Lecture 03: The Decorator Pattern

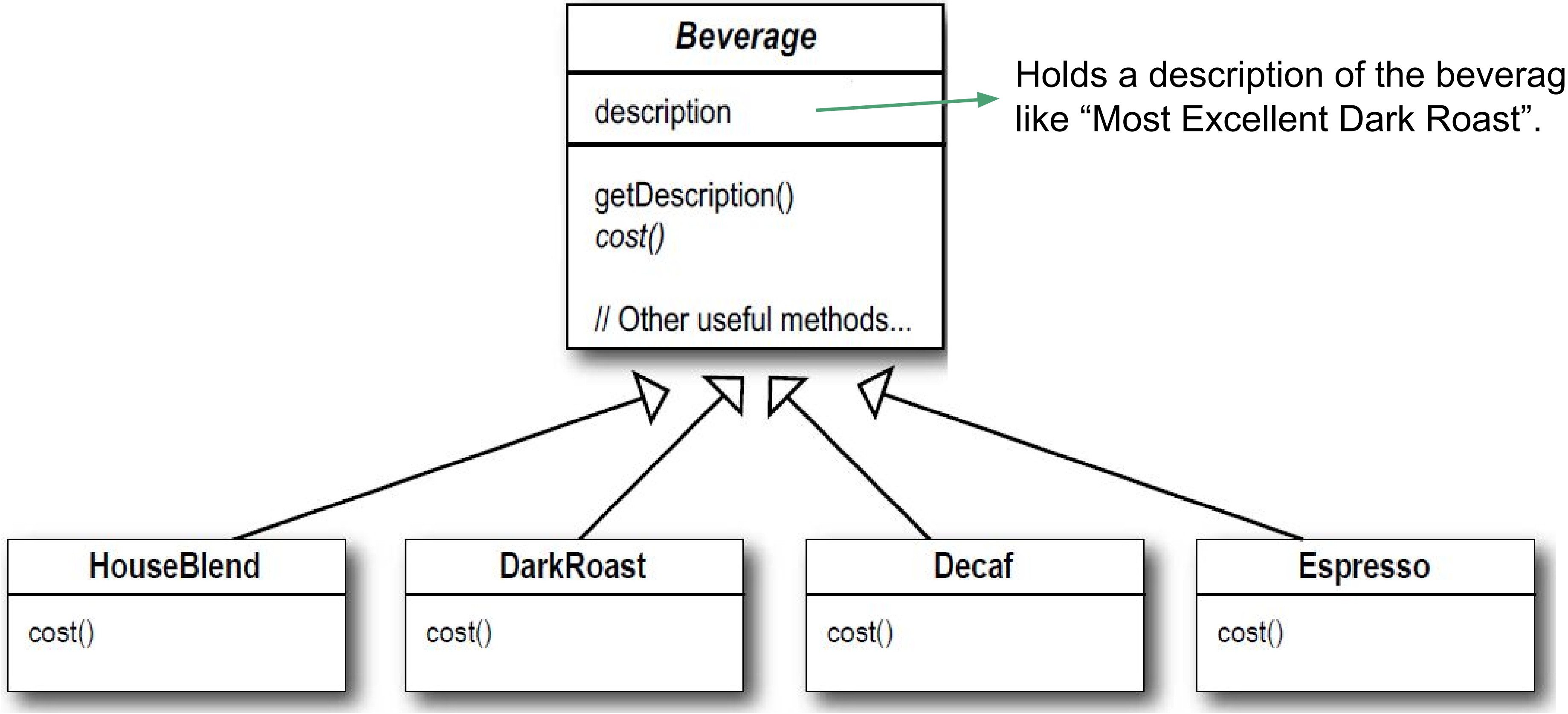
SE313, Software Design and Architecture
Damla Oguz

Chapter 3: The Decorator Pattern

- We will learn how to decorate our classes at runtime using a form of object composition.
- Once we know the techniques of decorating, we will be able to give our (or someone else's) objects new responsibilities without making any code changes to the underlying classes.

Welcome to Starbuzz Coffee

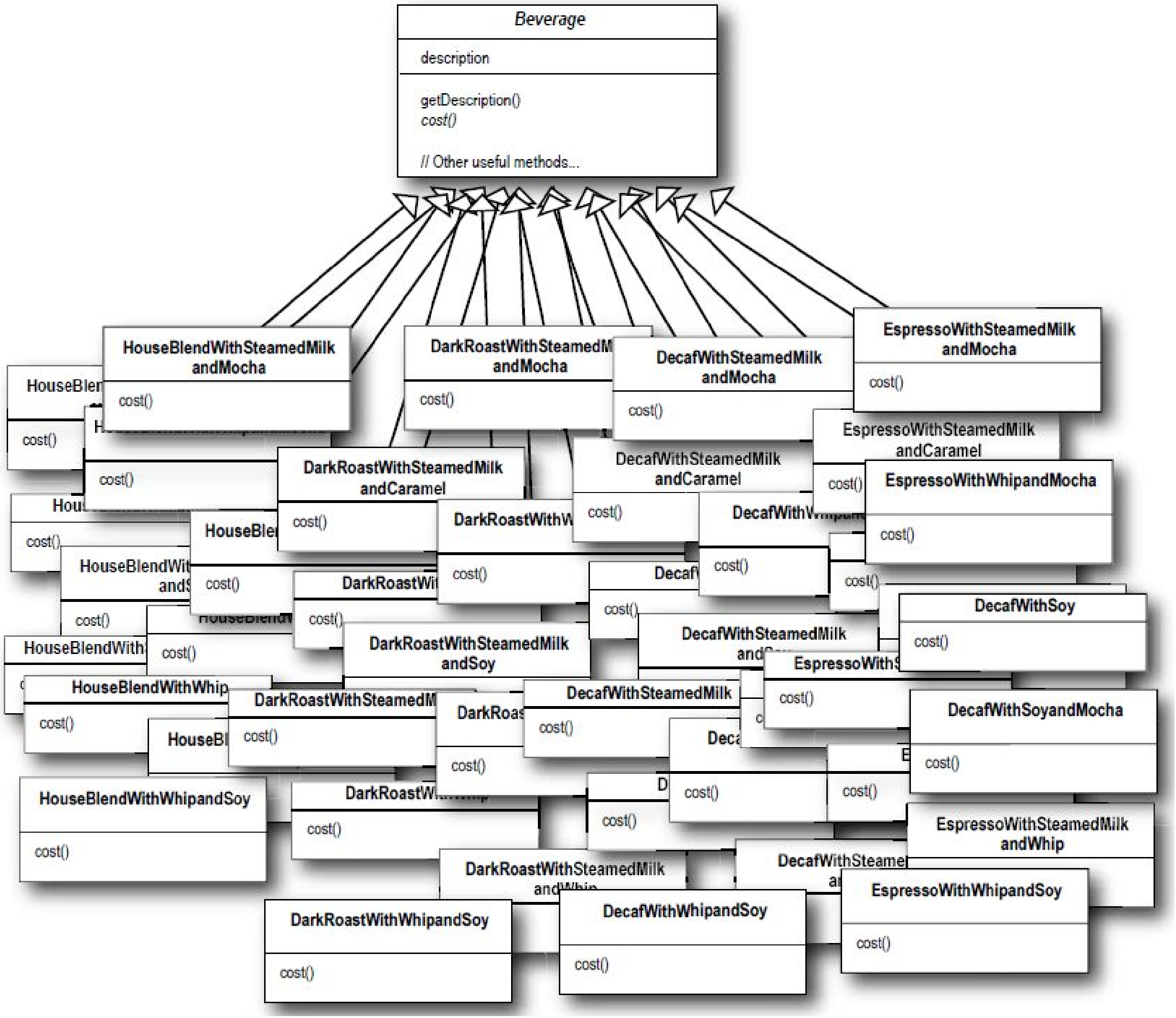
- Starbuzz Coffee is the fastest growing coffee shop.
- So, they are scrambling to update their ordering systems to match their beverage offerings.
- When they first went into business they designed their classes like this…



Holds a description of the beverage, like “Most Excellent Dark Roast”.

Welcome to Starbuzz Coffee (cont.)

- In addition to our coffee, we can also ask for several condiments like steamed milk, soy, and mocha, 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.
- Their first attempt is…

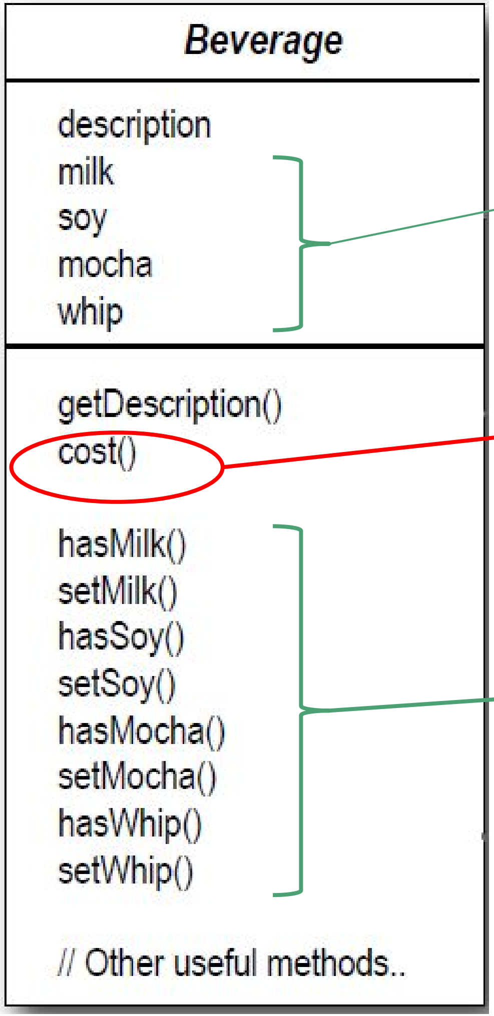

Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
DecafWithSteamedMilk andMocha
EspressoWithSteamedMilk andMocha
EspressoWithSteamedMilk andCaramel
EspressoWithWhipandMocha
DarkRoastWithSteamedMilk andCaramel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

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

- What happens when the price of milk goes up?
- What do they do when they add a new caramel topping?

A new idea

- We can add instance variables to represent whether or not each beverage has milk, soy, mocha and whip… to the Beverage base class.



Beverage

description
milk
soy
mocha
whip

getDescription()
cost()

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

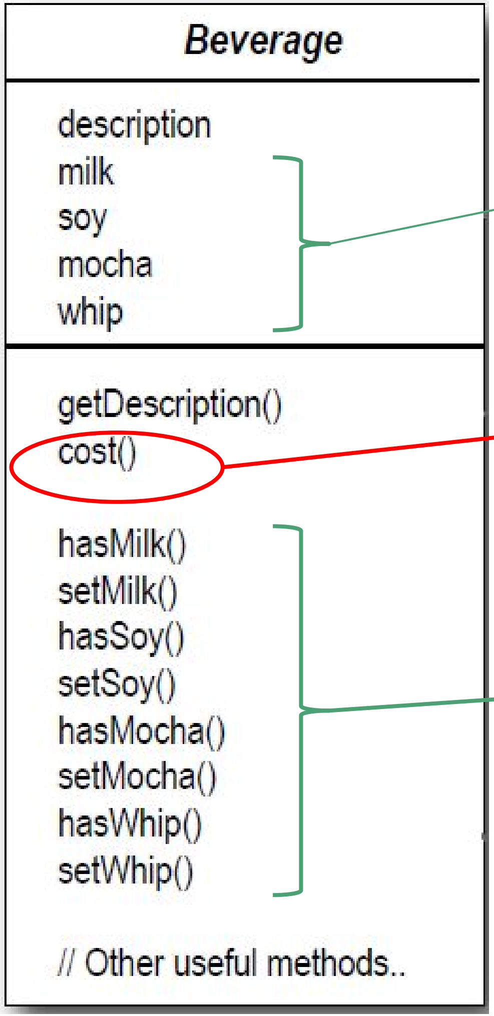
// Other useful methods..

New boolean values for each condiment.

These get and set the boolean values for the condiments.

- 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.

What do you think about this design?



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.

These get and set the boolean values for the condiments.

- We'll implement cost() in Beverage (instead of keeping it abstract).
- Subclasses will still override cost(), but they will also invoke the super version.
- 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.

What do you think about this design?

Examples which show the drawbacks of 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**?

About inheritance

- While inheritance is powerful, it doesn’t always lead to the most flexible or maintainable designs.
- When we inherit behavior by subclassing, that behavior is set statically at compile time. In addition, all subclasses must inherit the same behavior.
- If however, we can extend an object’s behavior through composition, then we can do this dynamically at runtime.

Power of composition

- We can add multiple new responsibilities to objects through composition. And, we don’t have to touch the code!
- By dynamically composing objects, we can add new functionality by writing new code rather than altering existing code.
- 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.

Night

Day

The Open-Closed Principle

Design 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.**
- **Feel free to extend our classes with any new behavior you like.**
- 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.**

The Open-Closed Principle (cont.)

Q: Open for extension and closed for modification? How can a design be both?

- 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.

The Open-Closed Principle (cont.)

Q: 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 our code from being modified by supplying a means of extension.
- In this lecture we'll see a good example of using the Decorator pattern to follow the Open-Closed principle.

The Open-Closed Principle (cont.)

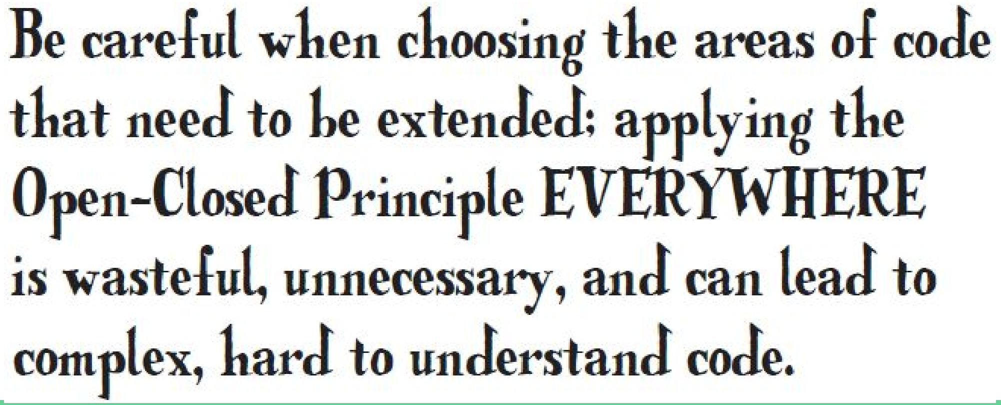
Q: How can I make every part of my design follow the Open-Closed Principle?

- Usually, you can’t.
- 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.

The Open-Closed Principle (cont.)

Q: 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 knowing the domain you are working in.
- Looking at other examples will help you learn to identify areas of change in your own designs.



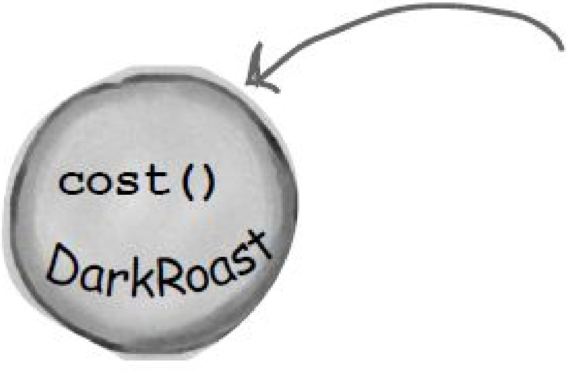
Be careful when choosing the areas of code that need to be extended; applying the Open-Closed Principle EVERYWHERE is wasteful, unnecessary, and can lead to complex, hard to understand code.

Meet the Decorator Pattern

- Okay, we’ve seen that representing our beverage plus condiment pricing scheme with inheritance has not worked out very well.
- So, here’s what we’ll do instead: we’ll start with a beverage and “decorate” it with the condiments at runtime.
- For ex., 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

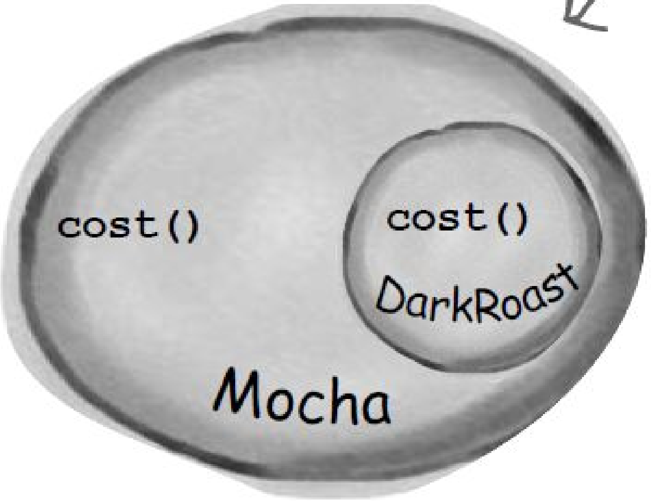
Constructing a drink order with Decorators

❶ **We start with our DarkRoast object.**



Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

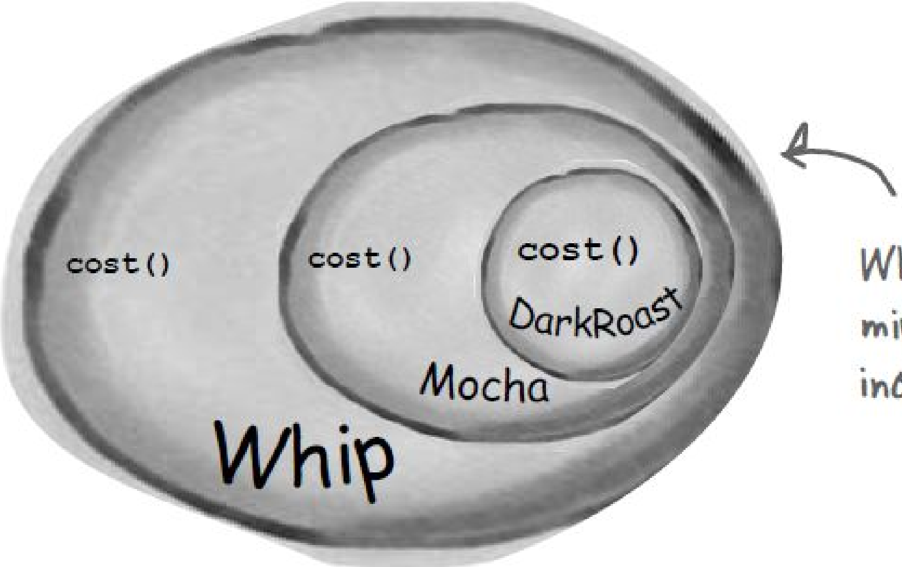
❷ The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.



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

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

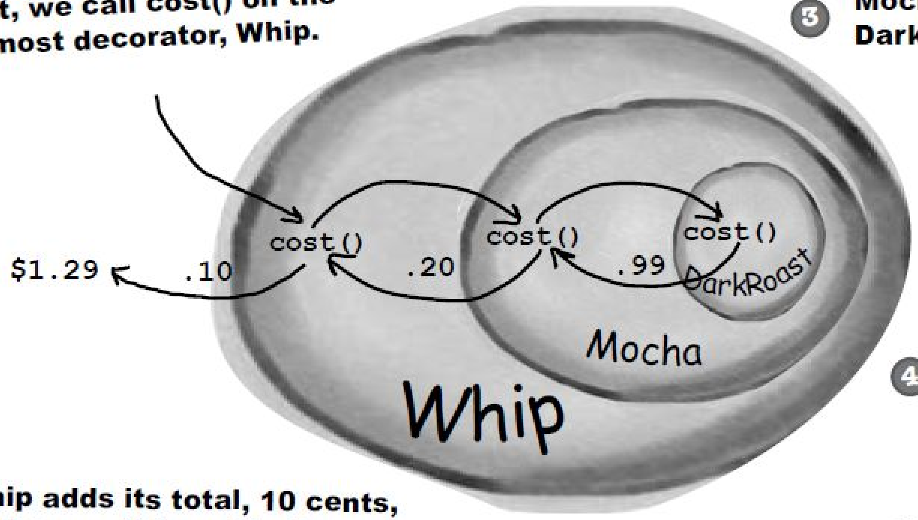
❸ The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.



Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

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

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



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

2 Whip calls cost() on Mocha.

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

3 Mocha calls cost() on DarkRoast.

4 DarkRoast returns its cost, 99 cents.

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

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

Here’s what we know so far…

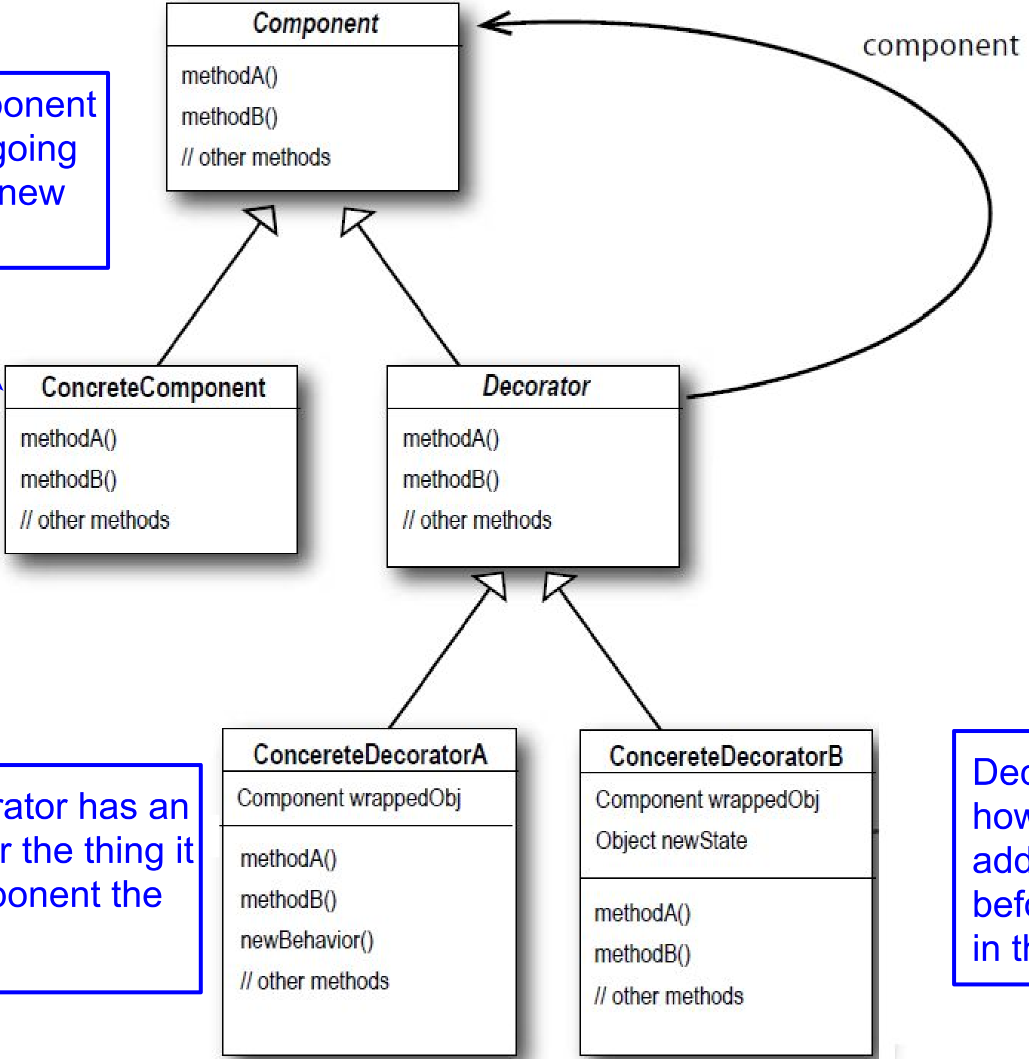
- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- The decorator has the same supertype as the object it decorates, so we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own 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 description

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

The ConcreteComponent is the object we're going to dynamically add new behavior to.

Each component can be used on its own, or wrapped by a decorator.



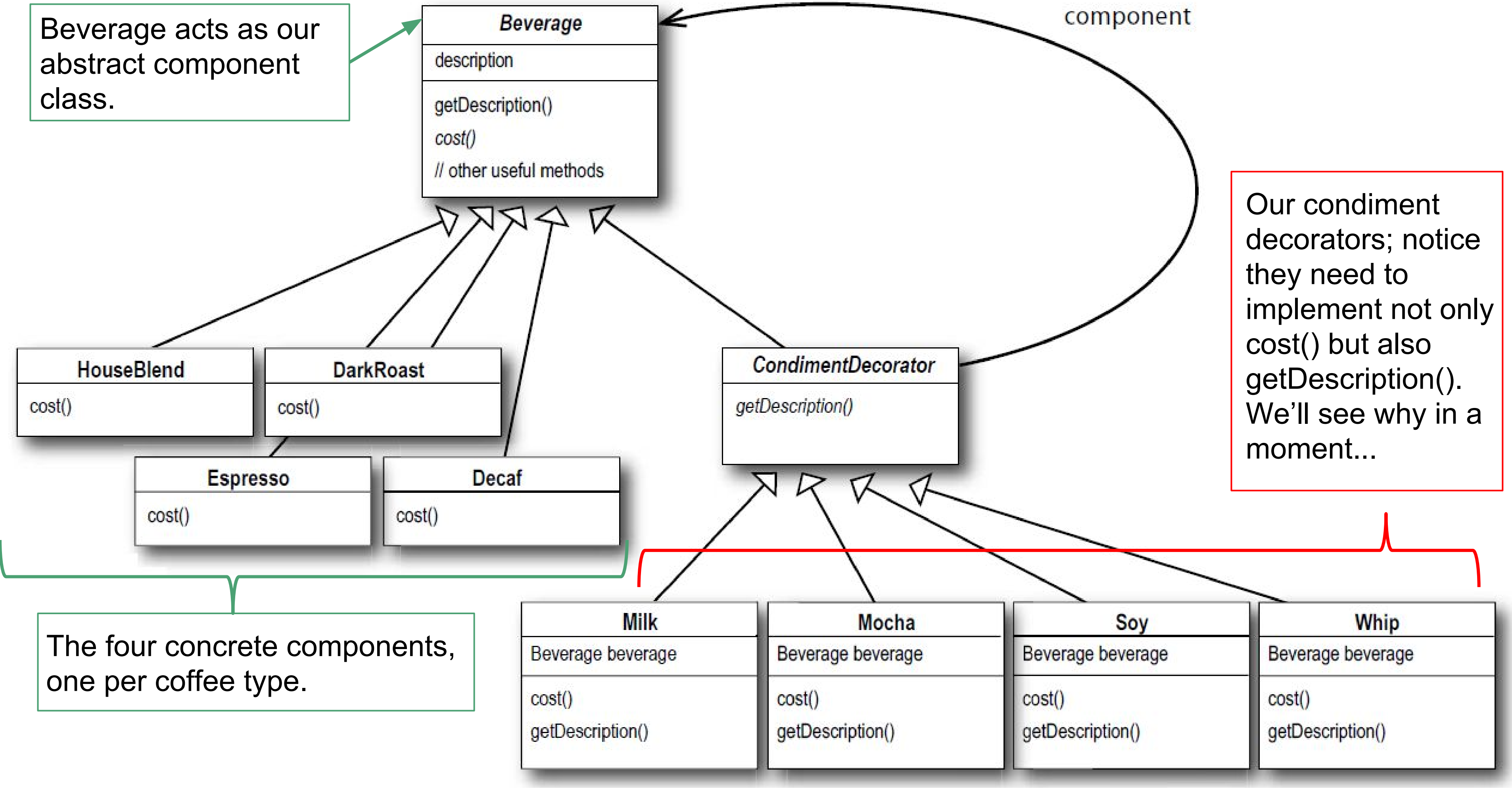
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.

The ConcreteDecorator has an instance variable for the thing it decorate (the Component the Decorator wraps).

Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

Decorating our Beverages

Let’s work our Starbuzz beverages into this framework…



Beverage acts as our abstract component class.

Our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

The four concrete components, one per coffee type.

Some confusion over inheritance versus composition

- 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*.
- Decorators need the same "interface" as the components they wrap because they need to stand in place of the component.
- 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.

Some confusion over inheritance versus composition

- 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.
- And because we are using object composition, we get a whole lot more flexibility about how to mix and match condiments and beverages.

Some confusion over inheritance versus composition

- 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.
- 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.

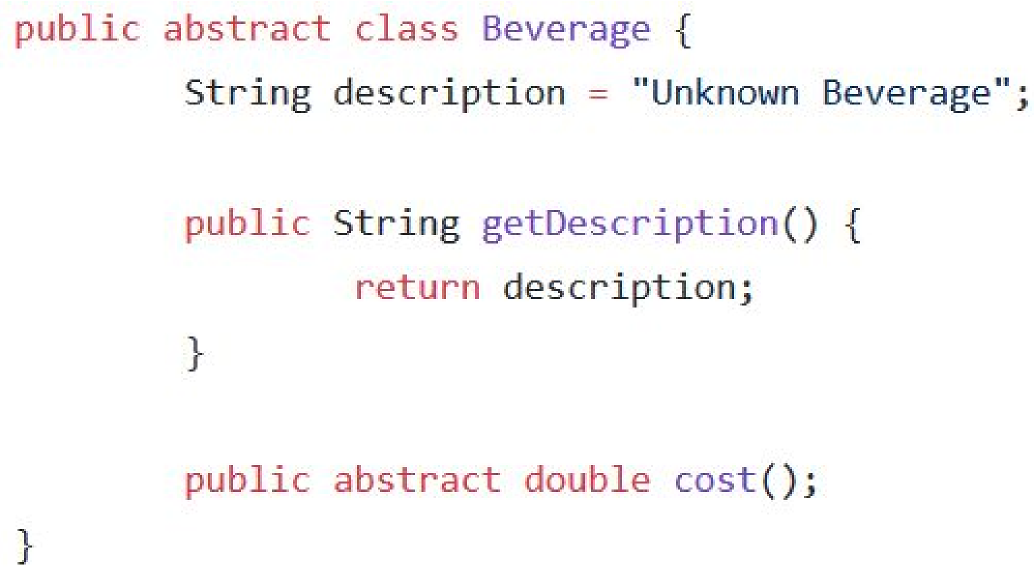
Some confusion over inheritance versus composition

Q: 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?

- 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.

Writing the Starbuzz code

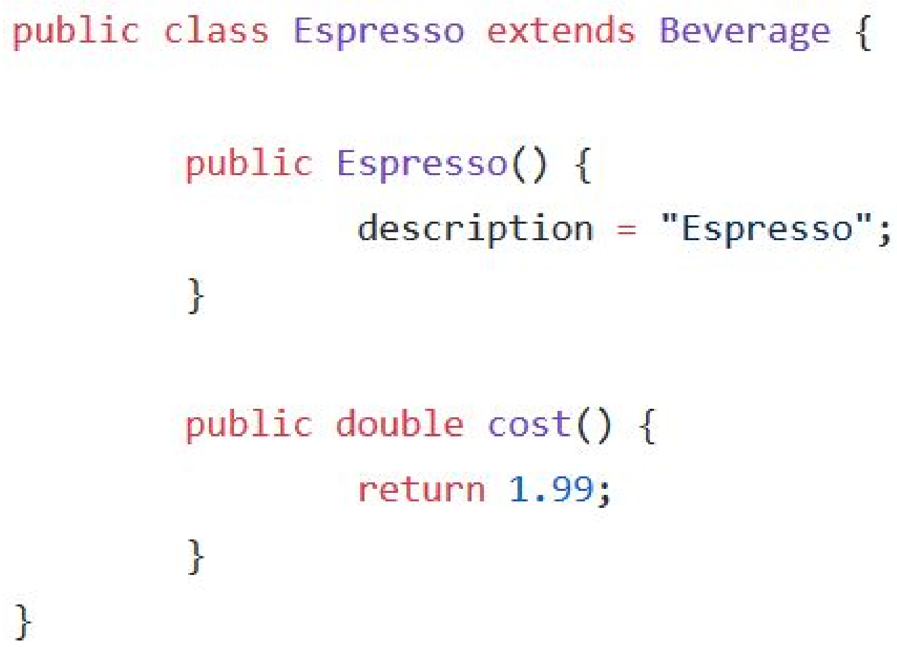
Let’s start with the Beverage class, which doesn’t need to change from Starbuzz’s original design.



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

        public String getDescription() {
                return description;
        }

        public abstract double cost();
}
```

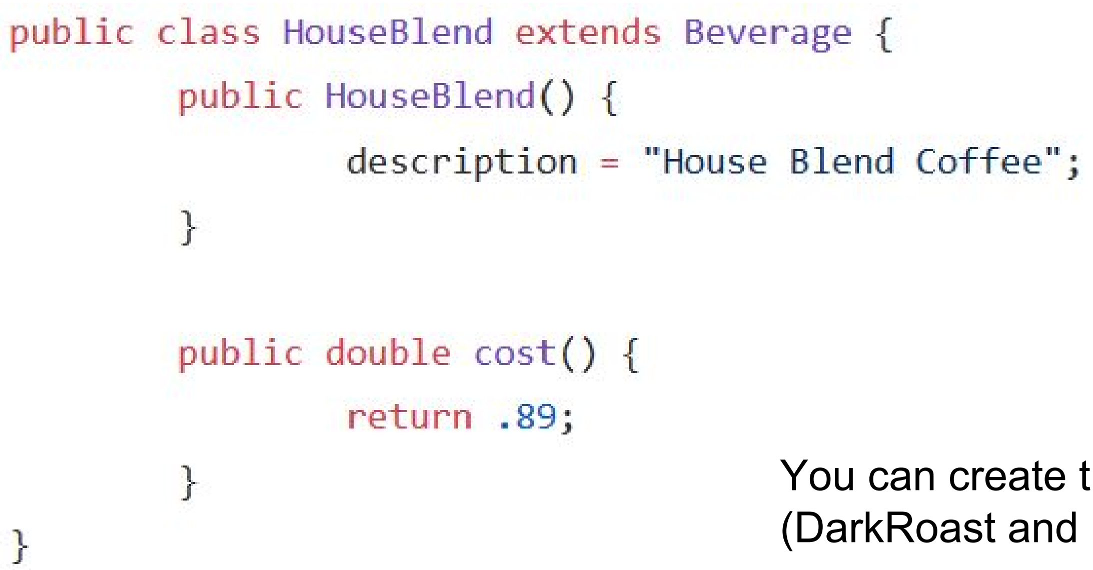
Coding beverages

Let’s implement some beverages. We’ll start with Espresso.



```
public class Espresso extends Beverage {

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

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

Coding beverages (cont.)



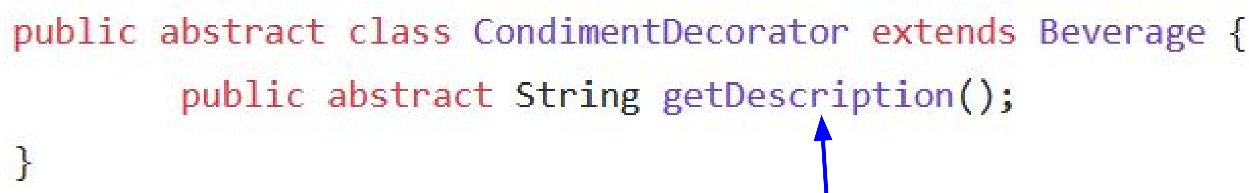
```
public class HouseBlend extends Beverage {
        public HouseBlend() {
                description = "House Blend Coffee";
        }

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

Here’s another Beverage. All we do is set the appropriate description, “House Blend Coffee,” and then return the correct cost..

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

Writing the Starbuzz code (cont.)

Let’s implement the abstract class for the Condiments (Decorator):



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

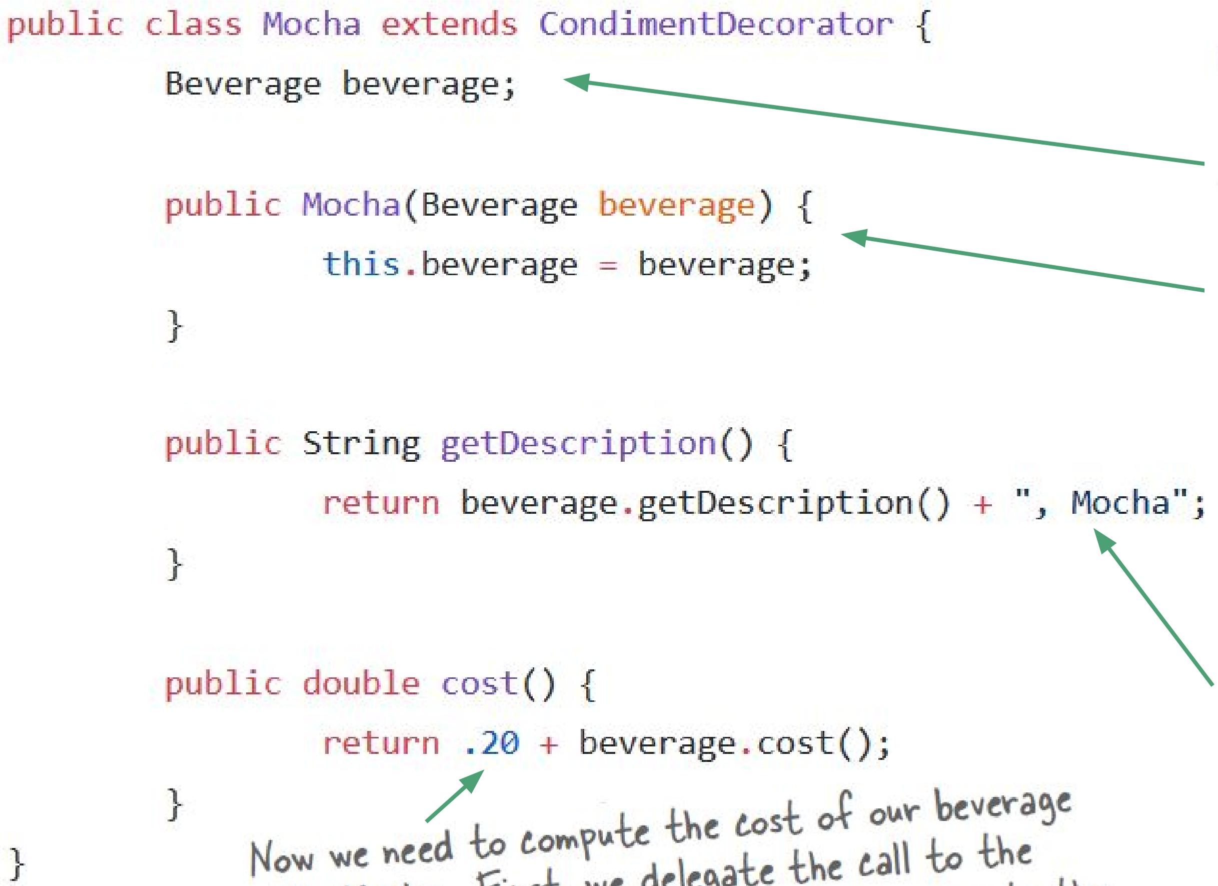
First, we need to be interchangeable with a Beverage, so we extend the Beverage class.

We’re also going to require that the condiment decorators all reimplement the getDescription() method.

Coding condiments

Now it’s time to implement the concrete decorators.

- Let’s begin with the Mocha.



```java
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.

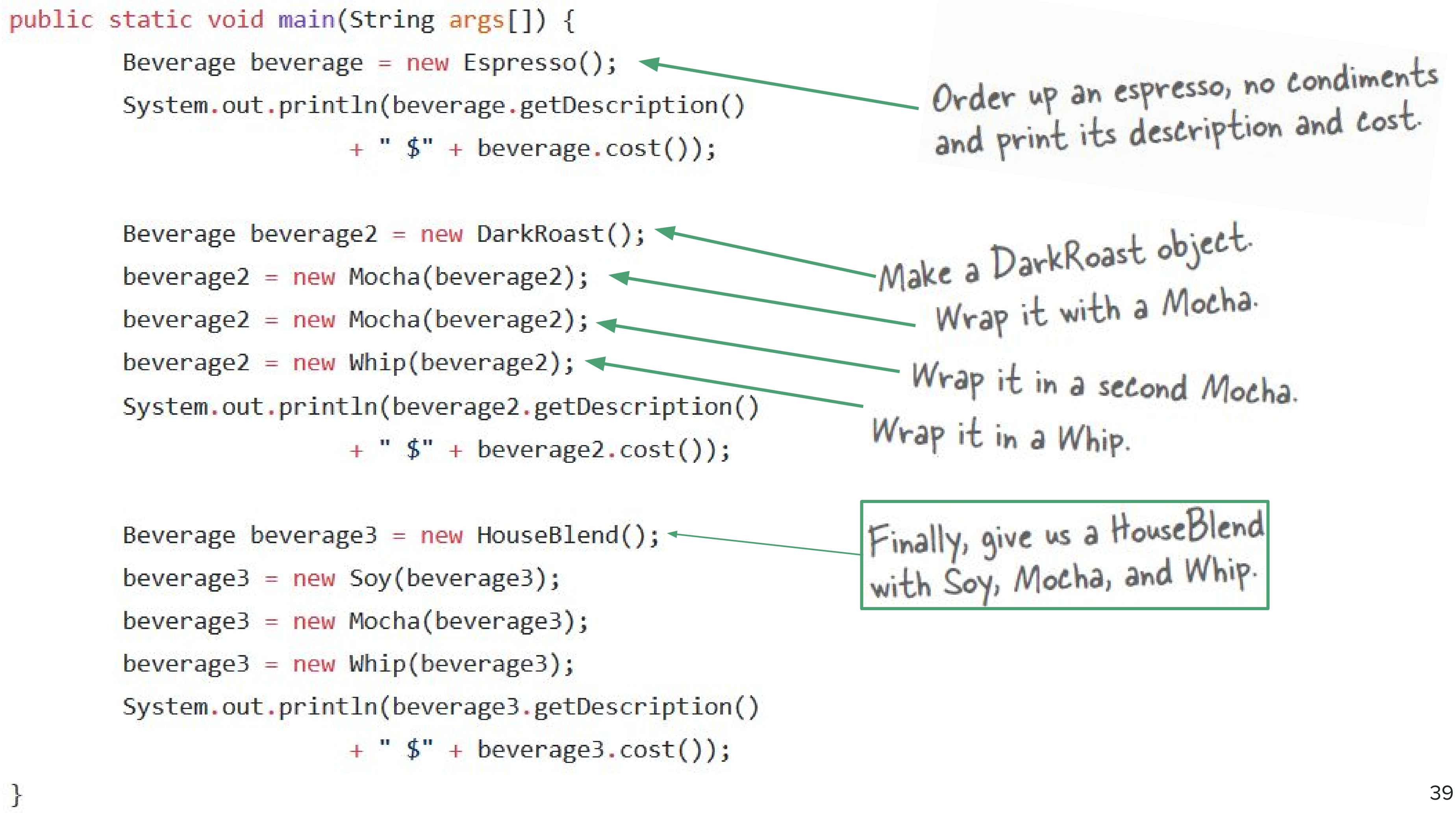
(2) A way to set this instance variable to the object we are wrapping. Here, we're going to pass the beverage we're wrapping to the decorator's constructor.

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

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



```java
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.

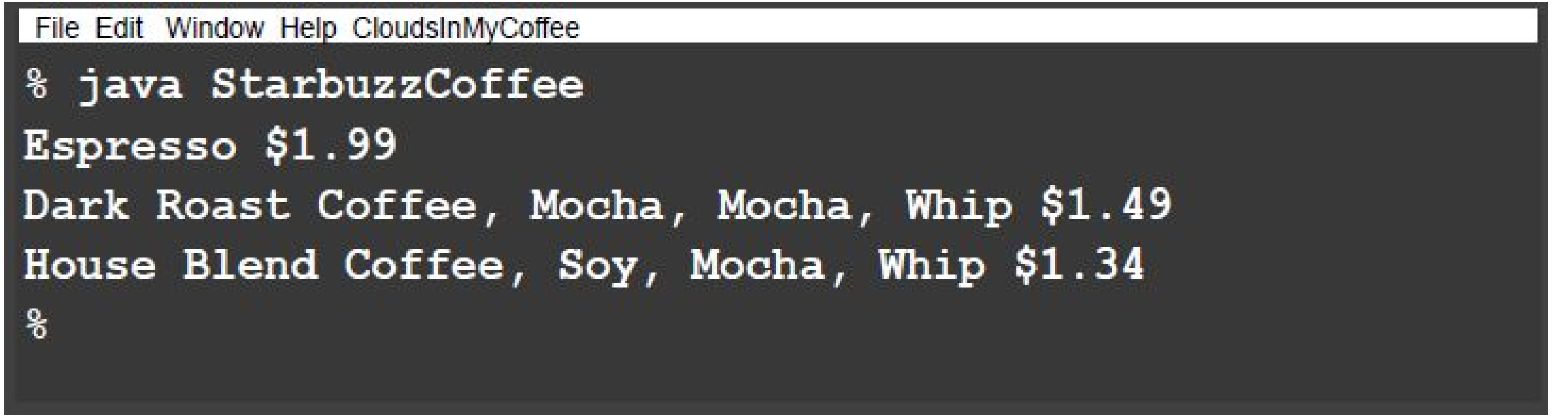
Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

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

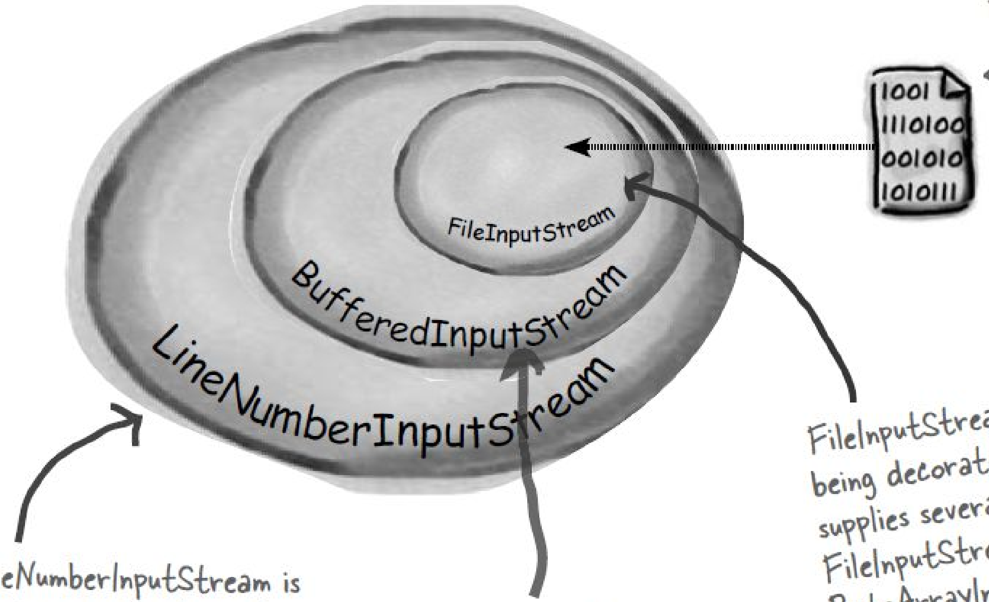
Now, let’s get those orders in:



```
% 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 large number of classes in the java.io package is *overwhelming*.
- The java.io package is largely based on Decorator.
- Here’s a typical set of objects that use decorators to add functionality to reading data from a file…



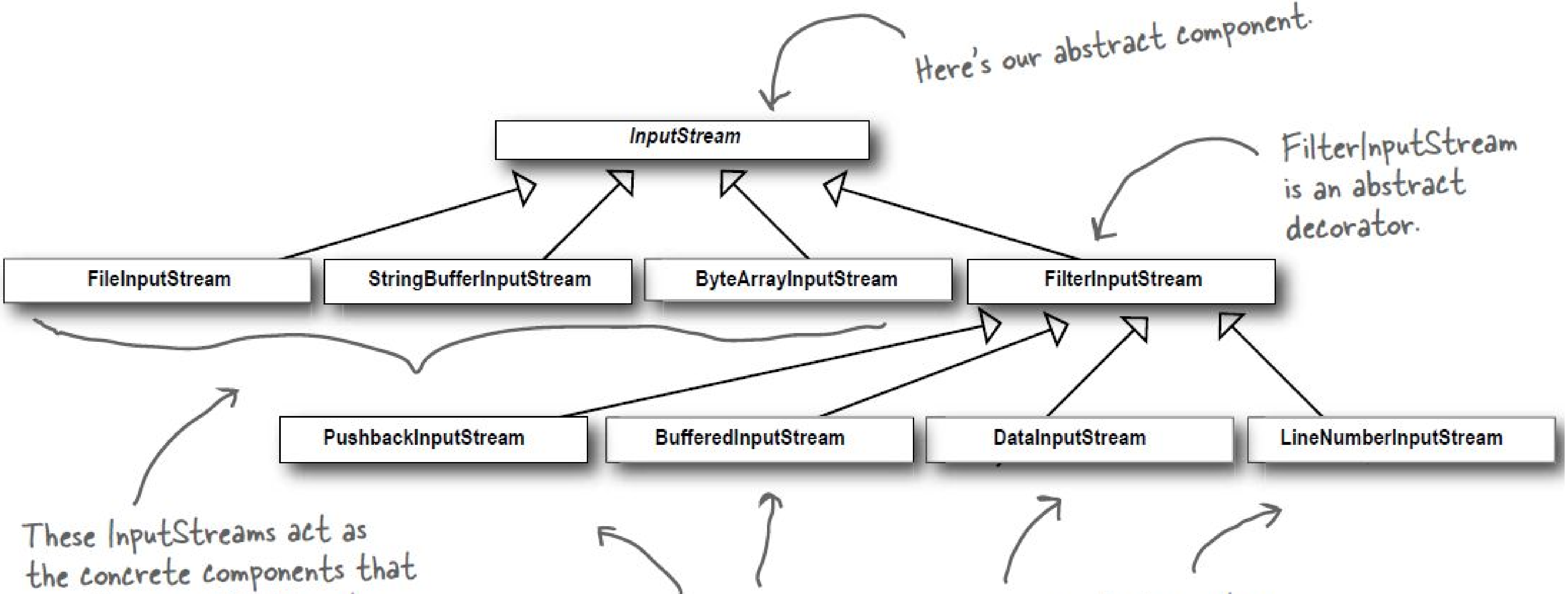
A text file for reading.

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.

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



InputStream

FileInputStream

StringBufferInputStream

ByteArrayInputStream

FilterInputStream

PushbackInputStream

BufferedInputStream

DataInputStream

LineNumberInputStream

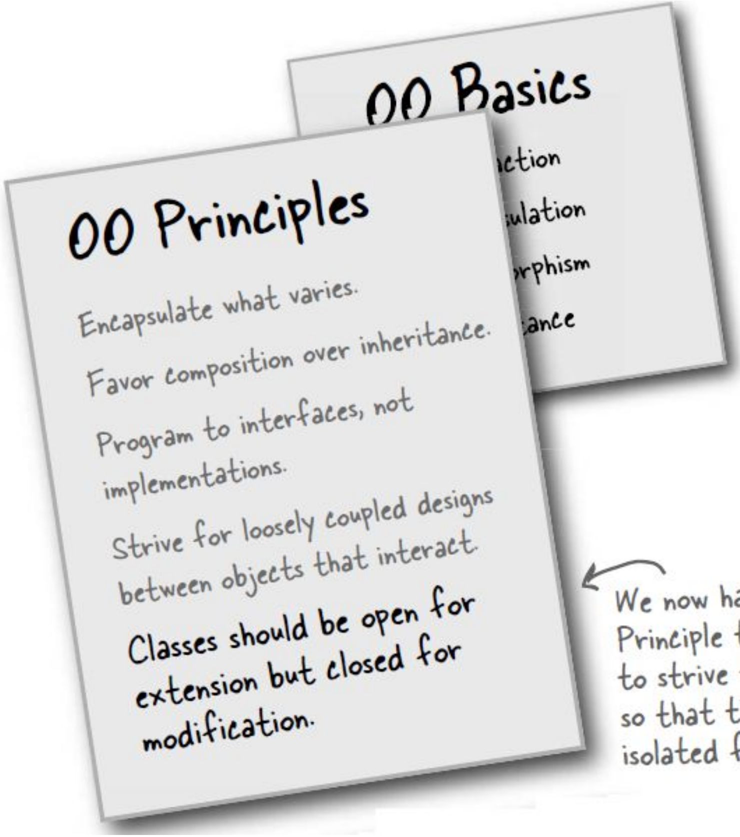
Here's our abstract component.

FilterInputStream is an abstract decorator.

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.

Tools for your Design Toolbox



OO Basics

ction

ulation

rphism

ance

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

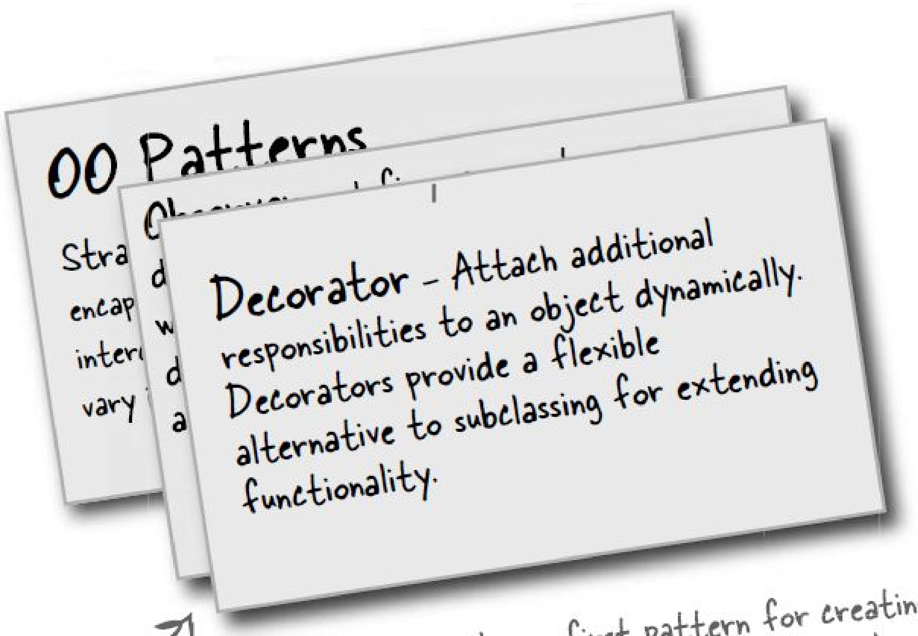
Program to interfaces, not implementations.

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.

Tools for your Design Toolbox (cont.)



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?

References

- Material in this lecture is taken from Freeman, E., Robson, E., Bates, B., & Sierra, K., *Head First Design Patterns: A Brain-Friendly Guide*, O'Reilly Media, Inc., 2004.
- http://www.chrislewiscasey.com/wp-content/uploads/2015/11/lotus-timelapse.jpg (Lotus image)