COMP 3111: Software Engineering

Unit Testing in ASP.NET Using NUnit
Tutorial and Lab Notes

It is usually good practice to have automated unit tests while developing your code to ensure the methods you write are doing what *you* expect them to do. Doing so helps find bugs early in the development cycle and saves time in the long run. The developers who are implementing the methods usually write the unit tests. Once constructed, a unit test that tests the behaviour of a method can be run multiple times for different input sets to verify the method's behaviour before it becomes part of the system and goes into production. Usually a method is tested in isolation from the other methods of the system under test. .NET provides two options to write unit tests: MS Test and NUnit. These notes discuss how to use NUnit to do unit testing in .NET.

NUNIT

NUnit is an open source, unit-testing framework for all .Net languages. It initially was ported from JUnit, but has been completely rewritten for the .NET platforms. NUnit provides the following capabilities.

- Tests can be run within Visual Studio, from a console runner or through third party runners[1].
- Tests can be run in parallel.
- Strong support is provided for data driven tests.
- Every test case can be added to one or more categories to allow for selective running.

A class is tested by writing unit tests for the class' methods. A Calculator class, whose listing is shown in Figure 2, will be used to illustrate unit testing in Visual Studio using NUnit [2]. Before creating a test class and test methods, it is useful to understand the structure of a Visual Studio test project. Since the project name is "Calculator", Visual Studio names the test project "Calculator.Tests" as shown in Figure 1. The name of the class to test is `Calculator`, so Visual Studio names the test class `CalculatorTests`. Finally, Visual Studio appends "Test" to the names the test methods[2].

NUnit Attributes

Attributes, which add *metadata* to code, provide extra information about the code. While .NET mainly uses attributes to document code, NUnit uses them to provide information for testing. NUnit scans the code looking

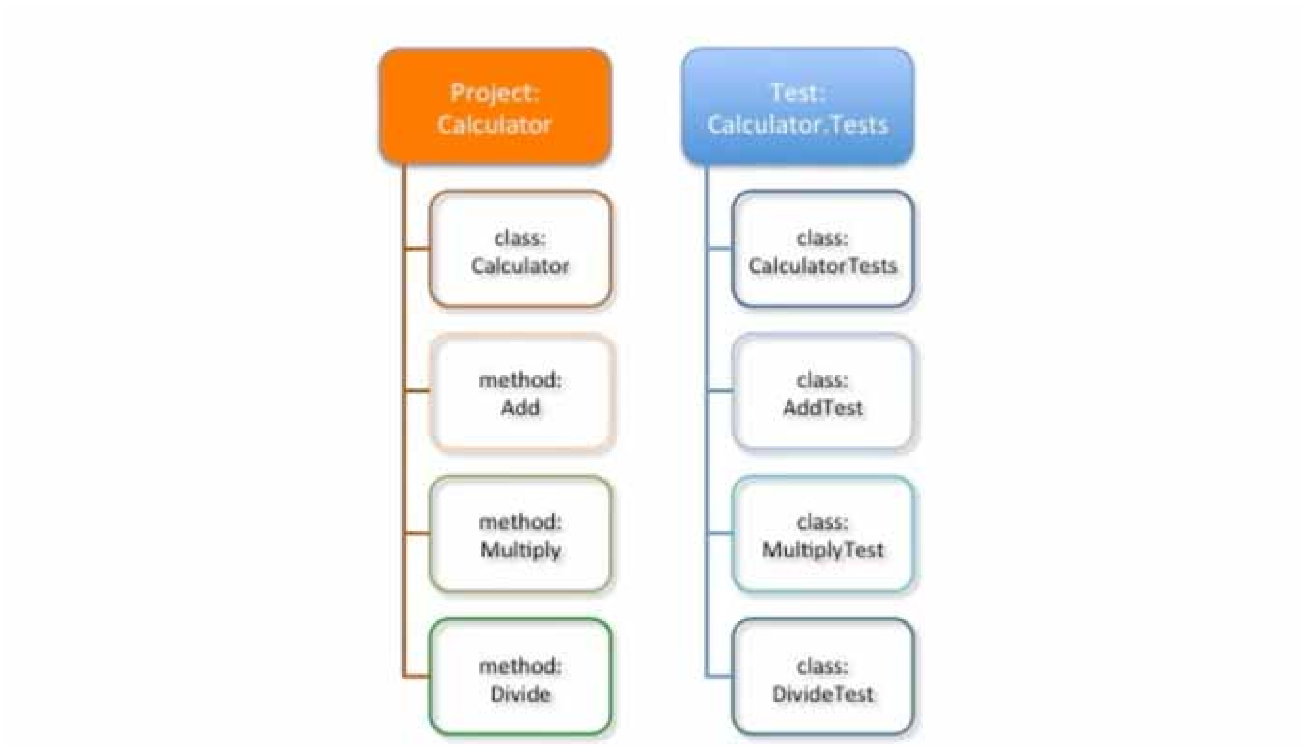


Figure 1: Relationship between Visual Studio project being developed and test project.

[1] A runner is a tool that executes (runs) unit tests and produces a report on the result of executing the unit tests.
[2] How a test class and its test methods are named can be customized in Visual Studio.

for attributes that tell it which classes and methods are tests. It then uses reflection to execute the test methods. This section describes the most commonly used NUnit attributes[3].

Category

The `Category` attribute provides a way to group tests as the code fragment on the right illustrates. Either individual tests or test fixtures may be identified as belonging to a particular category. When categories are used, only the tests in the selected categories will be run. Tests in categories that are not selected are not reported at all.

```
[TestFixture]
[Category("LongRunning")]
public class LongRunningTests
   { // Do something... }

[TestFixture]
public class SuccessTests
{
   [Test]
   [Category("Long")]
   public void VeryLongTest()
   { // Do something... }
}
```

Ignore

The `Ignore` attribute marks a test fixture or test that *should not be run* as the code fragment on the right illustrates. The required string argument, which will be displayed in the test result, should contain the reason that the test is being ignored. Ignored tests are displayed as warnings in order to provide a reminder that the test needs to be corrected or otherwise changed and re-instated.

```
[TestFixture]
public class SomeTests
{
   [Test]
   [Ignore("Skipping this test.")]
   public void TestOne()
   { // Do something... }
}
```

OneTimeSetUp and OneTimeTearDown

The `OneTimeSetUp` and `OneTimeTearDown` attributes mark methods inside a test fixture that are performed *once prior to executing any of the tests* in the test fixture or *once after all tests are completed* in the test fixture, respectively. A test fixture would normally have only one method marked with a `OneTimeSetUp` attribute or a `OneTimeTearDown` attribute.

One time setup and teardown might be used in the following cases.

- Opening a database connection before execution of the first test.
- Creating an instance of a particular object before execution of the first test.
- Deleting a particular file from the system before execution of any test.
- Closing all connections to a database after execution of all the tests.

```
[TestFixture]
public class SuccessTests
   {
   [OneTimeSetUp]
   public void Initialize()
   { // Do something... }

   [OneTimeTearDown]
   public void Cleanup()
   { // Do something... }

   [Test]
   public void Add()
   { // Do something... }
}
```

TestFixture

The `TestFixture` attribute marks a class that contains test methods as shown in the following code fragment. The test class has to be public and have a default constructor.

```
[TestFixture]
public class SomeTests
{ // Do something... }
```

[3] See https://github.com/nunit/docs/wiki/Attributes for a full list of NUnit attributes.

Setup and Teardown

When constructing unit tests a number of things often have to be done before or after each test. While calling a private method from each and every test method could do this, a better approach is to use the `Setup` and `Teardown` attributes as the code fragment on the right illustrates.

The `SetUp` attribute marks a method that contains initialization code that should be executed *before every test method* in the test fixture. This avoids the problem of repeating this code in each test method. A test fixture would normally have only one method marked with the `SetUp` attribute. A common use of this attribute is to create dependent objects (e.g., database connections).

The `Teardown` attribute marks a method that should be executed *after every test method* in the test fixture. A test fixture would normally have only one method marked with the `TearDown` attribute. A common use of this attribute is to close connections to a file system or database.

```
[TestFixture]
public class SomeTests
{
   private int _someValue;

   [SetUp]
   public void Setup()
   {
      _someValue = 5;
   }

   [TearDown]
   public void TearDown()
   {
      _someValue = 0;
   }

   [Test]
   public void TestOne()
   { // Do something... }
}
```

Test

The `Test` attribute marks a method that contains test code as the code fragment on the right illustrates. The `Test` attribute is normally used for simple (non-parameterized) tests, but may also be applied to parameterized tests. The test method may be either an instance or a static method. If the test method returns a value, the expected result must be passed in as a named parameter to the `Test` attribute as the second test on the right illustrates. This expected return value is checked for equality with the test method's return value.

```
[TestFixture]
public class SomeTests
{
   [Test]
   public void TestOne()
   { // Do something... }

   [Test(ExpectedResult = 4)]
   public int AddTest()
   { return 2 + 2; }
}
```

TestCase

A `TestCase` attribute serves the dual purpose of marking a method as a test method with parameters *and* providing inline data to be used when invoking that method as the code fragment below on the left illustrates.

By using the named parameter `ExpectdResult`, the preceding test set can be written as shown by the code fragment below on the right.

```
[TestFixture]
public class SomeTests
{
   [TestCase(12,3,4)]
   [TestCase(12,4,3)]
   public void DivideTest(int n,
      int d, int q)
   {
      Assert.That( q, Is.EqualTo,
         n / d );
   }
}
```

```
[TestFixture]
public class SomeTests
{
   [TestCase(12,3, ExpectedResult=4)]
   [TestCase(12,4, ExpectedResult=3)]
   public int DivideTest(int n, int d)
   {
      return( n / d );
   }
}
```

TestCaseSource

The `TestCaseSource` attribute is used on a parameterized test method to identify the source from which the required arguments will be provided as the code fragment on the right illustrates. The data is kept separate from the test itself and may be used by multiple test methods. The single attribute string argument represents the name of the source used to provide test cases and has the following characteristics:

- It may be a field, property or method in the test class.
- It *must* be static.
- It must return an IEnumerable or a type that implements IEnumerable. For fields an array is generally used. For properties and methods, an array may be returned or an iterator implemented.
- The individual items returned by the enumerator must be compatible with the signature of the method on which the attribute appears.

```
[TestFixture]
public class SomeTests
{
   [TestCaseSource(“DivideCases”)]
   public void DivideTest(int n, int
      d, int q)
   {
      Assert.That( q, Is.EqualTo, n /
         d );
   }

   static object[] DivideCases = {
      new object[] {12, 3, 4 },
      new object[] {12, 2, 6 },
      new object[] {12, 4, 3 },
   };
}
```

The test cases can be obtained from another class by using the following form of the `TestCaseSource` attribute where `AnotherClass` is the name of the class containing the test cases.

```
[TestCaseSource(typeof(AnotherClass), "DivideCases")]
```

Assertions

An assertion is used to test that the result of a test is what was expected. NUnit uses a *constraint-based* model with a single method of the `Assert` class for all assertions, passing a `Constraint` object that specifies the test to be performed. Using this model, all assertions are constructed using one of the forms of the `Assert.That()` method as the code fragment on the right illustrates.

If an assertion fails, the method call does not return and an error is reported. If a test contains multiple assertions, any that follow the one that failed will not be executed. For this reason, it's usually best to use only one assertion per test. Appendix A provides a list of the kinds of assertions supported in NUnit.

```
[TestFixture]
public class SomeTests
{
   [Test]
   public void TestOne()
   {
      int i = 4;
      Assert.That( 4, Is.EqualTo,
         i );
   }
}
```

Mock (Driver) Object

A mock (driver) object is a simulation of a real object. A mock object acts just like a real object, but in a controlled way. In unit testing, a mock object is used to test the behaviour of a real object.

Stub Object

A stub object is an object that implements an interface of a component. A stub object can be configured to return a value as required, but it is usually of a static nature meaning that no matter what input is passed in the same response is produced.

USING NUNIT IN VISUAL STUDIO[4]

To demonstrate how to do unit testing in Visual Studio using NUnit, we will construct unit tests for a `Calculator` class whose interface and listing is shown in Figure 2.

1. Create a new web application in Visual Studio named “Calculator” with unit tests added to the project.
2. Add a new folder named “Code_File” in the `Solution Explorer`.
3. Add a new C# Interface to the Code_File folder named `ICalculator.cs`.
4. Add the code shown in Figure 2 to the `ICalculator.cs` interface.
5. Add a new C# class to the “Code_File” folder named `Calculator.cs`.
6. Add the code shown in Figure 2 for the “Add”, “`Divide`” and “`Multiply`” methods to the `Calculator` class. Note that the `Multiply` method is intentionally implemented incorrectly so that its unit test will fail.

Creating Unit Tests

A unit test can be created manually in a test project or Visual Studio can be used to assist in the process. Right-clicking inside a class, but outside any method, and selecting “Create Unit Tests” from the popup menu, will create one new unit test for each method in the class. Right-clicking inside a method in a class and selecting “Create Unit Tests” from the popup menu, will create a new unit test for the method.

We will create one new unit test for each method in the `Calculator` class.

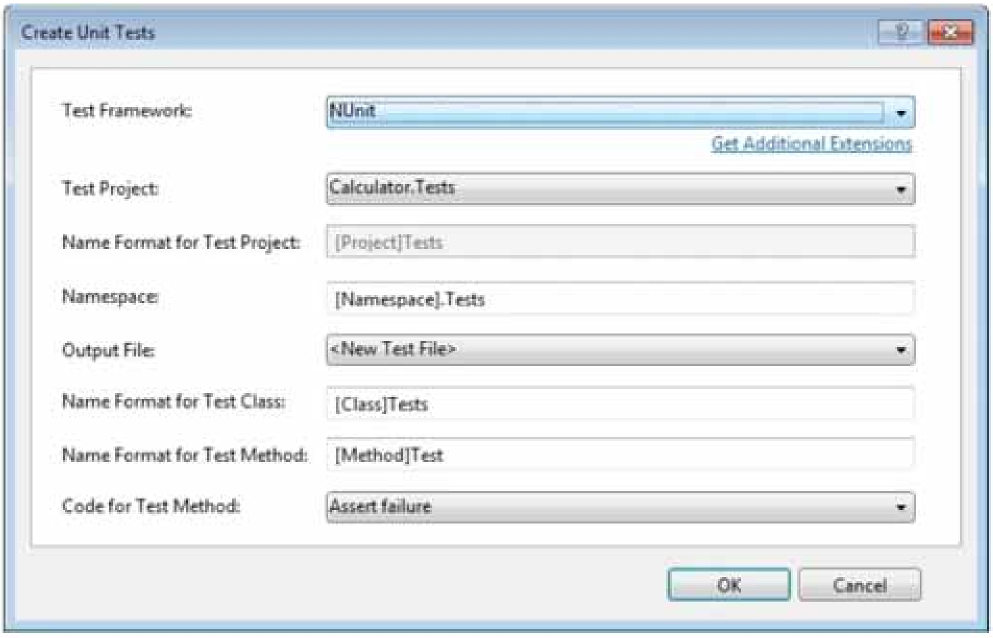
1. In the `Code Editor` window, right-click inside the `Calculator` class, but outside a method, and select “Create Unit Tests” from the popup menu.
2. In the `Create Unit Tests` window shown in Figure 3, do the following.
   a) Select “NUnit” in the *Test Framework* dropdown list.
   b) Select “Calculator.Tests” in the *Test Project* dropdown list to add the unit tests to the unit test project of the `Calculator` class.
   c) Leave the other fields as shown in Figure 3 and click the “OK” button[5].

   Visual Studio creates the test class and templates for the test methods as shown in Figure 4.

```
public interface ICalculator
{
   int Add(int num1, int num2);
   int Divide(int num1, int num2);
   int Multiply(int num1, int num2);
}

public class Calculator
{
   public int Add(int num1, int num2)
   {
      int result = num1 + num2;
      return result;
   }

   public int Divide(int num1, int num2)
   {
      int result = num1 / num2;
      return result;
   }

   public int Mulltipy(int num1, int num2)
   {
      int result = num1 + num2;
      return result;
   }
}
```

Figure 2: Code for `Calculator` interface and class.

[4] The NUnit packages described in Appendix B need to be installed to use NUnit in Visual Studio.
[5] If desired you can edit the *Name Format for Test Project*, *Name Format for Test Class* and *Name Format for Test Method* fields to customize the names of your test project, test class and test methods, respectively.



Create Unit Tests

Test Framework: NUnit

Get Additional Extensions

Test Project: Calculator.Tests

Name Format for Test Project: [Project]Tests

Namespace: [Namespace].Tests

Output File: <New Test File>

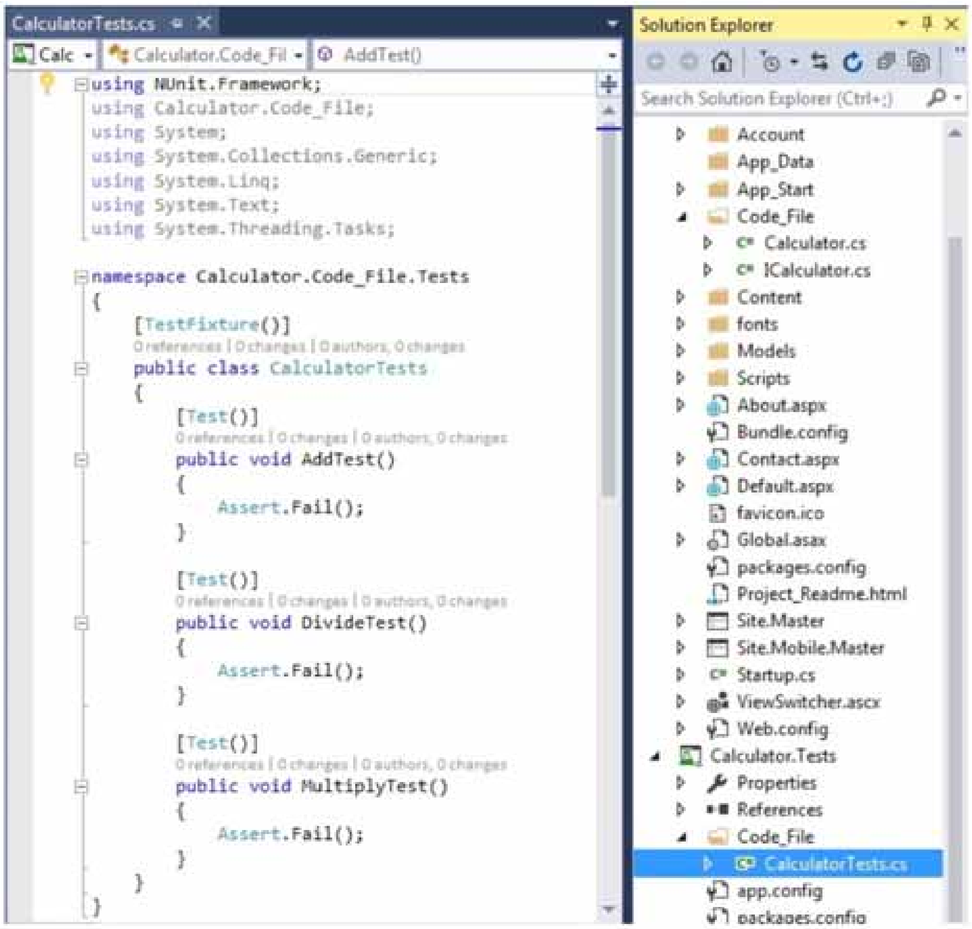
Name Format for Test Class: [Class]Tests

Name Format for Test Method: [Method]Test

Code for Test Method: Assert failure

OK Cancel

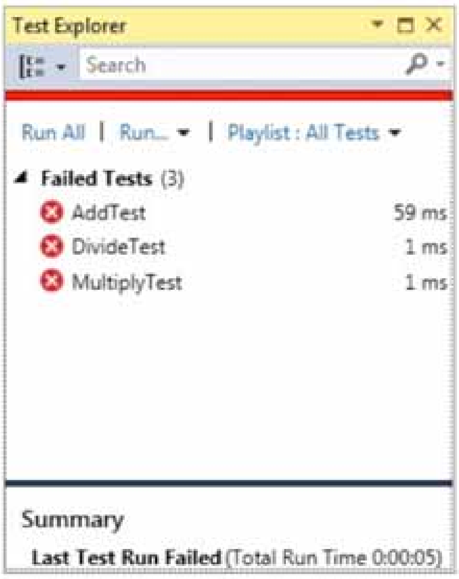
Figure 3: `Create Unit Tests` window.



```
using NUnit.Framework;
using Calculator.Code_File;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Calculator.Code_File.Tests
{
    [TestFixture()]
    public class CalculatorTests
    {
        [Test()]
        public void AddTest()
        {
            Assert.Fail();
        }

        [Test()]
        public void DivideTest()
        {
            Assert.Fail();
        }

        [Test()]
        public void MultiplyTest()
        {
            Assert.Fail();
        }
    }
}
```

Figure 4: Unit tests added for Calculator class.

3. Build the solution.
4. Select "Test→Windows→Test Explorer" in the Visual Studio toolbar to open the `Test Explorer` window shown in Figure 5(a).
5. Click "Run All" in the `Test Explorer` window.



Test Explorer

Run All | Run... | Playlist : All Tests

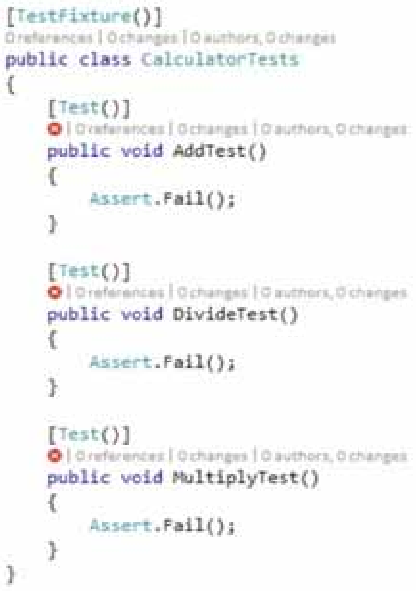
Failed Tests (3)

AddTest	59 ms
DivideTest	1 ms
MultiplyTest	1 ms

Summary

Last Test Run Failed (Total Run Time 0:00:05)

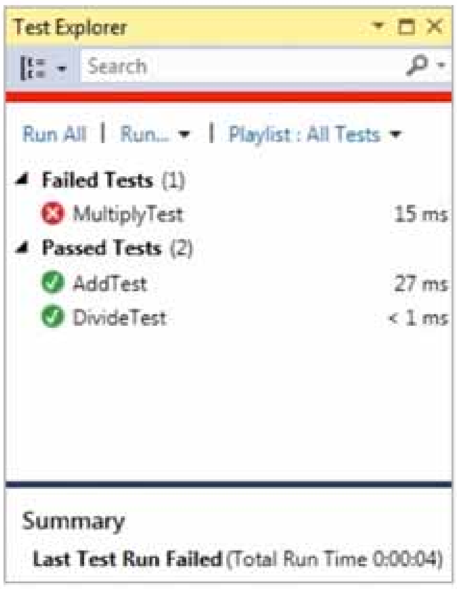
(a) `Test Explorer` window.



```
[TestFixture()]
0 references | 0 changes | 0 authors, 0 changes
public class CalculatorTests
{
    [Test()]
    0 references | 0 changes | 0 authors, 0 changes
    public void AddTest()
    {
        Assert.Fail();
    }

    [Test()]
    0 references | 0 changes | 0 authors, 0 changes
    public void DivideTest()
    {
        Assert.Fail();
    }

    [Test()]
    0 references | 0 changes | 0 authors, 0 changes
    public void MultiplyTest()
    {
        Assert.Fail();
    }
}
```

(b) Test result shown in `Code Editor` window.

Figure 5: Result of running unit tests for the `Calculator` class without test code.



Test Explorer

Run All | Run... | Playlist : All Tests

Failed Tests (1)

MultiplyTest	15 ms

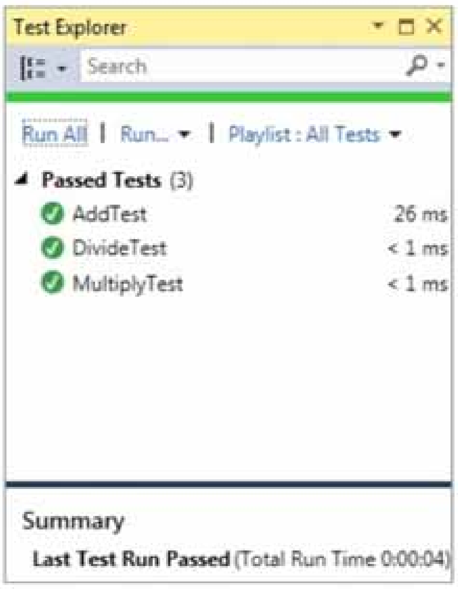
Passed Tests (2)

AddTest	27 ms
DivideTest	< 1 ms

Summary

Last Test Run Failed (Total Run Time 0:00:04)

(a) Before correcting `Multiply` method.



Test Explorer

Run All | Run... | Playlist : All Tests

Passed Tests (3)

AddTest	26 ms
DivideTest	< 1 ms
MultiplyTest	< 1 ms

Summary

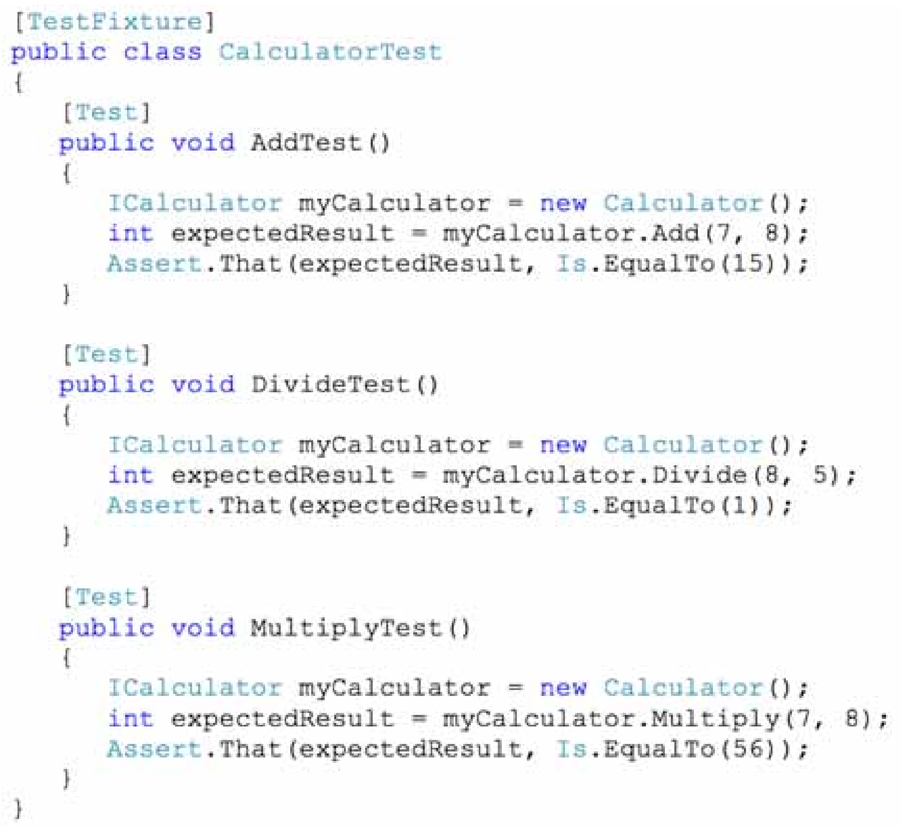
Last Test Run Passed (Total Run Time 0:00:04)

(b) After correcting `Multiply` method.

Figure 6: Results of running unit tests for the `Calculator` class after adding test code.

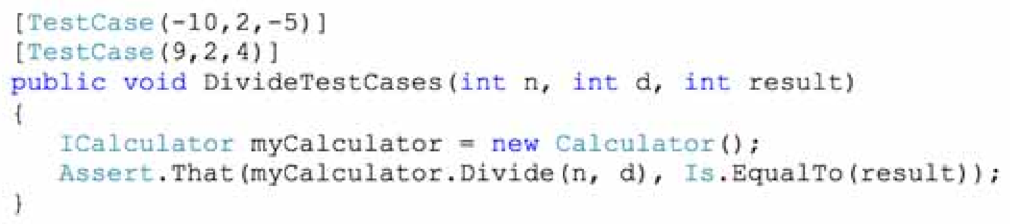
As shown in Figure 5(a), all the unit tests fail, as expected, since we have not added any code to the test methods. Visual Studio also shows the test results in the `Code Editor` window for the `CalculatorTests` class as shown in Figure 5(b).

Add the code shown in Figure 7 to each test method and rerun all tests in the `Test Explorer`. The test results are shown in Figure 6(a). The `MultiplyTest` method still fails since we have not corrected the error in its code. Correct the code of the `Multiply` method of the `Calculator` class and run the test again. All tests should pass as shown in Figure 6(b).



```
[TestFixture]
public class CalculatorTest
{
   [Test]
   public void AddTest()
   {
      ICalculator myCalculator = new Calculator();
      int expectedResult = myCalculator.Add(7, 8);
      Assert.That(expectedResult, Is.EqualTo(15));
   }

   [Test]
   public void DivideTest()
   {
      ICalculator myCalculator = new Calculator();
      int expectedResult = myCalculator.Divide(8, 5);
      Assert.That(expectedResult, Is.EqualTo(1));
   }

   [Test]
   public void MultiplyTest()
   {
      ICalculator myCalculator = new Calculator();
      int expectedResult = myCalculator.Multiply(7, 8);
      Assert.That(expectedResult, Is.EqualTo(56));
   }
}
```

Figure 7: Test class and test methods for `Calculator` class.



```
[TestCase(-10,2,-5)]
[TestCase(9,2,4)]
public void DivideTestCases(int n, int d, int result)
{
   ICalculator myCalculator = new Calculator();
   Assert.That(myCalculator.Divide(n, d), Is.EqualTo(result));
}
```

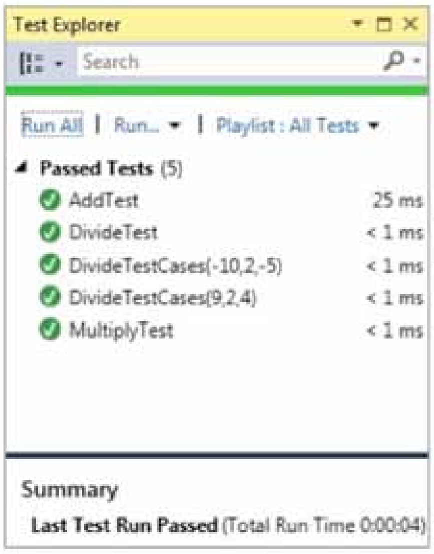
Figure 8: Test method with multiple test cases.

Running Multiple Cases For A Test

We will demonstrate how to provide multiple different inputs to a test method.

1. In the `Code Editor` window for the `Calculator` class, right-click inside the `Divide` method and select "Create Unit Tests" from the popup menu.
2. Modify the unit test to that shown in Figure 8.
3. Build the solution.
4. Click "Run All" in the `Test Explorer` window.

The result should be as shown in Figure 9.



```
Passed Tests (5)
  AddTest                       25 ms
  DivideTest                    < 1 ms
  DivideTestCases(-10,2,-5)     < 1 ms
  DivideTestCases(9,2,4)        < 1 ms
  MultiplyTest                  < 1 ms

Summary
  Last Test Run Passed (Total Run Time 0:00:04)
```

Figure 9: Results of running unit tests after adding test code.

```
[TestFixture]
public class CalculatorTest
{
   ICalculator myCalculator;

   [OneTimeSetUp]
   public void TestSetUp()
   {
      myCalculator = new Calculator();
   }

   [Test]
   public void AddTest()
   {
      int expectedResult = myCalculator.Add(7, 8);
      Assert.That(expectedResult, Is.EqualTo(15));
   }

   [Test]
   public void DivideTest()
   {
      int expectedResult = myCalculator.Divide(8, 5);
      Assert.That(expectedResult, Is.EqualTo(1));
   }

   [Test]
   public void MultiplyTest()
   {
      int expectedResult = myCalculator.Multiply(7, 8);
      Assert.That(expectedResult, Is.EqualTo(56));
   }

   [TestCase(-10,2,-5)]
   [TestCase(9,2,4)]
   public void DivideTestCases(int n, int d, int result)
   {
      Assert.That(myCalculator.Divide(n, d), Is.EqualTo(result));
   }

   [OneTimeTearDown]
   public void TestTearDown()
   {
      myCalculator = null;
   }
}
```

Figure 10: Test class with `OneTimeSetUp` and `OneTimeTearDown` methods.

Doing OneTimeSetUp and OneTimeTearDown

In the `CalculatorTests` class, objects of the `Calculator` class are created in all tests. This is not good testing practice. All the objects required for the tests should be created before executing any test. Any objects created should also be disposed of after running all the tests. Figure 10 shows the revised `CalculatorTests` class with onetime setup and teardown methods.

References

1. Andrew Hunt, David Thomas: *Pragmatic Unit Testing in C# with NUnit, 2nd Ed.* The Pragmatic Bookshelf, Raleigh 2007.
2. Kumar, Dhananjay, "Getting started with .NET unit testing using NUnit", Jul 27, 2015. Available at http://www.infragistics.com/community/blogs/dhananjay_kumar/archive/2015/07/27/getting-started-with-net-unit-testing-using-nunit.aspx

3. Lamsal, Neerjan, “Unit Testing Using NUnit”, Jul. 25, 2013. Available at http://www.codeproject.com/Articles/178635/Unit-Testing-Using-NUnit
4. Provost, Peter, “Test-Driven Development in .NET”, Mar. 17, 2003. Available at http://www.codeproject.com/Articles/3781/Test-Driven-Development-in-NET
5. Wikipedia, *NUnit*, https://en.wikipedia.org/wiki/NUnit.

APPENDIX A: NUNIT CONSTRAINT[6]

Constraint Model

Constraint	Syntax Helper	Operation
Equal	Is.EqualTo(object)	test that two objects are equal
Same as	Is.SameAs(object)	test that two object references refer to the same object
Condition	Is.Null	test for null
	Is.True	tests for true
	Is.False	tests for false
	Is.Nan	tests for floating NAN
	Is.Empty	tests for empty string or collection
	Is.Unique	tests that collection contains unique items
Comparison	Is.GreaterThan(IComparable)	tests that one value is greater than another
	Is.GreaterThanOrEqualTo(IComparable)	tests that one value is greater than or equal to another
	Is.AtLeast(IComparable)	tests that one value is greater than or equal to another
	Is.LessThan(IComparable)	tests that one value is less than another
	Is.LessThanOrEqualTo(IComparable)	tests that one value is less than or equal to another
	Is.AtMost(IComparable)	tests that one value is less than or equal to another
Type	Is.TypeOf(Type)	tests that an object is an exact Type
	Is.InstanceOfType(Type)	tests that an object is an instance of a Type
	Is.AssignableFrom(Type)	tests that one type is assignable from another

String	Text.Contains(string)
	Text.DoesNotContain(string)
		tests for a substring

	Text.StartsWith(string)
	Text.DoesNotStartWith(string)
		tests for an initial string

	Text.EndsWith(string)
	Text.DoesNotEndWith(string)
		tests for an ending string

	Text.Matches(string)
	Text.DoesNotMatch(string)
		tests that a pattern is matched

Collection	Is.All... Has.All...	applies a constraint to each item in a collection, succeeding only if all of them succeed
	Has.Some...	applies a constraint to each item in a collection, succeeding if at least one of them succeeds
	Has.None...	applies a constraint to each item in a collection, succeeding only if all of them fail
	Is.Unique	tests that a collection contains only unique items
	Has.Member(object)	tests that a collection contains an object
	Is.EquivalentTo(IEnumerable)	tests that two collections are equivalent

[6] NUnit Constraint-Based Assert Model (http://www.nunit.org/index.php?p=constraintModel&r=2.4.8)

	Is.SubsetOf(IEnumerable)	tests that one collection is a subset of another

Property	PropertyExistsConstraint(string name)
	Has.Property(string)
	PropertyConstraint(string name)
	Has.Property(string)...	

Compound	Is.Not…	Negates or reverses the effect of a constraint
	Is.All...	Tests that all members of a collection match the constraint
	Constraint & Constraint	Tests that both of the constraints are met
	Constraint | Constraint	Tests that at least one of the constraints is met
Custom	Custom constraints can be implemented by creating a class that inherits from the **Constraint** abstract class.	

Examples Use of Constraints

Equal

```
Assert.That(2+2, Is.EqualTo(4))
Assert.That(2+2, Is.Not.EqualTo(5))
Assert.That( 5.0, Is.EqualTo( 5 )
Assert.That( 2.1 + 1.2, Is.EqualTo( 3.3 ).Within( .0005 )
Assert.That( double.PositiveInfinity,
   Is.EqualTo( double.PositiveInfinity ) )
Assert.That( double.NegativeInfinity,
   Is.EqualTo( double.NegativeInfinity ) )
Assert.That( double.NaN, Is.EqualTo( double.NaN ) );
```

Same as

```
Exception ex1 = new Exception();
Exception ex2 = ex1;
Assert.That( ex2, Is.SameAs( ex1 ) );

Exception ex3 = new Exception();
Assert.That( ex3, Is.Not.SameAs( ex1 ) );
```

Condition

```
Assert.That(anObject, Is.Null)
Assert.That(anObject, Is.Not.Null
Assert.That(condition, Is.True)
Assert.That(condition, Is.False)
Assert.That(aDouble, Is.Nan)
Assert.That(aDouble, Is.Not.Nan
Assert.That(aString, Is.Empty)
Assert.That(collection, Is.Empty)
Assert.That(collection, is.Unique)
```

Comparison

```
Assert.That(7, Is.GreaterThan(3))
Assert.That(7, Is.GreaterThanOrEqualTo(3))
Assert.That(7, Is.AtLeast(3))

Assert.That(3, Is.LessThan(7))
Assert.That(3, Is.LessThanOrEqualTo(3))
Assert.That(3, Is.AtMost(3))
```

Type

```
Assert.That(“Hello”, Is.TypeOf(typeof(string)))
Assert.That(“Hello”, Is.Not.TypeOf(typeof(int)))

Assert.That(“Hello”, Is.InstanceOfType(typeof(string)))
Assert.That(5, Is.Not.InstanceOfType(typeof(string)))

Assert.That(“Hello”, Is.AssignableFrom(typeof(string)))
Assert.That(5, Is.Not.AssignableFrom(typeof(string)))
```

String

```
string phrase = "Make your tests fail before passing!"

Assert.That( phrase, Text.Contains( "tests fail" ) );
Assert.That( phrase, Text.Contains( "make" ).IgnoreCase );

Assert.That( phrase, Text.StartsWith( "Make" ) );
Assert.That( phrase, Text.Not.StartsWith( "Break" ) );
Assert.That( phrase, Text.DoesNotStartWith( "Break" ) );

Assert.That( phrase, Text.EndsWith( "!" ) );
Assert.That( phrase, Text.EndsWith( "PASSING!" ).IgnoreCase );

Assert.That( phrase, Text.Matches( "Make.*tests.*pass" ) );
Assert.That( phrase, Text.Not.Matches( "your.*passing.*tests" ) );
Assert.That( phrase, Text.DoesNotMatch( "your.*passing.*tests" ) );
```

Collection

```
int[] iarray = new int[] { 1, 2, 3 };
string[] sarray = new string[] { "a", "b", "c" };

Assert.That( iarray, Is.All.Not.Null );
Assert.That( sarray, Is.All.InstanceOfType(typeof(string)) );
Assert.That( iarray, Is.All.GreaterThan(0) );

Assert.That( sarray, Is.Unique );

Assert.That( iarray, List.Contains(3) );
Assert.That( sarray, List.Contains("b") );
Assert.That( sarray, List.Not.Contains("x") );

Assert.That( new string[] { "c", "a", "b" },
   Is.EquivalentTo( sarray ) );
Assert.That( new int[] { 1, 2, 2 }, Is.Not.EquivalentTo( iarray ) );
Assert.That( new int[] { 1, 3 }, Is.SubsetOf( iarray ) );
```

Property

```
Assert.That( someObject, Has.Property( "Version" ) );
Assert.That(someObject, Has.Property("Version").EqualTo("2.0"));
Assert.That(collection, Has.Property("Count").GreaterThan(10));

Assert.That(someObject, Has.Property("Version").EqualTo("2.0"));
Assert.That(collection, Has.Property("Count").GreaterThan(10));

Assert.That(someObject, Has.Length...
Assert.That(someObject, Has.Count...
Assert.That(someObject, Has.Message...
Assert.That(someObject, Has.InnerException...
```

Compound

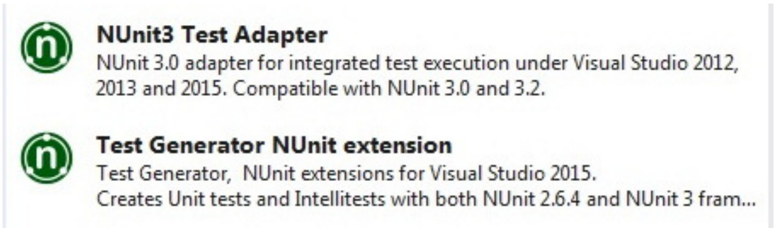
```
Assert.That( 2 + 2, Is.Not.EqualTo( 5 )
Assert.That( new int[] { 1, 2, 3 }, Is.All.GreaterThan( 0 ) )
Assert.That( 2.3, Is.GreaterThan( 2.0 ) & Is.LessThan( 3.0 ) )
Assert.That( 3, Is.LessThan( 5 ) | Is.GreaterThan( 10 ) );
```

APPENDIX B: INSTALLING NUNIT IN VISUAL STUDIO

To use NUnit in Visual Studio in the lab, the NUnit packages shown in Figure 11 should be installed in Visual Studio.

If a package is not installed, then use the NuGet Package Manager to install it.

After installing the NUnit packages, right-click the test project node in the `Solution Explorer` (e.g., “Calculator.Tests”) and add a reference to “NUnitFramework.NET4.5”.



NUnit3 Test Adapter
NUnit 3.0 adapter for integrated test execution under Visual Studio 2012, 2013 and 2015. Compatible with NUnit 3.0 and 3.2.

Test Generator NUnit extension
Test Generator, NUnit extensions for Visual Studio 2015.
Creates Unit tests and Intellitests with both NUnit 2.6.4 and NUnit 3 fram...

Figure 11: NUnit packages to install in Visual Studio.