1)Open-Close Principle（OCP）

**开-闭原则**，讲的是设计要对扩展有好的支持，而对修改要严格限制。这是最重要也是最为抽象的原则，通俗的说就是：软件系统中包含的各种组件，例如[模块](http://baike.baidu.com/view/22845.htm)（Modules）、[类](http://baike.baidu.com/view/2390.htm)（Classes）以及功能（Functions）等等，应该在不修改现有代码的基础上，引入新功能。开闭原则中“开”，是指对于组件功能的扩展是开放的，是允许对其进行功能扩展的；开闭原则中“闭”，是指对于原有代码的修改是封闭的，即不应该修改原有的代码。

实现开闭原则的关键就在于“抽象”。把系统的所有可能的行为抽象成一个抽象底层，这个抽象底层规定出所有的具体实现必须提供的方法的特征。作为[系统设计](http://baike.baidu.com/view/170106.htm)的抽象层，要预见所有可能的扩展，从而使得在任何扩展情况下，系统的抽象底层不需修改；同时，由于可以从抽象底层导出一个或多个新的具体实现，可以改变系统的行为，因此系统设计对扩展是开放的。

我们在软件开发的过程中，一直都是提倡需求导向的。这就要求我们在设计的时候，要非常清楚地了解用户需求，判断需求中包含的可能的变化，从而明确在什么情况下使用开闭原则。

关于系统可变的部分，还有一个更具体的对可变性封装原则（Principle of Encapsulation of Variation, EVP），它从[软件工程](http://baike.baidu.com/view/1659.htm)实现的角度对开闭原则进行了进一步的解释。EVP要求在做系统设计的时候，对系统所有可能发生变化的部分进行评估和分类，每一个可变的因素都单独进行封装。

我们在实际开发过程的设计开始阶段，就要罗列出来系统所有可能的行为，并把这些行为加入到抽象底层，根本就是不可能的，这么去做也是不经济的。因此我们应该现实的接受修改拥抱变化，使我们的代码可以对扩展开放，对修改关闭

好处：

a可复用性好。

我们可以在软件完成以后，仍然可以对软件进行扩展，加入新的功能，非常灵活。因此，这个软件系统就可以通过不断地增加新的组件，来满足不断变化的需求。

b可维护性好。

由于对于已有的软件系统的组件，特别是它的抽象底层不去修改，因此，我们不用担心软件系统中原有组件的稳定性，这就使变化中的软件系统有一定的稳定性和延续性。

2)Liskov Substituition Principle（LSP）

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。也就是说，当一个子类的实例应该能够替换任何其超类的实例时，它们之间才具有is-A关系。

3)Dependence Inversion Principle（DIP）

依赖倒置原则，“设计要依赖于抽象而不是具体化”。

A.高层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象。

B.抽象不应该依赖于具体实现，具体实现应该依赖于抽象。

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。

[面向对象](http://baike.baidu.com/view/125370.htm)的开发很好的解决了这个问题，一般情况下抽象的变化概率很小，让[用户程序](http://baike.baidu.com/view/1976812.htm)依赖于抽象，实现的细节也依赖于抽象。即使实现细节不断变动，只要抽象不变，客户程序就不需要变化。这大大降低了客户程序与实现细节的[耦合度](http://baike.baidu.com/view/1599212.htm)。

4)Interface Segregation Principle（ISP）

接口隔离原则，“将大的接口打散成多个小接口”，这样做的好处很明显。

使用多个专门的接口比使用单一的总接口要好。

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的。

一个接口代表一个角色，不应当将不同的角色都交给一个接口。没有关系的接口合并在一起，形成一个臃肿的大接口，这是对角色和接口的污染。

“不应该强迫客户依赖于它们不用的方法。接口属于客户，不属于它所在的[类层次结构](http://baike.baidu.com/subview/2405425/2405425.htm)。”这个说得很明白了，再通俗点说，不要强迫客户使用它们不用的方法，如果强迫用户使用它们不使用的方法，那么这些客户就会面临由于这些不使用的方法的改变所带来的改变。

5)Composition/Aggregation Reuse Principle（CARP）

合成/聚合复用原则，设计者首先应当考虑复合/聚合，而不是继承（因为它很直观，第一印象就是“哦，这个就是OO 啊”）。这个就是所谓的“Favor Composition over Inheritance”，在实践中复合/聚合会带来比继承更大的利益，所以要优先考虑。

6)Law of Demeter or Least Knowlegde Principle（LoD or LKP）

迪米特法则或最少知识原则，这个原则首次在Demeter系统中得到 正式运用，所以定义为迪米特法则。它讲的是“一个对象应当尽可能少的去了解其他对象”。也就是又一个关于如何松耦合（Loosely-Coupled）的法则