Eduardo Pons

Coffee Decorator Pattern Implementation

Overview

This document provides a detailed overview of the implementation and benefits of the Decorator pattern applied to the `Coffee` class. The Decorator pattern enhances the functionality of objects dynamically at runtime, offering a flexible alternative to subclassing for adding features.

Problem Statement

In a coffee shop scenario, customers may want to customize their coffee with various add-ons, such as milk or caramel. If we were to use traditional inheritance to represent each possible combination of add-ons, the number of subclasses would grow exponentially. This would lead to a complex and unmanageable class hierarchy.

Solution

The Decorator pattern provides a scalable and flexible solution to this problem. By using decorators, we can dynamically add functionality to `Coffee` objects, creating a customizable coffee experience without the need for numerous subclasses. This approach allows for combining different add-ons in any order and quantity.

Code Implementation

Coffee Interface

The `Coffee` interface defines the essential operations for any coffee object, including getting a description and calculating the cost.

java
Copiar código

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

SimpleCoffee Class

The `SimpleCoffee` class implements the `Coffee` interface and represents a basic coffee without any add-ons.

java
Copiar código

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

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

CoffeeDecorator Abstract Class

The `CoffeeDecorator` abstract class implements the `Coffee` interface and contains a reference to a `Coffee` object. It forwards the calls to the wrapped coffee object, allowing decorators to add or modify behavior.

java
Copiar código

```
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();
    }
}
```

MilkDecorator Class

The `MilkDecorator` class extends `CoffeeDecorator` and adds milk to the coffee, modifying the description and cost accordingly.

java
Copiar código

```
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() + 1.50;
    }
}
```

CaramelDecorator Class

The `CaramelDecorator` class extends `CoffeeDecorator` and adds caramel to the coffee, altering the description and cost.

java
Copiar código

```
public class CaramelDecorator extends CoffeeDecorator {

    public CaramelDecorator(Coffee coffee){
        super(coffee);
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Caramel";
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 2.5;
    }
}
```

Test Cases

Simple Coffee

Verifies that a `SimpleCoffee` object has the correct description and cost.

java
Copiar código

```
@Test
public void testSimpleCoffee() {
    Coffee coffee = new SimpleCoffee();
    assertEquals("Simple Coffee", coffee.getDescription());
    assertEquals(5.00, coffee.getCost());
}
```

Coffee with Milk

Checks that adding milk to `SimpleCoffee` updates the description and cost correctly.

java
Copiar código

```
@Test
public void testCoffeeWithMilk() {
    Coffee coffee = new MilkDecorator(new SimpleCoffee());
    assertEquals("Simple Coffee, Milk", coffee.getDescription());
    assertEquals(6.50, coffee.getCost());
}
```

Coffee with Milk and Caramel

Tests the combination of milk and caramel on `SimpleCoffee`, ensuring the correct description and cost.

java
Copiar código

```
@Test
public void testCoffeeWithMilkAndCaramel() {
       Coffee coffee = new CaramelDecorator(new MilkDecorator(new
SimpleCoffee()));
                assertEquals("Simple Coffee, Milk, Caramel",
coffee.getDescription());
    assertEquals(9.00, coffee.getCost());
}
```

Multiple Caramels

Verifies that multiple caramel decorators correctly accumulate their effects on the coffee's description and cost.

java
Copiar código

```
@Test
public void testMultipleCaramel() {
      Coffee coffee = new CaramelDecorator(new CaramelDecorator(new
SimpleCoffee()));
              assertEquals("Simple Coffee, Caramel, Caramel",
coffee.getDescription());
    assertEquals(10.00, coffee.getCost());
}
```

Benefits of Using the Decorator Pattern

1. **Flexible Object Customization**: Allows for the dynamic addition of features to objects without altering their structure, providing a flexible way to customize coffee.
2. **Scalability**: Supports adding new decorators or combinations of decorators without modifying existing code, facilitating easy expansion.
3. **Reduced Subclassing**: Avoids a proliferation of subclasses by combining behaviors dynamically, leading to a cleaner and more manageable class hierarchy.
4. **Enhanced Maintainability**: Separates the responsibilities of adding features from the core `Coffee` implementation, making the code easier to maintain and extend.
5. **Dynamic Behavior**: Enables runtime composition of behaviors, offering greater flexibility in how objects are configured and used.

Conclusion

The Decorator pattern offers a robust solution for managing the customization of coffee orders. By allowing for dynamic composition of features, it simplifies the addition of new behaviors and configurations, resulting in a more maintainable and extensible codebase.