MODULE - 2

UNIFIED MODELING LANGUAGE

UML Views & Diagrams

There are five different views that the UML aims to visualize through different modeling diagrams. These five views are:

1. User's View
2. Structural Views
3. Behavioral Views
4. Environmental View
5. Implementation View

Now, these views just provide the thinking methodology and expectations (that people have formed the software) of different groups of people. However, we still need to diagrammatically document the system requirements, and this is done through 9 different types of UML diagrams. These diagrams are divided into different categories. These categories are nothing else but these views that we mentioned earlier.

The different software diagrams according to the views they implement are:

1) User's view

This contains the diagrams in which the user's part of interaction with the software is defined. No internal working of the software is defined in this model. The diagrams contained in this view are:

- Use case Diagram

2) Structural view

In the structural view, only the structure of the model is explained. This gives an estimate of what the software consists of. However, internal working is still not defined in this model. The diagram that this view includes are:

- Class Diagrams
- Object Diagrams

3) Behavioral view

The behavioral view contains the diagrams which explain the behavior of the software. These diagrams are:

- Sequence Diagram
- Collaboration Diagram
- State chart Diagram
- Activity Diagram

4) Environmental view

The environmental view contains the diagram which explains the after deployment behavior of the software model. This diagram usually explains the user interactions and software effects on the system. The diagrams that the environmental model contain are:

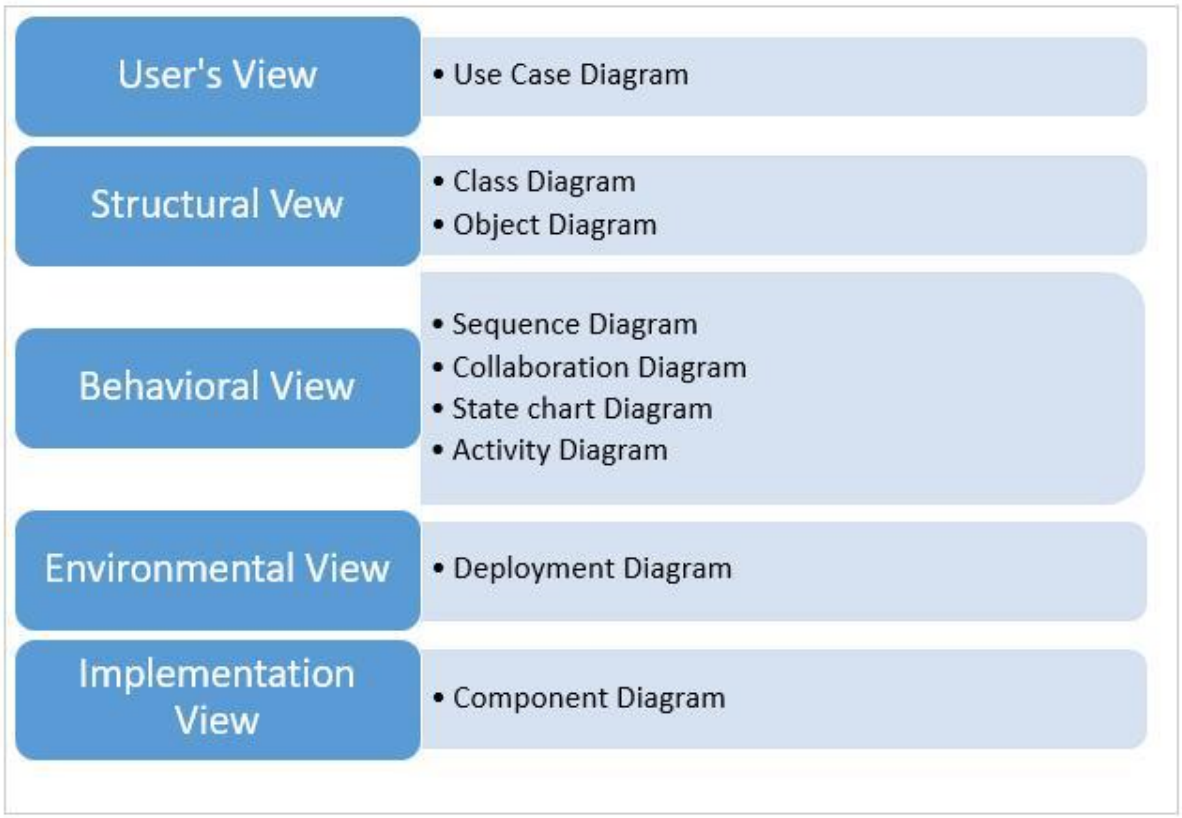
- Deployment diagram

5) Implementation view

The implementation view consists of the diagrams which represent the implementation part of the software. This view is related to the developer’s views. This is the only view in which the internal workflow of the software is defined. The diagram that this view contains is as follows:

- Component Diagram

The following diagram well explains the distribution of these 9 software diagrams according to the five mentioned views:



Object and Class Concepts

Objects--

The purpose of class modeling is to describe objects. For example , ***Joe Smith. Simplex company, process number 7648*** and ***the top window*** are objects.

An ***object*** is a concept, abstraction, or thing with identity that has meaning for an application.

Classes--

An object is an ***instance*** of a class. A ***class*** describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships, and semantics. ***Person, company, process,*** and ***window*** are all classes.

Class Diagrams--

There are two kinds of models of structure—classes diagrams and object diagrams.

Class diagrams provide a graphic notation for modeling classes and their relationships, thereby describing possible objects. Class diagrams are useful both for abstract modeling and for designing actual programs. An ***object diagram*** shows individual objects and their relationships. Figure shows a class (left) and instances (right) described by it.

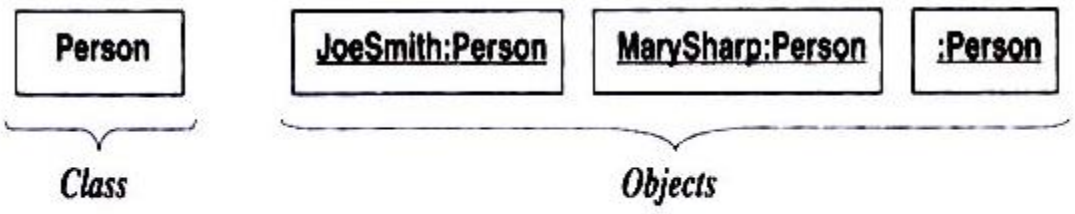


Figure **A class and objects.** Objects and classes are the focus of class modeling.

Values and Attributes--

A ***value*** is a piece of data. An ***attribute*** is a named property of a class that describes a value held by each object of the class. ***Name, birthdate,*** and ***weight*** arc attributes of ***Person*** objects. ***Color, model Year,*** and ***weight*** are attributes of ***Car*** objects. Each attribute has a value for each object.

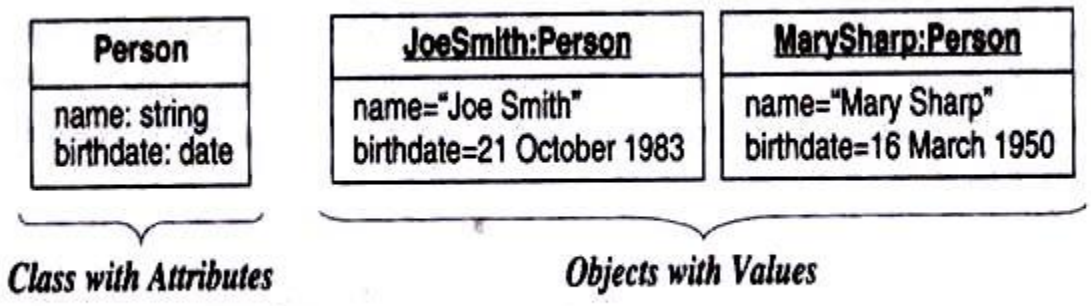


Figure **Attributes and values.** Attributes elaborate classes.

Operations and Methods--

An ***operation*** is a function or procedure that may be applied to or by objects in a class. ***Hire, fire,*** and ***payDividend*** are operations on class ***Company. Open, close, hide,*** and ***redisplay*** are operations on class ***Window.*** A ***method*** is the implementation of an operation for a class. For example, the class ***File*** may have an operation ***print.***

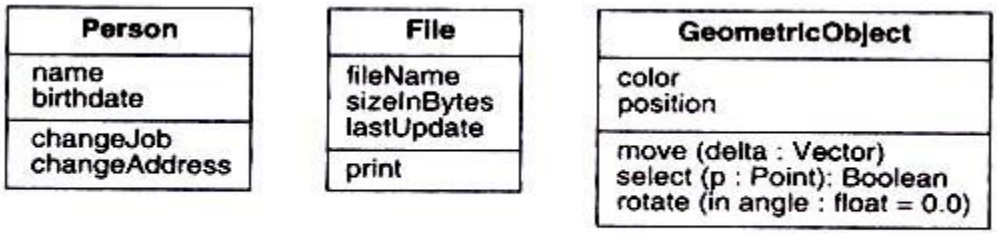


Figure **Operations.** An operation is a function or procedure that may be applied to or by objects in a class.

UML Class Notation--

A class represent a concept which encapsulates state (**attributes**) and behavior (**operations**). Each attribute has a type. Each **operation** has a **signature**. *The class name is the* ***only mandatory information***.

Class Name:

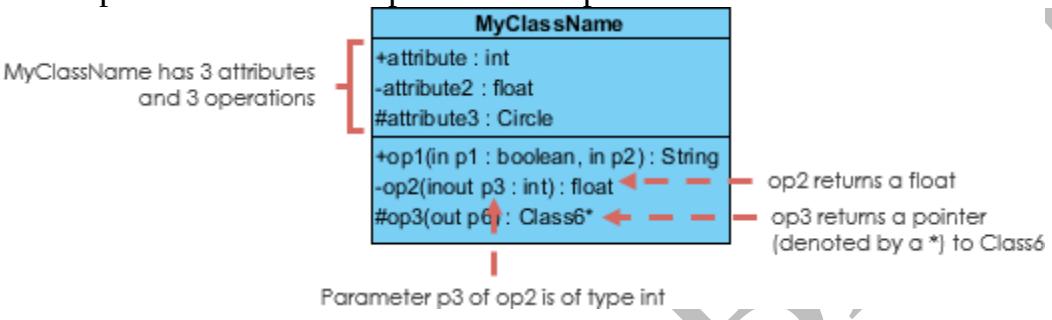
- The name of the class appears in the first partition.

Class Attributes:

- Attributes are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

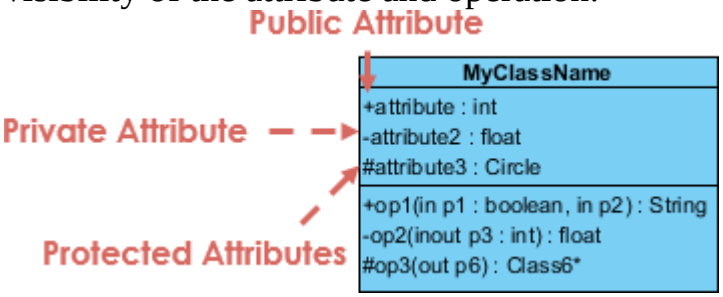
Class Operations (Methods):

- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code



Class Visibility

The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



- \+ denotes public attributes or operations
- \- denotes private attributes or operations
- # denotes protected attributes or operations

Links and Associations--A ***link*** is a physical or conceptual connection among objects. For example, Joe Smith ***Works-For*** Simplex company. A link is an instance of an association. An ***association*** is a description of a group of links with common structure and common semantics. For example ,a person ***WorksFor*** a company.

Multiplicity--***Multiplicity*** specifies the number of instances of one class that may relate to a single instance of an associated class. Multiplicity constrains the number of related objects. The UML specifies multiplicity with an interval, such as 1" (exactly one), "1...*" (one or

more), or "3...5" (three to five, inclusive). The special symbol "*" is a shorthand notation that denotes "many" (zero or more).

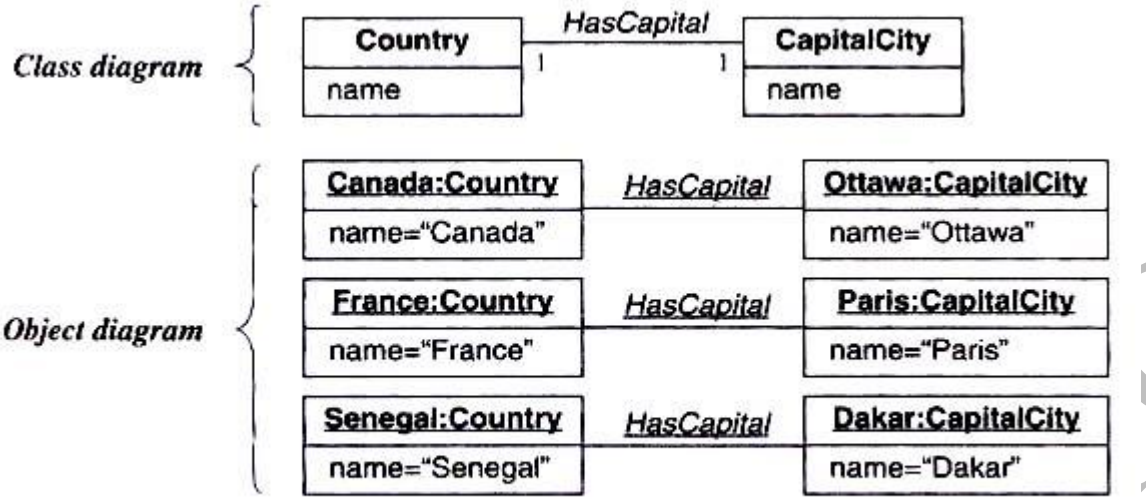
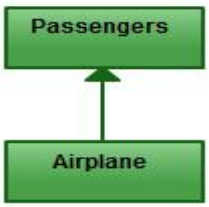


Figure **One-to-one association.** Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.

Figure illustrates zero-or-one multiplicity. A workstation may have one of its windows designated as the console to receive general error messages.

Figure **Zero-or-one multiplicity**. It may be optional whether an object is involved in an association.

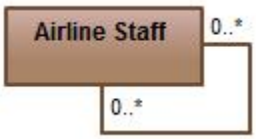
Directed Association



Directed Association

It refers to a directional relationship represented by a line with an arrowhead. The arrowhead depicts a container-contained directional flow.

Reflexive Association



Reflexive Association

This occurs when a class may have multiple functions or responsibilities. For example, a staff member working in an airport may be a pilot, aviation engineer, a ticket dispatcher, a guard, or a maintenance crew member. If the maintenance crew member is managed by the

aviation engineer there could be a managed by relationship in two instances of the same class.

Association End Names--The name given to the association end. In the figure ***Person*** and ***Company*** participate in association ***WorksFor.*** A person is an ***employee*** with respect to a company; a company is an ***employer*** with respect to a person. Use of association end names is optional.

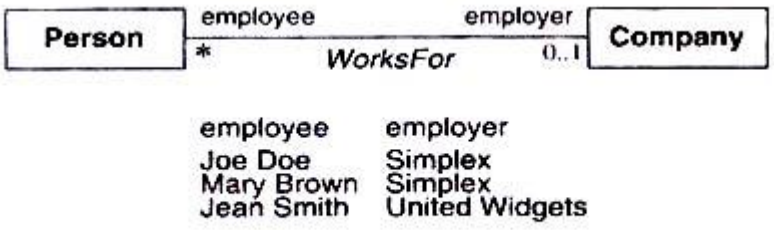


Figure **Association end names**. Each end of an association can have a name.

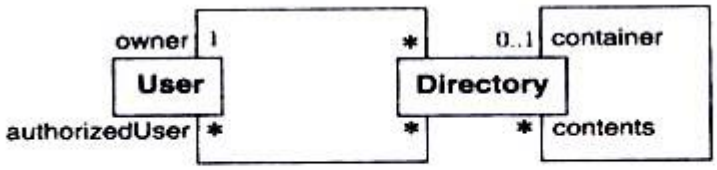


Figure **Association end names**. Association end names are necessary for associations between two objects of the same class. They can also distinguish multiple associations between a pair of classes.

Ordering--

The objects have an explicit order. The ordering is an inherent part of the association. We can indicate an ordered set of objects by writing "{ordered}" next to the appropriate association end.

Figure **Ordering the objects for an association end**. Ordering sometimes occurs for "many" multiplicity.

Bags and Sequences--

A ***bag*** is a collection of elements with duplicates allowed. A ***sequence*** is an ordered collection of elements with duplicates allowed. In Figure an itinerary is a sequence of airports and the same airport can be visited more than once.

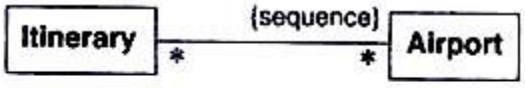


Figure **An example of a sequence**. An itinerary may visit multiple airports, so you should use *{sequence}* and not *{ordered}*.

Association Classes—

An ***association class*** is an association that is also a class. In Figure ***accessPermission*** is an attribute of ***AccessibleBy.*** The sample data at the bottomn of the figure shows the value for each link. The UML notation for an association class is a box (a class box) attached to the association by a dashed line.

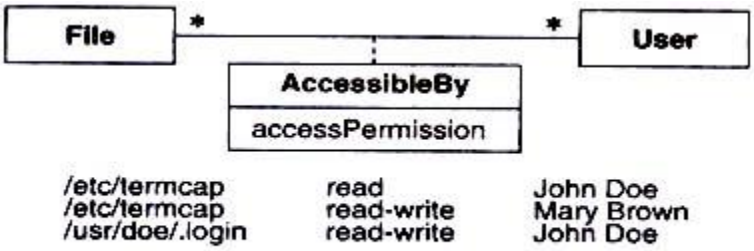


Figure **An association class.** The links of an association can have attributes.

Qualified Associations--

A ***qualified association*** is an association in which an attribute called the ***qualifier*** disambiguates the objects for a "many" association end. It is possible to define qualifiers for one-to-many and many-to-many associations. ***Bank*** and ***Account*** are classes and ***accountNumber*** is the qualifier. Qualification reduces the effective multiplicity of this association from one-to-many to one-to-one.

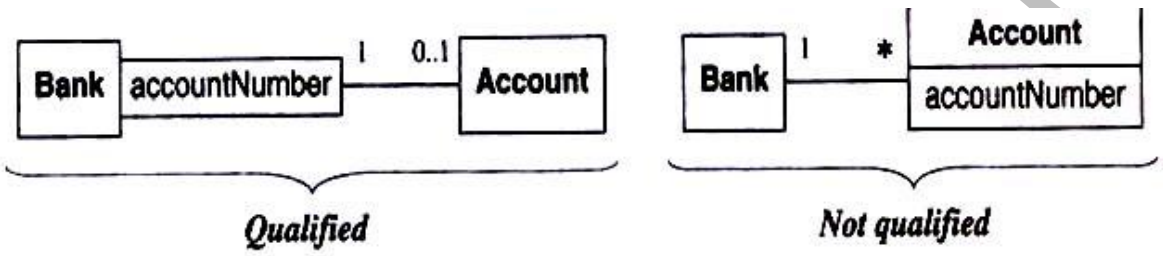


Figure **Qualified association.** Qualification increases the precision of a model.

N-ary Associations--

we may occasionally encounter ***n-ary associations*** (associations among three or more classes.)

Figure shows a genuine n-ary (ternary) association: Programmers use computer languages on projects. This n-ary association is an atomic unit and cannot he subdivided into binary associations without losing information. A programmer may know a language and work on a project, but might not use the language on the project. The UML symbol for n-ary associations is a diamond with lines connecting to related classes. If the association has a name, it is written in italics next to the diamond.

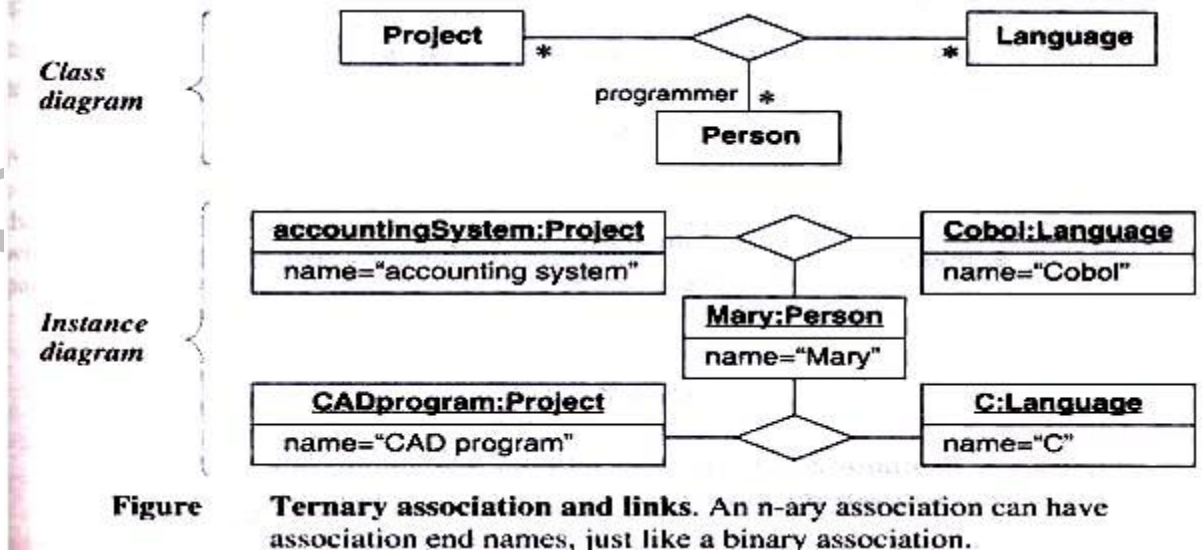


Figure **Ternary association and links.** An n-ary association can have association end names, just like a binary association.

Generalization and Inheritance--

Generalization is the relationship between a class (the ***superclass)*** and one or more variations of the class (the ***subclasses).*** Generalization organizes classes by their similarities and differences, structuring the description of objects. The superclass holds common attributes, operations, and associations; the subclasses add specific attributes, operations, and associations. Each subclass is said to ***inherit*** the features of its superclass. Generalization is sometimes called the "is-a" relationship, because each instance of a subclass is an instance of the superclass as well.

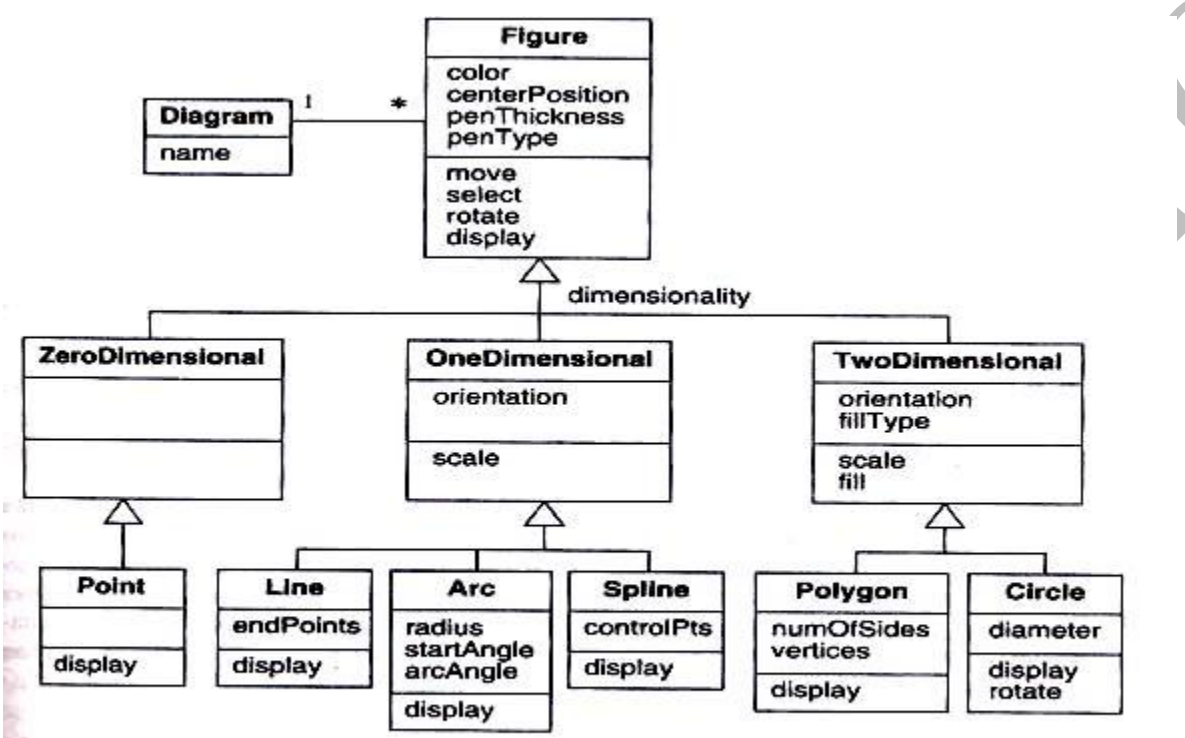
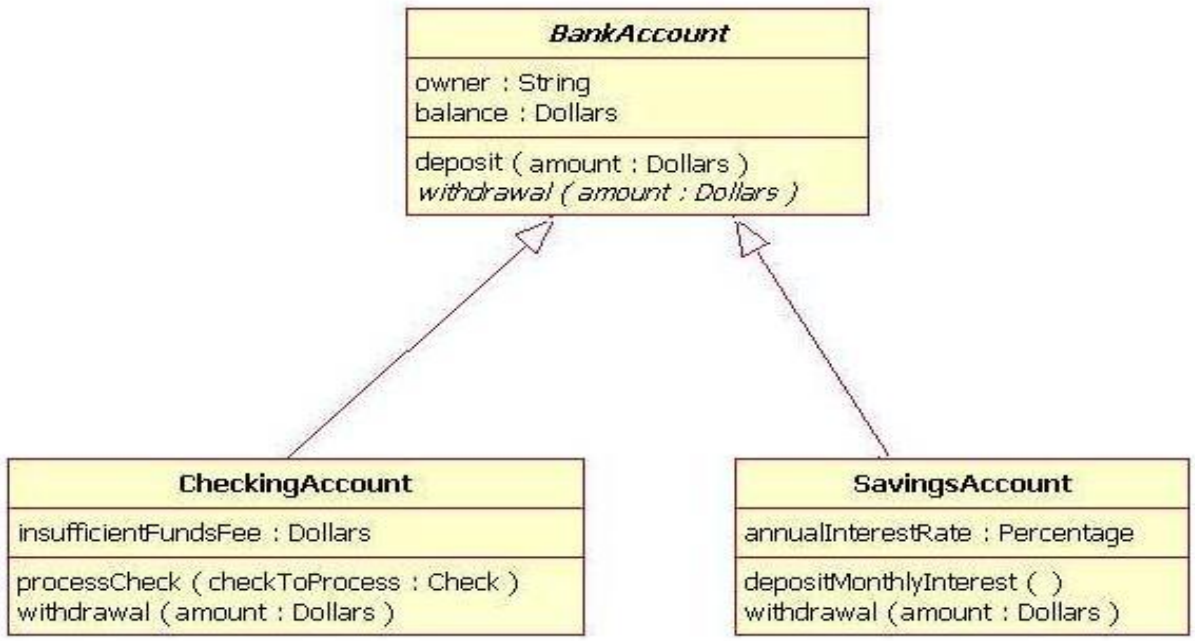


Figure **Inheritance for graphic figures.** Each subclass inherits the attributes, operations, and associations of its superclasses.



Fig—Inheritance example

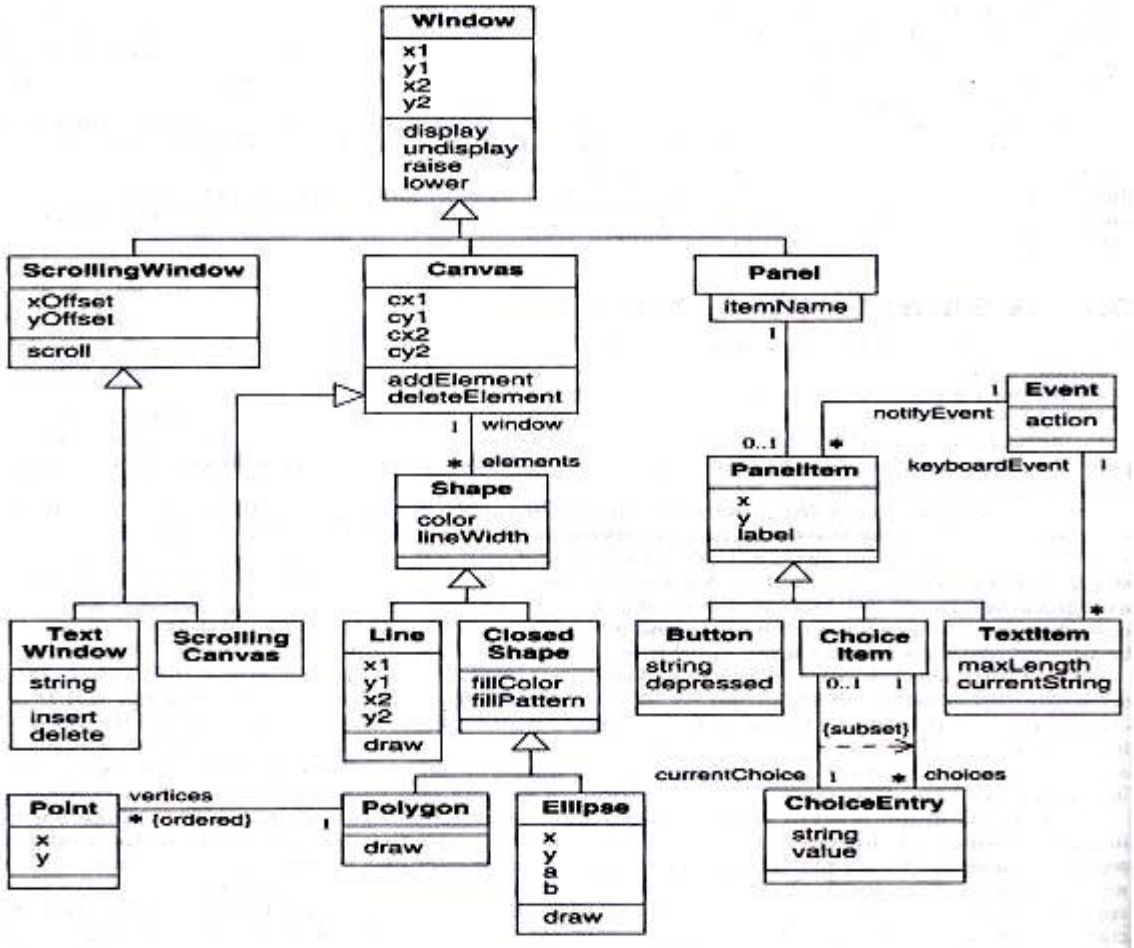


Figure **Class model of a windowing system**

Multiple Inheritance--

Multiple inheritance permits a class to have more than one superclass and to inherit features from all parents. The advantage of multiple inheritance is greater power in specifying classes and an increased opportunity for reuse. The disadvantage is a loss of conceptual and implementation simplicity.

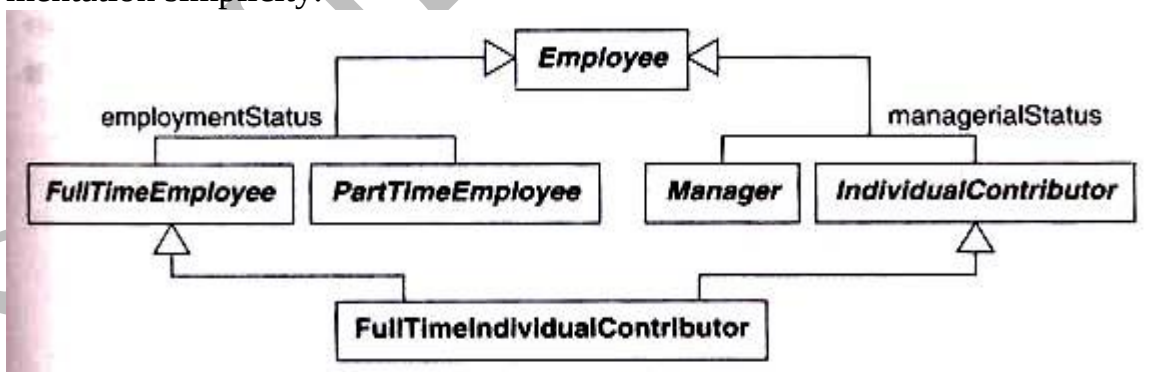


Figure **Multiple inheritance from disjoint classes.** This is the most common form of multiple inheritance.

Multiple inheritance can also occur with overlapping classes. In Figure ***Amphibi-onsVehicle*** is both ***LandVehicle*** and ***WaterVehicle. LandVehicle*** and ***WaterVehicle*** overlap, because some vehicles travel on both land and water. The UML uses a constraint to indicate an

overlapping generalization set; the notation is a dotted line cutting across the affected generalizations with keywords in braces.

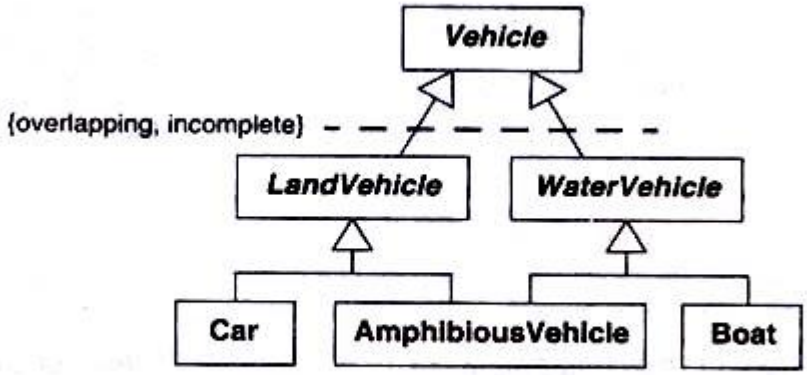


Figure **Multiple inheritance from overlapping classes**. This form of multiple inheritance occurs less often than with disjoint classes.

Realization

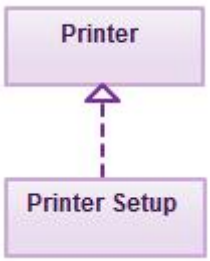


Fig--Realization

denotes the implementation of the functionality defined in one class by another class. To show the relationship in UML, a broken line with an unfilled solid arrowhead is drawn from the class that defines the functionality of the class that implements the function. In the example, the printing preferences that are set using the printer setup interface are being implemented by the printer.

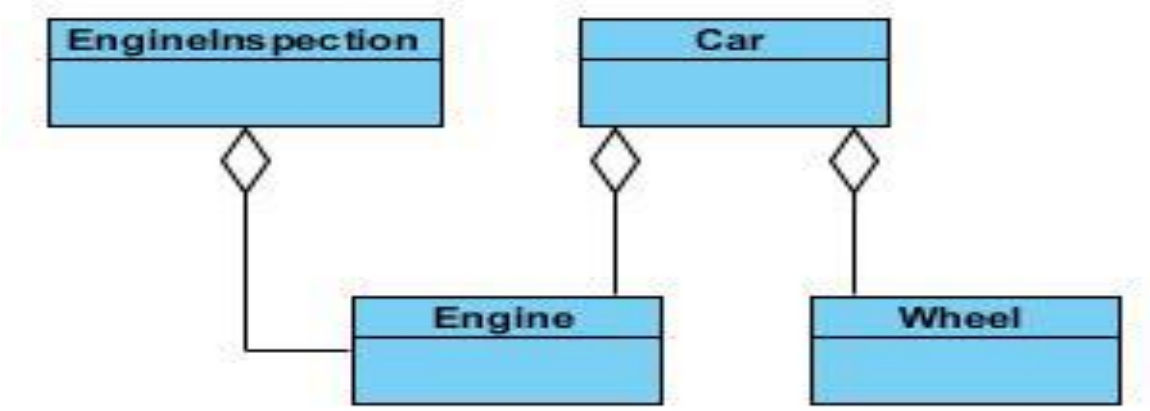
Aggregation--

Aggregation as relation of an assembly class to ***one*** constituent part class. An assembly with many kinds of constituent parts corresponds to many aggregationsThis definition emphasizes that aggregation is a special form of binary association.

The most significant property of aggregation is ***transitivity***— that is, if A is part of ***B*** and ***B*** is part of ***C,*** then ***A*** is part of ***C.*** Aggregation is also ***antisymmetric***—that is, if A is part of ***B,*** then is not part of ***A.***

Examples

Examples--



Aggregation

Composition--

Composition is a form of aggregation with two additional constraints. A constituent part can belong to at most one assembly. It has a coincident lifetime with the assembly. Thus composition implies ownership of the parts by the whole.

In Figure a company consists of divisions, which in turn consist of departments: a company is indirectly a composition of departments. A company is not a composition of its employees, since company and person are independent objects of equal stature.

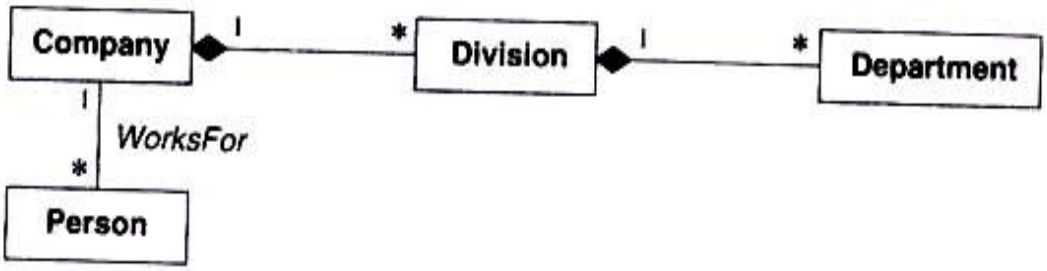
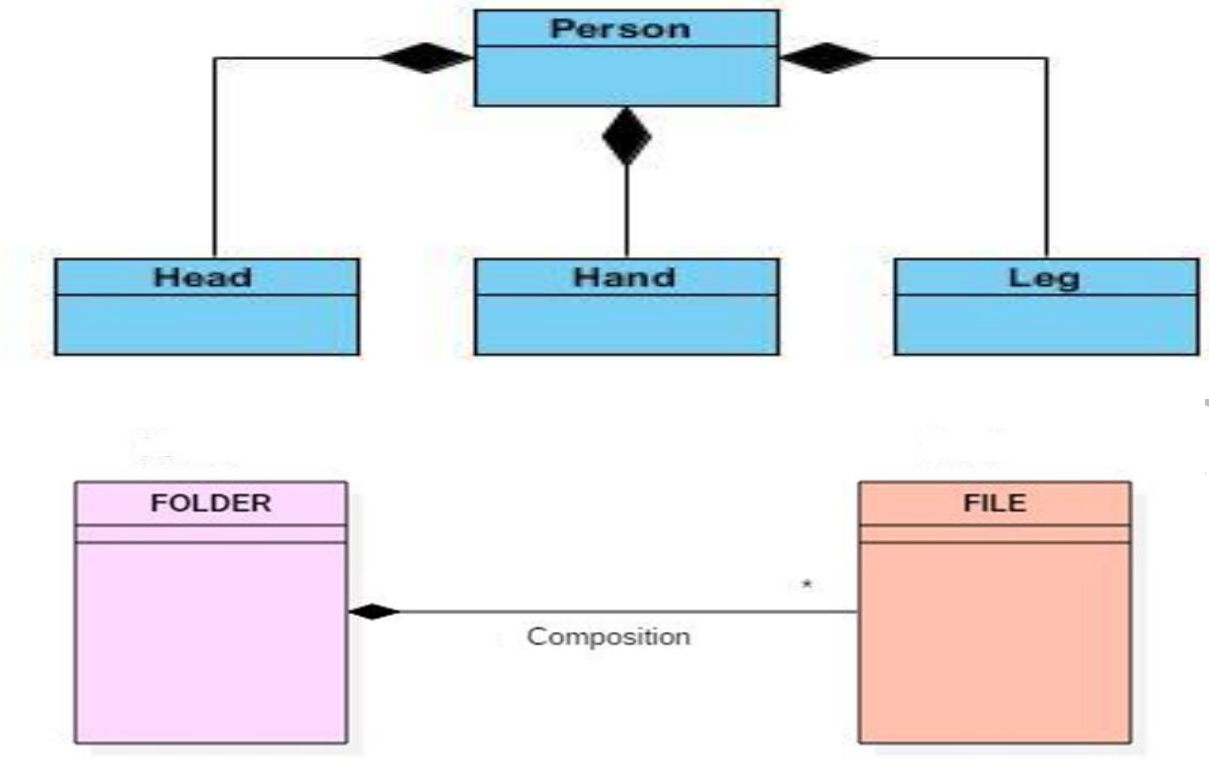


Figure **Composition.** With composition a constituent part belongs to at most one assembly and has a coincident lifetime with the assembly.

Examples--



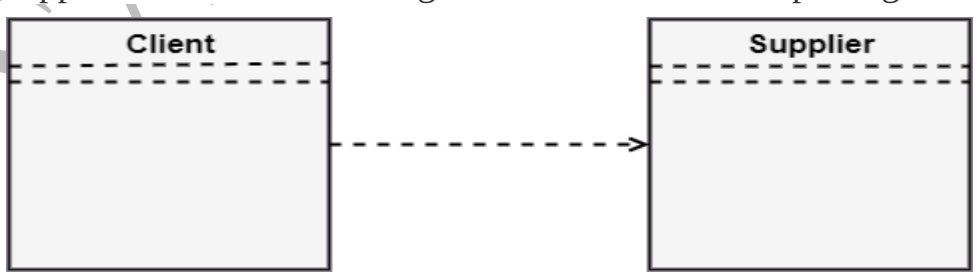
Composition

Packages--A ***package*** is a group of elements (classes, associations, generalizations, and lesser packages) with a common theme. A package partitions a model, making it easier to understand and manage.

Figure **Notation for a package.** Packages let you organize large models so that persons can more readily understand them.

Dependency

Dependency depicts how various things within a system are dependent on each other. In UML, a dependency relationship is the kind of relationship in which a client (one element) is dependent on the supplier (another element). It is used in class diagrams, component diagrams, deployment diagrams, and use-case diagrams, which indicates that a change to the supplier necessitates a change to the client. An example is given below:



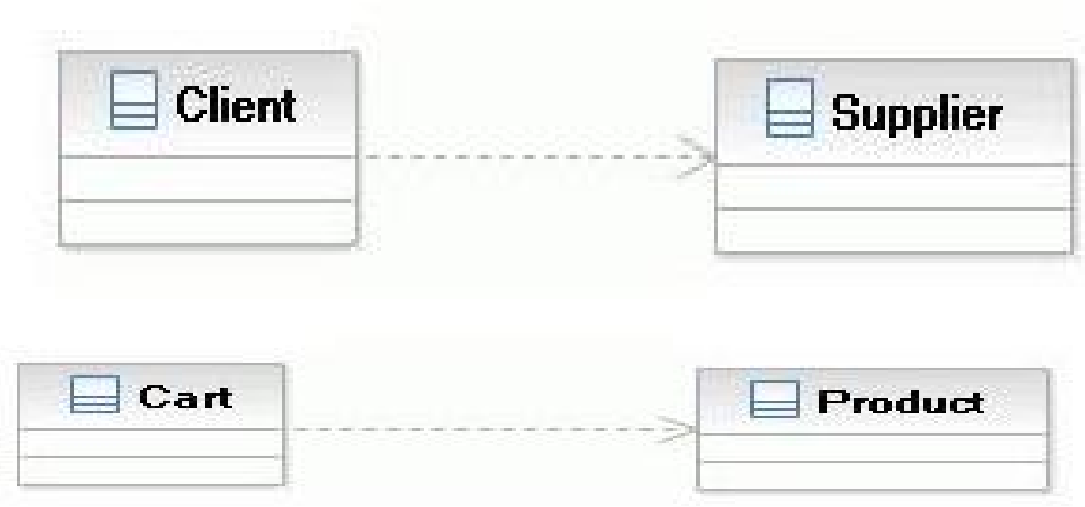
Types of Dependency Relationship

Following are the type of dependency relationships, keywords, or stereotypes given below:

- **<<derive>>** -It is a constraint that specifies the template can be initialized by the source at the target location utilizing given parameters.
- **<<derive>>** -It represents that the source object's location can be evaluated from the target object.
- **<<friend>>** -It states the uniqueness of the source in the target object.
- **<<instanceOf>>** -It states that an instance of a target classifier is the source object.
- **<<instantiate>>** -It defines the capability of the source object, creating instances of a target object.
- **<<refine>>** -It states that the source object comprises of exceptional abstraction than that of the target object.
- **<<use>>** -When the packages are created in UML, the use of stereotype is used as it describes that the elements of the source package can also exist in the target package. It specifies that the source package uses some of the elements of the target package.
- **<<substitute>>** -The substitute stereotype state that the client can be substituted at the runtime for the supplier.
- **<<access>>** -It is also called as private merging in which the source package accesses the element of the target package.
- **<<import>>** -It specifies that target imports the source package's element as they are defined within the target. It is also known as public merging.
- **<<permit>>** -It describes that the source element can access the supplier element or whatever visibility is provided by the supplier.
- **<<extend>>** -It states that the behavior of the source element can be extended by the target.
- **<<include>>** -It describes the source element, which can include the behavior of another element at a specific location, just like a function call in C/C++.
- **<<become>>** -It states that target is similar to the source with distinct roles and values.
- **<<call>>** -It specifies that the target object can be invoked by the source.
- **<<copy>>** -It states that the target is an independent replica of a source object.
- **<<parameter>>** -It describes that the supplier is a parameter of the client's actions.

- **<<send>>** -The client act as an operation, which sends some unspecified targets to the supplier.

Examples---



Note

A note is the capability of attaching several remarks to the element. It basically carries useful information for the modelers.

Events--

An event is an occurrence at a point in time, such as *user depresses left button* or *flight 123 departs from Chicago ,power turned on, alarm set, paper tray becomes empty, temperature becomes lower than freezing.*

Two events that are causally unrelated are said to be *concurrent:* they have no effect on each other. Flight 123 must depart Chicago before it can arrive in San Francisco; the two events are causally related. Flight 123 may depart before or after flight 456 departs Rome; the two events are causally unrelated.

There are several kinds of events. The most common are the signal event, the change event, and the time event.

Signal Event--

A *signal* is an explicit one-way transmission of information from one object to another. An object sending a signal to another object may expect a reply, but the reply is a separate signal under the control of the second object, which may or may not choose to send it.

We can give each signal class a name to indicate common structure and behavior. For example,

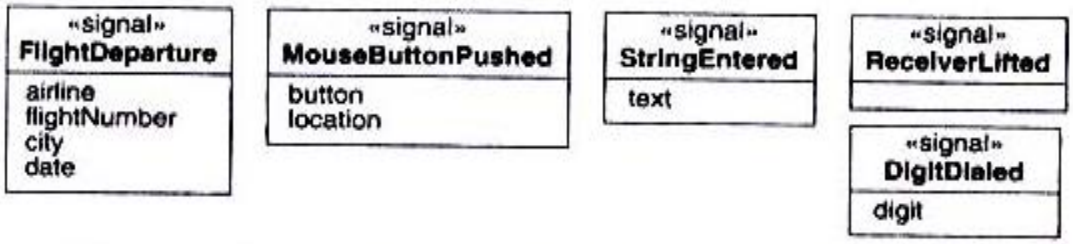


Figure **Signal classes and attributes.** A signal is an explicit one-way transmission of information from one object to another.

Change Event--

A *change event* is an event that is caused by the satisfaction of a boolean expression.

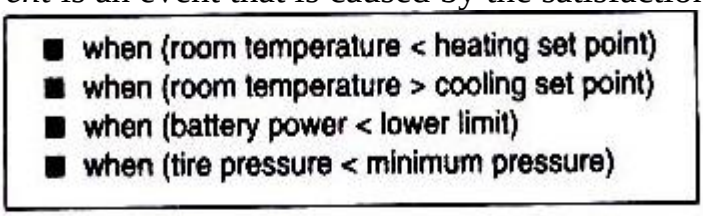


Figure **Change events.** A change event is an event that is caused by the satisfaction of a boolean expression.

Time Event--

A *time event* is an event caused by the occurrence of an absolute time or the elapse of a time interval.

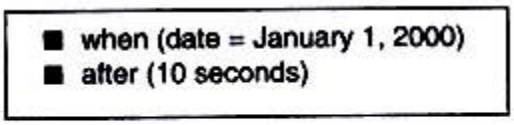


Figure **Time events.** A time event is an event caused by the occurrence of an absolute time or the elapse of a time interval.

States--

A *state* is an abstraction of the values and links of an object. Sets of values and links are grouped together into a state according to the gross behavior of objects. Figure 5.4 shows the UML notation for a state—a rounded box containing an optional state name.

Figure **States.** A state is an abstraction of the values and links of an object.

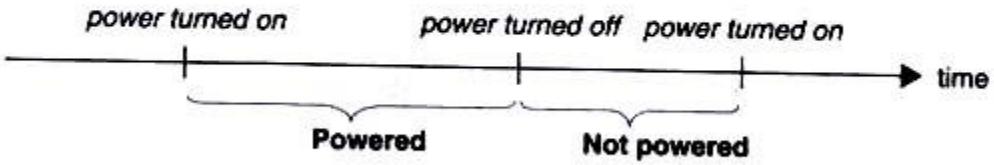
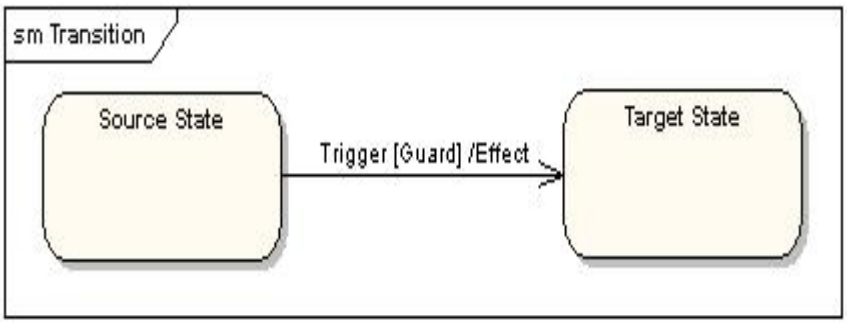


Figure **Event vs. state.** Events represent points in time; states represent intervals of time.

Guard--

Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below.



"Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

A *guard condition* is a boolean expression that must be true in order for a transition to occur.

State Diagrams--

A *state diagram* is a graph whose nodes are states and whose directed arcs are transitions between states. A state diagram specifies the state sequences caused by event sequences.

Sample State Diagram--

Figure shows a state diagram for a telephone line. The diagram concerns a phone line and not the caller nor callee. The diagram contains sequences associated with normal calls as well as some abnormal sequences, such as timing out while dialing or getting busy lines.

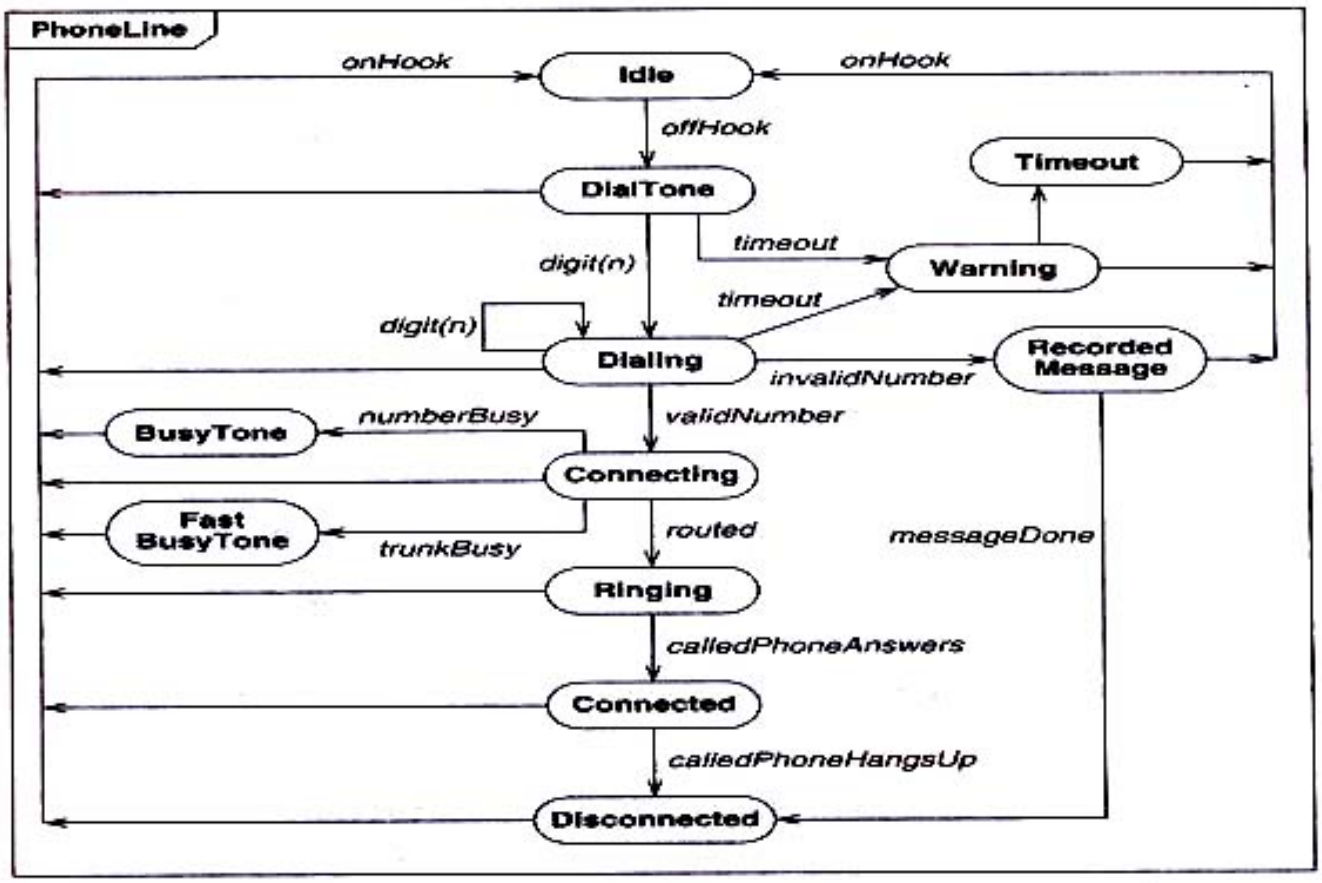


Figure **State diagram for a telephone line.** A state diagram specifies the state sequences caused by event sequences.

One-shot State Diagrams--

One-shot state diagrams represent objects with finite lives and have initial and final states. The initial state is entered on creation of an object; entry of the final state implies destruction of the object. Figure 5.9 shows a simplified life cycle of a chess game with a default initial state (solid circle) and a default final state (bull's eye).

As an alternate notation,We can indicate initial and final states via entry and exit points. In Figure the *start* entry point leads to white's first turn, and the chess game eventually ends with one of three possible outcomes. Entry points (hollow circles) and exit points (circles enclosing an "x") appear on the state diagram's perimeter and may be named.

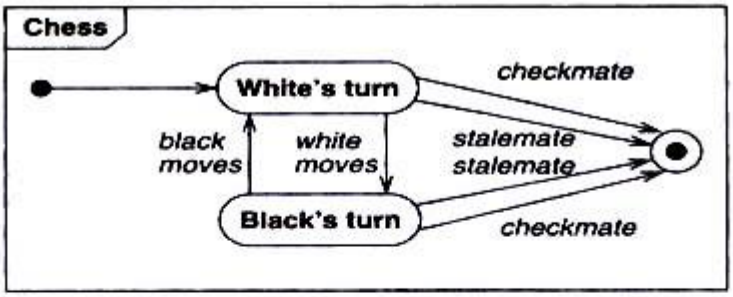


Figure **State diagram for chess game**. One-shot diagrams represent objects with finite lives.

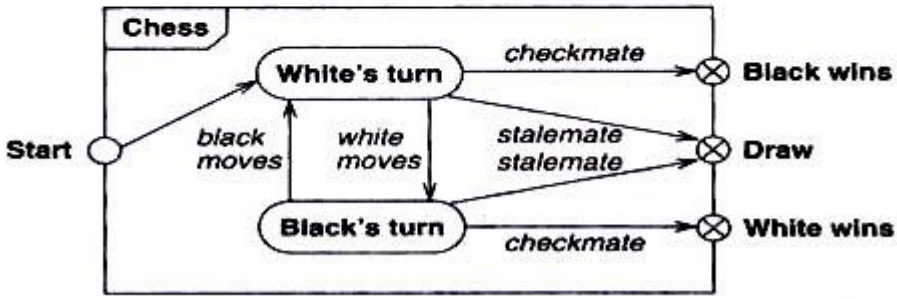


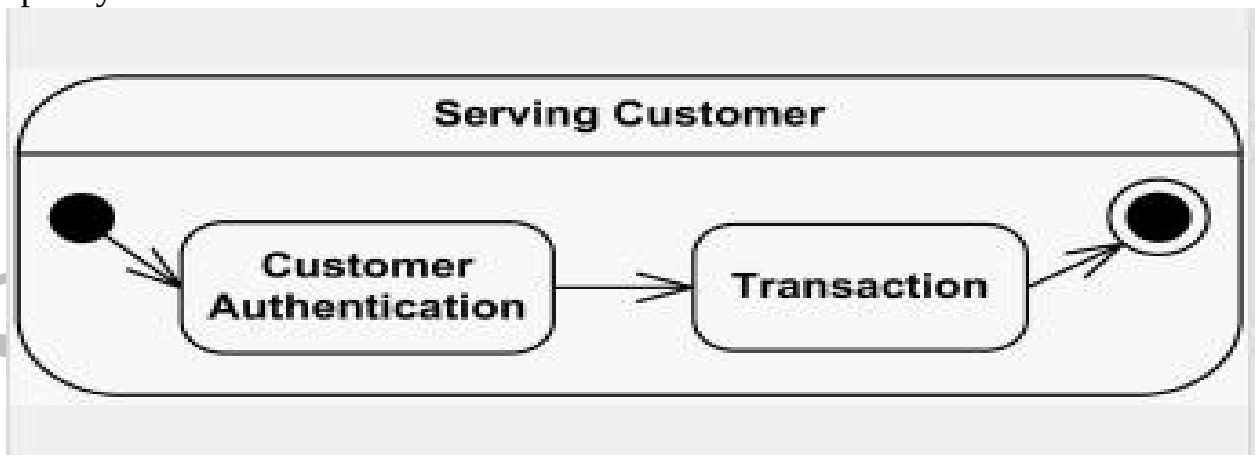
Figure **State diagram for chess game**. You can also show one-shot diagrams by using entry and exit points.

Composite state--

A composite state is a state that contains one or more other substates. It can be used to group states into logical compounds and thus make the statechart more comprehensible.

When a composite state is entered, its entry node denotes the substate to be activated. A composite state can specify multiple entry nodes with unique names. Incoming transitions of the composite state can specify the desired entry node to take for entering the composite state.

When a composite state is left, all active substates are also left. A composite state can specify multiple exit nodes with unique names. Outgoing transitions of the composite state can specify the relevant exit node for them.



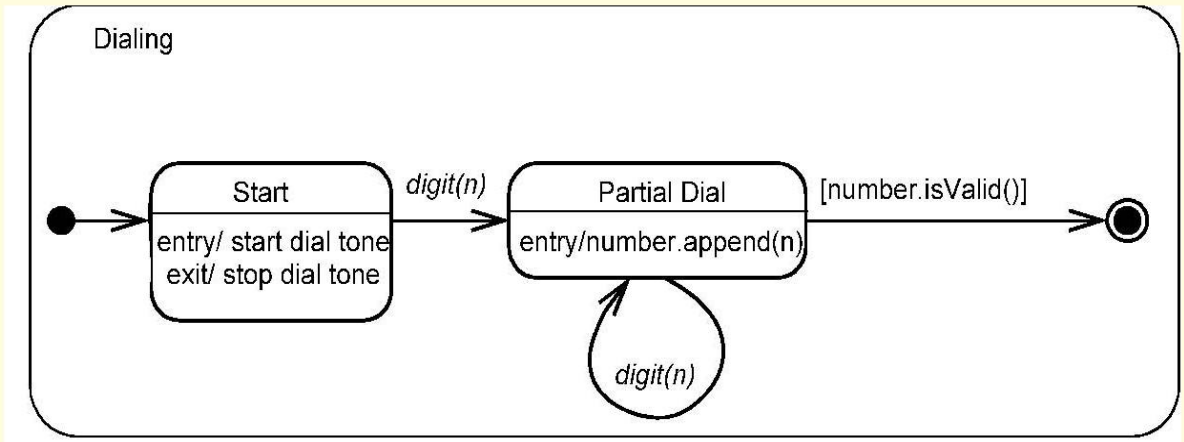


Figure - Composite state with two states

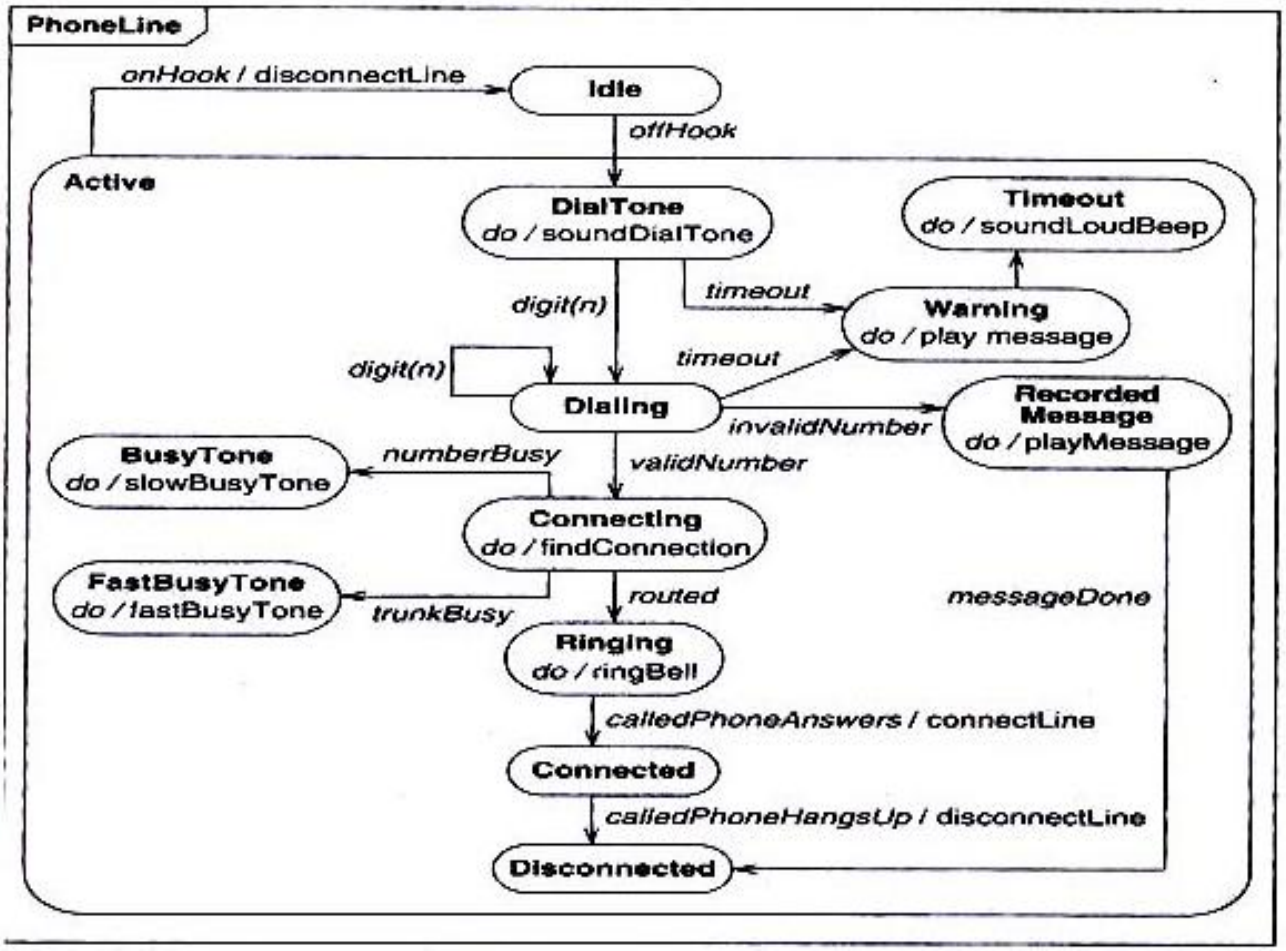
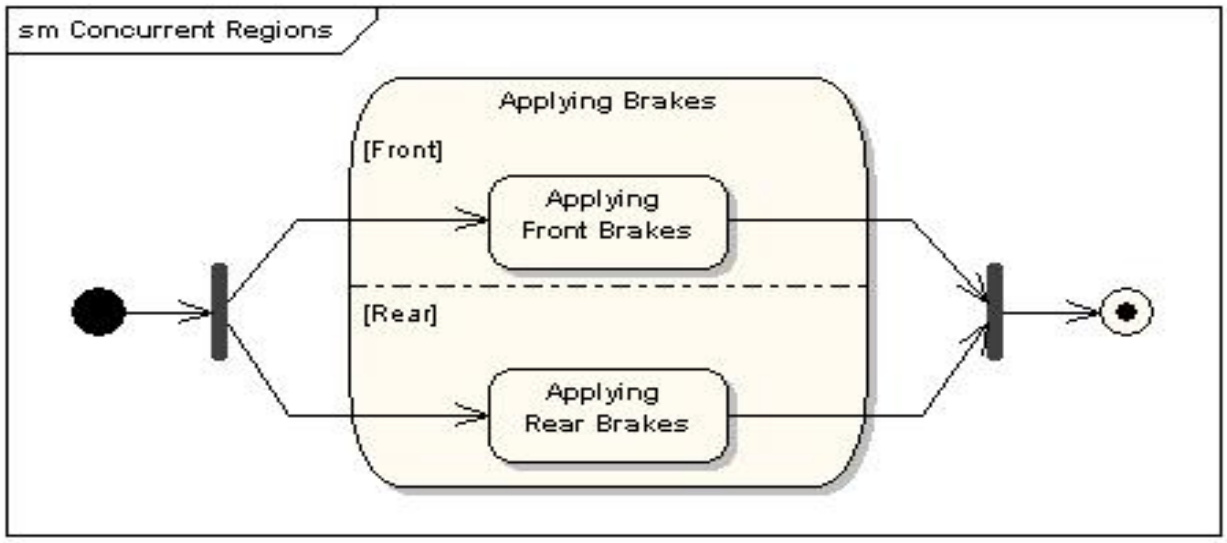


Figure **Nested states for a phone line.** A nested state receives the outgoing transitions of its enclosing state.

Concurrent States--

A state may be divided into regions containing sub-states that exist and execute concurrently. The example below shows that within the state "Applying Brakes", the front and rear brakes will be operating simultaneously and independently. Notice the use of fork and join pseudo-states, rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.



Examples—

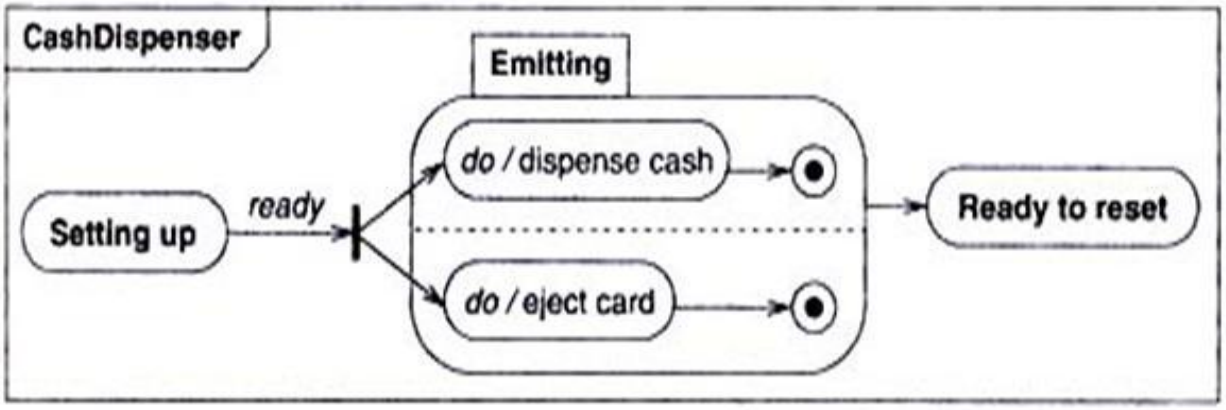
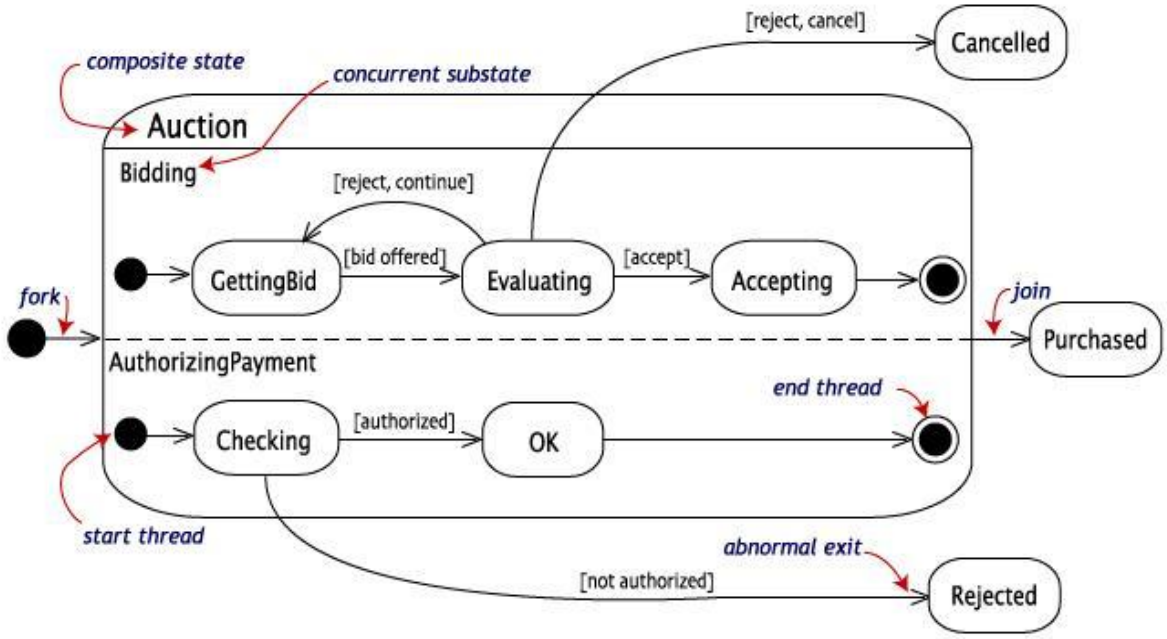
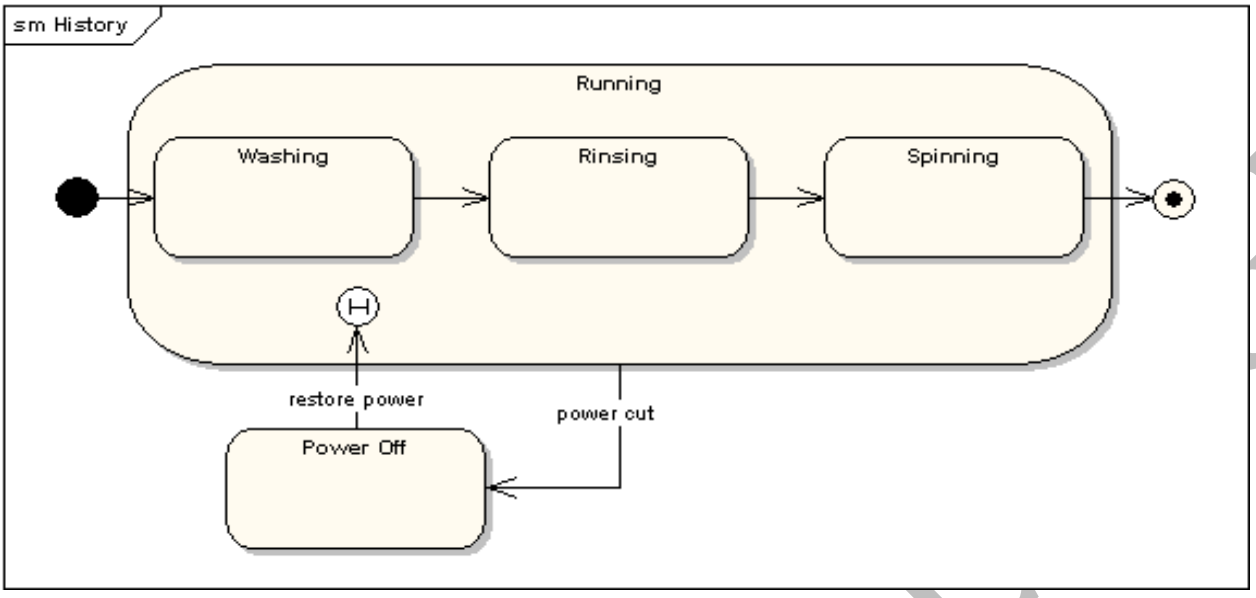


Figure **Synchronization of control.** Control can split into concurrent activities that subsequently merge.



History States--

A history state is used to remember the previous state of a state machine when it was interrupted. The following diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.



In this state machine, when a washing machine is running, it will progress from "Washing" through "Rinsing" to "Spinning". If there is a power cut, the washing machine will stop running and will go to the "Power Off" state. Then when the power is restored, the Running state is entered at the "History State" symbol meaning that it should resume where it last left-off.

Example—

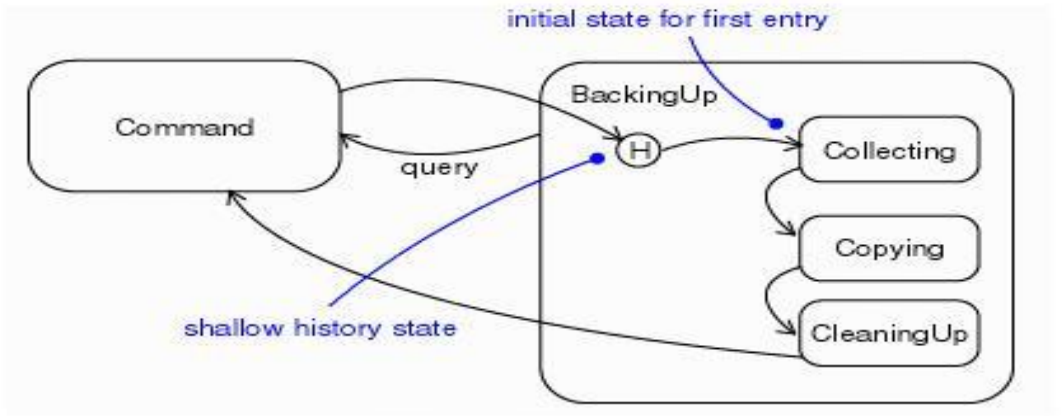
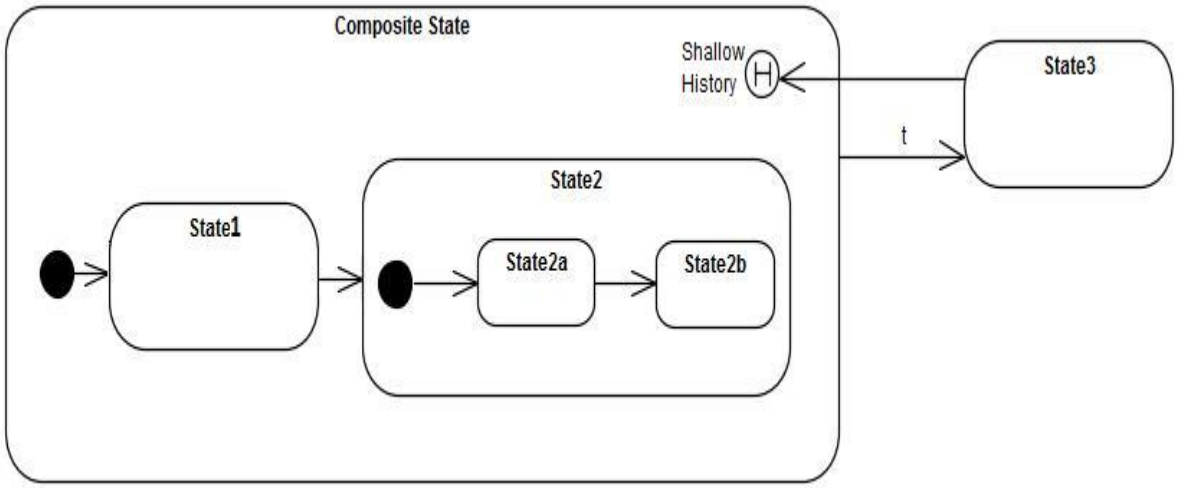


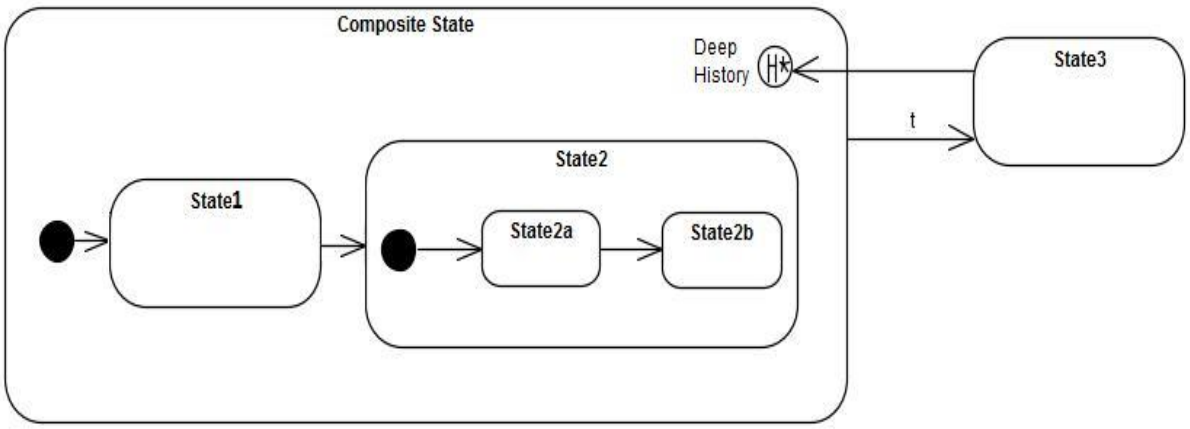
Figure History State.

Shallow History--



- Represents the most recent **active substate** of its containing Region, but not the substates of that substate.
- Transition terminating on this Pseudostate implies restoring the Region to that substate.
- A single outgoing Transition from this Pseudostate may be defined terminating on a substate of the composite State.
- Shallow History can only be defined for composite States and, at most one such Pseudostate can be included in a Region of a composite State.

Deep History--



- Represents the most recent **active state configuration** of its owning Region.
- Transition terminating on this Pseudostate implies restoring the Region to that same state configuration.
- The entry Behaviors of all States in the restored state configuration are performed in the appropriate order starting with the outermost State.
- Deep History Pseudostate can only be defined for composite States and, at most one such Pseudostate can be contained in a Region of a composite State.

Use Case Models

Actors--

An *actor* is a direct external user of a system—an object or set of objects that communicates directly with the system but that is not part of the system. Each actor represents those objects that behave in a particular way toward the system. For example, *customer* and *repair technician* are different actors of a vending machine. Actors can be persons, devices, and other systems—anything that interacts directly with the system.

Use Cases--

A *use case* is a coherent piece of functionality that a system can provide by interacting with actors. For example, a *customer* actor can *buy a beverage* from a vending machine. The customer inserts money into the machine, makes a selection, and ultimately receives a beverage. Similarly, a *repair technician* can *perform scheduled maintenance* on a vending machine. Figure 7.1 summarizes several use cases for a vending machine.

- **Buy a beverage.** The vending machine delivers a beverage after a customer selects and pays for it.
- **Perform scheduled maintenance.** A repair technician performs the periodic service on the vending machine necessary to keep it in good working condition.
- **Make repairs.** A repair technician performs the unexpected service on the vending machine necessary to repair a problem in its operation.
- **Load items.** A stock clerk adds items into the vending machine to replenish its stock of beverages.

Figure **Use case summaries for a vending machine.** A use case is a coherent piece of functionality that a system can provide by interacting with actors.

Use Case Diagrams--

A system involves a set of use cases and a set of actors. Each use case represents a slice of the functionality the system provides. The set of use cases shows the complete functionality of the system at some level of detail. Similarly, each actor represents one kind of object for which the system can perform behavior. The set of actors represents the complete set of objects that the system can serve. Objects accumulate behavior from all the systems with which they interact as actors.

The UML has a graphical notation for summarizing use cases and Figure shows an example. A rectangle contains the use cases for a system with the actors listed on the outside. The name of the system may be written near a side of the rectangle. A name within an ellipse denotes a use case. A "stick man" icon denotes an actor, with the name being placed below or adjacent to the icon. Solid lines connect use cases to participating actors.

In the figure, the actor *Repair technician* participates in two use cases, the others in one each. Multiple actors can participate in a use case, even though the example has only one actor per use case.

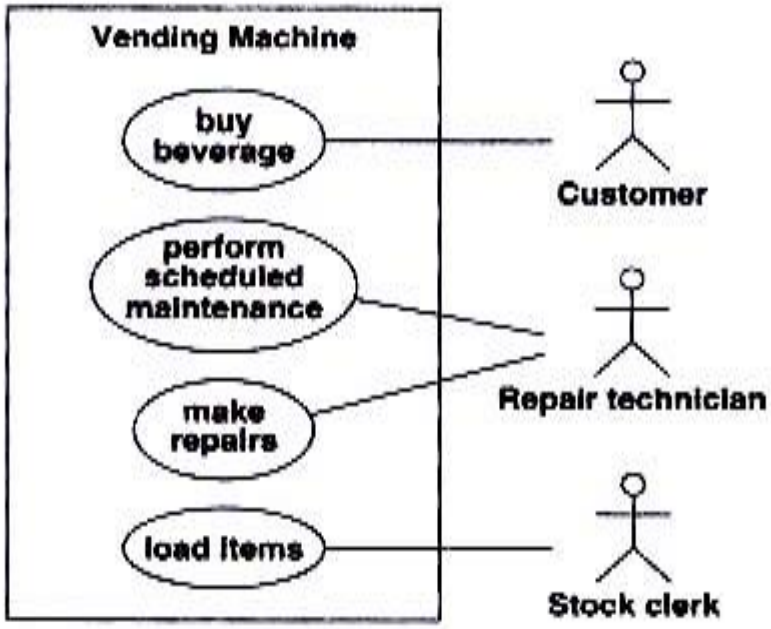


Figure **Use case diagram for a vending machine.** A system involves a set of use cases and a set of actors.

Guidelines for Use Case Models--

■ First determine the system boundary. Ensure that actors are focused. Each actor should have a single, coherent purpose.

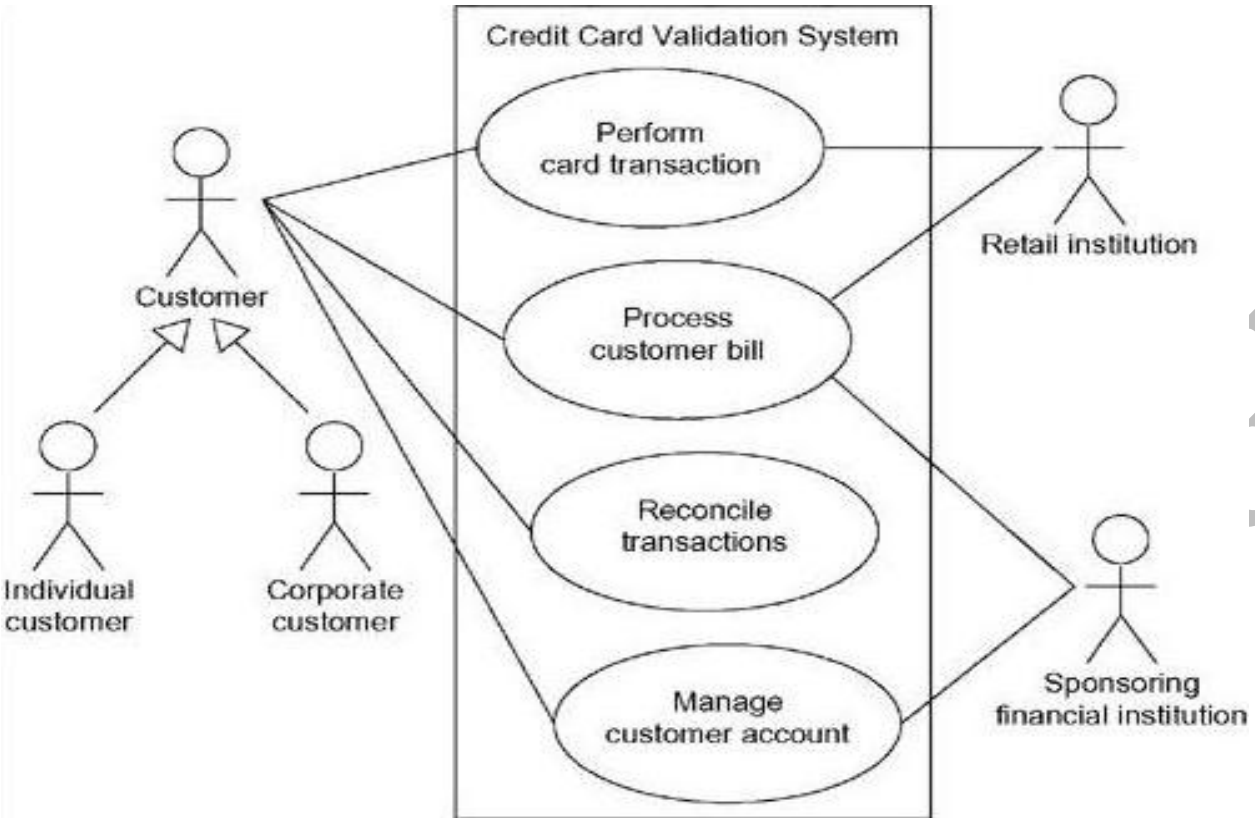
■ Each use case must provide value to users. A use case should represent a complete transaction that provides value to users and should not be defined too narrowly.

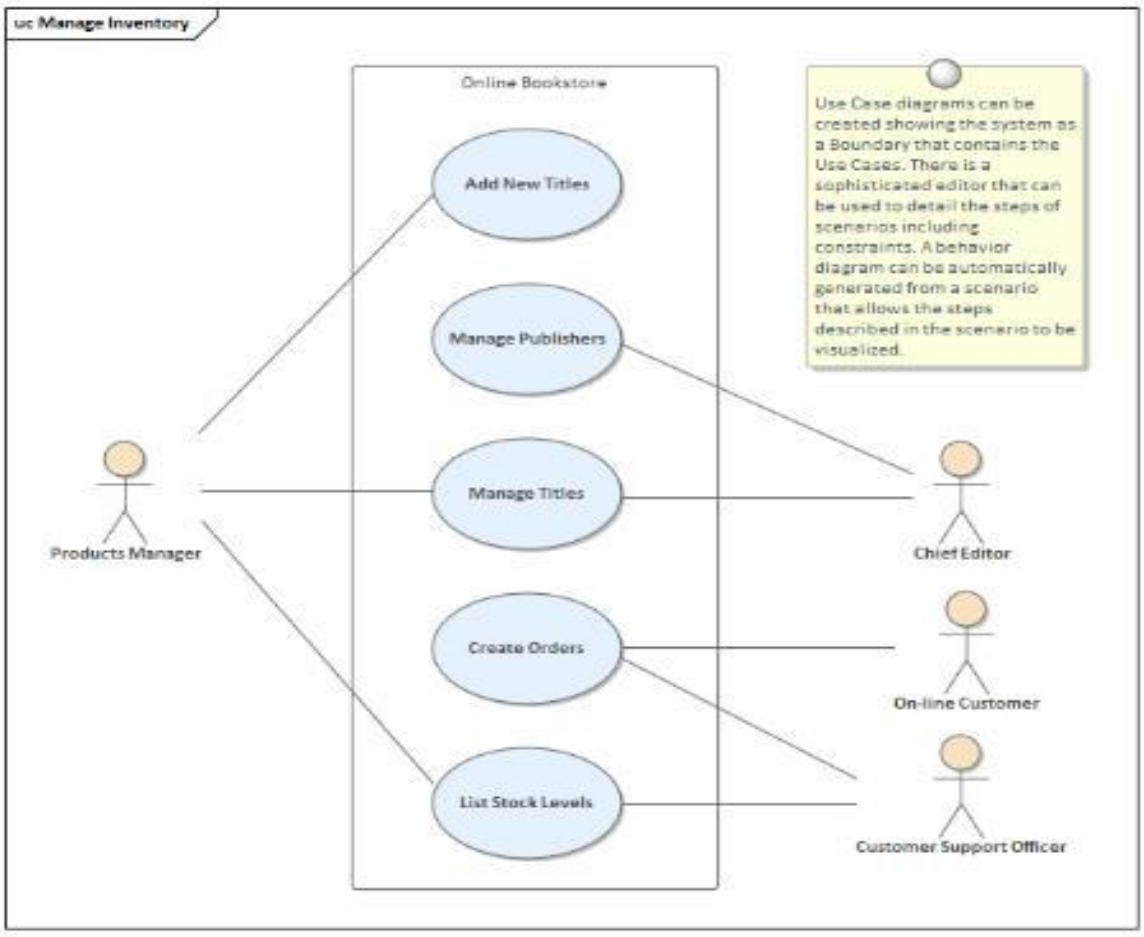
■ Relate use cases and actors. Every use case should have at least one actor, and every actor should participate in at least one use case. A use case may involve several actors, and an actor may participate in several use cases.

■ Remember that use cases are informal. It is important not to be obsessed by formalism in specifying use cases. They are not intended as a formal mechanism but as a way to identify and organize system functionality from a user-centered point of view. It is acceptable if use cases are a bit loose at first. Detail can come later as use cases are expanded and mapped into implementations.

■ Use cases can be structured. For many applications, the individual use cases are completely distinct. For large systems, use cases can be built out of smaller fragments using relationships

Examples---





Use Case Relationships--
Include Relationship--

The *include* relationship incorporates one use case within the behavior sequence of another use case. An included use case is like a subroutine.

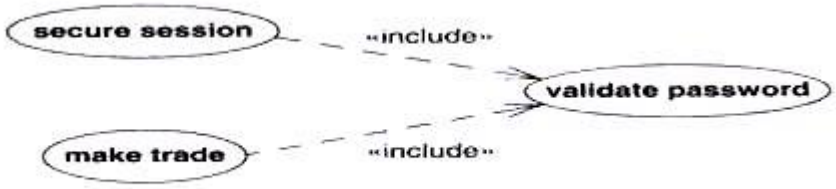


Figure **Use case inclusion.** The *include* relationship lets a base use case incorporate behavior from another use case.

Extend Relationship--

The *extend* relationship adds incremental behavior to a use case. It is like an include relationship looked at from the opposite direction, in which the extension adds itself to the base, rather than the base explicitly incorporating the extension.

Use case extension. The *extend* relationship is like an *include* relationship looked at from the opposite direction. The extension adds itself to the base.

Generalization--

Generalization can show specific variations on a general use case, analogous to generalization among classes. A parent use case represents a general behavior sequence. Child use cases specialize the parent by inserting additional steps or by refining steps.

Figure **Use case generalization.** A parent use case has common behavior and child use cases add variations, analogous to generalization among classes.

Combinations of Use Case Relationships--

A single diagram may combine several kinds of use case relationships. Figure shows a use case diagram from a stock brokerage system.

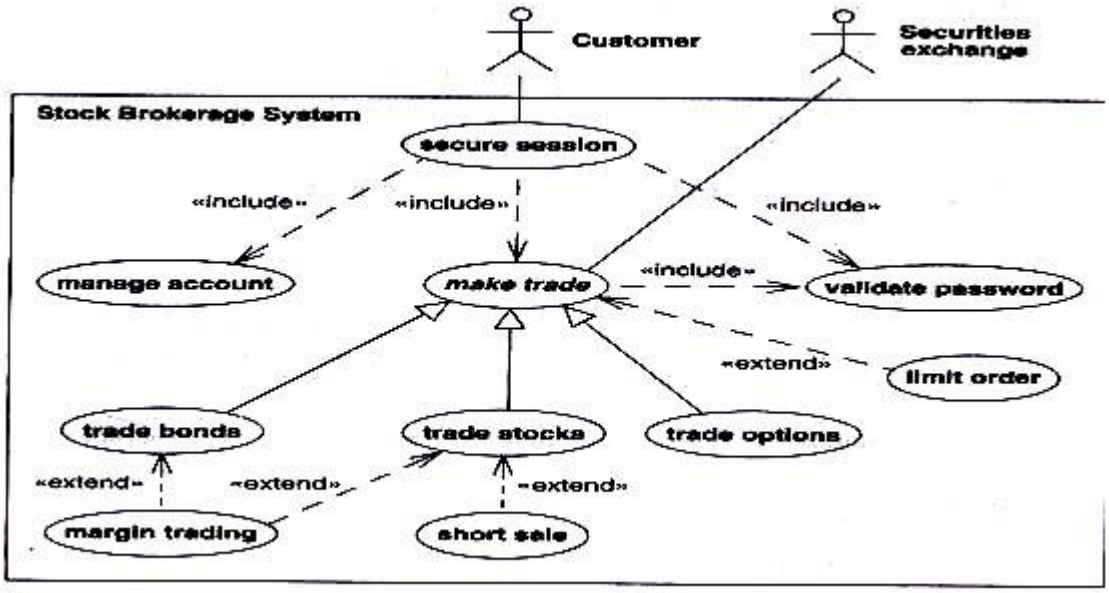
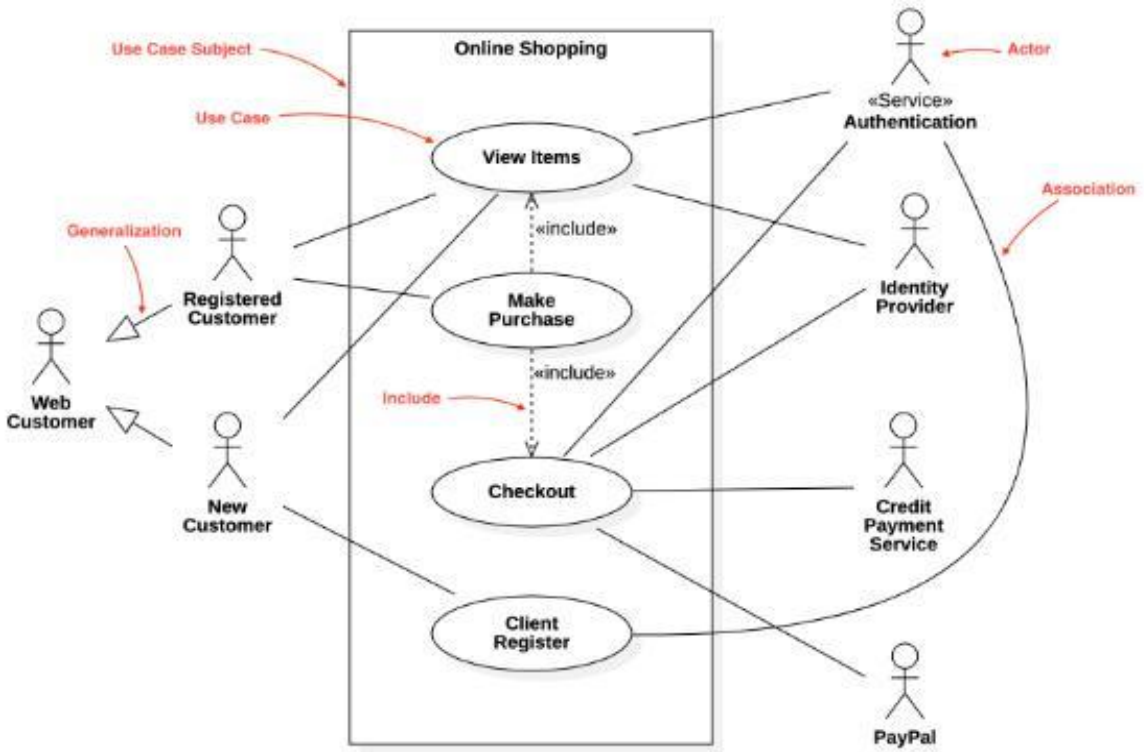
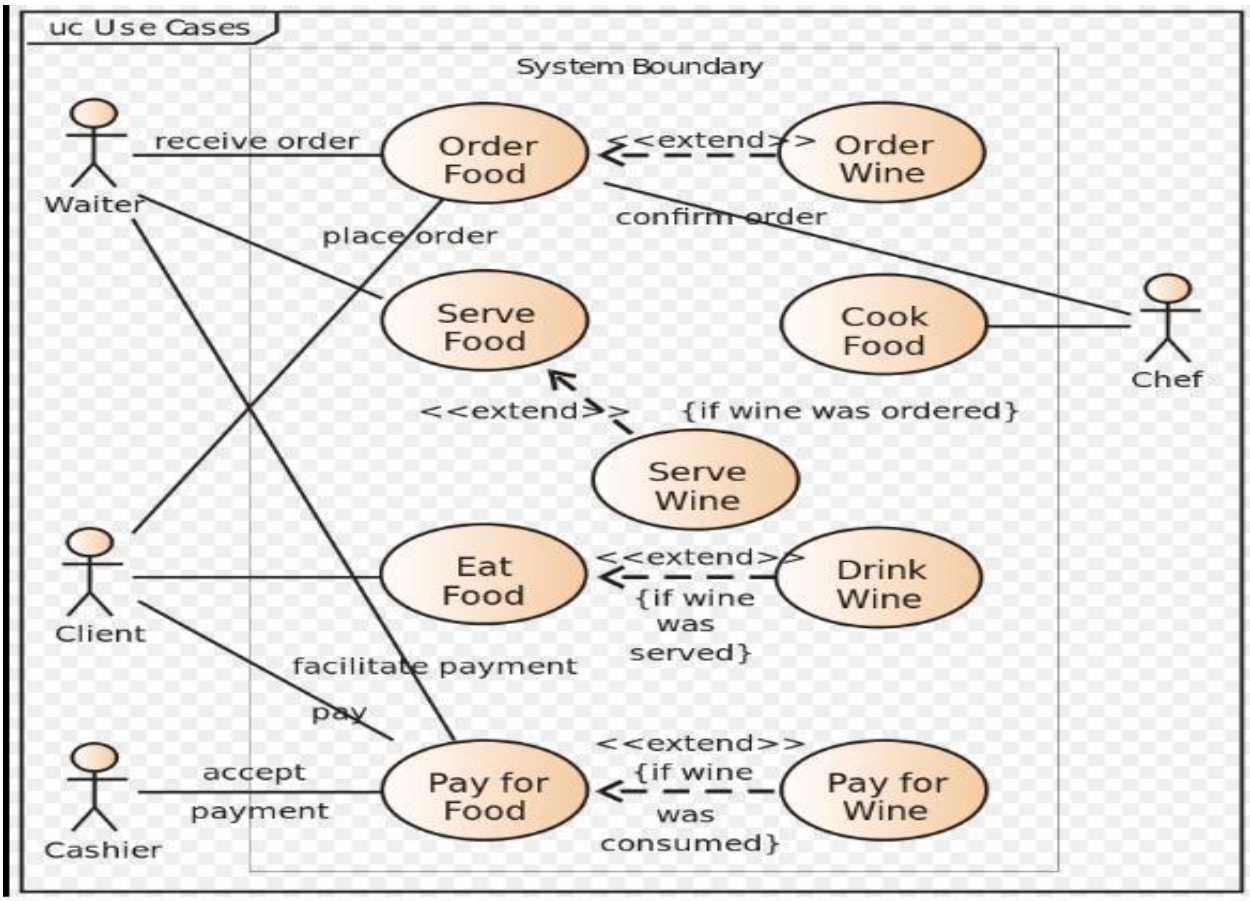


Figure **Use case relationships.** A single use case diagram may combine several kinds of relationships.

Examples---





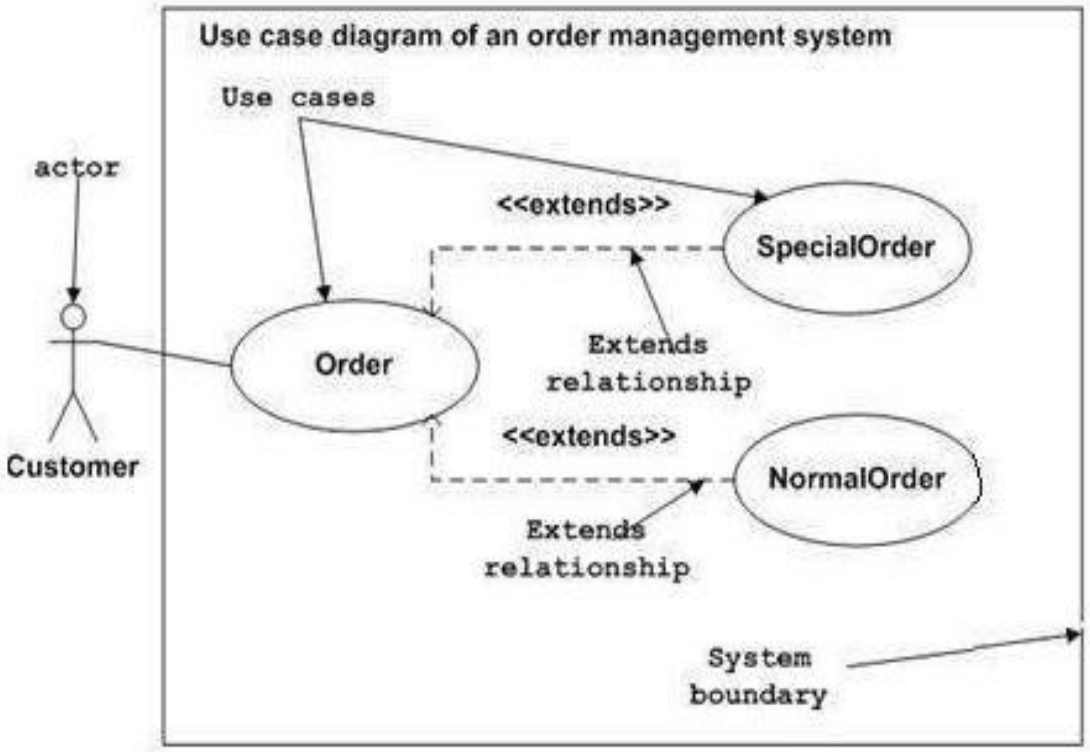
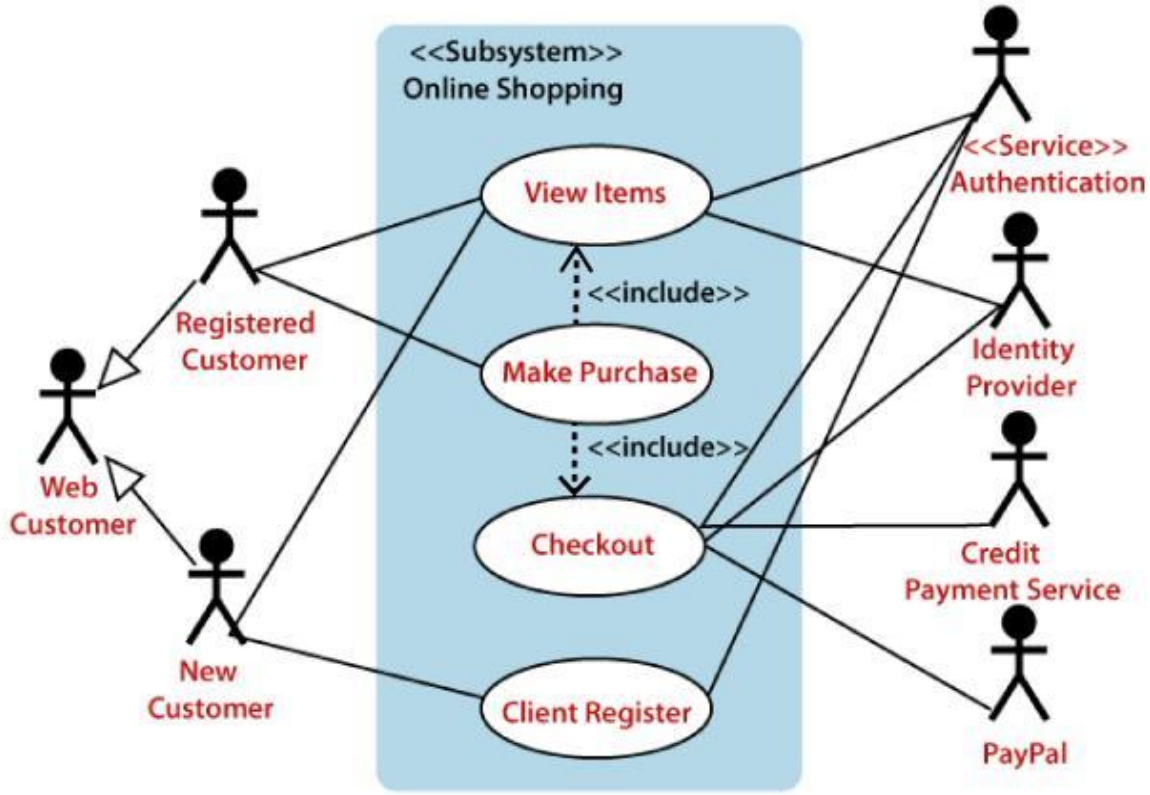
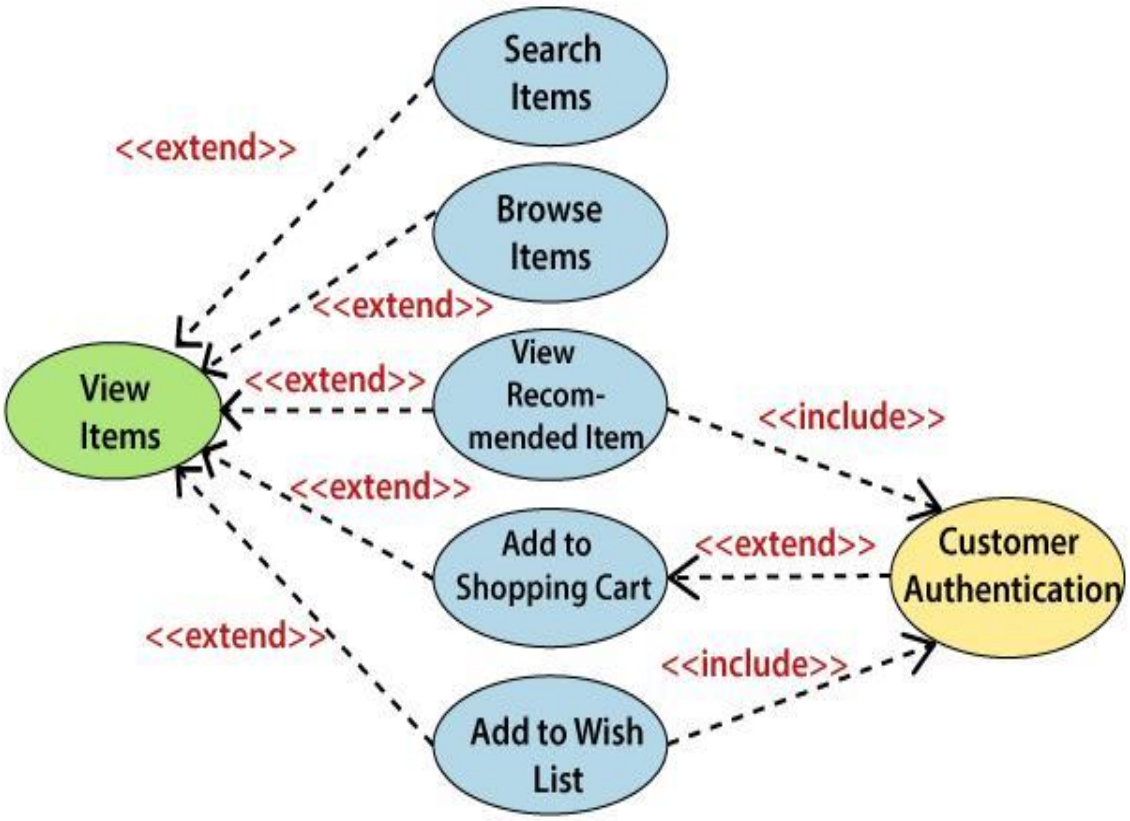
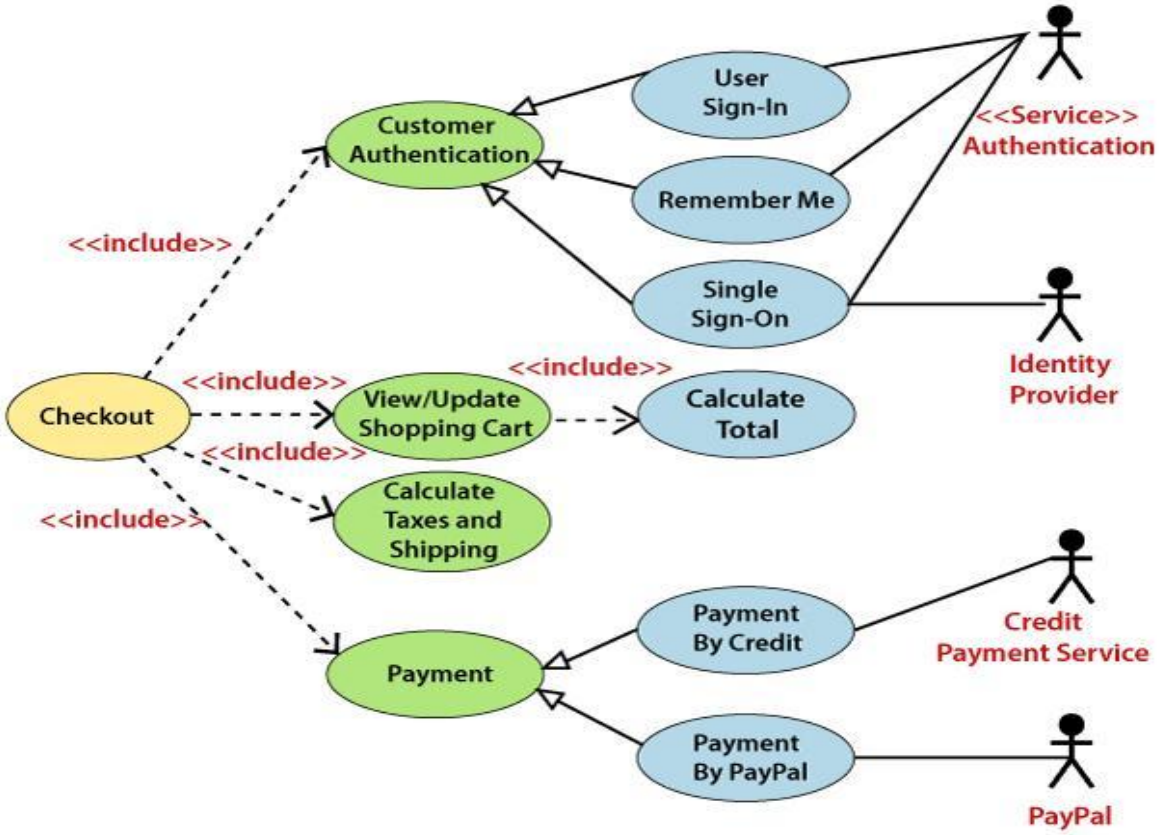


Figure: Sample Use Case diagram


<<Subsystem>>
Online Shopping
View Items
<<include>>
Make Purchase
<<include>>
Checkout
Client Register
Registered Customer
Web Customer
New Customer
<<Service>>
Authentication
Identity Provider
Credit Payment Service
PayPal


Search Items
<<extend>>
Browse Items
<<extend>>
View Items
<<extend>>
View Recom-mended Item
<<include>>
<<extend>>
Add to Shopping Cart
<<extend>>
Customer Authentication
<<extend>>
<<include>>
Add to Wish List



Interaction Diagrams

The purpose of interaction diagrams is to visualize the interactive behavior of the system. Visualizing the interaction is a difficult task. Hence, the solution is to use different types of models to capture the different aspects of the interaction.

Sequence and collaboration diagrams are used to capture the dynamic nature but from a different angle.

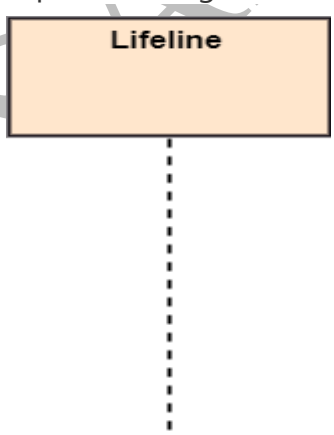
The purpose of interaction diagram is −

- To capture the dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe the structural organization of the objects.
- To describe the interaction among objects.

Notations of a Sequence Diagram

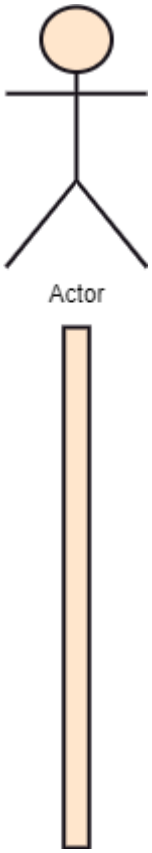
Lifeline

An individual participant in the sequence diagram is represented by a lifeline. It is positioned at the top of the diagram.



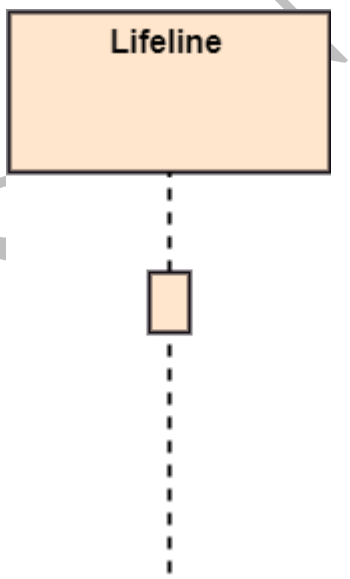
Actor

A role played by an entity that interacts with the subject is called as an actor. It is out of the scope of the system. It represents the role, which involves human users and external hardware or subjects. An actor may or may not represent a physical entity, but it purely depicts the role of an entity. Several distinct roles can be played by an actor or vice versa.



Activation

It is represented by a thin rectangle on the lifeline. It describes that time period in which an operation is performed by an element, such that the top and the bottom of the rectangle is associated with the initiation and the completion time, each respectively.

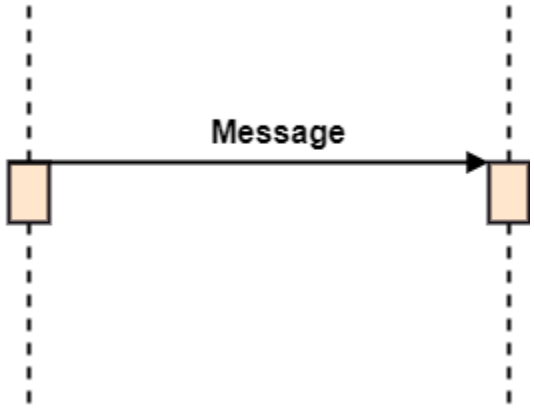


Messages--

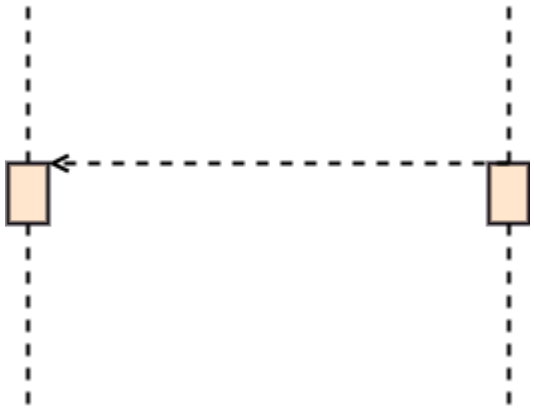
The messages depict the interaction between the objects and are represented by arrows. They are in the sequential order on the lifeline. The core of the sequence diagram is formed by messages and lifelines.

Following are types of messages enlisted below:

- **Call Message:** It defines a particular communication between the lifelines of an interaction, which represents that the target lifeline has invoked an operation.

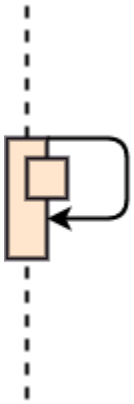


- **Return Message:** It defines a particular communication between the lifelines of interaction that represent the flow of information from the receiver of the corresponding caller message.

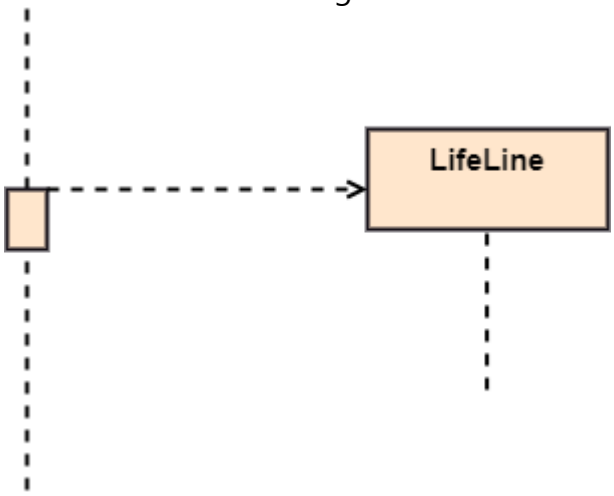


- **Self Message:** It describes a communication, particularly between the lifelines of an interaction that represents a message of the same lifeline, has been invoked.

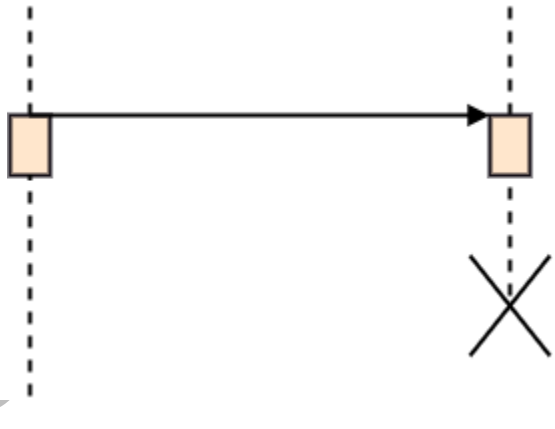
- **Recursive Message:** A self message sent for recursive purpose is called a recursive message. In other words, it can be said that the recursive message is a special case of the self message as it represents the recursive calls.



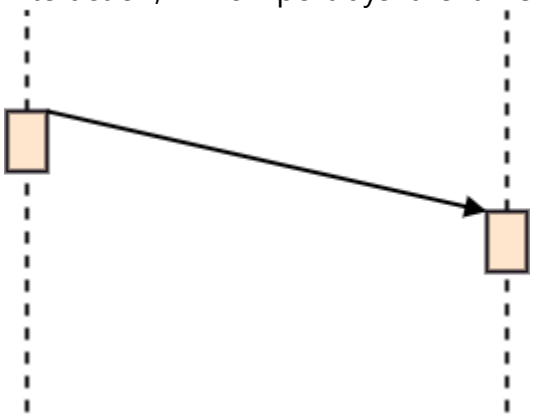
- **Create Message:** It describes a communication, particularly between the lifelines of an interaction describing that the target (lifeline) has been instantiated.



- **Destroy Message:** It describes a communication, particularly between the lifelines of an interaction that depicts a request to destroy the lifecycle of the target.



- **Duration Message:** It describes a communication particularly between the lifelines of an interaction, which portrays the time passage of the message while modeling a system.

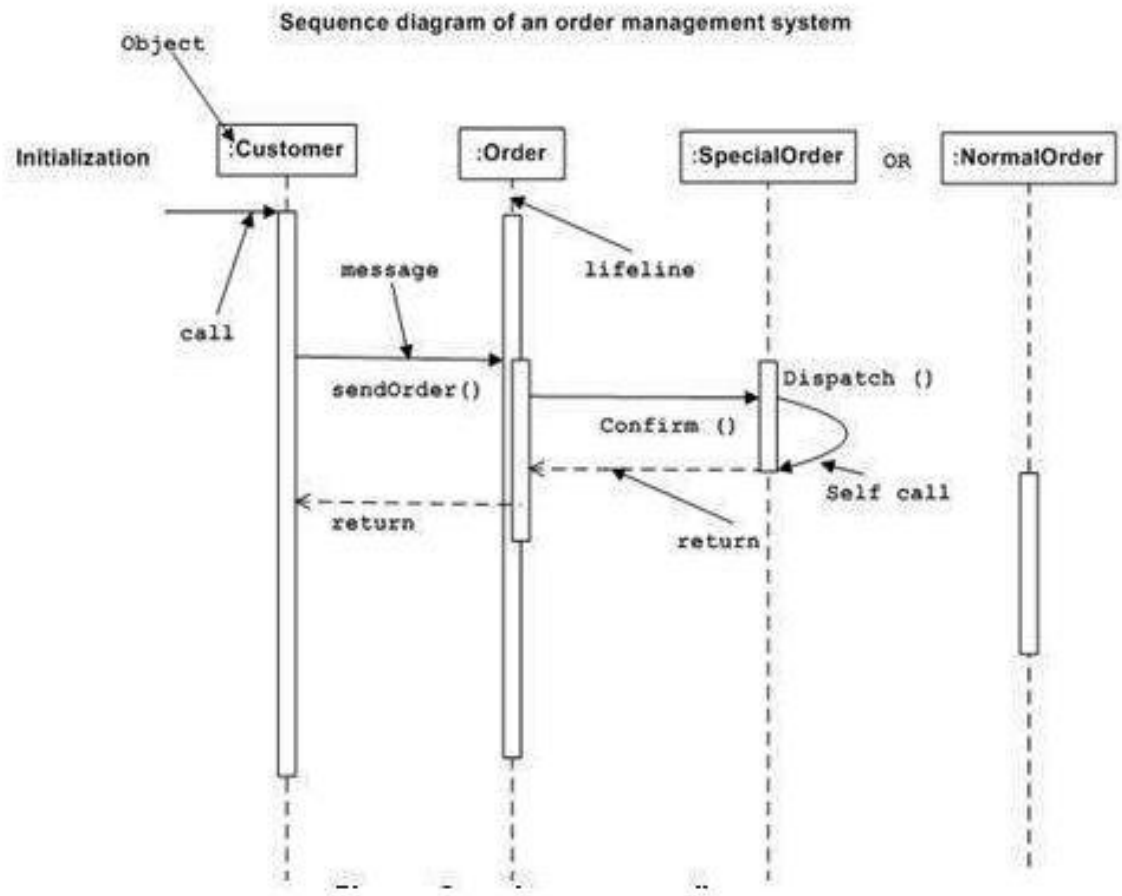


The Sequence Diagram---

The sequence diagram has four objects (Customer, Order, SpecialOrder and NormalOrder).

The following diagram shows the message sequence for *SpecialOrder* object and the same can be used in case of *NormalOrder* object. It is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.

The first call is *sendOrder ()* which is a method of *Order object*. The next call is *confirm ()* which is a method of *SpecialOrder* object and the last call is *Dispatch ()* which is a method of *SpecialOrder* object. The following diagram mainly describes the method calls from one object to another, and this is also the actual scenario when the system is running.



Benefits of a Sequence Diagram

1. It explores the real-time application.
2. It depicts the message flow between the different objects.
3. It has easy maintenance.
4. It is easy to generate.
5. Implement both forward and reverse engineering.
6. It can easily update as per the new change in the system.

The drawback of a Sequence Diagram

1. In the case of too many lifelines, the sequence diagram can get more complex.
2. The incorrect result may be produced, if the order of the flow of messages changes.
3. Since each sequence needs distinct notations for its representation, it may make the diagram more complex.
4. The type of sequence is decided by the type of message.

Example---

Figure shows a sequence diagram corresponding to the stock broker scenario. Each actor as well as the system is represented by a vertical line called a *lifeline* and each message by a horizontal arrow from the sender to the receiver. Time proceeds from top to bottom, but the spacing is irrelevant; the diagram shows only the sequence of messages, not their exact timing.

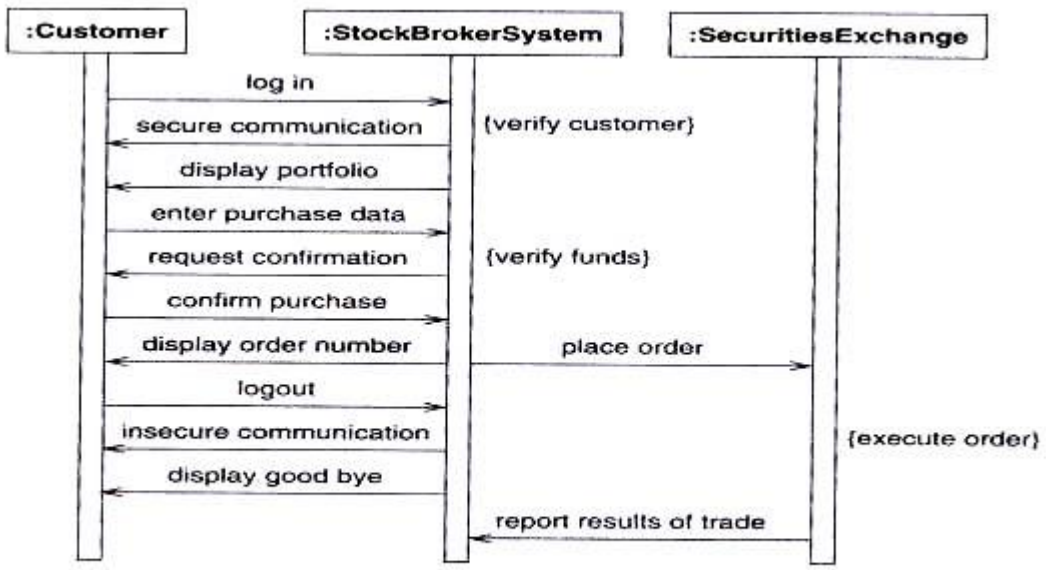


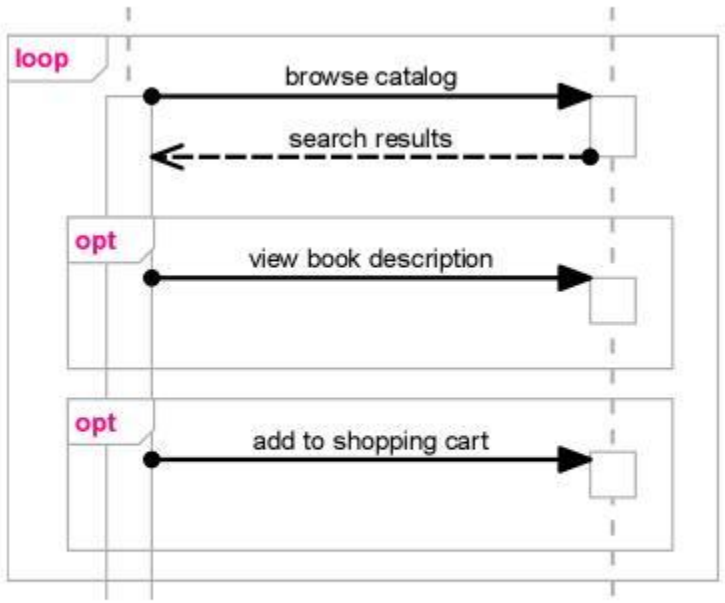
Figure **Sequence diagram for a session with an online stock broker.** A sequence diagram shows the participants in an interaction and the sequence of messages among them.

Fragments--

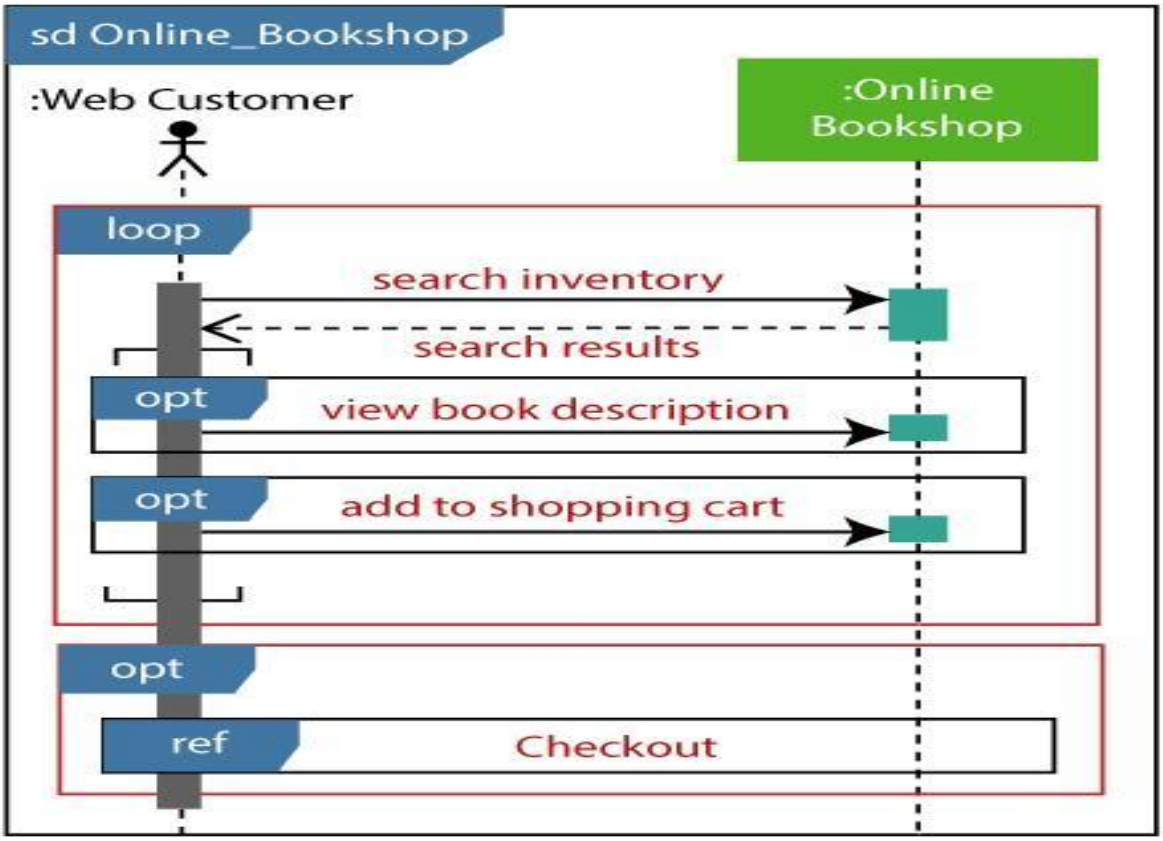
A fragment in a UML sequence diagram can be used as an overlay over a number of messages to specify whether these are a sequence, alternatives, optional, parallel, or a loop. There are a few more kinds of fragments specified in the UML standard that can also be used.

An operator is used to indicate the fragment kind, and it is shown in the upper left corner of the fragment. The default fragment kind is sequence (**seq**).

Example of fragments with different operators



Example of fragments with different operators



UML Collaboration / Communication Diagram

The collaboration diagram is used to show the relationship between the objects in a system. Both the sequence and the collaboration diagrams represent the same information but differently. Instead of showing the flow of messages, it depicts the architecture of the object residing in the system as it is based on object-oriented programming. An object consists of several features. Multiple objects present in the system are connected to each other. The collaboration diagram, which is also known as a communication diagram, is used to portray the object's architecture in the system.

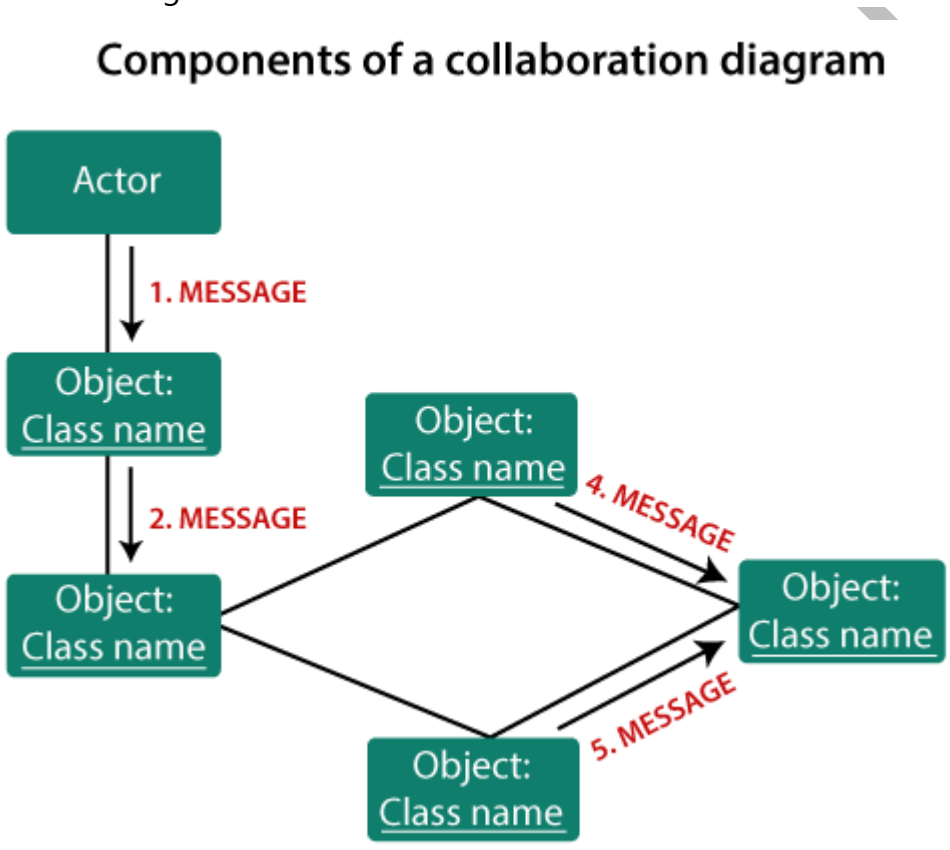
Notations of a Collaboration/Communication Diagram

Following are the components of a component diagram that are enlisted below:

1. **Objects:** The representation of an object is done by an object symbol with its name and class underlined, separated by a colon.
   In the collaboration diagram, objects are utilized in the following ways:
   - The object is represented by specifying their name and class.
   - It is not mandatory for every class to appear.
   - A class may constitute more than one object.
   - In the collaboration diagram, firstly, the object is created, and then its class is specified.

- To differentiate one object from another object, it is necessary to name them.

2. **Actors:** In the collaboration diagram, the actor plays the main role as it invokes the interaction. Each actor has its respective role and name. In this, one actor initiates the use case.
3. **Links:** The link is an instance of association, which associates the objects and actors. It portrays a relationship between the objects through which the messages are sent. It is represented by a solid line. The link helps an object to connect with or navigate to another object, such that the message flows are attached to links.
4. **Messages:** It is a communication between objects which carries information and includes a sequence number, so that the activity may take place. It is represented by a labeled arrow, which is placed near a link. The messages are sent from the sender to the receiver, and the direction must be navigable in that particular direction. The receiver must understand the message.

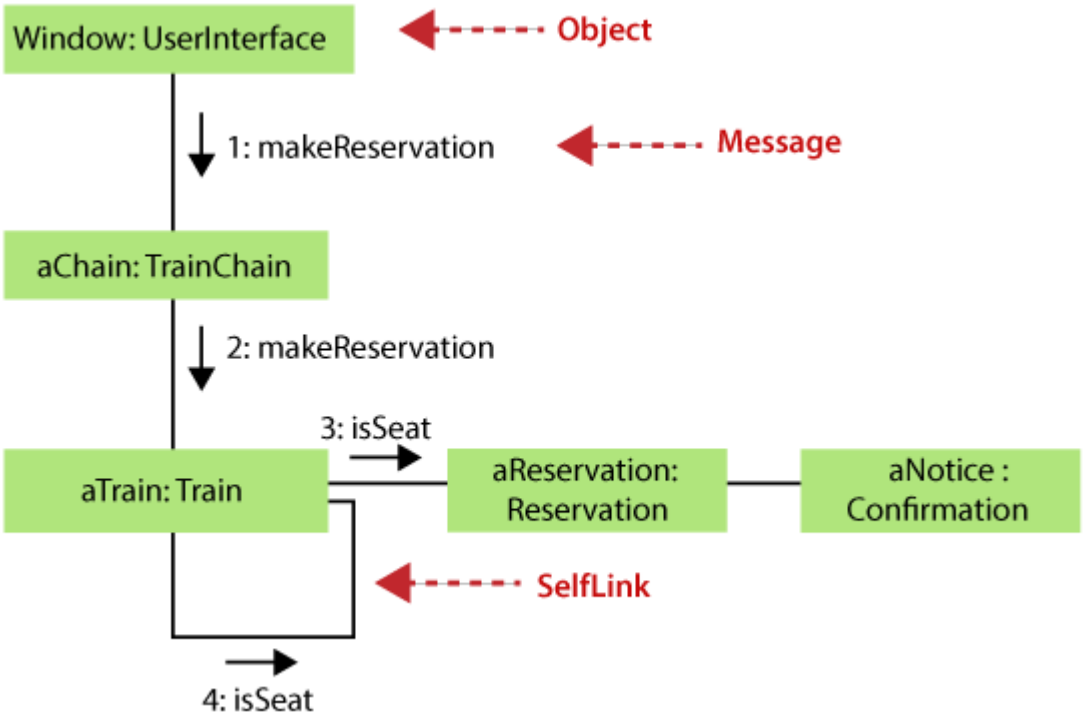


Steps for creating a Collaboration Diagram

1. Determine the behavior for which the realization and implementation are specified.
2. Discover the structural elements that are class roles, objects, and subsystems for performing the functionality of collaboration.
    - Choose the context of an interaction: system, subsystem, use case, and operation.

3. Think through alternative situations that may be involved.
    - Implementation of a collaboration diagram at an instance level, if needed.
    - A specification level diagram may be made in the instance level sequence diagram for summarizing alternative situations.

Example of a Collaboration Diagram



Benefits of a Collaboration Diagram

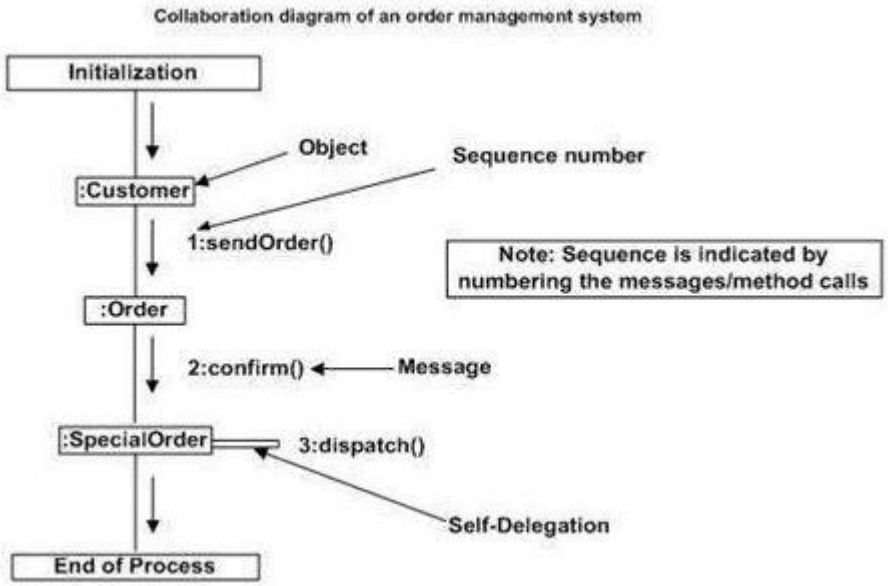
1. The collaboration diagram is also known as Communication Diagram.
2. It mainly puts emphasis on the structural aspect of an interaction diagram, i.e., how lifelines are connected.
3. The syntax of a collaboration diagram is similar to the sequence diagram; just the difference is that the lifeline does not consist of tails.
4. The messages transmitted over sequencing is represented by numbering each individual message.
5. The collaboration diagram is semantically weak in comparison to the sequence diagram.
6. The special case of a collaboration diagram is the object diagram.
7. It focuses on the elements and not the message flow, like sequence diagrams.
8. Since the collaboration diagrams are not that expensive, the sequence diagram can be directly converted to the collaboration diagram.

9. There may be a chance of losing some amount of information while implementing a collaboration diagram with respect to the sequence diagram.

The drawback of a Collaboration Diagram

1. Multiple objects residing in the system can make a complex collaboration diagram, as it becomes quite hard to explore the objects.
2. It is a time-consuming diagram.
3. After the program terminates, the object is destroyed.
4. As the object state changes momentarily, it becomes difficult to keep an eye on every single that has occurred inside the object of a system.

Example---



Activity Diagram--

Activity diagram is another important diagram in UML to describe the dynamic aspects of the system.

Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system.

The control flow is drawn from one operation to another. This flow can be sequential, branched, or concurrent. Activity diagrams deal with all type of flow control by using different elements such as fork, join, etc

The purpose of an activity diagram can be described as −

- Draw the activity flow of a system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.

Notation of an Activity diagram

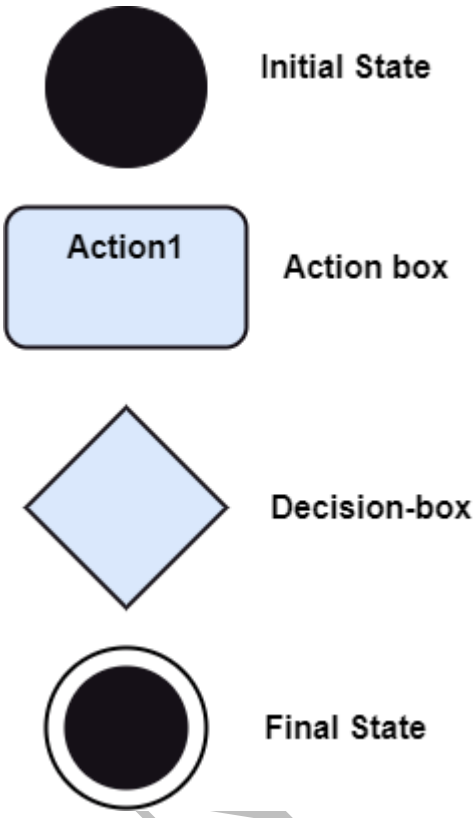
Activity diagram constitutes following notations:

Initial State: It depicts the initial stage or beginning of the set of actions.

Final State: It is the stage where all the control flows and object flows end.

Decision Box: It makes sure that the control flow or object flow will follow only one path.

Action Box: It represents the set of actions that are to be performed.



Following is an example of an activity diagram for order management system. In the diagram, four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and is mainly used by the business users

Following diagram is drawn with the four main activities −

- Send order by the customer
- Receipt of the order
- Confirm the order
- Dispatch the order

After receiving the order request, condition checks are performed to check if it is normal or special order. After the type of order is identified, dispatch activity is performed and that is marked as the termination of the process.

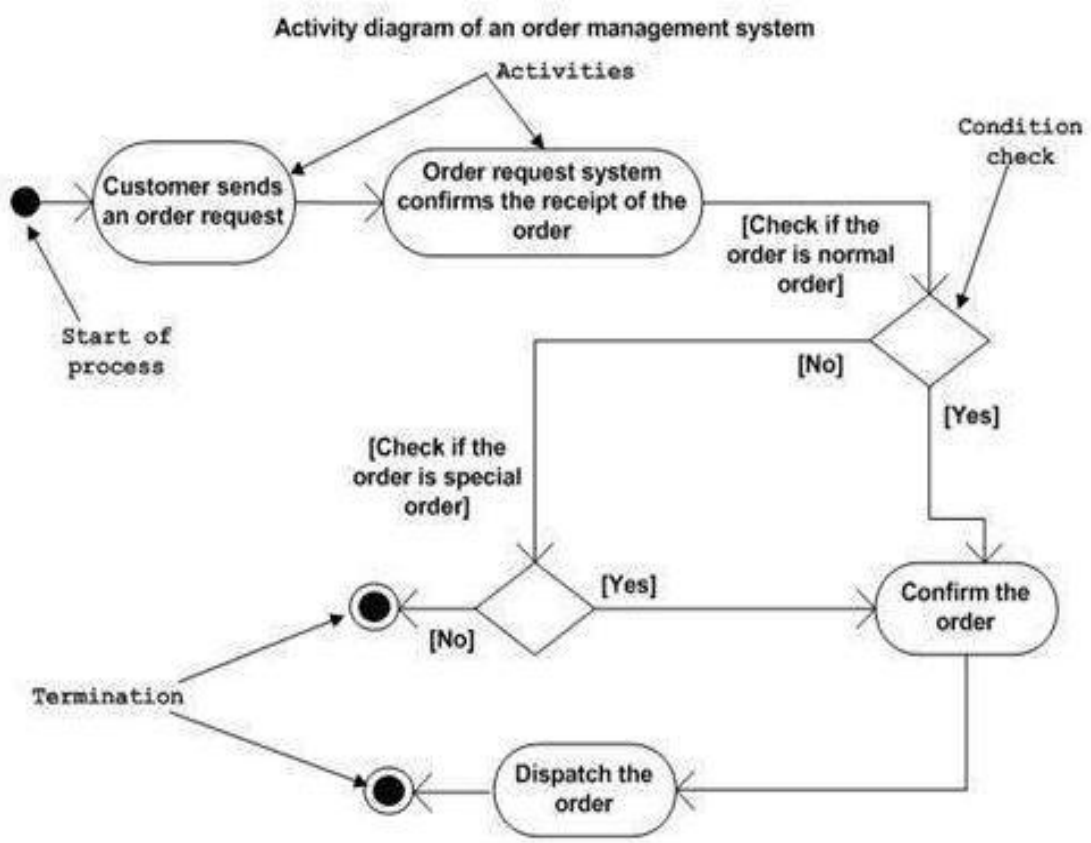


Figure below shows an activity diagram for the processing of a stock trade order that has been received by an online stock broker. The elongated ovals show activities and the arrows show their sequencing. The diamond shows a decision point and the heavy bar shows splitting or merging of concurrent threads.

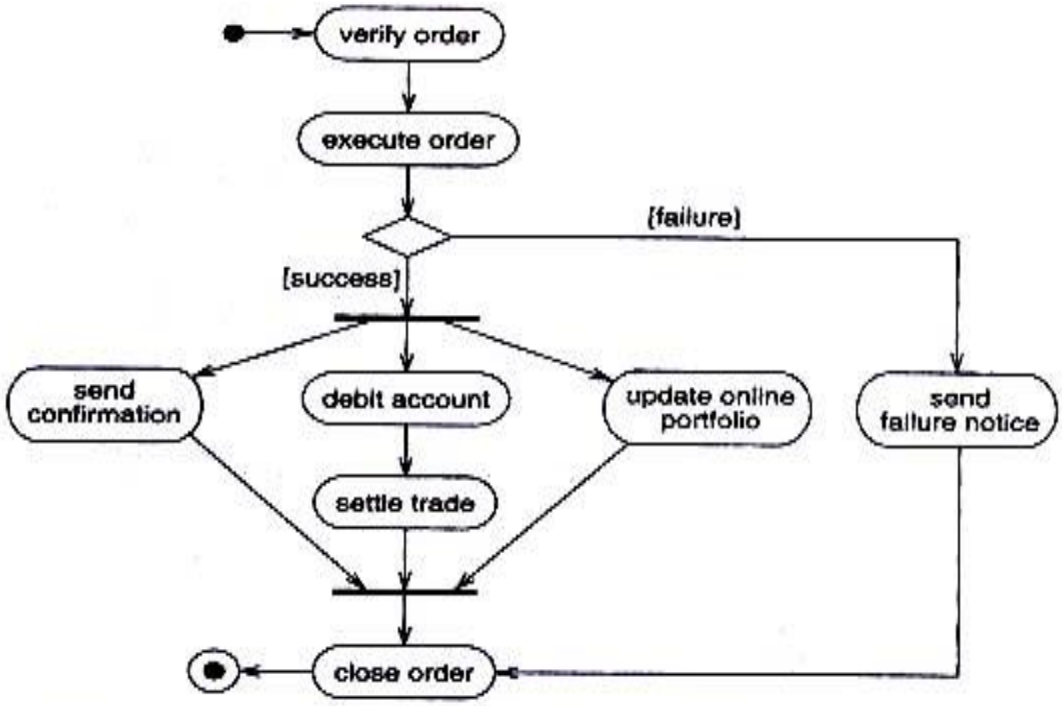
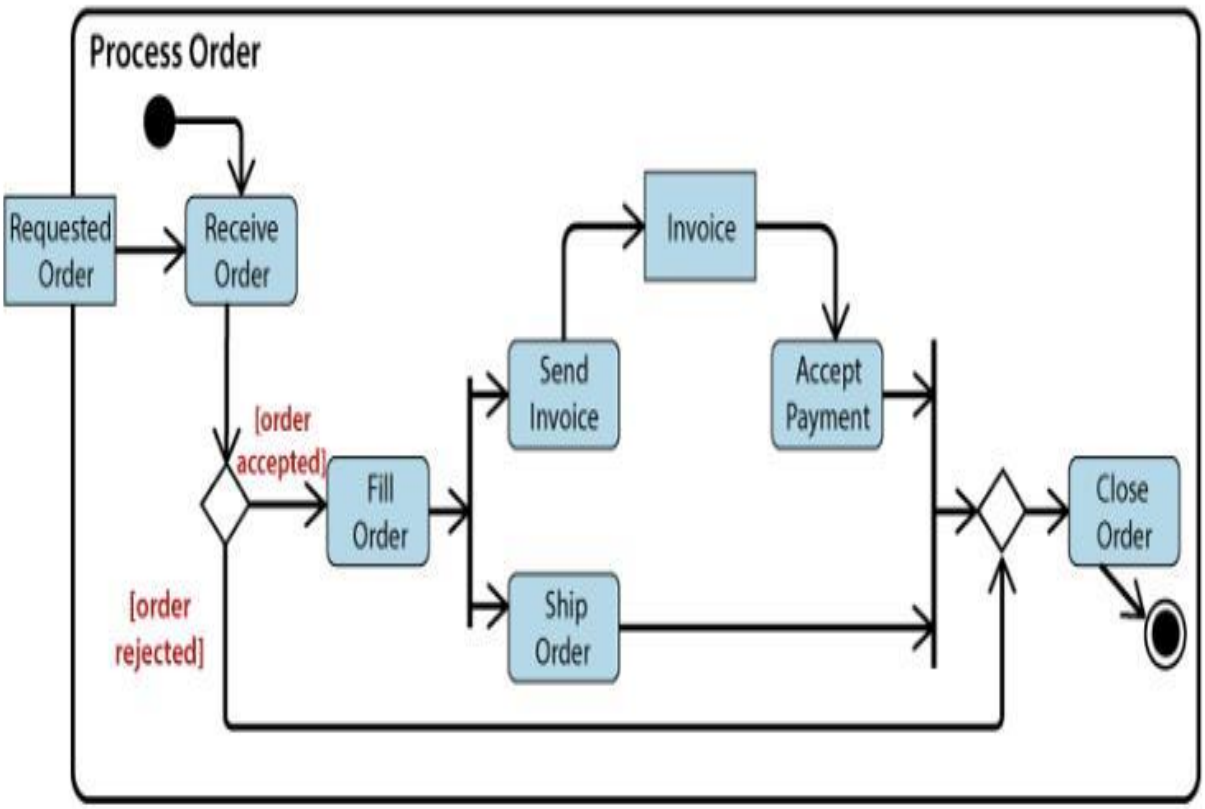


Figure **Activity diagram for stock trade processing.** An activity diagram shows the sequence of steps that make up a complex process.



Guidelines for Activity Models--

■ Don't misuse activity diagrams. Activity diagrams are intended to elaborate use case and sequence models so that a developer can study algorithms and workflow.

■Level diagrams. Activities on a diagram should be at a consistent level of detail. Place additional detail for an activity in a separate diagram.
■Be careful with branches and conditions. If there are conditions, at least one must be satisfied when an activity completes—consider using an *else* condition.
■Be careful with concurrent activities. Concurrency means that the activities can complete in any order and still yield an acceptable result. Before a merge can happen, all inputs must first complete.
■Consider executable activity diagrams. Executable activity diagrams can help developers understand their systems better.

Swimlanes--

We can show such an activity diagram by dividing it into columns and lines. Each column is called *a swimlane* by analogy to a swimming pool. Placing an activity within a particular swimlane indicates that it is performed by a person or persons within the organization. Lines across swimlane boundaries indicate interactions among different organizations, which must usually be treated with more care than interactions within an organization.
Figure shows a simple example for servicing an airplane. The flight attendants must clean the trash, the ground crew must add fuel, and catering must load food and drink before a plane is serviced and ready for its next flight.

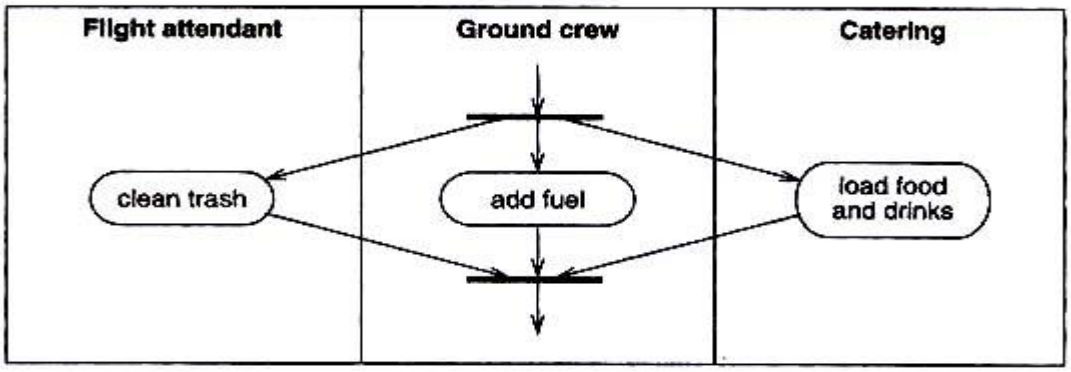


Figure **Activity diagram with swimlanes.** Swimlanes can show organizational responsibility for activities.

Object Flows--

Frequently the same object goes through several states during the execution of an activity diagram. The same object may be an input to or an output from several activities/states. The UML shows an object value in a particular state by placing the state name in square brackets following the object name. If the objects have state names, the activity diagram shows both the flow of control and the progression of an object from state to state as activities act on it. In Figure an airplane goes through several states as it leaves the gate, flies, and then lands again.

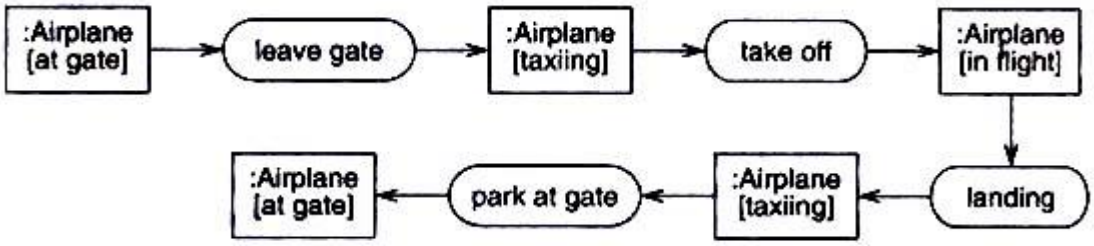


Figure **Activity diagram with object flows.** An activity diagram can show the objects that are inputs or outputs of activities.

Component Diagram--

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.

Thus from that point of view, 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 is used to represent the whole.

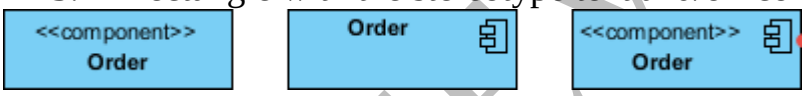
The purpose of the component diagram can be summarized as −

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

Basic Concepts of Component Diagram

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. In UML 2, a component is drawn as a rectangle with optional compartments stacked vertically. A high-level, abstracted view of a component in UML 2 can be modeled as:

1. A rectangle with the component's name
2. A rectangle with the component icon
3. A rectangle with the stereotype text and/or icon

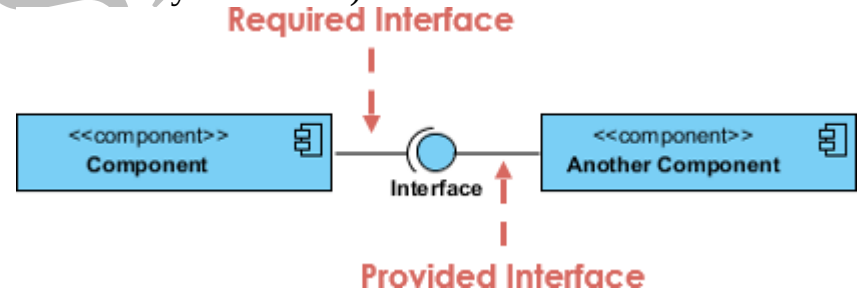


Interface

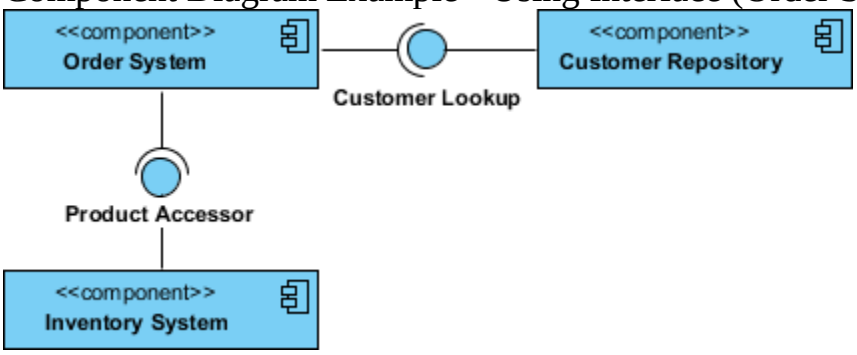
In the example below shows two type of component interfaces:

Provided interface symbols with a complete circle at their end represent an interface that the component provides - this "lollipop" symbol is shorthand for a realization relationship of an interface classifier.

Required Interface symbols with only a half circle at their end (a.k.a. sockets) represent an interface that the component requires (in both cases, the interface's name is placed near the interface symbol itself).

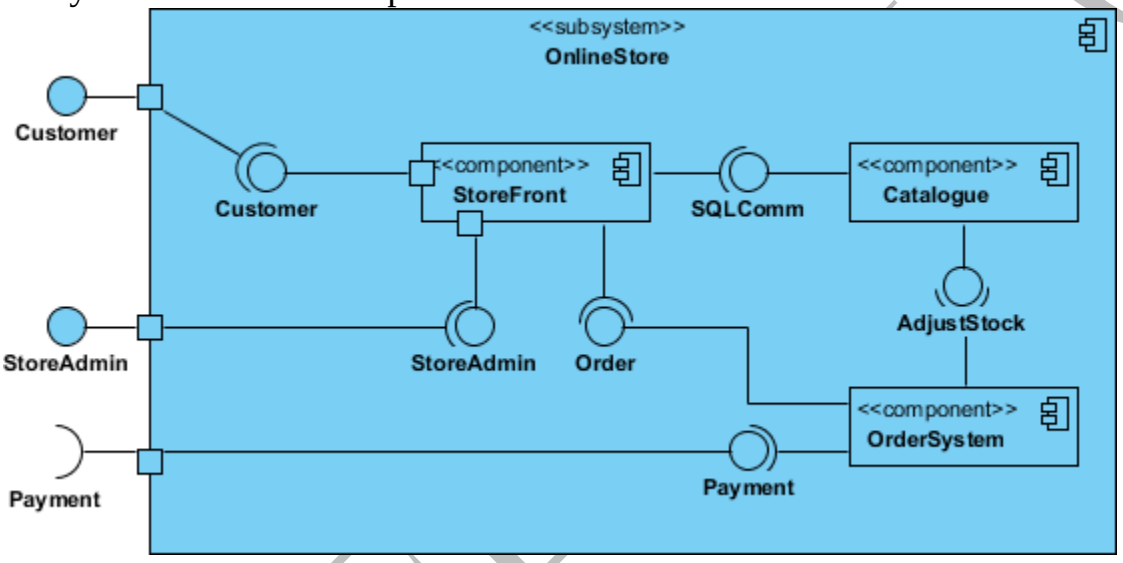


Component Diagram Example - Using Interface (Order System)



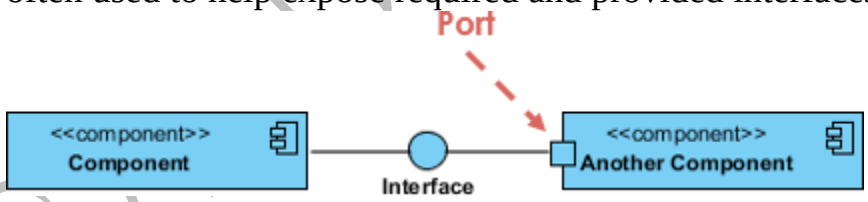
Subsystems

The subsystem classifier is a specialized version of a component classifier. Because of this, the subsystem notation element inherits all the same rules as the component notation element. The only difference is that a subsystem notation element has the keyword of subsystem instead of component.



Port

Ports are represented using a square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component.



Relationships

Graphically, a component diagram is a collection of vertices and arcs and commonly contain components, interfaces and dependency, aggregation, constraint, generalization, association, and realization relationships. It may also contain notes and constraints.

Relationships **Notation**

Association:

- An association specifies a semantic relationship that can occur between typed instances.
- It has at least two ends represented by properties, each of which is connected to the type of the end. More than one end of the association may have the same type.

Composition:

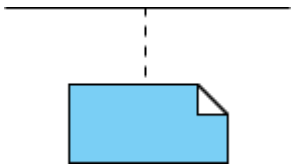
- Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time.
- If a composite is deleted, all of its parts are normally deleted with it.

Aggregation

- A kind of association that has one of its end marked shared as kind of aggregation, meaning that it has a shared aggregation.

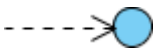
Constraint

- A condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element.



Dependency

- A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for



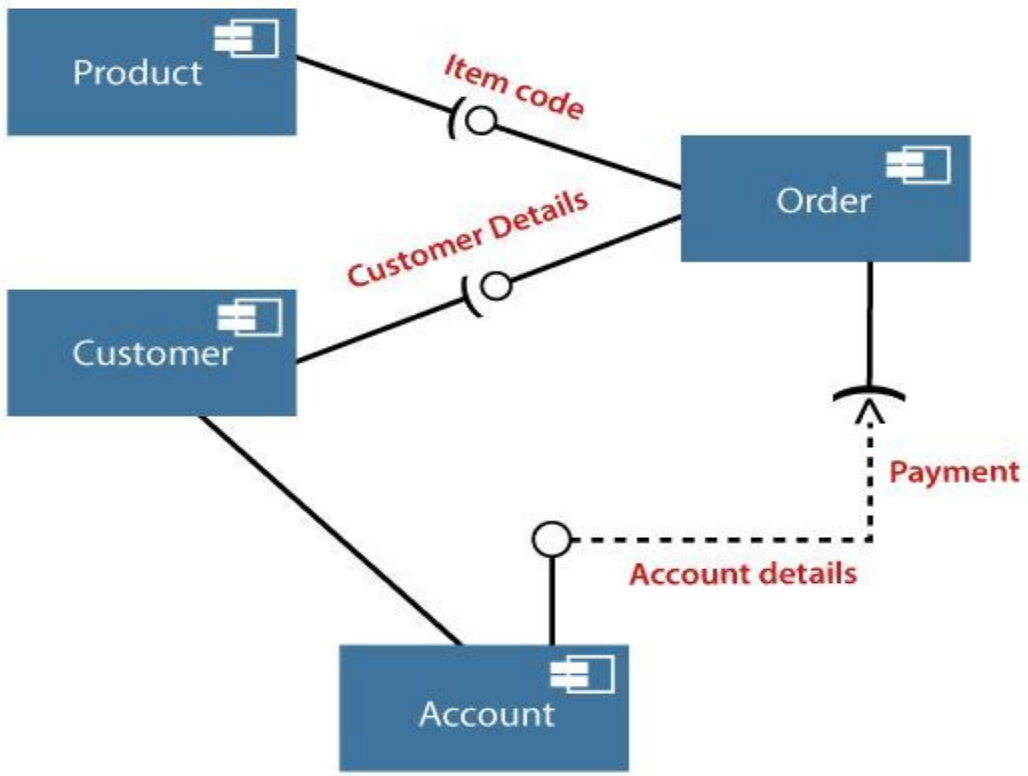
their specification or implementation.

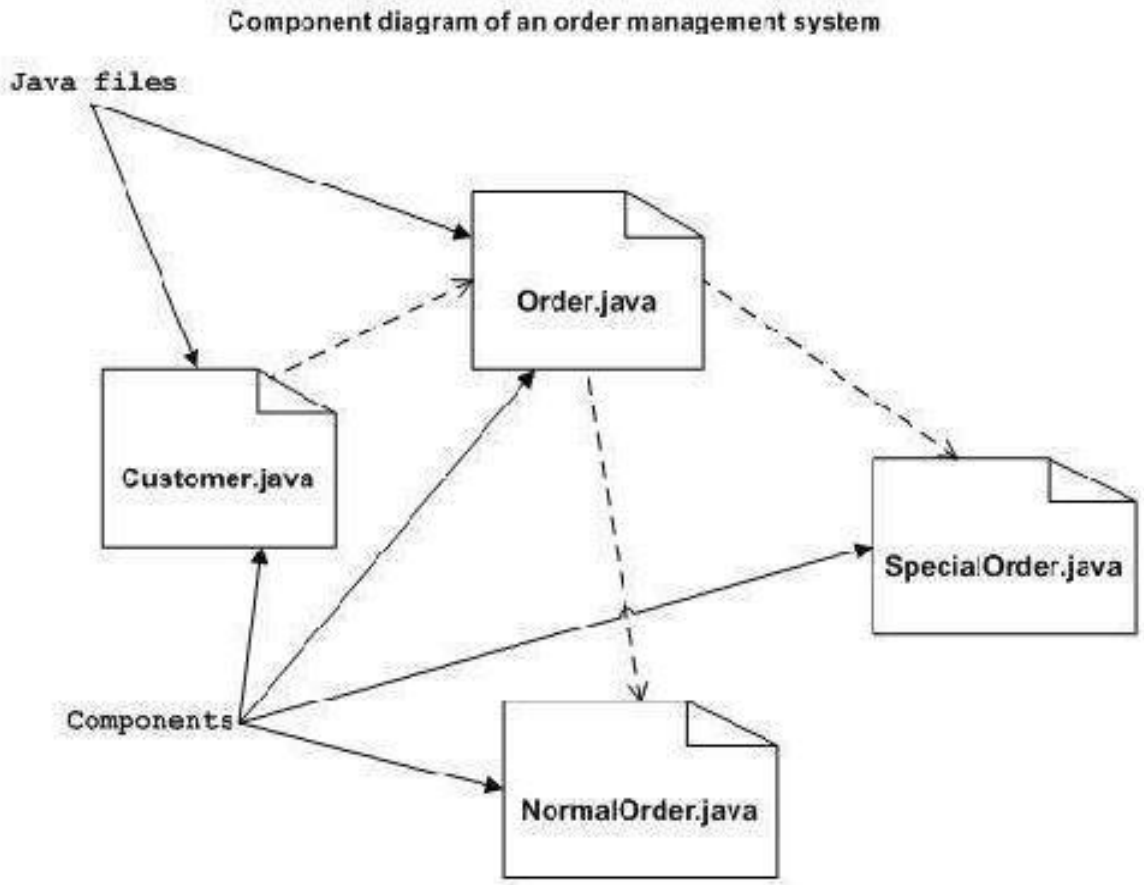
- This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s).

Links:

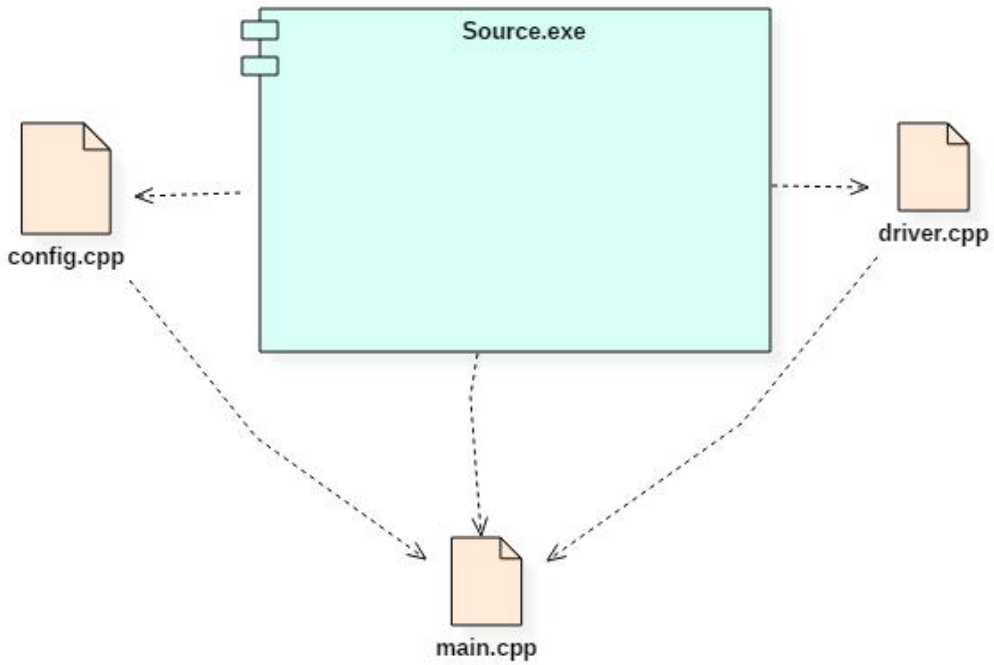
- A generalization is a taxonomic relationship between a more general classifier and a more specific classifier.
- Each instance of the specific classifier is also an indirect instance of the general classifier.
- Thus, the specific classifier inherits the features of the more general classifier.

A component diagram for an online shopping system is given below:





Example of a component diagram



component diagram Example

UML Deployment Diagram

The deployment diagram visualizes the physical hardware on which the software will be deployed. It portrays the static deployment view of a system. It involves the nodes and their relationships.
It ascertains how software is deployed on the hardware. It maps the software architecture created in design to the physical system architecture, where the software will be executed as a node. Since it involves many nodes, the relationship is shown by utilizing communication paths.

Purpose of Deployment Diagram

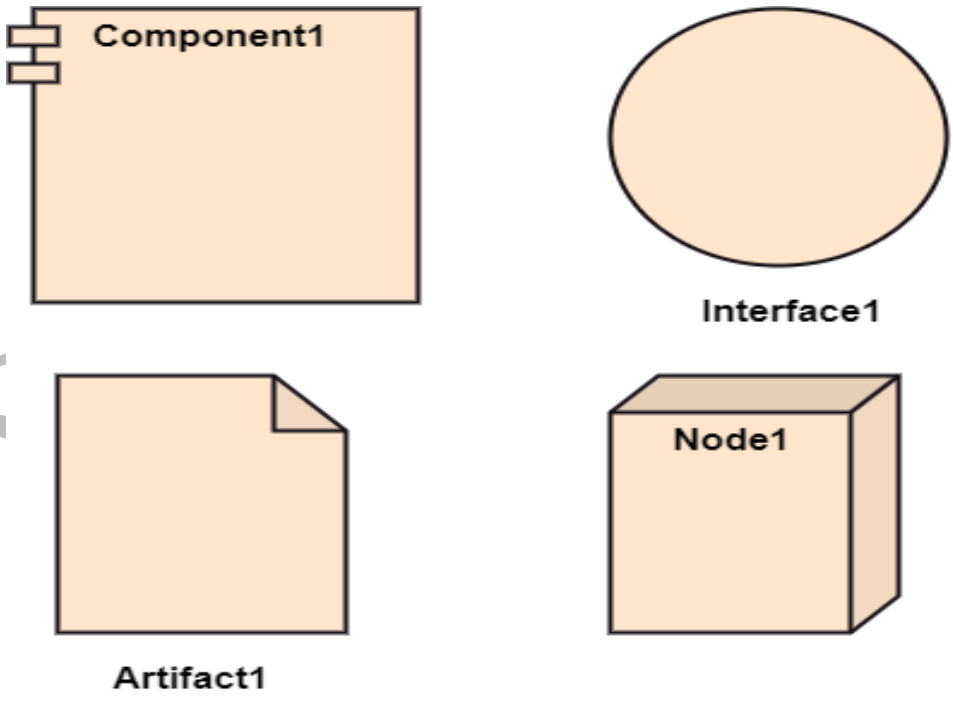
The main purpose of the deployment diagram is to represent how software is installed on the hardware component. It depicts in what manner a software interacts with hardware to perform its execution.
Both the deployment diagram and the component diagram are closely interrelated to each other as they focus on software and hardware components. The component diagram represents the components of a system, whereas the deployment diagram describes how they are actually deployed on the hardware.
The deployment diagram does not focus on the logical components of the system, but it put its attention on the hardware topology.
Following are the purposes of deployment diagram enlisted below:

1. To envision the hardware topology of the system.
2. To represent the hardware components on which the software components are installed.
3. To describe the processing of nodes at the runtime.

Symbol and notation of Deployment diagram

The deployment diagram consist of the following notations:

1. A component
2. An artifact
3. An interface
4. A node



Artifact--

An artifact represents the specification of a concrete real-world entity related to software development. You can use the artifact to describe a framework which is used during the software development process or an executable file. Artifacts are deployed on the nodes. The most common artifacts are as follows,

1. Source files
2. Executable files
3. Database tables
4. Scripts
5. DLL files
6. User manuals or documentation
7. Output files

Artifacts are deployed on the nodes. It can provide physical manifestation for any UML element. Generally, they manifest components. Artifacts are labeled with the stereotype **<<artifact>>,** and it may have an artifact icon on the top right corner.

Each artifact has a filename in its specification that indicates the physical location of the artifact. An artifact can contain another artifact. It may be dependent on one another.

Artifacts have their properties and behavior that manipulates them.

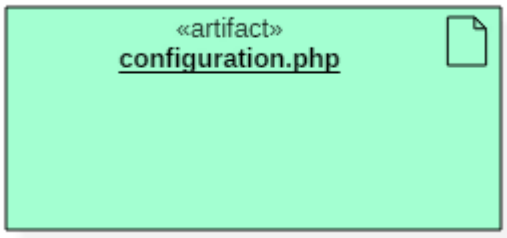
Generally, an artifact is represented as follows in the unified modeling language.

artifact

Artifact Instances

An artifact instance represents an instance of a particular artifact. An artifact instance is denoted with same symbol as that of the artifact except that the name is underlined. UML diagram allows this to differentiate between the original artifact and the instance. Each physical copy or a file is an instance of a unique artifact.

Generally, an artifact instance is represented as follows in the unified modeling language.



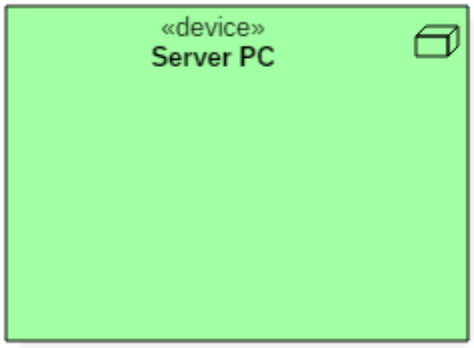
artifact instance

Node--

Node is a computational resource upon which artifacts are deployed for execution. A node is a physical thing that can execute one or more artifacts. A node may vary in its size depending upon the size of the project.

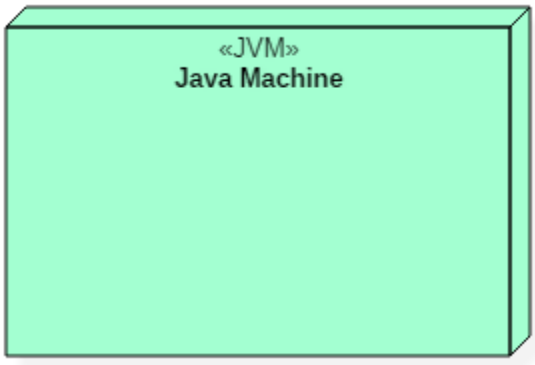
Node is an essential UML element that describes the execution of code and the communication between various entities of a system. It is denoted by a 3D box with the node-name written inside of it. Nodes help to convey the hardware which is used to deploy the software.
An association between nodes represents a communication path from which information is exchanged in any direction.
Generally, a node has two stereotypes as follows:

- **<< device >>**
  It is a node that represents a physical machine capable of performing computations. A device can be a router or a server PC. It is represented using a node with stereotype <<device>>.
  In the UML model, you can also nest one or more devices within each other.
  Following is a representation of a device in UML:



  device node
- **<< execution environment >>**
  It is a node that represents an environment in which software is going to execute. For example, Java applications are executed in java virtual machine (JVM). JVM is considered as an execution environment for Java applications. We can nest an execution environment into a device node. You can net more than one execution environments in a single device node.

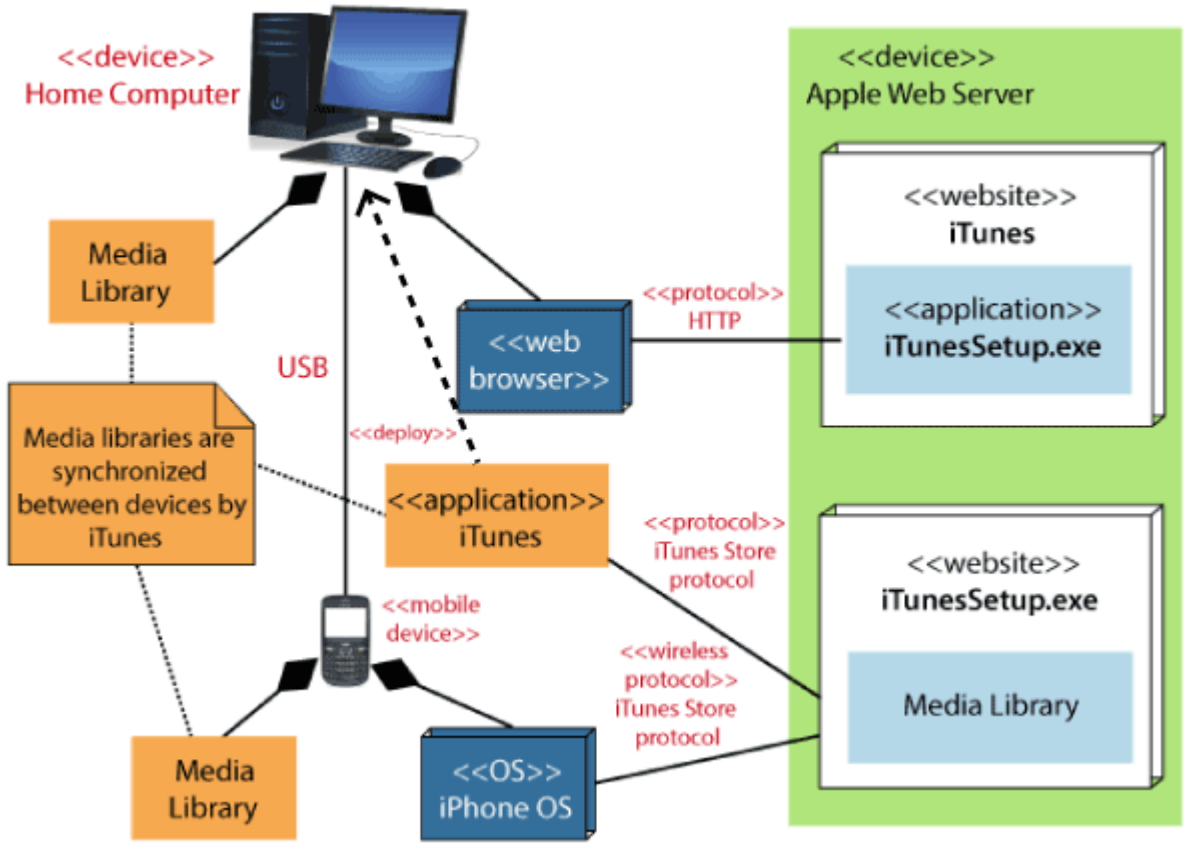
Following is a representation of an execution environment in UML:



execution environment node

A deployment diagram for the Apple iTunes application is given below.
The iTunes setup can be downloaded from the iTunes website, and also it can be installed on

the home computer. Once the installation and the registration are done, iTunes application can easily interconnect with the Apple iTunes store. Users can purchase and download music, video, TV serials, etc. and cache it in the media library.
Devices like Apple iPod Touch and Apple iPhone can update its own media library from the computer with iTunes with the help of USB or simply by downloading media directly from the Apple iTunes store using wireless protocols, for example; Wi-Fi, 3G, or EDGE.



Examples---

Following is a sample deployment diagram to provide 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 connects to the application using the Internet. The control flows from the caching server to the clustered environment.
The following deployment diagram has been drawn considering all the points mentioned above.

Deployment diagram of an order management system