Introduction to Object-Oriented Development

There have been basically 3 approaches in information system development area:

process-oriented

data-oriented and

object-oriented approaches.

As information technology (both hardware and software) has been advancing, people have moved from the earliest **process-oriented** approach to **data-oriented approach** and now begun to adopt the latest **object-oriented analysis** methodology.

Unlike its two predecessors that focus either on **process or data**, the object-oriented approach combines **data and processes (called methods)** into single entities called **objects.** Objects usually correspond to the real things an information system deals with, such as customers, suppliers, contracts, and rental agreements. Object-oriented model is able to thoroughly represent complex relationships and to represent data and data processing with a consistent notation, which allows an easier blending of analysis and design in an evolutionary process. The goal of **object-oriented approach is to make system elements more reusable, thus improving system quality and the productivity of systems analysis and design** (Hoffer et al. 2002). Though systems analysis is closed associated with design, this paper tries to focus on analysis part of the methodology.

Mechanism of Object-oriented Approach

Object Oriented Methodology (OOM) is a system development approach encouraging and facilitating re-use of software components

The principals of **objects, encapsulation, inheritance, and polymorphism are the foundation for object-oriented systems development**. To understand and express the essential and interesting features of an application in the complex real world, an object-oriented

model is built around objects. An object encapsulates both data and behavior, implying that analysts can use the object-oriented approach for both data modeling and process modeling.

Object is a real world entity such as student, bag, pen, college etc. Class is a group of similar objects. Object is an instance of a class. Class is a blueprint or template from which objects are created. .Specific objects in a system can inherit characteristics from the global instance of an object. For example, many types of objects may have a name and a creation date.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference. **Encapsulation is one of the fundamentals of OOP . It refers to the bundling of data with the methods that operate on that data.**

Inheritance is **the procedure in which one class inherits the attributes and methods of another class**. The class that inherits the properties from the parent class is the Child class. Inheritance is a mechanism in which one class acquires the property of another class. **Specific objects can inherit these global characteristics from parent objects that include only global characteristics. Objects can inherit characteristics from more than one parent object. Inheritance attempts to avoid the redundant definition of similar characteristics that can be embodied at higher levels in the system** (Cackowski 2000).

Polymorphism is one of the core concepts of object-oriented programming (OOP) and describes **situations in which something occurs in several different forms.** The **process of having same name of a function and different task** is generally known as the polymorphism.

By a concept called **polymorphism, functionality that is conceptually similar among differing objects is extracted to a global level**. This process limits the production of parallel functionality and streamlines the information interface. Polymorphism directs the specification writer to understand the functionality of a process and make it available to any object that requires a similar instance of functionality (Cackowski 2000).

Object-Oriented Analysis

Object–Oriented Analysis (OOA) **is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.** The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions. They are modelled after real-world objects that the system interacts with. In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

Grady Booch has defined OOA as, **"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".**

The primary tasks in object-oriented analysis (OOA) are:

Identifying objects

Organizing the objects by creating object model diagram

Defining the internals of the objects, or object attributes

Defining the behavior of the objects, i.e., object actions

Describing how the objects interact

The common models used in OOA are use cases and object models.

Object-Oriented Design

Object–Oriented Design **(OOD) involves implementation of the conceptual model produced during object-oriented analysis**. In OOD, concepts in the analysis model, which are technology−independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies. The implementation details generally include: Restructuring the class data (if necessary),

Implementation of methods,

i.e., internal data structures and algorithms,

Implementation of control, and

Implementation of associations.

Grady Booch has defined **object-oriented design as "a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design".**

Object-oriented analysis and design (OOAD)

It is **a technical approach for analyzing and designing an application, system, or business by applying object-oriented programming**, as well as using visual modeling throughout the software development process to guide stakeholder communication and product quality. Due to its **maintainability OOAD** is becoming more popular day by day. It provides re-usability. It reduce the development time & cost. It improves the quality of the system due to program reuse. **Followings are the advantages of object oriented design.**

Focuses on **data rather than the procedures** as in Structured Analysis.

The principles of **encapsulation and data hiding** help the developer to develop systems that cannot be tampered by other parts of the system.

It allows effective management of **software complexity by the virtue of modularity.** It can be **upgraded from small to large systems at a greater ease** than in systems following **structured analysis.** **Object oriented** models are used to fill the gap between problem and solution. It performs well in situation where **systems** are undergoing continuous **design**

Unified Modeling Language

The Unified Modeling Language (UML) is an object-oriented language for **specifying, visualizing, constructing, and documenting** the artifacts of software systems, as well as for business modeling (UML Document Set, 2001). The UML was **developed by Rational Software and its partners**. It is the successor to the modeling languages found in the Booch (Booch 1994), OOSE/Jacobson, **object modeling technique** (OMT) and other methods.

By offering a common blueprinting language, UML relieves developers of the proprietary ties that are so common in this industry. **Major vendors including IBM, Microsoft, and Oracle are brought together under the UML umbrella.** And because UML uses simple, intuitive notation, nonprogrammers can also understand UML models. In fact, many of the language's supporters claim that UML's simplicity is its chief benefit. If developers, customers, and implementers can all understand a UML diagram, they are more likely to agree on the intended functionality, thereby improving their chances of creating an application that truly solves a business problem (Apicella 2000).

The UML, a visual modeling language, is not intended to be a visual programming language. The **UML notation is useful for graphically depicting object-oriented analysis and design models**. It not only allows you to specify the requirements of a system and capture the design decisions, **but it also promotes communication among key persons involved in the development effort** (Hoffer et al. 2002). The **emphasis in modeling should be on analysis and design, focusing on front-end conceptual issues, rather than back-end implementation issues**, which unnecessarily restrict design choices (Rumbaugh et al. 1991).

Analysis Process

In the analysis phase, **a model of the real-world application is developed showing its important properties.** It abstracts concepts from the application domain and describes what the intended system must do, rather than how it will be done.

Most proponents of object-oriented analysis (OOA) claim that the use of object-oriented concepts in the analysis phase (i.e., OOA) **increases the understanding of problem domains, that OOA promotes a smooth transition from the analysis phase to the design phase,** and that OOA provides a more natural way of organizing specifications (Coad & Yourdon, 1991).Extant object-oriented approaches can be classified into three categories (Monarchi & Puhr, 1992):

(1) **combinative approaches** use different modeling techniques in different stages of the system development process.

(2) **adaptive approaches** apply existing techniques (e.g., data-flow diagram and entity-relationship approach) in object-oriented ways to analyze the problem domain.

(3) **pure approaches** adopt an object-oriented perspective in systems analysis and design.

Despite various ways to do modeling, this paper focuses on **use-case modeling and class modeling** to explore how system analysis are conducted under different methods.

Use-case Modeling

First adopted by Jacobson et al. (1992), use-case modeling is developed in the analysis phase of the object-oriented system development life cycle. **Use-case modeling is done in the early stages of system development to help developers gain a clear understanding of the functional requirement of the system, without worrying about how those requirements will be implemented.**

A use-case is a representation of a discrete set of work performed by a use (or another system) using the operational system (). A use-case model consists of **actors and use cases**. **An actor is an external entity that interacts with the system** and **a use case represents a sequence of related actions initiated by an actor to accomplish a specific goal** (Hoffer et al. 2002).

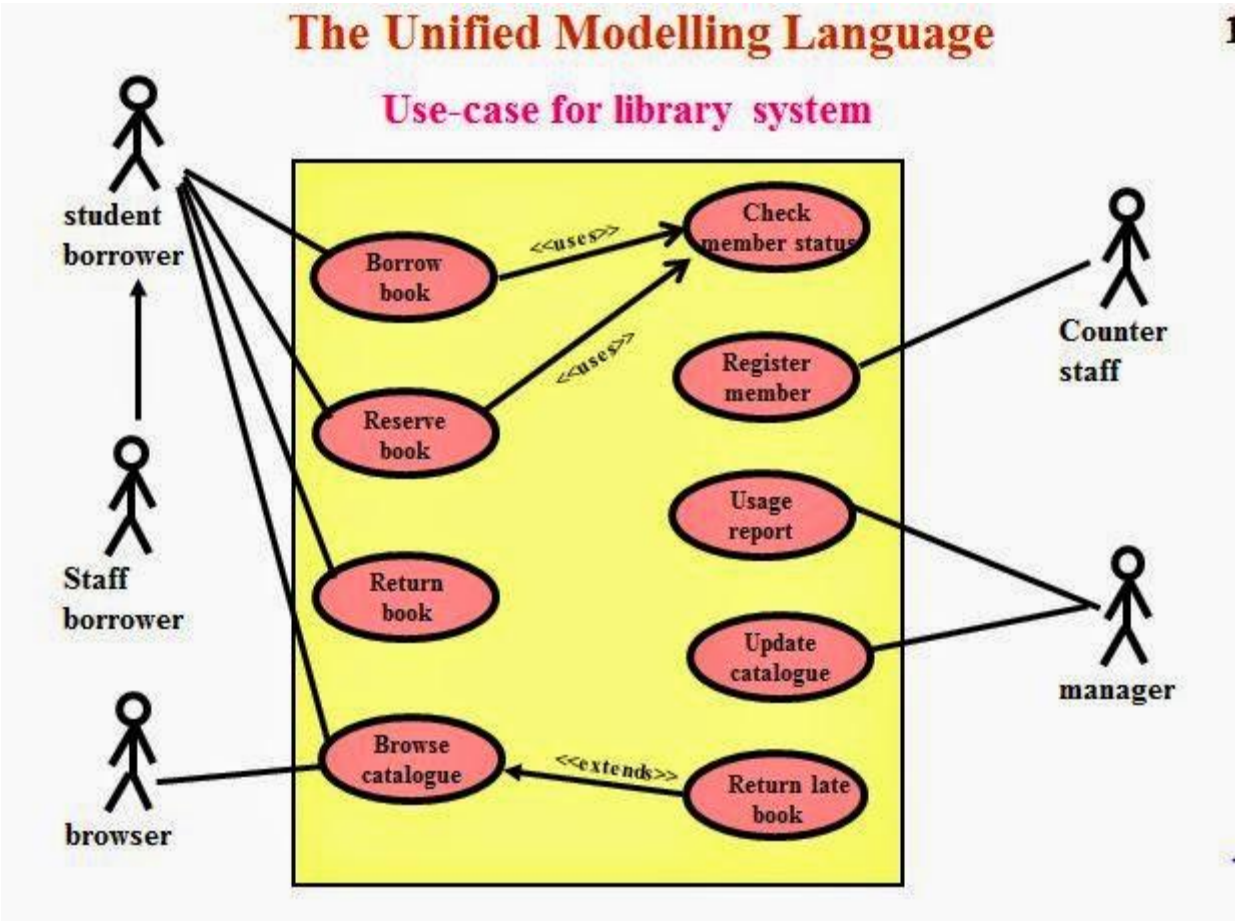
For identifying use cases, Jacobson et al. (1992) recommend to ask the following questions:

·What are the main tasks performed by each actor?
·Will the actor read or update any information in the system?
·Will the actor have to inform the system about changes outside the system?
· Does the actor have to be informed about unexpected changes?

In UML, a use-case model is depicted in a use-case diagram that contains the use cases and actors for a system. Begin working with the UML by modeling all scenarios in the system or business with Use Case diagrams. Describe the system in terms of actors, which are external agents that request a service from the system, and Use Cases. Each Use Case can be defined simply by a textual statement that describes the scenario, or via other definitions, such as the sequence of steps that are performed within the scenario, or the pre- and post-conditions of the scenario (Popking 2001).

A use-case diagram for a library system is shown below

1. **An actor is shown using a stickman symbol with its name below.** There are five actors outside the box: student borrower,staff borrower,browser,counter staff andmanager. Inside



the box are 9 use cases – borrow book,reserve book,return book,browse catalogue ,check member status,register member,usage reportupdate catalogue, return book etc.

Use-cases are shown as ellipses with their names inside and are performed by the actors outside the system. A use-case is always initiated by an actor. For example, **register member** is initiated by **Counter staff**.

A use-case may interact with other use-cases. Some of these relationships include extend and use and are reflected by single hollow arrow lines. For instance, **borrow book** use case uses information from **check member status**, **Return late book** case extends **browse catalogue** .

While a use-case diagram shows all the use cases in the system, it does not describe how those use cases are carried out by the actors. The contents of a use case are normally described in plain text. While describing a use case, you should focus on its external behavior, that is, how is interacts with the actors, rather than how the use case is performed inside the system (Eriksson and Penker, 1998).

Also called system usage modeling, a use case modeling, at requirements analysis stage, consists of a Use Case Diagram plus a set of descriptions as well as illustrations of prototype screens.

One of the benefits of use-case or system usage modeling is its simplicity. The strength of the technique is in its non-technical simplicity, which allows users to participate in a way that is seldom possible using the abstractions of Class Modeling alone. It also helps the analyst get to grip with specific user needs before analyzing the internal mechanics of a system. Use cases also fit particularly well within an evolutionary and incremental process in that they provide a basis for early prototyping and readily identifiable units for incremental delivery. They also provide a means of traceability for functional requirements upstream in the process and for constructing test plans downstream in the process. It is no accident therefore that use cases have become such a popular technique (Artisan 2001). In summary, (Artisan 2001) provides a complete list of the steps in system usage modeling as follows:

· **Identify the actors.**
· **Identify the use cases.**
· **Create a Use Case Diagram.**
· **Describe the use cases.**
· **Complete the use case descriptions.**

Class Modeling

There are many new terms in object-oriented approach. Some have already been introduced above. **An object is the most fundamental element in OO approach, which has a well-defined role in the application domain, and has state, behavior, and identity**. A class is a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment.

Object modeling, or class modeling is the key activity in object-oriented development. If the use cases contain errors, then all is not lost. If the class model contains errors then all may well be lost. The quality of the resulting system in object-oriented development is essentially a reflection of the quality of the class model. This is because the class model sets the underlying foundation upon which objects will be put to work. A quality class model should provide a flexible foundation upon which systems can be assembled in component-like fashion. A poor class model results in a shaky foundation upon which systems will grind to a halt and buckle under the threat of change (Artisan 2001).

Each of the diagrams used in UML lets you see a business process from a different angle. Business users, for example, can view use case diagrams to see the business scenario overview and understand who's doing what, while developers can use class and object diagrams to get accurate descriptions of how to build those components into their code. The class and object diagrams are so detailed, describing elements such as interfaces and attributes, that translating UML notation into actual programming code is a virtual no-brainer (Apicella 2000).

In UML, a class is represented by a rectangle with three compartments separated by horizontal lines, which hold, from top to bottom, class name, the list of attributes, and the list of operations. A class of person is shown in Exhibit 2.

A class diagram allows you to document how the class relates to other classes. The class diagram doesn't fix the actual implementation. The actual code might not be a direct translation of the diagram. However the functionality of the code will remain the same. In the Exhibit 3 you can observe the following:

Class A is a class that has some attributes (variable) and functions (function).

Class A is the parent class of Sub-Class

Class A is associated with Class B. In this case only 1 Class B is associated with possibly multiple versions of Class A. The association is named something. Depending on the type of UML diagrams you are using, something can be implemented using a variable or by something else.

Class T is a template class.

Class B is an instance of Class T where the parameter T is bound to int.

General is an associate class that can add attributes (or function) to the association between Class A and Class B. In the present case, it adds the variable period.

UML - Behavioral Diagram vs Structural Diagram

The Unified Modeling Language is a standardized general-purpose modeling language and nowadays is managed as a de facto industry standard by the **Object Management Group (OMG)**. The creation of UML was originally motivated by the desire to standardize the disparate notational systems and approaches to software design. It was developed by Grady Booch, Ivar Jacobson, and James Rumbaugh at Rational Software in 1994–1995, with further development led by them through 1996.

tatic vs Dynamic View

Static modeling is used to specify the structure of the objects, classes or components that exist in the problem domain. These are expressed using class, object or component. While dynamic modeling refers to representing the object interactions during runtime. It is represented by sequence, activity, collaboration, and state. UML diagrams represent these two aspects of a system:

Structural (or Static) view: emphasizes the **static structure** of the system using objects, attributes, operations and relationships. It includes class diagrams and composite structure diagrams.

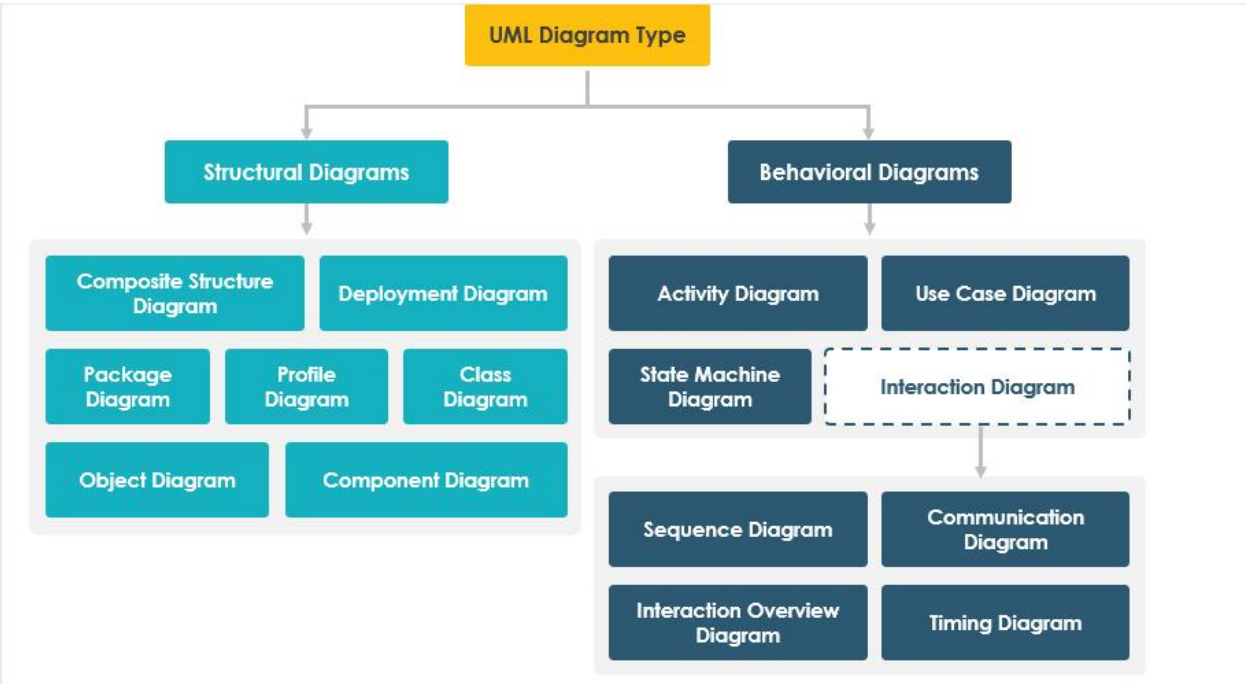
Behavioral (or Dynamic) view: emphasizes the **dynamic behavior** of the system by showing **collaborations** among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams, and state machine diagrams.

In UML 2.2 there are 14 types of UML diagrams, which are divided into these two categories:

7 diagram types represent **structural** information

Another 7 represents general UML diagram types for **behavioral** modeling, including four that represent different aspects of interactions.

These diagrams can be categorized hierarchically as shown in the following UML diagram map:



Behavioral Diagrams

UML's five behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system. It shows how the system behaves and interacts with itself and other entities (users, other systems). They show how data

moves through the system, how objects communicate with each other, how the passage of time affects the system, or what events cause the system to change internal states. Since behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems. As an example, the activity diagram describes the business and operational step-by-step activities of the components in a system.

In other words, a behavioral diagram shows how the system works 'in motion', that is how the system interacts with external entities and users, how it responds to input or event and what constraints it operates under. There are seven behavioral diagrams that you can model the dynamics of a system as listed in the Table below:

Behavioral Diagrams

Activity Diagram: It is a graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency.

Use Case Diagram: It describes a system's functional requirements in terms of use cases that enable you to relate what you need from a system to how the system delivers on those needs.

State Machine Diagram: It shows the discrete behavior of a part of a designed system through finite state transitions.

Sequence Diagram: It shows the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.

Communication Diagram: It shows interactions between objects and/or parts (represented as lifelines) using sequenced messages in a free-form arrangement.

Interaction Overview Diagram: It depicts a control flow with nodes that can contain other interaction diagrams.

Timing Diagram: It shows interactions when the primary purpose of the diagram is to reason about time by focusing on conditions changing within and among lifelines along a linear time axis.

tructural Diagrams

Structure diagrams depict the static structure of the elements in your system. i.e., how one object relates to another. It shows the things in the system – classes, objects, packages or modules, physical nodes, components, and interfaces. For example, just as the static aspects of a house encompass the existence and placement of such things as walls, doors, windows, pipes, wires, and vents. The Seven UML structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.

Since structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems.

For example, the component diagram describes how a software system is split up into components and shows the dependencies among these components.

Composite Structure Diagram: It shows the internal structure of a classifier, classifier interactions with the environment through ports, or behavior of a collaboration.

Deployment Diagram: It shows a set of nodes and their relationships that illustrates the static deployment view of an architecture.

Package Diagram: It groups related UML elements into a collection of logically related UML structure.

Class Diagram: It shows a set of classes, interfaces, and collaborations and their relationships, typically, found in modeling object-oriented systems.

Object Diagram: It shows a set of objects and their relationships, which is the static snapshots of instances of the things found in class diagrams.

Component Diagram: It shows a set of components and their relationships that illustrates the static implementation view of a system.

Structural Diagrams

Structure diagrams depict the static structure of the elements in your system. i.e., how one object relates to another. It shows the things in the system – classes, objects, packages or modules, physical nodes, components, and interfaces. For example, just as the static aspects of a house encompass the existence and placement of such things as walls, doors, windows, pipes, wires, and vents. The Seven UML structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.Since structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems.For example, the component diagram describes how a software system is split up into components and shows the dependencies among these components.

Composite Structure Diagram: It shows the internal structure of a classifier, classifier interactions with the environment through ports, or behavior of a collaboration.

Deployment Diagram: It shows a set of nodes and their relationships that illustrates the static deployment view of an architecture.

Package Diagram: It groups related UML elements into a collection of logically related UML structure.

Class Diagram: It shows a set of classes, interfaces, and collaborations and their relationships, typically, found in modeling object-oriented systems.

Object Diagram: It shows a set of objects and their relationships, which is the static snapshots of instances of the things found in class diagrams.

Component Diagram: It shows a set of components and their relationships that illustrates the static implementation view of a system.