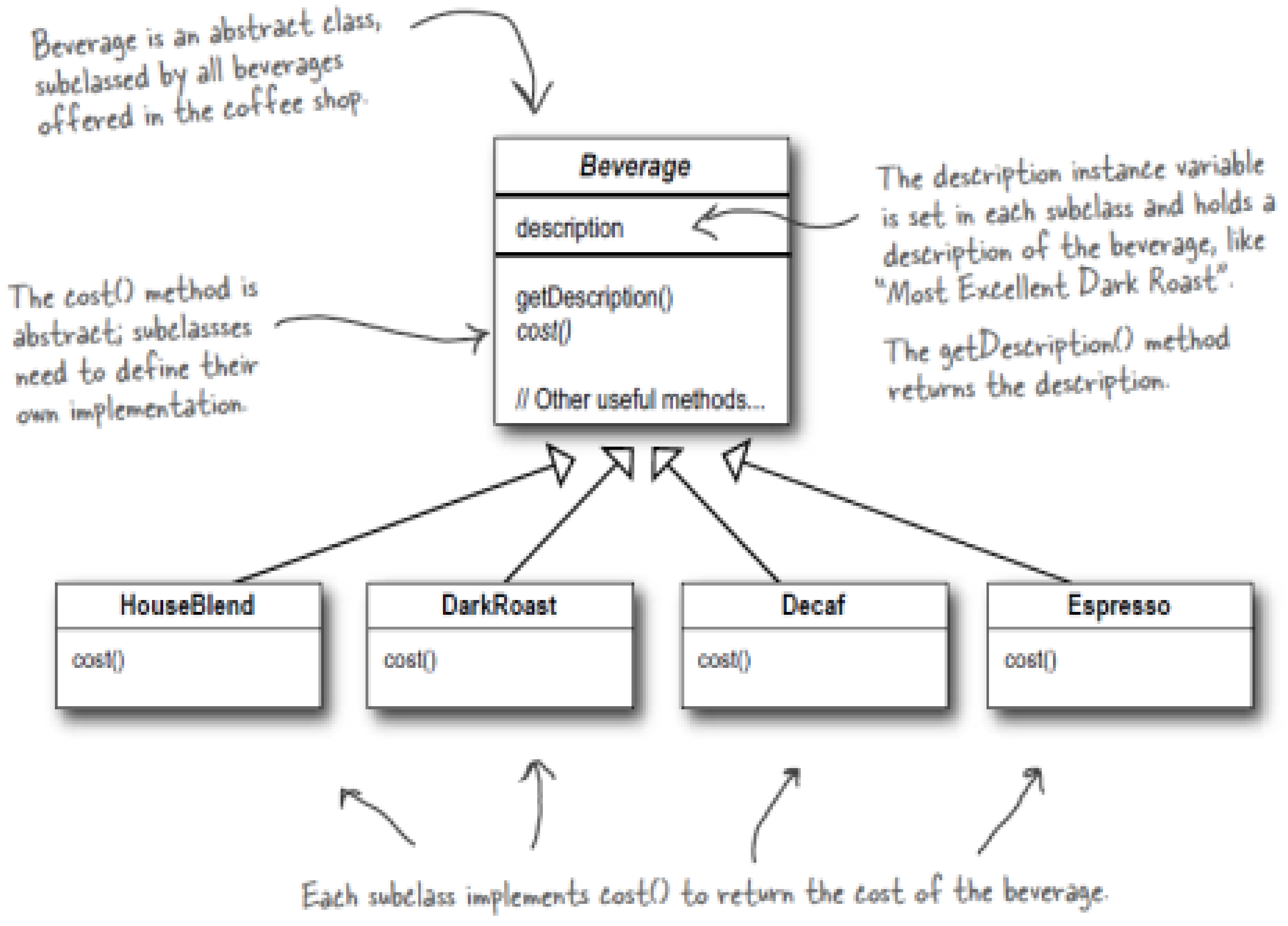
Decorator Pattern

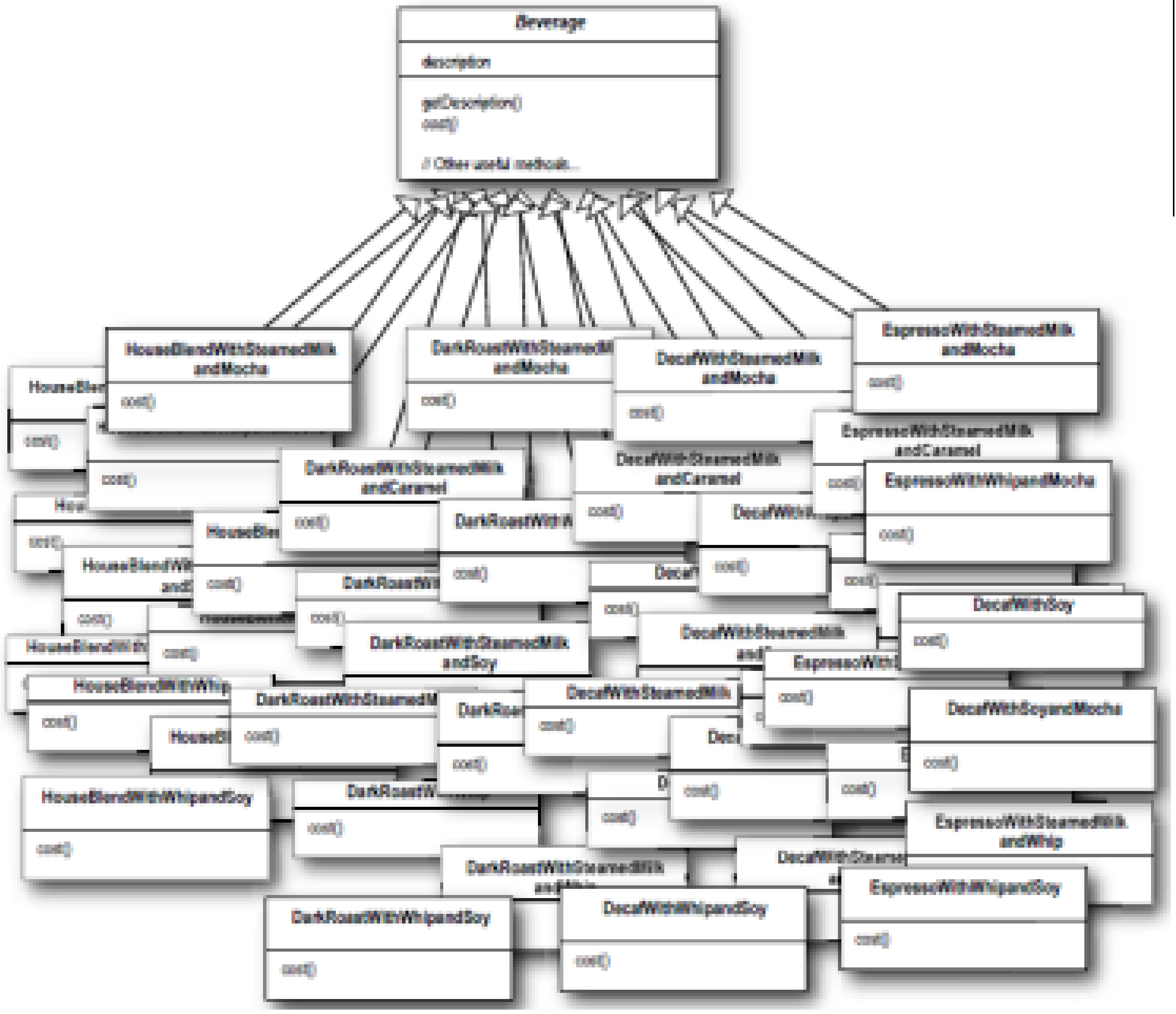
Welcome to Starbuzz Coffee

- 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...


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.

Coffee

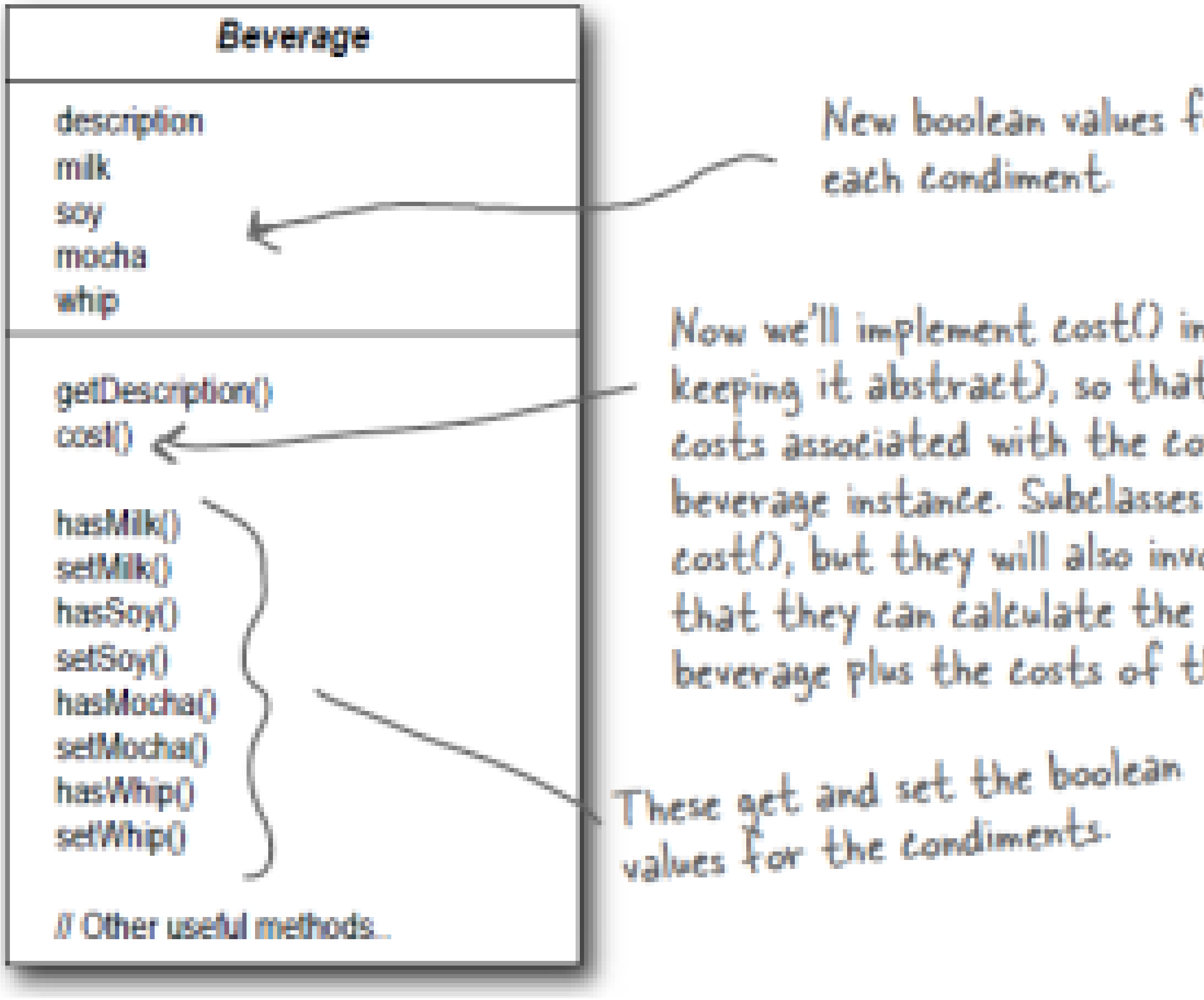
- In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they really need to get them built into their order system.
- Here’s their first attempt...


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

Problems

- “class explosion?”
- 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?

Variables

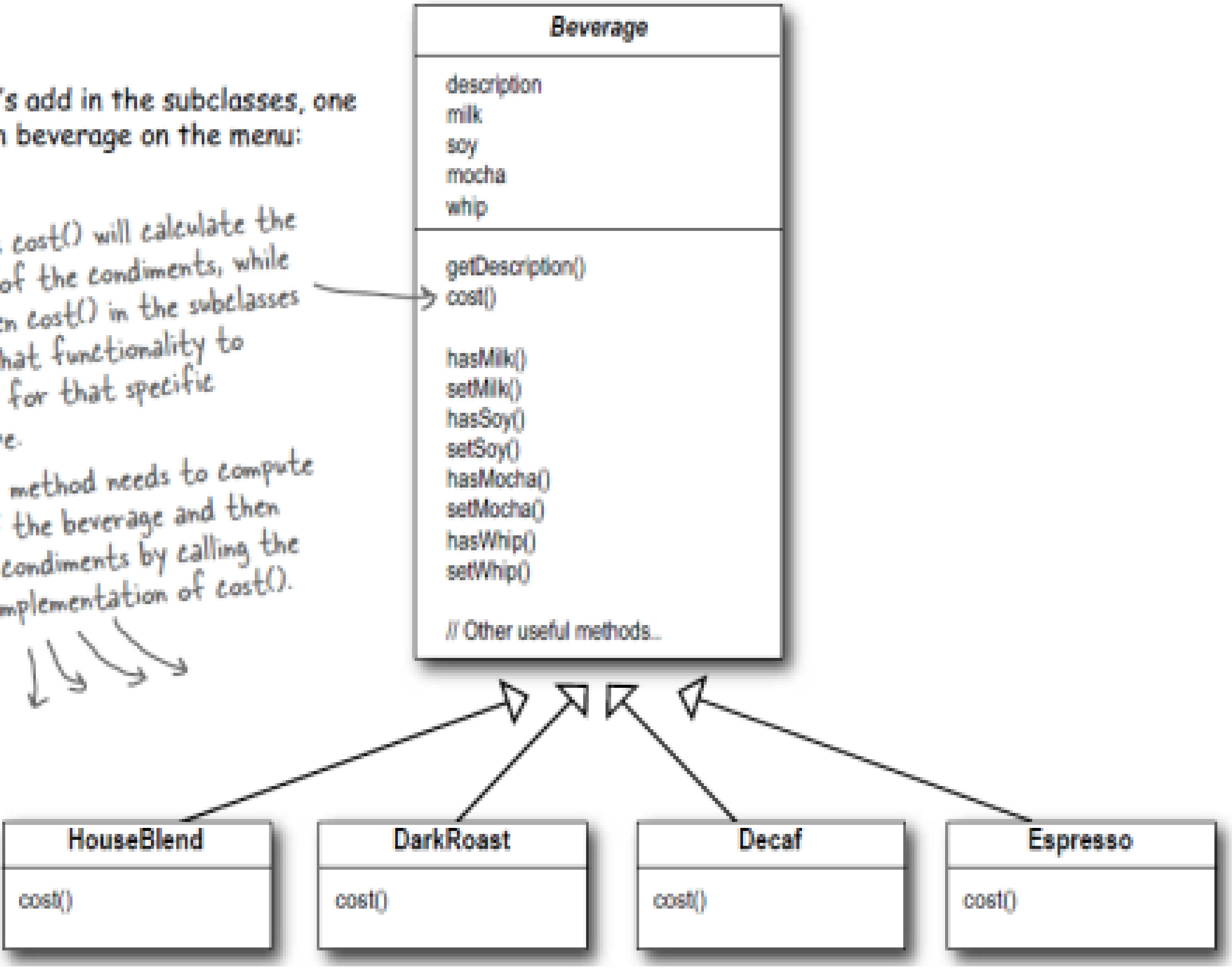


Now we'll implement cost() in Beverage (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override cost(), but they will also invoke the super version so that they can calculate the total cost of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

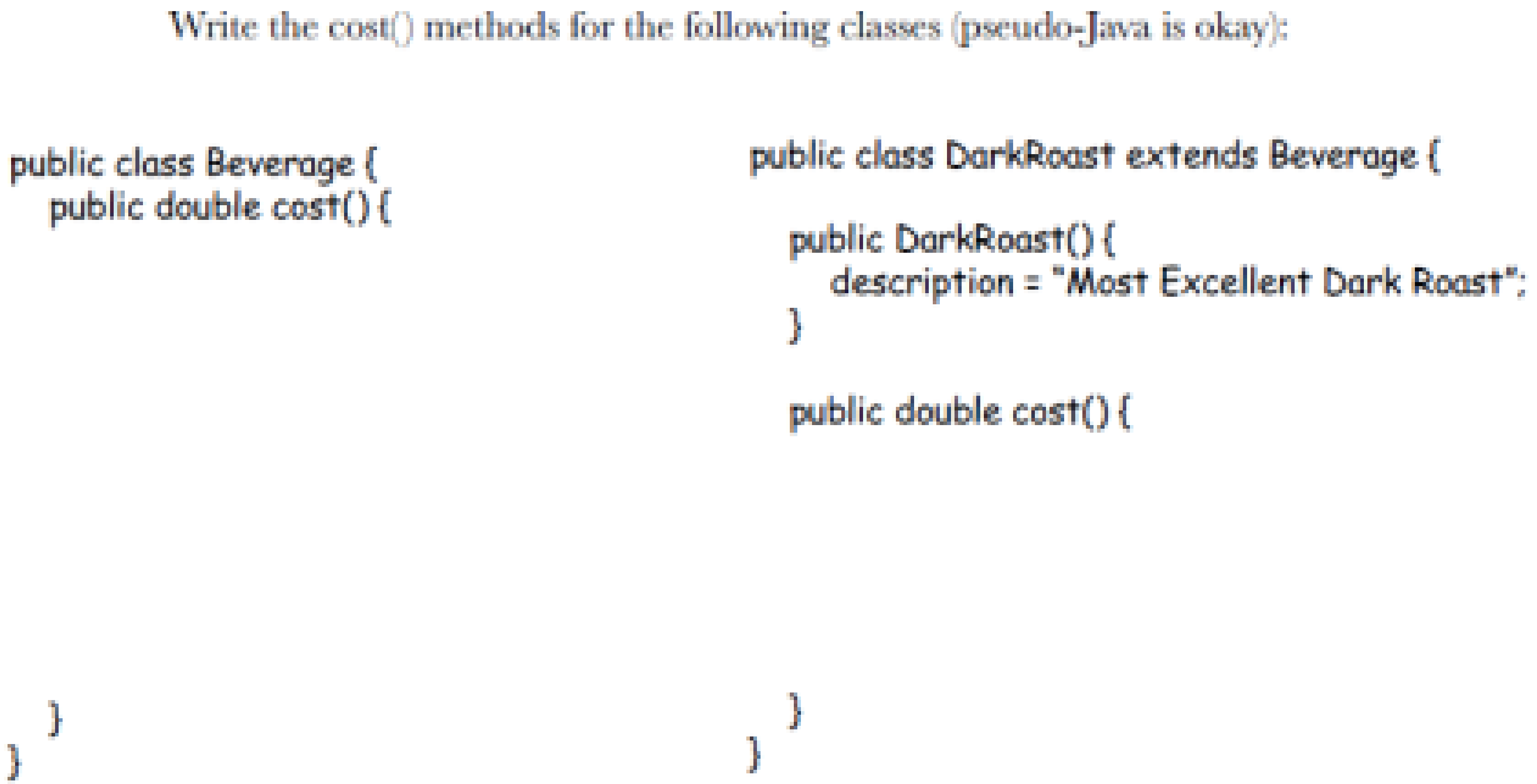
Now let's add in the subclasses, one for each beverage on the menu:

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().



Try



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() {






    }
}
```

```
public class Beverage {

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

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;
    }
}
```

```
public class DarkRoast extends Beverage {

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

   public double cost() {

      return 1.99 + super.cost();

   }
}
```

Problems with this design

- 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().
  - What if a customer wants a double mocha?

The Open-Closed Principle

- *Classes should be open for extension, but closed for modification.*
- Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code.

Q & A

- **Open for extension and closed for modification? That sounds very contradictory. How can a design be both?**
- 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 & A

- **I understand Observable, but how do I generally design something to be extensible, yet closed for modification?**
- 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 & A

- **How can I make every part of my design follow the Open-Closed Principle?**
- 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 typing 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 & A

- **How do I know which areas of change are more important?**
- 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.

Open close

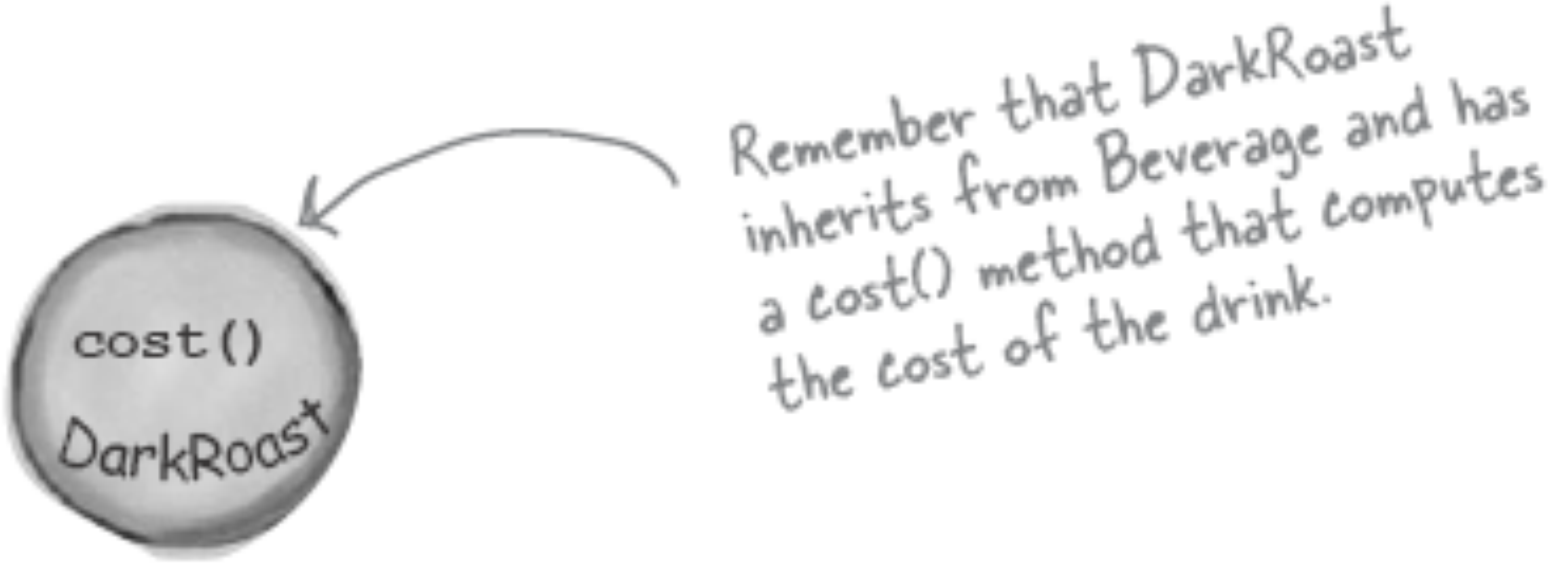
- While it may seem like a contradiction, there are techniques for allowing code to be extended without direct modification.
- Be careful when choosing the areas of code that need to be extended; applying the Open-Closed Principle EVERYWHERE is wasteful, unnecessary, and can lead to complex, hard to understand code.

Decorator Pattern-- wrappers

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

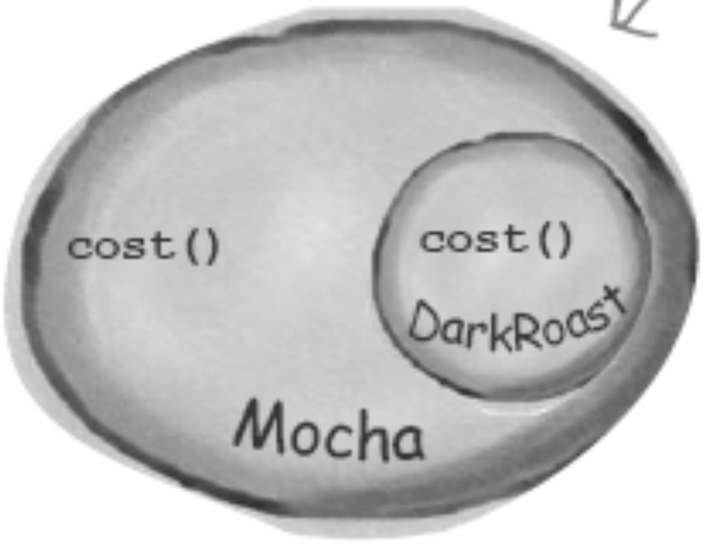
Constructing a drink order with Decorators 1

- We start with our DarkRoast object.


cost()
DarkRoast
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.

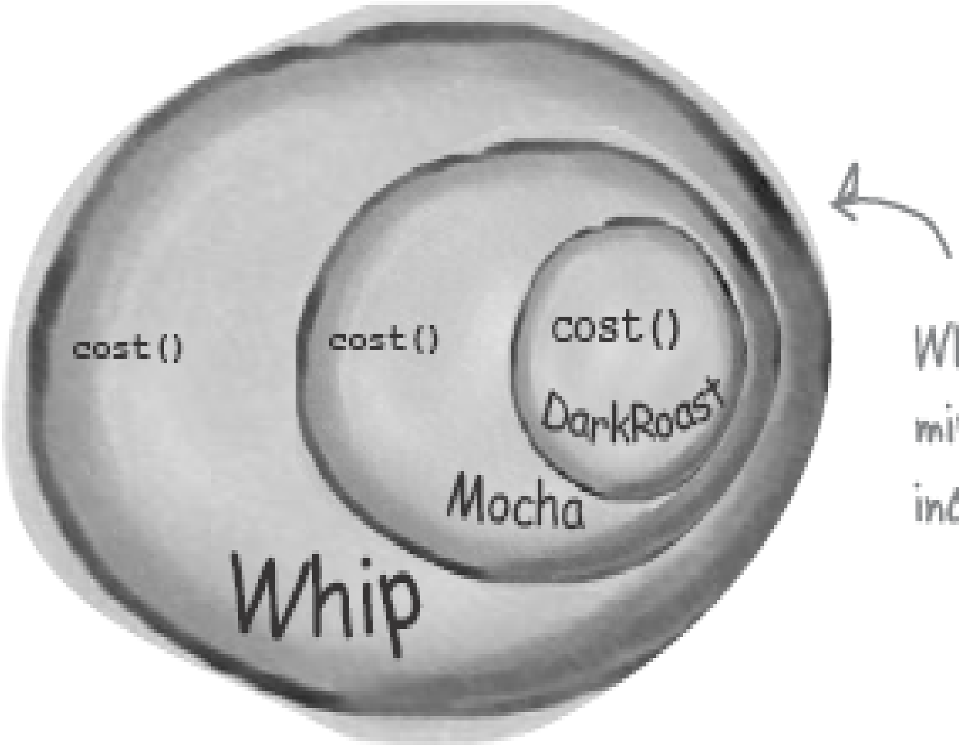


The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type..)

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.

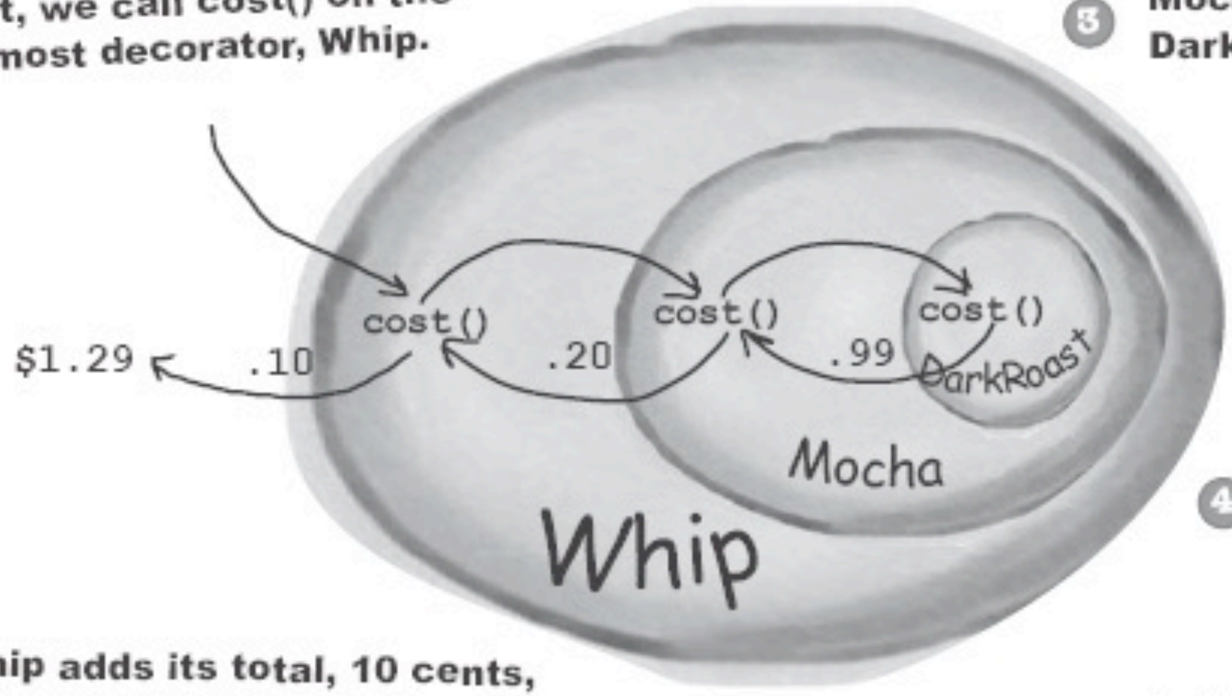
5

2 Whip calls cost() on Mocha.

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

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

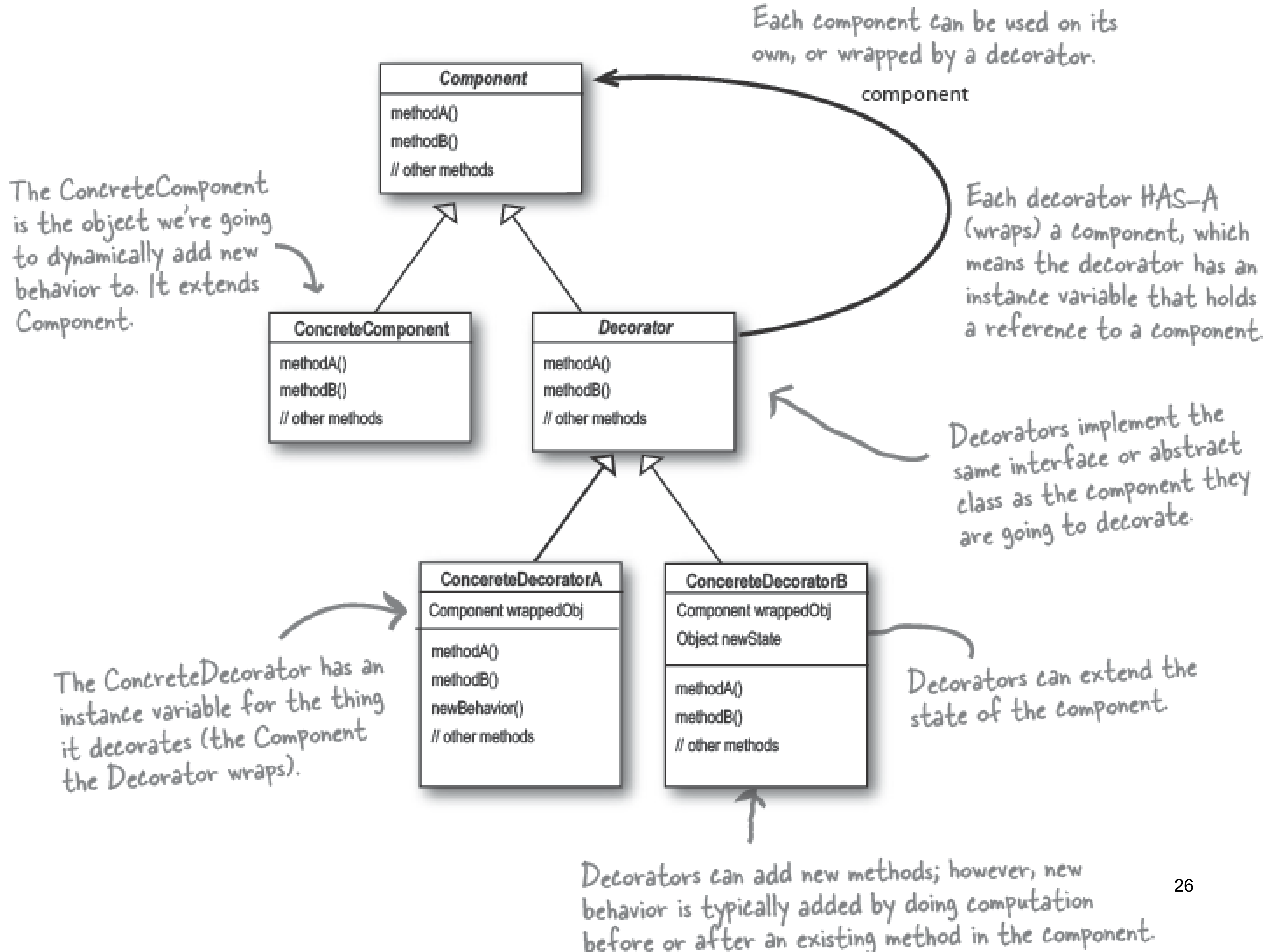
5 Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

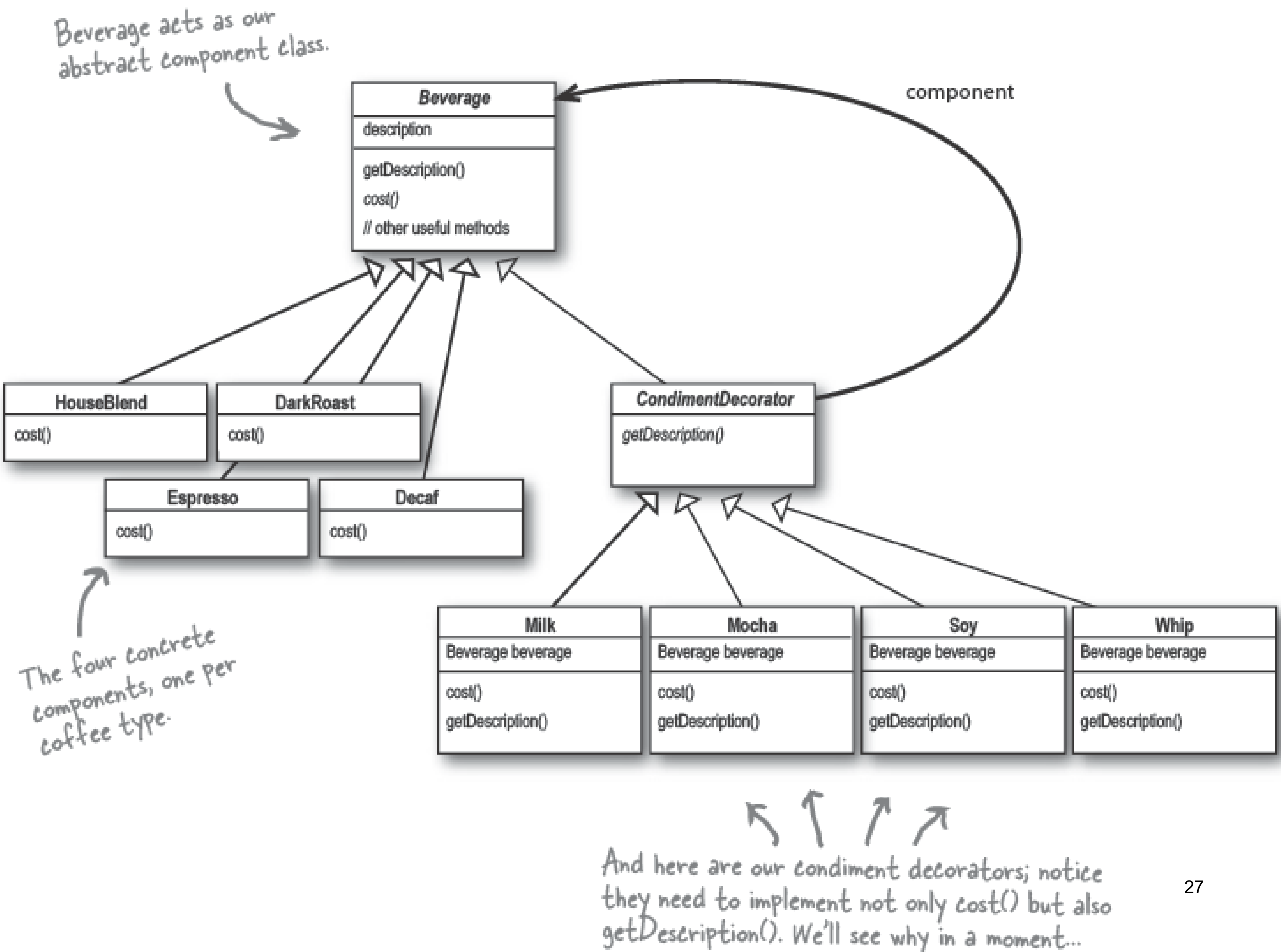
What we know so far

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

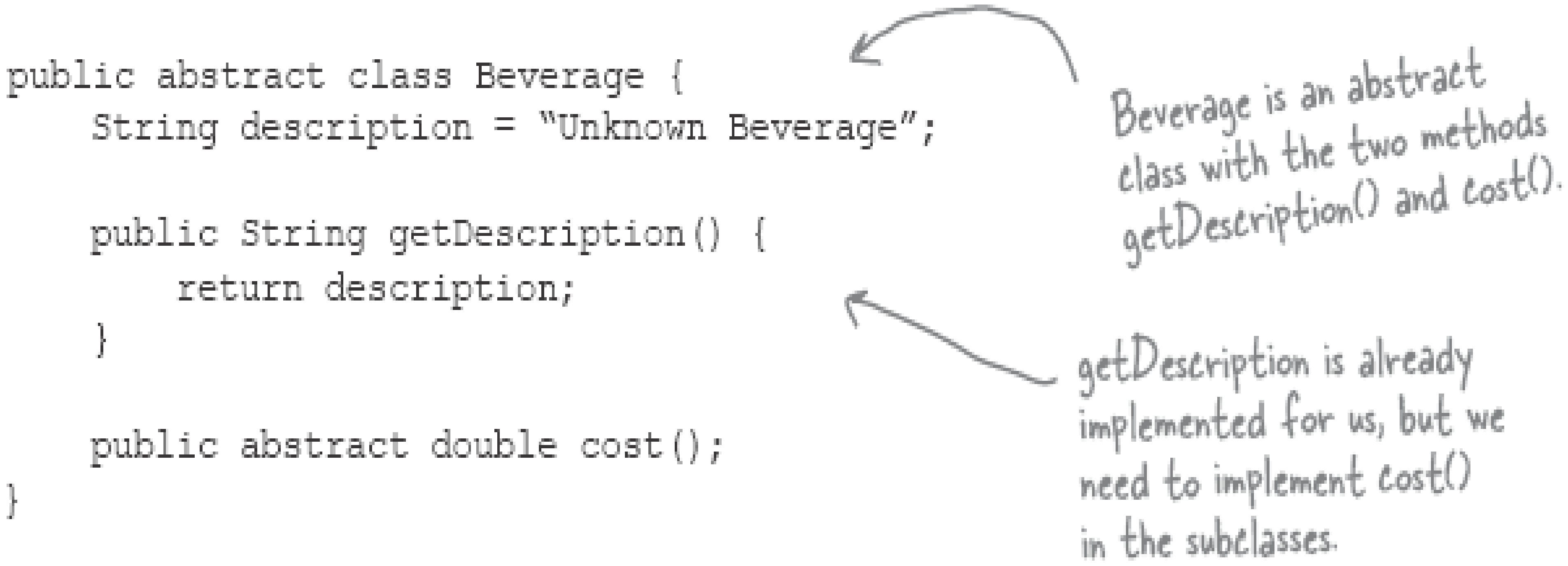
The Decorator Pattern defined

- **The Decorator Pattern** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Let’s see the class diagram.


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
methodA()
methodB()
// other methods
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcereteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcereteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods
The ConcreteDecorator has 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.


Beverage acts as our abstract component class.
Beverage
description
getDescription()
cost()
// other useful methods
component
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
CondimentDecorator
getDescription()
Milk
Beverage beverage
cost()
getDescription()
Mocha
Beverage beverage
cost()
getDescription()
Soy
Beverage beverage
cost()
getDescription()
Whip
Beverage beverage
cost()
getDescription()
The four concrete components, one per coffee type.
And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

The code



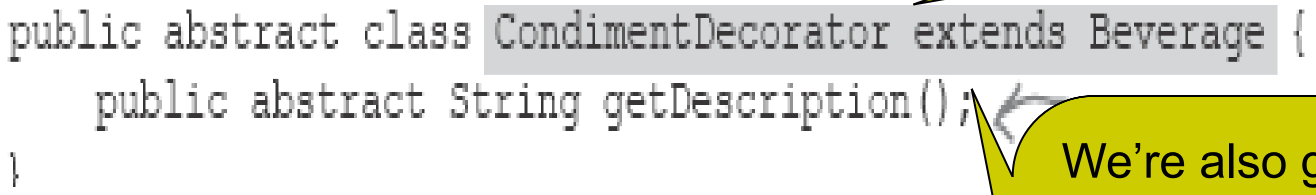
```
public abstract class Beverage {
    String description = "Unknown Beverage";

    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.

CondimentDecorator

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 - Espresso

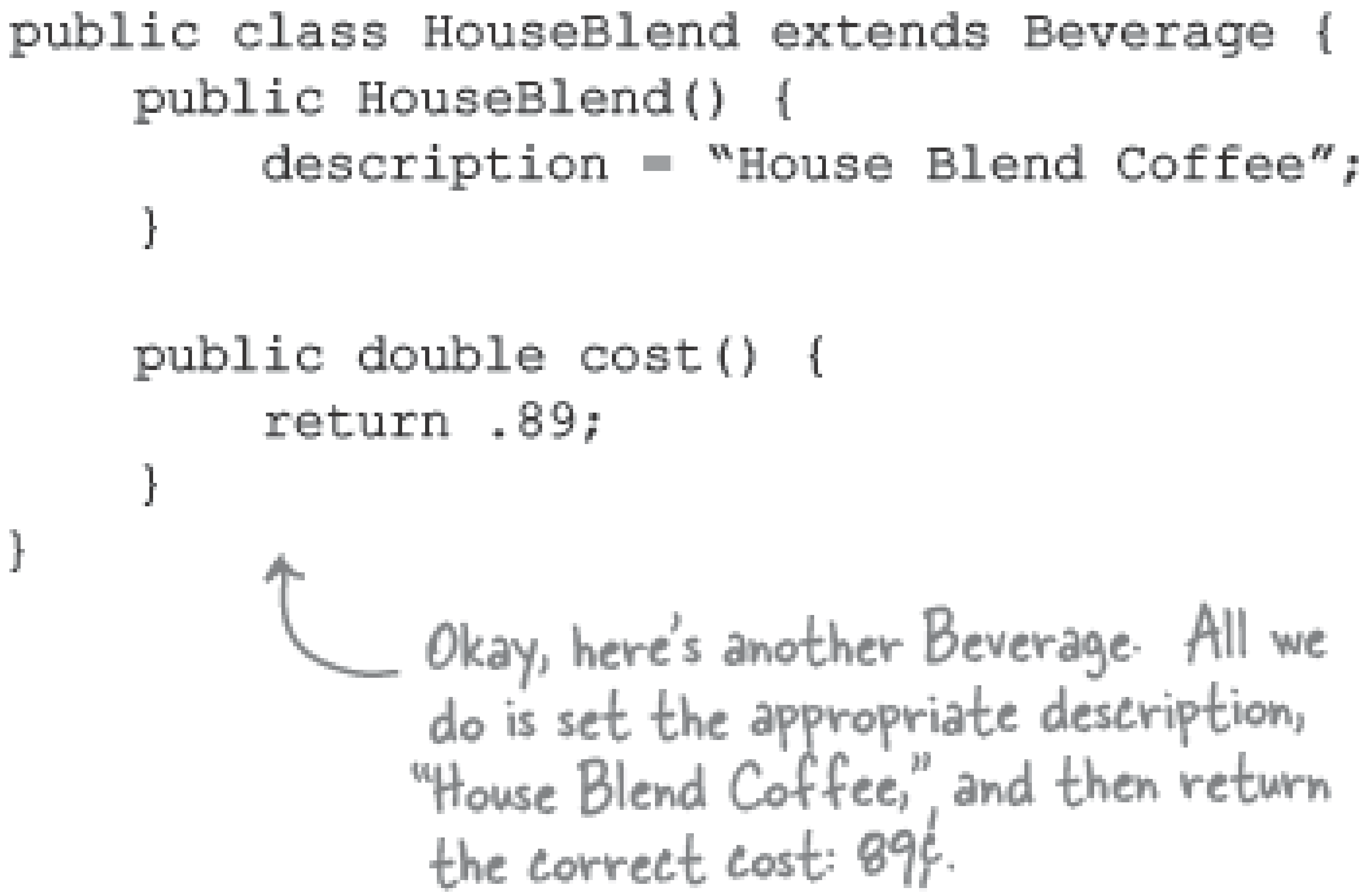
First we extend the Beverage class, since this is a beverage.

```
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.

Beverage - HouseBlend



```
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¢.

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.

Serving some coffees

```
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.

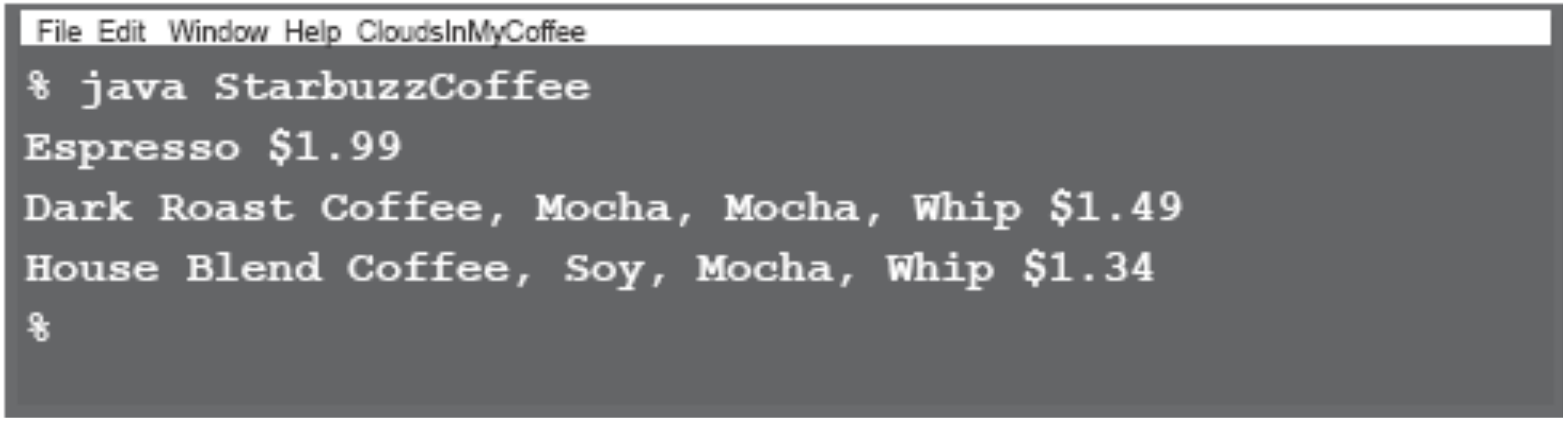
Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

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

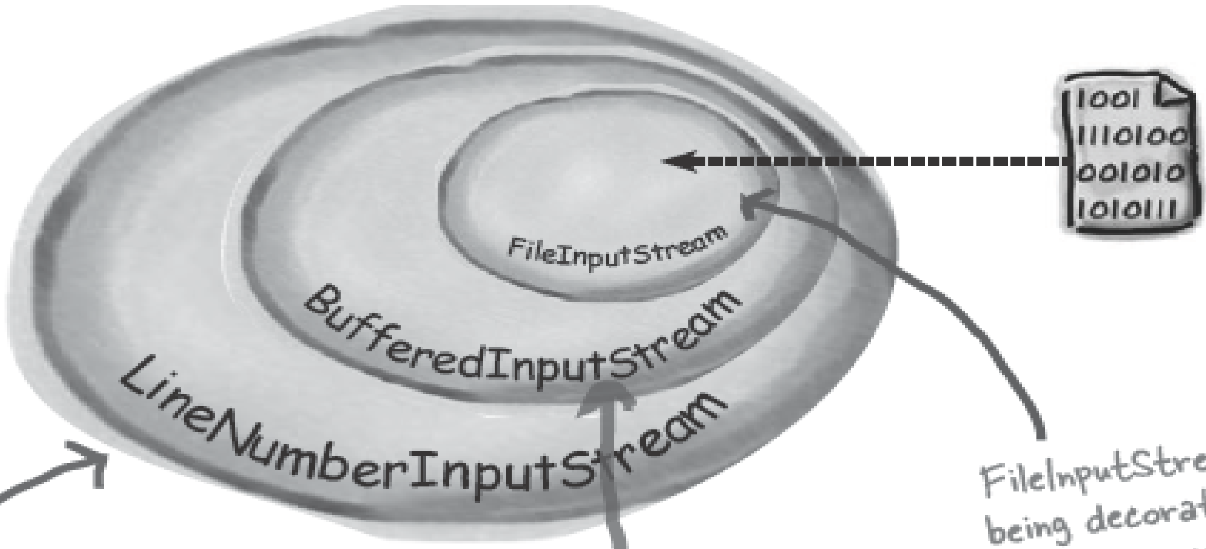
Result



```
% 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

A text file for reading.

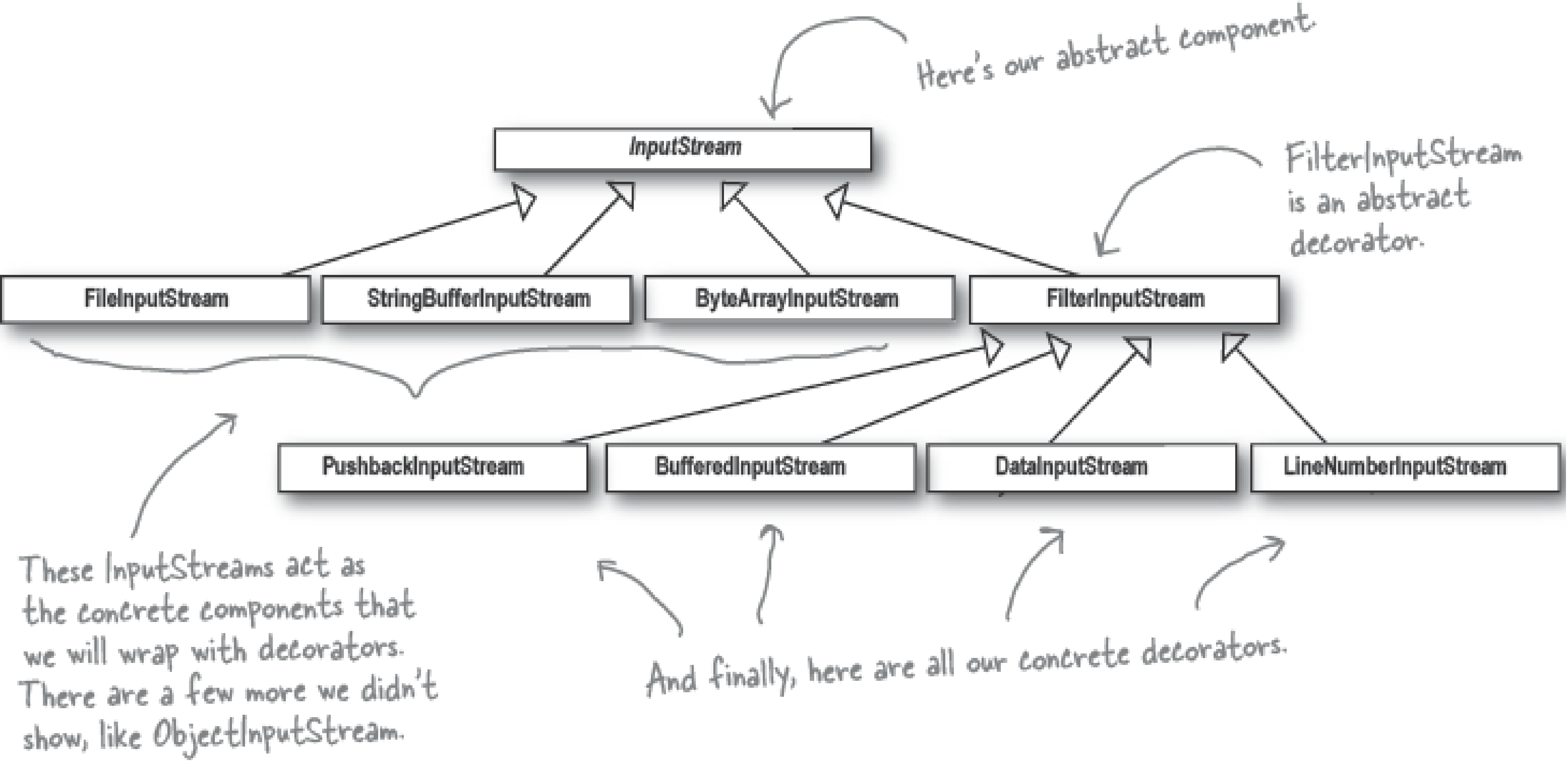


FileInputStream is the component that's being decorated. The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

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

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.

Decorating the java.io classes


Here's our abstract component
InputStream
FilterInputStream is an abstract decorator.
FileInputStream
StringBufferInputStream
ByteArrayInputStream
FilterInputStream
PushbackInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
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.

Writing your own Java I/O Decorator

- 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;
    }
}
```

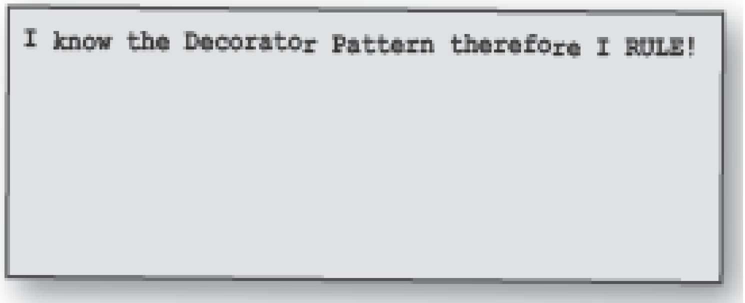
Now we
read me
byte (o
and con
represe

Test

```
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.

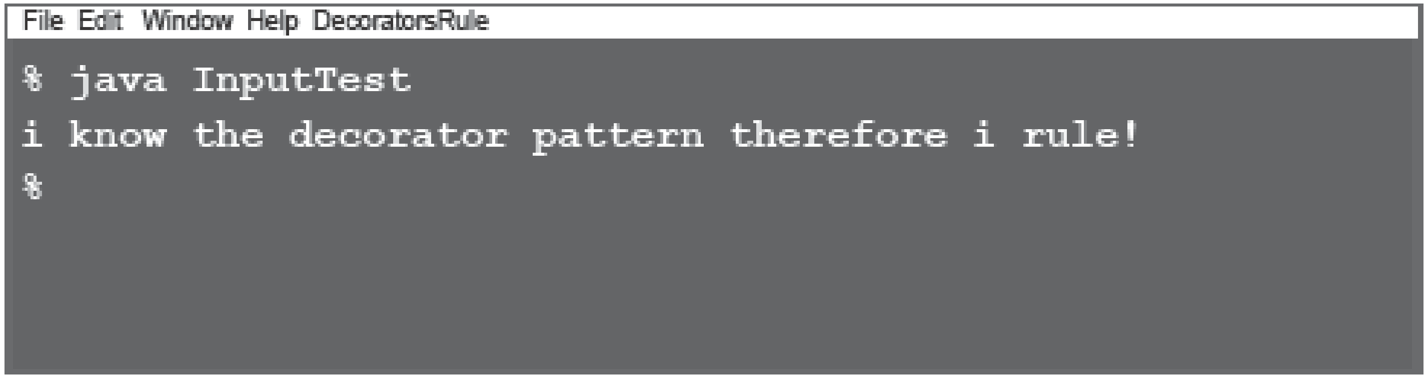
Just use the stream to read characters until the end of file and print as we go.



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

test.txt file

Result



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

Tools for your Design Toolbox

- 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.
- OO Principles
  - Decorator

Intent

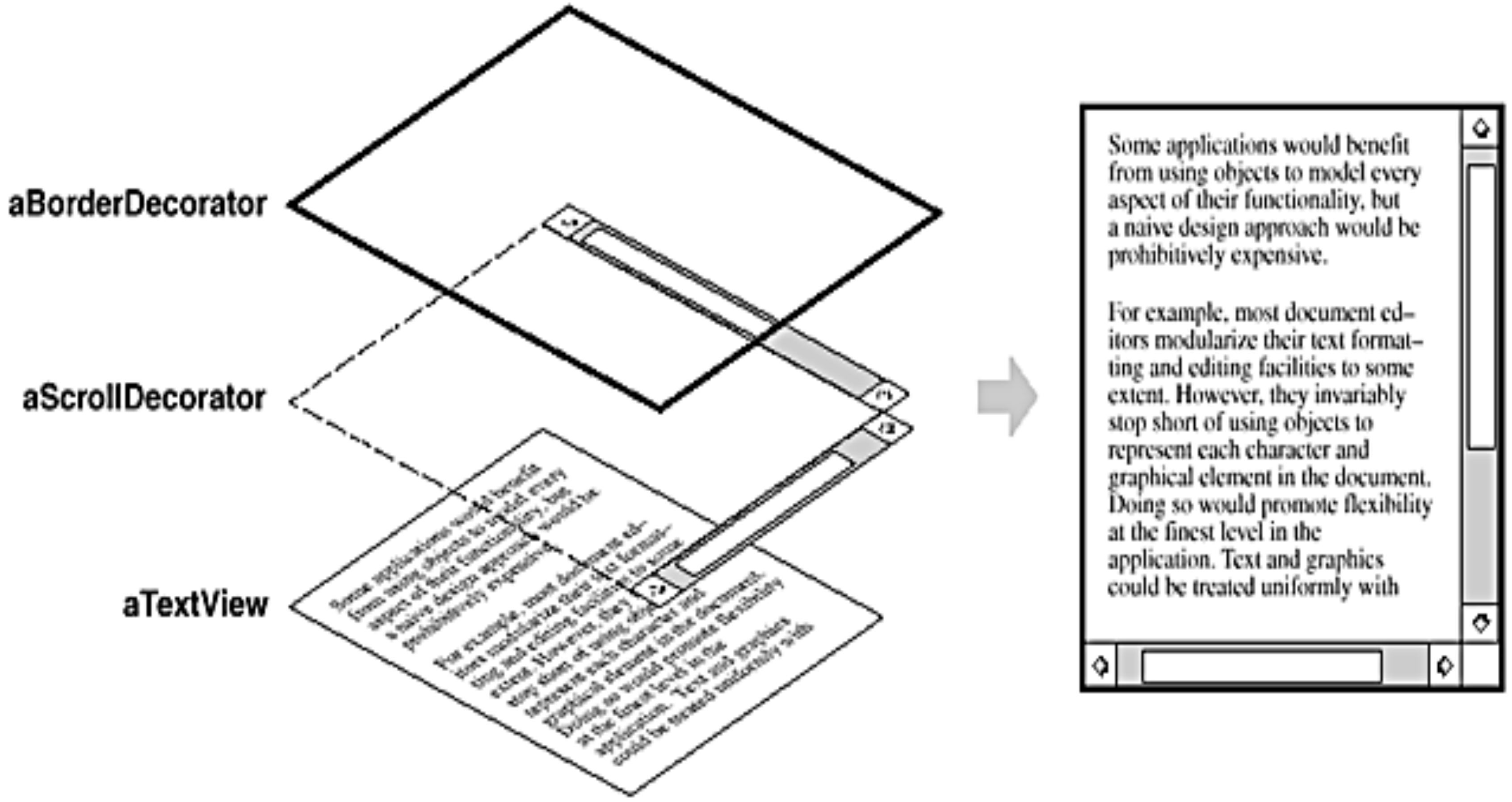
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Also Known As

- Wrapper

Motivation

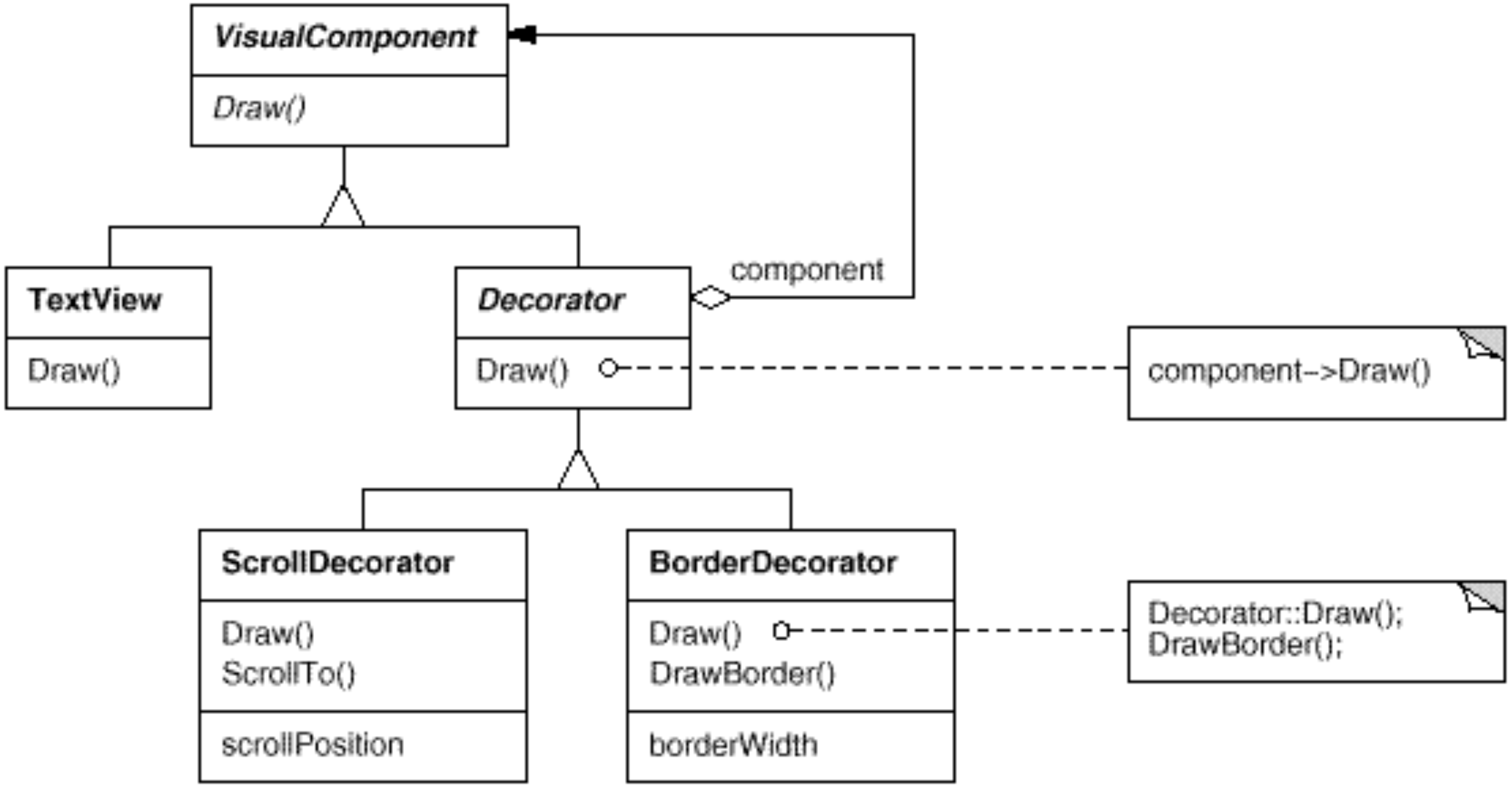
- Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.
- One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with a border.
- A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**.



Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

Structure



VisualComponent
Draw()
TextView
Draw()
Decorator
Draw()
component
component->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();

Implemetation

- *Keeping Component classes lightweight.*

To ensure a conforming interface, components and decorators must descend from a common Component class. It's important to keep this common class lightweight; that is, it should focus on defining an interface, not on storing data. The definition of the data representation should be deferred to subclasses; otherwise the complexity of the Component class might make the decorators too heavyweight to use in quantity. Putting a lot of functionality into Component also increases the probability that concrete subclasses will pay for features they don't need.

- *Changing the skin of an object versus changing its guts.*

We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts. The Strategy pattern is a good example of a pattern for changing the guts. Strategies are a better choice in situations where the Component class is intrinsically heavyweight, thereby making the Decorator pattern too costly to apply. In the Strategy pattern, the component forwards some of its behavior to a separate strategy object. The Strategy pattern lets us alter or extend the component's functionality by replacing the strategy object.

Reviews-1

- 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.

Reviews-2

- 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.)

Reviews-3

- 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.