Coffee Shop Application: Implementing the Decorator Design Pattern

Objective

This project aims to familiarize students with the Decorator design pattern, demonstrating its capability to dynamically enhance object behavior without modifying its core structure. Through this project, we've developed a coffee ordering system that allows users to customize their beverage with various condiments.

Introduction to the Decorator Design Pattern

The Decorator pattern is a structural design pattern used to add new responsibilities to an object dynamically. Instead of creating numerous subclasses to extend functionality, decorators provide a flexible alternative.

In our scenario, consider a basic coffee order. Customers might want to add different condiments such as milk, sugar, or vanilla. Instead of creating a separate class for every possible combination (which is impractical), we use decorators to dynamically "decorate" or "wrap" the coffee object with additional features.

Implementation

1. The Coffee Base Component:

This represents our basic idea of what any coffee should entail: a description and a cost.

```
9 usages   5 inheritors
public abstract class Coffee {
    2 usages
    String description = "Unknown Coffee";

    4 usages   4 overrides
    public String getDescription() {
        return description;
    }

    5 usages   4 implementations
    public abstract double cost();
}
```

2. Concrete Coffee:

```
public class BasicCoffee extends Coffee {

    1 usage
    public BasicCoffee() {
        description = "Basic Coffee";
    }

    5 usages
    public double cost() {
        return 2.00;
    }
}
```

This is the plain coffee, representing the base product.

3. Abstract Decorators:

Our rulebook of potential add-ons. These "decorators" will enhance the base coffee.

```
3 usages   3 inheritors
public abstract class CoffeeDecorator extends Coffee {
    4 usages   3 implementations
    public abstract String getDescription();
}
```

4. Concrete Decorators:

Specific items derived from the rulebook, each one knows how to add its distinct touch to the coffee.

```
1 usage
public class MilkDecorator extends CoffeeDecorator {
    3 usages
    Coffee coffee;

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

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

    5 usages
    public double cost() {
        return 0.50 + coffee.cost();
    }
}
```

5. User Interaction - CoffeeShop Application:

Here, the user can order their desired coffee, adding condiments of their choice.

```
while(continueShopping) {
    Coffee order = new BasicCoffee();
    boolean continueOrdering = true;

    System.out.println("\nStarting with a basic coffee for $" + order.cost());

    while(continueOrdering) {
```

Advantages of Using the Decorator Pattern in our Project

Flexibility: Instead of having a rigid class structure, decorators allow us to add functionalities on-the-fly.

Scalability: Introducing a new condiment (like caramel or whipped cream) is straightforward without needing to redesign the existing system.

Maintainability: Each decorator class has a single responsibility, adhering to the Single Responsibility Principle. This makes the system easier to understand and modify.

Conclusion

The coffee shop application effectively leverages the Decorator design pattern to provide a customizable coffee ordering experience. Through decorators, we can dynamically add as many condiments as desired without altering the core coffee object, showcasing the pattern's power and flexibility.