Education Solutions

TOMORROW'S HERE

Subject: Object Oriented Analysis and Design

Module Number- 4: Basic Behavioural Modelling-II

SME NAME: ENTER NAME
SUBMISSION DATE: MENTION DATE
VERSION CODE: TO BE FILLED AT HO
RELEASED DATE: TO BE FILLED AT HO

Basic Behavioral Modeling- II

Events and Signals

- 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.

Basic Behavioral Modeling- II

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**.

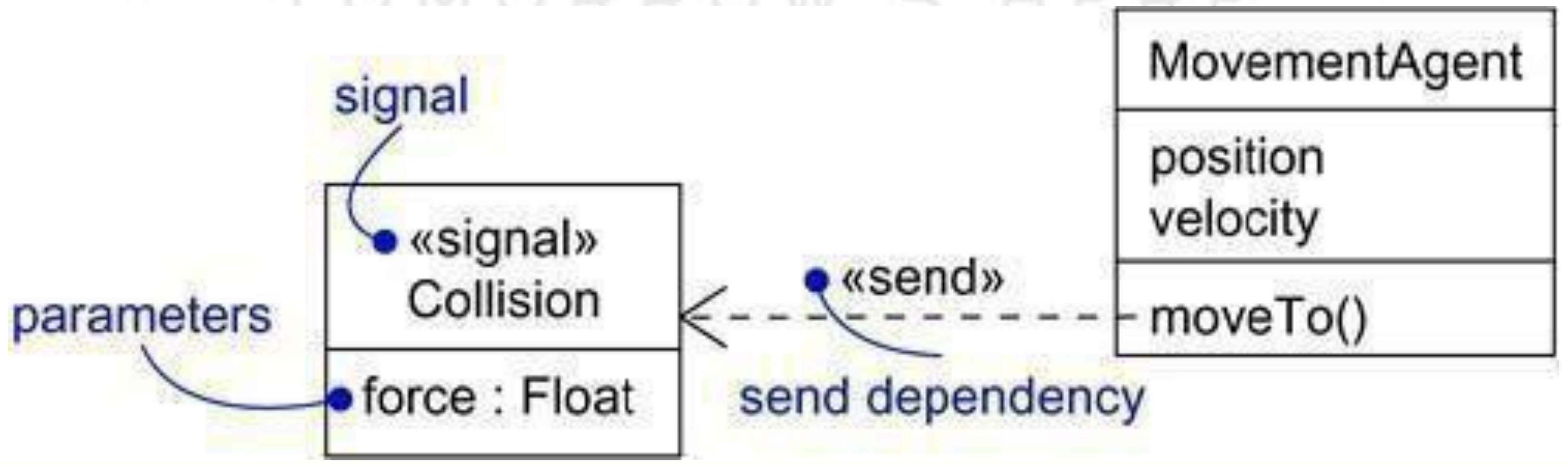
Basic Behavioral Modeling- II

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.

Basic Behavioral Modeling- II

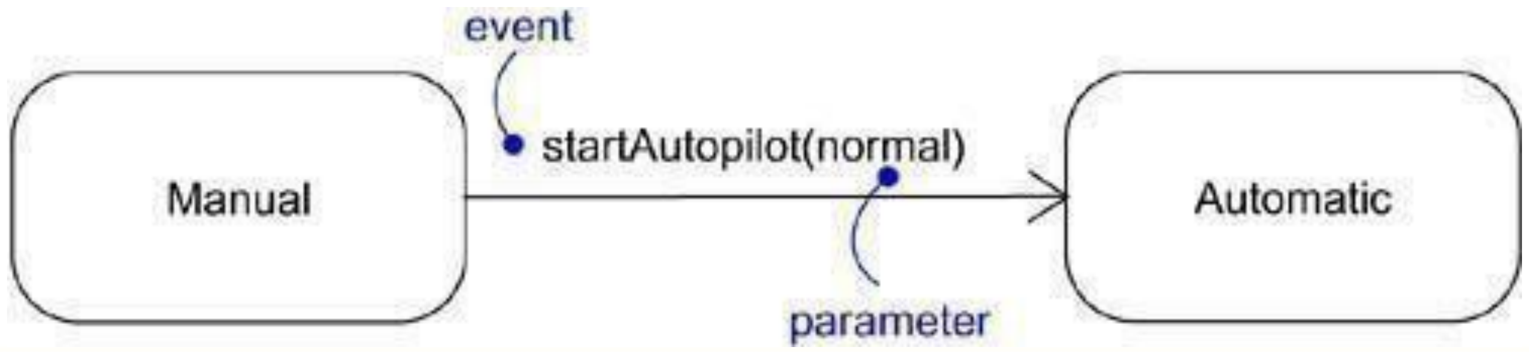
- A signal may be sent as the action of a state transition in a state machine or the sending of a message in an interaction.
- The execution of an operation can also send signals. In fact, when you model a class or an interface, an important part of specifying the behavior of that element is specifying 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.



Basic Behavioral Modeling- II

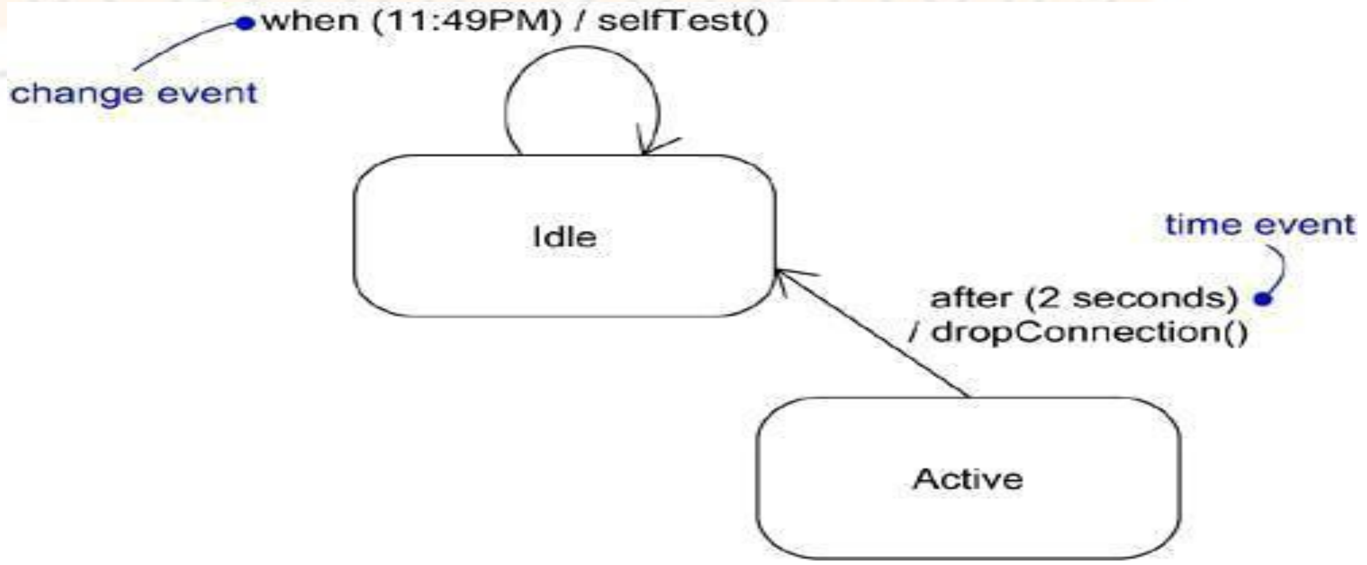
Call Events

- 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.



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.



Basic Behavioral Modeling- II

- 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 absolute time (such as **when time = 11:59**) or for the continuous test of an expression (for example, **when altitude < 1000**).

Basic Behavioral Modeling- II

Sending and Receiving Events

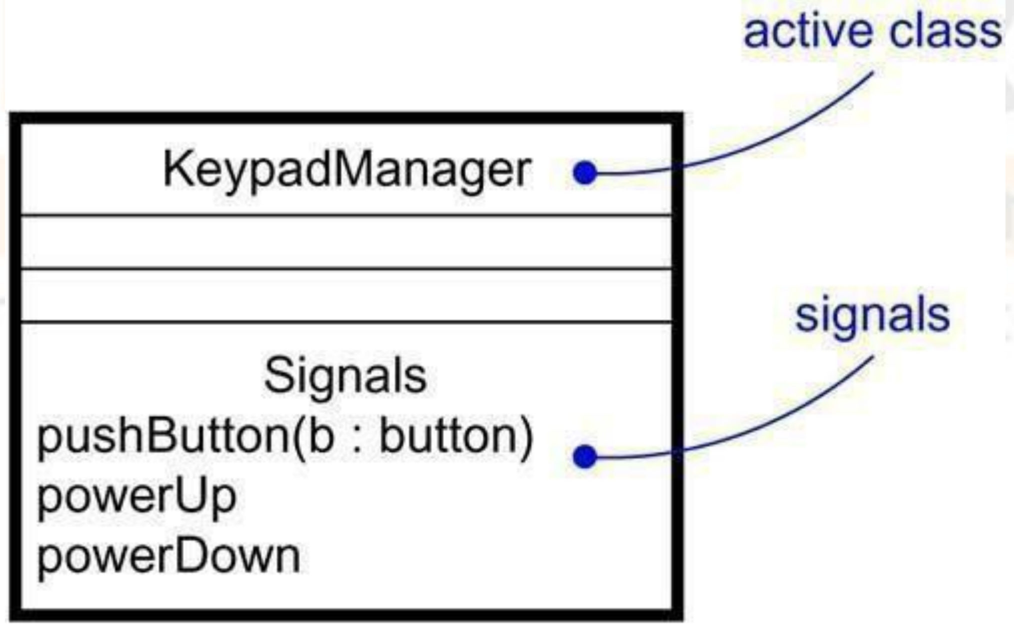
- Signal events and call events involve at least two objects: the object that sends the signal or invokes the operation, and the object to which the event is directed.
- Because signals are asynchronous, and because asynchronous calls are themselves signals, the semantics of events interact with the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

Basic Behavioral Modeling- II

- 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.

Basic Behavioral Modeling- II

- 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 .



Basic Behavioral Modeling- II

Common Modeling Techniques

Modeling a Family of Signals

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

Basic Behavioral Modeling- II

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

Basic Behavioral Modeling- II

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.

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

Basic Behavioral Modeling- II

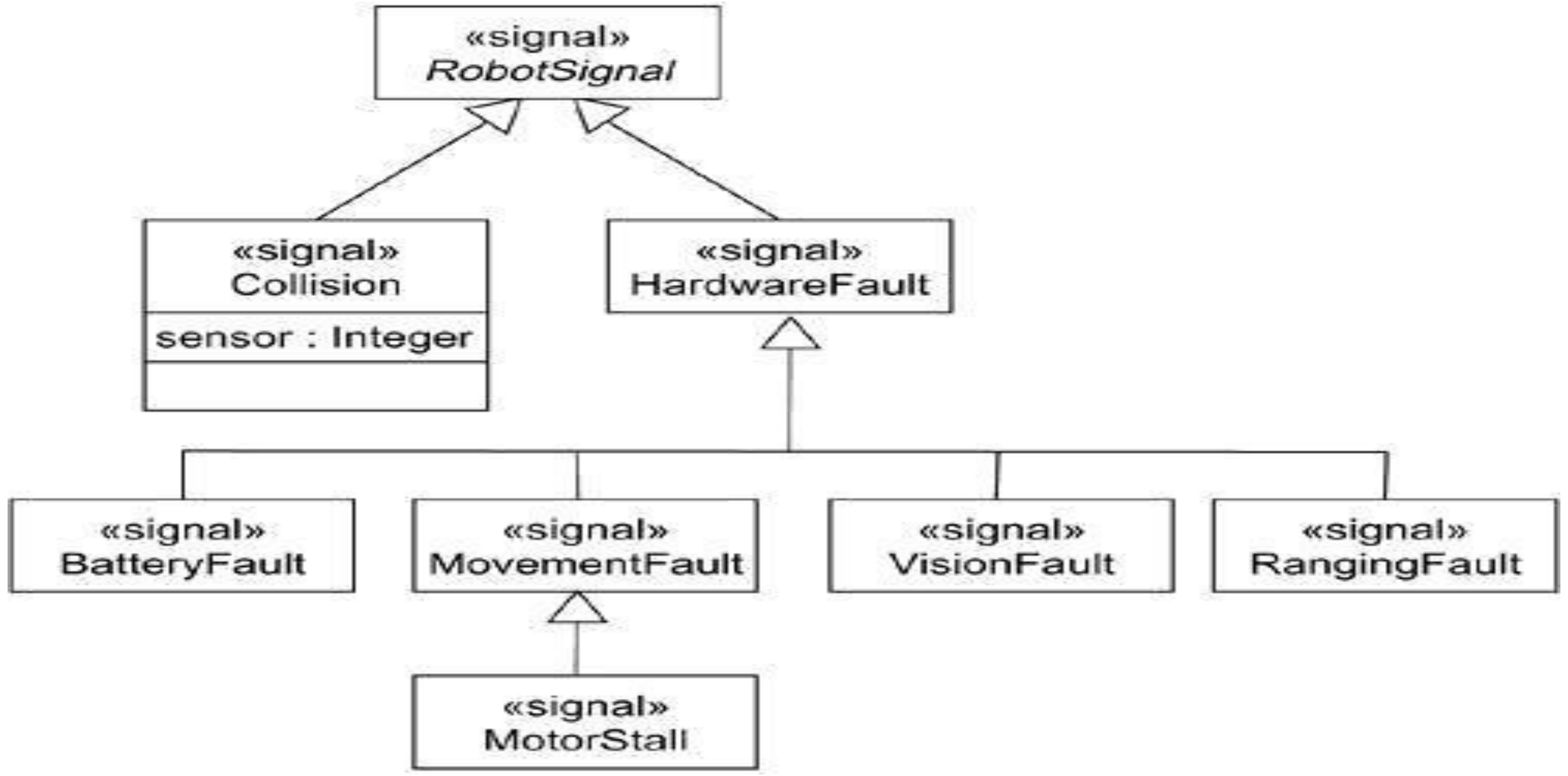


Figure: Modeling Families of Signals

Basic Behavioral Modeling- II

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.

Basic Behavioral Modeling- II

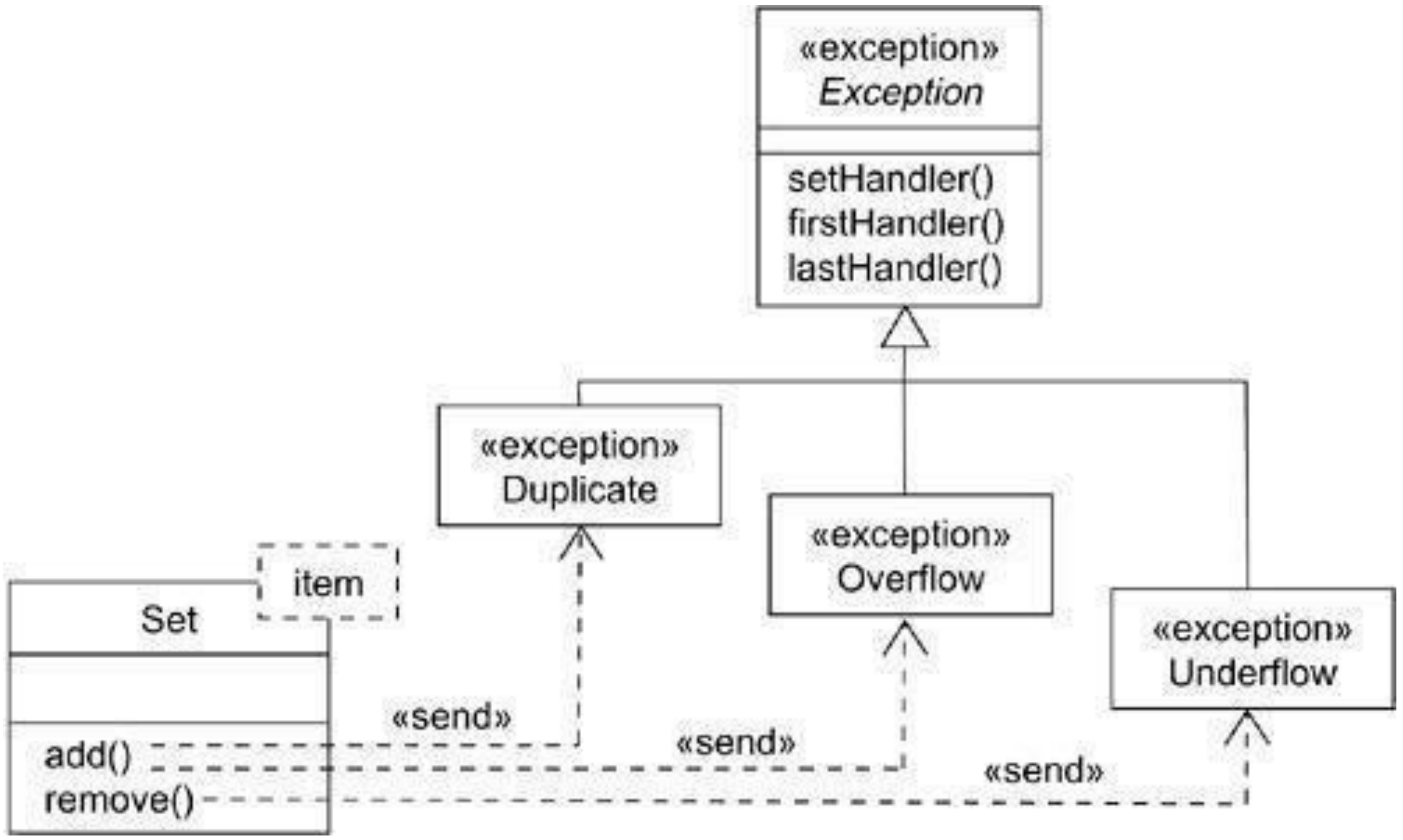
To model exceptions,

- 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.

Basic Behavioral Modeling- II

- Following Figure models a hierarchy of exceptions that may be raised by a standard library of container 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, and the 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.

Basic Behavioral Modeling- II



Basic Behavioral Modeling- II

State Machines

- **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.

Basic Behavioral Modeling- II

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.

Basic Behavioral Modeling- II

- 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).

Basic Behavioral Modeling- II

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**.

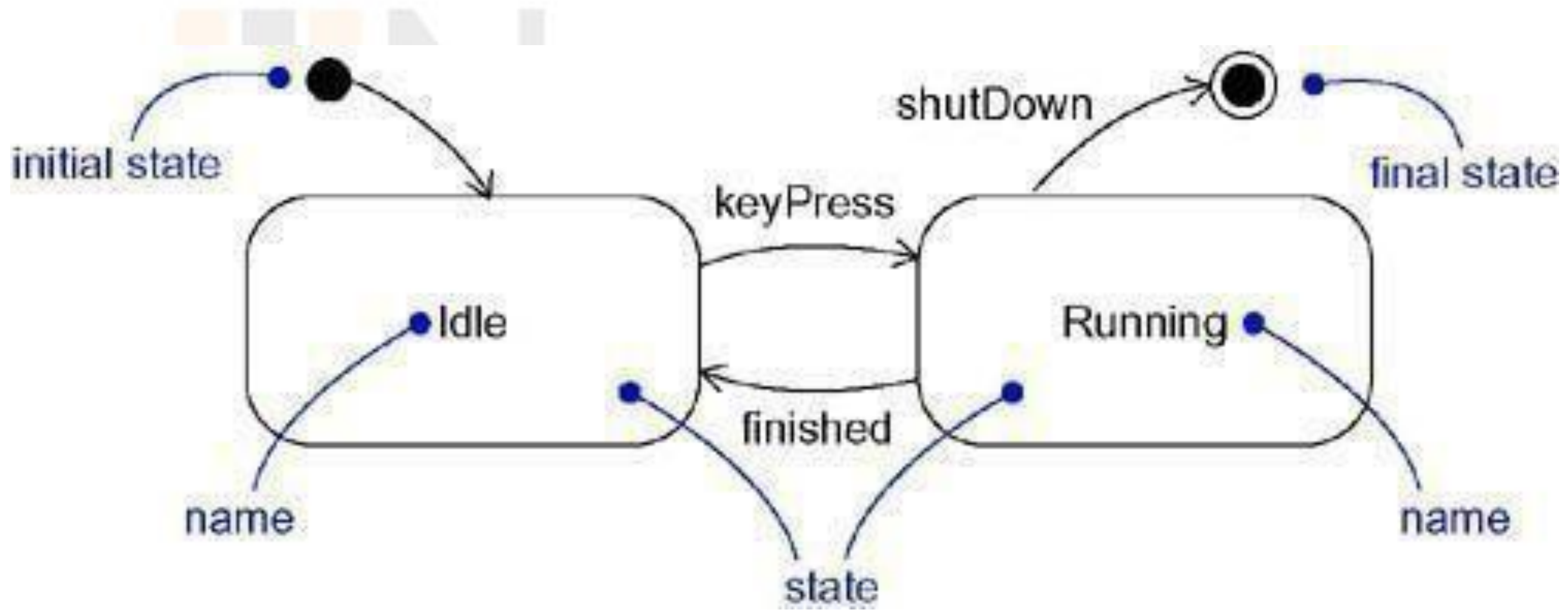
Basic Behavioral Modeling- II

A state has several parts.

1.	**Name**	A textual string that distinguishes the state from other states; a state may be anonymous, meaning that it has no name
2.	**Entry/exit 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

Basic Behavioral Modeling- II

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



Basic Behavioral Modeling- II

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).

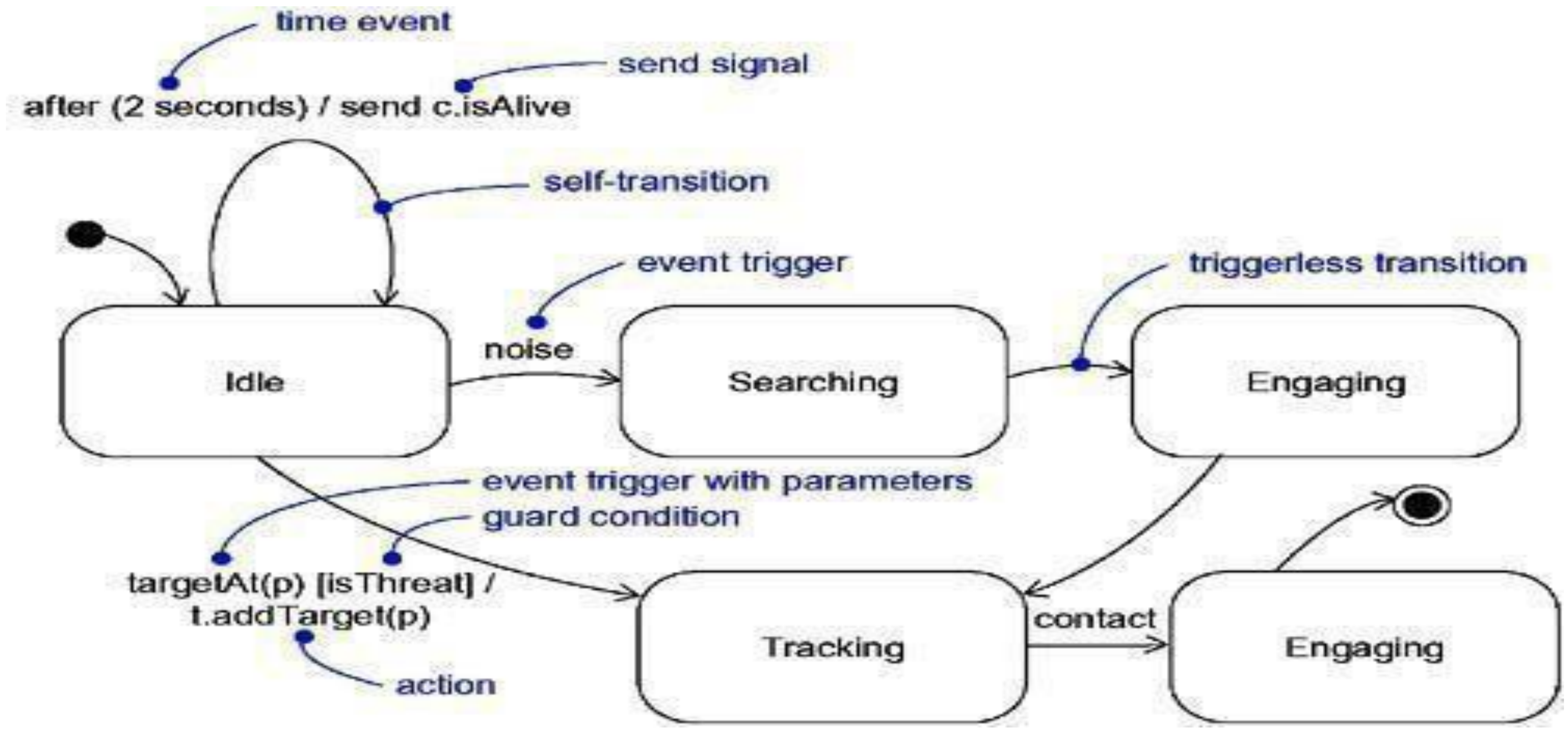
Basic Behavioral Modeling- II

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 the following 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.

Basic Behavioral Modeling- II



Basic Behavioral Modeling- II

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.

Basic Behavioral Modeling- II

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).

Basic Behavioral Modeling- II

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.

- 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.

Basic Behavioral Modeling- II

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.

Basic Behavioral Modeling- II

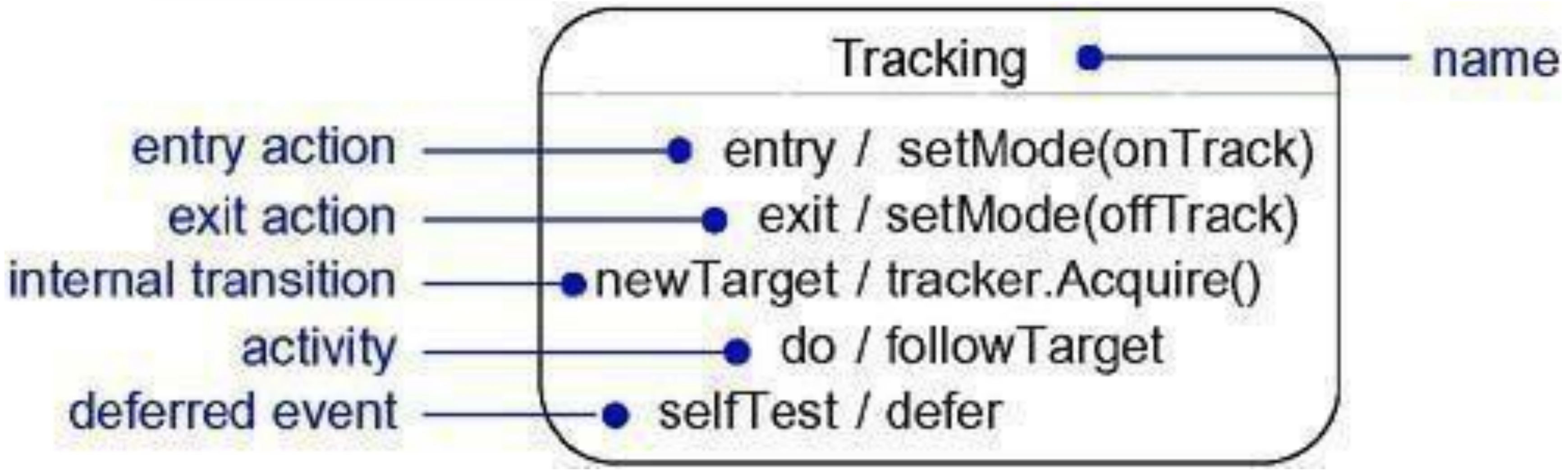
Entry and Exit Actions

- 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.

Basic Behavioral Modeling- II

- 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



Basic Behavioral Modeling- II

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 the above 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.

Basic Behavioral Modeling- II

- As the 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.

Basic Behavioral Modeling- II

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.

Basic Behavioral Modeling- II

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 maintenance agent in the system.

Basic Behavioral Modeling- II

- 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 it appears as if they just occurred.

Basic Behavioral Modeling- II

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 all graphic compartment that shows a nested state machine. Substates may be nested to any level.

Basic Behavioral Modeling- II

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 and process multiple transactions after **Validating** the account and before **Printing** a final receipt.

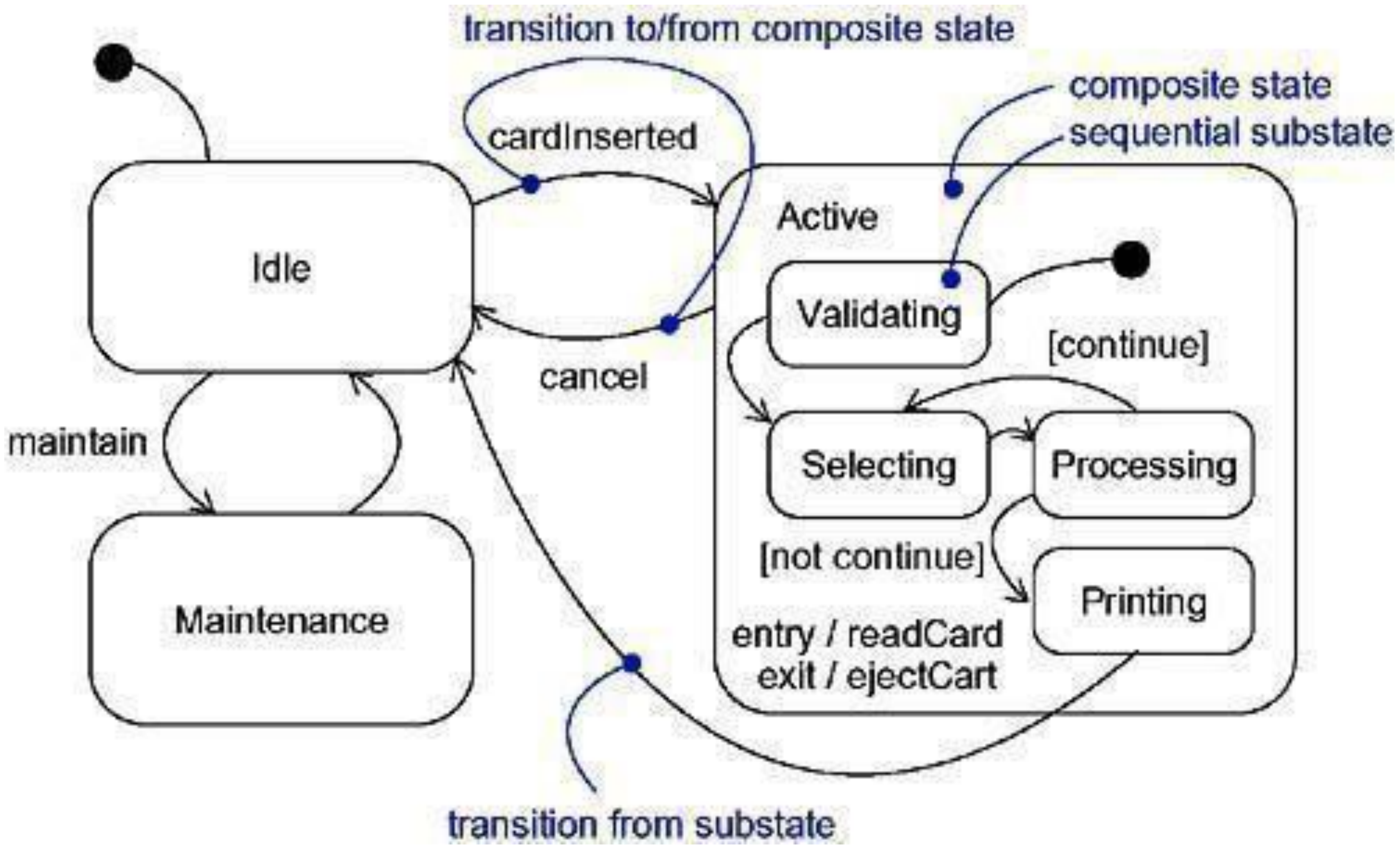
Basic Behavioral Modeling- II

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

Basic Behavioral Modeling- II

- 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.

Basic Behavioral Modeling- II



Basic Behavioral Modeling- II

History States

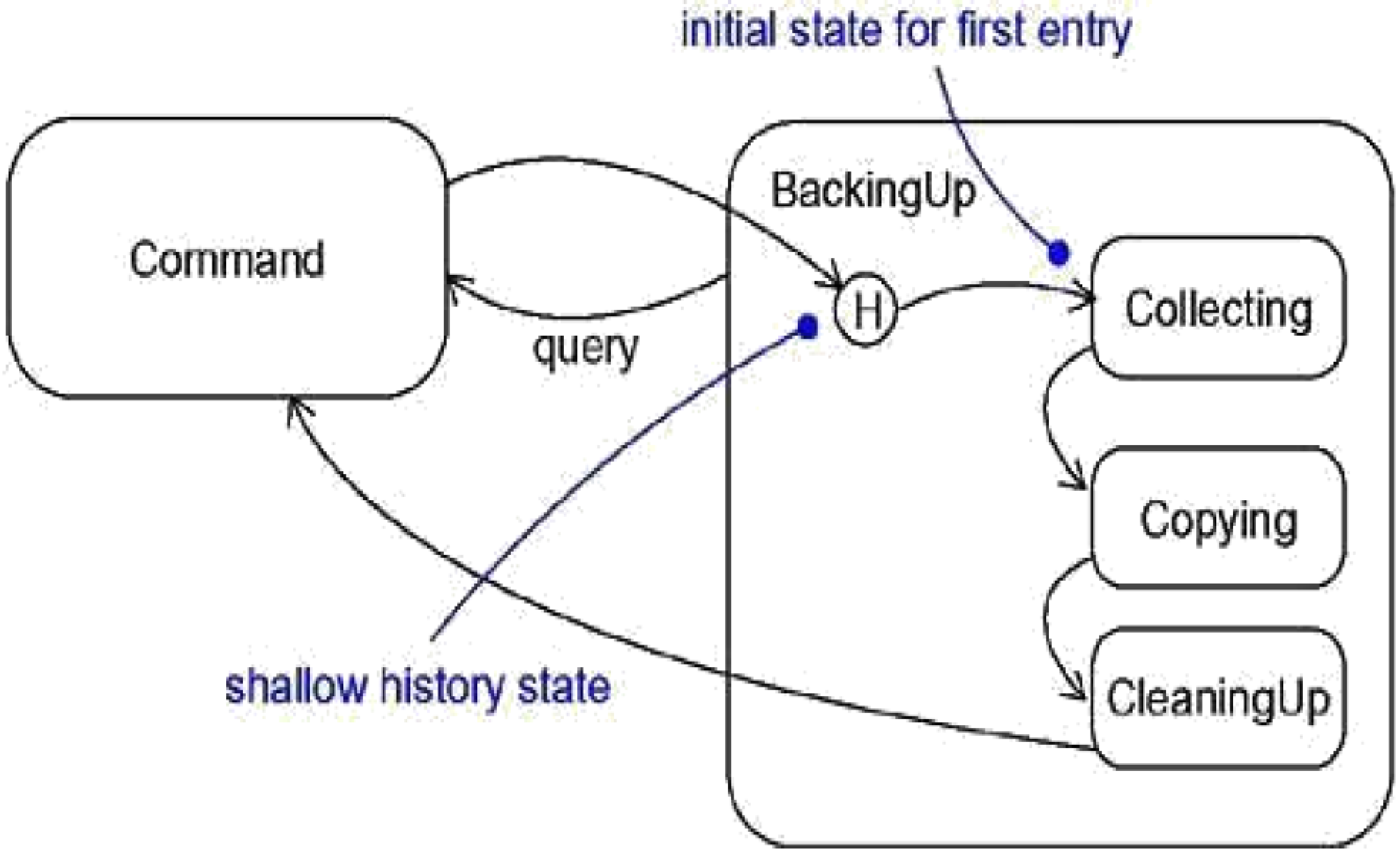
- A state machine describes the dynamic aspects of an object whose current behavior depends on its past. A state machine in effect specifies the legal ordering of states an object may go through during its lifetime.
- Unless otherwise specified, when a transition enters a composite state, the action 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.

Basic Behavioral Modeling- II

- 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 a shallow history state as a small circle containing the symbol **H**.

Basic Behavioral Modeling- II

Figure History State



Basic Behavioral Modeling- II

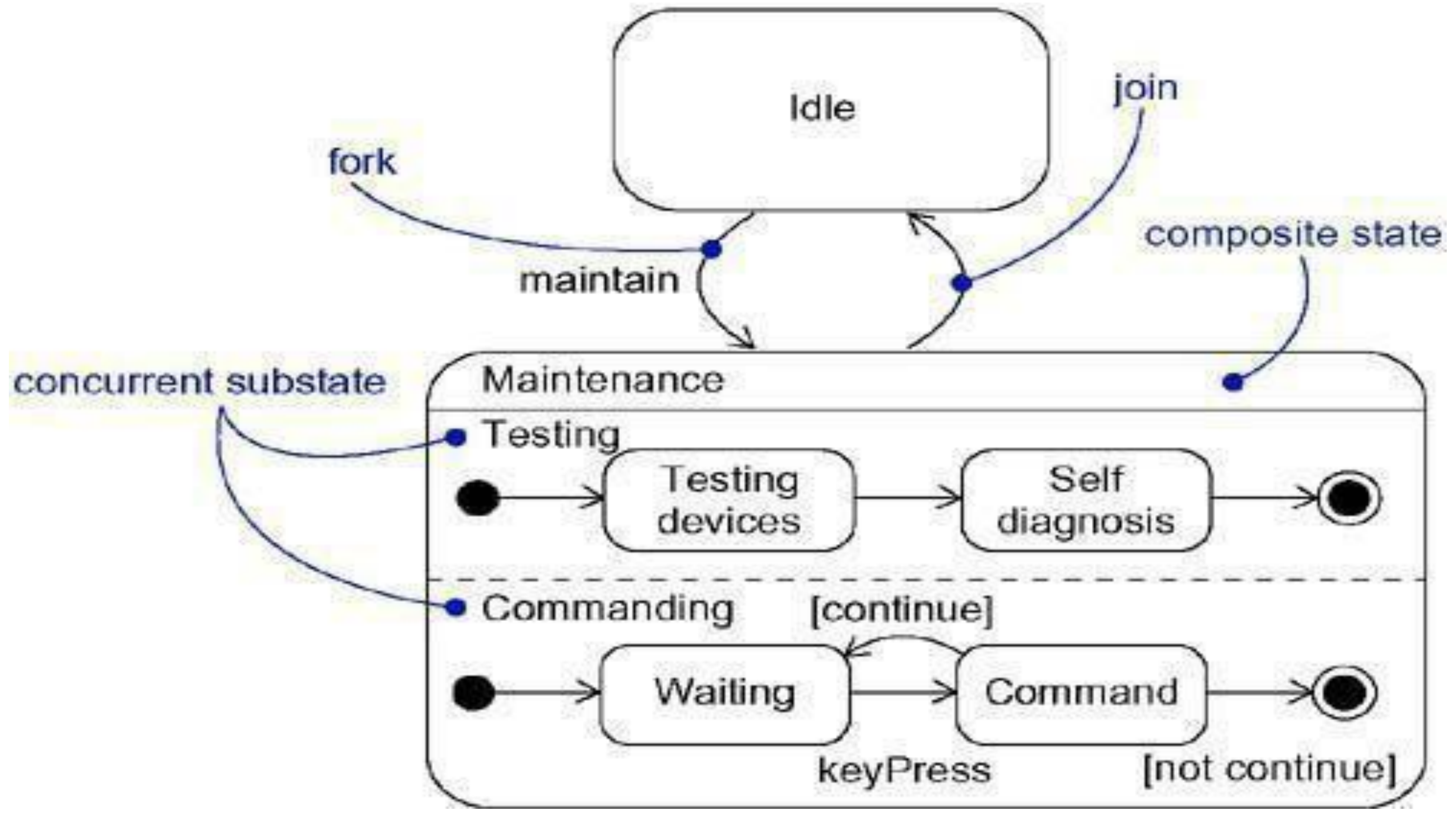
- If you want a transition to activate the last substate, you show a transition from outside the composite state directly to the history state. The first time you enter a composite state, it has no history. This is the meaning of the single transition from the history state to a sequential substate such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

Basic Behavioral Modeling- II

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.

Basic Behavioral Modeling- II



Basic Behavioral Modeling- II

Processes and thread

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

Basic Behavioral Modeling- II

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.

Basic Behavioral Modeling- II

- 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.

Basic Behavioral Modeling- II

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 heavyweight flow that can execute concurrently with other processes **process**
    2. **thread** Specifies a lightweight flow that can execute concurrently with other threads within the 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.

Basic Behavioral Modeling- II

Communication

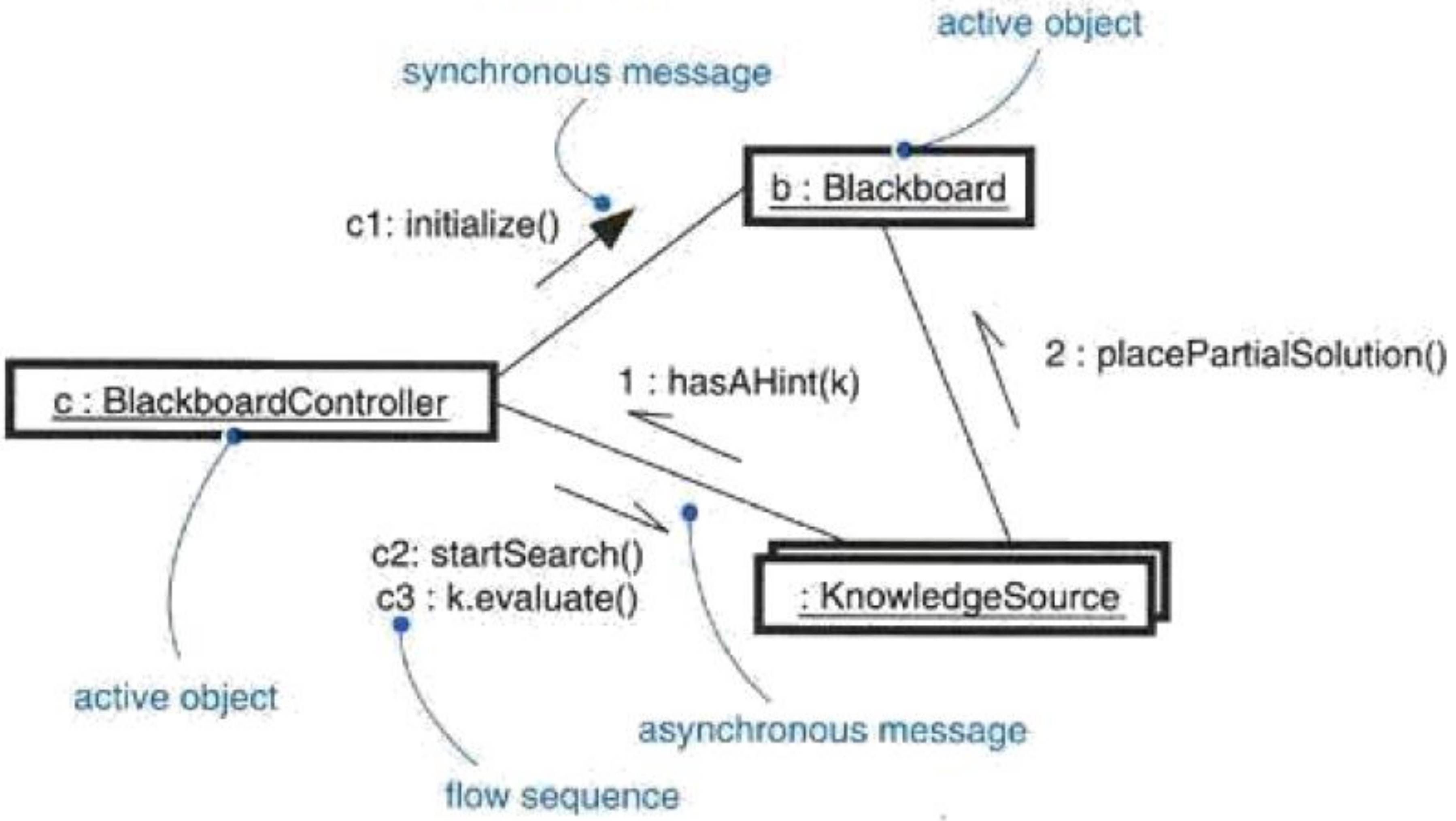
- When objects collaborate with one another, they interact by passing messages from one to the other. In a system with both **active** and **passive objects**, there are **four possible combinations** of interaction that you must consider.

  - First, a message may be passed from one passive object to another.
  - Second, a message may be passed from one active object to another.
  - Third, a message may be passed from an active object to a passive object.
  - Fourth, a message may be passed from a passive object to an active one.

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

Basic Behavioral Modeling- II

Figure Communication



Basic Behavioral Modeling- II

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.

Basic Behavioral Modeling- II

- 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**.

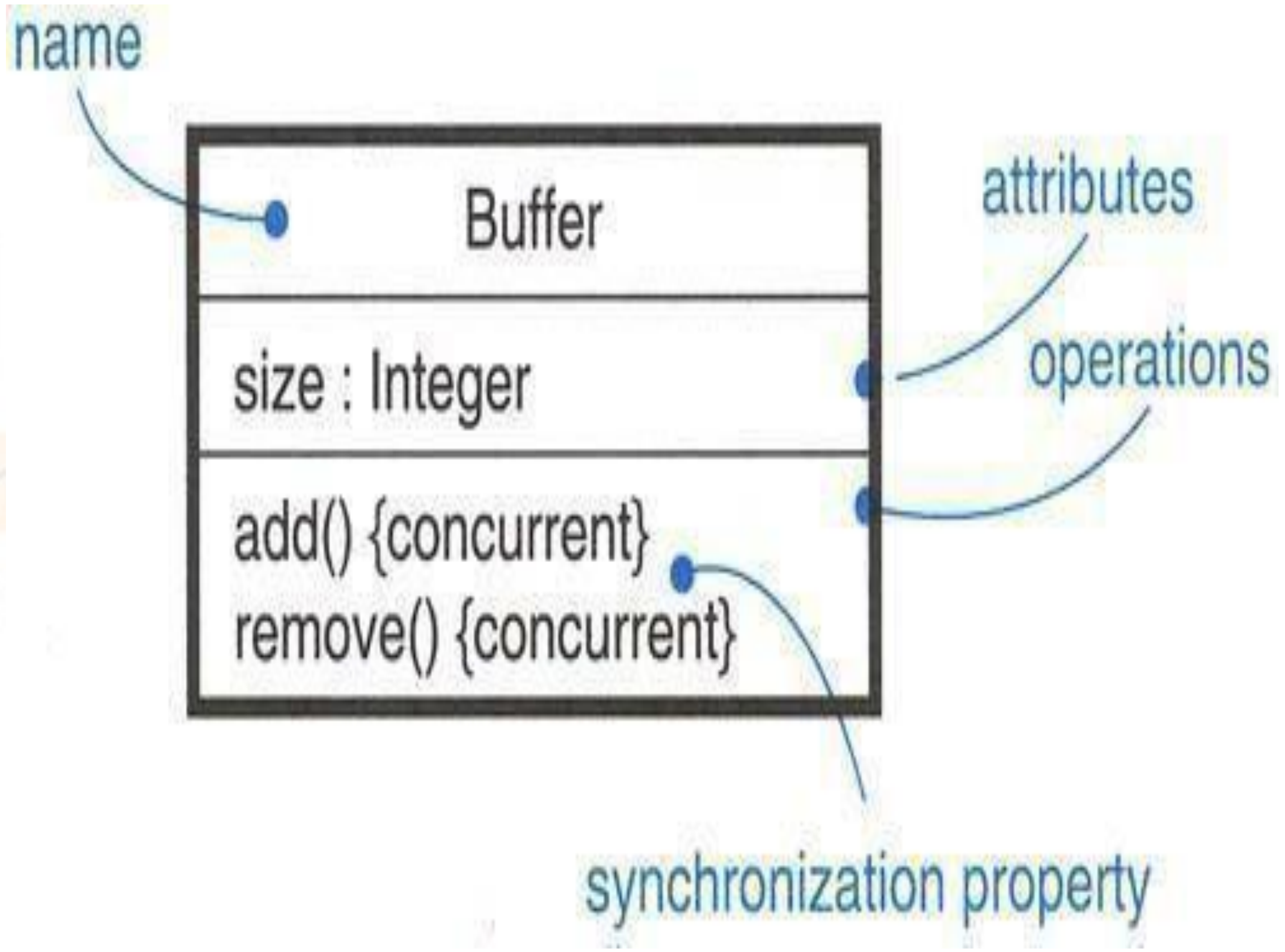
1. Sequential Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semantics and integrity of the object cannot be guaranteed.

2. Guarded The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by sequential zing 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.

Basic Behavioral Modeling- II

- Some programming languages support these constructs directly. Java, for example, has the synchronized property, which is equivalent to the UML's concurrent property. In every language that supports concurrency, you can build support for all these properties by constructing them out of semaphores.
- As Figure shows, you can attach these properties to an operation, which you can render in the UML by using constraint notation.



Basic Behavioral Modeling- II

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 state chart diagrams, but with a focus on the active classes that represent these threads and processes.

Basic Behavioral Modeling- II

Time and Space

- 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 brackets 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.

Basic Behavioral Modeling- II

Time

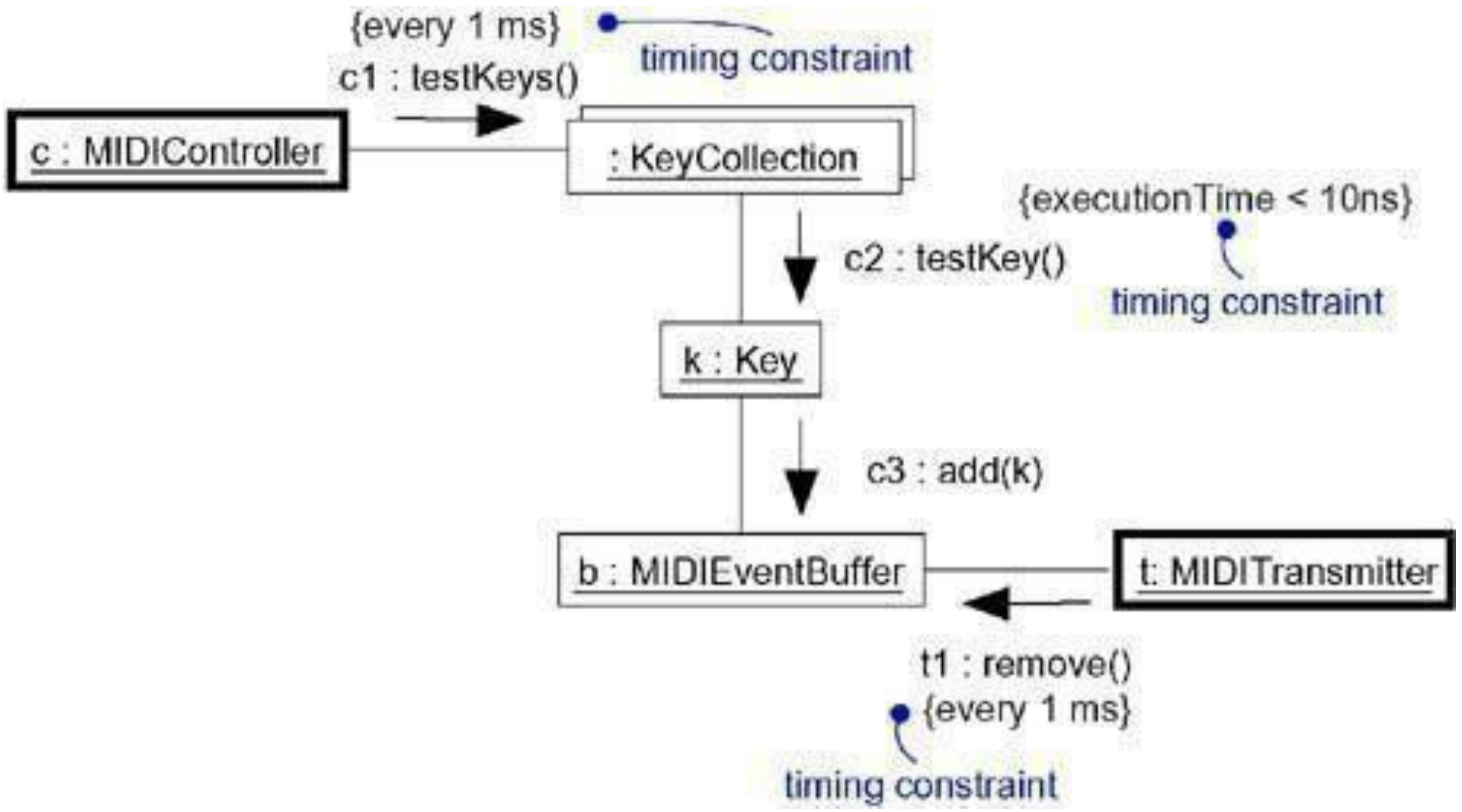
- Real time systems are, by their very name, time-critical systems. Events may happen at regular or irregular times; the response to an event must happen at predictable absolute times or at predictable times relative to the event itself.
- The passing of messages represents the dynamic aspect of any 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.

Basic Behavioral Modeling- II

- 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 arbitrarily complex time expressions, perhaps even using weights or offsets that are either constants or variables (as long as those variables can be bound at execution time). Finally, as shown in Figure , you can place these time expressions in a timing constraint to specify the timing behavior of the system. As constraints, you can render them by placing them adjacent to the appropriate message, or you can explicitly attach them using dependency relationships.

Basic Behavioral Modeling- II

Figure Time



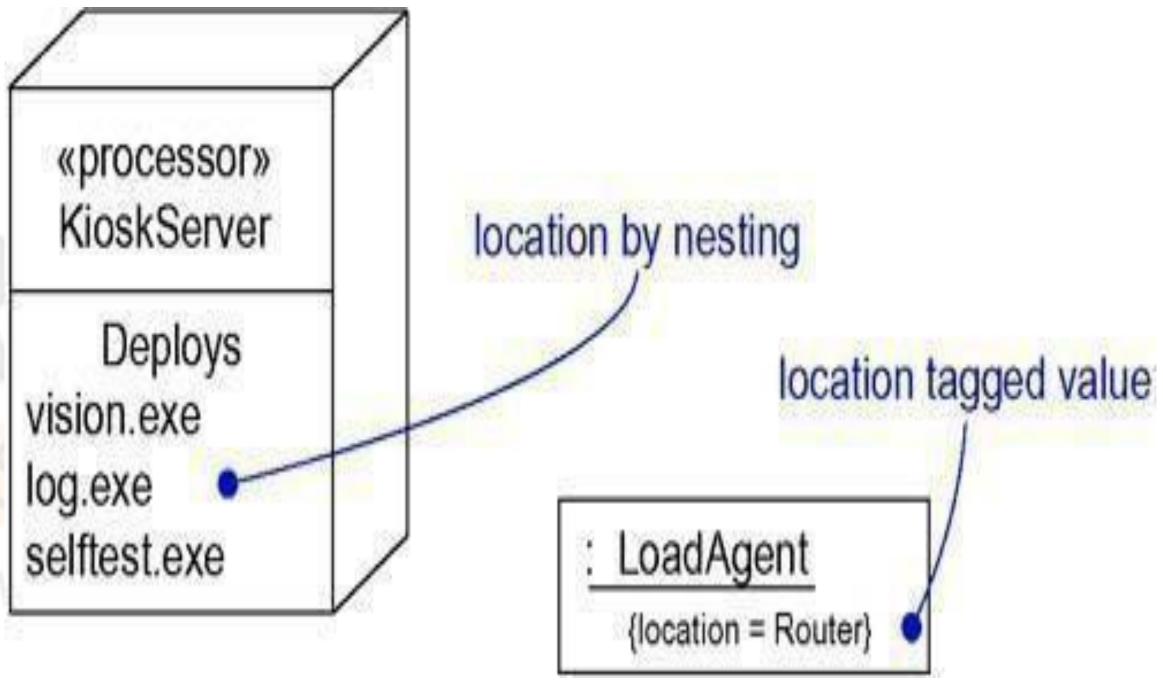
Basic Behavioral Modeling- II

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 **KioskServer**.

Basic Behavioral Modeling- II

- 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.



Basic Behavioral Modeling- II

Statechart Diagrams

- 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.

Basic Behavioral Modeling- II

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 name 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.

Basic Behavioral Modeling- II

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

Basic Behavioral Modeling- II

- 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.

Basic Behavioral Modeling- II

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. Whereas interactions 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.

Basic Behavioral Modeling- II

- 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.

Basic Behavioral Modeling- II

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 post-conditions 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.

NURTURE Education Solutions TOMORROW'S HERE

Basic Behavioral Modeling- II

- 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.

Basic Behavioral Modeling- II

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

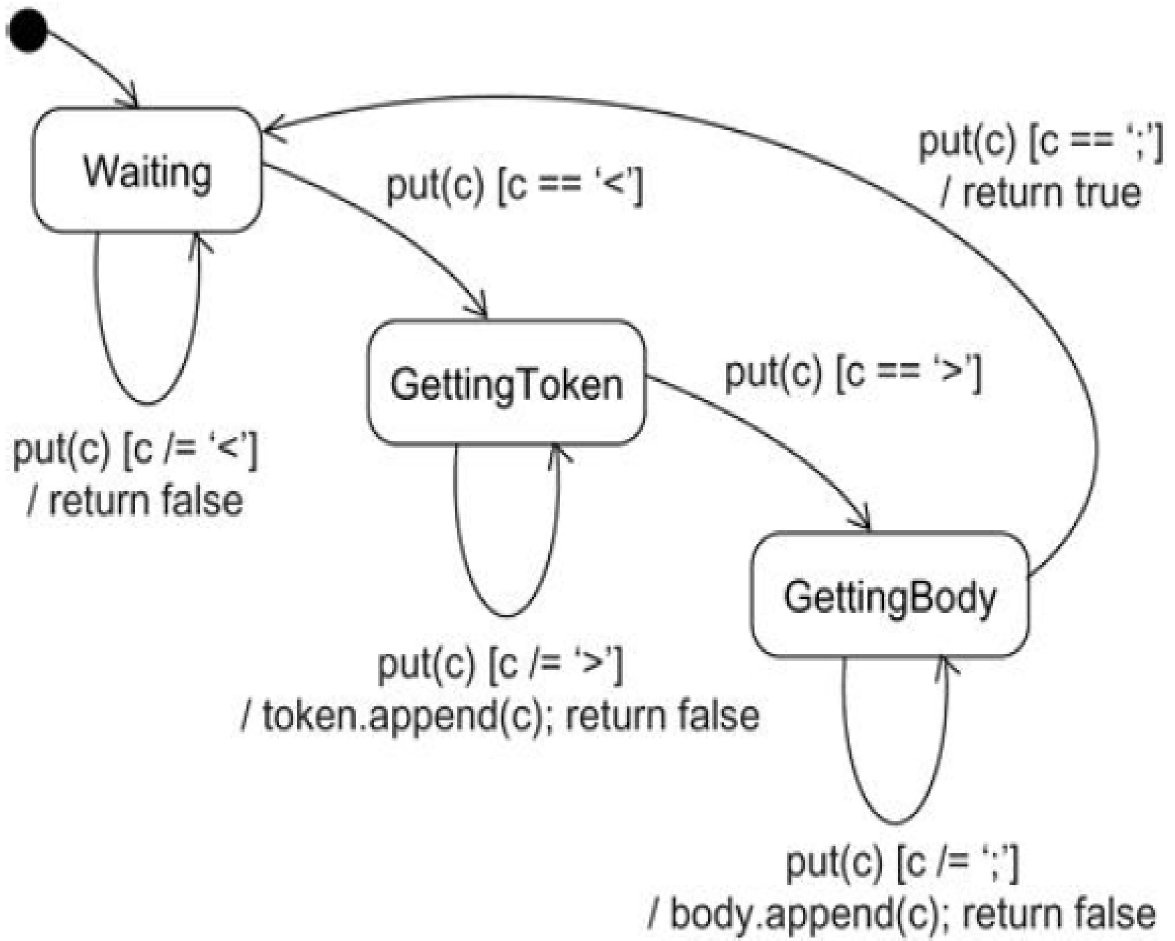


Figure: Modelling Reactive Objects

Basic Behavioral Modeling- II

- 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, with actions tied to transitions.
- In fact, there is only one event of interest in this state machine, the invocation of **put** with the actual parameter **c** (a character).
- While **Waiting**, this machine throws away any character that does not designate the start of a token (as specified by the guard condition).
- When the start of a token is received, the state of the object changes to **GettingToken**.
- While in that state, the machine saves any character that does not designate the end of a token (as specified by the guard condition).

Basic Behavioral Modeling- II

- 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.

Subject (Helvetica Bold- 24pt)

Self Assessment Question

1. Question (Times New Roman-20pt- Line spacing 1.5pt)

   a. Options

   b. Options

   c. Options

   d. Options

   Answer: Options

Subject (Helvetica Bold- 24pt)

Document Link

Topic	URL	Notes
Times New Roman-14PT	Times New Roman-14PT	Times New Roman-14PT
Times New Roman-14PT	Times New Roman-14PT	Times New Roman-14PT
Times New Roman-14PT	Times New Roman-14PT	Times New Roman-14PT

iNURTURE Education Solutions TOMORROW'S HERE

Subject (Helvetica Bold- 24pt)

Video Link

Topic	URL	Notes
Times New Roman-14PT	Times New Roman-14PT	Times New Roman-14PT
Times New Roman-14PT	Times New Roman-14PT	Times New Roman-14PT
Times New Roman-14PT	Times New Roman-14PT	Times New Roman-14PT

Subject (Helvetica Bold- 24pt)

E- Book Link

Ebook name	Chapter	Page No.	Notes	URL
Times New Roman-14PT	Times New Roman-14PT	Times New Roman-14PT	Times New Roman-14PT	Times New Roman-14PT
Times New Roman-14PT	Times New Roman-14PT	Times New Roman-14PT	Times New Roman-14PT	Times New Roman-14PT