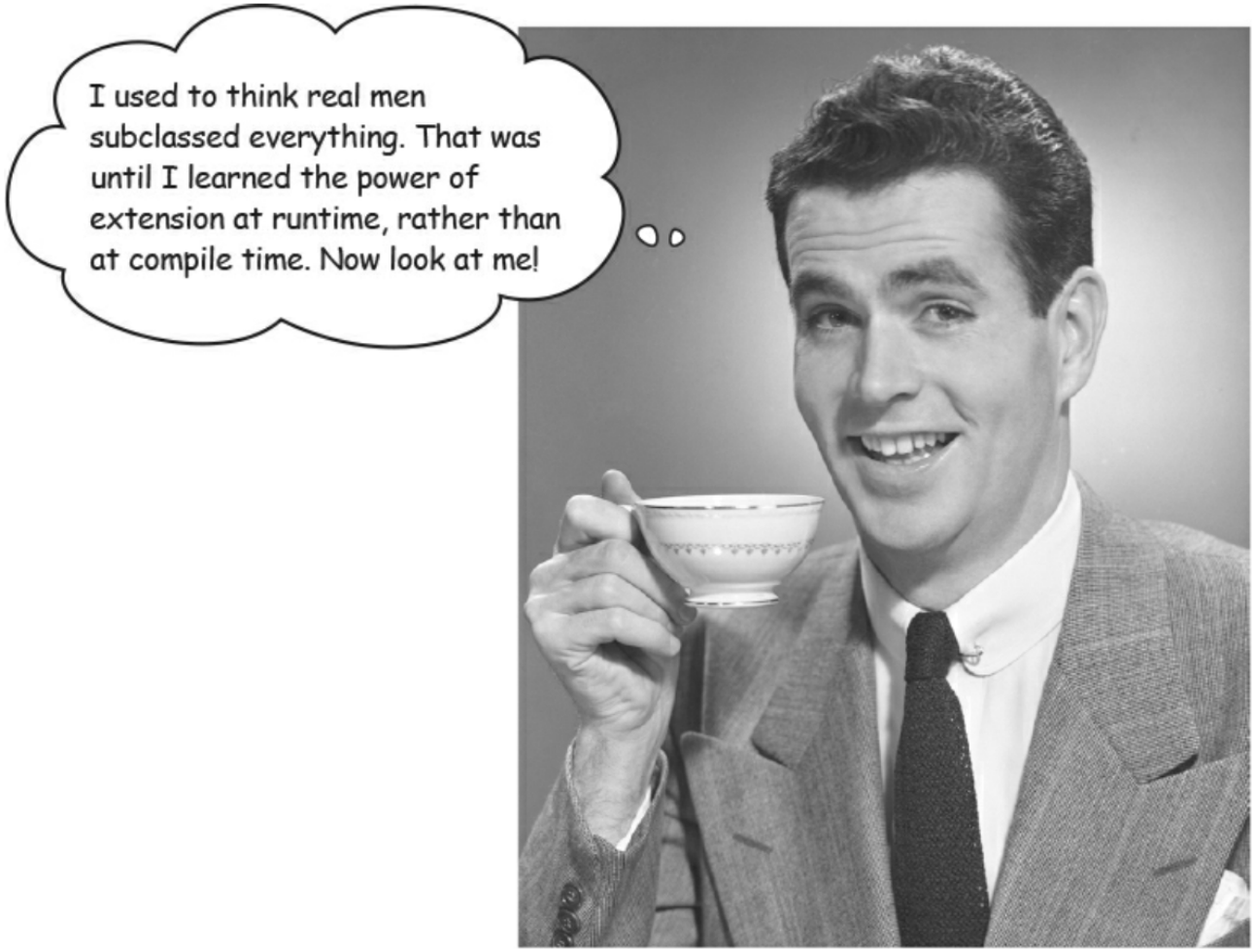
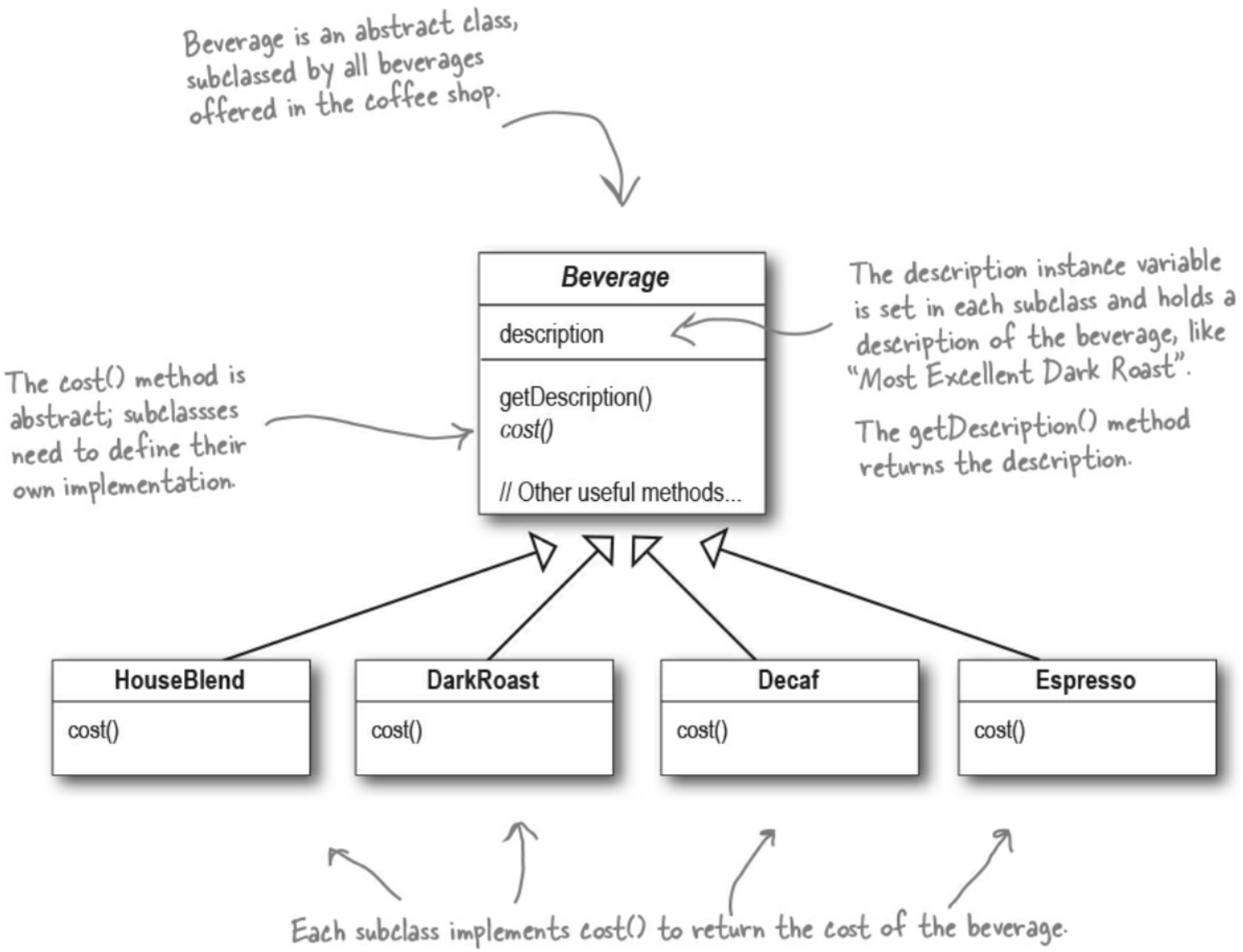
Decorator Pattern

Decorator Pattern



The problem

- IIITD decided to change the Coffee shop in academic building and you won the tender.
- You started with a choice of different types of coffees:
  - Houseblen
  - DarkRoast
  - Cappuccino
  - Espresso
  - Decaf
- You learned that people did not want off-the-self beverages, e.g., coffee, but they wanted to personalize decorate their beverages, e.g. coffee with soy milk and caramel.


Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.
Beverage
description
getDescription()
cost()
// Other useful methods...
The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".
The getDescription() method returns the description.
The cost() method is abstract; subclassses need to define their own implementation.
HouseBlend
cost()
DarkRoast
cost()
Decaf
cost()
Espresso
cost()
Each subclass implements cost() to return the cost of the beverage.

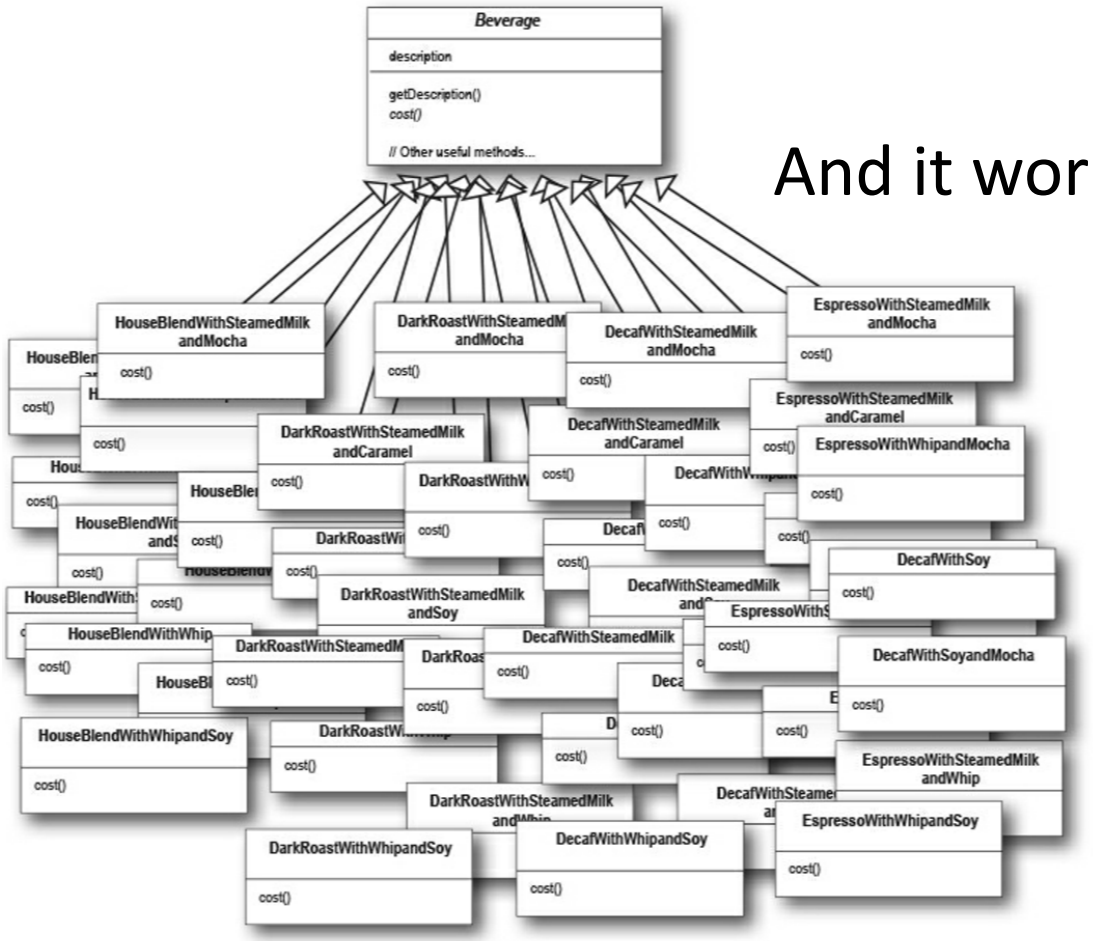
The problem

- Over the time, you learned that people did not want off-the-self beverages, e.g., cappuccino, but they wanted to personalize/decorate their beverages,
  - Coppuccino with soy milk and caramel.
  - Double shot Expresso
  - Houseblend with steamed milk…

- You also learned that another Starbucks already offers that and if you do not implement, they may be asked to replace you…
  - Now you need to extend the code

Solution

- You recall that your OOPD instructor taught you inheritance in the 4th lecture…
- Inheritance and polymorphism solves all such problems



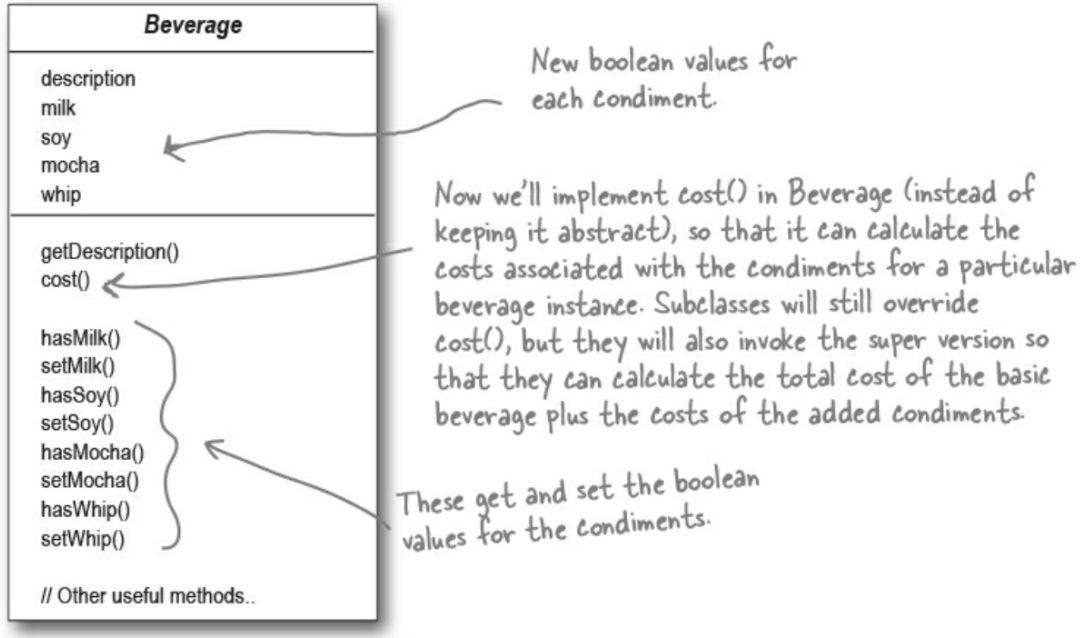
And it works ☺

Each cost method computes the cost of the coffee along with the other condiments in the order.

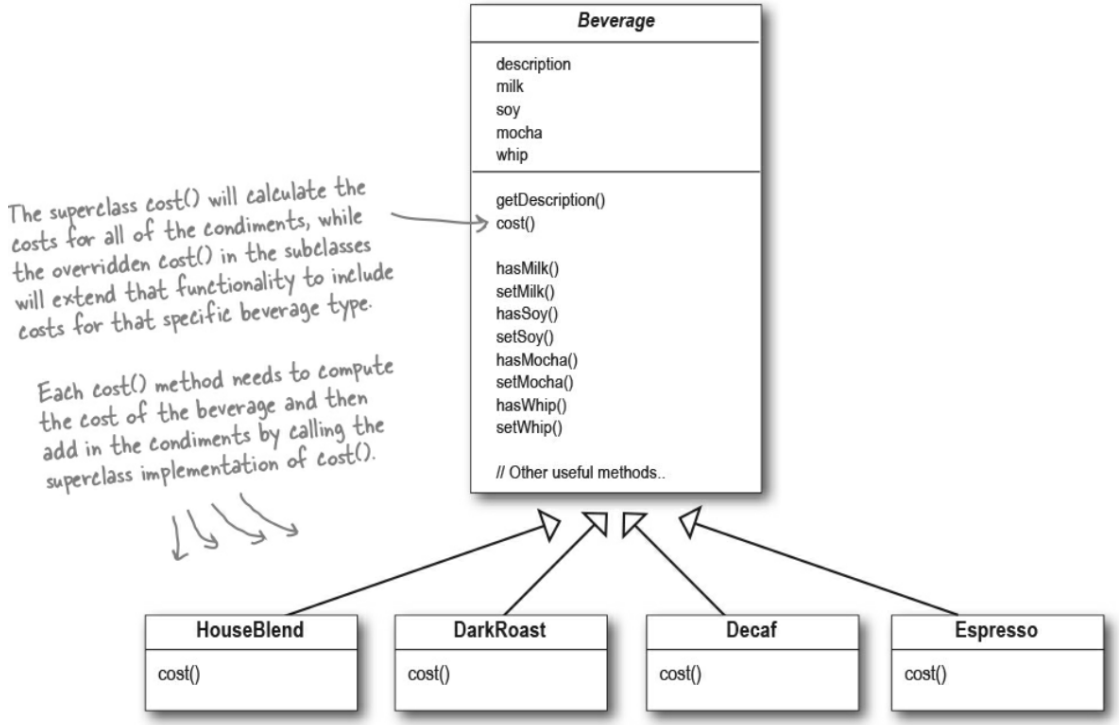
Problem

- Students complained that the vendor does not offer enough choices ☹
- You either go back to edit you code or think of using a different approach


This is stupid; why do we need all these classes? Can't we just use instance variables and inheritance in the superclass to keep track of the condiments?



And it works too ☺




See, five classes total. This is definitely the way to go.


I'm not so sure; I can see some potential problems with this approach by thinking about how the design might need to change in the future.

Potential problems

- Price change will force us to alter existing code
- New condiments will force us to alter existing code
- For some of the beverages the condiments would not be appropriate and yet they should be inherited
- …

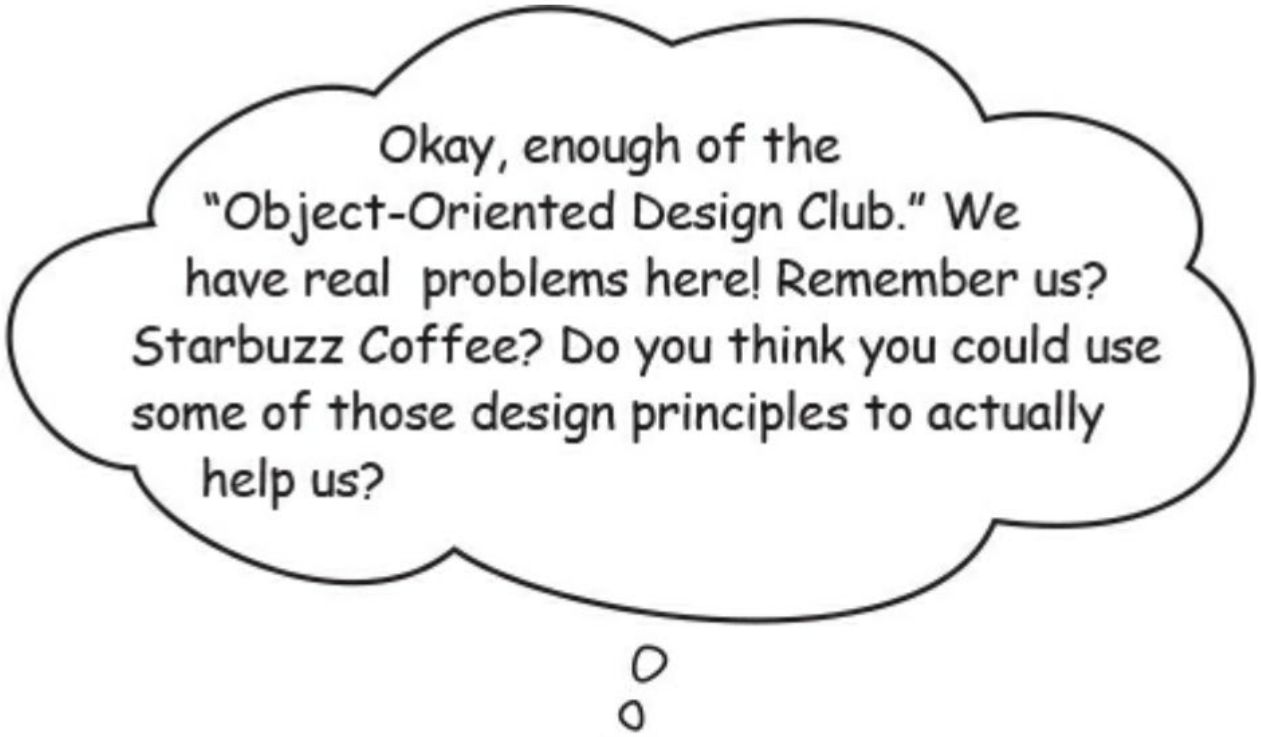
The Open-closed Principle

Design Principle

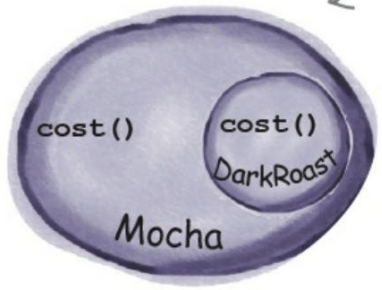
Classes should be open for extension, but closed for modification.

The Decorator Pattern

- Start with a DarkRoast object.
- Decorate it with a Mocha object.
- Decorate it with a Whip object.
- Call the cost() method and rely on delegation to add up the condiment costs.



2. **The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**



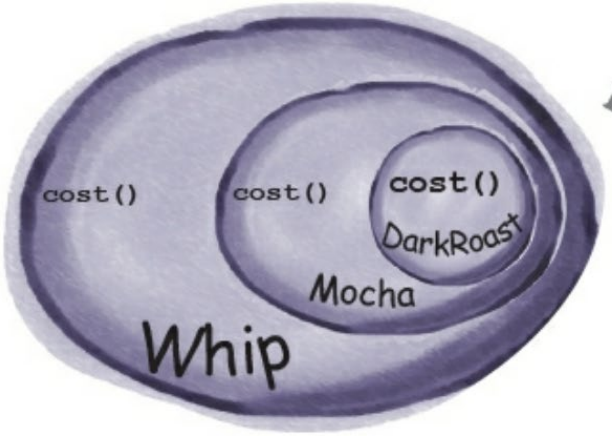
The Mocha object is a decorator. Its type mirrors the object it is decorating—in this case, a Beverage. (By "mirror," we mean it is the same type.)

So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

1. **We start with our DarkRoast object.**

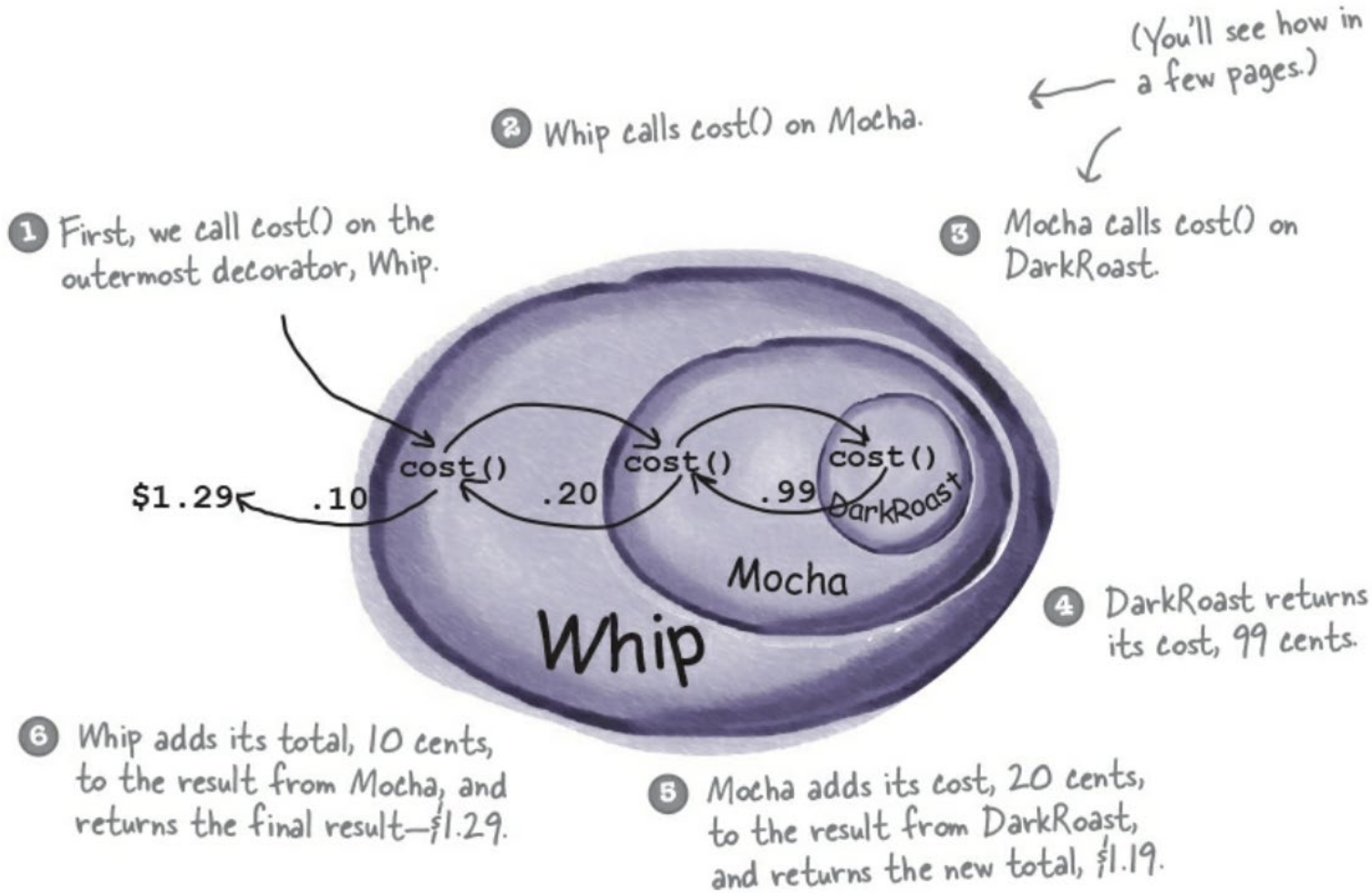
Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

3. **The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.**



Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

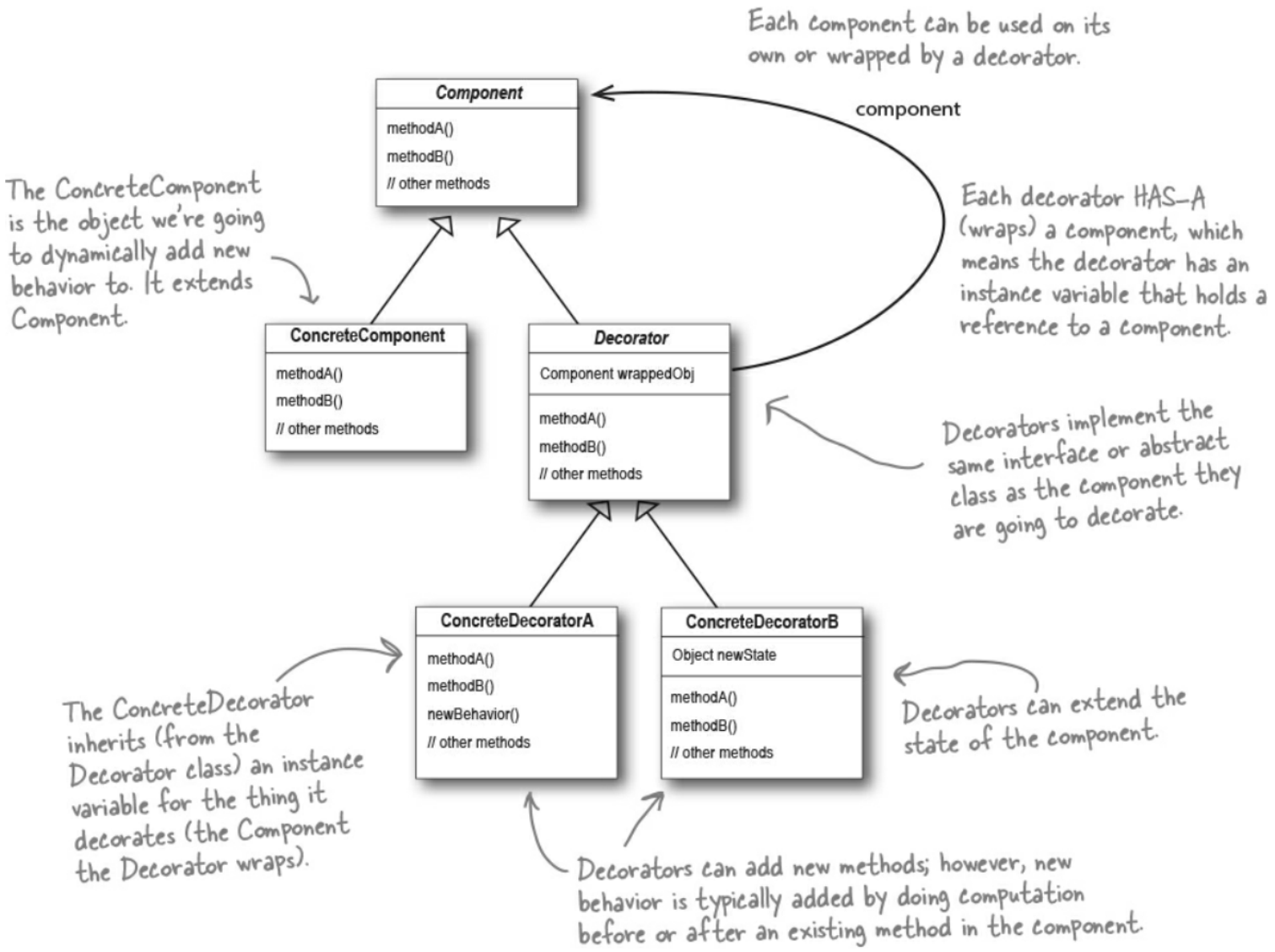
4. **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. And so on. Let's see how this works:**

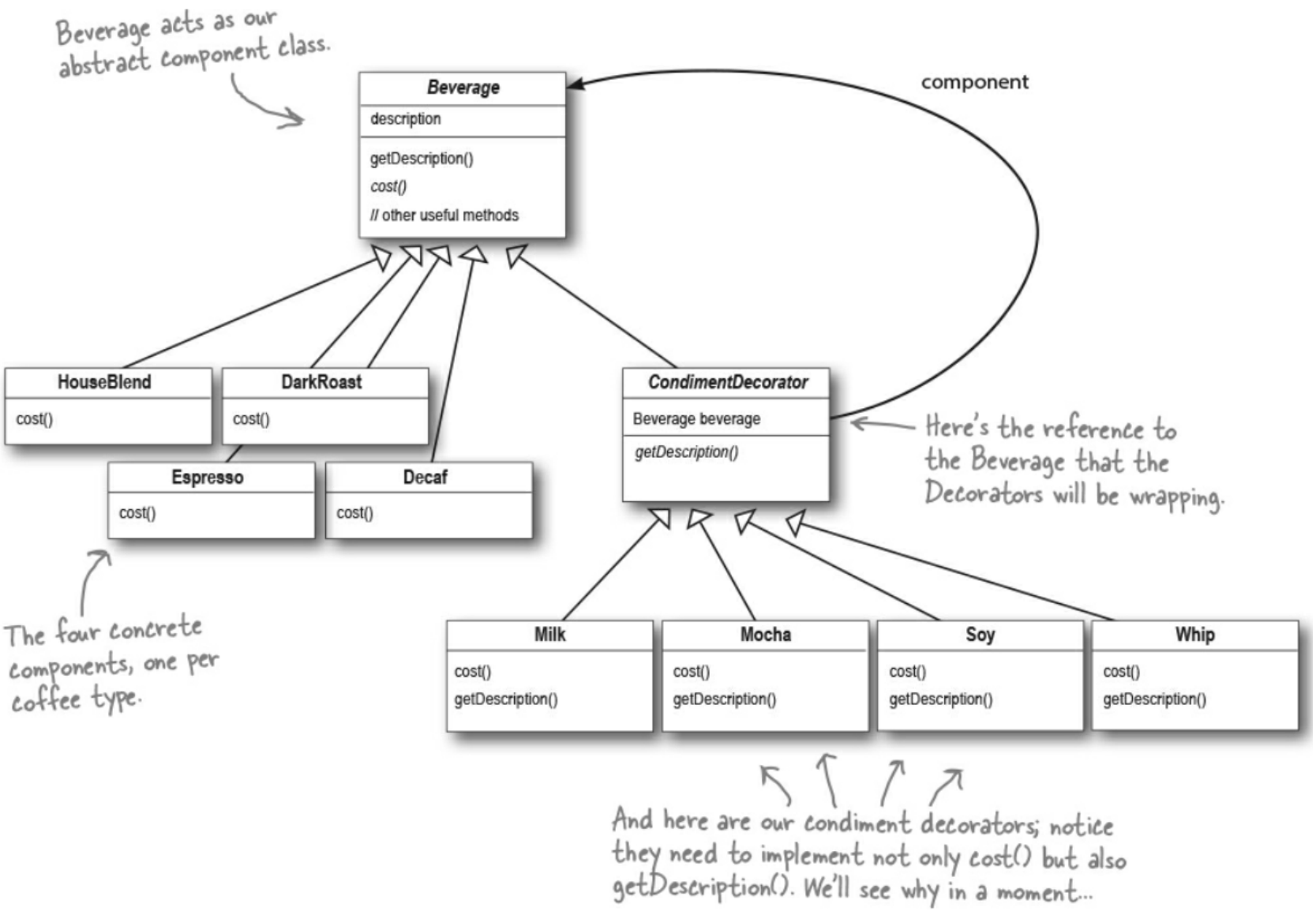


The Decorator Pattern

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior before and/or after delegating to the object it decorates to do the rest of the job.
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.


Each component can be used on its own or wrapped by a decorator.
Component
methodA()
methodB()
// other methods
component
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
Component wrappedObj
methodA()
methodB()
// other methods
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcreteDecoratorA
methodA()
methodB()
newBehavior()
// other methods
ConcreteDecoratorB
Object newState
methodA()
methodB()
// other methods
The ConcreteDecorator inherits (from the Decorator class) an instance variable for the thing it decorates (the Component the Decorator wraps).
Decorators can extend the state of the component.
Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.


Beverage acts as our abstract component class.
Beverage
description
getDescription()
cost()
// other useful methods
component
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
CondimentDecorator
Beverage beverage
getDescription()
Here's the reference to the Beverage that the Decorators will be wrapping.
The four concrete components, one per coffee type.
Milk
cost()
getDescription()
Mocha
cost()
getDescription()
Soy
cost()
getDescription()
Whip
cost()
getDescription()
And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

Inheritance in Decorator pattern

- Using Inheritance for type Matching and not to get behavior
  - It is vital that the decorators have the same type as the objects they are going to decorate.

Singleton Pattern

- There are many objects we only need one of: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards.

The Classic Implementation

Let's rename MyClass to Singleton.

We have a static variable to hold our one instance of the class Singleton.

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    / other useful methods here
}
```

Our constructor is declared private; only Singleton can instantiate this class!

The getInstance() method gives us a way to instantiate the class and also to return an instance of it.

Of course, Singleton is a normal class; it has other useful instance variables and methods.

Reference

Erich, Gamma; Helm Richard; Johnson Ralph; Vlissides John. Design Patterns. Pearson Education. Kindle Edition.

Freeman, Eric; Robson, Elisabeth. Head First Design Patterns. O'Reilly Media. Kindle Edition.