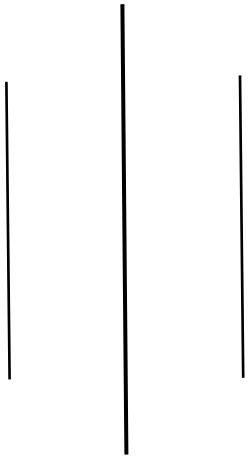
Tribhuvan University

Institute of Science and Technology

Central Department of Computer Science and Information Technology



OBJECT-ORIENTED SOFTWARE ENGINEERING (OOSE)

Assignment - II

Submitted by:
Priya Shrestha
Roll No: 24

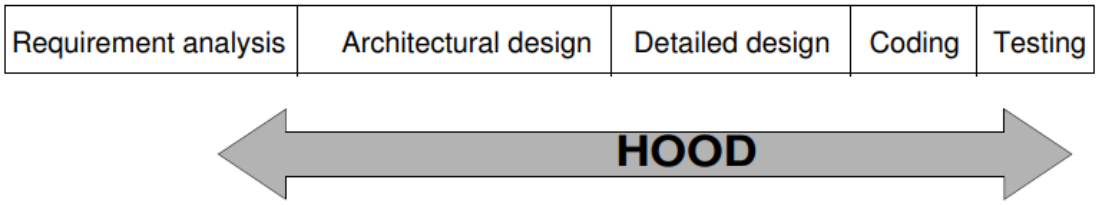
Submitted to:
Prof. Dr. Subarna Shakya
TU, CDCSIT

Date: May 5, 2025

1. Comparison between Hierarchical Object-Oriented Design vs Object Modeling Technique vs Responsibility Driven Design.

Hierarchical Object-Oriented Design (HOOD)

- HOOD is a detailed software design method based on hierarchical decomposition of a software problems which comprises textual and graphical representations of a design.
- It primarily focuses on the hierarchical decomposition of systems into objects with clear parent-child relationships.
- This method starts after requirement analysis which extends down to coding and testing.



- Initially created for European Space Agency (ESA) and remains influential in real-time and embedded systems.
- It consists of following key principles:
  - ❖ Top-down Hierarchical decomposition
  - ❖ Object-oriented paradigm.
  - ❖ Design phases and formalized design process
  - ❖ Graphical and Textual representation
  - ❖ Language Independence

Top-down Hierarchical decomposition

- ➢ Hood starts by decomposing a system into a hierarchy of objects, where each object represents a component of function of the system.
- ➢ This top-down approach ensures that complexity is managed by breaking the system into manageable levels.

Object-oriented paradigm

- ➢ Hood emphasizes strong encapsulation, inheritance and polymorphism.
- ➢ Each object has a clear interface and well-defined responsibilities.

Design phases and Formalized Design process

HOOD typically includes three main design levels:

- ➢ **Abstract Level**: Defines high-level objects and their interactions (what the system does).
- ➢ **Design Level**: Refines abstract objects into concrete ones (how the system does it).
- ➢ **Implementation Level**: Maps the design into actual code structures in the chosen programming language.

HOOD follows a structured design approach with defined steps:

- **Problem Definition** – Identify system requirements.
- **Object Identification** – Decompose the system into objects.
- **Object Definition** – Specify interfaces and behaviors.
- **Hierarchy Construction** – Organize objects in a parent-child structure.
- **Verification & Validation** – Ensure correctness and consistency.

Graphical and Textual Representation

- The system hierarchy and object relationships are often visualized through diagrams.
- Textual specifications define interfaces, behavior, and data structures.

Language Independence

- Although object-oriented, HOOD is not tied to a specific programming language.
- It has been used with languages like Ada, C++, and Java.

- They are used when systems require strict modularity and when the system is embedded/real-time.
- They are not directly related to Unified Modeling Language.
- They use HOOD diagrams and textual descriptions as a notation/tools.
- Applications in diverse domains: aerospace, defense systems, industrial automation, safety-critical embedded systems etc.

Strengths

- Modularity: Encourages reusable and maintainable components.
- Abstraction: Hides implementation details behind interfaces.
- Scalability: Hierarchical structure supports large systems.
- Traceability: Links design to requirements.

Weakness

- Rigid hierarchy may limit flexibility.
- Less emphasis on dynamic behavior.

Object Modeling Technique (OMT)

- OMT is a real-world-based object modeling approach for software modeling and designing.
- It primarily focuses on the combination of data, behavior, and interaction modeling using three main models (object, dynamic, function).
- OMT was introduced by James Rumbaugh et al. as a precursor/predecessor of the Unified Modeling Language (UML). Thus, many OMT modeling elements are common to UML.
- It has proposed three main orthogonal models:
    - ❖ Object Model (Static structure)
    - ❖ Dynamic Model (State transitions)
    - ❖ Functional Model (Data Flow and Processes)

Object Model (Static structure)

- ➢ It represents the static structure of the system using objects, classes and relationships.
- ➢ It uses class diagrams (Similar to UML class diagrams).
- ➢ Key elements: classes, associations, inheritance and aggregation.

Dynamic Model (State transitions)

- ➢ It represents the time dependent behavior of the system.
- ➢ It uses state transition diagrams (similar to UML state machines) and event-flow diagrams.
- ➢ Key concepts: states, transitions, and events.

Functional Model (Data Flow and Processes)

- ➢ It redescribes the data transformations and functional processes.
- ➢ It uses data flow diagrams (DFDs) similar to traditional structures analysis.
- ➢ Key concepts: processes, dataflows, actors and data stores.

- Various phases of OMT includes: Analysis, System Design, Object Designn and Implementation.

Analysis

This is the first phase of the object modeling technique. This phase involves the preparation of precise and correct modelling of the real world problems. Analysis phase starts with setting a goal i.e. finding the problem statement. Problem statement is further divided into above discussed three models i.e. object, dynamic and functional model.

System Design

This is the second phase of the object modeling technique and it comes after the analysis phase. It determines all system architecture, concurrent tasks and data storage. High level architecture of the system is designed during this phase.

Object Design

Object design is the third phase of the object modelling technique and after system design is over, this phase comes. Object design phase is concerned with classification of objects into different classes and about attributes and necessary operations needed. Different issues related with generalization and aggregation are checked.

Implementation

This is the last phase of the object modeling technique. It is all about converting prepared design into the software. Design phase is translated into the Implementation phase.

- They are used for the data-intensive systems and early UML adopters.
- They are direct precursor to UML (merged with Booch and others).
- As a notation/tools , OMT diagrams were used and later integrated into UML.
- Diverse Application Domains: Banking Transaction systems, middleware and distributed systems, Flight control systems, supply chain management systems etc.

Strengths

- It provides a structures approach for analysing, designing and conceptualizing software systems based on object-oriented paradigm.
- Divides the system into **three complementary models** (object, dynamic, and functional), which improves clarity and modularity.
- There is a comprehensive coverage of system aspects.
- It uses the precise notation (evolved into UML).
- Scales well for large and complex systems due to its structured and layered approach.

Weakness

- **Steep Learning Curve:** Beginners may find the three-model system complex and overwhelming. Less emphasis on dynamic behavior.
- **Limited Support for Iterative Development:** More aligned with **waterfall-style** development; adapting it to agile methodologies requires effort.
- **Less Emphasis on Implementation:** Focuses more on analysis and design, with limited guidance on actual **coding or deployment**.
- **Redundancy Across Models:** Some overlap between models (e.g., between object and dynamic models) can lead to **duplication of effort**.
- **Not suited for all paradigms:** Works best for object-oriented systems; less applicable to procedural or functional programming models.

Responsibility Driven Design (RDD)

- RDD is software design that emphasizes identifying and assigning responsibilities to objects within a software system.
- It primarily focuses on assigning responsibilities to objects and collaboration between them.
- **Responsibilities in RDD** refer to the specific tasks or behaviors that an object is responsible for performing within the software system.
- There are three types of responsibilities in RDD, namely:
  Collaborative Responsibility: This type of responsibility refers to the responsibility of an object to collaborate with other objects to achieve a specific goal.
  Informational Responsibility: This type of responsibility refers to the responsibility of an object to provide information to other objects in the system.
  Computational Responsibility: This type of responsibility refers to the responsibility of an object to perform computations or calculations within the system.

- Key Principles of RDD:
  - **Maximize Abstraction**: Abstract away unnecessary details and focus on the essential behavior of each object.
  - **Distribute Behavior:** Assign responsibilities to different objects rather than concentrating them in a single class.
  - **Preserve Flexibility:** Design in a way that allows for changes and additions without disrupting the existing structure.

- Key Concepts in RDD:
  Application: A set of interacting objects.
  Object: An implementation of one or more roles.
  Role: A set of related responsibilities.
  Responsibility: An obligation to perform a task or know information.
  Collaboration: An interaction between objects or roles.
  Contract: An agreement outlining the terms of a collaboration.

- It was proposed by Rebecca Wirfs-Brock as a collaborative design approach.
- They are used for agile/iterative development and collaborative design sessions.
- They are influenced UML collaboration diagrams but not a core part.
- As a notation/tools, CRC cards and Informal sketches were used.
- Diverse Application Domains: E-commerce systems, API Design, Test-Driven Development, service boundaries etc.

Strengths

- It encourages flexible, adaptable designs.
- They are human centric and intuitive.
- Encourages loose coupling (objects interact via contracts, not tight dependencies).
- Adaptable to change (new responsibilities can be added without major redesign).
- Improves maintainability (clear separation of concerns).

Weakness

- They may lack rigor for large-scale systems.
- They use less formal notation.
- Less formal than UML/HOOD (no standard notation).
- Requires experience to assign responsibilities effectively.
- Not ideal for real-time systems (HOOD is better for strict hierarchies).