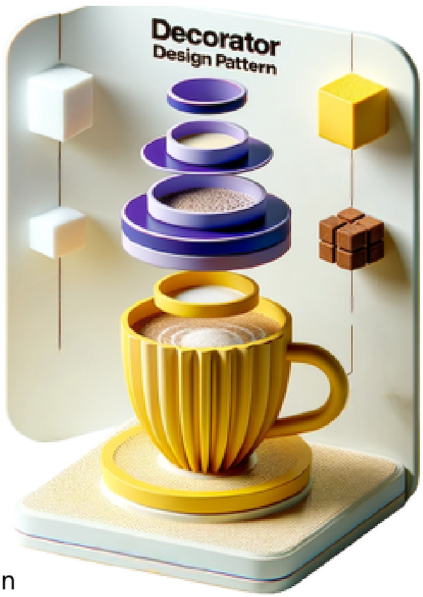
Decorator Pattern

Real-World Example

Imagine you're at a **coffee shop**, and you order just plain coffee. That's your basic beverage. Now, say you want to add some extras to it - like milk, whipped cream, or chocolate. In the world of software design, the **Decorator pattern is like being able to add these extras to your coffee** without having to prepare a whole new batch of coffee for each combination.

Definition

The Decorator pattern is a structural design pattern that allows for the dynamic addition of behaviors to objects without modifying their structure.



You start with your plain coffee. This is your main object. When you decide to add milk, instead of changing the coffee itself, you just wrap it with a "milk layer." Want whipped cream too? Wrap it again with a "whipped cream layer." Each addition is like a new layer around the original coffee, enhancing it without altering the coffee itself.

The Decorator pattern allows you to **build up complex objects step by step**, by wrapping them with additional functionalities (milk, chocolate, whipped cream), all **without altering the original object's core**.

Fun fact

Originally, the Decorator Pattern was extensively used in GUI development for dynamically adding features like scrolling, borders, and other behaviors to windows or components.

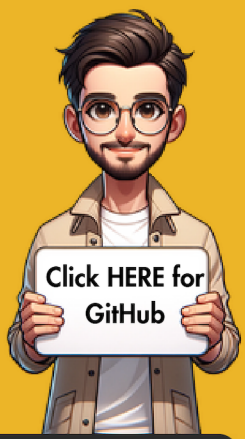
```csharp
public interface ICoffee
{
    string GetDescription();
    double GetCost();
}

public class PlainCoffee : ICoffee
{
    public string GetDescription()
                  => "Plain Coffee";

    public double GetCost()
                  => 2.0;
}
```

Component interface representing the plain coffee

Concrete component representing plain coffee




```csharp
public abstract class CoffeeDecorator : ICoffee
{
    protected ICoffee coffee;

    public CoffeeDecorator(ICoffee coffee)
    {
        this.coffee = coffee;
    }

    public virtual string GetDescription()
    {
        return coffee.GetDescription();
    }

    public virtual double GetCost()
    {
        return coffee.GetCost();
    }
}
```

```csharp
public class MilkDecorator : CoffeeDecorator
{
    public MilkDecorator(ICoffee coffee)
    : base(coffee) { }

    public override string GetDescription()
    {
        return $"{coffee.GetDescription()} + Milk";
    }

    public override double GetCost()
    {
        return coffee.GetCost() + 0.5;
    }
}
```

```csharp
ICoffee coffee = new PlainCoffee();
Console.WriteLine($"Order: {coffee.GetDescription()}, Cost: ${coffee.GetCost()}");


coffee = new MilkDecorator(coffee);
Console.WriteLine($"Order: {coffee.GetDescription()}, Cost: ${coffee.GetCost()}");


coffee = new WhippedCreamDecorator(coffee);
Console.WriteLine($"Order: {coffee.GetDescription()}, Cost: ${coffee.GetCost()}");


coffee = new ChocolateDecorator(coffee);
Console.WriteLine($"Order: {coffee.GetDescription()}, Cost: ${coffee.GetCost()}");
```

Add milk to the coffee

Add whipped cream to the coffee

Add chocolate to the coffee

PROS

Single Responsibility:
Encourages single responsibility for decorators

Modular Design:
Promotes reusable and modular code.

Customization:
Fine-tune objects by combining decorators.

Flexibility:
Dynamically modify object behavior.

CONS

Complexity:
Can become complex with many decorators.

Order Matters:
Decorator order affects behavior.

Unwrapping Challenge:
It can be tricky to remove a specific decorator from the stack of wrappers.

Use the Decorator...

when you want to **add additional functionalities to objects dynamically** without disrupting existing code that relies on these objects.

when **extending an object's** behavior through inheritance **is cumbersome** or not feasible.