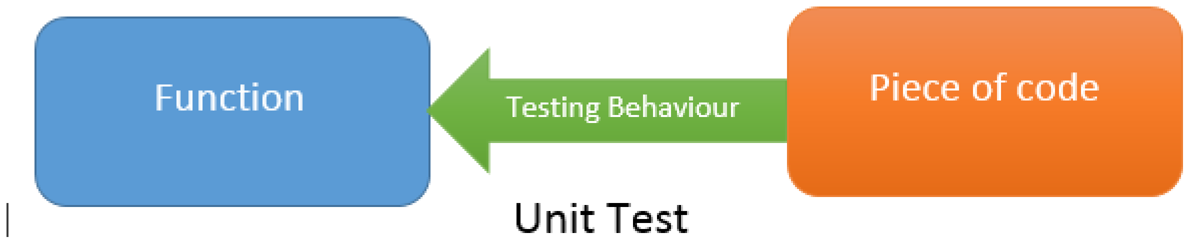
Automated unit tests while developing your code are considered good practice. Doing so helps you find bugs early in the development cycle and saves you time in the long run. The developer can run these tests multiple times to verify the behavior of a particular unit of code for different input sets.

Unit tests essentially check the behavior of a particular unit of the code or the function and are written by the developers who are implementing the functionalities.



We test the behavior of a function before it becomes part of the whole system and goes into production by writing a piece of code to test this behavior under different conditions. Usually, a function gets tested in isolation with the other functions of the system under test (SUT).

In .NET, there are two options to write Unit Tests:

1. Using MS Test
2. Using NUnit

In this article, we will learn to start Unit testing of C# classes using NUnit. We will walk through:

- Creating the system under test
- Setting up NUnit
- Creating the Test project
- Creating Test classes and methods
- Different options for running the tests
- Understanding TestFixtures and TestTearDown
- Working with Ignore test

System Under Test

In this example, we are going to write a Unit Test for a Calculator system. Let's say the interface of the Calculator is defined as shown in the listing below:

```
namespace Calculator
{
  public interface ICalculator
   {
     int Add(int num1, int num2);
     int Mul(int num1, int num2);
   }
}
```

The Calculator class is implemented as shown in the listing below:

```
namespace Calculator
```

```
{
 public class Calculator :ICalculator
  {

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

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

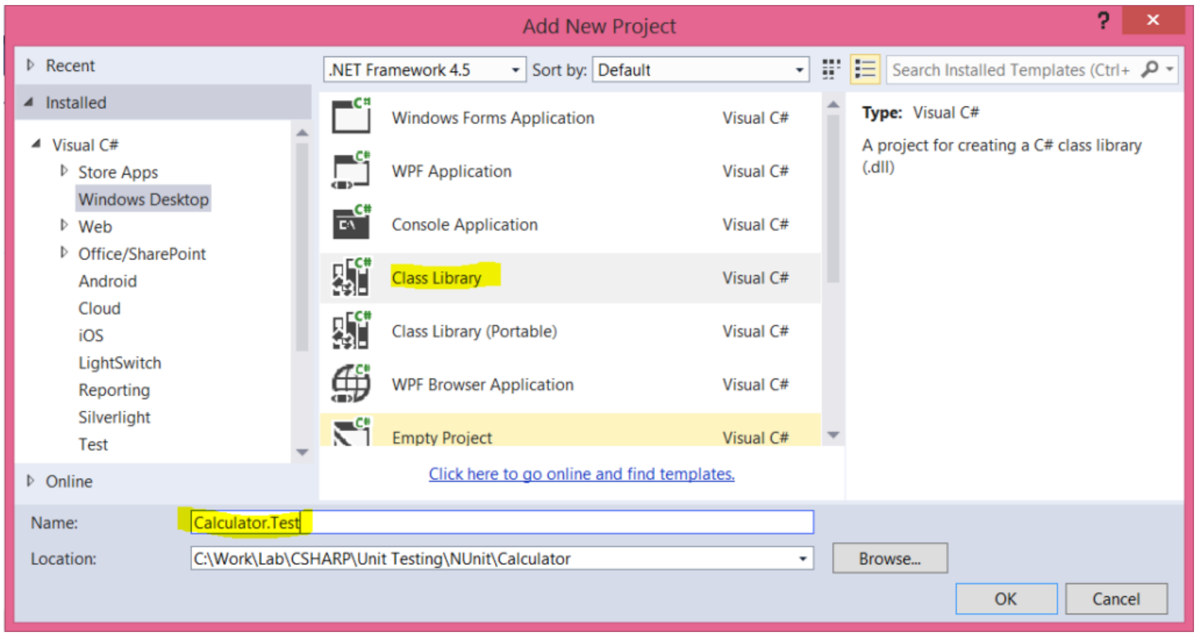
You'll notice that we have implemented Mul function incorrectly on purpose.

To start working with NUnit and writing the test, we need to follow the following steps:

1. Create a test project
2. Add a reference to NUnit library
3. Add a reference to System under test project
4. Create a test class and write the test method

Setting up the Test Project

To create a test project, add a class library project as shown in the listing below. Usually, we should follow the naming convention to name the test project as ProjectUnderTest.Test. Here we are going to test the Calculator project, so the name of the test project should be **Calculator.Test**.


Add New Project

Recent
Installed
Visual C#
Store Apps
Windows Desktop
Web
Office/SharePoint
Android
Cloud
iOS
LightSwitch
Reporting
Silverlight
Test
Online

.NET Framework 4.5 | Sort by: Default | Search Installed Templates (Ctrl+

Windows Forms Application	Visual C#
WPF Application	Visual C#
Console Application	Visual C#
Class Library	Visual C#
Class Library (Portable)	Visual C#
WPF Browser Application	Visual C#
Empty Project	Visual C#

Click here to go online and find templates.

Type: Visual C#

A project for creating a C# class library (.dll)

Name: Calculator.Test

Location: C:\Work\Lab\CSHARP\Unit Testing\NUnit\Calculator

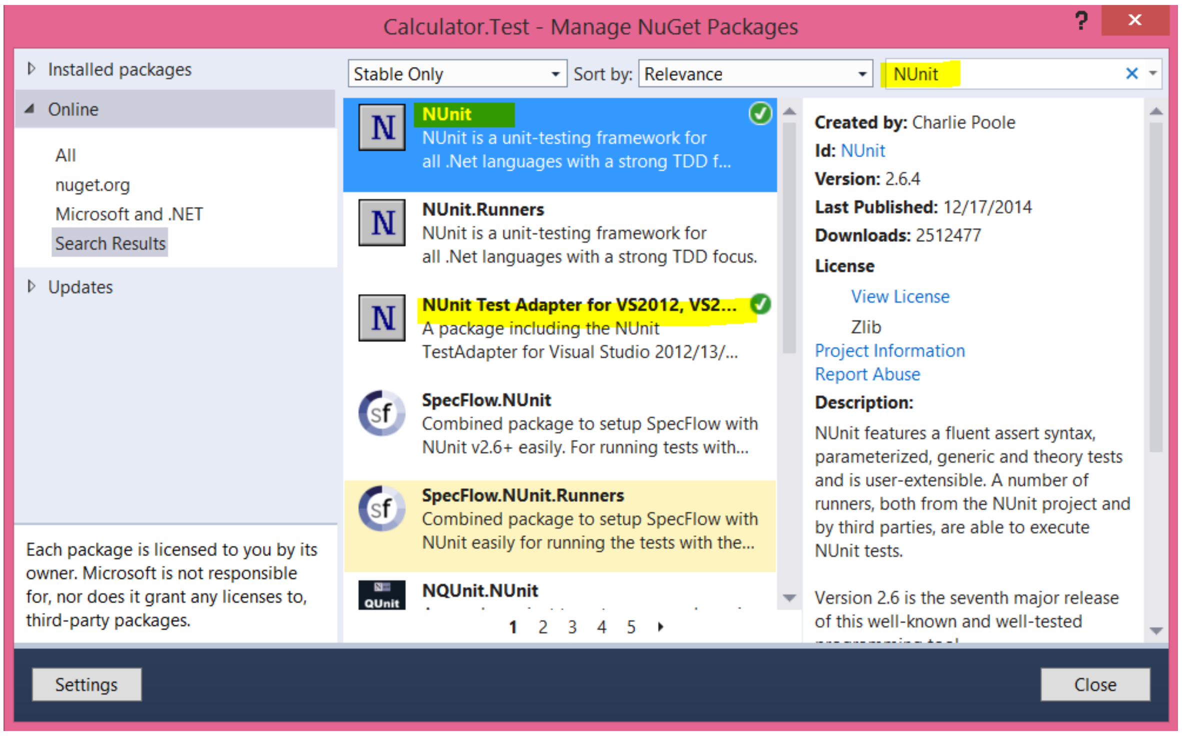
Browse... OK Cancel

Once a Test project is created you have two options to add a reference to NUnit in the project:

1. Using the extension and updates
2. Using the NuGet package

To work with extensions and updates, click on the Tool->Extension and Updates and then select NUnit Test Adapter.

To work with the NuGet package, right click on the project and from the context menu select the option of Manage NuGet packages. In the NuGet package manager, search for the NUnit and install NUnit Test Adapter for VS and NUnit in the project as shown in the image below:


Calculator.Test - Manage NuGet Packages

Installed packages
Online
All
nuget.org
Microsoft and .NET
Search Results
Updates

Stable Only | Sort by: Relevance | NUnit

NUnit
NUnit is a unit-testing framework for all .Net languages with a strong TDD f...

NUnit.Runners
NUnit is a unit-testing framework for all .Net languages with a strong TDD focus.

NUnit Test Adapter for VS2012, VS2...
A package including the NUnit TestAdapter for Visual Studio 2012/13/...

SpecFlow.NUnit
Combined package to setup SpecFlow with NUnit v2.6+ easily. For running tests with...

SpecFlow.NUnit.Runners
Combined package to setup SpecFlow with NUnit easily for running the tests with the...

NQUnit.NUnit

1 2 3 4 5

Created by: Charlie Poole
Id: NUnit
Version: 2.6.4
Last Published: 12/17/2014
Downloads: 2512477
License
View License
Zlib
Project Information
Report Abuse
Description:
NUnit features a fluent assert syntax, parameterized, generic and theory tests and is user-extensible. A number of runners, both from the NUnit project and by third parties, are able to execute NUnit tests.

Version 2.6 is the seventh major release of this well-known and well-tested

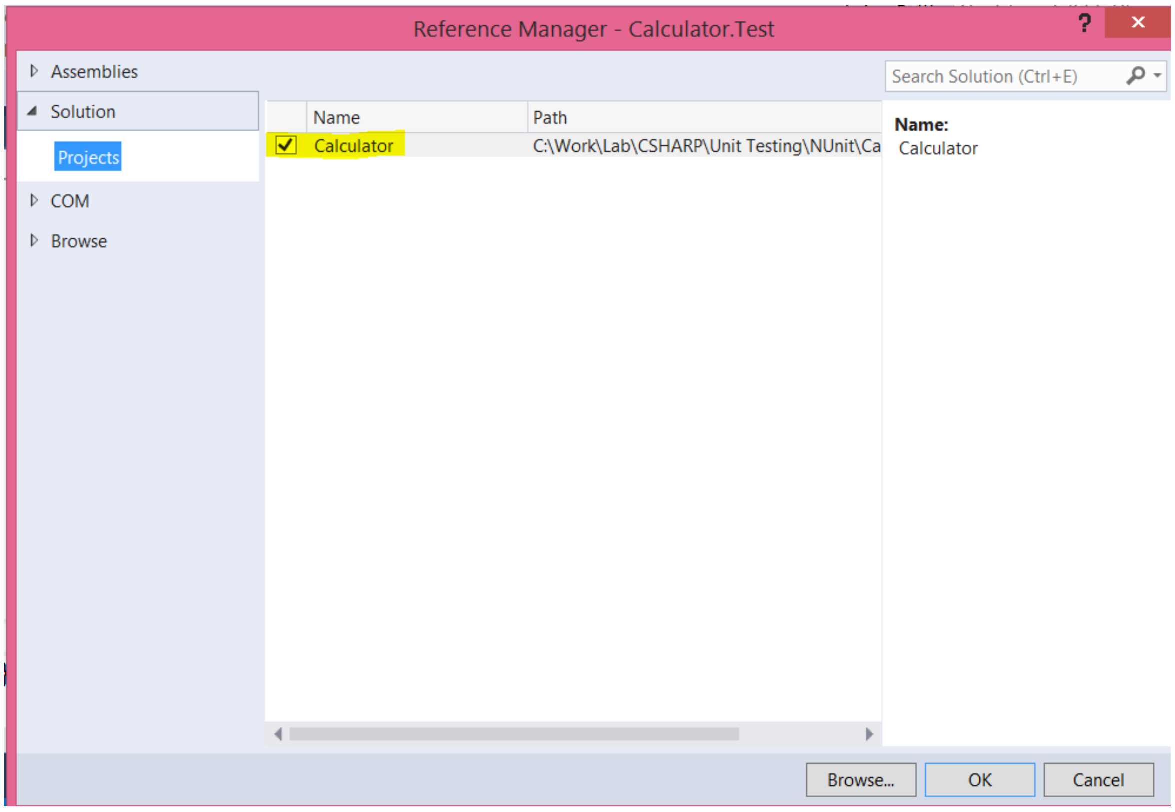
Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Settings Close

After installing the NUnit Test Adapter, we can work with NUnit.

Add a Reference to the System Under Test Project

Next, we need to add a reference to the Calculator project (system under test) in the test project. To add that right click on the test project and add a reference to the Calculator project.



Reference Manager - Calculator.Test

Search Solution (Ctrl+E)

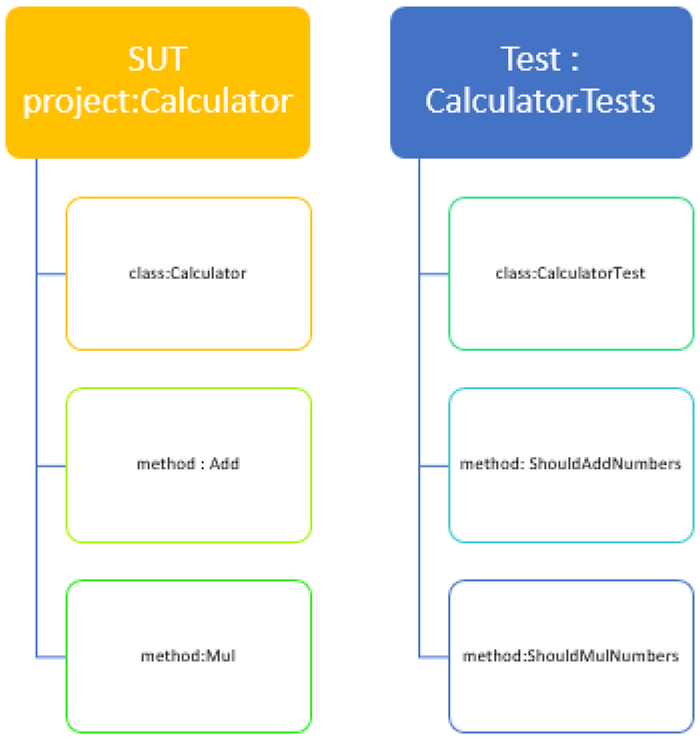
	Name	Path
✓	Calculator	C:\Work\Lab\CSHARP\Unit Testing\NUnit\Ca

Name: Calculator

Create Test class and Test methods

Before we go ahead and create a test class and test methods, let us understand the structure of the test project. Since our SUT project name is Calculator, we should name the test project as **Calculator.Tests**. The name of the class under the test is Calculator, so the name of the test should be CalculatorTest. Finally, the name of the test function should be the name of the function under the test, prepended by the word “should”. The idea here is to give a readable name to the test project, class and function.

The relationship between the SUT project and the test project can be depicted as shown in the image below:



Next, let us go ahead and create the Test Class and the Test Methods. We need to use:

1. TestFixture attribute to create the test class
2. Test attribute to create the test method

A test class can be created as shown in the listing below:

```
using NUnit.Framework;

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

     [Test]
     public void ShouldMulTwoNumbers()
     {
            ICalculator sut = new Calculator();
            int expectedResult = sut.Mul(7, 8);
            Assert.That(expectedResult, Is.EqualTo(56));
        }


    }
}
```

In the above listing, we are:

1. Creating an object of the SUT Calculator class
2. Calling the methods
3. Asserting the result using NUnit assert (we will talk more about asserting in later posts)

After building the test project, we need to run the test.

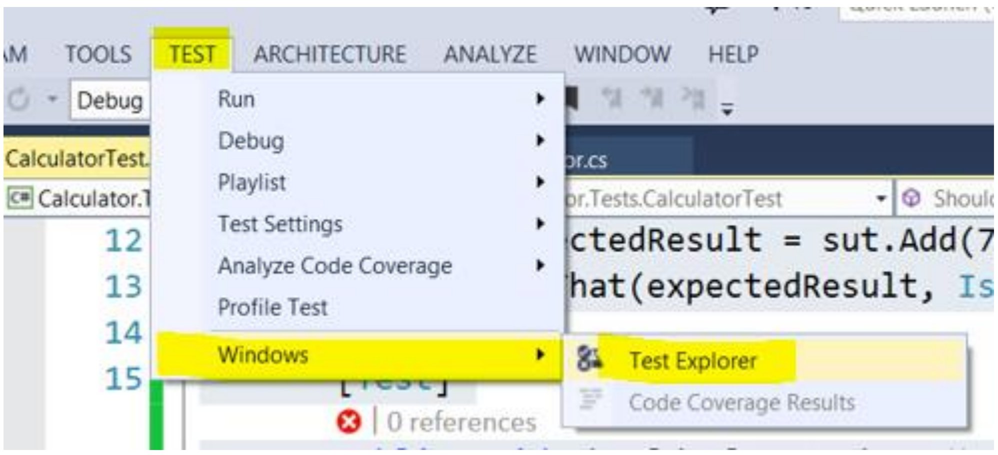
Running the Test

There are three options to run the test:

1. By using the Visual Studio Test Explorer
2. By using the NUnit GUI
3. By using the NUnit command prompt

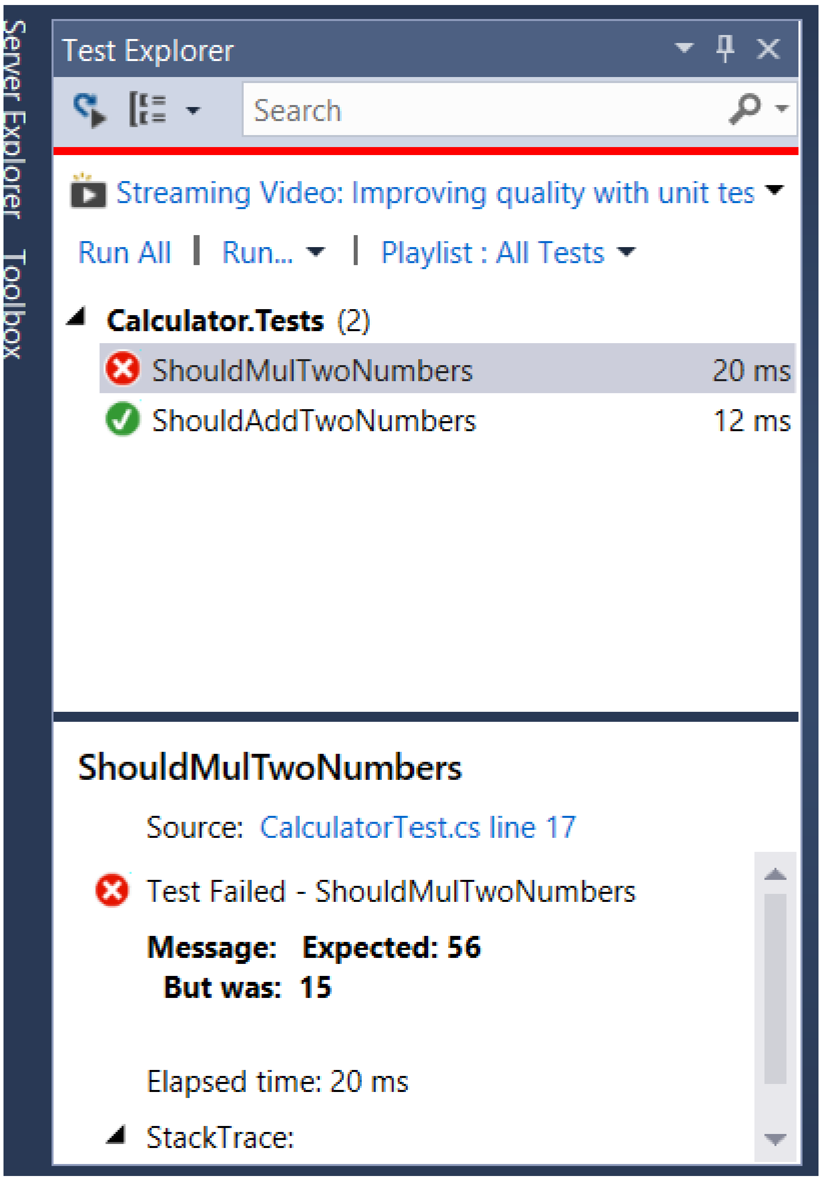
Visual Studio Test Explorer

To use Visual Studio Test Runner, launch the Test Explorer by clicking on the TEST-WINDOWS-TEST EXPLORER option as shown in the image below:



```
ctedResult = sut.Add(7
hat(expectedResult, Is
```

You should able to see the unit test listed in the Test Explorer. You can either select a particular test and run it or choose to run all the tests. After running the tests of Calculator.Tests project, in the Test Explorer, the result can be seen as shown in the image below. We can see that ShouldAddTwoNumbers test is passed whereas ShouldMulTwoNumbers test is failed (which is expected).



Calculator.Tests (2)

ShouldMulTwoNumbers 20 ms

ShouldAddTwoNumbers 12 ms

ShouldMulTwoNumbers

Source: CalculatorTest.cs line 17

Test Failed - ShouldMulTwoNumbers

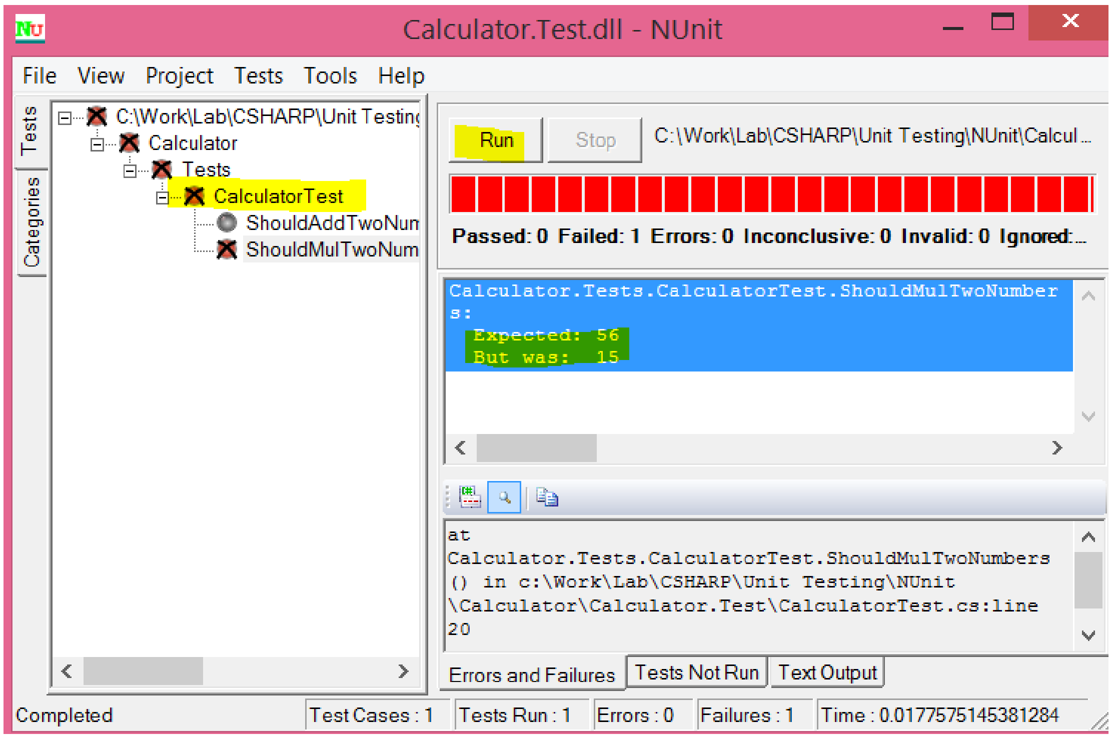
Message: Expected: 56
But was: 15

Elapsed time: 20 ms

StackTrace:

NUnit GUI

You can download the MSI of NUnit GUI here. Once downloaded, install the NUnit GUI, and in the File menu, select the option of Open Project, then add DLL to the Calculator.Test project. After adding the DLL, you should able to see the tests loaded in the UI. To run a test, select and click on Run. In the NUnit UI, you can clearly see the message for the failed test.


Calculator.Test.dll - NUnit
File View Project Tests Tools Help
Tests
Categories
C:\Work\Lab\CSHARP\Unit Testing
Calculator
Tests
CalculatorTest
ShouldAddTwoNum
ShouldMulTwoNum
Run
Stop
C:\Work\Lab\CSHARP\Unit Testing\NUnit\Calcul...
Errors and Failures
Tests Not Run
Text Output

```
Passed: 0  Failed: 1  Errors: 0  Inconclusive: 0  Invalid: 0  Ignored:...
```

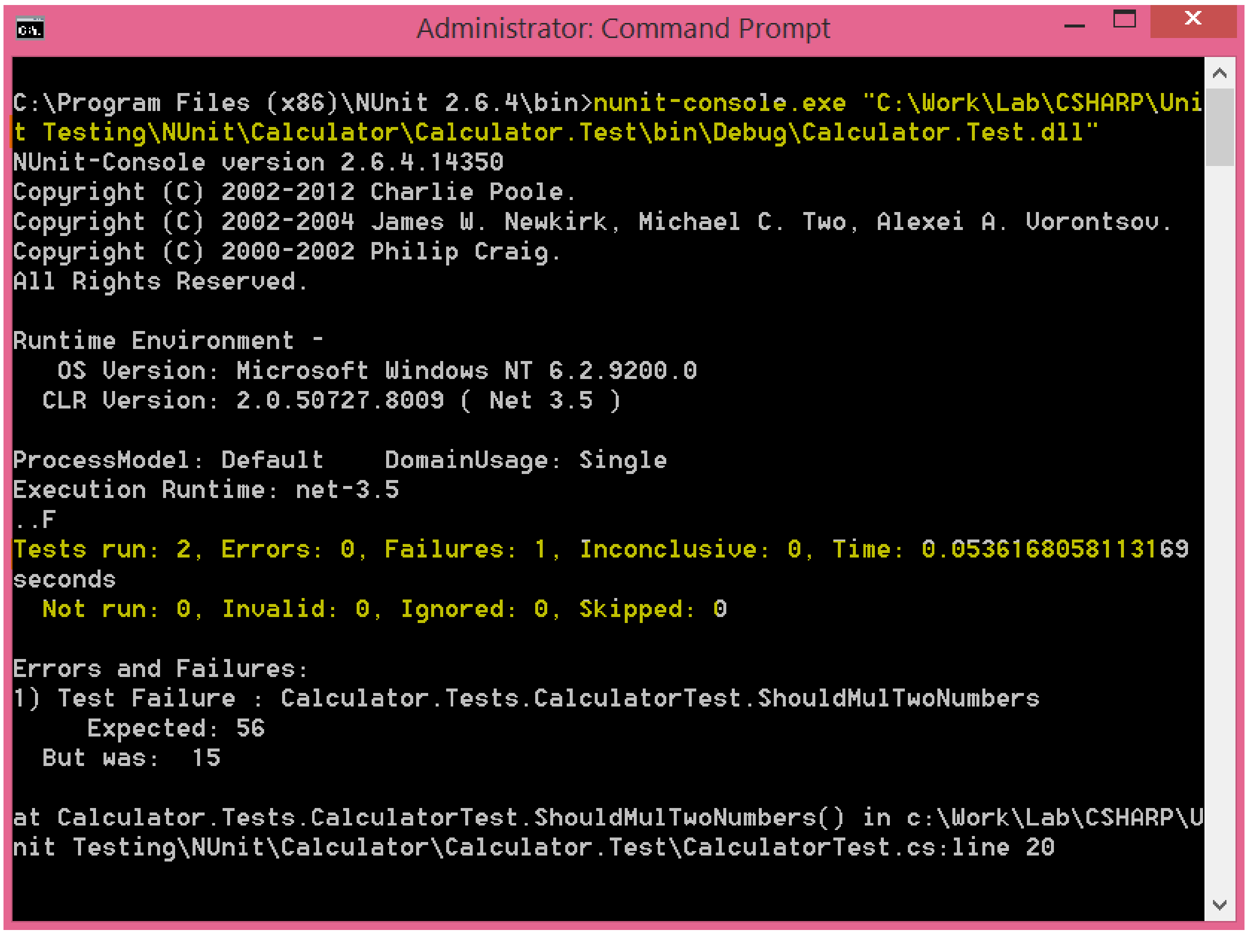
```
Calculator.Tests.CalculatorTest.ShouldMulTwoNumber
s:
  Expected: 56
  But was:  15
```

```
at
Calculator.Tests.CalculatorTest.ShouldMulTwoNumbers
() in c:\Work\Lab\CSHARP\Unit Testing\NUnit
\Calculator\Calculator.Test\CalculatorTest.cs:line
20
```

```
Completed    Test Cases : 1    Tests Run : 1    Errors : 0    Failures : 1    Time : 0.0177575145381284
```

NUnit Command Prompt

Your third option to run the test is by using NUnit command prompt. Navigate to NUnit\Bin folder under the Program Files (its location depends on the installation path you selected) and run the file nunit-console.exe. In the double quotes, you need to pass the full path of the test project dll as the second parameter to run the test as shown in the image below:


Administrator: Command Prompt

```
C:\Program Files (x86)\NUnit 2.6.4\bin>nunit-console.exe "C:\Work\Lab\CSHARP\Uni
t Testing\NUnit\Calculator\Calculator.Test\bin\Debug\Calculator.Test.dll"
NUnit-Console version 2.6.4.14350
Copyright (C) 2002-2012 Charlie Poole.
Copyright (C) 2002-2004 James W. Newkirk, Michael C. Two, Alexei A. Vorontsov.
Copyright (C) 2000-2002 Philip Craig.
All Rights Reserved.

Runtime Environment -
   OS Version: Microsoft Windows NT 6.2.9200.0
  CLR Version: 2.0.50727.8009 ( Net 3.5 )

ProcessModel: Default    DomainUsage: Single
Execution Runtime: net-3.5
..F
Tests run: 2, Errors: 0, Failures: 1, Inconclusive: 0, Time: 0.0536168058113169
seconds
  Not run: 0, Invalid: 0, Ignored: 0, Skipped: 0

Errors and Failures:
1) Test Failure : Calculator.Tests.CalculatorTest.ShouldMulTwoNumbers
     Expected: 56
  But was:  15

at Calculator.Tests.CalculatorTest.ShouldMulTwoNumbers() in c:\Work\Lab\CSHARP\U
nit Testing\NUnit\Calculator\Calculator.Test\CalculatorTest.cs:line 20
```

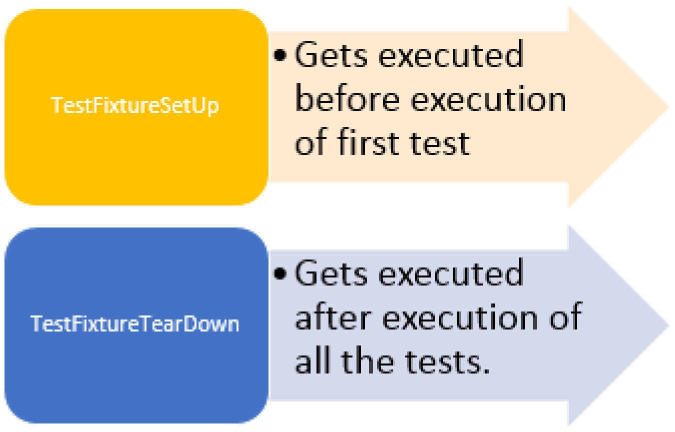
The NUnit command prompt option is very useful for the continuous integration.

TestFixtureSetUp and TestFixtureTearDown

Two very important concepts of NUnit testing are Test Setup and Teardown. In the Calculator.Test, we are creating objects of the SUT Calculator in both tests. This is not the right practice. We should create all the objects required for the test before execution of any test. Other scenarios could be:

1. Create a database connection before execution of the first test
2. Create an instance of a particular object before execution of the first test
3. Delete all connection to the database after execution of all the tests
4. Delete a particular file from the system before execution of any test

We can solve the above scenarios using the TestFixtureSetUp and the TestFixtureTearDown. TestFixtureSetUp is a code which gets executed before the execution of any test, and the TestFixtureTearDown is a piece of code which gets executed after execution of all the tests.


TestFixtureSetUp
Gets executed before execution of first test
TestFixtureTearDown
Gets executed after execution of all the tests.

Let us consider Calculator.Test. In both tests, we are creating an object of the SUT Calculator. Whereas we need an object of the Calculator class before execution of any test and we need to dispose of that after execution of all the tests. We can modify the test class as with SetUp and TearDown as shown in the listing below:

```
using NUnit.Framework;


namespace Calculator.Tests
{

  [TestFixture]
  public class CalculatorTest
   {
     ICalculator sut;
     [TestFixtureSetUp]
     public void TestSetup()
     {
       sut = new Calculator();
     }
     [Test]
     public void ShouldAddTwoNumbers()
     {

       int expectedResult = sut.Add(7, 8);
       Assert.That(expectedResult, Is.EqualTo(15));
     }
     [Test]
     public void ShouldMulTwoNumbers()
     {
            int expectedResult = sut.Mul(7, 8);
            Assert.That(expectedResult, Is.EqualTo(56));
        }

        [TestFixtureTearDown]
        public void TestTearDown()
        {
            sut = null;
        }

    }
}
```

In the above listing, you will notice:

1. There is a function attributed with [TestFixtureSetUp]. This function will be executed before the execution of any test. We are creating an object of the SUT Calculator inside this function such that the created instance will be available to all the tests.
2. There is a function attributed with [TestFixtureTearDown]. This function will be executed after the execution of all the tests. We are disposing the object of SUT Calculator inside this function.

Ignore Test

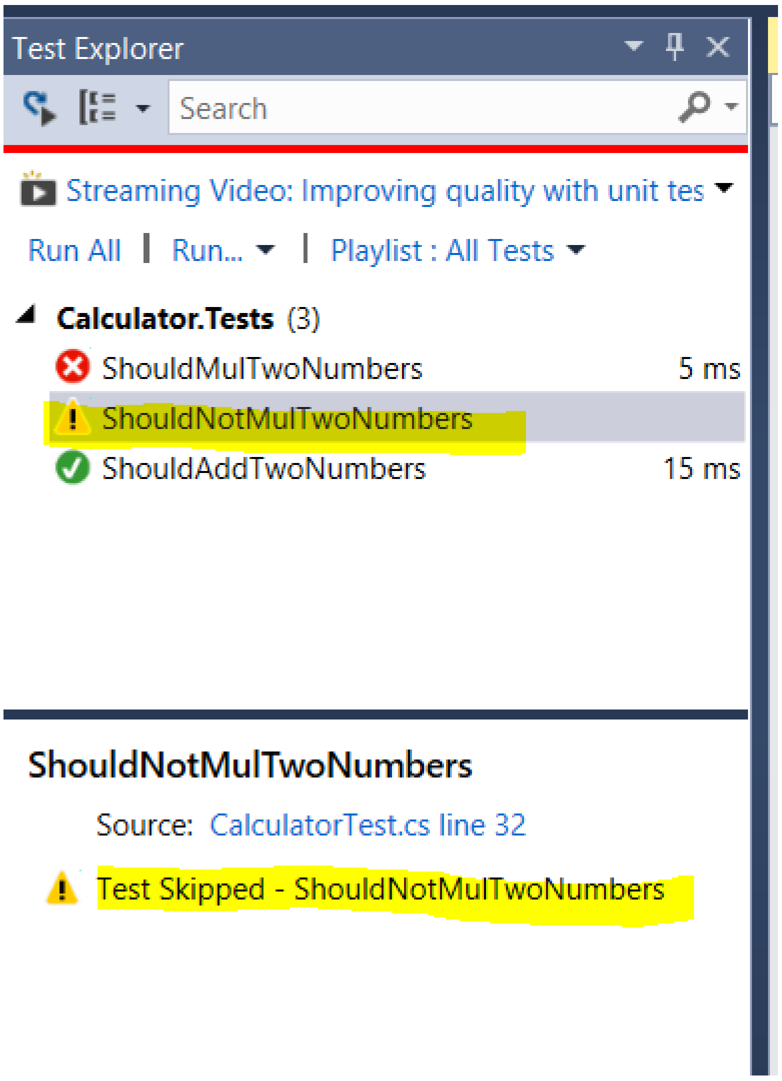
Another scenario you may encounter while running a test is that you may want to ignore certain tests to be executed. For example, there are three tests and you want one of the tests to be ignored while running the rest of your tests. This can be done by attributing the test to be ignored with the **[ignore]** attribute.

A test can be ignored as shown in the listing below:

```
 [Test]
        [Ignore]
        public void ShouldNotMulTwoNumbers()
        {

            int expectedResult = sut.Mul(7, 8);
            Assert.That(expectedResult, Is.EqualTo(15));
        }
```

When you run all the tests in the test explorer, for the ignored test you will get the message as shown in the image below:



Calculator.Tests (3)

✗ ShouldMulTwoNumbers 5 ms

⚠ ShouldNotMulTwoNumbers

✓ ShouldAddTwoNumbers 15 ms

ShouldNotMulTwoNumbers

Source: CalculatorTest.cs line 32

⚠ Test Skipped - ShouldNotMulTwoNumbers