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.

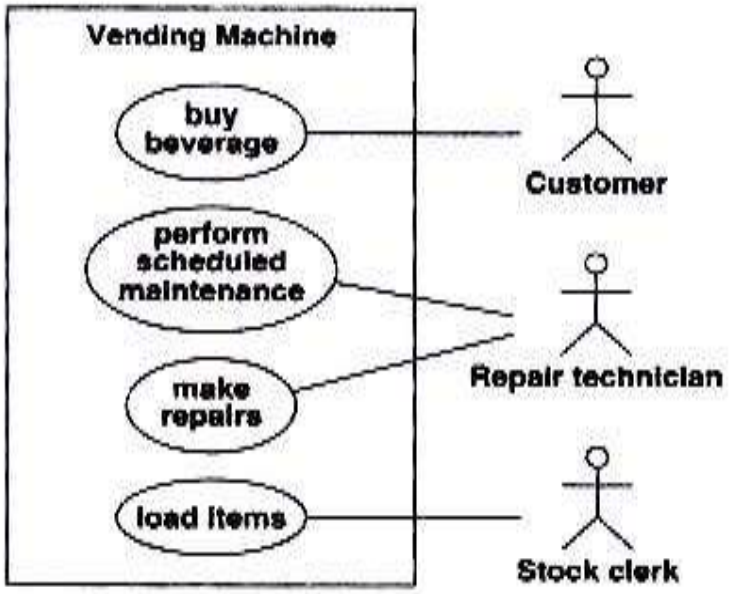


Figure **Use case diagram for a vending machine.** A system involves a set of use cases and a set of actors.

Guidelines for Use Case Models--

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

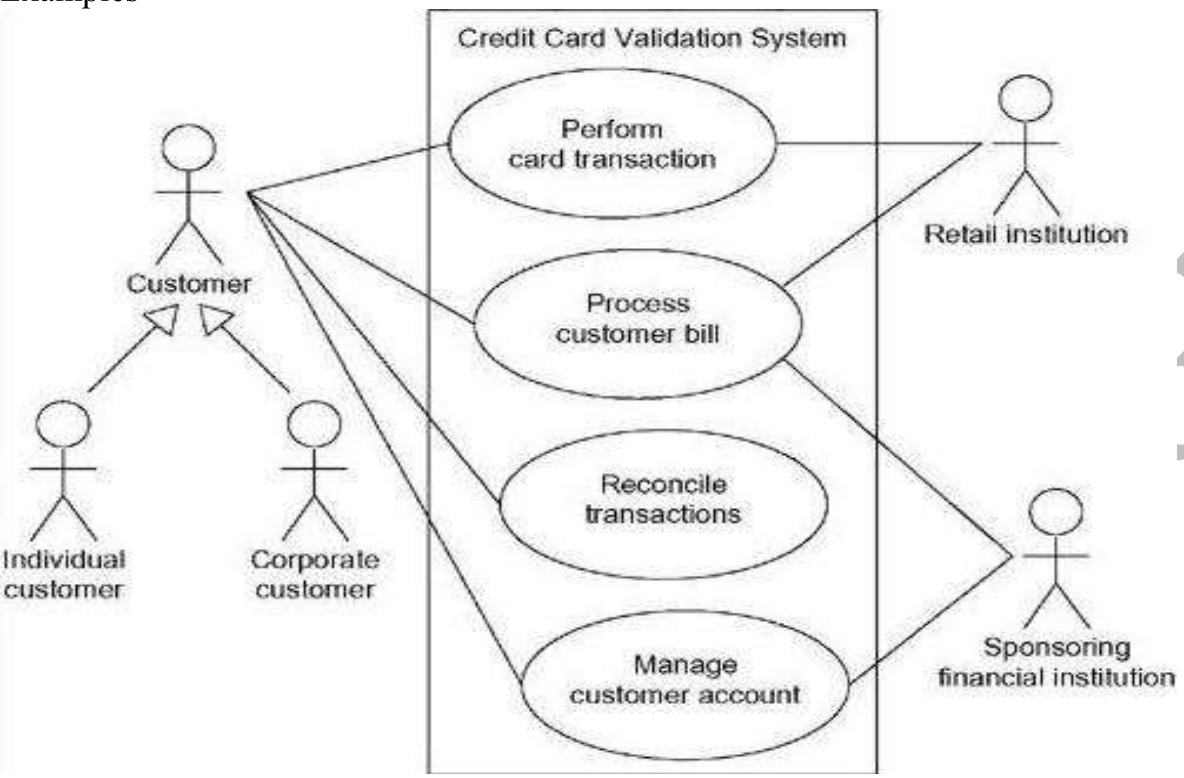
■ Each use case must provide value to users. A use case should represent a complete transaction that provides value to users and should not be defined too narrowly.

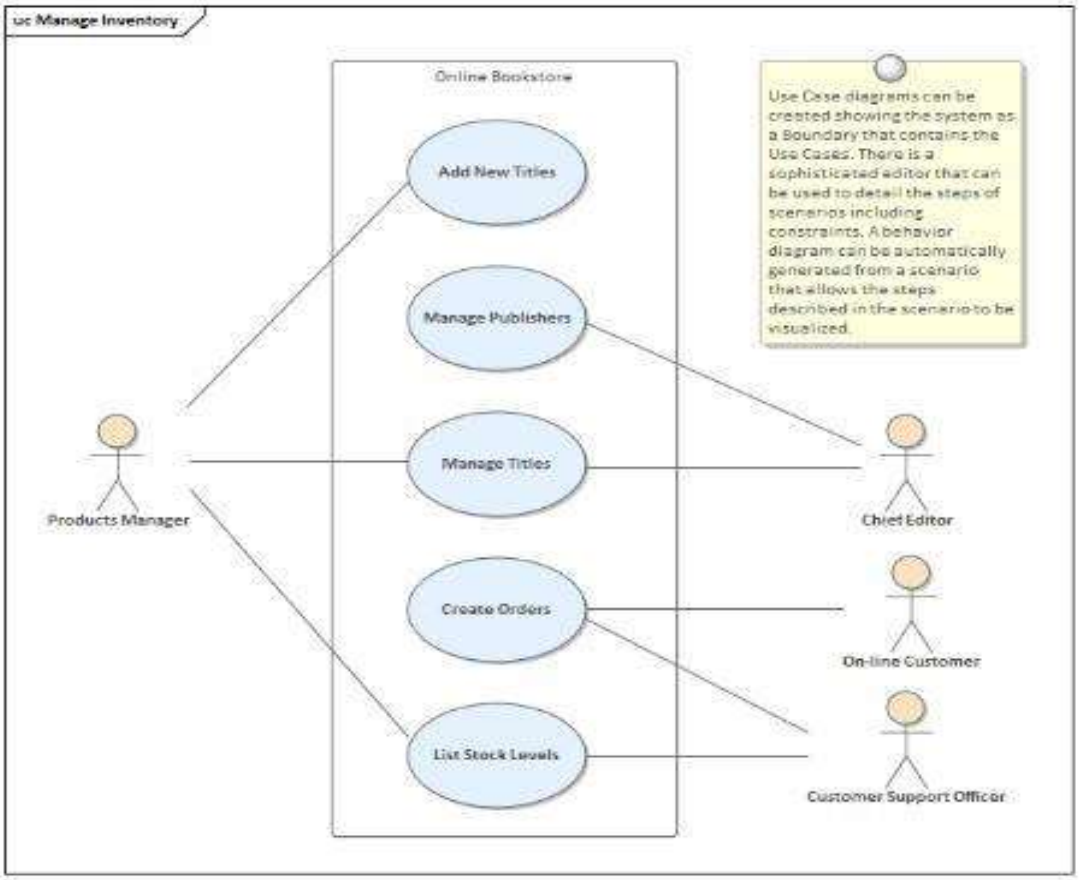
■ Relate use cases and actors. Every use case should have at least one actor, and every actor should participate in at least one use case. A use case may involve several actors, and an actor may participate in several use cases.

■ 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---


Credit Card Validation System
Perform card transaction
Process customer bill
Reconcile transactions
Manage customer account
Customer
Individual customer
Corporate customer
Retail institution
Sponsoring financial institution


uc Manage Inventory
Online Bookstore
Add New Titles
Manage Publishers
Manage Titles
Create Orders
List Stock Levels
Use Case diagrams can be created showing the system as a Boundary that contains the Use Cases. There is a sophisticated editor that can be used to detail the steps of scenarios including constraints. A behavior diagram can be automatically generated from a scenario that allows the steps described in the scenario to be visualized.
Products Manager
Chief Editor
On-line Customer
Customer Support Officer

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.

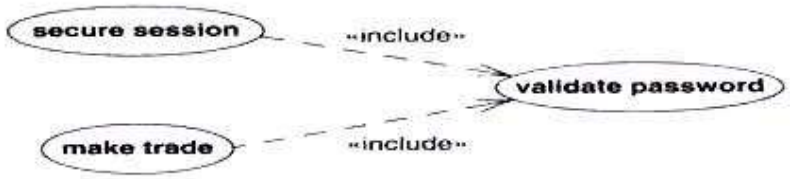


Figure **Use case inclusion.** The *include* relationship lets a base use case incorporate behavior from another use case.

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.

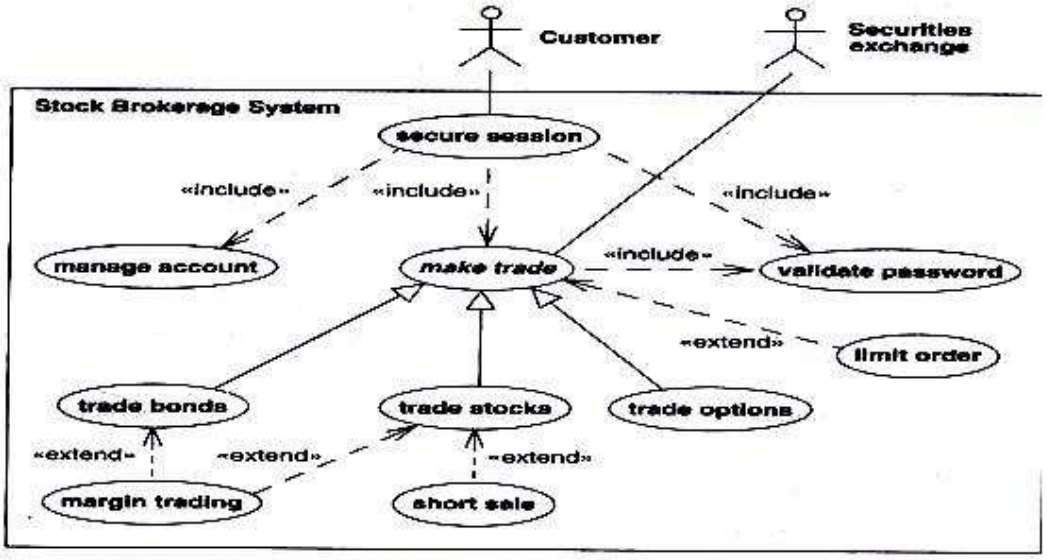
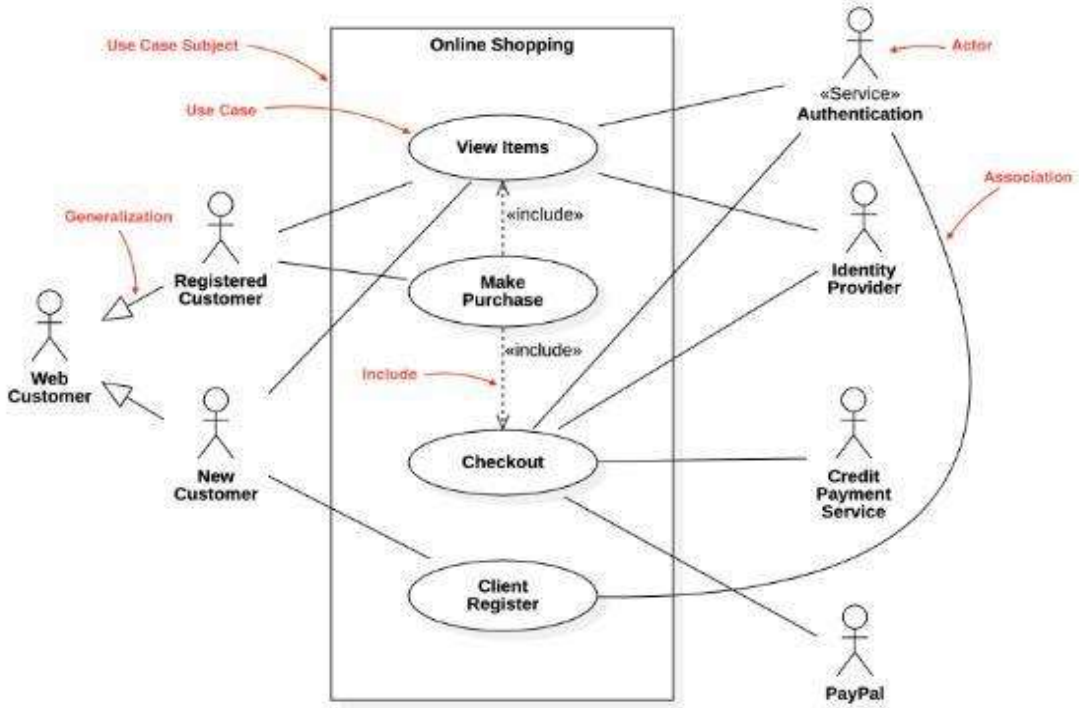
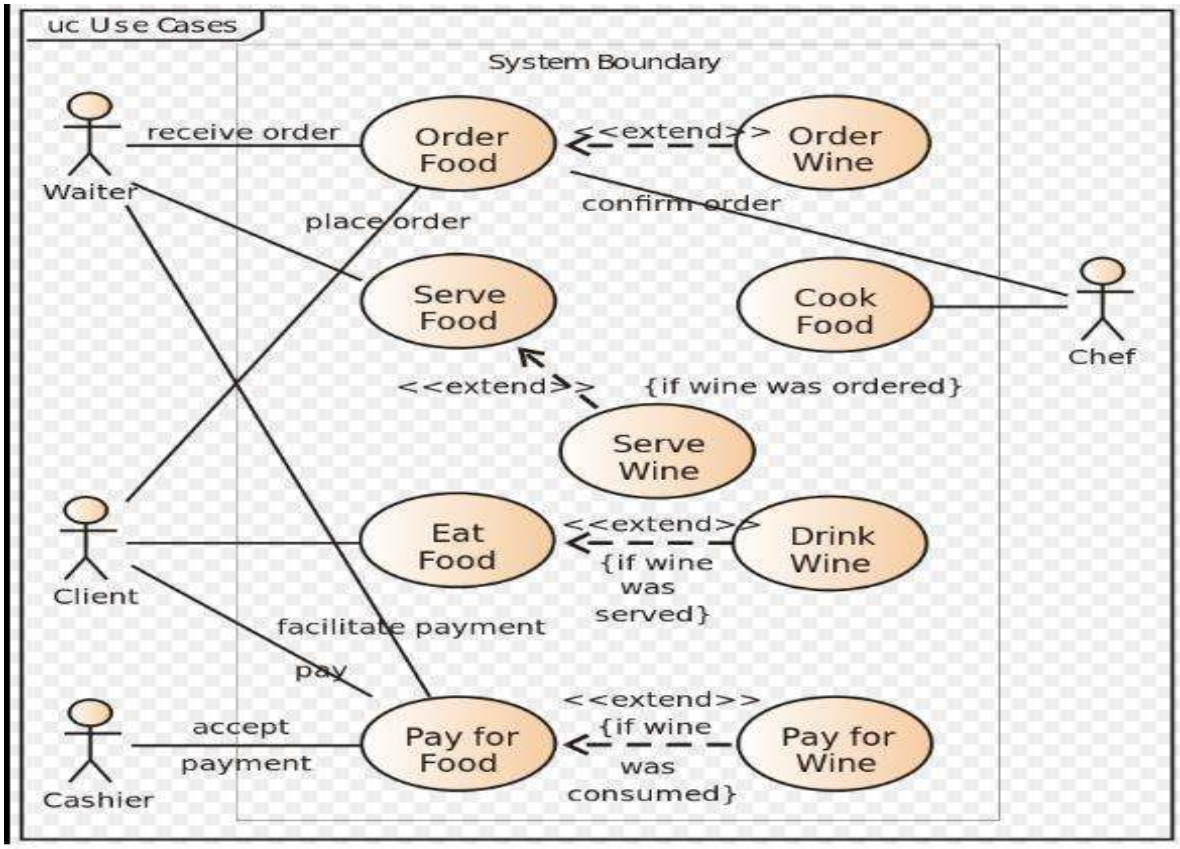


Figure **Use case relationships.** A single use case diagram may combine several kinds of relationships.

Examples---





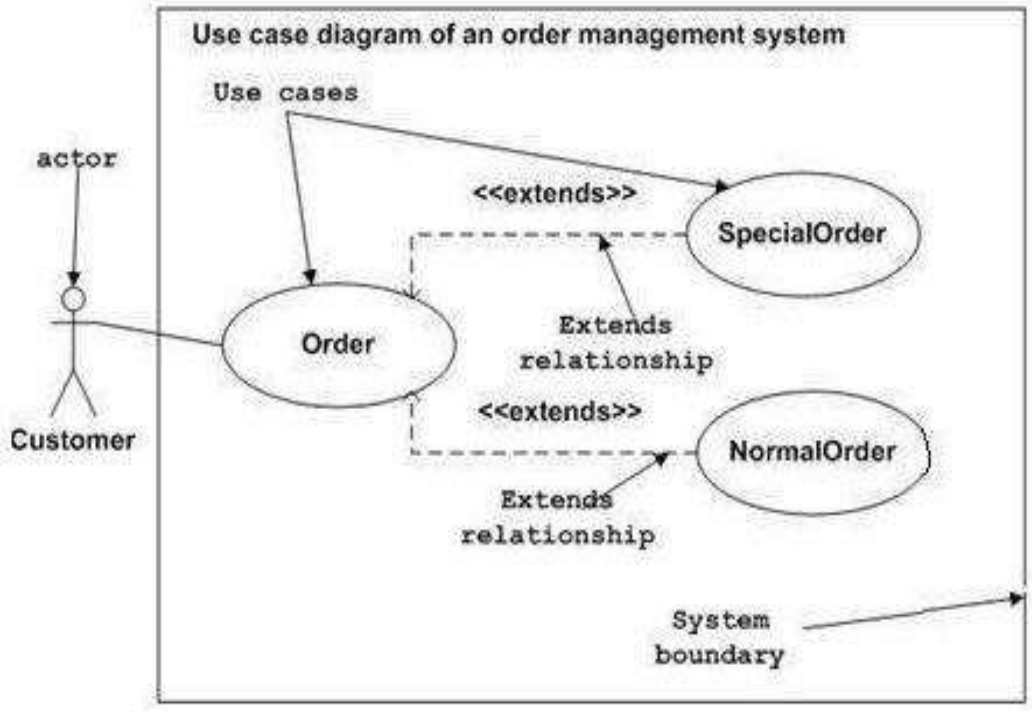
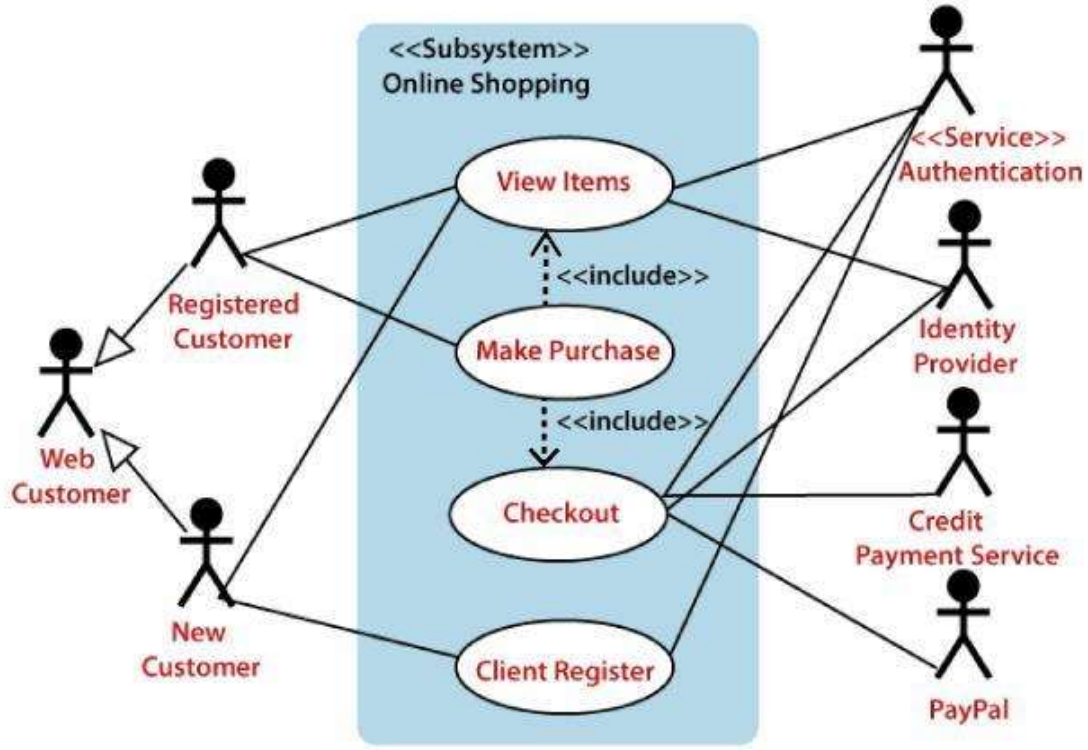
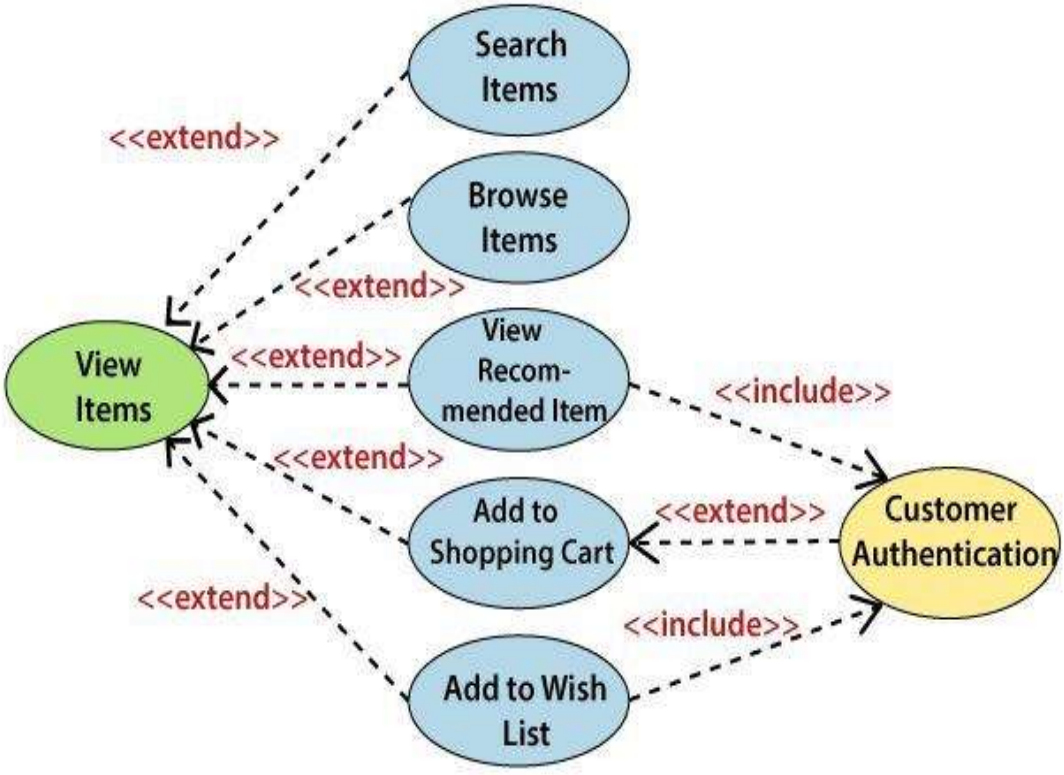
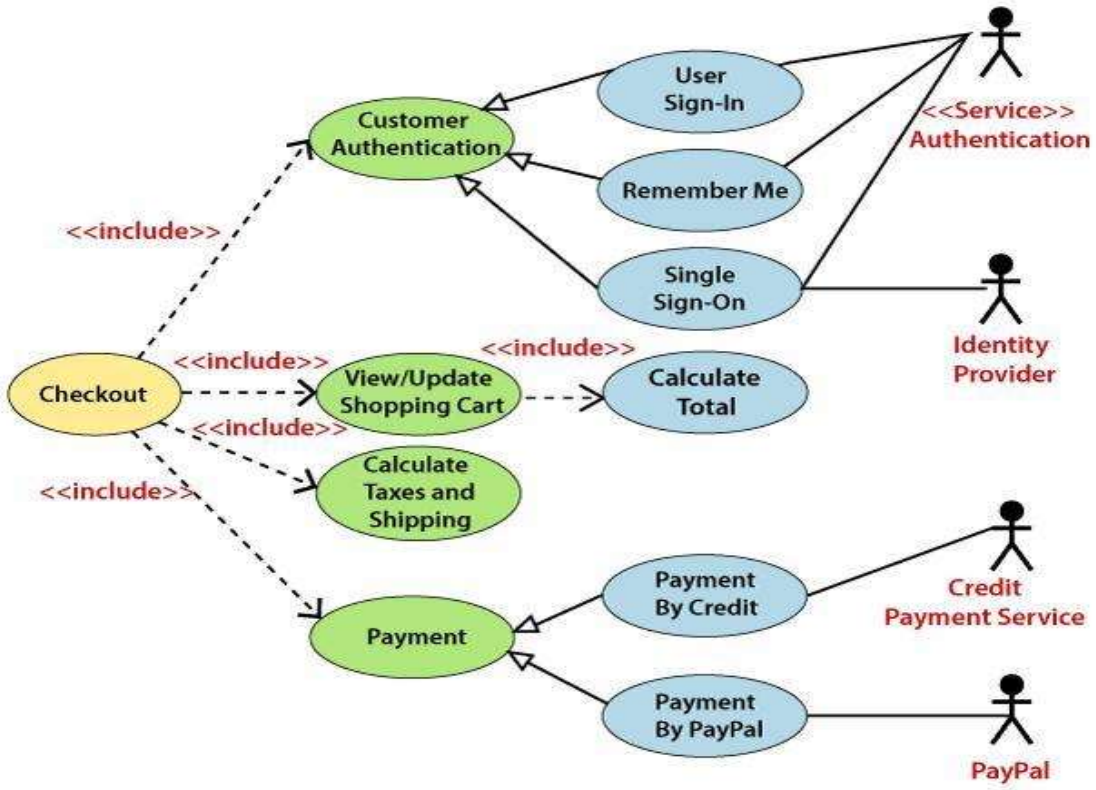


Figure: Sample Use Case diagram


<<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.

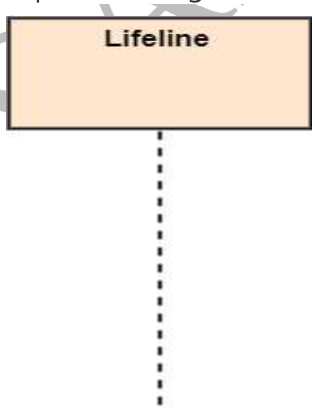
The purpose of interaction diagram is −

- 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

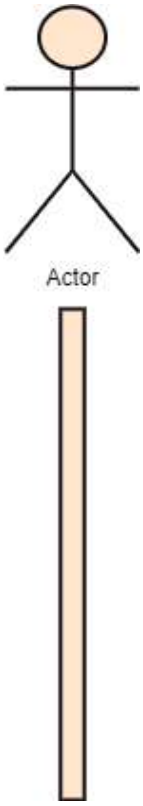
Lifeline

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



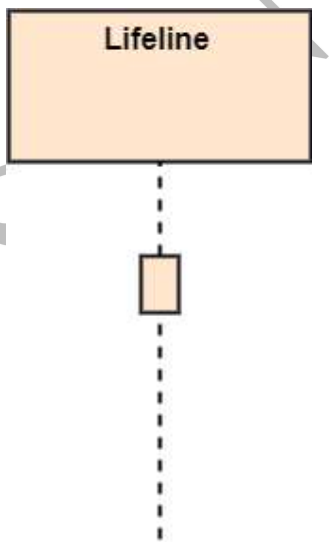
Actor

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.

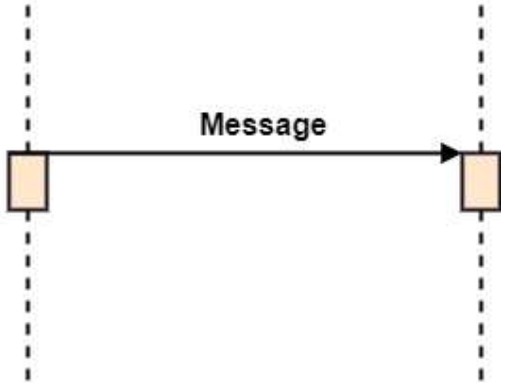


Messages--

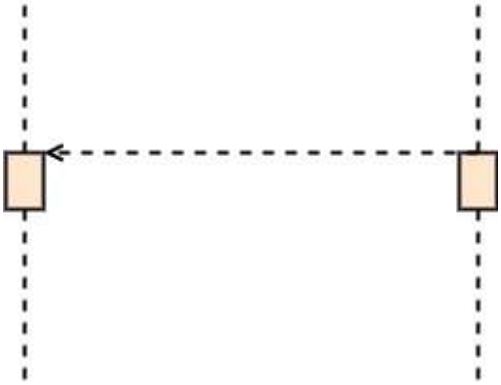
The messages depict the interaction between the objects and are represented by arrows. They are in the sequential order on the lifeline. The core of the sequence diagram is formed by messages and lifelines.

Following are types of messages enlisted below:

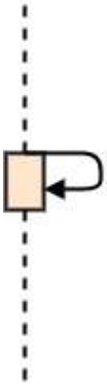
- **Call Message:** It defines a particular communication between the lifelines of an interaction, which represents that the target lifeline has invoked an operation.



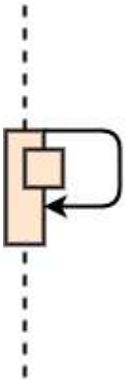
- **Return Message:** It defines a particular communication between the lifelines of interaction that represent the flow of information from the receiver of the corresponding caller message.



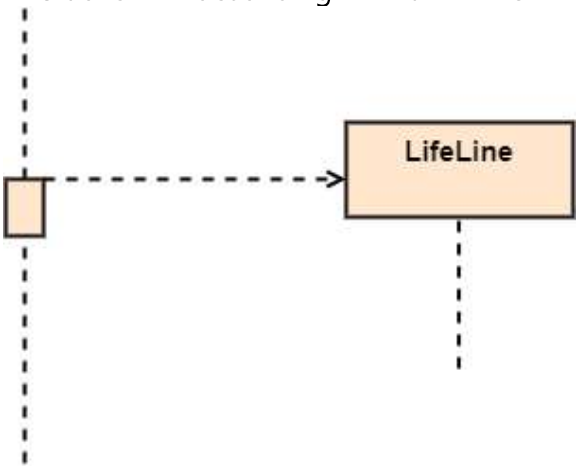
- **Self Message:** It describes a communication, particularly between the lifelines of an interaction that represents a message of the same lifeline, has been invoked.



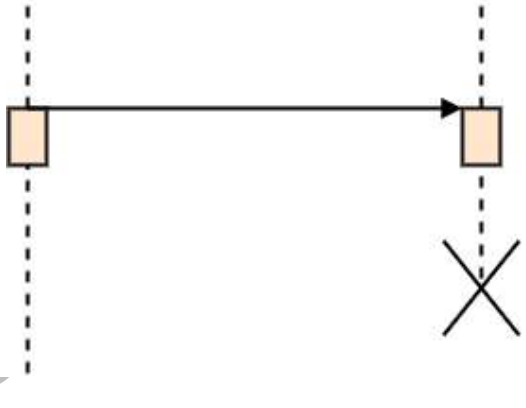
- **Recursive Message:** A self message sent for recursive purpose is called a recursive message. In other words, it can be said that the recursive message is a special case of the self message as it represents the recursive calls.



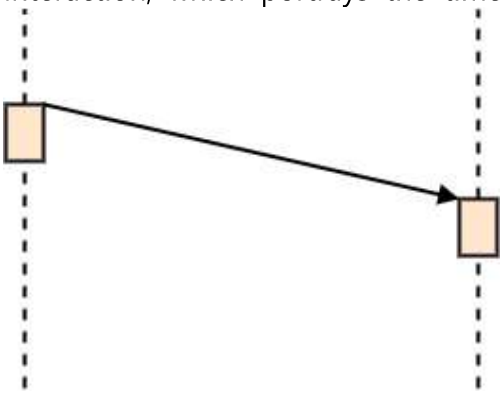
- **Create Message:** It describes a communication, particularly between the lifelines of an interaction describing that the target (lifeline) has been instantiated.



- **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.

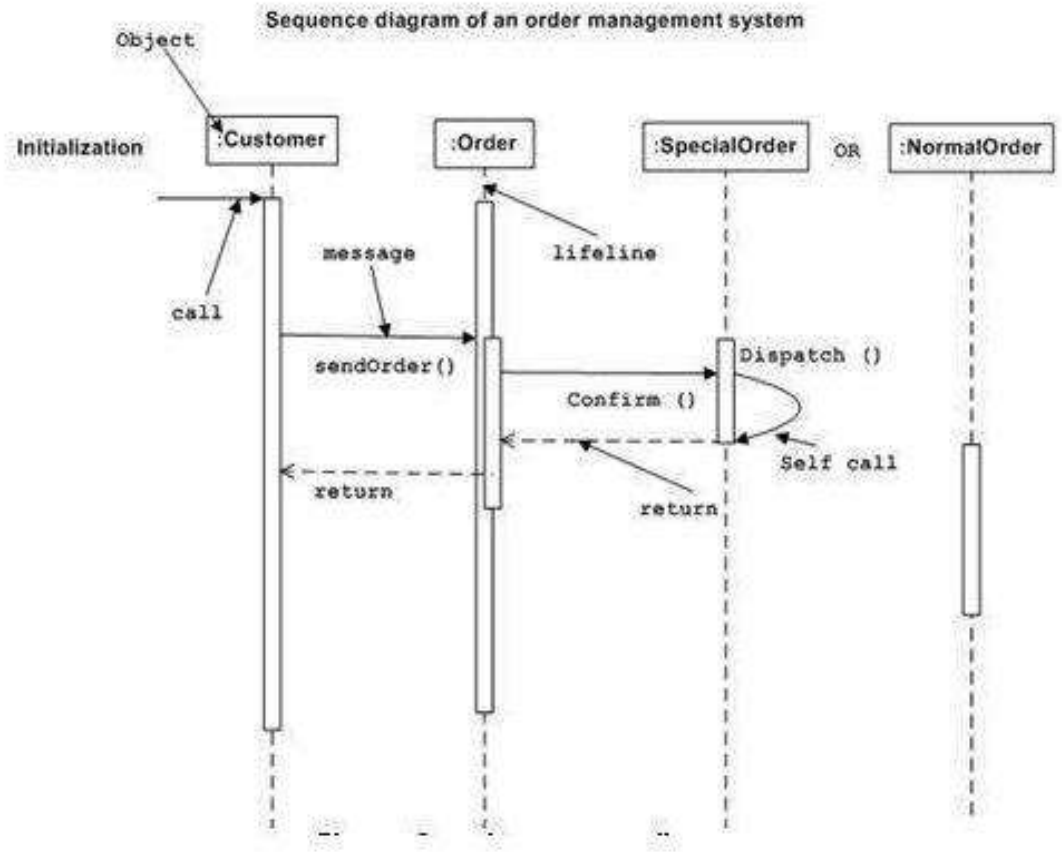


The Sequence Diagram---

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.

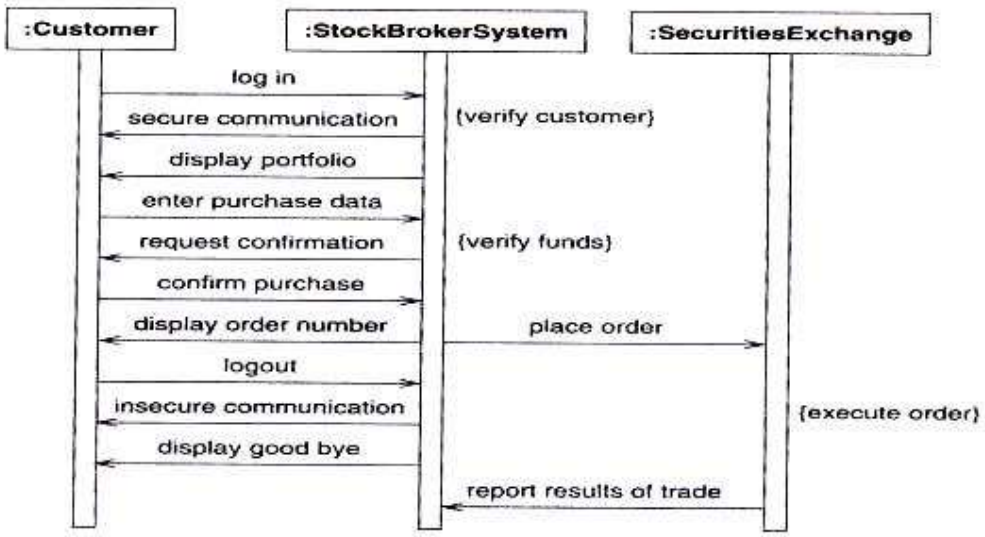


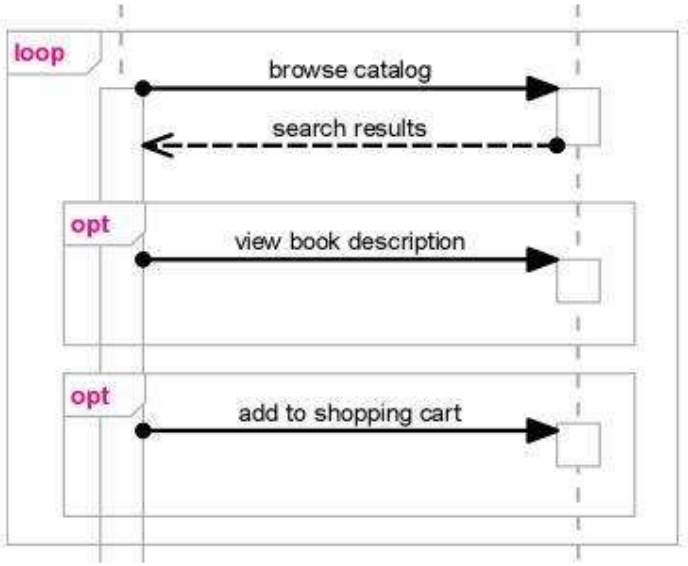
Figure **Sequence diagram for a session with an online stock broker.** A sequence diagram shows the participants in an interaction and the sequence of messages among them.

Fragments--

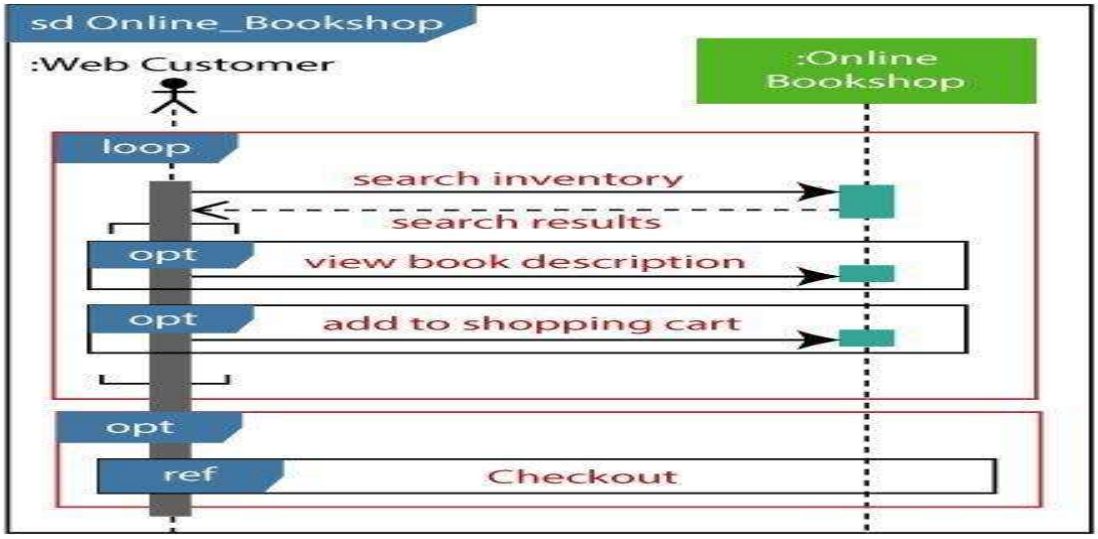
A fragment in a UML sequence diagram can be used as an overlay over a number of messages to specify whether these are a sequence, alternatives, optional, parallel, or a loop. There are a few more kinds of fragments specified in the UML standard that can also be used.

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

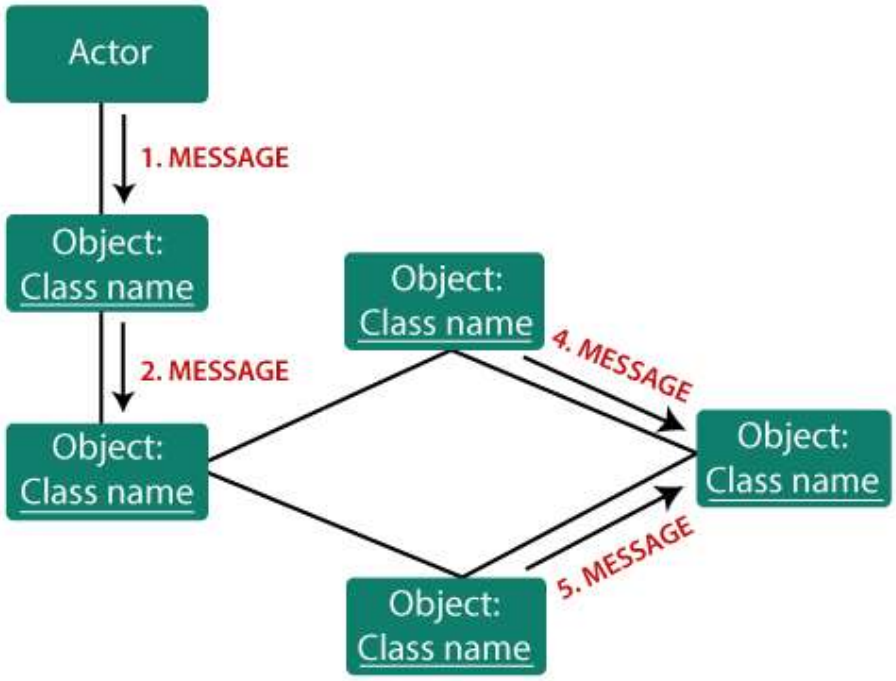
The collaboration diagram is used to show the relationship between the objects in a system. Both the sequence and the collaboration diagrams represent the same information but differently. Instead of showing the flow of messages, it depicts the architecture of the object residing in the system as it 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.

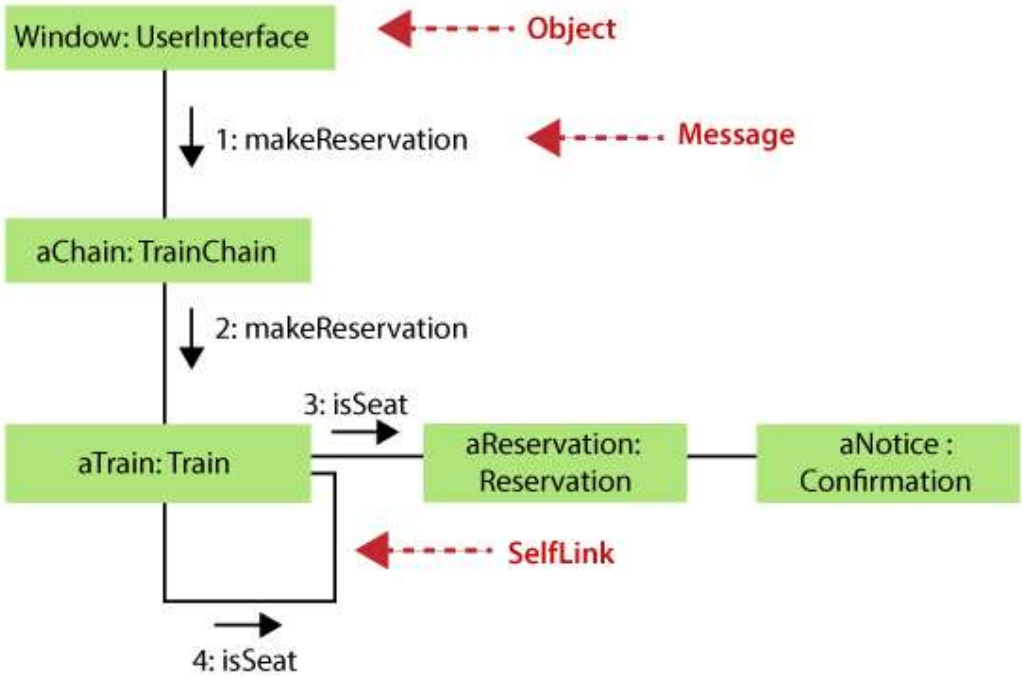
Components of a collaboration diagram



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



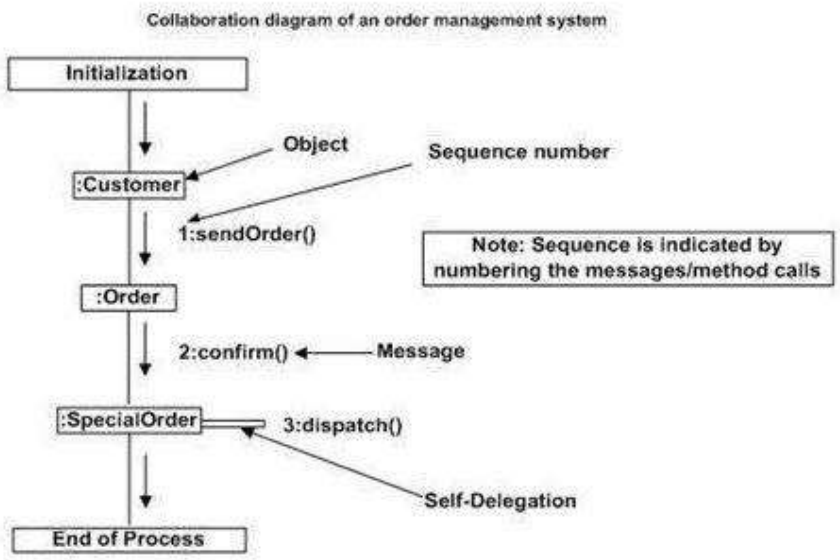
Benefits of a Collaboration Diagram

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

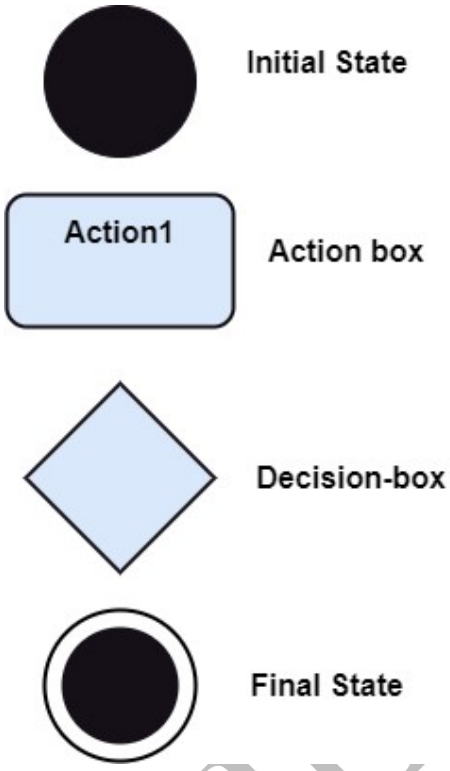
Activity diagram constitutes following notations:

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.

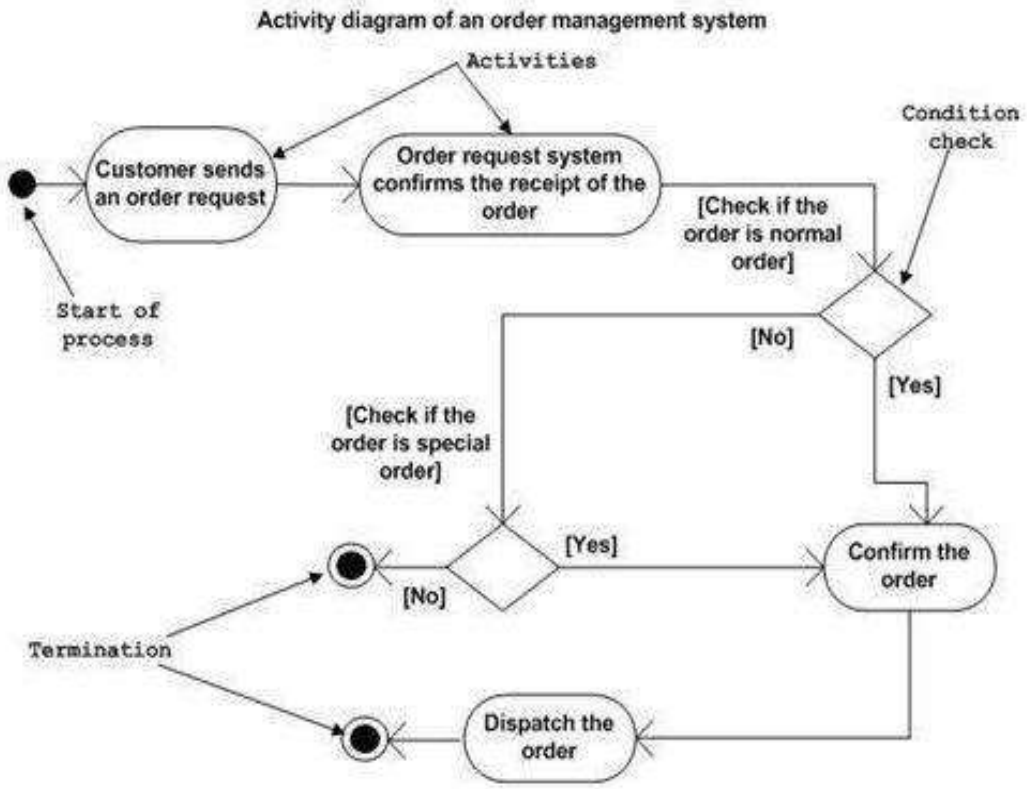


Figure below shows an activity diagram for the processing of a stock trade order that has been received by an online stock broker. The elongated ovals show activities and the arrows show their sequencing. The diamond shows a decision point and the heavy bar shows splitting or merging of concurrent threads.

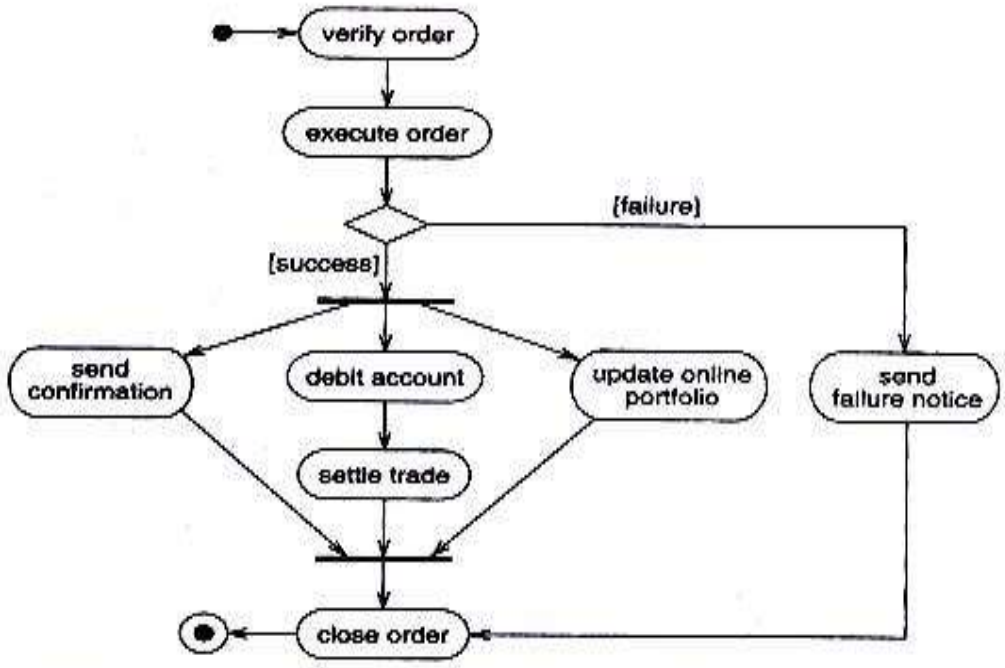
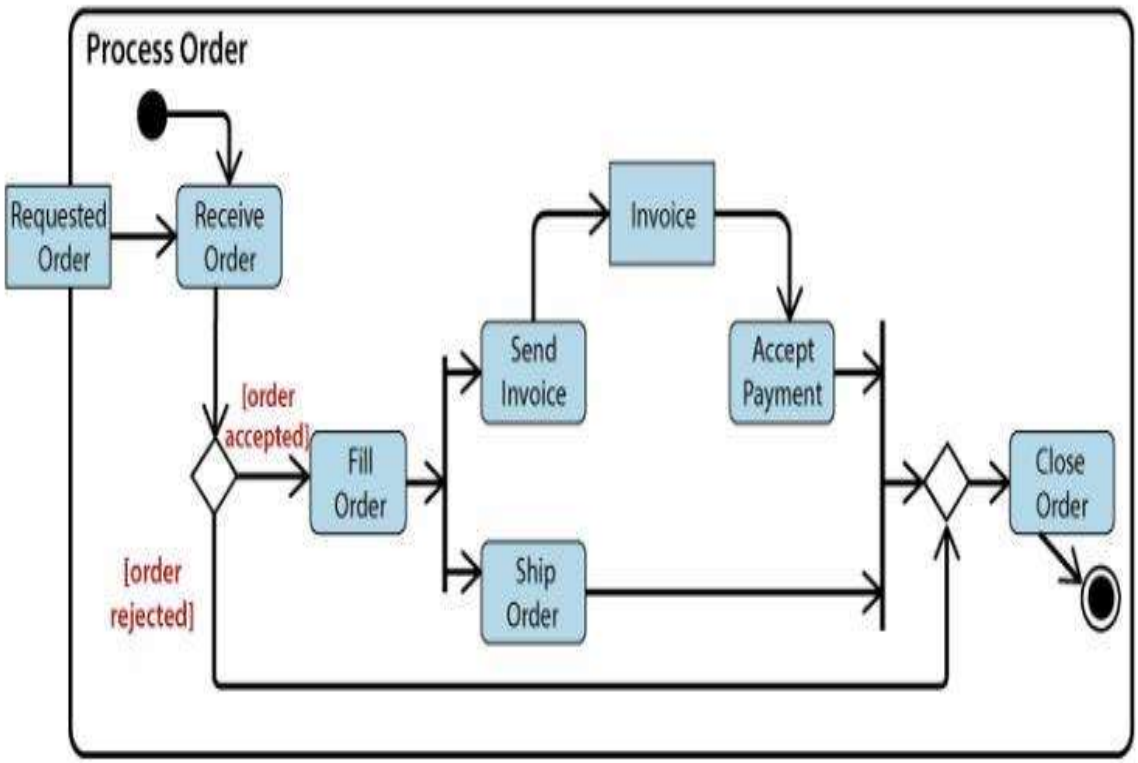


Figure **Activity diagram for stock trade processing.** An activity diagram shows the sequence of steps that make up a complex process.



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.

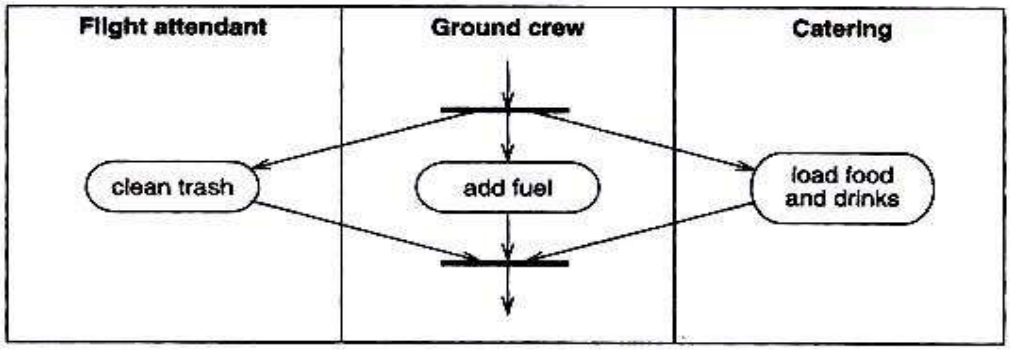


Figure **Activity diagram with swimlanes.** Swimlanes can show organizational responsibility for activities.

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.

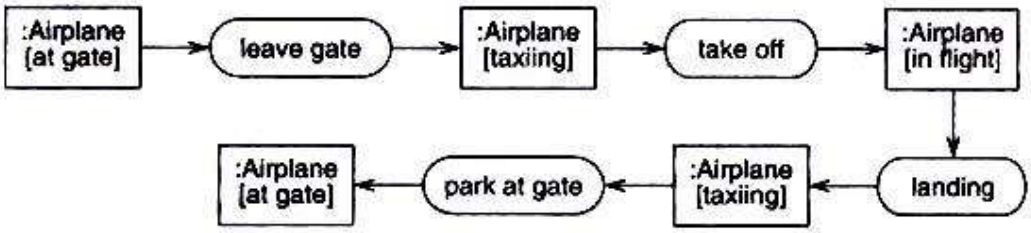


Figure **Activity diagram with object flows.** An activity diagram can show the objects that are inputs or outputs of activities.

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 −

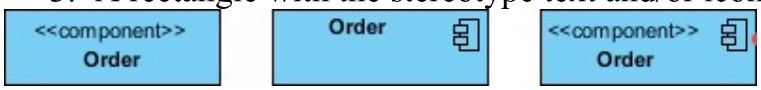
- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

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

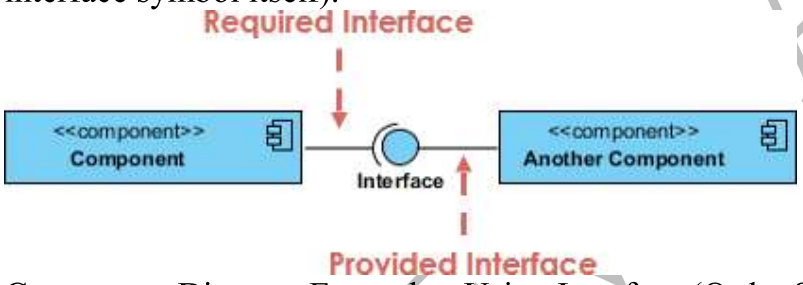


Interface

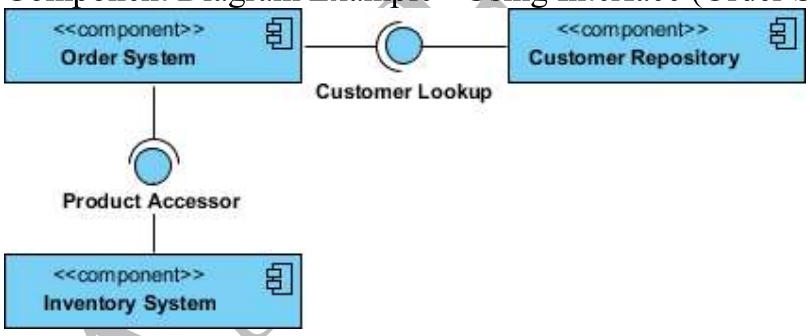
In the example below shows two type of component interfaces:

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).

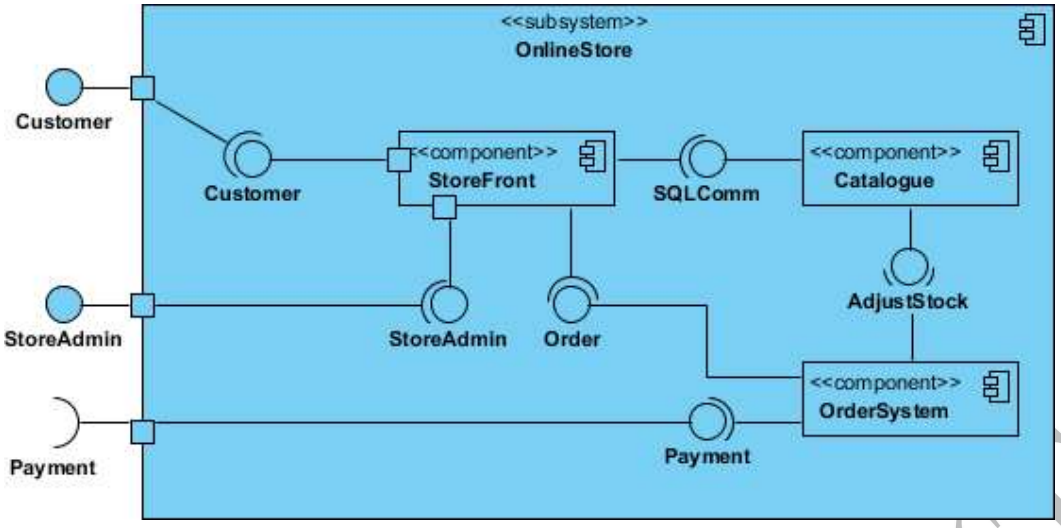


Component Diagram Example - Using Interface (Order System)



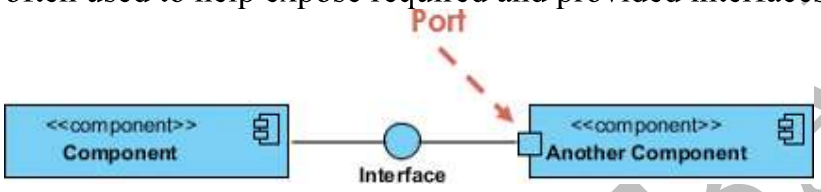
Subsystems

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.

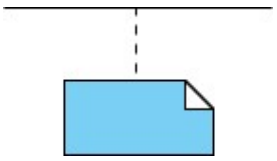
- If a composite is deleted, all of its parts are normally deleted with it.

Aggregation

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

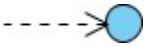
Constraint

- 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).

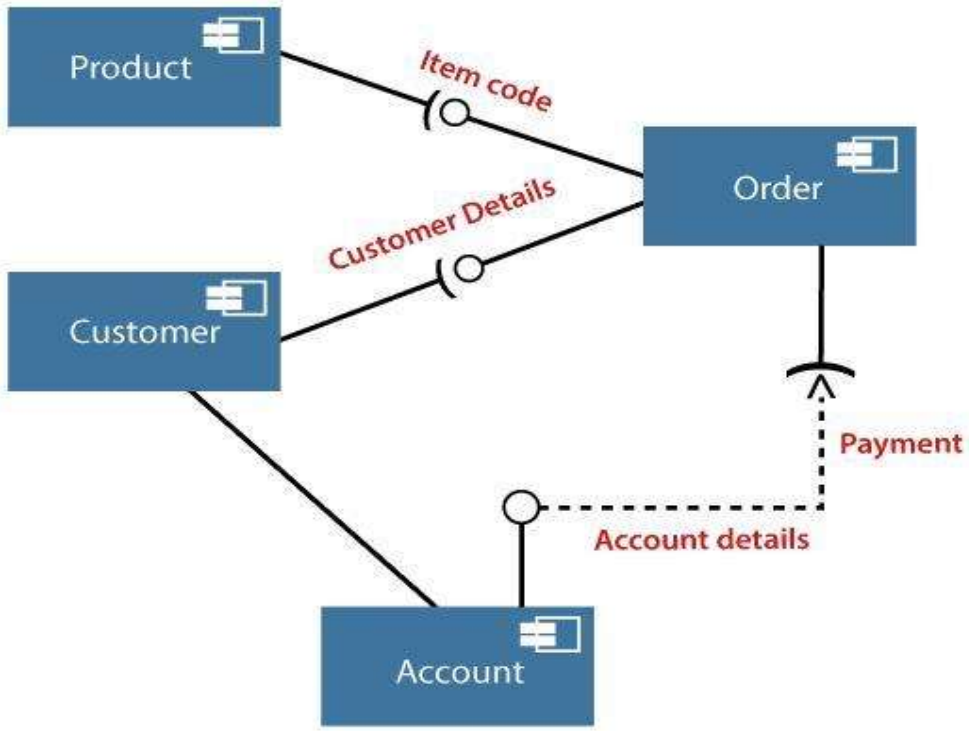


Links:

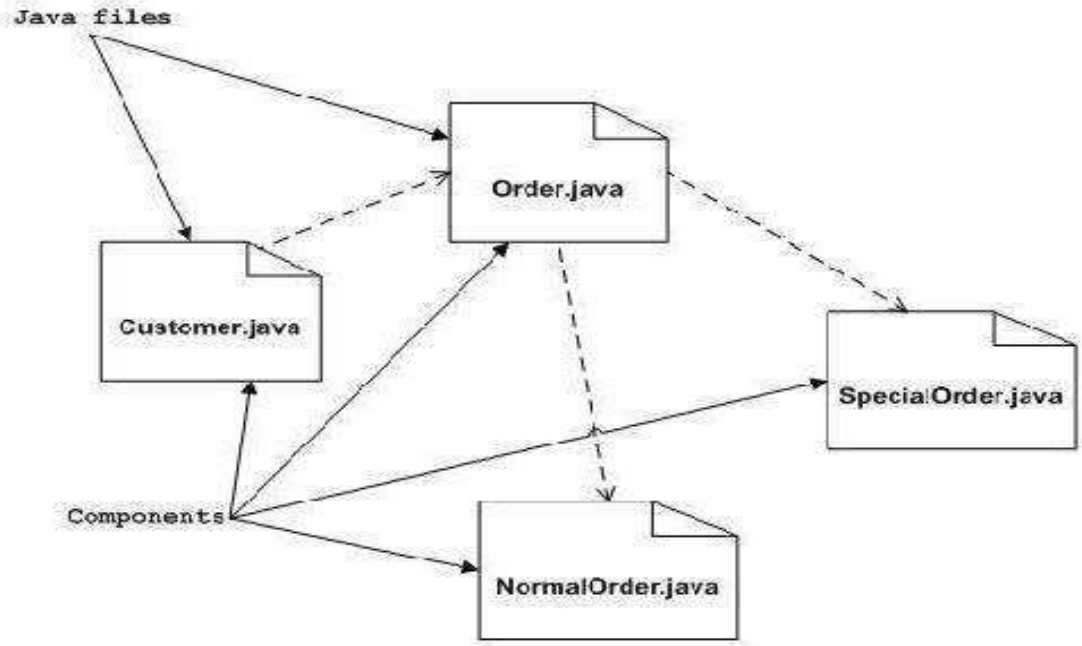
- A generalization is a taxonomic relationship between a more general classifier and a more specific classifier.
- Each instance of the specific classifier is also an indirect instance of the general classifier.

- Thus, the specific classifier inherits the features of the more general classifier.

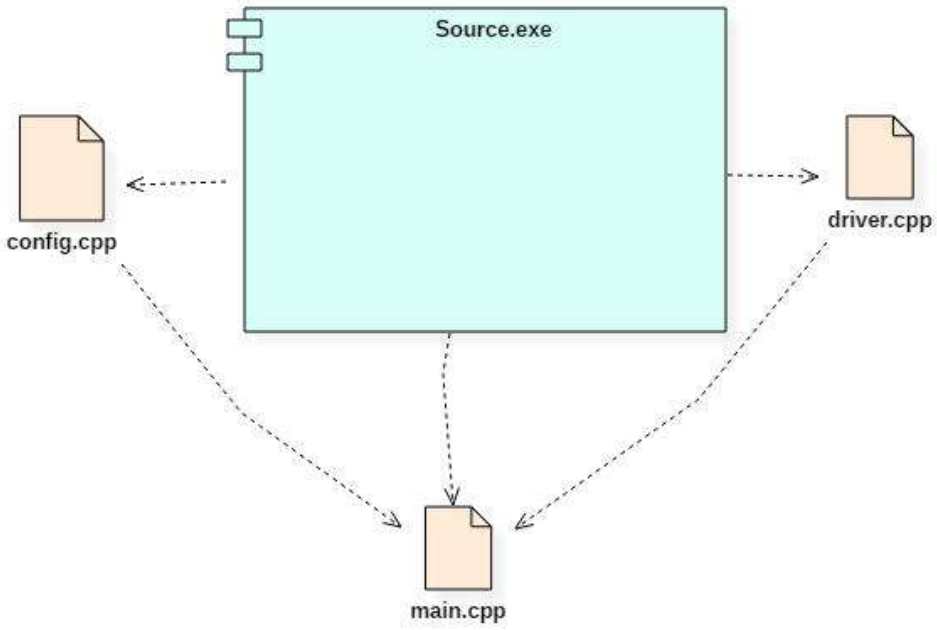
A component diagram for an online shopping system is given below:



Component diagram of an order management system



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.

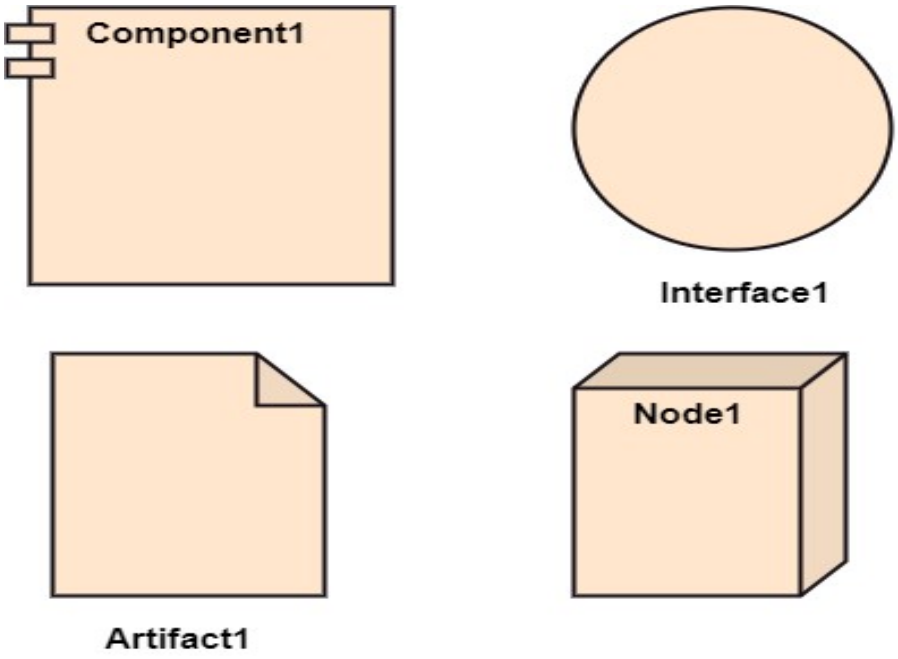
Following are the purposes of deployment diagram enlisted below:

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.

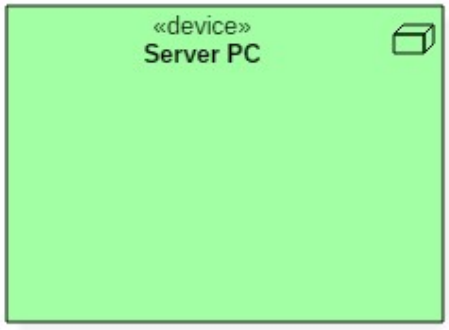
Generally, a node has two stereotypes as follows:

- **<< 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.

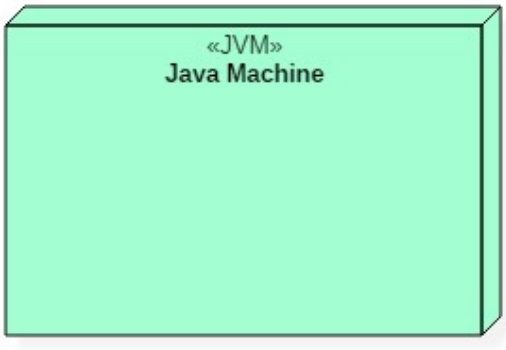
  Following is a representation of a device in UML:



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:

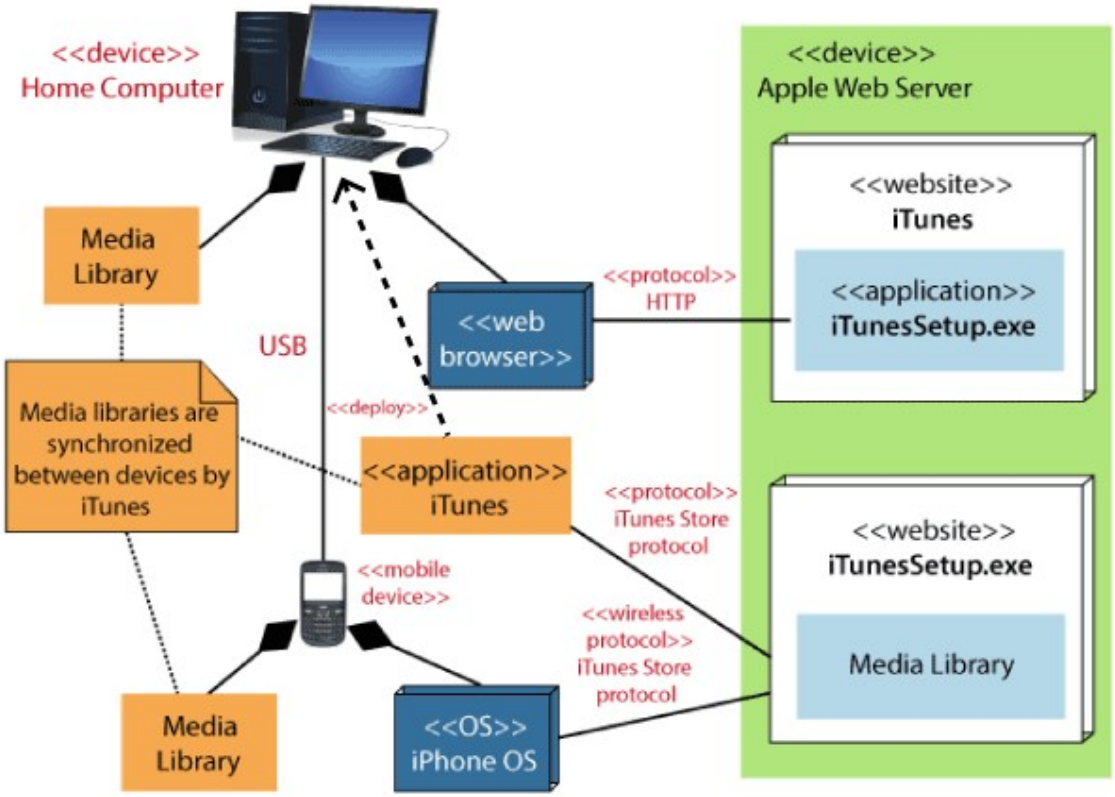


execution environment node

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