SNS COLLEGE OF ENGINEERING

Kurumbapalayam (po), Coimbatore – 641 107

Accredited by NAAC-UGC with 'A' Grade

Approved by AICTE & Affiliated to Anna University, Chennai

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Unit 3

Functional Independence in Object-Oriented Software Engineering

Definition

Functional Independence refers to designing software components (classes, modules, or functions) in a way that they have **minimal dependencies** on each other. This makes the system **easier to maintain, test, and scale**.

Key Principles of Functional Independence

Functional independence is achieved through **two key properties**:

1. **High Cohesion** – A module should focus on a **single task**.
2. **Low Coupling** – A module should have **minimal dependence** on other modules.

Importance of Functional Independence

- **Enhances Reusability** – Independent modules can be reused in different parts of the application.
- **Improves Maintainability** – Changes in one module do not significantly impact others.
- **Simplifies Testing** – Independent modules can be tested separately.
- **Boosts Scalability** – New functionalities can be added without affecting existing components.

Example of Functional Independence

Scenario: Library Management System

Imagine a **Library Management System** with the following modules:

1. **User Management Module** – Handles user registration and authentication.
2. **Book Catalog Module** – Manages book details (title, author, availability).
3. **Borrowing System Module** – Handles book borrowing and returning.

With Functional Independence:

- The **User Management Module** does not depend on the **Book Catalog Module** to function.
- The **Borrowing System Module** interacts with both but is not tightly linked to their internal implementations.
- Changes in **Book Catalog** (e.g., adding a new book category) do not affect **User Management**.

Thus, each module operates **independently**, following **high cohesion** (each module has a specific responsibility) and **low coupling** (minimal dependency between modules).

Design Patterns in Object-Oriented Software Engineering

1. Introduction to Design Patterns

Design patterns are **reusable solutions to common problems** that occur in software design. Instead of solving problems from scratch, developers can use these templates to create **efficient, maintainable, and scalable** software.

Why Use Design Patterns?

- **Reusability** – Reduces redundancy in code.
- **Maintainability** – Makes code easier to modify and updates
- **Scalability** – Allows adding new features without major changes.
- **Best Practices** – Follows industry-standard approaches.

2. Categories of Design Patterns

Design patterns are broadly classified into **three main categories**:

(A) Creational Design Patterns – Focus on object creation in a flexible and reusable way.

- **Example Patterns**: Singleton, Factory Method, Builder
- **Use Case**: When object creation is complex or needs controlled access.

(B) Structural Design Patterns – Focus on object composition and relationships.

- **Example Patterns**: Adapter, Composite, Decorator
- **Use Case**: When objects need to work together efficiently despite interface differences.

(C) Behavioral Design Patterns – Focus on object interaction and communication.

- **Example Patterns**: Observer, Strategy, Command
- **Use Case**: When objects need to communicate dynamically.

3. Common Design Patterns (With Detailed Examples)

(A) Singleton Pattern (Creational)

Purpose:

- Ensures that only **one instance** of a class is created.
- Provides **a global access point** to that instance.

Example Scenario:
Imagine a **National Identity Database** that stores all citizen records.

- There should be only **one central database instance**.
- If multiple instances exist, it could cause **data inconsistency**.

Real-Life Analogy:

- A **president of a country** – there is only **one leader at a time**.
- A **printer spooler** – only **one** instance manages printing jobs to avoid conflicts.

(B) Factory Method Pattern (Creational)

Purpose:

- Provides an interface for creating objects, **but lets subclasses decide which class to instantiate**.
- Promotes **loose coupling** (object creation logic is separated).

Example Scenario:
Imagine a **Shape Factory** that creates different shapes (Circle, Rectangle, Square).

- Instead of manually creating objects, we call the factory, and it decides which shape to return.

Real-Life Analogy:

- A **car factory** – you place an order for a car model, and the factory **produces it** without you building it manually.
- A **restaurant menu** – you order a dish, and the kitchen **prepares the correct meal** based on your request.

(C) Adapter Pattern (Structural)

Purpose:

- Allows two **incompatible interfaces** to work together.
- Acts as a **bridge** between systems that otherwise couldn't communicate.

Example Scenario:
A **USB-to-HDMI Adapter** allows a **USB device** to connect to an **HDMI port**, even though their interfaces are different.

Real-Life Analogy:

- A **language translator** – a person who speaks **English and Chinese** can help two people communicate even if they don't speak the same language.
- A **power socket adapter** – lets you use an **American plug in a European socket**.

(D) Observer Pattern (Behavioral)

Purpose:

- Defines a **dependency** between objects so that when one object changes, all dependent objects are **notified automatically**.
- Used in event-driven applications where multiple objects need to react to state changes.

Example Scenario:
A **news website** allows users to **subscribe** to updates.

- When a new article is published, **all subscribers receive a notification** automatically.

Real-Life Analogy:

- A **YouTube subscription** – when a YouTuber uploads a new video, all **subscribers get notified**.
- A **fire alarm system** – when smoke is detected, **all alarms in the building go off simultaneously**.

4. Choosing the Right Design Pattern-Design Patterns use when with example

Design Pattern	**Use When**	**Example**
Singleton	You need **only one instance** of a class	**Database connection, Printer spooler**
Factory Method	Object creation **needs to be controlled**	**Car factory, Restaurant order system**
Adapter	Two incompatible interfaces need to **work together**	**Power adapter, Language translator**
Observer	One object's change needs to **notify multiple objects**	**YouTube subscriptions, Fire alarm system**