Unit – 3 OOSE

Object-Oriented Analysis (OOA): Object-Oriented Analysis (OOA) is the first technical activity performed as part of object-oriented software engineering. OOA introduces new concepts to investigate a problem. It is based on a set of basic principles, which are as follows-

1. The information domain is modeled.
2. Behavior is represented.
3. The function is described.
4. Data, functional, and behavioral models are divided to uncover greater detail.
5. Early models represent the essence of the problem, while later ones provide implementation details.

Advantages/Disadvantages of Object Oriented Analysis

Advantages	Disadvantages
Focuses on data rather than the procedures as in Structured Analysis.	Functionality is restricted within objects. This may pose a problem for systems which are intrinsically procedural or computational in nature.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	It cannot identify which objects would generate an optimal system design.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	The object-oriented models do not easily show the communications between the objects in the system.
It allows effective management of software complexity by the virtue of modularity.	All the interfaces between the objects cannot be represented in a single diagram.
It can be upgraded from small to large systems at a greater ease than in systems following structured analysis.	

- Object-oriented software engineering (OOSE) proposes two analysis models for understanding the **problem domain**
  - Requirements Model
  - Analysis Model

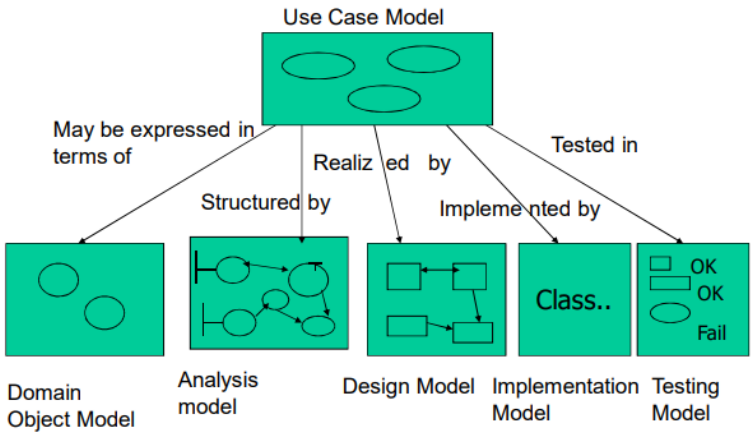
The **requirements model** serves two main **purposes**
- To delimit the **system**
- To define the **system functionality**

Requirement Model

- **Conceptual model** of the system is developed using:
  - **Problem domain objects**
  - **Specific interface descriptions** of the system (if meaningful to the system being developed)

⊗ The system is described as a *number of use cases* that are performed by a *number of actors*
- **Actors** constitute the **entities** in the **environment** of the system
- **Use cases** describe what takes places **within the system**
- A use case is a **specific way** of **using** the system by performing **some part** of the system functionality

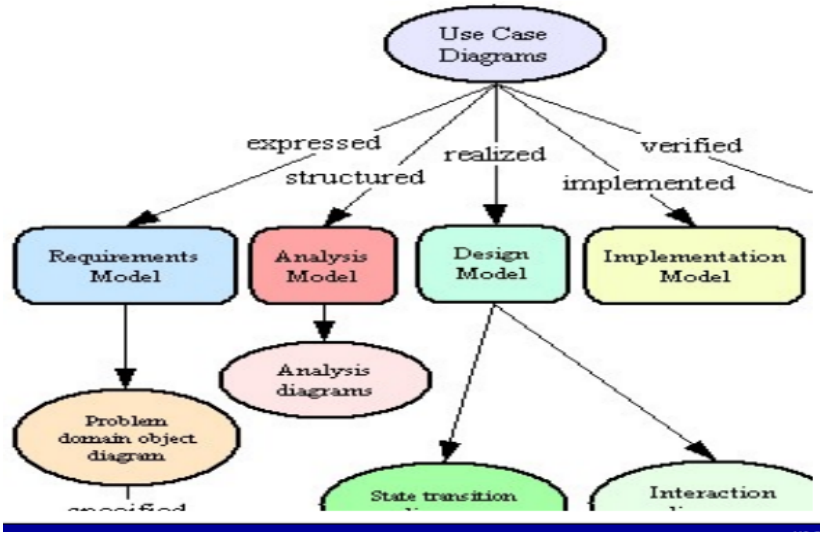
Requirement Model



Requirement Model

- The requirements model for the will comprise three main models of representation:
  - The use case model
  - The problem domain model
  - User interface descriptions

Requirement Model



Actors

- In order to identify use cases to be performed in the system, we need to first **identify system users**
- The **system users** are referred to as **actors**.
- Actors model the **prospective** '***users***' of the system.
- An actor is a **user type** or **category**. When an actor does something, the actor acts as an **occurrence** of that **type**.
- An actor may represent a **person** or **another system interacting** with the **intended system**

Actors and Role Play

- One person can **instantiate** (play the roles of) several different actors
- Actors **define** the **roles** that **users** can play
- Actors model anything that needs to **exchange information** with the **system**.
- Actors can **model** **human users** but they can also model other systems communicating with the intended system

Identifying Actors

- Actors constitute anything **external** to the system
- Identifying all the relevant actors for a system may require several **iterations**
- **General guidelines** include the following:
  - Ask yourself why the system is been developed
  - Who are people the system is intended to help?
  - What other systems are likely to interface with new system?

Primary and Secondary Actors

- Actors who ***use the system directly*** (or in their daily work) are known as **Primary actors**
- Primary actors are **associated** with one or more of the **main tasks** of the system
- Primary actors **govern** the **system structure**. Thus when ***identifying use cases***, we first start with the primary actors
- Actors who are concerned with **supervising** and **maintaining** the system are called **secondary actors**
- The **distinction** between the primary and secondary actors has a *bearing on the system structuring*

Use Cases

- After the actors have been identified the next step is to ***define the functionality of the system***. This is done by **specifying use cases.**
- Actors are a **major tool** in finding use cases. Each actor will perform a number of **use cases** in the system.
- Each use case constitutes a *complete course of events* initiated by an actor and specifies the **interaction** that takes place between the **actor** and the **system**
- A use case is a **special sequence** of **related transactions** performed by an **actor** and the **system** in dialogue.
- The *collective use cases* should specify all the **existing ways** of **using the system**

Use Case Relationships- Include

- **Use case include** is a directed relationship between two **use cases** which is used to show that behaviour of the **included** use case (the addition) is inserted into the **behaviour** of the **including** (the base) use case.
- The **include** relationship could be used:
  - To **simplify large use case** by splitting it into several use cases,
  - To **extract common parts** of the behaviours of two or more use cases.
- A large use case could have some behaviours which might be **detached** into distinct smaller use cases to be **included back** into the base use case using the UML **include** relationship.
- The **purpose** of this action is **modularization of behaviours**, making them more **manageable**.

Use Case Relationships- Extend

Extend is a **directed relationship** that specifies how and when the behaviour defined in usually supplementary (optional) **extending use case** can be inserted into the **behaviour** defined in the **extended use case**.

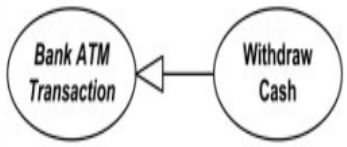
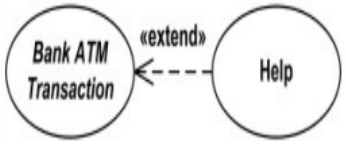
Extended use case is meaningful on its own, it is **independent** of the extending use case.

Extending use case typically defines **optional behaviour** that is not necessarily meaningful by itself.

The extension takes place at one or more **extension points** defined in the **extended use case**.

Extend relationship is shown as a dashed line with an open arrowhead directed from the **extending use case** to the **extended (base) use case**.

A Comparative Study

Generalization	Extend	
		
Base use case could be abstract use case (incomplete) or concrete (complete).	Base use case is complete (concrete) by itself, defined independently.	Base use case is incor
Specialized use case is required, not optional, if base use case is abstract.	Extending use case is optional, supplementary.	Included use case rec
No explicit location to use specialization.	Has at least one explicit extension location.	No explicit inclusion

Requirement Engineering

Requirements

- Goal
  - To understand the problem
- Necessary to Understand Requirements
  - Organization
  - Existing Systems
  - Processes
  - Improvements
- Once you have all this information, now what?

Requirement Elicitation

- ✓Techniques
  - Interview / Meeting
  - Survey / Questionnaire
  - Observation
  - Ethnography / Temporary Assignment
  - Business Plans
  - Review Internal / External Documents
  - Review Software

Requirement Analysis

Requirements Analysis

- Goal
  - To bridge the gap between the problem domain and the technical domain
- Tasks
  - Problem Recognition
  - Evaluation and synthesis
  - Modeling
  - Specification
  - Review

Requirement Analysis Principles

Requirements Analysis Principles

- Information domain of a problem must be represented and understood
- Models that depict system information, function, and behavior should be developed
- Models must be partitioned in a manner that uncovers detail in a layered fashion
- Analysis process should move from essential information toward implementation detail

Object Oriented Analysis

- **Identifying objects**: Using concepts, CRC cards, stereotypes, etc.
- **Organising the objects**: classifying the objects identified, so similar objects can later be defined in the same class.
- **Identifying relationships between objects**: this helps to determine inputs and outputs of an object.
- **Defining operations of the objects**: the way of processing data within an object.
- **Defining objects internally**: information held within the objects.

The Analysis Phase

- Begins with a **problem statements** generated during **system conception.**
- In software engineering, analysis is the process of **converting** the user requirements to system specification (system means the software to be developed).
- System specification, also known as the **logic structure**, is the developer's view of the system.
- **Function-oriented analysis**
  - Concentrating on the **decomposition** of complex functions to simply ones.
- **Object-oriented analysis**
  - Identifying **objects** and the **relationship** between objects.

Analysis Model

⊗ The analysis model gives a **conceptual configuration** of the system.

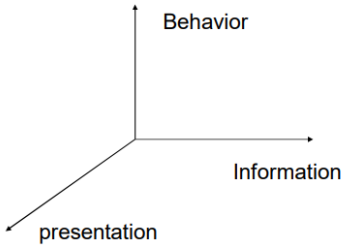
It consists of:

- The entity objects
- Control objects
- Interface objects

⊗ The analysis model forms the **initial transition** to **object-oriented design**

Dimensions of Analysis Model

Dimensions of the analysis model



Analysis Model- *Objects*

Entity object

- Information about an entity object is stored even after a use case is completed.

Control object

- A control object shows functionality that is not contained in any other object in the system

Interface object

- Interface objects interact directly with the environment

Design Model

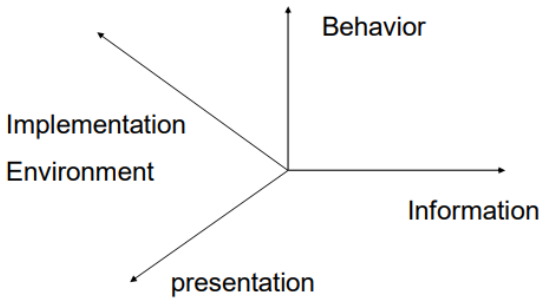
- Developed based on the analysis model
  - Implementation environment is taken into consideration
- The considered **environment factors** includes
  - Platform
  - Language
  - DBMS
  - Constraints
  - Reusable Components
  - Libraries
  - so on..

Design Model

- Design objects are different from analysis objects
- **Models**
  - Design object interactions
  - Design object interface
  - Design object semantics
    - ✓**(i.e., algorithms of design objects' operations)**
- More closer to the **actual source code**

Design Model Dimensions

Dimensions of the Design model



Design Model

- Use **block term** in place of object
- Sent from one block to another to trigger an execution
- A typical **block** is **mapped** to **one file**
- To manage system **abstractly** subsystem concept is introduced
- **Analysis Model** is viewed as **conceptual** and **logical model,** whereas the **design model** should take as closer to the **actual source code**
- Consist of explained source code
- OO language is desirable since all fundamentals concepts can easily be mapped onto **language constructs**
- Strongly desirable to have an easy match between a **block** and the **actual code module**

Implementation Model

- Consists of **annotated source code.**
- Object oriented language is desirable since all **fundamental concepts** can be easily **mapped** onto **language constructs**.
- Strongly desirable to have an easy **match** between a **block** and the **actual code module.**

Test Model

Fundamental concepts are **test specifications** and the **test results**

Rumbaugh Methodology (OMT)

The Rumbaugh methodology also known as OMT (Object Modeling Technique) is an approach used to develop manageable object-oriented systems and host object oriented programming. The purpose is to allow for class attributes, methods, inheritance, and association to be easily expressed. OMT is used in the real world for software modeling and designing.

According to Rumbaugh, there are several main reasons to utilize this modeling approach. One is to simulate entities before constructing them and another is to make communication with customers easier. Additionally, it help to reduce complexity through visualization.

OMT consists of four stages:

- Analysis
- Systems Design
- Object Design
- Implementation

Additionally, OMT is always broken down into three separate parts. These parts are the:

- An object model
- A dynamic model
- A functional model.

We will explain the bullets in greater detail later on.

Notation/Diagramming Technique

In OMT, abstract or concrete classes are represented as rectangles. Italicized font indicates abstraction while a normal font represents a concrete class.

CONCRETE	***ABSTRACT***

Lines indicate an association. There can be different multiplicities of association:

- A regular line indicates exactly one association
- Numbers above the line will indicate the multiplicity of associations
- A line with an open circle indicates optional multiplicity
- A line with a filled circle indicates many multiplicities
- A line with a diamond indicates an aggregation

Note: Multiplicity indicates the number of objects that can participate in a relationship or the number of instances of one class as it relates to another. Also, an aggregation is a tighter-defined association.

Additionally, objects are portrayed as rounded rectangles, the $ sign represents class operations or attributes, and a triangle in between associations represents a generalization or inheritance.

Phases

OMT consists of the following four phases:

Analysis: Assigns an object, dynamic and functional model to the design. Determines important properties and domain.
Systems Design: Outlines the basic systems structure of the program. Accounts for data storage and concurrency among other things.
Object Design: Classifies objects and determines operations and data structures. Inheritance and different associations are also checked.
Implementation: Conveys design through code.

Models

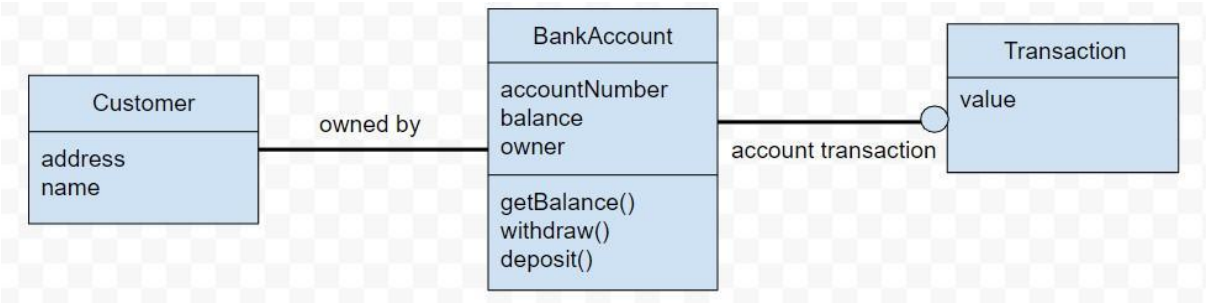
There are three main types of models:

Object Model: This model is the most stable and static of the models. Concerns itself with mainly classes and their associations with attributes. Divides the model into objects.
Dynamic Model: Concerns itself with the interaction between objects through events, states, and transitions.
Functional Model: Concerns itself with data flows, data storage, constraints, and processes

Example

Below is a simple example of a bank account. We've described the relationship and process of simple transactions between customers and bank accounts. We've also listed the instances and methods (only in the BankAccount class). And we have optional multiplicity of association between our transaction and bank account.

Take a look.



The Object Model

- It describes the structure of objects in the system : their identity, relationships to other objects, attributes and operations.
- It is represented graphically with an **object model** and the **data dictionary**.

- It contains classes interconnected by Association lines.
- Each class represents a set of individual objects. The association lines established relationships among the classes. Each Association line represents a set of links from the objects of one class to the objects of another class.

The OMT Dynamic model

- It provides detailed and comprehensive dynamic model, which depicts states, transitions, events and actions.
- It is a network of states and events and actions. Each state receives one or more events at which it makes the transition to the next step. The next step depends on the current state as well as events.

The OMT Functional model

The OMT data flow diagram (DFD) shows the flow of data between different processes in a business. It provides a simple and intuitive method describing business processes without focusing on the details of computer system.

It uses 4 primary symbol :

- **The process** : It is any function being performed for example verify password or PIN the ATM system.
- **The data flow** : It shows the direction of data element movement for example PIN code
- **Data store** : It is the location where data are stored for example account is a data store in the ATM example.
- **The external entity** : It is a source or destination of data element example the ATM card reader

Overall, the Rumbaugh et al. OMT methodology provides one of the strongest tool set for the analysis and design of object oriented