

CST205: OBJECT ORIENTED DESIGN AND PROGRAMMING USING JAVA

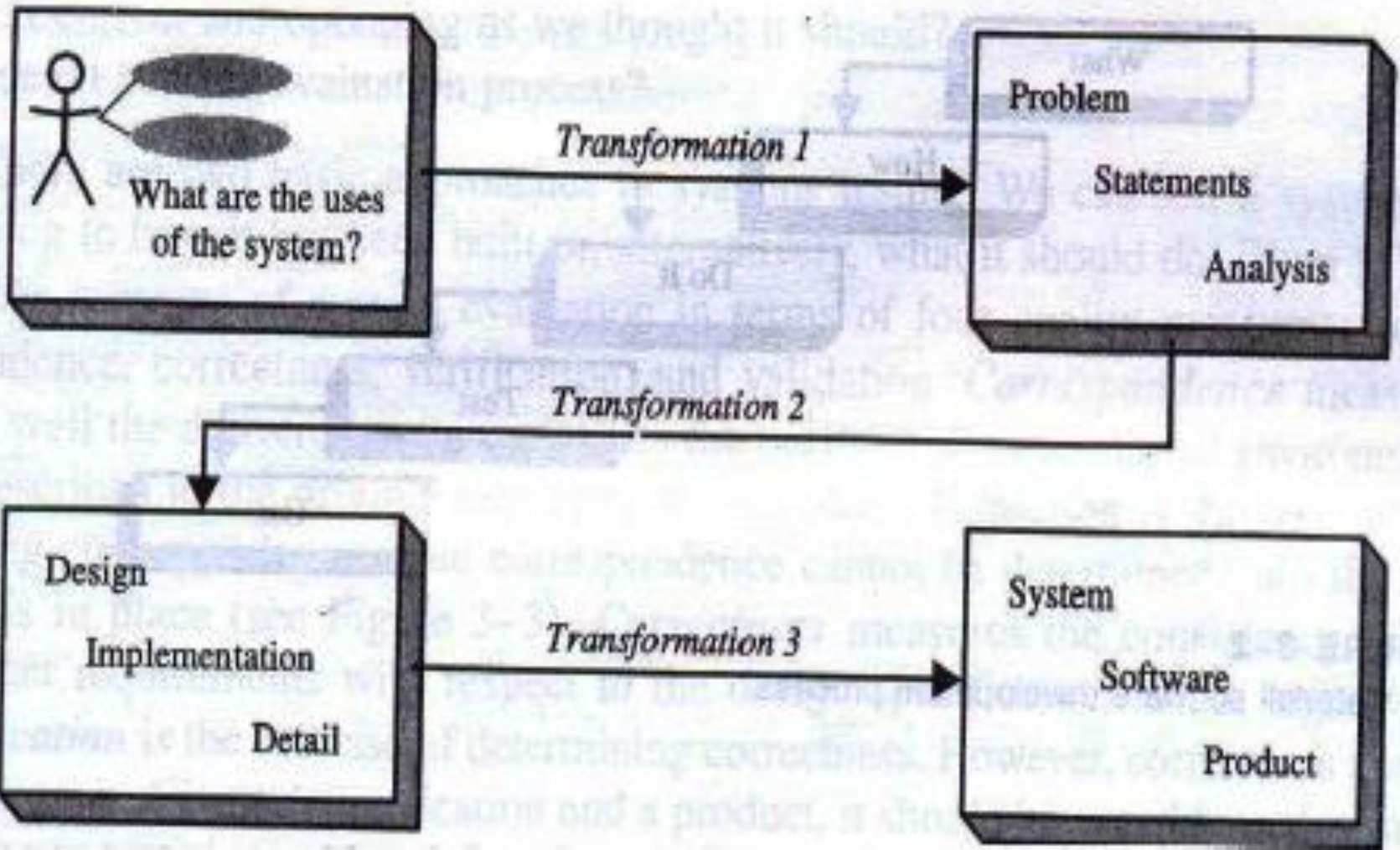
B.Tech. CSE

Semester III

Viswajyothi College of Engineering and Technology

Software development process

- Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- Software development process consists of analysis, design, implementation, testing, and refinement.
- At the first level the focus is on deciding which modules are needed for the system on the basis of SRS (Software Requirement Specification) and how the modules should be interconnected.
- Generally software development can be seen as a series of transformations, where the output of one transformation becomes the input of the subsequent transformation.



Transformation1(Analysis):

- Translates the user's needs into system requirements and responsibilities.
- The way the system is used can provide insight into the user's requirements.
- For example: in library management system, identify who are the users and what are their requirements.

Transformation 2 (Design):

- Begins with a problem statement and ends with a detailed design that can be transformed into a operational system.
- This transformation includes the bulk of the software development activity, including the definition of how to build the software, its development, and its testing.
- It also includes design descriptions, the programs and the testing material.

Transformation 3(Implementation):

- Refines the detailed design into the system deployment that will satisfy the user's needs.
- With inputs from system design, the system is first developed in small programs called units, which are integrated in the next phase.
- Finally the software product is embedded within its operational environment.

Object-Oriented Systems Development:

- The object oriented software development life cycle(SDLC) consists of three macro processes:
 - Object-oriented analysis
 - Object-oriented design
 - Object-oriented implementation

Object-Oriented Design

- The goal of *object-oriented design (OOD)* is to design the classes identified during the analysis phase and the user interface.
- During this phase, additional objects and classes that supports implementation of the requirements are defined.

Following are the guidelines to use in the object-oriented design:

- Reuse, rather than build, a new class. Know the existing classes.
- Design a large number of simple classes, rather than a small number of complex classes.
- Design methods.
- Critique what have been proposed. If possible, go back and refine the classes.

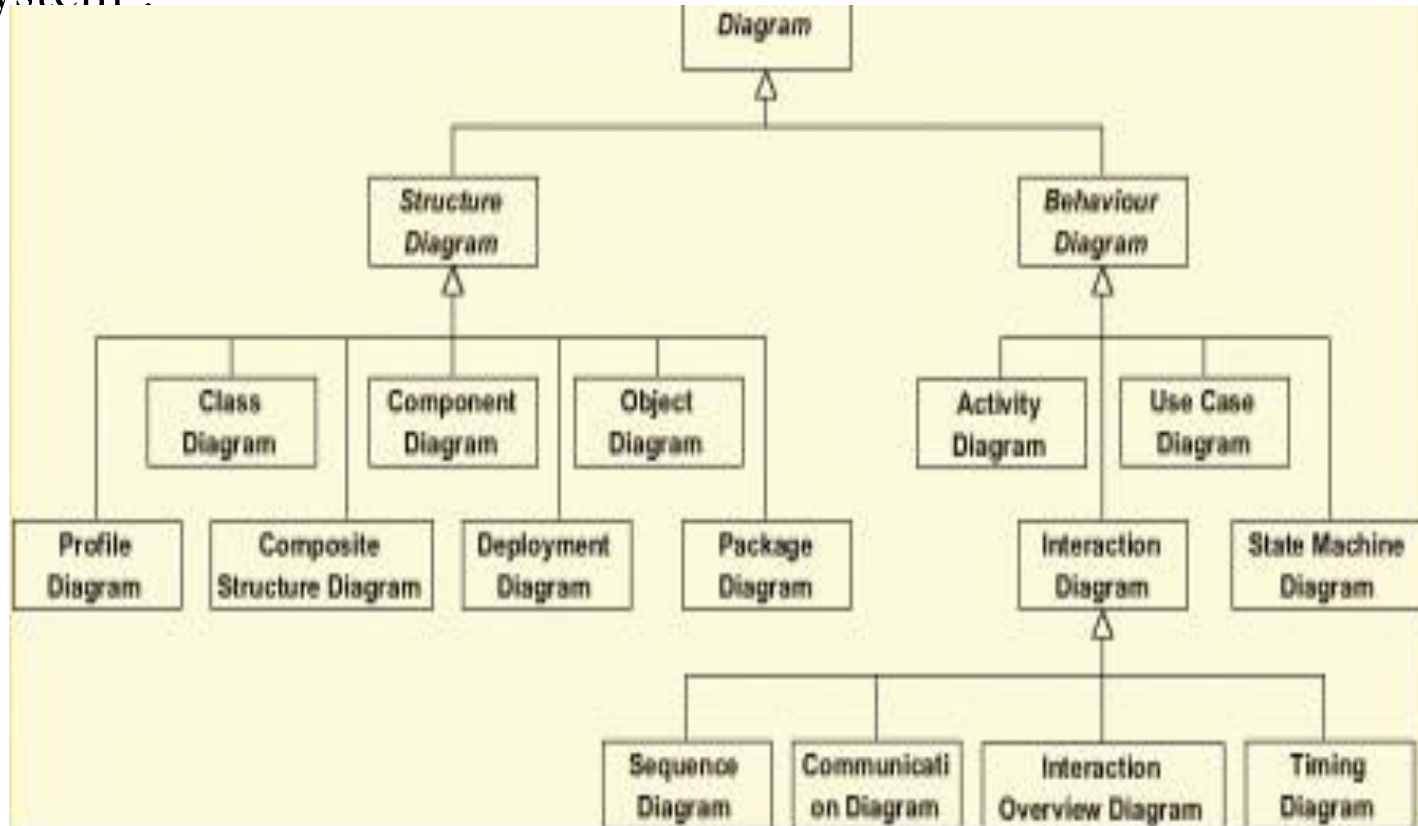
Unified Modeling Language(UML)

- Modeling is building a model for a software system prior to its construction.
- It is used to describe a system at a high level of abstraction that includes user requirements and specifications.
- UML is an industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- The UML uses mostly graphical notations to express the OO analysis and design of software projects.
- Simplifies the complex process of software design.

Why UML for Modeling?

- Use graphical notation to communicate more clearly than natural language (imprecise) and code(too detailed).
- Help acquire an overall view of a system.
- UML is not dependent on any one language or technology.
- Encourage the use of object oriented concepts.
- Integrate best practices and methodologies and support higher-level development concepts

- UML is linked with object oriented design and analysis .
- Diagrams in UML can be broadly classified as:
 - **Structure Diagrams:** Capture static aspects or structure of a system
 - **Behavior Diagrams:** Capture dynamic aspects or behavior of the system .



Types of UML Diagrams

- Use Case Diagram
- Class Diagram
- Interaction Diagram
- Activity Diagram
- State chart Diagram

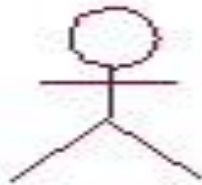
Use Case Diagram

- Depicts the functionality of the system.
- Used for describing a set of user scenarios and is mainly used for capturing user requirements.
- Use cases represent specific flows of events in the system.
- A use-case diagram is a graph of actors, a set of use cases enclosed by a system boundary, communication/associations between the actors and the use cases, and generalization among the use cases.

Use Case Diagram (core components)

Actors: A role that a user plays with respect to the system. i.e. they are the entities that interact with the system. An actor may be people, computer hardware, other systems, etc.

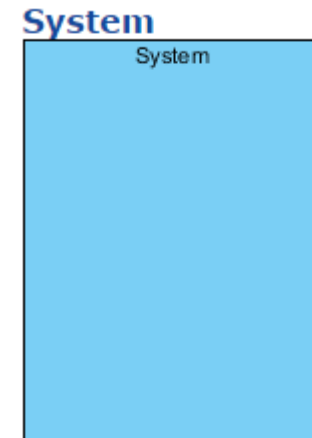
Use case: A set of scenarios that describes an interaction between a user and a system. It is shown as an ellipse.



Actor



Use Case



UML system

System boundary: rectangle diagram representing the boundary between the actors and the system. The use cases of the system are placed inside the system shape, while the actor who interact with the system are put outside the system.

Purposes of use case diagram

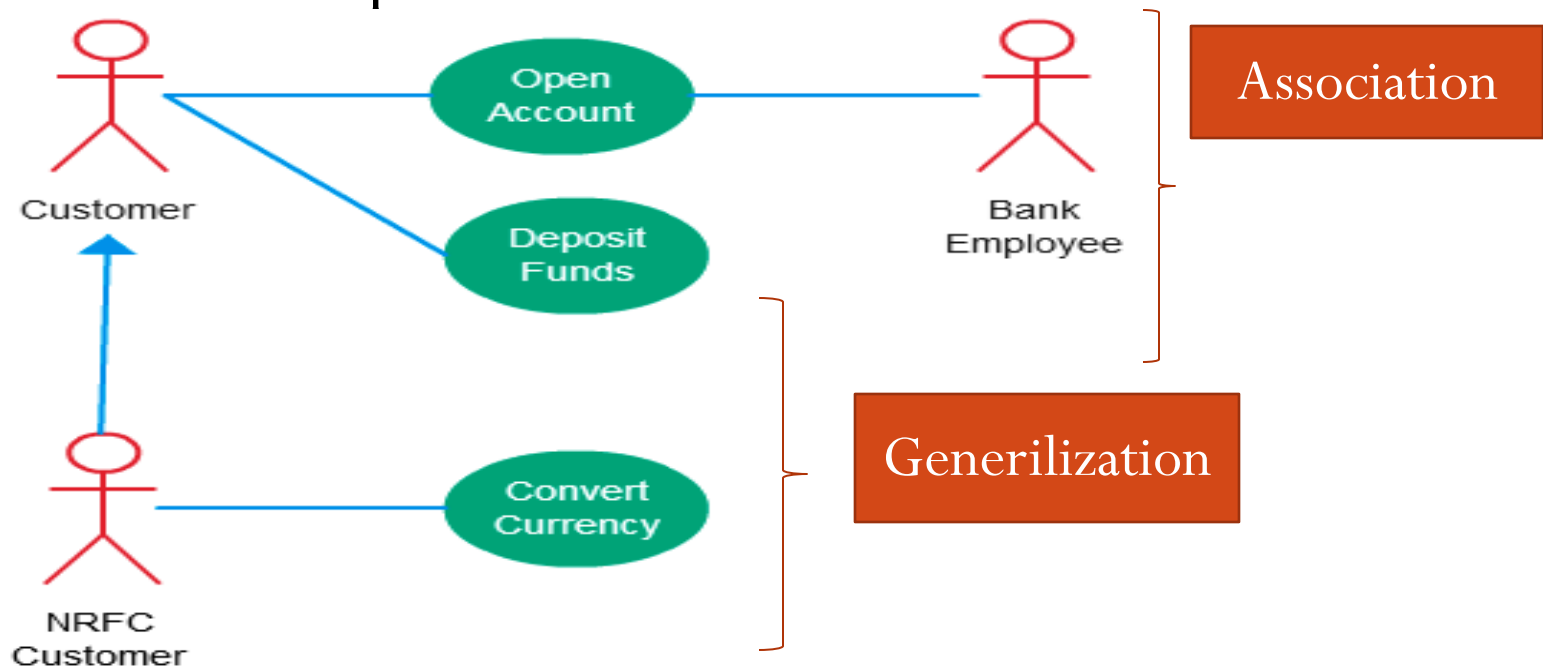
- Used to gather the requirements of a system.
- Used to get an outside view of a system.
- Identify the external and internal factors influencing the system.
- Show the interaction among the requirements and actors

Relationships shown in an use-case diagram

There can be 5 relationship types in a use case diagram.

1.Association/communication: communication between an actor and a use case; Represented by a solid line.

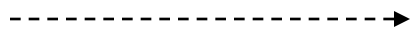
2.Generalization of actor: Generalization of an actor means that one actor can inherit the role of an other actor. The descendant inherits all the use cases of the ancestor. The descendant have one or more use cases that are specific to that role.



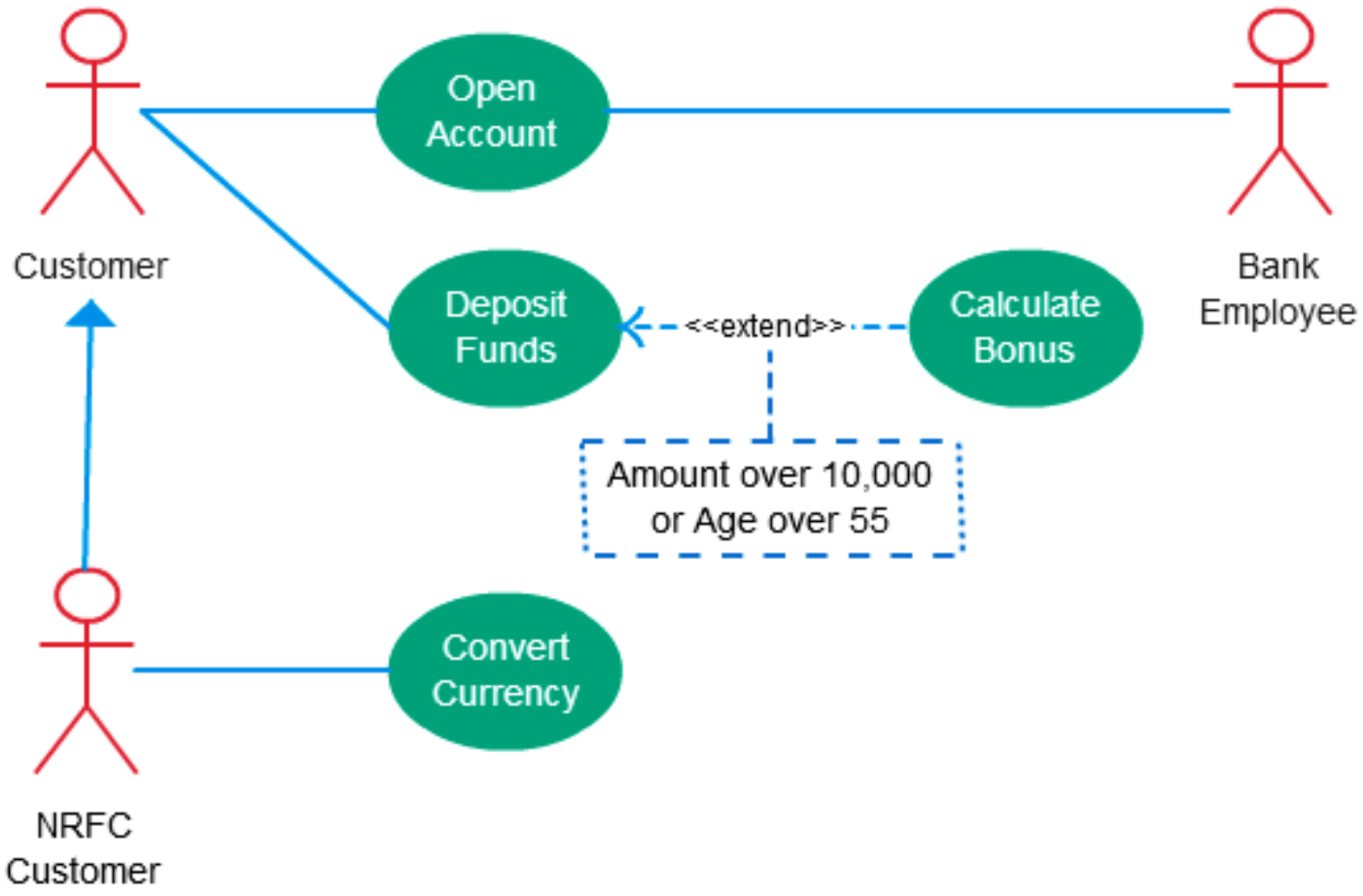
A generalized actor in an use case diagram

- **3.Extend**: a dotted line labeled <<extend>> with an arrow toward the base case. The extending use case may add behavior to the base use case. The base class declares “extension points”.

<<extend>>

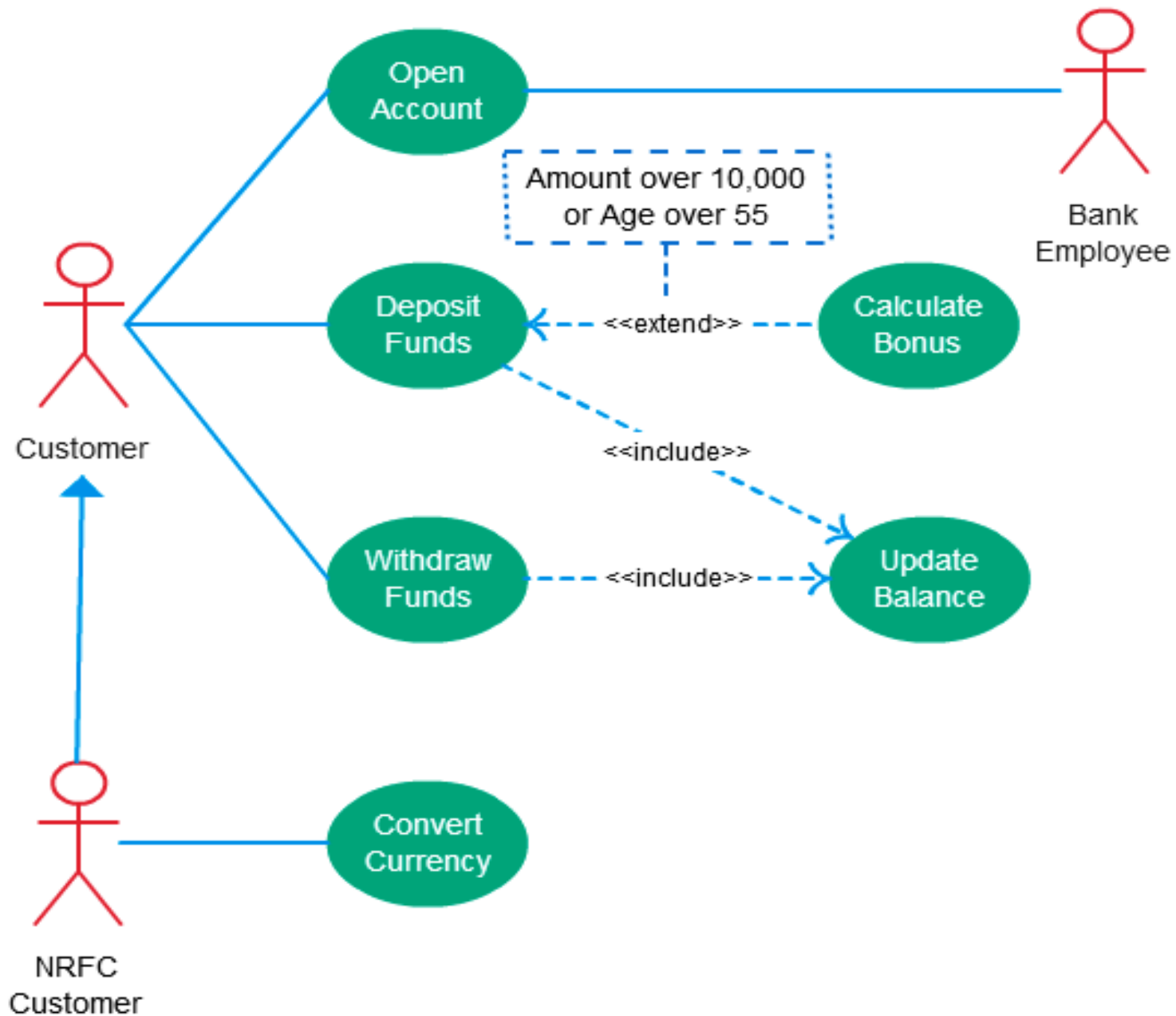


- **The extending use case is dependent on the extended (base) use case.** In the below diagram the “Calculate Bonus” use case doesn’t make much sense without the “Deposit Funds” use case.
- **The extending use case is usually optional** and can be triggered conditionally. In the diagram the extending 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.

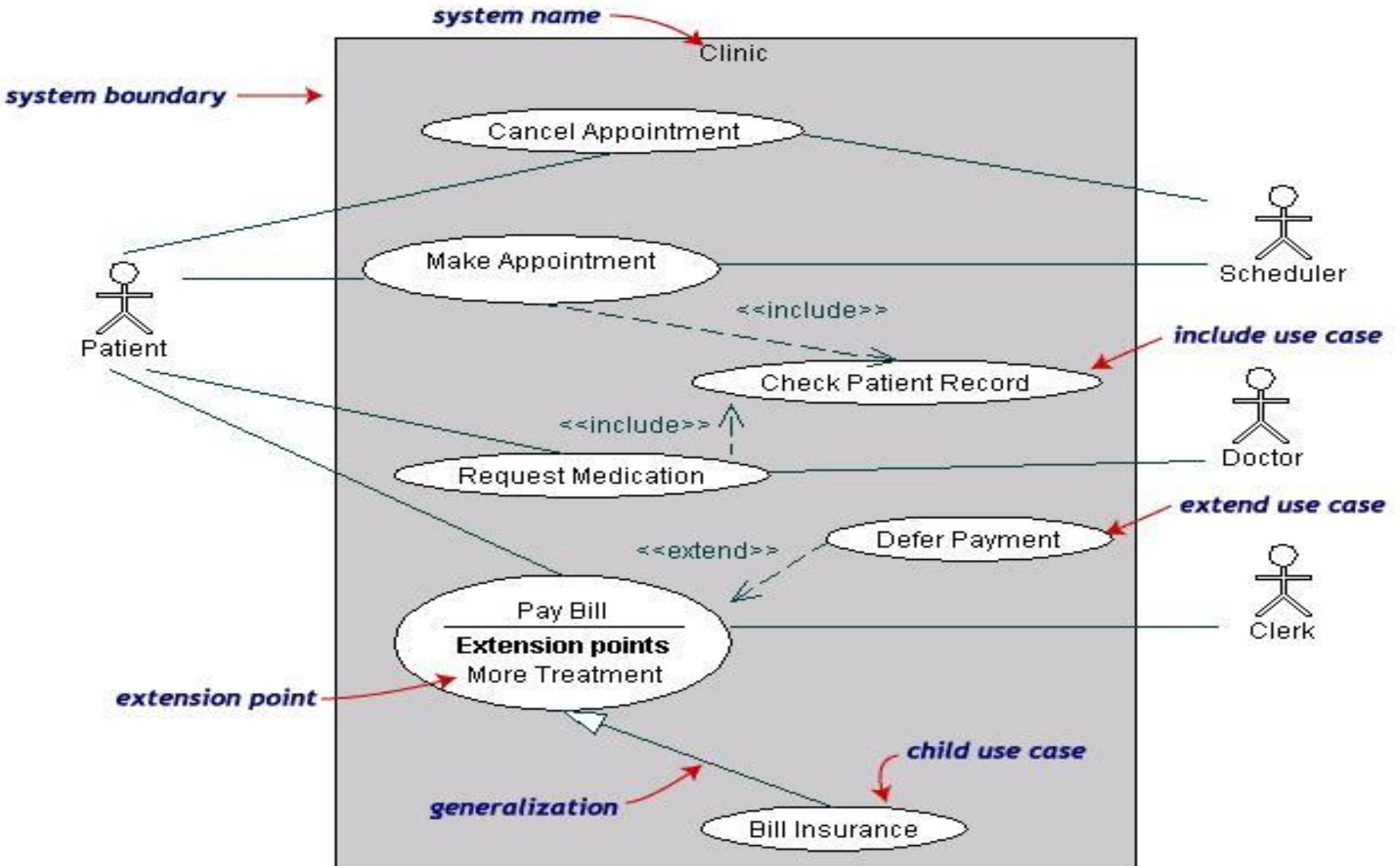


4.Include Relationship Between Two Use Cases

- Include relationship show that the behaviour of the included use case is part of the including (base) use case.
- The main reason for this is to reuse the 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.



Eg: Use case Diagram for a hospital management system

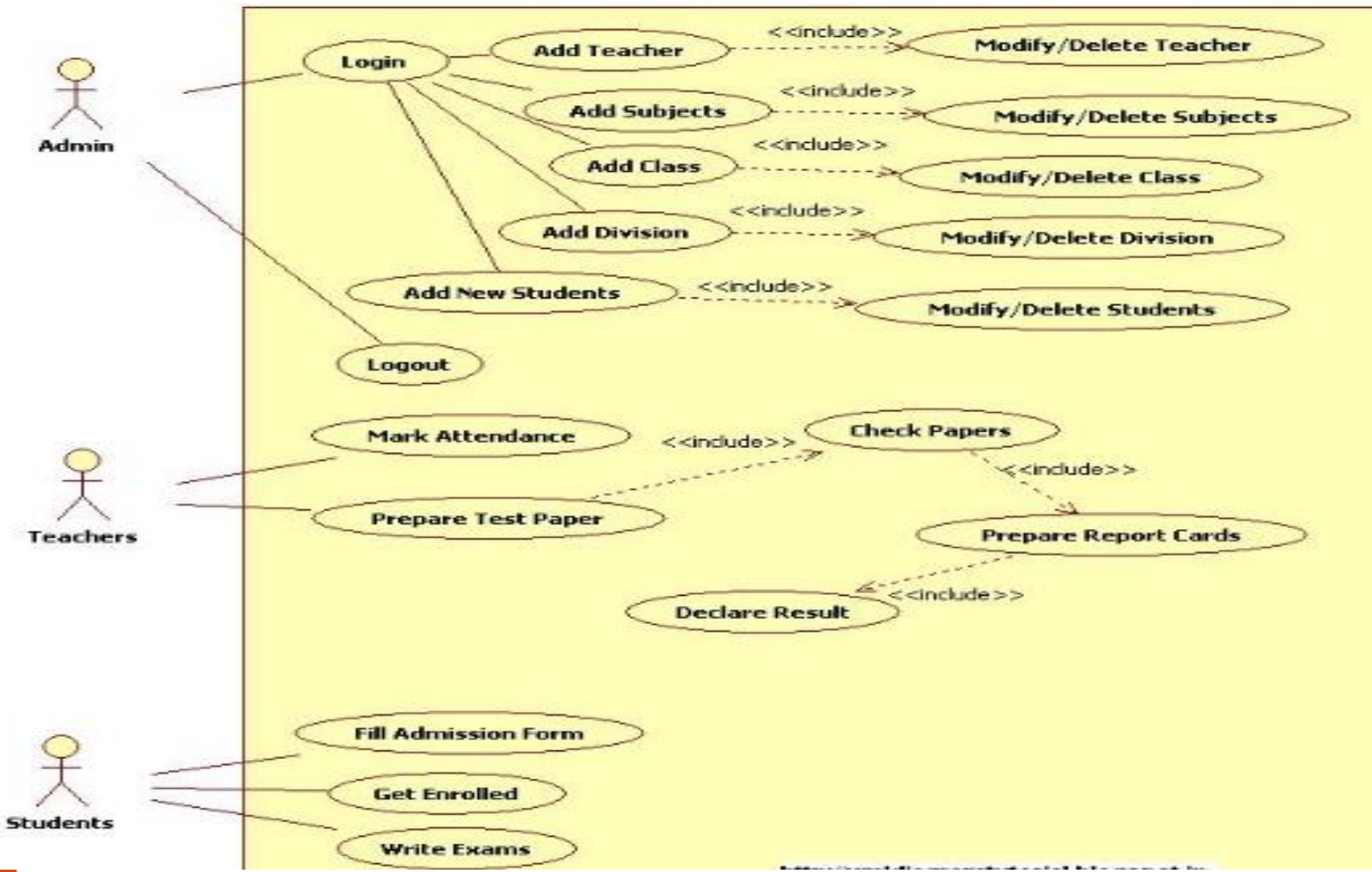


5.Generalization of a Use Case

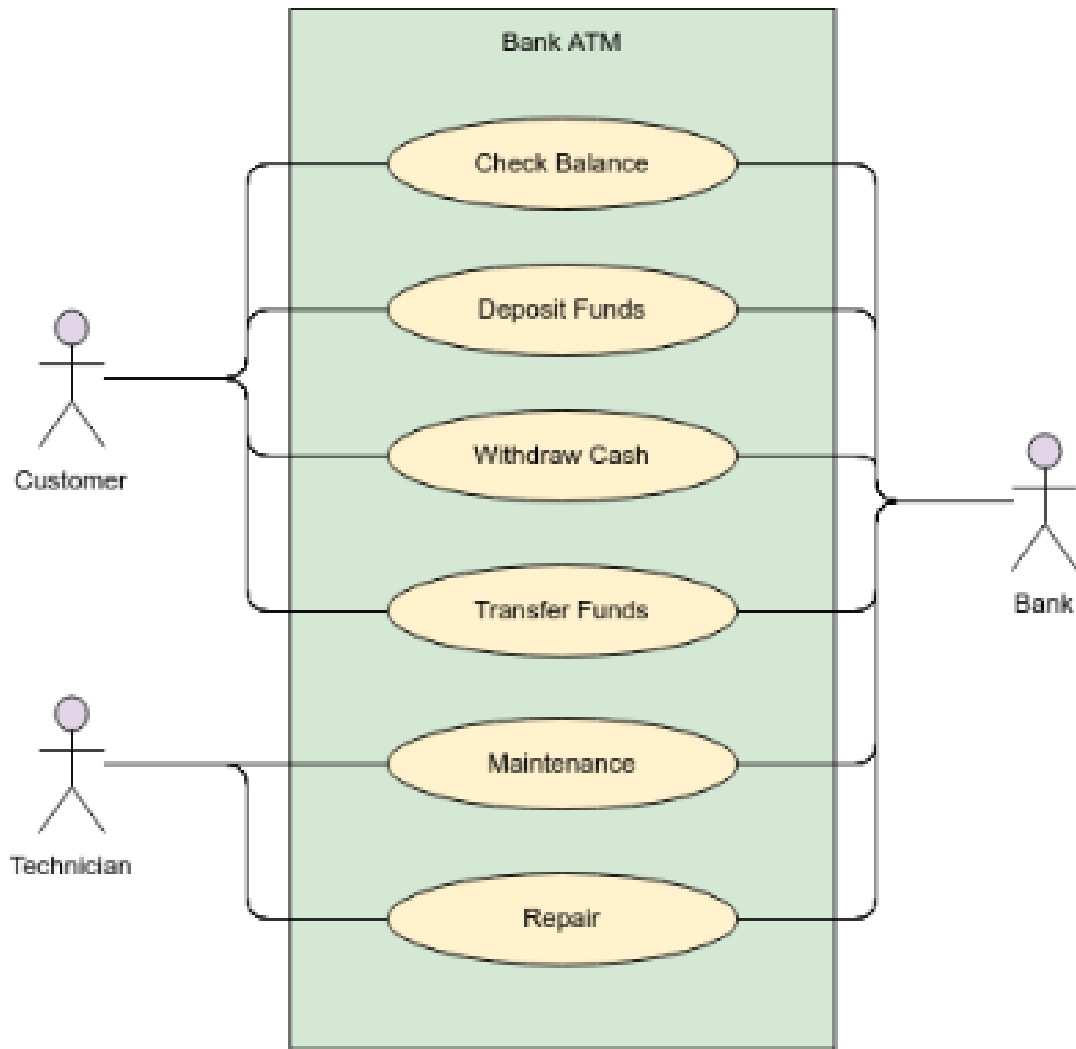
For example,

- **Pay Bill** is a parent use case and **Bill Insurance** is the child use case. (generalization)
- Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask.(include)
- The **extension point** is written inside the base case **Pay bill**; the extending class **Defer payment** adds the behavior of this extension point. (extend)

Use Case Diagram for Student Management System



Use Case Diagram Example: Bank ATM



Class diagram

- A class diagram is a collection of static modeling elements, such as classes and their relationships, connected as a graph.
- Class diagram do not show temporal information.
- Also known as object modeling.
- The main task of class diagram/object modeling is to
 - Graphically show what each object will do in the problem domain.
 - Describe its structure such as class hierarchy.
 - Describe the relationship between objects.

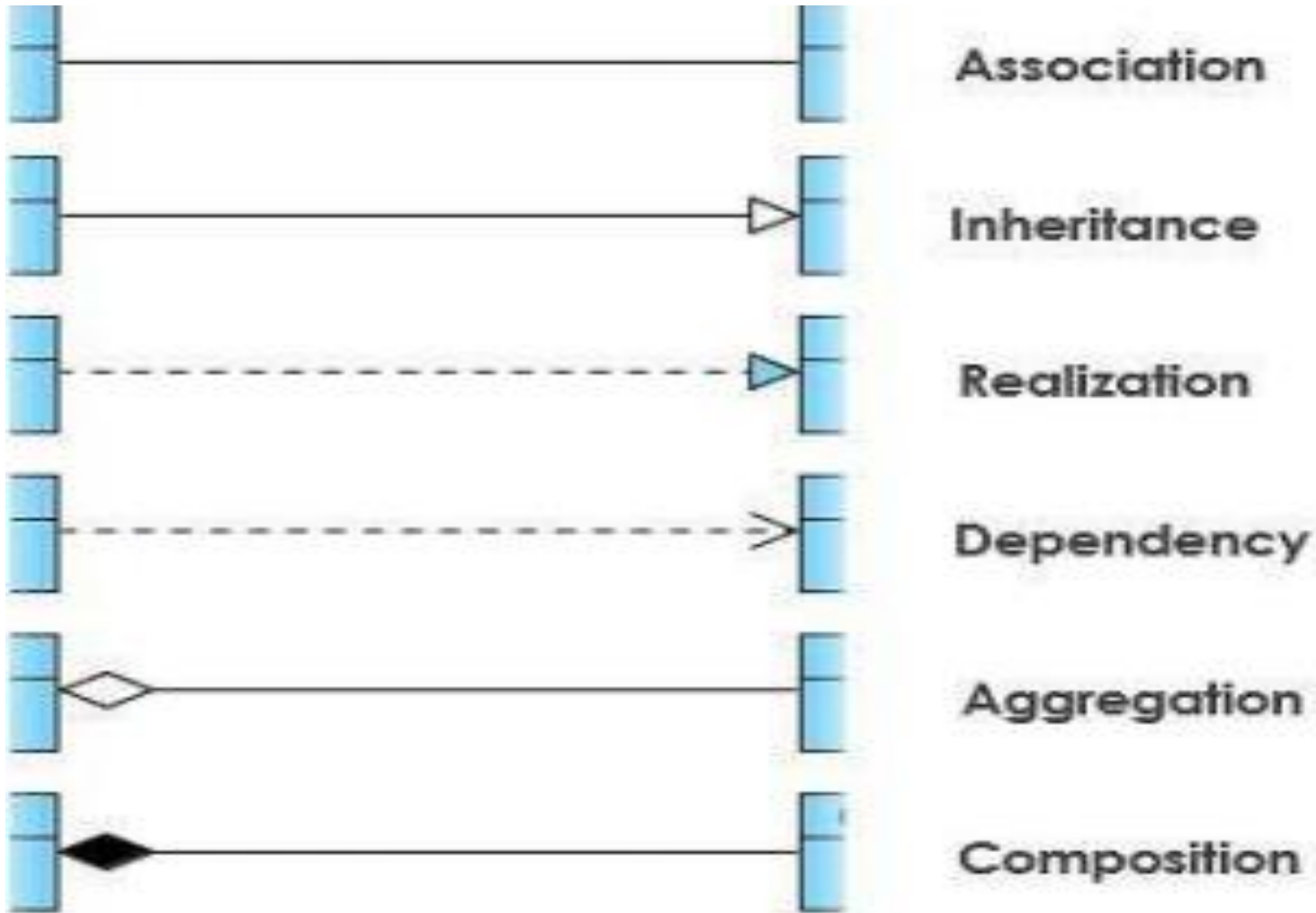
Class representation

- Each class is represented by a rectangle subdivided into three compartments separated by horizontal lines.
 - **Class Name** - The name of the class appears in the first partition
 - **Attributes-**
 - Attributes are shown in the second partition.
 - The attribute type is shown after the colon.
 - Attributes map onto member variables (data members) in code.
 - **Operations**
 - Operations are shown in the third partition.
 - They are services the class provides.
 - The return type of a method is shown after the colon at the end of the method signature.
 - The return type of method parameters are shown after the colon following the parameter name.
 - Operations map onto class methods in code

- Modifiers are used to indicate visibility of attributes and operations.
 - ‘+’ is used to denote *Public* visibility (everyone)
 - ‘#’ is used to denote *Protected* visibility (friends and derived)
 - ‘-’ is used to denote *Private* visibility (no one)

MyClass
+attribute1 : int
-attribute2 : float
#attribute3 : Circle
+op1(in p1 : bool, in p2) : String
-op2(input p3 : int) : float
#op3(out p6) : Class6*

Relationships between classes



1. Dependency

- A dependency means the **relation between two or more classes** in which a change in one may force changes in the other.
- Dependency indicates that one class depends on another.
- A dashed line with an open arrow



2. Inheritance (or Generalization)

- A generalization helps to connect a subclass to its superclass.
- A sub-class is inherited from its superclass.
- A solid line with a hollow arrowhead that point from the child to the parent class

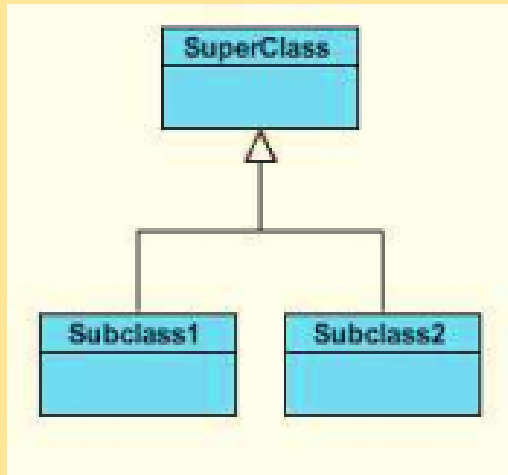


Fig: Inheritance (or Generalization)

3. Association

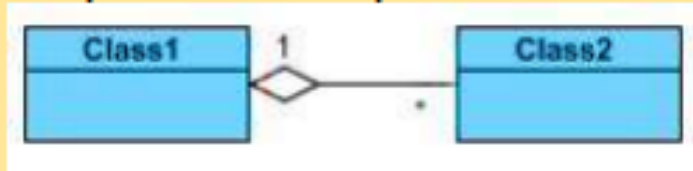
- This kind of relationship represents static relationships between classes A and B.
- There is an association between Class1 and Class2
- A solid line connecting two classes



Fig: Association

4. Aggregation

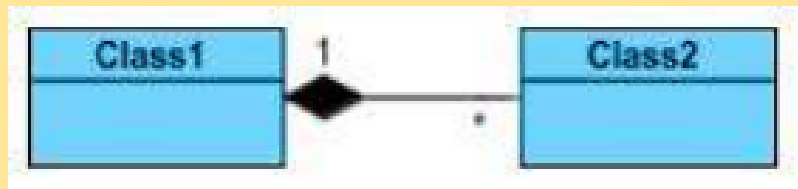
- A special type of association. It represents a "part of" relationship
- Class2 is part of Class1.



- Many instances (denoted by the *) of Class2 can be associated with Class1.
- A solid line with an unfilled diamond at the association end connected to the class of composite

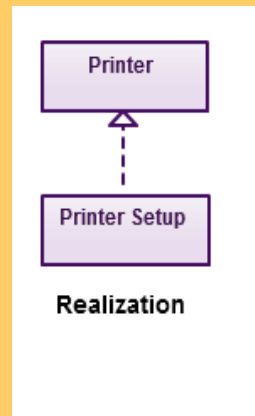
5. Composition

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.
- A solid line with a filled diamond at the association connected to the class of composite



6. Realization

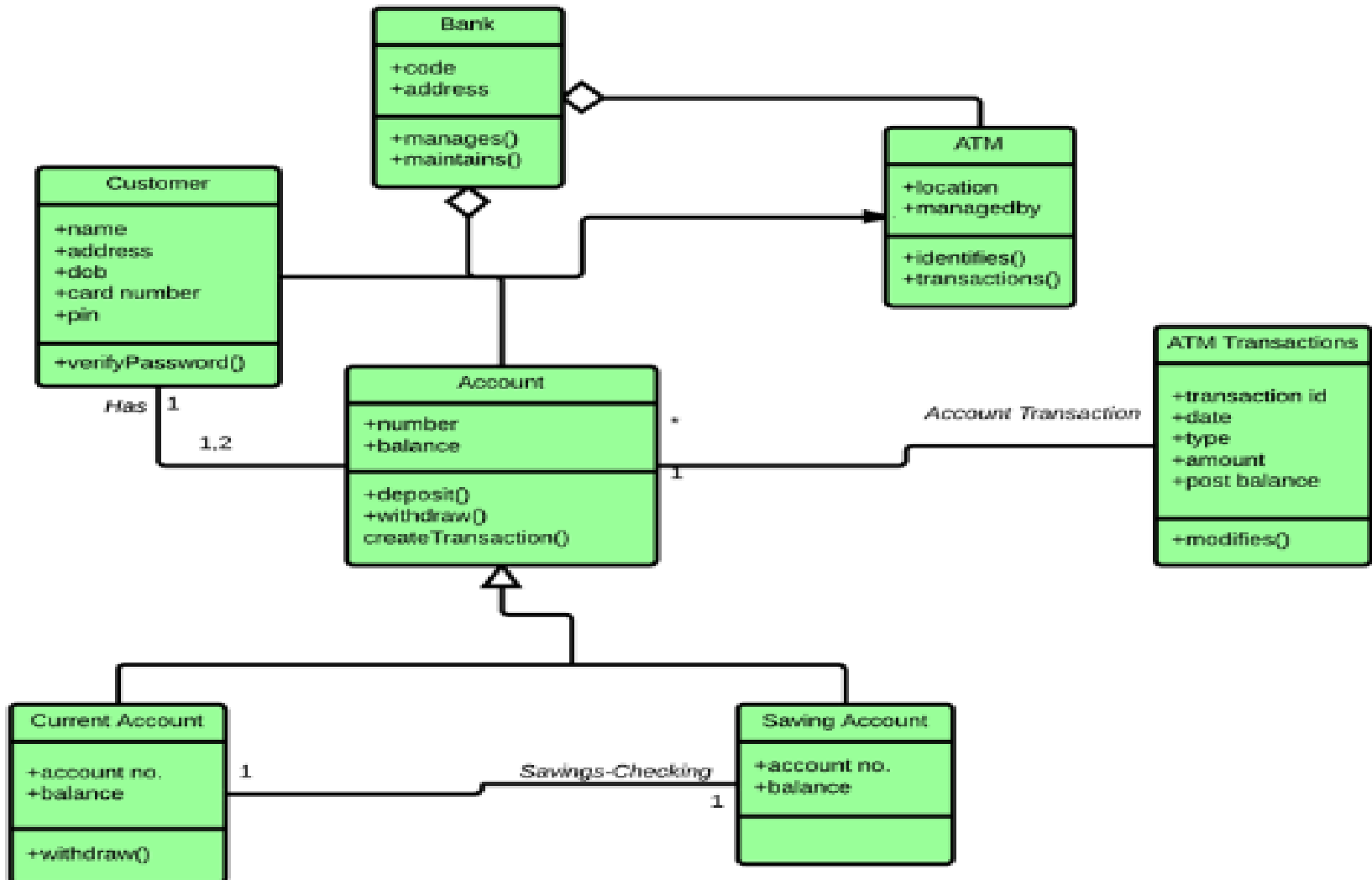
- In a realization relationship of UML, one entity denotes some responsibility which is not implemented by itself and the other entity that implements them.
- This relationship is mostly found in the case of **interfaces**.



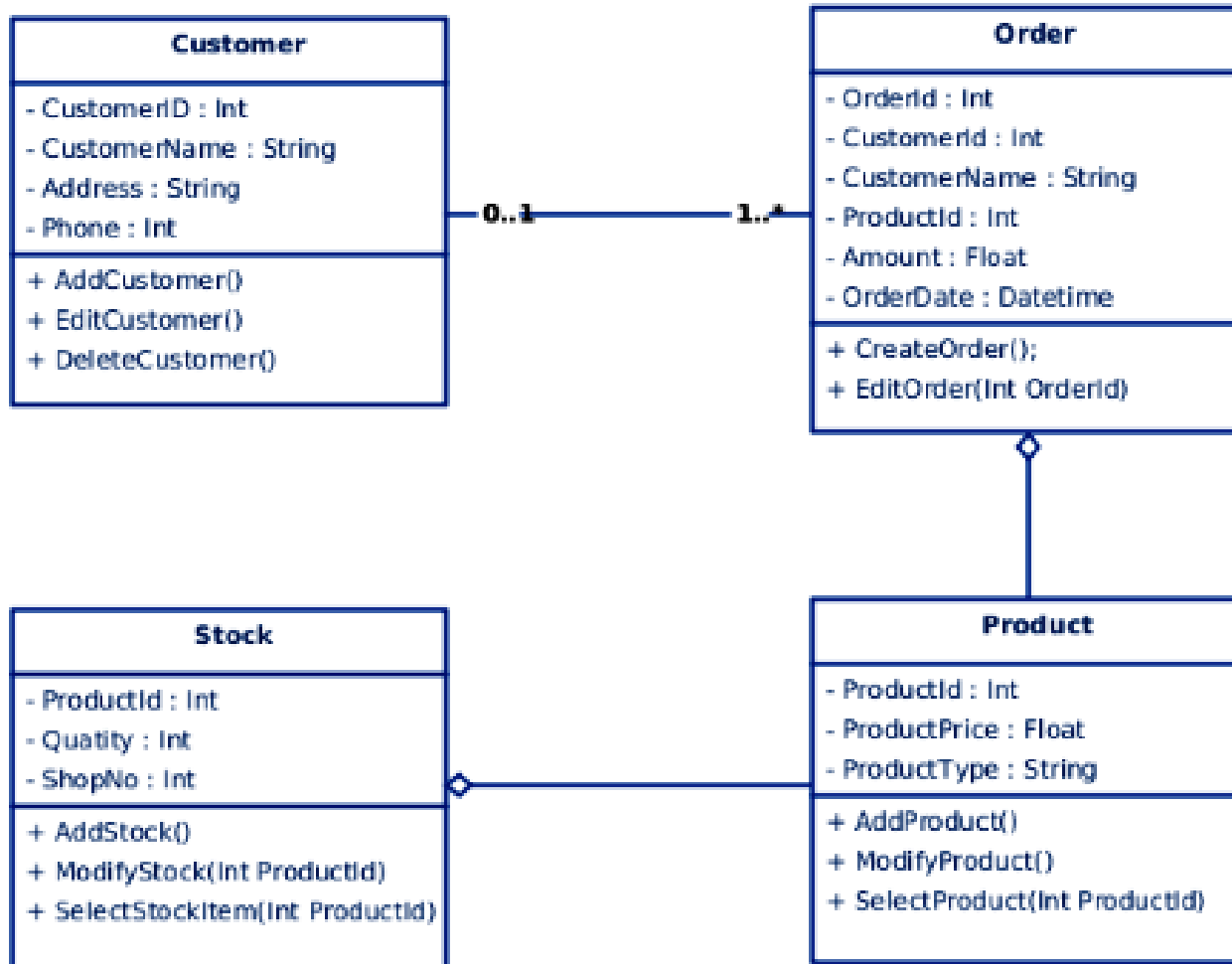
Multiplicity

- It means, how many objects of each class take part in the relationships
- Exactly one - 1
- Zero or one - 0..1
- Many - 0..* or *
- One or more - 1..*
- Exact Number - e.g. 3..4 or 6
- Or a complex relationship - e.g. 0..1, 3..4, 6.* would mean any number of objects other than 2 or 5

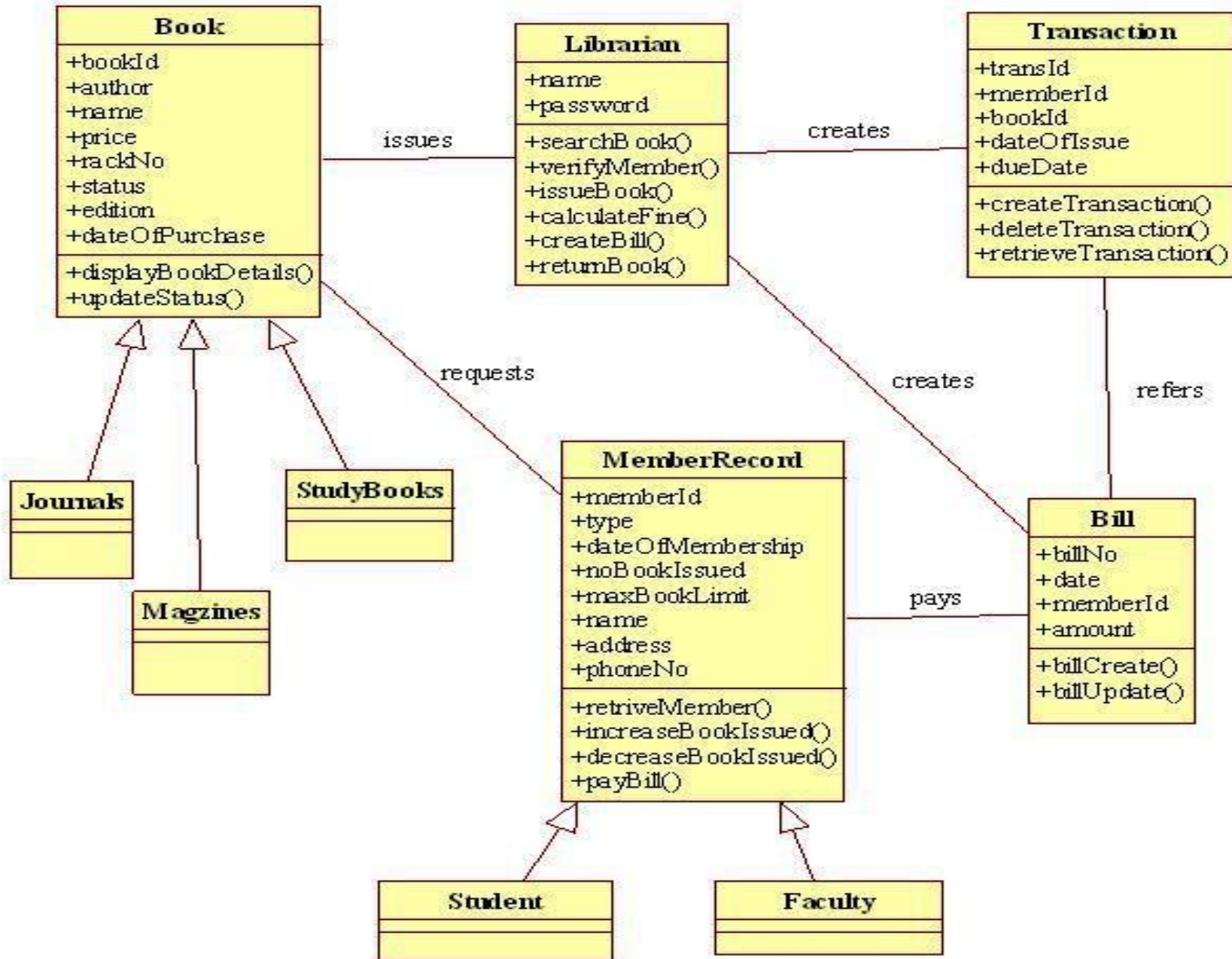
Class diagram for an ATM system



Class Diagram for Order Processing System



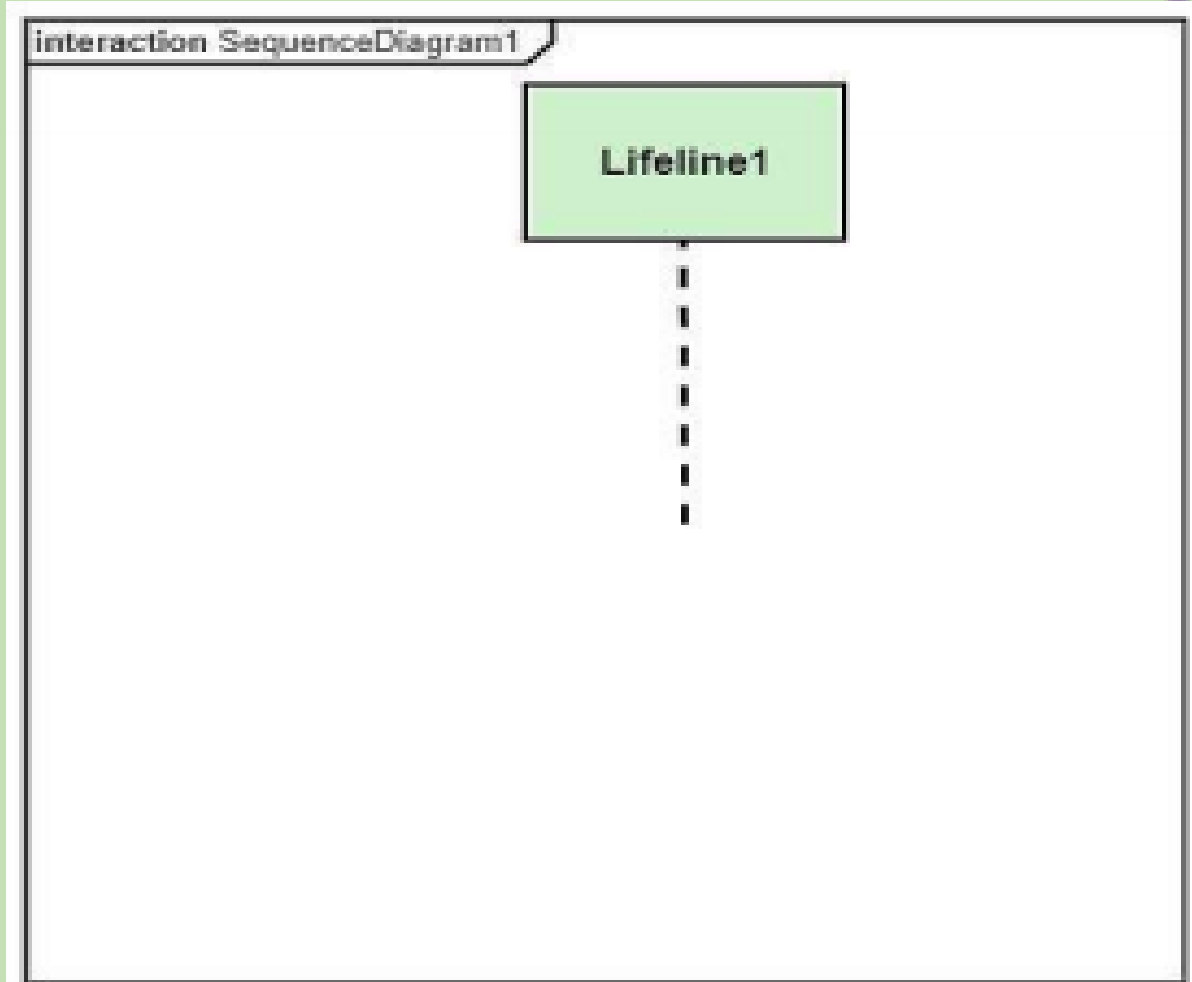
Eg: Class diagram for library management system



INTERACTION DIAGRAM

- INTERACTION DIAGRAMS are used in UML to establish communication between objects
- Interaction diagrams mostly focus on message passing and how these messages make up one functionality of a system
- The **critical component** in an interaction diagram is **lifeline and messages**.
- Interaction diagrams capture the dynamic behavior of any system
- The details of interaction can be shown using several notations such as sequence diagram, timing diagram, collaboration diagram.

Notation of an Interaction Diagram



Purpose of an Interaction Diagram

- To capture the dynamic behavior of a system.
- To describe the message flow in the system.
- To describe the structural organization of the objects.
- To describe the interaction among objects.
- Interaction diagram visualizes the communication and sequence of message passing in the system.
- Interaction diagram represents the ordered sequence of interactions within a system.
- Interaction diagrams can be used to explain the architecture of an object-oriented system.

Different types of Interaction Diagrams

1. **Sequence diagram**

- Purpose - To visualize the sequence of a message flow in the system
- Shows the interaction between two lifelines

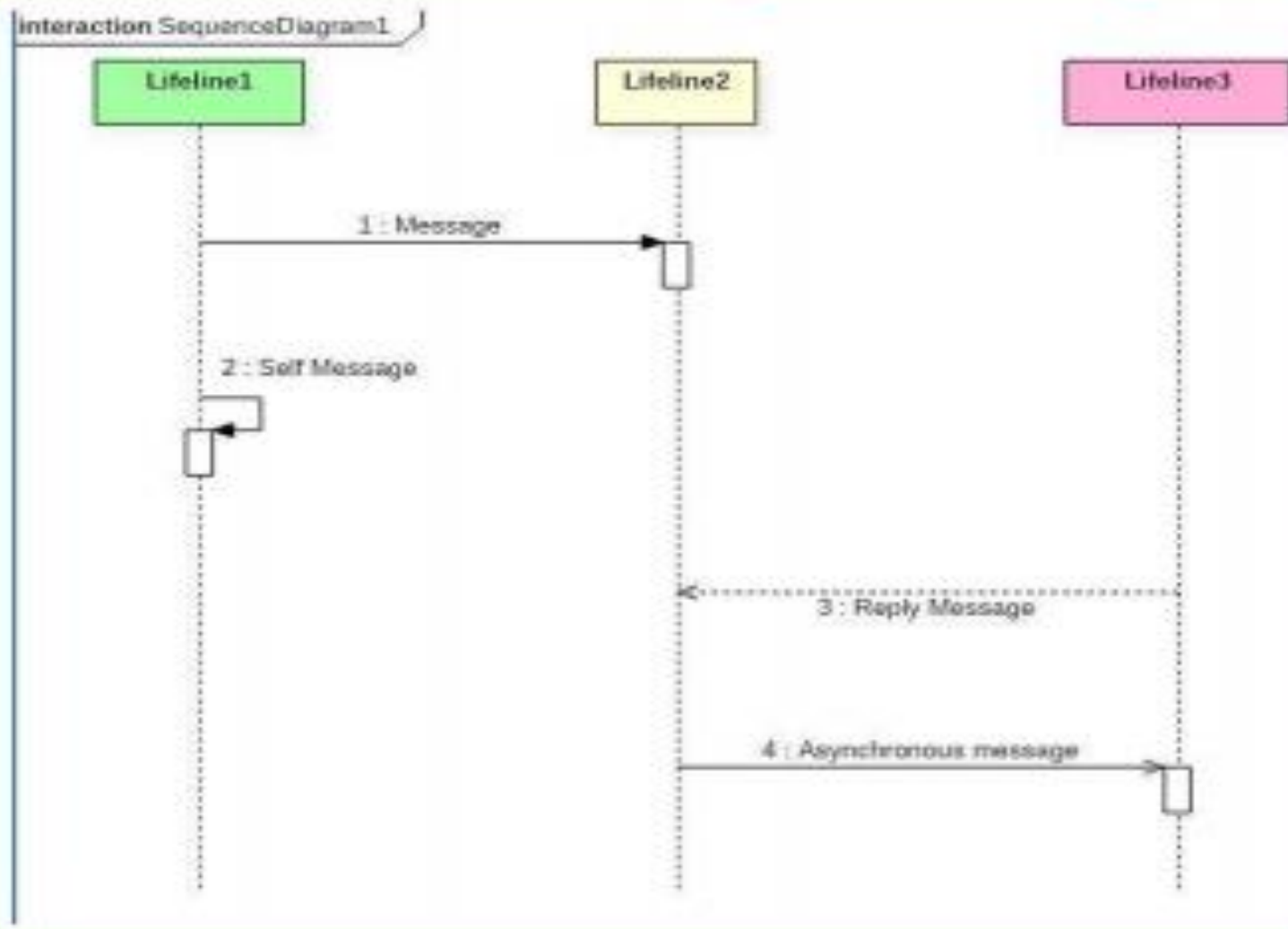
2. **Collaboration diagram**

- Also called as a communication diagram
- Shows how various lifelines in the system connects.

3. **Timing diagram**

- Focus on the instance at which a message is sent from one object to another object.

How to draw a Sequence Diagram

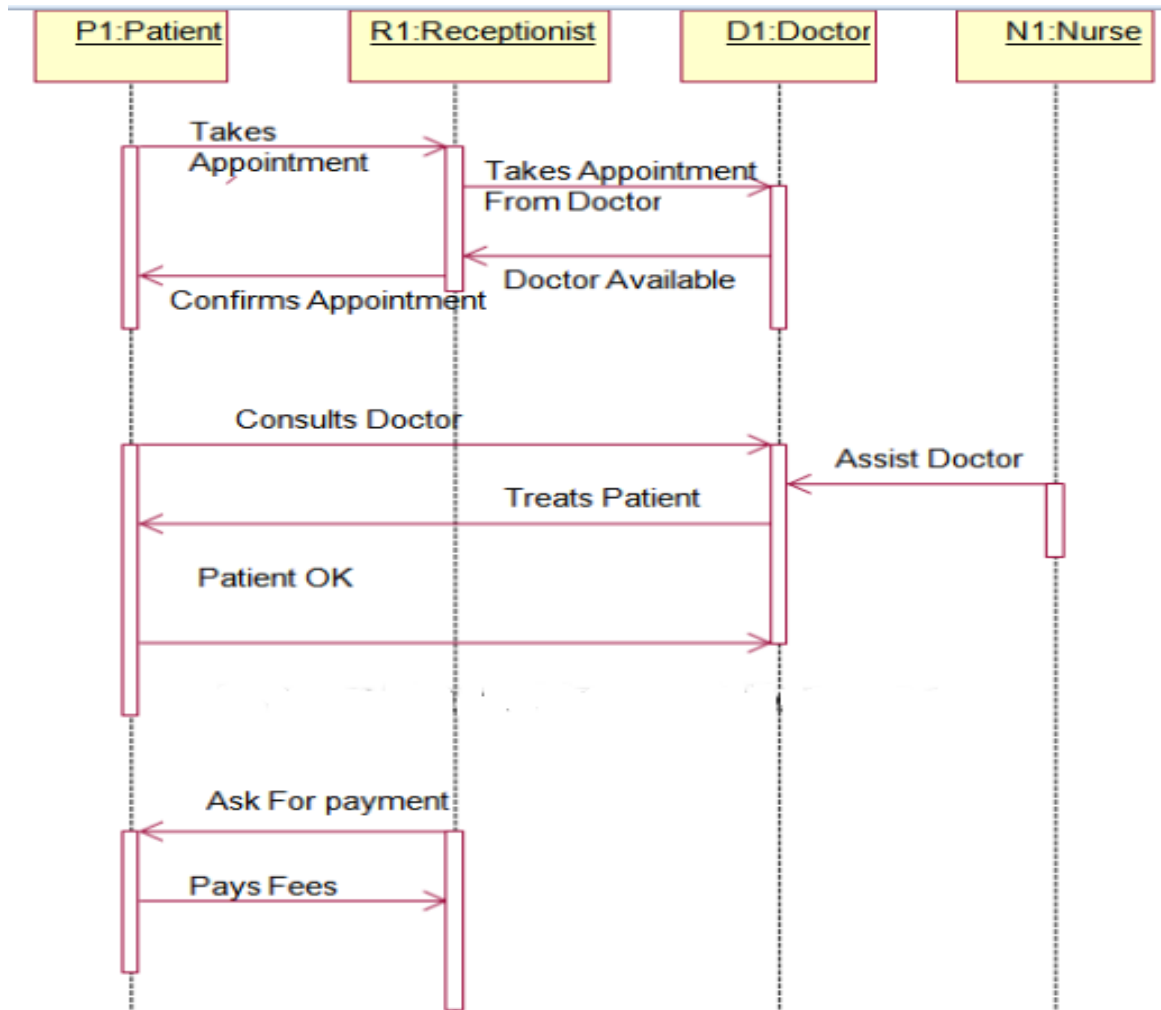


- A **lifeline** represents **an individual participant** in a sequence diagram
- A **lifeline** will usually have **a rectangle containing its object name**
- Communication between objects is depicted using **messages**. The messages appear in a sequential order on the lifeline. We represent **messages using arrows**. **Lifelines and messages** form the **core of a sequence diagram**.
- In a sequence diagram, different types of messages and operators are used
- In a sequence diagram, iteration and branching are also used.

Messages used

Message Name	Meaning
Synchronous message	The sender of a message keeps waiting for the receiver to return control from the message execution.
Asynchronous message	The sender does not wait for a return from the receiver; instead, it continues the execution of a next message.
Return message	The receiver of an earlier message returns the focus of control to the sender.
Object creation	The sender creates an instance of a classifier.
Object destruction	The sender destroys the created instance.
Found message	The sender of the message is outside the scope of interaction.
Lost message	The message never reaches the destination, and it is lost in the interaction.

Sequence diagram for Hospital Management System



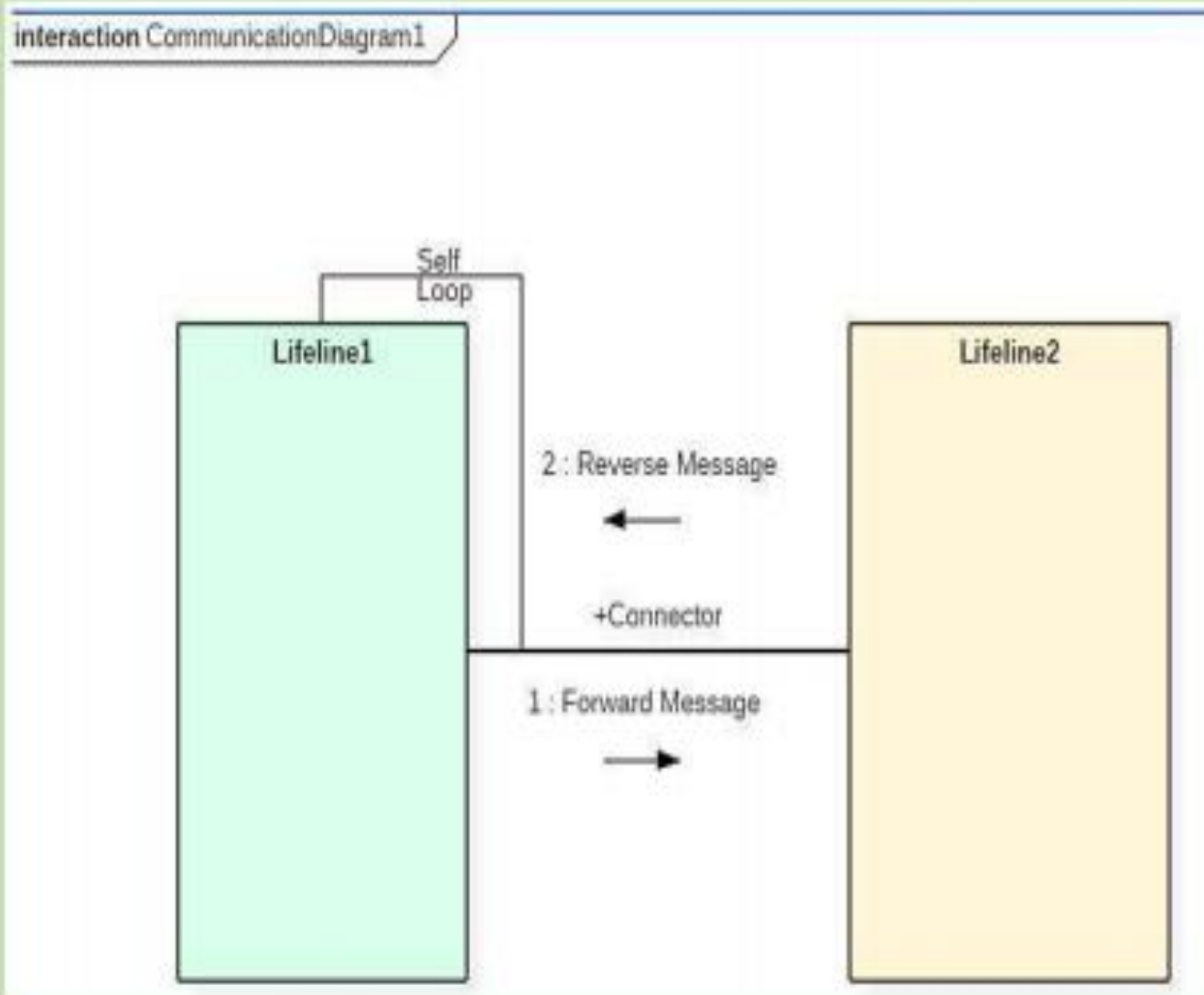
Benefits of a Sequence Diagram

- Sequence diagrams are used to explore any **real application or a system**.
- Sequence diagrams are used to **represent message flow** from one object to another object.
- Sequence diagrams are **easier to maintain**.
- Sequence diagrams are **easier to generate**.
- Sequence diagrams can be **easily updated** according to the changes within a system.
- Sequence diagram allows **reverse as well as forward engineering**.

Drawbacks of a sequence diagram

- Sequence diagrams can become complex when too many lifelines are involved in the system.
- If the order of message sequence is changed, then incorrect results are produced.
- Each sequence needs to be represented using different message notation, which can be a little complex.
- The type of message decides the type of sequence inside the diagram

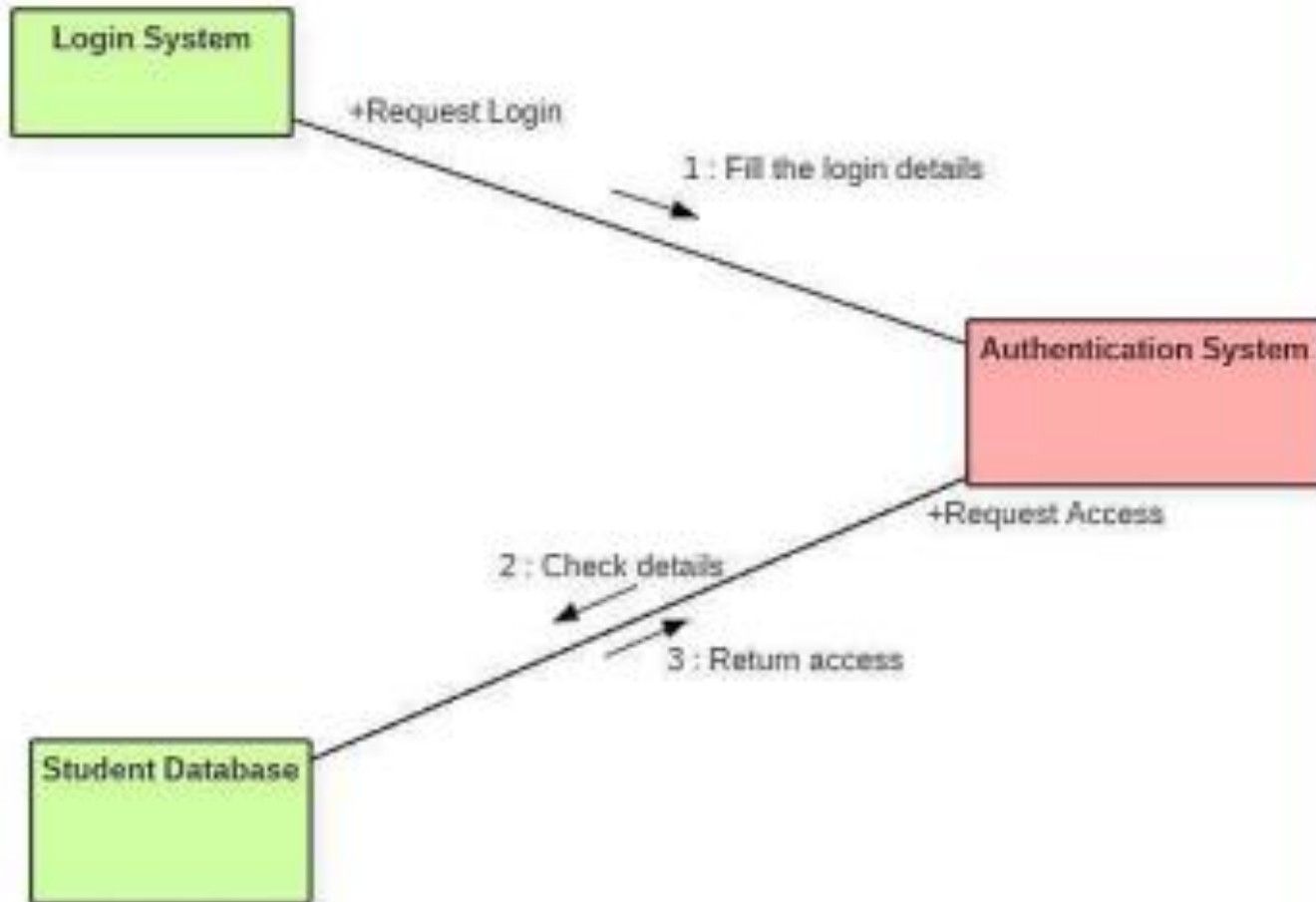
How to draw a Collaboration /Communication Diagram



- As per Object-Oriented Programming (OOPs), an object entity has various attributes associated with it.
- Usually, there are multiple objects present inside an object-oriented system where each object can be associated with any other object inside the system.
- Collaboration Diagrams are used to explore the **architecture of objects inside the system**.
- The **message flow** between the objects can be represented using a collaboration diagram.

Collaboration Diagram Example

interaction Student Management System



- The above collaboration diagram represents a student information management system. The flow of communication in the above diagram is given by,
- A student requests a login through the login system.
 - An authentication mechanism of software checks the request.
 - If a student entry exists in the database, then the access is allowed; otherwise, an error is returned.

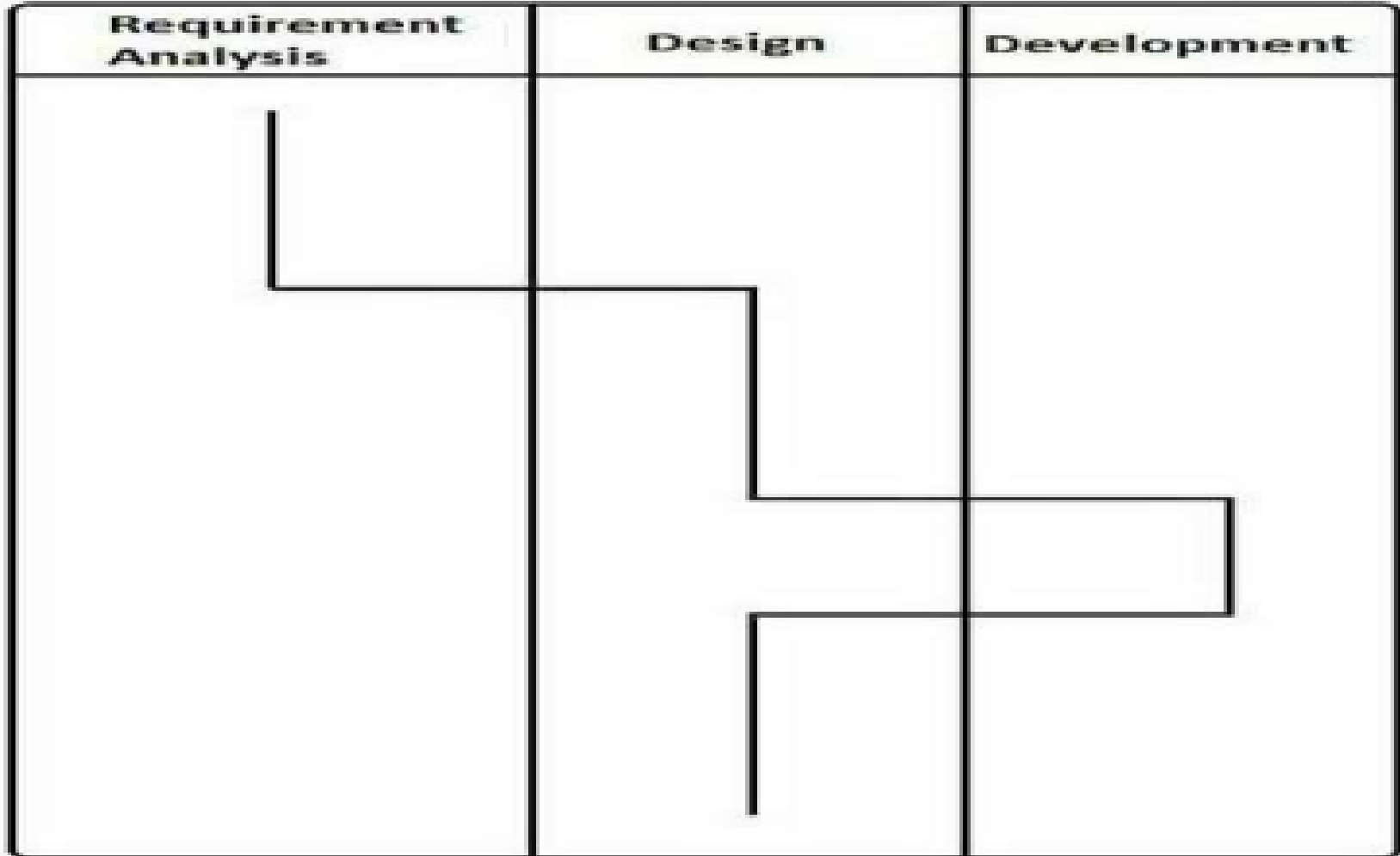
Benefits of Collaboration Diagram

- It is also called as a communication diagram.
- It emphasizes the structural aspects of an interaction diagram - how lifeline connects.
- Its syntax is similar to that of sequence diagram except that lifeline don't have tails.
- Messages passed over sequencing is indicated by numbering each message hierarchically.
- It allows you to focus on the elements rather than focusing on the message flow as described in the sequence diagram.
- Sequence diagrams can be easily converted into a collaboration diagram as collaboration diagrams are not very expressive.

Drawbacks of a Collaboration Diagram

- Collaboration diagrams can become complex when too many objects are present within the system.
- It is hard to explore each object inside the system.
- Collaboration diagrams are time consuming.
- The object is destroyed after the termination of a program.
- The state of an object changes momentarily, which makes it difficult to keep track of every single change that occurs within an object of a system.

How to draw a Timing Diagram



- In the above diagram, first, the software passes through the requirements phase then the design and later the development phase.
- The output of the previous phase at that given instance of time is given to the second phase as an input
- Thus, the timing diagram can be used to describe SDLC (Software Development Life Cycle) in UML

Benefits of a Timing Diagram

- Timing diagrams are used to represent the state of an object at a particular instance of time.
- Timing diagram allows reverse as well as forward engineering.
- Timing diagram can be used to keep track of every change inside the system.

Drawbacks of a Timing Diagram

- Timing diagrams are difficult to understand.
- Timing diagrams are difficult to maintain.

ACTIVITY DIAGRAM

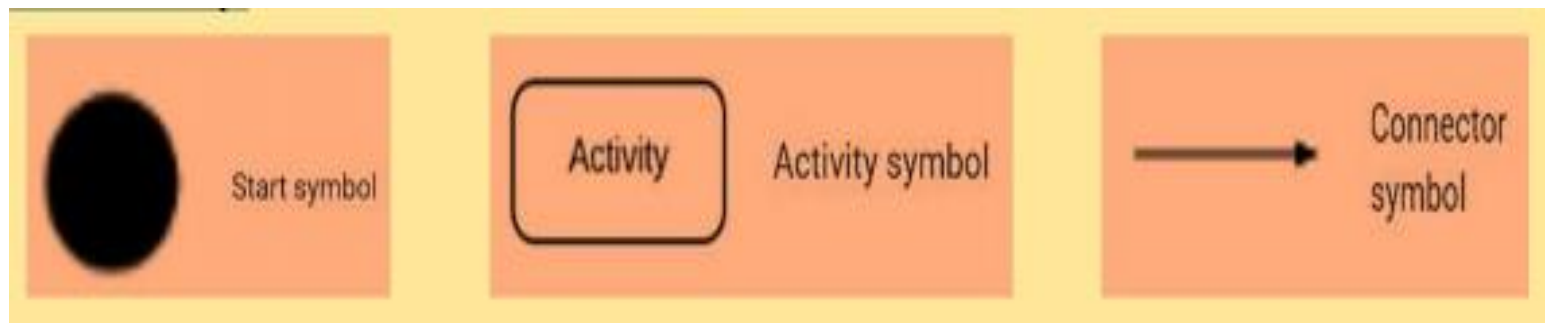
- ACTIVITY DIAGRAM is basically a flowchart to represent the flow from one activity to another activity.
- The activity can be described as an operation of the system.
- The basic purpose of activity diagrams is to capture the dynamic behavior of the system.
- It is also called object-oriented flowchart.
- Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable system by using forward and reverse engineering techniques.

Basic components of an activity diagram

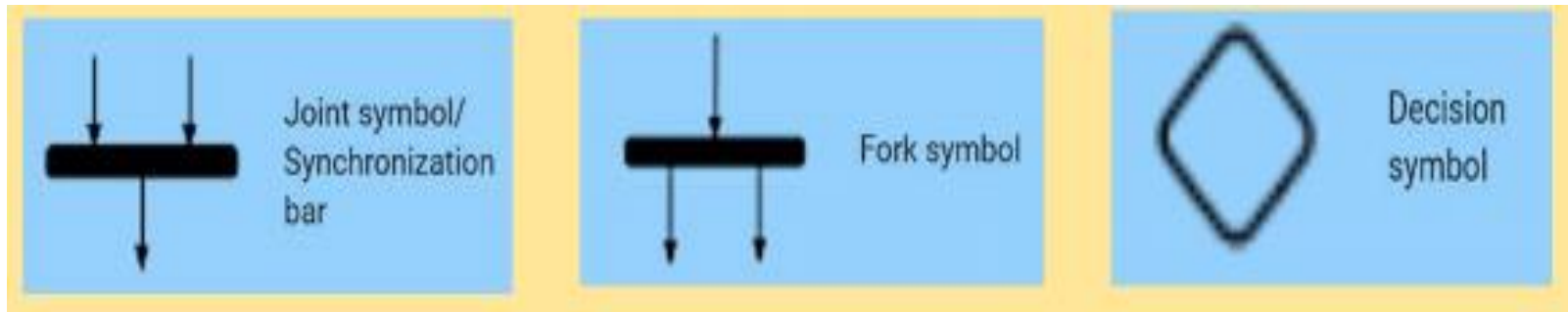
- Action: A step in the activity wherein the users or software perform a given task.
- Decision node: A conditional branch in the flow that is represented by a diamond. It includes a single input and two or more outputs.
- Control flows: Another name for the connectors that show the flow between steps in the diagram.
- Start node: Symbolizes the beginning of the activity. The start node is represented by a black circle.
- End node: Represents the final step in the activity. The end node is represented by an outlined black circle.

Activity diagram symbols

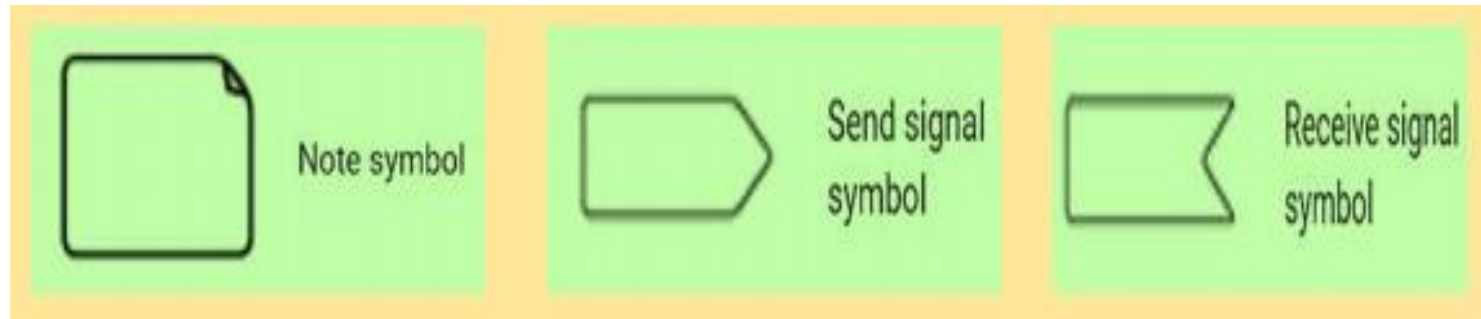
- Start symbol - Represents the beginning of a process or workflow in an activity diagram.
- Activity symbol - Indicates the activities that make up a modeled process. These symbols, which include short descriptions within the shape, are the main building blocks of an activity diagram.
- Connector symbol - Shows the directional flow, or control flow, of the activity.



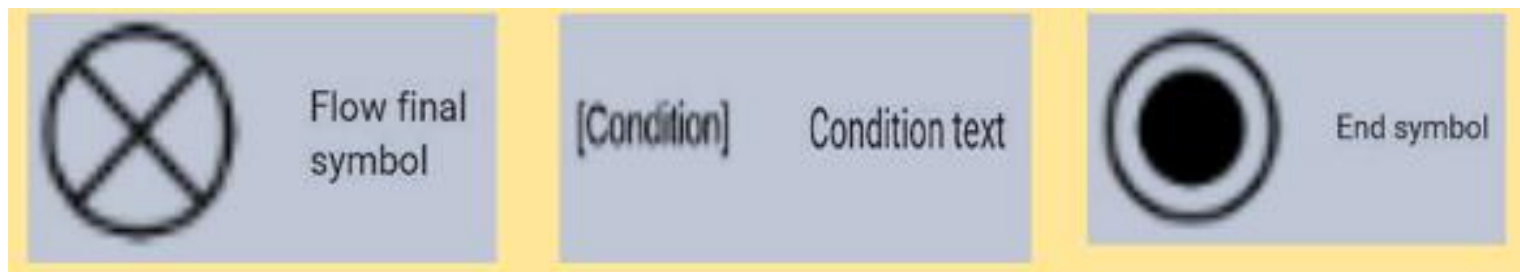
- Joint symbol / Synchronization bar - Combines two concurrent activities and re-introduces them to a flow where only one activity occurs at a time. Represented with a thick vertical or horizontal line.
- Fork symbol - Splits a single activity flow into two concurrent activities. Symbolized with multiple arrowed lines from a join.
- Decision symbol - Represents a decision and always has at least two paths branching out with condition text.



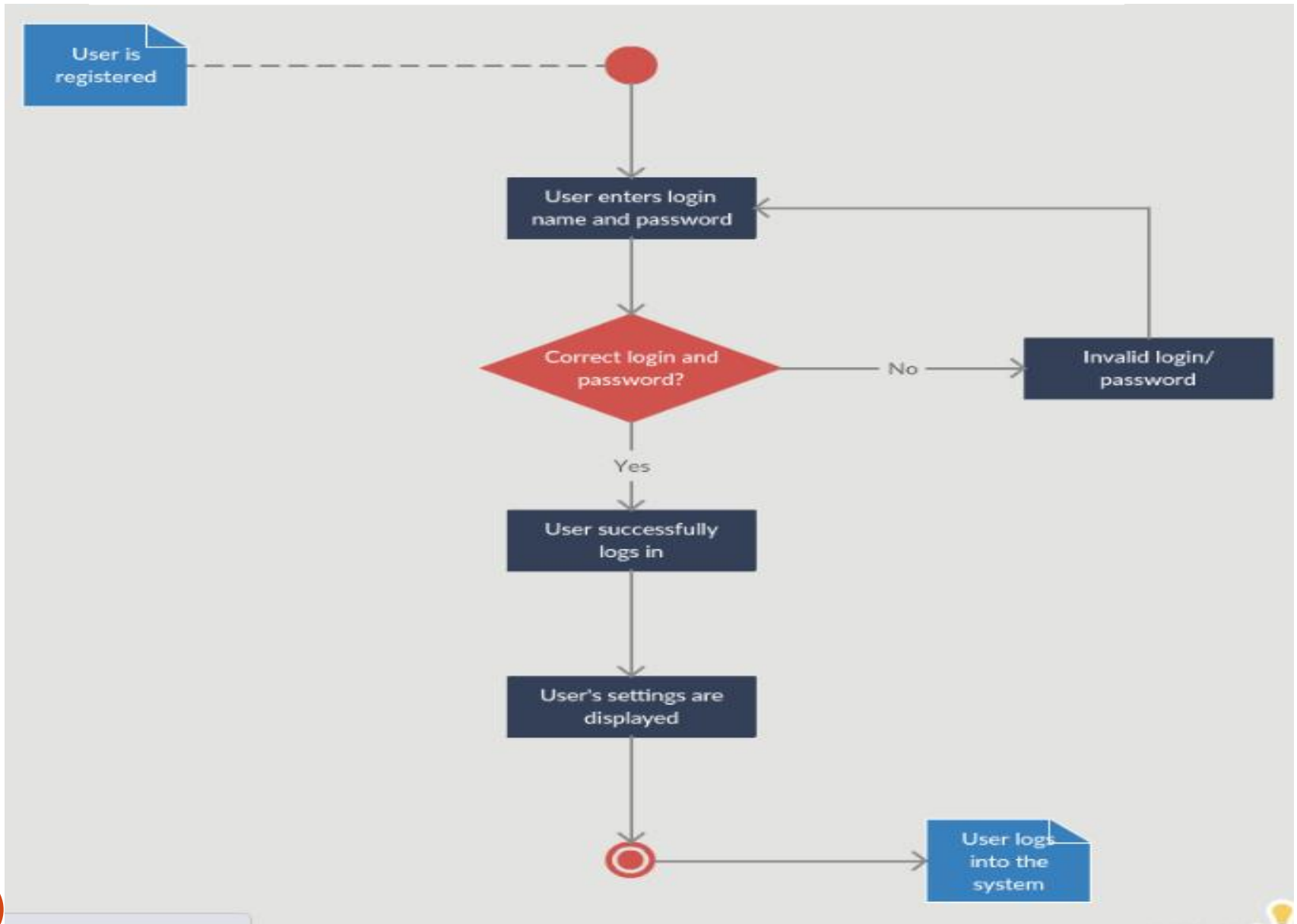
- Note symbol - Allows the diagram creators or collaborators to communicate additional messages that don't fit within the diagram itself. Leave notes for added clarity and specification.
- Send signal symbol - Indicates that a signal is being sent to a receiving activity
- Receive signal symbol - Demonstrates the acceptance of an event. After the event is received, the flow that comes from this action is completed.



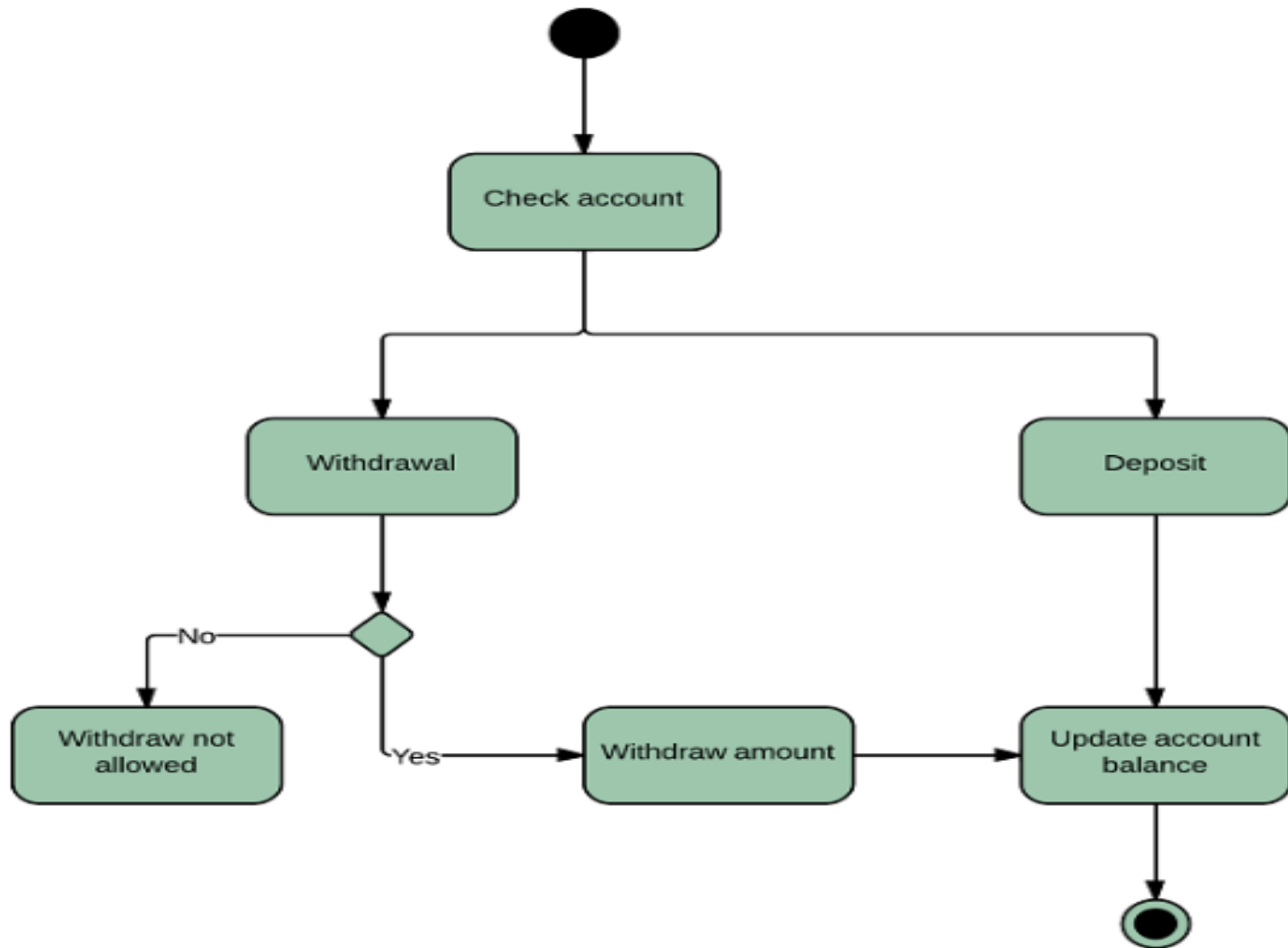
- Flow final symbol - Represents the end of a specific process flow. This symbol shouldn't represent the end of all flows in an activity. The flow final symbol should be placed at the end of a single activity flow.
- Condition text - Placed next to a decision marker to let you know under what condition an activity flow should split off in that direction
- End symbol - Marks the end state of an activity and represents the completion of all flows of a process.



Activity Diagram for a Login page



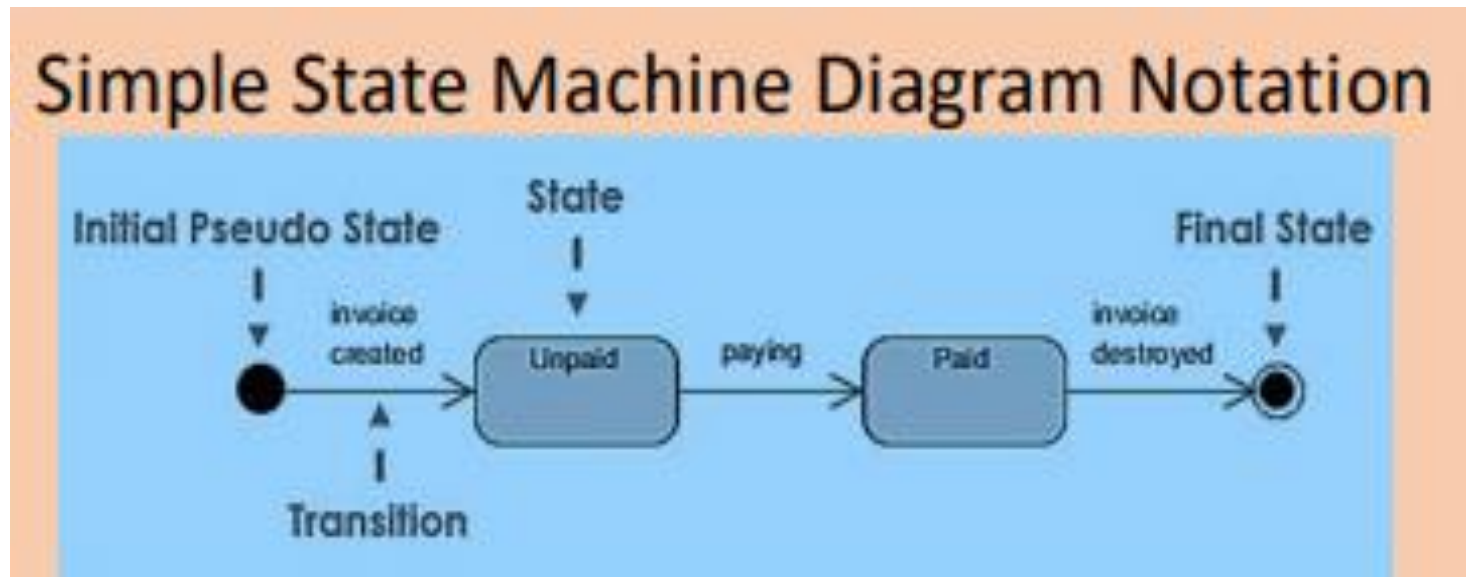
Activity diagram for Banking System




STATE CHART DIAGRAM


- State chart diagram is used to capture the dynamic aspect of a system
- An object goes through various states during its lifespan. The lifespan of an object remains until the program is terminated. The object goes from multiple states depending upon the event that occurs within the object.
- Each state represents some unique information about the object.
- State chart diagram visualizes the flow of execution from one state to another state of an object.
- It represents the state of an object from the creation of an object until the object is destroyed or terminated.

- The primary purpose of a state chart diagram is to model interactive systems and define each and every state of an object.
- State chart diagrams are also referred to as State machines and state diagrams.
- A state machine consists of states, linked by transitions. A state is a condition of an object in which it performs some activity or waits for an event




Notation and Symbol for State Machine / State Chart Diagram

 initial state

 state-box

 decision-box

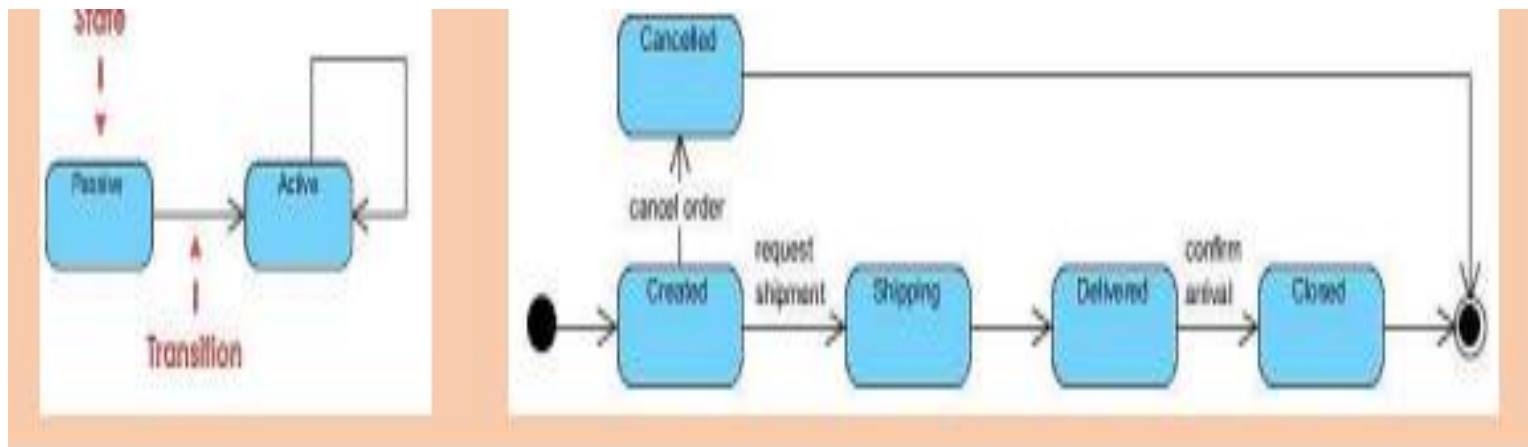
 final-state

UML state diagram notations

- Initial state - The initial state symbol is used to indicate the beginning of a state machine diagram.
- Final state - This symbol is used to indicate the end of a state machine diagram.
- Decision box - It contains a condition. Depending upon the result of an evaluated guard condition, a new path is taken for program execution.
- Transition - A transition is a change in one state into another state which is occurred because of some event. A transition causes a change in the state of an object.

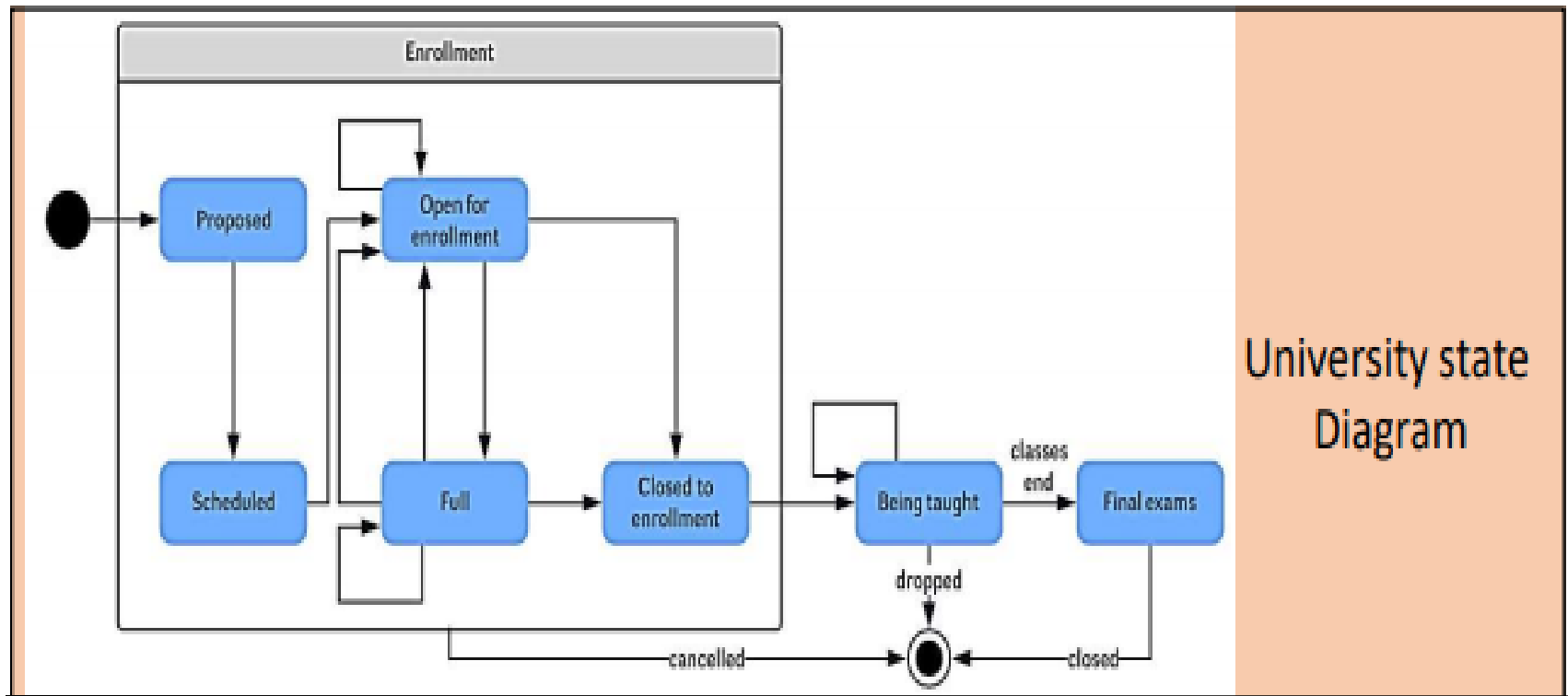
State box

- States represent situations during the life of an object.
- It is denoted using a rectangle with round corners.
- The name of a state is written inside the rounded rectangle.
- A state can be either active or inactive.
- When a state is in the working mode, it is active, as soon as it stops executing and transits into another state, the previous state becomes inactive, and the current state becomes active.



Types of State

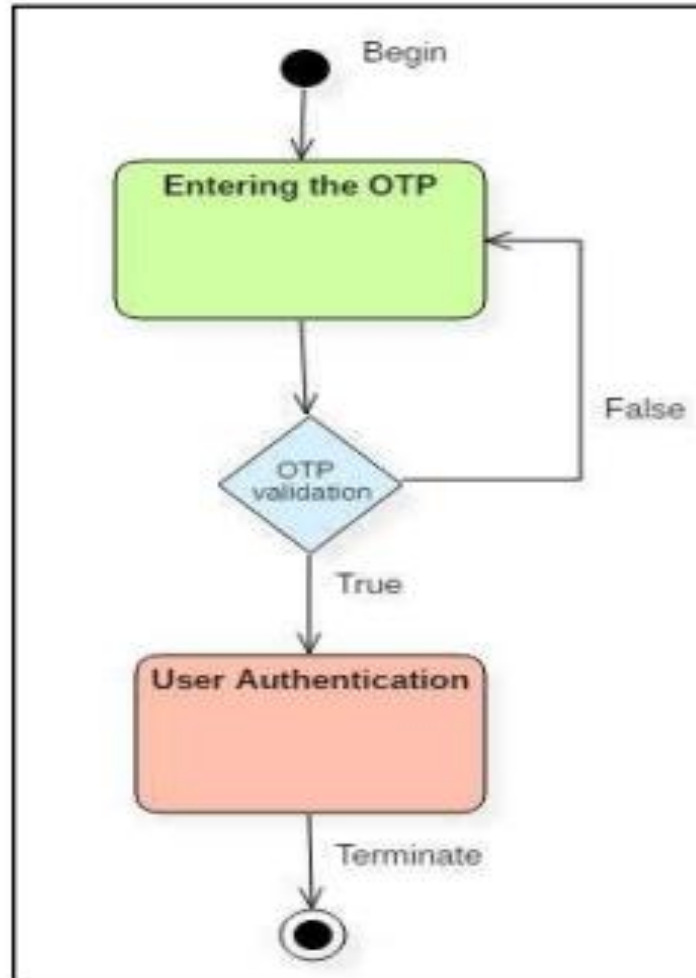
- Simple state
 - They do not have any sub state.
- Composite state
 - These types of states can have one or more than one sub state.
 - A composite state with two or more sub states is called an orthogonal state.
- Submachine state
 - These states are semantically equal to the composite states
 - Unlike the composite state, we can reuse the submachine states.



University state
Diagram

- The composite state “Enrollment” is made up of various sub states that will lead students through the enrollment process.
- Once the student has enrolled, they will proceed to “Being taught” and finally to “Final exams.”

Eg: State chart diagram of User Authentication Process



State machine vs. Flowchart

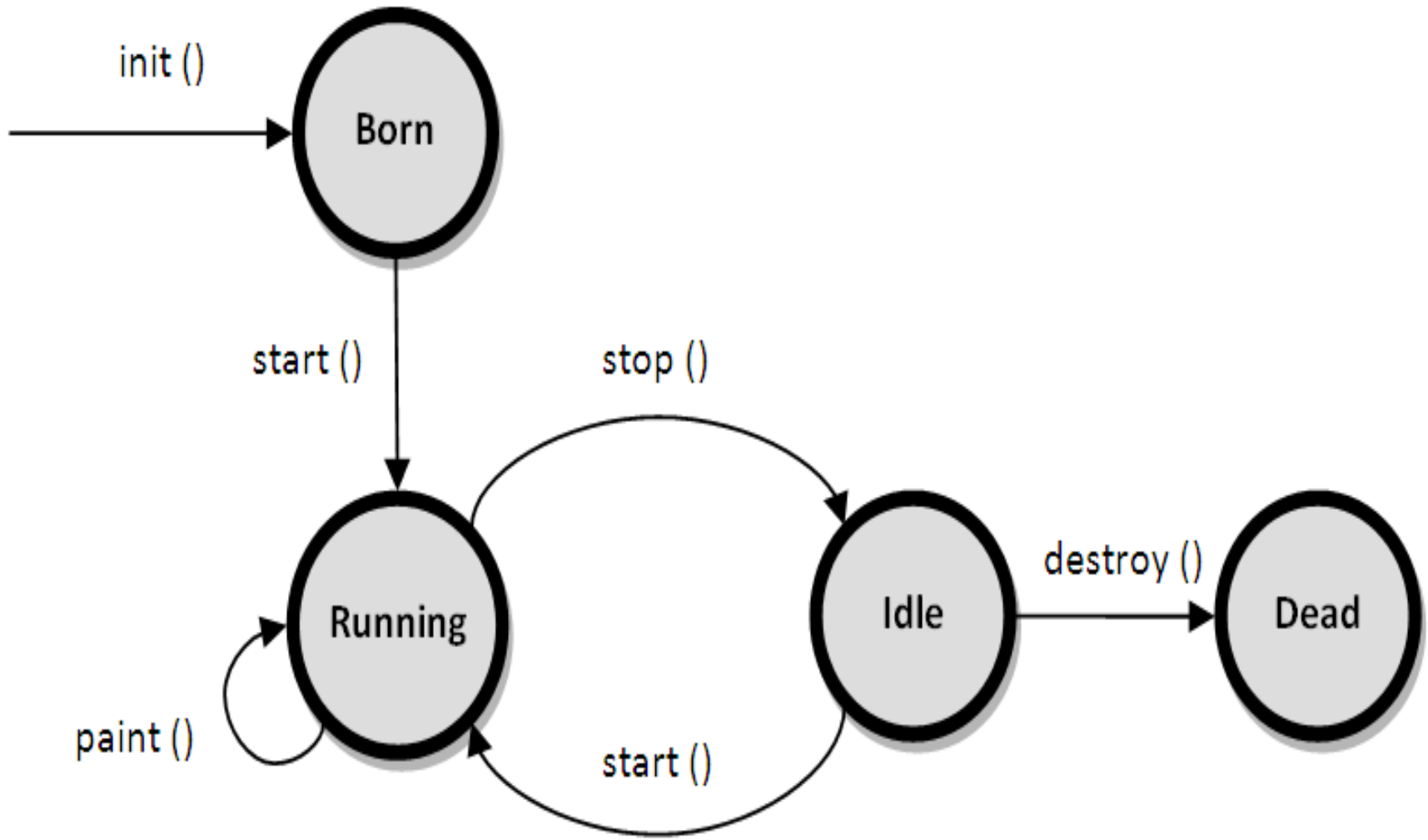
Statemachine	FlowChart
It represents various states of a system.	The Flowchart illustrates the program execution flow.
The state machine has a WAIT concept, i.e., wait for an action or an event.	The Flowchart does not deal with waiting for a concept.
State machines are used for a live running system.	Flowchart visualizes branching sequences of a system.
The state machine is a modeling diagram.	A flowchart is a sequence flow or a DFD diagram.
The state machine can explore various states of a system.	Flowchart deal with paths and control flow.

JAVA APPLET

- An applet is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser
- It runs inside the web browser and works at client side
- Applets are used to make the web site more dynamic and entertaining
- Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
- In general, execution of an applet does not begin at `main()` method.

- There are two types of applets:
 - These applets use the Abstract Window Toolkit (AWT) to provide the graphic user interface (or use no GUI at all). This style of applet has been available since Java was first created
 - The second type of applets are those based on the Swing class **JApplet**. **Swing applets use** the Swing classes to provide the GUI. Swing offers a richer and often easier-to-use user interface than does the AWT

Applet Life Cycle



1. Born on initialization state :

- Applet enters the initialization state when it is first loaded.
- This is achieved by calling **init()**.
- This occurs only once in the applet's life cycle.
- At this stage, we may do the following
 - Create objects needed by the applet
 - Initialize variables
 - Load images or fonts.
 - Setup colors.

2. Running state :

- Applet enters this state when the system calls the **start()** method
- This occurs automatically after the applet is initialized(**init()**)
- Starting can also occur if the applet is already in stopped(Idle) state.
- **start()** can be called more than once.
- **start()** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start()**.

3. Idle state or Stopped state :

- An applet becomes idle when it is stopped from running. We can do so by calling `stop()` explicitly.
- Stopping occurs automatically when we leave the page containing the currently running applet.
 - Example : when it goes to another page.
- If the user returns to the page, we can restart them by calling **`start()`**.

4. Dead or destroyed state :

- An applet is said to be dead when it is removed from memory.
- This occurs by invoking **`destroy()`**.
- At this point, we should free up any resources the applet may be using. The **`stop()`** method is always called before **`destroy()`**.
- This occurs only once in the applet life cycle.

5. Display state:

- Applet moves to this state whenever it has to perform some output operations on the screen.
- This happens immediately after the applet enters into the running state.
- **paint()** is called to accomplish this task.
- The **paint()** method is called each time our applet's output must be redrawn.
- This situation can occur for several reasons. For example,
 - the window in which the applet is running may be overwritten by another window and then uncovered.
 - the applet window may be minimized and then restored.
- The **paint()** method has one parameter of type **Graphics**. This parameter describes the graphics environment in which the applet is running.

Java Application vs. Applet

Java Application	Applet
Java application contains a main method	An applet does not contain a main method
Does not require internet connection to execute	Requires internet connection to execute
Is stand alone application	Is a part of web page
Can be run without a browser	Requires a Java compatible browser
Uses stream I/O classes	Use GUI interface provided by AWT or Swings
Entry point is main method	Entry point is init method
Generally used for console programs	Generally used for GUI interfaces