Java Design Patterns – Details Guide

1. Singleton Pattern

Goal: Ensure only one instance of a class is created globally.

Technologies Used:

- Private constructor
- Static instance method
- Enum-based Singleton (recommended)

Implementation:

```
// Lazy Initialization Singleton
public class Logger {
    private static Logger instance;

    private Logger() {
        // private constructor
    }

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

// Enum Singleton
public enum DatabaseConnection {
    INSTANCE;
    public void connect() {
        // Connection logic here
    }
}
```

Use Cases:

- Logging systems
- Database connection pool

2. Factory Pattern

Goal: Create objects without exposing the instantiation logic.

Technologies Used:

- Factory classes or static methods
- Interfaces and polymorphism

Implementation:

```
// Shape Interface
public interface Shape {
    void draw();
}

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

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

// Factory Class
public class ShapeFactory {
    public static Shape getShape(String type) {
        switch(type.toLowerCase()) {
```

```
            case "circle": return new Circle();
            case "square": return new Square();
            default: throw new IllegalArgumentException("Unknown shape
type");
        }
    }
}
```

Use Cases:

- UI components
- JDBC DriverManager.getConnection()

3. Observer Pattern

Goal: One-to-many dependency between objects so when one changes, dependents are notified.

Technologies Used:

- Java Observer/Observable (Deprecated, use custom implementation or listeners)
- Event-driven programming

Implementation:

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

// Concrete Observer
public class EmailSubscriber implements Observer {
    public void update(String message) {
        System.out.println("Email received: " + message);
    }
}
```

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

    public void subscribe(Observer o) {
        observers.add(o);
    }

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

Use Cases:

- UI event listeners (e.g., button clicks)
- Notification systems (e.g., Kafka consumers)

4. Strategy Pattern

Goal: Define a family of algorithms and make them interchangeable at runtime.

Technologies Used:

- Strategy interface
- Composition over inheritance

Implementation:

```
// Strategy Interface
public interface PaymentStrategy {
    void pay(int amount);
}

// Concrete Strategies
```

```
public class CreditCardPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " via Credit Card");
    }
}

public class PayPalPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " via PayPal");
    }
}

// Context Class
public class PaymentContext {
    private PaymentStrategy strategy;

    public void setStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void pay(int amount) {
        strategy.pay(amount);
    }
}
```

Use Cases:

- Payment gateways
- Sorting with different comparators

5. Builder Pattern

Goal: Construct complex objects step-by-step.

Technologies Used:

- Fluent API

- Java Builder pattern or Lombok `@Builder`

Implementation:

```
// POJO
public class User {
    private String name;
    private String address;
    private String phone;

    private User(UserBuilder builder) {
        this.name = builder.name;
        this.address = builder.address;
        this.phone = builder.phone;
    }

    public static class UserBuilder {
        private String name;
        private String address;
        private String phone;

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

        public UserBuilder address(String address) {
            this.address = address;
            return this;
        }

        public UserBuilder phone(String phone) {
            this.phone = phone;
            return this;
        }

        public User build() {
            return new User(this);
        }
    }
```

```
}
```

Use Cases:

- User object with optional fields
- Constructing HTTP requests in web clients

6. Adapter Pattern

Goal: Allow incompatible interfaces to work together.

Technologies Used:

- Wrapper classes

Implementation:

```
// Existing Interface
public interface MediaPlayer {
    void play(String audioType, String fileName);
}

// Advanced Interface
public interface AdvancedMediaPlayer {
    void playMp4(String fileName);
}

// Concrete Implementation
public class Mp4Player implements AdvancedMediaPlayer {
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file: " + fileName);
    }
}

// Adapter Class
public class MediaAdapter implements MediaPlayer {
    private AdvancedMediaPlayer advancedMusicPlayer;
```

```
    public MediaAdapter(String audioType) {
        if(audioType.equalsIgnoreCase("mp4")) {
            advancedMusicPlayer = new Mp4Player();
        }
    }

    public void play(String audioType, String fileName) {
        if(audioType.equalsIgnoreCase("mp4")) {
            advancedMusicPlayer.playMp4(fileName);
        }
    }
}
```

Use Cases:

- Integrating legacy APIs
- Interface compatibility

7. Prototype Pattern

Goal: Create new objects by copying existing ones.

Technologies Used:

- `clone()` method or copy constructors

Implementation:

```
public abstract class Shape implements Cloneable {
    private String id;
    protected String type;

    public abstract void draw();

    public Object clone() {
        Object clone = null;
```

```
        try {
            clone = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clone;
    }
}
```

Use Cases:

- Object creation overhead reduction
- When object creation is costly

8. Command Pattern

Goal: Encapsulate a request as an object.

Technologies Used:

- Command interface
- Invoker and Receiver classes

Implementation:

```
public interface Command {
    void execute();
}

public class Light {
    public void on() {
        System.out.println("Light is ON");
    }
}

public class LightOnCommand implements Command {
    private Light light;
```

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

    public void execute() {
        light.on();
    }
}

public class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}
```

Use Cases:

- GUI buttons
- Task scheduling

9. Decorator Pattern

Goal: Add behavior to objects dynamically.

Technologies Used:

- Wrapper classes

Implementation:

```
public interface Coffee {
    String getDescription();
    double cost();
}

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

public class MilkDecorator implements Coffee {
    private Coffee coffee;

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

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

    public double cost() {
        return coffee.cost() + 1.5;
    }
}
```

Use Cases:

- UI component enhancements
- Flexible feature additions

10. Proxy Pattern

Goal: Provide a placeholder or surrogate for another object.

Technologies Used:

- RealSubject and Proxy classes

Implementation:

```
public interface Image {
    void display();
}

public class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();
    }

    private void loadFromDisk() {
        System.out.println("Loading " + filename);
    }

    public void display() {
        System.out.println("Displaying " + filename);
    }
}

public class ProxyImage implements Image {
    private RealImage realImage;
    private String filename;

    public ProxyImage(String filename) {
        this.filename = filename;
    }
```

```
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);
        }
        realImage.display();
    }
}
```

Use Cases:

- Lazy loading
- Access control