UNIT-5

Unit 5: More UML diagrams: State-Chart diagrams, Activity diagrams, Component Diagrams, Deploymentdiagrams, Object diagrams.

Activity Diagrams

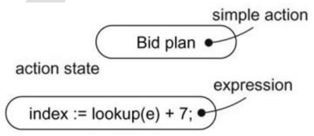
- An activity diagram shows the flow from activity to activity. An is an ongoing nonatomic execution within a state machine.
- Activities ultimately result in some action, which is made up of executable atomic computations that result in a change in state of the system or the return of a value.
- Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression.
- Graphically, an activity diagram is a collection of vertices and arcs.

Contents

- Activity diagrams commonly contain
    - Activity states and action states
    - Transitions
    - Objects
- Like all other diagrams, activity diagrams may contain notes and constraints.

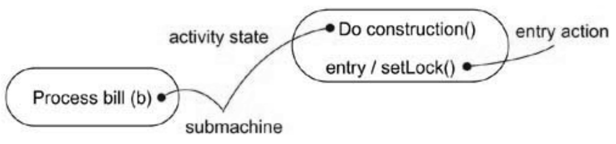
Action States and Activity States

- Executable, atomic computations are called **action states** because they are states of the system, each representing the execution of an action.
- We represent an action state using a lozenge shape (a symbol with horizontal top and bottom and convex sides). Inside that shape, you may write any expression.
- 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.
- Finally, the work of an action state is generally considered to take insignificant execution time.



Action States

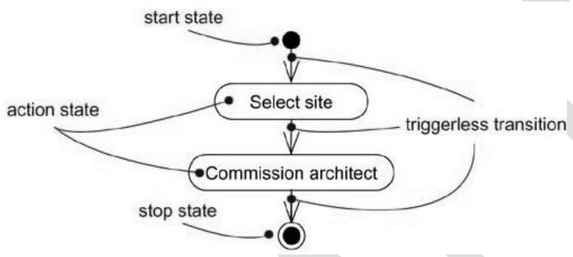
- **activity states** can be further decomposed, their activity being represented by other activity diagrams
- Furthermore, activity states are not atomic, meaning that they may be interrupted and, in general, are considered to take some duration to complete.
- An action state is an activity state that cannot be further decomposed.
- We can think of an activity state as a composite, whose flow of control is made up of other activity states and action states.



Activity States

Transitions

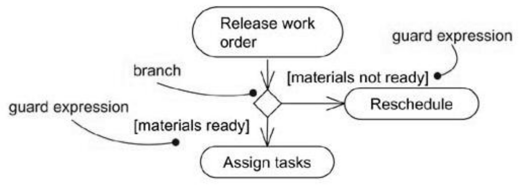
- When the action or activity of a state completes, flow of control passes immediately to the next action or activity state.
- We specify this flow by using transitions to show the path from one action or activity state to the next action or activity state.
- In the UML, you represent a transition as a simple directed line



Triggerless Transitions

Branching

- As in a flowchart, you can include a branch, which specifies alternate paths taken based on some Boolean expression.
- We represent a branch as a diamond. A branch may have one incoming transition and two or more outgoing ones.
- On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch.
- On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch. Across all these outgoing transitions, guards should not overlap (otherwise, the flow of control would be ambiguous), but they should cover all possibilities (otherwise, the flow of control would freeze).
- As a convenience, you can use the keyword else to mark one outgoing transition, representing the path taken if no other guard expression evaluates to true.

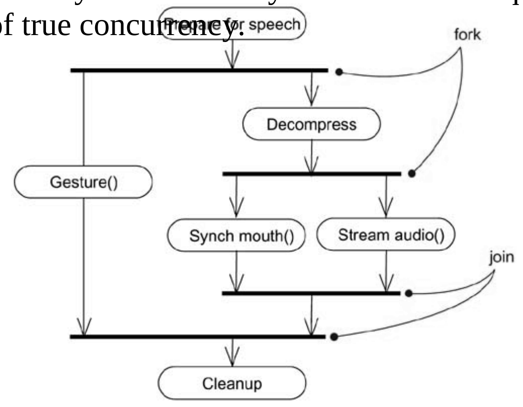


Branching

Forking and Joining

- When we are modeling workflows of business processes—we 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.

- **Fork** represents the splitting of a single flow of control into two or more concurrent flows of control
- 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.
- Conceptually, the activities of each of these flows are truly concurrent, although, in a running system, these flows may be either truly concurrent or sequential yet interleaved, thus giving only the illusion of true concurrency.



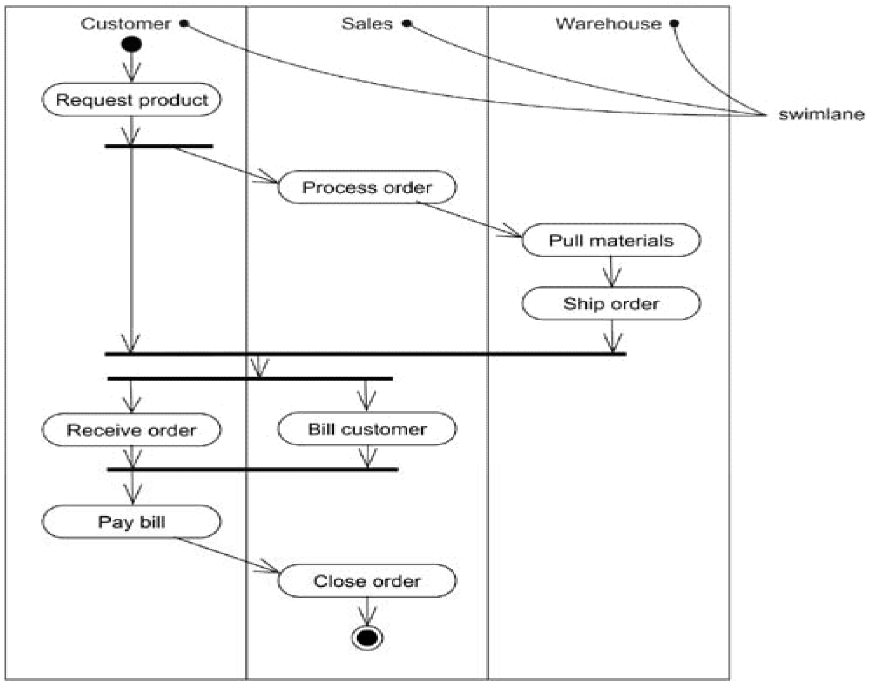
<u>Forking and Joining</u>

- **A Join** represents the synchronization of two or more concurrent flows of control.
- 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.
- At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join.

Swimlanes

- We'll find it useful, especially when you are modeling workflows of business processes, to partition the activity states on an activity diagram into groups, each group representing the business organization responsible for those activities.
- In the UML, each group is called a swimlane because, visually, each group is divided from its neighbor by a vertical solid line
- A swimlane specifies a locus of activities
- Each swimlane has a name unique within its diagram.

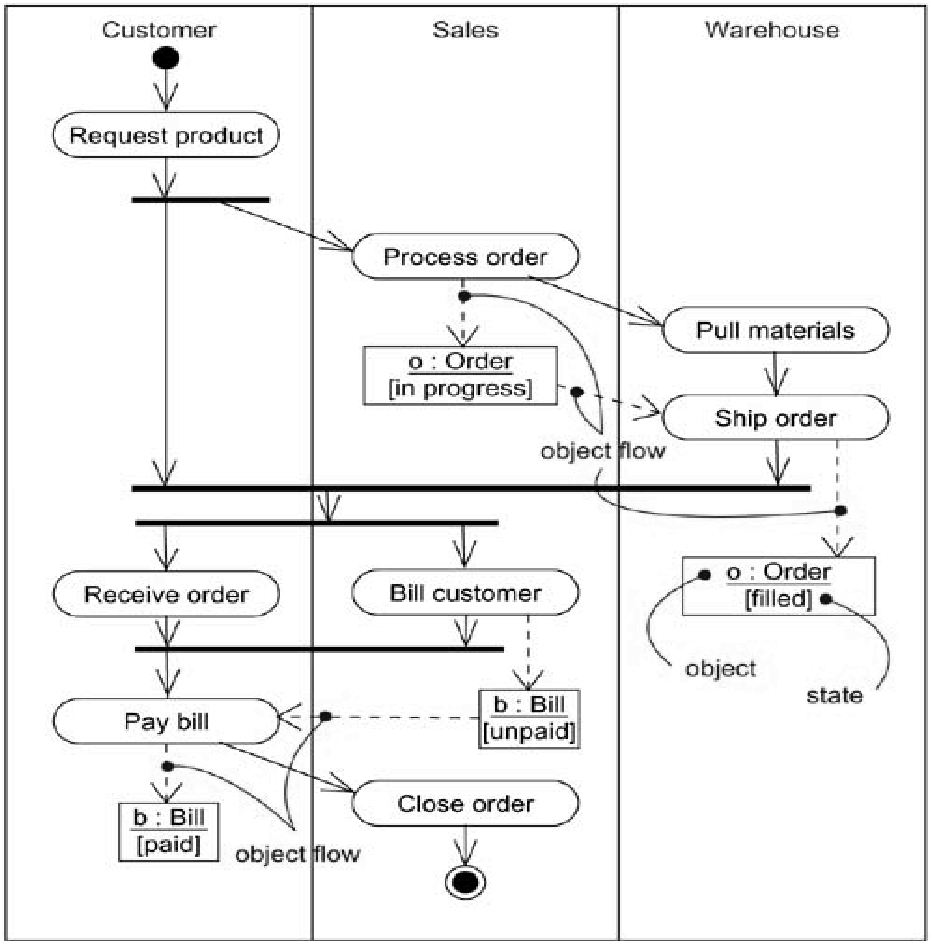
- Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram, and each swimlane may eventually be implemented by one or more classes.
- In an activity diagram partitioned into swimlanes, every activity belongs to exactly one swimlane, but transitions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



Swimlanes

Object Flow

- Objects may be involved in the flow of control associated with an activity diagram.
- We can specify the things that are involved in an activity diagram by placing these objects in the diagram, connected using a dependency to the activity or transition that creates, destroys, or modifies them.
- This use of dependency relationships and objects is called an object flow because it represents the participation of an object in a flow of control.
- We can also show how its role, state and attribute values change.
- We represent the state of an object by naming its state in brackets below the object's name.
- Similarly, We can represent the value of an object's attributes by rendering them in a compartment below the object's name.



Object Flow

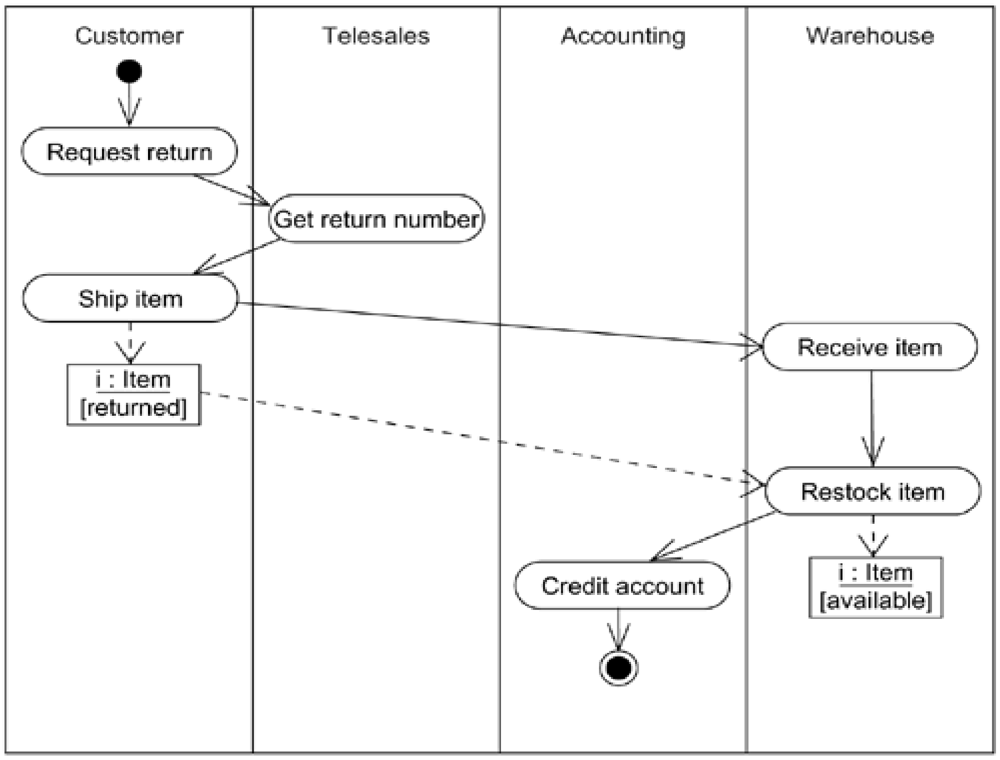
Common Uses

- We use activity diagrams to model the dynamic aspects of a system
- These dynamic aspects may involve the activity of any kind of abstraction in any view of a system's architecture, including classes, interfaces, components, and nodes.
- When you model the dynamic aspects of a system, we'll typically use activity diagrams in two ways.
    - To model a workflow
    - To model an operation

Common Modeling Techniques

Modeling a Workflow

- No software-intensive system exists in isolation; there's always some context in which a system lives, and that context always encompasses actors that interact with the system.
- Especially for mission critical, enterprise software, you'll find automated systems working in the context of higher-level business processes.
- These business processes are kinds of workflows because they represent the flow of work and objects through the business.
- 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.



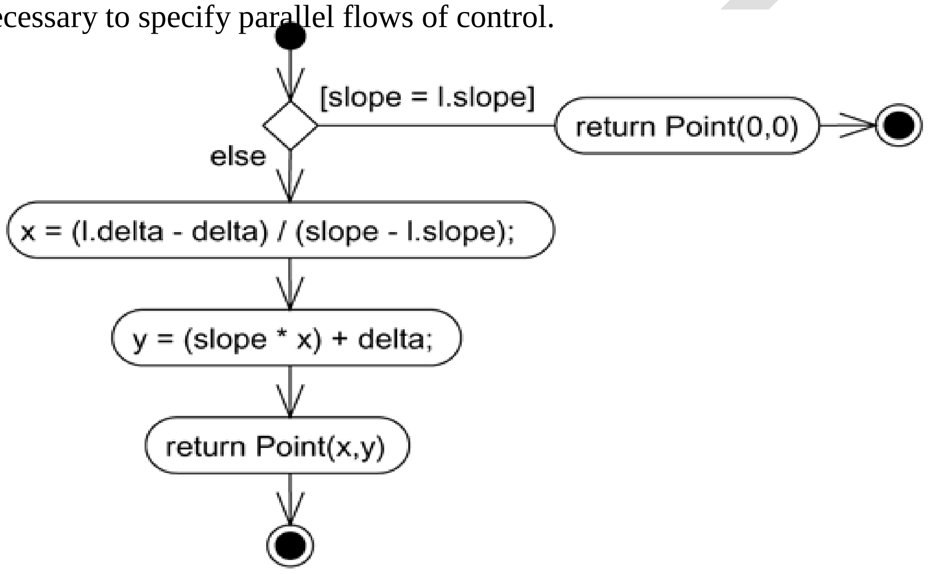
Modeling a Workflow

Modeling an Operation

- An activity diagram can be attached to any modeling element for the purpose of visualizing, specifying, constructing, and documenting that element's behavior.
- You can attach activity diagrams to classes, interfaces, components, nodes, use cases, and collaborations.
- The most common element to which you'll attach an activity diagram is an operation.
- An activity diagram is simply a flowchart of an operation's actions.
- An activity diagram's primary advantage is that all the elements in the diagram are semantically tied to a rich underlying model.
- 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.



Modeling an Operation

Forward and Reverse Engineering

- **Forward engineering** (the creation of code from a model) is possible for activity diagrams, especially if the context of the diagram is an operation.
- For example, using the previous activity diagram, a forward engineering tool could generate the following C++ code for the operation intersection.

```
Point Line::intersection (l : Line) {

  if (slope == l.slope) return Point(0,0);

  int x = (l.delta - delta) / (slope - l.slope);

  int y = (slope * x) + delta;
return Point(x, y);

}
```

- **Reverse engineering** (the creation of a model from code) is also possible for activity diagrams, especially if the context of the code is the body of an operation.
- In particular, the previous diagram could have been generated from the implementation of the class Line.

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.

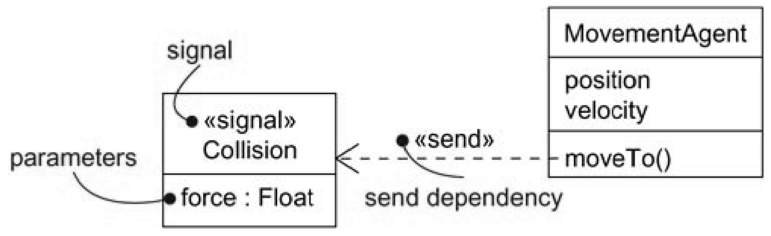
Kinds of Events

- Events may be external or internal.
- External events are those that pass between the system and its actors.
- 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.

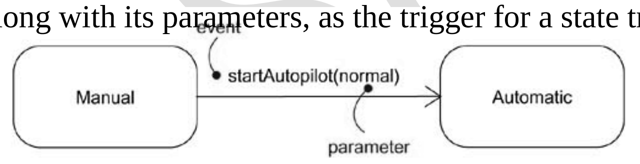
- We model signals (and exceptions) as stereotyped classes. We can use a dependency, stereotyped as **send**, to indicate that an operation sends a particular signal.



Signals

Call Events

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



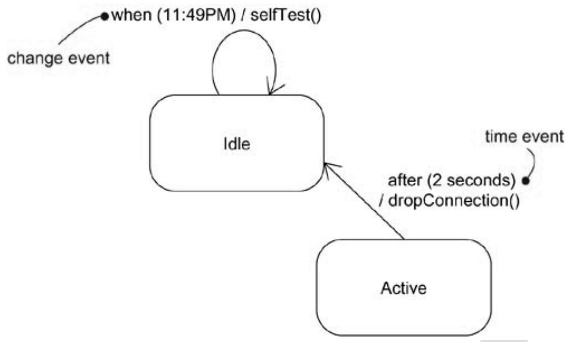
Call Events

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

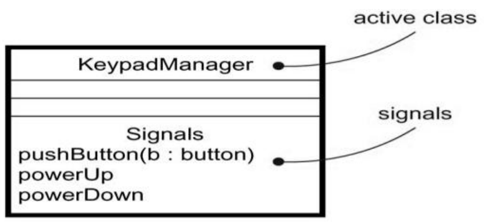


Time and Change Events

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

Sending and Receiving Events

- Signal events and call events involve at least two objects:
  - The object that sends the signal or invokes the operation
  - 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

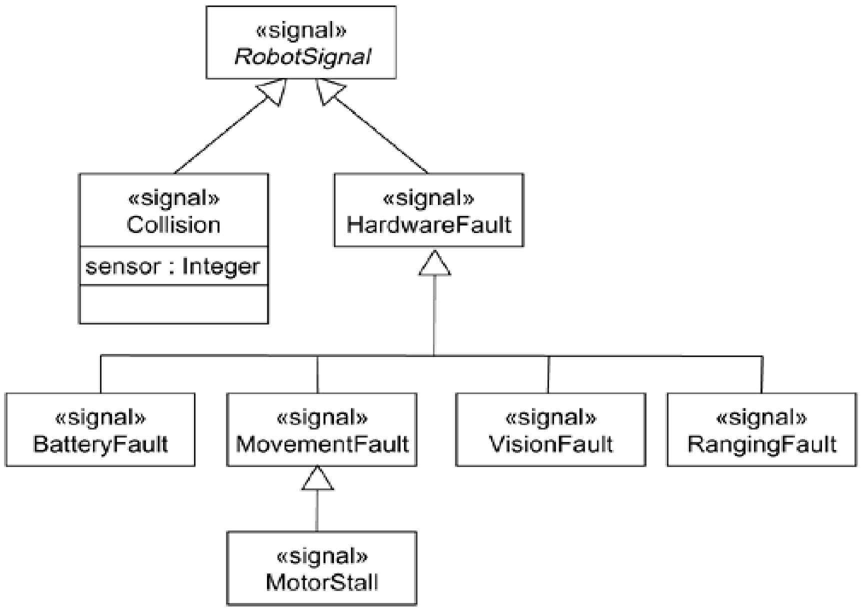


Signals and Active Classes

Common Modeling Techniques

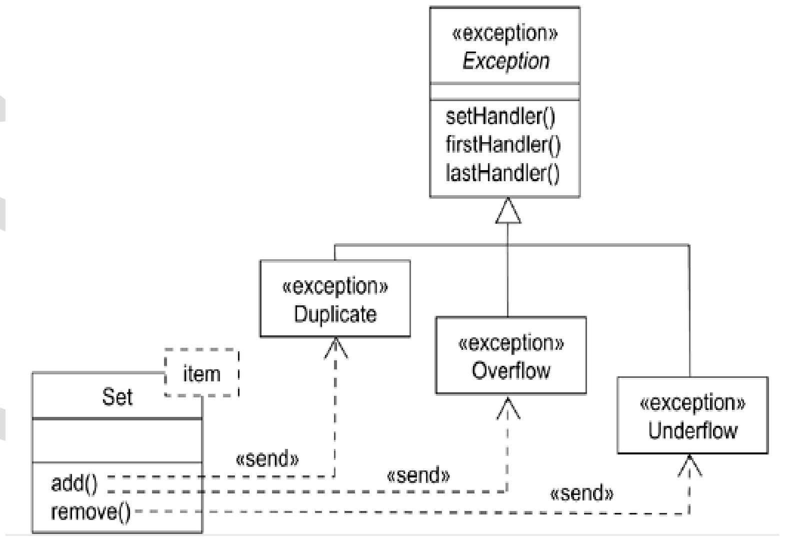
Modeling a Family of Signals

- 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 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
- <u>To model exceptions</u>
    - 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.



<u>Modeling Exceptions</u>

State-Chart diagrams:

A statechart diagram is a view of a **state machine** that models the changing behavior of a state. Statechart diagrams show the various states that an object goes through, as well as the events that cause a transition from one state to another.

Statechart diagram model elements

The common model elements that statechart diagrams contain are:

- States
- Start and end states
- Transitions
- Entry, do, and exit actions

A state represents a condition during the life of an object during which it satisfies some condition or waits for some event. Start and end states represent the beginning or ending of a process. A state transition is a relationship between two states that indicates when an object can move the focus of control on to another state once certain conditions are met. In a statechart diagram, a transition to self element is similar to a state transition. However, it does not move the focus of control. A state transition contains the same source and target state.

Actions in a Statechart diagram

Each state on a statechart diagram can contain multiple internal actions. An action is best described as a task that takes place within a state. There are four possible actions within a state:

- On entry
- On exit
- Do
- On event

Creating a statechart diagram in Rational Rose

A statechart diagram is usually placed under the Logical View package. Right-click on the Logical View package and select New>Statechart Diagram to create a Statechart Diagram. Name your diagram and then double-click on the name to open the diagram work area.

States

Place the start state, , end state, , and states, , on the diagram work area by selecting the respective icon from the diagram toolbox and then clicking on the work area at the point where you want to place the states.

To name the states, double-click on the state. This action will bring up the State Specification dialog box. In the General tab, type the name of your state in the Name text box.

Actions

To add an action to a state, select the Actions tab in the State Specification dialog box, right-click anywhere in the white area and select Insert from the shortcut menu. An action will be automatically placed. Double-click the action item to bring up the Action Specification dialog box. Select an action from the When drop-down list box. Type the action description in the Name field. Click OK and then click OK again to exit the State Specification dialog box.

Transitions

To create a transition to self , click the icon and then click on the state. To create transitions between the states, click the icon and then click on the first state and drag and release on the next state. To name the transitions, double-click on the transition to bring up the State Transition Specification dialog box. Type the name or label in the Event text box and click OK.

Figure 1.shows a Statechart Diagram depicting the various elements of a state machine.

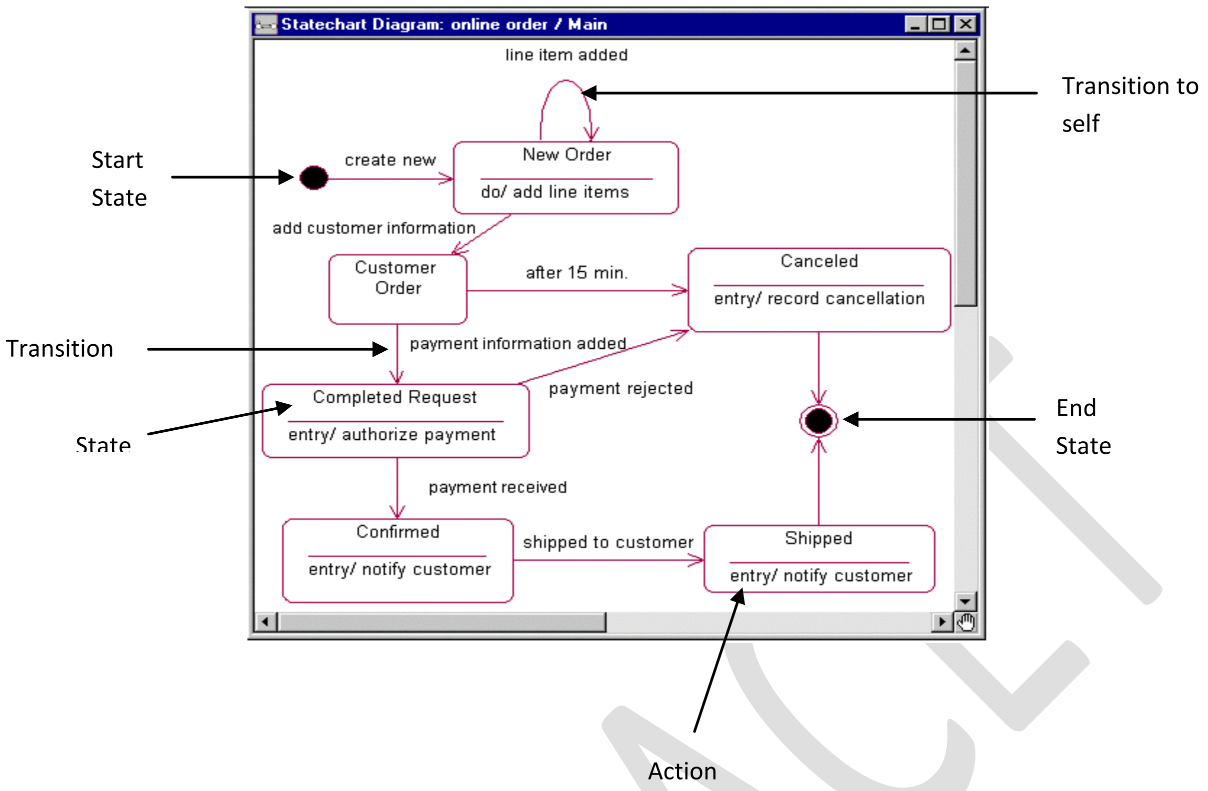


Figure 1. A state machine

<u>Component diagrams:</u>

Component diagrams are different in terms of nature and behavior. Component diagrams are used to model physical aspects of a system. Now the question is what are these physical aspects? Physical aspects are the elements like executable, libraries, files, documents etc which resides in a node.So component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

So from that point component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

A single component diagram cannot represent the entire system but a collection of diagrams are used to represent the whole.

So the purpose of the component diagram can be summarized as:

- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

Component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executable, libraries etc.So the purpose of this diagram is different, and Component diagrams are used during the implementation phase of an application. But it is prepared well in advance to visualize the implementation details. Initially the system is designed using different UML diagrams and then when the artifacts are ready component diagrams are used to get an idea of the implementation.This diagram is very important because without it the application cannot be implemented efficiently. A well prepared component diagram is also important for other aspects like application performance, maintenance etc.

So before drawing a component diagram the following artifacts are to be identified clearly:

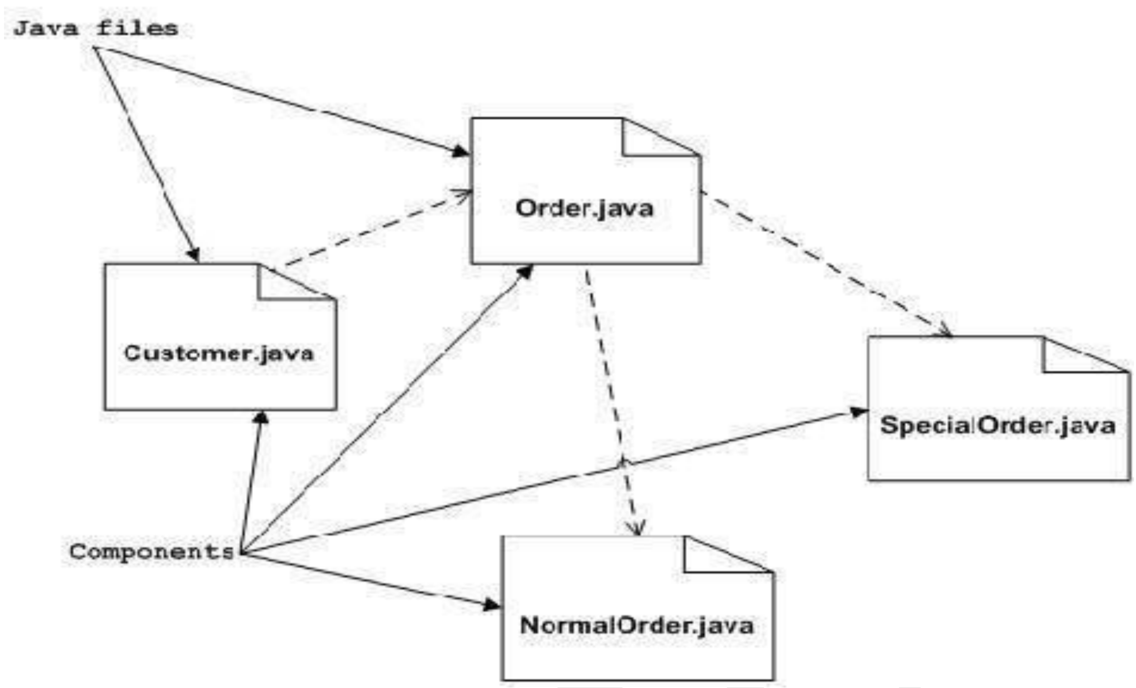
- Files used in the system.
- Libraries and other artifacts relevant to the application.
- Relationships among the artifacts.

Now after identifying the artifacts the following points needs to be followed:

- Use a meaningful name to identify the component for which the diagram is to be drawn.
- Prepare a mental layout before producing using tools.
- Use notes for clarifying important points.

The following is a component diagram for order management system. Here the artifacts are files. So the diagram shows the files in the application and their relationships. In actual the component diagram also contains dlls, libraries, folders etc.In the following diagram four files are identified and their relationships are produced. Component diagram cannot be matched directly with other UML diagrams discussed so far. Because it is drawn for completely different purpose.So the following component diagram has been drawn considering all the points mentioned above:

Component diagram of an order management system



Deployment diagrams:

Deployment diagrams are used to visualize the topology of the physical components of a system where the software components are deployed. So deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams consist of nodes and their relationships. The name *Deployment* itself describes the purpose of the diagram. Deployment diagrams are used for describing the hardware components where software components are deployed.

Component diagrams and deployment diagrams are closely related. Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware. UML is mainly designed to focus on software artifacts of a system. But these two diagrams are special diagrams used to focus on software components and hardware components. So most of the UML diagrams are used to handle logical components but deployment diagrams are made to focus on hardware topology of a system. Deployment diagrams are used by the system engineers.

The purpose of deployment diagrams can be described as:

- Visualize hardware topology of a system.
- Describe the hardware components used to deploy software components.
- Describe runtime processing nodes.

Deployment diagram represents the deployment view of a system. It is related to the component diagram. Because the components are deployed using the deployment diagrams. A deployment diagram consists of nodes. Nodes are nothing but physical hardwares used to deploy the application.

Deployment diagrams are useful for system engineers. An efficient deployment diagram is very important because it controls the following parameters

- Performance
- Scalability
- Maintainability
- Portability

So before drawing a deployment diagram the following artifacts should be identified:

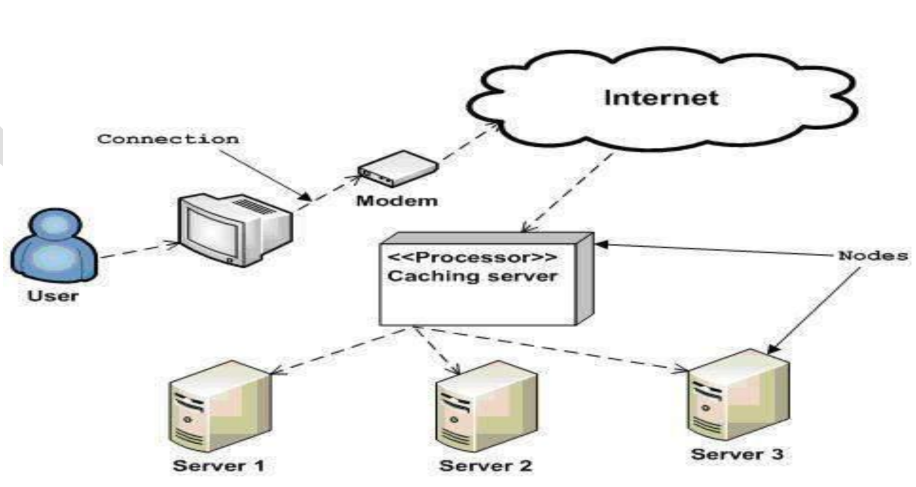
- Nodes
- Relationships among nodes

The following deployment diagram is a sample to give an idea of the deployment view of order management system. Here we have shown nodes as:

- Monitor
- Modem
- Caching server
- Server

The application is assumed to be a web based application which is deployed in a clustered environment using server 1, server 2 and server 3. The user is connecting to the application using internet. The control is flowing from the caching server to the clustered environment.So the following deployment diagram has been drawn considering all the points mentioned above:

Deployment diagram of an order management system


Internet
Connection
Modem
User
<<Processor>>
Caching server
Nodes
Server 1
Server 2
Server 3