Great! Let's now explore the **Decorator Design Pattern** with a **real-world example**, **problem-first explanation**, a clean **solution**, a **class diagram**, and a breakdown of **pros and cons** — just like we did earlier.

🎯 What is the Decorator Pattern?

The **Decorator Pattern** allows behavior to be added to an individual object **dynamically** without affecting the behavior of other objects from the same class.

Think of it as *wrapping* an object with new functionality — like adding toppings on a pizza without modifying the original dough.

🧩 Problem Scenario: Coffee Shop Billing System

You are building a billing system for a **Coffee Shop**. Your menu includes:

- Base drinks: Espresso, Latte
- Add-ons: Milk, Mocha, Whip, etc.

A customer can choose:

☕ Espresso + Milk + Whip

❌ Naive Implementation (Problem)

You might try this using a giant inheritance tree like:

```
class EspressoWithMilk extends Beverage {}
class EspressoWithMilkAndWhip extends Beverage {}
class LatteWithMocha extends Beverage {}
```

😖 Problems:

- 🚫 Explosion of subclasses (combinatorial hell)
- 🚫 Not scalable or reusable
- 🚫 Violates Open/Closed Principle
- 😵 Hard to test, debug, and extend

✅ Solution: Use Decorator Pattern

We use the Decorator Pattern to **wrap add-ons (milk, whip)** around the base beverage (Espresso, Latte) dynamically.

🧠 Pattern Breakdown

🔷 Step 1: Component Interface

```
public interface Beverage {
    String getDescription();
    double cost();
}
```

🔷 Step 2: Concrete Components (Base Drinks)

```
public class Espresso implements Beverage {
    public String getDescription() { return "Espresso"; }
    public double cost() { return 2.0; }
}

public class Latte implements Beverage {
    public String getDescription() { return "Latte"; }
    public double cost() { return 2.5; }
}
```

🔷 Step 3: Abstract Decorator

```
public abstract class AddOnDecorator implements Beverage {
    protected Beverage beverage;
    public AddOnDecorator(Beverage beverage) {
        this.beverage = beverage;
    }
}
```

🔷 Step 4: Concrete Decorators (Add-ons)

```
public class Milk extends AddOnDecorator {
    public Milk(Beverage beverage) {
        super(beverage);
    }

    public String getDescription() {
        return beverage.getDescription() + ", Milk";
    }

    public double cost() {
        return beverage.cost() + 0.5;
    }
}
```

```
public class Whip extends AddOnDecorator {
    public Whip(Beverage beverage) {
        super(beverage);
    }

    public String getDescription() {
        return beverage.getDescription() + ", Whip";
    }

    public double cost() {
        return beverage.cost() + 0.3;
    }
}
```

🧪 Runtime Usage

```
Beverage order = new Espresso(); // Base
order = new Milk(order);         // Add-on 1
order = new Whip(order);         // Add-on 2

System.out.println(order.getDescription()); // Espresso, Milk, Whip
System.out.println(order.cost());           // 2.0 + 0.5 + 0.3 = 2.8
```

✓ No class explosion
✓ Add-ons are modular and stackable
✓ Core Espresso class never changed

🏗️ Class Diagram

```
                +-----------------+
                |   Beverage      | ←— (Component Interface)
                +-----------------+
                | + cost()        |
                | + getDescription() |
                +-----------------+
                       ▲
                       |
     +-----------------+-------------------+
     |                                     |
+------------+                  +----------------------+
|  Espresso  |                  |   AddOnDecorator     | ←— Abstract
Decorator
+------------+                  +----------------------+
| + cost()   |                  | - beverage: Beverage |
| + desc()   |                  +----------------------+
                                ▲           ▲
                                |           |
                         +-----------+ +-----------+
                         |   Milk    | |   Whip    | ←— Concrete
Decorators
                         +-----------+ +-----------+
```

✅ Benefits Gained

Benefit	Explanation
✅ No subclass explosion	Add-ons don’t require new subclasses
✅ Runtime flexibility	Add behavior dynamically
✅ Open/Closed Principle	Extend functionality without modifying core
✅ Composition > Inheritance	Behavior is layered using object composition

---

❌ Drawbacks

Drawback	Solution
⚠️ Many small classes	Group or auto-register decorators
⚠️ Debugging wrapper chain	Add logging or use visual composition tools
⚠️ Order matters	Be cautious when wrapping (e.g., Whip before Milk or after)

📦 Summary Table

Aspect	Description
Pattern Name	Decorator
Intent	Dynamically add responsibilities to objects
Use Cases	UI widgets, billing systems, file IO wrappers
Core Idea	Wrap a base object with layered functionality
Flexibility	High — reuses base logic and adds enhancements
