Author: Pedro Jahir Hinojosa García

Instructor: Miguel Rugerio

Activity: Observer method

Date: 31/08/2024

The decorator pattern allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class. To practice this pattern, I created a Cheeseburger as the main product, which can be customized with various toppings such as lettuce, pickles, extra meat, and extra cheese. Each topping is implemented as a separate decorator class that extends the functionality of the Cheeseburger object dynamically. For example, as shown in the provided Image 1, Order 1 adds lettuce, pickles, extra meat, and extra cheese to the cheeseburger, resulting in a total cost with discounts applied if necessary. Each decorator modifies the behavior or properties of the Cheeseburger without altering the original class, demonstrating the flexibility and modularity of the decorator pattern.

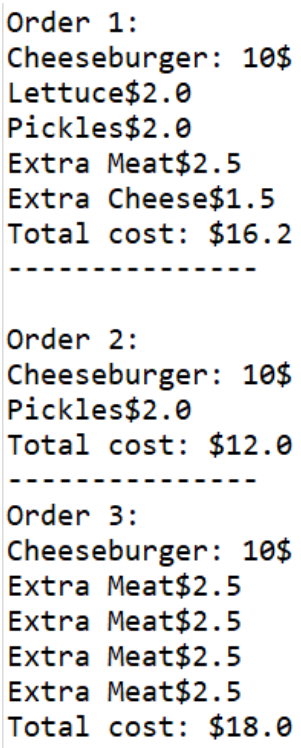


Image 1: Output example.

The implementation of the decorator pattern consists of creating a base component interface that defines the methods for objects, as seen in Code 1. The Component interface is the base type for both the main object and its decorators. Next, an abstract decorator class implements the Component interface and contains a reference to a Component object, allowing dynamic extension of functionalities without modifying the original object, as shown in Code 2. Concrete decorators, such as Lettuce, Pickles, ExtraMeat, and ExtraCheese, inherit from the abstract decorator class and override its methods to add their specific behavior or functionality, as demonstrated in Code 3. Finally, these decorators can be combined or nested, allowing the main object, like a Cheeseburger, to be extended with additional behaviors or properties, such as adding toppings, while still maintaining the original object structure and functionality, as illustrated in the example in Code 4.

```java
package DecoratorWithTest;

public interface Component {
    String getNameAndPrice();
    double getTotalCost();

}
```

Code 1: Component Interface.

```java
package DecoratorWithTest;

public abstract class ToppingDecorator implements Component{

    private Component component;
    String toppingName;
    double toppingPrice;

    public ToppingDecorator(Component component) {
        this.component=component;
    }

    @Override
    public String getNameAndPrice() {
    return component.getNameAndPrice()+"\n"+ toppingName+"$"+toppingPrice;
    }

    public double getTotalCost() {
        return component.getTotalCost() + toppingPrice;
```

Code 2: Abstract Decorator Class.

```java
package DecoratorWithTest;

public class Lettuce extends ToppingDecorator {

    public Lettuce(Component component) {
        super(component);
        toppingName = "Lettuce";
        toppingPrice = 2.00;
    }
}
```

Code 3: Example of a decorator.

```java
package DecoratorWithTest;

public class Principal {
    public static void main(String[] args) {

        // Creating a cheeseburger with all toppings
        Component burger1 = new ExtraCheese(new ExtraMeat(new Pickles(new Lettuce(new Cheeseburger()))));
        Component burger2 = new Pickles(new Cheeseburger());

        Component burger3 = new ExtraMeat(new ExtraMeat(new ExtraMeat(new ExtraMeat(new Cheeseburger()))));

        // Print and calculate the total cost for each order
        System.out.println("Order 1:");
        printCostWithDiscount(burger1);
        System.out.println("---------------");

        System.out.println("\nOrder 2:");
        printCostWithDiscount(burger2);
        System.out.println("---------------");
        System.out.println("Order 3:");
        printCostWithDiscount(burger3);


    }
```

Code 4: Main Object and Decorators in Action

Test Part:

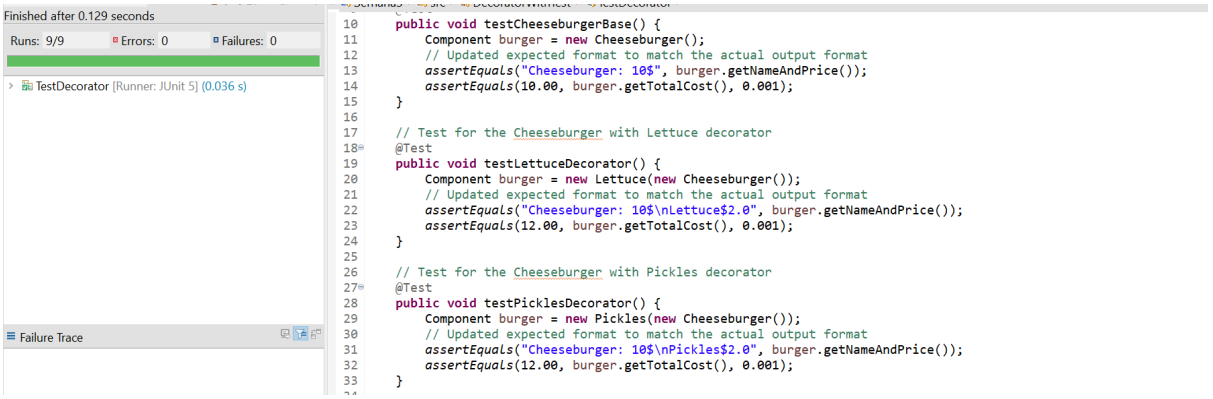
The test is made focused on checking if the prices are set correctly, implementing a new method to adjust the price based on the total amount, as shown in Code 5.

Some cases were added and all the tests were completed because the code was built in order to accomplish every test, Code 6. The test are focused on prove if the setted price is correct and the discount is applied.

```java
    public static void printCostWithDiscount(Component burger) {
        System.out.println(burger.getNameAndPrice());
        double totalCost = calculateTotalCostWithDiscount(burger.getTotalCost());
        System.out.println("Total cost: $" + totalCost);
    }

    // Public method to calculate total cost with discount
    public static double calculateTotalCostWithDiscount(double totalCost) {
        if (totalCost > 16) {
            totalCost *= 0.90;
        }
        return totalCost;
    }
}
```

Code 5: Methods to assign the price and discount.



Code 6: Test cases completed.