UNIT II – SOFTWARE DESIGN AND UML DIAGRAMS [9 hours]

Design Principles (Modularity, Reusability, Abstraction), UML Diagrams: Use Case, Class, Activity, Sequence, Introduction to Design Patterns (Singleton, Factory, MVC),Building Simple System Architecture (Layered & Client-Server).

UML DIAGRAMS

Unified Modeling Language (UML) is a general purpose modelling language. The main aim of UML is to define a standard way to visualize the way a system has been designed. It is quite similar to blueprints used in other fields of engineering.

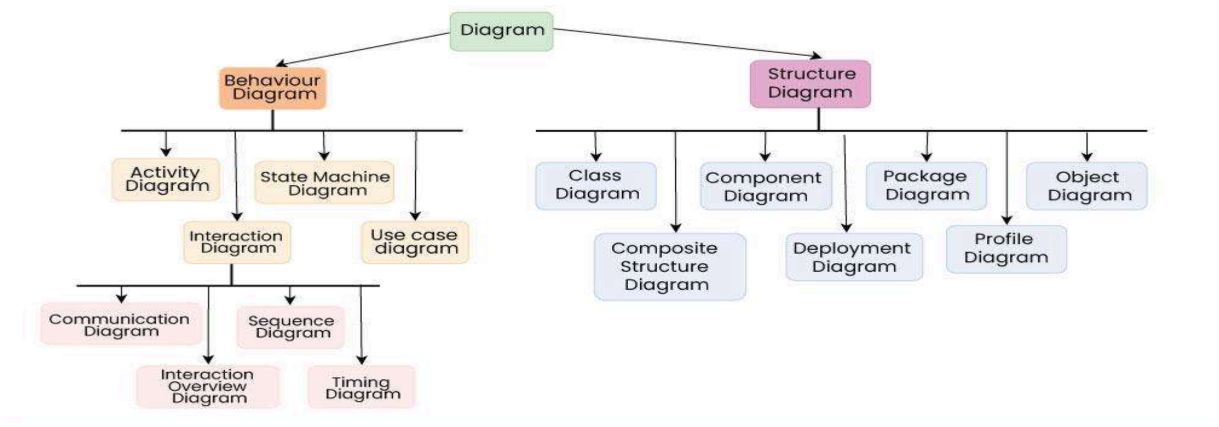
UML is not a programming language, it is rather a visual language. We use UML diagrams to portray the behavior and structure of a system. UML helps software engineers, businessmen and system architects with modelling, design and analysis. The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997. It's been managed by OMG ever since. The International Organization for Standardization (ISO) published UML as an approved standard in 2005. UML has been revised over the years and is reviewed periodically.

Advantages of UML

- Most-Used and Flexible.
- Provides standards for software development.
- Reducing costs to develop diagrams of UML using supporting tools.
- Development time is reduced.
- The past faced issues by the developers no longer exist.
- Has large visual elements to construct and easy to follow.

Types of UML Diagrams

UML is linked with object-oriented design and analysis. UML makes use of elements and forms associations between them to form diagrams. Diagrams in UML can be broadly classified as:



Structural UML Diagrams

1. **Class Diagram** – The most widely used UML diagram is the class diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing the system's classes, their methods and attributes. Class diagrams also help us identify relationships between different classes or objects.
2. **Composite Structure Diagram** – We use composite structure diagrams to represent the internal structure of a class and its interaction points with other parts of the system. A composite structure diagram represents the relationship between parts and their configuration which determine how the classifier (class, a component, or a deployment node) behaves. They represent the internal structure of a structured classifier making the use of parts, ports, and connectors. We can also model collaborations using composite structure diagrams. They are similar to class diagrams except they represent individual parts in detail as compared to the entire class.
3. **Object Diagram** – An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behaviour of the system at a particular instant. An object diagram is similar to a class diagram except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand,

an Object Diagram represents specific instances of classes and relationships between them at a point of time.

4. **Component Diagram** – Component diagrams are used to represent how the physical components in a system have been organized. We use them for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development. Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.
5. **Deployment Diagram** – Deployment Diagrams are used to represent system hardware and its software. It tells us what hardware components exist and what software components run on them. We illustrate system architecture as distribution of software artifacts over distributed targets. An artifact is the information that is generated by system software. They are primarily used when a software is being used, distributed or deployed over multiple machines with different configurations.
6. **Package Diagram** – We use Package Diagrams to depict how packages and their elements have been organized. A package diagram simply shows us the dependencies between different packages and internal composition of packages. Packages help us to organise UML diagrams into meaningful groups and make the diagram easy to understand. They are primarily used to organise class and use case diagrams.

Behavior Diagrams

1. **State Machine Diagrams** – A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as State machines and State-chart Diagrams . These terms are often used interchangeably. So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.
2. **Activity Diagrams** – We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the

execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram. An activity diagram focuses on the condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.

3. **Use Case Diagrams** – Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors). A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high level view of what the system or a part of the system does without going into implementation details.
4. **Sequence Diagram** – A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place. We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.
5. **Communication Diagram** – A Communication Diagram(known as Collaboration Diagram) is used to show sequenced messages exchanged between objects. A communication diagram focuses primarily on objects and their relationships. We can represent similar information using Sequence diagrams, however, communication diagrams represent objects and links in a free form.
6. **Timing Diagram** – Timing Diagram are a special form of Sequence diagrams which are used to depict the behavior of objects over a time frame. We use them to show time and duration constraints which govern changes in states and behavior of objects.
7. **Interaction Overview Diagram** – An Interaction Overview Diagram models a sequence of actions and helps us simplify complex interactions into simpler occurrences. It is a mixture of activity and sequence diagrams.

USE CASE DIAGRAMS

A Use Case Diagram in Unified Modeling Language (UML) is a visual representation that illustrates **the interactions between users (actors) and a system.** It captures the functional requirements of a system, showing how different users engage with various use cases, or specific functionalities, within the system. Use case diagrams provide a high-level overview of a system's behavior, making them useful for stakeholders, developers, and analysts to understand how a system is intended to operate from the user's perspective, and how different processes relate to one another. They are crucial for defining system scope and requirements.

To build one, we use a set of specialized symbols and connectors. An effective use case diagram can help the team discuss and represent:

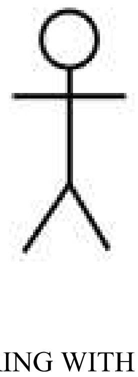
- Scenarios in which system or application interacts with people, organizations, or external systems
- Goals that the system or application helps those entities (known as actors) achieve
- The scope of the system

Use Case Diagram Notations

UML notations provide a visual language that enables software developers, designers, and other stakeholders to communicate and document system designs, architectures, and behaviors in a consistent and understandable manner.

1. Actors

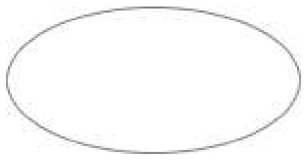
Actors are external entities that interact with the system. These can include users, other systems, or hardware devices. In the context of a Use Case Diagram, actors initiate use cases and receive the outcomes. Proper identification and understanding of actors are crucial for accurately modeling system behavior.



.

2. Use Cases

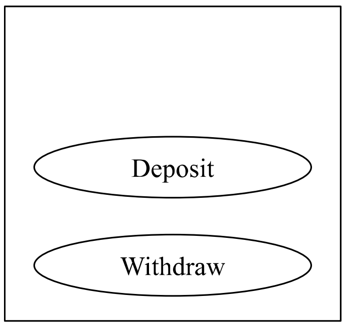
Use cases are like scenes in the play. They represent specific things your system can do. In the online shopping system, examples of use cases could be "Place Order," "Track Delivery," or "Update Product Information". Use cases are represented by ovals.



3. System Boundary

The system boundary is a visual representation of the scope or limits of the system modeling. It defines what is inside the system and what is outside. The boundary helps to establish a clear distinction between the elements that are part of the system and those that are external to it. The system boundary is typically represented by a rectangular box that surrounds all the use cases of the system.

The purpose of system boundary is to clearly outline the boundaries of the system, indicating which components are internal to the system and which are external actors or entities interacting with the system.



Use Case Diagram Relationships

In a Use Case Diagram, relationships play a crucial role in depicting the interactions between actors and use cases. These relationships provide a comprehensive view of the system's functionality and its various scenarios. Let's delve into the key types of relationships and explore examples to illustrate their usage.

1. Association Relationship

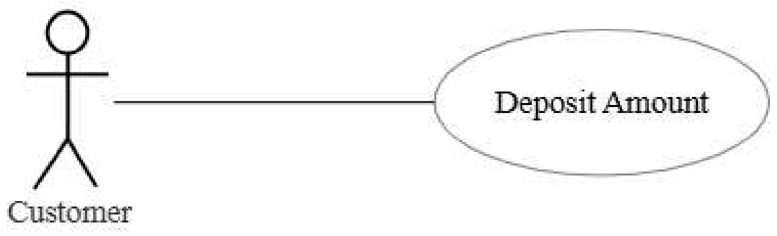
The Association Relationship represents a communication or interaction between an actor and a use case. It is depicted by a line connecting the actor to the use case. This relationship signifies that the actor is involved in the functionality described by the use case.

Example: Online Banking System

Actor: Customer

Use Case: Deposit Amount

Association: A line connecting the "Customer" actor to the "Deposit Amount" use case, indicating the customer's involvement in the funds transfer process.



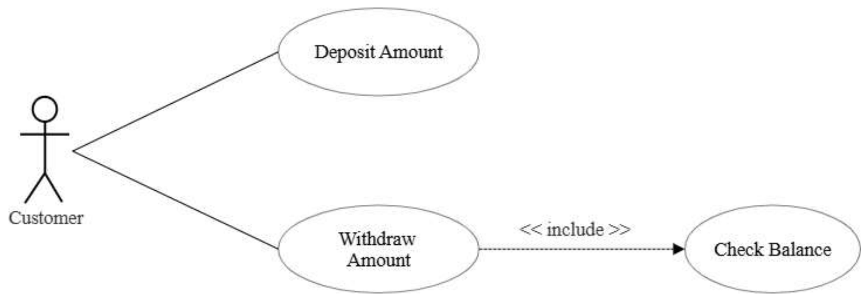
2. Include Relationship

The Include Relationship indicates that a use case includes the functionality of another use case. It is denoted by a dashed arrow pointing from the including use case to the included use case. This relationship promotes modular and reusable design.

Example: Online Banking System

Use Cases: Withdraw Amount, Check Balance

Include Relationship: The "Withdraw Amount" use case includes the functionality of "Check Balance." Therefore, Withdraw Amount includes the action of Check Balance.



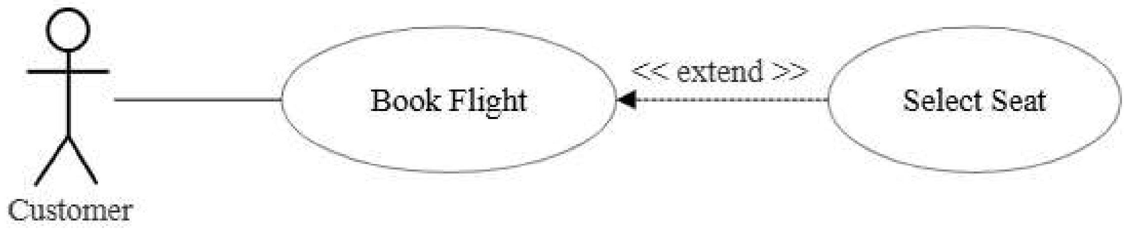
3. Extend Relationship

The Extend Relationship illustrates that a use case can be extended by another use case under specific conditions. It is represented by a dashed arrow with the keyword "extend." This relationship is useful for handling optional or exceptional behavior.

Example: Flight Booking System

Use Cases: Book Flight, Select Seat

Extend Relationship: The "Select Seat" use case may extend the "Book Flight" use case when the user wants to choose a specific seat, but it is an optional step.



.

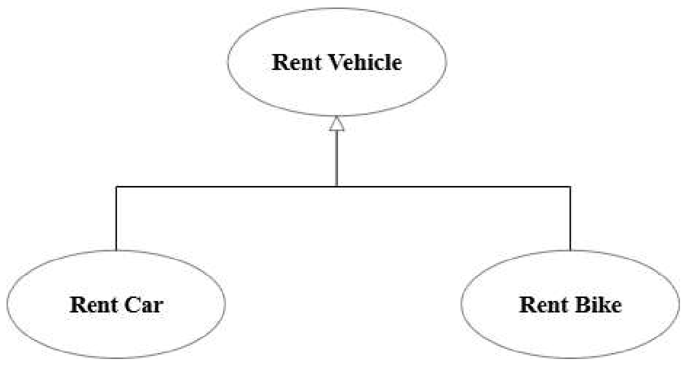
4. Generalization Relationship

The Generalization Relationship establishes an "is-a" connection between two use cases, indicating that one use case is a specialized version of another. It is represented by an arrow pointing from the specialized use case to the general use case.

Example: Vehicle Rental System

Use Cases: Rent Car, Rent Bike

Generalization Relationship: Both "Rent Car" and "Rent Bike" are specialized versions of the general use case "Rent Vehicle."



.

How to draw a Use Case diagram in UML?

Below are the main steps to draw use case diagram in UML:

Step 1: Identify Actors: Determine who or what interacts with the system. These are your actors. They can be users, other systems, or external entities.

Step 2: Identify Use Cases: Identify the main functionalities or actions the system must perform. These are your use cases. Each use case should represent a specific piece of functionality.

Step 3: Connect Actors and Use Cases: Draw lines (associations) between actors and the use cases they are involved in. This represents the interactions between actors and the system.

Step 4: Add System Boundary: Draw a box around the actors and use cases to represent the system boundary. This defines the scope of your system.

Step 5: Define Relationships: If certain use cases are related or if one use case is an extension of another, you can indicate these relationships with appropriate notations.

Step 6: Review and Refine: Step back and review your diagram. Ensure that it accurately represents the interactions and relationships in your system. Refine as needed.

Step 7: Validate: Share your use case diagram with stakeholders and gather feedback. Ensure that it aligns with their understanding of the system's functionality.

Use Case Diagram example(Online Shopping System)

Let's understand how to draw a Use Case diagram with the help of an Online Shopping System:

Actors:

Customer

Admin

Use Cases:

Browse Products

Add to Cart

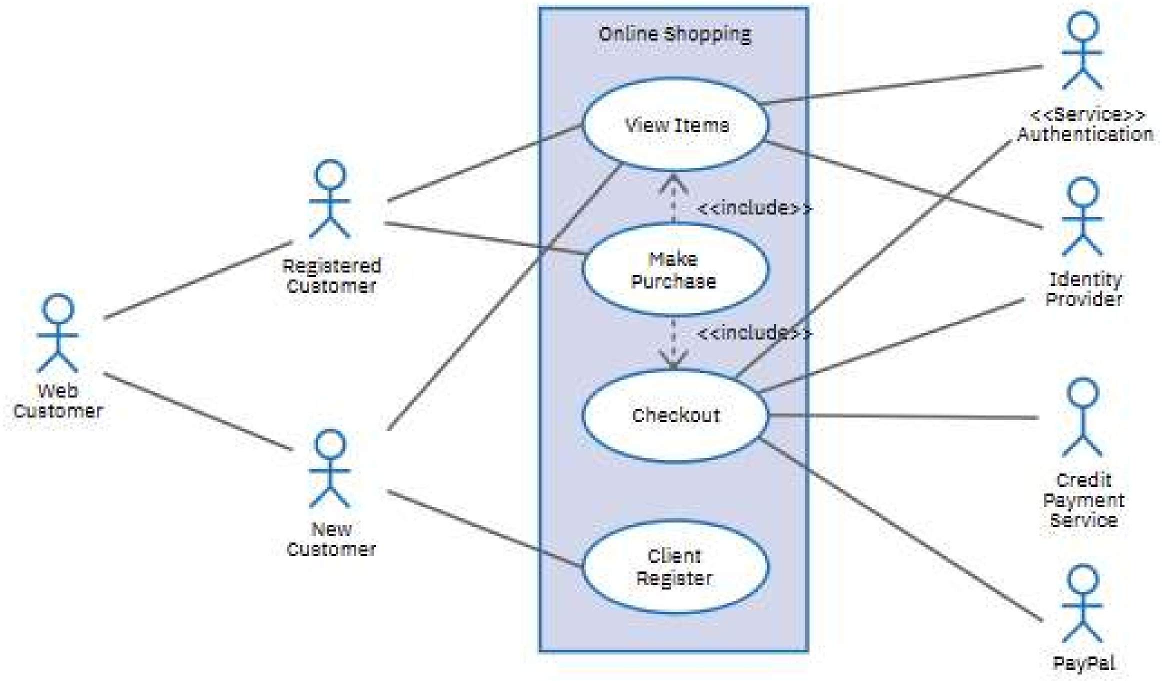
Checkout

Manage Inventory (Admin)

Relations:

The Customer can browse products, add to the cart, and complete the checkout.

The Admin can manage the inventory.



ACTIVITY DIAGRAM:

Activity diagrams are an essential part of the Unified Modeling Language (UML) that help visualize workflows, processes, or activities within a system. They depict how different actions are connected and how a system moves from one state to another. By offering a clear picture of both simple and complex workflows, activity diagrams make it easier for developers and stakeholders to understand how various elements interact in a system.

Activity diagrams show the steps involved in how a system works, helping us understand the flow of control. They display the order in which activities happen and whether they occur one after the other (sequential) or at the same time (concurrent). These diagrams help explain what triggers certain actions or events in a system.

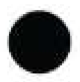
- An activity diagram starts from an initial point and ends at a final point, showing different decision paths along the way.
- They are often used in business and process modeling to show how a system behaves over time.

Activity diagrams may stand alone to visualize, specify, construct, and document the dynamics of a society of objects, or they may be used to model the flow of control of an operation. Whereas interaction diagrams emphasize the flow of control from object to object, activity diagrams emphasize the flow of control from activity to activity.

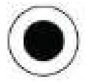
Notations of Activity Diagrams

1. Control Nodes (Flow Control)

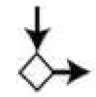
- **Initial Node (Start):** The starting state before an activity takes place is depicted using the initial state. It is represented by a Solid black circle



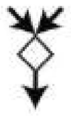
- **Activity Final Node:** This shows the end of the flow in the activity diagram. It is represented as a solid circle with a hollow circle.



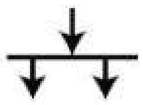
- **Flow Final Node:** A point in an Activity diagram where a flow splits into several mutually exclusive guarded flows. It has one incoming transition and two outgoing transitions. Circle with an 'X', ends a single flow.
- **Decision Node:** Diamond, splits flow based on conditions (guards).


Decision node

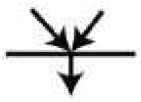
- Merge Node: Diamond, combines alternate flows.


Merge

- **Fork Node:** We use a synchronization bar to specify the forking and joining of parallel flows of control. A synchronization bar is a thick horizontal or vertical line. Thick horizontal bar, splits flow into parallel activities. A Fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control.


Fork

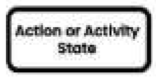
- **Join Node:** A Join may have two or more incoming transitions and one outgoing transition. Thick horizontal bar, merges parallel activities.


Join

2. Action/Activity Nodes (Steps/Behaviors)

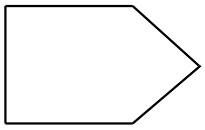
- **Action**: An activity represents execution of an action on objects or by objects. We represent an activity using a rectangle with rounded corners. Basically any action or event that takes place is represented using an activity. Rounded rectangle, a single executable step (e.g., "Validate User").

- **Activity**: Larger rounded rectangle, a complex behavior or sub-process.



Activity State

- **Send Signal:** Convex pentagon, sends a signal.



3. Object Nodes (Data & Objects)

- **Object** Node: Rectangle, represents data or objects.
- **Pin**: Small rectangle on action edges, input/output for data.
- **Data Store:** Object node with «datastore» keyword, persistent data.

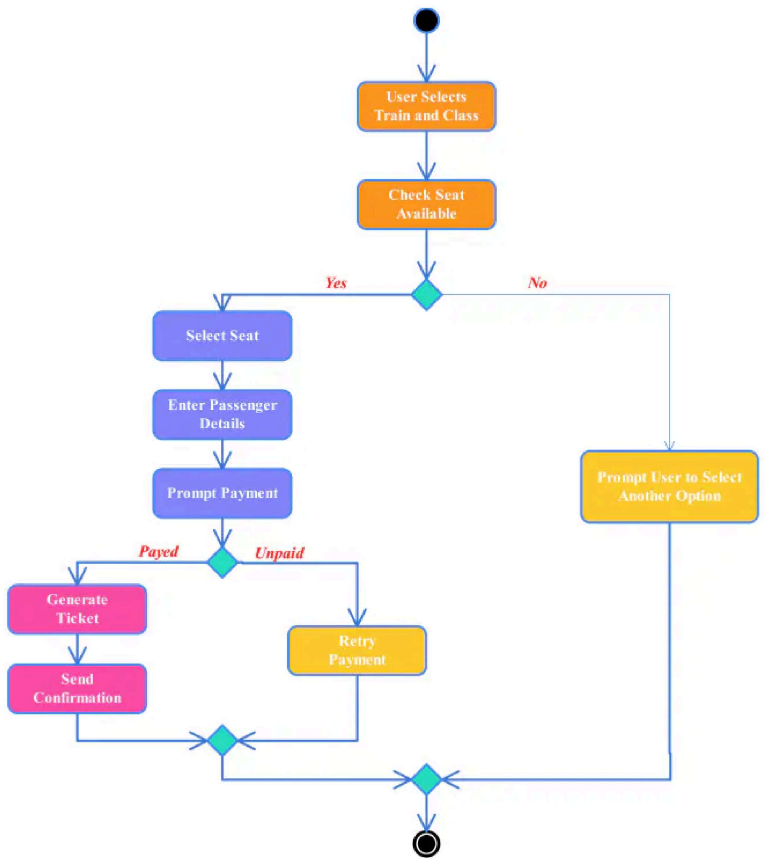
4. Partitions (Swimlanes) (Responsibilities)

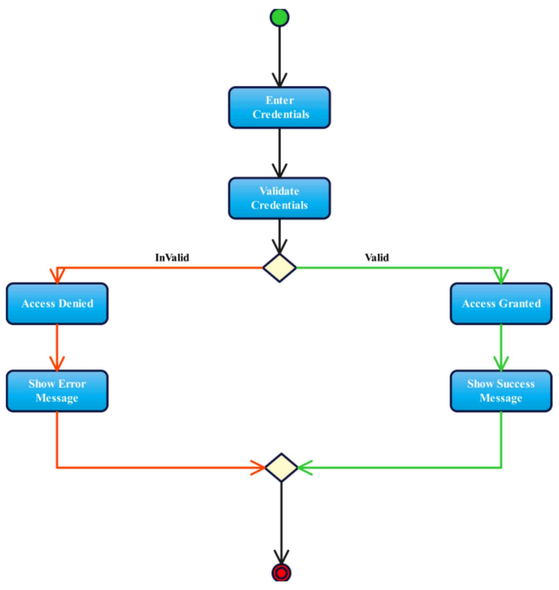
- **Partition/Swimlane:** Vertical or horizontal lanes, group actions by participant/department.

5. Flow Edges (Connections)

- **Control Flow:** Solid arrow, shows sequence of actions.
- **Object Flow:** Dashed arrow, shows data movement between actions.
- **Interrupting Edge:** Lightning bolt, interrupts an action.

Activity diagram for Booking ticket:





SEQUENCE DIAGRAM:

A Sequence Diagram is a key component of Unified Modeling Language (UML) used to visualize the interaction between objects in a sequential order. It focuses on how objects communicate with each other over time, making it an essential tool for modeling dynamic behavior in a system. Sequence diagrams illustrate object interactions, message flows, and the sequence of operations, making them valuable for understanding use cases, designing system architecture, and documenting complex processes.

Uses of sequence diagram:

Sequence diagrams are used because they offer a clear and detailed visualization of the interactions between objects or components in a system, focusing on the order and timing of these interactions. Here are some key reasons for using sequence diagrams:

- **Visualizing Dynamic Behavior**: Sequence diagrams depict how objects or systems interact with each other in a sequential manner, making it easier to understand dynamic processes and workflows.
- **Clear Communication**: They provide an intuitive way to convey system behavior, helping teams understand complex interactions without diving into code.
- **Use Case Analysis**: Sequence diagrams are useful for analyzing and representing use cases, making it clear how specific processes are executed within a system.
- **Designing System Architecture**: They assist in defining how various components or services in a system communicate, which is essential for designing complex, distributed systems or service-oriented architectures.
- **Documenting System Behavior**: Sequence diagrams provide an effective way to document how different parts of a system work together, which can be useful for both developers and maintenance teams.
- **Debugging and Troubleshooting**: By modeling the sequence of interactions, they help identify potential bottlenecks, inefficiencies, or errors in system processes.

Sequence Diagram Notations

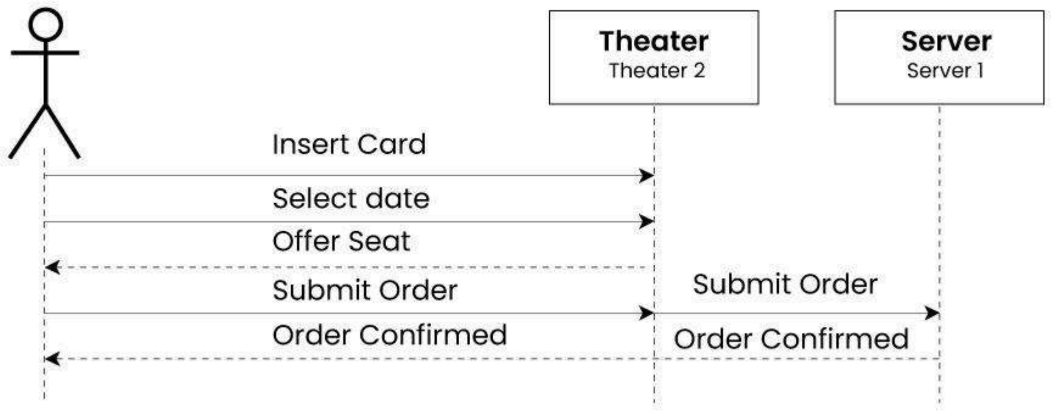
1. Actors

An actor in a UML diagram represents a type of role where it interacts with the system and its objects. It is important to note here that an actor is always outside the scope of the system we aim to model using the UML diagram.

We use actors to depict various roles including human users and other external subjects. We represent an actor in a UML diagram using a stick person notation. We can have multiple actors in a sequence diagram.

For example:

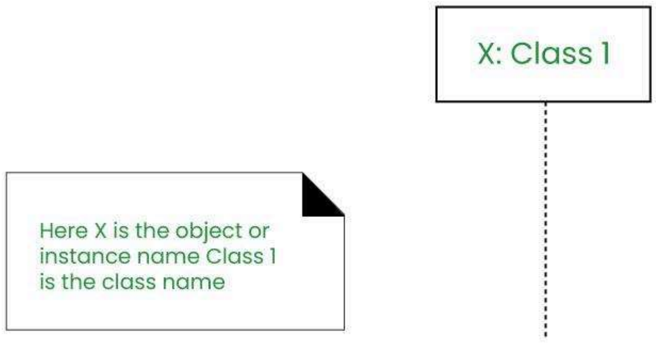
Here the user in seat reservation system is shown as an actor where it exists outside the system and is not a part of the system.



2. Lifelines

A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a sequence diagram is represented by a lifeline. Lifeline elements are located at the top in a sequence diagram. The standard in UML for naming a lifeline follows the following format:

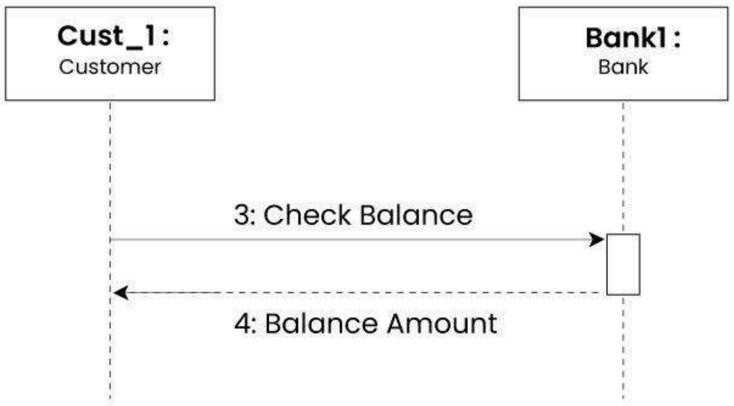
Instance Name : Class Name


X: Class 1
Here X is the object or instance name Class 1 is the class name

We display a lifeline in a rectangle called head with its name and type. The head is located on top of a vertical dashed line (referred to as the stem) as shown above.

- If we want to model an unnamed instance, we follow the same pattern except now the portion of lifeline's name is left blank.
- **Difference between a lifeline and an actor:** A lifeline always portrays an object internal to the system whereas actors are used to depict objects external to the system.

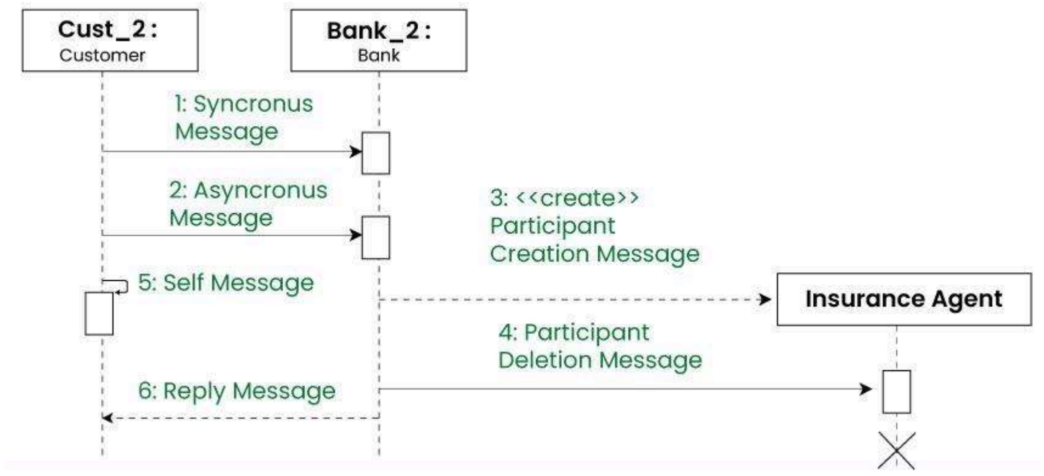
The following is an example of a sequence diagram:



3. Messages

Communication between objects is depicted using messages. The messages appear in a sequential order on the lifeline.

- We represent messages using arrows.
- Lifelines and messages form the core of a sequence diagram.

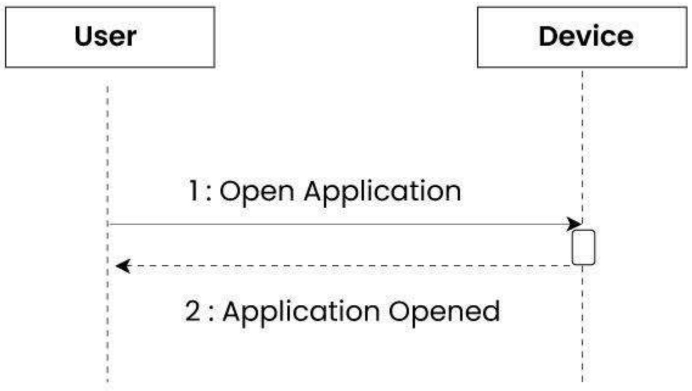


Messages can be broadly classified into the following categories:

1. Synchronous messages

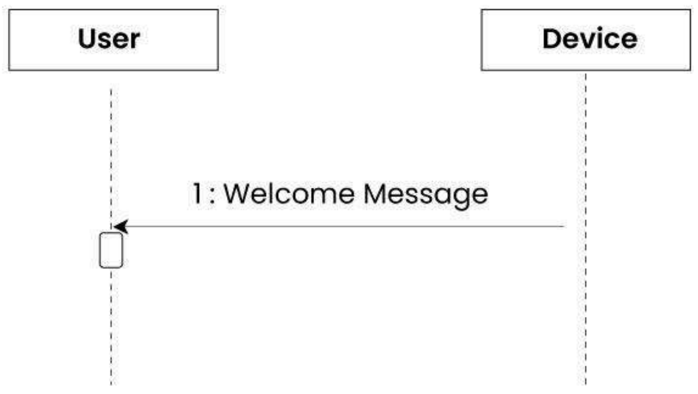
A synchronous message waits for a reply before the interaction can move forward. The sender waits until the receiver has completed the processing of the message. The caller continues only when it knows that the receiver has processed the previous message i.e. it receives a reply message.

- A large number of calls in object oriented programming are synchronous.
- We use a **solid arrow head** to represent a synchronous message.



2. Asynchronous Messages

An asynchronous message does not wait for a reply from the receiver. The interaction moves forward irrespective of the receiver processing the previous message or not. We use a lined arrow head to represent an asynchronous message.

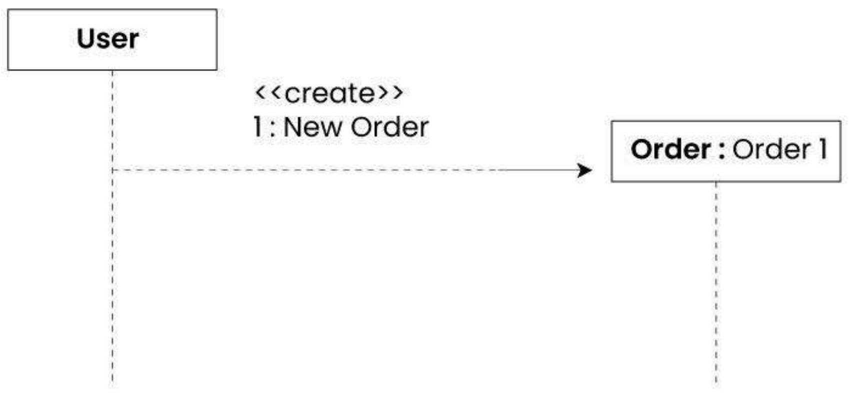


4. Create message

We use a Create message to instantiate a new object in the sequence diagram. There are situations when a particular message call requires the creation of an object. It is represented with a dotted arrow and create word labelled on it to specify that it is the create Message symbol.

For example:

The creation of a new order on a e-commerce website would require a new object of Order class to be created.

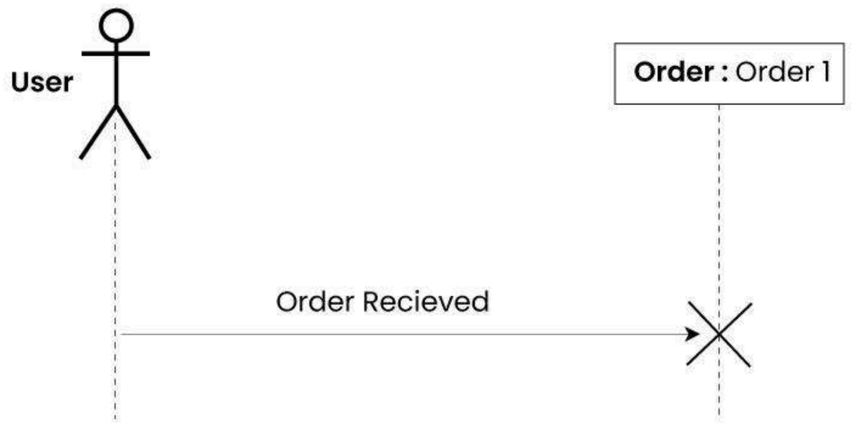


5. Delete Message

We use a Delete Message to delete an object. When an object is deallocated memory or is destroyed within the system we use the Delete Message symbol. It destroys the occurrence of the object in the system.It is represented by an arrow terminating with a x.

For example:

In the scenario below when the order is received by the user, the object of order class can be destroyed.

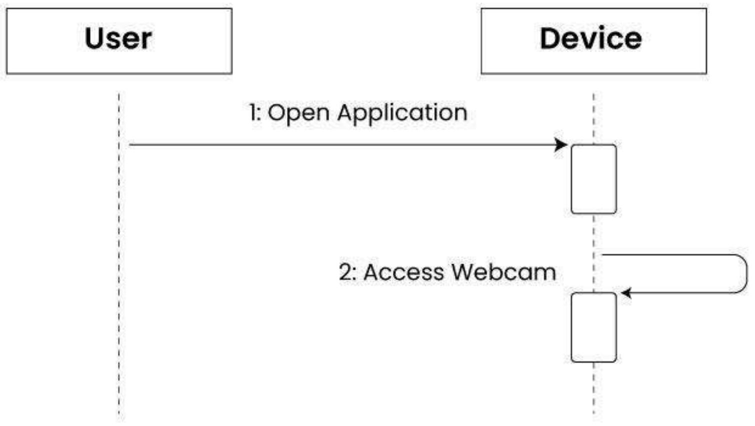


6. Self Message

Certain scenarios might arise where the object needs to send a message to itself. Such messages are called Self Messages and are represented with a **U shaped arrow**.

For example:

Consider a scenario where the device wants to access its webcam. Such a scenario is represented using a self message.

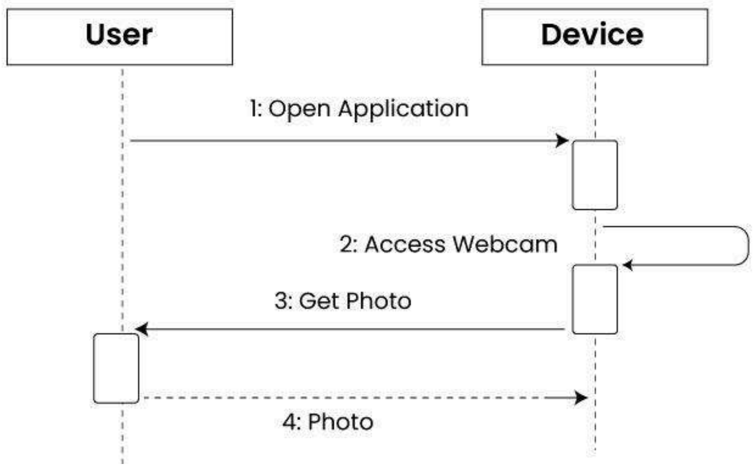


7. Reply Message

Reply messages are used to show the message being sent from the receiver to the sender. We represent a return/reply message using an **open arrow head with a dotted line**. The interaction moves forward only when a reply message is sent by the receiver.

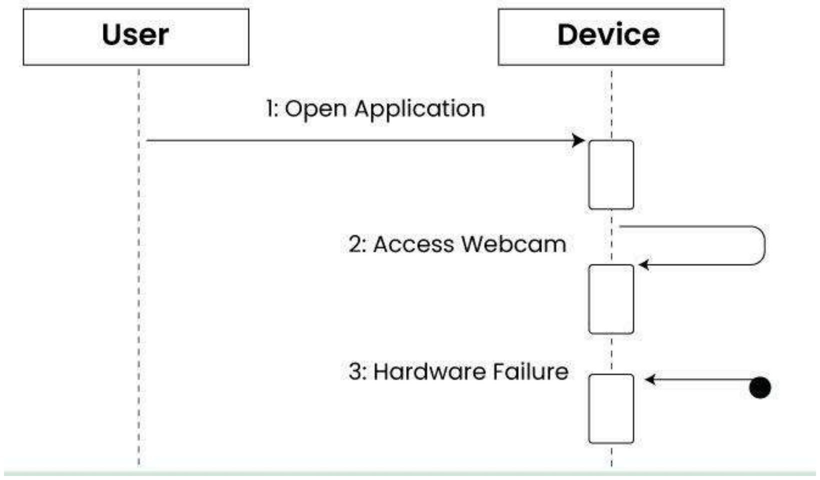
For example:

Consider the scenario where the device requests a photo from the user. Here the message which shows the photo being sent is a reply message.



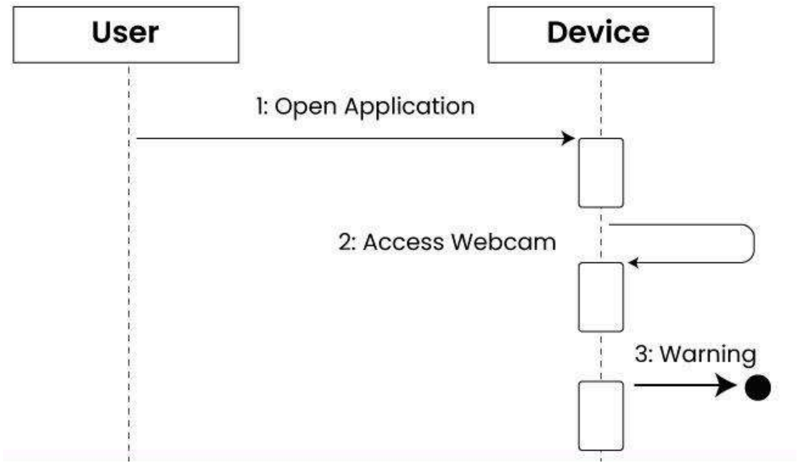
8. Found Message

A Found message is used to represent a scenario where an unknown source sends the message. It is represented using an **arrow directed towards a lifeline** from an end point. It can be due to multiple reasons and we are not certain as to what caused the hardware failure.



9. Lost Message

A Lost message is used to represent a scenario where the recipient is not known to the system. It is represented using an arrow directed towards an end point from a lifeline. The warning might be generated for the user or other software/object that the lifeline is interacting with. Since the destination is not known before hand, we use the Lost Message symbol.

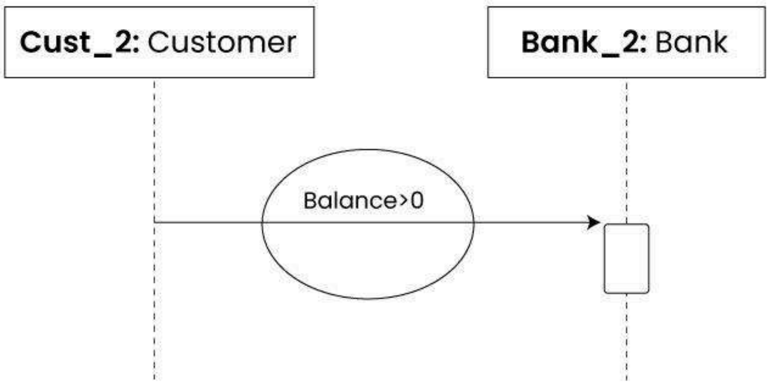


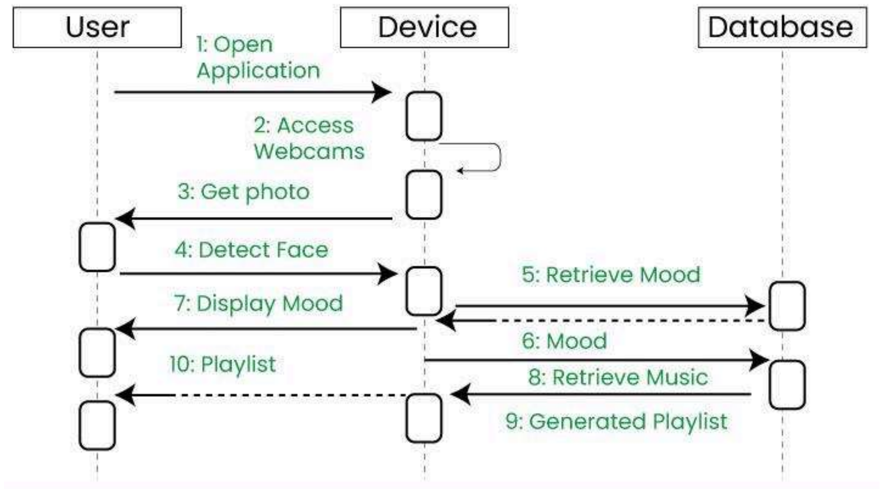
10. Guards

To model conditions we use guards in UML. They are used when we need to restrict the flow of messages on the pretext of a condition being met. Guards play an important role in letting software developers know the constraints attached to a system or a particular process.

For example:

In order to be able to withdraw cash, having a balance greater than zero is a condition that must be met as shown below.

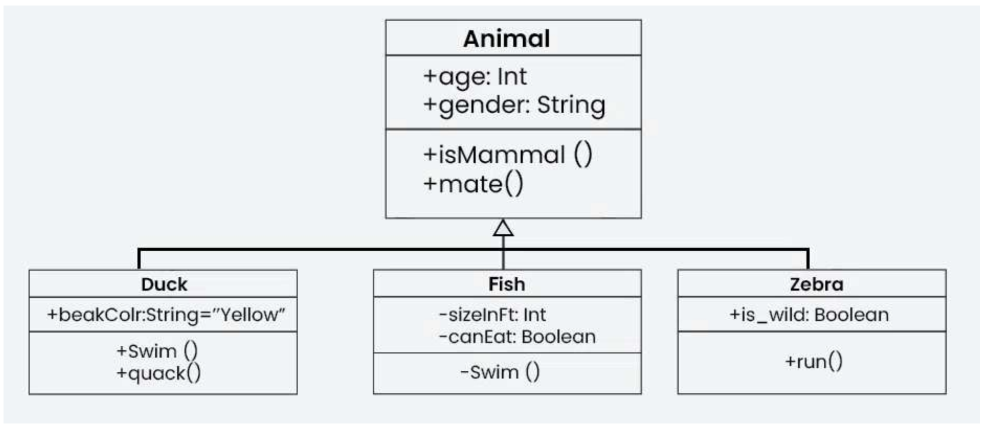




CLASS DIAGRAM

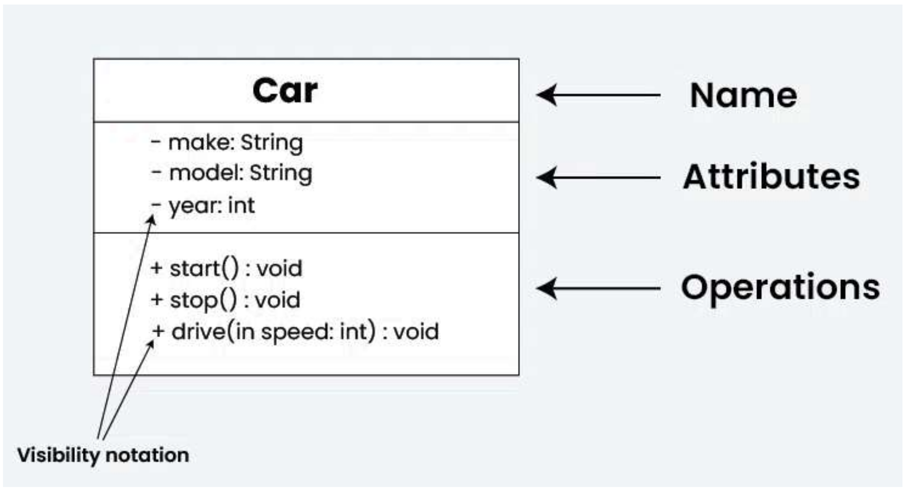
A UML class diagram visually represents the structure of a system by showing its classes, attributes, methods, and the relationships between them.

1. Helps everyone involved in a project—like developers and designers—understand how the system is organized and how its components interact.
2. Helps to communicate and document the structure of the software.



Class Notation

Classes are depicted as boxes, each containing three compartments for the class name, attributes, and methods.



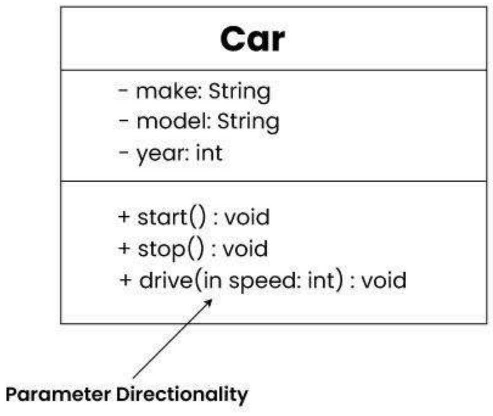
1. **Class Name:** The name of the class is typically written in the top compartment of the class box and is centered and bold.
2. **Attributes:** Attributes, also known as properties or fields, represent the data members of the class. They are listed in the second compartment of the class box and often include the visibility (e.g., public, private) and the data type of each attribute.
3. **Methods:** Methods, also known as functions or operations, represent the behavior or functionality of the class. They are listed in the third compartment of the class box

and include the visibility (e.g., public, private), return type, and parameters of each method.

4. **Visibility Notation:** Visibility notations indicate the access level of attributes and methods. Common visibility notations include:
   - + for public (visible to all classes)
   - - for private (visible only within the class)
   - # for protected (visible to subclasses)
   - ~ for package or default visibility (visible to classes in the same package)

Parameter Directionality

- In class diagrams, parameter directionality refers to the indication of the flow of information between classes through method parameters.
- It helps to specify whether a parameter is an input, an output, or both. This information is crucial for understanding how data is passed between objects during method calls.



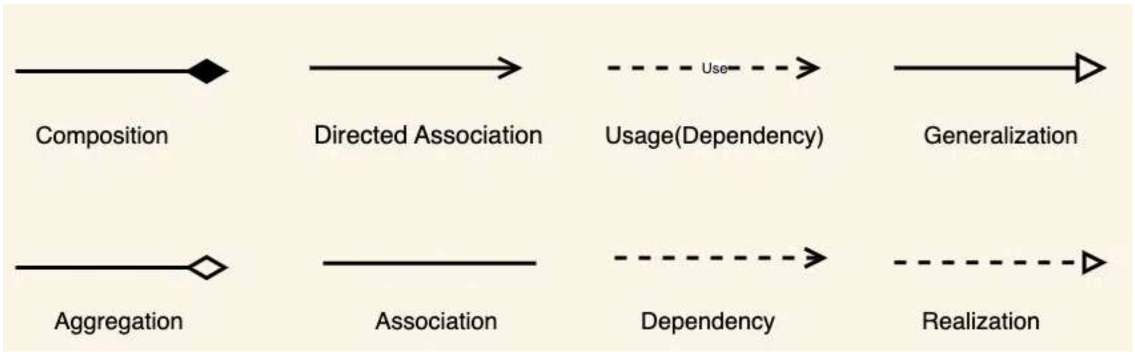
There are three main parameter directionality notations used in class diagrams:

- **In (Input):**
  - An input parameter is a parameter passed from the calling object (client) to the called object (server) during a method invocation.
  - It is represented by an arrow pointing towards the receiving class (the class that owns the method).
- **Out (Output):**

    - An output parameter is a parameter passed from the called object (server) back to the calling object (client) after the method execution.
    - It is represented by an arrow pointing away from the receiving class.
- **InOut (Input and Output):**
    - An InOut parameter serves as both input and output. It carries information from the calling object to the called object and vice versa.
    - It is represented by an arrow pointing towards and away from the receiving class.

Relationships between classes

In class diagrams, relationships between classes describe how classes are connected or interact with each other within a system. Here are some common types of relationships in class diagrams:



1. Association

An association represents a bi-directional relationship between two classes. It indicates that instances of one class are connected to instances of another class. Associations are typically depicted as a solid line connecting the classes, with optional arrows indicating the direction of the relationship.

2. Directed Association

A directed association in a UML class diagram represents a relationship between two classes where the association has a direction, indicating that one class is associated with another in a specific way.

3. Aggregation

Aggregation is a specialized form of association that represents a "whole-part" relationship. It denotes a stronger relationship where one class (the whole) contains or is composed of another class (the part). Aggregation is represented by a diamond shape on the side of the whole class. In this kind of relationship, the child class can exist independently of its parent class.

4. Composition

Composition is a stronger form of aggregation, indicating a more significant ownership or dependency relationship. In composition, the part class cannot exist independently of the whole class. Composition is represented by a filled diamond shape on the side of the whole class.

5. Generalization(Inheritance)

Inheritance represents an "is-a" relationship between classes, where one class (the subclass or child) inherits the properties and behaviors of another class (the superclass or parent). Inheritance is depicted by a solid line with a closed, hollow arrowhead pointing from the subclass to the superclass.

6. Realization (Interface Implementation)

Realization indicates that a class implements the features of an interface. It is often used in cases where a class realizes the operations defined by an interface. Realization is depicted by a dashed line with an open arrowhead pointing from the implementing class to the interface.

7. Dependency Relationship

A dependency exists between two classes when one class relies on another, but the relationship is not as strong as association or inheritance. It represents a more loosely coupled connection between classes.

8. Usage(Dependency) Relationship

A usage dependency relationship in a UML class diagram indicates that one class (the client) utilizes or depends on another class (the supplier) to perform certain tasks or access certain functionality. The client class relies on the services provided by the supplier class but does not own or create instances of it.

- In UML class diagrams, usage dependencies are typically represented by a dashed arrowed line pointing from the client class to the supplier class.
- The arrow indicates the direction of the dependency, showing that the client class depends on the services provided by the supplier class.

Purpose of Class Diagrams

The main purpose of using class diagrams is:

- This is the only UML that can appropriately depict various aspects of the OOPs concept.
- Proper design and analysis of applications can be faster and efficient.
- It is the base for deployment and component diagrams.
- It incorporates forward and reverse engineering.

Benefits of Class Diagrams

Below are the benefits of class diagrams:

- Class diagrams represent the system's classes, attributes, methods, and relationships, providing a clear view of its architecture.
- They show various relationships between classes, such as associations and inheritance, helping stakeholders understand component connectivity.
- Class diagrams serve as a visual tool for communication among team members and stakeholders, bridging gaps between technical and non-technical audiences.
- They guide developers in coding by illustrating the design, ensuring consistency between the design and actual implementation.
- Many development tools allow for code generation from class diagrams, reducing manual errors and saving time.
