31 Days of Refactoring

Useful refactoring techniques you have to know

October 2009
Sean Chambers, Simone Chiaretta

Originally released as a series of 33 blog posts on Sean Chambers blog:

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

Majority of Refactorings courtesy of Martin Fowler via refactoring.com:

http://refactoring.com/

Converted to eBook by Simone Chiaretta (aka CodeClimber)

http://codeclimber.net.nz/

Contents

Introduction

Refactoring is an integral part of continually improving your code while it moves forward through time. Without refactoring you accrue technical debt, forget what portions of code do and create code that is resistant to any form of testing. It is an easy concept to get started with and opens the door to much better practices such as unit testing, shared code ownership and more reliable, bug-free code in general.

Because of the importance of refactoring, throughout the month of August I will be describing one refactoring a day for the 31 days of August. Before I begin, let me prefix this series with the fact that I am not claiming ownership of the refactorings I will describe, although I will try to bring some additional description and perhaps some discussion around each. I have chosen not to list the full set of refactorings on this first post until I actually post the refactoring. So as the month progresses I will update this post with links to each of the refactorings.

First on the list is credit where it is due. The majority of these refactorings can be found Refactoring.com, some are from Code Complete 2nd Edition and others are refactorings that I find myself doing often and from various other places on the interwebs. I don't think its important to note on each refactoring where it came from, as you can find refactorings of similar names found on various blogs and articles online.

On that note, I will be publishing the first post starting tomorrow that begins the 31 day marathon of various refactorings. I hope you all enjoy and learn something from the refactorings!

Refactoring Day 1 : Encapsulate Collection

In certain scenarios it is beneficial to not expose a full collection to consumers of a class. Some of these circumstances is when there is additional logic associated with adding/removing items from a collection. Because of this reason, it is a good idea to only expose the collection as something you can iterate over without modifying the collection. Let's take a look at some code

```
 1: public class Order
 2: {
 3:     private List<OrderLine> _orderLines;
 4:
 5:     public IEnumerable<OrderLine> OrderLines
 6:     {
 7:         get { return _orderLines; }
 8:     }
 9:
10:     public void AddOrderLine(OrderLine orderLine)
11:     {
12:         _orderTotal += orderLine.Total;
13:         _orderLines.Add(orderLine);
14:     }
15:
16:     public void RemoveOrderLine(OrderLine orderLine)
17:     {
18:         orderLine = _orderLines.Find(o => o == orderLine);
19:         if (orderLine == null) return;
20:
21:         _orderTotal -= orderLine.Total
22:         _orderLines.Remove(orderLine);
23:     }
24: }
```

As you can see, we have encapsulated the collection as to not expose the Add/Remove methods to consumers of this class. There is some other types in the .Net framework that will produce different behavior for encapsulating a collection such as ReadOnlyCollection but they do have different caveats with each. This is a very straightforward refactoring and one worth noting. Using this can ensure that consumers do not mis-use your collection and introduce bugs into the code.

Refactoring Day 2 : Move Method

The refactoring today is pretty straightforward, although often overlooked and ignored as being a worthwhile refactoring. Move method does exactly what it sounds like, move a method to a better location. Let's look at the following code before our refactoring:

```
 1: public class BankAccount
 2: {
 3:     public BankAccount(int accountAge, int creditScore,
                           AccountInterest accountInterest)
 4:     {
 5:         AccountAge = accountAge;
 6:         CreditScore = creditScore;
 7:         AccountInterest = accountInterest;
 8:     }
 9:
10:     public int AccountAge { get; private set; }
11:     public int CreditScore { get; private set; }
12:     public AccountInterest AccountInterest { get; private set; }
13:
14:     public double CalculateInterestRate()
15:     {
16:         if (CreditScore > 800)
17:             return 0.02;
18:
19:         if (AccountAge > 10)
20:             return 0.03;
21:
22:         return 0.05;
23:     }
24: }
25:
26: public class AccountInterest
27: {
28:     public BankAccount Account { get; private set; }
29:
30:     public AccountInterest(BankAccount account)
31:     {
32:         Account = account;
33:     }
34:
35:     public double InterestRate
36:     {
37:         get { return Account.CalculateInterestRate(); }
38:     }
39:
40:     public bool IntroductoryRate
41:     {
42:         get { return Account.CalculateInterestRate() < 0.05; }
43:     }
44: }
```

The point of interest here is the BankAccount.CalculateInterest method. A hint that you need the Move Method refactoring is when another class is using a method more often then the class in which it lives. If this is the case it makes sense to move the method to the class where it is primarily used. This doesn't work in every instance because of dependencies, but it is overlooked often as a worthwhile change.

In the end you would end up with something like this:

```
1: public class BankAccount
2: {
3:     public BankAccount(int accountAge, int creditScore,
```

```
                                AccountInterest accountInterest)
 4:     {
 5:         AccountAge = accountAge;
 6:         CreditScore = creditScore;
 7:         AccountInterest = accountInterest;
 8:     }
 9:
10:     public int AccountAge { get; private set; }
11:     public int CreditScore { get; private set; }
12:     public AccountInterest AccountInterest { get; private set; }
13: }
14:
15: public class AccountInterest
16: {
17:     public BankAccount Account { get; private set; }
18:
19:     public AccountInterest(BankAccount account)
20:     {
21:         Account = account;
22:     }
23:
24:     public double InterestRate
25:     {
26:         get { return CalculateInterestRate(); }
27:     }
28:
29:     public bool IntroductoryRate
30:     {
31:         get { return CalculateInterestRate() < 0.05; }
32:     }
33:
34:     public double CalculateInterestRate()
35:     {
36:         if (Account.CreditScore > 800)
37:             return 0.02;
38:
39:         if (Account.AccountAge > 10)
40:             return 0.03;
41:
42:         return 0.05;
43:     }
44: }
```

Simple enough!

Refactoring Day 3 : Pull Up Method

The Pull Up Method refactoring is the process of taking a method and “Pulling” it up in the inheritance chain. This is used when a method needs to be used by multiple implementers.

```
 1: public abstract class Vehicle
 2: {
 3:     // other methods
 4: }
 5:
 6: public class Car : Vehicle
 7: {
 8:     public void Turn(Direction direction)
 9:     {
10:         // code here
11:     }
12: }
13:
14: public class Motorcycle : Vehicle
15: {
16: }
17:
18: public enum Direction
19: {
20:     Left,
21:     Right
22: }
```

As you can see, our Turn method is currently only available to the car class, we also want to use it in the motorcycle class so we create a base class if one doesn’t already exist and “pull up” the method into the base class making it available to both. The only drawback is we have increased surface area of the base class adding to it’s complexity so use wisely. Only place methods that need to be used by more that one derived class. Once you start overusing inheritance it breaks down pretty quickly and you should start to lean towards composition over inheritance. Here is the code after the refactoring:

```
 1: public abstract class Vehicle
 2: {
 3:     public void Turn(Direction direction)
 4:     {
 5:         // code here
 6:     }
 7: }
 8:
 9: public class Car : Vehicle
10: {
11: }
12:
13: public class Motorcycle : Vehicle
14: {
15: }
16:
17: public enum Direction
18: {
19:     Left,
20:     Right
21: }
```

Refactoring Day 4 : Push Down Method

Yesterday we looked at the pull up refactoring to move a method to a base class so mutiple derived classes can use a method. Today we look at the opposite. Here is the code before the refactoring:

```
 1: public abstract class Animal
 2: {
 3:     public void Bark()
 4:     {
 5:         // code to bark
 6:     }
 7: }
 8:
 9: public class Dog : Animal
10: {
11: }
12:
13: public class Cat : Animal
14: {
15: }
```

So here we have some code with a base class that has a Bark method. Perhaps at one time our cat could bark, but now we no longer need that functionality on the Cat class. So we “Push Down” the Bark method into the Dog class as it is no longer needed on the base class but perhaps it is still needed when dealing explicitly with a Dog. At this time, it’s worthwhile to evaluate if there is any behavior still located on the Animal base class. If not, it is a good opportunity to turn the Animal abstract class into an interface instead as no code is required on the contract and can be treated as a marker interface.

```
 1: public abstract class Animal
 2: {
 3: }
 4:
 5: public class Dog : Animal
 6: {
 7:     public void Bark()
 8:     {
 9:         // code to bark
10:     }
11: }
12:
13: public class Cat : Animal
14: {
15: }
```

Refactoring Day 5 : Pull Up Field

Today we look at a refactoring that is similar to the Pull Up method. Instead of a method, it is obviously done with a field instead!

```
 1: public abstract class Account
 2: {
 3: }
 4:
 5: public class CheckingAccount : Account
 6: {
 7:     private decimal _minimumCheckingBalance = 5m;
 8: }
 9:
10: public class SavingsAccount : Account
11: {
12:     private decimal _minimumSavingsBalance = 5m;
13: }
```

In this example, we have a constant value that is duplicated between two derived classes. To promote reuse we can pull up the field into the base class and rename it for brevity.

```
 1: public abstract class Account
 2: {
 3:     protected decimal _minimumBalance = 5m;
 4: }
 5:
 6: public class CheckingAccount : Account
 7: {
 8: }
 9:
10: public class SavingsAccount : Account
11: {
12: }
```

Refactoring Day 6 : Push Down Field

Opposite of the Pull Up Field refactoring is push down field. Again, this is a pretty straight forward refactoring without much description needed

```
 1: public abstract class Task
 2: {
 3:     protected string _resolution;
 4: }
 5:
 6: public class BugTask : Task
 7: {
 8: }
 9:
10: public class FeatureTask : Task
11: {
12: }
```

In this example, we have a string field that is only used by one derived class, and thus can be pushed down as no other classes are using it. It's important to do this refactoring at the moment the base field is no longer used by other derived classes. The longer it sits the more prone it is for someone to simply not touch the field and leave it be.

```
 1: public abstract class Task
 2: {
 3: }
 4:
 5: public class BugTask : Task
 6: {
 7:     private string _resolution;
 8: }
 9:
10: public class FeatureTask : Task
11: {
12: }
```

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

This refactoring I use most often and is one of the most useful refactoring. All too often we do not name methods/classes/parameters properly that leads to a misunderstanding as to what the method/class/parameter's function is. When this occurs, assumptions are made and bugs are introduced to the system. As simple of a refactoring this seems, it is one of the most important to leverage.

```
 1: public class Person
 2: {
 3:     public string FN { get; set; }
 4:
 5:     public decimal ClcHrlyPR()
 6:     {
 7:         // code to calculate hourly payrate
 8:         return 0m;
 9:     }
10: }
```

As you can see here, we have a class/method/parameter that all have very non-descriptive, obscure names. They can be interpreted in a number of different ways. Applying this refactoring is as simple as renaming the items at hand to be more descriptive and convey what exactly they do. Simple enough.

```
 1: // Changed the class name to Employee
 2: public class Employee
 3: {
 4:     public string FirstName { get; set; }
 5:
 6:     public decimal CalculateHourlyPay()
 7:     {
 8:         // code to calculate hourly payrate
 9:         return 0m;
10:     }
11: }
```

Refactoring Day 8 : Replace Inheritance with Delegation

All too often inheritance is used in the wrong scenarios. Inheritance should only be used in *logical* circumstances but it is often used for convenience purposes. I've seen this many times and it leads to complex inheritance hierarchies that don't make sense. Take the following code:

```
 1: public class Sanitation
 2: {
 3:     public string WashHands()
 4:     {
 5:         return "Cleaned!";
 6:     }
 7: }
 8:
 9: public class Child : Sanitation
10: {
11: }
```

In this instance, a Child is not a "Sanitation" and therefore doesn't make sense as an inheritance hierarchy. We can refactor by initializing an instance of Sanitation in the Child constructor and delegating the call to the class rather than via inheritance. If you were using Dependency Injection, you would pass in the Sanitation instance via the constructor, although then you would need to register your model in your IoC container which is a smell IMO, you get the idea though. Inheritance should ONLY be used for scenarios where inheritance is warranted. Not instances where it makes it quicker to throw down code.

```
 1: public class Sanitation
 2: {
 3:     public string WashHands()
 4:     {
 5:         return "Cleaned!";
 6:     }
 7: }
 8:
 9: public class Child
10: {
11:     private Sanitation Sanitation { get; set; }
12:
13:     public Child()
14:     {
15:         Sanitation = new Sanitation();
16:     }
17:
18:     public string WashHands()
19:     {
20:         return Sanitation.WashHands();
21:     }
22: }
```

Refactoring Day 9 : Extract Interface

Today we look at an often overlooked refactoring. Extract Interface. When you notice more than one class using a similar subset of methods on a class, it is useful to break the dependency and introduce an interface that the consumers to use. It's easy to implement and has benefits from loose coupling.

```
 1: public class ClassRegistration
 2: {
 3:     public void Create()
 4:     {
 5:         // create registration code
 6:     }
 7:
 8:     public void Transfer()
 9:     {
10:         // class transfer code
11:     }
12:
13:     public decimal Total { get; private set; }
14: }
15:
16: public class RegistrationProcessor
17: {
18:     public decimal ProcessRegistration(ClassRegistration registration)
19:     {
20:         registration.Create();
21:         return registration.Total;
22:     }
23: }
```

In the after example, you can see we extracted the methods that both consumers use and placed them in an interface. Now the consumers don't care/know about the class that is implementing these methods. We have decoupled our consumer from the actual implementation and depend only on the contract that we have created.

```
 1: public interface IClassRegistration
 2: {
 3:     void Create();
 4:     decimal Total { get; }
 5: }
 6:
 7: public class ClassRegistration : IClassRegistration
 8: {
 9:     public void Create()
10:     {
11:         // create registration code
12:     }
13:
14:     public void Transfer()
15:     {
16:         // class transfer code
17:     }
18:
19:     public decimal Total { get; private set; }
20: }
21:
22: public class RegistrationProcessor
23: {
24:     public decimal ProcessRegistration(IClassRegistration registration)
25:     {
26:         registration.Create();
```

```
27:         return registration.Total;
28:     }
29: }
```

Refactoring Day 10 : Extract Method

Today we look at the Extract Method refactoring. This is an extremely easy refactoring with several benefits. First, it helps to make code more readable by placing logic behind descriptive method names. This reduces the amount of investigation the next developer needs to do as a method name can describe what a portion of code is doing. This in turn reduces bugs in the code because less assumptions need to be made. Here's some code before we apply the refactoring:

```
 1: public class Receipt
 2: {
 3:     private IList<decimal> Discounts { get; set; }
 4:     private IList<decimal> ItemTotals { get; set; }
 5:
 6:     public decimal CalculateGrandTotal()
 7:     {
 8:         decimal subTotal = 0m;
 9:         foreach (decimal itemTotal in ItemTotals)
10:             subTotal += itemTotal;
11:
12:         if (Discounts.Count > 0)
13:         {
14:             foreach (decimal discount in Discounts)
15:                 subTotal -= discount;
16:         }
17:
18:         decimal tax = subTotal * 0.065m;
19:
20:         subTotal += tax;
21:
22:         return subTotal;
23:     }
24: }
```

You can see that the CalculateGrandTotal method is actually doing three different things here. It's calculating the subtotal, applying any discounts and then calculating the tax for the receipt. Instead of making a developer look through that whole method to determine what each thing is doing, it would save time and readability to seperate those distinct tasks into their own methods like so:

```
 1: public class Receipt
 2: {
 3:     private IList<decimal> Discounts { get; set; }
 4:     private IList<decimal> ItemTotals { get; set; }
 5:
 6:     public decimal CalculateGrandTotal()
 7:     {
 8:         decimal subTotal = CalculateSubTotal();
 9:
10:         subTotal = CalculateDiscounts(subTotal);
11:
12:         subTotal = CalculateTax(subTotal);
13:
14:         return subTotal;
15:     }
16:
17:     private decimal CalculateTax(decimal subTotal)
18:     {
19:         decimal tax = subTotal * 0.065m;
20:
21:         subTotal += tax;
22:         return subTotal;
```

```
23:     }
24:
25:     private decimal CalculateDiscounts(decimal subTotal)
26:     {
27:         if (Discounts.Count > 0)
28:         {
29:             foreach (decimal discount in Discounts)
30:                 subTotal -= discount;
31:         }
32:         return subTotal;
33:     }
34:
35:     private decimal CalculateSubTotal()
36:     {
37:         decimal subTotal = 0m;
38:         foreach (decimal itemTotal in ItemTotals)
39:             subTotal += itemTotal;
40:         return subTotal;
41:     }
42: }
```

Refactoring Day 11 : Switch to Strategy

Todays refactoring doesn't come from any one source, rather I've used different versions over the years and I'm sure other have different variations of the same aim.

This refactoring is used when you have a larger switch statement that continually changes because of new conditions being added. In these cases it’s often better to introduce the strategy pattern and encapsulate each condition in it’s own class. The strategy refactoring I’m showing here is refactoring towards a dictionary strategy. There is several ways to implement the strategy pattern, the benefit of using this method is that consumers needn’t change after applying this refactoring.

```
 1: namespace LosTechies.DaysOfRefactoring.SwitchToStrategy.Before
 2: {
 3:     public class ClientCode
 4:     {
 5:         public decimal CalculateShipping()
 6:         {
 7:             ShippingInfo shippingInfo = new ShippingInfo();
 8:             return shippingInfo.CalculateShippingAmount(State.Alaska);
 9:         }
10:     }
11:
12:     public enum State
13:     {
14:         Alaska,
15:         NewYork,
16:         Florida
17:     }
18:
19:     public class ShippingInfo
20:     {
21:         public decimal CalculateShippingAmount(State shipToState)
22:         {
23:             switch(shipToState)
24:             {
25:                 case State.Alaska:
26:                     return GetAlaskaShippingAmount();
27:                 case State.NewYork:
28:                     return GetNewYorkShippingAmount();
29:                 case State.Florida:
30:                     return GetFloridaShippingAmount();
31:                 default:
32:                     return 0m;
33:             }
34:         }
35:
36:         private decimal GetAlaskaShippingAmount()
37:         {
38:             return 15m;
39:         }
40:
41:         private decimal GetNewYorkShippingAmount()
42:         {
43:             return 10m;
44:         }
45:
46:         private decimal GetFloridaShippingAmount()
47:         {
48:             return 3m;
49:         }
50:     }
51: }
```

To apply this refactoring take the condition that is being tested and place it in it's own class that adheres to a common interface. Then by passing the enum as the dictionary key, we can select the proper implementation and execute the code at hand. In the future when you want to add another condition, add another implementation and add the implementation to the ShippingCalculations dictionary. As I stated before, **this is not the only option to implement the strategy pattern**. I bold that because I know someone will bring this up in the comments :) Use what works for you. The benefit of doing this refactoring in this manner is that none of your client code will need to change. All of the modifications exist within the ShippingInfo class.

Jayme Davis pointed out that doing this refactoring really only ceates more classes because the binding still needs to be done via the ctor, but would be more beneficial if the binding of your IShippingCalculation strategies can be placed into IoC and that allows you to wire up strategies more easily.

```
 1: using System.Collections.Generic;
 2:
 3: namespace LosTechies.DaysOfRefactoring.SwitchToStrategy.After
 4: {
 5:     public class ClientCode
 6:     {
 7:         public decimal CalculateShipping()
 8:         {
 9:             ShippingInfo shippingInfo = new ShippingInfo();
10:             return shippingInfo.CalculateShippingAmount(State.Alaska);
11:         }
12:     }
13:
14:     public enum State
15:     {
16:         Alaska,
17:         NewYork,
18:         Florida
19:     }
20:
21:     public class ShippingInfo
22:     {
23:         private IDictionary<State, IShippingCalculation> ShippingCalculations
                                                { get; set; }
24:
25:         public ShippingInfo()
26:         {
27:             ShippingCalculations = new Dictionary<State, IShippingCalculation>
28:             {
29:                 { State.Alaska, new AlaskShippingCalculation() },
30:                 { State.NewYork, new NewYorkShippingCalculation() },
31:                 { State.Florida, new FloridaShippingCalculation() }
32:             };
33:         }
34:
35:         public decimal CalculateShippingAmount(State shipToState)
36:         {
37:             return ShippingCalculations[shipToState].Calculate();
38:         }
39:     }
40:
41:     public interface IShippingCalculation
42:     {
43:         decimal Calculate();
44:     }
45:
46:     public class AlaskShippingCalculation : IShippingCalculation
47:     {
```

```
48:         public decimal Calculate()
49:         {
50:             return 15m;
51:         }
52:     }
53:
54:     public class NewYorkShippingCalculation : IShippingCalculation
55:     {
56:         public decimal Calculate()
57:         {
58:             return 10m;
59:         }
60:     }
61:
62:     public class FloridaShippingCalculation : IShippingCalculation
63:     {
64:         public decimal Calculate()
65:         {
66:             return 3m;
67:         }
68:     }
69: }
```

To take this sample full circle, Here is how you would wire up your bindings if you were using Ninject as your IoC container in the ShippingInfo constructor. Quite a few things changed here, mainly the enum for the state now lives in the strategy and ninject gives us a IEnumerable of all bindings to the constructor of IShippingInfo. We then create a dictionary using the state property on the strategy to populate our dictionary and the rest is the same. (thanks to Nate Kohari and Jayme Davis)

```
 1: public interface IShippingInfo
 2: {
 3:     decimal CalculateShippingAmount(State state);
 4: }
 5:
 6: public class ClientCode
 7: {
 8:     [Inject]
 9:     public IShippingInfo ShippingInfo { get; set; }
10:
11:     public decimal CalculateShipping()
12:     {
13:         return ShippingInfo.CalculateShippingAmount(State.Alaska);
14:     }
15: }
16:
17: public enum State
18: {
19:     Alaska,
20:     NewYork,
21:     Florida
22: }
23:
24: public class ShippingInfo : IShippingInfo
25: {
26:     private IDictionary<State, IShippingCalculation> ShippingCalculations
                              { get; set; }
27:
28:     public ShippingInfo(IEnumerable<IShippingCalculation> shippingCalculations)
29:     {
30:         ShippingCalculations = shippingCalculations.ToDictionary(
                                       calc => calc.State);
31:     }
32:
33:     public decimal CalculateShippingAmount(State shipToState)
34:     {
35:         return ShippingCalculations[shipToState].Calculate();
```

```
36:     }
37: }
38:
39: public interface IShippingCalculation
40: {
41:     State State { get; }
42:     decimal Calculate();
43: }
44:
45: public class AlaskShippingCalculation : IShippingCalculation
46: {
47:     public State State { get { return State.Alaska; } }
48:
49:     public decimal Calculate()
50:     {
51:         return 15m;
52:     }
53: }
54:
55: public class NewYorkShippingCalculation : IShippingCalculation
56: {
57:     public State State { get { return State.NewYork; } }
58:
59:     public decimal Calculate()
60:     {
61:         return 10m;
62:     }
63: }
64:
65: public class FloridaShippingCalculation : IShippingCalculation
66: {
67:     public State State { get { return State.Florida; } }
68:
69:     public decimal Calculate()
70:     {
71:         return 3m;
72:     }
73: }
```

Refactoring Day 12 : Break Dependencies

Today's refactoring is useful if you are trying to introduce unit tests into your code base as testing “seams” are needed to properly mock/isolate areas you don’t wish to test. In this example we have client code that is using a static class to accomplish some work. The problem with this when it comes to unit testing because there is no way to mock the static class from our unit test. To work around this you can apply a wrapper interface around the static to create a seam and break the dependency on the static.

```
 1: public class AnimalFeedingService
 2: {
 3:     private bool FoodBowlEmpty { get; set; }
 4:
 5:     public void Feed()
 6:     {
 7:         if (FoodBowlEmpty)
 8:             Feeder.ReplenishFood();
 9:
10:         // more code to feed the animal
11:     }
12: }
13:
14: public static class Feeder
15: {
16:     public static void ReplenishFood()
17:     {
18:         // fill up bowl
19:     }
20: }
```

All we did to apply this refactoring was introduce an interface and class that simply calls the underlying static class. So the behavior is still the same, just the manner in which it is invoked has changed. This is good to get a starting point to begin refactoring from and an easy way to add unit tests to your code base.

```
 1: public class AnimalFeedingService
 2: {
 3:     public IFeederService FeederService { get; set; }
 4:
 5:     public AnimalFeedingService(IFeederService feederService)
 6:     {
 7:         FeederService = feederService;
 8:     }
 9:
10:     private bool FoodBowlEmpty { get; set; }
11:
12:     public void Feed()
13:     {
14:         if (FoodBowlEmpty)
15:             FeederService.ReplenishFood();
16:
17:         // more code to feed the animal
18:     }
19: }
20:
21: public interface IFeederService
22: {
23:     void ReplenishFood();
24: }
25:
26: public class FeederService : IFeederService
27: {
28:     public void ReplenishFood()
```

```
29:     {
30:         Feeder.ReplenishFood();
31:     }
32: }
33:
34: public static class Feeder
35: {
36:     public static void ReplenishFood()
37:     {
38:         // fill up bowl
39:     }
40: }
```

We can now mock IFeederService during our unit test via the AnimalFeedingService constructor by passing in a mock of IFeederService. Later we can move the code in the static into FeederService and delete the static class completely once we have some tests in place.

Refactoring Day 13 : Extract Method Object

Today's refactoring comes from Martin Fowlers list of refactorings. You can find his original article here with a brief description.

This is a more infrequent refactoring that I see myself using but it comes in handy at times. When trying to apply an Extract Method refactoring, and multiple methods are needing to be introduced, it is sometimes gets ugly because of multiple local variables that are being used within a method. Because of this reason, it is better to introduce an Extract Method Object refactoring and to segregate the logic required to perform the task.

```
 1: public class OrderLineItem
 2: {
 3:     public decimal Price { get; private set; }
 4: }
 5:
 6: public class Order
 7: {
 8:     private IList<OrderLineItem> OrderLineItems { get; set; }
 9:     private IList<decimal> Discounts { get; set; }
10:     private decimal Tax { get; set; }
11:
12:     public decimal Calculate()
13:     {
14:         decimal subTotal = 0m;
15:
16:         // Total up line items
17:         foreach (OrderLineItem lineItem in OrderLineItems)
18:         {
19:             subTotal += lineItem.Price;
20:         }
21:
22:         // Subtract Discounts
23:         foreach (decimal discount in Discounts)
24:             subTotal -= discount;
25:
26:         // Calculate Tax
27:         decimal tax = subTotal * Tax;
28:
29:         // Calculate GrandTotal
30:         decimal grandTotal = subTotal + tax;
31:
32:         return grandTotal;
33:     }
34: }
```

This entails passing a reference to the class that will be returning the computation to a new object that has the multiple methods via the constructor, or passing the individual parameters to the constructor of the method object. I will be showing the former here.

```
 1: public class OrderLineItem
 2: {
 3:     public decimal Price { get; private set;}
 4: }
 5:
 6: public class Order
 7: {
 8:     public IEnumerable<OrderLineItem> OrderLineItems { get; private set;}
 9:     public IEnumerable<decimal> Discounts { get; private set; }
10:     public decimal Tax { get; private set; }
```

```
11:
12:     public decimal Calculate()
13:     {
14:         return new OrderCalculator(this).Calculate();
15:     }
16: }
17:
18: public class OrderCalculator
19: {
20:     private decimal SubTotal { get; set;}
21:     private IEnumerable<OrderLineItem> OrderLineItems { get; set; }
22:     private IEnumerable<decimal> Discounts { get; set; }
23:     private decimal Tax { get; set; }
24:
25:     public OrderCalculator(Order order)
26:     {
27:         OrderLineItems = order.OrderLineItems;
28:         Discounts = order.Discounts;
29:         Tax = order.Tax;
30:     }
31:
32:     public decimal Calculate()
33:     {
34:         CalculateSubTotal();
35:
36:         SubtractDiscounts();
37:
38:         CalculateTax();
39:
40:         return SubTotal;
41:     }
42:
43:     private void CalculateSubTotal()
44:     {
45:         // Total up line items
46:         foreach (OrderLineItem lineItem in OrderLineItems)
47:             SubTotal += lineItem.Price;
48:     }
49:
50:     private void SubtractDiscounts()
51:     {
52:         // Subtract Discounts
53:         foreach (decimal discount in Discounts)
54:             SubTotal -= discount;
55:     }
56:
57:     private void CalculateTax()
58:     {
59:         // Calculate Tax
60:         SubTotal += SubTotal * Tax;
61:     }
62: }
```

Refactoring Day 14 : Break Responsibilities

When breaking apart responsibilities on a class this is enforcing Single Responsiblity Principle from SOLID. It's an easy approach to apply this refactoring although it's often disputed as what consitutes a "responsibility". While I won't be answering that here, I will show a clear cut example of a class that can be broken into several classes with specific responsibilities.

```
 1: public class Video
 2: {
 3:     public void PayFee(decimal fee)
 4:     {
 5:     }
 6:
 7:     public void RentVideo(Video video, Customer customer)
 8:     {
 9:         customer.Videos.Add(video);
10:     }
11:
12:     public decimal CalculateBalance(Customer customer)
13:     {
14:         return customer.LateFees.Sum();
15:     }
16: }
17:
18: public class Customer
19: {
20:     public IList<decimal> LateFees { get; set; }
21:     public IList<Video> Videos { get; set; }
22: }
```

As you can see here, the Video class has two responsibilities, once for handling video rentals, and another for managing how many rentals a customer has. We can break out the customer logic into it's own class to help seperate the responsibilities.

```
 1: public class Video
 2: {
 3:     public void RentVideo(Video video, Customer customer)
 4:     {
 5:         customer.Videos.Add(video);
 6:     }
 7: }
 8:
 9: public class Customer
10: {
11:     public IList<decimal> LateFees { get; set; }
12:     public IList<Video> Videos { get; set; }
13:
14:     public void PayFee(decimal fee)
15:     {
16:     }
17:
18:     public decimal CalculateBalance(Customer customer)
19:     {
20:         return customer.LateFees.Sum();
21:     }
22: }
```

Refactoring Day 15 : Remove Duplication

This is probably one of the most used refactoring in the forms of methods that are used in more than one place. Duplication will quickly sneak up on you if you're not careful and give in to apathy. It is often added to the codebase through laziness or a developer that is trying to produce as much code as possible, as quickly as possible. I don't think we need anymore description so let's look at the code.

```
 1: public class MedicalRecord
 2: {
 3:     public DateTime DateArchived { get; private set; }
 4:     public bool Archived { get; private set; }
 5:
 6:     public void ArchiveRecord()
 7:     {
 8:         Archived = true;
 9:         DateArchived = DateTime.Now;
10:     }
11:
12:     public void CloseRecord()
13:     {
14:         Archived = true;
15:         DateArchived = DateTime.Now;
16:     }
17: }
```

We move the duplicated code to a shared method and voila! No more duplication. Please enforce this refactoring whenever possible. It leads to much fewer bugs because you aren't copy/pasting the bugs throughout the code.

```
 1: public class MedicalRecord
 2: {
 3:     public DateTime DateArchived { get; private set; }
 4:     public bool Archived { get; private set; }
 5:
 6:     public void ArchiveRecord()
 7:     {
 8:         SwitchToArchived();
 9:     }
10:
11:     public void CloseRecord()
12:     {
13:         SwitchToArchived();
14:     }
15:
16:     private void SwitchToArchived()
17:     {
18:         Archived = true;
19:         DateArchived = DateTime.Now;
20:     }
21: }
```

Refactoring Day 16 : Encapsulate Conditional

Sometimes when doing a number of different checks within a conditional the intent of what you are testing for gets lost in the conditional. In these instances I like to extract the conditional into an easy to read property, or method depending if there is parameters to pass or not. Here is an example of what the code might look like before:

```
 1: public class RemoteControl
 2: {
 3:     private string[] Functions { get; set; }
 4:     private string Name { get; set; }
 5:     private int CreatedYear { get; set; }
 6:
 7:     public string PerformCoolFunction(string buttonPressed)
 8:     {
 9:         // Determine if we are controlling some extra function
10:         // that requires special conditions
11:         if (Functions.Length > 1 && Name == "RCA" &&
                              CreatedYear > DateTime.Now.Year - 2)
12:             return "doSomething";
13:     }
14: }
```

After we apply the refactoring, you can see the code reads much easier and conveys intent:

```
 1: public class RemoteControl
 2: {
 3:     private string[] Functions { get; set; }
 4:     private string Name { get; set; }
 5:     private int CreatedYear { get; set; }
 6:
 7:     private bool HasExtraFunctions
 8:     {
 9:         get { return Functions.Length > 1 && Name == "RCA" &&
                              CreatedYear > DateTime.Now.Year - 2; }
10:     }
11:
12:     public string PerformCoolFunction(string buttonPressed)
13:     {
14:         // Determine if we are controlling some extra function
15:         // that requires special conditions
16:         if (HasExtraFunctions)
17:             return "doSomething";
18:     }
19: }
```

Refactoring Day 17 : Extract Superclass

Today's refactoring is from Martin Fowler's refactoring catalog. You can find the original description here

This refactoring is used quite often when you have a number of methods that you want to “pull up” into a base class to allow other classes in the same hierarchy to use. Here is a class that uses two methods that we want to extract and make available to other classes.

```
 1: public class Dog
 2: {
 3:     public void EatFood()
 4:     {
 5:         // eat some food
 6:     }
 7:
 8:     public void Groom()
 9:     {
10:         // perform grooming
11:     }
12: }
```

After applying the refactoring we just move the required methods into a new base class. This is very similar to the [pull up refactoring], except that you would apply this refactoring when a base class doesn’t already exist.

```
 1: public class Animal
 2: {
 3:     public void EatFood()
 4:     {
 5:         // eat some food
 6:     }
 7:
 8:     public void Groom()
 9:     {
10:         // perform grooming
11:     }
12: }
13:
14: public class Dog : Animal
15: {
16: }
```

Refactoring Day 18 : Replace exception with conditional

Today's refactoring doesn't come from any place specifically, just something I've picked up over time that I find myself using often. Any variations/comments would be appreciated to this approach. I think there's some other good refactorings around these type of problems.

A common code smell that I come across from time to time is using exceptions to control program flow. You may see something to this effect:

```
 1: public class Microwave
 2: {
 3:     private IMicrowaveMotor Motor { get; set;}
 4:
 5:     public bool Start(object food)
 6:     {
 7:         bool foodCooked = false;
 8:         try
 9:         {
10:             Motor.Cook(food);
11:             foodCooked = true;
12:         }
13:         catch(InUseException)
14:         {
15:             foodcooked = false;
16:         }
17:
18:         return foodCooked;
19:     }
20: }
```

Exceptions should only be there to do exactly what they are for, handle *exceptional* behavior. Most of the time you can replace this type of code with a proper conditional and handle it properly. This is called design by contract in the after example because we are ensuring a specific state of the Motor class before performing the necessary work instead of letting an exception handle it.

```
 1: public class Microwave
 2: {
 3:     private IMicrowaveMotor Motor { get; set; }
 4:
 5:     public bool Start(object food)
 6:     {
 7:         if (Motor.IsInUse)
 8:             return false;
 9:
10:         Motor.Cook(food);
11:
12:         return true;
13:     }
14: }
```

Refactoring Day 19 : Extract Factory Class

Todays refactoring was first coined by the GangOfFour and has many resources on the web that have different usages of this pattern. Two different aims of the factory pattern can be found on the GoF website here and here.

Often in code some involved setup of objects is required in order to get them into a state where we can begin working with them. Uusally this setup is nothing more than creating a new instance of the object and working with it in whatever manner we need. Sometimes however the creation requirements of this object may grow and clouds the original code that was used to create the object. This is where a Factory class comes into play. For a full description of the factory pattern you can read more here. On the complex end of the factory pattern is for creating families of objects using Abstract Factory. Our usage is on the basic end where we have one factory class creating one specific instance for us. Take a look at the code before:

```
 1: public class PoliceCarController
 2: {
 3:     public PoliceCar New(int mileage, bool serviceRequired)
 4:     {
 5:         PoliceCar policeCar = new PoliceCar();
 6:         policeCar.ServiceRequired = serviceRequired;
 7:         policeCar.Mileage = mileage;
 8:
 9:         return policeCar;
10:     }
11: }
```

As we can see, the new action is responsible for creating a PoliceCar and then setting some initial properties on the police car depending on some external input. This works fine for simple setup, but over time this can grow and the burden of creating the police car is put on the controller which isn't really something that the controller should be tasked with. In this instance we can extract our creation code and place in a Factory class that has the distinct responsibility of create instances of PoliceCar's

```
 1: public interface IPoliceCarFactory
 2: {
 3:     PoliceCar Create(int mileage, bool serviceRequired);
 4: }
 5:
 6: public class PoliceCarFactory : IPoliceCarFactory
 7: {
 8:     public PoliceCar Create(int mileage, bool serviceRequired)
 9:     {
10:         PoliceCar policeCar = new PoliceCar();
11:         policeCar.ReadForService = serviceRequired;
12:         policeCar.Mileage = mileage;
13:         return policeCar;
14:     }
15: }
16:
17: public class PoliceCarController
18: {
19:     public IPoliceCarFactory PoliceCarFactory { get; set; }
20:
21:     public PoliceCarController(IPoliceCarFactory policeCarFactory)
22:     {
23:         PoliceCarFactory = policeCarFactory;
```

```
24:     }
25:
26:     public PoliceCar New(int mileage, bool serviceRequired)
27:     {
28:         return PoliceCarFactory.Create(mileage, serviceRequired);
29:     }
30: }
```

Now that we have the creation logic put off to a factory, we can add to that one class that is tasked with creating instances for us without the worry of missing something during setup or duplicating code.

Refactoring Day 20 : Extract Subclass

Todays refactoring comes from Martin Fowlers catalog of patterns. You can find this refactoring in his catalog here

This refactoring is useful when you have methods on a base class that are not shared amongst all classes and needs to be pushed down into it's own class. The example I'm using here is pretty straightforward. We start out with a single class called Registration. This class handles all information related to a student registering for a course.

```
1: public class Registration
2: {
3:     public NonRegistrationAction Action { get; set; }
4:     public decimal RegistrationTotal { get; set; }
5:     public string Notes { get; set; }
6:     public string Description { get; set; }
7:     public DateTime RegistrationDate { get; set; }
8: }
```

There is something that we've realized after working with this class. We are using it in two different contexts. The properties NonRegistrationAction and Notes are only ever used when dealing with a NonRegistration which is used to track a portion of the system that is slightly different than a normal registration. Noticing this, we can extract a subclass and move those properties down into the NonRegistration class where they more appropriately fit.

```
 1: public class Registration
 2: {
 3:     public decimal RegistrationTotal { get; set; }
 4:     public string Description { get; set; }
 5:     public DateTime RegistrationDate { get; set; }
 6: }
 7:
 8: public class NonRegistration : Registration
 9: {
10:     public NonRegistrationAction Action { get; set; }
11:     public string Notes { get; set; }
12: }
```

Refactoring Day 21 : Collapse Hierarchy

Todays refactoring comes from Martin Fowlers catalog of patterns. You can find this refactoring in his catalog here

Yesterday we looked at extracting a subclass for moving responsibilities down if they are not needed across the board. A Collapse Hierarchy refactoring would be applied when you realize you no longer need a subclass. When this happens it doesn't really make sense to keep your subclass around if it's properties can be merged into the base class and used strictly from there.

```
 1: public class Website
 2: {
 3:     public string Title { get; set; }
 4:     public string Description { get; set; }
 5:     public IEnumerable<Webpage> Pages { get; set; }
 6: }
 7:
 8: public class StudentWebsite : Website
 9: {
10:     public bool IsActive { get; set; }
11: }
```

Here we have a subclass that isn't doing too much. It just has one property to denote if the site is active or not. At this point maybe we realize that determing if a site is active is something we can use across the board so we can collapse the hierarchy back into only a Website and eliminate the StudentWebsite type.

```
1: public class Website
2: {
3:     public string Title { get; set; }
4:     public string Description { get; set; }
5:     public IEnumerable<Webpage> Pages { get; set; }
6:     public bool IsActive { get; set; }
7: }
```

Refactoring Day 22 : Break Method

Today's refactoring didn't really come from any one source. It just named it although someone else may have something similar that's named differently. If you know of anyone that has a name for this other than Break Method, please let me know.

This refactoring is kind of a meta-refactoring in the fact that it's just extract method applied over and over until you decompose one large method into several smaller methods. This example here is a tad contrived because the AcceptPayment method isn't doing as much as I wanted. Imagine that there is much more supporting code around each action that the one method is doing. That would match a real world scenario if you can picture it that way.

Below we have the AcceptPayment method that can be decomposed multiple times into distinct methods.

```
 1: public class CashRegister
 2: {
 3:     public CashRegister()
 4:     {
 5:         Tax = 0.06m;
 6:     }
 7:
 8:     private decimal Tax { get; set; }
 9:
10:     public void AcceptPayment(Customer customer, IEnumerable<Product> products,
                                  decimal payment)
11:     {
12:         decimal subTotal = 0m;
13:         foreach (Product product in products)
14:         {
15:             subTotal += product.Price;
16:         }
17:
18:         foreach(Product product in products)
19:         {
20:             subTotal -= product.AvailableDiscounts;
21:         }
22:
23:         decimal grandTotal = subTotal * Tax;
24:
25:         customer.DeductFromAccountBalance(grandTotal);
26:     }
27: }
28:
29: public class Customer
30: {
31:     public void DeductFromAccountBalance(decimal amount)
32:     {
33:         // deduct from balance
34:     }
35: }
36:
37: public class Product
38: {
39:     public decimal Price { get; set; }
40:     public decimal AvailableDiscounts { get; set; }
41: }
```

As you can see the AcceptPayment method has a couple of things that can be decomposed into targeted methods. So we perform the Extract Method refactoring a number of times until we come up with the result:

```
 1: public class CashRegister
 2: {
 3:     public CashRegister()
 4:     {
 5:         Tax = 0.06m;
 6:     }
 7:
 8:     private decimal Tax { get; set; }
 9:     private IEnumerable<Product> Products { get; set; }
10:
11:     public void AcceptPayment(Customer customer, IEnumerable<Product> products,
                                  decimal payment)
12:     {
13:         decimal subTotal = CalculateSubtotal();
14:
15:         subTotal = SubtractDiscounts(subTotal);
16:
17:         decimal grandTotal = AddTax(subTotal);
18:
19:         SubtractFromCustomerBalance(customer, grandTotal);
20:     }
21:
22:     private void SubtractFromCustomerBalance(Customer customer, decimal grandTotal)
23:     {
24:         customer.DeductFromAccountBalance(grandTotal);
25:     }
26:
27:     private decimal AddTax(decimal subTotal)
28:     {
29:         return subTotal * Tax;
30:     }
31:
32:     private decimal SubtractDiscounts(decimal subTotal)
33:     {
34:         foreach(Product product in Products)
35:         {
36:             subTotal -= product.AvailableDiscounts;
37:         }
38:         return subTotal;
39:     }
40:
41:     private decimal CalculateSubtotal()
42:     {
43:         decimal subTotal = 0m;
44:         foreach (Product product in Products)
45:         {
46:             subTotal += product.Price;
47:         }
48:         return subTotal;
49:     }
50: }
51:
52: public class Customer
53: {
54:     public void DeductFromAccountBalance(decimal amount)
55:     {
56:         // deduct from balance
57:     }
58: }
59:
60: public class Product
61: {
62:     public decimal Price { get; set; }
63:     public decimal AvailableDiscounts { get; set; }
64: }
```

Refactoring Day 23 : Introduce Parameter Object

This refactoring comes from Fowler's refactoring catalog and can be found here

Sometimes when working with a method that needs several parameters it becomes difficult to read the method signature because of five or more parameters being passed to the method like so:

```
1: public class Registration
2: {
3:     public void Create(decimal amount, Student student,
                   IEnumerable<Course> courses, decimal credits)
5:     {
6:         // do work
7:     }
8: }
```

In this instances it's useful to create a class who's only responsibility is to carry parameters into the method. This helps make the code more flexible because to add more parameters, you need only to add another field to the parameter object. Be careful to only use this refactoring when you find that you have a large number of parameters to pass to the method however as it does add several more classes to your codebase and should be kept to a minimum.

```
 1: public class RegistrationContext
 2: {
 3:     public decimal Amount { get; set; }
 4:     public Student Student { get; set; }
 5:     public IEnumerable<Course> Courses { get; set; }
 6:     public decimal Credits { get; set; }
 7: }
 8:
 9: public class Registration
10: {
11:     public void Create(RegistrationContext registrationContext)
12:     {
13:         // do work
14:     }
15: }
```

Refactoring Day 24 : Remove Arrowhead Antipattern

Today's refactoring is based on the c2 wiki entry and can be found here. Los Techies own Chris Missal also did a very informative post on the antipattern that you can find here.

Simply put, the arrowhead antipattern is when you have nested conditionals so deep that they form an arrowhead of code. I see this very often in different code bases and it makes for high cyclomatic complexity in code.

A good example of the arrowhead antipattern can be found in this sample here:

```
 1: public class Security
 2: {
 3:     public ISecurityChecker SecurityChecker { get; set; }
 4:
 5:     public Security(ISecurityChecker securityChecker)
 6:     {
 7:         SecurityChecker = securityChecker;
 8:     }
 9:
10:     public bool HasAccess(User user, Permission permission,
                                    IEnumerable<Permission> exemptions)
11:     {
12:         bool hasPermission = false;
13:
14:         if (user != null)
15:         {
16:             if (permission != null)
17:             {
18:                 if (exemptions.Count() == 0)
19:                 {
20:                     if (SecurityChecker.CheckPermission(user, permission) ||
                                            exemptions.Contains(permission))
21:                     {
22:                         hasPermission = true;
23:                     }
24:                 }
25:             }
26:         }
27:
28:         return hasPermission;
29:     }
30: }
```

Refactoring away from the arrowhead antipattern is as simple as swapping the conditionals to leave the method as soon as possible. Refactoring in this manner often starts to look like Design By Contract checks to evaluate conditions before performing the work of the method. Here is what this same method might look like after refactoring.

```
1: public class Security
2: {
3:     public ISecurityChecker SecurityChecker { get; set; }
4:
5:     public Security(ISecurityChecker securityChecker)
6:     {
7:         SecurityChecker = securityChecker;
8:     }
9:
```

```
10:     public bool HasAccess(User user, Permission permission,
                                      IEnumerable<Permission> exemptions)
11:     {
12:         if (user == null || permission == null)
13:             return false;
14:
15:         if (exemptions.Contains(permission))
16:             return true;
17:
18:         return SecurityChecker.CheckPermission(user, permission);
19:     }
20: }
```

As you can see, this method is much more readable and maintainable going forward. It's not as difficult to see all the different paths you can take through this method.

Refactoring Day 25 : Introduce Design By Contract checks

Design By Contract or DBC defines that methods should have defined input and output verifications. Therefore, you can be sure you are always working with a usable set of data in all methods and everything is behaving as expected. If not, exceptions or errors should be returned and handled from the methods. To read more on DBC read the wikipedia page here.

In our example here, we are working with input parameters that may possibly be null. As a result a NullReferenceException would be thrown from this method because we never verify that we have an instance. During the end of the method, we don't ensure that we are returning a valid decimal to the consumer of this method and may introduce methods elsewhere.

```
 1: public class CashRegister
 2: {
 3:     public decimal TotalOrder(IEnumerable<Product> products, Customer customer)
 4:     {
 5:         decimal orderTotal = products.Sum(product => product.Price);
 6:
 7:         customer.Balance += orderTotal;
 8:
 9:         return orderTotal;
10:     }
11: }
```

The changes we can make here to introduce DBC checks is pretty easy. First we will assert that we don't have a null customer, check that we have at least one product to total. Before we return the order total we will ensure that we have a valid amount for the order total. If any of these checks fail in this example we should throw targeted exceptions that detail exactly what happened and fail gracefully rather than throw an obscure NullReferenceException.

It seems as if there is some DBC framework methods and exceptions in the Microsoft.Contracts namespace that was introduced with .net framework 3.5. I personally haven't played with these yet, but they may be worth looking at. This is the only thing I could find on msdn about the namespace.

```
 1: public class CashRegister
 2: {
 3:     public decimal TotalOrder(IEnumerable<Product> products, Customer customer)
 4:     {
 5:         if (customer == null)
 6:             throw new ArgumentNullException("customer", "Customer cannot be null");
 7:         if (products.Count() == 0)
 8:             throw new ArgumentException("Must have at least one product to total",
                                        "products");
 9:
10:         decimal orderTotal = products.Sum(product => product.Price);
11:
12:         customer.Balance += orderTotal;
13:
14:         if (orderTotal == 0)
15:             throw new ArgumentOutOfRangeException("orderTotal",
                                               "Order Total should not be zero");
16:
17:         return orderTotal;
18:     }
```

```
19: }
```

It does add more code to the method for validation checks and you can go overboard with DBC, but I think in most scenarios it is a worthwhile endeavor to catch sticky situations. It really stinks to chase after a NullReferenceException without detailed information.

Refactoring Day 26 : Remove Double Negative

Today's refactoring comes from Fowler's refactoring catalog and can be found here.

This refactoring is pretty simple to implement although I find it in many codebases that severely hurts readability and almost always conveys incorrect intent. This type of code does the most damage because of the assumptions made on it. Assumptions lead to incorrect maintenance code written, which in turn leads to bugs. Take the following example:

```
 1: public class Order
 2: {
 3:     public void Checkout(IEnumerable<Product> products, Customer customer)
 4:     {
 5:         if (!customer.IsNotFlagged)
 6:         {
 7:             // the customer account is flagged
 8:             // log some errors and return
 9:             return;
10:         }
11:
12:         // normal order processing
13:     }
14: }
15:
16: public class Customer
17: {
18:     public decimal Balance { get; private set; }
19:
20:     public bool IsNotFlagged
21:     {
22:         get { return Balance < 30m; }
23:     }
24: }
```

As you can see the double negative here is difficult to read because we have to figure out what is positive state of the two negatives. The fix is very easy. If we don't have a positive test, add one that does the double negative assertion for you rather than make sure you get it correct.

```
 1: public class Order
 2: {
 3:     public void Checkout(IEnumerable<Product> products, Customer customer)
 4:     {
 5:         if (customer.IsFlagged)
 6:         {
 7:             // the customer account is flagged
 8:             // log some errors and return
 9:             return;
10:         }
11:
12:         // normal order processing
13:     }
14: }
15:
16: public class Customer
17: {
18:     public decimal Balance { get; private set; }
19:
20:     public bool IsFlagged
21:     {
22:         get { return Balance >= 30m; }
23:     }
```

```
24: }
```

Refactoring Day 27 : Remove God Classes

Often with legacy code bases I will often come across classes that are clear SRP violations. Often these classes will be suffixed with either “Utils” or “Manager”. Sometimes they don’t have this indication and are just classes with multiple grouped pieces of functionality. Another good indicator of a God class is methods grouped together with using statements or comments into seperate roles that this one class is performing.

Over time, these classes become a dumping ground for a method that someone doesn’t have time/want to put in the proper class. The refactoring for situations like these is to break apart the methods into distinct classes that are responsible for specific roles.

```
 1: public class CustomerService
 2: {
 3:     public decimal CalculateOrderDiscount(IEnumerable<Product> products,
                                              Customer customer)
 4:     {
 5:         // do work
 6:     }
 7:
 8:     public bool CustomerIsValid(Customer customer, Order order)
 9:     {
10:         // do work
11:     }
12:
13:     public IEnumerable<string> GatherOrderErrors(IEnumerable<Product> products,
                                              Customer customer)
14:     {
15:         // do work
16:     }
17:
18:     public void Register(Customer customer)
19:     {
20:         // do work
21:     }
22:
23:     public void ForgotPassword(Customer customer)
24:     {
25:         // do work
26:     }
27: }
```

The refactoring for this is very straight forward. Simply take the related methods and place them in specific classes that match their responsibility. This makes them much finer grained and defined in what they do and make future maintenance much easier. Here is the end result of splitting up the methods above into two distinct classes.

```
 1: public class CustomerOrderService
 2: {
 3:     public decimal CalculateOrderDiscount(IEnumerable<Product> products,
                                              Customer customer)
 4:     {
 5:         // do work
 6:     }
 7:
 8:     public bool CustomerIsValid(Customer customer, Order order)
 9:     {
10:         // do work
11:     }
12:
```

```
13:     public IEnumerable<string> GatherOrderErrors(IEnumerable<Product> products,
                                                     Customer customer)
14:     {
15:         // do work
16:     }
17: }
18:
19: public class CustomerRegistrationService
20: {
21:
22:     public void Register(Customer customer)
23:     {
24:         // do work
25:     }
26:
27:     public void ForgotPassword(Customer customer)
28:     {
29:         // do work
30:     }
31: }
```

Refactoring Day 28 : Rename boolean method

Today's refactoring doesn't necessarily come from Fowlers refactoring catalog. If anyone knows where this "refactoring" actually comes from, please let me know.

Granted, this could be viewed as not being a refactoring as the methods are actually changing, but this is a gray area and open to debate. Methods with a large number of boolean parameters can quickly get out of hand and can produce unexpected behavior. Depending on the number of parameters will determine how many methods need to be broken out. Let's take a look at where this refactoring starts:

```
1: public class BankAccount
2: {
3:     public void CreateAccount(Customer customer, bool withChecking,
                   bool withSavings, bool withStocks)
4:     {
5:         // do work
6:     }
7: }
```

We can make this work a little better simple by exposing the boolean parameters via well named methods and in turn make the original method private to prevent anyone from calling it going forward. Obviously you could have a large number of permutations here and perhaps it makes more sense to refactor to a Parameter object instead.

```
 1: public class BankAccount
 2: {
 3:     public void CreateAccountWithChecking(Customer customer)
 4:     {
 5:         CreateAccount(customer, true, false);
 6:     }
 7:
 8:     public void CreateAccountWithCheckingAndSavings(Customer customer)
 9:     {
10:         CreateAccount(customer, true, true);
11:     }
12:
13:     private void CreateAccount(Customer customer, bool withChecking,
                             bool withSavings)
14:     {
15:         // do work
16:     }
17: }
```

Refactoring Day 29 : Remove Middle Man

Today's refactoring comes from Fowler's refactoring catalog and can be found here.

Sometimes in code you may have a set of "Phantom" or "Ghost" classes. Fowler calls these "Middle Men". Middle Men classes simply take calls and forward them on to other components without doing any work. This is an unneeded layer and can be removed completely with minimal effort.

```
 1: public class Consumer
 2: {
 3:     public AccountManager AccountManager { get; set; }
 4:
 5:     public Consumer(AccountManager accountManager)
 6:     {
 7:         AccountManager = accountManager;
 8:     }
 9:
10:     public void Get(int id)
11:     {
12:         Account account = AccountManager.GetAccount(id);
13:     }
14: }
15:
16: public class AccountManager
17: {
18:     public AccountDataProvider DataProvider { get; set; }
19:
20:     public AccountManager(AccountDataProvider dataProvider)
21:     {
22:         DataProvider = dataProvider;
23:     }
24:
25:     public Account GetAccount(int id)
26:     {
27:         return DataProvider.GetAccount(id);
28:     }
29: }
30:
31: public class AccountDataProvider
32: {
33:     public Account GetAccount(int id)
34:     {
35:         // get account
36:     }
37: }
```

The end result is straightforward enough. We just remove the middle man object and point the original call to the intended receiver.

```
 1: public class Consumer
 2: {
 3:     public AccountDataProvider AccountDataProvider { get; set; }
 4:
 5:     public Consumer(AccountDataProvider dataProvider)
 6:     {
 7:         AccountDataProvider = dataProvider;
 8:     }
 9:
10:     public void Get(int id)
11:     {
12:         Account account = AccountDataProvider.GetAccount(id);
13:     }
```

```
14: }
15:
16: public class AccountDataProvider
17: {
18:     public Account GetAccount(int id)
19:     {
20:         // get account
21:     }
22: }
```

Refactoring Day 30 : Return ASAP

This topic actually came up during the Remove Arrowhead Antipattern refactoring. The refactoring introduces this as a side effect to remove the arrowhead. To eliminate the arrowhead you return as soon as possible.

```
 1: public class Order
 2: {
 3:     public Customer Customer { get; private set; }
 4:
 5:     public decimal CalculateOrder(Customer customer, IEnumerable<Product> products,
                                      decimal discounts)
 6:     {
 7:         Customer = customer;
 8:         decimal orderTotal = 0m;
 9:
10:         if (products.Count() > 0)
11:         {
12:             orderTotal = products.Sum(p => p.Price);
13:             if (discounts > 0)
14:             {
15:                 orderTotal -= discounts;
16:             }
17:         }
18:
19:         return orderTotal;
20:     }
21: }
```

The idea is that as soon as you know what needs to be done and you have all the required information, you should exit the method as soon as possible and not continue along.

```
 1: public class Order
 2: {
 3:     public Customer Customer { get; private set; }
 4:
 5:     public decimal CalculateOrder(Customer customer, IEnumerable<Product> products,
                                      decimal discounts)
 6:     {
 7:         if (products.Count() == 0)
 8:             return 0;
 9:
10:         Customer = customer;
11:         decimal orderTotal = products.Sum(p => p.Price);
12:
13:         if (discounts == 0)
14:             return orderTotal;
15:
16:         orderTotal -= discounts;
17:
18:         return orderTotal;
19:     }
20: }
```

Refactoring Day 31 : Replace conditional with Polymorphism

The last day of refactoring comes from Fowlers refactoring catalog and can be found here.

This shows one of the foundations of Object Oriented Programming which is Polymorphism. The concept here is that in instances where you are doing checks by type, and performing some type of operation, it's a good idea to encapsulate that algorithm within the class and then use polymorphism to abstract the call to the code.

```
 1: public abstract class Customer
 2: {
 3: }
 4:
 5: public class Employee : Customer
 6: {
 7: }
 8:
 9: public class NonEmployee : Customer
10: {
11: }
12:
13: public class OrderProcessor
14: {
15:     public decimal ProcessOrder(Customer customer, IEnumerable<Product> products)
16:     {
17:         // do some processing of order
18:         decimal orderTotal = products.Sum(p => p.Price);
19:
20:         Type customerType = customer.GetType();
21:         if (customerType == typeof(Employee))
22:         {
23:             orderTotal -= orderTotal * 0.15m;
24:         }
25:         else if (customerType == typeof(NonEmployee))
26:         {
27:             orderTotal -= orderTotal * 0.05m;
28:         }
29:
30:         return orderTotal;
31:     }
32: }
```

As you can see here, we're not leaning on our inheritance hierarchy to put the calculation, or even the data needed to perform the calculation lest we have a SRP violation. So to refactor this we simply take the percentage rate and place that on the actual customer type that each class will then implement. I know this is really remedial but I wanted to cover this as well as I have seen it in code.

```
 1: public abstract class Customer
 2: {
 3:     public abstract decimal DiscountPercentage { get; }
 4: }
 5:
 6: public class Employee : Customer
 7: {
 8:     public override decimal DiscountPercentage
 9:     {
10:         get { return 0.15m; }
```

```
11:     }
12: }
13:
14: public class NonEmployee : Customer
15: {
16:     public override decimal DiscountPercentage
17:     {
18:         get { return 0.05m; }
19:     }
20: }
21:
22: public class OrderProcessor
23: {
24:     public decimal ProcessOrder(Customer customer, IEnumerable<Product> products)
25:     {
26:         // do some processing of order
27:         decimal orderTotal = products.Sum(p => p.Price);
28:
29:         orderTotal -= orderTotal * customer.DiscountPercentage;
30:
31:         return orderTotal;
32:     }
33: }
```

Appendix A

The code samples for this eBook can be download from Sean's repository on GitHub:

http://github.com/schambers/days-of-refactoring