

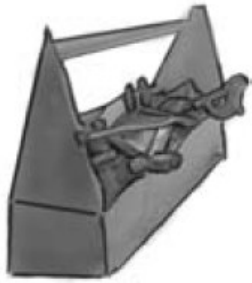
# Decorator Pattern

---

„Vererbst du noch oder designst du schon?“



# Werkzeugkasten



## OO Basics

Abstraction  
Encapsulation  
Inheritance  
Polymorphism  
Interface

## OO Principles

Encapsulate what varies.  
Favor composition over inheritance.  
Program to interfaces, not implementations.  
Strive for loosely coupled designs between objects that interact.

Here's your newest principle. Remember, loosely coupled designs are much more flexible and resilient to change.

## OO Patterns

Strategy  
Encapsulation  
Interface  
Variation

**Observer** – defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

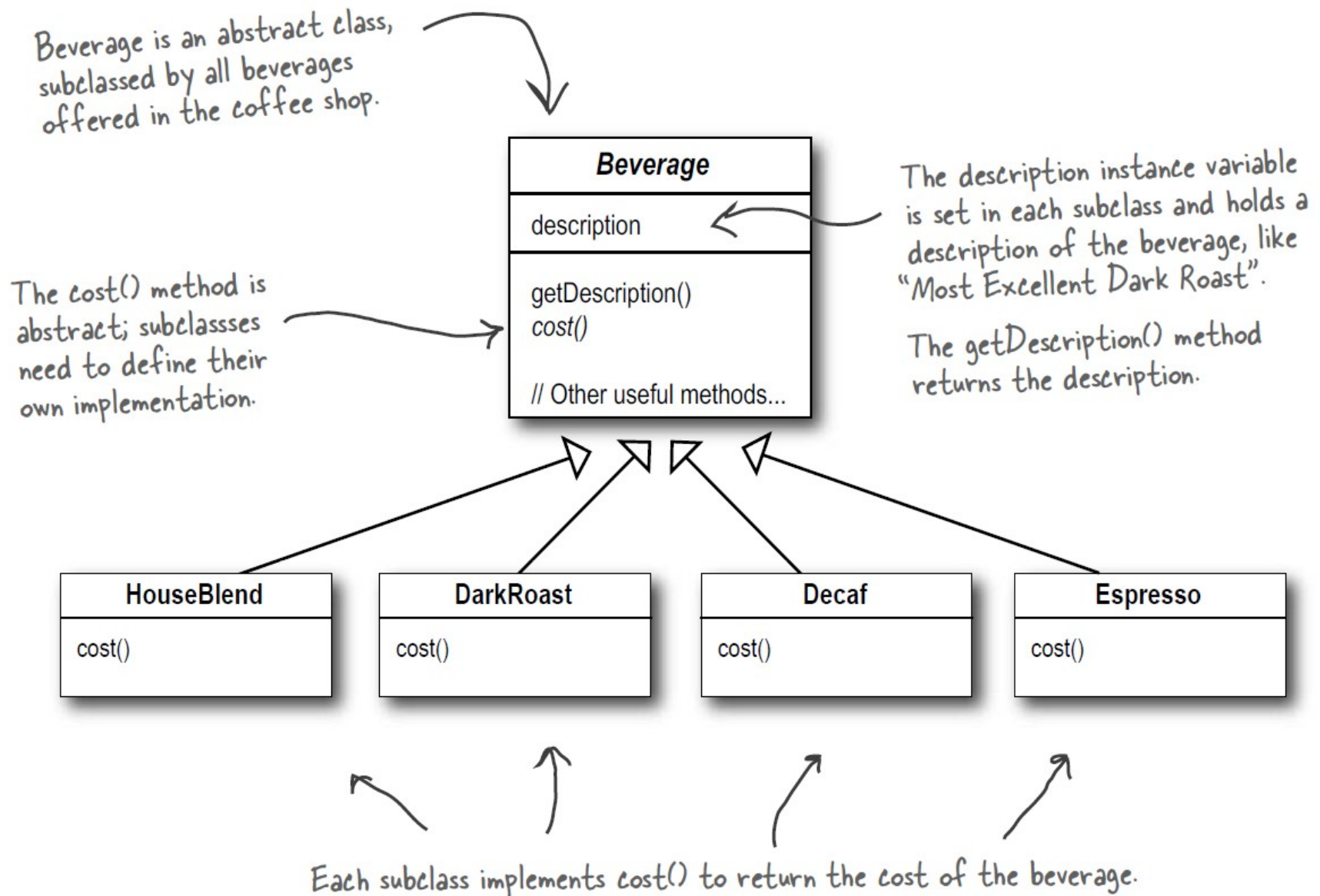
A new pattern for communicating state to a set of objects in a loosely coupled manner. We haven't seen the last of the Observer Pattern – just wait until we talk about MVC!

# Willkommen bei „Sternback-Kaffee“

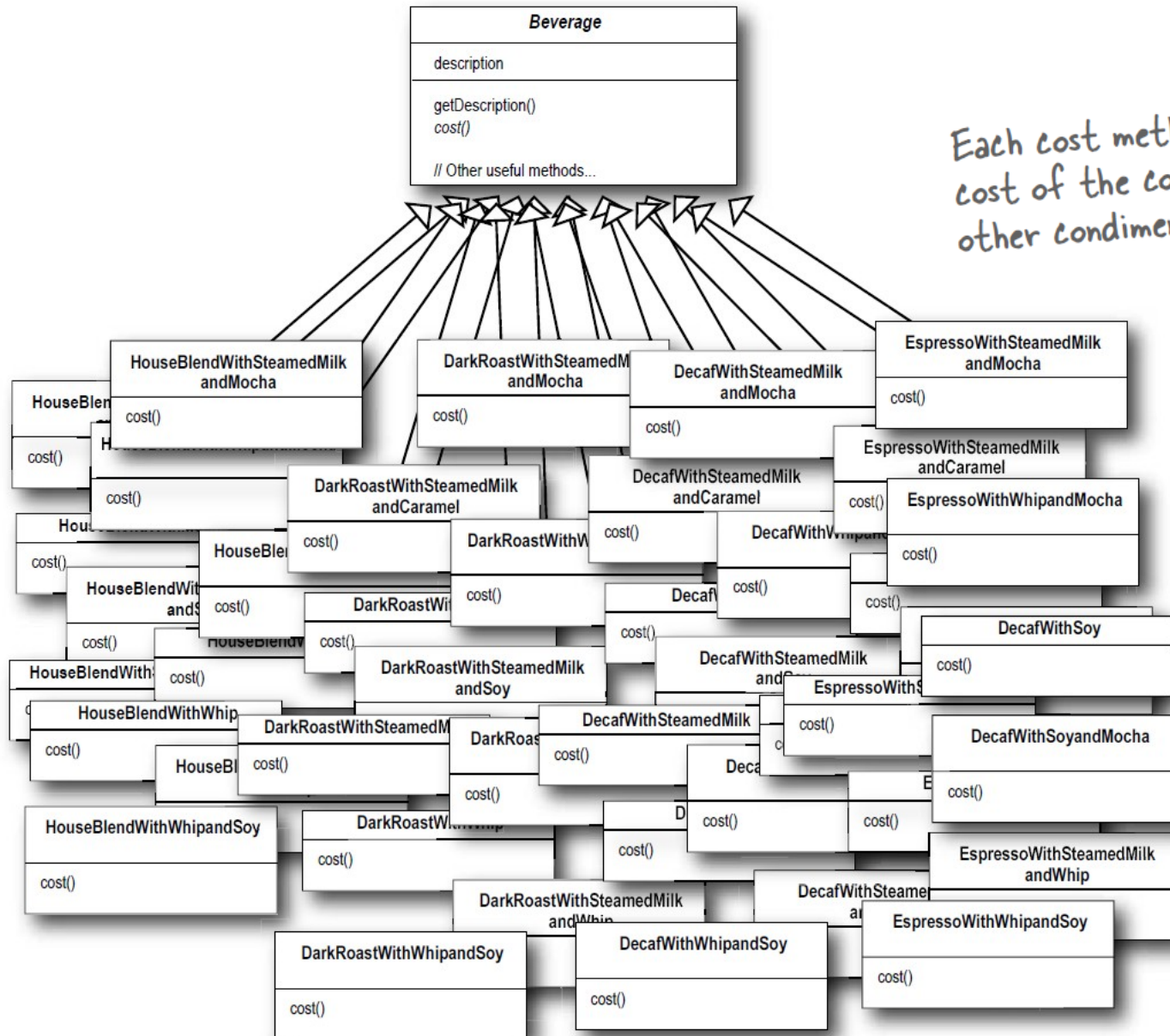
---

- Kaffeehauskette expandiert enorm
- Bestellsystem muss aktualisiert werden, um dem Getränkeangebot zu entsprechen
  - Zum Kaffee kann man natürlich zusätzliche Zutaten bestellen
  - Heiße Milch, Soja, Schokolade, Milchschaum ... und monatlich werden es mehr!
- Bestehendes System wurde ursprünglich folgendermaßen designt ...

# Ursprüngliches Design



# Neues Design ?



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

# Der Wartungs Albtraum

---

- Was wenn der Milchpreis steigt?
- Was wenn es eine neue Karamell-Garnierung gibt?

**Mal abgesehen von der Verletzung unserer schon bekannten Design-Prinzipien!**



# Einfacher Ansatz



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?

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.

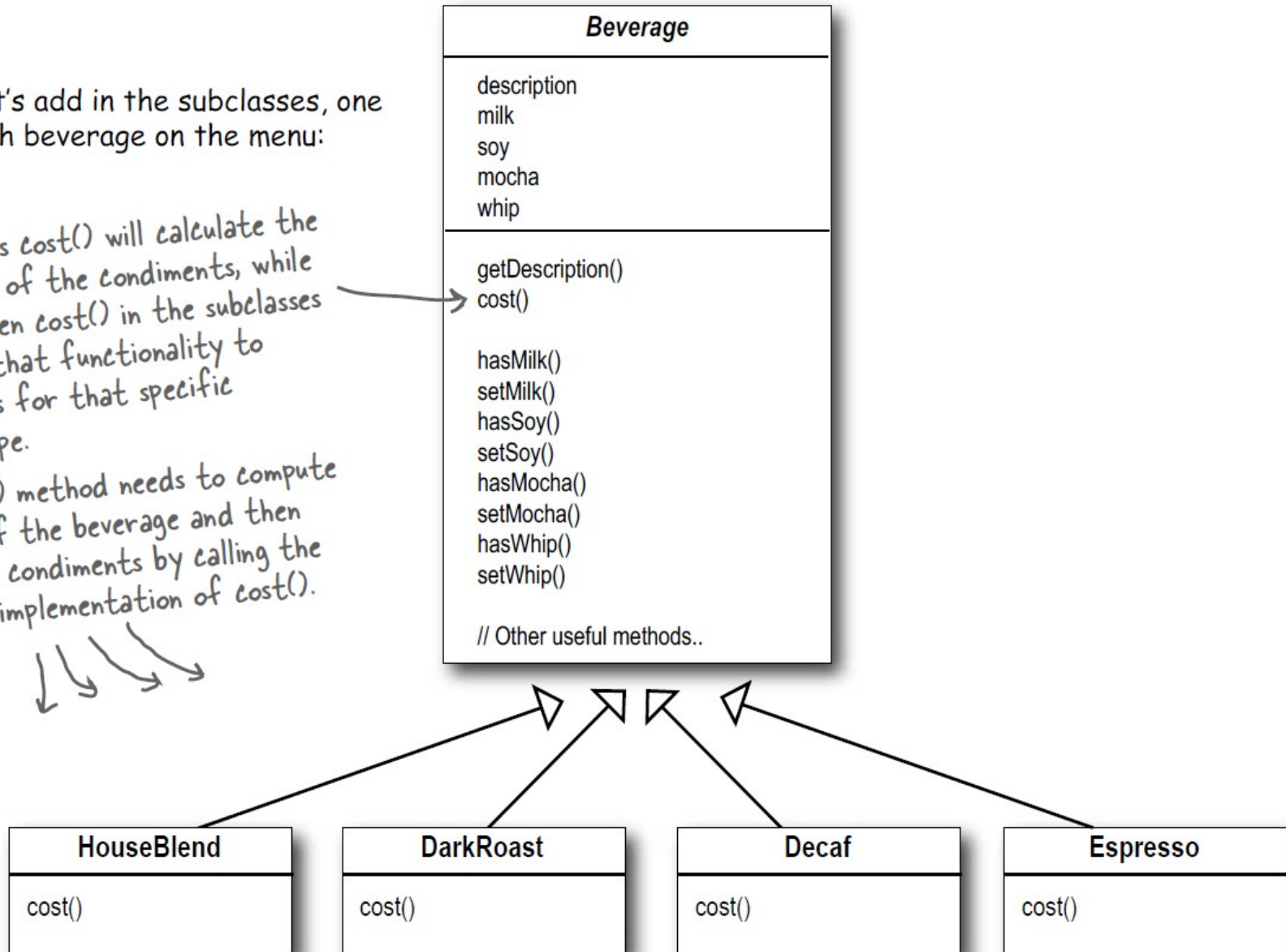
These get and set the boolean values for the condiments.

# Einfacher Ansatz !

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()`.





# Einfacher Ansatz ?

---



See, five  
classes total. This is  
definitely the way to go.

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.



# ~~Einfacher Ansatz~~

---

- Preisänderungen bei den Zutaten könnten Codeänderungen erforderlich machen
- Neue Zutaten würden Veränderungen der Superklasse und der preis()-Methode von Nöten machen
- Was geschieht bei neuen Getränken? Und was wenn dort die Zutaten unpassend sind (z.B. Eistee ↔ Milchschaum?)
- Was wenn ein Kunde die doppelte Menge an Schokolade wünscht?

# Veränderung von bestehendem Code und die Sache mit der Vererbung

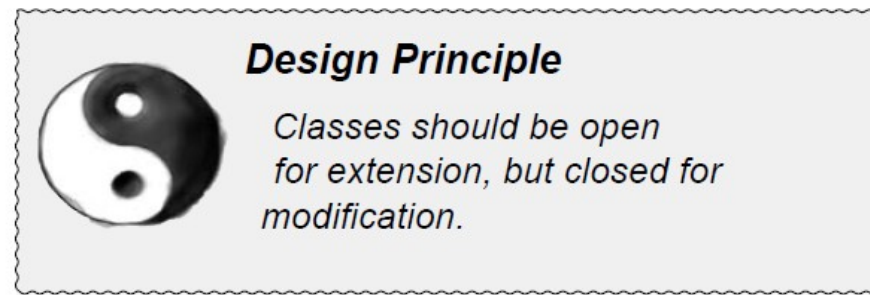
---

- Verhalten durch Vererbung ist statisch zur Kompilierzeit fixiert
  - alle Unterklassen müssen das gleiche Verhalten erben
  - Besser: Verhalten eines Objekts durch Komposition erweitern → dynamisch zur Laufzeit möglich!
- Bestehenden Code zu ändern birgt Gefahr Fehler einzufügen bzw. unerwünschte Nebeneffekte zu verursachen
  - Deswegen: **Finger weg von Codeänderungen!**

# Das Offen/Geschlossen Prinzip

---

- Eines der wichtigsten Entwurfsprinzipien:



- Erweitere bestehende Klassen mit jedem neuen Verhalten, welches dir gefällt.
- Schreibe Deine eigenen Erweiterungen!
- Es wurde sehr viel Zeit investiert, den bestehenden Code richtig und fehlerfrei zu machen: Finger weg!
- Code muss für Änderungen geschlossen bleiben!

# Das Offen/Geschlossen Prinzip

---

- Wie kann ein Entwurf beides zugleich sein?
  - Denke an das Observer Pattern ...
- Es gibt Techniken, Code zu erweitern, ohne ihn direkt zu modifizieren
- Das Offen/Geschlossen Prinzip ÜBERALL anzuwenden ist
  - Verschwendung,
  - unnötig
  - und kann zu komplexem, schwer verständlichem Code führen!

# Das Decorator Pattern

---

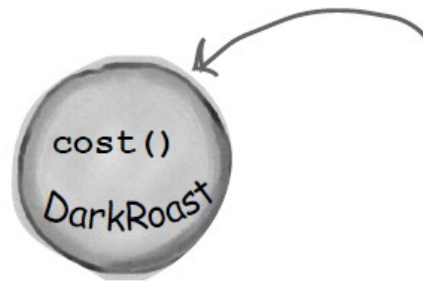
1. Wir nehmen ein *DunkleRöstung*-Objekt,
2. dekorieren es mit einem *Schoko*-Objekt,
3. dekorieren es mit einem *Milchschaum*-Objekt,
4. rufen die Methode *preis()* auf und stützen uns auf Delegation, um den Preis für die Zutaten hinzuzufügen.



# Das Decorator Pattern

---

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

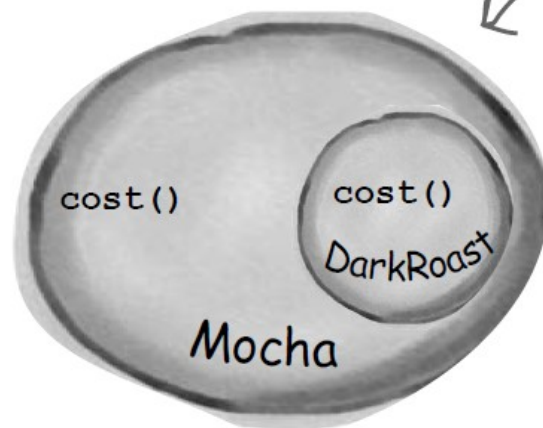


Remember that DarkRoast inherits from Beverage and has a `cost()` method that computes the cost of the drink.

# Das Decorator Pattern

---

- ② The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.



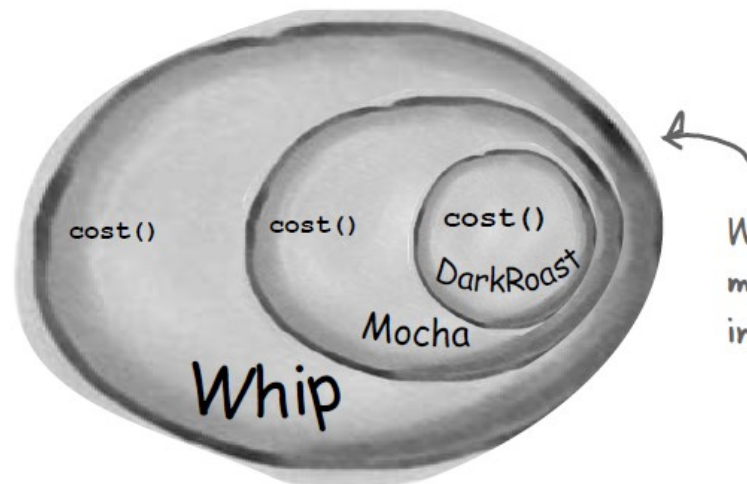
The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type..)

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

# Das Decorator Pattern

---

- ③ The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.

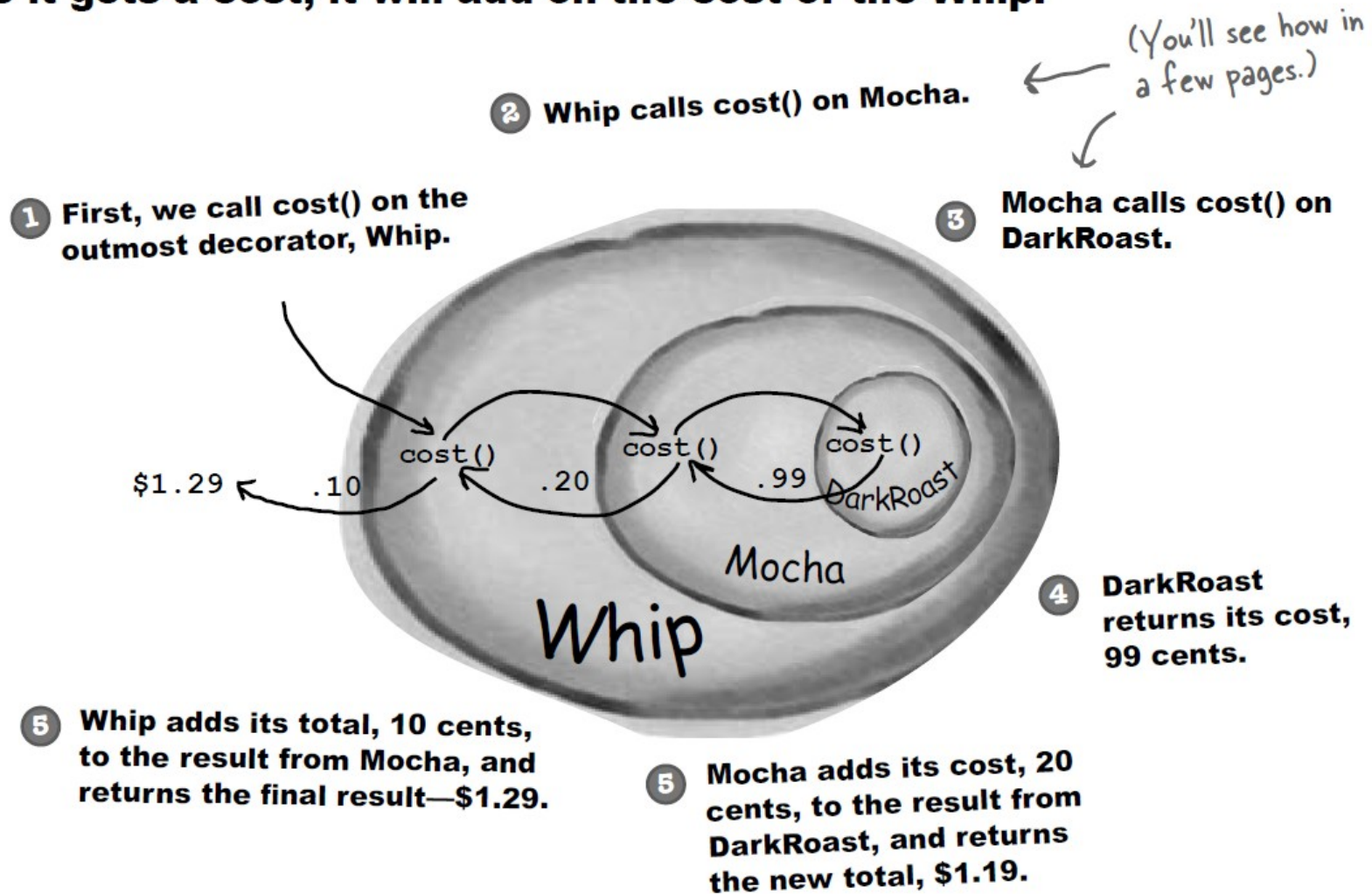


Whip is a decorator, so it also mirrors DarkRoast's type and includes a `cost()` method.

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

# Das Decorator Pattern

- 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.



# Das wissen wir bisher

---

- Dekorierer haben den gleichen Supertyp wie die dekorierten Objekte
- Man kann ein oder mehrer Objekte verwenden, um ein Objekt einzupacken
- Da der Dekorierer vom selben Typ ist wie das dekorierte Objekt, kann man es an die Stelle des ursprünglichen Objekts herumreichen
- Der Dekorierer fügt sein eigenes Verhalten hinzu, bevor und/oder nach dem Aufruf an das dekorierte Objekt
- Objekte können jederzeit dekoriert werden → dynamisch zur Laufzeit



# Definiere, das Decorator Pattern

**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. It extends Component.

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

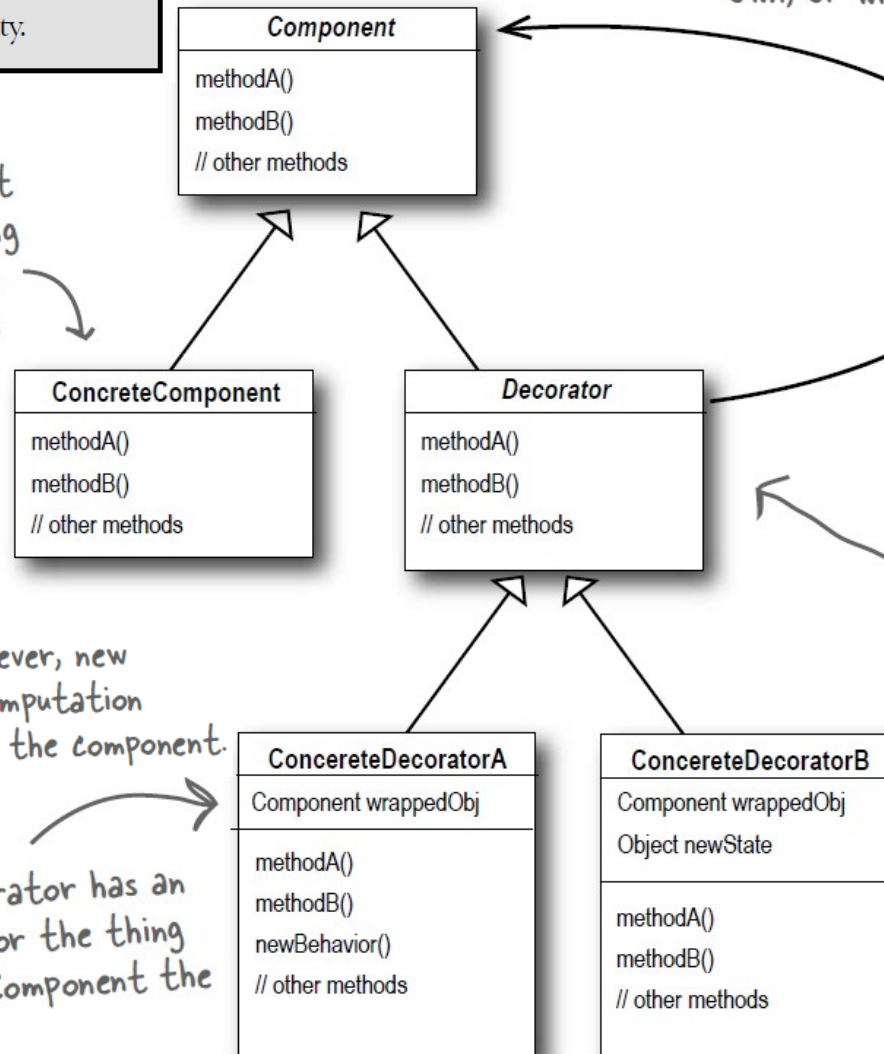
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.

Decorators implement the same interface or abstract class as the component they are going to decorate.

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

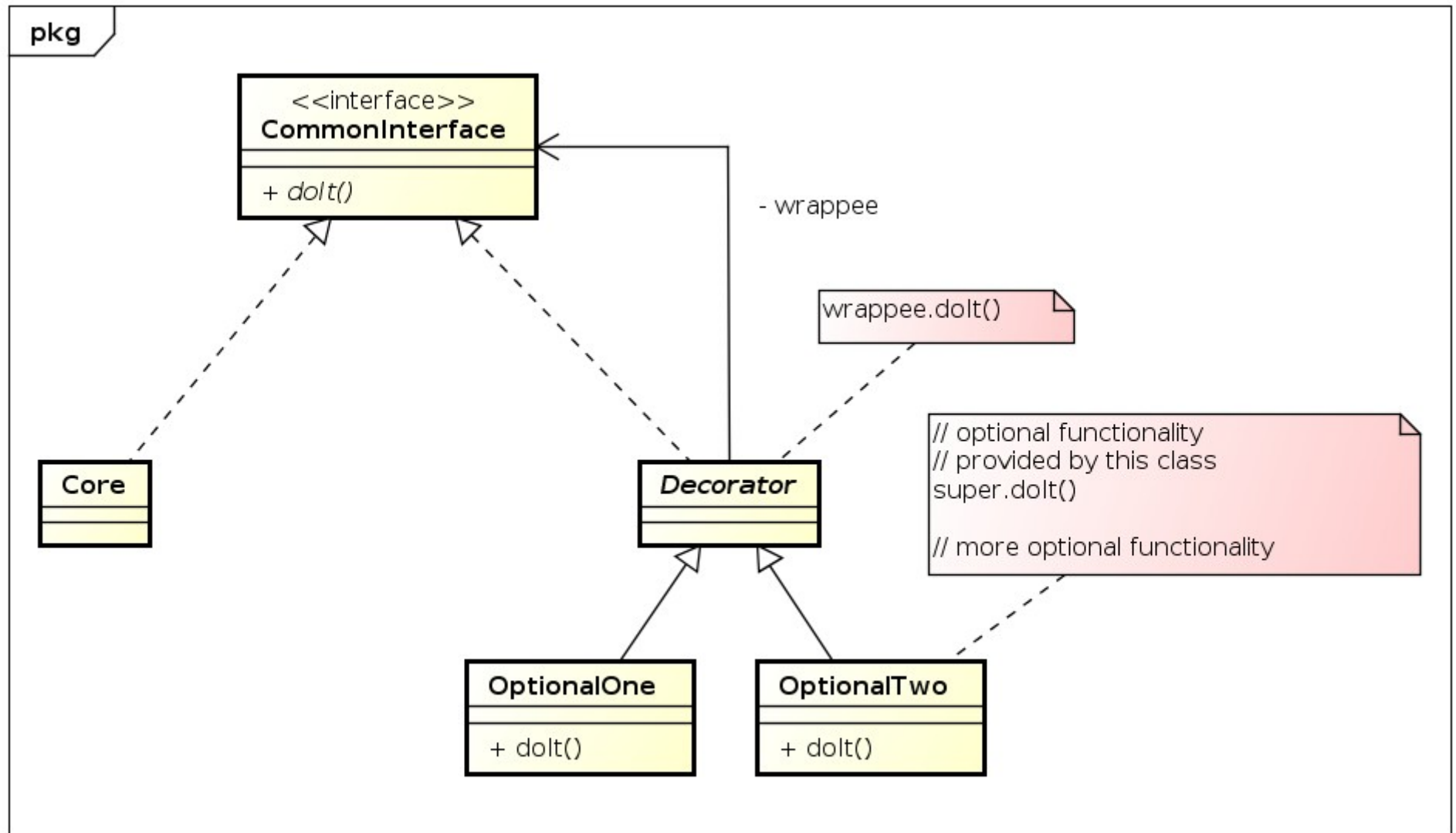
The ConcreteDecorator has an instance variable for the thing it decorate (the Component the Decorator wraps).

Decorators can extend the state of the component.

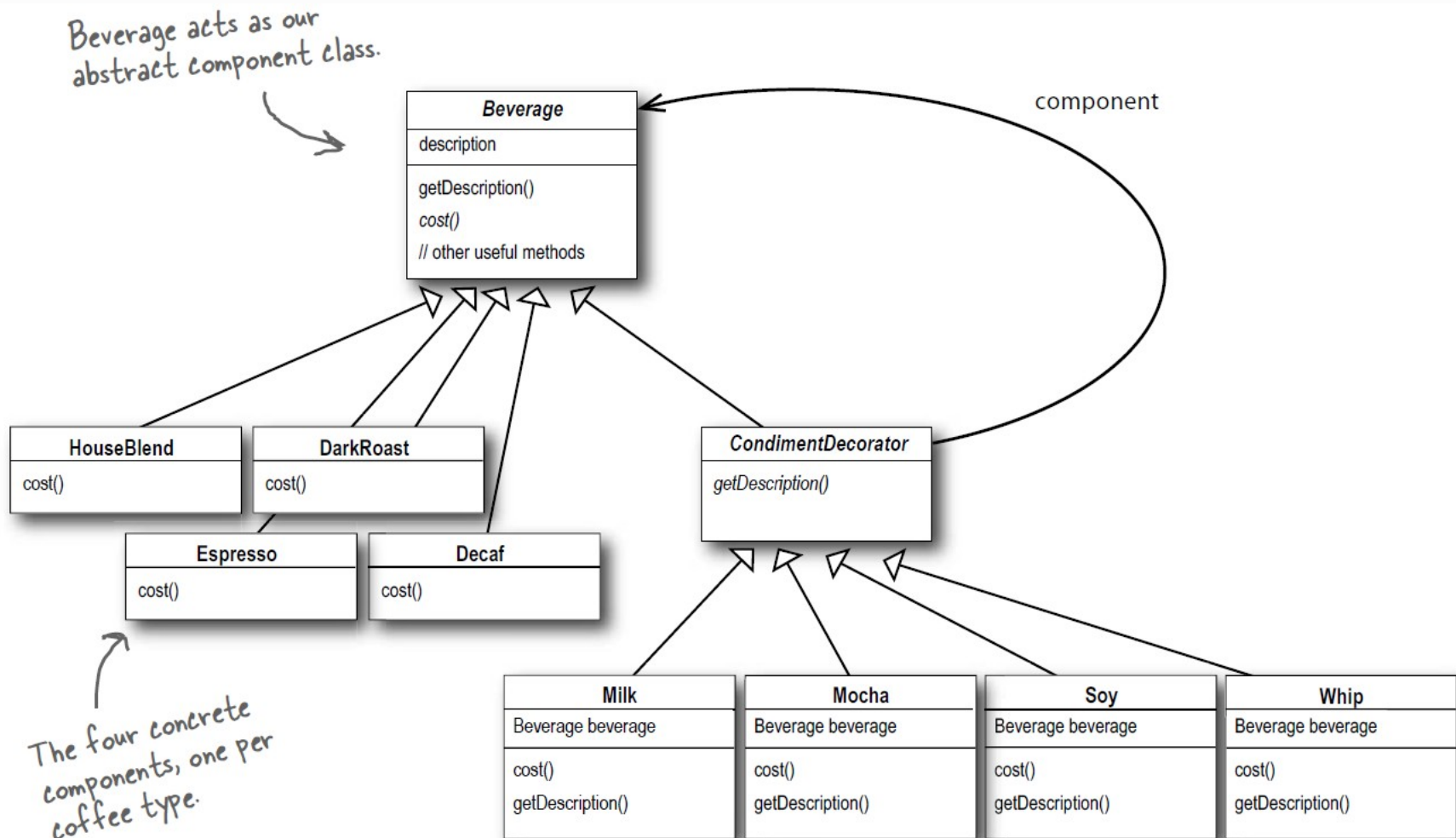




# Definiere, das Decorator Pattern



# Decorator Pattern und Sternback-Kaffee



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

# Implementierung „Getränk“ und „ZutatDekorierer“

```
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.

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

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

We're also going to require that the condiment decorators all reimplement the `getDescription()` method. Again, we'll see why in a sec...

# Implementierung „Espresso“ und „Hausmischung“

```
public class Espresso extends Beverage {
```

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

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

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

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.

```
public class HouseBlend extends Beverage {
```

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

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

Okay, here's another Beverage. All we do is set the appropriate description, "House Blend Coffee," and then return the correct cost: 89¢.

## Starbuzz Coffee

<u>Coffees</u>	
House Blend	.89
Dark Roast	.99
Decaf	1.05
Espresso	1.99

<u>Condiments</u>	
Steamed Milk	.10
Mocha	.20
Soy	.15
Whip	.10



# Implementierung der Zutaten

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();  
    }  
}
```

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.

# Servieren wir Kaffee ...

```
public class StarbuzzCoffee {
```

```
    public static void main(String args[]) {  
        Beverage beverage = new Espresso();  
        System.out.println(beverage.getDescription()  
            + " $" + beverage.cost());
```

Order up an espresso, no condiments  
and print its description and cost.

```
        Beverage beverage2 = new DarkRoast();
```

Make a DarkRoast object.  
Wrap it with a Mocha.

```
        beverage2 = new Mocha(beverage2);
```

```
        beverage2 = new Mocha(beverage2);
```

Wrap it in a second Mocha.

```
        beverage2 = new Whip(beverage2);
```

Wrap it in a Whip.

```
        System.out.println(beverage2.getDescription()  
            + " $" + beverage2.cost());
```

```
        Beverage beverage3 = new HouseBlend();
```

```
        beverage3 = new Soy(beverage3);
```

```
        beverage3 = new Mocha(beverage3);
```

```
        beverage3 = new Whip(beverage3);
```

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

```
        System.out.println(beverage3.getDescription()  
            + " $" + beverage3.cost());
```

```
    }
```

```
}
```

File Edit Window Help CloudsInMyCoffee

```
% java StarbuzzCoffee
```

```
Espresso $1.99
```

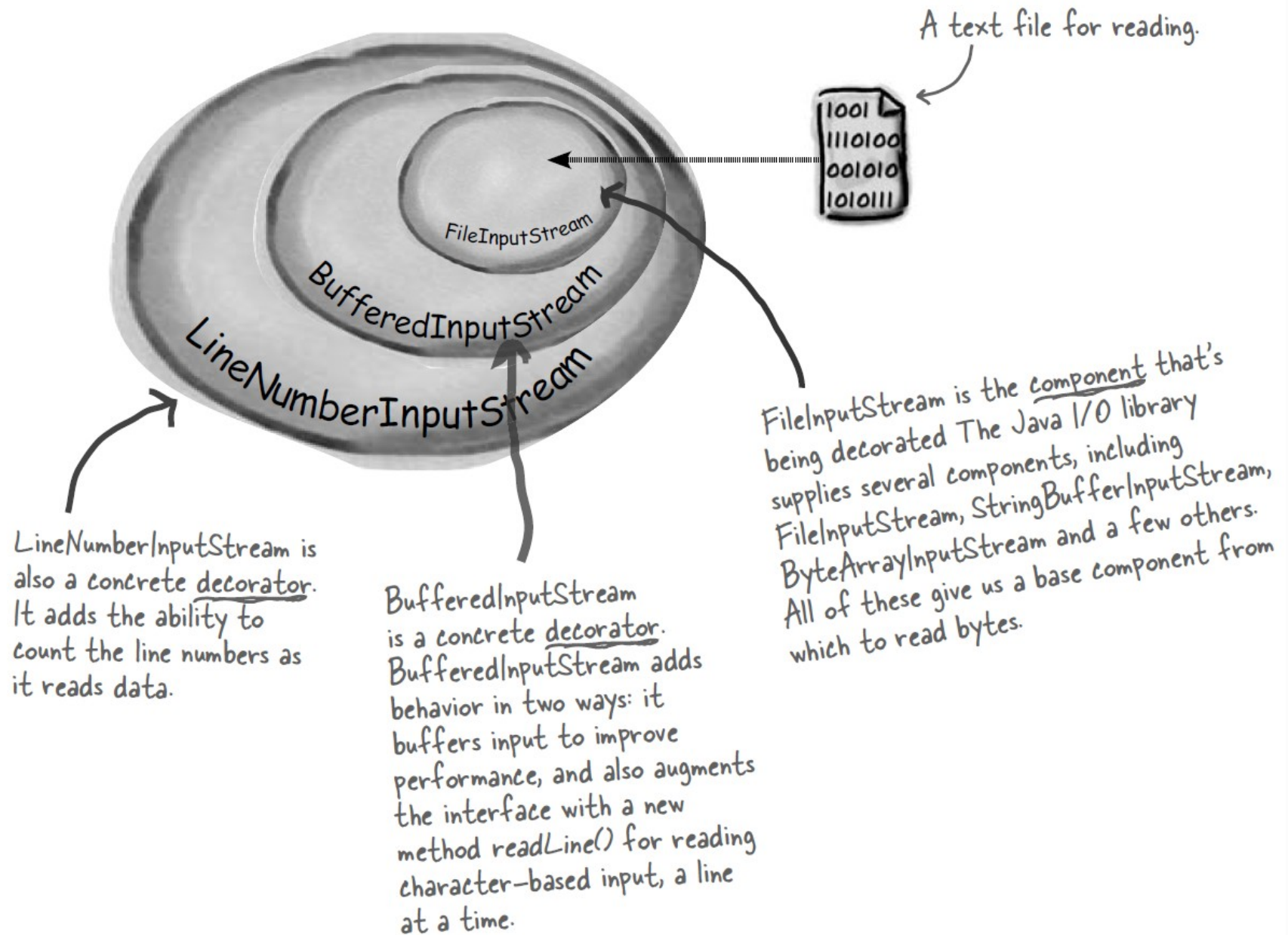
```
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
```

```
House Blend Coffee, Soy, Mocha, Whip $1.34
```

```
%
```




# Dekorierer aus der Praxis



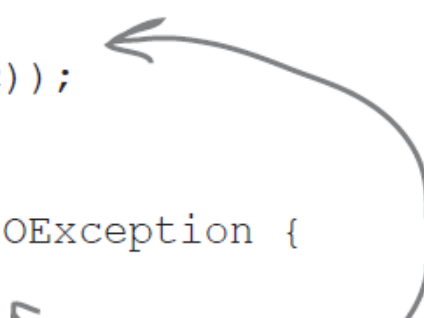
# Eigenen Java I/O Dekorierer schreiben

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

First, extend the `FilterInputStream`, the  
abstract decorator for all `InputStream`s.



```
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.

# Eigenen Java I/O Dekorierer schreiben

```
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();  
        }  
    }  
}
```

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

Set up the `FileInputStream` and decorate it, first with a `BufferedInputStream` and then our brand new `LowerCaseInputStream` filter.

I know the Decorator Pattern therefore I RULE!

test.txt file

You need to make this file.

Give it a spin:

```
File Edit Window Help DecoratorsRule  
% java InputTest  
i know the decorator pattern therefore i rule!  
%
```

# Zusammengefasst

---

- Dekorierer-Klassen werden verwendet um konkrete Komponenten einzupacken
- Dekorierer-Klassen spiegeln den Typ der Komponente wider, die sie dekorieren
- Dekorierer ändern das Verhalten der Komponenten, indem sie vor und/oder nach Methodenaufrufen auf der Komponente neue Funktionalitäten hinzufügen
- Man kann eine Komponente mit einer beliebigen Anzahl von Dekorieren einpacken
- Dekorierer sind für die Clients der Komponente üblicherweise transparent
- Dekorierer können zu vielen kleinen Objekten führen  
→ übermäßige Verwendung → unübersichtlicher Code !

# Neue Werkzeuge in der Design-Toolbox

---

## OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

## OO Basics

action  
ulation  
orphism  
ance

## OO Patterns

Strat  
encap  
inter  
vary

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?

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.

*to be continued ...*