# 面向对象设计原则概述

1. 软件的可维护性和可复用性是用于衡量软件质量的两个重要属性，软件的可维护性是指软件能够被理解、改正、适应以及扩展的难易程度，软件的可复用性是指软件能够被重复使用的难易程度。
2. 面向对象设计的目标之一就是在于支持可维护复用，一方面需要实现设计方案或者源代码的复用，另一方面要确保系统能够易于扩展和修改，具有良好的可维护性，而面向对象设计原则就是为支持可维护复用而诞生，这些原则蕴含在设计模式中，它们是从许多设计方案中总结出来的指导性原则，但并不是强制的。
3. 面向对象设计原则是学习设计模式的基础，每一个设计模式都符合某一个或多个面向对象设计原则，面向对象设计原则是用于评价一个设计模式的使用效果的重要指标之一，使用这些面向对象设计原则可以提高软件的可维护性和复用性，实现可维护性复用的目标。

最常见的7个面向对象设计原则如下：

|  |  |
| --- | --- |
| **设计原则名称** | **定义** |
| 单一职责原则 | 一个对象应该只包含单一的职责，并且该职责被完整地封装在一个类中 |
| 开闭原则 | 软件实体应当对扩展开放，对修改关闭 |
| 里式代换原则 | 所有引用基类的地方必须能透明地使用其子类的对象 |
| 依赖倒转原则 | 高层模块不应该依赖底层模块，它们都应该依赖抽象。抽象不应该依赖于细节，细节应该依赖于抽象 |
| 接口隔离原则 | 客户端不应该依赖那些它不需要的接口 |
| 合成复用原则 | 优先使用对象组合，而不是通过继承来达到复用的目的 |
| 迪米特法则 | 每一个软件单位对其他单位都只有最少的知识，而且局限于那些本单位密切相关的软件单位 |

# 单一职责原则

## 定义

单一职责用于控制类的粒度大小，它的定义为：

1）一个对象应该只包含单一的职责，并且该职责被完整的封装在一个类中。

2）就一个类而言，应该仅有一个引起它变化的原因。

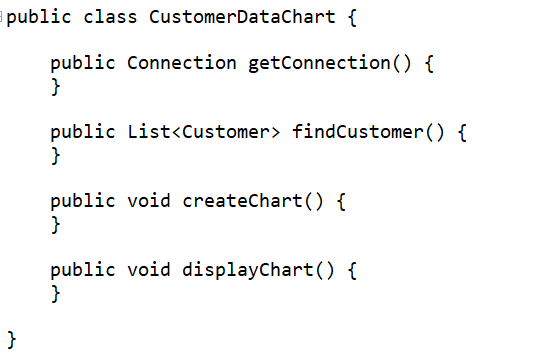
## 解释

在软件系统中，一个类（大到模块，小到方法）承担的职责越多，它被复用的可能性就越小，而且一个类承担的职责过多，相当于将这些职责耦合在一起，当其中一个职责变化时可能会影响其他职责的运作，因此要将这些职责进行分离，将不同的职责封装在不同的类中。

单一职责原则是实现高内聚、低耦合的指导方针，需要设计人员发现类的不同职责并将其分离，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实践经验。

## 案例

某软件公司开发人员针对CRM（客户关系管理）系统中的客户信息图形设计模块提出了以下的初始设计方案：



在CustomerDataChart类的方法中，getConnection( )用于连接数据库，findCustomers( )用于查询所有的客户信息，createChart( )用于创建图表，displayChart( )用于显示图表，可以看出CustomerDataChart类承担了太多的职责，既包含了数据库相关的方法，又包含了与图表生成和显示的相关方法，如果在其他类中也需要连接数据库或者使用findCustomers( )方法查询客户信息，则难以实现代码的重用。无论是修改数据库连接方式还是需要修改图表显示方式都需要修改该类，这个类就拥有了不止一个引起它变化的原因，违背了单一职责原则，因此需要对其进行拆分，利用单一职责原则对其进行重构，使其满足单一职责原则，可将CustomerDataChart类可拆分为以下3个类：

1. DBUtil：负责连接数据库，包含数据库连接方法getConnection( )；
2. CustomerDAO：负责操作数据库中的Customer表，包含对Customer表的增删查改等方法，例如findCustomers( )方法；
3. CustomerDataChart：负责图表的生存和显示，包含createChart( )和displayChart( )方法；

# 开闭原则

## 定义

开闭原则的定义为：软件实体应当对扩展开放，对修改关闭。

在开闭原则的定义中，软件实体可以指一个软件模块、一个由多个类组成的局部结构或一个独立的类。

开闭原则就是指软件实体应尽量在不修改原有代码的情况下进行扩展。

## 解释

任何软件都需要面临一个问题，即它们的需求会随时间的推移而发生变化，当软件系统需要面对新的需求时应该尽量保证系统的设计框架是稳定的。如果一个软件设计符合开闭原则，那么可以非常方便地对系统进行扩展，而且在扩展时无需修改现有代码，使得软件系统在拥有适应性和灵活性的同时具备较好的稳定性和延续性。随着软件规模越来越大，软件寿命越来越长，软件维护成本越来越高，设计满足开闭原则的软件系统也变得越来越重要。

为了满足开闭原则，需要对系统进行抽象画设计，抽象化时开闭原则的关键。在Java、C#等编程语言中可以为系统定义一个相对稳定的抽象层，而将不同的实现行为移至具体的实现层中完成。在很多的面向对象编程语言中都提供了接口、抽象类等机制，可以通过它们定义系统的抽象层，再通过具体类进行扩展。如果需要修改系统的行为，无需对抽象层进行任何改动，只需要增加新的具体类来实现新的业务功能即可，实现在不修改已有代码的基础上扩展系统的功能，达到开闭原则的要求。

# 里式替换原则

## 定义

里式替换原则的定义为：所有引用基类（父类）的地方必须能够透明地使用其子类的对象。

## 解释

里式替换原则表明，在软件中将一个基类（父类）对象替换成它的子类对象，程序将不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类对象，那么它不一定会能够使用基类（父类）对象。

里式替换原则是实现开闭原则的重要方式之一，由于在使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。

在运用里式替换原则时，应该将父类设计为抽象类或接口，让子类继承父类或实现接口，并实现在父类中声明的方法，在运行时子类实例替换父类实例，可以很方便地扩展系统的功能，无须修改原有子类中的代码，增加新的功能可以通过增加一个新的子类来实现。

# 依赖倒转原则

## 定义

依赖倒转原则的定义为：高层模块不应该依赖于底层模块，它们都应该依赖于抽象。抽象不应该依赖于细节，细节应该依赖于抽象。

## 解释

依赖倒转原则要求在程序代码中传递参数时或在关联关系中尽量引用层次高的抽象层类，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等，而不要用具体类来做这些事情。为了确保该原则的应用，一个具体类应当只实现接口或抽象类中声明过的方法，而不要给出多于的方法，否则将无法调用到在子类中增加的新方法。

# 接口隔离原则

## 定义

接口隔离原则的定义为：客户端不应该依赖那些它不需要的接口。

## 解释

当一个接口太大时需要将它分割成一些更细小的接口，使用该接口的客户端仅需知道与之相关的方法即可。

每一个接口都应该承担一种相对独立的角色，不干不该干的事，该干的都要干。

在面向对象编程语言中，实现一个接口需要实现该接口中定义的所有方法，因此大的总接口使用起来不一定方便，为了使接口的职责单一，需要将大接口中的方法根据其职责不同分别放在不同的小接口中，以确保每个接口使用起来都较为方便，并都承担某一单一角色。

接口应该尽量细化，同时接口中的方法应该尽量少，每个接口中只包含一个子模块或业务逻辑类所需的方法即可。

如果某接口中定义了太多方法，即该接口承担了太多职责，一方面导致该接口的实现类很庞大，在不同的实现类中都不得不实现接口中定义的所有方法，灵活性变差，如果实现类中出现了大量的空方法，将导致系统中产生了大量的无用代码，影响代码质量。

在使用接口隔离原则时也需要注意，接口不能太小，如果太小会导致系统中的接口泛滥，不利于维护，接口也不能太大，太大的接口将违背接口隔离原则，灵活性较差，使用起来不方便。

# 合成复用原则

## 定义

合成复用原则的定义为：优先使用对象组合，而不是通过继承来达到复用的目的。

## 解释

合成复用原则就是在一个对象里通过关联关系来使用一些已有的对象，使之成为新对象的一部分，新对象通过委派调用已有对象的方法达到复用功能的目的。（说白了就是将一个对象作为另一对象的属性）

# 迪米特法则

## 定义

迪米特法则的定义为：每一个软件单位对其他单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位。

## 解释

如果一个系统符合迪米特法则，那么当其中某一个模块发送修改时就会尽量少地影响其他模块，扩展会相对容易，这是对软件实体之间通信的限制，迪米特法则要求限制软件实体之间通信的宽度和深度。应用迪米特法则可以降低系统的耦合度，使类与类之间保持松散的耦合关系。

迪米特法则要求在设计系统时应该尽量减少对象之间的交互，如果两个对象之间不必彼此直接通信，那么这两个对象就不应该发生任何直接的相互作用，如果其中一个对象需要调用另一个对象的方法，可以通过“第三者”转发这个调用，即通过引入一个合理的“第三者”来降低现有对象之间的耦合度。

在将迪米特法则运用到系统设计中时要注意下面几点：在类的划分上应当尽量创建松耦合的类，类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改不会对关联的类造成太大影响；在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限；在类的设计上，只要有可能，一个类型应当设计成不变类；在对其他类的引用上，一个对象对其他对象的引用应当降到最低。