

Agile .NET Development – Refactoring

Jason Gorman

What is Refactoring?

- Improving the design of existing code *without* changing what it does
- Making the code easier to change
 - More loosely coupled
 - More cohesive modules
 - More comprehensible
- Each refactoring is *small* and *reversible*
- **Automated unit tests** highlight any problems inadvertently caused by a refactoring - “side effects”
- Term first coined by [Martin Fowler](#) in his book [*Refactoring : Improving The Design Of Existing Code*](#)

Refactoring & Agile Methods

- Agile Development assumes that the best designs will evolve and emerge through many iterations
- Refactoring is essential to an agile, evolutionary approach to design because designs need to change as we learn through constant feedback what works and what doesn't

The Refactoring Process

- Make a small change – a single refactoring
- Run all the tests to ensure everything still works
- If everything works, move on to the next refactoring
- If not, fix the problem, or undo the change, so you still have a working system

“Code Smells”

- Code that can make the design harder to change:
- Eg,
 - Duplicate code
 - Long methods
 - Big classes
 - Big switch statements
 - Long navigations (ag, a.b().c().d())
 - Too much checking for null objects
 - Data clumps (eg, a Contact class that has fields for address, phone, email etc etc) – similar to non-normalised tables in relational design
 - Data classes (classes that have mainly fields/properties and little or no methods)
 - Un-encapsulated fields (public member variables)

Common Refactorings

Extract Class

Having the phone details as part of the Customer class is not a realistic OO model, and also breaks the Single Responsibility design principle. We can refactor this into two separate classes, each with the appropriate responsibility.

```
public class Customer
{
    private string name;
    private string workPhoneAreaCode;
    private string workPhoneNumber;
}
```



```
public class Customer
{
    private string name;
    private Phone workPhone;
}

public class Phone
{
    private string areaCode;
    private string number;
}
```

Extract Interface

Some clients may need to know a Customer's name, while others may only need to know that certain objects can be serialized to XML. Having ToXml() as part of the Customer interface breaks the Interface Segregation design principle which tells us that it's better to have more specialised interfaces than to have one multi-purpose interface.

```
public class Customer
{
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public string ToXml()
    {
        return "<Customer><Name>" +
            name + @"</Name></Customer>";
    }
}
```



```
public class Customer
{
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    string SerializableToXml.ToXml()
    {
        return "<Customer><Name>" +
            name + @"</Name></Customer>";
    }
}

public interface SerializableToXml
{
    string ToXml();
}
```

Extract Method

```
public void PrintAccountDetails(Account account)
{
    // print summary
    Console.WriteLine("Account: " + account.Id);
    Console.WriteLine("Balance: " + account.Balance);
    // print history
    foreach(Transaction tx in account.Transactions)
    {
        Console.WriteLine("Type: " + tx.Type +
            "Date: " + tx.Date + " Amount: "
            + tx.Amount);
    }
}
```

```
public void PrintAccountDetails(Account account)
{
    PrintSummary(account);
    PrintHistory(account);
}

private void PrintSummary(Account account)
{
    Console.WriteLine("Account: " + account.Id);
    Console.WriteLine("Balance: " + account.Balance);
}

private void PrintHistory(Account account)
{
    foreach(Transaction tx in account.Transactions)
    {
        Console.WriteLine("Type: " + tx.Type +
            "Date: " + tx.Date + " Amount: "
            + tx.Amount);
    }
}
```

Sometimes we have methods that do *too much*. The more code in a single method, the harder it is to understand and get right. It also means that logic embedded in that method cannot be reused elsewhere. The **Extract Method** refactoring is one of the most useful for reducing the amount of duplication in code.

Extract Subclass

When a class has features (attributes and methods) that would only be useful in specialised instances, we can create a specialisation of that class and give it those features. This makes the original class less specialised (ie, *more abstract*), and good design is about binding to abstractions wherever possible.

```
public class Person
{
    private string name;
    private string jobTitle;
}
```



```
public class Person
{
    protected string name;
}

public class Employee : Person
{
    private string jobTitle;
}
```

Extract Super-class

When you find two or more classes that share common features, consider abstracting those shared features into a super-class. Again, this makes it easier to bind clients to an abstraction, and removes duplicate code from the original classes.

```
public class Employee
{
    private string name;
    private string jobTitle;
}

public class Student
{
    private string name;
    private Course course;
}
```



```
public abstract class Person
{
    protected string name;
}

public class Employee : Person
{
    private string jobTitle;
}

public class Student : Person
{
    private Course course;
}
```

Form Template Method - Before

```
public abstract class Party
{
}

public class Person : Party
{
    private string firstName;
    private string lastName;
    private DateTime dob;
    private string nationality;

    public void PrintNameAndDetails()
    {
        Console.WriteLine("Name: " + firstName + " " + lastName);
        Console.WriteLine("DOB: " + dob.ToShortDateString() +
            ", Nationality: " + nationality);
    }
}

public class Company : Party
{
    private string name;
    private string companyType;
    private DateTime incorporated;

    public void PrintNameAndDetails()
    {
        Console.WriteLine("Name: " + name + " " + companyType);
        Console.WriteLine("Incorporated: " +
            incorporated.ToShortDateString());
    }
}
```

When you find two methods in subclasses that perform the same steps, but do different things in each step, create methods for those steps with the same signature and move the original method into the base class.

Form Template Method - Refactored

```
public abstract class Party
{
    public void PrintNameAndDetails()
    {
        PrintName();
        PrintDetails();
    }
    public abstract void PrintName();
    public abstract void PrintDetails();
}

public class Person : Party
{
    private string firstName;
    private string lastName;
    private DateTime dob;
    private string nationality;

    public override void PrintName()
    {
        Console.WriteLine("Name: " + firstName + " " + lastName);
    }
    public override void PrintDetails()
    {
        Console.WriteLine("DOB: " + dob.ToShortDateString() +
            ", Nationality: " + nationality);
    }
}

public class Company : Party
{
    ...etc
}
```

Move Method - Before

If a method on one class uses (or is used by) another class more than the class on which its defined, move it to the other class

```
public class Student
{
    public bool IsTaking(Course course)
    {
        return (Array.IndexOf(course.Students, this, 0,
            course.Students.Length) >= 0);
    }
}

public class Course
{
    private Student[] students;

    public Student[] Students
    {
        get { return students; }
    }
}
```

Move Method – Refactored

The student class now no longer needs to know about the Course interface, and the IsTaking() method is closer to the data on which it relies -0 making the design of Course more cohesive and the overall design more loosely coupled

```
public class Student
{
}

public class Course
{
    private Student[] students;

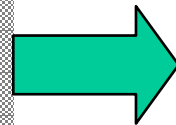
    public bool IsTaking(Student student)
    {
        return (Array.IndexOf(students, this, 0,
            students.Length) >= 0);
    }
}
```

Encapsulate Field

Un-encapsulated data is an absolute no-no in OO application design. Use property get and set procedures to provide public access to private (encapsulated) member variables.

```
public class Course
{
    public Student[] students;
}
```

```
int classSize = course.students.Length;
```



```
public class Course
{
    private Student[] students;

    public Student[] Students
    {
        get { return students; }
    }
}
```

```
int classSize = course.Students.Length;
```

More Refactorings

- <http://www.refactoring.com/catalog>

.NET Refactoring Tools

- Several Visual Studio Add-ins are available to aid with the mechanics of moving and renaming code elements:
- [Flywheel \(Velocitis\)](#)
- [.NET Refactoring](#)
- [C# Refactory \(Xtreme Simplicity\)](#)

Refactoring in VS Whidbey

- The next version of Visual Studio will have built-in support for common refactorings