CCSW 413 Lab 6: Java Decorator Design Patterns Lab

March 23, 2024

1 Introduction

1.1 Overview

In this lab manual, we will explore the Decorator design pattern, a structural pattern widely used in software development. The Decorator pattern allows behavior to be added to individual objects dynamically, at runtime. Through hands-on exercises and practical examples in Java, students will gain a solid understanding of how to implement and apply the Decorator pattern effectively.

1.2 Objective

The objective of this lab manual is to introduce students to the Decorator design pattern and provide them with hands-on experience in implementing and applying it in software development projects. By the end of this lab, students should be able to understand the concepts and principles of the Decorator design pattern, identify scenarios where the Decorator pattern can be applied effectively, design and implement flexible and extensible software systems using the Decorator pattern, and evaluate the benefits and trade-offs of using the Decorator pattern in different contexts.

2 Lab Session 1: Decorator Design Pattern

Definition

The Decorator design pattern is a structural pattern that allows behaviour to be added to individual objects, dynamically, without affecting the behaviour of other objects from the same class. It is useful when you want to add responsibilities to objects without subclassing.

Benefits

The key benefits of using the Decorator pattern include:

- Enhanced flexibility and extensibility: Decorators can be added and removed dynamically, allowing for flexible combinations of behaviours.
- Open-closed principle: The pattern allows for extending functionality without modifying existing code, adhering to the open-closed principle.
- Separation of concerns: Responsibilities are divided between different decorators, promoting better code organization and maintainability.

When to Use the Decorator Pattern

The Decorator pattern is commonly used in the following scenarios:

- When you need to add or remove responsibilities to objects dynamically at runtime.
- When subclassing to extend behaviour is impractical or leads to a combinatorial explosion of classes.
- When you want to avoid altering existing code or breaking the open-closed principle.

- When you need to add functionality to individual objects without affecting other instances of the same class.

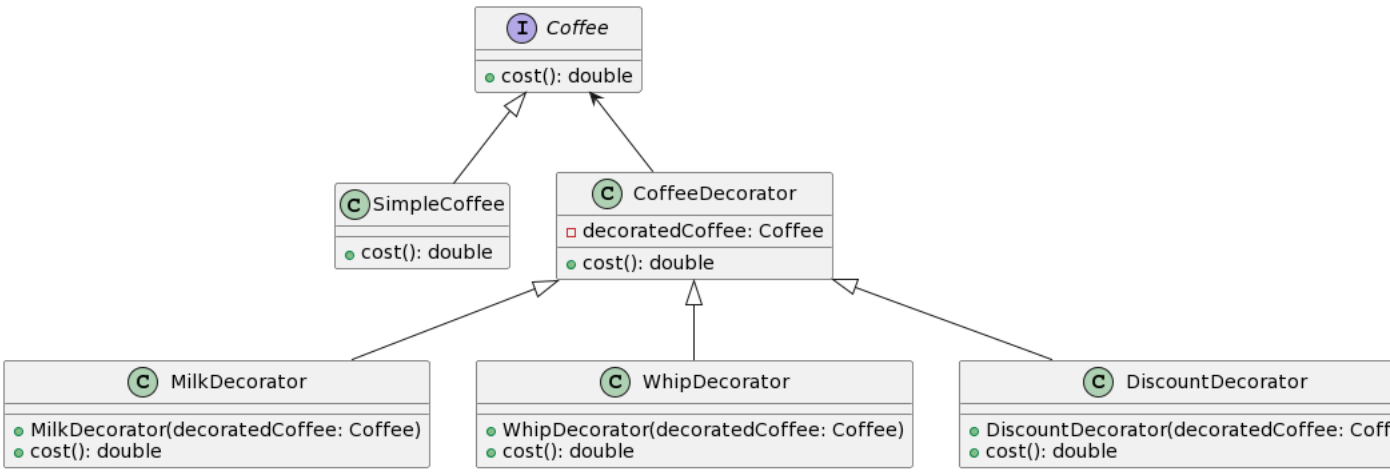

Figure 1: Class Diagrams of the coffee ordering system using the Decorator design pattern

Problem Scenario Example 1

Problem Description

Imagine you are developing a coffee ordering system for a cafe. The system allows customers to customize their coffee orders with various options such as size, milk type, and toppings. Each customization option adds to the final price of the coffee. However, the cafe also offers seasonal promotions where certain customizations are free of charge or discounted. You need to design a system that can handle these dynamic pricing changes without modifying the core coffee order class.

Java Code Example

```java
// Component interface
interface Coffee {
    double cost();
}

// Concrete component
class SimpleCoffee implements Coffee {
    @Override
    public double cost() {
        return 2.0;
    }
}

// Decorator
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee decoratedCoffee) {
        this.decoratedCoffee = decoratedCoffee;
    }

    public double cost() {
        return decoratedCoffee.cost();
    }
}

// Concrete decorators
```

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

    @Override
    public double cost() {
        return super.cost() + 0.5; // Milk costs $0.5 extra
    }
}

class WhipDecorator extends CoffeeDecorator {
    public WhipDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public double cost() {
        return super.cost() + 0.7; // Whip costs $0.7 extra
    }
}

class DiscountDecorator extends CoffeeDecorator {
    public DiscountDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public double cost() {
        // Apply 20% discount
        return super.cost() * 0.8;
    }
}
```

Listing 1: Java code implementation of the coffee ordering system using the Decorator design pattern

Guided Code Walkthrough

This code implements the coffee ordering system using the Decorator pattern. The `Coffee` interface defines the component, and `SimpleCoffee` is a concrete component. The `CoffeeDecorator` is an abstract decorator class, and `MilkDecorator`, `WhipDecorator`, and `DiscountDecorator` are concrete decorators. The class diagram in Figure 1 illustrates the relationships between these classes.

In Lab Task 1

Extend the Java code implementation of the coffee ordering system to include the following:

1. Implement the base classes for different types of coffee drinks, such as Espresso, Latte, and Cappuccino.
2. Create decorator classes for various additional ingredients, such as MilkDecorator, SugarDecorator, and WhippedCreamDecorator.
3. Implement the cost calculation logic in the decorator classes to calculate the total cost of each coffee drink with added ingredients.

Lab Assignment

Assignment 1: Real-World Scenario (Adapter Pattern)

Suppose you are developing a pizza ordering system. The system should allow customers to customize their pizzas with various toppings such as cheese, pepperoni, mushrooms, etc. However, the available

toppings may vary, and new toppings could be added in the future. You need to design a flexible solution to handle this scenario.

Requirements:

1. Customers should be able to choose the type of crust for their pizza (e.g., thin crust, thick crust, stuffed crust).
2. Customers should be able to select the size of the pizza (e.g., small, medium, large).
3. Customers should be able to add toppings to their pizza, with the option to select multiple toppings.
4. The system should calculate the total price of the pizza based on the selected options.
5. Customers should be able to add special instructions or comments for each order.

Assignment Deliverables

Deliverables:

1. You are required to implement the pizza ordering system. Write Java code to create the necessary classes and interfaces to demonstrate the decorator pattern. Ensure that your implementation follows the guidelines provided during the lab session and adheres to object-oriented design principles.
2. Create a class diagram to illustrate the structure of your solution.