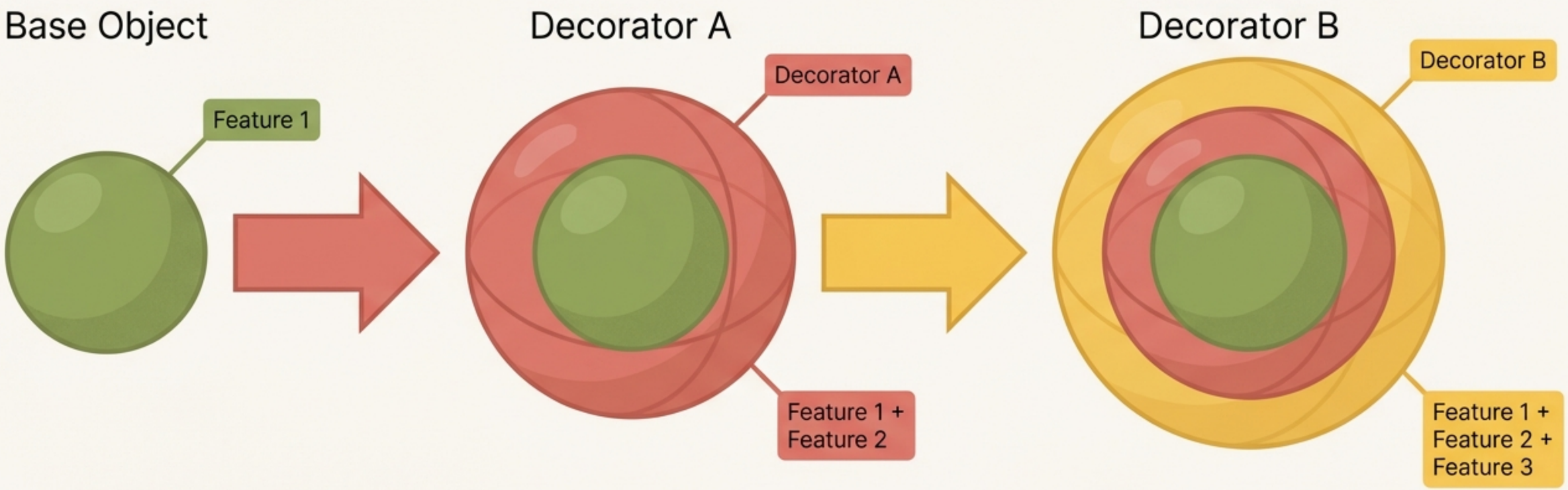
Building Objects, Not Class Hierarchies

An Introduction to the Decorator Design Pattern

The Core Idea: Wrap and Enhance

The Decorator pattern allows you to dynamically add new features or behaviors to an object by placing it inside a special ‘wrapper’ object.



You can keep wrapping objects to add more functionality, like nesting dolls.

You Already Understand This Pattern

The logic of decoration is everywhere in the services we use.

Pizza Shop

You start with a **Base Pizza** and decorate it with toppings like **Extra Cheese** and **Mushrooms**. A pizza with cheese is still a pizza, ready for more toppings.

Coffee Shop

You order a **Base Coffee** (like an espresso) and decorate it with **Cream**, **Extra Milk**, or other additions.

Car Customization

You buy a **Base Car** and add optional features (decorations) like **Air Conditioning**, **Power Steering**, or **Fog Lights**.

The Coding Challenge: Modeling a Pizza Shop

Let's build the software for a pizzeria. We start with a few base pizzas: `Margherita` and `VegDelight`. We also have toppings: `Extra Cheese` and `Mushrooms`.

The Question

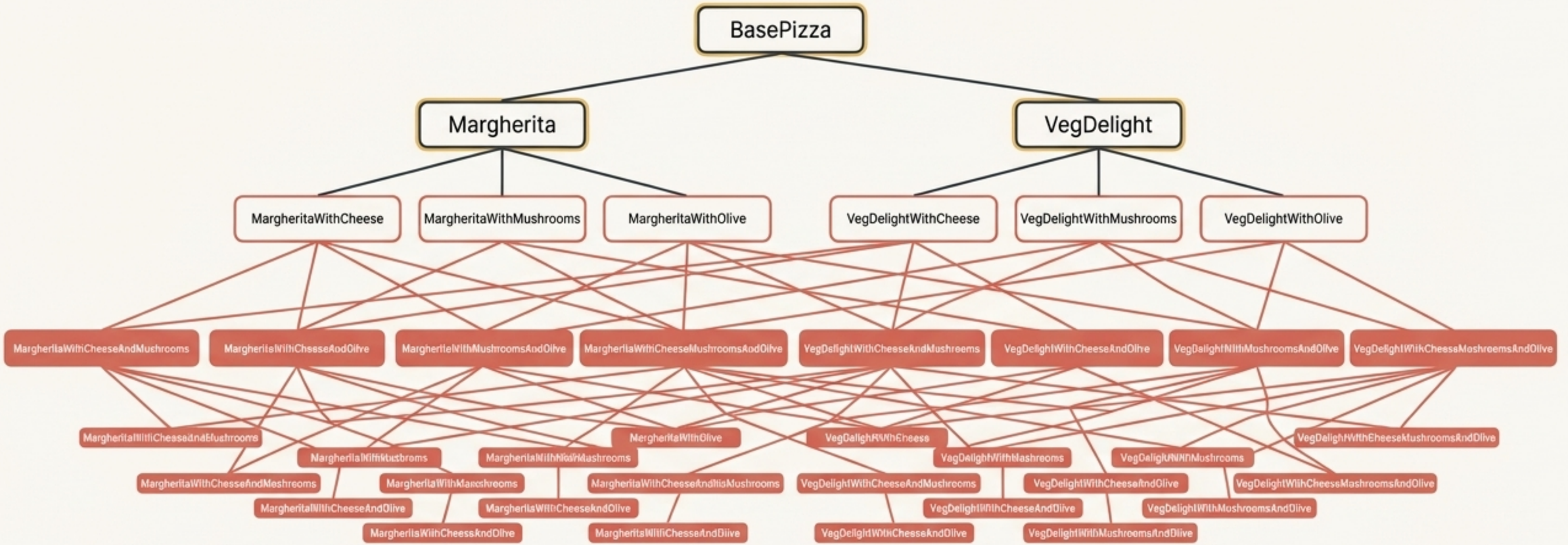
A customer can order any base pizza with any combination of toppings. How should we structure our classes?

Initial Thought

Inheritance seems like a natural fit. A Margherita with Extra Cheese *is a* type of Margherita.

The Problem with Inheritance: The ‘Class Explosion’

If we create a separate class for every possible combination, our system quickly becomes unmanageable.

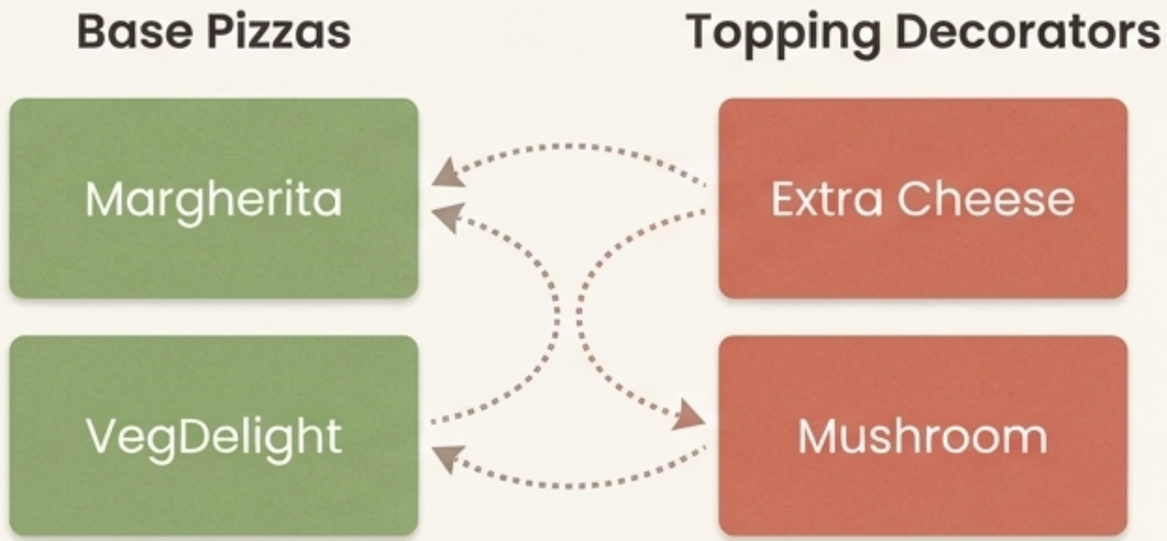


"Imagine adding a new topping. You'd have to create a new class for every existing combination. It becomes very difficult to manage."

The Solution: Decorate, Don’t Subclass

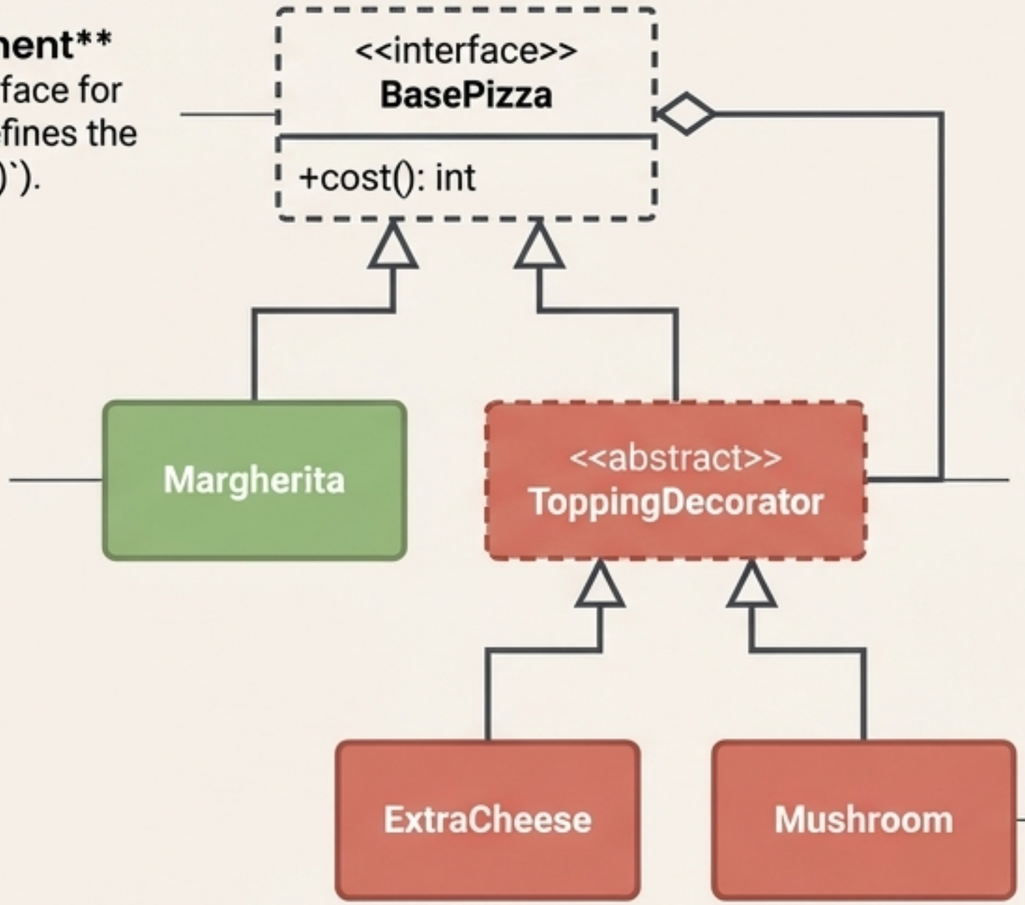
Main Idea: Instead of creating a massive inheritance tree, we can use composition. We’ll have our base pizzas and a separate set of "Topping Decorator” classes.

Goal: Avoid the “Class Explosion” by allowing us to mix and match toppings with any base pizza at runtime. This approach is flexible and scalable.



The Blueprint: Anatomy of the Decorator

1. **Component**
The abstract class or interface for all objects in the chain. Defines the core methods (e.g., `cost()`).



2. **Concrete Component**
A basic object that can be decorated. Implements the Component interface.

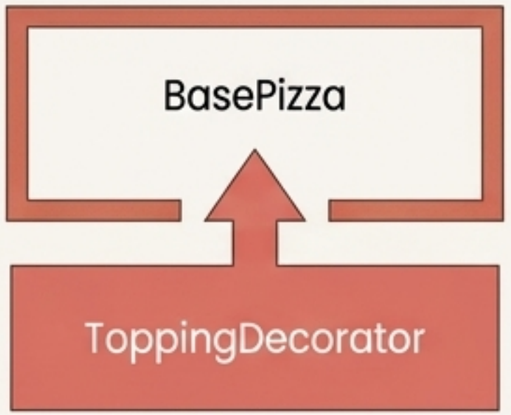
3. **Decorator**
An abstract class that *both extends* the Component (`BasePizza`) and *contains* a reference to a Component (`BasePizza`).

4. **Concrete Decorator**
Adds specific functionality by wrapping a Component.

The Secret Ingredient: The Decorator's Dual Identity

The power of the **ToppingDecorator** comes from two key relationships, allowing for infinite, type-safe layering.

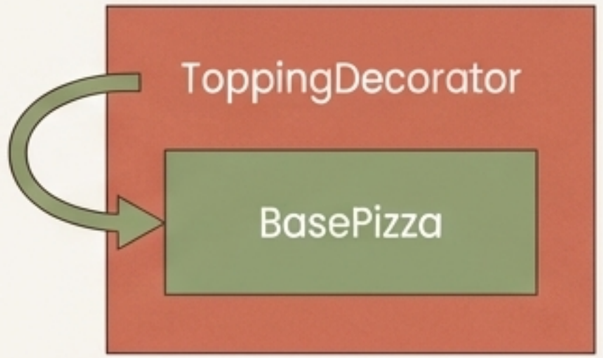
IS-A



`ToppingDecorator` **extends** `BasePizza`.

This means: An `ExtraCheese` decorator can be treated just like a `Margherita`. It **is a** `BasePizza`. This is why you can wrap a decorator with another decorator.

HAS-A



`ToppingDecorator` **contains** a `BasePizza` object.

This means: It holds a reference to the object it is wrapping. This is how it adds its own behavior on top of the wrapped object's behavior.

"The decorator has a base pizza, and it **is** a pizza itself. This is the most complex but most important part of the pattern."

Code: The Foundation (`BasePizza` & `Margherita`)

First, we define our abstract component, `BasePizza`, with an abstract `cost()` method. Then, we create concrete components that provide a base implementation.

```
// The Component
public abstract class BasePizza {
    public abstract int cost();
}

// A Concrete Component
public class Margherita extends BasePizza {
    @Override
    public int cost() {
        return 100; // Base cost of a Margherita pizza
    }
}
```

Code: The Wrappers (`ToppingDecorator` & `ExtraCheese`)

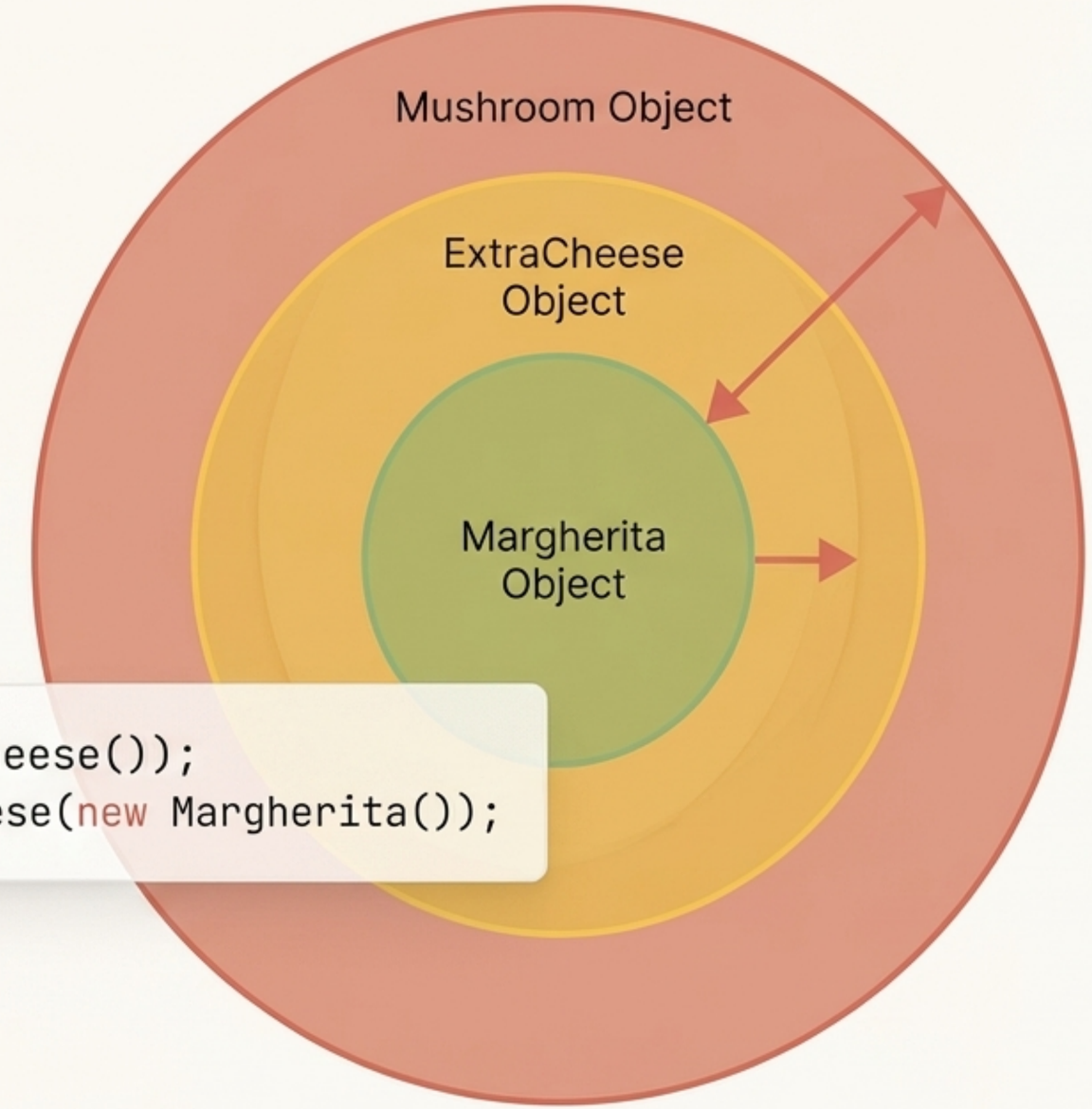
The abstract decorator extends `BasePizza` (**IS-A**) and holds a `BasePizza` instance (**HAS-A**). Concrete decorators implement the logic.


```
// The Abstract Decorator
public abstract class ToppingDecorator extends BasePizza { // <-- IS-A
    // HAS-A relationship
    BasePizza pizza; // <-- HAS-A
}

// A Concrete Decorator
public class ExtraCheese extends ToppingDecorator {
    public ExtraCheese(BasePizza pizza) {
        this.pizza = pizza; // Set the pizza to wrap
    }

    @Override
    public int cost() {
        // Delegate to the wrapped pizza, then add own cost
        return this.pizza.cost() + 10;
    }
}
```

Putting It All Together: Building Our Pizza

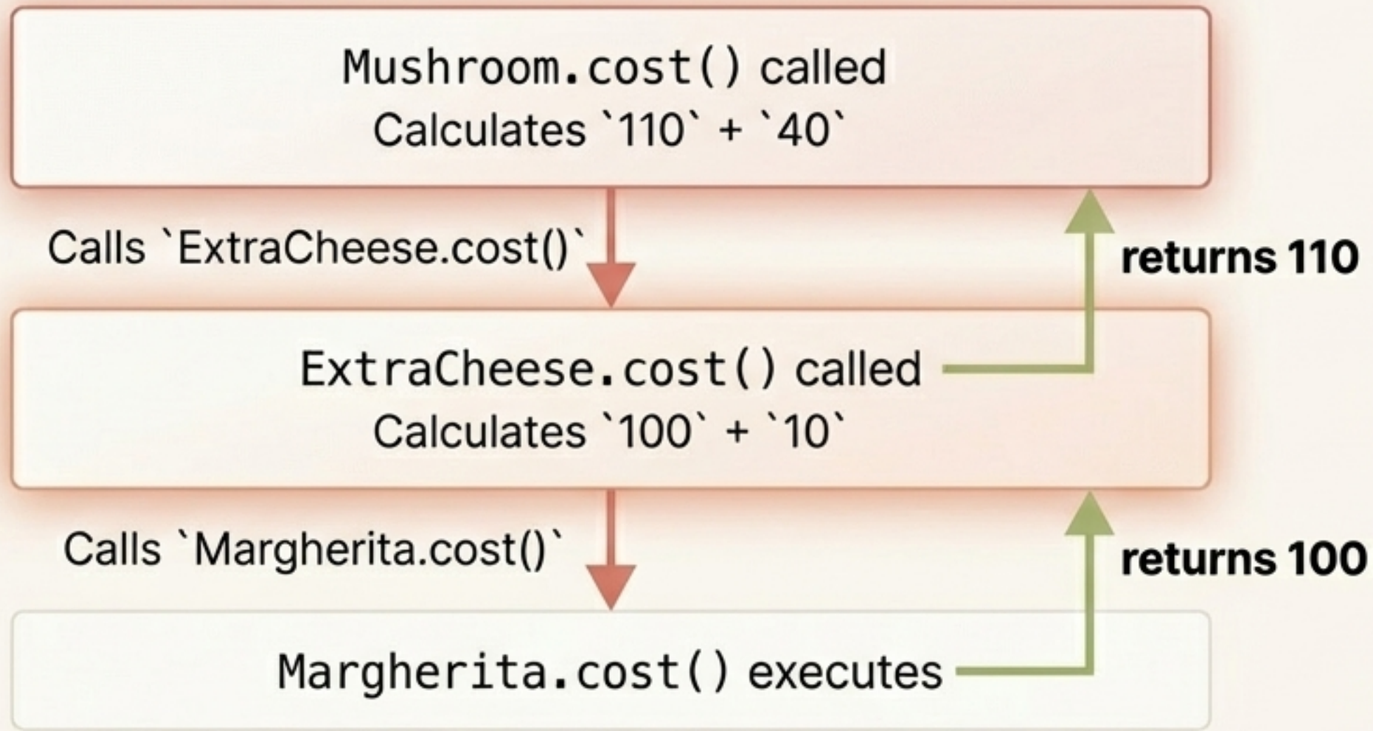
With our classes in place, creating a complex pizza is as simple as nesting constructors. The innermost object is the base, and each layer wraps the one inside it.

```
BasePizza pizza = new Mushroom(new ExtraCheese());
BasePizza pizza = new Mushroom(new ExtraCheese(new Margherita());
```



How the Final Price is Calculated

When we call `pizza.cost()`, the request travels down through each decorator to the base, and the costs are summed on the way back up.



Final Result: 150

When to Use the Decorator Pattern

Use this pattern when:

- You need to add responsibilities to individual objects dynamically and transparently, without affecting other objects.

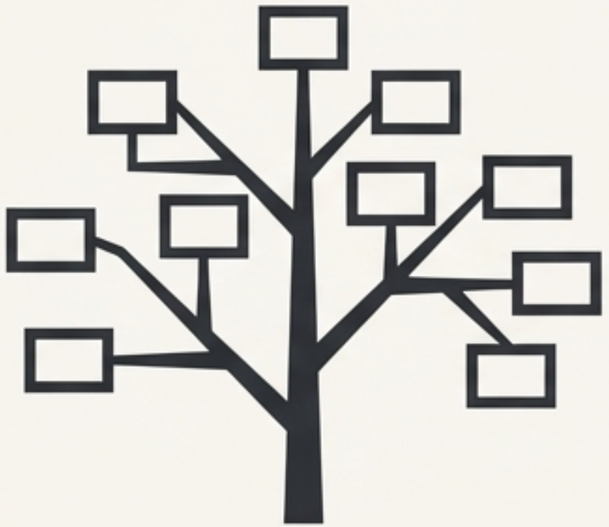
- Subclassing is impractical. Use it to avoid a 'class explosion' from a large number of independent extensions.

- You want responsibilities to be removable. (Though not shown in this example, you can design decorators to be unwrapped).

The Principle: Favor Composition Over Inheritance

The Decorator pattern is a classic demonstration of this core design principle. Instead of building functionality through a rigid family tree of classes (inheritance), we assemble it by combining independent, reusable objects (composition).

Inheritance



Composition

This approach leads to more flexible, maintainable, and scalable systems. You build complex behavior from simple pieces, just like adding toppings to a pizza.