面向对象设计原则有6个：开放封闭原则，单一职责原则，依赖倒置原则，liskov替换原则，迪米特法则，合成/聚合复用原则。

**单一职责原则（Single Responsibility Principle SRP）**

单一职责原则就是一个类只负责一个职责，只有一个引起变化的原因（如果这个职责发生变化，这个类也发生变化）。如果一个类承担的职责过多，等于把这些职责耦合在一起，一个职责的变化会削弱或抑制这个类完成其他职责的能力，这个耦合会导致脆弱的设计。

软件设计真正要做的许多内容，就是发现职责并把这些职责相互分离，如果能够想到多于一个动机去改变一个类，那么这个类就具有多于一个职责，就应该考虑类的分离。

**开放封闭原则（Open-Closed Principle OCP）**

讲的是设计要对扩展有好的支持，而对修改要严格限制

所谓开放封闭原则就是软件实体应该对扩展开发，而对修改封闭。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。开放封闭原则是所有面向对象原则的核心，软件设计本身所追求的目标就是封装变化，降低耦合，而开放封闭原则正是对这一目标的最直接体现。

开放封闭原则主要体现在两个方面：

对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况。

对修改封闭，意味着类一旦设计完成，就可以独立工作，而不要对类进行任何修改。

**为什么要用到开放封闭原则呢?**

1.软件需求总是变化的，世界没有一个软件是不变的，因此对软件设计人员来说，必须在不需要对原有系统进行修改的情况下，实现灵活的系统扩展。

2. 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。

3. 这样的系统同时满足了可复用性与可维护性。

**如何做到扩展开放，对修改封闭呢？**

实现开放封闭的核心就是对抽象编程，而不对具体编程，因为抽象相对稳定。让类依赖于固定的抽象，所以对修改就是封闭的；而通过面向对象的继承和多态机制，可以实现对抽象体的继承，通过覆写其方法来改变固有行为，实现新的扩展方法，所以对于扩展就是开放的，对于违反这一原则的类，必须通过重构来进行改善。常用于实现的设计模式主要有Template Method模式和Strategy模式。而封装变化，是实现这一原则的重要手段，将经常变化的状态封装为一个类。

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。

**设计建议：**

开放封闭原则，是最为重要的设计原则，liskov替换原则和合成/聚合复用原则为开放封闭原则提供保护；

可以通过Template Method模式和Strategy模式进行重构，实现对修改封闭，对扩展开放的设计思路。

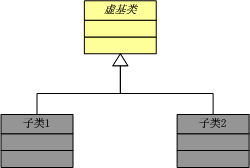
封装变化，是实现开放封闭原则的重要手段，对于经常发现变化的状态，一般将其封装为一个抽象，

拒绝滥用抽象，只将经常变化的部分进行抽象。

**里氏替换原则（Liskov Substitution Principle LSP）**

里氏替换原则是面向对象设计的基本原则之一。），里氏代换原则，很严格的原则，规则是“子类必须能够替换基类，否则不应当设计为其子类。”也就是说，子类只能去扩展基类，而不是隐藏或覆盖基类，

如有这方面需要的设计就应当参考以下两种方法替换：

   1. 

   2.http://images.cnblogs.com/cnblogs_com/cavingdeep/diagram2.gif

**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。

任何基类可以出现的地方，子类一定可以出现，LSP是继承复用的基石，只有当子类可以替换基类，软件单位的功能不受影响时，基类才能真正被复用，而子类也可以在基类的基础上增加新的行为。

Liskov提出了关于继承原则：继承必须确保超类中所拥有的性质在子类中仍然成立。（子类必须能够替换成他们的基类）

里氏替换原则讲的是基类和子类的关系，只有这种关系存在的时候里氏替换原则才能成立。里氏替换原则是实现开放封闭原则的具体规范。只是因为：实现开放封闭原则的关键是抽象，而继承关系又是抽象的一种具体实现。

**Dependence Inversion Principle（DIP），依赖倒换原则，**

“设计要依赖于抽象而不是具体化”。 要针对接口编程，不要针对实现编程。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。换句话说就是设计的时候我们要用抽象来思考，而不是一上来就开始划分我需要哪些哪些类，因为这些是具体。这样做有什么好处呢？人的思维本身实际上就是很抽象的，我们分析问题的时候不是一下子就考虑到细节，而是很抽象的将整个问题都构思出来，所以面向抽象设计是符合人的思维的。另外这个原则会很好的支持OCP，面向抽象的设计使我们能够不必太多依赖于实现，这样扩展就成为了可能，这个原则也是另一篇文章《Design by Contract》的基石。

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。

2**关系** “开-闭”原则与依赖倒转原则是目标和手段的关系**。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。

**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。

**迪米特法则**

迪米特法则（Law of Demeter）又叫最少知识原则（Least Knowledge Principle LKP）就是说一个对象应当对其他对象有尽可能少的了解，不和陌生人说话。即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】

对面向对象来说，一个软件实体应当尽可能少的与其他实体发生相互作用，每一个软件单位对其他单位都只有最少的知识，而其局限于那些于本单位密切相关的软件单位。

迪米特法则的目的在于降低类之间的耦合。对于每个类尽量减少对其他类的依赖，因此，很容易使得系统的功能模块相互独立，相互之间不存在依赖关系。应用迪米特法则有可能造成的一个后果就是，系统中存在的大量的中介类，这些类之所以存在完全是为了传递类之间的相互调用关系，------这在一定程度上增加系统的复杂度。

**迪米特法则与设计模式**

设计模式中的门面模式（Façade）和中介模式(Mediator)都是迪米特法则的应用的例子。

**狭义的迪米特法则的缺点：**

在系统里面造出大量的小方法，这些方法仅仅是传递间接，与系统的商业逻辑无关。

遵循类之间的迪特米法则会使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之间的关联。但是，这也会导致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的不容易协调。

广义的迪特米法则在类的设计上的体现：

优先考虑将一个类设置为不变类，

尽量降低一个类的访问权限；

尽量降低成员的访问权限

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。

**组合/聚集复用原则**

组合/聚合复用原则（Composite/Aggregate Reuse Principle CARP）组合和聚合都是对象建模中关联（Association）关系的一种聚合表示整体与部分的关系，表示“含有”，整体有部分组合而成，部分可以脱离整体作为一个独立的个体存在。组合则是一种更强的聚合，部分组成整体，而且不可分割，部分不能脱离整体而单独存在。在合成关系中，部分和整体的生命周期一样，组合的新的对象完全支配其组成部分，包括他们的创建和销毁。一个合成关系中成分对象是不能与另外一个合成关系共享。

组合/聚合和继承是实现复用的两个基本途径。合成复用则是指尽量使用合成/聚合，而不是使用继承

**只有当以下的条件全部被满足时，才应当使用继承关系：**

1. 子类是超类的一个特殊种类，而不是超类的一个角色，也就是区分“Has-A”和”is-A”,只有”is-A:关系才符合继承关系，”Has-A”关系应当使用聚合来描述。
2. 2永远不会出现需要将子类换成另外一个类的子类的情况，如果不能肯定将来是否会变成另外一个子类的话，就不要使用继承。
3. 子类具有扩展超类的责任，而不是具有置换掉或注销掉超类的责任，如果一个子类需要大量置换掉超类的行为，那么这个类就不应该是这个超类的子类。

错误的使用继承而不是合成/聚合的一个常见错误是把“Has-A”当成了“is-A”,“is-A”代表一个类是另外一个类的一种，而“Has-A”代表一个类是另外一个类的一个角色，而不是另外一个类的特殊种类。

**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。

**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。