UML

Unified Modeling Language (UML) Diagrams

- Unified Modeling Language (UML) is a general-purpose modeling language.
- The main aim of UML is to define a standard way to visualize the way a system has been designed.
- Unified Modeling Language (UML) is a standardized visual modeling language that is a versatile, flexible, and user-friendly method for visualizing a system’s design. Software system artifacts can be specified, visualized, built, and documented with the use of UML.

Why do we need UML?

- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.
- Businessmen do not understand code. So UML becomes essential to communicate with non-programmers about essential requirements, functionalities, and processes of the system.
- A lot of time is saved down the line when teams can visualize processes, user interactions, and the static structure of the system.

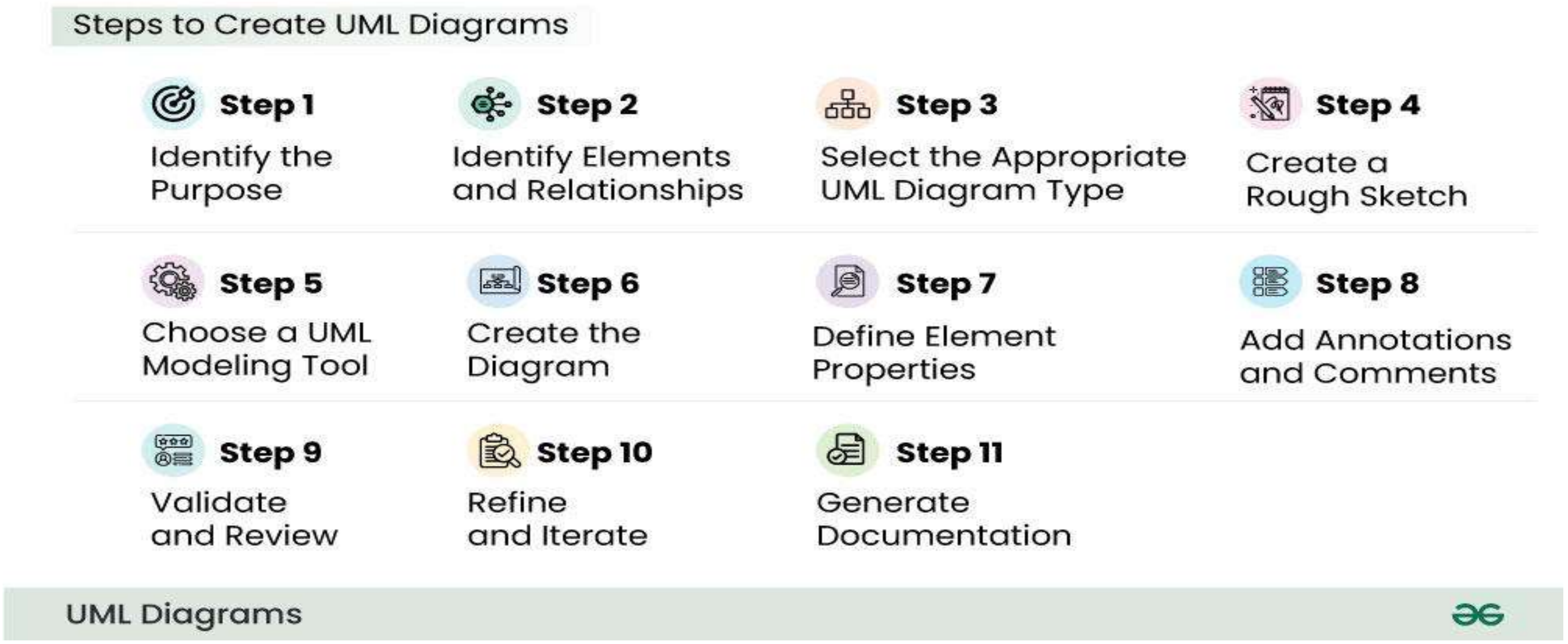
Object-Oriented Concepts Used in UML Diagrams

- **Class**: An object’s structure and behavior are defined by its class, which serves as a blueprint.
- **Objects**: We may divide complex systems into smaller, more manageable components by using objects. Because of its modularity, we can concentrate on easily understood components and develop the system gradually.
- **Inheritance**: Child classes can inherit the characteristics and functions of their parent classes.
- **Abstraction**: The main characteristics of a system or object are highlighted in UML abstraction, while extraneous details are ignored. Stakeholder communication and understanding are improved by this simplification.
- **Encapsulation**: Encapsulation is the process of integrating data and restricting external access in order to maintain the integrity of the data.
- **Polymorphism**: Flexibility in their use is made possible by polymorphism, the capacity of functions or entities to take on multiple forms.

Tools for creating UML Diagrams

- Lucidchart
- Draw.io
- Visual Paradigm
- StarUML
- Microsoft Visio

Steps to create UML Diagrams



Steps to Create UML Diagrams

Step 1
Identify the Purpose

Step 2
Identify Elements and Relationships

Step 3
Select the Appropriate UML Diagram Type

Step 4
Create a Rough Sketch

Step 5
Choose a UML Modeling Tool

Step 6
Create the Diagram

Step 7
Define Element Properties

Step 8
Add Annotations and Comments

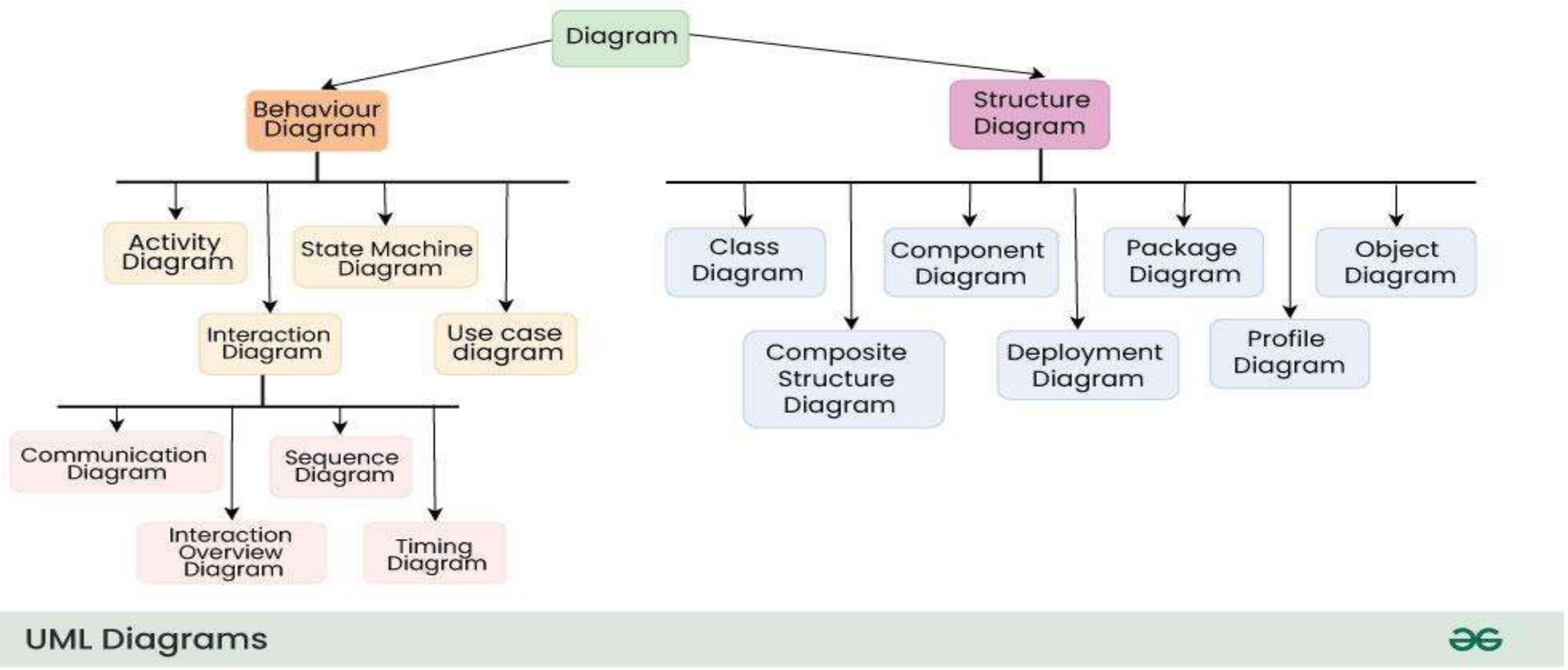
Step 9
Validate and Review

Step 10
Refine and Iterate

Step 11
Generate Documentation

UML Diagrams

Types of UML Diagrams



- Structural UML diagrams are visual representations that depict the static aspects of a system, including its classes, objects, components, and their relationships, providing a clear view of the system’s architecture.

- Behavioral UML diagrams are visual representations that depict the dynamic aspects of a system, illustrating how objects interact and behave over time in response to events.

- Use Case Diagrams
- Class Diagram
- State Transition Diagrams
- Interaction Diagram

Advantages of UML

- **Visualization**: Provides a clear, visual representation of system architecture and behaviors.
- **Standardization**: Universal symbols allow team members to communicate ideas effectively.
- **Scalability**: Suitable for both small-scale and large-scale projects.
- **Better Documentation**: Helps document the system, making it easier to maintain and extend.

UML in Software Development Process

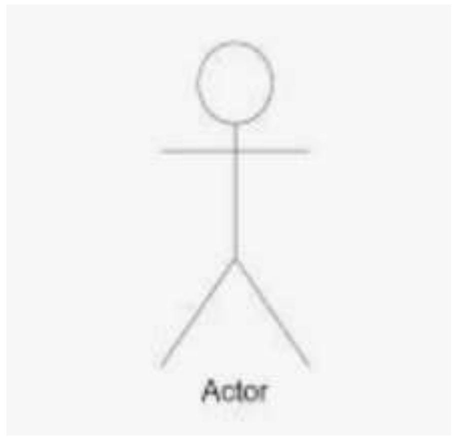
- UML can be used in various stages of software development:
- **Requirements Analysis**: Use case diagrams can identify system functionality.
- **System Design**: Class, component, and deployment diagrams specify system architecture.
- **Implementation**: Sequence, state, and activity diagrams help developers understand process flows.
- **Testing and Maintenance**: Object and state diagrams are useful for checking the correct behavior and structure of the system during testing.

Use Case Diagram

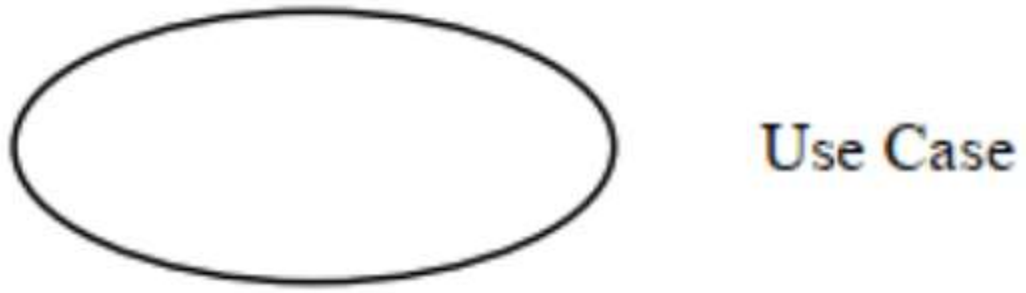
- A **Use Case Diagram** in UML is used to visually represent the functional requirements of a system, showing how users (actors) interact with the system to achieve specific goals (use cases).
- It provides a high-level view of what the system does without detailing how it does it.
- Use Case Diagrams are typically employed during the requirements gathering phase to ensure that all stakeholder expectations are captured.

Key Components of a Use Case Diagram

- **Actors**:
- Represent users or external systems that interact with the system.
- Depicted as stick figures outside the system boundary.
- Can be a human user, another system, or a hardware device.
- **Primary Actor**: The main actor that initiates the interaction.
- **Secondary Actor**: Assists in the use case but doesn't initiate it.



- **Use Cases**:
- Represent functions or features provided by the system.
- Depicted as ovals within the system boundary.
- Describes what the system does in response to an actor's action.



- **System Boundary**:
- Defines the scope of the system.
- Represented as a rectangle surrounding the use cases.

System boundary is represented by a rectangular box.

- **Relationships**:
- **Association**: A line between an actor and a use case showing an interaction.
- **Include**: A dashed arrow with the label «include» showing a use case that is part of another use case. It represents mandatory functionality.
- **Extend**: A dashed arrow with the label «extend» showing an optional extension of a use case. It's used to show additional, conditional functionality.
- **Generalization**: An arrow representing inheritance, used when actors or use cases have a parent-child relationship.

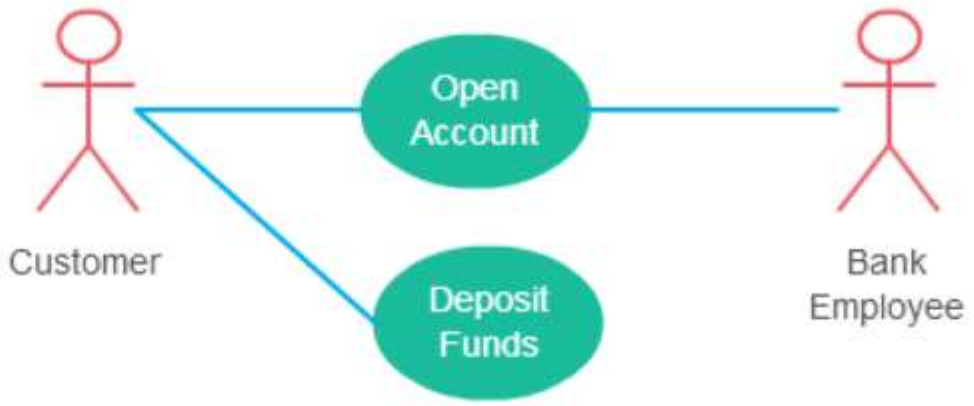
- There can be 5 relationship types in a use case diagram.
- Association between actor and use case
- Generalization of an actor
- Extend between two use cases
- Include between two use cases
- Generalization of a use case

How to Create a Use Case Diagram

1. Identifying Actors
2. Identifying Use Cases
3. Look for Common Functionality to Reuse
4. Is it Possible to Generalize Actors and Use Cases
5. Optional Functions or Additional Functions
6. Validate and Refine the Diagram

Association Between Actor and Use Case

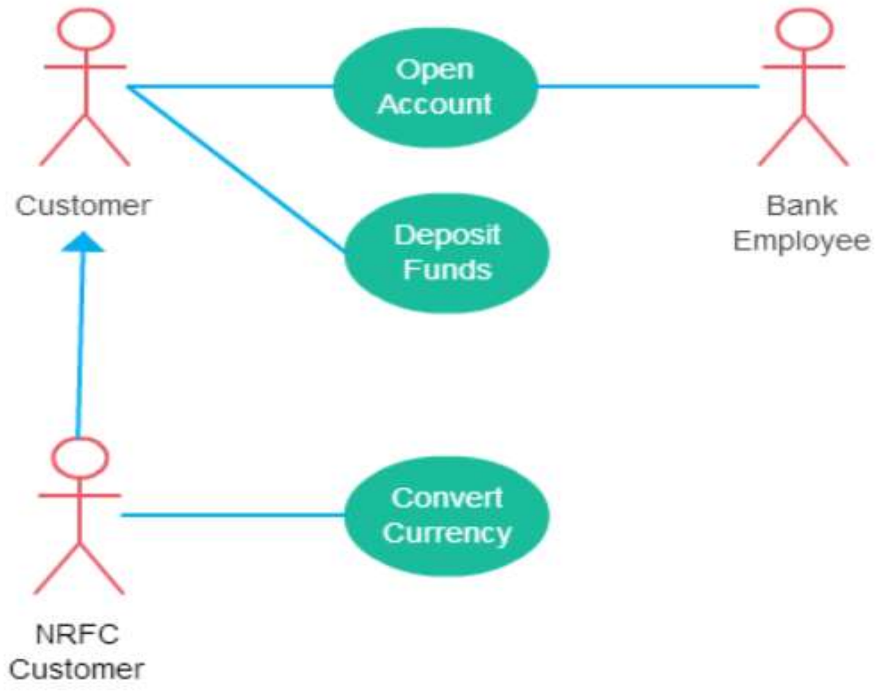
- An actor must be associated with at least one use case.
- An actor can be associated with multiple use cases.
- Multiple actors can be associated with a single use case.



Different ways association relationship appears in use case diagrams

Generalization of an Actor

Generalization of an actor means that one actor can inherit the role of the other actor. The descendant inherits all the use cases of the ancestor. The descendant has one or more use cases that are specific to that role. Let's expand the previous use case diagram to show the generalization of an actor.

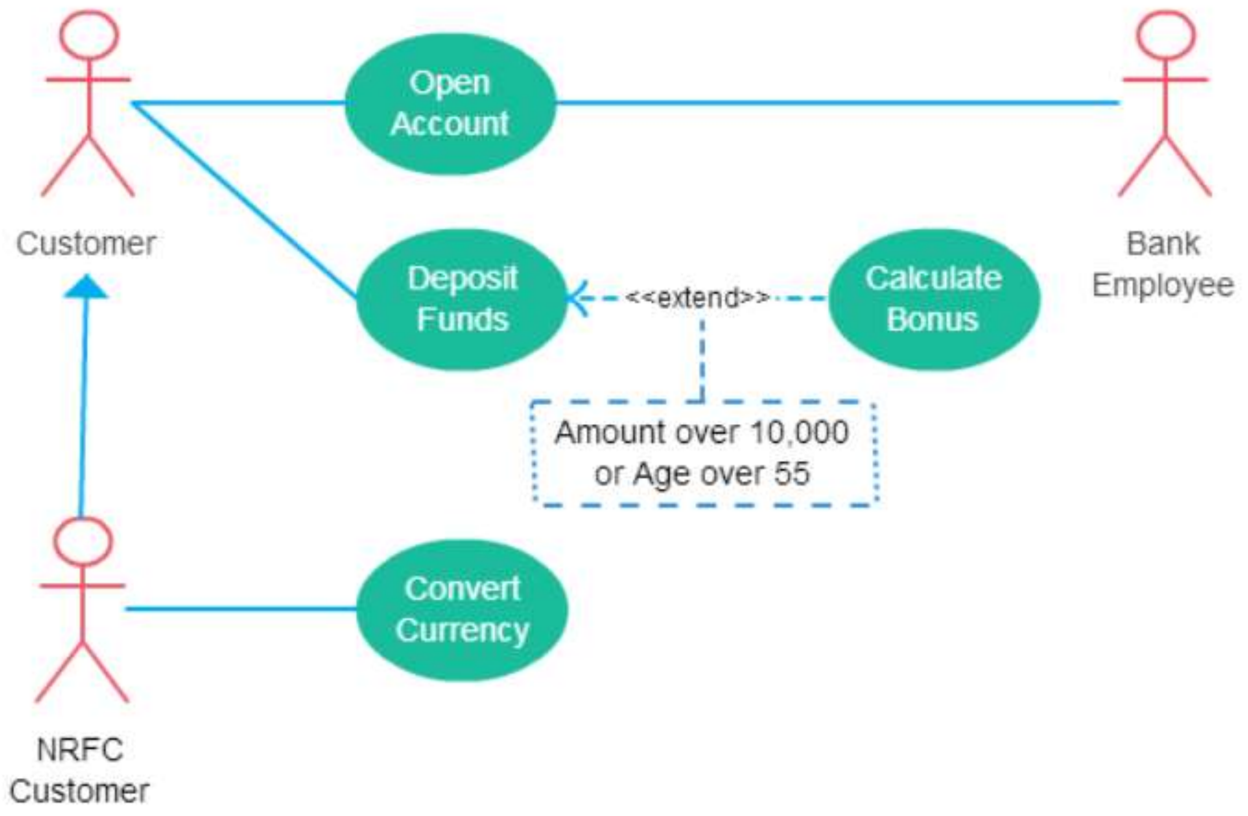


A generalized actor in an use case diagram

Extend Relationship Between Two Use Cases

- As the name implies it extends the base use case and adds more functionality to the system. Here are a few things to consider when using the <<extend>> relationship.

- The extending use case is dependent on the extended (base) use case. In the below diagram the “Calculate Bonus” use case doesn’t make much sense without the “Deposit Funds” use case.
- The extending use case is usually optional and can be triggered conditionally. In the diagram, you can see that the extending use case is triggered only for deposits over 10,000 or when the age is over 55.
- The extended (base) use case must be meaningful on its own. This means it should be independent and must not rely on the behavior of the extending use case.

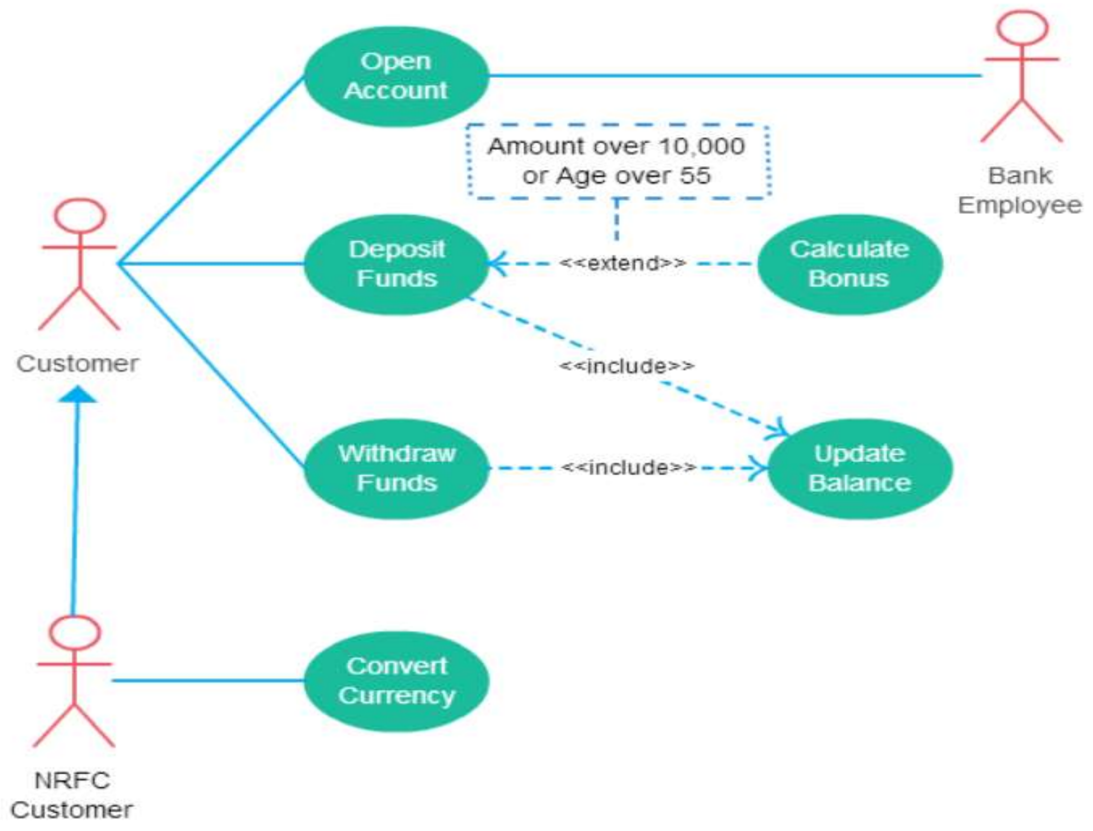


Extend relationship in use case diagrams

Include Relationship Between Two Use Cases

- Include relationship show that the behavior of the included use case is part of the including (base) use case. The main reason for this is to reuse common actions across multiple use cases. In some situations, this is done to simplify complex behaviors. Few things to consider when using the <<include>> relationship.

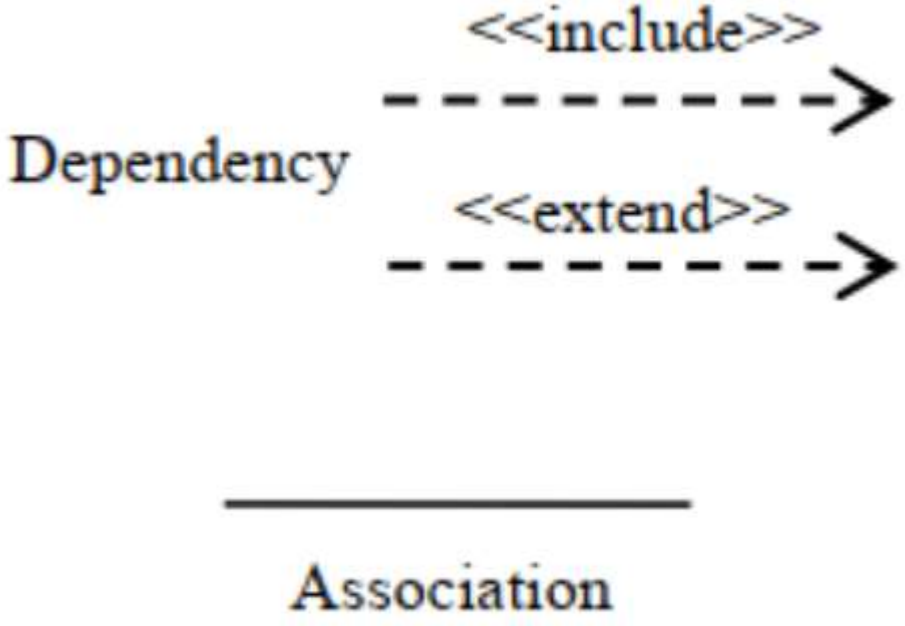
- The base use case is incomplete without the included use case.
- The included use case is mandatory and not optional.



Includes is usually used to model common behavior

Generalization of a Use Case

- This is similar to the generalization of an actor. The behavior of the ancestor is inherited by the descendant. This is used when there is common behavior between two use cases and also specialized behavior specific to each use case.
- For example, in the previous banking example, there might be a use case called “Pay Bills”. This can be generalized to “Pay by Credit Card”, “Pay by Bank Balance” etc.


Dependency
<<include>>
<<extend>>
Association

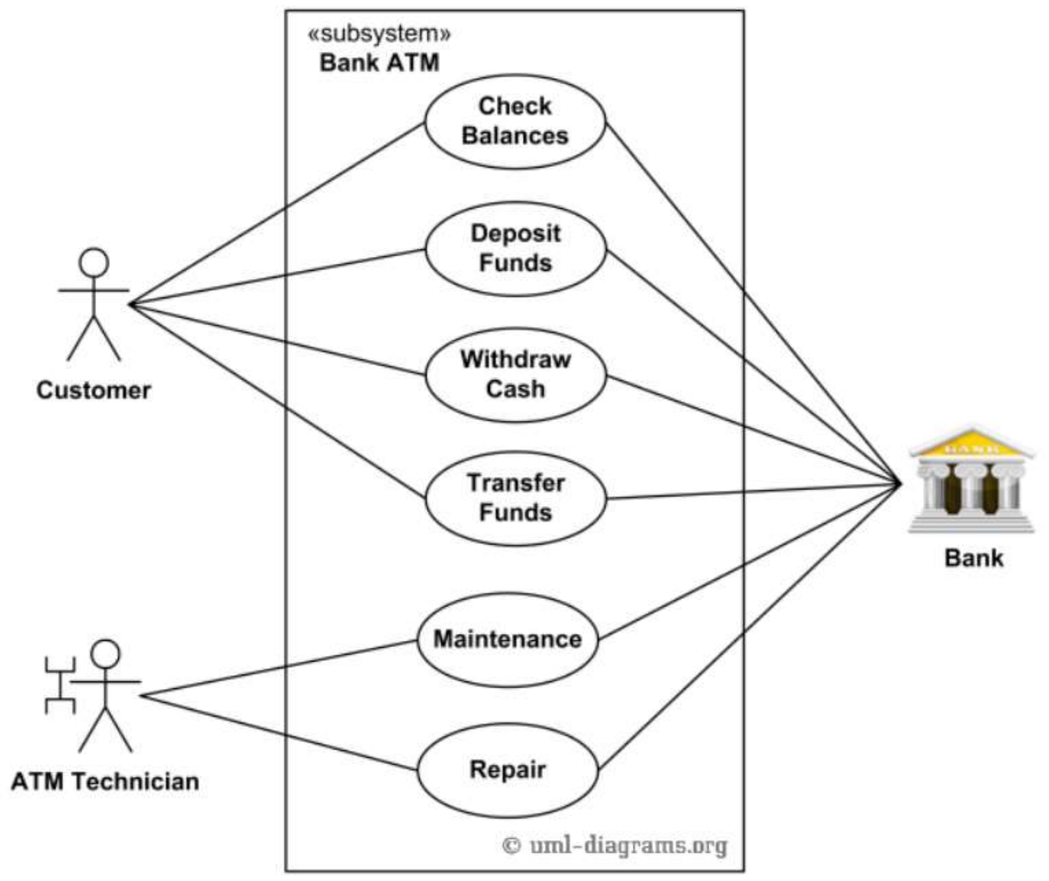
- In UML (Unified Modeling Language), extend and include are relationships used to describe associations between use cases, particularly in a use case diagram.

- Include Relationship
- The include relationship represents a mandatory dependency, where a base use case incorporates the behavior of another use case.
- This relationship is usually created when you have common behavior that multiple use cases need to perform, so you put that shared behavior in an included use case.
- The included use case is always executed as part of the base use case.
- Example:
- In an online shopping system:
- Make Payment could include the Validate Payment Method use case, as payment validation is essential whenever a payment is made.
- Notation:
- In UML, the include relationship is shown with a dashed arrow with the label <<include>> pointing from the base use case to the included use case.

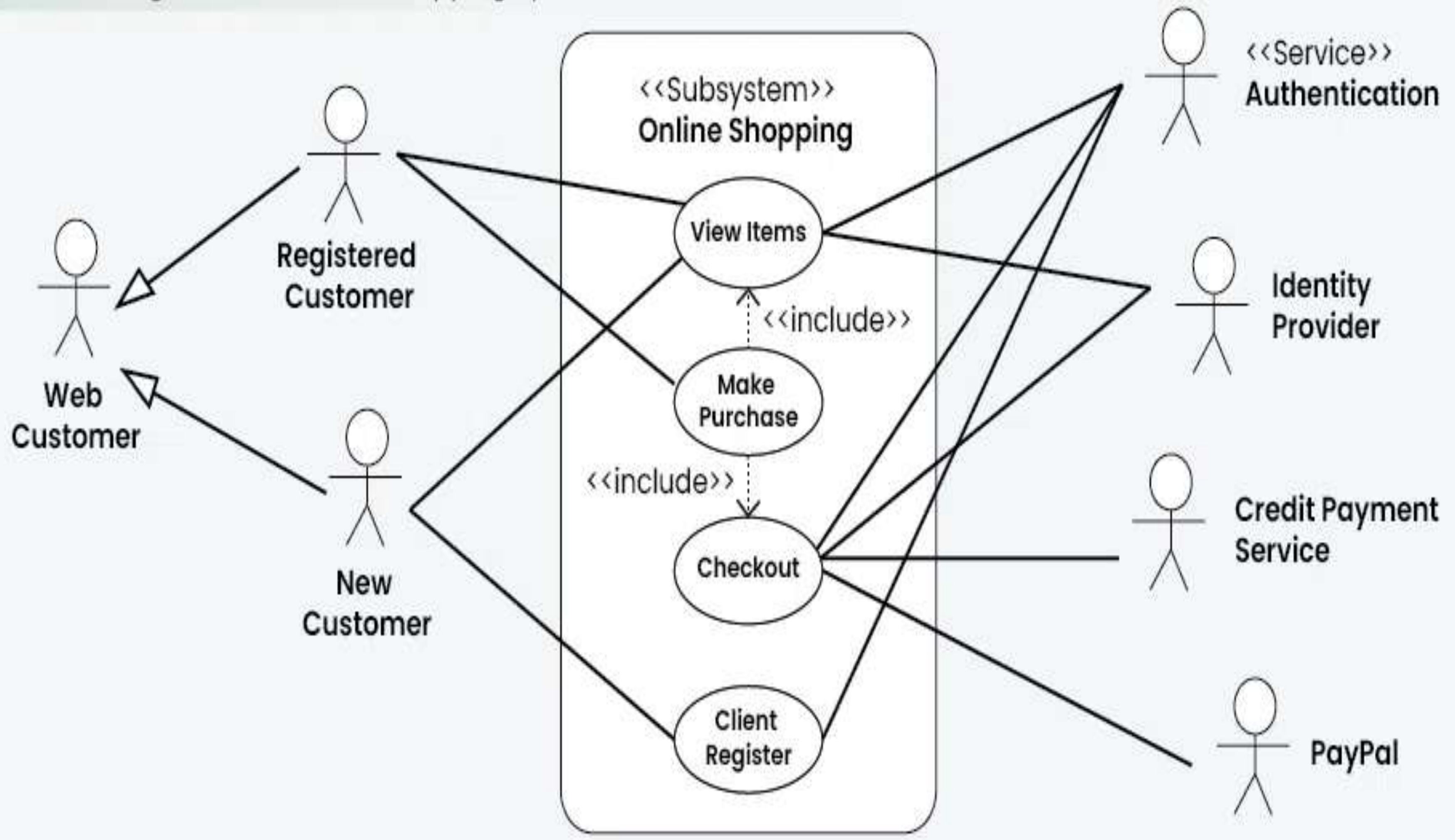
- Extend Relationship
- The extend relationship represents an optional behavior that extends the base use case under certain conditions.
- It defines a situation where a use case is enhanced or modified by another use case, but this additional behavior only happens under specific circumstances.
- The extending use case is not always executed; it only occurs if certain conditions are met.
- Example:
- In an ATM system:
- Withdraw Cash could extend Authenticate User to account for scenarios where additional security checks are required.
- Notation:
- In UML, the extend relationship is shown with a dashed arrow with the label <<extend>> pointing from the extending use case to the base use case.

Aspect	`Include`	`Extend`
Purpose	Mandatory, common behavior inclusion	Optional, conditional behavior
Execution	Always executed in the base use case	Executed only if certain conditions apply
Arrow Direction	From base to included use case	From extending to base use case
Notation	`<<include>>`	`<<extend>>`

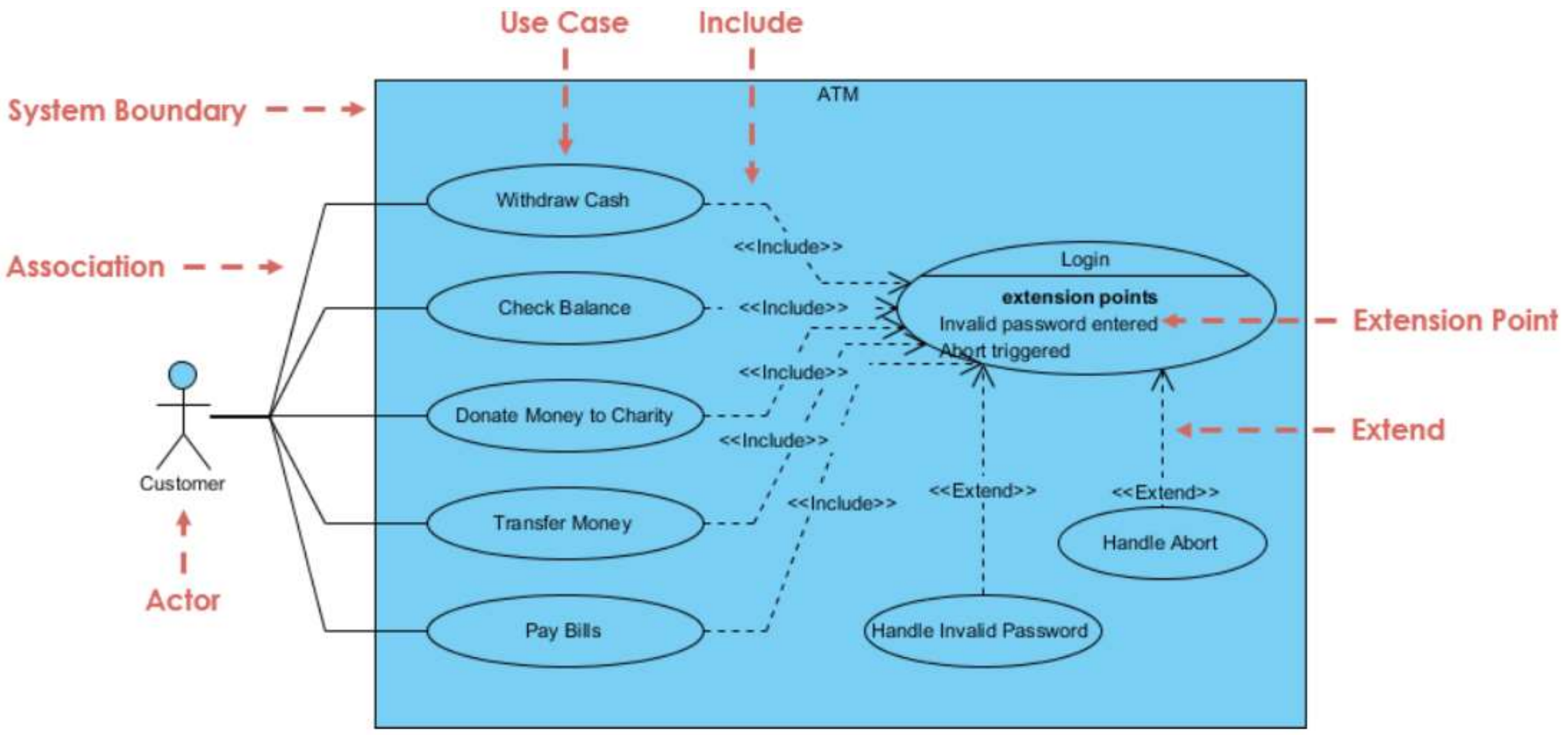
Bank ATM-*UML Use Case Diagram*



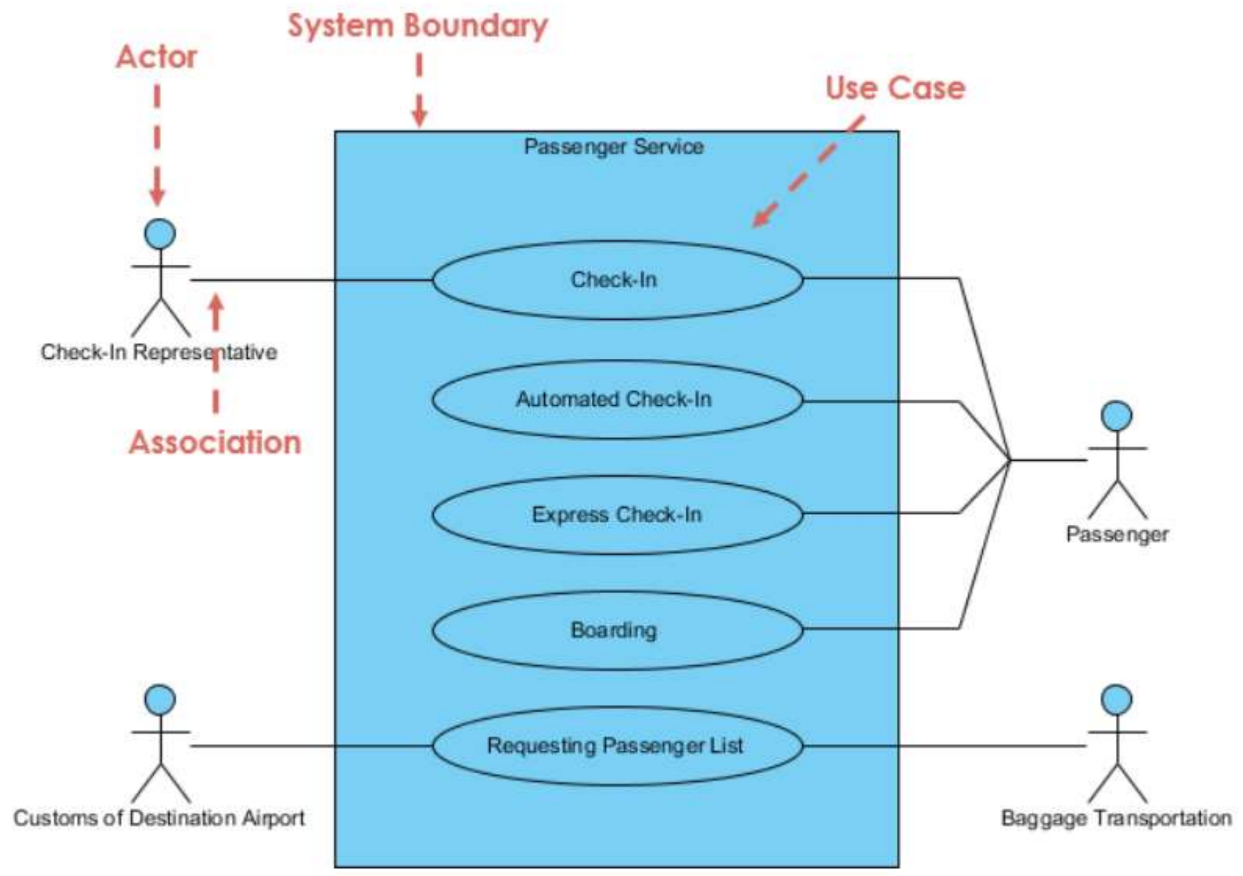
Use Case diagram of an Online Shopping System

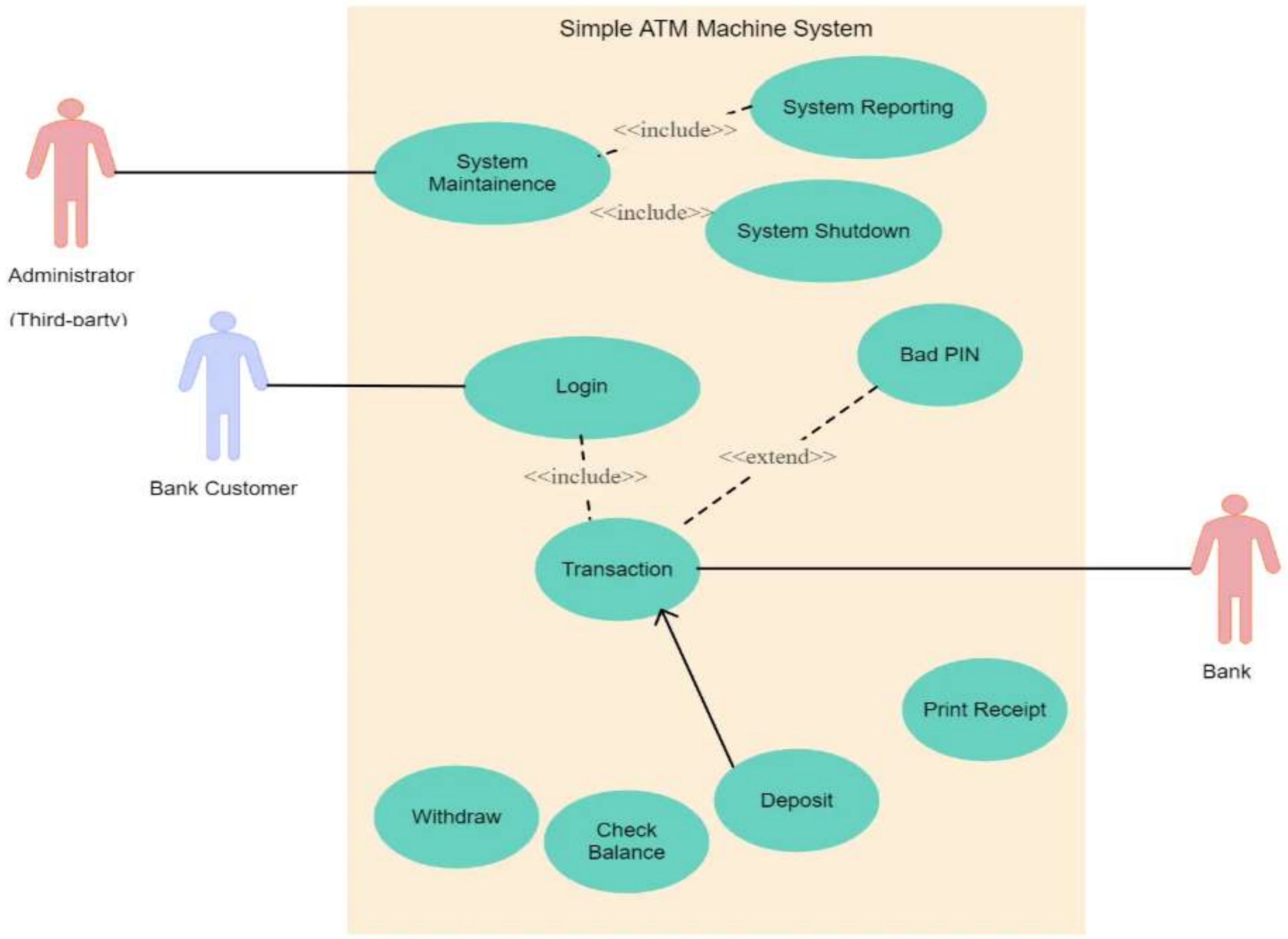


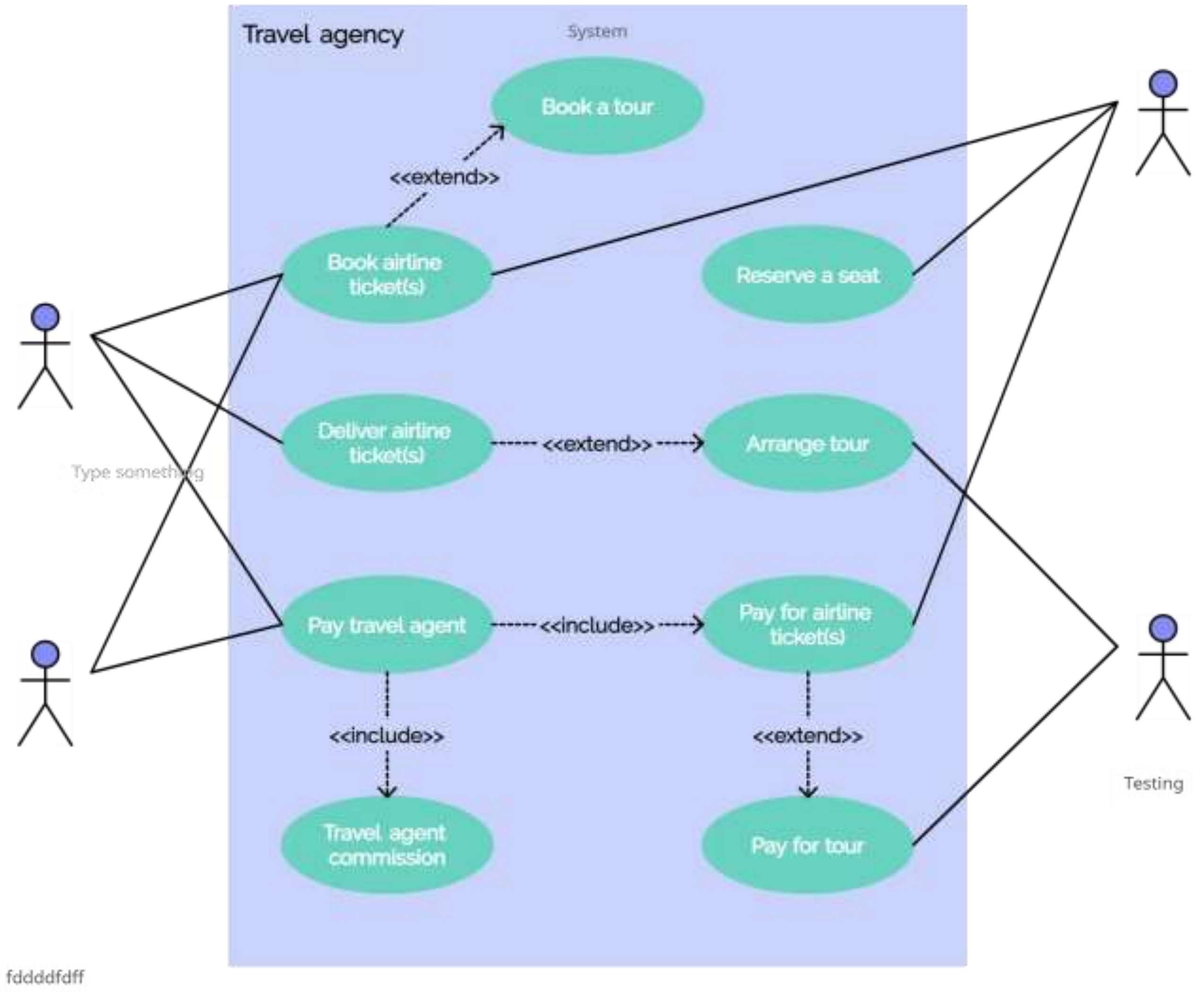
Use Case Diagrams | Unified Modeling Language (UML)

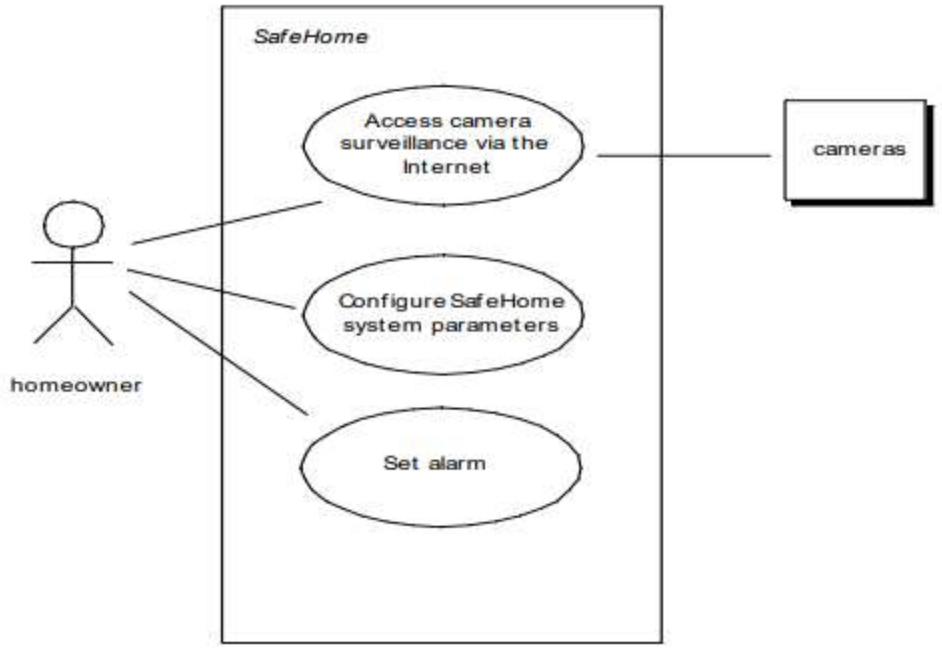

Use Case
Include
System Boundary
ATM
Withdraw Cash
Association
Check Balance
Donate Money to Charity
Transfer Money
Pay Bills
Customer
Actor
<<Include>>
<<Include>>
<<Include>>
<<Include>>
<<Include>>
Login
extension points
Invalid password entered
Abort triggered
Extension Point
Extend
<<Extend>>
<<Extend>>
Handle Abort
Handle Invalid Password

use case diagram example of Passenger Service.

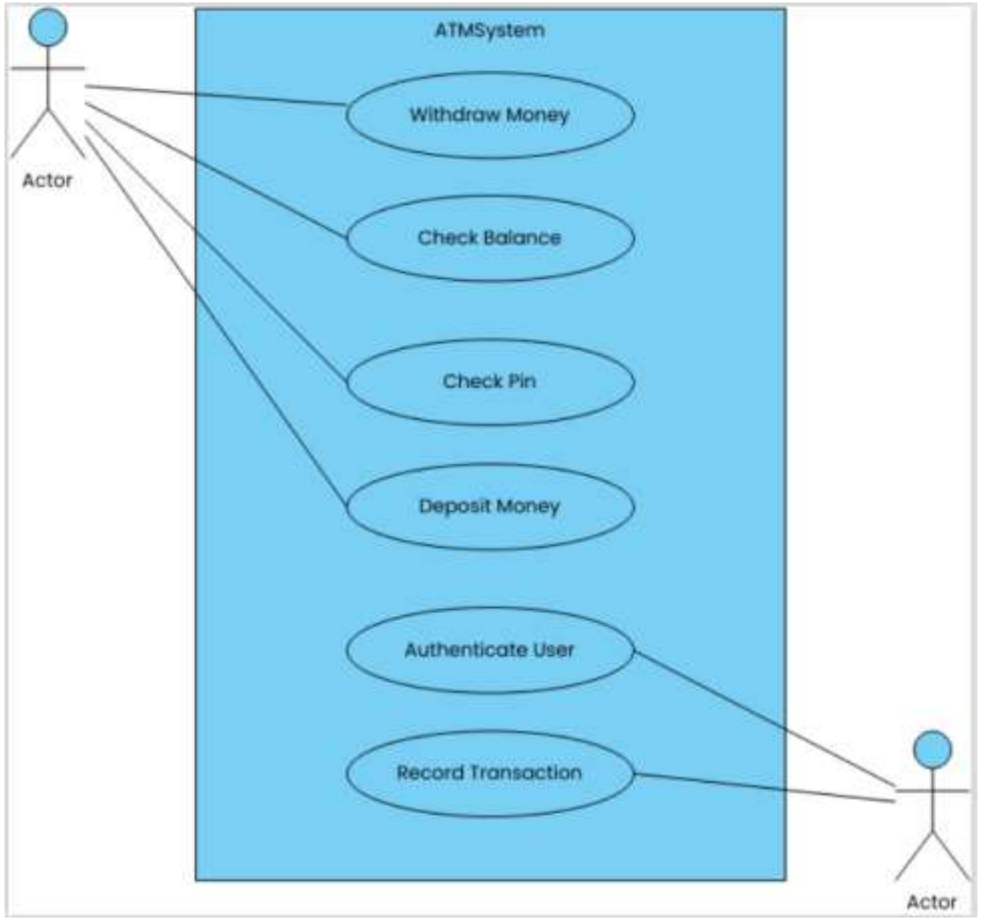


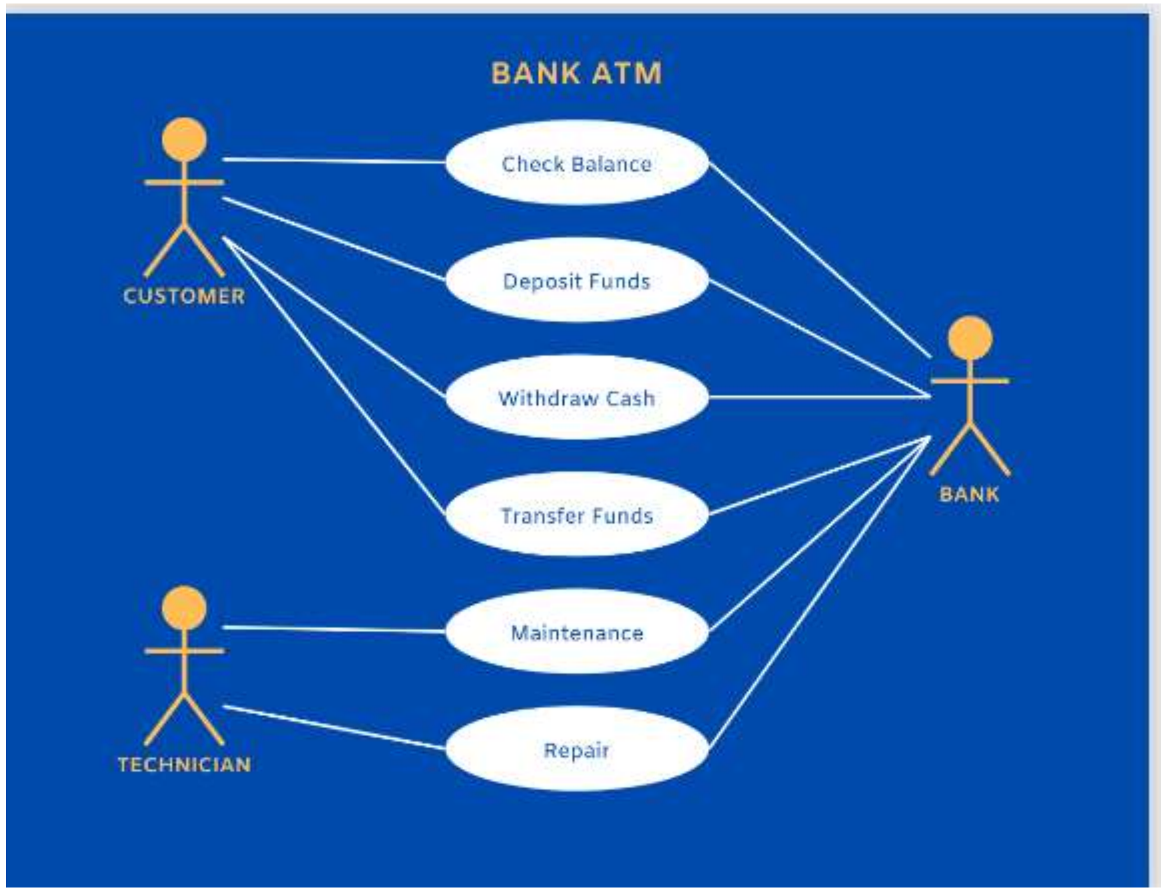

Simple ATM Machine System
System Reporting
<<include>>
System Maintainence
<<include>>
System Shutdown
Administrator
(Third-party)
Bad PIN
Login
Bank Customer
<<extend>>
<<include>>
Transaction
Bank
Print Receipt
Withdraw
Check Balance
Deposit


Travel agency
System
Book a tour
<<extend>>
Book airline ticket(s)
Reserve a seat
Deliver airline ticket(s)
<<extend>>
Arrange tour
Type something
Pay travel agent
<<include>>
Pay for airline ticket(s)
<<include>>
<<extend>>
Testing
Travel agent commission
Pay for tour
fddddfdff


SafeHome
Access camera surveillance via the Internet
Configure SafeHome system parameters
Set alarm
homeowner
cameras

use case diagram for ATM system




BANK ATM
CUSTOMER
Check Balance
Deposit Funds
Withdraw Cash
Transfer Funds
BANK
TECHNICIAN
Maintenance
Repair

Library Management System - UML Class Diagram

- Components of a Use Case Diagram
- Actors
- Use Cases
- Relationships

- **Actors**
- **Represents users or external systems interacting with the system.**
  - **Primary Actors:** Directly use the system (e.g., Customer, Librarian).
  - **Secondary Actors:** Support the system (e.g., Payment System)
- **Use Cases**
- **Represent system functionalities (actions users can perform).**
- Depicted as an **oval shape** with the name of the use case inside.
- Example: "Borrow Book", "Login", "Make Payment".

- **Relationships**
- **Association (—)**: A straight line connects actors to use cases to show interaction.
- **Include (<<include>>) Relationship**: A use case **must** include another use case (mandatory dependency).
- **Extend (<<extend>>) Relationship**: A use case may optionally extend another use case.

Library Management System Use Case Diagram

- **Actors:**
- **Librarian** (Manages books & members)
- **Member** (Borrows & returns books)
- **System** (Handles book records)

- **Use Cases:**
- "Search Book"
- "Borrow Book"
- "Return Book"
- "Manage Members"
- "Issue Fine" (extends "Return Book" if overdue)

- **Actors:**

1. **Member** 👤:
    1. Can **Search Book**, **Borrow Book**, and **Return Book**.
2. **Librarian** 👤:
    1. Manages library records by using **Manage Members** and **Issue Fine**.
3. **System** ⍰ (Optional Actor):
    1. Handles book records automatically.

Use Cases & Relationships:

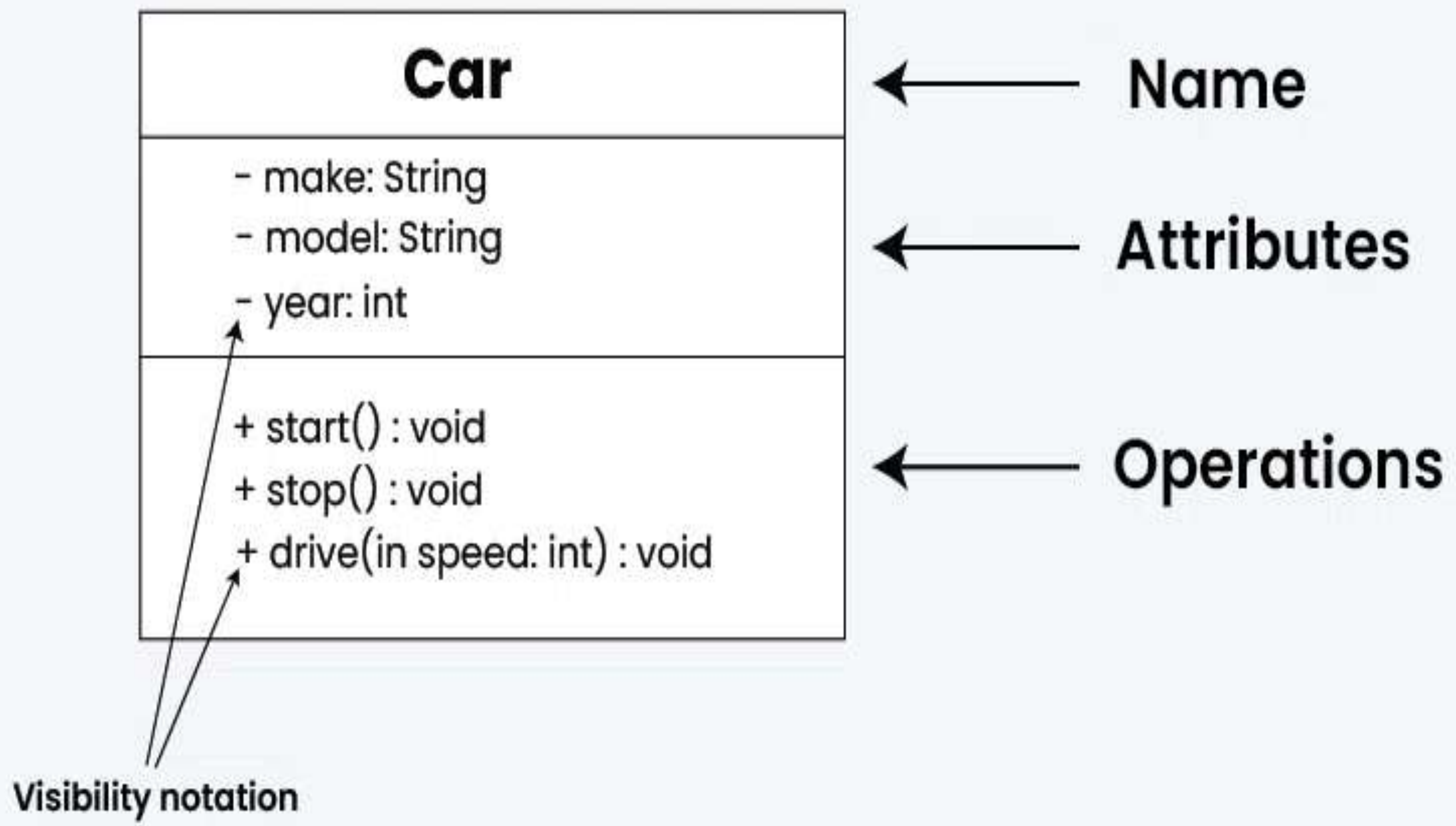
1. **Search Book 📚**:
    1. Allows members to find books in the library.
    2. Direct interaction with **Member**.
2. **Borrow Book** ⍰:
    1. Members can borrow a book if available.
    2. Direct interaction with **Member**.
3. **Return Book 🔄**:
    1. Members return borrowed books.
    2. Connected to **Issue Fine** via the **<<extend>>** relationship (A fine is issued if the return is late).

- **Manage Members** 📝:
- The Librarian can add, update, or remove members from the system.
- **Issue Fine** 💰:
- Extends the Return Book use case when a book is returned late.
- **Key UML Relationships Used**:
- **Straight Line (—):** Shows interaction between an actor and a use case.
- **Extend (<<extend>> Relationship):** The Issue Fine use case is triggered only if a book is returned late.

Class Diagram

- In UML, a class diagram is a static structure diagram that describes the classes in a system and their relationships.
- It models the attributes, operations, and associations among objects within the system. Class diagrams are one of the most common and essential parts of UML for illustrating the structure of an object-oriented system.
- Class Representation
- Each class is represented by a rectangle divided into three sections:

- Top section: Contains the class name.
- Middle section: Contains the attributes (fields or properties).
- Bottom section: Contains the methods (functions or operations).

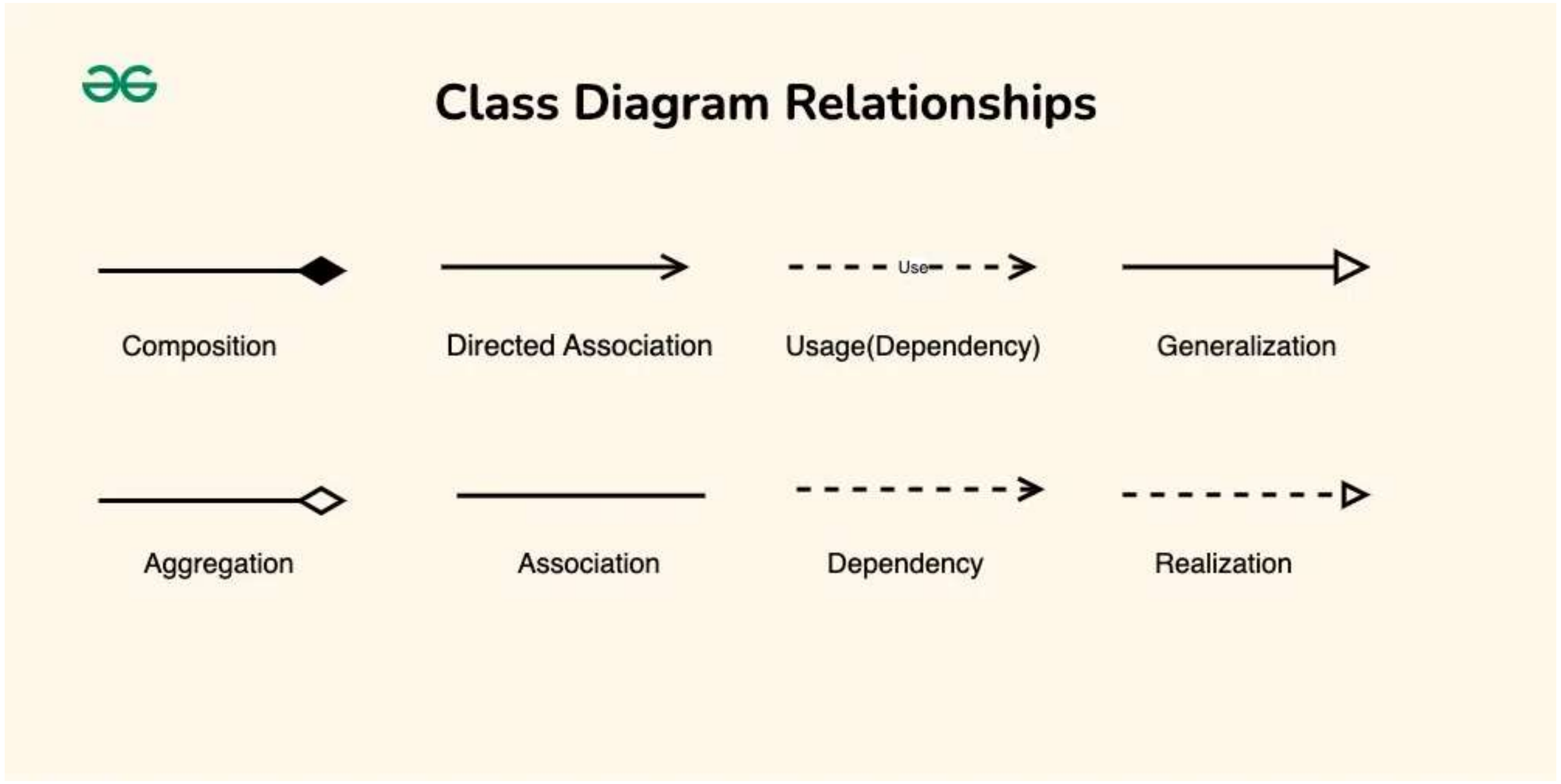


Class Notation

- Access Modifiers:
- + for public (visible to all classes)
- - for private (visible only within the class)
- # for protected (visible to subclasses)
- ~ for 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.

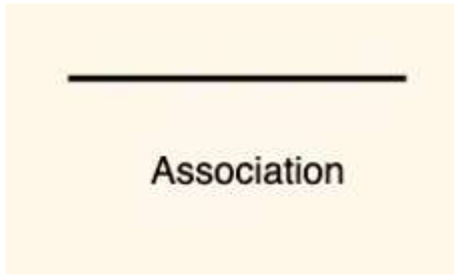
- There are three main parameter directionality notations used in class diagrams:
- **In (Input):**
  - An input parameter is a parameter passed from the calling object (client) to the called object (server) during a method invocation.
  - It is represented by an arrow pointing towards the receiving 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.


Class Diagram Relationships
Use
Composition
Directed Association
Usage(Dependency)
Generalization
Aggregation
Association
Dependency
Realization

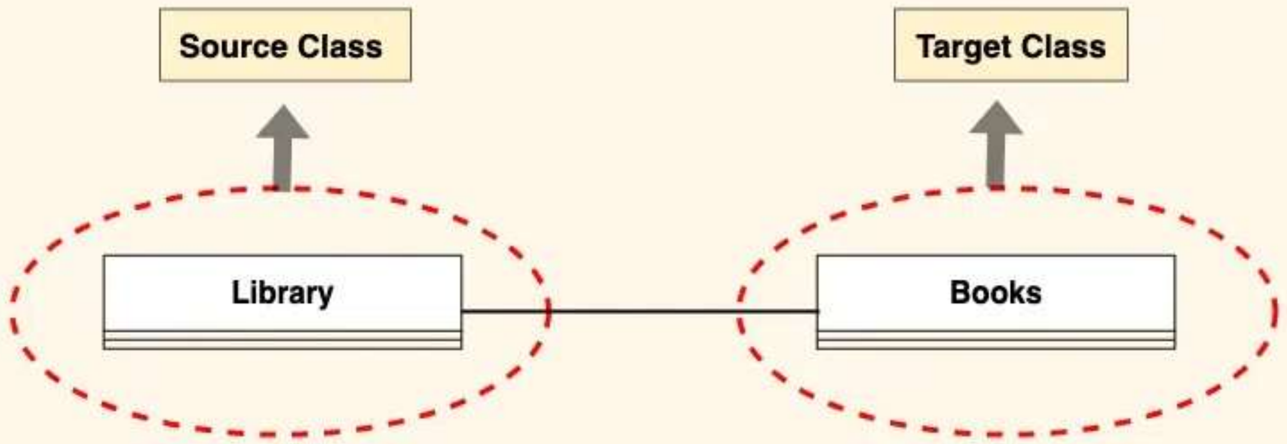
- **Association**

- 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.



Association Relationship

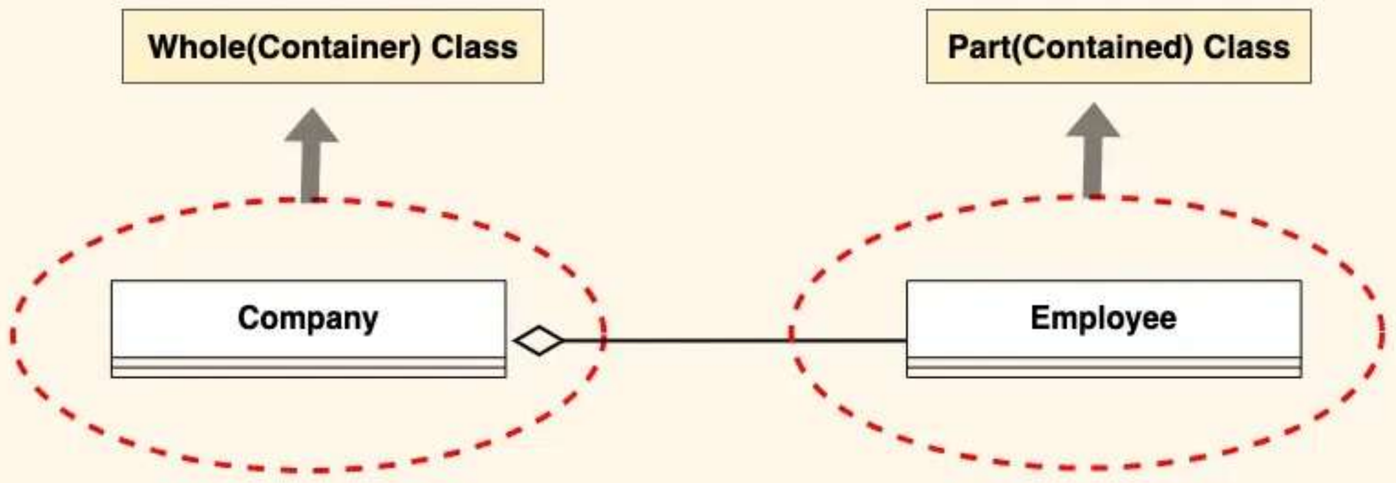


Each Library contains multiple Books, and each Book belongs to a specific Library

Aggregation

- Aggregation is a specialized form of association that represents a “whole-part” relationship.
- In this kind of relationship, the child class can exist independently of its parent class.

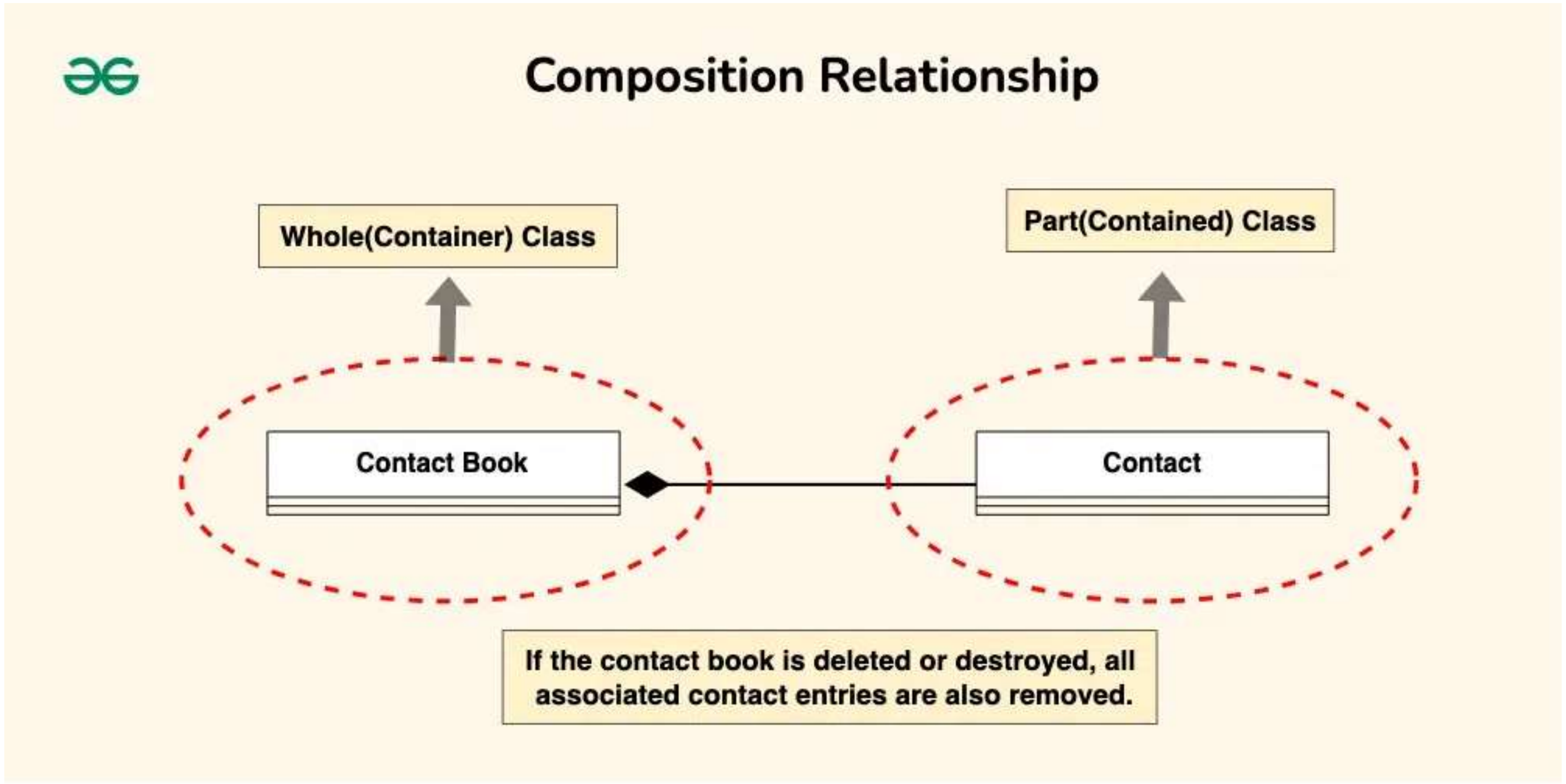
Aggregation Relationship



if the company ceases to exist, the employees can still exist independently.

Composition

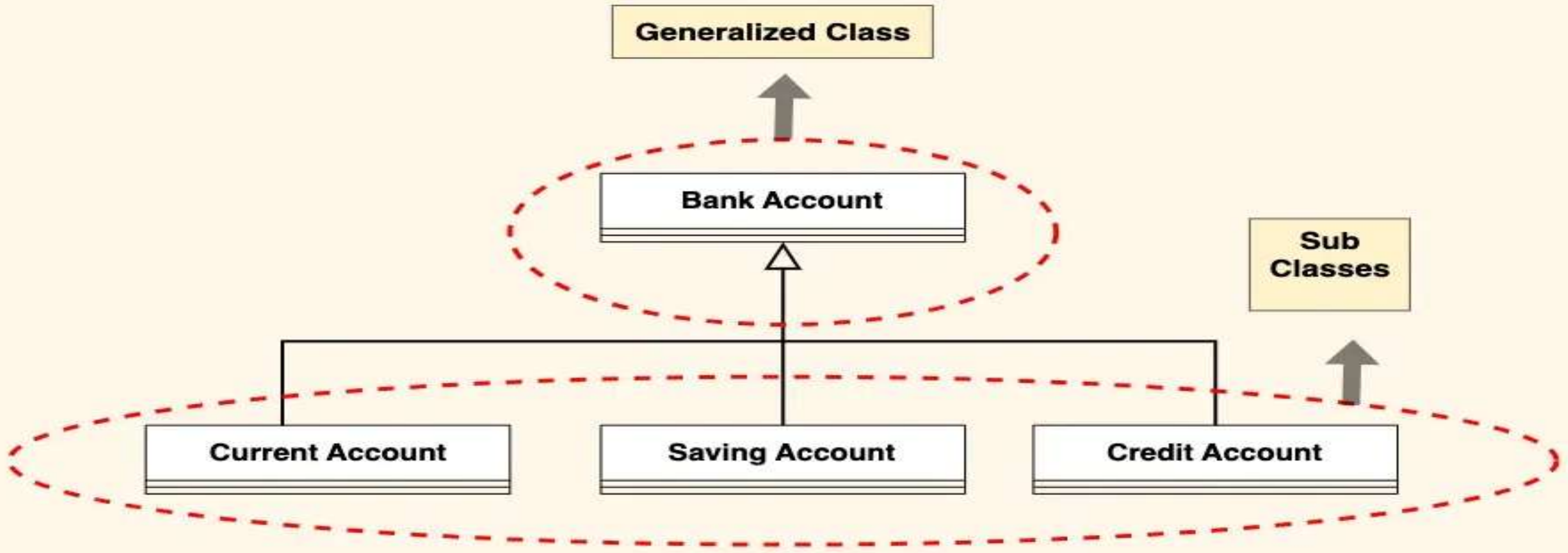
- Composition is a stronger form of aggregation, indicating a more significant ownership or dependency relationship.
- In composition, the part class cannot exist independently of the whole class.
- Composition is represented by a filled diamond shape on the side of the whole class.


Composition Relationship
Whole(Container) Class
Part(Contained) Class
Contact Book
Contact
If the contact book is deleted or destroyed, all associated contact entries are also removed.

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).

Generalization Relationship



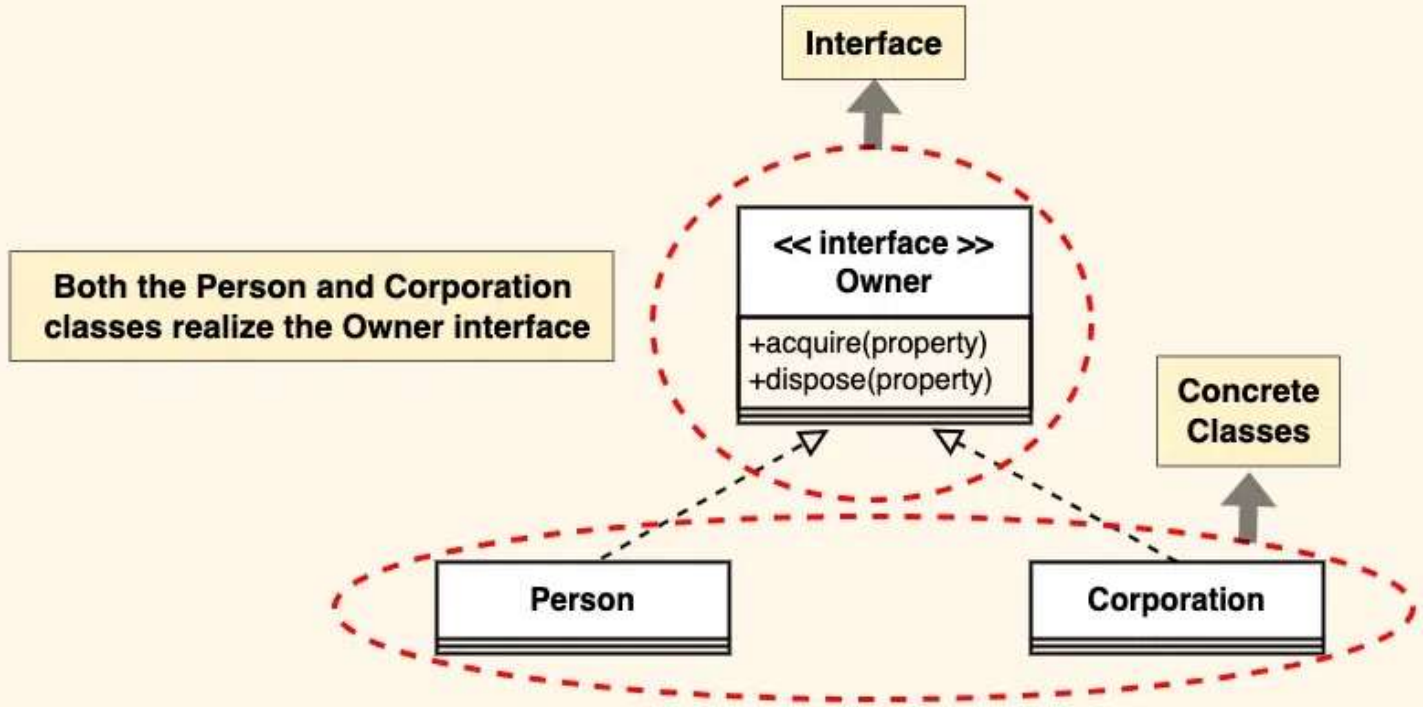
Current accounts, savings accounts, and credit accounts specialize from the account superclass, inheriting common banking functionalities

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.

- Let’s consider the scenario where a “Person” and a “Corporation” both realizing an “Owner” interface.
- **Owner Interface:** This interface now includes methods such as “acquire(property)” and “dispose(property)” to represent actions related to acquiring and disposing of property.
- **Person Class (Realization):** The Person class implements the Owner interface, providing concrete implementations for the “acquire(property)” and “dispose(property)” methods. For instance, a person can acquire ownership of a house or dispose of a car.
- **Corporation Class (Realization):** Similarly, the Corporation class also implements the Owner interface, offering specific implementations for the “acquire(property)” and “dispose(property)” methods. For example, a corporation can acquire ownership of real estate properties or dispose of company vehicles.

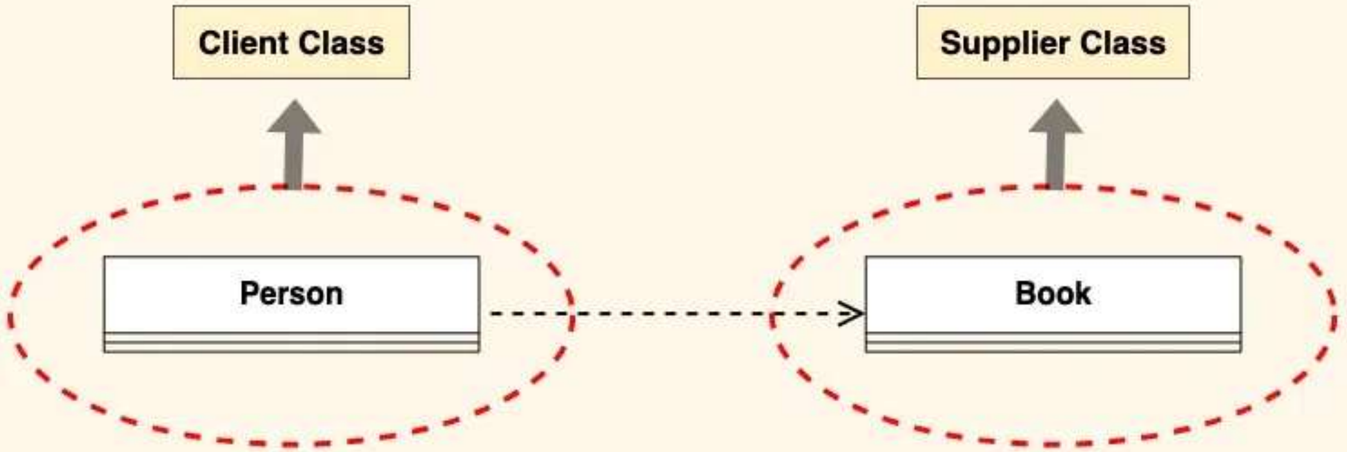
Realization Relationship



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.
- Dependencies are often depicted as a dashed arrow.

Dependency Relationship



The Person class depends on the Book class because it requires access to a book to read its content

- **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 diagram.
- 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 shows 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

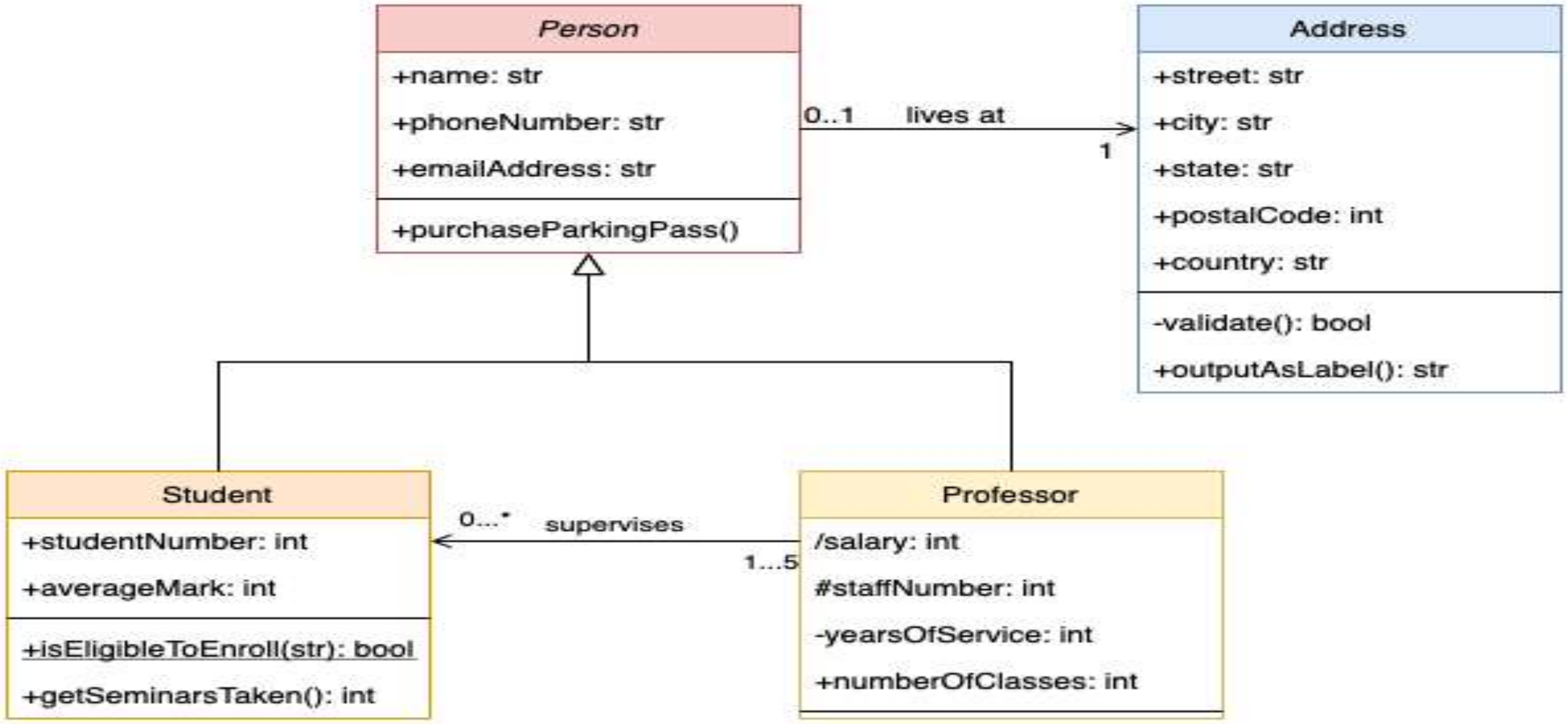
How to draw Class Diagram

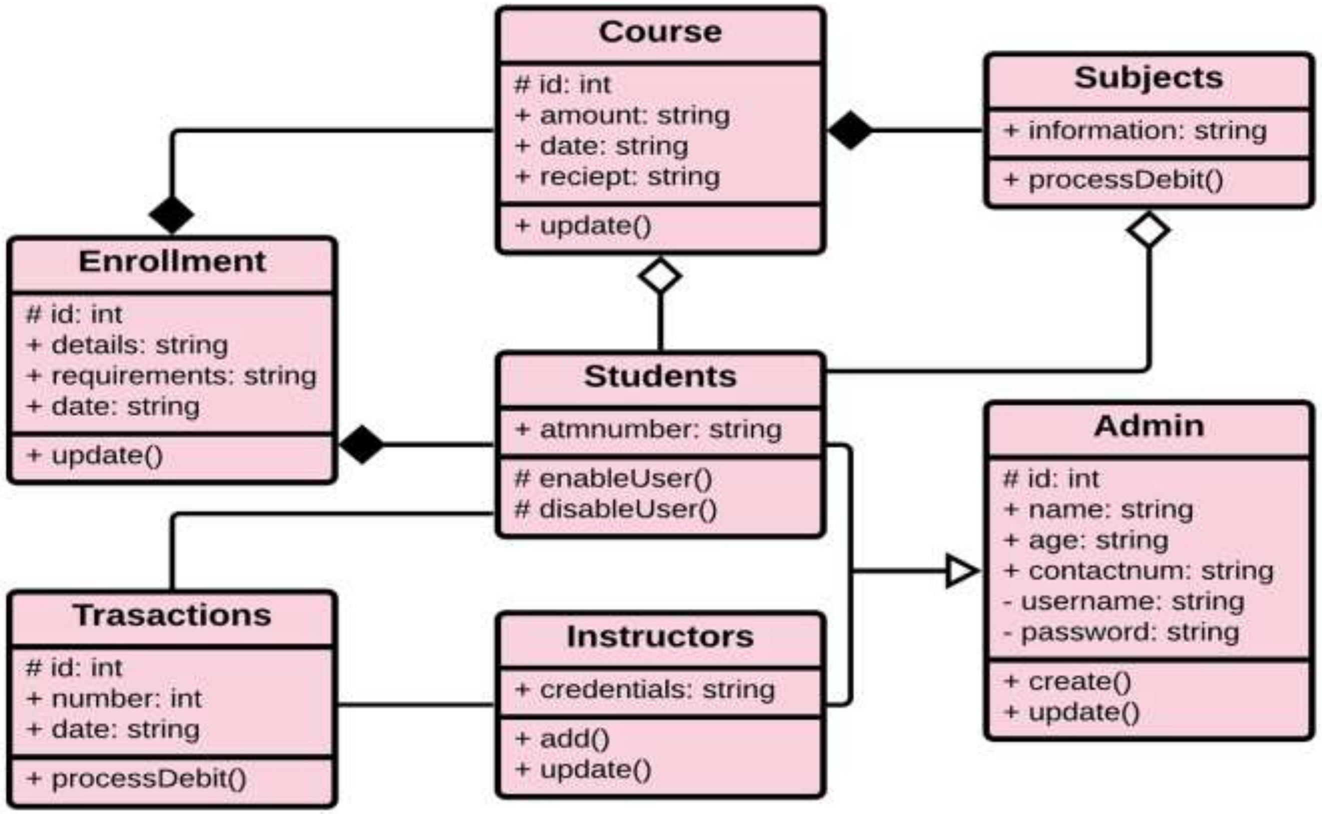
- **Step 1: Identify Classes:**
- **Step 2: List Attributes and Methods:**
- **Step 3: Identify Relationships:**
- **Step 4: Create Class Boxes:**
- **Step 5: Add Attributes and Methods:**
- **Step 6: Draw Relationships:**
- **Step 7: Label Relationships:**
- **Step 8: Review and Refine:**

Relationships Between Classes

- Association: A structural relationship that represents a connection between classes. Associations are drawn with a solid line between classes.

- Multiplicity: Specifies the number of instances in the relationship. For example:
- 1 for exactly one
- 0..1 for zero or one
- * for many
- 1..* for one or more
- Aggregation: A special form of association that represents a "whole-part" relationship where the part can exist independently of the whole. It’s represented by a line with an open diamond at the "whole" end.

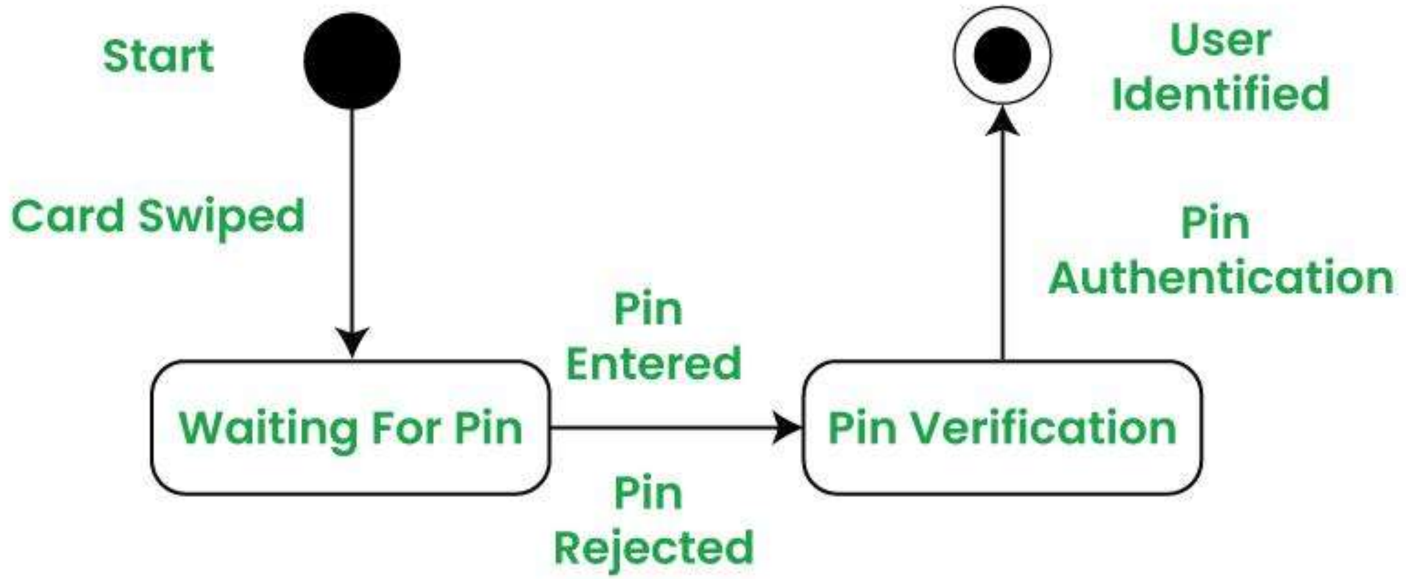

Person
+name: str
+phoneNumber: str
+emailAddress: str
+purchaseParkingPass()
0..1
lives at
1
Address
+street: str
+city: str
+state: str
+postalCode: int
+country: str
-validate(): bool
+outputAsLabel(): str
Student
+studentNumber: int
+averageMark: int
+isEligibleToEnroll(str): bool
+getSeminarsTaken(): int
0...*
supervises
1...5
Professor
/salary: int
#staffNumber: int
-yearsOfService: int
+numberOfClasses: int


Course
# id: int
+ amount: string
+ date: string
+ reciept: string
+ update()
Subjects
+ information: string
+ processDebit()
Enrollment
# id: int
+ details: string
+ requirements: string
+ date: string
+ update()
Students
+ atmnumber: string
# enableUser()
# disableUser()
Admin
# id: int
+ name: string
+ age: string
+ contactnum: string
- username: string
- password: string
+ create()
+ update()
Trasactions
# id: int
+ number: int
+ date: string
+ processDebit()
Instructors
+ credentials: string
+ add()
+ update()

State Machine Diagrams

- A **State Transition Diagram** (also called a **State Machine Diagram**) is a UML diagram that shows how an **object** transitions between different **states** based on **events or conditions**.

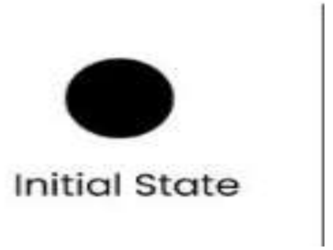
A State Machine Diagram for user verification



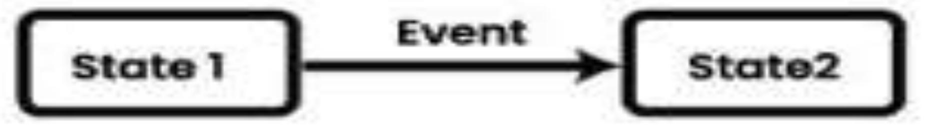
Basic components and notations of a State Machine diagram

- **a. States**
- Represents **different conditions** of an object.
- Shown as **rounded rectangles** (▯).
- **b. Transitions**
- Arrows **(→)** connecting states indicate **movement from one state to another**.
- Triggered by **events or conditions**.
- **c. Initial State**
- **Starting point** of the system.
- Represented by a **black circle (●)**.
- **d. Final State**
- Represents **end of the process**.
- Shown as a **black circle inside a white circle (○)**.
- **e. Events & Actions**
- **Events** trigger transitions (e.g., "User clicks login").
- **Actions** are responses to events (e.g., "Validate password").

- **1. Initial state**
- We use a black filled circle represent the initial state of a System or a Class.

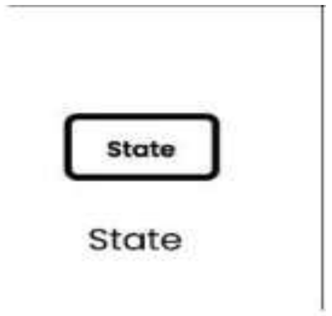


- **Transition**
- We use a solid arrow to represent the transition or change of control from one state to another. The arrow is labelled with the event which causes the change in state.



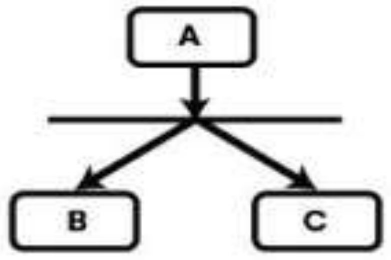
Transition

- **State**
- We use a rounded rectangle to represent a state. A state represents the conditions or circumstances of an object of a class at an instant of time.



- **Fork**

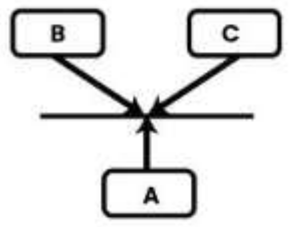
- We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states. We use the fork notation to represent a state splitting into two or more concurrent states.



Fork

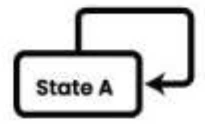
- **Join**

- We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events



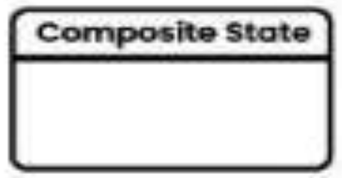
Join

- **Self transition**
- We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases.



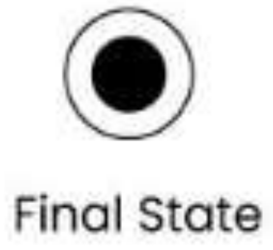
Self Transition

- **Composite state**
- We use a rounded rectangle to represent a composite state also. We represent a state with internal activities using a composite state.

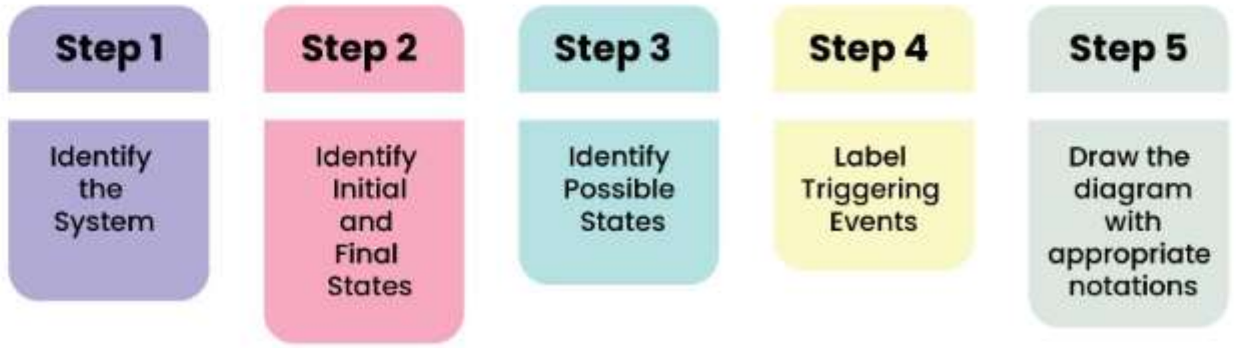


Composite State

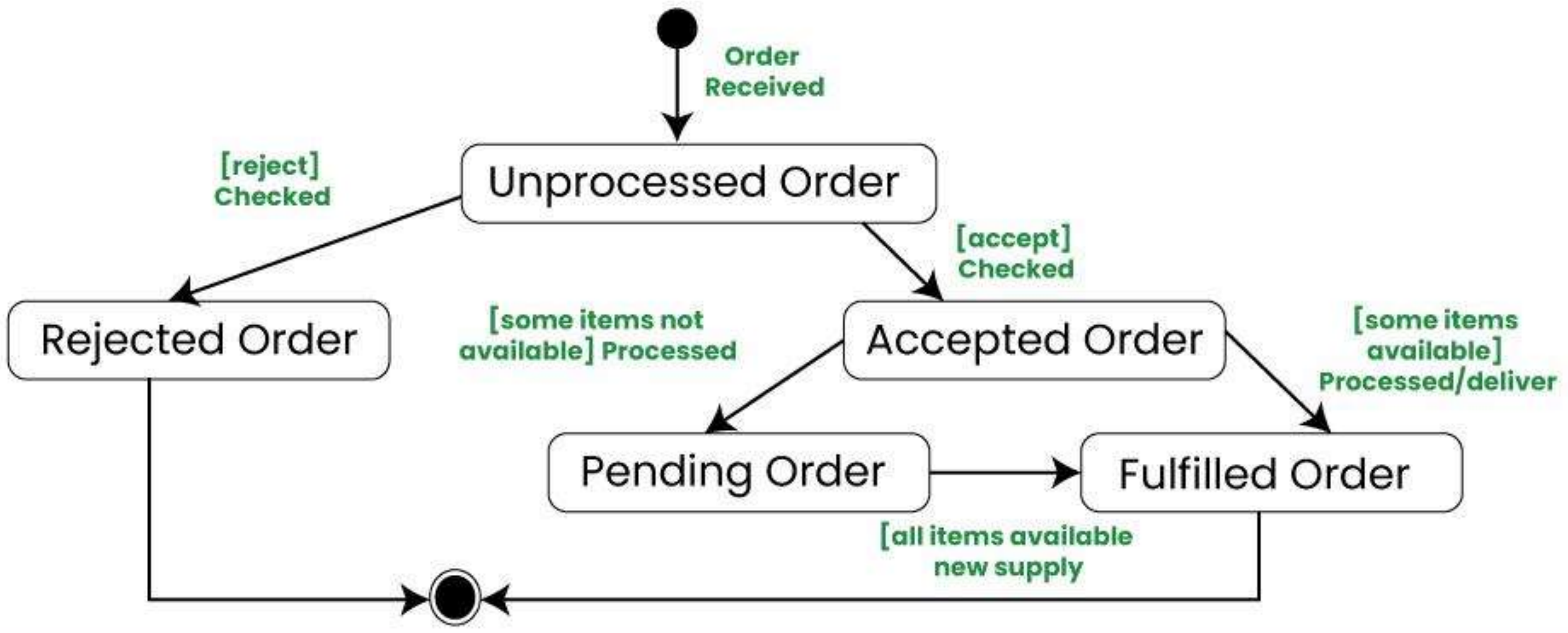
- **Final State**
- We use a filled circle within a circle notation to represent the final state in a state machine diagram.



How to draw a State Machine diagram in UML?



State machine diagram for an online order

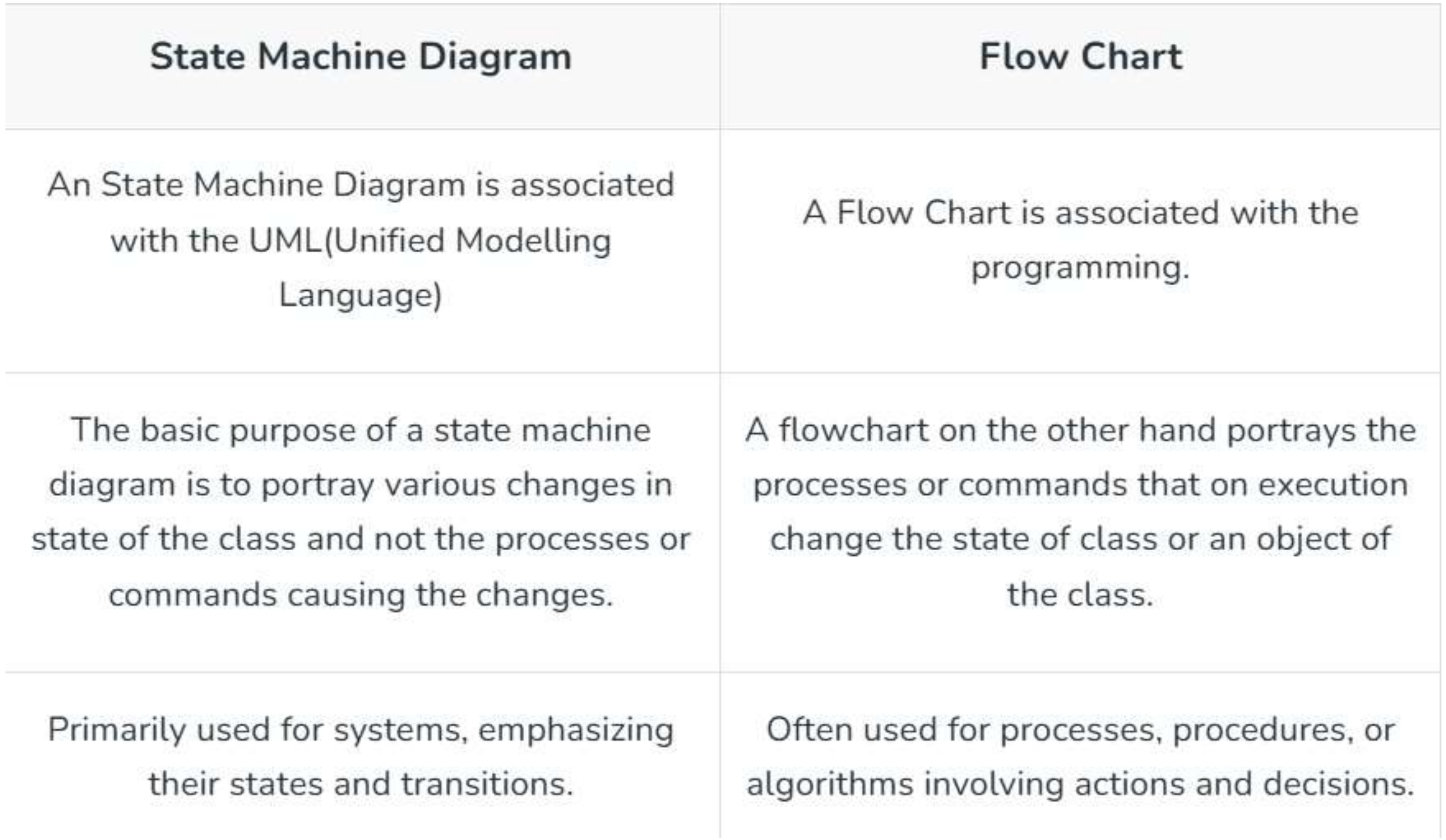


- On the event of an order being received, we transit from our initial state to Unprocessed order state.
- The unprocessed order is then checked.
- If the order is rejected, we transit to the Rejected Order state.
- If the order is accepted and we have the items available we transit to the fulfilled order state.
- However if the items are not available we transit to the Pending Order state.
- After the order is fulfilled, we transit to the final state. In this example, we merge the two states i.e. Fulfilled order and Rejected order into one final state.

Applications of State Machine Diagram

- State Machine Diagrams are very useful for modeling and visualizing the dynamic behavior of a system.
- They are also used in UI design where they help to illustrate how the interface changes in response to user actions, helping designers to create a better use experience.
- In game design, state machine diagrams can help model the behavior of characters or objects, detailing how they change states based on player interactions or game events
- In embedded systems, where hardware interacts with software to perform tasks, State Machine Diagrams are valuable for representing the control logic and behavior of the system.

Differences between a State Machine Diagram and a Flowchart?



State Machine Diagram	Flow Chart
An State Machine Diagram is associated with the UML(Unified Modelling Language)	A Flow Chart is associated with the programming.
The basic purpose of a state machine diagram is to portray various changes in state of the class and not the processes or commands causing the changes.	A flowchart on the other hand portrays the processes or commands that on execution change the state of class or an object of the class.
Primarily used for systems, emphasizing their states and transitions.	Often used for processes, procedures, or algorithms involving actions and decisions.