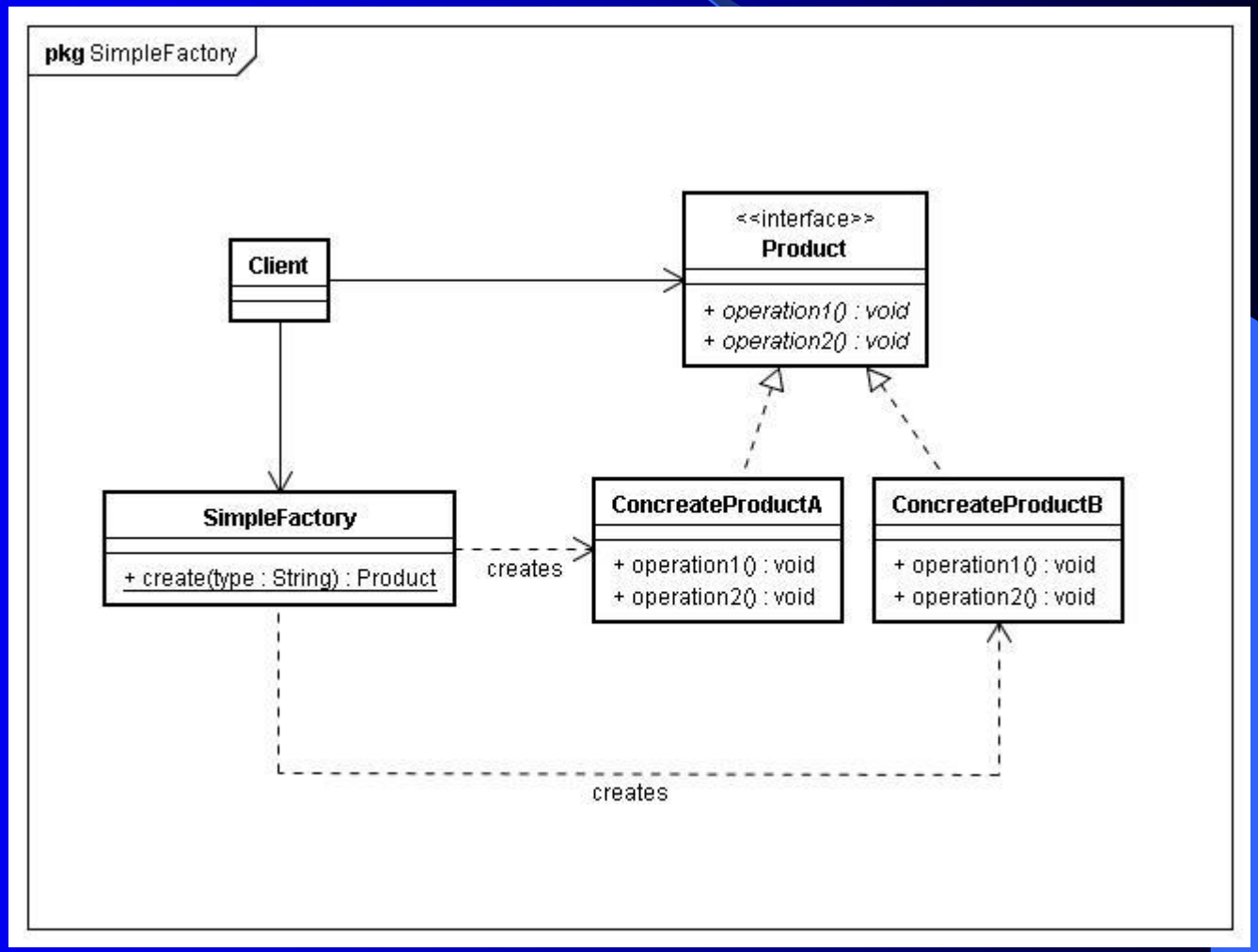
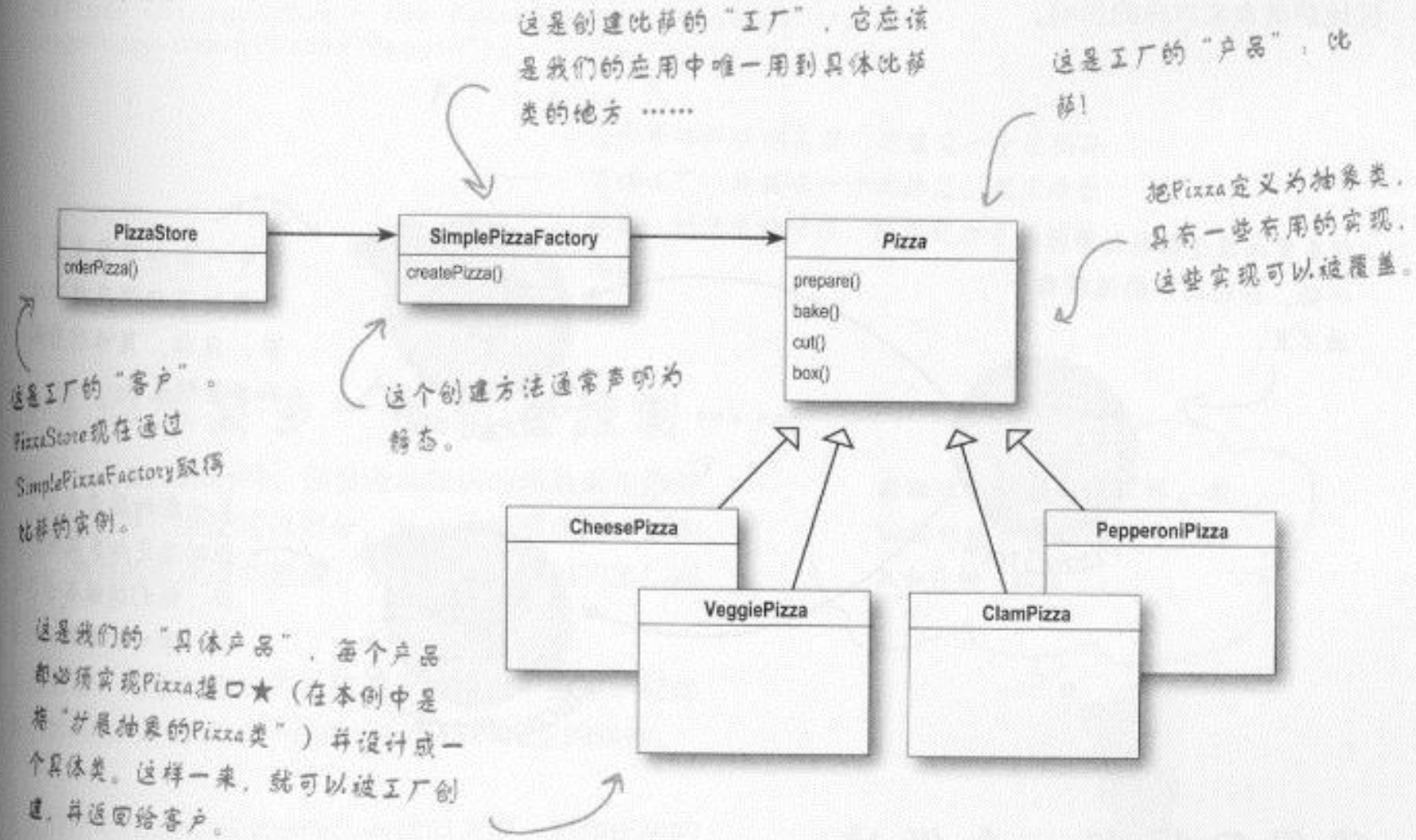
回顾：创建型模式

- 简单工厂
- 工厂方法
- 抽象工厂

简单工厂

- 通过接收的参数的不同来返回不同的对象实例。





谢谢简单工厂来为我们暖身。接下来登场的是两个重量级的模式，它们都是工厂。但是别担心，未来还有更多的比萨！

★ 再提醒一次：在设计模式中，所谓的“实现一个接口”并“不一定”表示“写一个类，并利用implement关键词来实现某个Java接口”。“实现一个接口”泛指“实现某个超类型（可以是类或接口）的某个方法”。

建立一个简单比萨工厂

我们将从工厂的实现开始。定义一个类来封装所有pizza对象创建的行为……

这是我们新建的一个类，SimplePizzaFactory，它只有一个职责，就是为它的客户端生产pizza

首先我们在工厂中定义了一个createPizza()方法，其所有客户端都是通过这个方法来实例化新的对象。

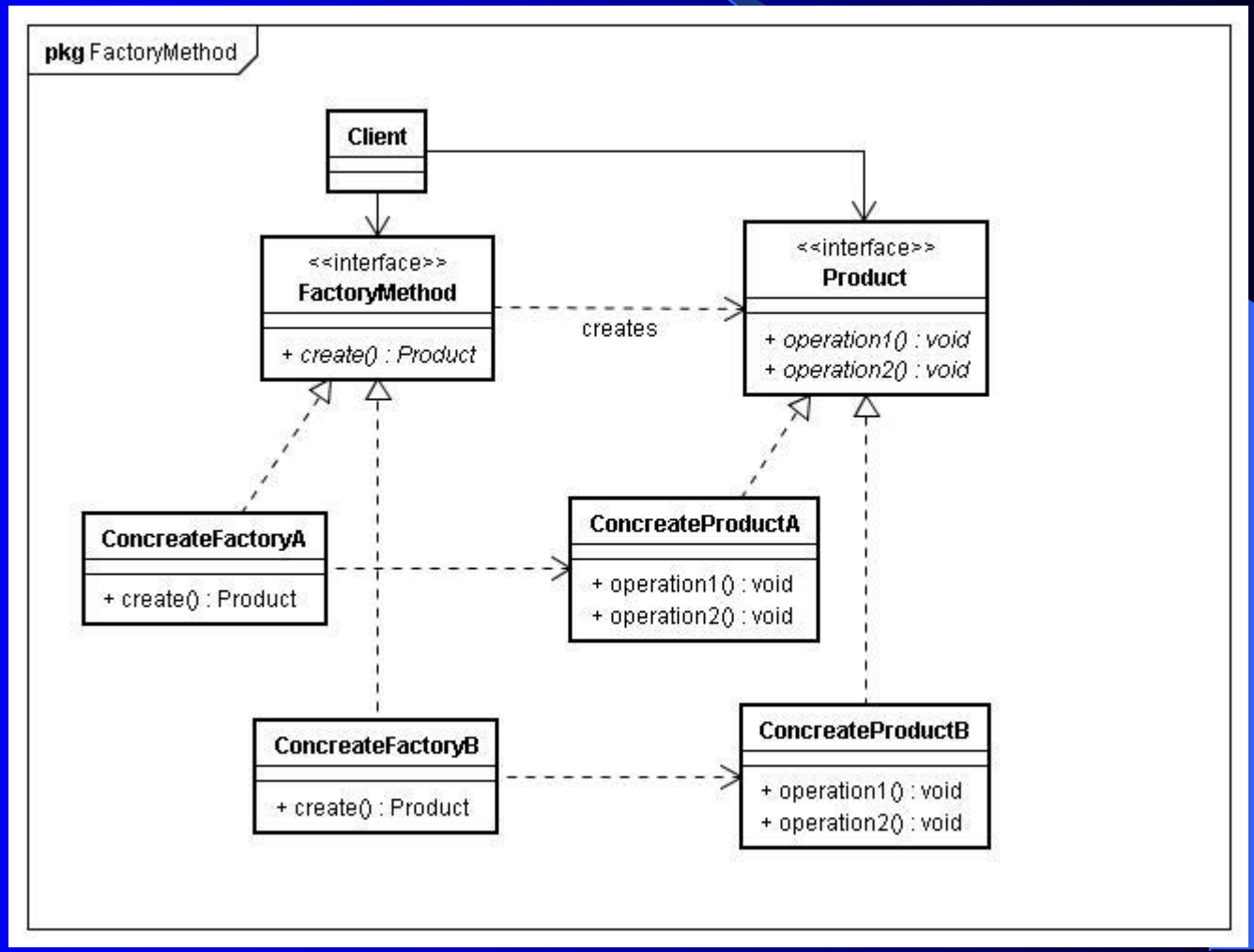
```
public class SimplePizzaFactory {
        public Pizza createPizza(String type){
                Pizza pizza = null;

                if(type.equals("chesse")){
                        pizza = new CheesePizza();
                } else if (type.equals("pepperoni")){
                        pizza = new PepperoniPizza();
                } else if (type.equals("clam")){
                        pizza = new ClamPizza();
                } else if (type.equals("veggie")){
                        pizza = new VeggiePizza();
                }
                return pizza;
        }
} ///:~
```

这些代码是我们从orderPizza()方法中移出来的。

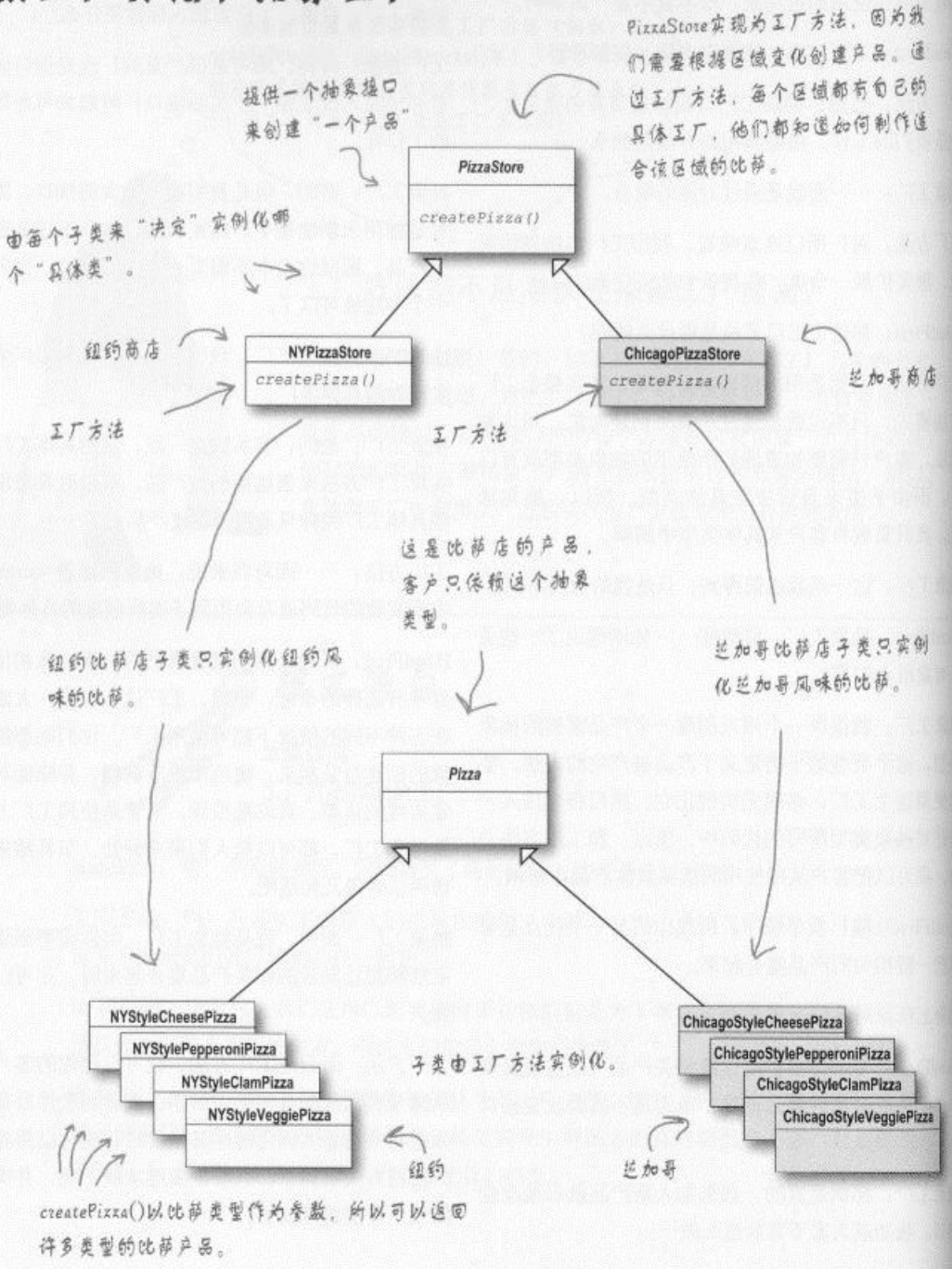
和原来的代码一样，这些代码通过pizza类型而参数化的。

工厂方法

- 工厂方法是针对每一种产品提供一个工厂类。通过不同的工厂实例来创建不同的产品实例。



比较工厂方法和抽象工厂



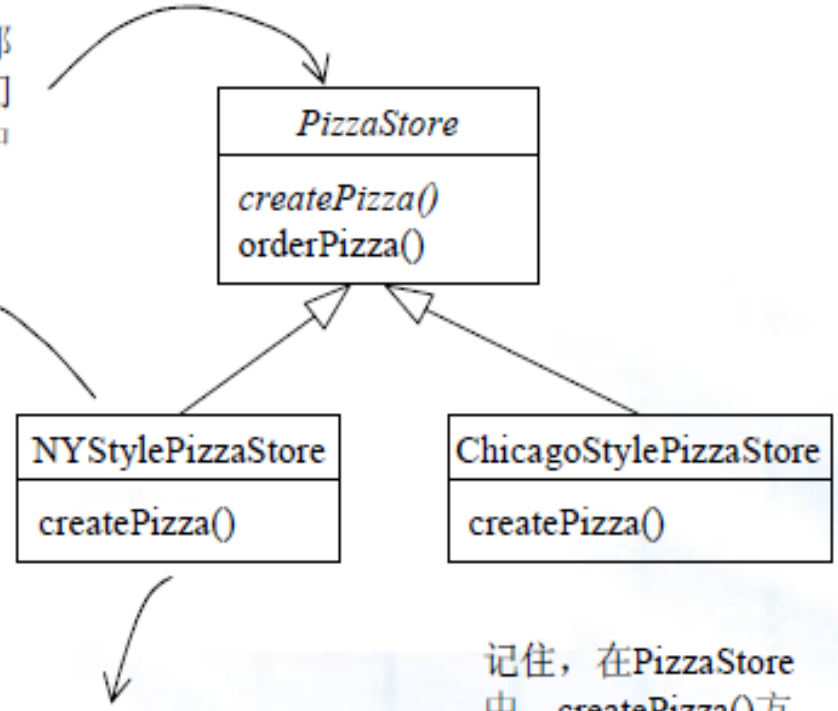
允许子类自己来决定如何做.....

记住，PizzaStore系统中已经有一个很好的订单系统，即orderPizza()方法，我们可以确认这个方法在所有的专卖店都是一致的。

变化的部分是Pizza的制造部分-----纽约风味需要薄面包皮，芝加哥需要厚面包皮等等。我们打算把这些变化的部分都封装在createPizza()方法中，并使之可以制作合适的pizza种类。我们允许每个PizzaStore的子类实现具体的createPizza(),那么我我们将拥有一些PizzaStore具体子类，每一个子类都可以制造一类不同的pizza。

每个子类都重载了createPizza()方法，并且每个子类都使用PizzaStore中定义的orderPizza()方法时，如果我们希望子类都强制执行orderPizza()方法，在PizzaStore中我们可以把这个方法定义为final。

```
public Pizza createPizza (type) {
    if (type.equals("cheese")){
        pizza = new NYStyleCheesePizza();
        }
    else if (type.equals("pepperoni")){
        pizza = new NYStylePepperoniPizza();
        }
    else if (type.equals("clam")){
        pizza = new NYStyleClamPizza();
        }
    else if (type.equals("vggie")) {
        pizza =new NYStyleVeggiePizza();
        }
}
```

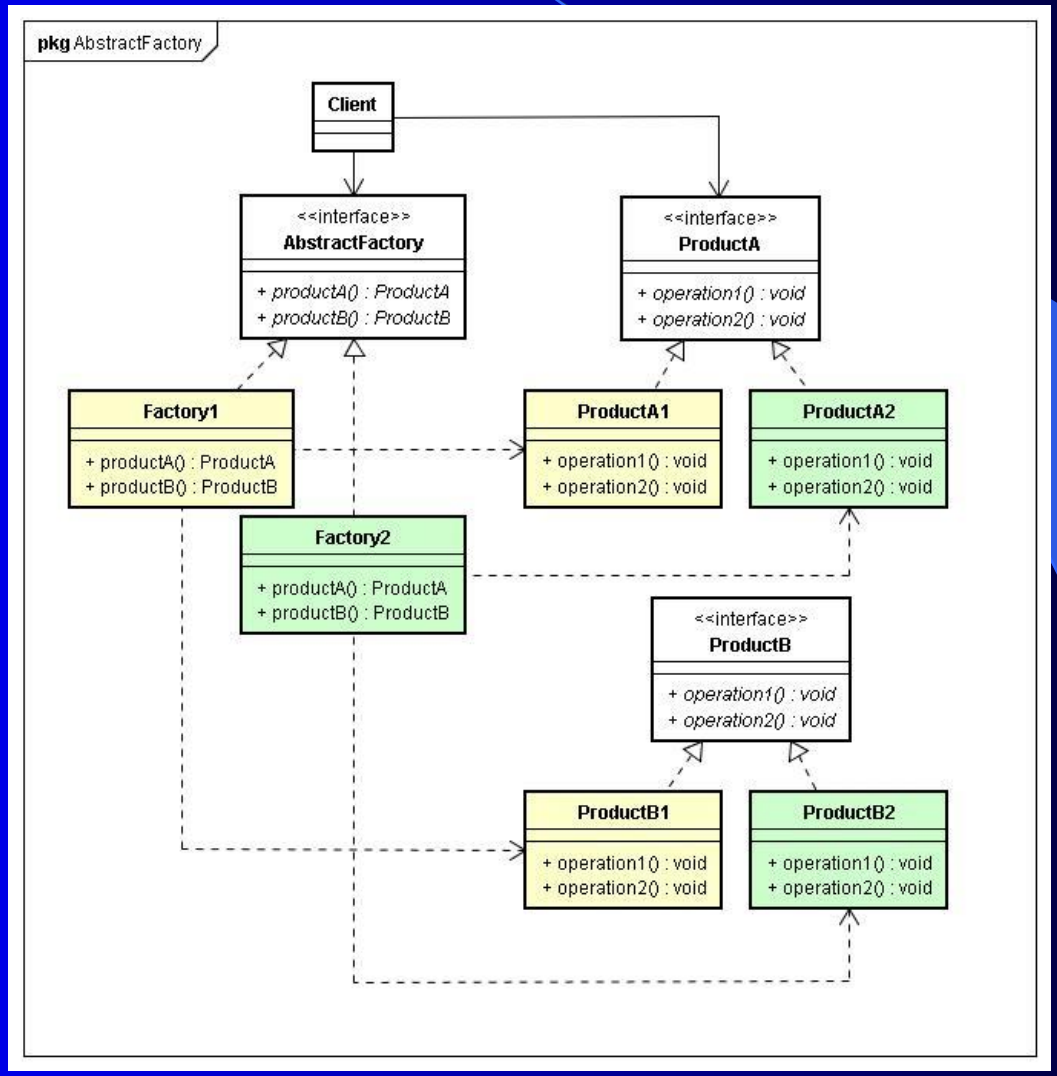


如果某个经销商想为顾客提供纽约风味的pizza，他可以使用这个子类，它有自己的createPizza()方法，可以制造纽约风味的pizza。

记住，在PizzaStore中，createPizza()方法是抽象的，它所有的子类必须实现它。

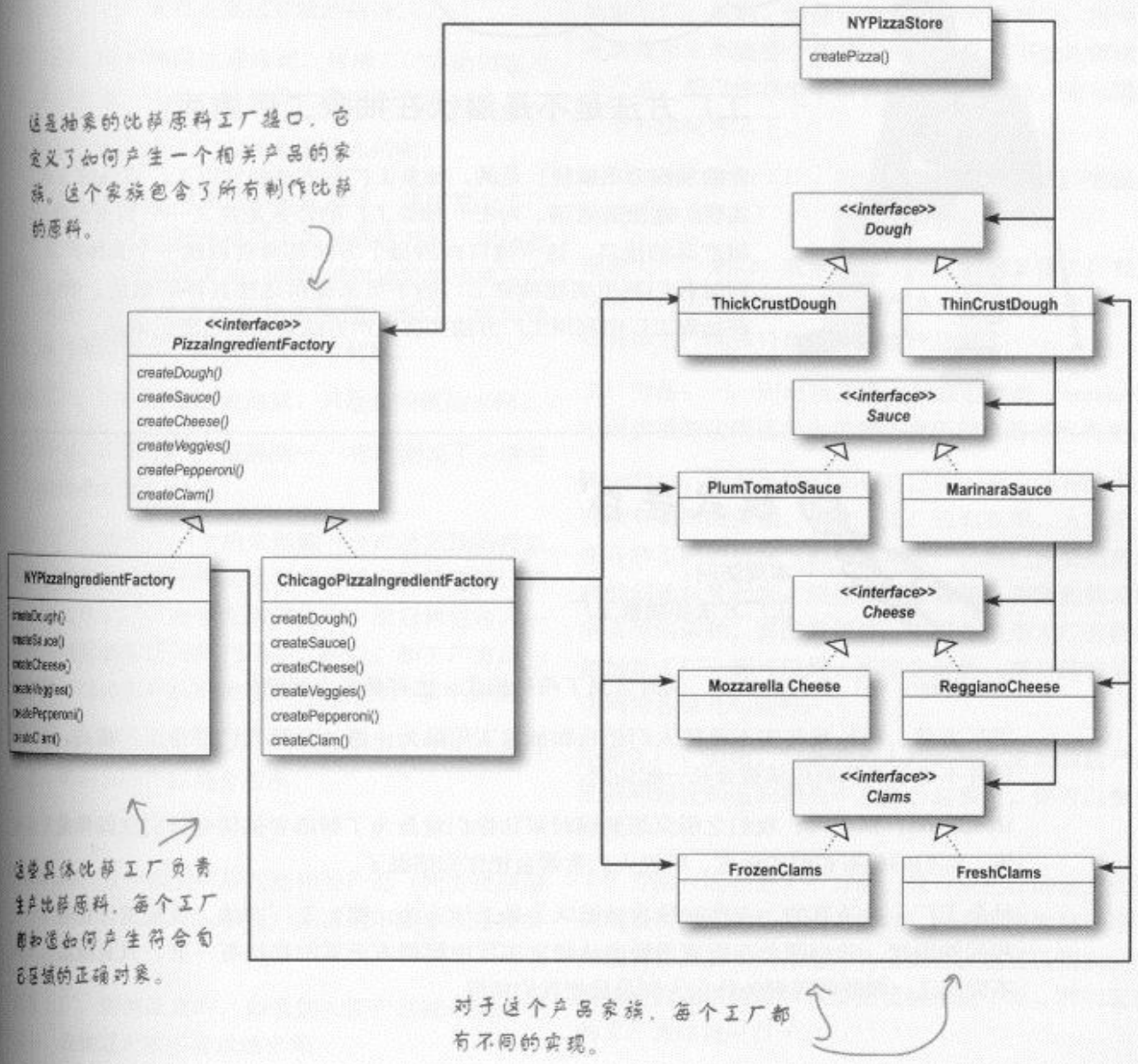
抽象工厂

- 抽象工厂是应对产品族概念的。



这是一张相当复杂的类图，让我们从PizzaStore的观点来看一看它：

比萨店的两个具体实例（NYPizzaStore、ChicagoPizzaStore）是抽象工厂的客户。



接下来，就不一样了，因为我们现在使用了原料工厂

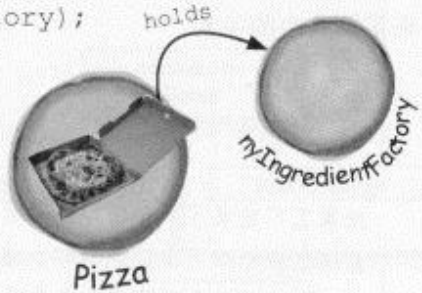
幕后花絮

❹ 当createPizza()方法被调用时，也就开始涉及原料工厂了：

选择原料工厂，接着在PizzaStore中实例化，然后将它传进每个比萨的构造器中。

```
Pizza pizza = new CheesePizza(nyIngredientFactory);
```

创建一个比萨的实例，然后将它和纽约原料工厂结合在一起。



❺ 接下来需要准备比萨。一旦调用了prepare()方法，工厂将被要求准备原料：

prepare()

```
void prepare() {
    dough = factory.createDough();
    sauce = factory.createSauce();
    cheese = factory.createCheese();
}
```

薄饼

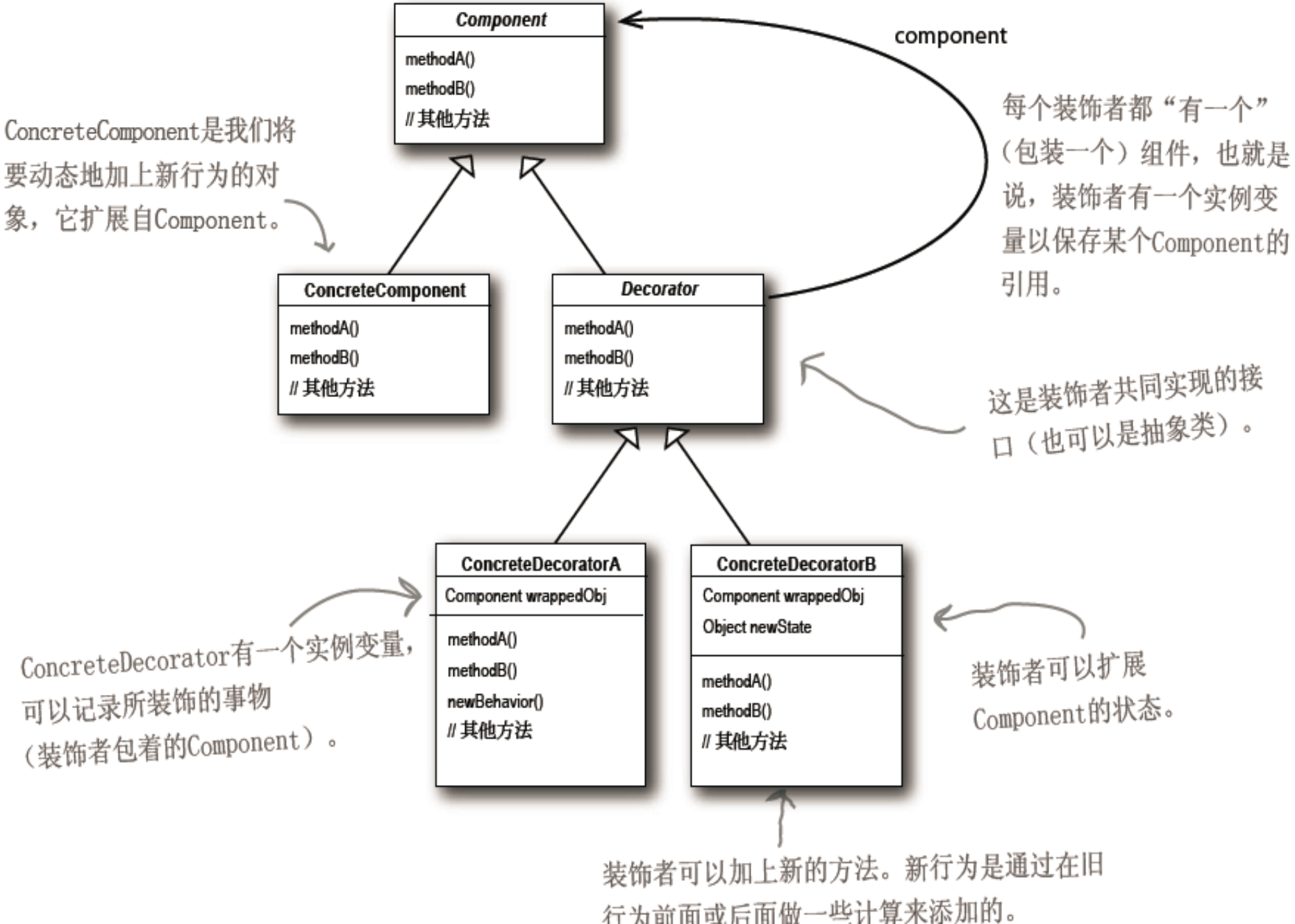
大蒜番茄酱料

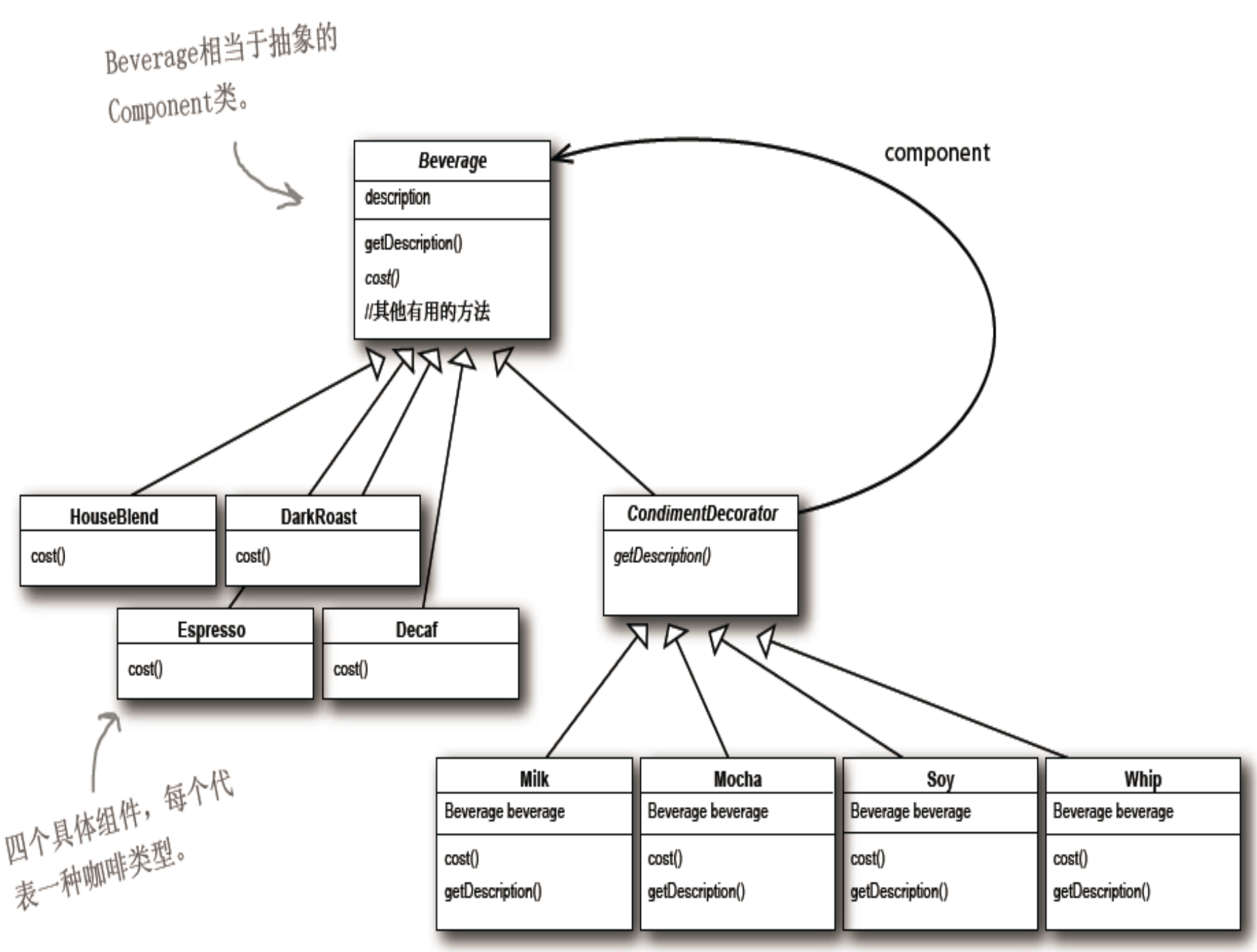
Reggiano干酪

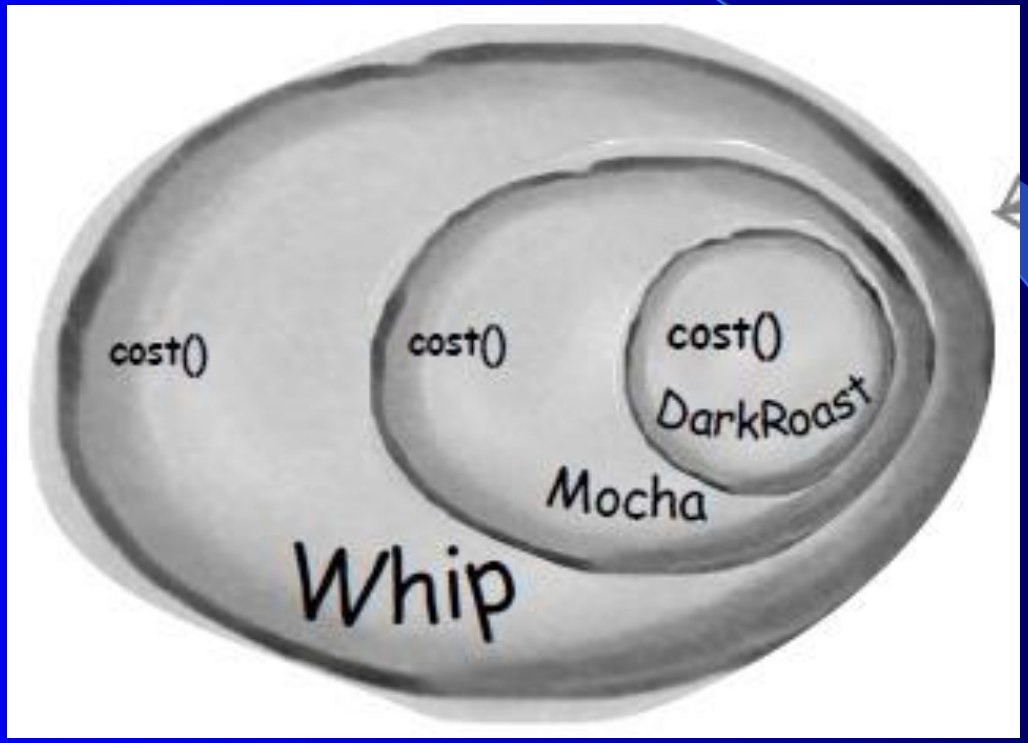
对Ethan的比萨来说，使用了纽约原料工厂，取得了纽约的原料。

回顾：　装饰者模式

每个组件都可以单独使用，或者被装饰者包起来使用。




Beverage相当于抽象的Component类。
Beverage
description
getDescription()
cost()
//其他有用的方法
component
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
CondimentDecorator
getDescription()
四个具体组件，每个代表一种咖啡类型。
Milk
Beverage beverage
cost()
getDescription()
Mocha
Beverage beverage
cost()
getDescription()
Soy
Beverage beverage
cost()
getDescription()
Whip
Beverage beverage
cost()
getDescription()

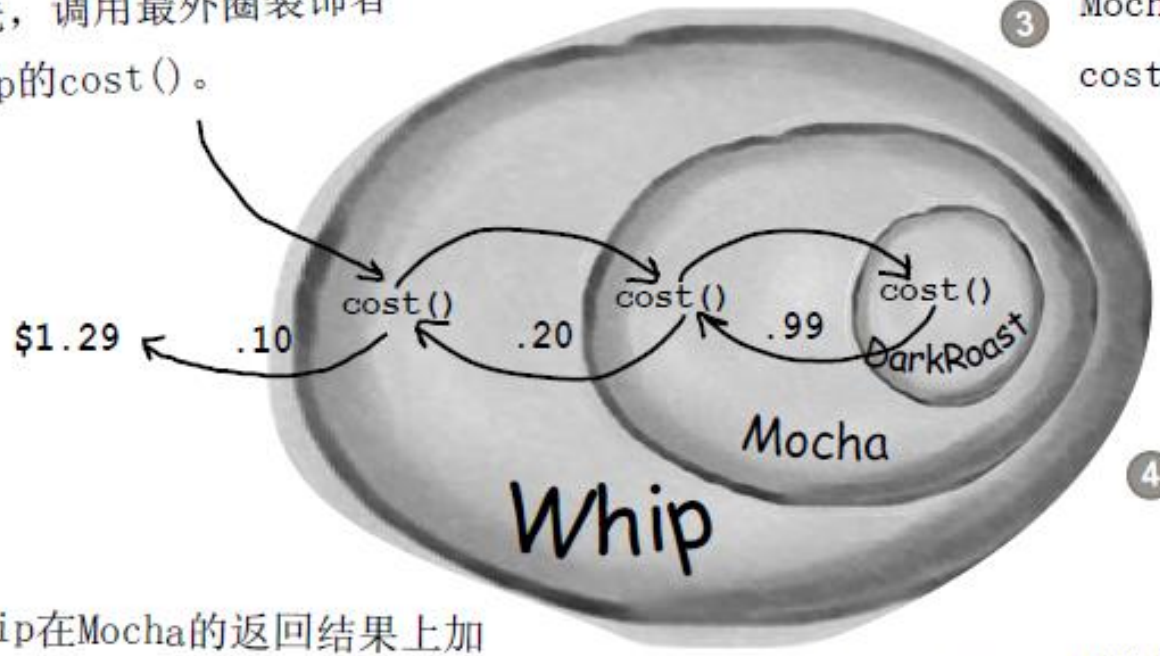

cost()
cost()
cost()
DarkRoast
Mocha
Whip

再过几页，你就会知道这是如何办到的。

2 Whip调用Mocha的cost()。

1 首先，调用最外圈装饰者Whip的cost()。

3 Mocha调用DarkRoast的cost()。



4 DrakRoast返回它的价钱$0.99。

5 Mocha在DarkRoast的结果上，加上自己的价钱$0.20，返回新的价钱$1.19。

6 Whip在Mocha的返回结果上加上自己的价钱$0.10，然后返回最后结果$1.29。

这是用来下订单的一些测试代码★：

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

订一杯Espresso，不需要调料，打印出它的描述与价钱。

制造出一个DarkRoast对象。

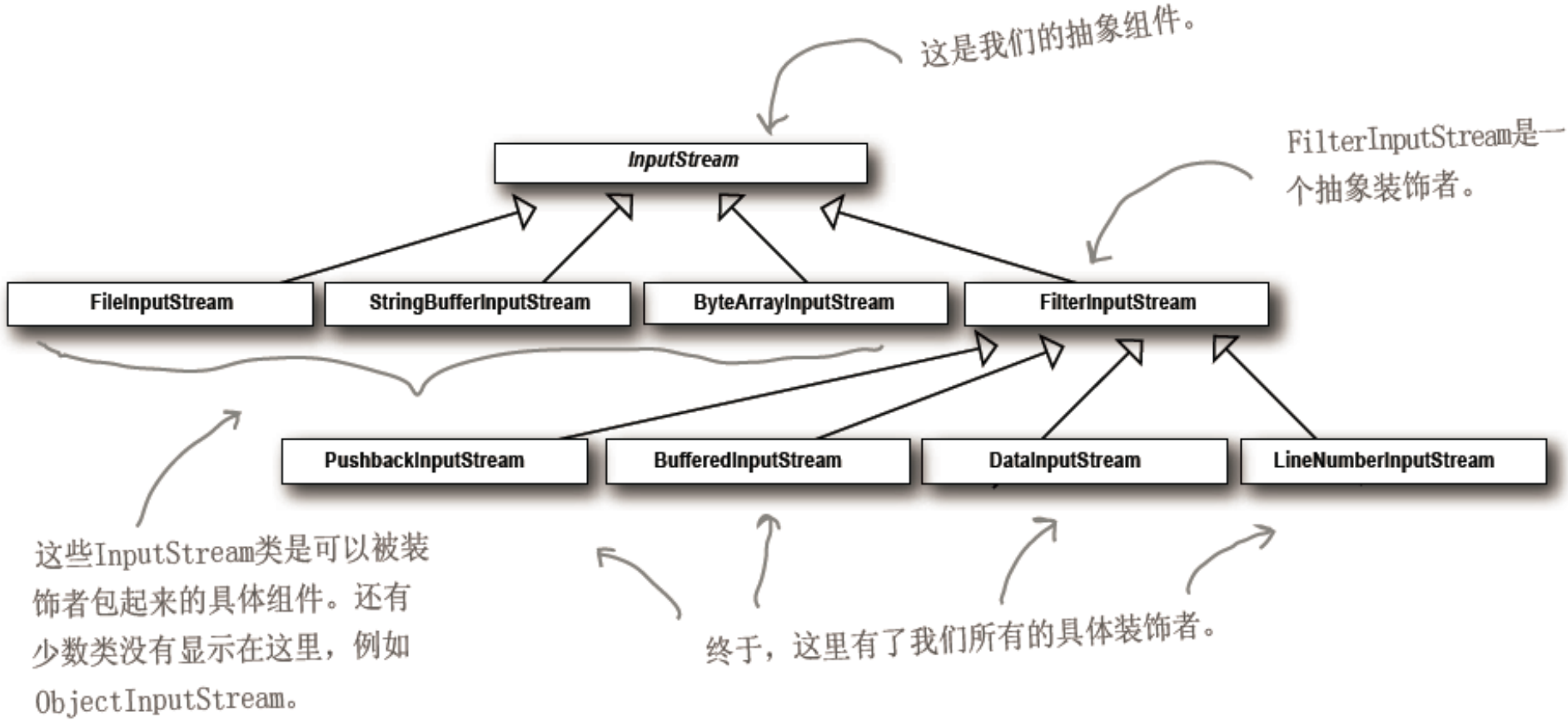
用Mocha装饰它。

用第二个Mocha装饰它。

用Whip装饰它。

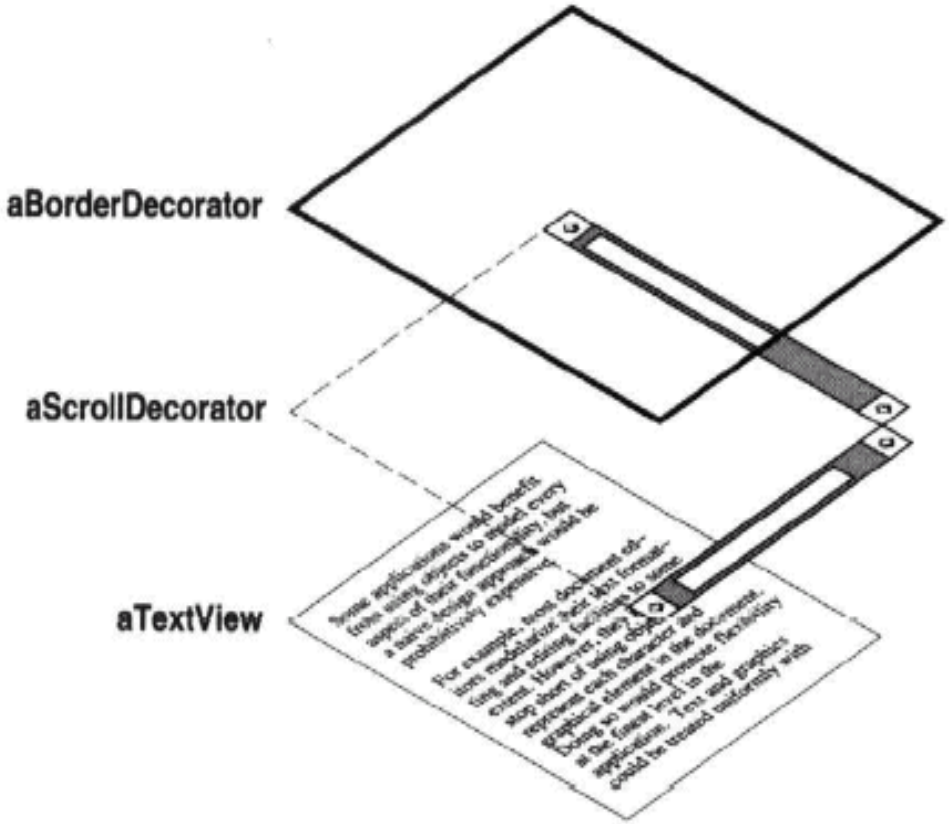
最后，再来一杯调料为豆浆、摩卡、奶泡的HouseBlend咖啡。

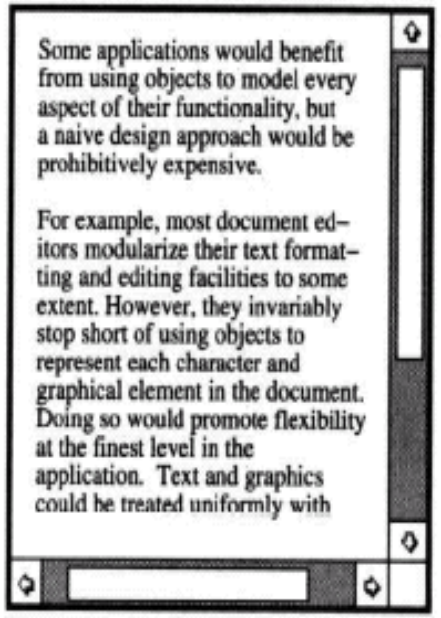
装饰java.io类

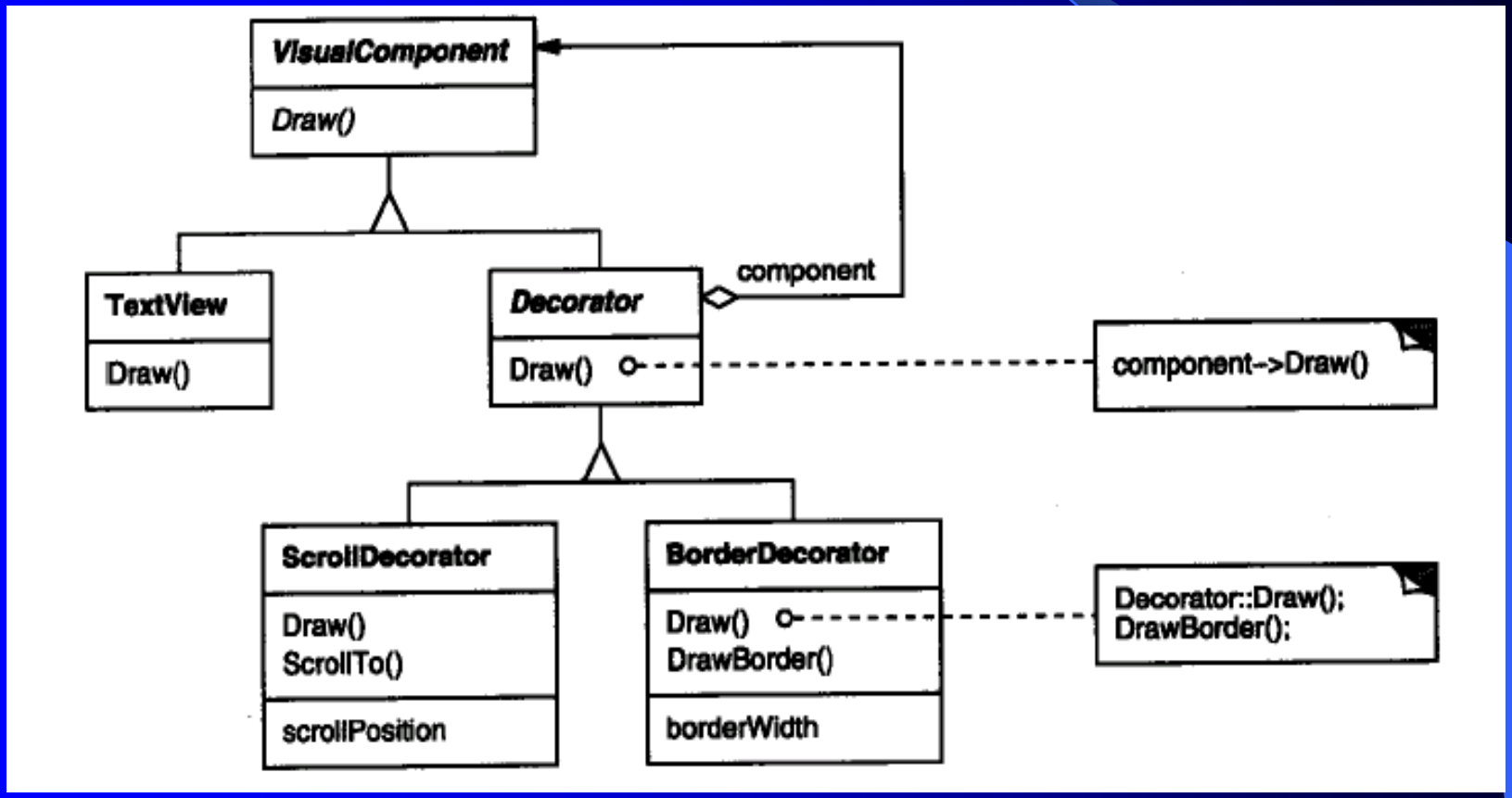

这是我们的抽象组件。
InputStream
FilterInputStream是一个抽象装饰者。
FileInputStream
StringBufferInputStream
ByteArrayInputStream
FilterInputStream
PushbackInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
这些InputStream类是可以被装饰者包起来的具体组件。还有少数类没有显示在这里，例如ObjectInputStream。
终于，这里有了我们所有的具体装饰者。

练习一

- 设计一个文档编辑器

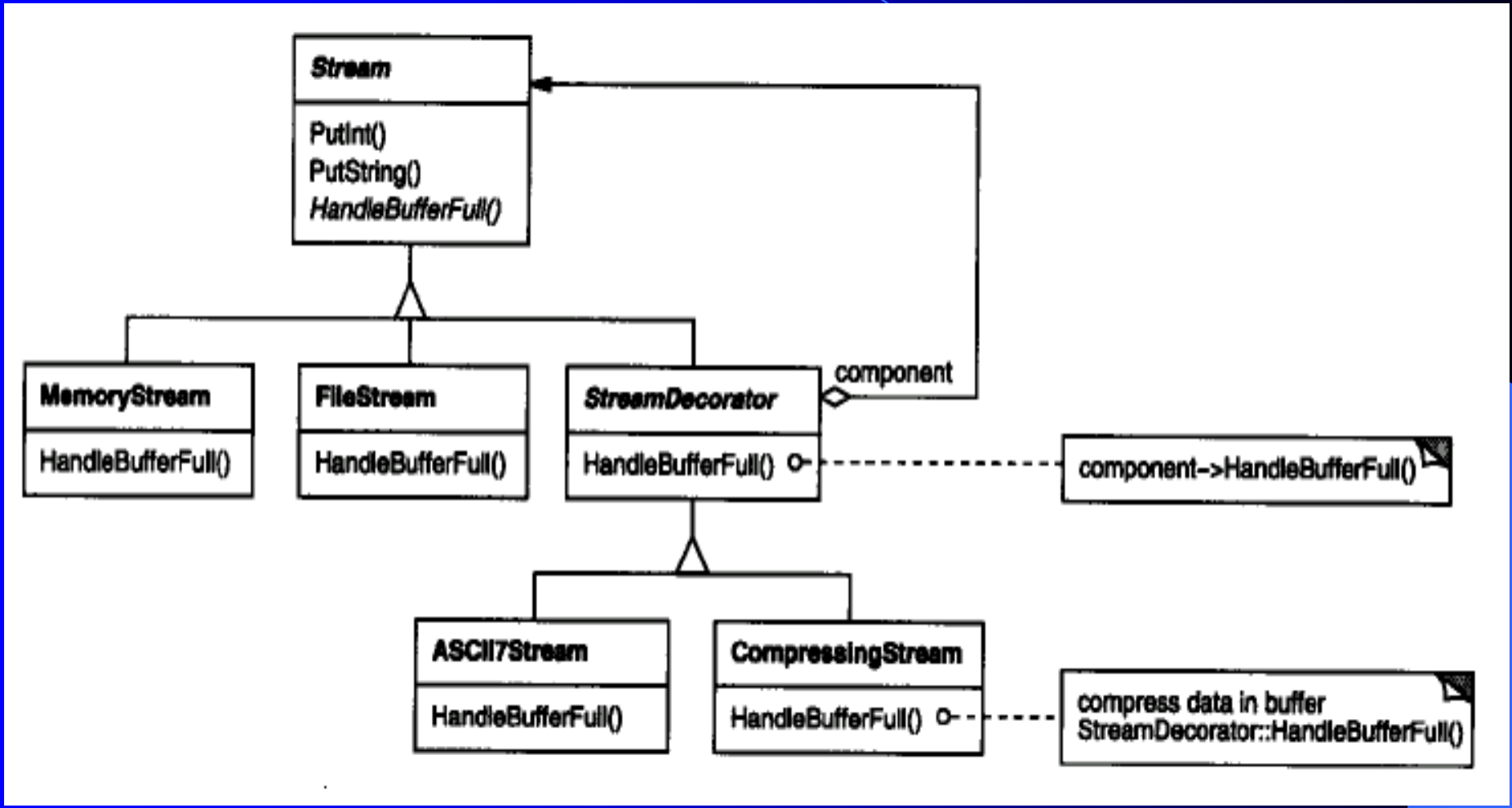

aBorderDecorator
aScrollDecorator
aTextView


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 editors modularize their text formatting 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


VisualComponent
Draw()
TextView
Draw()
Decorator
Draw()
component
component->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();

练习二

- Streams是大多数I/O设备的基础抽象结构，它提供了将**对象**转换成为**字节或字符流**的操作接口，使我们可以将一个对象转变成一个**文件**或**内存**中的字符串，可以在以后恢复使用。
- 用不同的压缩算法对数据流进行压缩。
- 将流数据简化为7位ASCII码字符。


Stream
PutInt()
PutString()
HandleBufferFull()
component
MemoryStream
HandleBufferFull()
FileStream
HandleBufferFull()
StreamDecorator
HandleBufferFull()
component->HandleBufferFull()
ASCII7Stream
HandleBufferFull()
CompressingStream
HandleBufferFull()
compress data in buffer
StreamDecorator::HandleBufferFull()

Singleton模式

```
public class Singleton
{
    private static Singleton _instance = null;
    private Singleton(){}

    public static Singleton getInstance()
    {
        if (null==_instance)
            _instance = new Singleton();

        return _instance;
    }
}
```

保存全类唯一的实例

对客户隐藏构造函数

客户只能从此处获得实例

Singleton Pattern

- 用来创建独一无二的，只能有一个实例的对象的入场券。
- 线程池（threadpool），缓存（cache），日志，打印机或显卡等设备的驱动的对象

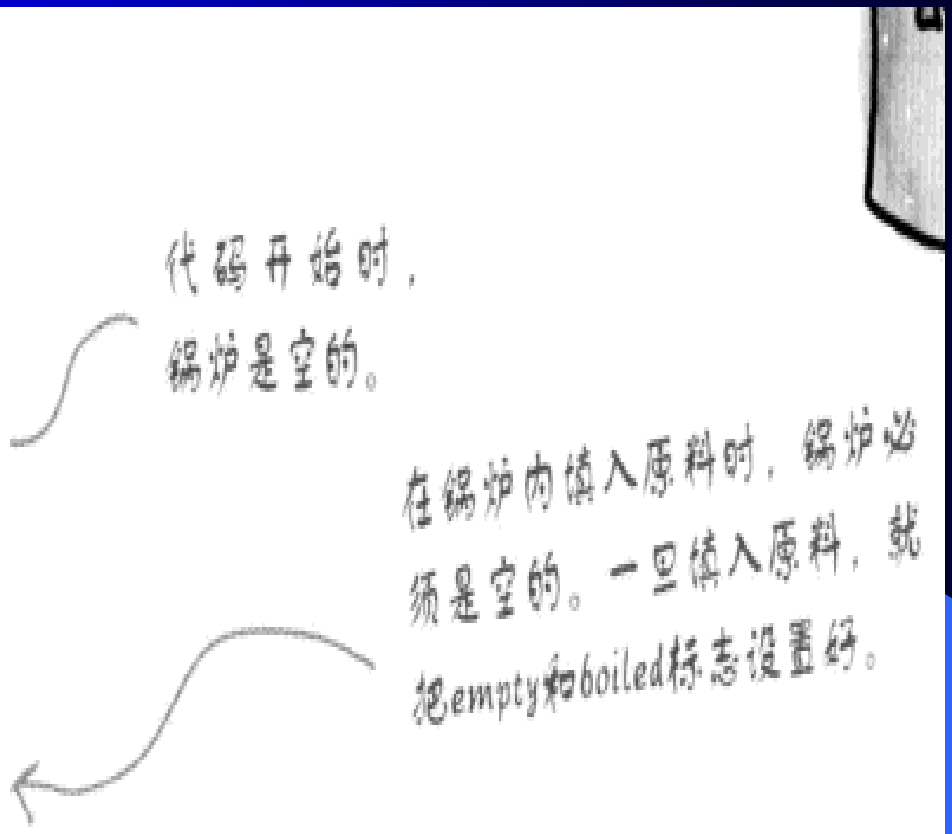
巧克力工厂

- 现代化的巧克力工厂具备计算机控制的巧克力锅炉。
- 锅炉做的事就是把巧克力和牛奶融在一起，然后送到下一个阶段，以制造成巧克力棒。

巧克力锅炉控制器代码

- 需要注意的事情：
- 1，排出未煮沸的混合物。
- 2，已经满了还继续放原料
- 3，或者锅炉内还没放原料就开始空烧。

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

public ChocolateBoiler() {
    empty = true;
    boiled = false;
}


public void fill () {
    if (isEmpty () ) {
        empty = false;
        boiled = false;
        //在锅炉内填满巧克力
        和牛奶的混合物
}
```



```
public void drain () {
  if (!isEmpty () && isBoiled () ) {
       //排出煮沸的巧克力和牛奶
       empty = true;
   }
}

public void boil () {
     if (!isEmpty () && !isBoiled () ) {
          //将炉内物煮沸
          boiled = true;
     }
}

public boolean isEmpty () {
      return empty;
}

public boolean isBoiled () {
       return boiled;
}
```

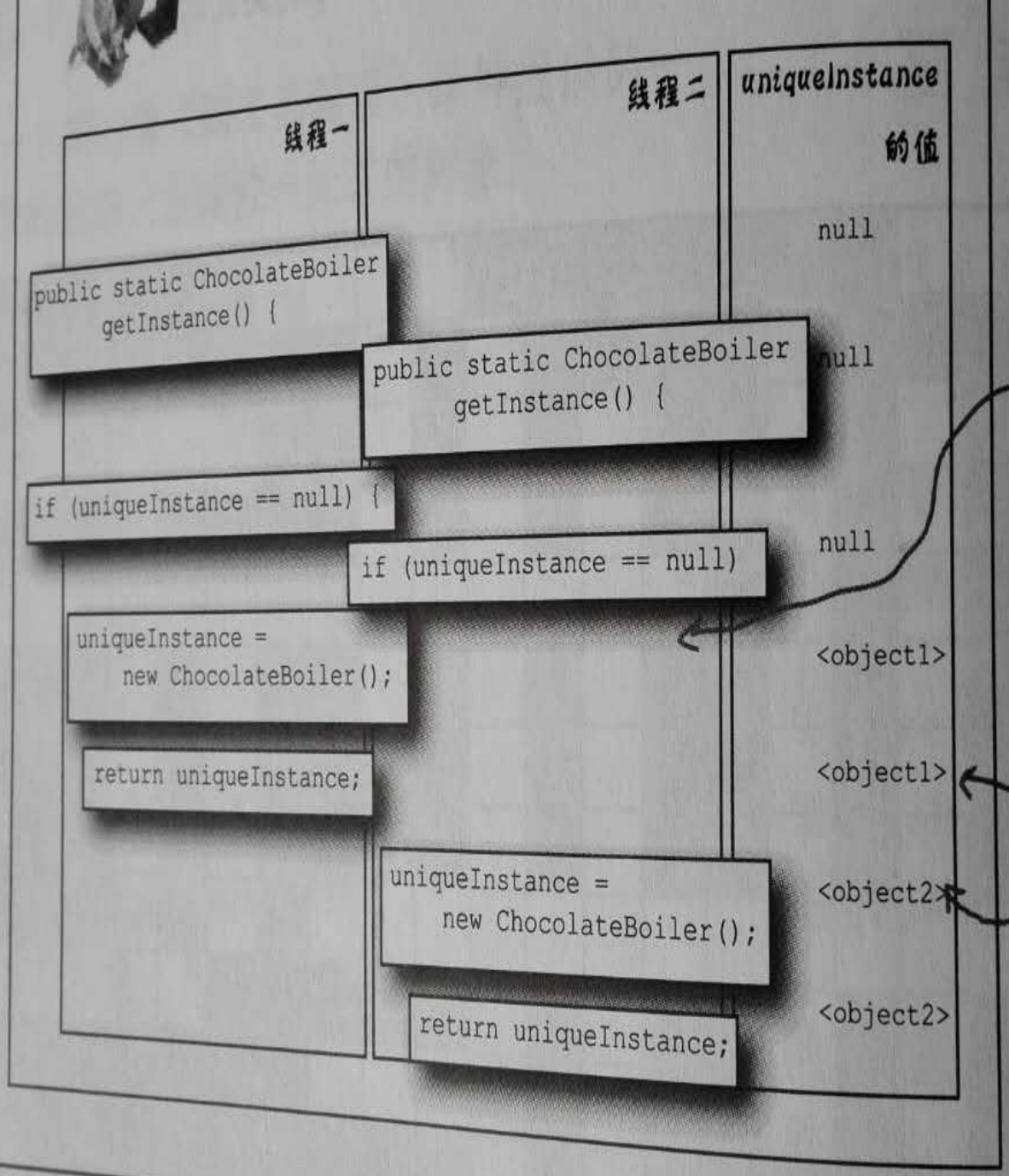
锅炉排出时，必须是满的（不可以是空的）而且是煮过的。排出完毕后，把empty标志设回true。

煮混合物时，锅炉必须是满的，并且是没有煮过的。一旦煮沸后，就把boiled标志设为true。

请给出单件模式的实现

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    ChocolateBoiler () {
        empty = true ;
        boiled = false;
    }

public void fill () {
    if (isEmpty () ) {
        empty = false;
        boiled = false;
        //在锅炉内填充巧克力和牛奶的混合物
    }
}
//其他的部分省略不列出来
}
```

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

     private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler () {
        empty = true ;
        boiled = false;
     }

     public static ChocolateBoiler getInstance () {
        if (uniqueInstance == null) {
            uniqueInstance = new ChocolateBoiler () ;
         }
         return uniqueInstance ;
     }

public void fill () {
     if (isEmpty () ) {
        empty = false;
        boiled = false;
        //在锅炉内填充巧克力和牛奶的混合物
     }
}
//其他的部分省略不列出来
}
```


线程一
线程二
uniqueInstance 的值
null
public static ChocolateBoiler getInstance() {
public static ChocolateBoiler getInstance() {
null
if (uniqueInstance == null) {
if (uniqueInstance == null)
null
uniqueInstance = new ChocolateBoiler();
<object1>
return uniqueInstance;
<object1>
uniqueInstance = new ChocolateBoiler();
<object2>
return uniqueInstance;
<object2>
噢！不！这看起来不妙！
糟糕！返回了两个不同对象，变成有两个巧克力锅炉了！！

解决办法 一

- 把getinstance（）变成同步（synchronized）方法

```
public class Singleton {
    private static Singleton uniqueInstance;

    // 其他有用的实例化的变量
    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // 其他有用的方法
}
```

通过增加synchronized关
getInstance()方法中，我们
个线程在进入这个方法之
先等候别的线程离开该方
就是说，不会有两个线程
时进入这个方法。

解决办法 二

- 急切创建实例

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

在静态初
(static in
中创建单
段代码保
程安全 (
safe) 。

已经有实例了，
直接使用它。

解决办法 三

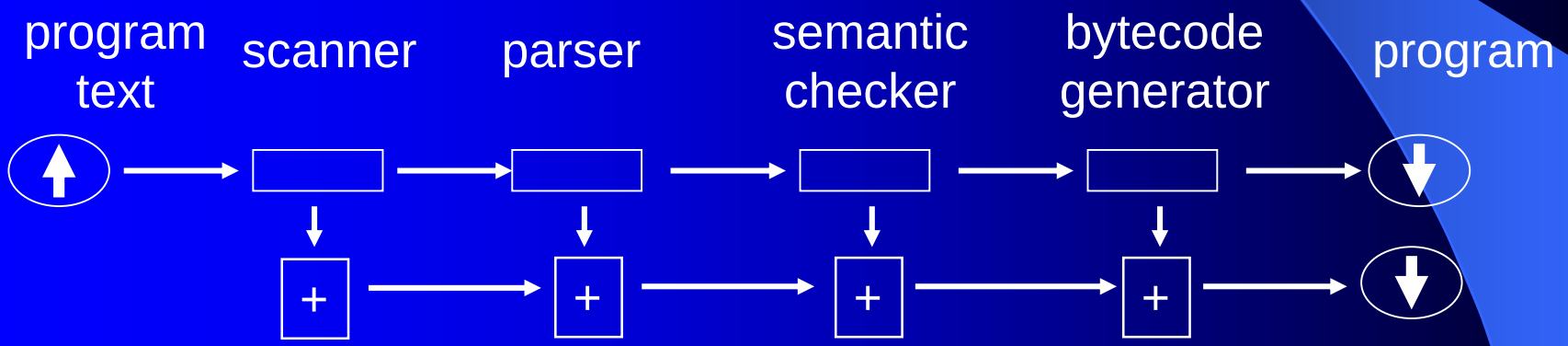
- 双重检查加锁

```
public class Singleton  {
    private volatile static Singleton uniqueInstance ;

    private Singleton ()  {}

    public static Singleton getInstance () {
            if  (uniqueInstance == null )  {
                synchronized  (Singleton.class )  {
                    if   (uniqueInstance == null)  {
                        uniqueInstance = new Singleton () ;
                    }
                }
            }
            return uniqueInstance ;
        }
}
```

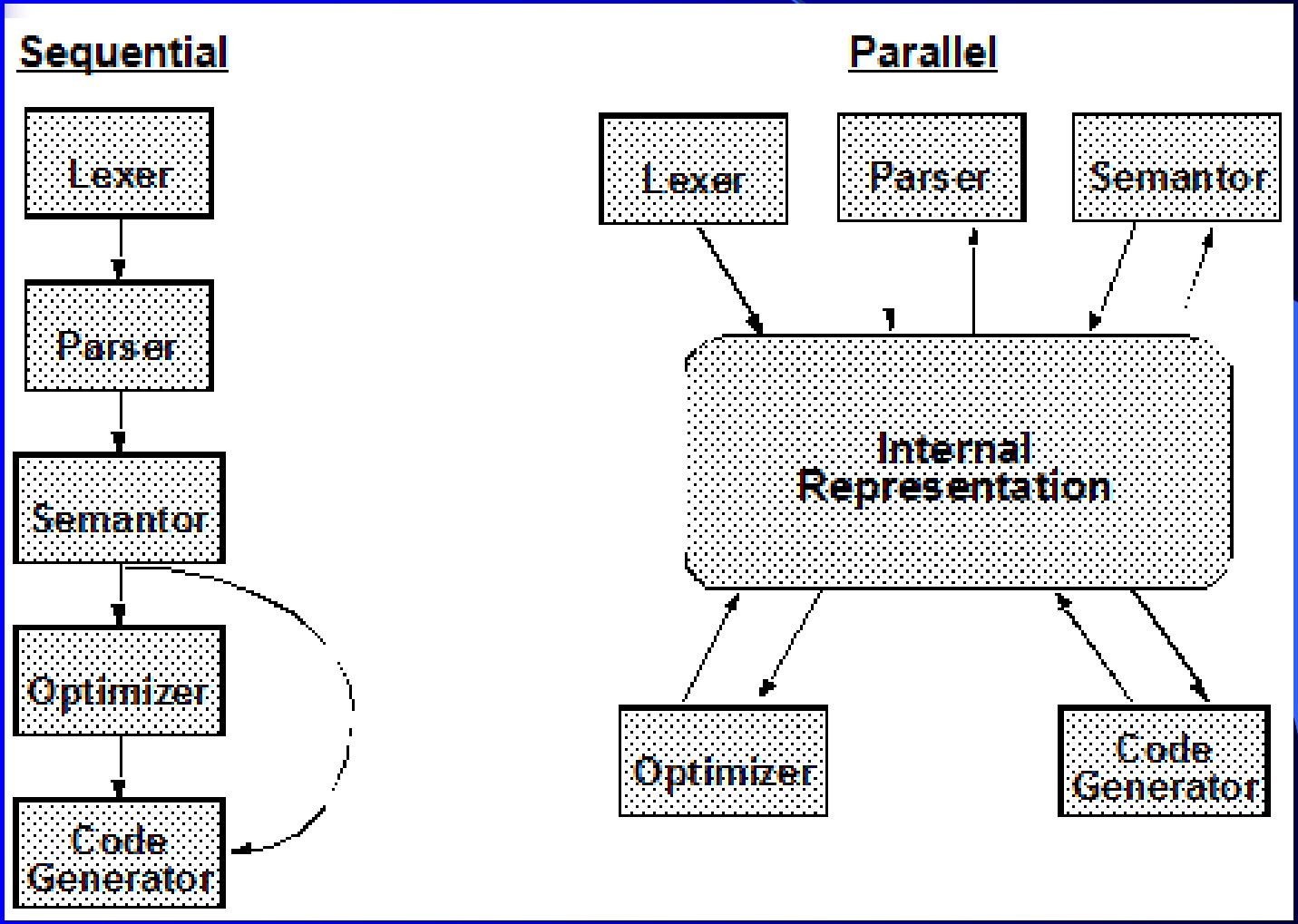
编译原理

- 如何实现一个编译器呢？

An Example — Compiler



Compiler Architecture

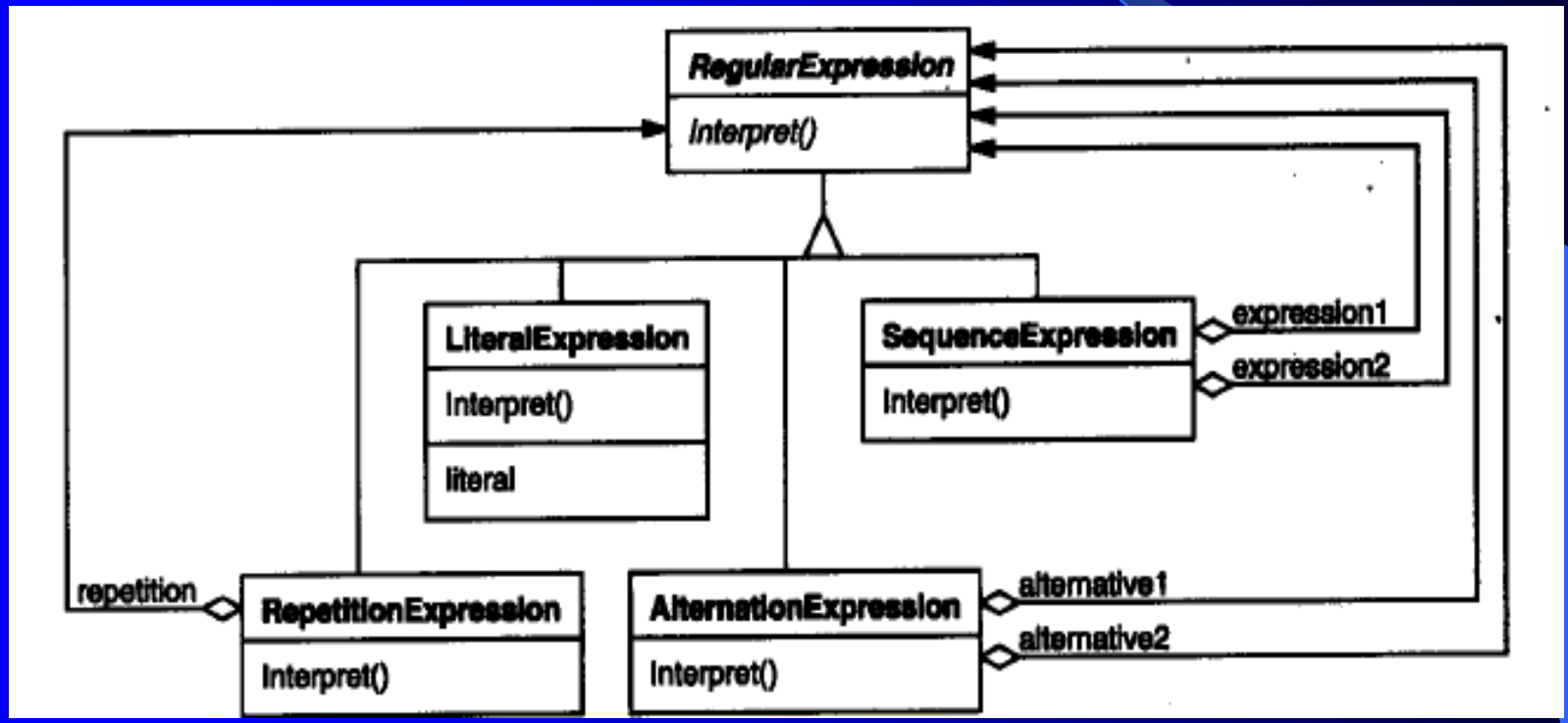


解释器

- 给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

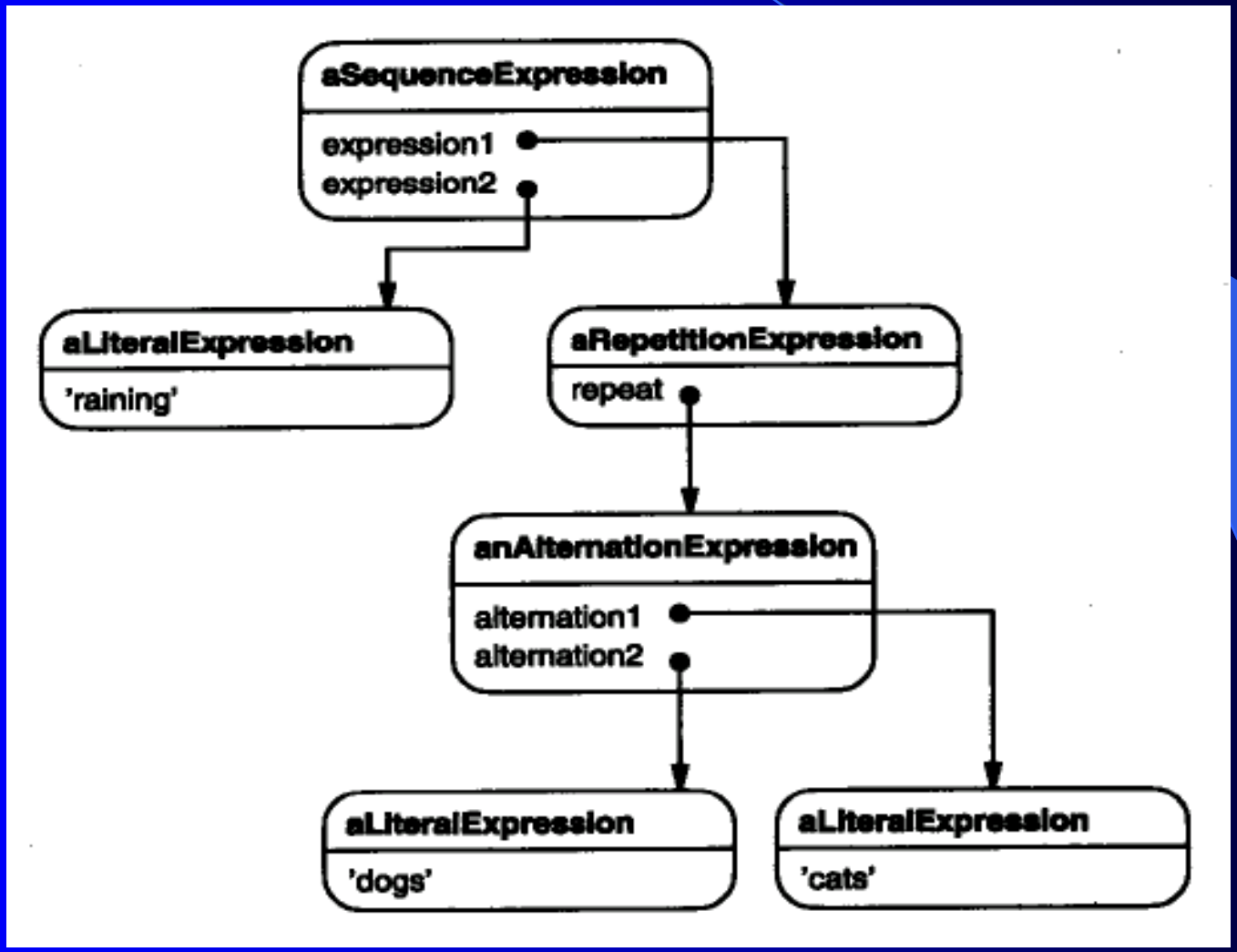
```
expression ::= literal | alternation | sequence | repetition |
               '(' expression ')'
alternation ::= expression  '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

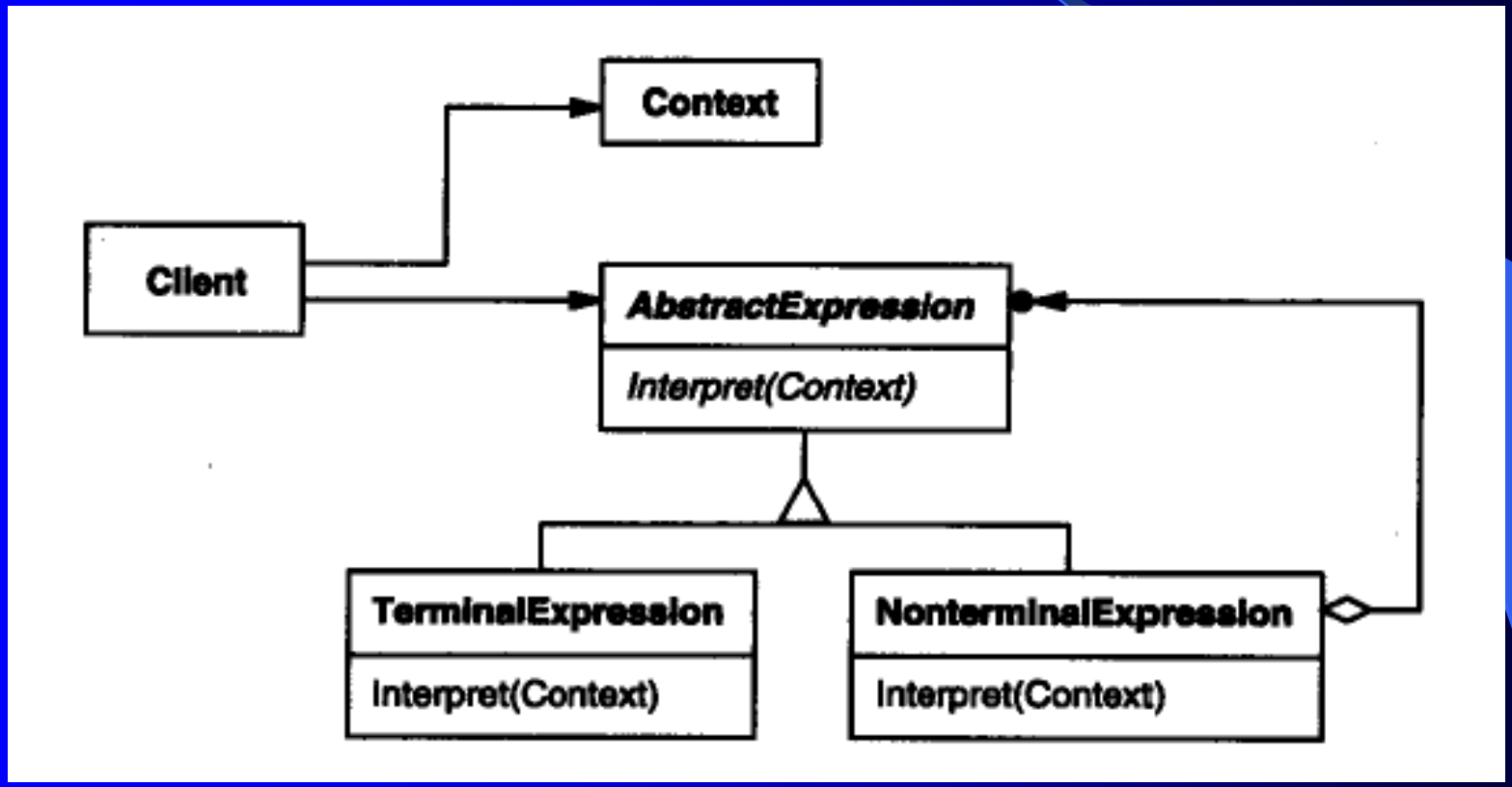
解释器模式使用类来表示每一条文法规则



- raining & (dogs | cats) *

raining & (dogs | cats) *




Context
Client
AbstractExpression
Interpret(Context)
TerminalExpression
Interpret(Context)
NonterminalExpression
Interpret(Context)

飞行模拟器

- 起飞、降落、向前、向上、向下、向左、向右
- 一直向前飞
- 当遇到云层的时候向上