学习新技术的方法

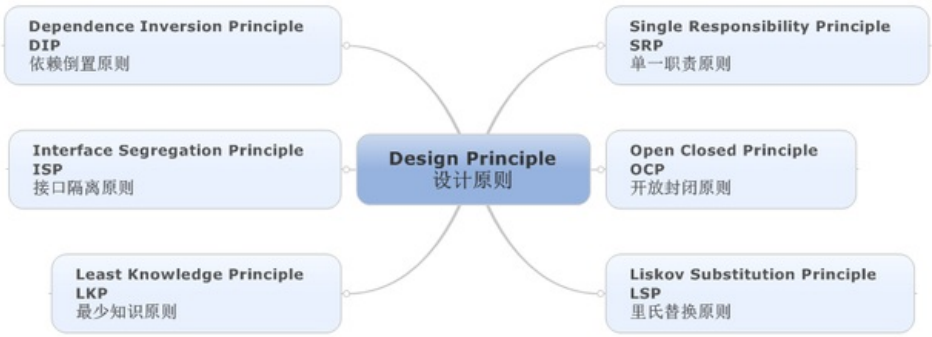
学习新技术的方法随笔：

1. 能用-->会用-->灵活运用-->精通

一个新的技术进入我们的视野，首先我们要清楚该技术的基本的概念和知识其中包括：

2. 六大设计原则

先看一幅图吧：



这幅图清晰地表达了六大设计原则，但仅限于它们叫什么名字而已，它们具体是什么意思呢？下面我将从原文、译文、理解、应用，这四个方面分别进行阐述。

1. 单一职责原则（Single Responsibility Principle - SRP）

原文：There should never be more than one reason for a class to change.

译文：永远不应该有多于一个原因来改变某个类。

理解：对于一个类而言，应该仅有一个引起它变化的原因。说白了就是，不同的类具备不同的职责，各施其责。这就好比一个团队，大家分工协作，互不影响，各做各的事情。

应用：当我们做系统设计时，如果发现有一个类拥有了两种的职责，那就问自己一个问题：可以将这个类分成两个类吗？如果真的有必要，那就分吧。千万不要让一个类干的事情太多！

2. 开放封闭原则（Open Closed Principle - OCP）

原文：Software entities like classes, modules and functions should be open for extension but closed for modifications.

译文：软件实体，如：类、模块与函数，对于扩展应该是开放的，但对于修改应该是封闭的。

理解：简言之，对扩展开放，对修改封闭。换句话说，可以去扩展类，但不要去修改类。

应用：当需求有改动，要修改代码了，此时您要做的是，尽量用继承或组合的方式来扩展类的功能，而不是直接修改类的代码。当然，如果能够确保对整体架构不会产生任何影响，那么也没必要搞得那么复杂了，直接改这个类吧。

3. 里氏替换原则（Liskov Substitution Principle - LSP）

原文：Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

译文：使用基类的指针或引用的函数，必须是在不知情的情况下，能够使用派生类的对象。

理解：父类能够替换子类，但子类不一定能替换父类。也就是说，在代码中可以将父类全部替换为子类，程序不会报错，也不会在运行时出现任何异常，但反过来却不一定成立。

应用：在继承类时，务必重写（Override）父类中所有的方法，尤其需要注意父类的 protected 方法（它们往往是让您重写的），子类尽量不要暴露自己的 public 方法供外界调用。

该原则由麻省理工学院的 Barbara Liskov 女士提出，她是美国第一位获取计算机博士学位的女性，曾经也获得过计算机图灵奖。

4. 最少知识原则（Least Knowledge Principle - LKP）

原文：Only talk to you immediate friends.

译文：只与你最直接的朋友交流。

理解：尽量减少对象之间的交互，从而减小类之间的耦合。简言之，一定要做到：低耦合，高内聚。

应用：在做系统设计时，不要让一个类依赖于太多的其他类，需尽量减小依赖关系，否则，您死都不知道自己怎么死的。

该原则也称为“迪米特法则（Law of Demeter）”，由 Ian Holland 提出。这个人不太愿意和陌生人说话，只和他走得最近的朋友们交流。

5. 接口隔离原则（Interface Segregation Principle - ISP）

   原文：The dependency of one class to another one should depend on the smallest possible interface.

   译文：一个类与另一个类之间的依赖性，应该依赖于尽可能小的接口。

   理解：不要对外暴露没有实际意义的接口。也就是说，接口是给别人调用的，那就不要去为难别人了，尽可能保证接口的实用性吧。她好，我也好。

   应用：当需要对外暴露接口时，需要再三斟酌，如果真的没有必要对外提供的，就删了吧。一旦您提供了，就意味着，您将来要多做一件事情，何苦要给自己找事做呢。

6. 依赖倒置原则（Dependence Inversion Principle - DIP）

   原文：High level modules should not depends upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.

   译文：高层模块不应该依赖于低层模块，它们应该依赖于抽象。抽象不应该依赖于细节，细节应该依赖于抽象。

   理解：应该面向接口编程，不应该面向实现类编程。面向实现类编程，相当于就是论事，那是正向依赖（正常人思维）；面向接口编程，相当于通过事物表象来看本质，那是反向依赖，即依赖倒置（程序员思维）。

   应用：并不是说，所有的类都要有一个对应的接口，而是说，如果有接口，那就尽量使用接口来编程吧。

将以上六大原则的英文首字母拼在一起就是 SOLID（稳定的），所以也称之为 SOLID 原则。

只有满足了这六大原则，才能设计出稳定的软件架构！但它们毕竟只是原则，只是四人帮给我们的建议，有些时候我们还是要学会灵活应变，千万不要生搬硬套，否则只会把简单问题复杂化，切记！

- 补充设计原则

1. 组合/聚合复用原则（Composition/Aggregation Reuse Principle - CARP）

   当要扩展类的功能时，优先考虑使用组合，而不是继承。这条原则在 23 种经典设计模式中频繁使用，如：代理模式、装饰模式、适配器模式等。可见江湖地位非常之高！

2. 无环依赖原则（Acyclic Dependencies Principle - ADP）

   当 A 模块依赖于 B 模块，B 模块依赖于 C 模块，C 依赖于 A 模块，此时将出现循环依赖。在设计中应该避免这个问题，可通过引入“中介者模式”解决该问题。

3. 共同封装原则（Common Closure Principle - CCP）

   应该将易变的类放在同一个包里，将变化隔离出来。该原则是“开放-封闭原则”的延生。

4. 共同重用原则（Common Reuse Principle - CRP）

   如果重用了包中的一个类，那么也就相当于重用了包中的所有类，我们要尽可能减小包的大小。

5. 好莱坞原则（Hollywood Principle - HP）

   好莱坞明星的经纪人一般都很忙，他们不想被打扰，往往会说：Don't call me, I'll call you. 翻译为：不要联系我，我会联系你。对应于软件设计而言，最著名的就是“控制反转”（或称为“依赖注入”），我们不需要在代码中主动的创建对象，而是由容器帮我们来创建并管理这些对象。

- 其他设计原则

1. 不要重复你自己（Don't repeat yourself - DRY）

   不要让重复的代码到处都是，要让它们足够的重用，所以要尽可能地封装。

2. 保持它简单与傻瓜（Keep it simple and stupid - KISS）

   不要让系统变得复杂，界面简洁，功能实用，操作方便，要让它足够的简单，足够的傻瓜。

3. 高内聚与低耦合（High Cohesion and Low Coupling - HCLC）

   模块内部需要做到内聚度高，模块之间需要做到耦合度低。

4. 惯例优于配置（Convention over Configuration - COC）

   尽量让惯例来减少配置，这样才能提高开发效率，尽量做到“零配置”。很多开发框架都是这样做的。

5. 命令查询分离（Command Query Separation - CQS）

   在定义接口时，要做到哪些是命令，哪些是查询，要将它们分离，而不要揉到一起。

6. 关注点分离（Separation of Concerns - SOC）

   将一个复杂的问题分离为多个简单的问题，然后逐个解决这些简单的问题，那么这个复杂的问题就解决了。难就难在如何进行分离。

7. 契约式设计（Design by Contract - DBC）

   模块或系统之间的交互，都是基于契约（接口或抽象）的，而不要依赖于具体实现。该原则建议我们要面向契约编程。

8. 你不需要它（You aren't gonna need it - YAGNI）

   不要一开始就把系统设计得非常复杂，不要陷入“过度设计”的深渊。应该让系统足够的简单，而却又不失扩展性，这是其中的难点。