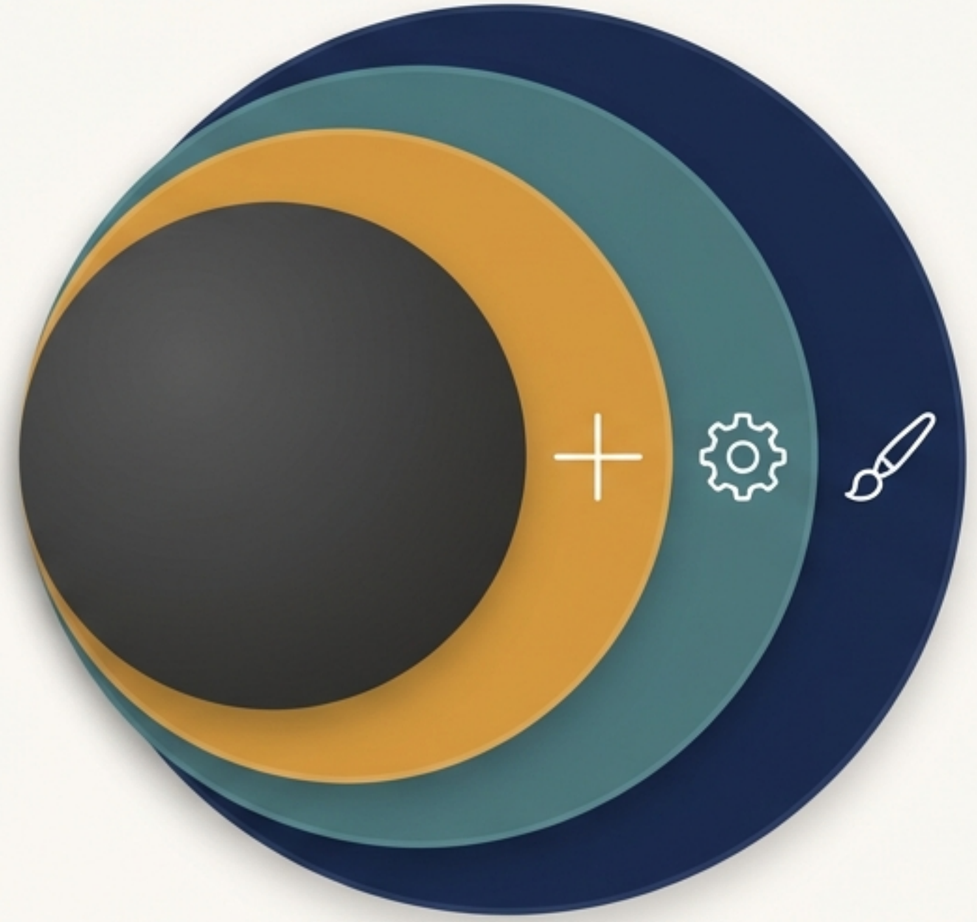
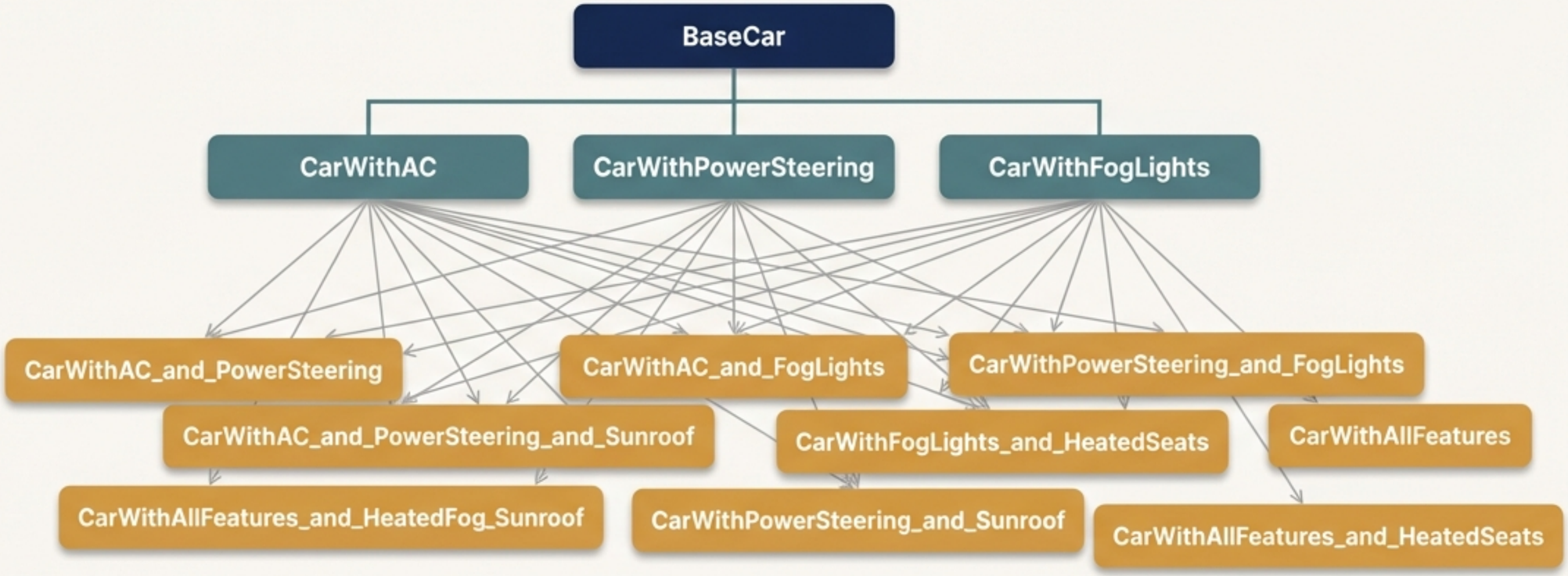
The Decorator Pattern: Adding Features Without the Chaos

A guide to dynamically extending object behaviour without creating a complex web of subclasses.



The Problem: The Inflexible World of Subclassing

Adding features through inheritance seems logical at first, but it leads to a “class explosion” for every possible combination.

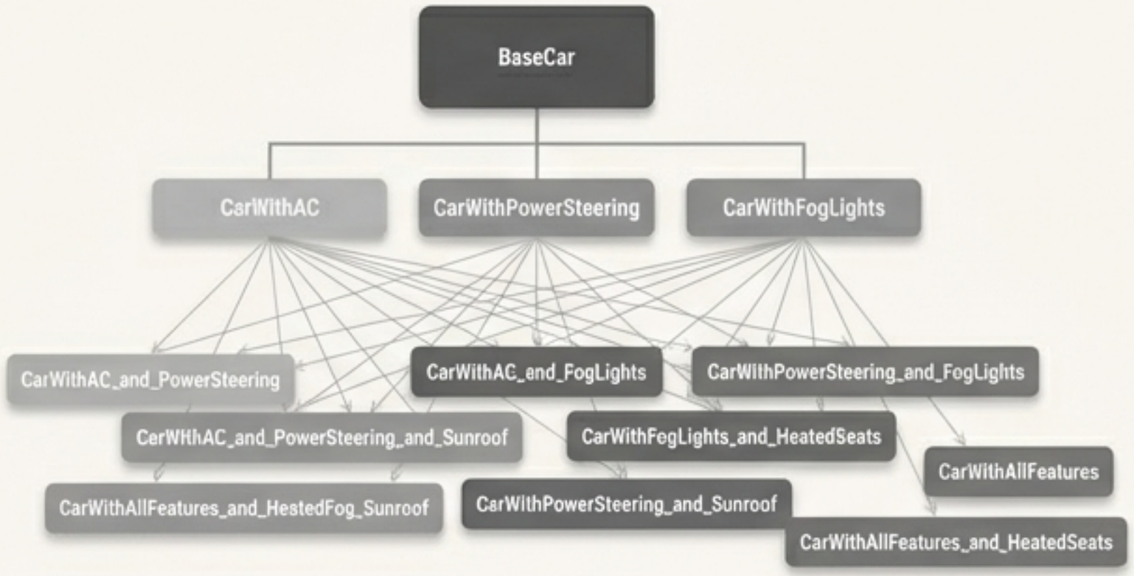


This approach is rigid. What happens when a new feature is introduced? You have to create a whole new set of subclasses.

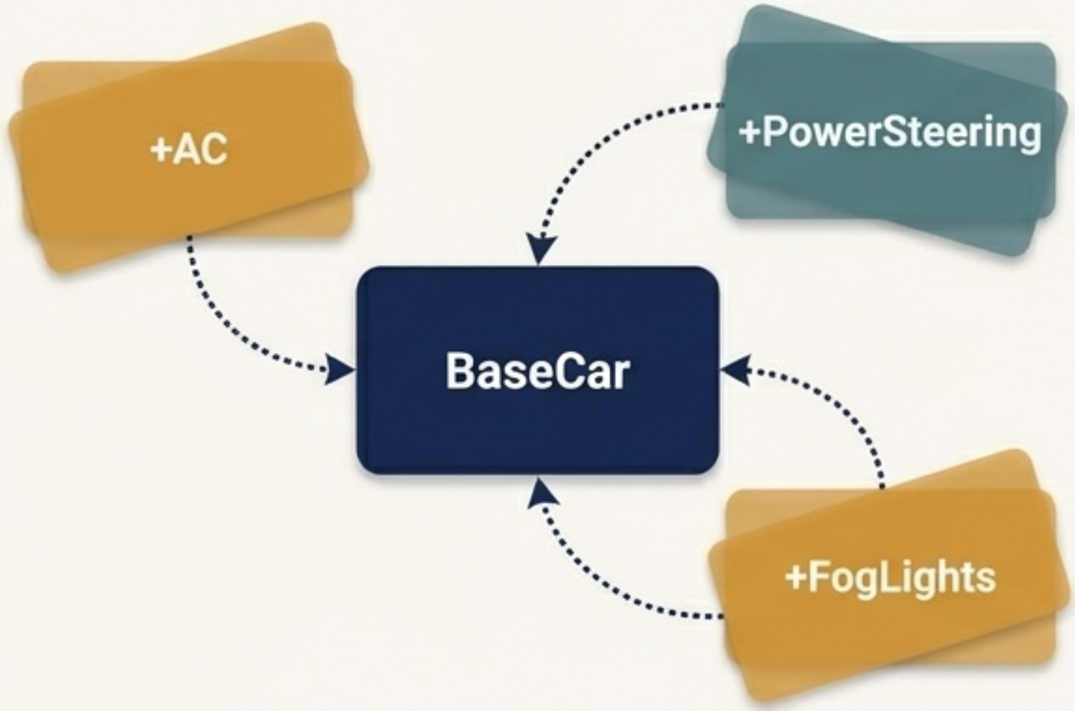
The Insight: Wrap Objects Instead of Subclassing Them

The Decorator Pattern lets us add new behaviour to objects by placing them inside special wrapper objects that contain the behaviour.

The Old Way: Rigid Hierarchy



The Decorator Way: Flexible Layers



An Intuitive Analogy: Assembling a Pizza

Let’s explore this pattern using a simple, real-world example. A pizza base is still a pizza, no matter how many toppings you add. Each topping is a ‘decorator’ that adds flavour (and cost) without changing the fundamental nature of the pizza.

ConcreteComponent: Margherita

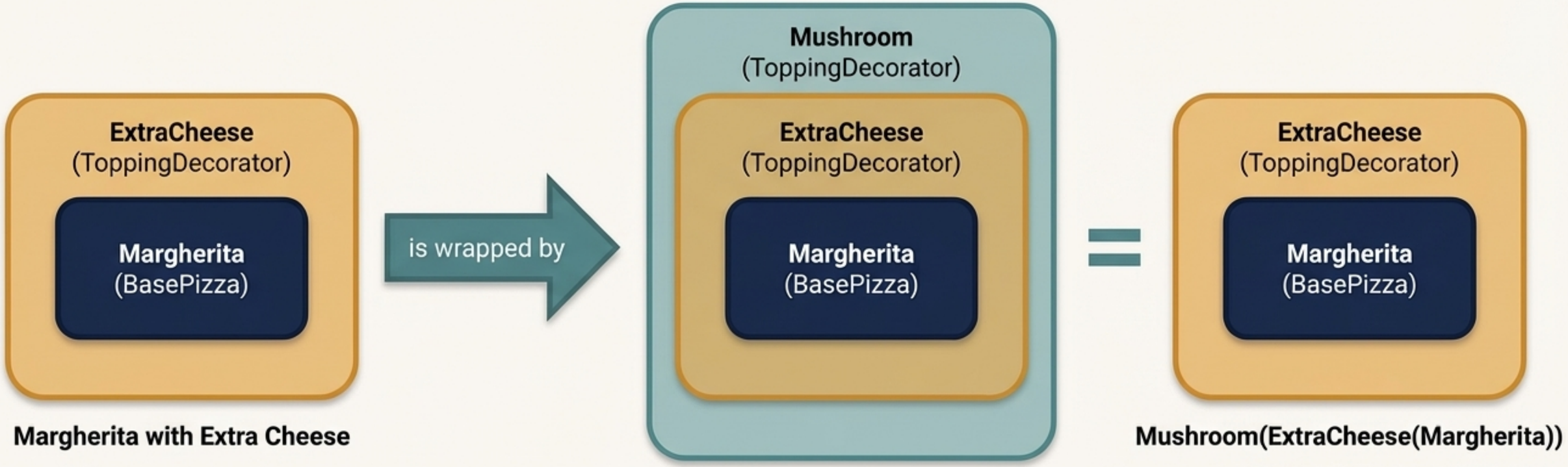
Step 1: Adding the First Topping

We start with our base `Margherita` pizza. To add extra cheese, we don't change the `Margherita` class. Instead, we wrap the `Margherita` object in an `ExtraCheese` decorator.

...which is still of type BasePizza

Step 2: Stacking More Layers

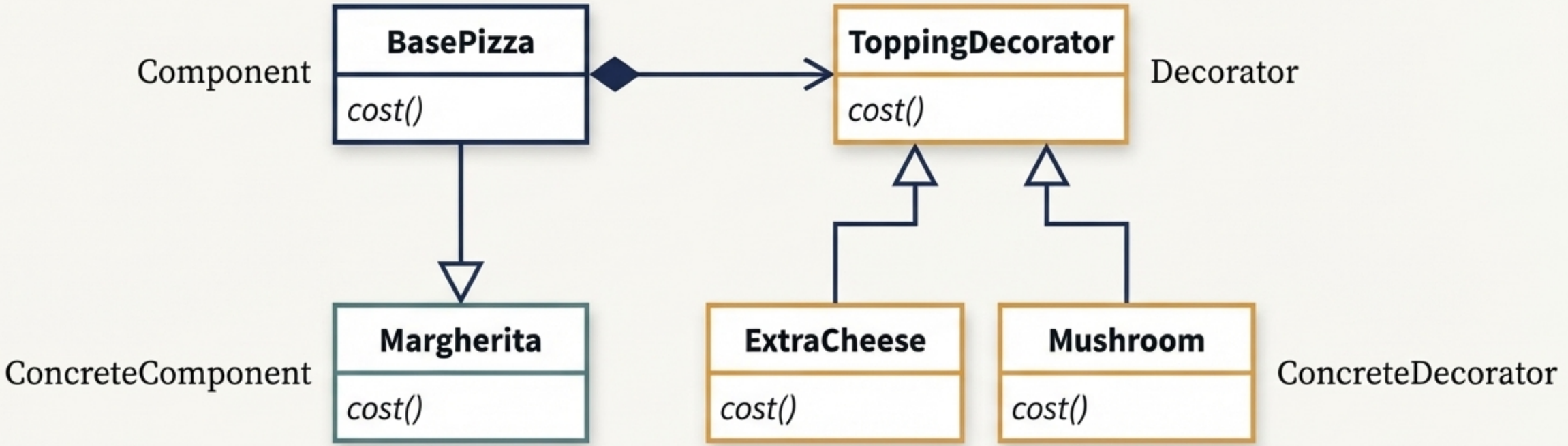
The beauty of the pattern is that a decorated object can be decorated again. Our `Margherita with Extra Cheese` is still a `BasePizza`, so we can wrap it in a `Mushroom` decorator.



You can continue adding layers as needed. The core object is unaware of the decorators surrounding it.

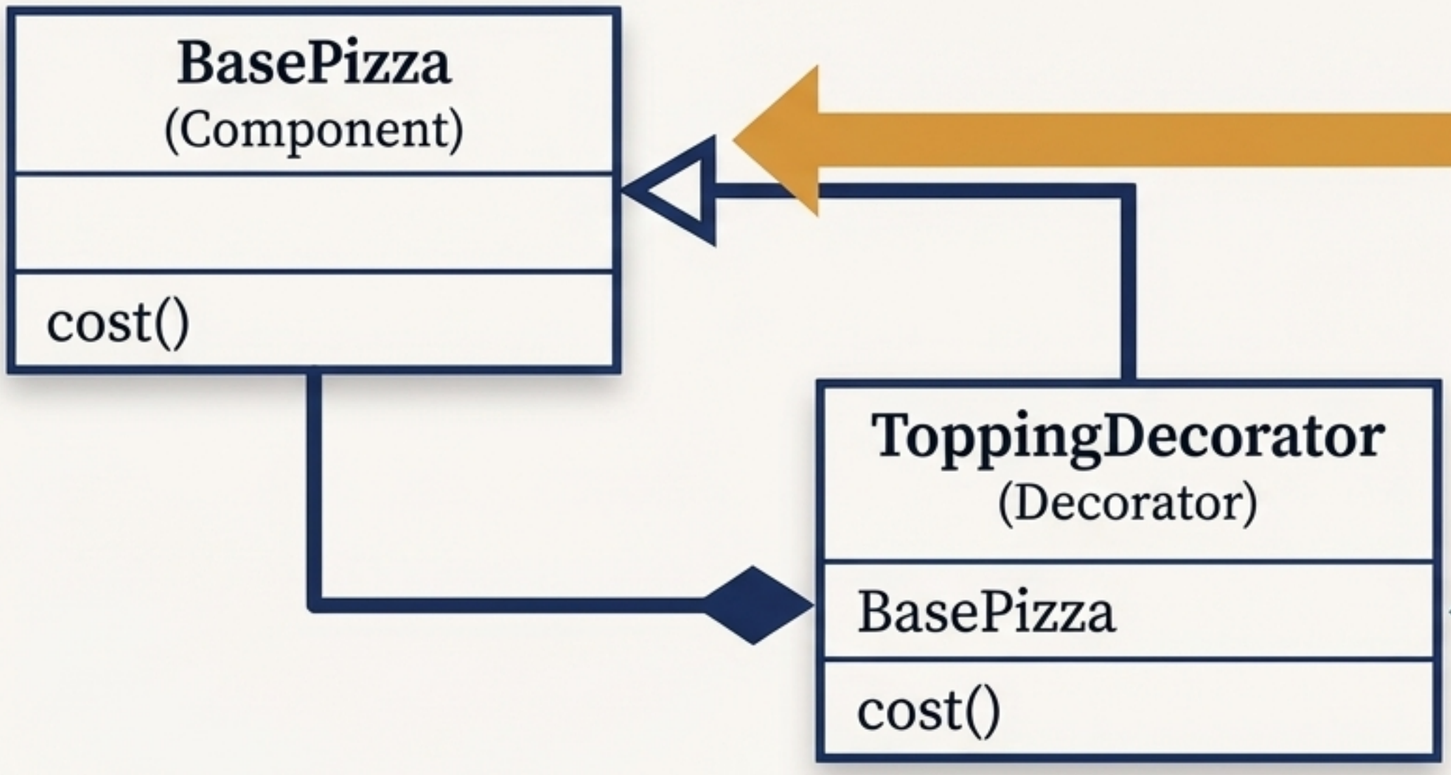
The Pattern’s Formal Blueprint

The pizza analogy maps directly to a formal software structure with four key participants.



The Core Mechanism: A Union of `IS-A` and `HAS-A`

The pattern's power comes from a specific combination of two fundamental object-oriented relationships.



`IS-A`

The `ToppingDecorator` *is a* `BasePizza`. This allows us to use a decorated pizza anywhere a `BasePizza` is expected. It's why we can keep wrapping.

`HAS-A`

The `ToppingDecorator` *has a* `BasePizza`. This is the link to the object it wraps, allowing it to delegate calls to the next layer in the chain.

The Code: Defining the Base Components

```
// The Component
// Defines the interface for objects that can have
// responsibilities added to them dynamically.
public abstract class BasePizza {
    public abstract int cost();
}
```

```
// A Concrete Component
// An object to which additional responsibilities
// can be attached.
public class Margherita extends BasePizza {
    @Override
    public int cost() {
        return 100;
    }
}
```

The Code: Building the Decorators

```java
// The abstract Decorator
public abstract class ToppingDecorator extends BasePizza {
    BasePizza pizza;
}
```

'IS-A": Conforms to the Component interface.

HAS-A': Holds a reference to a Component.

```java
// A Concrete Decorator
public class ExtraCheese extends ToppingDecorator {
    public ExtraCheese(BasePizza pizza) {
        this.pizza = pizza;
    }

    @Override
    public int cost() {
        return this.pizza.cost() + 10;
    }
}
```

Delegates to the wrapped object, then adds its own value.

Assembling Our Decorated Pizza in a Single Line

To create a Margherita with extra cheese and mushrooms, the client code nests the constructors. The resulting object can be treated as a simple BasePizza.

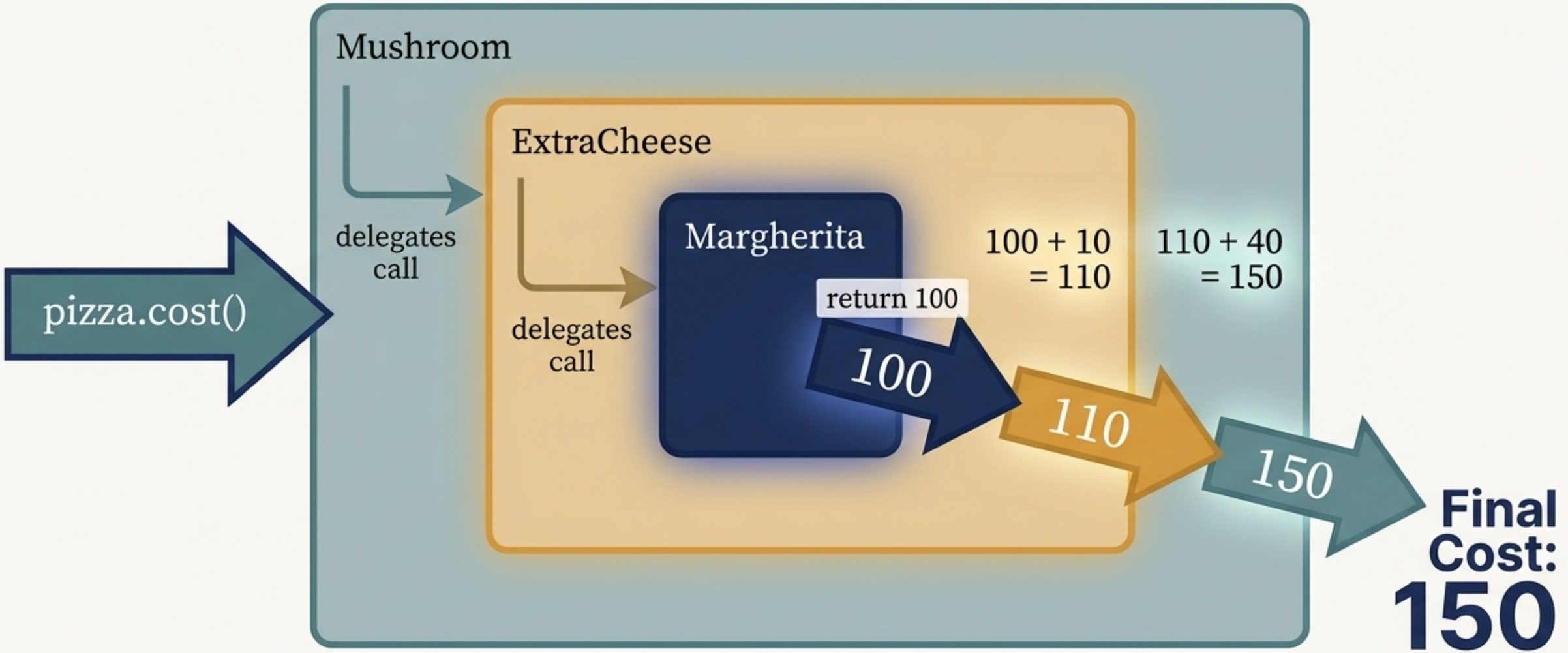
```
// 1. Create a base Margherita
// 2. Wrap it with Extra Cheese
// 3. Wrap the result with Mushroom

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

int totalCost = pizza.cost();
```

How is the final cost calculated?

Tracing the `cost()` Call



The Payoff: Flexible, Maintainable Code

Avoids Class Explosion

Provides a flexible alternative to subclassing for extending functionality. Solves our initial problem.

Dynamic and Composable Behaviour

Responsibilities can be added and removed at runtime by simply adding or removing wrappers.

Adheres to the Open/Closed Principle

You can introduce new decorators (extending functionality) without changing existing component classes (which are closed for modification).

Single Responsibility

Functionality is broken down into smaller, manageable classes (each decorator has one job).

The Decorator Pattern in the Wild

Once you understand the structure, you start seeing this versatile pattern everywhere.

Coffee Shops

A base coffee (`Espresso`) is decorated with cream, syrup, or extra shots.

Vehicle Customisation

A base car is decorated with optional features like Air Conditioning or Power Steering.

GUI Toolkits

A base window component can be decorated with borders, scrollbars, or shadows.

Java I/O Streams

The classic library example: `new BufferedReader(new InputStreamReader(System.in))` wraps a basic stream with buffering capabilities.

A Key Pattern for System Design Interviews

Scenarios like “Design a coffee machine” or ‘Model a pizza ordering service’ are common interview questions designed to test your knowledge of scalable design.

Demonstrating your understanding of the Decorator Pattern shows you can design flexible systems that are open to extension and avoid the rigidity of deep inheritance hierarchies. It is a powerful tool in any software architect’s toolkit.