**Decorator Design Pattern**

**Assignment - 1**

Name - Heet Dobariya Roll No. - 22BCP177 Group - G5

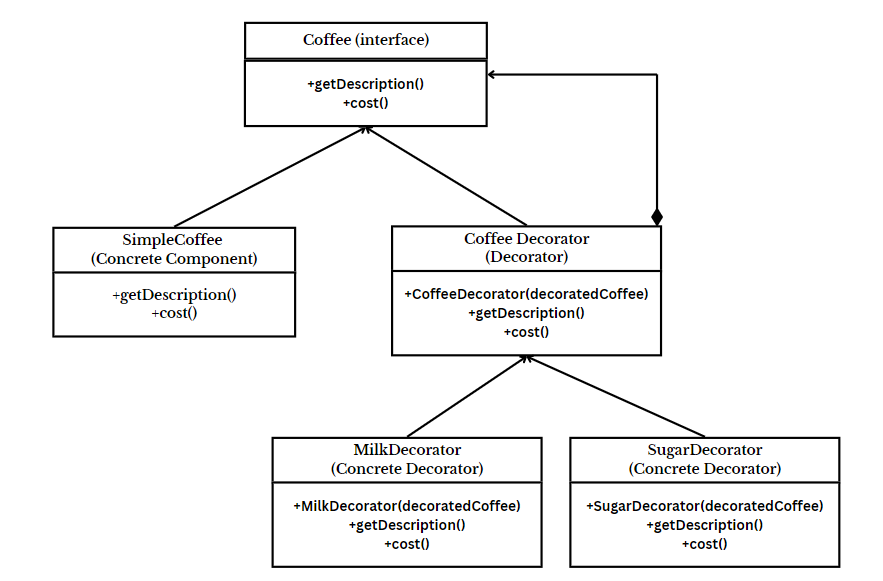
* **Decorator Design Pattern :**

The Decorator Design Pattern is a structural design pattern that allows behavior to be added to individual objects dynamically, without affecting the behavior of other objects from the same class. It's often used when you need to add new functionality to objects at runtime, or when it's impractical to extend them using subclassing.

Here's how the Decorator Pattern works:

1. Component Interface: Define an interface or abstract class for the objects that can have responsibilities added dynamically. This interface usually contains common methods that all components must implement.
2. Concrete Component: Implement the Component interface with a concrete class. This class represents the base object to which additional functionality can be added.
3. Decorator: Create an abstract class (or interface) representing the decorator. This class has a reference to a Component object and implements the Component interface itself. It acts as a base class for all concrete decorators.
4. Concrete Decorators: Implement concrete decorator classes by extending the Decorator class. Each concrete decorator adds its own behavior or responsibilities to the component by overriding methods of the Component interface and calling the methods of the wrapped component.

* **Program :** Implement decorator design pattern for coffee example.
* **UML Diagram :**



* **Code :**

interface Coffee

{

String getDescription();

double cost();

}

class SimpleCoffee implements Coffee

{

@Override

public String getDescription()

{

return "Simple Coffee";

}

@Override

public double cost()

{

return 30.0;

}

}

abstract class CoffeeDecorator implements Coffee

{

protected Coffee decoratedCoffee;

public CoffeeDecorator(Coffee decoratedCoffee)

{

this.decoratedCoffee = decoratedCoffee;

}

@Override

public String getDescription()

{

return decoratedCoffee.getDescription();

}

@Override

public double cost()

{

return decoratedCoffee.cost();

}

}

class MilkDecorator extends CoffeeDecorator

{

public MilkDecorator(Coffee decoratedCoffee)

{

super(decoratedCoffee);

}

@Override

public String getDescription()

{

return super.getDescription() + ", with Milk";

}

@Override

public double cost()

{

return super.cost() + 5.5;

}

}

class SugarDecorator extends CoffeeDecorator

{

public SugarDecorator(Coffee decoratedCoffee)

{

super(decoratedCoffee);

}

@Override

public String getDescription()

{

return super.getDescription() + ", with Sugar";

}

@Override

public double cost()

{

return super.cost() + 3.5;

}

}

public class DecoratorPatternExample

{

public static void main(String[] args)

{

// Create a simple coffee

Coffee coffee = new SimpleCoffee();

System.out.println("Cost: " + coffee.cost() + ", Description: " + coffee.getDescription());

// Decorate the simple coffee with milk

Coffee milkCoffee = new MilkDecorator(coffee);

System.out.println("Cost: " + milkCoffee.cost() + ", Description: " + milkCoffee.getDescription());

// Decorate the simple coffee with sugar

Coffee sweetCoffee = new SugarDecorator(milkCoffee);

System.out.println("Cost: " + sweetCoffee.cost() + ", Description: " + sweetCoffee.getDescription());

}

}

* **Output :**

