**Decorator Pattern: Coffee Shop**

**Description:**

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 a flexible alternative to subclassing for extending functionality.

**Components:**

1. **Component Interface (Coffee):**  
   Defines the interface for objects that can have responsibilities added dynamically. In this case, it defines methods to get the description and cost of coffee.
2. **Concrete Component (SimpleCoffee):**  
   Implements the component interface. Represents a basic coffee with a description and cost.
3. **Decorator (CoffeeDecorator):**  
   Abstract class that implements the component interface and contains a reference to a Coffee object. It delegates the operations to the referenced object and can add additional responsibilities.
4. **Concrete Decorators (MilkDecorator, SugarDecorator):**  
   Extend the decorator class to add specific responsibilities. Each decorator modifies the description and cost of the coffee by adding its own behavior.

**How It Works:**

1. **Component Interface:** Defines the methods for getting the description and cost of the coffee.
2. **Concrete Component:** Implements the basic coffee functionality.
3. **Decorator Class:** Contains a reference to a Coffee object and delegates calls to it. Concrete decorators add their own behavior before or after delegating calls.
4. **Concrete Decorators:** Implement additional features (e.g., milk, sugar) by modifying the description and cost of the coffee.

**Real-Life Example: Coffee Shop**

In a coffee shop scenario:

* **Coffee (Component Interface):** Defines methods to get the description and cost of the coffee.
* **SimpleCoffee (Concrete Component):** Represents a basic coffee with a fixed description ("Simple Coffee") and cost ($5.00).
* **CoffeeDecorator (Decorator):** Abstract class that implements the Coffee interface and holds a reference to a Coffee object. It delegates method calls to the wrapped Coffee instance.
* **MilkDecorator and SugarDecorator (Concrete Decorators):** Add specific features to the coffee. MilkDecorator adds "Milk" to the description and increases the cost by $1.50. SugarDecorator adds "Sugar" to the description and increases the cost by $0.50.

**Scenario:**

1. **Basic Coffee:** The client creates a SimpleCoffee object and gets its description and cost. Output: "Simple Coffee -> Cost: $5.00".
2. **Adding Milk:** The client wraps the SimpleCoffee with a MilkDecorator, modifying the description to "Simple Coffee, Milk" and increasing the cost to $6.50.
3. **Adding Sugar:** The client further wraps the milk coffee with a SugarDecorator, updating the description to "Simple Coffee, Milk, Sugar" and increasing the cost to $7.00.

**Benefits:**

1. **Flexibility:** Allows dynamic addition of behavior without altering existing code. You can mix and match decorators as needed.
2. **Single Responsibility Principle:** Each decorator class focuses on a specific feature, adhering to the principle of having a single responsibility.
3. **Scalability:** New features can be added easily by creating new decorators, without modifying existing code.