**The Scenario**:

Imagine you're working with a coffee shop application where customers can order different types of coffee. Each coffee can be customized with additional ingredients like milk or sugar.

**Decorator Design Pattern Explained**:

* **Enhancing Objects Dynamically**: The Decorator design pattern allows you to add or modify functionalities to objects dynamically without changing their actual class.
* **Common Interface**: You start by defining a common interface called IBeverage. This interface includes methods to get the description and cost of a beverage.
* **Concrete Component**: You implement the Coffee class, which represents a basic coffee. This is a concrete component that implements the IBeverage interface.
* **Abstract Decorator**: You create an abstract class called CondimentDecorator. This class is also an implementation of IBeverage, but it holds a reference to another IBeverage instance. This forms the base for all decorators.
* **Concrete Decorators**: You then implement concrete decorator classes (MilkDecorator and SugarDecorator) that inherit from CondimentDecorator. These classes add specific functionalities to the beverages, like adding milk or sugar.
* **Using Decorators**: In the Main method, you demonstrate how decorators work. You create a base coffee, then wrap it with decorators to add functionalities. The decorators enhance the description and cost of the beverage.

**In Simple Words**:

Think of the Decorator pattern like dressing up a basic outfit. The outfit (base component) can be a plain T-shirt. Now, you can add different layers of clothing (decorators) like jackets, scarves, or hats to enhance the outfit's appearance. Each layer adds something new, but the base outfit remains the same.

In this code, the Decorator pattern allows you to take a basic coffee and add extra ingredients or customizations (decorators) like milk or sugar. Each decorator modifies the coffee's description and cost, giving you flexibility in creating customized beverages while keeping the code clean and modular.

**Step 1: Define the Base Interface**

In this step, you define the IBeverage interface, which represents the base component. It has methods to get the description and cost of a beverage.

internal class Program

{

**// Base interface for beverages**

public interface IBeverage

{

string GetDescription();

double GetCost();

}

**// ...**

}

**Step 2: Implement the Concrete Component**

In this step, you implement the Coffee class, which implements the IBeverage interface. This class represents a basic coffee.

public class Coffee : IBeverage

{

**// ...**

}

**Step 3: Create the Decorator**

In this step, you create an abstract class CondimentDecorator, which also implements the IBeverage interface. This class serves as the base for all decorators. It contains an instance of the base IBeverage and defines abstract methods for description and cost.

public abstract class CondimentDecorator : IBeverage

{

protected IBeverage \_beverage;

public CondimentDecorator(IBeverage beverage)

{

\_beverage = beverage;

}

public abstract string GetDescription();

public abstract double GetCost();

}

**Step 4: Implement Concrete Decorators**

In this step, you implement concrete decorators that inherit from CondimentDecorator. These decorators add specific functionality (e.g., milk or sugar) to existing beverages.

public class MilkDecorator : CondimentDecorator

{

**// ...**

}

public class SugarDecorator : CondimentDecorator

{

**// ...**

}

**Step 5: Using the Decorator Pattern**

In the Main method, you create instances of a base component (Coffee) and then wrap it with decorators (MilkDecorator and SugarDecorator). You call methods on these decorated instances to get descriptions and costs.

static void Main(string[] args)

{

**// Creating a base coffee**

IBeverage coffee = new Coffee();

Console.WriteLine("Base beverage: " + coffee.GetDescription() + " $" + coffee.GetCost());

**// Adding milk to the coffee**

IBeverage milkCoffee = new MilkDecorator(coffee);

Console.WriteLine("With milk: " + milkCoffee.GetDescription() + " $" + milkCoffee.GetCost());

**// Adding sugar to the milk coffee**

IBeverage sugarMilkCoffee = new SugarDecorator(milkCoffee);

Console.WriteLine("With sugar: " + sugarMilkCoffee.GetDescription() + " $" +

sugarMilkCoffee.GetCost());

}

**Explanation of Decorator Pattern**:

The Decorator design pattern allows you to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

In this example,

* IBeverage interface defines the common operations for all beverages.
* The Coffee class is a concrete component that implements the IBeverage interface. It's the basic beverage without any additional decorations.
* The CondimentDecorator is the abstract decorator class that inherits from IBeverage. It holds a reference to a IBeverage instance and provides abstract methods for description and cost.
* The concrete decorator classes (MilkDecorator and SugarDecorator) inherit from CondimentDecorator and add specific behaviors (adding milk or sugar) to the beverages.
* In the Main method, you demonstrate the Decorator pattern by creating instances of a base beverage and wrapping them with decorators. This allows you to dynamically add functionalities (decorations) to the beverages and get the updated descriptions and costs.

In this code example, the Decorator pattern lets you create and modify beverage objects with different combinations of functionalities (decorators) in a flexible and dynamic manner.