31 天重构速成

你必须知道的重构技巧

2009-11-01
Sean Chambers, Simone Chiaretta
麒麟.NET 译

Sean Chambers 在其博客中发表的 33 篇随笔系列：
http://www.lostechies.com/blogs/sean_chambers/archive/2009/07/31/31-days-of-refactoring.aspx

更多关于重构的内容参见 Martin Fowler 的网站：
http://refactoring.com

Simmone Chiaretta（即 CodeClimber）转换为 eBook：
http://codeclimber.net.nz/

麒麟.NET 将其翻译为中文：
http://www.cnblogs.com/kirinboy

目录

简介

重构是持续改进代码的基础。抵制重构将带来技术麻烦：忘记代码片段的功能、创建无法测试的代码等等。而有了重构，使用单元测试、共享代码以及更可靠的无 bug 的代码这些最佳实践就显得简单多了。

鉴于重构的重要性，我决定在整个 8 月份每天介绍一个重构。在开始之前，请允许我事先声明，尽管我试着对每个重构进行额外的描述和讨论，但我并不是在声明它们的所有权。

我介绍的大多数重构都可以在 Refactoring.com 中找到，有一些来自《代码大全（第 2 版）》，剩下的则是我自己经常使用或从其他网站找到的。我觉得注明每个重构的出处并不是重要的，因为你可以在网上不同的帖子或文章中找到名称类似的重构。

本着这一精神，我将在明天发布第一篇帖子并开始长达 31 天的重构马拉松之旅。希望你们能够享受重构并从中获益。

Refactoring Day 1 : Encapsulate Collection

在某些场景中，向类的使用者隐藏类中的完整集合是一个很好的做法，比如对集合的 add/remove 操作中包含其他的相关逻辑时。因此，以可迭代但不直接在集合上进行操作的方式来暴露集合，是个不错的主意。我们来看代码：

```
public class Order
{
    private int _orderTotal;

    private List<OrderLine> _orderLines;

    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);
    }
}
```

如你所见，我们对集合进行了封装，没有将 Add/Remove 方法暴露给类的使用者。在.NET Framework 中，有些类如 ReadOnlyCollection，会由于封装集合而产生不同的行为，但它们各自都有防止误解的说明。这是一个非常简单但却极具价值的重构，可以确保用户不会误用你暴露的集合，避免代码中的一些 bug。

Refactoring Day 2 : Move Method

今天的重构同样非常简单，以至于人们并不认为这是一个有价值的重构。迁移方法（Move Method），顾名思义就是将方法迁移到合适的位置。在开始重构前，我们先看看一下代码：

```
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; }
        }
    }
```

这里值得注意的是 BankAccount.CalculateInterest 方法。当一个方法被其他类使用比在它所在类中的使用还要频繁时，我们就需要使用迁移方法重构了——将方法迁移到更频繁地使用它的类中。由于依赖关系，该重构并不能应用于所有实例，但人们还是经常低估它的价值。

最终的代码应该是这样的：

```
    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 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;
        }
    }
```

够简单吧？

Refactoring Day 3 : Pull Up Method

上移方法（Pull Up Method）重构是将方法向继承链上层迁移的过程。用于一个方法被多个实现者使用时。

```
public abstract class Vehicle
{
    // other methods
}

public class Car : Vehicle
{
    public void Turn(Direction direction)
    {
        // code here
    }
}

public class Motorcycle : Vehicle
{
}

public enum Direction
{
    Left,
    Right
}
```

如你所见，目前只有 Car 类中包含 Turn 方法，但我们也希望在 Motorcycle 类中使用。因此，如果没有基类，我们就创建一个基类并将该方法“上移”到基类中，这样两个类就都可以使用 Turn 方法了。这样做唯一的缺点是扩充了基类的接口、增加了其复杂性，因此需谨慎使用。只有当一个以上的子类需要使用该方法时才需要进行迁移。如果滥用继承，系统将会很快崩溃。这时你应该使用组合代替继承。重构之后的代码如下：

```
public abstract class Vehicle
{
    public void Turn(Direction direction)
    {
        // code here
    }
}

public class Car : Vehicle
{
}

public class Motorcycle : Vehicle
```

```
{
}

public enum Direction
{
    Left,
    Right
}
```

Refactoring Day 4 : Push Down Method

昨天我们介绍了将方法迁移到基类以供多个子类使用的上移方法重构，今天我们来看看相反的操作。重构前的代码如下：

```
public abstract class Animal
{
    public void Bark()
    {
        // code to bark
    }
}

public class Dog : Animal
{
}

public class Cat : Animal
{
}
```

这里的基类有一个 Bark 方法。或许我们的猫咪们一时半会也没法学会汪汪叫（bark），因此 Cat 类中不再需要这个功能了。尽管基类不需要这个方法，但在显式处理 Dog 类时也许还需要，因此我们将 Bark 方法“下移”到 Dog 类中。这时，有必要评估 Animal 基类中是否还有其他行为。如果没有，则是一个将 Animal 抽象类转换成接口的好时机。因为契约中不需要任何代码，可以认为是一个标记接口。

```
public abstract class Animal
{
}

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

public class Cat : Animal
{
}
```

Refactoring Day 5 : Pull Up Field

今天我们来看看一个和上移方法十分类似的重构。我们处理的不是方法，而是字段。

```
public abstract class Account
{
}

public class CheckingAccount : Account
{
    private decimal _minimumCheckingBalance = 5m;
}

public class SavingsAccount : Account
{
    private decimal _minimumSavingsBalance = 5m;
}
```

在这个例子中，两个子类中包含重复的常量。为了提高复用性我们将字段上移到基类中，并简化其名称。

```
public abstract class Account
{
    protected decimal _minimumBalance = 5m;
}

public class CheckingAccount : Account
{
}

public class SavingsAccount : Account
{
}
```

Refactoring Day 6 : Push Down Field

与上移字段相反的重构是下移字段。同样，这也是一个无需多言的简单重构。

```
public abstract class Task
{
    protected string _resolution;
}

public class BugTask : Task
{
}

public class FeatureTask : Task
{
}
```

在这个例子中，基类中的一个字符串字段只被一个子类使用，因此可以进行下移。只要没有其他子类使用基类的字段时，就应该立即执行该重构。保留的时间越长，就越有可能不去重构而保持原样。

```
public abstract class Task
{
}

public class BugTask : Task
{
    private string _resolution;
}

public class FeatureTask : Task
{
}
```

Refactoring Day 7 : Rename(method,class,parameter)

这是我最常用也是最有用的重构之一。我们对方法/类/参数的命名往往不那么合适，以至于误导阅读者对于方法/类/参数功能的理解。这会造成阅读者的主观臆断，甚至引入 bug。这个重构看起来简单，但却十分重要。

```
public class Person
{
    public string FN { get; set; }

    public decimal ClcHrlyPR()
    {
        // code to calculate hourly payrate
        return 0m;
    }
}
```

如你所见，我们的类/方法/参数的名称十分晦涩难懂，可以理解为不同的含义。应用这个重构你只需随手将名称修改得更具描述性、更容易传达其含义即可。简单吧。

```
// Changed the class name to Employee
public class Employee
{
    public string FirstName { get; set; }

    public decimal CalculateHourlyPay()
    {
        // code to calculate hourly payrate
        return 0m;
    }
}
```

Refactoring Day 8 : Replace Inheritance with Delegation

继承的误用十分普遍。它只能用于逻辑环境，但却经常用于简化，这导致复杂的没有意义的继承层次。看下面的代码：

```
public class Sanitation
{
    public string WashHands()
    {
        return "Cleaned!";
    }
}

public class Child : Sanitation
{
}
```

在该例中，Child 并不是 Sanitation，因此这样的继承层次是毫无意义的。我们可以这样重构：在 Child 的构造函数里实现一个 Sanitation 实例，并将方法的调用委托给这个实例。如果你使用依赖注入，可以通过构造函数传递 Sanitation 实例，尽管在我看来还要向 IoC 容器注册模型是一种坏味道，但领会精神就可以了。继承**只能**用于严格的继承场景，并不是用来快速编写代码的工具。

```
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();
    }
```

```
}
```

Refactoring Day 9 : Extract Interface

今天我们来介绍一个常常被忽视的重构：提取接口。如果你发现多于一个类使用另外一个类的某些方法，引入接口解除这种依赖往往十分有用。该重构实现起来非常简单，并且能够享受到松耦合带来的好处。

```
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;
    }
}
```

在下面的代码中，你可以看到我提取出了消费者所使用的两个方法，并将其置于一个接口中。现在消费者不必关心和了解实现了这些方法的类。我们解除了消费者与实际实现之间的耦合，使其只依赖于我们创建的契约。

```
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;
    }
}
```

Refactoring Day 10 : Extract Method

今天我们要介绍的重构是提取方法。这个重构极其简单但却大有裨益。首先，将逻辑置于命名良好的方法内有助于提高代码的可读性。当方法的名称可以很好地描述这部分代码的功能时，可以有效地减少其他开发者的研究时间。假设越少，代码中的 bug 也就越少。重构之前的代码如下：

```
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;
    }
}
```

你会发现 CalculateGrandTotal 方法一共做了 3 件不同的事情：计算总额、折扣和发票税额。开发者为了搞清楚每个功能如何处理而不得不将代码从头看到尾。相比于此，向下面的代码那样将每个任务分解成单独的方法则要节省更多时间，也更具可读性：

```
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);

        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;
    }
}
```

Refactoring Day 11 : Switch to Strategy

今天的重构没有固定的形式，多年来我使用过不同的版本，并且我敢打赌不同的人也会有不同的版本。该重构适用于这样的场景：switch 语句块很大，并且会随时引入新的判断条件。这时，最好使用策略模式将每个条件封装到单独的类中。实现策略模式的方式是很多的。我在这里介绍的策略重构使用的是字典策略，这么做的好处是调用者不必修改原来的代码。

```
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;
        }
    }
}
```

要应用该重构，需将每个测试条件至于单独的类中，这些类实现了一个共同的接口。然后将枚举作为字典的键，这样就可以获取正确的实现，并执行其代码了。以后如果希望添加新的条件，只需添加新的实现类，并将其添加至 ShippingCalculations 字典中。正如前面说过的，**这不是实现策略模式的唯一方式**。我在这里将字体加粗显示，是因为肯定会有人在评论里指出这点：）用你觉得好用的方法。我用这种方式实现重构的好处是，不用修改客户端代码。所有的修改都在 ShippingInfo 类内部。

Jayme Davis 指出这种重构由于仍然需要在构造函数中进行绑定，所以只不过是增加了一些类而已，但如果绑定 IShippingCalculation 的策略可以置于 IoC 中，带来的好处还是很多的，它可以使你更灵活地捆绑策略。

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

    public enum State
    {
        Alaska,
        NewYork,
        Florida
    }

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

```
    public ShippingInfo()
    {
        ShippingCalculations = new Dictionary<State, IShippingCalculation>
        {
            { State.Alaska, new AlaskShippingCalculation() },
            { State.NewYork, new NewYorkShippingCalculation() },
            { State.Florida, new FloridaShippingCalculation() }
        };
    }

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

public interface IShippingCalculation
{
    decimal Calculate();
}

public class AlaskShippingCalculation : IShippingCalculation
{

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

public class NewYorkShippingCalculation : IShippingCalculation
{
    public decimal Calculate()
    {
        return 10m;
    }
}

public class FloridaShippingCalculation : IShippingCalculation
{
    public decimal Calculate()
    {
        return 3m;
    }
```

```
    }
}
```

为了使这个示例圆满，我们来看看在 ShippingInfo 构造函数中使用 Ninject 为 IoC 容器时如何进行绑定。需要更改的地方很多，主要是将 state 的枚举放在策略内部，以及 Ninject 向构造函数传递一个 IShippingInfo 的 IEnumerable 泛型。接下来我们使用策略类中的 state 属性创建字典，其余部分保持不变。（感谢 Nate Kohari 和 Jayme Davis）

```
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;
    }
}
```

Refactoring Day 12 : Break Dependencies

有些单元测试需要恰当的测试“缝隙”（test seam）来模拟/隔离一些不想被测试的部分。如果你正想在代码中引入这种单元测试，那么今天介绍的重构就十分有用。在这个例子中，我们的客户端代码使用一个静态类来实现功能。但当需要单元测试时，问题就来了。我们无法在单元测试中模拟静态类。解决的方法是使用一个接口将静态类包装起来，形成一个缝隙来切断与静态类之间的依赖。

```
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
    }
}
```

重构时我们所要做的就是引入一个接口和简单调用上面那个静态类的类。因此行为还是一样的，只是调用的方式产生了变化。这是一个不错的重构起始点，也是向代码添加单元测试的简单方式。

```
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
        }
    }
```

现在，我们可以在单元测试中将模拟的 IFeederService 传入 AnimalFeedingService 构造函数。测试成功后，我们可以将静态方法中的代码移植到 FeederService 类中，并删除静态类。

Refactoring Day 13 : Extract Method Object

今天的重构来自于 Martin Fowler 的重构目录。你可以在这里找到包含简介的原始文章。
在我看来，这是一个比较罕见的重构，但有时却终能派上用场。当你尝试进行提取方法的重构时，需要引入大量的方法。在一个方法中使用众多的本地变量有时会使代码变得丑陋。因此最好使用提取方法对象这个重构，将执行任务的逻辑分开。

```
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;
    }
}
```

我们通过构造函数，将返回计算结果的类的引用传递给包含多个计算方法的新建对象，或者向方法对象的构造函数中单独传递各个参数。如下面的代码：

```
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;
    }
}
```

Refactoring Day 14 : Break Responsibilities

把一个类的多个职责进行拆分，这贯彻了 SOLID 中的单一职责原则（SRP）。尽管对于如何划分“职责”经常存在争论，但应用这项重构还是十分简单的。我这里并不会回答划分职责的问题，只是演示一个结构清晰的示例，将类划分为多个负责具体职责的类。

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

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

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

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

如你所见，Video 类包含两个职责，一个负责处理录像租赁，另一个负责管理管理用户的租赁总数。要分离职责，我们可以将用户的逻辑转移到用户类中。

```
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();
    }
}
```

Refactoring Day 15 : Remove Duplication

这大概是处理一个方法在多处使用时最常见的重构。如果不加以注意的话，你会慢慢地养成重复的习惯。开发者常常由于懒惰或者在想要尽快生成尽可能多的代码时，向代码中添加很多重复的内容。我想也没必要过多解释了吧，直接看代码把。

```
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;
    }
}
```

我们用共享方法的方式来删除重复的代码。看！没有重复了吧？请务必在必要的时候执行这项重构。它能有效地减少 bug，因为你不会将有 bug 的代码复制/粘贴到各个角落。

```
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;
```

```
    }
}
```

Refactoring Day 16 : Encapsulate Conditional

当代码中充斥着若干条件判断时，代码的真正意图会迷失于这些条件判断之中。这时我喜欢将条件判断提取到一个易于读取的属性或方法（如果有参数）中。重构之前的代码如下：

```
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";
    }
}
```

重构之后，代码的可读性更强，意图更明显：

```
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";
    }
}
```

Refactoring Day 17 : Extract Superclass

今天的重构来自于 Martin Fowler 的重构目录，其原始介绍在此。
当一个类有很多方法希望将它们“提拔”到基类以供同层次的其他类使用时，会经常使用该重构。下面的类包含两个方法，我们希望提取这两个方法并允许其他类使用。

```
public class Dog
{
    public void EatFood()
    {
        // eat some food
    }

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

重构之后，我们仅仅将需要的方法转移到了一个新的基类中。这很类似“Pull Up”重构，只是在重构之前，并不存在基类。

```
public class Animal
{
    public void EatFood()
    {
        // eat some food
    }

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

public class Dog : Animal
{
}
```

Refactoring Day 18 : Replace exception with conditional

今天的重构没有什么出处，是我平时经常使用而总结出来的。欢迎您发表任何改进意见或建议。我相信一定还有其他比较好的重构可以解决类似的问题。
我曾无数次面对的一个代码坏味道就是，使用异常来控制程序流程。您可能会看到类似的代码：

```
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;
    }
}
```

异常应该仅仅完成自己的本职工作：处理异常行为。大多数情况你都可以将这些代码用恰当的条件判断替换，并进行恰当的处理。下面的代码可以称之为契约式设计，因为我们在执行具体工作之前明确了 Motor 类的状态，而不是通过异常来进行处理。

```
public class Microwave
{
    private IMicrowaveMotor Motor { get; set; }

    public bool Start(object food)
    {
        if (Motor.IsInUse)
            return false;

        Motor.Cook(food);

        return true;
```

```
    }
}
```

Refactoring Day 19 : Extract Factory Class

今天的重构是由 GangOfFour 首先提出的，网络上有很多关于该模式不同用法的资源。在 GoF 网站上的这里和这里有关于工厂模式的两个不同用法。
在代码中，通常需要一些复杂的对象创建工作，以使这些对象达到一种可以使用的状态。通常情况下，这种创建不过是新建对象实例，并以我们需要的方式进行工作。但是，有时候这种创建对象的需求会极具增长，并且混淆了创建对象的原始代码。这时，工厂类就派上用场了。关于工厂模式更全面的描述可以参考这里。最复杂的工厂模式是使用抽象工厂创建对象族。而我们只是使用最基本的方式，用一个工厂类创建一个特殊类的实例。来看下面的代码：

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

        return policeCar;
    }
}
```

如您所见，New 方法负责创建 PoliceCar 并根据一些外部输入初始化 PoliceCar 的某些属性。对于简单的创建工作来说，这样做可以从容应对。但是久而久之，创建的工作量越来越大，并且被附加在 controller 类上，但这并不是 controller 类的职责。这时，我们可以将创建代码提取到一个 Factory 类中去，由该类负责 PoliceCar 实例的创建。

```
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);
        }
    }
```

由于将创建的逻辑转移到了工厂中，我们可以添加一个类来专门负责实例的创建，而不必担心在创建或复制代码的过程中有所遗漏。

Refactoring Day 20 : Extract Subclass

今天的重构来自于 Martin Fowler 的模式目录。你可以在他的目录中找到该重构。
当一个类中的某些方法并不是面向所有的类时，可以使用该重构将其迁移到子类中。我这里举的例子十分简单，它包含一个 Registration 类，该类处理与学生注册课程相关的所有信息。

```
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; }
}
```

当使用了该类之后，我们就会意识到问题所在——它应用于两个完全不同的场景。属性 NonRegistrationAction 和 Notes 只有在处理与普通注册略有不同的 NonRegistration 时才会使用。因此，我们可以提取一个子类，并将这两个属性转移到 NonRegistration 类中，这样才更合适。

```
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; }
}
```

Refactoring Day 21 : Collapse Hierarchy

今天的重构来自于 Martin Fowler 的模式目录。你可以在他的目录中找到该重构。
昨天，我们通过提取子类来下放职责。而今天，当我们意识到不再需要某个子类时，可以使用 Collapse Hierarchy 重构。如果某个子类的属性（以及其他成员）可以被合并到基类中，这时再保留这个子类已经没有任何意义了。

```
public class Website
{
    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; }
}
```

这里的子类并没有过多的功能，只是表示站点是否激活。这时我们会意识到判断站点是否激活的功能应该是通用的。因此可以将子类的功能放回到 Website 中，并删除 StudentWebsite 类型。

```
public class Website
{
    public string Title { get; set; }
    public string Description { get; set; }
    public IEnumerable<Webpage> Pages { get; set; }
    public bool IsActive { get; set; }
}
```

Refactoring Day 22 : Break Method

今天的重构没有任何出处。可能已经有其他人使用了相同的重构，只是名称不同罢了。如果你知道谁的名字比 Break Method 更好，请转告我。
这个重构是一种元重构（meta-refactoring），它只是不停地使用提取方法重构，直到将一个大的方法分解成若干个小的方法。下面的例子有点做作，AcceptPayment 方法没有丰富的功能。因此为了使其更接近真实场景，我们只能假设该方法中包含了其他大量的辅助代码。
下面的 AcceptPayment 方法可以被划分为多个单独的方法。

```
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; }
}
```

如您所见，AcceptPayment 方法包含多个功能，可以被分解为多个子方法。因此我们多次使用提取方法重构，结果如下：

```
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);
    }

    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; }
    }
}
```

Refactoring Day 23 : Introduce Parameter Object

该重构来自于 Fowler 的重构目录，参见这里。
有时当使用一个包含多个参数的方法时，由于参数过多会导致可读性严重下降，如：

```
public void Create(decimal amount, Student student,
    IEnumerable<Course> courses, decimal credits)
{
    // do work
}
```

这时有必要新建一个类，负责携带方法的参数。如果要增加更多的参数，只需为对参数对象增加其他的字段就可以了，代码显得更加灵活。要注意，仅仅在方法的参数确实过多时才使用该重构，否则会使类的数量暴增，而这本应该越少越好。

```
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
    }
}
```

Refactoring Day 24 : Remove Arrowhead Antipattern

今天的重构基于 c2 的 wiki 条目。Los Techies 的 Chris Missal 同样也些了一篇关于反模式的 post。
简单地说，当你使用大量的嵌套条件判断时，形成了箭头型的代码，这就是箭头反模式（arrowhead antipattern）。我经常在不同的代码库中看到这种现象，这提高了代码的圈复杂度（cyclomatic complexity）。
下面的例子演示了箭头反模式：

```
    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;
        }
    }
```

移除箭头反模式的重构和封装条件判断一样简单。这种方式的重构在方法执行之前往往会评估各个条件，这有点类似于契约式设计验证。下面是重构之后的代码：

```
    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);
        }
    }
```

如你所见，该方法大大整价了可读性和以后的可维护性。不难看出，该方法的所有可能的路径都会经过验证。

Refactoring Day 25 : Introduce Design By Contract checks

契约式设计（DBC，Design By Contract）定义了方法应该包含输入和输出验证。因此，可以确保所有的工作都是基于可用的数据，并且所有的行为都是可预料的。否则，将返回异常或错误并在方法中进行处理。要了解更多关于 DBC 的内容，可以访问 wikipedia。
在我们的示例中，输入参数很可能为 null。由于没有进行验证，该方法最终会抛出 NullReferenceException。在方法最后，我们也并不确定是否为用户返回了一个有效的 decimal，这可能导致在别的地方引入其他方法。

```
public class CashRegister
{
    public decimal TotalOrder(IEnumerable<Product> products, Customer customer)
    {
        decimal orderTotal = products.Sum(product => product.Price);

        customer.Balance += orderTotal;

        return orderTotal;
    }
}
```

在此处引入 DBC 验证是十分简单的。首先，我们要声明 customer 不能为 null，并且在计算总值时至少要有一个 product。在返回订单总值时，我们要确定其值是否有效。如果此例中任何一个验证失败，我们将以友好的方式抛出相应的异常来描述具体信息，而不是抛出一个晦涩的 NullReferenceException。
在.NET Framework 3.5 的 Microsoft.Contracts 命名空间中包含一些 DBC 框架和异常。我个人还没有使用，但它们还是值得一看的。关于该命名空间只有在 MSDN 上能找到点资料。

```
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;
```

```
        }
    }
```

在验证过程中确实增加了不少代码，你也许会认为过度使用了 DBC。但我认为在大多数情况下，处理这些棘手的问题所做的努力都是值得的。追踪无详细内容的 NullReferenceException 的确不是什么美差。

Refactoring Day 26 : Remove Double Negative

今天的重构来自于 Fowler 的重构目录，参见这里。
尽管我在很多代码中发现了这种严重降低可读性并往往传达错误意图的坏味道，但这种重构本身还是很容易实现的。这种毁灭性的代码所基于的假设导致了错误的代码编写习惯，并最终导致 bug。如下例所示：

```
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; }
    }
}
```

如你所见，这里的双重否定十分难以理解，我们不得不找出什么才是双重否定所要表达的肯定状态。修改代码是很容易的。如果我们找不到肯定的判断，可以添加一个处理双重否定的假设，而不要在得到结果之后再去验证。

```
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; }
    }
}
```

Refactoring Day 27 : Remove God Classes

在传统的代码库中，我们常常会看到一些违反了 SRP 原则的类。这些类通常以 Utils 或 Manager 结尾，有时也没有这么明显的特征而仅仅是普通的包含多个功能的类。这种 God 类还有一个特征，使用语句或注释将代码分隔为多个不同角色的分组，而这些角色正是这一个类所扮演的。
久而久之，这些类成为了那些没有时间放置到恰当类中的方法的垃圾桶。这时的重构需要将方法分解成多个负责单一职责的类。

```
    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
        }
    }
```

使用该重构是非常简单明了的，只需把相关方法提取出来并放置到负责相应职责的类中即可。这使得类的粒度更细、职责更分明、日后的维护更方便。上例的代码最终被分解为两个类：

```
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
        }
    }
```

Refactoring Day 28 : Rename boolean method

今天的重构不是来自于 Fowler 的重构目录。如果谁知道这项“重构”的确切出处，请告诉我。
当然，你也可以说这并不是一个真正的重构，因为方法实际上改变了，但这是一个灰色地带，可以开放讨论。一个拥有大量布尔类型参数的方法将很快变得无法控制，产生难以预期的行为。参数的数量将决定分解的方法的数量。来看看该重构是如何开始的：

```
public class BankAccount
{
    public void CreateAccount(Customer customer, bool withChecking, bool withSavings, bool
withStocks)
    {
        // do work
    }
}
```

要想使这样的代码运行得更好，我们可以通过命名良好的方法暴露布尔参数，并将原始方法更改为 private 以阻止外部调用。显然，你可能需要进行大量的代码转移，也许重构为一个 Parameter Object 会更有意义。

```
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
    }
}
```

Refactoring Day 29 : Remove Middle Man

今天的重构来自于 Fowler 的重构目录，见这里。
有时你的代码里可能会存在一些“Phantom”或“Ghost”类，Fowler 称之为“中间人（Middle Man）”。这些中间人类仅仅简单地将调用委托给其他组件，除此之外没有任何功能。
这一层是完全没有必要的，我们可以不费吹灰之力将其完全移除。

```
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
    }
}
```

最终结果已经足够简单了。我们只需要移除中间人对象，将原始调用指向实际的接收者。

```
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
    }
}
```

Refactoring Day 30 : Return ASAP

该话题实际上是诞生于移除箭头反模式重构之中。在移除箭头时，它被认为是重构产生的副作用。为了消除箭头，你需要尽快地 return。

```
    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;
        }
    }
```

该重构的理念就是，当你知道应该处理什么并且拥有全部需要的信息之后，立即退出所在方法，不再继续执行。

```
        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;
    }
}
```

Refactoring Day 31 : Replace conditional with Polymorphism

最后一天的重构来自于 Fowler 的重构目录，参见这里。
多态（Polymorphism）是面向对象编程的基本概念之一。在这里，是指在进行类型检查和执行某些类型操作时，最好将算法封装在类中，并且使用多态来对代码中的调用进行抽象。

```
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;
    }
}
```

如你所见，我们没有利用已有的继承层次进行计算，而是使用了违反 SRP 原则的执行方式。要进行重构，我们只需将百分率的计算置于实际的 customer 类型之中。我知道这只是一项补救措施，但我还是会这么做，就像在代码中那样。

```
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;
    }
}
```

附录 A

本电子书的代码示例可在 Sean 的 GitHub 资源库中下载：
http://github.com/schambers/days-of-refactoring