Tecniche di Progettazione: Design Patterns

GoF: Decorator

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Santa

Franka

SPYY

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.
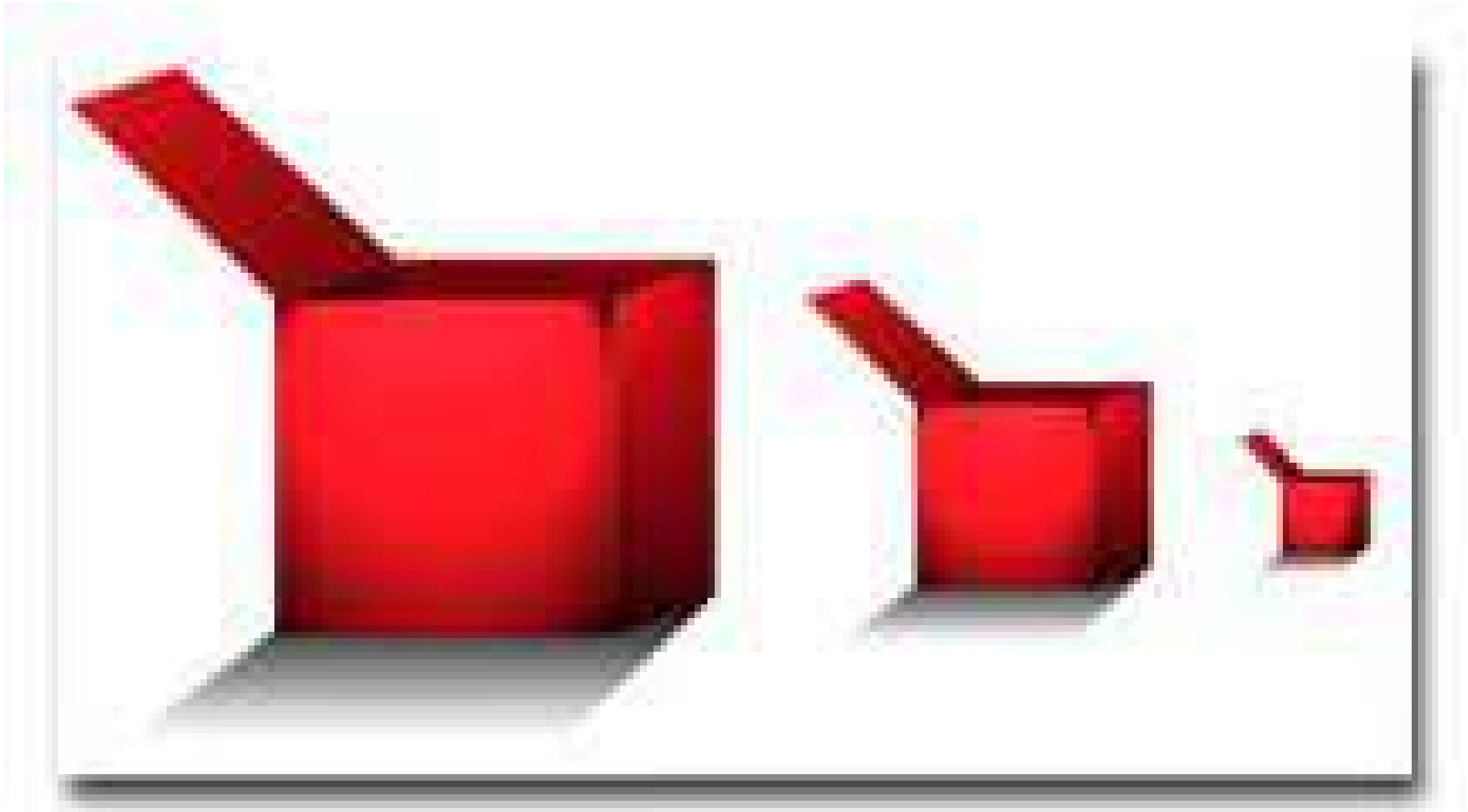
Decorator

- Intent
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
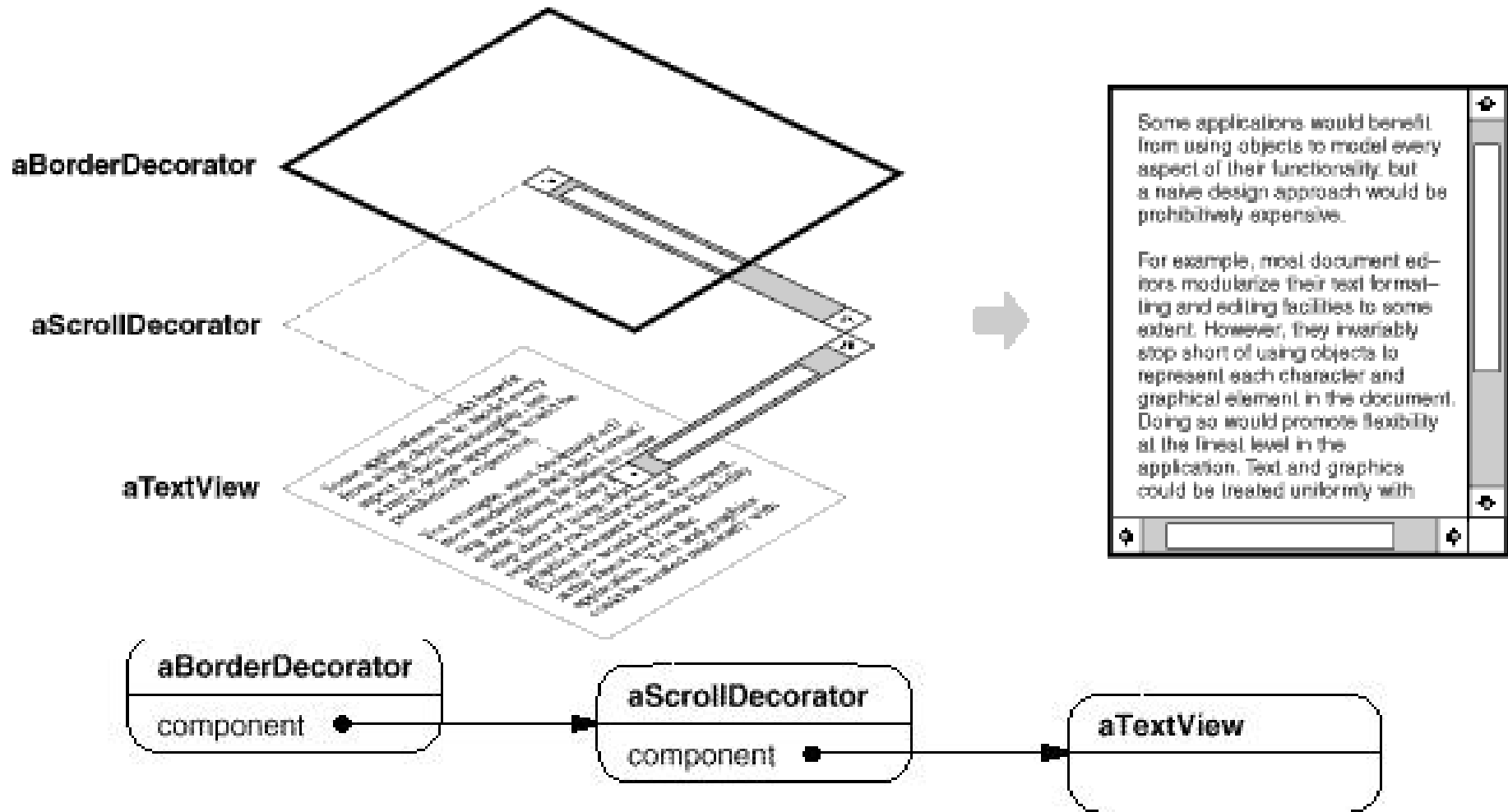
- Also Known As
  - Wrapper

- Motivation
  - We want to add properties, such as borders or scrollbars to a GUI component. We can do this with inheritance (subclassing), but this limits our flexibility. A better way is to use composition!
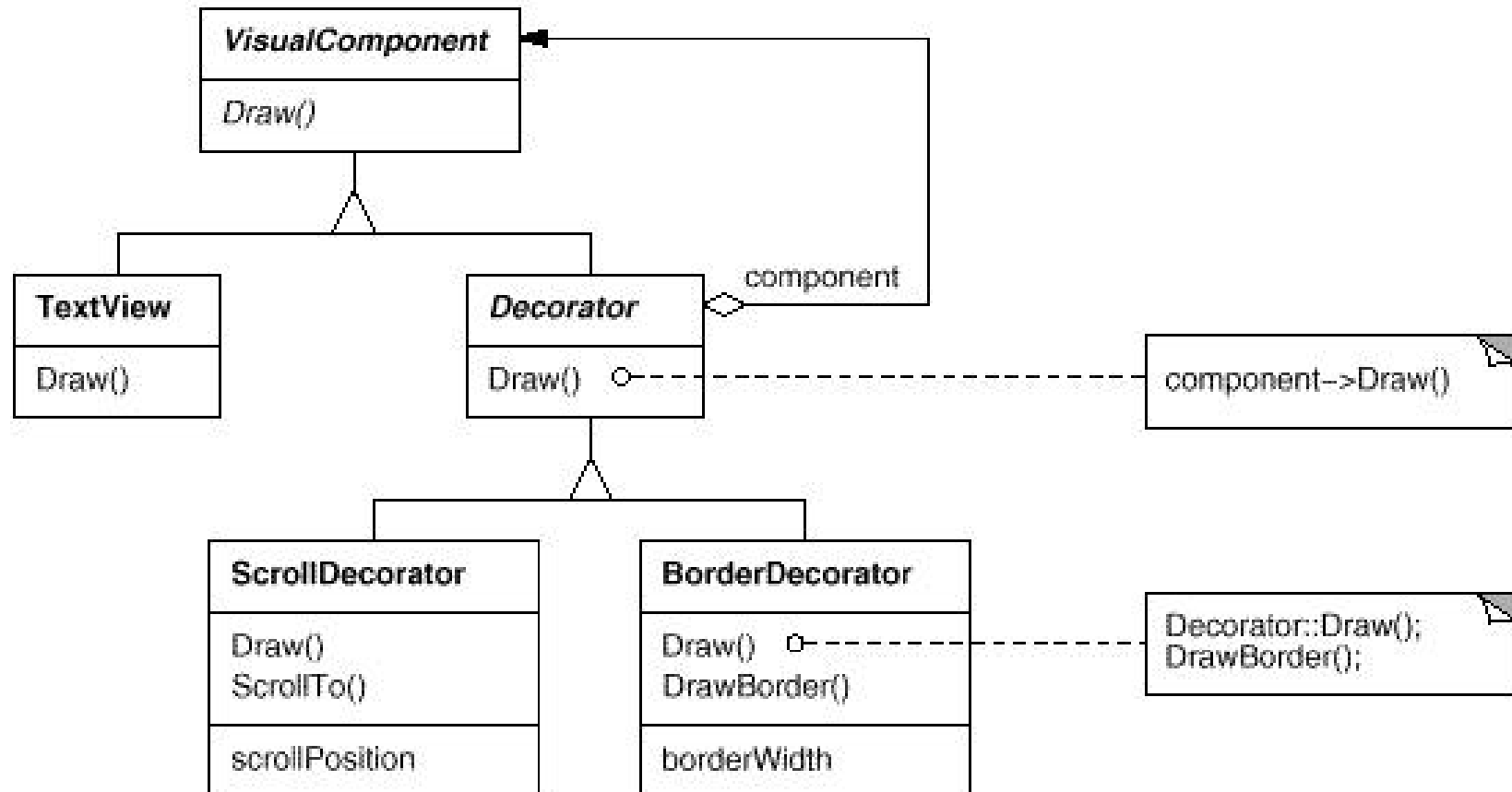
Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Motivation

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

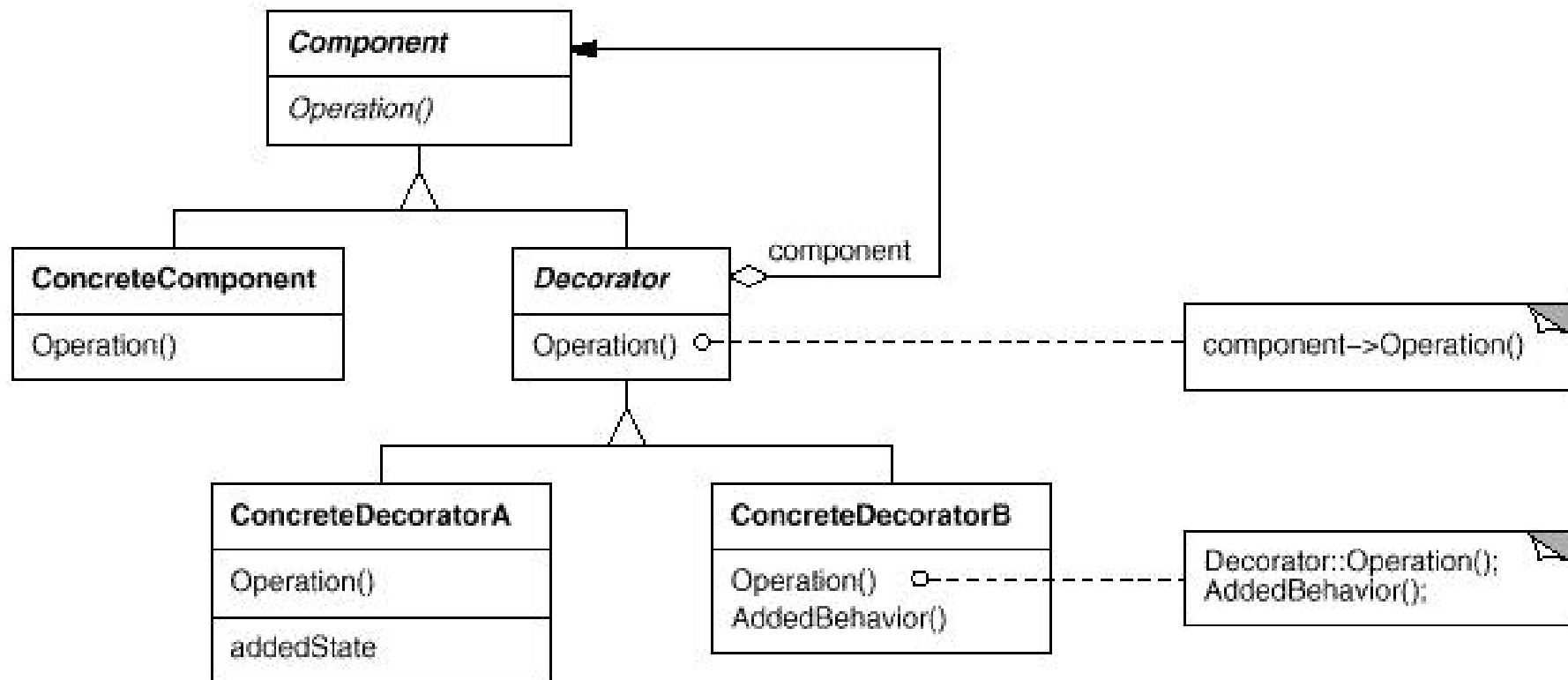# Decorator in Lexi

▶ Il problema

  ▶ Attaccare al glifo altri elementi, quali scrollbar e bordi

  ▶ Nel contempo si vogliono tenere questi elementi separati, visto che sono necessari o meno a seconda della situazione

▶ La soluzione: applicare Decorator

Structure: the TextView example

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Structure

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Decorator: partecipanti

‣ Component

   ‣ L'interfaccia comune degli oggetti da decorare

‣ ConcreteComponent

   ‣ La classe degli oggetti base che possono ricevere nuove responsabilità

‣ Decorator

   ‣ Definisce un'interfaccia conforme a quella comune e mantiene un riferimento a un solo oggetto Component (eventualmente già decorato)

‣ ConcreteDecorator

   ‣ Definisce una nuova responsabilità

Decorator: collaborazione

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.
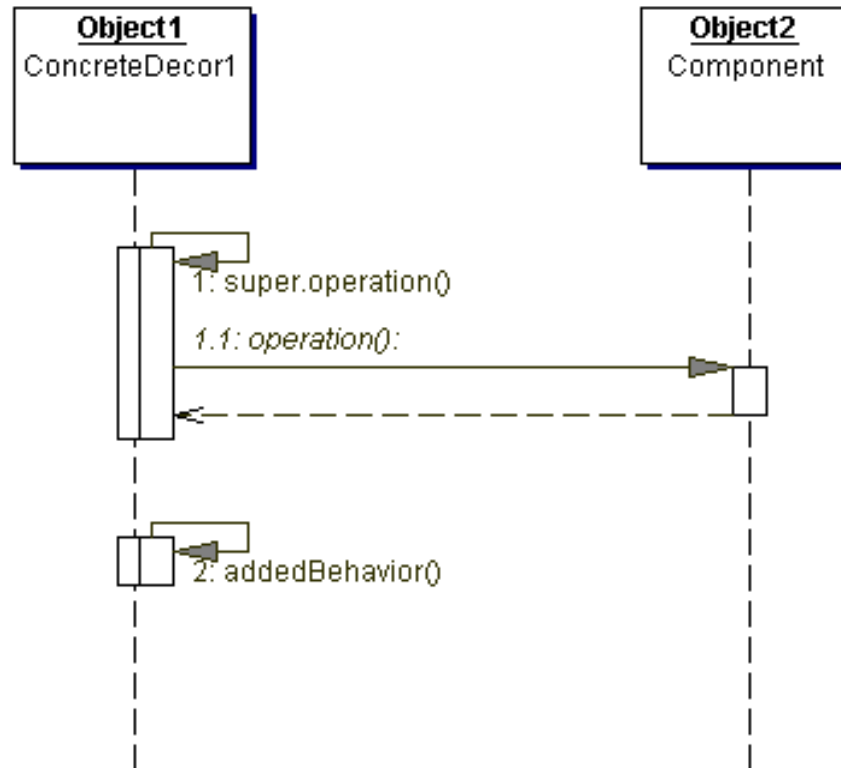
```
public class AdministrativeManager extends ResponsibleWorker {

    public AdministrativeManager( Employee empl ) {
        super( empl );
    }

    public void whoIs() {
        sayIamBoss();
        super.whoIs();
    }

    private void sayIamBoss(){
        System.out.print( "I am a boss. " );
    }

}
```

<- Extends

onsibleWorker

1

ne(): String
ce(): String

AdministrativeManger

+whoIs()
-sayIamBoss()

ProjectManger

project: String

+whoIs()

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

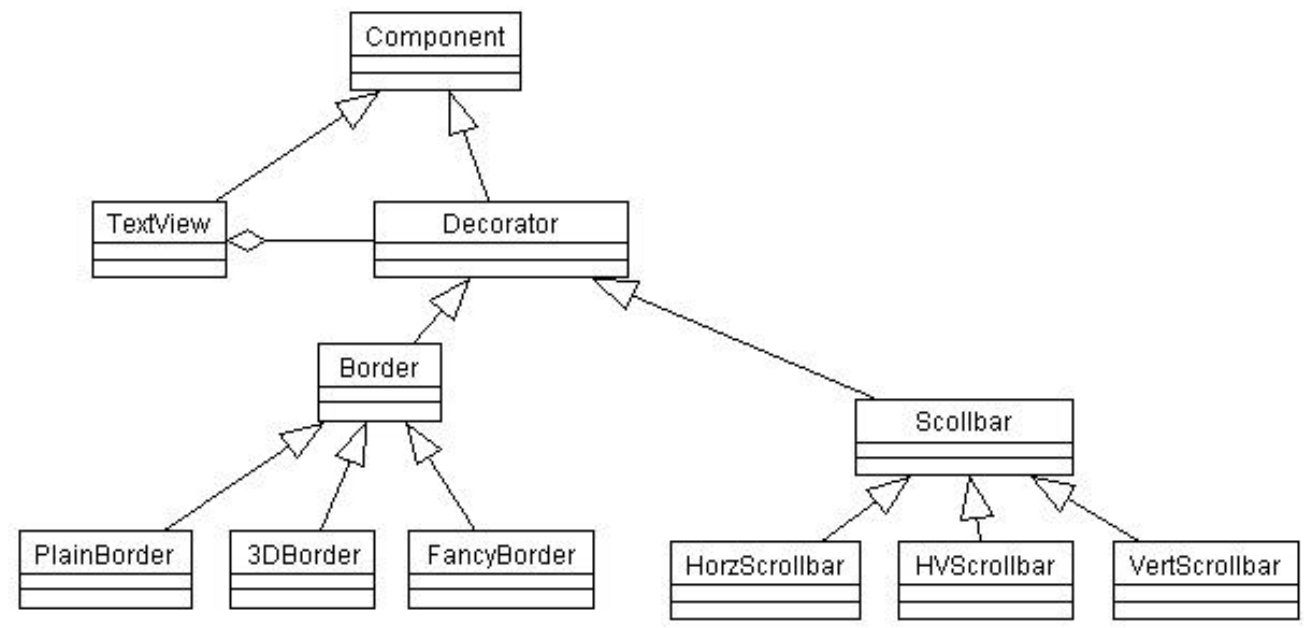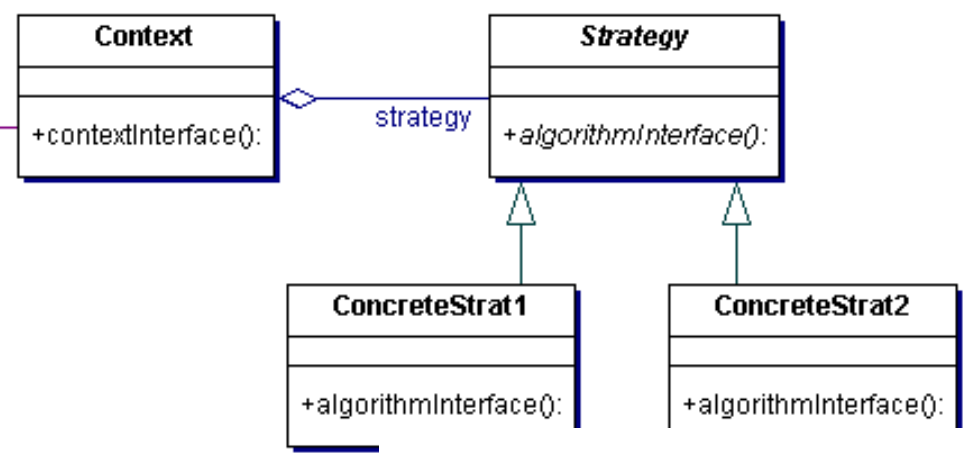# Motivation for the Decorator pattern in a little more detail.

- Suppose we have a TextView GUI component and we want to add different kinds of borders and scrollbars to it.

- Suppose we have three types of borders:

  - Plain, 3D, Fancy

- And two types of scrollbars:

  - Horizontal, Vertical

- Solution 1: Let's use inheritance first. We'll generate subclasses of TextView for all the required cases. We'll need the 15 subclasses:

TextView-Plain	TextView-Plain-Horizontal-Vertical
TextView-Fancy	TextView-3D-Horizontal
TextView-3D	TextView-3D-Vertical
TextView-Horizontal	TextView-3D-Horizontal-Vertical
TextView-Vertical	TextView-Fancy-Horizontal
TextView-Horizontal-Vertical	TextView-Fancy-Vertical
TextView-Plain-Horizontal	TextView-Fancy-Horizontal-Vertical
TextView-Plain-Vertical	

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Bad solution

- We already have an explosion of subclasses. What if we add another type of border? Or an entirely different property?

  - We have to instantiate a specific subclass to get the behavior we want.

- This choice is made statically and a client can't control how and when to decorate the component.

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

at some point , strategy.algorithmInterface()

Context
+contextInterface():

strategy

Strategy
+algorithmInterface():

ConcreteStrat1
+algorithmInterface():

ConcreteStrat2
+algorithmInterface():

Component

TextView

Decorator

Border

PlainBorder

3DBorder

FancyBorder

Scollbar

HorzScrollbar

HVScrollbar

VertScrollbar

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Using Strategy

▸ Now the TextView Class looks like this:

```
public class TextView extends Component {

    private Border border;

    private Scrollbar sb;

    public TextView(Border border, Scrollbar sb) {

            this.border = border;

            this.sb = sb;

    }

    public void draw() {

            border.draw();

            sb.draw();

            // Code to draw the TextView object itself.

} }
```

Using Strategy: pro and cons

▶ Pro:

  ▶ we can add or change properties to the TextView component dynamically. For example, we could have mutators for the border and sb attributes and we could change them at run-time.

▶ Cons:

  ▶ But note that the TextView object itself had to be modified and it has knowledge of borders and scrollbars! If we wanted to add another kind of property or behavior, we would have to again modify TextView.

Let's turn Strategy inside out to get the Decorator pattern

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.
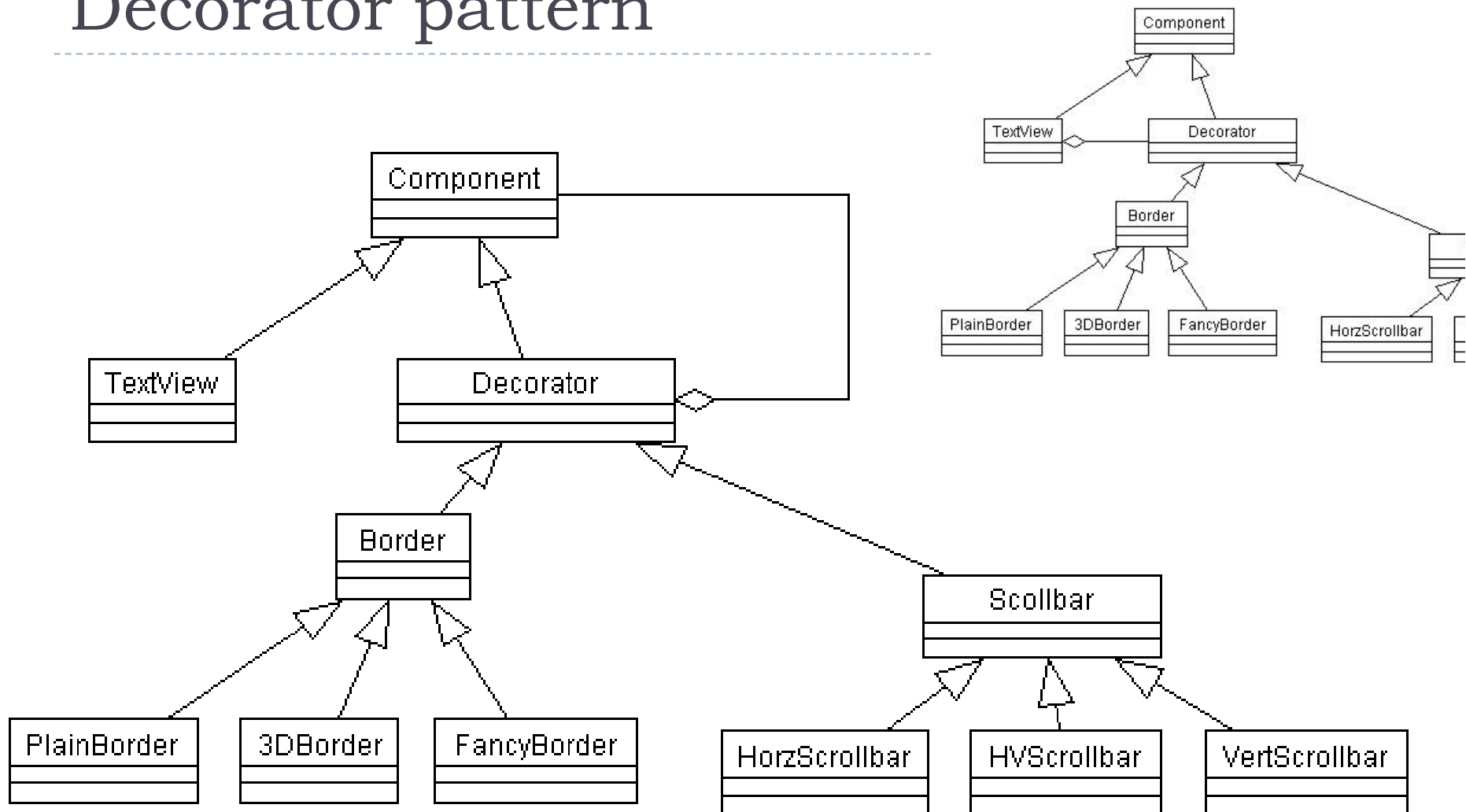
Implementing the Decorator solution

▸ Now the TextView class knows nothing about borders and scrollbars:

```
public class TextView extends Component {

    public void draw() {

    // Code to draw the TextView object itself.

    }

}
```

Implementing the Decorator solution (cont'd)

▶ But the decorators need to know about components:

```
public class FancyBorder extends Decorator {
        private Component component;
        public FancyBorder(Component component) {
                this.component = component;
        }
        public void draw() {
                component.draw();
                // Code to draw the FancyBorder object itself.
        }
}
```

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Implementing the Decorator solution (cont'd)

▸ Now a client can add borders as follows:

```
public class Client {
        public static void main(String[] args) {
                TextView data = new TextView();
                Component borderData = new FancyBorder(data);
                Component scrolledData = new VertScrollbar(borderData);
                Component borderAndScrolledData = new
                        HorzScrollbar(scrolledData);
        }
}
```

▸ Decorator: Changing the skin of an object

▸ Strategy: Changing the guts (viscere) of an object

Actor
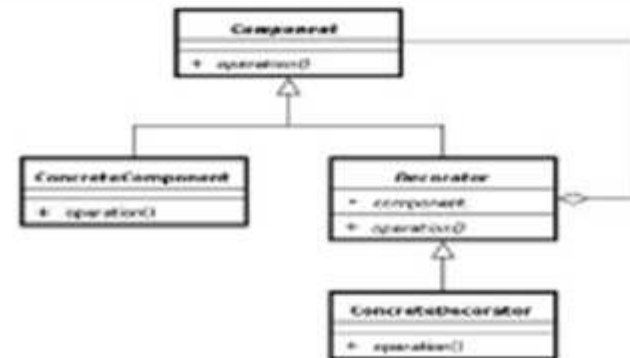
```
class Stage{
Actor forAct1 = new Santa(new Actor)
Actor forAct2 = new Spy(new Frank(new Actor)
}
```

Act1 is a Santa act.
Act2 the actor is a Frankenstein who Disguises as a Spy.

STAGE

Santa

Franka

SPY

Concrete Decorators

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.
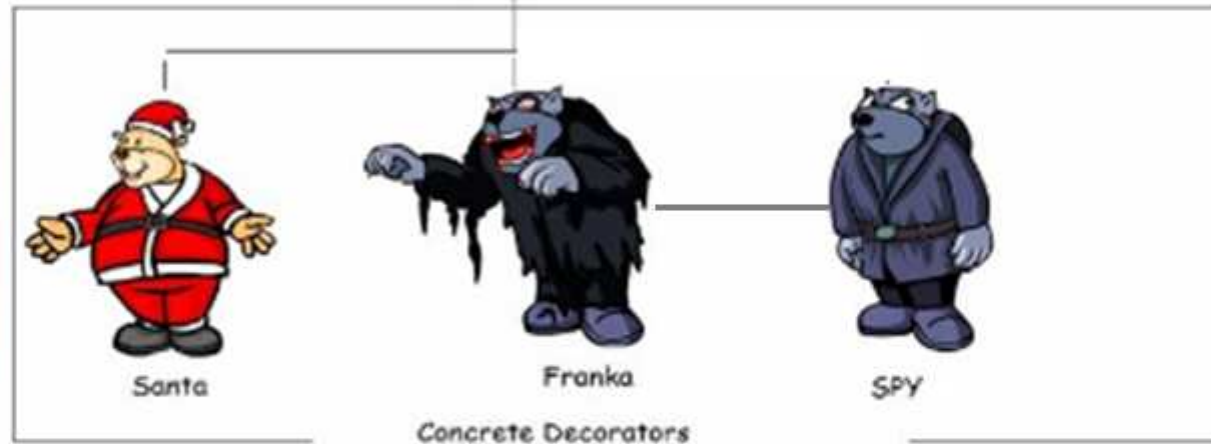
An example

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...

Your first idea of implementation

Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.
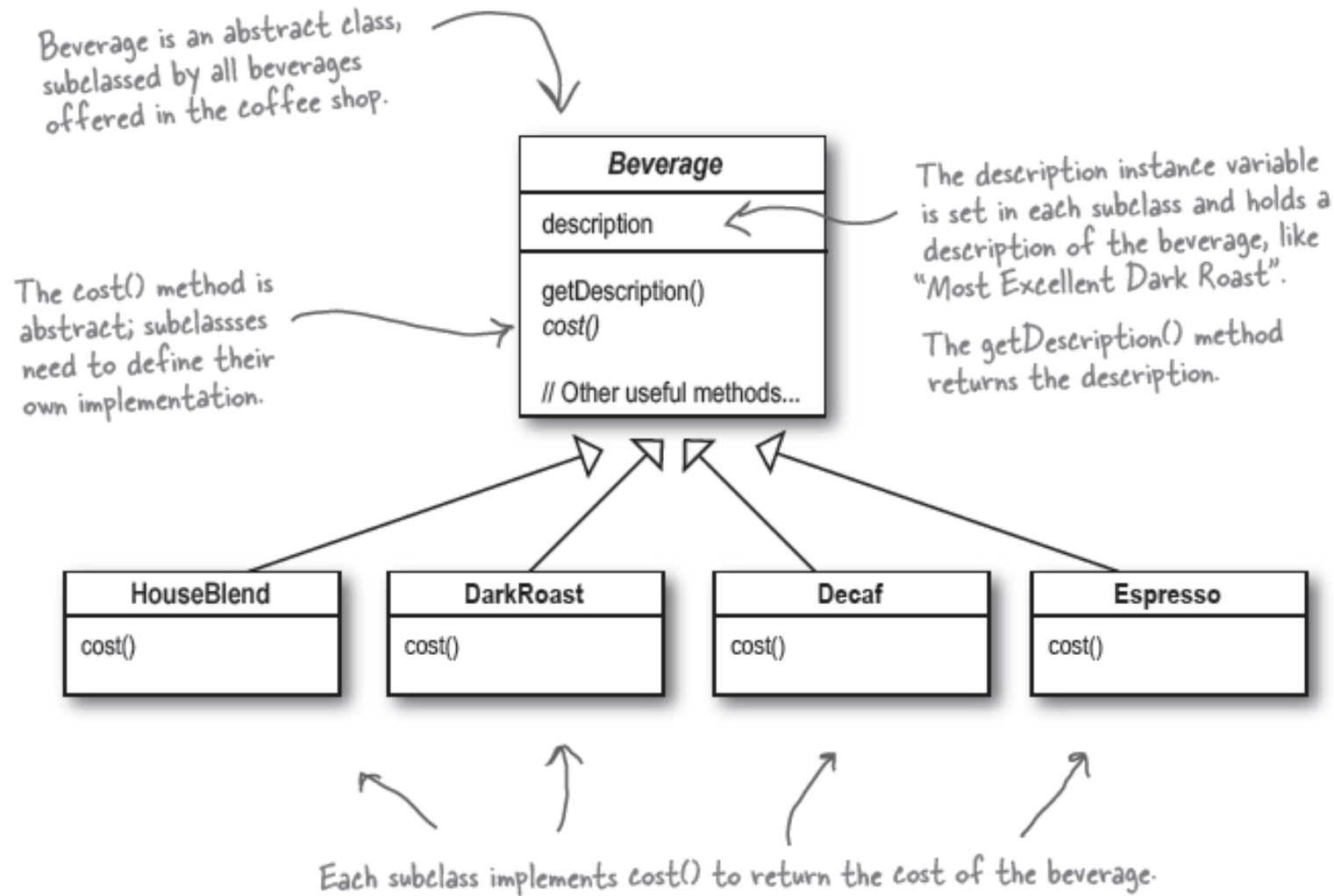
The cost() method is abstract; subclassses need to define their own implementation.

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.

HouseBlend	**DarkRoast**	**Decaf**	**Espresso**
cost()	cost()	cost()	cost()

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

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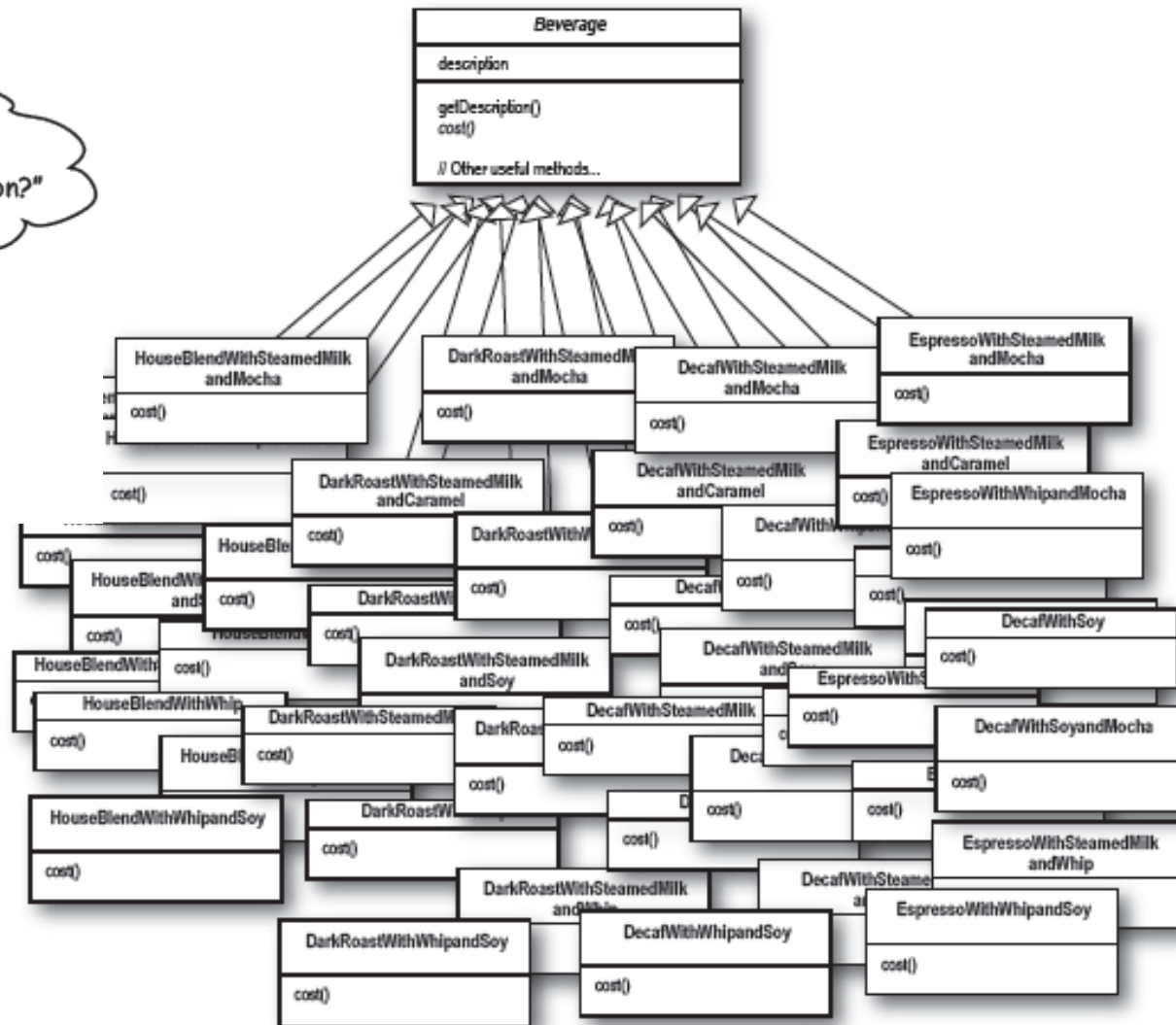
In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they really need to get them built into their order system.

Here's their first attempt...

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Now a beverage can be mixed from different condiment to form a new beverage

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

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.

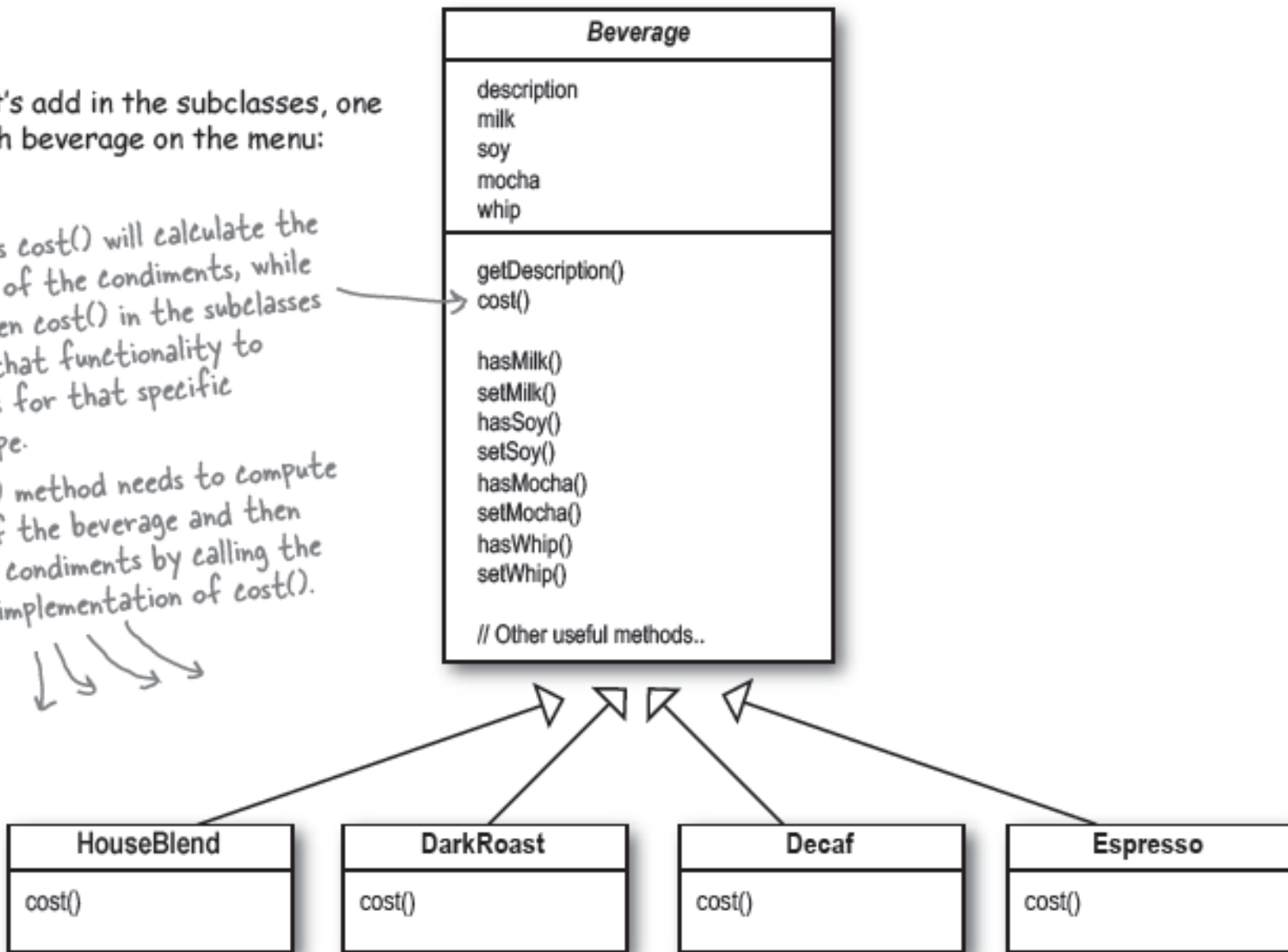Now we'll implement cost() in Beverage (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override cost(), but they will also invoke the super version so that they can calculate the total cost of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

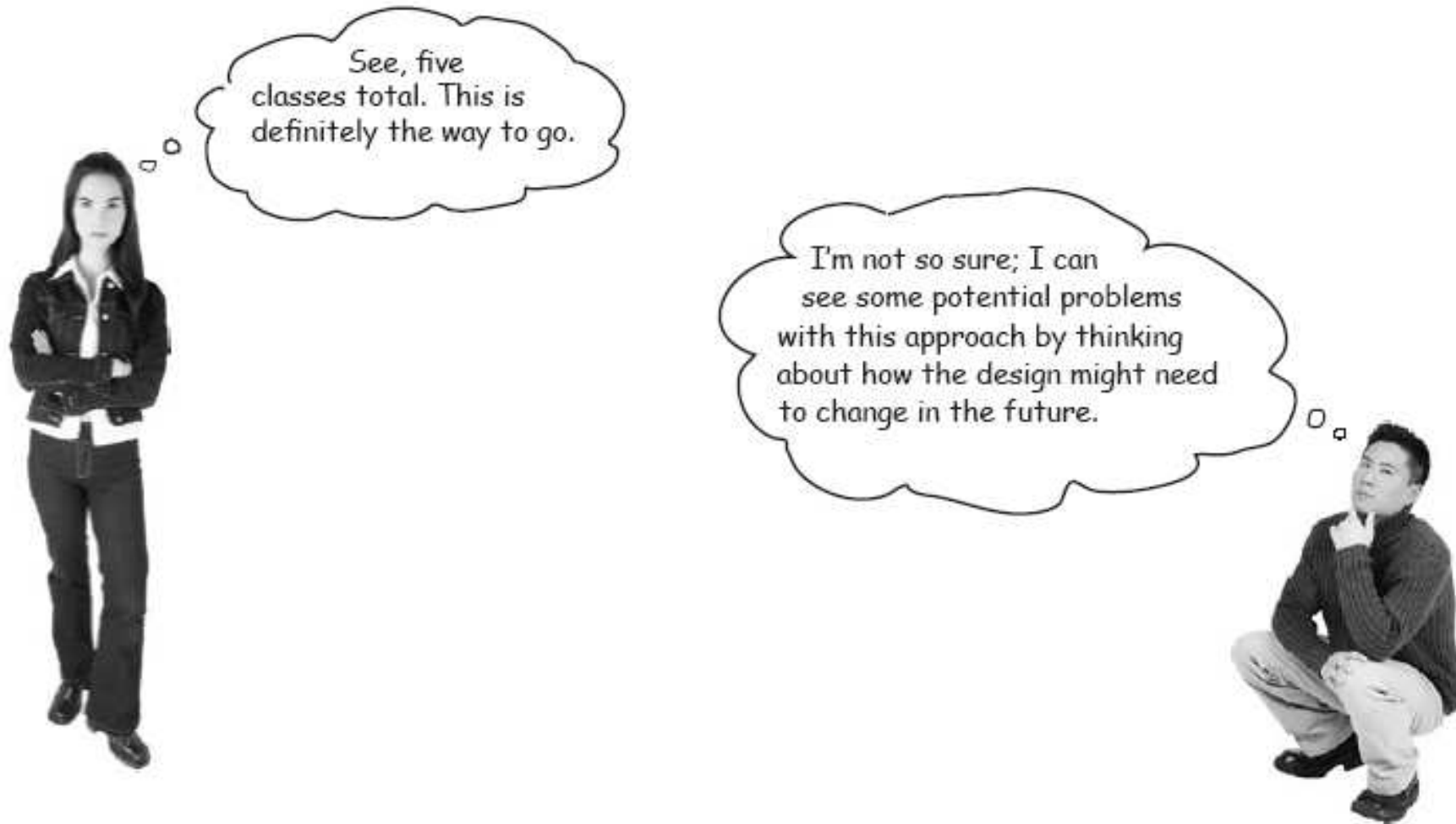These get and set the boolean values for the condiments.

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().

Beverage

description
milk
soy
mocha
whip

getDescription()
cost()

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

// Other useful methods..

HouseBlend

cost()

DarkRoast

cost()

Decaf

cost()

Espresso

cost()

Now, your turns. It is a good solution?

Thought bubble 1: See, five classes total. This is definitely the way to go.

Thought bubble 2: 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.

Try several minutes to complete

Sharpen your pencil

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

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



    }
}
```

```java
public class DarkRoast extends Beverage {

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

    public double cost() {



    }
}
```

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

What can you criticize about this inheritance architecture?

▸ Write down your notes to see if you are right

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:

Design Principle

Classes should be open for extension, but closed for modification.

▸ **SOLID 2: Open Closed Principle :**

  ▸ Extending a class shouldn't require modification of that class.

  ▸ Software entities like classes, modules and functions should be open for extension but closed for modifications.

    ▸ OPC is a generic principle. You can consider it when writing your classes to make sure that when you need to extend their behavior you don't have to change the class but to extend it. The same principle can be applied for modules, packages, libraries.

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.

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

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.

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Decorator Pattern

▸ **The problems of two previous designs**

  ▸ we get class explosions, rigid designs,

  ▸ or we add functionality to the base class that isn't appropriate for some of the subclasses.

Revisit the problem again

▶ **If a customer wants a Dark Roast with Mocha and Whip**

  ▶ 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
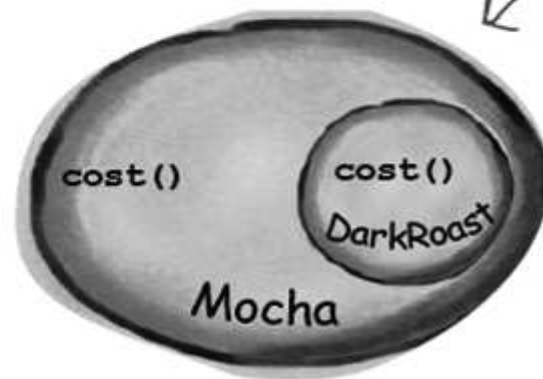
Constructing a drink order with Decorators

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.**

cost()   cost()

DarkRoast

Mocha

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).

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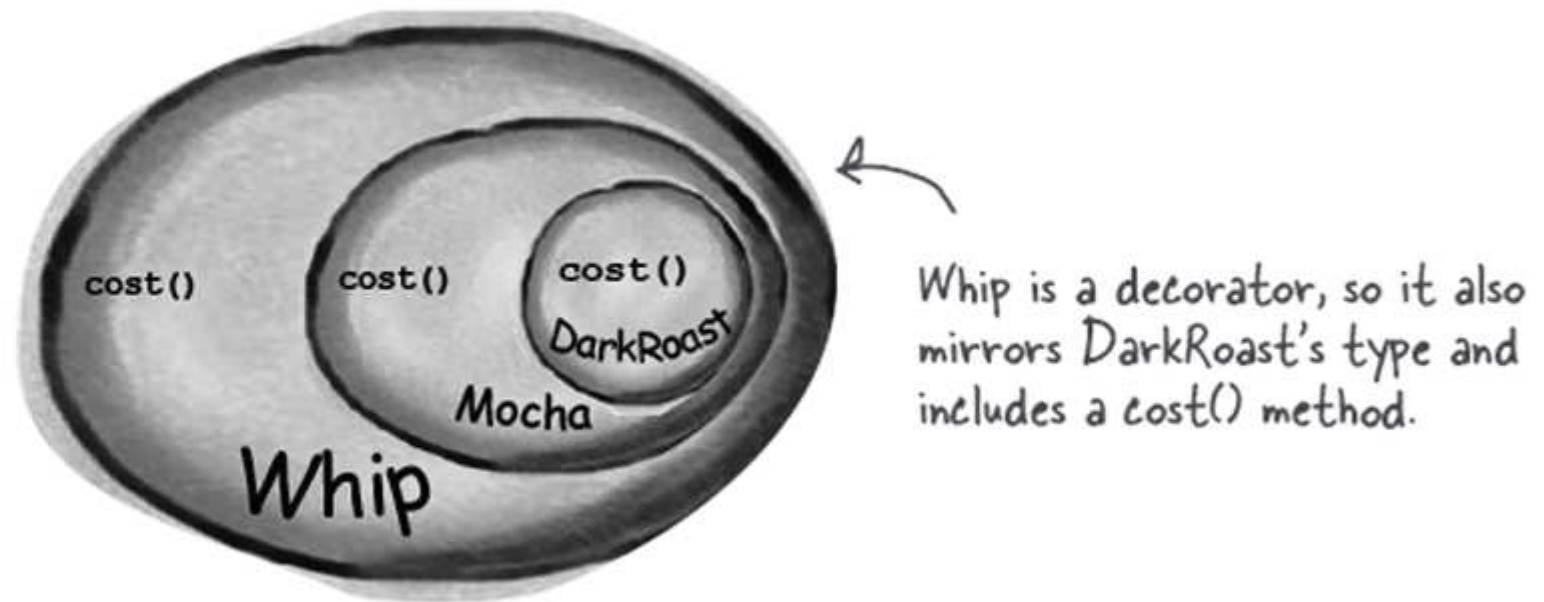
So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

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.

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

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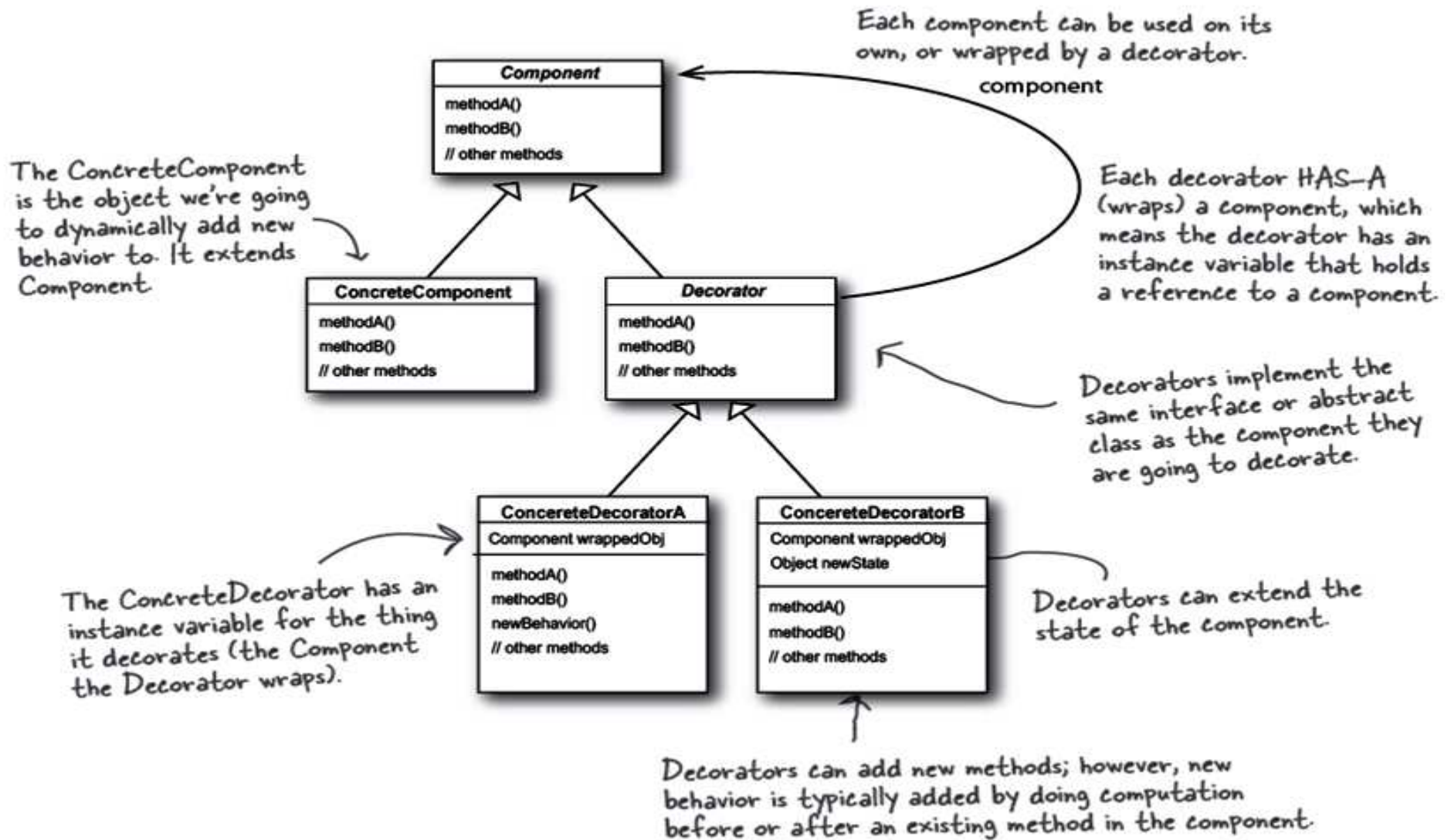
1 First, we call cost() on the outmost decorator, Whip.

3 Mocha calls cost() on DarkRoast.

$1.29 ← .10 cost() .20 cost() .99 cost() DarkRoast

Mocha

Whip

4 DarkRoast returns its cost, 99 cents.

5 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.

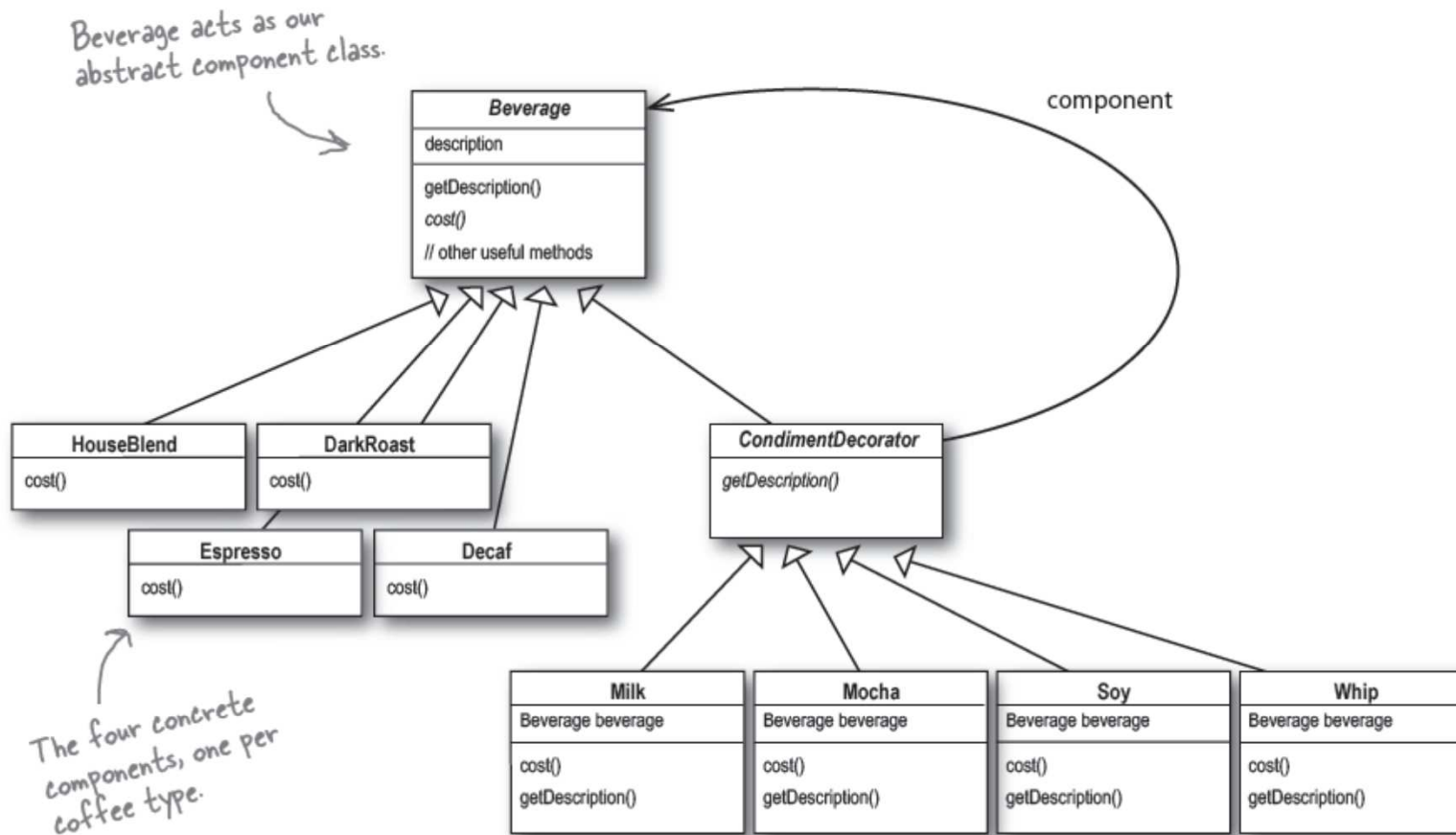Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
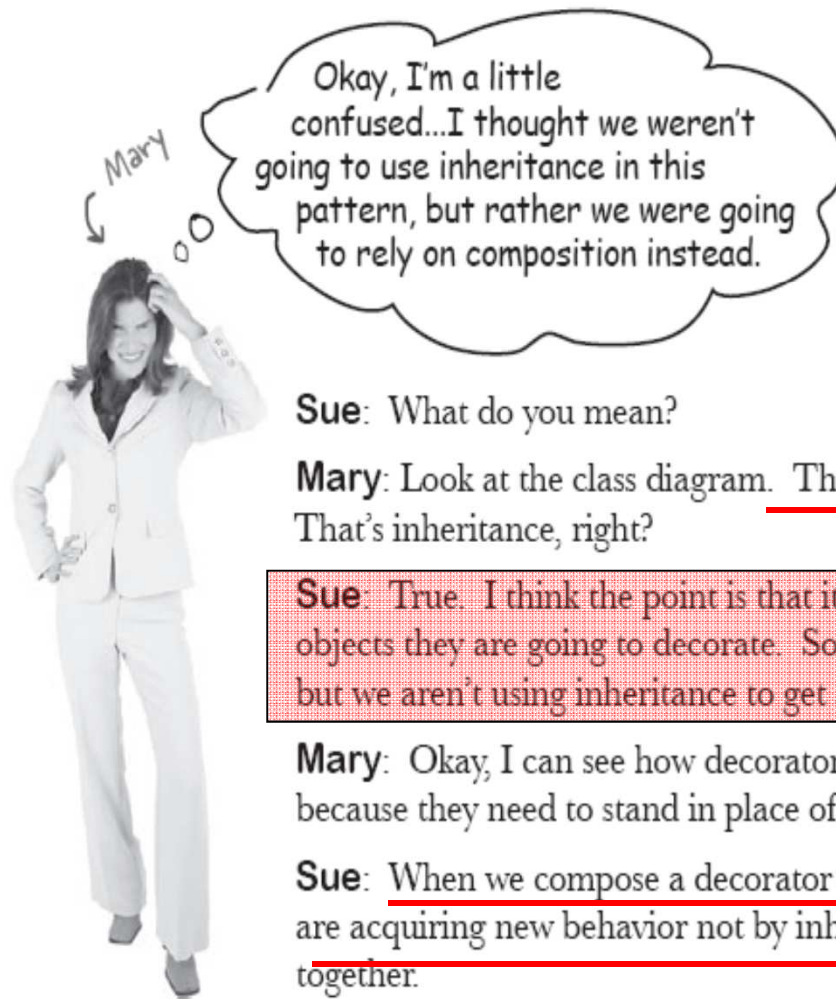
Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Decorator Pattern defined

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.

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

The decorator pattern for Starbuzz beverages

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Some confusion over Inheritance versus Composition

Mary (thought): Okay, I'm a little confused...I thought we weren't going to use inheritance in this pattern, but rather we were going to rely on composition instead.

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.

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

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.

Let's see 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.

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

The abstract class of condiments

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

```
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...

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Concrete Base Classes of Beverages

```
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
```

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.

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

A concrete Condiment class

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();
    }
}
```

When Mocha price changed, we only need to change this

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.

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.

Constructing new beverages from decorator classes dynamically

```java
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());
    }
}
```
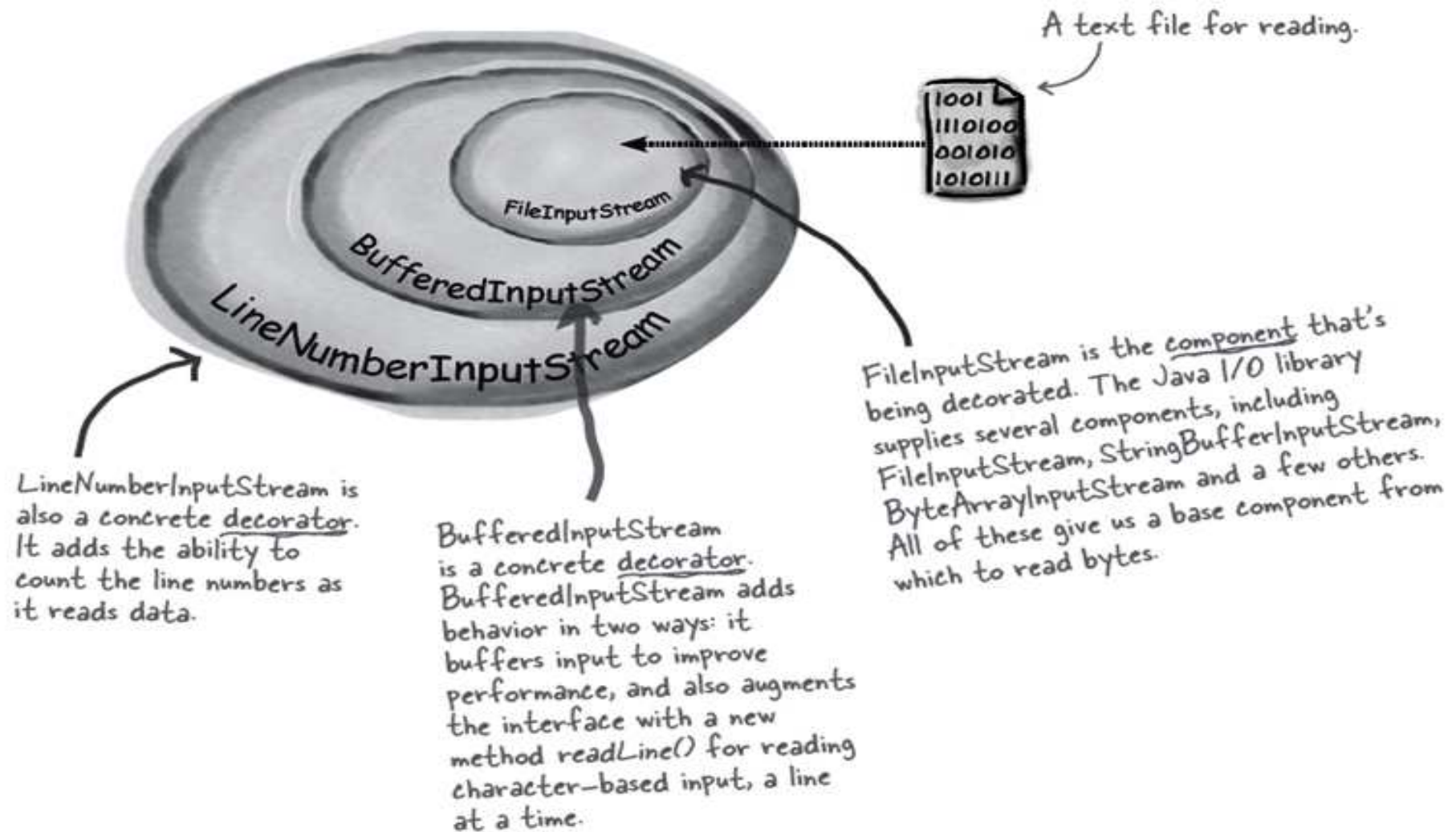
Order up an espresso, no condiments and print its description and cost.

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.

Real world decorator – Java I/O

A text file for reading.

FileInputStream

BufferedInputStream

LineNumberInputStream

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.

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.

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Comments

▸ 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 decorator s 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).

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Let's write a new decorator

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

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

```java
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.

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Test out your new Java I/O decorator

```java
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();
        }
    }
}
```
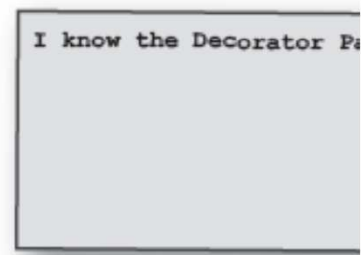
Set up th·
and decor
a Buffer
and then
LowerC·

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

I know the Decorator P·

test.

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Dark Side

▸ You can usually insert decorators transparently and the client never has to know it's dealing with a decorator

▸ However, if you write some code is dependent on specific types -> Bad things happen

▸ Java library is notorious to be used badly by people who do not know decorator pattern

```
Beverage beverage2 = new DarkRoast();
beverage2 = new Mocha(beverage2);
beverage2 = new Mocha(beverage2);
beverage2 = new Whip(beverage2);
System.out.println(beverage2.getDescription()
+ " $" + beverage2.cost());
```
The right way

```
Beverage beverage2 = new DarkRoast();
beverage2 = new Mocha(beverage2);
beverage2 = new Mocha(beverage2);
Whip beverage3 = new Whip(beverage2);
System.out.println(beverage3.getDescription()
+ " $" + beverage2.cost());
```
The poor way

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();

    }
}
```

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.

Decorator: Consequences

▸ **Good**

  ▸ More Flexibility than static inheritance

    ▸ Much easier to use than multiple inheritance

    ▸ Can be used to mix and match features

    ▸ Can add the same property twice

    ▸ Allows to easily add new features incrementally

Decorator: Consequences

- **Bad**
  - If Decorator is complex, it becomes costly to use in quantity
  - A decorator and its component aren't identical
    - From an object identity point of view, a decorated component is not identical to the component itself
    - Don't rely on object identity when using decorators
  - Lots of little objects
    - Often end up with systems composed of lots of little objects
    - Can be hard to learn and debug

Implementation Issues

▶ **Several issues should be considered when applying the Decorator pattern:**

1. **Interface conformance:**

   A decorator object's interface must conform to the interface of the component it decorates.

2. **Omitting the abstract Decorator class:**

   If only one responsibility is needed, don't define abstract Decorator. Merge Decorator's responsibility into the ConcreteDecorator.

Implementation Issues

3. Keeping Component classes light weight:

   The Component class is inherited by components and decorators. Component class should be dedicated to defining an interface, no other functions. E.g. The Component class should not be used for storing data, and defining data. That should be done in subclasses. If the Component class becomes complex, it might make the decorators too heavyweight to use in quantities. Keep it light and simple. A complex Component class might make Decorator too costly to use in quantity.

4. Changing the skin of an object versus its guts:

   Decorator classes should act as a layer of skin over an object. If there's a need to change the object's guts, use Strategy pattern.

References

- [GoF95]: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley, Reading Mass. 1995.

- [AGCS]: AG Communication Systems, http://www.agcs.com/

- SDSU & Roger Whitney (2001) CS 635 Advanced Object-Oriented Design & Programming, Decorator, Chain of Responsibility, OO Recursion, San Diego State University, http://www.eli.sdsu.edu/courses/spring01/cs635/notes/decorator/decorator.html

- Bob Tarr, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, http://www.research.umbc.edu/~tarr/dp/lectures/Decorator-2pp.pdf