CS525 Advanced Software Development

Lesson 4 – The Decorator Pattern

Design Patterns
Elements of Reusable Object-Oriented Software

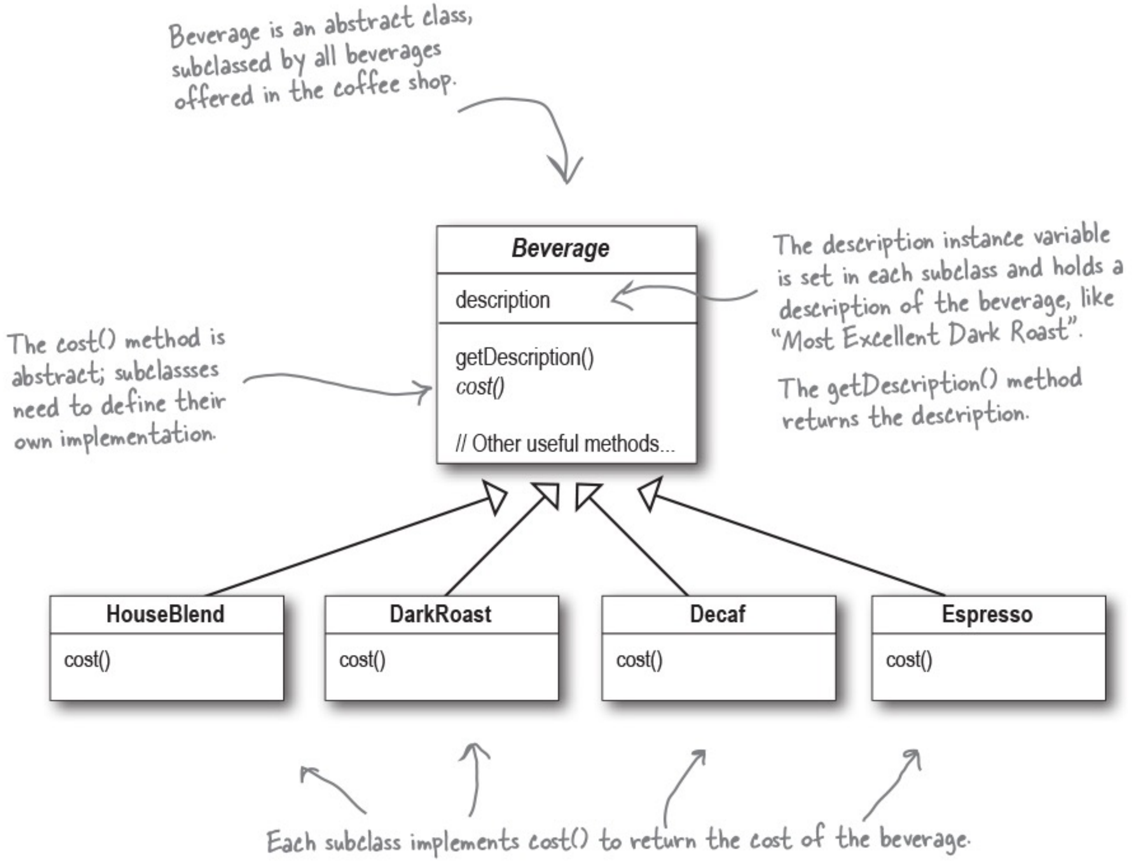
Payman Salek, M.S.
March 2022

Introduction

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.

Setting the stage (Starbuzz Coffee)



Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.

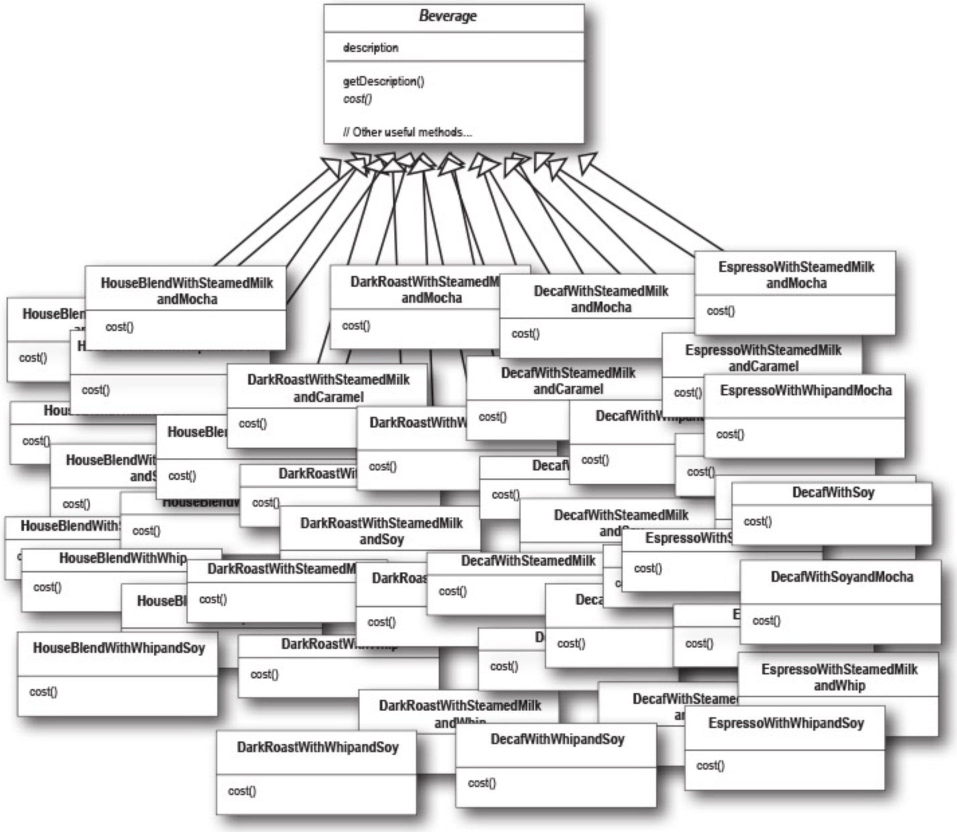
The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".

The getDescription() method returns the description.

The cost() method is abstract; subclassses need to define their own implementation.

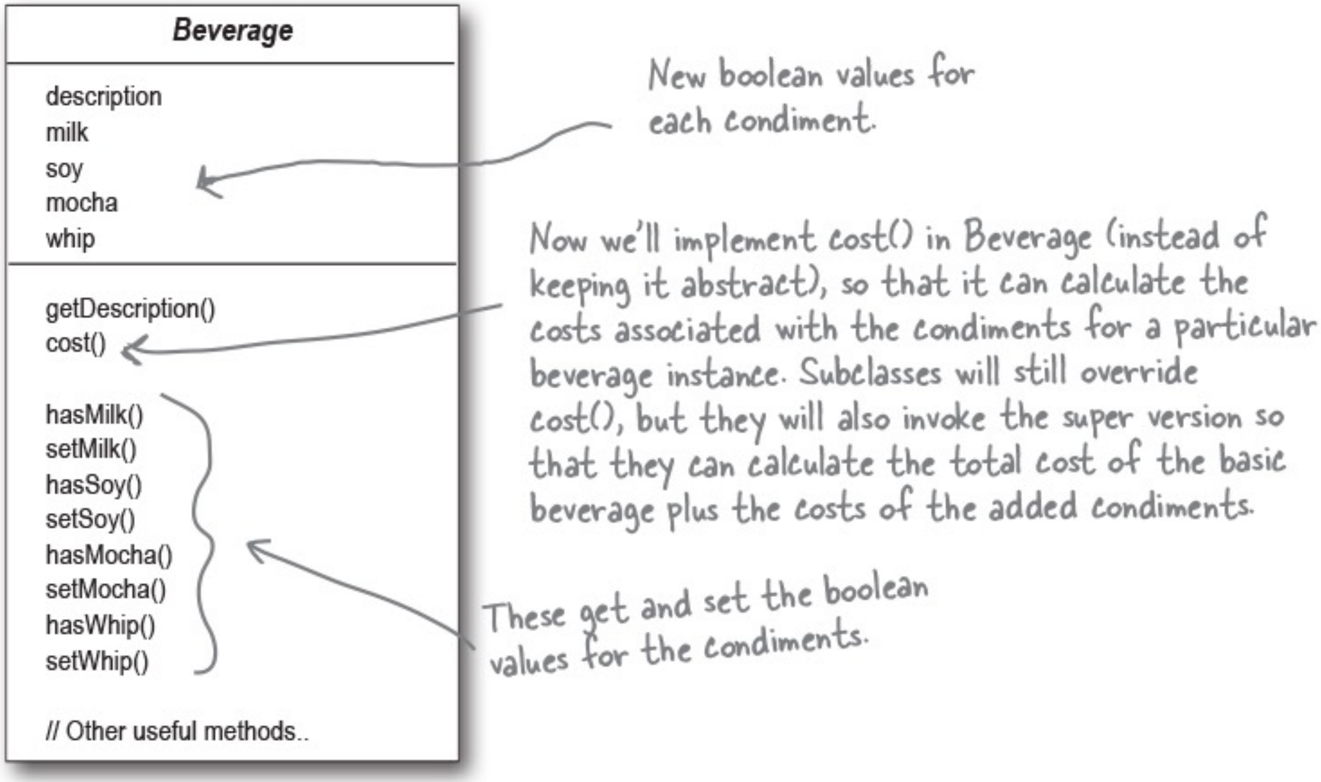
Each subclass implements cost() to return the cost of the beverage.

Subclass Explosion!


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
cost()
DecafWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andCaramel
cost()
EspressoWithWhipandMocha
cost()
DarkRoastWithSteamedMilk andCaramel
cost()
DecafWithSteamedMilk andCaramel
cost()
DecafWithSoy
cost()
DarkRoastWithSteamedMilk andSoy
DecafWithSteamedMilk
HouseBlendWithWhip
DarkRoastWithSteamedMilk
DecafWithSoyandMocha
cost()
HouseBlendWithWhipandSoy
cost()
EspressoWithSteamedMilk andWhip
DarkRoastWithSteamedMilk andWhip
DecafWithWhipandSoy
cost()
EspressoWithWhipandSoy
cost()
DarkRoastWithWhipandSoy
cost()

2/24/22

Instance Variables??



Beverage

description
milk
soy
mocha
whip

getDescription()
cost()

hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()
hasWhip()
setWhip()

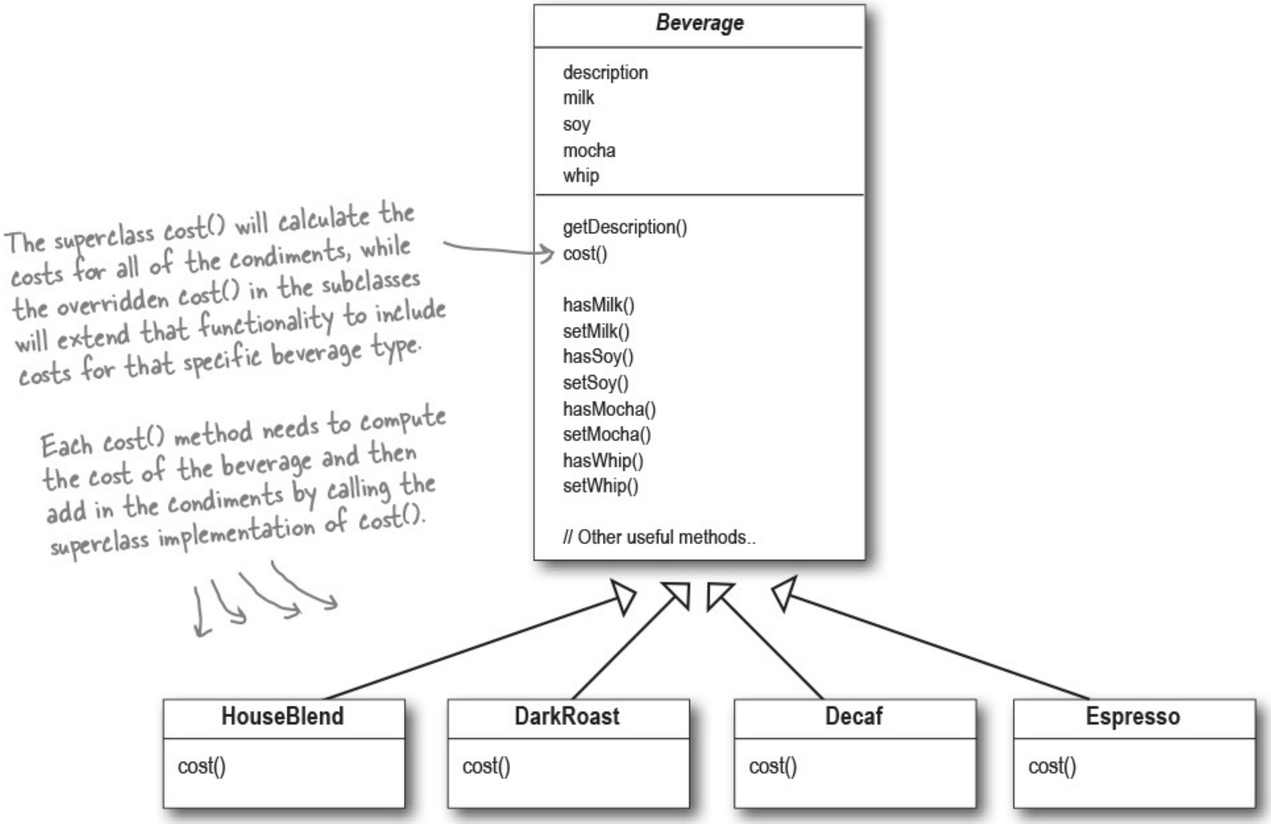
// Other useful methods..

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.

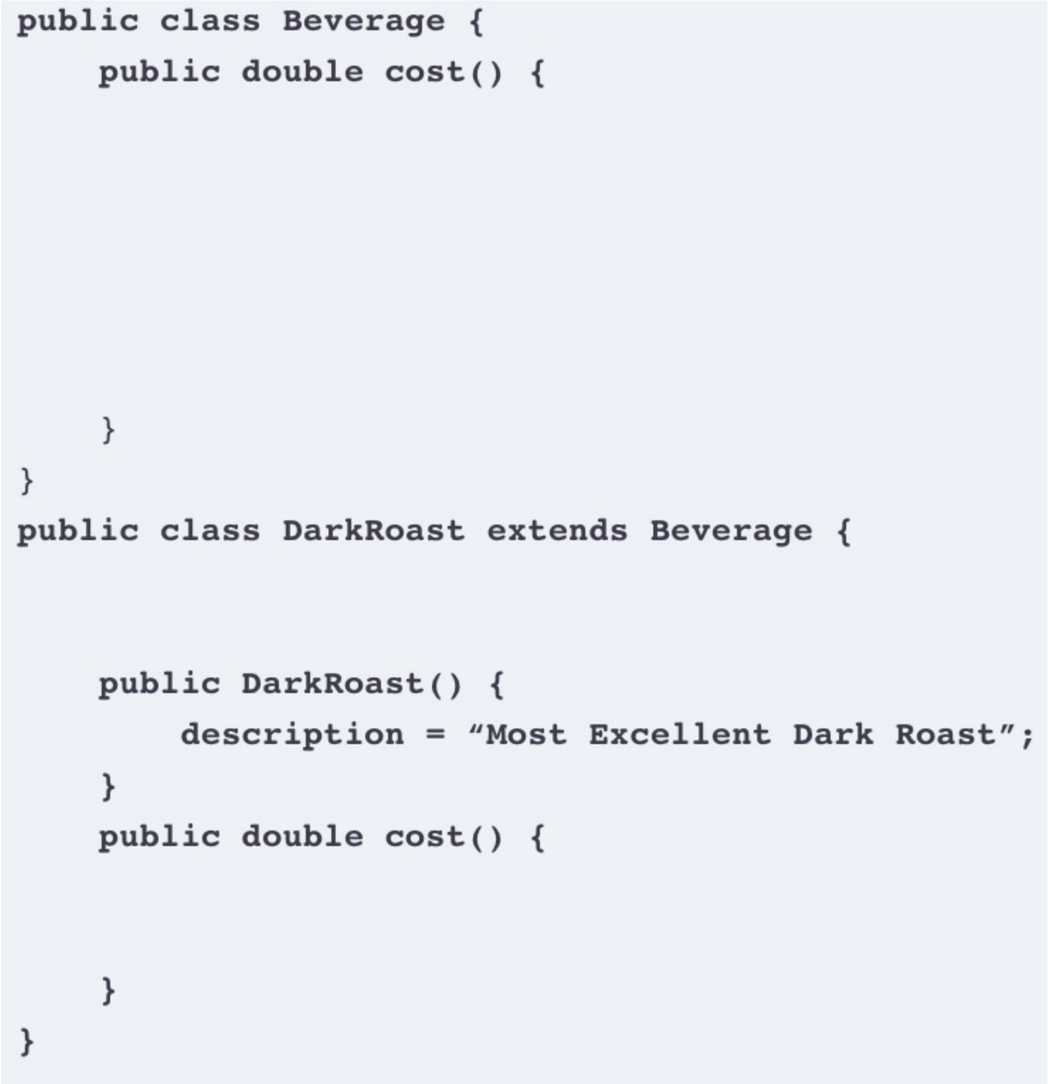
We have a solution??



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().

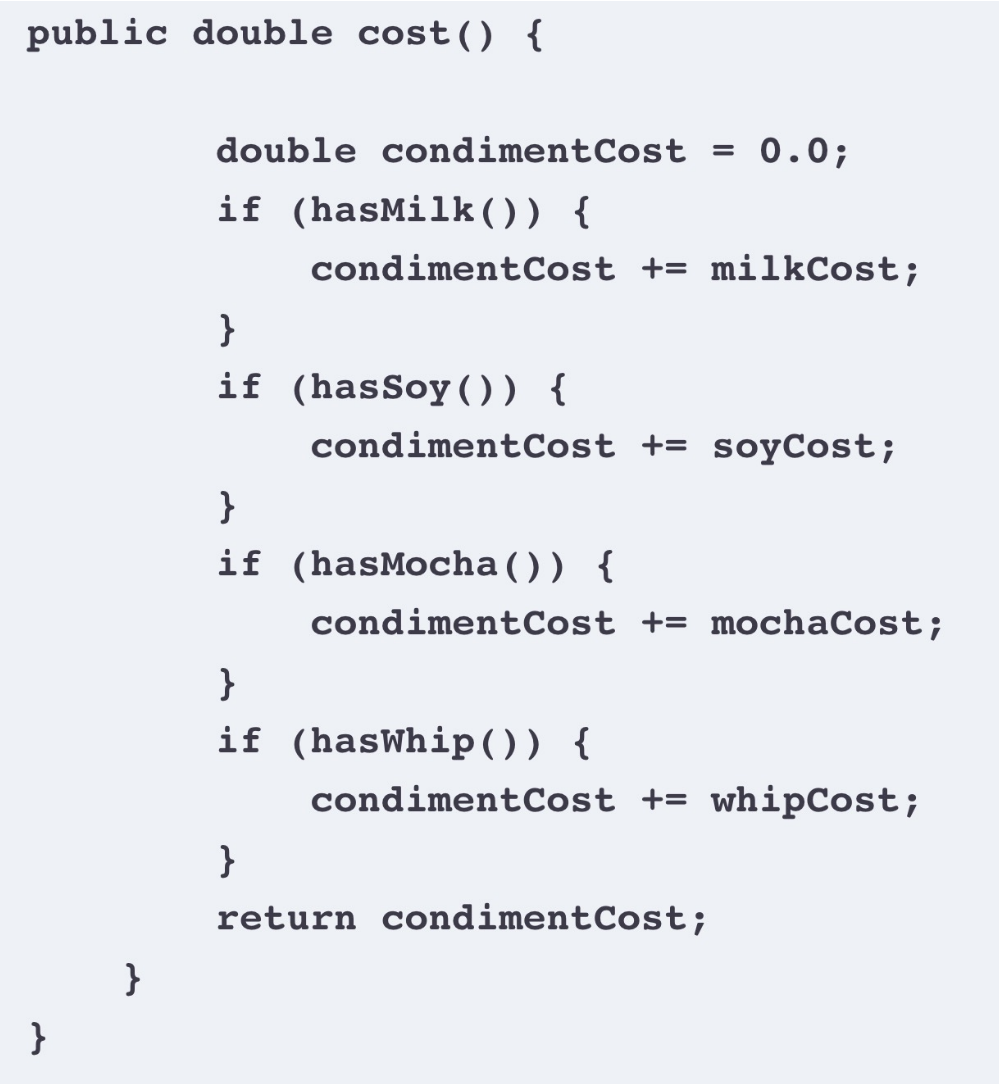
Practice



```
public class Beverage {
    public double cost() {







    }
}
public class DarkRoast extends Beverage {


    public DarkRoast() {
        description = "Most Excellent Dark Roast";
    }
    public double cost() {



    }
}
```

Practice

2/24/22



```
public double cost() {

        double condimentCost = 0.0;
        if (hasMilk()) {
            condimentCost += milkCost;
        }
        if (hasSoy()) {
            condimentCost += soyCost;
        }
        if (hasMocha()) {
            condimentCost += mochaCost;
        }
        if (hasWhip()) {
            condimentCost += whipCost;
        }
        return condimentCost;
    }
}
```

Practice

```
public class DarkRoast extends Beverage {

    public DarkRoast() {
        description = “Most Excellent Dark Roast”;
    }

    public double cost() {
        return 1.99 + super.cost();
    }
}
```

Open-Closed Principle

“Classes should be open for extension but closed for modification.”

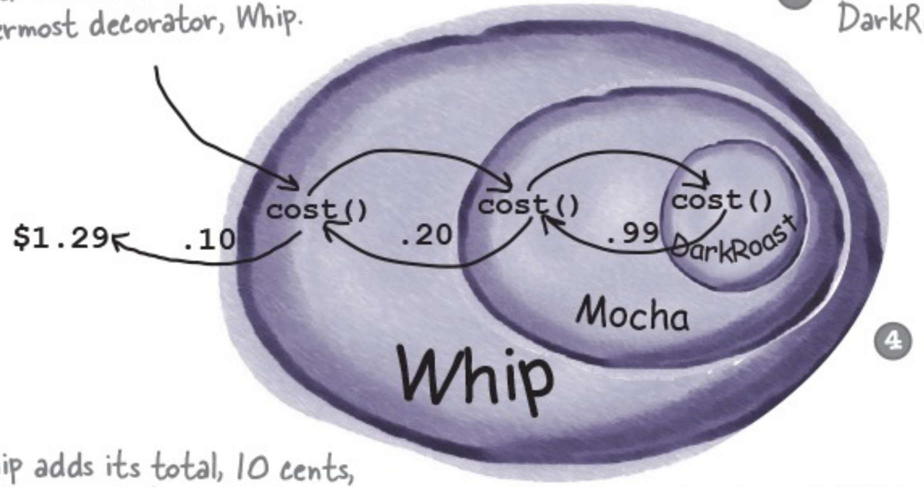
Open Means:

Feel free to extend our classes with any new behavior you like. If your needs or requirements change (and we know they will), just go ahead and make your own extensions.

Closed Means:

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. If you don’t like it, you can speak to the manager.

Solution: The Decorator Pattern



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

❷ Whip calls cost() on Mocha.

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

❸ Mocha calls cost() on DarkRoast.

❹ DarkRoast returns its cost, 99 cents.

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

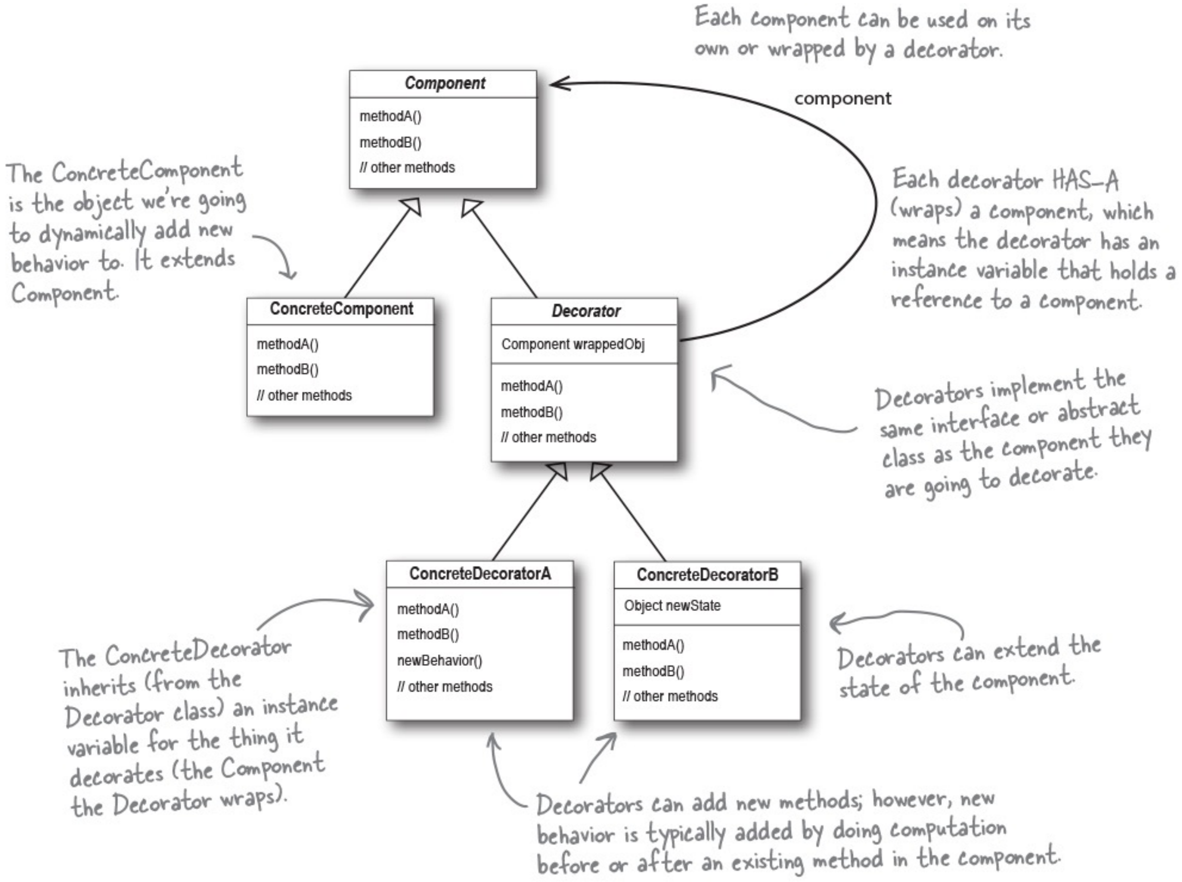
❺ Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

The Decorator Pattern

attaches additional responsibilities to an object dynamically.

Decorators provide a flexible alternative to subclassing for extending functionality.

Classic UML for Decorator



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.

ConcreteComponent

methodA()

methodB()

// other methods

Decorator

Component wrappedObj

methodA()

methodB()

// other methods

Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.

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()

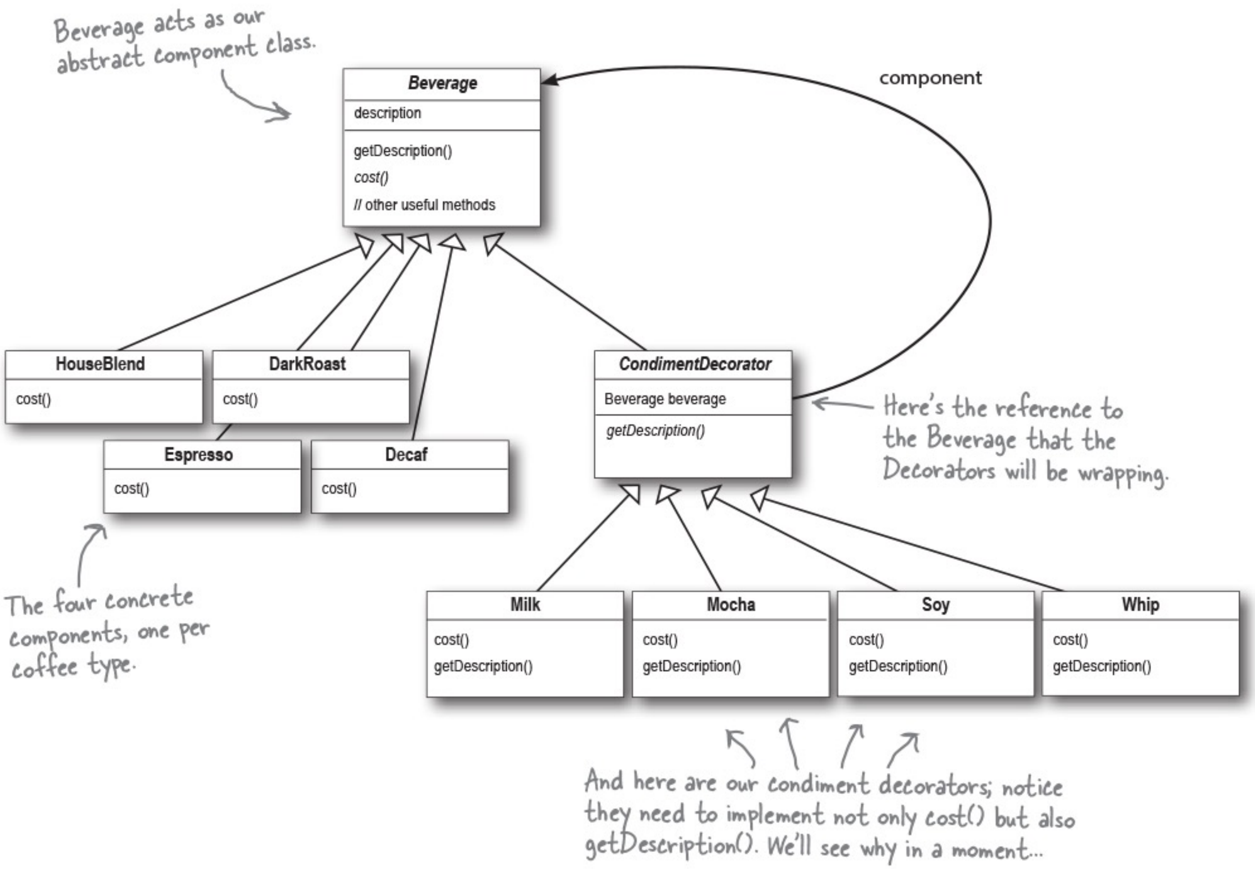
// other methods

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.

Finished Solution



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()

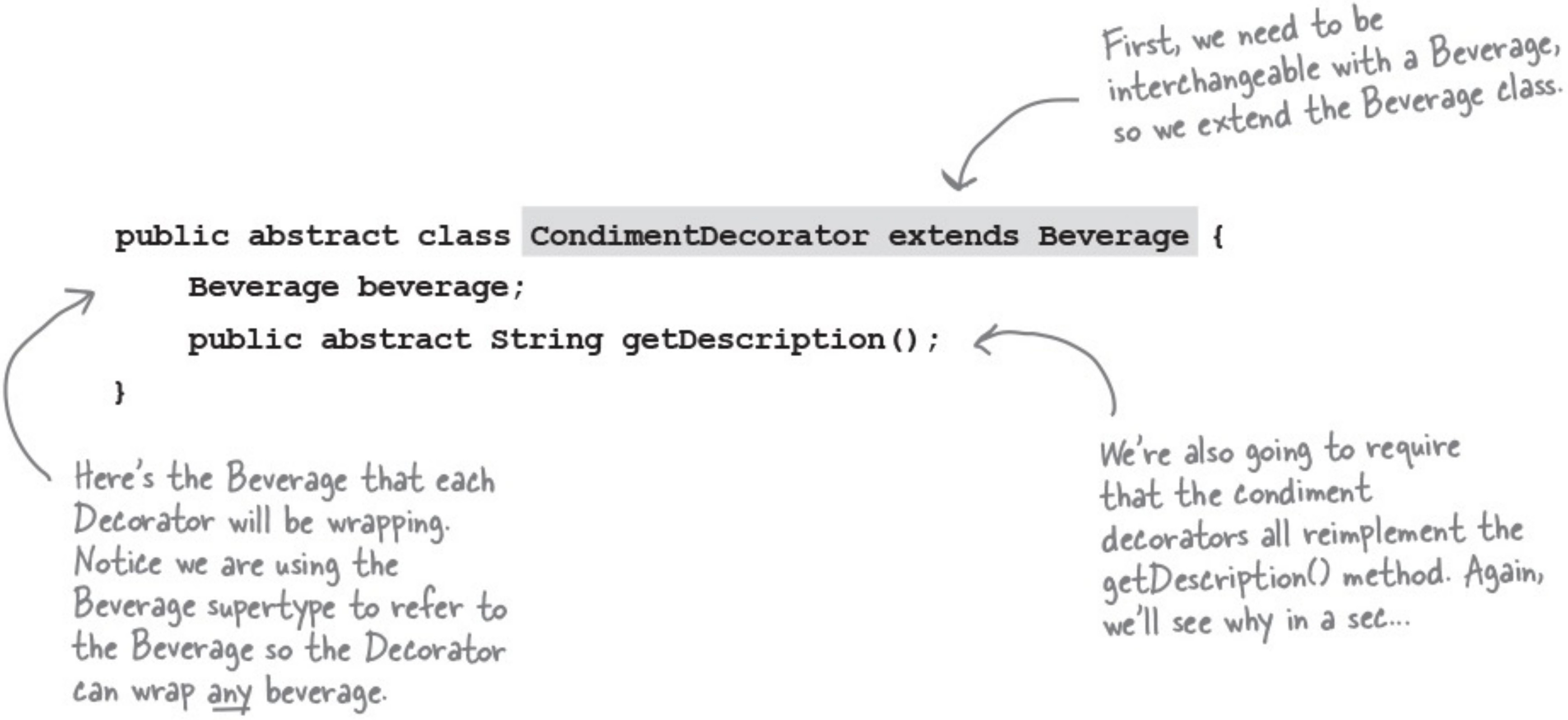
Mocha
cost()
getDescription()

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...

Coding the Condiments



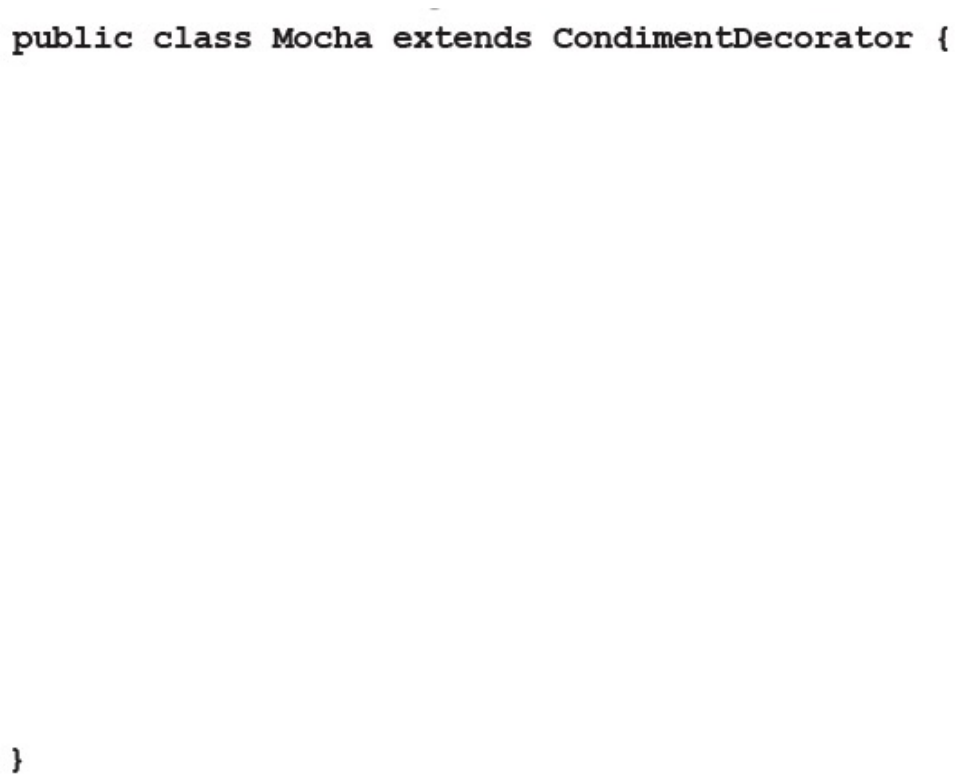
First, we need to be interchangeable with a Beverage, so we extend the Beverage class.

```
public abstract class CondimentDecorator extends Beverage {
    Beverage beverage;
    public abstract String getDescription();
}
```

Here's the Beverage that each Decorator will be wrapping. Notice we are using the Beverage supertype to refer to the Beverage so the Decorator can wrap *any* beverage.

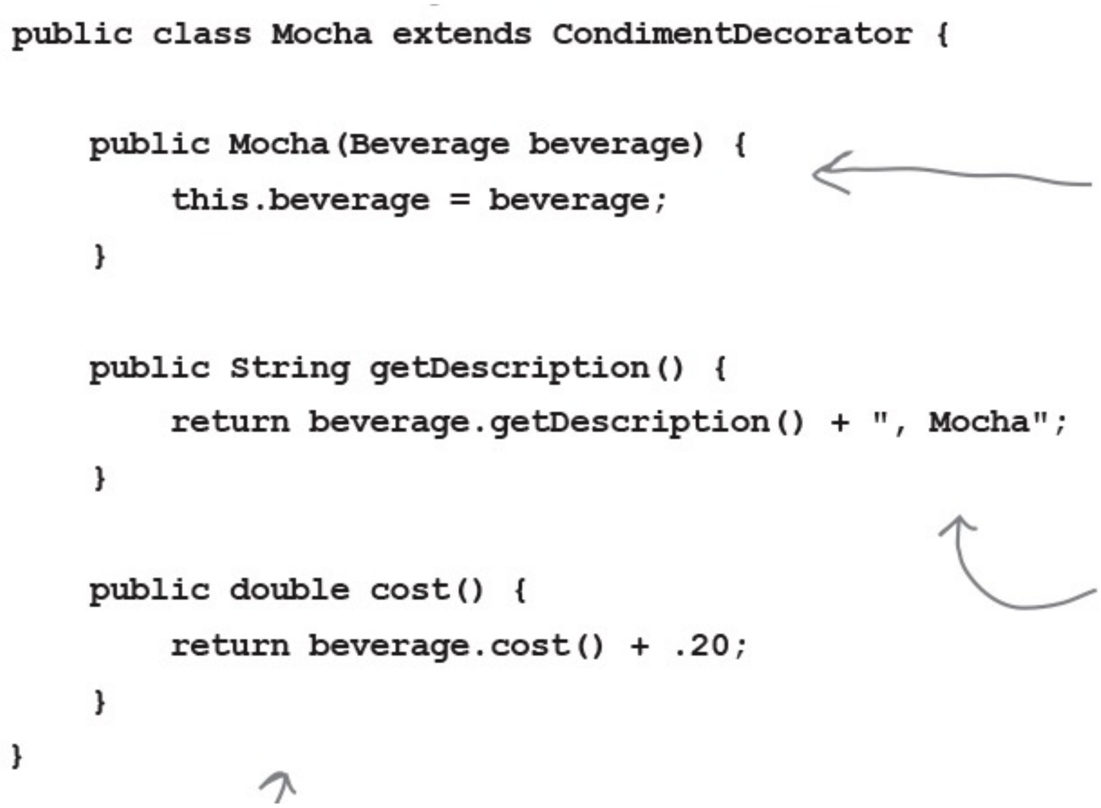
We're also going to require that the condiment decorators all reimplement the getDescription() method. Again, we'll see why in a sec...

Quiz: Coding the Condiments



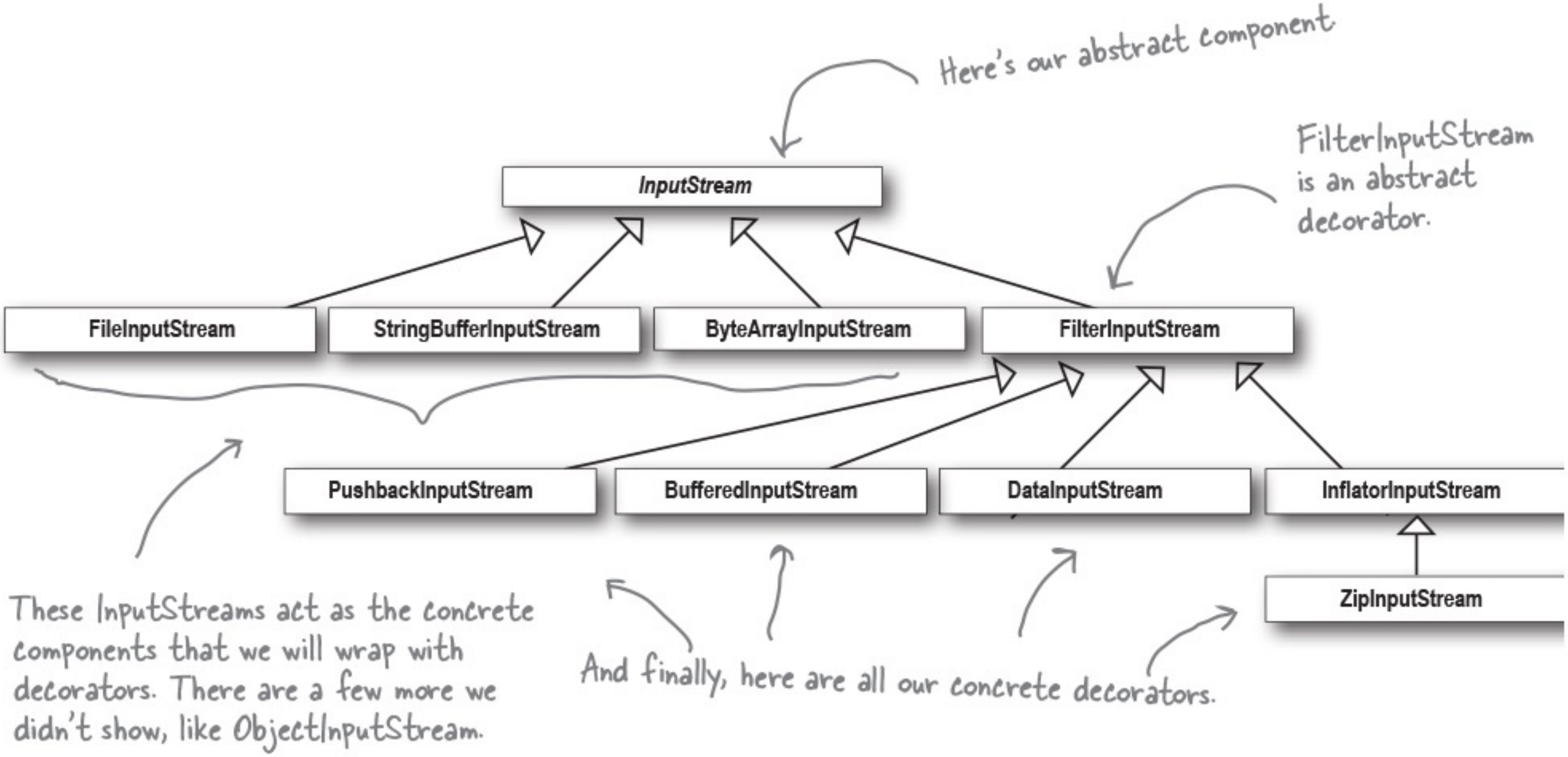
```
public class Mocha extends CondimentDecorator {








}
```

Quiz: Coding the Condiments

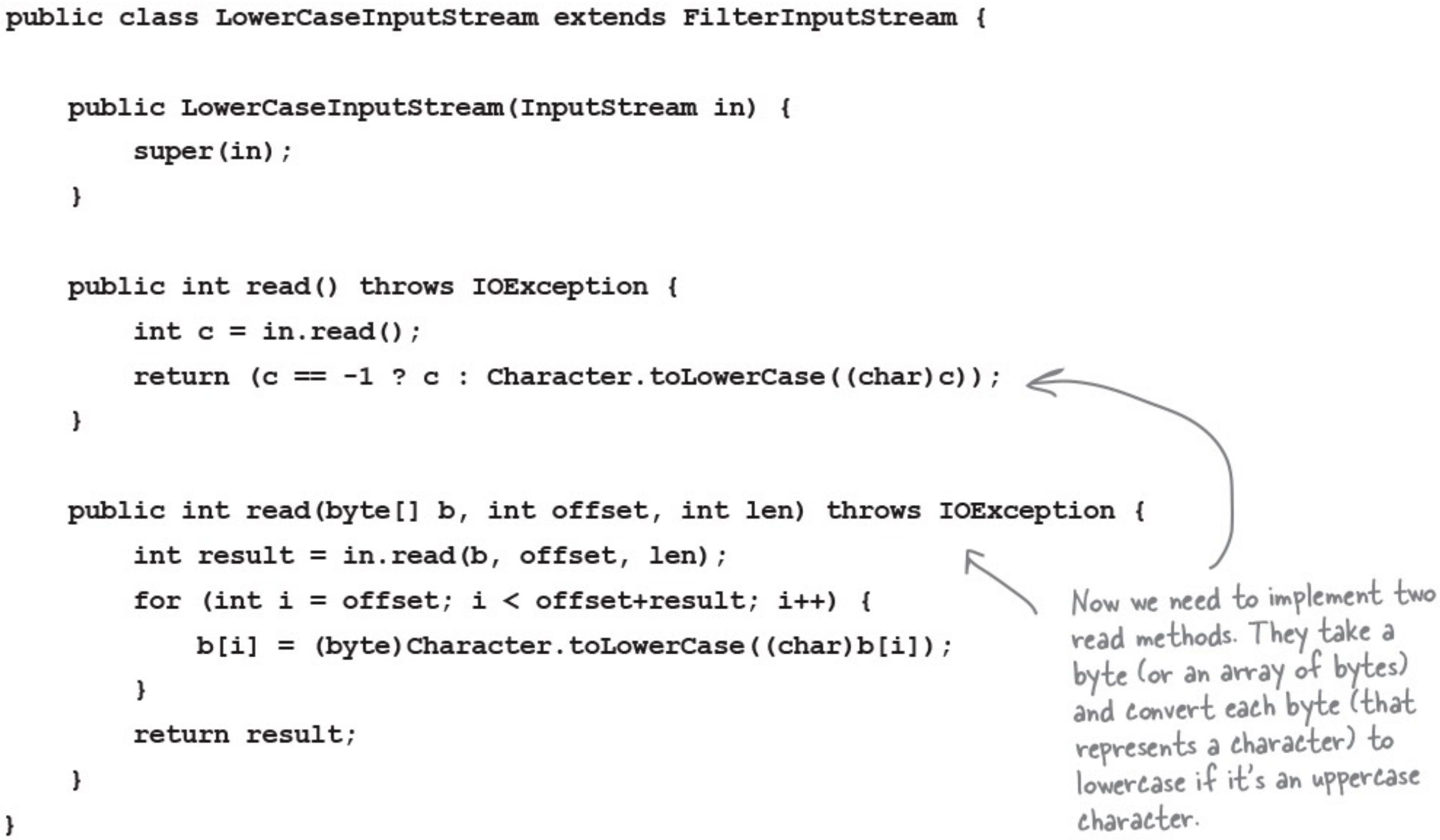


```
public class Mocha extends CondimentDecorator {

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return beverage.cost() + .20;
    }
}
```

Decorator Example from java.io


Here's our abstract component
InputStream
FilterInputStream is an abstract decorator.
FileInputStream
StringBufferInputStream
ByteArrayInputStream
FilterInputStream
PushbackInputStream
BufferedInputStream
DataInputStream
InflatorInputStream
ZipInputStream
These InputStreams act as the concrete components that we will wrap with decorators. There are a few more we didn't show, like ObjectInputStream.
And finally, here are all our concrete decorators.

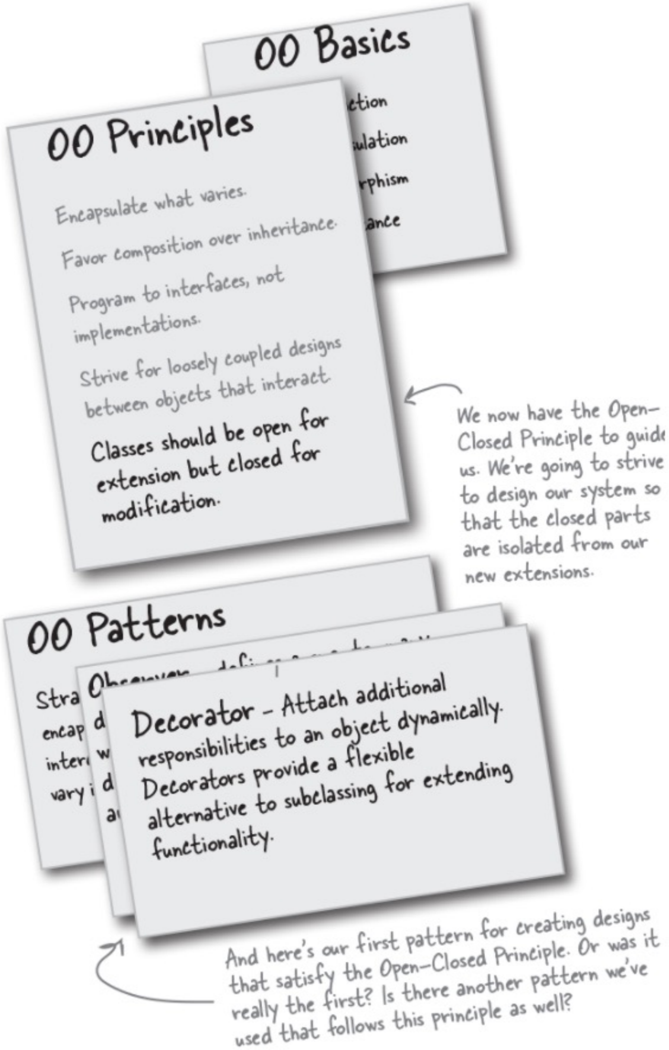
Extending the java.io Decorator



```
public class LowerCaseInputStream extends FilterInputStream {

    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = in.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = in.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

Now we need to implement two read methods. They take a byte (or an array of bytes) and convert each byte (that represents a character) to lowercase if it's an uppercase character.

Summary



OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

We now have the Open–Closed Principle to guide us. We're going to strive to design our system so that the closed parts are isolated from our new extensions.

OO Patterns

Decorator - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

And here's our first pattern for creating designs that satisfy the Open–Closed Principle. Or was it really the first? Is there another pattern we've used that follows this principle as well?

2/24/22