Tecniche di Progettazione: Design Patterns

GoF: Decorator

An example

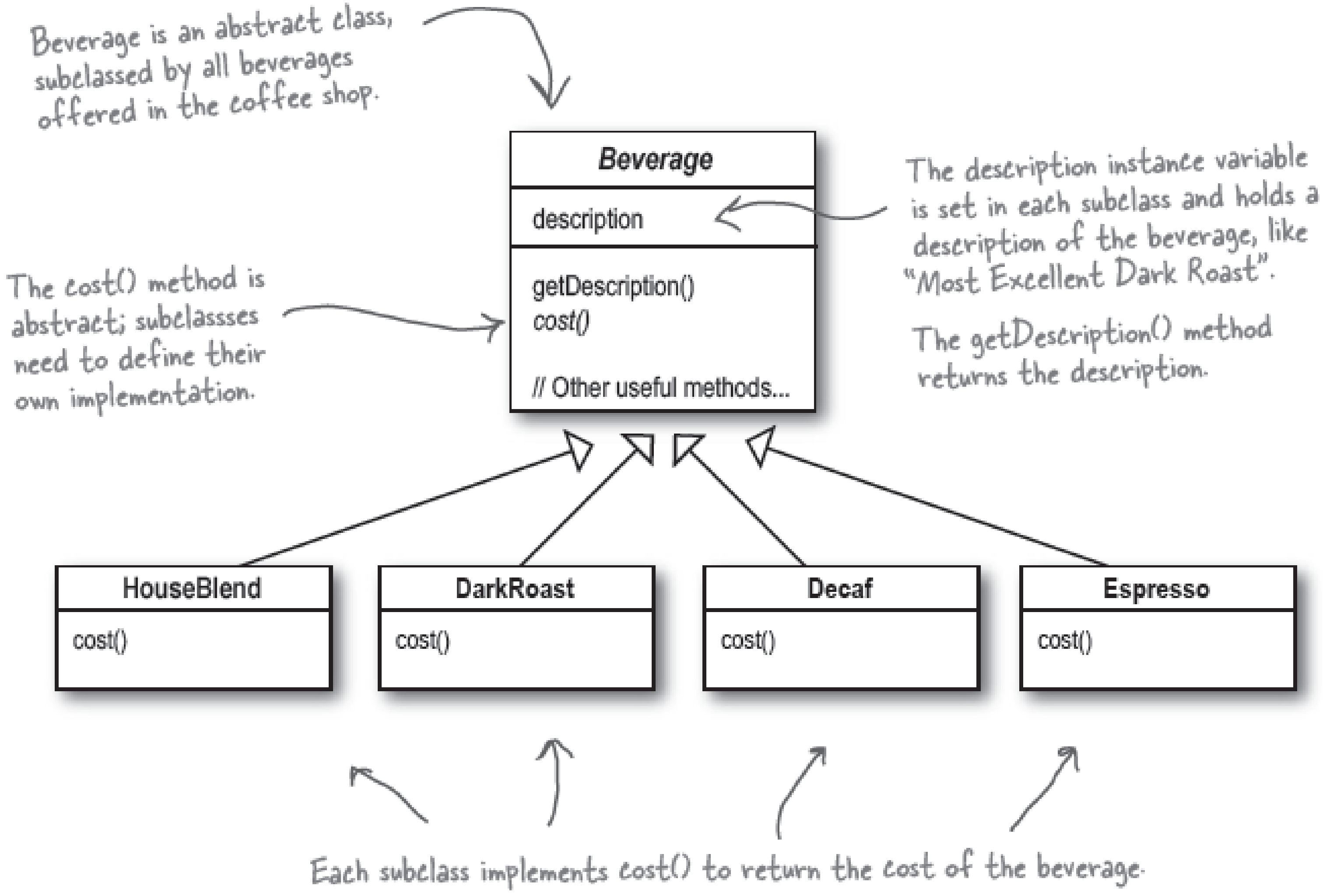
Welcome to Starbuzz Coffee

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

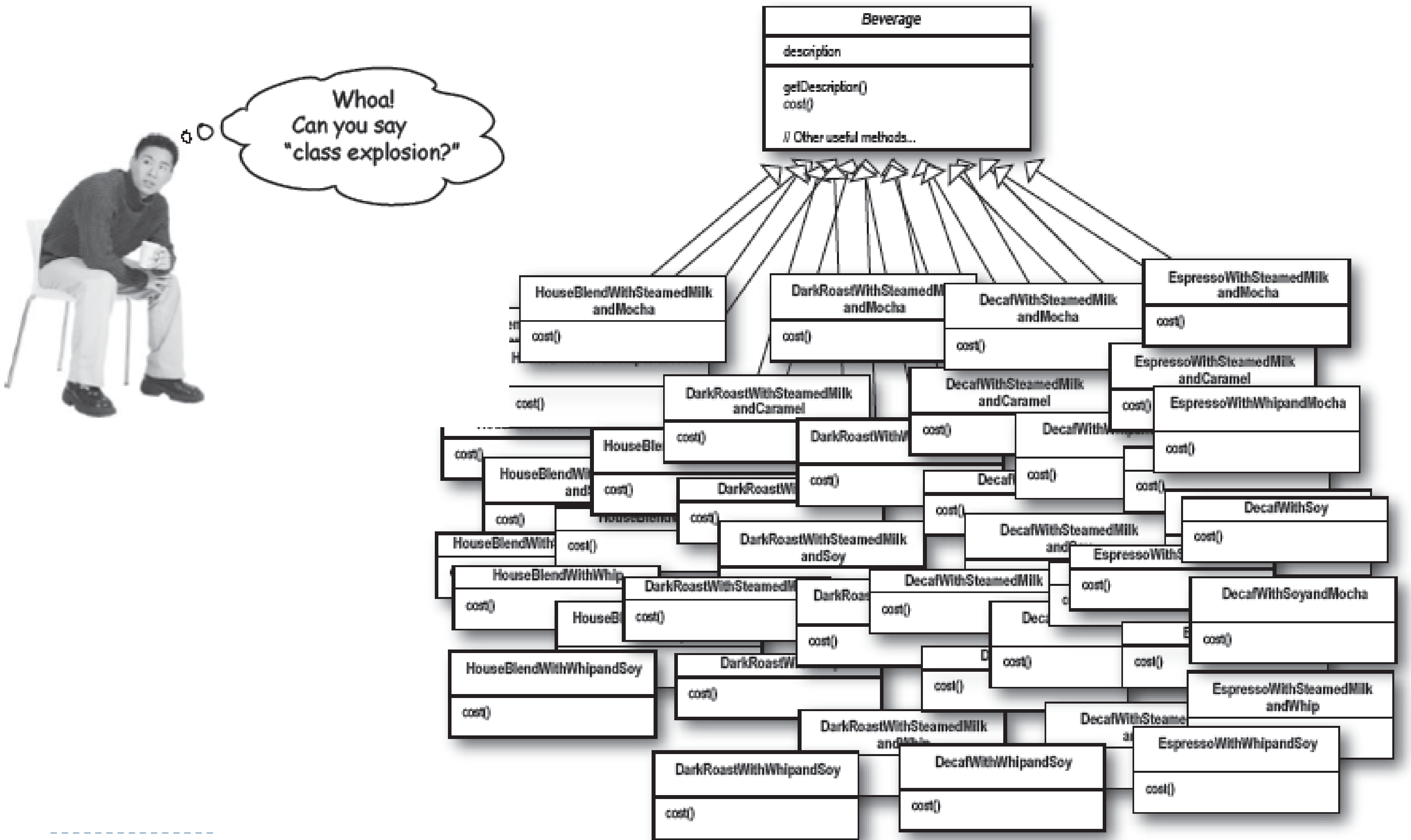


In reality

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...

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



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...

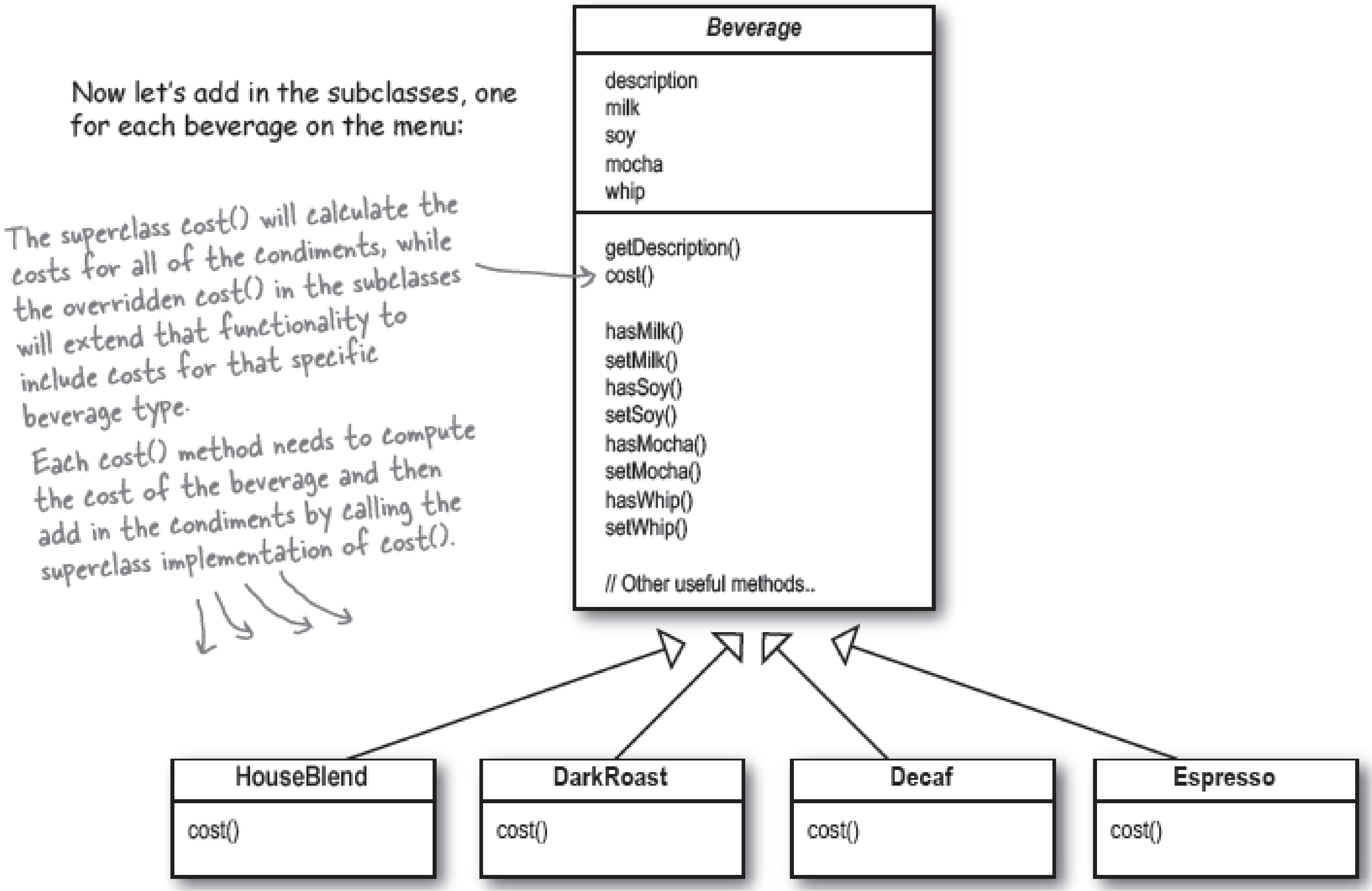
Beverage
description milk soy mocha whip
getDescription() cost() hasMilk() setMilk() hasSoy() setSoy() hasMocha() setMocha() hasWhip() setWhip() // Other useful methods..

New boolean values for each condiment.

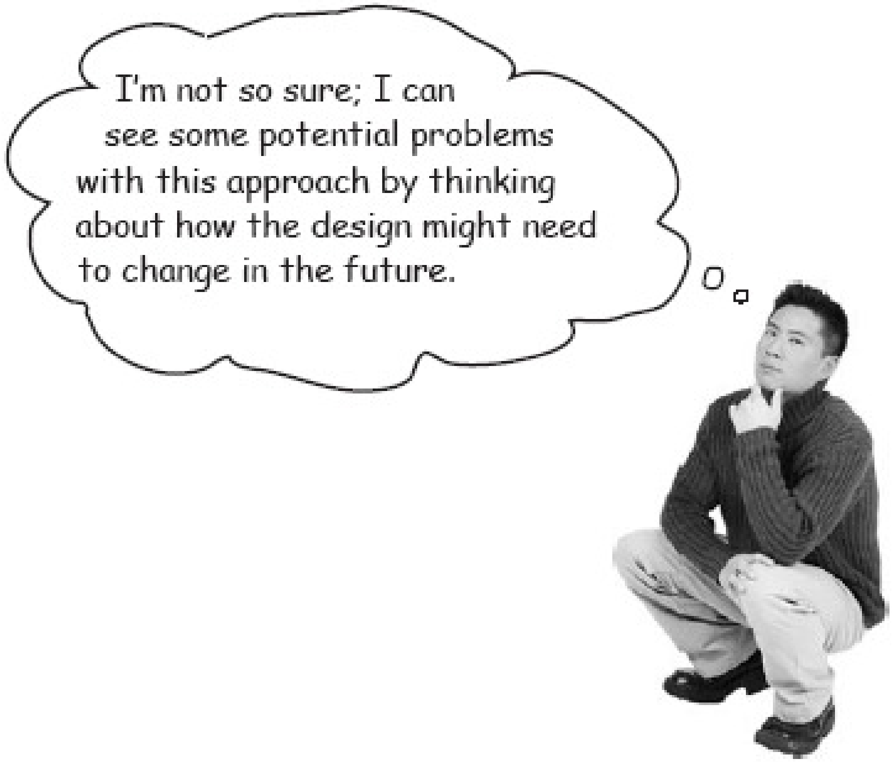
Now we'll implement cost() in Beverage (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override cost(), but they will also invoke the super version so that they can calculate the total cost of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



Now, your turns. It is a good solution?



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.

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.

Decorator Pattern

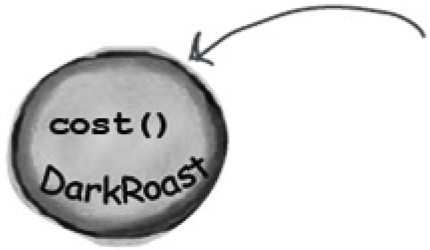
- The problems of two previous designs
  - we get class explosions, rigid designs,
  - or we add functionality to the base class that isn't appropriate for some of the subclasses.

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

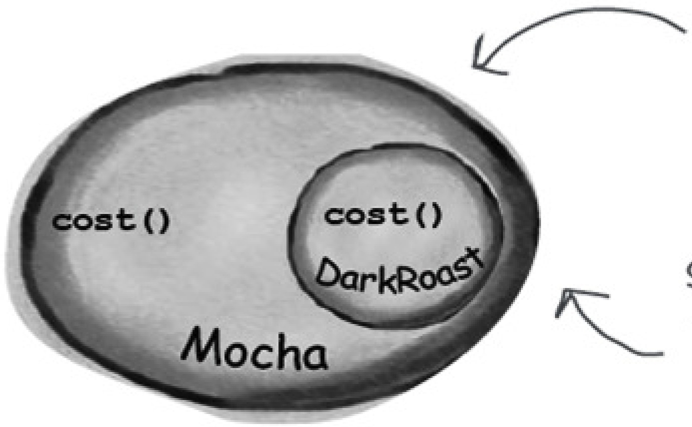
❶ We start with our DarkRoast object.



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

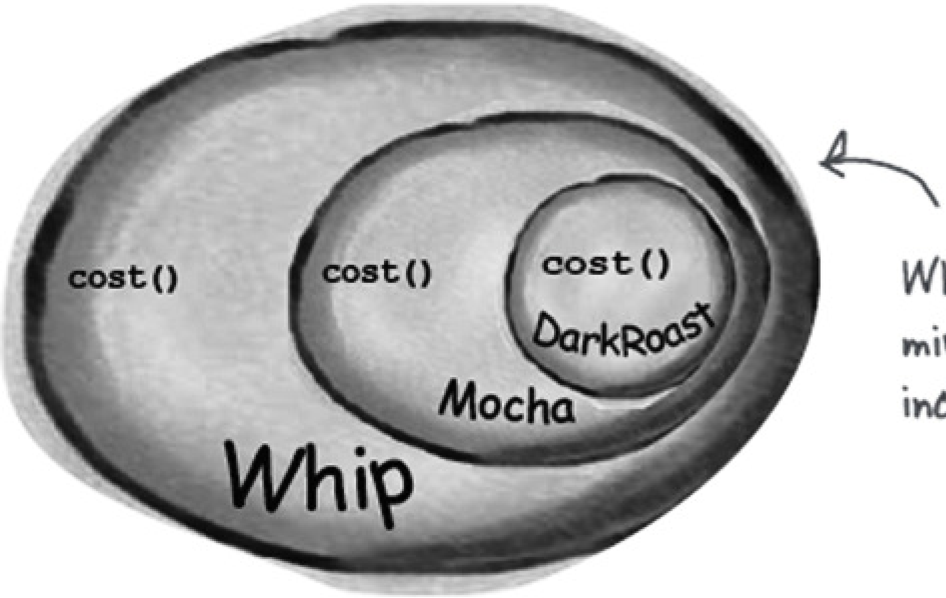
❷ The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.

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).

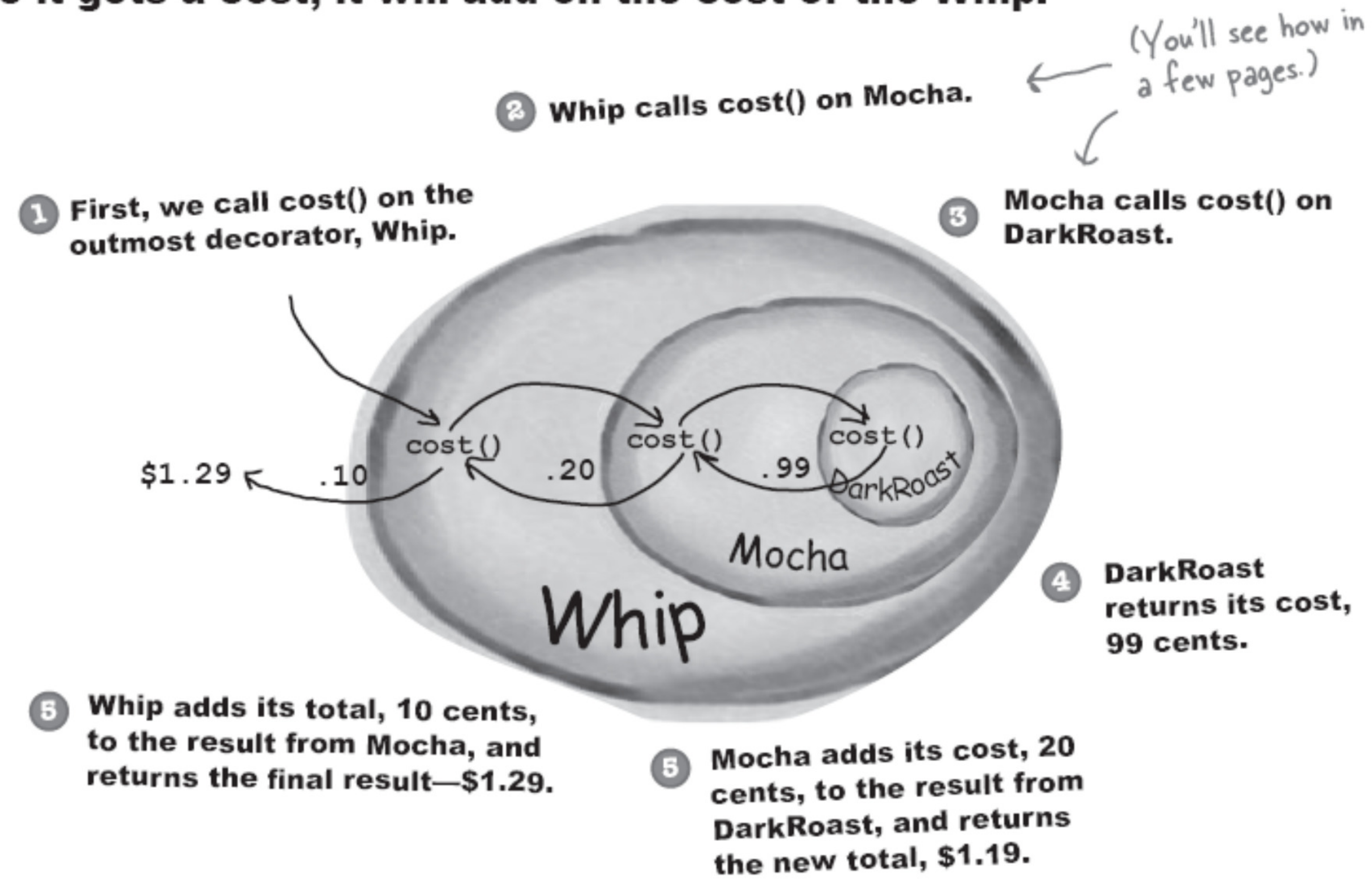
The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.



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

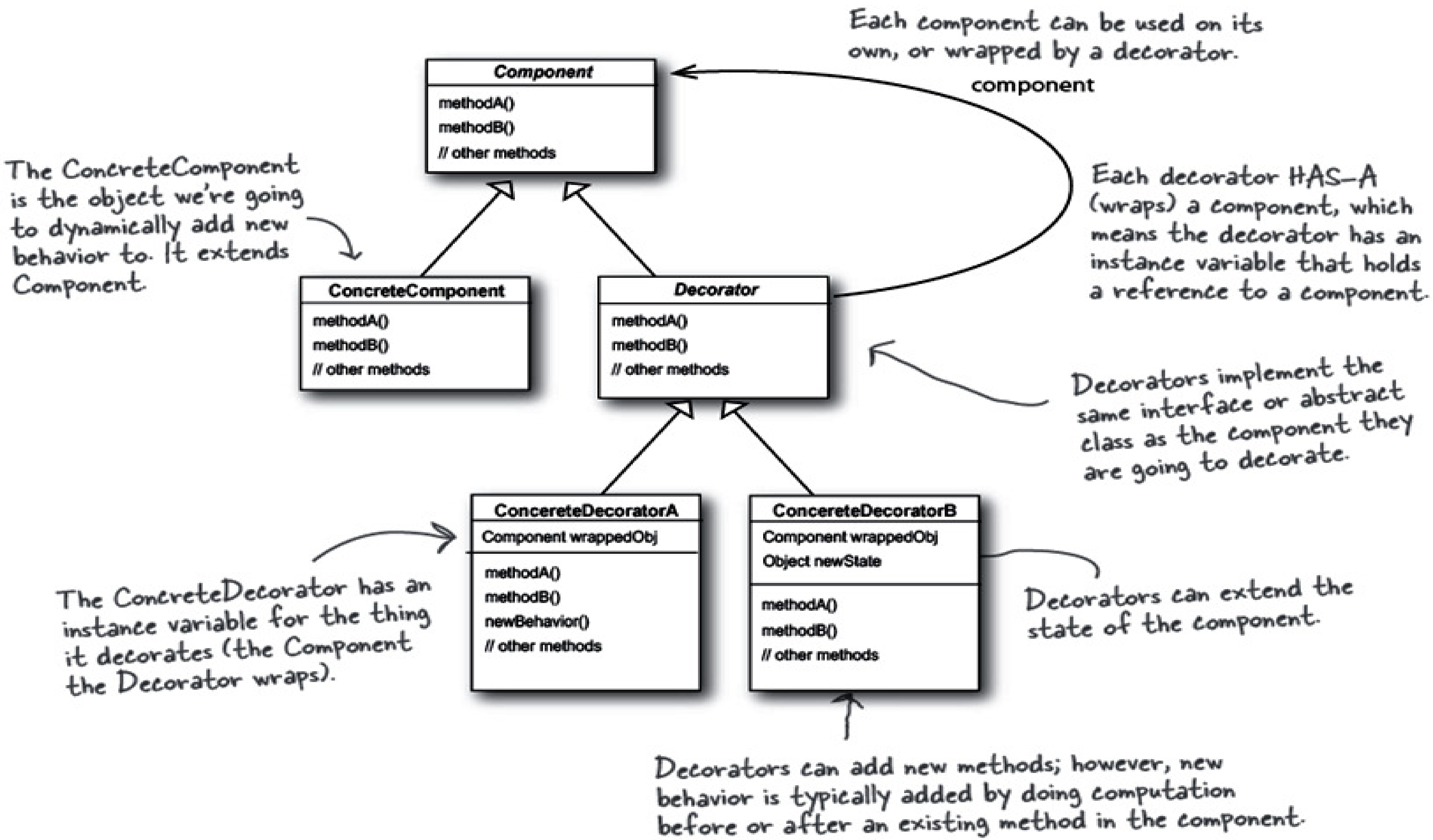
So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

❹ 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.

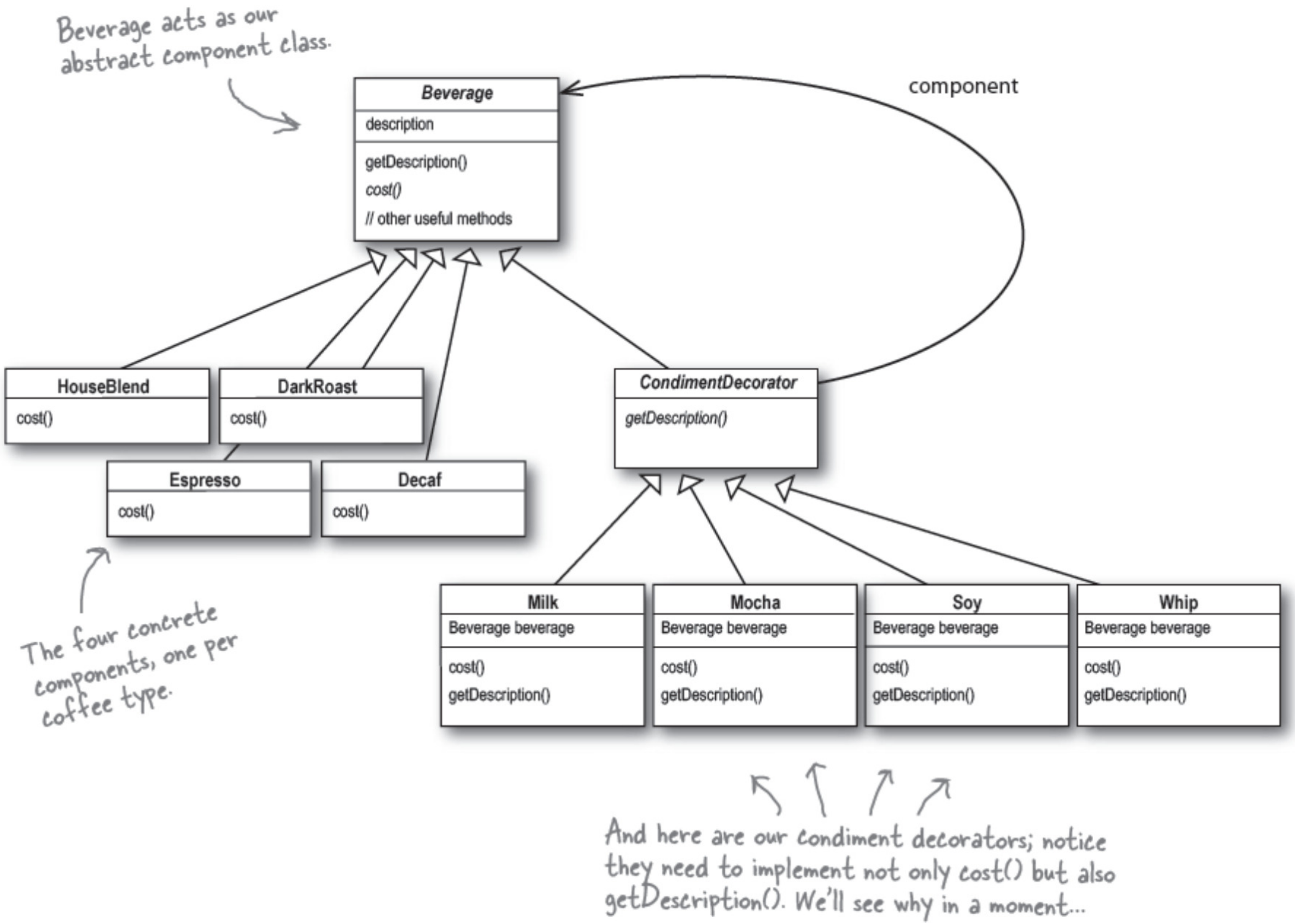


The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

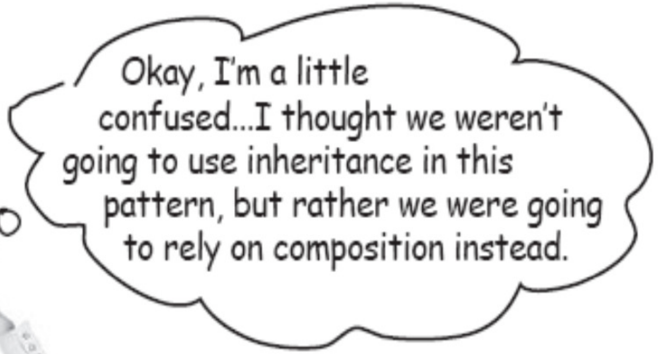
Decorator Pattern defined



The decorator pattern for Starbuzz beverages



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.

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().

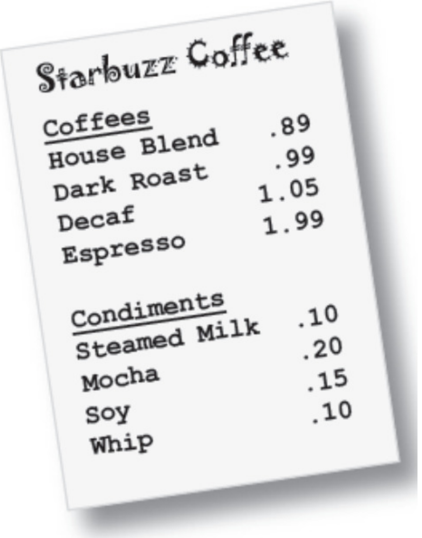
getDescription is already implemented for us, but we need to implement cost() in the subclasses.

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...



Concrete Base Classes of Beverages

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.

A concrete Condiment class

Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

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

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

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

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

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

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

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

When Mocha price changed, we only need to change this

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.

Constructing new beverages from decorator classes dynamically

```
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.

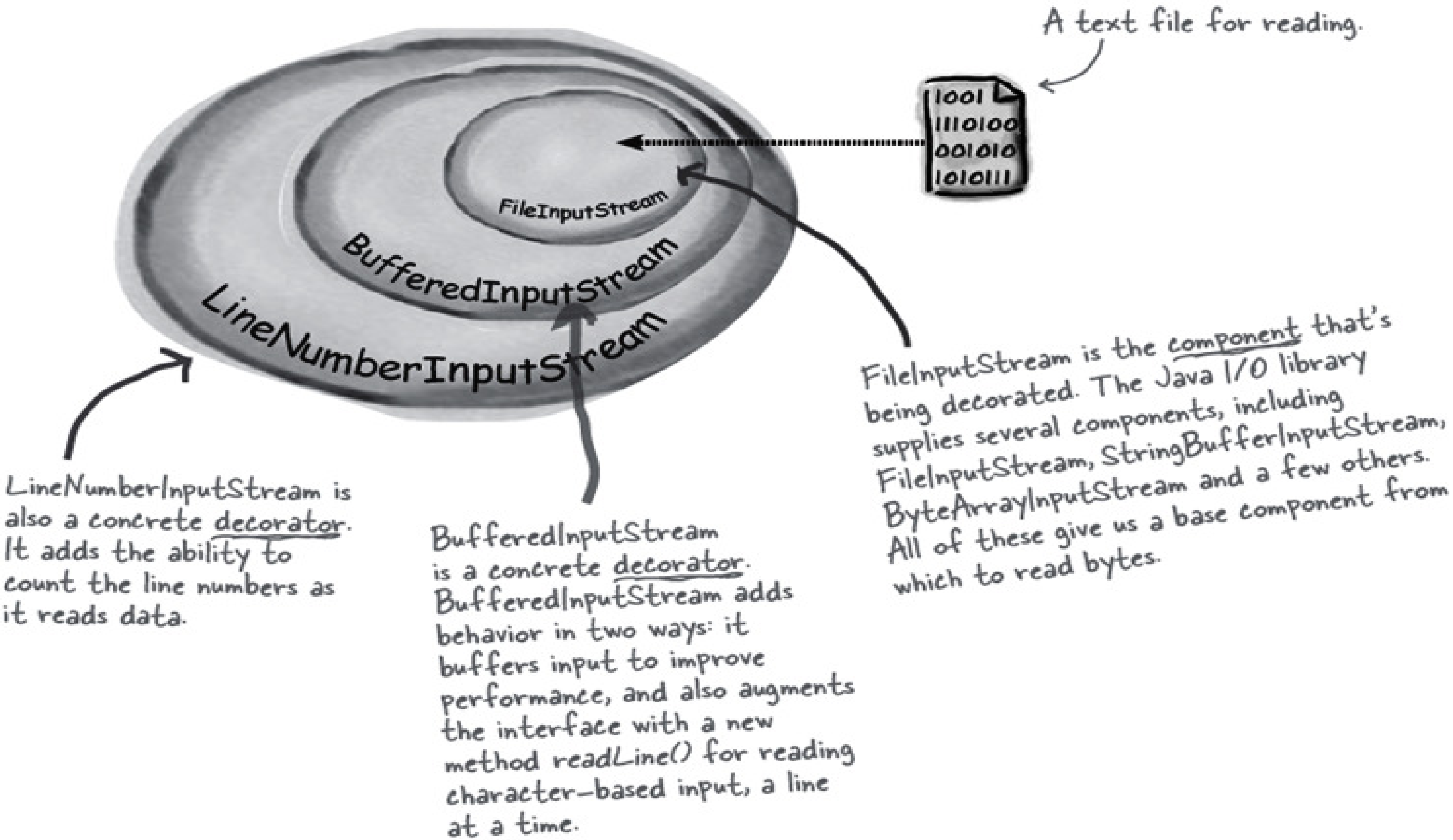
Wrap it with a Mocha.

Wrap it in a second Mocha.

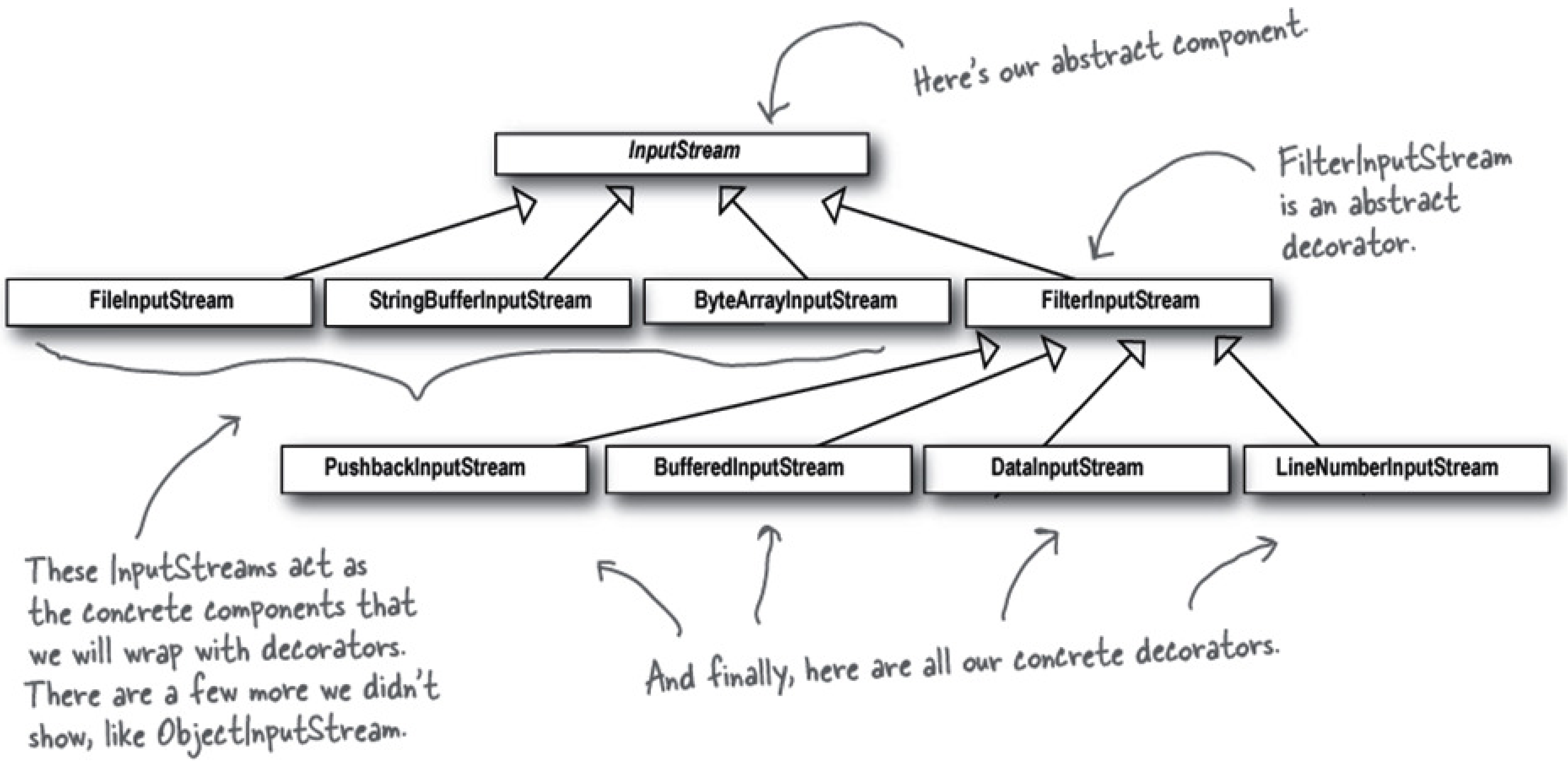
Wrap it in a Whip.

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

Real world decorator – Java I/O



Decorating the java.io classes



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).

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.

```
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.

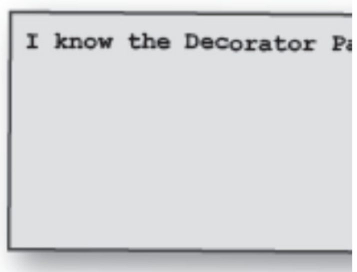
Test out your new Java I/O decorator

```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



test.

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

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();


    }
}
```

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.

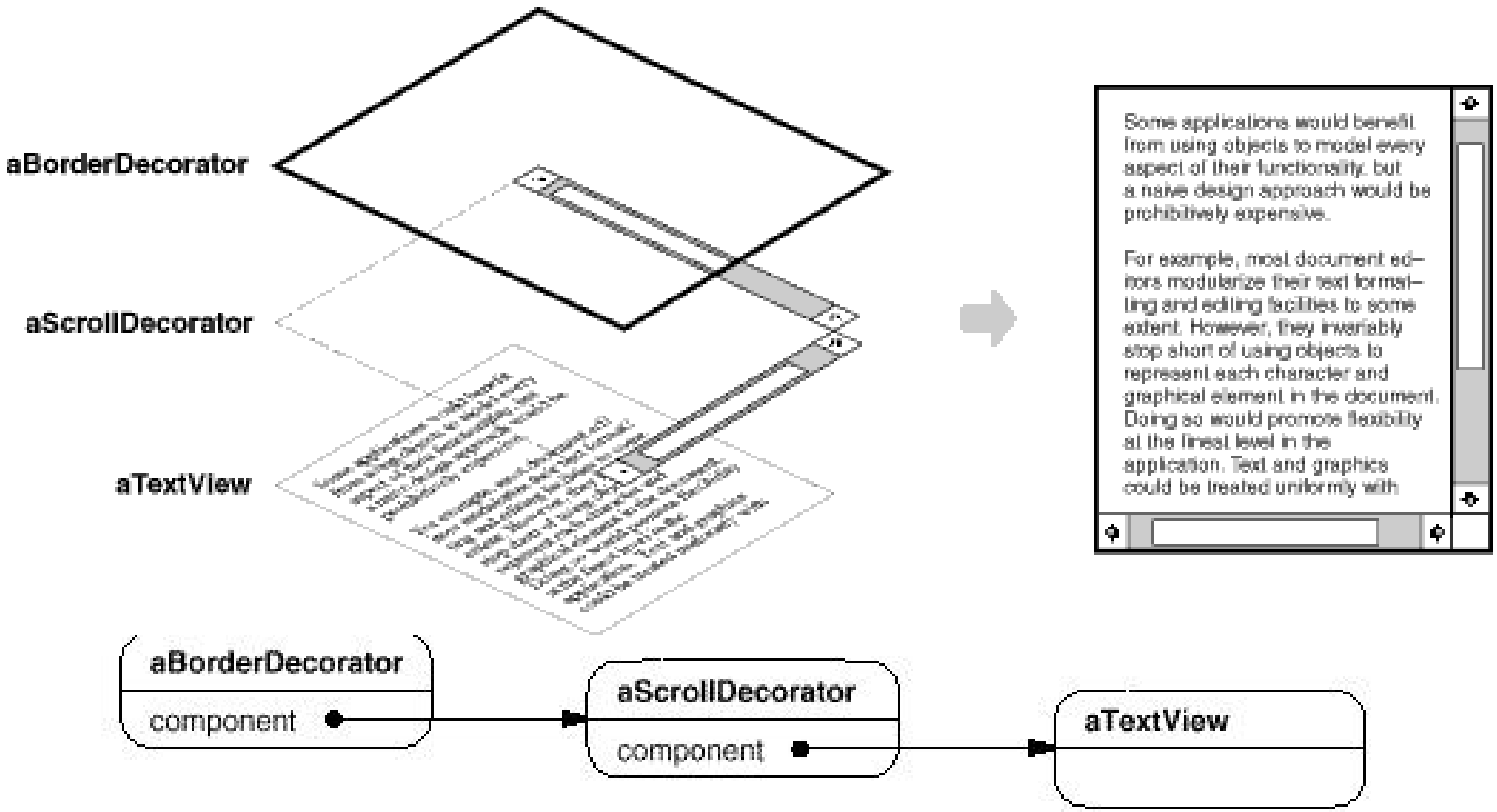
4. Changing the skin of an object versus its guts:

   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.

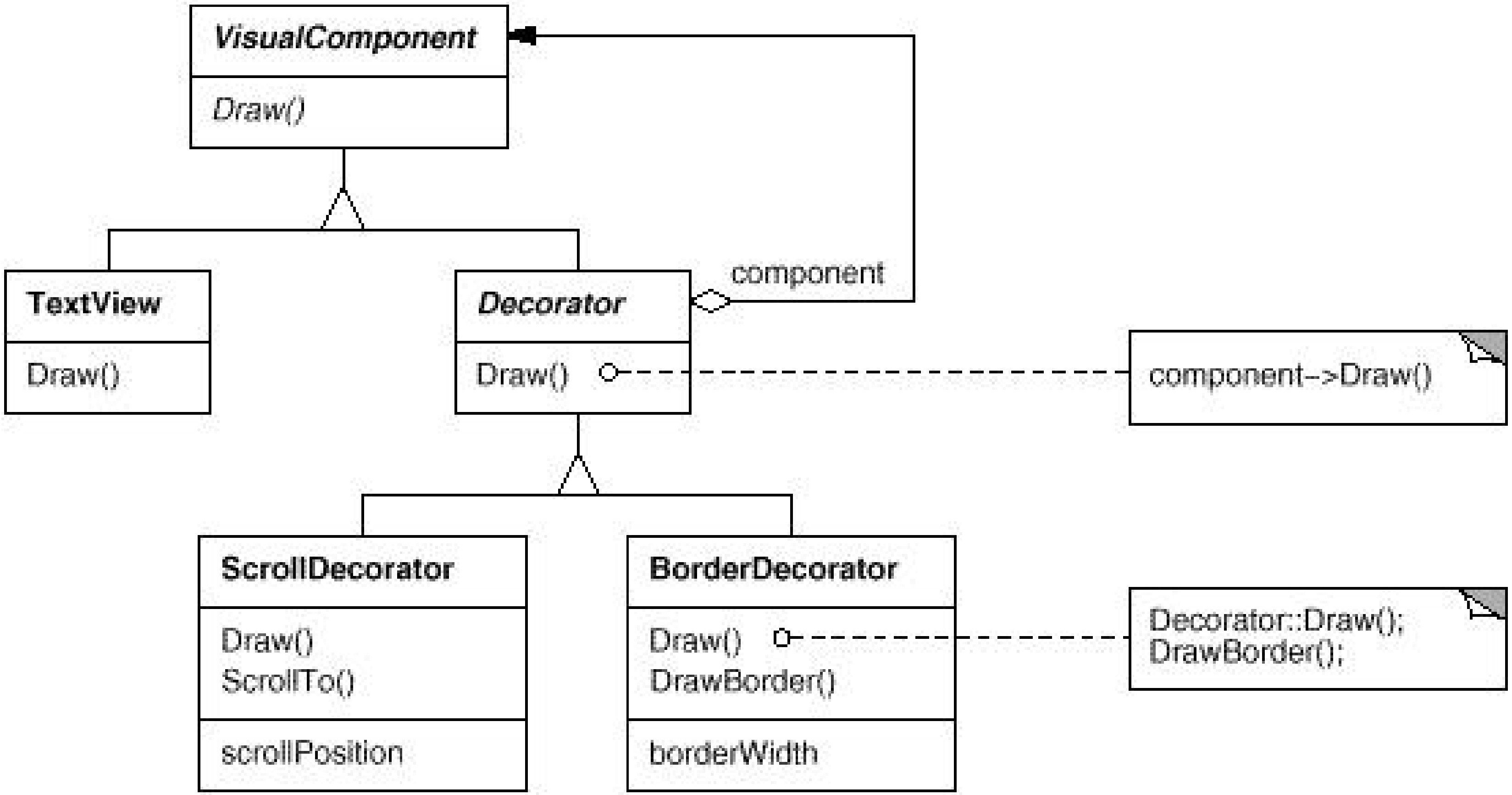
Decorator

- 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!

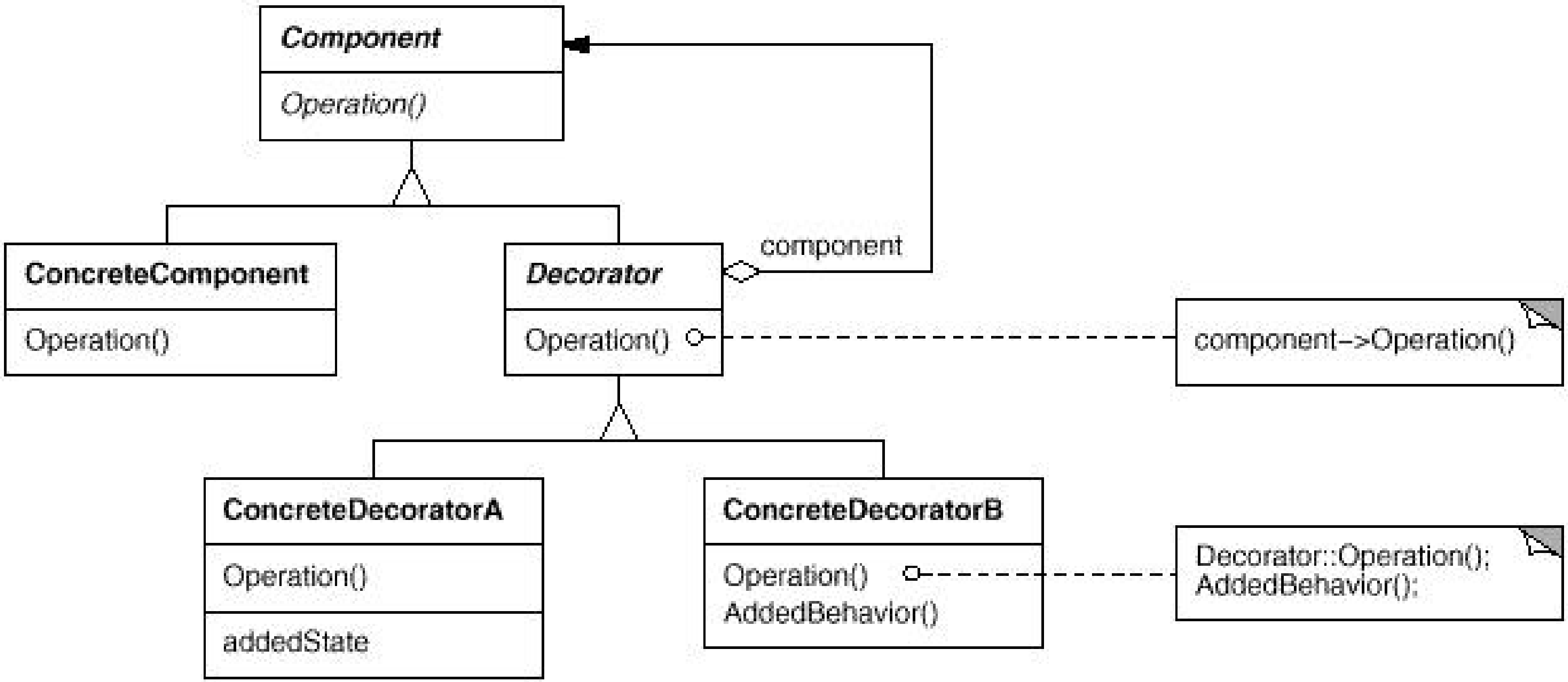
Motivation


aBorderDecorator
aScrollDecorator
aTextView
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 ed–
itors modularize their text format–
ting 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
aBorderDecorator
component
aScrollDecorator
component
aTextView

Structure: the TextView example



Structure



Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();

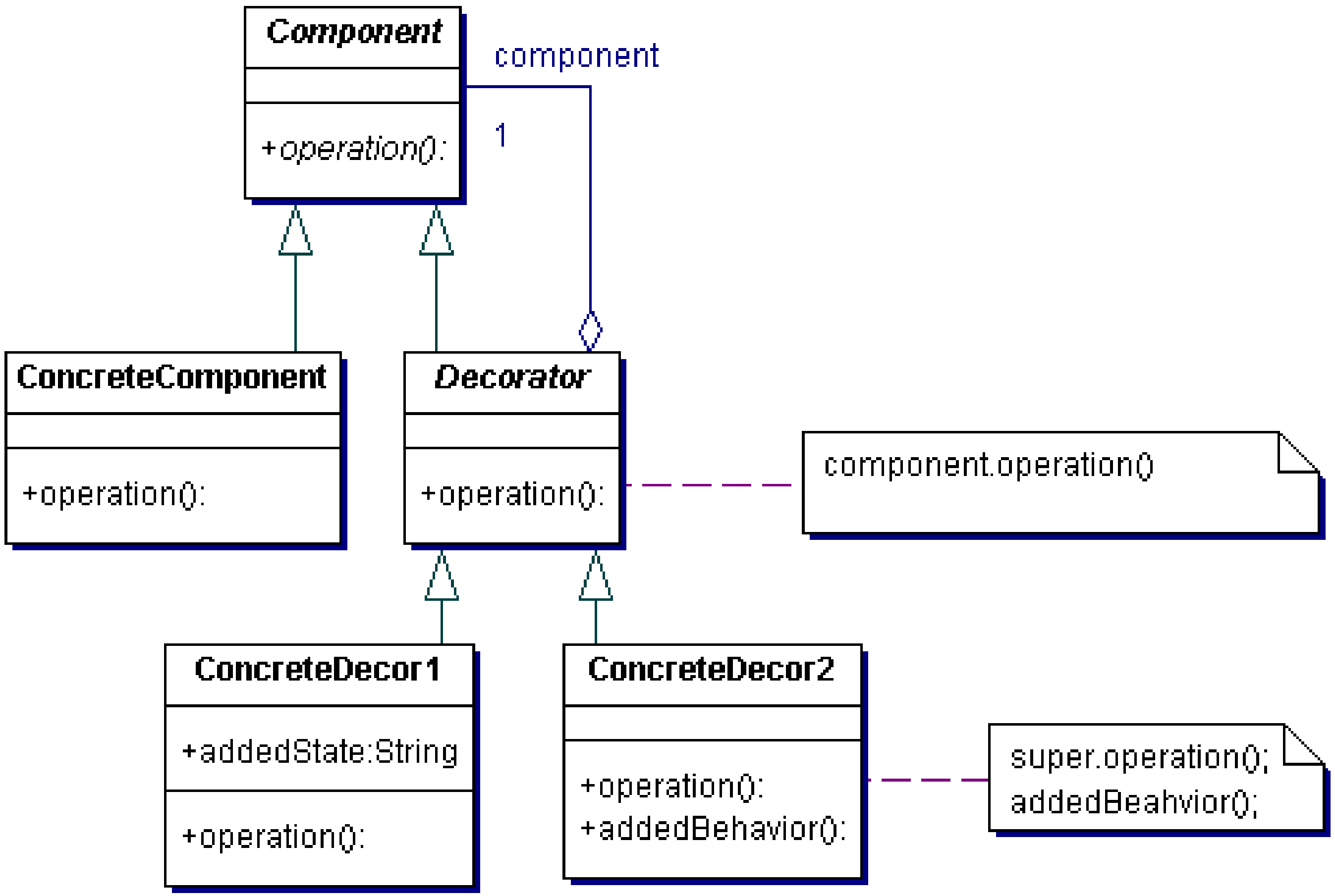
Decorator in Lexi

- 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

Decorator pattern (Wrapper)

- Scopo
  - Aggiungere dinamicamente responsabilità a un oggetto
- Motivazioni
  - Spesso può essere necessario aggiungere responsabilità a un oggetto di una classe e magari successivamente toglierle: ad esempio le barre di scorrimento al testo contenuto in una finestra
  - Se si usano le sottoclassi ci può essere un problema di proliferazione, se si vogliono combinare diverse responsabilità: con decorator si aggiunge una classe per ogni responsabilità e si combinano a piacere le responsabilità, dinamicamente

Decorator: structure



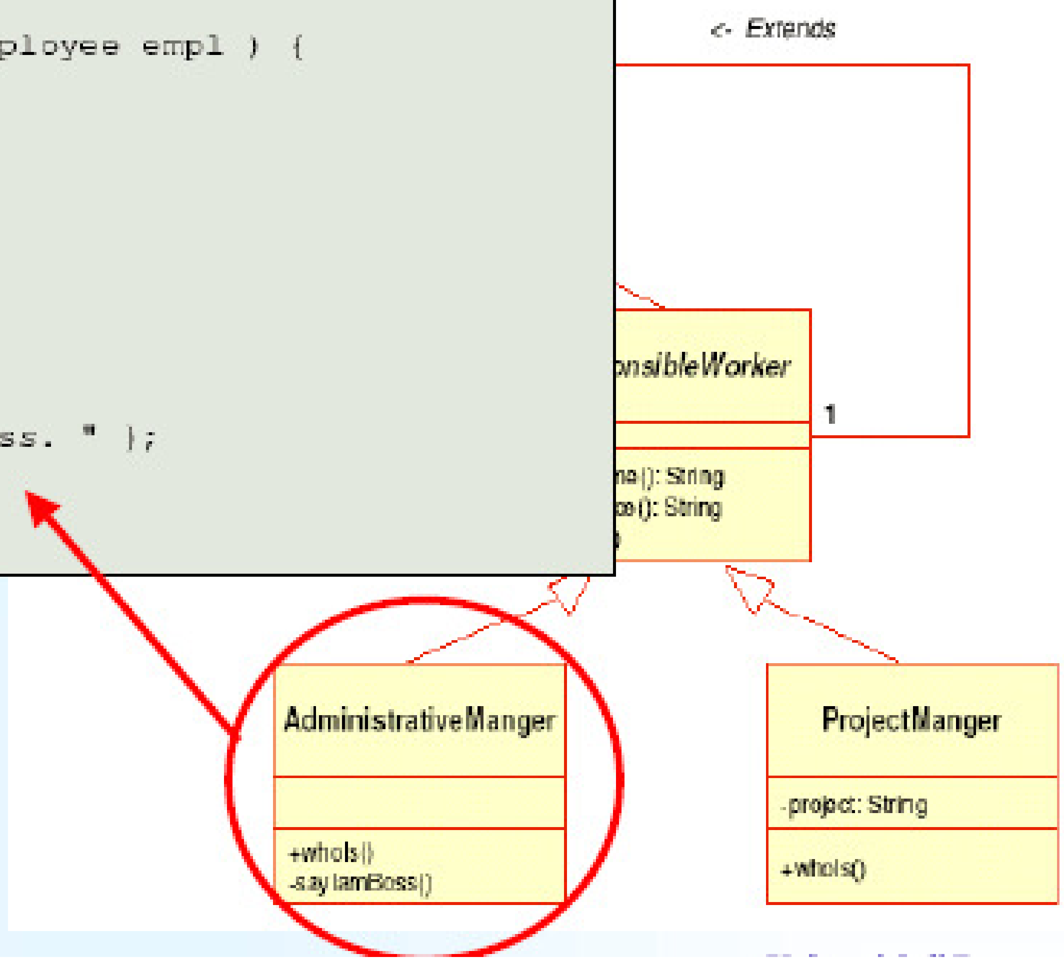
```java
public class AdministrativeManager extends ResponsibleWorker {

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

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

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

}
```



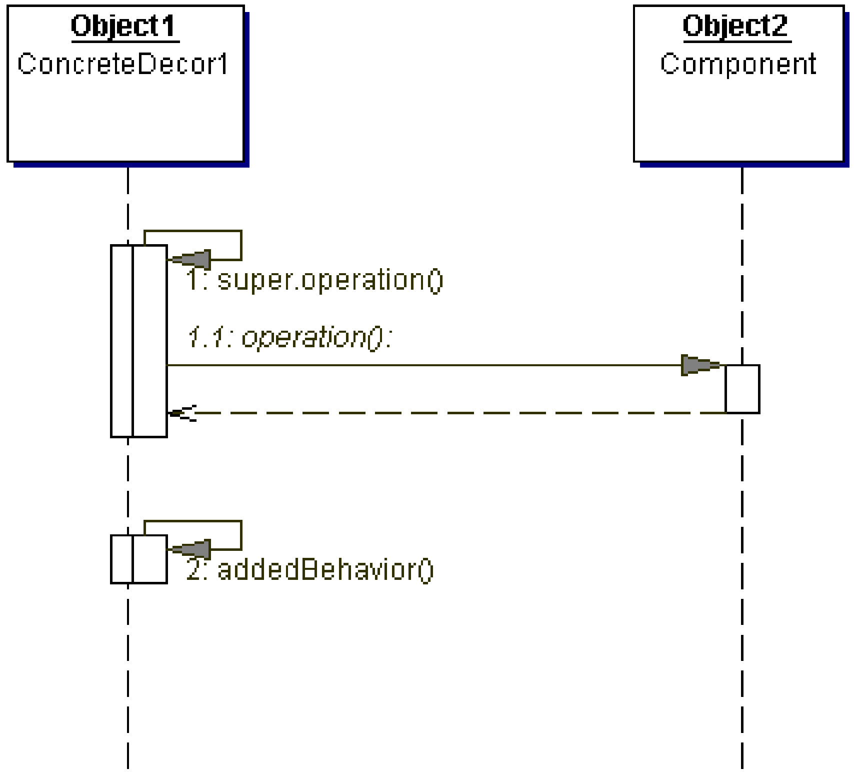
Decorator: participants

- Component
  - Interface of the decorated objects
- ConcreteComponent
  - Base class of objects that can receive new responsibilities
- Decorator
  - Defines an interface conform to the common one and maintains a reference to one object of type component (it can be already decorated or not)

ConcreteDecorator

- Defines a new responsibility

Decorator: collaborazione



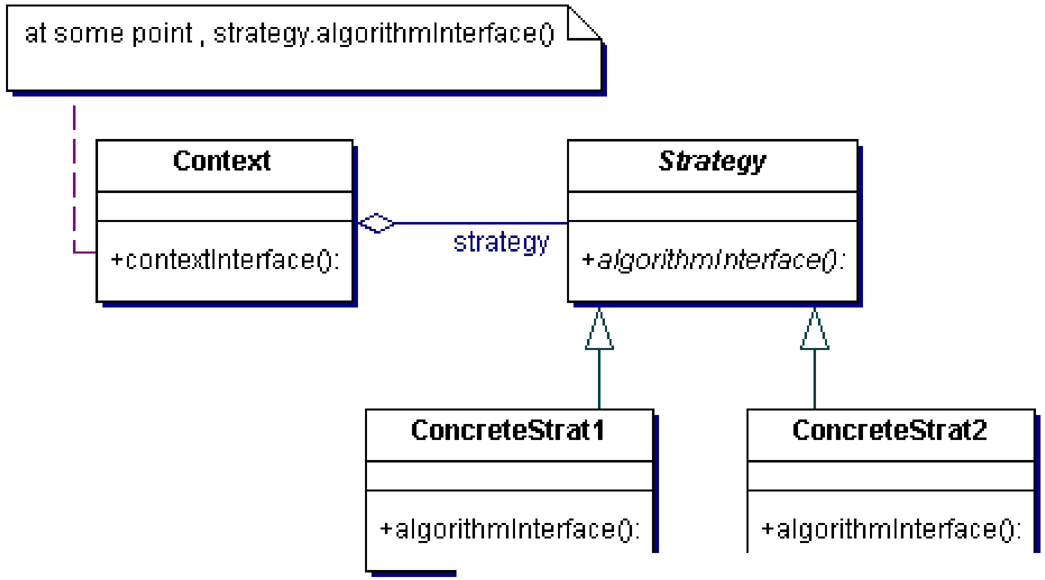
Motivation for the Decorator pattern in a little more detail.

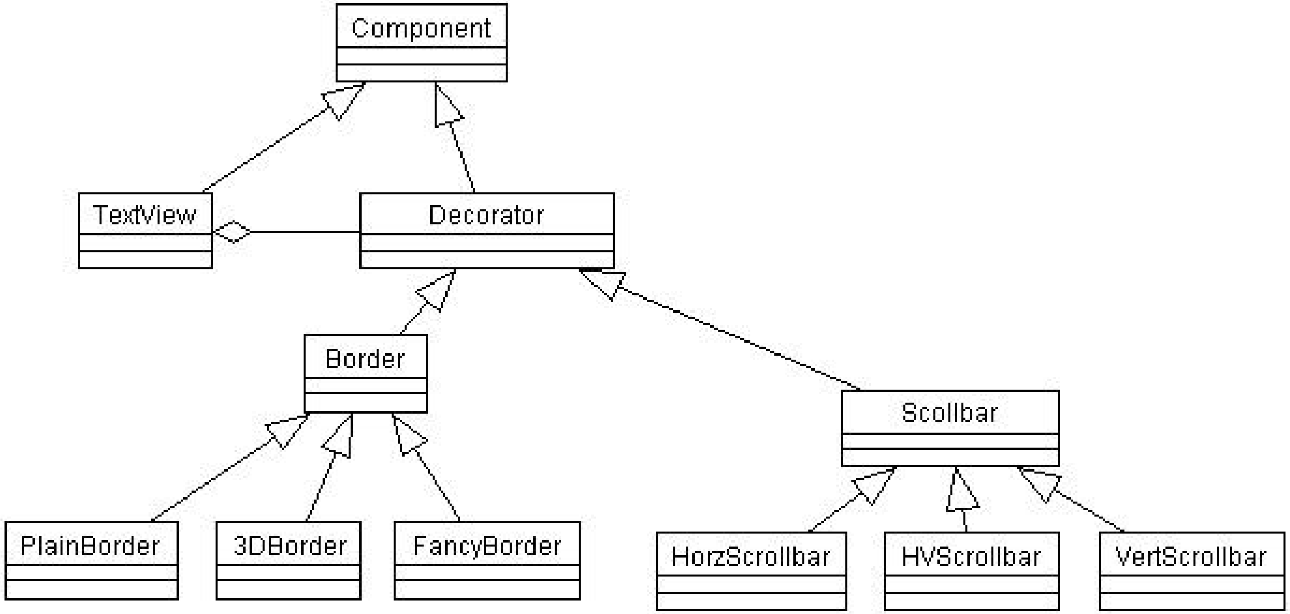
- Suppose we have a TextView GUI component and we want to add different kinds of borders and scrollbars to it.
- Suppose we have three types of borders:
  - Plain, 3D, Fancy
- And two types of scrollbars:
  - Horizontal, Vertical
- Solution 1: Let's use inheritance first. We'll generate subclasses of 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	

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.


at some point , strategy.algorithmInterface()
Context
+contextInterface():
strategy
Strategy
+algorithmInterface():
ConcreteStrat1
+algorithmInterface():
ConcreteStrat2
+algorithmInterface():


Component
TextView
Decorator
Border
Scollbar
PlainBorder
3DBorder
FancyBorder
HorzScrollbar
HVScrollbar
VertScrollbar

Using Strategy

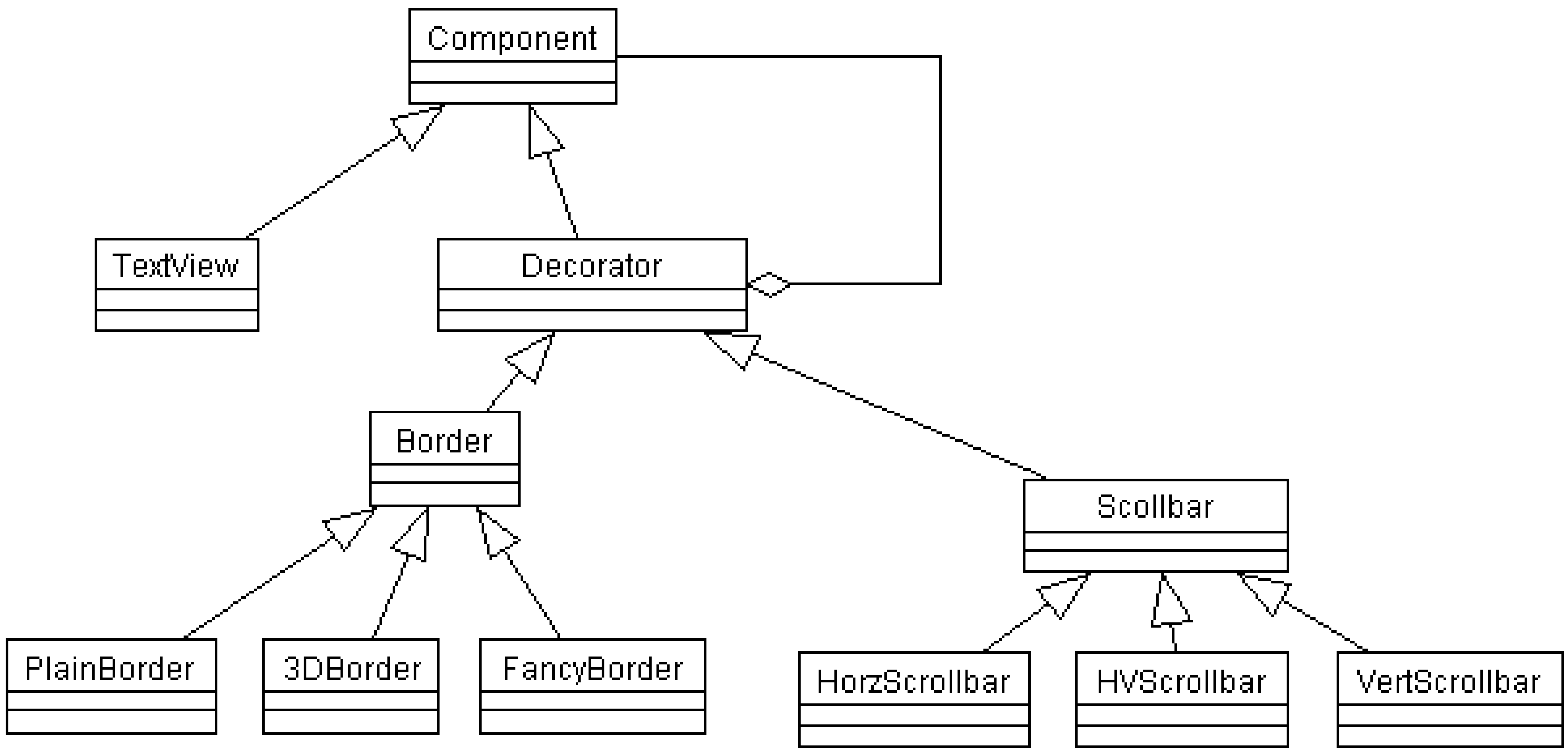
- Now the TextView Class looks like this:

```
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



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.
        }
}
```

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

Homework

The winter holidays will be here (again) before you know it! Being the organized individual you are, you have a plan for next year's holiday tree. Implement a software system that allows you to calculate the price of any tree plus any combination of decorations. The system must be easily extendable in the sense that whenever new decorations are added in the store you will have to at most add one class.

Homework (cont'd)

Here are two tables representing costs of trees and decorations, respectively

Trees	**Cost**
Fraser Fir	12
Colorado Blue Spruce	20

Decorations	**Cost**
Star	4
Balls Red	1
Balls Silver	3
Lights	5

Homework (cont'd)

A very important requirement is that a tree can only have one star. (not trivial to implement this requirement) When a user wants to decorate a tree with a star with a new star you must print a warning that the tree already has a star and not add the price of a star to tree. Users must be able to continue decorating their tree if they add another star to it:

```
Tree mytree = new BlueSpruce();
mytree = new Star(mytree);
mytree = new BallsRed(mytree);
mytree = new Star(mytree);
mytree = new Lights(mytree);
printtree(mytree);
```

should lead to:
Tree already has a star!
Blue spruce tree decorated with, a Star, BallsRed, Lights costs $30.00