В процессе написания ПО у меня возникло понимание о целесообразности применения unit-тестов.

Так в моей практике появилось несколько проектов, в которых мне довелось писать unit-тесты, каждый из которых выполнял определенную роль – поиск ошибок в основных алгоритмах кода, нагрузочное тестирование и отладка бэкенда веб-приложения. В каждой из поставленных задач unit-тесты оказались эффективны, позволив существенно сократить время работы и обеспечить своевременное обнаружение ошибок кода.

Согласно данным[1] исследований, цена ошибки в ходе разработки и поддержании ПО экспоненциально возрастает при несвоевременном их обнаружении.

На представленном рисунке видно, что при выявлении ошибки на этапе формирования требований мы получим экономию средств в соотношении 200:1 по сравнению с их обнаружением на этапе поддержки.

Среди всех тестов львиную долю занимают именно unit-тесты. В классическом понимании unit-тесты позволяют быстро и автоматически протестировать отдельные части ПО независимо от остальных.

Рассмотрим простой пример создания unit-тестов. Для этого создадим консольное приложение Calc, которое умеет делить и суммировать числа.

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

namespace Calc
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

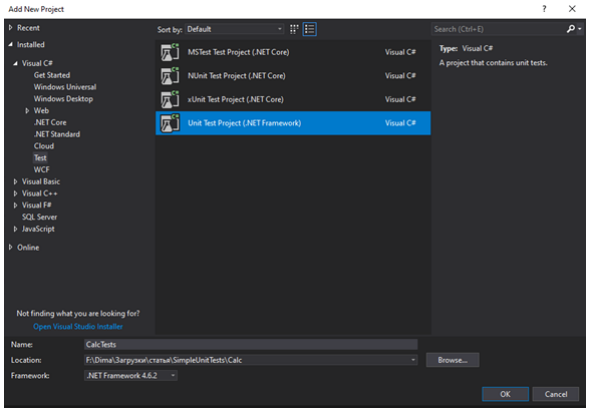
Добавляем класс, в котором будут производиться математические операции.

```
using System;

namespace Calc
{
    /// <summary>
    /// Выполнение простых математических действий
над числами
    /// </summary>
    public class Calculator
    {
        /// <summary>
        /// Получаем результат операции деления (n1
/ n2)
        /// </summary>
        /// <param name="n1">Первое число</param>
        /// <param name="n2">Второе число</param>
        /// <returns>Результат</returns>
        public double Div(double n1, double n2)
        {
            // Проверка деления на "0"
            if (n2 == 0.0D)
                throw new DivideByZeroException();
            return n1 / n2;
        }

        /// <summary>
        /// Получаем результат сложения чисел и их
увеличения на единицу
        /// </summary>
        /// <param name="n1"></param>
        /// <param name="n2"></param>
        /// <returns></returns>
        public double AddWithInc(double n1, double
n2)
        {
            return n1 + n2 + 1;
        }
    }
}
```

Так, в методе Div производится операция деления числа n1 на число n2. Если передаваемое число n2 будет равняться нулю, то такая ситуация приведет к исключению. Для этого знаменатель этой операции проверяется на равенство нулю.

Метод AddWithInc производит сложение двух передаваемых чисел и инкрементацию полученного результата суммирования на единицу.

На следующем шаге добавим в решение проект тестов.



Пустой проект unit-тестов:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace CalcTests
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

Переименуем наш проект: «SimpleCalculatorTests». Добавляем ссылку на проект Calc.

В проекте Calc содержатся 2 метода, которые надо протестировать на корректность работы. Для этого создадим 3 теста, которые будут проверять операцию деления двух чисел, операцию деления на нуль и операцию сложения двух чисел и инкрементацию полученной суммы.

Добавляем в проект тест для проверки метода AddWithInc.

```
/// <summary>
/// Тест проверки метода AddWithInc
/// </summary>
[TestMethod]
public void AddWithInc_2Plus3Inc1_Returned6()
{
    // arrange
    var calc = new Calculator();
    double arg1 = 2;
    double arg2 = 3;
    double expected = 6;
    // act
    double result = calc.AddWithInc(arg1, arg2);
    // assert
```

```
    Assert.AreEqual(expected, result);
}
```

В тесте создаются 3 переменные — это аргументы, передаваемые в метод AddWithInc, и ожидаемый результат, возвращаемый этим методом. Результат выполнения метода будет записан в переменную result. На следующем шаге происходит сравнение ожидаемого результата с реальным числом метода AddWithInc. При совпадении результата с ожидаемым числом, то есть числом 6, тест будет считаться положительным и пройденным. Если полученный результат будет отличаться от числа 6, то тест считается проваленным.

Следующим тестом мы будем проверять метод Div

```
[TestMethod]
public void Div_4Div2_Returned2()
{
    // arrange
    var calc = new Calculator();
    double arg1 = 4;
    double arg2 = 2;
    double expected = 2;
    // act
    double result = calc.Div(arg1, arg2);
    // assert
    Assert.AreEqual(expected, result);
}
```

Аналогичным образом создаются два аргумента и ожидаемый результат выполнения метода Div. Если результат деления 4/2 в методе равен 2, то тест считается пройдённым. В противном случае – не пройденным.

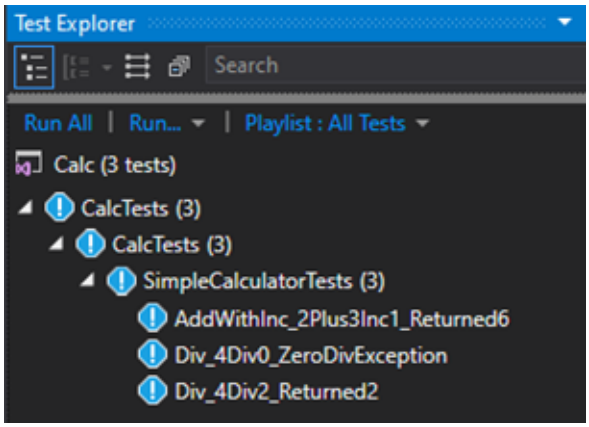
Следующий тест будет проверять операцию деления на нуль в методе Div.

```
[TestMethod]
[ExpectedException(typeof(DivideByZeroException),
    "Oh my god, we can't divison on zero")]
public void Div_4Div0_ZeroDivException()
{
    // arrange
    var calc = new Calculator();
    double arg1 = 4;
    double arg2 = 0;
    // act
    double result = calc.Div(arg1, arg2);
    // assert
}
```

Тест будет считаться пройденным в случае возникновения исключения DivideByZeroException – деление на нуль. В отличии от двух предыдущих тестов, в этом тесте нет оператора Assert. Здесь обработка ожидаемого результата

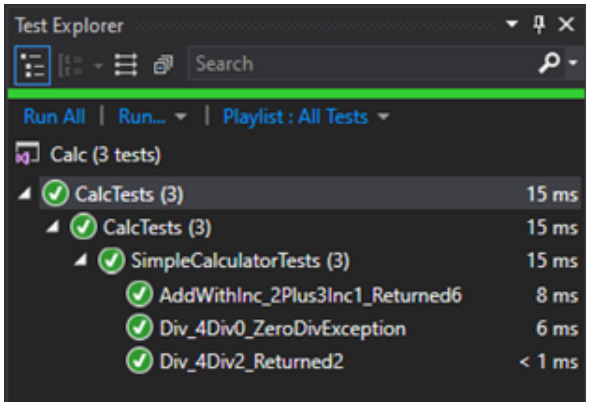
производится с помощью атрибута «ExpectedException». Если аргумент 2 равен нулю, то в методе Div возникнет исключение – деление на нуль. В таком случае тест считается пройденным. В случае, когда аргумент 2 будет отличен от нуля, тест считается проваленным.

Для запуска теста необходимо открыть окно Test Explorer. Для этого нажмите Test -> Windows -> Test Explorer (Ctrl+E,T). В появившемся окне можно увидеть 3 добавленных теста:

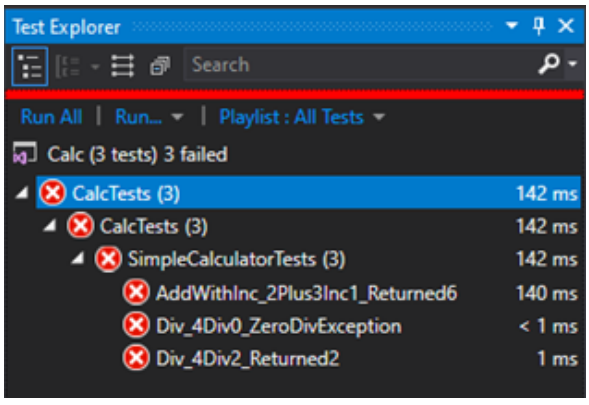


Для запуска всех тестов нажмите Test -> Run -> All tests (Ctrl+R,A).

Если тесты выполнятся успешно, в окне Test Explorer отобразятся зеленые пиктограммы, обозначающие успешность выполнения.



В противном случае пиктограммы будут красными.



Unit-тесты имеют обширную, строго не регламентированную область применения — зачастую фантазия самого автора кода подсказывает решение нестандартных задач с помощью этого инструмента.

Случай написания тестов для бэкенда веб-приложения в моей практике является не совсем стандартным вариантом применения unit-тестов. В данной ситуации unit-тесты вызывали методы контроллера MVC-приложения, в то же время передавая тестовые данные в контроллеры. Далее в режиме отладки шаг за шагом выполнялись все действия алгоритма. В этом случае применение тестов позволило произвести быструю отладку бэкенда веб-приложения.

Существуют случаи, когда модульные тесты применять нецелесообразно. Например, если вы веб-разработчик, который делает сайты, где мало логики. В таких случаях имеются только представления, как, например, для сайтов-визиток, рекламных сайтов, или, когда вам поставлена задача реализовать пилотный проект «на посмотреть, что получится». У вас ограниченные ресурсы и время. А ПО будет работать только один день – для показа руководству.

Сжатые сроки, малый бюджет, размытые цели или довольно несложные требования – случаи, в которых вы не получите пользы от написания тестов.

Для определения целесообразности использования unit-тестов можно воспользоваться следующим методом: возьмите лист бумаги и ручку и проведите оси X и Y. X — алгоритмическая сложность, а Y – количество зависимостей. Ваш код поделим на 4 группы.

1. Простой код (без каких-либо зависимостей)
2. Сложный код (содержащий много зависимостей)
3. Сложный код (без каких-либо зависимостей)
4. Не очень сложный код (но с зависимостями)

Первое – это случай, когда все просто и тестировать здесь ничего не нужно.

Второе – случай, когда код состоит только из плотно переплетенных в один клубок реализаций, перекрестно вызывающих друг друга. Тут неплохо было бы провести рефакторинг. Именно поэтому тесты писать в этом случае не стоит, так как код все равно будет переписан.

Третье – случай алгоритмов, бизнес-логики и т.п. Важный код, поэтому его нужно покрыть тестами.

Четвертый случай – код объединяет различные компоненты системы. Не менее важный случай.

Последние два случая — это ответственная логика. Особенно важно писать тесты для ПО, которые влияют на жизни людей, экономическую безопасность, государственную безопасность и т.п.

Подводя итог всего описанного выше хочется отметить, что тестирование делает код стабильным и предсказуемым. Поэтому код, покрытый тестами, гораздо проще масштабировать и поддерживать, т.к. появляется большая доля уверенности, что в случае добавления нового функционала нигде ничего не сломается. И что не менее важно — такой код легче рефакторить.