总体来说设计模式分为三大类：创建型模式（抽象工厂模式，单例模式，工厂方法模式），结构型模式（适配器模式，装饰器模式，代理模式，门面模式），行为型模式（观察者模式，责任链模式）

**工厂模式：**

主要优点有：可以使代码结构清晰，有效地封装变化；对调用者屏蔽具体的产品类；降低耦合度

**简单工厂模式**

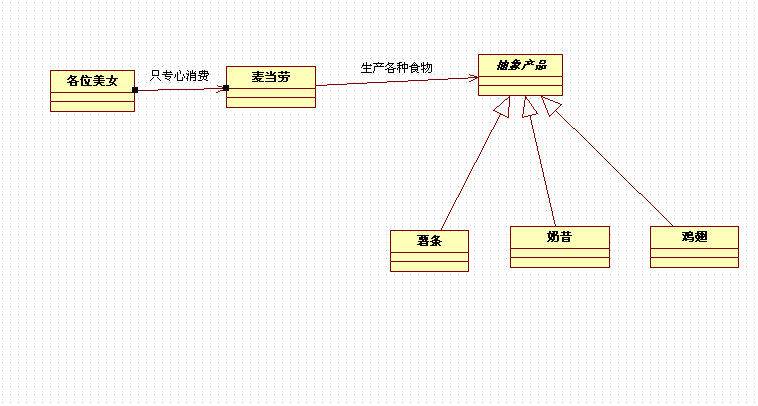
简单工厂模式（Simple Factory Pattern）属于类的创新型模式，又叫静态工厂方法模式（Static FactoryMethod Pattern）,是通过专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

 工厂角色（Creator）：这是简单工厂模式的核心，由它负责创建所有的类的内部逻辑。当然工厂类必须能够被外界调用，创建所需要的产品对象。

       抽象（Product）产品角色：简单工厂模式所创建的所有对象的父类，注意，这里的父类可以是接口也可以是抽象类，它负责描述所有实例所共有的公共接口。

       具体产品（Concrete Product）角色：简单工厂所创建的具体实例对象，这些具体的产品往往都拥有共同的父类。

**简单工厂模式使用场景分析及代码实现：**



       实现代码如下：

       新建立一个食物的接口：

|  |
| --- |
| **package** com.diermeng.designPattern.SimpleFactory;    /\*   \* 产品的抽象接口   \*/  **public interface** Food {      /\*       \* 获得相应的食物       \*/  **public void** get();  } |

接下来建立具体的产品：麦香鸡和薯条

|  |
| --- |
| **package** com.diermeng.designPattern.SimpleFactory.impl;  **import** com.diermeng.designPattern.SimpleFactory.Food;    /\*   \* 麦香鸡对抽象产品接口的实现   \*/  **public class** McChicken **implements** Food{      /\*       \* 获取一份麦香鸡       \*/  **public void** get(){          System.*out*.println("我要一份麦香鸡");      }  } |

|  |
| --- |
| **package** com.diermeng.designPattern.SimpleFactory.impl;  **import** com.diermeng.designPattern.SimpleFactory.Food;    /\*   \* 薯条对抽象产品接口的实现   \*/  **public class** Chips **implements** Food{      /\*       \* 获取一份薯条       \*/  **public void** get(){          System.*out*.println("我要一份薯条");      }  } |

现在建立一个食物加工工厂：

|  |
| --- |
| **package** com.diermeng.designPattern.SimpleFactory.impl;  **import** com.diermeng.designPattern.SimpleFactory.Food;      **public class** FoodFactory {    **public static** Food getFood(String type) **throws** InstantiationException, IllegalAccessException, ClassNotFoundException {  **if**(type.equalsIgnoreCase("mcchicken")) {  **return** McChicken.**class**.newInstance();            } **else if**(type.equalsIgnoreCase("chips")) {  **return** Chips.**class**.newInstance();          } **else** {              System.*out*.println("哎呀！找不到相应的实例化类啦！");  **return null**;          }          }  } |

最后我们建立测试客户端：

|  |
| --- |
| **package** com.diermeng.designPattern.SimpleFactory.client;  **import** com.diermeng.designPattern.SimpleFactory.Food;  **import** com.diermeng.designPattern.SimpleFactory.impl.FoodFactory;    /\*   \* 测试客户端   \*/  **public class** SimpleFactoryTest {  **public static void** main(String[] args) **throws** InstantiationException, IllegalAccessException, ClassNotFoundException {            //实例化各种食物          Food mcChicken = FoodFactory.*getFood*("McChicken");          Food chips = FoodFactory.*getFood*("Chips");          Food eggs = FoodFactory.*getFood*("Eggs");            //获取食物  **if**(mcChicken!=**null**){              mcChicken.get();          }  **if**(chips!=**null**){              chips.get();          }  **if**(eggs!=**null**){              eggs.get();          }          }  } |

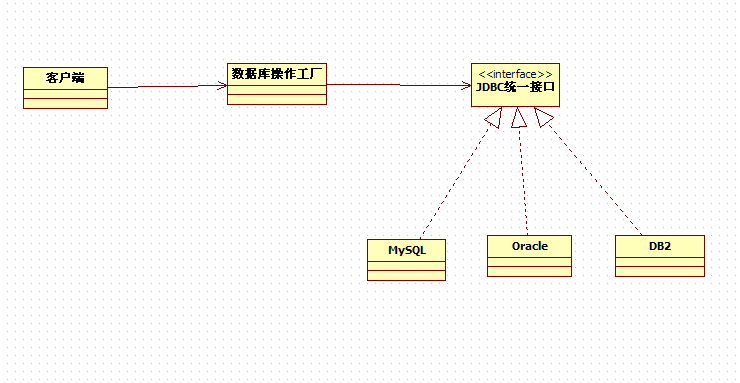
输出的结果如下：

|  |
| --- |
| 哎呀！找不到相应的实例化类啦！  我要一份麦香鸡  我要一份薯条 |

**简单工厂模式的优缺点分析：**

作为一个最基本和最简单的设计模式，简单工厂模式却有很非常广泛的应用，我们这里以**[Java](http://lib.csdn.net/base/17" \t "_blank" \o "Java EE知识库)**中的JDBC操作**[数据库](http://lib.csdn.net/base/14" \t "_blank" \o "MySQL知识库)**为例来说明。

        JDBC是SUN公司提供的一套数据库编程接口API，它利用Java语言提供简单、一致的方式来访问各种关系型数据库。Java程序通过JDBC可以执行SQL语句，对获取的数据进行处理，并将变化了的数据存回数据库，因此，JDBC是Java应用程序与各种关系数据进行对话的一种机制。用JDBC进行数据库访问时，要使用数据库厂商提供的驱动程序接口与数据库管理系统进行数据交互。



客户端要使用使用数据时，只需要和工厂进行交互即可，这就导致操作步骤得到极大的简化，操作步骤按照顺序依次为：注册并加载数据库驱动，一般使用Class.forName();创建与数据库的链接Connection对象；创建SQL语句对象preparedStatement(sql)；提交SQL语句，根据实际情况使用executeQuery()或者executeUpdate()；显示相应的结果；关闭数据库。

      优点：工厂类是整个模式的关键所在。它包含必要的判断逻辑，能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。用户在使用时可以直接根据工厂类去创建所需的实例，而无需了解这些对象是如何创建以及如何组织的。有利于整个软件体系结构的优化。

      缺点：由于工厂类集中了所有实例的创建逻辑，这就直接导致一旦这个工厂出了问题，所有的客户端都会受到牵连；而且由于简单工厂模式的产品室基于一个共同的抽象类或者接口，这样一来，但产品的种类增加的时候，即有不同的产品接口或者抽象类的时候，工厂类就需要判断何时创建何种种类的产品，这就和创建何种种类产品的产品相互混淆在了一起，违背了单一职责，导致系统丧失灵活性和可维护性。而且更重要的是，简单工厂模式违背了“开放封闭原则”，就是违背了“系统对扩展开放，对修改关闭”的原则，因为当我新增加一个产品的时候必须修改工厂类，相应的工厂类就需要重新编译一遍。

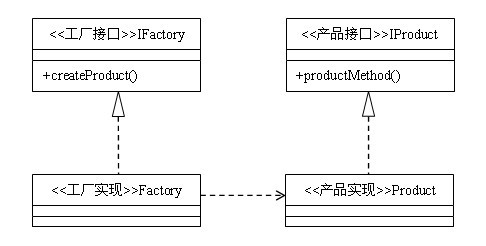
      总结一下：简单工厂模式分离产品的创建者和消费者，有利于软件系统结构的优化；但是由于一切逻辑都集中在一个工厂类中，导致了没有很高的内聚性，同时也违背了“开放封闭原则”。另外，简单工厂模式的方法一般都是静态的，而静态工厂方法是无法让子类继承的，因此，简单工厂模式无法形成基于基类的继承树结构。

**工厂方法模式**

定义一个用于创建对象的接口，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。

四要素：

**类图：**



 要说明工厂模式的优点，可能没有比组装汽车更合适的例子了。场景是这样的：汽车由发动机、轮、底盘组成，现在需要组装一辆车交给调用者。假如不使用工厂模式，代码如下：

**[java]** [view plain](http://blog.csdn.net/zhengzhb/article/details/7348707) [copy](http://blog.csdn.net/zhengzhb/article/details/7348707)

1. **class** Engine {
2. **public** **void** getStyle(){
3. System.out.println("这是汽车的发动机");
4. }
5. }
6. **class** Underpan {
7. **public** **void** getStyle(){
8. System.out.println("这是汽车的底盘");
9. }
10. }
11. **class** Wheel {
12. **public** **void** getStyle(){
13. System.out.println("这是汽车的轮胎");
14. }
15. }
16. **public** **class** Client {
17. **public** **static** **void** main(String[] args) {
18. Engine engine = **new** Engine();
19. Underpan underpan = **new** Underpan();
20. Wheel wheel = **new** Wheel();
21. ICar car = **new** Car(underpan, wheel, engine);
22. car.show();
23. }
24. }

        可以看到，调用者为了组装汽车还需要另外实例化发动机、底盘和轮胎，而这些汽车的组件是与调用者无关的，严重违反了迪米特法则，耦合度太高。并且非常不利于扩展。另外，本例中发动机、底盘和轮胎还是比较具体的，在实际应用中，可能这些产品的组件也都是抽象的，调用者根本不知道怎样组装产品。假如使用工厂方法的话，整个架构就显得清晰了许多。

**[java]** [view plain](http://blog.csdn.net/zhengzhb/article/details/7348707) [copy](http://blog.csdn.net/zhengzhb/article/details/7348707)

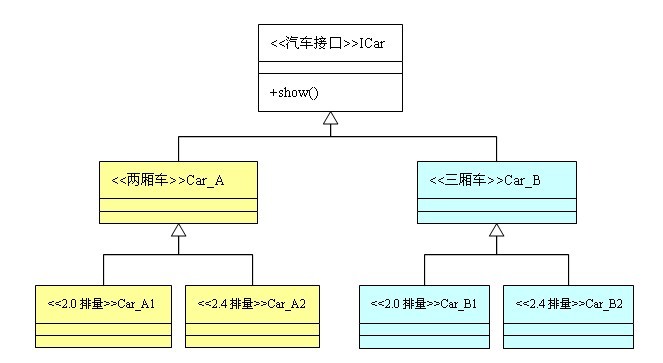
1. **interface** IFactory {
2. **public** ICar createCar();
3. }
4. **class** Factory **implements** IFactory {
5. **public** ICar createCar() {
6. Engine engine = **new** Engine();
7. Underpan underpan = **new** Underpan();
8. Wheel wheel = **new** Wheel();
9. ICar car = **new** Car(underpan, wheel, engine);
10. **return** car;
11. }
12. }
13. **public** **class** Client {
14. **public** **static** **void** main(String[] args) {
15. IFactory factory = **new** Factory();
16. ICar car = factory.createCar();
17. car.show();
18. }
19. }

        使用工厂方法后，调用端的耦合度大大降低了。并且对于工厂来说，是可以扩展的，以后如果想组装其他的汽车，只需要再增加一个工厂类的实现就可以。无论是灵活性还是稳定性都得到了极大的提高。

**抽象工厂模式**

为创建一组相关或相互依赖的对象提供一个接口，而且无需指定他们的具体类。工厂方法模式提供的所有产品都是衍生自同一个接口或抽象类，而抽象工厂模式所提供的产品则是衍生自不同的接口或抽象类

我们依然拿生产汽车的例子来说明他们之间的区别。



在上面的类图中，两厢车和三厢车称为两个不同的等级结构；而2.0排量车和2.4排量车则称为两个不同的产品族。再具体一点，2.0排量两厢车和2.4排量两厢车属于同一个等级结构，2.0排量三厢车和2.4排量三厢车属于另一个等级结构；而2.0排量两厢车和2.0排量三厢车属于同一个产品族，2.4排量两厢车和2.4排量三厢车属于另一个产品族。

        明白了等级结构和产品族的概念，就理解工厂方法模式和抽象工厂模式的区别了，如果工厂的产品全部属于同一个等级结构，则属于工厂方法模式；如果工厂的产品来自多个等级结构，则属于抽象工厂模式。在本例中，如果一个工厂模式提供2.0排量两厢车和2.4排量两厢车，那么他属于工厂方法模式；如果一个工厂模式是提供2.4排量两厢车和2.4排量三厢车两个产品，那么这个工厂模式就是抽象工厂模式，因为他提供的产品是分属两个不同的等级结构。当然，如果一个工厂提供全部四种车型的产品，因为产品分属两个等级结构，他当然也属于抽象工厂模式了。

**抽象工厂模式代码**

**[java]** [view plain](http://blog.csdn.net/zhengzhb/article/details/7359385) [copy](http://blog.csdn.net/zhengzhb/article/details/7359385)

1. **interface** IProduct1 {
2. **public** **void** show();
3. }
4. **interface** IProduct2 {
5. **public** **void** show();
6. }
8. **class** Product1 **implements** IProduct1 {
9. **public** **void** show() {
10. System.out.println("这是1型产品");
11. }
12. }
13. **class** Product2 **implements** IProduct2 {
14. **public** **void** show() {
15. System.out.println("这是2型产品");
16. }
17. }
19. **interface** IFactory {
20. **public** IProduct1 createProduct1();
21. **public** IProduct2 createProduct2();
22. }
23. **class** Factory **implements** IFactory{
24. **public** IProduct1 createProduct1() {
25. **return** **new** Product1();
26. }
27. **public** IProduct2 createProduct2() {
28. **return** **new** Product2();
29. }
30. }
32. **public** **class** Client {
33. **public** **static** **void** main(String[] args){
34. IFactory factory = **new** Factory();
35. factory.createProduct1().show();
36. factory.createProduct2().show();
37. }
38. }

**抽象工厂模式的优点**

        抽象工厂模式除了具有工厂方法模式的优点外，最主要的优点就是可以在类的内部对产品族进行约束。所谓的产品族，一般或多或少的都存在一定的关联，抽象工厂模式就可以在类内部对产品族的关联关系进行定义和描述，而不必专门引入一个新的类来进行管理。

**抽象工厂模式的缺点**

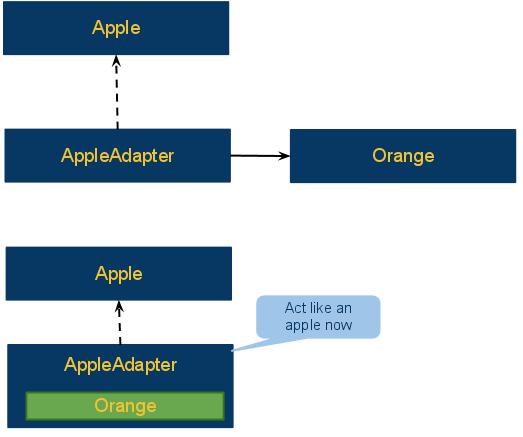
       产品族的扩展将是一件十分费力的事情，假如产品族中需要增加一个新的产品，则几乎所有的工厂类都需要进行修改。所以使用抽象工厂模式时，对产品等级结构的划分是非常重要的。

**适配器模式**

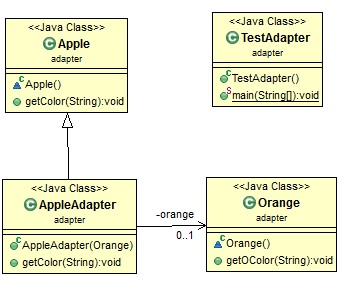
适配器模式在现代的Java框架中十分常用。这种模式适用于以下场景：想使用一个已存在的类，但是该类不符合接口需求；或者需要创建一个可重用的类，适配没有提供合适接口的其它类。

**1、适配器模式的故事**

适配器的思想可以通过下面这个简单的例子说明。这个示例要让一个桔子被“适配”成一个苹果。



从图中下半部分可以看到，适配器包含了一个桔子实例并且继承了苹果类。桔子对象被包装了一个适配器，于是桔子表现得就像苹果一样了。



class Apple {

    public void getAColor(String str) {

        System.out.println("Apple color is: " + str);

    }

}

class Orange {

    public void getOColor(String str) {

        System.out.println("Orange color is: " + str);

    }

}

class AppleAdapter extends Apple {

    private Orange orange;

    public AppleAdapter(Orange orange) {

        this.orange = orange;

    }

    public void getAColor(String str) {

        orange.getOColor(str);

    }

}

public class TestAdapter {

    public static void main(String[] args) {

        Apple apple1 = new Apple();

        Apple apple2 = new Apple();

        apple1.getAColor("green");

        Orange orange = new Orange();

        AppleAdapter aa = new AppleAdapter(orange);

        aa.getAColor("red");

    }

}

实际上，这大概是适配器模式最简单的实现。更多情况下使用的是双向适配器。为了创建一个双向适配器，适配器类需要实现2个接口并包含2个接口的实例。即使如此，

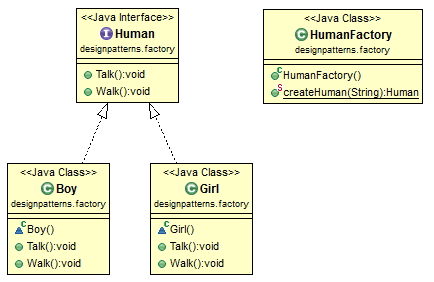
**适配器模式在Java SDK中的运用**

java.io.InputStreamReader(InputStream) (返回一个Reader)  
java.io.OutputStreamWriter(OutputStream) (返回一个Writer)

**1.关于工厂模式**

工厂模式是根据不同的参数创建对象。例如用工厂创建人。 如果我们想要一个男孩，工厂就会为我们生产一个男孩；如果我们需要一个女孩，工厂则会为我们生产一个女孩。工厂会根据不同的参数，为我们提供不同的物品。

**2.工厂模式类图**

[](http://www.importnew.com/6718.html/factory-design-pattern)

**3.工厂模式Java代码**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | interface Human {      public void Talk();      public void Walk();  }    class Boy implements Human{      @Override      public void Talk() {          System.out.println("Boy is talking...");      }        @Override      public void Walk() {          System.out.println("Boy is walking...");      }  }    class Girl implements Human{        @Override      public void Talk() {          System.out.println("Girl is talking...");      }        @Override      public void Walk() {          System.out.println("Girl is walking...");      }  }    public class HumanFactory {      public static Human createHuman(String m){          Human p = null;          if(m == "boy"){              p = new Boy();          }else if(m == "girl"){              p = new Girl();          }            return p;      }  } |

**4.工厂模式在Java标准库中的应用**

根据不同的参数，getInstance()方法会返回不同的Calendar（日历）对象。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | java.util.Calendar – getInstance()  java.util.Calendar – getInstance(TimeZone zone)  java.util.Calendar – getInstance(Locale aLocale)  java.util.Calendar – getInstance(TimeZone zone, Locale aLocale)    java.text.NumberFormat – getInstance()  java.text.NumberFormat – getInstance(Locale inLocale) |

**单例模式**

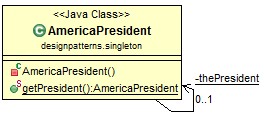
单例模式是Java中最常用的模式之一，它通过阻止外部实例化和修改，来控制所创建的对象的数量。这个概念可以被推广到仅有一个对象能更高效运行的系统，或者限制对象实例化为特定的数目的系统中。例如：

1. 私有构造函数 - 其他类不能实例化一个新的对象。
2. 私有化引用 - 不能进行外部修改。
3. 公有静态方法是唯一可以获得对象的方式。

**单例模式的故事**

来看一下使用情况：一个国家只能有一位总统（可能在正常情况下）。所以不管任何时候我们需要一位总统，使用AmericaPresident就能返回一个。getPresident()方法将确保只有一个总统对象被创建。否者，就不妙了。

**类图**



**单例模式Java示例代码**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | package com.programcreek.designpatterns.singleton;    public class AmericaPresident {      private AmericaPresident() {    }        private static final AmericaPresident thePresident=new AmericaPresident();        public static AmericaPresident getPresident(){          return thePresident;      }  } |

**单例模式在 Java 标准库中的使用**

java.lang.Runtime#getRuntime() 是Java 标准库中常用的方法，它返回与当前Java应用关联的运行时对象。

下面是getRunTime() 的一个简单例子，它在windows系统上读取一个网页。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Process p = Runtime.getRuntime().exec(          "C:/windows/system32/ping.exe programcreek.com");  //get process input stream and put it to buffered reader  BufferedReader input = new BufferedReader(new InputStreamReader(          p.getInputStream()));    String line;  while ((line = input.readLine()) != null) {      System.out.println(line);  }    input.close(); |

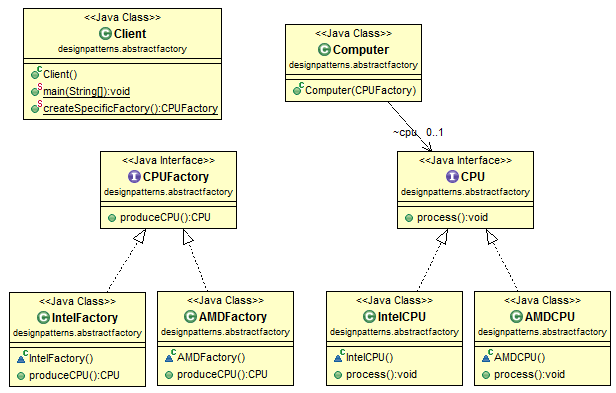
**输出结果**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Pinging programcreek.com [198.71.49.96] with 32 bytes of data:  Reply from 198.71.49.96: bytes=32 time=53ms TTL=47  Reply from 198.71.49.96: bytes=32 time=53ms TTL=47  Reply from 198.71.49.96: bytes=32 time=52ms TTL=47  Reply from 198.71.49.96: bytes=32 time=53ms TTL=47  Ping statistics for 198.71.49.96:  Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),  Approximate round trip times in milli-seconds:  Minimum = 52ms, Maximum = 53ms, Average = 52ms |

**抽象工厂模式**

抽象工厂模式是在工厂模式的基础上增加的一层抽象概念。如果比较抽象工厂模式和工厂模式，我们不难发现前者只是增加了一层抽象的概念。抽象工厂是一个父类工厂，可以创建其它工厂类。故我们也叫它“工厂的工厂”。

**1、抽象工厂类图**

[](http://www.importnew.com/6914.html/abstract-factory-design-pattern)

**2、抽象工厂Java示例代码**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | interface CPU {      void process();  }    interface CPUFactory {      CPU produceCPU();  }    class AMDFactory implements CPUFactory {      public CPU produceCPU() {          return new AMDCPU();      }  }    class IntelFactory implements CPUFactory {      public CPU produceCPU() {          return new IntelCPU();      }  }    class AMDCPU implements CPU {      public void process() {          System.out.println("AMD is processing...");      }  }    class IntelCPU implements CPU {      public void process() {          System.out.println("Intel is processing...");      }  }    class Computer {      CPU cpu;        public Computer(CPUFactory factory) {          cpu = factory.produceCPU();          cpu.process();      }  }    public class Client {      public static void main(String[] args) {          new Computer(createSpecificFactory());      }        public static CPUFactory createSpecificFactory() {          int sys = 0; // 基于特定要求          if (sys == 0)              return new AMDFactory();          else              return new IntelFactory();      }  } |

**3、实例**

在当今的架构中，抽象工厂是一个非常重要的概念。

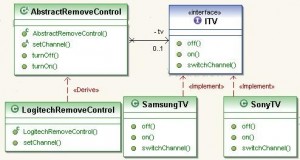
**桥接模式**

简单来讲，桥接模式是一个两层的抽象。

*桥接模式是用于“把抽象和实现分开，这样它们就能独立变化”。 桥接模式使用了封装、聚合，可以用继承将不同的功能拆分为不同的类。*

**1、桥接模式的故事**

电视和遥控器（图中有错字）是一个完美展示两层抽象的例子。你有一个电视机的接口，还有一个遥控器的抽象类。我们都知道，将它们中任何一个定义为一个具体类都不是好办法，因为其它厂家会有不同的实现方法。



**2、桥接模式Java示例代码**

首先定义电视机的接口：ITV

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface ITV {      public void on();      public void off();      public void switchChannel(int channel);  } |

实现三星的 ITV 接口。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class SamsungTV implements ITV {      @Override      public void on() {          System.out.println("Samsung is turned on.");      }        @Override      public void off() {          System.out.println("Samsung is turned off.");      }        @Override      public void switchChannel(int channel) {          System.out.println("Samsung: channel - " + channel);      }  } |

再实现索尼的ITV接口。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class SonyTV implements ITV {        @Override      public void on() {          System.out.println("Sony is turned on.");      }        @Override      public void off() {          System.out.println("Sony is turned off.");      }        @Override      public void switchChannel(int channel) {          System.out.println("Sony: channel - " + channel);      }  } |

遥控器要包含对TV的引用。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public abstract class AbstractRemoteControl {      /\*\*       \* @uml.property  name="tv"       \* @uml.associationEnd       \*/      private ITV tv;        public AbstractRemoteControl(ITV tv){          this.tv = tv;      }        public void turnOn(){          tv.on();      }        public void turnOff(){          tv.off();      }        public void setChannel(int channel){          tv.switchChannel(channel);      }  } |

定义遥控器的具体类。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class LogitechRemoteControl extends AbstractRemoteControl {        public LogitechRemoteControl(ITV tv) {          super(tv);      }        public void setChannelKeyboard(int channel){          setChannel(channel);          System.out.println("Logitech use keyword to set channel.");      }  } |
| 1  2  3  4  5  6  7 | public class Main {      public static void main(String[] args){          ITV tv = new SonyTV();          LogitechRemoteControl lrc = new LogitechRemoteControl(tv);          lrc.setChannelKeyboard(100);      }  } |

输出如下:

|  |  |
| --- | --- |
| 1  2 | Sony: channel – 100  Logitech use keyword to set channel. |

总结一下, 桥接模式允许两层实现的抽象，上面的电视机和遥控器就是很好的例子。可见，桥接模式提供了更多的灵活性。

**3、[Eclipse](http://res.importnew.com/eclipse" \t "_blank" \o "Eclipse ImportNew主页) 平台上的桥接模式**

在Eclipse 架构使用的模式中，[桥接模式](http://www.programcreek.com/2013/02/eclipse-design-patterns-proxy-and-bridge-in-workspace/" \t "_blank)占有重要的地位。

责任链模式

责任链模式，有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，直到某一对象决定处理该请求。但是发出者并不清楚到底最终那个对象会处理该请求，所以，责任链模式可以实现，在隐瞒客户端的情况下，对系统进行动态的调整。

|  |  |
| --- | --- |
| 1  2  3  4  5 | package com.model.behaviour;    public interface Handler {      public void operator();  } |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.model.behaviour;    public abstract class AbstractHandler {        private Handler handler;        public Handler getHandler() {          return handler;      }        public void setHandler(Handler handler) {          this.handler = handler;      }    } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | package com.model.behaviour;    public class MyHandler extends AbstractHandler implements Handler {        private String name;        public MyHandler(String name) {          this.name = name;      }        @Override      public void operator() {          System.out.println(name + "deal!");          if (getHandler() != null) {              getHandler().operator();          }      }  } |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.model.behaviour;    public class Test {        public static void main(String[] args) {          MyHandler h1 = new MyHandler("h1");          MyHandler h2 = new MyHandler("h2");          MyHandler h3 = new MyHandler("h3");            h1.setHandler(h2);          h2.setHandler(h3);            h1.operator();      }  } |

运行结果：

|  |  |
| --- | --- |
| 1  2  3 | h1deal!  h2deal!  h3deal! |

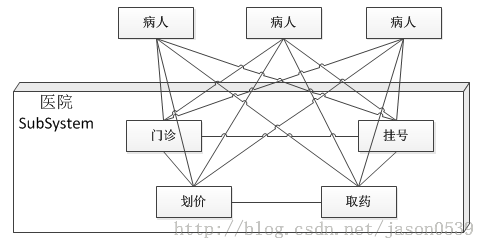
此处强调一点就是，链接上的请求可以是一条链，可以是一个树，还可以是一个环，模式本身不约束这个，需要我们自己去实现，同时，在一个时刻，命令只允许由一个对象传给另一个对象，而不允许传给多个对象。

**门面模式**

# 医院的例子

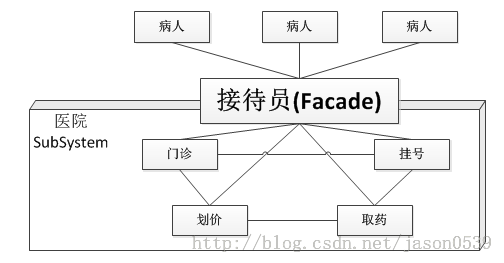
　　现代的软件系统都是比较复杂的，设计师处理复杂系统的一个常见方法便是将其“分而治之”，把一个系统划分为几个较小的子系统。如果把医院作为一个子系统，按照部门职能，这个系统可以划分为挂号、门诊、划价、化验、收费、取药等。看病的病人要与这些部门打交道，就如同一个子系统的客户端与一个子系统的各个类打交道一样，不是一件容易的事情。

　　首先病人必须先挂号，然后门诊。如果医生要求化验，病人必须首先划价，然后缴费，才可以到化验部门做化验。化验后再回到门诊室。



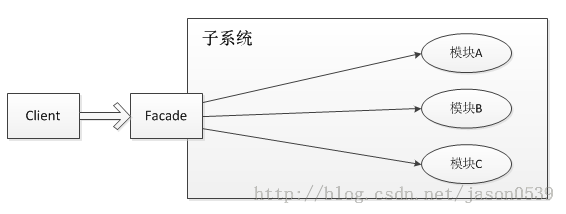
上图描述的是病人在医院里的体验，图中的方框代表医院。

　　解决这种不便的方法便是引进门面模式，医院可以设置一个接待员的位置，由接待员负责代为挂号、划价、缴费、取药等。这个接待员就是门面模式的体现，病人只接触接待员，由接待员与各个部门打交道。

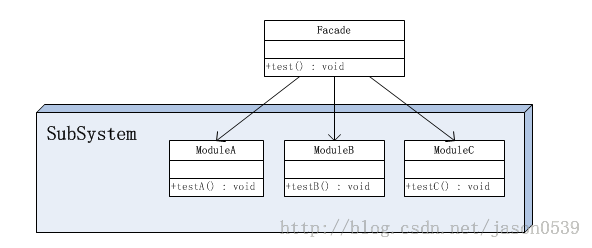


# 门面模式的结构

　　门面模式没有一个一般化的类图描述，最好的描述方法实际上就是以一个例子说明。



　由于门面模式的结构图过于抽象，因此把它稍稍具体点。假设子系统内有三个模块，分别是ModuleA、ModuleB和ModuleC，它们分别有一个示例方法，那么此时示例的整体结构图如下：



在这个对象图中，出现了两个角色：

　　●　　**门面(Facade)角色 ：**客户端可以调用这个角色的方法。此角色知晓相关的（一个或者多个）子系统的功能和责任。在正常情况下，本角色会将所有从客户端发来的请求委派到相应的子系统去。

　　●　　**子系统(SubSystem)角色 ：**可以同时有一个或者多个子系统。每个子系统都不是一个单独的类，而是一个类的集合（如上面的子系统就是由ModuleA、ModuleB、ModuleC三个类组合而成）。每个子系统都可以被客户端直接调用，或者被门面角色调用。子系统并不知道门面的存在，对于子系统而言，门面仅仅是另外一个客户端而已。

### 源代码

　　子系统角色中的类：

**[java]** [view plain](http://blog.csdn.net/jason0539/article/details/22775311) [copy](http://blog.csdn.net/jason0539/article/details/22775311)

[print?](http://blog.csdn.net/jason0539/article/details/22775311)[在CODE上查看代码片](https://code.csdn.net/snippets/269823)

1. **public** **class** ModuleA {
2. //示意方法
3. **public** **void** testA(){
4. System.out.println("调用ModuleA中的testA方法");
5. }
6. }

**[java]** [view plain](http://blog.csdn.net/jason0539/article/details/22775311) [copy](http://blog.csdn.net/jason0539/article/details/22775311)

[print?](http://blog.csdn.net/jason0539/article/details/22775311)[在CODE上查看代码片](https://code.csdn.net/snippets/269823)

1. **public** **class** ModuleB {
2. //示意方法
3. **public** **void** testB(){
4. System.out.println("调用ModuleB中的testB方法");
5. }
6. }

**[java]** [view plain](http://blog.csdn.net/jason0539/article/details/22775311) [copy](http://blog.csdn.net/jason0539/article/details/22775311)

[print?](http://blog.csdn.net/jason0539/article/details/22775311)[在CODE上查看代码片](https://code.csdn.net/snippets/269823)

1. **public** **class** ModuleC {
2. //示意方法
3. **public** **void** testC(){
4. System.out.println("调用ModuleC中的testC方法");
5. }
6. }

　门面角色类：

**[java]** [view plain](http://blog.csdn.net/jason0539/article/details/22775311) [copy](http://blog.csdn.net/jason0539/article/details/22775311)

[print?](http://blog.csdn.net/jason0539/article/details/22775311)[在CODE上查看代码片](https://code.csdn.net/snippets/269823)

1. **public** **class** Facade {
2. //示意方法，满足客户端需要的功能
3. **public** **void** test(){
4. ModuleA a = **new** ModuleA();
5. a.testA();
6. ModuleB b = **new** ModuleB();
7. b.testB();
8. ModuleC c = **new** ModuleC();
9. c.testC();
10. }
11. }

客户端角色类：

**[java]** [view plain](http://blog.csdn.net/jason0539/article/details/22775311) [copy](http://blog.csdn.net/jason0539/article/details/22775311)

[print?](http://blog.csdn.net/jason0539/article/details/22775311)[在CODE上查看代码片](https://code.csdn.net/snippets/269823)

1. **public** **class** Client {
3. **public** **static** **void** main(String[] args) {
5. Facade facade = **new** Facade();
6. facade.test();
7. }
9. }

　Facade类其实相当于A、B、C模块的外观界面，有了这个Facade类，那么客户端就不需要亲自调用子系统中的A、B、C模块了，也不需要知道系统内部的实现细节，甚至都不需要知道A、B、C模块的存在，客户端只需要跟Facade类交互就好了，从而更好地实现了客户端和子系统中A、B、C模块的解耦，让客户端更容易地使用系统。

# 门面模式的实现

　　使用门面模式还有一个附带的好处，就是能够有选择性地暴露方法。一个模块中定义的方法可以分成两部分，一部分是给子系统外部使用的，一部分是子系统内部模块之间相互调用时使用的。有了Facade类，那么用于子系统内部模块之间相互调用的方法就不用暴露给子系统外部了。

　　比如，定义如下A、B、C模块。

**[java]** [view plain](http://blog.csdn.net/jason0539/article/details/22775311) [copy](http://blog.csdn.net/jason0539/article/details/22775311)

[print?](http://blog.csdn.net/jason0539/article/details/22775311)[在CODE上查看代码片](https://code.csdn.net/snippets/269823)

1. **public** **class** Module {
2. /\*\*
3. \* 提供给子系统外部使用的方法
4. \*/
5. **public** **void** a1(){};
7. /\*\*
8. \* 子系统内部模块之间相互调用时使用的方法
9. \*/
10. **private** **void** a2(){};
11. **private** **void** a3(){};
12. }

**[java]** [view plain](http://blog.csdn.net/jason0539/article/details/22775311) [copy](http://blog.csdn.net/jason0539/article/details/22775311)

[print?](http://blog.csdn.net/jason0539/article/details/22775311)[在CODE上查看代码片](https://code.csdn.net/snippets/269823)

1. **public** **class** ModuleB {
2. /\*\*
3. \* 提供给子系统外部使用的方法
4. \*/
5. **public** **void** b1(){};
7. /\*\*
8. \* 子系统内部模块之间相互调用时使用的方法
9. \*/
10. **private** **void** b2(){};
11. **private** **void** b3(){};
12. }

**[java]** [view plain](http://blog.csdn.net/jason0539/article/details/22775311) [copy](http://blog.csdn.net/jason0539/article/details/22775311)

[print?](http://blog.csdn.net/jason0539/article/details/22775311)[在CODE上查看代码片](https://code.csdn.net/snippets/269823)

1. **public** **class** ModuleC {
2. /\*\*
3. \* 提供给子系统外部使用的方法
4. \*/
5. **public** **void** c1(){};
7. /\*\*
8. \* 子系统内部模块之间相互调用时使用的方法
9. \*/
10. **private** **void** c2(){};
11. **private** **void** c3(){};
12. }

**[java]** [view plain](http://blog.csdn.net/jason0539/article/details/22775311) [copy](http://blog.csdn.net/jason0539/article/details/22775311)

[print?](http://blog.csdn.net/jason0539/article/details/22775311)[在CODE上查看代码片](https://code.csdn.net/snippets/269823)

1. **public** **class** ModuleFacade {
3. ModuleA a = **new** ModuleA();
4. ModuleB b = **new** ModuleB();
5. ModuleC c = **new** ModuleC();
6. /\*\*
7. \* 下面这些是A、B、C模块对子系统外部提供的方法
8. \*/
9. **public** **void** a1(){
10. a.a1();
11. }
12. **public** **void** b1(){
13. b.b1();
14. }
15. **public** **void** c1(){
16. c.c1();
17. }
18. }

这样定义一个ModuleFacade类可以有效地屏蔽内部的细节，免得客户端去调用Module类时，发现一些不需要它知道的方法。比如a2()和a3()方法就不需要让客户端知道，否则既暴露了内部的细节，又让客户端迷惑。对客户端来说，他可能还要去思考a2()、a3()方法用来干什么呢？其实a2()和a3()方法是内部模块之间交互的，原本就不是对子系统外部的，所以干脆就不要让客户端知道。

### 一个系统可以有几个门面类

　　在门面模式中，通常只需要一个门面类，并且此门面类只有一个实例，换言之它是一个单例类。当然这并不意味着在整个系统里只有一个门面类，而仅仅是说对每一个子系统只有一个门面类。或者说，如果一个系统有好几个子系统的话，每一个子系统都有一个门面类，整个系统可以有数个门面类。

### 为子系统增加新行为

　　初学者往往以为通过继承一个门面类便可在子系统中加入新的行为，这是错误的。门面模式的用意是为子系统提供一个集中化和简化的沟通管道，而不能向子系统加入新的行为。比如医院中的接待员并不是医护人员，接待员并不能为病人提供医疗服务。

# 门面模式的优点

　　门面模式的优点：

**●　　松散耦合**

门面模式松散了客户端与子系统的耦合关系，让子系统内部的模块能更容易扩展和维护。

**●　　简单易用**

门面模式让子系统更加易用，客户端不再需要了解子系统内部的实现，也不需要跟众多子系统内部的模块进行交互，只需要跟门面类交互就可以了。

**●　　更好的划分访问层次**

通过合理使用Facade，可以帮助我们更好地划分访问的层次。有些方法是对系统外的，有些方法是系统内部使用的。把需要暴露给外部的功能集中到门面中，这样既方便客户端使用，也很好地隐藏了内部的细节。