# Decorator (装饰器, Structural Pattern)


Kai SHI

# Problem: Cafe Ordering System

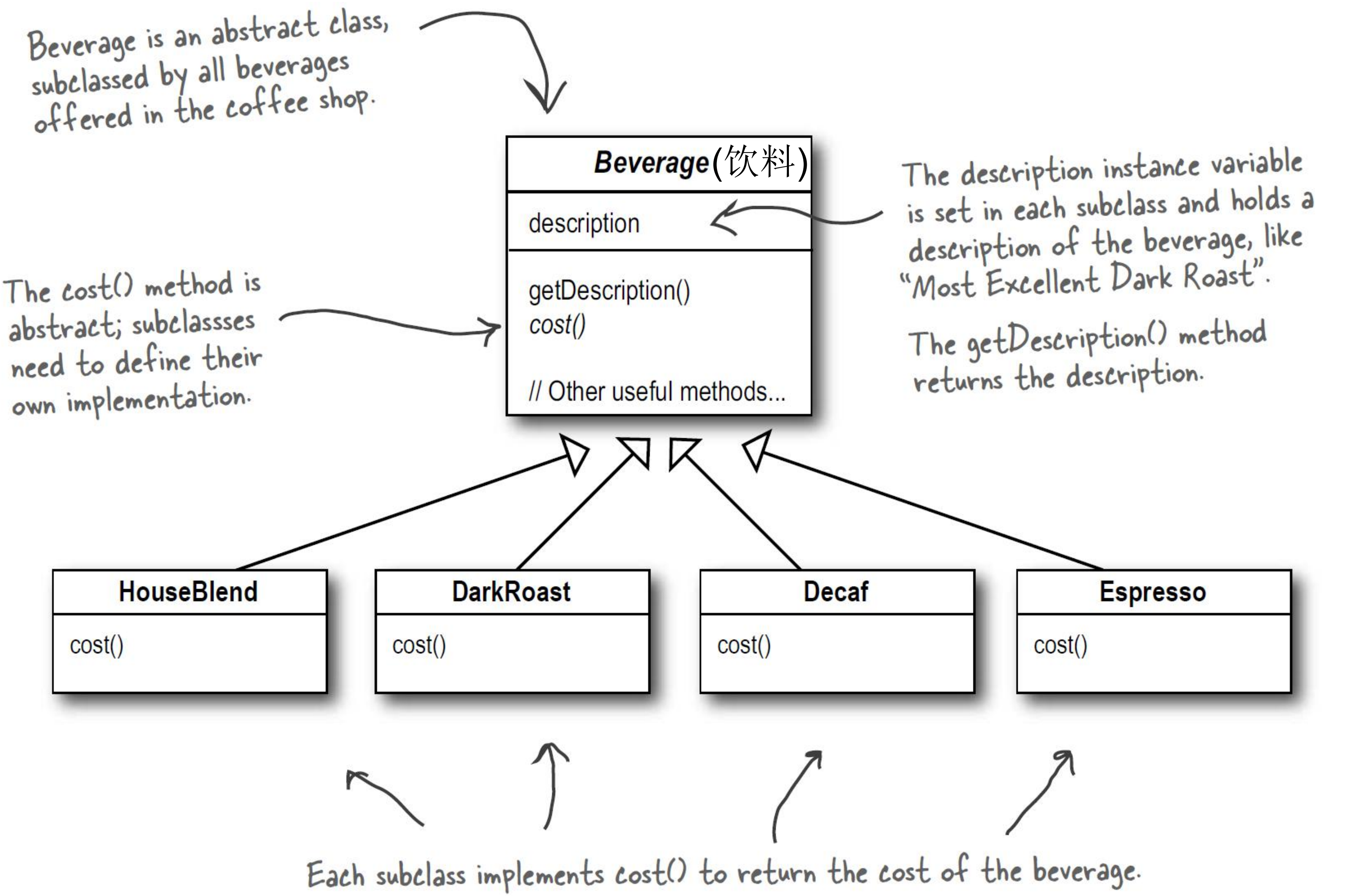
- House blend (混合咖啡)
- Dark Roast (深度烘培)
- Decaf (低咖啡因咖啡)
- Espresso (意式浓缩咖啡)





- Draw class diagram NOW

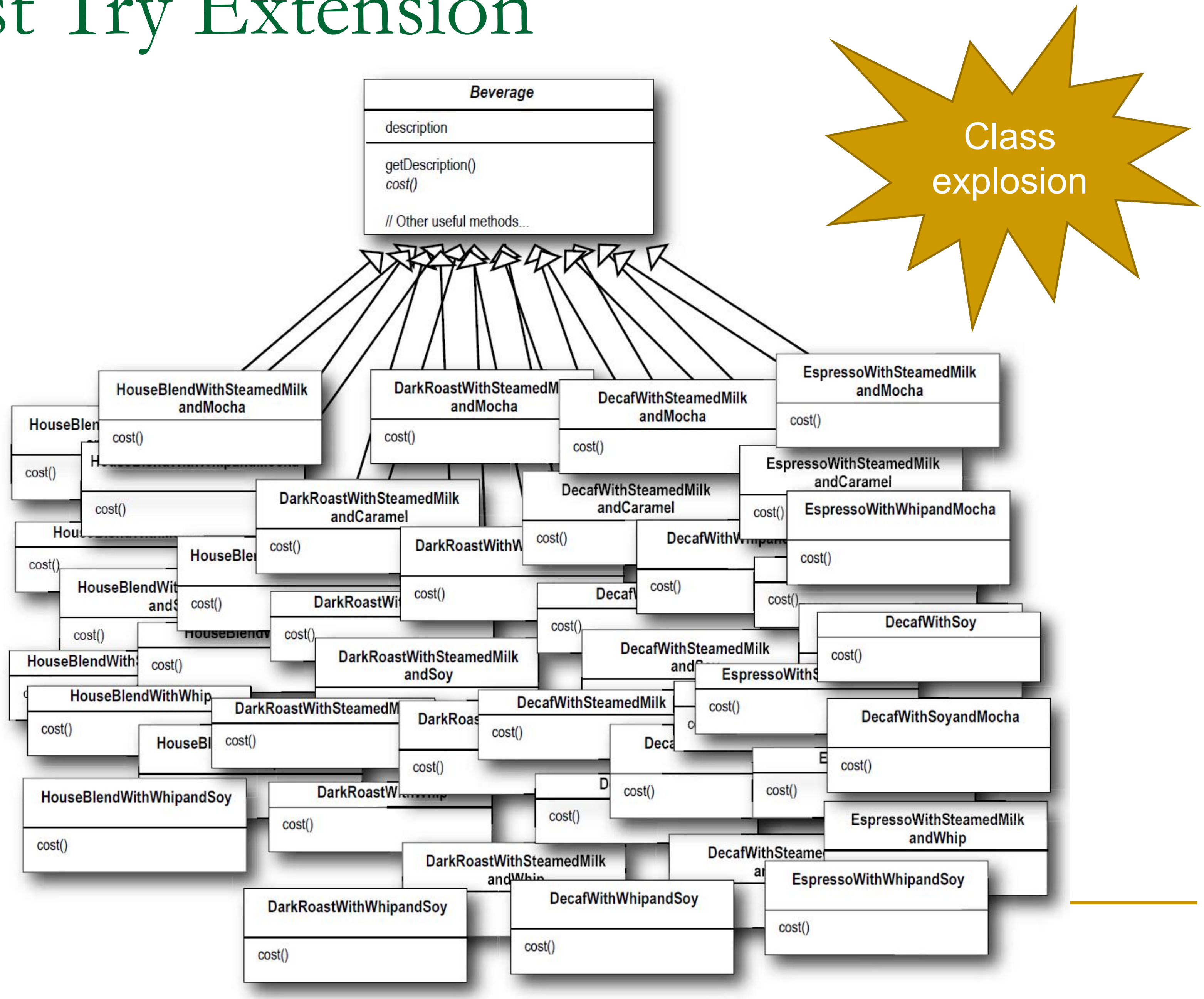
# First Try


Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.
Beverage(饮料)
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.

# Requirements Change: Customers can add condiments

- Customers can ask for several condiments
  - steamed milk,
  - soy,
  - mocha (i.e., chocolate),
  - topped off with whipped milk.
- Cafe charges a bit for each of these, so they need to get them built into their order system.

# First Try Extension

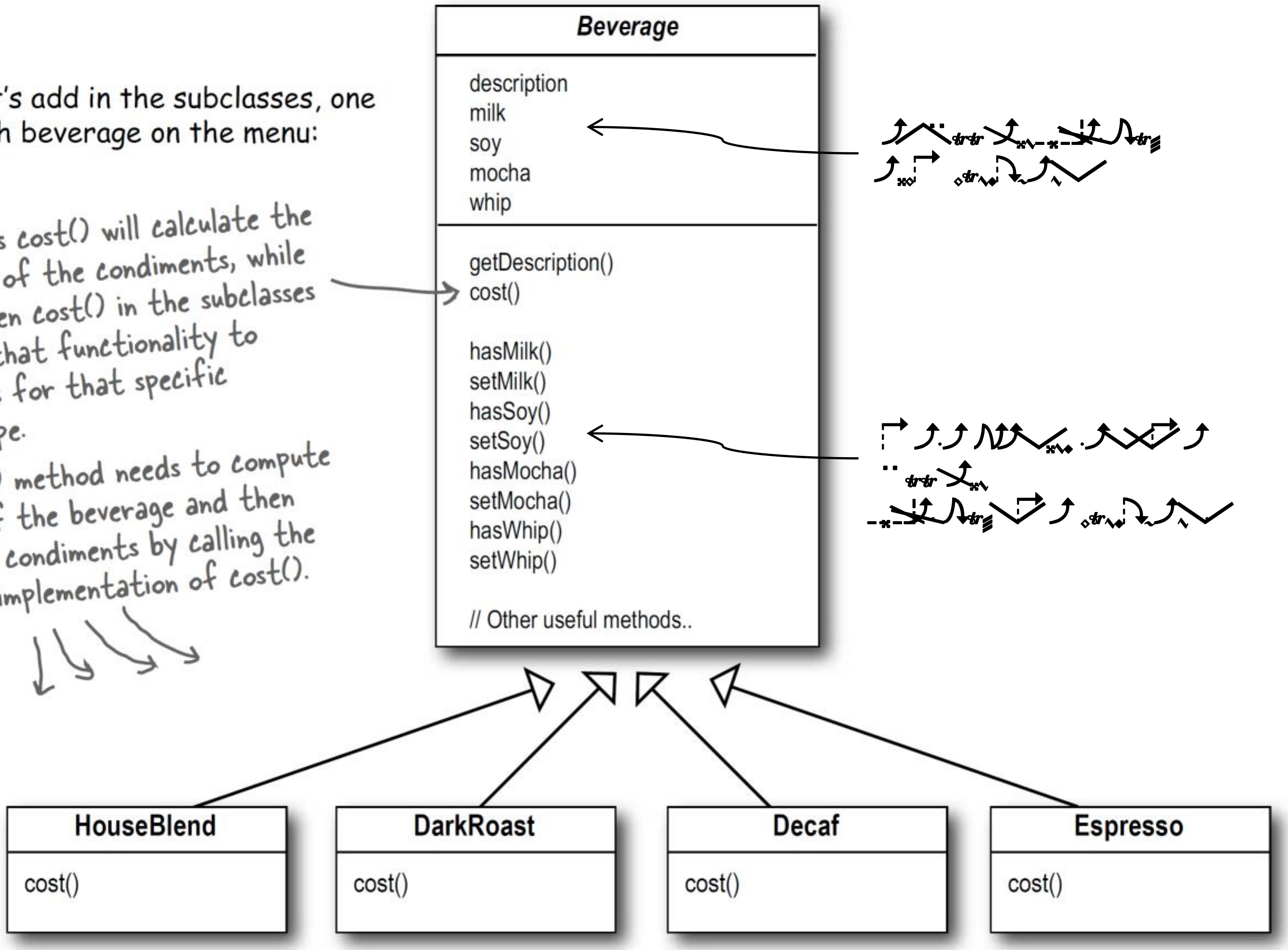


# Second Try

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

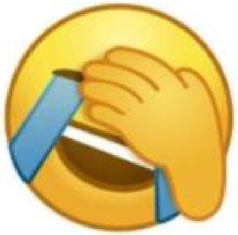
The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().



# Requirements Change:

1. Price changes of condiments
2. New condiments

- We have to modify class Beverage. 

# Design Principle

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

# Rethink (1/2)


WHAT'S IN YOUR COFFEE?
COFFEE
MILK
FROTHED MILK
WATER
CHOCOLATE
ICE CREAM
ESPRESSO
MACCHIATO
AMERICANO
LATTE
CAPPUCCINO
FLAT WHITE
MOCHA
AFFOGATO
ICED COFFEE

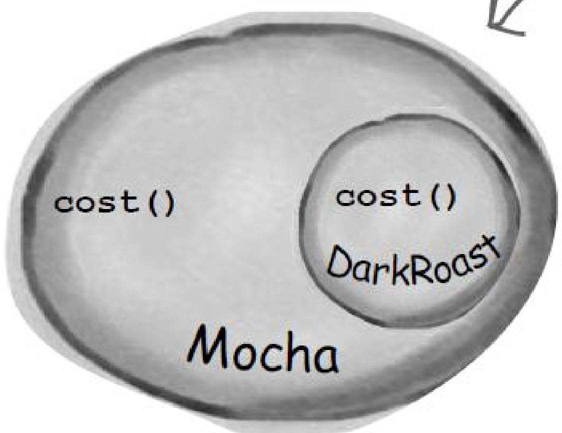
# Rethink (2/2): Parfait

# Constructing a drink order with Decorators

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


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

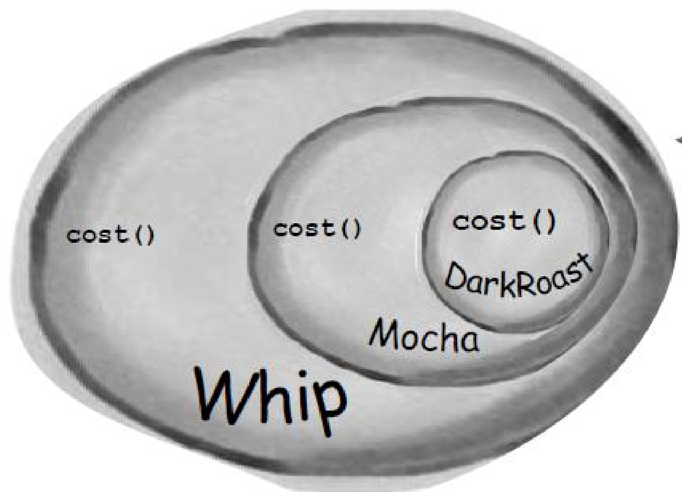
**❷ The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**




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).

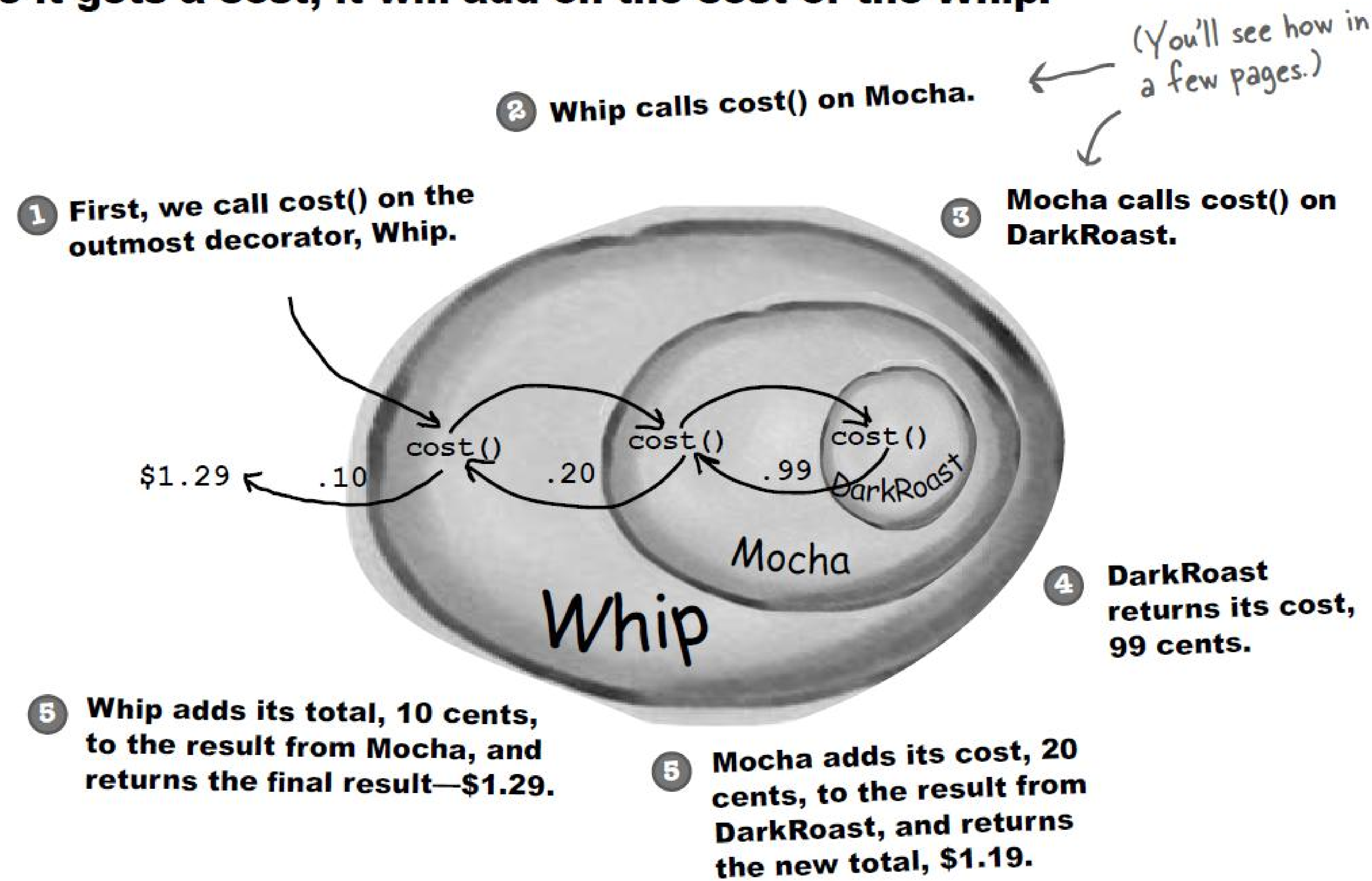
❸ **The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.**




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.**

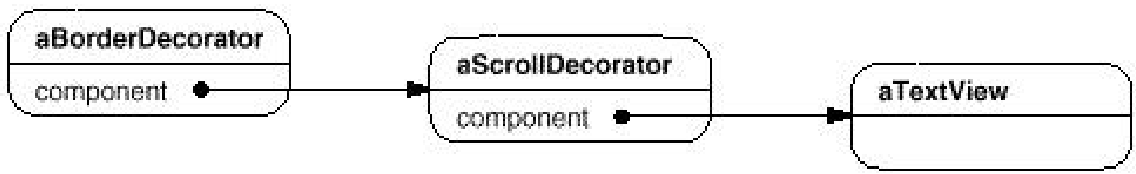


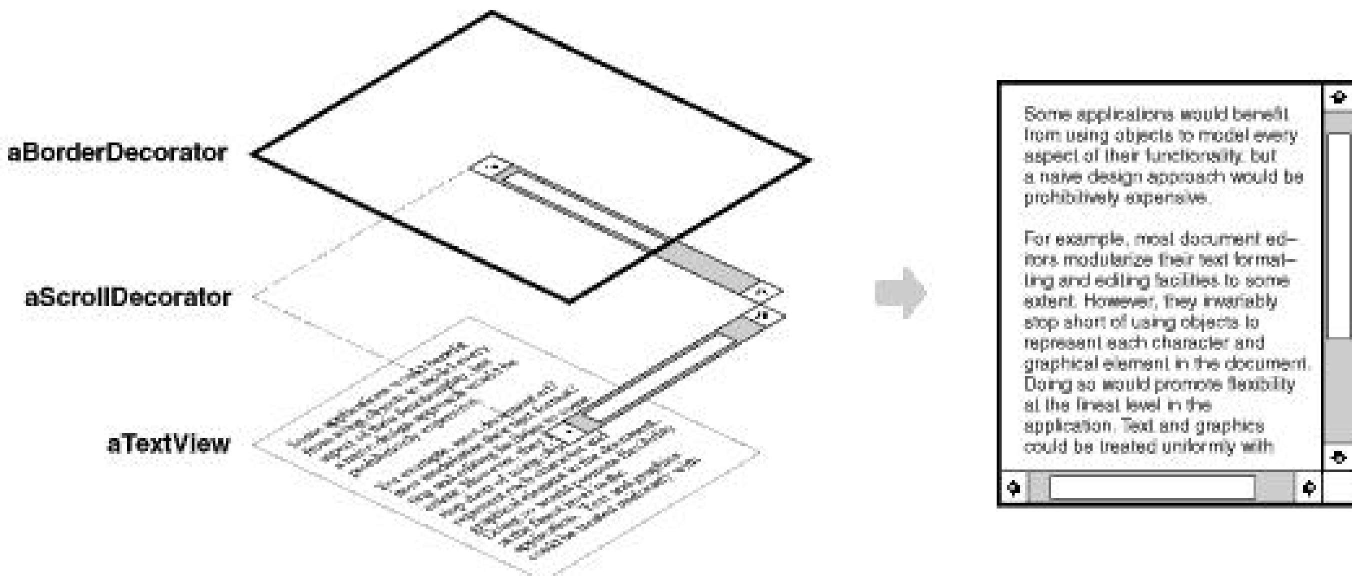
# Decorator Pattern

- Intent
  - Attach additional responsibilities to an object dynamically.
  - Decorators provide a flexible alternative to subclassing for extending functionality.
    - Dynamically extension;
    - Better than inheritance;
- Also Known As
  - Wrapper (包装类)

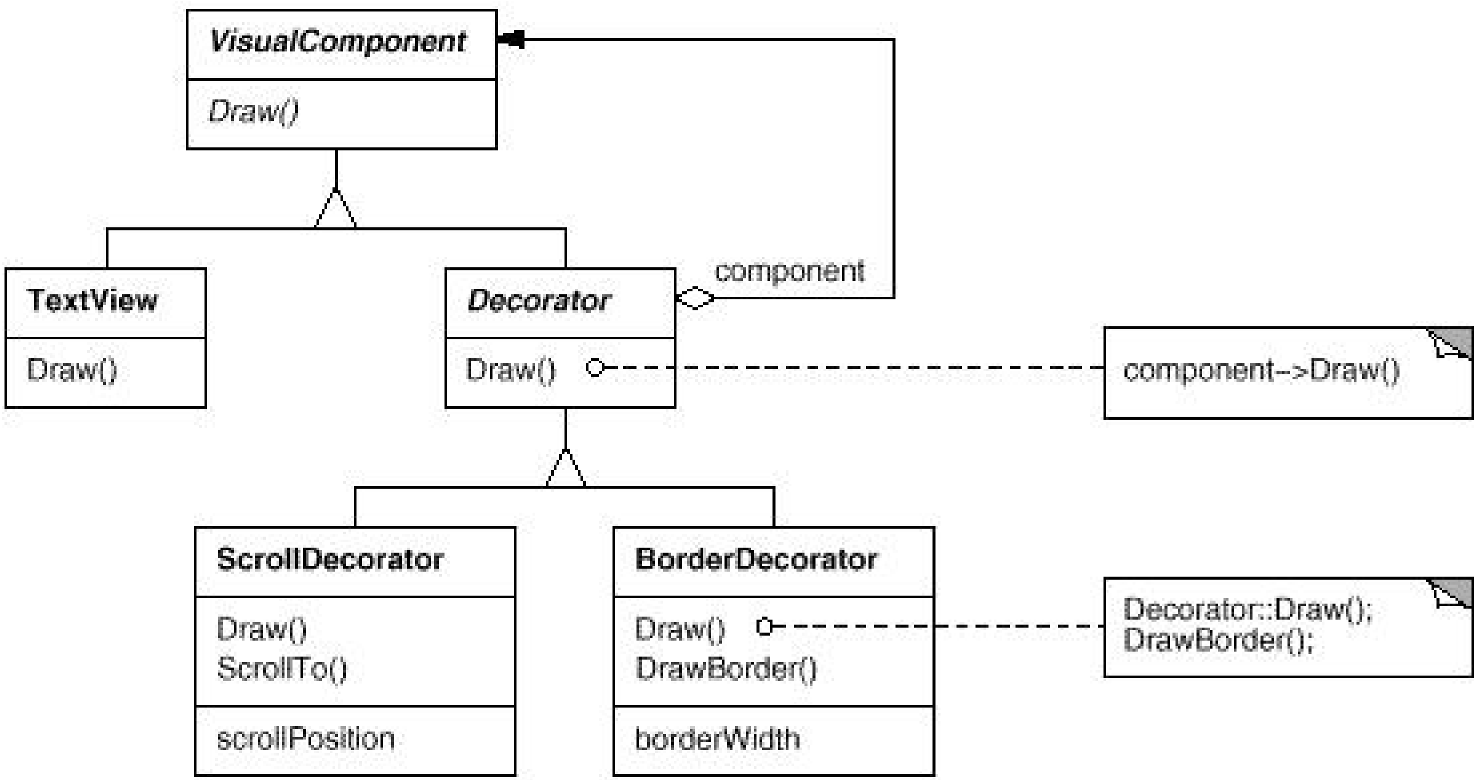
# Motivation

- We want to add properties, such as borders or scrollbars to a GUI component. We can do this with inheritance, but this limits flexibility. A better way is to use composition!



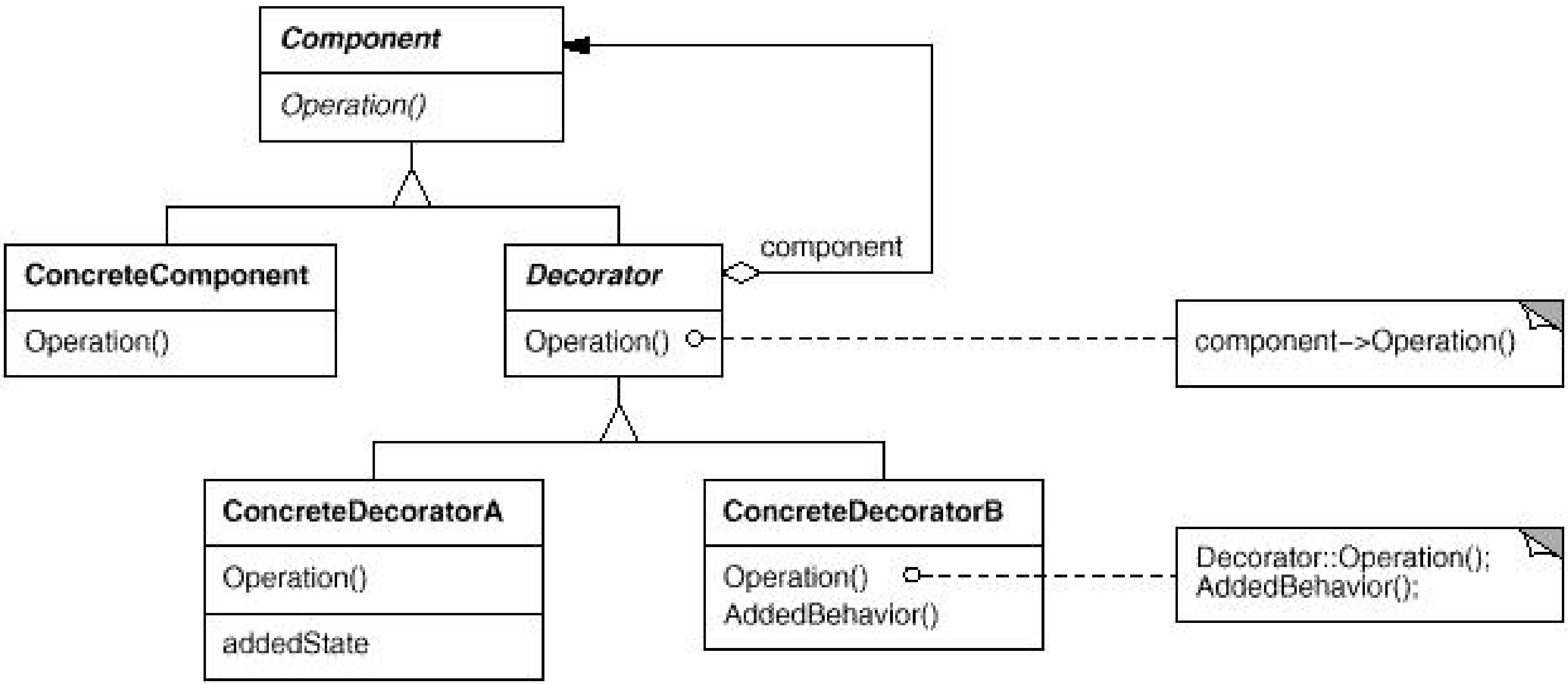
# Motivation (2/2): Class Diagram



# Applicability:
# Use Decorator when:

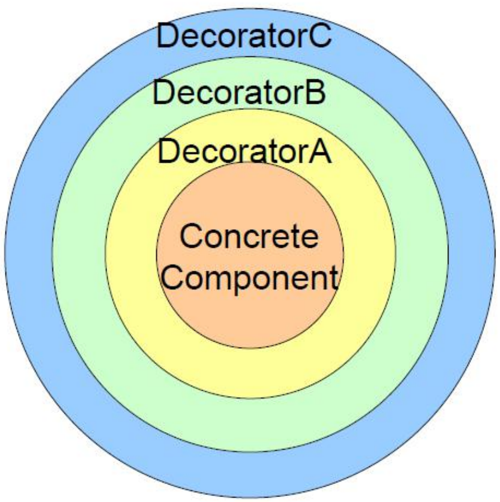
- To add responsibilities to individual objects dynamically without affecting other objects.
- When extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

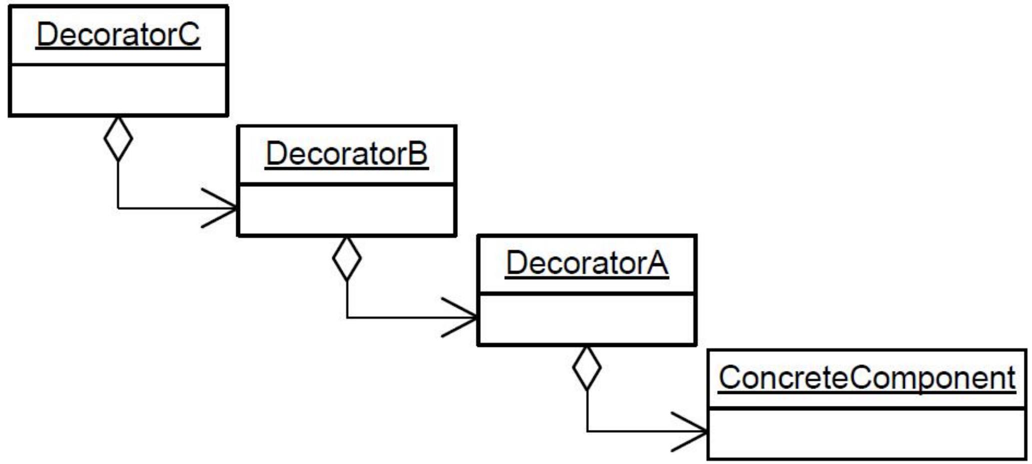
# Class Diagram (GoF)



# Participants

- Component: defines the interface for objects that can have responsibilities added to them dynamically.
- ConcreteComponent: defines an object to which additional responsibilities can be attached.
- Decorator: maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- ConcreteDecorator: adds responsibilities to the component.




```java
Component component =
   new DecoratorC(
       new DecoratorB(
           new DecoratorA(
               new ConcreteComponent())));
```

# Collaborations

- Decorator forwards requests to its Component object.
- It may optionally perform additional operations before and/or after forwarding the request.

# Consequences: Advantages

- More flexibility than static inheritance.
  - With Decorators, responsibilities can be added and removed at runtime simply by attaching and detaching them.
- Avoid “Class explosion”.
- By permutation and combination, lots of behavioral combinations can be created.

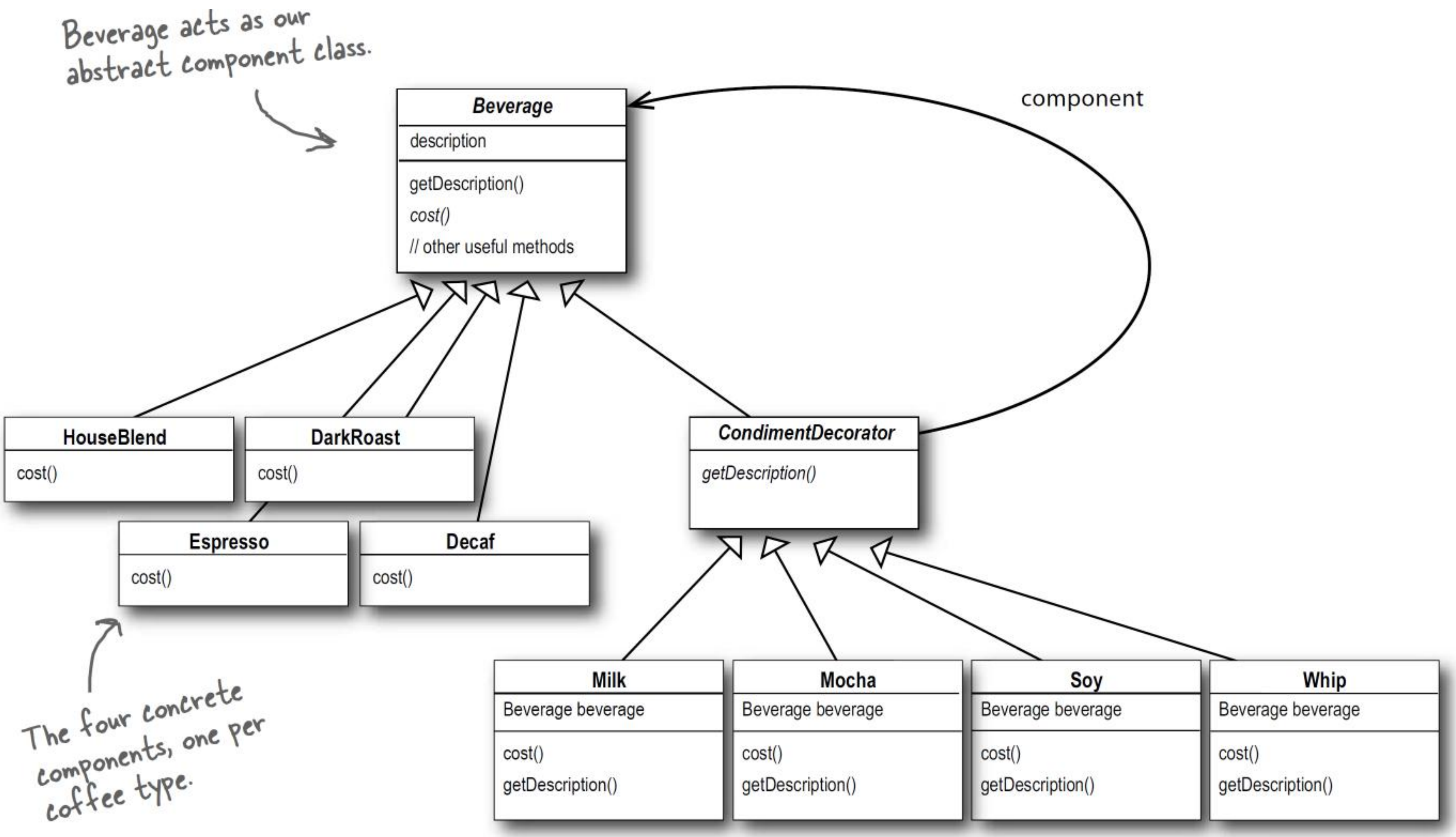
# Consequences: Disadvantages

- A decorator and its component are not same.
  - A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not same to the component itself. Hence you shouldn't rely on object identity when you use decorators.
- Lots of little objects.
  - Hard to learn and debug.

# Implementation

- Interface conformance (一致)
  - A decorator object's interface must conform to the interface of the component it decorates
- Keeping Component classes lightweight
  - To ensure a conforming interface, components and decorators must inherit from a common Component class.
  - The complexity of the Component class might make the decorators too heavyweight to used.
  - Putting a lot of functionality into Component also increases the probability that concrete subclasses will pay for features they don't need.

# Cafe Decorator Version: Class Diagram



# Cafe Decorator Version: Code

- code: net.dp.decorator

# Example: Java I/O classes (0/4)

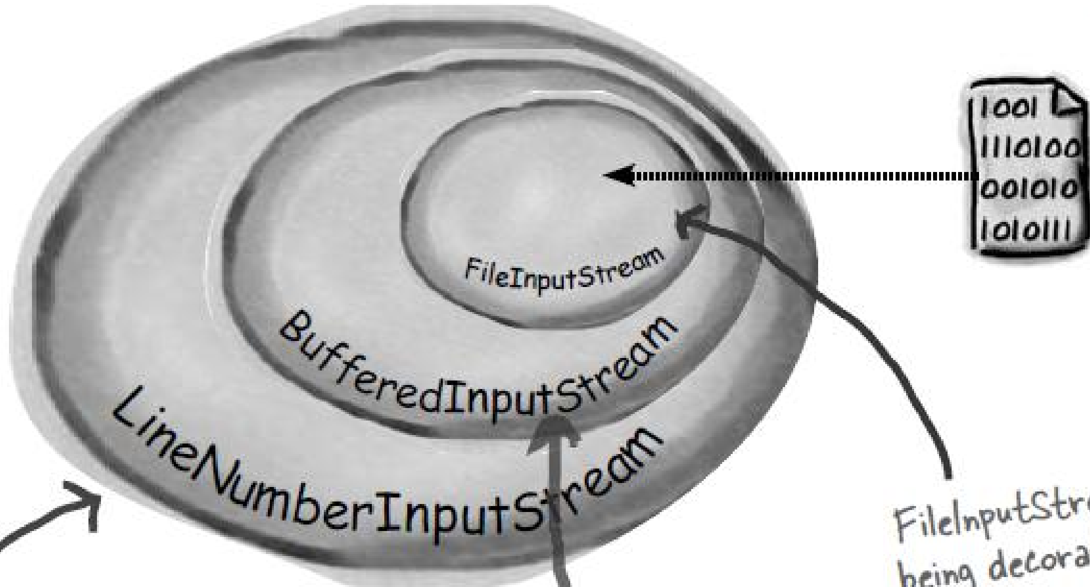
- The large number of classes in the java.io package is... overwhelming.
- BufferedInputStream BufferedOutputStream BufferedReader BufferedWriter ByteArrayInputStream ByteArrayOutputStream CharArrayReader CharArrayWriter Console DataInputStream DataOutputStream File FileDescriptor FileInputStream FileOutputStream FilePermission FileReader FileWriter FilterInputStream FilterOutputStream FilterReader FilterWriter InputStream InputStreamReader LineNumberInputStream LineNumberReader ObjectInputStream ObjectInputStream.GetField ObjectOutputStream ObjectOutputStream.PutField ObjectStreamClass ObjectStreamField OutputStream OutputStreamWriter PipedInputStream PipedOutputStream PipedReader PipedWriter PrintStream PrintWriter PushbackInputStream PushbackReader RandomAccessFile Reader SequenceInputStream SerializablePermission StreamTokenizer StringBufferInputStream StringReader StringWriter Writer

# Example: Java I/O classes (1/4)

- Java I/O classes use the Decorator pattern
- The basic I/O classes are InputStream, OutputStream, Reader and Writer. These classes have a very basic set of behaviors.
- We would like to add additional behaviors to an existing stream to yield, for example:
  - Buffered Stream - adds buffering for the stream
  - Data Stream - allows I/O of primitive Java data types
  - Pushback Stream - allows undo operation
- We really do not want to modify the basic I/O classes to achieve these behaviors, so we use decorator classes, which Java calls filter classes, to add the desired properties using composition.

# Example: Java I/O classes (2/4)

A text file for reading.



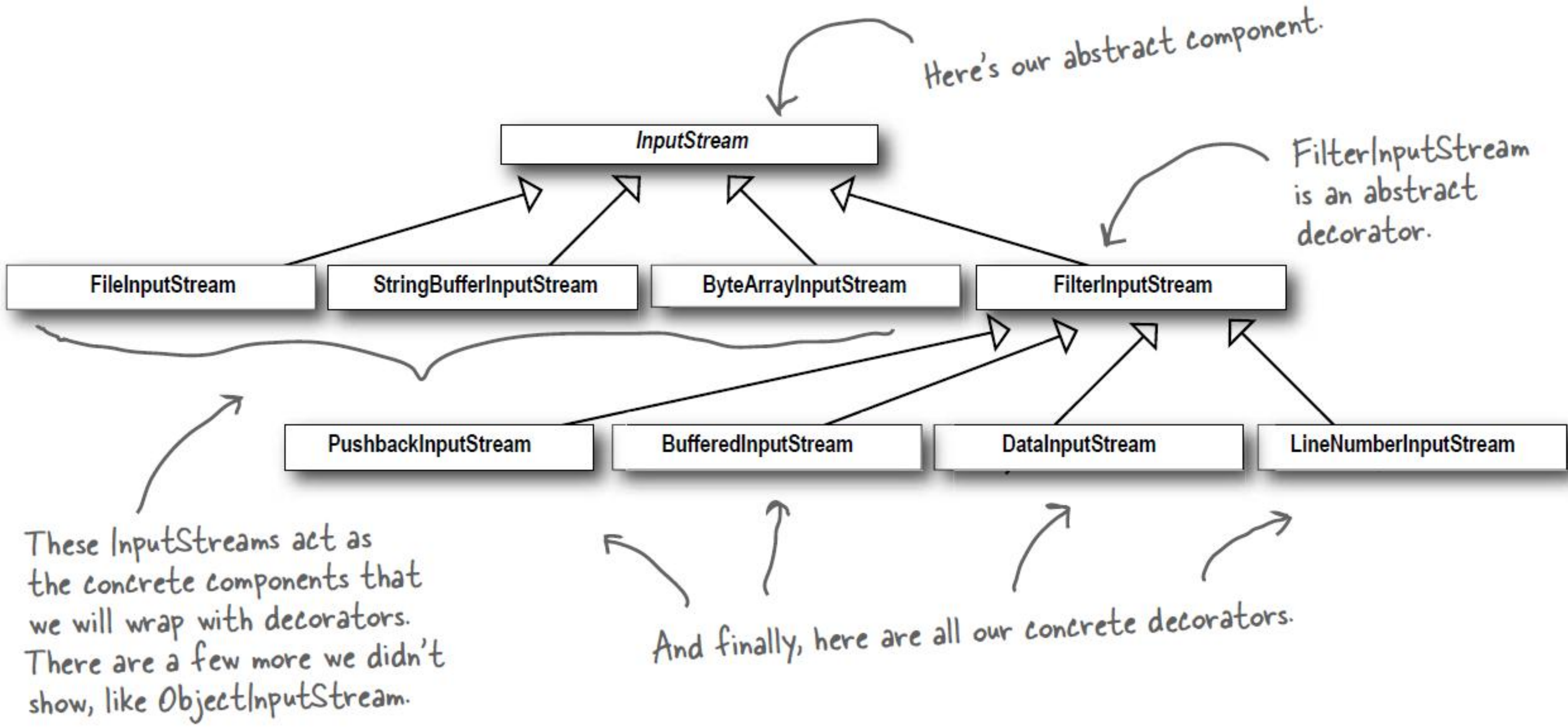

FileInputStream is the component that's being decorated The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

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.

# Example: Java I/O classes (3/4)

## Decorating the java.io classes



# Example: Java I/O classes (4/4)

```
public class JavaIO {
    public static void main(String[] args) throws FileNotFoundException {
        // Open an InputStream.
        FileInputStream in = new FileInputStream("test.dat");
        // Create a buffered InputStream.
        BufferedInputStream bin = new BufferedInputStream(in);
        // Create a buffered, data InputStream.
        DataInputStream dbin = new DataInputStream(bin);
        // Create an unbuffered, data InputStream.
        DataInputStream din = new DataInputStream(in);
        // Create a buffered, pushback, data InputStream.
        PushbackInputStream pbdbin = new PushbackInputStream(dbin);
    }
}
```

Code: decorator.javaio.JavaIO