设计之道

张逸 **著**

目 录

设计，看上去很美

设计没有标准，模式充满变化，我们对设计与模式的探讨，就是希望能从没有标准的设计中体验设计的乐趣，从充满变化的模式中寻求问题的解决之道。

我这里所谓“设计没有标准”，其实并非没有标准，现实是设计的标准实在太多了。我们都希望找到最好的设计方案，然而什么是最好，每个人都有自己的“哈姆雷特”。满足客户需求的设计就是最好的，这个结论我想不会有人反对，前提是，怎样通过设计来满足客户需求？

计划的设计和演进的设计

通常来说，软件设计不外乎两种方式：计划的设计和演进的设计。很多人看来，计划的设计更符合工程学的理念。如果你要建一间茅屋，那么你只需夯好土墙，再胡乱堆放一些茅草置于屋顶之上就可以了。然而，如果要你建一座苏州的拙政园，就必须事先有计划的设计了。哪里应该堆放假山，哪里应该开辟池塘，亭子的形状，院落的分布，乃至于园内的一花一木，无不需要独具匠心。软件设计也是如此，且过之而无不及。接手项目的时候，首先考虑的不是编码，而是考虑整个系统的架构，根据需求考虑系统中的重大问题。模块的功能，模块间的关系和系统分布的层次，都需要匠心独运，从一个抽象的层面来考虑。

演进的设计恰好与之相反，它是一种渐进的过程。它并不要求前期的设计有多么的完美，实现的需求有多么的完整，你只需要把现阶段考虑的问题编码实现就可以了，随着演进的深入，编码也会随之而修正，最后设计会逐渐丰满起来，经过一系列的方法，最后的设计也渐趋完美。

你也许会认为“演进的设计”如此的简陋与平庸。没有计划，只会令设计一团遭。但我需要提醒你的是，虽然都是工程学，软件的设计并没有建筑设计那么简单。因为，你很难在设计之初，考虑到客户的全部需求，甚至于实现未来的扩展。在设计一开始，你能确信：

你对客户的需求都理解了吗？

你能确定客户的需求不再变化吗？

你设计的软件架构真的能满足需求吗？

是的，你无法给出肯定的回答。总之我在这里不是想说服每个人，要采取哪一种设计方式。事实上，我也面临抉择的困难。

过度设计，还是简单设计？

Kent 在《解析极限编程——拥抱变化》中为简单系统制定了四个评价标准，依次为（最重要的排在最前面）：

通过所有测试；

体现所有意图；

避免重复；

类或者方法数量最少；

这些标准写出来简单，要根据这个标准来实现，就不是那么容易的事了。我相信，软件设计人员都希望自己的设计尽可能简单。然而，在设计时，我们不仅仅要考虑软件的功能，

我们还要考虑软件的性能、扩展性，模块间的耦合关系，系统的稳定、部署和更新，版本的管理，系统的安全，界面的友好程度。要想简单，何其之难！

Do the simplest thing that could possibly work! 这是 XP 人士大声疾呼的口号，我也举双手赞成。问题是，我们需要让简单的事情，同时又有效。很多人在设计时，并不满足于实现眼前的功能。看到加法，他们可能还会想到乘法；虽然目前的需求是整数，他们可能想到今后可能会扩展到实数，甚至于复数。他们希望能利用某种设计，使其具有更好的扩展性。从眼前的需求来看，可能是过度设计；然后对于未来，这个设计才是最完美的方案。

问题不在于设计是否过度，关键还是在于设计的理念。是只做目前需要的事，还是未雨绸缪，想好今后的功能扩展？这个问题的答案还需要实际的项目开发来检验，根据不同的需求，答案会因此而异。

需要设计模式吗？

答案是肯定的，但你需要确定的是模式的应用是否过度？我得承认，世界上有很多天才的程序员，他可以在一段代码中包含 6 种设计模式，也可以不用模式而把设计做得很好。但我们的目标是追求有效的设计，而设计模式可以为这个目标提供某种参考模型、设计方法。我们不需要奉 GOF 的设计模式为圭臬，但合理的运用设计模式，才是正确的抉择。

很多人看过 GOF 的《Design Patterns》，对这 23 种模式也背得滚瓜烂熟。但重要的不是你熟记了多少个模式的名称，关键还在于付诸实践的运用。为了有效地设计，而去熟悉某种模式所花费的代价是值得的，因为很快你会在设计中发现这种模式真的很好，很多时候它令得你的设计更加简单了。

其实在软件设计人员中，唾弃设计模式的可能很少，盲目夸大设计模式功用的反而更多。言必谈“模式”，并不能使你成为优秀的架构师。真正出色的设计师，懂得判断运用模式的时机。

还有一个问题是，很多才踏入软件设计领域的人员，往往对设计模式很困惑。对于他们来说，由于没有项目的实际经验，OO 的思想也还未曾建立，设计模式未免过于高深了。其实，即使是非常有经验的程序员，也不敢夸口对各种模式都能合理应用。

重构是必然的！

既然我们无法给出一个完美的设计方案，因为客户的需求总是变化的，重构也就成为必然。问题是，这样没有添加任何功能的重构，你是否愿意为此付出精力、时间去完成。当客户要求的 Deadline 将要到来的时候，你还认为你的重构工作是必要的吗？

有时候，软件设计常常身不由己。然而，纯从技术的角度来看，重构非但必然，而且重要。既然我们都明白，复杂的未尝就是好的，简单的也不一定是容易的。要保持你的设计尽可能的简单，可能你还需要时时借助重构的利器，来“改善你既有代码的设计”。

对于重构，Martin Fowler 给出了很多条款。这些条款并不是政治课本的教条，也不是“日月神教”的神奇咒语，念着它们就可以防身。这些条款确实很重要，但你需要的是学会他后，然后忘记他，就象张无忌学太极拳那样。我不是故弄玄虚，事实上只有这样，重构的精神才能完全融入到你的设计中。

UML 重要吗？

我现在看一个设计方案的时候，更希望先看看 UML 图，然后再看文档的实际描述。如

果让我读一段代码，我希望能先看看类图，或许更容易理解代码的含义。UML 在 OO 世界里像是世界语，它便于程序员间的交流，让别人更容易理解你的意图。同时，在设计 UML 图的过程中，也是一种对思路的清理，对客户需求的把握，设计思想的跟踪。

UML 是一种基于对象的统一建模语言，它能够为系统设计提供清晰直观的设计。在面向对象世界里，UML 的地位弥足轻重，甚至被称为是软件设计的一场革命。对于有计划的设计，UML 的价值就体现得淋漓尽致。如果我们要清晰地表现模块的功能，模块间的关系和系统分布的层次，使用 UML 可以使设计师减少很多麻烦，同时降低了语义描述的二义性。然而，如果我们在做演进的设计时，UML 还有那么重要吗？我们只需要对眼前的需求进行编码、测试，然后重构。可能我们只需要在 Pair Team 中讨论设计方案，在预定技术框架内探讨实现的可能和细节。我们完全可以抛开 UML 繁琐而死板的设计，毕竟最能忠实体现设计思想的，不是文档，不是用例图或是什么类图，而是代码。

那么，有多少人是这样想的？

TDD、单元测试和其他

软件的生命是什么，是质量！而保证质量的唯一方法，就是测试。传统的软件开发过程，强调首先进行需求分析，再从需求分析中抽象出概要设计，进而作出详细设计，然后编码，最后才是测试以验证代码的正确性。而测试驱动开发（TDD）改变了编码的过程，开发仅仅包括三方面的活动：编写测试用例，编码并进行测试，重构代码以消除重复代码使其更简单、更灵活、更容易理解。通过测试来驱动开发，听起来是那么的离经叛道，然而实施起来，又是那么合理、正确和简单，前提是：我们不能在一开始就获得正确的设计！TDD 避免了对不完整需求造成的不成熟的设计。通过单元测试，保证了代码的正确性与高质量；通过重构，使设计更加简单、灵活。

设计，由你掌握

前言：XP 中有个准则，就是只做目前你需要做的。例如，我需要加法运算，你就没有必要实现乘法运算，因为这不是客户需要的。因此，在开发中，我们可以不去考虑程序对于未来的扩展性。“简单最好！”那么，是否就不需要设计了呢？至于设计模式，是否也可以不去了解了呢？答案至少是否定的。因为客户的需求是“与时俱进”的，现在不实现，并不等于今后不实现。在实现中，不管是重构，还是重新设计，通过应用设计模式，能令你如虎添翼。关键不在于设计模式是否重要，而在于你怎么应用它，以及选择什么样的时机。总而言之，设计，由你掌握！

一、从需求开始

在我们的项目中，作费用结算的时候，客户要求将该过程与结果写入到日志文件中。不过他们的要求很善良，只需要知道日志记录结算开始与结束的时间而已。是的，按照 XP 的理念，我们只需要做好客户需要的事情就 OK 了。既然是这样，事情就好办，代码轻易而举就实现了：

```
public class Fee
{
    //结算程序将调用数据层的相关方法，访问数据库；
    //为简单起见，我用累加数取代；
    public double SettleFee(double money,int records)
    {
        double result = 0.0;
        //我用控制台输出来表示写日志；
        Console.WriteLine("Start settling fee at {0}",DateTime.Now);
        for (int i=0;i<records;i++)
        {
            result+=money;
        }
        Console.WriteLine("Settling fee finished at {0}",DateTime.Now);
        return result;
    }
}
```

写好这个，还差点什么？不错，我们还需要为 Fee 类撰写相应的测试代码，做好单元测试。可能对于传统的程序员来说，更喜欢的是在编码完成后，再根据测试计划编写测试样例，最后测试。但敏捷开发的要求却是测试先行，单元测试是必不可少的环节。不过，我认为单元测试毕竟只是一种手段。我们在实际的项目开发中，对于单元测试不可拘泥教科书的要求，按部就班地一步一步进行；而应该根据实际情况，比如开发者对语言的掌握程度，对设计的理解等等，来决定你单元测试的步骤，乃至于重构的步伐。

```
[TestFixture]
public class TestFee
```

```
{
    [Test]
    public void Settle()
    {
        Fee fee = new Fee();
        Assert.IsNotNull(fee);
        Assert.AreEqual(6,fee.SettleFee(2.0,3));
    }
}
```

在 NUnit 中打开这个测试类，并运行。毫无疑义，你会看到测试的绿灯全部都亮了。你可以在 NUnit 中看看控制台输出的结果。自然你也可以在 AreEqual()方法中，故意将预期的值设置错误，来看看运行 NUnit 是什么情况，以及出现的错误提示信息。不过，这些都不是本文关注的重点。

二、当需求改变了

在 XP 中，客户的重要是举足轻重的。在客户提出需求的时候，你需要和他尽可能地沟通，并保证意见最后要达成一致。然而客户对产品的理解可能有时候会出现偏差，也许有时候对方的要求也会随着产品的应用而逐渐发生改变。所幸的是，这一次需求的改变，发生在项目开发过程中，且是在你和他结对交流的时候，最终发现的缺陷。因为客户认为，这个日志过于简单了，并不利于今后对产品的维护。我得承认这是一个好的要求。

事实上，日志记录得越详细，对于开发人员自己也是有好处的。最后，我们决定，日志不仅仅要记载结算的起止时间，还应该记载可能会出现的错误信息，最好还能记载这个结算的过程代码，比如我们执行的是哪一个存储过程，读取了哪些表的数据，包括这些表的字段。

然而有个不利的因素是，这个费用结算的过程可能会很频繁的使用。如果写入的日志太复杂了，会否影响产品的性能？而且频繁写日志的话，日志文件会不会越来越大？如果我们这个产品已经非常健壮，还有必要去记载这些信息吗？毕竟有很多信息，对于普通用户而言，并没有实际用处，反而干扰了他有效获取日志的有用信息。

所以后来我们想到一个方法，就是将日志进行分级，从最简单到最详尽。用户在进行费用结算的时候，可以根据自身需要，选择日志的级别。无疑，这是一个令人满意的策略。

三、如果不熟悉设计模式

假设我们的开发人员对于设计模式一概不知，经过分析客户的需求，他会直接了当的做出如此的解决方案。首先定义三种级别的日志：SimplestLog，NormalLog，DetailedLog。SimplestLog 只记录结算的起止时间和耗费的时间，同时还要记录结算后的结果。NormalLog 则除此之外，还要记录可能会出现的错误信息。而 DetailedLog 最详尽，它不仅包含了 NormalLog 记录的信息，还包括记录结算的实现方法，如用到的存储过程，数据表和相应的字段。

我们最初设想为这三种级别建立三个不同的私有方法，然后在 SettleFee()方法中，引入一个日志级别参数，然后根据日志级别的值，决定调用哪一个私有方法，例如：

```
private void WriteSimplestLog();
private void WriteNormalLog();
private void writeDetailedLog();

public enum LogLevelEnum{Simple=0,Normal,Detail};

public double SettleFee(double money,int records,LogLevelEnum logLevel)
```

```
{
    switch (logLevel)
    {
        case LogLevel.Simple:
            WriteSimplestLog();
            break;
        case LogLevel.Normal:
            WriteNormalLog();
            break;
        case LogLevel.Detail:
            writeDetailedLog();
            break;
    }
    for (int i=0;i<records;i++)
    {
        result+=money;
    }
}
```

不用说，我们的程序员遇到麻烦了。因为在记录日志信息的时候，可能会在结算的前后来进行。也就是说，结算的那一段代码必须放到记录日志的方法中，才可以实现。幸运的是，我们的程序员应该还具备重构的知识，他决定把结算的那一段代码专门抽取出来，形成一个单独的方法，再放到日志方法中调用。“Extract Method”，不是吗？很聪明的做法。

好吧，我们来实现它吧，看看会是怎样？

首先，实现专门的结算方法：

```
private double Settle(double money,int records)
{
    double result = 0.0;
    for (int i=0;i<records;i++)
    {
        result+=money;
    }
    return result;
}
```

再来实现日志方法：

```
private void WriteSimplestLog()
{
    DateTime startTime,endTime;
    startTime = DateTime.Now;
    Console.WriteLine("Start settling fee at {0}",startTime);
    result = Settle(money,records);
    endTime = DateTime.Now;
    Console.WriteLine("Settling fee finished at {0}",endTime);
    TimeSpan wasted = endTime - startTime;
    Console.WriteLine("It wasted time {0}",wasted);
```

```
        Console.WriteLine("The result is {0}",result);
    }
```

在这个方法中，result 是 Fee 类中的一个私有变量，用来保存结算后的结果。假设不使用这个变量，而是在方法中引入局部变量，那么 WritesimplestLog()方法就必须返回 double 类型了，这个设计可够糟糕的！同样的，money 和 records 也应该是通过私有变量传递值，否则这个日志方法就必须带上这两个参数了。

接着实现下面两个方法。我们已经注意到根据日志级别的不同，最详尽的日志内容总是包含了其低一级日志的内容。并且，后两级日志没有包括性能的记录，因此记录的日志并未要求必须出现在结算方法的前后。

```
    private void WriteNormalLog()
    {
        try
        {
            WriteSimplestLog();
            Console.WriteLine("Settling operation succeed!");
        }
        catch (Exception ex)
        {
            Console.WriteLine("The error occured while settling the fee.");
            Console.WriteLine("The error is " + ex.Message);
        }
    }
    private void WriteDetailedLog()
    {
        WriteNormalLog();
        Console.WriteLine("The StoreProcedure whick was invoked is SpSettleFee.");
        Console.WriteLine("Data table is: UserFee, OnLineRecord.");
    }
```

剩下的代码就简单了：

```
    private double result = 0.0;
    private doulbe money = 0.0;
    private int record = 0;

    public double SettleFee(double money,int records,LogLevelEnum logLevel)
    {
        this.money = money;
        this.record = record;
        switch (logLevel)
        {
            case LogLevel.Simple:
                WriteSimplestLog();
                break;
            case LogLevel.Normal:
                WriteNormalLog();
```

```
                break;
            case LogLevel.Detail:
                writeDetailedLog();
                break;
        }
        return result;
    }
```

嘿嘿，看起来还不错！

当然，与之相应的测试代码也要发生改变：

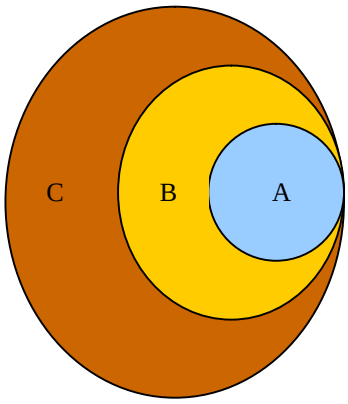
```
[Test]
public void Settle()
{
    Fee fee = new Fee();
    Assert.IsNotNull(fee);
    Assert.AreEqual(6,fee.SettleFee(2.0,3,LogLevelEnum.Simple));
    Assert.AreEqual(6,fee.SettleFee(2.0,3,LogLevelEnum.Normal));
    Assert.AreEqual(6,fee.SettleFee(2.0,3,LogLevelEnum.Detail));
}
```

四、问题又出现了

我们程序员都应该有这个信仰，就是：简单最好。我不喜欢那些常常卖弄自己水平的人，仅仅为了一个简单的要求，却故意把代码弄得非常复杂，以为卖弄高深就是学问。其实不然。我倾向于简单的算法，即使它的性能稍差。因为性能差，我们还可以通过提升硬件等多种方式来解决；而如果整个项目都充斥着难于理解的算法，想一想，如果写代码的“牛人”走了，而目前又需要对程序做改进。那么，项目的后任者，在 deadline 的压力下，面对这一大堆“高深”的算法，会是怎样的抓狂！？

所以，从目前来看，以上实现的代码并没有什么不合适的地方。简单易懂，也完成了客户的需求。不过，很多时候事情并非尽如人意。客户的需求会随着对项目的跟进，而逐渐发生改变。

一周后，我们的客户提出了新的要求。首先，他希望这个日志功能，能够展现它更灵活的一面。不是死板的从最简单到最详尽，而是根据日志记载的内容，任意灵活的组合。原来的日志层次如图一：



注：

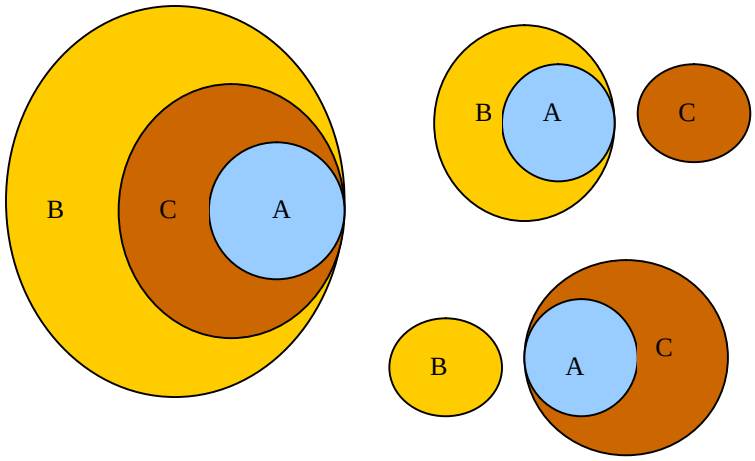
A：SimplestLog

B：NormalLog

C：DetailedLog

图一：最简单的日志组合

而根据现在的需求，可能会有多种组合：



注：
A：SimplestLog
B：NormalLog
C：DetailedLog

图二：灵活的组合，日志变得多种多样

我们算一算，如果按照前面的方式来实现新的需求，可能会写出多少个方法？此时的你应该怎样办呢？

或许应该将逻辑抽象出来，为日志建立一个基类；然后根据实际的需求派生不同的子类。在调用的时候，可以通过多态的方式，决定调用何种具体的日志对象方法。

但是问题接踵而至。首先是写日志方法和费用结算方法如何结合起来？尤其是 A 类日志，写日志的时候必须是在费用结算的前后，以确定结算的起止时间。也许，我们可以考虑将该方法分离为两个方法 BeforeWriteLog()和 AfterWriteLog()。那么与之对应的，凡是和 A 类日志有关的其他日志，也必须实现这种分离策略。再想想继承的子类，根据日志这种灵活的组合，我们需要创建多少个子类对象。一旦需求再增加呢？这个无底洞我是不愿意去跳的。

而客户的要求并没有结束。他还需要在与费用有关的其他方法中，也实现日志的功能。例如费用查询。由于之前的策略是在日志方法中包含费用结算的功能。如果需要记录费用查询的日志，岂不是要为它再建立不同的日志方法？

总之，如果你不了解设计模式的话，你可能会非常棘手。你会用尽所有已掌握的知识，通过你对 OOP 的理解，使用接口，继承和聚合，最后你可能会发现你碰壁了。乐观的是，你最终解决了问题。可是看看解决方案呢？要么拙劣不忍目睹，要么幸运的是你采用了正确的策略。你已经达到了 GOF 的水平，自己创造出我们应该正确使用的模式了。不过，可惜的是，你会沮丧地听到，有人会告诉你，你采用的其实就是设计模式中的 Decorator 模式。与其这样，为什么不好好地学习一下设计模式呢？

五、结果完全不一样了

如果你已经熟悉了设计模式的话，面对客户提出的新的需求，你就会很快获得完美的解决方案了。

分析一下。虽然日志的级别会非常之多，但其根本的功能却是一致的，那就是所有的日志信息都是费用结算（自然也包括费用查询）的包装而已。形象地说，日志就好比油漆工人，而费用结算就是一间房子，需要油漆工人来给墙壁粉刷上美丽的色彩。如此而已。

在使用 Decorator 模式来实现如上需求之前，我想表明自己的态度：

1、 设计模式的重要性已经不言而喻了；

2、 不要为了模式而去学习模式，设计模式必须和项目实际开发结合；

3、 如果目前的需求很简单，不用设计模式并不是一个坏的选择；

4、 因为我们有重构；

5、 但必须记住，重构的每一步，需要以单元测试来保证；

6、 你必须深入理解设计模式，否则当需求复杂之后，你会束手无策；

7、 设计模式是人创造出来的，但既然已经有了前人的成果，为什么不用？

写到这里，诸位已经可以结束本文的阅读了。不过我还得继续下去，作业没有做完，不能交卷。

六、大结局

因为现在的需求比较复杂了，所以你在重构每一步时，必须小心翼翼。别忘了单元测试，有了它，才可以保证你的正确无误。

首先，利用“Extract Interface”原则，为装饰的对象 Fee 类 Extract 一个接口，并让 Fee 类实现它：

```
public interface IFee
{
    double SettleFee(double money, int records);
}
public class Fee:IFee {}
```

当然，我们需要把 SettleFee()方法恢复成原来的模样。记住这个过程仍然需要小心翼翼。因为，在实现这一步时，可能已经离最初的简单实现已经有一周的时间了。所以，再恢复原样的过程中，我希望仍然不要放弃使用单元测试。当然在这里，我为了行文简洁，省略了这些过程。

修改测试代码，然后在 NUnit 中运行它：

```
[Test]
public void Settle()
{
    IFee fee = new Fee();
    Assert.IsNotNull(fee);
    Assert.AreEqual(6,fee.SettleFee(2.0,3));
}
```

现在来分析日志。根据对前面 A 类、B 类、C 类日志的分析。我们不再从是否详尽的角度来分类日志，而是从日志的内容或者说日志实现的功能来分类。我们可以将日志分为基本日志类、错误日志类、实现日志类三种。

基本日志类：实现日志的基本功能，包括费用结算的耗时和结算后的结果。

错误日志类：记录可能会出现的错误消息。

实现日志类：将费用结算的具体实现记录下来，便于以后对于产品的维护。

因为日志就是 Decorator 模式的油漆工，它们都需要具备包装费用结算的功能，我为它们定义了一个共同的抽象类：

```
public abstract class LogDecorator
{
    private IFee decoratee;
    public LogDecorator(IFee decoratee)
    {
```

```
            this.decoratee = decoratee;
        }
        public IFee Decoratee
        {
            get {return this.decoratee;}
        }
    }
```

基本日志类、错误日志类、实现日志类都继承该抽象类。注意抽象类的自定义构造函数，它是实现装饰功能的关键。该构造函数负责传递一个被装饰的对象进来，并赋给属性Decoratee。这个初始化的过程，就好比刷油漆的刷子，对于所有油漆工人来说，都应该是一样的，只是他们刷的油漆不同而已。

要装饰 Fee 类，仅仅依靠构造函数来传递被装饰对象还不够。我们还需要把原来 Fee 类所做的工作，转移到装饰类中，如此才能完成装饰的功用。所以，这三个日志类，不仅要继承 LogDecorator 类，还需要实现 IFee 接口，即与 Fee 类实现同一个接口。

首先是基本日志类：

```
    public class BasicLogDecorator:LogDecorator,IFee
    {
        public BasicLogDecorator(IFee decoratee):base(decoratee){}

        public double SettleFee(double money,int records)
        {
            DateTime startTime,endTime;
            startTime = DateTime.Now;
            Console.WriteLine("Start settling fee at {0}",startTime);
            double result = Decoratee.SettleFee(money,records);
            endTime = DateTime.Now;
            Console.WriteLine("Settling fee finished at {0}",endTime);
            TimeSpan wasted = endTime - startTime;
            Console.WriteLine("It wasted time {0}",wasted);
            Console.WriteLine("The result is {0}",result);
            return result;
        }
    }
```

做到这一步时，先别急着去实现另外两个类。我们应该先做单元测试。修改单元测试代码：

```
    [Test]
    public void SettleBasicLog()
    {
        IFee fee = new Fee();
        IFee basicLogFee = new BasicLogDecorator(fee);
        Assert.IsNotNull(fee);
        Assert.IsNotNull(basicLogFee);
        Assert.AreEqual(6,fee.SettleFee(2.0,3));
```

```
        Assert.AreEqual(6, basicLogFee.SettleFee(2.0,3));
    }
```

不过这个单元测试代码似乎有点乱，我们应该根据具体的实现，对测试方法分类，同时将类对象的初始化放到[SetUp]中。因此，新的测试代码如下：

```
    [TestFixture]
    public class TestFee
    {
        [TestFixture]
            public class TestFee
        {
            private IFee fee;
            [SetUp]
            public void Init()
            {
                fee = new Fee();
            }
            [Test]
            [Category("SettleWithoutLog")]
            public void Settle()
            {
                Assert.IsNotNull(fee);
                Assert.AreEqual(6,fee.SettleFee(2.0,3));
            }
            [Test]
            [Category("SettleWithBasicLog")]
            public void SettleBasicLog()
            {
                IFee basicLogFee = new BasicLogDecorator(fee);
                Assert.IsNotNull(basicLogFee);
                Assert.AreEqual(6,basicLogFee.SettleFee(2.0,3));
            }
            [TearDown]
            public void Dispose()
            {
                /*---*/
            }
        }
    }
```

通过 Category 对测试方法进行分类（也可以对测试类进行分类）。这样，我们就可以在 Nunit 中，根据测试的情况，选择要测试的分类，或 Exclude（排除）不测试的分类。我们运行一下，看 NUnit 的绿灯是否都亮了？测试通过后，就可以接着实现另外的日志类了。

```
    public class ErrorLogDecorator:LogDecorator,IFee
    {
        public ErrorLogDecorator(IFee decoratee):base(decoratee){}
```

```
    public double SettleFee(double money, int records)
    {
        try
        {
            double result = Decoratee.SettleFee(money,records);
            Console.WriteLine("Settling operation succeed!");
            return result;
        }
        catch (Exception ex)
        {
            Console.WriteLine("The error occured while settling the fee.");
            Console.WriteLine("The error is " + ex.Message);
            return 0;
        }
    }
}

public class ImplLogDecorator:LogDecorator,IFee
{
    public ImplLogDecorator(IFee decoratee):base(decoratee)
    {}
    public double SettleFee(double money, int records)
    {
        double result = Decoratee.SettleFee(money,records);
        Console.WriteLine("The StoreProcedure whick was invoked is SpSettleFee.");
        Console.WriteLine("Data table is: UserFee, OnLineRecord.");
        return result;
    }
}
```

当然每做一步改进后，都需要修改测试代码进行单元测试。最后的单元测试代码：

```
using System;
using NUnit.Framework;
using FeeManagement;

namespace UnitTest
{
    [TestFixture]
    public class TestFee
    {
        private IFee fee;

        [SetUp]
        public void Init()
        {
```

```
        fee = new Fee();
    }

    [Test]
    [Category("SettleWithoutLog")]
    public void Settle()
    {
        Assert.IsNotNull(fee);
        Assert.AreEqual(6,fee.SettleFee(2.0,3));
    }

    [Test]
    [Category("SettleWithBasicLog")]
    public void SettleBasicLog()
    {
        IFee basicLogFee = new BasicLogDecorator(fee);
        Assert.IsNotNull(basicLogFee);
        Assert.AreEqual(6,basicLogFee.SettleFee(2.0,3));
    }

    [Test]
    [Category("SettleWithErrorLog")]
    public void SettleErrorLog()
    {
        IFee errorLogFee = new ErrorLogDecorator(fee);
        Assert.IsNotNull(errorLogFee);
        Assert.AreEqual(6,errorLogFee.SettleFee(2.0,3));
    }

    [Test]
    [Category("SettleWithImplLog")]
    public void SettleImplLog()
    {
        IFee implLogFee = new ImplLogDecorator(fee);
        Assert.IsNotNull(implLogFee);
        Assert.AreEqual(6,implLogFee.SettleFee(2.0,3));
    }

    [Test]
    [Category("SettleWithBasic&ErrorLog")]
    public void SettleBasicErrorLog()
    {
        IFee basicLogFee = new BasicLogDecorator(fee);
        IFee errorLogFee = new ErrorLogDecorator(basicLogFee);
```

```
            Assert.IsNotNull(basicLogFee);
            Assert.IsNotNull(errorLogFee);
            Assert.AreEqual(6,errorLogFee.SettleFee(2.0,3));
        }

        [Test]
        [Category("SettleWithBasic&ImplLog")]
        public void SettleBasicImplLog()
        {
            IFee basicLogFee = new BasicLogDecorator(fee);
            IFee implLogFee = new ImplLogDecorator(basicLogFee);
            Assert.IsNotNull(basicLogFee);
            Assert.IsNotNull(implLogFee);
            Assert.AreEqual(6,implLogFee.SettleFee(2.0,3));
        }

        [Test]
        [Category("SettleWithAllLog")]
        public void SettleAllLog()
        {
            IFee basicLogFee = new BasicLogDecorator(fee);
            IFee implLogFee = new ImplLogDecorator(basicLogFee);
            IFee errorLogFee = new ErrorLogDecorator(implLogFee);
            Assert.IsNotNull(basicLogFee);
            Assert.IsNotNull(implLogFee);
            Assert.IsNotNull(errorLogFee);
            Assert.AreEqual(6, errorLogFee.SettleFee(2.0,3));
        }

        [TearDown]
        public void Dispose()
        {
            /*---*/
        }
    }
}
```

由于每一步都严格进行了单元测试，所以，我们对代码的正确性充满了信心。这也是单元测试的重要性及必要性所在。

七、真的结束了吗？

从测试代码中看出，目前的解决方案还存在一个问题，就是日志对象的创建。由于日志对象可能会根据不同的情况，组合成不同的对象。如果不采取相应的方法来解决对象创建的问题，可能会造成对象管理的混乱。因此，我们还有必要引入工厂模式，专门负责日志对象的创建。

我最初考虑在工厂方法中，将这些日志类型放到一个 Type[]数组中，然后再通过反射的

方式创建对象。然而，由于创建日志对象的组合会很麻烦，采用这样的设计，反而会有过度设计的嫌疑。（这也是我为什么在 Decorator 类中使用构造函数而非采用专门的方法来设置 Decoratee 对象的原因。）

所以，只需要直接根据日志的情况为其分别创建相关的工厂方法就可以了。

```
public class DecoratorFactory
{
    private static IFee fee = new Fee();
    public static IFee CreateFee()
    {
        return fee;
    }
    public static IFee CreateBasicLogFee()
    {
        IFee basicLog = new BasicLogDecorator(fee);
        return basicLog;
    }
    public static IFee CreateErrorLogFee()
    {
        IFee errorLog = new ErrorLogDecorator(fee);
        return errorLog;
    }
    public static IFee CreateImplLogFee()
    {
        IFee implLog = new ImplLogDecorator(fee);
        return implLog;
    }
    public static IFee CreateBasicErrorLogFee()
    {
        IFee basicLog = new BasicLogDecorator(fee);
        IFee errorLog = new ErrorLogDecorator(basicLog);
        return errorLog;
    }
    public static IFee CreateBasicImplLogFee()
    {
        IFee basicLog = new BasicLogDecorator(fee);
        IFee implLog = new ImplLogDecorator(basicLog);
        return implLog;
    }
    public static IFee CreateAllLogFee()
    {
        IFee basicLog = new BasicLogDecorator(fee);
        IFee implLog = new ImplLogDecorator(basicLog);
        IFee errorLog = new ErrorLogDecorator(implLog);
        return errorLog;
```

```
    }
}
```

然后再修改 NUnit 的测试代码：

```
[TestFixture]
public class TestFee
{
    private IFee fee;
    [SetUp]
    public void Init()
    {
        fee = DecoratorFactory.CreateFee();
    }
    [Test]
    [Category("SettleWithoutLog")]
    public void Settle()
    {
        Assert.IsNotNull(fee);
        Assert.AreEqual(6,fee.SettleFee(2.0,3));
    }
    [Test]
    [Category("SettleWithBasicLog")]
    public void SettleBasicLog()
    {
        IFee basicLogFee = DecoratorFactory.CreateBasicLogFee();
        Assert.IsNotNull(basicLogFee);
        Assert.AreEqual(6,basicLogFee.SettleFee(2.0,3));
    }
    [Test]
    [Category("SettleWithErrorLog")]
    public void SettleErrorLog()
    {
        IFee errorLogFee = DecoratorFactory.CreateErrorLogFee();
        Assert.IsNotNull(errorLogFee);
        Assert.AreEqual(6,errorLogFee.SettleFee(2.0,3));
    }
    [Test]
    [Category("SettleWithImplLog")]
    public void SettleImplLog()
    {
        IFee implLogFee = DecoratorFactory.CreateImplLogFee();
        Assert.IsNotNull(implLogFee);
        Assert.AreEqual(6,implLogFee.SettleFee(2.0,3));
    }
    [Test]
```

```
        [Category("SettleWithBasic&ErrorLog")]
        public void SettleBasicErrorLog()
        {
            IFee log = DecoratorFactory.CreateBasicErrorLogFee();

            Assert.IsNotNull(log);
            Assert.AreEqual(6,log.SettleFee(2.0,3));
        }
        [Test]
        [Category("SettleWithBasic&ImplLog")]
        public void SettleBasicImplLog()
        {
            IFee log = DecoratorFactory.CreateBasicImplLogFee();
            Assert.IsNotNull(log);
            Assert.AreEqual(6,log.SettleFee(2.0,3));
        }
        [Test]
        [Category("SettleWithAllLog")]
        public void SettleAllLog()
        {
            IFee log = DecoratorFactory.CreateAllLogFee();
            Assert.IsNotNull(log);
            Assert.AreEqual(6,log.SettleFee(2.0,3));
        }
        [TearDown]
        public void Dispose()
        {
            /*---*/
        }
    }
```

经过这么多阶段的修改和完善，目前看来解决方案已经比较完善了。如果在 Fee 类中还有其他的方法，需要日志功能，方法仍然大同小异。因为在 C#中可以同时实现多个接口，如果实现其他接口的类也要增加该日志功能，则日志的 Decorator 类同时还要去实现这个新的接口。好处是，你只需要修改这些实现，而调用的代码，却不用作大的修改了。因为要求提供日志功能的需求可能会不断增加，但只要日志的种类不变，用作装饰功能的日志对象个数就不会改变。

自然，本文讨论日志功能是完全站在 OOP 的角度来考虑的。如果引入 AOP 的思想，将日志看作是一个方面(Aspect)，那么对于客户而言，可能会更简单。但这已不是本文要讨论的问题了。

重构初体验

设计大师 Martin Fowler 在《重构——改善既有代码的设计》一书中，以其精妙的概括能力，彻底对重构技术作了全方位的总结。该书既具备大百科全书般提纲挈领的重构大纲，同时更通过实例展现了在软件设计中重构的魅力。

有感于重构艺术予我的震撼，我逐渐尝试在项目设计中开始重构之旅。在这个旅程中，存在尝试的犹豫和领悟的感动，然而最终却令我折服。如今，我希望能通过一个实际的例子，让读者也能初次体验重构的魅力。举例来说，我打算作一个容器，用来存放每个整数的阶乘结果。最初的设计是这样：

```
public class FactorialContainer
{
    public FactorialContainer()
    {
        factorialList = new ArrayList();
    }
    public FactorialContainer(int capacity)
    {
        m_Capacity = capacity;
        factorialList = new ArrayList(capacity);
    }
    private ArrayList factorialList;
    private int m_Capacity;
    public ArrayList FacotorialList
    {
        get {return factorialList;}
        set {factorialList = value;}
    }
    public int Capacity
    {
        get {return m_Capacity;}
        set {m_Capacity = value;}
    }
    public long this[int index]
    {
        get {return (long)factorialList[index];}
        set {factorialList[index] = value;}
    }
    public void Add()
    {
        long result = 1;
```

```
            int seed = factorialList.Count + 1;
            for (int i=1;i<=seed;i++)
            {
                result*=i;
            }
            factorialList.Add(result);
        }
        public void Clear()
        {
            factorialList.Clear();
        }
        public void RemoveAt(int index)
        {
            factorialList.RemoveAt(index);
        }
    }
```

熟悉重构的人是否已经嗅到了代码的坏味道了呢？是的，在 Add()方法里，将计算阶乘的算法也放到了里面。由于这些代码实现了独立的算法，因此应该利用 Extract Method 规则，将这些代码提炼出来，形成独立的方法。

```
    public void Add()
    {
        long result = CountFactorial();
        factorialList.Add(result);
    }
    private long CountFactorial()
    {
        long result = 1;
        int seed = factorialList.Count + 1;
        for (int i=1;i<=seed;i++)
        {
            result*=i;
        }
        return result;
    }
```

我们还进一步简化 Add()方法：

```
    public void Add()
    {
        factorialList.Add(CountFactorial ());
    }
```

现在我希望扩充这个容器的功能，加入菲波那契数列的计算。由于两者的计算方式是完全不同的，因此需要重新创建一个菲波那契容器：

```
    public class FibonacciContainer
    {
        public FibonacciContainer()
```

```
{
    fibonacciList = new ArrayList();
}
public FibonacciContainer(int capacity)
{
    m_Capacity = capacity;
    fibonacciList = new ArrayList();
}
private ArrayList fibonacciList;
private int m_Capacity;
public ArrayList FibonacciList
{
    get {return fibonacciList;}
    set {fibonacciList = value;}
}
public int Capacity
{
    get {return m_Capacity;}
    set {m_Capacity = value;}
}
public long this[int index]
{
    get {return (long)fibonacciList[index];}
    set {fibonacciList[index] = value;}
}
public void Add()
{
    fibonacciList.Add(CountFibonacci ());
}
public void RemoveAt(int index)
{
    fibonacciList.RemoveAt(index);
}
public void Clear()
{
    fibonacciList.Clear();
}
private long CountFibonacci ()
{
    long result = 0;
    int seed = fibonacciList.Count;
    if (seed == 0 || seed == 1)
    {
        result = 1;
```

```
            }
            else
            {
                result = this[seed-1] + this[seed-2];
            }
            return result;
        }
    }
```

比较上面两段容器的代码，会有很多相似之处。又是时候拿起重构的利器了。首先我们根据 Rename Method 规则，将计算阶乘和菲波那契数列的方法改名为统一的名字。为什么要改名呢？既然两个容器有着相似之处，为什么不能定义一个基类，然后从其派生出各自的类呢？为了保证类方法的一致性，当然有必要对方法重新命名了。实际上，我们不仅要重命名方法名，而且还要改变属性的名字。

FacotorialList 、FibonacciList：改名为 MathList；

CountFactorial()、CountFibonacci()：改名为 Count()；

然后再通过 Extract Class 和 Extract SubClass 规则抽象出基类 MathClass。

最后的代码为：

基类：MathContainer

```
public abstract class MathContainer
{
    public MathContainer()
    {
        mathList = new ArrayList();
    }
    public MathContainer(int capacity)
    {
        m_Capacity = capacity;
        mathList = new ArrayList(capacity);
    }
    private ArrayList mathList;
    private int m_Capacity;
    public ArrayList MathList
    {
        get {return mathList;}
        set {mathList = value;}
    }
    public int Capacity
    {
        get {return m_Capacity;}
        set {m_Capacity = value;}
    }
    public long this[int index]
    {
        get {return (long)mathList[index];}
```

```
            set {mathList[index] = value;}
        }
        public void Add()
        {
            mathList.Add(Count());
        }
        public void RemoveAt(int index)
        {
            mathList.RemoveAt(index);
        }
        public void Clear()
        {
            mathList.Clear();
        }
        protected abstract long Count();
}
```

然后从基类分别派生出计算阶乘和菲波那契数列的容器类：

派生类：FactorialContainer

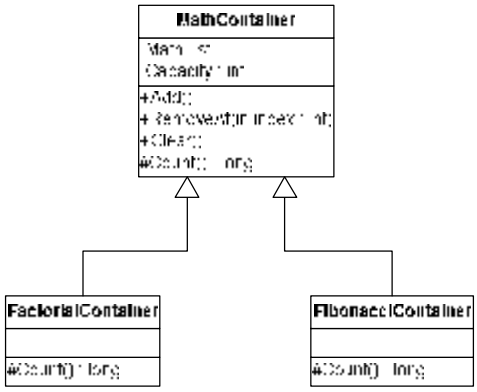
```
public class FactorialContainer:MathContainer
{
    public FactorialContainer():base(){}
    public FactorialContainer(int capacity):base(capacity){}
    protected override long Count()
    {
        long result = 1;
        int seed = MathList.Count + 1;
        for (int i=1;i<=seed;i++)
        {
            result*=i;
        }
        return result;
    }
}
```

派生类：FibonacciContainer

```
public class FibonacciContainer:MathContainer
{
    public FibonacciContainer():base(){}
    public FibonacciContainer(int capacity):base(capacity){}
    protected override long Count()
    {
        long result = 0;
        int seed = MathList.Count;
        if (seed == 0 || seed == 1)
```

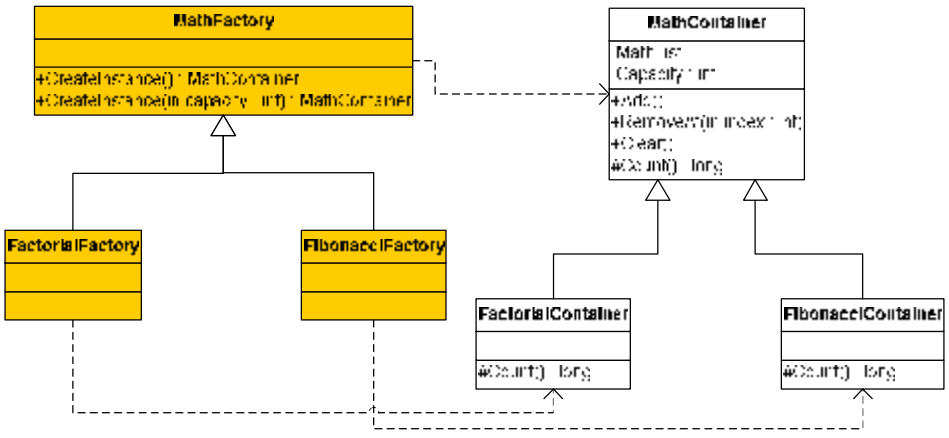
```
            {
                result = 1;
            }
            else
            {
                result = this[seed-1] + this[seed-2];
            }
            return result;
        }
    }
```

UML 类图如下：



对于这样的程序结构，要扩展起来是很容易的，例如素数的容器，我们只需要定义 PrimeNumberContainer 类，然后重写 Count()方法，并派生 MathContainer 类即可。

经过重构，程序的结构变得愈发清晰。如果我们再仔细分析现在的结构，对于算法的扩展是非常容易的，但如何创建每个容器的实例，似乎还有更好的方法，那就是通过工厂来管理每个实例的创建。因为产品只有一类，所以可以参用工厂方法模式（Factory Method）。首先我们来看看 UML 类图：



实现代码如下：

基类工厂：MathFacotry 类

```
public abstract class MathFactory
{
```

```
    public abstract MathContainer CreateInstance();
    public abstract MathContainer CreateInstance(int capacity);
}
```

阶乘容器工厂：FactorialFactory

```
public class FactorialFactory:MathFactory
{
    public override MathContainer CreateInstance()
    {
        return new FactorialContainer();
    }

    public override MathContainer CreateInstance(int capacity)
    {
        return new FactorialContainer(capacity);
    }
}
```

菲波那契数列容器工厂：

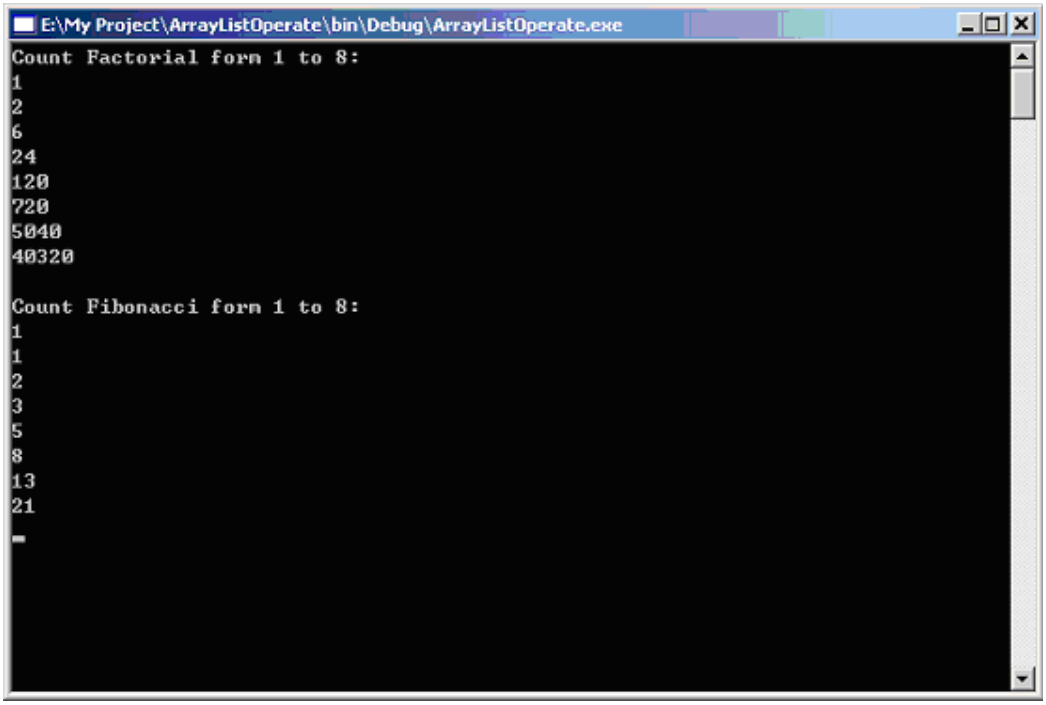
```
public class FibonacciFactory:MathFactory
{
    public override MathContainer CreateInstance()
    {
        return new FibonacciContainer();
    }
    public override MathContainer CreateInstance(int capacity)
    {
        return new FibonacciContainer(capacity);
    }
}
```

有了工厂，就可以通过工厂来创建我们所需要的具体容器类对象了：

```
[STAThread]
static void Main(string[] args)
{
    MathFactory factory1 = new FactorialFactory();
    MathFactory factory2 = new FibonacciFactory();

    MathContainer factorial = factory1.CreateInstance();
    MathContainer fibonacci = factory2.CreateInstance();

    Console.WriteLine("Count Factorial form 1 to 8:");
    for (int i=1;i<=8;i++)
    {
        factorial.Add();
    }
    for (int i=0;i<8;i++)
```

```
        {
            Console.WriteLine(factorial[i].ToString());
        }

        Console.WriteLine();
        Console.WriteLine("Count Fibonacci form 1 to 8:");

        for (int i=1;i<=8;i++)
        {
            fibonacci.Add();
        }
        for (int i=0;i<8;i++)
        {
            Console.WriteLine(fibonacci[i].ToString());
        }
        Console.ReadLine();
    }
```

最后的输出结果如下图所示：



本来是一个简单的例子，似乎到后来越来越复杂了。然后仔细分析程序结构，你会发现这个的扩充性和灵活性是很好的。通过重构，并运用设计模式的工厂模式，我们逐步创建了这样一个渐趋完美的数学运算容器。大家可以试试为这个容器添加其他算法。也许在这个过程中你会发现结构还存在一些不足，那么不用担心，运用重构的武器吧。虽然这个武器可能比不上 CS 高手手里的 AK47，不过对于对付大多数问题，也足以所向披靡了。

从企业的运行价值链说起

——我眼中的测试驱动开发

看了一期中央电视台的《对话》栏目，节目对三星 CEO 尹钟龙进行了访谈。其中，现场一位女士的一句话令我印象深刻。她提到一个企业的运行价值链，共分为三步：首先是发现价值，找到目标市场；第二步是生产价值，将高质量的产品生产出来；最后是保护价值或收获价值，做好品牌。

怎么理解呢？这位女士以 Nike 作比喻。第一步是设计 Nike 鞋，这就是发现价值，可能获得 100 美元的价值；然后再拿到中国来生产，大约是 10 美元；最后再将生产好的鞋子，贴上 Nike 的商标送回到美国去卖，又可以收获 90 美元。一双鞋售价 200 美元，而生产价值所能收获的却只有 10 美元。这一步获取利益最低，我们中国的公司却做得最好。而怎么去发现价值，然后又怎样去巩固自己的品牌和知名度，中国的公司就做得不那么好了。

据我的了解，国内的软件开发应用 TDD（测试驱动开发）相对较少，很多人认为：测试驱动开发是个好东东，但似乎不符合中国国情。说到原因，最多的一条就是项目时间紧，没有时间写测试代码。在项目中，到底该不该使用 TDD，大多数人持怀疑或观望的态度。这种态度与观点，就让我想起了如上《对话》中的这一段话。

再仔细分析企业运行价值链的三步走，我觉得和软件开发的 TDD 价值链很相似。第一步，是发现价值。应用到 TDD 中，就是测试先行，通过测试来驱动我们编写代码。第二步，生产价值。毋庸置疑，这正是编写代码的一个阶段。而第三步，就是收获价值，在 TDD 中，我们收获的不仅有开发后完整的产品，同时还收获了完整的测试套件。和 Nike 鞋的生产一样，我们在软件开发中，过度地重视了第二步生产价值阶段，而对于第一步和第三步，要么是忽略了，要么就是没有提高到相应的高度。

一、发现价值与生产价值

习惯了传统开发模式的程序员，非常不适应写代码之前，先写测试的方法，这其中也包括我。那么，我们一般是怎样去发现价值的呢？首先通过需求分析，然后进入设计阶段。在设计阶段期间，再围绕需求分析的结果，更多的是从实现的角度，而非从客户应用的角度出发。TDD 颠覆了这种模式。因为需要测试先行，就驱动了程序员必须从功能出发、从应用出发。在写测试代码的过程中，我们需要考虑要实现那些功能，相应的类的名称、对象的创建方式，以及可能会应用到的模式和策略，如此种种，在这个过程中，如剥笋子一般逐渐地规定出来了。

在这个过程中，我们要审慎地选择测试的步子。昂首阔步虽然显得气势轩昂，行进快速，但往往会忽略沿途的风景。在测试驱动开发中，我建议你小心的规划测试样例，从测试样例的逐步完善中，渐进地驱动出你更加完善的代码。例如，我需要开发一个智能的个人助理，它目前能提供的功能是：能够让用户定制自己感兴趣的类别，然后个人助理根据用户的定制进行搜索，并将搜索得到的结果按不同的类别进行存储。

我们来尝试一下 TDD 的过程。根据对功能的分析，我们首先应该有一个智能助理对象，测试代码如下：

```
[Test]
public void TestSmartAssistor()
{
```

```
        SmartAssistor assistor = new SmartAssistor();
        Assert.IsNotNull(assistor);
    }
```

当然，这段代码是连编译都无法通过的，我们还需要创建 SmartAssistor 类型。然而，不要小瞧了这一步，它实际上促使你对项目进行初步的理解，至少，你需要想好这个将要创建的类型，它的名字是什么？这就是一种驱动力。（为了简便起见，在本文只列出测试代码。）

然后，这个类型能够做些什么呢？我们把个人智能助理的功能进行分类，应该包括三个功能：

1、 定制；

2、 搜索；

3、 存储。

仔细想想，实际上只有搜索和存储才是智能助理的职责所在，而定制不过是智能助理要运转的一个条件罢了。既然如此，从客户应用的顺序来考虑，我们应该先实现定制的功能。要定制类别，就应该具备类别类型，而定制类别这项功能，则应该由一个专门的控制器来承担责任。

```
    [SetupUp]
    public void InitObject()
    {
        Category cg1;
        Category cg2;
        CategoryContainer cgContainer;
        SmartController control;
    }
    [Test]
    public void TestCategory()
    {
        cg1 = new Category("SoftWare Engineering","TDD");
        cg2 = new Category("SoftWare Engineering","Design Pattern");
        cgContainer = new CategoryContainer();
        cgContainer.Add(cg1);
        cgContainer.Add(cg2);

        Assert.IsNotNull(cgContainer);
        Assert.AreEqual(cg1,cgContainer[0]);
        Assert.AreEqual(cg2,cgContainer[1]);
    }
    [Test]
    public void TestController()
    {
        control = new SmartController();
        Assert.IsNotNull(control);
        Assert.IsTrue(control.CustomizeCategories(cgContainer));
    }
```

上面的测试代码完全从用户的应用角度来考虑的。要定制类别，必须具备类别类型

Category，它应该实现一个带参的构造函数，传递主类别和子类别。由于定制的类别可能会很多，所以需要一个类别容器 CategoryContainer。

而定制类别，则由控制器 SmartController 完成。定制的方法 CustomizeCategories()定制多个类别，并返回布尔型，以确定定制是否成功。通过上述的测试代码，写出相应的程序应该不难吧。

接下来应该考虑搜索和存储功能了。在前面我提到，这两个功能应该是 SmartAssistor 类型的职责。先修改最初的测试代码：

```
[Test]
public void TestSmartAssistor()
{
    SmartAssistor assistor = new SmartAssistor();
    Assert.IsNotNull(assistor);
    assistor.Search(control.Categaries);
    assistor.Store();
}
```

此时，我发现在写 Search()和 Store()方法的断言时，有些问题存在。这两个方法返回的结果应该是什么？是布尔值吗？那么搜索得到的结果呢？存储后形成的文件呢？对于用户而言，是否只需要这两个行为呢？

仔细分析，我认为，Search()和 Store()方法返回布尔值是可行的，但 Search()还应该返回搜索结果。Store()方法也应该指定存储的路径和文件的格式。而用户不仅仅需要搜索和存储，同时还应该提供显示的功能。所以，上面的测试代码需要完善：

```
[Test]
public void TestSmartAssistor()
{
    SmartAssistor assistor = new SmartAssistor();
    Assert.IsNotNull(assistor);

    SearchResult result = new SearchResult();
    Assert.IsTrue(assistor.Search(control.Categaries,out result));
    Assert.IsTrue(assistor.Store(result,@"D:\Smart Assistor\","result.xml"));

    assistor.List(result);
}
```

此时根据测试代码写出的程序，应该说就具备个人智能助理的雏形了（所谓智能，还应具备自动搜索，自动匹配，自动分类等诸多功能，本文只是根据该项目提出 TDD 的一些观点，因此这些功能省略）。

二、通过 TDD 进行重构

“发现价值”的过程远远没有结束。通过测试代码，我们从客户的角度来考虑，会发现一些问题。在已经实现的代码中，SmartAssistor 类型实现了 Search，Store 和 List 的功能。但这些职责是否真的应该由它承担呢？表面上来看，是这样的。然而根据 OO 的思想来看，这个 SmartAssistor 所承担的责任是否太多了？它和搜索的结果、显示的方式耦合度是否太紧密了？这个设计将实现抽象出来了吗？这些都应该是我们考虑的重点。考虑的时机，可以是设计之初，也可以是重构之时。

在重构的时候，仍然不能放弃 TDD，只有它才能保证程序的可靠性，重构的正确性。

开始重构吧。

首先从行为来考虑。搜索的功能会很复杂吗？可能会有精确搜索，模糊搜索；可能是在网上搜索，也可能是本机搜索。那么，存储的功能呢？IO 的操作是否频繁，存储的要求是否会根据安全级别而逐步升级？再考虑显示，对于个人智能助理来讲，显示的方式需要多样化吗？显然，以上的行为都是复杂的。

再从抽象性考虑。需要把这些行为抽象出来吗？也就是说，这些行为的载体是否会有多种类型？显然，搜索可能会是文件的搜索，可能会是文本的搜索，也可能会是数据库的搜索；存储的格式也会有多种多样，文本文件，xml 文件，数据库文件。显示的方式可能会通过浏览器显示，也可能会在 WinForm 中显示。也许用户要求是带滚动条的文本框，也许只是简单的文本显示。对象的形式很多吧，需要抽象吗？显然是的！

这样考虑之后，我发觉需要重构的东西太多了，应该怎么入手？首先，我们把 SmartAssistor 的职责先剥离出来，用更单一的对象来完成各自的功能。然后，分别将这些对象提炼出各自的接口。还是先写测试代码吧，考虑搜索功能，首先需要将对象分离出来：

```
[Test]
public void TestSearching()
{
    SearchEngine engine = new SearchEngine();
    Assert.IsNotNull(engine);

    SearchResult result1 = new SearchResult();
    SearchResult result2 = new SearchResult();
    Assert.IsNotNull(result1);
    Assert.IsNotNull(result2);
    result1 = engine.ExactSearch(control.Categaries);
    result2 = engine.BlurSearch(control.Categaries);

    SearchResult tempResult1 = new SearchResult(control.Categaries,"contents");
    SearchResult tempResult2 = new SearchResult(control.Categaries,"more contents");

    Assert.AreEqual(tempResult1,result1);
    Assert.AreEqual(tempResult2,result2);
}
```

在 NUnit 中运行测试代码，未能通过。然后在程序中创建 SearchEngine 类型，并实现 ExactSearch 和 BlurSearch 方法。直到在 NUnit 中运行通过，全部显示绿灯。

接下来抽象出 SearchEngine 的接口 ISearchEngine，并让 SearchEngine 实现该接口。其中接口方法包括 ExactSearch 和 BlurSearch 方法。将前面的测试代码作小小的修改，修改后同样需要在 NUnit 中运行，保证顺利通过：

```
[Test]
public void TestSearching()
{
    ISearchEngine engine = new SearchEngine();
    Assert.IsNotNull(engine);
    //……
```

```
}
```

考察 SearchResult 类型，该类型的对象应该在整个程序中只保留一个对象，因此，应对此采用单例模式。修改测试代码：

```
[Test]
public void TestSearching()
{
    ISearchEngine engine = SearchEngine.Instance;
    Assert.IsNotNull(engine);
    //……
}
```

根据测试代码修改程序代码，将 SearchResult 类型的构造函数改为 private，并提供只读的静态属性 Instante，以此来获得单例对象。

仅仅是这样还不够的。考虑到搜索的范围有多种情况，如 internet，local machine，DB 等。SearchEngine 类型应该具体化不同类型，并同时实现 ISearchEngine 接口。例如搜索范围在 internet，测试代码如下：

```
[Test]
public void TestInternetSearching()
{
    ISearchEngine engine = InternetSearchEngine.Instante;
    Assert.IsNotNull(engine);
    //……
}
```

既然有如此多的类型，类型的创建就必须通过工厂进行管理。此时测试代码需要做进一步的修改：

```
[Test]
public void TestInternetSearching()
{
    ISearchEngineFacotry fatory = new InternetSearchEngineFactory();
    ISearchEngine engine = factory.CreateInstante("Internet");
    Assert.IsNotNull(engine);
    //……
}
```

同理，我们应该分别实现测试方法 TestLocalSearching()和 TestDBSearching()。

按照这样的思路，分别对存储功能和显示功能进行重构。记住，每做一步重构，都需要严格按照 TDD 的方式。首先写出测试代码，然后在 NUnit 运行。如果是红灯，需要写出相应的代码，再运行 NUnit，直到全部均为绿灯为止。

三、第一步的小结

表面上看，这样繁复地写测试代码，程序代码，确实是有些 Kill Time 了。但我们需要认真地思考所谓“发现价值”的意义。通过测试先行的方式，以模拟客户应用的状态来考量客户的需求，并通过此驱动程序员一步一步地到达“生产价值”的终点。“发现”与“生产”并行不悖，同时“质检员”一直跟随其间，保证了产品的质量。

就好比 Nike 鞋的生产，必须以体贴用户的角度出发，设计出吸引人的样式，那么大规模的生产才会有盈利的可能。

TDD 的生产过程也许慢了一点，但请不要忽略了它其实已经省去了编码后单元测试的

时间。相加相减之后，又会浪费多少时间呢？所以，千万不要以“时间紧”的理由来搪塞我哦。

四、考察第三步——收获价值

传统的方式，在产品生产出来之后，紧接着的是大量的测试，其中也包括单元测试；最后收获了产品、一大堆源代码和文档。而 TDD 的方式，既省去了单元测试的过程，同时还收获了另外一样上帝赐予的礼物——测试类或测试套件。

测试类绝对是一件奇妙的礼物。必须认识到它的价值不只是在于“发现价值”的阶段，它同样是我们的“收获”。

第一：比代码更好的文档、比文档更好的代码

有了它，不用钻进浩如烟海的文档里，四顾茫然了。文档的文字描述既不准确，容易产生歧义，又容易产生文档同步的问题。也许它能促进你对业务和架构的理解，但对于程序本身，你无法从文档中得到基本的启示。

那么看程序的源代码吗？你会在众多的类对象和方法中绕来绕去，最后一头雾水，精疲力尽之后，还是一无所获。

而看测试代码就不同了，你不需要了解每个方法的具体实现，因为测试代码是从客户的应用角度来书写的，看完测试代码，你会很轻松地理清程序结构的脉络。

第二：新兵训练营的绝佳教材

也许你的项目组新进了员工，如果他熟悉 TDD，那么，这些测试类是他熟悉项目的最好文档；如果他还没听说过 TDD，不用着急，先把这些测试类给他。只要他不是程序设计的新手，我想这个新兵会很快熟悉项目组开发的方式。再让他写几个测试样例，他会立即投入到 TDD 的怀抱中来的。

第三：满载而归的信心

项目开发中，成员最宝贵的除了认真、努力、团队精神之外，就是信心了。这里所谓的信心，并非是对自己能力充满乐观的估计和客观地评价后，表现出来的精神面貌，而指的是程序员对代码正确性的信心。无论这些代码是自己写的，还是他人写的，只要严格按照 TDD 的要求进行，你都会对它们充满信心。虽然不能保证没有 bug，但必须承认的是通过单元测试，我们已经将 bug 降低到最小了。

五、结论

中国企业在企业运行价值链上，走好了利润最低的第二步，却忽略了“发现价值”和“收获价值”对于一个企业的重要性。韩国三星在几年之前还是一个亏损 600 多亿美元的企业，如今它已经成功地扭亏为盈，并跻身世界五百强。原因很多，但不可忽视的是，他在价值链的首尾两步中作得很好。从高端产品中发现价值，找到了目标市场；从品牌创造中收获了价值，走向了世界。

我不是说软件开发一定要采用 TDD 的方式，它自然也有很多缺陷。然而，我们在开发的过程中，同样要重视设计的“发现价值”阶段，然后在收获产品的同时，不要忽略了还应该收获其他同样值得珍视的“价值”。从这一点来看，也许 TDD 更符合这种价值链的模式。而我们程序员千万不要舍本逐末，过于偏执地重视“生产价值”，以致于在软件开发方法上，总是落后于人，进而受制于人!

最后，谨以我之愚见，思考 TDD 的方式，认为 TDD 内力精深，大约分为四种无上之力：

1、 驱动力——驱动程序代码编写；

2、 学习力——新兵训练营之绝佳教材；

3、 自信力与他信力——bug 降到最低；

4、 控制力——与重构紧密接合，牢牢控制开发过程。

使用极限编程改善项目的设计和灵活性

作者：Will Stott & James NewKirk
译者：张逸

本文讨论：
测试驱动开发（TDD）的全面性描述；
TDD 和极限编程的好处；
NUnit 介绍；
与传统开发技术的比较；

随着项目的逐渐成熟，难道你不愿意自己编写代码的道路通向坦途，而非走向荆棘丛生的崎岖小径？似乎看起来不管你采取什么方式，编码之路迟早会走向迷途。项目越大，困难越多。有多少次当你以一个近乎完美的设计开始你的项目，然而一旦开始编码，你会发现自己事实上只看到了项目的局部？

测试驱动开发（TDD）改变了编码的过程，并且这种改变不仅是可能的，同时也是值得去做的。开发包括三方面的活动：编写测试用例，编码并进行测试，重构代码以消除重复代码使其更简单、更灵活、更容易理解。

这个过程会频繁地重复，每次进行测试均是为了保证产品的正确性。设计、编码和测试三者之间的鸿沟将不再存在，这样可以促进你对整个环境更好地理解。因此，你的设计（和编码）将随着项目的成熟逐步得到改善而非降低。

使 TDD 更加有效的原因是单元测试自动化，而且这些自动化单元测试的工具可以从 Internet 上免费获得。虽然没有简化功能版的商业产品，但开发人员可以合理地使用这些高质量的软件。本文将指导你怎样获得和使用 Nunit 并通过 C#（或者任意一种基于 Microsoft .Net Framework 的语言）开发实践 TDD。注意到类似的工具对于 C++和 Java 开发人员也是可用的，因为他们支持大多数语言和操作系统。这些工具与极限编程紧密地结合起来，扩大了 TDD 应用的范围。

为什么设计会降低性能

大多数传统软件开发过程是基于你在设计之初正确的设计与估计，并通过开发以形成完美的产品。这种开发方式保证了产品的统一性和之间的最小差异。然而这一过程却忽略了交流与反馈，同时也不利于生成错误信息（测试失败），并因此制定策略采取相应的措施（修复设计），而这些正是 TDD 所重视的。

为什么我们不能在开始就获得正确的设计？因为在开发时，处于项目之初的我们无法获得有关这个软件完整的知识。迭代式开发虽然也认同这一事实，并帮助你在项目开发初期识别一些重要的问题，而不是把这些问题留在后期解决。然而，迭代法无法终止开发过程，使你回到设计阶段来解决问题，哪怕这只是因为一个命名糟糕的公共类名。没有人愿意关注这些细小的问题，且不幸地，这种设计过程也禁止这种更新，因为在各个阶段不停重复的代价太高。

传统开发过程中的这些小问题一旦积累起来，会导致大问题的发生。你或许认为与其将时间耗费在这些对功能影响不大的细枝末节上，还不如将精力放在更重要的环节。然而，这个命名糟糕的公共类在代码中保留的时间越久，则相关的使用也会越来越多，改变起来就越来越困难。之后，团队会在编码的时候非正式地修改这些问题，很快，在计划发布整个产品时，你会做大量的工作尽量使代码和设计文档保持一致。对于这种情况来说，你之前的设计是没有价值的，因为代码本身就说明了设计。

测试驱动开发允许你推迟决定，直到你更好地理解了问题之所在。当你只是了解到产品开发的一些基本信息时，你不必设计出完美的体系架构。这对于传统软件开发中已经确定的理念来说，是一种挑战，从某种角度来说，甚至是违反常规的。因此我们建议你以一种开放的思想来尝试 TDD，你会发现 TDD 的强大。

TDD 的另一个好处是更容易掌握。你开发的一系列测试说明了代码的运行机制，这种程序自我编档的方式促进了交流。它要求你通过概览测试用例并阅读自己的代码来获得反馈，从而有利于你创建的对象和组件更趋于松散耦合。同时，TDD 可以推迟你的设计决策，从而简化设计，使你能集中精力修正设计的问题。最后，它通过给出的一系列测试减轻你编码的压力，因为当测试发生中断时，它可以立即告诉你究竟发生了什么。

简介

要真正理解 TDD，唯一的办法就是实践。让我们从一个简单的例子开始，这个例子并不要求任何特殊的工具。我们要写一个小程序，来帮助我们规划住宅前的一块长方形的草坪，但在设计之初，我们需要作两个测试：当长度为 3，宽度为 2 时，计算得到的面积应该为 6；同时计算其周长应该为 10。

写下这些测试有助于我们将焦点集中到程序的一些要点。在这个例子中，似乎创建一个对象来模拟问题域是合理的，因此我们创建了一个类，取名为 Quad，来代表我们要建造的长方形草坪，并在一个简单的控制台应用程序中实例化它。下面就是我们实现的步骤：

1、启动桌面的 Visual Studio，选择文件 | 新建 | 项目，选择 C#项目类型中的“控制台应用程序”。为项目取名为“QuadApp”，然后点击 OK。

2、在主函数中输入下列几行代码，创建 Quad 实例，并通过 Assert 断言，当长宽各为 3 和 2 时，返回的结果应该是 6。

```
static void Main(string[] args)
{
    Quad q = new Quad();
    System.Diagnostics.Debug.Assert(q.Area(3,2) == 6);
}
```

3、选择项目 | 添加类，在对话框中输入“Quad.cs”，点击“打开”以创建类。

4、输入下列代码，创建在 Main 函数中用到的 Quad 类的成员方法 Area：

```
public class Quad
{
    public int Area(int length, int width)
    {
        return 0;
    }
}
```

5、选择生成 | 生成 QuadApp 菜单项，此时编译应该是成功的，然后当你运行该程序

时，会报告错误，因为此时返回的面积值不等于 6。

6、指定 Area 方法的返回值为 6，再重新编译程序，并运行，此时没有错误发生。

7、改变 Area 方法的实现，返回值用输入参数的乘积来代替起初指定的返回值常量。再次重新编译程序，并运行测试。

第二个测试（计算周长为 10），与前面的方法相似。首先在主函数中通过 assert 写测试代码，接着执行第 4 到第 7 步，只是方法名用 Perimeter 而不是 Area。当你在第七步中试图改变设计时，你可能会得出一个结果，就是可以通过构造函数来传递长和高的值，并将其存储为对象的属性，而不是通过成员方法的参数来传递。设计改变后的代码如下：

```
public class Quad
{
    private int m_length;
    private int m_width;
    public Quad( int length, int width)
    {
        m_length = length;
        m_width = width;
    }
    public int Area()
    {
        return m_length * m_width;
    }
}
```

重新编译程序，并运行之，以保证你在写代码是否因粗心而漏掉了什么。

我们通过一种最简单的形式完成了演示 TDD 工作原理的练习。下面是每一步的总结：

1、写一个失败的测试：我们选择了一个最容易实现的测试（虽然在这个例子中所有的都很简单）来实现。接下来，我们写了个 Quad 类的最简单实现，以保证程序能够通过编译。然而，当我们执行程序时，调试窗口会出现 assert 的错误信息，因为返回的面积值不是 6。

2、修改代码以通过测试：要修改代码，首先我们做一个最简单的方法，就是直接返回常量值 6。该值刚好使程序能够运行且通过测试。接着下一个测试必须使用不同的参数值，迫使我们实现正确的算法，以通过所有的测试，即使我们还没有开始下一个环节的重构。

3、重构代码：既然在我们的程序中达到了正确的预期目的，我们应该试图去掉一些重复代码使代码更易维护，更简单，更灵活，也更易于理解。我们可以发现这种改变在运行测试时，并没有影响到程序的行为。

修改代码使其易于维护，要求不能修改程序可观察的行为，这个思想并不是全新的理念。然而这里重要的是通过重构，而不是一遍一遍地整理代码。重构必须是软件开发活动中一个关键的环节，通过合适的工具，并采取系统的方法，可以逐步地改善代码的质量。通过执行贯穿于产品整个生命周期的许多次小的重构，最后达致最完美的目标，从而使得软件易于维护。

倘若必要的反馈要求你的设计（和代码）在产品开发中得到改进，则重构是测试驱动开发中不可缺少的步骤。当然你也可以不执行 TDD 方式，而进行重构以去除重复代码，使其更简单、灵活、更易于理解。无论你采取什么开发方式，你都需要将重构作为一种常规的活动，以保证你的更改是有效的。

重构的力量在于其能够减少因为改变正在工作的代码所带来的危险。好的工具能够帮助你减少潜在的危险，而同等重要的是要采取一系列严谨的步骤和正确的结构与规则。Martin Fowler 的大作《重构：改善既有代码的设计》对于重构有非常精彩的描述。他定义了一系列非常有用的重构模式，并提供许多相关的样例指导你怎样进行重构。

目前，Visual Studio.Net 除了提供查找和替换功能外，对重构没有太多的支持。然而，相信在不远的将来 MS 会提供更多有力的工具，使你能完成更多复杂的工作，例如符号的重命名，在编译-生成级上更改类名，而不是在你的源文件中作替换操作。你甚至可以期待这样一天的到来——当你选中一段代码后，通过 Visual Studio 提供的菜单，直接应用重构模式，然后继续下一步工作。你有充足的信心保证你的程序是更加易于维护的，而不会引入 bugs 或者会改变其可观察的行为。

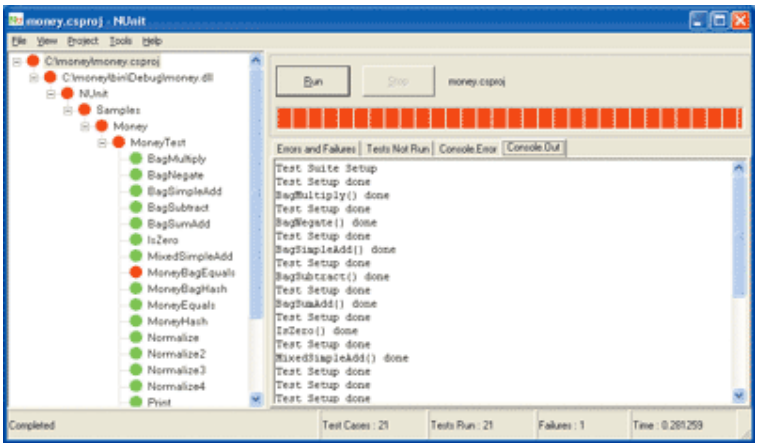
经验总结

迄今为止，你应该明白一条最重要的道理就是 TDD 很简单。事实上，TDD 能帮助你更好地理解任意一门新的语言、新的技术和组件。通过 TDD，你可以决定将要采取的步伐的跨度。有经验的专家可以采取大的步伐，以避免中间环节。一旦发现开发有误，可以令过程回滚，采取更小的步伐重复前面的过程。关于 TDD 我们需注意下列事项：

1、 测试使代码文档化（The tests document the code）。从测试的类 Quad 中可以清晰地看到这点；
2、 随着测试的进行，我们可以把握进度。通过测试，可以验证每个功能是否正确，如此每个人都可以运行测试来了解过程是否如我们说期望的那样被良好的建立。
3、 测试让人对代码的修改充满信心。即使你是一个 C#的新手，也可以象专家那样老练的编程，运行所有的测试确保你所做的没有违反既定的规则。
4、 类似于“将长度和宽度以参数的形式传递给面积和周长，而非在构造函数里指定”这样的错误，可以通过重构来修正。
5、 当你将 TDD 应用到现实世界时，你可以生成许多测试，并通过如 Nunit 的工具来组织它们。

使用 Nunit 使测试自动化

TDD 可能会生成成千上万个测试。假使项目中的每一个人都在周而复始地重复“写测试，修改代码，重构”，当然有必要使你的测试工作更加有效。如 Nunit 这样的测试框架就是设计用来帮助你提高效率的利器。它允许你象在 Visual Studio 中建立项目的方式，将你的测试用例装载到 NUnit 中，以合理地安排各自的项目。你可以将所有测试用例以层次结构显示出来，以单用例或测试包（一组测试用例）的方式运行测试，观察执行后的结果是成功（显示绿色进度条）还是失败（显示红色进度条），同时还将显示每次失败的详细信息。



图一：NUnit 中的测试状态

另外，NUnit 还提供了一些特殊的方法，使你能够在测试包开始和完成时执行初始化（initialize）和清除（clean up）被不同的测试用例所共享的“测试套件（test fixtures）（例如：长文件，数据库连接，公共链接）”。你也可以定义 Setup 和 TearDown 方法使其分别在执行测试前和完成后运行。它可以帮助你通过重新更改每个用例间的系统设置，使特定的用例从其它用例中独立出来。

Nunit 允许你用与应用程序相同的语言和环境来写测试用例。不需要学习特定的测试语言，使开发测试迅速而直接。同时还有一个完整的 Assert 类提供特定用例失败后的详细信息。

最后，NUnit 还可以作为控制台应用程序运行，只需要通过命令方式就可以输出结果。支持自动化创建合法的进程，举例来说，这样你就可以重建应用程序，并运行完整的测试包。同时，NUnit 控制台应用程序还可以生成 Xml 格式的测试结果日志。

有着软件测试背景的人们应该能认识到这些特性的价值，但即使你不具备这些经验，也能清楚地明白它能为作为开发人员的你做出多大的贡献。不过，不管你是专家还是新手，要明白 NUnit 是怎样帮助你组织测试用例的，最好的办法还是实践。你可以到 http://www.nunit.org 去下载 NUnit。关于使用该软件的详细细节可以从其许可文件中获得。

本文使用的 NUnit 是 2.1 beta 版，你也可以去下载最新版本的 MSI 文件（大约 1.5MB）。有了这个文件后，你可以双击它进行 Windows 下的安装。它可以在.Net Framework1.0 和 1.1 版本下运行，当然你也可以参考 NUnit 文档获得有关系统需求的更多信息。

安装好后，选择“Test”的子菜单项运行你的测试包，以确保产品是否安装正确。单击了菜单项后，等待你桌面上的 NUnit 打开后，然后单击“Run”按钮。

不到一分钟的时间，treeview 下的所有节点都将显示为绿色，这表明测试正常执行并通过。测试运行的数目会在状态条中显示出来，同时会显示执行测试所花的时间以及失败用例的数目。有时候你可能会得到一些显示为红色的节点，这表明测试失败，你可能需要重新安装 NUnit，或者从 NUnit.org 站点获得建议。

在 NUnit 中使用测试用例

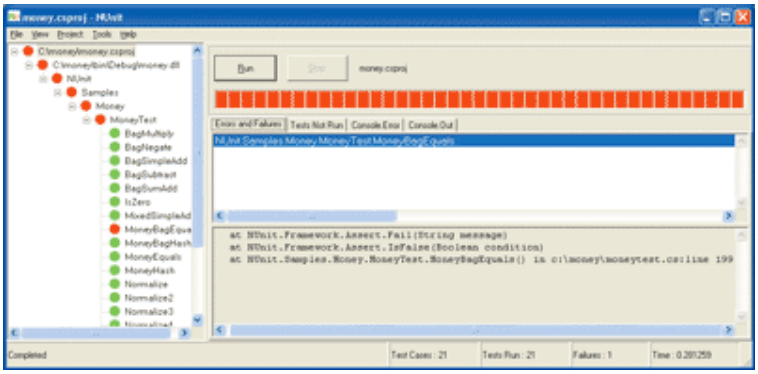
NUnit 有一些附带样例，你可以使用他们其中的样例，如 Money 项目，来学习使用 NUnit 的开发流程。

在你的桌面上启动 Visual Studio，选择文件 | 打开 | 项目 菜单，在 NUnit 的安装目录（Program Files\NUnit）的子目录 src\samples\money 下找到 Money.csproj 文件，单击“打开”按钮并等待 Visual Studio 将其加载。在解决方案资源管理器窗口中，查看引用文件夹中 nunit.framework 引用是否正确。选择 Build | Build Money 编译 Money 项目。编译应该成功通过，没有错误和警告。

在桌面上启动 NUnit。Treeview 中应该是空的——如果不是，请选择 File | Close 菜单关闭所有已存在的项目。然后在 File 菜单中选择 Open 菜单项打开 Money.csproj 文件，就像你在 Visual Studio 中打开该文件一样。现在 Treeview 中应该包含测试样例的集合。

选择 Treeview（文件 Money.csproj）中的根节点，单击 NUnit 右侧的“Run”按钮。测试用例的所有包均被执行，并恰好获得一个失败的测试——MoneyBag Equals。

选择 MoneyBag Equals 测试用例(红色的叶结点)，并定位到该文件的出错行，此时 NUnit 的右侧下方会显示相关信息，如图二：



图二：NUnit 错误跟踪

切换到 Visual Studio，打开该文件并定位到出错行。将该行注释掉（在行开始处插入//）并重新编译（生成 | 重新生成 Money），此时编译将完全通过，没有错误和警告。

现在再切换到 NUnit，注意到 Treeview 的所有节点均为灰色，这表明你更新后的程序还没有运行。选择 Treeview 的根节点，单击“Run”按钮。你可以看到所有的测试均通过且没有错误。

以上就是 NUnit 的全部工作方法。现在可以为你自己的项目创建测试用例了。

测试用例

测试用例定义了一系列输入，同时也可以定义一系列输出，以验证被执行的某些程序特性没有引入任何 Bug。测试可能会失败是因为产品里有 Bug，但也可能是测试代码自身有 Bug——可以称为 false negative。相似的，测试能通过也有可能恰好是因为测试中有 Bug，而非你的产品准确无误，我们称为 false positive。[译注：意思是测试中有 Bug，而产品中也有 Bug,巧的是这两处的 Bug 恰好负负得正，抵消了 Bug 所带来的影响，反而使测试通过。] 好的测试应该避免出现 false negatives，且绝对不能包含 false positives。另外，好的测试包不仅要测试正确的场景，也要测试错误的用例。

测试对象和其环境在测试用例被执行前总是保持相同的状态，因此一旦有必要就需重复执行并给出恒定的结果。当测试失败时，定位到错误的地方就能获得相关的信息。通过这些信息就能推断出问题能否被纠正。这就是为什么要求你对每个测试用例都要验证其合法性的原因所在。

现在我们为 Quad 例子写一些测试用例来实践上述的原理。在 Visual Studio 和 NUnit 中打开 Quad 项目，和起初打开 Money 项目的方法一样——选择文件 | 打开菜单，并定位到 QuadApp.csproj。选择项目 | 添加引用，找到文件 nunit.framework.dll（在 NUnit 安装目录的 bin 子目录下），单击确定，添加引用 nunit.framework。

在 Visual Studio 中创建 QuadTest 类：选择项目 | 添加类，输入名字 QuadTest.cs，然后单击“打开”。输入下列代码到 QuadTest 中：

```
using System;
using NUnit.Framework;

namespace QuadApp
{
    [TestFixture]
    public class QuadTest
    {
        [TestFixtureSetUp]
        public void DoTestSuiteSetUp()
        {
            Console.WriteLine("TestSuiteSetUp");
        }
        [TestFixtureTearDown]
        public void DoTestSuiteTearDown()
        {
            Console.WriteLine("TestSuiteTearDown");
        }
        [TearDown]
        public void DoTestCaseTearDown()
        {
            Console.WriteLine("TestCaseTearDown");
        }
        [SetUp]
        public void DoTestCaseSetup()
        {
            Console.WriteLine("TestCaseSetup");
        }
        [Test]
        public void Area()
        {
            Quad q = new Quad();
            Assert.IsTrue( q.Area(2, 3) == 6);
            Console.WriteLine("Area");
        }
        [Test]
        public void Perimeter()
        {
```

```
            Quad q = new Quad();
            Assert.IsTrue( q.Perimeter(2, 3) == 10);
            Console.WriteLine("Perimeter");
        }
    }
}
```

编译后，在 NUnit 中运行测试。单击 NUnit 右侧的 Console.Out 标签，可以查看执行的顺序。

现在你可以看到，不同的 Setup 和 TearDown 方法是怎样允许你控制测试环境，从而保证你的测试用例能根据需要而重复执行并给出恒定的结果。这些方法将帮助你确保运行一个测试用例而不会影响到另外的用例（测试用例独立性 test case isolation）。测试独立性的最佳证明就是具备以任意顺序运行测试用例的能力。试着改变一下测试用例的顺序，以确认他们是否具有独立性。

测试套件和测试包

一个测试套件是一个或多个测试用例共享的一个对象，这些测试用例都与测试对象的初始化或提供相应资源有关。在 NUnit 的术语中，测试套件是一个具备特性（attribute）[TestFixture]的类，并提供下列方法：

1、[Test]方法形成不同的测试用例。这些测试样例应该是测试的核心操作，并且与其它测试无关。

2、[SetUp]和[TearDown]方法提供测试用例间的环境设置。它们分别在测试开始前和结束后执行。

3、[TestFixtureSetUp]和[TestFixtureTearDown]方法要求对象被测试用例共享。它们分别在测试套件开始和结束时执行。

你希望运行测试的任意对象都可以在[TestFixture]类（被所有测试用例共享）里创建实例变量，也可以在方法（对于单个测试用例中是私有的）中创建局部变量，这就是为什么要描述为测试套件（test fixture）的原因。然而，你也可以将 NUnit 的[TestFixture]看作是组织测试包的一种方法，它将测试包组织为每个类，形成 treeview 的每一个分支。

既然你已经能够为自己的项目创建测试包，并且通过 NUnit 来运行它们，那么就让我们开始 TDD 之旅吧。当然，首先还要看看我们在开发工作中可能会遇到的问题。

在现实世界中使用 TDD

在使用 TDD 时，首先要考虑的是：既然商业产品是设计用来组织你的程序的，那么产品代码就应该很容易从代码中分离出来，以用来测试。在产品开发期间，你可以运行你的测试程序，而为了发布产品，你仍然能够轻易地将其从代码中移除。

你可能会遇到的另一个难题就是测试 GUI 应用程序的困难，因为 GUI 应用程序是被鼠标和键盘输入而驱动的。举例来说，你应该怎样写测试程序来激活用户点击 dropdown list 控件的事件呢？如果用户选择的是国名列表，那么又该怎样验证选中的内容呢？

解决这些问题的方案就是：将代码根据编译、测试和部署分类为不同的组件。举例来说，我们不应创建一个作为主应用程序的 Quad 类，生成相同的可执行文件；而应该允许这个类能够被包含在一个单独的库文件（.dll）中。[译注：实质就是将 Quad 以类库的形式创建，而非应用程序。] 这样我们就可以开发单独的测试程序和域程序作为各自的可执行文件

（.exe），并共享同一个包含 Quad 类的公共库文件（.dll）。注意，如果你仅仅创建了库，NUnit 和你的测试包会被当作接口使用，而不是所需的单独的业务外观（separate harness）。

记住这个原则：“使主程序尽量简单，而将复杂的业务作为类放到库中”。它将帮助你解决测试 GUI 应用程序的问题。TDD 的其中一个原则就是你不能测试第三方代码，因此你没有必要测试 GUI 框架类，虽然当框架相对简单的时候，有时会用来测试接口。这意味着你可以捕捉类里的用户事件，因为你知道这个类是怎么工作的；然后再将其传递到类里进行处理。此外，你也可以将测试程序和域程序从单独的可执行文件中分离出来，而该执行文件则共享了你竭尽心力开发的公共库。

让我们看看怎样实际运用 TDD 来开发包含有 Combobox 控件的简单窗体应用程序。

开发一个测试和创建库

在 Visual Studio 里选择文件|新建|项目菜单新建一个控制台应用程序，取名为 CountryTest。然后如第一个例子那样，为项目添加 nunit.framework 引用。选择项目|添加类菜单，输入名字 CountryBoxTest.cs，点击“打开”按钮，创建类 CountryBoxTest。然后在 CountryBoxTest 类中输入下列代码：

```
[TestFixture]
public class CountryBoxTest
{
    [Test]
    public void CheckContent()
    {
        CountryLib.CountryList list = new CountryLib.CountryList();

        Assert.AreEqual("UK", list.GetCountry(1));
        Assert.AreEqual("US", list.GetCountry(2));
        Assert.AreEqual("CH", list.GetCountry(3));
        Assert.AreEqual(null, list.GetCountry(4));
    }
}
```

生成项目。编译失败，因为 CountryList 类不存在。

使用 Visual Strudio，选择文件|新建|项目（添加到业已存在的解决方案中），创建一个新的类库，命名为 CountryLib。选择项目|添加类输入名字 CountryList.cs，单击“打开”，创建新类 CountryList。然后将下面的代码输入到 CountryList 中，并编译。

```
public class CountryList
{
    public String GetCountry(int No)
    {
        return null;
    }
}
```

首先我们为项目添加引用 CountryLib.dll，然后重新编译控制台应用程序 CountryTest。此时项目运行正常，因为我们已经创建了 CountryList 类。

在 NUnit 中打开 CountryTest.csproj，并运行测试。显示失败，因为 CountryList.GetCountry 方法返回 null。切换到 Visual Studio，在类 CountryList 中修改 GetCountry 方法，使其返回测试所需要的值——仅仅添加一些字符串常量，如下所示：

```
public class CountryList
{
    string one = "UK";
    string two = "US";
    string three = "CH";

    public string GetCountry(int No)
    {
        if ( No == 1 )
            return one;
        if ( No == 2 )
            return two;
        if ( No == 3 )
            return three;
        else
            return null;
    }
}
```

这段代码主要是为了程序正常进行，而写的最简单的例子。重新编译 CountryList 类库，并切换到 NUnit，然后运行测试——运行成功。

对 CountryList 进行你认为必要的重构，每次重构后都进行测试，以检查是否出现错误。你可能认为在 ArrayList 中存储字符串可以使代码更简单，容易，更利于理解，同时避免重复，但是你也可以选择其它解决方案。

创建 GUI

使用 Visual Studio 选择文件|新建|项目（添加到现有的解决方案中）创建新的 Windows 应用程序，并为项目添加引用 CountryList.dll。接下来，在工具栏中拖动 combobox 控件到主窗体上。然后将下面的代码输入到 CountryApp 中，并编译应用程序。

```
private void InitializeComponent()
{
    //...
    CountryLib.CountryList list = new CountryLib.CountryList();

    String country;
    int index = 1;

    while ((country = list.GetCountry(index++)) != null)
        this.comboBox1.Items.Add(country);
    //...
}
```

现在你的项目已经有了基本的结构，它允许你采用 TDD 技术通过进一步开发 CountryList 类库来创建 GUI 应用程序。考虑一下 CountryList 怎样才能处理用户接口事件（鼠标单击，键盘），并通过这个方法使其能够被 CountryTest 测试，并被 CountryApp 使用。对于 CountryList，你可能会有比本文的示例更好的解决方案，不过本示例试图说明：测试驱动开发能够改善设计，即使设计已经成熟。之前你不必一定要拿出完美的设计方案，这对于开发人员来说，无疑是一个福音。

结论

大多数设计都是从上而下创建，并基于可观察的特性进行分类，促进对问题的理解和解决。换句话说，我们试图创建层次结构的对象，它模型化了问题域。相反，TDD 则是从下而上，通过对一些小问题，依次实施一系列简单的解决方案，最后演进为设计。

重构确保了设计集中于一个好的解决方案而不是逐渐脱离设计。如果你认为这是设计之根本，你就对了。TDD 将在这十年潜在地改变软件开发的方法，就像过去十年面向对象技术所做到的那样。

NUnit 提示：*当我们使用 Visual Studio 打开 NUnit 附带的预先建好的测试样例时，你可能会发现引用 nunit.framework.dll 显示未找到，那么请打开解决方案资源管理器窗口的引用文件夹。此时你需要删除已经存在的引用（选择文件，右击“移除”），然后再次添加（选择引用，右击“添加引用”，浏览并定位到该文件，单击“确定”）。nunit.framework.dll 文件可以在 NUnit 安装文件夹的 bin 子目录下找到。*

从实例谈 OOP、工厂模式和重构

——《让僵冷的翅膀飞起来》系列之一

有了翅膀才能飞，欠缺灵活的代码就象冻坏了翅膀的鸟儿。不能飞翔，就少了几许灵动的气韵。我们需要给代码带去温暖的阳光，让僵冷的翅膀重新飞起来。结合实例，通过应用 OOP、设计模式和重构，你会看到代码是怎样一步一步复活的。

为了更好的理解设计思想，实例尽可能简单化。但随着需求的增加，程序将越来越复杂。此时就有修改设计的必要，重构和设计模式就可以派上用场了。最后当设计渐趋完美后，你会发现，即使需求不断增加，你也可以神清气闲，不用为代码设计而烦恼了。

假定我们要设计一个媒体播放器。该媒体播放器目前只支持音频文件 mp3 和 wav。如果不谈设计，设计出来的播放器可能很简单：

```
public class MediaPlayer
{
    private void PlayMp3()
    {
        MessageBox.Show("Play the mp3 file.");
    }
    private void PlayWav()
    {
        MessageBox.Show("Play the wav file.");
    }
    public void Play(string audioType)
    {
        switch (audioType.ToLower())
        {
            case ("mp3"):
                PlayMp3();
                break;
            case ("wav"):
                PlayWav();
                break;
        }
    }
}
```

自然，你会发现这个设计非常的糟糕。因为它根本没有为未来的需求变更提供最起码的扩展。如果你的设计结果是这样，那么当你为应接不暇的需求变更而焦头烂额的时候，你可能更希望让这份设计到它应该去的地方，就是桌面的回收站。仔细分析这段代码，它其实是一种最古老的面向结构的设计。如果你要播放的不仅仅是 mp3 和 wav,你会不断地增加相应地播放方法，然后让 switch 子句越来越长，直至达到你视线看不到的地步。

好吧，我们先来体验对象的精神。根据 OOP 的思想，我们应该把 mp3 和 wav 看作是一

个独立的对象。那么是这样吗？

```
public class MP3
{
    public void Play()
    {
        MessageBox.Show("Play the mp3 file.");
    }
}
public class WAV
{
    public void Play()
    {
        MessageBox.Show("Play the wav file.");
    }
}
```

好样的，你已经知道怎么建立对象了。更可喜的是，你在不知不觉中应用了重构的方法，把原来那个垃圾设计中的方法名字改为了统一的 Play()方法。你在后面的设计中，会发现这样改名是多么的关键！但似乎你并没有击中要害，以现在的方式去更改 MediaPlayer 的代码，实质并没有多大的变化。

既然 mp3 和 wav 都属于音频文件，他们都具有音频文件的共性，为什么不为它们建立一个共同的父类呢？

```
public class AudioMedia
{
    public void Play()
    {
        MessageBox.Show("Play the AudioMedia file.");
    }
}
```

现在我们引入了继承的思想，OOP 也算是象模象样了。得意之余，还是认真分析现实世界吧。其实在现实生活中，我们播放的只会是某种具体类型的音频文件，因此这个 AudioMedia 类并没有实际使用的情况。对应在设计中，就是：这个类永远不会被实例化。所以，还得动一下手术，将其改为抽象类。好了，现在的代码有点 OOP 的感觉了：

```
public abstract class AudioMedia
{
    public abstract void Play();
}
public class MP3:AudioMedia
{
    public override void Play()
    {
        MessageBox.Show("Play the mp3 file.");
    }
}
public class WAV:AudioMedia
```

```
{
    public override void Play()
    {
        MessageBox.Show("Play the wav file.");
    }
}
public class MediaPlayer
{
    public void Play(AudioMedia media)
    {
        media.Play();
    }
}
```

看看现在的设计，即满足了类之间的层次关系，同时又保证了类的最小化原则，更利于扩展（到这里，你会发现 play 方法名改得多有必要）。即使你现在又增加了对 WMA 文件的播放，只需要设计 WMA 类，并继承 AudioMedia，重写 Play 方法就可以了，MediaPlayer 类对象的 Play 方法根本不用改变。

是不是到此就该画上圆满的句号呢？然后刁钻的客户是永远不会满足的，他们在抱怨这个媒体播放器了。因为他们不想在看足球比赛的时候，只听到主持人的解说，他们更渴望看到足球明星在球场奔跑的英姿。也就是说，他们希望你的媒体播放器能够支持视频文件。你又该痛苦了，因为在更改硬件设计的同时，原来的软件设计结构似乎出了问题。因为视频文件和音频文件有很多不同的地方，你可不能偷懒，让视频文件对象认音频文件作父亲啊。你需要为视频文件设计另外的类对象了，假设我们支持 RM 和 MPEG 格式的视频：

```
public abstract class VideoMedia
{
    public abstract void Play();
}
public class RM:VideoMedia
{
    public override void Play()
    {
        MessageBox.Show("Play the rm file.");
    }
}
public class MPEG:VideoMedia
{
    public override void Play()
    {
        MessageBox.Show("Play the mpeg file.");
    }
}
```

糟糕的是，你不能一劳永逸地享受原有的 MediaPlayer 类了。因为你要播放的 RM 文件并不是 AudioMedia 的子类。

不过不用着急，因为接口这个利器你还没有用上（虽然你也可以用抽象类，但在 C#里

只支持类的单继承）。虽然视频和音频格式不同，别忘了，他们都是媒体中的一种，很多时候，他们有许多相似的功能，比如播放。根据接口的定义，你完全可以将相同功能的一系列对象实现同一个接口：

```
public interface IMedia
{
    void Play();
}
public abstract class AudioMedia:IMedia
{
    public abstract void Play();
}
public abstract class VideoMedia:IMedia
{
    public abstract void Play();
}
```

再更改一下 MediaPlayer 的设计就 OK 了：

```
public class MediaPlayer
{
    public void Play(IMedia media)
    {
        media.Play();
    }
}
```

现在可以总结一下，从 MediaPlayer 类的演变，我们可以得出这样一个结论：在调用类对象的属性和方法时，尽量避免将具体类对象作为传递参数，而应传递其抽象对象，更好地是传递接口，将实际的调用和具体对象完全剥离开，这样可以提高代码的灵活性。

不过，事情并没有完。虽然一切看起来都很完美了，但我们忽略了这个事实，就是忘记了 MediaPlayer 的调用者。还记得文章最开始的 switch 语句吗？看起来我们已经非常漂亮地除掉了这个烦恼。事实上，我在这里玩了一个诡计，将 switch 语句延后了。虽然在 MediaPlayer 中，代码显得干净利落，其实烦恼只不过是转嫁到了 MediaPlayer 的调用者那里。例如，在主程序界面中：

```
public void BtnPlay_Click(object sender,EventArgs e)
{
    IMedia media;
    switch (cbbMediaType.SelectItem.ToString().ToLower())
    {
        case ("mp3"):
            media = new MP3();
            break;
        case ("wav"):
            media = new WAV();
            break;
            //其它类型略；
    }
```

```
        MediaPlayer player = new MediaPlayer();
        player.Play(media);
    }
```

用户通过选择 cbbMediaType 组合框的选项，决定播放哪一种文件，然后单击 Play 按钮执行。

现在该设计模式粉墨登场了，这种根据不同情况创建不同类型的方式，工厂模式是最拿手的。先看看我们的工厂需要生产哪些产品呢？虽然这里有两种不同类型的媒体 AudioMedia 和 VideoMedia（以后可能更多），但它们同时又都实现 IMedia 接口，所以我们可以将其视为一种产品，用工厂方法模式就可以了。首先是工厂接口：

```
    public interface IMediaFactory
    {
        IMedia CreateMedia();
    }
```

然后为每种媒体文件对象搭建一个工厂，并统一实现工厂接口：

```
    public class MP3MediaFactory:IMediaFactory
    {
        public IMedia CreateMedia()
        {
            return new MP3();
        }
    }
    public class RMMediaFactory:IMediaFactory
    {
        public IMedia CreateMedia()
        {
            return new RM();
        }
    }
    //其它工厂略;
```

写到这里，也许有人会问，为什么不直接给 AudioMedia 和 VideoMedia 类搭建工厂呢？很简单，因为在 AudioMedia 和 VideoMedia 中，分别还有不同的类型派生，如果为它们搭建工厂，则在 CreateMedia()方法中，仍然要使用 Switch 语句。而且既然这两个类都实现了 IMedia 接口，可以认为是一种类型，为什么还要那么麻烦去请动抽象工厂模式，来生成两类产品呢？

可能还会有人问，即使你使用这种方式，那么在判断具体创建哪个工厂的时候，不是也要用到 switch 语句吗？我承认这种看法是对的。不过使用工厂模式，其直接好处并非是要解决 switch 语句的难题，而是要延迟对象的生成，以保证的代码的灵活性。当然，我还有最后一招杀手锏没有使出来，到后面你会发现，switch 语句其实会完全消失。

还有一个问题，就是真的有必要实现 AudioMedia 和 VideoMedia 两个抽象类吗？让其子类直接实现接口不更简单？对于本文提到的需求，我想你是对的，但不排除 AudioMedia 和 VideoMedia 它们还会存在区别。例如音频文件只需要提供给声卡的接口，而视频文件还需要提供给显卡的接口。如果让 MP3、WAV、RM、MPEG 直接实现 IMedia 接口，而不通过 AudioMedia 和 VideoMedia，在满足其它需求的设计上也是不合理的。当然这已经不包括在本文的范畴了。

现在主程序界面发生了稍许的改变：

```
public void BtnPlay_Click(object sender,EventArgs e)
{
    IMediaFactory factory = null;
    switch (cbbMediaType.SelectItem.ToString().ToLower())
    {
        case ("mp3"):
            factory = new MP3MediaFactory();
            break;
        case ("wav"):
            factory = new WAVMediaFactory();
            break;
            //其他类型略;
    }
    MediaPlayer player = new MediaPlayer();
    player.Play(factory.CreateMedia());
}
```

写到这里，我们再回过头来看 MediaPlayer 类。这个类中，实现了 Play 方法，并根据传递的参数，调用相应媒体文件的 Play 方法。在没有工厂对象的时候，看起来这个类对象运行得很好。如果是作为一个类库或组件设计者来看，他提供了这样一个接口，供主界面程序员调用。然而在引入工厂模式后，在里面使用 MediaPlayer 类已经多余了。所以，我们要记住的是，重构并不仅仅是往原来的代码添加新的内容。当我们发现一些不必要的设计时，还需要果断地删掉这些冗余代码。

```
public void BtnPlay_Click(object sender,EventArgs e)
{
    IMediaFactory factory = null;
    switch (cbbMediaType.SelectItem.ToString().ToLower())
    {
        case ("mp3"):
            factory = new MP3MediaFactory();
            break;
        case ("wav"):
            factory = new WAVMediaFactory();
            break;
            //其他类型略;
    }
    IMedia media = factory.CreateMedia();
    media.Play();
}
```

如果你在最开始没有体会到 IMedia 接口的好处，在这里你应该已经明白了。我们在工厂中用到了该接口；而在主程序中，仍然要使用该接口。使用接口有什么好处？那就是你的主程序可以在没有具体业务类的时候，同样可以编译通过。因此，即使你增加了新的业务，你的主程序是不用改动的。

不过，现在看起来，这个不用改动主程序的理想，依然没有完成。看到了吗？在 BtnPlay_Click()中，依然用 new 创建了一些具体类的实例。如果没有完全和具体类分开，一

旦更改了具体类的业务，例如增加了新的工厂类，仍然需要改变主程序，何况讨厌的 switch 语句仍然存在，它好像是翅膀上滋生的毒瘤，提示我们，虽然翅膀已经从僵冷的世界里复活，但这双翅膀还是有病的，并不能正常地飞翔。

是使用配置文件的时候了。我们可以把每种媒体文件类类型的相应信息放在配置文件中，然后根据配置文件来选择创建具体的对象。并且，这种创建对象的方法将使用反射来完成。首先，创建配置文件：

```
<appSettings>
      <add key="mp3" value="WingProject.MP3Factory" />
      <add key="wav" value="WingProject.WAVFactory" />
      <add key="rm" value="WingProject.RMFactory" />
      <add key="mpeg" value="WingProject.MPEGFactory" />
</appSettings>
```

然后，在主程序界面的 Form_Load 事件中，读取配置文件的所有 key 值，填充 cbbMediaType 组合框控件：

```
public void Form_Load(object sender, EventArgs e)
{
    cbbMediaType.Items.Clear();
    foreach (string key in ConfigurationSettings.AppSettings.AllKeys)
    {
        cbbMediaType.Item.Add(key);
    }
    cbbMediaType.SelectedIndex = 0;
}
```

最后，更改主程序的 Play 按钮单击事件：

```
public void BtnPlay_Click(object sender,EventArgs e)
{
    string mediaType = cbbMediaType.SelectItem.ToString().ToLower();
    string factoryDllName = ConfigurationSettings.AppSettings[mediaType].ToString();

    //MediaLibray为引用的媒体文件及工厂的程序集;
    IMediaFactory factory = (IMediaFactory)Activator.CreateInstance
        ("MediaLibrary",factoryDllName).Unwrap();
    IMedia media = factory.CreateMedia();
    media.Play();
}
```

现在鸟儿的翅膀不仅仅复活，有了可以飞的能力；同时我们还赋予这双翅膀更强的功能，它可以飞得更高，飞得更远！

享受自由飞翔的惬意吧。设想一下，如果我们要增加某种媒体文件的播放功能，如 AVI 文件。那么，我们只需要在原来的业务程序集中创建 AVI 类，并实现 IMedia 接口，同时继承 VideoMedia 类。另外在工厂业务中创建 AVIMediaFactory 类，并实现 IMediaFactory 接口。假设这个新的工厂类型为 WingProject.AVIFactory，则在配置文件中添加如下一行：

```
<add key="avi" value="WingProject.AVIFactory" />
```

而主程序呢？根本不需要做任何改变，甚至不用重新编译，这双翅膀照样可以自如地飞行！

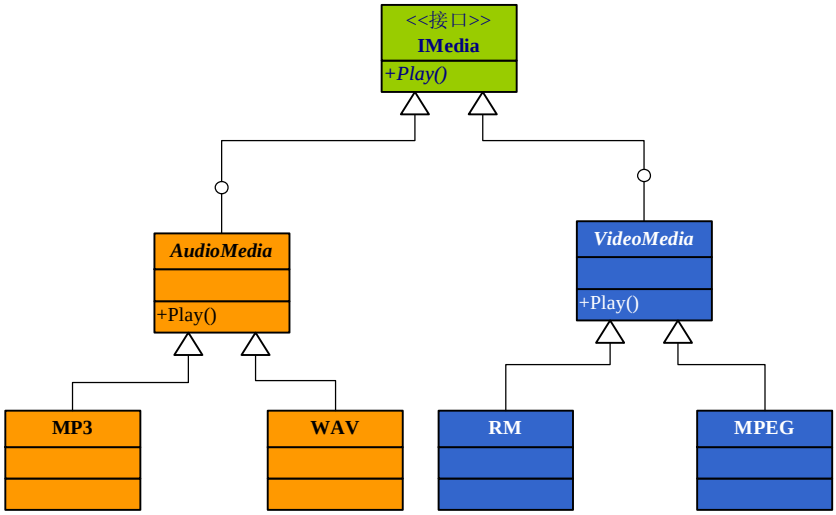
从实例谈 Adapter 模式

——《让僵冷的翅膀飞起来》系列之二

在系列一中，RM 和 MPEG 类继承了 VideoMedia 抽象类，而 VideoMedia 类又实现了 IMedia 接口，该接口仅仅提供了 Play()方法。如果我们需要为 RM 和 MPEG 提供与 AudioMedia 不同的属性和方法。例如，对于视频媒体而言，应该有一个调整画面大小的方法，如 Resize()。而这个方法则是 IMedia 接口所不具备的。

那么怎样为 RM，MPEG 类提供 IMedia 接口所不具备的 Resize()方法呢？非常自然地，通过这个问题我们就引出 Adapter 模式的命题了。首先，要假设一个情况，就是原文的所有代码，我们是无法改变的，这包括暴露的接口，类与接口的关系等等，都无法通过编码的方式实现新的目标。只有这样，引入 Adapter 模式才有意义。

熟悉 Adapter 模式的人都知道，Adapter 模式分为两种：类的 Adapter 模式、对象的 Adapter 模式。下面我试图根据本例对两种方式进行说明及实现。在实现 Adapter 模式之前，有必要看看原来的类结构：

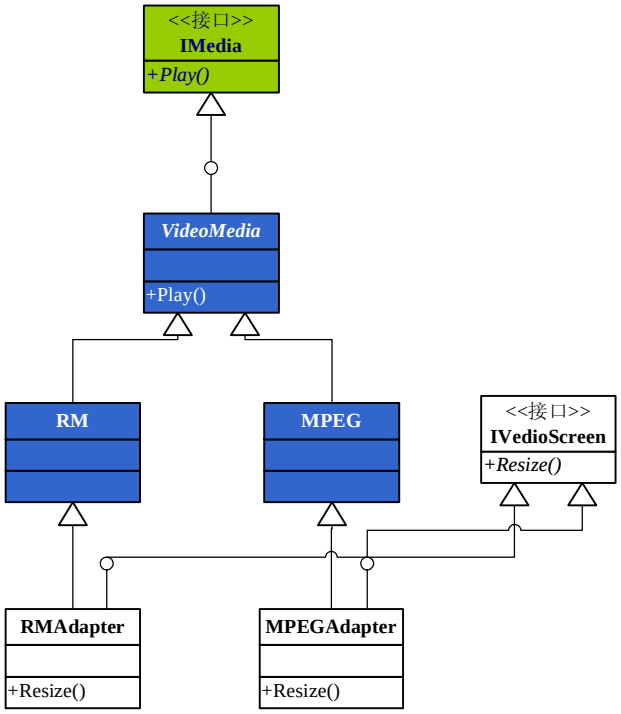


左边橙色的类为音频媒体类型，右边蓝色的类为视频媒体类型。所有的这些类型，包括类和接口都是无法改变的。现在我们的目的就是要让 RM、MPEG 具有 Resize()方法。那么首先定义一个接口 IVideoMedia，该接口具有 Resize()方法。

下面我们就根据 Adapter 模式来实现需求。

一、类的 Adapter 模式

既然要让 RM、MPEG 具有 Resize()方法，最好的办法就是让它们直接实现 IVedioScreen 接口。然而受到条件的限制，这两个类类型是不可修改的。唯一可行的办法就是为相应的类新引入一个类类型，这就是 Adapter 模式中所谓的 Adapter 类了。它好比是一个转接头，通过它去实现 IVedioScreen 接口，同时又令其继承原有的 RM 或 MPEG 类，以保留原有的行为。类图如下：



图中的类 RMAdapter 和 MPEGAdapter 就是通过 Adapter 模式获得的对象，它在保留了原有行为的同时，又拥有了 IVedioScreen 的功能。

代码如下：

```
public interface IVedioScreen
{
    void Resize();
}
public class RMAdapter:RM,IVedioScreen
{
    public void Resize()
    {
        MessageBox.Show("Change the RM screen's size.");
    }
}
public class MPEGAdapter:MPEG,IVedioScreen
{
    public void Resize()
    {
        MessageBox.Show("Change the MPEG screen's size.");
    }
}
```

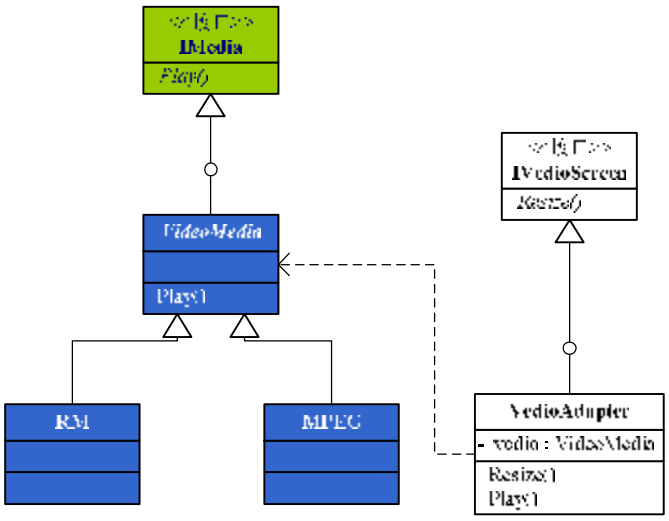
也许很多人已经注意到了，在使用这种方式建立 Adapter 时，存在一个局限，就是我们必须为每一个要包裹（Wrapping）的类，建立一个相应的 Adapter 类。如上所述的 RM 对应 RMAdapter，MPEG 对应 MPEGAdapter。必须如此，为什么呢？虽然 RM 和 MPEG 继承了同一个抽象类 VedioMedia，但其 Play()方法，可能是不相同的。此时，相对应的 Adpater 类

只有直接继承该具体类，方才可以保留其原来的 Play()方法的行为本质。

OOP 中很重要的思想是，尽量使用聚合而非继承。让我们换一种思路来考察 Adapter 模式。

二、对象的 Adapter 模式

对象的 Adapter 模式，与类的 Adapter 模式，最大的区别就是不采用继承的方式，而是将要包裹的对象，以聚合的方式放到 Adapter 中，然后用委托的方式调用其对象的方法，实现类图如下：



比较两种实现方式的类图，可以得出两个结论：

1、 对象的 Adapter 模式，减少了对象的个数；

2、 耦合度更加松散；

代码如下：

```
public interface IVedioScreen
{
    void Resize();
}
public class VedioAdapter:IVedioScreen
{
    private vedioMedia _vedio;

    public VedioAdapter(vedioMedia vedio)
    {
        _vedio = vedio;
    }
    public void Play()
    {
        _vedio.Play();
    }
    public void Resize()
    {
        if (_vedio is RM)
```

```
                MessageBox.Show("Change the RM screen's size.");
            else
                MessageBox.Show("Change the MPEG screen's size.");
        }
    }
```

以这种方式形成的 VedioAdapter，由于没有和 RM、MPEG 直接发生关系，并通过在构造函数传递参数的方式，等待客户端使用 Adapter 时，才将具体的 VedioMedia 对象传递给 Adapter，显得耦合度更加松散，更加灵活。

我们来看客户端调用时，两者的区别：

1、 类的 Adapter 模式

```
public class Client
{
    public static void Main()
    {
        RMAdapter rmAdapter = new RMAdapter();
        MPEGAdapter mpegAdapter = new MPEGAdapter();

        rmAdapter.Play();
        rmAdapter.Resize();
        mpegAdapter.Play();
        mpegAdapter.Resize();
    }
}
```

2、 对象的 Adapter 模式

```
public class Client
{
    public static void Main()
    {
        VedioAdapter rmAdapter = new VedioAdapter(new RM());
        VedioAdapter mpegAdapter = new VedioAdapter(new MPEG());

        rmAdapter.Play();
        rmAdapter.Resize();
        mpegAdapter.Play();
        mpegAdapter.Resize();
    }
}
```

其实，对于对象的 Adapter 模式，还可以做一些改进，就是用属性或方法来取代构造函数传递被包裹对象的方式。代码修改如下：

```
public class VedioAdapter:IVedioScreen
{
    private vedioMedia _vedio;

    public VedioMedia Vedio
```

```
        {
            set {_vedio = value;}
        }
        ……
    }
```

这样，上面的客户端调用就更简单了：

```
public class Client
{
    public static void Main()
    {
        VedioAdapter adapter = new VedioAdapter();
        adapter.Vedio = new RM();
        adapter.Play();
        adapter.Resize();

        adapter.Vedio = new MPEG();
        adapter.Play();
        adapter.Resize();
    }
}
```

通过运用 Adapter 模式，扩展了新的接口，而原有的类型并不需要做任何改变，这就是 Adapter 模式的实质，也是为什么取名为 Adapter 的原因之所在了。同时，我们要注意的是，在运用 Adapter 模式时，必须审时度势，根据具体的情况，抉择最优的方式，或者采用类的 Adapter 模式，或者采用对象的 Adapter 模式。决定权在与你，菜单给你送上来了，看看自己的腰包，想想点什么样的菜吧。

从 Adapter 模式到 Decorator 模式

——《让僵冷的翅膀飞起来》系列之三

一、考察对象的 Adapter 模式

从系列之二看到，经过引入 Adapter 模式，原有的结构得到了改进。但我们还需要从客户的角度分析程序，使结构更加地合理。（这里，我们仅限于考察对象的 Adapter 模式。类的 Adapter 模式不存在下述问题。这也印证了一个事实，就是：对象的 Adapter 模式和类的 Adapter 模式各有优势，也各有缺点，设计时应根据实际情况考察。）

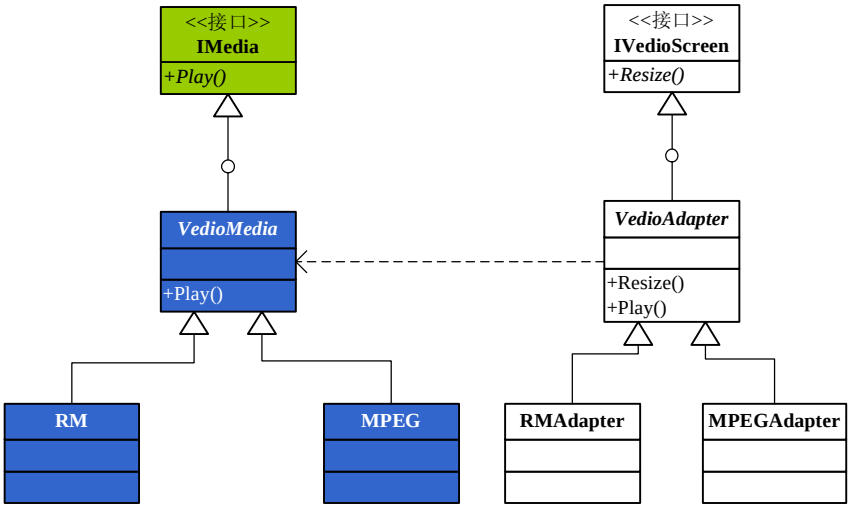
1、扩展的功能是否合理？

假设用户希望调用 VedioMedia 同时具有 Play()和 Resize()功能。从前面的描述来看，客户只需要实例化 VedioAdapter 类对象，就可以调用了。看来结构是正确的。

2、类型的扩展是否合理？

从目前的需求来看，要调用 RM 和 MPEG 类型的对象，没有任何问题。但是正如吕震宇所说，在 VedioMedia 类的 Resize()方法中有一股腐化的味道。坏味道的根源就是 if 条件语句。如果要增加新的视频类型，就需要修改 Resize()方法了。这是一个设计的权衡。其实这个味道虽然够坏，但好处是简单，也不用更多的对象；但耦合性比较差。

如果我们的目标是希望更好的架构以支持耦合的松散，目前的结构就需要微调了。调整后的类图如下：



这样需要改变 VedioAdapter 类的代码：

```
public abstract class VedioAdapter:IVedioScreen
{
    protected vedioMedia _vedio;

    public VedioAdapter(vedioMedia vedio)
    {
        _vedio = vedio;
```

```
        }

        public void Play()
        {
            _vedio.Play();
        }

        public abstract void Resize();
    }
```

然后实现 RMAdapter 和 MPEGAdatper：

```
    public class RMAdapter:VedioAdapter
    {
        public RMAdapter(VedioMedia vedio):base(vedio){}

        public override void Resize()
        {
            MessageBox.Show("Change the RM screen's size.");
        }
    }
    //MPEGAdapter 的代码省略
```

这样一改，要扩展就容易了，不过比之以前设计要复杂些，希望不会有人说我过度设计。如果要考虑正确性的话，RMAdapter 的构造函数还需要考虑异常情况。由于构造函数的参数为 VedioMedia 类型，因此，客户在调用时可能会传入 MPEG 类型，此时 RMAdapter 类型的 Play()行为就会发生改变。这也是和 Decorator 最大的不同，就是我们必须限制对象的 Play()方法不能做任何改变。

```
    public RMAdapter(VedioMedia vedio):base(vedio)
    {
        if (!(vedio is RM))
            throw new Exception("VedioMedia object is not correct!");
    }
```

3、 是否与原有客户系统兼容？

如果在原有的客户系统中提供了如下的类及方法：

```
    public class MediaFile
    {
        public static void Play(IMedia media)
        {
            media.Play();
        }
    }
```

那么客户如下的调用是没有任何问题的：

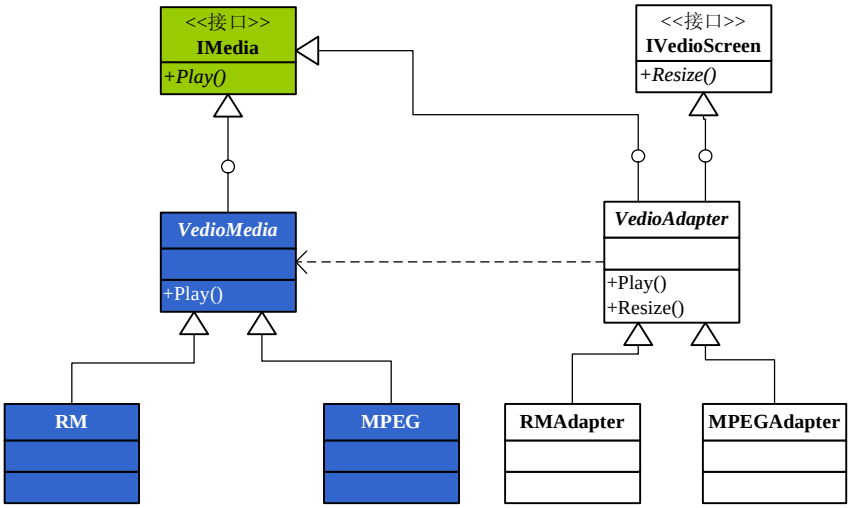
```
    MediaFile.Play(new RM());
```

然而， 当客户要使用新的 Adapter 对象呢？例如：

```
    MediaFile.Play(new RMAdapter());
```

显然是有问题了，因为 RMAdpater 类没有实现 IMedia 接口，且 RMAdpater 类的 Play()

方法和 IMedia 接口的 Play()方法在性质上也是有区别的。此时，采用原有的设计就不正确了。改进的方法很简单，就是让 VedioAdapter 类实现 IMedia 接口就可以了：



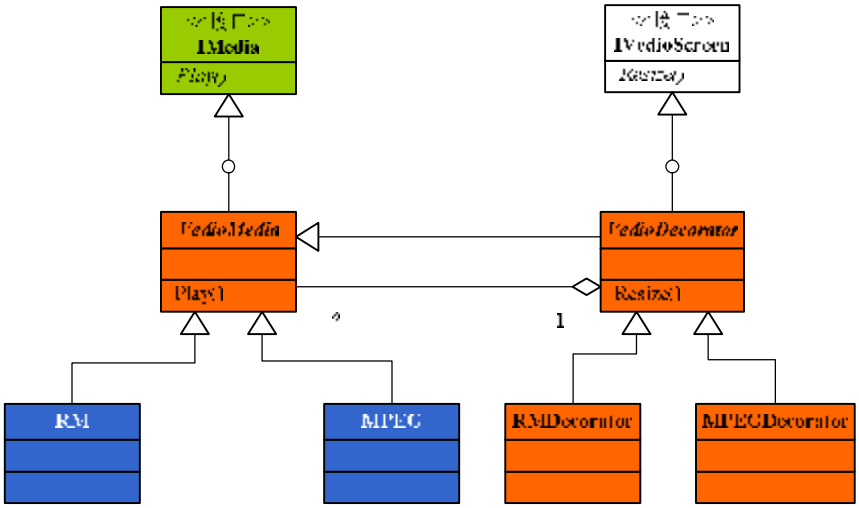

```
public abstract class VedioAdapter:IVedioScreen,IMedia
{
    //……
}
```

有人说，当 VedioAdapter 类实现 IMedia 接口时，言外之意就是该 Adapter 也适合 AudioMedia 类型了。是否如此呢？可以说是一半对，一半不对。对的原因，是由于 AudioMedia 类也实现了 IMedia 接口；但别忘了，我们适配的并非类，而是对象，也就是在 VedioAdapter 中传递进来的 VedioMedia 对象。（如果我们将传递进来的对象扩展为 IMedia，那就糟糕了。震宇兄的结论就完全成立了。）同时，我们在构造函数的异常处理，也保证了 AudioMedia 类型对于 VedioAdapter 是非法的。

写到这里，我觉得本文已经超出了原来的设想，有些研究的味道了。嗯，还算不错。那么就继续研究下去吧。

二、引入 Decorator 模式

按照最初的需求，我引入 Decorator 模式试一试。最初的需求是，需要为 RM 和 MPEG 类在不改变原有代码的情况下，添加 Resize()方法，而其原来的 Play()方法不变。调整设计类图：



上图的橙红色区域为 Decorator 模式的主体，至于 IVedioScreen 接口，仅仅是为 VedioDecorator 增加 Resize()方法而引入的，其本身与 Decorator 无关，除非我要实现的 Decorator 功能，通过该接口来实现。代码如下：

```
public abstract class VedioDecorator:VedioMedia,IVedioScreen
{
    private VedioMedia _vedio;
    public VedioAdapter(vedioMedia vedio)
    {
        _vedio = vedio;
    }
    public VedioMedia Vedio
    {
        get {return _vedio;}
    }
    public abstract void Resize();
}
```

注意看，与前面 Adapter 模式的 VedioAdapter 类比较，除增加了对 VedioMedia 的派生外，还减少了 Play()，因为该方法已经从 VedioMedia 类中派生获得。这样的话，RMDecorator 和 MPEGDecorator，也需要做相应的改变：

```
public class RMDecorator:VedioDecorator
{
    public RMDecorator (VedioMedia vedio):base(vedio)
    {
        if (!(vedio is RM))
            throw new Exception("VedioMedia object is not correct!");
    }
    public override void Play()
    {
        Vedio.Play();
    }
```

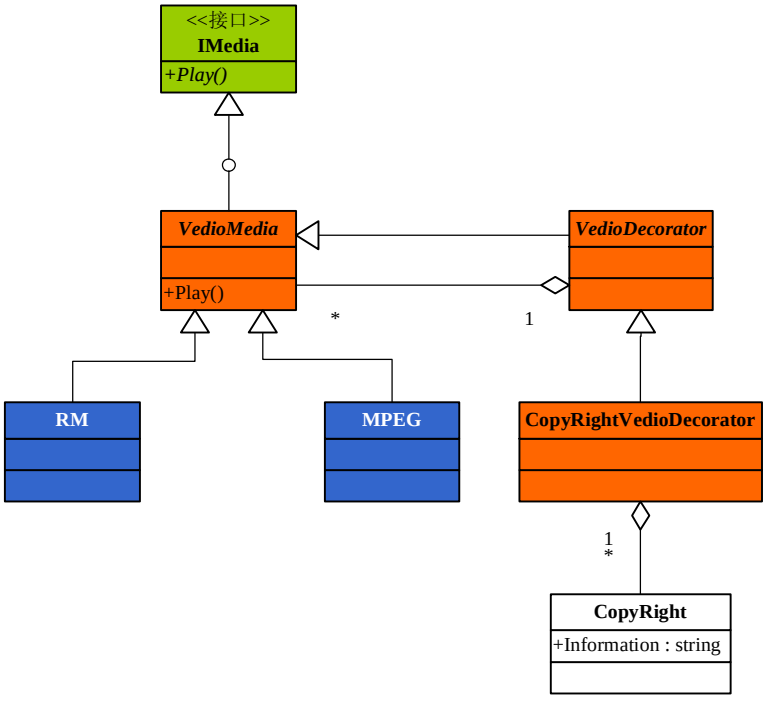
```
        public override void Resize()
        {
            MessageBox.Show("Change the RM screen's size.");
        }
    }
```

到这个时候，我忽然觉得引入 Decorator 模式，已经有些力不从心了。为什么呢？最大的障碍就是我们的需求不能更改 VedioMedia 类型 Play()方法的既有行为。这个时候，所谓的 Decorator 已经失去了原来的意义。其实我觉得，此时的设计，应该是结合了 Adapter 模式与 Decorator 模式而衍生出的新的结构。

那么，我为什么还要在本文提出引入 Decorator 模式呢。这来源我对于设计模式一向的观点：不要为了模式而模式！GOF 的 23 种模式，并非茴香豆的“茴”字，我也并非孔乙己，要你回答“茴”字的写法，却忽略了使用设计模式的真正精神。设计模式归根结底是拿来用的。只要符合你的要求，各种模式随你怎么变都可以。因此，不管是前文所述的 Adapter 模式，还是改进后的 Adapter 模式，或者引入的 Decorator 模式，其中的变化是灵活的，选择权最终还是你。

三、正宗的 Decorator 模式

不过，我还是很有兴趣继续探讨下去，仍然借助媒体播放这个例子，来谈一谈 Decorator 模式的一般应用。现在我们要求 RM 和 MPEG 媒体在播放前，首先要显示媒体文件的版权信息。请注意，这个需求，并非是为 RM 等媒体增加 ShowCopyright()方法，而 Play()方法保持不变。恰恰相反，新的需求装饰了 Play()的行为，它要求 Play()的同时能够支持 ShowCopyright 的功能。类图如下：



在这里，VedioDecorator 是装饰类的抽象类，而 CopyRightVedioDecorator 类则具体装饰了 Play()的功能。

```
    public abstract class VedioDecorator:VedioMedia
    {
```

```
        private VedioMedia _vedio;
        public VedioAdapter(vedioMedia vedio)
        {
            _vedio = vedio;
        }
        public VedioMedia Vedio
        {
            get {return _vedio;}
        }
    }
```

然后实现 CopyRightVedioDecorator 类，为 Play()方法装饰显示版权信息的功能：

```
    public class CopyRightVedioDecorator:VedioDecorator
    {
        private CopyRight _copyRightMark;
        public CopyRight CopyRightMark
        {
            get {return _copyRightMark;}
            set {_copyRightMark = value;}
        }
        public override void Play()
        {
            _copyRightMark.ShowCopyRight();
            Vedio.Play();
        }
    }
```

我们还可以继续装饰 VedioMedia 的 Play 行为，例如，要求在播放媒体文件之前，必须放一段广告，那么我们可以继续提供一个 AdvertisementVedioDecorator 装饰类。道理与上一样，不再赘述。

通过本例，我们可以看到 Decorator 模式与对象的 Adapter 模式的区别。

实现的区别：

1、 Decorator 抽象类应继承要装饰的类，同时又聚合该类的实例对象；而对象的 Adapter 模式则只聚合，不继承；

2、 Decor 模式并没有引入新的接口，除非要装饰的行为需要使用该接口；而对象的 Adapter 模式则引入了新的接口，以此来装配原有的对象，使其具有了新接口的方法；

因此，适用的场景也就有所不同：

1、 Decorator 模式如其名，一般并不提供新的行为，而是在原有的行为上进行补充，即装饰的含义。

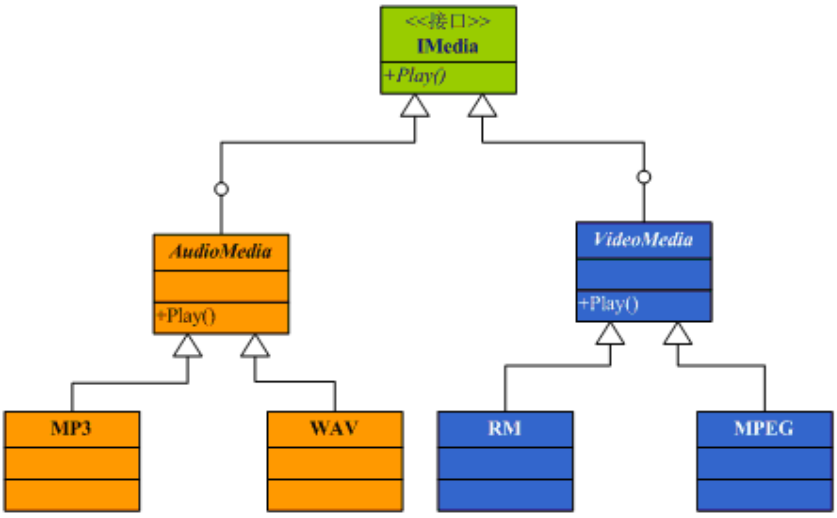
2、 Adapter 模式则是为对象引入新的行为，使其匹配新的接口，即为适配的意义所在。

Visitor 模式之可行与不可爱

——《让僵冷的翅膀飞起来》系列之四

可否使用 Visitor 模式实现系列二的例子呢？经过我的悉心研究，结论是 Visitor 模式对于本例而言，可行但不可爱！

我们先看看本例最初的类图：



我们可以将这个类图看作是一个树形结构。那么各个类即可以看作是树的枝叶节点了。这样一说，似乎和 Composite 模式有些关系了。其实不然，因为我所谓的树枝节点，如 AudioMedia 类，与 MP3 及 WAV 类并非是聚合的关系。认识到这一点很重要，为避免混淆视听，我将这些类只称作节点好了，就别提树枝树叶，避免产生误会。

而 Play()以及 Resize()方法，就可以认为是这些节点共同的行为，然而其行为的内涵则是不相同的。也就是说，RM 和 MPEG 虽然都有 Play()方法，但 Play 的操作不同。也就是说，现在，我们可能会为这些不同的媒体类型不断提供更多的行为。而 Visitor 模式呢？其优势就在于：

为各节点增加新的行为，变得非常的容易。

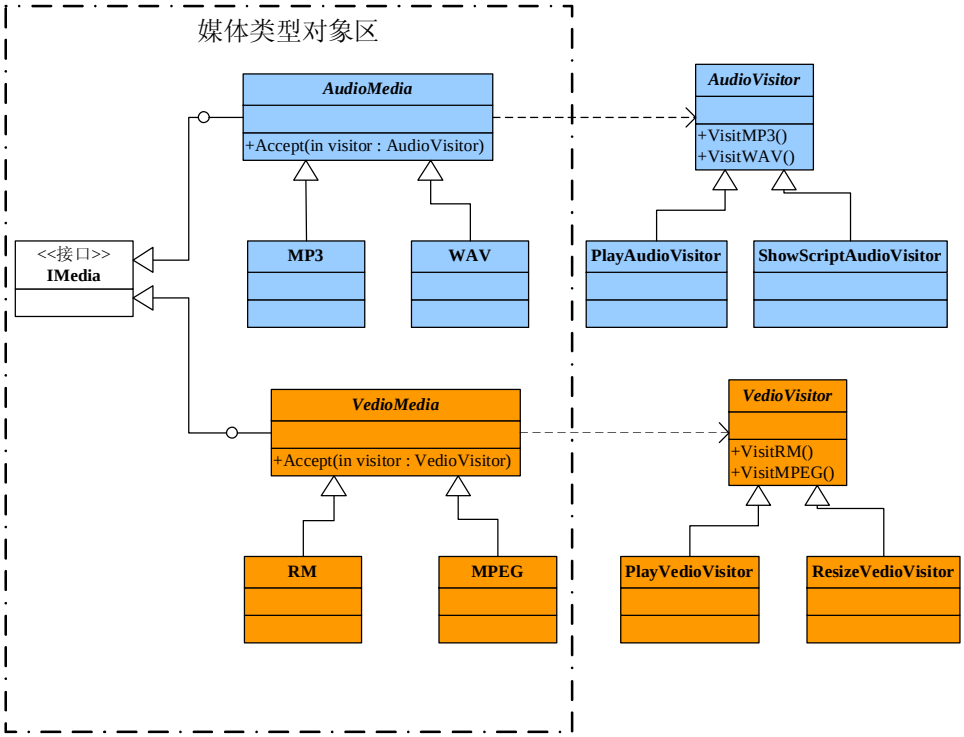
也就是说，Visitor 模式对于本例而言，是可行的。但这破坏了一个前提，就是如果适用 Visitor 模式必须要更改原来的设计架构，除非你愿意再为这些媒体类分别进行 Wrap！好吧，既然是出于研究目的，我们就来看看 Visitor 模式的应用。修改前提为：

重构原来的设计，使旧有的媒体播放器，能够非常容易地扩充行为。

使用 Visitor 模式，最大的特点是将被访问对象（通常称为节点或元素）的行为，与其对象分离。并用专门的 Visitor 对象，管理节点的行为。在本例中，抽象节点包括：AudioMedia 和 VedioMedia。其下的具体节点分别为：MP3、WAV 和 RM、MPEG。那么在 Vistor 对象，就应该包括四个访问行为。

是这样吗？看来还有些问题。从实际需求来分析，本例的节点应分为两大类型：视频媒体和音频媒体。而这两类媒体的行为方法，可能是不相同的。例如，视频媒体除了要求 Play()方法外，可能还需要 Resize()方法。而音频媒体就没有必要使用 Resize()方法了，反而会需

要视频媒体不需要的方法 ShowScript()。因此结论是：我们应该分别为 AudioMedia 和 VedioMedia 建立不同的抽象 Visitor 类。最后获得应用 Visitor 模式的类图如下：



其实此时 IMedia 接口已经可以去掉，我之所以保留出于两个目的：

1、为两种类型媒体保持类型的抽象；

2、可以提供两种类型媒体同时具有，且不需要 Visitor 的行为；

（注：此时的 Play 方法其实可以放到 IMedia 接口中。我仍然将其放到 Visitor 的目的是，便于说明 Visitor 模式，同时通过这种方式，使对 Play 方法的修改较为容易。）

请大家注意上图的类图左侧，即我定义的媒体类型对象区，在这种 Visitor 模式设计的前提下，对于类型的行为来说，是非常稳定的。即：无论你要为这四种媒体类型增加什么行为，都不需要修改该区域的任何类或接口；而你只需要在 Visitor 下增加行为的具体 Visitor 类即可。这就符合了 OO 思想中非常著名的开—闭原则。左区即为相对的闭区间（我不能说是绝对的对修改关闭，因此，我才用虚线来作界定），而右区则为相对的开区间。

还是来看看源代码：

首先是被访问的类型：

```
public interface IMedia
{}
public abstract class AudioMedia:IMedia
                                        {
    public abstract void Accept(AudioVisitor visitor);
}
public class MP3:AudioMedia
{
    public override void Accept(AudioVisitor visitor)
    {
        visitor.VisitMP3();
```

```
        }
    }
    public class WAV:AudioMedia
    {
        public override void Accept(AudioVisitor visitor)
        {
            visitor.VisitWAV();
        }
    }
    public abstract class VedioMedia,IMedia
                                          {
        public abstract void Accept(VedioVisitor visitor);
    }
    public class RM:VedioMedia
    {
        public override void Accept(VedioVisitor visitor)
        {
            visitor.VisitRM();
        }
    }
    public class MPEG:VedioMedia
    {
        public override void Accept(VedioVisitor visitor)
        {
            visitor.VisitMPEG();
        }
    }
```

请注意，在我的 Visitor 方法中，并没有将节点或元素对象传递给 Visitor。这一点，与通常的 Visitor 模式实现小有区别。

下面是 Visitor 的实现：

```
    public abstract class AudioVisitor
    {
        public abstract void VisitMP3();
        public abstract void VisitWAV();
    }
    public class PlayAudioVisitor:AudioVisitor
    {
        public override void VisitMP3()
        {
            //实现MP3的Play方法；
        }

        public override void VisitWAV()
        {
```

```
            //实现WAV的Play方法；
        }
    }
    public class ShowScriptAudioVisitor:AudioVisitor
    {
        public override void VisitMP3()
        {
            //实现MP3的ShowScript方法；
        }

        public override void VisitWAV()
        {
            //实现WAV的ShowScript方法；
        }
    }
    public abstract class VedioVisitor
    {
        public abstract void VisitRM();
        public abstract void VisitMPEG();
    }
    public class PlayVedioVisitor:VedioVisitor
    {
        public override void VisitRM()
        {
            //实现RM的Play方法；
        }

        public override void VisitMPEG()
        {
            //实现MPEG的Play方法；
        }
    }
    public class ResizeVedioVisitor:VedioVisitor
    {
        public override void VisitRM()
        {
            //实现RM的ShowScript方法；
        }

        public override void VisitMPEG()
        {
            //实现MPEG的ShowScript方法；
        }
    }
```

现在我解释一下为什么不在 Visitor 方法中传递节点对象。从上面的 Visitor 实现，可以看到。每个 Visitor 的 Visit 方法，实际上代表的就是各自的行为，或者是 Play，或者是 Resize，等等。而这些行为均是在 Visitor 中实现的，而非它访问的节点对象。也就是说，通过 Visitor 模式，我将各个媒体对象的行为都交给 Visitor 了。既然干活的对象发生了转移，那么发生了什么责任，也就去找 Visitor 吧，这个责任可与媒体对象本身没有关系了啊。

根据 Visitor 模式，一般还应该提供结构对象(ObjectStructure)角色。然而我在开篇名义之处，就提到本例中，媒体类型对象是不存在聚合关系的，因此不需要劳烦 ObjectStructure 来枚举每个节点了。也许，这个 Visitor 模式有些不伦不类吧，没关系，我们只需要了解这个模式的思想。

从容自若的 CTO

——《让僵冷的翅膀飞起来》系列之五

让我们假设这样一个场景：一年以前，Media 公司开发出一套通过电脑接收广播的 Radio 仿真软件产品。（有这样的产品吗，能真正接收广播的软件？我表示怀疑）这个产品早已投入市场，客户已经在使用了。后来，Media 公司将开发重心转移到数字媒体上。于是他们投入了大量的人力物力，最后开发出了完美的媒体播放器软件。这个播放器支持大多数媒体文件，包括音频媒体和视频媒体。该产品取得了成功，也得到了用户的好评。

不过，现实生活中总有些刁钻的客户，比如说 wayfarer，就是鄙人了，素爱怀旧。在使用媒体播放器的时候，想起了在初中的时候就使用的收录机。磁带、广播，一机两用，真是令人怀念。于是我向 Media 公司提出了建议，希望能在媒体播放器中增加收音的功能。Media 的 CEO 对这个似乎有些嗤之以鼻。可是像 wayfarer 这样的用户越来越多，呼声也越来越高。为了产品的市场，为了公司的前景，这位 CEO 不得不慎重考虑这个需求了。当首席执行官就是好，赶紧把这个烫手山芋抛给了 CTO。

却看这位 CTO 仍然是从容不迫，脸上挂满自信的微笑。CEO 不解，问他何故如此从容？CTO 淡然一笑，吐出一字真言：“Adapter”。

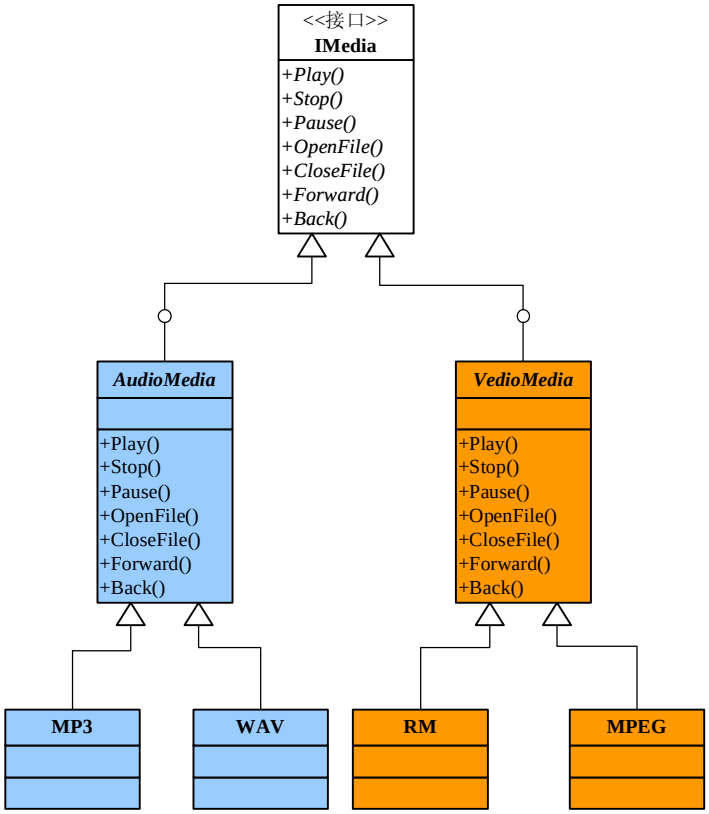
呵呵，笑话了。设计模式可不是什么 Bible，也非神奇的魔咒。不过对于以上场景，使用 Adapter 却是最佳的应用！且请听我慢慢道来。

已有产品：MediaPlayer、RadioPlayer；

分析：MediaPlayer 是面向客户的外观，即表示层，它调用了对应的业务层，该层实现了 IMedia 接口。同理 RadioPlayer 也是面向客户的外观，它调用的业务层，是收听广播的业务，并实现了 IRadio 接口。

目的：将 RadioPlayer 的业务添加到 MediaPlayer 的外观中。原有的 RadioPlayer 不再使用。

既然与 MediaPlayer、RadioPlayer 的业务有关，所以我们有必要分析其各自的业务结构。MediaPlayer 业务层结构：



为了简化，我这里将所有的方法都放在一个接口 IMedia 里（这个设计还有很多重构的空间，我会在后续文章中继续关注）。在本文的结构中，视频媒体和音频媒体的方法是相同的，本来我可以令各媒体文件继承同一个抽象类 Media。但现实情况显然不是这样，所以我仍然保留这个系列文章中原有的结构。以下是每个方法的说明：

Play()：播放媒体文件；

Stop()：停止播放；

Pause()：暂停播放；

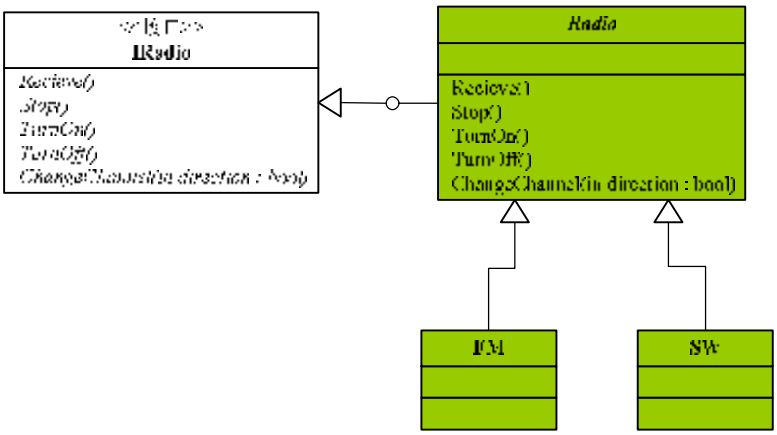
OpenFile()：打开媒体文件；

CloseFile()：关闭媒体文件；

Forward()：前进播放文件；

Back()：后退播放文件；

OK，我们再来看看 RadioPlayer 的业务层结构：



RadioPlayer 的业务均抽象为 IRadio 接口。并由抽象类 Radio 实现该接口。FM 为调频收音，SW 为短波收音。另外还有其他的，例如中波等，就不在详细列出。各方法的功能说明如下：

Receive()：接收广播；

Stop()：停止接收广播；

TurnOn()：打开收音；

TurnOff()：关闭收音；

ChangeChannel(bool direction)：切换频率。参数 direction 为 true 时，则往上；否则往下。当然也可以使用枚举类型。

媒体播放器的业务由一个统一的 Client 类进行处理，它包括一系列的静态方法以实现对原有媒体类型的调用：

```
public class Client
{
    public static void Play(IMedia media)
    {
        media.Play();
    }
    public static void OpenFile(IMedia media)
    {
        media.OpenFile();
    }
    //……其他方法略；
}
```

MediaPlayer 播放器本身，其外观则是一个 WinForm 应用程序，该应用程序将调用 Client 的相关静态方法。如：

```
Client.Play(new MP3());
```

现在看看我们需要实现的。我需要将 RadioPlayer 的业务，即抽象为 IRadio 接口的对象，放到 MediaPlayer 中。糟糕的是，Client 的各个方法传递的参数类型，为 IMedia 接口。怎么才能将实现 IRadio 接口的对象传递到 Client 的方法中呢？对了，这就是适配，就是为 IRadio 对象适配成符合 IMedia 接口行为的过程。打一个不好听的比方，就好比一只狼，要让自己钻进羊群里，而不被发现，就需要找一张羊皮来披上。俗语云：“披着羊皮的狼”是也。不过，我们要注意的是，狼虽然不是羊，但有着和羊相似的属性。它和羊体形相似，照样能跑，

能吃，只是吃的不是草，而是肉而已。你总不能为一张桌子披上羊皮，去装羊吧。而文中的 IMedia 类型和 IRadio 类型，还是有很多相似之处的。

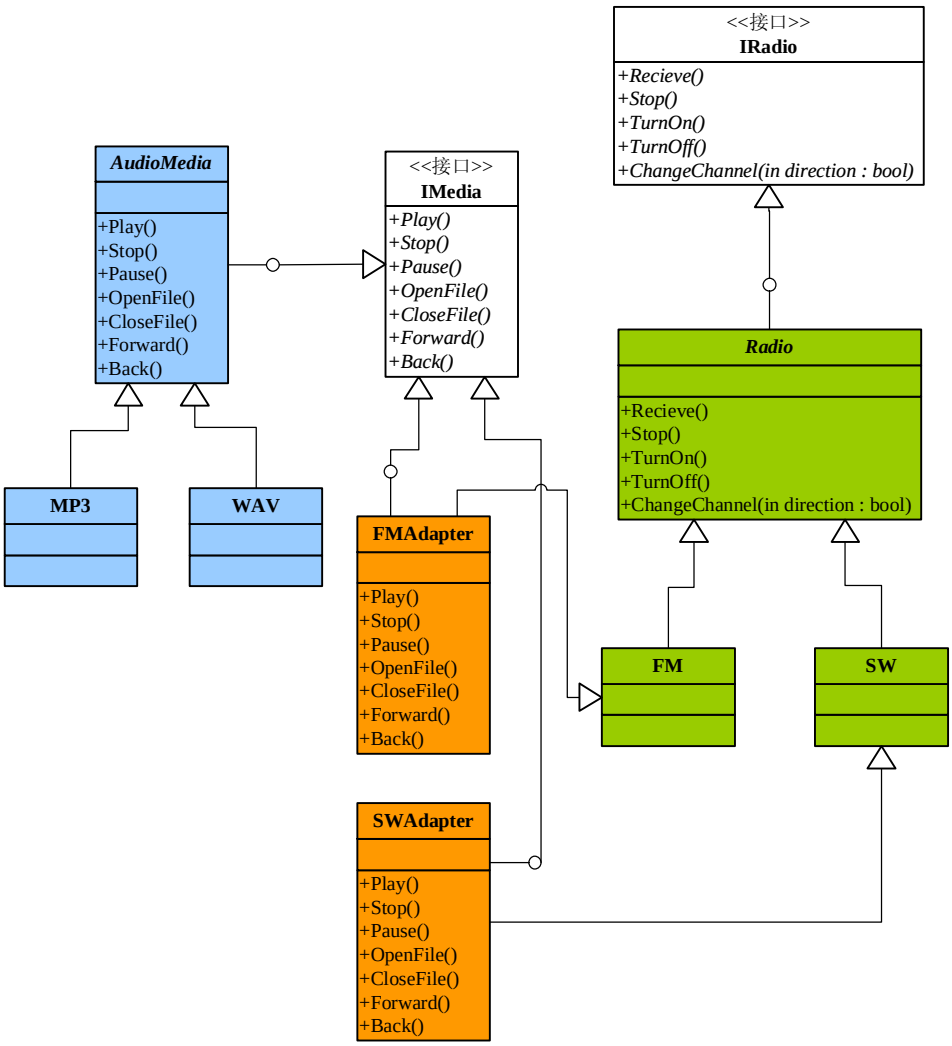
现在，我们就为 IRadio 接口进行适当的包装。由于这是两个接口进行匹配的过程，所以我们通常名之为“适配”，而非“包裹”。那么它们之间有相似性吗？有！

IMedia	**IRadio**
Play()	Receive()
Stop()	Stop()
OpenFile()	TurnOn()
CloseFile()	TurnOff()
Forward()	ChangeChannel(true)
Back()	ChangeChannel(false)

当然现实情况并非总是那么完美。可能 IMedia 的方法中，IRadio 可能并不需要。没关系，我们只提供该方法就可以了，方法的实现可以为空，如 Pause()方法。也有可能 IRadio 的一些方法，IMedia 并没有，此时的 Adaptor 模式，就将被适配对象的接口变宽了，也就是说引入了新的行为，这就类似于我系列文章之二所描述的。

不管现实的某些情况是多么的不如意，但至少通过引入 Adapter 模式，我们就不需要改变原有的 IMedia 和 IRadio 的相关对象与业务了。要修改的，仅仅是客户端，以及增加一个新的 Adapter 结构而已。

分析结束，开始动手术吧。先看类的 Adapter 模式：



类图好像很复杂，不过请大家主要关注橙色的两个类 FMAdapter 和 SWAdapter。FMAdapter 类是 FM 类型的 Adapter，它继承了 FM 类，并实现了 IMedia 接口。通过这种方式，原有的 FM 类型的行为，就被适配为符合 IMedia 类型的新类型。代码如下：

```
public class FMAdapter:FM,IMedia
{
    public void Play()
    {
        this.Receive();
    }
    public void Forward()
    {
        this.ChangeChannel(true);
    }
    public void Pause(){}//Radio类型没有该行为，令其为空，或引入异常机制；
    //其他方法略
    ……
}
```

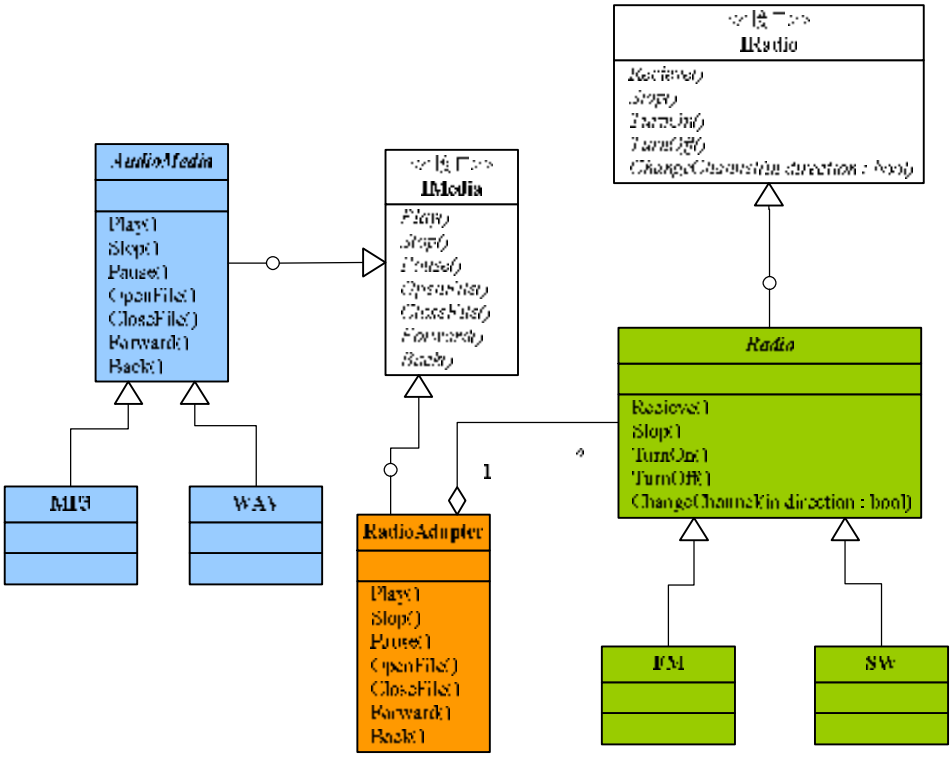
SWAdapter 的实现方式完全相同，就不赘述。

由于新的 Adapter 类均实现了 IMedia 接口，因此，该类型的对象可以安全正确地作为 Client 静态方法的参数对象传入。从外部行为的表现来看，没有区别。如：

```
Client.Play(new FMAdapter());
```

它调用了 FMAdapter 的 Play 方法，而其内部，实质上调用的是 FM 的 Receiver()方法。

再看对象的 Adapter 模式，就更简单了。



只需要一个 Adapter 类 RadioAdapter，然后实现 IMedia 接口。没有继承关系了，而是聚合了 Radio 对象。注意，这里聚合的是抽象类对象 Radio，而不是具体的 FM 或 SW。

```
public class RadioAdapter:IMedia
{
    private Radio _radio;
    public RadioAdapter(Radio radio)
    {
        this._radio = radio;
    }
    public void Play()
    {
        _radio.Receive();
    }
    public void Forward()
    {
        _radio.ChangeChannel(true);
    }
```

```
        public void Pause(){}//Radio类型没有该行为，另其为空，或引入异常机制；
        //其他方法略
        ……
    }
```

调用 Client 的静态方法：

```
Client.Play(new RadioAdapter(new FM()));
```

通过引入 Adapter 模式，我们在不改变原有 IMedia 和 IRadio 的情况下，顺利地将 IRadio 类型适配成了 IMedia 类型。此时，我们只需要在 MediaPlayer 的客户端调用中加入原来 RadioPlayer 的业务即可，基本保证了原有系统的稳定性。

上述实例，才真正体现了 Adapter 的价值（请大家一定注意区分本文实例需求，与系列之二实例需求的区别）。因此，我们可以得到两个结论：

1、通过 Adapter 模式，为适配对象引入以前不具备的行为；此时建议使用类的 Adapter 模式。理由请参考：系列文章之二与之三；

2、将一个固有对象适配为另一种接口对象；这是 Adapter 模式最重要的功能。使用类的 Adapter 模式与对象的 Adapter 模式均可，但感觉使用对象的 Adapter 模式更简单。

怎么样，够简单吧？难怪我们的 CTO 如此从容，因为他已经找到了终南捷径！

Strategy 模式的应用实践

在工作中，我需要写一个话单转换工具，在写这个工具的过程中，发现整个实现恰恰可以说是策略模式最好的体现。用这个例子来说明策略模式的应用，最合适不过了。

话单转换工具的目的：将某个服务提供商的话单文本文件，转换为另一个服务提供商的话单文本文件。如将联通的话单格式转换为移动的话单格式。

话单转换工具的要求：能够实现多个服务提供商话单文本文件的互相转换。

我们首先来分析一下任务。首先，各种服务提供商的话单格式，无疑是不相同的。例如，话单采集后，字段的顺序，字段的宽度以及字段间的分割符，都不相同。但是，从总体上来说，话单的表现形式是大致相同的，这位我们实现话单转换提供了一个技术上可行的前提。

所谓话单转换，就是需要将一个话单文本文件读出，然后对每一行的字符串进行识别后，再转换为符合相对应的服务提供商标准的话单文本文件。操作很简单，就是文本文件的读写而已，不同的就在于转换的方法。根据服务提供商标准的不同，我们应该为每一种转换提供相对应的算法。而所谓策略模式，正是对算法的包装和抽象，将算法的责任和其本身分离。所以，我们现在要做的工作就是将转换话单的算法抽象出来。

根据工具的要求，话单转换应该包括 3 个方法：

1、将文件读出的一行字符串转换为对应的话单对象；

2、将一种话单对象转换为另一种话单对象；

3、将话单对象转换为字符串，以方便写入话单文本文件；

根据以上的分析，我们还需要为不同的话单格式建立相应的对象。例如：网通、联通和移动的话单格式对象分别为：

```
public class CNCCdr
{
    //网通话单格式对象的公共属性;
}
public class CUCCdr
{
    //联通话单格式对象的公共属性;
}
public class CMCdr
{
    //移动话单格式对象的公共属性;
}
```

接下来就应该实现话单转换的算法了。首先需要将算法进行抽象，而进行抽象的最佳选择莫过于使用接口，例如我们定义一个用于话单转换的接口 ICdrConvert：

```
public interface ICdrConvert
{
}
```

按照前面的分析，在接口中应该包括三个转换方法。但是现在有个问题，就是转换的话单对象。由于方法在接口中，为一个抽象。而话单对象可能有多种，具体转换为何种话单对

象，需要到具体实现时才能决定。因此，在接口方法中，无论返回类型，还是传入参数，涉及到的话单对象只能是抽象的。也许我们可以考虑 System.Object 来表示话单对象，但更好的办法是为所有的话单对象也提供一个抽象接口。

由于从目前的分析来看，抽象话单对象并没有一个公共的方法，所以这个抽象的话单接口，是一个标识接口：

```
public interface ICdrRecord
{
}
```

现在，可以对转换接口的方法进行定义了：

```
public interface ICdrConvert
{
    ICdrRecord Convert(string record);
    ICdrRecord Convert(ICdrRecord record);
    string Convert(ICdrRecord record);
}
```

自然，这样的接口定义无法通过编译。为什么呢？是因为第二个方法的签名与第三个方法的签名重复了（方法的签名和返回类型无关）。因此，我们需要为第三个方法修改名字。

但我们仔细想想，第三个方法的转换在转换接口中是必要的吗？该方法的任务是将一个话单对象转换为 string 类型。实际上这个责任，并不需要专门的转换对象来完成，而应属于话单对象本身的责任。再想想.Net 中，所有对象均派生于 System.Object，而 object 类型均提供了 ToString()方法。

从设计的角度来看，最好的办法，是在具体的话单对象中 override System.Object 的 ToString()方法，而不是在转换对象中，提供该转换算法。

不过考虑到，在话单处理中，更多地会调用抽象话单接口类型的对象，也许我们将 ToString()方法抽象到接口 ICdrRecord 中会更好。

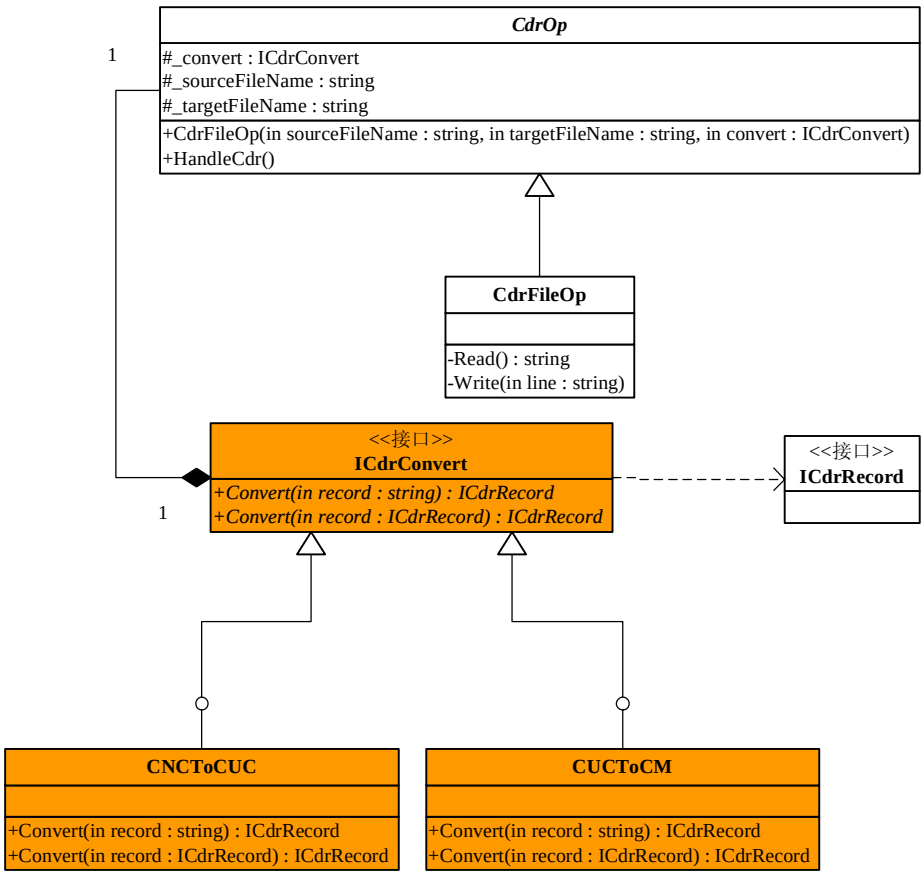
```
public interface ICdrRecord
{
    string ToString();
}
public interface ICdrConvert
{
    ICdrRecord Convert(string record);
    ICdrRecord Convert(ICdrRecord record);
}
```

而具体的话单对象，就应该实现 ICdrRecord 接口了。因为各话单对象均继承了 System.Object，则间接地继承了 object 对象的 ToString()方法，所以，话单对象应该重写该方法：

```
public class CNCCdr: ICdrRecord
{
    //网通话单格式对象的公共属性；

    //重写object的ToString()方法，同时也实现了接口ICdrRecord的ToString()方法；
    public override string ToString()
    {
```

```
            //实现具体的内容；
        }
    }
    public class CUCCdr: ICdrRecord
    {
        //联通话单格式对象的公共属性；

        //重写object的ToString()方法，同时也实现了接口ICdrRecord的ToString()方法；
        public override string ToString()
        {
            //实现具体的内容；
        }
    }
    public class CMCdr: ICdrRecord
    {
        //移动话单格式对象的公共属性；

        //重写object的ToString()方法，同时也实现了接口ICdrRecord的ToString()方法；
        public override string ToString()
        {
            //实现具体的内容；
        }
    }
```

下面就是关键的实现了。由于我们已经为转换算法进行了抽象，因此根据策略模式来实现具体的转换算法，就是水到渠成的事情了。实现代码之前，先来看看 UML 类图：



注意看橙色部分，这一部分即为策略模式的主体。接口 ICdrConvert 为抽象策略角色，类 CNCToCUC，CUCToCM 为具体策略角色，它们分别实现了将网通话单转换为联通话单，联通话单转换为中国移动话单的算法。根据实际需要，还可以添加多个类似的具体策略角色，并实现 ICdrConvert 接口：

```
public class CNCToCUC:ICdrConvert
{
    public ICdrRecord Convert(string record)
    {
        //实现具体的转换算法;
    }
    public ICdrRecord Convert(ICdrRecord record)
    {
        //实现具体的转换算法;
    }
}
```

类 CUCToCM 的实现相似，不再重复。

那么通过策略模式实现，究竟有什么好处呢？请大家注意上图的 CdrOp 类。该类是抽象类，它提供了一个构造函数，可以传递 ICdrConvert 对象：

```
public abstract class CdrOp
{
```

```
    protected ICdrConvert _convert;
    protected string _sourceFileName;
    protected string _targetFileName;

    public CdrOp(string soureFileName,string targetFileName,ICdrConvert convert)
    {
        _sourceFileName = soureFileName;
        _targetFileName = targetFileName;
        _convert = convert;
    }

    public abstract void HandleCdr();
}
```

类 CdrFileOp 继承了抽象类 CdrOp：

```
public class CdrFileOp
{
    public override void HandleCdr()
    {
        Read();
        Write();
    }
    private string Read()
    {
        using (StreamReader sd = new StreamReader(_sourceFileName))
        {
            ICdrRecord cdr = null;
            string line;
            while ((line = sd.ReadLine()) != null)
            {
                //首先调用ICdrRecord Convert(string record)方法；
                //再调用ICdrRecord Convert(ICdrRecord record)方法；
                //至于实现的是何种转换，由构造函数传入的ICdrConvert对象决定；
                cdr = _convert.Convert(_convert.Convert(line));
                _list.Add(cdr);
            }
        }
    }
    private void Write()
    {
        using (StreamWriter sw = new StreamWriter(_targetFileName,true))
        {
            sw.Write(head.ToString());
            foreach (ICdrRecord record in _list)
            {
```

```
                    sw.Write(record.ToString());
                }
            }
        }
        private ArrayList _list = new ArrayList();
    }
```

这个类，实现了抽象类 CdrOp 的 HandleCdr()方法。但具体的实现细节则是在私有方法 Read()和 Write()中完成的（根据实际情况，也可以把 Read 和 Write 方法作为公共抽象方法或保护方法，放到抽象类 CdrOp 中，而在抽象类 CdrOp 中具体提供 HandleCdr 方法的实现，该方法调用 Read 和 Write 方法，这样就使用了模版方法模式）。

注意看 Read 和 Write 方法中，没有一个具体类。不管是话单对象，还是话单转换对象，均是抽象接口对象。尤其是在 Read()方法中，调用了_Convert 的 Convert 方法：

```
cdr = _convert.Convert(_convert.Convert(line));
```

内部的 Convert 方法，即_convert.Convert(line)，是将读出来的字符串转换为 ICdrRecord 对象，然后通过调用_convert.Convert(ICdrRecord record)方法，再将该对象转换为另一种话单格式对象，但类型仍然属于 ICdrRecord。

那么在这些转换过程中，究竟转换成了哪一种话单格式对象呢？这是由_convert 字段来决定的。而这个对象则是由构造函数的参数中传递进来的。

同样的道理，在 Write()方法中，大家也可以看到为所有话单对象抽象为一个接口 ICdrRecord 的好处。通过 ICdrRecord 调用 ToString()方法，避免了在 CdrFileOp 中引入具体对象。要知道，程序一旦引入具体对象，则耦合性就高了。一旦需求发生改变，就需要对编码进行修改。

有了以上的架构，客户端调用就非常方便了：

```
public class Client
{
    public static void Main()
    {
        string sourceFileName = @"c:\CNCCdr.txt";
        string target1= @"c:\CUCCdr.txt";
        string target2= @"c:\CMCdr.txt";

        //将网通话单转换为联通话单；
        ICdrOp op1 = new CdrFileOp(sourceFileName,target1,new CNCToCUC());
        op1.HandleCdr();

        //将刚才转换生成的联通话单转换为移动话单；
        ICdrOp op2 = new CdrFileOp(target1,target2,new CUCToCM());
        op2.HandleCdr();
    }
}
```

当然，我们还可以引入工厂模式来创建 CdrFileOp 对象。或者将 ICdrConvert 对象设置为 CdrFileOp 的公共属性，而非通过构造函数来传入。然而，通过本文，策略模式的精要已经体现得淋漓尽致了。

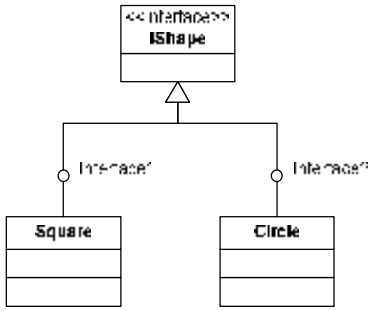
Factory Method 模式

——《.Net 中的设计模式》系列

一、模式概述

也许 Factory Method 模式是设计模式中应用最广泛的模式。在面向对象的设计中，关于对象的管理是其核心所在，而其中对象的创建则是对象管理的第一步。对象的创建非常简单，在 C#中，只需要应用 new 操作符调用对象的构造函数即可，然而创建对象的时机却非常重要。

首先我们从对象的特征来看，代表抽象关系的类型，如接口和抽象类，是不能创建的，换句话说，我们要创建的对象都是与具体的对象类型有关。因此，对象的创建工作必然涉及到设计中的实现细节，从而导致创建者与具体的被创建者之间耦合度增强。举例来说，如果在一个项目中我们需要创建一些图形对象，例如 Circle、Square。这些对象的结构如下：

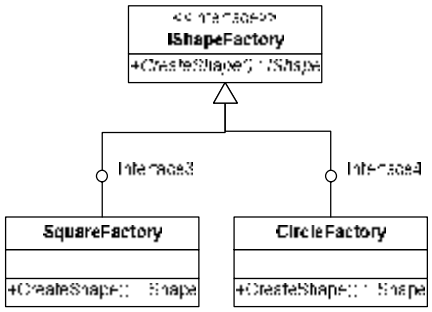


这个结构是非常符合 OO 思想的，它通过 IShape 接口将 Square 和 Circle 对象抽象出来，根据多态的原理，我们完全可以在程序中用 IShape 来代替具体的 Square 和 Circle 类，从而将具体的对象类型绑定留到运行时。然而，上文说到，接口对象是不能创建的，因此，项目一旦要创建 IShape 类型的对象，必然要针对具体的对象 Square 或 Circle 进行创建操作。例如：

IShape shape = new Square();

如果是开发一个图形工具，诸如 Square 和 Circle 之类的对象，其创建工作必然非常频繁。可以设想，在这个项目的各个模块中，将会大量充斥着如上的代码行，导致的结果是各个模块无法与 Square 对象结耦，这意味着，如果我们改变创建的对象为 Circle，就需要修改所有调用 new Square()操作的模块。这既加大了工作量，同时也导致了项目的不可扩展性，以及模块的不可重用性。而对于图形对象的抽象 IShape 来说，也是不必要而失败的。

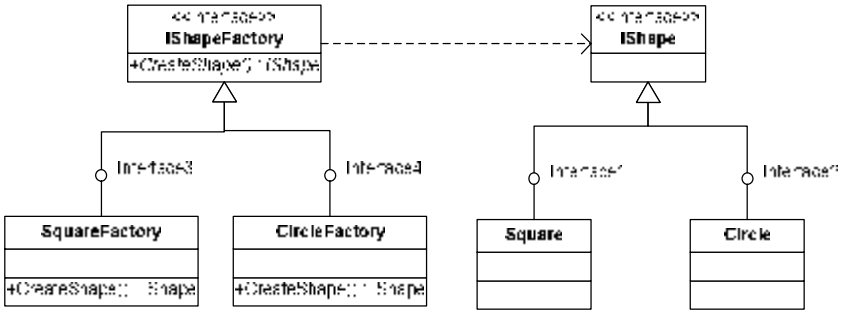
在面向对象的设计中，我们常常将可能变化的操作进行封装，封装的内容可能仅是某种行为，也可能是一种状态，或者是某些职责。而在当前的案例中，我们需要将对象的创建行为进行封装，这就引入了 Factory Method 模式。此时的对象就是 Factory 要生产的产品。既然产品有两种，相对应的工厂也应该是两个，即 SquareFactory 和 CircleFactory。在 Factory Method 模式中，工厂对象的结构应与产品的结构平行，并与之一一对应，所以，对这两个工厂而言，还需要为其抽象出一个共同的工厂接口 IShapeFactory：



代码如下：

```
public interface IShapeFactory
{
    IShape CreateShape();
}
public class SquareFactory:IShapeFactory
{
    public IShape CreateShape()
    {
        return new Square();
    }
}
public class CircleFactory:IShapeFactory
{
    public IShape CreateShape()
    {
        return new Circle();
    }
}
```

通过 Factory Method 模式，我们完成了对象创建的封装，将前面诸如 IShape shape = new Square()的代码全部移到了各自的工厂对象中，并放到 CreateShape()方法中实现。整个结构如下图所示：



请注意 CreateShape()方法的返回类型是 IShape 类型，这就有效地避免了工厂对象与具体产品对象的依赖。

也许会有人认为，虽然通过工厂方法，将创建 IShape 对象的职责转交给工厂对象，然而在工厂类的结构中，仍然存在具体的工厂类对象。如此以来，虽然我们解除了模块与具体 Shape 对象的依赖，却增加了对具体工厂对象的依赖，这会带来何种益处？

让我们从对象创建的频率来分析。对于一个图形工具而言，IShape 对象的创建无疑是频繁的，最大的可能性是在这个项目的各个模块中都可能存在创建 IShape 对象的需要。而工厂对象则不尽然，我们完全可以集中在一个模块中，初始化这个工厂对象，而在需要 IShape 对象的时候，直接调用工厂实例的 CreateShape()就可以达到目的。

举例来说，假设在图形工具中，有三个模块：ModuleA，ModuleB，ModuleC；这三个模块中都需要创建 Square 对象，则按照原来的设计方案，这三个模块都包含这样一行代码：

```
IShape shape = new Square();
```

此时，与 Square 对象有依赖关系的就包括了 ModuleA，ModuleB，ModuleC 三个模块。如果我们需要修改 shape 对象为 Circle 类型，则这个变动无疑会影响到上述的三个模块。现在，我们引入 Factory Method 模式，并增加一个模块名为 ModuleFactory，在这个模块中，我们创建一个工厂对象：

```
IShapeFactory shapeFactory = new SquareFactory();
```

如此以来，原来的三个模块有关 Square 对象的创建，就相应地修改为：

```
IShape shape = shapeFactory.CreateShape();
```

此时，即使需求发生改变，需要对 shape 对象进行修改，那么我们只需要修改 ModuleFactory 模块中的代码：

```
IShapeFactory shapeFactory = new CircleFactory();
```

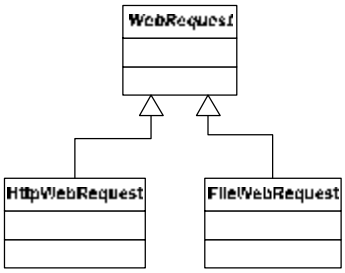
而 ModuleA，ModuleB，ModuleC 三个模块则根本不需要作任何改变。如此的设计改进，虽然在项目中增加了三个工厂对象，并引入了 ModuleFactory，但它却完成了 ModuleA，ModuleB，ModuleC 与具体的 Square 对象的解耦，从而将这三个模块与产品对象的依赖性转嫁到 ModuleFactory 上。如此以来，牵一发而不动其全身，极大地提高了模块的重用性。从上述的分析可知，引入工厂对象并不是简单地为产品建立相应的工厂，而是要注意划分各个模块的职责，将工厂对象的创建放到合适的地方。最佳方案莫过于将创建工厂对象的职责集中起来，放到一个模块中；而不是在需要创建产品时，才创建工厂对象。错误的例子是在创建产品时将工厂对象的创建于产品对象的创建放在一起，并分布在各个模块中：

```
IShapeFactory shapeFactory = new SquareFactory();
IShape shape = shapeFactory.CreateShape();
```

这样引入 Factory Method 模式的方式，无异于画蛇添足了。

二、.Net Framework 中的 Factory Method 模式

Factory Method 模式在项目设计中应用非常广泛，在.Net Framework 中自然也不例外。例如，在.Net 中为处理 Web 请求，在框架类库中提供了基类 WebRequest。它能够通过传入的 Uri 对象，创建基于不同协议的 Web 请求。例如，当 Uri 的前缀为“https://”或“http://”时，则返回 HttpWebRequest 对象，如果是“file://”，则返回 FileWebRequest 对象。HttpWebRequest 和 FileWebRequest 对象是 WebRequest 的派生类。WebRequest 的类结构如图：



如何创建一个 WebRequest 的实例呢？我们可以直接调用该类的静态方法 Create()：

```
WebRequest myRequest = WebRequest.Create("http://www.cnblogs.com");
```

然后我们可以根据该 WebRequest 对象获得 WebResponse：

```
WebResponse myResponse = myRequest.GetResponse();
//……
myResponse.Close();
```

从上面一段代码来看，Create()静态方法似乎是简单工厂模式的一种实现，它可以根据方法传递进来的参数判断 WebRequest 的类型，创建具体的 WebRequest 对象，并返回该对象。那么最简单的实现方法，就是通过 if/else 条件判断参数的类型，以决定创建的对象类型。显然，这并非一个好的方法，它直接导致的就是 WebRequest 的具体子类与其静态方法 Create()直接依赖，一旦增加新的 WebRequest 子类，必然要修改 Create()方法。

既然涉及到对象的创建，最好的方式就是使用 Factory Method 模式。在.Net 中，为创建 WebRequest 对象提供了一个专门的工厂接口 IWebRequestCreate，该接口仅有一个方法，即 Create()方法：

```
public interface IWebRequestCreate
{
    WebRequest Create(Uri uri);
}
```

对应不同的 Web 请求，工厂模式为其提供了不同的具体工厂类，这些类均实现了 IWebRequestCreate 接口。例如 HttpRequestCreator：

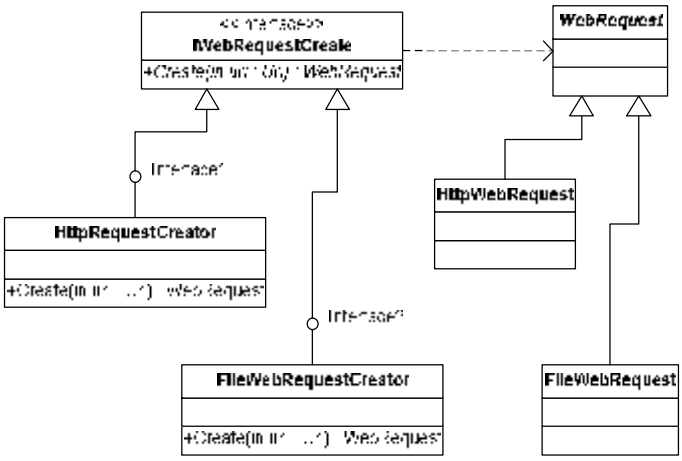
```
internal class HttpRequestCreator : IWebRequestCreate
{
    internal HttpRequestCreator() {}

    public WebRequest Create( Uri Uri )
    {
        return new HttpWebRequest(Uri);
    }
}
```

还有类 FileRequestCreator：

```
internal class FileWebRequestCreator : IWebRequestCreate
{
    internal FileWebRequestCreator() {}

    public WebRequest Create(Uri uri)
    {
        return new FileWebRequest(uri);
    }
}
```

这些类都实现了接口 IWebRequestCreate 的 Create()方法，返回各自对应的 WebRequest 对象，即工厂模式所要生产的产品。

这样，我们就为产品类建立了一个和其完全一一对应的工厂类结构：



请注意工厂类和产品类之间，只存在接口 IWebRequestCreate 和抽象类 WebRequest 之间的依赖关系，这正是 OOP 中面向抽象编程的实质。

三、进一步探索

根据前面的描述，当我们在创建 WebRequest 对象时，需要在系统中预先创建其对应的工厂类对象。如下所示：

```
IWebRequestCreate webRequestCreate = new HttpRequestCreator();
```

然而与一般对象不同的是，WebRequest 的类型是随时可能变化的，这就导致其对应的工厂类型也会经常改变。如果将上面的代码写到一个专门的模块中，并供客户端修改，会缺乏一定的灵活性。并且对于客户而言，WebRequest 对象的创建实在太麻烦了。因为.Net Framework 是一个类库，而类库的设计理念，就是要尽可能让用户更容易更方便地使用类库，至于内部的实现细节，用户是不用理会的。因此，.Net 对 Factory Method 模式进行了一些加工。下面，我将对 WebReuest 的创建过程基于.Net 的实现进行深入分析。

前面提到，WebRequest 是一个抽象类，但它提供了一个静态方法 Create()，能够根据方法中传入的 Uri 地址，创建相对应的 WebRequest 对象。它的实现代码如下：

```
public static WebRequest Create(Uri requestUri)
{
    if (requestUri == null)
    {
        throw new ArgumentNullException("requestUri");
    }
    return Create(requestUri, false);
}
```

该方法实质是调用了 WebRequest 中的私有静态方法 Create()：

```
private static WebRequest Create(Uri requestUri, bool useUriBase)
{
    string LookupUri;
    WebRequestPrefixElement Current = null;//①
    bool Found = false;

    if (!useUriBase)
```

```
        {
            LookupUri = requestUri.AbsoluteUri;
        }
        else
        {
            LookupUri = requestUri.Scheme + ':';
        }

        int LookupLength = LookupUri.Length;
        ArrayList prefixList = PrefixList; //②

        for (int i = 0; i < prefixList.Count; i++)
        {
            Current = (WebRequestPrefixElement)prefixList[i]; //④

            // See if this prefix is short enough.
            if (LookupLength >= Current.Prefix.Length)
            {
                // It is. See if these match.
                if (String.Compare(Current.Prefix,
                    0,
                    LookupUri,
                    0,
                    Current.Prefix.Length,
                    true,
                    CultureInfo.InvariantCulture) == 0)
                {
                    Found = true;
                    break;
                }
            }
        }
        if (Found)
        {
            return Current.Creator.Create(requestUri); //③
        }

        throw new NotSupportedException(SR.GetString(SR.net_unknown_prefix));
    }
```

注意该方法中被我标注的几行代码。第①行代码定义了一个 WebRequestPrefixElement 对象 Current，第②行代码则定义了一个 ArrayList 对象，并将一个已经存在的 ArrayList 对象 PrefixList 赋给它。第③行代码则通过 Current 对象的 Creator 字段来完成创建工作。整体来讲，该方法就是在一个 ArrayList 中，根据参数 Uri 的值进行查找，如果找到，则创建相关的对象并返回，否则抛出异常。

现在我们需要明白两个问题：

1、 WebRequestPrefixElement 类的定义什么？而该类型的 Creator 字段又属于什么类型？

2、 PrefixList 对象存储的内容是什么？

我们首先看看 WebRequestPrefixElement 类的定义：

```
internal class WebRequestPrefixElement
{
    public    string             Prefix;
    public    IWebRequestCreate  Creator;
    public WebRequestPrefixElement(string P, IWebRequestCreate C)
    {
        Prefix = P;
        Creator = C;
    }
}
```

很显然，该类仅仅是提供一个将 Uri 前缀与 IWebRequestCreate 类型关联的一个简单对象而已。而 Creator 字段的类型正是 IWebRequestCreate 类型。Uri 和 IWebRequestCreate 的关联对象，可通过 WebRequestPrefixElement 的构造函数来传入。

那么， Current.Creator.Create(requestUri) 创建对象的实质，就是通过调用 IWebRequestCreate 类型对象的工厂方法，来完成对 WebRequest 对象的创建。然而，究竟调用的是哪一个具体工厂类呢？也就是 Current 字段，其代表的 IWebRequestCreate 对象究竟是什么？

根据第④行代码，Current 的值是从 prefixList 列表中获得的 IWebRequestCreate 对象。而 prefixList 值在该方法中就是 PrefixList 对象。而 PrefixList 其实是 WebRequest 类的一个私有属性：

```
private static ArrayList PrefixList
{
    get
    {
        if (s_PrefixList == null)
        {
            lock (typeof(WebRequest))
            {
                if (s_PrefixList == null)
                {
                    GlobalLog.Print("WebRequest::Initialize(): calling
                        ConfigurationSettings.GetConfig()");
                    ArrayList prefixList = (ArrayList)ConfigurationSettings.GetConfig
                        ("system.net/webRequestModules");

                    if (prefixList == null)
                    {
                        GlobalLog.Print("WebRequest::Initialize():
                                                creating default settings");
```

```
                        HttpRequestCreator Creator = new HttpRequestCreator();

                        // longest prefixes must be the first
                        prefixList = new ArrayList();
                        prefixList.Add(new WebRequestPrefixElement
                                ("https", Creator)); // [0]
                        prefixList.Add(new WebRequestPrefixElement
                                ("http", Creator));  // [1]
                        prefixList.Add(new WebRequestPrefixElement
                                ("file", new FileWebRequestCreator())); // [2]
                    }
                    s_PrefixList = prefixList;
                }
            }
        }
        return s_PrefixList;
    }
    set
    {
        s_PrefixList = value;
    }
}
```

PrefixList 属性的 Get 访问器中，进行了一系列的判断以及初始化工作，其中最重要的工作则是在其内部自动添加了三个元素，均为 WebRequestPrefixElement 对象，而通过该对象，在 prefixList 中建立了 Uri 和 IWebRequestCreate 之间的关系：

```
prefixList.Add(new WebRequestPrefixElement("https", Creator)); // [0]
prefixList.Add(new WebRequestPrefixElement("http", Creator));  // [1]
prefixList.Add(new WebRequestPrefixElement("file", new FileWebRequestCreator())); // [2]
```

前两个对象中，工厂类型均为 HttpWebRequestCreator，而第三个对象则为 FileWebRequestCreator。这正好是.Net 提供的两种继承 WebRequest 的具体工厂类。

调用 WebRequest 的静态方法 Create()时，系统会根据传入的 Uri 对象，在 prefixList 中搜索前缀与 PrefixList 列表中的 uri 匹配的 WebRequestPrefixElement 类型对象（通过 String.Compare()方法）。如果找到，则根据一一映射的关系，去调用对应的工厂类，创建相应的 Web 请求实例，即 Create()方法中的第③行代码。

再回过头来看创建 WebRequest 对象的代码：

```
WebRequest myRequest = WebRequest.Create("http://www.cnblogs.com");
```

根据前面分析的过程，这行代码封装的内部实现应该是如下步骤：

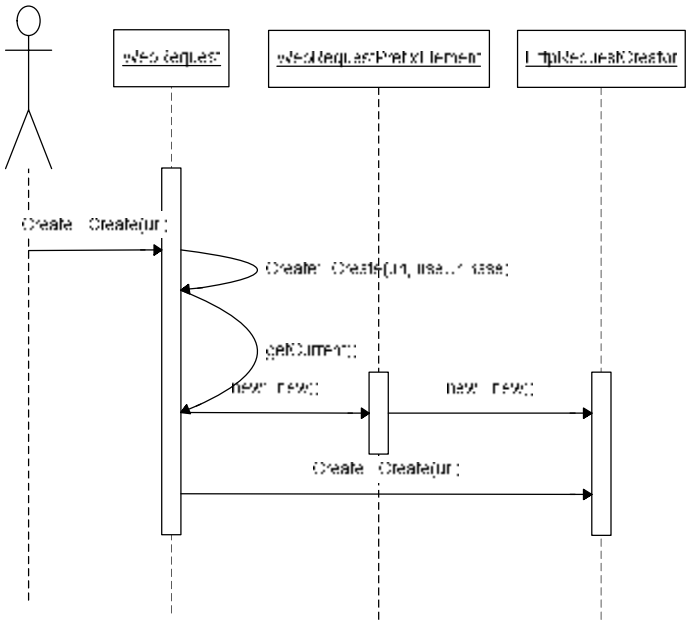
1、将字符串”http://www.cnblogs.com”传递到 WebRequest 类的静态私有方法 Create()中；

2、将 WebRequest 的私有属性 PrefixList 值赋给方法内的局部变量 prefixList 对象。此时会调用 PrefixList 的 Get 访问器。该访问器会初始化 PrefixList 对象，将默认的 Uri 和 IWebRequestCreate 类型的值添加到 PrefixList 中；

3、解析传入的 Uri，得到值”http”，对应的 IWebRequestCreate 对象为 HttpWebRequestCreator 对象；

4、 调用 HttpWebRequestCreator 对象的 Create()方法，创建 HttpWebRequest 对象，并返回。

执行步骤的时序图如下：



现在，考虑扩展的情况。如果此时派生于 WebRequest 类的不仅仅有 HttpWebRequest 和 FileWebRequest 子类，还有新增加的其他子类，例如 FtpRequest 类，它对应的 Uri 为”ftp”。对应的工厂类为 FtpWebRequestCreator。此时应该如何创建？根据前面的分析来看，由于 PrefixList 的 Get 访问器中，并没有添加其他的 WebRequestPrefixElement 对象，如果输入 Uri 前缀”ftp”，Create()方法是无法找到合适的 IWebRequestCreate 工厂对象的。难道，当我们扩展新的 WebRequest 子类时，还是需要修改 WebRequest 类中的 Prefix 属性代码吗？如果真是这样，前面引入的工厂类，以及许多设计就显得拙劣可笑了。

显然，这种可笑的错误，.Net Framework 的设计师们是不可能犯下的。事实上，.Net 在 WebRequest 的创建过程中，引入了 WebRequestPrefixElement 类以及 ArrayList 对象 PrefixList，就已经提示我们，Framework 已经考虑到了扩展的可能。因为，ArrayList 对象是允许我们动态加入对象的。事实正是如此，在 WebRequest 类中，还提供了一个注册 WebRequestPrefixElement 的公共静态方法 RegisterPrefix()：

```
public static bool RegisterPrefix(string prefix, IWebRequestCreate creator)
{
    bool Error = false;
    int i;
    WebRequestPrefixElement Current;

    if (prefix == null)
    {
        throw new ArgumentNullException("prefix");
    }
```

```
        if (creator == null)
        {
            throw new ArgumentNullException("creator");
        }

        lock(typeof(WebRequest))
        {
            ArrayList prefixList = (ArrayList) PrefixList.Clone();
            i = 0;
            while (i < prefixList.Count)
            {
                Current = (WebRequestPrefixElement)prefixList[i];
                if (prefix.Length > Current.Prefix.Length)
                {
                    // It is. Break out of the loop here.
                    break;
                }
                if (prefix.Length == Current.Prefix.Length)
                {
                    // They're the same length.
                    if (String.Compare(Current.Prefix, prefix, true,
CultureInfo.InvariantCulture) == 0)
                    {
                        // ...and the strings are identical. This is an error.
                        Error = true;
                        break;
                    }
                }
                i++;
            }

            if (!Error)
            {
                prefixList.Insert(i,new WebRequestPrefixElement(prefix, creator));
                PrefixList = prefixList;
            }
        }
        return !Error;
    }
```

只要在已有的 PrefixList 中没有找到参数中传递进来的 prefix 值，就将新的 prefix 值和 IWebRequestCreate 对象插入到 PrefixList 中。通过该方法，就可以动态添加新的工厂类了。例如：

```
WebRequest.RegisterPrefix("ftp” ",new FtpWebRequestCreator());
WebRequest ftpRequest = WebRequest.Create("ftp://www.cnblogs.com");
```

通过这种实现方式，就可以解开具体工厂类、具体产品类与 WebRequest 静态方法 Create() 之间的耦合。

在.Net Framework 中，关于 WebRequest 类对象的创建，其实现方式虽然看似复杂，但其本质仍然属于 Factory Method 模式。但为了类库的更易于使用，并考虑到通用性和扩展性，又引入了类似于映射的 WebRequestPrefixElement 类以及 ArrayList 对象，同时又将具体工厂类对象设定为 internal，并包装到抽象产品基类 WebRequest 的静态方法中。这种设计方法是我们在应用 Factory Method 模式设计自己的类库时，值得借鉴的。

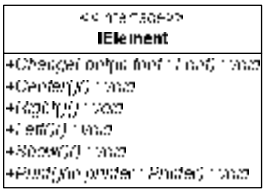
Composite 模式

——《.Net 中的设计模式》系列

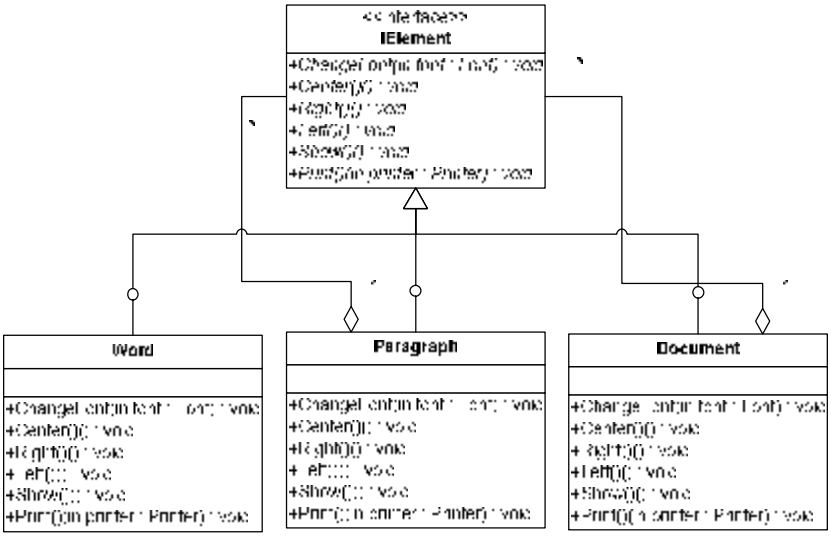
一、模式概述

描述 Composite 模式的最佳方式莫过于树形图。从抽象类或接口为根节点开始，然后生枝发芽，以形成树枝节点和叶结点。因此，Composite 模式通常用来描述部分与整体之间的关系，而通过根节点对该结构的抽象，使得客户端可以将单元素节点与复合元素节点作为相同的对象来看待。

由于 Composite 模式模糊了单元素和复合元素的区别，就使得我们为这些元素提供相关的操作时，可以有一个统一的接口。例如，我们要编写一个字处理软件，该软件能够处理文字，对文章进行排版、预览、打印等功能。那么，这个工具需要处理的对象，就应该包括单个的文字、以及由文字组成的段落，乃至整篇文档。这些对象从软件处理的角度来看，对外的接口应该是一致的，例如改变文字的字体，改变文字的位置使其居中或者右对齐，也可以显示对象的内容，或者打印。而从内部实现来看，我们对段落或者文档进行操作，实质上也是对文字进行操作。从结构来看，段落包含了文字，文档又包含了段落，是一个典型的树形结构。而其根节点正是我们可以抽象出来暴露在外的统一接口，例如接口 IElement：



既然文字、段落、文档都具有这些操作，因此它们都可以实现 IElement 接口：

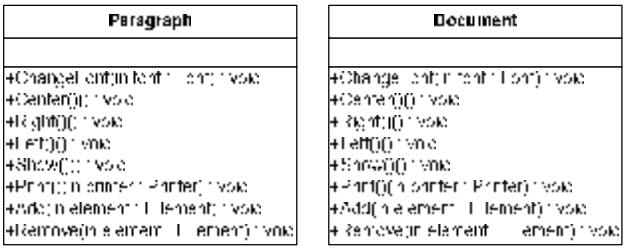


从上图可以看到，对象 Word、Paragraph、Document 均实现了 IElement 接口，但 Paragraph 和 Document 与 Word 对象不同的是，这两者处除了实现了 IElement 接口，它们还与 IElement 接口对象之间具有聚合的关系，且是一对多。也就是说 Paragraph 与 Document 对象内可以

包含 0 个到多个 IElement 对象，这也是与前面对字处理软件分析获得的结果是一致的。从整个结构来看，完全符合树形结构的各个要素，接口 IElement 是根节点，而 Paragraph 和 Document 类为枝节点，Word 对象为叶节点。既然作为枝节点，它就具有带叶节点的能力，从上图的聚合关系中我们体现出了这一点。也就是说，Paragraph 和 Document 类除了具有排版、打印方面的职责外，还能够添加、删除叶节点的操作。那么这些操作应该放在哪里呢？管理对子节点的管理，Composite 模式提供了两种方式：一个是透明方式，也就是说在根节点中声明所有用来管理子元素的方法，包括 Add()、Remove()等方法。这样一来，实现根节点接口的子节点同时也具备了管理子元素的能力。这种实现策略，最大的好处就是完全消除了叶节点和枝节点对象在抽象层次的区别，它们具备完全一致的接口。而缺点则是不够安全。由于叶节点本身不具备管理子元素的能力，因此提供的 Add()、Remove()方法在实现层次是无意义的。但客户端调用时，却看不到这一点，从而导致在运行期间有出错的可能。

另一种策略则是安全方式。与透明方式刚好相反，它只在枝节点对象里声明管理子元素的方法，由于叶节点不具备这些方法，当客户端在操作叶节点时，就不会出现前一种方式的安全错误。然而，这种实现方式，却导致了叶节点和枝节点接口的不完全一致，这给客户端操作时带来了不便。

这两种方式各有优缺点，我们在实现时，应根据具体的情况，作出更加合理的抉择。在字处理软件一例中，我选择了安全方式来实现，因为对于客户端而言，在调用 IElement 接口时，通常是将其视为可被排版、打印等操作的对象。至于为 Paragraph 和 Document 对象添加、删除子对象，往往是一种初始化的行为，完全可以放到一个单独的模块中。根据单一职责原则（SRP），我们没有必要让 IElement 接口负累太重。所以，我们需要对上图作稍许的修改，在 Paragraph 和 Document 对象中增加 Add()和 Remove()方法：



以下是 IElement 对象结构的实现代码：

```
public interface IElement
{
    void ChangeFont(Font font);
    void Show();
    //其他方法略;
}
public class Word
{
    public void ChangeFont(Font font)
    {
        this.font = font;
    }
    public void Show()
    {
```

```
            Console.WriteLine(this.ToString());
        }
        //其他方法略;
    }
    public class Paragraph
    {
        private ArrayList elements = new ArrayList();
        public void Add(IElement element)
        {
            elements.Add(element);
        }
        public void Remove(IElement element)
        {
            elements.Remove(element);
        }
        public void ChangeFont(Font font)
        {
            foreach (IElement element in elements)
            {
                element.ChangeFont(font);
            }
        }
        public void Show()
        {
            foreach (IElement element in elements)
            {
                element.Show(font);
            }
        }
        //其他方法略;
    }
    //Document 类略;
```

实际上，我们在为叶节点实现 Add()，Remove()方法时，还需要考虑一些异常情况。例如在 Paragraph 类中，添加的子元素就不能是 Document 对象和 Paragraph 对象。所以在添加 IElement 对象时，还需要做一些条件判断，以确定添加行为是否正确，如果错误，应抛出异常。

采用 Composite 模式，我们将 Word、Paragraph、Document 抽象为 IElement 接口。虽然各自内部的实现并不相同，枝节点和叶节点的实质也不一样，但对于调用者而言，是没有区别的。例如在类 WordProcessor 中，包含一个 GetSelectedElement()静态方法，它能够获得当前选择的对象：

```
    public class WordProcessor
    {
        public static IElement GetSelectedElement(){……}
    }
```

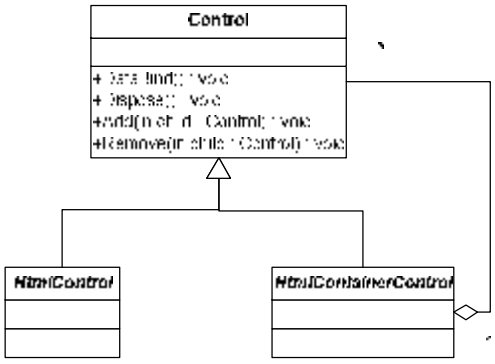
对于字处理软件的 UI 来说，如果要改变选中对象的字体，则可以在命令按钮 cmdChangeFont 的 Click 事件中写下如下代码：

```
public void cmdChangeFont_Click(object sender, EventArgs e)
{
    WordProcessor.GetSelectedElement().ChangeFont(currentFont);
}
```

不管当前选中的对象是文字、段落还是整篇文档，对于 UI 而言，操作都是完全一致的，根本不需要去判断对象的类别。因此，如果在 Business Layer 的类库设计时，采用 Composite 模式，将极大地简化 UI 表示层的开发工作。此外，应用该模式也较好的支持项目的可扩展性。例如，我们为 IElement 接口增加了 Sentence 类，对于前面的例子而言，只需要修改 GetSelectedElement()方法，而 cmdChangeFont 命令按钮的 Click 事件以及 Business Layer 类库原有的设计，都不需要做任何改变。这也符合 OO 的开放-封闭原则（OCP），即对于扩展是开放的(Open for extension)，对于更改则是封闭的（Closed for modification）。

二、.Net Framework 中的 Composite 模式

在.Net 中，最能体现 Composite 模式的莫过于 Windows 或 Web 的控件。在这些控件中，有的包含子控件，有的则不包含且不能包含子控件，这正好符合叶节点和枝节点的含义。所有 Web 控件的基类为 System.Web.UI.Contril 类（如果是 Windows 控件，则基类为 System.Windows.Forms.Control 类）。其子类包含有 HtmlControl、HtmlContainerControl 等。按照 Composite 模式的结构，枝节点和叶节点属于根节点的不同分支，同时枝节点与根节点之间应具备一个聚合关系，可以通过 Add()、Remove()方法添加和移除其子节点。设定 HtmlControl 为叶节点，而 HtmlContaiinerControl 为枝节点，那么采用透明方式的设计方法，在.Net 中控件类的结构，就应该如下图所示：



虽然根据透明方式的 Composite 模式，HtmlControl 类与其父类 Control 之间也应具备一个聚合关系，但实质上该类并不具备管理子控件的职责，因此我在类图中忽略了这个关系。此时，HtmlControl 类中的 Add()、Remove()方法，应该为空，或者抛出一个客户端能够捕获的异常。

然而，从具体实现来考虑，由于 HtmlControl 类和 HtmlContainerControl 类在实现细节层次，区别仅在于前者不支持子控件，但从控件本身的功能来看，很多行为是相同或者相近的。例如 HtmlControl 类的 Render()方法，调用了方法 RenderBeginTag()方法：

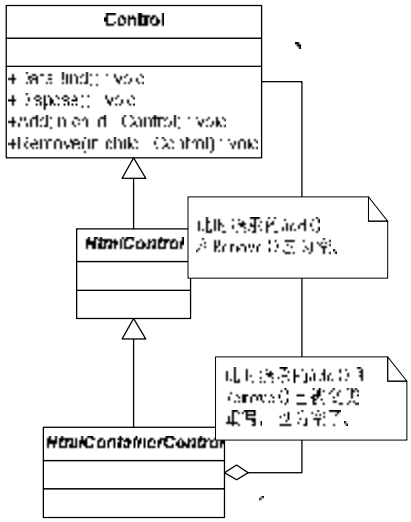
```
protected override void Render(HtmlTextWriter writer)
{
    this.RenderBeginTag(writer);
```

```
    }
    protected virtual void RenderBeginTag(HtmlTextWriter writer)
    {
        writer.WriteBeginTag(this.TagName);
        this.RenderAttributes(writer);
        writer.Write('>');
    }
```

而 HtmlContainerControl 类也具有 Render()方法，在这个方法中也调用了 RenderBeginTag()方法，且 RenderBeginTag 方法的实现和前者完全一致：

```
    protected override void Render(HtmlTextWriter writer)
    {
        this.RenderBeginTag(writer);
        this.RenderChildren(writer);
        this.RenderEndTag(writer);
    }
```

按照上面的结构，由于 HtmlControl 和 HtmlContainerControl 之间并无继承关系，这就要求两个类中，都要重复实现 RenderBeginTag()方法，从而导致产生重复代码。根据 OO 的特点，解决的办法，就是让 HtmlContainerControl 继承自 HtmlControl 类（因为 HtmlContainerControl 的接口比 HtmlControl 宽，所以只能令 HtmlContainerControl 作为子类），并让 RenderBeginTag()方法成为 HtmlControl 类的 protected 方法，子类 HtmlContainerControl 可以直接调用这个方法。然而与之矛盾的是，HtmlContainerControl 却是一个可以包含子控件的枝节点，而 HtmlControl 则是不能包含子控件的叶节点，那么这样的继承关系还成立吗？



HtmlControl 类对 Add()方法和 Remove()方法的重写后，这两个方法内容为空。由于 HtmlContainerControl 类继承 HtmlControl 类，但我们又要求它的 Add()和 Remove()方法和 Control 类保持一致，而父类 HtmlControl 已经重写这两个方法，此时是无法直接继承来自父类的方法的。以上是采用透明方式的设计。

如果采用安全方式，仍然有问题。虽然在 HtmlControl 类中不再有 Add()和 Remove()方法，但由于 Control 类和 HtmlContainerControl 类都允许添加子控件，它们包含的 Add()、Remove()方法，只能分别实现。这样的设计必然会导致重复代码。这也是与我们的期望不符的。

那么在.Net 中，Control 类究竟是怎样实现的呢？下面，我将根据.Net 实现 Control 控件的源代码，来分析 Control 控件的真实结构，以及其具体的实现细节。

三、深入分析.Net 中的 Composite 模式

首先，我们来剖析 Web 控件的基类 Control 类的内部实现：

```
public class Control : IComponent, IDisposable, IParserAccessor, IDataBindingsAccessor
{
    // Events;略
    // Methods
    public Control()
    {
        if (this is INamingContainer)
        {
            this.flags[0x80] = true;
        }
    }
    public virtual bool HasControls()
    {
        if (this._controls != null)
        {
            return (this._controls.Count > 0);
        }
        return false;
    }
    public virtual void DataBind()
    {
        this.OnDataBinding(EventArgs.Empty);
        if (this._controls != null)
        {
            string text1 =
this._controls.SetCollectionReadOnly("Parent_collections_readonly");
            int num1 = this._controls.Count;
            for (int num2 = 0; num2 < num1; num2++)
            {
                this._controls[num2].DataBind();
            }
            this._controls.SetCollectionReadOnly(text1);
        }
    }
    protected virtual void Render(HtmlTextWriter writer)
    {
        this.RenderChildren(writer);
    }
```

```
        protected virtual ControlCollection CreateControlCollection()
        {
            return new ControlCollection(this);
        }

        // Properties
        public virtual ControlCollection Controls
        {
            get
            {
                if (this._controls == null)
                {
                    this._controls = this.CreateControlCollection();
                }
                return this._controls;
            }
        }
        // Fields
        private ControlCollection _controls;
    }
```

Control 基类中的属性和方法很多，为清晰起见，我只保留了几个与模式有关的关键方法与属性。在上述的源代码中，我们需要注意几点：

1、Control 类不是抽象类，而是具体类。这是因为在设计时，我们可能会创建 Control 类型的实例。根据这一点来看，这并不符合 OOP 的要求。一般而言，作为抽象出来的基类，必须定义为接口或抽象类。不过在实际的设计中，也不应拘泥于这些条条框框，而应审时度势，根据实际的情况来抉择最佳的设计方案。

2、公共属性 Controls 为 ControlCollection 类型，且该属性为 virtual 属性。也就是说，这个属性可以被它的子类 override。同时，该属性为只读属性，在其 get 访问器中，调用了方法 CreateControlCollection()；这个方法为 protected 虚方法，默认的实现是返回一个 ControlCollection 实例。

3、方法 HasControls()，功能为判断 Control 对象是否有子控件。它判断的依据是根据私有字段_controls（即公共属性 Controls）的 Count 值。但是需要注意的是，通过 HasControls()方法的返回值，并不能决定对象本身属于叶节点，还是枝节点。因为即使是枝节点其内部仍然可以不包含任何子对象。

4、方法 DataBind()的实现中，首先调用了自身的 OnDataBinding()方法，然后又遍历了 Controls 中的所有控件，并调用其 DataBind()方法。该方法属于控件的共有行为，从这里可以看出不管是作为叶节点的控件，还是作为枝节点的控件，它们都实现统一的接口。对于客户端调用而言，枝节点和叶节点是没有区别的。

5、Control 类的完整源代码中，并不存在 Add()、Remove()等类似的方法，以提供添加和移除子控件的功能。事实上，继承 Control 类的所有子类均不存在 Add()、Remove()等方法。

显然，在 Control 类的定义和实现中，值得我们重视的是公共属性 Controls 的类型 ControlCollection。顾名思义，该类必然是一个集合类型。是否有关子控件的操作，都是在 ControlCollection 类型中实现呢？我们来分析一下 ControlCollection 的代码：

```
    public class ControlCollection : ICollection, IEnumerable
    {
        // Methods
        public ControlCollection(Control owner)
        {
            this._readOnlyErrorMsg = null;
            if (owner == null)
            {
                throw new ArgumentNullException("owner");
            }
            this._owner = owner;
        }
        public virtual void Add(Control child)
        {
            if (child == null)
            {
                throw new ArgumentNullException("child");
            }
            if (this._readOnlyErrorMsg != null)
            {
                throw new
HttpException(HttpRuntime.FormatResourceString(this._readOnlyErrorMsg));
            }
            if (this._controls == null)
            {
                this._controls = new Control[5];
            }
            else if (this._size >= this._controls.Length)
            {
                Control[] controlArray1 = new Control[this._controls.Length * 4];
                Array.Copy(this._controls, controlArray1, this._controls.Length);
                this._controls = controlArray1;
            }
            int num1 = this._size;
            this._controls[num1] = child;
            this._size++;
            this._version++;
            this._owner.AddedControl(child, num1);
        }
        public virtual void Remove(Control value)
        {
            int num1 = this.IndexOf(value);
            if (num1 >= 0)
            {
```

```
                this.RemoveAt(num1);
            }
        }
        // Indexer
        public virtual Control this[int index]
        {
            get
            {
                if ((index < 0) || (index >= this._size))
                {
                    throw new ArgumentOutOfRangeException("index");
                }
                return this._controls[index];
            }
        }
        // Properties
        public int Count
        {
            get
            {
                return this._size;
            }
        }
        protected Control Owner
        {
            get
            {
                return this._owner;
            }
        }
        protected Control Owner { get; }

        // Fields
        private Control[] _controls;
        private const int _defaultCapacity = 5;
        private const int _growthFactor = 4;
        private Control _owner;
    }
```

一目了然，正是 ControlCollection 的 Add()、Remove()方法完成了对子控件的添加和删除。例如：

```
Control parent = new Control();
Control child = new Child();
//添加子控件child;
parent.Controls.Add(child);
```

```
//移除子控件child;
parent.Controls.Remove(child);
```

为什么要专门提供 ControlCollection 类型来管理控件的子控件呢？首先，作为类库使用者，自然希望各种类型的控件具有统一的接口，尤其是自定义控件的时候，不希望自己重复定义管理子控件的操作；那么采用透明方式自然是最佳方案。然而，在使用控件的时候，安全也是需要重点考虑的，如果不考虑子控件管理的合法性，一旦使用错误，会导致整个应用程序出现致命错误。从这样的角度考虑，似乎又应采用安全方式。这里就存在一个抉择。故而，.Net 在实现 Control 类库时，利用了职责分离的原则，将控件对象管理子控件的属性与行为和控件本身分离，并交由单独的 ControlCollection 类负责。同时采用聚合而非继承的方式，以一个公共属性 Controls，存在于 Control 类中。这种方式，集保留了透明方式和安全方式的优势，又摒弃了这两种方式固有的缺陷，因此我名其为“复合方式”。

“复合方式”的设计，其对安全的保障，不仅仅是去除了 Control 类关于子控件管理的统一接口，同时还通过异常管理的方式，在 ControlCollection 类的子类中实现：

```
public class EmptyControlCollection : ControlCollection
{
    // Methods
    public EmptyControlCollection(Control owner) : base(owner)
    {
    }
    public override void Add(Control child)
    {
        this.ThrowNotSupportedException();
    }
    private void ThrowNotSupportedException()
    {
        throw new HttpException(HttpRuntime.FormatResourceString
            ("Control_does_not_allow_children", base.Owner.GetType().ToString()));
    }
}
```

EmptyControlCollection 继承了 ControlCollection 类，并重写了 Add()等添加子控件的方法，使其抛出一个异常。注意，它并没有重写父类的 Remove()方法，这是因为 ControlCollection 类在实现 Remove()方法时，对集合内的数据进行了非空判断。而在 EmptyControlCollection 类中，是不可能添加子控件的，直接调用父类的 Remove()方法，是不会出现错误的。

既然管理子控件的职责由 ControlCollection 类型负责，且 Control 类中的公共属性 Controls 即为 ControlCollection 类型。所以，对于控件而言，如果是树形结构中的叶节点，它不能包含子控件，它的 Controls 属性就应为 EmptyControlCollection 类型，假如用户调用了 Controls 的 Add()方法，就会抛出异常。如果控件是树形结构中的枝节点，它支持子控件，那么 Controls 属性就是 ControlCollection 类型。究竟是枝节点还是叶节点，决定权在于公共属性 Controls：

```
public virtual ControlCollection Controls
{
    get
    {
        if (this._controls == null)
```

```
            {
                this._controls = this.CreateControlCollection();
            }
            return this._controls;
        }
    }
```

在属性的 get 访问器中，调用了 protected 方法 CreateControlCollection()，它创建并返回了一个 ControlCollection 实例：

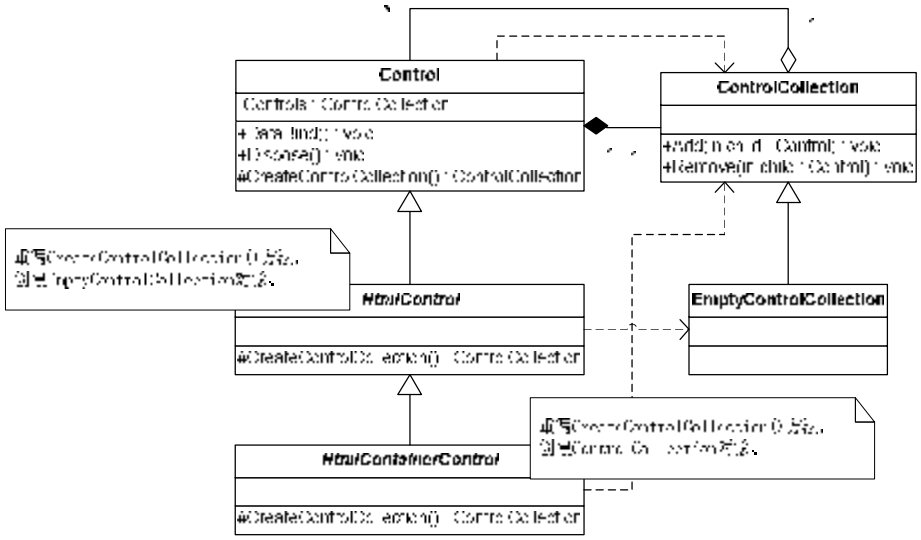
```
protected virtual ControlCollection CreateControlCollection()
{
    return new ControlCollection(this);
}
```

很明显，在 Control 基类实现 Controls 属性时，采用了 Template Method 模式,它推迟了 ControlCollection 的创建，将决定权交给了 CreateControlCollection()方法。

如果我们需要定义一个控件，要求它不能管理子控件，就重写 CreateControlCollection()方法，返回 EmptyControlCollection 对象：

```
protected override ControlCollection CreateControlCollection()
{
    return new EmptyControlCollection(this);
}
```

现在再回过头来看 HtmlControl 和 HtmlContainerControl 类。根据前面的分析，我们要求 HtmlContainerControl 继承 HtmlControl 类，同时，HtmlContainerControl 应为枝节点，能够管理子控件；HtmlControl 则为叶节点，不支持子控件。通过引入 ControlCollection 类和其子类 EmptyControlCollection，以及 Template Method 模式后，这些类之间的关系与结构如下所示：



HtmlContainerControl 继承了 HtmlControl 类，这两个类都重写了自己父类的 protected 方法 CreateControlCollection()。HtmlControl 类，该方法返回 EmptyControlCollection 对象，使其成为了不包含子控件的叶节点；HtmlContainerControl 类中，该方法则返回 ControlCollection 对象，从而被赋予了管理子控件的能力，成为了枝节点：

```
public abstract class HtmlControl : Control, IAttributeAccessor
{
    // Methods
    protected override ControlCollection CreateControlCollection()
    {
        return new EmptyControlCollection(this);
    }
}

public abstract class HtmlContainerControl : HtmlControl
{
    // Methods
    protected override ControlCollection CreateControlCollection()
    {
        return new ControlCollection(this);
    }
}
```

HtmlControl 和 HtmlContainerControl 类均为抽象类。要定义它们的子类，如果不重写其父类的 CreateControlCollection()方法，那么它们的 Controls 属性，就与父类完全一致。例如 HtmlImage 控件继承自 HtmlControl 类，该控件不能添加子控件；而 HtmlForm 控件则继承自 HtmlContainerControl 类，显然，HtmlForm 控件是支持添加子控件的操作的。

.Net 的控件设计采用 Composite 模式的“复合方式”，较好地将控件的透明性与安全性结合起来，它的特点是：

1、在统一接口中消除了 Add()、Remove()等子控件的管理方法，而由 ControlCollection 类实现，同时通过 EmptyControlCollection 类保障了控件进一步的安全；

2、控件能否管理子控件，不由继承的层次决定；而是通过重写 CreateControlCollection()方法，由 Controls 属性的真正类型来决定。

如此一来，要定义自己的控件就更加容易。我们可以任意地扩展自己的控件类。不管继承自 Control，还是 HtmlControl 或 HtmlContainerControl，都可以轻松地定义出具有枝节点或叶节点属性的新控件。如果有新的需求要求改变管理子控件的方式，我们还可以定义继承自 ControlCollection 的类，并在控件类的方法 CreateControlCollection()中创建并返回它的实例。

Iterator 模式

——《.Net 中的设计模式》系列

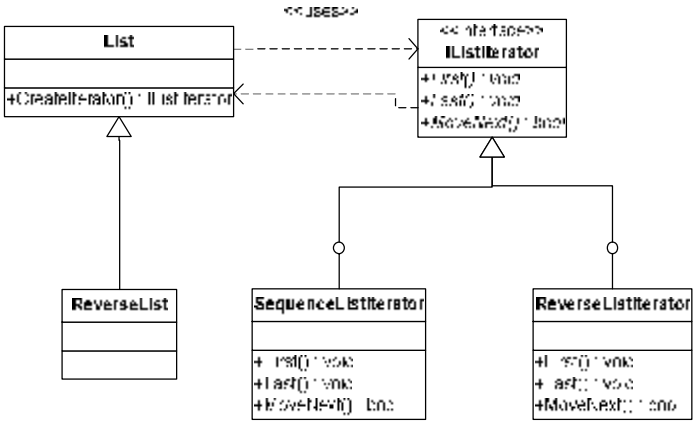
一、模式概述

在面向对象设计时，我们常常需要辨认对象的职责。理想的状态下，我们希望自己建立的对象只具有一个职责。对象的责任越少，则该对象的稳定性就越好，受到的约束也就越少。职责分离，可以最大限度地减少彼此之间的耦合程度，从而建立一个松散耦合的对象网络。职责分离的要点是对被分离的职责进行封装，并以抽象的方式建立起彼此之间的关系。在C#中，我们往往将这些可能变化的对象抽象为接口和抽象类，从而将原来的具体依赖改变为抽象依赖。对象不再受制于具体的实现细节，这就代表他们是可被替换的。

要在设计上做到这一点，首先就要学会分辨职责，学会分辨哪些职责是对象中可变的。以集合对象为例，集合是一个管理和组织数据对象的数据结构。这就表明集合首先应具备一个基本属性，就是集合能够存储数据。这其中包含存储数据的类型、存储空间的大小、存储空间的分配、以及存储的方式和顺序。不具备这些特点，则该对象就不成其为集合对象。也就是说，上述这些属性是集合对象与身俱来的，是其密不可分的职责。然而，集合对象除了能够存储数据外，还必须提供访问其内部数据的行为方式，这是一种遍历机制。同时这种遍历方式，或会根据不同的情形提供不同的实现，如顺序遍历，逆序遍历，或是二叉树结构的中序、前序、后序遍历。

现在我们已经分辨出集合对象拥有的两个职责：一是存储内部数据；二是遍历内部数据。从依赖性来看，前者为集合对象的根本属性，属于一生俱生，一亡俱亡的关系；而后者既是可变化的，又是可分离的。因此，我们将遍历行为分离出来，抽象为一个迭代器，专门提供遍历集合内部数据对象的行为。这就是 Iterator 模式的本质。

如一个列表对象 List，它提供了遍历列表各元素的能力，这种遍历的行为可能包含两种：顺序和逆序遍历。对于一般的 List 对象而言，采用顺序遍历的方式；而对于特定的 List 对象，如 ReverseList，则按照逆序遍历的方式访问其内部数据。如果不将存储数据和访问数据的职责分离，为实现 ReverseList 类，就需要重写其父类 List 中所有用于遍历数据的方法。现在我们采用 Iterator 模式，在 List 对象中分离出迭代器 IListIterator，那就只需要为这个继承自 List 对象的 ReverseList 对象，提供逆序迭代器 ReverseListIterator 就可以了，如下面的类图所示：



代码如下：

```
public interface IListIterator
{
     void First();
     void Last();
     bool MoveNext();
}
public class SequenceListIterator:IListIterator
{
     private List list = null;
     private int index;
     public SequenceListIterator(List list)
     {
          index = -1;
          this.list = list;
     }
     public void First()
     {
          index = 0;
     }
     public void Last()
     {
          index = list.Count;
     }
     public bool MoveNext()
     {
          index ++;
          return list.Count > index;
     }
}
public class ReverseListIterator:IListIterator
{
```

```
        private List list = null;
        private int index;
        public ReverseListIterator(List list)
        {
            index = list.Count;
            this.list = list;
        }
        public void First()
        {
            index = list.Count - 1;
        }
        public void Last()
        {
            index = 0;
        }
        public bool MoveNext()
        {
            index --;
            return index >= 0;
        }

    }
    public class List
    {
        public virtual IListIterator CreateIterator()
        {
            return new SequenceListIterator(this);
        }
    }
    public class ReverseList:List
    {
        public override IListIterator CreateIterator()
        {
            return new ReverseListIterator(this);
        }
    }
```

我们看到，List 类通过方法 CreateIterator()，创建了 SequenceListIterator 对象，使得 List 集合对象能够用顺序迭代的方式遍历内部元素。而要使 ReverseList 采用逆序的方式遍历其内部元素，则只需重写父类 List 的 CreateIterator()方法，通过创建 ReverseListIterator 对象，来建立集合与具体迭代器之间的关系。

二、.Net 中的 Iterator 模式

在.Net 中，IEnumerator 接口就扮演了 Iterator 模式中迭代器的角色。IEnumerator 的定义

如下：

```
public interface IEnumerator
{
    bool MoveNext();
    Object Current {get; }
    void Reset();
}
```

该接口提供了遍历集合元素的方法，其中主要的方法是 MoveNext()。它将集合中的元素下标移到下一个元素。如果集合中没有元素，或已经移到了最后一个，则返回 false。能够提供元素遍历的集合对象，在.Net 中都实现了 IEnumerator 接口。

我们在使用.Net 集合对象时，会发现一个方法 GetEnumerator()方法，它类似前面提到的 List 对象的 CreateIterator()方法。该方法返回的类型是 IEnumerator，其内部则是创建并返回一个具体的迭代器对象。正是通过这个方法，建立了具体的集合对象与迭代器之间的关系。与通常的 Iterator 模式实现不同，.Net Framework 将 GetEnumerator()方法单独抽象出来，定义了接口 IEnumerable：

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

IEnumerable 接口就像迭代功能的标识，如果集合对象需要具备迭代遍历的功能，就必须实现该接口，并在具体实现中，创建与自身有关系的具体迭代器对象。而要获得该集合对象对应的迭代器，就可以通过该方法，如：

```
ArrayList al = new ArrayList();
IEnumerator iterator = al.GetEnumerator();
```

下面，我就以 ArrayList 对象为例，讨论一下.Net 中 Iterator 模式的实现方式。首先，我们来看看 ArrayList 类的定义：

```
public class ArrayList : IList, ICloneable
```

它分别实现了 IList 和 ICloneable 接口。其中，ICloneable 接口提供了 Clone 方法，与本文讨论的内容无关，略过不提。IList 接口则提供了和链表相关的操作，如 Add，Remove 等。这也是 ArrayList 和 Array 的不同之处。而 IList 接口又实现了 ICollection 接口。

```
public interface IList : ICollection
```

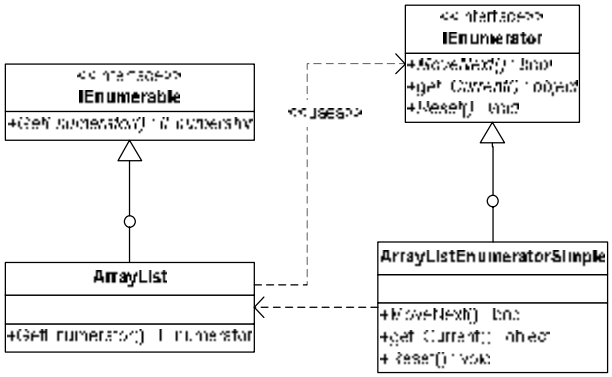
ICollection 接口是所有集合类型的公共接口，它提供了获得集合长度和同步处理的一些方法，不过在这里，我们需要注意的是它实现了 IEnumerable 接口：

```
public interface ICollection : IEnumerable
```

追本溯源，ArrayList 类型间接地实现了 IEnumerable 接口。在 ArrayList 中，IEnumerable 接口的 GetEnumerator()方法实现代码如下：

```
public virtual IEnumerator GetEnumerator()
{
    return new ArrayListEnumeratorSimple(this);
}
```

GetEnumerator()方法是一个虚方法，这说明，我们可以自定义一个集合类型继承 ArrayList，重写这个方法，创建和返回不同的 IEnumerator 对象，从而实现不同的遍历方式。为了实现 ArrayList 的遍历功能，采用的 Iterator 模式结构如下图所示：



其中，类 ArrayListEnumeratorSimple 的实现如下所示：

```
[Serializable]
private class ArrayListEnumeratorSimple : IEnumerator, ICloneable
{
    // Methods
    internal ArrayListEnumeratorSimple(ArrayList list)
    {
        this.list = list;
        this.index = -1;
        this.version = list._version;
        this.currentElement = list;
    }
    public object Clone()
    {//实现略}
        public virtual bool MoveNext()
        {
            if (this.version != this.list._version)
            {
                throw new InvalidOperationException
        (Environment.GetResourceString("InvalidOperation_EnumFailedVersion"));
            }
            if (this.index < (this.list.Count - 1))
            {
                this.index++;
                this.currentElement = this.list[this.index];
                return true;
            }
            this.currentElement = this.list;
            this.index = this.list.Count;
            return false;
        }

    public virtual void Reset()
    {
```

```
            if (this.version != this.list._version)
            {
                throw new InvalidOperationException
        (Environment.GetResourceString("InvalidOperation_EnumFailedVersion"));
            }
            this.currentElement = this.list;
            this.index = -1;
        }
        // Properties
        public virtual object Current
        {
            get
            {
                object obj1 = this.currentElement;
                if (obj1 != this.list)
                {
                    return obj1;
                }
                if (this.index == -1)
                {
                    throw new InvalidOperationException
            (Environment.GetResourceString("InvalidOperation_EnumNotStarted"));
                }
                throw new InvalidOperationException
            (Environment.GetResourceString("InvalidOperation_EnumEnded"));
            }
        }
        // Fields
        private object currentElement;
        private int index;
        private ArrayList list;
        private int version;
    }
```

ArrayListEnumeratorSimple 实现了 IEnumerator 接口，实现了 MoveNext()、Current、Reset()方法或属性。该类是一个私有类型，其构造函数则被 internal 修饰符限制。在自定义的构造函数中，传入的参数类型是 ArrayList。正是通过构造函数传递需要遍历的 ArrayList 对象，来完成 MoveNext()、Current、Reset()等操作。

下面，我们来看看如何通过 IEnumerator 来实现对 ArrayList 的遍历操作。

```
using System;
using System.Collections;
using NUnit.Framework;

[TestFixture]
public class Tester
```

```
{
    [Test]
    public void TestArrayList()
    {
        ArrayList al = new ArrayList();
        al.Add(5);
        al.Add("Test");
        IEnumerator e = al.GetEnumerator();
        e.MoveNext();
        Assert.AreEqual(5,e.Current);
        e.MoveNext();
        Assert.AreEqual("Test",e.Current);
    }
}
```

而要遍历 ArrayList 内部所有元素，方法也很简单：

```
while (e.MoveNext())
{
    Console.WriteLine(e.Current.ToString());
}
```

事实上，为了用户更方便地遍历集合对象的所有元素，在 C#中提供了 foreach 语句。该语句的实质正是通过 IEnumerator 的 MoveNext()方法来完成遍历的。下面的语句与刚才那段代码是等价的：

```
foreach (object o in al)
{
    Console.WriteLine(o.ToString());
}
```

为了验证 foreach 语句与迭代器的关系，我们来自定义一个 ReverseArrayList 类。要求遍历这个类的内部元素时，访问顺序是逆序的。要定义这样的一个类，很简单，只需要继承 ArrayList 类，并重写 GetEnumerator()方法既可。

```
public class ReverseArrayList:ArrayList
{
    public override IEnumerator GetEnumerator()
    {
        return new ReverseArrayListEnumerator(this);
    }
}
```

其中，类 ReverseArrayListEnumerator，实现了接口 IEnumerator，它提供了逆序遍历的迭代器：

```
public class ReverseArrayListEnumerator:IEnumerator
{
    public ReverseArrayListEnumerator(ArrayList list)
    {
        this.list = list;
        this.index = list.Count;
```

```
            this.currentElement = list;
        }

        #region IEnumerator Members
        public virtual void Reset()
        {
            this.currentElement = this.list;
            this.index = this.list.Count;
        }
        public virtual object Current
        {
            get
            {
                object obj1 = this.currentElement;
                if (obj1 != this.list)
                {
                    return obj1;
                }
                if (this.index == -1)
                {
                    throw new InvalidOperationException("Out of the Collection");
                }
                throw new InvalidOperationException("Out of the Collection");
            }
        }
        public virtual bool MoveNext()
        {
            if (this.index > 0)
            {
                this.index--;
                this.currentElement = this.list[this.index];
                return true;
            }
            this.currentElement = this.list;
            this.index = 0;
            return false;
        }
        #endregion

        private object currentElement;
        private int index;
        private ArrayList list;
    }
```

注意 ReverseArrayListEnumerator 类与前面的 ArrayListEnumeratorSimple 类的区别，主

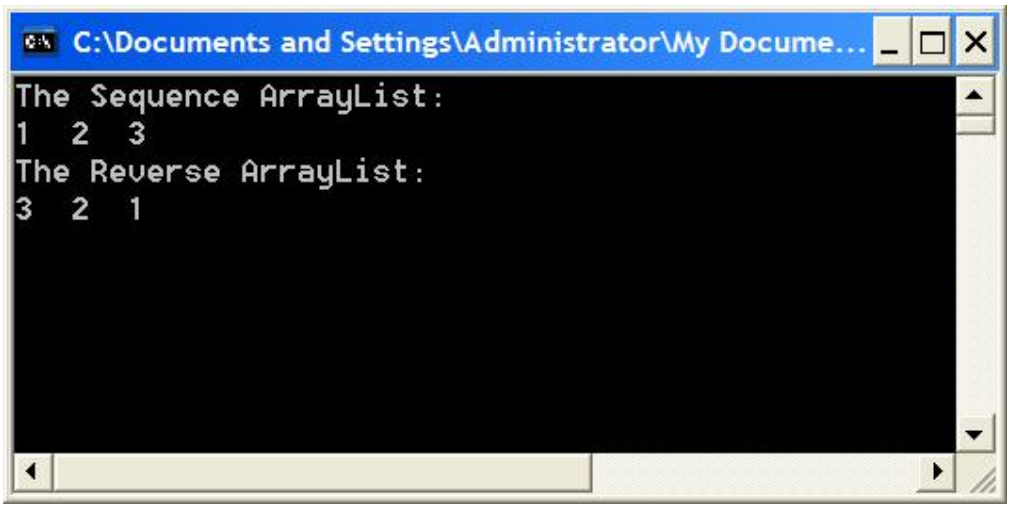
要在于遍历下一个元素的顺序。ReverseArrayListEnumerator 中的 MoveNext()方法，将下标往前移动，以保证元素遍历的逆序。同时在构造函数初始化时，将整个 ArrayList 对象的元素个数赋予下标的初始值：

```
this.index = list.Count;
```

我们来比较一下 ArrayList 和 ReversieArrayList 类之间，通过 foreach 遍历后的结果。

```
[STAThread]
public static void Main(string[] args)
{
    ArrayList al = new ArrayList();
    al.Add(1);
    al.Add(2);
    al.Add(3);
    ReverseArrayList ral = new ReverseArrayList();
    ral.Add(1);
    ral.Add(2);
    ral.Add(3);
    Console.WriteLine("The Sequence ArrayList:");
    foreach (int i in al)
    {
        Console.Write("{0}  ",i);
    }
    Console.WriteLine();
    Console.WriteLine("The Reverse ArrayList:");
    foreach (int i in ral)
    {
        Console.Write("{0}  ",i);
    }
    Console.ReadLine();
}
```

我们分别将数字 1，2，3 以同样的顺序添加到 ArrayList 和 ReverseArrayList 对象中，然后再通过 foreach 语句遍历输出其内部元素。运行后，很明显可以看到遍历 ArrayList 对象 al，其顺序为 1，2，3；而 ReverseArrayList 则为 3，2，1。



由于我们应用 Iterator 模式，将迭代器与集合对象完全分离，所以，即便我们完全修改了 ReverseArrayList 的遍历方式，实现 ReverseArrayList 也是非常容易的，同时它并没有影响到集合对象本身存储数据对象的职能。