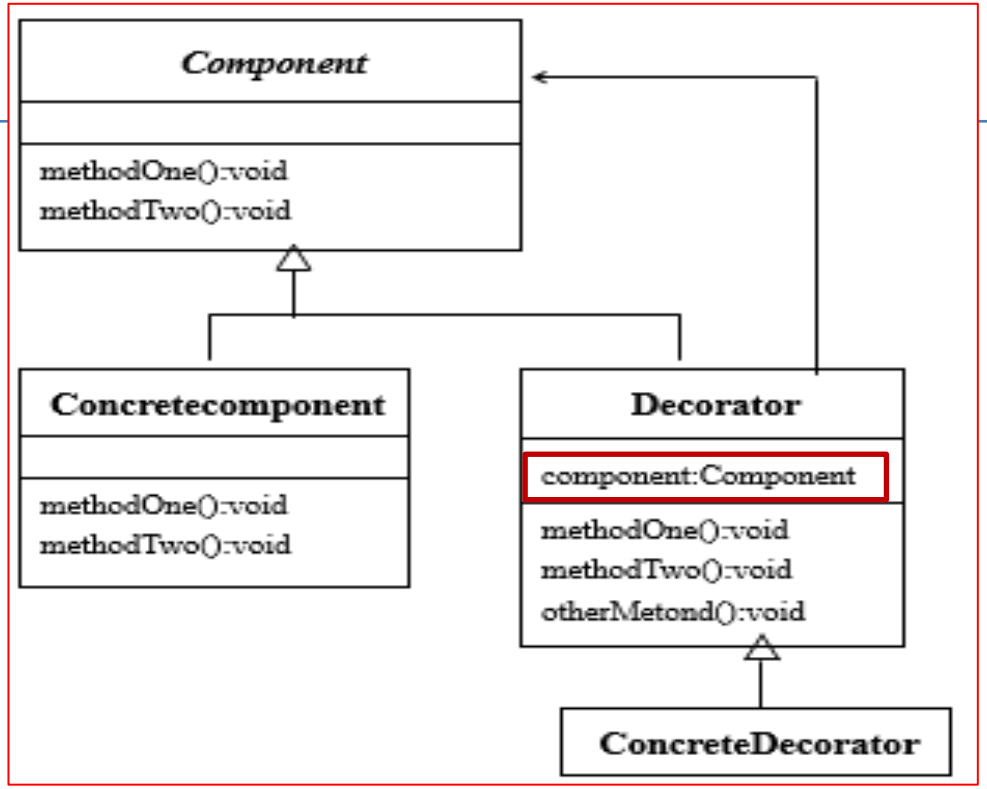
完善类的设计_2

主讲老师：申雪萍

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;
  }
}
```

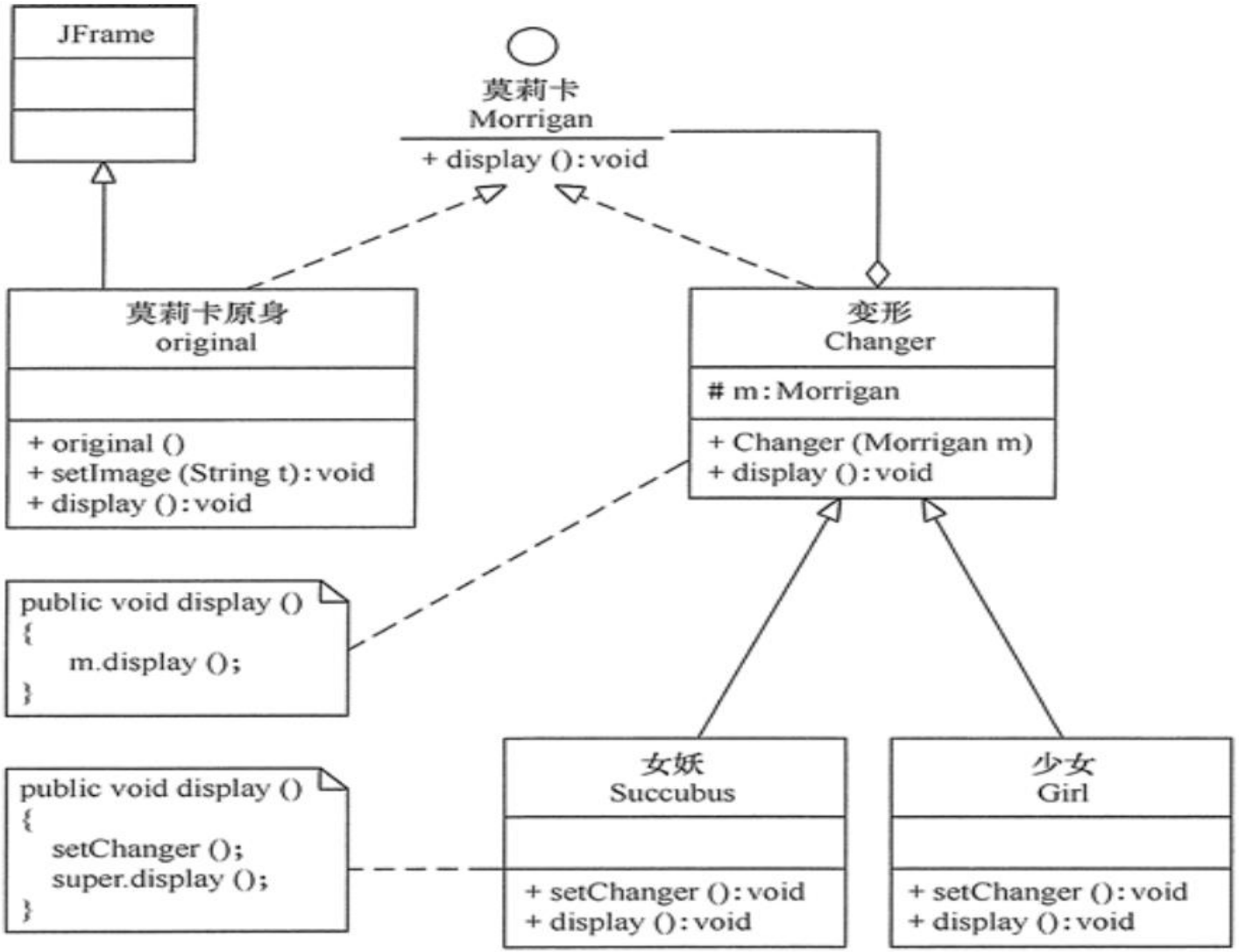
模式的使用：

```
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（1）

- 通过继承也可以改进对象的行为，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设计的基本原则之一：少用继承，多用组合。例如：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

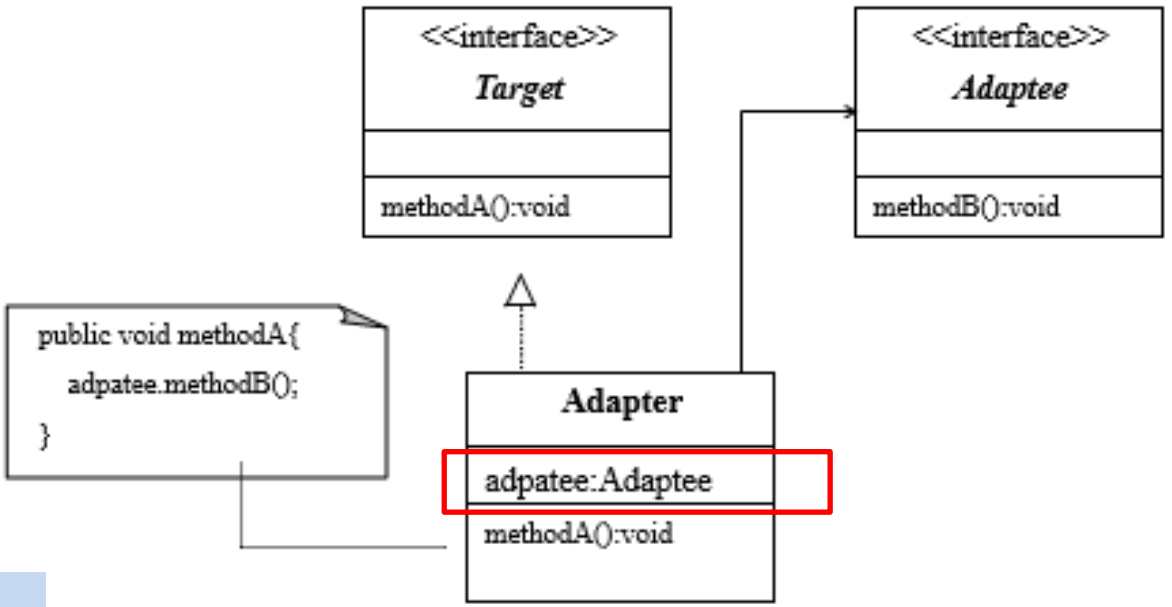
- 此案例中，被装饰者和装饰者是松耦合关系。由于装饰（Decorator）仅仅依赖于抽象组件（Component），因此具体装饰只知道它要装饰的对象是抽象组件的某一个子类的实例，但不需要知道是哪一个具体子类装饰模式，相比生成子类更为灵活。程序的弹性和可扩展性更优。

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。

- 不希望采用继承来增强对象功能，以免影响系统的扩展和维护。

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
    }
}
```

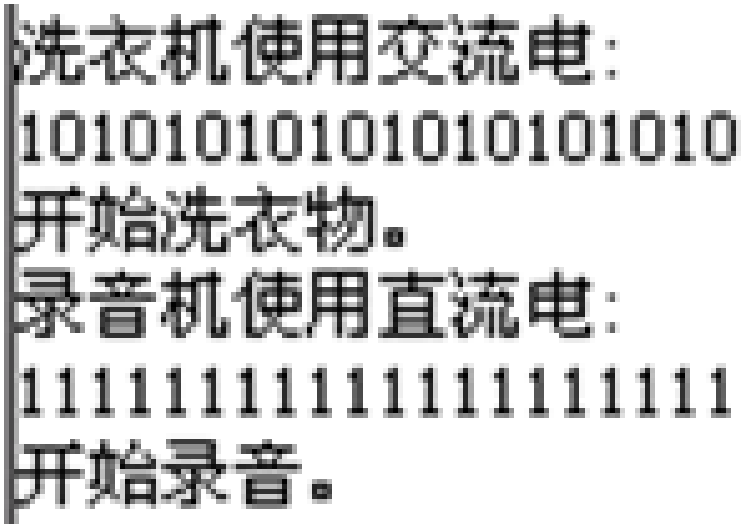
```
class Recorder { //录音机使用直流电
   String name;
   Recorder(){
      name="录音机";
   }
   Recorder(String s){
      name=s;
   }
   public void turnOn(DirectCurrent a){
      String s=a.giveDirectCurrent();
      System.out.println(name+"使用直流电:\n"+s);
      System.out.println("开始录音。");
   }
}
```

```
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。");
   }
}
```

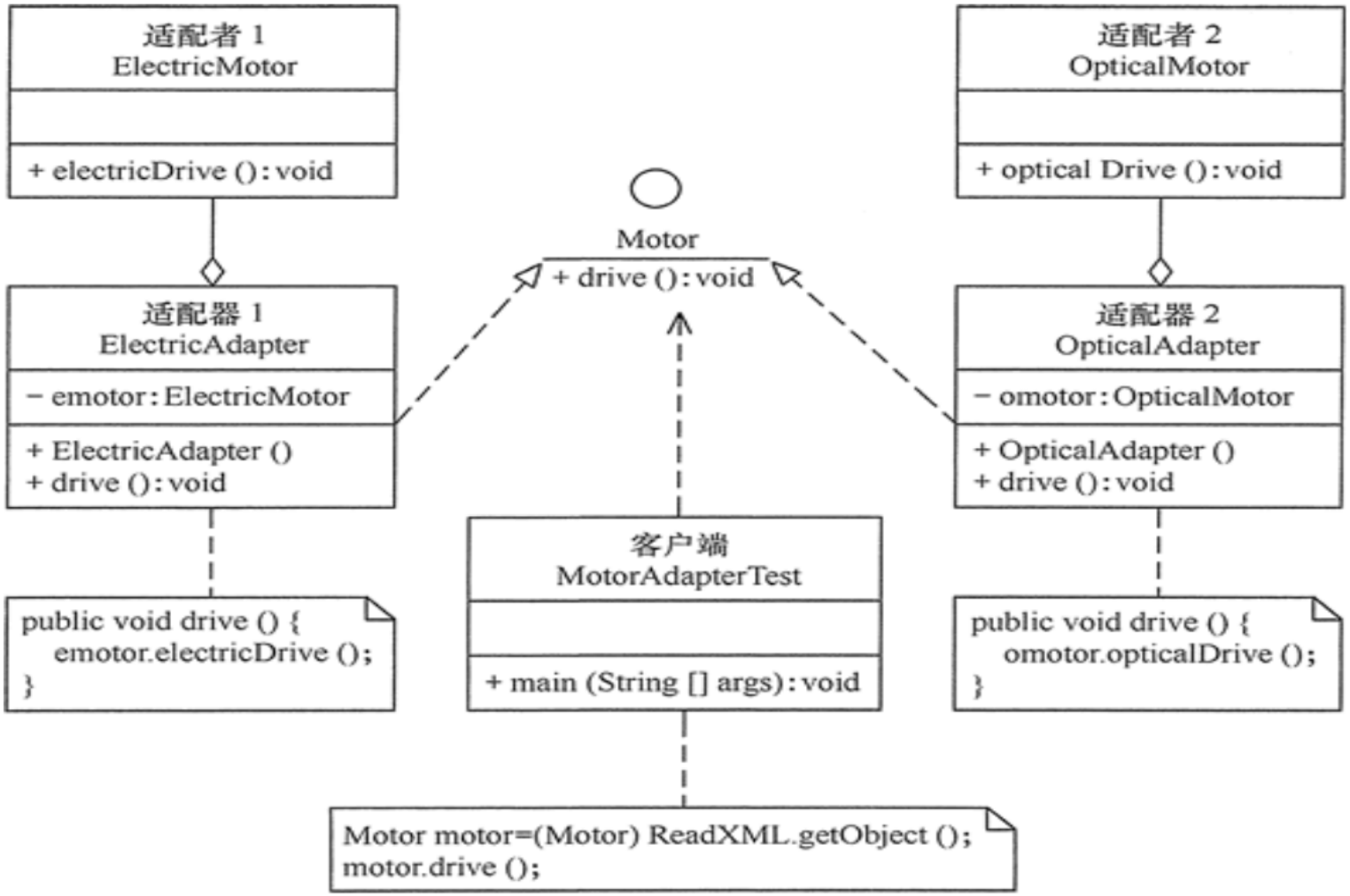
模式的使用：

```
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：



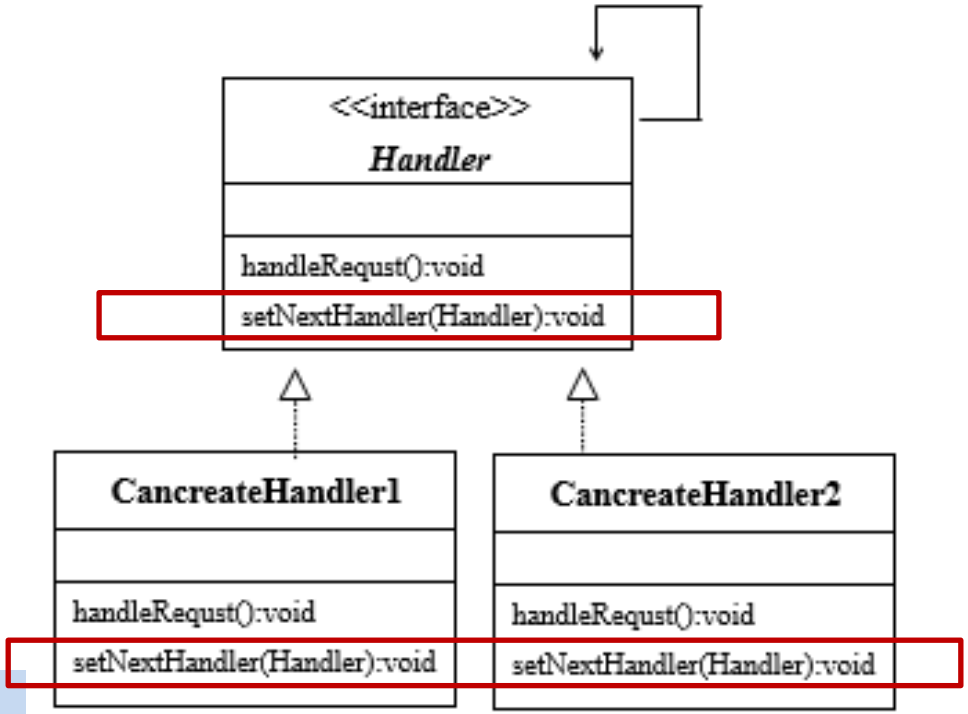
案例：发动机适配器的结构图



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。

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。

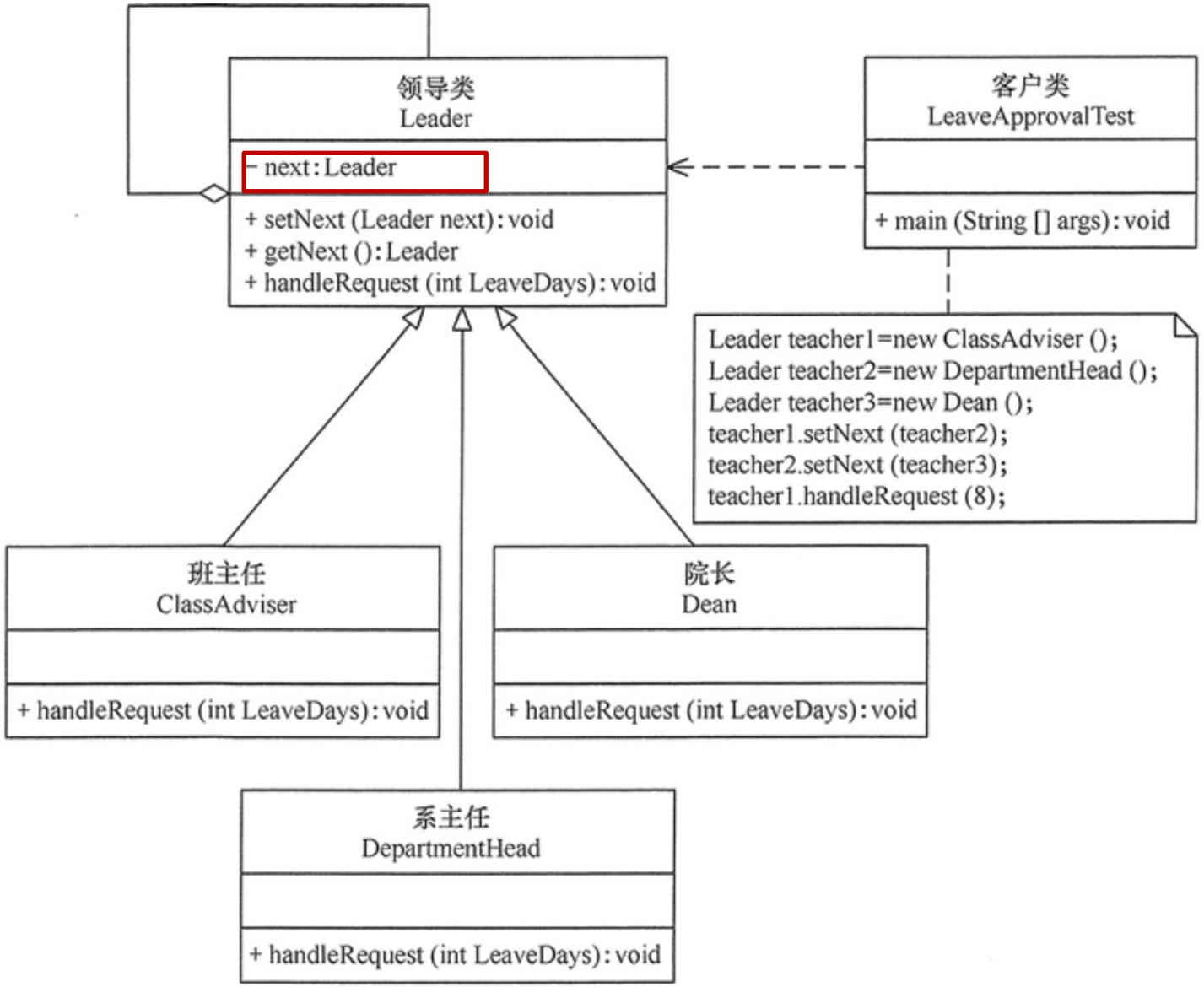
模式的结构

- **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；

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

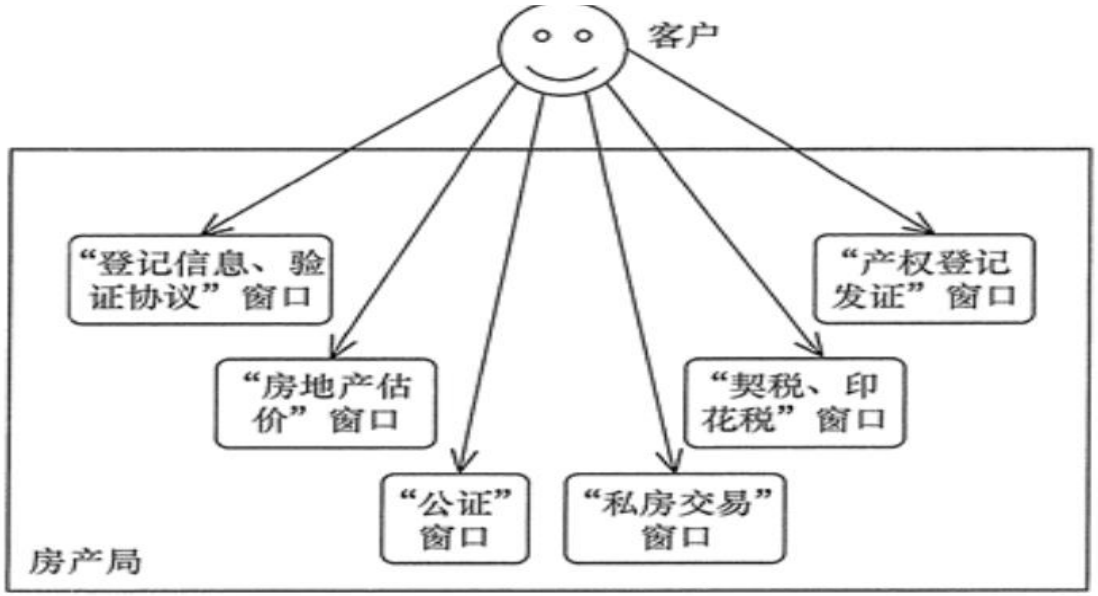
- 假如电影票7元，购票者购买2张电影票，给售票员100元面值钞票一张。那么实际上售票员找赎86元过程如下：
  - 首先在50元面值的钱盒中看能否完成任务，或部分任务。50元面值的钱盒发现能找赎86元中的50元（贡献一张50元的钞票），即只完成部分任务，因此把剩余的36元任务交给下一个20元面值的钱盒，20元面值的钱盒发现能完成36元中的20元（贡献一张20面值的钞票），因此把剩余的16元任务交给下一个10元面值的钱盒。依次类推，如果在后续某个钱盒完成了自己的找赎任务，那么售票员就完成了找赎任务，如果一直到最后一个钱盒（假设是1元面值的钱盒），该钱盒也无法完成自己的找赎任务，那么售票员就无法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

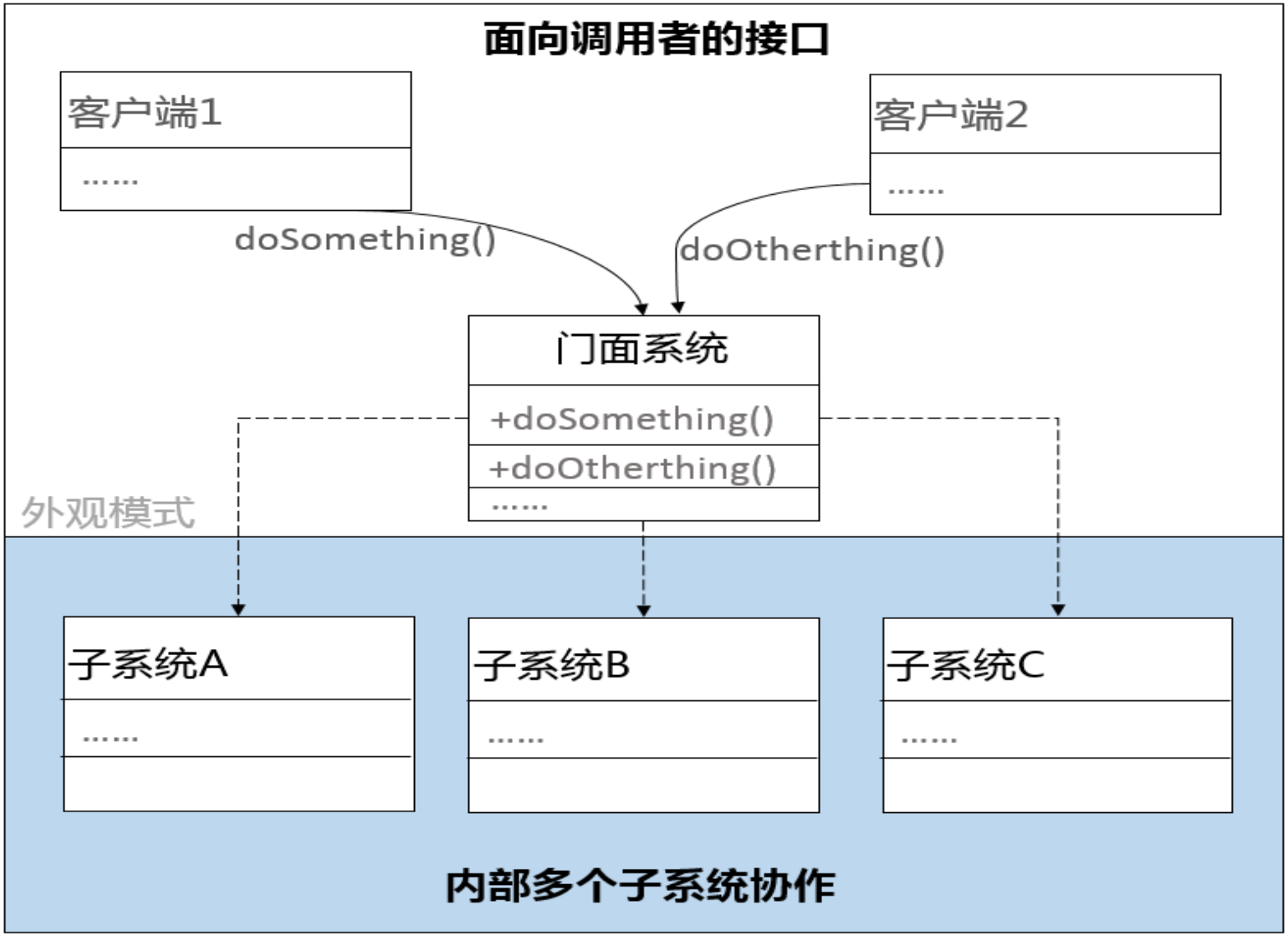
外观模式（Facade模式，也叫门面模式）

- 客户去当地房产局办理房产证过户要遇到的相关部门；



- 现实生活中，上面的例子比比皆是，例如注册一家公司，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。

外观（Facade）模式的结构图

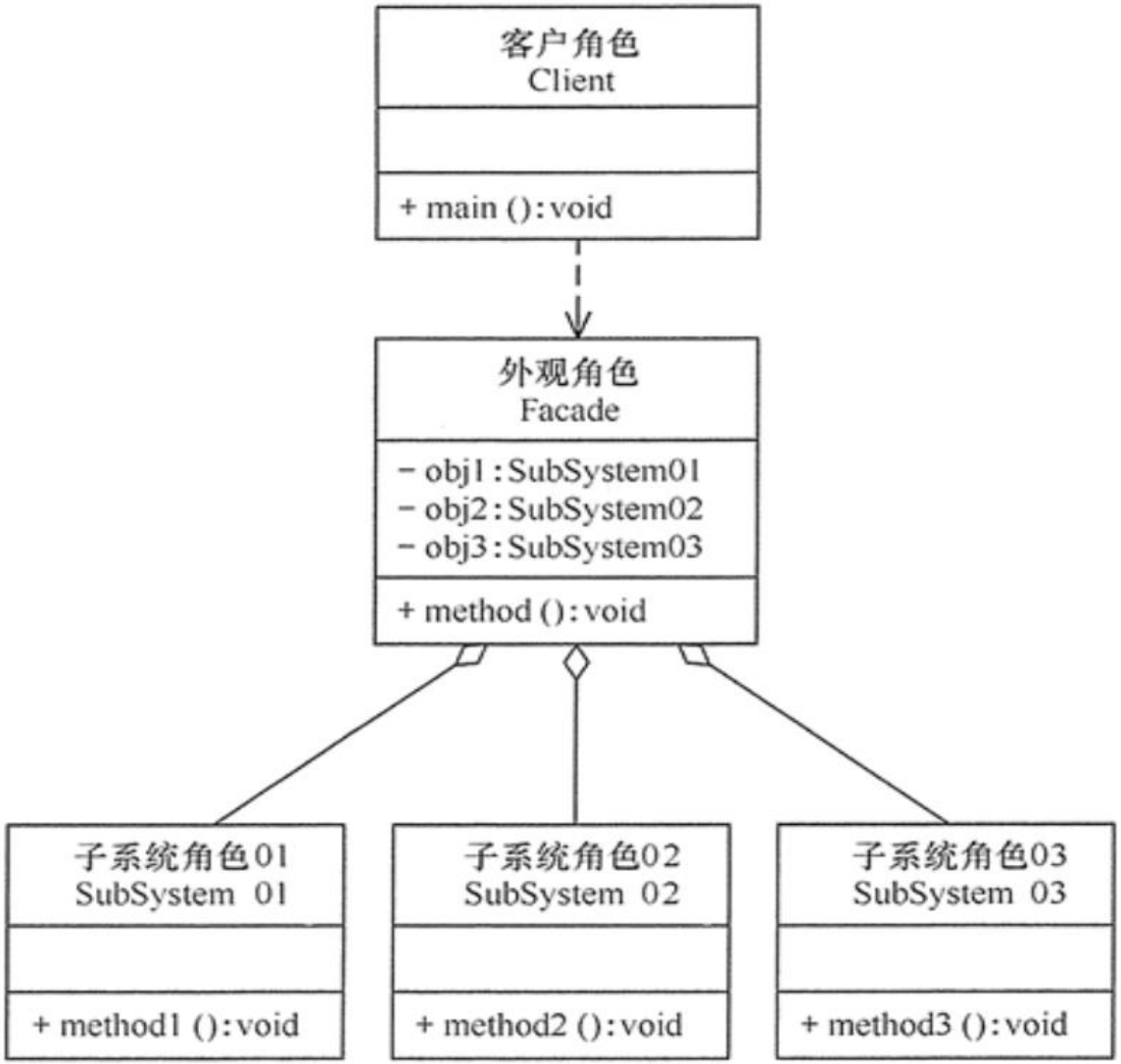


外观模式的定义与特点

- 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。

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

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

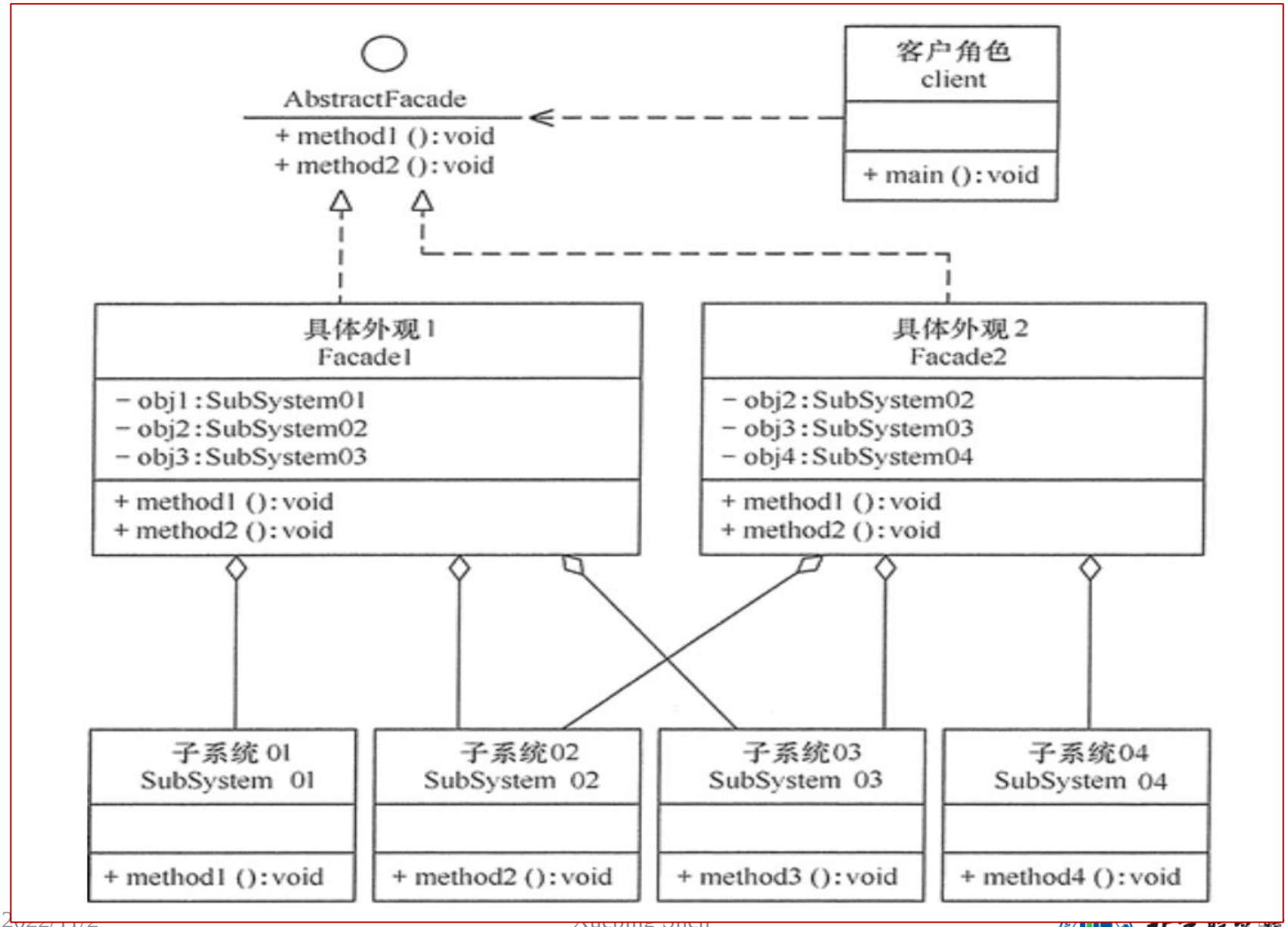
外观（Facade）模式的主要缺点

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

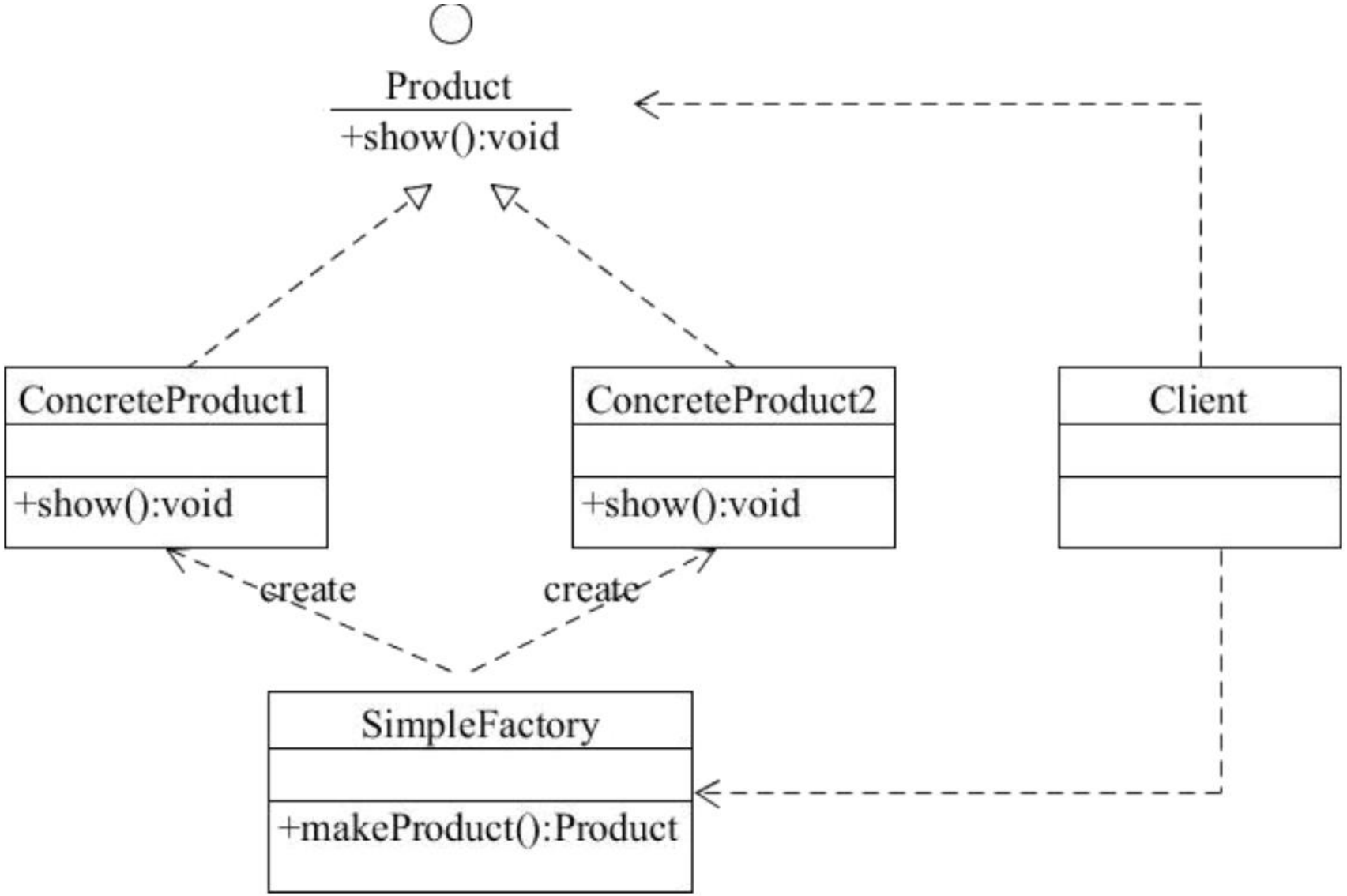
简单工厂模式（Simple Factory Pattern）

- 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。

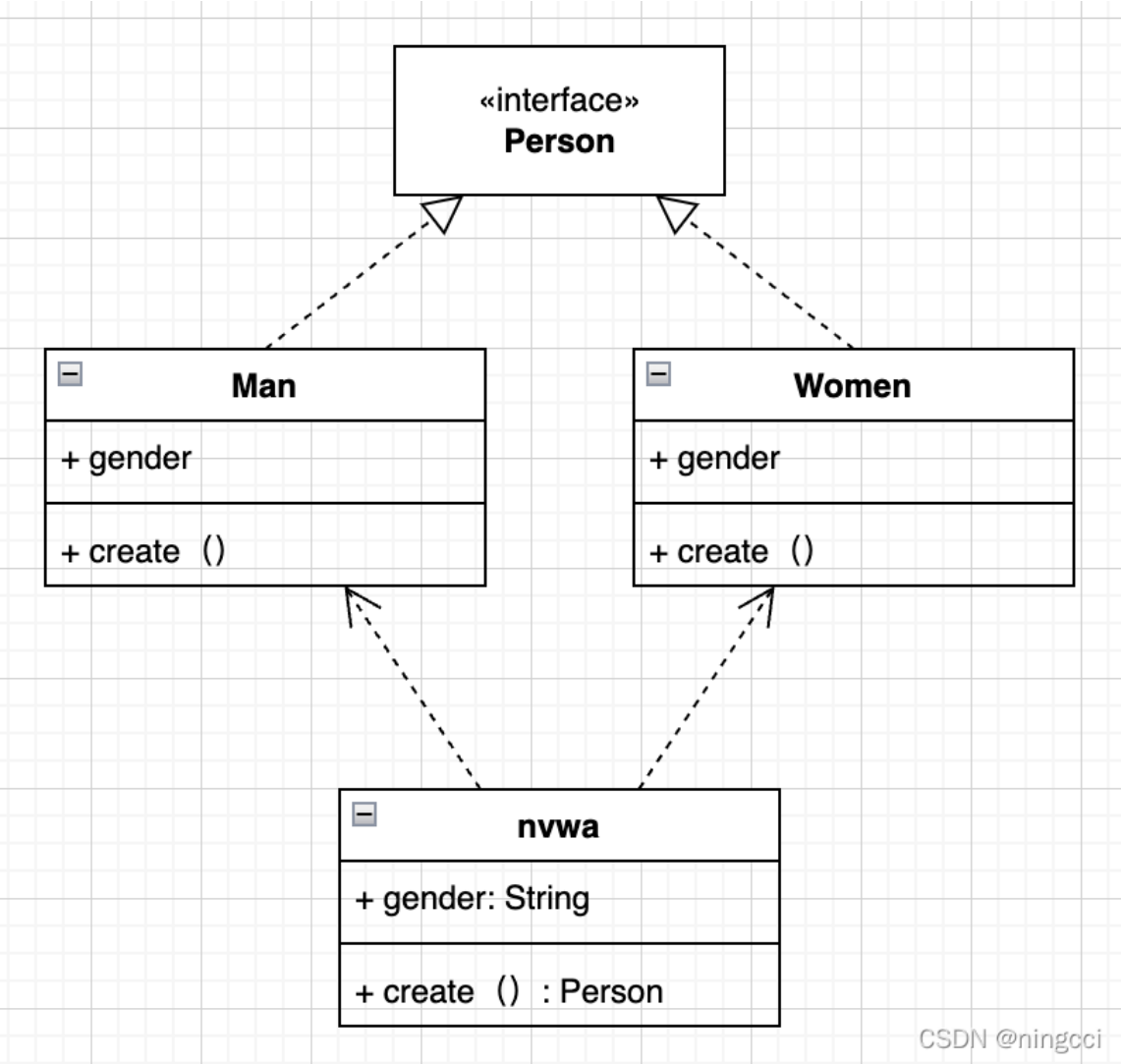
简单工厂模式的结构图


Product
+show():void
ConcreteProduct1
+show():void
ConcreteProduct2
+show():void
Client
create
create
SimpleFactory
+makeProduct():Product

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...");
    }
}
```

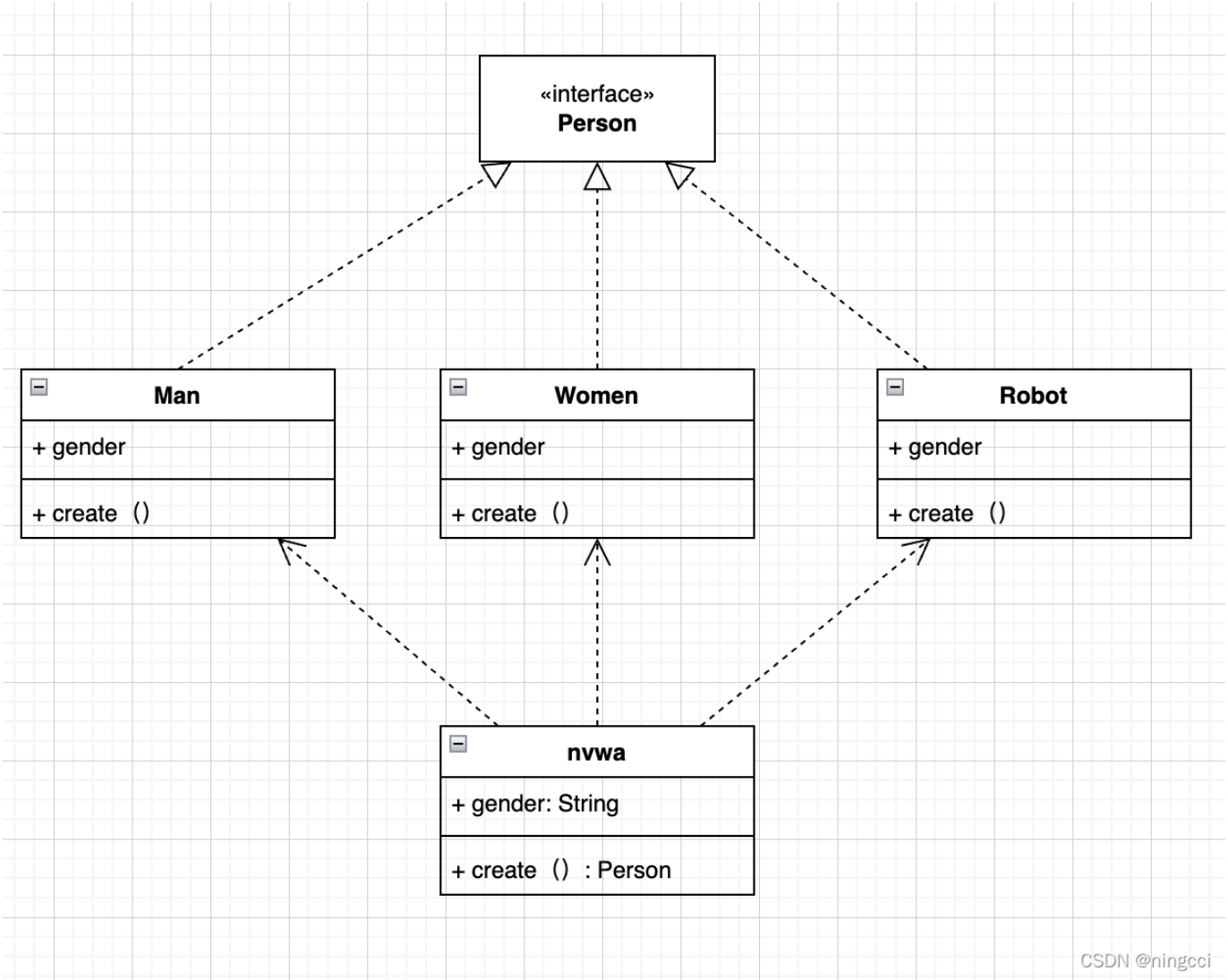
```java
package simple_factory_pattern;

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

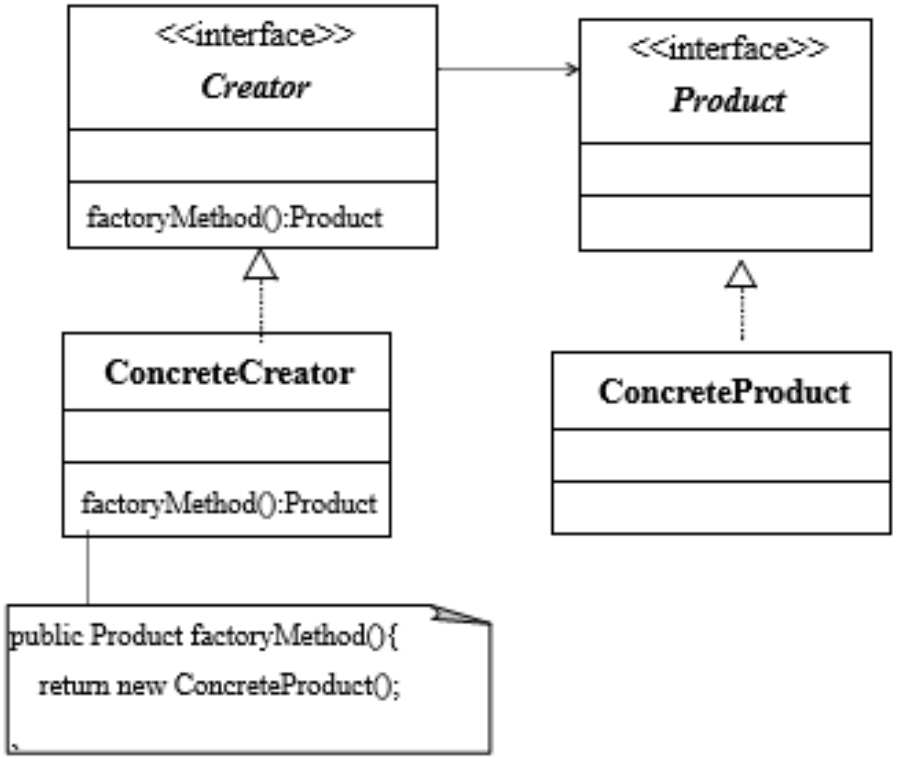
优点:

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。

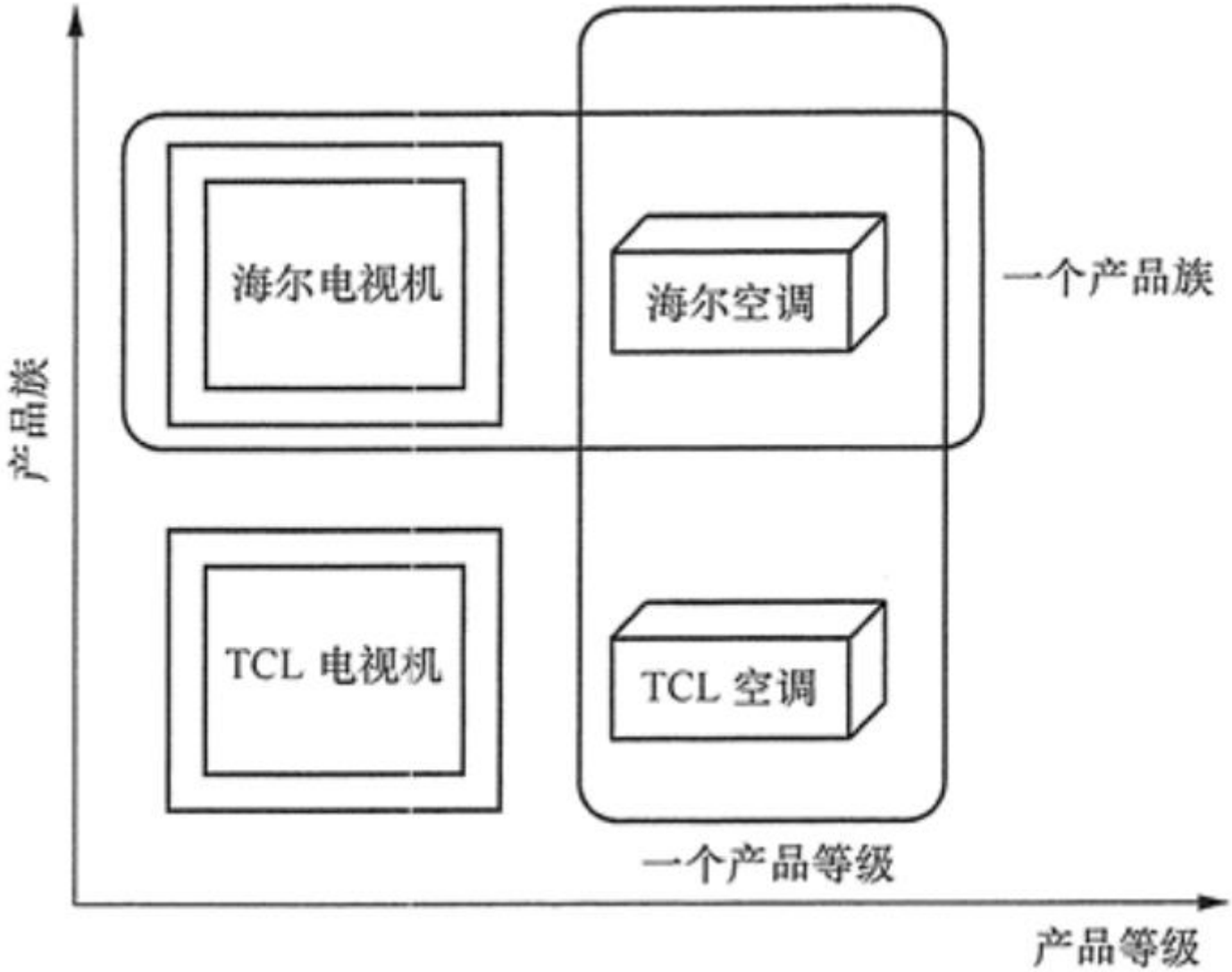
工厂方法模式的使用场景

- 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

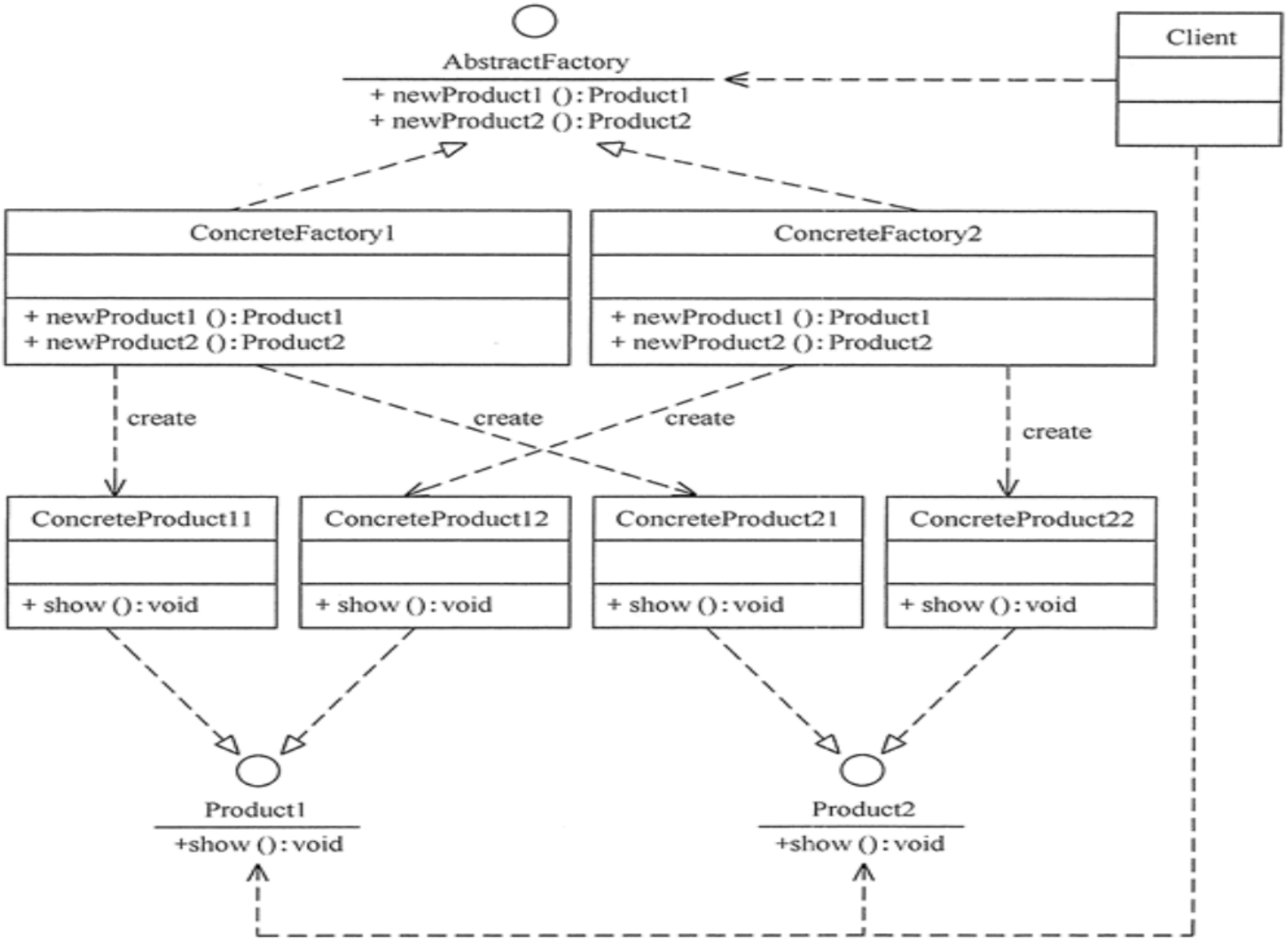
抽象工厂（AbstractFactory）模式

- 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

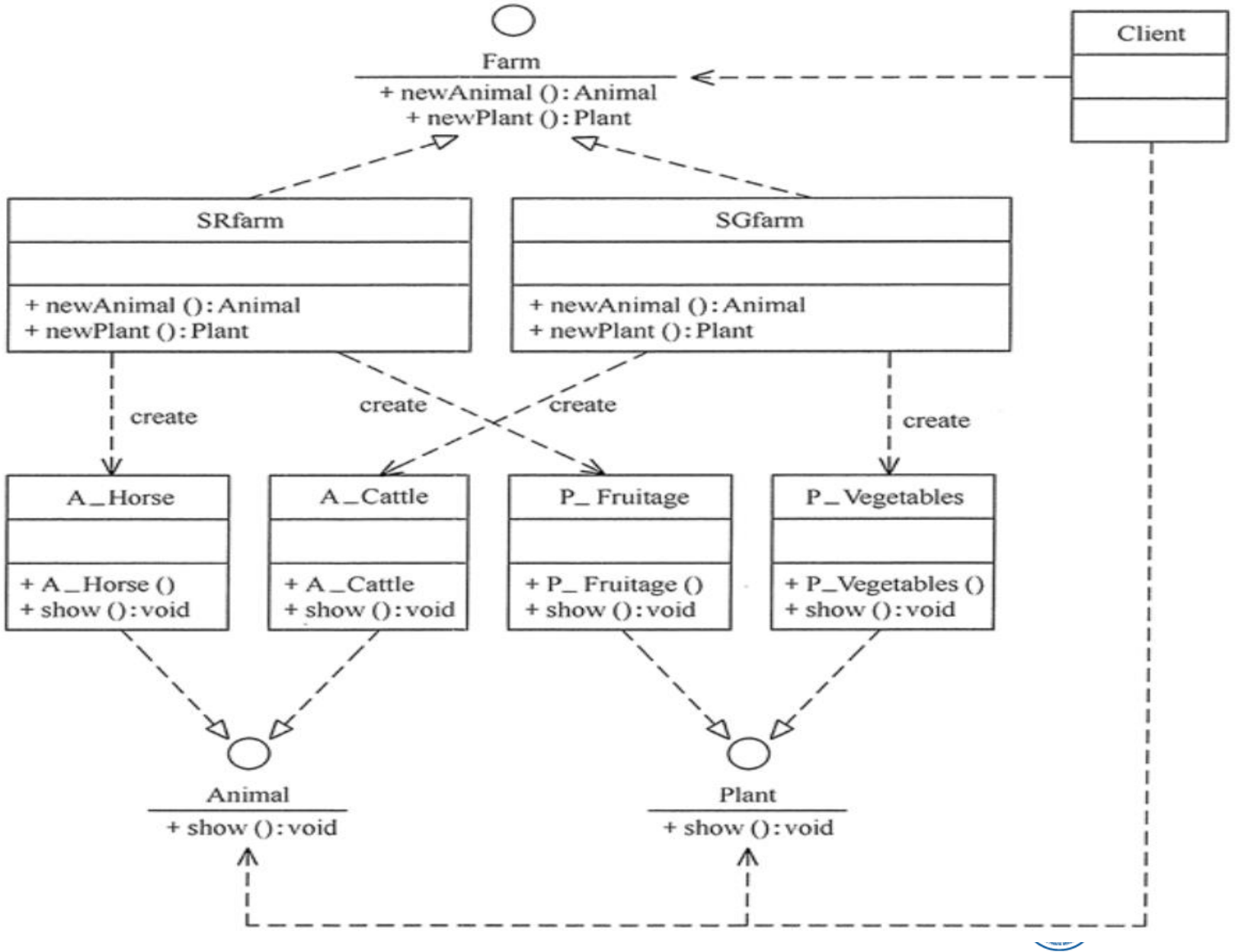

AbstractFactory
+ newProduct1 ():Product1
+ newProduct2 ():Product2
Client
ConcreteFactory1
+ newProduct1 ():Product1
+ newProduct2 ():Product2
ConcreteFactory2
+ newProduct1 ():Product1
+ newProduct2 ():Product2
create
create
create
create
ConcreteProduct11
+ show ():void
ConcreteProduct12
+ show ():void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