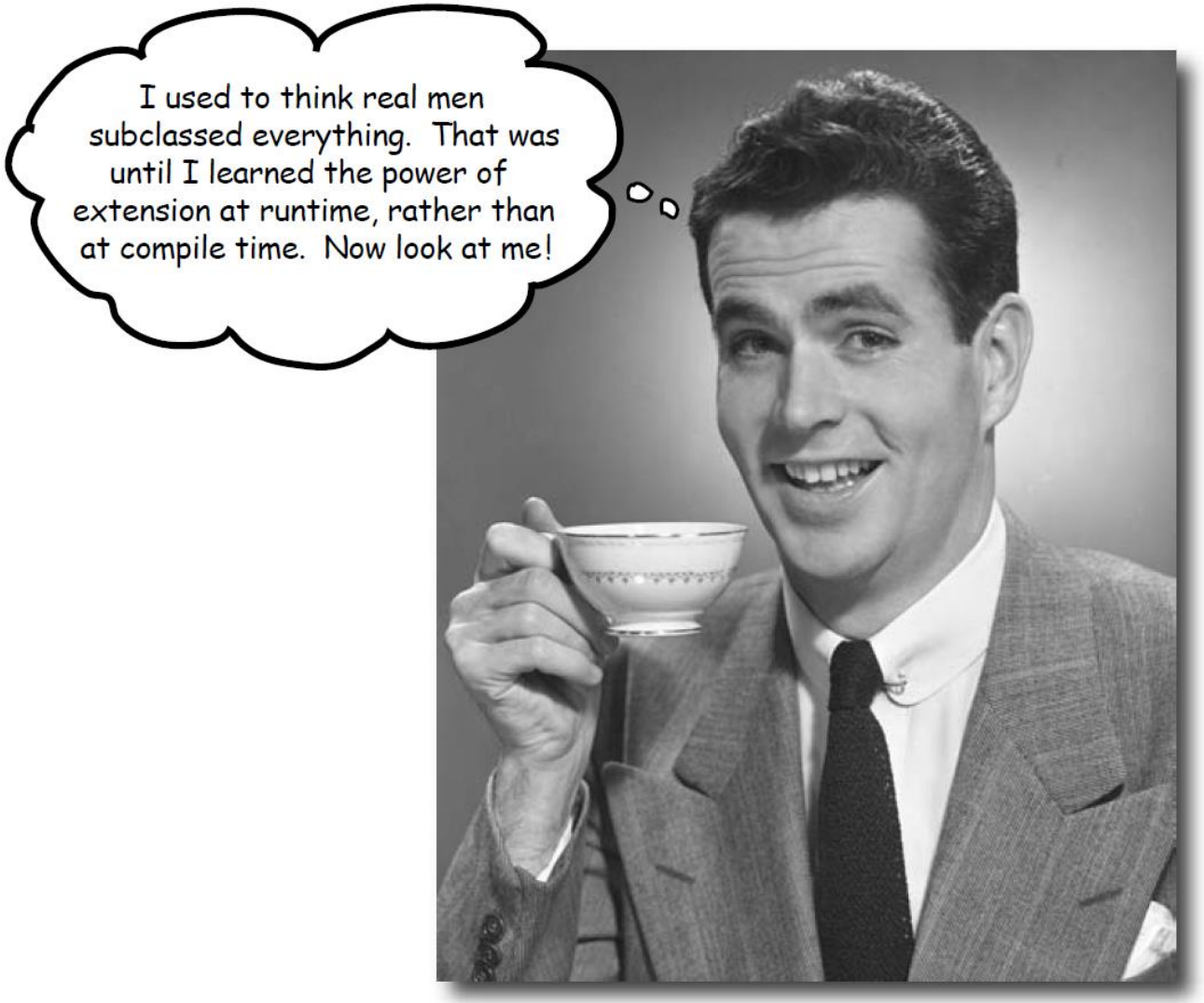
BIM492 Design Patterns



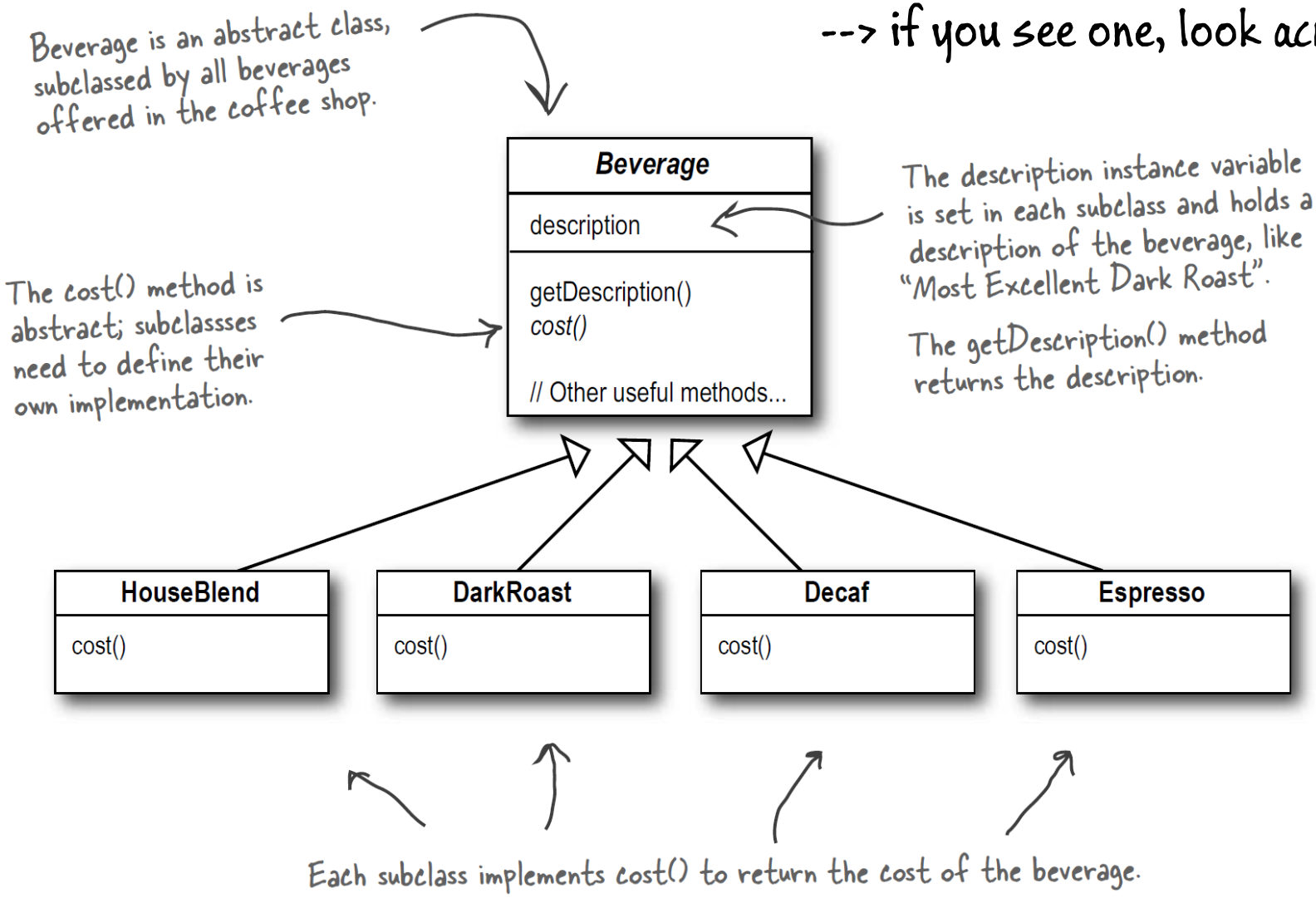
03. Decorator Pattern

Design Eye
for the Inheritance Guy

Welcome to Starbuzz Coffee

Starbuzz Coffee is the fastest growing coffee shop around

--> if you see one, look across the street, you'll see another one :]



They need to update their ordering system to match beverage offerings.

Their initial design for beverages <-- is like this...

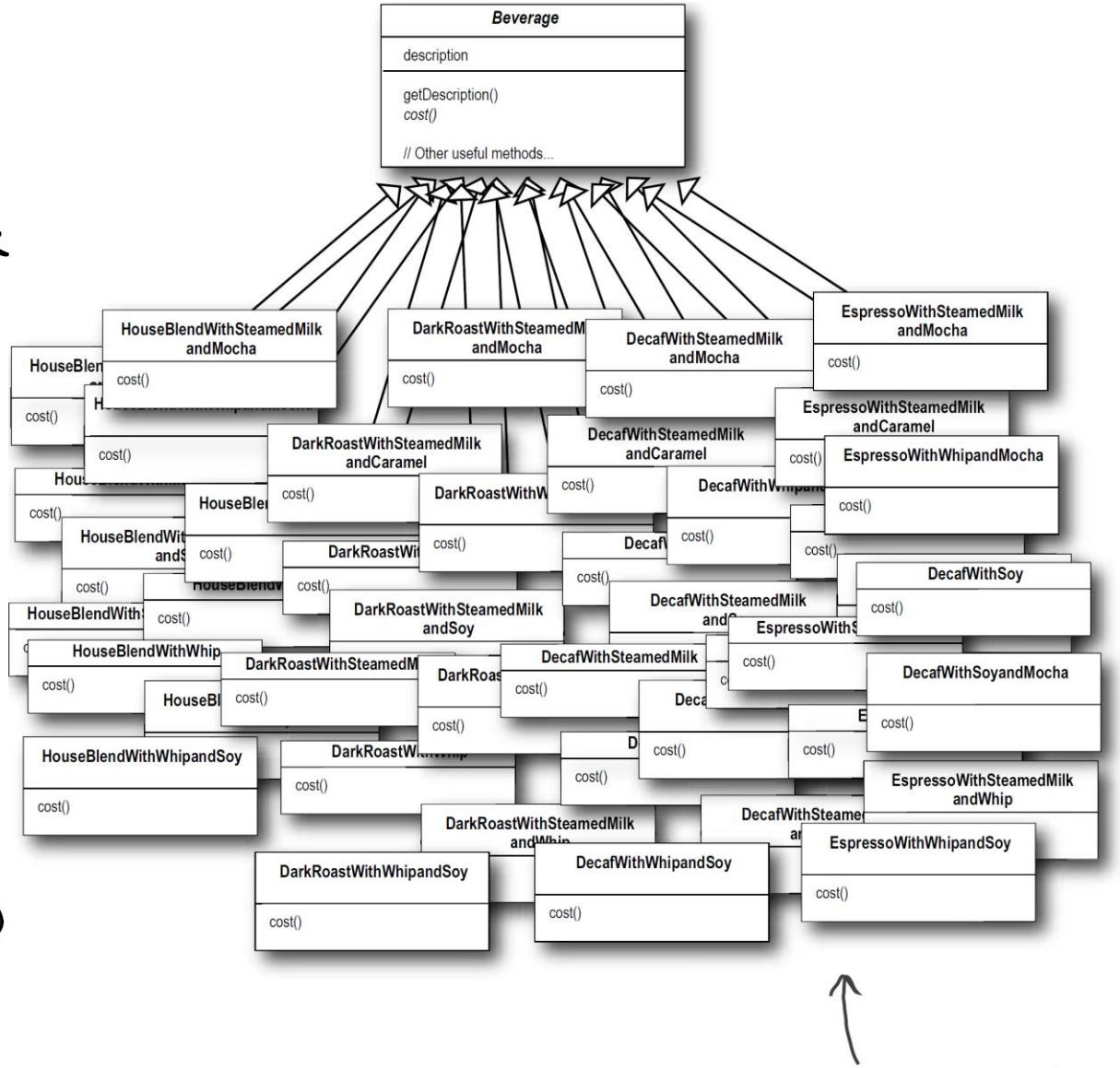
aannnd condiments, the best..

What is best with Starbuzz is you can ask for several condiments for any beverage:

- steamed milk
- soy
- mocha (known as chocolate)
- all topped with whipped milk

Starbuzz charges a bit for each of these.

Ok, let's build options...



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

Power up your head - violating design principles

It's pretty obvious that Starbuzz has created a maintenance nightmare for themselves.

? What happens when the price of milk goes up?

? What do they do when they add a new caramel topping?

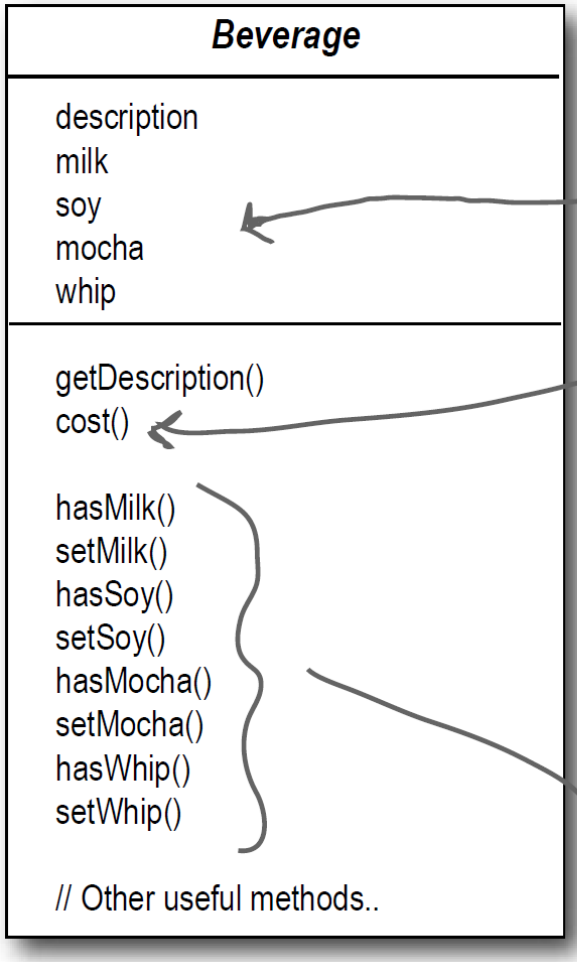
Thinking beyond the maintenance problem, which of the design principles that we've covered so far are they violating?

Hint: they're violating two of them in a big way!

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.

Sharpen your pencil

Write the `cost()` methods for the following classes (pseudo-Java is okay):

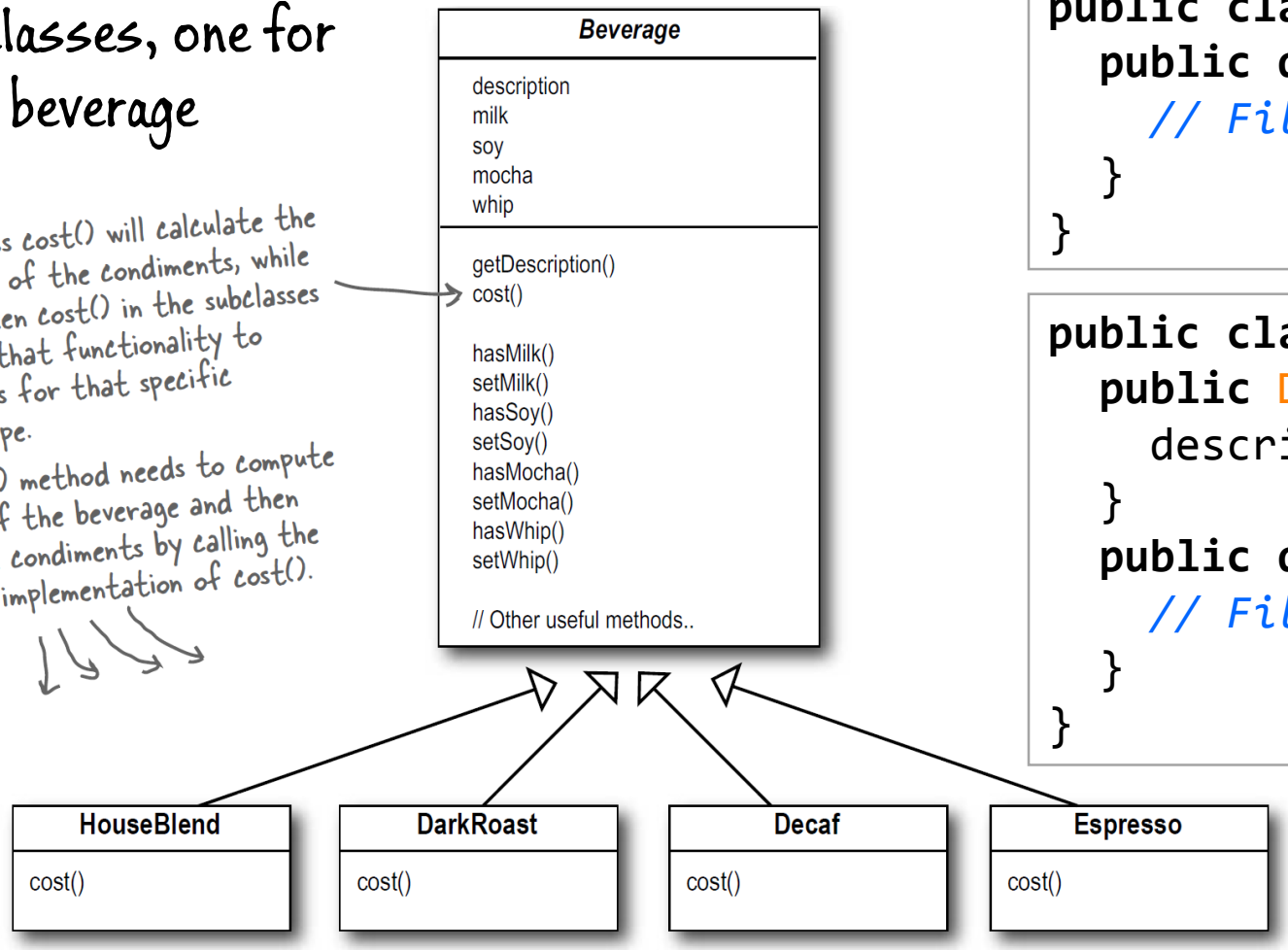
```
public class Beverage {
  public double cost() {
    // Fill this in
  }
}
```

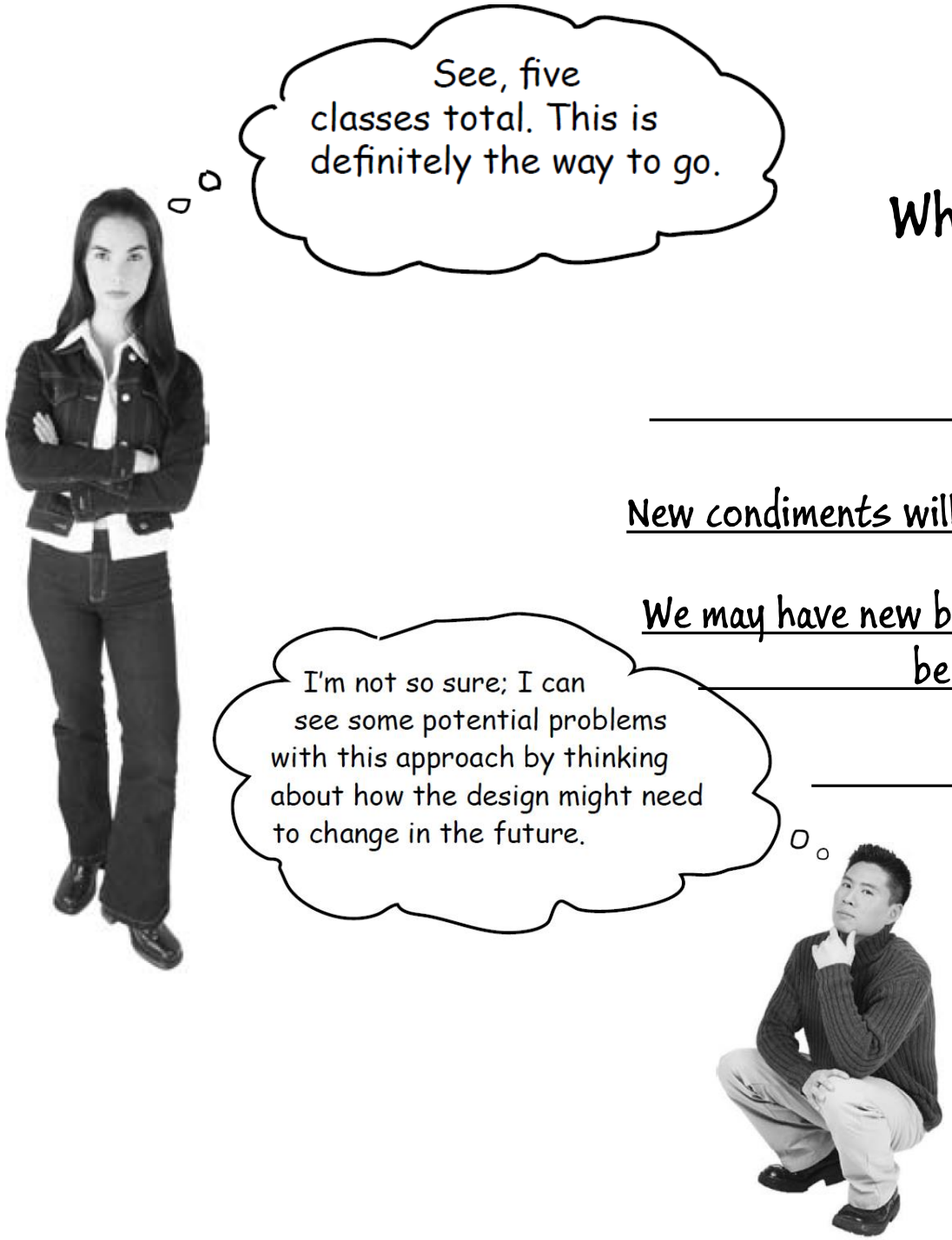
```
public class DarkRoast extends Beverage {
  public DarkRoast() {
    description = "Excellent Dark Roast";
  }
  public double cost() {
    // Fill this in
  }
}
```

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

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().





Sharpen your pencil

What requirements or other factors might change that will impact this design?

Price changes for condiments will force us to alter existing code

New condiments will force us to add new methods and alter the cost method in the superclass

We may have new beverages. For some of these beverages (ice tea?), the condiments may not be approriate , yet the Tea subclass will still inherit methods like hasWhip().

What if a customer wants a double mocha?

Master and Grasshopper

Master: Grasshopper, it has been some time since our last meeting. Have you been deep in meditation on inheritance?

Student: Yes, Master. While inheritance is powerful, I have learned that it doesn't always lead to the most flexible or maintainable designs.

Master: Ah yes, you have made some progress. So, tell me my student, how then will you achieve reuse if not through inheritance?

Student: Master, I have learned there are ways of "inheriting" behavior at runtime through composition and delegation.

Master: Please, go on...

Student: When I inherit behavior by subclassing, that behavior is set statically at compile time. In addition, all subclasses must inherit the same behavior. If however, I can extend an object's behavior through composition, then I can do this dynamically at runtime.

Master: Very good, Grasshopper, you are beginning to see the power of composition.

Student: Yes, it is possible for me to add multiple new responsibilities to objects through this technique, including responsibilities that were not even thought of by the designer of the superclass. And, I don't have to touch their code!

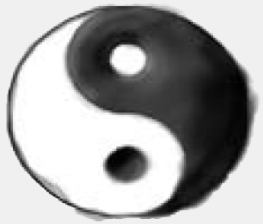
Master: What have you learned about the effect of composition on maintaining your code?

Student: Well, that is what I was getting at. By dynamically composing objects, I can add new functionality by writing new code rather than altering existing code. Because I'm not changing existing code, the chances of introducing bugs or causing unintended side effects in pre-existing code are much reduced.

Master: Very good. Enough for today, Grasshopper. I would like for you to go and meditate further on this topic... Remember, code should be closed (to change) like the lotus flower in the evening, yet open (to extension) like the lotus flower in the morning.

The Open-Closed Principle

Grashopper is on to one of the most important design principles



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 (and we know they will), just go ahead and make your own extensions.

Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code.

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

there are no Dumb Questions

Q: Open for extension and closed for modification? That sounds very contradictory. How can a design be both?

A: That's a very good question. It certainly sounds contradictory at first. After all, the less modifiable something is, the harder it is to extend, right? As it turns out, though, there are some clever OO techniques for allowing systems to be extended, even if we can't change the underlying code. Think about the Observer Pattern... By adding new Observers, we can extend the Subject at any time, without adding code to the Subject. You'll see quite a few more ways of extending behavior with other OO design techniques.

Q: Okay, I understand Observable, but how do I generally design something to be extensible, yet closed for modification?

A: Many of the patterns give us time tested designs that protect your code from being modified by supplying a means of extension. You'll see a good example of using the Decorator pattern to follow the Open-Closed principle.

While it may seem like a contradiction, there are techniques for allowing code to be extended without direct modification.

there are no Dumb Questions

Q: How can I make every part of my design follow the Open-Closed Principle?

A: Usually, you can't. Making OO design flexible and open to extension without the modification of existing code takes time and effort. In general, we don't have the luxury of tying down every part of our designs (and it would probably be wasteful). Following the Open-Closed Principle usually introduces new levels of abstraction, which adds complexity to our code. You want to concentrate on those areas that are most likely to change in your designs and apply the principles there.

Q: How do I know which areas of change are more important?

A: That is partly a matter of experience in designing OO systems and also a matter of the knowing the domain you are working in. Looking at other examples will help you learn to identify areas of change in your own designs.

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

Meet the Decorator Pattern

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?

Okay, we've seen beverage + condiment pricing hasn't worked out very well with inheritance.

We got class explosions, rigid designs, or we add functionality to the base class that isn't appropriate for some of the subclasses.

So, here's what we'll do instead:

Start with a beverage and "decorate" it with the condiments at runtime.

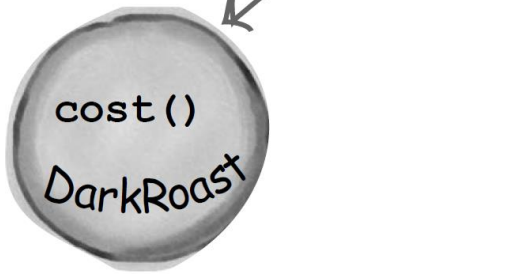
For example; if a customer wants a *Dark Roast* with *Mocha* and *Whip*

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

? How do you "decorate" an object, and how does delegation come into this?

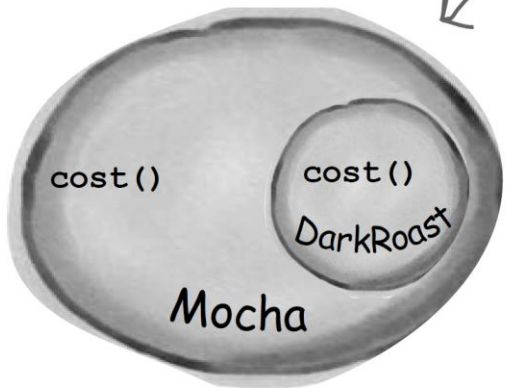
Constructing a drink order with Decorators

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.

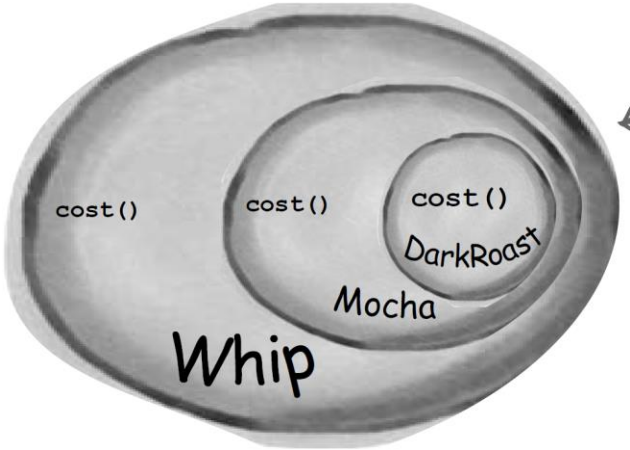
2. We create a Mocha object and Wrap it around the DarkRoast



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..)

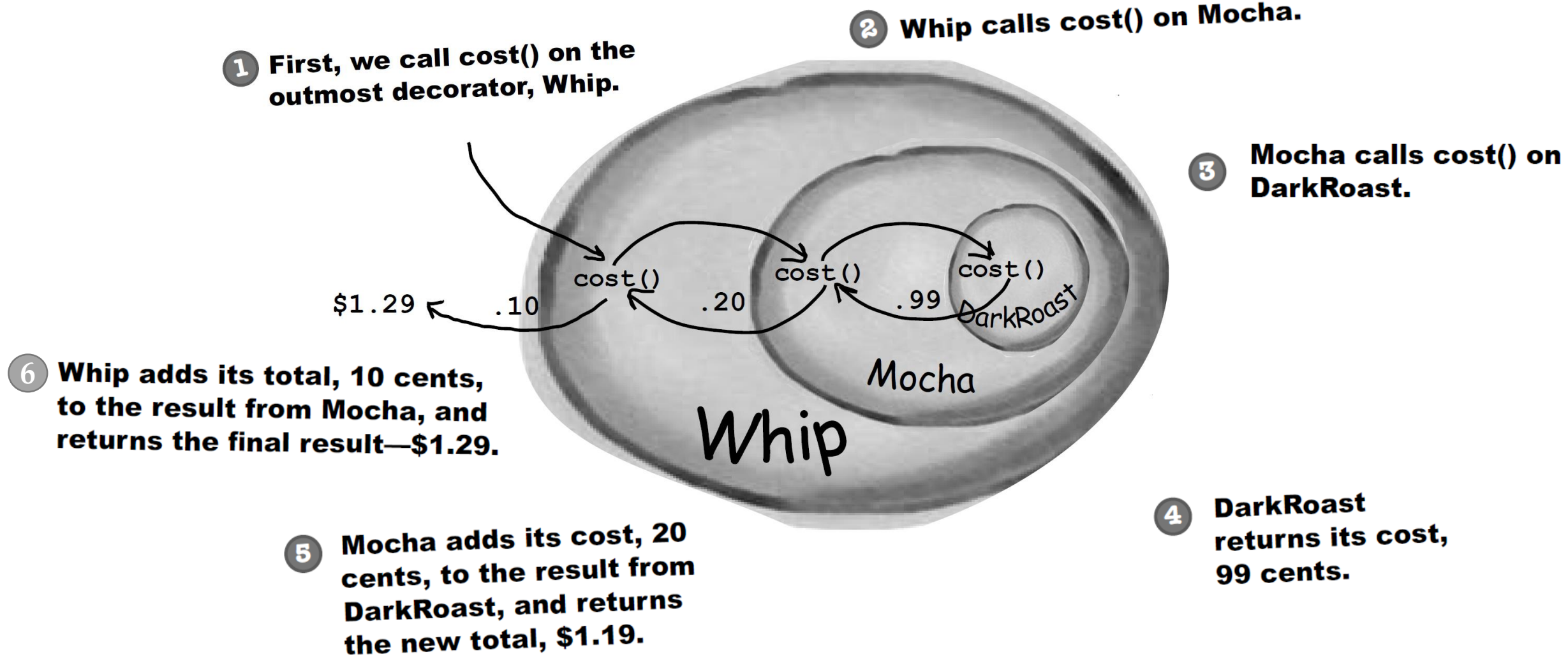
So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

3. We also 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. It's time to compute cost for the Customer. We do this by calling `cost()` on the outermost decorator, Whip, and Whip is going to delegate computing the cost of the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.



Okay, here's what we know so far...

- 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 either before and/or after delegating to the object it decorates to do the rest of the job.

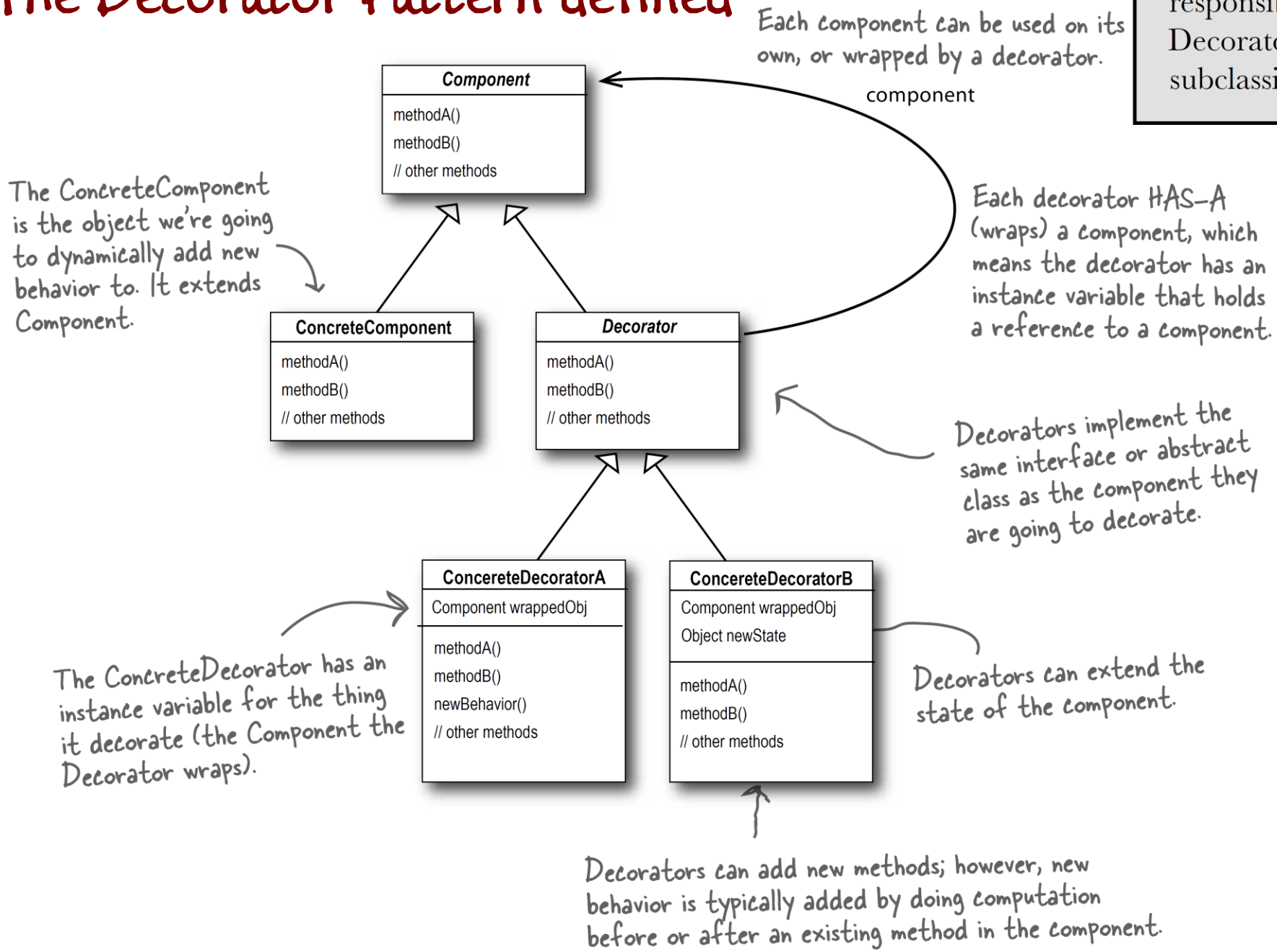
Key point!

- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

Now let's see how this all really works by looking at the Decorator Pattern definition and writing some code

The Decorator Pattern defined

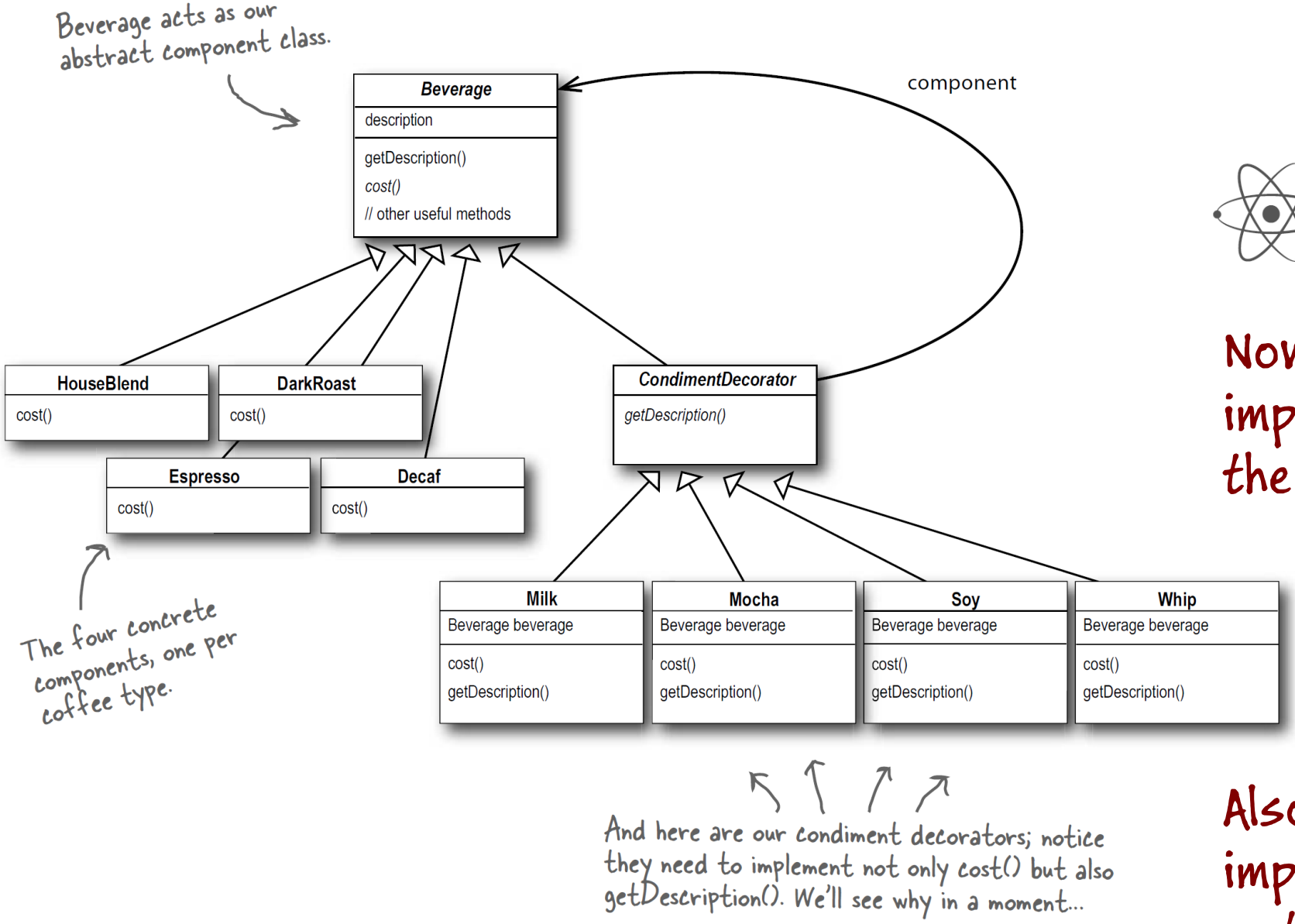
The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Consequences of Decorator Patterns

- Benefits
  - The Decorator pattern provides a more flexible way to add responsibilities to objects than inheritance. With decorators, responsibilities can be added at run-time simply by attaching them. In contrast, inheritance requires creating a new class for each additional responsibility.
  - Decorators also make it easy to add a property twice.
- Liability
  - A design that uses Decorator often results in systems composed of lots of little objects that all look alike. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.

Decorating our Beverages



BRAIN POWER

Now, think about how you'd implement the `cost()` method of the coffees and the condiments.

Also think about how you'd implement `getDescription()` method of the condiments.

Confusions

- Yes, CondimentDecorator is extending Beverage class and this is inheritance
- it is vital that decorators have the same type as the objects they decorate
- So, we use inheritance for type matching, not to get behavior
- We add new behavior by composing decorator with a component
  - Not by inheriting from a superclass, but by composing objects together
- We get a whole lot more flexibility about how to mix and match condiments and beverages.
- If we rely on inheritance, then behavior can be determined statically at compile time
  - With composition, we can mix and match decorators any way we like at runtime

Writing the Starbuzz code

Let's start with the Beverage class, actually it is the same with Starbuzz's original design.

```
public abstract class Beverage {
  String description = "Unknown Beverage";

  public String getDescription() {
    return description;
  }

  public abstract double cost();
}
```

Beverage is an abstract class with the two methods getDescription() and cost().

getDescription is already implemented for us, but we need to implement cost() in the subclasses.

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

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

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

Beverage is simple enough. Let's implement the abstract class for the Condiments (Decorators) as well.

Coding beverages

```
public class Espresso extends Beverage {
  public Espresso() {
    description = "Espresso";
  }

  public double cost() {
    return 1.99;
  }
}
```

First we extend the Beverage class, since this is a beverage.

To take care of the description, we set this in the constructor for the class. Remember the description instance variable is inherited from Beverage.

Finally, we need to compute the cost of an Espresso. We don't need to worry about adding in condiments in this class, we just need to return the price of an Espresso: $1.99.


```
public class HouseBlend extends Beverage {
  public HouseBlend() {
    description = "House Blend Coffee";
  }

  public double cost() {
    return .89;
  }
}
```

Okay, here's another Beverage. All we do is set the appropriate description, "House Blend Coffee," and then return the correct cost: 89¢.

You can create other two Beverage classes (DarkRoast and Decaf) in exactly the same way.

Coding condiments

Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

```
public class Mocha extends CondimentDecorator {
  Beverage beverage;

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

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

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

We're going to instantiate Mocha with a reference to a Beverage using:

(1) An instance variable to hold the beverage we are wrapping.

(2) A way to set this instance variable to the object we are wrapping. Here, we're going to to pass the beverage we're wrapping to the decorator's constructor.

We want our description to not only include the beverage – say "Dark Roast" – but also to include each item decorating the beverage, for instance, "Dark Roast, Mocha". So we first delegate to the object we are decorating to get its description, then append ", Mocha" to that description.

Now we need to compute the cost of our beverage with Mocha. First, we delegate the call to the object we're decorating, so that it can compute the cost; then, we add the cost of Mocha to the result.

Serving some coffees – test code to make orders

```
public class StarBuzzCoffee {
  public static void main(String args[]) {
    Beverage beverage = new Espresso();
    System.out.println(beverage.getDescription() + " $" +
                       beverage.cost());
    Beverage beverage2 = new DarkRoast();
    beverage2 = new Mocha(beverage2);
    beverage2 = new Mocha(beverage2);
    beverage2 = new Whip(beverage2);
    System.out.println(beverage2.getDescription() + " $" +
                       beverage2.cost());
    Beverage beverage3 = new HouseBlend();
    beverage3 = new Soy(beverage3);
    beverage3 = new Mocha(beverage3);
    beverage3 = new Whip(beverage3);
    System.out.println(beverage3.getDescription()
          + " $" + beverage2.cost());
  }
}
```

Order up an espresso, no condiments and print its description and cost.

Make a DarkRoast object.

Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

Finally, give us a HouseBlend with Soy, Mocha, and Whip.

```
File  Edit  Window  Help  CloudsInMyCoffee
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```

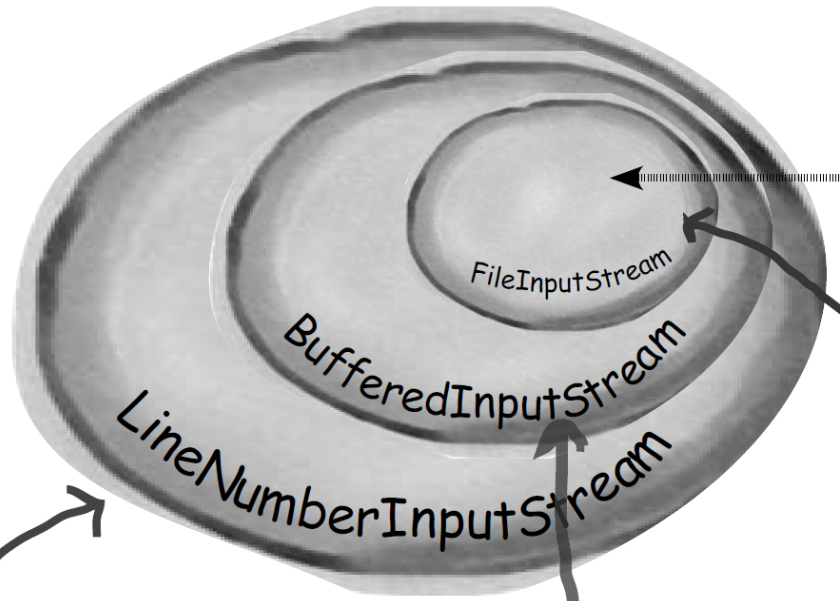
Real World Decorators: Java I/O

The number of classes in `java.io` package is *overwhelming*.

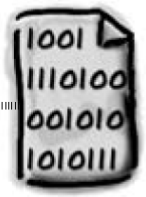
You're not alone if you said "whoa" when you looked at this API.

But now that you know the Decorator Pattern, the I/O classes should make more sense since `java.io` package is largely based on Decorator.

Here's an example to read data from file:



A text file for reading.



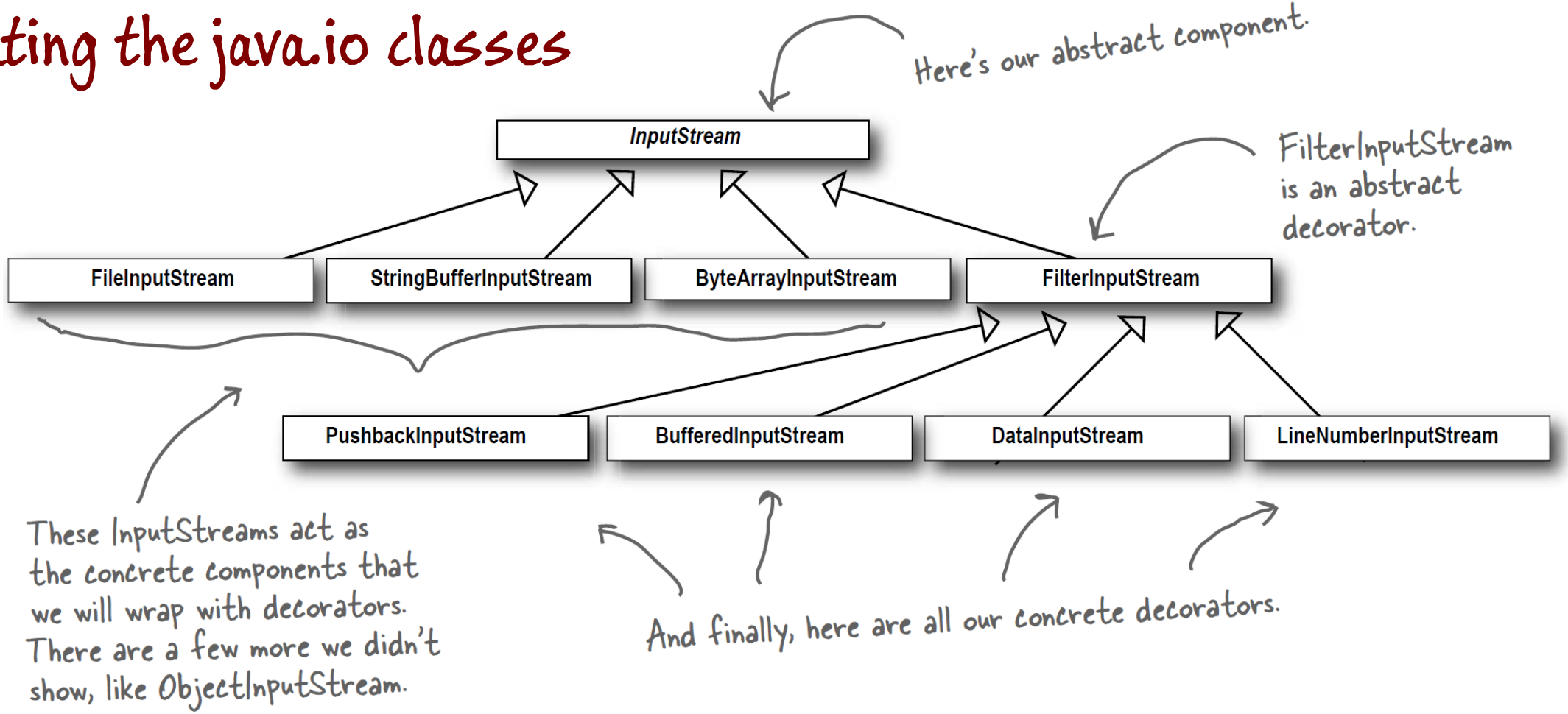
FileInputStream is the component that's being decorated The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

BufferedInputStream is a concrete decorator. BufferedInputStream adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method readLine() for reading character-based input, a line at a time.

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

BufferedInputStream and ***LineNumber***InputStream both extend ***Filter***InputStream, which acts as the abstract decorator class.

Decorating the java.io classes



- Not so different from Starbuzz design, huh?
- Now you can more easily understand `java.io` API docs.
- Downside of Decorator Pattern is that they often result in a large number of classes.
  - But now that you know how Decorator Works, you can keep things in perspective

Writing your own Java I/O Decorator

Write a decorator that converts all uppercase characters to lowercase.

No problem. I just have to extend the FilterInputStream class and override the read() methods.

```
public class LowerCaseInputStream extends FilterInputStream {
  public LowerCaseInputStream(InputStream in) {
    super(in);
  }
  public int read() throws IOException {
    int c = super.read();
    return (c == -1 ? c : Character.toLowerCase((char)c));
  }
  public int read(byte[] b, int offset, int len) throws IOException {
    int result = super.read(b, offset, len);
    for (int i = offset; i < offset+result; i++) {
      b[i] = (byte)Character.toLowerCase((char)b[i]);
    }
    return result;
  }
}
```

First, extend the FilterInputStream, the abstract decorator for all InputStreams.

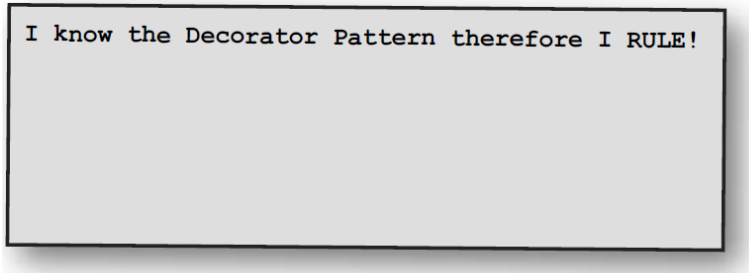
Now we need to implement two read methods. They take a byte (or an array of bytes) and convert each byte (that represents a character) to lowercase if it's an uppercase character.

Don't forget to import java.io... (not shown)

Test out your new Java I/O Decorator

```
public class InputTest {
  public static void main(String[] args) throws IOException {
    int c;
    try {
      InputStream in =
        new LowerCaseInputStream(
          new BufferedInputStream(
            new FileInputStream("test.txt")));
      while((c = in.read()) >= 0) {
        System.out.print((char)c);
      }
      in.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Set up the FileInputStream and decorate it, first with a BufferedInputStream and then our brand new LowerCaseInputStream filter.



test.txt file

Just use the stream to read characters until the end of file and print as we go.

You need to make this file.

```
File Edit Window Help DecoratorsRule
% java InputTest
i know the decorator pattern therefore i rule!
%
```

Tools for your Design Toolbox

Let's look at the tools you've put in your OO toolbox.

Bullet Points

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.
- Composition and delegation can often be used to add new behaviors at runtime.
- The Decorator Pattern provides an alternative to subclassing for extending behavior.
- Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)
- Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.
- You can wrap a component with any number of decorators.
- Decorators can result in many small objects in our design, and overuse can be complex.