Lesson9
完善类的设计_2

主讲老师：申雪萍

主要内容（1）

- 面向对象的几个基本原则
  - UML类图简介
  - 面向抽象（接口、抽象类）原则
  - 优先使用组合少用继承原则
  - 开-闭原则
  - 高内聚-低耦合原则
  - 其它原则

主要内容（2）

- 几个重要的设计模式
  - 设计模式简介
  - 策略模式
  - 访问者模式
  - 装饰模式
  - 适配器模式
  - 责任链模式
  - 门面模式
  - 简单工厂模式
  - 工厂方法模式
  - 抽象工厂模式

面向对象的几个基本原则

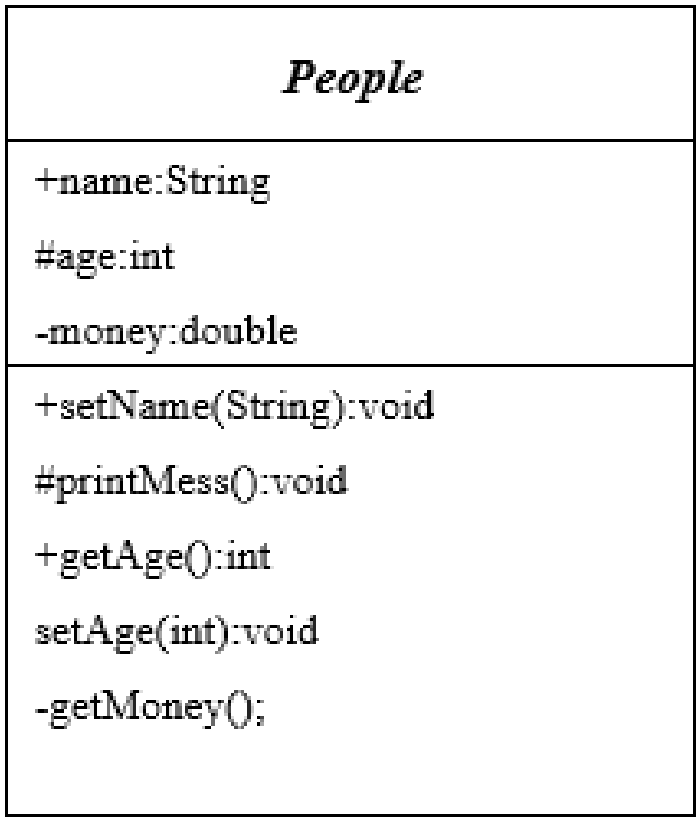
必要性（什么才是优秀的软件架构？）

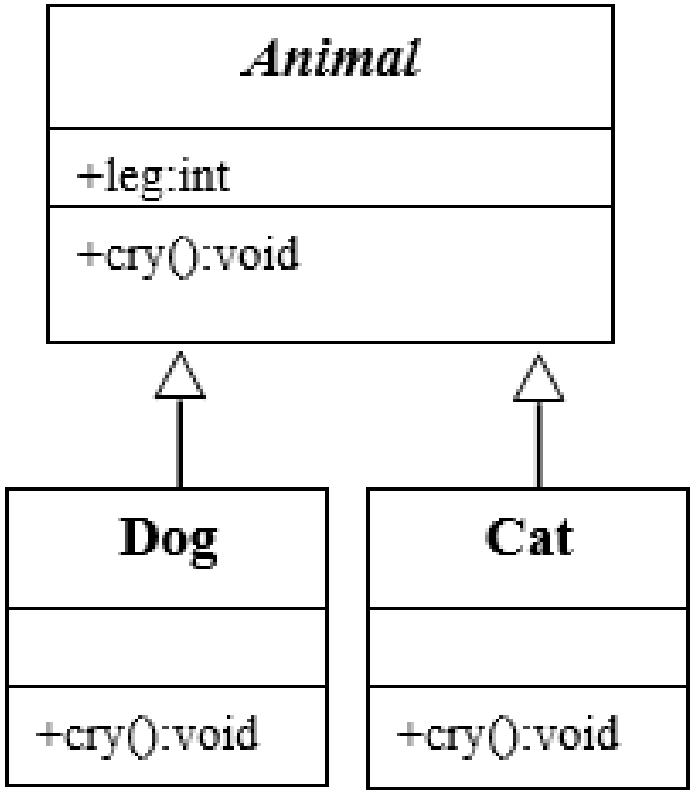
- 了解面向对象设计的基本原则，有助于知道如何合理使用面向对象语言编写出：
  - 低耦合、
  - 易维护、
  - 易扩展
  - 和易复用的程序代码。

类的UML图

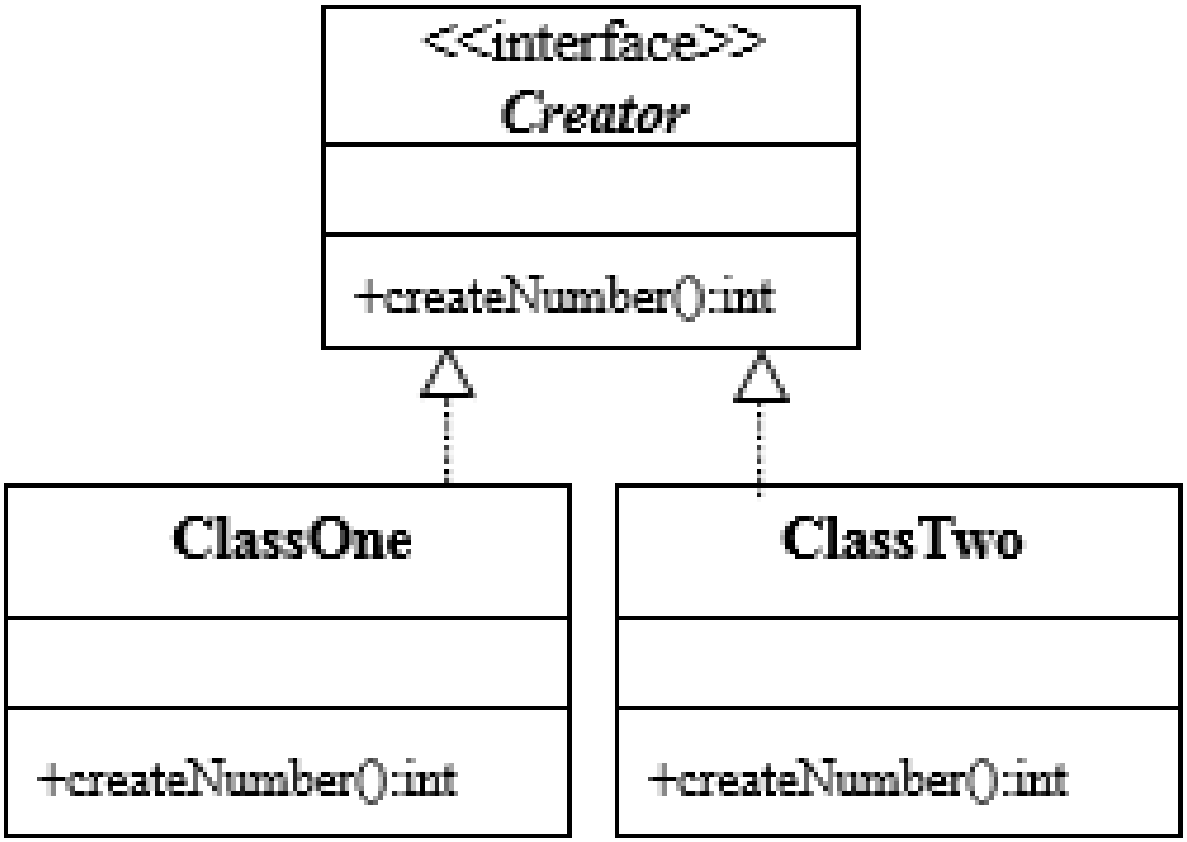
- 接口（Interface）
- 泛化关系（Generalization）
- 关联关系（Association）
- 依赖关系（Dependency）
- 实现关系（Realization）

类以及类的继承关系

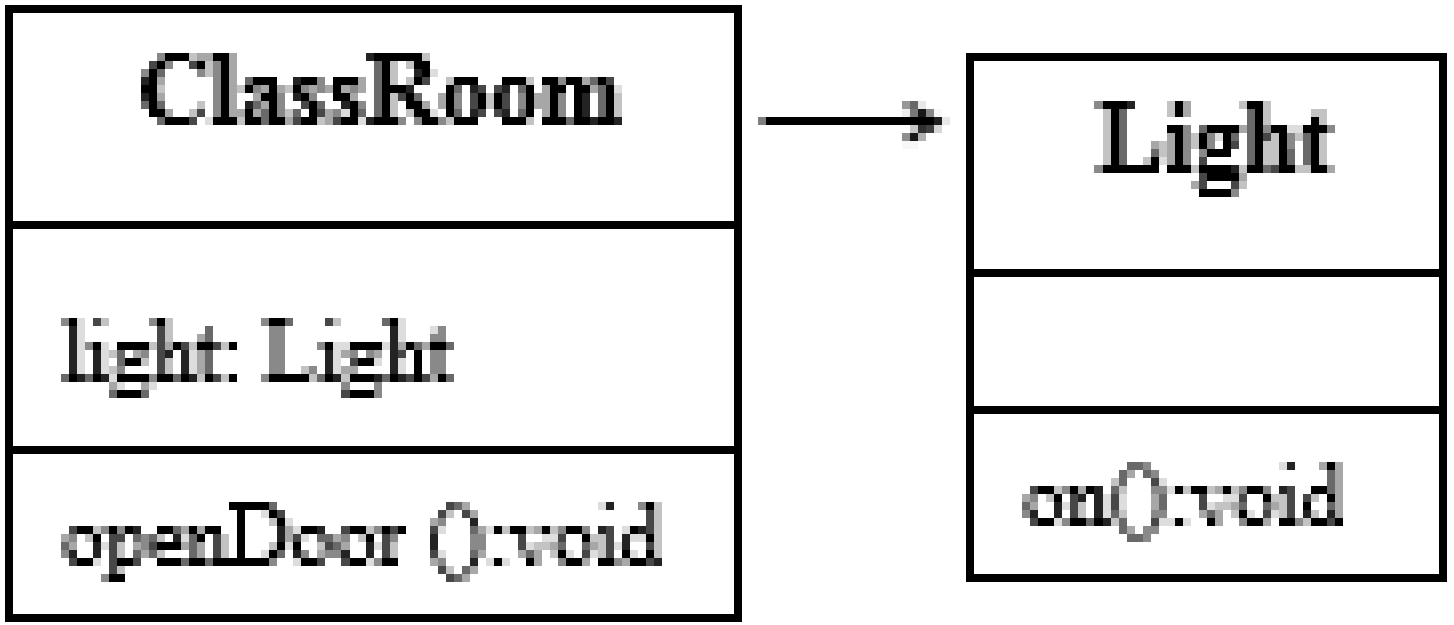

People
+name:String
#age:int
-money:double
+setName(String):void
#printMess():void
+getAge():int
setAge(int):void
-getMoney();


Animal
+leg:int
+cry():void
Dog
+cry():void
Cat
+cry():void

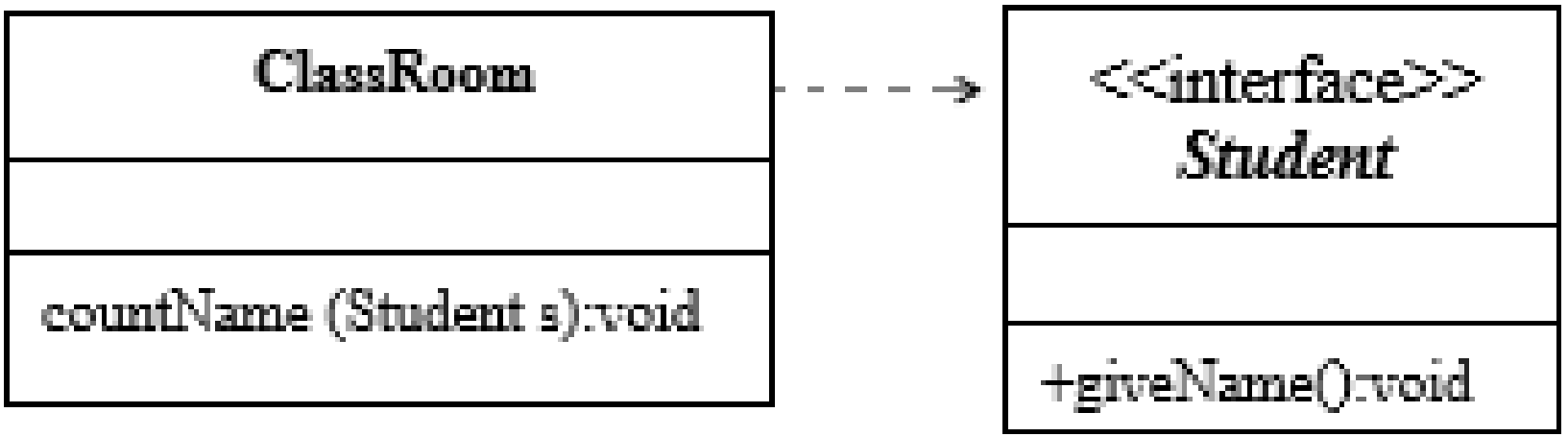
接口和实现关系


<<interface>>
Creator
+createNumber():int
ClassOne
+createNumber():int
ClassTwo
+createNumber():int

关联关系



依赖关系


ClassRoom
countName (Student s):void
<<interface>>
Student
+giveName():void

面向接口的编程（面向抽象的原则）（1）

- 面向接口的编程：是面向对象设计（OOD）的非常重要的基本原则之一。
- 它的含义是：使用接口和同类型的组件通讯，即，对于所有完成相同功能的组件，应该抽象出一个接口，它们都实现该接口。

面向接口的编程（2）

- 具体到JAVA中，可以是接口（interface），或者是抽象类（abstract class），所有完成相同功能的组件都实现该接口，或者从该抽象类继承。
- 客户代码只应该和该接口通讯。

面向接口的编程（3）

- **场景1：** 当我们需要用其它组件完成任务时，只需要替换该接口的实现，代码的其它部分不需要改变。
- **场景2：** 当现有的组件不能满足要求时，我们可以创建新的组件，实现该接口，或者，直接对现有的组件进行扩展，由子类去完成扩展的功能。
- **减少了代码耦合，实现了代码复用**

面向抽象原则（抽象类和接口）

- 当设计一个类时，不让该类面向具体的类，而是面向抽象类或接口，即所设计类中的重要数据是抽象类或接口声明的变量，而不是具体类声明的变量
- **案例：**
  - 已有Circle类，该类创建的对象circle调用getArea()方法可以计算圆的面积。
  - 现在要设计一个Pillar类（柱类），该类的对象调用getVolume()方法可以计算柱体的体积

面向抽象原则（抽象类和接口）

```
public class Pillar {
    Circle  bottom;    //将Circle对象作为成员，bottom是用具体类声明的变量
    double height;
    Pillar (Circle bottom,double height) {
        this.bottom=bottom;this.height=height;
    }
    public double getVolume() {
        return bottom.getArea()*height;
    }
 }
```

面向抽象编程的思想

- 如果不涉及用户需求的变化，上面Pillar类的设计没有什么不妥，但是在某个时候，用户希望Pillar类能创建出底是三角形的柱体。显然上述Pillar类无法创建出这样的柱体，即上述设计的Pillar类不能应对用户的这种需求。

- 因此，我们在设计Pillar类时**不应当让它的底是某个具体类声明的变量**，一旦这样做，Pillar类就依赖该具体类，缺乏弹性，难以应对需求的变化。

面向抽象编程的思想

- 首先编写一个抽象类Geometry（或接口），该抽象类（接口）中定义了一个抽象的getArea()方法

```
public abstract class Geometry { //如果使用接口需用interface来定义Geometry。
    public abstract double getArea();
}
```

```
public class Pillar {
    Geometry  bottom;  //bottom是抽象类Geometry声明的变量
    double height;
    Pillar (Geometry bottom,double height) {
        this.bottom = bottom; this.height=height;
    }
    public double getVolume() {
        return bottom.getArea()*height;
    }
}
```

Pillar类的设计不再依赖具体类，而是面向Geometry类

面向抽象编程的思想

```
public class Circle extends Geometry
{
    double r;
    Circle(double r) {
        this.r=r;
    }
    public double getArea() {
        return(3.14*r*r);
    }
}
```

```
public class Rectangle extends Geometry {
    double a,b;
    Lader(double a,double b) {
        this.a=a; this.b=b;
    }
    public double getArea() {
        return a*b;
    }
}
```

```
public class Application{
    public static void main(String args[]){
        Pillar pillar;
        Geometry bottom;
        bottom=new Rectangle(12,22,100);
        pillar = new Pillar (bottom,58);  //pillar是具有矩形底的柱体
        System.out.println("矩形底的柱体的体积"+pillar.getVolume());
        bottom=new Circle(10);
        pillar = new Pillar (bottom,58); //pillar是具有圆形底的柱体
        System.out.println("圆形底的柱体的体积"+pillar.getVolume());
    }
}
```

Pillar类不再依赖具体类，因此每当系统增加新的Geometry的子类时，比如增加一个Triangle子类，那么不需要修改Pillar类的任何代码，就可以使用Pillar创建出具有三角形底的柱体。

优先使用组合少用继承原则

- 方法复用的两种最常用的技术就是类继承和对象组合
- 通过继承复用方法的缺点是：
  - 继承破坏了封装性，通过继承进行复用也称“**白盒”复用**，其缺点是父类的内部细节对于子类而言是可见的。
  - 子类和父类的关系是**强耦合关系**，也就是说当父类的方法的行为更改时，必然导致子类发生变化
  - 子类从父类继承的方法在编译时刻就确定下来了，所以**无法在运行期间改变从父类继承的方法的行为**。

组合与复用

- 组合对象来复用方法的优点是：
  - 对象组合是类继承之外的另一种复用选择。新的更复杂的功能可以通过组合对象来获得。
  - 对象组合要求对象具有良好定义的接口。这种复用风格被称为**黑箱复用(black-box reuse)**，因为被组合的对象的内部细节是不可见的,对象只以"黑箱"的形式出现。
- **对象与所包含的对象属于弱耦合关系**，因为，如果修改当前对象所包含的对象的类的代码，不必修改当前对象的类的代码。
- 当前对象可以在**运行时刻动态指定所包含的对象**。

```
public abstract class Person {
   public abstract String getMess();
}
```

```
public  class Driver1 extends Person {
   public  String getMess(){
        return "我是中国驾驶员";
   }
}
```

```
public  class Driver5 extends Person {
   public  String getMess(){
        return "我是女驾驶员";
   }
}
```

```
public  class Driver7 extends Person {
   public  String getMess(){
        return "我是男驾驶员";
   }
}
```

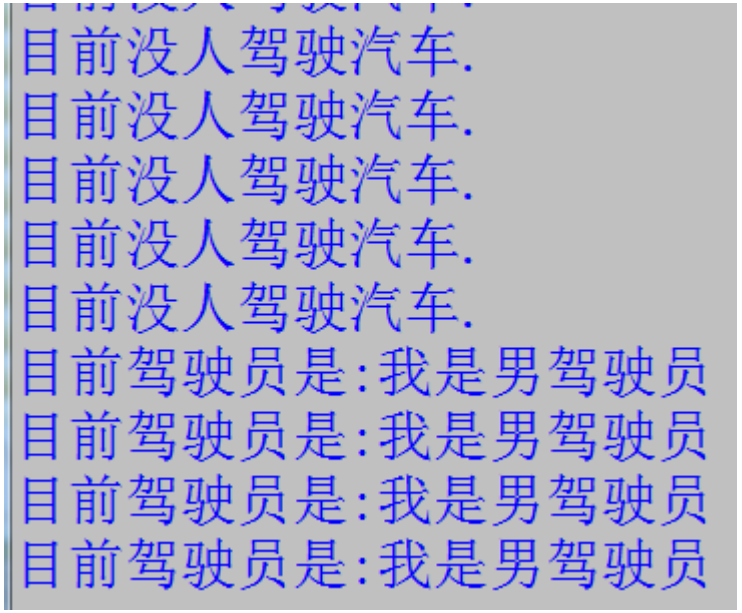
```
public class Car {
    Person person;    //组合驾驶员
    public void setPerson(Person p) {
        person = p;
    }
    public void show() {
        if(person == null) {
            System.out.println("目前没人驾驶汽车.");
        }
        else {
            System.out.println("目前驾驶员是:"+person.getMess());
        }
    }
}
```

```
public class MainClass {
  public static void main(String arg[]) {
    Car car =new Car();
    int i=1;
    while(true) {
      try{
        car.show();
        Thread.sleep(2000);  //每隔2000毫秒更换驾驶员
        Class<?> cs=Class.forName("Driver"+i);
        Person p=(Person)cs.getDeclaredConstructor().newInstance();
        //如果没有第i个驾驶员就触发异常，跳到catch,即无人驾驶或当前驾驶员继续驾驶:
        car.setPerson(p);      //更换驾驶员
        i++;
      }
      catch(Exception exp){
        i++;
      }
      if(i>10) i=1;          //最多10个驾驶员轮换开车
    }
  }
}
```

模拟汽车动态更换驾驶员（维护代码时，不停止软件的运行）

运行主类MainClass之后，不要关闭程序，然后编写Person的子类。名字可以是Driver1,Driver2…。

重新打开一个命令行编译这些子类，比如Driver7.java。然后就会看到程序运行结果的变化（更换驾驶员）



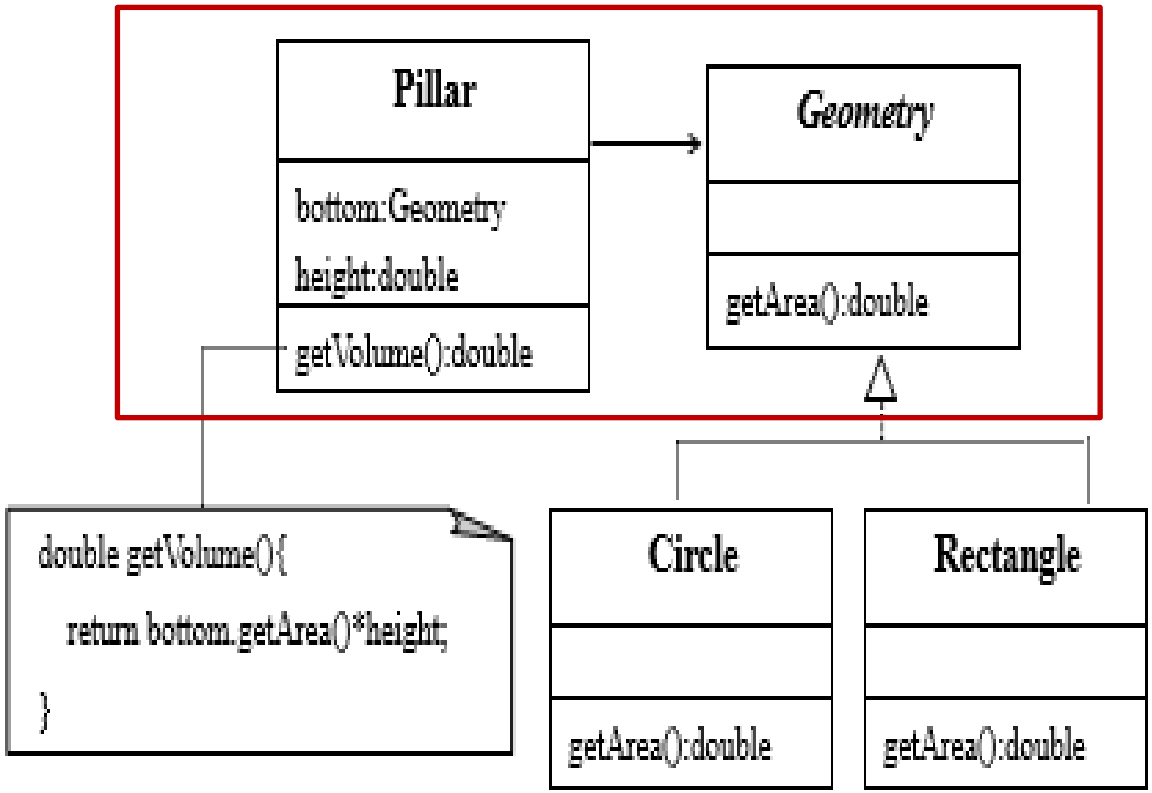
多用组合少用继承原则

- 之所以提倡多用组合，少用继承，是因为在许多设计中，人们希望系统的类之间尽量是低耦合的关系，而不希望是强偶合关系。即在许多情况下需要避开继承的缺点，而需要组合的优点。
- 怎样合理地使用组合，而不是使用继承来获得方法的复用需要经过一定时间的认真思考、学习和编程实践才能悟出其中的道理。

开-闭原则

- 所谓 开-闭原则（Open-Closed Principle）就是让你的设计应当对扩展开放，对修改关闭。
- 怎么理解对扩展开放，对修改关闭呢？实际上这句话的本质是指当一个设计中增加新的模块时，不需要修改现有的模块。
- 在给出一个设计时，应当首先考虑到**用户需求的变化**，将应对用户变化的部分设计为对扩展开放，而设计的核心部分是经过精心考虑之后确定下来的基本结构，这部分应当是对修改关闭的，即不能因为用户的需求变化而再发生变化，因为这部分不是用来应对需求变化的

开-闭原则



该设计中的Geometry和Pillar类就是系统中对修改关闭的部分，而Geometry的子类是对扩展开放的部分。

经常需要面向抽象来考虑系统的总体设计，不要考虑具体类，这样就容易设计出满足“开-闭原则”的系统。

高内聚-低耦合原则

- 耦合主要描述模块之间的关系
- 内聚主要描述模块内部

低耦合

- 低耦合：是指软件系统中，模块与模块之间的直接依赖程度低。
- 模块之间存在依赖,模块独立性越差，耦合越强,导致一个模块的改动可能会影响到其他模块 。比如模块A直接操作了模块B中数据, 则视为强耦合, **若A只是通过数据与模块B交互, 则视为弱耦合**。
- **好处：**独立的模块便于扩展, 维护, 写单元测试, 如果模块之间重重依赖, 会极大降低开发效率，代码可维护性差。

高内聚

- **高内聚**：系统存在AB两个模块儿进行交互，如果修改了A模块儿不影响B模块的工作，那么认为A模块儿有足够的内聚。
- 一个模块应当尽可能独立完成某个功能。模块内部的元素, 关联性越强, 则内聚越高, 模块单一性更强。
- **危害**：低内聚的模块代码, 不管是维护, 扩展还是重构都相当麻烦, 难以下手。

软件设计七大原则

- 单一职责原则: 一个类只负责一个功能领域中的相应职责。
- 开闭原则: 一个软件实体应当对扩展开放，对修改关闭。
- 里氏代换原则: 所有引用基类（父类）的地方必须能透明地使用其子类的对象。
- 合成复用原则：尽量使用组合或者聚合关系实现代码复用，少使用继承。

软件设计七大原则

- 依赖倒转原则: 抽象不应该依赖于细节, 细节应当依赖于抽象。 换言之, 要针对接口编程, 而不是针对实现编程。
- 接口隔离原则: 使用多个专门的接口, 而不使用单一的总接口, 即客户端不应该依赖那些它不需要的接口。
- 迪米特法则: 一个软件实体应当尽可能少地与其他实体发生相互作用。
  - 例如外观模式, 对外暴露统一接口。

小结

设计原则	一句话归纳	目的
开闭原则	对扩展开放，对修改关闭	降低维护带来的新风险
依赖倒置原则	高层不应该依赖低层，要面向接口编程	更利于代码结构的升级扩展
单一职责原则	一个类只干一件事，实现类要单一	便于理解，提高代码的可读性
接口隔离原则	一个接口只干一件事，接口要精简单一	功能解耦，高聚合、低耦合
迪米特法则	不该知道的不要知道，一个类应该保持对其它对象最少的了解，降低耦合度	只和朋友交流，不和陌生人说话，减少代码臃肿
里氏替换原则	不要破坏继承体系，子类重写方法功能发生改变，不应该影响父类方法的含义	防止继承泛滥
合成复用原则	尽量使用组合或者聚合关系实现代码复用，少使用继承	降低代码耦合

小结：

- 在程序设计时，我们应该将程序功能最小化，每个类只干一件事。
- 若在类似功能基础之上添加新功能，则要合理使用继承。
- 对于多方法的调用，要会运用接口，同时合理设置接口功能与数量。
- 最后类与类之间做到低耦合高内聚。

记忆口诀：访问加限制，函数要节俭，依赖不允许，动态加接口，父类要抽象，扩展不更改。

几个重要的设计模式

主要内容（2）

- 几个重要的设计模式
  - 设计模式简介
  - 门面模式
  - 策略模式
  - 访问者模式
  - 装饰模式
  - 适配器模式
  - 责任链模式
  - 工厂方法模式

设计模式简介

- **需求驱动**：设计模式不是为每个人准备的，而是基于业务来选择设计模式，需要时就能想到它。要明白一点，技术永远为业务服务，技术只是满足业务需要的一个工具。
- 我们需要掌握每种设计模式的应用场景、特征、优缺点，以及每种设计模式的关联关系，这样就能够很好地满足日常业务的需要。

什么是设计模式？

- 一个设计模式（pattern）是针对某一类问题的最佳解决方案，而且已经被成功应用于许多系统的设计中，它解决了在某种特定情景中重复发生的某个问题，即一个设计模式是从许多优秀的软件系统中总结出的成功的可复用的设计方案。
- **学习设计模式的必要性**
  - 列举几个设计模式的目的不仅是要掌握、使用这些模式，更重要的是可以是通过讲解这些设计模式体现面向对象的设计思想，这非常有利于更好地使用面向对象语言解决设计中的诸多问题。

注意事项

- 在进行设计时，尽可能用最简单的方式满足系统的要求，而不是费尽心机地琢磨如何在这个问题中使用设计模式。
- 事实上，真实世界中的许多设计实例都没有使用过那些所谓的经典设计模式。
- 一个设计可能并不需要使用设计模式就可以很好地满足系统的要求，如果牵强地使用某个设计模式可能会在系统中增加许多额外的类和对象，影响系统的性能。

设计模式分类

- 创建型：
  - 涉及对象的实例化，这类模式的特点是，不让用户代码依赖于对象的创建或排列方式，避免用户直接使用new运算符创建对象。例如：工厂方法模式
- 行为型
  - 涉及怎样合理地设计对象之间的交互通信，以及怎样合理为对象分配职责，让设计富有弹性，易维护，易复用。例如：策略模式，中介者模式，责任链模式，访问者模式等。
- 结构型
  - 涉及如何组合类和对象以形成更大的结构，和类有关的结构型模式涉及如何合理地使用继承机制，和对象有关的结构型模式涉及如何合理地使用对象组合机制。例如：装饰模式等

创建型模式分为以下几种

- **单例（Singleton）模式**：某个类只能生成一个实例，该类提供了一个全局访问点供外部获取该实例，其拓展是有限多例模式。
- 原型（Prototype）模式：将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例。
- **工厂方法（FactoryMethod）模式**：定义一个用于创建产品的接口，由子类决定生产什么产品。
- **抽象工厂（AbstractFactory）模式**：提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。
- 建造者（Builder）模式：将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。

结构型模式分为以下 7 种

1. 代理（Proxy）模式：为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。
2. **适配器（Adapter）模式**：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。
3. 桥接（Bridge）模式：将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现的，从而降低了抽象和实现这两个可变维度的耦合度。
4. **装饰（Decorator）模式**：动态地给对象增加一些职责，即增加其额外的功能。
5. **外观（Facade）模式**：为多个复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。
6. 享元（Flyweight）模式：运用共享技术来有效地支持大量细粒度对象的复用。
7. 组合（Composite）模式：将对象组合成树状层次结构，使用户对单个对象和组合对象具有一致的访问性。

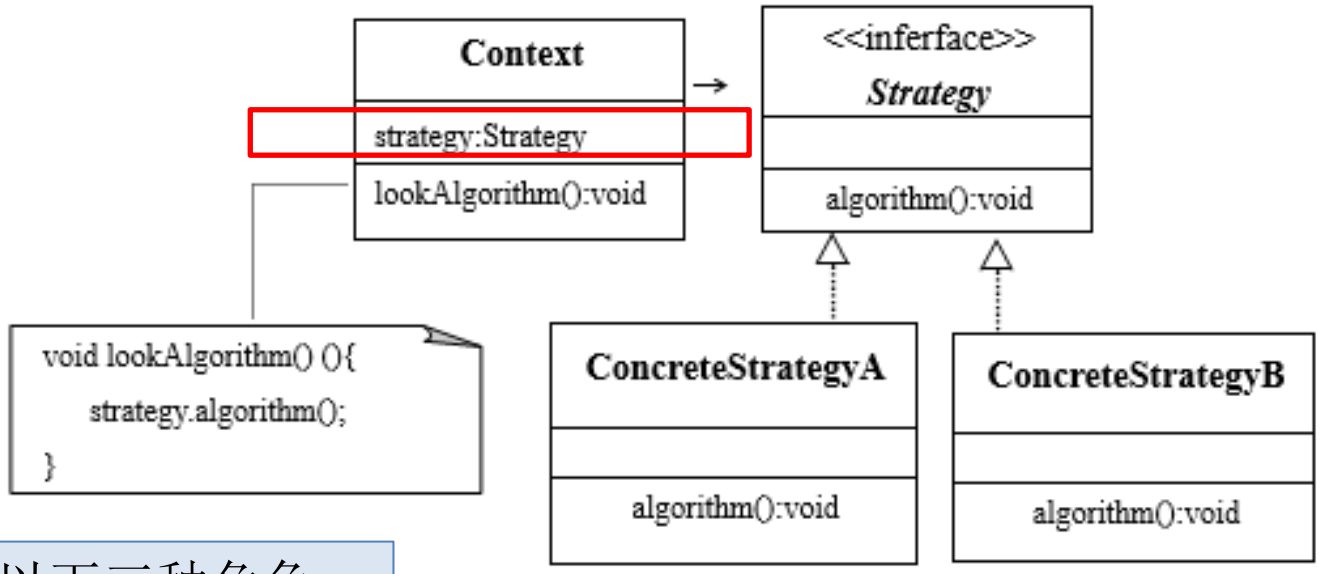
行为型模式是 GoF 设计模式中最为庞大的一类，它包含以下 11 种模式

1. 模板方法（Template Method）模式：定义一个操作中的算法骨架，将算法的一些步骤延迟到子类中，使得子类在可以不改变该算法结构的情况下重定义该算法的某些特定步骤。
2. **策略（Strategy）模式**：定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。
3. 命令（Command）模式：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。
4. **职责链（Chain of Responsibility）模式**：把请求从链中的一个对象传到下一个对象，直到请求被响应为止。通过这种方式去除对象之间的耦合。
5. 状态（State）模式：允许一个对象在其内部状态发生改变时改变其行为能力。
6. **观察者（Observer）模式**：多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。

行为型模式是 GoF 设计模式中最为庞大的一类，它包含以下 11 种模式

7. 中介者（Mediator）模式：定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。
8. 迭代器（Iterator）模式：提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。
9. **访问者（Visitor）模式**：在不改变集合元素的前提下，为一个集合中的每个元素提供多种访问方式，即每个元素有多个访问者对象访问。
10. 备忘录（Memento）模式：在不破坏封装性的前提下，获取并保存一个对象的内部状态，以便以后恢复它。
11. 解释器（Interpreter）模式：提供如何定义语言的文法，以及对语言句子的解释方法，即解释器。

策略模式



结构中包含以下三种角色：

策略（Strategy）：策略是一个接口，该接口定义若干个算法标识，即定义了若干个抽象方法。

上下文（Context）：上下文是依赖于策略接口的类（是面向策略设计的类），即上下文包含有用策略声明的变量。上下文中提供一个方法，该方法委托策略变量调用具体策略所实现的策略接口中的方法。

具体策略（ConcreteStrategy）：具体策略是实现策略接口的类。具体策略实现策略接口所定义的抽象方法，即给出算法标识的具体算法。

策略模式结构

- 策略（Strategy）：核心就是将类中经常需要变化的部分分割出来，并将每种可能的变化对应地交给抽象类的一个子类或实现接口的一个类去负责，从而让类的设计者不去关心具体实现，避免所设计的类依赖于具体的实现。
- 上下文（Context）：上下文（Context）面向策略，即是面向接口**Strategy**的类。
- 具体策略：具体策略是实现Strategy接口的类，即必须重写接口中的方法。

案例

- 问题：在多个裁判负责打分的比赛中，每位裁判给选手一个得分，选手的最后得分是根据全体裁判的得分计算出来的。请给出几种计算选手得分的评分方案（策略），对于某次比赛，可以从你的方案中选择一种方案作为本次比赛的评分方案。
  - 在这里我们把策略接口命名为：**Strategy**。在具体应用中，这个角色的名字可以根据具体问题来命名。
  - 在本问题中将上下文命名为AverageScore，即让AverageScore类依赖于Strategy接口。
  - 每个具体策略负责一系列算法中的一个。

1. 策略（Strategy）Strategy.java

```
public interface Strategy {
    public double computerAverage(double [] a);
}
```

2. 上下文（Context）AverageScore.java

```
public class  AverageScore{
    Strategy strategy;
    public void setStrategy(Strategy strategy){
        this.strategy=strategy;
    }
    public double getAverage (double [] a){
       if(strategy!=null)
         return strategy.computerAverage(a);
       else {
         System.out.println("没有求平均值的算法,得到的-1不代表平均值");
         return -1;
       }
    }
}
```

3. 具体策略StrategyA.java

```
public class StrategyA implements Strategy{
    public double computerAverage(double [] a){
        double score=0,sum=0;
        for(int i=0;i<a.length;i++){
            sum=sum+a[i];
        }
        score=sum/a.length;
        return score;
    }
}
```

4. 具体策略StrategyB.java

```
import java.util.Arrays;
public class StrategyB implements Strategy{
    public double computerAverage(double [] a){
        if(a.length<=2)
            return 0;
        double score=0,sum=0;
        Arrays.sort(a);   //排序数组
        for(int i=1;i<a.length-1;i++){
          sum=sum+a[i];
        }
        score=sum/(a.length-2);
        return score;
    }
}
```

5. 模式的使用

```
class Person{
   String name;
   double score;
   public void setScore(double t){
      score=t;
   }
   public void setName(String s){
      name=s;
   }
   public double getScore(){
      return score;
   }
   public String getName(){
      return name;
   }
}
```

5. 模式的使用

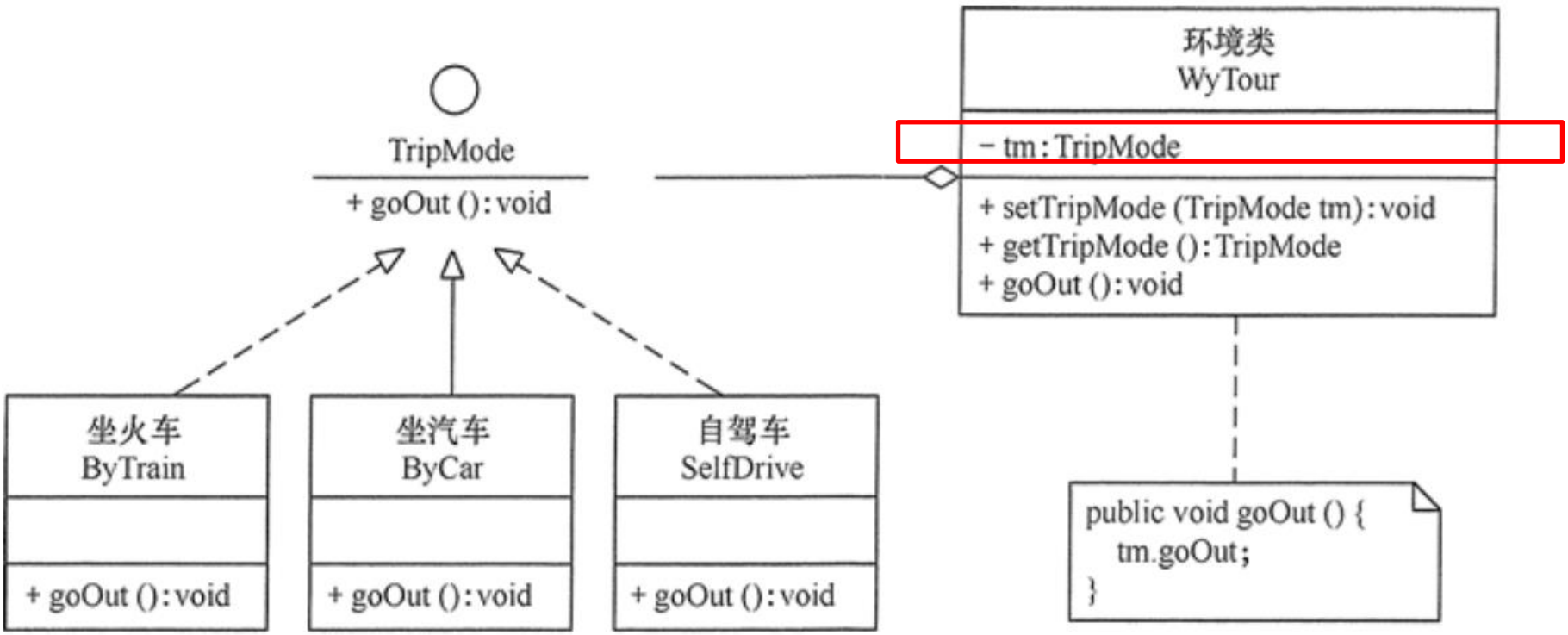
```
public class Application{
  public static void main(String args[]){
    AverageScore game=new AverageScore();//上下文对象game
    game.setStrategy(new StrategyA());   //上下文对象使用具体策略
    Person zhang=new Person();
    zhang.setName("张三");
    double [] a={9.12,9.25,8.87,9.99,6.99,7.88};
    double aver = game.getAverage(a) ;    //上下文对象得到平均值
    zhang.setScore(aver);
    System.out.println("算法A:");
    System.out.printf("%s最后得分:%5.3f%n",zhang.getName(),zhang.getScore());
    game.setStrategy(new StrategyB());
    aver = game.getAverage(a) ;          //上下文对象得到平均值
    zhang.setScore(aver);
    System.out.println("算法B:");
    System.out.printf("%s最后得分:%5.3f%n",zhang.getName(),zhang.getScore());
  }
}
```

5. 模式的使用

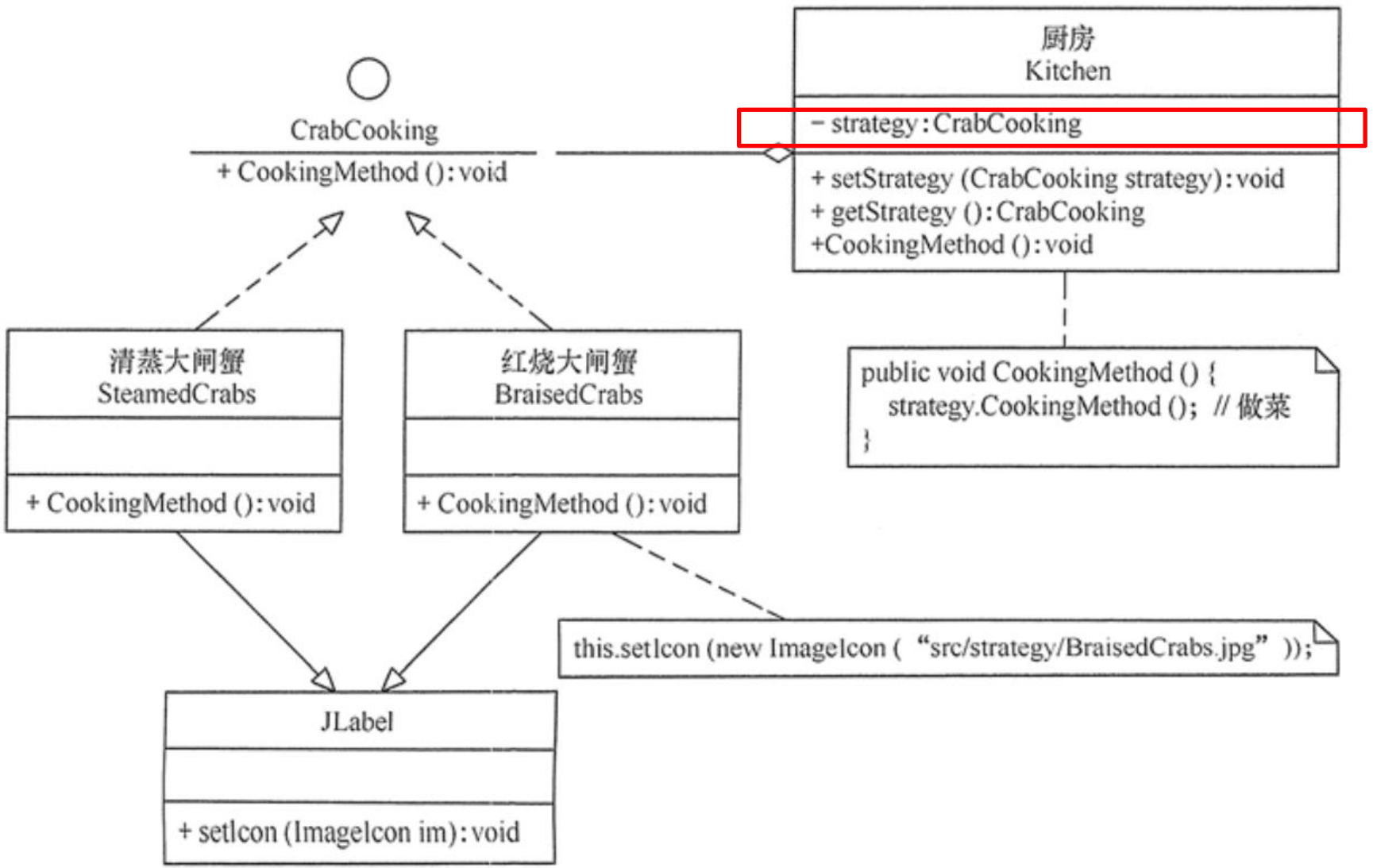
- 已经使用策略模式给出了可以使用的类，可以将这些类看作是一个小框架，用户就可以使用这个小框架中的类编写应用程序了。

```
算法A:
张三最后得分:8.683
算法B:
张三最后得分:8.780
```

案例


TripMode
+ goOut ():void
环境类
WyTour
- tm:TripMode
+ setTripMode (TripMode tm):void
+ getTripMode ():TripMode
+ goOut ():void
坐火车
ByTrain
+ goOut ():void
坐汽车
ByCar
+ goOut ():void
自驾车
SelfDrive
+ goOut ():void
public void goOut () {
tm.goOut;
}

案例


厨房
Kitchen
- strategy:CrabCooking
+ setStrategy (CrabCooking strategy):void
+ getStrategy ():CrabCooking
+CookingMethod ():void
CrabCooking
+ CookingMethod ():void
清蒸大闸蟹
SteamedCrabs
+ CookingMethod ():void
红烧大闸蟹
BraisedCrabs
+ CookingMethod ():void
public void CookingMethod () {
strategy.CookingMethod (); // 做菜
}
this.setIcon (new ImageIcon ("src/strategy/BraisedCrabs.jpg"));
JLabel
+ setIcon (ImageIcon im):void

模式的优点

- 上下文（Context）和具体策略（ConcreteStrategy）是松耦合关系。因此上下文只知道它要使用某一个实现Strategy接口类的实例，但不需要知道具体是哪一个类。

- 策略模式满足“开-闭原则”。当增加新的具体策略时，不需要修改上下文类的代码，上下文就可以引用新的具体策略的实例。

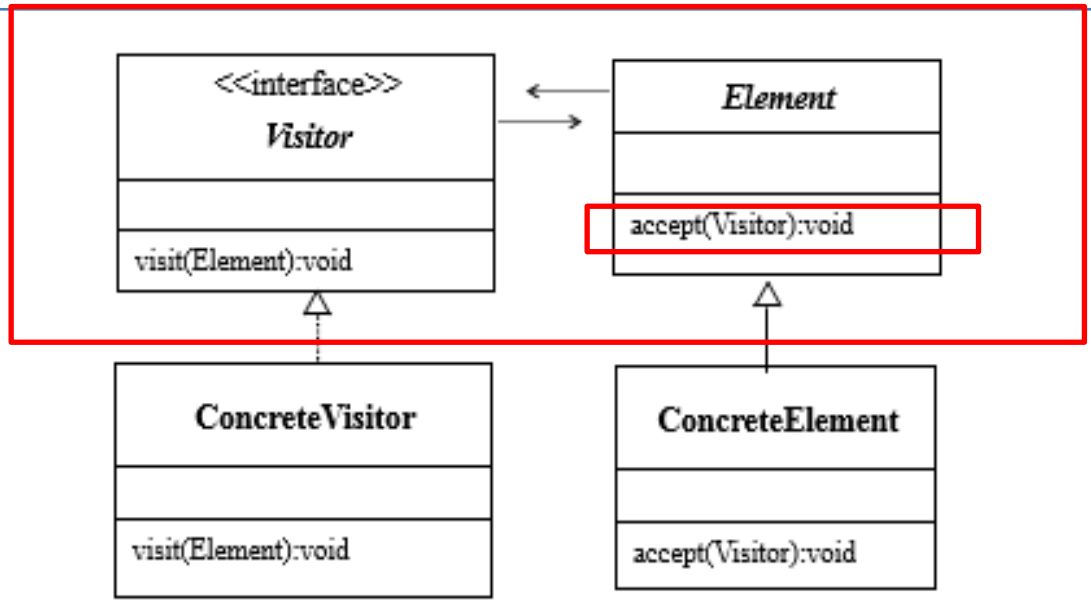
适合的场景

- 程序的主要类（相当于上下文角色）不希望暴露复杂的、与算法相关的数据结构，那么可以使用策略模式封装算法，即将算法分别封装到具体策略中

相对继承机制的优势

- 考虑到系统扩展性和复用性，就应当注意面向对象的一个基本原则之一：少用继承，多用组合。
- 策略模式的应用层次采用的是组合结构，即将上下文类的某个方法的内容的不同变体分别封装在不同的具体策略中。
- 如果将父类的某个方法的内容的不同变体交给对应的子类去实现，就使得这些实现和父类中的其它代码是紧耦合关系，因为父类的任何改动都会影响到子类。

访问者模式



结构中包含以下4种角色：

- **抽象元素**（Element）：一个抽象类，该类定义了接收访问者的accept操作。
- **具体元素**（Concrete Element）：Element的子类。
- **抽象访问者**（Visitor）：一个接口，该接口定义操作具体元素的方法。
- **具体访问者**（Concrete Visitor）：实现Visitor接口的类。

访问者模式模式结构

- 抽象访问者（Visitor）:某个类可能用自己的实例方法操作自己的数据，但在某些设计中，可能需要定义作用于类中的数据的新操作，而且这个新的操作不应当由该类的中的某个实例方法来承担。
  - 比如，电表有自己的显示用电量的方法（用显示盘显示），但需要定义一个方法来计算电费，即需要定义作用于电量的新操作，显然这个新的操作不应当由电表来承担。
  - 在实际生活中，应当由物业部门的“计表员”观察电表的用电量，然后按着有关收费标准计算出电费。**让一个称作访问者的对象访问电表**，并根据用电量来计算电费。
- 抽象元素（Element）:访问者需要访问元素，元素必须提供允许访问者访问它的方法。
- 具体访问者（Concrete Visitor）

案例

- 问题：根据电表显示的用电量计算用户的电费。用户包括居民和企业。访问同一个电表，即分别按家用电标准和工业用电标准计算了电费。
  - 抽象的访问者：
  - 具体的访问者：居民和企业用户
  - 抽象元素：抽象类AmmeterElement
  - 具体元素：Ammeter(模拟电表)

1. 抽象访问者（Visitor）

```
public interface Visitor{
  public double visit(AmmeterElement elment);
}
```

2. 抽象元素（Element）

```
public abstract class AmmeterElement{
    public abstract void accept(Visitor v);
    public abstract double showElectricAmount();
    public abstract void setElectricAmount(double n);
}
```

3. 具体访问者（Concrete Visitor）

```
public class HomeAmmeterVisitor implements Visitor{
    public double visit(AmmeterElement ammeter){
      double charge=0;
      double unitOne=0.6,unitTwo=1.05;
      int basic = 6000;
      double n=ammeter.showElectricAmount();
      if(n<=basic) {
          charge = n*unitOne;
      }
      else {
         charge =basic*unitOne+(n-basic)*unitTwo;
      }
      return charge;
    }
}
```

3. 具体访问者（Concrete Visitor）

```
public class IndustryAmmeteVisitor implements Visitor{
   public double visit(AmmeterElement ammeter){
      double charge=0;
      double unitOne=1.52,unitTwo=2.78;
      int basic = 15000;
      double n=ammeter.showElectricAmount();
      if(n<=basic) {
          charge = n*unitOne;
      }
      else {
         charge =basic*unitOne+(n-basic)*unitTwo;
      }
      return charge;
   }
}
```

4. 具体元素（Concrete Element）

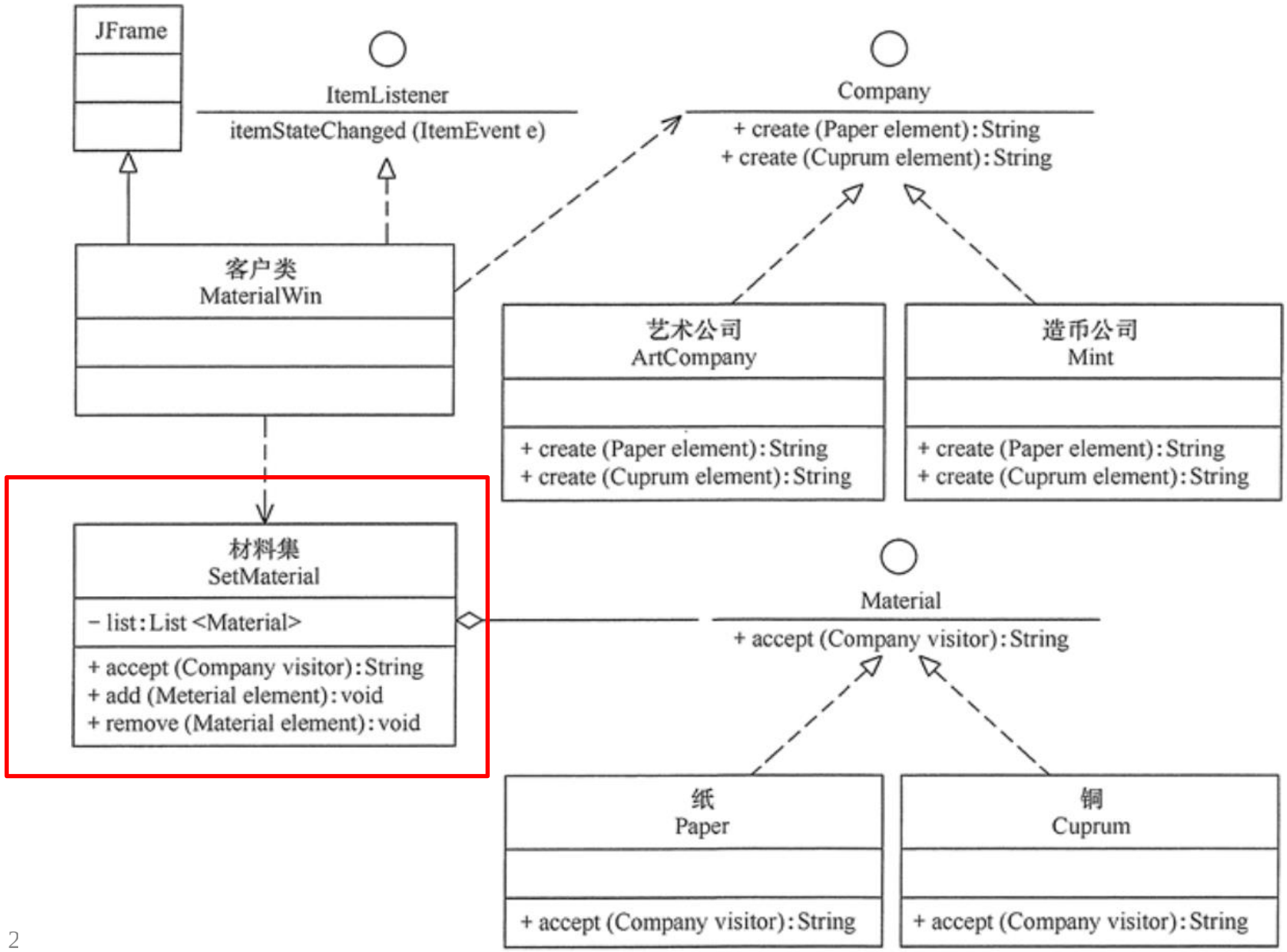
```
public class Ammeter extends AmmeterElement{
   double electricAmount;    //电表的电量
   public void setElectricAmount(double n) {
      electricAmount = n;
   }
   public void accept(Visitor visitor){
      double feiyong=visitor.visit(this); //让访问者访问当前元素
      System.out.println("当前电表的用户需要交纳电费:"+feiyong+"元");
   }
   public double showElectricAmount(){
      return electricAmount;
   }
}
```

模式的使用：

```
public class Application{
   public static void main(String args[]) {
     Visitor 计表员=new HomeAmmeterVisitor(); //按家用电标准计算电费的"计表员"
     Ammeter 电表=new Ammeter();
     电表.setElectricAmount(5678);
     电表.accept(计表员);
     计表员=new IndustryAmmeteVisitor(); //按工业用电标准计算电费的"计表员"
     电表.setElectricAmount(5678);
     电表.accept(计表员);
   }
}
```

```
当前电表的用户需要交纳电费:3406.7999999999997元
当前电表的用户需要交纳电费:8630.56元
```

案例：艺术公司和造币公司结构图



模式的使用：

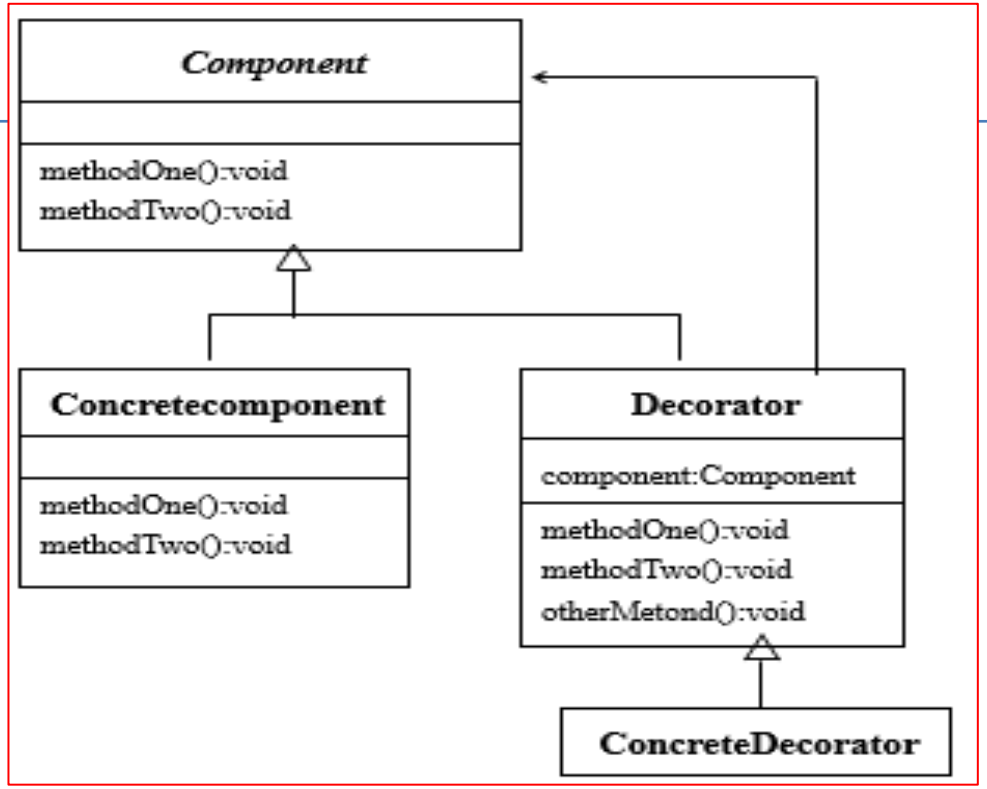

利用访问者模式设计艺术公司和造币公司
原材料有：铜和纸，请选择生产公司：
艺术公司
造币公司


利用访问者模式设计艺术公司和造币公司
中国人民银行
100
壹佰圆
原材料有：铜和纸，请选择生产公司：
艺术公司
造币公司

模式优点和使用场景

- 模式优点：在不改变一个集合中的元素的类的情况下，可以增加新的施加于该元素上的新操作。保持一定的扩展性。
- 使用场景：需要对集合中的对象进行很多不同的并且不相关的操作，而我们又不想修改对象的类，就可以使用访问者模式。访问者模式可以在Visitor类中集中定义一些关于集合中对象的操作。

装饰模式



结构中包含以下4种角色：

- **抽象组件**（Component）：抽象组件（是抽象类）定义了需要进行装饰的方法。抽象组件就是“被装饰者”角色。
- **具体组件**（ConcreteComponent）：具体组件是抽象组件的一个子类。
- **装饰**（Decorator）：该角色是抽象组件的一个子类，是“装饰者”角色，其作用是装饰具体组件。Decorator角色需要包含**抽象组件**的引用。
- **具体装饰**（ConcreteDecotator）：具体装饰是Decorator角色的一个非抽象子类

模式结构

- 1.抽象组件：装饰模式是动态地扩展一个对象的功能，而不需要改变原始类代码的一种成熟模式。在许多设计中，可能需要改进类的某个对象的功能，而不是该类创建的全部对象
- 2.具体组件：具体组件是抽象组件的一个子类，具体组件的实例称作“被装饰者”。
- 3.装饰（Decorator）：装饰（Decorator）角色是抽象组件的一个子类，**需要包含被装饰者（抽象组件）的引用**。装饰（Decorator）角色也是抽象组件的子类，**但需要额外提供一些方法，用来装饰抽象组件**。
- 4.具体装饰：具体装饰负责用新的方法去装饰“被装饰者”的方法。

案例

- 给麻雀安装智能电子翅膀
  - 抽象组件的名字是Bird
  - 装饰（Decorator）角色是抽象组件的一个子类，需要包含被装饰者（抽象组件）的引用。
  - 具体组件角色：Sparrow类，该类的实例模拟麻雀
  - 具体装饰是SparrowDecorator类，该类使用eleFly()方法去装饰fly()方法

1. 抽象组件

Bird.java

```
public abstract class Bird{
    public abstract int fly();
}
```

2．具体组件

```
public class Sparrow extends Bird{
    public final int DISTANCE=100;
    public int fly(){
        return DISTANCE;
    }
}
```

3. 装饰（Decorator）

```
public abstract class Decorator extends Bird{
   Bird bird;      //被装饰者
   public Decorator(){
   }
   public Decorator(Bird bird){
      this.bird=bird;
   }
   public abstract int eleFly();//用于装饰fly()的方法,行为由具体装饰者去实现
}
```

4. 具体装饰

```
public class SparrowDecorator extends Decorator{
  public final int DISTANCE=50;  //eleFly方法(模拟电子翅膀)能飞50米
  SparrowDecorator(Bird bird){
    super(bird);
  }
  public int fly(){   //被装饰的方法
    int distance=0;
    distance=bird.fly()+eleFly();//让装饰者bird首先调用fly()，然后再调用eleFly()
    return distance;
  }
  public int eleFly(){      //具体装饰者重写装饰者中用于装饰的方法
    return DISTANCE;
  }
}
```

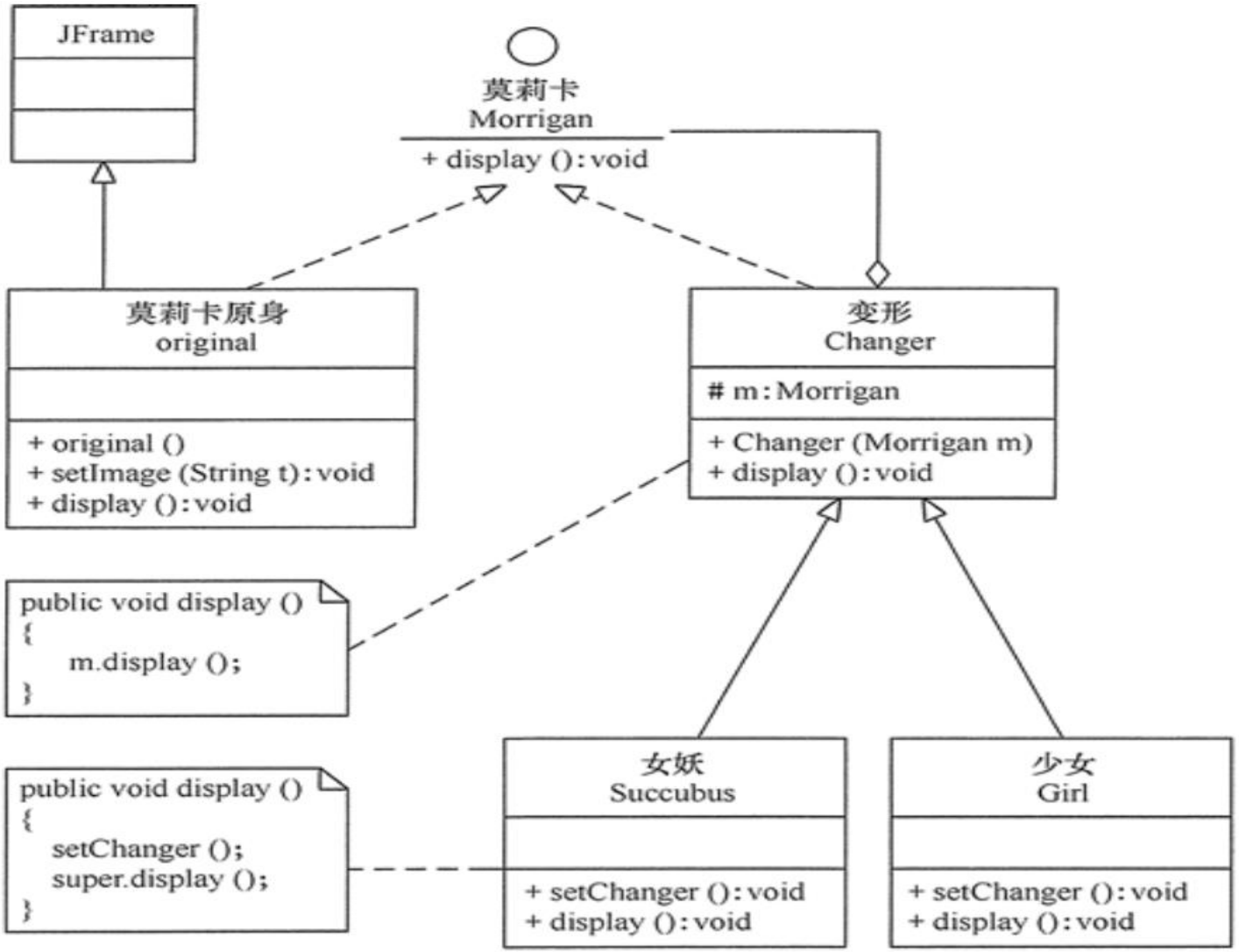
模式的使用：

```
public class Application{
   public static void main(String args[]){
      Bird bird=new Sparrow();
      System.out.println("没有安装电子翅膀的小鸟飞行距离:"+bird.fly());
      bird=new SparrowDecorator(bird); //bird通过"装饰"安装了1个电子翅膀
      System.out.println("安装1个电子翅膀的小鸟飞行距离:"+bird.fly());
      bird=new SparrowDecorator(bird); //bird通过"装饰"安装了2个电子翅膀
      System.out.println("安装2个电子翅膀的小鸟飞行距离:"+bird.fly());
      bird=new SparrowDecorator(bird); //bird通过"装饰"安装了3个电子翅膀
      System.out.println("安装3个电子翅膀的小鸟飞行距离:"+bird.fly());
   }
}
```

演示了一只没有安装电子翅膀的小鸟只能飞行100米，对该鸟进行“装饰”，即给它安装一个电子翅膀，那么安装了1个电子翅膀后的鸟就能飞行150米，然后在继续给它安装电子翅膀，那么安装了2个电子翅膀后的鸟就能飞行200米

没有安装电子翅膀的小鸟飞行距离:100
安装1个电子翅膀的小鸟飞行距离:150
安装2个电子翅膀的小鸟飞行距离:200
安装3个电子翅膀的小鸟飞行距离:250

案例：游戏角色“莫莉卡·安斯兰”的结构图



模式的使用：

相对继承机制的优势

- 通过继承也可改进对象的行为，对于某些简单的问题这样做未尝不可，但是如果考虑到系统扩展性，就应当注意面向对象的一个基本原则之一：少用继承，多用组合。就功能来说装饰模式相比生成子类更为灵活。
- 如果继续采用继承机制来维护上面的系统，就必须修改系统，不断增加新的Bird的子类，这简直是维护的一场灾难。

模式的优点

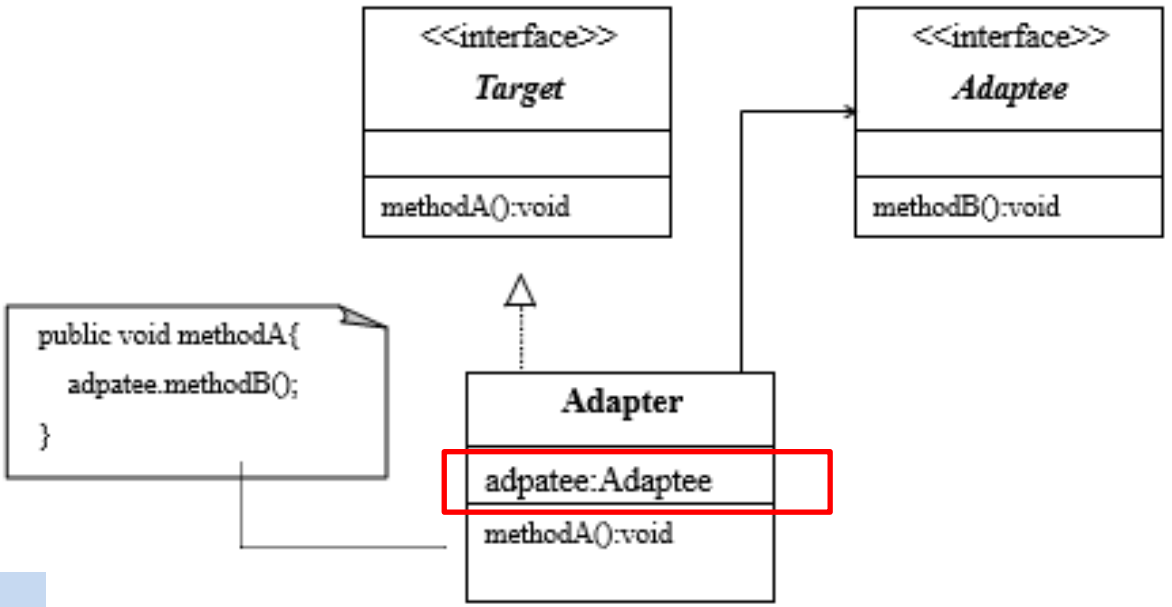
- 被装饰者和装饰者是松耦合关系。由于装饰（Decorator）仅仅依赖于抽象组件（Component），因此具体装饰只知道它要装饰的对象是抽象组件的某一个子类的实例，但不需要知道是哪一个具体子类装饰模式，相比生成子类更为灵活。

适合的情景

- 程序希望动态地增强类的某个对象的功能，而又不影响到该类的其他对象。

- 采用继承来增强对象功能不利于系统的扩展和维护。

适配器模式



结构中包括三种角色：

- **目标（Target）：**目标是一个接口，该接口是客户想使用的接口。
- **被适配者（Adaptee）：**被适配者是一个已经存在的接口或抽象类，这个接口或抽象类需要适配。
- **适配器（Adapter）：**适配器是一个类，该类实现了目标接口并包含有被适配者的引用，即适配器的职责是对被适配者接口（抽象类）与目标接口进行适配。

模式的结构

- 目标（Target）：目标是一个接口，该接口是客户想使用的接口。
- 被适配者：被适配者是一个已经存在的接口或抽象类，这个接口或抽象类需要适配。
- 适配器：适配器是一个类，该类实现了目标接口并包含有被适配者的引用，即适配器的职责是对被适配者接口与目标接口进行适配。

案例

- 用户家里现有一台洗衣机，使用交流电，现在用户新买了一台录音机，录音机只能使用直流电。
- 由于供电系统供给用户家里是交流电，因此用户需要用适配器将交流电转化为直流电供录音机使用
  - 目标（Target）：是名字为DirectCurrent的接口
  - 被适配者（Adaptee）：是名字为AlternateCurrent的接口
  - 适配器：是名字为ElectricAdapter类，该类实现了DirectCurrent接口并包含有AlternateCurrent接口变量。
  - 被适配者（Adaptee）的具体类：PowerCompany

1. 目标（Target）

```
public interface DirectCurrent{
    public String giveDirectCurrent();
}
```

2. 被适配者

```
public interface AlternateCurrent{
    public String giveAlternateCurrent();
}
```

3. 适配器

```
public class ElectricAdapter implements DirectCurrent{
   AlternateCurrent out;
   ElectricAdapter(AlternateCurrent out){
      this.out=out;
   }
   public String giveDirectCurrent(){
     String  m = out.giveAlternateCurrent();  //先由out得到交流电
     StringBuffer str =new StringBuffer(m);
     //以下将交流电转为直流电:
     for(int i=0;i<str.length();i++) {
        if(str.charAt(i)=='0') {
           str.setCharAt(i,'1');
        }
     }
     m =new String(str);
     return m;  //返回直流电
   }
}
```

PowerCompany.java

```
class PowerCompany implements AlternateCurrent { //交流电提供者
    public String giveAlternateCurrent(){
        return "10101010101010101010"; //用这样的串表示交流电
    }
}
```

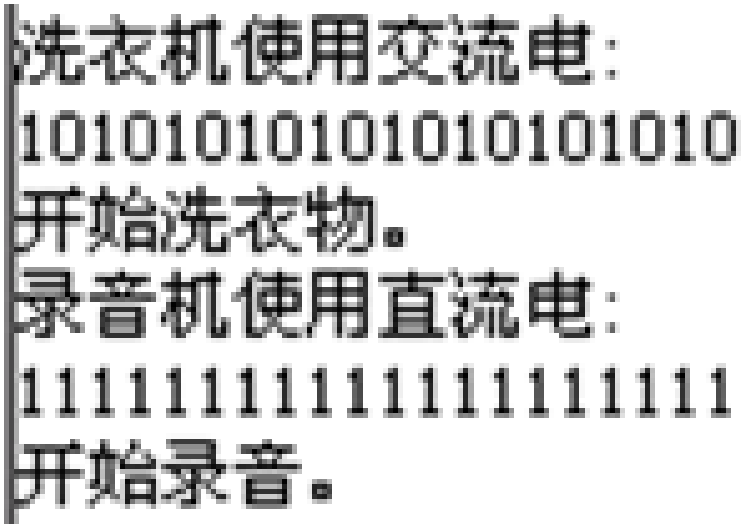
```
class Recorder { //录音机使用直流电
   String name;
   Recorder(){
      name="录音机";
   }
   Recorder(String s){
      name=s;
   }
   public void turnOn(DirectCurrent a){
      String s=a.giveDirectCurrent();
      System.out.println(name+"使用直流电:\n"+s);
      System.out.println("开始录音。");
   }
}
```

```
class Wash { //洗衣机使用交流电
   String name;
   Wash(){
     name="洗衣机";
   }
   Wash(String s){
     name=s;
   }
   public void turnOn(AlternateCurrent a){
      String s=a.giveAlternateCurrent();
      System.out.println(name+"使用交流电:\n"+s);
      System.out.println("开始洗衣物。");
   }
}
```

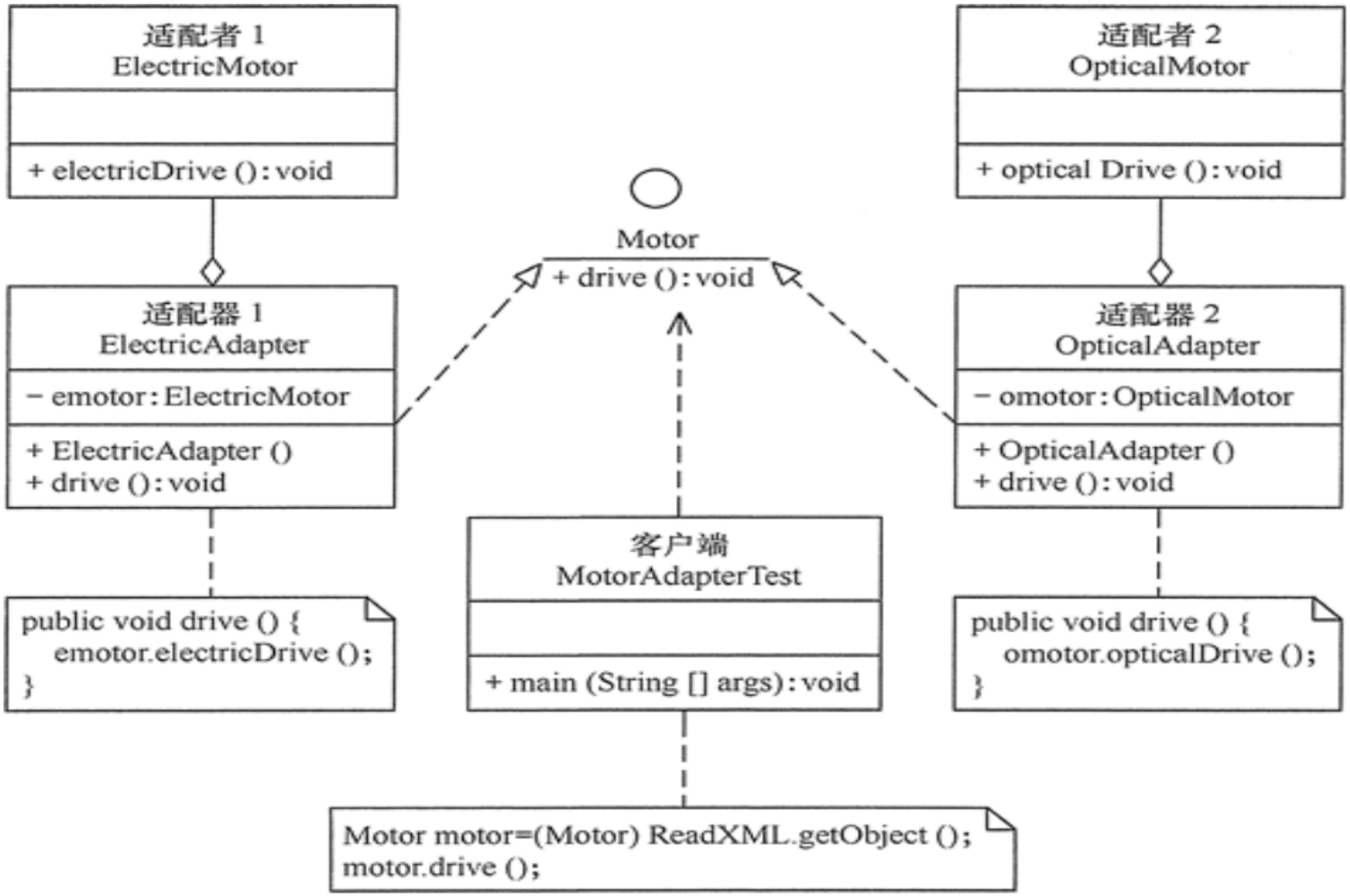
模式的使用：

```
public class Application{
  public static void main(String args[]){
    AlternateCurrent aElectric =new PowerCompany(); //交流电
    Wash wash =new Wash();
    wash.turnOn(aElectric); //洗衣机使用交流电
    //对交流电aElectric进行适配得到直流电dElectric:
    DirectCurrent dElectric = new ElectricAdapter(aElectric); //将交流电适配成直流电
    Recorder recorder =new Recorder();
    recorder.turnOn(dElectric); //录音机使用直流电
  }
}
```

模式的使用：



案例：发动机适配器的结构图



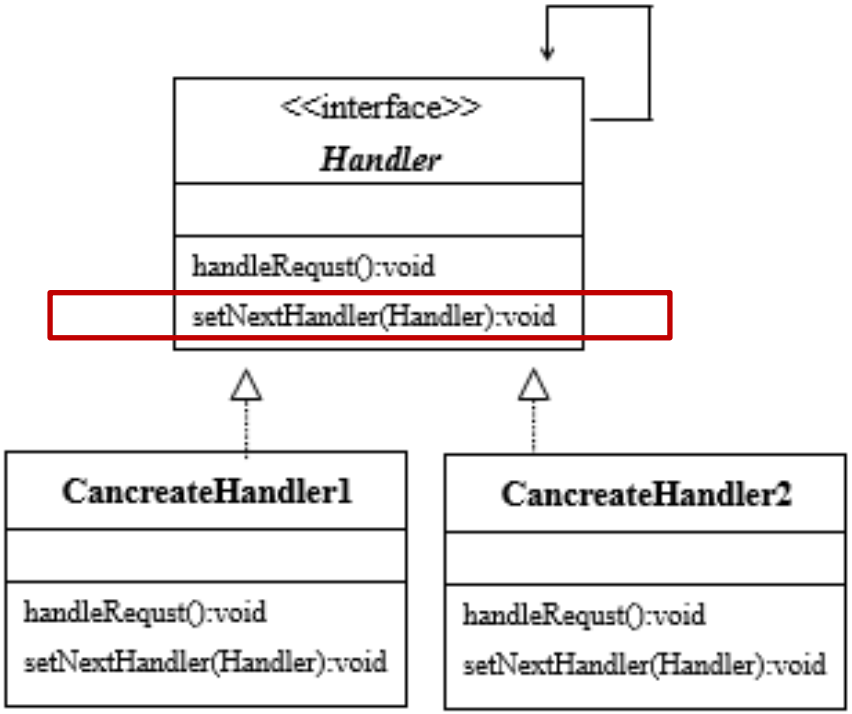
适配器的适配程度

- 1. 完全适配
  - 如果目标（Target）接口中的方法数目与被适配者（Adaptee）接口的方法数目相等，那么适配器（Adapter）可将被适配者接口（抽象类）与目标接口进行完全适配。
- 2. 不完全适配
  - 如果目标（Target）接口中的方法数目少于被适配者（Adaptee）接口的方法数目，那么适配器（Adapter）只能将被适配者接口（抽象类）与目标接口进行部分适配。
- 3. 剩余适配
  - 如果目标（Target）接口中的方法数目大于被适配者（Adaptee）接口的方法数目，那么适配器（Adapter）可将被适配者接口（抽象类）与目标接口进行完全适配，但必须将目标多余的方法给出用户允许的默认实现。

单接口适配器

- 如果一个接口中有多个方法，如果一个类实现该接口，该类就必须要实现接口中的全部方法。
- 针对一个接口的“单接口适配器”就是已经实现了该接口的类，并对接口中的每个方法都给出了一个默认的实现。
- 在Java API中，如果一个接口中的方法多于一个，Java API就针对该接口提供相应的单接口适配器。
  - 比如，java.awt.event包中的MouseAdapter就是MouseListener接口的单接口适配器，MouseAdapter将MouseListener接口中的5个方法全部实现为不进行任何操作。

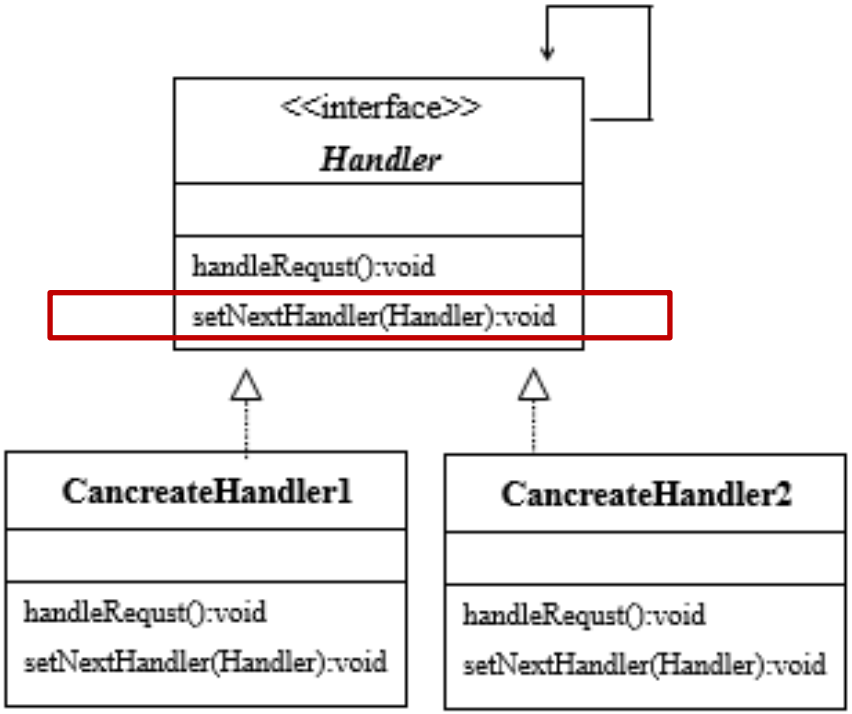
责任链模式



结构中包含以下2种角色：

- **处理者（Handler）：**处理者是一个接口，负责规定具体处理者处理用户的请求的方法以及具体处理者设置后继对象的方法。
- **具体处理者（ConcreteHandler）：**具体处理者是实现处理者接口的类的实例。具体处理者通过调用处理者接口规定的方法处理用户的请求，即在接到用户的请求后，处理者将调用接口规定的方法，在执行该方法的过程中，如果发现能处理用户的请求，就处理有关数据，否则就反馈无法处理的信息给用户，然后将用户的请求传递给自己的后继对象。

责任链模式



结构中包含以下2种角色：

1、处理者

2、具体处理者

责任链模式是使用多个对象处理用户请求的成熟模式，责任链模式的关键是将用户的请求分派给许多对象。

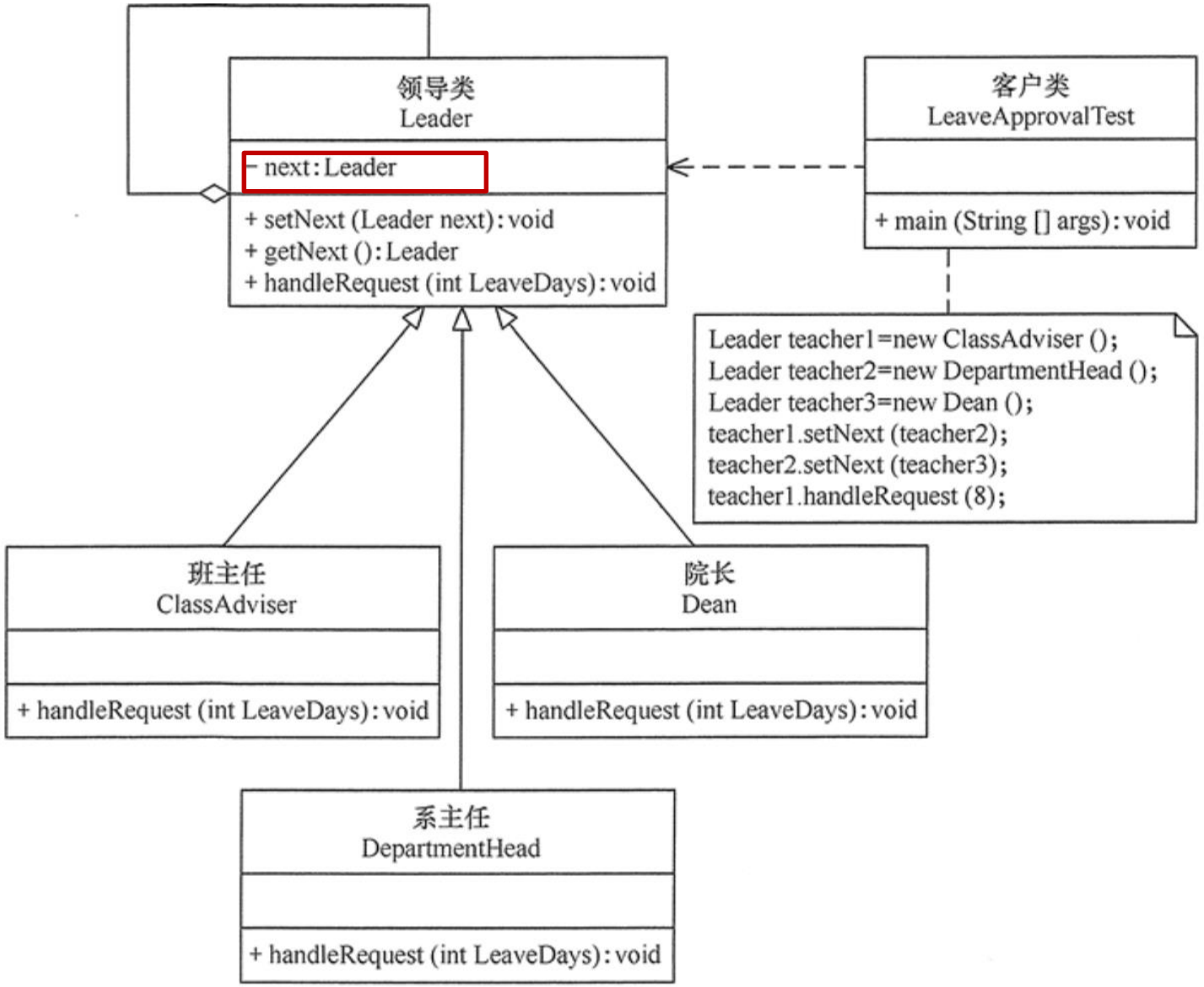
模式的结构

- **处理者（Handler）：** 处理者是一个接口，负责规定具体处理者处理用户的请求的方法以及具体处理者设置后继对象的方法。
- **具体处理者（ConcreteHandler）：** 具体处理者是实现处理者接口的类的实例。具体处理者通过调用处理者接口规定的方法处理用户的请求，即在接到用户的请求后，处理者将调用接口规定的方法，在执行该方法的过程中，如果发现能处理用户的请求，就处理有关数据，否则就反馈无法处理的信息给用户，然后将用户的请求传递给自己的后继对象。

案例：用责任链模式设计一个请假条审批模块

- 假如规定学生请假小于或等于 2 天，班主任可以批准；小于或等于 7 天，系主任可以批准；小于或等于 10 天，院长可以批准；其他情况不予批准；这个实例适合使用职责链模式实现。

案例：请假条审批模块的结构图



1、Leader.java

```
//抽象处理者：领导类
abstract class Leader {
    private Leader next;

    public void setNext(Leader next) {
        this.next = next;
    }

    public Leader getNext() {
        return next;
    }

    //处理请求的方法
    public abstract void handleRequest(int LeaveDays);
}
```

2、ClassAdviser.java

```
//具体处理者1：班主任类
class ClassAdviser extends Leader {
    public void handleRequest(int LeaveDays) {
        if (LeaveDays <= 2) {
            System.out.println("班主任批准您请假" + LeaveDays + "天。");
        } else {
            if (getNext() != null) {
                getNext().handleRequest(LeaveDays);
            } else {
                System.out.println("请假天数太多，没有人批准该假条！");
            }
        }
    }
}
```

3、DepartmentHead.java

```
//具体处理者2：系主任类
class DepartmentHead extends Leader {
    public void handleRequest(int LeaveDays) {
        if (LeaveDays <= 7) {
            System.out.println("系主任批准您请假" + LeaveDays + "天。");
        } else {
            if (getNext() != null) {
                getNext().handleRequest(LeaveDays);
            } else {
                System.out.println("请假天数太多，没有人批准该假条！");
            }
        }
    }
}
```

4、Dean.java

```
//具体处理者3：院长类
class Dean extends Leader {
    public void handleRequest(int LeaveDays) {
        if (LeaveDays <= 10) {
            System.out.println("院长批准您请假" + LeaveDays + "天。");
        } else {
            if (getNext() != null) {
                getNext().handleRequest(LeaveDays);
            } else {
                System.out.println("请假天数太多，没有人批准该假条！");
            }
        }
    }
}
```

测试类

```
public class LeaveApprovalTest {
    public static void main(String[] args) {
        //组装责任链
        Leader teacher1 = new ClassAdviser();
        Leader teacher2 = new DepartmentHead();
        Leader teacher3 = new Dean();
        //Leader teacher4=new DeanOfStudies();
        teacher1.setNext(teacher2);
        teacher2.setNext(teacher3);
        //teacher3.setNext(teacher4);
        //提交请求
        teacher1.handleRequest(8);
    }
}
```

程序运行结果如下：

```
院长批准您请假8天。
```

假如增加一个教务处长类，可以批准学生请假 20 天，也非常简单

```
//具体处理者4：教务处长类
class DeanOfStudies extends Leader {
    public void handleRequest(int LeaveDays) {
        if (LeaveDays <= 20) {
            System.out.println("教务处长批准您请假" + LeaveDays + "天。");
        } else {
            if (getNext() != null) {
                getNext().handleRequest(LeaveDays);
            } else {
                System.out.println("请假天数太多，没有人批准该假条！");
            }
        }
    }
}
```

责任链模式

- **模式的优点：**
  - 当在处理者中分配职责时,责任链给应用程序更多的灵活性。
- 使用场景
  - 有许多对象可以处理用户的请求。
  - 程序希望动态制定可处理用户请求的对象集合

思考题：电影院售票员与现金找赎

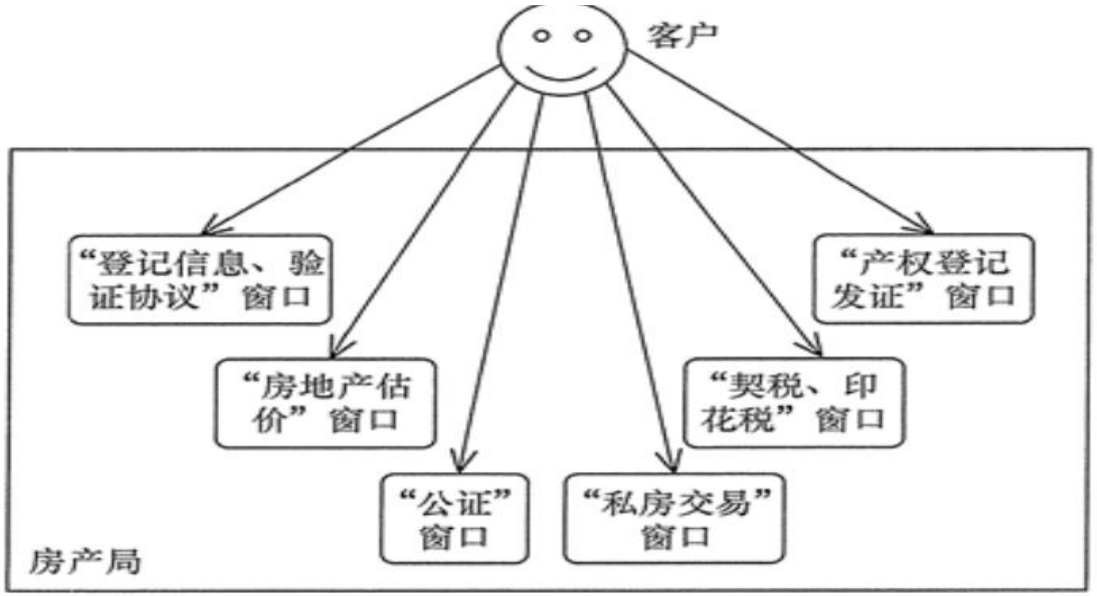
- 假如电影票7元，购票者购买2张电影票，给售票员100元面值钞票一张。那么实际上售票员找赎86元过程如下：
  - 首先在50元面值的钱盒中看能否完成任务，或部分任务。50元面值的钱盒发现能找赎86元中的50元（贡献一张50元的钞票），即只完成部分任务，因此把剩余的36元任务交给下一个20元面值的钱盒，20元面值的钱盒发现能完成36元中的20元（贡献一张20面值的钞。票），因此把剩余的16元任务交给下一个10元面值的钱盒。依次类推，如果在后续某个钱盒完成了自己的找赎任务，那么售票员就完成了找赎任务，如果一直到最后一个钱盒（假设是1元面值的钱盒），该钱盒也无法完成自己的找赎任务，那么售票员就无法完成找赎。使用责任链模拟售票员找赎，那么责任链上的对象就是钱盒，即责任链模式中的处理者（Handler）。

案例：电影院售票员与现金找赎

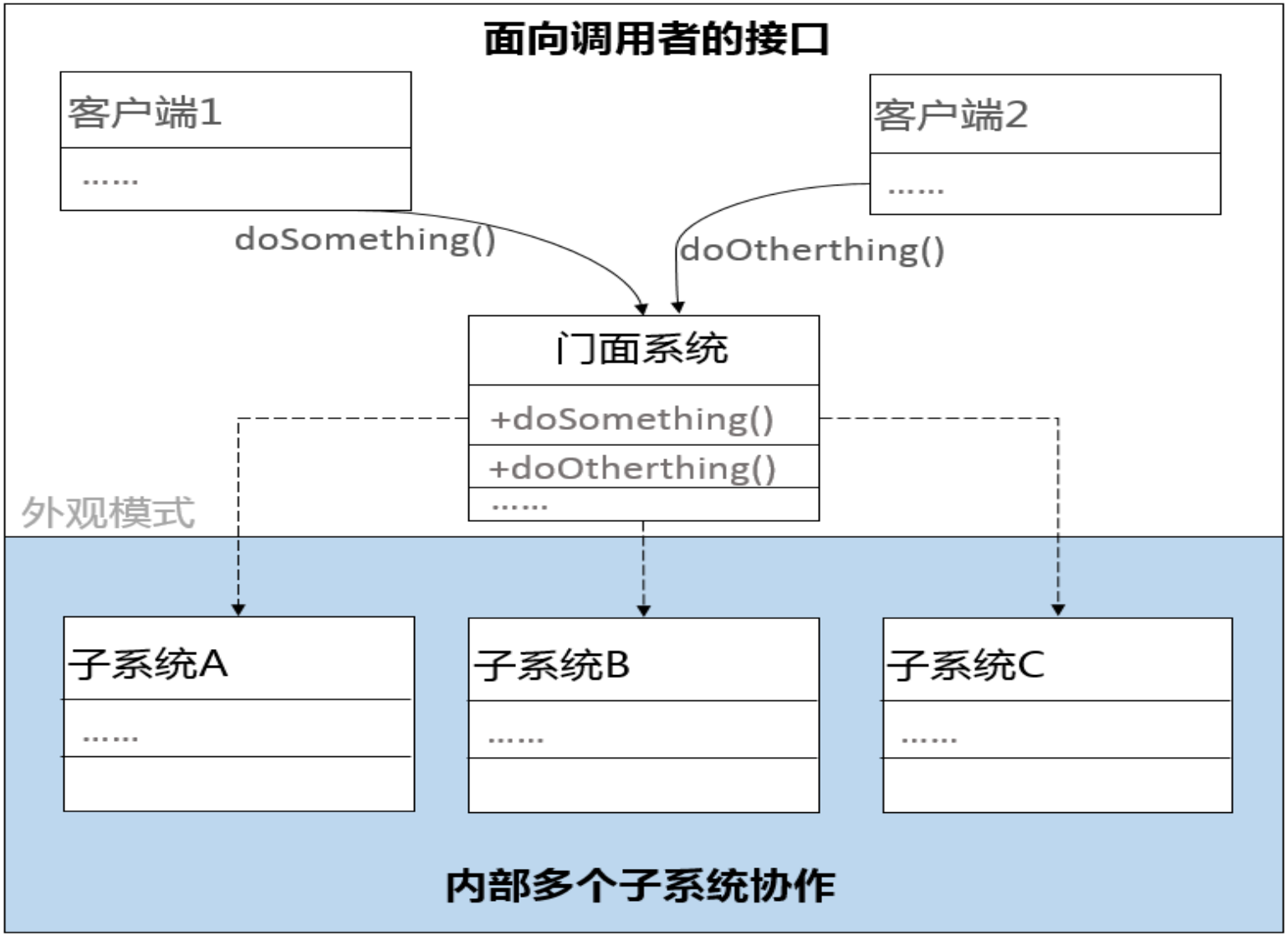
- 问题中,责任链上的处理者接口的名字是MoneyHandler,负责规定具体处理者使用哪些方法来处理用户的请求以及规定具体处理者设置后继对象的方法。
- 具体处理者就是实现处理着接口 MoneyHandler的类,即模拟钱盒的类MoneyBox 。
- TickerSeller将使用框架中的 MoneyBox创建责任链,并使用该责任链进行找赎。
- Application主类模拟售票员卖票并找赎

外观模式（Facade模式，也叫门面模式）

- 在现实生活中，常常存在办事较复杂的例子，如办房产证或注册一家公司，有时要同多个部门联系，这时要是有一个综合部门能解决一切手续问题就好了。
- 客户去当地房产局办理房产证过户要遇到的相关部门



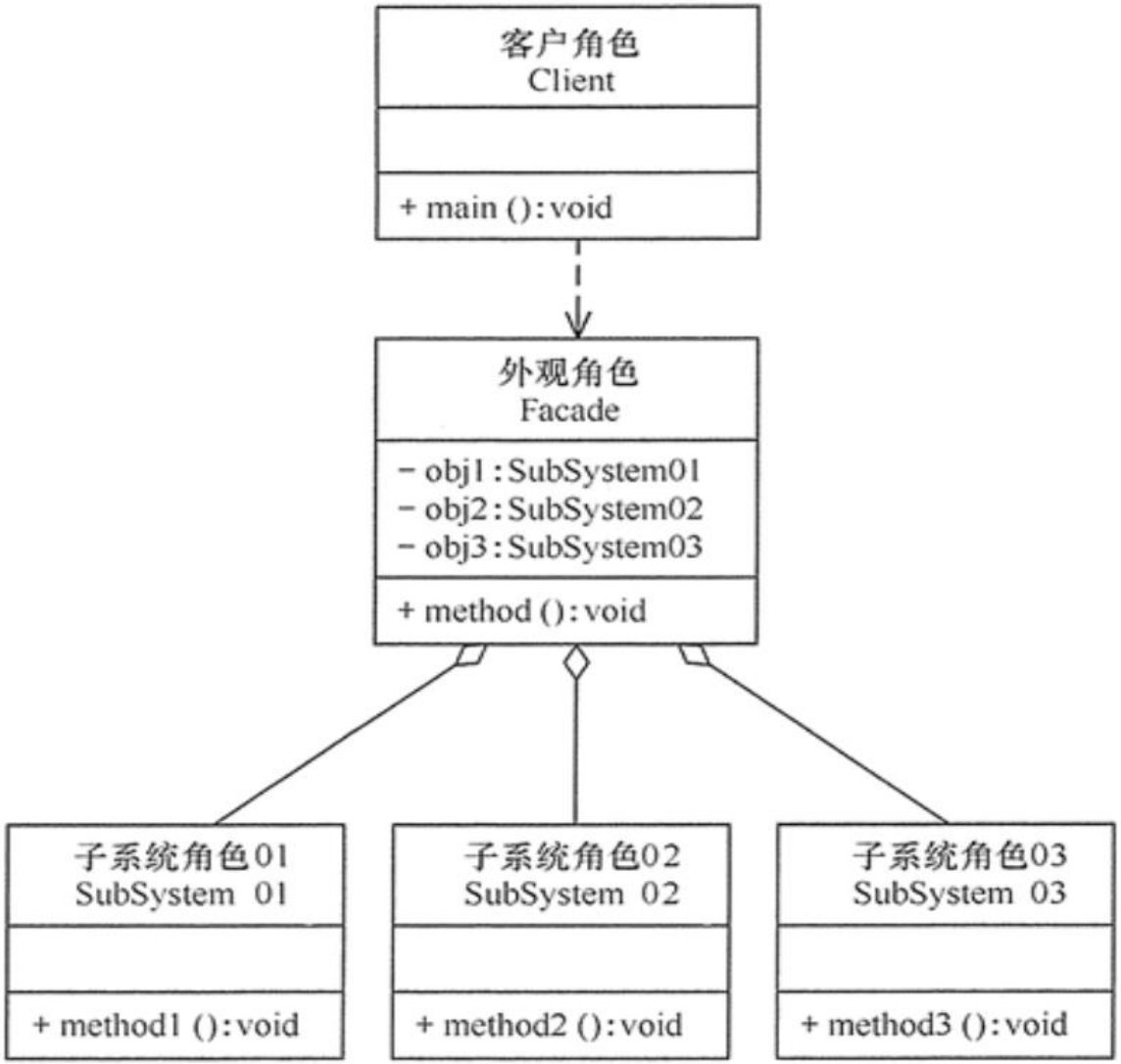
外观（Facade）模式的结构图



外观模式的定义与特点

- 外观（Facade）模式又叫作门面模式，是一种通过为多个复杂的子系统提供一个一致的接口，而使这些子系统更加容易被访问的模式。
- 该模式对外有一个统一接口，外部应用程序不用关心内部子系统的具体细节，这样会大大降低应用程序的复杂度，降低其与子系统的耦合，提高了程序的可维护性。
- 外观（Facade）模式是“迪米特法则”的典型应用

案例：外观（Facade）模式的结构图



示例代码：

```
//子系统角色
class SubSystem01 {
    public void method1() {
        System.out.println("子系统01的method1()被调用！");
    }
}

//子系统角色
class SubSystem02 {
    public void method2() {
        System.out.println("子系统02的method2()被调用！");
    }
}

//子系统角色
class SubSystem03 {
    public void method3() {
        System.out.println("子系统03的method3()被调用！");
    }
}
```

```java
public class FacadePattern {
    public static void main(String[] args) {
        Facade f = new Facade();
        f.method();
    }
}

//外观角色
class Facade {
    private SubSystem01 obj1 = new SubSystem01();
    private SubSystem02 obj2 = new SubSystem02();
    private SubSystem03 obj3 = new SubSystem03();

    public void method() {
        obj1.method1();
        obj2.method2();
        obj3.method3();
    }
}
```

外观（Facade）模式的优点

外观（Facade）模式是“迪米特法则”的典型应用，它有以下主要优点。

1. 降低了子系统与客户端之间的耦合度，使得子系统的变化不会影响调用它的客户类。
2. 对客户屏蔽了子系统组件，减少了客户处理的对象数目，并使得子系统使用起来更加容易。
3. 降低了大型软件系统中的编译依赖性，简化了系统在不同平台之间的移植过程，因为编译一个子系统不会影响其他的子系统，也不会影响外观对象

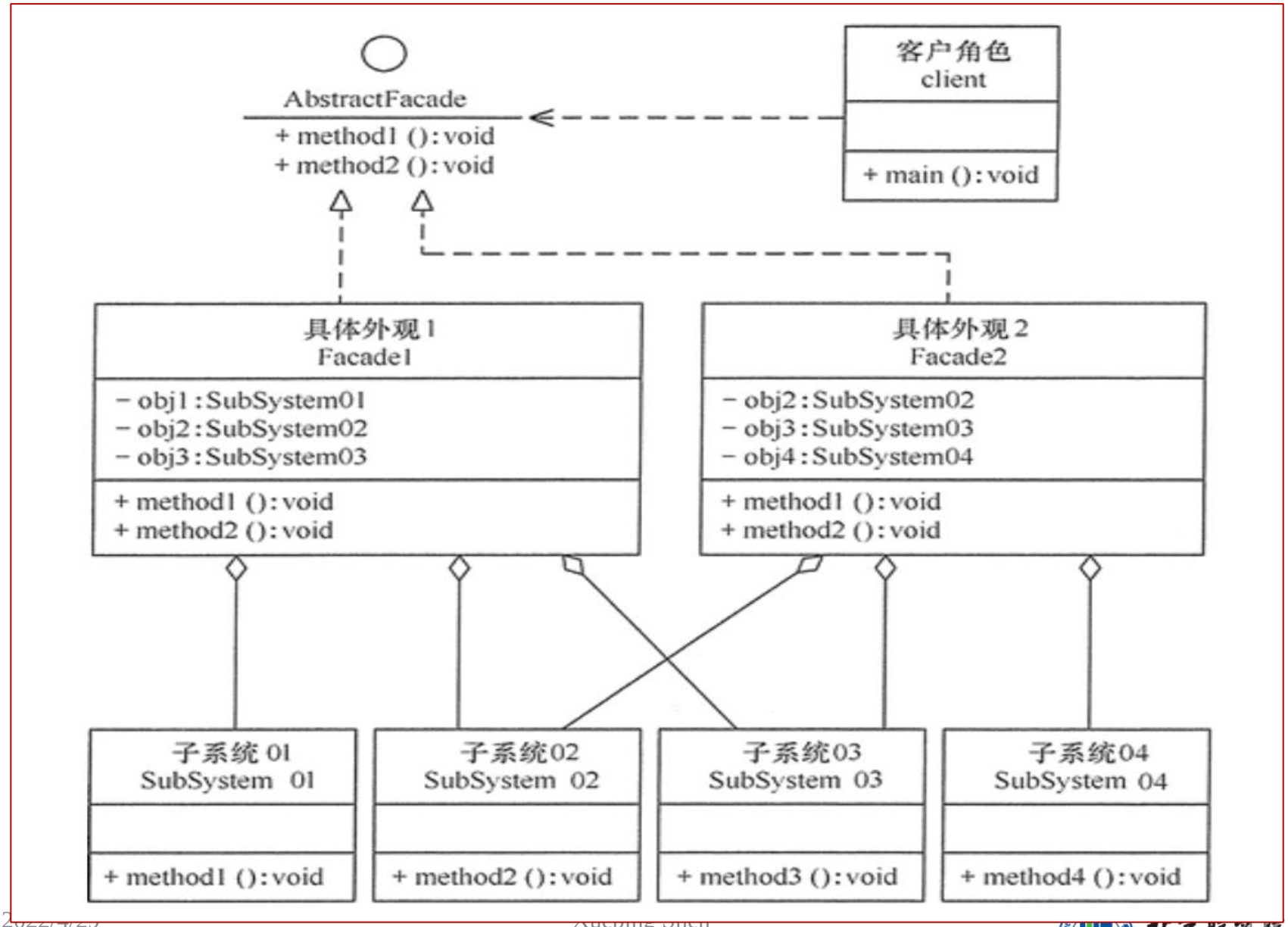
外观（Facade）模式的主要缺点

外观（Facade）模式的主要缺点如下：

1、不能很好地限制客户使用子系统类，很容易带来未知风险。

2、增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”。

外观模式的扩展


AbstractFacade
+ method1 ():void
+ method2 ():void
客户角色
client
+ main ():void
具体外观1
Facade1
- obj1:SubSystem01
- obj2:SubSystem02
- obj3:SubSystem03
+ method1 ():void
+ method2 ():void
具体外观2
Facade2
- obj2:SubSystem02
- obj3:SubSystem03
- obj4:SubSystem04
+ method1 ():void
+ method2 ():void
子系统01
SubSystem 01
+ method1 ():void
子系统02
SubSystem 02
+ method2 ():void
子系统03
SubSystem 03
+ method3 ():void
子系统04
SubSystem 04
+ method4 ():void

外观模式的扩展

- 在外观模式中，当增加或移除子系统时需要修改外观类，这违背了“开闭原则”。

- 如果引入抽象外观类，则在一定程度上解决了该问题。

工厂模式

- 现实生活中，原始社会自给自足（没有工厂），农耕社会小作坊（简单工厂，民间酒坊），工业革命流水线（工厂方法，自产自销），现代产业链代工厂（抽象工厂，富士康）。
- 我们的项目代码同样是由简到繁一步一步迭代而来的，但对于调用者来说，却越来越简单。

工厂模式

- 工厂模式的定义：定义一个创建产品对象的工厂接口，将产品对象的实际创建工作推迟到具体子工厂类当中。这满足**创建型模式**中所要求的**“创建与使用相分离”**的特点。
  - 简单工厂模式（Simple Factory Pattern）
    - 简单工厂模式又叫作静态工厂方法模式（Static Factory Method Pattern）
  - 工厂方法模式
  - 抽象工厂模式

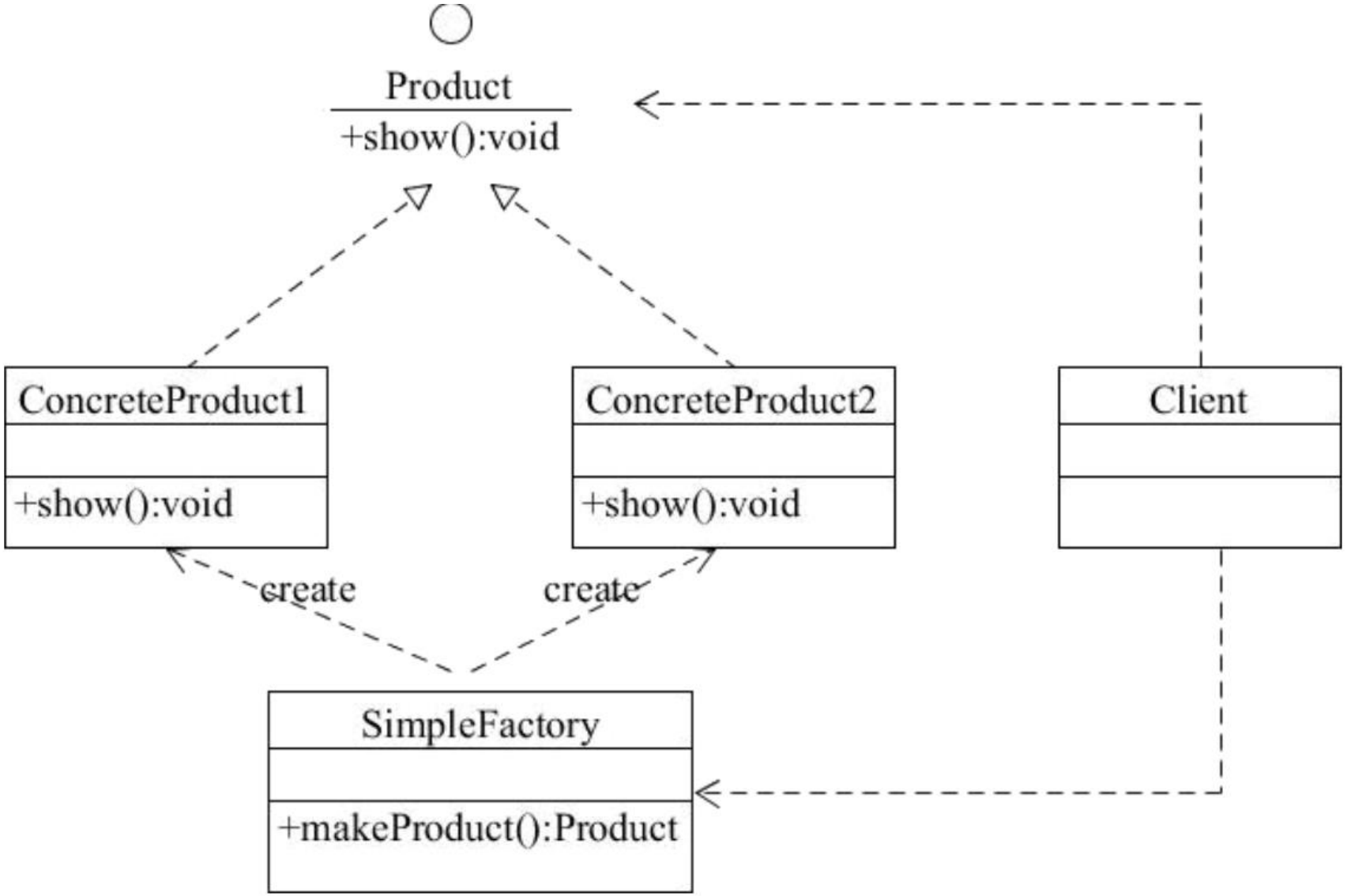
简单工厂模式（Simple Factory Pattern）

- 简单工厂模式又叫作静态工厂方法模式（Static Factory Method Pattern）
- 简单来说，简单工厂模式有一个具体的工厂类，可以生成多个不同的产品，属于创建型设计模式。
- 简单工厂模式不在 GoF 23 种设计模式之列。

模式的结构与实现

- 简单工厂模式的主要角色如下：
  - 简单工厂（SimpleFactory）：是简单工厂模式的核心，负责实现创建所有实例的内部逻辑。工厂类的创建产品类的方法可以被外界直接调用，创建所需的产品对象。
  - 抽象产品（Product）：是简单工厂创建的所有对象的父类，负责描述所有实例共有的公共接口。
  - 具体产品（ConcreteProduct）：是简单工厂模式的创建目标。

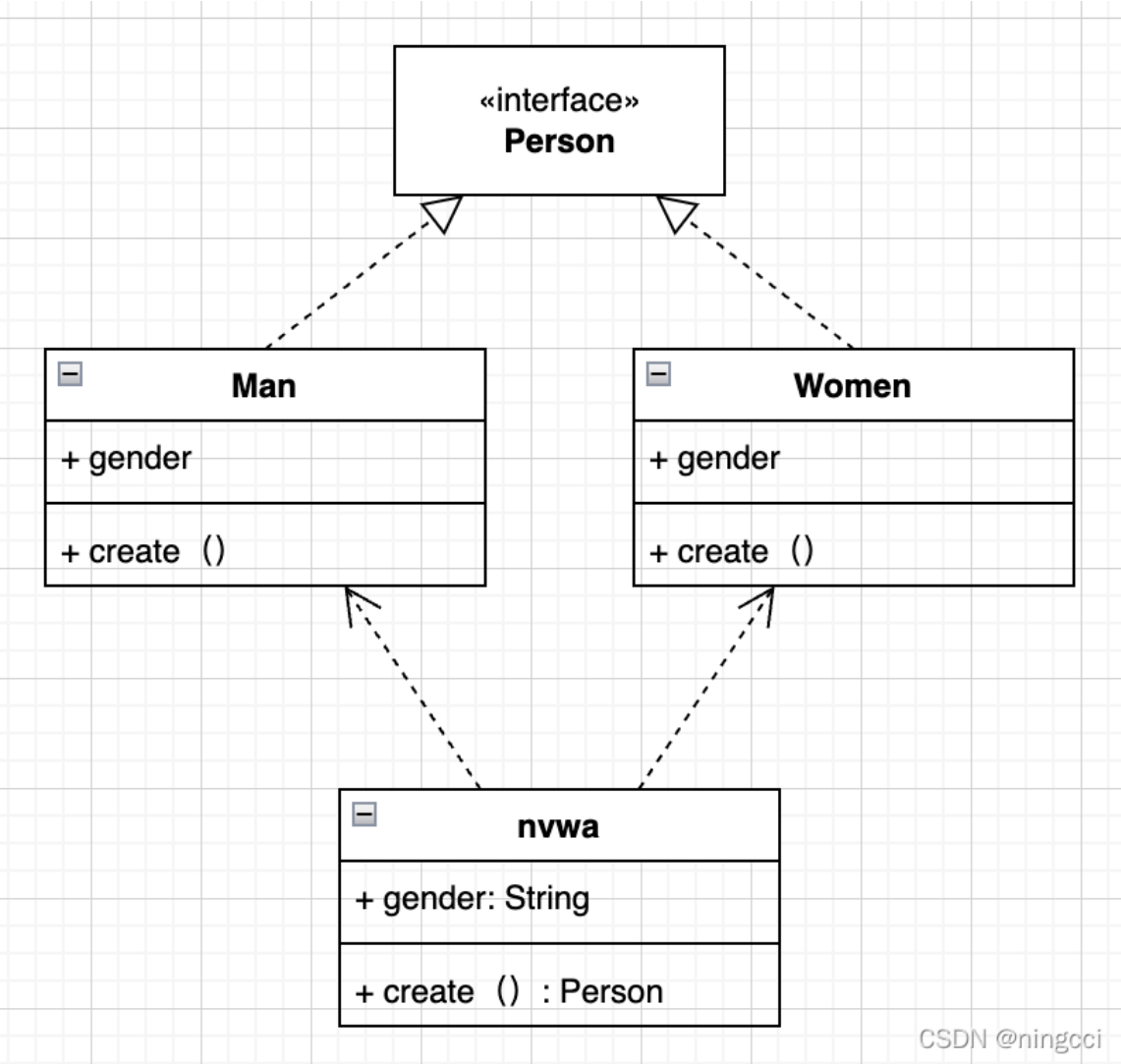
简单工厂模式的结构图



案例：

- 使用简单工厂模式模拟女娲（Nvwa）造人（Person），如果传入参数M，则返回一个Man对象，如果传入参数W，则返回一个Woman对象，请实现该场景。
- 现需要增加一个新的Robot类，如果传入参数R，则返回一个Robot对象，对代码进行修改并注意女娲的变化。

未增加Robot之前的类图



```java
package simple_factory_pattern;

public interface Person {
    public void create();
}
```

```java
package simple_factory_pattern;

public class Man implements Person{
    @Override
    public void create() {
        System.out.println("成功创造了男人...");
    }
}
```

```java
package simple_factory_pattern;

public class Women implements Person{
    @Override
    public void create() {
        System.out.println("成功创建了女人...");
    }
}
```

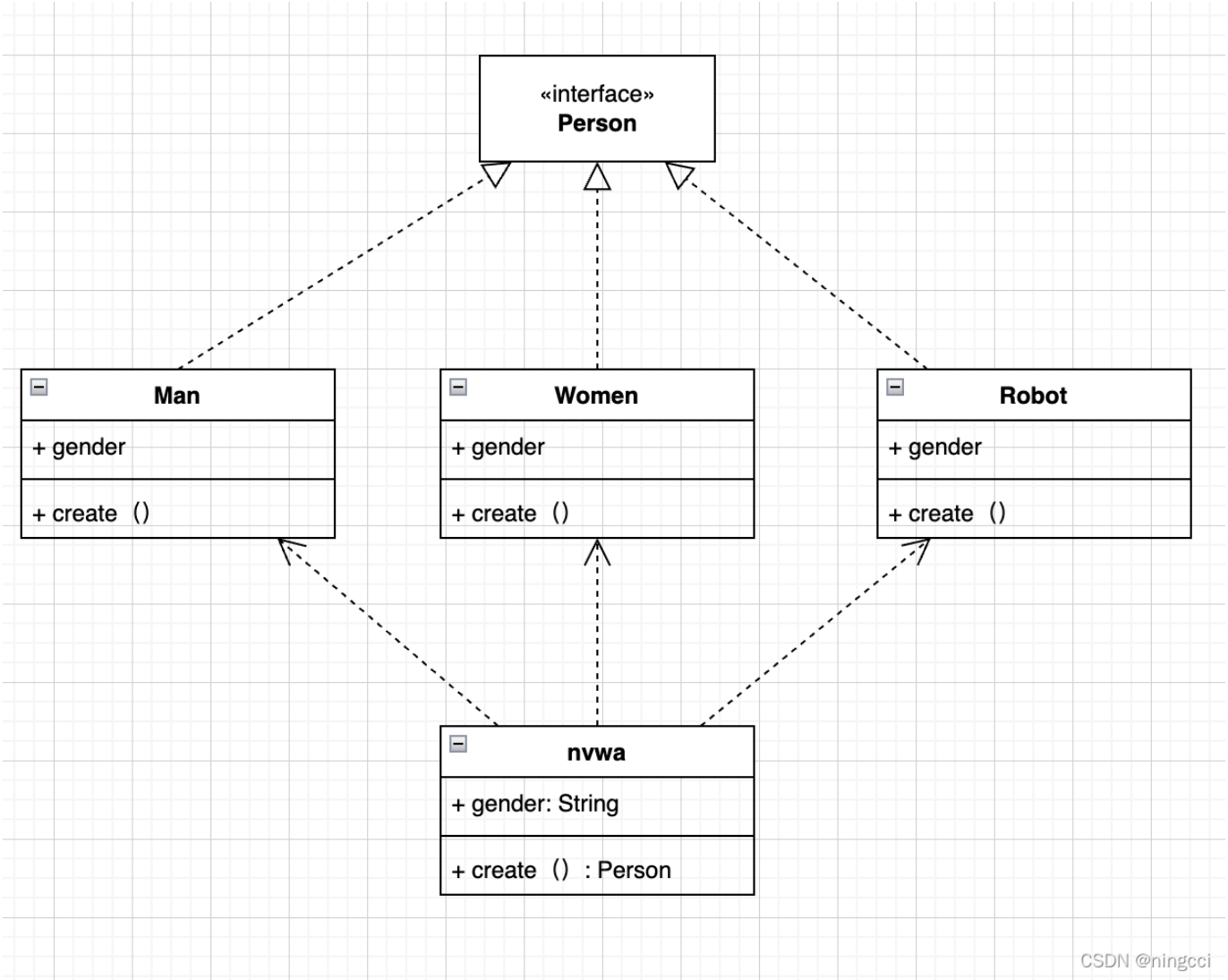
```java
package simple_factory_pattern;

public class Nvwa {
    public static Person getPerson(String person1) throws Exception
    {
        if (person1.equalsIgnoreCase("M")){
            return new Man();
        }
        else if (person1.equalsIgnoreCase("W")){
            return new Women();
        }else
        {
            throw new Exception("对不起，暂时无法创造该人。");
        }
    }
}
```

```java
package simple_factory_pattern;

public class Test {
    public static void main(String[] args) throws Exception {
        Person person;
        String person1 = "W";
        person = Nvwa.getPerson(person1);
        person.create();
    }
}
```

增加Robot之后的类图



```java
package simple_factory_pattern;

public class Robot implements Person{
    @Override
    public void create() {
        System.out.println("成功创造了机器人...");
    }
}
```

```java
package simple_factory_pattern;

public class Nvwa {
    public static Person getPerson(String person1) throws Exception
    {
        if (person1.equalsIgnoreCase("M")){
            return new Man();
        }
        else if (person1.equalsIgnoreCase("W")){
            return new Women();
        }else if (person1.equalsIgnoreCase("R")){
            return new Robot();
        }
        else
        {
            throw new Exception("对不起，暂时无法创造该人。");
        }
    }
}
```

简单工厂模式（Simple Factory Pattern）

- 简单工厂模式每增加一个产品就要增加一个具体产品类和一个对应的具体工厂类，这增加了系统的复杂度，违背了“开闭原则”。

- 应用场景：对于产品种类相对较少的情况，考虑使用简单工厂模式。使用简单工厂模式的客户端只需要传入工厂类的参数，不需要关心如何创建对象的逻辑，可以很方便地创建所需产品。

优点和缺点

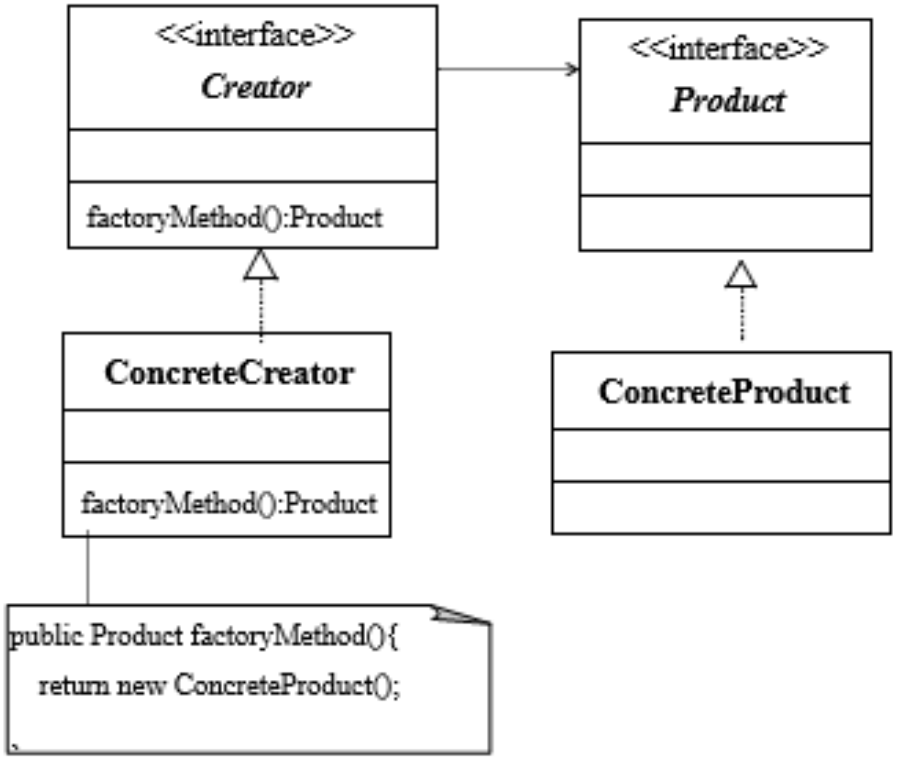
优点:

1. 工厂类包含必要的逻辑判断，可以决定在什么时候创建哪一个产品的实例。客户端可以免除直接创建产品对象的职责，很方便的创建出相应的产品。工厂和产品的职责区分明确。
2. 客户端无需知道所创建具体产品的类名，只需知道参数即可。
3. 也可以引入配置文件，在不修改客户端代码的情况下更换和添加新的具体产品类。

缺点:

1. 简单工厂模式的工厂类单一，负责所有产品的创建，职责过重，一旦异常，整个系统将受影响。且工厂类代码会非常臃肿，违背高聚合原则。
2. 使用简单工厂模式会增加系统中类的个数（引入新的工厂类），增加系统的复杂度和理解难度
3. 系统扩展困难，一旦增加新产品不得不修改工厂逻辑，在产品类型较多时，可能造成逻辑过于复杂
4. 简单工厂模式使用了 static 工厂方法，造成工厂角色无法形成基于继承的等级结构。

工厂方法模式



结构中包含以下4种角色：

- **抽象产品（Product）**：抽象类或接口，负责定义具体产品必须实现的方法。
- **具体产品（ConcreteProduct）**：如果Product是一个抽象类，那么具体产品是Product的子类；如果Product是一个接口，那么具体产品是实现Product接口的类。
- **构造者（Creator）**：一个接口或抽象类。构造者负责定义一个称作工厂方法的抽象方法，该方法返回具体产品类的实例。
- **具体构造者（ConcreteCreator）**：如果构造者是抽象类，具体构造者是构造者的子类；如果构造者是接口，具体构造者是实现构造者的类。具体构造者重写工厂方法使该方法返回具体产品的实例。

模式的结构

- 抽象产品（Product）
  - 当系统准备为用户提供某个类的子类的实例，又不想让用户代码和该子类形成耦合时，就可以使用工厂方法模式来设计系统。
  - 工厂方法模式的关键是在一个接口或抽象类中定义一个抽象方法，该方法要求返回某个类的子类的实例。
- 具体产品（ConcreteProduct）
- 构造者（Creator）
- 具体构造者（ConcreteCreator）

案例：圆珠笔与笔芯

- 用户希望自己的圆珠笔能使用不同颜色（红、蓝、黑）的笔芯。
  - 用户的圆珠笔希望使用各种颜色的笔芯，因此这里抽象产品（Product）角色是名字为**PenCore的抽象类，该类的不同子类可以提供相应颜色的笔芯。**
  - RedPenCore，BluePenCore和BlackPenCore类是三个具体产品角色
  - 构造者（Creator）角色是：名字为CreatorPenCore的抽象类
  - RedCoreCreator，BlueCoreCreator，BlackCoreCreator类是具体构造者角色。

1 抽象产品（Product）

```
public abstract class PenCore{
   String color;
   public abstract void writeWord(String s);
}
```

2. 具体产品（ConcreteProduct）

```
public class RedPenCore extends PenCore{
    RedPenCore(){
      color="红色";
    }
    public void writeWord(String s){
       System.out.println("写出"+color+"的字:"+s);
    }
}
public class BluePenCore extends PenCore{
    BluePenCore(){
      color="蓝色";
    }
    public void writeWord(String s){
       System.out.println("写出"+color+"的字:"+s);
    }
}
public class BlackPenCore extends PenCore{
    BlackPenCore(){
      color="黑色";
    }
    public void writeWord(String s){
       System.out.println("写出"+color+"的字:"+s);
    }
}
```

3. 构造者（Creator）

```
public abstract class PenCoreCreator{
    public abstract PenCore getPenCore(); //工厂方法
}
```

4. 具体构造者（ConcreteCreator）

```
public class RedCoreCreator extends PenCoreCreator{
   public PenCore getPenCore() { //重写工厂方法
      return new RedPenCore();
   }
}

public class BlueCoreCreator extends PenCoreCreator{
   public PenCore getPenCore() { //重写工厂方法
      return new BluePenCore();
   }
}

public class BlackCoreCreator extends PenCoreCreator{
    public PenCore getPenCore() { //重写工厂方法
      return new BlackPenCore();
   }
}
```

5、模式的使用

```
public class BallPen{
   PenCore core;
   public void usePenCore(PenCore core){
      this.core=core;
   }
   public void write(String s) {
      core.writeWord(s);
   }
}

public class Application{
    public static void main(String args[]){
       PenCore penCore;   //笔芯
       PenCoreCreator creator;    //笔芯构造者
       BallPen ballPen=new BallPen(); //圆珠笔
       creator =new RedCoreCreator();
       penCore = creator.getPenCore(); //使用工厂方法返回笔芯
       ballPen.usePenCore(penCore);
       ballPen.write("你好,很高兴认识你");
       creator =new BlueCoreCreator();
       penCore = creator.getPenCore();
       ballPen.usePenCore(penCore);
       ballPen.write("nice to meet you");
       creator =new BlackCoreCreator();
       penCore = creator.getPenCore();
       ballPen.usePenCore(penCore);
       ballPen.write("how are you");
    }
}
```

优点

- 用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程。
- 灵活性增强，对于新产品的创建，只需多写一个相应的工厂类。
- 典型的解耦框架。高层模块只需要知道产品的抽象类，无须关心其他实现类，满足迪米特法则、依赖倒置原则和里氏替换原则。

缺点

- 类的个数容易过多，增加复杂度
- 增加了系统的抽象性和理解难度
- 抽象产品只能生产一种产品，此弊端可使用抽象工厂模式解决。

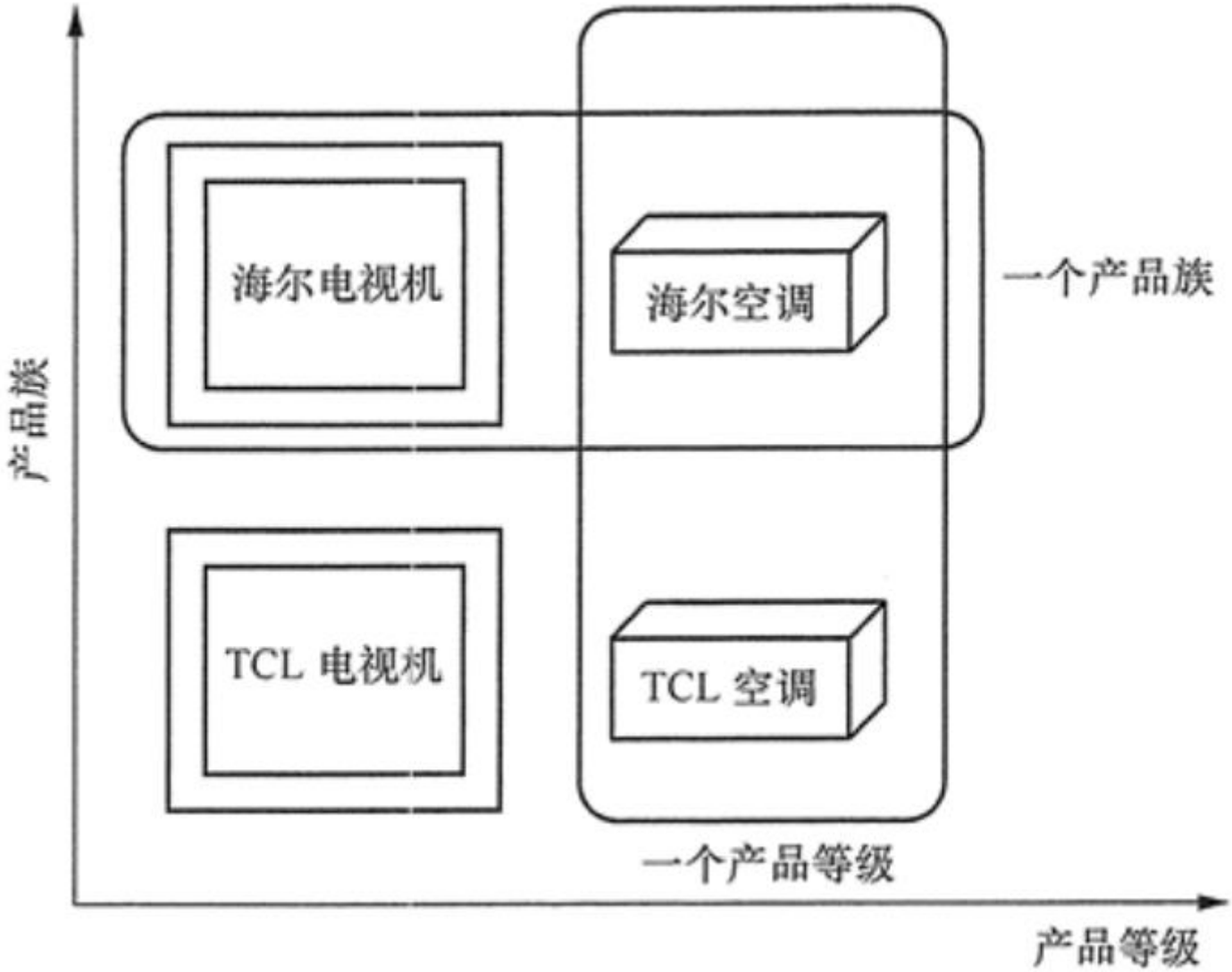
工厂方法模式的使用场景

- 客户只知道创建产品的工厂名，而不知道具体的产品名。如 TCL 电视工厂、海信电视工厂等。
- 创建对象的任务由多个具体子工厂中的某一个完成，而抽象工厂只提供创建产品的接口。
- 客户不关心创建产品的细节，只关心产品的品牌

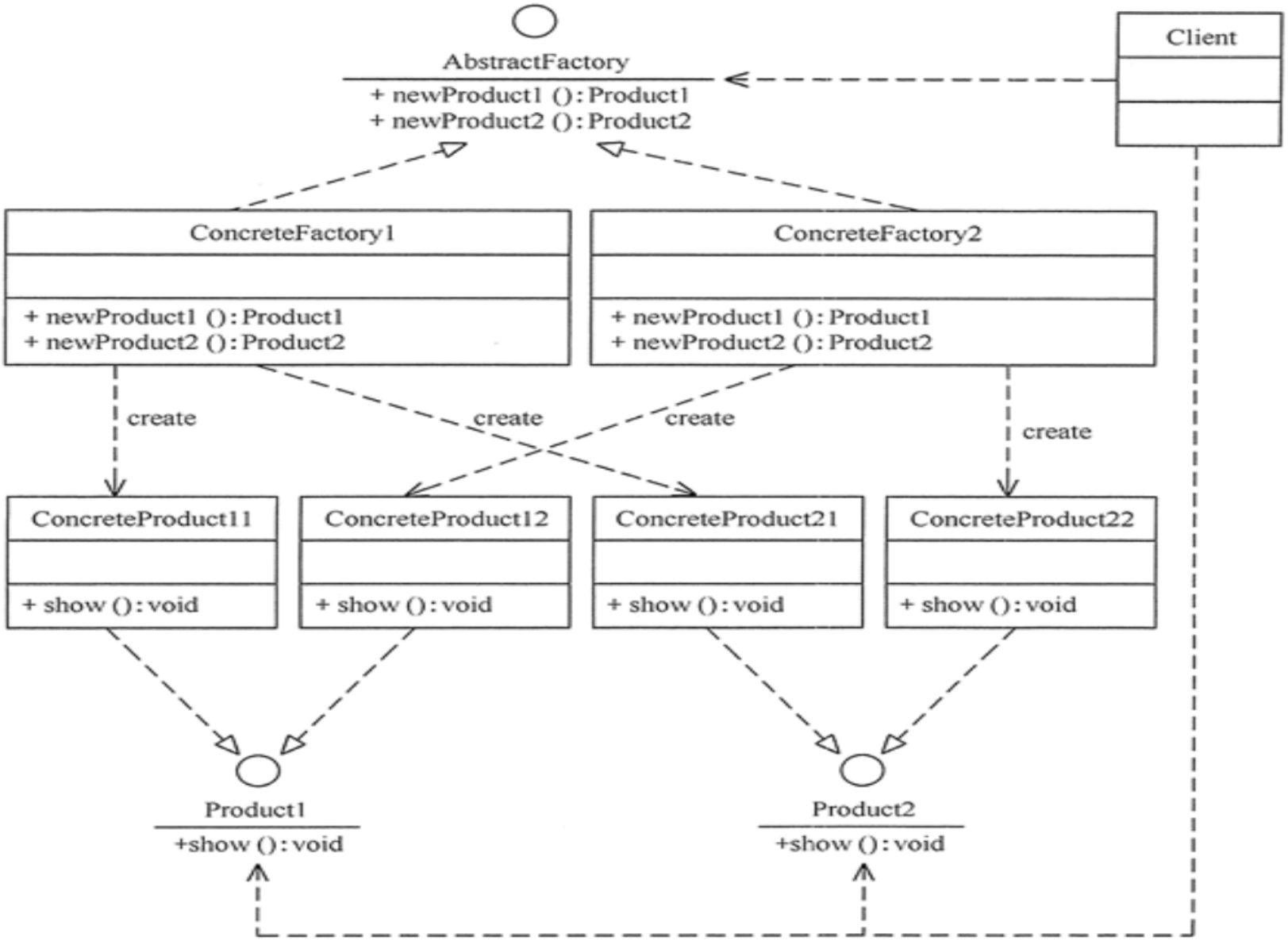
抽象工厂（AbstractFactory）模式

- 是一种为访问类提供一个创建一组相关或相互依赖对象的接口，且访问类无须指定所要产品的具体类就能得到同族的不同等级的产品的模式结构。
- 抽象工厂模式是工厂方法模式的升级版本，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品

电器工厂的产品等级与产品族



抽象工厂模式的结构图

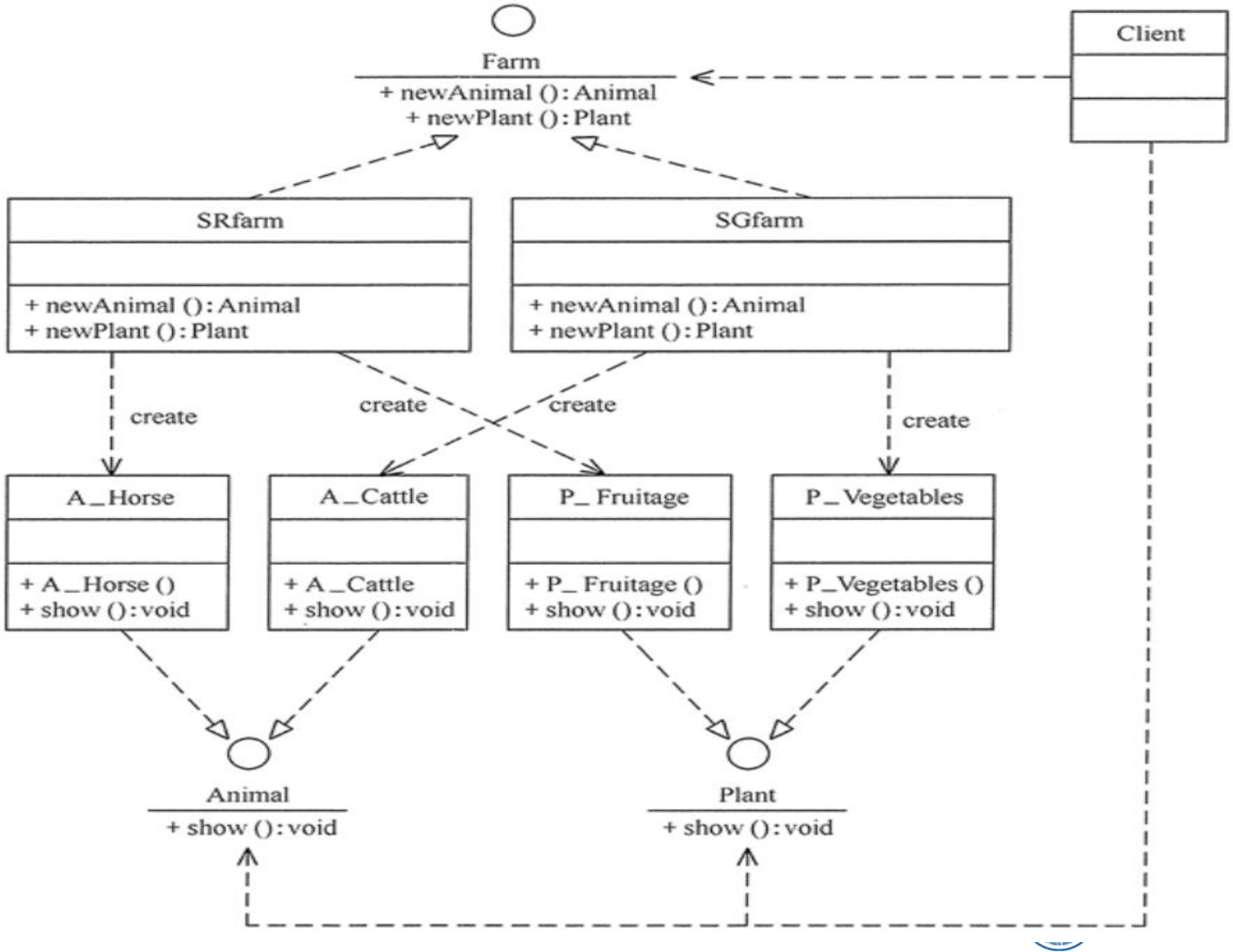

AbstractFactory
+ newProduct1 ():Product1
+ newProduct2 ():Product2
Client
ConcreteFactory1
+ newProduct1 ():Product1
+ newProduct2 ():Product2
ConcreteFactory2
+ newProduct1 ():Product1
+ newProduct2 ():Product2
create
create
create
create
ConcreteProduct11
+ show ():void
ConcreteProduct12
+ show ():void
ConcreteProduct21
+ show ():void
ConcreteProduct22
+ show ():void
Product1
+show ():void
Product2
+show ():void

模式的结构

抽象工厂模式的主要角色如下。

1. 抽象工厂（Abstract Factory）：提供了创建产品的接口，它包含多个创建产品的方法 newProduct()，可以创建多个不同等级的产品。
2. 具体工厂（Concrete Factory）：主要是实现抽象工厂中的多个抽象方法，完成具体产品的创建。
3. 抽象产品（Product）：定义了产品的规范，描述了产品的主要特性和功能，抽象工厂模式有多个抽象产品。
4. 具体产品（ConcreteProduct）：实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间是多对一的关系。

案例：农场类的结构图



思考题：

- 如何编写代码？

抽象工厂模式优点

- 抽象工厂模式除了具有工厂方法模式的优点外，其他主要优点如下。
  - 可以在类的内部对产品族中相关联的多等级产品共同管理，而不必专门引入多个新的类来进行管理。
  - 当需要产品族时，抽象工厂可以保证客户端始终只使用同一个产品的产品组。
  - 抽象工厂增强了程序的可扩展性，当增加一个新的产品族时，不需要修改原代码，满足开闭原则。

抽象工厂模式缺点

- 其缺点是：
  - 当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。增加了系统的抽象性和理解难度。

抽象工厂模式通常适用于以下场景

1. 当需要创建的对象是一系列相互关联或相互依赖的产品族时，如电器工厂中的电视机、洗衣机、空调等。
2. 系统中有多个产品族，但每次只使用其中的某一族产品。如有人只喜欢穿某一个品牌的衣服和鞋。
3. 系统中提供了产品的类库，且所有产品的接口相同，客户端不依赖产品实例的创建细节和内部结构。

小结

- 1.策略模式的核心是定义一系列算法,把它们一个个封装起来,并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。
- 2.访问者模式的核心是在不改变各个元素的类的前提下定义作用于这些元素的新操作。
- 3.装饰模式的核心是动态地给对象添加一些额外的职责。
- 4.适配器模式的核心是将一个类的接口转换成客户希望的另外一个接口。
- 5.工厂方法模式的核心是把类的实例化延迟到其子类。