Karen Giselle Valdez Muñoz

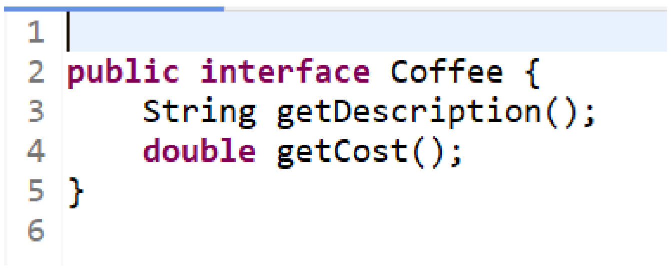
Decorator Pattern

Introduction

The Decorator pattern is a structural design pattern that allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class. It provides an alternative to subclassing for extending functionality. In this document, we will explore how the Decorator pattern is implemented in Java through a coffee shop example, where different decorators (add-ons) are used to enhance the basic coffee.

The Base Component

The Coffee interface serves as the base component that defines the methods any coffee object should implement. In our case, we have two methods: getDescription() and getCost().

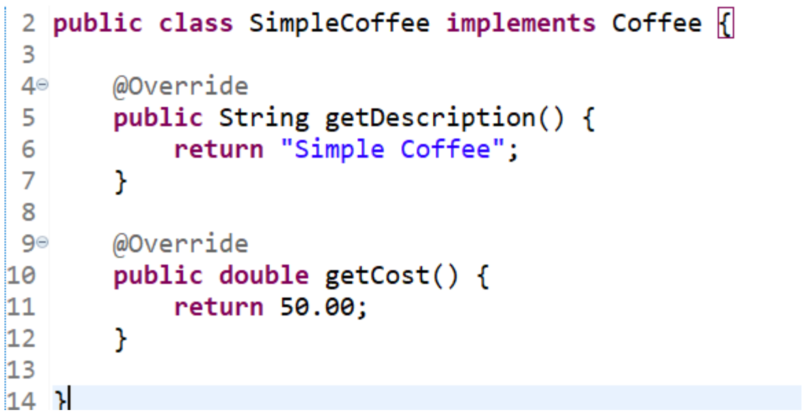


```java

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

```

- getDescription(): This method returns a description of the coffee, including any additional ingredients.
- getCost(): This method returns the cost of the coffee, taking into account any additional charges from the decorators.

The Concrete Component

The SimpleCoffee class is a concrete implementation of the Coffee interface. It represents the most basic version of a coffee, with no additional ingredients or decorations.

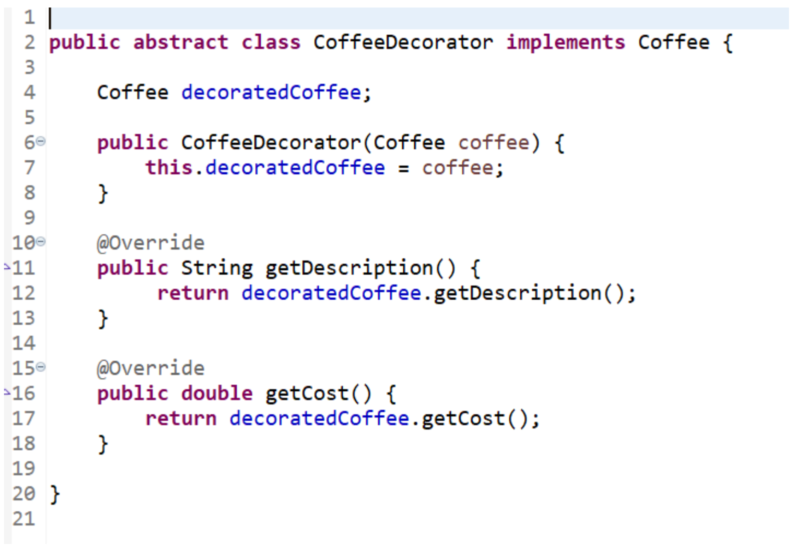


```java
public class SimpleCoffee implements Coffee {

    @Override
    public String getDescription() {
        return "Simple Coffee";
    }

    @Override
    public double getCost() {
        return 50.00;
    }

}
```

- SimpleCoffee: This class implements the Coffee interface and provides the basic functionality for a simple coffee with a description and cost of $50.00.

The Decorator Abstract Class

The CoffeeDecorator class is an abstract class that implements the Coffee interface. It is used as a base class for all decorators, ensuring that they comply with the Coffee interface and can be used interchangeably with any Coffee object.

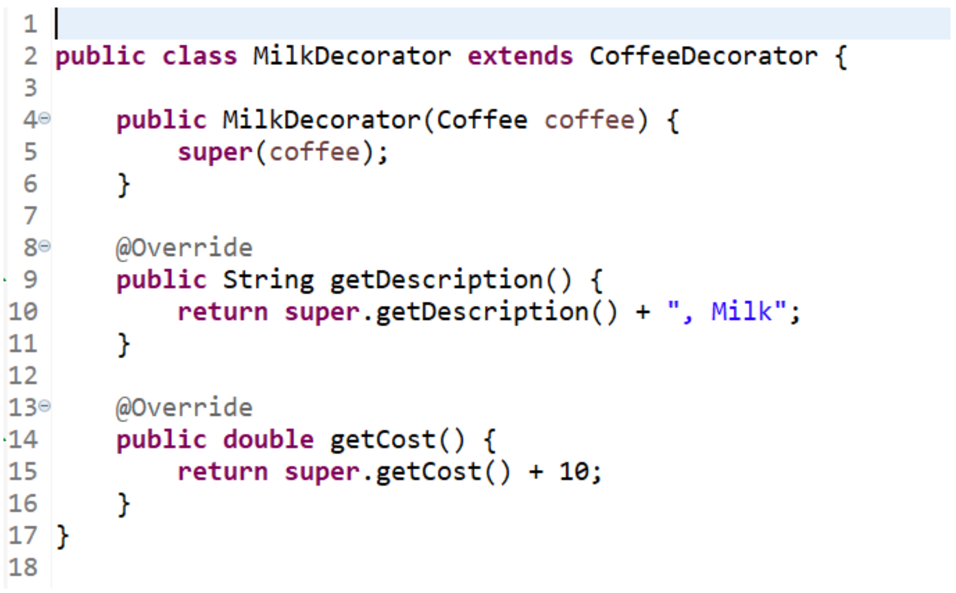


```java

public abstract class CoffeeDecorator implements Coffee {

    Coffee decoratedCoffee;

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

    @Override
    public String getDescription() {
         return decoratedCoffee.getDescription();
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }

}

```

- decoratedCoffee: This protected member holds a reference to the Coffee object that is being decorated.
- Constructor: The constructor takes a Coffee object and assigns it to decoratedCoffee, which allows the decorator to call the wrapped object's methods.
- getDescription() and getCost(): These methods call the corresponding methods on the decorated coffee object, essentially passing the request along to the wrapped object.

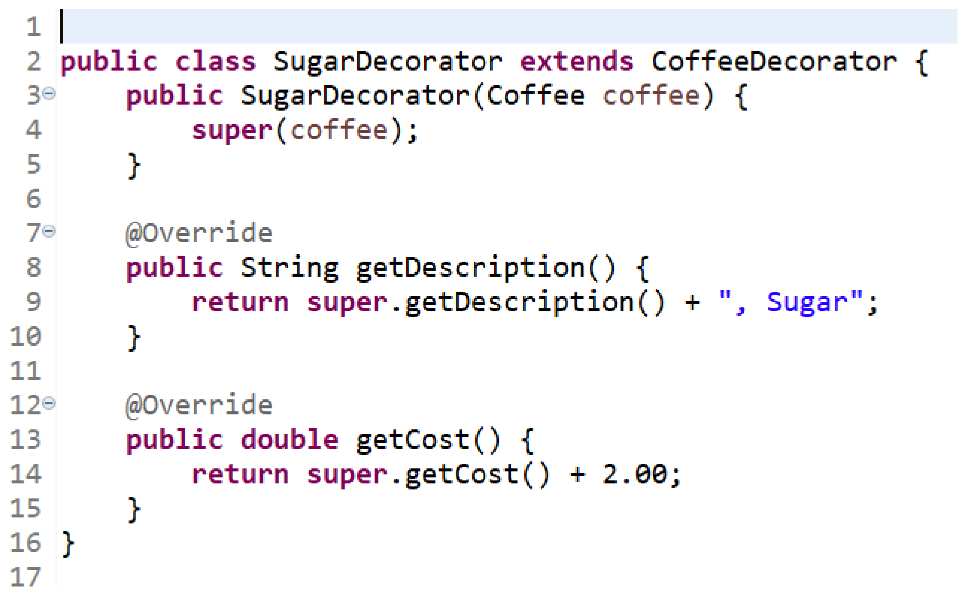
The Concrete Decorators

Concrete decorators extend the CoffeeDecorator class and add specific behavior to the coffee object. In this example, we have three decorators: MilkDecorator, SugarDecorator, and SyrupDecorator.



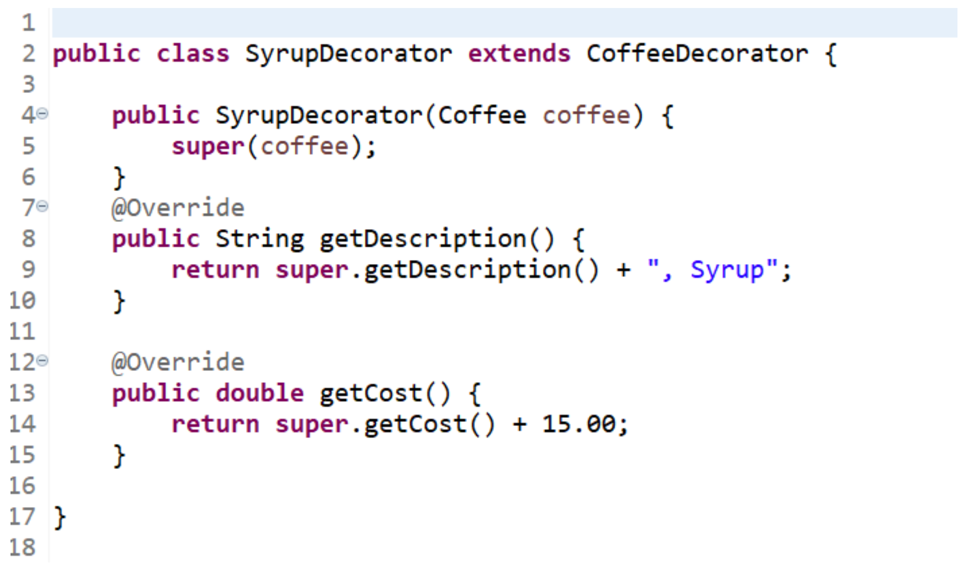
```java

public class MilkDecorator extends CoffeeDecorator {

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

    @Override
    public String getDescription() {
        return super.getDescription() + ", Milk";
    }

    @Override
    public double getCost() {
        return super.getCost() + 10;
    }
}

```

- getDescription(): Appends ", Milk" to the description of the decorated coffee.
- getCost(): Adds $10 to the cost of the decorated coffee.



```java

public class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", Sugar";
    }

    @Override
    public double getCost() {
        return super.getCost() + 2.00;
    }
}

```

- getDescription(): Appends ", Sugar" to the description of the decorated coffee.
- getCost(): Adds $2.00 to the cost of the decorated coffee.



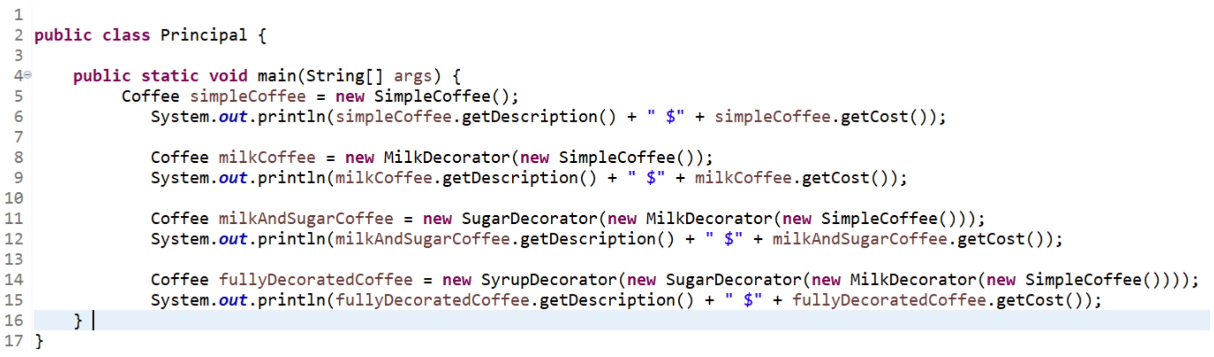
```java

public class SyrupDecorator extends CoffeeDecorator {

    public SyrupDecorator(Coffee coffee) {
        super(coffee);
    }
    @Override
    public String getDescription() {
        return super.getDescription() + ", Syrup";
    }

    @Override
    public double getCost() {
        return super.getCost() + 15.00;
    }

}

```

- getDescription(): Appends ", Syrup" to the description of the decorated coffee.
- getCost(): Adds $15.00 to the cost of the decorated coffee.

Karen Giselle Valdez Muñoz

Main Method

The main method is where we bring everything together to demonstrate how the Decorator pattern works in a practical scenario. It simulates a coffee shop where different coffee orders are created using various combinations of decorators.



```java

public class Principal {

    public static void main(String[] args) {
         Coffee simpleCoffee = new SimpleCoffee();
            System.out.println(simpleCoffee.getDescription() + " $" + simpleCoffee.getCost());

            Coffee milkCoffee = new MilkDecorator(new SimpleCoffee());
            System.out.println(milkCoffee.getDescription() + " $" + milkCoffee.getCost());

            Coffee milkAndSugarCoffee = new SugarDecorator(new MilkDecorator(new SimpleCoffee()));
            System.out.println(milkAndSugarCoffee.getDescription() + " $" + milkAndSugarCoffee.getCost());

            Coffee fullyDecoratedCoffee = new SyrupDecorator(new SugarDecorator(new MilkDecorator(new SimpleCoffee())));
            System.out.println(fullyDecoratedCoffee.getDescription() + " $" + fullyDecoratedCoffee.getCost());
    }
}
```

- SimpleCoffee:
    - Code: Coffee simpleCoffee = new SimpleCoffee();
    - Explanation: Creates a basic coffee with no additional ingredients. When getDescription() and getCost() are called, they return "Simple Coffee" and $50.00, respectively.

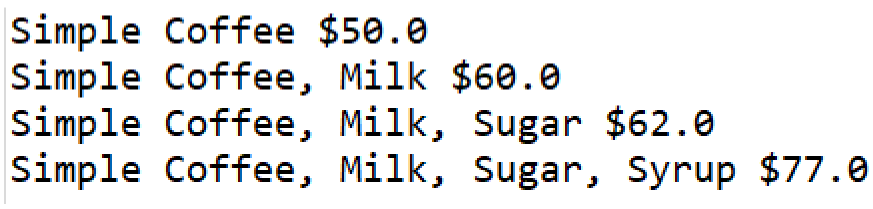
- MilkDecorator Applied:
    - Code: Coffee milkCoffee = new MilkDecorator(new SimpleCoffee());
    - Explanation: Wraps the SimpleCoffee with a MilkDecorator, adding milk to the coffee. The cost increases by $10.00, and the description updates to "Simple Coffee, Milk".

- MilkDecorator and SugarDecorator Applied:
    - Code: Coffee milkAndSugarCoffee = new SugarDecorator(new MilkDecorator(new SimpleCoffee()));
    - Explanation: This creates a coffee with both milk and sugar by wrapping the SimpleCoffee first with a MilkDecorator and then with a SugarDecorator. The description becomes "Simple Coffee, Milk, Sugar" and the cost rises to $62.00.

- All Decorators Applied:
    - Code: Coffee fullyDecoratedCoffee = new SyrupDecorator(new SugarDecorator(new MilkDecorator(new SimpleCoffee())));
    - Explanation: This example fully decorates the coffee with milk, sugar, and syrup. The decorators are applied in sequence, and the description reflects all the added ingredients. The cost increases to $77.00.

Karen Giselle Valdez Muñoz

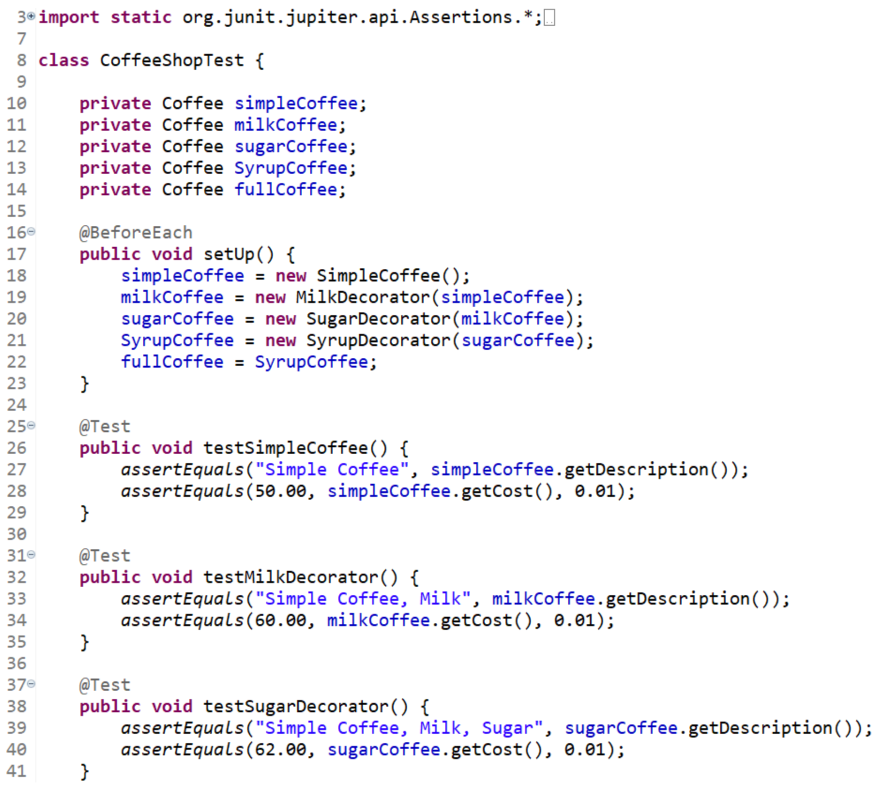
Output:



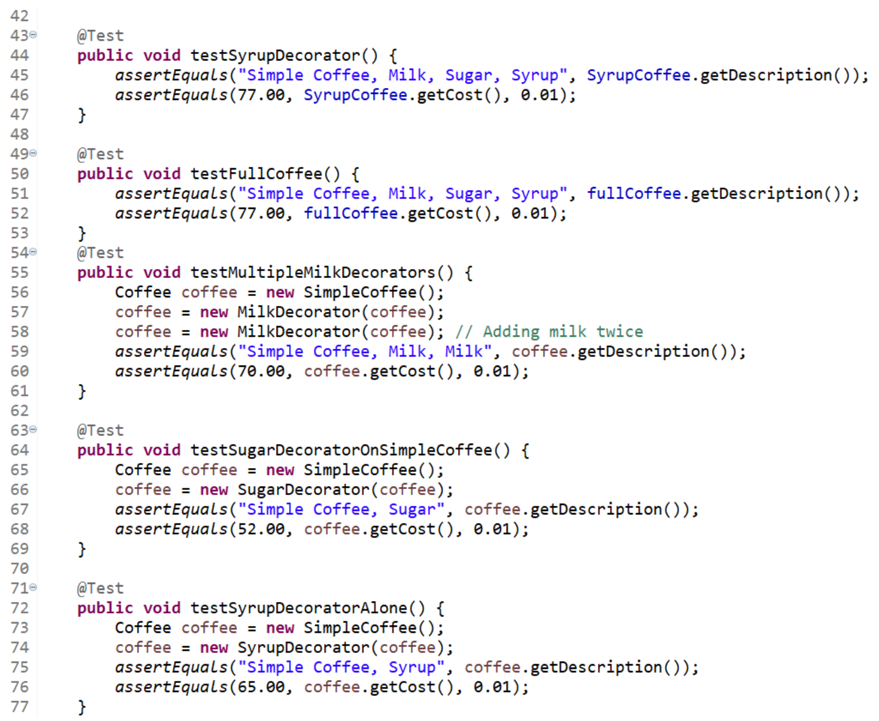
```
Simple Coffee $50.0
Simple Coffee, Milk $60.0
Simple Coffee, Milk, Sugar $62.0
Simple Coffee, Milk, Sugar, Syrup $77.0
```

Unit Testing with JUnit

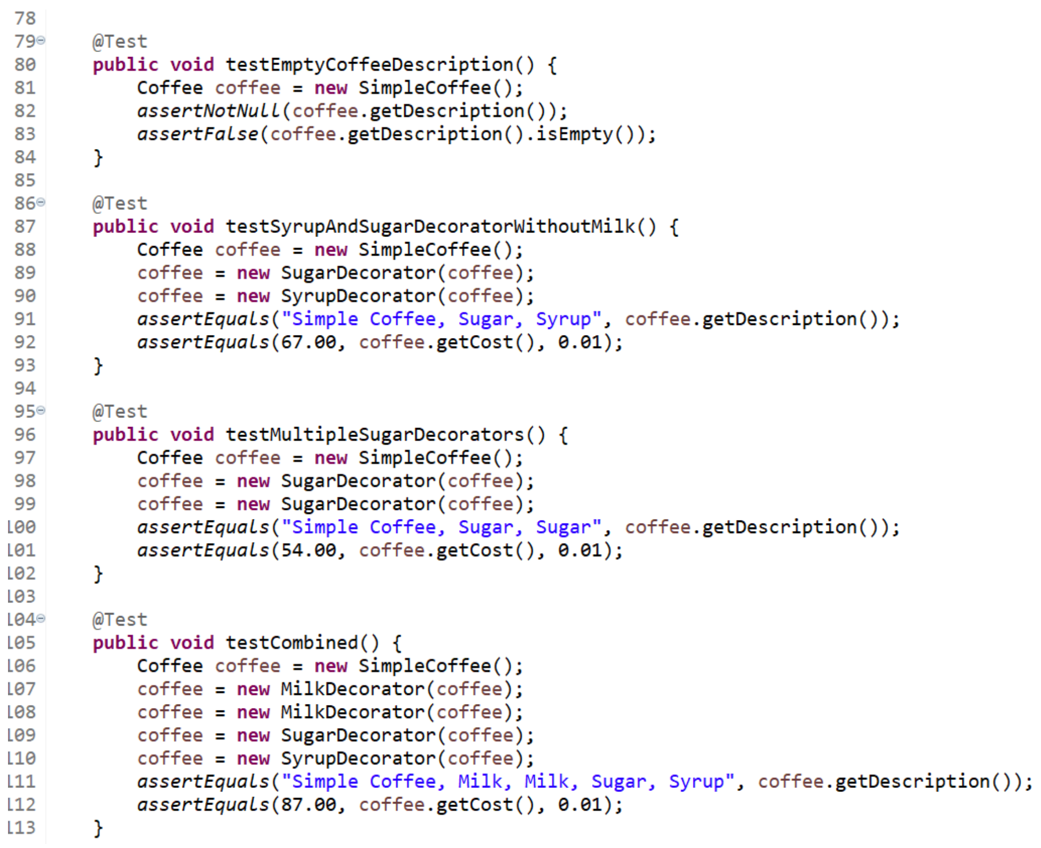
Unit tests are crucial to ensure that the decorators function correctly. Here is a summary of the unit tests that have been created to cover the code.



```java
import static org.junit.jupiter.api.Assertions.*;

class CoffeeShopTest {

    private Coffee simpleCoffee;
    private Coffee milkCoffee;
    private Coffee sugarCoffee;
    private Coffee SyrupCoffee;
    private Coffee fullCoffee;

    @BeforeEach
    public void setUp() {
        simpleCoffee = new SimpleCoffee();
        milkCoffee = new MilkDecorator(simpleCoffee);
        sugarCoffee = new SugarDecorator(milkCoffee);
        SyrupCoffee = new SyrupDecorator(sugarCoffee);
        fullCoffee = SyrupCoffee;
    }

    @Test
    public void testSimpleCoffee() {
        assertEquals("Simple Coffee", simpleCoffee.getDescription());
        assertEquals(50.00, simpleCoffee.getCost(), 0.01);
    }

    @Test
    public void testMilkDecorator() {
        assertEquals("Simple Coffee, Milk", milkCoffee.getDescription());
        assertEquals(60.00, milkCoffee.getCost(), 0.01);
    }

    @Test
    public void testSugarDecorator() {
        assertEquals("Simple Coffee, Milk, Sugar", sugarCoffee.getDescription());
        assertEquals(62.00, sugarCoffee.getCost(), 0.01);
    }
```



```java

    @Test
    public void testSyrupDecorator() {
        assertEquals("Simple Coffee, Milk, Sugar, Syrup", SyrupCoffee.getDescription());
        assertEquals(77.00, SyrupCoffee.getCost(), 0.01);
    }

    @Test
    public void testFullCoffee() {
        assertEquals("Simple Coffee, Milk, Sugar, Syrup", fullCoffee.getDescription());
        assertEquals(77.00, fullCoffee.getCost(), 0.01);
    }
    @Test
    public void testMultipleMilkDecorators() {
        Coffee coffee = new SimpleCoffee();
        coffee = new MilkDecorator(coffee);
        coffee = new MilkDecorator(coffee); // Adding milk twice
        assertEquals("Simple Coffee, Milk, Milk", coffee.getDescription());
        assertEquals(70.00, coffee.getCost(), 0.01);
    }

    @Test
    public void testSugarDecoratorOnSimpleCoffee() {
        Coffee coffee = new SimpleCoffee();
        coffee = new SugarDecorator(coffee);
        assertEquals("Simple Coffee, Sugar", coffee.getDescription());
        assertEquals(52.00, coffee.getCost(), 0.01);
    }

    @Test
    public void testSyrupDecoratorAlone() {
        Coffee coffee = new SimpleCoffee();
        coffee = new SyrupDecorator(coffee);
        assertEquals("Simple Coffee, Syrup", coffee.getDescription());
        assertEquals(65.00, coffee.getCost(), 0.01);
    }
```



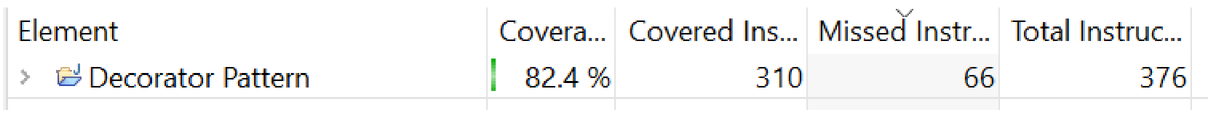
```java
    @Test
    public void testEmptyCoffeeDescription() {
        Coffee coffee = new SimpleCoffee();
        assertNotNull(coffee.getDescription());
        assertFalse(coffee.getDescription().isEmpty());
    }

    @Test
    public void testSyrupAndSugarDecoratorWithoutMilk() {
        Coffee coffee = new SimpleCoffee();
        coffee = new SugarDecorator(coffee);
        coffee = new SyrupDecorator(coffee);
        assertEquals("Simple Coffee, Sugar, Syrup", coffee.getDescription());
        assertEquals(67.00, coffee.getCost(), 0.01);
    }

    @Test
    public void testMultipleSugarDecorators() {
        Coffee coffee = new SimpleCoffee();
        coffee = new SugarDecorator(coffee);
        coffee = new SugarDecorator(coffee);
        assertEquals("Simple Coffee, Sugar, Sugar", coffee.getDescription());
        assertEquals(54.00, coffee.getCost(), 0.01);
    }

    @Test
    public void testCombined() {
        Coffee coffee = new SimpleCoffee();
        coffee = new MilkDecorator(coffee);
        coffee = new MilkDecorator(coffee);
        coffee = new SugarDecorator(coffee);
        coffee = new SyrupDecorator(coffee);
        assertEquals("Simple Coffee, Milk, Milk, Sugar, Syrup", coffee.getDescription());
        assertEquals(87.00, coffee.getCost(), 0.01);
    }
```

Explanation of the Tests

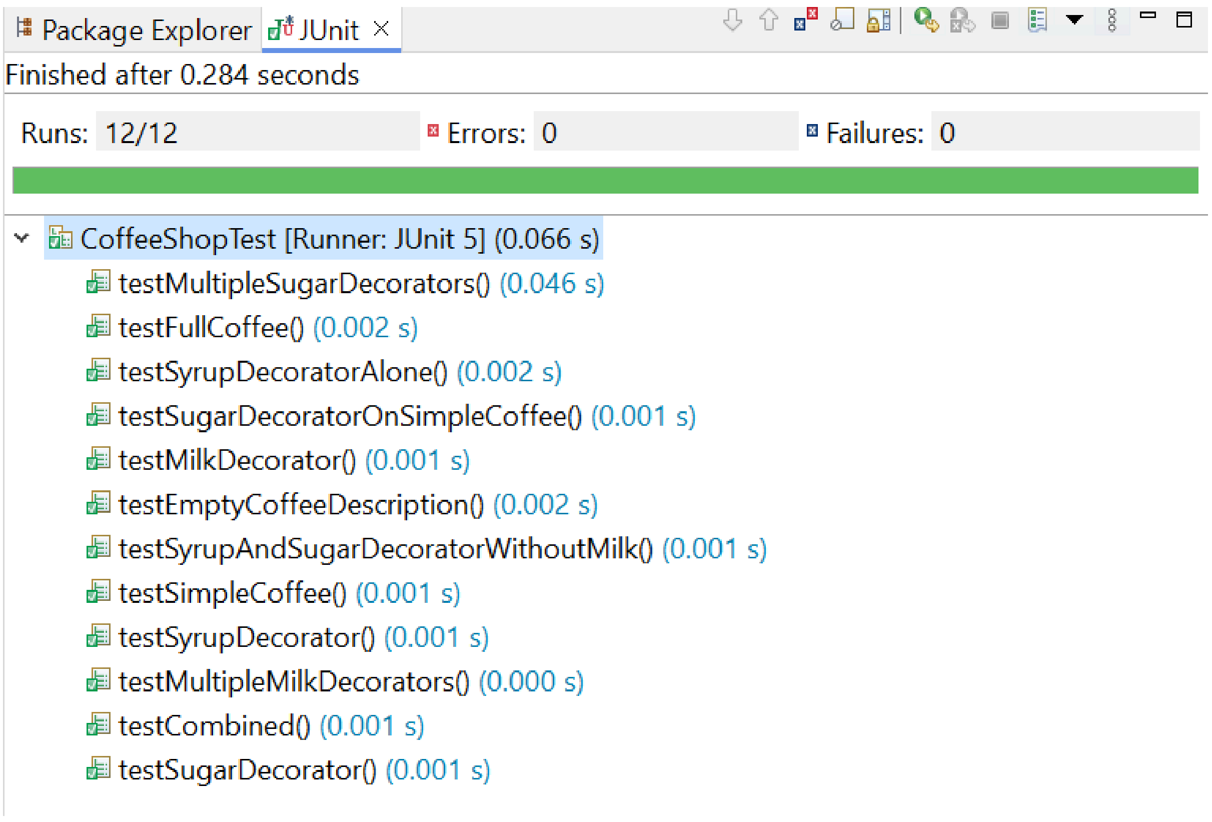
- testSimpleCoffee(): Verifies that a SimpleCoffee has the correct description and cost.
- testMilkDecorator(): Tests that adding milk to the coffee modifies the description and increases the cost correctly.
- testSugarDecorator(): Ensures that adding sugar to the coffee correctly updates the description and cost.
- testSyrupDecorator(): Checks that adding syrup modifies the description and cost as expected.
- testFullCoffee(): Tests the complete combination of decorators, verifying the final description and cost.
- testMultipleMilkDecorators(): Verifies that adding milk twice results in the correct description and cost.
- testSugarDecoratorOnSimpleCoffee(): Checks that adding sugar to a SimpleCoffee updates the description and cost correctly.
- testSyrupDecoratorAlone(): Ensures that applying a SyrupDecorator alone results in the correct description and cost.
- testEmptyCoffeeDescription(): Validates that the description of a SimpleCoffee is neither null nor empty.
- testSyrupAndSugarDecoratorWithoutMilk(): Confirms that adding both syrup and sugar to a SimpleCoffee gives the correct description and cost.
- testMultipleSugarDecorators(): Ensures that applying sugar multiple times updates the description and cost accurately.

- testCombined(): Verifies that combining multiple decorators results in the correct description and cost.

Results of the test:



Element	Covera...	Covered Ins...	Missed Instr...	Total Instruc...
Decorator Pattern	82.4 %	310	66	376



Package Explorer | JUnit

```
Finished after 0.284 seconds

Runs: 12/12        Errors: 0        Failures: 0

CoffeeShopTest [Runner: JUnit 5] (0.066 s)
    testMultipleSugarDecorators() (0.046 s)
    testFullCoffee() (0.002 s)
    testSyrupDecoratorAlone() (0.002 s)
    testSugarDecoratorOnSimpleCoffee() (0.001 s)
    testMilkDecorator() (0.001 s)
    testEmptyCoffeeDescription() (0.002 s)
    testSyrupAndSugarDecoratorWithoutMilk() (0.001 s)
    testSimpleCoffee() (0.001 s)
    testSyrupDecorator() (0.001 s)
    testMultipleMilkDecorators() (0.000 s)
    testCombined() (0.001 s)
    testSugarDecorator() (0.001 s)
```

Conclusion

The Decorator pattern is an effective way to dynamically add responsibilities to objects without modifying their code. By using the pattern, we can create flexible and extensible designs that support various combinations of functionalities. The example of the coffee shop with multiple decorators demonstrates how this pattern can be applied in real-world scenarios.