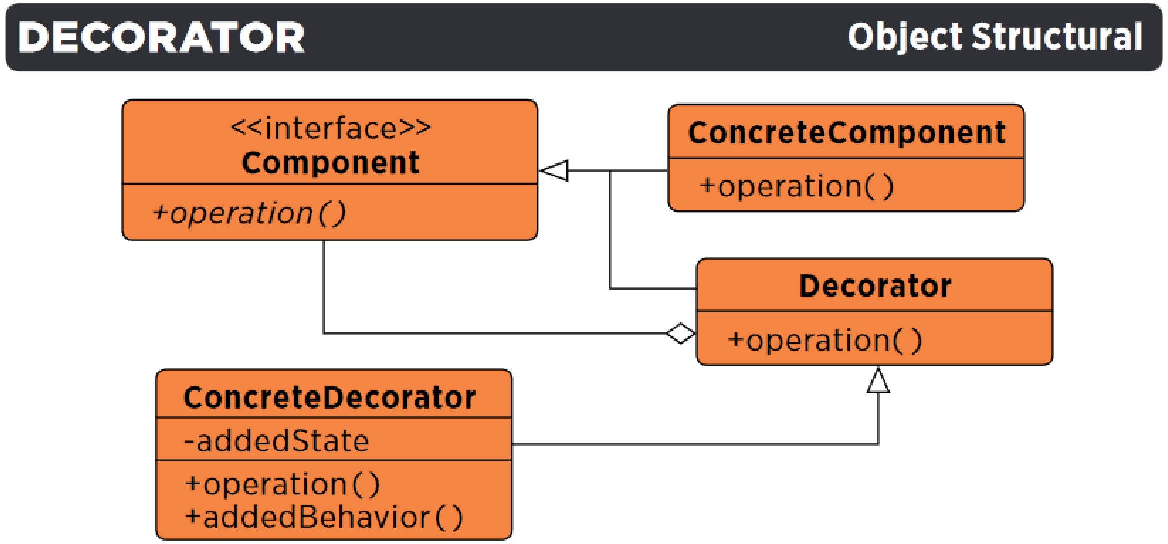
Decorator Pattern

For the Complete Code, See the "Official" Head-First Design Patterns GitHub Repo:

https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/

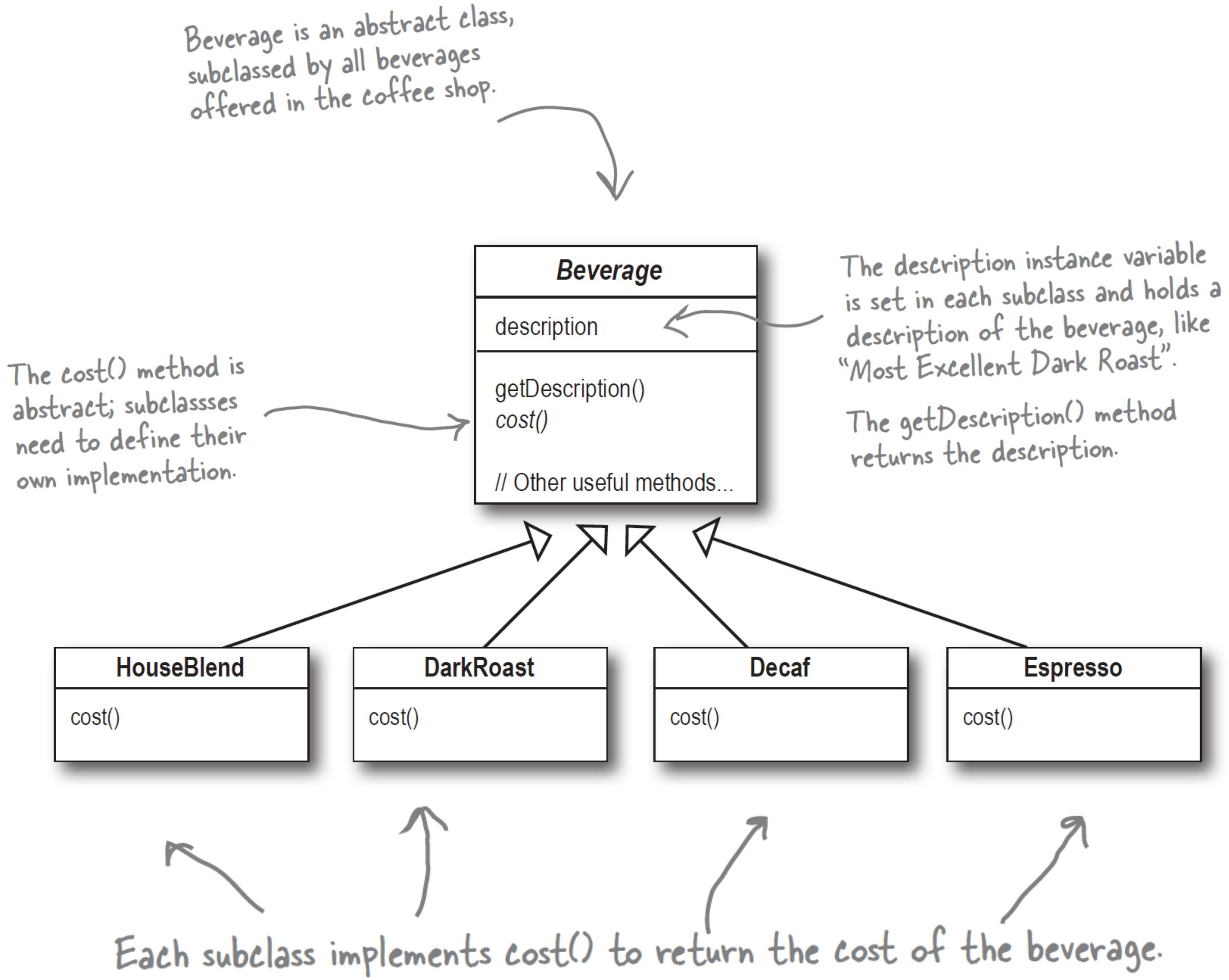
And the course SVN repo:

svn://cosc436.net:65436/Examples/trunk

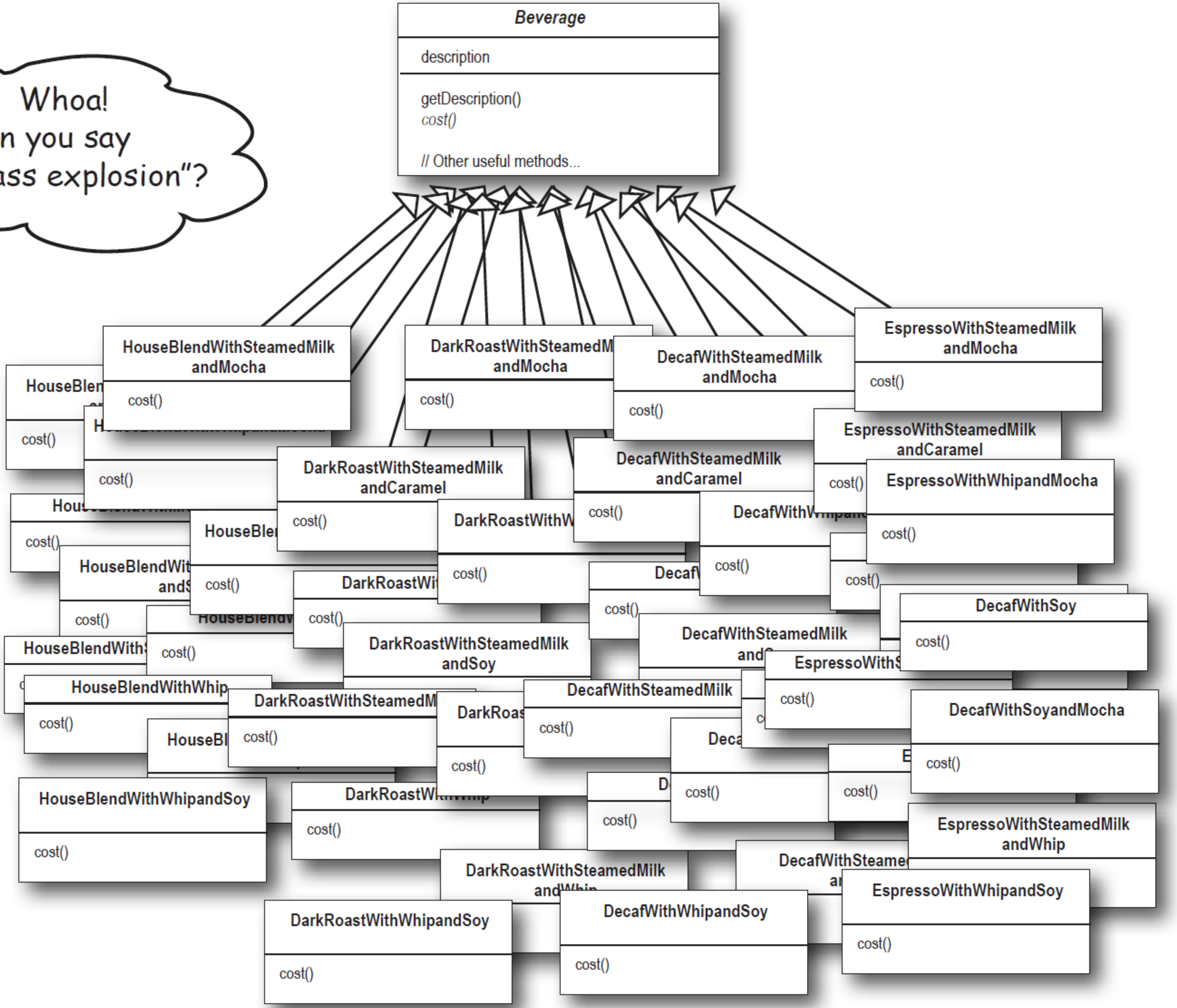


We'll re-examine the typical overuse of inheritance and you'll learn how to decorate your classes at runtime using a form of object composition. Why? Once you know the techniques of decorating, you'll be able to give your (or someone else's) objects new responsibilities *without making any code changes to the underlying classes.*

Purpose	Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviors.
Use When	• Object responsibilities and behaviors should be dynamically modifiable. • Concrete implementations should be decoupled from responsibilities and behaviors. • Subclassing to achieve modification is impractical or impossible. • Specific functionality should not reside high in the object hierarchy. • A lot of little objects surrounding a concrete implementation is acceptable.
Example	Many businesses set up their mail systems to take advantage of decorators. When messages are sent from someone in the company to an external address the mail server decorates the original message with copyright and confidentiality information. As long as the message remains internal the information is not attached. This decoration allows the message itself to remain unchanged until a runtime decision is made to wrap the message with additional information.


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.

Whoa!
Can you say
"class explosion"?



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

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?

Beverage
description milk soy mocha whip
getDescription() cost() hasMilk() setMilk() hasSoy() setSoy() hasMocha() setMocha() hasWhip() setWhip() // Other useful methods..

New boolean values for each condiment.

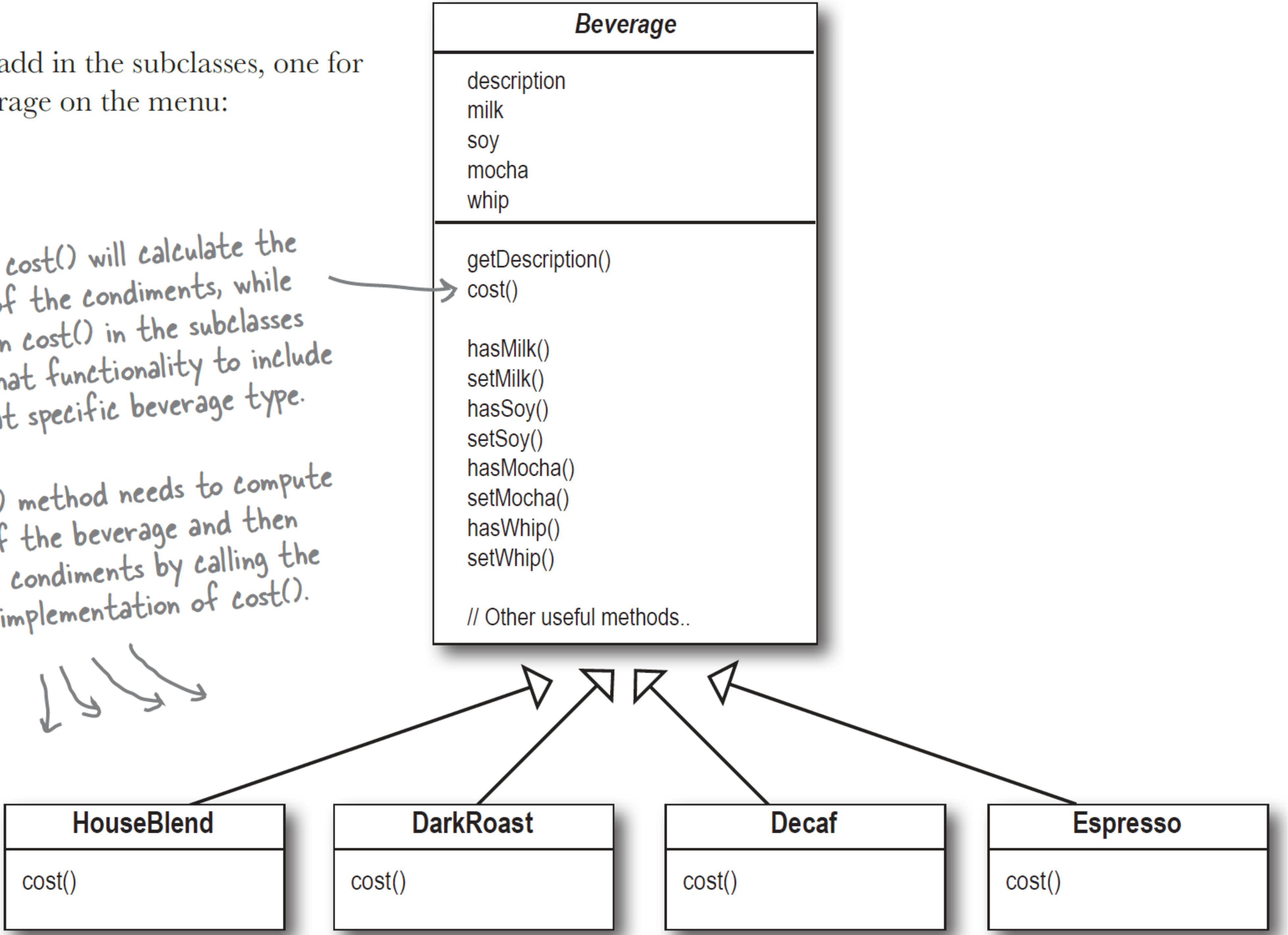
Now we'll implement cost() in Beverage (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override cost(), but they will also invoke the super version so that they can calculate the total cost of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

These get and set the boolean values for the condiments.

Now let's add in the subclasses, one for each beverage on the menu:

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.

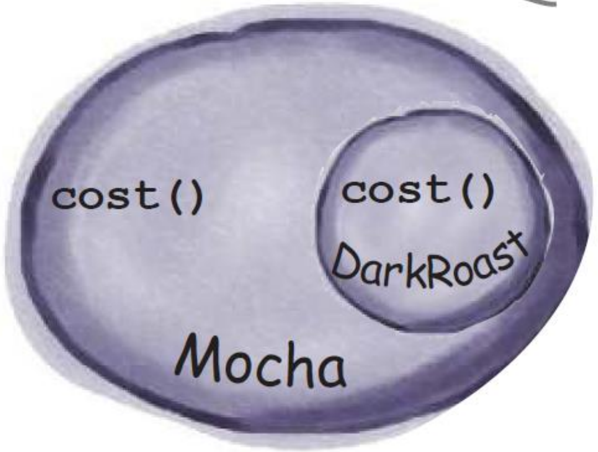
Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().



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.

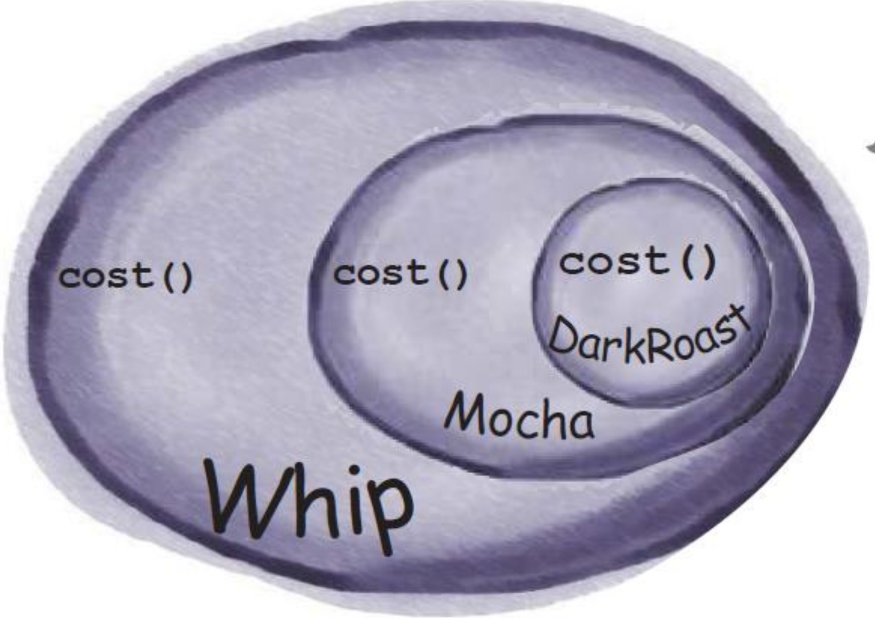
2. **The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**



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).

❸ **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.

So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

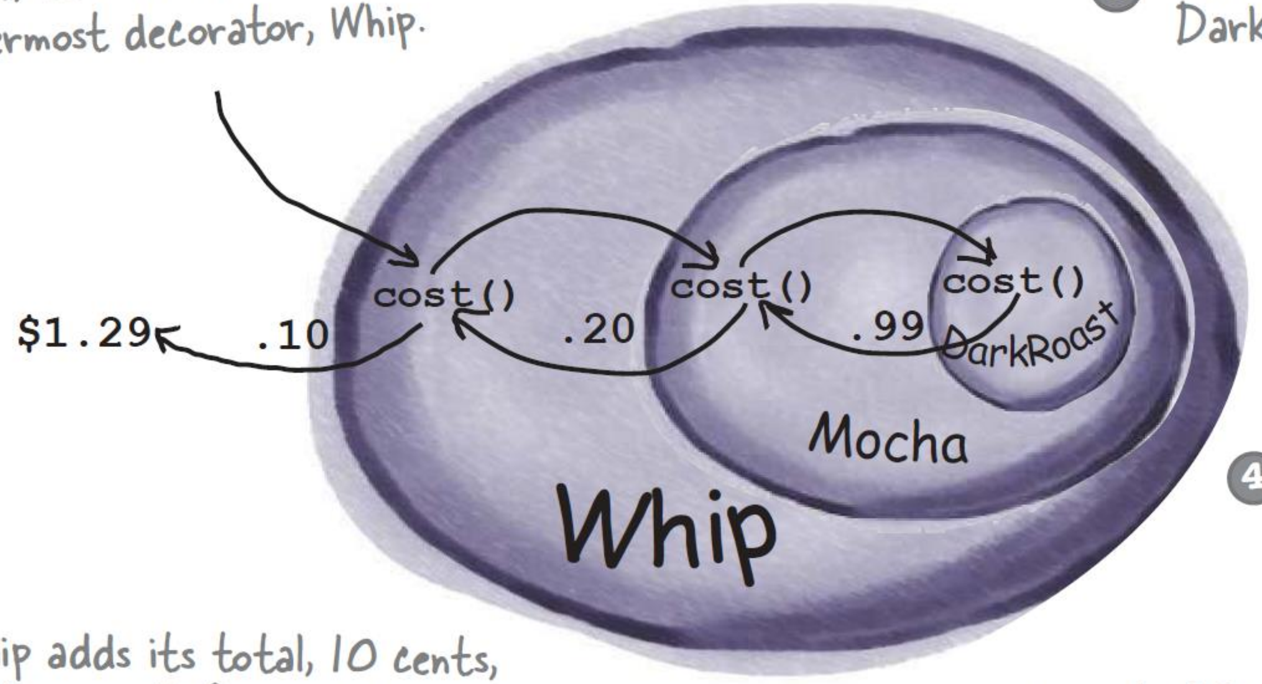
4 **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. And so on. Let's see how this works:**

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

2 Whip calls cost() on Mocha.

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

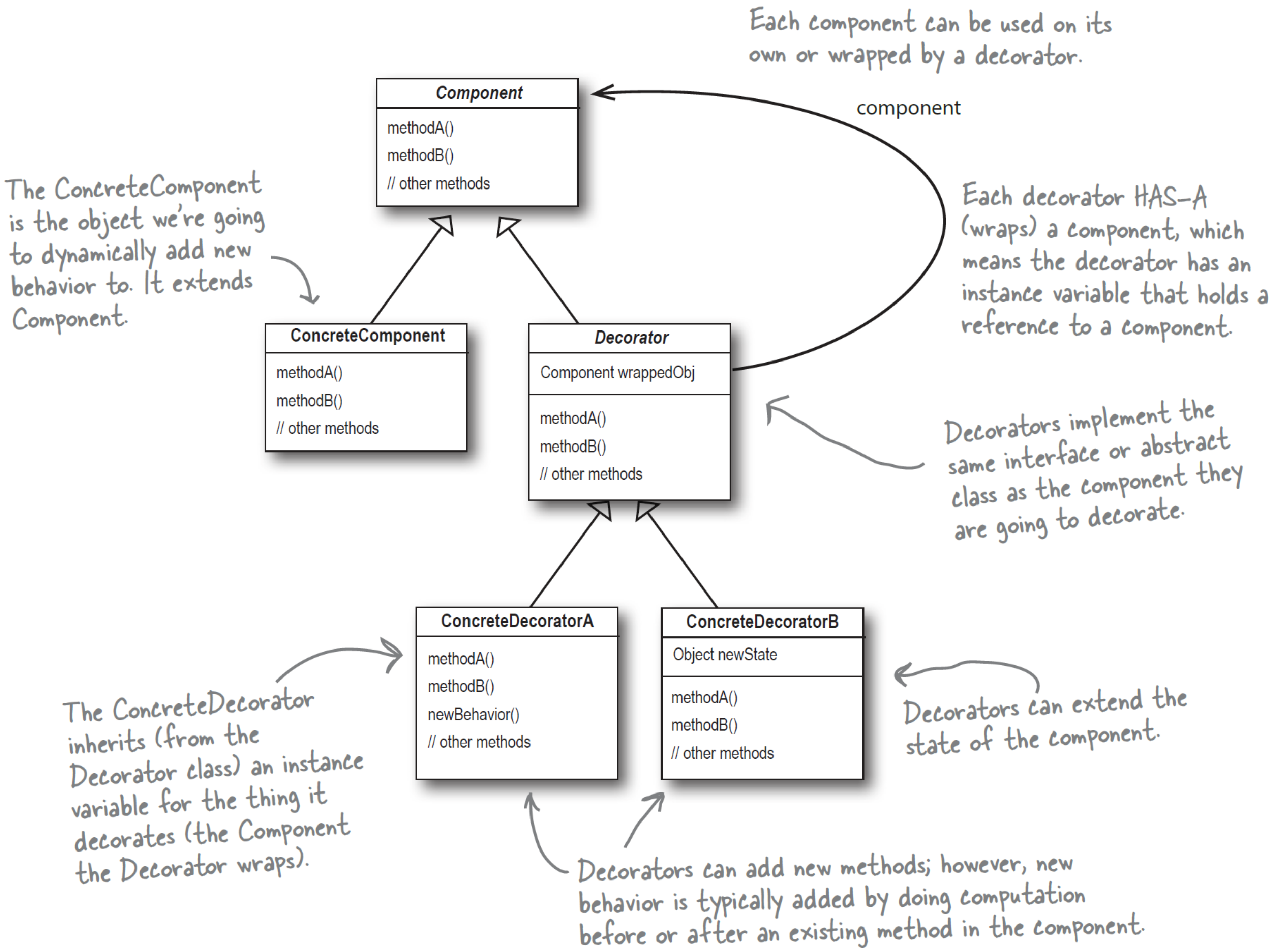
3 Mocha calls cost() on DarkRoast.

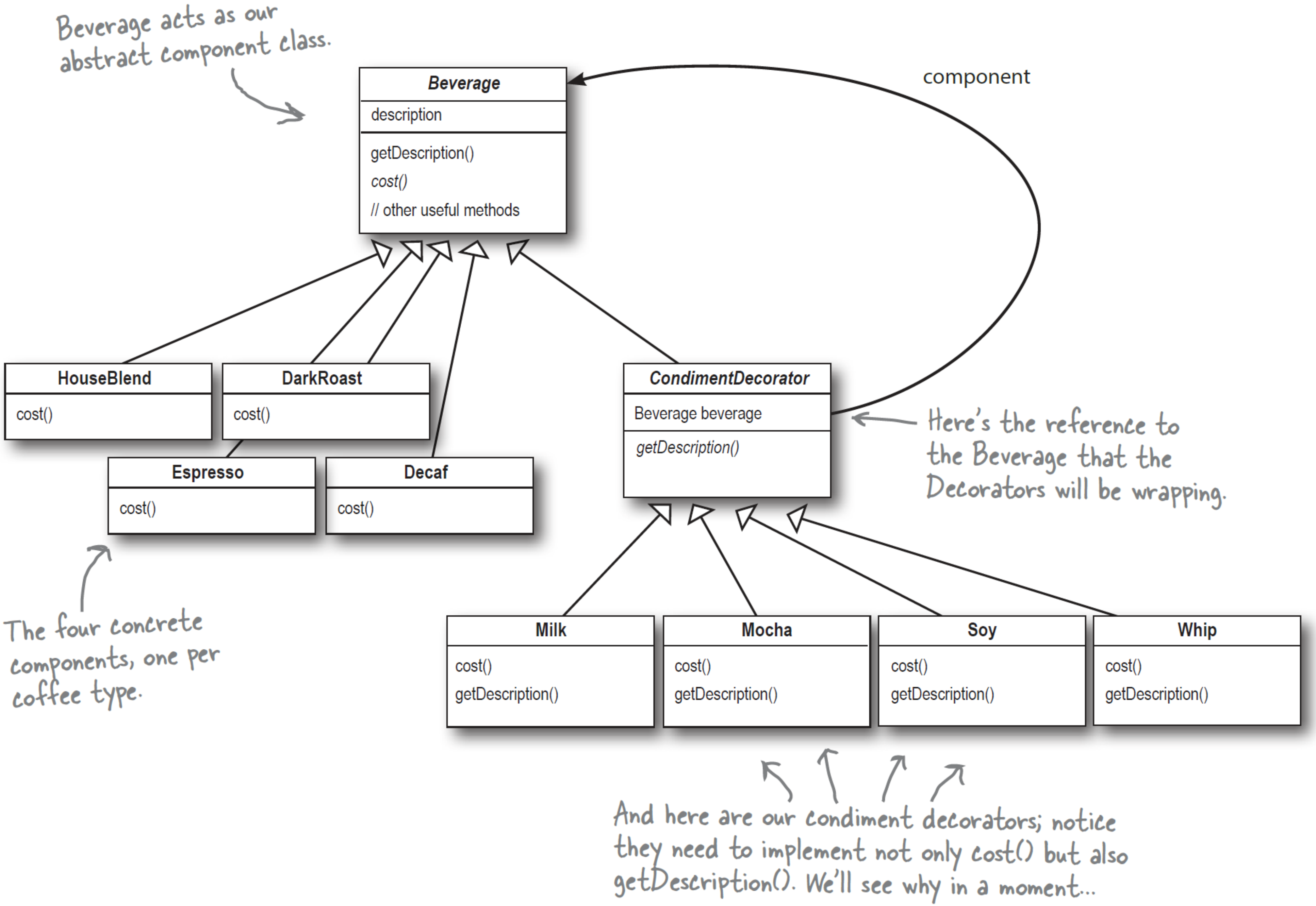


4 DarkRoast returns its cost, 99 cents.

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

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

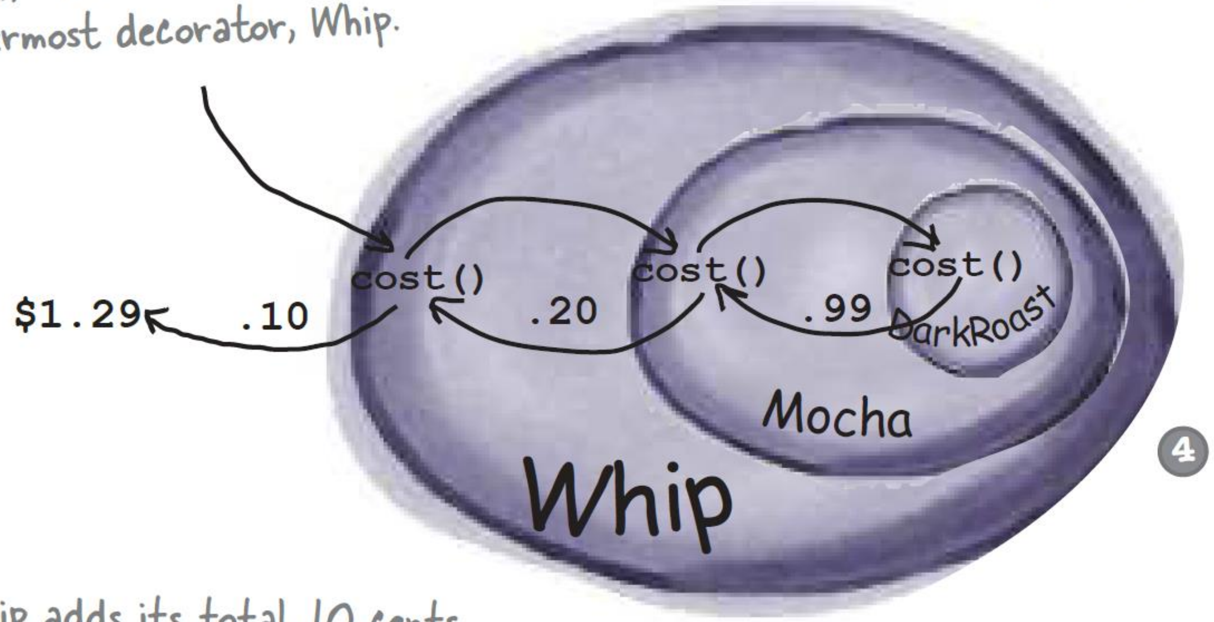

Each component can be used on its own or wrapped by a decorator.
component
Component
methodA()
methodB()
// other methods
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
Each decorator HAS-A (wraps) a component, which means the decorator has an 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
Decorators can extend the state of the component.
The ConcreteDecorator inherits (from the Decorator class) an instance variable for the thing it decorates (the Component the 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.


Beverage acts as our abstract component class.
component
Beverage
description
getDescription()
cost()
// other useful methods
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...

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

2 Whip calls cost() on Mocha.

3 Mocha calls cost() on DarkRoast.



This picture was for a "dark roast mocha whip" beverage.

4 DarkRoast returns its cost, 99 cents.

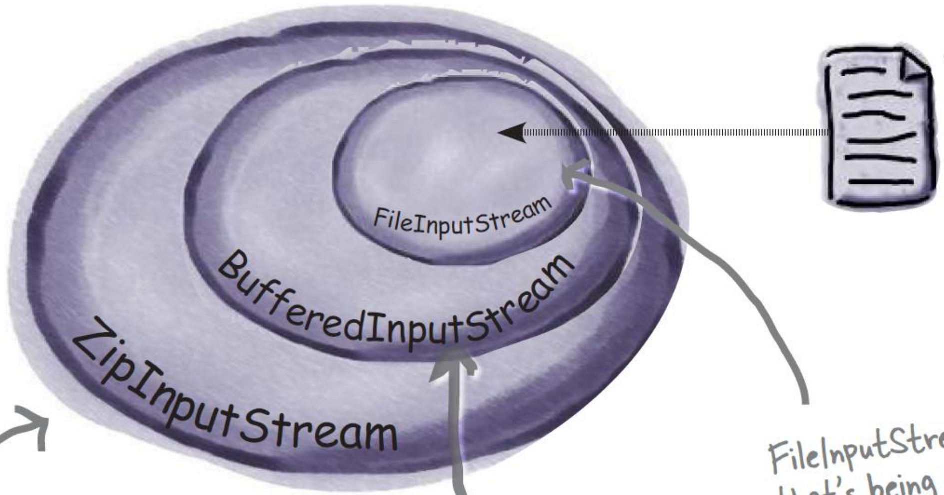
5 Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

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

Real-World Decorators: Java I/O

The large number of classes in the java.io package is...*overwhelming*. Don't feel alone if you said "whoa" the first (and second and third) time you looked at this API. But now that you know the Decorator Pattern, the I/O classes should make more sense since the java.io package is largely based on Decorator. Here's a typical set of objects that use decorators to add functionality to reading data from a file:

A text file for reading.



ZipInputStream is also a concrete decorator. It adds the ability to read zip file entries as it reads data from a zip file.

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, ByteArrayInputStream, and a few others. All of these give us a base component from which to read bytes.