1. 单一职责原则SRP

书目：大话设计模式

类和类之间的关系：继承 实现 依赖(作为方法参数，是比较弱的关系只有调方法时才依赖) 关联(作为成员变量 比依赖更强 一对一 一对多(List) 多(List)对多(List)) 聚合 组合 语法上都和关联一样 都是一对多的关联关系 聚合(整体与部分 两者生命周期不同) 组合(整体和部分的生命周期是相同的 人和他的四肢)

适配器模式：将一个类的接口转换成客户希望的另外一个接口。解决的问题是：使得原本由于接口不兼容而不能一起工作的那些类可以在一起工作。例子，字节流的System.in，不能跟字符流的BufferedReader一起工作，但适配器InputStreamReader就可以将他们的功能整合起来；

三方：适配者；适配器；目标接口

类适配器 继承适配者类 因为是继承关系，静态的

对象适配器 关联适配者实例 动态的，可以传适配者类的子类

1、概念

单一职责原则 SRP --- Single Responsibility Principle

There should never be more than one reason for a class to change。

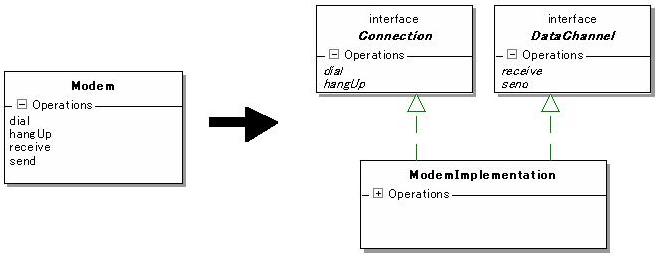
应该有且仅有一个原因引起类的变更

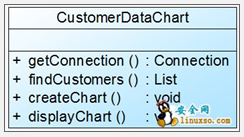
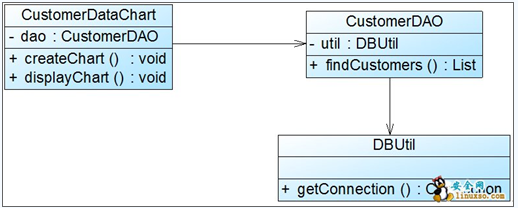
* 系统中的每一个类都应该只有一个职责，而所有类所关注的就是自身职责的完成
* 职责是指为“变化的原因”
* 如果能想到多个原因去改变一个类，这个类就具有多个职责
* 并不是单一功能原则，并不是每个类只能有一个方法，而是单一“变化的原因”原则
* 如果一个类有多个职责，这些职责就耦合在了一起，当一个职责发生变化时，可能会影响其它的职责
* 多个职责耦合在一起，会影响复用性（可能只需要复用该类某一个职责，但该职责跟其它职责耦合在一起，很难分离出来）

2、好处

* 单一职责原则的意思就是经常说的“高内聚、低耦合”

3、举例



4、总结

* 单一职能原则是所有原则中最简单的、最难应用的一个；要注意过犹不及
* “变化的原因”，只有实际发生时才有意义。可能预测到会有多个原因引起这个类的变化，但这仅仅是预测，并没有真的发生，这个类仍可看做具有单一职责，不需要分离
* JavaEE中的分层框架模式实际上体现了单一职责原则

2、开闭原则OCP

1、概念

开闭原则OCP--Open Closed Principle

Software entities should be open for extension,but closed for modification。

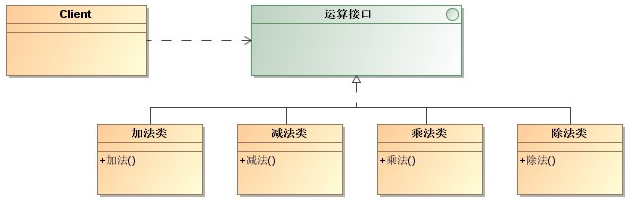
软件实体应当对扩展开放，对修改关闭。更通俗翻译：软件系统中的各种组件，如模块（Modules）、类（Classes）以及功能（Functions）等，应该在不修改现有代码的基础上，引入新功能。

* 实现开闭原则的关键是抽象
* 定义一个抽象层，只规定功能而不提供实现，实现通过定义具体类来完成
* 当需求变化时，不是通过修改抽象层来完成，而是通过定义抽象层的新实现(扩展)完成
* 通过抽象类及接口，规定了具体类的特征作为抽象层，相对稳定，不需修改，从而满足“对修改关闭”；从抽象类导出的具体类可以改变系统的行为，从而满足“对扩展开放”

2、好处

* 通过扩展已有软件系统，可提供新的行为，以满足对软件的新需求，提高了软件系统的适应性和灵活性
* 已有的软件模块，特别是最重要的抽象层模块不能再修改，提高了软件系统的一定的稳定性和延续性
* 这样的设计同时也满足了可复用性与可维护性

3、举例



运算器接口 interface calculator{

Calculate(Number num1,Number num2);

}

4、总结

* 实现开闭原则的关键是抽象
* 抽象层相对稳定，不需修改，需求变化后通过重新定义抽象层的新实现来完成
* 即使无法百分之百的做到开闭原则，但朝这个方向努力，可以显著改善一个系统的结构
* 对系统每个部分都肆意地进行抽象也不是一个好主意，应该仅仅对程序中需求频繁变化部分进行抽象。拒绝不成熟的抽象和抽象本身一样重要
* 开闭原则具有理想主义的色彩，它是面向对象设计的终极目标。其他设计原则都可以看作是开闭原则的实现手段或方法
* 软件系统的构建是一个需要不断重构的过程，100%满足开闭原则的软件系统是相当困难的，这就是开闭原则的相对性。但在设计过程中，通过对模块功能的抽象（接口定义），模块之间的关系的抽象（通过接口调用），抽象与实现的分离（面向接口的程序设计）等，可以尽量接近满足开闭原则

3、里氏替代原则LSP

1、概念

里氏替代原则LSP--Liskov Substitution Principle

functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.  
所有引用基类的地方必须能透明地使用其子类的对象。

通俗点讲只要父类能出现的地方子类就可以出现，而且调用子类还不产生任何的错误或异常，调用者可能根本就不需要知道是父类还是子类。但是反过来就不成了，有子类出现的地方，父类未必就能适应。

里氏替换法则包含了四层意思：

1. 子类必须完全实现父类的方法

2. 子类可以有自己的个性

3. 覆盖和实现父类的方法时，输入参数可以被放大，但不能被缩小

4. 覆盖和实现父类的方法时，输出结果可以被缩小，但不能被放大

2、好处

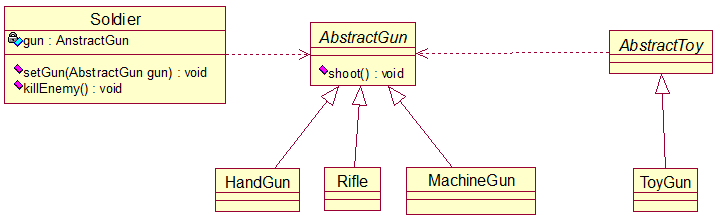
* 而里氏代换原则对如何良好继承提出了衡量依据

3、举例

1、正方形是长方形吗？

2、企鹅/鸵鸟是鸟吗？

3、玩具手枪是手枪吗？



建议：如果子类不能完整实现父类的方法，或者是父类的某些方法在子类中已经发生“畸变”，那么建议断开父子继承关系，采用依赖、聚集、组合等关系代替继承。

4、Q:请从里氏替换原则的角度考察java.util库中的Properties与Hashtable关系是否合适。

A：从LSP的角度来看，Properties与Hashtable的关系是不合适的。Properties是一种特殊的Hashtable，它只接受String类型的键(Key)和值(Value)。但是，其超类型则可以接受任何类型的键和值。这就意味着，在一些需要非String类型的键和值的地方，Properties不能够取代Hashtable。这是一个Java语言API本身违反LSP原则的反面教材。

4、总结

* 采用开闭原则必然用到抽象和多态，而这离不开继承。而里氏代换原则对如何良好继承提出了衡量依据。里氏代换原则是使代码符合开闭原则的一个重要保证。

4、依赖倒置原则DIP

1、概念

依赖倒置原则DIP --Dependence Inversion Principle

High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions。翻译过来，包含三层含义：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。

抽象：即抽象类或接口，两者是不能够实例化的。

细节：即具体的实现类，实现接口或者继承抽象类的类，可通过关键字new直接被实例化。

依赖正置就是类间的依赖是实实在在的实现类间的依赖，也就是面向实现编程

依赖倒置原则的本质其实就是通过抽象（抽象类或接口）使各个类或模块的实现彼此独立，不相互影响，实现模块间的松耦合

依赖倒置有三种方式来实现

通过构造函数传递依赖对象；通过setter方法传递依赖对象；接口声明实现依赖对象

依赖倒置原则的几大规则

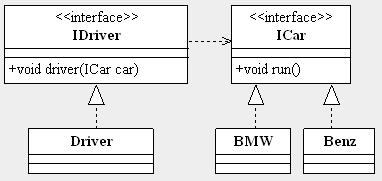
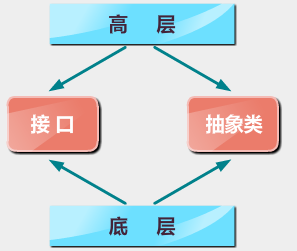
* 每个类尽量都有接口和抽象类，或者抽象类和接口都有。
* 变量的表面类型尽量是接口或者是抽象类。
* 任何类都不应该从具体类派生。（做二次开发时，无法获得高层代码的时候例外）
* 尽量不要覆写基类已经实现好的方法。

2、好处

* 采用依赖倒置原则可以减少类间的耦合性，提高系统的稳定性，减少并行开发引起的风险，提高代码的可读性和可维护性。

3、举例

* 早期电脑所有硬件整合在一起，一个模块坏全部坏，现在的电脑依赖于插槽（规范），更换cpu、内存、卡等方便。

4、总结

* 依赖倒置原则核心就是要面向接口编程，理解了面向接口编程，也就理解了依赖倒置
* 如果没有实现依赖倒置原则，那么也就意味着开闭原则也无法实现。
* 结合实际情况使用此原则，要考虑生产和成本，不能生搬硬套

5、接口分离原则ISP

1、概念

接口分离原则ISP-- Interface Segregation Principle

有两种定义

第一种：Clients should not be forced to depend upon interfaces that they don't use.客户端不应该强行以来它不需要的接口

第二种：The dependency of one class to another one should depend on the smallest possible interface.类间的依赖关系应该建立在最小的接口上

接口隔离原则的含义：

建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，要为各个类建立专用接口，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。

在使用接口隔离原则的时候需要有一些规范：

1、接口尽量小.保证一个接口只服务一个子模块或者业务逻辑

2、接口高内聚。接口高内聚是对内高度依赖，对外尽可能隔离。

3. 定制服务。为调用者提供且只提供他需要的方法

4、接口设计有限度。根据业务及经验，仔细思考筹划，合理适度隔离接口

接口隔离原则和单一职能原则辨析

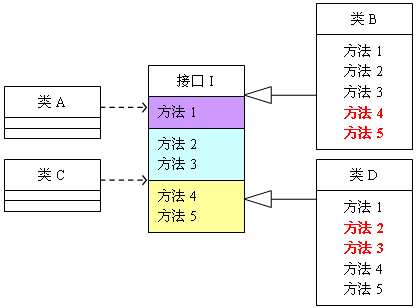
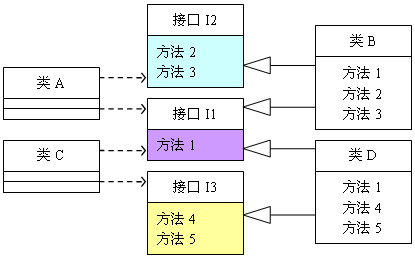
* 单一职责原则注重的是职责；而接口隔离原则注重对接口依赖的隔离。
* 单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口接口，主要针对抽象，针对程序整体框架的构建。

2、好处

* 防止庞大、臃肿的接口，避免“接口污染”，提高灵活性和可维护性

3、举例

* 在车站售票窗口排队的人有买票的，有查开车信息，有退票的，不必排在同一窗口中。多开几窗口，每个窗不同功能，让不同需求的人排在不同窗口，可以节约时间和人力。

4、总结

* 注意控制接口的粒度，接口不能太小，如果太小会导致系统中接口泛滥，不利于维护；
* 接口也不能太大，太大的接口将违背接口隔离原则，灵活性较差，使用起来很不方便。
* 一般而言，接口中仅包含为某一类用户定制的方法即可，不应该强迫客户依赖于那些它们不用的方法。

6、迪米特原则LOD

1、概念

迪米特原则LOD-- Law of Demeter

talk only to your immediate friends

只与你直接的朋友通信（不跟陌生人说话，朋友越少越好）

如果两个类不必彼此直接通信,那么这两个类就不应当发生直接的相互作用.如果其中一个类需要调用另一个类的方法的话,可以通过第三者转发这个调用.

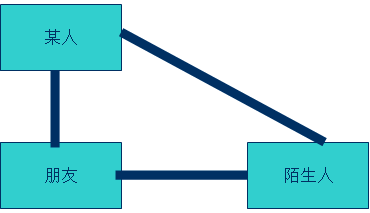
迪米特法则的初衷在于降低类之间的耦合。由于每个类尽量减少对其他类的依赖，因此，很容易使得系统的功能模块功能独立，相互之间不存在（或很少有）依赖关系。

应用迪米特法则有可能造成的一个后果就是：系统中存在大量的中介类，这些类之所以存在完全是为了传递类之间的相互调用关系——这在一定程度上增加了系统的复杂度。

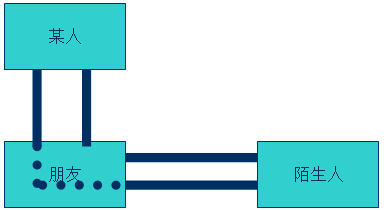
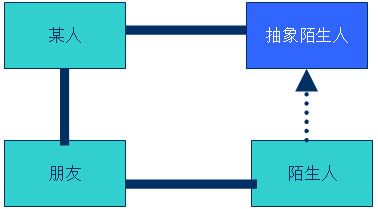
2、好处

* 尽量降低类与类之间的耦合

3、举例



不符合LOD

符合LOD 优化的LOD



4、总结

* 过分的使用迪米特原则，会产生大量这样的中介和传递类，导致系统复杂度变大。所以在采用迪米特法则时要反复权衡，既做到结构清晰，又要高内聚低耦合。
* 为了克服狭义迪米特法则的缺点,可以使用依赖倒转原则,引入一个抽象的类型引用"抽象陌生人"对象,使"某人"依赖于"抽象陌生人",换言之,就是将"抽象陌生人"变成朋友.
* 外观模式和中介者模式都是迪米特法则的应用

7、合成/聚合复用原则CARP

1、概念

合成/聚合复用原则CARP-- Composite Aggregate Reuse Principle

Favor object composition over class inheritance.

优先使用对象组合，而不是类继承.（要尽量使用合成和聚合，尽量不要使用继承）

合成聚合复用原则是指在一个新对象中通过关联关系（组合和聚合关系）使用原来已经存在的一些对象，是之成为新对象的一部分，新的对象通过向这些原来已经具有的对象委派相应的动作或者命令达到复用已有功能的目的。

为何“要尽量使用合成和聚合，尽量不要使用继承”呢？这是因为：

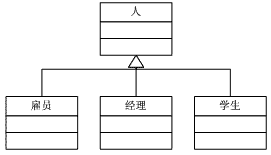
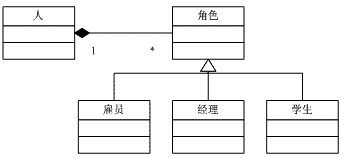
1. 继承复用破坏包装，它把超类的实现细节直接暴露给子类，这违背了信息隐藏的原则；
2. 第二：如果超类发生了改变，那么子类也要发生相应的改变，这就直接导致了类与类之间的高耦合，不利于类的扩展、复用、维护等，也带来了系统僵硬和脆弱的设计。
3. 第三：从超类继承而来的实现是静态的，不可能再运行时间内发生改变，因此没有足够的灵活性。

而是用合成和聚合的时候新对象和已有对象的交互往往是通过接口或者抽象类进行的，就可以很好的避免上面的不足，而且这也可以让每一个新的类专注于实现自己的任务，符合单一职责原则。

2、好处

* 非常有利于构建可维护、可复用、可扩展和灵活性好的软件系统。

3、举例

4、总结

* 组合与继承都是重要的复用方法。组合称为黑箱复用，继承称为白箱复用。
* 在OO开发的早期，继承被过度地使用
* 随着时间的发展，人们发现优先使用组合可以获得复用性与简单性更佳的设计
* 并非不要使用继承，并非继承一无是处，而是不要滥用继承。合成/聚合也有自己的缺点
* 优先使用对象组合，而不是类继承。使用继承必须符合里氏替代原则

|  |  |
| --- | --- |
| 单一职责原则 | ★★★★ |
| 开闭原则 | ★★★★★ |
| 里氏代换原则 | ★★★★ |
| 依赖倒置原则 | ★★★★★ |
| 接口分离原则 | ★★★ |
| 迪米特原则 | ★★ |
| 组合/聚合复用原则 | ★★★★ |