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

1.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.

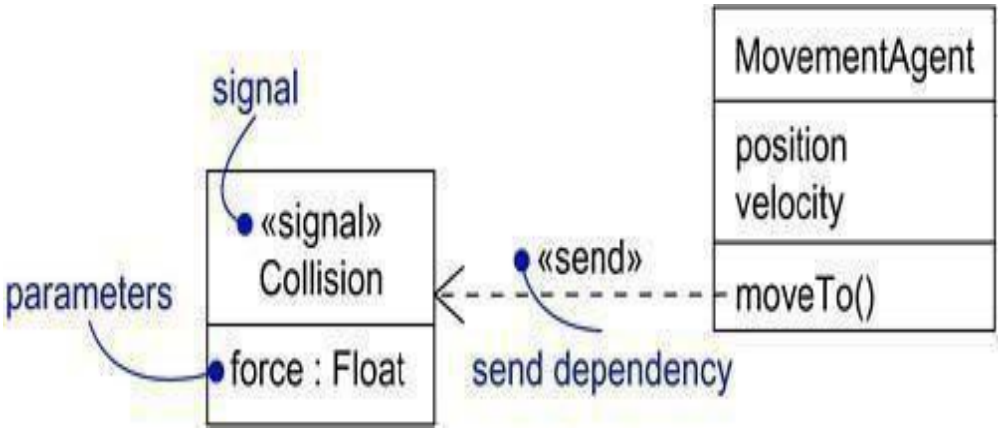
In the UML, you can model four kinds of events: signals, calls, the passing of time, and a change in state.

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 may also be involved in generalization relationships, permitting you to model hierarchies of events, some of which are general and some of which are specific
- 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.

- We model signals (and exceptions) as stereotyped classes. We can use a dependency, stereotyped as send, to indicate that an operation sends a particular signal

Figure Signals

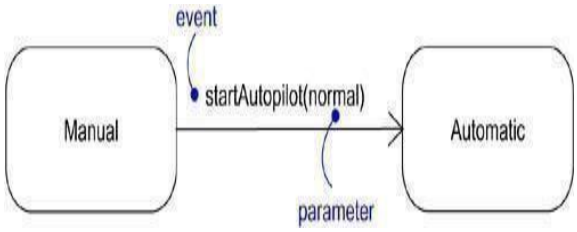

MovementAgent
position
velocity
moveTo()
signal
«signal»
Collision
force : Float
parameters
«send»
send dependency

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

.

Figure Call Events



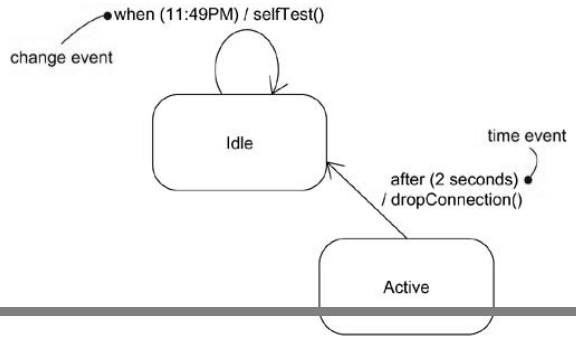
Time and Change Events

Call Events

Time and Change Events

- **A time event** is an event that represents the passage of time
- in the UML you model a time event by using the keyword after followed by some expression that evaluates to a period of time.
- Unless you specify it explicitly, the starting time of such an expression is the time since entering the current state.

- **A change event** is an event that represents a change in state or the satisfaction of some condition
- 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



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

Sending and Receiving Events

- Signal events and call events involve at least two objects:
    - The object that sends the signal or invokes the operation
    - The object to which the event is directed.
- 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.
- 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(assignation) 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
- In the UML, you model the named signals that an object may receive by naming them in an extra compartment of the class

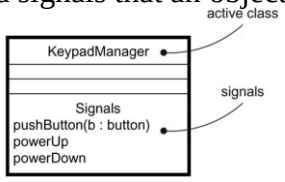
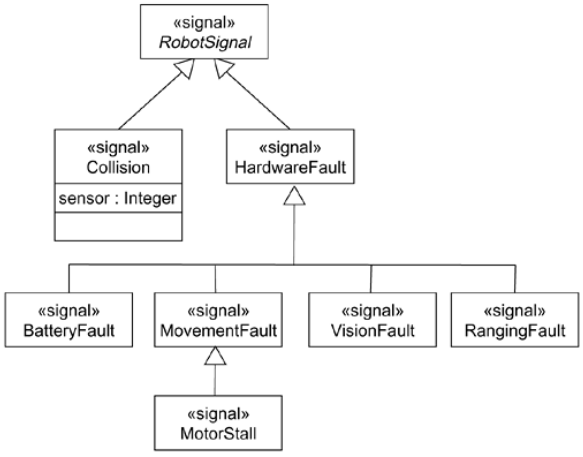


Fig:Signals and Active Classes

Common Modeling Techniques

Modeling a Family of Signals

- In most event-driven systems, signal events are hierarchical.
- External and internal signals need not be disjoint, however. Even within these two broad classifications, you might find specializations
- 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.



Modeling Abnormal Occurrence(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.
- 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.
- We model a family of signals primarily to specify the kinds of signals an active object may receive
- We model exceptions primarily to specify the kinds of exceptions that an object may throw through its operations

To model Abnormal Occurrence(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.

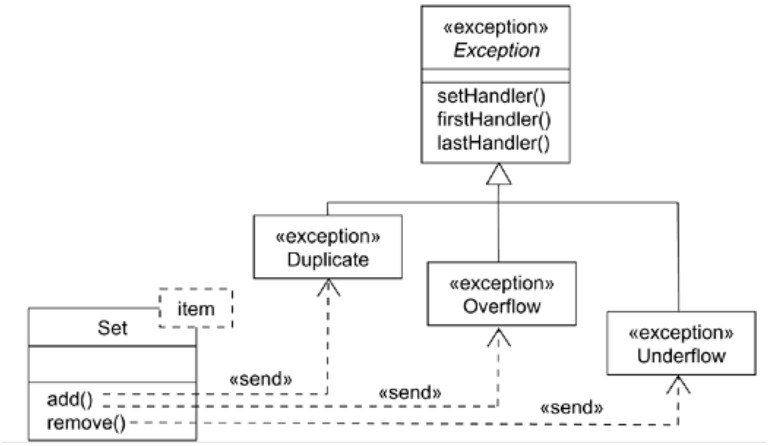


Fig: Modeling Exceptions

2. State Machines

- **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

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 other kinds of systems, you'll encounter objects that must respond to signals, which are asynchronous stimuli communicated between instances.
- 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.
- We'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.

- When an object's state machine is in a given state, the object is said to be in that state. 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

We represent a state as a rectangle with rounded corners.

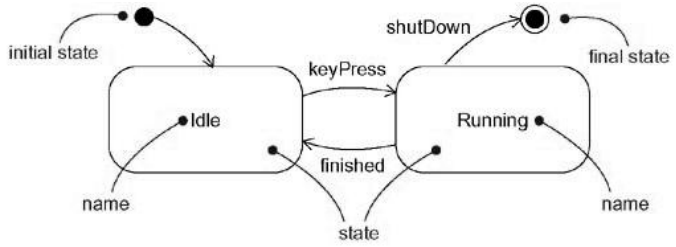


Fig: **States**

Initial and Final States

- There are two special states that may be defined for an object's state machine.
- **initial state**, which indicates the default starting place for the state machine or substate.
- An initial state is represented as a filled black circle
- **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.

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.
- A transition has five parts

1. Source state	The state affected by the transition; if an object is in the source state, an outgoing transition may fire when the object receives the trigger event of the transition and if the guard condition, if any, is satisfied

2. Event trigger	The event whose reception by the object in the source state makes the transition eligible to fire, providing its guard condition is satisfied
3. Guard condition	A Boolean expression that is evaluated when the transition is triggered by the reception of the event trigger; if the expression evaluates True, the transition is eligible to fire; if the expression evaluates False, the transition does not fire and if there is no other transition that could be triggered by that same event, the event is lost
4. Action	An executable atomic computation that may directly act on the object that owns the state machine, and indirectly on other objects that are visible to the object
5. Target state	The state that is active after the completion of the transition

- 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.

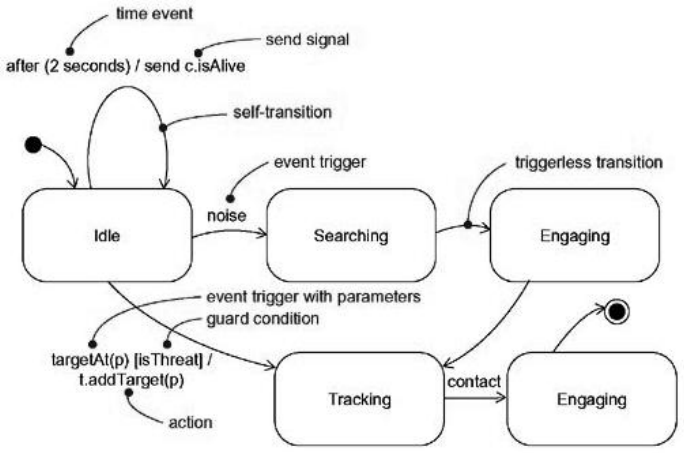


Fig:Transitions

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.
- 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

- 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

Action

- An action is an executable atomic computation. Actions may include operation calls), the creation or destruction of another object, or the sending of a signal to an object.
- An action is atomic, meaning that it cannot be interrupted by an event and therefore runs to completion.

Advanced States and Transitions

- 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.
- 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.

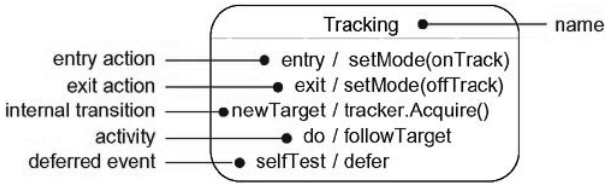


Fig:Advanced States and Transitions

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 an event triggers the transition, you leave the state, an action (if any) is dispatched, and then you reenter the same state.
- 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.
- 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
- 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

- 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.

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.

Substates

- 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 simple state is a state that has no substructure.
- A substate is a state that's nested inside another one.
- 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 optional graphic compartment that shows a nested state machine. Substates may be nested to any level

Sequential Substates

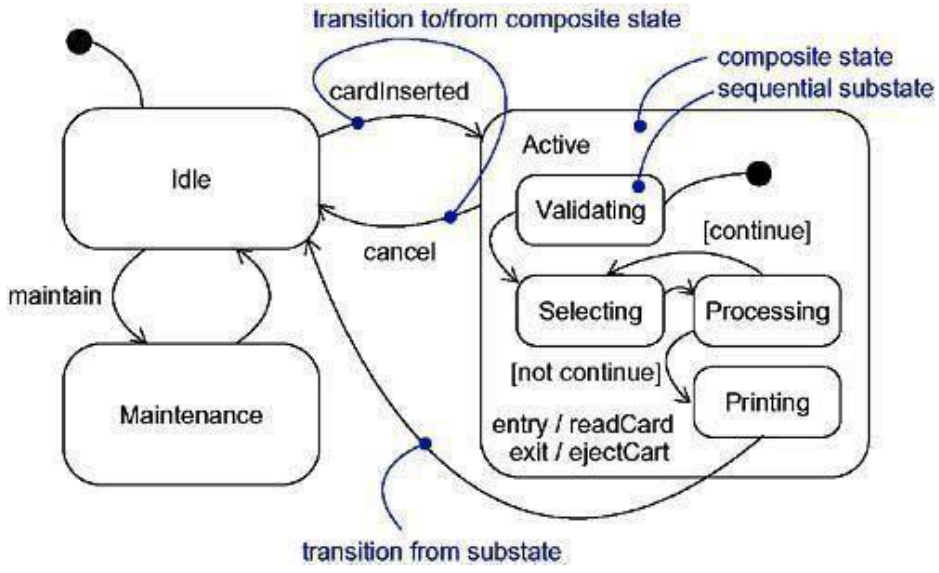
Consider the problem of modeling the behavior of an ATM. There are three basic states in which this system might be: **Idle** (waiting for customer interaction), **Active** (handling a customer's transaction), and **Maintenance** (perhaps having its cash store replenished). While **Active,** the behavior 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. Because the customer might cancel the transaction at any point, you'd have to include a suitable transition from every state in the **Active** sequence. That's messy because it's easy to forget to include these transitions in all the right places, and many such interrupting events means you end up with a multitude of transitions zeroing in on the same target state from various sources, but with the same event trigger, guard condition, and action.

Using 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



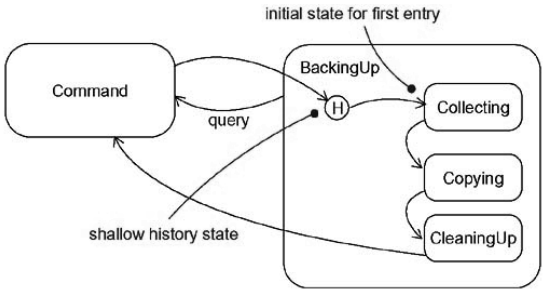
Notice also the transition from the **Active** state to the **Idle** state, triggered by the event **cancel.** In any substate of **Active,** the customer might cancel the transaction, and that returns the ATM to the **Idle** state (but only after ejecting the customer's credit card, which is the exit action dispatched on leaving the **Active** state, no matter what caused a transition out of that

state). Without substates, you'd need a transition triggered by **cancel** on every substructure state.

Substates such as **Validating** and **Processing** are called 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.

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
- When a transition enters a composite state, the action of the nested state machine starts over again at its initial state.
- 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. 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. 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 a shallow history state as a small circle containing the symbol H.

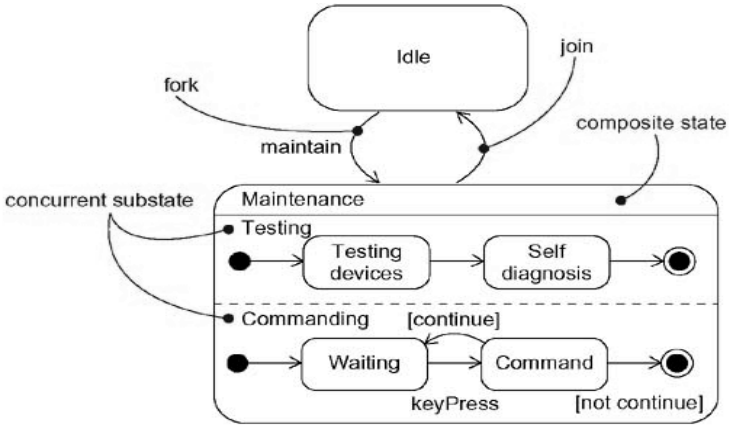


History State

- 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.
- 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(non orthogonal states)

- 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.



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.

Note: A nested concurrent state machine does not have an initial, final, or history state. However, the sequential substates that compose a concurrent state may have these features.

Common Modeling Techniques

Modeling the Lifetime of an Object

- 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).

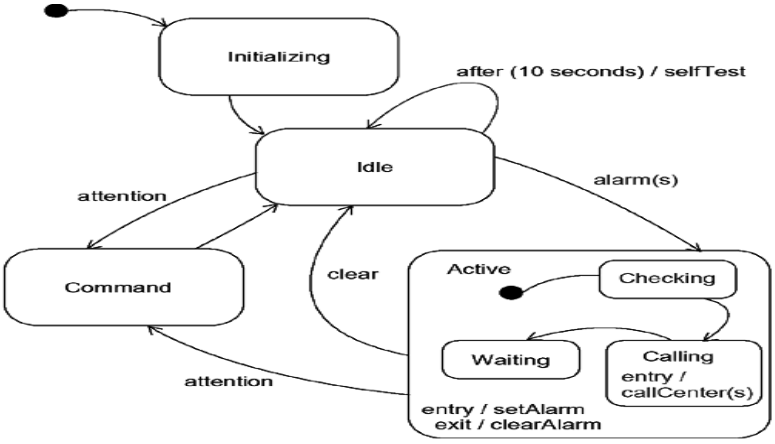
    - Expand these states as necessary by using substates.

    - 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.



3.Processes and Threads

- **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

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.
- A sequential program will process only one event at a time, queuing or discarding any concurrent external events.
- 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.

Note: You can achieve true concurrency in one of three ways:

- By distributing active objects across multiple nodes
- By placing active objects on nodes with multiple processors
- By a combination of both methods.

Classes and Events

- Active classes are just classes, albeit ones with a very special property. An active class represents an independent flow of control, whereas a plain class embodies no such flow.
- You use active classes to model common families of processes or threads.
- 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 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

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

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

- A process is heavyweight, Each program runs as a process in its own address space
- 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. 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.

Communication

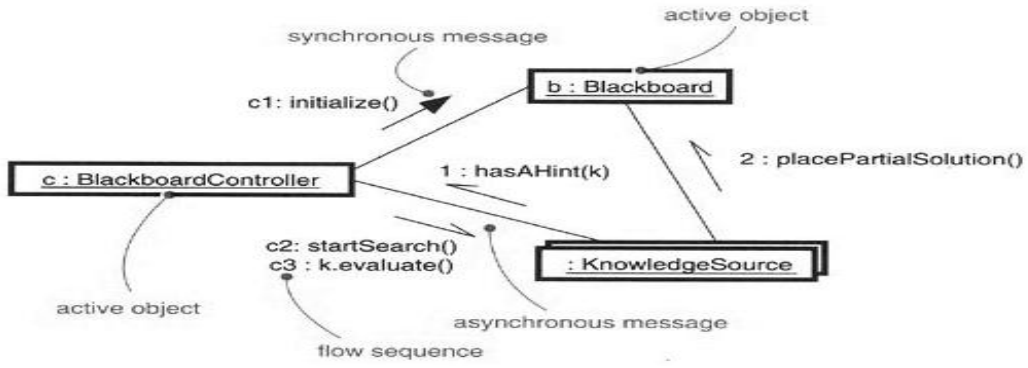
- When objects collaborate with one another, they interact by passing messages from one to the other. In a system with both active and passive objects, there are four possible combinations of interaction that you must consider
    1. A message may be passed from one passive object to another. Assuming 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.
    2. 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

        1.) One active object might synchronously call an operation of another.

        2.) One active object might asynchronously send a signal or call an operation of another object

        That kind of communication has mailbox semantics

- In the UML, we render a synchronous message as a full arrow and an asynchronous message as a half arrow



Communication

3. 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.

4. 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

- 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. 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

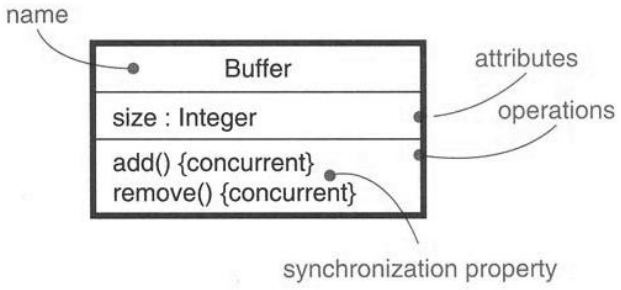


FIG:Synchronization

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

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.

1. 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.
2. Capture these static decisions in class diagrams, explicitly highlighting each active class.
3. 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.
4. Pay close attention to communication among active objects. Apply synchronous and asynchronous messaging, as appropriate.
5. 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 :find three objects that push information into the system concurrently: a StockTicker, an IndexWatcher, and aCNNNewsFeed(named s, i, and c, respectively). Two of these objects (sandi)communicate with their ownAnalystinstances (a1anda2).

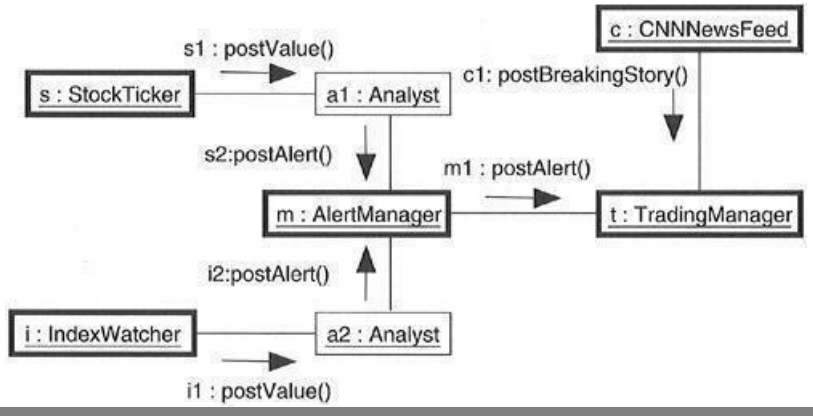


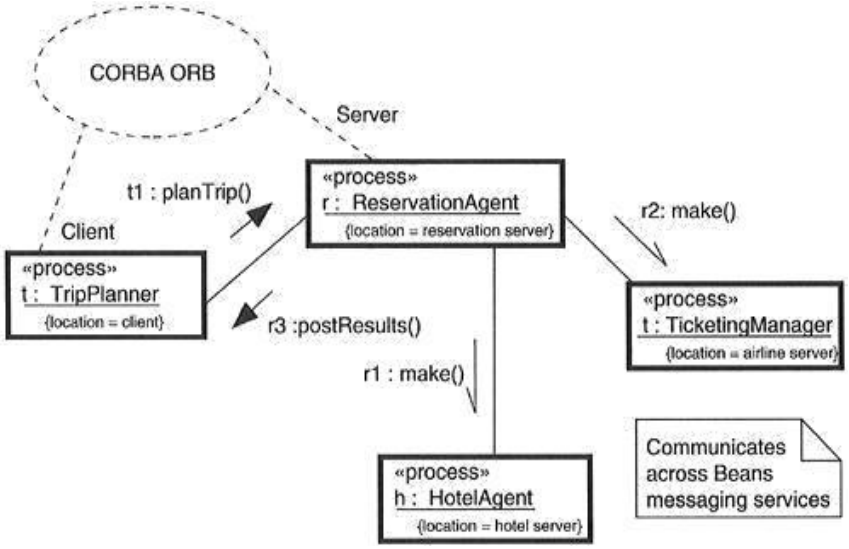
Figure Modeling Flows of Control

Modeling Inter process Communication:

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,

1. Model the multiple flows of control.
2. Consider which of these active objects represent processes and which represent threads. Distinguish them using the appropriate stereotype.
3. Model messaging using asynchronous communication; model remote procedure calls using synchronous communication.
4. Informally specify the underlying mechanism for communication by using notes, or more formally by using collaborations



4.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*

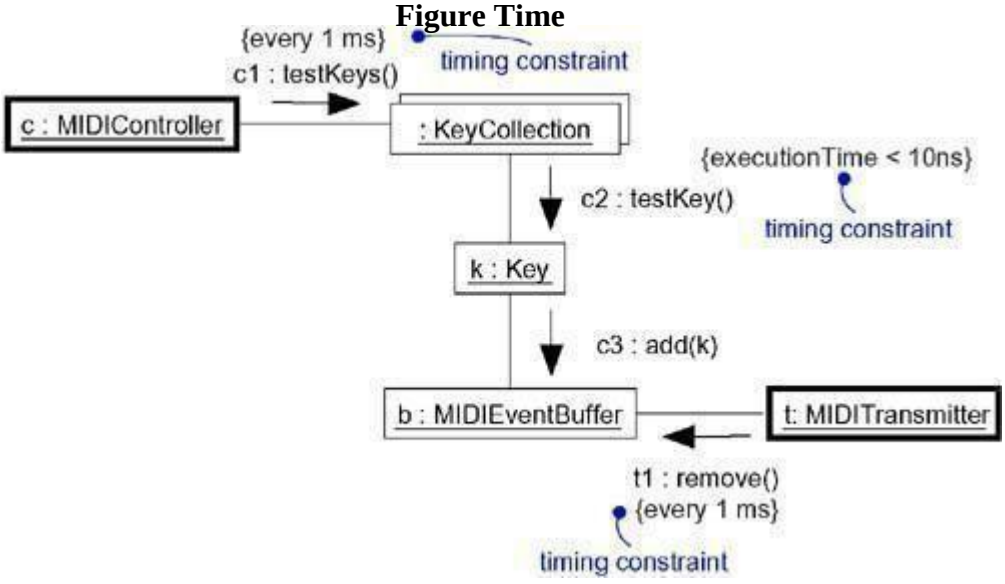
constraint is a semantic statement about the relative or absolute value of time. Graphically, a timing constraint is **rendered as for any constraint• that is, a string enclosed by bracke**ts and generally connected to an element by a

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.
- 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,
- you can place these time expressions in a timing constraint to specify the timing behavior of the system. As constraints, you can render them by placing them adjacent to the appropriate message, or you can explicitly attach them using dependency relationships.

Figure Time



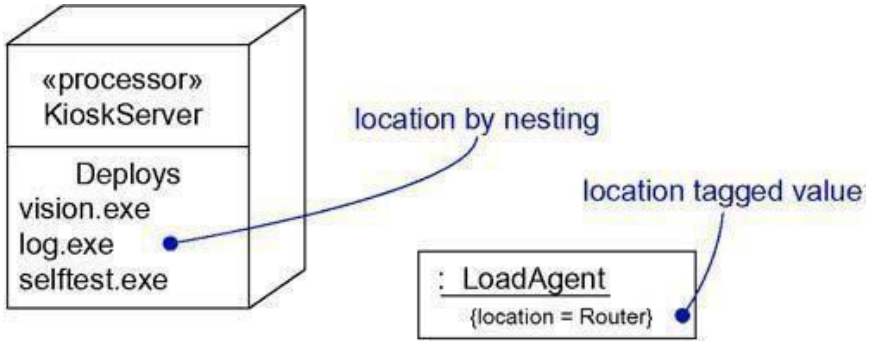
Location

- 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



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.

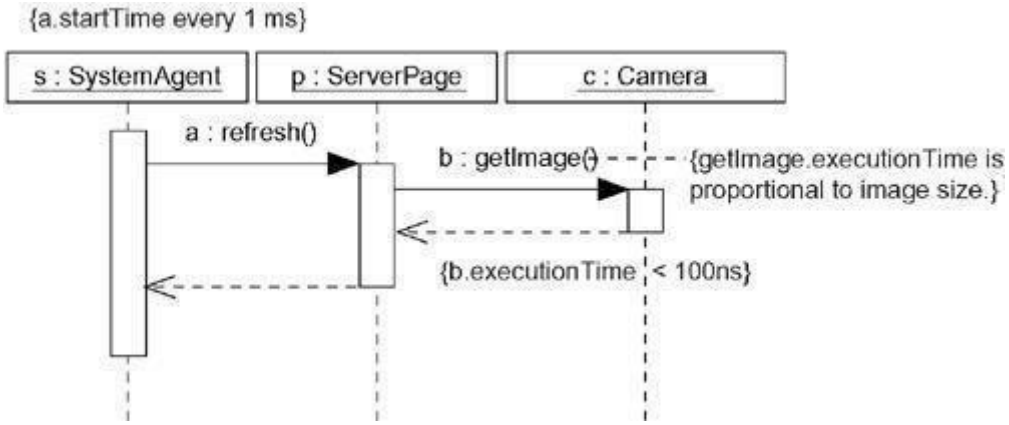
To model timing constraints,

- 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



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.
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.

To model the distribution of objects,

- For each interesting class of objects in your system, consider its locality of reference. In other words,

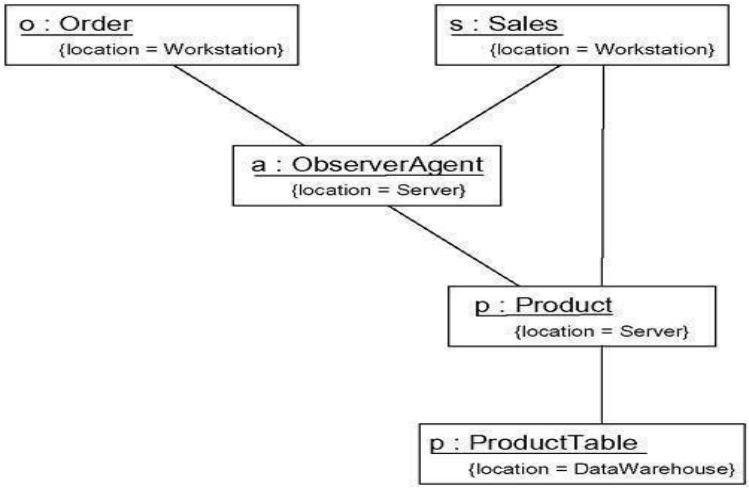
  consider all its neighbors and their locations. A tightly coupled locality will have neighboring objects close by; a loosely coupled one will have distant objects (and thus, there will be latency in communicating with them). Tentatively allocate objects closest to the actors that manipulate them.

- 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



5.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.

Common Properties

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.

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.We 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).

Common Modeling Technique

- To model reactive objects

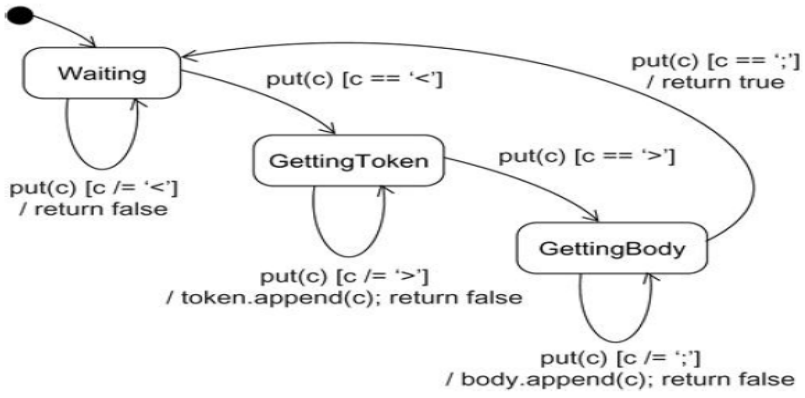
A reactive — or event-driven — object is one whose behavior is best characterized by its response to events dispatched from outside its context. A reactive object is typically idle until it receives an event. When it receives an event, its response usually depends on previous events. After the object 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.

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.

Figure Modeling Reactive Objects



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
      Waiti
      ng:   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;
}
```

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.