

# Unified Modeling Language Diagram



**Marwadi**  
University

Department of  
Information and  
Communication  
Technology

Software  
Engineering

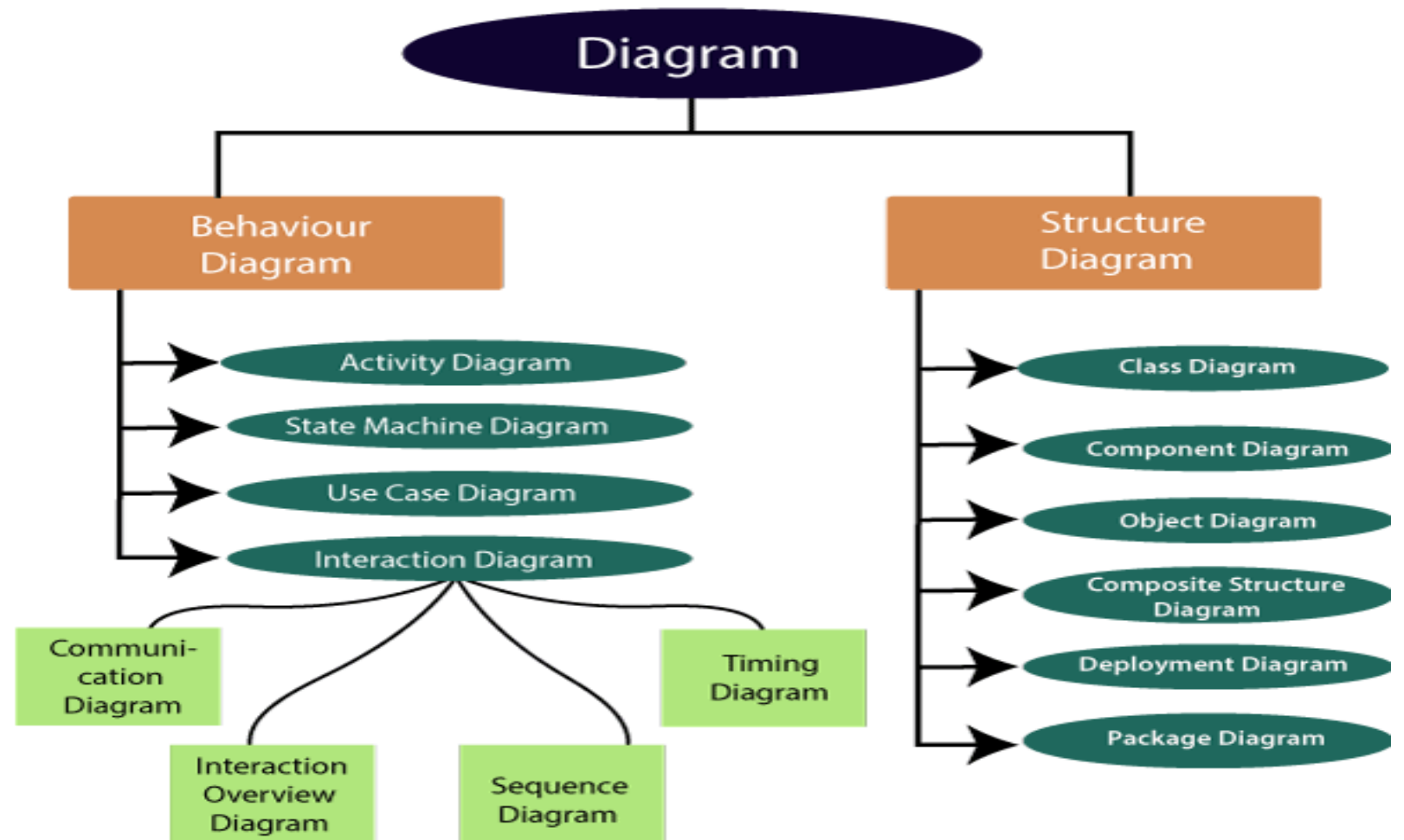
By  
Prof. Suhag Baldaniya

# Introduction

- A UML diagram is a diagram based on the UML (Unified Modeling Language) with the purpose of visually representing a system along with its main actors, roles, actions, artifacts or classes, in order to better understand, alter, maintain, or document information about the system

# UML- Diagrams

- The UML diagrams are categorized into structural diagrams, behavioral diagrams, and also interaction overview diagrams. The diagrams are hierarchically classified in the following figure:



# Structural Diagrams

- Structural diagrams depict a static view or structure of a system.
- It is widely used in the documentation of software architecture.
- It embraces **class diagrams**, **composite structure diagrams**, component diagrams, deployment diagrams, object diagrams, and package diagrams.
- It presents an outline for the system.
- It stresses the elements to be present that are to be modeled.

# Class Diagram

- The class diagram depicts a static view of an application.
- It represents the types of objects residing in the system and the relationships between them.
- A class consists of its objects, and also it may inherit from other classes.
- A class diagram is used to visualize, describe, document various different aspects of the system, and also construct executable software code.
- It shows the attributes, classes, functions, and relationships to give an overview of the software system.
- It constitutes class names, attributes, and functions in a separate compartment that helps in software development.
- Since it is a collection of classes, interfaces, associations, collaborations, and constraints, it is termed as a structural diagram.

# Purpose of Class Diagrams

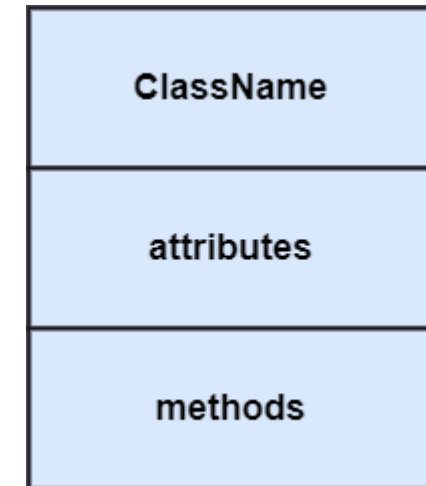
- It analyses and designs a static view of an application.
- It describes the major responsibilities of a system.
- It is a base for component and deployment diagrams.
- It incorporates forward and reverse engineering.

# Benefits of Class Diagrams

- It can represent the object model for complex systems.
- It reduces the maintenance time by providing an overview of how an application is structured before coding.
- It provides a general schematic of an application for better understanding.
- It represents a detailed chart by highlighting the desired code, which is to be programmed.
- It is helpful for the stakeholders and the developers.

# Vital components of a Class Diagram

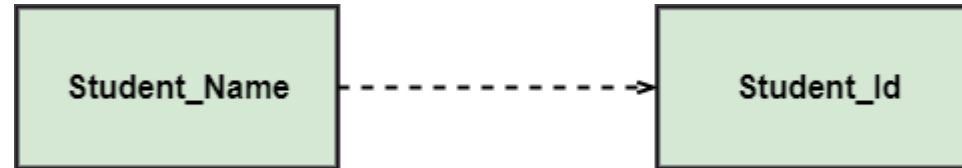
- Upper Section: The upper section encompasses the name of the class. A class is a representation of similar objects that shares the same relationships, attributes, operations, and semantics
- Middle Section: The middle section constitutes the attributes, which describe the quality of the class
- Lower Section: The lower section contain methods or operations. The methods are represented in the form of a list, where each method is written in a single line. It demonstrates how a class interacts with data.





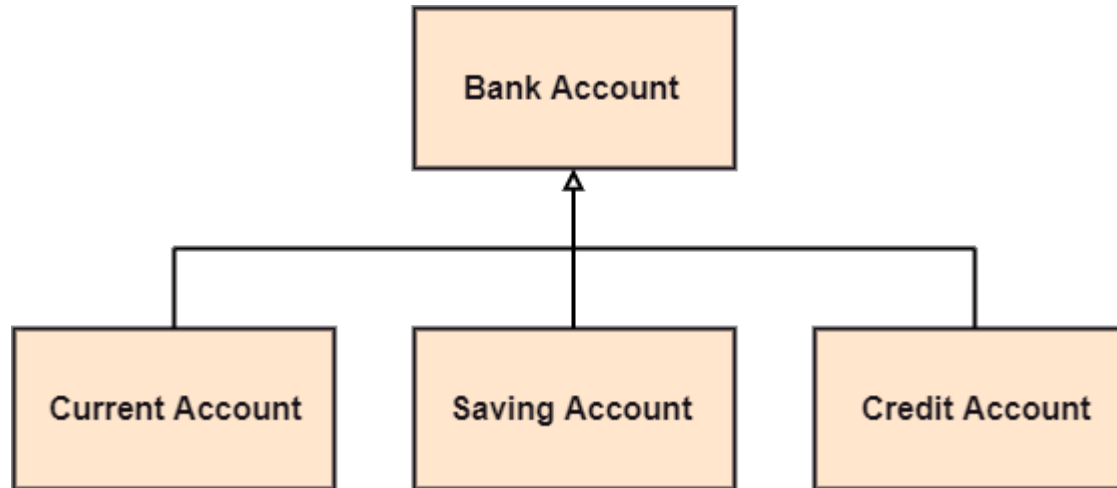
# Relationships

- In UML, relationships are of three types:
- **Dependency**: A dependency is a semantic relationship between two or more classes where a change in one class cause changes in another class.
- It forms a weaker relationship. In the following example, **Student Name** is dependent on the Student\_Id.



Continue...

- **Generalization:** A generalization is a relationship between a parent class (superclass) and a child class (subclass). In this, the child class is inherited from the parent class.
- For example, The Current Account, Saving Account, and Credit Account are the generalized form of Bank Account



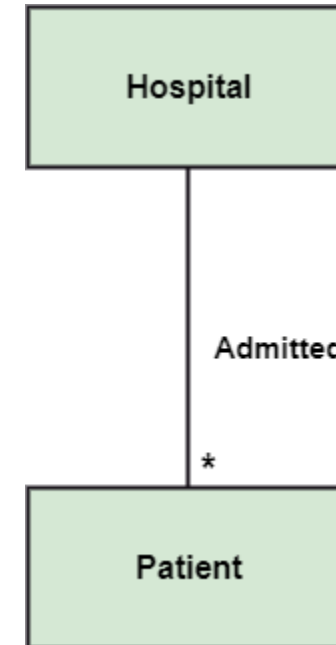
Continue...

- **Association:** It describes a static or physical connection between two or more objects. It depicts how many objects are there in the relationship.
- For example, a department is associated with the college.



# Multiplicity

- Multiplicity: It defines a specific range of allowable instances of attributes. In case if a range is not specified, one is considered as a default multiplicity



# Aggregation

- Aggregation: An aggregation is a subset of association, which represents has a relationship.
- It is more specific than association.
- It defines a part-whole or part-of relationship.
- In this kind of relationship, the child class can exist independently of its parent class.
- The company encompasses a number of employees, and even if one employee resigns, the company still exists.



# Composition

- Composition: The composition is a subset of aggregation. It portrays the dependency between the parent and its child, which means if one part is deleted, then the other part also gets discarded. It represents a whole-part relationship.
- A contact book consists of multiple contacts, and if you delete the contact book, all the contacts will be lost

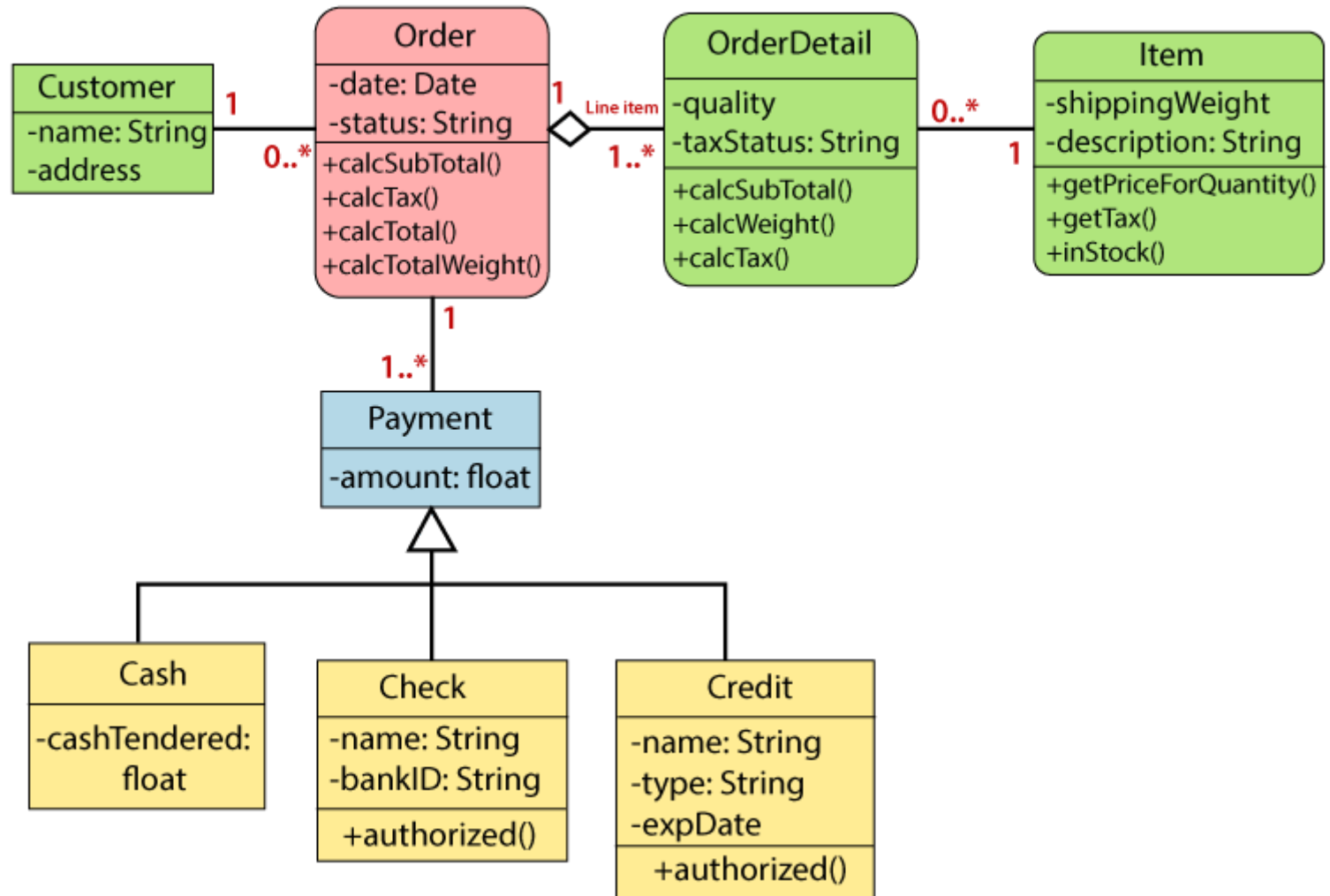


# How to draw a Class Diagram?

- To describe a complete aspect of the system, it is suggested to give a meaningful name to the class diagram.
- The objects and their relationships should be acknowledged in advance.
- The attributes and methods (responsibilities) of each class must be known.
- A minimum number of desired properties should be specified as more number of the unwanted property will lead to a complex diagram.
- Notes can be used as and when required by the developer to describe the aspects of a diagram.
- The diagrams should be redrawn and reworked as many times to make it correct before producing its final version.

# Example

- A class diagram describing the sales order system is given below.





# Usage of Class diagrams

- The class diagram is used to represent a static view of the system.
- It plays an essential role in the establishment of the component and deployment diagrams.
- It helps to construct an executable code to perform forward and backward engineering for any system, or we can say it is mainly used for construction.
- It represents the mapping with object-oriented languages that are C++, Java, etc.
- Class diagrams can be used for the following purposes:
  - To describe the static view of a system.
  - To show the collaboration among every instance in the static view.
  - To describe the functionalities performed by the system.
  - To construct the software application using object-oriented languages.

# Component Diagram

- A component diagram is used to break down a large object-oriented system into the smaller components, so as to make them more manageable.
- It models the physical view of a system such as executables, files, libraries, etc. that resides within the node.
- It visualizes the relationships as well as the organization between the components present in the system.
- It helps in forming an executable system. A component is a single unit of the system, which is replaceable and executable.
- The implementation details of a component are hidden, and it necessitates an interface to execute a function. It is like a black box whose behavior is explained by the provided and required interfaces.

# Purpose of a Component Diagram

- It describes all the individual components that are used to make the functionalities, but not the functionalities of the system
- It visualizes the physical components inside the system. The components can be a library, packages, files, etc.
- The main purpose of the component diagram are enlisted below:
  - It visualize each component of a system.
  - It constructs the executable by incorporating forward and reverse engineering.
  - It depicts the relationships and organization of components.

# Why use Component Diagram?

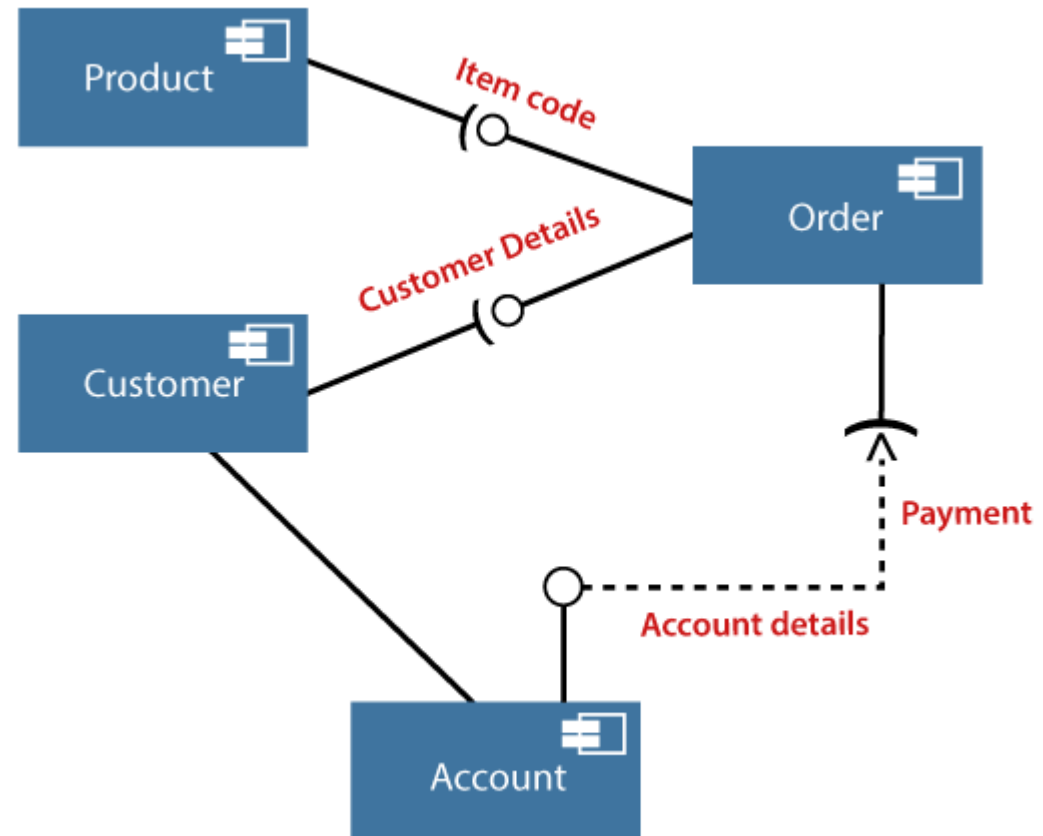
- It is used to portray the functionality and behavior of all the components present in the system, unlike other diagrams that are used to represent the architecture of the system, working of a system, or simply the system itself.
- Following are some reasons for the requirement of the component diagram:
  - It portrays the components of a system at the runtime.
  - It is helpful in testing a system.
  - It envisions the links between several connections.

# How to Draw a Component Diagram?

- Following are some artifacts that are needed to be identified before drawing a component diagram:
  - What files are used inside the system?
  - What is the application of relevant libraries and artifacts?
  - What is the relationship between the artifacts?
- Following are some points that are needed to be kept in mind after the artifacts are identified:
  - Using a meaningful name to ascertain the component for which the diagram is about to be drawn.
  - Before producing the required tools, a mental layout is to be made.
  - To clarify the important points, notes can be incorporated.

# Example

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



# 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 determine 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.

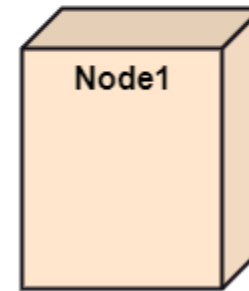
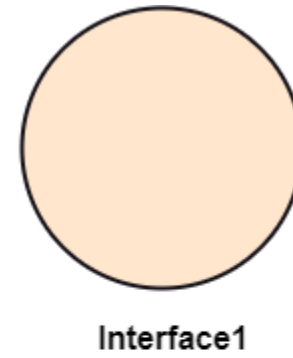
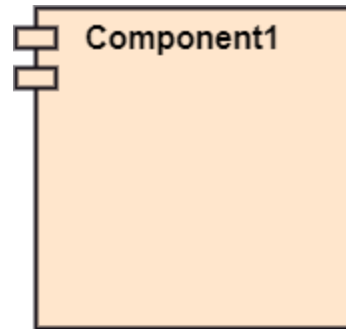
# Purpose of Deployment Diagram

- 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.
- 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.

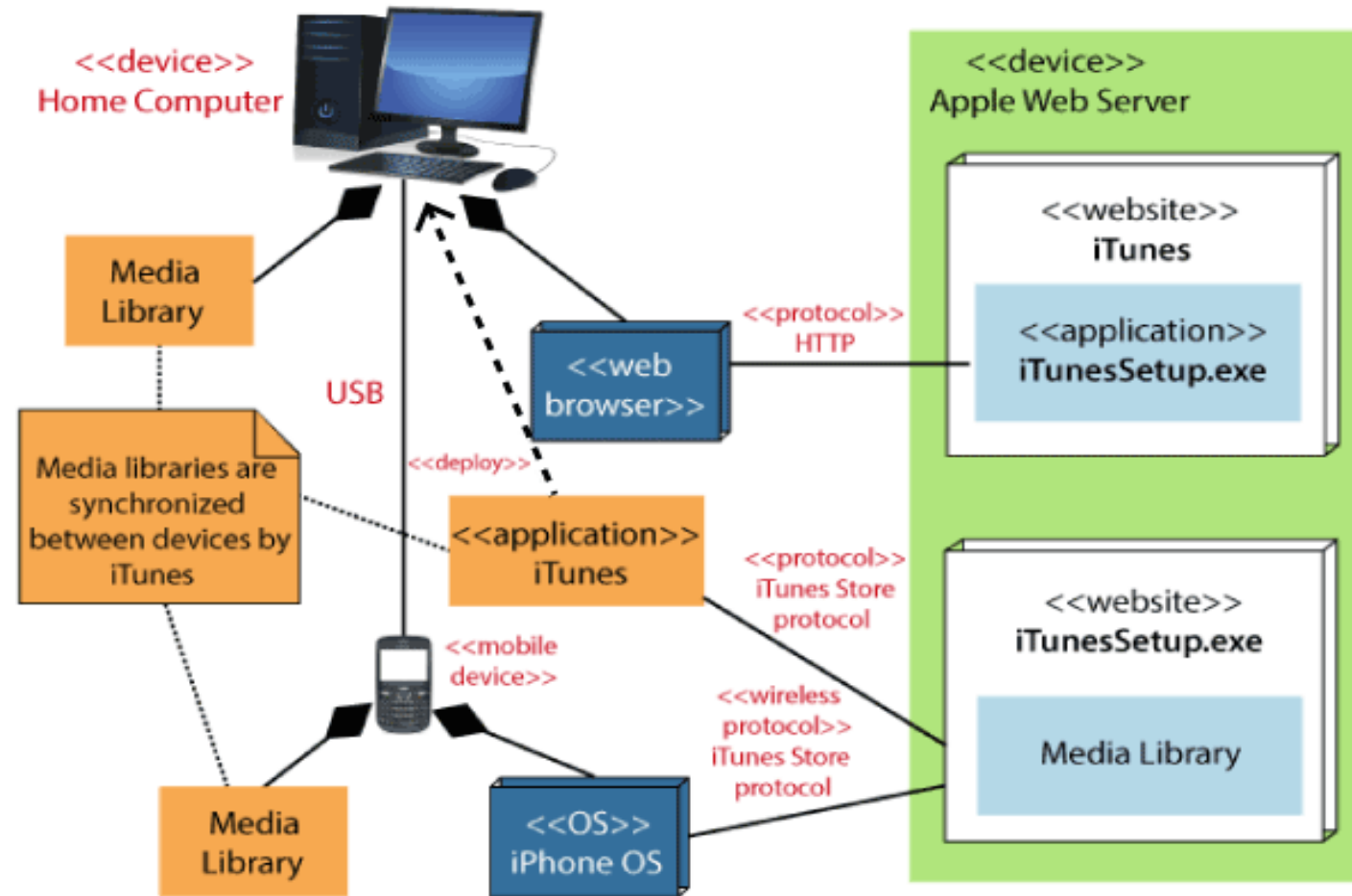


# Symbol and notation of Deployment diagram

- The deployment diagram consist of the following notations:
  - A component
  - An artifact
  - An interface
  - A node



# Example of a Deployment diagram



# UML diagram creation tips

- **Determine the purpose and scope of the diagram:** This will help you decide which type of UML diagram to create and what elements to include.
- **Use standard symbols and notations:** attach to UML standards to ensure that your diagram is easily readable and understandable by others.
- **Keep it simple:** Avoid adding unnecessary details or making the diagram too complex. The goal is to communicate the design effectively, not to show every single detail.
- **Organize the diagram:** Use clear and consistent layout, grouping related elements together, and using lines to show relationships between elements.
- **Use the right level of abstraction:** Show only what's necessary for the purpose of the diagram, and leave out unnecessary details.
- **Label elements clearly:** Provide meaningful names for classes, attributes, and operations, and use clear and concise labels for relationships.
- **Validate the diagram:** Ensure that the diagram accurately represents the design and that it can be used to generate code or provide a clear understanding of the system being modeled.
- **Update the diagram as needed:** UML diagrams are living documents that should be updated as the design evolves.

# UML Behavioral Diagram

- Behavioral diagrams portray a dynamic view of a system or the behavior of a system, which describes the functioning of the system.
- It includes use case diagrams, state diagrams, and activity diagrams. It defines the interaction within the system.

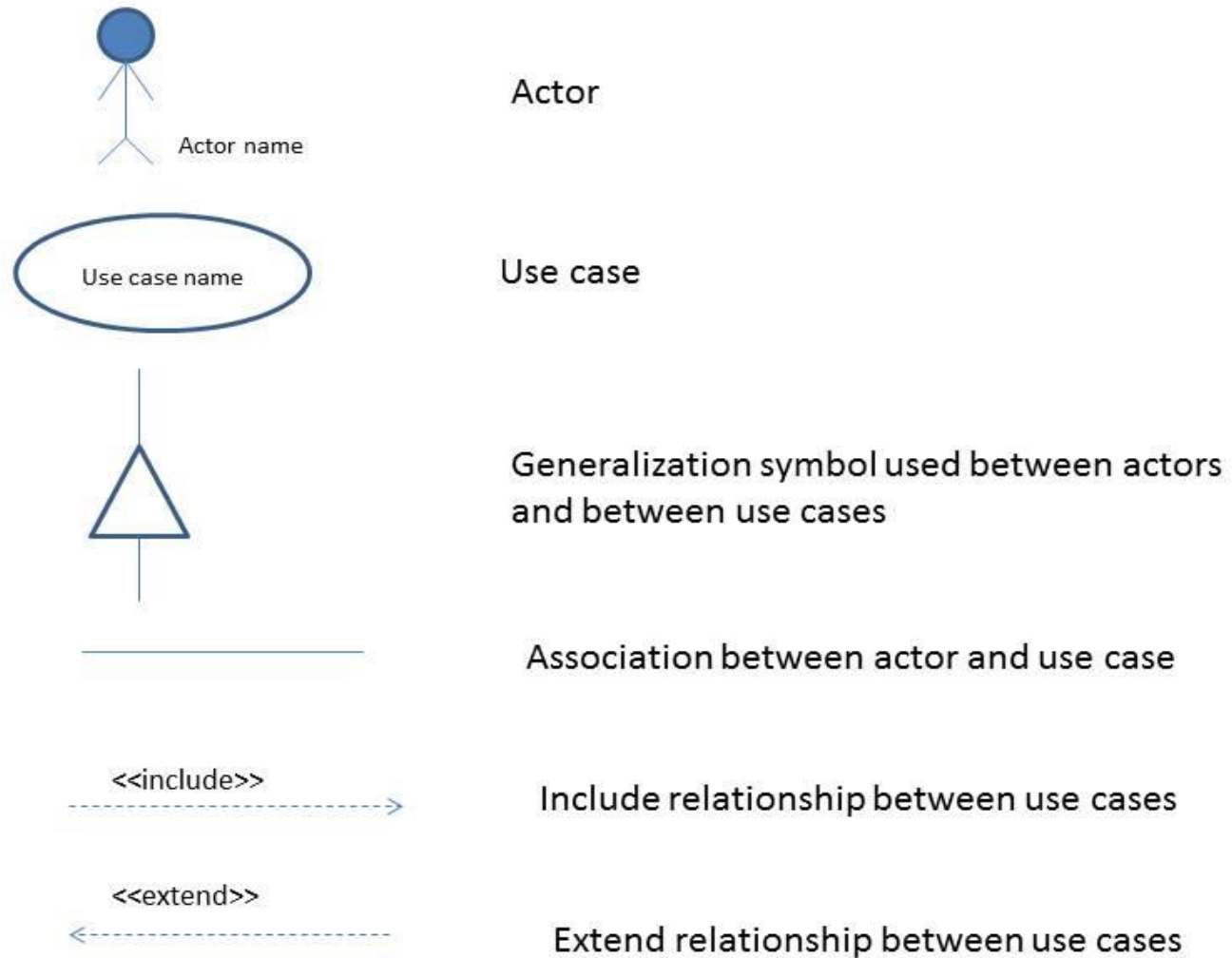
# Use case diagram

- A use case diagram is used to represent the dynamic behavior of a system.
- It encapsulates the system's functionality by incorporating use cases, actors, and their relationships.
- It models the tasks, services, and functions required by a system/subsystem of an application.
- It depicts the high-level functionality of a system and also tells how the user handles a system.

# Purpose of Use Case Diagrams

- The main purpose of a use case diagram is to portray the dynamic aspect of a system.
- It accumulates the system's requirement, which includes both internal as well as external influences.
- It invokes persons, use cases, and several things that invoke the actors and elements accountable for the implementation of use case diagrams.
- It represents how an entity from the external environment can interact with a part of the system.
- Following are the purposes of a use case diagram given below:
  - It gathers the system's needs.
  - It portray the external view of the system.
  - It recognizes the internal as well as external factors that influence the system.
  - It represents the interaction between the actors.

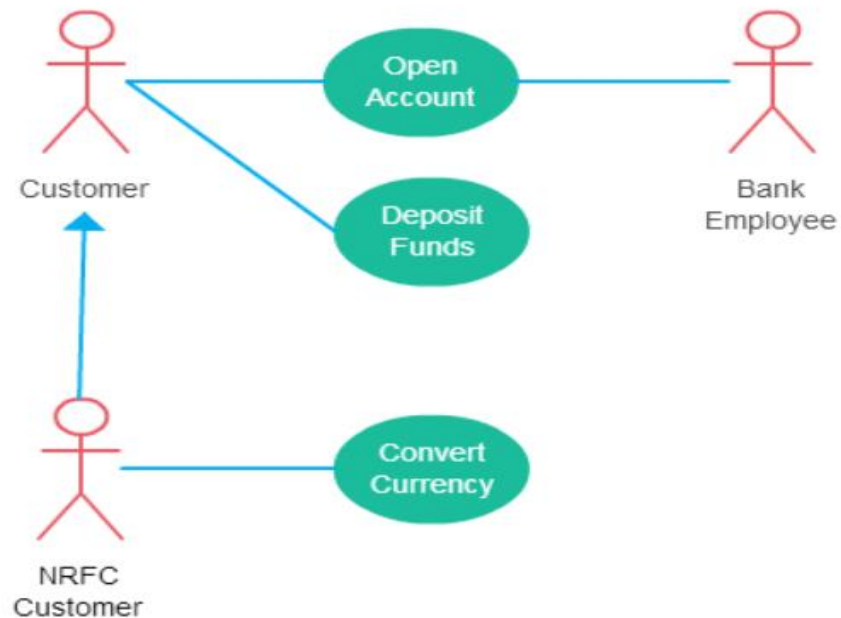
# Notations



**Symbols in a use case diagram**

# Notations

- **Actor:** An actor represents an external entity that interacts with a system. Since it is external to the system, the actor itself is not fully modeled by the system.
- **Use Case:** A use case represents a functionality (typically a requirement) that is expected to be implemented by the system.
- **Generalization:** This represents a relationship between actors or between use cases

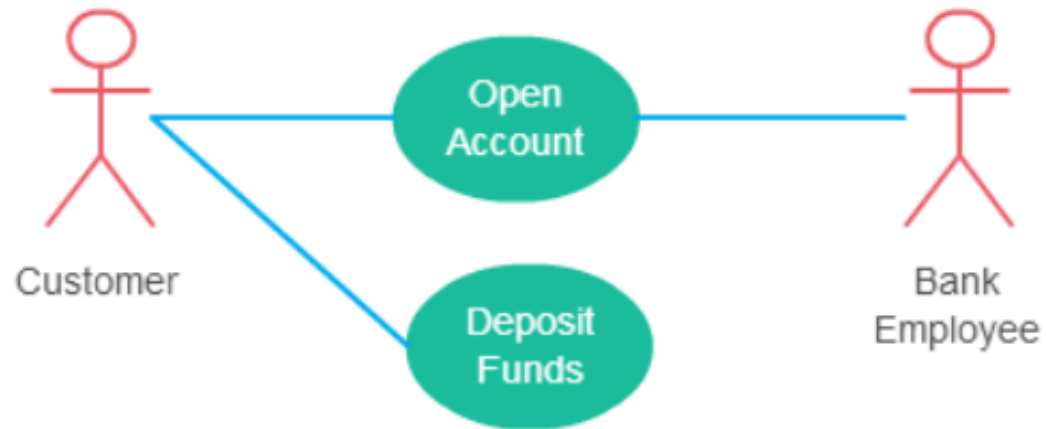


*A generalized actor in an use case diagram*



# Notations

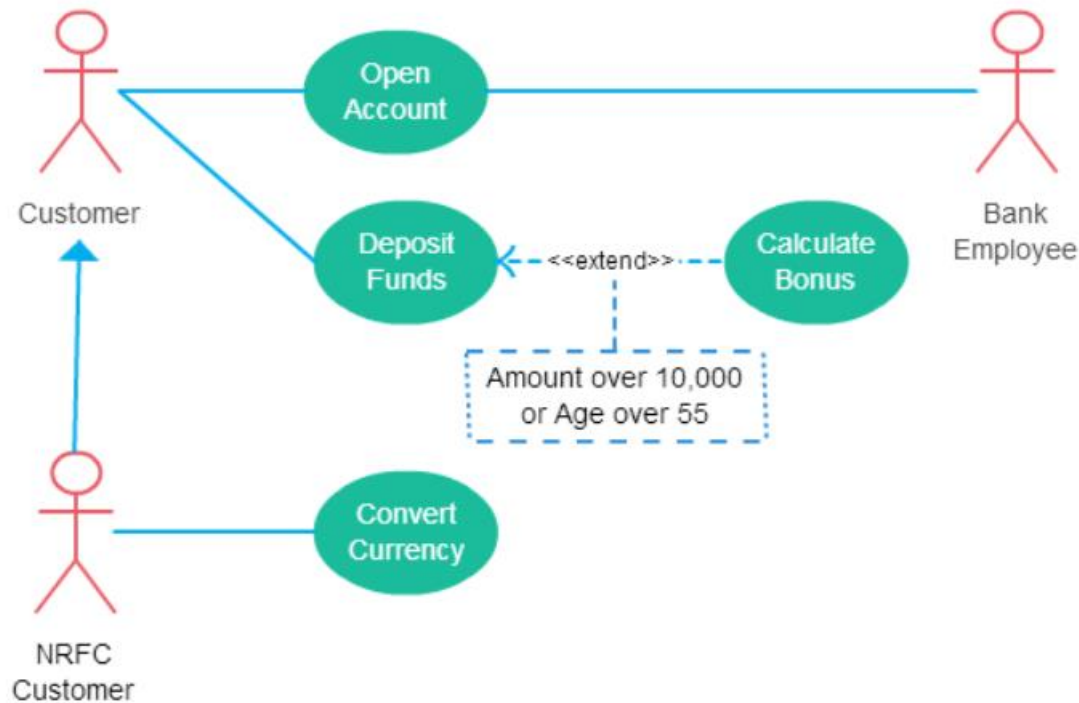
- **Association:** This represents a two-way communication between an actor and a use case and hence is a binary relation.



*Different ways association relationship appears in use case diagrams*

# Notations

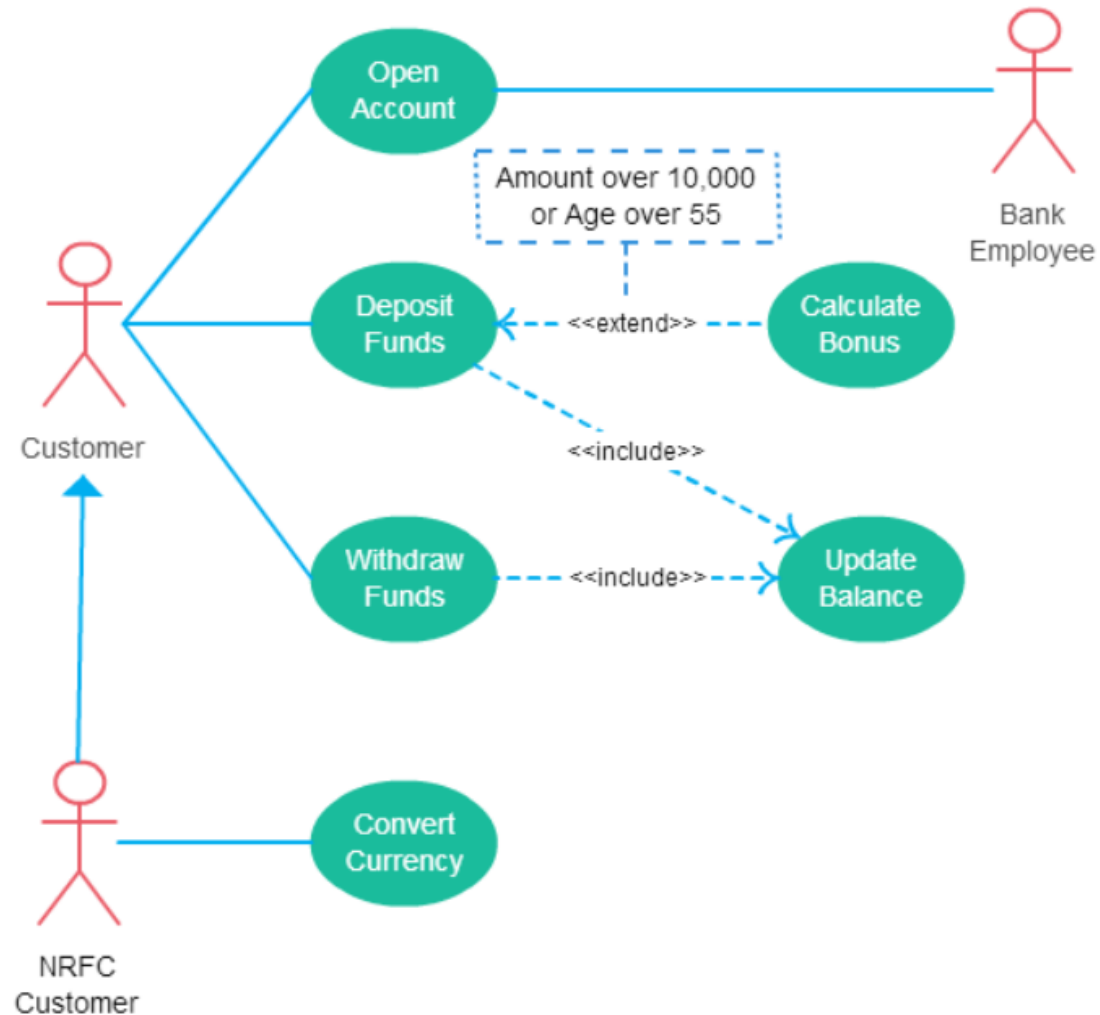
- **Extend** : Many people confuse the extend relationship in use cases. As the name implies it extends the base use case and adds more functionality to the system
- **NOTE** : The extending use case is dependent on the extended (base) use case, The extending use case is usually optional and can be triggered conditionally, The extended (base) use case must be meaningful on its own.



*Extend relationship in use case diagrams*

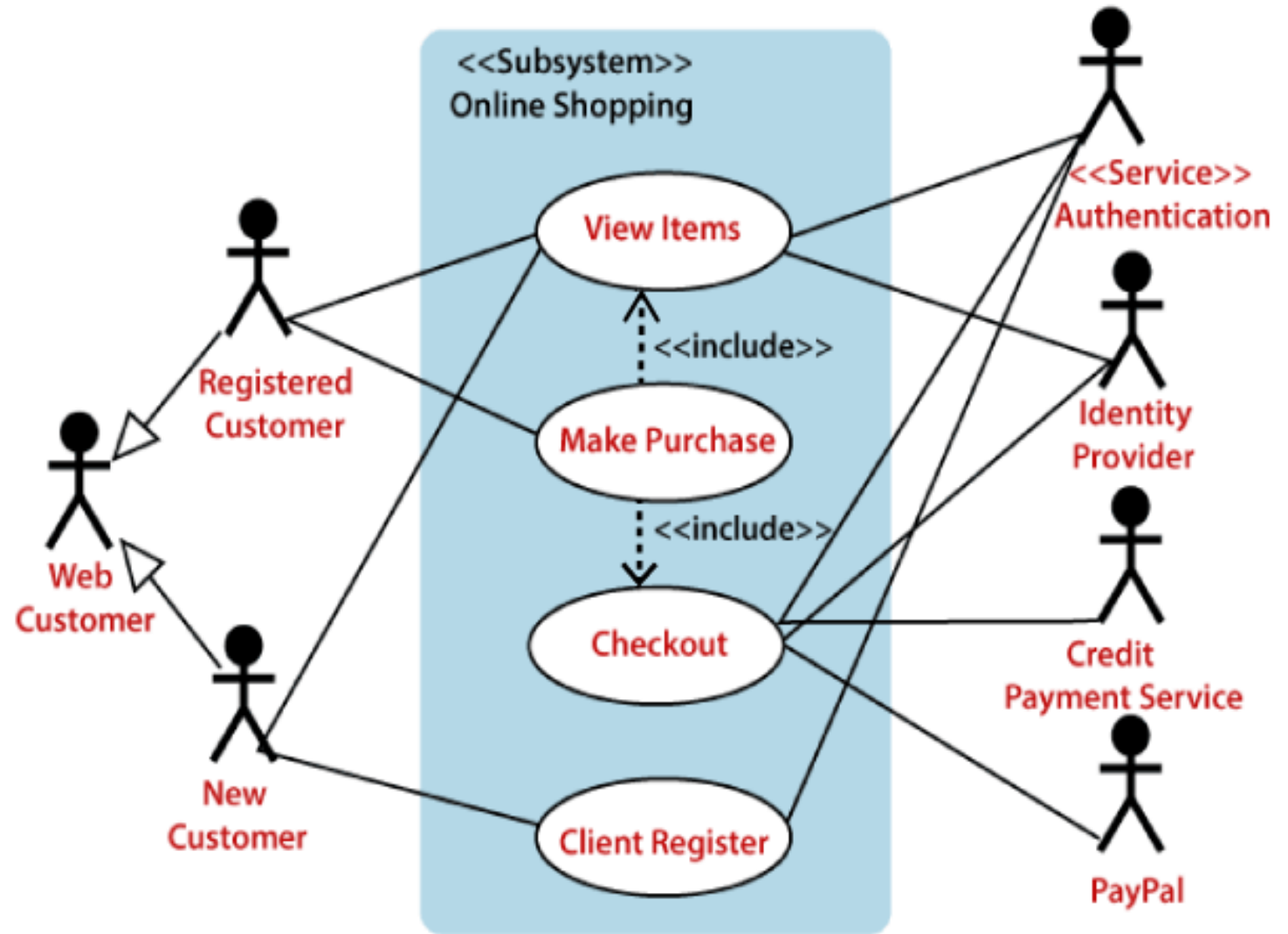
# Notations

- **Include:** 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.

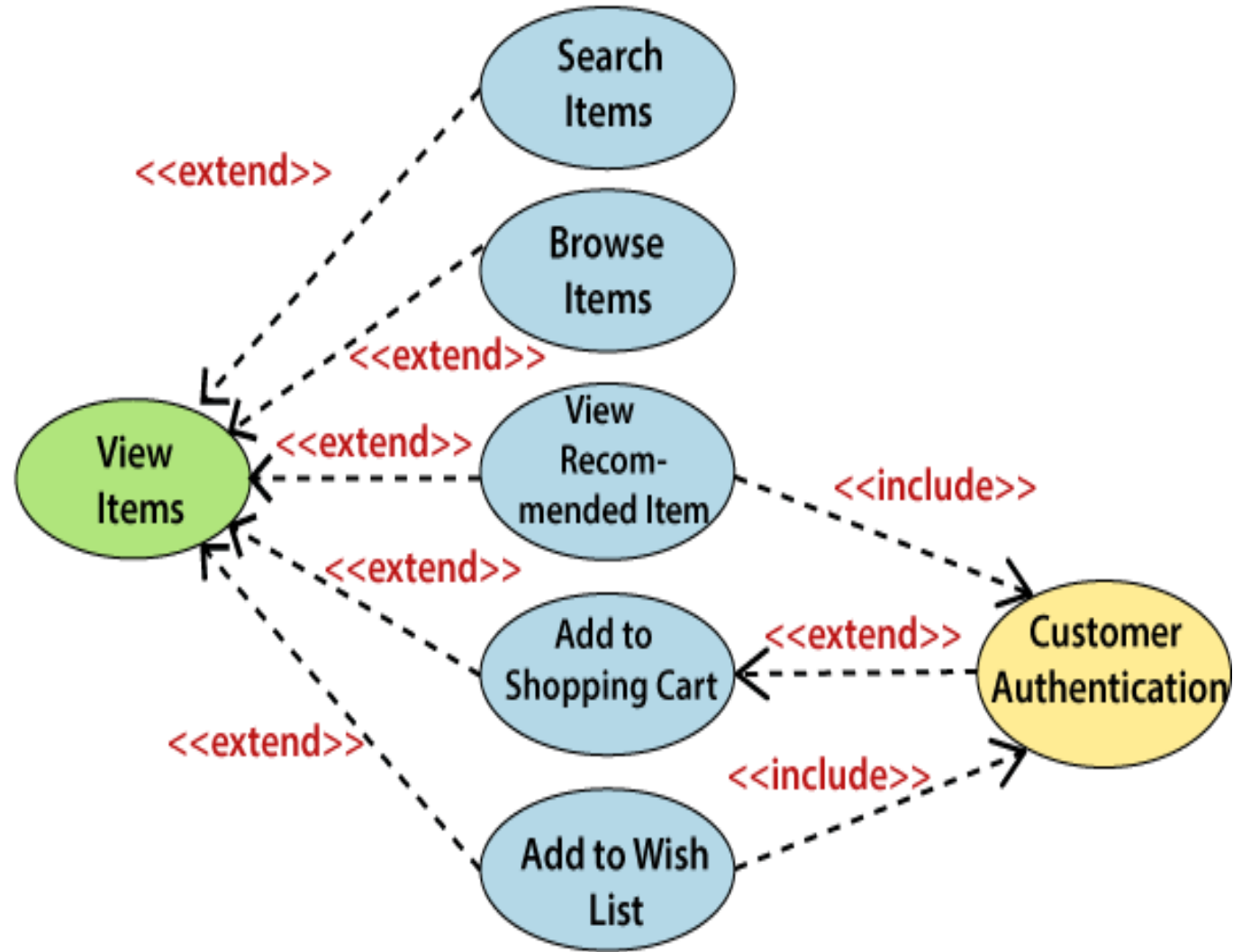


*Includes is usually used to model common behavior*

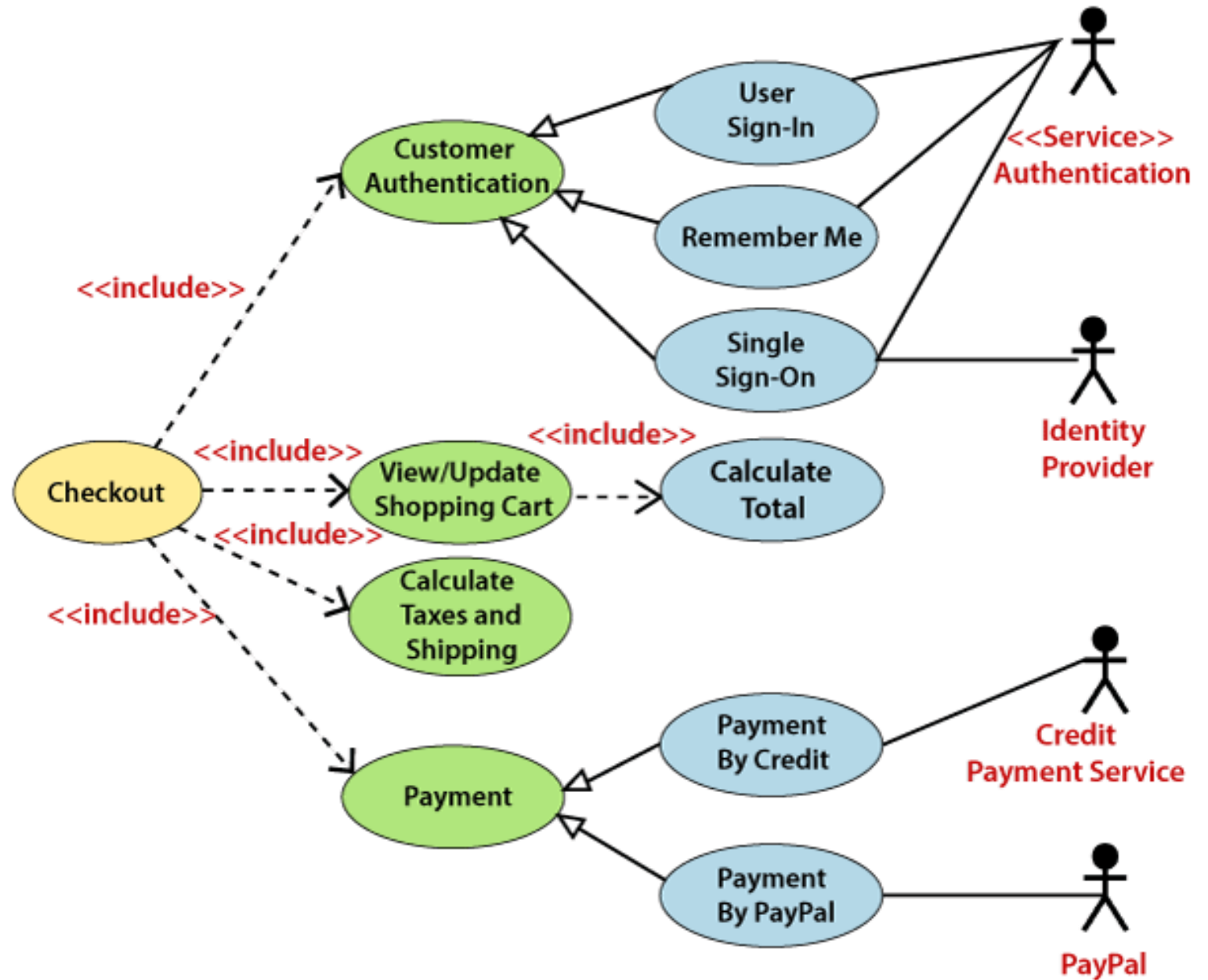
# Example



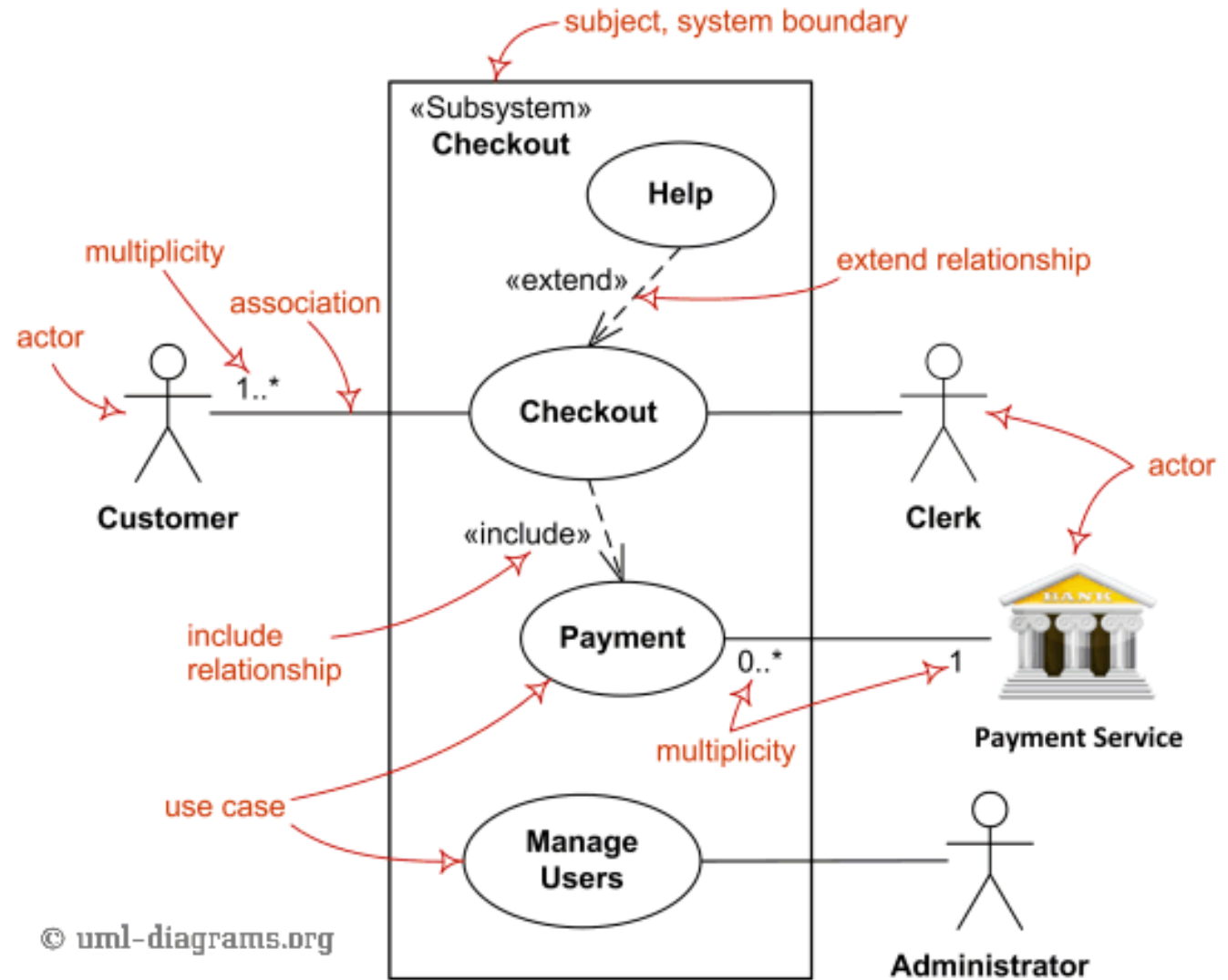
# Example



# Example



# Example



# Sequence diagram

- The sequence diagram represents the flow of messages in the system and is also termed as an event diagram
- It portrays the communication between any two lifelines as a time-ordered sequence of events, such that these lifelines took part at the run time.
- In UML, the lifeline is represented by a vertical bar, whereas the message flow is represented by a vertical dotted line that extends across the bottom of the page. It incorporates the iterations as well as branching.

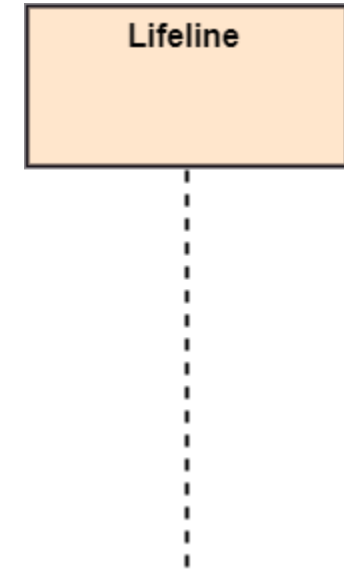
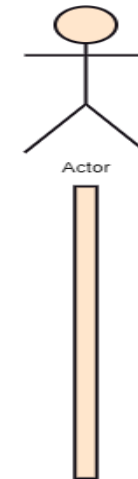


# Purpose of a Sequence Diagram

- To model high-level interaction among active objects within a system.
- To model interaction among objects inside a collaboration realizing a use case.
- It either models generic interactions or some certain instances of interaction.

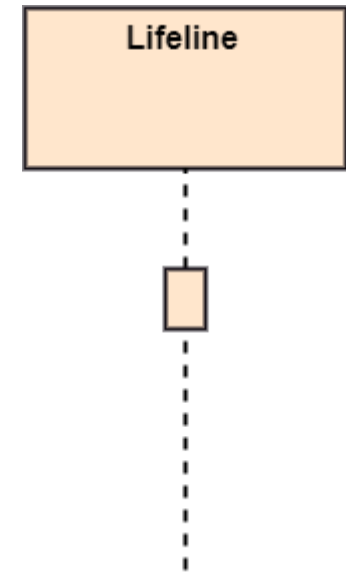
# Notations of a Sequence Diagram

- 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.. Several distinct roles can be played by an actor or vice versa.



# Notations of a Sequence Diagram

- **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.

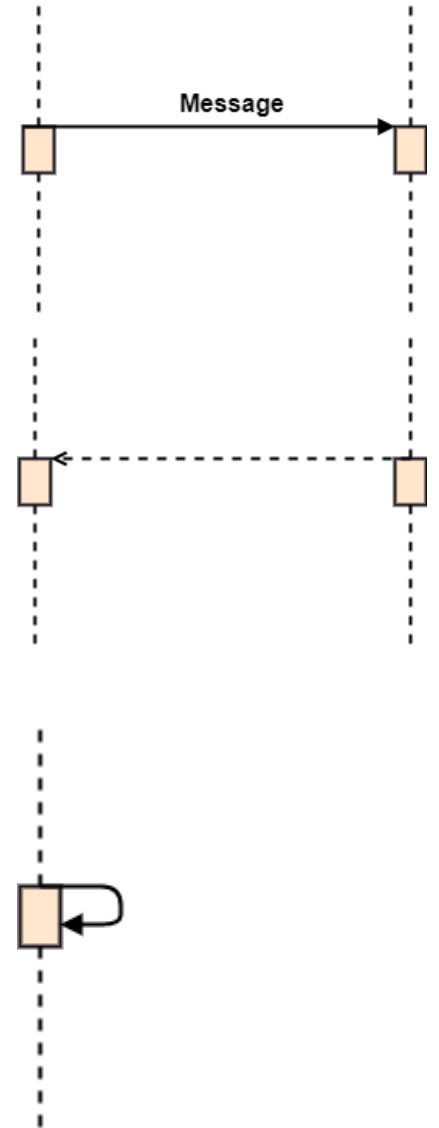


# Notations of a Sequence Diagram

- **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
  - Return Message
  - Self Message
  - Recursive Message
  - Create Message
  - Destroy Message
  - Duration Message:

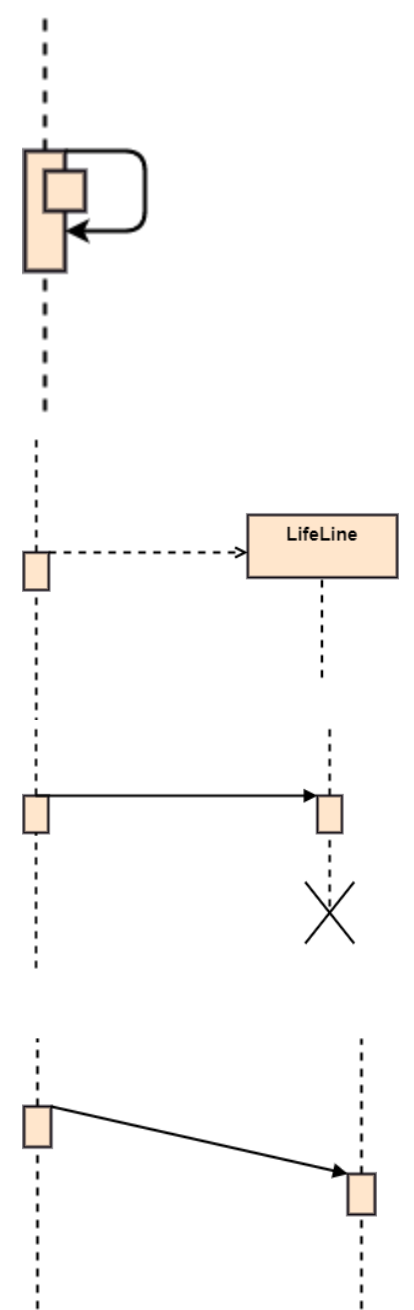
# Notations of a Sequence Diagram

- **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.



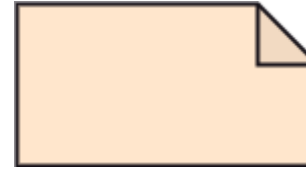
# Notations of a Sequence Diagram

- **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 expressed.
- **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.

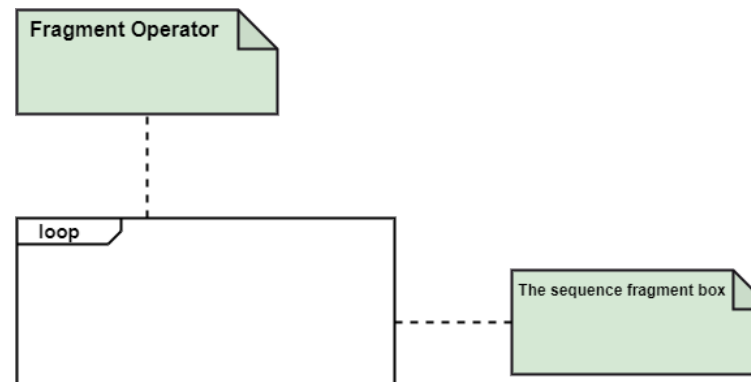


# Notations of a Sequence Diagram

- **Note :** A note (comment) gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.

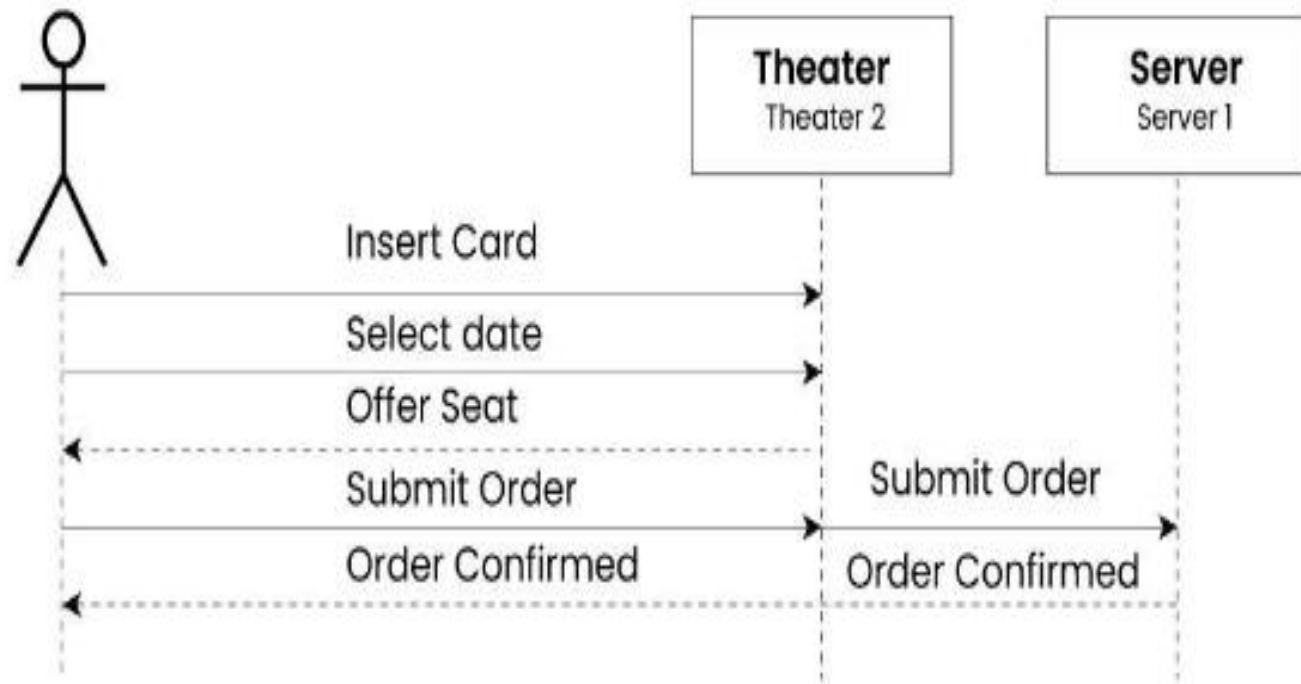


- **Sequence Fragments :**
  - Sequence fragments have been introduced by UML 2.0, which makes it quite easy for the creation and maintenance of an accurate sequence diagram.
  - It is represented by a box called a combined fragment, encloses a part of interaction inside a sequence diagram.
  - The type of fragment is shown by a fragment operator.



# Notations of a Sequence Diagram

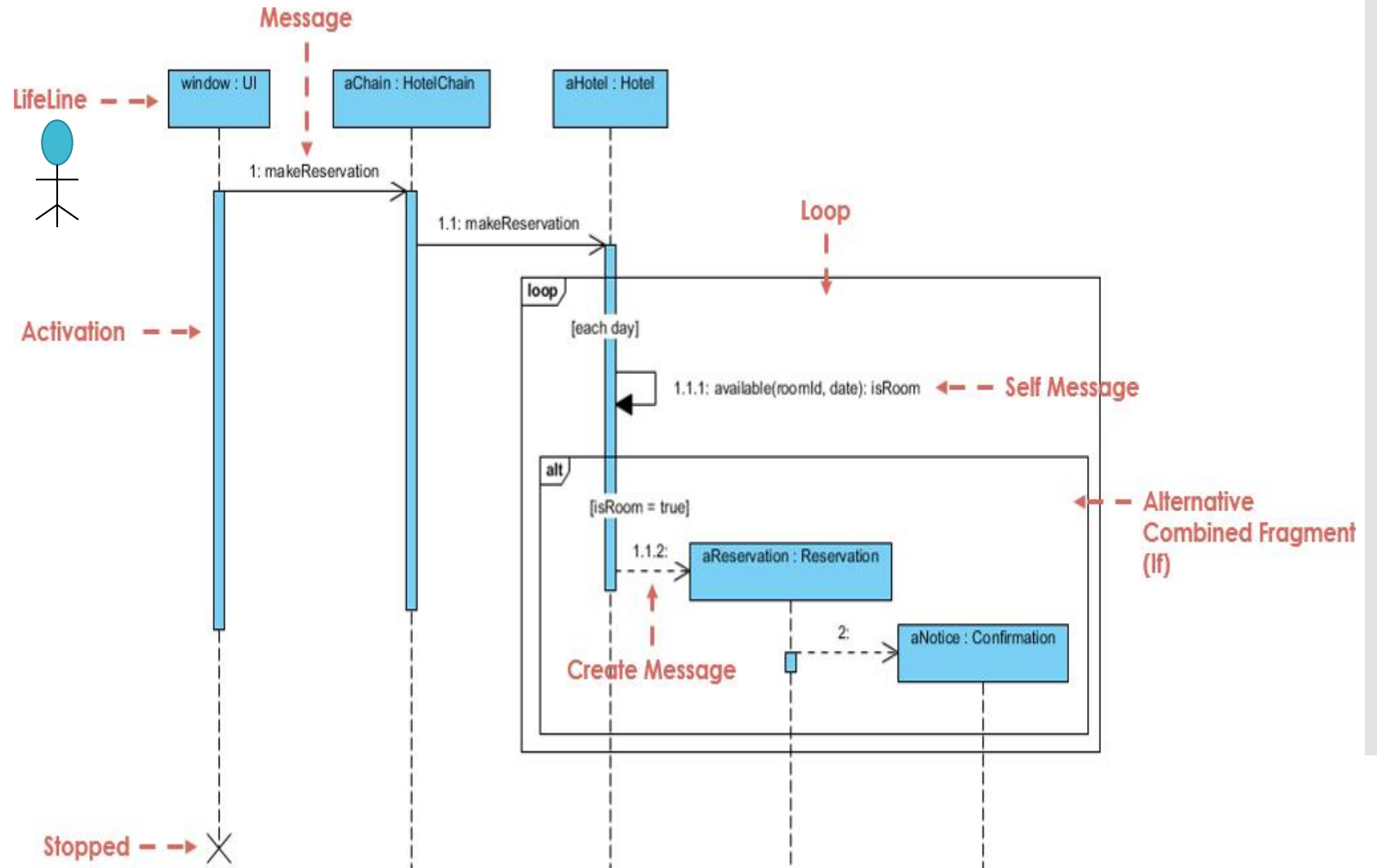
User interacting with seat reservation system





# Example

- Below is a sequence diagram for making a hotel reservation. The object initiating the sequence of messages is a Reservation window.



# Data Flow Diagrams

- A Data Flow Diagram (DFD) is a traditional visual representation of the information flows within a system.
- A neat and clear DFD can depict the right amount of the system requirement graphically.
- It can be manual, automated, or a combination of both.
- It shows how data enters and leaves the system, what changes the information, and where data is stored.
- The objective of a DFD is to show the scope and boundaries of a system as a whole.
- The DFD is also called as a data flow graph or bubble chart.


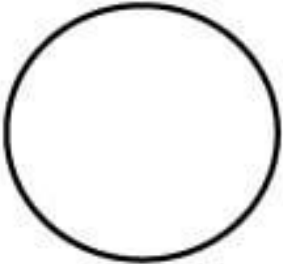

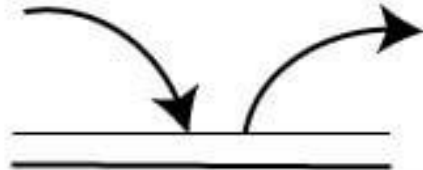
## observations about DFDs are essential

- All names should be unique. This makes it easier to refer to elements in the DFD.
- Remember that DFD is not a flow chart. Arrows is a flow chart that represents the order of events; arrows in DFD represents flowing data. A DFD does not involve any order of events.
- Suppress logical decisions. If we ever have the urge to draw a diamond-shaped box in a DFD, suppress that urge! A diamond-shaped box is used in flow charts to represents decision points with multiple exists paths of which the only one is taken. This implies an ordering of events, which makes no sense in a DFD.
- Do not become bogged down with details. Defer error conditions and error handling until the end of the analysis.

# Difference Between Flowchart and DFD

Flow Chart	Data Flow Diagram (DFD)
❖ The main objective is to represent the flow of control in the program.	❖ The main objective is to represent the processes and data flow between them.
❖ It has only a single type of arrow is used to show the control flow in the flow chart.	❖ It defines the flow and process of data input, data output, and storing data.
❖ It is the view of the system at a lower level.	❖ It is the view of the system at a high level.
❖ It deals with the physical aspect of the action.	❖ It deals with the logical aspect of the action.
❖ It shows how to make the system function.	❖ It defines the functionality of the system.
❖ It is not very suitable for a complex system.	❖ It is used for complex systems.

# Standard symbols for DFD

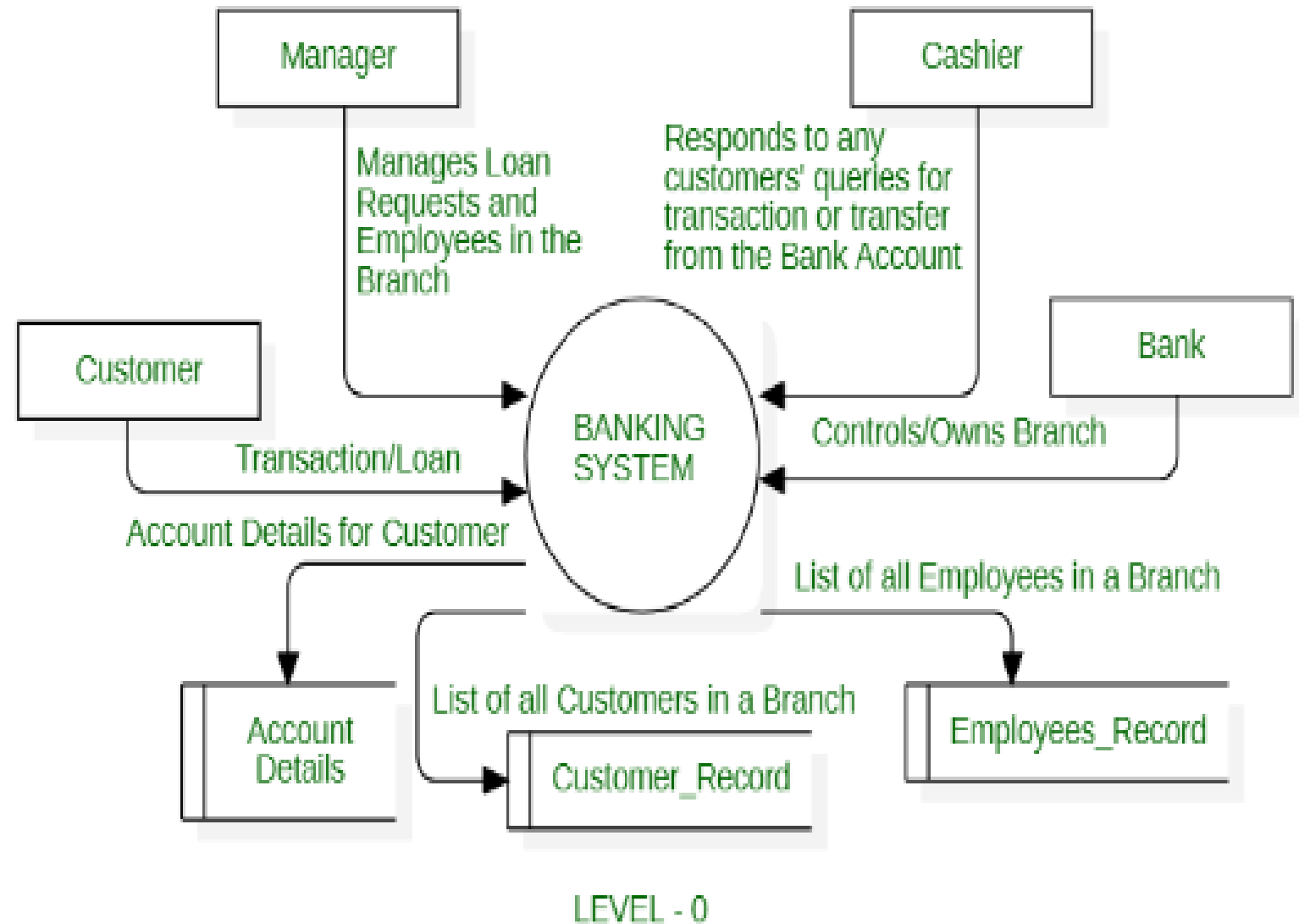
Symbol	Name	Function
	Data flow	Used to Connect Processes to each other, to sources or Sinks; the arrow head indicates direction of data flow.
	Process	Performs Some transformation of Input data to yield output data.
	Source of Sink (External Entity)	A Source of System inputs or Sink of System outputs.
	Data Store	A repository of data; the arrow heads indicate net inputs and net outputs to store.

Symbols for Data Flow Diagrams

# Levels in Data Flow Diagrams (DFD)

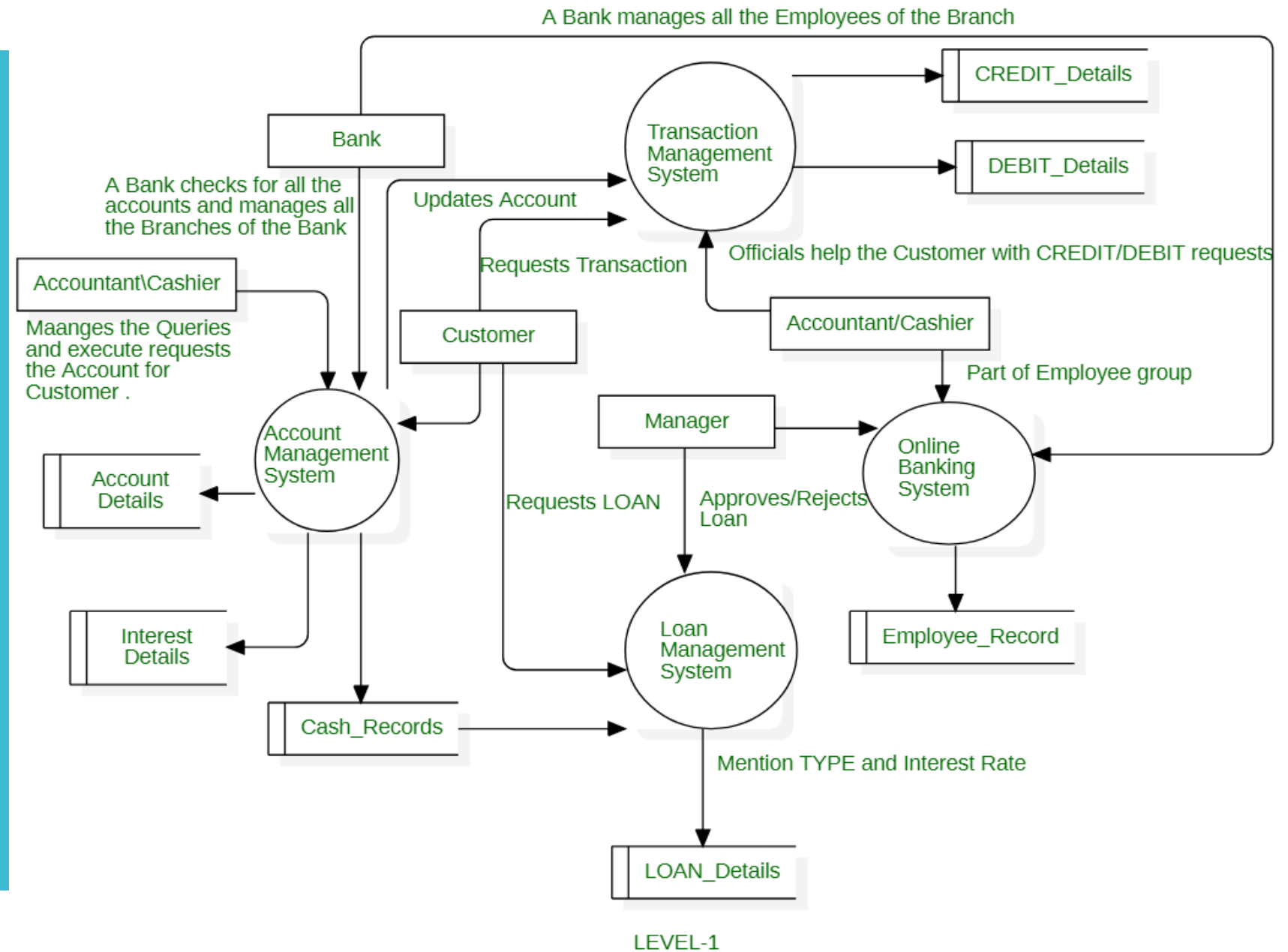
- The DFD may be used to perform a system or software at any level of abstraction.
- Infact, DFDs may be partitioned into levels that represent increasing information flow and functional detail.
- Levels in DFD are numbered 0, 1, 2 or beyond. Here, we will see primarily three levels in the data flow diagram, which are: 0-level DFD, 1-level DFD, and 2-level DFD

# Example- Online Banking System-Level 0



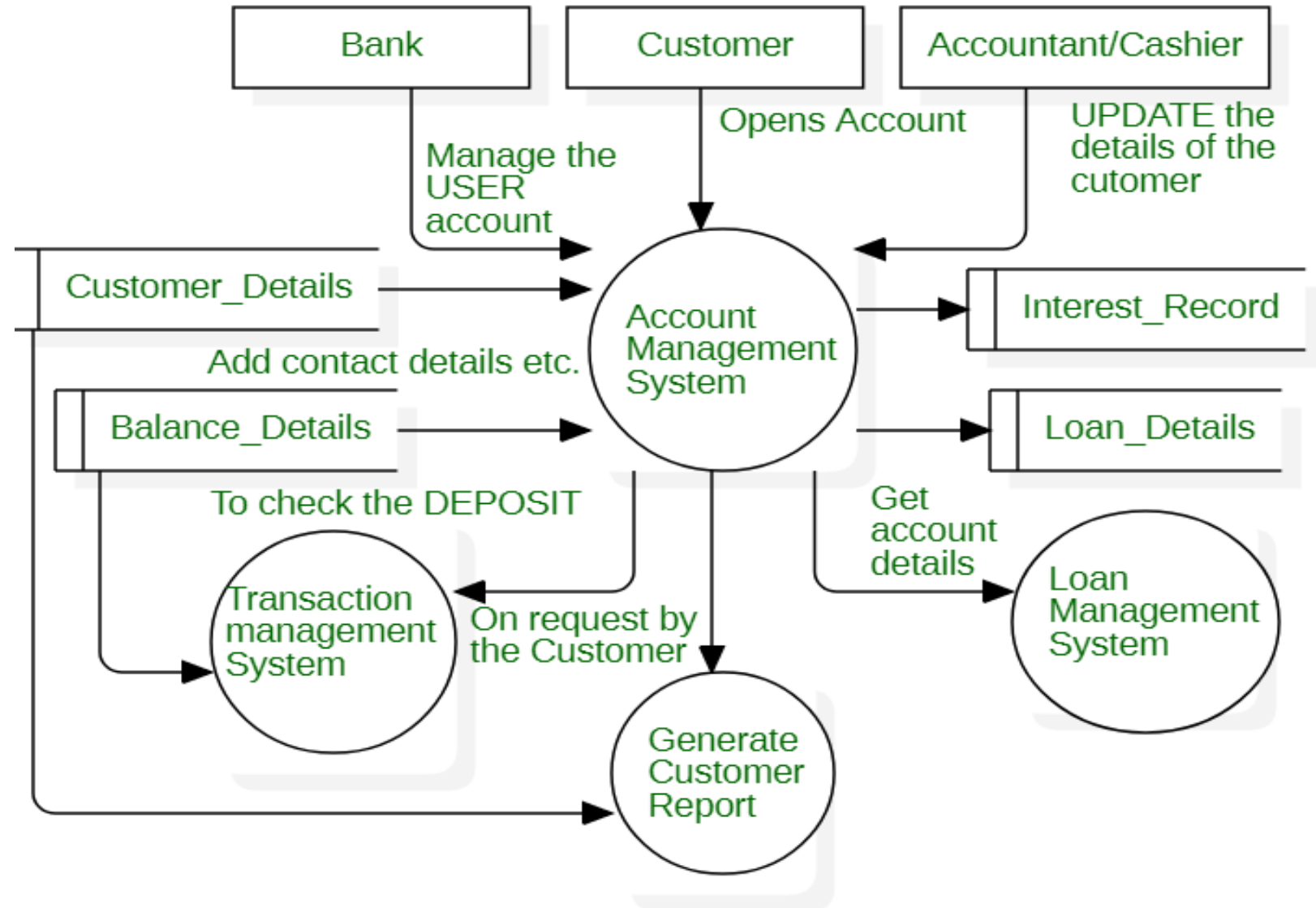
*Level-0 DFD — Online Banking System*

# Example- Online Banking System-Level 1



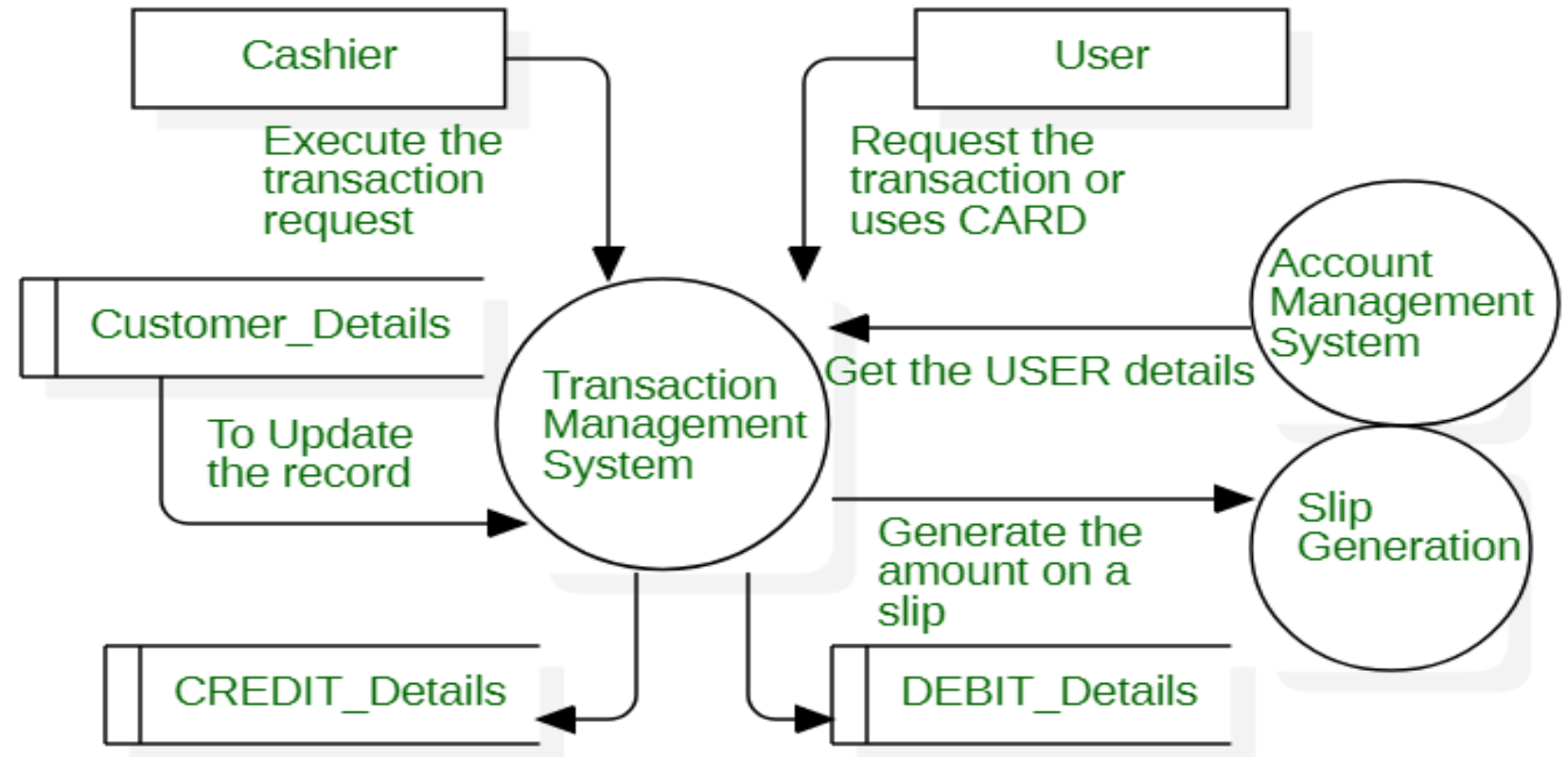


# Example- Online Banking System-Level 2



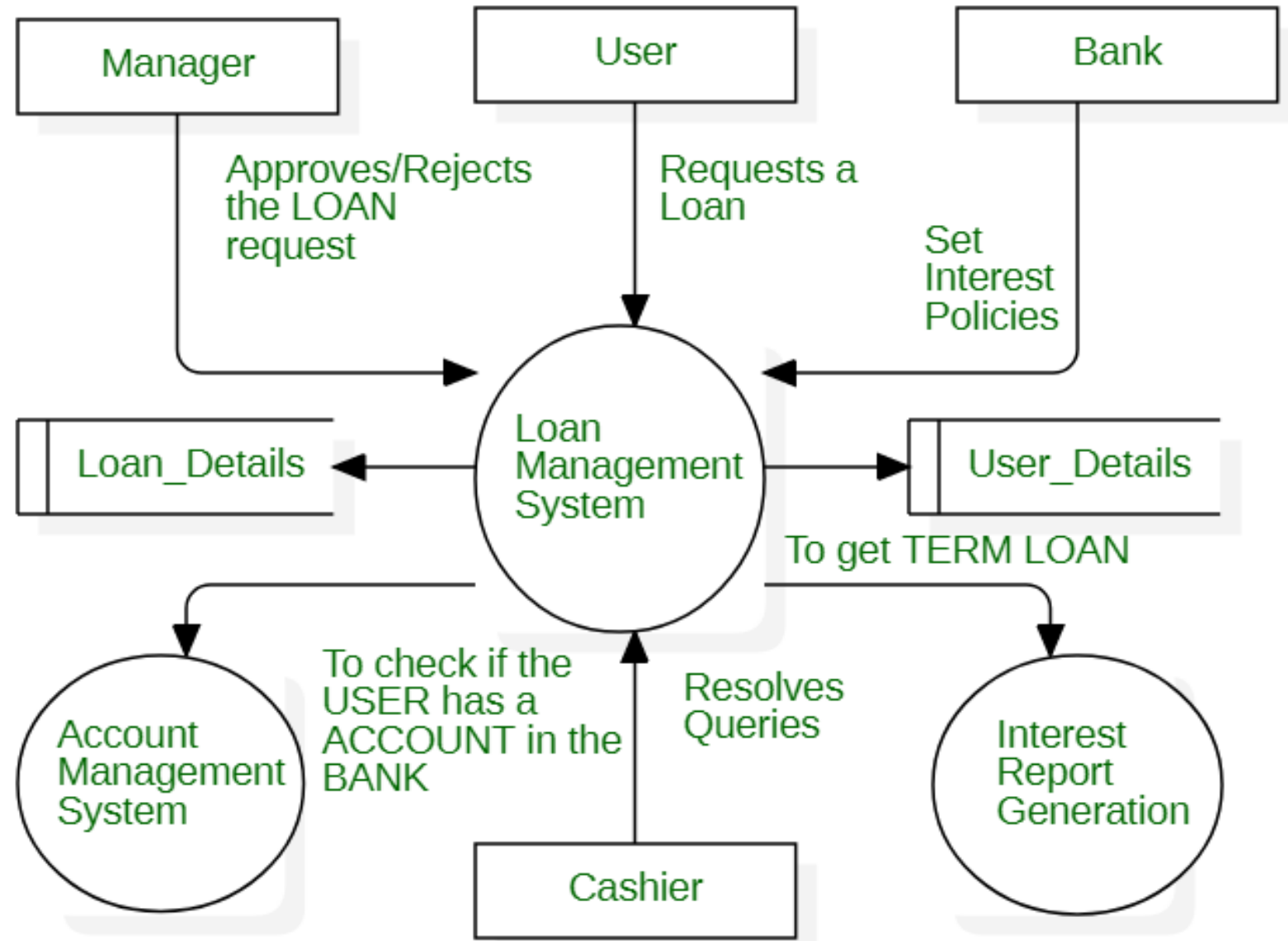
1. Account Management System

# Example- Online Banking System-Level 2



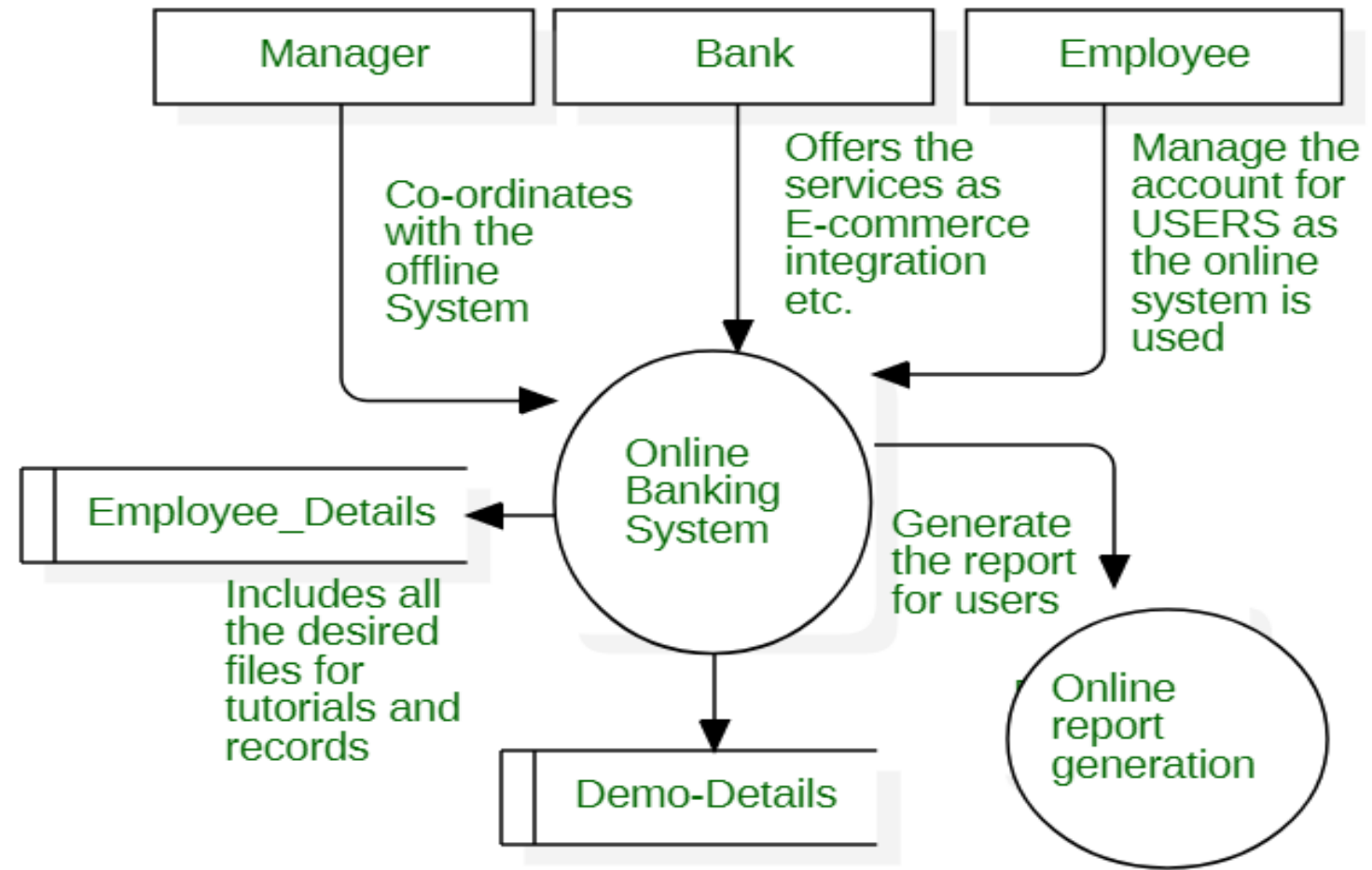
## 2. Transaction Management System

# Example- Online Banking System-Level 2



3. Loan Management System

# Example- Online Banking System-Level 2



4 . Online Banking System

# Rules for DFD

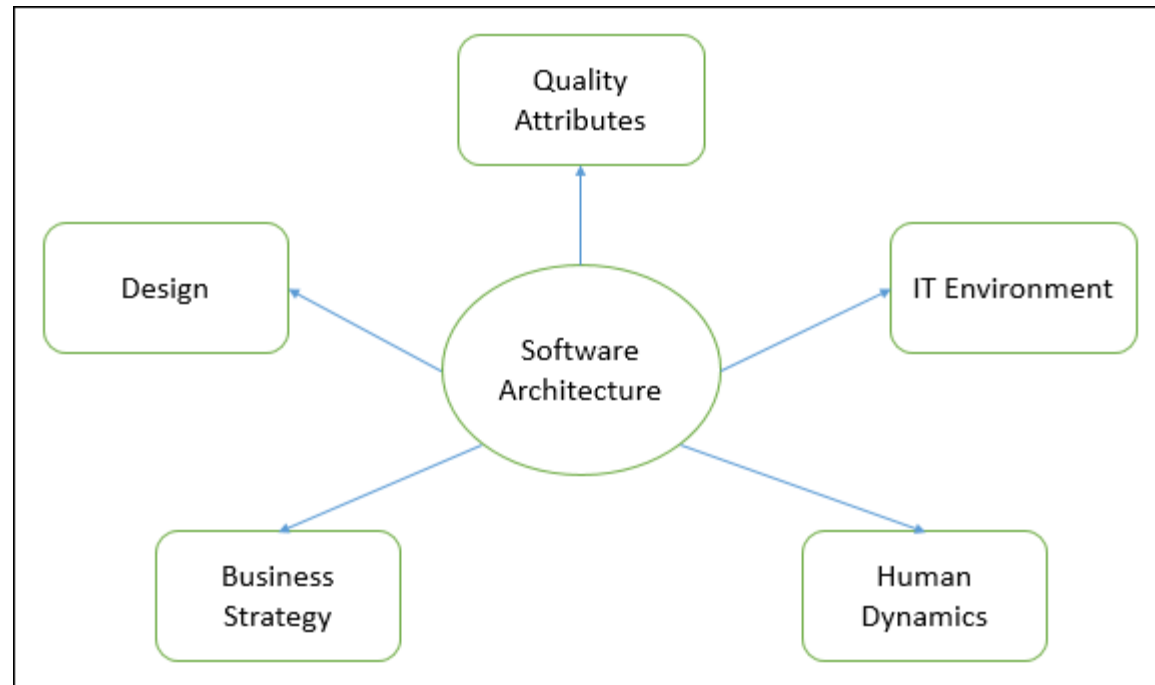
- **Data can not flow between two entities. –**  
Data flow must be from entity to a process or a process to an entity. There can be multiple data flows between one entity and a process.
- **Data can not flow between two data stores**  
Data flow must be from data store to a process or a process to an data store. Data flow can occur from one data store to many process.
- **Data can not flow directly from an entity to data store –**  
Data Flow from entity must be processed by a process before going to data store and vice versa.
- **A process must have at least one input data flow and one output data flow –**  
Every process must have input data flow to process the data and an output data flow for the processed data.

# Rules for DFD

- **A data store must have at least one input data flow and one output data flow –**  
Every data store must have input data flow to store the data and an output data flow for the retrieved data.
- Two data flows can not cross each other.
- All the process in the system must be linked to minimum one data store or any other process.

# Software Architecture

- The architecture of a system describes its major components, their relationships (structures), and how they interact with each other.
- Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



# Continue..

- We can separate Software Architecture and Design into two distinct phases:
- **Software Architecture** and **Software Design**.
- In Architecture, **nonfunctional** decisions are cast and separated by the functional requirements.
- In Design, **functional** requirements are accomplished.



# Goals of Architecture

- Expose the structure of the system, but hide its implementation details.
- Realize all the use-cases and scenarios.
- Try to address the requirements of various stakeholders.
- Handle both functional and quality requirements.
- Reduce the goal of ownership and improve the organization's market position.
- Improve quality and functionality offered by the system.
- Improve external confidence in either the organization or system.

# Limitations

- Lack of tools and standardized ways to represent architecture.
- Lack of analysis methods to predict whether architecture will result in an implementation that meets the requirements.
- Lack of awareness of the importance of architectural design to software development.
- Lack of understanding of the role of software architect and poor communication among stakeholders.
- Lack of understanding of the design process, design experience and evaluation of design

# Different Software Architecture Patterns

- Layered Pattern
- Client-Server Pattern
- Event-Driven Pattern
- Microkernel Pattern
- Microservices Pattern

# Layered Pattern

- As the name suggests, components(code) in this pattern are separated into layers of subtasks and they are arranged one above another.
- Each layer has unique tasks to do and all the layers are independent of one another.
- Since each layer is independent, one can modify the code inside a layer without affecting others.
- It is the most commonly used pattern for designing the majority of software. This layer is also known as 'N-tier architecture'.
- Basically, this pattern has 4 layers.
  - **Presentation layer** (The user interface layer where we see and enter data into an application.)
  - **Business layer** (this layer is responsible for executing business logic as per the request.)
  - **Application layer** (this layer acts as a medium for communication between the 'presentation layer' and 'data layer'.
  - **Data layer** (this layer has a database for managing data.)
- E-commerce web applications development like **Amazon**.

# Client-Server Pattern

- The client-server pattern has two major entities. They are a server and multiple clients.
- Here the server has resources(data, files or services) and a client requests the server for a particular resource. Then the server processes the request and responds back accordingly.
- Examples of software developed in this pattern:
  - Email.
  - WWW.
  - File sharing apps.
  - Banking, etc....

# Event-Driven Pattern

- Event-Driven Architecture is an agile approach in which services (operations) of the software are triggered by events.
- Well, what does an event mean?
- When a user takes action in the application built using the EDA approach, a state change happens and a reaction is generated that is called an event.
- E.g. A new user fills the signup form and clicks the signup button on Facebook and then a FB account is created for him, which is an event.
- Ideal for:
  - Building websites with JavaScript and e-commerce websites in general

# Microkernel Pattern

- Microkernel pattern has two major components. They are a core system and plug-in modules.
- The core system handles the fundamental and minimal operations of the application.
- The plug-in modules handle the extended functionalities (like extra features) and customized processing.
- Microkernel pattern is ideal for:
  - Product-based applications and scheduling applications.

# Microservices Pattern

- The collection of small services that are combined to form the actual application is the concept of microservices pattern.
- Instead of building a bigger application, small programs are built for every service (function) of an application independently.
- And those small programs are bundled together to be a full-fledged application.
- Example Netflix is one of the most popular examples of software built-in microservices architecture. This pattern is most suitable for websites and web apps having small components.



# Design Patterns

- The term design pattern is used in object-oriented terminology to perform the tasks such as defining the objects, classes, interfaces hierarchy and factoring them into classes with relationships.
- Once all these steps are considered as a pattern they can reuse them by applying them to several common problems.
- In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design.
- A design pattern isn't a finished design that can be transformed directly into code.
- It is a description or template for how to solve a problem that can be used in many different situations.

# Uses of Design Patterns

- Design patterns can speed up the development process by providing tested, proven development paradigms.
- Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.
- Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.
- Patterns allow developers to communicate using well-known, well understood names for software interactions.
- Common design patterns can be improved over time, making them more robust than ad-hoc designs.

# Catalog of Design Patterns

- The Design Patterns are organized into a form of a catalog.
- These Design Patterns collectively assist in software engineering by finding objects, specifying objects implementations, objects interfaces, determining objects granularity, implementing reuse mechanisms, etc.

Some of the patterns with their names and intents are as follows,

- Adapter
  - Match interfaces of different classes
- Bridge
  - Separates an object's interface from its implementation
- Composite
  - A tree structure of simple and composite objects
- Decorator
  - Add responsibilities to objects dynamically
- Facade
  - A single class that represents an entire subsystem
- Flyweight
  - A fine-grained instance used for efficient sharing
- Private Class Data
  - Restricts accessor/mutator access
- Proxy
  - An object representing another object