Software Engineering 3
Practical No 5 - Unit Testing

1. Unit Testing Overview

- Writing unit tests will enable you test your code to ensure it works as expected. Test code, if written correctly, saves you a lot of time and effort in the long run in ensuring your code runs as expected.

- .NET C# has a vast number of unit testing frameworks available. For the scope of this document, we'll focus on using the integrated Visual Studio 2019 Unit Testing Framework (namespace: Microsoft.VisualStudio.TestTools.UnitTesting) which should be available for all versions of VS2019.

2. Creating a new Unit Test Project

1) Extract the **UnitTestExercise.zip** archive and open up the UnitTestExercise.ln solution. This is a .NET C# class library with two projects called *Statistics.Lib* and *TextParser.Lib*. As you may have already guessed, the former consists of methods to do mathematical calculations whilst the latter is used to parse text. Each library consists of a single class. There is no need to modify any code within the UnitTestExercise project. The goal here is to write unit tests for each method in this library so that any future amendments and/or changes to either class can be easily tested in an automated manner. We will focus on writing a test for the Calculate.cs class for now.

2) To begin, lets add a new Unit Test Project to the existing solution (Figure 1). Call the new project **Statistics.Test**.

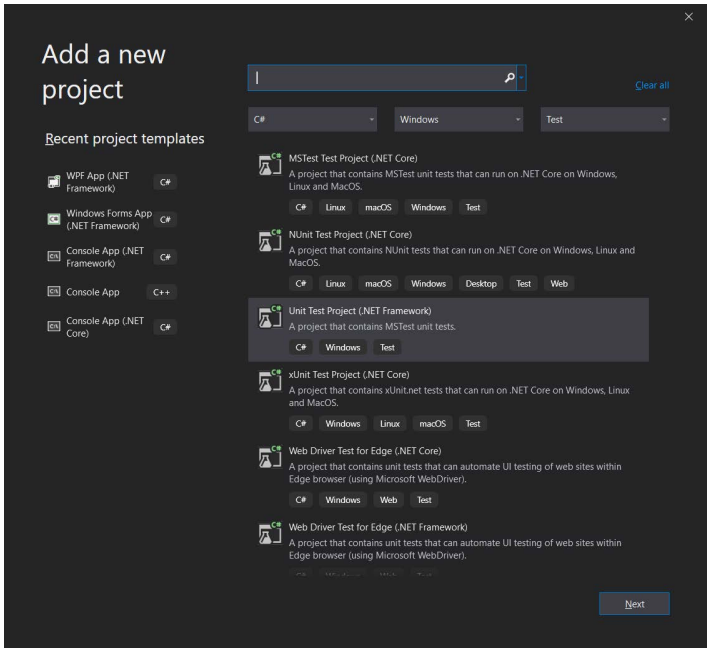

Figure 1 - Create a new Unit Test Project

3) Since we wish to test the Calculate.cs class located in the Statistics.Lib project, we need to add a reference to this class library. Right-click the *References* folder and add Statistics.Lib project to the Statistics.Test project (Figure 2).

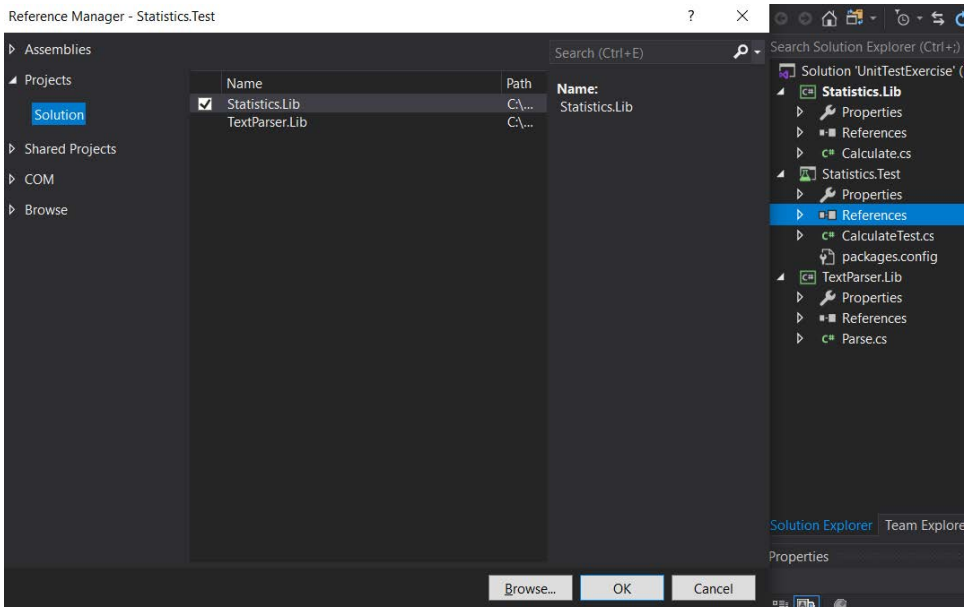

Figure 2 - Adding a reference to Statistics.Lib

4) Rename UnitTest1.cs to CalculateTest.cs. The solution structure should now look similar to Figure 3.

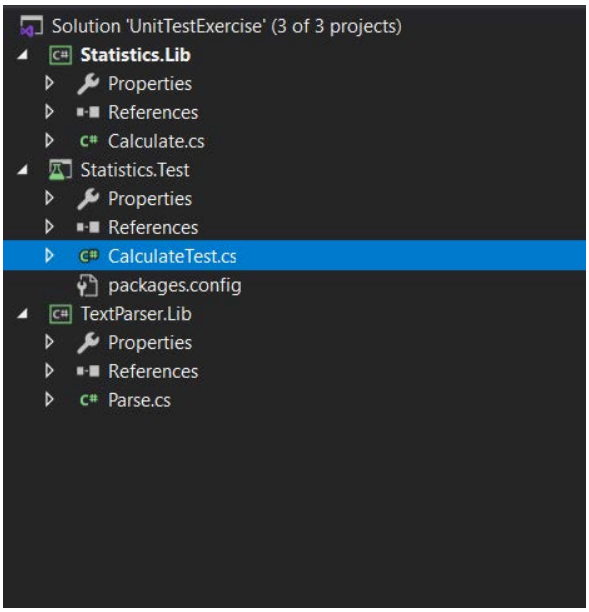

Figure 3 - Project Structure After Adding Statistics.Test

5) In order to use the test class, two minimum attribute tags are required:
   - [TestClass] : Used to define a test class.
   - [TestMethod] : Used to define any test methods within the class which should be ran when testing takes place.

6) Currently, our test class is rather empty and incomplete. The goal of this exercise is to write an automated unit test for each of the methods found in both Calculate.cs and TextParser.cs class. The next section provides a step-by-step guide on creating a test method for the Summation() method.

3. Writing Our First Test

1) Within CalculateTest.cs, rename TestMethod1() to something more sensible such as SummationTest().

2) Generally, the purpose of each test will be to provide the method under test with some form of input (e.g. string, int, double etc), define the expected outcome and assert whether or not the outcome is correct.

3) The purpose of the Summation() method is to simply calculate the total sum of a double array. Let us first define an array of double with the values {1, 2, 3...20} with the name t:

```
double[] t = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };
```

4) Next, we need to define the expected outcome of the Summation() method. The summation of {1, 2, 3...20} can be calculated by **20(20+1)/2 = 210**. So we define a double named expectedResult like so:

```
double expectedResult = 210;
```

5) The actual result (which will be computed by our Summation() method) should be assigned to another double like so:

```
double actualResult = Calculate.Summation(t);
```

6) Finally, we may then use the Assert.AreEqual method, provided by Microsoft's .NET unit testing library, to compare the expected result against the actual result. Your implemented method should look similar to this:

```
[TestMethod]
public void SummationTest()
{
    double[] t = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };
    double expectedResult = 210;
    double actualResult = Calculate.Summation(t);

    Assert.AreEqual(expectedResult, actualResult);
}
```

7) Having implemented a test method, we can finally run the first test! Go to **Test** > **Windows** > **Test Explorer** (Figure 4).

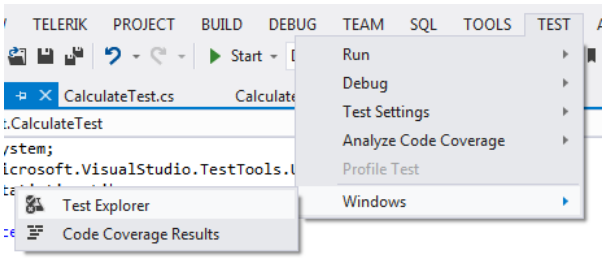

Figure 4 - Display Test Window

8) Within the Test Explorer panel, you can either click the blue "Run All" link to run tests or enable "Run Tests After Build" button (circled red in Figure 5) so that automated testing takes place each time you build your project.

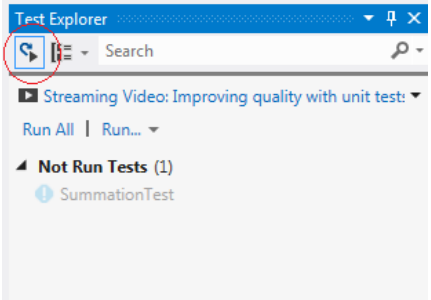

Figure 5 - Run Tests After Build Option

9) Assuming everything has been implemented correctly, the test should pass. To see an example of a failed test, simply change the double expectedResult = 210; to any other value. As shown in Figure 6, the test will fail since the Summation method returns 210 (correct) whilst our test code expected a value of 200. This would be an example of an incorrectly written *test method* since the Sumation() method itself is correct!

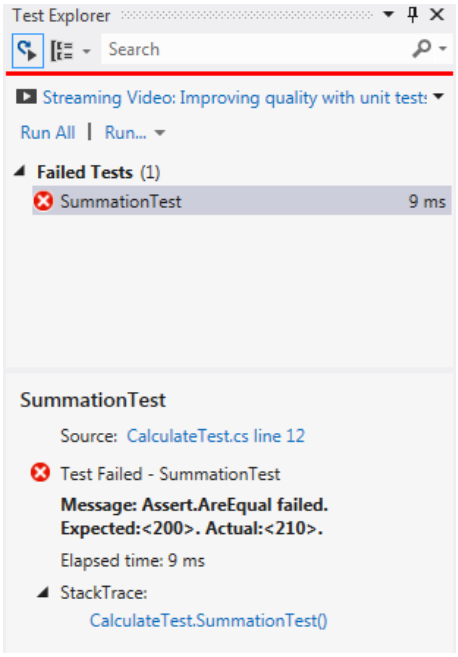

Figure 6 - Example of Failed Test

It's worth noting at this point that although the SummationTest() method we have written seems a waste of time this would not be the case in more complex methods. Furthermore, suppose the Summation() method is accidently or intentionally updated in an incorrect manner in the future. The unit test we've written for this method (which may run automatically when performing a compilation) would instantly spot such issues thus reducing the chance of hidden bugs in the software.

4. Calculate.cs Unit Tests Exercise

- The previous section has provided a step-by-step example of implementing test code for the Summation() method.

- The goal of this exercise is to write a unit test for each of the other methods found in Calculate.cs and ensure the unit test passes. Table 1 defines the test input for each method and expected results of each test. To complete this exercise, all tests must pass.

Table 1 - Calculate.cs Test Unit Exercise

Calculate.cs Method Name	**Test Input**	**Expected Result**
Mean(double[] t)	double[] t = 2, 4, 6, 8, 10, 12	7
RandomNumber(int min, int max)	int min = 50 int max = 100	57
Variance(double[] t)	double[] t = 17, 32, 34, 65, 67, 99, 102	1110.95
StandardDeviation(double[] t)	double[] t = 4, 12, 14, 21, 22, 25, 27, 28, 34, 42, 58, 67	18.46126
NormalDistribution(double[] t, double x)	double[] t = 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 double x = 10	0.06499085

- For the Variance(...), StandardDeviation(...) and NormalDistribution(...) test, you will need to provide a third parameter known as **Delta** when calling the Assert.AreEqual method.

- Delta defines the required level of accuracy in comparing the expected and actual result.

- For testing of Variance(), a delta of 0.01 should work whilst a delta of 0.00001 will satisfy StandardDeviation() and NormalDistribution().

- See MSDN for further details on Delta: http://msdn.microsoft.com/en-us/library/ms243458.aspx

5. Parse.cs Unit Tests Exercise

- The goal of the exercise is similar to the previous one however, instead of working with doubles, we'll be working with strings when writing our tests.

- Begin by looking at Parse.cs which consists of some methods for parsing information about a simple string input using simple regular expressions.

- To given an example, the method ExtractVowels(string text) will return a string of all the vowels which exist based on the input given:

```
TextParser.ExtractVowels("hello world one two three");
```

would return the result of:

```
e o o o e o e e
```

- Each result is separated by a single space.

- To give another example, the method ExtractLowerCaseWords(string text) will return a string of all words which start with lower case:

```
TextParser.ExtractLowerCaseWords("My name is John Doe");
```

would return the result of:

```
name is
```

- Begin the exercise by adding a new Unit Test Project to the existing solution called **TextParser.Test**

- The structure of your project should look similar to Figure 7.

- Similar to the previous exercise, write a test for each method contained in Parse.cs. Unlike the tests for Calculate.cs, you must define your own expected result for each the methods given in Table 2.

- Remember to add a reference to TextParse.Lib in your new unit test project otherwise you can't call the methods in Parse.cs!

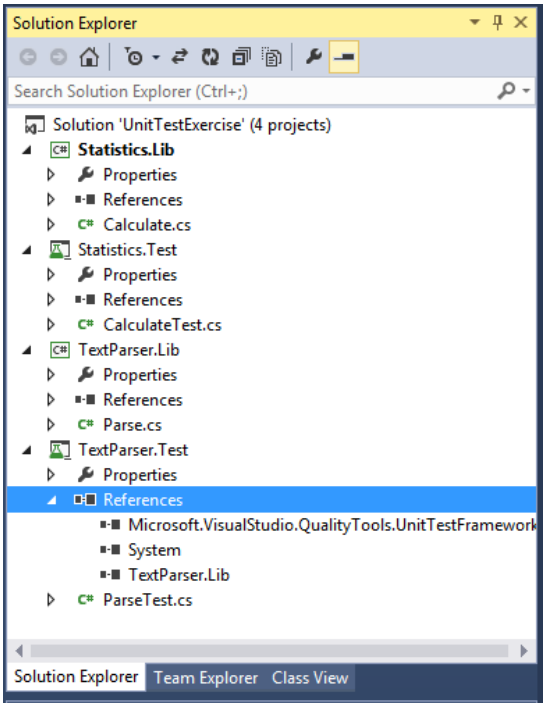

Figure 7 - Project Structure after adding TestParse.Test

Table 2 - TextParse.cs Unit Test Exercise

TextParser.cs Method	**Test Input**	**Expected Result (define your own)**
ExtractVowels(string text)	string text = "To be or not to be that is the question"	
ExtractConjunctions(string text)	string text = "and he is neither here nor there or anywhere else"	
ExtractUpperCaseWords(string text)	string text = "John Doe went to visit the National Museum in London"	
ExtractLowerCaseWords(string text)	string text = " the Quick brown fox Jumps over the lazy Dog""	

6. Additional Points to Consider

- How does one assert that two collections data, e.g. List<T>, are equal? Note that Assert.Equal may not work... Think about why it doesn't work and research the solution.

- Debugging Unit Test code is useful in ensure the test code itself works as expected. Have look at how one may go about debugging unit test code.

- MSDN Unit Testing Basics which provides another example of writing unit tests along with creating mock objects - http://msdn.microsoft.com/en-us/library/hh694602.aspx

- Unit Testing Exceptions - http://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.expectedexceptionattribute.aspx