正如牛顿三大定律在经典力学中的位置一样，“开-闭”原则（Open-Closed Principle）是面向对象的可复用设计（Object Oriented Design或OOD）的基石。其他设计原则（里氏代换原则、依赖倒转原则、合成/聚合复用原则、迪米特法则、接口隔离原则）是实现“开-闭”原则的手段和工具。  
**一、“开-闭”原则（Open-Closed Principle,OCP）**  
 1.1“开-闭”原则的定义及优点  
1）定义：一个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.)。即在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展。  
2）满足“开-闭”原则的系统的优点  
a)通过扩展已有的软件系统，可以提供新的行为，以满足对软件的新需求，使变化中的软件系统有一定的适应性和灵活性。  
b)已有的软件模块，特别是最重要的抽象层模块不能再修改，这就使变化中的软件系统有一定的稳定性和延续性。  
c)这样的系统同时满足了可复用性与可维护性。  
1.2如何实现“开-闭”原则  
在面向对象设计中，不允许更改的是系统的抽象层，而允许扩展的是系统的实现层。换言之，定义一个一劳永逸的抽象设计层，允许尽可能多的行为在实现层被实现。  
解决问题关键在于抽象化，抽象化是面向对象设计的第一个核心本质。  
对一个事物抽象化，实质上是在概括归纳总结它的本质。抽象让我们抓住最最重要的东西，从更高一层去思考。这降低了思考的复杂度，我们不用同时考虑那么多的东西。换言之，我们封装了事物的本质，看不到任何细节。  
在面向对象编程中，通过抽象类及接口，规定了具体类的特征作为抽象层，相对稳定，不需更改，从而满足“对修改关闭”；而从抽象类导出的具体类可以改变系统的行为，从而满足“对扩展开放”。  
对实体进行扩展时，不必改动软件的源代码或者二进制代码。关键在于抽象。  
1.3对可变性的封装原则  
“开-闭”原则也就是“对可变性的封装原则”（Principle of Encapsulation of Variation ，EVP）。即找到一个系统的可变因素，将之封装起来。换言之，在你的设计中什么可能会发生变化，应使之成为抽象层而封装，而不是什么会导致设计改变才封装。  
 “对可变性的封装原则”意味着：  
a)一种可变性不应当散落在代码的许多角落，而应当被封装到一个对象里面。同一可变性的不同表象意味着同一个继承等级结构中的具体子类。因此，此处可以期待继承关系的出现。继承是封装变化的方法，而不仅仅是从一般的对象生成特殊的对象。  
b)一种可变性不应当与另一种可变性混合在一起。作者认为类图的继承结构如果超过两层，很可能意味着两种不同的可变性混合在了一起。  
使用“可变性封装原则”来进行设计可以使系统遵守“开-闭”原则。  
即使无法百分之百的做到“开-闭”原则，但朝这个方向努力，可以显著改善一个系统的结构。  
**二、里氏代换原则（Liskov Substitution Principle, LSP）**  
2.1概念  
定义：如果对每一个类型为T1的对象O1，都有类型为T2 的对象O2，使得以T1定义的所有程序P在所有的对象O1都代换为O2时，程序P的行为没有变化，那么类型T2是类型T1的子类型。  
即，一个软件实体如果使用的是一个基类的话，那么一定适用于其子类。而且它觉察不出基类对象和子类对象的区别。也就是说,在软件里面,把基类都替换成它的子类,程序的行为没有变化。  
反过来的代换不成立，如果一个软件实体使用的是一个子类的话，那么它不一定适用于基类。  
任何基类可以出现的地方，子类一定可以出现。  
基于契约的设计、抽象出公共部分作为抽象基类的设计。  
2.2里氏代换原则与“开-闭”原则的关系  
 实现“开-闭”原则的关键步骤是抽象化。基类与子类之间的继承关系就是抽象化的体现。因此里氏代换原则是对实现抽象化的具体步骤的规范。  
 违反里氏代换原则意味着违反了“开-闭”原则，反之未必。  
**三、 依赖倒转原则（dependence inversion principle, DIP）**  
 3.1概念  
 依赖倒转原则就是要依赖于抽象，不要依赖于实现。（Abstractions should not depend upon details. Details should depend upon abstractions.）要针对接口编程，不要针对实现编程。（Program to an interface, not an implementation.）  
也就是说应当使用接口和抽象类进行变量类型声明、参数类型声明、方法返还类型说明，以及数据类型的转换等。而不要用具体类进行变量的类型声明、参数类型声明、方法返还类型说明，以及数据类型的转换等。要保证做到这一点，一个具体类应当只实现接口和抽象类中声明过的方法，而不要给出多余的方法。  
传统的过程性系统的设计办法倾向于使高层次的模块依赖于低层次的模块，抽象层次依赖于具体层次。倒转原则就是把这个错误的依赖关系倒转过来。  
面向对象设计的重要原则是创建抽象化，并且从抽象化导出具体化，具体化给出不同的实现。继承关系就是一种从抽象化到具体化的导出。  
抽象层包含的应该是应用系统的商务逻辑和宏观的、对整个系统来说重要的战略性决定，是必然性的体现。具体层次含有的是一些次要的与实现有关的算法和逻辑，以及战术性的决定，带有相当大的偶然性选择。具体层次的代码是经常变动的，不能避免出现错误。  
从复用的角度来说，高层次的模块是应当复用的，而且是复用的重点，因为它含有一个应用系统最重要的宏观商务逻辑，是较为稳定的。而在传统的过程性设计中，复用则侧重于具体层次模块的复用。  
依赖倒转原则则是对传统的过程性设计方法的“倒转”，是高层次模块复用及其可维护性的有效规范。  
特例：对象的创建过程是违背“开—闭”原则以及依赖倒转原则的，但通过工厂模式，能很好地解决对象创建过程中的依赖倒转问题。  
3.2关系  
“开-闭”原则与依赖倒转原则是目标和手段的关系。如果说开闭原则是目标,依赖倒转原则是到达"开闭"原则的手段。如果要达到最好的"开闭"原则,就要尽量的遵守依赖倒转原则，依赖倒转原则是对"抽象化"的最好规范。  
里氏代换原则是依赖倒转原则的基础，依赖倒转原则是里氏代换原则的重要补充。  
3.3耦合（或者依赖）关系的种类：  
零耦合（Nil Coupling）关系：两个类没有耦合关系  
具体耦合（Concrete Coupling）关系：发生在两个具体的（可实例化的）类之间，经由一个类对另一个具体类的直接引用造成。  
抽象耦合（Abstract Coupling）关系：发生在一个具体类和一个抽象类（或接口）之间，使两个必须发生关系的类之间存有最大的灵活性。  
3.3.1如何把握耦合  
我们应该尽可能的避免实现继承，原因如下：  
1 失去灵活性，使用具体类会给底层的修改带来麻烦。  
2 耦合问题，耦合是指两个实体相互依赖于对方的一个量度。程序员每天都在(有意识地或者无意识地)做出影响耦合的决定：类耦合、API耦合、应用程序耦合等等。在一个用扩展的继承实现系统中，派生类是非常紧密的与基类耦合，而且这种紧密的连接可能是被不期望的。如B extends A ，当B不全用A中的所有methods时，这时候，B调用的方法可能会产生错误!  
我们必须客观的评价耦合度，系统之间不可能总是松耦合的，那样肯定什么也做不了。  
3.3.2我们决定耦合的程度的依据何在呢？  
简单的说，就是根据需求的稳定性，来决定耦合的程度。对于稳定性高的需求，不容易发生变化的需求，我们完全可以把各类设计成紧耦合的(我们虽然讨论类之间的耦合度，但其实功能块、模块、包之间的耦合度也是一样的)，因为这样可以提高效率，而且我们还可以使用一些更好的技术来提高效率或简化代码，例如c# 中的内部类技术。可是，如果需求极有可能变化，我们就需要充分的考虑类之间的耦合问题，我们可以想出各种各样的办法来降低耦合程度，但是归纳起来，不外乎增加抽象的层次来隔离不同的类，这个抽象层次可以是抽象的类、具体的类，也可以是接口，或是一组的类。我们可以用一句话来概括降低耦合度的思想："针对接口编程，而不是针对实现编程。  
在我们进行编码的时候，都会留下我们的指纹，如public的多少，代码的格式等等。我们可以耦合度量评估重新构建代码的风险。因为重新构建实际上是维护编码的一种形式，维护中遇到的那些麻烦事在重新构建时同样会遇到。我们知道在重新构建之后，最常见的随机bug大部分都是不当耦合造成的 。  
如果不稳定因素越大，它的耦合度也就越大。  
某类的不稳定因素=依赖的类个数/被依赖的类个数  
依赖的类个数＝ 在编译此类的时被编译的其它类的个数总和  
3.3.3怎样将大系统拆分成小系统  
解决这个问题的一个思路是将许多类集合成一个更高层次的单位,形成一个高内聚、低耦合的类的集合，这是我们设计过程中应该着重考虑的问题！  
耦合的目标是维护依赖的单向性，有时我们也会需要使用坏的耦合。在这种情况下，应当小心记录下原因，以帮助日后该代码的用户了解使用耦合真正的原因。  
3.4怎样做到依赖倒转？  
以抽象方式耦合是依赖倒转原则的关键。抽象耦合关系总要涉及具体类从抽象类继承，并且需要保证在任何引用到基类的地方都可以改换成其子类，因此，里氏代换原则是依赖倒转原则的基础。  
在抽象层次上的耦合虽然有灵活性，但也带来了额外的复杂性，如果一个具体类发生变化的可能性非常小，那么抽象耦合能发挥的好处便十分有限，这时可以用具体耦合反而会更好。  
层次化：所有结构良好的面向对象构架都具有清晰的层次定义，每个层次通过一个定义良好的、受控的接口向外提供一组内聚的服务。  
依赖于抽象：建议不依赖于具体类，即程序中所有的依赖关系都应该终止于抽象类或者接口。尽量做到：  
1、任何变量都不应该持有一个指向具体类的指针或者引用。  
2、任何类都不应该从具体类派生。  
3、任何方法都不应该覆写它的任何基类中的已经实现的方法。  
3.5依赖倒转原则的优缺点  
依赖倒转原则虽然很强大，但却最不容易实现。因为依赖倒转的缘故，对象的创建很可能要使用对象工厂，以避免对具体类的直接引用，此原则的使用可能还会导致产生大量的类，对不熟悉面向对象技术的工程师来说，维护这样的系统需要较好地理解面向对象设计。  
依赖倒转原则假定所有的具体类都是会变化的，这也不总是正确。有一些具体类可能是相当稳定，不会变化的，使用这个具体类实例的应用完全可以依赖于这个具体类型，而不必为此创建一个抽象类型。  
**四、合成/聚合复用原则（Composite/Aggregate Reuse Principle或CARP）**  
 4.1概念  
定义：在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分；新的对象通过向这些对象的委派达到复用这些对象的目的。  
应首先使用合成/聚合，合成/聚合则使系统灵活，其次才考虑继承，达到复用的目的。而使用继承时，要严格遵循里氏代换原则。有效地使用继承会有助于对问题的理解，降低复杂度，而滥用继承会增加系统构建、维护时的难度及系统的复杂度。  
如果两个类是“Has-a”关系应使用合成、聚合，如果是“Is-a”关系可使用继承。"Is-A"是严格的分类学意义上定义，意思是一个类是另一个类的"一种"。而"Has-A"则不同，它表示某一个角色具有某一项责任。  
4.2什么是合成？什么是聚合？  
合成（Composition）和聚合（Aggregation）都是关联（Association）的特殊种类。  
聚合表示整体和部分的关系，表示“拥有”。如奔驰S360汽车，对奔驰S360引擎、奔驰S360轮胎的关系是聚合关系，离开了奔驰S360汽车，引擎、轮胎就失去了存在的意义。在设计中, 聚合不应该频繁出现，这样会增大设计的耦合度。  
合成则是一种更强的“拥有”，部分和整体的生命周期一样。合成的新的对象完全支配其组成部分，包括它们的创建和湮灭等。一个合成关系的成分对象是不能与另一个合成关系共享的。  
 换句话说，合成是值的聚合（Aggregation by Value），而一般说的聚合是引用的聚合（Aggregation by Reference）。  
明白了合成和聚合关系，再来理解合成/聚合原则应该就清楚了，要避免在系统设计中出现，一个类的继承层次超过3层，则需考虑重构代码，或者重新设计结构。当然最好的办法就是考虑使用合成/聚合原则。  
4.3通过合成/聚合的优缺点  
优点：  
1) 新对象存取成分对象的唯一方法是通过成分对象的接口。  
2) 这种复用是黑箱复用，因为成分对象的内部细节是新对象所看不见的。  
3) 这种复用支持包装。  
4) 这种复用所需的依赖较少。  
5) 每一个新的类可以将焦点集中在一个任务上。  
6) 这种复用可以在运行时间内动态进行，新对象可以动态的引用与成分对象类型相同的对象。  
7) 作为复用手段可以应用到几乎任何环境中去。  
缺点:就是系统中会有较多的对象需要管理。  
4.4通过继承来进行复用的优缺点  
优点：  
新的实现较为容易，因为超类的大部分功能可以通过继承的关系自动进入子类。  
修改和扩展继承而来的实现较为容易。  
 缺点：  
继承复用破坏包装，因为继承将超类的实现细节暴露给子类。由于超类的内部细节常常是对于子类透明的，所以这种复用是透明的复用，又称“白箱”复用。  
如果超类发生改变，那么子类的实现也不得不发生改变。  
从超类继承而来的实现是静态的，不可能在运行时间内发生改变，没有足够的灵活性。  
继承只能在有限的环境中使用。  
**五、 迪米特法则（Law of Demeter，LoD）**  
 5.1概述  
 定义：一个软件实体应当尽可能少的与其他实体发生相互作用。  
 这样，当一个模块修改时，就会尽量少的影响其他的模块。扩展会相对容易。  
 这是对软件实体之间通信的限制。它要求限制软件实体之间通信的宽度和深度。  
5.2迪米特法则的其他表述：  
1）只与你直接的朋友们通信。  
2）不要跟“陌生人”说话。  
3）每一个软件单位对其他的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。  
5.3狭义的迪米特法则  
如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中的一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。  
朋友圈的确定  
“朋友”条件：  
1）当前对象本身（this）  
2）以参量形式传入到当前对象方法中的对象  
3）当前对象的实例变量直接引用的对象  
4）当前对象的实例变量如果是一个聚集，那么聚集中的元素也都是朋友  
5）当前对象所创建的对象  
任何一个对象，如果满足上面的条件之一，就是当前对象的“朋友”；否则就是“陌生人”。  
缺点：会在系统里造出大量的小方法，散落在系统的各个角落。  
与依赖倒转原则互补使用  
5.4狭义的迪米特法则的缺点：  
在系统里造出大量的小方法，这些方法仅仅是传递间接的调用，与系统的商务逻辑无关。  
遵循类之间的迪米特法则会是一个系统的局部设计简化，因为每一个局部都不会和远距离的对象有直接的关联。但是，这也会造成系统的不同模块之间的通信效率降低，也会使系统的不同模块之间不容易协调。  
5.5迪米特法则与设计模式  
门面（外观）模式和调停者（中介者）模式实际上就是迪米特法则的具体应用。  
5.6广义的迪米特法则  
迪米特法则的主要用意是控制信息的过载。在将迪米特法则运用到系统设计中时，要注意下面的几点：  
1）在类的划分上，应当创建有弱耦合的类。  
2）在类的结构设计上，每一个类都应当尽量降低成员的访问权限。  
3）在类的设计上，只要有可能，一个类应当设计成不变类。  
4）在对其他类的引用上，一个对象对其对象的引用应当降到最低。  
5.7广义迪米特法则在类的设计上的体现  
1）优先考虑将一个类设置成不变类  
2）尽量降低一个类的访问权限  
3）谨慎使用Serializable  
4）尽量降低成员的访问权限  
5）取代C Struct  
迪米特法则又叫作最少知识原则（Least Knowledge Principle或简写为LKP），就是说一个对象应当对其他对象有尽可能少的了解。  
5.8如何实现迪米特法则  
迪米特法则的主要用意是控制信息的过载，在将其运用到系统设计中应注意以下几点：  
1) 在类的划分上，应当创建有弱耦合的类。类之间的耦合越弱，就越有利于复用。  
2) 在类的结构设计上，每一个类都应当尽量降低成员的访问权限。一个类不应当public自己的属性，而应当提供取值和赋值的方法让外界间接访问自己的属性。  
3) 在类的设计上，只要有可能，一个类应当设计成不变类。  
4) 在对其它对象的引用上，一个类对其它对象的引用应该降到最低。  
**六、 接口隔离原则（interface separate principle, ISP）**  
 6.1概念  
 接口隔离原则：使用多个专门的接口比使用单一的总接口要好。也就是说，一个类对另外一个类的依赖性应当是建立在最小的接口上。   
这里的"接口"往往有两种不同的含义：一种是指一个类型所具有的方法特征的集合，仅仅是一种逻辑上的抽象；另外一种是指某种语言具体的"接口"定义，有严格的定义和结构。比如c# 语言里面的Interface结构。对于这两种不同的含义，ISP的表达方式以及含义都有所不同。(上面说的一个类型，可以理解成一个类，我们定义了一个类，也就是定义了一种新的类型)   
当我们把"接口"理解成一个类所提供的所有方法的特征集合的时候，这就是一种逻辑上的概念。接口的划分就直接带来类型的划分。这里，我们可以把接口理解成角色，一个接口就只是代表一个角色，每个角色都有它特定的一个接口，这里的这个原则可以叫做"角色隔离原则"。  
如果把"接口"理解成狭义的特定语言的接口，那么ISP表达的意思是说，对不同的客户端，同一个角色提供宽窄不同的接口，也就是定制服务，个性化服务。就是仅仅提供客户端需要的行为，客户端不需要的行为则隐藏起来。  
 应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口。  
 这也是对软件实体之间通信的限制。但它限制的只是通信的宽度，就是说通信要尽可能的窄。  
遵循迪米特法则和接口隔离原则，会使一个软件系统功能扩展时，修改的压力不会传到别的对象那里。  
6.2如何实现接口隔离原则  
不应该强迫用户依赖于他们不用的方法。  
1、利用委托分离接口。  
2、利用多继承分离接口。

开闭原则、依赖倒转原则、里氏替换原则、迪米特法则、接口隔离原则、CARP