# 设计模式说明文档

## 设计模式须知

### 产生背景

“设计模式”这个术语最初并不是出现在软件设计中，而是被用于建筑领域的设计中。

1977 年，美国著名建筑大师、加利福尼亚大学伯克利分校环境结构中心主任克里斯托夫·亚历山大（Christopher Alexander）在他的著作《建筑模式语言：城镇、建筑、构造（A Pattern Language: Towns Building Construction）中描述了一些常见的建筑设计问题，并提出了 253 种关于对城镇、邻里、住宅、花园和房间等进行设计的基本模式。

1979 年他的另一部经典著作《建筑的永恒之道》（The Timeless Way of Building）进一步强化了设计模式的思想，为后来的建筑设计指明了方向。

1987 年，肯特·贝克（Kent Beck）和沃德·坎宁安（Ward Cunningham）首先将克里斯托夫·亚历山大的模式思想应用在 Smalltalk 中的图形用户接口的生成中，但没有引起软件界的关注。

直到 1990 年，软件工程界才开始研讨设计模式的话题，后来召开了多次关于设计模式的研讨会。

1995 年，艾瑞克·伽马（ErichGamma）、理査德·海尔姆（Richard Helm）、拉尔夫·约翰森（Ralph Johnson）、约翰·威利斯迪斯（John Vlissides）等 4 位作者合作出版了《设计模式：可复用面向对象软件的基础》（Design Patterns: Elements of Reusable Object-Oriented Software）一书，在本教程中收录了 23 个设计模式，这是设计模式领域里程碑的事件，导致了软件设计模式的突破。这 4 位作者在软件开发领域里也以他们的“四人组”（Gang of Four，GoF）匿名著称。

直到今天，狭义的设计模式还是本教程中所介绍的 23 种经典设计模式。

### 概念与意义

有关软件设计模式的定义很多，有些从模式的特点来说明，有些从模式的作用来说明。本文给出的定义是大多数学者公认的，从以下两个方面来说明：

#### 1. 软件设计模式的概念

软件设计模式（Software Design Pattern），又称设计模式，是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。

它描述了在软件设计过程中的一些不断重复发生的问题，以及该问题的解决方案。

也就是说，它是解决特定问题的一系列套路，是前辈们的代码设计经验的总结，具有一定的普遍性，可以反复使用。

其目的是为了提高代码的可重用性、代码的可读性和代码的可靠性。

#### 2. 学习设计模式的意义

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

1. 可以提高程序员的思维能力、编程能力和设计能力
2. 使程序设计更加标准化、代码编制更加工程化，使软件开发效率大大提高，从而缩短软件的开发周期
3. 使设计的代码可重用性高、可读性强、可靠性高、灵活性好、可维护性强

当然，软件设计模式只是一个引导。在具体的软件幵发中，必须根据设计的应用系统的特点和要求来恰当选择。

对于简单的程序开发，可能写一个简单的算法要比引入某种设计模式更加容易。

但对大项目的开发或者框架设计，用设计模式来组织代码显然更好。

### 基本要素

软件设计模式使人们可以更加简单方便地复用成功的设计和体系结构，它通常包含以下几个基本要素：模式名称、别名、动机、问题、解决方案、效果、结构、模式角色、合作关系、实现方法、适用性、已知应用、例程、模式扩展和相关模式等，其中最关键的元素包括以下4个主要部分：

#### 1.模式名称

每一个模式都有自己的名字，通常用一两个词来描述，可以根据模式的问题、特点、解决方案、功能和效果来命名。

模式名称（PatternName）有助于我们理解和记忆该模式，也方便我们来讨论自己的设计。

#### 2.问题

问题（Problem）描述了该模式的应用环境，即何时使用该模式。它解释了设计问题和问题存在的前因后果，以及必须满足的一系

列先决条件。

#### 3.解决方案

模式问题的解决方案（Solution）包括设计的组成成分、它们之间的相互关系及各自的职责和协作方式。

因为模式就像一个模板，可应用于多种不同场合，所以解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合（类或对象的组合）来解决这个问题。

#### 4.效果

描述了模式的应用效果以及使用该模式应该权衡的问题，即模式的优缺点。

主要是对时间和空间的衡量，以及该模式对系统的灵活性、扩充性、可移植性的影响，也考虑其实现问题。

显式地列出这些效果（Consequence）对理解和评价这些模式有很大的帮助。

### 分类及功能

设计模式有两种分类方法，即根据模式的目的来分和根据模式的作用的范围来分。

#### 1.根据目的来分

根据模式是用来完成什么工作来划分，这种方式可分为创建型模式、结构型模式和行为型模式 3 种。

##### 创建型模式

用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”。

GoF 中提供了单例、原型、工厂方法、抽象工厂、建造者等 5 种创建型模式。

##### 结构型模式

用于描述如何将类或对象按某种布局组成更大的结构。

GoF 中提供了代理、适配器、桥接、装饰、外观、享元、组合等 7 种结构型模式。

##### 行为型模式

用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责。

GoF 中提供了模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等 11 种行为型模式。

#### 2.根据作用范围来分

根据模式是主要用于类上还是主要用于对象上来分，这种方式可分为类模式和对象模式两种。

##### 类模式

用于处理类与子类之间的关系，这些关系通过继承来建立，是静态的，在编译时刻便确定下来了。

GoF中的工厂方法、（类）适配器、模板方法、解释器属于该模式。

##### 对象模式

用于处理对象之间的关系，这些关系可以通过组合或聚合来实现，在运行时刻是可以变化的，更具动态性。

GoF 中除了以上 4 种，其他的都是对象模式。

#### 3. GoF的23种设计模式的功能

前面说明了 GoF 的 23 种设计模式的分类，现在对各个模式的功能进行介绍。

##### 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（Factory Method）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。

##### 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。

##### 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。

##### 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，从而降低了抽象和实现这两个可变维度的耦合度。

##### 装饰（Decorator）模式

动态的给对象增加一些职责，即增加其额外的功能。

##### 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。

##### 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。

##### 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。

##### 模板方法（TemplateMethod）模式

定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类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。

##### 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。

##### 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。

##### 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。

##### 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。

##### 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。

##### 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。

##### 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。

##### 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。

##### 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。

##### 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。

#### 特别说明

以上23 种设计模式并不是孤立存在的，很多模式之间存在一定的关联关系，在大的系统开发中常常同时使用多种设计模式。

### 设计原则

在软件开发中，为了提高软件系统的可维护性和可复用性，增加软件的可扩展性和灵活性，程序员要尽量根据7条原则来开发程序，从而提高软件开发效率、节约软件开发成本和维护成本。

我们将在下面依次来介绍这7条原则（开闭原则、里氏替换原则、依赖倒置原则、单一职责原则、接口隔离原则、迪米特原则、合成复用原则）。

#### 开闭原则

##### 定义

开闭原则（Open Closed Principle，OCP）由勃兰特·梅耶（Bertrand Meyer）提出，他在 1988 年的著作《面向对象软件构造》（Object Oriented Software Construction）中提出：软件实体应当对扩展开放，对修改关闭（Software entities should be open for extension，but closed for modification），这就是开闭原则的经典定义。

这里的软件实体包括以下几个部分：

1.项目中划分出的模块

2.类与接口

3.方法

开闭原则的含义：当应用的需求改变时，在不修改软件实体的源代码或者二进制代码的前提下，可以扩展模块的功能，使其满足新的需求。

##### 作用

开闭原则是面向对象程序设计的终极目标，它使软件实体拥有一定的适应性和灵活性的同时具备稳定性和延续性。具体来说，其作用如下：

###### 1.对软件测试的影响

软件遵守开闭原则的话，软件测试时只需要对扩展的代码进行测试就可以了，因为原有的测试代码仍然能够正常运行。

###### 2.可以提高代码的可复用性

粒度越小，被复用的可能性就越大；在面向对象的程序设计中，根据原子和抽象编程可以提高代码的可复用性。

###### 3.可以提高软件的可维护性

遵守开闭原则的软件，其稳定性高和延续性强，从而易于扩展和维护。

##### 实现方法

可以通过“抽象约束、封装变化”来实现开闭原则，即通过接口或者抽象类为软件实体定义一个相对稳定的抽象层，而将相同的可变因素封装在相同的具体实现类中。

因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。而软件中易变的细节可以从抽象派生来的实现类来进行扩展，当软件需要发生变化时，只需要根据需求重新派生一个实现类来扩展就可以了。

##### 说明示例

下面以 Windows 的桌面主题为例介绍开闭原则的应用。

分析：Windows 的主题是桌面背景图片、窗口颜色和声音等元素的组合。用户可以根据自己的喜爱更换自己的桌面主题，也可以从网上下载新的主题。

这些主题有共同的特点，可以为其定义一个抽象类（Abstract Subject），而每个具体的主题（Specific Subject）是其子类。

用户窗体可以根据需要选择或者增加新的主题，而不需要修改原代码，所以它是满足开闭原则的，其类图如下：



#### 里氏替换原则

##### 定义

里氏替换原则（Liskov Substitution Principle，LSP）由麻省理工学院计算机科学实验室的里斯科夫（Liskov）女士在 1987 年的“面向对象技术的高峰会议”（OOPSLA）上发表的一篇文章《数据抽象和层次》（Data Abstraction and Hierarchy）里提出来的，她提出：继承必须确保超类所拥有的性质在子类中仍然成立（Inheritance should ensure that any property proved about supertype objects also holds for subtype objects）。

里氏替换原则主要阐述了有关继承的一些原则，也就是什么时候应该使用继承，什么时候不应该使用继承，以及其中蕴含的原理。

里氏替换原是继承复用的基础，它反映了基类与子类之间的关系，是对开闭原则的补充，是对实现抽象化的具体步骤的规范。

##### 作用

里氏替换原则的主要作用如下:

1.里氏替换原则是实现开闭原则的重要方式之一。

2.它克服了继承中重写父类造成的可复用性变差的缺点。

3.它是动作正确性的保证。即类的扩展不会给已有的系统引入新的错误，降低了代码出错的可能性。

##### 实现方法

里氏替换原则通俗来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。

也就是说：子类继承父类时，除添加新的方法完成新增功能外，尽量不要重写父类的方法。

如果通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的概率会非常大。

如果程序违背了里氏替换原则，则继承类的对象在基类出现的地方会出现运行错误。这时其修正方法是：取消原来的继承关系，重新设计它们之间的关系。

关于里氏替换原则的例子，最有名的是“正方形不是长方形”。当然，生活中也有很多类似的例子，例如，企鹅、鸵鸟和几维鸟从生物学的角度来划分，它们属于鸟类；但从类的继承关系来看，由于它们不能继承“鸟”会飞的功能，所以它们不能定义成“鸟”的子类。同样，由于“气球鱼”不会游泳，所以不能定义成“鱼”的子类；“玩具炮”炸不了敌人，所以不能定义成“炮”的子类等。

##### 说明示例

下面以“几维鸟不是鸟”为例来说明里氏替换原则。

分析：鸟一般都会飞行，如燕子的飞行速度大概是每小时 120 千米。但是新西兰的几维鸟由于翅膀退化无法飞行。假如要设计一个实例，计算这两种鸟飞行 300 千米要花费的时间。显然，拿燕子来测试这段代码，结果正确，能计算出所需要的时间；但拿几维鸟来测试，结果会发生“除零异常”或是“无穷大”，明显不符合预期，其类图如下图所示：



程序代码如下：

package principle;

public class LSPtest

{

public static void main(String[] args)

{

Bird bird1=new Swallow();

Bird bird2=new BrownKiwi();

bird1.setSpeed(120);

bird2.setSpeed(120);

System.out.println("如果飞行300公里：");

try

{

System.out.println("燕子将飞行"+bird1.getFlyTime(300)+"小时.");

System.out.println("几维鸟将飞行"+bird2.getFlyTime(300)+"小时。");

}

catch(Exception err)

{

System.out.println("发生错误了!");

}

}

}

//鸟类

class Bird

{

double flySpeed;

public void setSpeed(double speed)

{

flySpeed=speed;

}

public double getFlyTime(double distance)

{

return(distance/flySpeed);

}

}

//燕子类

class Swallow extends Bird{}

//几维鸟类

class BrownKiwi extends Bird

{

public void setSpeed(double speed)

{

flySpeed=0;

}

}

程序的运行结果如下：

如果飞行300公里：

燕子将飞行2.5小时.

几维鸟将飞行Infinity小时。

程序运行错误的原因：几维鸟类重写了鸟类的 setSpeed(double speed) 方法，这违背了里氏替换原则。正确的做法是：取消几维鸟原来的继承关系，定义鸟和几维鸟的更一般的父类，如动物类，它们都有奔跑的能力。几维鸟的飞行速度虽然为 0，但奔跑速度不为 0，可以计算出其奔跑 300 千米所要花费的时间。其类图如下图所示：



#### 依赖倒置原则

##### 定义

依赖倒置原则（Dependence Inversion Principle，DIP）是 Object Mentor 公司总裁罗伯特·马丁（Robert C.Martin）于 1996 年在 [C++](http://c.biancheng.net/cplus/) Report 上发表的文章。

依赖倒置原则的原始定义：高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象（High level modules shouldnot depend upon low level modules.Both should depend upon abstractions.Abstractions should not depend upon details. Details should depend upon abstractions）。其核心思想是：要面向接口编程，不要面向实现编程。

依赖倒置原则是实现开闭原则的重要途径之一，它降低了客户与实现模块之间的耦合。

由于在软件设计中，细节具有多变性，而抽象层则相对稳定，因此以抽象为基础搭建起来的架构要比以细节为基础搭建起来的架构要稳定得多。这里的抽象指的是接口或者抽象类，而细节是指具体的实现类。

使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给它们的实现类去完成。

##### 作用

依赖倒置原则的主要作用如下：

1.依赖倒置原则可以降低类间的耦合性

2.依赖倒置原则可以提高系统的稳定性

3.依赖倒置原则可以减少并行开发引起的风险

4.依赖倒置原则可以提高代码的可读性和可维护性。

##### 实现方法

依赖倒置原则的目的是通过要面向接口的编程来降低类间的耦合性，所以我们在实际编程中只要遵循以下4点，就能在项目中满足这个规则。

1.每个类尽量提供接口或抽象类，或者两者都具备

2.变量的声明类型尽量是接口或者是抽象类

3.任何类都不应该从具体类派生

4.使用继承时尽量遵循里氏替换原则。

##### 说明示例

下面以“顾客购物程序”为例来说明依赖倒置原则的应用。

分析：本程序反映了 “顾客类”与“商店类”的关系。商店类中有 sell() 方法，顾客类通过该方法购物以下代码定义了顾客类通过韶关网店 ShaoguanShop 购物：

class Customer

{

public void shopping(ShaoguanShop shop)

{

//购物

System.out.println（shop.sell());

}

}

但是，这种设计存在缺点，如果该顾客想从另外一家商店（如婺源网店 WuyuanShop）购物，就要将该顾客的代码修改如下：

class Customer

{

public void shopping(WuyuanShop shop)

{

//购物

System.out.println(shop.sell());

}

}

顾客每更换一家商店，都要修改一次代码，这明显违背了开闭原则。存在以上缺点的原因是：顾客类设计时同具体的商店类绑定了，这违背了依赖倒置原则。解决方法是：定义“婺源网店”和“韶关网店”的共同接口 Shop，顾客类面向该接口编程，其代码修改如下：

class Customer

{

public void shopping(Shop shop)

{

//购物

System.out.println(shop.sell());

}

}

这样，不管顾客类 Customer 访问什么商店，或者增加新的商店，都不需要修改原有代码了，其类图如下图所示：



程序代码如下：

package principle;

public class DIPtest

{

public static void main(String[] args)

{

Customer wang=new Customer();

System.out.println("顾客购买以下商品：");

wang.shopping(new ShaoguanShop());

wang.shopping(new WuyuanShop());

}

}

//商店

interface Shop

{

public String sell(); //卖

}

//韶关网店

class ShaoguanShop implements Shop

{

public String sell()

{

return "韶关土特产：香菇、木耳……";

}

}

//婺源网店

class WuyuanShop implements Shop

{

public String sell()

{

return "婺源土特产：绿茶、酒糟鱼……";

}

}

//顾客

class Customer

{

public void shopping(Shop shop)

{

//购物

System.out.println(shop.sell());

}

}

程序的运行结果如下：

顾客购买以下商品：

韶关土特产：香菇、木耳……

婺源土特产：绿茶、酒糟鱼……

#### 单一职责原则

##### 定义

单一职责原则（Single Responsibility Principle，SRP）又称单一功能原则，由罗伯特·C.马丁（Robert C. Martin）于《敏捷软件开发：原则、模式和实践》一书中提出的。

这里的职责是指类变化的原因，单一职责原则规定一个类应该有且仅有一个引起它变化的原因，否则类应该被拆分（There should never be more than one reason for a class to change）。

该原则提出对象不应该承担太多职责，如果一个对象承担了太多的职责，至少存在以下两个缺点：

1.一个职责的变化可能会削弱或者抑制这个类实现其他职责的能力；

2.当客户端需要该对象的某一个职责时，不得不将其他不需要的职责全都包含进来，从而造成冗余代码或代码的浪费。

##### 作用

单一职责原则的核心就是控制类的粒度大小、将对象解耦、提高其内聚性。

如果遵循单一职责原则将有以下优点。

###### 1.降低类的复杂度

一个类只负责一项职责，其逻辑肯定要比负责多项职责简单得多。

###### 2.提高类的可读性

复杂性降低，自然其可读性会提高。

###### 3.提高系统的可维护性

可读性提高，那自然更容易维护了。

###### 4.变更引起的风险降低

变更是必然的，如果单一职责原则遵守得好，当修改一个功能时，可以显著降低对其他功能的影响。

##### 实现方法

单一职责原则是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，再封装到不同的类或模块中。

而发现类的多重职责需要设计人员具有较强的分析设计能力和相关重构经验。

##### 说明示例

下面以大学学生工作管理程序为例介绍单一职责原则的应用。

分析：大学学生工作主要包括学生生活辅导和学生学业指导两个方面的工作，其中生活辅导主要包括班委建设、出勤统计、心理辅导、费用催缴、班级管理等工作，学业指导主要包括专业引导、学习辅导、科研指导、学习总结等工作。

如果将这些工作交给一位老师负责显然不合理，正确的做 法是生活辅导由辅导员负责，学业指导由学业导师负责，其类图如下所示：



注意：单一职责同样也适用于方法。一个方法应该尽可能做好一件事情。如果一个方法处理的事情太多，其颗粒度会变得很粗，不利于重用。

#### 接口隔离原则

##### 定义

接口隔离原则（Interface Segregation Principle，ISP）要求程序员尽量将臃肿庞大的接口拆分成更小的和更具体的接口，让接口中只包含客户感兴趣的方法。

2002 年罗伯特·C.马丁给“接口隔离原则”的定义是：客户端不应该被迫依赖于它不使用的方法（Clients should not be forced to depend on methods they do not use）。

该原则还有另外一个定义：一个类对另一个类的依赖应该建立在最小的接口上（The dependency of one class to another one should depend on the smallest possible interface）。

以上两个定义的含义是：要为各个类建立它们需要的专用接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。

接口隔离原则和单一职责都是为了提高类的内聚性、降低它们之间的耦合性，体现了封装的思想，但两者是不同的：

1.单一职责原则注重的是职责，而接口隔离原则注重的是对接口依赖的隔离

2.单一职责原则主要是约束类，它针对的是程序中的实现和细节；接口隔离原则主要约束接口，主要针对抽象和程序整体框架的构建。

##### 作用

接口隔离原则是为了约束接口、降低类对接口的依赖性，遵循接口隔离原则有以下 5 个优点：

1.将臃肿庞大的接口分解为多个粒度小的接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。

2.接口隔离提高了系统的内聚性，减少了对外交互，降低了系统的耦合性。

3.如果接口的粒度大小定义合理，能够保证系统的稳定性；但是，如果定义过小，则会造成接口数量过多，使设计复杂化；如果定义太大，灵活性降低，无法提供定制服务，给整体项目带来无法预料的风险。

4.使用多个专门的接口还能够体现对象的层次，因为可以通过接口的继承，实现对总接口的定义。

5.能减少项目工程中的代码冗余。过大的大接口里面通常放置许多不用的方法，当实现这个接口的时候，被迫设计冗余的代码。

##### 实现方法

在具体应用接口隔离原则时，应该根据以下几个规则来衡量：

###### 1.接口尽量小，但是要有限度

一个接口只服务于一个子模块或业务逻辑。

###### 2.为依赖接口的类定制服务

只提供调用者需要的方法，屏蔽不需要的方法。

###### 3.了解环境，拒绝盲从

每个项目或产品都有选定的环境因素，环境不同，接口拆分的标准就不同深入了解业务逻辑。

###### 4.提高内聚，减少对外交互

使接口用最少的方法去完成最多的事情。

##### 说明示例

下面以学生成绩管理程序为例介绍接口隔离原则的应用。

分析：学生成绩管理程序一般包含插入成绩、删除成绩、修改成绩、计算总分、计算均分、打印成绩信息、査询成绩信息等功能，如果将这些功能全部放到一个接口中显然不太合理，正确的做法是将它们分别放在输入模块、统计模块和打印模块等 3 个模块中，其类图如下所示：



程序代码如下：

package principle;

public class ISPtest

{

    public static void main(String[] args)

    {

        InputModule input =StuScoreList.getInputModule();

        CountModule count =StuScoreList.getCountModule();

        PrintModule print =StuScoreList.getPrintModule();

        input.insert();

        count.countTotalScore();

        print.printStuInfo();

        //print.delete();

    }

}

//输入模块接口

interface InputModule

{

    void insert();

    void delete();

    void modify();

}

//统计模块接口

interface CountModule

{

    void countTotalScore();

    void countAverage();

}

//打印模块接口

interface PrintModule

{

    void printStuInfo();

    void queryStuInfo();

}

//实现类

class StuScoreList implements InputModule,CountModule,PrintModule

{

    private StuScoreList(){}

    public static InputModule getInputModule()

    {

        return (InputModule)new StuScoreList();

    }

    public static CountModule getCountModule()

    {

        return (CountModule)new StuScoreList();

    }

    public static PrintModule getPrintModule()

    {

        return (PrintModule)new StuScoreList();

    }

    public void insert()

    {

        System.out.println("输入模块的insert()方法被调用！");

    }

    public void delete()

    {

        System.out.println("输入模块的delete()方法被调用！");

    }

    public void modify()

    {

        System.out.println("输入模块的modify()方法被调用！");

    }

    public void countTotalScore()

    {

        System.out.println("统计模块的countTotalScore()方法被调用！");

    }

    public void countAverage()

    {

        System.out.println("统计模块的countAverage()方法被调用！");

    }

    public void printStuInfo()

    {

        System.out.println("打印模块的printStuInfo()方法被调用！");

    }

    public void queryStuInfo()

    {

        System.out.println("打印模块的queryStuInfo()方法被调用！");

    }

}

程序的运行结果如下：

输入模块的insert()方法被调用！

统计模块的countTotalScore()方法被调用！

打印模块的printStuInfo()方法被调用！

#### 迪米特原则

##### 定义

迪米特法则（Law of Demeter，LoD）又叫作最少知识原则（Least Knowledge Principle，LKP)，产生于 1987 年美国东北大学（Northeastern University）的一个名为迪米特（Demeter）的研究项目，由伊恩·荷兰（Ian Holland）提出，被 UML 创始者之一的布奇（Booch）普及，后来又因为在经典著作《程序员修炼之道》（The Pragmatic Programmer）提及而广为人知。

迪米特法则的定义：只与你的直接朋友交谈，不跟“陌生人”说话（Talk only to your immediate friends and not to strangers）。其含义是：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

迪米特法则中的“朋友”：当前对象本身、当前对象的成员对象、当前对象所创建的对象、当前对象的方法参数等，这些对象同当前对象存在关联、聚合或组合关系，可以直接访问这些对象的方法。

##### 作用

迪米特法则要求限制软件实体之间通信的宽度和深度，正确使用迪米特法则将有以下两个优点：

1.降低了类之间的耦合度，提高了模块的相对独立性

2.由于亲合度降低，从而提高了类的可复用率和系统的扩展性。

但是，过度使用迪米特法则会使系统产生大量的中介类，从而增加系统的复杂性，使模块之间的通信效率降低。

所以，在釆用迪米特法则时需要反复权衡，确保高内聚和低耦合的同时，保证系统的结构清晰。

##### 实现方法

从迪米特法则的定义和特点可知，它强调以下两点：

1.从依赖者的角度来说，只依赖应该依赖的对象

2.从被依赖者的角度说，只暴露应该暴露的方法。

所以，在运用迪米特法则时要注意以下 6点：

1.在类的划分上，应该创建弱耦合的类。类与类之间的耦合越弱，就越有利于实现可复用的目标

2.在类的结构设计上，尽量降低类成员的访问权限

3.在类的设计上，优先考虑将一个类设置成不变类

4.在对其他类的引用上，将引用其他对象的次数降到最低

5.不暴露类的属性成员，而应该提供相应的访问器（set 和 get 方法）

6.谨慎使用序列化（Serializable）功能。

##### 说明示例

下面以明星与经纪人的关系为例介绍迪米特法则的应用。

分析：明星由于全身心投入艺术，所以许多日常事务由经纪人负责处理，如与粉丝的见面会，与媒体公司的业务洽淡等。

这里的经纪人是明星的朋友，而粉丝和媒体公司是陌生人，所以适合使用迪米特法则，其类图如下所示：



程序代码如下：

package principle;

public class LoDtest

{

public static void main(String[] args)

{

Agent agent=new Agent();

agent.setStar(new Star("林心如"));

agent.setFans(new Fans("粉丝韩丞"));

agent.setCompany(new Company("中国传媒有限公司"));

agent.meeting();

agent.business();

}

}

//经纪人

class Agent

{

private Star myStar;

private Fans myFans;

private Company myCompany;

public void setStar(Star myStar)

{

this.myStar=myStar;

}

public void setFans(Fans myFans)

{

this.myFans=myFans;

}

public void setCompany(Company myCompany)

{

this.myCompany=myCompany;

}

public void meeting()

{

System.out.println(myFans.getName()+"与明星"+myStar.getName()+"见面了。");

}

public void business()

{

System.out.println(myCompany.getName()+"与明星"+myStar.getName()+"洽淡业务。");

}

}

//明星

class Star

{

private String name;

Star(String name)

{

this.name=name;

}

public String getName()

{

return name;

}

}

//粉丝

class Fans

{

private String name;

Fans(String name)

{

this.name=name;

}

public String getName()

{

return name;

}

}

//媒体公司

class Company

{

private String name;

Company(String name)

{

this.name=name;

}

public String getName()

{

return name;

}

}

程序的运行结果如下：

粉丝韩丞与明星林心如见面了。

中国传媒有限公司与明星林心如洽淡业务。

#### 合成复用原则

##### 定义

合成复用原则（Composite Reuse Principle，CRP）又叫组合/聚合复用原则（Composition/Aggregate Reuse Principle，CARP）。

它要求在软件复用时，要尽量先使用组合或者聚合等关联关系来实现，其次才考虑使用继承关系来实现。

如果要使用继承关系，则必须严格遵循里氏替换原则。

合成复用原则同里氏替换原则相辅相成的，两者都是开闭原则的具体实现规范。

##### 作用

通常类的复用分为继承复用和合成复用两种，继承复用虽然有简单和易实现的优点，但它也存在以下缺点：

1. 继承复用破坏了类的封装性

因为继承会将父类的实现细节暴露给子类，父类对子类是透明的，所以这种复用又称为“白箱”复用。

1. 子类与父类的耦合度高

父类的实现的任何改变都会导致子类的实现发生变化，这不利于类的扩展与维护。

3.它限制了复用的灵活性

从父类继承而来的实现是静态的，在编译时已经定义，所以在运行时不可能发生变化。

采用组合或聚合复用时，可以将已有对象纳入新对象中，使之成为新对象的一部分，新对象可以调用已有对象的功能，它有以下优点：

1. 它维持了类的封装性

因为成分对象的内部细节是新对象看不见的，所以这种复用又称为“黑箱”复用。

2.新旧类之间的耦合度低

这种复用所需的依赖较少，新对象存取成分对象的唯一方法是通过成分对象的接口。

3.复用的灵活性高

这种复用可以在运行时动态进行，新对象可以动态地引用与成分对象类型相同的对象。

##### 实现方法

合成复用原则是通过将已有的对象纳入新对象中，作为新对象的成员对象来实现的，新对象可以调用已有对象的功能，从而达到复用。

##### 说明示例

下面以汽车分类管理程序为例来介绍合成复用原则的应用。

分析：汽车按“动力源”划分可分为汽油汽车、电动汽车等；按“颜色”划分可分为白色汽车、黑色汽车和红色汽车等。

如果同时考虑这两种分类，其组合就很多。下图所示是用继承关系实现的汽车分类的类图：



从上图可以看出用继承关系实现会产生很多子类，而且增加新的“动力源”或者增加新的“颜色”都要修改源代码，这违背了开闭原则，显然不可取。

但如果改用组合关系实现就能很好地解决以上问题，其类图如下所示：



#### 设计原则总结

7 种设计原则是软件设计模式必须尽量遵循的原则，各种原则要求的侧重点不同。具体如下：

1. 开闭原则是总纲，它告诉我们要对扩展开放，对修改关闭
2. 里氏替换原则告诉我们不要破坏继承体系
3. 依赖倒置原则告诉我们要面向接口编程
4. 单一职责原则告诉我们实现类要职责单一
5. 接口隔离原则告诉我们在设计接口的时候要精简单一
6. 迪米特法则告诉我们要降低耦合度
7. 合成复用原则告诉我们要优先使用组合或者聚合关系复用，少用继承关系复用。

## 设计模式详解

### 单例模式详解

在有些系统中，为了节省内存资源、保证数据内容的一致性，对某些类要求只能创建一个实例，这就是所谓的单例模式。

在有些系统中，为了节省内存资源、保证数据内容的一致性，对某些类要求只能创建一个实例，这就是所谓的单例模式。

#### 定义与特点

##### 定义

一个类只有一个实例，且该类能自行创建这个实例的一种模式。例如，Windows 中只能打开一个任务管理器，这样可以避免因打开多个任务管理器窗口而造成内存资源的浪费，或出现各个窗口显示内容的不一致等错误。

在计算机系统中，还有 Windows 的回收站、操作系统中的文件系统、多线程中的线程池、显卡的驱动程序对象、打印机的后台处理服务、应用程序的日志对象、数据库的连接池、网站的计数器、Web 应用的配置对象、应用程序中的对话框、系统中的缓存等常常被设计成单例。

##### 特点

单例模式有 3 个特点：

1.单例类只有一个实例对象

2.该单例对象必须由单例类自行创建

3.单例类对外提供一个访问该单例的全局访问点。

#### 结构与实现

单例模式是设计模式中最简单的模式之一。

通常，普通类的构造函数是公有的，外部类可以通过“new 构造函数()”来生成多个实例。

但是，如果将类的构造函数设为私有的，外部类就无法调用该构造函数，也就无法生成多个实例。

这时该类自身必须定义一个静态私有实例，并向外提供一个静态的公有函数用于创建或获取该静态私有实例。

下面来分析其基本结构和实现方法。

##### 结构

单例模式的主要角色如下：

单例类：包含一个实例且能自行创建这个实例的类。

访问类：使用单例的类。

其结构如下图所示：



##### 实现

Singleton 模式通常有两种实现形式。

###### 懒汉式单例

该模式的特点是类加载时没有生成单例，只有当第一次调用 getlnstance 方法时才去创建这个单例。

代码如下：

public class LazySingleton

{

private static volatile LazySingleton instance=null; //保证 instance 在所有线程中同步

private LazySingleton(){} //private 避免类在外部被实例化

public static synchronized LazySingleton getInstance()

{

//getInstance 方法前加同步

if(instance==null)

{

instance=new LazySingleton();

}

return instance;

}

}

注意：如果编写的是多线程程序，则不要删除上例代码中的关键字 volatile 和 synchronized，否则将存在线程非安全的问题。如果不删除这两个关键字就能保证线程安全，但是每次访问时都要同步，会影响性能，且消耗更多的资源，这是懒汉式单例的缺点。

###### 饿汉式单例

该模式的特点是类一旦加载就创建一个单例，保证在调用 getInstance 方法之前单例已经存在了。

代码如下：

public class HungrySingleton

{

private static final HungrySingleton instance=new HungrySingleton();

private HungrySingleton(){}

public static HungrySingleton getInstance()

{

return instance;

}

}

饿汉式单例在类创建的同时就已经创建好一个静态的对象供系统使用，以后不再改变，所以是线程安全的，可以直接用于多线程而不会出现问题。

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1338.html

#### 应用场景

前面分析了单例模式的结构与特点，以下是它通常适用的场景的特点：

1.在应用场景中，某类只要求生成一个对象的时候，如一个班中的班长、每个人的身份证号等

2.当对象需要被共享的场合。由于单例模式只允许创建一个对象，共享该对象可以节省内存，并加快对象访问速度。如 Web 中的配置对象、数据库的连接池等。

3.当某类需要频繁实例化，而创建的对象又频繁被销毁的时候，如多线程的线程池、网络连接池等。

#### 扩展

单例模式可扩展为有限的多例（Multitcm）模式，这种模式可生成有限个实例并保存在 ArmyList 中，客户需要时可随机获取，其结构图如下图所示：



### 工厂方法模式详解

#### 定义与特点

##### 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。

我们把被创建的对象称为“产品”，把创建产品的对象称为“工厂”。如果要创建的产品不多，只要一个工厂类就可以完成，这种模式叫“简单工厂模式”，它不属于 GoF 的 23 种经典设计模式，它的缺点是增加新产品时会违背“开闭原则”。

本节介绍的“工厂方法模式”是对简单工厂模式的进一步抽象化，其好处是可以使系统在不修改原来代码的情况下引进新的产品，即满足开闭原则。

##### 主要优点

1.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

2.在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，满足开闭原则。

##### 缺点

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。

#### 结构与实现

工厂方法模式由抽象工厂、具体工厂、抽象产品和具体产品等4个要素构成。

本节来分析其基本结构和实现方法。

##### 模式的结构

工厂方法模式的主要角色如下。

###### 抽象工厂（Abstract Factory）

提供了创建产品的接口，调用者通过它访问具体工厂的工厂方法 newProduct() 来创建产品。

###### 具体工厂（ConcreteFactory）

主要是实现抽象工厂中的抽象方法，完成具体产品的创建。

###### 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。

###### 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，它同具体工厂之间一一对应。

其结构图如下图所示：



##### 模式的实现

根据上图写出该模式的代码如下：

package FactoryMethod;

public class AbstractFactoryTest

{

public static void main(String[] args)

{

try

{

Product a;

AbstractFactory af;

af=(AbstractFactory) ReadXML1.getObject();

a=af.newProduct();

a.show();

}

catch(Exception e)

{

System.out.println(e.getMessage());

}

}

}

//抽象产品：提供了产品的接口

interface Product

{

public void show();

}

//具体产品1：实现抽象产品中的抽象方法

class ConcreteProduct1 implements Product

{

public void show()

{

System.out.println("具体产品1显示...");

}

}

//具体产品2：实现抽象产品中的抽象方法

class ConcreteProduct2 implements Product

{

public void show()

{

System.out.println("具体产品2显示...");

}

}

//抽象工厂：提供了厂品的生成方法

interface AbstractFactory

{

public Product newProduct();

}

//具体工厂1：实现了厂品的生成方法

class ConcreteFactory1 implements AbstractFactory

{

public Product newProduct()

{

System.out.println("具体工厂1生成-->具体产品1...");

return new ConcreteProduct1();

}

}

//具体工厂2：实现了厂品的生成方法

class ConcreteFactory2 implements AbstractFactory

{

public Product newProduct()

{

System.out.println("具体工厂2生成-->具体产品2...");

return new ConcreteProduct2();

}

}

package FactoryMethod;

import javax.xml.parsers.\*;

import org.w3c.dom.\*;

import java.io.\*;

class ReadXML1

{

//该方法用于从XML配置文件中提取具体类类名，并返回一个实例对象

public static Object getObject()

{

try

{

//创建文档对象

DocumentBuilderFactory dFactory=DocumentBuilderFactory.newInstance();

DocumentBuilder builder=dFactory.newDocumentBuilder();

Document doc;

doc=builder.parse(new File("src/FactoryMethod/config1.xml"));

//获取包含类名的文本节点

NodeList nl=doc.getElementsByTagName("className");

Node classNode=nl.item(0).getFirstChild();

String cName="FactoryMethod."+classNode.getNodeValue();

//System.out.println("新类名："+cName);

//通过类名生成实例对象并将其返回

Class<?> c=Class.forName(cName);

Object obj=c.newInstance();

return obj;

}

catch(Exception e)

{

e.printStackTrace();

return null;

}

}

}

程序运行结果如下：

具体工厂1生成-->具体产品1...

具体产品1显示...

如果将XM配置文件中的ConcreteFactory1改为ConcreteFactory2，则程序运行结果如下：

具体工厂2生成-->具体产品2...

具体产品2显示...

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1348.html

#### 应用场景

工厂方法模式通常适用于以下场景：

1.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

2.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

3.客户不关心创建产品的细节，只关心产品的品牌。

#### 扩展

当需要生成的产品不多且不会增加，一个具体工厂类就可以完成任务时，可删除抽象工厂类。

这时工厂方法模式将退化到简单工厂模式，其结构图如下图所示：



### 代理模式详解

在有些情况下，一个客户不能或者不想直接访问另一个对象，这时需要找一个中介帮忙完成某项任务，这个中介就是代理对象。例如，购买火车票不一定要去火车站买，可以通过 12306 网站或者去火车票代售点买。又如找女朋友、找保姆、找工作等都可以通过找中介完成。

在软件设计中，使用代理模式的例子也很多，例如，要访问的远程对象比较大（如视频或大图像等），其下载要花很多时间。还有因为安全原因需要屏蔽客户端直接访问真实对象，如某单位的内部数据库等。

#### 定义与特点

##### 定义

由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。

##### 主要优点

1.代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用

2.代理对象可以扩展目标对象的功能

3.代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度。

##### 主要缺点

1.在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢

2.增加了系统的复杂度。

#### 结构与实现

代理模式的结构比较简单，主要是通过定义一个继承抽象主题的代理来包含真实主题，从而实现对真实主题的访问，下面来分析其基本结构和实现方法。

##### 模式的结构

代理模式的主要角色如下：

###### 抽象主题（Subject）类

通过接口或抽象类声明真实主题和代理对象实现的业务方法。

###### 真实主题（Real Subject）类

实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象。

###### 代理（Proxy）类

提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或扩展真实主题的功能。

其结构图如下图所示：



##### 模式的实现

代理模式的实现代码如下：

package proxy;

public class ProxyTest

{

public static void main(String[] args)

{

Proxy proxy=new Proxy();

proxy.Request();

}

}

//抽象主题

interface Subject

{

void Request();

}

//真实主题

class RealSubject implements Subject

{

public void Request()

{

System.out.println("访问真实主题方法...");

}

}

//代理

class Proxy implements Subject

{

private RealSubject realSubject;

public void Request()

{

if (realSubject==null)

{

realSubject=new RealSubject();

}

preRequest();

realSubject.Request();

postRequest();

}

public void preRequest()

{

System.out.println("访问真实主题之前的预处理。");

}

public void postRequest()

{

System.out.println("访问真实主题之后的后续处理。");

}

}

程序运行的结果如下：

访问真实主题之前的预处理。

访问真实主题方法...

访问真实主题之后的后续处理。

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1359.html

#### 应用场景

前面分析了代理模式的结构与特点，现在来分析以下的应用场景：

1.远程代理，这种方式通常是为了隐藏目标对象存在于不同地址空间的事实，方便客户端访问。例如，用户申请某些网盘空间时，会在用户的文件系统中建立一个虚拟的硬盘，用户访问虚拟硬盘时实际访问的是网盘空间。

2.虚拟代理，这种方式通常用于要创建的目标对象开销很大时。例如，下载一幅很大的图像需要很长时间，因某种计算比较复杂而短时间无法完成，这时可以先用小比例的虚拟代理替换真实的对象，消除用户对服务器慢的感觉。

3.安全代理，这种方式通常用于控制不同种类客户对真实对象的访问权限。

4.智能指引，主要用于调用目标对象时，代理附加一些额外的处理功能。例如，增加计算真实对象的引用次数的功能，这样当该对象没有被引用时，就可以自动释放它。

5.延迟加载，指为了提高系统的性能，延迟对目标的加载。例如，Hibernate 中就存在属性的延迟加载和关联表的延时加载。

#### 扩展

在前面介绍的代理模式中，代理类中包含了对真实主题的引用，这种方式存在两个缺点。

1.真实主题与代理主题一一对应，增加真实主题也要增加代理

2.设计代理以前真实主题必须事先存在，不太灵活。采用动态代理模式可以解决以上问题，如 SpringAOP，其结构图如下图所示：



### 适配器模式详解

在现实生活中，经常出现两个对象因接口不兼容而不能在一起工作的实例，这时需要第三者进行适配。

例如，讲中文的人同讲英文的人对话时需要一个翻译，用直流电的笔记本电脑接交流电源时需要一个电源适配器，用计算机访问照相机的 SD 内存卡时需要一个读卡器等。

在软件设计中也可能出现：需要开发的具有某种业务功能的组件在现有的组件库中已经存在，但它们与当前系统的接口规范不兼容，如果重新开发这些组件成本又很高，这时用适配器模式能很好地解决这些问题。

#### 定义与特点

##### 定义

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。

适配器模式分为类结构型模式和对象结构型模式两种，前者类之间的耦合度比后者高，且要求程序员了解现有组件库中的相关组件的内部结构，所以应用相对较少些。

##### 主要优点

1.客户端通过适配器可以透明地调用目标接口

2.复用了现存的类，程序员不需要修改原有代码而重用现有的适配者类

3.将目标类和适配者类解耦，解决了目标类和适配者类接口不一致的问题。

##### 缺点

对类适配器来说，更换适配器的实现过程比较复杂。

#### 结构与实现

对类适配器来说，更换适配器的实现过程比较复杂。类适配器模式可采用多重继承方式实现，如 C++ 可定义一个适配器类来同时继承当前系统的业务接口和现有组件库中已经存在的组件接口；Java 不支持多继承，但可以定义一个适配器类来实现当前系统的业务接口，同时又继承现有组件库中已经存在的组件。

对象适配器模式可釆用将现有组件库中已经实现的组件引入适配器类中，该类同时实现当前系统的业务接口。

现在来介绍它们的基本结构。

##### 模式的结构

适配器模式（Adapter）包含以下主要角色：

1.目标（Target）接口

当前系统业务所期待的接口，它可以是抽象类或接口。

1. 适配者（Adaptee）类

它是被访问和适配的现存组件库中的组件接口。

3.适配器（Adapter）类

它是一个转换器，通过继承或引用适配者的对象，把适配者接口转换成目标接口，让客户按目标接口的格式访问适配者。

类适配器模式的结构图如下图所示：



对象适配器模式的结构图如下图所示：



##### 模式的实现

类适配器模式的代码如下：

package adapter;

//目标接口

interface Target

{

public void request();

}

//适配者接口

class Adaptee

{

public void specificRequest()

{

System.out.println("适配者中的业务代码被调用！");

}

}

//类适配器类

class ClassAdapter extends Adaptee implements Target

{

public void request()

{

specificRequest();

}

}

//客户端代码

public class ClassAdapterTest

{

public static void main(String[] args)

{

System.out.println("类适配器模式测试：");

Target target = new ClassAdapter();

target.request();

}

}

程序的运行结果如下：

类适配器模式测试：

适配者中的业务代码被调用！

对象适配器模式的代码如下：

package adapter;

//对象适配器类

class ObjectAdapter implements Target

{

private Adaptee adaptee;

public ObjectAdapter(Adaptee adaptee)

{

this.adaptee=adaptee;

}

public void request()

{

adaptee.specificRequest();

}

}

//客户端代码

public class ObjectAdapterTest

{

public static void main(String[] args)

{

System.out.println("对象适配器模式测试：");

Adaptee adaptee = new Adaptee();

Target target = new ObjectAdapter(adaptee);

target.request();

}

}

程序的运行结果如下：

对象适配器模式测试：

适配者中的业务代码被调用！

特别说明：对象适配器模式中的“目标接口”和“适配者类”的代码同类适配器模式一样，只要修改适配器类和客户端的代码即可。

#### 应用实例

详细应用实例请到原文链接查看：<http://c.biancheng.net/view/1361.html>

#### 应用场景

适配器模式（Adapter）通常适用于以下场景：

1.以前开发的系统存在满足新系统功能需求的类，但其接口同新系统的接口不一致

2.使用第三方提供的组件，但组件接口定义和自己要求的接口定义不同。

#### 扩展

适配器模式（Adapter）可扩展为双向适配器模式，双向适配器类既可以把适配者接口转换成目标接口，也可以把目标接口转换成适配者接口，其结构图如下图所示：



程序代码如下：

package adapter;

//目标接口

interface TwoWayTarget

{

public void request();

}

//适配者接口

interface TwoWayAdaptee

{

public void specificRequest();

}

//目标实现

class TargetRealize implements TwoWayTarget

{

public void request()

{

System.out.println("目标代码被调用！");

}

}

//适配者实现

class AdapteeRealize implements TwoWayAdaptee

{

public void specificRequest()

{

System.out.println("适配者代码被调用！");

}

}

//双向适配器

class TwoWayAdapter implements TwoWayTarget,TwoWayAdaptee

{

private TwoWayTarget target;

private TwoWayAdaptee adaptee;

public TwoWayAdapter(TwoWayTarget target)

{

this.target=target;

}

public TwoWayAdapter(TwoWayAdaptee adaptee)

{

this.adaptee=adaptee;

}

public void request()

{

adaptee.specificRequest();

}

public void specificRequest()

{

target.request();

}

}

//客户端代码

public class TwoWayAdapterTest

{

public static void main(String[] args)

{

System.out.println("目标通过双向适配器访问适配者：");

TwoWayAdaptee adaptee=new AdapteeRealize();

TwoWayTarget target=new TwoWayAdapter(adaptee);

target.request();

System.out.println("-------------------");

System.out.println("适配者通过双向适配器访问目标：");

target=new TargetRealize();

adaptee=new TwoWayAdapter(target);

adaptee.specificRequest();

}

}

程序的运行结果如下：

目标通过双向适配器访问适配者：

适配者代码被调用！

-------------------

适配者通过双向适配器访问目标：

目标代码被调用！

### 装饰模式详解

在现实生活中，常常需要对现有产品增加新的功能或美化其外观，如房子装修、相片加相框等。

在软件开发过程中，有时想用一些现存的组件。这些组件可能只是完成了一些核心功能。但在不改变其结构的情况下，可以动态地扩展其功能。

所有这些都可以釆用装饰模式来实现。.

#### 定义与特点

##### 定义

在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即增加其额外功能）的模式，它属于对象结构型模式。

##### 主要优点

1.采用装饰模式扩展对象的功能比采用继承方式更加灵活

2.可以设计出多个不同的具体装饰类，创造出多个不同行为的组合。

##### 主要缺点

装饰模式增加了许多子类，如果过度使用会使程序变得很复杂。

#### 结构与实现

通常情况下，扩展一个类的功能会使用继承方式来实现。

但继承具有静态特征，耦合度高，并且随着扩展功能的增多，子类会很膨胀。

如果使用组合关系来创建一个包装对象（即装饰对象）来包裹真实对象，并在保持真实对象的类结构不变的前提下，为其提供额外的功能，这就是装饰模式的目标。下面来分析其基本结构和实现方法。

##### 模式的结构

装饰模式主要包含以下角色

1.抽象构件（Component）角色

定义一个抽象接口以规范准备接收附加责任的对象。

2.具体构件（Concrete Component）角色

实现抽象构件，通过装饰角色为其添加一些职责。

3.抽象装饰（Decorator）角色

继承抽象构件，并包含具体构件的实例，可以通过其子类扩展具体构件的功能。

4.具体装饰（ConcreteDecorator）角色

实现抽象装饰的相关方法，并给具体构件对象添加附加的责任。

装饰模式的结构图下图 所示：



##### 模式的实现

装饰模式的实现代码如下：

package decorator;

public class DecoratorPattern

{

public static void main(String[] args)

{

Component p=new ConcreteComponent();

p.operation();

System.out.println("---------------------------------");

Component d=new ConcreteDecorator(p);

d.operation();

}

}

//抽象构件角色

interface Component

{

public void operation();

}

//具体构件角色

class ConcreteComponent implements Component

{

public ConcreteComponent()

{

System.out.println("创建具体构件角色");

}

public void operation()

{

System.out.println("调用具体构件角色的方法operation()");

}

}

//抽象装饰角色

class Decorator implements Component

{

private Component component;

public Decorator(Component component)

{

this.component=component;

}

public void operation()

{

component.operation();

}

}

//具体装饰角色

class ConcreteDecorator extends Decorator

{

public ConcreteDecorator(Component component)

{

super(component);

}

public void operation()

{

super.operation();

addedFunction();

}

public void addedFunction()

{

System.out.println("为具体构件角色增加额外的功能addedFunction()");

}

}

程序运行结果如下：

创建具体构件角色

调用具体构件角色的方法operation()

---------------------------------

调用具体构件角色的方法operation()

为具体构件角色增加额外的功能addedFunction()

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1366.html

#### 应用场景

前面讲解了关于装饰模式的结构与特点，下面介绍其适用的应用场景，装饰模式通常在以下几种情况使用：

1.当需要给一个现有类添加附加职责，而又不能采用生成子类的方法进行扩充时。例如，该类被隐藏或者该类是终极类或者采用继承方式会产生大量的子类

2.当需要通过对现有的一组基本功能进行排列组合而产生非常多的功能时，采用继承关系很难实现，而采用装饰模式却很好实现

3.当对象的功能要求可以动态地添加，也可以再动态地撤销时。

装饰模式在 Java 语言中的最著名的应用莫过于 Java I/O 标准库的设计了。

例如，InputStream 的子类 FilterInputStream，OutputStream 的子类 FilterOutputStream，Reader 的子类 BufferedReader 以及 FilterReader，还有 Writer 的子类 BufferedWriter、FilterWriter 以及 PrintWriter 等，它们都是抽象装饰类。

下面代码是为 FileReader 增加缓冲区而采用的装饰类 BufferedReader 的例子：

BufferedReader in=new BufferedReader(new FileReader("filename.txtn));

String s=in.readLine();

#### 扩展

装饰模式所包含的 4 个角色不是任何时候都要存在的，在有些应用环境下模式是可以简化的，如以下两种情况：

1.如果只有一个具体构件而没有抽象构件时，可以让抽象装饰继承具体构件，其结构图如下图所示：



2.如果只有一个具体装饰时，可以将抽象装饰和具体装饰合并，其结构图如下图所示：



### 模板方法模式详解

在面向对象程序设计过程中，程序员常常会遇到这种情况：设计一个系统时知道了算法所需的关键步骤，而且确定了这些步骤的执行顺序，但某些步骤的具体实现还未知，或者说某些步骤的实现与具体的环境相关。

例如，去银行办理业务一般要经过以下4个流程：取号、排队、办理具体业务、对银行工作人员进行评分等，其中取号、排队和对银行工作人员进行评分的业务对每个客户是一样的，可以在父类中实现，但是办理具体业务却因人而异，它可能是存款、取款或者转账等，可以延迟到子类中实现。

这样的例子在生活中还有很多，例如，一个人每天会起床、吃饭、做事、睡觉等，其中“做事”的内容每天可能不同。我们把这些规定了流程或格式的实例定义成模板，允许使用者根据自己的需求去更新它，例如，简历模板、论文模板、Word 中模板文件等。

以下介绍的模板方法模式将解决以上类似的问题。

#### 定义与特点

##### 定义

定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类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。它是一种类行为型模式。

##### 主要优点

1.它封装了不变部分，扩展可变部分。它把认为是不变部分的算法封装到父类中实现，而把可变部分算法由子类继承实现，便于子类继续扩展

2.它在父类中提取了公共的部分代码，便于代码复用

3.部分方法是由子类实现的，因此子类可以通过扩展方式增加相应的功能，符合开闭原则。

##### 主要缺点

1.对每个不同的实现都需要定义一个子类，这会导致类的个数增加，系统更加庞大，设计也更加抽象

2.父类中的抽象方法由子类实现，子类执行的结果会影响父类的结果，这导致一种反向的控制结构，它提高了代码阅读的难度。

#### 结构与实现

模板方法模式需要注意抽象类与具体子类之间的协作。

它用到了虚函数的多态性技术以及“不用调用我，让我来调用你”的反向控制技术。现在来介绍它们的基本结构。

##### 模式的结构

模板方法模式包含以下主要角色：

###### 抽象类（Abstract Class）

负责给出一个算法的轮廓和骨架。它由一个模板方法和若干个基本方法构成。这些方法的定义如下：

1.模板方法：定义了算法的骨架，按某种顺序调用其包含的基本方法。

2.基本方法：是整个算法中的一个步骤，包含以下几种类型：

抽象方法：在抽象类中申明，由具体子类实现。

具体方法：在抽象类中已经实现，在具体子类中可以继承或重写它。

钩子方法：在抽象类中已经实现，包括用于判断的逻辑方法和需要子类重写的空方法两种。

###### 具体子类（Concrete Class）

实现抽象类中所定义的抽象方法和钩子方法，它们是一个顶级逻辑的一个组成步骤。

模板方法模式的结构图下图所示：



##### 模式的实现

模板方法模式的代码如下：

package templateMethod;

public class TemplateMethodPattern

{

public static void main(String[] args)

{

AbstractClass tm=new ConcreteClass();

tm.TemplateMethod();

}

}

//抽象类

abstract class AbstractClass

{

public void TemplateMethod() //模板方法

{

SpecificMethod();

abstractMethod1();

abstractMethod2();

}

public void SpecificMethod() //具体方法

{

System.out.println("抽象类中的具体方法被调用...");

}

public abstract void abstractMethod1(); //抽象方法1

public abstract void abstractMethod2(); //抽象方法2

}

//具体子类

class ConcreteClass extends AbstractClass

{

public void abstractMethod1()

{

System.out.println("抽象方法1的实现被调用...");

}

public void abstractMethod2()

{

System.out.println("抽象方法2的实现被调用...");

}

}

程序的运行结果如下：

抽象类中的具体方法被调用...

抽象方法1的实现被调用...

抽象方法2的实现被调用...

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1376.html

#### 应用场景

模板方法模式通常适用于以下场景：

1.算法的整体步骤很固定，但其中个别部分易变时，这时候可以使用模板方法模式，将容易变的部分抽象出来，供子类实现。

2.当多个子类存在公共的行为时，可以将其提取出来并集中到一个公共父类中以避免代码重复。首先，要识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。

3.当需要控制子类的扩展时，模板方法只在特定点调用钩子操作，这样就只允许在这些点进行扩展。

#### 扩展

在模板方法模式中，基本方法包含：抽象方法、具体方法和钩子方法，正确使用“钩子方法”可以使得子类控制父类的行为。

如下面例子中，可以通过在具体子类中重写钩子方法 HookMethod1() 和 HookMethod2() 来改变抽象父类中的运行结果，其结构图如下图 所示：



程序代码如下：

package templateMethod;

public class HookTemplateMethod

{

public static void main(String[] args)

{

HookAbstractClass tm=new HookConcreteClass();

tm.TemplateMethod();

}

}

//含钩子方法的抽象类

abstract class HookAbstractClass

{

public void TemplateMethod() //模板方法

{

abstractMethod1();

HookMethod1();

if(HookMethod2())

{

SpecificMethod();

}

abstractMethod2();

}

public void SpecificMethod() //具体方法

{

System.out.println("抽象类中的具体方法被调用...");

}

public void HookMethod1(){} //钩子方法1

public boolean HookMethod2() //钩子方法2

{

return true;

}

public abstract void abstractMethod1(); //抽象方法1

public abstract void abstractMethod2(); //抽象方法2

}

//含钩子方法的具体子类

class HookConcreteClass extends HookAbstractClass

{

public void abstractMethod1()

{

System.out.println("抽象方法1的实现被调用...");

}

public void abstractMethod2()

{

System.out.println("抽象方法2的实现被调用...");

}

public void HookMethod1()

{

System.out.println("钩子方法1被重写...");

}

public boolean HookMethod2()

{

return false;

}

}

程序的运行结果如下：

抽象方法1的实现被调用...

钩子方法1被重写...

抽象方法2的实现被调用...

如果钩子方法 HookMethod1() 和钩子方法 HookMethod2() 的代码改变，则程序的运行结果也会改变。

### 观察者模式详解

在现实世界中，许多对象并不是独立存在的，其中一个对象的行为发生改变可能会导致一个或者多个其他对象的行为也发生改变。例如，某种商品的物价上涨时会导致部分商家高兴，而消费者伤心；还有，当我们开车到交叉路口时，遇到红灯会停，遇到绿灯会行。

这样的例子还有很多，例如，股票价格与股民、微信公众号与微信用户、气象局的天气预报与听众、小偷与警察等。

在软件世界也是这样，例如，Excel 中的数据与折线图、饼状图、柱状图之间的关系；MVC 模式中的模型与视图的关系；事件模型中的事件源与事件处理者。所有这些，如果用观察者模式来实现就非常方便。

#### 定义与特点

##### 定义

多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

这种模式有时又称作发布-订阅模式、模型-视图模式，它是对象行为型模式。

##### 主要优点

1.降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系

2.目标与观察者之间建立了一套触发机制。

##### 主要缺点

1.目标与观察者之间的依赖关系并没有完全解除，而且有可能出现循环引用

2.当观察者对象很多时，通知的发布会花费很多时间，影响程序的效率。

#### 结构与实现

实现观察者模式时要注意具体目标对象和具体观察者对象之间不能直接调用，否则将使两者之间紧密耦合起来，这违反了面向对象的设计原则。

##### 模式的结构

观察者模式的主要角色如下:

###### 抽象主题（Subject）角色

也叫抽象目标类，它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。

###### 具体主题（Concrete Subject）角色

也叫具体目标类，它实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。

###### 抽象观察者（Observer）角色

它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。

###### 具体观察者（Concrete Observer）角色

实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。

观察者模式的结构图如下图所示:



##### 模式的实现

观察者模式的实现代码如下：

package observer;

import java.util.\*;

public class ObserverPattern

{

public static void main(String[] args)

{

Subject subject=new ConcreteSubject();

Observer obs1=new ConcreteObserver1();

Observer obs2=new ConcreteObserver2();

subject.add(obs1);

subject.add(obs2);

subject.notifyObserver();

}

}

//抽象目标

abstract class Subject

{

protected List<Observer> observers=new ArrayList<Observer>();

//增加观察者方法

public void add(Observer observer)

{

observers.add(observer);

}

//删除观察者方法

public void remove(Observer observer)

{

observers.remove(observer);

}

public abstract void notifyObserver(); //通知观察者方法

}

//具体目标

class ConcreteSubject extends Subject

{

public void notifyObserver()

{

System.out.println("具体目标发生改变...");

System.out.println("--------------");

for(Object obs:observers)

{

((Observer)obs).response();

}

}

}

//抽象观察者

interface Observer

{

void response(); //反应

}

//具体观察者1

class ConcreteObserver1 implements Observer

{

public void response()

{

System.out.println("具体观察者1作出反应！");

}

}

//具体观察者1

class ConcreteObserver2 implements Observer

{

public void response()

{

System.out.println("具体观察者2作出反应！");

}

}

程序运行结果如下：

具体目标发生改变...

--------------

具体观察者1作出反应！

具体观察者2作出反应！

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1390.html

#### 应用场景

通过前面的分析与应用实例可知观察者模式适合以下几种情形：

1.对象间存在一对多关系，一个对象的状态发生改变会影响其他对象

2.当一个抽象模型有两个方面，其中一个方面依赖于另一方面时，可将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

#### 扩展

在Java中，通过java.util.Observable类和java.util.Observer接口定义了观察者模式，只要实现它们的子类就可以编写观察者模式实例。

##### Observable类

Observable 类是抽象目标类，它有一个 Vector 向量，用于保存所有要通知的观察者对象，下面来介绍它最重要的 3 个方法：

void addObserver(Observer o) 方法：用于将新的观察者对象添加到向量中。

void notifyObservers(Object arg) 方法：调用向量中的所有观察者对象的update方法，通知它们数据发生改变。通常越晚加入向量的观察者越先得到通知。

void setChange() 方法：用来设置一个 boolean 类型的内部标志位，注明目标对象发生了变化。当它为真时，notifyObservers() 才会通知观察者。

##### Observer 接口

Observer 接口是抽象观察者，它监视目标对象的变化，当目标对象发生变化时，观察者得到通知，并调用void update(Observable o,Object arg) 方法，进行相应的工作。

##### 示例

利用 Observable 类和 Observer 接口实现原油期货的观察者模式实例。

分析：当原油价格上涨时，空方伤心，多方高兴；当油价下跌时，空方高兴，多方伤心。本实例中的抽象目标（Observable)类在 Java 中已经定义，可以直接定义其子类，即原油期货（OilFutures)类，它是具体目标类，该类中定义一个 SetPriCe(float price) 方法，当原油数据发生变化时调用其父类的 notifyObservers(Object arg) 方法来通知所有观察者；另外，本实例中的抽象观察者接口（Observer）在 Java 中已经定义，只要定义其子类，即具体观察者类（包括多方类 Bull 和空方类 Bear），并实现 update(Observable o,Object arg) 方法即可。其结构图如下图所示：



程序代码如下：

package observer;

import java.util.Observer;

import java.util.Observable;

public class CrudeOilFutures

{

public static void main(String[] args)

{

OilFutures oil=new OilFutures();

Observer bull=new Bull(); //多方

Observer bear=new Bear(); //空方

oil.addObserver(bull);

oil.addObserver(bear);

oil.setPrice(10);

oil.setPrice(-8);

}

}

//具体目标类：原油期货

class OilFutures extends Observable

{

private float price;

public float getPrice()

{

return this.price;

}

public void setPrice(float price)

{

super.setChanged() ; //设置内部标志位，注明数据发生变化

super.notifyObservers(price); //通知观察者价格改变了

this.price=price ;

}

}

//具体观察者类：多方

class Bull implements Observer

{

public void update(Observable o,Object arg)

{

Float price=((Float)arg).floatValue();

if(price>0)

{

System.out.println("油价上涨"+price+"元，多方高兴了！");

}

else

{

System.out.println("油价下跌"+(-price)+"元，多方伤心了！");

}

}

}

//具体观察者类：空方

class Bear implements Observer

{

public void update(Observable o,Object arg)

{

Float price=((Float)arg).floatValue();

if(price>0)

{

System.out.println("油价上涨"+price+"元，空方伤心了！");

}

else

{

System.out.println("油价下跌"+(-price)+"元，空方高兴了！");

}

}

}

程序运行结果如下：

油价上涨10.0元，空方伤心了！

油价上涨10.0元，多方高兴了！

油价下跌8.0元，空方高兴了！

油价下跌8.0元，多方伤心了！

### 策略模式详解

在现实生活中常常遇到实现某种目标存在多种策略可供选择的情况，例如，出行旅游可以乘坐飞机、乘坐火车、骑自行车或自己开私家车等，超市促销可以釆用打折、送商品、送积分等方法。

在软件开发中也常常遇到类似的情况，当实现某一个功能存在多种算法或者策略，我们可以根据环境或者条件的不同选择不同的算法或者策略来完成该功能，如数据排序策略有冒泡排序、选择排序、插入排序、二叉树排序等。

如果使用多重条件转移语句实现（即硬编码），不但使条件语句变得很复杂，而且增加、删除或更换算法要修改原代码，不易维护，违背开闭原则。如果采用策略模式就能很好解决该问题。

#### 定义与特点

##### 定义

该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，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

##### 主要优点

1.多重条件语句不易维护，而使用策略模式可以避免使用多重条件语句

2.策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转移到父类里面，从而避免重复的代码

3.策略模式可以提供相同行为的不同实现，客户可以根据不同时间或空间要求选择不同的

4.策略模式提供了对开闭原则的完美支持，可以在不修改原代码的情况下，灵活增加新算法

5.策略模式把算法的使用放到环境类中，而算法的实现移到具体策略类中，实现了二者的分离。

##### 主要缺点

1.客户端必须理解所有策略算法的区别，以便适时选择恰当的算法类

2.策略模式造成很多的策略类。

#### 结构与实现

策略模式是准备一组算法，并将这组算法封装到一系列的策略类里面，作为一个抽象策略类的子类。

策略模式的重心不是如何实现算法，而是如何组织这些算法，从而让程序结构更加灵活，具有更好的维护性和扩展性，现在我们来分析其基本结构和实现方法。

##### 模式的结构

策略模式的主要角色如下：

1. 抽象策略（Strategy）类

定义了一个公共接口，各种不同的算法以不同的方式实现这个接口，环境角色使用这个接口调用不同的算法，一般使用接口或抽象类实现。

1. 具体策略（Concrete Strategy）类

实现了抽象策略定义的接口，提供具体的算法实现。

3.环境（Context）类

持有一个策略类的引用，最终给客户端调用。

其结构图如下图所示：



##### 模式的实现

策略模式的实现代码如下：

package strategy;

public class StrategyPattern

{

public static void main(String[] args)

{

Context c=new Context();

Strategy s=new ConcreteStrategyA();

c.setStrategy(s);

c.strategyMethod();

System.out.println("-----------------");

s=new ConcreteStrategyB();

c.setStrategy(s);

c.strategyMethod();

}

}

//抽象策略类

interface Strategy

{

public void strategyMethod(); //策略方法

}

//具体策略类A

class ConcreteStrategyA implements Strategy

{

public void strategyMethod()

{

System.out.println("具体策略A的策略方法被访问！");

}

}

//具体策略类B

class ConcreteStrategyB implements Strategy

{

public void strategyMethod()

{

System.out.println("具体策略B的策略方法被访问！");

}

}

//环境类

class Context

{

private Strategy strategy;

public Strategy getStrategy()

{

return strategy;

}

public void setStrategy(Strategy strategy)

{

this.strategy=strategy;

}

public void strategyMethod()

{

strategy.strategyMethod();

}

}

程序运行结果如下：

具体策略A的策略方法被访问！

-----------------

具体策略B的策略方法被访问！

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1378.html

#### 应用场景

策略模式在很多地方用到，如 Java SE 中的容器布局管理就是一个典型的实例，Java SE 中的每个容器都存在多种布局供用户选择。

在程序设计中，通常在以下几种情况中使用策略模式较多：

1.一个系统需要动态地在几种算法中选择一种时，可将每个算法封装到策略类中

2.一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现，可将每个条件分支移入它们各自的策略类中以代替这些条件语句

3.系统中各算法彼此完全独立，且要求对客户隐藏具体算法的实现细节时

4.系统要求使用算法的客户不应该知道其操作的数据时，可使用策略模式来隐藏与算法相关的数据结构

5.多个类只区别在表现行为不同，可以使用策略模式，在运行时动态选择具体要执行的行为。

#### 扩展

在一个使用策略模式的系统中，当存在的策略很多时，客户端管理所有策略算法将变得很复杂，如果在环境类中使用策略工厂模式来管理这些策略类将大大减少客户端的工作复杂度，其结构图如下图所示：



## 设计模式总结

## 文档参考

大话设计模式文档下载

<https://blog.csdn.net/hemingyang97/article/details/82025608>

设计模式-在线手册教程

<http://www.php.cn/course/58.html>

ProcessOn设计模式

<https://www.processon.com/view/5b503678e4b0f8477d8a22fa#map>

ProcessOn设计模式之间的关系

<https://www.processon.com/view/5a371fa8e4b0dce99f02f005>

Java设计模式：23种设计模式全面解析（超级详细）

<http://c.biancheng.net/design_pattern>

【设计模式】GoF设计模式学习总结

<https://www.cnblogs.com/chenpi/p/5222597.html>