



NEW PLAN

# 软件高级架构师

• 一段新征程 •



| 设计原则名称                                           | 定 义                                          | 使用频率  |
|--------------------------------------------------|----------------------------------------------|-------|
| 单一职责原则<br>(Single Responsibility Principle, SRP) | 一个对象应该只包含单一的职责，并且该职责被完整地封装在一个类中              | ★★★★☆ |
| 开闭原则<br>(Open-Closed Principle, OCP)             | 软件实体应当对扩展开放，对修改关闭                            | ★★★★★ |
| 里氏代换原则<br>(Liskov Substitution Principle, LSP)   | 所有引用基类的地方必须能透明地使用其子类的对象                      | ★★★★★ |
| 依赖倒转原则<br>(Dependence Inversion Principle, DIP)  | 高层模块不应该依赖低层模块，它们都应该依赖抽象。抽象不应该依赖于细节，细节应该依赖于抽象 | ★★★★★ |
| 接口隔离原则<br>(Interface Segregation Principle, ISP) | 客户端不应该依赖那些它不需要的接口                            | ★★★☆☆ |
| 合成复用原则<br>(Composite Reuse Principle, CRP)       | 优先使用对象组合，而不是继承来达到复用的目的                       | ★★★★★ |
| 迪米特法则<br>(Law of Demeter, LoD)                   | 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   | ★★★★☆ |

## (1)单一责任原则:

- 这个原则就是让一个类只做一件事情，不要把太多的任务放在一个类里。这样做的好处是，当你需要修改某个功能时，只需要关注一个类，而不用担心影响其他功能。
- 举例：想象你正在开发一个学生管理系统。你有一个 `Student` 类，它负责存储学生的信息，比如姓名和年龄。你还有一个 `StudentManager` 类，它负责管理学生的添加、删除等操作。这样，每个类只负责一个特定的责任。
- 举例：想象你是一名学生。你每天要面对多门课程，每门课程都有不同的老师和作业。如果你把所有课程的笔记、作业和书都放在一个文件夹里，当你需要找到特定课程的资料时会变得非常混乱。相反，如果你为每门课程都准备一个专用的文件夹，你就能更轻松地管理和找到所需的信息。每个文件夹就代表了一个类，它们只负责一个特定的任务，即存储与该课程相关的资料。

## (2) 开放封闭原则:

- 对修改封闭，对拓展开放。这个原则意味着你可以扩展现有的代码，但不需要修改已有的代码。你应该允许新功能的添加，而不会影响到已经运行良好的功能。
- 举例：假设你正在编写一个图形绘制软件，你有一个 `Shape` 类，代表各种形状。现在，你想添加一个新的形状，比如三角形。你应该能够通过创建一个新的类（例如 `Triangle` 类），而不是修改已有的 `Shape` 类。
- 举例：想象你是一名家庭主妇，你正在准备一顿丰盛的晚餐。你已经在规划中有一些菜肴，但客人可能会有特殊的饮食要求。你可以轻松地加入一个新的菜肴或调整配方，而不会影响到你已经准备好的菜肴。这就是开放封闭原则，你的晚餐计划是“封闭”的，因为已经准备好了，但你可以“开放”地添加新的菜肴，以满足不同的需求。

### (3)里氏替换原则:

- 这个原则强调子类应该能够替换父类而不会影响程序的正确性。换句话说，你应该能够使用子类的实例来替代父类的实例，而不引发错误。
- 举例：假设我们有一个银行账户的基类BankAccount，它有一个方法deposit(double amount)用于存款。现在我们有一个子类CheckingAccount，它继承自BankAccount，并且增加了一些特性，比如允许透支。在这个例子中，CheckingAccount类遵守了里氏替换原则。因为子类可以直接使用父类的存款方法而不引起任何的问题。

#### (4)依赖倒置原则:

- 这个原则强调细节应该依赖于抽象，而不是相反。高层模块不应该直接依赖于低层模块的细节，而应该通过抽象进行交互。
- 举例：假设你正在开发一个电子商务平台。你有一个 `OrderProcessor` 类负责处理订单。而这个类不应该直接依赖于具体的支付方式，而是依赖于一个抽象的 `PaymentGateway` 接口。这样，你可以轻松地更改支付方式，而不必修改 `OrderProcessor`。
- 举例：想象你是一名旅行者，你需要租一辆车去探索一个城市。你不需要亲自去了解车子的每个零件如何工作，你只需要知道如何使用它们。租车公司为你提供了一辆可用的车，而不是让你去修理引擎或更换轮胎。在这个例子中，你是高层模块，租车公司是低层模块，你依赖于租车公司提供的抽象服务，而不是直接与车辆细节打交道。

### (5)接口分离原则:

- 这个原则强调客户端不应该被强制依赖它们不需要的的方法。接口应该只包含客户端需要的方法，避免造成冗余和不必要的复杂性。
- 举例：想象你正在设计一个媒体播放器。你应该根据功能拆分成不同的接口，如 `AudioPlayer` 和 `VideoPlayer`。这样，如果你只需要一个音频播放器，你就不会被迫实现视频播放相关的方法，从而遵循了接口分离原则。
- 举例：假设你正在考虑加入一个运动俱乐部。你有多个选项可供选择，如游泳、篮球和瑜伽。不同的人有不同的兴趣，你可能只想参加其中一种活动。运动俱乐部应该将这些活动分开成不同的项目，以便每个人只关注他们感兴趣的部分。这样，你不需要强制自己参加所有的活动，而是可以选择与你有兴趣的活动接口。



T H E E N D

功不唐捐，玉汝于成！

• 开 启 新 征 程 •

