Decorator Pattern

LCSCI5202: Object Oriented Design Week 11

Waleed Iqbal – 2025/26

Learning Outcomes

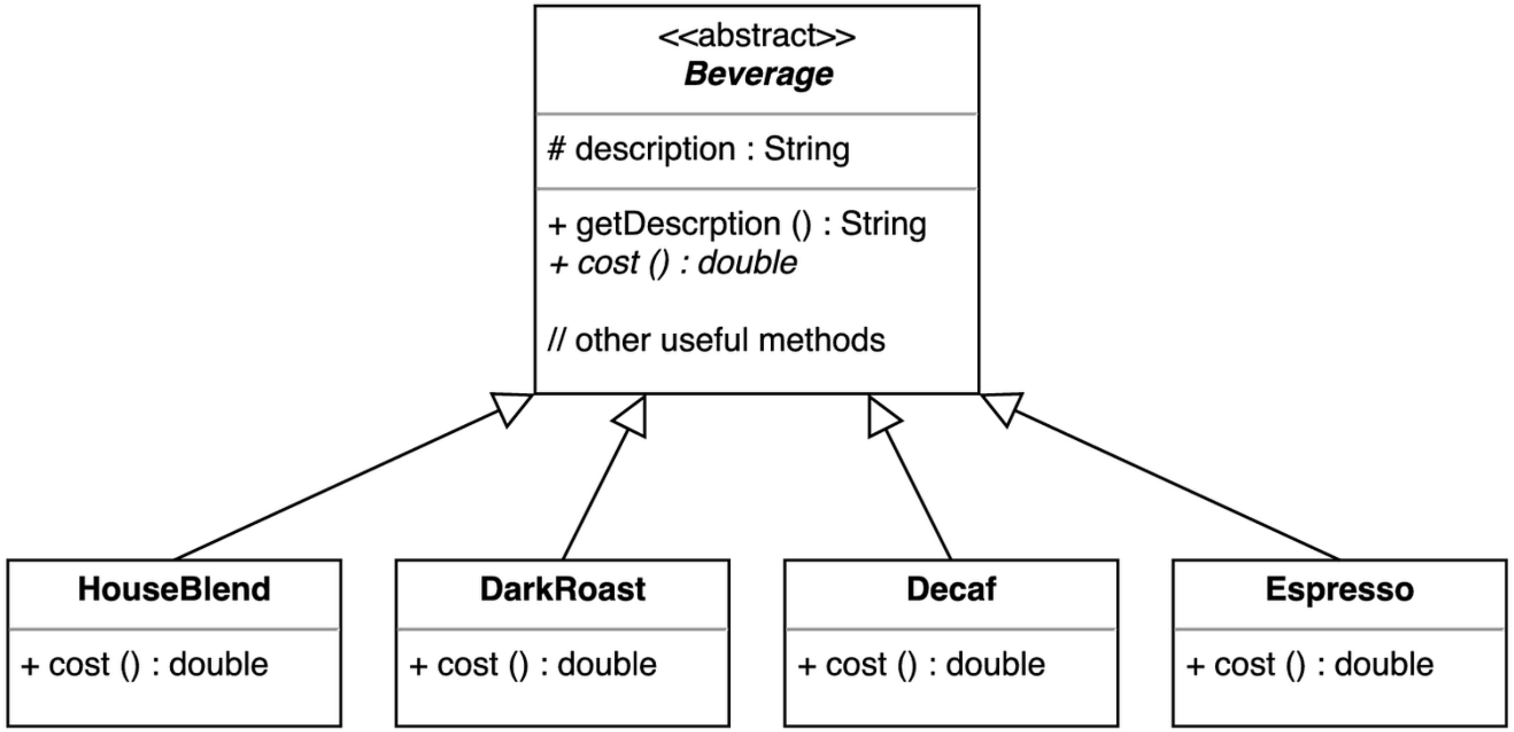
- By the end of this lecture, student would be able to:
  - Understand issues with traditional Object Oriented Design when dealing with Class Explosion issue.
  - Understand need of Decorator Pattern.
  - Implement Decorator Pattern with different variations

Coffee Shop Scenario

- Imagine you're building a system for a coffee shop. You need to calculate the cost of beverages.
- Basic Beverages:
    - Espresso: £1.99
    - House Blend: £0.89
    - Dark Roast: £0.99
    - Decaf: £1.05

Coffee Shop Scenario

- This is simple with inheritance:



Customers want Add-ons in their coffee:

- Milk (+£0.20)
- Mocha (+£0.30)
- Whip Cream (+£0.15)
- Soy Milk (+£0.25)
- Caramel (+£0.35)
- Customer Order Example:
  - "I'd like a Dark Roast with Mocha, Whip Cream, and Double Milk"
    Cost = £0.99 + £0.30 + £0.15 + (£0.20 × 2) = £1.84

How do we handle these combinations flexibly?

Inheritance

```
abstract class Beverage {
            public abstract cost(){}
}

public class EspressoWithMilk : Beverage{
            cost(){return 0.5}
}

public class EspressoWithMilkAndMochaAndWhip : Beverage {
            cost(){return 1.0}
}

public class HouseBlendWithMilk : Beverage {
            cost(){return 2.0}
}
public class HouseBlendWithMocha : Beverage {
            cost(){return 2.0}
}
```

Inheritance

```
abstract class Beverage {
		public abstract cost(){}
}

public class EspressoWithMilk : Beverage{
		cost(){return 0.5}
}

public class EspressoWithMilkAndMochaAndWhip : Beverage {
		cost(){return 1.0}
}

public class HouseBlendWithMilk : Beverage {
		cost(){return 2.0}
}
public class HouseBlendWithMocha : Beverage {
		cost(){return 2.0}
}
```

- Problems:
  - Class explosion: With 4 beverages and 5 Add-ons, you need 1024 classes!
  - Not maintainable: Adding a new Add-on means creating dozens of new classes
  - Not flexible: Can't add multiple of the same Add-on (double mocha)

Boolean Flags

```
Abstract class Coffee
{
public bool HasMilk { get; set; }
public bool HasMocha { get; set; }
public bool HasWhip { get; set; }
public bool HasSoy { get; set; }
public bool HasCaramel { get; set; }
public double GetCost()
{
double cost = baseCost;
if (HasMilk) cost += 0.20;
if (HasMocha) cost += 0.30;
if (HasWhip) cost += 0.15;
if (HasSoy) cost += 0.25;
if (HasCaramel) cost += 0.35;
return cost;
}
}
```

Boolean Flags

```
Abstract class Coffee
{
public bool HasMilk { get; set; }
public bool HasMocha { get; set; }
public bool HasWhip { get; set; }
public bool HasSoy { get; set; }
public bool HasCaramel { get; set; }
public double GetCost()
{
double cost = baseCost;
if (HasMilk) cost += 0.20;
if (HasMocha) cost += 0.30;
if (HasWhip) cost += 0.15;
if (HasSoy) cost += 0.25;
if (HasCaramel) cost += 0.35;
return cost;
}
}
```

- Problems:
  - Violates Open/Closed Principle: Must modify class for new Add-ons
  - Can't handle multiples (e.g., double mocha)
  - Not scalable: Imagine 50 Add-ons!

What would be the ideal solution for these problems?

The Ideal Solution

- Must Have:
  - Add behavior dynamically
  - Wrap objects at runtime
  - Allow multiple wrappers
  - Keep classes closed for modification
  - Open for extension

- Must Avoid:
  - Class explosion
  - Modifying existing code
  - Rigid structure
  - Tight coupling
  - Complexity

The Decorator Pattern provides exactly this solution!

The Decorator Pattern

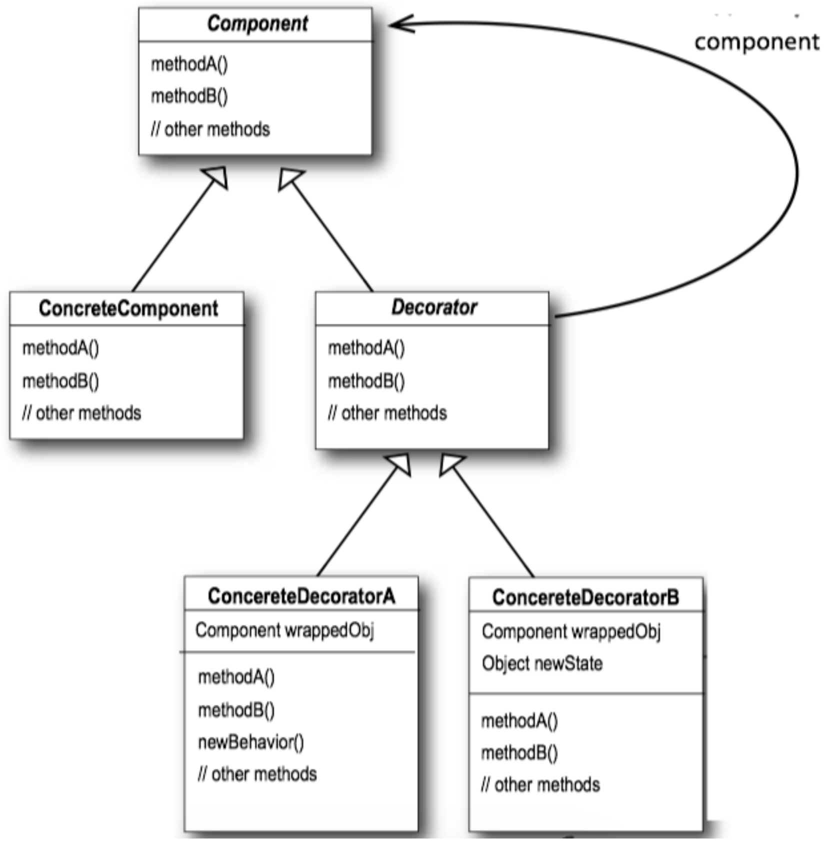
The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

- Wrapping: One object wraps another
- Same Interface: Decorator implements same interface as object it decorates
- Transparency: Client code (Main) doesn't know if it's using decorated or plain object
- Recursive: Can wrap decorators with more decorators

Decorator Pattern Structure

Four Key Players:

- Component: Defines interface for objects that can have responsibilities added
- Concrete Component: Base object that decorators wrap
- Decorator: Abstract class that wraps a Component
- Concrete Decorators: Add specific responsibilities



How Decorator Works - The Flow

```
Order: Espresso with Mocha and Whip
Client calls: beverage.GetCost()
   ↓
Whip Decorator:
   - Calls wrapped.GetCost()  → goes to Mocha
   - Adds $0.15
   ↓
Mocha Decorator:
   - Calls wrapped.GetCost()  → goes to Espresso
   - Adds $0.30
   ↓
Espresso (Base):
   - Returns $1.99
```

Each decorator adds its responsibility and delegates to the wrapped object!

Define Component

- First, we define the Component - the common interface for both base objects and decorators.

```
public abstract class Beverage

{
public string Description { get; set; } = "Unknown Beverage";

public abstract string GetDescription();

public abstract double GetCost();

}
```

Concrete Components

```
public class Espresso : Beverage{
public Espresso(){
Description = "Espresso";
}
public override string GetDescription(){
return Description;
}
public override double GetCost(){
return 1.99;
}}

public class HouseBlend : Beverage{
public HouseBlend(){
Description = "House Blend Coffee";
}
public override string GetDescription(){
return Description;
}
public override double GetCost(){
return 0.89;}}
```

Abstract Decorator

```
public abstract class AddOnDecorator : Beverage
{

protected Beverage wrappedBeverage;

public AddOnDecorator(Beverage beverage) {

wrappedBeverage = beverage;

}

public abstract override string GetDescription();

}
```

Concrete Decorators

```
public class Milk : AddOnDecorator
{
public Milk(Beverage beverage) : base(beverage) {}
public override string GetDescription() {
return wrappedBeverage.GetDescription() + ", Milk";
}
public override double GetCost() {
return wrappedBeverage.GetCost() + 0.20;
}}


public class Mocha : AddOnDecorator {
public Mocha(Beverage beverage) : base(beverage) {}
public override string GetDescription() {
return wrappedBeverage.GetDescription() + ", Mocha";
}
public override double GetCost() {
return wrappedBeverage.GetCost() + 0.30;
}}
```

Add More Concrete Decorators

```
public class Whip : CondimentDecorator{
public Whip(Beverage beverage) : base(beverage){}

public override string GetDescription(){
return wrappedBeverage.GetDescription() + ", Whip";
}
public override double GetCost(){
return wrappedBeverage.GetCost() + 0.15;
}}


public class Soy : CondimentDecorator{
public Soy(Beverage beverage) : base(beverage){}
public override string GetDescription(){
return wrappedBeverage.GetDescription() + ", Soy";
}
public override double GetCost(){
return wrappedBeverage.GetCost() + 0.25;
}}
```

Client Code (Main)

```
static void Main(string[] args)
{
Beverage beverage1 = new Espresso();
Console.WriteLine($"{beverage1.GetDescription()}:
${beverage1.GetCost()}");
Beverage beverage2 = new HouseBlend();
beverage2 = new Mocha(beverage2);
beverage2 = new Whip(beverage2);
Console.WriteLine($"{beverage2.GetDescription()}:
${beverage2.GetCost()}");
Beverage beverage3 = new Espresso();
beverage3 = new Mocha(beverage3);
beverage3 = new Mocha(beverage3);
beverage3 = new Whip(beverage3);
Console.WriteLine($"{beverage3.GetDescription()}:
${beverage3.GetCost()}");
}
```

Summary

- Decorator adds responsibilities dynamically, Pay-as-you-go approach to adding features.
- Follows Open/Closed Principle, No existing code modification needed
- Allows flexible behavior combinations, don't need to define all combinations upfront
- Keeps classes focused (Single Responsibility)

Activity

Design a notification system for a social media app

- Base notification sends a simple message
- Decorators can add:
  - SMS sending
  - Email sending
  - Push notification
  - Slack message
  - Sound alert