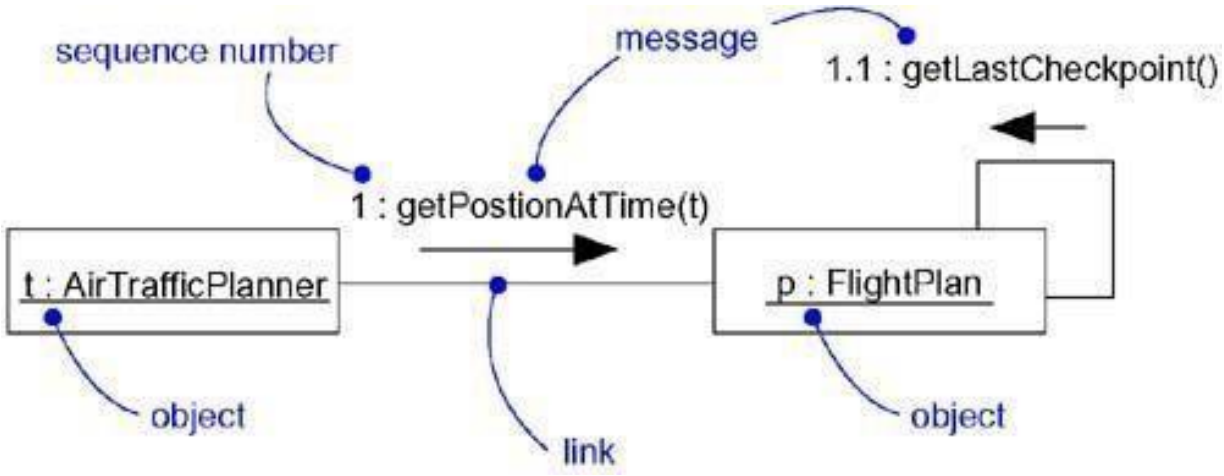
INTERACTIONS

An *interaction* is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose. A *message* is a specification of a communication between objects that conveys information with the expectation that activity will ensue.

Fig: Messages, Links, and Sequencing



Context

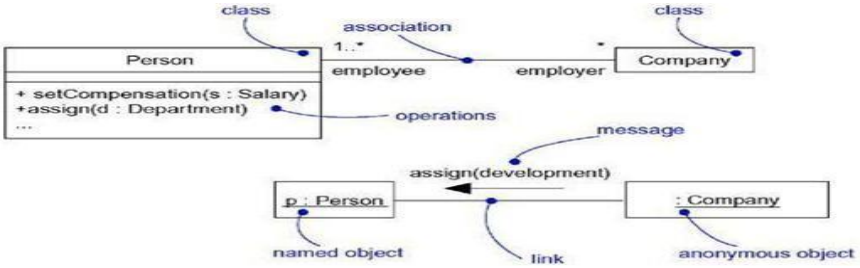
- Interaction can find wherever objects are linked to one another.
- Interaction can find in the collaboration of objects that exist in the context of your system or subsystem.
- It also find interactions in the context of an operation.
- Finally, you'll find interactions in the context of a class.

Object and Roles:

- The objects that participate in an interaction are either concrete things or prototypical things.
- A concrete thing, an object represents something in the real world. For example, p, an instance of the class Person, might denote a particular human.
- A prototypical thing, p might represent any instance of Person.

Links

- A link is a semantic connection among objects.
- In general, a link is an instance of an association.
- Following fig. shows, wherever a class has an association to another class, there may be a link between the instances of the two classes; wherever there is a link between two objects, one object can send a message to the other object.



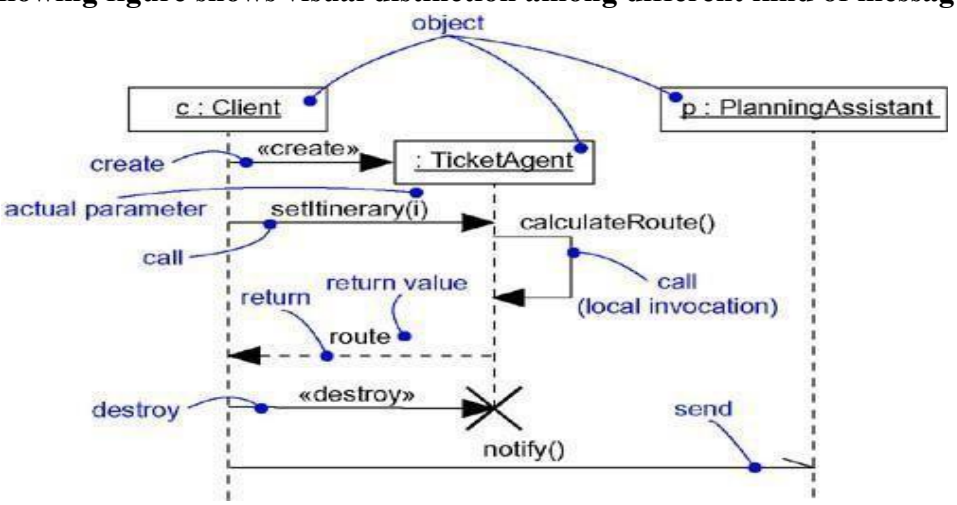
Following five standard stereotypes you can use

- association – corresponding object is visible by association.
- self – dispatches of operation.
- global – represents enclosing scope.
- local – local scope
- parameter – parameter visibility.

Messages

- A message is the specification of a communication among objects that conveys information with the expectation that activity will ensue.
- The receipt of a message instance may be considered an instance of an event.
- When you pass a message, the action that results is an executable statement that forms an abstraction of a computational procedure.
- An action may result in a change in state.
- **UML can model several kind of actions**:
- **call** - invoke an operation **Return** - return a value to the caller
- **Send** - send signal to an object **Create** - creates an object
- **Destroy** - destroys an object

Following figure shows visual distinction among different kind of messages



Sequencing

- When an object passes a message to another object (in effect, delegating some action to the receiver), the receiving object might in turn send a message to another object, which might send a message to yet a different object, and so on. This stream of messages forms a sequence. Any sequence must have a beginning.
- Most commonly, you can specify a procedural or nested flow of control, rendered using a filled solid arrowhead, as Figure shows. In this case, the message findAt is specified as the first message nested in the second message of the sequence (2.1).

Procedural Sequence

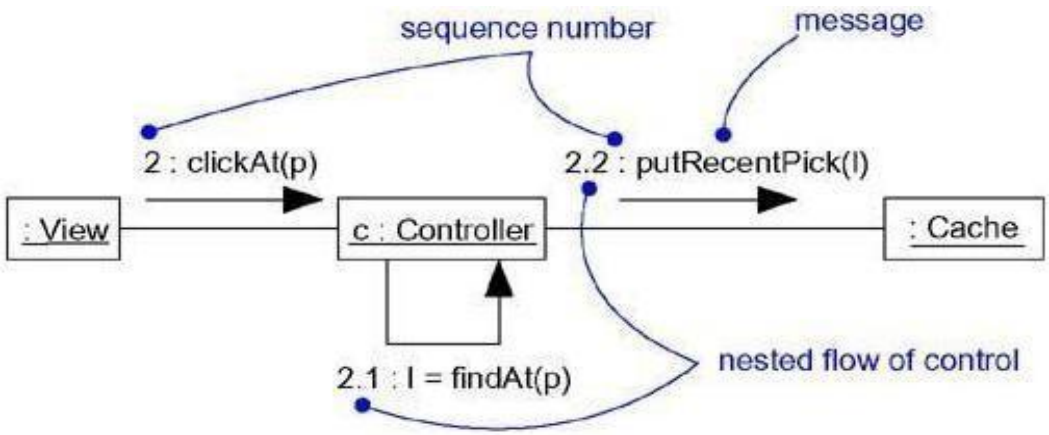
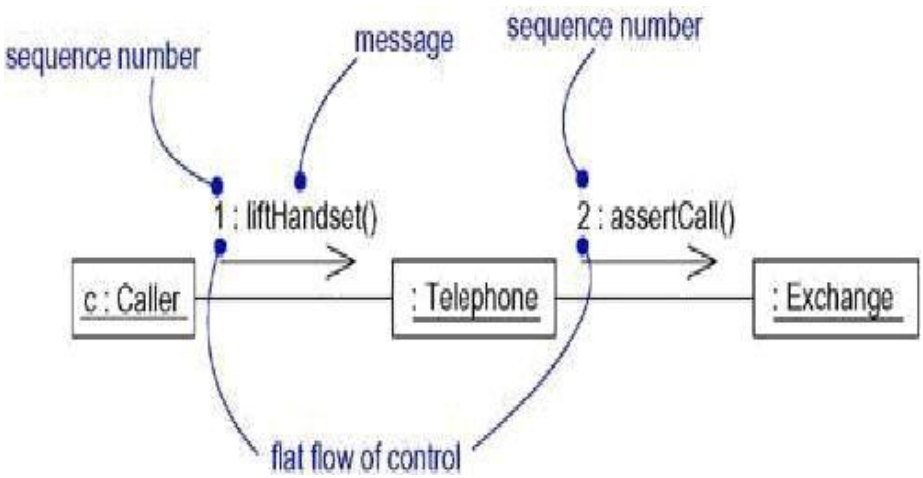


Figure below shows, a flat flow of control, rendered using a stick arrowhead, to model the nonprocedural progression of control from step to step.
In this case, the message assertCall is specified as the second message in the sequence.



Creation , Modification and Destruction :

- To specify if an object or link enters and/or leaves during an interaction you can attach one of the following constraints to the element:
- **New** – Specifies that the instance or link is created during execution of the enclosing Interaction
- **Destroyed** – Specifies that the instance or link is destroyed prior to completion of execution of the enclosing interaction
- **Transient**– Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution

- When you model an interaction, you typically include both objects (each one playing a specific role) and messages (each one representing the communication between objects, with some resulting action).
- You can visualize those objects and messages involved in an interaction in two ways:
  1. by emphasizing the time ordering of its messages
  2. by emphasizing the structural organization of the objects that send and receive messages.

- In the UML, the first kind of representation is called a sequence diagram; the second kind of representation is called a collaboration diagram.
- Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams.

Common Modeling Techniques

1. Modeling a flow control

To model a flow of control

- Set the context for the interaction, whether it is the system as a whole, a class, or an individual operation.
- Set the stage for the interaction by identifying which objects play a role; set their initial properties, including their attribute values, state, and role.
- If your model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of communication that take place in this interaction. Specify the nature of the links using the UML's standard stereotypes and constraints, as necessary.

Modeling a flow control contd..

- In time order, specify the messages that pass from object to object. As necessary, distinguish the different kinds of messages; include parameters and return values to convey the necessary detail of this interaction.
- Also to convey the necessary detail of this interaction, adorn each object at every moment in time with its state and role.

This figure is an example of a sequence diagram, which emphasizes the time order of messages.

Eg. Flow of control by time

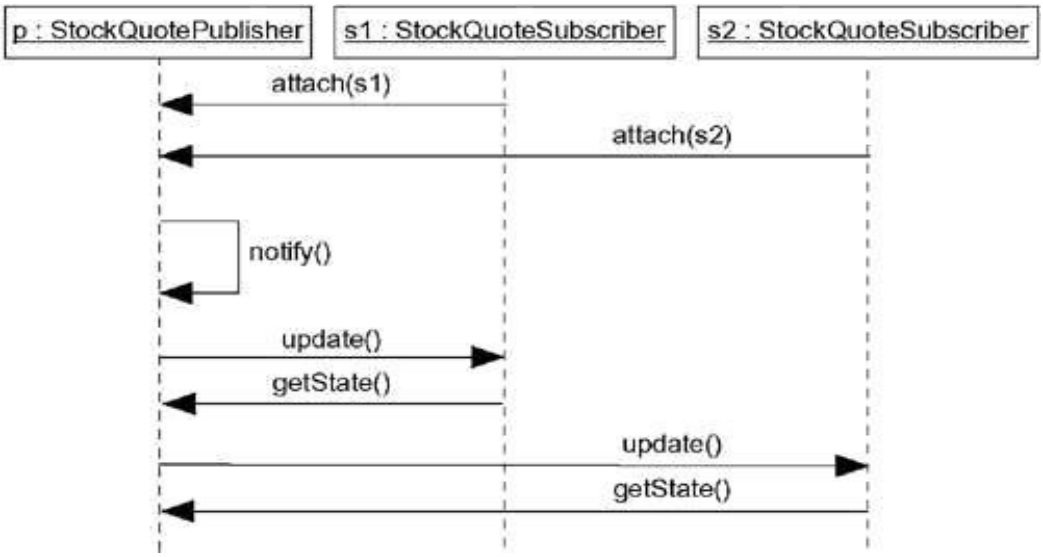
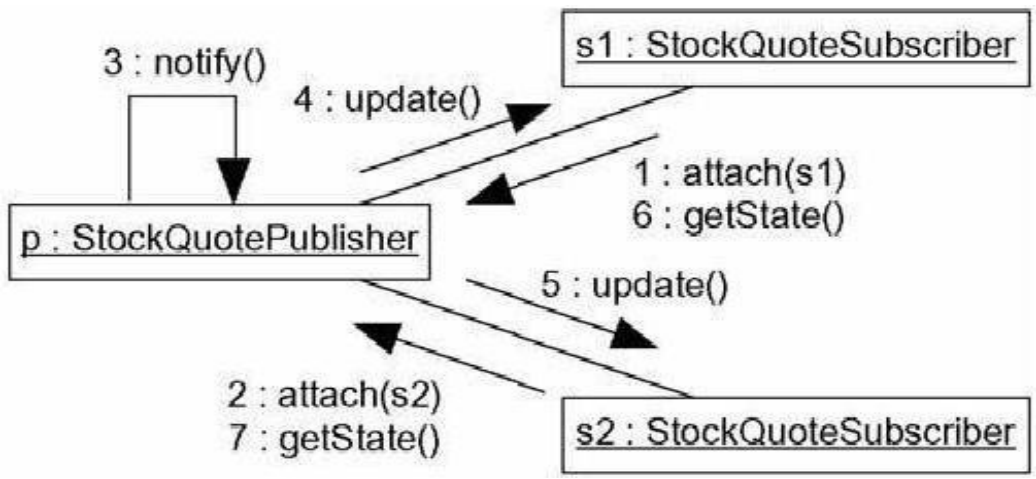


Figure is semantically equivalent to the previous one but it is drawn as a collaboration diagram, which emphasizes the structural organization of the objects. This figure shows the same flow of control, but it also provides a visualization of the links among these objects. **Eg. Flow of control by organization**



INTERACTION DIAGRAMS

- Interaction diagrams are not only important for modeling the dynamic aspects of a system, but also for constructing executable systems through forward and reverse engineering.
- An *interaction diagram shows an interaction, consisting of a set of objects and their relationships,* including the messages that may be dispatched among them.
- A ***sequence diagram*** is an interaction diagram that emphasizes the time ordering of messages.
- A **C*ollaboration diagram*** is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.

Common Properties

- An interaction diagram is just a special kind of diagram and shares the same common properties as do all other diagrams
- A name and graphical contents that are a projection into a model.

Contents

Interaction diagrams commonly contain

- Objects
- Links
- Messages

Sequence Diagrams

- Describe the flow of messages, events, actions between objects
- Show concurrent processes and activations
- Show time sequences that are not easily depicted in other diagrams
- Typically used during analysis and design to document and understand the logical flow of your system.
- A sequence diagram emphasizes the time ordering messages
- **Sequence Diagram Key Parts**
- **participant:** object or entity that acts in the diagram
- – diagram starts with an unattached "found message" arrow
- **message:** communication between participant objects
- the **axes** in a sequence diagram:
- – horizontal: which object/participant is acting
- – vertical: time (down -> forward in time)

SEQUENCE DIAGRAM

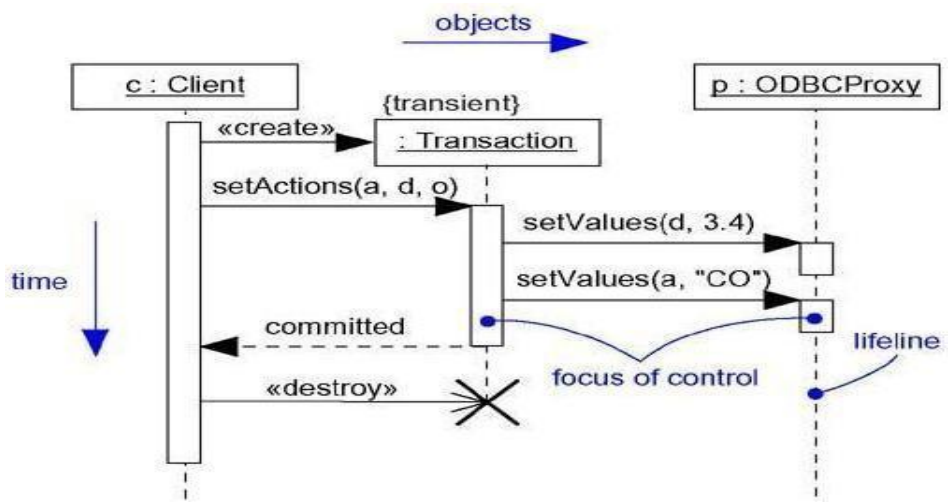
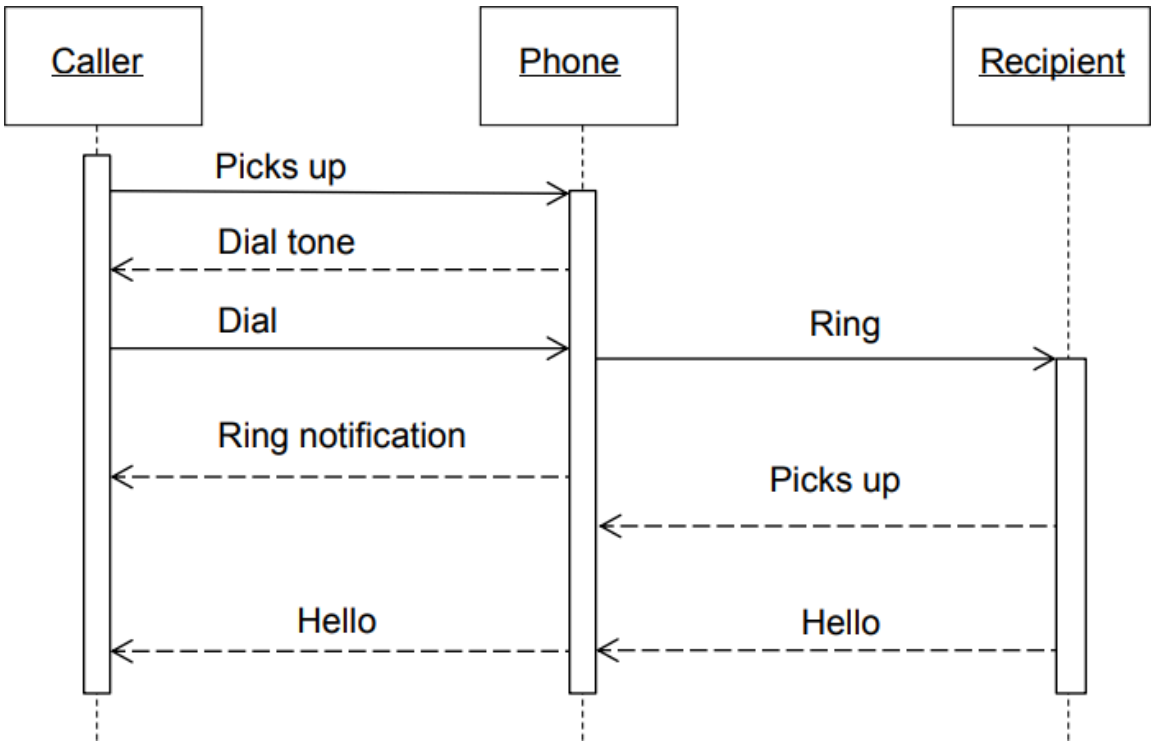


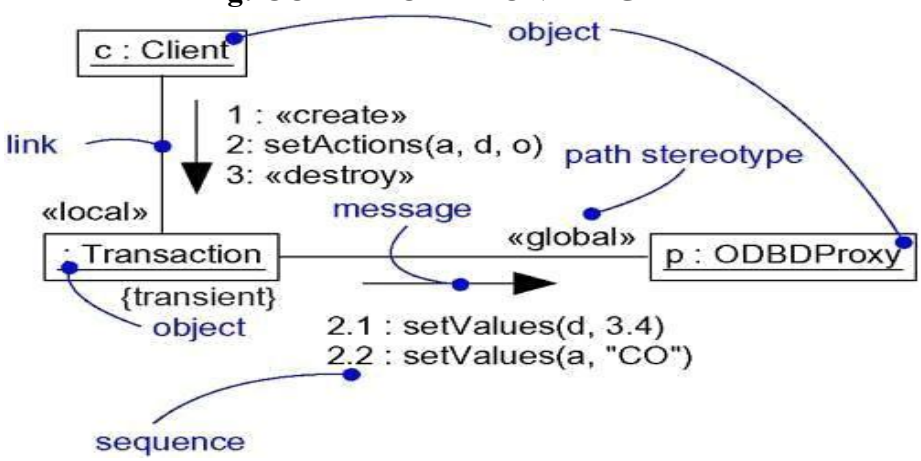
Fig: Sequence Diagram (make a phone call)



Collaboration Diagrams

- A collaboration diagram emphasizes the organization of the objects that participate in an interaction.
- In the collaboration diagram, the method call sequence is indicated by some numbering technique.
- The number indicates how the methods are called one after another.
- Collaboration diagrams have two features that distinguish them from sequence diagrams.
- First, there is the path. To indicate how one object is linked to another, you can attach a path stereotype to the far end of a link (such as <<local>>, indicating that the designated object is local to the sender).
- Second, there is the sequence number. To indicate the time order of a message, you prefix the message with a number (starting with the message numbered 1), increasing monotonically for each new message in the flow of control (2, 3, and so on).

Fig: COLLABORATION DIAGRAM

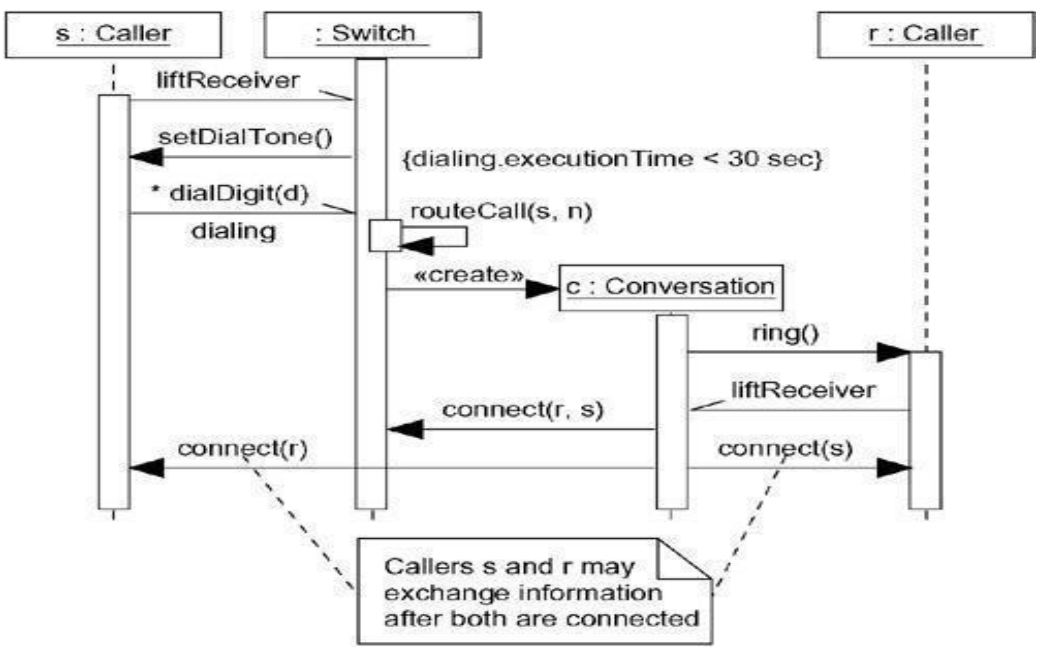


Common Modeling Techniques

1. Modeling flow control by Time ordering

- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the sequence diagram from left to right, placing the more important objects to the left and their neighboring objects to the right.
- Set the lifeline for each object. In most cases, objects will persist through the entire interaction. For those objects that are created and destroyed during the interaction, set their lifelines, as appropriate, and explicitly indicate their birth and death with appropriately stereotyped messages
- Starting with the message that initiates this interaction, lay out each subsequent message from top to bottom between the lifelines, showing each message's properties (such as its parameters), as necessary to explain the semantics of the interaction.
- If you need to visualize the nesting of messages or the points in time when actual computation is taking place, adorn each object's lifeline with its focus of control.
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

Eg. Modeling Flows of Control by Time Ordering



2. Modeling Flows of control by organization

- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the collaboration diagram as vertices in a graph, placing the more important objects in the center of the diagram and their neighboring objects to the

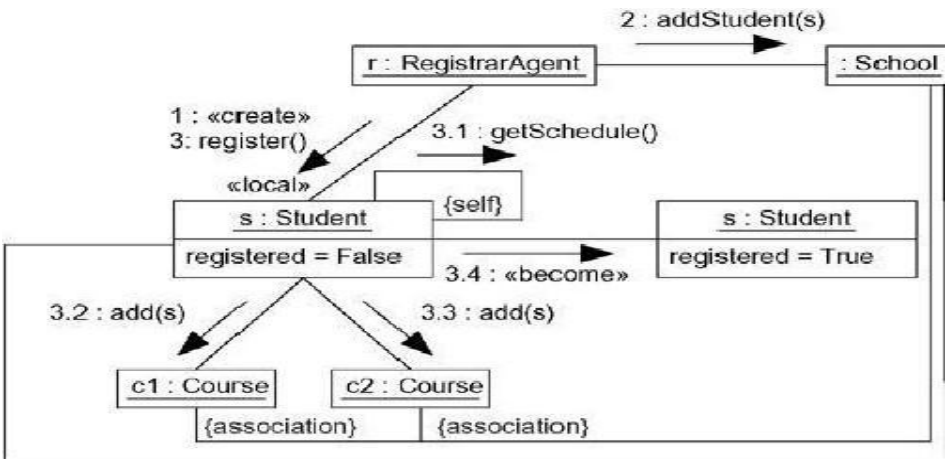
outside.

- Set the initial properties of each of these objects. If the attribute values, tagged values, state, or role of any object changes in significant ways over the duration of the interaction, place a

duplicate object on the diagram, update it with these new values, and connect them by a message stereotyped as **become** or **copy** (with a suitable sequence number).

- Specify the links among these objects, along which messages may pass.
  - Lay out the association links first; these are the most important ones, because they represent structural connections.
  - Lay out other links next, and adorn them with suitable path stereotypes (such as **global** and **local**) to explicitly specify how these objects are related to one another.
- Starting with the message that initiates this interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate. Show nesting by using Dewey decimal numbering.
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and post conditions to each message.

Fig: Modeling Flows of control by organization



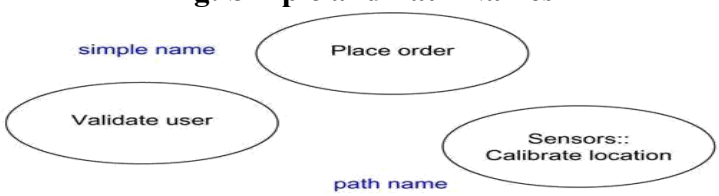
Use Cases

- A *use case* is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor.
- Graphically, a use case is rendered as an ellipse.

Names

- Every use case must have a name that distinguishes it from other use cases. A *name* is a textual string.
- name alone is known as a *simple name;* a *path name* is the use case name prefixed by the name of the package in which that use case lives.
- A use case is typically drawn showing only its name.

Fig: Simple and Path Names



Note

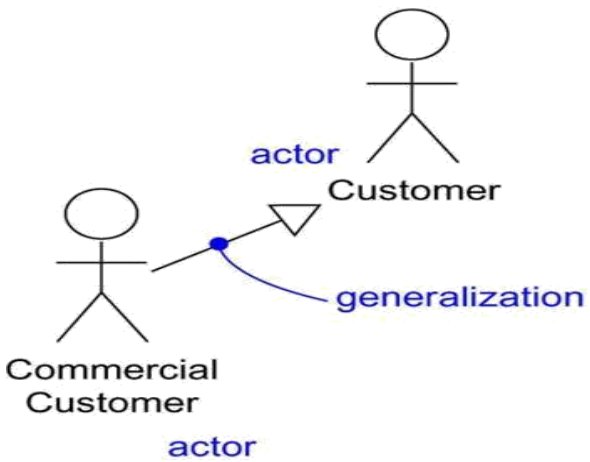
- A use case name may be text consisting of any number of letters, numbers, and most punctuation marks and may continue over several lines

Use Cases and Actors

- An actor represents a coherent set of roles that users of use cases play when interacting with these use cases.
- An actor represents a role that a human, a hardware device, or eve another system plays with a system.

As Figure indicates, actors are rendered as stick figures. You can define general kinds of actors (such as Customer) and specialize them (such as CommercialCustomer) using generalization relationships.

Fig: Actors



Use Cases and Flow of Events

A use case describes *what* a system (or a subsystem, class, or interface) does but it does not specify *how* it does it. When you model, it's important that you keep clear the separation of concerns between this outside and inside view.

For example, in the context of an ATM system, you might describe the use case ValidateUser in the following way:

Main flow of events:

- The use case starts when the system prompts the *Customer* for a PIN number. The *Customer* can now enter a PIN number via the keypad. The *Customer* commits the entry by pressing the Enter button. The system then checks this PIN number to see if it is valid. If the PIN number is valid, the system acknowledges the entry, thus ending the use case.

Exceptional flow of events:

- The *Customer* can cancel a transaction at any time by pressing the Cancel button, thus

restarting the use case. No changes are made to the *Customer*'s account.

Exceptional flow of events:

- The *Customer* can clear a PIN number anytime before committing it and reenter a new PIN number.

Exceptional flow of events:

- If the *Customer* enters an invalid PIN number, the use case restarts. If this happens three times in a row, the system cancels the entire transaction, preventing the *Customer* from interacting with the ATM for 60 seconds.

Use Cases and Scenarios

A Scenario is a specific sequence of actions that illustrates behavior.

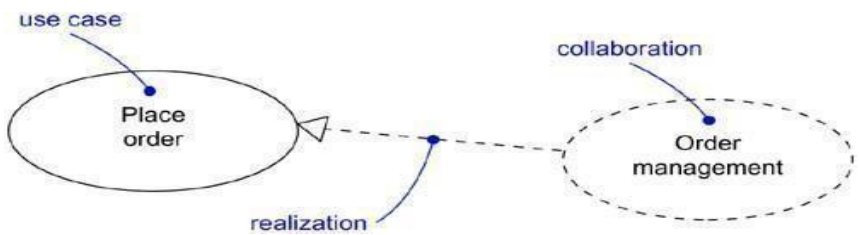
- Scenarios are to use cases as instances are to classes, meaning that a scenario is basically one instance of a use case.

Use Cases and Collaborations

- A use case captures the intended behavior of the system (or subsystem, class, or interface) you are developing, without having to specify how that behavior is implemented.
- That's an important separation because the analysis of a system (which specifies behavior) should, as much as possible, not be influenced by implementation issues (which specify how that behavior is to be carried out)

As Figure shows, you can explicitly specify the realization of a use case by a collaboration.

Fig: Use Cases and Collaborations

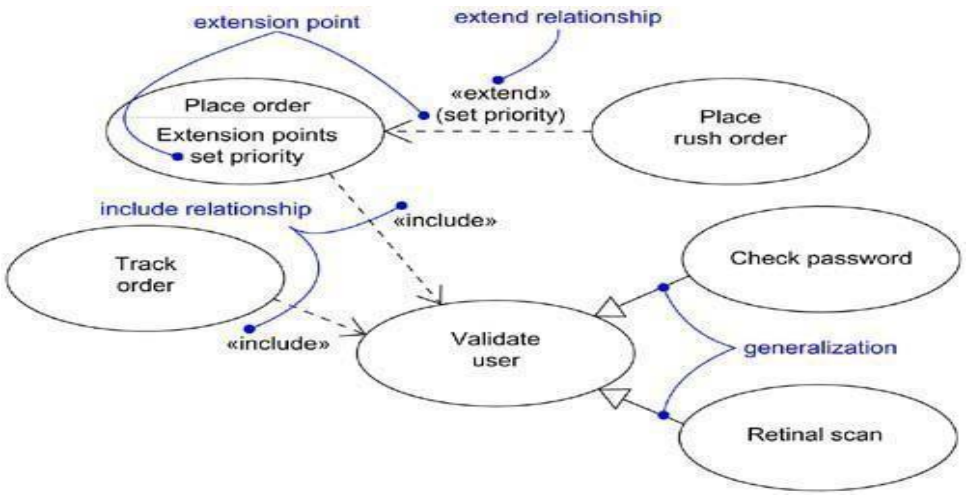


Organizing Use Cases

It is also possible to organize use cases by specifying generalization, include, and extend relationships among them.

- Apply these relationships in order to factor common behavior (by pulling such behavior from other use cases that it includes) and in order to factor variants (by pushing such behavior into other use cases that extend it).
- An **include** relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base.
- An **extend** relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case.

Fig: Generalization, Include, and Extend



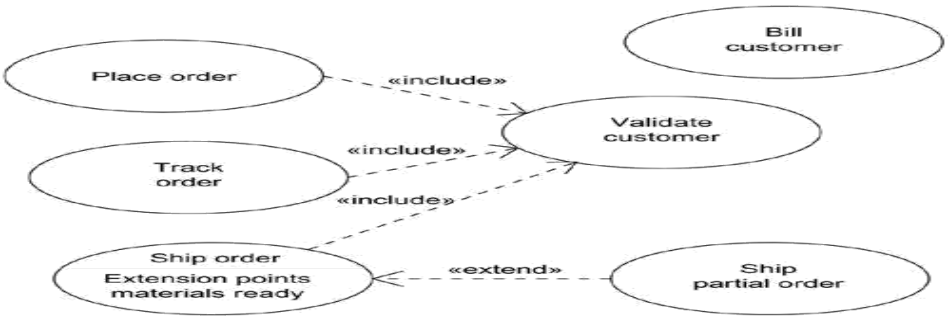
Common Modeling Techniques

1. Modeling the Behavior of an Element

To model the behavior of an element,

- Identify the actors that interact with the element. Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element's functions.
- Organize actors by identifying general and more specialized roles.
- For each actor, consider the primary ways in which that actor interacts with the element. Consider also interactions that change the state of the element or its environment or that involve a response to some event.
- Consider also the exceptional ways in which each actor interacts with the element.
- Organize these behaviors as use cases, applying include and extend relationships to factor common behavior and distinguish exceptional behavior

Fig: Modeling the Behavior of an Element



Use Case Diagrams

- A *use case diagram* is a diagram that shows a set of use cases and actors and their relationships.

Common Properties

- A use case diagram is just a special kind of diagram and shares the same common properties as do all other diagrams.
- a name and graphical contents that are a projection into a model.

Contents

Use case diagrams commonly contain

- Use cases
- Actors
- Dependency, generalization, and association relationships

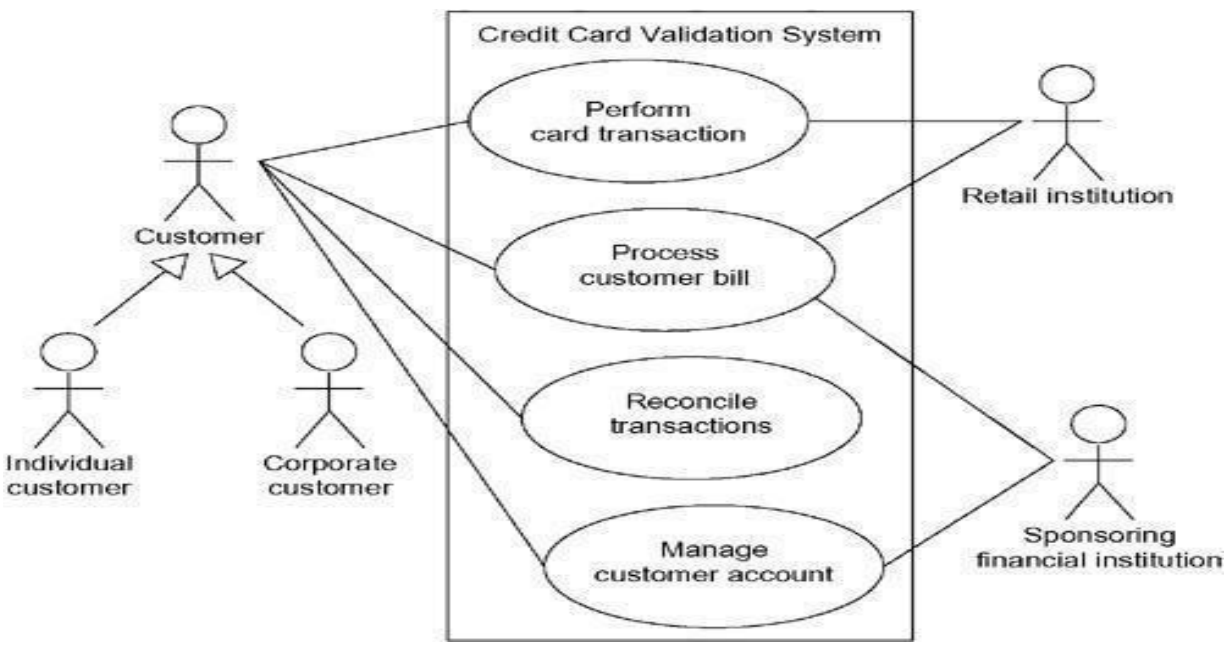
Like all other diagrams, use case diagrams may contain notes and constraints.

Common Modeling Techniques

1. Modeling the Context of a System

To model the context of a system,

- Identify the actors that surround the system by considering which groups require help from the system to perform their tasks; which groups are needed to execute the system's functions; which groups interact with external hardware or other software systems; and which groups perform secondary functions for administration and maintenance.
- Organize actors that are similar to one another in a generalization/specialization hierarchy.
- Where it aids understandability, provide a stereotype for each such actor.
- Populate a use case diagram with these actors and specify the paths of communication from each actor to the system's use cases.
- **Fig: Modeling the Context of a System – Credit card validation system**

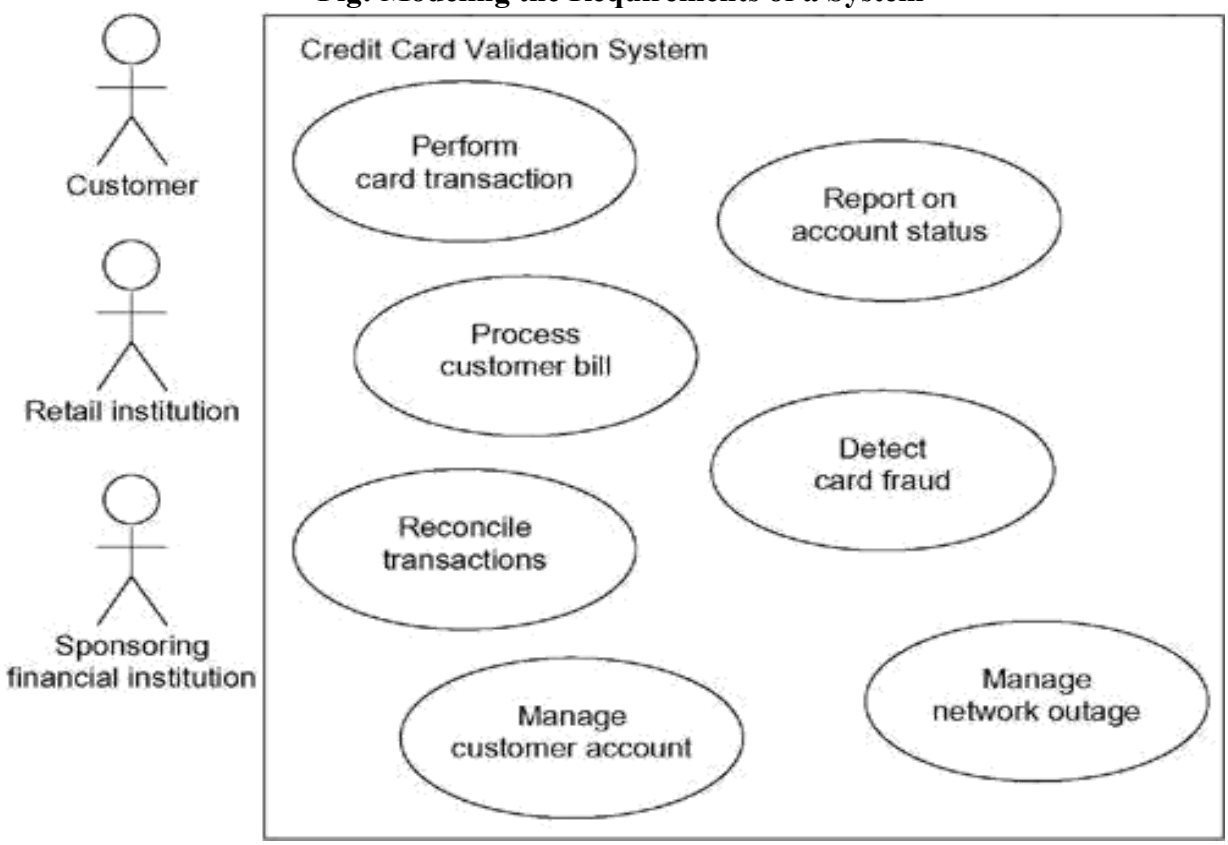


2. Modeling the Requirements of a System

To model the requirements of a system,

- Establish the context of the system by identifying the actors that surround it.
- For each actor, consider the behavior that each expects or requires the system to provide.
- Name these common behaviors as use cases.
- Factor common behavior into new use cases that are used by others; factor variant behavior into new use cases that extend more main line flows.
- Model these use cases, actors, and their relationships in a use case diagram.
- Adorn these use cases with notes that assert nonfunctional requirements; you may have to attach some of these to the whole system.

Fig: Modeling the Requirements of a System



3. Forward and Reverse Engineering

Forward engineering is the process of transforming a model into code through a mapping to an implementation language

To forward engineer a use case diagram

- For each use case in the diagram, identify its flow of events and its exceptional flow of events.
- Depending on how deeply you choose to test, generate a test script for each flow, using the flow's preconditions as the test's initial state and its postconditions as its success criteria.
- As necessary, generate test scaffolding to represent each actor that interacts with the use

case. Actors that push information to the element or are acted on by the element may either be simulated or substituted by its real-world equivalent.

- Use tools to run these tests each time you release the element to which the use case diagram applies.

To reverse engineer a use case diagram,

- Identify each actor that interacts with the system.
- For each actor, consider the manner in which that actor interacts with the system, changes the state of the system or its environment, or responds to some event.
- Trace the flow of events in the executable system relative to each actor. Start with primary flows and only later consider alternative paths.
- Cluster related flows by declaring a corresponding use case. Consider modeling variants using extend relationships, and consider modeling common flows by applying include relationships.
- Render these actors and use cases in a use case diagram, and establish their relationships.

Activity Diagrams

- Activity diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems.
- An *activity diagram* shows the flow from activity to activity.
- Activity diagrams can use to model the dynamic aspects of a system. It involves modeling the sequential (and possibly concurrent) steps in a computational process.
- With an activity diagram, you can also model the flow of an object as it moves from state to state at different points in the flow of control.

Activity diagrams commonly contain

- Activity states and action states
- Transitions
- Objects

Action States and Activity States

- Action states are atomic and cannot be decomposed
  - 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.Work of the action state is not interrupted.
- Activity states can be further decomposed
  - Their activity being represented by other activity diagrams
  - Activity states are not atomic, meaning that they may be interrupted and, in general, are considered to take some duration to complete. They may be interrupted.

Fig: Action States

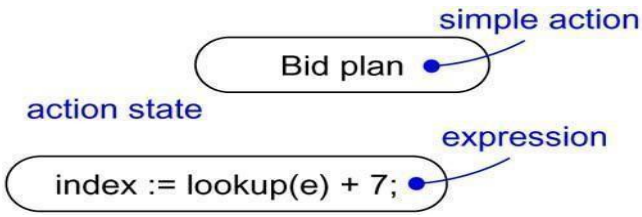
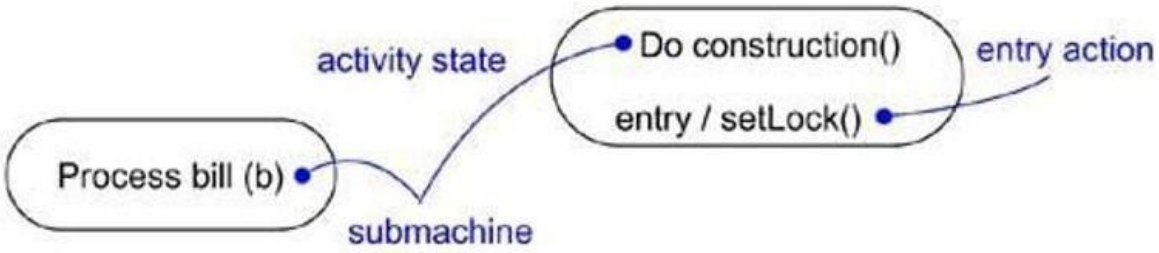
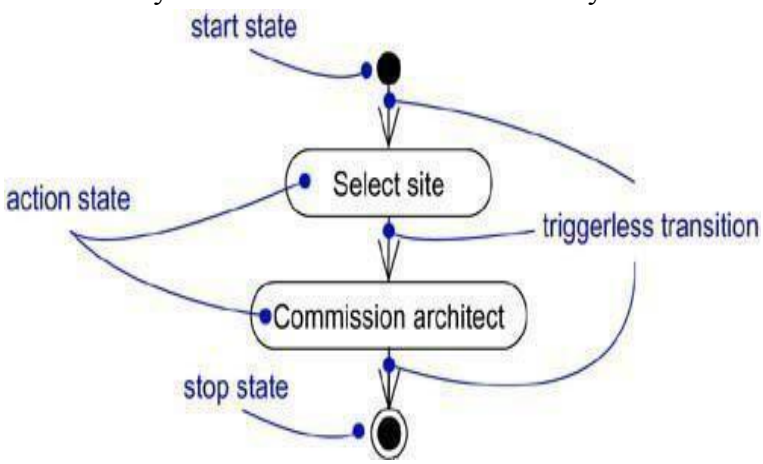


Fig: Activity States



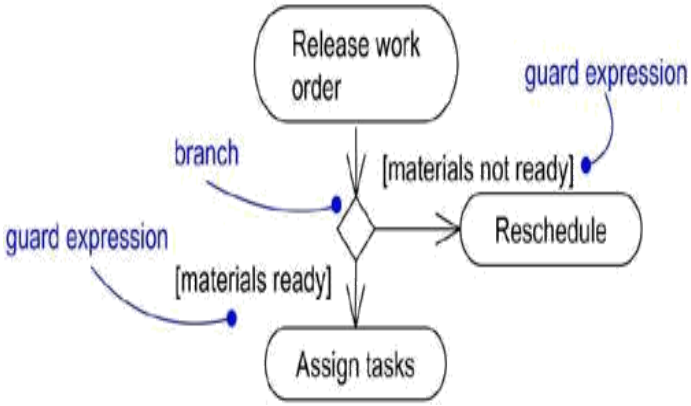
Transitions

- Triggerless transitions may have guard conditions, meaning that such a transition will fire only if that condition is met; guard conditions.
- 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

- *Branches are a notational convenience, semantically equivalent to multiple transitions with guards.*
- *Include a branch, which specifies alternate paths taken based on some Boolean expression.*
- *A branch may have one incoming transition and two or more outgoing ones.*
- *On each outgoing transition, place a Boolean expression, which is evaluated only once on entering the branch.*



Forking and Joining

- Use a synchronization bar to specify the forking and joining of parallel flows of control
- A synchronization bar is rendered as a thick horizontal or vertical line.

Fork

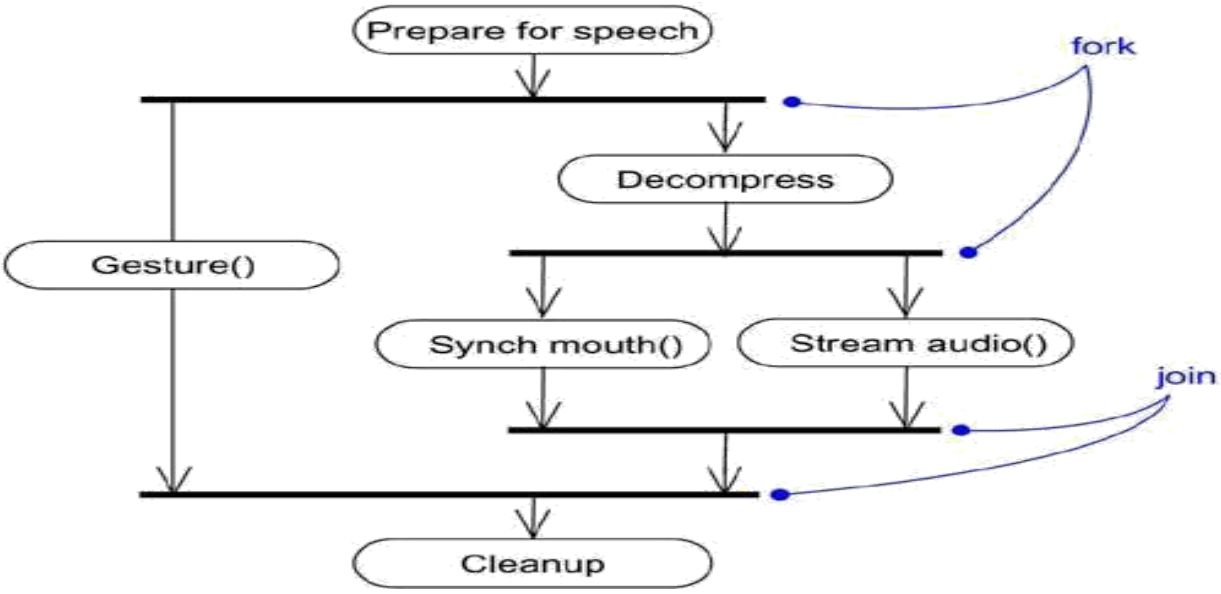
- A fork may have one incoming transitions and two or more outgoing transitions
  - each transition represents an independent flow of control
  - conceptually, the activities of each of outgoing transitions are concurrent
    - either truly concurrent (multiple nodes)
    - or sequential yet interleaved (one node)

Join

- 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
    - each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join

For example, consider the concurrent flows involved in controlling an audio-animatronic device that mimics human speech and gestures.

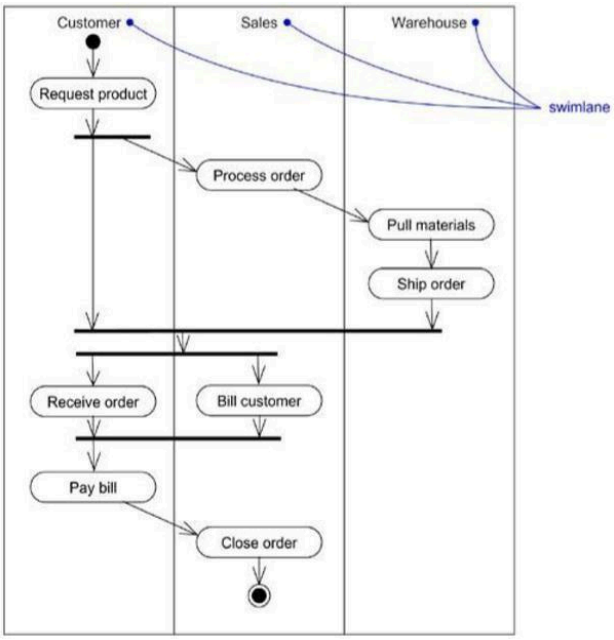
Figure - Forking and Joining



Swimlanes

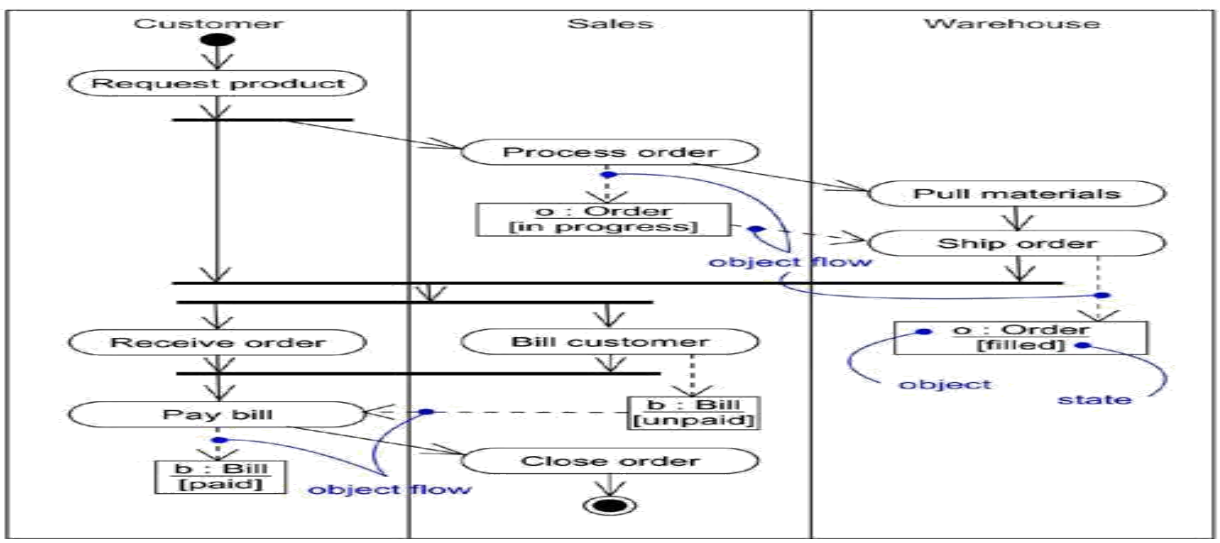
- In the UML, each group is called a swimlane because, visually, each group is divided from its neighbor by a vertical solid line.
- A swimlane specifies a locus of activities.
- Each swimlane has a name unique within its diagram. A swimlane really has no deep semantics, except that it may represent some real-world entity.
- Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram.
- 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.

Fig: Swimlanes



Object Flow

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. As shown in the figure, you represent the state of an object by naming its state in brackets below the object's name. Similarly, you can represent the value of an object's attributes by rendering them in a compartment below the object's name.



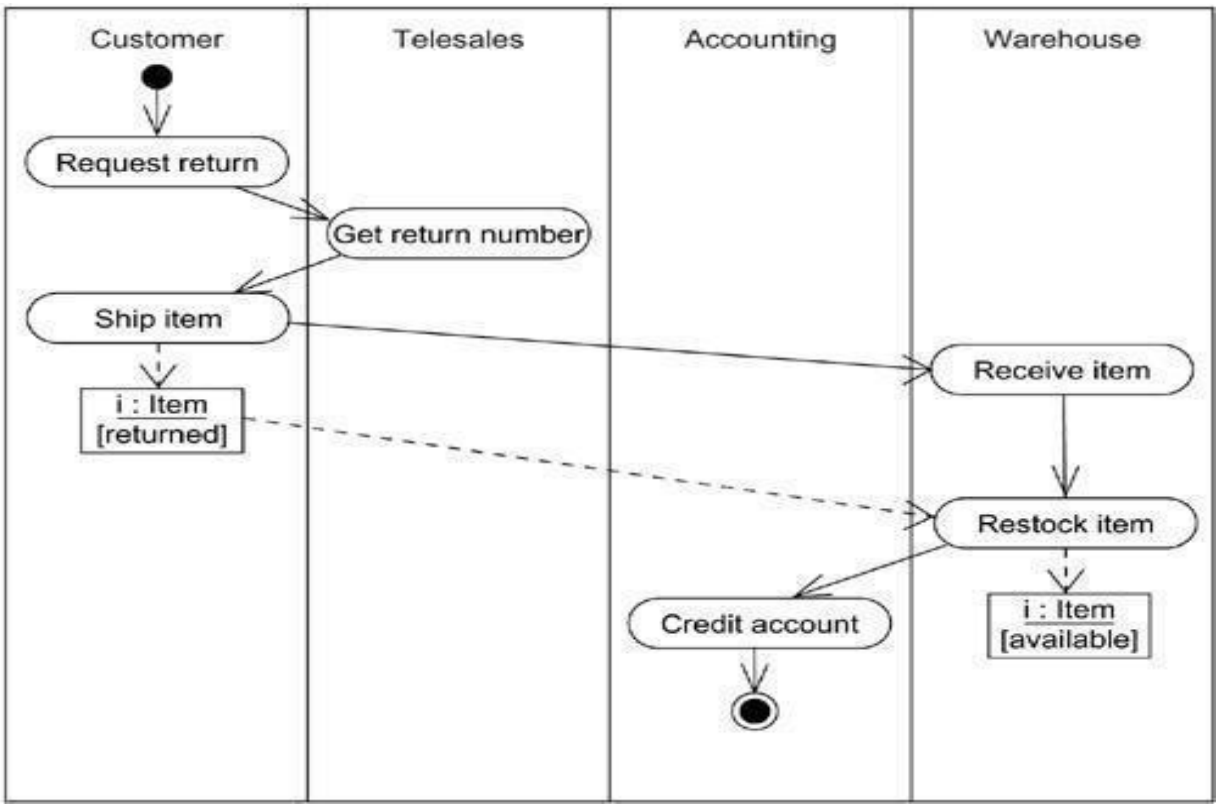
Common Modeling Techniques

1. 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.

Figure : Modeling a Workflow



2. **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.

Figure : Modeling an Operation