Chapter 11: Design Patterns in Java

Introduction

In software engineering, **design patterns** are tried-and-tested solutions to common problems that occur in software design. In Java, design patterns enable developers to write cleaner, more modular, and more maintainable code by promoting code reuse and best practices. These patterns serve as blueprints for solving design problems across various software development scenarios.

This chapter covers the **three main categories of design patterns—Creational**, **Structural**, and **Behavioral**—with practical examples and real-world applications in Java.

11.1 What are Design Patterns?

A **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design. It is **not code** itself, but a **template** for how to solve a problem.

Why Use Design Patterns?

- Encourage best practices and robust architecture
- Increase development speed by reusing proven solutions
- Improve code readability and maintainability
- Promote **loose coupling** and **high cohesion**

11.2 Categories of Design Patterns

Design Patterns are generally grouped into **three categories**:

1. **Creational Patterns** – Concerned with object creation
2. **Structural Patterns** – Concerned with object composition
3. **Behavioral Patterns** – Concerned with object interaction and responsibility

---

11.3 Creational Design Patterns

These patterns deal with object creation mechanisms.

11.3.1 Singleton Pattern

Ensures a class has only one instance and provides a global point of access to it.

```
javaCopy codepublic class Singleton {
    private static Singleton instance;

    private Singleton() { }

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

11.3.2 Factory Method Pattern

Defines an interface for creating an object but lets subclasses alter the type of objects that will be created.

```
javaCopy codeinterface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

class ShapeFactory {
    public Shape getShape(String type) {
        if (type.equalsIgnoreCase("circle")) return new Circle();
        return null;
    }
}
```

11.3.3 Abstract Factory Pattern

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

```
javaCopy codeinterface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

class WinFactory implements GUIFactory {
    public Button createButton() {
        return new WinButton();
    }

    public Checkbox createCheckbox() {
        return new WinCheckbox();
    }
}
```

11.3.4 Builder Pattern

Used to build complex objects step-by-step.

```
javaCopy codeclass Computer {
    private String CPU;
    private String RAM;

    public static class Builder {
        private String CPU;
        private String RAM;

        public Builder setCPU(String CPU) {
            this.CPU = CPU;
            return this;
        }

        public Builder setRAM(String RAM) {
            this.RAM = RAM;
            return this;
        }

        public Computer build() {
            Computer c = new Computer();
            c.CPU = this.CPU;
            c.RAM = this.RAM;
            return c;
        }
    }
}
```

11.3.5 Prototype Pattern

Used to create duplicate objects while keeping performance in mind.

```
javaCopy codeclass Shape implements Cloneable {
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

11.4 Structural Design Patterns

These patterns deal with object composition—how classes and objects can be combined.

11.4.1 Adapter Pattern

Allows the interface of an existing class to be used as another interface.

```
javaCopy codeclass MediaPlayer {
    public void play(String audioType, String fileName) {
        // Implementation
    }
}

class AudioAdapter extends MediaPlayer {
    private AdvancedPlayer advancedPlayer = new AdvancedPlayer();

    @Override
    public void play(String audioType, String fileName) {
        if(audioType.equals("vlc")) {
            advancedPlayer.playVLC(fileName);
        }
    }
}
```

11.4.2 Decorator Pattern

Adds responsibilities to objects dynamically.

```
javaCopy codeinterface Coffee {
    String getDescription();
    int getCost();
}

class SimpleCoffee implements Coffee {
    public String getDescription() { return "Simple Coffee"; }
    public int getCost() { return 5; }
}

class MilkDecorator implements Coffee {
    private Coffee coffee;
    public MilkDecorator(Coffee coffee) { this.coffee = coffee; }

    public String getDescription() {
        return coffee.getDescription() + ", Milk";
    }

    public int getCost() {
        return coffee.getCost() + 2;
    }
}
```

11.4.3 Composite Pattern

Used where you need to treat individual objects and compositions of objects uniformly.

```
javaCopy codeinterface Employee {
    void showDetails();
}

class Developer implements Employee {
    public void showDetails() {
        System.out.println("Developer");
    }
}

class Manager implements Employee {
    List<Employee> employees = new ArrayList<>();

    public void add(Employee e) { employees.add(e); }

    public void showDetails() {
        for (Employee e : employees) {
            e.showDetails();
        }
    }
}
```

11.5 Behavioral Design Patterns

These patterns are about how objects communicate with each other.

11.5.1 Observer Pattern

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.

```
javaCopy codeinterface Observer {
    void update(String message);
}

class ConcreteObserver implements Observer {
    public void update(String message) {
        System.out.println("Message received: " + message);
    }
}

class Subject {
    private List<Observer> observers = new ArrayList<>();
    public void addObserver(Observer o) { observers.add(o); }
    public void notifyObservers(String message) {
        for (Observer o : observers) {
            o.update(message);
        }
    }
}
```

11.5.2 Strategy Pattern

Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

```
javaCopy codeinterface Strategy {
    int execute(int a, int b);
}

class AddStrategy implements Strategy {
    public int execute(int a, int b) {
        return a + b;
    }
}

class Context {
    private Strategy strategy;
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}
```

11.5.3 Command Pattern

Encapsulates a request as an object, thereby letting users parameterize clients with different requests.

```
javaCopy codeinterface Command {
    void execute();
}

class Light {
    void turnOn() {
        System.out.println("Light ON");
    }
}

class LightOnCommand implements Command {
    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }
}
```

11.6 Real-world Applications of Design Patterns in Java

- **Singleton**: Logger, Configuration manager
- **Factory**: GUI libraries, JDBC drivers
- **Observer**: Event-driven systems (e.g., UI)
- **Strategy**: Sorting algorithms
- **Decorator**: Java I/O Streams (`BufferedReader`, `InputStreamReader`)
- **Adapter**: `java.util.Arrays.asList()`

Summary

Design patterns provide elegant and scalable solutions to common software design problems. They allow for code reusability, easier maintenance, and improved software architecture. In Java, mastering design patterns is essential for building robust enterprise applications.

Key Takeaways:

- Design patterns are templates for solving common problems.
- There are 3 main categories: **Creational**, **Structural**, and **Behavioral**.
- Common patterns include **Singleton**, **Factory**, **Adapter**, **Observer**, and **Strategy**.

- Java’s standard libraries themselves make extensive use of these patterns.
