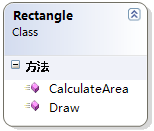
**一 Single Responsibility Principle——单一职责原则**

核心思想: 一个类应该只有一个引起它变化的原因.

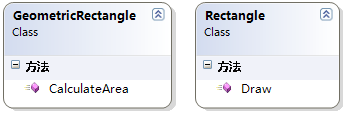
假设存在这样的设计. Rectangle类具有两个方法,一个方法是计算矩形的面积 , 另一个方法是把矩形绘制在屏幕上.



CaculateArea方法只会进行简单的数学运算,而Draw方法则调用GUI组件实现绘制矩形的功能. 显然,这个类就包含了两个不同的职责了. 那这样又会带来什么问题呢? 考虑这样一个场景:现在有一个几何学应用程序调用了这一个类,已便实现计算面积的功能,在这个程序中不需要用到绘制矩形的功能. 问题一:部署几何应用程序需要把GUI组件一同部署,而且这个组件根本没有使用到.问题二:对Rectangle类的改变,比如Draw方法改用另外一套GUI组件,必须对几何应用程序进行一次重新部署.

可见,一个类如果承担的职责过多,就等于把职责耦合在一起了,容易导致脆弱的设计,带来额外的麻烦. 在实际开发中, 业务规则的处理和数据持久化一般是不同时存在同一个类中的,业务规则往往会频繁地变化,而持久化的方式却不会经常性地变化.如果这两个职责混合在同一个类中,业务规则频繁变化导致类的修改,只调用持久化方法的类也必须跟着重新编译,部署的次数常常会超过我们希望的次数. 对业务规则和持久化任务的职责分离就是遵循单一职责原则的体现.

对上述Recangle类可进行这样的修改：



**二 Open Closed Principle——开放封闭原则**

核心思想:对扩展开放,对修改封闭.

"需求总是变化的." 拥抱变化似乎就是软件开发的真理之一. 经常会有这样令人沮丧的情景出现:新的需求来了,对不起,我的代码设计必须大幅度推倒重来. 设计的坏味道让我们深受其害,那么怎样的设计才能面对需求的改变却可以保持相对稳定呢?

针对这样的问题,OCP给了我们如下的建议:在发生变化的时候,不要修改类的源代码,要通过添加新代码来增强现有类的行为.

对扩展开放,对修改封闭,这两个特征似乎就是相互矛盾的. 通常观念来讲,扩展不就是修改源代码吗？怎么可能在不改动源代码的情况下去更改它的行为呢？

答案就是抽象（Interface 和 抽象基类）.实现OCP的核心思想就是对抽象编程. 让类依赖于固定的抽象,对修改就是封闭的; 而通过面向对象的继承和多态机制,通过覆写方法改变固有行为,实现新的扩展方法,对于扩展就是开放的.

来看一个例子. 实现一个能够根据客户端的调用要求绘制圆形和长方形的应用程序. 初始设计如下:

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gifView Code

public class Draw

{

public void DrawRectangle()

{

//绘制长方形

}

public void DrawCircle()

{

//绘制圆形

}

}

public enum Sharp

{

/// <summary>

/// 长方形

/// </summary>

Rectangle ,

/// <summary>

/// 圆形

/// </summary>

Circle ,

}

public class DrawProcess

{

private Draw \_draw = new Draw();

public void Draw(Sharp sharp)

{

switch (sharp)

{

case Sharp.Rectangle:

\_draw.DrawRectangle();

break;

case Sharp.Circle:

\_draw.DrawCircle();

break;

default:

throw new Exception("调用出错!");

}

}

}

//调用代码

DrawProcess draw = new DrawProcess();

draw.Draw(Sharp.Circle);

 现在的代码可以正确地运行. 一切似乎都趋近于理想. 然而,需求的变更总是让人防不胜防. 现在程序要求要实现可以绘制正方形. 在原本的代码设计下,必须做如下的改动.

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gifView Code

//在Draw类中添加

public void DrawSquare()

{

//绘制正方形

}

//在枚举Sharp中添加

/// <summary>

/// 正方形

/// </summary>

Square ,

//在DrawProcess类的switch判断中添加

case Sharp.Square:

\_draw.DrawSquare();

break;

 需求的改动产生了一系列相关模块的改动,设计的坏味道悠然而生. 现在运用OCP, 来看一下如何对代码进行一次重构.

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gifView Code

/// <summary>

/// 绘制接口

/// </summary>

public interface IDraw

{

void Draw();

}

public class Circle:IDraw

{

public void Draw()

{

//绘制圆形

}

}

public class Rectangle:IDraw

{

public void Draw()

{

//绘制长方形

}

}

public class DrawProcess

{

private IDraw \_draw;

public IDraw Draw { set { \_draw = value; } }

private DrawProcess() { }

public DrawProcess(IDraw draw)

{

\_draw = draw;

}

public void DrawSharp()

{

\_draw.Draw();

}

}

//调用代码

IDraw circle = new Circle();

DrawProcess draw = new DrawProcess(circle);

draw.DrawSharp();

假如现在需要有绘制正方形的功能,则只需添加一个类Square 即可.

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gifView Code

public class Square:IDraw

{

public void Draw()

{

//绘制正方形

}

}

只需新增加一个类且对其他的任何模块完全没有影响，OCP出色地完成了任务.

如果一开始就采用第二种代码设计,在需求的暴雨来临时,你会欣喜地发现你已经到家了, 躲过了被淋一身湿的悲剧. 所以在一开始设计的时候,就要时刻地思考,根据对应用领域的理解来判断最有可能变化的种类,然后构造抽象来隔离那些变化**.** 经验在这个时候会显得非常宝贵,可能会帮上你的大忙.

OCP很美好,然而绝对的对修改关闭是不可能的,都会有无法对之封闭的变化. 同时必须清楚认识到遵循OCP的代价也是昂贵的,创建适当的抽象是要花费开发时间和精力的. 如果滥用抽象的话,无疑引入了更大的复杂性,增加维护难度.

**三 Liskov Subsitution Principle——里氏替换原则**

核心思想: 子类必须能够替换掉它们的父类型.

考虑如下情况:

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gifView Code

public class ProgrammerToy

{

private int \_state;

public int State

{

get { return \_state; }

}

public virtual void SetState(int state)

{

\_state = state;

}

}

public class CustomProgrammerToy:ProgrammerToy

{

public override void SetState(int state)

{

//派生类缺乏完整访问能力,即无法访问父类的私有成员\_state

//因此该类型也许不能完成其父类型能够满足的契约

}

}

//控制台应用程序代码

class Program

{

static void Main(string[] args)

{

ProgrammerToy toy = new CustomProgrammerToy();

toy.SetState(5);

Console.Write(toy.State.ToString());

}

}

从语法的角度来看, 代码没有任何问题. 不过从行为的角度来看 , 二者却存在不同. 在使用CustomProgrammerToy替换父类的时候, 输出的是0而不是5, 与既定的目标相差千里. 所以不是所有的子类都能安全地替换其父类使用.

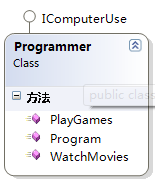
前面谈到的开发封闭原则和里氏替换原则存在着密切的关系. 实现OCP的核心是对抽象编程, 由于子类型的可替换性才使得使用父类类型的模块在无需修改的情况下就可以扩展, 所以违反了里氏替换原则也必定违反了开放封闭原则.

庆幸的是, 里氏替换原则还是有规律可循的.父类尽可能使用接口或抽象类来实现,同时必须从客户的角度理解,按照客户程序的预期来保证子类和父类在行为上的相容.

**四 InterFace Segregation Principle——接口隔离原则**

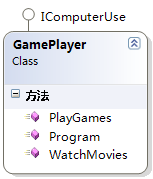
核心思想:使用多个小的专门的接口,而不要使用一个大的总接口.

直接来看一个例子: 假设有一个使用电脑的接口

程序员类实现接口IComputerUse, 玩游戏,编程,看电影, 多好的事情.

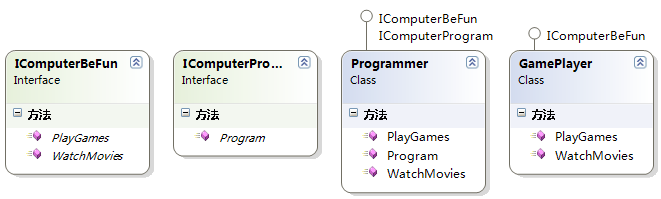
现在有一个游戏发烧友,他也要使用电脑, 为了重用代码 , 实现OCP, 他也实现接口IComputerUse



看出什么问题了吗? GamePlayer PlayGame无可厚非,WatchMovies小消遣, 但要编程干什么?

这就是胖接口带来的弊端,会导致实现的类必须完全实现接口的所有方法, 而有些方法对客户来说是无任何用处的,在设计上这是一种"浪费". 同时,如果对胖接口进行修改, 比如程序员要使用电脑配置为服务器, 在IComputerUse上添加Server方法, 同样GamePlayer也要修改(这种修改对GamePlayer是毫无作用的),是不是就引入了额外的麻烦?

所以应该避免出现胖接口,要使接口实现高内聚(高内聚是指一个模块中各个部分都是为完成一项具体功能而协同工作,紧密联系,不可分割). 当出现了胖接口,就要考虑重构.优先推荐的方法是使用多重继承分离,即实现小接口.



将IComputerUse拆分为IComputerBeFun和IComputerProgram, Progammer类则同时实现IComputerBeFun和IComputerProgram接口,现在就各取所需了.

与OCP类似, 接口也并非拆分地越小越好, 因为太多的接口会影响程序的可读性和维护性,带来难以琢磨的麻烦. 所以设计接口的时刻要着重考虑高内聚性, 如果接口中的方法都归属于同一个逻辑划分而协同工作,那么这个接口就不应该再拆分.

**五 Dependency Inversion Principle——依赖倒置原则**

核心思想: 高层模块不应该依赖底层模块，两者都应该依赖抽象。抽象不应该依赖细节，细节应该依赖抽象。

当一个类A存在指向另一个具体类B的引用的时候，类A就依赖于类B了。如：

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gifView Code

/// <summary>

/// 商品类

/// </summary>

public class Product

{

public int Id { get; set; }

}

/// <summary>

/// 商品持久化类

/// </summary>

public class ProductRepository

{

public IList<Product> FindAll()

{

//假设从SQL Server数据库中获取数据

return null;

}

}

/// <summary>

/// 商品服务类

/// </summary>

public class ProductService

{

private ProductRepository \_productRepository;

public IList<Product> GetProducts()

{

\_productRepository = new ProductRepository();

return \_productRepository.FindAll();

}

}

 （在前面单一职责原则中有提到，业务逻辑处理和对象持久化分属两个职责，所以应该拆分为两个类。）高层模块ProductService类中引用了底层模块具体类ProductRepository，所以ProductService类就直接依赖于ProductRepository了。那么这样的依赖会带来什么问题呢？

"需求总是那么不期而至"。原本ProductRepository是从SQL Server[**数据库**](http://lib.csdn.net/base/14)中读存数据，现在要求从[**MySQL**](http://lib.csdn.net/base/14)数据库中读存数据。由于高层模块依赖于底层模块，现在底层模块ProductRepository发生了更改，高层模块ProductService也需要跟着一起修改，回顾之前谈到的设计原则，这是不是就违反了OCP呢？OCP的核心思想是对抽象编程，DIP的思想是依赖于抽象，这也让我们更清楚地认识到，面向对象设计的时候，要综合所有的设计原则考虑。DIP给出了解决方案：在依赖之间定义一个接口，使得高层模块调用接口，而底层模块实现接口，以此来控制耦合关系。（在上面OCP的例子中，也是使用了这一个方法。）所以可以对代码做如下的重构：

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gifView Code

/// <summary>

/// 商品持久化接口

/// </summary>

public interface IProductRepository

{

List<Product> FindAll();

}

/// <summary>

/// 商品持久化类

/// </summary>

public class ProductRepository:IProductRepository

{

public IList<Product> FindAll()

{

//假设从SQL Server数据库中获取数据

return null;

}

}

/// <summary>

/// 商品服务类

/// </summary>

public class ProductService

{

private IProductRepository \_productRepository;

private ProductService() { }

//使用构造函数依赖注入

public ProductService(IProductRepository productRepository)

{

\_productRepository = productRepository;

}

public IList<Product> GetProducts()

{

return \_productRepository.FindAll();

}

}

现在已对变化进行了抽象隔离，再根据OCP，我相信实现从MySQL数据库中读存数据的需求已经可以被轻松地解决掉了。