Decorator

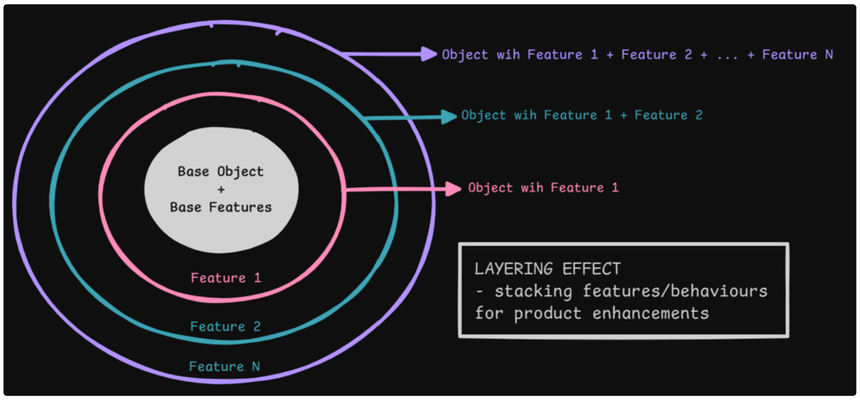
- Definition
- Use cases
  - Example 1: Pizza Shop
  - Example 2: Coffee Cafe
- Why do we need the Decorator Pattern?
- Decorator Pattern as a Solution
  - Class Diagram
  - Understanding the Structure (e.g., Pizza Shop)
  - Breakdown of Relationships
- Implementation
  - Code
  - Output

Resources

- Video → 32. All Structural Design Patterns | Decorator, Proxy, Composite, Adapter, Bridge, Facade, FlyWeight
- Video → 4. Decorator Design Pattern Explanation with Java Coding (Hindi) | LLD System Design

Definition

> ***Decorator pattern allows you to add new functionality to objects dynamically without altering their original structure.***



Object wih Feature 1 + Feature 2 + ... + Feature N
Object wih Feature 1 + Feature 2
Base Object
+
Base Features
Object wih Feature 1
Feature 1
Feature 2
Feature N
LAYERING EFFECT
- stacking features/behaviours
for product enhancements

Use cases

Example 1: Pizza Shop

- **What we have:**
  - **Basic Pizza:** Margherita (Crust + Cheese)
  - **Available Toppings:** Extra Cheese, Olives, Jalapenos, Pepperoni, Veggies, Spicy Red Pepper, etc.
- **Pizzas**
  - **Type 1:** Margherita + Extra Cheese
  - **Type 2:** Margherita + Olives + Jalapenos

  - **Type 3:** Margherita + Olives + Jalapenos + Veggies + Extra Cheese
  - **Type 4:** Margherita + Pepperoni + Spicy Red Pepper

Example 2: Coffee Cafe

- **What we have:**
  - **Espresso:** A strong shot of pure coffee.
  - **Available add-ons:** Sugar, Hot water, Cold Water, Ice, Steamed Milk, Milk Foam, Cold Milk, Chocolate Syrup, Vanilla ice cream, etc.
- **Coffee Beverages**
  - **Type 1:** Doppio → A double shot of espresso.
  - **Type 2:** Americano → Espresso + Hot water.
  - **Type 3:** Cappuccino → Espresso + Steamed Milk + Milk Foam.
  - **Type 4:** Mocha → Espresso + Steamed Milk + Chocolate Syrup.
  - **Type 5:** Cold Coffee → Espresso + Cold Water + Cold Milk + Ice.
  - **Type 6:** MakeYourOwnCoffee → Espresso + Cold Water + Cold Milk + Ice + Vanilla Ice Cream + Chocolate Syrup.

Why do we need the Decorator Pattern?

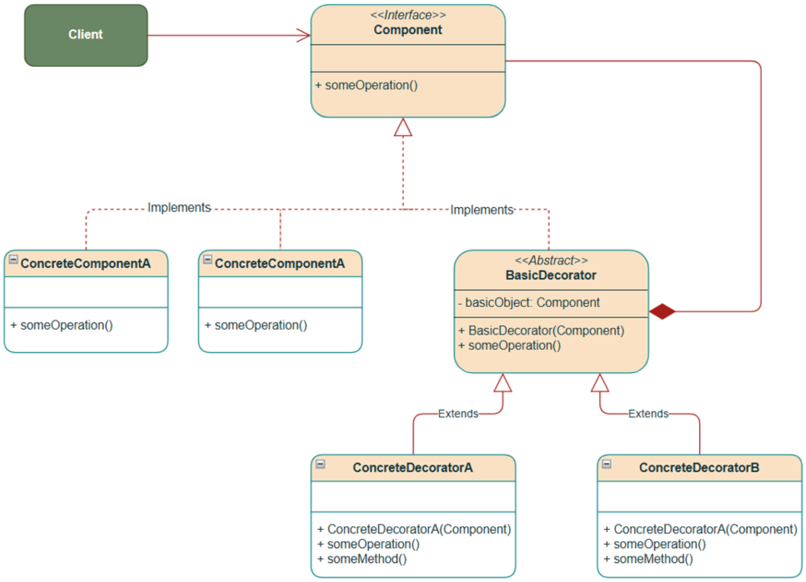
To grasp the necessity of the Decorator Pattern, we must first look into the challenges encountered when implementing the problem using a naive approach, i.e., without utilizing the Decorator Pattern.

Issue: Class Explosion

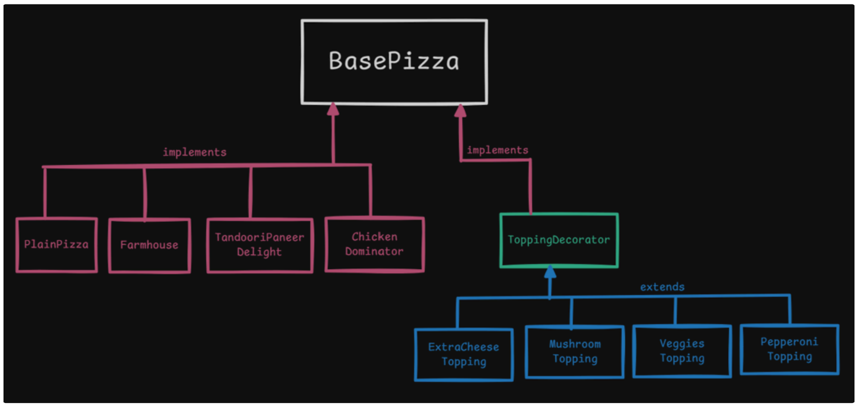
- Class explosion refers to a situation where the number of classes in your system grows rapidly and uncontrollably, making the codebase hard to navigate, maintain, and understand.
- This happens when **every combination of additional behavior requires a new decorator class**(if implemented in a naive way).
- You have a base component in the Coffee Cafe example, i.e., `Coffee`, and which has decorators like:
  - `MilkDecorator`, `SugarDecorator`, `VanillaIceCreamDecorator`, `ChocolateSyrupDecorator,` etc.
  - To support all combinations of these decorators, we need to either manually chain them(which doesn’t allow changes at runtime) or create **new classes for every common combination**, e.g., `MilkAndSugarCoffeeDecorator`, `SugarAndVanillaDecorator`, `MilkVanillaIceCreamDecorator`, `ChocolateSyrupVanillaIceCreamDecorator`, etc.
  - That’s where **class explosion** kicks in; we begin to have **too many classes(extreme subclassing)** to represent all possible combinations of features/behaviours.

Decorator Pattern as a Solution

Class Diagram



Understanding the Structure (e.g., Pizza Shop)



1. **Component Interface (** `BasePizza` **)**
   a. Defines the contract that both the base object and the decorators must follow.
   b. Contains methods like `getDescription()` and `getCost()`.
2. **Concrete Component (** `PlainPizza`, `Farmhouse`, `TandooriPaneerDelight`, `ChickenDominator` **)**
   - The base implementation that provides core functionality.
   - `PlainPizza` represents a simple pizza without any toppings.
   - Can contain a pre-formulated combination of features like `Farmhouse`, `TandooriPaneerDelight`, and `ChickenDominator`.
3. **Base Decorator (** `ToppingDecorator` **)**
   a. An abstract class that implements the Pizza interface.
   b. Contains a reference to a Pizza object - the component itself (composition).

    c. Delegates call the wrapped pizza object.

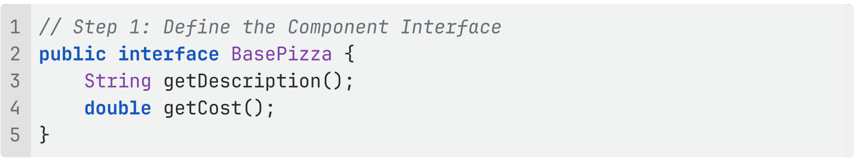
4. **Concrete Decorators (** `ExtraCheeseTopping`, `VeggiesTopping`, `MushroomTopping`, `PepperoniTopping` **, etc.)**
    - Extend the base decorator to add specific functionality.
    - Override methods to add their behavior while calling the wrapped object.
    - Can modify both the description and cost.

Breakdown of Relationships

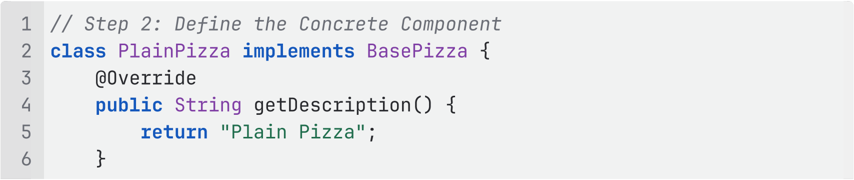
1. **IS-A Relationship** (Inheritance-based)
    - In the above example, **Concrete Components (** `PlainPizza`, `Farmhouse`, `TandooriPaneerDelight`, `ChickenDominator` **)** all implement the **Component Interface (** `BasePizza` **)**.
    - Hence, they are types of Pizza → `IS-A` Relationship.
    - Thant means
        - `PlainPizza` **is-a** `BasePizza`.
        - `ExtraCheeseTopping` **is-a** `ToppingDecorator`.
        - `ToppingDecorator` **is-a** `BasePizza`.
        - Therefore, `ExtraCheeseTopping` **is-a** `BasePizza`.
2. **HAS-A Relationship** (Composition-based)
    - In the above example, `ToppingDecorator` **has a** `BasePizza`.
        - `ExtraCheeseTopping` has a `PlainPizza`, which is a `BasePizza` (e.g., Plain Pizza with extra cheese).
        - `MushroomTopping` has a `PlainPizza` (e.g., Mushroom Pizza).
    - Each new decorator wraps around another `Pizza`, adding its behavior (like toppings).
    - That means
        - `ToppingDecorator` **has-a** `Pizza`.
        - Decorators like `ExtraCheeseTopping`, `MushroomTopping`, etc., **have a** reference to another `Pizza` (result of another decorated component or the component itself).

Implementation

Code



```java
// Step 1: Define the Component Interface
public interface BasePizza {
    String getDescription();
    double getCost();
}
```



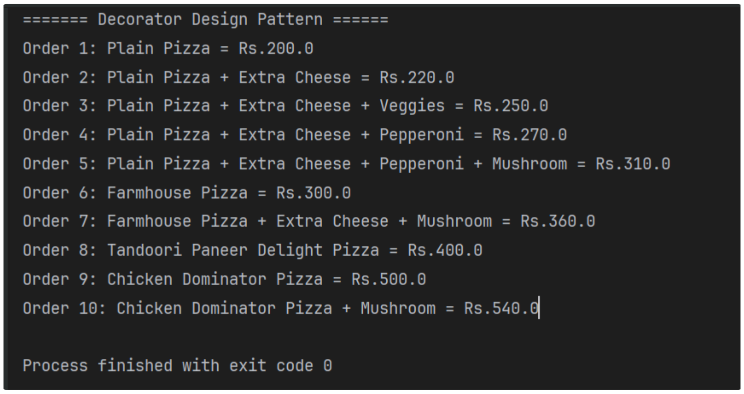
```java
// Step 2: Define the Concrete Component
class PlainPizza implements BasePizza {
    @Override
    public String getDescription() {
        return "Plain Pizza";
    }
    @Override
    public double getCost() {
        return 200.00;
    }
}
public class Farmhouse implements BasePizza {
    @Override
    public String getDescription() {
        return "Farmhouse Pizza";
    }
    @Override
    public double getCost() {
        return 300.0;
    }
}
public class TandooriPaneerDelight implements BasePizza {
    @Override
    public String getDescription() {
        return "Tandoori Paneer Delight Pizza";
    }
    @Override
    public double getCost() {
        return 400.0;
    }
}
public class ChickenDominator implements BasePizza {
    @Override
    public String getDescription() {
        return "Chicken Dominator Pizza";
    }

    @Override
    public double getCost() {
        return 500.0;
    }
}
```

```java
// Step 3: Define the Abstract Base Decorator
public abstract class ToppingDecorator implements BasePizza {
    BasePizza pizza;

    public ToppingDecorator(BasePizza pizza) {
        this.pizza = pizza;
    }
}
```

```java
// Step 4: Define the Concrete Decorator
public class ExtraCheeseTopping extends ToppingDecorator {

    public ExtraCheeseTopping(BasePizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + " + Extra Cheese";
    }

    @Override
    public double getCost() {
        return pizza.getCost() + 20;
    }
}
public class VeggiesTopping extends ToppingDecorator {

    public VeggiesTopping(BasePizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + " + Veggies";
    }

    @Override
    public double getCost() {
        return pizza.getCost() + 30;
    }
}
public class MushroomTopping extends ToppingDecorator {

    public MushroomTopping(BasePizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + " + Mushroom";
    }

    @Override
    public double getCost() {
        return pizza.getCost() + 40;
    }
}
public class PepperoniTopping extends ToppingDecorator {

    public PepperoniTopping(BasePizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + " + Pepperoni";
    }

    @Override
    public double getCost() {
        return pizza.getCost() + 50;
    }
}
```

```java
// Step 5: Client Demonstration
public class PizzaShop {
    public static void main(String[] args) {
        System.out.println("======= Decorator Design Pattern ======");
        // Create a plain pizza
        BasePizza pizza1 = new PlainPizza();
        System.out.println("Order 1: " + pizza1.getDescription() + " = Rs." + pizza1.getCost());

        // Add toppings to the PlainPizza - Extra Cheese Only
        BasePizza pizza2 = new ExtraCheeseTopping(new PlainPizza());
        System.out.println("Order 2: " + pizza2.getDescription() + " = Rs." + pizza2.getCost());

        // Add toppings to the PlainPizza - Extra Cheese and Veggies
        BasePizza pizza3 = new VeggiesTopping(new ExtraCheeseTopping(new PlainPizza()));
        System.out.println("Order 3: " + pizza3.getDescription() + " = Rs." + pizza3.getCost());

        // Add toppings to the PlainPizza - Extra Cheese and Pepperoni
        BasePizza pizza4 = new PepperoniTopping(new ExtraCheeseTopping(new PlainPizza()));
        System.out.println("Order 4: " + pizza4.getDescription() + " = Rs." + pizza4.getCost());

        // Add toppings to the PlainPizza - Extra Cheese, Mushroom and Pepperoni
        BasePizza pizza5 = new MushroomTopping(new PepperoniTopping(new ExtraCheeseTopping(new PlainPizza())));
        System.out.println("Order 5: " + pizza5.getDescription() + " = Rs." + pizza5.getCost());

        // Farmhouse Pizza
        BasePizza pizza6 = new Farmhouse();
        System.out.println("Order 6: " + pizza6.getDescription() + " = Rs." + pizza6.getCost());

        // Farmhouse Pizza with Extra Cheese and Mushroom
        BasePizza pizza7 = new MushroomTopping(new ExtraCheeseTopping(new Farmhouse()));
        System.out.println("Order 7: " + pizza7.getDescription() + " = Rs." + pizza7.getCost());

        // Tandoori Paneer Delight Pizza
        BasePizza pizza8 = new TandooriPaneerDelight();
        System.out.println("Order 8: " + pizza8.getDescription() + " = Rs." + pizza8.getCost());

        // Chicken Dominator
        BasePizza pizza9 = new ChickenDominator();
        System.out.println("Order 9: " + pizza9.getDescription() + " = Rs." + pizza9.getCost());

        // Chicken Dominator with Mushroom
        BasePizza pizza10 = new MushroomTopping(new ChickenDominator());
        System.out.println("Order 10: " + pizza10.getDescription() + " = Rs." + pizza10.getCost());
    }
}
```

Output



```
======= Decorator Design Pattern ======
Order 1: Plain Pizza = Rs.200.0
Order 2: Plain Pizza + Extra Cheese = Rs.220.0
Order 3: Plain Pizza + Extra Cheese + Veggies = Rs.250.0
Order 4: Plain Pizza + Extra Cheese + Pepperoni = Rs.270.0
Order 5: Plain Pizza + Extra Cheese + Pepperoni + Mushroom = Rs.310.0
Order 6: Farmhouse Pizza = Rs.300.0
Order 7: Farmhouse Pizza + Extra Cheese + Mushroom = Rs.360.0
Order 8: Tandoori Paneer Delight Pizza = Rs.400.0
Order 9: Chicken Dominator Pizza = Rs.500.0
Order 10: Chicken Dominator Pizza + Mushroom = Rs.540.0

Process finished with exit code 0
```

The key benefit of the Decorator pattern is that it provides a **flexible alternative** to subclassing for extending functionality, allowing you to **mix and match behaviors as needed at runtime.**