

Learning hard

博客园

首页

新随笔

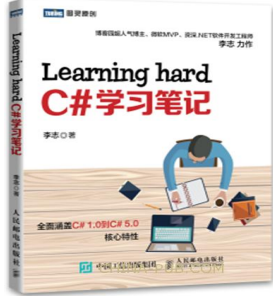
联系

订阅

管理

随笔 - 159 文章 - 43 评论 - 2563

《Learning hard C#学习笔记》



- [1] 互动网
- [2] 京东
- [3] 当当
- [4] 亚马逊
- [5] 天猫

.NET高级交流群:  加入QQ群
程序员内推群:  加入QQ群



程序员俱乐部公众号:



昵称: Learning hard
园龄: 4年9个月
荣誉: 推荐博客
粉丝: 2572
关注: 168
+加关注

我的标签

[DDD](#)(9)
[WPF](#)(8)
[SignalR](#)(4)
[KnockoutJs](#)(3)
[领域驱动设计](#)(3)
[Bootstrap](#)(3)
[Asp.net MVC](#)(3)
[MSMQ](#)(2)
[Asp.net SignalR](#)(2)
[Owin](#)(2)
[更多](#)

随笔分类(176)

C#设计模式总结

一、引言

经过这段时间对设计模式的学习，自己的感触还是很多的，因为我现在在写代码的时候，经常会想想这里能不能用什么设计模式来进行重构。所以，学完设计模式之后，感觉它会慢慢地影响到你写代码的思维方式。这里对设计模式做一个总结，一来可以对所有设计模式进行一个梳理，二来可[赞助](#)一个索引来帮助大家收藏。

PS: 其实，很早之前我就看过所有的设计模式了，但是并没有写博客，但是不久就很快忘记了，也没有起到什么作用，这次以博客的形式总结出来，发现效果还是不错的，因为通过这种总结的方式，我对它理解更深刻了，也记住了的更牢靠了，也影响了自己平时实现功能的思维。所以，我鼓励大家可以通过做笔记的方式来把自己学到的东西进行梳理，这样相信可以理解更深，更好，我也会一直写下来，之后打算写WCF一系列文章。

其实WCF内容很早也看过了，并且博客园也有很多前辈写的很好，但是，我觉得我还是需要自己总结，因为只有这样，知识才是自己的，别人写的多好，你看了之后，其实还是别人了，所以鼓励大家几点（对于这几点，也是对自己的一个提醒）：

1. 要动手实战别人博客中的例子；
2. 实现之后进行总结，可以写博客也可以自己记录云笔记等；
3. 想想能不能进行扩展，进行举一反三。

系列导航:

[C#设计模式\(1\)——单例模式](#)[C#设计模式\(2\)——简单工厂模式](#)[C#设计模式\(3\)——工厂方法模式](#)[C#设计模式\(4\)——抽象工厂模式](#)[C#设计模式\(5\)——建造者模式 \(Builder Pattern\)](#)[C#设计模式\(6\)——原型模式 \(Prototype Pattern\)](#)[C#设计模式\(7\)——适配器模式 \(Adapter Pattern\)](#)[C#设计模式\(8\)——桥接模式 \(Bridge Pattern\)](#)[C#设计模式\(9\)——装饰者模式 \(Decorator Pattern\)](#)[C#设计模式\(10\)——组合模式 \(Composite Pattern\)](#)[C#设计模式\(11\)——外观模式 \(Facade Pattern\)](#)[C#设计模式\(12\)——享元模式 \(Flyweight Pattern\)](#)[C#设计模式\(13\)——代理模式 \(Proxy Pattern\)](#)[C#设计模式\(14\)——模板方法模式 \(Template Method\)](#)[C#设计模式\(15\)——命令模式 \(Command Pattern\)](#)[C#设计模式\(16\)——迭代器模式 \(Iterator Pattern\)](#)[C#设计模式\(17\)——观察者模式 \(Observer Pattern\)](#)[C#设计模式\(18\)——中介者模式 \(Mediator Pattern\)](#)[C#设计模式\(19\)——状态者模式 \(State Pattern\)](#)[C#设计模式\(20\)——策略者模式 \(Strategy Pattern\)](#)[C#设计模式\(21\)——责任链模式](#)[C#设计模式\(22\)——访问者模式 \(Visitor Pattern\)](#)[C#设计模式\(23\)——备忘录模式 \(Memento Pattern\)](#)

二、设计原则

使用设计模式的根本原因是适应变化，提高代码复用率，使软件更具有可维护性和可扩展性。并且，在进行设计的时候，也

.NET领域驱动设计系列(12)
Asp.net(5)
Asp.net MVC(8)
Asp.net Web API
C# 互操作入门系列(4)
C#多线程处理系列(7)
C#基础知识梳理系列(23)
C#进阶系列(2)
C#开发技巧篇(7)
C#网络编程系列(12)
CLR(12)
Index文章索引(4)
Javascript
Lua+Cocos2d-x 3.1.1脚本游戏开发系列(1)
No-Sql(2)
SQL Server
Unity3D
VSTO之旅系列(5)
WCF(13)
Windows Azure
WPF(10)
编程之美学习笔记
程序人生(6)
跟我学Asp.net开发(3)
跟我一起学C# 设计模式(24)
跟我一起学STL(2)
构建高性能应用系列
好文摘录(1)
开源代码
你必须知道的并行编程系列
你必须知道的异步编程系列(4)
前端(6)
深入浅出话VC++(3)
算法与数据结构
微服务&SOA

随笔档案(159)

2016年5月 (3)
2016年4月 (8)
2016年1月 (1)
2015年10月 (1)
2015年8月 (2)
2015年7月 (2)
2015年6月 (6)
2015年5月 (4)
2015年4月 (1)
2015年3月 (4)
2014年12月 (9)
2014年11月 (7)
2014年10月 (7)
2014年9月 (11)
2014年7月 (1)
2014年2月 (1)
2014年1月 (3)
2013年12月 (2)
2013年10月 (9)
2013年9月 (5)
2013年8月 (3)
2013年7月 (3)
2013年6月 (4)
2013年5月 (6)
2013年3月 (5)
2013年2月 (1)
2013年1月 (6)
2012年12月 (6)
2012年11月 (6)
2012年10月 (8)
2012年9月 (7)
2012年8月 (5)
2012年7月 (10)
2012年6月 (2)

文章档案(9)

2016年8月 (1)

需要遵循以下几个原则：单一职责原则、开放封闭原则、里氏代替原则、依赖倒置原则、接口隔离原则、合成复用原则和迪米特法则。下面就分别介绍了每种设计原则。

2.1 单一职责原则

就一个类而言，应该只有一个引起它变化的原因。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影响到其他的职责，另外，把多个职责耦合在一起，也会影响复用性。

2.2 开闭原则(Open-Closed Principle)

开闭原则即OCP（Open-Closed Principle缩写）原则，该原则强调的是：一个软件实体（指的类、函数、模块等）应该对扩展开放，对修改关闭。即每次发生变化时，要通过添加新的代码来增强现有类型的行为，而不是修改原有的代码。

符合开闭原则的最好方式是提供一个固有的接口，然后让所有可能发生变化的类实现该接口，让固定的接口与相关对象进行交互。

2.3 里氏代替原则(Liskov Substitution Principle)

Liskov Substitution Principle,LSP（里氏代替原则）指的是子类必须替换掉它们的父类型。也就是说，在软件开发过程中，子类替换父类后，程序的行为是一样的。只有当子类替换掉父类后，此时软件的功能不受影响时，父类才能真正地被复用，而子类也可以在父类的基础上添加新的行为。为了来看看违反LSP原则的例子，具体代码如下所示：

```
public class Rectangle
{
    public virtual long Width { get; set; }
    public virtual long Height { get; set; }
}

// 正方形
public class Square : Rectangle
{
    public override long Height
    {
        get
        {
            return base.Height;
        }
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }

    public override long Width
    {
        get
        {
            return base.Width;
        }
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }
}

class Test
{
    public void Resize(Rectangle r)
    {
        while (r.Height >= r.Width)
        {
            r.Width += 1;
        }
    }

    var r = new Square() { Width = 10, Height = 10 };
    new Test().Resize(r);
}
```

上面的设计，正如上面注释的一样，在执行SmartTest的resize方法时，如果传入的是长方形对象，当高度大于宽度时，会自动增加宽度直到超出高度。但是如果传入的是正方形对象，则会陷入死循环。此时根本原因是，矩形不能作为正方形的父类，既然出现了问题，可以进行重构，使它们俩都继承于四边形类。重构后的代码如下所示：

2016年3月 (1)
2015年8月 (2)
2014年10月 (1)
2014年9月 (2)
2013年8月 (1)
2013年5月 (1)

友情链接

Asp.net MVC
SQL Server Study 1
SQL Server Study 2
SQL Server Study 3
VC 知识库
WCF
WPF
WPF 2
WPF 3
个人开发历程知识库
框架设计
框架设计2

积分与排名

积分 - 418762
排名 - 271

最新评论

1. Re:[.NET领域驱动设计实战系列]专题九: DDD案例: 网上书店AOP和站点地图的实现
无限支持! 老李!
--monkey's

2. Re:[Asp.net 开发系列之SignalR篇]专题三: 使用SignalR实现聊天室的功能
无限支持!
--monkey's


3. Re:[Asp.net 开发系列之SignalR篇]专题四: 使用SignalR实现发送图片
这个有逼格!
--monkey's

4. Re:[后端人员耍前端系列]AngularJs篇: 使用AngularJs打造一个简易权限系统PrivilegeManagement 这个没有数据库吗?
--monkey's

5. Re:[后端人员耍前端系列]AngularJs篇: 使用AngularJs打造一个简易权限系统写的不错!
--monkey's

阅读排行榜

1. [你必须知道的异步编程]C# 5.0 新特性——Async和Await使异步编程更简单(23078)
2. VSTO之旅系列(一): VSTO入门(18918)
3. 如何知道自己的CPU支持SLAT(18812)
4. [C# 网络编程系列]专题五: TCP编程(17904)
5. C#设计模式总结(17416)
6. [C# 网络编程系列]专题八: P2P编程(15544)
7. [C# 网络编程系列]专题九: 实现类似QQ的即时通信程序(15057)
8. C#设计模式(1)——单例模式(14492)
9. [C# 基础知识系列]专题一: 深入解析委托——C#中为什么要引入委托(14312)
10. [C# 网络编程系列]专题六: UDP编程(12599)
11. C# 基础知识系列文章索引(12307)
12. [C#]网络编程系列专题二: HTTP协议详解(11131)
13. [C# 网络编程系列]专题四: 自定义Web浏览器(10717)
14. C#设计模式(2)——简单工厂模式(10



```
// 四边形
public abstract class Quadrangle
{
    public virtual long Width { get; set; }
    public virtual long Height { get; set; }
}

// 矩形
public class Rectangle : Quadrangle
{
    public override long Height { get; set; }

    public override long Width { get; set; }
}


// 正方形
public class Square : Quadrangle
{
    public long _side;

    public Square(long side)
    {
        _side = side;
    }
}

class Test
{
    public void Resize(Quadrangle r)
    {
        while (r.Height >= r.Width)
        {
            r.Width += 1;
        }
    }

    static void Main(string[] args)
    {
        var s = new Square(10);

        new Test().Resize(s);
    }
}
```



2.4 依赖倒置原则

依赖倒置（Dependence Inversion Principle, DIP）原则指的是抽象不应该依赖于细节，细节应该依赖于抽象，也就是提出的“面向接口编程，而不是面向实现编程”。这样可以降低客户与具体实现的耦合。

2.5 接口隔离原则

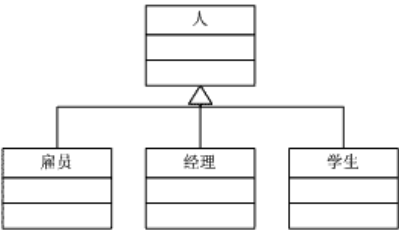
接口隔离原则（Interface Segregation Principle, ISP）指的是使用多个专门的接口比使用单一的总接口要好。也就是说不要让一个单一的接口承担过多的职责，而应把每个职责分离到多个专门的接口中，进行接口分离。过于臃肿的接口是对接口的一种污染。

2.6 合成复用原则

合成复用原则（Composite Reuse Principle, CRP）就是在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分。新对象通过向这些对象的委派达到复用已用功能的目的。简单地说，就是要尽量使用合成/聚合，尽量不要使用继承。

要使用好合成复用原则，首先需要区分“Has—A”和“Is—A”的关系。

“Is—A”是指一个类是另一个类的“一种”，是属于的关系，而“Has—A”则不同，它表示某一个角色具有某一项责任。导致错误的使用继承而不是聚合的常见的原因是错误地把“Has—A”当成“Is—A”。例如：



640)
15. [C# 网络编程系列]专题一：网络协议简介(10405)

评论排行榜

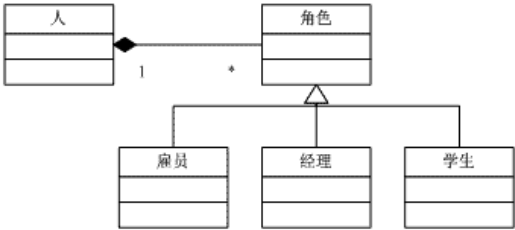
- 1. 《Learninghard C# 学习笔记》回馈网友，免费送书5本(178)
- 2. [你必须知道的异步编程]C# 5.0 新特性——Async和Await使异步编程更简单(91)
- 3. [C# 网络编程系列]专题八：P2P编程(87)
- 4. [C# 网络编程系列]专题九：实现类似QQ的即时通信程序(82)
- 5. [.NET领域驱动设计实战系列]专题二：结合领域驱动设计的面向服务架构来搭建网上书店(80)

推荐排行榜

- 1. [C# 基础知识系列]专题一：深入解析委托——C#中为什么要引入委托(91)
- 2. [你必须知道的异步编程]C# 5.0 新特性——Async和Await使异步编程更简单(83)
- 3. [C# 网络编程系列]专题八：P2P编程(82)
- 4. C#设计模式总结(81)
- 5. C# 基础知识系列文章索引(75)
- 6. [C# 网络编程系列]专题九：实现类似QQ的即时通信程序(72)
- 7. [C#网络编程系列]专题一：网络协议简介(71)
- 8. [C#]网络编程系列专题二：HTTP协议详解(56)
- 9. C#设计模式(1)——单例模式(56)
- 10. [C# 网络编程系列]专题十：实现简单的邮件收发器(52)
- 11. 我的微软最有价值专家(Microsoft MVP)之路(49)
- 12. [C#基础知识系列]专题十七：深入理解动态类型(49)
- 13. [C# 基础知识系列]专题二：委托的本质论(43)
- 14. [你必须知道的异步编程]——异步编程模型(APM)(42)
- 15. [.NET领域驱动设计实战系列]专题二：结合领域驱动设计的面向服务架构来搭建网上书店(39)

实际上，雇员、经历、学生描述的是一种角色，比如一个人是“经理”必然是“雇员”。在上面的设计中，一个人无法同时拥有多个角色，是“雇员”就不能再是“学生”了，这显然不合理，因为现在很多在职研究生，即使雇员也是学生。

上面的设计的错误源于把“角色”的等级结构与“人”的等级结构混淆起来了，误把“Has—A”当作“Is—A”。具体的解决方法就是抽象出一个角色类：



2.7 迪米特法则

迪米特法则（Law of Demeter, LoD）又叫最少知识原则（Least Knowledge Principle, LKP），指的是一个对象应当对其他对象有尽可能少的了解。也就是说，一个模块或对象应尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立，这样当一个模块修改时，影响的模块就会越少，扩展起来更加容易。

关于迪米特法则其他的一些表述有：只与你直接的朋友们通信；不要跟“陌生人”说话。

外观模式（Facade Pattern）和中介者模式（Mediator Pattern）就使用了迪米特法则。

三、创建型模式

创建型模式就是用来创建对象的模式，抽象了实例化的过程。所有的创建型模式都有两个共同点。第一，它们都将系统使用哪些具体类的信息封装起来；第二，它们隐藏了这些类的实例是如何被创建和组织的。创建型模式包括单例模式、工厂方法模式、抽象工厂模式、建造者模式和原型模式。

- 单例模式：解决的是实例化对象的个数的问题，比如抽象工厂中的工厂、对象池等，除了Singleton之外，其他创建型模式解决的都是 new 所带来的耦合关系。
- 抽象工厂：创建一系列相互依赖对象，并能在运行时改变系列。
- 工厂方法：创建单个对象，在Abstract Factory有使用到。
- 原型模式：通过拷贝原型来创建新的对象。

工厂方法，抽象工厂，建造者都需要一个额外的工厂类来负责实例化“一个对象”，而Prototype则是通过原型（一个特殊的工厂类）来克隆“易变对象”。

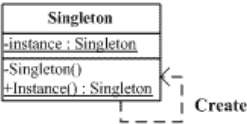
下面详细介绍下它们。

3.1 单例模式

单例模式指的是确保某一个类只有一个实例，并提供一个全局访问点。解决的是实体对象个数的问题，而其他的建造者模式都是解决new所带来的耦合关系问题。其实现要点有：

- 类只有一个实例。问：如何保证呢？答：通过私有构造函数来保证类外部不能对类进行实例化
- 提供一个全局的访问点。问：如何实现呢？答：创建一个返回该类对象的静态方法

单例模式的结构图如下所示：

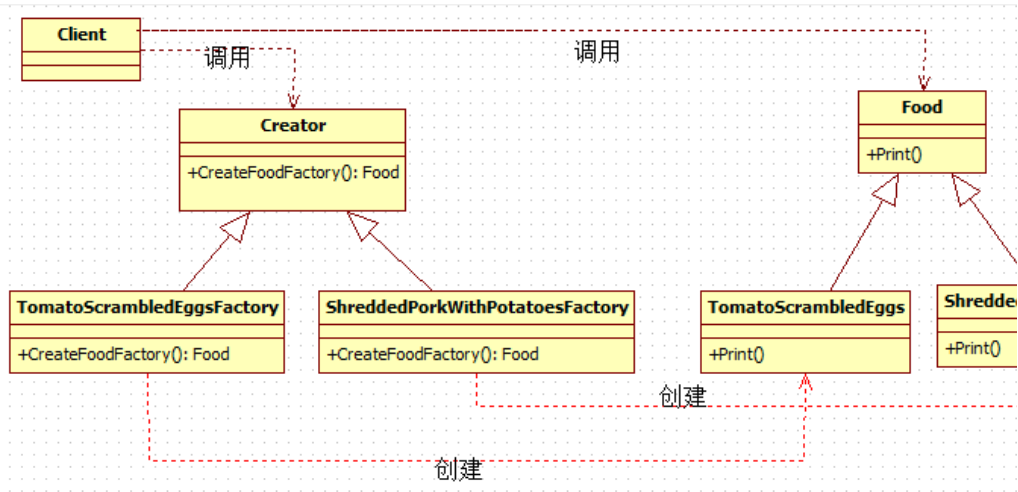


3.2 工厂方法模式

工厂方法模式指的是定义一个创建对象的工厂接口，由其子类决定要实例化的类，将实际创建工作推迟到子类中。它强调的是“单个对象”的变化。其实现要点有：

- 定义一个工厂接口。问：如何实现呢？答：声明一个工厂抽象类
- 由其具体子类创建对象。问：如何去实现呢？答：创建派生于工厂抽象类，即由具体工厂去创建具体产品，既然要创建产品，自然需要产品抽象类和具体产品类了。

其具体的UML结构图如下所示：



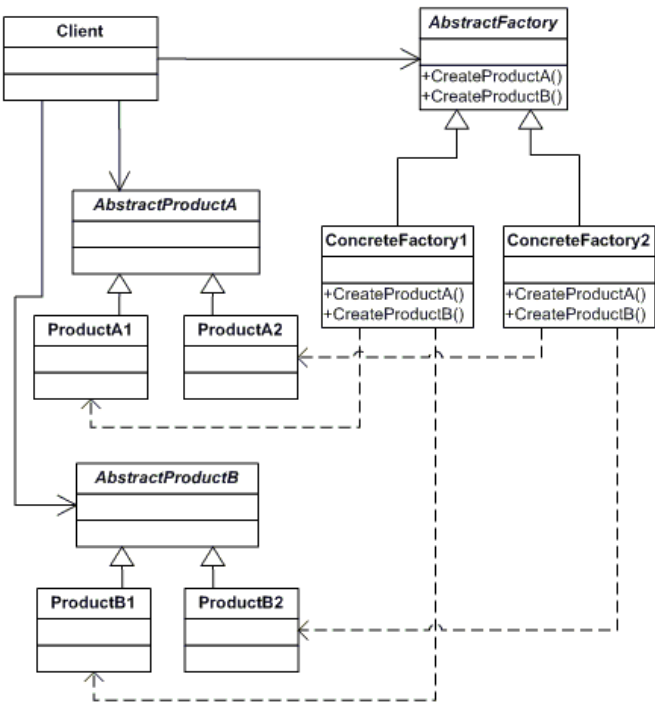
在工厂方法模式中，工厂类与具体产品类具有平行的等级结构，它们之间是一一对应关系。

3.3 抽象工厂模式

抽象工厂模式指的是提供一个创建一系列相关或相互依赖对象的接口，使得客户端可以在不必指定产品的具体类型的情况下，创建多个产品族中的产品对象，强调的是“系列对象”的变化。其实现要点有：

- 提供一系列对象的接口。问：如何去实现呢？答：提供多个产品的抽象接口
- 创建多个产品族中的多个产品对象。问：如何做到呢？答：每个具体工厂创建一个产品族中的多个产品对象，多个具体工厂就可以创建多个产品族中的多个对象了。

具体的UML结构图如下所示：

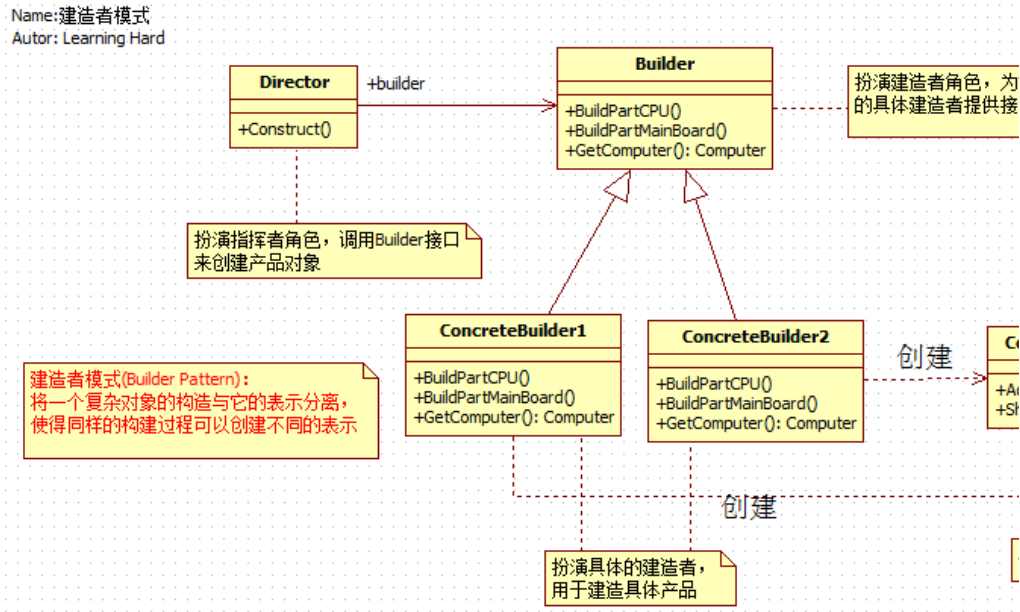


3.4 建造者模式

建造者模式指的是将一个产品的内部表示与产品的构造过程分割开来，从而可以使一个建造过程生成具体不同的内部表示的产品对象。强调的是产品的构造过程。其实现要点有：

- 将产品的内部表示与产品的构造过程分割开来。问：如何把它们分割开呢？答：不要把产品的构造过程放在产品类中，而是由建造者类来负责构造过程，产品的内部表示放在产品类中，这样不就分割开了嘛。

具体的UML结构图如下所示：

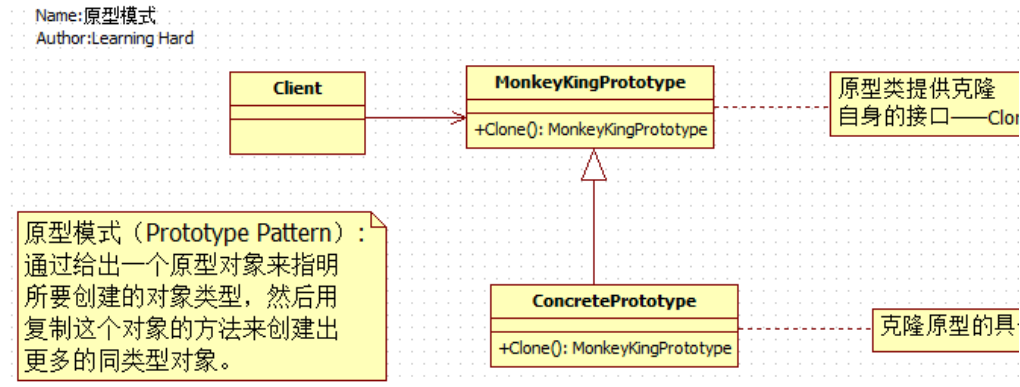


3.5 原型工厂模式

原型模式指的是通过给出一个原型对象来指明所要创建的对象类型, 然后用复制的方法来创建出更多的同类型对象。其实关键点有:

- 给出一个原型对象。问: 如何办到呢? 答: 很简单嘛, 直接给出一个原型类就好了。
- 通过复制的方法来创建同类型对象。问: 又是如何实现呢? 答: .NET可以直接调用MemberwiseClone方法来实现浅拷贝

具体的UML结构图如下所示:



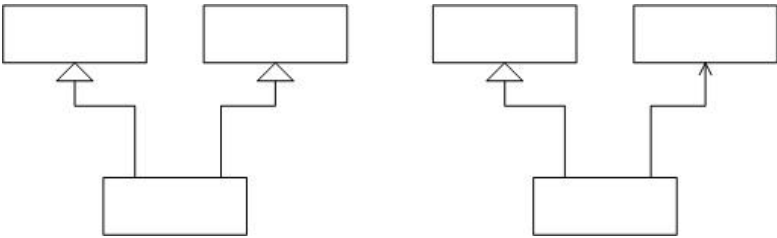
四、结构型模式

结构型模式, 顾名思义讨论的是类和对象的结构, 主要用来处理类或对象的组合。它包括两种类型, 一是类结构型模式, 指的是采用继承机制来组合接口或实现; 二是对象结构型模式, 指的是通过组合对象的方式来实现新的功能。它包括适配器模式、桥接模式、装饰者模式、组合模式、外观模式、享元模式和代理模式。

- 适配器模式注重转换接口, 将不吻合的接口适配对接
- 桥接模式注重分离接口与其实现, 支持多维度变化
- 组合模式注重统一接口, 将“一对多”的关系转化为“一对一”的关系
- 装饰者模式注重稳定接口, 在此前提下为对象扩展功能
- 外观模式注重简化接口, 简化组件系统与外部客户端程序的依赖关系
- 享元模式注重保留接口, 在内部使用共享技术对对象存储进行优化
- 代理模式注重假借接口, 增加间接层来实现灵活控制

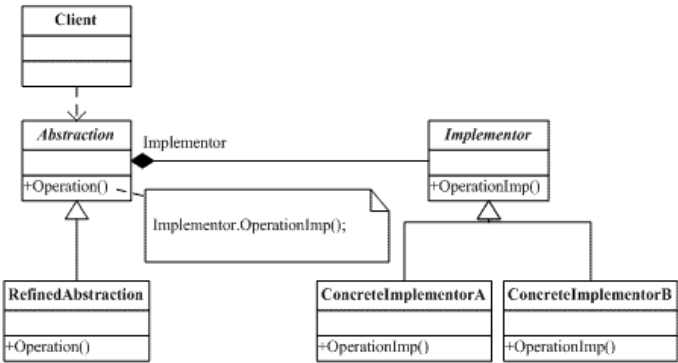
4.1 适配器模式

适配器模式意在转换接口, 它能够使原本不能在一起工作的两个类一起工作, 所以经常用来在类库的复用、代码迁移等方面。例如DataAdapter类就应用了适配器模式。适配器模式包括类适配器模式和对象适配器模式, 具体结构如下图所示, 左边是类适配器模式, 右边是对象适配器模式。



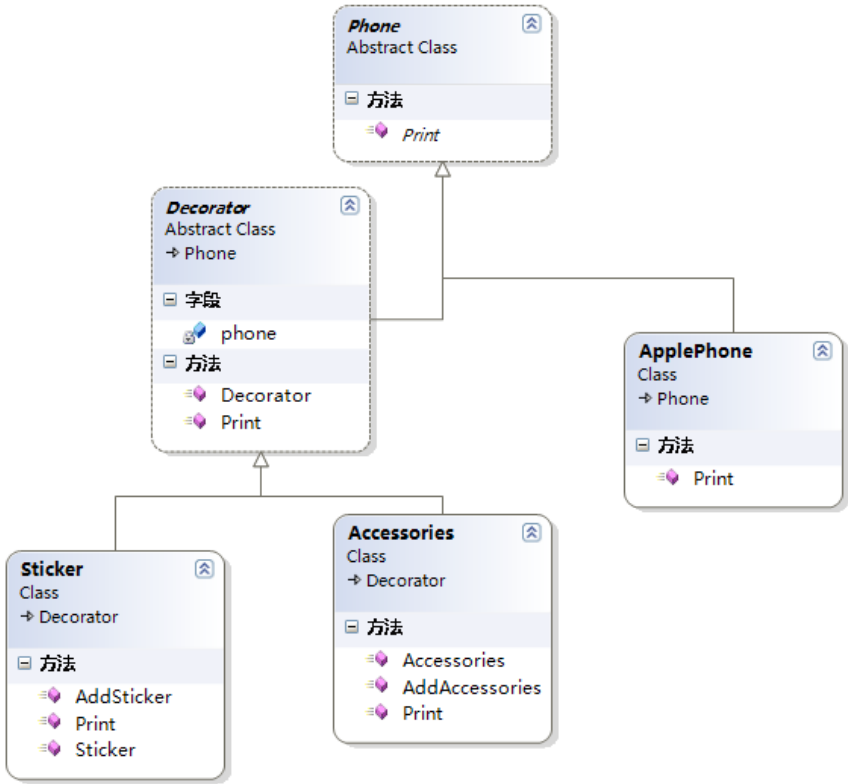
4.2 桥接模式

桥接模式旨在将抽象化与实现化解耦，使得两者可以独立地变化。意思就是说，桥接模式把原来基类的实现化细节再进一步进行抽象，构造到一个实现化的结构中，然后再把原来的基类改造成一个抽象化的等级结构，这样就可以实现系统在多个维度的独立变化，桥接模式的结构图如下所示。



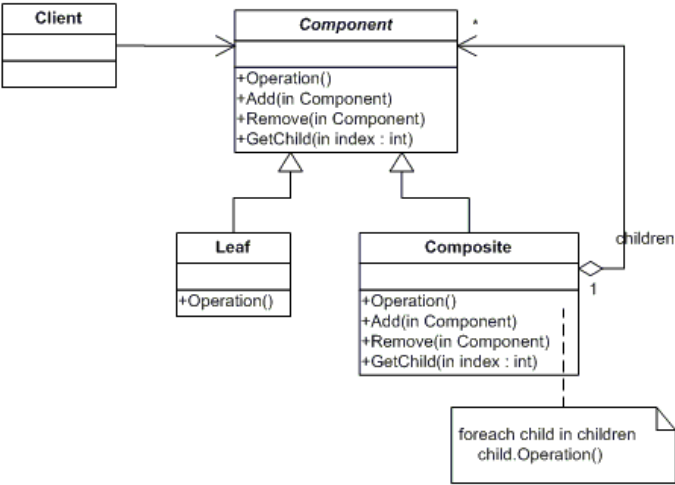
4.3 装饰者模式

装饰者模式又称包装（Wrapper）模式，它可以动态地给一个对象添加一些额外的功能，装饰者模式较继承生成子类的方式更加灵活。虽然装饰者模式能够动态地将职责附加到对象上，但它也会造成产生一些细小的对象，增加了系统的复杂度。具体的结构图如下所示。



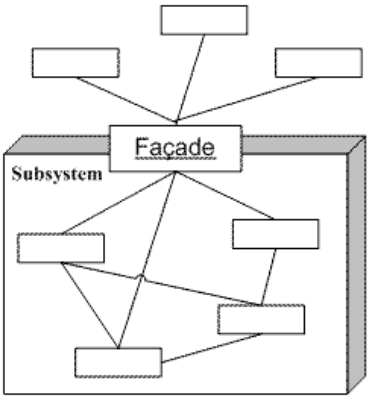
4.4 组合模式

组合模式又称为部分—整体模式。组合模式将对象组合成树形结构，用来表示整体与部分的关系。组合模式使得客户端将单个对象和组合对象同等对待。如在.NET中WinForm中的控件，TextBox、Label等简单控件继承与Control类，同时GroupBox这样的组合控件也是继承于Control类。组合模式的具体结构图如下所示。



4.5 外观模式

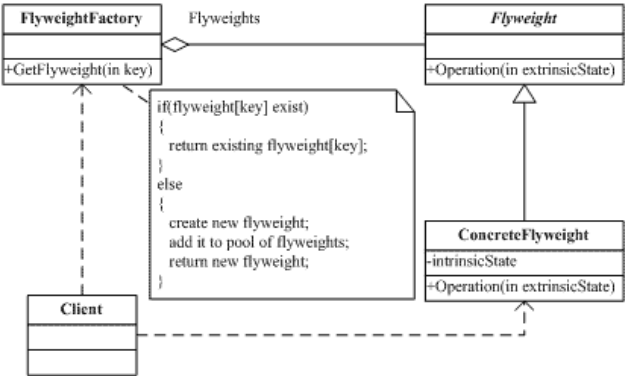
在系统中，客户端经常需要与多个子系统进行交互，这样导致客户端会随着子系统的变化而变化，此时可以使用外观模式把客户端与各个子系统解耦。外观模式指的是为子系统的一组接口提供一个一致的门面，它提供了一个高层接口，这个接口使子系统更加容易使用。如电信的客户专员，你可以让客户专员来完成冲话费，修改套餐等业务，而不需要自己去与各个子系统进行交互。具体类结构图如下所示：



4.6 享元模式

在系统中，如何我们需要重复使用某个对象时，此时如果重复地使用new操作符来创建这个对象的话，这对系统资源是一个极大的浪费，既然每次使用的都是同一个对象，为什么不能对其共享呢？这也是享元模式出现的原因。

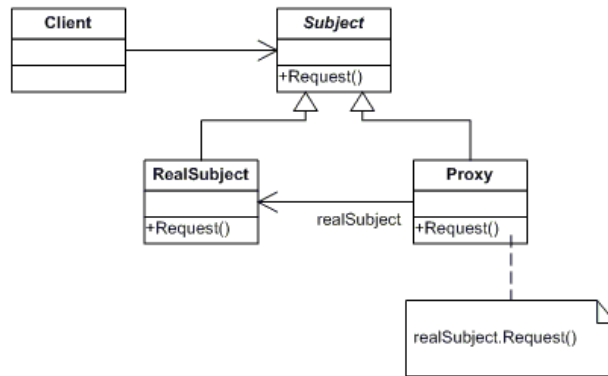
享元模式运用共享的技术有效地支持细粒度的对象，使其进行共享。在.NET类库中，String类的实现就使用了享元模式，String类采用字符串驻留池的来使字符串进行共享。更多内容参考博文：<http://www.cnblogs.com/artech/archive/2010/11/25/internedstring.html>。享元模式的具体结构图如下所示。



4.7 代理模式

在系统开发中，有些对象由于网络或其他的障碍，以至于不能直接对其访问，此时可以通过一个代理对象来实现对目标对象的访问。如.NET中的调用Web服务等操作。

代理模式指的是给某一个对象提供一个代理，并由代理对象控制对原对象的访问。具体的结构图如下所示。



注：外观模式、适配器模式和代理模式区别？

解答：这三个模式的相同之处是，它们都是作为客户端与真实被使用的类或系统之间的一个中间层，起到让客户端间接调用真实类的作用，不同之处在于，所应用的场合和意图不同。

代理模式与外观模式主要区别在于，代理对象无法直接访问对象，只能由代理对象提供访问，而外观对象提供对各个子系统简化访问调用接口，而适配器模式则不需要虚构一个代理者，目的是复用原有的接口。外观模式是定义新的接口，而适配器则是复用一个原有的接口。

另外，它们应用设计的不同阶段，外观模式用于设计的前期，因为系统需要前期就需要依赖于外观，而适配器应用于设计完成之后，当发现设计完成的类无法协同工作时，可以采用适配器模式。然而很多情况下在设计初期就要考虑适配器模式的使用，如涉及到大量第三方应用接口的情况；代理模式是模式完成后，想以服务的方式提供给其他客户端进行调用，此时其他客户端可以使用代理模式来对模块进行访问。

总之，代理模式提供与真实类一致的接口，旨在用来代理类来访问真实的类，外观模式旨在简化接口，适配器模式旨在转换接口。

五、行为型模式

行为型模式是对在不同对象之间划分责任和算法的抽象化。行为模式不仅仅关于类和对象，还关于它们之间的相互作用。行为型模式又分为类的行为模式和对象的行为模式两种。

- 类的行为模式——使用继承关系在几个类之间分配行为。
- 对象的行为模式——使用对象聚合的方式来分配行为。

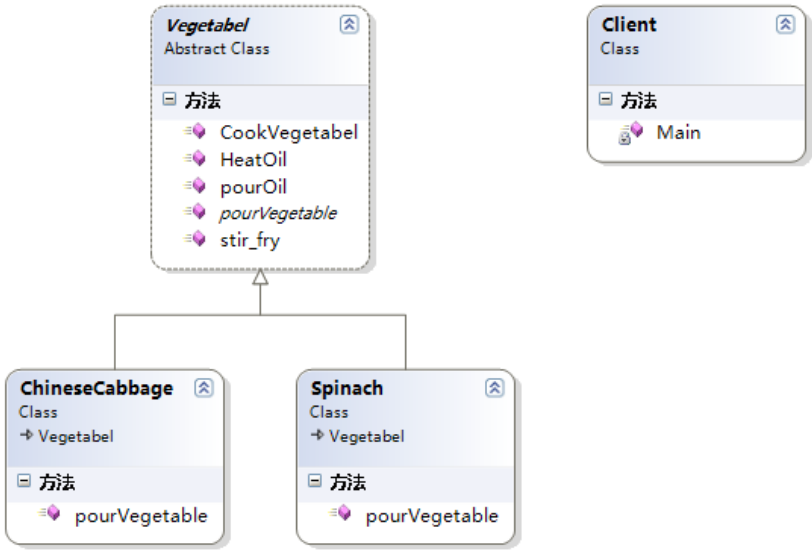
行为型模式包括11种模式：模板方法模式、命令模式、迭代器模式、观察者模式、中介者模式、状态模式、策略模式、责任链模式、访问者模式、解释器模式和备忘录模式。

- 模板方法模式：封装算法结构，定义算法骨架，支持算法子步骤变化。
- 命令模式：注重将请求封装为对象，支持请求的变化，通过将一组行为抽象为对象，实现行为请求者和行为实现者之间的解耦。
- 迭代器模式：注重封装特定领域变化，支持集合的变化，屏蔽集合对象内部复杂结构，提供客户程序对它的透明遍历。
- 观察者模式：注重封装对象通知，支持通信对象的变化，实现对象状态改变，通知依赖它的对象并更新。
- 中介者模式：注重封装对象间的交互，通过封装一系列对象之间的复杂交互，使他们不需要显式相互引用，实现解耦。
- 状态模式：注重封装与状态相关的行为，支持状态的变化，通过封装对象状态，从而在其内部状态改变时改变它的行为。
- 策略模式：注重封装算法，支持算法的变化，通过封装一系列算法，从而可以随时独立于客户替换算法。
- 责任链模式：注重封装对象责任，支持责任的变化，通过动态构建职责链，实现事务处理。
- 访问者模式：注重封装对象操作变化，支持在运行时为类结构添加新的操作，在类层次结构中，在不改变各类的前提下定义了作用于这些类实例的新的操作。
- 备忘录模式：注重封装对象状态变化，支持状态保存、恢复。
- 解释器模式：注重封装特定领域变化，支持领域问题的频繁变化，将特定领域的问题表达为某种语法规则下的句子，然后构建一个解释器来解释这样的句子，从而达到解决问题的目的。

5.1 模板方法模式

在现实生活中，有论文模板，简历模板等。在现实生活中，模板的概念是给定一定的格式，然后其他所有使用模板的人可以根据自己的需求去实现它。同样，模板方法也是这样的。

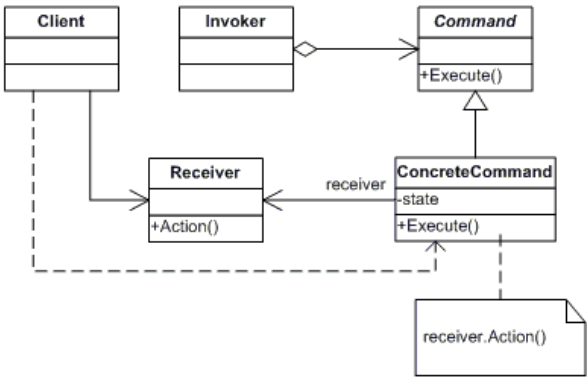
模板方法模式是在一个抽象类中定义一个操作中的算法骨架，而将一些具体步骤实现延迟到子类中去实现。模板方法使得子类可以不改变算法结构的前提下，重新定义算法的特定步骤，从而达到复用代码的效果。具体的结构图如下所示。



以生活中做菜为例子实现的模板方法结构图

5.2 命令模式

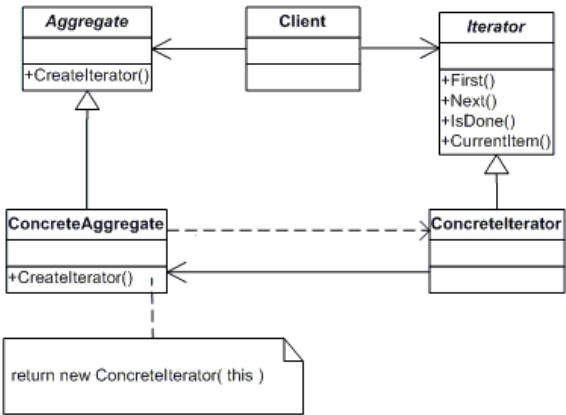
命令模式属于对象的行为模式，命令模式把一个请求或操作封装到一个对象中，通过对命令的抽象化来使得发出命令的责任和执行命令的责任分隔开。命令模式的实现可以提供命令的撤销和恢复功能。具体的结构图如下所示。



5.3 迭代器模式

迭代器模式是针对集合对象而生的，对于集合对象而言，必然涉及到集合元素的添加删除操作，也肯定支持遍历集合元素的操作，此时如果把遍历操作也放在集合对象的话，集合对象就承担太多的责任了，此时可以进行责任分离，把集合的遍历放在另一个对象中，这个对象就是迭代器对象。

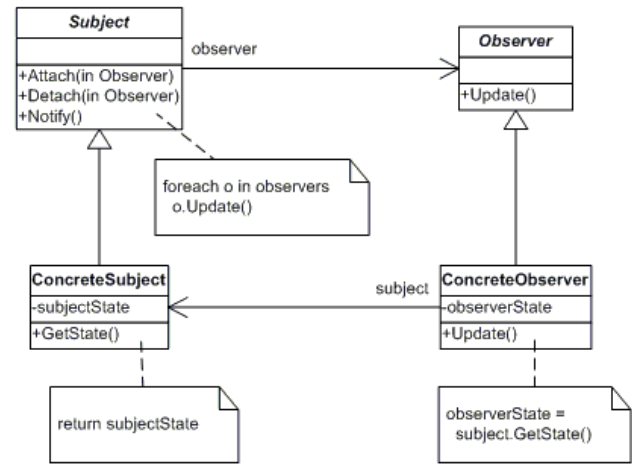
迭代器模式提供了一种方法来顺序访问一个集合对象中各个元素，而又无需暴露该对象的内部表示，这样既可以做到不暴露集合的内部结构，又可以让外部代码透明地访问集合内部元素。具体的结构图如下所示。



5.4 观察者模式

在现实生活中，处处可见观察者模式，例如，微信中的订阅号，订阅博客和QQ微博中关注好友，这些都属于观察者模式的应用。

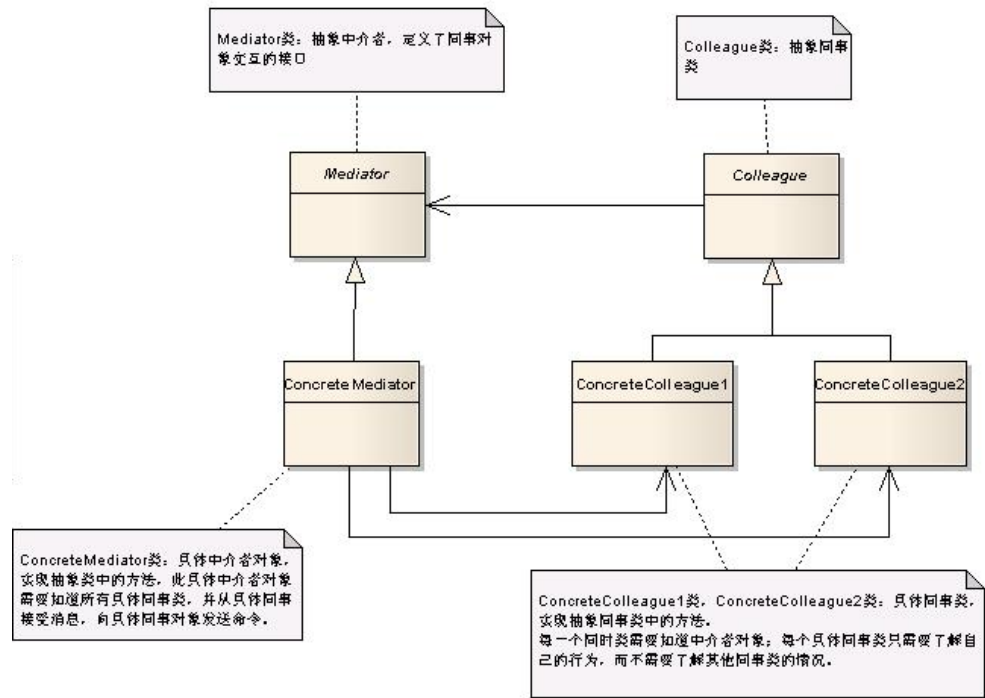
观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象，这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己的行为。具体结构图如下所示：



5.5 中介者模式

在现实生活中，有很多中介者模式的身影，例如QQ游戏平台，聊天室、QQ群和短信平台，这些都是中介者模式在现实生活中的应用。

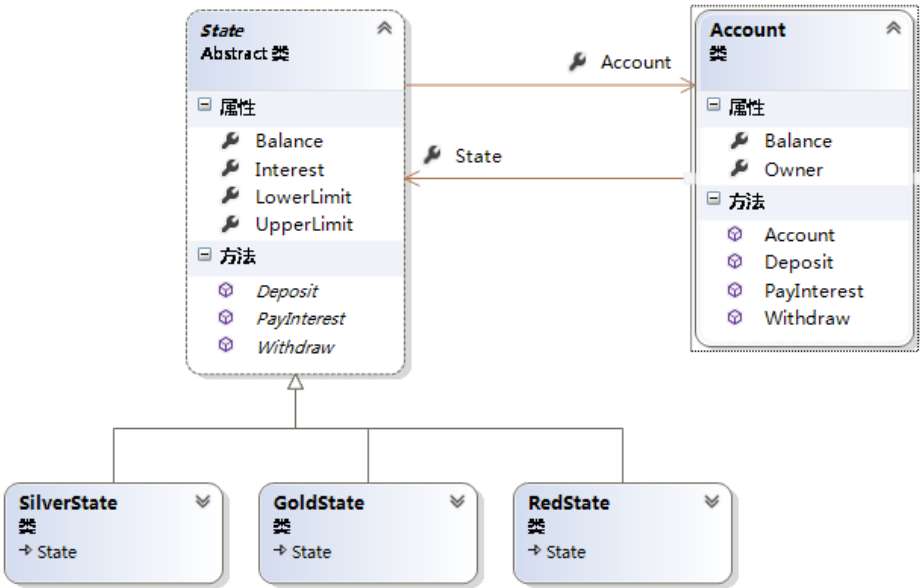
中介者模式，定义了一个中介对象来封装一系列对象之间的交互关系。中介者使各个对象之间不需要显式地相互引用，从而使耦合性降低，而且可以独立地改变它们之间的交互行为。具体的结构图如下所示：



5.6 状态模式

每个对象都有其对应的状态，而每个状态又对应一些相应的行为，如果某个对象有多个状态时，那么就会对应很多的行为。那么对这些状态的判断和根据状态完成的行为，就会导致多重条件语句，并且如果添加一种新的状态时，需要更改之前现有的代码。这样的设计显然违背了开闭原则，状态模式正是用来解决这样的问题的。

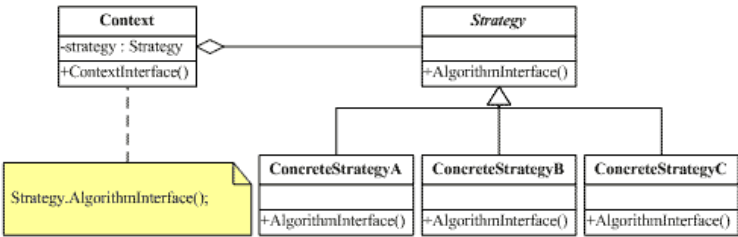
状态模式——允许一个对象在其内部状态改变时自动改变其行为，对象看起来就像是改变了它的类。具体的结构图如下所示：



5.7 策略模式

在现实生活中，中国的所得税，分为企业所得税、外商投资企业或外商企业所得税和个人所得税，针对于这3种所得税，每种所计算的方式不同，个人所得税有个人所得税的计算方式，而企业所得税有其对应计算方式。如果不采用策略模式来实现这样一个需求的话，我们会定义一个所得税类，该类有一个属性来标识所得税的类型，并且有一个计算税收的CalculateTax()方法，在该方法体内需要对税收类型进行判断，通过if-else语句来针对不同的税收类型来计算其所得税。这样的实现确实可以解决这个场景，但是这样的设计不利于扩展，如果系统后期需要增加一种所得税时，此时不得不回去修改CalculateTax方法来多添加一个判断语句，这样明白违背了“开放——封闭”原则。此时，我们可以考虑使用策略模式来解决这个问题，既然税收方法是这个场景中的变化部分，此时自然可以想到对税收方法进行抽象，这也是策略模式实现的精髓所在。

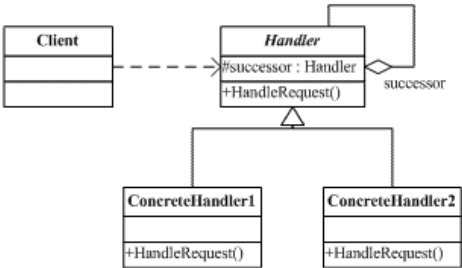
策略模式是对算法的包装，是把使用算法的责任和算法本身分割开，委派给不同的对象负责。策略模式通常把一系列的算法包装到一系列的策略类里面。用一句话概括策略模式就是——“将每个算法封装到不同的策略类中，使得它们可以互换”。下面是策略模式的结构图：



5.8 责任链模式

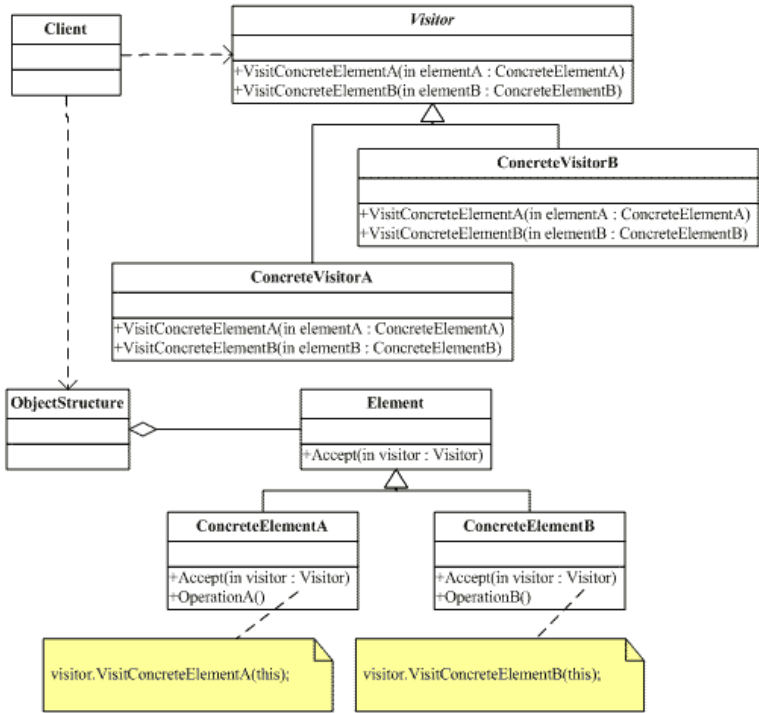
在现实生活中，有很多请求并不是一个人说了就算的，例如面试时的工资，低于1万的薪水可能技术经理就可以决定了，但是1万~1万5的薪水可能技术经理就没这个权利批准，可能需要请求技术总监的批准。

责任链模式——某个请求需要多个对象进行处理，从而避免请求的发送者和接收之间的耦合关系。将这些对象连成一条链子，并沿着这条链子传递该请求，直到有对象处理它为止。具体结构图如下所示：



5.9 访问者模式

访问者模式是封装一些施加于某种数据结构之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构则可以保持不变。访问者模式适用于数据结构相对稳定的系统，它把数据结构和作用于数据结构之上的操作之间的耦合度降低，使得操作集合可以相对自由地改变。具体结构图如下所示：



5.10 备忘录模式

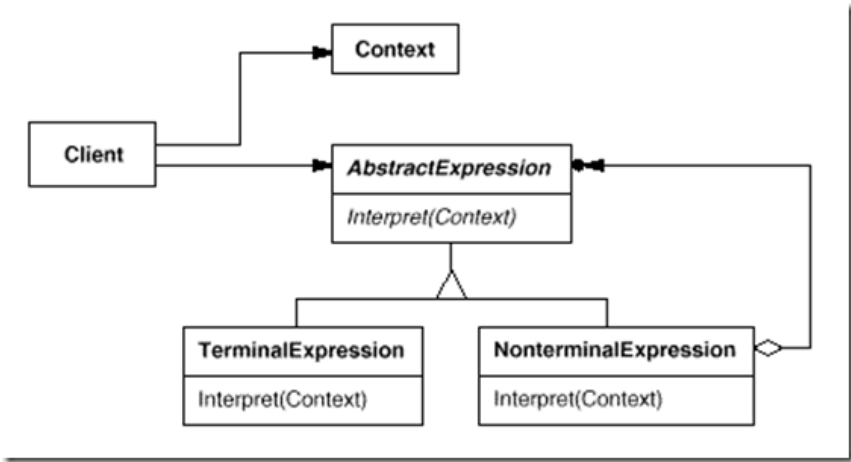
生活中的手机通讯录备忘录，操作系统备份点，数据库备份等都是备忘录模式的应用。备忘录模式是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样以后就可以把该对象恢复到原先的状态。具体的结构图如下所示：



5.11 解释器模式

解释器模式是一个比较少用的模式，所以我自己也没有对该模式进行深入研究，在生活中，英汉词典的作用就是实现英文和中文互译，这就是解释器模式的应用。

解释器模式是给定一种语言，定义它文法的一种表示，并定义一种解释器，这个解释器使用该表示来解释器语言中的句子。具体的结构图如下所示：



六、总结

23种设计模式，其实前辈们总结出来解决问题的方式，它们追求的宗旨还是保证系统的低耦合高内聚，指导它们的原则无非就是封装变化，责任单一，面向接口编程等设计原则。之后，我会继续分享自己WCF的学习过程，尽管博客园中有很多WCF系列，之前觉得没必要写，觉得会用就行了，但是不写，总感觉知识不是自己的，感觉没有深入，所以还是想写这样一个系列，希望各位博友后面多多支持。

PS： 很多论坛都看到初学者问，WCF现在还有没有必要深入学之类的问题，因为他们觉得这些技术可能会过时，说不定到时候微软又推出了一个新的SOA的实现方案了，那岂不是白花时间去深入学了，所以就觉得没必要深入去学，知道用就可以了。对于这个问题，我之前也有这样同样的感觉，但是现在我觉得，尽管WCF技术可能会被替换，但深入了解一门技术，重点不是知道

一些更高深API的调用啊，而是了解它的实现机制和思维方式，即使后面这个技术被替代了，其背后机制也肯定是相似的。所以深入了解了一个技术，你就会感觉新的技术熟悉，对其感觉放松。并且，你深入了解完一门技术之后，你面试时也敢说你好掌握了这门技术，而不至于说平时使用的很多，一旦深入问时却不知道背后实现原理。这也是我要写WCF系列的原因。希望这点意见对一些初学者有帮助。

如果您认为这篇文章还不错或者有所收获，您可以通过[右边的“打赏”功能](#) 打赏我一杯咖啡【物质支持】的【[店长推荐](#)】按钮【精神支持】，因为这两种支持都是我继续写作，分享的最大动力。

支付宝扫一扫，向我付款



微信扫一扫转账



打赏给我

分类: [跟我一起学C# 设计模式](#),[Index文章索引](#)

标签: [设计模式总结](#)


好文要顶

关注我

收藏该文







[Learning hard](#)
[关注 - 168](#)
[粉丝 - 2572](#)

81

1

荣誉: [推荐博客](#)
[+加关注](#)

« 上一篇: [C#设计模式\(23\)——备忘录模式（Memento Pattern）](#)
» 下一篇: [跟我一起学WCF\(1\)——MSMQ消息队列](#)

posted @ 2014-09-29 00:28 Learning hard 阅读(17416) 评论(53) 编辑 收藏

[< Prev](#)

[1](#)

[2](#)

评论列表

#51楼	2016-05-08 15:25	KMSFan	马克	支持(0) 反对(0)
#52楼	2016-06-01 08:03	Billy King	学会了这些 可以做架构师了么？	支持(0) 反对(0)
#53楼	2016-06-21 22:42	BigBar	mark	支持(0) 反对(0)

[< Prev](#)

[1](#)

[2](#)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

最新IT新闻：

- 苹果供应商三倍工资激励员工过年加班
- 富士康机器人战略进展顺利：部分厂区几乎完全实现自动化
- 展望VR音乐的未来：有线还是无线？
- 连胜柯洁等中日韩高手的是AlphaGo升级版？古力悬赏10万无人领走
- 暴风影音擅自直播央视春晚被诉侵权 遭索赔300万元
- » 更多新闻...

最新知识库文章：

- 写给未来的程序媛
- 高质量的工程代码为什么难写
- 循序渐进地代码重构
- 技术的正宗与野路子
- 陈皓：什么是工程师文化？
- » 更多知识库文章...

Copyright ©2017 Learning hard