



**Filipe Klinger Lima** • 2nd

Senior Software Engineer | C# | Angular | React | Full Stack | AWS | Azure

1w • 🌐

[Follow](#)



## 📦 Decorator Pattern in C# – Extending Behavior Without Modifying Code

Sometimes we need to add validation, logging, or extra processing to existing logic—without modifying the core class. That's where the Decorator Pattern shines: it lets you wrap behavior around objects, one step at a time.

### 🚧 The Problem – Hard-to-Extend Processing Logic

Imagine we're building an order processing system. At first, it just needs to save the order. Later, we need to add:

- Validation
- Logging
- Fraud checks

A naive solution would modify the core `OrderProcessor` every time, violating:

- ◆ Open/Closed Principle (OCP) – Classes should be open for extension, closed for modification
- ◆ Single Responsibility Principle (SRP) – One class doing too many unrelated tasks

### ✅ Applying the Decorator Pattern

Instead of adding logic to the main class, we:

- Create a common `IOrderProcessor` interface
- Implement a simple `BaseOrderProcessor`
- Add decorators that wrap it with additional steps like validation and logging

Each decorator focuses on one responsibility and delegates the rest.

### 🎯 Using the Decorator Pattern

We build the processing pipeline by wrapping processors. For example, we can first validate the order, then log it, and finally process it.


This approach is modular, testable, and easy to extend.

### 🔥 Benefits of the Decorator Pattern

- ✅ Each step is reusable and independent
- ✅ Behaviors can be composed dynamically
- ✅ No need to modify existing code
- ✅ Great for adding cross-cutting concerns (validation, logging, authorization, etc.)

💬 Have you used the Decorator Pattern this way? Let's talk in the comments! 🙋

#CSharp #DesignPatterns #DecoratorPattern #SOLID #OrderProcessing #CodeQuality

 **Applying the Decorator Pattern**

```
// Step 1 - Interface
public interface IOrderProcessor
{
    void Process(Order order);
}

// Step 2 - Base Processor
public class BaseOrderProcessor : IOrderProcessor
{
    public void Process(Order order)
    {
        Console.WriteLine($"✅ Order {order.Id} processed.");
    }
}
```

```
// Step 3 - Abstract Decorator
public abstract class OrderProcessorDecorator : IOrderProcessor
{
    protected readonly IOrderProcessor _inner;

    public OrderProcessorDecorator(IOrderProcessor inner)
    {
        _inner = inner;
    }

    public virtual void Process(Order order)
    {
        _inner.Process(order);
    }
}

public override void Process(Order order)
{
    Console.WriteLine($"❌ Validating order {order.Id}...");
    if (Logging.IsDebugEnabled) Console.WriteLine($"Order {order.Id} is being validated.");
    base.Process(order);
}

public class LoggingProcessor : OrderProcessorDecorator
{
    public LoggingProcessor(IOrderProcessor inner) : base(inner) {}

    public override void Process(Order order)
    {
        Console.WriteLine($"🔍 Logging order {order.Id} at {DateTime.Now}");
        base.Process(order);
    }
}

public class FraudCheckProcessor : OrderProcessorDecorator
{
    // ...
}

// Step 4 - Order Class
public class Order
{
    public int Id { get; set; }
    public string Customer { get; set; }
}

// Step 5 - Usage
var processor = new ValidationProcessor(
    new FraudCheckProcessor(
        new LoggingProcessor(
            new BaseOrderProcessor()
        )
    )
);

processor.Process(new Order { Id = 1001, Customer = "Willow" });
```