31天重构系列笔记

31 Days of Refactoring

前言

前两天写了一篇程序猿也爱学英语（上），有图有真相的文章，写作那篇文章只是自己一时兴起，或者说是自己的兴趣使然。文中的观点只是自己的学习心得和体会，属一家之言且鉴于本人不是学英语出身，所以也肯定有不正确的地方，也欢迎大家积极讨论并给我留言，再次感谢大家的热烈支持。关于大家询问下篇的发布问题，我想我会尽力在周末完成。

这几天由于刚发布完项目，所以有比较充裕的时间整理自己的知识库，发现一年多以前学习并记录了31天重构系列笔记，至今仍回味无穷，索性重新阅读、纠正错误并重新排版整理出来，希望再次和大家一起分享。

对31天重构系列文章最早接触是在2009年10月份，由于当时没有订阅Sean Chambers的 blog，所以是在国外的社区上闲逛的时候链接过去的。基于文章中的重构点都比较常用，所以一口气看完了整个系列，同时由于这些重构Tips基本上项目都在使用，只是我们没有专门把它标示和整理出来，所以当时也没有引起多大的重视。

但就在三年前，当时我们在做一个WPF的重构项目且鉴于团队成员技术和经验参差不齐，非常必要专门整理一个重构的纲要，所以就收集和整理了很多的资料（31天重构也在其中）。当然这个系列除了用语重构Tips.之外，也非常适合做新系统的代码规范参考。总而言之：只要有代码的地方，这个重构规范就很有价值。同时鉴于当时自己刚到新加坡这个美丽的城市，没有亲戚或者朋友，周末也不想出去闲逛，所以才静下心来用了足足两天时间学习并写完了这个系列笔记。

31天重构这个系列和《代码大全》、《重构：改善既有代码的设计》比较起来最大的特点就是比较简单且浅显易懂。我这系列文章也都是学习并概括Sean Chambers的31天重构的知识要领，所以如果大家对这个笔记有任何的问题或者异议也可以指出，或者大家可以直接去看原文（即可掌握了技术，又可以学习英语！）：

http://www.lostechies.com/blogs/sean_chambers/archive/2009/07/31/31-days-of-refactoring.aspx。

代码下载地址：http://github.com/schambers/days-of-refactoring

原作者：Sean Chambers

笔记者：圣殿骑士

31 Days of Refactoring

Useful refactoring techniques you have to know

October 2009

Sean Chambers, Simone Chiaretta

Contents

1. 封装集合

概念：本文所讲的封装集合就是把集合进行封装，只提供调用端需要的接口。

正文：在很多时候，我们都不希望把一些不必要的操作暴露给调用端，只需要给它所需要的操作或数据就行，那么做法就是封装。这个重构在微软的代码库也经常遇到。比如最经典的属性对字段的封装就是一个很好的例子，那么下面我们将看到对集合的封装，如下代码所示，调用端只需要一个集合的信息，而我们则提供了一个IList的集合，大家都知道IList具有对集合的所有操作，所以这会带来很多隐患，最好的做法就是对它进行重构。

```
using System.Collections.Generic;
namespace LosTechies.DaysOfRefactoring.EncapsulateCollection.Before
{
    public class Order
    {
        private List<OrderLine> _orderLines;
        private double _orderTotal;
        public IList<OrderLine> OrderLines
        {
            get { return _orderLines; }
        }
        public void AddOrderLine(OrderLine orderLine)
        {
            _orderTotal += orderLine.Total;
            _orderLines.Add(orderLine);
        }
        public void RemoveOrderLine(OrderLine orderLine)
        {
            orderLine = _orderLines.Find(o => o == orderLine);
            if (orderLine == null)
                return;
            _orderTotal -= orderLine.Total;
            _orderLines.Remove(orderLine);
        }
    }
    public class OrderLine
    {
        public double Total { get; private set; }
    }
}
```

那么重构之后，我们把IList换成了IEnumerable，大家都知道只包括一个返回值为IEnumerator的GetEnumerator()方法，所以这样只能遍历取出它的值，而不能对这个集合做出改变，这正是我们所需要的结果，具体代码如下：

```
using System.Collections.Generic;
namespace LosTechies.DaysOfRefactoring.EncapsulateCollection.After
{
    public class Order
    {
        private List<OrderLine> _orderLines;
        private double _orderTotal;
        public IEnumerable<OrderLine> OrderLines
        {
            get { return _orderLines; }
        }
        public void AddOrderLine(OrderLine orderLine)
        {
            _orderTotal += orderLine.Total;
            _orderLines.Add(orderLine);
        }
        public void RemoveOrderLine(OrderLine orderLine)
        {
            orderLine = _orderLines.Find(o => o == orderLine);
            if (orderLine == null)
                return;
            _orderTotal -= orderLine.Total;
            _orderLines.Remove(orderLine);
        }
    }
    public class OrderLine
    {
        public double Total { get; private set; }
    }
}
```

总结：这个例子很容易让我们想到以前系统间耦合常喜欢用数据库。每个系统都会操作数据库，并且有些系统还会对数据库的表结构或字段进行修改，那么这很容易就会造成维护的地狱，很明智的一个做法就是使用SOA来隔开这些耦合，让一些只需要数据展示的系统得到自己需要的数据即可。

2. 移动方法

概念：本文所讲的移动方法就是方法放在合适的位置（通常指放在合适的类中）。

正文：移动方法是一个很简单也很常见的重构，只要是系统就会存在很多类，那么类里面包括很多方法，如果一个方法经常被另外一个类使用（比本身的类使用还多）或者这个方法本身就不应该放在这个类里面，那么这个适合应该考虑把它移到合适的类中。代码如下：

```
namespace LosTechies.DaysOfRefactoring.MoveMethod.Before
{
    public class BankAccount
    {
        public BankAccount(int accountAge, int creditScore, AccountInterest accountInterest)
        {
            AccountAge = accountAge;
            CreditScore = creditScore;
            AccountInterest = accountInterest;
        }

        public int AccountAge { get; private set; }
        public int CreditScore { get; private set; }
```

```
        public AccountInterest AccountInterest { get; private set; }

        public double CalculateInterestRate()
        {
            if (CreditScore > 800)
                return 0.02;

            if (AccountAge > 10)
                return 0.03;

            return 0.05;
        }
    }

    public class AccountInterest
    {
        public BankAccount Account { get; private set; }

        public AccountInterest(BankAccount account)
        {
            Account = account;
        }

        public double InterestRate
        {
            get { return Account.CalculateInterestRate(); }
        }

        public bool IntroductoryRate
        {
            get { return Account.CalculateInterestRate() < 0.05; }
        }
    }
}
```

移动以后大家可以看到BankAccount类的职责也单一，同时CalculateInterestRate也放到了经常使用且适合它的类中了，所以此重构是一个比较好的重构，能让整个代码变得更加合理。

```
namespace LosTechies.DaysOfRefactoring.MoveMethod.After
{
    public class AccountInterest
    {
        public BankAccount Account { get; private set; }
        public AccountInterest(BankAccount account)
        {
            Account = account;
        }
        public double InterestRate
        {
            get { return CalculateInterestRate(); }
        }
        public bool IntroductoryRate
        {
            get { return CalculateInterestRate() < 0.05; }
        }
        public double CalculateInterestRate()
        {
            if (Account.CreditScore > 800)
                return 0.02;
            if (Account.AccountAge > 10)
                return 0.03;
            return 0.05;
        }
    }
}


namespace LosTechies.DaysOfRefactoring.MoveMethod.After
{
    public class BankAccount
    {
        public BankAccount(int accountAge, int creditScore, AccountInterest accountInterest)
        {
            AccountAge = accountAge;
            CreditScore = creditScore;
            AccountInterest = accountInterest;
        }
        public int AccountAge { get; private set; }
        public int CreditScore { get; private set; }
        public AccountInterest AccountInterest { get; private set; }
    }
}
```

总结：这个重构法则在很多时候能让我们把代码组织的结构调整得更合理，同时也能给以后的维护带来方便。

3. 提升方法

概念：提升方法是指将一个很多继承类都要用到的方法提升到基类中。

正文：提升方法是指将一个很多继承类都要用到的方法提升到基类中，这样就能减少代码量，同时让类的结构更清晰。如下代码所示，Turn方法在子类Car 和Motorcycle 都会用到，因为Vehicle 都会有这个方法，所以我们就会想到把它提到基类中。

```
namespace LosTechies.DaysOfRefactoring.PullUpMethod.Before
{
    public abstract class Vehicle
    {
        // other methods
```

```
        // other methods
    }
    public class Car : Vehicle
    {
        public void Turn(Direction direction)
        {
            // code here
        }
    }
    public class Motorcycle : Vehicle
    {
    }
    public enum Direction
    {
        Left,
        Right
    }
}
```

重构后的代码如下，那么现在Car 和Motorcycle 都具有Turn这个方法，如果这个方法修改也只需要修改基类即可，所以给维护和以后的重构带来了方便。

```
namespace LosTechies.DaysOfRefactoring.PullUpMethod.After
{
    public abstract class Vehicle
    {
        public void Turn(Direction direction)
        {
            // code here
        }
    }
    public class Car : Vehicle
    {
    }
    public class Motorcycle : Vehicle
    {
    }
    public enum Direction
    {
        Left,
        Right
    }
}
```

总结：这个重构要根据具体情况使用，如果不是每个子类都有这个方法的话，可以考虑使用接口或者其他方式。

4. 降低方法

概念：本文中的降低方法和前篇的提升方法整好相反，也就是把个别子类使用到的方法从基类移到子类里面去。

正文：如下代码所示，Animal 类中的方法Bark只有在其子类Dog 中使用，所以最好的方案就是把这个方法移到子类Dog 中。

```
namespace LosTechies.DaysOfRefactoring.PushDownMethod.Before
{
    public abstract class Animal
    {
        public void Bark()
        {
            // code to bark
        }
    }

    public class Dog : Animal
    {
    }

    public class Cat : Animal
    {
    }
}
```

重构后的代码如下，同时如果在父类Animal 中如果没有其他的字段或者公用方法的话，可以考虑把Bark方法做成一个接口，从而去掉Animal 类。

```
namespace LosTechies.DaysOfRefactoring.PushDownMethod.After
{
    public abstract class Animal
    {
    }

    public class Dog : Animal
    {
        public void Bark()
        {
            // code to bark
        }
    }

    public class Cat : Animal
    {
    }
}
```

总结：面向对象三大特征（继承、封装、多态）很多时候可以帮助我们，但同时也可能会造成使用过度或者使用不当，所以如何把握好设计，这个就变得至关重要。在什么时候使用继承的方式，在什么时候使用组合和聚合，接口和继承类的选择等久成了我们的重点。

5. 提升字段

概念：本文中的提升字段和前面的提升方法颇为相似，就是把子类公用的字段提升到基类中，从而达到公用的目的。

正文：如下代码所示， Account 的两个子类CheckingAccount 和SavingsAccount 都有minimumCheckingBalance 字段，所以可以考虑把这个字段提到基类中。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace LosTechies.DaysOfRefactoring.PullUpField.Before
{
    public abstract class Account
    {
    }
    public class CheckingAccount : Account
    {
        private decimal _minimumCheckingBalance = 5m;
    }
    public class SavingsAccount : Account
    {
        private decimal _minimumSavingsBalance = 5m;
    }
}
```

重构后的代码如下，这样提的前提是这些子类有一个基类或者有很多相似的字段和方法，不然为了一个字段而单独建立一个抽象类是不可取的，所以这个就需要具体权衡。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace LosTechies.DaysOfRefactoring.PullUpField.After
{
    public abstract class Account
    {
        protected decimal _minimumBalance = 5m;
    }
    public class CheckingAccount : Account
    {
    }
    public class SavingsAccount : Account
    {
    }
}
```

总结：这个重构的策略比较简单，同时也是比较常用的一些做法，最主要就是要注意权衡是否真的有这个必要，看这样做究竟有没有什么好处（比如只需要改一个地方，维护简便了，同时代码量也更少了等）。

6. 降低字段

概念：本文中的降低字段和前篇的提升字段正好相反，就是把基类中只有某些少数类用到的字段降低到使用它们的子类中。

正文：如下代码所示，基类Task 类中的_resolution字段只会在子类BugTask 中用到，所以就考虑把它放到BugTask 类中。

```
namespace LosTechies.DaysOfRefactoring.PushDownField.Before
{
    public abstract class Task
    {
        protected string _resolution;
    }
    public class BugTask : Task
    {
    }
    public class FeatureTask : Task
    {
    }
}
```

重构后的代码如下所示，这样做的好处可以简化基类，同时让其他没有使用它的子类也变得更加简单，如果这样的字段比较多的话，使用此重构也能节约一部分内存。

```
namespace LosTechies.DaysOfRefactoring.PushDownField.After
{
    public abstract class Task
    {
    }
    public class BugTask : Task
    {
        private string _resolution;
    }
    public class FeatureTask : Task
    {
    }
}
```

总结：此重构也是一个非常简单的重构，在很多时候我们都会不自觉的使用它。

7. 重命名（方法，类，参数）

概念：本文中的改名（方法，类，参数）是指在写代码的时候对类、方法、参数、委托、事件等等元素取一个有意义的名称。

正文：如下代码所示，加入一个公司建立一个员工的类，类中有一个员工名字的字段和一个按照小时计算员工收入的方法，那么下面代码的取名就显得很难理解了，所以我们会重构名称。

```
namespace LosTechies.DaysOfRefactoring.Rename.Before
{
    public class Person
    {
        public string FN { get; set; }
        public decimal ClcHrlyPR()
        {
            // code to calculate hourly payrate
```

```
            return 0m;
        }
    }
}
```

重构后代码如下所示，这样看起来就非常清晰，如果有新进项目组的成员，也会变得很乐意看这个代码。

```
namespace LosTechies.DaysOfRefactoring.Rename.After
{
    // Changed the class name to Employee
    public class Employee
    {
        public string FirstName { get; set; }
        public decimal CalculateHourlyPay()
        {
            // code to calculate hourly payrate
            return 0m;
        }
    }
}
```

总结：此重构经常被广大程序员所忽视，但是带来的隐患是不可估量的，也许老板要修改功能，那我们来看这段没有重构的代码（就算是自己写的，但由于时间和项目多等关系，我们也很难理解了），然后就会变得焦头烂额。相反重构后的代码就会觉得一目了然、赏心悦目。

8. 使用委派代替继承

概念：本文中的“使用委派代替继承”是指在根本没有父子关系的类中使用继承是不合理的，可以用委派的方式来代替。

如下代码所示，Child 和Sanitation （公共设施）是没有逻辑上的父子关系，因为小孩不可能是一个公共设施吧！所以我们为了完成这个功能可以考虑使用委派的方式。

```
namespace LosTechies.DaysOfRefactoring.ReplaceInheritance.Before
{
    public class Sanitation
    {
        public string WashHands()
        {
            return "Cleaned!";
        }
    }

    public class Child : Sanitation
    {
    }
}
```

重构后的代码如下，把Sanitation 委派到Child 类中，从而可以使用WashHands这个方法，这种方式我们经常会用到，其实IOC也使用到了这个原理，可以通过构造注入和方法注入等。

```
namespace LosTechies.DaysOfRefactoring.ReplaceInheritance.After
{
    public class Sanitation
    {
        public string WashHands()
        {
            return "Cleaned!";
        }
    }

    public class Child
    {
        private Sanitation Sanitation { get; set; }

        public Child()
        {
            Sanitation = new Sanitation();
        }

        public string WashHands()
        {
            return Sanitation.WashHands();
        }
    }
}
```

总结：这个重构是一个很好的重构，在很大程度上解决了滥用继承的情况，很多设计模式也用到了这种思想（比如桥接模式、适配器模式、策略模式等）。

9. 提取接口

概念：本文中的“提取接口” 是指超过一个的类要使用某一个类中部分方法时，我们应该解开它们之间的依赖，让调用者使用接口，这很容易实现也可以降低代码的耦合性。

正文：如下代码所示，RegistrationProcessor 类只使用到了ClassRegistration 类中的Create方法和Total 字段，所以可以考虑把他们做成接口给RegistrationProcessor 调用。

```
namespace LosTechies.DaysOfRefactoring.ExtractInterface.Before
{
    public class ClassRegistration
    {
        public void Create()
        {
            // create registration code
        }
        public void Transfer()
        {
            // class transfer code
        }
        public decimal Total { get; private set; }
    }
    public class RegistrationProcessor
    {
```

```
    {
        public decimal ProcessRegistration(ClassRegistration registration)
        {
            registration.Create();
            return registration.Total;
        }
    }
}
```

重构后的代码如下，我们提取了一个IClassRegistration 接口，同时让ClassRegistration 继承此接口，然后调用端RegistrationProcessor 就可以直接通过IClassRegistration 接口进行调用。

```
namespace LosTechies.DaysOfRefactoring.ExtractInterface.After
{
    public interface IClassRegistration
    {
        void Create();
        decimal Total { get; }
    }
    public class ClassRegistration : IClassRegistration
    {
        public void Create()
        {
            // create registration code
        }
        public void Transfer()
        {
            // class transfer code
        }
        public decimal Total { get; private set; }
    }
    public class RegistrationProcessor
    {
        public decimal ProcessRegistration(IClassRegistration registration)
        {
            registration.Create();
            return registration.Total;
        }
    }
}
```

总结：这个重构策略也是一个常见的运用，很多设计模式也会在其中运用此思想（如简单工程、抽象工厂等都会通过接口来解开依赖）。

10. 提取方法

概念：本文中的把某些计算复杂的过程按照功能提取成各个小方法，这样就可以使代码的可读性、维护性得到提高。

正文：如下代码所示，CalculateGrandTotal方法里面包含了多个逻辑，第一计算subTotal 的总和，第二subTotal 要循环减去discount，也就是计算Discounts，第三就是计算Tax。所以我们可以根据功能把他们拆分成三个小方法。

```
using System.Collections.Generic;

namespace LosTechies.DaysOfRefactoring.ExtractMethod.Before
{
    public class Receipt
    {
        private IList<decimal> Discounts { get; set; }
        private IList<decimal> ItemTotals { get; set; }

        public decimal CalculateGrandTotal()
        {
            decimal subTotal = 0m;
            foreach (decimal itemTotal in ItemTotals)
                subTotal += itemTotal;

            if (Discounts.Count > 0)
            {
                foreach (decimal discount in Discounts)
                    subTotal -= discount;
            }

            decimal tax = subTotal * 0.065m;

            subTotal += tax;

            return subTotal;
        }
    }
}
```

重构后的代码如下，然后CalculateGrandTotal方法就直接调用CalculateSubTotal、CalculateDiscounts、CalculateTax，从而是整个逻辑看起来更加清晰，并且可读性和维护性也得到了大大提高。

```
using System.Collections.Generic;

namespace LosTechies.DaysOfRefactoring.ExtractMethod.After
{
    public class Receipt
    {
        private IList<decimal> Discounts { get; set; }
        private IList<decimal> ItemTotals { get; set; }

        public decimal CalculateGrandTotal()
        {
            decimal subTotal = CalculateSubTotal();

            subTotal = CalculateDiscounts(subTotal);

            subTotal = CalculateTax(subTotal);
```

```
            subTotal = CalculateTax(subTotal);

            return subTotal;
        }

        private decimal CalculateTax(decimal subTotal)
        {
            decimal tax = subTotal * 0.065m;

            subTotal += tax;
            return subTotal;
        }

        private decimal CalculateDiscounts(decimal subTotal)
        {
            if (Discounts.Count > 0)
            {
                foreach (decimal discount in Discounts)
                    subTotal -= discount;
            }
            return subTotal;
        }

        private decimal CalculateSubTotal()
        {
            decimal subTotal = 0m;
            foreach (decimal itemTotal in ItemTotals)
                subTotal += itemTotal;
            return subTotal;
        }
    }
}
```

总结：这个重构在很多公司都有一些的代码规范作为参考，比如一个类不能超过多少行代码，一个方法里面不能超过多少行代码，这在一定程度上也能使程序员把这些复杂的逻辑剥离成意义很清楚的小方法。

11. 使用策略类

概念：本文中的“使用策略类” 是指用设计模式中的策略模式来替换原来的switch case和if else语句，这样可以解开耦合，同时也使维护性和系统的可扩展性大大增强。

正文：如下面代码所示，ClientCode 类会更加枚举State的值来调用ShippingInfo 的不同方法，但是这样就会产生很多的判断语句，如果代码量加大，类变得很大了的话，维护中改动也会变得很大，每次改动一个地方，都要对整个结构进行编译（假如是多个工程），所以我们想到了对它进行重构，剥开耦合。

```
namespace LosTechies.DaysOfRefactoring.SwitchToStrategy.Before
{
    public class ClientCode
    {
        public decimal CalculateShipping()
        {
            ShippingInfo shippingInfo = new ShippingInfo();
            return shippingInfo.CalculateShippingAmount(State.Alaska);
        }
    }

    public enum State
    {
        Alaska,
        NewYork,
        Florida
    }

    public class ShippingInfo
    {
        public decimal CalculateShippingAmount(State shipToState)
        {
            switch (shipToState)
            {
                case State.Alaska:
                    return GetAlaskaShippingAmount();
                case State.NewYork:
                    return GetNewYorkShippingAmount();
                case State.Florida:
                    return GetFloridaShippingAmount();
                default:
                    return 0m;
            }
        }

        private decimal GetAlaskaShippingAmount()
        {
            return 15m;
        }

        private decimal GetNewYorkShippingAmount()
        {
            return 10m;
        }

        private decimal GetFloridaShippingAmount()
        {
            return 3m;
        }
    }
}
```

重构后的代码如下所示，抽象出一个IShippingCalculation 接口，然后把ShippingInfo 类里面的GetAlaskaShippingAmount、GetNewYorkShippingAmount、

GetFloridaShippingAmount三个方法分别提炼成三个类，然后继承自IShippingCalculation 接口，这样在调用的时候就可以通过IEnumerable<IShippingCalculation> 来解除之前的switch case语句，这和IOC的做法颇为相似。

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace LosTechies.DaysOfRefactoring.SwitchToStrategy.After_WithIoC
{
    public interface IShippingInfo
    {
        decimal CalculateShippingAmount(State state);
    }

    public class ClientCode
    {
        [Inject]
        public IShippingInfo ShippingInfo { get; set; }

        public decimal CalculateShipping()
        {
            return ShippingInfo.CalculateShippingAmount(State.Alaska);
        }
    }

    public enum State
    {
        Alaska,
        NewYork,
        Florida
    }

    public class ShippingInfo : IShippingInfo
    {
        private IDictionary<State, IShippingCalculation> ShippingCalculations { get; set; }

        public ShippingInfo(IEnumerable<IShippingCalculation> shippingCalculations)
        {
            ShippingCalculations = shippingCalculations.ToDictionary(calc => calc.State);
        }

        public decimal CalculateShippingAmount(State shipToState)
        {
            return ShippingCalculations[shipToState].Calculate();
        }
    }

    public interface IShippingCalculation
    {
        State State { get; }
        decimal Calculate();
    }

    public class AlaskShippingCalculation : IShippingCalculation
    {
        public State State { get { return State.Alaska; } }

        public decimal Calculate()
        {
            return 15m;
        }
    }

    public class NewYorkShippingCalculation : IShippingCalculation
    {
        public State State { get { return State.NewYork; } }

        public decimal Calculate()
        {
            return 10m;
        }
    }

    public class FloridaShippingCalculation : IShippingCalculation
    {
        public State State { get { return State.Florida; } }

        public decimal Calculate()
        {
            return 3m;
        }
    }
}
```

总结：这种重构在设计模式当中把它单独取了一个名字——策略模式，这样做的好处就是可以隔开耦合，以注入的形式实现功能，这使增加功能变得更加容易和简便，同样也增强了整个系统的稳定性和健壮性。

12. 分解依赖

概念：本文中的“分解依赖” 是指对部分不满足我们要求的类和方法进行依赖分解，通过装饰器来达到我们需要的功能。

正文：正如下面代码所示，如果你要在你的代码中加入单元测试但有一部分代码是你不想测试的，那么你应用使用这个的重构。下面的例子中我们应用静态类来完成某些工作，但问题是在单元测试时我们无法mock静态类，所以我们只能引入静态类的装饰接口来分解对静态类的依赖。从而我们使我们的调用类只需要依赖于装饰接口就能完成这个操作。

```
namespace LosTechies.DaysOfRefactoring.BreakDependencies.Before
{
    public class AnimalFeedingService
```

```
    public class AnimalFeedingService
    {
        private bool FoodBowlEmpty { get; set; }
        public void Feed()
        {
            if (FoodBowlEmpty)
                Feeder.ReplenishFood();
            // more code to feed the animal
        }
    }
    public static class Feeder
    {
        public static void ReplenishFood()
        {
            // fill up bowl
        }
    }
}
```

重构后代码如下，我们添加一个接口和一个实现类，在实现类中调用静态类的方法，所以说具体做什么事情没有改变，改变的只是形式，但这样做的一个好处是增加了了代码的可测试性。在应用了分解依赖模式后，我们就可以在单元测试的时候mock一个IFeederService对象并通过AnimalFeedingService的构造函数传递给它。这样就可以完成我们需要的功能。

```
namespace LosTechies.DaysOfRefactoring.BreakDependencies.After
{
    public class AnimalFeedingService
    {
        public IFeederService FeederService { get; set; }
        public AnimalFeedingService(IFeederService feederService)
        {
            FeederService = feederService;
        }
        private bool FoodBowlEmpty { get; set; }
        public void Feed()
        {
            if (FoodBowlEmpty)
                FeederService.ReplenishFood();
            // more code to feed the animal
        }
    }
    public interface IFeederService
    {
        void ReplenishFood();
    }
    public class FeederService : IFeederService
    {
        public void ReplenishFood()
        {
            Feeder.ReplenishFood();
        }
    }
    public static class Feeder
    {
        public static void ReplenishFood()
        {
            // fill up bowl
        }
    }
}
```

总结：这个重构在很多时候和设计模式中的一些思想类似，使用中间的装饰接口来分解两个类之间的依赖，对类进行装饰，然后使它满足我们所需要的功能。

13. 提取方法对象

概念：本文中的“提取方法对象”是指当你发现一个方法中存在过多的局部变量时，你可以通过使用“提取方法对象”重构来引入一些方法，每个方法完成任务的一个步骤，这样可以使得程序变得更具有可读性。

正文：如下代码所示，Order 类中的Calculate方法要完成很多功能，在之前我们用“提取方法”来进行重构，现在我们采取“提取方法对象”来完成重构。

```
using System.Collections.Generic;

namespace LosTechies.DaysOfRefactoring.ExtractMethodObject.Before
{
    public class OrderLineItem
    {
        public decimal Price { get; private set; }
    }

    public class Order
    {
        private IList<OrderLineItem> OrderLineItems { get; set; }
        private IList<decimal> Discounts { get; set; }
        private decimal Tax { get; set; }

        public decimal Calculate()
        {
            decimal subTotal = 0m;

            // Total up line items
            foreach (OrderLineItem lineItem in OrderLineItems)
            {
                subTotal += lineItem.Price;
            }

            // Subtract Discounts
            foreach (decimal discount in Discounts)
                subTotal -= discount;
```

```
            // Calculate Tax
            decimal tax = subTotal * Tax;

            // Calculate GrandTotal
            decimal grandTotal = subTotal + tax;

            return grandTotal;
        }
    }
}
```

正如下代码所示，我们引入了OrderCalculator类，该类实现了所有的计算方法，Order类将自身传递给 OrderCalculator类并调用Calculate方法完成计算过程。

```
using System.Collections.Generic;

namespace LosTechies.DaysOfRefactoring.ExtractMethodObject.After
{
    public class OrderLineItem
    {
        public decimal Price { get; private set; }
    }

    public class Order
    {
        public IEnumerable<OrderLineItem> OrderLineItems { get; private set; }
        public IEnumerable<decimal> Discounts { get; private set; }
        public decimal Tax { get; private set; }

        public decimal Calculate()
        {
            return new OrderCalculator(this).Calculate();
        }
    }

    public class OrderCalculator
    {
        private decimal SubTotal { get; set; }
        private IEnumerable<OrderLineItem> OrderLineItems { get; set; }
        private IEnumerable<decimal> Discounts { get; set; }
        private decimal Tax { get; set; }

        public OrderCalculator(Order order)
        {
            OrderLineItems = order.OrderLineItems;
            Discounts = order.Discounts;
            Tax = order.Tax;
        }

        public decimal Calculate()
        {
            CalculateSubTotal();

            SubtractDiscounts();

            CalculateTax();

            return SubTotal;
        }

        private void CalculateSubTotal()
        {
            // Total up line items
            foreach (OrderLineItem lineItem in OrderLineItems)
                SubTotal += lineItem.Price;
        }

        private void SubtractDiscounts()
        {
            // Subtract Discounts
            foreach (decimal discount in Discounts)
                SubTotal -= discount;
        }

        private void CalculateTax()
        {
            // Calculate Tax
            SubTotal += SubTotal * Tax;
        }
    }
}
```

总结：本文的重构方法在有的时候还是比较有用，但这样会造成字段的增加，同时也会带来一些维护的不便，它和“提取方法”最大的区别就是一个通过方法返回需要的数据，另一个则是通过字段来存储方法的结果值，所以在很大程度上我们都会选择“提取方法”。

14. 分离职责

概念：本文中的“分离职责”是指当一个类有许多职责时，将部分职责分离到独立的类中，这样也符合面向对象的五大特征之一的单一职责原则，同时也可以使代码的结构更加清晰，维护性更高。

正文：如下代码所示，Video类有两个职责，一个是处理video rental，另一个是计算每个客户的总租金。我们可以将这两个职责分离出来，因为计算每个客户的总租金可以在Customer计算，这也比较符合常理。

```
using System.Collections.Generic;
using System.Linq;

namespace LosTechies.DaysOfRefactoring.BreakResponsibilities.Before
{
```

```
{
    public class Video
    {
        public void PayFee(decimal fee)
        {
        }

        public void RentVideo(Video video, Customer customer)
        {
            customer.Videos.Add(video);
        }

        public decimal CalculateBalance(Customer customer)
        {
            returncustomer.LateFees.Sum();
        }
    }

    public class Customer
    {
        public IList<decimal> LateFees { get; set; }
        public IList<Video> Videos { get; set; }
    }
}
```

重构后的代码如下，这样Video 的职责就变得很清晰，同时也使代码维护性更好。

```
using System.Collections.Generic;
using System.Linq;

namespace LosTechies.DaysOfRefactoring.BreakResponsibilities.After
{
    public class Video
    {
        public void RentVideo(Video video, Customer customer)
        {
            customer.Videos.Add(video);
        }
    }

    public class Customer
    {
        public IList<decimal> LateFees { get; set; }
        public IList<Video> Videos { get; set; }

        public void PayFee(decimal fee)
        {
        }

        public decimal CalculateBalance(Customer customer)
        {
            return customer.LateFees.Sum();
        }
    }
}
```

总结：这个重构经常会用到，它和之前的“移动方法”有几分相似之处，让方法放在合适的类中，并且简化类的职责，同时这也是面向对象五大原则之一和设计模式中的重要思想。

15. 移除重复内容

概念：本文中的“移除重复内容”是指把一些很多地方都用到的逻辑提炼出来，然后提供给调用者统一调用。

正文：如下代码所示，ArchiveRecord和CloseRecord都会用到Archived = true; 和DateArchived = DateTime.Now; 这两条语句，所以我们就可以对它进行重构。

```
using System;

namespace LosTechies.DaysOfRefactoring.RemoveDuplication.Before
{
    public class MedicalRecord
    {
        public DateTime DateArchived { get; private set; }
        public bool Archived { get; private set; }

        public void ArchiveRecord()
        {
            Archived = true;
            DateArchived = DateTime.Now;
        }

        public void CloseRecord()
        {
            Archived = true;
            DateArchived = DateTime.Now;
        }
    }
}
```

重构后的代码如下所示，我们提炼了SwitchToArchived方法来封装公用的操作，然后给ArchiveRecord和CloseRecord统一调用。

```
using System;

namespace LosTechies.DaysOfRefactoring.RemoveDuplication.After
{
    public class MedicalRecord
    {
        public DateTime DateArchived { get; private set; }
        public bool Archived { get; private set; }
```

```
        public void ArchiveRecord()
        {
            SwitchToArchived();
        }

        public void CloseRecord()
        {
            SwitchToArchived();
        }

        private void SwitchToArchived()
        {
            Archived = true;
            DateArchived = DateTime.Now;
        }
    }
}
```

总结：这个重构很简单，绝大多数程序员都会使用这种重构方法，但有时由于习惯、时间、赶进度等原因而忽略它，所以会使得整个系统杂乱无章，到处都是Ctrl+C和Ctrl+V的痕迹。

16. 封装条件

概念：本文中的“封装条件”是指条件关系比较复杂时，代码的可读性会比较差，所以这时我们应当根据条件表达式是否需要参数将条件表达式提取成可读性更好的属性或者方法，如果条件表达式不需要参数则可以提取成属性，如果条件表达式需要参数则可以提取成方法。

正文：如下代码所示，PerformCoolFunction里面的if条件判断比较复杂，看起来有点杂乱，所以就把它提出来。

```
using System;
namespace LosTechies.DaysOfRefactoring.EncapsulateConditional.Before
{
    public class RemoteControl
    {
        private string[] Functions { get; set; }
        private string Name { get; set; }
        private int CreatedYear { get; set; }
        public string PerformCoolFunction(string buttonPressed)
        {
            // Determine if we are controlling some extra function
            // that requires special conditions
            if (Functions.Length > 1 && Name == "RCA" && CreatedYear > DateTime.Now.Year - 2)
                return "doSomething";
        }
    }
}
```

如下代码所示，我们把条件表达式封装成HasExtraFunctions属性，这样先前的条件判断就成了if (HasExtraFunctions)，所以这样就在很大程度上提高了可读性。

```
using System;
namespace LosTechies.DaysOfRefactoring.EncapsulateConditional.After
{
    public class RemoteControl
    {
        private string[] Functions { get; set; }
        private string Name { get; set; }
        private int CreatedYear { get; set; }
        private bool HasExtraFunctions
        {
            get { return Functions.Length > 1 && Name == "RCA" && CreatedYear > DateTime.Now.Year - 2; }
        }
        public string PerformCoolFunction(string buttonPressed)
        {
            // Determine if we are controlling some extra function
            // that requires special conditions
            if (HasExtraFunctions)
                return "doSomething";
        }
    }
}
```

总结：这个重构在很大程度上能改善代码的可读性，尤其是在一个逻辑很复杂的应用中，把这些条件判断封装成一个有意义的名字，这样很复杂的逻辑也会立刻变得简单起来。

17. 提取父类

概念：本文中的“提取父类”是指类中有一些字段或方法，你想把它们提取到父类中以便同一继承层次的其它类也可以访问他们，这个和之前的很多重构有异曲同工之处。

正文：Dog 类中的EatFood和IGroom有可能被其他类用到，因为他们都是动物的一些公有性质，所以这个时候我们就会考虑对它进行提炼。

```
namespace LosTechies.DaysOfRefactoring.ExtractSuperclass.Before
{
    public class Dog
    {
        public void EatFood()
        {
            // eat some food
        }

        public void Groom()
        {
        // perform grooming
        }
    }
}
```

代码如下所示，提取了Animal 方法来封装公用的EatFood和Groom类，从而使其他继承了Animal 类的子类都可以使用这两个方法了。

代码如下所示，提取了Animal 方法来封装公用的EatFood和Groom类，从而使其他继承了Animal 类的子类都可以使用这两个方法了。

```
namespace LosTechies.DaysOfRefactoring.ExtractSuperclass.After
{
    public class Animal
    {
        public void EatFood()
        {
            // eat some food
        }
        public void Groom()
        {
            // perform grooming
        }
    }
    public class Dog : Animal
    {
    }
}
```

总结：这个重构是典型的继承用法，很多程序员都会选择这样做，但是要注意正确的使用，不要造成过度使用了继承，如果过度使用了，请考虑用接口、组合和聚合来实现。

18. 使用条件判断代替异常

概念：本文中的“使用条件判断代替异常”是指把没有必要使用异常做判断的条件尽量改为条件判断。

正文：如下代码所示，在日常的编码中我们经常需要用到异常来控制程序流，Start方法里面用try catch 做条件判断，我们知道这里没有必要使用这种方式，因为你不需要做类型不可控的类型转换，也不需要处理异常行为，所以我们应该对它进行重构。

```
namespace LosTechies.DaysOfRefactoring.ReplaceException.Before
{
    public class Microwave
    {
        private IMicrowaveMotor Motor { get; set; }
        public bool Start(object food)
        {
            bool foodCooked = false;
            try
            {
                Motor.Cook(food);
                foodCooked = true;
            }
            catch (InUseException)
            {
                foodcooked = false;
            }
            return foodCooked;
        }
    }
}
```

重构后的代码如下所示，try catch 做条件判断的语句改成了if return 的方式，这样在很多程度上统一了代码的书写，同时也提高了性能。

```
namespace LosTechies.DaysOfRefactoring.ReplaceException.After
{
    public class Microwave
    {
        private IMicrowaveMotor Motor { get; set; }
        public bool Start(object food)
        {
            if (Motor.IsInUse)
                return false;
            Motor.Cook(food);
            return true;
        }
    }
}
```

总结： 这个重构在项目代码中也经常用到，因为对于一部分程序员，是很难把握什么时候用try catch，什么地方该用try catch。记得之前大家还专门讨论过这些，比如如何用好以及在大中型项目中应该把它放在哪一个组件中等。

19. 提取工厂类

概念：本文中的“提取工厂类”是指如果要创建的对象很多，则代码会变的很复杂。一种很好的方法就是提取工厂类。

正文：一般来说我们需要在代码中设置一些对象，以便获得它们的状态，从而使用对象，所谓的设置通常来说就是创建对象的实例并调用对象的方法。有时如果要创建的对象很多，则代码会变的很复杂。这便是工厂模式发挥作用的情形。工厂模式的复杂应用是使用抽象工厂创建对象集，但我们在这里只是使用基本的工厂类创建对象的一个简单应用。

如下代码所示，New方法包含创建类的整个逻辑，如果现在要创建的类比较多而且逻辑比较复杂的话（如根据不同条件创建对象，什么时候创建对象），我们的New方法逻辑会变得很大，同时代码也变得很难维护。所以我们就会采用提取工厂类的方式进行提炼。

```
namespace LosTechies.DaysOfRefactoring.ExtractServiceClass.Before
{
    public class PoliceCarController
    {
        public PoliceCar New(int mileage, bool serviceRequired)
        {
            PoliceCar policeCar = new PoliceCar();
            policeCar.ServiceRequired = serviceRequired;
            policeCar.Mileage = mileage;

            return policeCar;
        }
    }
}
```

那么重构后的代码如下，New方法变得很简单了，指需要调用实现接IPoliceCarFactory 接口的PoliceCarFactory 类就可以返回对象，这样就隔开了创建对象的逻辑，如果需求现在变为根据不同的条件创建不同的对象，什么时候创建对象等都变成了比较简单的事情，在后期可以把对象都配置在XML里面，使用反射的方式实现IOC注入创建。

```
namespace LosTechies.DaysOfRefactoring.ExtractServiceClass.After
{
    public interface IPoliceCarFactory
    {
        PoliceCar Create(int mileage, bool serviceRequired);
    }
    public class PoliceCarFactory : IPoliceCarFactory
    {
        public PoliceCar Create(int mileage, bool serviceRequired)
        {
            PoliceCar policeCar = new PoliceCar();
            policeCar.ReadForService = serviceRequired;
            policeCar.Mileage = mileage;
            return policeCar;
        }
    }
    public class PoliceCarController
    {
        public IPoliceCarFactory PoliceCarFactory { get; set; }
        public PoliceCarController(IPoliceCarFactory policeCarFactory)
        {
            PoliceCarFactory = policeCarFactory;
        }
        public PoliceCar New(int mileage, bool serviceRequired)
        {
            return PoliceCarFactory.Create(mileage, serviceRequired);
        }
    }
}
```

总结：这个重构经常会在项目中使用，如果要创建的对象是一个，你可以采用简单工厂，但是这种方式还是会存在很多依赖，维护起来也比较不方便。所以推荐使用工厂方法模式，把实例化延迟到子类。如果你要创建一系列的对象，那么就推荐你使用抽象工厂模式，但是要注意不要过度设计，只要能满足不断变化的需求和给以后的维护和重构带来方便即可。

20. 提取子类

概念：本文中的”提取子类”是指把基类中的一些不是所有子类都需要访问的方法调整到子类中。

正文：当你的基类中存在一些方法不是所有的子类都需要访问，你想将它们调整到子类中时，这个重构会变得很有用了。如下代码所示，我们需要一个 Registration类用来处理学生选课的信息。但是当Registration类开始工作后，我们意识到我们会在两种不同的上下文中使用Registration类，NonRegistrationAction和Notes只有在我们处理未注册情况下才用到。

所以我们将NonRegistration和Notes提到单独的NonRegistration类中。

```
using System;
namespace LosTechies.DaysOfRefactoring.SampleCode.ExtractSubclass.Before
{
    public class Registration
    {
        public NonRegistrationAction Action { get; set; }
        public decimal RegistrationTotal { get; set; }
        public string Notes { get; set; }
        public string Description { get; set; }
        public DateTime RegistrationDate { get; set; }
    }
}
```

重构后的代码如下所示，这样也满足面向对象五大原则之一的单一职责。同时也让类的结构变得更加清晰，增强了可维护性。

```
using System;
namespace LosTechies.DaysOfRefactoring.SampleCode.ExtractSubclass.After
{
    public class Registration
    {
        public decimal RegistrationTotal { get; set; }
        public string Description { get; set; }
        public DateTime RegistrationDate { get; set; }
    }
    public class NonRegistration : Registration
    {
        public NonRegistrationAction Action { get; set; }
        public string Notes { get; set; }
    }
}
```

总结：这个重构方法经常用来规范类的职责，和之前的一些重构方法也有些类似。

21. 合并继承

概念：本文中的”合并继承”是指如果子类的属性和方法也适合于基类，那么就可以移除子类，从而减少依赖关系。

正文：上一篇我们讲到“提取子类”重构是指当基类中的一个责任不被所有的子类所需要时，将这些责任提取到合适的子类中。而我们今天所要讲的的“合并继承”重构一般用在当我们觉得不需要子类的时候。

如下代码所示，StudentWebSite子类除了有一个属性用来说明网站是否是活动的外没有别的责任，在这种情形下我们意识到IsActive属性可以应用到所有的网站，所以我们可以将IsActive属性上移到基类中，并去掉StudentWebSite类。

```
using System.Collections.Generic;

namespace LosTechies.DaysOfRefactoring.SampleCode.CollapseHierarchy.Before
{
    public class Website
    {
        public string Title { get; set; }
```

```
        public string Title { get; set; }
        public string Description { get; set; }
        public IEnumerable<Webpage> Pages { get; set; }
    }

    public class StudentWebsite : Website
    {
        public bool IsActive { get; set; }
    }
}
```

重构后的代码如下：

```
using System.Collections.Generic;

namespace LosTechies.DaysOfRefactoring.SampleCode.CollapseHierarchy.After
{
    public class Website
    {
        public string Title { get; set; }
        public string Description { get; set; }
        public IEnumerable<Webpage> Pages { get; set; }
        public bool IsActive { get; set; }
    }
}
```

总结： 这篇和上篇其实最主要论述了子类和父类的继承关系以及如何判断什么时候需要使用继承，一般我们都能处理好这些关系，所以相对比较简单。

22. 分解方法

概念：本文中的”分解方法”是指把我们所做的这个功能不停的分解方法，直到将一个大方法分解为名字有意义且可读性更好的若干个小方法。

正文：如下代码所示，因为现实中AcceptPayment方法不会做这么多的事情。，所以我们通过几次分解将 AcceptPayment拆分成若干个名字有意义且可读性更好的小方法。

```
using System.Collections.Generic;
using System.Linq;
namespace LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.Before
{
    public class CashRegister
    {
        public CashRegister()
        {
            Tax = 0.06m;
        }
        private decimal Tax { get; set; }
        public void AcceptPayment(Customer customer, IEnumerable<Product> products, decimal payment)
        {
            decimal subTotal = 0m;
            foreach (Product product in products)
            {
                subTotal += product.Price;
            }
            foreach (Product product in products)
            {
                subTotal -= product.AvailableDiscounts;
            }
            decimal grandTotal = subTotal * Tax;
            customer.DeductFromAccountBalance(grandTotal);
        }
    }
    public class Customer
    {
        public void DeductFromAccountBalance(decimal amount)
        {
            // deduct from balance
        }
    }
    public class Product
    {
        public decimal Price { get; set; }
        public decimal AvailableDiscounts { get; set; }
    }
}
```

重构后的代码如下，我们把AcceptPayment的内部逻辑拆分成了CalculateSubtotal、SubtractDiscounts、AddTax、SubtractFromCustomerBalance四个功能明确且可读性更好的小方法。

```
using System.Collections.Generic;
namespace LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After
{
    public class CashRegister
    {
        public CashRegister()
        {
            Tax = 0.06m;
        }
        private decimal Tax { get; set; }
        private IEnumerable<Product> Products { get; set; }
        public void AcceptPayment(Customer customer, IEnumerable<Product> products, decimal payment)
        {
            decimal subTotal = CalculateSubtotal();
            subTotal = SubtractDiscounts(subTotal);
            decimal grandTotal = AddTax(subTotal);
            SubtractFromCustomerBalance(customer, grandTotal);
        }
        private void SubtractFromCustomerBalance(Customer customer, decimal grandTotal)
        {
            customer.DeductFromAccountBalance(grandTotal);
```

```
            customer.DeductFromAccountBalance(grandTotal);
        }
        private decimal AddTax(decimal subTotal)
        {
            return subTotal * Tax;
        }
        private decimal SubtractDiscounts(decimal subTotal)
        {
            foreach (Product product in Products)
            {
                subTotal -= product.AvailableDiscounts;
            }
            return subTotal;
        }
        private decimal CalculateSubtotal()
        {
            decimal subTotal = 0m;
            foreach (Product product in Products)
            {
                subTotal += product.Price;
            }
            return subTotal;
        }
    }
    public class Customer
    {
        public void DeductFromAccountBalance(decimal amount)
        {
            // deduct from balance
        }
    }
    public class Product
    {
        public decimal Price { get; set; }
        public decimal AvailableDiscounts { get; set; }
    }
}
```

总结：其实这个重构和我们前面讲的“提取方法”和“提取方法对象”如出一辙，尤其是“提取方法”，所以大家只要知道用这种思想重构就行。

23. 引入参数对象

概念：本文中的“引入参数对象”是指当一个方法的参数过多或者过为复杂时，可以考虑把这些参数封装成一个单独的类。

正文：如果一个方法所需要的参数大于5个，理解该方法的签名就变得比较困难，因为这样感觉参数很长、样式不好并且没有分类，所以我们有必要把参数进行封装。

```
namespace LosTechies.DaysOfRefactoring.SampleCode.ParameterObject.Before
{
    public class Registration
    {
        public void Create(decimal amount, Student student, IEnumerable<Course> courses, decimal credits)
        {
            // do work
        }
    }
}
```

通常这种情形下创建一个用户传递参数的类是很有帮助的，这会使得代码更容易明白也更灵活，因为当你需要增加参数时，只需要给参数类添加一个属性即可。请注意只有当你发现方法的参数比较多时才应该应用该重构，如果方法的参数比较少，就没有必要应用此重构，因为该重构会增加系统中类的数量，同时也会加大维护负担。所以要看参数情况而定。
重构后的代码如下：

```
using System.Collections.Generic;
namespace LosTechies.DaysOfRefactoring.SampleCode.ParameterObject.After
{
    public class RegistrationContext
    {
        public decimal Amount { get; set; }
        public Student Student { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public decimal Credits { get; set; }
    }
    public class Registration
    {
        public void Create(RegistrationContext registrationContext)
        {
            // do work
        }
    }
}
```

总结：这种重构很重要，尤其是当一个方法的参数比较多的时候，不管是大中型项目还是小型项目，都会遇到这种场景，所以建议大家多使用这个重构。这种封装的思想在SOA里面也经常运用到，封装输入Message，封装输出Message，消息来和消息去以及消息间的交互就构成了整个应用体系。

24. 分解复杂判断

概念：本文中的”分解复杂判断”是指把原来复杂的条件判断等语句用尽快返回等方式简化代码。

正文：简单的来说，当你的代码中有很深的嵌套条件时，花括号就会在代码中形成一个长长的箭头。我们经常在不同的代码中看到这种情况，并且这种情况也会扰乱代码的可读性。

如下代码所示，HasAccess方法里面包含一些嵌套条件，如果再加一些条件或者增加复杂度，那么代码就很可能出现几个问题：1，可读性差。 2，很容易出现异常。3，性能较差。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Text;
namespace LosTechies.DaysOfRefactoring.SampleCode.ArrowheadAntipattern.Before
{
    public class Security
    {
        public ISecurityChecker SecurityChecker { get; set; }
        public Security(ISecurityChecker securityChecker)
        {
            SecurityChecker = securityChecker;
        }
        public bool HasAccess(User user, Permission permission, IEnumerable<Permission> exemptions)
        {
            bool hasPermission = false;
            if (user != null)
            {
                if (permission != null)
                {
                    if (exemptions.Count() == 0)
                    {
                        if (SecurityChecker.CheckPermission(user, permission) || exemptions.Contains(permission))
                        {
                            hasPermission = true;
                        }
                    }
                }
            }
            return hasPermission;
        }
    }
}
```

那么重构上面的代码也很简单，如果有可能的话，尽量将条件从方法中移除，我们让代码在做处理任务之前先检查条件，如果条件不满足就尽快返回，不继续执行。下面是重构后的代码：

```
using System.Collections.Generic;
using System.Linq;
namespace LosTechies.DaysOfRefactoring.SampleCode.ArrowheadAntipattern.After
{
    public class Security
    {
        public ISecurityChecker SecurityChecker { get; set; }
        public Security(ISecurityChecker securityChecker)
        {
            SecurityChecker = securityChecker;
        }
        public bool HasAccess(User user, Permission permission, IEnumerable<Permission> exemptions)
        {
            if (user == null || permission == null)
                return false;
            if (exemptions.Contains(permission))
                return true;
            return SecurityChecker.CheckPermission(user, permission);
        }
    }
}
```

总结：这个重构很重要，它和后面讲的”尽快返回“有些类似，我们在做复杂的处理过程时，要经常考虑这个重构，用好了它，会对我们的帮助很大。

25. 引入契约式设计

概念：本文中的”引入契约式设计”是指我们应该对应该对输入和输出进行验证，以确保系统不会出现我们所想象不到的异常和得不到我们想要的结果。

正文：契约式设计规定方法应该对输入和输出进行验证，这样你便可以保证你得到的数据是可以工作的，一切都是按预期进行的，如果不是按预期进行，异常或是错误就应该被返回，下面我们举的例子中，我们方法中的参数可能会值为null的情况，在这种情况下由于我们没有验证，NullReferenceException异常会报出。另外在方法的结尾处我们也没有保证会返回一个正确的decimal值给调用方法的对象。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LosTechies.DaysOfRefactoring.SampleCode.Day25_DesignByContract
{
    public class CashRegister
    {
        public decimal TotalOrder(IEnumerable<Product> products, Customer customer)
        {
            decimal orderTotal = products.Sum(product => product.Price);

            customer.Balance += orderTotal;

            return orderTotal;
        }
    }
}
```

对上面的代码重构是很简单的，首先我们处理不会有一个null值的customer对象，检查我们最少会有一个product对象。在返回订单总和之前先确保我们会返回一个有意义

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Contracts;

namespace LosTechies.DaysOfRefactoring.SampleCode.DesignByContract.After
{
    public class CashRegister
    {
        public decimal TotalOrder(IEnumerable<Product> products, Customer customer)
```

```
        {
            if (customer == null)
                throw new ArgumentNullException("customer", "Customer cannot be null");
            if (products.Count() == 0)
                throw new ArgumentException("Must have at least one product to total", "products");

            decimal orderTotal = products.Sum(product => product.Price);

            customer.Balance += orderTotal;

            if (orderTotal == 0)
                throw new ArgumentOutOfRangeException("orderTotal", "Order Total should not be zero");

            return orderTotal;
        }
    }
}
```

上面的代码中添加了额外的代码来进行验证，虽然看起来代码复杂度增加了，但我认为这是非常值得做的，因为当NullReferenceException发生时去追查异常的详细信息真是很令人讨厌的事情。

总结：微软在处理代码乃至产品的时候，很喜欢应用此重构，你如果认真看它的代码库，认真看一下WCF的设计，就不难发现了。这个重构建议大家经常使用，这会增强整个系统的稳定性和健壮性。

26. 避免双重否定

概念：本文中的“避免双重否定”是指把代码中的双重否定语句修改成简单的肯定语句，这样即让代码可读，同时也给维护带来了方便。

正文：避免双重否定重构本身非常容易实现，但我们却在太多的代码中见过因为双重否定降低了代码的可读性以致于非常让人容易误解真正意图。存在双重否定的代码具有非常大的危害性，因为这种类型的代码容易引起错误的假设，错误的假设又会导致书写出错误的维护代码，最终会导致bug产生。具体可以看下面的代码：

```
using  System.Collections.Generic;
using  LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
namespace  LosTechies.DaysOfRefactoring.SampleCode.DoubleNegative.Before
{
    public class  Order
    {
        public void  Checkout(IEnumerable<Product> products, Customer  customer)
        {
            if  (!customer.IsNotFlagged)
            {
                // the customer account is flagged
                // log some errors and return
                return;
            }
            // normal order processing
        }
    }
    public class  Customer
    {
        public decimal  Balance { get; private set; }
        public bool  IsNotFlagged
        {
            get  { return  Balance < 30m; }
        }
    }
}
```

如上代码中的双重否定可读性非常低，因为我们很难搞明白双重否定的正确值。要重构它也非常容易，如下是重构后的代码：

```
using System.Collections.Generic;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
namespace LosTechies.DaysOfRefactoring.SampleCode.DoubleNegative.After
{
    public class Order
    {
        public void Checkout(IEnumerable<Product> products, Customer customer)
        {
            if (customer.IsFlagged)
            {
                // the customer account is flagged
                // log some errors and return
                return;
            }
            // normal order processing
        }
    }
    public class Customer
    {
        public decimal Balance { get; private set; }
        public bool IsFlagged
        {
            get { return Balance >= 30m; }
        }
    }
}
```

总结： “双重否定”很容易让人产生错误的判断，也很难让人理解你的代码，所以这个重构在我们的代码中是很重要的，尤其是在判断条件很多且业务复杂的时候。

27. 去除上帝类

概念：本文中的“去除上帝类”是指把一个看似功能很强几很难维护的类，按照职责把自己的属性或方法分派到各自的类中或分解成功能明确的类，从而去掉上帝类。

正文：我们经常可以在一些原来的代码中见到一些类明确违反了SRP原则（单一原则），这些类通常以“Utils”或“Manager”后缀结尾，但有时这些类也没有这些特征，它仅仅是多个类多个方法的组合。另一个关于上帝类的特征是通常这些类中的方法被用注释分隔为不同的分组。那么久而久之，这些类被转换为那些没有人愿意进行归并到合适类的方法的聚集地，对这些类进行重构是将类中的代码按照职责分派到各自的类中，这样就解除了上帝类，也减轻了维护的负担。

```
using System.Collections.Generic;
```

```
using System.Collections.Generic;
using LosTechies.DaysOfRefactoring.EncapsulateCollection.After;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
using Customer = LosTechies.DaysOfRefactoring.BreakResponsibilities.After.Customer;
namespace LosTechies.DaysOfRefactoring.SampleCode.RemoveGodClasses.Before
{
    public class CustomerService
    {
        public decimal CalculateOrderDiscount(IEnumerable<Product> products, Customer customer)
        {
            // do work
        }
        public bool CustomerIsValid(Customer customer, Order order)
        {
            // do work
        }
        public IEnumerable<string> GatherOrderErrors(IEnumerable<Product> products, Customer customer)
        {
            // do work
        }
        public void Register(Customer customer)
        {
            // do work
        }
        public void ForgotPassword(Customer customer)
        {
            // do work
        }
    }
}
```

我们看到要重构上面的代码是很简单的，只要将相关的方法按职责分派到对应的类中即可，带来的好处就是这会降低代码的颗粒度并减少未来维护代码的成本。下面是重构的代码按照职责分为了两个不同的类。

```
using System.Collections.Generic;
using LosTechies.DaysOfRefactoring.EncapsulateCollection.After;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
using Customer = LosTechies.DaysOfRefactoring.BreakResponsibilities.After.Customer;
namespace LosTechies.DaysOfRefactoring.SampleCode.RemoveGodClasses.After
{
    public class CustomerOrderService
    {
        public decimal CalculateOrderDiscount(IEnumerable<Product> products, Customer customer)
        {
            // do work
        }
        public bool CustomerIsValid(Customer customer, Order order)
        {
            // do work
        }
        public IEnumerable<string> GatherOrderErrors(IEnumerable<Product> products, Customer customer)
        {
            // do work
        }
    }
    public class CustomerRegistrationService
    {
        public void Register(Customer customer)
        {
            // do work
        }
        public void ForgotPassword(Customer customer)
        {
            // do work
        }
    }
}
```

总结： “去除上帝类”是我们经常容易造成的，第一是因为简便，看到有一个现成的类，大家都会喜欢把代码往里面写，最后导致越写越大，并且声明功能都有，这样即降低了可读性，也造成了维护的负担。

28. 为布尔方法命名

概念：本文中的“为布尔方法命名”是指如果一个方法带有大量的bool参数时，可以根据bool参数的数量，提取出若干个独立的方法来简化参数。

正文：我们现在要说的重构并不是普通字面意义上的重构，它有很多值得讨论的地方。当一个方法带有大量的bool参数时，会导致方法很容易被误解并产生非预期的行为，

根据布尔型参数的数量，我们可以决定提取出若干个独立的方法来。具体代码如下：

```
using LosTechies.DaysOfRefactoring.BreakResponsibilities.After;
namespace LosTechies.DaysOfRefactoring.SampleCode.RenameBooleanMethod.Before
{
    public class BankAccount
    {
        public void CreateAccount(Customer customer, bool withChecking, bool withSavings, bool withStocks)
        {
            // do work
        }
    }
}
```

我们可以将上面的bool参数以独立方法的形式暴露给调用端以提高代码的可读性，同时我们还需要将原来的方法改为private以限制其可访问性。显然我们关于要提取的独立方法会有一个很大的排列组合，这是一大缺点，所以我们可以考虑引入”参数对象“重构。

```
using LosTechies.DaysOfRefactoring.BreakResponsibilities.After;
```

```
using LosTechies.DaysOfRefactoring.BreakResponsibilities.After;
namespace LosTechies.DaysOfRefactoring.SampleCode.RenameBooleanMethod.After
{
    public class BankAccount
    {
        public void CreateAccountWithChecking(Customer customer)
        {
            CreateAccount(customer, true, false);
        }
        public void CreateAccountWithCheckingAndSavings(Customer customer)
        {
            CreateAccount(customer, true, true);
        }
        private void CreateAccount(Customer customer, bool withChecking, bool withSavings)
        {
            // do work
        }
    }
}
```

总结： “为布尔方法命名”这个重构在很多时候都不常用，如果用户的参数可枚举，我们一般会枚举它的值，不过使用这种重构也有好处，就是分解开来以后，方法多了，参数少了，代码维护起来方便了一些。

29. 去除中间人对象

概念：本文中的“去除中间人对象”是指把 在中间关联而不起任何其他作用的类移除，让有关系的两个类直接进行交互。

正文：有些时候在我们的代码会存在一些“幽灵类”，设计模式大师Fowler称它们为“中间人”类，“中间人”类除了调用别的对象之外不做任何事情，所以“中间人”类没有存在的必要，我们可以将它们从代码中删除，从而让交互的两个类直接关联。

如下代码所示，Consumer 类要得到AccountDataProvider 的数据，但中间介入了没起任何作用的 AccountManager 类来关联，所以我们应当移除。

```
using LosTechies.DaysOfRefactoring.PullUpField.After;

namespace LosTechies.DaysOfRefactoring.SampleCode.RemoveMiddleMan.Before
{
    public class Consumer
    {
        public AccountManager AccountManager { get; set; }

        public Consumer(AccountManager accountManager)
        {
            AccountManager = accountManager;
        }

        public void Get(int id)
        {
            Account account = AccountManager.GetAccount(id);
        }
    }

    public class AccountManager
    {
        public AccountDataProvider DataProvider { get; set; }

        public AccountManager(AccountDataProvider dataProvider)
        {
            DataProvider = dataProvider;
        }

        public Account GetAccount(int id)
        {
            return DataProvider.GetAccount(id);
        }
    }

    public class AccountDataProvider
    {
        public Account GetAccount(int id)
        {
            // get account
        }
    }
}
```

重构后的代码如下所示，Consumer 和AccountDataProvider 直接进行关联，这样代码就简单了。

```
using LosTechies.DaysOfRefactoring.PullUpField.After;

namespace LosTechies.DaysOfRefactoring.SampleCode.RemoveMiddleMan.After
{
    public class Consumer
    {
        public AccountDataProvider AccountDataProvider { get; set; }

        public Consumer(AccountDataProvider dataProvider)
        {
            AccountDataProvider = dataProvider;
        }

        public void Get(int id)
        {
            Account account = AccountDataProvider.GetAccount(id);
        }
    }

    public class AccountDataProvider
    {
```

```
    {
        public Account GetAccount(int id)
        {
            // get account
        }
    }
}
```

总结： “去除中间人对象”很多时候都会很有作用，尤其是在误用设计模式的代码中最容易见到，设计模式中的适配器模式和代理模式等都用中间的类是两者进行关联，这是比较合理的，因为中间类做了很多事情，而对于没有任何作用的中间类应该移除。

30. 尽快返回

概念： 本文中的“尽快返回”是指把原来复杂的条件判断等语句用尽快返回的方式简化代码。

正文： 如首先声明的是前面讲的“分解复杂判断”，简单的来说，当你的代码中有很深的嵌套条件时，花括号就会在代码中形成一个长长的箭头。我们经常在不同的代码中看到这种情况，并且这种情况也会扰乱代码的可读性。下代码所示，HasAccess方法里面包含一些嵌套条件，如果再加一些条件或者增加复杂度，那么代码就很可能出现几个问题：1，可读性差 2，很容易出现异常 3，性能较差

```
using System.Collections.Generic;
using System.Linq;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
using Customer = LosTechies.DaysOfRefactoring.BreakResponsibilities.After.Customer;
namespace LosTechies.DaysOfRefactoring.SampleCode.ReturnASAP.Before
{
    public class Order
    {
        public Customer Customer { get; private set; }
        public decimal CalculateOrder(Customer customer, IEnumerable<Product> products, decimal discounts)
        {
            Customer = customer;
            decimal orderTotal = 0m;
            if (products.Count() > 0)
            {
                orderTotal = products.Sum(p => p.Price);
                if (discounts > 0)
                {
                    orderTotal -= discounts;
                }
            }
            return orderTotal;
        }
    }
}
```

那么重构上面的代码也很简单，如果有可能的话，尽量将条件判断从方法中移除，我们让代码在做处理任务之前先检查条件，如果条件不满足就尽快返回，不继续执行。

下面是重构后的代码：

```
using System.Collections.Generic;
using System.Linq;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
using Customer = LosTechies.DaysOfRefactoring.BreakResponsibilities.After.Customer;
namespace LosTechies.DaysOfRefactoring.SampleCode.ReturnASAP.After
{
    public class Order
    {
        public Customer Customer { get; private set; }
        public decimal CalculateOrder(Customer customer, IEnumerable<Product> products, decimal discounts)
        {
            if (products.Count() == 0)
                return 0;
            Customer = customer;
            decimal orderTotal = products.Sum(p => p.Price);
            if (discounts == 0)
                return orderTotal;
            orderTotal -= discounts;
            return orderTotal;
        }
    }
}
```

总结： 总结：这个重构很重要，它和前面讲的“分解复杂判断”有些类似，我们在做复杂的处理过程时，要经常考虑这个重构，用好了它，会对我们的帮助很大。

31. 使用多态代替条件判断

概念： 本文中的“使用多态代替条件判断”是指如果你需要检查对象的类型或者根据类型执行一些操作时，一种很好的办法就是将算法封装到类中，并利用多态性进行抽象调用。

正文： 本文展示了面向对象编程的基础之一“多态性”，有时你需要检查对象的类型或者根据类型执行一些操作时，一种很好的办法就是将算法封装到类中，并利用多态性进行抽象调用。

如下代码所示，OrderProcessor 类的ProcessOrder方法根据Customer 的类型分别执行一些操作，正如上面所讲的那样，我们最好将OrderProcessor 类中这些算法（数据或操作）封装在特定的Customer 子类中。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
namespace LosTechies.DaysOfRefactoring.SampleCode.ReplaceWithPolymorphism.Before
{
    public abstract class Customer
    {
    }
    public class Employee : Customer
    {
    }
```

```
    public class NonEmployee : Customer
    {
    }
    public class OrderProcessor
    {
        public decimal ProcessOrder(Customer customer, IEnumerable<Product> products)
        {
            // do some processing of order
            decimal orderTotal = products.Sum(p => p.Price);
            Type customerType = customer.GetType();
            if (customerType == typeof(Employee))
            {
                orderTotal -= orderTotal * 0.15m;
            }
            else if (customerType == typeof(NonEmployee))
            {
                orderTotal -= orderTotal * 0.05m;
            }
            return orderTotal;
        }
    }
}
```

重构后的代码如下，每个Customer 子类都封装自己的算法，然后OrderProcessor 类的ProcessOrder方法的逻辑也变得简单并且清晰了。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using LosTechies.DaysOfRefactoring.SampleCode.BreakMethod.After;
namespace LosTechies.DaysOfRefactoring.SampleCode.ReplaceWithPolymorphism.After
{
    public abstract class Customer
    {
        public abstract decimal DiscountPercentage { get; }
    }
    public class Employee : Customer
    {
        public override decimal DiscountPercentage
        {
            get { return 0.15m; }
        }
    }
    public class NonEmployee : Customer
    {
        public override decimal DiscountPercentage
        {
            get { return 0.05m; }
        }
    }
    public class OrderProcessor
    {
        public decimal ProcessOrder(Customer customer, IEnumerable<Product> products)
        {
            // do some processing of order
            decimal orderTotal = products.Sum(p => p.Price);
            orderTotal -= orderTotal * customer.DiscountPercentage;
            return orderTotal;
        }
    }
}
```

总结： “使用多态代替条件判断”这个重构在很多时候会出现设计模式中（常见的工厂家族、策略模式等都可以看到它的影子），因为运用它可以省去很多的条件判断，同时也能简化代码、规范类和对象之间的职责。

代码下载

代码下载地址：http://github.com/schambers/days-of-refactoring