

THE DECORATOR PATTERN

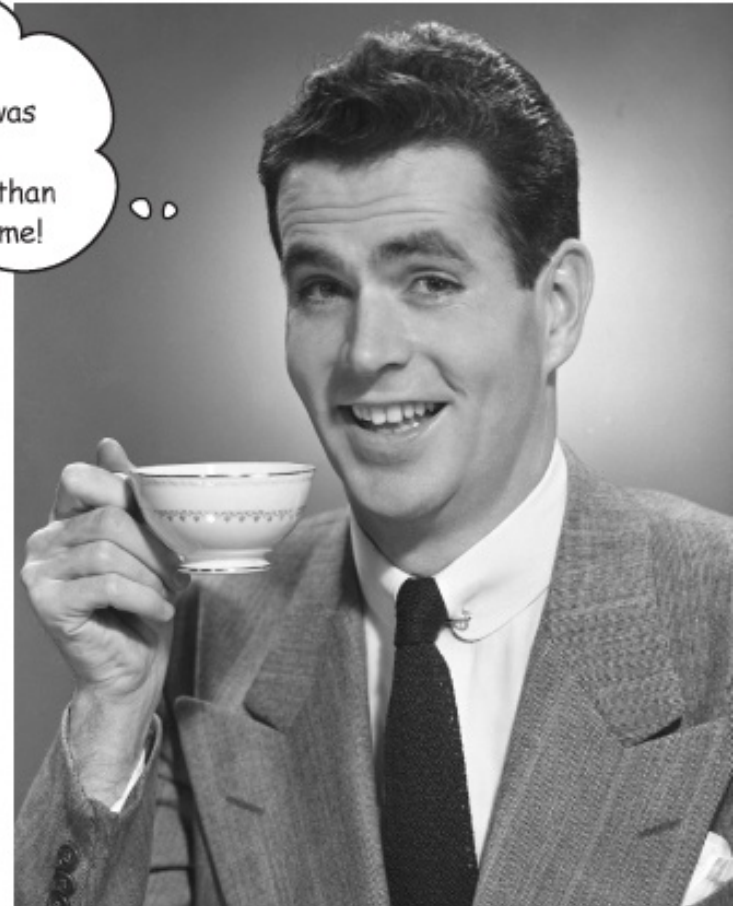
Chandan R. Rupakheti

Week 2-1

Today ...

I used to think real men subclassed everything. That was until I learned the power of extension at runtime, rather than at compile time. Now look at me!

- The Decorator Pattern
- The Open-Closed Principle



A Few Words of Caution!

CSSE 374 is a technical course. If you do not pay attention during lectures, Lab/Homework can become a challenge.

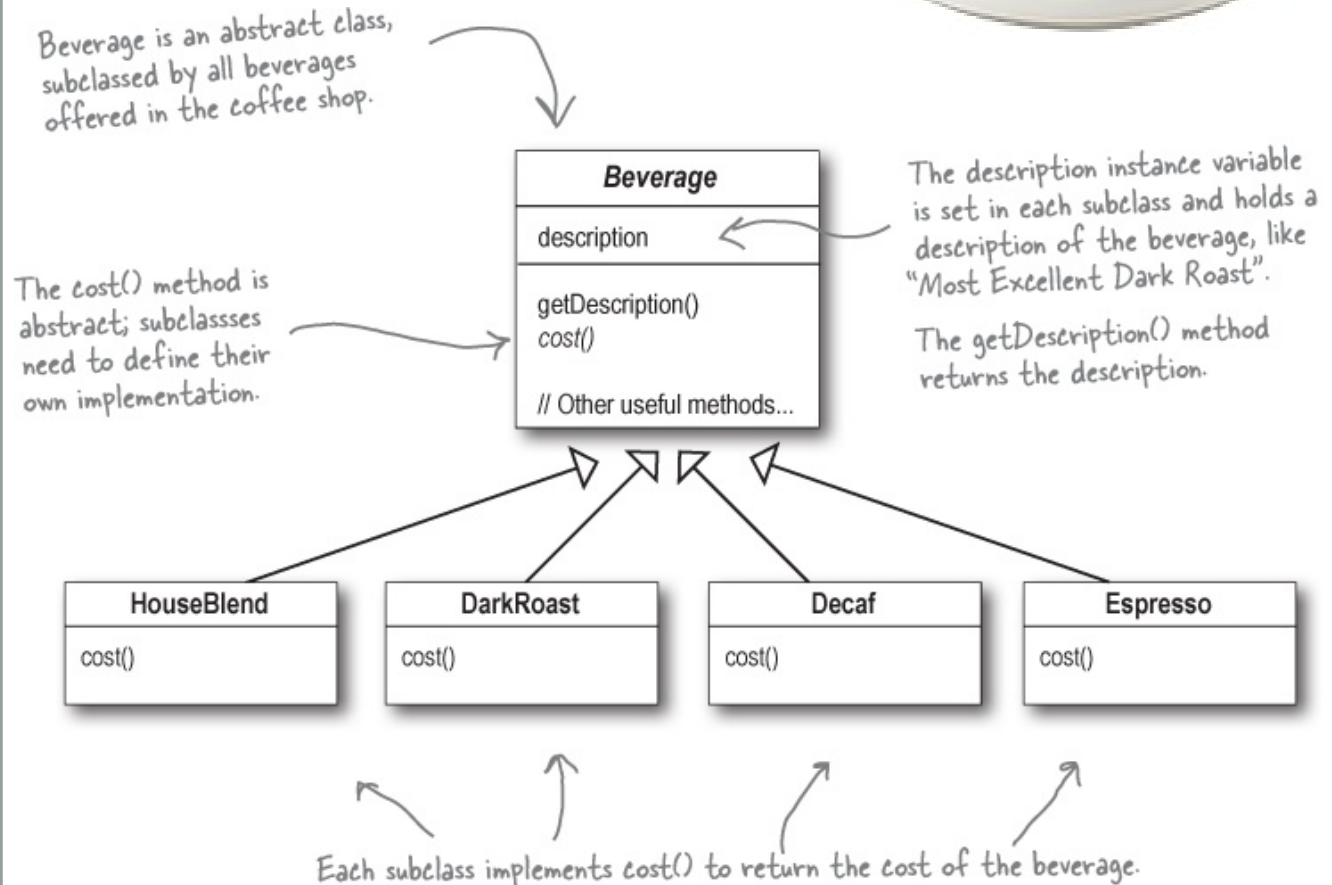
Welcome to Starbuzz Coffee



Starbuzz Coffee -
the fastest growing
coffee shop around
the world

Need to update their
ordering systems to
match their
beverage offerings

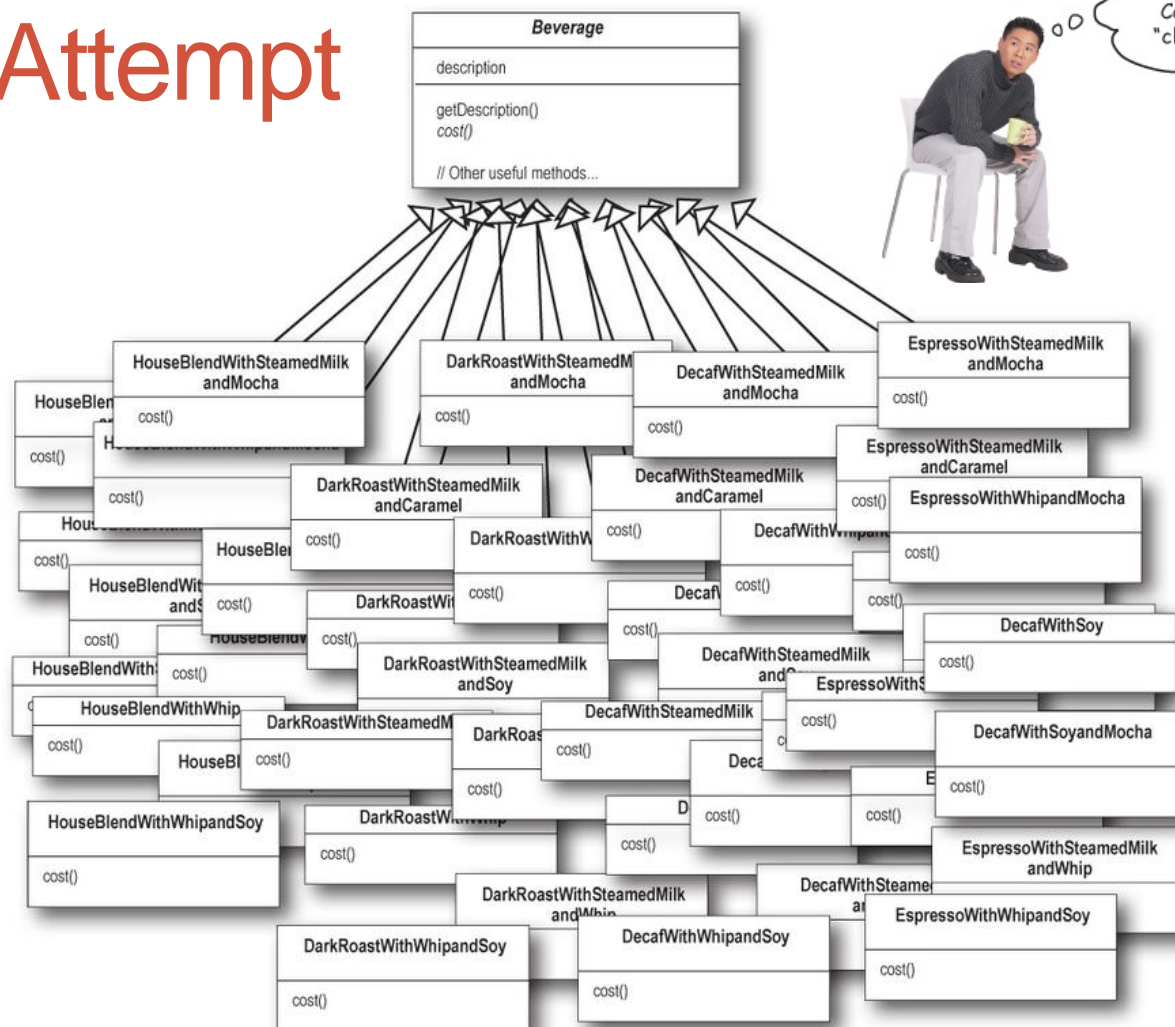
When they first went
into business they
designed their
classes like this...



The First Attempt

In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk.

Starbuzz charges a bit for each of these, so they really need to get them built into their order system.



Whoa!
Can you say
"class explosion"?

Q1

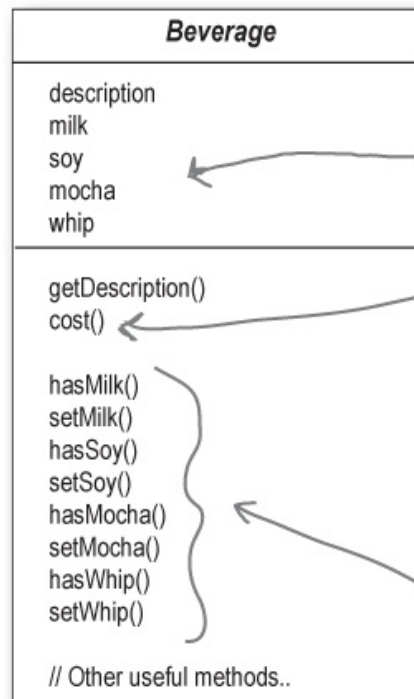
Each cost method computes the cost of the coffee along with the other condiments in the order.

Next Attempt ...

1/2

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?

Well, let's give it a try. Let's start with the Beverage base class and add instance variables to represent whether or not each beverage has milk, soy, mocha, and whip...



New boolean values for each condiment.

Now we'll implement `cost()` in `Beverage` (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override `cost()`, but they will also invoke the super version so that they can calculate the total cost of the basic beverage plus the costs of the added condiments.

These get and set the boolean values for the condiments.

Next Attempt ...

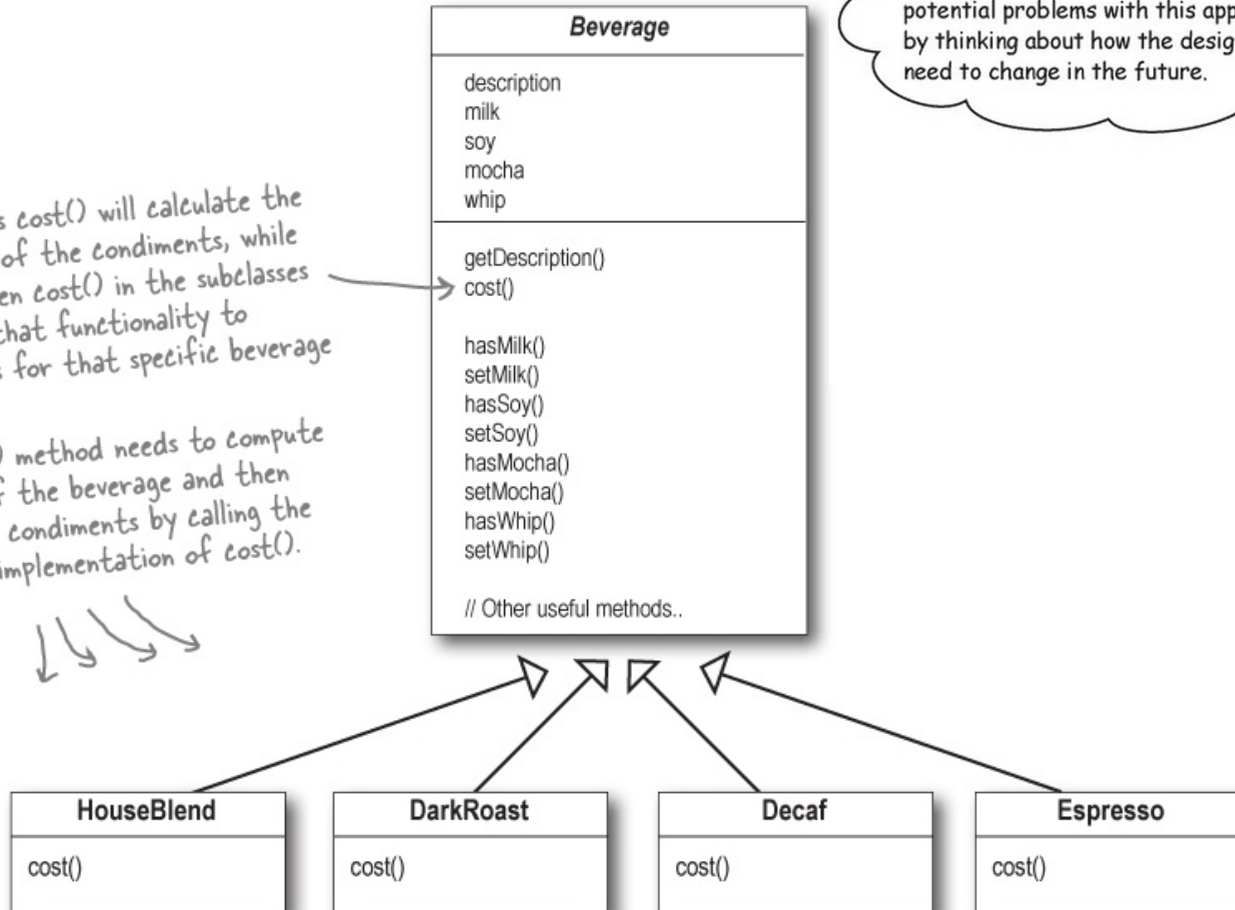
2/2

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



The superclass `cost()` will calculate the costs for all of the condiments, while the overridden `cost()` in the subclasses will extend that functionality to include costs for that specific beverage type.

Each `cost()` method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of `cost()`.



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.



Try Out: Implement `Beverage.cost()` and `HouseBlend.cost()`

The Open-Closed Principle

Design Principle:

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



Come on in; we're *open*. Feel free to extend our classes with any new behavior you like. If your needs or requirements change, just go ahead and make your own extensions.



Sorry, we're *closed*. That's right, we spent a lot of time getting this code correct and bug free, so we can't let you alter the existing code. It must remain closed to modification.

Be careful when choosing the areas of code that need to be extended; applying the Open-Closed Principle EVERYWHERE is wasteful and unnecessary, and can lead to complex, hard-to-understand code.

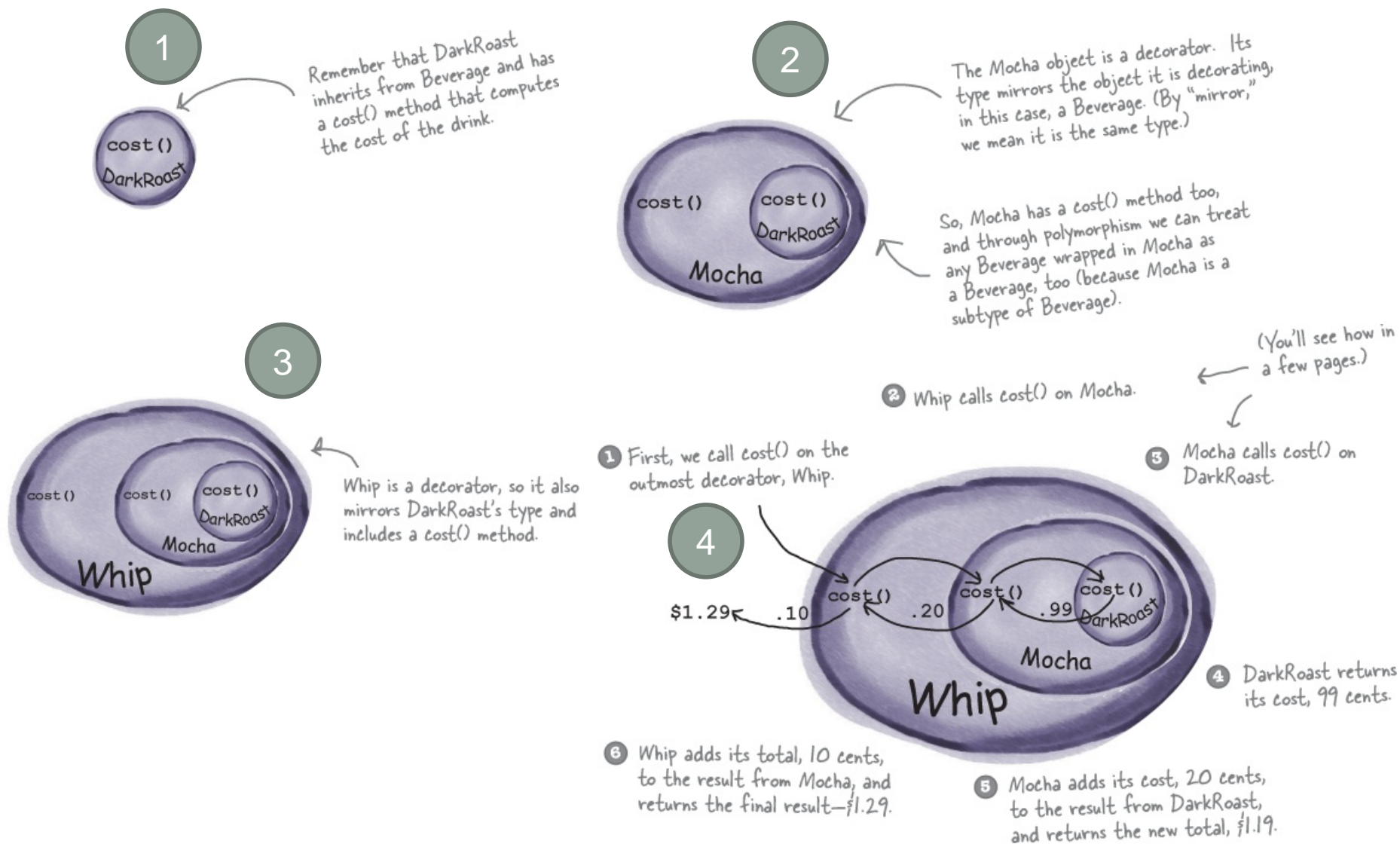
Meet the Decorator

Using inheritance so far, we got class explosions and rigid designs, or we added functionality to the base class that wasn't appropriate for some of the subclasses.

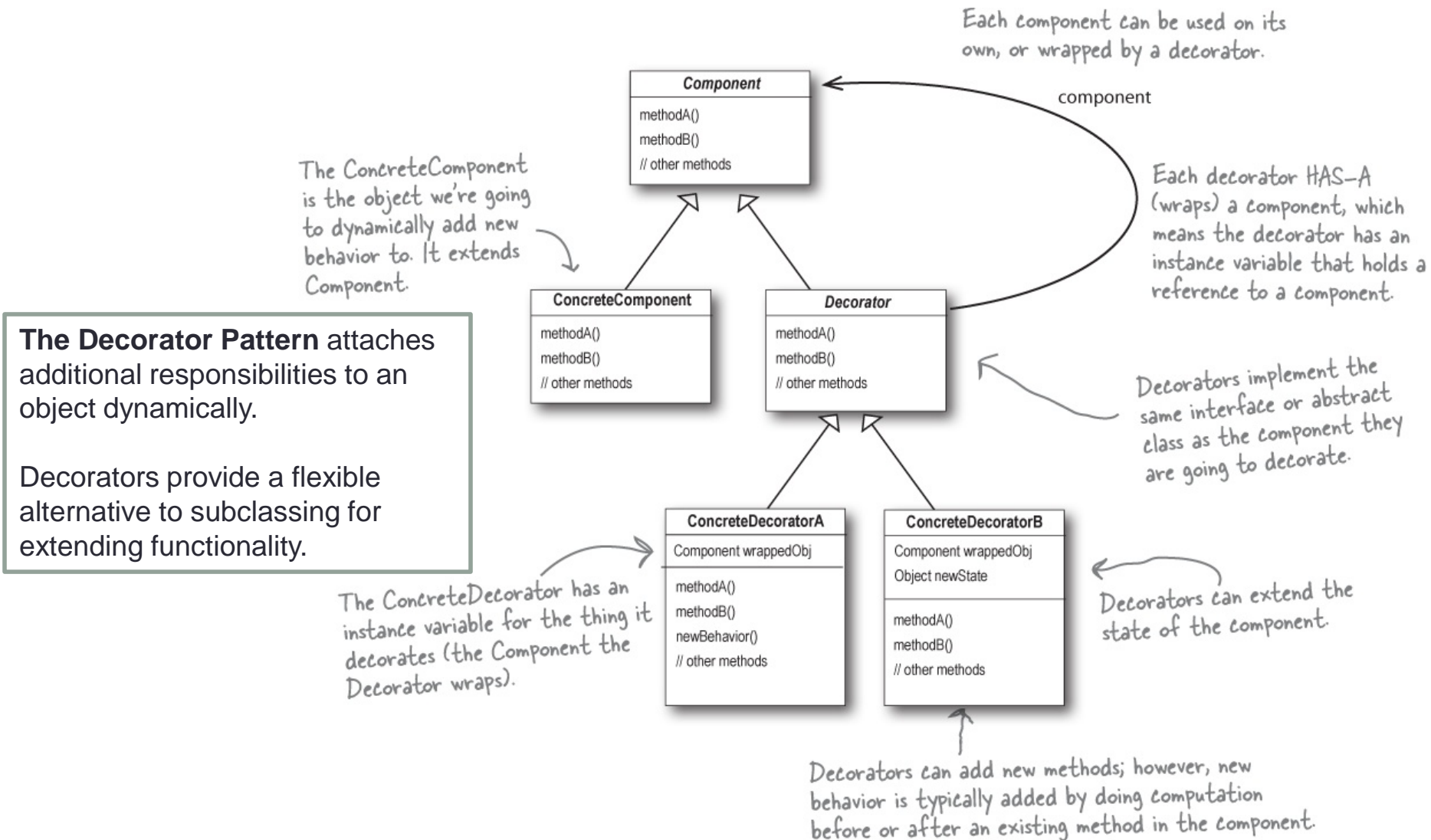
1. Take a DarkRoast object
2. Decorate it with a Mocha object
3. Decorate it with a Whip object
4. Call the cost() method and rely on delegation to add on the condiment costs



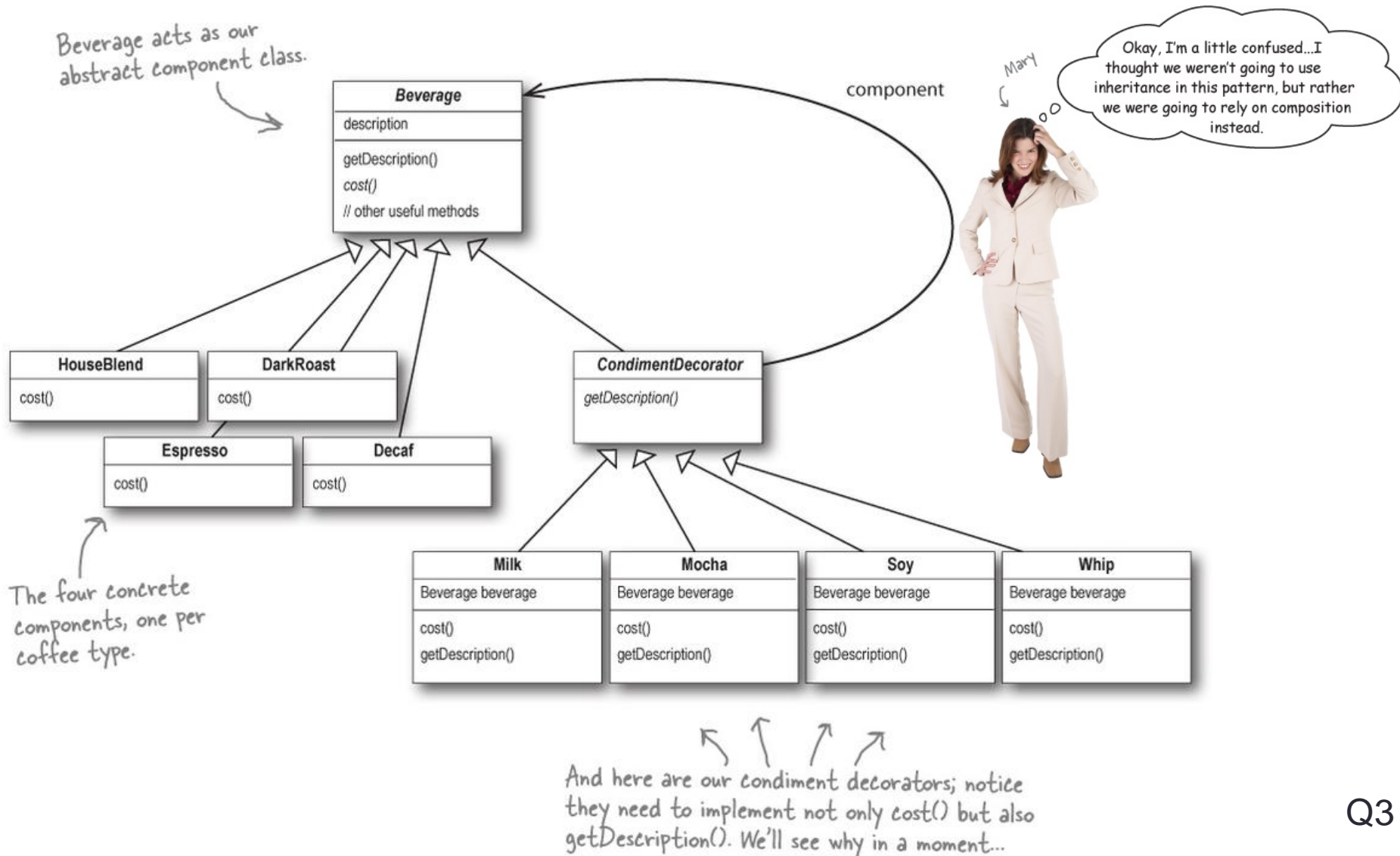
Constructing a drink order with Decorators



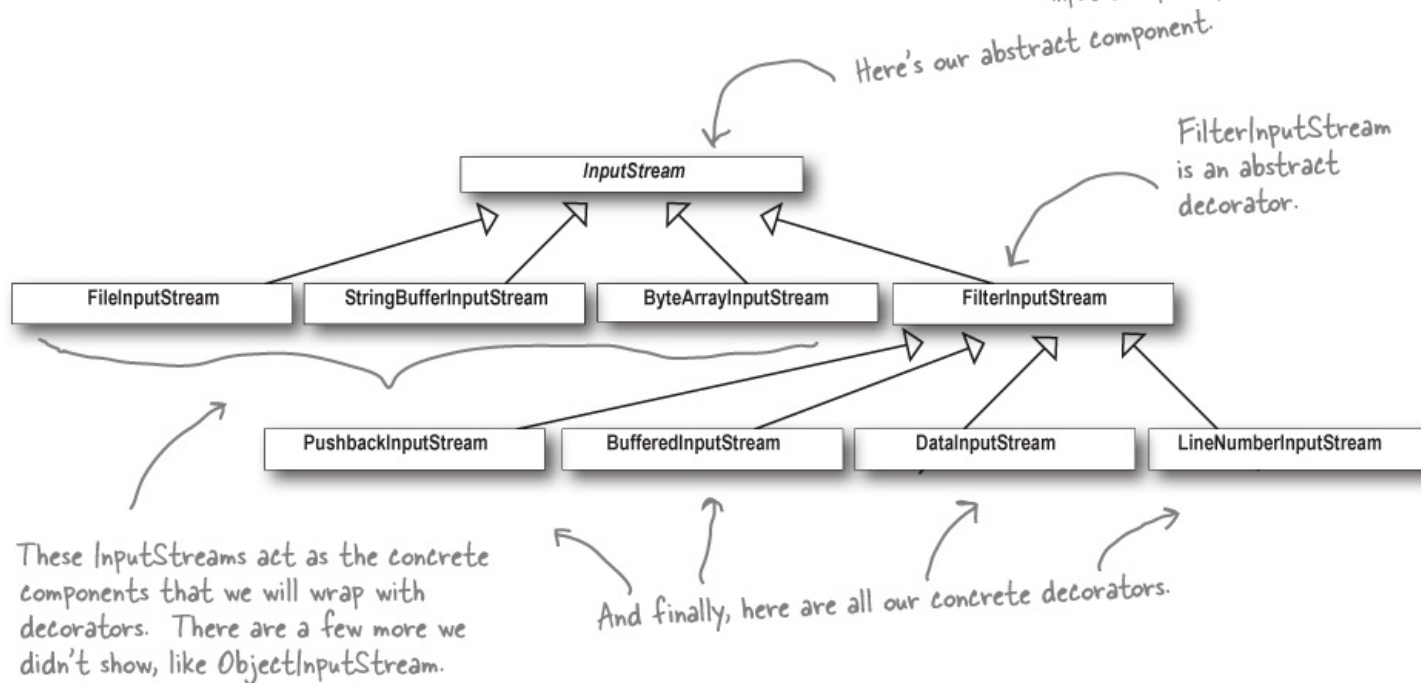
The Decorator Pattern Defined



Decorating our Beverages



Real World Decorators: Java I/O

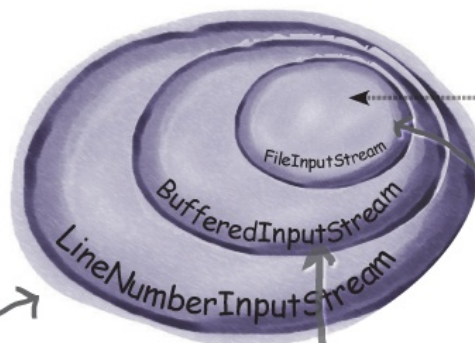


LineNumberInputStream is also a concrete decorator. It adds the ability to count the line numbers as it reads data.

BufferedInputStream is a concrete decorator. BufferedInputStream adds buffering behavior to a FileInputStream: it buffers input to improve performance.

FileInputStream is the component that's being decorated. The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, and a few others. All of these give us a base component from which to read bytes.

A text file for reading.



Recap

Design Principle:

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

Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.

The Decorator Pattern provides an alternative to subclassing for extending behavior.

Decorators can result in many small objects in our design, and overuse can be complex.