**1. Introduction**

The Decorator design 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 is used to extend the functionality of objects in a flexible and reusable way. In this report, I will explain the Decorator pattern, provide code screenshots, and share insights from implementing the pattern in a simple coffee ordering system.

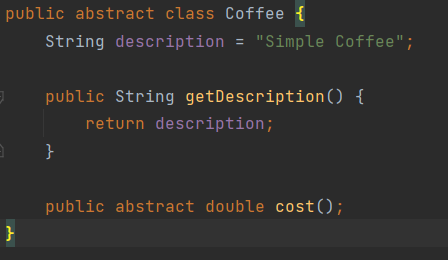
**2. Decorator Design Pattern**

The Decorator pattern is used when you want to add responsibilities to objects without subclassing. It allows you to add or alter the behavior of individual objects without affecting the behavior of other objects of the same class.

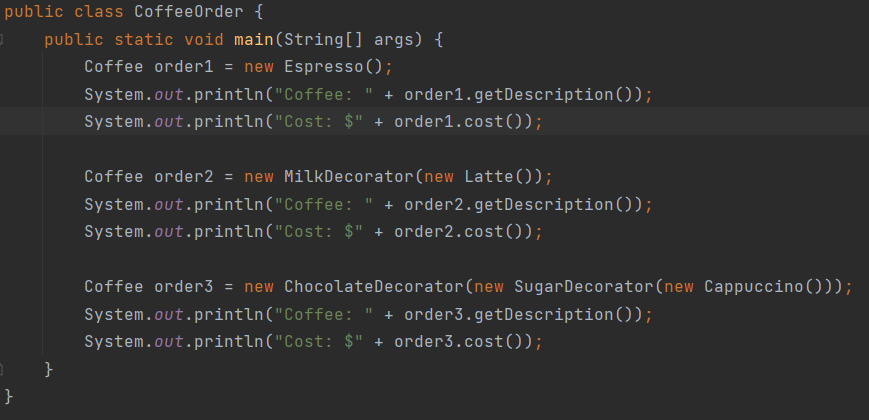
**Use Case of Decorator Pattern:**

In our implementation, we created a coffee ordering system. The base class, **Coffee**, represents a simple coffee order. Concrete coffee classes (**Espresso**, **Latte**, **Cappuccino**) extend the **Coffee** class. Decorator classes (**MilkDecorator**, **SugarDecorator**, **ChocolateDecorator**) are used to add additional condiments to coffee orders without modifying the base coffee classes. This allows for dynamic customization of coffee orders with different combinations of condiments.

**3. Code Screenshots and Results**

**Base Coffee Class and Concrete Coffee Classes:** 

**Decorator Classes (MilkDecorator):** 

**Main Class and Coffee Orders:** 

**Results:**

* **Order 1: Espresso**
* Coffee: Espresso
* Cost: $2.0
* **Order 2: Latte with Milk**
* Coffee: Latte, Milk
* Cost: $5.0
* **Order 3: Cappuccino with Sugar and Chocolate**
* Coffee: Cappuccino, Sugar, Chocolate
* Cost: $6.0

**4. Additional Insights**

* The Decorator pattern is a powerful way to add functionality to objects at runtime. It promotes code reusability and flexibility by allowing for a combination of decorators to customize objects.
* It's essential to design your base component class and decorators carefully to ensure they follow a consistent interface. This makes it easier to combine them effectively.
* In the context of the coffee ordering system, it becomes apparent how easy it is to create various combinations of coffee orders with different condiments using the Decorator pattern. This demonstrates the pattern's effectiveness in extending object behavior without altering the base class.
* While we used a console-based demonstration in this report, you can further extend the implementation by creating a graphical user interface (GUI) for the coffee ordering system, making it more user-friendly.

**5. Conclusion**

The Decorator pattern is a valuable design pattern for dynamically extending the functionality of objects. In our coffee ordering system example, it allowed us to create customized coffee orders with various combinations of condiments, enhancing user experience. This pattern provides a clean and flexible way to modify object behavior while adhering to the open-closed principle of object-oriented design.