Design Patterns

1. Observer Design Pattern (Behavioral Pattern)

What is Observer Pattern?

The Observer Pattern defines a **one-to-many relationship** between objects. When the state of one object (called the **Subject**) changes, all its dependent objects (called **Observers**) are **automatically notified and updated**.

Simple Explanation

- One object changes
- Many other objects get notified automatically
- Observers do not need to constantly check the subject

Real-Life Example

YouTube Subscription System - YouTube Channel → Subject - Subscribers → Observers - When a new video is uploaded, all subscribers receive notifications automatically

Key Components

- **Subject**: Maintains a list of observers and sends notifications
- **Observer**: Receives updates from the subject

Simple Code Structure (Java)

```
interface Observer {
    void update(int value);
}

interface Subject {
    void attach(Observer o);
    void notifyObservers();
}
```

Step 3: Concrete Subject

```
class Data implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private int value;
    public void attach(Observer o) {
        observers.add(o);

    }
```

```
    public void setValue(int value) {
        this.value = value;
        notifyObservers();
    }
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(value);
        }
    }
}
```

Step 4: Concrete Observer

```
class Display implements Observer {
    public void update(int value) {
        System.out.println("Updated value: " + value);
    }
}
```

Step 5: Usage

```
Data data = new Data();
Display d1 = new Display();
data.attach(d1);
data.setValue(10);
```

✓ Output:

```
Updated value: 10
```

Why Use Observer Pattern?

- Event handling systems
- Notification systems
- GUI listeners

Exam One-Liner

Observer Pattern defines a one-to-many dependency where observers are notified automatically when the subject changes state.

2. Strategy Design Pattern (Behavioral Pattern)

What is Strategy Pattern?

The Strategy Pattern allows selecting an **algorithm or behavior at runtime** from a family of algorithms.

Simple Explanation

- Same task
- Different ways to do it
- Behavior can be changed without modifying the main class

Real-Life Example

Google Maps Navigation - Destination is the same - Travel methods can change: Car, Walking, Bicycle

Key Components

- **Strategy Interface**: Declares a common method
- **Concrete Strategies**: Implement different behaviors
- **Context**: Uses a strategy object

Simple Code Structure (Java)

```java
interface PaymentStrategy {
    void pay(int amount);
}

class CashPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid using cash");
    }
}
```

Why Use Strategy Pattern?

- Avoid large if-else statements
- Easily switch behaviors
- Clean and flexible design

Exam One-Liner

Strategy Pattern allows selecting an algorithm's behavior at runtime.

3. Decorator Design Pattern (Structural Pattern)

What is Decorator Pattern?

The Decorator Pattern allows adding **new functionality to an object dynamically** without modifying its existing class.

Simple Explanation

- Add features without changing original code
- Objects are wrapped with additional behavior

Real-Life Example

Coffee Shop System - Base coffee - Add milk, sugar, chocolate as extra features - No need to create separate classes for every combination

Key Components

- **Component Interface**: Defines basic behavior
- **Concrete Component**: Base object
- **Decorator**: Wraps the base object and adds features

Simple Code Structure (Java)

```java
interface Coffee {
    int cost();
}

class SimpleCoffee implements Coffee {
    public int cost() {
        return 50;
    }
}
```

Why Use Decorator Pattern?

- Add responsibilities dynamically
- Avoid subclass explosion
- Follow Open–Closed Principle

Exam One-Liner

Decorator Pattern adds new behavior to objects dynamically without modifying their class.

Quick Comparison Table

Pattern	Category	Main Purpose
Observer	Behavioral	Automatic notification
Strategy	Behavioral	Change behavior at runtime
Decorator	Structural	Add functionality dynamically

Conclusion

These three design patterns help in building **flexible, reusable, and maintainable software systems**. They are widely used in real-world applications and are important for exams, presentations, and software design understanding.

```
interface Observer {
    void update(int value);
}
interface Subject {
    void attach(Observer o);
    void notifyObservers();
}
class Data implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private int value;
    public void attach(Observer o) {
        observers.add(o);
    }
    public void setValue(int value) {
        this.value = value;
        notifyObservers();
    }
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(value);
        }
    }
}
class Display implements Observer {
    public void update(int value) {
        System.out.println("Updated value: " + value);
    }
}
Data data = new Data();
Display d1 = new Display();
data.attach(d1);
data.setValue(10);
```