Dynamic Extensibility with the Decorator Pattern

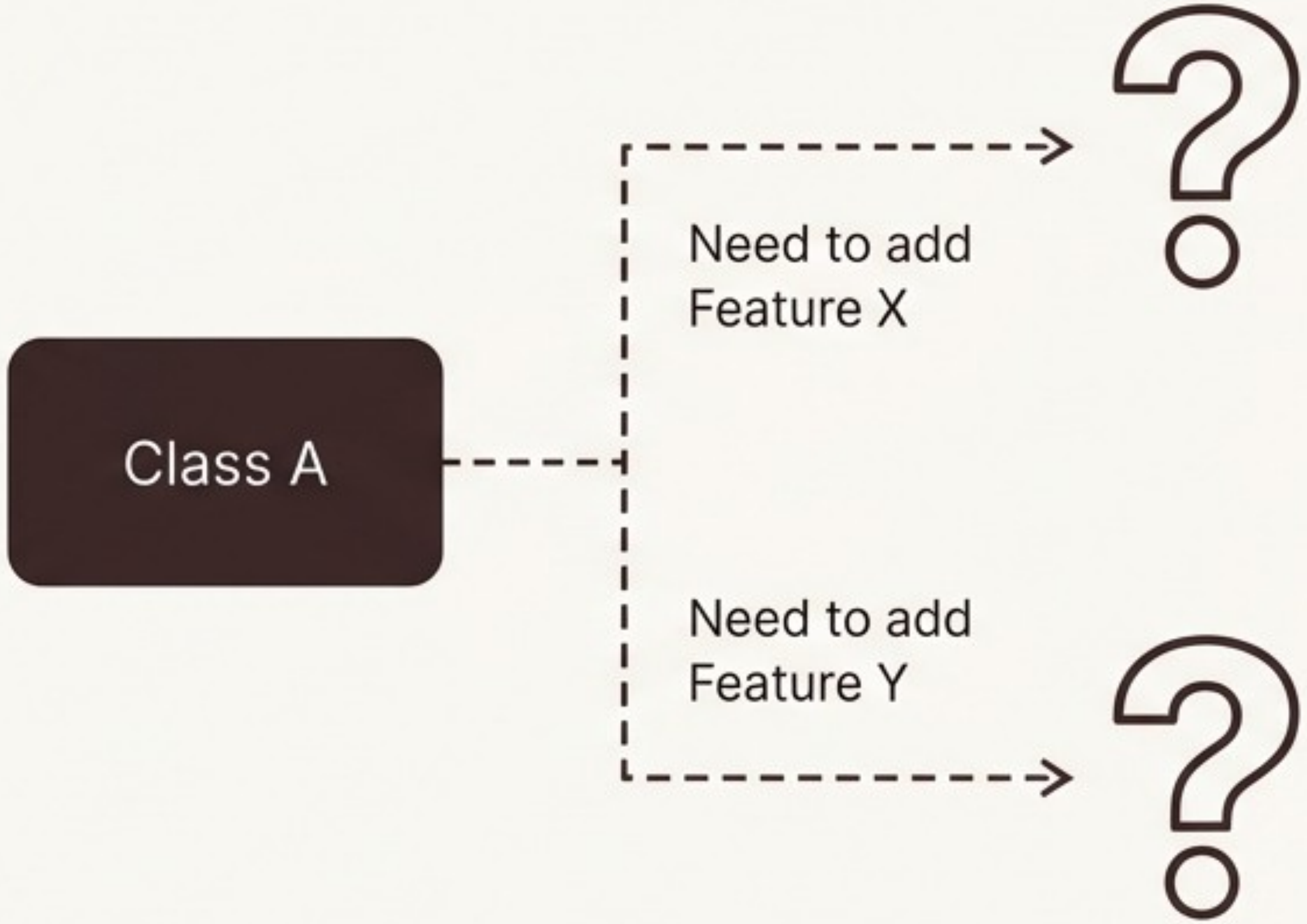
Adding behaviour to objects, one layer at a time.

? The Challenge: Adding Features Without Creating Chaos

We often need to add new features or responsibilities to an object. But doing so presents a dilemma.

We want to avoid:

- ✗ **Modifying the original class:** This violates the Open/Closed Principle and can introduce bugs into stable code.
- ✗ **An explosion of subclasses:** Creating a new subclass for every possible combination of features is unmanageable.

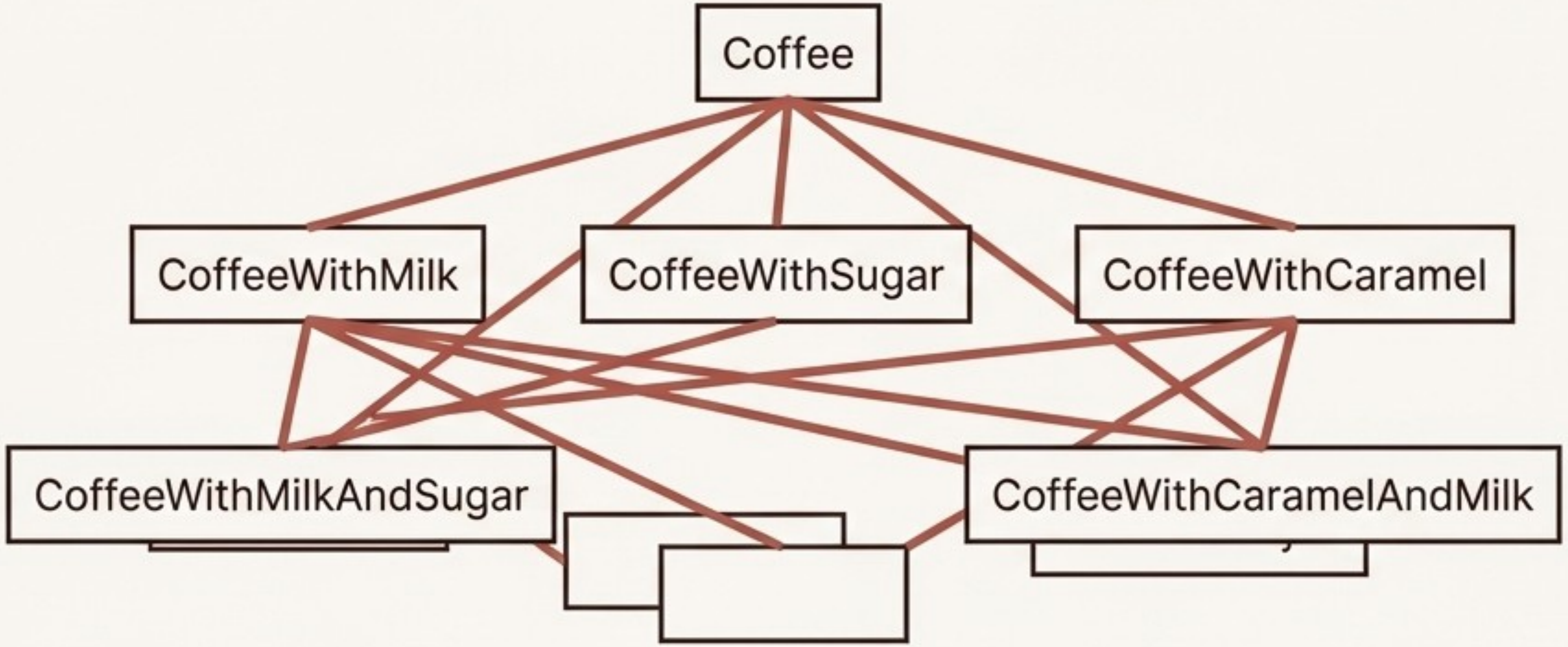


☕ Let’s Think About It Like Ordering a Coffee

- You start with a base product: a simple coffee.
- You then add extras (decorations) one by one: milk, sugar, caramel, chocolate.
- Each extra adds its own cost and flavour, wrapping the drink that came before it.
- Crucially, the original coffee object doesn’t change. The additions are layered on top dynamically.

❌ The Obvious (and Flawed) Approach: Inheritance Explosion

If we use inheritance to represent every coffee combination, our class hierarchy quickly becomes unmanageable.



⚠ **Class Explosion:** The number of subclasses grows exponentially.

⚠ **Rigid:** What if you want Sugar but not Milk? You need a whole new class tree.

⚠ **Maintenance Nightmare:** A change in a base feature ripples through dozens of subclasses.

❌ The Inheritance Approach in Code

```
// The Base Class
public class Coffee
{
    public virtual double Cost() => 50;
}

// First Subclass
public class CoffeeWithMilk : Coffee
{
    public override double Cost() => base.Cost() + 10;
}

// A Subclass of a Subclass...
public class CoffeeWithMilkAndSugar : CoffeeWithMilk
{
    public override double Cost() => base.Cost() + 5;
}
```

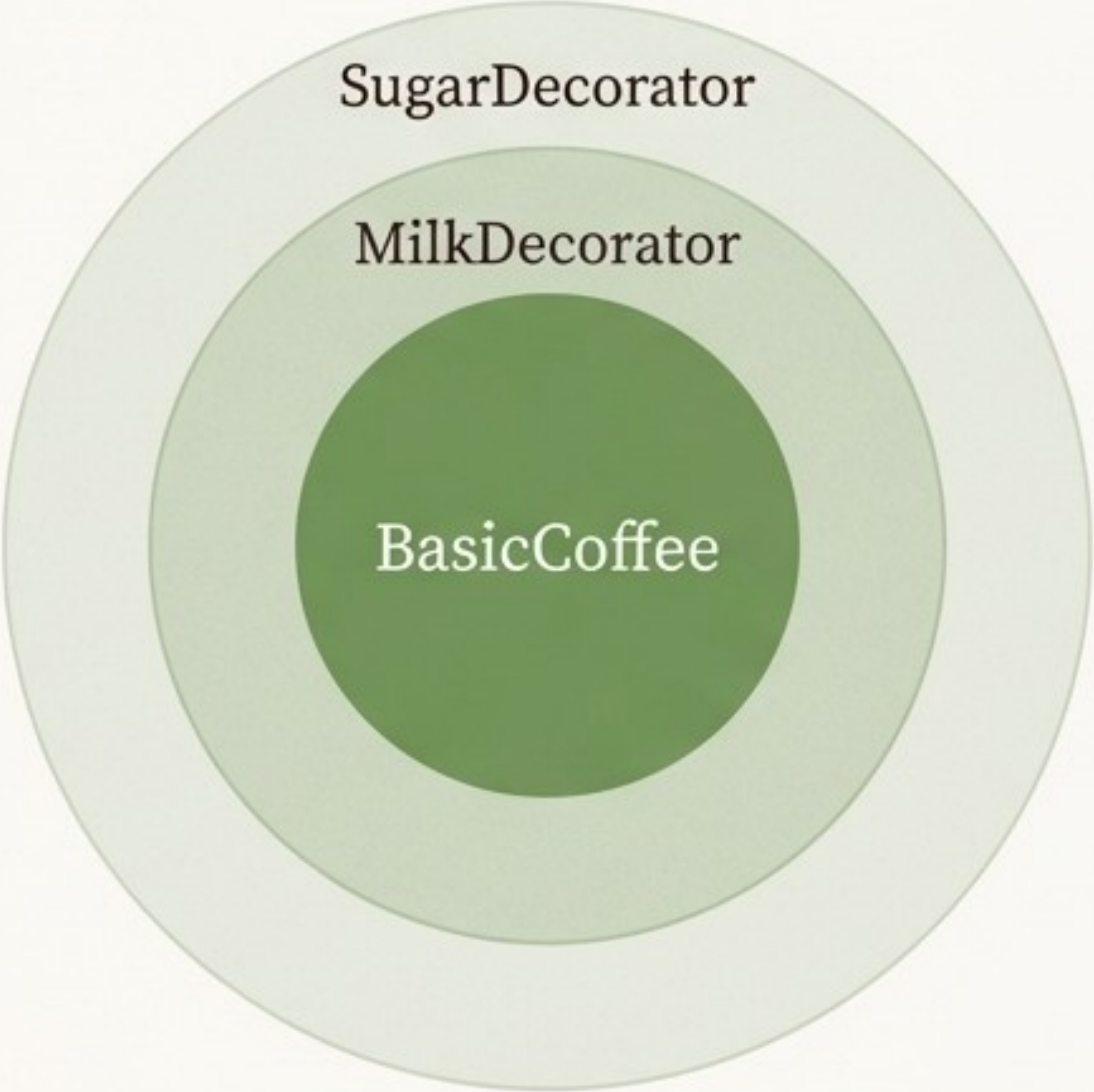
Static relationships defined at compile time.

Violates the Open/Closed Principle: adding a new ingredient like Caramel forces more class modifications.

Combinations are difficult to manage.

✅ The Solution: Add Behaviour by Wrapping, Not Inheriting

The Decorator pattern lets us attach new behaviours to objects by placing them inside special ‘wrapper’ objects that contain the behaviours.



We are using **composition** to build up an object’s functionality dynamically at runtime.

🧑‍💻 Step-by-Step Implementation: The Foundation

Step 1: Define a Common Interface

All objects—both the original and the wrappers—must share a common interface so they are interchangeable.

```
public interface ICoffee
{
    double Cost();
    string Description();
}
```

Step 2: Create the Concrete Component

This is the base object that we will later decorate. It's the starting point.

```
public class BasicCoffee : ICoffee
{
    public double Cost() => 50;
    public string Description() => "Basic Coffee";
}
```

🧑‍💻 Step-by-Step Implementation: The Base Decorator

Step 3: Create the Abstract Base Decorator

This class is the key to the pattern. It serves as the base for all concrete decorators.

Two Critical Rules for the Base Decorator:

1. It must **implement the same interface** (`ICoffee`) as the object it wraps. This makes it ‘invisible’ to the client.
2. It must **hold a reference** to the wrapped object, so it can delegate calls to it.

```
public abstract class CoffeeDecorator : ICoffee
{
    protected readonly ICoffee _coffee;

    protected CoffeeDecorator(ICoffee coffee)
    {
        _coffee = coffee;
    }

    // Delegate calls to the wrapped object by default
    public virtual double Cost() => _coffee.Cost();
    public virtual string Description() => _coffee.Description();
}
```

👨‍💻 Step-by-Step Implementation: The Concrete Decorators

Step 4: Create Concrete Decorators for Each Feature

Each decorator adds its own specific behaviour before or after delegating the call to the wrapped object.

```
// Code for MilkDecorator
public class MilkDecorator : CoffeeDecorator
{
    public MilkDecorator(ICoffee coffee) :
base(coffee) {}

    public override double Cost() =>
base.Cost() + 10;// Adds 10 to the wrapped cost

    public override string Description() =>
base.Description() + ", Milk"; // Appends to
the description
}
```

```
// Code for SugarDecorator
public class SugarDecorator : CoffeeDecorator
{
    public SugarDecorator(ICoffee coffee) :
base(coffee) {}

    public override double Cost() =>
base.Cost() + 5; // Adds 5 to the wrapped cost

    public override string Description() =>
base.Description() + ", Sugar"; // Appends to
the description
}
```

🧑‍💻 The Result: Building Combinations Dynamically

Step 5: The Client Composes the Object

The client can now mix and match decorators at runtime to create any combination it needs, without a single new subclass.

```
// Start with a basic coffee
ICoffee coffee = new BasicCoffee();

// Wrap it with Milk
coffee = new MilkDecorator(coffee);

// Wrap it again with Sugar
coffee = new SugarDecorator(coffee);

// Get the final result
Console.WriteLine(coffee.Description());
Console.WriteLine(coffee.Cost());
```

```
> Basic Coffee, Milk, Sugar
> 65.0
```

Key Benefits Highlighted

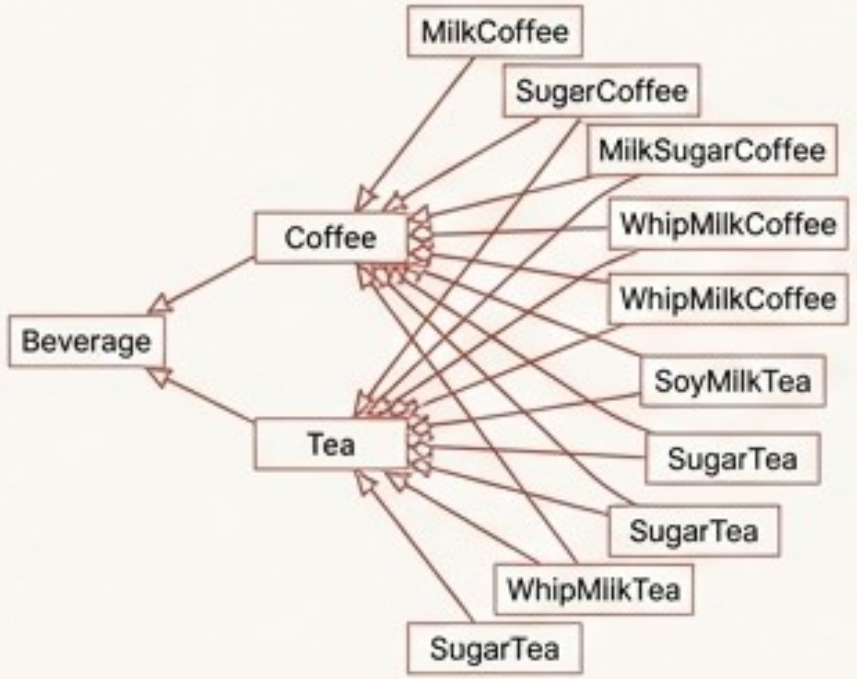
✅ No change to original `BasicCoffee` class.

✅ No subclass explosion.

✅ Fully flexible and dynamic.

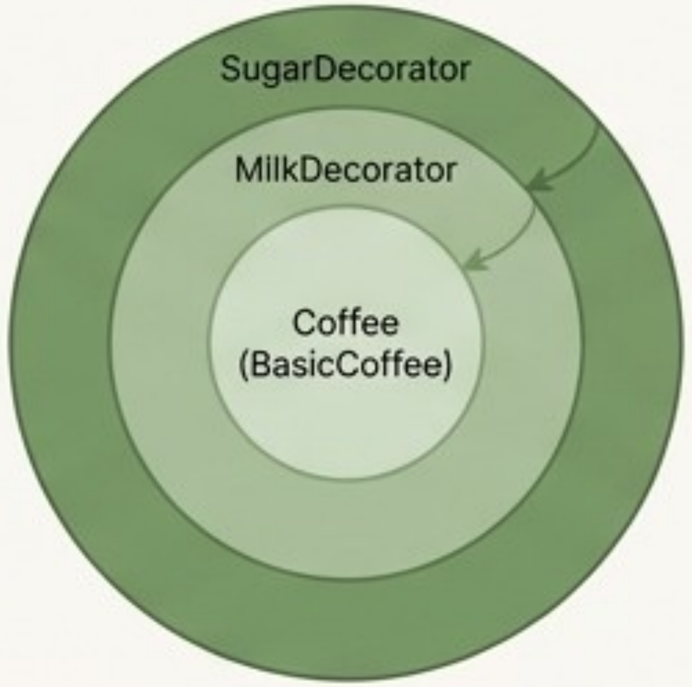
🧩 The Transformation: From Inheritance Hell to Compositional Heaven

The Old Way: Inheritance



- Modify the class or its children for every change.
- Compile-time rigidity.
- 'Inheritance Hell'.

The Decorator Way: Composition



- Add new features by creating new wrapper classes.
- Runtime flexibility.
- 'Plug-and-play decorations'.

🌍 Decorators in the Wild: Real-World Use Cases

This pattern is not just theoretical; it's a workhorse in modern software frameworks.

ASP.NET Core Middleware

Each piece of middleware (Authentication, Logging, Caching) wraps the next, decorating the HTTP request pipeline.

Stream APIs

The .NET `Stream` classes are a classic example.

A `FileStream` is being decorated by a `GZipStream to add compression behaviou

```
var stream = new GZipStream(new FileStream("file.txt",
    FileMode.Open), CompressionMode.Compress);
```

Common Application Wrappers

Logging wrappers, Caching wrappers, Retry logic (e.g., using the Polly library).

UI Frameworks

Adding borders, shadows, or scrolling to visual components.

🎯 When to Use the Decorator Pattern

Use Decorator when you need to add responsibilities to individual objects dynamically and transparently, without affecting other objects.

Use When...

✅ You want to add behaviour to objects without modifying their source code.

✅ You need to support many different combinations of features, and subclassing would be impractical.

✅ You want an object's behaviour to be determined at runtime, not compile time.

Signs You Might Need Decorator

“ Our `Product` class is getting bloated with optional features.

“ We keep creating subclasses just to add one small piece of functionality.

🚫 When to Avoid the Decorator Pattern

The pattern introduces many small objects that can complicate debugging and system design if overused.

Avoid When...

- ✗ The core object is simple and unlikely to need dynamic feature additions.
- ✗ You only need one or two simple feature additions that can be handled with a subclass or a simple boolean property.
- ✗ The additional behaviour is better
- ✗ The additional behaviour is better solved with a different pattern (like Strategy) or simple configuration.

⚠

Overusing this pattern can lead to **“Wrap Hell”:** a long chain of wrappers that is difficult to inspect and debug. Use it where the flexibility is genuinely needed.

The Decorator Pattern: A Final Blueprint

Key Principles Followed

Open/Closed Principle (OCP)

You can add new decorators (new features) without ever modifying existing code.

Single Responsibility Principle (SRP)

Each decorator has one job: to add its specific feature.

Composition Over Inheritance

The pattern's foundation. It favours flexible 'has-a' relationships over rigid 'is-a' relationships.

Item	Meaning
Problem	Need to add features without modifying a class.
Solution	Wrap an object to extend its behaviour dynamically.
Key Idea	Composition, not inheritance.
Benefits	Flexible, reusable, avoids subclass explosion.
Used In	Logging, Caching, Middleware, UI, Streams.