Documentation: Coffee Ordering System using Decorator Pattern in Java

1. Introduction

This documentation describes a coffee ordering system implemented using the **Decorator Pattern** in Java. The system allows for dynamic customization of beverages by adding condiments (e.g., Mocha, Whip, Soy, Milk) without modifying existing code, promoting flexibility and maintainability.

2. Design Pattern Used: Decorator Pattern

The **Decorator Pattern** is a structural design pattern that allows behavior to be added to individual objects dynamically. This is achieved by wrapping objects inside decorator classes that augment their behavior.

Benefits of Using the Decorator Pattern

- Enables runtime addition of functionality.
- Promotes the Open-Closed Principle (OCP), allowing extensions without modifying existing code.
- Prevents an explosion of subclasses compared to inheritance-based designs.

3. Class Descriptions

Abstract Class: Beverage

```
public abstract class Beverage {
   String description = "Unknown Beverage";
   public String getDescription() {
      return description;
   }
   public abstract double cost();
}
```

- Serves as the base class for all beverage types.
- Defines a `description` attribute.
- Declares an abstract `cost()` method that must be implemented by subclasses.

Concrete Beverage Classes

These are specific beverages extending `Beverage`.

HouseBlend.java

```
public class HouseBlend extends Beverage {
   public HouseBlend() {
```

```
        description = "HOUSE-BLEND COFFEE";
    }
    public double cost() {
        return .50;
    }
}
```

Espresso.java

```
public class Espresso extends Beverage {
    public Espresso() {
        description = "ESPRESSO";
    }
    public double cost() {
        return 2.25;
    }
}
```

Decaf.java

```
public class Decaf extends Beverage {
    public Decaf() {
        description = "DECAF COFFEE";
    }
    public double cost() {
        return 1.25;
    }
}
```

DarkRoast.java

```
public class DarkRoast extends Beverage {
    public DarkRoast() {
        description = "ROASTED DARK COFFEE";
    }
    public double cost() {
        return 1.25;
    }
}
```

4. Condiment Decorators

Abstract Decorator Class: CondimentDecorator

```
public abstract class CondimentDecorator extends Beverage {
    protected Beverage beverage;
    public abstract String getDescription();
}
```

- Extends Beverage, ensuring compatibility with all beverages.
- Holds a reference to the Beverage object it decorates.

- Declares an abstract method getDescription() to override descriptions.

Concrete Condiment Decorators

Each of these classes extends CondimentDecorator, wrapping a Beverage and modifying its behavior.

Mocha.java

```
public class Mocha extends CondimentDecorator {
   public Mocha(Beverage beverage) {
      this.beverage = beverage;
   }
   public String getDescription() {
      return beverage.getDescription() + ", +MOCHA";
   }
   public double cost() {
      return .50 + beverage.cost();
   }
}
```

Whip.java

```
public class Whip extends CondimentDecorator {
   public Whip(Beverage beverage) {
      this.beverage = beverage;
   }
   public String getDescription() {
      return beverage.getDescription() + ", +WHIP";
   }
   public double cost() {
      return .20 + beverage.cost();
   }
}
```

Soy.java

```
public class Soy extends CondimentDecorator {
   public Soy(Beverage beverage) {
      this.beverage = beverage;
   }
   public String getDescription() {
      return beverage.getDescription() + ", +SOY";
   }
   public double cost() {
      return .25 + beverage.cost();
   }
}
```

Milk.java

```
public class Milk extends CondimentDecorator {
   public Milk(Beverage beverage) {
```

```
        this.beverage = beverage;
    }
    public String getDescription() {
        return beverage.getDescription() + ", +MILK";
    }
    public double cost() {
        return .20 + beverage.cost();
    }
}
```

5. Main Class: StarbuzzCoffee

```
public class StarbuzzCoffee {
    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription() + " £" + beverage.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription() + " £" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription() + " £" + beverage3.cost());
    }
}
```

6. Execution and Expected Output

Example Output

```
ESPRESSO £2.25
ROASTED DARK COFFEE, +MOCHA, +MOCHA, +WHIP £2.45
HOUSE-BLEND COFFEE, +SOY, +MOCHA, +WHIP £1.45
```

7. Conclusion

This implementation effectively utilizes the **Decorator Pattern** to allow for dynamic customization of coffee orders. The code is modular, extendable, and avoids subclass explosion by using decorators instead of hard-coded inheritance structures.

This approach makes it easy to introduce new beverage types or condiments without modifying existing code, adhering to best software design practices.