THE DECORATOR PATTERN

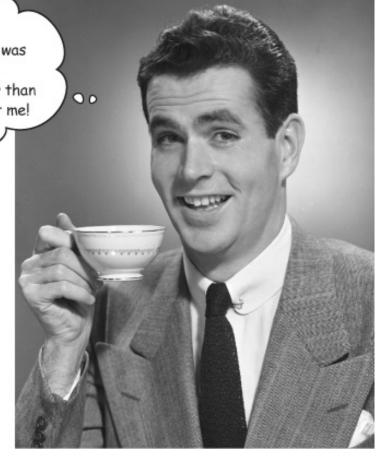
Chandan R. Rupakheti Week 2-1

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!

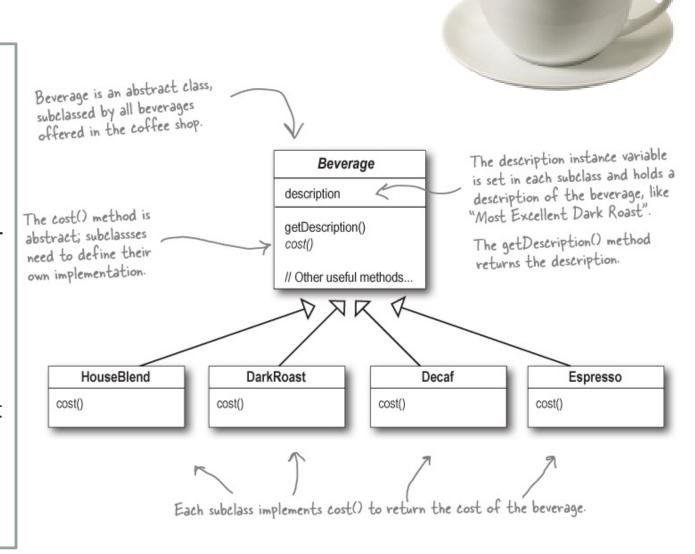
CSSE 374 is a technical course. If you do not pay attention during lectures, Lab/Homework can become a challenge.

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

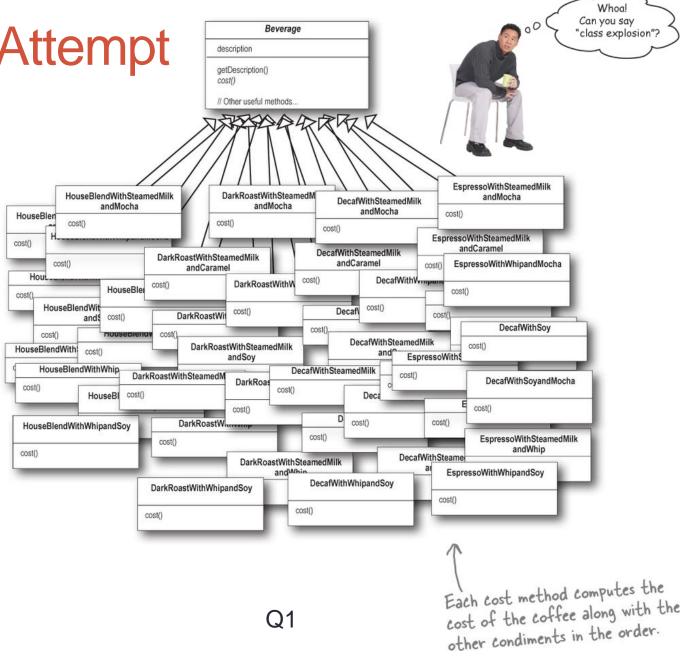
When they first went into business they designed their classes like this...



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.

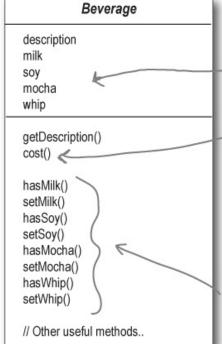
Starbuzz charges a bit for each of these, so they really need to get them built into their order system.



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?

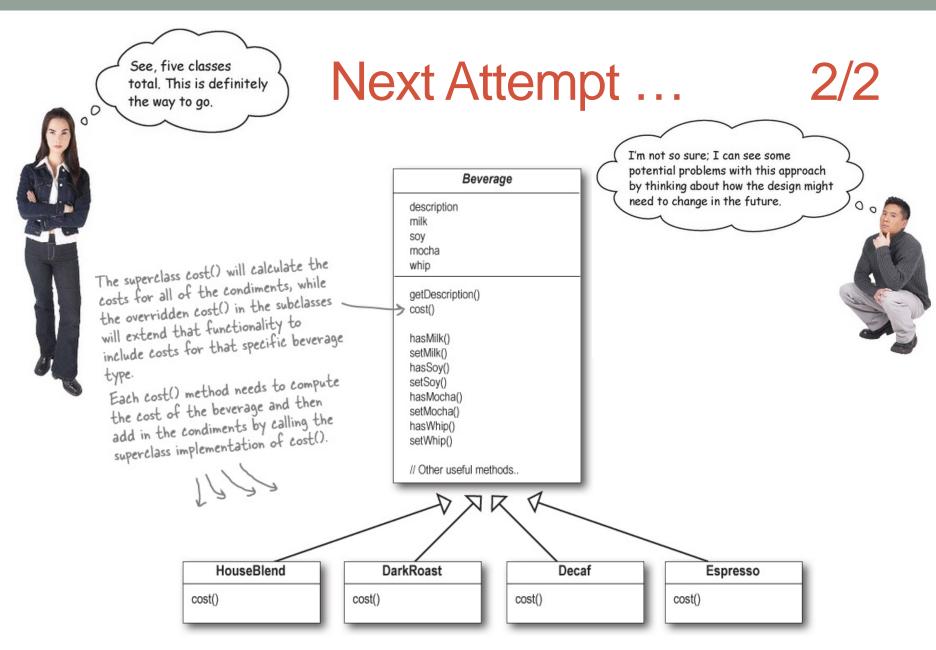
Well, let's give it a try. Let's start with the Beverage base class and add instance variables to represent whether or not each beverage has milk, soy, mocha, and whip...



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.



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

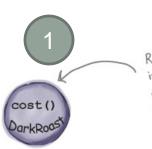
Using inheritance so far, we got class explosions and rigid designs, or we added functionality to the base class that wasn't appropriate for some of the subclasses.

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

Okay, enough of the
"Object Oriented Design Club." We
have real problems here! Remember
us? Starbuzz Coffee? Do you think you
could use some of those design principles to
actually help us?



Constructing a drink order with Decorators



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

cost() cost() DarkRoos Mocha

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.)

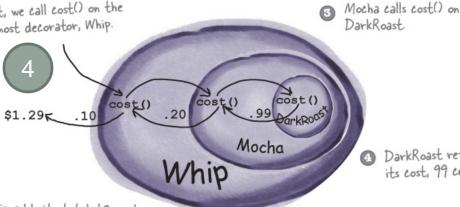
So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

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

Whip calls cost() on Mocha.

(You'll see how in a few pages.)

1 First, we call cost() on the outmost decorator, Whip.



DarkRoast returns its cost, 99 cents.

6 Whip adds its total, 10 cents, to the result from Mocha, and returns the final result-\$1.29.

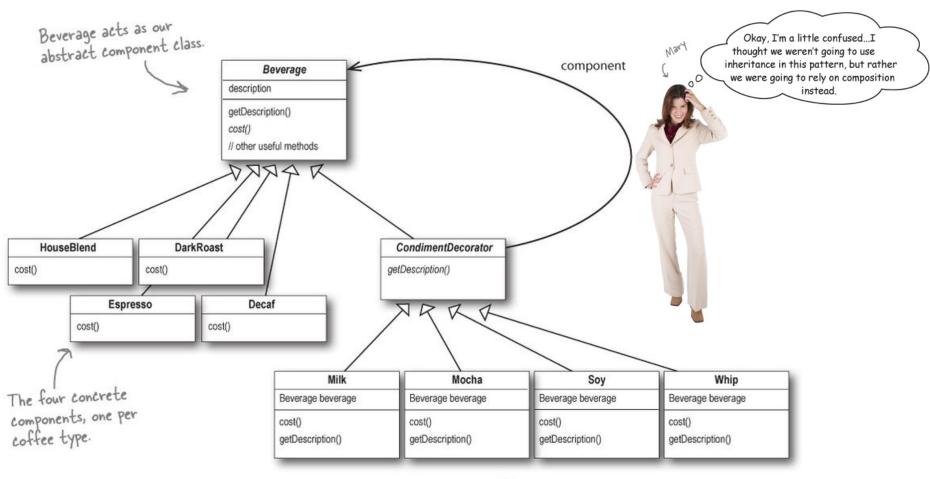
Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, \$1.19.

The Decorator Pattern Defined

Each component can be used on its own, or wrapped by a decorator. Component component methodA() methodB() The Concrete Component // other methods Each decorator HAS-A is the object we're going (wraps) a component, which to dynamically add new means the decorator has an behavior to. It extends instance variable that holds a Component. reference to a component. ConcreteComponent Decorator methodA() methodA() The Decorator Pattern attaches methodB() methodB() Decorators implement the additional responsibilities to an same interface or abstract // other methods // other methods object dynamically. class as the component they are going to decorate. Decorators provide a flexible alternative to subclassing for ConcreteDecoratorA ConcreteDecoratorB extending functionality. Component wrappedObj Component wrappedObj Object newState The Concrete Decorator has an Decorators can extend the methodA() instance variable for the thing it methodB() state of the component. methodA() decorates (the Component the newBehavior() methodB() // other methods // other methods Decorator wraps).

Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

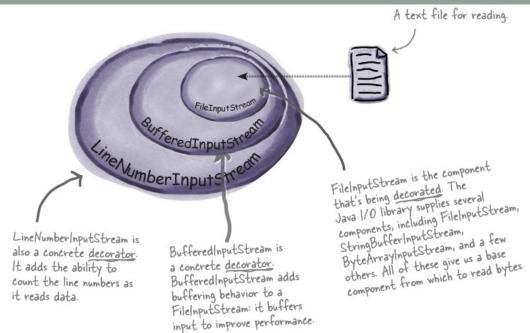
Decorating our Beverages

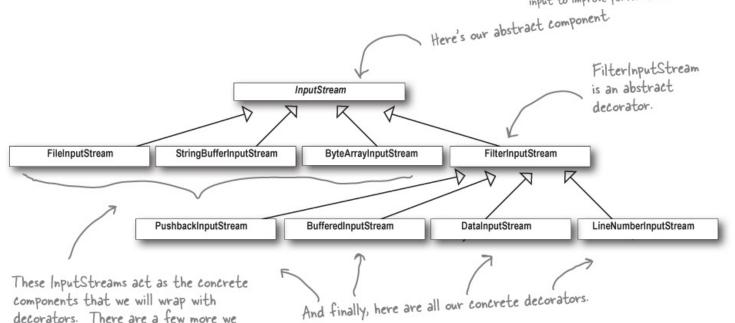


And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

Real World Decorators: Java I/O

didn't show, like ObjectInputStream.





Q4-5

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.