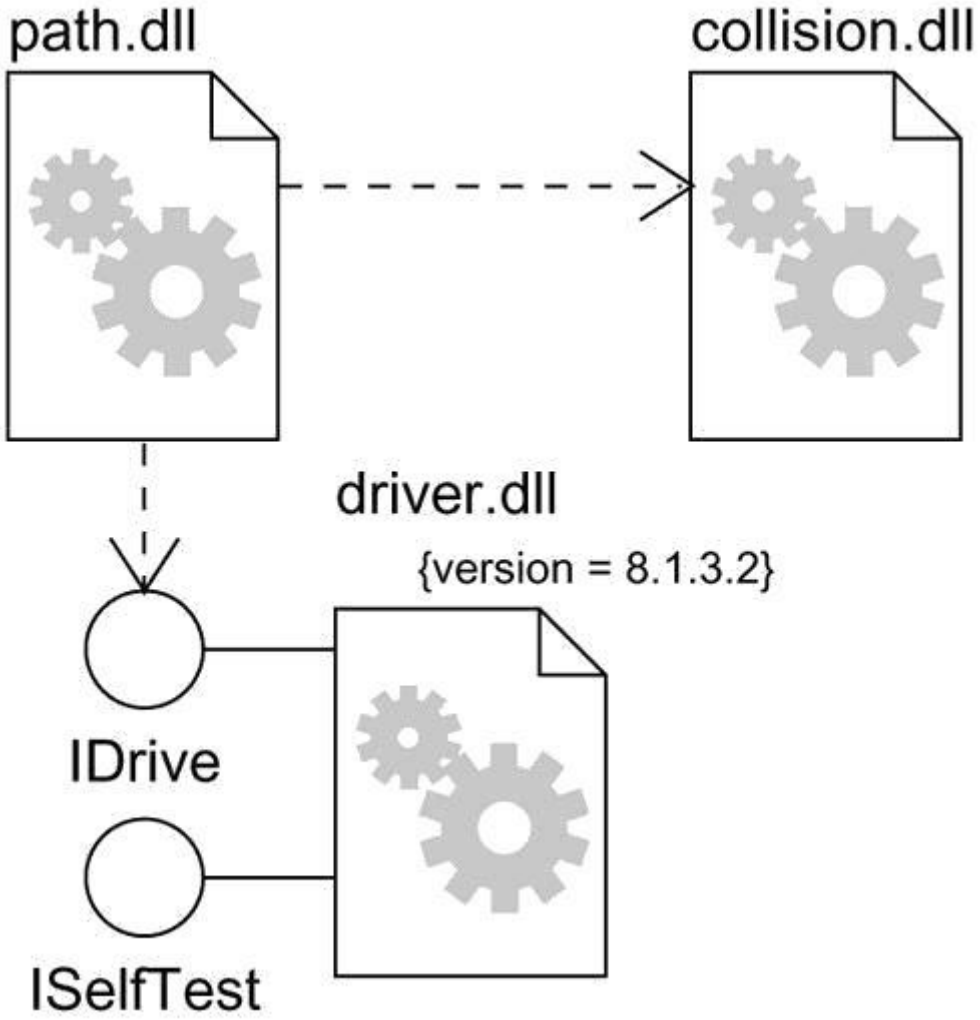
To model an executable release,

- Identify the set of components you'd like to model. Typically, this will involve some or all the components that live on one node, or the distribution of these sets of components across all the nodes in the system.
- Consider the stereotype of each component in this set. For most systems, you'll find a small number of different kinds of components (such as executables, libraries, tables, files, and documents). You can use the UML's extensibility mechanisms to provide visual cues for these stereotypes.
- For each component in this set, consider its relationship to its neighbors. Most often, this will involve interfaces that are exported (realized) by certain components and then imported (used) by others. If you want to expose the seams in your system, model these interfaces explicitly. If you want your model at a higher level of abstraction, elide these relationships by showing only dependencies among the components.

For example, Figure models part of the executable release for an autonomous robot. This figure focuses on the deployment components associated with the robot's driving and calculation functions. You'll find one component (**driver.dll**) that exports an interface (**IDrive**) that is, in turn, imported by another component (**path.dll**). **driver.dll** exports one other interface (**ISelfTest**) that is probably used by other

components in the system, although they are not shown here. There's one other component shown in this diagram (**collision.dll**), and it, too, exports a set of interfaces, although these details are elided: **path.dll** is shown with a dependency directly to **collision.dll.**

Figure Modeling an Executable Release



There are many more components involved in this system. However, this diagram only focuses on those deployment components that are directly involved in moving the robot. Note that in this component-based architecture, you could replace a specific version of **driver.dll** with another that realized the same (and perhaps additional) interfaces, and **path.dll** would still function properly. If you want to be explicit about the operations that **driver.dll** realizes, you could always render its interface using class notation, stereotyped as **»interface .**

Modeling a Physical Database

A logical database schema captures the vocabulary of a system's persistent data, along with the semantics of their relationships. Physically, these things are stored in a database for later retrieval, either a relational database, an object-oriented one, or a hybrid object/relational database. The UML is well suited to modeling physical databases, as well as logical database schemas.

*Physical database design is beyond the scope of this book; the focus here is simply to show you how you can model databases and tables using the UML.*Mapping a logical database schema to an object-oriented

database is straightforward because even complex inheritance lattices can be made persistent directly. Mapping a logical database schema to a relational database is not so simple, however. In the presence of inheritance, you have to make decisions about how to map classes to tables. Typically, you can apply one or a combination of three strategies.

1. Define a separate table for each class. This is a simple but naive approach because it introduces maintenance headaches when you add new child classes or modify your parent classes.

2. Collapse your inheritance lattices so that all instances of any class in a hierarchy has the same state. The downside with this approach is that you end up storing superfluous information for many instances.

3. Separate parent and child states into different tables. This approach best mirrors your inheritance lattice, but the downside is that traversing your data will require many cross-table joins.

When designing a physical database, you also have to make decisions about how to map operations defined in your logical database schema. Object- oriented databases make the mapping fairly transparent. But, with relational databases, you have to make some decisions about how these logical operations are implemented. Again, you have some choices.

1. For simple CRUD (create, read, update, delete) operations, implement them with standard SQL or ODBC calls.

2. For more-complex behavior (such as business rules), map them to triggers or stored procedures.

Given these general guidelines, to model a physical database,

- Identify the classes in your model that represent your logical database schema.

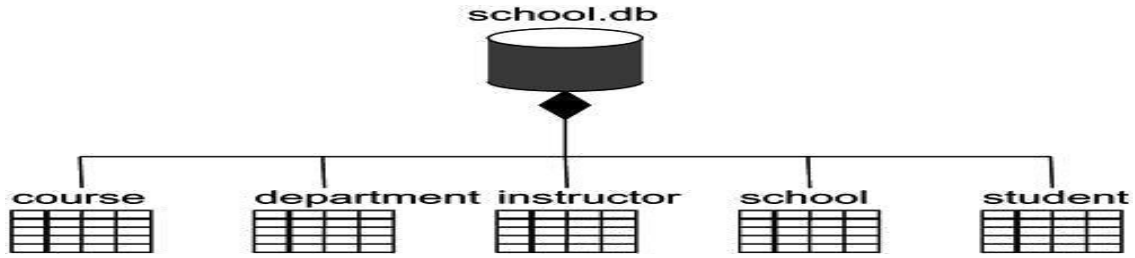
- Select a strategy for mapping these classes to tables. You will also want to consider the physical distribution of your databases. Your mapping strategy will be affected by the location in which you want your data to live on your deployed system.

- To visualize, specify, construct, and document your mapping, create a component diagram that contains components stereotyped as tables.

- Where possible, use tools to help you transform your logical design into a physical design.

Figure shows a set of database tables drawn from an information system for a school. You will find one database (**school.db**, rendered as a component stereotyped as **database**) that's composed of five tables: **student**, **class**, **instructor**, **department**, and **course** (rendered as a component stereotyped as **table,** one of the UML's standard elements). In the corresponding logical database schema, there was no inheritance, so mapping to this physical database design is straightforward.

Figure Modeling a Physical Database



Although not shown in this example, you can specify the contents of each table. Components can have attributes, so a common idiom when modeling physical databases is to use these attributes to specify the columns of each table. Similarly, components can have operations, and these can be used to denote stored procedures.

Modeling Adaptable Systems

All the component diagrams shown thus far have been used to model static views. Their components spend their entire lives on one node. This is the most common situation you'll encounter, but especially in the domain of complex, distributed systems, you'll need to model dynamic views. For example, you might have a system that replicates its databases across several nodes, switching the one that is the primary database when a server goes down. Similarly, if you are modeling a globally distributed 24x7 operation (that is, a system that's up 24 hours a day, 7 days a week), you will likely encounter mobile agents, components that migrate from node to node to carry out some transaction. To model these dynamic views, you'll need to use a combination of component diagrams, object diagrams, and interaction diagrams.

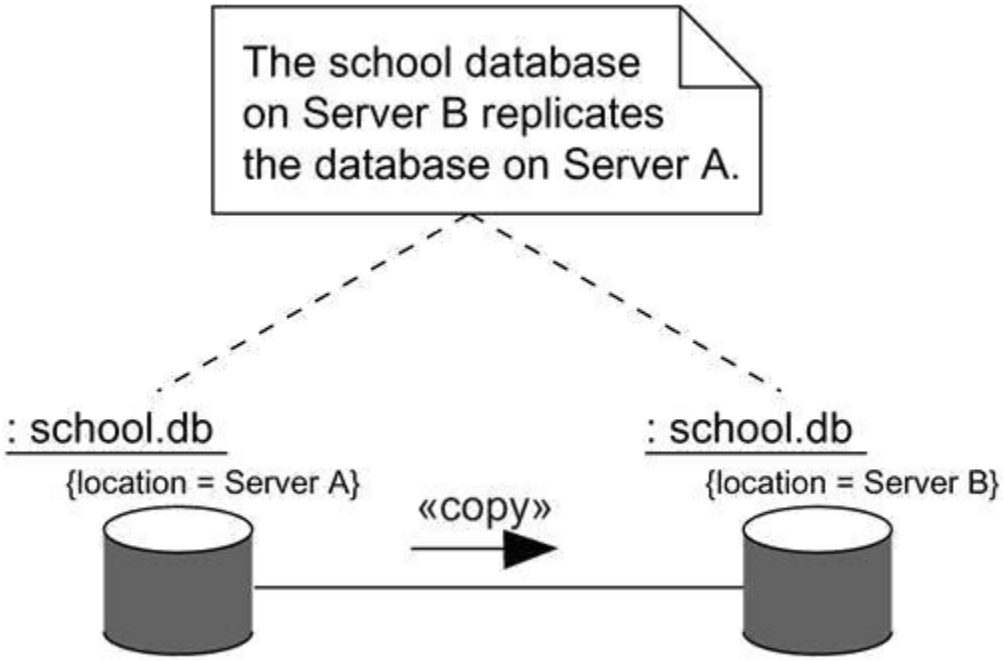
To model an adaptable system,

Consider the physical distribution of the components that may migrate from node to node. You can specify the location of a component instance by marking it with a location tagged value, which you can then render in a component diagram (although, technically speaking, a diagram that contains only instances is an object diagram).

- If you want to model the actions that cause a component to migrate, create a corresponding interaction diagram that contains component instances. You can illustrate a change of location by drawing the same instance more than once, but with different values for its location tagged value.

For example, Figure models the replication of the database from the previous figure. We show two instances of the component **school.db.** Both instances are anonymous, and both have a different value for their location tagged value. There's also a note, which explicitly specifies which instance replicates the other.

Figure Modeling Adaptable Systems



If you want to show the details of each database, you can render them in their canonical form• a component stereotyped as a **database.**Although not shown here, you could use an interaction diagram to model the dynamics of switching from one primary database to another.

Forward and Reverse Engineering

Forward engineering and reverse engineering components are pretty direct, because components are themselves physical things (executables, libraries, tables, files, and documents) that are therefore close to the running system. When you forward engineer a class or a collaboration, you really forward engineer to a component that represents the source code, binary library, or executable for that class or collaboration. Similarly, when you reverse engineer source code, binary libraries, or executables, you really reverse engineer to a component or set of components that, in turn, trace to classes or collaborations.

Choosing to forward engineer (the creation of code from a model) a class or collaboration to source code, a binary library, or an executable is a mapping decision you have to make. You'll want to take your logical models to source code if you are interested in controlling the configuration management of files that are then manipulated by a development environment. You'll want to take your logical models directly to binary libraries or executables if you are interested in managing the components that you'll actually deploy on a running system. In some cases, you'll want to do both. A class or collaboration may be denoted by source code, as well as by a binary library or executable.

To forward engineer a component diagram,

- For each component, identify the classes or collaborations that the component implements.
- Choose the target for each component. Your choice is basically between source code (a form that can be manipulated by development tools) or a binary library or executable (a form that can be dropped into a running system).
- Use tools to forward engineer your models.

Reverse engineering (the creation of a model from code) a component diagram is not a perfect process because there is always a loss of information. From source code, you can reverse engineer back to classes; this is the most common thing you'll do. Reverse engineering source code to components will uncover compilation dependencies among those files. For binary libraries, the best you can hope for is to denote the library as a component and then discover its interfaces by reverse engineering. This is the second most common thing you'll do with component diagrams. In fact, this is a useful way to approach a set of new libraries that may be otherwise poorly documented. For executables, the best you can hope **for is to denote the executable as a component and then disassemble its code• something you'll rarely** need to do unless you work in assembly language.

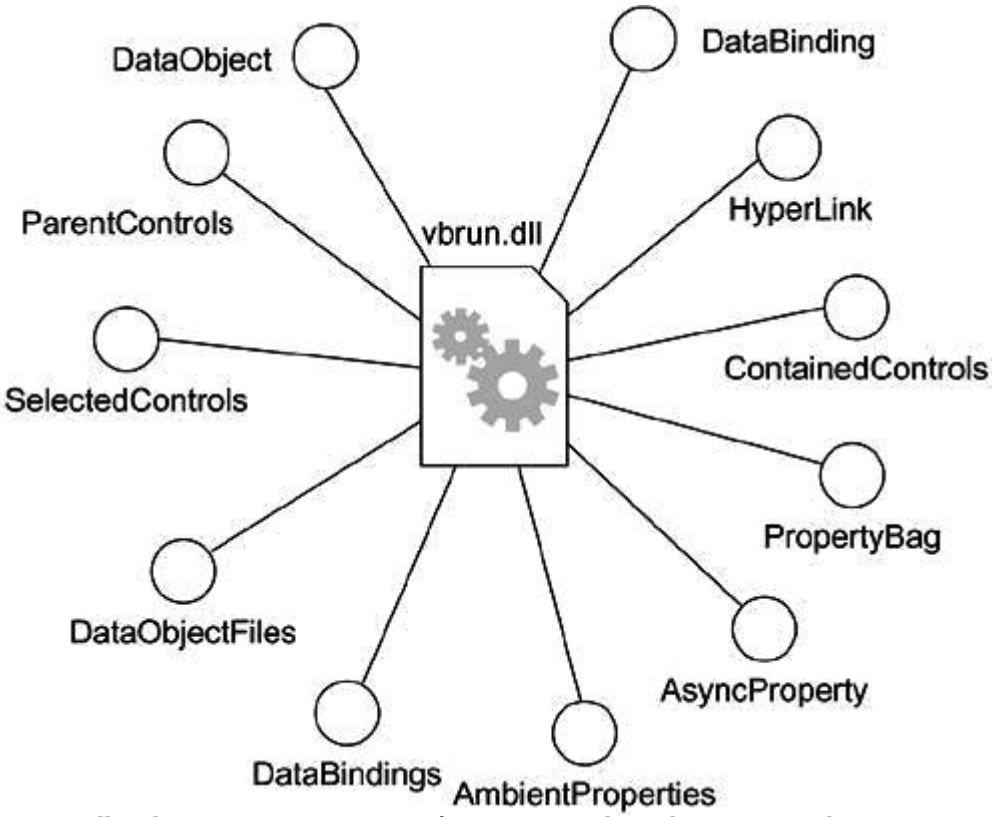
To reverse engineer a component diagram,

- Choose the target you want to reverse engineer. Source code can be reverse engineered to components and then classes. Binary libraries can be reverse engineered to uncover their interfaces. Executables can be reverse engineered the least.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or to modify an existing one that was previously forward engineered.
- Using your tool, create a component diagram by querying the model. For example, you might start with one or more components, then expand the diagram by following relationships or neighboring components. Expose or hide the details of the contents of this component diagram as necessary to communicate your intent.

For example, Figure provides a component diagram that represents the reverse engineering of the ActiveX component **vbrun.dll.** As the figure shows, the component realizes 11 interfaces. Given this diagram, you

can begin to understand the semantics of the component by next exploring the details of its interfaces.

Figure Reverse Engineering



Especially when you reverse engineer from source code, and sometimes when you reverse engineer from binary libraries and executables, you'll do so in the context of a configuration management system. This means that you'll often be working with specific versions of files or libraries, with all versions of a configuration compatible with one another. In these cases, you'll want to include a tagged value that represents the component version, which you can derive from your configuration management system. In this manner, you can use the UML to visualize the history of a component across various releases.

Deployment Diagrams

Terms and Concepts

A *deployment diagram* is a diagram that shows the configuration of run time processing nodes and the components that live on them. Graphically, a deployment diagram is a collection of vertices and arcs.

Common Properties

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

Contents

Deployment diagrams commonly contain

- Nodes
- Dependency and association relationships

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

Deployment diagrams may also contain components, each of which must live on some node. Deployment diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your deployment diagrams, as well, especially when you want to visualize one instance of a family of hardware topologies.

> **Note**
>
> In many ways, a deployment diagram is just a special kind of class diagram, which focuses on a system's nodes.

Common Uses

You use deployment diagrams to model the static deployment view of a system. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system.

There are some kinds of systems for which deployment diagrams are unnecessary. If you are developing a piece of software that lives on one machine and interfaces only with standard devices on that machine that are already managed by the host operating system (for example, a personal computer's keyboard, display, and modem), you can ignore deployment diagrams. On the other hand, if you are developing a piece of software that interacts with devices that the host operating system does not typically manage or that is physically distributed across multiple processors, then using deployment diagrams will help you reason about your system's software-to-hardware mapping.

When you model the static deployment view of a system, you'll typically use deployment diagrams in one of three ways.

1. To model embedded systems

An embedded system is a software-intensive collection of hardware that interfaces with the physical world. Embedded systems involve software that controls devices such as motors, actuators, and displays and that, in turn, is controlled by external stimuli such as sensor input, movement, and temperature changes. You can use deployment diagrams to model the devices and processors that comprise an embedded system.

2. To model client/server systems

A client/server system is a common architecture focused on making a clear separation of concerns between the system's user interface (which lives on the client) and the system's persistent data (which lives on the server). Client/ server systems are one end of the continuum of distributed systems and require you to make decisions about the network connectivity of clients

to servers and about the physical distribution of your system's software components across the nodes. You can model the topology of such systems by using deployment diagrams.

3. To model fully distributed systems

At the other end of the continuum of distributed systems are those that are widely, if not globally, distributed, typically encompassing multiple levels of servers. Such systems are often hosts to multiple versions of software components, some of which may even migrate from node to node. Crafting such systems requires you to make decisions that enable the continuous change in the system's topology. You can use deployment diagrams to visualize the system's current topology and distribution of components to reason about the impact of changes on that topology.

Common Modeling Techniques

Modeling an Embedded System

Developing an embedded system is far more than a software problem. You have to manage the physical world in which there are moving parts that break and in which signals are noisy and behavior is nonlinear. When you model such a system, you have to take into account its interface with the real world, and that means reasoning about unusual devices, as well as nodes.

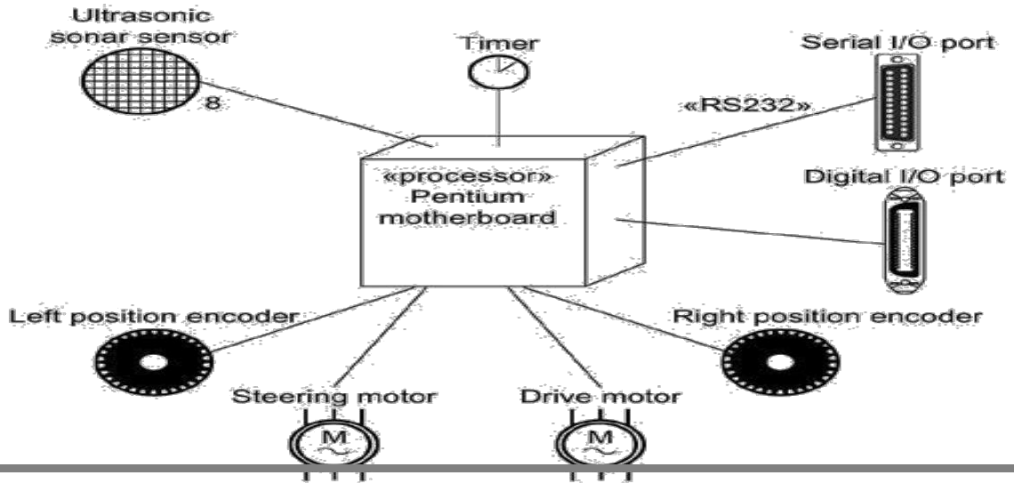
Deployment diagrams are useful in facilitating the communication between your project's hardware engineers and software developers. By using nodes that are stereotyped to look like familiar devices, you can create diagrams that are understandable by both groups. Deployment diagrams are also helpful in reasoning about hardware/software trade-offs. You'll use deployment diagrams to visualize, specify, construct, and document your system engineering decisions.

To model an embedded system,

- Identify the devices and nodes that are unique to your system.
- Provide visual cues, especially for unusual devices, by using the UML's extensibility mechanisms to define system-specific stereotypes with appropriate icons. At the very least, you'll want to distinguish processors (which contain software components) and devices (which, at that level of abstraction, don't directly contain software).
- Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.
- As necessary, expand on any intelligent devices by modeling their structure with a more detailed deployment diagram.

For example, Figure shows the hardware for a simple autonomous robot. You'll find one node (**Pentium motherboard**) stereotyped as a processor.

Figure Modeling an Embedded System



Surrounding this node are eight devices, each stereotyped as a device and rendered with an icon that offers a clear visual cue to its real-world equivalent.

Modeling a Client/Server System

The moment you start developing a system whose software no longer resides on a single processor, you are faced with a host of decisions: How do you best distribute your software components across these nodes? How do they communicate? How do you deal with failure and noise? At one end of the spectrum of distributed systems, you'll encounter client/server systems, in which there's a clear separation of concerns between the system's user interface (typically managed by the client) and its data (typically managed by the server).

There are many variations on this theme. For example, you might choose to have a thin client, meaning that it has a limited amount of computational capacity and does little more than manage the user interface and visualization of information. Thin clients may not even host a lot of components but, rather, may be designed to load components from the server, as needed, as with Enterprise Java Beans. On the other hand, you might chose to have a thick client, meaning that it has a goodly amount of computational capacity and does more than just visualization. A thick client typically carries out some of the system's logic and business rules. The choice between thin and thick clients is an architectural decision that's influenced by a number of technical, economic, and political factors.

Either way, partitioning a system into its client and server parts involves making some hard decisions about where to physically place its software components and how to impose a balanced distribution of responsibilities among those components. For example, most management information systems are essentially three-tier architectures, which means that the system's GUI, business logic, and database are physically distributed. Deciding where to place the system's GUI and database are usually fairly obvious, so the hard part lies in deciding where the business logic lives.

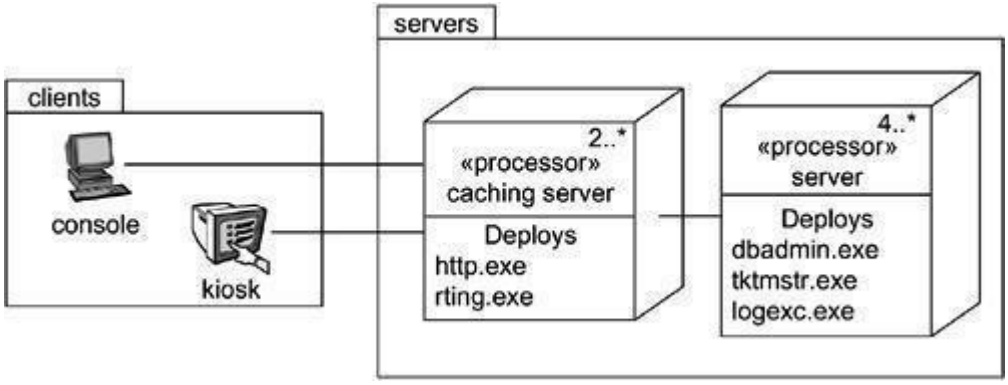
You can use the UML's deployment diagrams to visualize, specify, and document your decisions about the topology of your client/server system and how its software components are distributed across the client and server. Typically, you'll want to create one deployment diagram for the system as a whole, along with other, more detailed, diagrams that drill down to individual segments of the system.

To model a client/server system,

- Identify the nodes that represent your system's client and server processors.
- Highlight those devices that are germane to the behavior of your system. For example, you'll want to model special devices, such as credit card readers, badge readers, and display devices other than monitors, because their placement in the system's hardware topology are likely to be architecturally significant.
- Provide visual cues for these processors and devices via stereotyping.
- Model the topology of these nodes in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.

For example, Figure shows the topology of a human resources system, which follows a classical client/server architecture. This figure illustrates the client/server split explicitly by using the packages named **client** and **server.** The client package contains two nodes (**console** and **kiosk**), both of which are stereotyped and are visually distinguishable. The server package contains two kinds of nodes (**caching server** and **server**), and both of these have been adorned with some of the components that reside on each. Note also that **caching server** and **server** are marked with explicit multiplicities, specifying how many instances of each are expected in a particular deployed configuration. For example, this diagram indicates that there may be two or more **caching servers** in any deployed instance of the system.

Figure Modeling a Client/Server System



Modeling a Fully Distributed System

Distributed systems come in many forms, from simple two-processor systems to those that span many geographically dispersed nodes. The latter are typically never static. Nodes are added and removed as network traffic changes and processors fail; new and faster communication paths may be established in parallel with older, slower channels that are eventually decommissioned.

Not only may the topology of these systems change, but the distribution of their software components may change, as well. For example, database tables may be replicated across servers, only to be moved, as traffic dictates. For some global systems, components may follow the sun, migrating from server to server as the business day begins in one part of the world and ends in another.

Visualizing, specifying, and documenting the topology of fully distributed systems such as these are valuable activities for the systems administrator, who must keep tabs on an enterprise's computing assets. You can use the UML's deployment diagrams to reason about the topology of such systems. When you document fully distributed systems using deployment diagrams, you'll want to expand on the details of the system's networking devices, each of which you can represent as a stereotyped node.

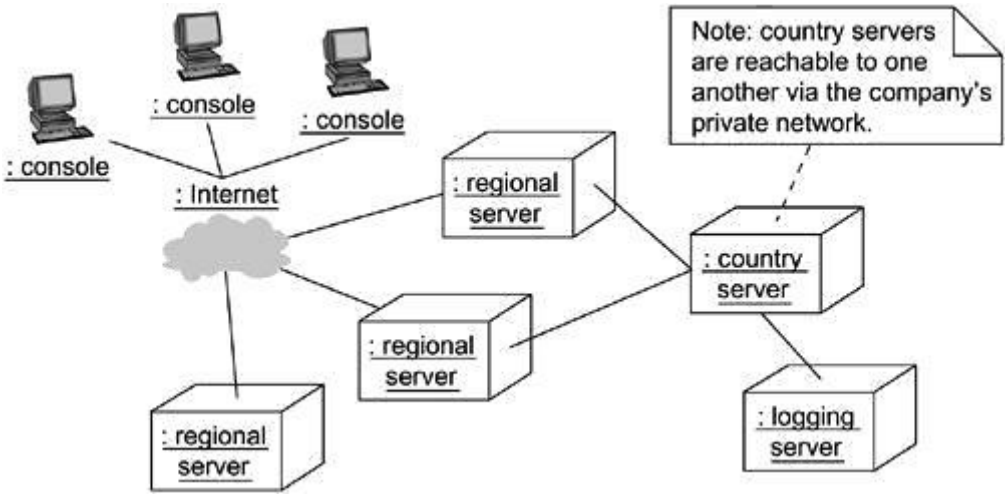
To model a fully distributed system,

- Identify the system's devices and processors as for simpler client/server systems.
- If you need to reason about the performance of the system's network or the impact of changes to the network, be sure to model these communication devices to the level of detail sufficient to make these assessments.
- Pay close attention to logical groupings of nodes, which you can specify by using packages.
- Model these devices and processors using deployment diagrams. Where possible, use tools that discover the topology of your system by walking your system's network.
- If you need to focus on the dynamics of your system, introduce use case diagrams to specify the kinds of behavior you are interested in, and expand on these use cases with interaction diagrams.

Figure shows the topology of a fully distributed system. This particular deployment diagram is also an object diagram, for it contains only instances. You can see three consoles (anonymous instances of the stereotyped node **console**), which are linked to the **Internet** (clearly a singleton node). In turn, there are three instances of **regional servers,** which serve as front ends of **country servers,** only one of which is shown. As the note indicates, country servers are connected to one another, but their relationships are not

shown in this diagram.

Figure Modeling a Fully Distributed System



In this diagram, the Internet has been reified as a stereotyped node.

Forward and Reverse Engineering

There's only a modest amount of forward engineering (the creation of code from models) that you can do with deployment diagrams. For example, after specifying the physical distribution of components across the nodes in a deployment diagram, it is possible to use tools that then push these components out to the real world. For system administrators, using the UML in this way helps you visualize what can be a very complicated task.

Reverse engineering (the creation of models from code) from the real world back to deployment diagrams is of tremendous value, especially for fully distributed systems that are under constant change. You'll want to supply a set of stereotyped nodes that speak the language of your system's network administrators, in order to tailor the UML to their domain. The advantage of using the UML is that it offers a standard language that addresses not only their needs, but the needs of your project's software developers, as well.

To reverse engineer a deployment diagram,

- Choose the target that you want to reverse engineer. In some cases, you'll want to sweep across your entire network; in others, you can limit your search.
- Choose also the fidelity of your reverse engineering. In some cases, it's sufficient to reverse engineer just to the level of all the system's processors; in others, you'll want to reverse engineer the system's networking peripherals, as well.
- Use a tool that walks across your system, discovering its hardware topology. Record that topology in a deployment model.
- Along the way, you can use similar tools to discover the components that live on each node, which you can also record in a deployment model. You'll want to use an intelligent search, for even a basic personal computer can contain gigabytes of components, many of which may not be relevant to your system.
- Using your modeling tools, create a deployment diagram by querying the model. For example,

you might start with visualizing the basic client/server topology, then expand on the diagram by populating certain nodes with components of interest that live on them. Expose or hide the details of the contents of this deployment diagram as necessary to communicate your intent.