Search articles...

Decorator Design Pattern

Topic Tags:

System Design LLD

Problem Statement: Extending Functionality Without Modifying the Core Code ✨

Imagine you're designing a coffee shop ordering system. The system needs to manage various coffee orders and their customizations. Customers can start with a basic coffee (e.g., Espresso, Cappuccino) and then add multiple customizations like milk, sugar, cream, or flavors (e.g., vanilla, hazelnut).

The Problem: Each coffee type and customization combination would require a new class if we follow a traditional inheritance-based approach. For example, you'd need separate classes for "EspressoWithMilk", "CappuccinoWithVanilla", or "LatteWithMilkAndSugar". This quickly becomes unmanageable as the number of combinations grows.

The Challenge: How can you dynamically add new functionalities (customizations) to objects without altering their code or creating a complex class hierarchy?

Solving It the Traditional Way: A Messy Solution 🛠️

Here's how you might approach the problem in a straightforward but inflexible

Java

```java
import java.util.Scanner;
public class CoffeeShop {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter your coffee order:");
    String coffeeOrder = scanner.nextLine();
    if (coffeeOrder.equalsIgnoreCase("Espresso with Milk and Sugar")) {
      System.out.println("Preparing Espresso with Milk and Sugar...");
    } else if (coffeeOrder.equalsIgnoreCase("Cappuccino with Vanilla")) {
      System.out.println("Preparing Cappuccino with Vanilla...");
    } else {
      System.out.println("Order not recognized!");
    }
    scanner.close();
  }
}
```

The Challenge: Avoiding an Explosion of Classes 🤔

An interviewer might ask:
• What if the coffee shop introduces a new base coffee type? Do you have to rewrite or create new combinations for every customization?
• How can you ensure that the system is flexible enough to handle any number or type of customizations, applied in any order?
• How can you make the code reusable and maintainable without duplicating logic?

Ugly Code: Why This Approach Fails 🤦‍♂️

While this code works for a few orders, it quickly becomes unmanageable as the number of coffee types and customizations grows:

1. Rigid Structure: Adding a new customization or coffee type requires modifying existing logic.
2. Code Duplication: Similar logic is repeated for different combinations.

3. Lack of Flexibility: The order of customizations and new features are difficult to handle dynamically.

Java

```java
import java.util.Scanner;
public class CoffeeShop {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter your coffee order:");
    String coffeeOrder = scanner.nextLine();
    if (coffeeOrder.equalsIgnoreCase("Espresso with Milk and Sugar")) {
      System.out.println("Preparing Espresso with Milk and Sugar...");
    } else if (coffeeOrder.equalsIgnoreCase("Cappuccino with Vanilla")) {
      System.out.println("Preparing Cappuccino with Vanilla...");
    } else if (coffeeOrder.equalsIgnoreCase("Latte with Caramel")) {
      System.out.println("Preparing Latte with Caramel...");
    } else if (coffeeOrder.equalsIgnoreCase("Mocha with Whipped Cream")) {
      System.out.println("Preparing Mocha with Whipped Cream...");
    } else if (coffeeOrder.equalsIgnoreCase("Black Coffee with Honey")) {
      System.out.println("Preparing Black Coffee with Honey...");
    } else {
      System.out.println("Order not recognized!");
    }
    scanner.close();
  }
}
```

To solve these issues, we need a way to dynamically wrap additional functionality around existing objects.

The Savior: Decorator Design Pattern 🦜

The Decorator Pattern is designed to address this problem by dynamically adding new functionalities to objects without modifying their code. It allows you to wrap objects in layers of functionality, creating flexible and extensible systems.

How the Decorator Pattern Works 🔧

The Decorator Pattern achieves this by:

1. Defining a common interface for the base object and its decorators.
2. Using decorators to wrap base objects, adding new behaviors while preserving the original object's interface.
3. Allowing multiple decorators to be stacked dynamically.

Solving the Problem with Decorator Design Pattern 🌿

Step 1: Define a Common Interface :

The first step is to define a common interface for all coffee types and customizations.

Coffee.java

Java

```java
// Coffee.java - Common interface for all coffee types
public interface Coffee {
    String getDescription();
    double getCost();
}
```

Step 2: Create Concrete Classes for Base Coffee Types :

These classes implement the Coffee interface and represent the core objects.

Espresso.java

Java

```java
public class Espresso implements Coffee {
    @Override
    public String getDescription() {
        return "Espresso";
    }
    @Override
    public double getCost() {
        return 2.00;
    }
}
```

Cappuccino.java

Java

```java
public class Cappuccino implements Coffee {
    @Override
    public String getDescription() {
        return "Cappuccino";
    }
    @Override
    public double getCost() {
        return 3.00;
    }
}
```

Step 3: Create Abstract Decorator Class :

The abstract decorator implements the Coffee interface and wraps a Coffee object. This allows additional functionality to be dynamically added to the Coffee object without modifying its structure.

CoffeeDecorator.java

Java

```java
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;
    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }
    @Override
    public String getDescription() {
        return coffee.getDescription();
    }
    @Override
    public double getCost() {
        return coffee.getCost();
    }
}
```

Step 4: Create Concrete Decorators for Customizations :

Each decorator adds a specific functionality (customization) to the base coffee object.

MilkDecorator.java

Java

```java
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }
    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Milk";
    }
    @Override
    public double getCost() {
        return coffee.getCost() + 0.50;
    }
}
```

SugarDecorator.java

Java

```java
public class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }
    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Sugar";
    }
    @Override
    public double getCost() {
        return coffee.getCost() + 0.25;
    }
}
```

VanillaDecorator.java

Java

```java
public class VanillaDecorator extends CoffeeDecorator {
    public VanillaDecorator(Coffee coffee) {
        super(coffee);
    }
    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Vanilla";
    }
    @Override
    public double getCost() {
        return coffee.getCost() + 0.75;
    }
}
```

Step 5: Use the Decorators in the Client :

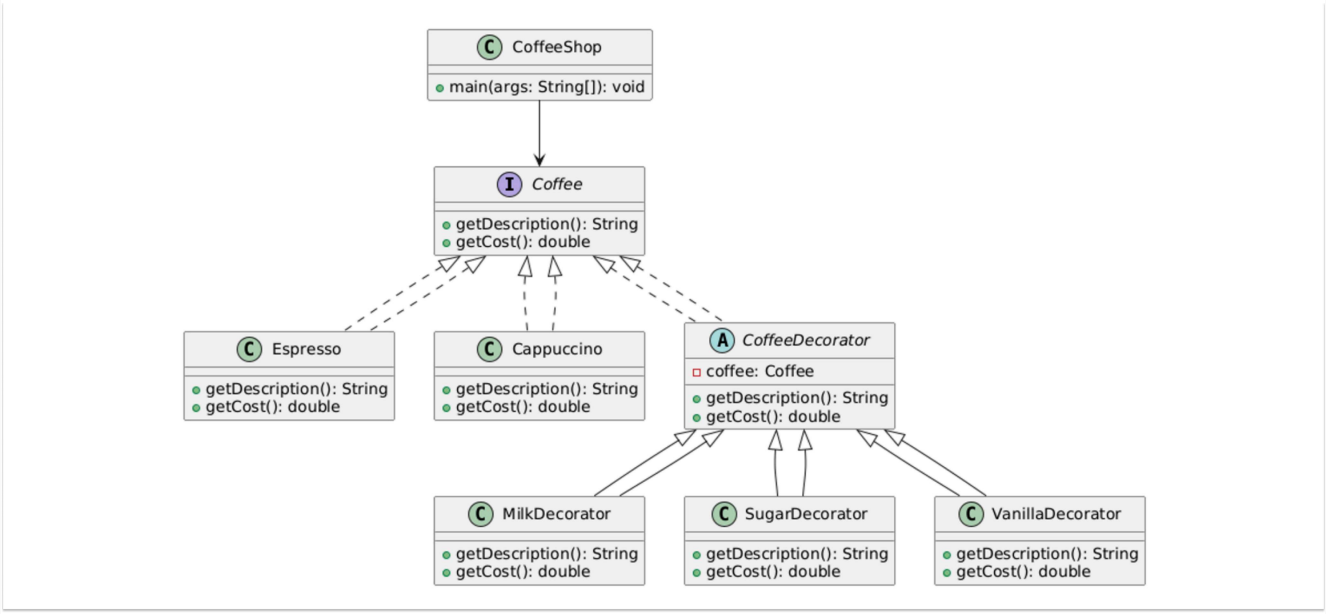
The client dynamically wraps the base coffee objects with the desired customizations. This approach allows for flexible and dynamic addition of features to the coffee objects without altering their structure.

CoffeeShop.java

Java

```java
public class CoffeeShop {
    public static void main(String[] args) {
        Coffee coffee = new Espresso();
        coffee = new MilkDecorator(coffee);
        coffee = new SugarDecorator(coffee);
        System.out.println("Order: " + coffee.getDescription());
        System.out.println("Total Cost: $" + coffee.getCost());
        Coffee anotherCoffee = new Cappuccino();
        anotherCoffee = new VanillaDecorator(anotherCoffee);
        System.out.println("nOrder: " + anotherCoffee.getDescription());
        System.out.println("Total Cost: $" + anotherCoffee.getCost());
```

```
12        }
13    }
```



Advantages of Using the Decorator Design Pattern 🏆

1. Extensibility
The pattern allows you to add new functionalities (customizations) without modifying existing classes.

2. Flexibility
Customizations can be applied dynamically, in any order, to any object.

3. Reusability
Decorators are reusable across different base objects and can be combined in different ways.

4. Open/Closed Principle
The pattern adheres to the Open/Closed Principle by allowing the system to be extended without modifying existing code.

Real-life Use Cases of the Decorator Pattern 🌍

1. Coffee Shop Systems
As shown in this example, the pattern is used to manage dynamic customizations of coffee orders.

2. GUI Frameworks
Decorators are used to add functionalities like borders, shadows, or scrollbars to graphical components.

3. Logging Frameworks
The pattern is used to dynamically add logging, authentication, or security layers to a system.

4. File I/O Streams
Java's I/O streams use decorators to add functionalities like buffering, compression, or encryption to file streams.

Conclusion 🎯

The Decorator Design Pattern provides a powerful way to extend functionality without modifying existing code. In our coffee shop example, it allows dynamic, flexible customization of coffee orders while keeping the code clean and maintainable.
By wrapping objects with layers of functionality, the Decorator Pattern ensures scalability, reusability, and adherence to solid design principles. It's an essential tool for building systems that need dynamic feature composition. 🌟