Object Oriented Analysis and Design

Introduction

Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.
Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting logical and physical as well as static and dynamic models of the system under design.

Object Oriented Development life cycle

The Object Oriented Methodology of Building Systems takes the objects as the basis. For this, first the system to be developed is observed and analyzed and the requirements are defined as in any other method of system development. Once this is done, the objects in the required system are identified. For example in case of a Banking System, a customer is an object, a chequebook is an object, and even an account is an object.
Object oriented development life cycle contains:

1. **System Analysis** As in any other system development model, system analysis is the first phase of development in case of Object Modeling too. In this phase, the developer interacts with the user of the system to find out the user requirements and analyses the system to understand the functioning. Based on this system study, the analyst prepares a model of the desired system. This model is purely based on what the system is required to do. At this stage the implementation details are not taken care of. Only the model of the system is prepared based on the idea that the system is made up of a set of interacting objects. The important elements of the system are emphasized.
2. **System Design** System Design is the next development stage where the overall architecture of the desired system is decided. The system is organized as a set of sub systems interacting with each other. While designing the system as a set of interacting subsystems, the analyst takes care of specifications as observed in system analysis as well as what is required out of the new system by the end user. As the basic philosophy of Object-Oriented method of system analysis is to perceive the system as a set of interacting objects, a bigger system may also be seen as a set of interacting smaller subsystems that in turn are composed of a set of interacting objects. While designing the system, the stress lies on the objects comprising the system and not on the processes being carried out in the system as in the case of traditional Waterfall Model where the processes form the important part of the system.
3. **Object Design** In this phase, the details of the system analysis and system design are implemented. The Objects identified in the system design phase are designed. Here the implementation of these objects is decided as the data structures get defined and also the interrelationships between the objects are defined. Object Oriented Philosophy is very much similar to real world and hence is gaining popularity as the systems here are seen as a set of interacting objects as in the real world. To implement this concept, the process-based structural programming is not used; instead objects are created using data structures. Just as every programming language provides various data types and various variables of that type can be created, similarly, in case of objects certain data types are predefined. For example, we can define a data type called pen and then create and use several objects of this data type. This concept is known as creating a class.

   Class: A class is a collection of similar objects. It is a template where certain basic characteristics of a set of objects are defined. The class defines the basic attributes and the operations of the objects of that type. Defining a class does not define any object, but it only creates a template. For objects to be actually created instances of the class are created as per the requirement of the case.

   Abstraction: Classes are built on the basis of abstraction, where a set of similar objects are observed and their common characteristics are listed. Of all these, the characteristics of concern to the system under observation are picked up and the class definition is made. The attributes of no concern to the

system are left out. This is known as abstraction. The abstraction of an object varies according to its application. For instance, while defining a pen class for a stationery shop, the attributes of concern might be the pen color, ink color, pen type etc., whereas a pen class for a manufacturing firm would be containing the other dimensions of the pen like its diameter, its shape and size etc.

Inheritance: Inheritance is another important concept in this regard. This concept is used to apply the idea of reusability of the objects. A new type of class can be defined using a similar existing class with a few new features. For instance, a class vehicle can be defined with the basic functionality of any vehicle and a new class called car can be derived out of it with a few modifications. This would save the developers time and effort as the classes already existing are reused without much change. Coming back to our development process, in the Object Designing phase of the Development process, the designer decides onto the classes in the system based on these concepts. The designer also decides on whether the classes need to be created from scratch or any existing classes can be used as it is or new classes can be inherited from them.

4. **Implementation** During this phase, the class objects and the interrelationships of these classes are translated and actually coded using the programming language decided upon. The databases are made and the complete system is given a functional shape.

The complete OO methodology revolves around the objects identified in the system. When observed closely, every object exhibits some characteristics and behavior. The objects recognize and respond to certain events. For example, considering a Window on the screen as an object, the size of the window gets changed when resize button of the window is clicked. Here the clicking of the button is an event to which the window responds by changing its state from the old size to the new size.

While developing systems based on this approach, the analyst makes use of certain models to analyze and depict these objects. The methodology supports and uses three basic Models:

Object Model - This model describes the objects in a system and their interrelationships. This model observes all the objects as static and does not pay any attention to their dynamic nature.

Dynamic Model - This model depicts the dynamic aspects of the system. It portrays the changes occurring in the states of various objects with the events that might occur in the system.

Functional Model - This model basically describes the data transformations of the system. This describes the flow of data and the changes that occur to the data throughout the system.

While the Object Model is most important of all as it describes the basic element of the system, the objects, all the three models together describe the complete functional system. As compared to the conventional system development techniques, OO modeling provides many benefits. Among other benefits, there are all the benefits of using the Object Orientation. Some of these are:

Reusability - The classes once defined can easily be used by other applications. This is achieved by defining classes and putting them into a library of classes where all the classes are maintained for future use. Whenever a new class is needed the programmer looks into the library of classes and if it is available, it can be picked up directly from there.

Inheritance - The concept of inheritance helps the programmer use the existing code in another way, where making small additions to the existing classes can quickly create new classes.

Programmer has to spend less time and effort and can concentrate on other aspects of the system due to the reusability feature of the methodology.

Data Hiding - Encapsulation is a technique that allows the programmer to hide the internal functioning of the objects from the users of the objects. Encapsulation separates the internal functioning of the object from the external functioning thus providing the user flexibility to change the external behaviour of the object making the programmer code safe against the changes made by the user.

The systems designed using this approach are closer to the real world as the real world functioning of the system is directly mapped into the system designed using this approach .

Advantages of object oriented methodology

Object Oriented Methodology closely represents the problem domain. Because of this, it is easier to produce and understand designs.

The objects in the system are immune to requirement changes. Therefore, allows changes more easily. Object Oriented Methodology designs encourage more re-use. New applications can use the existing modules, thereby reduces the development cost and cycle time.

Object Oriented Methodology approach is more natural. It provides nice structures for thinking and abstracting and leads to modular design.

The Unified Modeling Language

The **Unified Modeling Language** (**UML**) is a general-purpose modeling language in the field of software engineering, which is designed to provide a standard way to visualize the design of a system. It was created and developed by Grady Booch, Ivar Jacobson and James Rumbaugh.

UML is designed to enable users to develop an expressive, ready to use visual modeling language. In addition, it supports high level development concepts such as frameworks, patterns and collaborations. UML includes a collection of elements such as:

Programming Language Statements

Actors: specify a role played by a user or any other system interacting with the subject.

Activities: These are tasks, which must take place in order to fulfill an operation contract. They are represented in activity diagrams.

Business Process: includes a collection of tasks producing a specific service for customers and is visualized with a flowchart as a sequence of activities.

Logical and Reusable Software Components

UML diagrams can be divided into two categories. The first type includes six diagram types representing structural information. The second includes the remaining seven representing general types of behavior. Structure diagrams are used in documenting the architecture of software systems and are involved in the system being modeled. Different structure diagrams are:

1. Class Diagram: represents system class, attributes and relationships among the classes.
2. Component Diagram: represents how components are split in a software system and dependencies among the components.
3. Deployment Diagram: describes the hardware used in system implementations.

4. Composite Structure Diagram: describes internal structure of classes.

5. Object Diagram: represents a complete or partial view of the structure of a modeled system.

6. Package Diagram: represents splitting of a system into logical groupings and dependency among the grouping.

Behavior diagrams represent functionality of software system and emphasize on what must happen in the system being modeled. The different behavior diagrams are:

Activity Diagram: represents step by step workflow of business and operational components.

Use Case Diagram: describes functionality of a system in terms of actors, goals as use cases and dependencies among the use cases.

UML State Machine Diagram: represents states and state transition.

Communication Diagram: represents interaction between objects in terms of sequenced messages.

Timing Diagrams: focuses on timing constraints.

Interaction Overview Diagram: provides an overview and nodes representing communication diagrams.

Sequence Diagram: represents communication between objects in terms of a sequence of messages.

UML diagrams represent static and dynamic views of a system model. The static view includes class diagrams and composite structure diagrams, which emphasize static structure of systems using objects, attributes, operations and relations. The dynamic view represents collaboration among objects and changes to internal states of objects through sequence, activity and state machine diagrams.

Use Case Diagram

Use case diagram is used to capture the dynamic nature of a system. It consists of use cases, actors and their relationships. Use case diagram is used at a high level design to capture the requirements of a system. So it represents the system functionalities and their flow. Although the use case diagrams are not a good candidate for forward and reverse engineering but still they are used in a slightly differently way to model it.

Purpose of Use Case Diagram

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

Use Case Diagram objects

Use case diagrams consist of 4 objects.

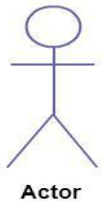
- Actor

- Use case
- System
- Package

The objects are further explained below.

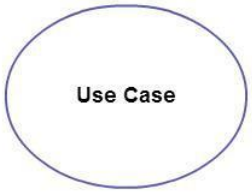
Actor

Actor in a use case diagram is **any entity that performs a role** in one given system. This could be a person, organization or an external system and usually drawn like skeleton shown below.



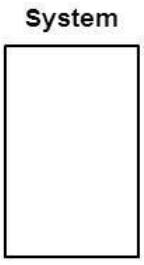
Use Case

A use case **represents a function or an action within the system**. Its drawn as an oval and named with the function.



System

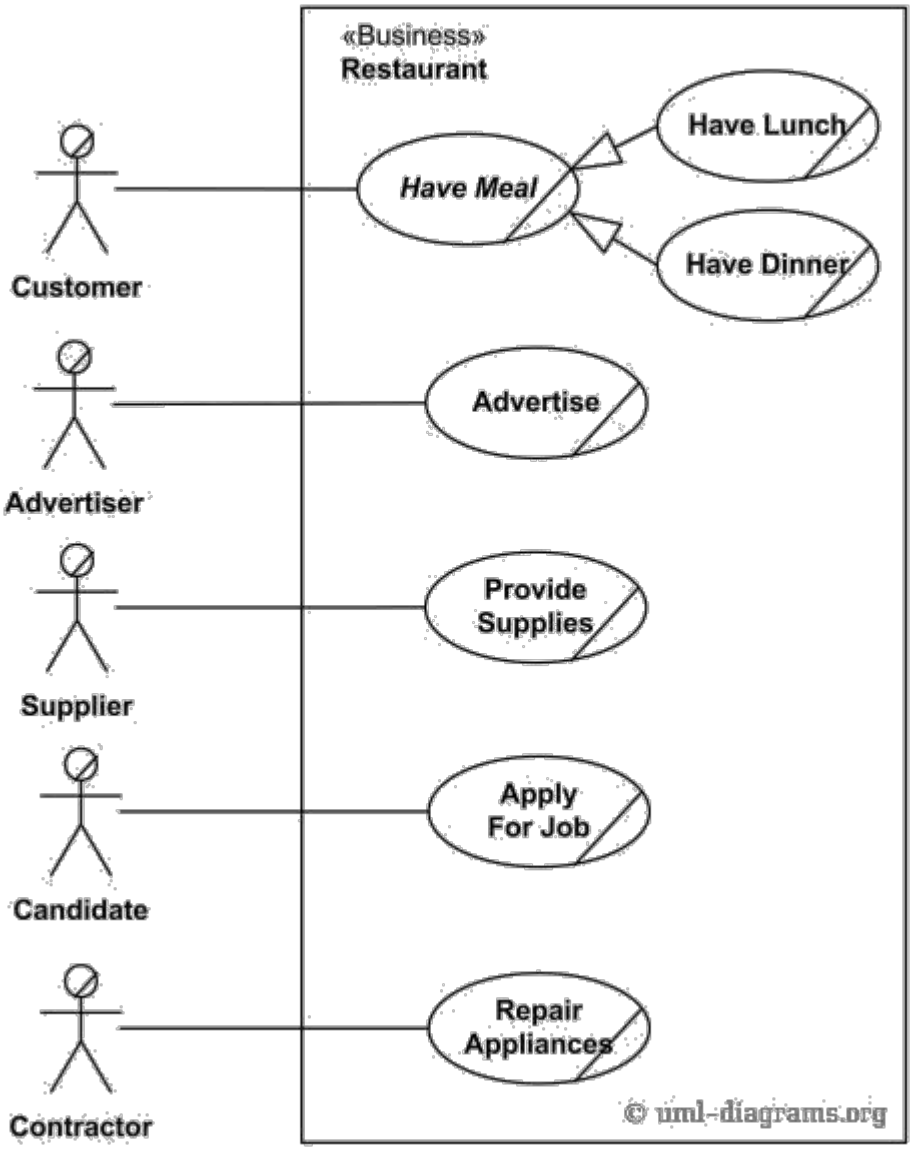
System is used to **define the scope of the use case** and drawn as a rectangle. This an optional element but useful when you are visualizing large systems. For example you can create all the use cases and then use the system object to define the scope covered by your project. Or you can even use it to show the different areas covered in different releases.



Package

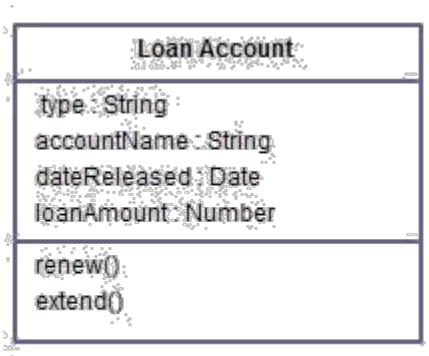
Package is another optional element that is extremely useful in complex diagrams. Similar to class diagrams, packages are **used to group together use cases**. They are drawn like the image shown below.

Example of use case diagram for a restaurant



Object modeling: Class Diagram

The class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of object oriented systems because they are the only UML diagrams which can be mapped directly with object oriented languages. The class diagram shows a collection of classes, interfaces, associations, collaborations and constraints. It is also known as a structural diagram. The purpose of the class diagram is to model the static view of an application. The class diagrams are the only diagrams which can be directly mapped with object oriented languages and thus widely used at the time of construction.



Simple class diagram with attributes and operations

In the example, a class called "loan account" is depicted. Classes in class diagrams are represented by boxes that are partitioned into three:

1. The top partition contains the name of the class.
2. The middle part contains the class's attributes.
3. The bottom partition shows the possible operations that are associated with the class.

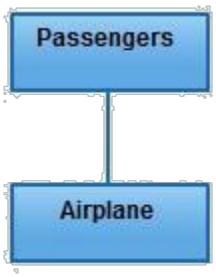
Those should be pretty easy to see in the example: the class being described is a loan account, some of whose attributes include the type of loan, the name of the borrower/loaner, the specific date the loan was released and the loan amount. As in the real world, various transactions or operations may be implemented on existing loans such as renew and extend. The example shows how class diagrams can encapsulate all the relevant data in a particular scenario in a very systematic and clear way.

In object-oriented modeling, class diagrams are considered the key building blocks that enable information architects, designers, and developers to show a given system's classes, their attributes, the functions or operations that are associated with them, and the relationships among the different classes that make up a system.

Relationships in Class Diagrams

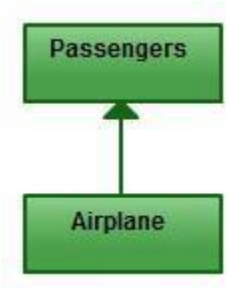
Classes are interrelated to each other in specific ways. In particular, relationships in class diagrams include different types of logical connections.

Association



Association is a broad term that encompasses just about any logical connection or relationship between classes. For example, passenger and airline may be linked as above.

Directed Association

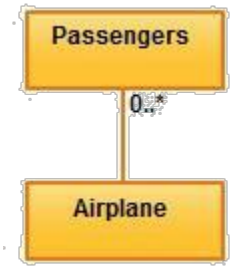


Directed Association refers to a directional relationship represented by a line with an arrowhead. The arrowhead depicts a container-contained directional flow.

Reflexive Association

Reflexive Association occurs when a class may have multiple functions or responsibilities. For example, a staff working in an airport may be a pilot, aviation engineer, a ticket dispatcher, a guard, or a maintenance crew member. If the maintenance crew member is managed by the aviation engineer there could be a managed by relationship in two instances of the same class.

Multiplicity

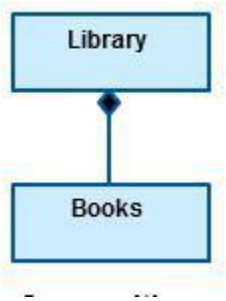


Multiplicity is the active logical association when the cardinality of a class in relation to another is being depicted. For example, one fleet may include multiple airplanes, while one commercial airplane may contain zero to many passengers. The notation 0..* in the diagram means “zero to many”.

Aggregation

Aggregation refers to the formation of a particular class as a result of one class being aggregated or built as a collection. For example, the class “library” is made up of one or more books, among other materials. In aggregation, the contained classes are not strongly dependent on the life cycle of the container. In the same example, books will remain so even when the library is dissolved. To render aggregation in a diagram, draw a line from the parent class to the child class with a diamond shape near the parent class.

Composition

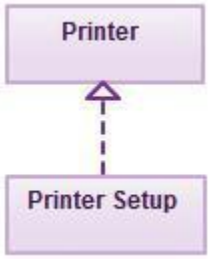


Composition is very similar to the aggregation relationship, with the only difference being its key purpose of emphasizing the dependence of the contained class to the life cycle of the container class. That is, the contained class will be obliterated when the container class is destroyed. For example, a shoulder bag’s side pocket will also cease to exist once the shoulder bag is destroyed. To depict a composition relationship in a UML diagram, use a directional line connecting the two classes, with a filled diamond shape adjacent to the container class and the directional arrow to the contained class.

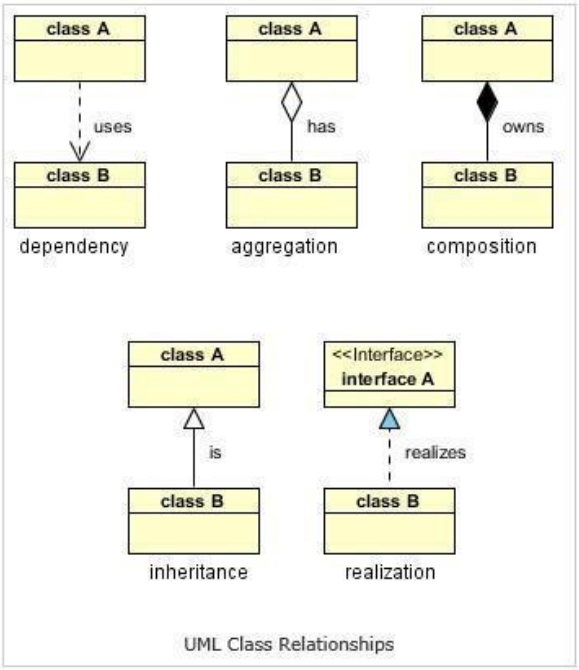
Inheritance / Generalization

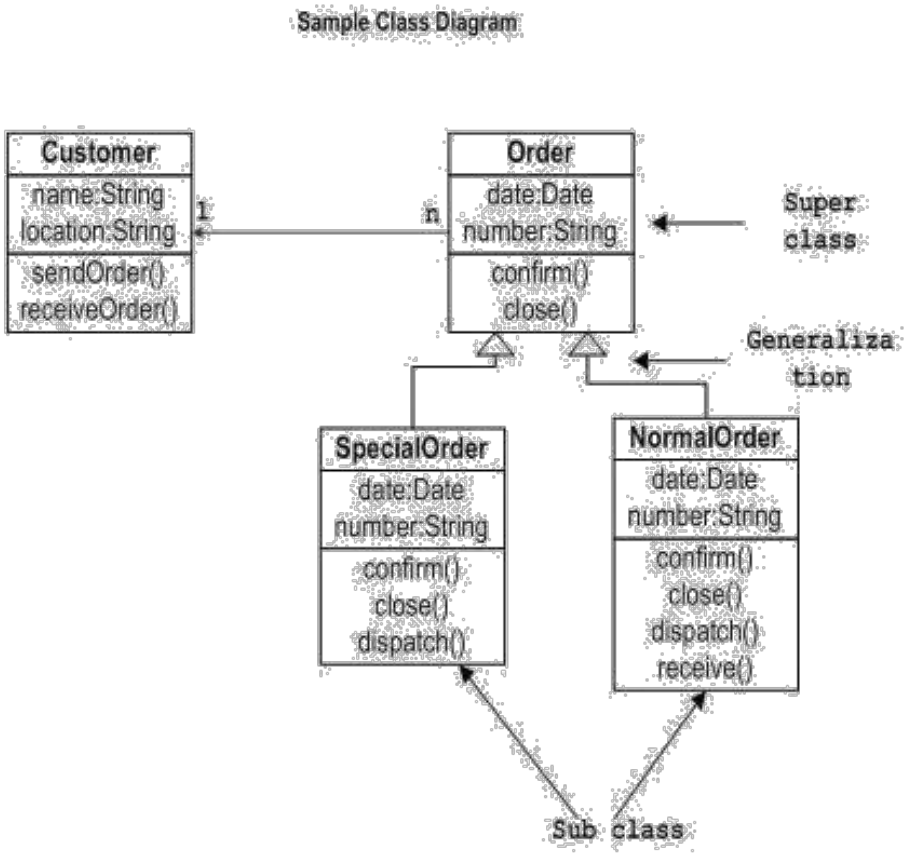
Inheritance refers to a type of relationship wherein one associated class is a child of another by virtue of assuming the same functionalities of the parent class. In other words, the child class is a specific type of the parent class. To depict inheritance in a UML diagram, a solid line from the child class to the parent class is drawn using an unfilled arrowhead

Realization



Realization denotes the implementation of the functionality defined in one class by another class. To show the relationship in UML, a broken line with an unfilled solid arrowhead is drawn from the class that defines the functionality to the class that implements the function. In the example, the printing preferences that are set using the printer setup interface are being implemented by the printer.





Dynamic Modeling: State Diagram

State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur. Each diagram usually represents objects of a single class and track the different states of its objects through the system. A state is a specific condition or situation of an element during its lifecycle.

Simple States

A simple state indicates a condition or situation of an element. For example, the project management system may be in one of the following simple states:

Inactive: Indicates that the project management system is not available to its users, because it is not started or has been shut down.

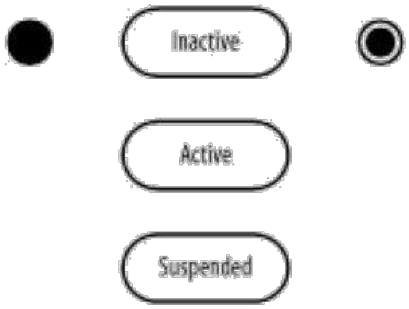
Active: Indicates that the project management system has been started and is available to its users.

Suspended: Indicates that the project management system has encountered some severe error, perhaps because it is running low on secondary storage and requires user intervention before becoming active again.

Initial and Final States

An initial state indicates the state of an element when it is created. In the UML, an initial state is shown using a small solid filled circle. A final state indicates the state of an element when it is destroyed. In the UML, a final state is shown using a circle surrounding a small solid filled circle.

A state diagram may have only one initial state, but may have any number of final states **initial, simple and final states**



Events

An event is an occurrence, including the reception of a request. There are a number of different types of events within the UML.

CallEvent. Associated with an operation of a class, this event is caused by a call to the given operation. The expected effect is that the steps of the operation will be executed.

SignalEvent. Associated with a signal, this event is caused by the signal being raised. TimeEvent. An event cause by expiration of a timing deadline.

ChangeEvent. An event caused by a particular expression (of attributes and associations) becoming true.

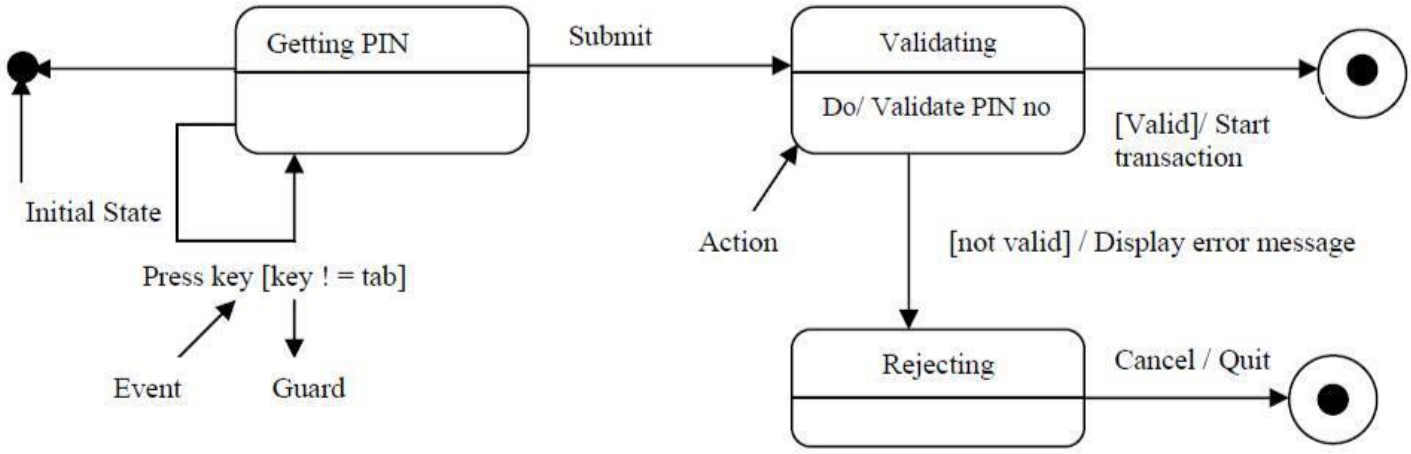


Figure: State diagram of online banking system

Logging in can be factored into four non-overlapping states: **Getting PIN**, **Validating**, and **Rejecting**. From each state comes a complete set of **transitions** that determine the subsequent state.

States are rounded rectangles. Transitions are arrows from one state to another. Events or conditions that trigger transitions are written beside the arrows. Our diagram has self-transition, on **Getting PIN**.

The initial state (black circle) is a dummy to start the action. Final states are also dummy states that terminate the action.

The action that occurs as a result of an event or condition is expressed in the form action. While in its **Validating** state, the object does not wait for an outside event to trigger a transition. Instead, it performs an activity. The result of that activity determines its subsequent state.

Dynamic Modeling: Sequence Diagram

A sequence diagram shows elements as they interact over time, showing an interaction or interaction instance. Sequence diagrams are organized along two axes: the horizontal axis shows the elements that are involved in the interaction, and the vertical axis represents time proceeding down the page. The elements on the horizontal axis may appear in any order.

Class Roles or Participants

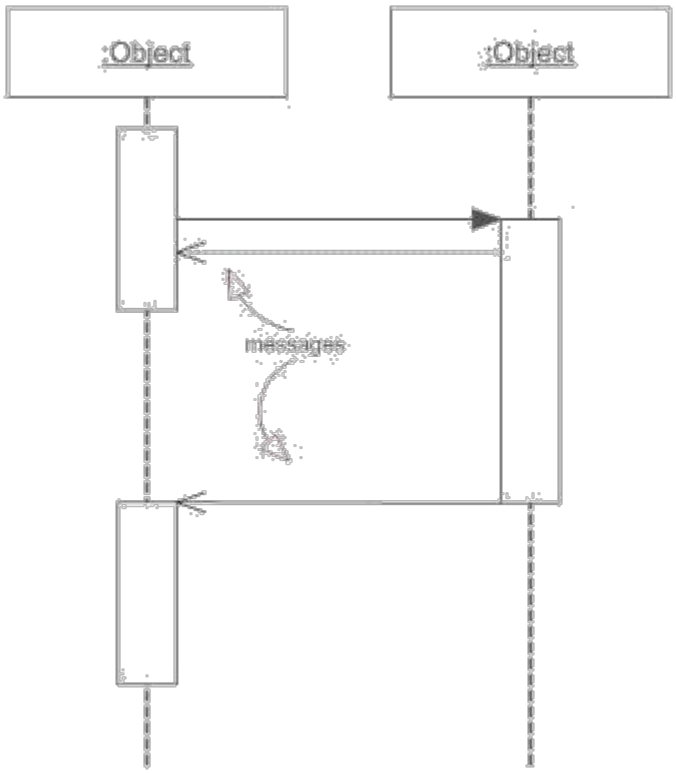
Class roles describe the way an object will behave in context. Use the UML object symbol to illustrate class roles, but don't list object attributes.

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.

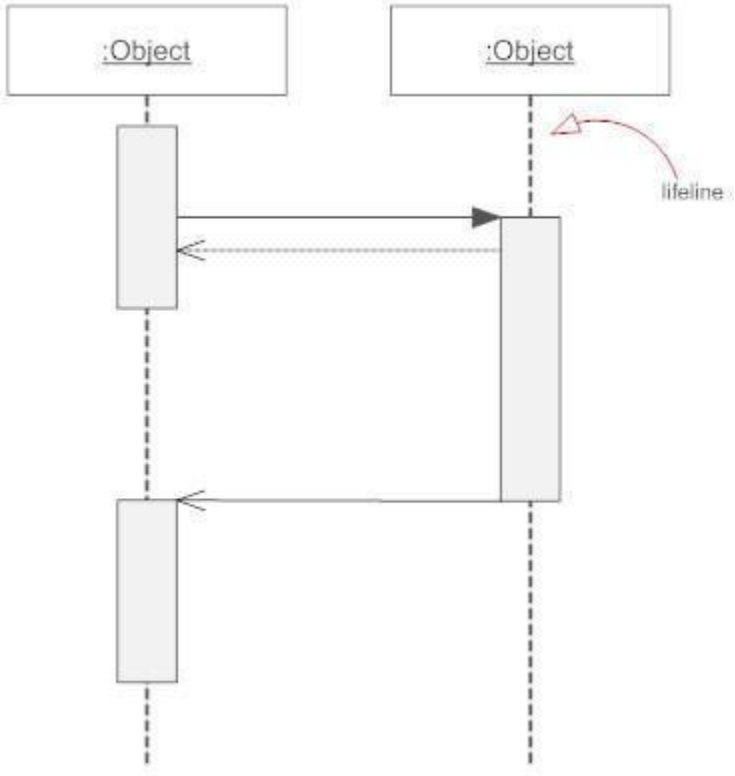
Messages

Messages are arrows that represent communication between objects. Use half-arrowed lines to represent asynchronous messages. Asynchronous messages are sent from an object that will not wait for a response from the receiver before continuing its tasks. For message types, see below.



Lifelines

Lifelines are vertical dashed lines that indicate the object's presence over time.



Destroying Objects

Objects can be terminated early using an arrow labeled "<< destroy >>" that points to an X. This

object is removed from memory. When that object's lifeline ends, you can place an X at the end of its lifeline to denote a destruction occurrence.

Loops

A repetition or loop within a sequence diagram is depicted as a rectangle. Place the condition for exiting the loop at the bottom left corner in square brackets [ ].

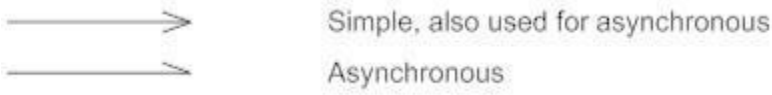
Types of messages in Sequence Diagram

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.

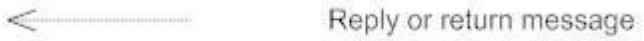
Asynchronous Message

Asynchronous messages don't need a reply for interaction to continue. Like synchronous messages, they are drawn with an arrow connecting two lifelines; however, the arrowhead is usually open and there's no return message depicted.



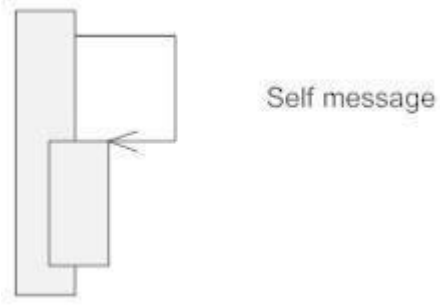
Reply or Return Message

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



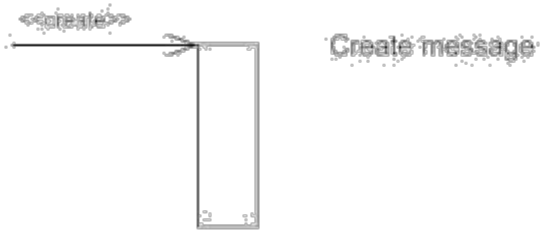
Self Message

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



Create Message

This is a message that creates a new object. Similar to a return message, it's depicted with a dashed line and an open arrowhead that points to the rectangle representing the object created.



Delete Message

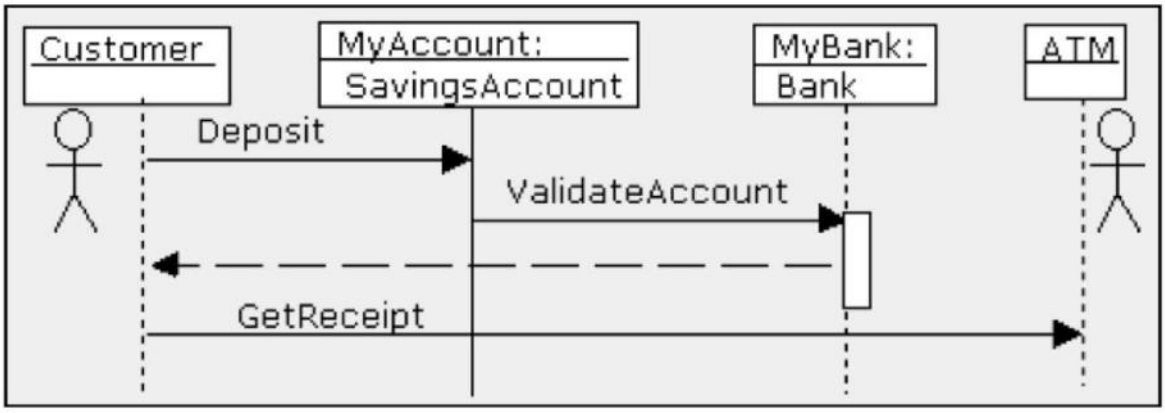
This is a message that destroys an object. It can be shown by an arrow with an x at the end.

Found Message

A message sent from an unknown recipient, shown by an arrow from an endpoint to a lifeline.

Lost Message

A message sent to an unknown recipient. It's shown by an arrow going from a lifeline to an endpoint, a filled circle or an x.



The Sequence diagram allows the person reading the documentation to ***follow the flow of messages*** from each object. The vertical lines with the boxes on top represent instances of the classes (or objects). The label to the left of the colon is the instance and the label to the right of the colon is the class. The horizontal arrows are the messages passed between the instances and are read from top to bottom. Here

a customer (user) depositing money into MyAccount which is an instance of Class SavingsAccount.
Then MyAccount object Validates the account by asking the Bank object, MyBank to ValidateAccount.
Finally, the Customer Asks the ATM object for a Receipt by calling the ATM‘s operation GetReceipt.

The white rectangle indicate the scope of a method or set of methods occurring on the Object My Bank. The dotted line is a return from the method ValidateAccount.

Analysis verses Design

Analysis	Design
1. Analysis focus on determining what the business needs is.	1. Design takes those business needs and determines how they will be met through a specific system implementation.
2. The analysis phase includes activities designed to discover and document the features and functions the system must have.	2. The focus in design phase is to figure out how to create a system technically that will provide all those needed features and functions.
3. Analyst thoroughly studies the organization's current procedures and the information systems used to perform organizational task.	3. Analyst convert the description of the recommended alternative solution into logical and then physical system specifications.
4. In this phase analysts work with users to determine what the users want from a proposed system.	4. In this phase analysts design all aspects of the system, from input and output screens to reports, databases and computer processes.
5. This phase comes before design phase.	5. This phase comes after analysis phase.
6. The output of the analysis phase is a description of the alternative solution recommended by the analysis team.	6. The output of the design phase is the physical system specifications in a form ready to be turned over to programmers and other system builders for construction.

University Exam questions

1. Explain the Unified Modeling Language with example. (2069)(5 marks)
2. Differentiate between object modeling and dynamic modeling (2070)(2071)(5 marks)
3. What are the major differences between analysis and design? (2071)(5 marks)