

## **Module 2**

### **Requirement Engineering**

#### **Object-Oriented Process**

**1. Focus on Objects:**

- The approach is all about using objects (like real-world items) to build systems. For example, in a banking system, a "customer" is an object.

**2. Understand the System:**

- First, you look at what the system needs to do. This is similar to other development methods—gathering requirements and understanding what users want.

**3. Identify Objects:**

- You find and define the key objects in the system (like customer, account, etc.) and their characteristics.

#### **Object Modeling Steps**

**1. System Analysis:**

- Figure out what the system should achieve by gathering information about user needs.

**2. System Design:**

- Plan how all the parts of the system will work together.

**3. Object Design:**

- Design the specific objects you identified, including their properties (attributes) and actions (methods).

**4. Implementation:**

- Write the actual code to create the system based on your designs

#### **Advantages of Object-Oriented Methodology**

**? Closer to Real-World:**

- The design closely represents real-world entities and their relationships, making it easier for developers and stakeholders to understand the system and its requirements.

**? Flexibility with Changes:**

- Objects are designed to be independent, allowing for easier modifications when requirements change. You can update one object without affecting others, leading to better adaptability.

#### 🔍 **Encourages Reuse:**

- Object-oriented designs promote the reuse of existing modules or objects in new applications. This reduces development costs and cycle time, as you don't need to create everything from scratch.

#### 🔍 **Intuitive Design:**

- Since the system's functioning is directly mapped to real-world processes, it becomes more intuitive to work with, helping both developers and users understand the system better.

## **Rumbaugh Object-Modeling Technique (OMT)**

### **1. What is OMT?**

- OMT is a modeling language created by Jim Rumbaugh and his team for designing software using an object-oriented approach. It helps in analyzing, designing, and implementing systems.

### **2. Benefits of OMT:**

- **Fast and Intuitive:** OMT makes it easier to identify and model the objects in a system quickly.

## **Purposes of Modeling with OMT**

- **Testing Before Building:** Allows for simulation of physical entities to identify issues early.
- **Communication:** Helps in discussing ideas and designs with customers and stakeholders.
- **Visualization:** Provides alternative ways to present information clearly.
- **Complexity Reduction:** Simplifies the system by breaking it down into manageable models.

## **Three Main Types of Models in OMT**

### **1. Object Model:**

- Focuses on the main concepts like classes (blueprints for objects) and associations (relationships between objects).
- Includes attributes (data) and operations (methods).

- Defines relationships like aggregation (whole-part) and generalization (inheritance).

## **2. Dynamic Model:**

- Represents how the system behaves over time, showing states (conditions) and transitions (changes between states).
- Includes events that trigger these transitions and actions that occur within states.

## **3. Functional Model:**

- Deals with the processes within the system, similar to data flow diagrams.
- Key concepts include processes (tasks), data stores (where data is kept), data flow (movement of data), and actors (users or other systems interacting with the system).

### **Relationship to UML**

- OMT is a predecessor of the Unified Modeling Language (UML), which is widely used today for modeling software systems.

## **Booch Methodology**

The Booch Methodology is an object-oriented software development approach created by Grady Booch that emphasizes the analysis and design phases of software engineering. It is structured around two main processes: macro (high-level) and micro (low-level).

### **Explanation**

#### **1. Macro Process:**

- This is the big-picture view of the software development lifecycle. It outlines the activities of the entire development team as a cohesive unit. The macro process includes phases like requirements gathering, system design, and overall project management.

#### **2. Micro Process:**

- In contrast, the micro process focuses on the specific technical tasks that individual developers and team members perform. This includes writing code, designing classes, and implementing algorithms. It provides a detailed view of how components of the system are built.

#### **3. Focus on Analysis and Design:**

- The Booch Methodology places a strong emphasis on the analysis and design phases, providing tools and techniques for creating detailed models of the

system. This helps in understanding requirements and designing a robust architecture.

- However, it does not give much detail about the implementation (coding) or testing phases. This means that while it's great for planning and design, it assumes that teams will handle coding and testing according to other practices or methodologies.

## **JACOBSON OOSE**

- **What is OOSE?**

Object-Oriented Software Engineering (OOSE) is a method for designing software using object-oriented programming principles.

- **Use Cases:**

OOSE was the first methodology to use use cases, which describe how users will interact with the system. This helps focus on user needs.

- **Phases of Development:**

OOSE includes several key phases:

- **Requirements:** Identify what the system should do.
- **Analysis:** Understand the necessary objects and their relationships.
- **Design:** Plan the system's architecture and object interactions.
- **Implementation:** Write the code for the system.
- **Testing:** Ensure the system meets the requirements and works correctly.

## **Requirement Elicitation (Gathering)**

- **What is a Requirement?**

A requirement is a feature that the system must have or a constraint it must meet to be accepted by the client.

- **Purpose of Requirements Engineering:**

The goal is to define the system's requirements clearly.

- **Main Activities in Requirements Engineering:**

1. **Requirements Elicitation:** Specifying the system in a way that clients can understand.
2. **Analysis:** Creating an analysis model that developers can work with.

- **Importance of Communication:**

Requirements elicitation involves communication between developers and users to define the new system.

- If communication fails, the system is likely to fail.
- Errors made during this phase are costly to fix, often discovered late in the development process—sometimes not until delivery.
- **Methods for Elicitation:**  
Various methods aim to enhance communication among developers, clients, and users.
- **Requirement Specification:**  
Together, the client, developers, and users identify a problem and define a system to address it.

## Requirements Elicitation Activities

1. **Identifying Actors:**
  - Developers determine the different types of users who will interact with the system.
2. **Identifying Scenarios:**
  - Developers observe users to create detailed scenarios that illustrate typical functionalities of the future system.
  - These scenarios help in understanding the application domain in depth.
3. **Identifying Use Cases:**
  - After agreeing on scenarios, developers derive use cases from them.
  - Use cases describe all possible interactions with the system, acting as abstractions for the scenarios.
4. **Refining Use Cases:**
  - Requirements specification is finalized by detailing each use case, including how the system behaves under errors and exceptional conditions.
5. **Identifying Relationships Among Use Cases:**
  - Developers identify dependencies between use cases and consolidate them by combining common functionalities.
6. **Identifying Nonfunctional Requirements:**
  - Developers, users, and clients agree on aspects like performance, documentation, resource consumption, security, and overall quality that are important but not directly related to functionality.

## Requirements Elicitation Techniques

- **Questionnaires:** Gathering information by asking users a set of predefined questions.
- **Task Analysis:** Observing users in their actual working environment to understand their needs.
- **Scenarios:** Describing how the system will be used through a series of interactions between a user and the system.
- **Use Cases:** Abstractions that represent a class of scenarios and describe how the system will respond in various situations.

## Summary

Requirements elicitation involves identifying users, scenarios, use cases, and nonfunctional requirements, utilizing various techniques to ensure comprehensive understanding and communication among stakeholders.

## Functional Requirements

- **Definition:**  
Functional requirements outline how the system should interact with its environment. They specify what the system must do, focusing on the functionality without detailing how it will be implemented.
- **Key Characteristics:**
  - They describe specific behaviors or functions the system must perform.
  - They are independent of the implementation details, meaning they focus on the "what" rather than the "how."
- **Example:**  
**SatWatch:** A watch that automatically resets itself without requiring any action from the user. This describes the functionality without explaining how it achieves this.

## Non-Functional Requirements

- **Definition:**  
Non-functional requirements describe aspects of the system that are not directly related to its functional behavior. They address how the system performs its functions rather than what functions it performs.

## Types of Non-Functional Requirements

1. **Usability:**
  - Refers to how easy it is for users to learn and operate the system.

- **Usability Requirements:** May include features like online help, user documentation, and user-friendly design elements (e.g., color schemes for the GUI).

## 2. **Reliability:**

- The ability of the system to perform required functions under specified conditions for a certain time.
- Reliability often encompasses dependability, which includes robustness and safety.

## 3. **Performance:**

- Concerns quantifiable aspects of the system, such as:
  - **Response Time:** How quickly the system responds to user inputs.
  - **Throughput:** The amount of work the system can handle in a given time.
  - **Availability:** The system's uptime and readiness for use.
  - **Accuracy:** The correctness of the system's outputs.

## 4. **Supportability:**

- Focuses on the ease of making changes to the system after it has been deployed.
- **Adaptability:** Ability to adjust to new concepts in the application domain.
- **Maintainability:** Ease of updating the system to accommodate new technology or fix defects.
- **Internationalization:** Ability to support various languages and regional formats.

## **Additional Non-Functional Requirements**

- **Implementation Requirements:**
  - Constraints on how the system should be implemented, including specific tools, programming languages, or hardware.
- **Interface Requirements:**
  - Constraints imposed by external systems, including data formats for interoperability.
- **Operations Requirements:**
  - Constraints related to the administration and management of the system.
- **Packaging Requirements:**

- Constraints on the delivery of the system, including installation media.
- **Legal Requirements:**
  - Concerns with licensing, regulations, and certification issues.
- **Quality Requirements:**
  - Standards that the system must meet regarding its performance and reliability.

## **Requirements Characteristics**

- **Continuous Validation:**  
Requirements are regularly checked with the client and users to ensure they meet their needs.
- **Validation Process:**  
This involves verifying that the requirements are complete, consistent, unambiguous, and correct.

## **Key Characteristics**

1. **Completeness:**
  - Requirements are complete if they describe all possible scenarios, including exceptional cases.
  - **Example of Incompleteness:** If SatWatch doesn't mention handling state boundaries, it's incomplete.
  - **Solution:** Add a feature to address state boundaries.
2. **Consistency:**
  - Requirements are consistent if there are no contradictions within the specification.
  - **Example of Inconsistency:** If one requirement states the system will have software faults while another requires upgrading mechanisms, they conflict.
  - **Solution:** Revise one of the conflicting requirements.
3. **Unambiguity:**
  - Requirements are unambiguous if they clearly define only one system interpretation.
  - **Example of Ambiguity:** If it's unclear whether SatWatch accounts for daylight saving time, it's ambiguous.
  - **Solution:** Specify that SatWatch must handle daylight saving time.



#### 4. **Correctness:**

- Requirements are correct if they accurately represent what the client needs and what developers intend to build.
- **Correct:** The requirements should only describe relevant features, not unintended ones.

### **Requirement Specification**

- **Desirable Properties of a Requirements Specification:**

- A good requirements specification should be realistic, verifiable, and traceable.

#### 1. **Realistic:**

- The requirements are realistic if the system can be implemented within the given constraints, such as time, budget, and technology.

#### 2. **Verifiable:**

- Requirements are verifiable if, after the system is built, you can conduct repeatable tests to confirm that the system meets the specified requirements.

#### 3. **Traceable:**

- Requirements are traceable if each one can be followed throughout the software development process back to its associated system functions, and vice versa.
- Traceability also involves tracking dependencies among requirements, system functions, and intermediate designs.

### **Use Case Diagrams**

- **Identifying Actors:**

- **Definition:** Actors are external entities that interact with the system. They can be either human users or other systems.

- **Naming Actors:**

- Use meaningful, business-relevant names for actors.
- Primary actors should be positioned on the left side of the diagram to emphasize their importance.

- **Modeling Roles:**

- Actors represent roles rather than specific positions. For example, in a hotel, both the front office executive and the shift manager can make reservations. Instead of naming them individually, you could use "Reservation Agent" to represent this role.
- **External Systems as Actors:**
  - If your use case involves an external system (like email management software), that software is also considered an actor. For example, in a use case for sending an email, the email software is an actor.
- **Interaction Rules:**
  - Actors do not interact with each other; they only interact with the system.
- **Inheritance of Actors:**
  - Place inheriting (sub) actors below their parent actor in the diagram.
- - After identifying actors, the next step is to determine what functionalities each actor can access.
  - Use cases provide an external (user) view of the system and model the interactions between users and the system.

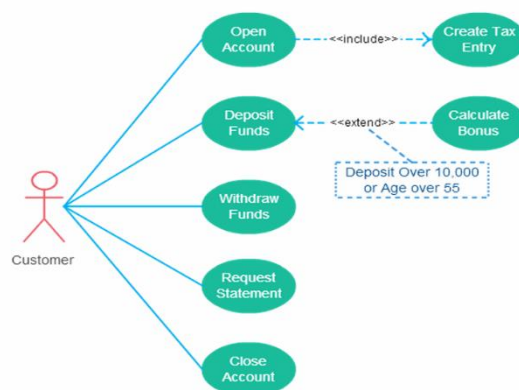
## Identifying Scenarios

- **Definition of a Scenario:**
  - A scenario is a narrative that describes how users interact with the system and what they experience.
  - It captures the system's behavior from a single actor's or user's perspective.
  - Scenarios can be created for various phases of the software development process, such as current usage, future development, evaluation, or training.

## Identifying Use Cases

- **Purpose:**
  - Use cases represent the actions that actors want the system to perform.
  - A use case illustrates a complete flow of events, capturing functions visible to the actor and achieving specific goals (like reading, writing, or modifying data).
- **Naming Use Cases:**
  - Use case names should start with a verb, reflecting the action (e.g., "Print Invoice" instead of just "Print").

- Names should be descriptive to provide clarity for anyone reviewing the diagram.
- **Logical Order:**
  - Arrange use cases in a logical sequence to make sense of the workflow. For instance, typical banking use cases might include "Open Account," "Deposit," and "Withdraw."
- **Diagram Structure:**
  - Place included use cases to the right of the invoking use case to enhance readability.
  - Inheriting use cases should be placed below their parent use cases to maintain clarity in the diagram.



#### Elements of use case diagram: Actor

Actor



#### Elements of use case diagram: Use Case

- System function (process – automated or manual).



Use Case

#### Elements of use case diagram

———— Connection between Actor and Use Case

□ Boundary of system

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

## 1) Association Between Actor and Use Case

- **Definition:**

This relationship is fundamental in every use case diagram, representing how actors interact with use cases.

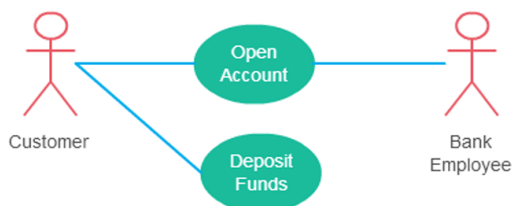
- **Key Points:**

- **Mandatory Association:** Each actor must be associated with at least one use case to illustrate their role in the system.
- **Multiple Associations:** An actor can be linked to multiple use cases, reflecting various interactions they have with the system.
- **Shared Use Cases:** Multiple actors can be associated with a single use case, indicating that different users can perform the same action.

### Summary

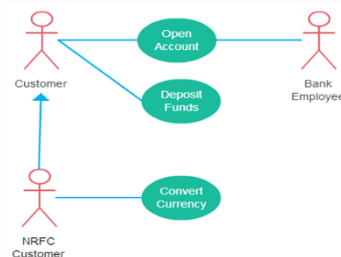
The association between actors and use cases is essential for demonstrating user interactions and ensuring that all roles are clearly defined in the system.

### Association Between Actor and Use Case



### Generalization of an Actor

- Generalization of an actor means that one actor can inherit the role of another actor.

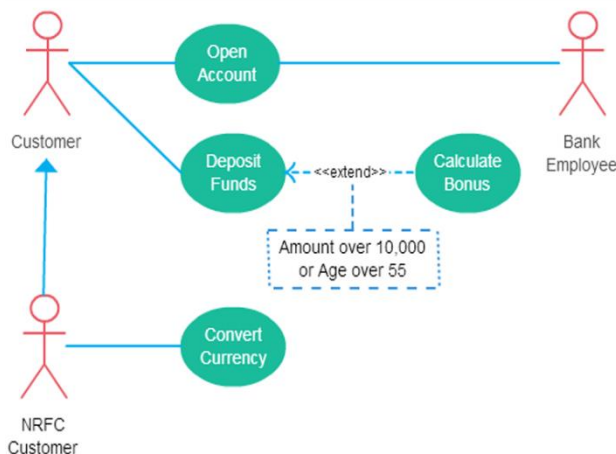


### 3) Extend Relationship Between Two Use Cases

- **Definition:**  
This relationship adds extra functionality to a base use case, enhancing the system's capabilities.
- **Key Points:**
  - **Dependency:** The extending use case relies on the base (extended) use case, meaning it cannot function independently without it.
  - **Optional Functionality:** The extending use case is typically optional and is triggered under specific conditions or scenarios.
  - **Meaningful Base Use Case:** The base use case should be complete and functional on its own, even without the extending use case.
  - **Notation:** In diagrams, the arrow points to the base use case and is labeled with <<extend>> to indicate the relationship.
  - **Optional Nature:** The extending use case is not mandatory for the base use case to operate.

#### Summary

The extend relationship allows for additional, optional functionality within a use case, providing flexibility and scalability in system design.



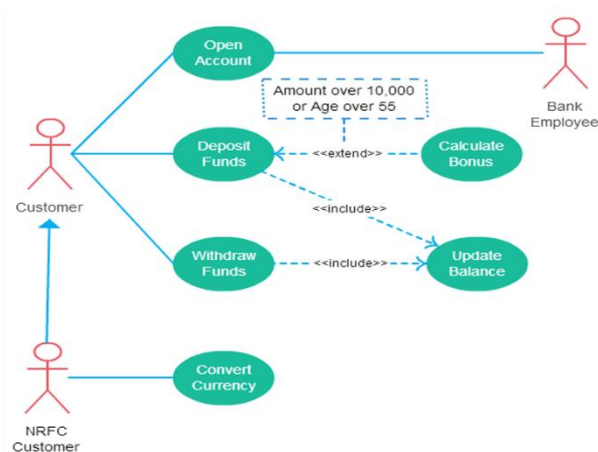
### 4) Include Relationship Between Two Use Cases

- **Definition:**  
The include relationship indicates that the behavior of one use case (the included use case) is a necessary part of another use case (the including or base use case).
- **Key Points:**

- **Dependency:** The base use case is incomplete without the included use case, meaning it cannot function properly on its own.
- **Mandatory Inclusion:** The included use case is required and cannot be omitted; it is essential for the base use case's functionality.
- **Subroutine Analogy:** This relationship is similar to a "use case subroutine," where the included use case performs a specific function that supports the overall process of the base use case.

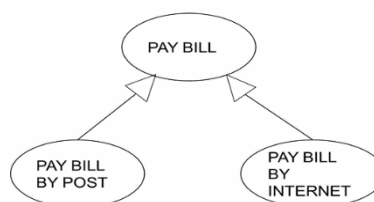
## Summary

The include relationship emphasizes essential functionality within a use case, ensuring that key processes are integrated and mandatory for the successful execution of the system's behavior.



## 5) Generalization of a Use Case

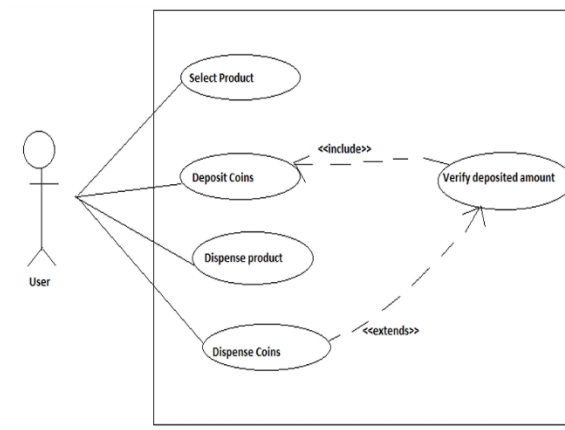
- **Definition:**  
This relationship is similar to the generalization of an actor and is used to represent a hierarchy among use cases.
- **Key Points:**
  - **Common Behavior:** Generalization is applied when two or more use cases share common behaviors or functionalities.
  - **Specialized Behavior:** Each specialized use case can also have unique behaviors that differentiate it from the generalized use case.
  - **Hierarchy:** The generalized use case captures the broader functionality, while the specialized use cases refine or extend this functionality for specific scenarios.



A vending machine accepts coins for a variety of products. The user selects the drink from products available through the selection panel. If the drink is available the price of the product is displayed. The user then deposits the coins depending on the price of the product. Coin collector collects the coins. After stipulated time, the controller will compare the deposited coins with the price. If the amount deposited is less than the price then error message will be displayed and all deposited coins will be dispensed by coin dispenser else the drink will be dispensed by the product dispenser. Draw use case diagram

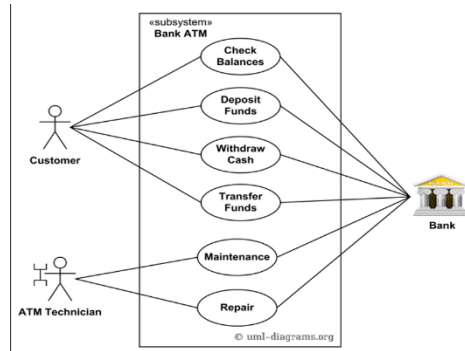
## Use Case Diagram for Vending Machine

- Actor- User
- Processes involved in the system
  - Select product
  - Deposit coins
  - Verify the deposited amount
  - Dispense product
  - Dispense coins

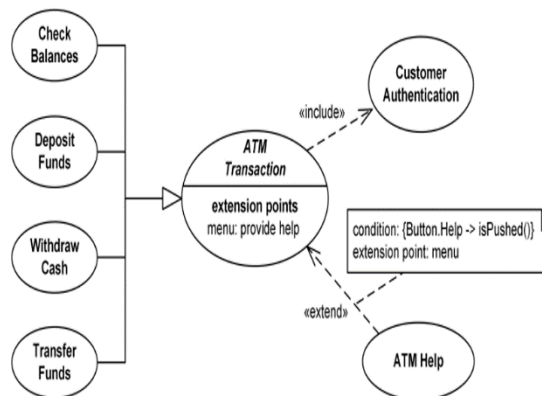


### Use Case Example

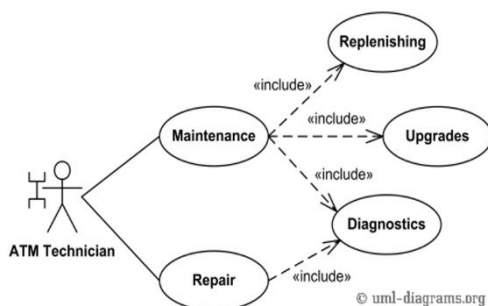
- Customer (actor) uses bank ATM to Check Balances of his/her bank accounts, Deposit Funds, Withdraw Cash and/or Transfer Funds (use cases). ATM Technician provides Maintenance and Repairs. All these use cases also involve Bank actor whether it is related to customer transactions or to the ATM servicing.



On most bank ATMs, the customer is authenticated by inserting a plastic ATM card and entering a personal identification number (PIN). Customer Authentication use case is required for every ATM transaction so we show it as include relationship. Including this use case as well as transaction generalizations make the ATM Transaction an abstract use case. Customer may need some help from the ATM. ATM Transaction use case is extended via extension point called menu by the ATM Help use case whenever ATM Transaction is at the location specified by the menu and the bank customer requests help, e.g. by selecting Help menu item.



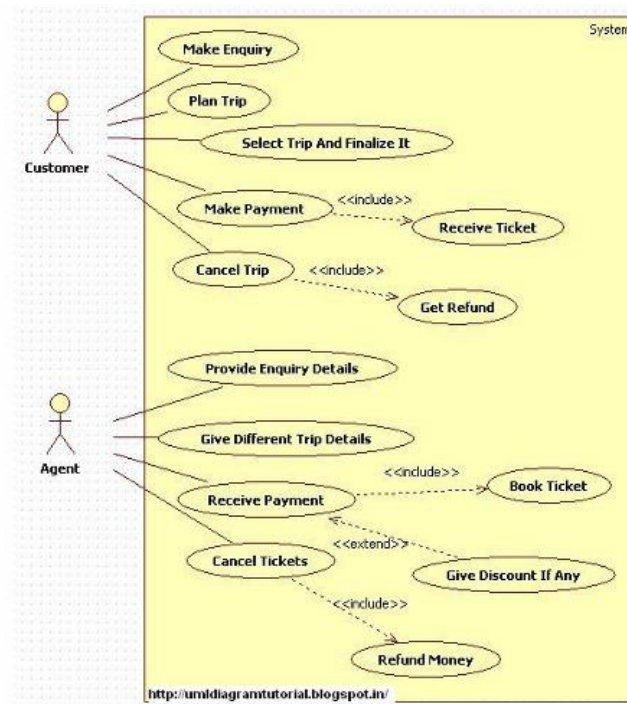
- ATM Technician maintains or repairs Bank ATM. Maintenance use case includes Replenishing ATM with cash, ink or printer paper, Upgrades of hardware, firmware or software, and remote or on-site Diagnostics. Diagnostics is also included in (shared with) Repair use case.



*Bank ATM Maintenance, Repair, Diagnostics Use Cases Example.*



1. A leading TRAVEL AGENCY has decided to develop application package to help its customer in planning tours. The agency provides services like tour, air, railway, luxury coach, hotel booking etc. Many a times customers do not have idea of availability of transport services to a particular destination. The agency also gives advice regarding economical planning of vacation/tour. Given the tour constraints like number of days, affordable cost and places to visit the software should present alternative tour plans. Alternatively the software may be just used for querying to know availability of transport services, hotels etc. Besides this main objective of this software should also have facilities for billing and accounting for the agency. You are appointed as a consultant to develop implementation strategy for Automated Tourist System. Draw use case and class diagram.



## Online Airline Reservation System



## Activity Diagram

- An activity diagram describes the behavior of a system in terms of activities.
- Activity diagram is basically a flow chart to represent the flow from one activity to another activity.
- Activity diagram: operation of a system.
- This flow can be sequential, branched or concurrent

### Activity Diagram (basic notations)

- **Initial node.**

- ✓ The filled in circle is the starting point of the diagram.
- ✓ It is shown as filled circle.



### Activity Diagram (basic notations)

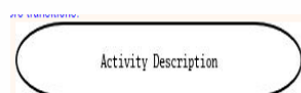
- **Activity final node.**

- ✓ The filled circle with a border is the ending point.
- ✓ The final activity is optional.



### Activity Diagram (basic notations)

- **Activity.** The rounded rectangles represent activities that occur.



### Activity Diagram (basic notations)

- **Flow/edge (Control Flow)**

- ✓ The arrows on the diagram. Within the control flow an incoming arrow starts a single step of an activity; after the step is completed the flow continues along the outgoing arrow.



## Decision in Activity Diagrams

- **Definition:**

Decisions represent branching points in the control flow of an activity diagram.

- **Key Features:**

- **Decisions as Control Points:**

In an Activity Diagram, decisions are represented as **diamond shapes** that show where the flow of control can branch based on certain conditions.

- **Incoming and Outgoing Arrows:**

**Incoming Arrows:** These represent the flow of activities leading into the decision point.

**Outgoing Arrows:** Each arrow represents a possible path the flow can take depending on the outcome of the decision. For example, different paths might be taken based on whether a user is authenticated or not.

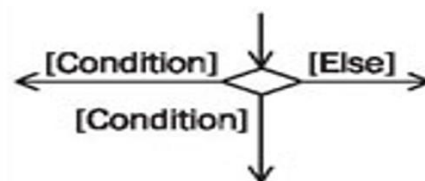
- **Labeling Conditions:**

Each outgoing edge from the decision point is labeled with the conditions that determine which path to follow. This makes it clear what criteria are used to select a particular branch.

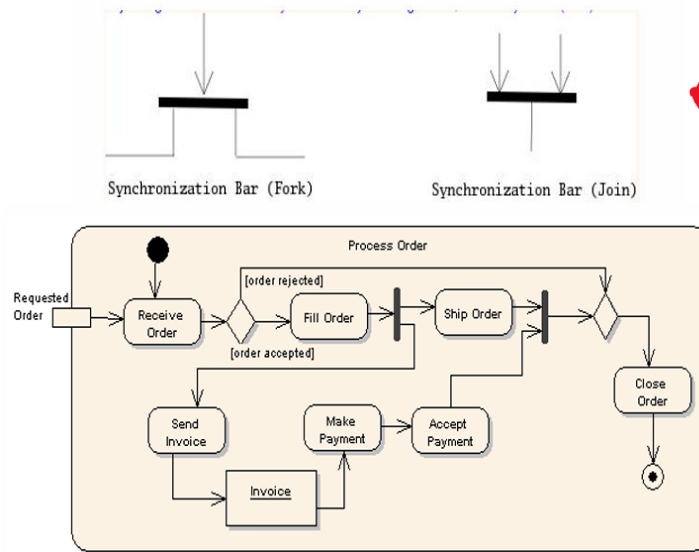
- **Set of Possible Outcomes:**

The outgoing edges collectively represent all possible outcomes from that decision point, illustrating the various paths the workflow can take depending on the evaluated conditions.

### Activity Diagram (basic notations)



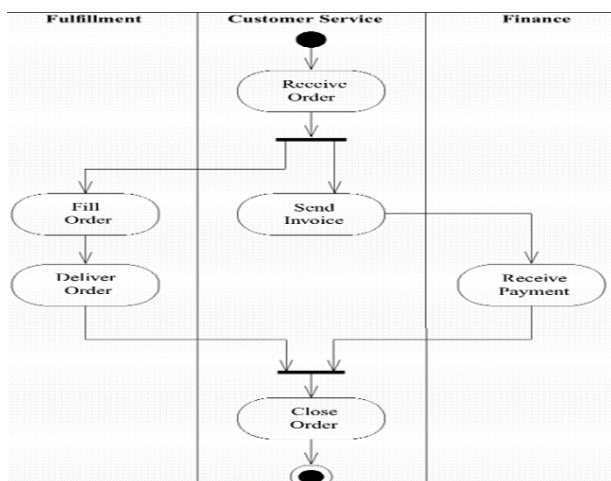
- Some activities occur simultaneously or in parallel. Such activities are called concurrent activities.
- Fork nodes and join nodes represent concurrency.
- Fork nodes denote the splitting of the flow of control into multiple threads.
- Join nodes denotes the synchronization of multiple threads and their merging of the flow of control into a single thread



This should be present in every Activity Diagram

## Swim Lanes

- The contents of an activity diagram may be organized into partitions (swimlanes) using solid vertical lines.
- A swim lane (or swimlane diagram) is a visual element used in process flow diagrams, or flowcharts, that visually distinguishes job sharing and responsibilities for sub-processes of a business process
- Order Swimlanes in a Logical Manner.
- Apply Swim Lanes To Linear Processes.
- Have Less Than Five Swimlanes.
- Consider Swim areas For Complex Diagrams



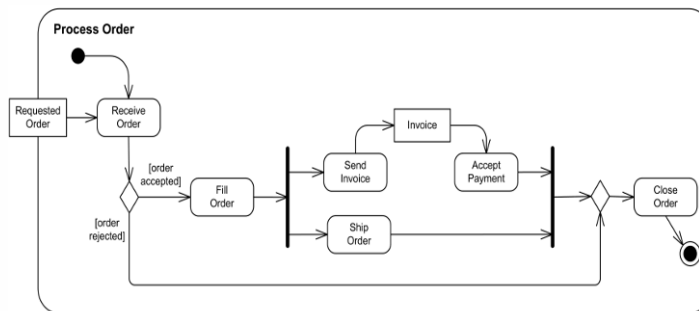
We need to draw at least 3 swimlanes

## Problem

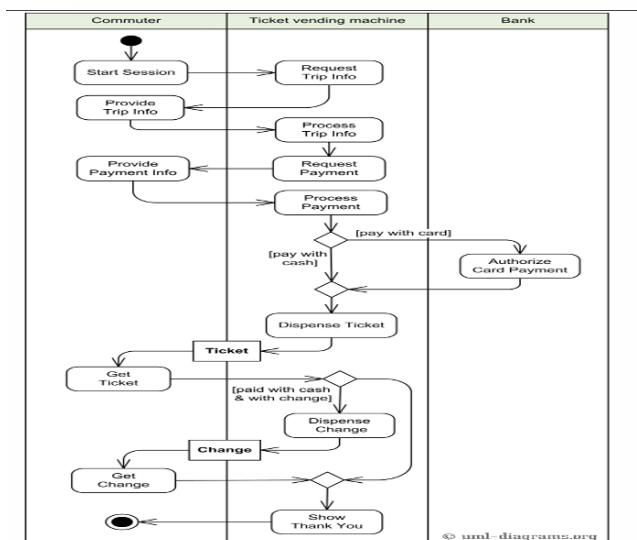
- Activity Diagram for Ticket Vending Machine
- Problem Definition: Draw Activity Diagram for Login Procedure of a system.

## Problem Definition

• Activity is started by Commuter actor who needs to buy a ticket. Ticket vending machine will request trip information from Commuter. This information will include number and type of tickets, e.g. whether it is a monthly pass, one way or round ticket, route number, destination or zone number, etc. Based on the provided trip info ticket vending machine will calculate payment due and request payment options. Those options include payment by cash, or by credit or debit card. If payment by card was selected by Commuter, another actor, Bank will participate in the activity by authorizing the payment. After payment is complete, ticket is dispensed to the Commuter. Cash payment might result in some change due, so the change is dispensed to the Commuter in this case. Ticket vending machine will show some "Thank You" screen at the end of the activity



- An example of business flow activity to process purchase order.



## Class Diagram

A Class Diagram is a diagram describing the structure of a system.

- It shows the system's
  - ✓ classes
  - ✓ Attributes
  - ✓ operations (or methods),
  - ✓ Relationships among the classes.

## Essential Elements of a Class Diagram

- Class: A class represents the blueprint of its objects.
- Attributes
- Operations
- Relationships
  - Associations
  - Generalization
  - Realization
  - Dependency

### **Class Definition**

#### **1. Description of a Class:**

- A class is a template that describes a group of objects sharing common characteristics, including:
  - **Attributes (Status):** These are the properties or data members that hold the state of an object. For instance, in a Car class, attributes might include color, model, and year.
  - **Operations (Behavior):** These are the functions or methods that define what actions can be performed on or by the objects of the class. For the Car class, operations could include drive(), stop(), and refuel().
  - **Relationships with Other Classes:** Classes can interact with one another through various relationships, such as associations, inheritance, and dependencies, as previously discussed.

### **Visibility Modifiers**

#### **2. Visibility of Attributes and Operations:**

- In UML Class Diagrams, you can specify the visibility of attributes and operations using the following symbols:

- **"+" (Public):** The attribute or operation is accessible from any other class. For example, a public method can be called by any object.
- **"#" (Protected):** The attribute or operation is accessible within its own class and by subclasses (derived classes). This restricts access from other unrelated classes.
- **"-" (Private):** The attribute or operation is only accessible within its own class. This encapsulates the data, ensuring it cannot be accessed directly from outside the class.
- **"~" (Package):** The attribute or operation is accessible to other classes in the same package (namespace), but not to classes outside that package.

## Example

Consider a BankAccount class:

- **Class: BankAccount**
  - **Attributes:**
    - -accountNumber (private)
    - +balance (public)
    - #accountHolder (protected)
  - **Operations:**
    - +deposit(amount) (public)
    - -withdraw(amount) (private)
    - #calculateInterest() (protected)

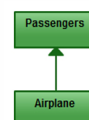
## Associations

- An association between two classes indicates that objects at one end of an association “recognize” objects at the other end and may send messages to them.
- Example: “An Employee works for a Company”



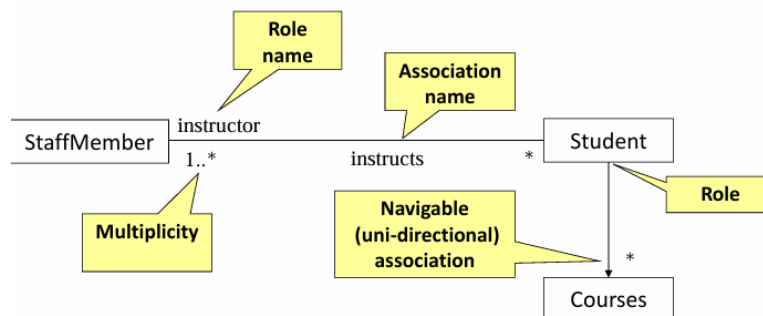
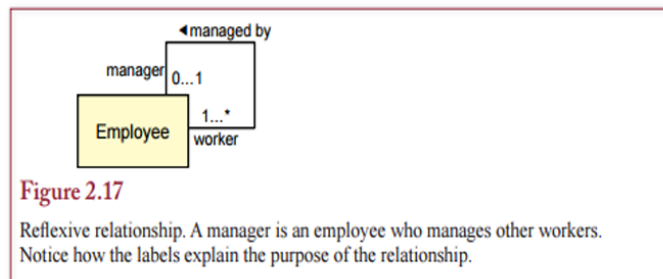
## Directed Association

- Refers to a **directional relationship** represented by a line with an arrowhead. The arrowhead depicts a container-contained directional flow.



## Reflexive Association

- A reflexive association is a relationship from one class back to itself.



## Association Class

### 1. Definition:

- An association class is used when you need to add valuable information about the relationship between two classes.

### 2. Purpose:

- It allows you to encapsulate attributes or operations specific to the association itself.

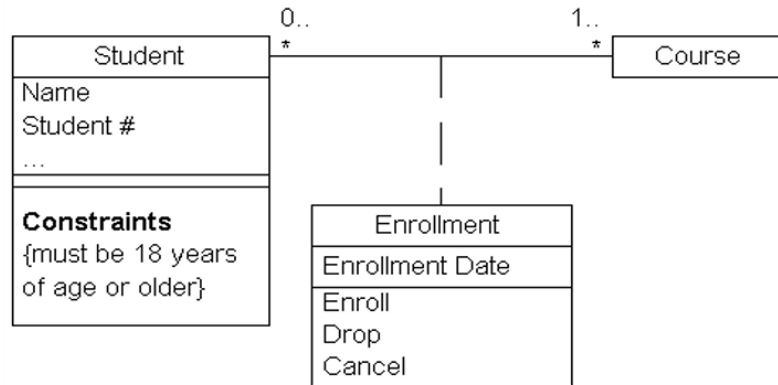
### 3. Visual Representation:

- Represented like a normal class.
- The association line between the primary classes intersects with a **dotted line** connecting to the association class.

### Example

For instance, if you have a **Student** and a **Course**, you might have an association class called **Enrollment** that includes attributes like **enrollmentDate** or **grade**.





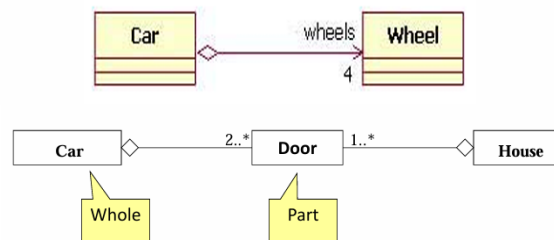
## Multiplicity

- the number of objects that participate in the association.
- Multiplicity Indicators**

Exactly one	1
Zero or more (unlimited)	* (0..*)
One or more	1..*
Zero or one (optional association)	0..1
Specified range	2..4
Multiple, disjoint ranges	2, 4..6, 8

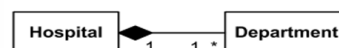
## Basic Aggregation

- Aggregation is a special type of association used to model a **"whole to its parts"** relationship.
- Example: Car as a whole entity and Car Wheel as part of the overall Car.



## Composition Aggregation

- A strong form of aggregation
  - The whole is the sole owner of its part.
    - The part object may belong to only one whole
  - Multiplicity on the whole side must be zero or one.
  - The life time of the part is dependent upon the whole.



*Hospital has 1 or more Departments, and each Department belongs to exactly one Hospital. If Hospital is closed, so are all of its Departments.*

## Generalization

### 1. Definition:

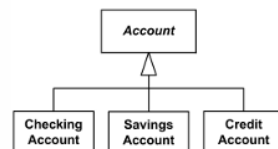
- Generalization is a relationship where a subclass (child) inherits attributes and operations from a superclass (parent). This represents an "is a" relationship.

### 2. Purpose:

- It allows for code reusability and a hierarchical classification of classes. For example, a Dog class is a subclass of an Animal class.



*Checking, Savings, and Credit Accounts are **generalized** by Account*



*Checking, Savings, and Credit Accounts are **generalized** by Account*

## Realization

- A realization relationship indicates that one class implements the behavior defined by another class, typically an interface or protocol.

## Dependency

### 1. Definition:

- A dependency relationship in UML indicates that one element (the client) relies on another element (the supplier) to fulfill its responsibilities.

### 2. Purpose:

- It signifies that changes in the supplier may affect the client. The client depends on the supplier for its functionality.

### 3. Example:

- **Client:** Cart
- **Supplier:** Product
- The Cart class depends on the Product class because it uses Product as a parameter in methods like addProduct(Product).

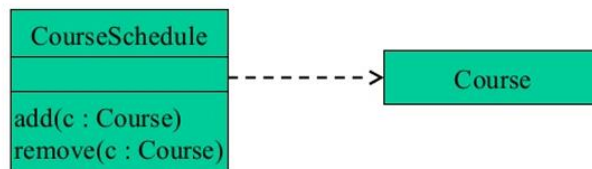
### 4. Visual Representation:

- Represented by a **dashed line** with an open arrow pointing from the client to the supplier.

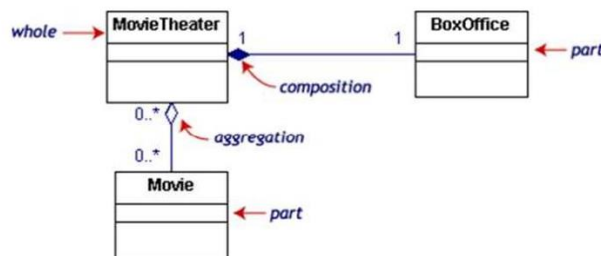


## Dependency Relationships

A **dependency** indicates a semantic relationship between two or more elements. The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.

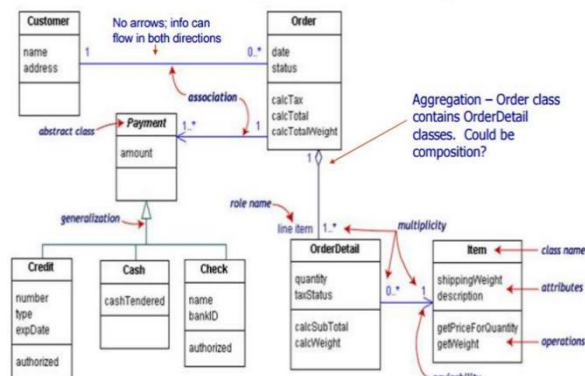


## Composition/aggregation example

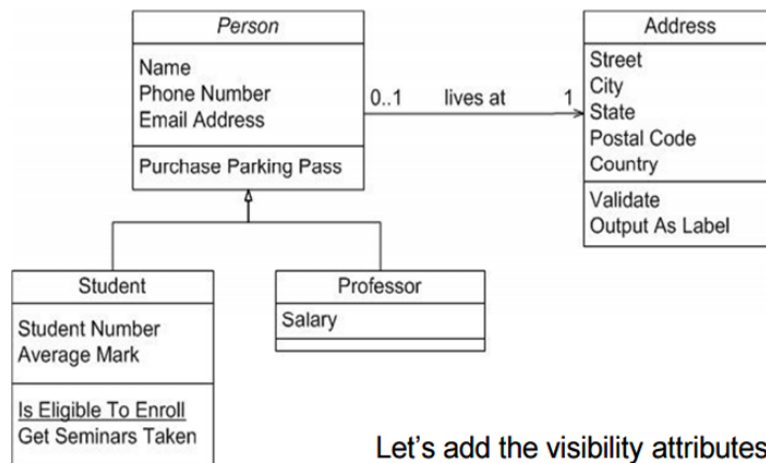


If the movie theater goes away  
so does the box office => composition  
but movies may still exist => aggregation

## Class diagram example



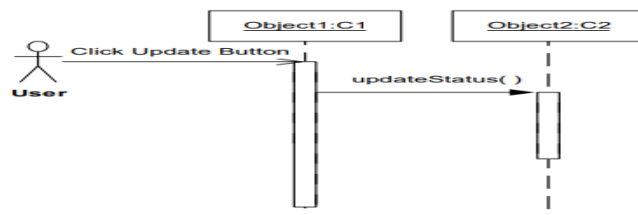
## UML example: people



## Sequence Diagram

### Elements of Sequence Diagram

- **ACTOR**
- **OBJECTS**
- **LIFELINES**
- **MESSAGES**
- **ACTIVATION**-represents time an object needs to complete task



Class Roles or Participants or Objects: Class roles describe the way an object will behave in context.



#### Elements of Sequence Diagram



Activation or Execution Occurrence

- **Activation or Execution Occurrence**

Activation boxes represent the time an object needs to complete a task.

- When an object is busy executing a process or waiting for a reply message, use a thin gray rectangle placed vertically on its lifeline.

### Synchronous Message

- A synchronous message requires a response before the interaction can continue.
- It's usually drawn using a line with a solid arrowhead pointing from one object to another



### Asynchronous Message

Asynchronous messages don't need a reply for interaction to continue.

- They are drawn with an arrow connecting two lifelines; however, the arrowhead is usually open and there's no return message depicted.



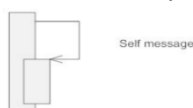
### Reply or Return Message

- A reply message is drawn with a dotted line and an open arrowhead pointing back to the original lifeline.



### Self Message

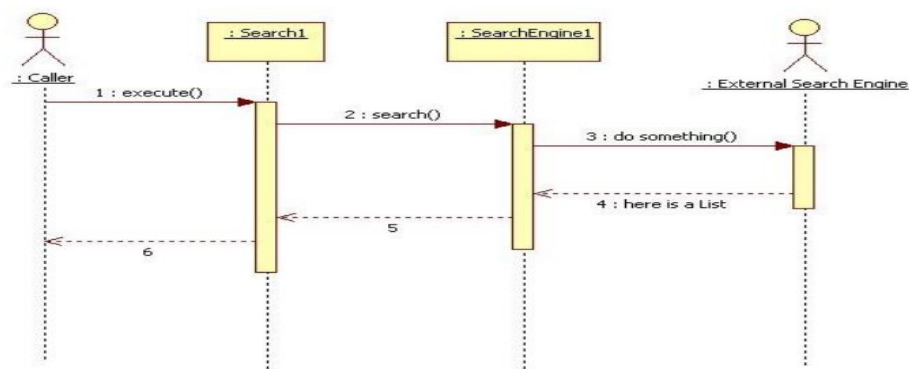
- A message an object sends to itself, usually shown as a U shaped arrow pointing back to itself.



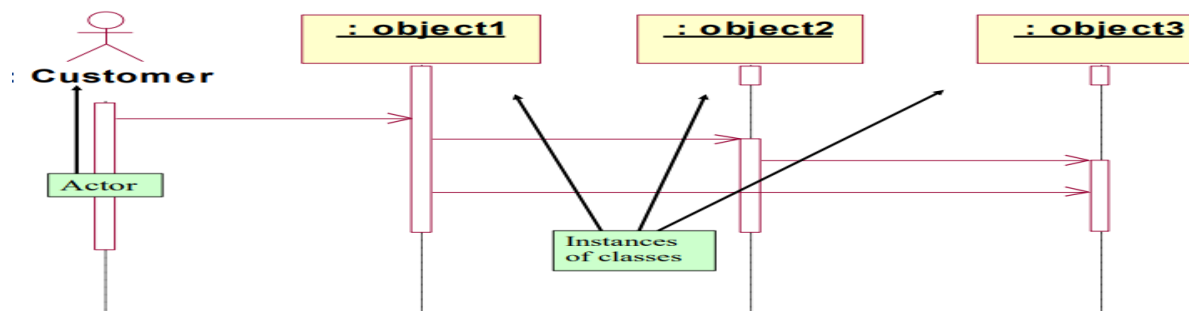
**Lost:** A lost message occurs when the sender of the message is known but there is no reception of the message.



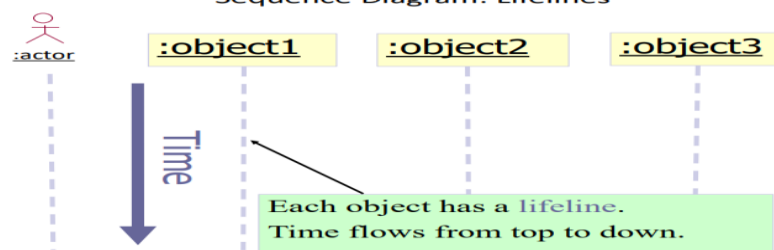
**Found:** A found message indicates that although the receiver of the message is known in the current interaction fragment, the sender of the message is unknown.



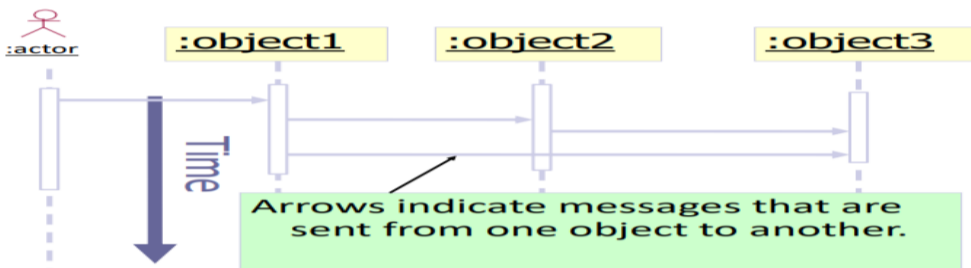
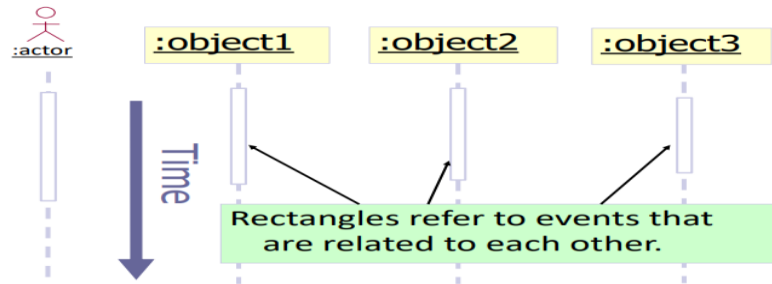
### Sequence diagrams: an example



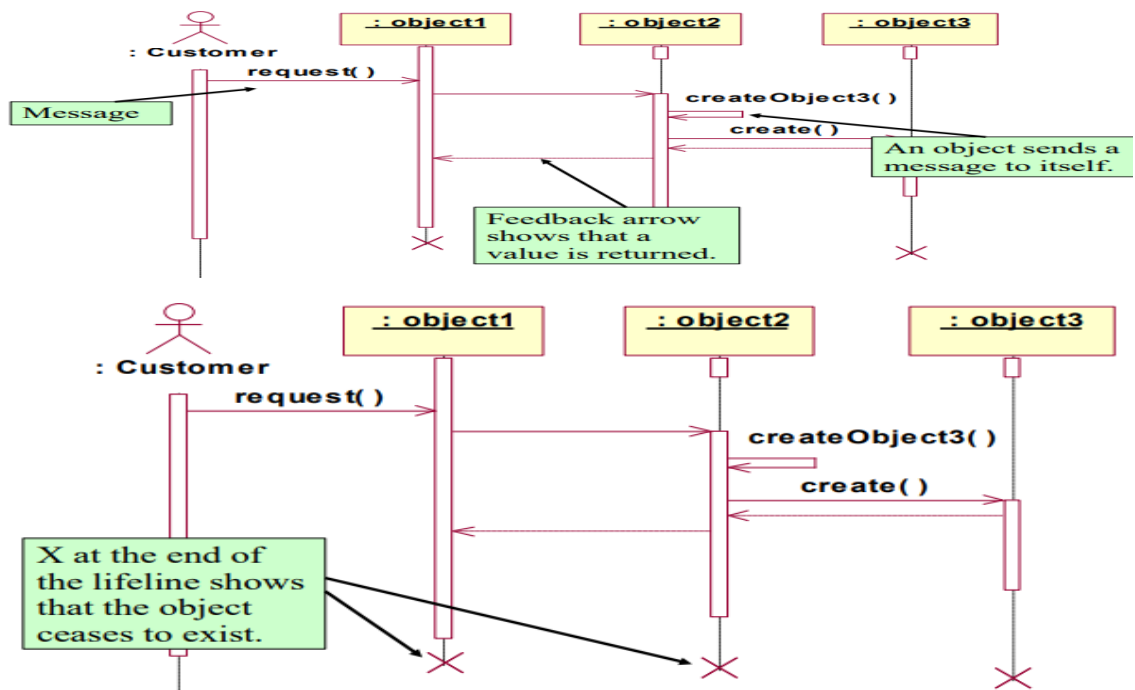
### Sequence Diagram: Lifelines



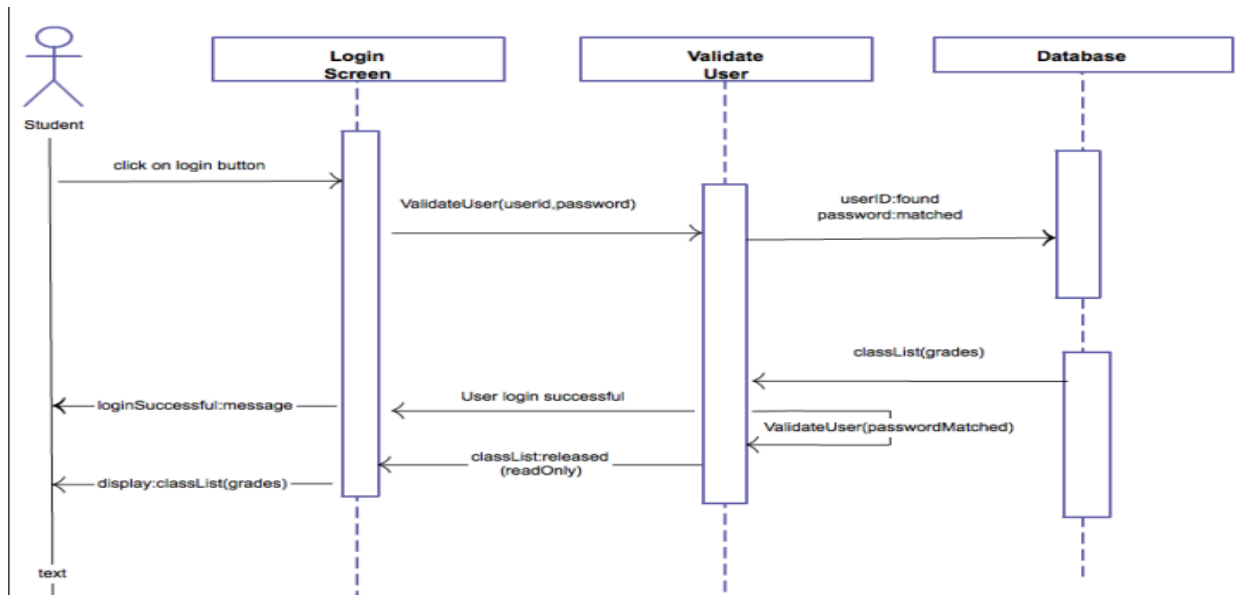
### Sequence diagrams: rectangles



### Sequence diagrams - different types of messages



Draw sequence diagram for login procedure of a system. Include all possible scenarios and also draw activity diagram. (10M)



## Collaboration Diagram

### Elements of a collaboration diagram

- Object

Object Name: Class Name

Object Name: Class Name

- Relation/Association

Function()

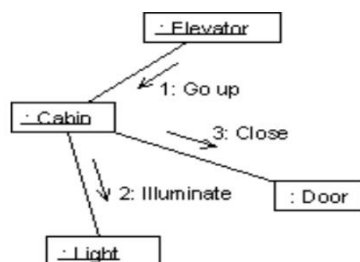
- Messages



### STEPS TO DRAW COLLABORATION DIAGRAM

- Place objects as vertices in a graph
- Add links to connect these object as arcs of graph
- Write messages on links with sequence numbers for send and receive

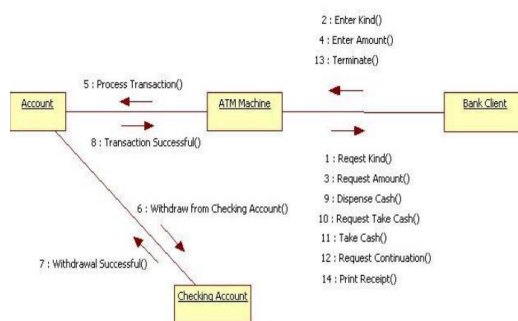
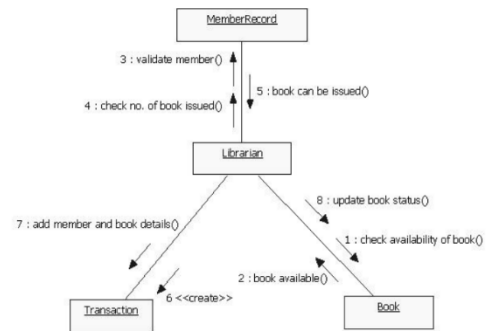
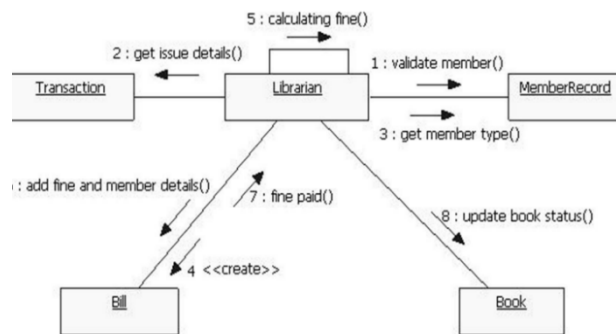
### Ex. Collaboration Diagram



### Collaboration Diagram Syntax

AN ACTOR	
AN OBJECT	
AN ASSOCIATION	
A MESSAGE	





## State Chart Diagram

- A state in UML is a condition or situation an object (in a system) might find itself in during its life time.
- A state chart diagram is normally used to model how the state of an object changes in its lifetime.
- Thus, a state machine diagram does not necessarily model all possible states, but rather the critical ones only. When we say “critical” states, we mean those that act as stimuli and prompt for response in the external world

**Initial state.** This is represented as a filled circle.

**Final state.** This is represented by a filled circle inside a larger circle.

**State.** These are represented by rectangles with rounded corners.


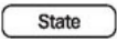
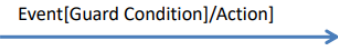

**Transition.** A transition is shown as an arrow between two states. Normally, the name of the event which causes the transition is placed along side the arrow.

A guard condition is a condition that has to be met in order to enable the transition to which it belongs

## [Guard Condition]

Guard conditions can be used to document that a certain event, depending on the condition, can lead to different transitions.

## Elements of state diagram

- Initial State 
- State 
- Transition 
- Final States 

## ATM Withdraw

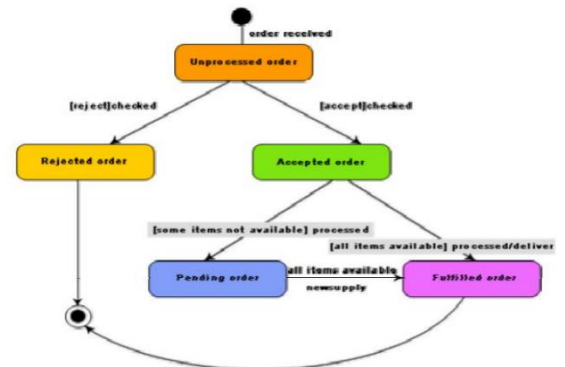
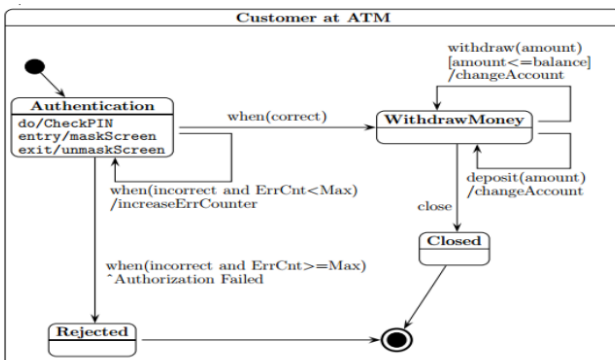


Fig. 7.16: State chart diagram for an order object

