Unit Testing with the Visual Studio Test Framework

Objectives

The aim of this tutorial is to provide a practical introduction to unit testing and the Visual Studio Test Framework.

Prerequisites

You should have read the accompanying Designing and Building Components lecture notes before attempting this tutorial.

Lab Setup

Before attempting the exercises in this document, you should download the Visual Studio solution for this practical from the module Web site. The solution is supplied in compressed form as a ZIP archive. You should extract the files from this archive into your chosen location on the local hard drive of your computer.

You should then view the README which goes over some of the basic concepts of testing

Unit Testing

Unit testing is a form of *developer testing* of software which aims to test individual units of source code in isolation to determine if they are fit for use. A *unit* is viewed as the smallest testable part of an application, typically individual methods, properties and functions of classes, or a class in its entirety. *Unit tests* are short code fragments created and run by programmers to test individual units and each test should be independent of any others.

The goal of unit testing is to test only the unit under test and not other pieces of code that the unit depends on. For this reason it is often necessary to introduce stubs or mock objects to ensure that a unit really is tested in isolation.

The Visual Studio Testing Framework

The Visual Studio testing framework supports many different forms of testing, including unit testing. Throughout this lab we shall use the Visual Studio testing framework to build an execute unit tests.

To make use of the Visual Studio unit testing framework all you need to do is add a new Test Project to your Visual Studio solution

Exercise 1: Creating a Unit Test Project

Download and extract the Visual Studio solution that accompanies these lab exercises from the module website. When you open the solution you should see that it contains a class library project called MathsLibrary which contains a single Maths class. We shall use this class to build our first unit tests, but before we can create any tests we need to add a Test Project to our solution.

Add a new project to the existing solution (either by right clicking the solution in the *Solution Explorer* and selecting *Add*, or by opening the *File* menu and selecting *Add*). Choose to add a new project and in the *Add New Project* dialog that appears (Figure 1).

Search for “MSTest” (ensure search filters are set to ‘C#’ or ‘All Languages’, ‘All Platforms’ and “All Project Types” or ‘Test’)

Select the C# *MSTest Test Project (.NET Core)* then click *Next* and call it `UnitTest`. Click *Create*.

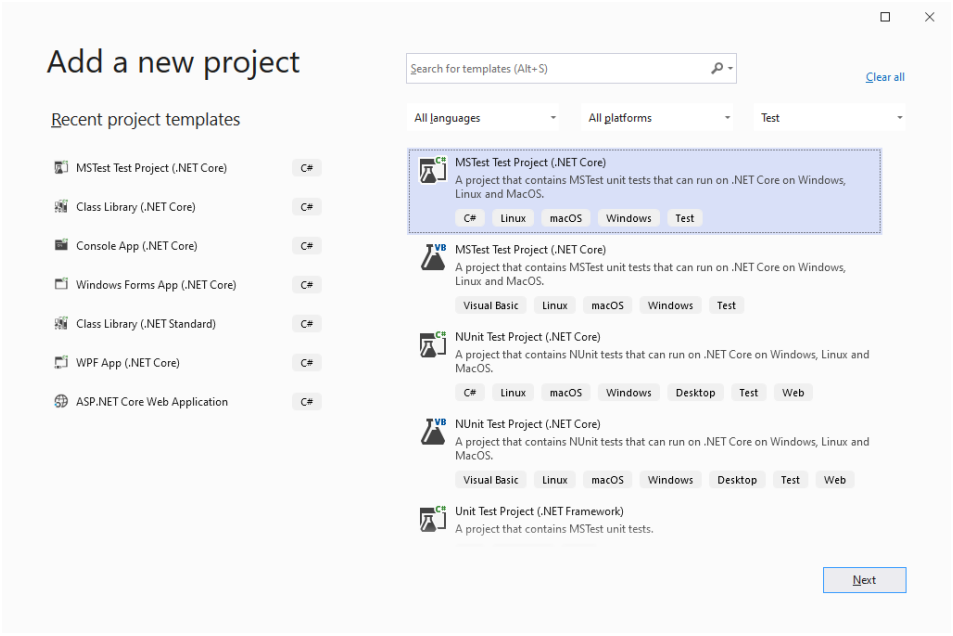

Figure 1: Adding a Unit Test project with the Add New Project dialog

A new project will be added to the solution in which you can create unit tests. To get you started, Visual Studio automatically creates a `UnitTest1.cs` class containing a skeleton unit test class implementation. Open this class in Visual Studio and examine the code it contains.

In the *Solution Explorer*, right click the *Dependencies* for your `UnitTest` project, click 'Add Project Reference' and a reference to the `MathsLibrary` assembly by checking it in the list and clicking Ok.

Visual Studio Unit Test Classes

In the Visual Studio unit testing framework, and in other test frameworks such as NUnit, you will find that unit tests are implemented as methods inside a standard C# class. If you examine the code of the `UnitTest1.cs` class you will find that it looks similar to Figure 2.

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

    }
}
```

Figure 2: A basic unit test class

The `[TestClass]` attribute applied to the class informs Visual Studio that this class contains unit tests and the test framework should look inside it to find tests that can be run.

The method `TestMethod1` is a unit test. We can tell this because it has the `[TestMethod]` attribute applied to it. Any method that implements a unit test must have the `[TestMethod]` attribute applied otherwise it won't be listed as a test that can be run.

There are a number of other attributes from the Visual Studio framework that can be applied to test methods and test classes.

View the README in the lab files if you haven't already and then check out the Microsoft documentation on the Visual Studio testing framework for more details:

https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting

To implement a unit test you just need to add the appropriate code to perform the test into your test method. Unit tests follow the general form of:

- Establish the pre-conditions (i.e. perform the necessary setup required before the test can take place)
- Perform the actual test
- Verify the post-conditions (i.e. check that what you expected to happen has happened)

Exercise 2: Testing the Add() method

In this exercise we implement a test to verify operation of the `Add()` method. You should do this by modifying the `TestMethod1()` in the UnitTest1 class that Visual Studio created.

The `Add()` method expects to be given two floating point arguments, so the pre-condition for this test is that two floating point values are available. Rename `TestMethod1()` to `AdditionTest()` Define two `float` variables in `AdditionTest()`, one of which is assigned the value 2, and the other is assigned the value 3. You have now established the pre-conditions of the test.

Define a float variable called `actual`. Add a line of code to call the static `Maths.Add()` method, passing in the two variables you have defined and storing the result in the actual variable. This line of code is actual test of the `Add()` unit. You may like to add a `using MathsLibrary` statement or instead call `MathsLibrary.Maths.Add()`.

We now need to establish the post-conditions of the test – i.e. verify that the expected result(s) were obtained. In our case, adding together 2 and 3 should produce the value 5, so the post-condition that we want to verify is that actual has the value 5.

The Visual Studio Testing Framework provides the static Assert class to help us in verifying test post-conditions:

https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert

It has a number of methods such as `IsNull`, `IsNotNull`, `IsTrue`, `IsFalse`, `AreEqual`, etc that we can use to verify that the results of the test are what we expected. Two other assertion classes exist- `CollectionAssert` which allows you to perform verification on collections of data, and `StringAssert` which allows you to verify strings. In writing tests you should use the assertion class most appropriate to the type of data you are working with.

In our case we need to use the `Assert.AreEqual()` method to verify that the actual variable has a value equal to 5.

Define a `float` variable called expected and assign it the value 5.

Then add a line of code at the end of your `TestMethod1()` using `Assert.AreEqual()` to verify that the actual equals the expected result of 5.

Compile your solution and make sure it builds correctly.

Running Tests

The Test menu inside Visual Studio gives you access to a number of features that enable you to run and evaluate tests. One of the most important of these is the Test Explorer window (Figure 3) which displays a list of tests found in your solution and allows you to run or debug one or more tests at a time. You can open the Test Explorer window by selecting the Test menu or View menu and choosing the Test Explorer option (or hit CTRL+E then release and hit T for the shortcut).

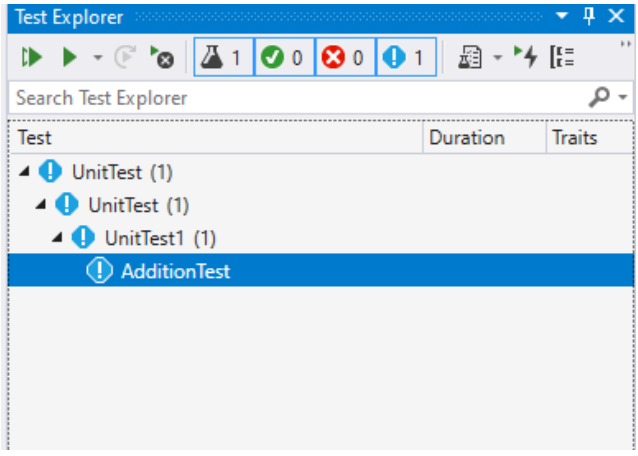

Figure 3: The Test Explorer window

Exercise 3: Running the Add() test

To run a test, highlight it in the Test Explorer window and select the *Run* option (play button). Alternatively, right-click it and select *Run*.

To debug a test, select the *Run* dropdown and click *Debug*, or right-click a test and select *Debug* from the content menu.

To run or debug multiple tests in one test session, select a project or class or several individual tests or projects from the list before selecting run or debug; or click the Run All Tests button.

Execute your test for the `Add()` method functionality by selecting `AdditionTest` in the Test Explorer windows and clicking *Run*. The test should run and, when it completes, the Test Explorer window should update to show that results of the testing.

A green circle with a white tick will be displayed next to tests that have passed whilst tests that have failed will show a red circle with white cross. Indicators at the top of the Test Explorer show the number of tests and the number passed or failed. There are also options to re-run only the failed tests, which may be useful when making code changes to fix errors.

Writing Useful Tests

What code should you implement to perform an actual test? The answer depends on what your unit and what it is supposed to do. As an absolute minimum you should write a test that verifies that it works as expected when you call it in the manner that you normally expected it to be called. You should also test how it behaves for the more unusual 'corner cases' and perhaps also how it behaves when presented with unexpected input.

Is the test we wrote for the `Add()` method sufficient to fully test it's behaviour? Let's explore this further by testing the `Divide()` method.

Exercise 4: Testing the Divide() method

Add a new method to your `UnitTest1.cs` class and tag it with the `[TestMethod]` attribute to denote that it is a test. Note that the name you give the method is the name that is used for the test in the Test Explorer window, so it is a good idea to give it a sensible name. We're going to use this method to test the `Divide()` operation, so calling it `Divide`, or `DivideTest` would make sense.

What test cases do we need to test `Divide()`? We should start by testing the normal expected operation. Using the code you wrote for the Add() test in Exercise 3 as a template, declare float variables to hold the values of 12 and 4 and an expected variable which holds the value of 3. Call the `Divide()` method and use the Assert.AreEqual() operation to verify that the result of the `Divide()` is 3.

What other cases might we want to test? How about dividing a positive number by a negative number? Or dividing a negative number by a negative number?

Add code to test the cases of 12 divided by -4 to return a result of -3, and -12 divided by -4 to return a result of 3.

Have we now fully tested our `Divide()` operation?
No.

The tests we have produced thus far are all good but only test the **expected** operation. What happens if someone attempts to divide a number by zero? Mathematically speaking, dividing any number by zero results in infinity. In floating point arithmetic in .NET, infinity is represented by a special value of the float class and we can test whether a floating point number is infinite by calling:

```
float.IsInfinity(floatValue)
```

where `floatValue` is the `float` variable we want to test to see whether it is infinite.

To make your unit tests more complete, implement a test for division by zero to ensure that the value infinity is returned.

Implement:

- Multiply operation tests
- Tests to test all use cases of your maths operations
- Add a dot product operation that makes use of add and multiply to calculate the dot product of two arrays of integers
- Tests to test all use cases of your dot product operation

What about the tests you wrote for your dot product operation?

They likely rely directly on the other methods that your dot product operation uses, which means that **they aren't individual unit tests** – they have a dependency.

This isn't testing your unit in isolation and so is not well decoupled. You should be able to test your methods without another method being the cause of a failure. In a future lab we'll take a look at how we can do this but it's important you understand the basics in this lab before we get to that point.

To finish this lab, make sure that you have read through the README slides and implement a range of tests that cover the considerations identified there.

Don't do it yet… but we will be adding another Class to MathsLibrary called `StoredMaths.cs` that allows you to perform the following behaviour:

- Create an instance of StoredMaths
- Call Add(float) on the instance and the float is added to a value stored in the StoredMaths object (initially zero)
- The result is returned to the caller

In this way, you could add 4 to 5 by calling

```
StoredMaths storedMaths = new StoredMaths();
storedMaths.Add(4f);
float result = storedMaths.Add(5f);
```

We will also additional behaviour to StoredMaths to:

- Clear the stored value
- Store a value, overwriting anything there currently
- Subtract a given value from the stored value
- Multiply the stored value by a given value
- Divide the stored value by a given value

Create a StoredMaths class and add in each of the relevant method signatures but keep their implementation as returning a 'NotImplementedException'.

Before you create the actual method functionality as explained above, add a new TestClass to UnitTest and add tests for each of the methods you have created. Decide how you can use the TestInitialise and other Attributes to make your tests clearer and cleaner.

Add test functionality for ALL possible test cases. It should all fail.

Now implement your methods so all of your tests pass. You have just followed Test Driven Development – considering all of the possible error cases and tests before implementation so you aren't tied to the implementation details when you write your tests and you're more likely to catch all eventualities!