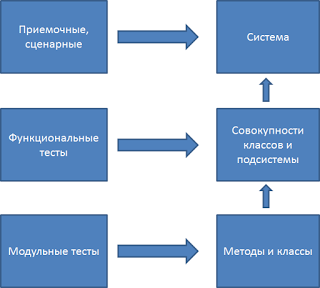
**Модульные тесты в Visual Studio 2012**

Под катом можно найти:  
 - Что такое модульные тесты?  
 - Пример написания модульного теста в VS 2012  
 - TDD  
 - Fakes Framework  
  
Так как сейчас, основной парадигмой разработки программного обеспечения является ООП, то многие программы состоят из классов, подсистем и систем. На каждом уровне, что логично, используется свой подход к тестированию:

[](http://3.bp.blogspot.com/-1H5ILGcLxVc/UPJp25lMY6I/AAAAAAAAA4E/5c_TK8s9XLs/s1600/01.png)

Само собой, эта картинка не раскрывает все виды тестирования, ведь еще есть нагрузочное тестирование, тестирование установки, тестирование на отказ и восстановление, тестирование безопасности и т.д.  
Т.к. про любой из видов тестирования можно писать целые книги, то сегодня, предлагаю ограничиться только модульным тестированием (unit testing).  
Для начала, давайте посмотрим на определения, что же такое модульные тесты:  
**Модульное тестирование, или юнит-тестирование** (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы. ([wikipedia.org](http://ru.wikipedia.org/wiki/%D0%AE%D0%BD%D0%B8%D1%82-%D1%82%D0%B5%D1%81%D1%82))  
**Модульный тест** — это автоматизированный фрагмент кода, который вызывает тестируемый метод или класс, а затем проверяет несколько предположений относительно логического поведения метода или класса. ([Джефф Левинсон](http://www.books.ru/books/testirovanie-po-s-pomoshchyu-visual-studio-2010-1562110/" \t "_blank))  
**Модульный тест** — это код, который обеспечивает выполнение части вашего рабочего кода с ожиданием результата. (кто автор не понятно, но взял [здесь](http://msdn.microsoft.com/ru-ru/subscriptions/gg655487.aspx))  
**Модульный тест** — это метод, который тестирует метод, класс или более крупный компонент вашего приложения изолированно от других частей, внешних систем и ресурсов. ([Ларри Брейдер, Алан Кэмерон Уиллс](http://msdn.microsoft.com/ru-ru/magazine/jj618302.aspx))  
Как видите, даже в определениях можно найти такое вот разнообразие. Мне больше всего нравится последние определение. Четко и однозначно определяющее, что модульные тесты, это изолированные тесты. Когда тестируемая сущность не взаимодействует с привычным окружением, которое тоже может быть причиной появления ошибок.  
Ладно, уже достаточно много слов, даже картинка есть, пора посмотреть, как все это работает.  
Создаем новый проект, я для простоты возьму Class Library и буду решать задачу нахождения корней квадратного уравнения.  
Для начала, я написал вот такой простой код:

public static double[] Calc(double a, double b, double c)

{

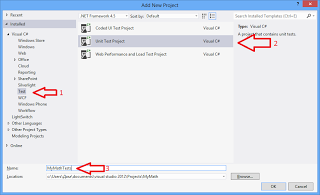
    double d = Math.Pow(b, 2) - 4 \* a \* c;

    double[] result = new double[] { (-b + Math.Sqrt(d)) / (2 \* a), (-b -Math.Sqrt(d)) / (2 \* a) };

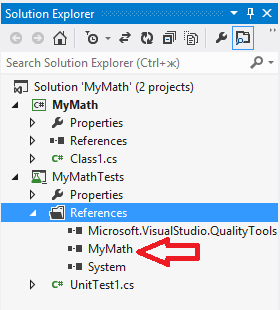
    return result;

}

Все просто, но надо проверить, а действительно ли работает так как надо? Напоминаю, у нас не приложение, а dll, поэтому проверить правильность запустив приложение и вызвав метод мы не сможем. И вот здесь, самое главное не сделать ошибку, не начать добавлять в проект консольное или виндовс приложение, для тестирования. Что мы хотим сделать? Протестировать! Значит и надо добавлять проект для тестирования.  
Правый клик на решении, добавить новый проект:

[](http://4.bp.blogspot.com/-idZoGYg8LyM/UPJxOyPePrI/AAAAAAAAA4k/9OQzWcs8PKM/s1600/02.png)

После добавления в тестовый проект ссылки на исходный, должно получиться нечто вот такое:

[](http://2.bp.blogspot.com/-4xVmETj68a4/UPJylCG0sOI/AAAAAAAAA5E/_rnihTlEg14/s1600/03.png)

Все, переходим в файл UnitTest1.cs и правим TestMethod1:

[TestClass]

public class UnitTest1

{

    [TestMethod]

    public void TestMethod1()

    {

        double a = 1;

        double b = -5;

        double c = 4;

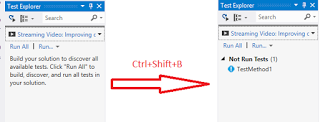
        double[] expected = { 4, 1 };

        double[] actual = SqrtRoots.Calc(a, b, c);

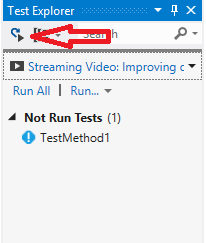
        CollectionAssert.AreEqual(expected, actual);

    }

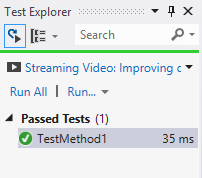
Обычно, тестовый метод состоит из части отвечающей за инициализацию данных необходимых для запуска теста, из самого теста и проверки одного или нескольких условий, по окончании работы.  
Для того чтобы работать с модульными тестами, нам понадобиться открыть окно Test Exlorer-а: Test -> Windows -> Test Explorer. Правда после открытия в нем тестов не будет, несмотря на то, что мы свой метод уже и написали. Для того, чтобы увидеть тест в Test Explorer-е, необходимо решение сбилдить:

[](http://4.bp.blogspot.com/-5pTGRUp2EHc/UPJ0VgdjK6I/AAAAAAAAA5k/Ivdi2zq7gbM/s1600/04.png)

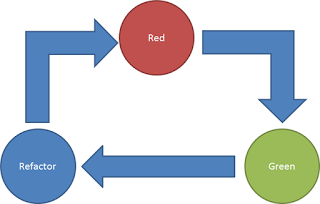
В Test Explorer мы можем запускать все тесты, открыв выпадающий список рядом с Run, только не прошедшие, не запускавшиеся, только прошедшие. Из контекстного меню на тесте, можно запустить конкретный тест, перейти к его описанию, отладить его и т.д. Но самая полезная кнопка, на мой взгляд, это кнопка принудительного запуска тестов после каждого построения:

[](http://4.bp.blogspot.com/-dl7A3cpTDoY/UPJ1Od9YyAI/AAAAAAAAA5w/QYzq_pzpsHg/s1600/05.png)

Включаем эту опцию и перестраиваем наше решение:

[](http://3.bp.blogspot.com/-nR_cxGq_P9E/UPJ1rnRPmsI/AAAAAAAAA6M/lDJBl9qIfkg/s1600/06.png)

Замечательно, тест пройден. Кстати, в VS 2012 тесты выполняются в отдельном потоке, поэтому, в то время как они проверяют ваш код, вы можете тестировать приложение ручками, дописывать новый код и т.д. Но, несмотря на эту замечательную особенность VS, тесты должны быть короткими и проверять изолированную часть кода, в следствии чего, время выполнения тестов должно быть как можно меньше. Ведь в реальном проекте тестов может быть несколько сотен, а то и тысяч.  
Несмотря на то, что одна из наиболее известных методик построенных на модульных тестах - TDD (о ней, чуть ниже), требует писать тесты до написания кода, иногда бывает оправдано написание тестов после кода. Например, вам дали задачу изменить логику работы метода, который был написан год назад, причем разработчиком, который уже уволился. Метод большой, логика сложная. Как удостовериться, что изменяя часть метода, вы не нарушите работу остальной части? Вот здесь и приходят на помощь тесты. Мы пишем тесты, которые тестируют модифицируемый метод, на правильность существующей логики. Раз метод уже год работает, то с высокой вероятностью, эти тесты будут проходить. Вносим правки и удостоверяемся, что те тесты которые проходили на не модифицированном методе, проходят на модифицированном. Другим примером, когда тесты пишутся после кода, может стать ситуация, с поиском ошибки. Устранив все зависимости от внешнего кода и покрыв метод модульными тестами, мы либо найдем ошибку, либо выясним, что виноват не наш метод, а те классы и методы от которых он зависит.  
Ладно, с первым тестом заканчиваем, переходим к проанонсированной чуть выше методологии TDD. С ней, чуть проще в плане определения, вот парочка:  
**Разработка через тестирование** (англ. test-driven development, TDD) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам. ([wikipedia.org](http://ru.wikipedia.org/wiki/%D0%E0%E7%F0%E0%E1%EE%F2%EA%E0_%F7%E5%F0%E5%E7_%F2%E5%F1%F2%E8%F0%EE%E2%E0%ED%E8%E5))  
**TDD** (сокр. от англ. test-driven development — «разработка через тестирование») — это специальна методика разработки ПО, которая основывается на коротких циклах работы, где сначала создаётся тест, а потом функционал. (с [Хабра](http://habrahabr.ru/post/116456/" \t "_blank))  
Все определения, сводятся к цикличности и первичности тестов. Визуально это можно представить вот так:

[](http://1.bp.blogspot.com/-TVOUiU-MJbw/UPKAgCk8hlI/AAAAAAAAA6s/GWdKWL8yoJE/s1600/07.png)

Красная зона, это зона написания нового теста, который проверяет функционал отсутствующий в приложении (тест не проходит, он красный). Затем идет зеленая зона, в рамках которой необходимо написать минимальное количества кода, чтобы тест стал "зеленым". Ну и если у нас получился не очень красивый код, то мы проводим его рефакторинг. Так как у нас весь ранее написанный функционал покрыт тестами, то при написании нового и при рефакторинге, если мы что то и сломаем, то сразу об этом узнаем.  
В качестве примера, чтобы не залезать в сильные дебри, давайте напишем метод решения квадратного уравнения в рамках этой парадигмы. Для этого, в уже имеющемся проекте, я удалю код из метода Calc и наш тестовый метод.

public static double[] Calc(double a, double b, double c)

{

    return null;

}

Пишем первый тестовый случай, возьмем самый простой. когда у нас нет корней:

[TestMethod]

public void NotRootsTest()

{

    double a = 1;

    double b = 4;

    double c = 5;

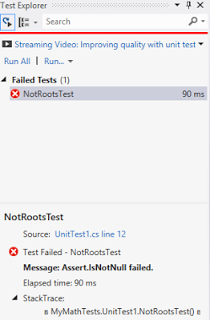
    double[] actual = SqrtRoots.Calc(a, b, c);

    Assert.IsNotNull(actual);

    Assert.IsTrue(actual.GetLength(0) == 0);

}

Т.к. мы только что написали этот тест, то построив решение, мы должны оказаться в красной зоне, с не прошедшим тестом:

[](http://1.bp.blogspot.com/-eKqF1mnNYvk/UPKDZgNExBI/AAAAAAAAA7M/4u_26kb4XMk/s1600/08.png)

Замечательно. Правим код:

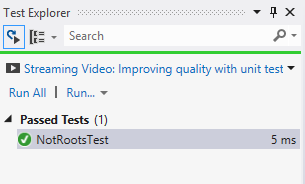
public static double[] Calc(double a, double b, double c)

{

    return new double[0];

}

Обратите внимание, я написал то минимально необходимое количество кода, чтобы тест прошел:

[](http://1.bp.blogspot.com/-WjDgnvW6u_c/UPKD08GeBXI/AAAAAAAAA7U/QgIVxhBK7zo/s1600/09.png)

Замечательно. Подвергать рефакторингу здесь пока нечего, поэтому переходим в следующую итерацию. Опять в красную зону написав новый тест:

[TestMethod]

public void OneRootTest()

{

    double a = 1;

    double b = -4;

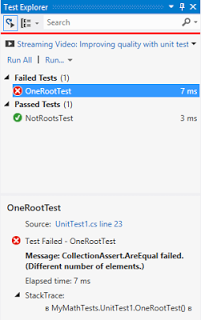
    double c = 4;

    double[] expected = { 2 };

    double[] actual = SqrtRoots.Calc(a, b, c);

    CollectionAssert.AreEqual(expected, actual);

}

[](http://1.bp.blogspot.com/-bkQxHFe84LA/UPKEpDcaz8I/AAAAAAAAA7g/wmBusVqbH4w/s1600/10.png)

Вносим изменения в код. Сразу оговорюсь, мы могли бы довести все до абсурда и сделать проверку, что если нам передали корни 1, -4, 4, то вернуть массив из одного элемента 2. В противном случае, вернуть пустой массив. Но, хотя это и минимальное количество кода, которое необходимо чтобы тест прошел, попахивать это будет дуролейством. Поэтому сразу, правильно обрабатываем такую ситуацию.

public static double[] Calc(double a, double b, double c)

{

    double d = Math.Pow(b, 2) - 4 \* a \* c;

    if (d < 0)

    {

        return new double[0];

    }

    else

    {

        return new double[] { -b / (2 \* a) };

    }

}

Ок. Оба теста прошли. Т.к. у нас в таком, достаточно простом методе две точки выхода, давайте проведем рефакторинг и перепишем его:

public static double[] Calc(double a, double b, double c)

{

    double[] result = new double[0];

    double d = Math.Pow(b, 2) - 4 \* a \* c;

    if (d >= 0)

    {

        result = new double[] { -b / (2 \* a) };

    }

    return result;

}

Пересобираем проект, удостоверяемся, что оба теста проходят. Следующая итерация, с новым тестом:

[TestMethod]

public void DoubleRootsTest()

{

    double a = 1;

    double b = -5;

    double c = 4;

    double[] expected = { 4, 1 };

    double[] actual = SqrtRoots.Calc(a, b, c);

    CollectionAssert.AreEqual(expected, actual);

}

И опять, мы в красной зоне. Правим код:

public static double[] Calc(double a, double b, double c)

{

    double[] result = new double[0];

    double d = Math.Pow(b, 2) - 4 \* a \* c;

    if (d == 0)

    {

        result = new double[] { -b / (2 \* a) };

    }

    else

    {

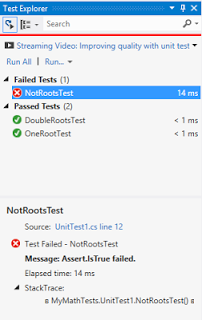
        result = new double[] { (-b + Math.Sqrt(d)) / (2 \* a), (-b -Math.Sqrt(d)) / (2 \* a) };

    }

    return result;

}

И что же мы видим, после запуска тестов:

[](http://2.bp.blogspot.com/-3OWiDhrFKhU/UPKIKcKOKMI/AAAAAAAAA8c/S9rB1x3Fey4/s1600/11.png)

Новый тест проходит, но мы поломали старый тест. Исправляем:

public static double[] Calc(double a, double b, double c)

{

    double[] result = new double[0];

    double d = Math.Pow(b, 2) - 4 \* a \* c;

    if (d == 0)

    {

        result = new double[] { -b / (2 \* a) };

    }

    else if (d > 0)

    {

        result = new double[] { (-b + Math.Sqrt(d)) / (2 \* a), (-b -Math.Sqrt(d)) / (2 \* a) };

    }

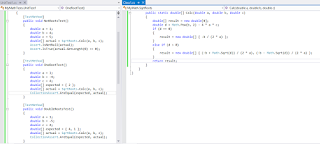
    return result;

}

Перестраиваем проект, удостоверяемся что мы в зеленой зоне. Рефакторинг не требуется, можно уходить на следующую итерацию.

Т.к. статья получается и так уже "много буков", то больше итераций приводить не буду. Мне просто интересно, кроме меня кто видит еще итерации и какие, в этом, достаточно тривиальном, методе?

Небольшая картинка, для холивара (увеличить не забудьте, а то так ничео не видно):

[](http://1.bp.blogspot.com/-BM9rtv8hJrQ/UPKJh7HfrmI/AAAAAAAAA9A/-S4A3_RocbQ/s1600/12.png)

Как видим, для достаточно тривиального метода, количество строк кода необходимых, чтобы его протестировать больше в почти в три раза самого метода. И это основная причина, почему программисты не любят модульные тесты. Кода приходится писать в 3-4 раза больше, чем необходимо для решения задачи. И это не единственный недостаток TDD. Можно привести, для затравки еще несколько:

1. Т.к. тесты и код пишет один и тот же человек, то если он неправильно понял требования, то он написал неправильный тест, который показывает, что корректен неправильный код. Все тесты проходят,  а ничего не работает.

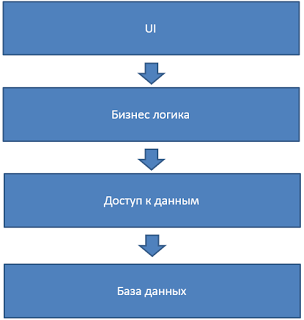
2. При изменении требований, нам придется разбираться, какие тестовые методы перестали работать не потому, что мы "сломали код", а потому, что тесты не удовлетворяют новым требованиям.

3. Не все ситуации, можно протестировать. Например, взаимодействие между процессами (нет, мы конечно посмотрим пример, как в некоторых случаях, можно изолированно тестировать код зависящий от внешней среды, но это работает не всегда и не со всем).

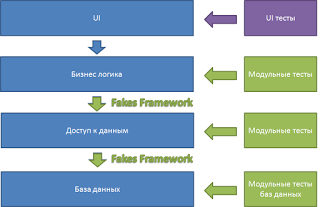
Недостатков вывалил целую кучу, теперь у кого то может возникнуть вопрос, а зачем я тут все это читал, если TDD имеет столько недостатков? Дело в том, что у TDD есть и достоинства. Главное, это снижение времени на отладку. Как только тест перестал проходить, вы сразу можете понять, где это. А если после изменения, которое вроде ничего не должно поломать, все тесты упали, то проще не отлаживать, а откатиться к стабильно работающей версии и попробовать обдумать проблему заново. Как было сказано на одной из конференций (вроде DevCon и вроде Александр Яковлев): "Клиентское приложение, да что там можно протестировать, но вот уровень бизнес-логики, должен быть покрыт тестами процентов на 85". Если во фразе что и изменил, то не много. Т.е. применять TDD надо там, где вы сможете от этого выграть. Сложная бизнес-логика - TDD, критическая обработка данных - TDD, управление ядерным реактором - TDD...

Если еще не совсем устали, то давайте еще коротенько про Fakes Framework и закругляемся.

Итак, классическое N-уровневое приложение будет иметь архитектуру вида:

[](http://2.bp.blogspot.com/-MfDpFis88nE/UPKZQ7lZbcI/AAAAAAAAA9k/8k1g3rbLnmA/s1600/13.png)

И тут возникает вопрос, как быстро и изолированно протестировать уровень доступа к данным? Ведь, чтобы работали его методы, необходимо наличие БД. Как протестировать уровень бизнес-логики, в отрыве от уровня доступа к данным? И тут нам на помощь и приходит Fakes Framework. Он позволяет изолировать классы друг от друга:

[](http://3.bp.blogspot.com/-RHlV4rC-DT0/UPKa6_SlG1I/AAAAAAAAA-I/tWl3hQsJ4h0/s1600/14.png)

UI тесты, специально выделены цветом, т.к. они это отдельная сказка, про которую расскажу в другой раз.

Ну и т.к. при решении квадратного уравнения, у нас подменять кроме методов Pow и Math нечего, а их подменять смысла нет, то давайте рассмотрим другой пример.

Пусть, стоит задача разбирать файлы логов следующего вида:

13.01.2013 12:45:37 сообщение 1

13.01.2013 12:45:37 сообщение 2

13.01.2013 12:45:38 сообщение 3

13.01.2013 12:45:40 сообщение 4

13.01.2013 12:45:40 сообщение 5

Создаем Class Library, в нем класс, для хранения данных, и метод разбора файлов.

public class LogMessage

{

    public DateTime Time { get; set; }

    public string Message { get; set; }

}

public class LogParser

{

    public static LogMessage[] Parse(string p\_filePath)

    {

        List<LogMessage> result = new List<LogMessage>();

        StreamReader reader = new StreamReader(p\_filePath);

        string[] messages = reader.ReadToEnd().Split('\n');

        foreach (var msg in messages)

        {

            result.Add(

                new LogMessage()

                {

                    Time = DateTime.Parse(msg.Remove(19)),

                    Message = msg.Substring(20)

                });

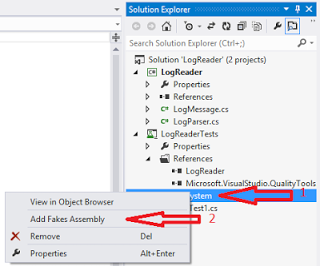
        }

        return result.ToArray();

    }

}

Вот метод, для разбора файла, нам и придется тестировать. Добавляем тестовый проект, делаем из него ссылку на наш проект с LogParser-ом. А дальше, самое интересное. Мы хотим подменить работу с классом StreamReader из System.dll. Для этого кликаем на ней правой клавишей мыши и добавляем ее Fake версию:

[](http://1.bp.blogspot.com/-Ct1tTyd9gSs/UPKkK_30sVI/AAAAAAAAA-s/wl8tUTit34w/s1600/15.png)

 Все, пишем тестовый метод:

[TestMethod]

public void TestMethod1()

{

    // Создаем контекст

    using (ShimsContext.Create())

    {

        // Переопределяем конструктор

        System.IO.Fakes.ShimStreamReader.ConstructorString = (@this, path) => { };

        // Указываем, какой метод и чем мы будем заменять

        System.IO.Fakes.ShimStreamReader.AllInstances.ReadToEnd = sr =>

        { return "13.01.2013 12:45:37 сообщение 1\n13.01.2013 12:45:37 сообщение 2\n13.01.2013 12:45:38 сообщение 3"; };

        // А дальше, как в обычном тестовом методе

        LogMessage[] expected =

        {

            new LogMessage { Time = DateTime.Parse("13.01.2013 12:45:37"), Message = "сообщение 1" },

            new LogMessage { Time = DateTime.Parse("13.01.2013 12:45:37"), Message = "сообщение 2" },

            new LogMessage { Time = DateTime.Parse("13.01.2013 12:45:38"), Message = "сообщение 3" }

        };

        LogMessage[] actual = LogParser.Parse("qwerty");

        Assert.AreEqual(expected.Length, actual.Length);

        for (int i = 0; i < actual.Length; i++)

        {

            Assert.AreEqual(expected[i].Time, actual[i].Time);

            Assert.AreEqual(expected[i].Message, actual[i].Message);

        }

    }

}

На мой взгляд, синтаксис достаточно интуитивный, а написав пару тестов, все это будет восприниматься естественно. Обратите внимание, что вызов метода LogParser.Parse , самый что ни на есть обычный, всю работу по подмене класса StreamReader мы провели предварительно и он об этом даже не догадался. Минусы у такого подхода конечно есть, заключаются в более медленном выполнении тестов, в необходимости написания методов, заменяющих оригинальные. Но, есть существенное достоинство: изолированное тестирование. У меня нет файла с тестовыми данными, но я протестировал его загрузку и разбор. Причем, никто не мешает, нам писать тесты проверяющие ситуацию, когда по указанному пути файла нет. В этом случае, в заменяемом конструкторе достаточно вызвать FileNotFoundException. И посмотреть, как наш метод будет отрабатывать в этой ситуации. Ну и так далее.