# 面向对象设计原则

## 单一职责原则（SRP）

1. 任何一个软件模块都应该只对某一类行为者负责
2. 变化的方向隐含着类的责任
3. 一个类只负责一个功能领域中的相应职责

## 开放封闭原则（OCP）

1. 一个软件实体（类、模块、函数等）应该是扩展的，但是不可以修改。
2. 即软件实体应尽量在不修改原有代码的情况下进行扩展

## Liskov替换原则（LSP）

* 子类型必须能够替换它们的基类（IS-A）

## 依赖倒置原则（DIP）

* 高层模块（稳定）不应该依赖于低层模块（变化），二都都应该依赖于抽象（稳定）
* 抽象（稳定）不应该依赖于实现细节（变化），实现细节应该依赖于抽象（稳定）

## 接口隔离原则（ISP）

* 不应该强迫客户依赖于他们不用的方法。接口属于客户，不属于他所在的类层次结构。
* 不应该强迫客户依赖于它们不用的方法
* 接口应该小而完备

|  |
| --- |
| 有时，迫使接口改变的，正是它们的使用者；类原来使用了foo方法，现在客户需要它改成bar()方法时，就需要一个新类来实现它；  *不应该强迫客户依赖于它们不用的方法*  如果强迫客户程序依赖于那些它们不使用的方法，那么这些客户程序就面临着由于这些未使用方法的改变所带来的变更；这无意中导致了所有客户程序之间的耦合。换种说法，如果一个客户程序依赖于一个含有它不使用的方法的类，但是其他客户程序却要使用该方法，那么当其他客户要求这个类改变时，就会影响到这个客户程序。 我们希望尽可能地避免这种耦合，因此我们希望分离接口； |

# 构件原则

## 重用发布等价原则(REP)

重用的粒度就是发布的粒度

## 共同重用原则(CCP)

一个包中的所有类应该是共同重用的。如果重用了包中的一个类，那么就要重用包中的所有类。相互之间没有紧密联系的类不应该在同一个包中。

## 共同封闭原则(CRP)

一个包中的所有类对于同一类性质的变化应该是共同封闭的。一个变化若对一个包影响，则将对包中的所有类产生影响，而对其他的包不造成影响。

## 无依赖原则(ADP)

在包的依赖关系中不允许存在环。细节不应该被依赖。

## 稳定依赖原则(SDP)

朝着稳定的方向进行依赖

## 稳定抽象原则(SAP)

一个包的抽象程序应该和其他稳定程序一致。

# 其他原则

## 优先使用对象组合，而不是类继承

* 类继承通常为“白箱复用“，对象组合通常为”黑箱复用“
* 继承在某种程序上破坏了封闭性，子类父类耦合度过高。
* 而对象组合则只要求被组合的对象具有良好定义的外部接口，耦合度低。
* 继承关系更准确的关系是类属关系。在一个类中使用一个工具类，最好的关系使用组合。

## 封装变化点

使用封装来创建对象之间的分界层，让设计者可以在分界层的一层进行修改，而不会对另一侧产生不良的影响（稳定），从而实现层次间的松耦合。

## 针对接口编程，而不是针对实现编程

* 不将变更类型声明为某个特定的具体类，而是声明为某个接口。
* 客户程序无需获知对象的具体类型，只需要知道对象所具有的接口。
* 减少系统中各部分的依赖关系，从而实现”高内聚、松耦合”的类型设计方案。

# 通用的职责分配软件原则（模式）

GRASP原则可以帮助设计人员理解面向对象设计的本质，并以一种条理的、理性的、可解释的方式应用这些设计原则

**GRASP(General Responsibility Assignment Software Patterns)**

## 创建者：由谁来负责创建某个类的新实例（对象）

谁来创建这个对象更合理；

B包含了A；B记录了A； B和A很密切；B有一些数据，是A在实例化必须的数据；

优先使用第一、二个选项；

## 信息专家

我们设计对象(类)的时候，如果某个类拥有完成某个职责所需要的所有信息，那么这个职责就应该分配给这个类来实现。这时，这个类就是相对于这个职责的信息专家。

### 优点：

封装性：

对象充分利用自身的信息

支持低耦合

系统行为分布到不同的类

支持高内聚

## 控制器

在领域层，由谁负责首先接收并协调来自UI层的系统操作？

有点像外观控制器，如果有很多的用例，每个用例可以有一个控制器；

还有一个会谈控制器

### 优点：

1. 容易适应UI层的变化
2. 领域层代码易于重用
3. 有助于保证应用所需要的操作顺序
4. 可以对系统的状态进行推理

### 缺点:

臃肿：它做的大量的工作，导致低内聚——做事不专注，做了太多的事

解决臃肿控制器的解决方法：

1. 增加更多的控制器；
2. 采用会话控制器替换外观控制器；
3. 控制器委托任务给别的对象，而不是自己做；
4. 高内聚的理念；

## 低耦合

如何保证设计方案支持低的依赖性、低的变化影响度、增加可重用性？

高耦合很内复用；

如果一个模块（java.util）很稳定可以不考虑耦合

## 高内聚

如何使对象功能专注、可理解、可管理，同时又支持低耦合？

最佳实践

一个对象完成的功能不要太多；

这些功能都是同一类的；

例如：教授：主要任务就是教学；研究员：主要任务是科研；

## 多态

## 纯虚构

如果依据信息专家原则获得的解决方案不合适，既不想违反低耦合、高内聚，也不想违反其他的原则，该如何把职责分配给对象？

### 解决办法：

1. 把高度内聚的职责分配给虚构出来的一个类，这个类在领域模型里没有对应的概念；
2. 虚构出来的类或模块是可重用才会用纯虚构；

## 间接

若二个对象直接连接，导致耦合太紧，如何解决？

### 解决方案：

把职责分配给一个中介对象，隔离对象与其他构件或者服务，使它们不产生直接耦合；

## 隔离变化

如何设计对象、系统和子系统，使得这些成分里面的变化因素、不稳定因素不会对系统的其余部分造成意想不到的影响？

### 解决方案:

标识出能够预计的变化点或者不稳定点，职责分配的时候创建一个稳定的接口，把它们与系统的其余部分隔离开来；

### 需要注意的两种可能的变化点：

1. 变化点:在当前系统或者当前需求中已经存在了；
2. 演化点：推测的类型变化可能发生在今后，但在当前的需求中不存在；

# 总结

设计模式主要描述的是“类与相互通信的对象之间的组织关系，包括它们的角色、职责、协作方式等方面”，解决变化中的复用性问题。

接口隔离原则和单一职责都是为了提高类的内聚性、降低它们之间的耦合性，体现了封装的思想，但两者是不同的：

1. 单一职责原则注重的是职责，而接口隔离原则注重的是对接口依赖的隔离。
2. 单一职责原则主要是约束类，它针对的是程序中的实现和细节；接口隔离原则主要约束接口，主要针对抽象和程序整体框架的构建。