Lecture5 设计模式2

1. 观察者模式 Observer Pattern

定义

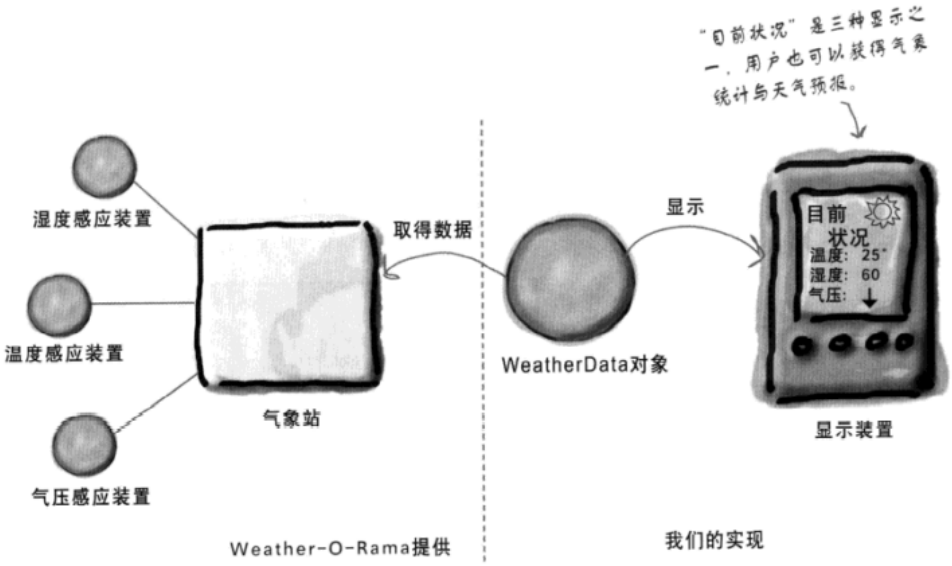
> The Observer Pattern defines a **one-to-many dependency** between objects so that when one object changes state, all its dependences are notified and updated automatically

问题引入

想要建立一个气象观测显示平台，其中要求

- `WeatherData` 对象，负责追踪目前的天气情况（温度、湿度、气压）
- 三种布告板，分别显示目前的状况，气象统计和简单的预报
- 当 `WeatherObject` 对象获得新的观测数据时，三种布告板必须实时更新
- 希望是一个可拓展的 API，让其它开发人员写出自己的布告板

整个架构大致如下



我们需要建立一个应用，利用 `WeatherData` 对象获得数据，并且更新三个布告板：目前状况，气象统计和天气预报

WeatherData

对方给我们提供了 `WeatherData` 对象，如下

第一个尝试

```
1  public class WeatherData{
2      public void measurementsChanged(){
3          float temp = getTeimerature();
4          float humidity = getHumidity();
5          float pressure = getPressure();
6
7          // 更新布告板
8          currentConditionsDisplay.update(temp, humidity, pressure);
9          statisticsDisplay.update(temp, humidity, pressure);
10         forecastDisplay.update(temp, humidity, pressure);
11     }
12 }
```

- 这里，**针对具体实现编程**，会导致以后增加或删除布告板的时候必须修改这段程序

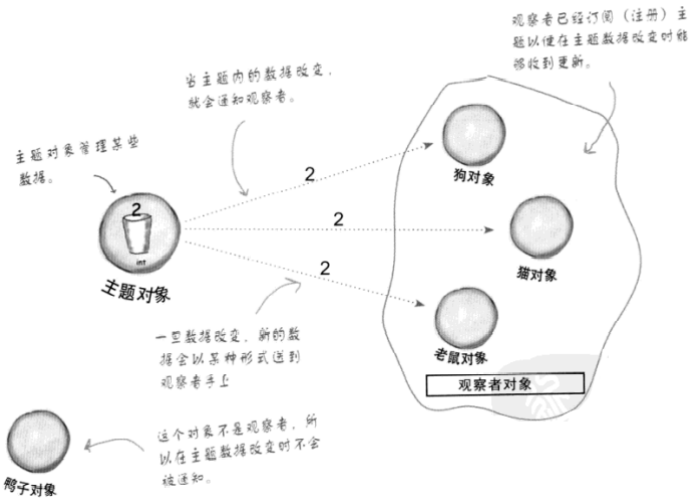
设计原则

报纸和杂志的订阅是怎么回事

- 报社的业务是出版报纸
- 向某家报社订阅报纸，只要他们有新的报纸出版，就会给你送过来，只要你是他们的订户，就会一直收到新的报纸
- 当你不想再看报纸的时候，取消订阅，他们就不会再送新的报纸来了
- 只要报社还在运营，就会一直有人向他们订阅或取消订阅报纸

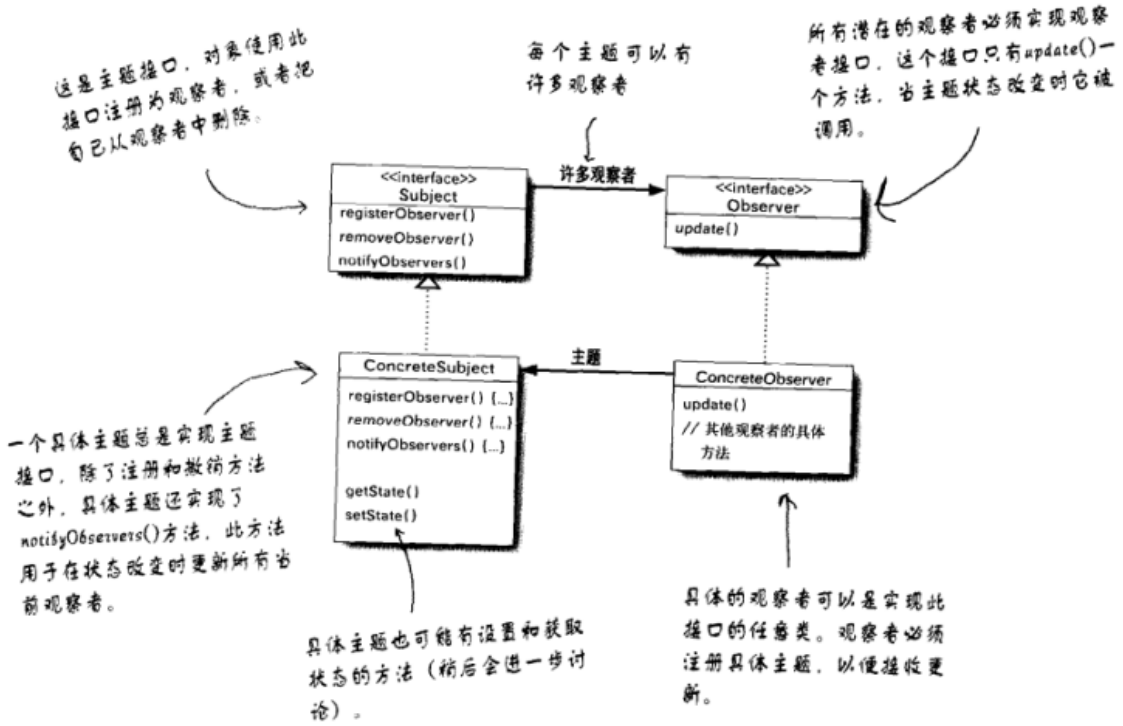
将它抽象成我们的观察者模式：

报社与订阅者	观察者模式
报社	`Subject`
订阅者	`Observer`
订阅者的订阅	`Observer.register()`
报社的通知	`Subject.notify()`



- `Subject` 和 `Observer` 之间定义了**一对多的关系**
- `Observer` 依赖于 `Subject`，只要 `Subject` 的状态有变化，`Observer` 就会被通知

观察者模式的抽象 UML 图如下



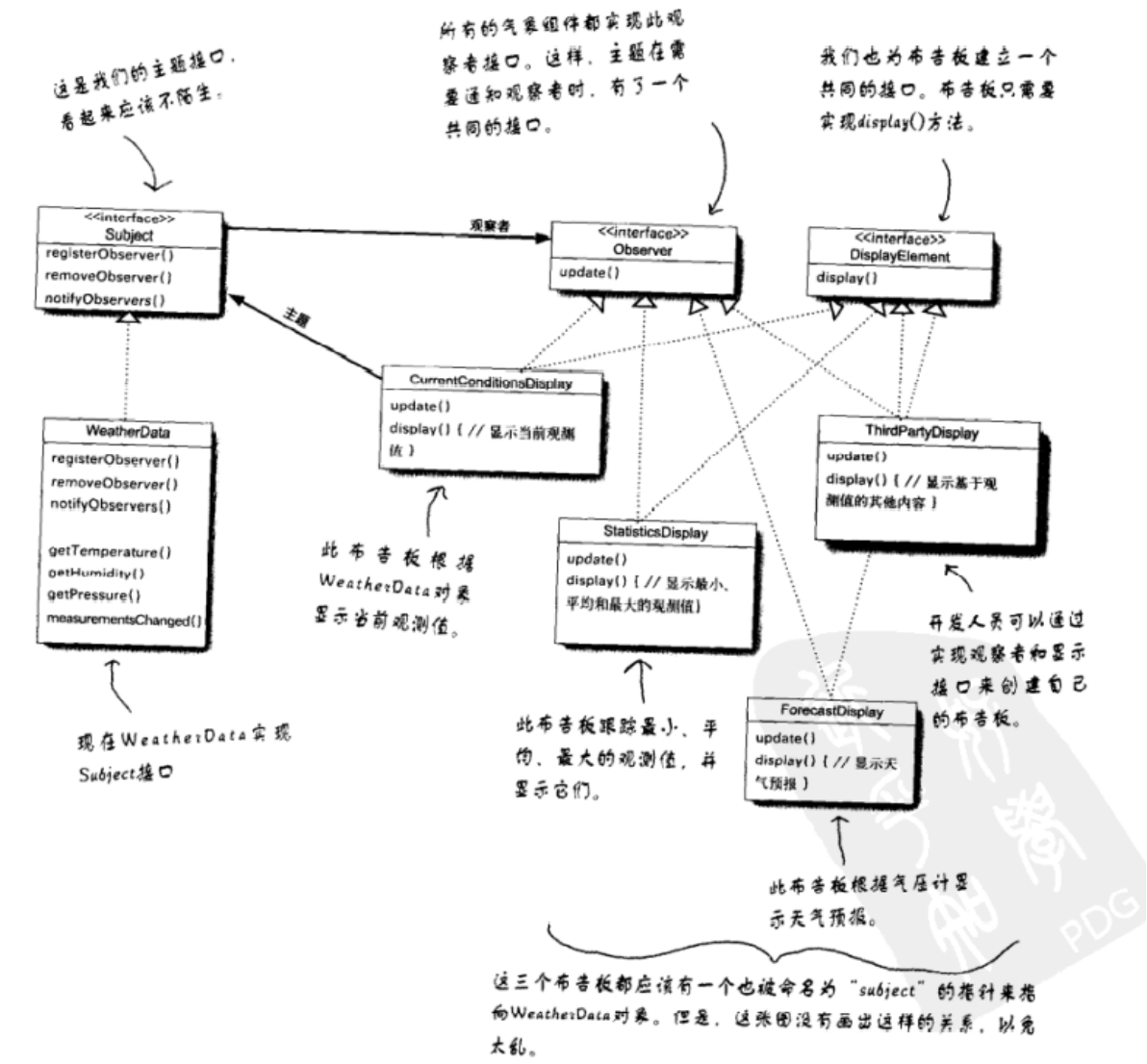
松耦合

观察者模式的实现满足了另一种设计原则

> Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependences between objects.
> 松散耦合设计允许我们构建灵活的OO系统，这些系统能够处理更改，因为它们最小化了对象之间的相互依赖关系

- `Subject` 只知道 `Observer` 实现了某个接口（其实也就是 `Observer` 接口），但是它不需要知道观察者的具体实现类是谁，做了什么以及其它细节
- 可以在任何时候添加新的 `Observer`
- 当新的 `Observer` 出现时， `Subject` 的代码不需要修改，它不在乎别的，只会发送通知给所有实现 `Observer` 接口的对象

实现



Subject 接口

```
public interface Subject{
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers(); // 当 Subject 状态改变，通知所有 Observer
}
```

Observer 接口

```
public interface Observer{
    // 当气象观测值改变时，Subject会把这些状态值当作方法的参数，传递给 Observer
    public void update(float temp, float humidity, float pressure);
}
```

DisplayElement 接口

```
public interface DisplayElement{
    public void display(); // 当布告板需要显示时，调用c
}
```

WeatherData 类

```
public class WeatherData implements Subject {

    private ArrayList<Observer> observers;// 观察者列表

    // 主题下的一些被观察属性
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<>();
    }

    // 注册 Observer
    ride
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    // 移除 Observer
    @Override
    public void removeObserver(Observer o) {
```

```
            int i = observers.indexOf(o);
            if (i >= 0) {
                observers.remove(i);
            }
        }

        // 通知所有的 Observer
        @Override
        public void notifyObservers() {
            for (int i = 0; i < observers.size(); i++) {
                Observer observer = observers.get(i);
                observer.update(temperature, humidity, pressure)
            }
        }

        public void measurementsChanged(){
            notifyObservers();
        }

        // 在测试中，假设这个代码表示了装置检测到了更新
        public void setMeasurements(float temperature, float humidity, float
    pressure){
            this.temperature = temperature;
            this.humidity = humidity;
            this.pressure = pressure;
            notifyObservers(temperature, humidity, pressure);
        }

        public float getPressure() {
            return pressure;
        }

        public float getHumidity() {
            return humidity;
        }

        public float getTemperature() {
            return temperature;
        }
    }
```

CurrentConditionDisplay 类 - 布告板1

```
public class CurrentConditionsDisplay implements Observer, Displayment {
    // 目前状况布告板
    private float temperature;
    private float humidity;
    private Subject weatherData;

    // 在weatherData处登记自己
```

```
    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    @Override
    public void display() {
        System.out.println("Current conditions: " + temperature + "F degrees amd
" + humidity + "% humidity");
    }

    // 调用update的时候，更新面板
    @Override
    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }
}
```

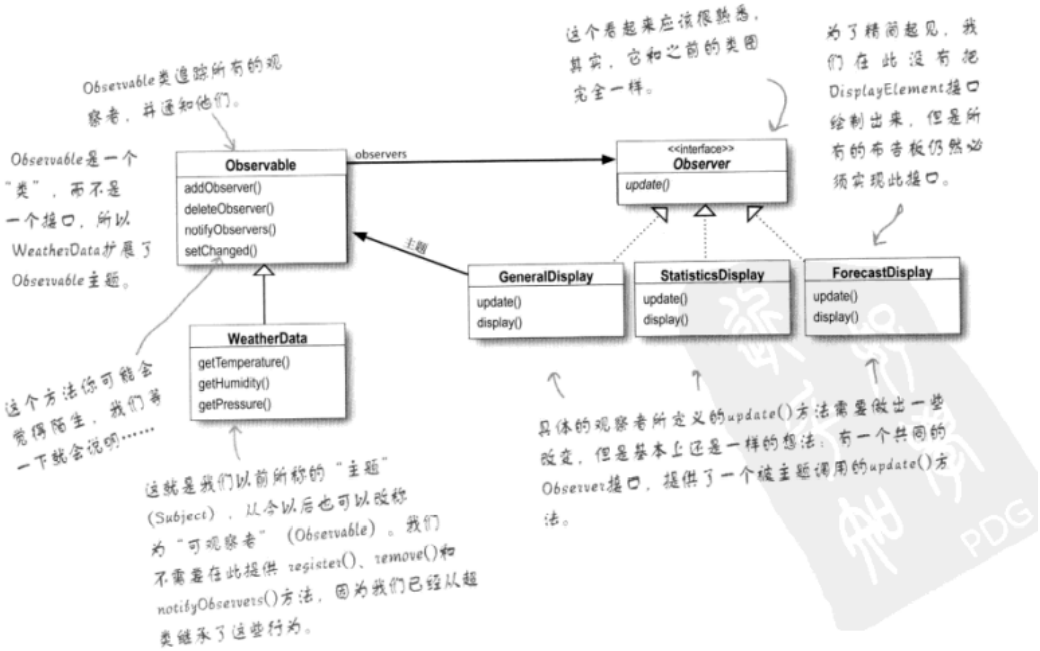
测试类

```
public class BoardTest {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();
        CurrentConditionsDisplay currentDisplay = new
CurrentConditionsDisplay(weatherData);
        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

Java 内置的观察者模式

Java API 有内置的观察者模式，在 `java.util` 包下提供了 `Observer` 接口和 `Observable` 类

- 它可以使用 `push` 和 `pull` 两种方式传送数据



- 这是适配于 Java 的气象站的 UML 图

如何把对象变成 Observer

- 实现观察者接口 `Observer`，然后调用任何 `Observable` 对象的 `addObserver()` 方法，当不想当观察者的时候，调用 `deleteObserver()` 方法

Subject 如何发出通知

- 继承 `java.util.Observable` 类产生 `Subject` 类
  - 先调用 `setChanged()` 方法，标记指定的对象的状态已经改变
  - 选择下面两种之一的通知方法
    - pull：`notifyObservers()`
    - push：`notifyObservers(Object arg)`

Observer 如何接收通知

- 观察者实现了更新的方法，但是方法的参数不太一样
- `update(Observable o, Object arg)`
  - `Subject` 本身作为第一个变量，让观察者知道是哪个 `Subject` 通知它的
  - 这是传入 `notifyObservers(Object arg)` 的数据对象

setChanged()

`setChanged()` 方法用来标记已经改变的事实

```
setChanged(){
    changed = true
}
notifyObservers(Object arg){
    if (changed){
        for every obsrver on the list {
            call update (this, arg)
    }
    changed = false
}
notifyObservers(){
    ...;
}
```

`setChanged()` 方法让你在更新观察者的时候，有更多的弹性，可以在适当的时候通知观察者

- 如果没有这个方法，比如说气象观测站温度计读数每上升 1/10 度就会更新，但是我们只希望在上升半度的时候才更新，调用 `setChanged()` 进行有效的更新

pull 数据

重写 WeatherData

```
public class WeatherData extends Observable {

    // 一些被观察属性
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {}

    public void measurementsChanged(){
        // 表示状态已经改变
        setChanged();
        // 注意这里我们调用了notifyObservers()的无参构造类型，表示采用了 pull 的方法
        notifyObservers();
    }

    // 在测试中，假设这个代码表示了装置检测到了更新
    public void setMeasurements(float temperature, float humidity, float
pressure){
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    public float getPressure() { return pressure;}
```

```
    public float getHumidity() { return humidity;}
    public float getTemperature() { return temperature;}
}
```

重写 CurrentConditionsDisplay

```
public class CurrentConditionsDisplay implements Observer, Displayment {
    // 目前状况布告板
    Observable observable;
    private float temperature;
    private float humidity;
    private Subject weatherData;

    // 在weatherData处登记自己
    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        weatherData.addObserver(this);
    }

    @Override
    public void display() {
        System.out.println("Current conditions: " + temperature + "F degrees amd
" + humidity + "% humidity");
    }

    // 调用update的时候，更新面板
    @Override
    public void update(Observable obs, Object arg) {
        if(obs instanceof WeatherData){
            WeatherData weatherData = (WeatherData) obs;
            // pull 数据
            // 调用 WeatherData 的 get 方法
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }
}
```

Java 观察者模式的缺点

- `Observable` 是一个**类而不是接口**
  - Java 不支持多重继承，限制了复用潜力
  - 因为没有 `Observable` 接口，无法建立自己的实现
- `setChanged()` 的权限修饰符是 `protected`
  - 除非你继承 `Observable`，否则无法创建 `Observable` 对象并组合到自己的对象中

2. 工厂方法模式 Factory Method Pattern

定义

> The Factory Method Pattern defines an interface for creating an object but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

问题引入

当你看见 `new` 的时候，就会想到具体的**实现**

- 代码绑着实现类会使得代码更加脆弱

当有一群相关的具体类的时候，通常会写出这样的代码

```
Duck duck;
if(picnic){
    duck = new MallardDuck();
}else if(hunting){
    duck = new DecoyDuck();
}else if(inBathTub){
    duck = new RubberDuck();
}
```

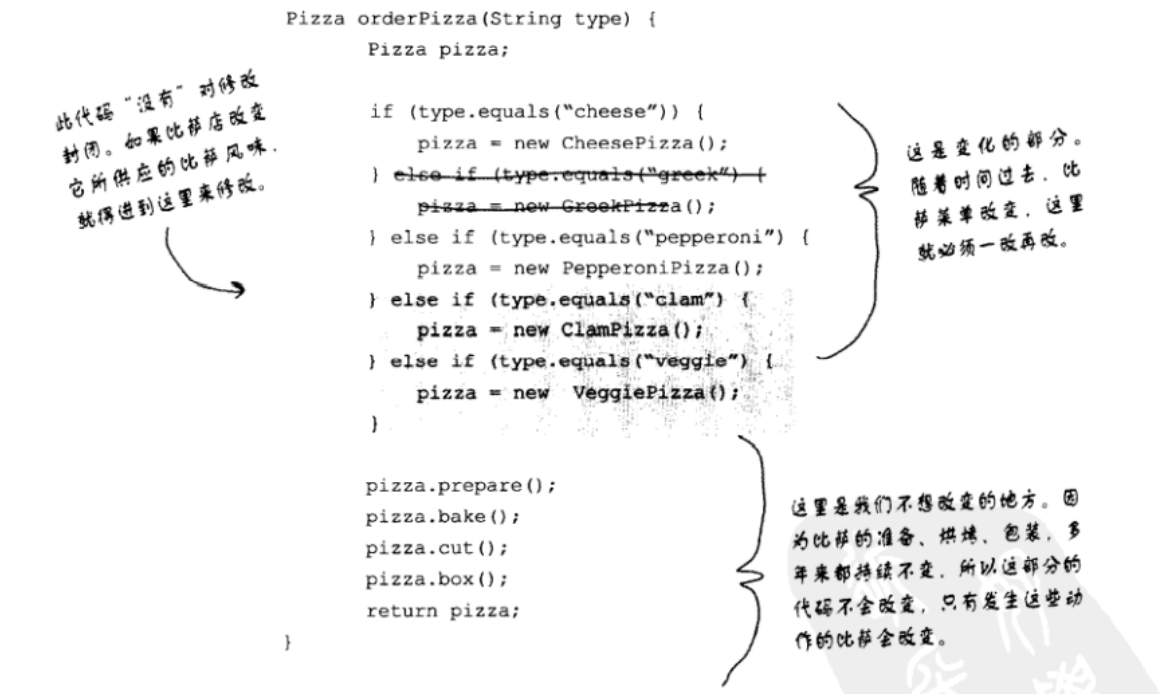
- 这样的代码，一旦有变化或拓展，就必须重新打开这段代码进行检查和修改
- 通常这样修改代码会造成部分系统难以维护和更新，更容易犯错

Pizza 店

假设你有一个 Pizza 店，你想要点一份 Pizza，你可能会这么写

```
Pizza orderPizza(String type){
    Pizza pizza;
    if(type.equals("cheese")){
        pizza = new CheesePizza();
    }else if(type.equals("greek")){
        pizza = new GreekPizza();
    }else if(type.equals("pepperoni")){
        pizza = new PepperoniPizza();
    }
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

如果你继续添加一些有风味的 Pizza，你的代码可能会有很多重复的 `if-else if`



- `orderPizza()` 方法并**没有对修改关闭**!

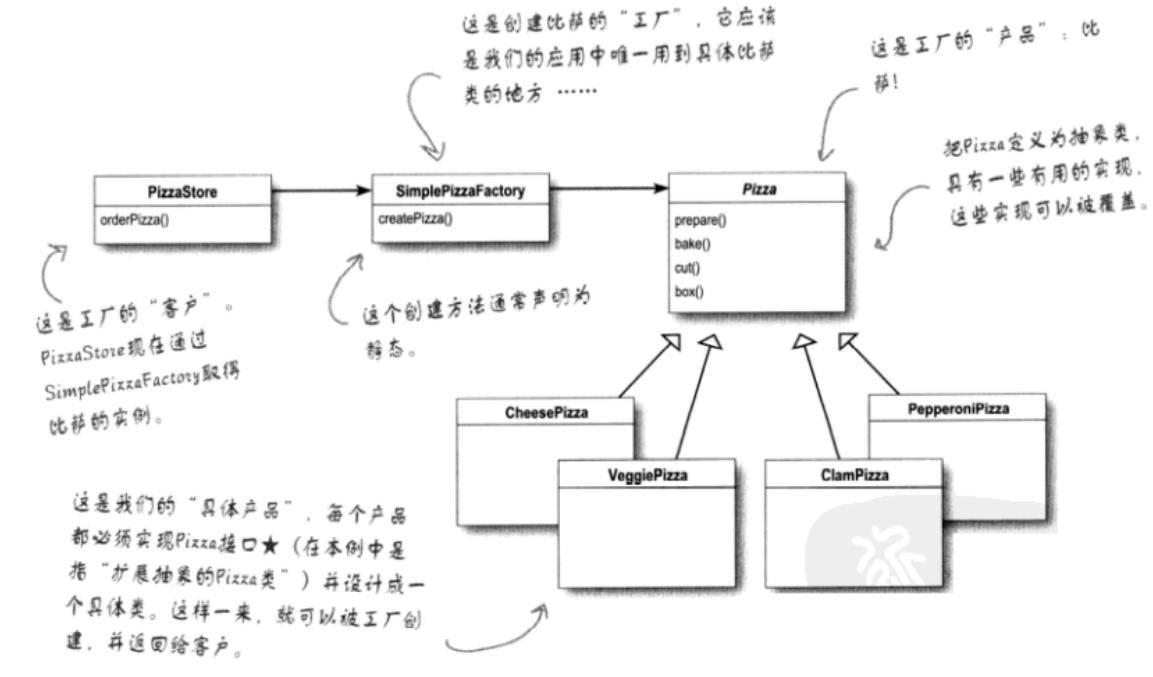
什么变了？什么没变

- Pizza 的构建类型可能会变化
- 处理 Pizza 的工序是不变的

简单工厂实现

最好将创建对象的部分移动到 `orderPizza()` 之外，我们可以把创建披萨的代码移到另一个对象，由这个对象专门创建披萨

- 这个对象被称为工厂 Factory



SimplePizzaFactory 类

```
public class SimplePizzaFactory {

    public Pizza createPizza(String type){
        Pizza pizza = null;
        if(type.equals("cheese")){
            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;
    }
}
```

PizzaStore 类

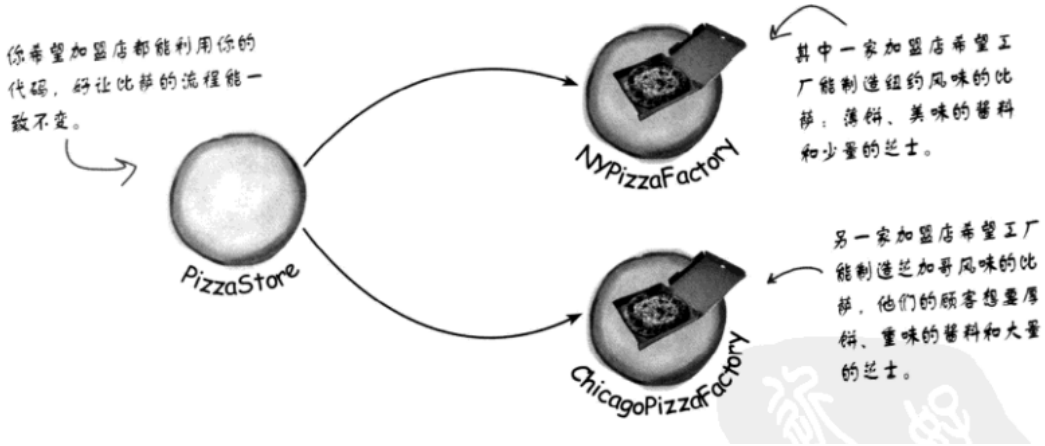
现在重写 PizzaStore 类

```
public class PizzaStore {
    SimplePizzaFactory factory;
    public PizzaStore(SimplePizzaFactory factory){
        this.factory = factory;
    }
    Pizza orderPizza(String type){
        Pizza pizza = factory.createPizza(type);
```

```
8           pizza.prepare();
9           pizza.bake();
10          pizza.cut();
11          pizza.bake();
12          return pizza;
13      }
14  }
```

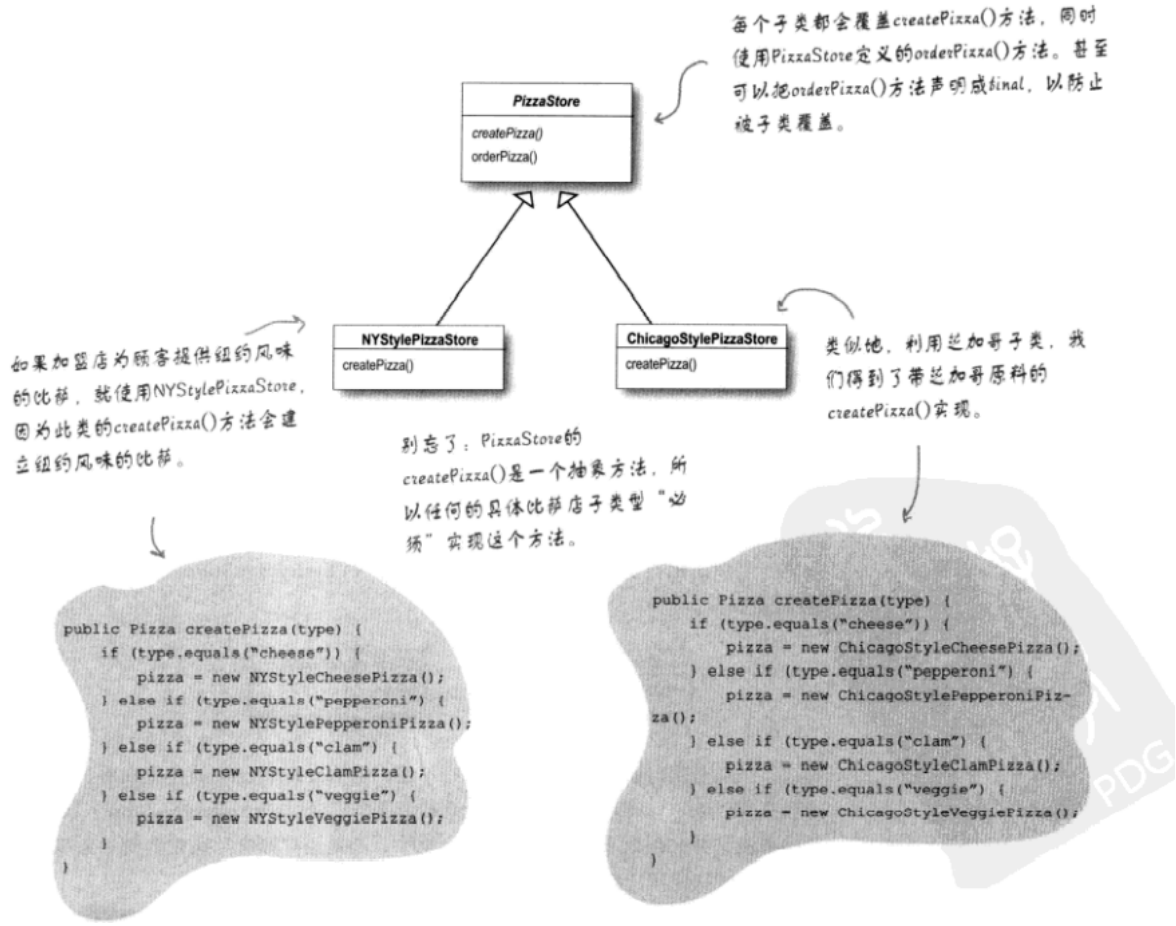
工厂方法实现

当然，你的披萨店要开加盟店了，不同加盟店想要提供不同风味的 Pizza，你希望每个加盟店都可以利用统一的代码



- 我们想要把制作 Pizza 的这个活动局限在 `PizzaStore` 类中（为了品质保证）
- 但是我们想要给加盟店自己创造不同风味 Pizza 的自由

常用的框架



- 让 `PizzaStore` 类变成 `abstract` 的
- 将 `createPizza()` 方法放回 `PizzaStore` 里，不过把它变成 `abstract` 的
- 创建 `PizzaStore` 子类作为加盟店

PizzaStore 抽象类

```java
public abstract class PizzaStore {
    Pizza orderPizza(String type){
        Pizza pizza = createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.bake();
        return pizza;
    }
    abstract Pizza createPizza(String type); // 这是一个工厂方法 Factory Method
}
```

- 我们修改了 `orderPizza()` 方法，它在所有子类中保持不变
- 但是现在加盟店可以在制作披萨的风格上有所不同
- 虽然 `orderPizza()` 定义在了 `PizzaStore` 类里，但是这个类是抽象的
    - 它事实上不能做任何事情
    - 事实上如果这个方法被调用，它调用的是具体的子类实现类的内容
    - 这些内容是仅在 `createPizza()` 方法调用时，才被确定的

NYPizzaStore 类

```
public class NYPizzaStore extends PizzaStore {

    @Override
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        switch (type) {
            case "cheese":
                pizza = new NYCheesePizza();
                System.out.println("order NYCheesePizza");
                break;
            case "pepperoni":
                pizza = new NYPepperoniPizza();
                System.out.println("order NYPepperoniPizza");
                break;
            case "clam":
                pizza = new NYClamPizza();
                System.out.println("order NYClamPizza");
                break;
            case "veggie":
                pizza = new NYVeggiePizza();
                System.out.println("order NYClamPizza");
                break;
        }
        return pizza;
    }
}
```

Pizza 抽象类

```
public abstract class Pizza {

    String name;
    String dough;
    String sauce;
    ArrayList<String> topping = new ArrayList<>();

    public void prepare(){
        System.out.println("Preparing " + name);
        System.out.println("Tossing dough " + dough);
        System.out.println("Adding sauce " + sauce);
        for(String top : topping){
            System.out.println(top + " ");
        }
    };

    public void bake(){
        System.out.println("Bake for 25 minutes");
```

```
    };

    public void cut(){
        System.out.println("Cutting the pizza into diagonal slices");
    };

    public void box(){
        System.out.println("Place pizza in official PizzaStore box");
    }

    public String getName(){
        return name;
    }
}
```

NYCheesePizza 类

```
public class NYCheesePizza extends Pizza {
    public NYCheesePizza(){
        name = "NY Style Sauce and Cheese Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";
        topping.add("Grated Reggiano Cheese");
    }
}
```

测试类

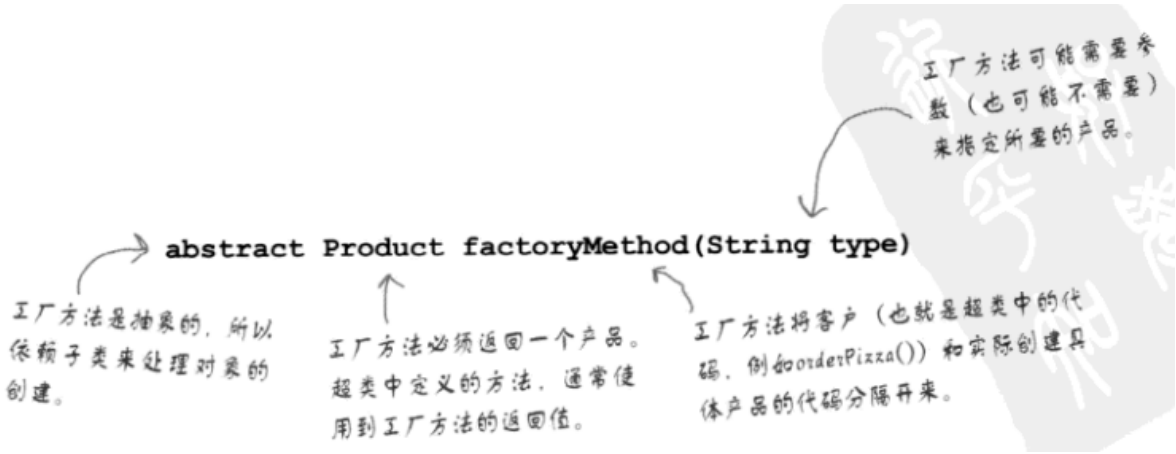
```
public static void main(String[] args) {
    PizzaStore nyPizzaStore = new NYPizzaStore();
    Pizza pizza = nyPizzaStore.orderPizza("cheese");
}
```

```
order NYCheesePizza
Preparing NY Style Sauce and Cheese Pizza
Tossing dough Thin Crust Dough
Adding sauce Marinara Sauce
Grated Reggiano Cheese
Bake for 25 minutes
Cutting the pizza into diagonal slices
Bake for 25 minutes
```

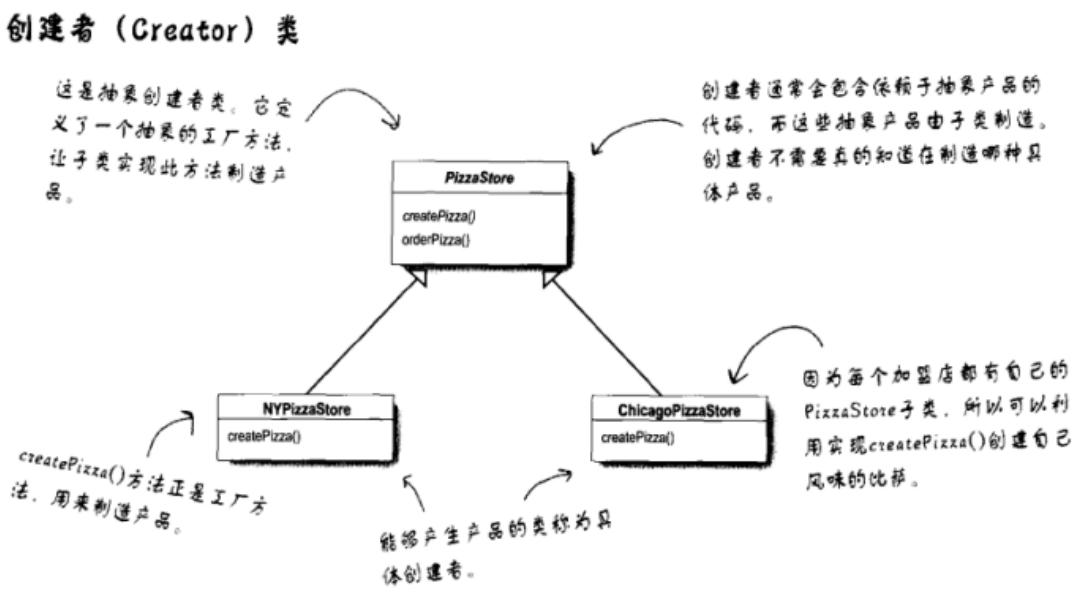
什么是工厂方法

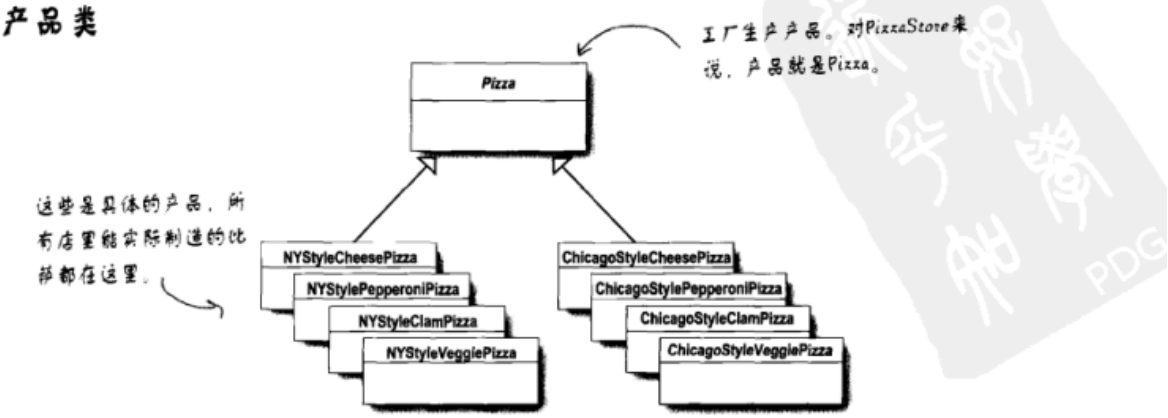
工厂方法用来处理对象的创建，并将这样的行为封装在子类中，这样，客户程序中关于父类的代码就和子类对象创建代码解耦了

```
1   abstract Product farcutyMethod(Strign type)
```



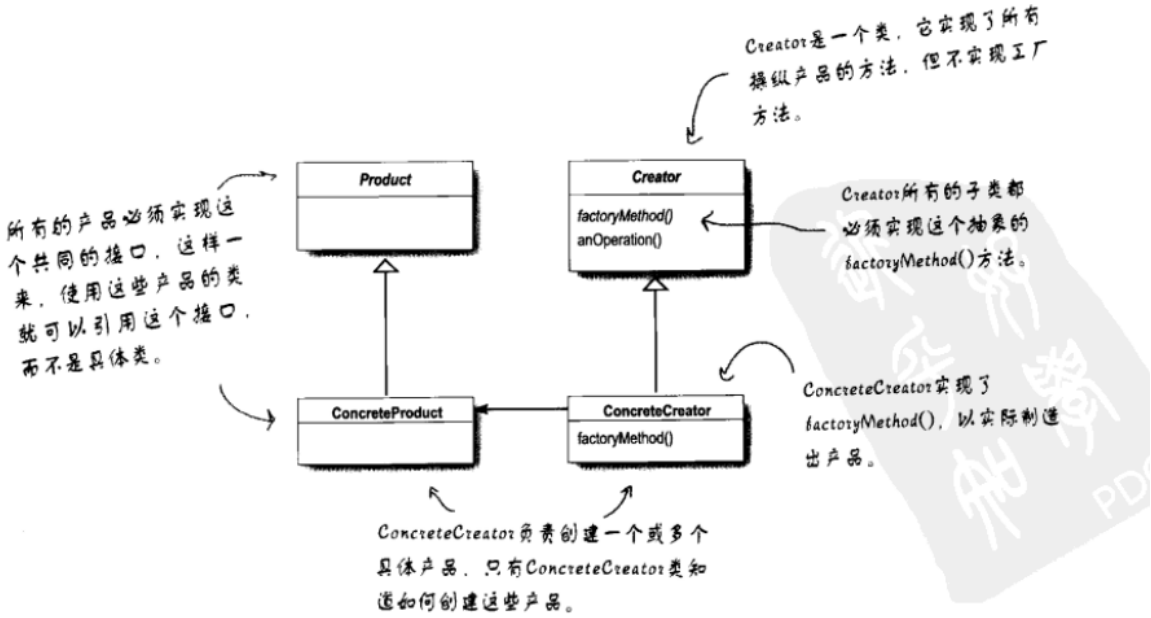
一般它有两类组成元素





UML

工厂方法定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个，工厂方法让类把实例化推迟到了子类



设计原则

> Open for Extension, Closed for Modification
> 对扩展开放，对修改关闭

- 允许很容易地扩展类以合并新的行为
- 不去修改已经存在的带啊吗
  - 因为每次修改它，都有可能引入新的错误
- 这导致设计能够适应变化，但也足够灵活地接受新功能，以满足不断变化的需求

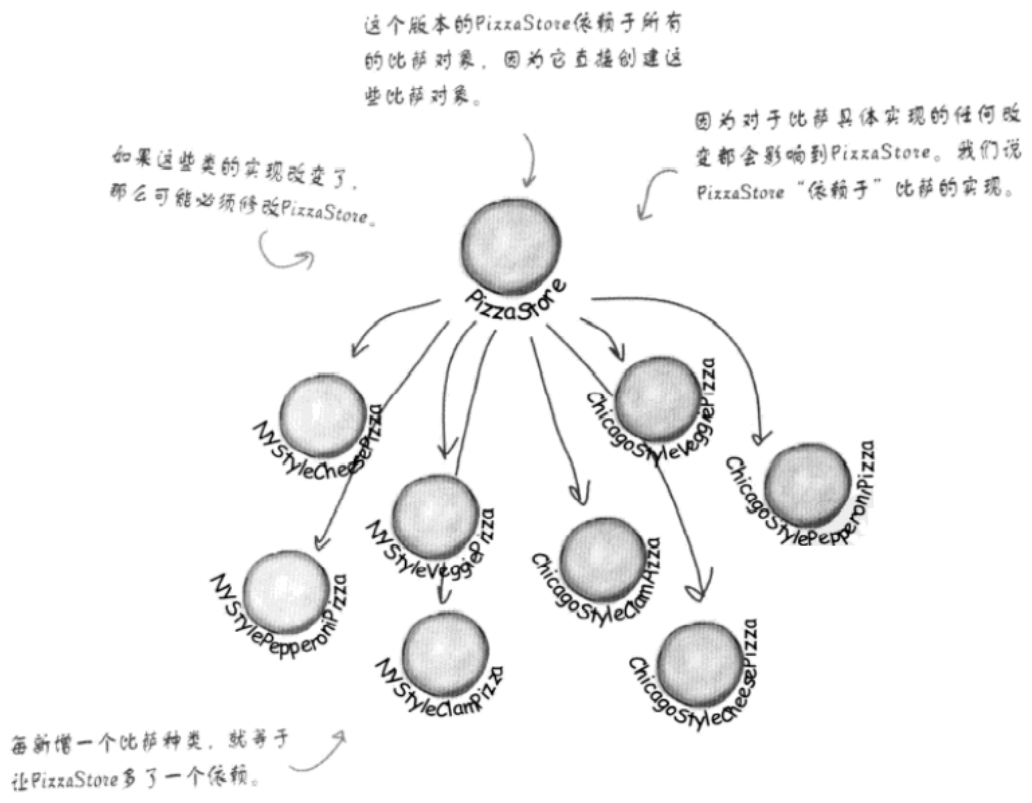
如果没有学过 OO 的话，你对 `PizzaStore` 的设计可能是这样的

```
public class DependentPizzaStore {

    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

处理所有纽约风味比萨。

处理所有芝加哥风味比萨。

- 当你实例化一个对象的时候，就是在依赖它的具体类，如果画成一张图，看起来是这样的

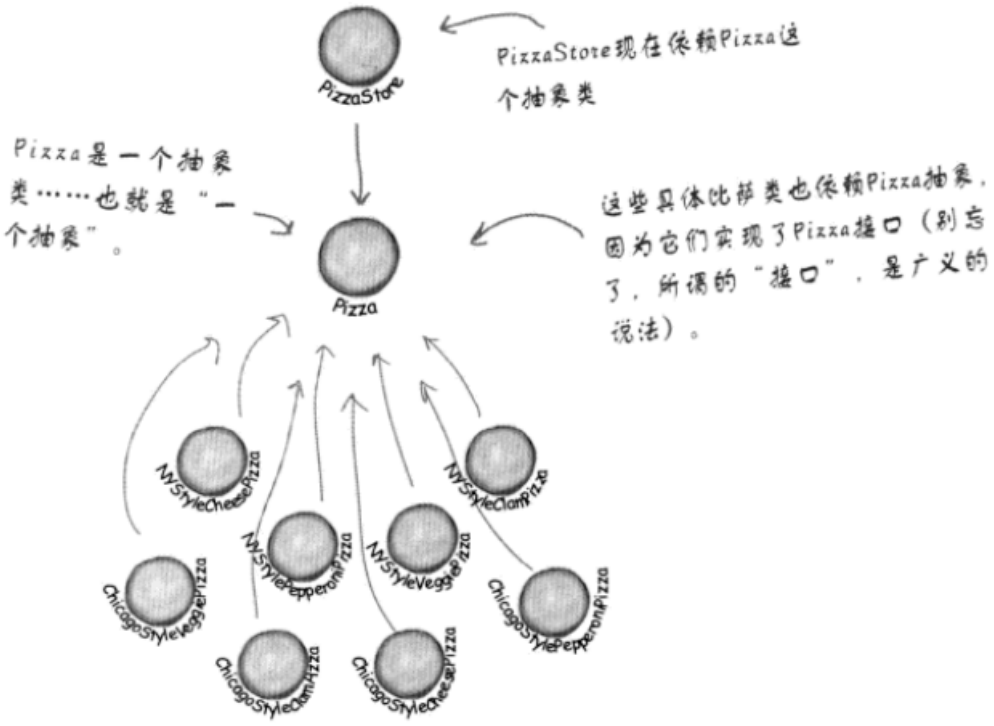


在代码里减少点依赖是好事，在 OO 中，有一个设计原则，叫**依赖倒置原则**

> Depend upon abstractions. Do not depend upon concrete class.
> 依赖于抽象。不要依赖于具体的类。

这里强调，不能让高层组件（ `PizzaStore` ）依赖底层组件（具体 Pizza 的实现类），两者都应该依赖于抽象

如果你采用了工厂方法，类图应该看起来像下面这样



- 高层组件和底层组件都依赖了 `Pizza` 抽象

下面几个指导方法帮助你遵循依赖倒置

- 变量不可以持有具体类的引用
  - 如果使用 `new` ，就会持有具体类的引用，可以是使用工厂方法来避开这样的做法
- 不要让类派生自具体类
  - 如果派生自具体类，就会依赖具体类，请派生一个抽象（接口 / 抽象类）
- 不要覆盖父类中已实现的方法
  - 如果覆盖父类中已经实现的方法，那么你的父类就不是一个真正适合被继承的抽象，父类中已经实现的方法，应该由所有的子类共享

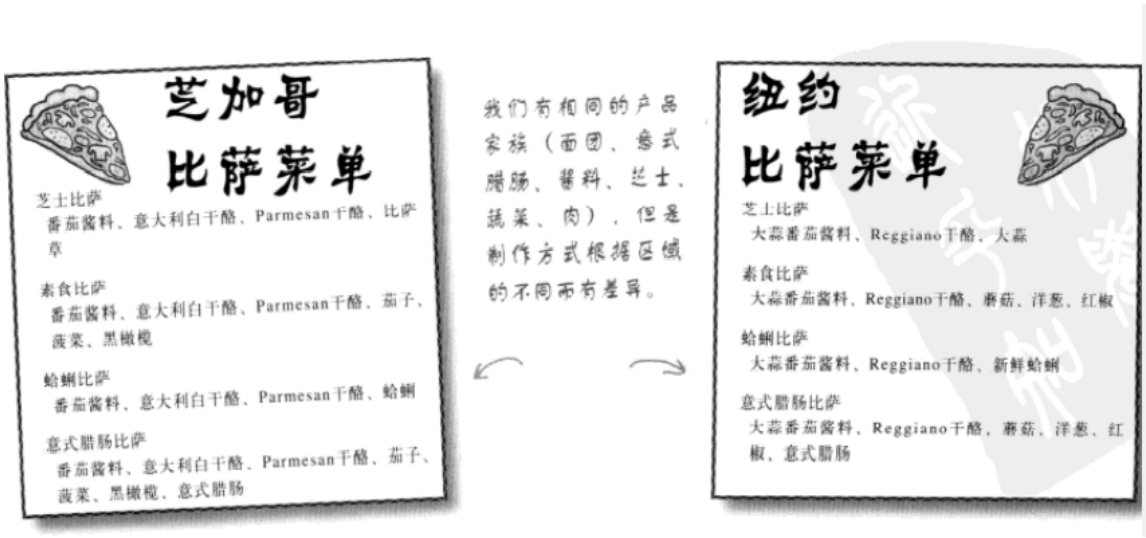
3. 抽象工厂 Abstract Pattern

定义

> The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

问题引入

回到披萨店，现在对于不同加盟店，想要使用的原料也不同了



- 每个加盟店实现了一个完整的原料家族

实现

首先为工厂定义一个接口，这个接口负责创建所有的原料

PizzaIngredientFactory 接口

```
public interface PizzaIngredientFactory{
    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();
}
```

- 要为每个区域创造一个工厂
- 实现一组原料供工厂使用
- 将原料工厂整合进 `PizzaStore` 里面

NYPizzaIngredientFactory 类

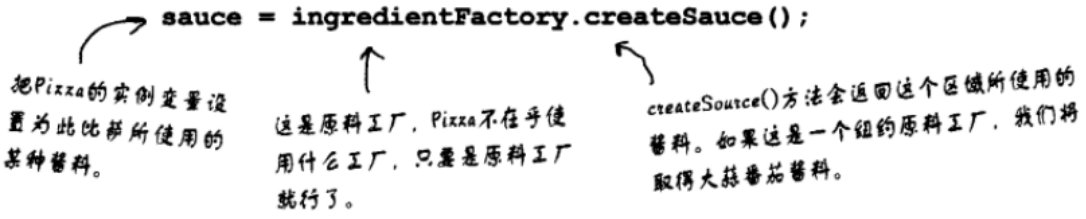
```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory{
    @Override
    public Dough createDough(){
        return new ThinCrustDough();
    }
    @Override
    public Sauce createSauce(){
        return new MarinaraSauce();
```

```
    }
    @Override
    public Cheese createCheese(){
        return new RegginaoCheese();
    }
    @Override
    public Veggies[] createVeggies(){
        Veggies [] veggies = {new Garlic(), new Onion(), new Mushroom(), new RedPepper()}
        return veggies;
    }
    @Override
    public Clams createClam(){
        return new FreshClams();
    }
}
```

Pizza 抽象类

```
public abstract Pizza{
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies;
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;
    abstract void prepare();
    public void bake(){
        System.out.println("Bake for 25 minutes");
    };

    public void cut(){
        System.out.println("Cutting the pizza into diagonal slices");
    };

    public void box(){
        System.out.println("Place pizza in official PizzaStore box");
    }

    public String getName(){
        return name;
    }
}
```

- `Pizza` 的代码利用相关工厂生产的原料，所生产的原料依赖使用的工厂，`Pizza` 类不关心这些原料，它只知道如何制作披萨



CheesePizza 类

```
public class CheesePizza extends Pizza{
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory){
        this.ingredientFactory = ingredientFactory;
    }
    void prepare(){
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

NYPizzaStore 类

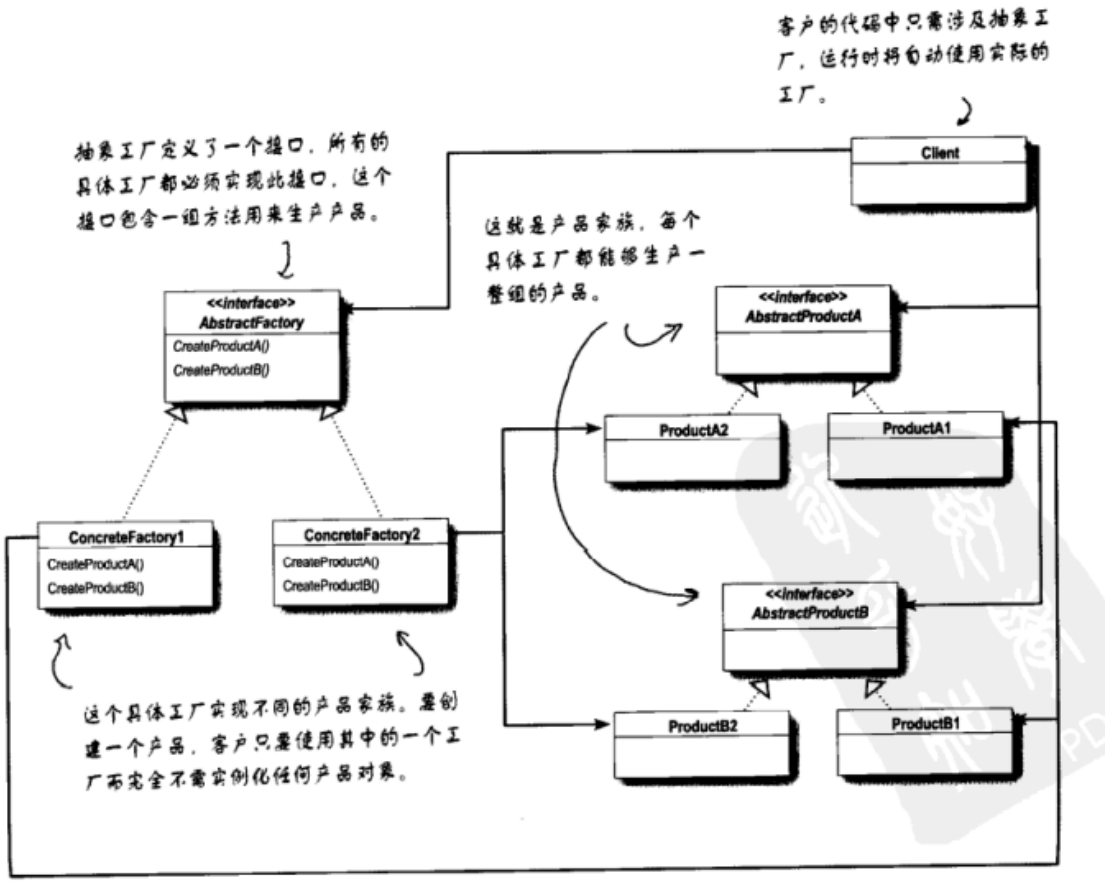
```
public class NYPizzaStore extends PizzaStore {

    @Override
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredentFactory = new NYPizzaIngredientFactory();
        switch (type) {
            case "cheese":
                pizza = new NYCheesePizza(ingredentFactory);
                System.out.println("order NYCheesePizza");
                break;
            case "pepperoni":
                pizza = new NYPepperoniPizza(ingredentFactory);
                System.out.println("order NYPepperoniPizza");
                break;
            case "clam":
                pizza = new NYClamPizza(ingredentFactory);
                System.out.println("order NYClamPizza");
                break;
            case "veggie":
                pizza = new NYVeggiePizza(ingredentFactory);
                System.out.println("order NYClamPizza");
                break;
        }
```

```
25            return pizza;
26        }
27    }
```

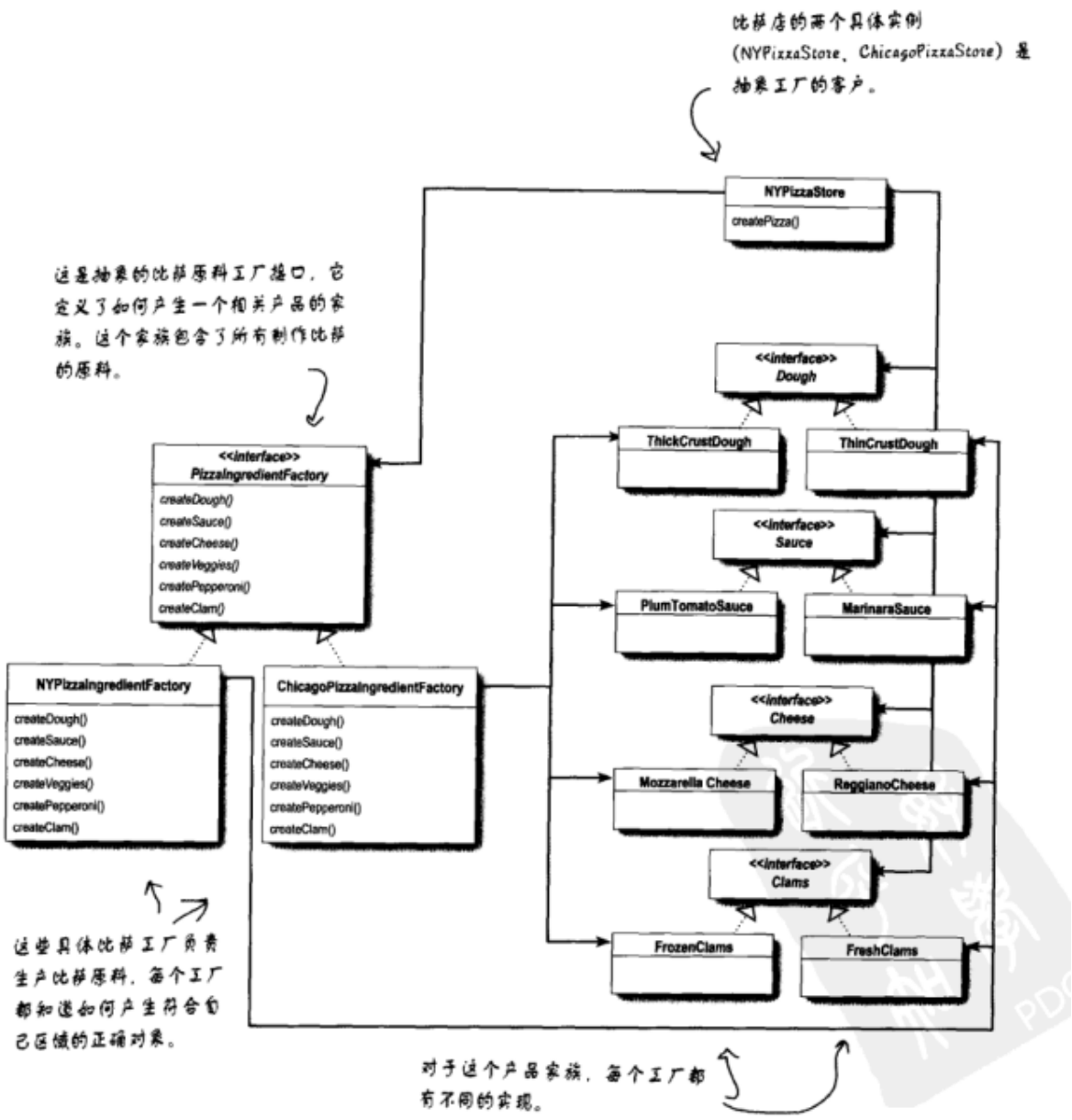
什么是抽象工厂

抽象工厂模式可以创建产品的家族，而不需要指明其具体类

它允许客户使用抽象的接口来创建一组相关的产品，而不需要关心实际的产品是什么，这样客户从具体的产品中被解耦



以我们的 PizzaIngredientFactory 为例的话



其实，工厂方法潜伏在抽象工厂里面，因为抽象工厂定义了负责创建一组产品的接口，接口内的每一个方法都负责创建一个具体的产品，自然，抽象工厂里的方法运用的都是工厂方法

工厂方法 vs 抽象工厂

工厂方法	抽象工厂
将应用程序与特定实现解耦	将应用程序与特定实现解耦
通过继承创建对象 通过扩展类和覆盖工厂方法来创建对象	通过对象组合创建对象 通过为一系列产品提供抽象类型来创建对象 子类定义了产品是如何产生的
如果你不知道需要哪些具体的类，这很有用	新产品必须改变接口