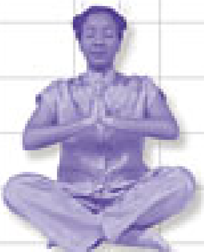
Your Brain on Design Patterns

Head First Design Patterns

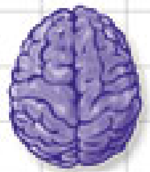
Avoid those embarrassing coupling mistakes

Learn why everything your friends know about Factory Pattern is probably wrong



Discover the secrets of the Patterns Guru

Load the patterns that matter straight into your brain



Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern

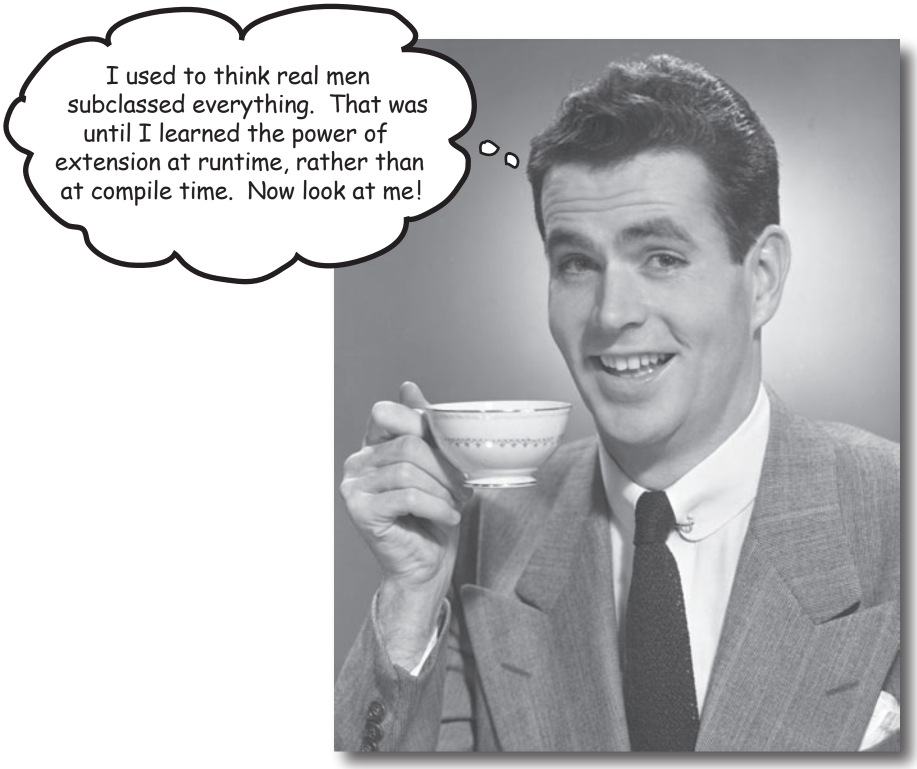
See why Jim's love life improved when he cut down his inheritance

O'REILLY®

Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

3 the Decorator Pattern

Decorating Objects



Just call this chapter "Design Eye for the Inheritance Guy."

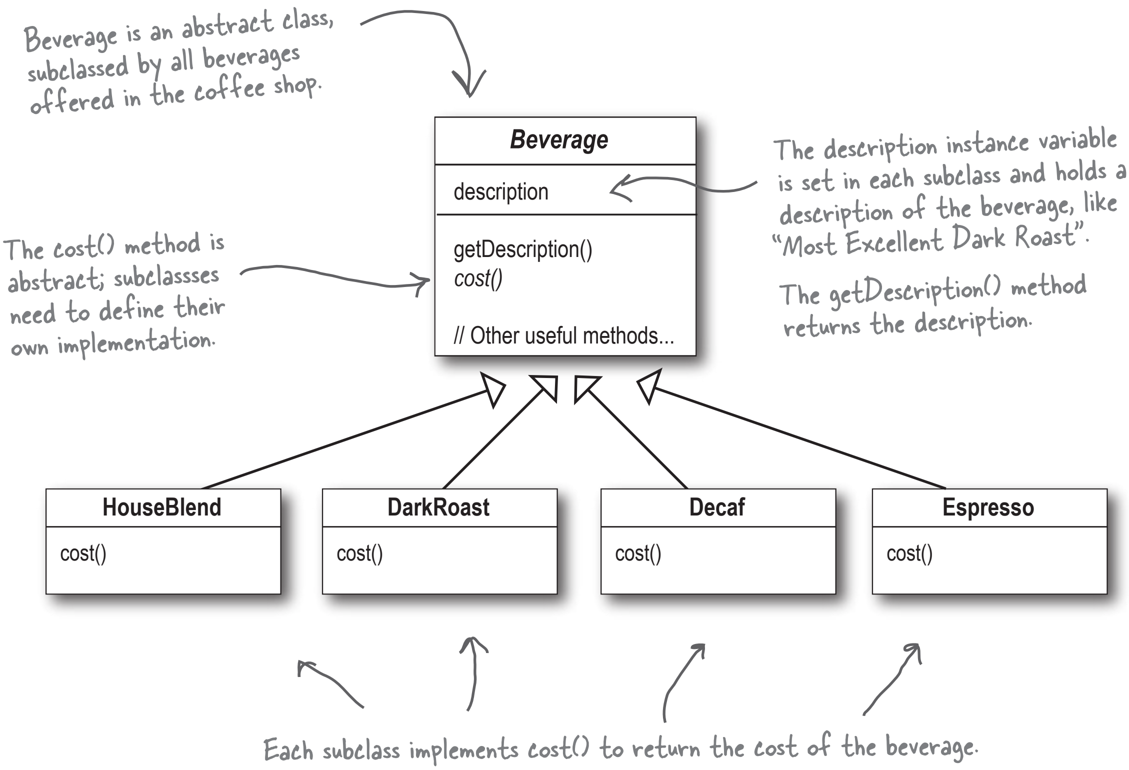
We'll re-examine the typical overuse of inheritance and you'll learn how to decorate your classes at runtime using a form of object composition. Why? Once you know the techniques of decorating, you'll be able to give your (or someone else's) objects new responsibilities *without making any code changes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

Welcome to Starbuzz Coffee

Starbuzz Coffee has made a name for itself as the fastest growing coffee shop around. If you've seen one on your local corner, look across the street; you'll see another one.

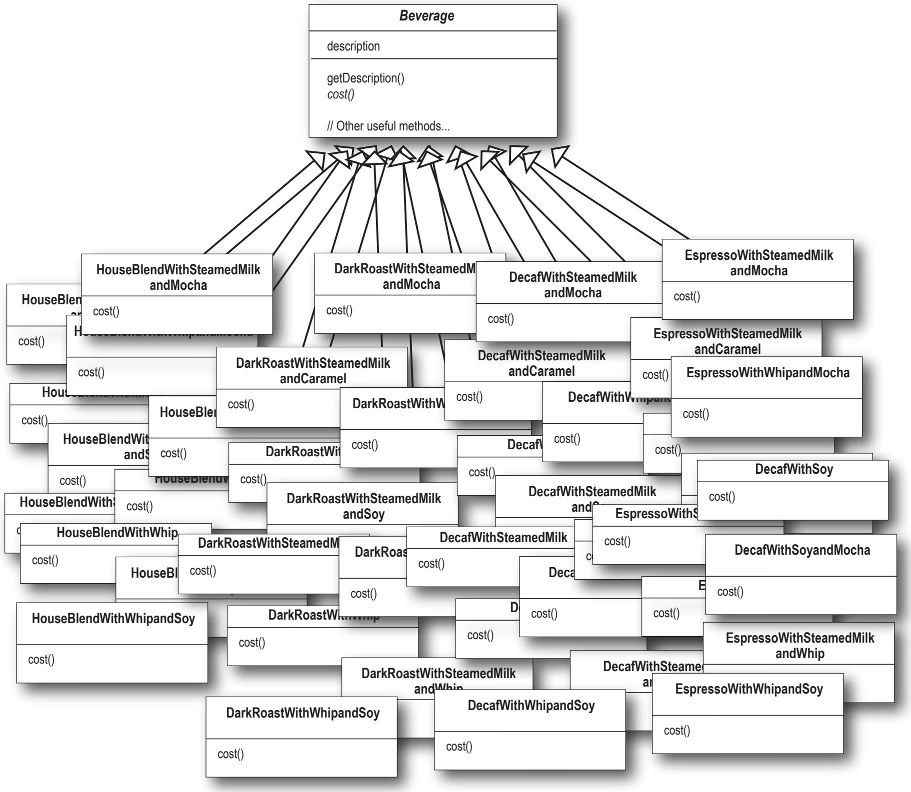
Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.

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



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.

Here's their first attempt...



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

BRAIN POWER

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...

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.

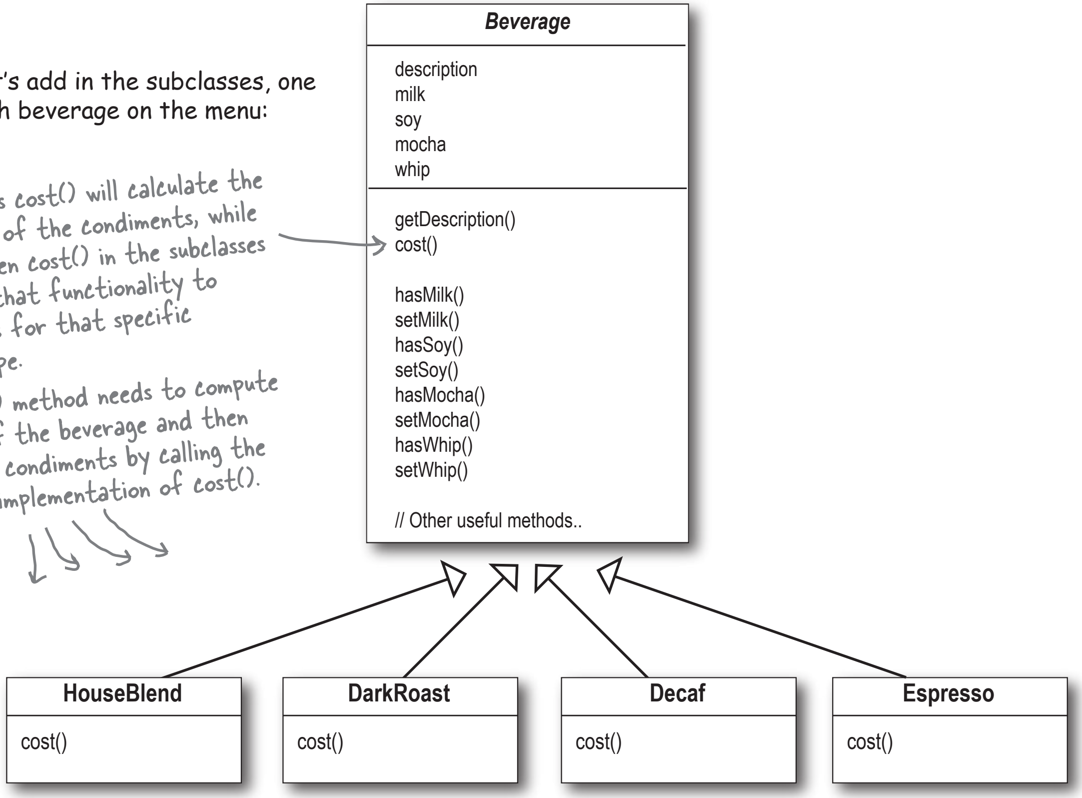
Now we'll implement cost() in Beverage (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override cost(), but they will also invoke the super version so that they can calculate the total cost of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



Sharpen your pencil

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

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




   }
}
```

```
public class DarkRoast extends Beverage {

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

   public double cost() {




   }
}
```

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

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.

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 (iced tea?), the condiments may not be appropriate, yet the Tea subclass will still inherit methods like hasWhip().

As we saw in Chapter 1, this is a very bad idea!

What if a customer wants a double mocha?

Your turn:

Master and Student...

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

Grasshopper 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.

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.

Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code. What do we get if we accomplish this? Designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements.

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 (in Chapter 2)... 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. In this chapter you'll see a good example of using the Decorator pattern to follow the Open-Closed principle.

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 wastefu). 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.

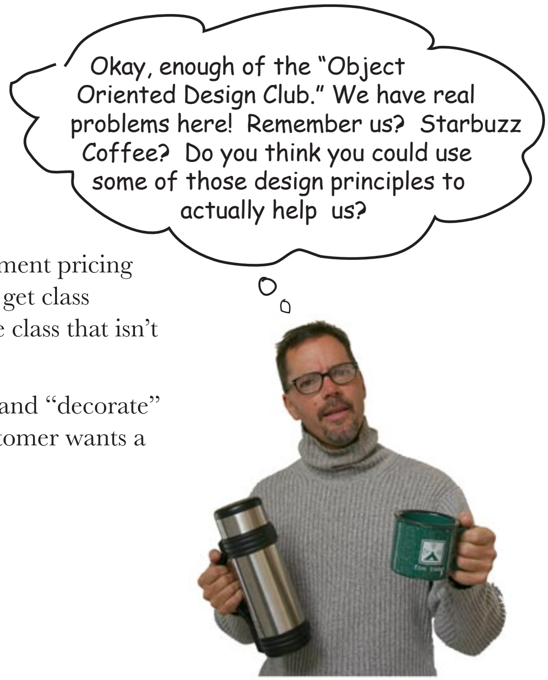
Q: How do I know which areas of change are more important?

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.

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

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, we've seen that representing our beverage plus condiment pricing scheme with inheritance has not worked out very well – we get 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: we'll start with a beverage and "decorate" it with the condiments at runtime. For example, if the customer wants a Dark Roast with Mocha and Whip, then we'll:

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**

Okay, but how do you "decorate" an object, and how does delegation come into this? A hint: think of decorator objects as "wrappers." Let's see how this works...

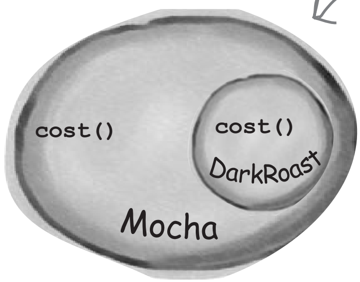
Constructing a drink order with Decorators

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.

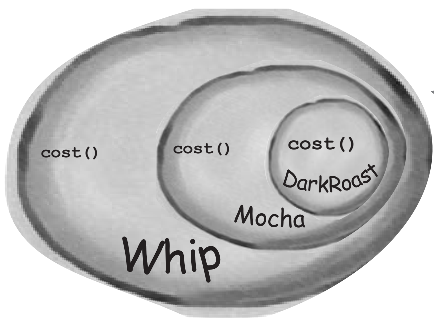
2 The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.

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).

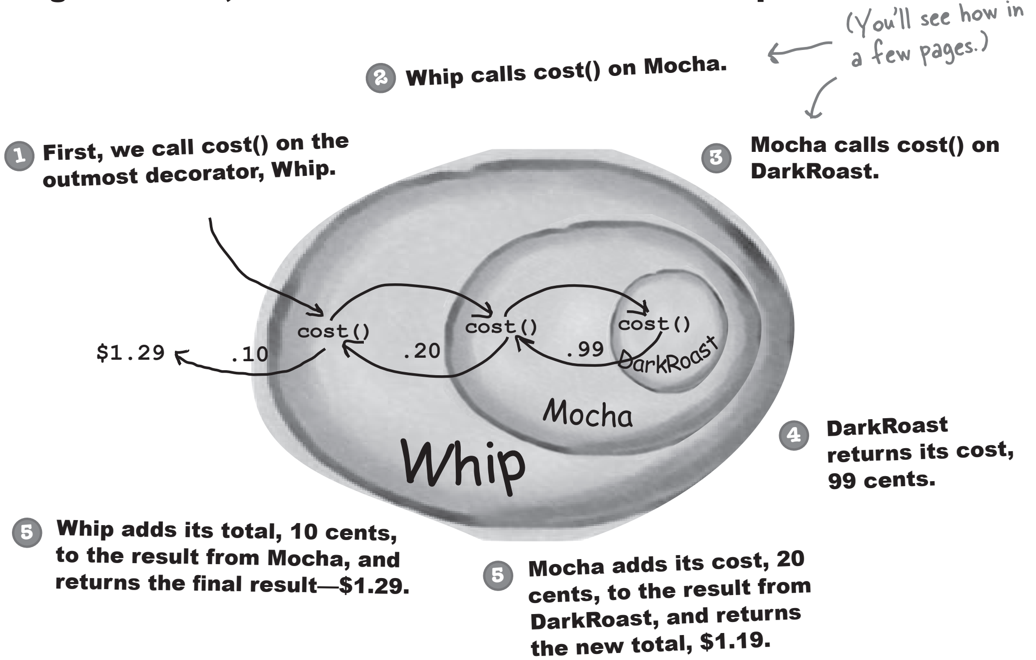
3 The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.



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. 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.

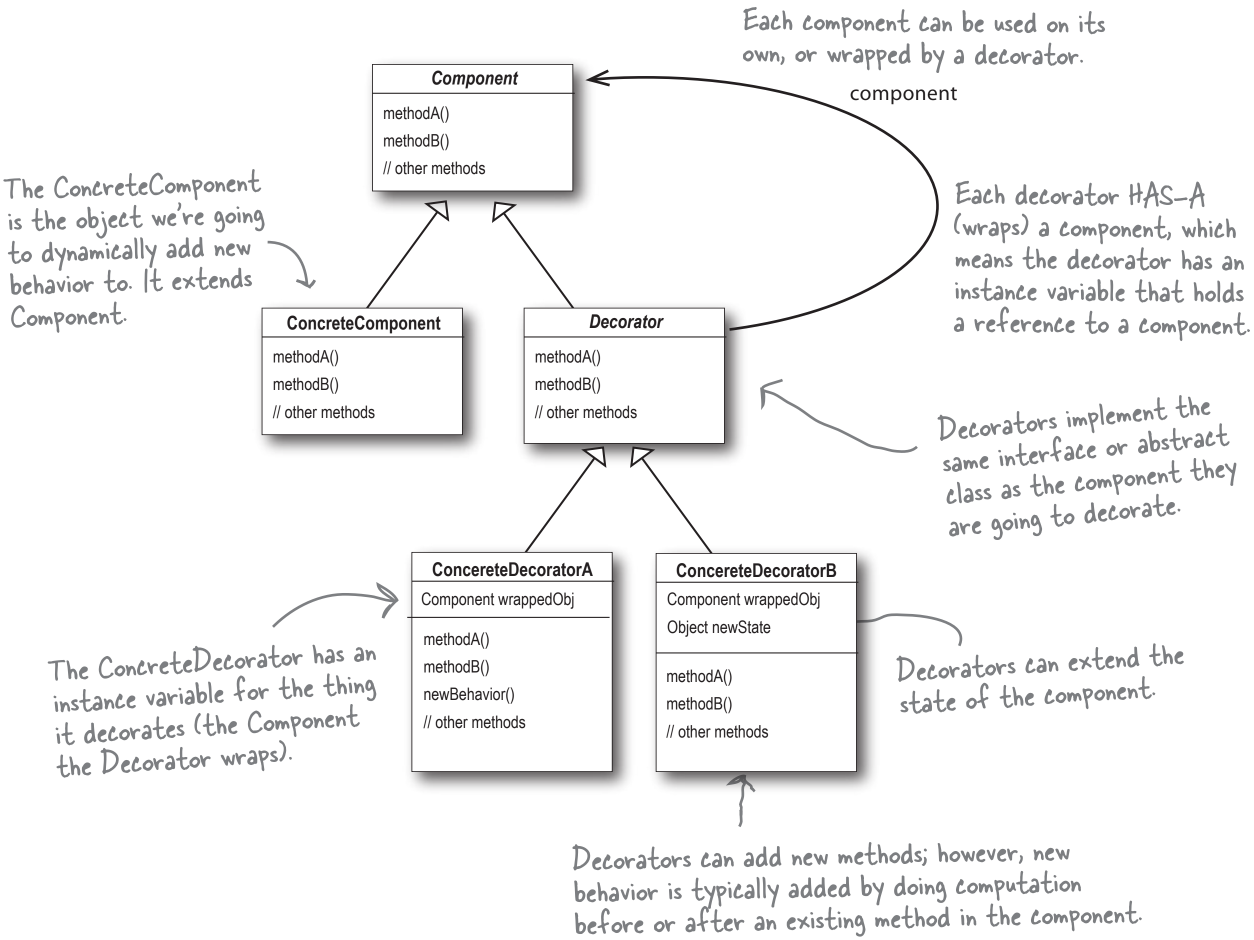
Now let's see how this all really works by looking at the Decorator Pattern definition and writing some code.

The Decorator Pattern defined

Let's first take a look at the Decorator Pattern description:

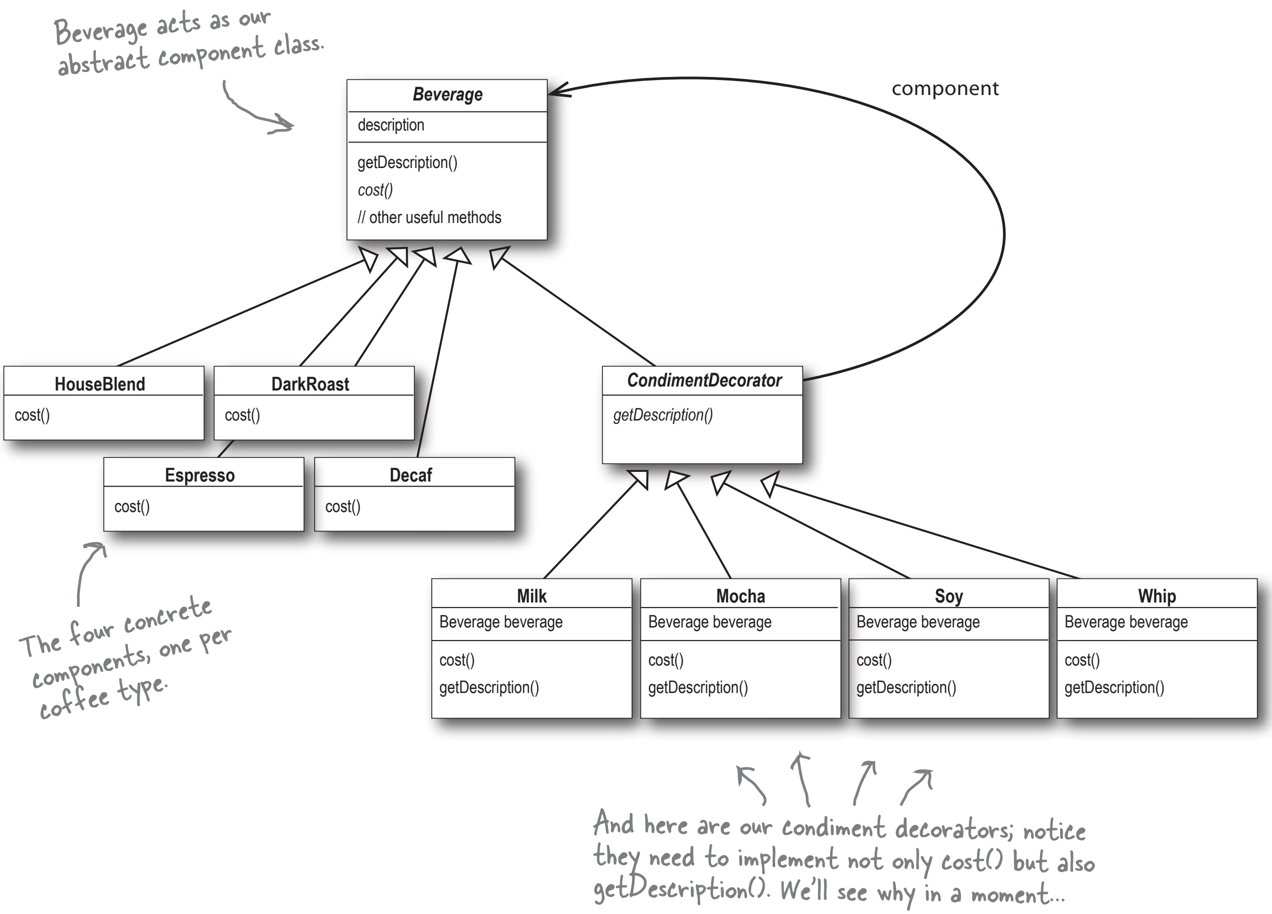
> **The Decorator Pattern** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

While that describes the *role* of the Decorator Pattern, it doesn't give us a lot of insight into how we'd *apply* the pattern to our own implementation. Let's take a look at the class diagram, which is a little more revealing (on the next page we'll look at the same structure applied to the beverage problem).



Decorating our Beverages

Okay, let's work our Starbuzz beverages into this framework...

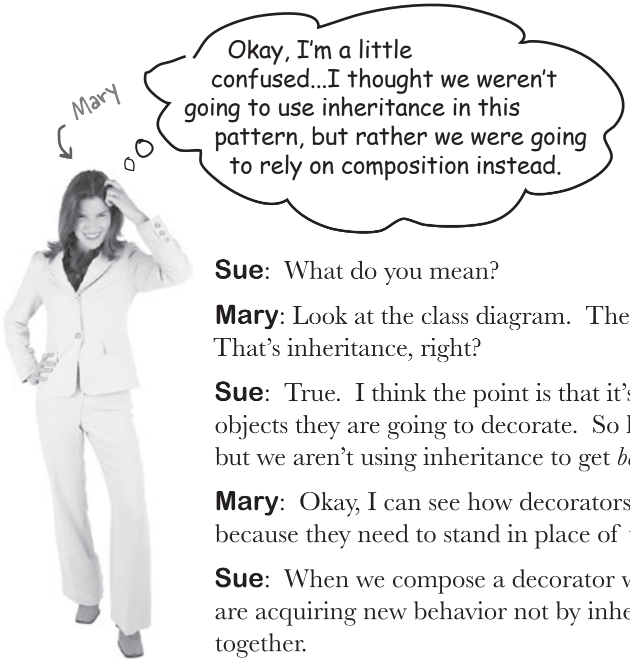


BRAIN POWER

Before going further, think about how you'd implement the cost() method of the coffees and the condiments. Also think about how you'd implement the getDescription() method of the condiments.

Cubicle Conversation

Some confusion over Inheritance versus Composition



Sue: What do you mean?

Mary: Look at the class diagram. The CondimentDecorator is extending the Beverage class. That's inheritance, right?

Sue: True. I think the point is that it's vital that the decorators have the same type as the objects they are going to decorate. So here we're using inheritance to achieve the *type matching*, but we aren't using inheritance to get *behavior*.

Mary: Okay, I can see how decorators need the same "interface" as the components they wrap because they need to stand in place of the component. But where does the behavior come in?

Sue: When we compose a decorator with a component, we are adding new behavior. We are acquiring new behavior not by inheriting it from a superclass, but by composing objects together.

Mary: Okay, so we're subclassing the abstract class Beverage in order to have the correct type, not to inherit its behavior. The behavior comes in through the composition of decorators with the base components as well as other decorators.

Sue: That's right.

Mary: Ooooh, I see. And because we are using object composition, we get a whole lot more flexibility about how to mix and match condiments and beverages. Very smooth.

Sue: Yes, if we rely on inheritance, then our behavior can only be determined statically at compile time. In other words, we get only whatever behavior the superclass gives us or that we override. With composition, we can mix and match decorators any way we like... *at runtime.*

Mary: And as I understand it, we can implement new decorators at any time to add new behavior. If we relied on inheritance, we'd have to go in and change existing code any time we wanted new behavior.

Sue: Exactly.

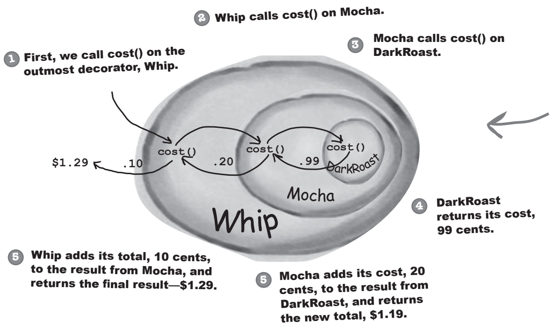
Mary: I just have one more question. If all we need to inherit is the type of the component, how come we didn't use an interface instead of an abstract class for the Beverage class?

Sue: Well, remember, when we got this code, Starbuzz already *had* an abstract Beverage class. Traditionally the Decorator Pattern does specify an abstract component, but in Java, obviously, we could use an interface. But we always try to avoid altering existing code, so don't "fix" it if the abstract class will work just fine.

New barista training

Okay, I need for you to make me a *double* mocha, soy latte with whip.

Make a picture for what happens when the order is for a "double mocha soy lotte with whip" beverage. Use the menu to get the correct prices, and draw your picture using the same format we used earlier (from a few pages back):



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

Sharpen your pencil

Draw your picture here.

Starbuzz Coffee

Coffees	
House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99
Condiments	
Steamed Milk	.10
Mocha	.20
Soy	.15
Whip	.10

Writing the Starbuzz code

It's time to whip this design into some real code.

Let's start with the Beverage class, which doesn't need to change from Starbuzz's original design. Let's take a look:

```
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.

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

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...

Coding beverages

Now that we've got our base classes out of the way, let's implement some beverages. We'll start with Espresso. Remember, we need to set a description for the specific beverage and also implement the cost() method.

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

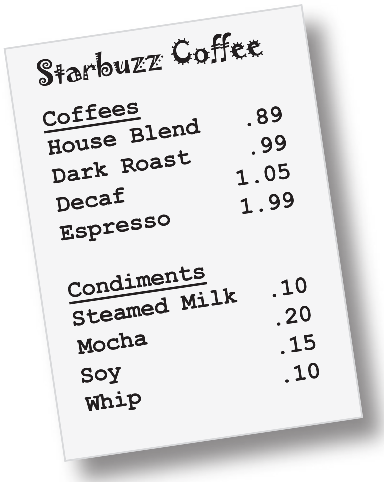
```
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

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

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 the other two Beverage classses (DarkRoast and Decaf) in exactly the same way.

Coding condiments

If you look back at the Decorator Pattern class diagram, you'll see we've now written our abstract component (Beverage), we have our concrete components (HouseBlend), and we have our abstract decorator (CondimentDecorator). Now it's time to implement the concrete decorators. Here's Mocha:

Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

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 pass the beverage we're wrapping to the decorator's constructor.

```
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 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.

On the next page we'll actually instantiate the beverage and wrap it with all its condiments (decorators), but first...

Sharpen your pencil

Write and compile the code for the other Soy and Whip condiments. You'll need them to finish and test the application.

Serving some coffees

Congratulations. It's time to sit back, order a few coffees and marvel at the flexible design you created with the Decorator Pattern.

Here's some 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()
                + " $" + beverage3.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.

*We're going to see a much better way of creating decorated objects when we cover the Factory and Builder Design Patterns.

Now, let's get those orders in:

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
%
```

there are no Dumb Questions

Q: I'm a little worried about code that might test for a specfic concrete component – say, HouseBlend – and do something, like issue a discount. Once I've wrapped the HouseBlend with decorators, this isn't going to work anymore.

A: That is exactly right. If you have code that relies on the concrete component's type, decorators will break that code. As long as you only write code against the abstract component type, the use of decorators will remain transparent to your code. However, once you start writing code against concrete components, you'll want to rethink your application design and your use of decorators.

Q: Wouldn't it be easy for some client of a beverage to end up with a decorator that isn't the outermost decorator? Like if I had a DarkRoast with Mocha, Soy, and Whip, it would be easy to write code that somehow ended up with a reference to Soy instead of Whip, which means it would not including Whip in the order.

A: You could certainly argue that you have to manage more objects with the Decorator Pattern and so there is an increased chance that coding errors will introduce the kinds of problems you suggest. However, decorators are typically created by using other patterns like Factory and Builder. Once we've covered these patterns, you'll see that the creation of the concrete component with its decorator is "well encapsulated" and doesn't lead to these kinds of problems.

Q: Can decorators know about the other decorations in the chain? Say, I wanted my getDecription() method to print "Whip, Double Mocha" instead of "Mocha, Whip, Mocha"? That would require that my outermost decorator know all the decorators it is wrapping.

A: Decorators are meant to add behavior to the object they wrap. When you need to peek at multiple layers into the decorator chain, you are starting to push the decorator beyond its true intent. Nevertheless, such things are possible. Imagine a CondimentPrettyPrint decorator that parses the final decription and can print "Mocha, Whip, Mocha" as "Whip, Double Mocha." Note that getDecription() could return an ArrayList of descriptions to make this easier.

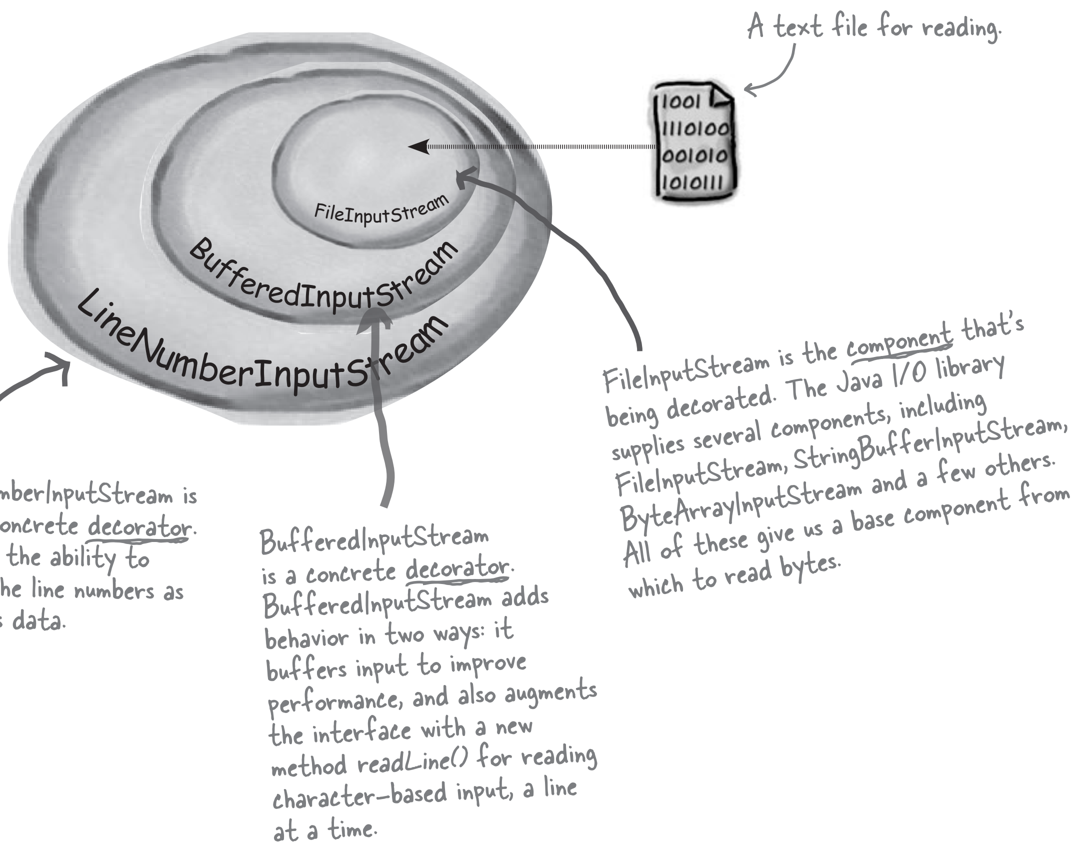
Sharpen your pencil

Our friends at Starbuzz have introduced sizes to their menu. You can now order a coffee in tall, grande, and venti sizes (translation: small, medium, and large). Starbuzz saw this as an intrinsic part of the coffee class, so they've added two methods to the Beverage class: setSize() and getSize(). They'd also like for the condiments to be charged according to size, so for instance, Soy costs 10¢, 15¢ and 20¢ respectively for tall, grande, and venti coffees.

How would you alter the decorator classes to handle this change in requirements?

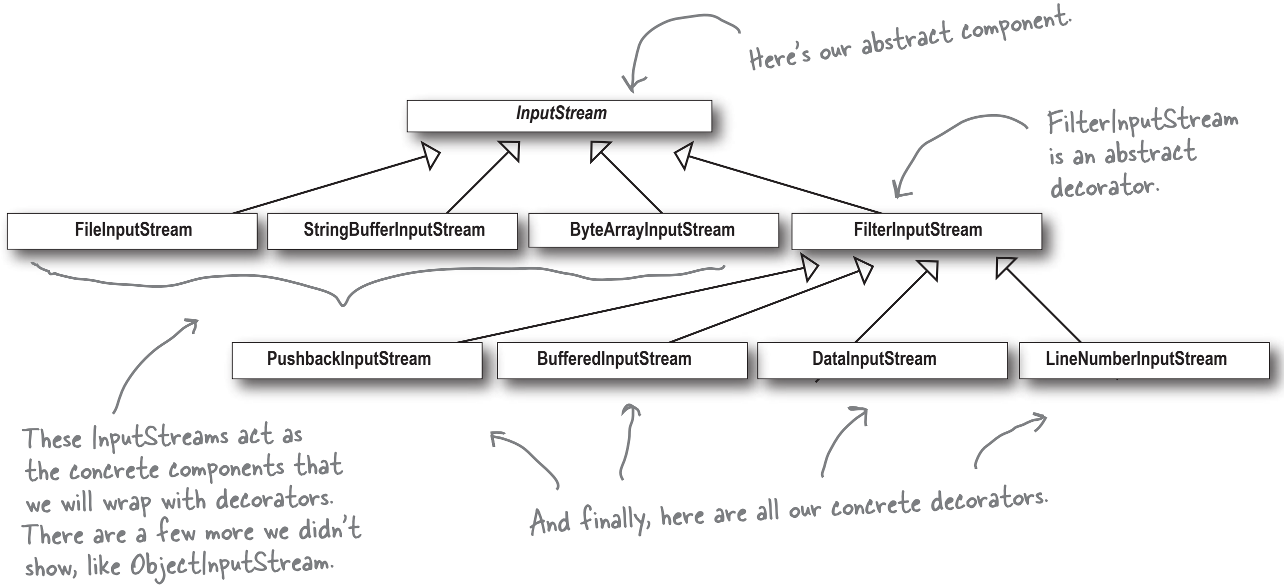
Real World Decorators: Java I/O

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:



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

Decorating the java.io classes



You can see that this isn't so different from the Starbuzz design. You should now be in a good position to look over the java.io API docs and compose decorators on the various *input* streams.

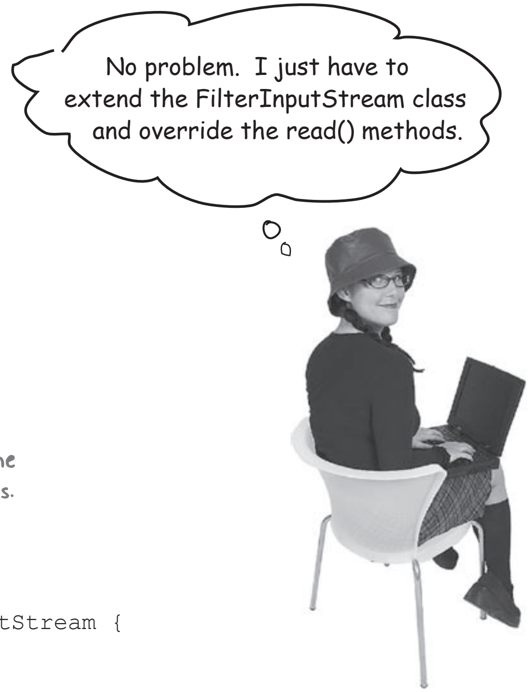
You'll see that the *output* streams have the same design. And you've probably already found that the Reader/Writer streams (for character-based data) closely mirror the design of the streams classes (with a few differences and inconsistencies, but close enough to figure out what's going on).

Java I/O also points out one of the *downsides* of the Decorator Pattern: designs using this pattern often result in a large number of small classes that can be overwhelming to a developer trying to use the Decorator-based API. But now that you know how Decorator works, you can keep things in perspective and when you're using someone else's Decorator-heavy API, you can work through how their classes are organized so that you can easily use wrapping to get the behavior you're after.

Writing your own Java I/O Decorator

Okay, you know the Decorator Pattern, you've seen the I/O class diagram. You should be ready to write your own input decorator.

How about this: write a decorator that converts all uppercase characters to lowercase in the input stream. In other words, if we read in "I know the Decorator Pattern therefore I RULE!" then your decorator converts this to "i know the decorator pattern therefore i rule!"



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

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

```
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;
    }
}
```

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.

REMEMBER: we don't provide import and package statements in the code listings. Get the complete source code from the wickedlysmart web site. You'll find the URL on page xxxiii in the Intro.

Test out your new Java I/O Decorator

Write some quick code to test the 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.

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

```
I know the Decorator Pattern therefore I RULE!
```

test.txt file

You need to make this file.

Give it a spin:

File Edit Window Help DecoratorsRule

```
% java InputTest
i know the decorator pattern therefore i rule!
%
```

Patterns Exposed

This week's interview:
Confessions of a Decorator

HeadFirst: Welcome Decorator Pattern. We've heard that you've been a bit down on yourself lately?

Decorator: Yes, I know the world sees me as the glamorous design pattern, but you know, I've got my share of problems just like everyone.

HeadFirst: Can you perhaps share some of your troubles with us?

Decorator: Sure. Well, you know I've got the power to add flexibility to designs, that much is for sure, but I also have a *dark side*. You see, I can sometimes add a lot of small classes to a design and this occasionally results in a design that's less than straightforward for others to understand.

HeadFirst: Can you give us an example?

Decorator: Take the Java I/O libraries. These are notoriously difficult for people to understand at first. But if they just saw the classes as a set of wrappers around an InputStream, life would be much easier.

HeadFirst: That doesn't sound so bad. You're still a great pattern, and improving this is just a matter of public education, right?

Decorator: There's more, I'm afraid. I've got typing problems: you see, people sometimes take a piece of client code that relies on specific types and introduce decorators without thinking through everything. Now, one great thing about me is that ***you can usually insert decorators transparently and the client never has to know it's dealing with a decorator.*** But like I said, some code is dependent on specific types and when you start introducing decorators, boom! Bad things happen.

HeadFirst: Well, I think everyone understands that you have to be careful when inserting decorators, I don't think this is a reason to be too down on yourself.

Decorator: I know, I try not to be. I also have the problem that introducing decorators can increase the complexity of the code needed to instantiate the component. Once you've got decorators, you've got to not only instantiate the component, but also wrap it with who knows how many decorators.

HeadFirst: I'll be interviewing the Factory and Builder patterns next week – I hear they can be very helpful with this?

Decorator: That's true; I should talk to those guys more often.

HeadFirst: Well, we all think you're a great pattern for creating flexible designs and staying true to the Open-Closed Principle, so keep your chin up and think positively!

Decorator: I'll do my best, thank you.

Tools for your Design Toolbox

You've got another chapter under your belt and a new principle and pattern in the toolbox.

OO Basics

...ction

...ulation

...rphism

...ance

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

Stra...
encap...
inter...
vary...

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?

BULLET POINTS

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.
- In our designs we should allow behavior to be extended without the need to modify existing code.
- Composition and delegation can often be used to add new behaviors at runtime.
- The Decorator Pattern provides an alternative to subclassing for extending behavior.
- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
- 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 are typically transparent to the client of the component; that is, unless the client is relying on the component's concrete type.
- Decorators can result in many small objects in our design, and overuse can be complex.

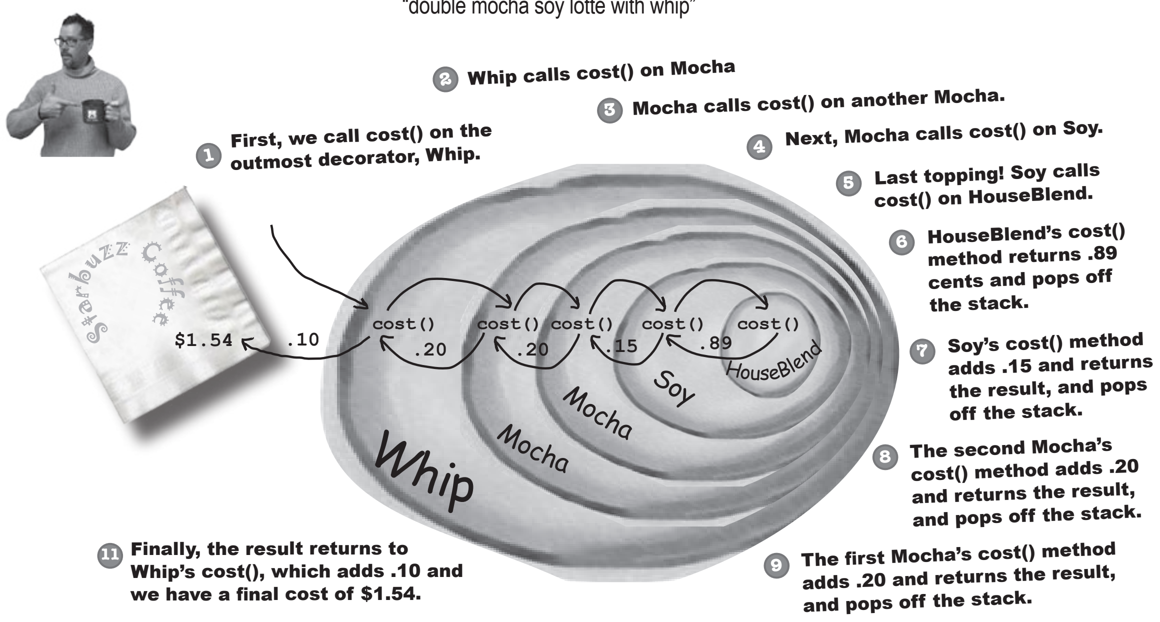
Exercise solutions

```
public class Beverage {

   // declare instance variables for milkCost,
  //  soyCost, mochaCost, and whipCost, and
   // getters and setters for milk, soy, mocha
   // and whip.

public float cost() {

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

```
public class DarkRoast extends Beverage {

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

   public float cost() {

      return 1.99 + super.cost();

   }
}
```

New barista training

"double mocha soy lotte with whip"



Exercise solutions

Our friends at Starbuzz have introduced sizes to their menu. You can now order a coffee in tall, grande, and venti sizes (for us normal folk: small, medium, and large). Starbuzz saw this as an intrinsic part of the coffee class, so they've added two methods to the Beverage class: setSize() and getSize(). They'd also like for the condiments to be charged according to size, so for instance, Soy costs 10¢, 15¢, and 20¢ respectively for tall, grande, and venti coffees.

How would you alter the decorator classes to handle this change in requirements?

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

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

    public getSize() {
        return beverage.getSize();
    }

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

    public double cost() {
        double cost = beverage.cost();
        if (getSize() == Beverage.TALL) {
           cost += .10;
        } else if (getSize() == Beverage.GRANDE) {
           cost += .15;
        }  else if (getSize() == Beverage.VENTI) {
           cost += .20;
        }
        return cost;
    }
}
```

Now we need to propagate the getSize() method to the wrapped beverage. We should also move this method to the abstract class since it's used in all condiment decorators.

Here we get the size (which propagates all the way to the concrete beverage) and then add the appropriate cost.