Self-Study Assignment: Part 1

Name: Bankar Krushna Lahanubhau

Batch : S2

DIV: 1

Design Patterns in Java

1. Introduction to Design Patterns

What are Design Patterns?

Design patterns are **proven solutions to common software design problems**. They represent best practices and help developers write **reusable, scalable, and maintainable code**.

Why Use Design Patterns?

- **Improves Code Reusability**: Standardized solutions can be reused across different projects.
- **Enhances Maintainability**: Well-structured code is easier to understand and modify.
- **Promotes Best Practices**: Encourages object-oriented principles like **Encapsulation, Abstraction, Inheritance, and Polymorphism**.
- **Increases Development Speed**: Reduces development time by providing tested solutions.

History of Design Patterns

The concept was popularized by the book **"Design Patterns: Elements of Reusable Object-Oriented Software"** (1994) by the **Gang of Four (GoF)**:

- **Erich Gamma**
- **Richard Helm**
- **Ralph Johnson**
- **John Vlissides**

---

2. Types of Design Patterns

There are **three main categories** of design patterns:

1. **Creational Patterns** (Deals with object creation)
2. **Structural Patterns** (Deals with object composition)
3. **Behavioral Patterns** (Deals with object interaction)

1. Creational Design Patterns

These patterns handle the process of **creating objects efficiently**.
Common Creational Patterns:

- **Singleton**: Ensures only one instance of a class exists.
- **Factory Method**: Creates objects without exposing the instantiation logic.
- **Abstract Factory**: Provides an interface for creating related objects.
- **Builder**: Constructs complex objects step-by-step.
- **Prototype**: Creates a new object by copying an existing object.

Singleton Pattern

```
class Singleton {
    private static Singleton instance;

    private Singleton() {} // Private constructor prevents instantiation

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

2. Structural Design Patterns

These patterns deal with **class composition** and **relationship between objects**.
Common Structural Patterns:

- **Adapter**: Converts one interface into another.
- **Bridge**: Separates abstraction from implementation.
- **Composite**: Treats individual objects and groups of objects the same way.
- **Decorator**: Adds behavior to an object dynamically.
- **Facade**: Provides a simplified interface to a complex system.
- **Flyweight**: Reduces memory usage by sharing objects.
- **Proxy**: Controls access to another object.

Adapter Pattern

```
// Existing interface
interface MediaPlayer {
    void play(String fileName);
}

// New interface
interface AdvancedMediaPlayer {
    void playAdvanced(String fileName);
}

// Adapter class
class MediaAdapter implements MediaPlayer {
    private AdvancedMediaPlayer advancedMediaPlayer;

    public MediaAdapter(AdvancedMediaPlayer advancedMediaPlayer) {
        this.advancedMediaPlayer = advancedMediaPlayer;
    }

    @Override
    public void play(String fileName) {
        advancedMediaPlayer.playAdvanced(fileName);
    }
}
```

3. Behavioral Design Patterns

These patterns define **how objects communicate** and interact.
Common Behavioral Patterns:

- **Chain of Responsibility**: Passes requests along a chain of handlers.
- **Command**: Encapsulates a request as an object.
- **Interpreter**: Implements a language interpreter.
- **Iterator**: Provides a way to access elements sequentially.
- **Mediator**: Centralizes communication between objects.
- **Memento**: Captures and restores an object's state.
- **Observer**: Defines a dependency between objects so that when one changes, others are notified.
- **State**: Changes behavior based on the object's state.
- **Strategy**: Defines a family of algorithms and selects one dynamically.
- **Template Method**: Defines the structure of an algorithm but lets subclasses implement specific steps.
- **Visitor**: Adds new behavior without modifying existing classes.

Observer Pattern

```
import java.util.ArrayList;
import java.util.List;

// Observer Interface
interface Observer {
    void update(String message);
}

// Subject (Observable)
class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}

// Concrete Observer
class ConcreteObserver implements Observer {
    private String name;

    public ConcreteObserver(String name) {
        this.name = name;
    }

    @Override
    public void update(String message) {
        System.out.println(name + " received update: " + message);
    }
}

// Usage
public class ObserverPatternExample {
    public static void main(String[] args) {
        Subject subject = new Subject();
        Observer observer1 = new ConcreteObserver("Observer 1");
        Observer observer2 = new ConcreteObserver("Observer 2");

        subject.addObserver(observer1);
        subject.addObserver(observer2);

        subject.notifyObservers("New Update!");
    }
}
```

---

3. Application of Design Patterns

Design patterns are used in **real-world applications** to improve software design.

Pattern	Application Example
Singleton	**Database connections, Logger, Configuration settings**
Factory Method	**GUI frameworks, JDBC Driver Manager**
Builder	**Creating complex objects like Documents, JSON parsers**
Adapter	**Connecting different API services, Third-party libraries**
Decorator	**Adding features dynamically in GUI components, Streams API**
Observer	**Event handling, Notification systems**
Strategy	**Sorting algorithms, Payment processing systems**
Command	**Undo/Redo functionality in text editors, GUI buttons**
Proxy	**Security proxies, Caching in databases**

Conclusion

- **Design patterns** provide standard solutions to **common software problems**.
- They are **categorized into three types**: **Creational, Structural, and Behavioral**.
- Using the **right design pattern** makes the code **scalable, reusable, and maintainable**.
- **Java provides built-in support** for several design patterns (e.g., Singleton using `enum`).
- Design patterns are widely used in **real-world applications** like **GUI development, Database Management, Web Frameworks, and Distributed Systems**.