Decorator Design Pattern - Comprehensive Guide

Table of Contents

1. Introduction
2. Pattern Structure and Components
3. UML Diagram
4. Real-world Examples
5. Implementation Details
6. Complete Code Example
7. Advanced Examples
8. Design Principles
9. Advantages and Disadvantages
10. When to Use and When to Avoid
11. Common Pitfalls
12. Related Patterns

Introduction

The **Decorator Design Pattern** is a structural design pattern that allows you to dynamically add new behaviors and responsibilities to objects at runtime without altering their structure. This pattern provides a flexible alternative to subclassing for extending functionality.

Key Concept

The Decorator pattern uses **composition over inheritance** to wrap objects in a series of decorator objects. Each decorator adds its own behavior either before and/or after delegating to the object it decorates.

Real-life Analogy

Think of decorating a Christmas tree:

- **Base tree** (Component): The plain Christmas tree
- **Decorations** (Decorators): Lights, ornaments, garland, star
- You can add decorations in any combination
- Each decoration enhances the tree without changing its core structure
- You can add or remove decorations dynamically

Problem It Solves

Traditional inheritance creates a static class hierarchy. If you need different combinations of features, you'd need to create numerous subclasses, leading to **class explosion**. The Decorator pattern solves this by allowing dynamic composition of behaviors.

Example Problem: Suppose you have a `Coffee` class and want to add features like milk, sugar, whipped cream, and chocolate. Without Decorator pattern, you'd need:

- `Coffee`
- `CoffeeWithMilk`
- `CoffeeWithSugar`
- `CoffeeWithMilkAndSugar`
- `CoffeeWithMilkAndSugarAndChocolate`
- ... and so on (2^n combinations!)

Pattern Structure and Components

The Decorator pattern consists of four main components:

1. Component Interface

- Defines the interface for objects that can have responsibilities added to them dynamically
- Establishes the contract that both concrete components and decorators must follow

2. Concrete Component

- The original object to which additional responsibilities can be attached
- Implements the basic functionality defined by the Component interface

3. Base Decorator

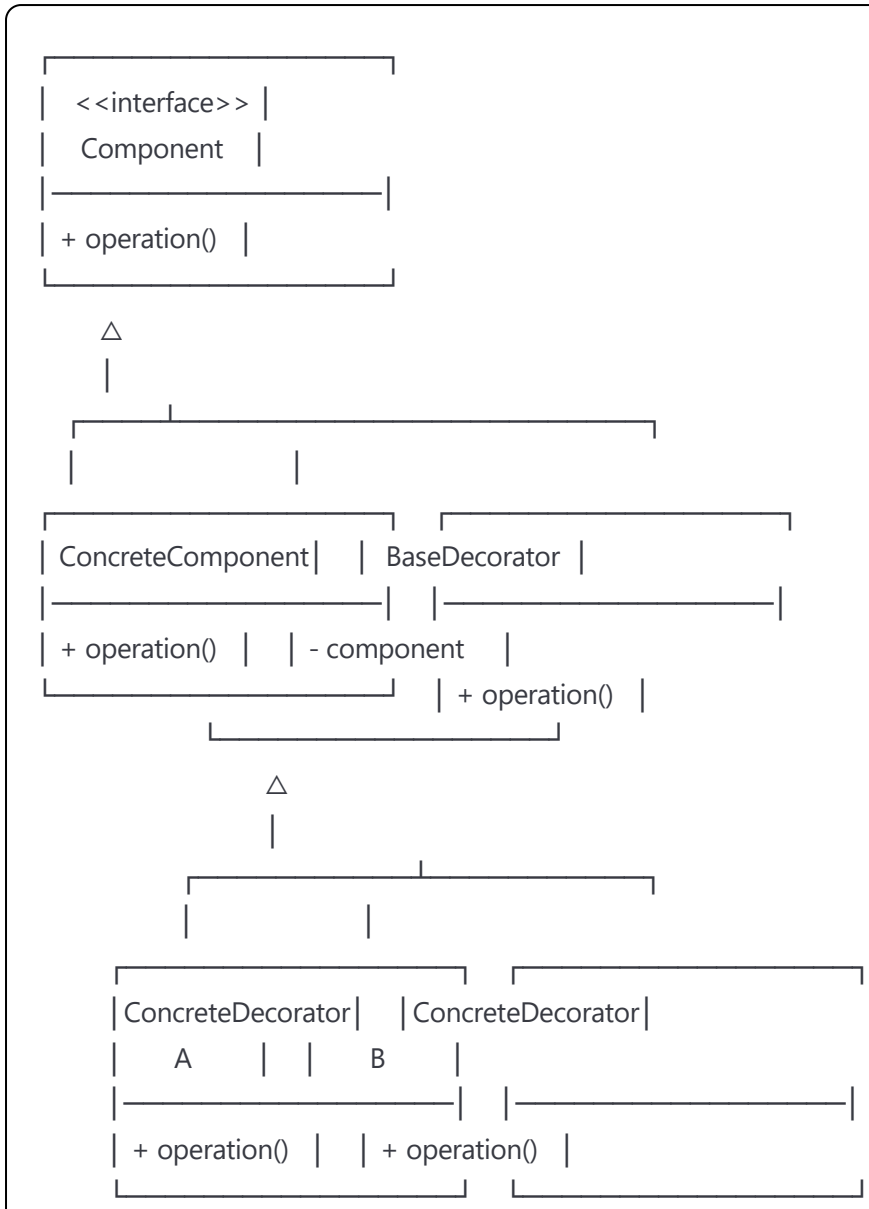
- An abstract class that implements the Component interface
- Maintains a reference to a Component object
- Delegates operations to the wrapped component
- Provides a base for concrete decorators

4. Concrete Decorators

- Extend the Base Decorator
- Add specific responsibilities/behaviors to the component
- Can modify the behavior before or after delegating to the wrapped component

---

UML Diagram



Real-world Examples

1. Java I/O Streams

```java
// Base stream
FileInputStream fileStream = new FileInputStream("file.txt");

// Decorated with buffering
BufferedInputStream bufferedStream = new BufferedInputStream(fileStream);

// Further decorated with data input capabilities
DataInputStream dataStream = new DataInputStream(bufferedStream);
```

2. GUI Components

- Base window → Add scrollbars → Add borders → Add title bar
- Each decorator adds visual/functional enhancements

3. Web Development

- HTTP Request → Authentication middleware → Logging middleware → Caching middleware
- Each middleware decorator adds specific functionality

4. Pizza Ordering System

- Base pizza → Add cheese → Add pepperoni → Add mushrooms
- Each topping is a decorator that adds cost and description

Implementation Details

Design Considerations

1. **Interface Consistency**: All decorators must implement the same interface as the component
2. **Transparent Encapsulation**: Clients should be unaware they're working with decorated objects
3. **Recursive Composition**: Decorators can wrap other decorators
4. **Stateless Decorators**: Decorators should ideally be stateless to avoid side effects

Memory and Performance

- Each decorator adds a layer of indirection
- Memory overhead increases with decoration depth
- Consider object pooling for frequently used decorators

Complete Code Example

Coffee Shop Implementation

java

```java
// 1. Component Interface
public interface Coffee {
    String getDescription();
    double getCost();
    String getSize(); // Additional method for demonstration
}

// 2. Concrete Component - Base Coffee
public class SimpleCoffee implements Coffee {
    private String size;

    public SimpleCoffee() {
        this.size = "Regular";
    }

    public SimpleCoffee(String size) {
        this.size = size;
    }

    @Override
    public String getDescription() {
        return size + " Coffee";
    }

    @Override
    public double getCost() {
        switch(size.toLowerCase()) {
            case "small": return 40.0;
            case "regular": return 50.0;
            case "large": return 60.0;
            default: return 50.0;
        }
    }

    @Override
    public String getSize() {
        return size;
    }
}

// 3. Base Decorator
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        if (coffee == null) {
```

```
            throw new IllegalArgumentException("Coffee cannot be null");
        }
        this.decoratedCoffee = coffee;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }

    @Override
    public String getSize() {
        return decoratedCoffee.getSize();
    }
}

// 4. Concrete Decorators
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + getMilkCost();
    }

    private double getMilkCost() {
        // Cost varies by coffee size
        String size = decoratedCoffee.getSize().toLowerCase();
        switch(size) {
            case "small": return 8.0;
            case "regular": return 10.0;
            case "large": return 12.0;
            default: return 10.0;
        }
    }
```

```
}

public class SugarDecorator extends CoffeeDecorator {
    private int cubes;

    public SugarDecorator(Coffee coffee) {
        this(coffee, 1); // Default 1 cube
    }

    public SugarDecorator(Coffee coffee, int cubes) {
        super(coffee);
        this.cubes = Math.max(1, cubes); // At least 1 cube
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() +
               ", " + cubes + " Sugar cube" + (cubes > 1 ? "s" : "");
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + (cubes * 2.0); // ₹2 per cube
    }
}

public class WhippedCreamDecorator extends CoffeeDecorator {
    public WhippedCreamDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Whipped Cream";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 15.0;
    }
}

public class ChocolateDecorator extends CoffeeDecorator {
    private String chocolateType;

    public ChocolateDecorator(Coffee coffee) {
        this(coffee, "Regular");
```

```
    }

    public ChocolateDecorator(Coffee coffee, String chocolateType) {
        super(coffee);
        this.chocolateType = chocolateType;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", " + chocolateType + " Chocolate";
    }

    @Override
    public double getCost() {
        double cost = decoratedCoffee.getCost();
        switch(chocolateType.toLowerCase()) {
            case "regular": return cost + 15.0;
            case "premium": return cost + 25.0;
            case "dark": return cost + 20.0;
            default: return cost + 15.0;
        }
    }
}

// 5. Utility class for building complex coffee orders
public class CoffeeBuilder {
    private Coffee coffee;

    public CoffeeBuilder(String size) {
        this.coffee = new SimpleCoffee(size);
    }

    public CoffeeBuilder addMilk() {
        this.coffee = new MilkDecorator(this.coffee);
        return this;
    }

    public CoffeeBuilder addSugar(int cubes) {
        this.coffee = new SugarDecorator(this.coffee, cubes);
        return this;
    }

    public CoffeeBuilder addWhippedCream() {
        this.coffee = new WhippedCreamDecorator(this.coffee);
        return this;
    }
```

```
    public CoffeeBuilder addChocolate(String type) {
        this.coffee = new ChocolateDecorator(this.coffee, type);
        return this;
    }

    public Coffee build() {
        return this.coffee;
    }
}
```

Demonstration Class

java

```
public class CoffeeShopDemo {
    public static void main(String[] args) {
        System.out.println("=== COFFEE SHOP DEMO ===\n");

        // 1. Simple orders
        demonstrateSimpleOrders();

        // 2. Complex orders
        demonstrateComplexOrders();

        // 3. Builder pattern integration
        demonstrateBuilderPattern();

        // 4. Different sizes demonstration
        demonstrateSizeVariations();
    }

    private static void demonstrateSimpleOrders() {
        System.out.println("1. SIMPLE ORDERS:");
        System.out.println("-".repeat(30));

        // Basic coffee
        Coffee coffee1 = new SimpleCoffee();
        printOrder(coffee1);

        // Coffee with milk
        Coffee coffee2 = new MilkDecorator(new SimpleCoffee());
        printOrder(coffee2);

        // Coffee with milk and sugar
        Coffee coffee3 = new SugarDecorator(
            new MilkDecorator(new SimpleCoffee())
        );
        printOrder(coffee3);

        System.out.println();
    }

    private static void demonstrateComplexOrders() {
        System.out.println("2. COMPLEX ORDERS:");
        System.out.println("-".repeat(30));

        // Premium coffee with all toppings
        Coffee premiumCoffee = new ChocolateDecorator(
            new WhippedCreamDecorator(
                new SugarDecorator(
```

```
                new MilkDecorator(
                    new SimpleCoffee("Large")
                ), 2
            )
        ), "Premium"
    );
    printOrder(premiumCoffee);

    // Custom chocolate coffee
    Coffee chocolateCoffee = new ChocolateDecorator(
        new ChocolateDecorator(
            new MilkDecorator(new SimpleCoffee()), "Dark"
        ), "Regular"
    );
    printOrder(chocolateCoffee);

    System.out.println();
}

private static void demonstrateBuilderPattern() {
    System.out.println("3. BUILDER PATTERN INTEGRATION:");
    System.out.println("-".repeat(40));

    // Using builder for complex orders
    Coffee specialCoffee = new CoffeeBuilder("Large")
        .addMilk()
        .addSugar(3)
        .addWhippedCream()
        .addChocolate("Premium")
        .build();

    printOrder(specialCoffee);

    Coffee simpleCoffee = new CoffeeBuilder("Small")
        .addMilk()
        .addSugar(1)
        .build();

    printOrder(simpleCoffee);

    System.out.println();
}

private static void demonstrateSizeVariations() {
    System.out.println("4. SIZE VARIATIONS:");
    System.out.println("-".repeat(25));
```

```
        String[] sizes = {"Small", "Regular", "Large"};

        for (String size : sizes) {
            Coffee coffee = new MilkDecorator(new SimpleCoffee(size));
            printOrder(coffee);
        }
    }

    private static void printOrder(Coffee coffee) {
        System.out.printf("%-50s | ₹%.2f%n",
            coffee.getDescription(), coffee.getCost());
    }
}
```

Expected Output:

```
=== COFFEE SHOP DEMO ===

1. SIMPLE ORDERS:
------------------------------
Regular Coffee                                     | ₹50.00
Regular Coffee, Milk                               | ₹60.00
Regular Coffee, Milk, 1 Sugar cube                 | ₹62.00

2. COMPLEX ORDERS:
------------------------------
Large Coffee, Milk, 2 Sugar cubes, Whipped Cream, Premium Chocolate | ₹111.00
Regular Coffee, Milk, Dark Chocolate, Regular Chocolate | ₹85.00

3. BUILDER PATTERN INTEGRATION:
----------------------------------------
Large Coffee, Milk, 3 Sugar cubes, Whipped Cream, Premium Chocolate | ₹113.00
Small Coffee, Milk, 1 Sugar cube                   | ₹50.00

4. SIZE VARIATIONS:
------------------------
Small Coffee, Milk                                 | ₹48.00
Regular Coffee, Milk                               | ₹60.00
Large Coffee, Milk                                 | ₹72.00
```

Advanced Examples

Thread-Safe Decorator Implementation

java

```
public class ThreadSafeCoffeeDecorator extends CoffeeDecorator {
    private final Object lock = new Object();

    public ThreadSafeCoffeeDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        synchronized(lock) {
            return super.getDescription();
        }
    }

    @Override
    public double getCost() {
        synchronized(lock) {
            return super.getCost();
        }
    }
}
```

Decorator with Validation

java

```
public class ValidatingDecorator extends CoffeeDecorator {
    public ValidatingDecorator(Coffee coffee) {
        super(coffee);
        validateCoffee(coffee);
    }

    private void validateCoffee(Coffee coffee) {
        if (coffee.getCost() < 0) {
            throw new IllegalStateException("Coffee cost cannot be negative");
        }
        if (coffee.getDescription() == null || coffee.getDescription().trim().isEmpty()) {
            throw new IllegalStateException("Coffee must have a description");
        }
    }

    @Override
    public String getDescription() {
        String desc = super.getDescription();
        validateCoffee(this);
        return desc;
    }

    @Override
    public double getCost() {
        double cost = super.getCost();
        validateCoffee(this);
        return cost;
    }
}
```

Design Principles

The Decorator pattern adheres to several key design principles:

1. Single Responsibility Principle (SRP)

Each decorator has a single, well-defined responsibility (e.g., adding milk, calculating tax, etc.).

2. Open/Closed Principle (OCP)

Classes are open for extension (through decoration) but closed for modification.

3. Composition over Inheritance

Uses object composition to combine behaviors instead of creating inheritance hierarchies.

4. Decorator Pattern follows the Liskov Substitution Principle

Decorated objects can be used anywhere the original component is expected.

5. Interface Segregation Principle

Components and decorators implement only the interfaces they need.

Advantages and Disadvantages

✅ Advantages

1. **Flexibility at Runtime**
   - Add or remove responsibilities dynamically
   - No need to modify existing code
2. **Avoid Class Explosion**
   - Prevents exponential growth of subclasses
   - Better than inheritance for feature combinations
3. **Incremental Enhancement**
   - Add features step by step
   - Each decorator focuses on a single concern
4. **Transparent to Clients**
   - Decorated objects implement the same interface
   - No client code changes required
5. **Supports Composition**
   - Multiple decorators can be chained
   - Flexible combinations of features
6. **Follows SOLID Principles**
   - Single responsibility per decorator
   - Open for extension, closed for modification

❌ Disadvantages

1. **Increased Complexity**
   - Can create deeply nested object structures
   - Harder to understand and debug
2. **Performance Overhead**
   - Each decoration layer adds method call overhead
   - Memory consumption increases

3. **Debugging Difficulties**
   - Stack traces become complex with multiple decorators
   - Harder to trace execution flow
4. **Object Identity Issues**
   - Wrapped objects lose their original identity
   - `instanceof` checks may fail
5. **Configuration Complexity**
   - Managing decorator combinations can be complex
   - Order of decoration may matter
6. **Testing Challenges**
   - Need to test various decorator combinations
   - Mock objects become more complex

When to Use and When to Avoid

✅ Use Decorator Pattern When:

1. **Dynamic Feature Addition**
   - Need to add responsibilities to objects at runtime
   - Features should be optional and combinable
2. **Avoiding Subclass Explosion**
   - Multiple optional features would create too many subclasses
   - Need flexible feature combinations
3. **Extending Third-party Classes**
   - Cannot modify existing classes (sealed classes, libraries)
   - Need to add functionality to external components
4. **Implementing Middleware/Filters**
   - Processing pipelines (web requests, data streams)
   - Cross-cutting concerns (logging, security, caching)
5. **UI Component Enhancement**
   - Adding visual elements (borders, scrollbars)
   - Behavioral modifications (validation, formatting)

❌ Avoid Decorator Pattern When:

1. **Simple Static Features**
   - Features are always used together

- No need for runtime flexibility

2. **Performance Critical Code**
   - Cannot afford method call overhead
   - Memory usage is strictly limited
3. **Few Feature Combinations**
   - Limited number of possible combinations
   - Simple inheritance hierarchy works fine
4. **Complex State Management**
   - Decorators need to share complex state
   - Order dependencies between decorators

Common Pitfalls

1. Decorator Order Dependency

```java
// Wrong - order matters but not enforced
Coffee coffee = new ChocolateDecorator(
    new MilkDecorator(new SimpleCoffee())
);

// Better - validate dependencies
public class ChocolateDecorator extends CoffeeDecorator {
    public ChocolateDecorator(Coffee coffee) {
        super(coffee);
        if (!hasMilk(coffee)) {
            throw new IllegalArgumentException("Chocolate requires milk");
        }
    }
}
```

2. Null Reference Handling

```java
```

```
// Always validate in base decorator
public CoffeeDecorator(Coffee coffee) {
    if (coffee == null) {
        throw new IllegalArgumentException("Coffee cannot be null");
    }
    this.decoratedCoffee = coffee;
}
```

3. Interface Consistency

```
java

// Wrong - adding methods not in interface
public class BadDecorator extends CoffeeDecorator {
    // This breaks interface consistency
    public void newMethod() { } // Clients can't access this
}
```

4. Excessive Decoration

```
java

// Consider using Builder pattern for complex decorations
public class CoffeeBuilder {
    private Coffee coffee;

    public CoffeeBuilder(Coffee base) {
        this.coffee = base;
    }

    public CoffeeBuilder addMilk() {
        coffee = new MilkDecorator(coffee);
        return this;
    }

    // ... other methods
}
```

Related Patterns

1. Adapter Pattern

- **Similarity**: Both wrap objects and delegate calls
- **Difference**: Adapter changes interface, Decorator preserves it

2. Composite Pattern

- **Similarity**: Both use recursive composition
- **Difference**: Composite represents part-whole, Decorator adds behavior

3. Strategy Pattern

- **Similarity**: Both change object behavior
- **Difference**: Strategy changes algorithm, Decorator adds responsibilities

4. Chain of Responsibility

- **Similarity**: Both chain objects together
- **Difference**: Chain passes requests, Decorator enhances behavior

5. Proxy Pattern

- **Similarity**: Both wrap objects and control access
- **Difference**: Proxy controls access, Decorator adds functionality

---

Summary

The Decorator Design Pattern is a powerful structural pattern that provides a flexible alternative to inheritance for extending object functionality. It allows you to:

- **Wrap objects** in layers of decorators
- **Add behaviors** dynamically at runtime
- **Combine features** in flexible ways
- **Maintain interface consistency** across decorations
- **Follow SOLID principles** for better design

Key Takeaway: Use Decorator pattern when you need to add optional, combinable features to objects without modifying their structure. It's particularly valuable in scenarios where inheritance would lead to class explosion or when you need runtime flexibility.

The pattern shines in areas like I/O streams, GUI components, middleware systems, and any domain where you need to apply multiple optional enhancements to objects.

Remember: With great flexibility comes great responsibility - use decorators judiciously to maintain code clarity and performance! ☕