设计模式之SOLID原则再回首

本科阶段学过设计模式，那时对设计模式的五大原则——SOLID原则的概念与理解还是比较模糊，此时过去了2年时间，在学习《高级软件工程》课程中老师又提到了设计模式，课程中还详细讨论了五大原则的过程，这次SOLID原则再回首作者提出了一些更通俗的理解吧~

一. 什么是设计模式?

那么,什么是设计模式呢?

从广义角度讲设计模式是可解决一类软件问题并能重复使用的设计方案;

从狭义角度讲设计模式是对被用来在特定场景下解决一般设计问题的类和相互通信的对象的描述,是在类和对象的层次描述的可重复使用的软件设计问题的解决方案.

模式体现的是程序整体的构思,也会出现在分析或者是概要设计阶段,包括创建型模式、结构型模式和行为型模式.

模式的核心思想是通过增加抽象层,把变化部分从那些不变部分里分离出来.

模式的四大基本要素包括:

1.模式名称（Pattern Name）

2.问题（Problem）：描述应该在何时使用模式,解释了设计问题和问题存在的前因后果,可能还描述模式必须满足的先决条件

3.解决方案（Solution）：描述了设计的组成成分、相互关系及各自的职责和协作方式.模式就像一个模板,可应用于多种场合,所以解决方案并不描述一个具体的设计或实现,而是提供设计问题的抽象描述和解决问题所采用的元素组合（类和对象）.

4.效果（Consequences）：描述模式的应用效果及使用模式应权衡的问题.

其中设计模式的SOLID原则(Principles)如下:

单一职责原则（Single Responsibility）
开闭原则（Open Closed）
里氏代换原则（Liskov Substitution）
接口隔离原原则（Interface Segregation）
依赖倒置原则（Dependency Inversion）

设计模式就是实现了上述原则,从而达到代码复用、增加可维护性的目的.

二. 单一职责原则

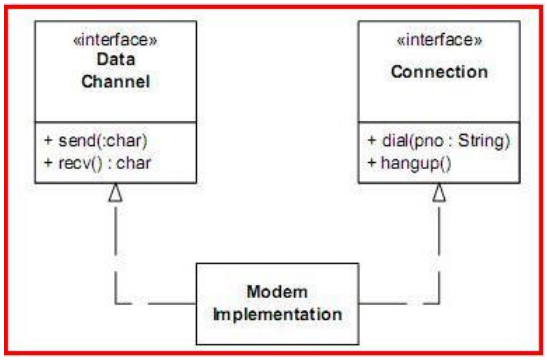
单一职责原则(Single Responsibility Principle,SRP)思想:就一个类而言,应仅有一个引起它变化的原因(一个类只有一个职责);每一个引起类变化的原因就是一个职责,当类具有多职责时,应该把多余职责分离出去,分别创建类来完成.

每一个职责都是一个变化的轴线,当需求变化时会反映为类的职责的变化.

例如Modem可以链接dial(拨号连接)\hangup(挂断拨号)和send(发送数据)\recv(接收数据).

```
interface Modem {
public void dial(String pno);
public void hangup();
public void send(char c);
public char recv();
}
```

Modem类有两个职责,链接管理和数据通信,应该将它们分离.一个类中可以有多个方法,但是一个类只干一件事.



就我而言,在做很多项目时都喜欢在一个类中去增加各种各样的功能,C#窗体应用程序中会在Form类中增加各式各样的代码,当需求变更时,总需要更改这个窗体类,代码越来越长,维护很麻烦且灵活性很低.

正如《大话设计模式》中所说: "如果一个类承担的职责过多,就等于把这些职责耦合在一起,一个职责的变化可能会削弱或抑制这个类完成其他职责的能力.这种耦合会导致脆弱的设计,当变化发生时,设计会遭受到意想不到的破坏." 而且在软件设计过程中需要完成的内容非常多,发现职责并把那些职责相互分离还是有难度的.

例子理解:

比如C#Winform设计俄罗斯方块游戏时,就需要把界面变化(方块绘制\擦除)和游戏逻辑(方块下落\旋转\碰撞\移动等)分离,至少应该讲程序分为两个类——游戏逻辑类和WinForm窗体界面类.当需要改变界面时,只需修改窗体类,和游戏逻辑无关,而且游戏逻辑是不太容易变化的,将它们分离有利于界面的修改,从而达到代码复用的目的.

编程过程中我们要在类的职责分离上多思考,做到单一职责,这样你的代码才是真正的易维护、易扩展、易复用、灵活多样.

三. 开闭原则

开闭原则(Open Closed Principle,OCP)思想:对功能扩展是开放的(增加新功能),但对修改原功能代码是关闭的.在进行扩展(类\模块\函数都可以扩展)时,不需要对原来的程序进行修改,即已有东西不变,可添加新功能.

它的好处是灵活可用,可加入新功能,新模块满足不断变化的新需求,由于不修改软件原来的模块,不用担心软件的稳定性.

主要原则包括:

1.抽象原则

把系统的所有可能的行为抽象成一个底层,由于可从抽象层导出多个具体类来改变系统行为,因此对于可变部分系统设计对扩展是开放的.

2.可变性封装原则

对系统所有可能发生变化的部分进行评估和分类,每个可变的因素都单独进行封装.

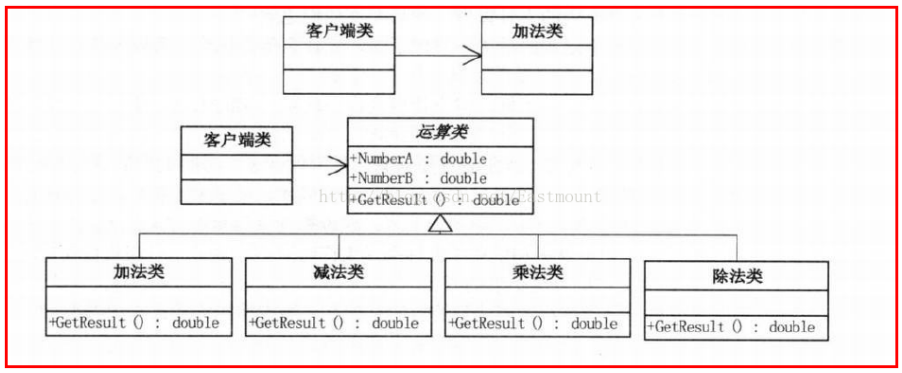
正如《大话设计模式》中所说:"我们在做任何系统时,需求都会存在一定的变化,那么如何面对需求变化时,设计软件可以相对容易修改,而不是把整个项目推倒重来.开闭原则可以让设计面对需求变化时保持相对稳定,从而使得系统在第一个版本后不断推出新的版本."

但是无论模块怎么封装,都会存在一些无法对之封装的变化,既然不可能完全封装,设计人员必须对设计的模块应该对哪种变化封装做出选择,他需要通过猜测最有可能发生的变化种类,然后构造抽象来隔离那些变化.通常设计人员会在发生小变化时,就及早去想办法应对发生更大变化的可能,即等到变化时立即采取行动.当变化发生时就创建抽象来隔离以后发生同类变化.

例子理解:(源自<<大话设计模式>>)

香港回归采用"一国两制"的体制就是开闭原则的代表例子,社会主义制度是不可更改,但可以增加新的制度实现共和.再如一个计算器的例子,原来客户端只有加法运算,现在需要添加新的功能减法,此时如果你去修改原来类就会违背"开放-封闭原则",而且当添加很多新的运算时,这样的代码很难维护.

此时你需要重构程序,增加一个抽象类的运算类,通过继承或多态等隔离具体加法、减法与client耦合,面对需求变化,对程序的改动是通过增加新代码进行的,而不是更改现有的代码,这就是开闭原则的精髓.



PS:想到以前图像处理软件,每学习一章新处理如图像灰度、增强、采样、二值化等,就在原来的基础上不断增加新的函数实现,而且最后一个.cpp文件4000多行,简直无法直视啊！可能还是我实际项目做得比较少的原因,拿到一个项目如何去规划、设计的理解依然不够,等工作几年后我再回来写一些实际经验的设计思想文章吧,拭目以待~

切记:开闭原则是面向对象设计的核心所在,这个原则可以带来面向对象技术所声称的巨大好处,即可维护、可扩展、可服用、灵活性好,开发人员对程序中呈现出频繁变化的那些部分作出抽象,然而对于应用程序中的每个部分都刻意地进行抽象同样不是一个好主意,拒绝不成熟的抽象和抽象本身一样重要.

四. 里氏替换原则

里氏替换原则(Liskov Substitution Principle,LSP)思想:继承必须确保父类所拥有的性质在子类中仍然成立,当一个子类的实例能够替换任何其父类的实例时,它们之间才具有is-a-kind-of-A关系.只有当一个子类(派生类)可以替换掉其父类(基类)而软件功能不受影响时,基类才能被复用,派生类也才能在基类的基础上增加新的行为.

其本质是在同一个继承体系中的对象应该有共同的行为特征.

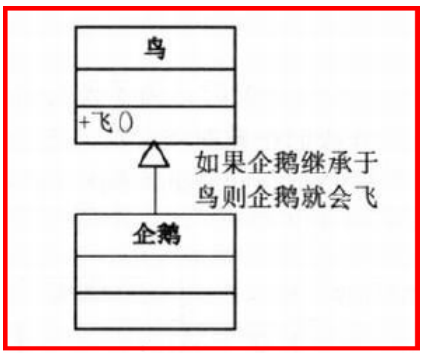
换句话说,一个软件实体如果使用的是一个父类的话,那么一定适用于其子类,而且觉察不出父类对象和子类对象的区别,在软件中把父类都替换成它的子类,程序的行为没有变化,子类必须能够替换掉它们的父类型.

例子理解:

父类为猫,子类有黑猫和白猫,如果一个方法使用于猫如捉老鼠,则必然使用于黑猫和白猫.再如<<大话设计模式>>中的问题“企鹅是鸟吗?”

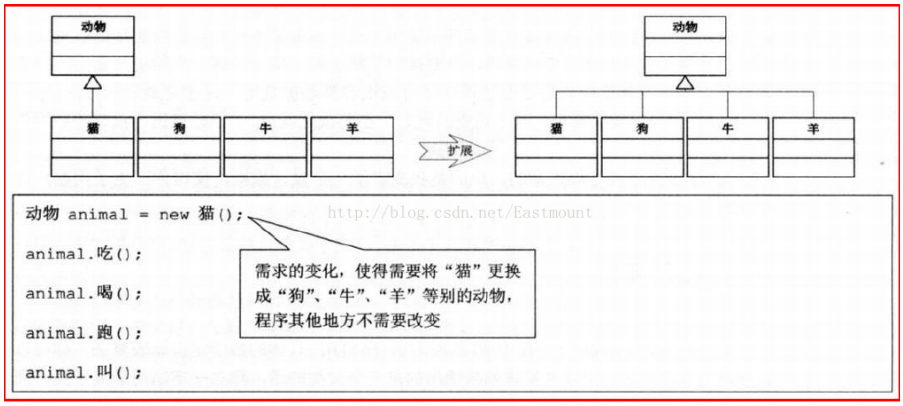
生物学:企鹅属于鸟类

LSP原则:企鹅不属于鸟类,因为企鹅不会"飞",所以企鹅不能继承鸟类



正是因为该原则,使得继承复用成为了可能,只有当子类可以替换掉父类,软件单位的功能不到影响时,父类才能真正被复用,而子类也能够在父类的基础上增加新的行为.由于子类的可替换性才使得父类类型的模块在无需修改的情况下就可以扩展,可以说由于里氏替换原则才使得开闭原则成为可能.

再如动物的例子,动物具有吃、喝、跑、叫等行为,如果需要其他动物也有类似行为,由于它们都继承于动物,只需要更改实例化的地方,程序其他不许改变.如下图所示:



五. 依赖倒置原则

依赖倒置原则(Dependence Inversion Principle)思想：高层模块不应该依赖低层模块,二者都应该依赖于抽象.

1.高层模块只应该包含重要的业务模型和策略选择

2.低层模块则是不同业务和策略的实现

3.高层抽象不依赖高层和底层模块的具体实现,最多依赖低层的抽象

4.低层抽象和实现也只依赖于高层抽象

换句话说：要依赖于抽象,而不要依赖于具体实现;高层是功能和策略,低层是具体实现;编程程序语言就是需要针对接口编程而不要针对实现编程.

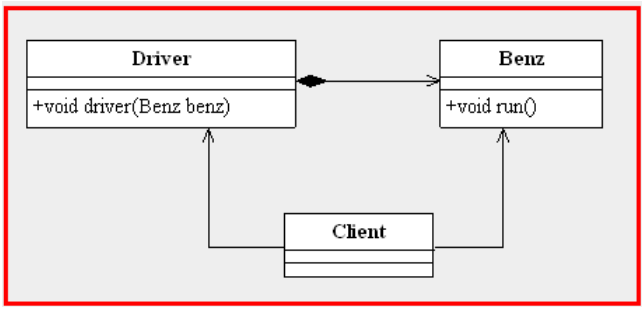
顺便说说与设计模式相关的两个知识点:

1.面向对象的四个优点：可维护、可扩展、可服用、灵活性好

2.高内聚低耦合：高内聚就是一个模块内各个元素彼此结合紧密程度高,一个软件模块只负责一个任务,也就是单一职责原则,提高模块的独立性;低耦合就是软件结构内不同模块之间相互连接的程度低,模块与模块之间尽可能独立存在,模块与模块之间的接口尽量少而且简单,具有更好的复用性和扩展性.

例子理解:(强推&参考博客园cbf4life的文章——依赖倒置原则)

下图表示司机驾驶奔驰车的类图.



代码如下所示,司机通过调用奔驰车run方法开动汽车,通过client场景描述eastmount开奔驰车.

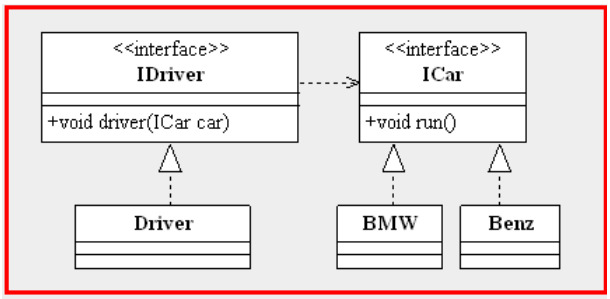
```
//司机类
public class Driver {
    //司机主要职责驾驶汽车
    public void drive(Benz benz) {
        benz.run();
    }
}
//奔驰类
public class Benz {
    public void run() {
        System.out.printIn("奔驰汽车跑");
    }
}
//场景类
```

```
public class Client {
                        public static void main(String[] args) {
        //eastmount开奔驰车
        Driver eastmount = new Driver();
        Benz benz = new Benz();
        eastmount.drive(benz);
    }
}
```

现在司机eastmount不仅要开奔驰车,还要开宝马车,又该怎么实现呢?自定义宝马汽车类BMW,但是不能开,因为eastmount没有开宝马车的方法,出现的问题就是:

司机类和奔驰车类之间是一个紧耦合关系,这导致了系统的可维护性大大降低,增加新的车类如宝马BWM汽车就需要修改司机类drive()方法,这不是稳定性而是易变的.被依赖者的变化竟然让依赖者来承担修改的代价,所以可以通过依赖倒置原则实现.

如下图所示,通过建立两个接口IDriver和ICar,分别定义了司机的职能就是驾驶汽车,通过drive()方法实现;汽车的职能就是运行,通过run()方法实现.



代码如下,接口只是一个抽象化的概念,是对一类事物的抽象描述,具体的实现代码由相应的类来完成,所以还需要定义Driver实现类;同时在IDriver接口中通过传入ICar接口实现抽象之间的依赖关系,Driver实现类也传入了ICar接口,具体开的是哪种车需要在高层模型中声明,具体开发方法在BMW和Benz类中定义.

```
//司机接口 驾驶汽车 抽象
public interface IDriver {
    public void drive(ICar car);
}
//司机类 具体实现
public class Driver implements IDriver{
    //司机的主要职责就是驾驶汽车
    public void drive(ICar car){
        car.run();
    }
}
//汽车接口
public interface ICar {
    public void run();
}
//奔驰车类
public class Benz implements ICar{
    public void run(){
        System.out.println("奔驰汽车跑");
    }
}
//宝马车类
public class BMW implements ICar{
    public void run(){
        System.out.println("宝马汽车跑");
    }
}
//实现eastmount开宝马车
public class Client {
```

```
    public static void main(String[] args) {
                                                    IDriver eastmount = new Driver();
        ICar bmw = new BMW();
        eastmount.drive(bmw);
    }
}
```

由于"抽象不应该依赖细节",所以我们认为抽象(ICar接口)不依赖BMW和Benz两个实现类(细节),在高层次的模块中应用都是抽象,Client就属于高层次业务逻辑,它对于低层次模块的依赖都是建立在抽象上,eastmount都是以IDriver接口类型进行操作,屏蔽了细节对抽象的影响,现在开不同类型的车,只需要修改业务场景类即可实现:

ICar benz = new Benz();

eastmount.drive(benz);

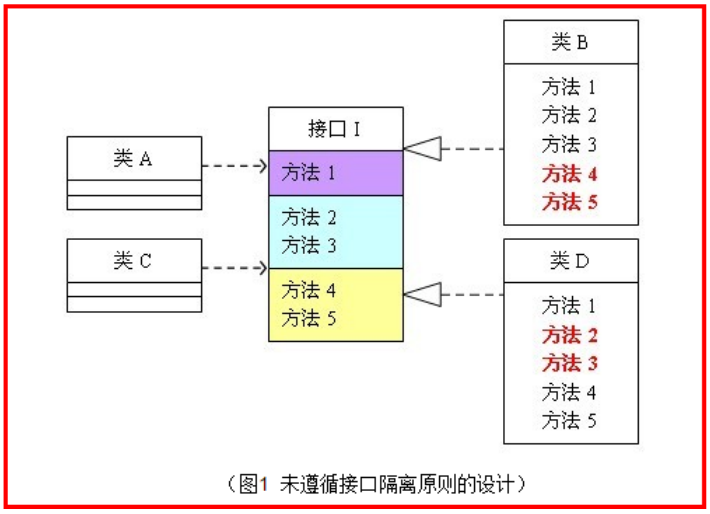
依赖倒置其实可以说是面向对象是设计的标志,用哪种语言来编写程序不重要,如果编写时考虑的都是如何针对抽象编程而不是针对细节编程,即程序中所有的依赖关系都是终止与抽象类或接口,那就是面向对象的设计,反之则是过程化的设计.

六. 接口隔离原则

接口隔离原则(Interface Segregation Principle,ISP)思想：多个和客户相关的接口要好于一个通用接口.如果一个类有几个使用者,与其让这个类再如所有使用这需要使用的所有方法,还不如为每个使用者创建一个特定接口,并让该类分别实现这些接口.

例子理解:(推荐&引用zhengzhb博客——接口隔离原则)

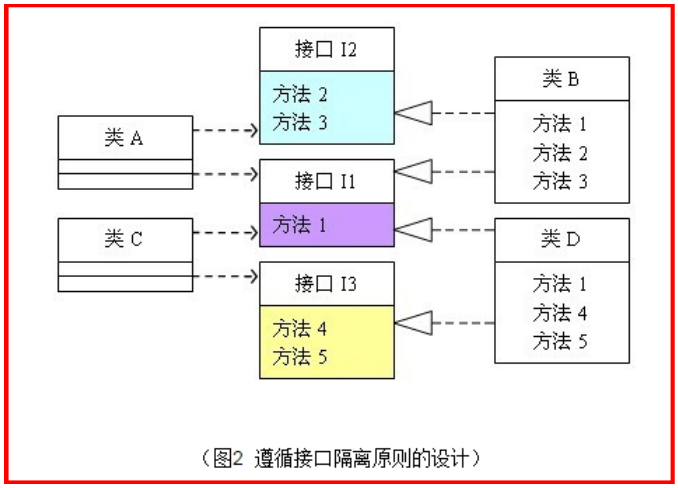
如下图所示,类A依赖接口I中方法1\方法2\方法3,类B是对类A依赖的实现,类C依赖于接口I中的方法方法1\方法4\方法5,类D是对类C依赖的实现.对于类B和类D中都存在用不到的方法,但是由于实现了接口I,所以需要实现这些不用的方法.(代码见原文链接)



（图1 未遵循接口隔离原则的设计）

可以发现接口过于臃肿,只要接口中出现的方法,不管对依赖于它的类有没有用,实现类中都必须去实现这些方法,如类B中方法4和方法5,类D中方法2和方法3,显然这样的设计不适用.

设计如下修改为符合接口隔离原则,必须对接口I进行拆分,拆分为三个接口I1\I2\I3.此时实现类B和类D中方法都是需要使用的方法,这就体现了建立单一接口而不是庞大接口的好处.(源码见上面的链接)



（图2 遵循接口隔离原则的设计）

接口隔离原则即尽量细化接口,接口中的方法尽量少,为各个类建立专用的接口,而不是试图去建立一个庞大而臃肿的接口提供所有依赖于他的类去调用.说到这里,很多人会觉的接口隔离原则跟之前的单一职责原则很相似,其实不然。

其一,单一职责原则原注重的是职责,而接口隔离原则注重对接口依赖的隔离.

其二,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,主要针对抽象,针对程序整体框架的构建.

总结:这是一篇关于回顾设计模式SOLID五大原则的文章,我非常喜欢文章中的例子,每个例子都是我精选了描述模式的,通过Modom讲述了单一职责原则、加减法计算器讲述了开闭原则、企鹅动物讲述了里氏替换原则、通过Driver和Car实现了依赖倒置原则,最后讲述了接口隔离原则.希望文章对大家有所帮助,尤其是学习设计模式的同学和代码写得不太规范或需要重构的同学,如果有错误或不足之处,还请海涵~

(By:Eastmount 2014-11-29 晚上8点 http://blog.csdn.net/eastmount/)

文章参考:

1.<<高级软件工程>>设计模式部分课件及老师讲述内容

2.<<大话设计模式>> 程杰著

3.**博客园cbf4life的文章——**依赖倒置原则

4.**zhengzhb博客——**接口隔离原则