# The Decorator Pattern

Or: a lesson in the Open/Closed principle

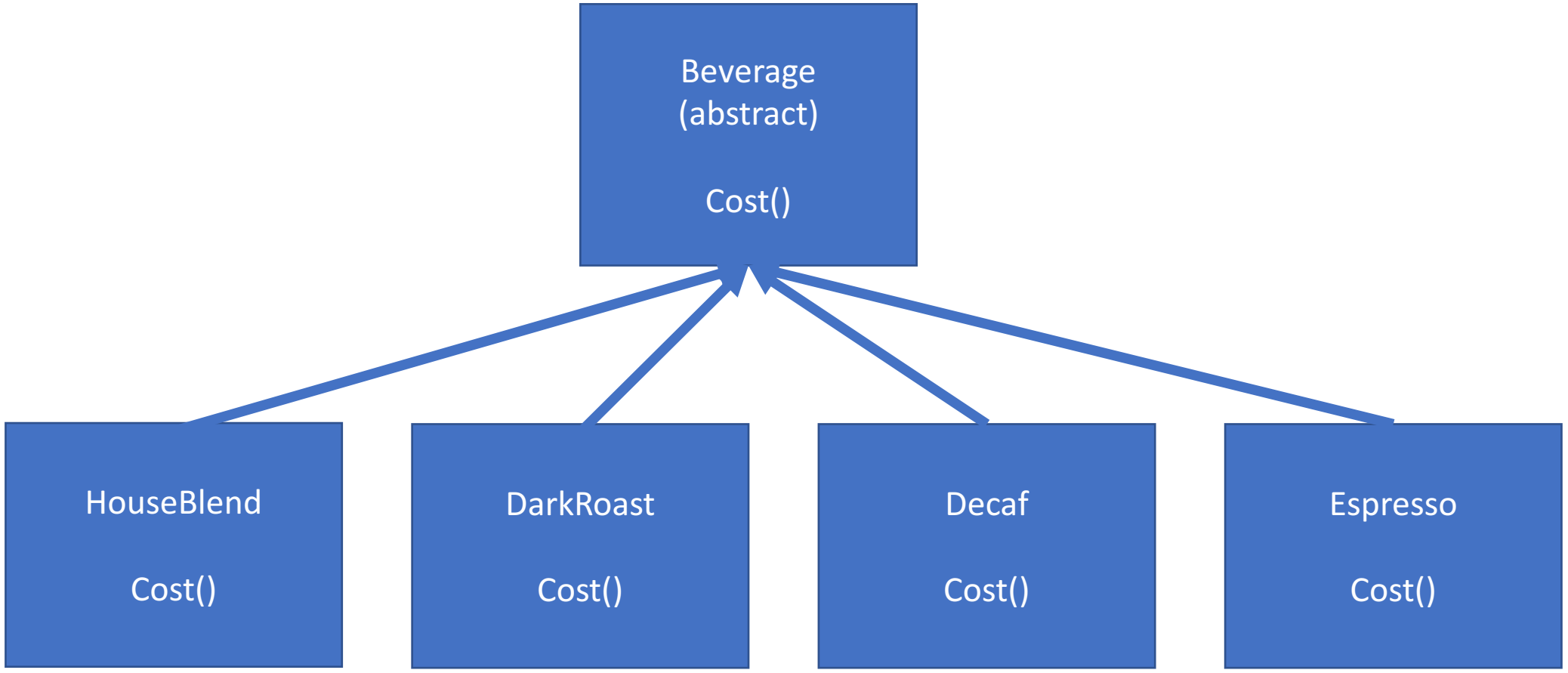
# How many times have we heard this phrase:

*“Favor composition over inheritance”*

*~some OOP guy who is smarter than I am*

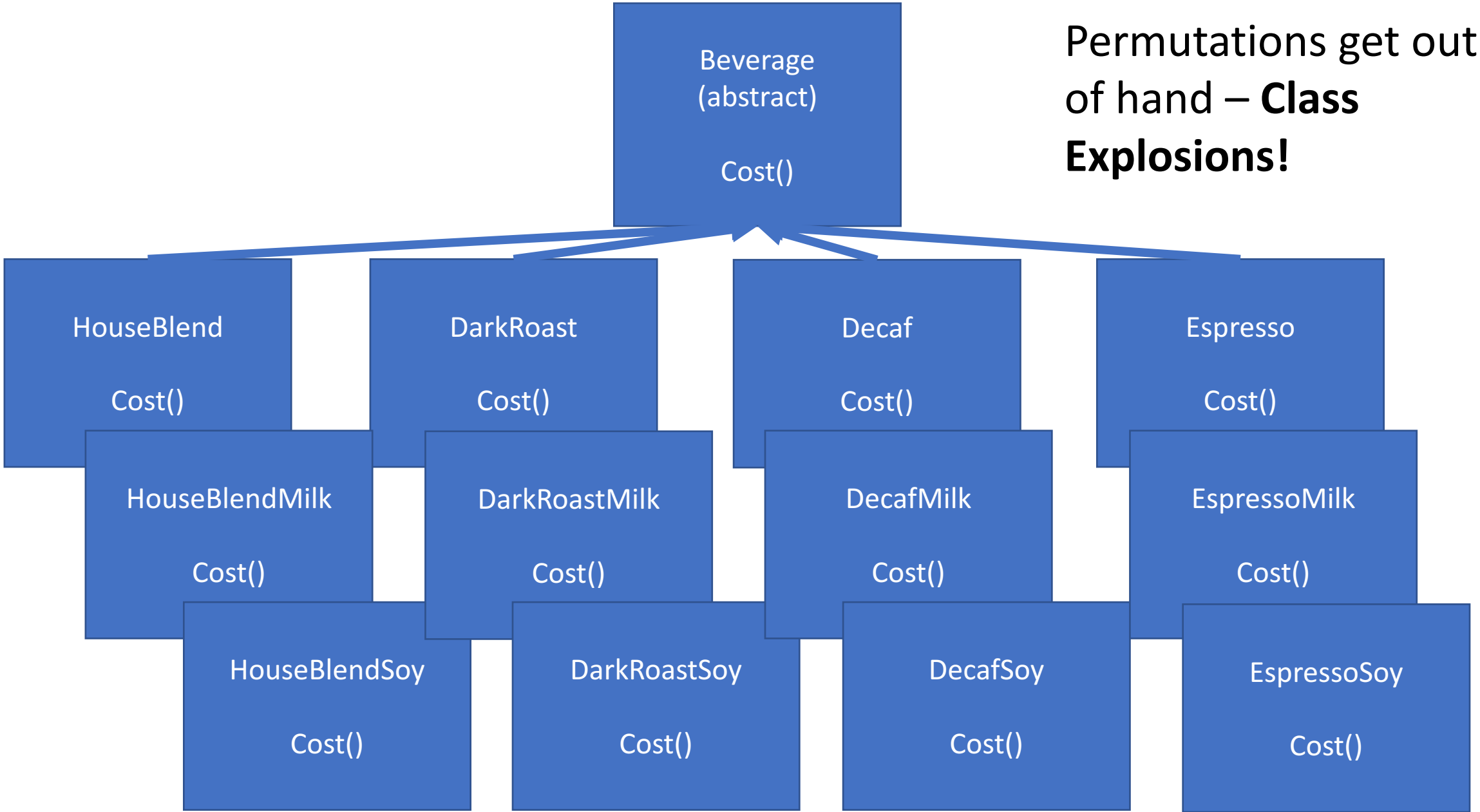
The **decorator pattern** is another example of how composition can lead to more flexible and ultimately more maintainable class design.

# Coffee Shop: Bad Class Design



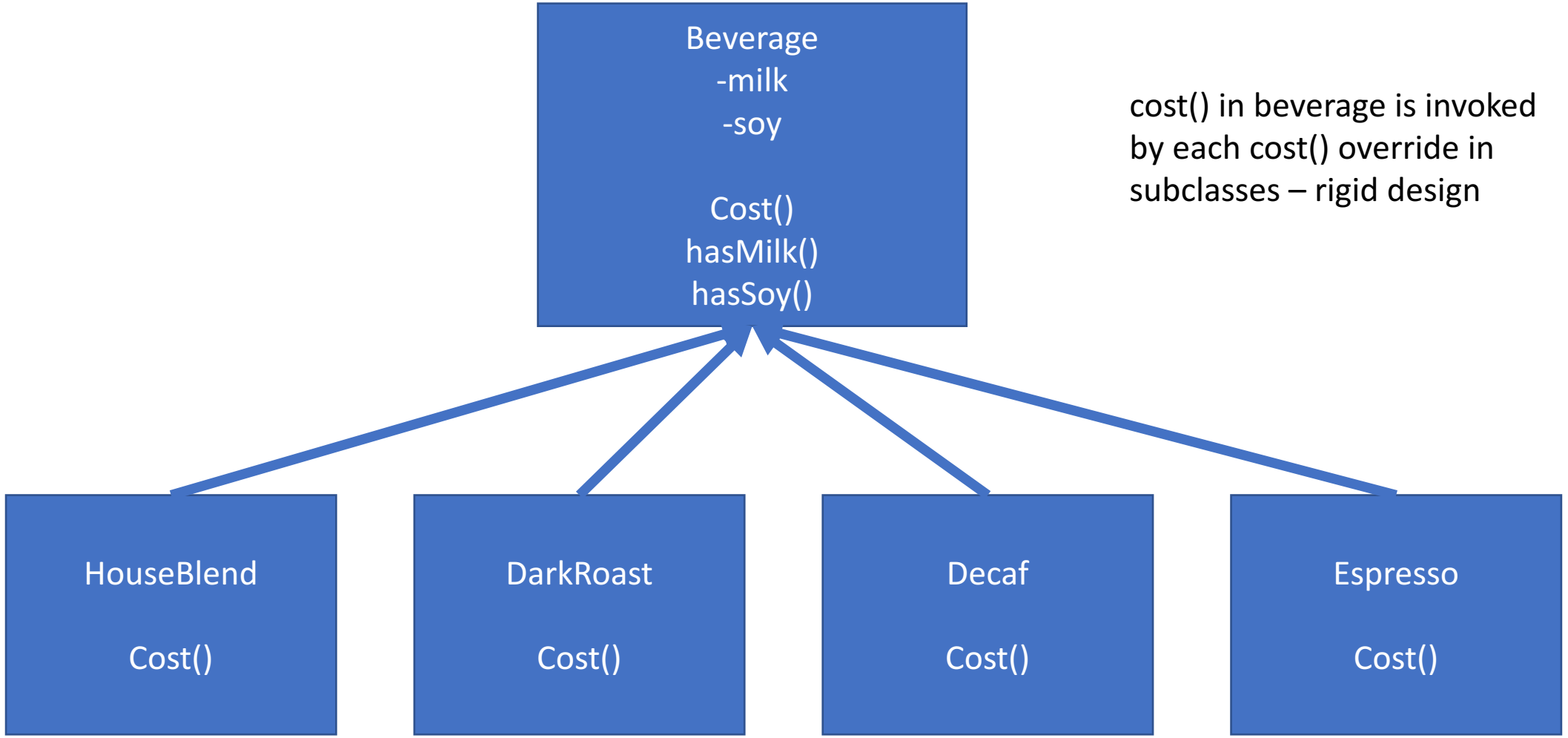

What happens if we want add-ons like milk / soy milk / mocha?

# Coffee Shop: Bad Class Design




Permutations get out of hand – **Class Explosions!**

# Coffee Shop: Better Class Design




cost() in beverage is invoked by each cost() override in subclasses – rigid design

# Why is the last design still not optimal?

- Price changes for add-ons force us to alter beverage class (existing code)
- New add-ons force us to alter the cost calculation method in the superclass
- New beverages will inherit condiment methods that might not be appropriate (i.e. why does Iced tea need mocha?)
- What if you want more than one pump of mocha?

# The Open/Closed Principle

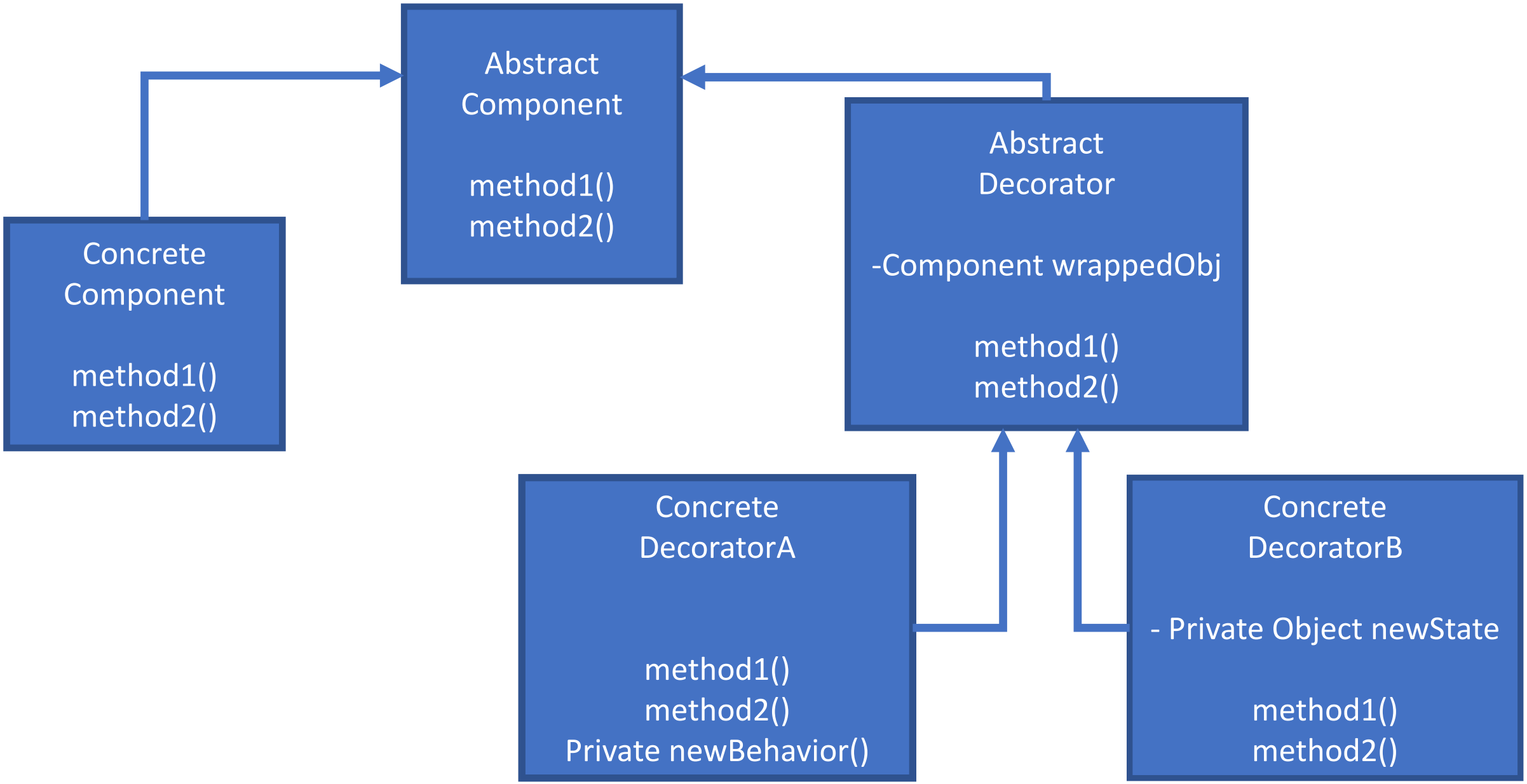
*“Classes should be open for extension,*
*yet closed for modification”*

*~Also some guy who is smarter than I am*

Goal is to allow classes to be easily extended to **incorporate new behavior**, without modifying existing code. The benefits of this are designs that are both **resilient to change and flexible enough to meet changing requirements.**

# Decorator to the Rescue!

# The Decorator Pattern: Class Diagram



# Applying the Decorator Pattern

Say we have a **Pizza Shop**:

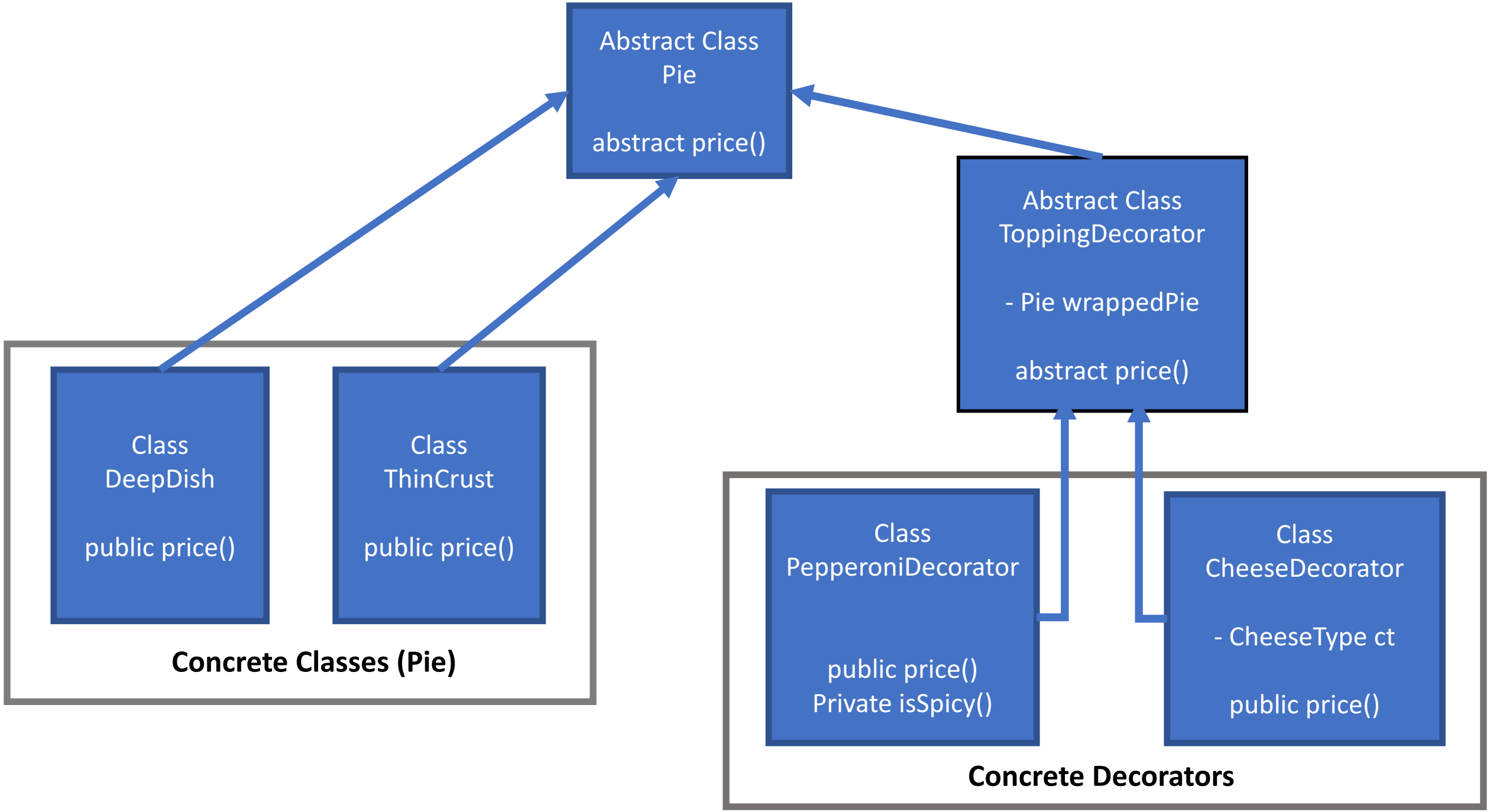
- Pizzas can be either Deep Dish or Thin Crust (cost different amounts)

- Pizzas can have various toppings, like Pepperoni and Cheese
  - There are 4 different types of cheese: Mozzarella, Bleu, Cheddar, and Parmesean. They all cost different amounts.

- We want to be able to calculate the **price** of each pizza that is made

**How would you apply the decorator pattern to design for this problem?**

Bonus: what if the prices also varied by the **size** of the pizza?


Abstract Class
Pie
abstract price()
Abstract Class
ToppingDecorator
- Pie wrappedPie
abstract price()
Class
DeepDish
public price()
Class
ThinCrust
public price()
Concrete Classes (Pie)
Class
PepperoniDecorator
public price()
Private isSpicy()
Class
CheeseDecorator
- CheeseType ct
public price()
Concrete Decorators

# Applying the Decorator Pattern

*“Talk is cheap, show me the code!”*

*~Puxuan He*

https://github.com/bambielli/DecoratorExample

# Limitations of Decorator

- If you have **code that relies on the concrete implementation's type** (i.e. DeepDish or ThinCrust) then Decorator will obscure that info from you.

- Decorating your objects manually (like in my example) **is a pain**.
  - Combining decorators with the **Factory** or **Builder** patterns makes creating decorated objects much simpler and less prone to error!

- It is *generally* against the mold of decorators to peak at other decorated layers to get more context
  - i.e. what if you wanted to know if the user ordered double pepperoni so you could print that out on their order? You'd have to know if the current pizza object is already wrapped by a pepperoni decorator.

- Large numbers of small classes… harder to understand code

# Decorators and Javascript

# Decorators and Javascript

- ES7 feature – allows decoration of both functions and classes

```
function readonly(target, key, descriptor) {
  descriptor.writable = false;
  return descriptor;
}
class Cat {
  @readonly
  meow() { return `${this.name} says Meow!`; }
}
var garfield = new Cat();
garfield.meow = function() {
  console.log('I want lasagne!');
}

// Exception: Attempted to assign to readonly property
```

# Decorators and Javascript

- ES7 feature – allows decoration of both functions and classes

```
function superhero(isSuperhero) {
  return function(target) {
    target.isSuperhero = isSuperhero
  }
}

@superhero(true)
class MySuperheroClass() { }
console.log(MySuperheroClass.isSuperhero) // true

@superhero(false)
class MySuperheroClass() { }
console.log(MySuperheroClass.isSuperhero) // false
```

- https://medium.com/google-developers/exploring-es7-decorators-76ecb65fb841 ← Addy Osmani medium post on decorators (a bit old)

# Decorators and Javascript

https://github.com/jayphelps/core-decorators.js

- Set of “core-decorators” that provide common annotations like:
  - Deprecate
  - ReadOnly
  - Enumerable
  - Mixin

- Seems like JS decorators are **still very much in flux**, though
  - Originally supported by Babel 5, but no longer in Babel 6
  - API for decorators is still being debated
  - Need to install babel-plugin-transform-decorators-legacy for support
  - That’s ok, though, since Javascript objects are built for composability out of the box ☺