**Unit-4**

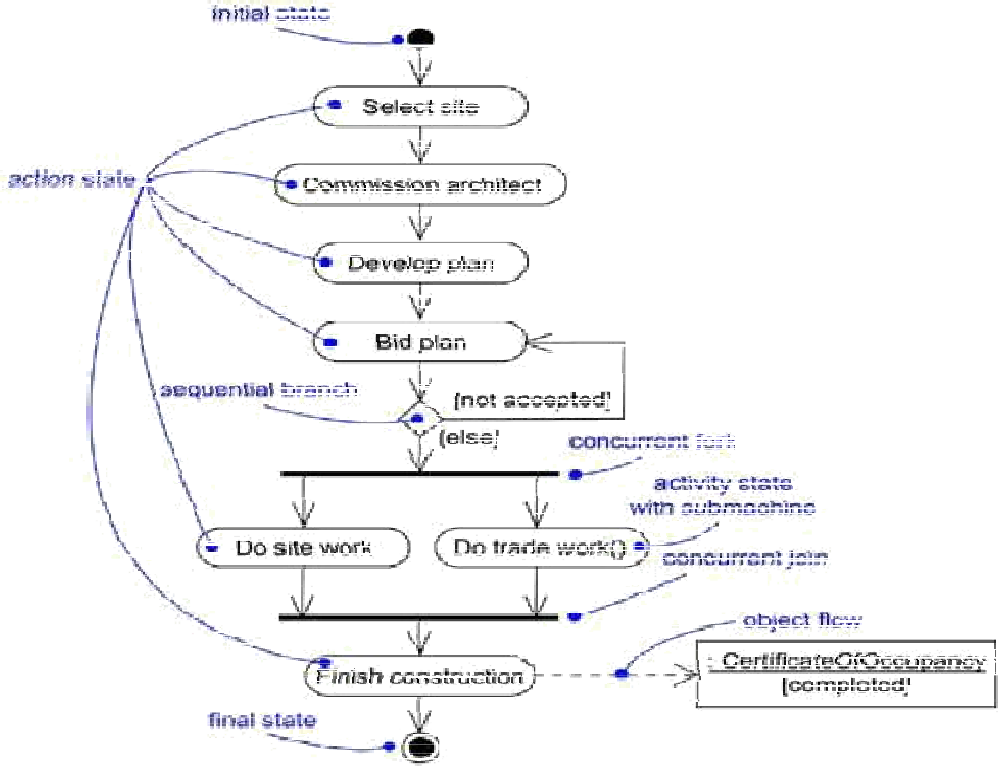
**Advanced Behavioral Modeling**

**Activity diagrams:**

Activity diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. An activity diagram is essentially a flowchart, showing flow of control from activity to activity.

* + - Activity diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.
    - Activities ultimately result in some *action,* which is made up of executable atomic computations that result in a change in state of the system or the return of a value.
    - Activity diagrams commonly contain
      * Activity states and action states
      * Transitions
      * Objects

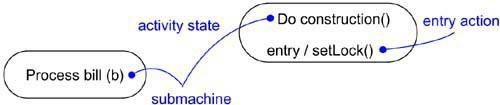
Like all other diagrams, activity diagrams may contain notes and constraints.



# Action States and Activity States:

In the flow of control modeled by an activity diagram, might evaluate some expression that sets the value of an attribute or that returns some value. Alternately, you might call an operation on an object, send a signal to an object, or even create or destroy an object.

* + - These executable, atomic computations are called action states because they are states of the system, each representing the execution of an action.
    - Action states can't be decomposed. Furthermore, action states are atomic, meaning that events may occur, but the work of the action state is not interrupted

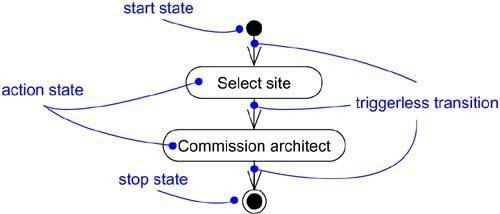


* + - In contrast, activity states can be further decomposed, their activity being represented by other activity diagrams. Furthermore, activity states are not atomic, meaning that they may be interrupted and, in general, are considered to take some duration to complete.

# Transition:

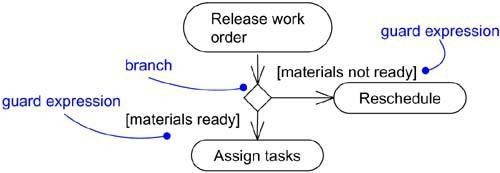
Trigger less transitions may have guard conditions, meaning that such a transition will fire only if that condition is met.

* + - When the action or activity of a state completes, flow of control passes immediately to the next action or activity state. You specify this flow by using transitions to show the path from one action or activity state to the next action or activity state



# Branching:

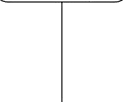
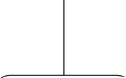
Sequential transitions are common; you can include a branch, which specifies alternate paths taken based on some Boolean expression.

* On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch.

# Forking and Joining:

Simple and branching sequential transitions are the most common paths you’ll find in activity diagrams. However especially when you are modeling workflows of business processes you might encounter flows that are concurrent.

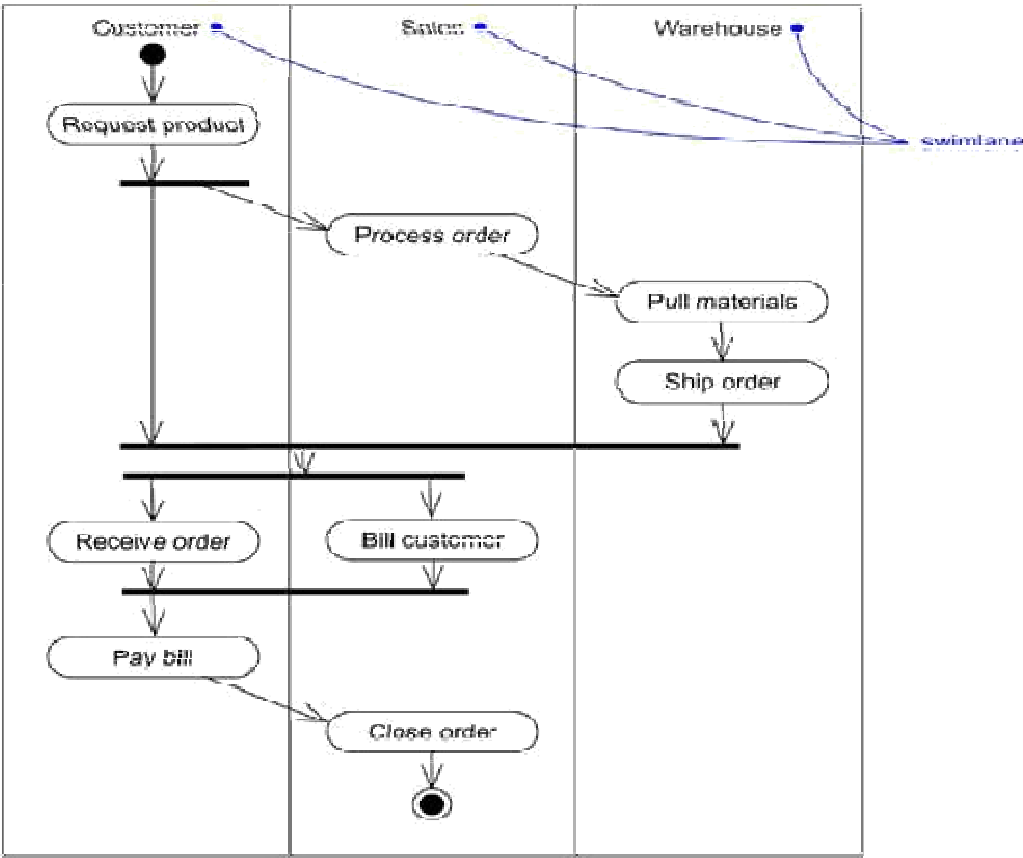
* In the UML, you use a synchronization bar to specify the forking and joining of these parallel flows of control. A synchronization bar is rendered as a thick horizontal or vertical line.
* A *fork* may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control. Below the fork, the activities associated with each of these paths continues in parallel.
* A *join* may have two or more incoming transitions and one outgoing transition. Above the join, the activities associated with each of these paths continues in parallel.



**Swimlanes:**

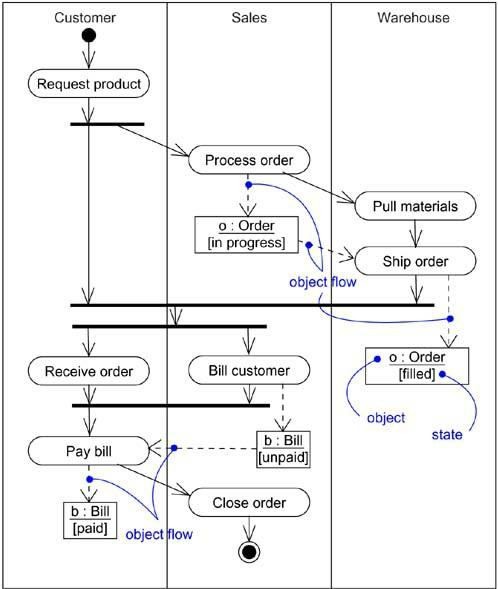
Each swimlane has a name unique within its diagram. A swimlane really has no deep semantics, except that it may represent some real-world entity.

* Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram, and each swimlane may eventually be implemented by one or more classes.
* In an activity diagram partitioned into swimlanes, every activity belongs to exactly one swimlane, but transitions may cross lanes.



# Object Flow:

Objects may be involved in the flow of control associated with an activity diagram.

* For example, in the workflow of processing an order as in the previous figure, the vocabulary of your problem space will also include such classes as Order and Bill. Instances of these two classes will be produced by certain activities.
* In addition to showing the flow of an object through an activity diagram, you can also show how its role, state and attribute values change.

**Common modeling techniques of Activity diagram:**

**Modeling a Workflow:**

To model a workflow,

* Establish a focus for the workflow. For nontrivial systems, it's impossible to show all interesting workflows in one diagram.
* Select the business objects that have the high-level responsibilities for parts of the overall workflow. These may be real things from the vocabulary of the system, or they may be more abstract. In either case, create a swimlane for each important business object.
* Identify the preconditions of the workflow's initial state and the postconditions of the workflow's final state. This is important in helping you model the boundaries of the workflow.
* Beginning at the workflow's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
* For complicated actions, or for sets of actions that appear multiple times, collapse these into activity states, and provide a separate activity diagram that expands on each.
* Render the transitions that connect these activity and action states. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
* If there are important objects that are involved in the workflow, render them in the activity diagram, as well. Show their changing values and state as necessary to communicate the intent of the object flow.



**Modeling an Operation:**

To model an operation,

* Collect the abstractions that are involved in this operation. This includes the operation's parameters (including its return type, if any), the attributes of the enclosing class, and certain neighboring classes.
* Identify the preconditions at the operation's initial state and the postconditions at the operation's final state. Also identify any invariants of the enclosing class that must hold during the execution of the operation.
* Beginning at the operation's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
* Use branching as necessary to specify conditional paths and iteration.
* Only if this operation is owned by an active class, use forking and joining as necessary to specify parallel flows of control.

****

**Advanced Behavioral Modeling:**

**Events and signals:**

**Events:**

* Things that happen are called events, and each one represents the specification of a significant occurrence that has a location in time and space.
* In the context of state machines, we use events to model the occurrence of a stimulus that can trigger a state transition.
* Events may include signals, calls, the passing of time, or a change in state.
* Events may be synchronous or asynchronous, so modeling events is wrapped up in the modeling of processes and threads.
* 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 eventsare 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 eventsare those that pass among the objects that live inside the system. An overflow exception is an example of an internal event.

In the UML, we 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 internal signal that we will need to model.
* Signals have a lot in common with plain classes. For example, signals may have instances, although we don't generally need to model them explicitly.
* Signals may also be involved in generalization relationships, permitting us 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 signal.
* We can use a dependency, stereotyped as send, to indicate that an operation sends a particular signal.



**Call Events:**

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.
* 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, we show the event, along with its parameters, as the trigger for a state transition.



**Time and Change Events:**

* A time event is an event that represents the passage of time.
* In the UML we 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 (after 2 seconds) or complex. Unless we 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 we model a change event by using the keyword **when** followed by some Boolean expression. We 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).



**Sending and Receiving Events:**

Signal events and call events involve at least two objects:

1. The object that sends the signal or invokes the operation

2. 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. In contrast, when an object calls an operation, the sender dispatches the operation and then waits for 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 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.

**State machines:**

We use an **interaction** to model the behavior of a society of objects that work together.

****

* We use a **state machine** to model the behavior of an individual object.
* A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
* 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 non atomic 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.
* We use state machines to model the dynamic aspects of a system.
* We can visualize a state machine in two ways:

1. By emphasizing the flow of control from activity to activity (using activity diagrams).
2. By emphasizing the potential states of the objects and the transitions among those states (using state chart diagrams).

* The UML provides a graphical representation of states, transitions, events, and actions. This notation permits we to visualize the behavior of an object in a way that emphasizes the important elements in the life of that object.

**States:**

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.

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)
* Shutting Down (its gas is off but its blower is on, flushing residual heat from the system).

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. Sub states:** The nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) sub states.

**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.



**Initial and Final States:**

There are two special states that may be defined for an object's state machine.

1. 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.
2. 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.

**Transitions:**

* A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter in to 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.

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

A transition has five parts:

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

2. **Event trigger**: The event whose 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 Guard eligible to fire; if the expression evaluates False, the transition does not fire and if condition 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 is 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.
* A transition may have multiple sources as well as multiple targets.

****

**State Chart Diagrams:**

* State chart diagrams are one of the five diagrams in the UML for modelling the dynamic aspects of systems. A state chart diagram shows a state machine.
* A *state chart 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.
* 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.



* Statechart diagrams commonly contain
* Simple states and composite states
* Transitions, including events and actions
* Statechart diagrams may contain branches, forks, joins, action states, activity states, objects, initial states, final states, history states, and so on. Indeed, a state chart diagram may contain any and all features of a state machine.

**Common Modeling Technique:**

**Modeling Reactive Objects:**

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.

