设计模式基础原则

1. **开闭原则**

**定义及来源**：

开闭原则（Open Closed Principle,简称 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：软件实体应当对外扩展开放，对内修改关闭 （Software entites should be open for extension，but closed for modification），这 就是开闭原则的经典定义了。

这些软件实体包括以下几个模块：

1. 项目中划分出来的模块
2. 类和接口
3. 方法

开闭原则的含义是：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，使其满足新需求。

**开闭原则的作用**

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。具体作用如下：

* 对软件测试的影响

软件遵守开闭原则话，软件测试时只需对扩展的代码进行测试即可，因为原有代码不受影响。

* + - * 可以提高代码的可复用性

粒度越小，被复用的可能性越大；在面向对象的程序设计中，根据原子和抽象编程

* 可以提高软件的可维护性

遵守开闭原则的软件，其稳定性和延续性强，从而易于扩展与维护。

**开闭原则的实现方式**

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，只要抽象合理，基本可以满足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体类来扩展就好了。

1. **里式替换原则**

**定义及来源**：

里式替换原则（liskov Substitution Principle,简称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）。

里式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用继承，什么时候不应该用继承，以及其中蕴含的原理。里式替换原则是继承复用的基础，它反映了基类与子类之间的关系，是对开闭原则的补充，是对实体抽象化的具体步骤的规范。

**里式替换的作用：**

里式替换原则的主要作用如下：

* 里式替换原则是实现开闭原则的重要方式之一。
* 它克服了继承中重写父类造成的可用性变差的缺点。
* 他是动作正确的保证。即类的扩展不会给已有的系统引进新的错误，降低了代码出错的可能性。

**里式替换原则的实现方式：**

里式替换原则通俗的来讲：子类可以扩展父类的功能，但是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，尽量不重写父类方法。

如果通过重新父类的方法来完成新功能，这样子虽然写起来简单，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，程序运行出现错误的概率会很大。

如果程序违反了里式替换原则，则继承的对象在基类出现的地方或出现运行错误。这时其修正的方法是：取消原来的继承关系，重新设计他们之间的关系。

关于里式替换原则的例子，最有名的是“正方形不是长方形”。

1. **单一职责原则**

**定义及来源**：

单一职责原则（Single Responsibility Principle,SRP）又称单一功能原则，由l罗伯特·C.马丁（Robert C. Martin）于《敏捷软件开发：原则、模式和实践》一书中提出。这里的职责是指类变化的原因，单一职责原则规定一个类应该有且只有一个引起它变化的原因，否则类应该被拆分（There should never be more than one reason for a class to change）。

**优点**：

1、降低类的复杂度。一个类只负责一项职责，其逻辑肯定比负责多项职 责简单得多。

2、提高类的可读性。复杂性降低，自然可读性上升。

3、提高系统的可维护性。可读性提高，自然更易维护。

4、变更引起的风险降低。变更是必然的，如果单一职责原则遵守的好， 当修改一个功能时，可以显著降低对其他功能的影响。

**缺点**：

1、一个职责的变化可能会削弱或者抑制这个类实现其他职责功能。

2、当客户端需要对象的某一职责时，不得不将其他的职责全部包含进 来，从而造成冗余代码或代码的浪费。

1. **依赖倒置原则**

**定义及来源**：

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++ Report 上发表的文章。

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。其核心思想是：要面向接口编程，不要面向实现编程（备注：A继承B B继承C 违反依赖倒置原则，应当共同继承A）

**依赖、倒置原则的作用：**

* 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。

**优点**：

它降低了客户与实现模块之间的耦合。

**实现**：

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。
5. **迪米特法则**

**定义及来源**：

迪米特法则（Law of Demeter，LoD）又叫作最少知识原则（Least Knowledge Principle，LKP)，产生于 1987 年美国东北大学（Northeastern University）的一个名为迪米特（Demeter）的研究项目，由伊恩·荷兰（Ian Holland）提出，被 UML 创始者之一的布奇（Booch）普及，后来又因为在经典著作《程序员修炼之道》（The Pragmatic Programmer）提及而广为人知。

只与你的直接朋友交谈，不跟“陌生人”说话（Talk only to your immediate friends and not to strangers）。其含义是：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

**优点**：

* 1. 降低了类之间的耦合度，提高了模块的相对独立性。
  2. 由于亲合度降低，从而提高了类的可复用率和系统的扩展性。

**缺点**：

增加了系统的复杂度。

**运用需注意**：

* 1. 在类的划分上，应该创建弱耦合的类。类与类之间的耦合越弱，就越有利于实现可复用的目标。
  2. 在类的结构设计上，尽量降低类成员的访问权限。
  3. 在类的设计上，优先考虑将一个类设置成不变类。
  4. 在对其他类的引用上，将引用其他对象的次数降到最低。
  5. 不暴露类的属性成员，而应该提供相应的访问器（set 和 get 方法）。
  6. 谨慎使用序列化（Serializable）功能。

**重点**：

从迪米特法则的定义和特点可知，它强调以下两点：

1. 从依赖者的角度来说，只依赖应该依赖的对象。
2. 从被依赖者的角度说，只暴露应该暴露的方法。
3. **接口隔离原则**

**来源以及定义**：

接口隔离原则（Interface Segregation Principle，ISP）要求程序员尽量将臃肿庞大的接口拆分成更小的和更具体的接口，让接口中只包含客户感兴趣的方法。

2002 年罗伯特·C.马丁给“接口隔离原则”的定义是：客户端不应该被迫依赖于它不使用的方法（Clients should not be forced to depend on methods they do not use）。该原则还有另外一个定义：一个类对另一个类的依赖应该建立在最小的接口上（The dependency of one class to another one should depend on the smallest possible interface）。

以上两个定义的含义是：要为各个类建立它们需要的专用接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。

**优点**：

1. 将臃肿庞大的接口分解为多个粒度小的接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。
2. 接口隔离提高了系统的内聚性，减少了对外交互，降低了系统的耦合性。
3. 如果接口的粒度大小定义合理，能够保证系统的稳定性；但是，如果定义过小，则会造成接口数量过多，使设计复杂化；如果定义太大，灵活性降低，无法提供定制服务，给整体项目带来无法预料的风险。
4. 使用多个专门的接口还能够体现对象的层次，因为可以通过接口的继承，实现对总接口的定义。
5. 能减少项目工程中的代码冗余。过大的大接口里面通常放置许多不用的方法，当实现这个接口的时候，被迫设计冗余的代码。

**实现方法**：

1. 接口尽量小，但是要有限度。一个接口只服务于一个子模块或业务逻辑。
2. 为依赖接口的类定制服务。只提供调用者需要的方法，屏蔽不需要的方法。
3. 了解环境，拒绝盲从。每个项目或产品都有选定的环境因素，环境不同，接口拆分的标准就不同深入了解业务逻辑。
4. 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。