# 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

##### 作用

##### 实现方法

#### 接口隔离原则

##### 定义

##### 作用

##### 实现方法

#### 迪米特原则

##### 定义

##### 作用

##### 实现方法

#### 合成复用原则

##### 定义

##### 作用

##### 实现方法

#### 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

## 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>