Decorator Design Pattern

The Decorator Pattern is a structural design pattern that allows behavior to be added to individual objects dynamically and transparently. It adheres to the Single Responsibility Principle by dividing functionality among classes with unique areas of concern.

Customer Requirement:

Imagine you are tasked with building a coffee shop application. The system needs to allow customers to order coffee with various optional add-ons like milk, sugar, and whipped cream. Each add-on has an additional cost, and customers can choose any combination of these add-ons. The system should be flexible enough to add new add-ons in the future without altering the core code for coffee.

Solution: Decorator Pattern

The Decorator Pattern is ideal for this scenario. It allows you to add responsibilities to objects at runtime. In this case, you can create decorators for each add-on (e.g., milk, sugar) and apply them dynamically to a coffee object.

Components of the Decorator Pattern:

1. Component Interface (Coffee): Defines the interface for objects that can have responsibilities added to them dynamically.

2. Concrete Component (SimpleCoffee): The core object to which additional responsibilities can be attached.
3. Decorator (CoffeeDecorator): Maintains a reference to a Component object and defines an interface that conforms to Component.

4. Concrete Decorators (MilkDecorator, SugarDecorator): Extend the functionality of the component by adding responsibilities.

Example: Coffee Shop System

1. Component Interface (Coffee):

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

2. Concrete Component (SimpleCoffee):

```
public class SimpleCoffee implements Coffee {
    @Override
    public String getDescription() {
        return 'Simple Coffee';
    }

    @Override
    public double getCost() {
        return 5.0;
    }
}
```

3. Decorator (CoffeeDecorator):

```
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }
    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }
}
```

4. Concrete Decorators (MilkDecorator, SugarDecorator):

```
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ', Milk';
    }

    @Override
    public double getCost() {
        return super.getCost() + 1.5;
    }
}

public class SugarDecorator extends CoffeeDecorator {

    public SugarDecorator(Coffee coffee) {

        super(coffee);

    }



    @Override

    public String getDescription() {

        return super.getDescription() + ', Sugar';

    }



    @Override

    public double getCost() {

        return super.getCost() + 0.5;

    }

}
```

5. Client Code:

```
public class DecoratorPatternDemo {

    public static void main(String[] args) {

        Coffee coffee = new SimpleCoffee();

        System.out.println(coffee.getDescription() + ' $' + coffee.getCost());



        coffee = new MilkDecorator(coffee);

        System.out.println(coffee.getDescription() + ' $' + coffee.getCost());



        coffee = new SugarDecorator(coffee);
        System.out.println(coffee.getDescription() + ' $' + coffee.getCost());
    }
}
```

How It Works:

- Component Interface (Coffee): The Coffee interface defines the operations that can be dynamically extended by decorators.

- Concrete Component (SimpleCoffee): This is the basic implementation of Coffee, which can have responsibilities added to it.

- Decorator (CoffeeDecorator): This abstract class implements the Coffee interface and holds a reference to a Coffee object. It serves as a base for concrete decorators.

- Concrete Decorators (MilkDecorator, SugarDecorator): These classes extend CoffeeDecorator and add specific functionalities like adding milk or sugar.

Why It Is a Structural Design Pattern:

The Decorator Pattern is a structural pattern because it deals with how objects are composed and how additional responsibilities can be dynamically assigned to them. It allows for extending the functionality of objects without modifying their underlying structure, making the system more flexible and easier to maintain.

Benefits of the Decorator Pattern:

1. Open/Closed Principle: The Decorator Pattern adheres to the Open/Closed Principle by allowing classes to be open for extension but closed for modification. You can add new functionality without altering existing code.

2. Flexibility: The pattern provides a flexible alternative to subclassing for extending functionality. You can mix and

match decorators to achieve different combinations of behavior.

3. Single Responsibility Principle: Decorators can be used to divide a complex task into simpler components, each with a single responsibility.

When to Use the Decorator Pattern:

- When you need to add responsibilities to individual objects dynamically and transparently, without affecting other objects.

- When extending a class is impractical, and you want to keep the class closed for modification.

- When you want to use composition rather than inheritance to extend functionality.