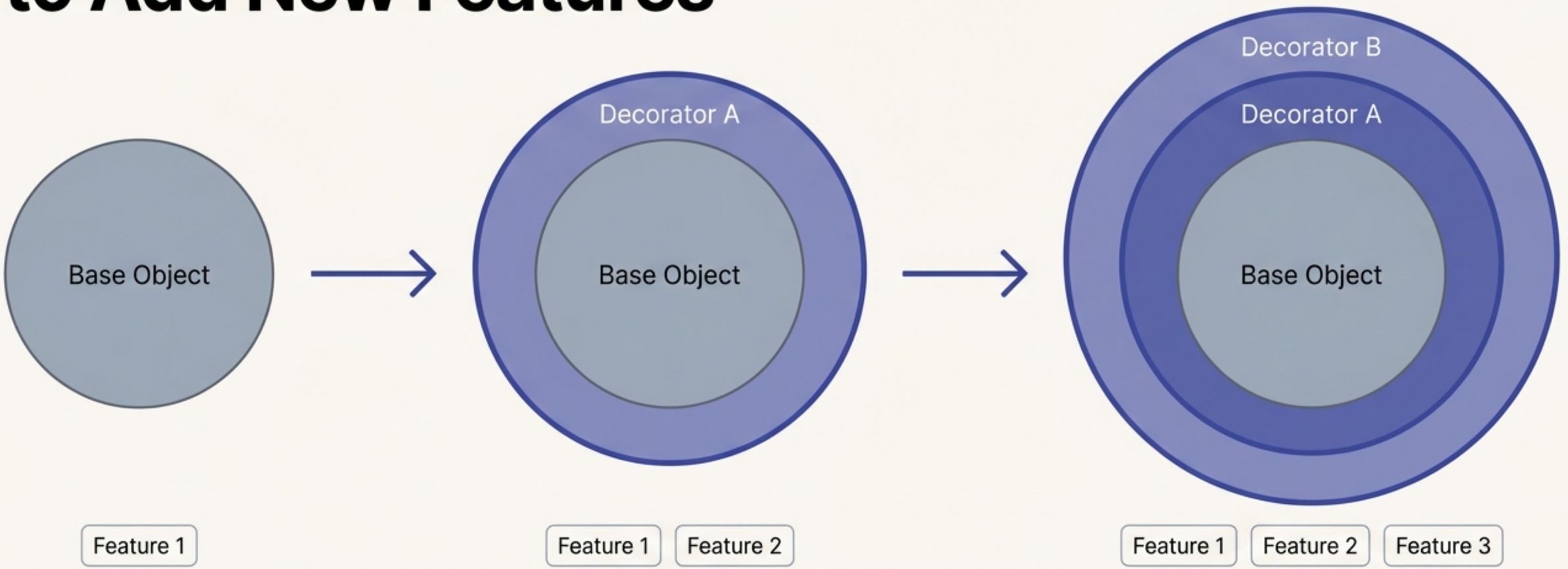
Adding Layers of Functionality: Mastering the Decorator Pattern

A guide to dynamically extending object behaviour without subclassing.

The Core Idea: Wrap Objects to Add New Features



The Decorator Pattern lets us attach new behaviours to objects by placing them inside special 'wrapper' objects. Crucially, the wrapped object and the wrapper are of the same type, allowing for infinite, stackable layers.

It’s Like Customising an Order

The Pizza Shop

You start with a base pizza (e.g., Margherita). You then ‘decorate’ it with toppings like extra cheese, mushrooms, or jalapeños. Each topping adds to the cost and description.

The Coffee Bar

You have a base coffee (e.g., Espresso). You can then add extras like whipped cream, extra milk, or caramel syrup, building your final custom beverage.

The Car Showroom

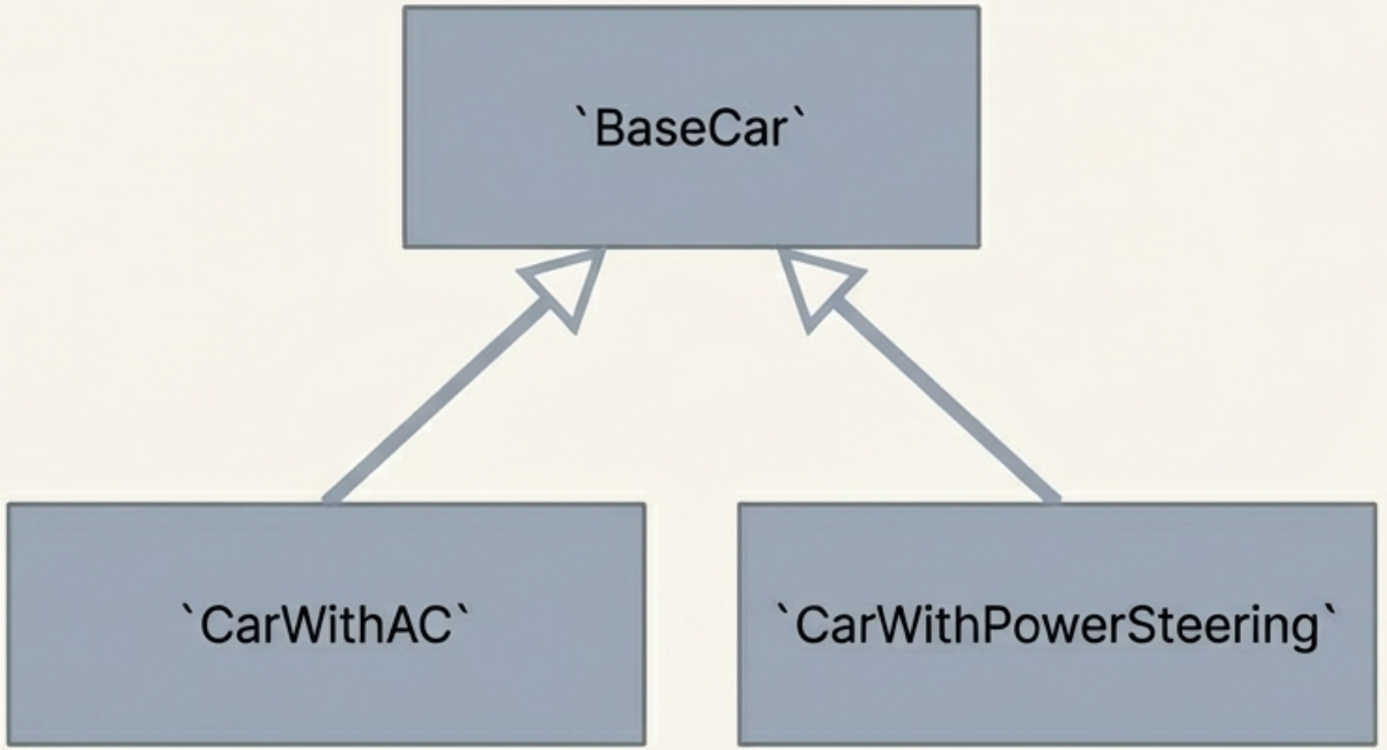
You begin with a base model car. You can then add optional features like air conditioning, power steering, or fog lights to create a customised vehicle.

So, Why Do We Need a Special Pattern for This?

A common first instinct for adding features is to use inheritance.

Let's create a new subclass for every possible combination of features.

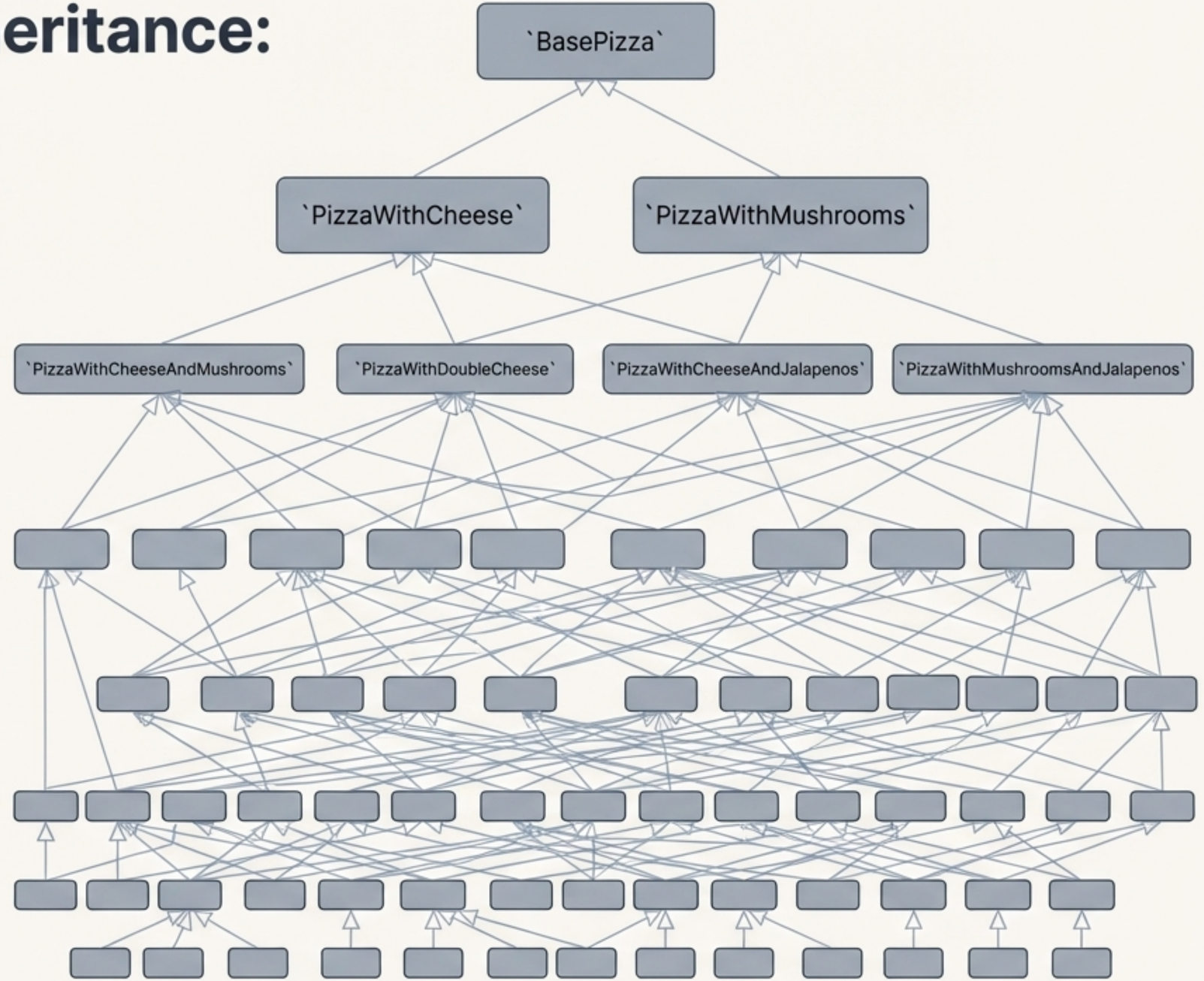
What could go wrong?



The Problem with Inheritance: A ‘Class Explosion’

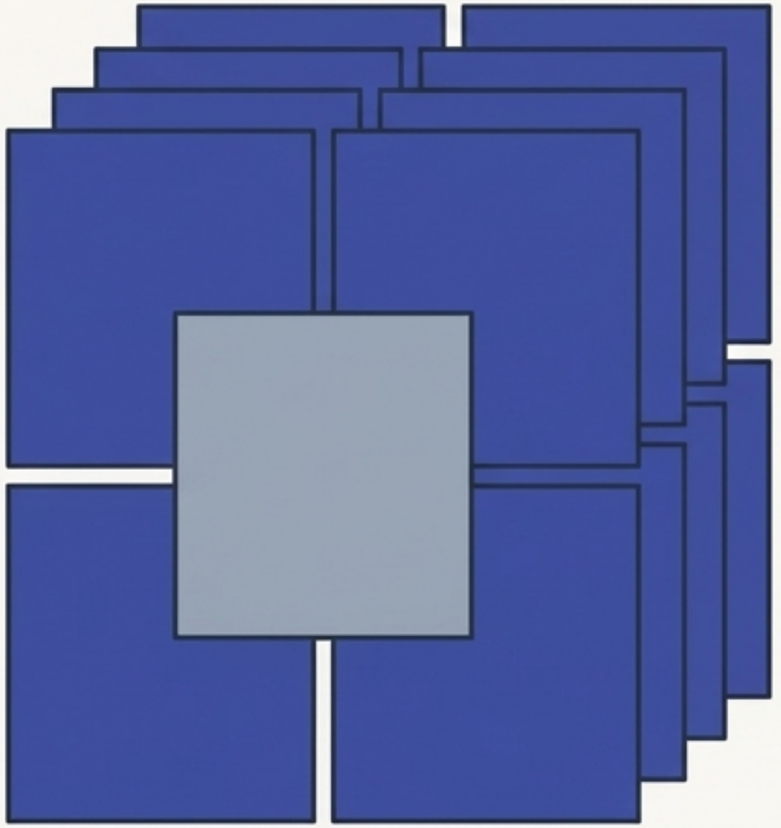
Key Insight: Using inheritance for every feature combination leads to an explosion of classes. The permutations become impossible to manage, especially when new toppings are introduced.

“Ultimately, you’ll have so many classes that managing them becomes very difficult.”



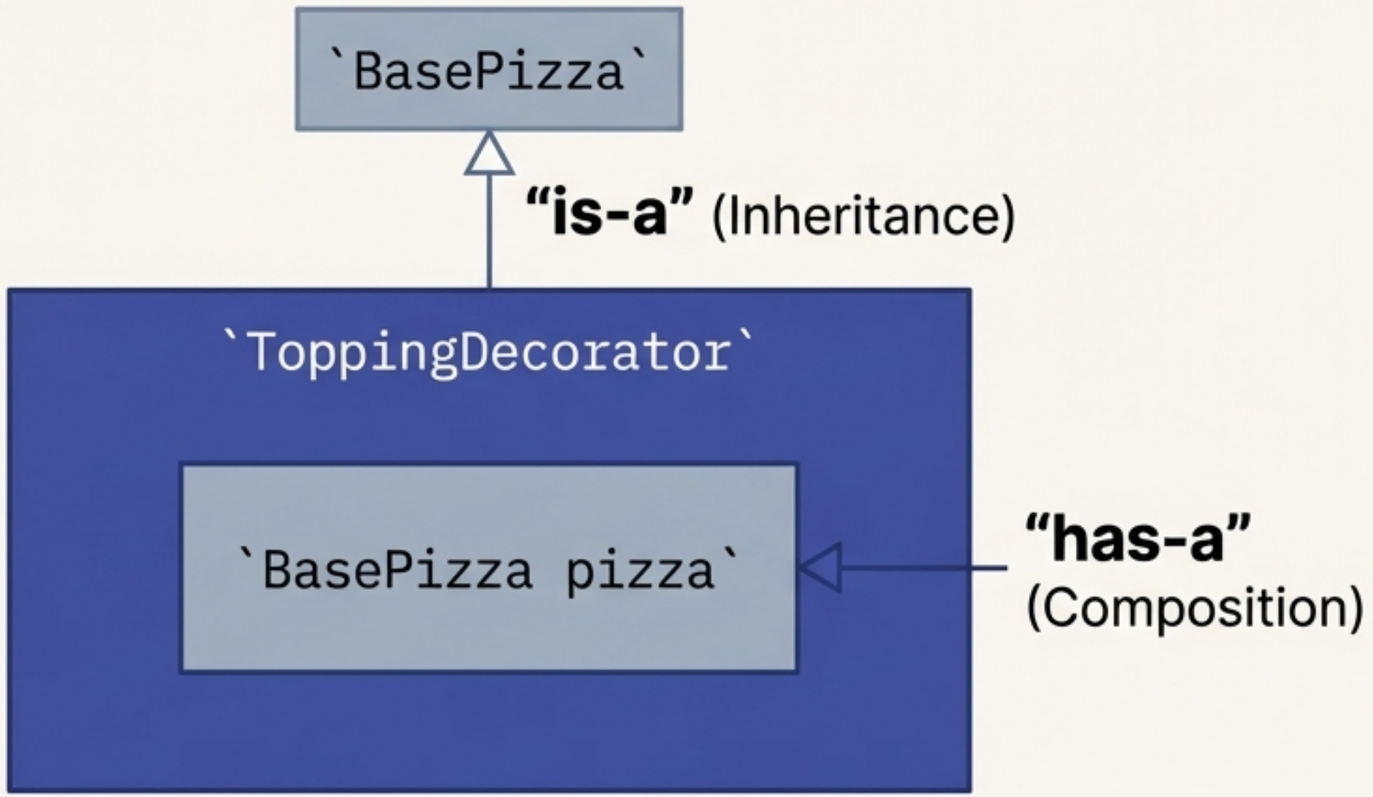
The Solution: Add Features Dynamically at Runtime

The Decorator Pattern avoids this explosion by favouring composition over inheritance. Instead of creating a new class for each feature set, you build objects by wrapping a base component with decorators at runtime.



This approach is crucial when you have a **stable base object** but require a multitude of optional, combinable features.

The Mechanic: How Decorators Use 'Is-A' and 'Has-A'



Is-A

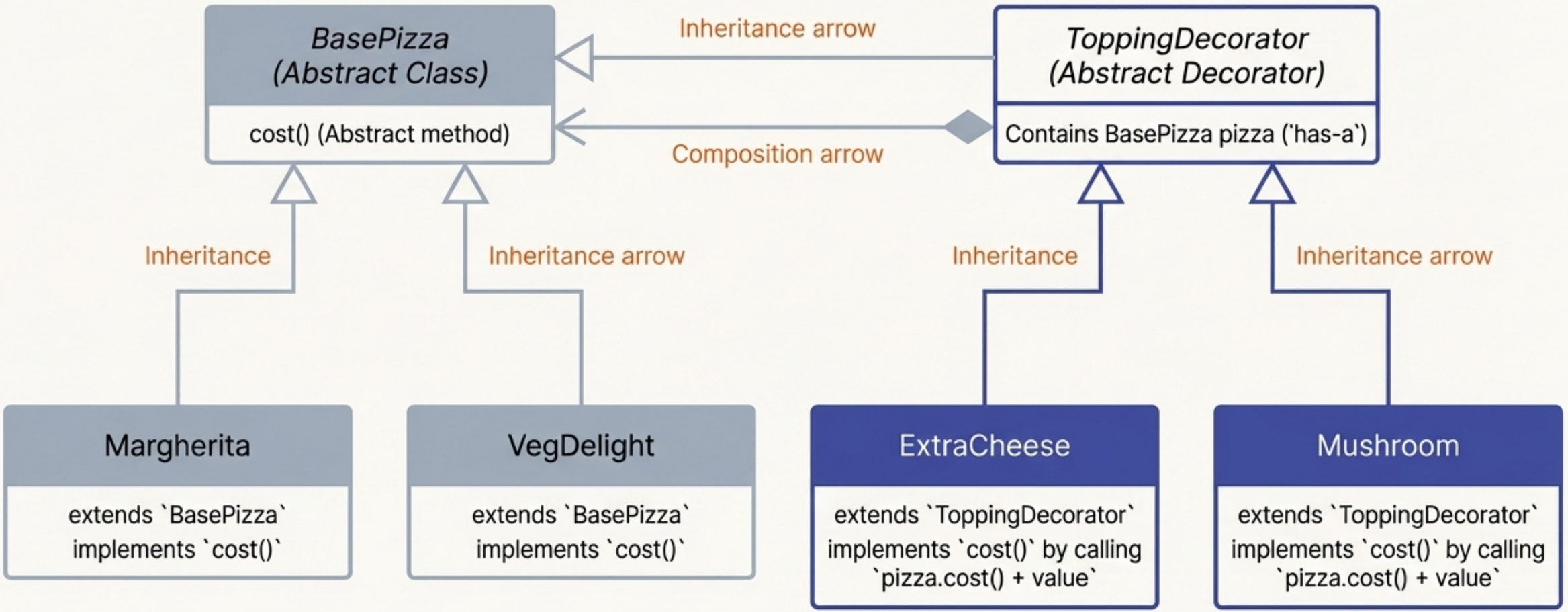
The decorator is the same type as the object it decorates. This means a `ToppingDecorator` *is a* `BasePizza`. This allows you to use a decorated pizza anywhere you would use a base pizza.

Has-A

The decorator holds a reference to the object it wraps. This allows it to delegate calls to the wrapped object and add its own behaviour before or after.

“This is the most complex part of the Decorator pattern. The decorator `has-a` base object, and it itself `is-a` base object.”

The Structural Blueprint



The Code: Step 1 - Define the Base Component

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

// Concrete components
public class Margherita extends BasePizza {
    @Override
    public int cost() { return 100; }
}

public class VegDelight extends BasePizza {
    @Override
    public int cost() { return 120; }
}
```

We start with an abstract `BasePizza` class that defines the common interface (`cost()`). Concrete classes like `Margherita` provide the base objects we will decorate.

The Code: Step 2 - Create the Abstract Decorator

```
// The abstract decorator
public abstract class ToppingDecorator extends BasePizza {
    // has-a relationship
    BasePizza pizza;
}
```

This establishes the **`is-a`** relationship. The decorator is a type of pizza.

This establishes the **`has-a`** relationship. The decorator holds a reference to the pizza it will wrap.

The Code: Step 3 - Implement Concrete Decorators

```
public class ExtraCheese extends ToppingDecorator {
    public ExtraCheese(BasePizza pizza) {
        this.pizza = pizza;
    }
    @Override
    public int cost() {
        return this.pizza.cost() + 10;
    }
}

public class Mushroom extends ToppingDecorator {
    public Mushroom(BasePizza pizza) {
        this.pizza = pizza;
    }
    @Override
    public int cost() {
        return this.pizza.cost() + 40;
    }
}
```


The constructor takes the pizza to be wrapped. The `cost()` method first calls the wrapped pizza's `cost()` and then adds its own value. This is how behaviour is 'layered' on.

Putting It All Together: Building a Custom Pizza

To create a Margherita pizza with extra cheese and mushrooms, we simply wrap the objects one inside the other.

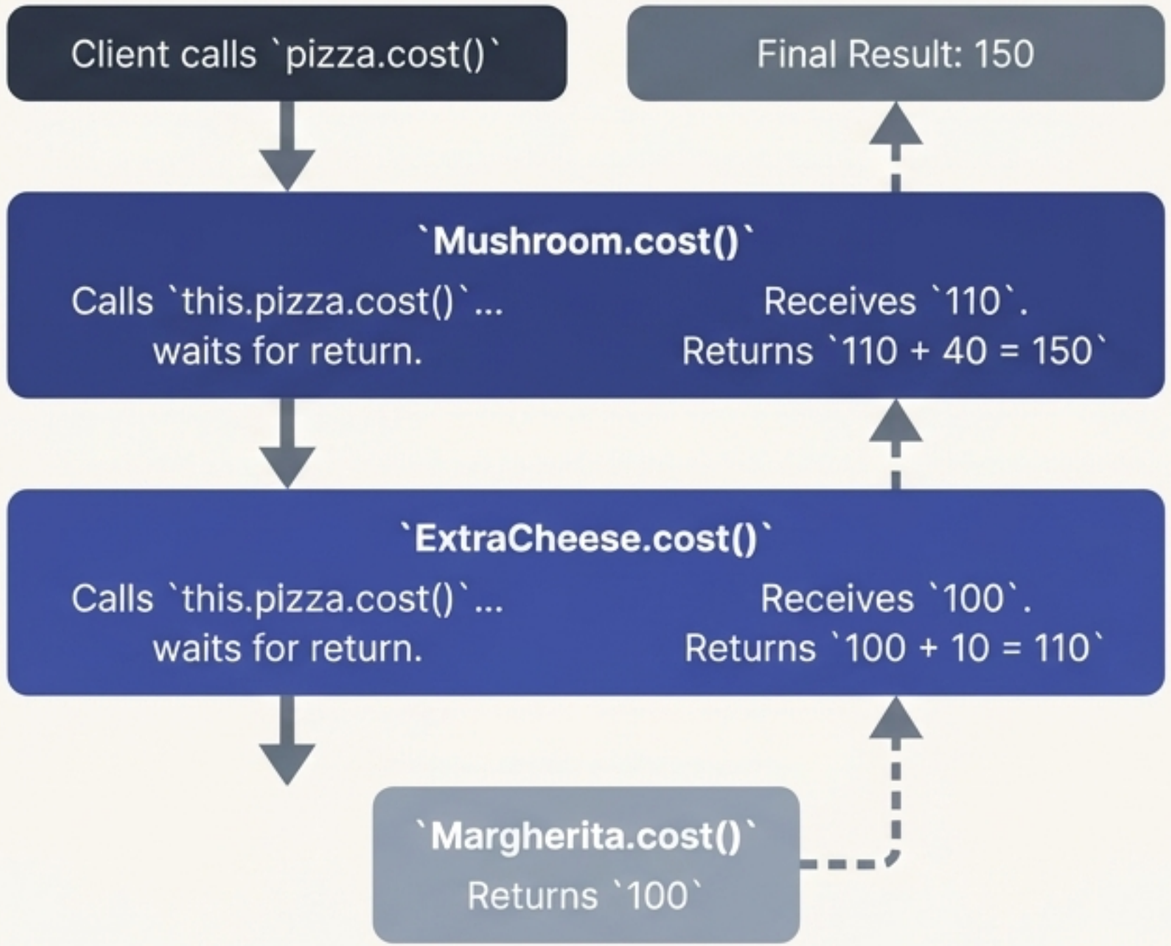
```
// Create a base Margherita pizza
BasePizza pizza = new Margherita();

// Decorate it with Extra Cheese
pizza = new ExtraCheese(pizza);

// Decorate it again with Mushroom
pizza = new Mushroom(pizza);

System.out.println("Cost: " + pizza.cost());
```

Tracing the Call to `cost()`



The request is passed down to the base component, and the results bubble back up, with each decorator adding its own modification along the way.

When Should You Use the Decorator Pattern?

When you need to add responsibilities to individual objects dynamically and transparently, without affecting other objects.

To avoid a 'class explosion' caused by the need for numerous subclasses to handle feature combinations.

When a class definition is hidden or otherwise unavailable for subclassing.

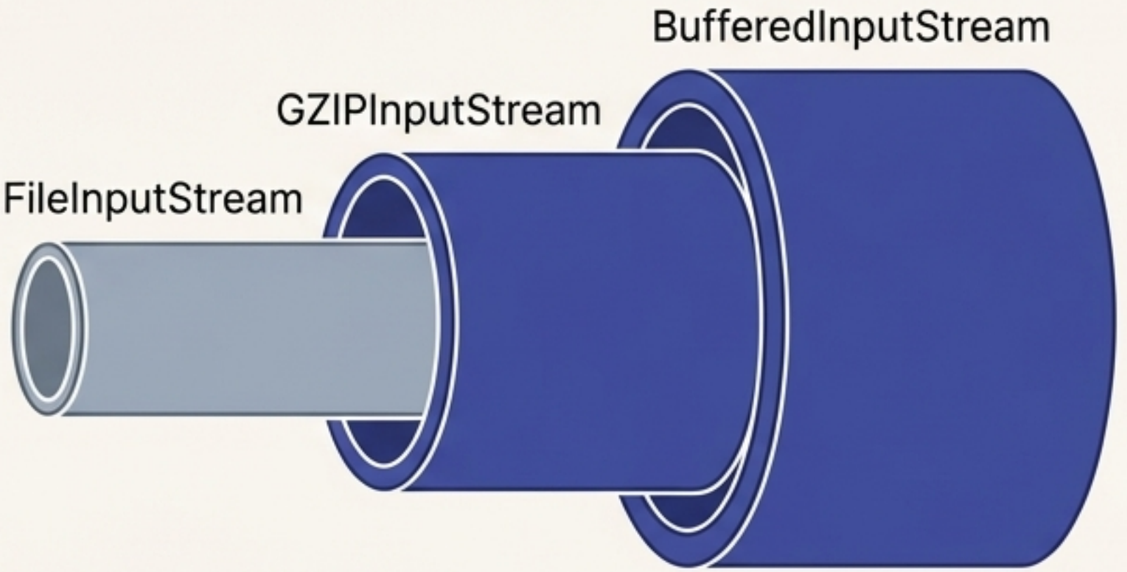
Use it when your base component is stable, but you need to add many optional, combinable layers of functionality.

The Decorator Pattern in the Wild

This pattern is not just theoretical; it's a widely used and powerful tool for building flexible systems.

Common Interview Questions

A frequent subject of system design interviews, particularly for tasks like 'Design a coffee machine' or 'Model a pizza shop'.



Java I/O Libraries

The classic example is Java's `java.io` package, where you wrap streams with decorators like `BufferedReader` or `GZIPInputStream` to add functionality like buffering or compression.

Mastering the Decorator Pattern means you can build complex objects from simple pieces, creating systems that are both powerful and easy to maintain.