面向对象的软件构造导论

第五章 设计模式导论

课程回顾

- 继承
- 接口与抽象类
- 多态与重写
- Java多继承问题
- 超类与super关键字
- 异常的继承框架

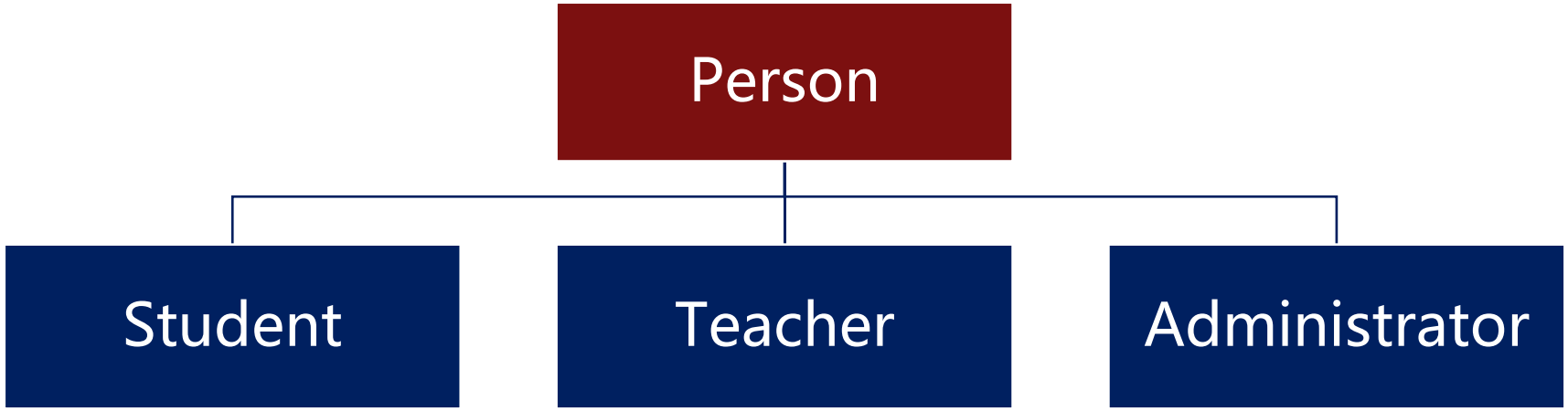
为什么要继承？

```
class Person {
    String name;
    int age;
    public String getName() {...}
    public void setName(String name) {...}
}
```

```
class Student {
    String name;
    int age;
    String school;
    public String getName() {...}
    public void setName(String name) {...}
    public String getSchool() {...}
}
```

大量重复的代码！

继承-父类和子类



- Person类与其他类(如Student类) 存在关联
- 在Student与Person之间存在着明显的is-a-kind-of关系，这是**继承**的一个明显特征

继承-类，超类和子类

- 关键词**extends**表明正在构造的新类派生于一个已存在的类。
  - 已存在的类：超类(superclass)，基类，父类
  - 新类：子类(subclass)，派生类，孩子类
- 继承能**复用**已存在的类的方法，并增加一些新的方法和字段，使得新类能够适应新的情况。
- 父类/超类刻画子类所**共同拥有的属性和方法**。
- 不同的子类有各自不同的属性和方法

继承-定义父类、子类

```
1. public class Person{
2.     private String name;
3.     private int age;
4.     public Person(String name, int age){
5.         this.name = name;
6.         this.age = age;
7.     }
8.     public String getName(){
9.         return name;
10.    }
11.    private String setName (String name){
12.        this.name = name;
13.    }
14.}
```

```
1. public class Student extends Person{
2.     String school;
3.     public Student(String name, int age,
   String school) {
4.      super(name, age); // 调用父类的构造方法
5.      this.school= school;
6.     }
7.
8.
9.     public string getSchool(){
10.        return this.school;
11.    }
12.}
```

- 利用extends关键字
- 子类不能直接继承父类的构造方法，需利用super关键字
- 子类能**复用**父类的属性和方法，子类能**增加**一些新的属性与方法

接口 VS. 抽象类

- 接口与抽象类比较

	Abstract class	Interface
抽象方法	可以定义抽象方法	可以定义抽象方法
字段	可以定义字段	无字段
继承	只能extends一个class	可以implements多个interface

- 思考：有了抽象类，为什么还需要接口？
  - 抽象类解决不了多继承的问题
  - 要实现的方法不是当前类的必要方法
    - 例如“会唱歌” 不是Person类的必要方法，如果设置成抽象方法会浪费资源
  - 为不同类型的多个类实现同样的方法
    - 例如：Person、鸟类、收音机、手机都能“唱歌”

多态

- 多态（Polymorphism）是面向对象编程的三大重要特性之一
- 多态是同一个行为具有多种表现形态的能力
  - 不同情况下的不同处理方式
  - 程序中定义的变量和方法在编程时并不确定，而是在程序运行期间才确定。

多态

- 多态的好处
  - 减少耦合
  - 增强可替换性
  - 增强可扩展性
  - 提高灵活性

- 使用多态的三个必要条件：
  - 继承
  - 重写
  - 父类引用指向子类

- 多态的三种实现方式：
  - 重写
  - 抽象类和抽象方法
  - 接口

多态的实现1：重写

```
1. public class Person{
2.     private String name;
3.     private int age;

4.     public void getTarget(){
5.         System.out.println( “美好生活” );
6.     }
7. }
```

```
1. public class Student extends Person{
2.     @Override
       public void getTarget(){
3.         System.out.println(“功夫到家");
4.     }
5. }
```

```
1. public class Teacher extends Person{
2.     @Override
       public void getTarget(){
3.         System.out.println(“传道授业解惑");
4.     }
5. }
```

- 在Student和Teacher类中，我们重写了getTarget方法，覆盖之前的getTarget.
- 这些方法有相同的名称、返回类型及参数。
- 编程技巧：加上@Override可以让编译器帮助检查是否进行了正确的覆写。

多态的实现1：重写

```
public static void show_target(Person P_x) {
     System.out.print(" 目标: ");
     P_x.getTarget();
 }
Person P1= new Student();
Person P2= new Teacher();
show_target(P1);
show_target(P2);
```

- 通过父类的引用调用子类重写的方法
- 在show_target()函数中我们只需要传入父类的引用
  - 父类类名 引用名称 = new 子类类名();

多继承

- 多继承指的是一个类可以同时从多个父类那里继承行为和特征，然而Java为了保证数据安全，**只允许单继承**。
- 多继承存在的问题：
  - 若子类继承的父类中**拥有相同的成员变量**，子类在引用该变量时将无法判别使用哪个父类的成员变量。
  - 若一个子类继承的**多个父类拥有相同方法**，同时子类并未覆盖该方法（若覆盖，则直接使用子类中该方法），那么调用该方法时将无法确定调用哪个父类的方法。

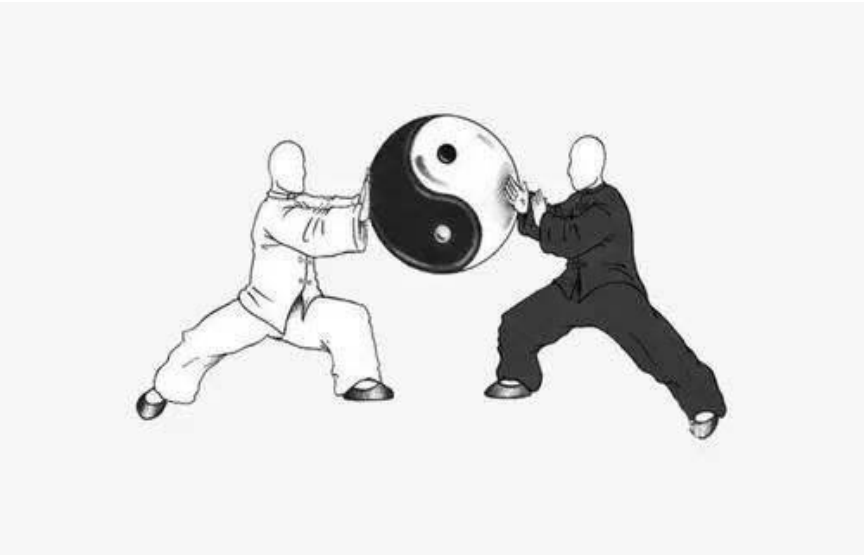
Java如何实现多继承效果

- 有时候开发人员确实需要实现**多重继承**，而且现实生活中真正地存在这样的情况。例如遗传，我们可以既继承父亲的行为和特征，又继承母亲的行为和特征。
- Java提供的两种方法让我们实现多继承：
  - 内部类
  - 接口

课程导航

- 面向对象设计原则
- 设计模式
- 单例模式
- 简单工厂模式
- 工厂模式
- 抽象工厂设计模式

面向对象思想与设计模式的方法：“道与术”



太极拳的“道与术”

道：

虚灵顶劲、松垮垂臀、沉肩坠肘、上下相随，用意不用力

拳劲旨在似松非松、掌力意在将展未展。

术：

白鹤亮翅、金刚倒锥、抱虎归山、进步搬拦捶.....

面向对象思想与设计模式的方法：“道与术”

面向对象的思维方法是“道”，是所有设计模式的指导性原则和思想

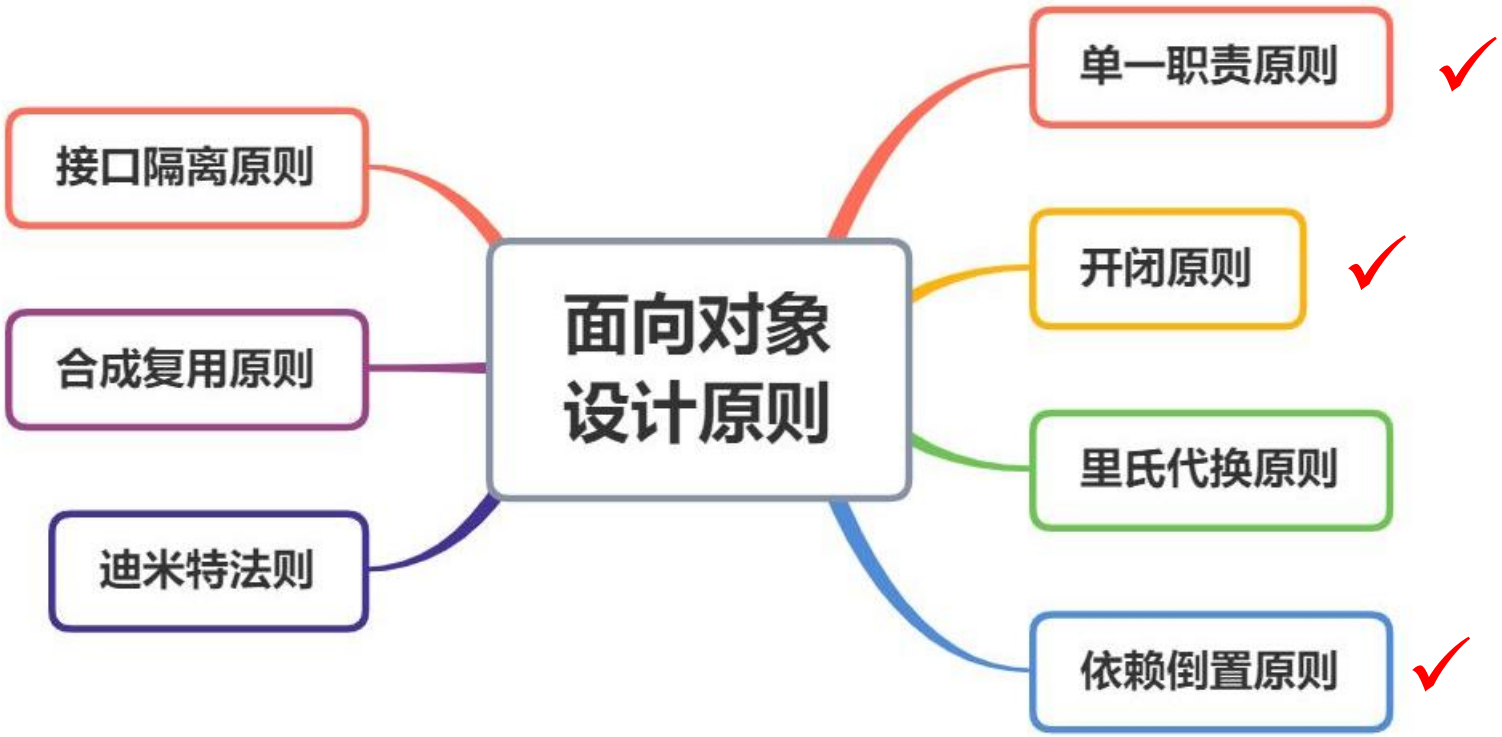
抽象、封装、继承、多态

设计模式的具体方法是“术”，是面向对象思想在解决不同的具体问题时的具象化方法。

单例模式、工厂模式、策略模式、观察者模式 等等

面向对象设计原则

- 在设计面向对象的程序时，需遵循的常用原则有七个，包括：



单一职责原则

单一职责原则（Single Responsibility Principle – SRP）定义：

就一个类而言，应该仅有一个引起它变化的原因。

- 不要把变化原因各不相同的职责放一起，因为不同变化会影响到不相干的职责。
- 如果一个类承担的职责过多，它被复用的可能性越小，就等于把这些职责耦合在一起，而一个职责的变化可能会削弱和抑制这个类完成其他职责的能力。
- 如果多于一个的动机去改变一个类，那么这个类就具有多余一个的职责，就应该要考虑类的职责分离。

开闭原则

开闭原则（Open-Closed Principle – OCP）定义：

一个软件实体应当对扩展开放，对修改关闭。即在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展。

- 通过扩展已有的软件系统，可以提供新的行为，以满足对软件的新需求，使变化中的软件系统有一定的适应性和灵活性。
- 在面向对象编程中，通过抽象类及接口，规定了具体类的特征作为抽象层，相对稳定，不需更改，从而满足“对修改关闭”；而从抽象类导出的具体类可以改变系统的行为，从而满足“对扩展开放”。

依赖倒转原则

依赖倒转原理（Dependency Inversion Principle – DIP）的定义：

高层模块不应该依赖低层模块，他们都应该依赖抽象。要针对接口编程，不要针对实现编程。

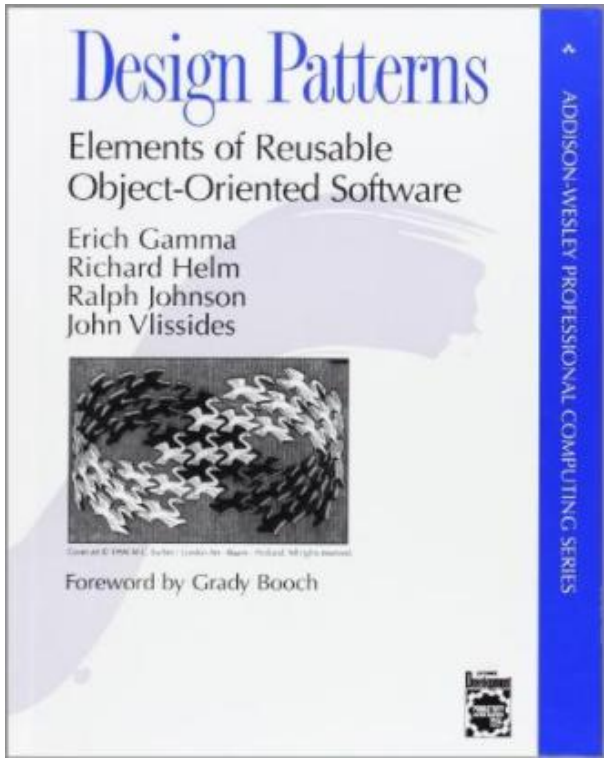
- 应当使用接口和抽象类进行变量类型声明、参数类型声明、方法返还类型说明，以及数据类型的转换等，而不要使用具体类。
- 传统的过程性系统的设计办法倾向于使高层次的模块依赖于低层次的模块，抽象层次依赖于具体层次。倒转原则就是把这个错误的依赖关系倒转过来。

课程导航

- 面向对象设计原则
- 设计模式
- 单例模式
- 简单工厂模式
- 工厂模式
- 抽象工厂设计模式

设计模式的发展史

“模式”起源于建筑，但同样适用于面向对象的软件设计。在软件开发中，我们用**对象和接口代替了墙壁和门窗**。



1995 年，由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 四人合著出版了一本名为《设计模式 - 可复用面向对象软件的基础》（Design Patterns - Elements of Reusable Object-Oriented Software）的书。

书中收录了 **23 个设计模式**，首次提到了软件开发中**设计模式**的概念。这是设计模式领域里程碑的事件，导致了软件设计模式的突破。

四位作者合称 **GOF**（四人帮，全拼 Gang of Four）。

设计模式的概念

设计模式（Design Pattern）

是一套被反复使用、经过分类编目的代码设计经验的总结。它描述了在软件设计过程中的一些不断重复发生的问题，以及该问题的核心解决方案。其目的是为了提高代码的可重用性、代码的可阅读性和代码的可靠性。

- 设计模式的本质是面向对象设计原则的实际运用，是对类的封装性、继承性和多态性以及类的关联关系和组合关系的充分理解。

设计模式的概念

设计模式的**四个基本要素**

要素	描述
模式名 Pattern Name	一个助记词，用一两个词来描述模式的问题、解决方案和效果。
问题 Problem	描述了该模式的应用环境，即何时使用该模式。它解释了设计问题和问题存在的前因后果，以及必须满足的一系列先决条件。
解决方案 Solution	描述了设计的组成成分、它们之间的相互关系及各自的职责和协作方式。解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合（类或对象的组合）来解决这个问题。
效果 Consequence	描述了模式的应用效果以及使用该模式应该权衡的问题，即模式的优缺点。主要是对时间和空间的衡量，以及该模式对系统的灵活性、扩充性、可移植性的影响等。

设计模式的作用

- 重用设计，重用设计比重用代码更有意义，它会自动带来代码的重用。
- 为设计提供共同的词汇，每个模式名就是一个设计词汇，其概念是的程序员间的交流更加方便。
- 在开发文档中采用模式词汇可以让其他人更容易理解你的想法和做法，编写开发文档也更方便。
- 应用设计模式可以让重构系统变得容易，可以确保开发正确的代码，并降低在设计或实现中出现错误的可能。
- 支持变化，可以为重写其他应用程序提供很好的系统架构。
- 正确使用设计模式，可以节省大量时间。

设计模式的作用

- 根据**目的**来分（模式是用来完成什么工作）
  - 创建型模式：用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”。
  - 结构型模式：用于描述如何将类或对象按某种布局组成更大的结构。
  - 行为型模式：用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责。
- 根据**范围**来分（模式主要用于类还是对象）
  - 类模式：用于处理类与子类之间的关系，这些关系通过继承来建立，是静态的，在编译时刻便确定下来了。
  - 对象模式：用于处理对象之间的关系，这些关系可以通过组合或聚合来实现，在运行时刻是可以变化的，更具动态性。

设计模式的分类

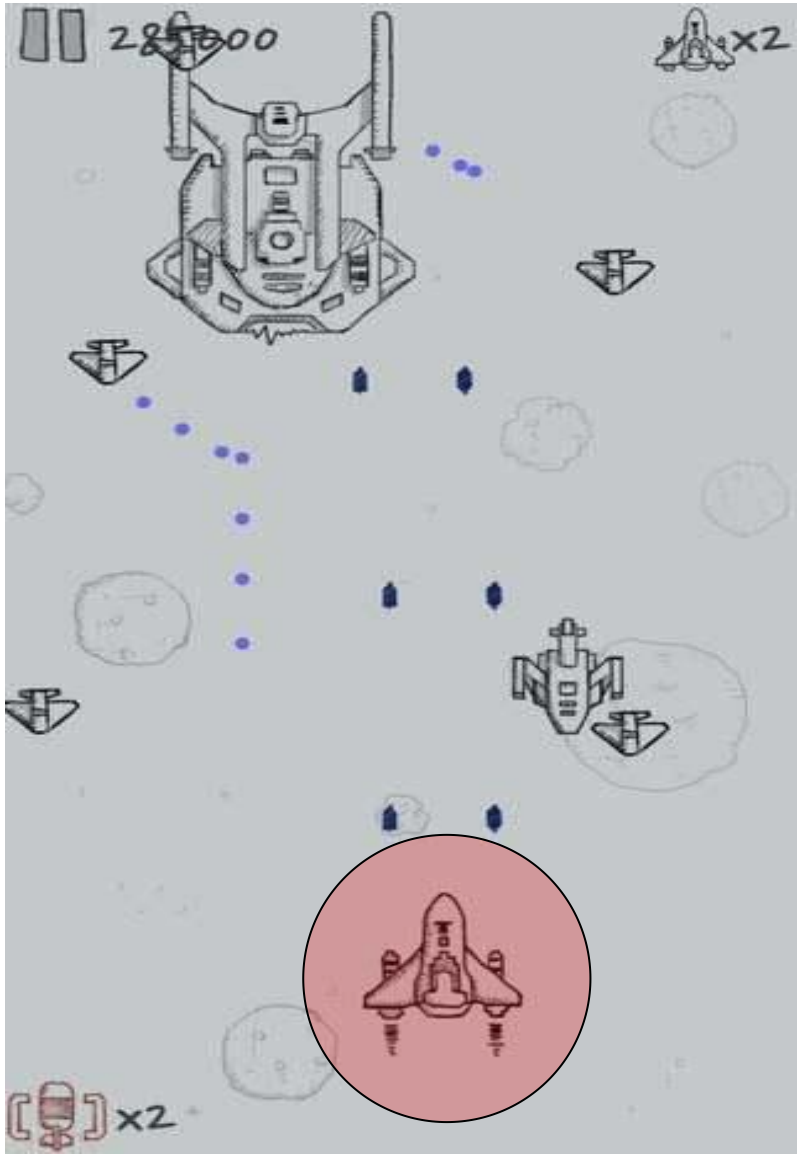
GoF 23种设计模式的分类图:

- MVC模式
- 数据访问对象模式
- 生产者、消费者模式

		目的		
		创建型	结构型	行为型
范围	类	工厂方法	适配器（类）	模板方法、解释器
	对象	抽象工厂 生成器 原型 单例	适配器（对象） 桥接 组合 装饰 外观 享元 代理	职责链 命令 迭代器 中介者 备忘录 观察者 状态 策略 访问者

课程导航

- 面向对象设计原则
- 设计模式
- 单例模式
- 简单工厂模式
- 工厂模式
- 抽象工厂设计模式



单例模式

- **单例模式**（Singleton Pattern）

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

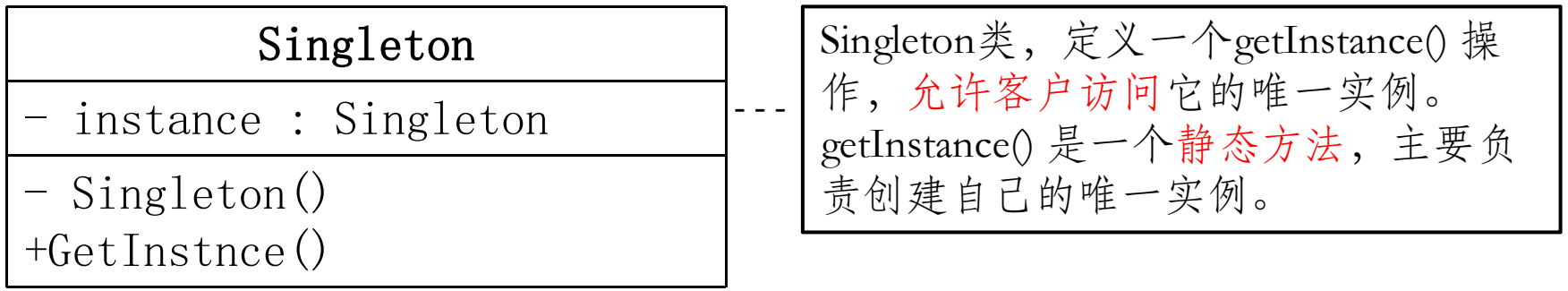
- 很多时候我们都要保证一个类只有一个实例
  - 整个应用程序中，只有一个连接数据库的connection实例
  - 操作系统只有一个时钟
  - 飞机大战游戏中只有一个英雄机
  - …

单例模式

如何保证一个类只有一个实例，且这个实例易于被访问呢?

全局变量使得一个对象可以被全局访问，但它不能防止你实例化多个对象。一个最好的办法就是让类自身负责保存它的唯一实例，这个类可以保证没有其他实例可以被创建，并且它可以提供一个访问该实例的方法。

单例模式（Singleton）结构图



- “+”表示 public;
- “-”表示 private;
- “#”表示 protected;
- 不带符号表示 default。

单例模式（1）：饿汉式（Eager Singleton）

经典单例模式实现

Singleton类，定义一个getInstance() 操作，允许客户访问它的唯一实例。getInstance() 是一个静态方法，主要负责创建自己的唯一实例。

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

    private Singleton(){
    }

    public static Singleton GetInstance () {
      return singleton;
    }
}
```

1.类初始化时，立即加载这个对象，天然的线性安全保障

2. 把构造方法声明为私有，使外界无法利用new创建此类实例

3.方法无需同步，调用效率高

单例模式（2）：懒汉式（Lazy Singleton）

经典单例模式实现

Singleton类，定义一个getInstance() 操作，允许客户访问它的唯一实例。getInstance() 是一个静态方法，主要负责创建自己的唯一实例。

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}

    public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
    }
}
```

1. 利用一个私有的静态变量来记录Singleton类的唯一实例

2. 把构造方法声明为私有，使外界无法利用new创建此类实例

3. 利用公有的静态方法getInstance()来实例化对象，并返回该唯一实例

单例模式（3）：同步锁

多线程同时访问Singleton类， 调用getInstance()方法，则有可能创建多个实例，如何避免?

通过增加**synchronized**关键字到 getInstance() 方法中，迫使每个线程在进入这个方法之前， 要先等候别的线程离开该方法。也就是说， 不会有两个线程可以同时进入这个方法。

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

线程安全， 但效率低!

课外拓展：双重检验锁

单例模式（3）：同步锁

多线程同时访问Singleton类， 调用getInstance()方法，则有可能创建多个实例，如何避免？

双重检验锁

```
public class Singleton {
    // 关键1：使用volatile关键字禁止指令重排序
    private static volatile Singleton instance;

    private Singleton() {} // 私有构造函数

    public static Singleton getInstance() {
        // 第一重检验：检查实例是否已存在（无需加锁，性能关键！）
        if (instance == null) {
            // 只有第一次创建实例时，才会进入同步块
            synchronized (Singleton.class) {
                // 第二重检验：进入同步块后再次检查
                if (instance == null) {
                    instance = new Singleton(); // 创建实例
                }
            }
        }
        return instance;
    }
}
```

单例模式

单例模式总结：

要素	描述
模式名 Pattern Name	单例模式（Singleton Pattern）
目的 Intent	只希望有一个对象，系统所有地方都可以访问这个对象，且不使用全局变量，也不需要传递对象的引用。
问题 Problem	几个不同的客户对象都希望引用同一个对象，如何保证?
解决方案 Solution	保证一个唯一实例 1）定义私有的静态成员变量保存单实例的引用；2）定义公有的静态方法getInstance() 获取唯一实例；3）该类自己负责“第一次使用时”实例化对象。
效果 Consequence	优点：1）对唯一实例的受控访问；2）在内存里只有一个实例，减少了内存的开销，避免频繁地创建销毁对象，提高性能；3）避免对共享资源的多重占用；4）允许可变数目的实例；5）可以全局访问。

课程导航

- 面向对象设计原则
- 设计模式
- 单例模式
- 简单工厂模式
- 工厂模式
- 抽象工厂设计模式

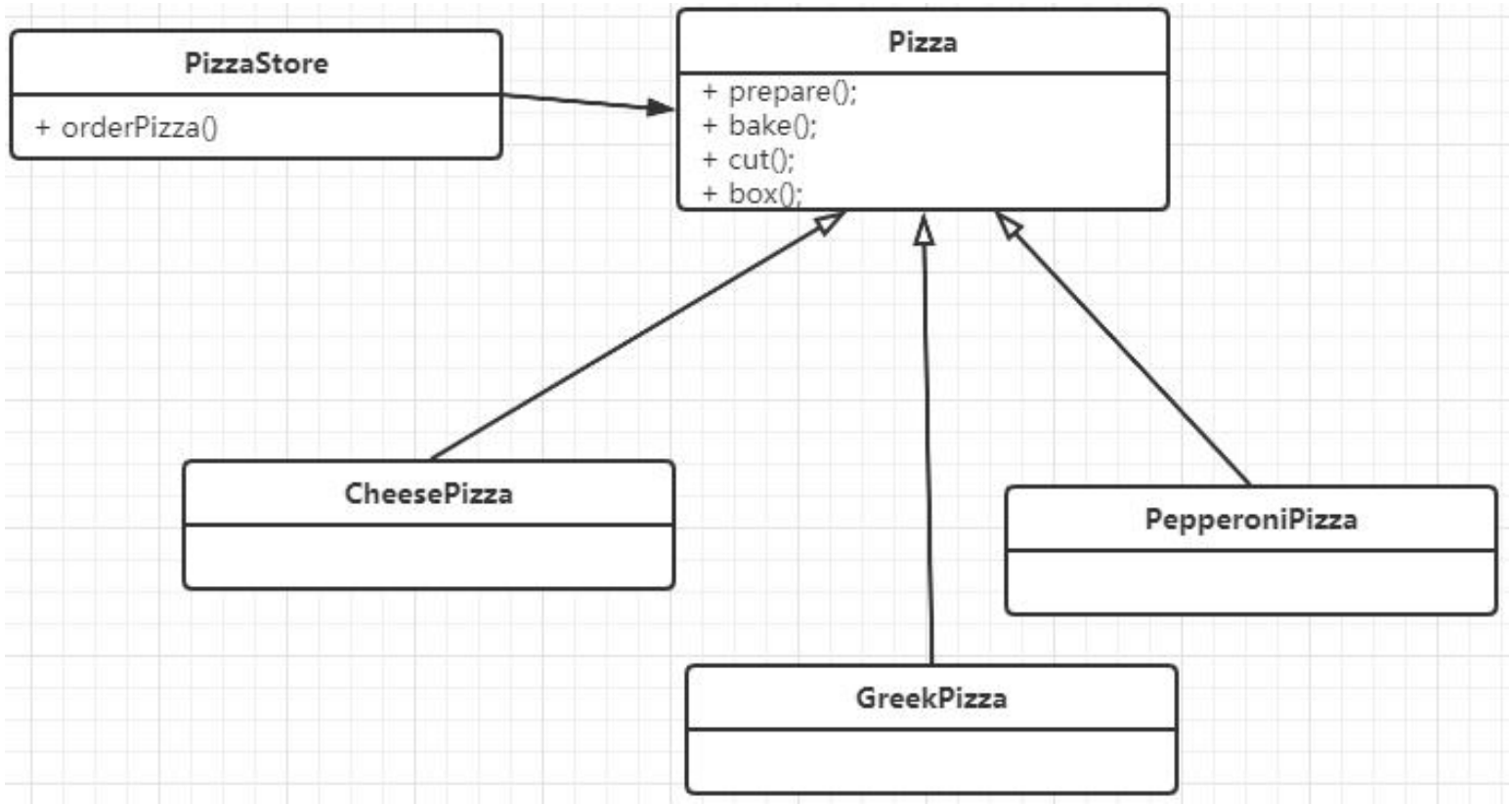
案列背景

- 我们在大学城食堂开了家简单的披萨店，向附近的学生提供多种口味的比萨：
  - 芝士比萨（CheesePizza）
  - 希腊比萨（GreekPizza）
  - 腊肠比萨（PepperoniPizza）

- 请设计一个模拟的比萨店
  - 软件系统

案列背景

设计UML图：



为了让系统更有弹性，我们希望Pizza是一个抽象类或接口，可以利用面向对象的继承机制。

案列背景

代码实现：

```
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;
}
```

通过参数传入Pizza类型

根据pizza类型，实例化正确的具体类，然后将其赋值给Pizza实例变量。

一旦我们获取了一个具体的Pizza类的实例，就可以调用它的方法来来做一些准备，然后烘烤、切片、装盒。

案列背景

根据市场变化，如何增加新口味的比萨或下架不受客户欢迎的比萨？在前面的设计中，哪些是不变化的？哪些可能是变化的？

```
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();
    }else if (type.equals("clam")) {
        pizza = new ClamPizza();
    }else if (type.equals("veggie")) {
        pizza = new VeggiePizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}
```

这是变化的部分，代码随着比萨菜单的改变而变化，没有对修改封闭，违反OCP原则（开闭原则）。

这是不变化的部分，代码保持不变。

案列背景

修改设计，封闭创建对象的代码：

- 将创建比萨对象的代码从orderPizza()方法种抽离
- 定义一个新的类SimplePizzaFactory，专门负责创建比萨对象

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

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        }else if (type.equals("greek")) {
            pizza = new GreekPizza();
        }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;
    }
}
```

定义createPizza()方法，使用这个方法来实例化新对象。

这是从orderPizza() 方法中移出来的代码，代码没有什么变动。

案列背景

修改设计，封闭创建对象的代码：

- 重做PizzaStore类

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }
}
```

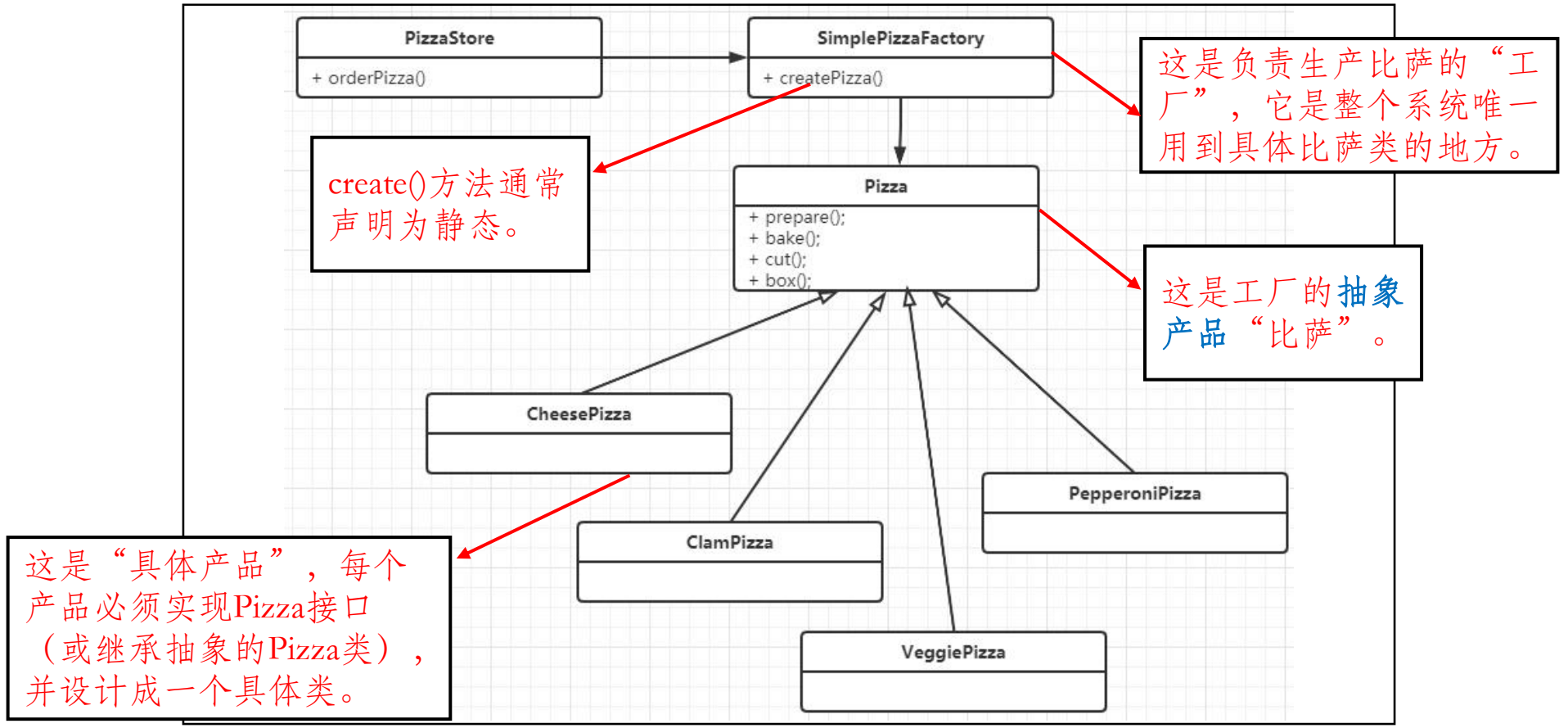
PizzaSore的构造器，需要一个工厂作为参数。

orderPizza() 方法通过简单传入订单类型来使用工厂创建比萨。

注意：我们把new操作符替换成工厂对象的创建方法。这里不再使用具体实例化。

简单工厂模式

简单工厂模式 (SimpleFactory Pattern) 并不是一个真正意义上设计模式，但工厂模式是建立在它之上。



单选题 1分

设置

简单工厂模式是否满足开闭原则?

A 是

B 否

C 不清楚

提交

课程导航

- 面向对象设计原则
- 设计模式
- 单例模式
- 简单工厂模式
- 工厂模式
- 抽象工厂设计模式

工厂方法模式

需求继续变化…

- 比萨店经营有成，接下来的计划就是在多个不同城市开设加盟店
  - 质量控制：确保每个加盟店营运质量
    - ✓ 要求每个加盟店采用统一的标准方式制作比萨
    - ✓ （准备、烘烤、切片、装盒）
  - 区域差异：每个加盟店需要适应当地口味
    - ✓ 纽约（NY）比萨工厂，生产纽约风味的各种比萨
    - ✓ 芝加哥（Chicago）比萨工厂，生产芝加哥风味的各种比萨

结论：需要建立一个框架，把PizzaStore和创建Pizza捆绑在一起， 同时又保持一定的弹性。

工厂方法模式

如何让披萨制作活动局限于PizzaStore类，而同时又能让这些加盟店可以自由地制作该地区的风味?

把createPizza()方法放回到PizzaStore中， 并设置为抽象方法，然后为每个区域创建一个PizzaStore的子类。

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {

        Pizza pizza;

        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box;

        return pizza;
    }

    abstract Pizza createPizza(String type);
}
```

createPizza()方法从工厂对象移回到PizzaStore。

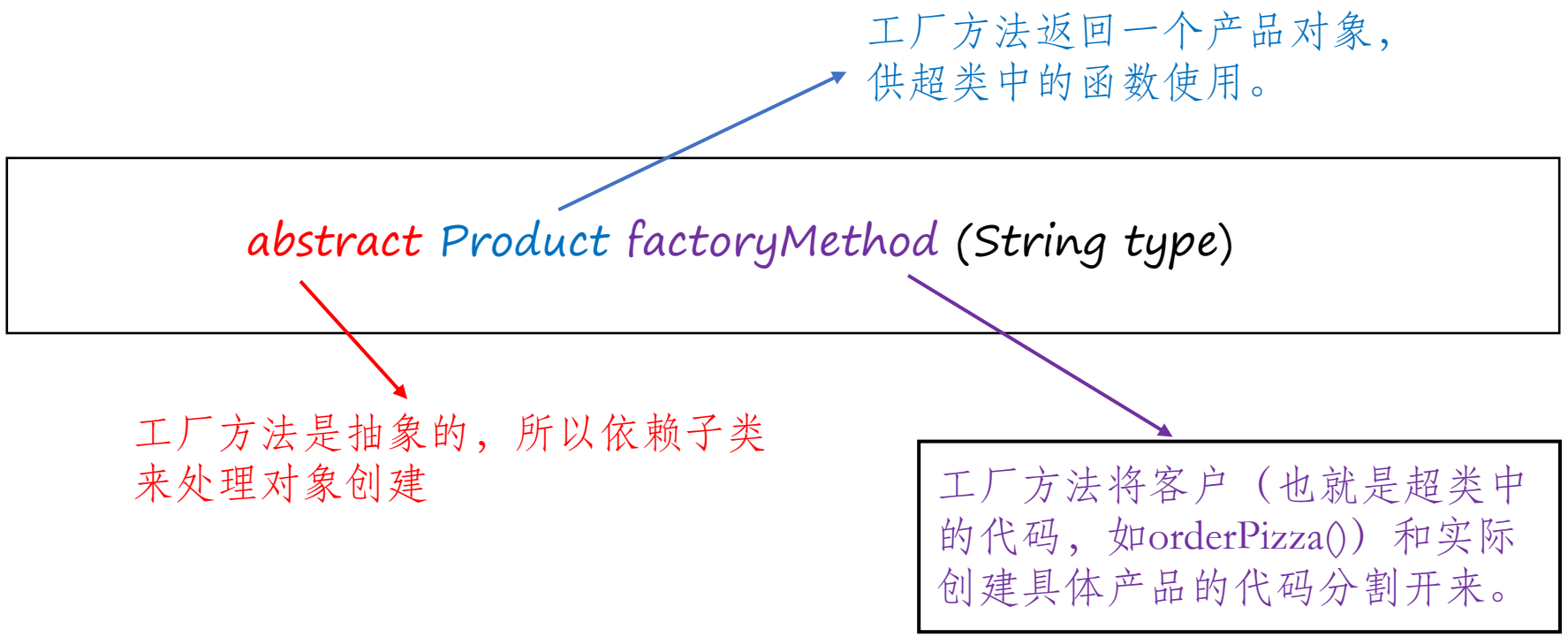
这些功能仍然不变

这个抽象方法又叫做工厂方法，位于PizzaStore。

工厂方法模式

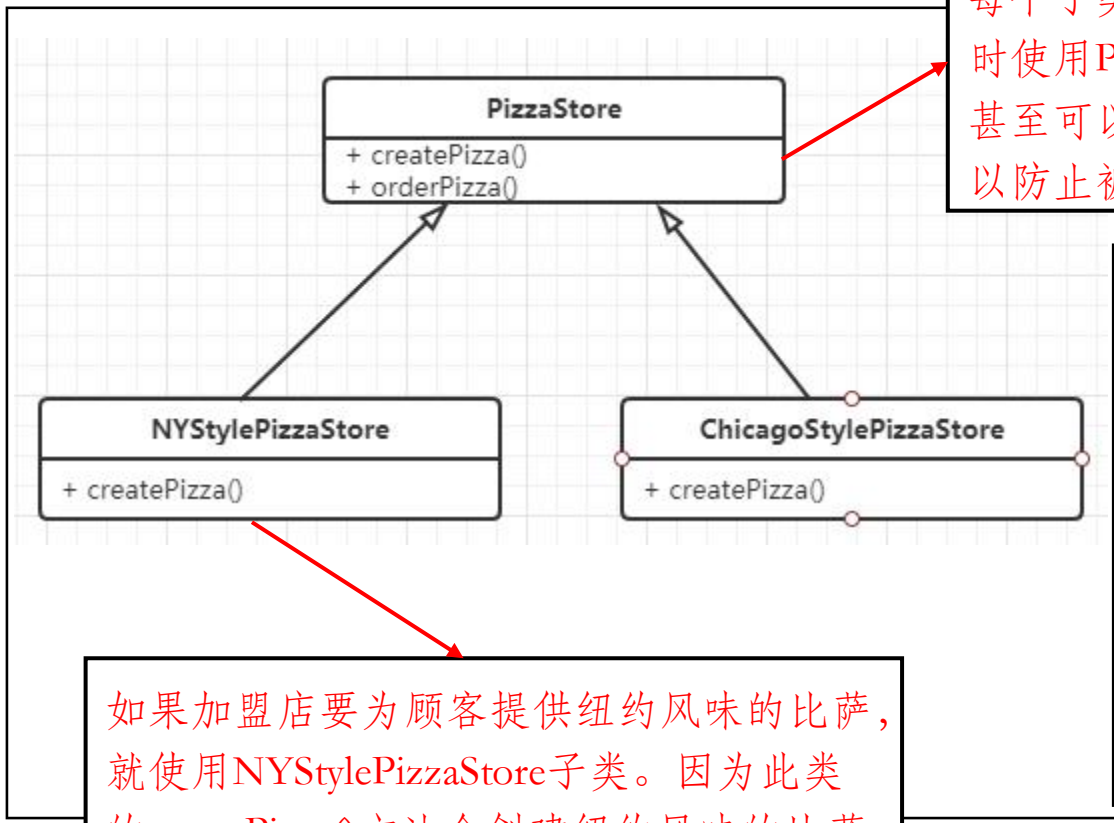
工厂方法（Factory Method）

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



工厂方法模式

如何让披萨制作活动局限于PizzaStore类，而同时又能让这些加盟店可以自由地制作该地区的风味？



每个子类都会覆盖createPizza()方法，同时使用PizzaStore定义的orderPizza()方法。甚至可以把orderPizza()方法声明为final，以防止被子类覆盖。

orderPizza()方法定义在抽象的PizzaStore类内，而不是在子类中。它通过调用createPizza()取得比萨对象，但它并不知道实际是哪个比萨店子类来制作比萨，它只负责后续的准备、烘烤等工作。比萨的类型是由具体的比萨店子类来决定。

这就是“解耦”(decouple)！

如果加盟店要为顾客提供纽约风味的比萨，就使用NYStylePizzaStore子类。因为此类的createPizza()方法会创建纽约风味的比萨。

工厂方法模式

比萨店子类（纽约分店）的实现：

```
public class NYPizzaStore extends PizzaStore {

    @Override
    Pizza createPizza(String type) {
        Pizza pizza = null;

        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("veggie")) {
            pizza = new NYStyleVeggiePizza();
        }

        return pizza;
    }
}
```

NYPizzaStore继承PizzaStore，所以拥有orderPizza()方法。

createPizza()方法返回一个抽象的Pizza对象，由子类全权负责该实例化哪一个具体类型的Pizza。

子类必须实现createPizza()方法，创建具体的比萨类。对于每一种披萨，都是创建纽约风味。

注意：超类PizzaStore的orderPizza()方法并不知道正在创建的比萨是哪一种，它只知道这个比萨可以被准备、烘烤、切片、装盒！

工厂方法模式

比萨超类的实现：

```
public abstract class Pizza {

    String name;
    String dough;
    String sauce;
    ArrayList toppings = new ArrayList();

    void prepare() {
        System.out.println("Preparing: " + name);
        System.out.println("Tossing dough...");
        System.out.println("Adding sauce...");
        System.out.println("Adding toppings: ");
        for(int i=0;i<toppings.size();i++) {
            System.out.println("   "+toppings.get(i));
        }
    }

    void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }

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

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

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

每个比萨都具有名称、面团类型、酱料类型和一套佐料。

准备工作需要以特定的顺序进行，有一连串的步骤。

Pizza 抽象类提供了某些默认的基本做法，用来进行烘烤、切片、装盒。

工厂方法模式

比萨子类的实现：

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

纽约风味的比萨有自己的大蒜番茄酱和薄饼。上面覆盖意大利reggiano高级干酪。

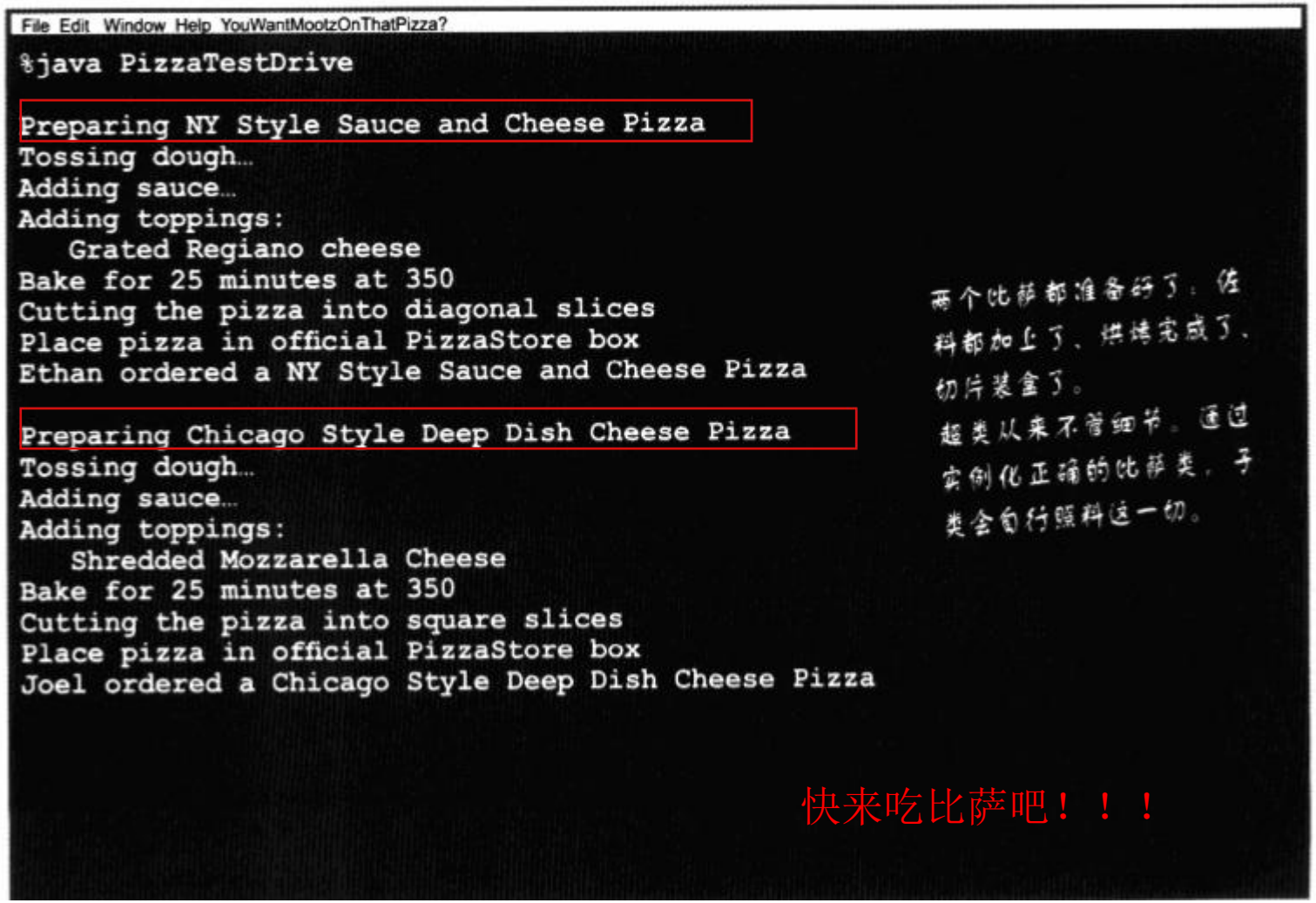
```
public class ChicagoStyleCheesePizza extends Pizza {
    public ChicagoStyleCheesePizza() {
        name = "Chicago Style Deep Dish Cheese Pizza";
        dough = "Extra Thick Crust Dough"
        sauce = "Plum Tomato Sauce"
        toppings.add("Shredded Mozzarella Cheese");
    }
}
```

芝加哥风味的比萨使用小番茄作为酱料，并使用厚饼。上面覆盖意大利白干酪。

订购比萨

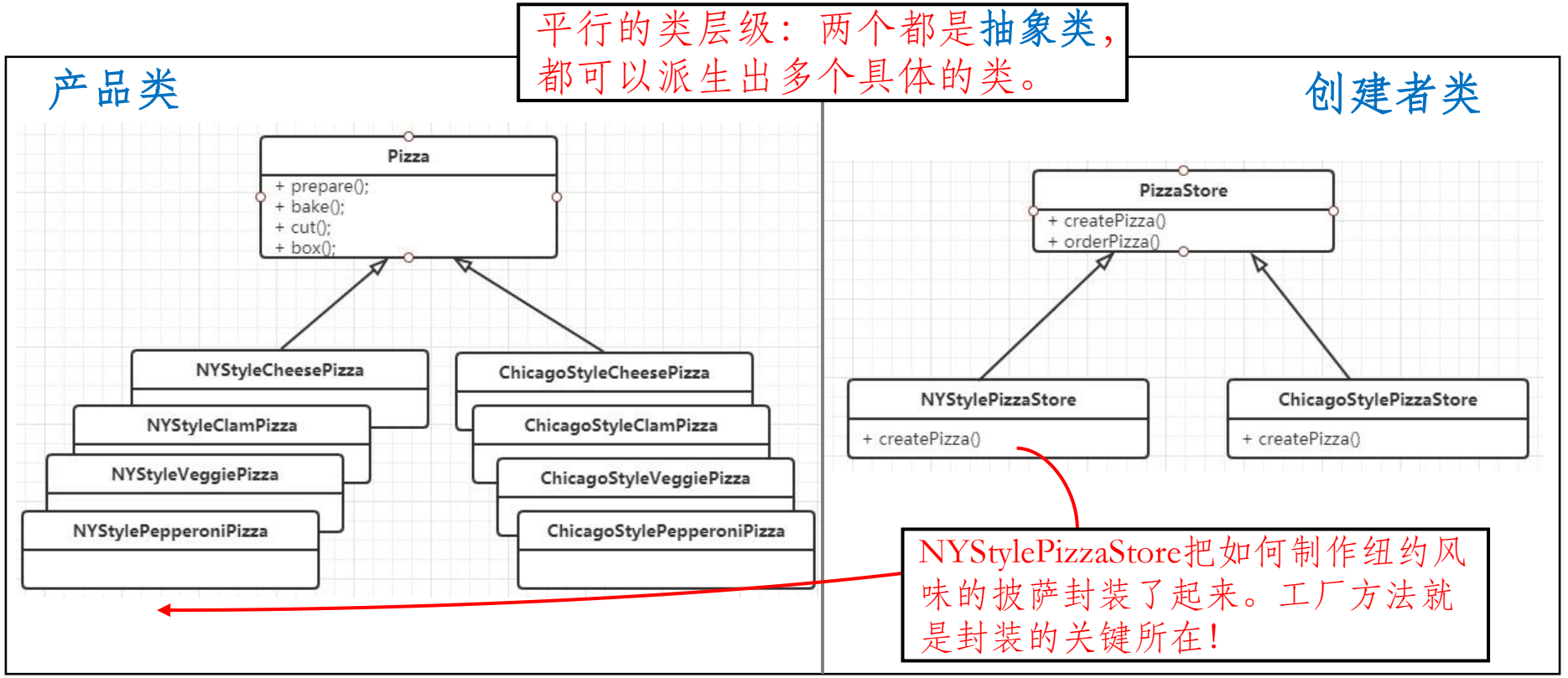
```
public class PizzatestDrive {

    public static void main(String[] args) {
        PizzaStore nyStore = new NYStylePizzaStore();
        PizzaStore chicagoStore = new ChicagoStylePizzaStore();

        Pizza pizza = nyStore.orderPizza("cheese");
        System.out.println("Ethan ordered a " + pizza.getName() +"\n");

        pizza = chicagoStore.orderPizza("cheese");
        System.out.println("Joel ordered a " + pizza.getName() +"\n");
    }
}
```

工厂方法模式


File Edit Window Help YouWantMootzOnThatPizza?
%java PizzaTestDrive
Preparing NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
Grated Regiano cheese
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box
Ethan ordered a NY Style Sauce and Cheese Pizza
Preparing Chicago Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
Shredded Mozzarella Cheese
Bake for 25 minutes at 350
Cutting the pizza into square slices
Place pizza in official PizzaStore box
Joel ordered a Chicago Style Deep Dish Cheese Pizza
两个比萨都准备好了，佐料都加上了，烘烤完成了，切片装盒了。
超类从来不管细节。通过实例化正确的比萨类，子类会自行照料这一切。
快来吃比萨吧！！！

工厂方法模式

修改设计，封装具体类型的实例化：

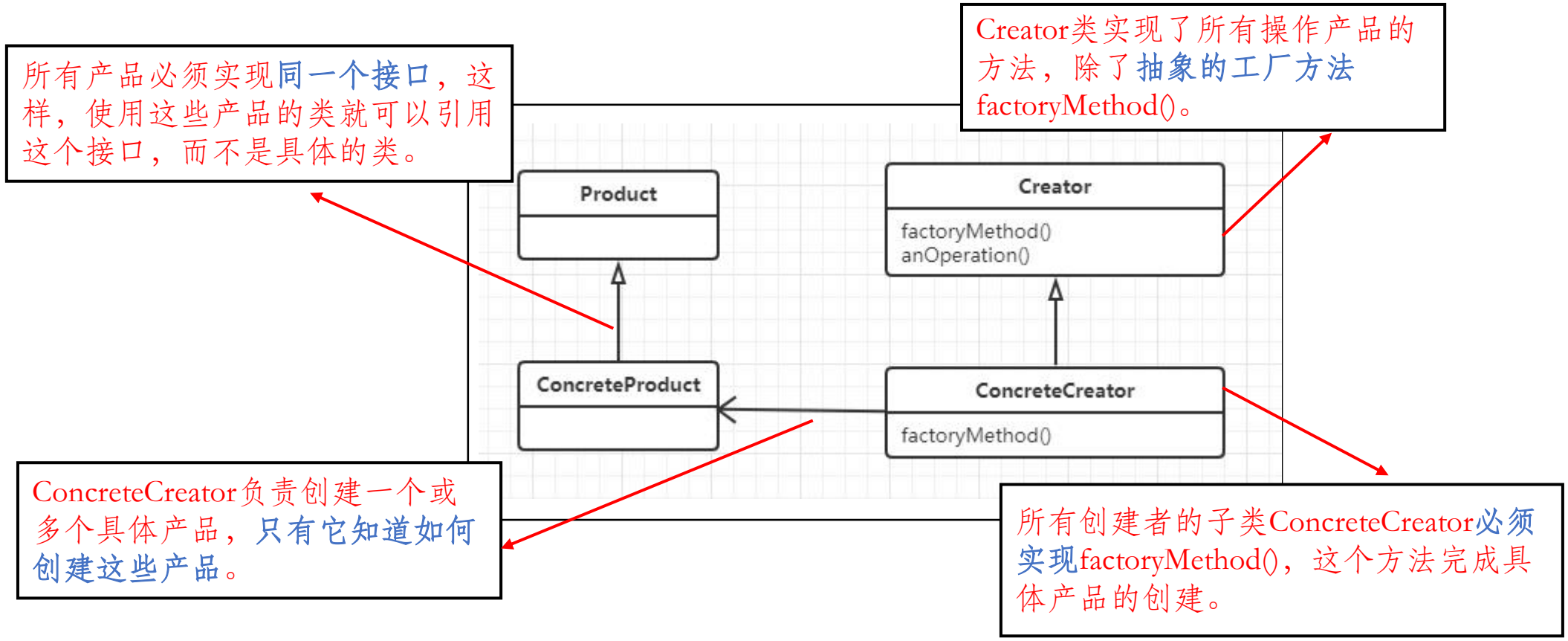


框架结构：将orderPizza()方法和一个工厂方法联合起来，并且工厂方法将产品制作的细节封装进各个创建者。

工厂方法模式

- **工厂方法模式**（Factory Method Pattern）

定义了一个创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。



简单工厂模式 VS 工厂方法模式

- 简单工厂模式
  - 创建对象的逻辑判断放在了工厂类中，客户不感知具体的类，但是其违背了开闭原则，如果要增加新的披萨类型，就必须修改工厂类。

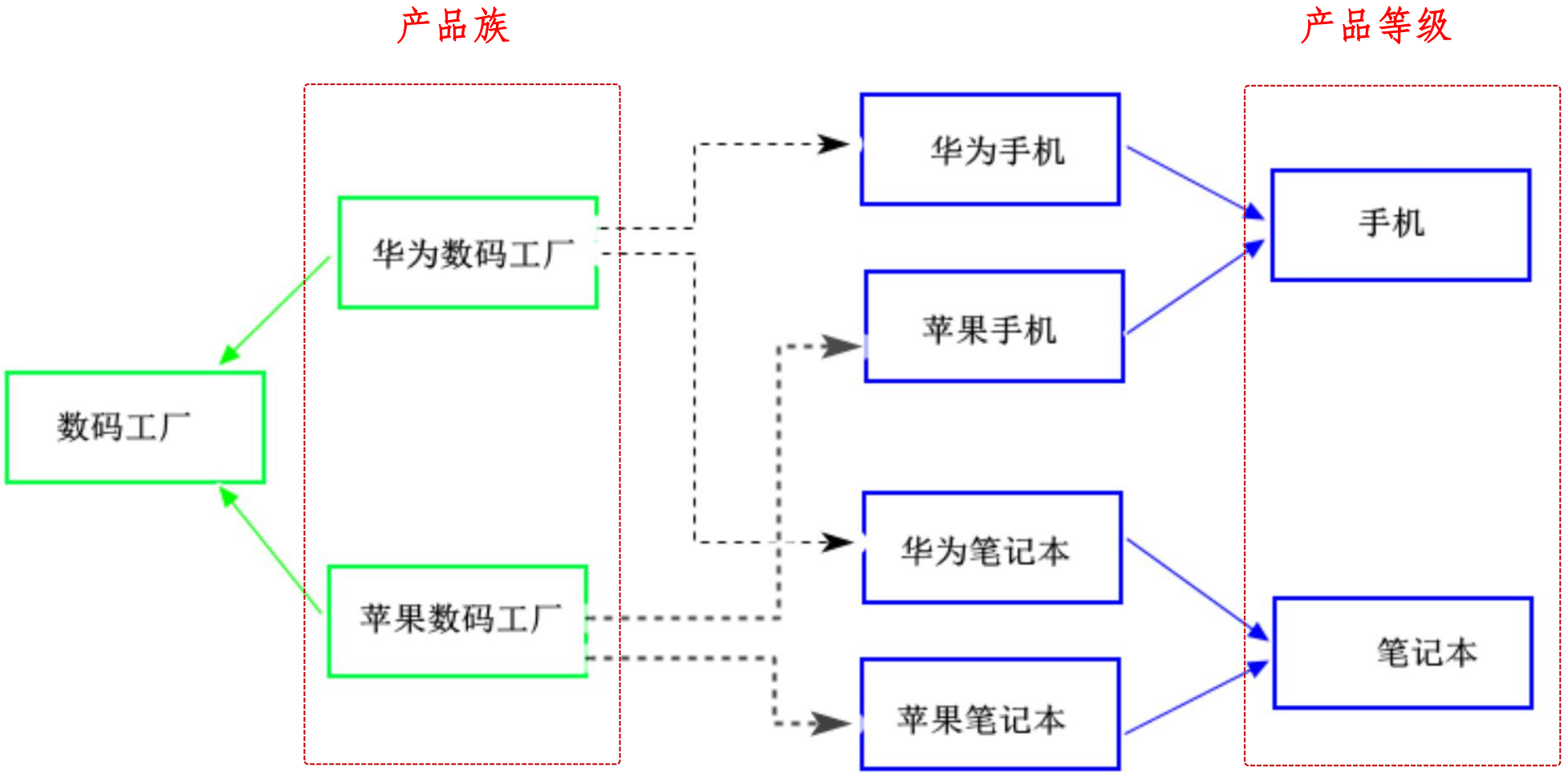
- 工厂方法模式
  - 通过扩展来新增具体类的，符合开闭原则，但是在客户端就必须要感知到具体的工厂类，也就是将判断逻辑由简单工厂的工厂类挪到客户端。
  - 工厂模式横向扩展很方便，假如又有新区域的加盟店，那么只需要创建相应的披萨店子类和披萨子类去实现抽象工厂接口和抽象产品接口即可，而不用去修改原有已经存在的代码。

课程导航

- 面向对象设计原则
- 设计模式
- 单例模式
- 简单工厂模式
- 工厂模式
- 抽象工厂设计模式

案例背景

- 理解产品族和产品等级



抽象工厂模式概述

- 抽象工厂模式（Abstract Factory Pattern）是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

- 提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。

案例背景

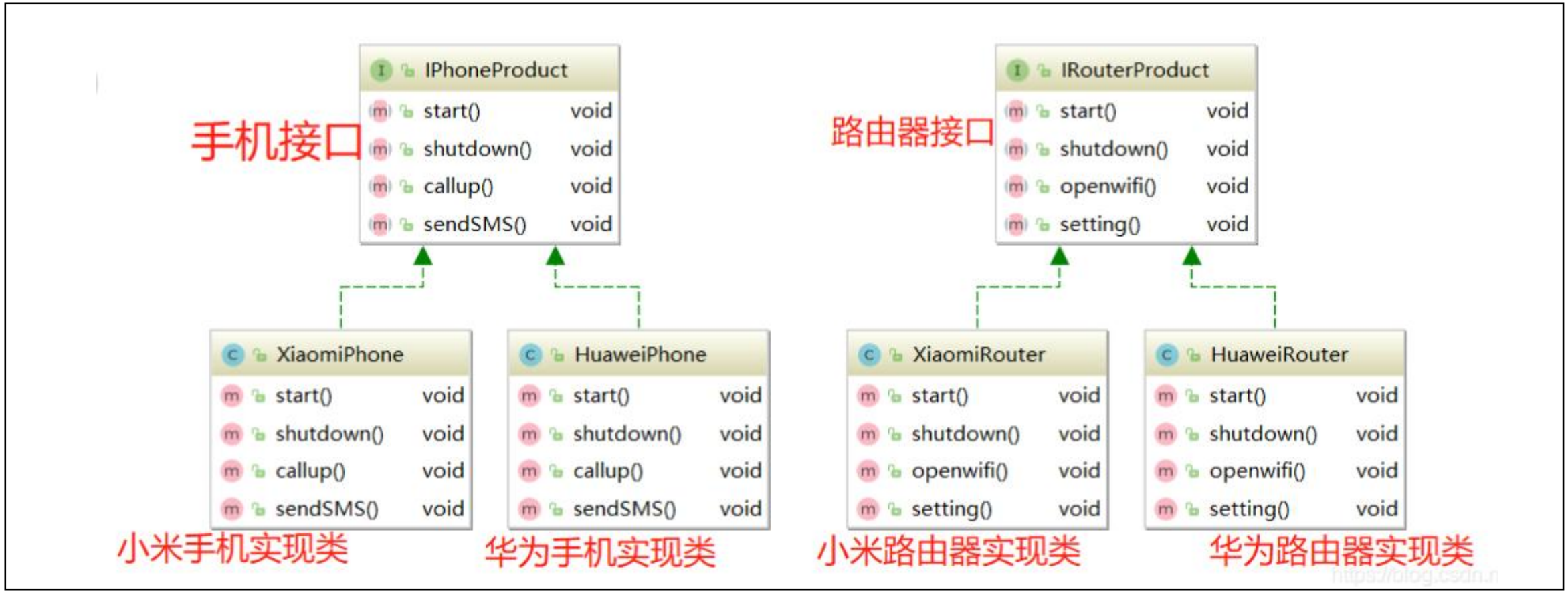
- 有手机和路由器两种产品，有华为和小米两种品牌，两种品牌都可以生产手机和路由器；
- 有手机和路由器两种产品，定义2个接口；
- 小米和华为都可以生产这两种产品，所以有4个实现类；
- 现在需要创建华为和小米的工厂类，先将工厂类进行抽象，里面有创建两个产品的方法，返回的是产品的接口类；
- 创建华为和小米的工厂实现类，继承工厂类接口，实现创建各自产品的方法；
- 客户端调用时，直接用工厂接口类创建需要的工厂，拿到对应的产品。

抽象工厂模式

UML类图分析：

抽象工厂模式

UML类图分析：



案列背景

代码实现：

```
1  //手机产品接口
2  public interface IPhoneProduct {
3      //开机
4      void start();
5      //关机
6      void shutdown();
7      //打电话
8      void callup();
9      //发邮件
10     void sendSMS();
11 }
12
13 //路由器产品接口
14 public interface IRouterProduct {
15     //开机
16     void start();
17     //关机
18     void shutdown();
19     //打开wifi
20     void openwifi();
21     //设置
22     void setting();
23 }
24
```

完整代码见课程qq群, Lecture5

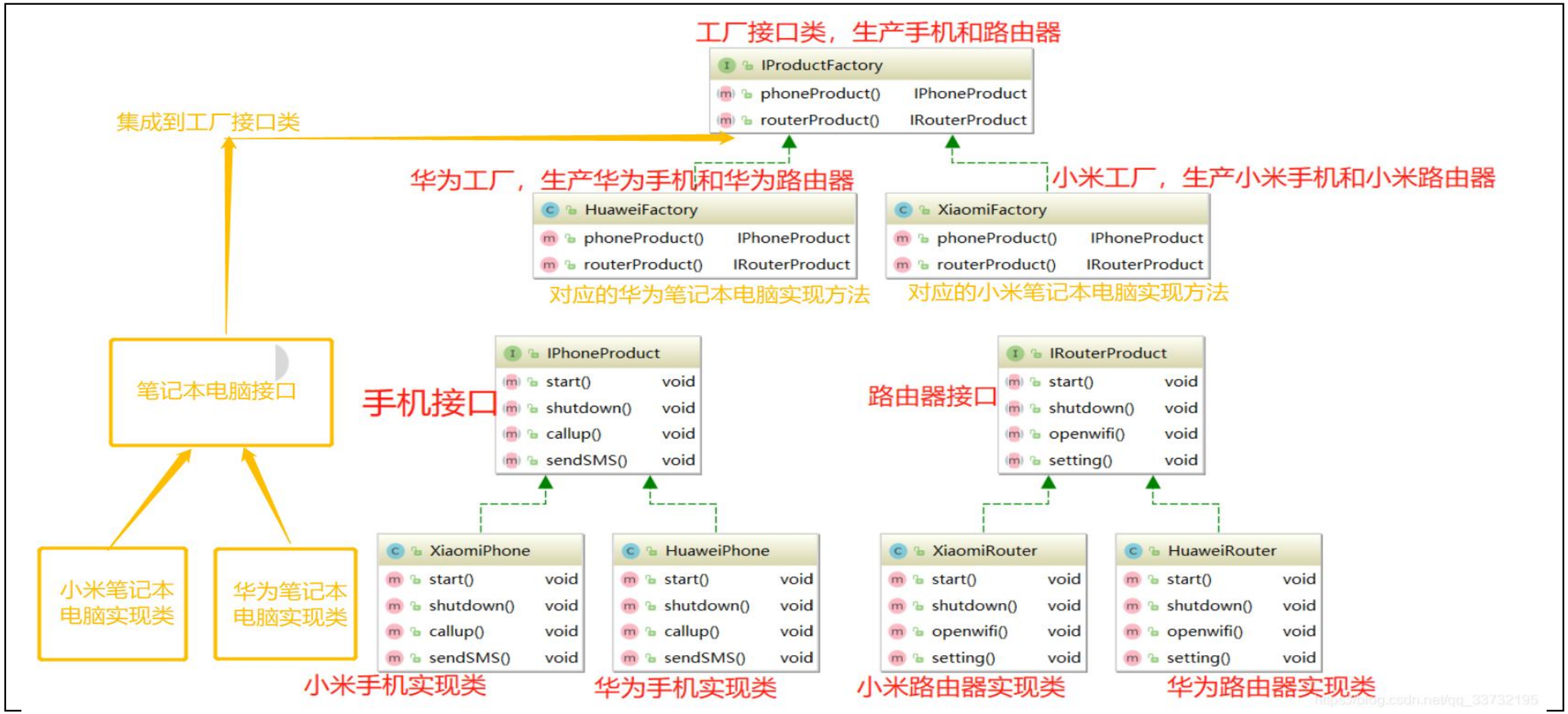
抽象工厂模式

以上就是抽象工厂模式的实现，现在回看产品族和产品等级的概念，如果新增一个产品族或产品等级会怎样?

拓展一个产品等级

- 我们会发现，拓展一个产品等级是非常困难的，例如产品等级中**新增笔记本电脑**，也就是说华为和小米现在可以生产电脑了，如下页图片所示（黄色字体为新增一个产品族需要做的事），对顶层的工厂接口类也要修改，这是非常麻烦的。

抽象工厂模式

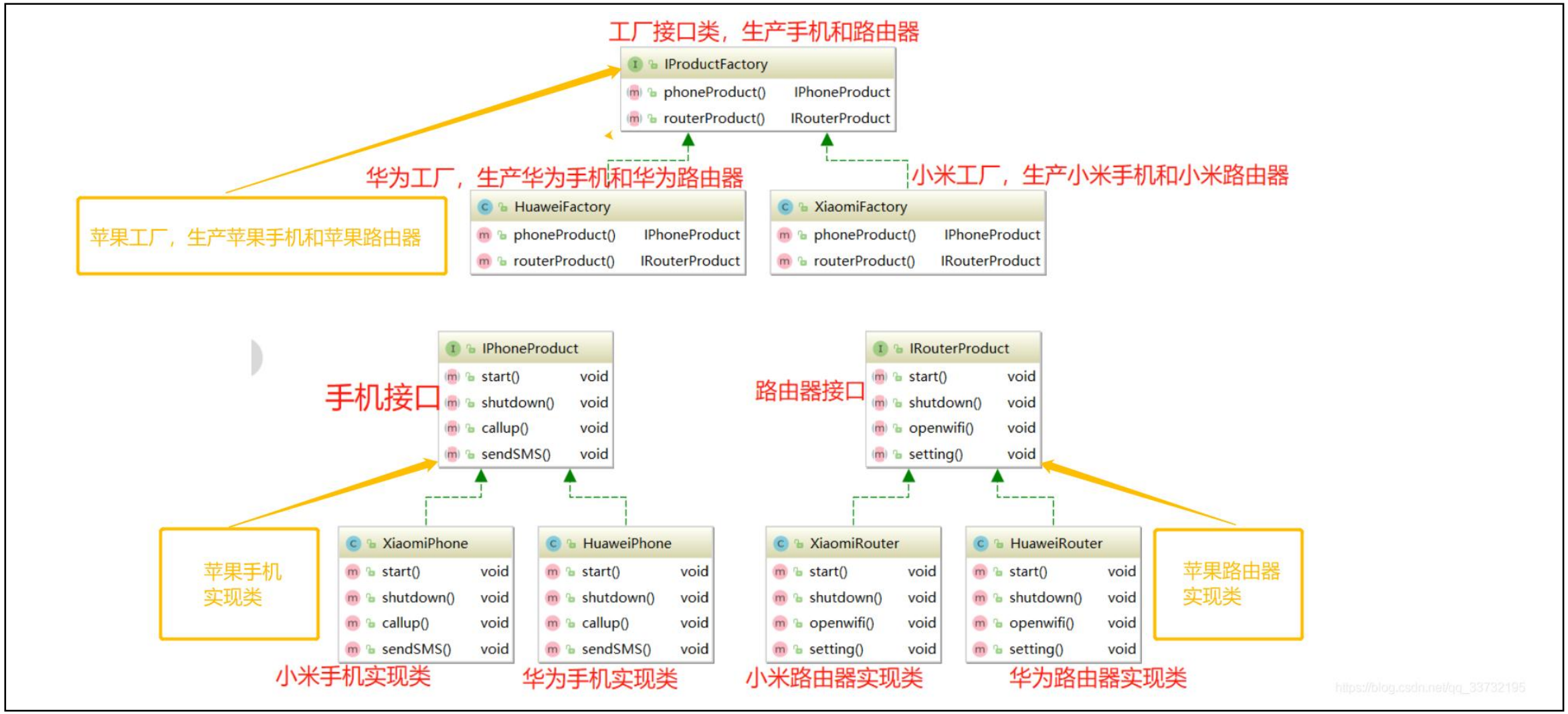

工厂接口类，生产手机和路由器
IProductFactory
phoneProduct() IPhoneProduct
routerProduct() IRouterProduct
集成到工厂接口类
华为工厂，生产华为手机和华为路由器
小米工厂，生产小米手机和小米路由器
HuaweiFactory
phoneProduct() IPhoneProduct
routerProduct() IRouterProduct
XiaomiFactory
phoneProduct() IPhoneProduct
routerProduct() IRouterProduct
对应的华为笔记本电脑实现方法
对应的小米笔记本电脑实现方法
笔记本电脑接口
手机接口
IPhoneProduct
start() void
shutdown() void
callup() void
sendSMS() void
路由器接口
IRouterProduct
start() void
shutdown() void
openwifi() void
setting() void
小米笔记本电脑实现类
华为笔记本电脑实现类
XiaomiPhone
HuaweiPhone
XiaomiRouter
HuaweiRouter
小米手机实现类
华为手机实现类
小米路由器实现类
华为路由器实现类

抽象工厂模式

拓展一个产品族

- 如果扩展一个产品族，例如新增一个**手机品牌**，也就是说新增一个厂商来生产手机，如下页图片所示（黄色字体为新增一个产品等级需要做的事），新增一个产品族不用修改原来的代码，符合OCP原则，十分方便。

抽象工厂模式


工厂接口类，生产手机和路由器
IProductFactory
phoneProduct() IPhoneProduct
routerProduct() IRouterProduct
华为工厂，生产华为手机和华为路由器
小米工厂，生产小米手机和小米路由器
苹果工厂，生产苹果手机和苹果路由器
HuaweiFactory
phoneProduct() IPhoneProduct
routerProduct() IRouterProduct
XiaomiFactory
phoneProduct() IPhoneProduct
routerProduct() IRouterProduct
手机接口
IPhoneProduct
start() void
shutdown() void
callup() void
sendSMS() void
路由器接口
IRouterProduct
start() void
shutdown() void
openwifi() void
setting() void
苹果手机
实现类
XiaomiPhone
start() void
shutdown() void
callup() void
sendSMS() void
HuaweiPhone
start() void
shutdown() void
callup() void
sendSMS() void
XiaomiRouter
start() void
shutdown() void
openwifi() void
setting() void
HuaweiRouter
start() void
shutdown() void
openwifi() void
setting() void
苹果路由器
实现类
小米手机实现类
华为手机实现类
小米路由器实现类
华为路由器实现类

抽象工厂模式优缺点

- 优点
  - 产品族扩展非常容易，只要新增具体工厂的实现和产品的实现。
  - 一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象（将一个系列的产品统一一起创建）。
- 缺点
  - 产品等级扩展非常困难，要新增一个产品类，既要修改工厂抽象类里加代码，又修改具体的实现类里面加代码。
  - 增加了系统的抽象性和理解难度。

投票

最多可选4项

设置

这个章节中不太清楚的内容是?

- A 面向对象的设计原则
- B 单例模式
- C 简单工厂模式
- D 工厂模式
- E 抽象工厂模式
- F 没有不清楚的地方

提交