Developer Tools for .NET

Lab Guide

Document Revision History

Date	Revision No.	Author	Summary of Changes
15-May-2016	1	Nachiket Inamdar	Added Microsoft Test Framework content.

Table of Contents

Lab 1: Using NUnit to Test Code

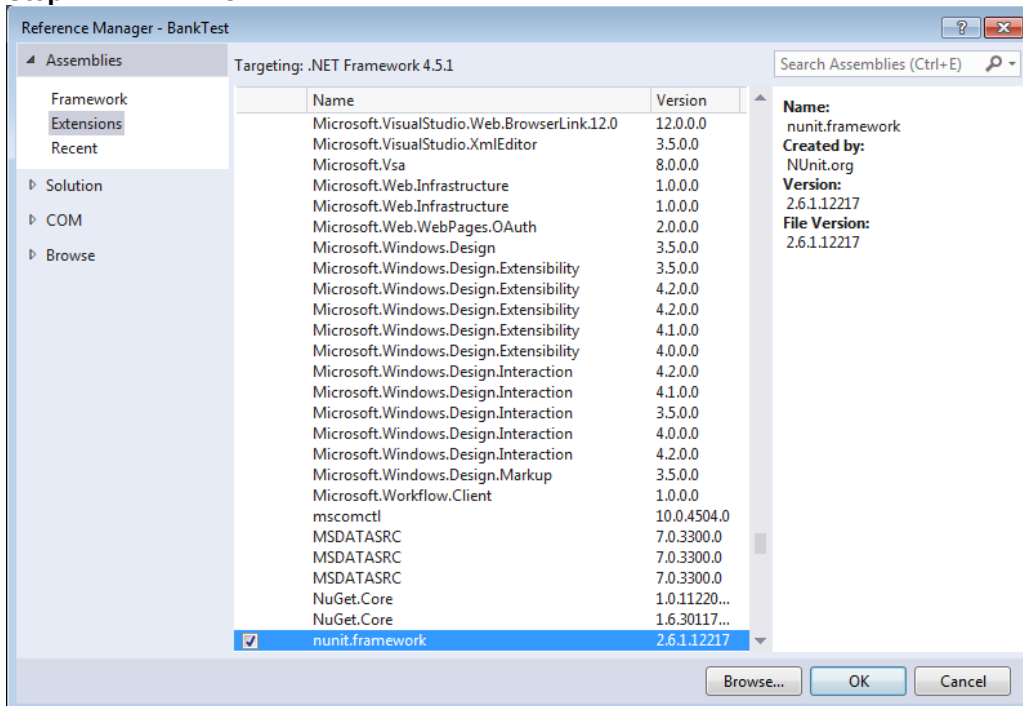| Goals	• Understand working of NUnit to Test the code • Understand the steps involved in writing correct code.
Time	90 Minutes
Lab Setup	VS.NET 2013 & NUnit 2.x Framework

Lab 1 A. Problem statement

Write the production code & check the same for correctness.

Solution:

Step 1: Create your project of Library Type.

Step 2: Add the NUnit reference

Step 3: Add your TestCase

Capgemini Internal

1. Add the reference in your code to the NUnit namespace:

<< to do>>

2. Add your testing class to your source file:

```
namespace TestDemo1
{
        [TestFixture]
        public class MyTestClass
        {
                [Test]
                public void TestFunction()
                {

                }
        }
}
```
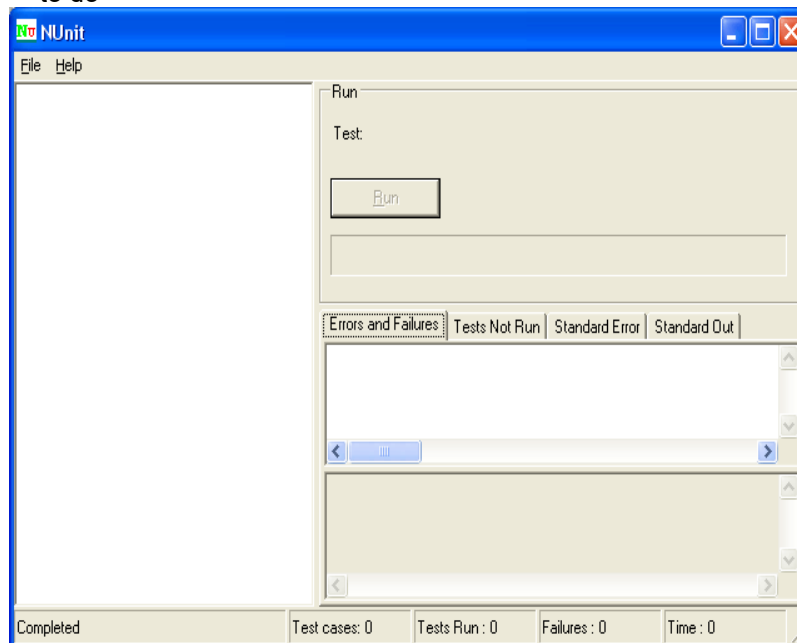
3. Compile your project.
<< to do >>

Step 4: Starting NUnit-GUI
1. Start the NUnit-GUI application
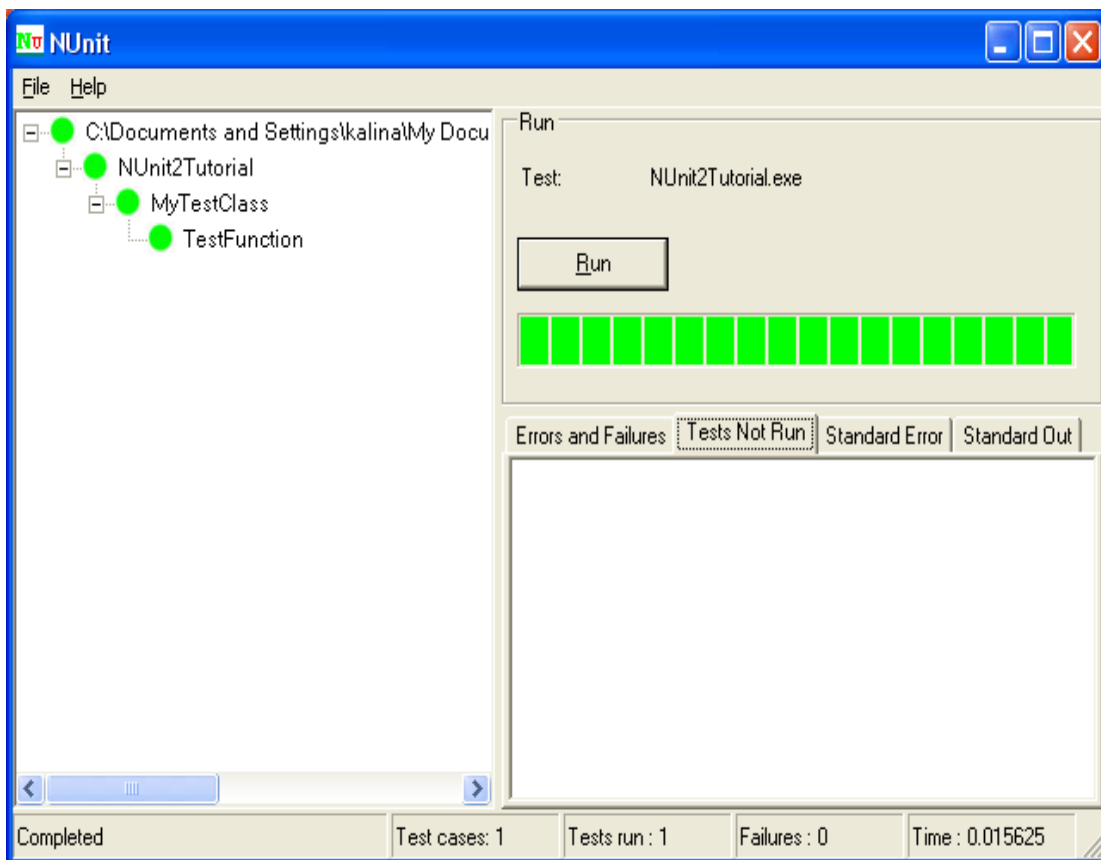<<to do>>

Capgemini Internal

2. Select the library of your project in order to run the containing tests.

<< to do>>

You will immediately notice that NUnit2 is looking through the complete compilation unit and finds any TestFixtures. In our case the TestFixture is called MyTestClass and the only test is TestFunction.

Step 5: Testing with NUnit-GUI

Click on the "Run" button.

You will see a green bar which tells you that all tests succeeded.

Step 6: Write the following code in the TestFunction.

<<to do>>

```
namespace TestDemo1
{
        [TestFixture]
        public class MyTestClass
        {
                [Test]
                public void TestFunction()
                {
                        Assert.AreEqual(1==2);
                }
        }
}
```

Step 7: << to do>>Compile your project with this new code and switch to NUnit-GUI. You will notice that NUnit has noticed that the library has changed and it has reloaded it for you. Now click run again.
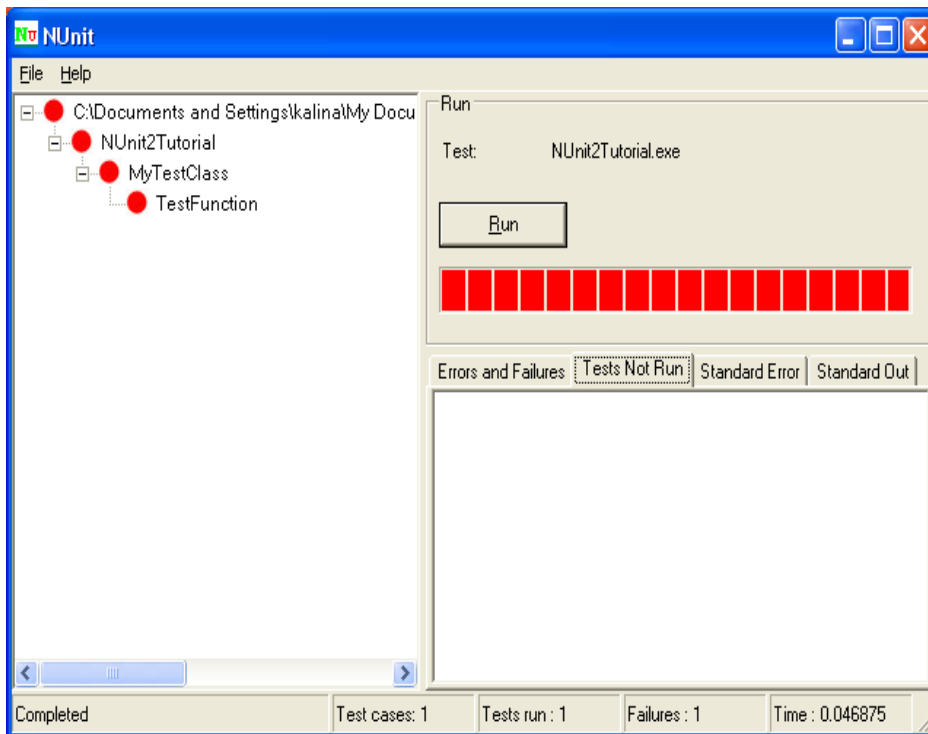
Step 8: <<to do>>Fix this broken test. You would simply change the assertion and go back to NUnit-GUI and hit the run button.

Lab 1 B: Problem statement

Suppose we are writing a bank application and we have a basic domain class – Account. Account supports operations to deposit, withdraw, and transfer funds.
Write the test cases for the above mentioned functions.

Solution:
Step 1: Create a new C# project of Library type.

Step 2: Create the Account class as follows
<< to do>>

```
namespace Bank
{
  public class Account
  {
    private float balance;
    public void Deposit(float amount)
    {
      balance+=amount;
    }

    public void Withdraw(float amount)
    {
      balance-=amount;
    }
    public void TransferFunds(Account destination, float amount)
    {
    }

    public float Balance
    {
      get{ return balance;}
    }
  }
}
```

Step 3: Create the New project to write Test Class. Write a Test for this class.

<< to do>> The method to be tested is TransferFunds()

```
namespace Bank
{
  using NUnit.Framework;

  [TestFixture]
  public class AccountTest
  {
```

```
    [Test]
    public void TransferFunds()
    {
      Account source = new Account();
      source.Deposit(200.00F);
      Account destination = new Account();
      destination.Deposit(150.00F);

      source.TransferFunds(destination, 100.00F);
      Assert.AreEqual(250.00F, destination.Balance);
      Assert.AreEqual(100.00F, source.Balance);
    }
  }
}
```

Step 4: Compile the code & generate a DLL.

Step 5: Start NUnit & Select the TestCase DLL created above.
        Run the Test.

The Test should fail. The test has failed because we have not implemented the TransferFunds method yet.

Step 6: Write the code for the TransferFunds Function
<< to do>>

```
public void TransferFunds(Account destination, float amount)
{
        destination.Deposit(amount);
        Withdraw(amount);
}
```

Step 7: Recompile the code & Run the Test again. The test should pass now.
<<to do>>

Step 8: Add the MinimumBalance property to the Account class. It should be readonly.
<< to do>>

Step 9: We will use an exception to indicate an overdraft: Add a new class as follows:

```
namespace Bank
{
  using System;
  public class InsufficientFundsException : ApplicationException
  {
  }
}
```

Step 10: Write a Test method to ensure that the function should throw exception of certain type.

Capgemini Internal

```
[Test]
[ExpectedException(typeof(InsufficientFundsException))]
public void TransferWithInsufficientFunds()
{
        Account source = new Account();
        source.Deposit(200.00F);
        Account destination = new Account();
        destination.Deposit(150.00F);
        source.TransferFunds(destination, 300.00F);
}
```

Note: This test method in addition to [Test] attribute has an [ExpectedException] attribute associated with it – this is the way to indicate that the test code is expecting an exception of a certain type; if such an exception is not thrown during the execution – the test will fail.

Step 11: Compile the code & Run the test.
The Test should fail.

Step 12: Fix the Account code.

```
public void TransferFunds(Account destination, float amount)
{
        destination.Deposit(amount);
        if(balance-amount<minimumBalance)
                throw new InsufficientFundsException();
        Withdraw(amount);
}
```

Step 13: Compile the code & Run the test.
The Test should pass.

Step 14: The code we've just written we can see that the bank may be losing money on every unsuccessful funds Transfer operation. Write a test to confirm our suspicions. Add this test method:

```
[Test]
public void TransferWithInsufficientFundsAtomicity()
{
        Account source = new Account();
        source.Deposit(200.00F);
        Account destination = new Account();
        destination.Deposit(150.00F);
        try
        {
                source.TransferFunds(destination, 300.00F);
        }
```

```
            catch(InsufficientFundsException expected)
            {
            }

            Assert.AreEqual(200.00F,source.Balance);
            Assert.AreEqual(150.00F,destination.Balance);
}
```

Step 15: Compile the code & Run the Test.
The test should fail

Step 16: Write the Correct code to fix the Error.

```
public void TransferFunds(Account destination, float amount)
{
        if(balance-amount<minimumBalance)
                throw new InsufficientFundsException();
        destination.Deposit(amount);
        Withdraw(amount);
}
```

Step 17: To temporarily ignore the test, add the following attribute to your test method

```
[Test]

[Ignore("Decide how to implement transaction management")]

public void TransferWithInsufficientFundsAtomicity()

{

        // code is the same

}
```

Step 18: Compile & Run the test. You should get a yellow bar. Click on "Tests Not Run" tab to check the reason.
    <<to do>>

Step 19: <<to do>> Perform some refactoring on the code. All test methods share a common set of test objects. Extract this initialization code into a setup method and reuse it in all of our tests.

Note that Init method should have the common initialization code, void return type, no parameters, and marked with [SetUp] attribute.

Step 20: Compile & run the test.

Capgemini Internal

Assignment to do:

Following is the user defined simple implementation of Stack. Write the test cases for this class in NUnit to test whether all operations are performed correctly.

```csharp
public class MyStackClass
{
        private ArrayList elements = new ArrayList();
        public bool IsEmpty
        {
           get
           {
               return (elements.Count == 0);
           }
        }
        public void Push(object element)
        {
           elements.Insert(0, element);
        }
        public object Pop()
        {
           object top = Top();
           elements.RemoveAt(0);
           return top;
        }
        public object Top()
        {
           if(IsEmpty)
               throw new InvalidOperationException("Stack is Empty");

           return elements[0];
        }
}
```

Lab 1 C: Problem statement

The purpose of this lab assignment is to acquaint you with the process of writing test cases using Microsoft Test Framework. In this lab assignment, we will be using Account class created in the lab assignment 1B.

Follow the steps given below to create a unit test project using Microsoft Visual Studio 2013:

1. On the **File** menu, choose **Add**, and then choose **New Project ...**.
2. In the New Project dialog box, expand **Installed**, expand **Visual C#**, and then choose **Test**.
3. From the list of templates, select **Unit Test Project**.

Capgemini Internal

4. In the **Name** box, enter BankTest, and then choose **OK**.
The **BankTests** project is added to the the **Bank** solution.
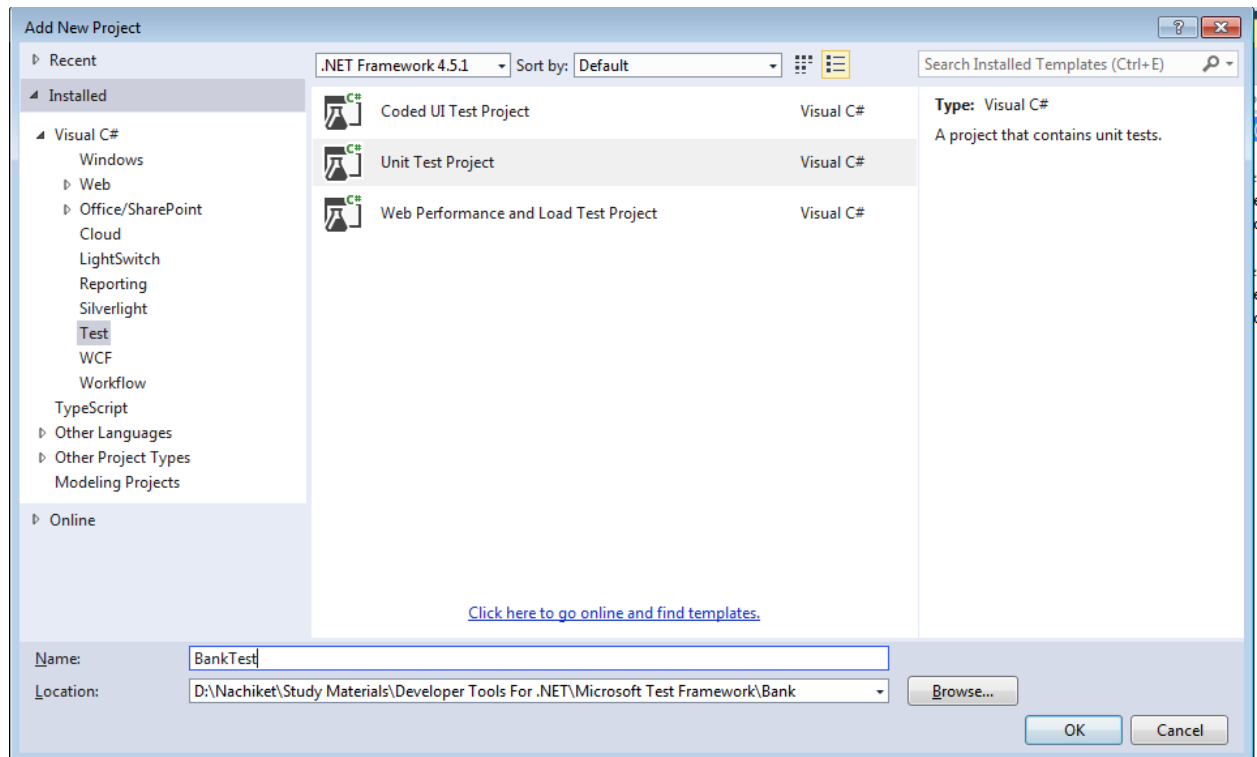
5. In the **BankTests** project, add a reference to the **Bank** solution.
In Solution Explorer, select **References** in the **BankTests** project and then choose **Add Reference...** from the context menu.

6. In the Reference Manager dialog box, expand the **Solution** and then check the **Bank** item.

7. Rename the UnitTest1.cs file in the BankTests project to BankAccountTests.

8. Observe that the following namespace is included by default in the BankAccountTests.cs file:

   using Microsoft.VisualStudio.TestTools.UnitTesting;

   Add the following line in the BankAccountTests.cs file:
   using Bank;

9. Apply the [TestClass] attribute to BankAccountTest class and add the Debit_WithValidAmount_UpdatesBalance() method to the class. After adding this method, the BankAccountTest.cs class should look like this:

| **13** / 21

Capgemini Internal

```
namespace BankTest
{
  [TestClass]
  public class BankAccountTests
  {
    [TestMethod]
    public void Debit_WithValidAmount_UpdatesBalance()
    {
      // arrange
      double beginningBalance = 11.99;
      double debitAmount = 4.55;
      double expected = 7.44;
      BankAccount account = new BankAccount("Mr. Bryan
                            Walton", beginningBalance);

      // act
      account.Debit(debitAmount);

      // assert
      double actual = account.Balance;
      Assert.AreEqual(expected, actual, 0.001, "Account not
                 Debited correctly");
    }
  }
}
```

Build and Run the Test:

To build and run the test

1. On the **Build** menu, choose **Build Solution**.
   If there are no errors, the UnitTestExplorer window appears with **Debit_WithValidAmount_UpdatesBalance** listed in the **Not Run Tests**group. If Test Explorer does not appear after a successful build, choose **Test** on the menu, then choose **Windows**, and then choose **Test Explorer**.

2. Choose **Run All** to run the test. As the test is running the status bar at the top of the window is animated. At the end of the test run, the bar turns green if all the test methods pass, or red if any of the tests fail.

3. Observe that in Test Explorer, the red/green bar has turned green, and the test is moved to the **Passed Tests** group.

Lab 2: Using NCover to Test Code Coverage

Goals	• Understand working of NCover to Test the code coverage • Understand the steps involved in checking the code coverage.
Time	90 Minutes
Lab Setup	VS.NET 2013, NUnit 2.x Framework, NCover 1.x & Testdriven.NET

Lab 2 A. Problem statement

Write the production code & check the same for code coverage.

Solution:

Step 1: Create your project of Library Type and add methods to it.

```
namespace NCoverDemo.Sample
{
   public class Calculation
   {
      public int AddNumbers(int firstNumber, int secondNumber)
      {
         return firstNumber + secondNumber;
      }

      public int MultiplyNumbers(int firstNumber, int secondNumber)
      {
         return firstNumber + secondNumber;
      }
   }
}
```

Step 2: Create another project in the same solution of Library type and add test cases to it

```
namespace NCoverDemo.SampleTest
{
   [TestFixture]
   public class CalculationTest
   {
      Calculation calc = null;

      [TestFixtureSetUp]
      public void Setup()
      {
         calc = new Calculation();
      }
```
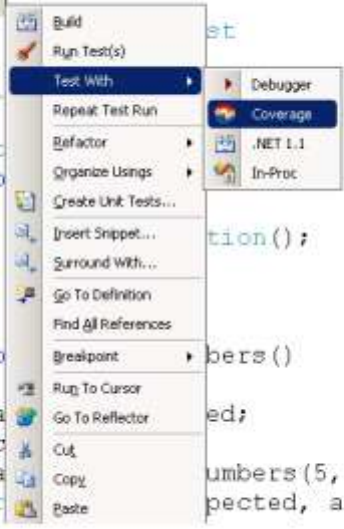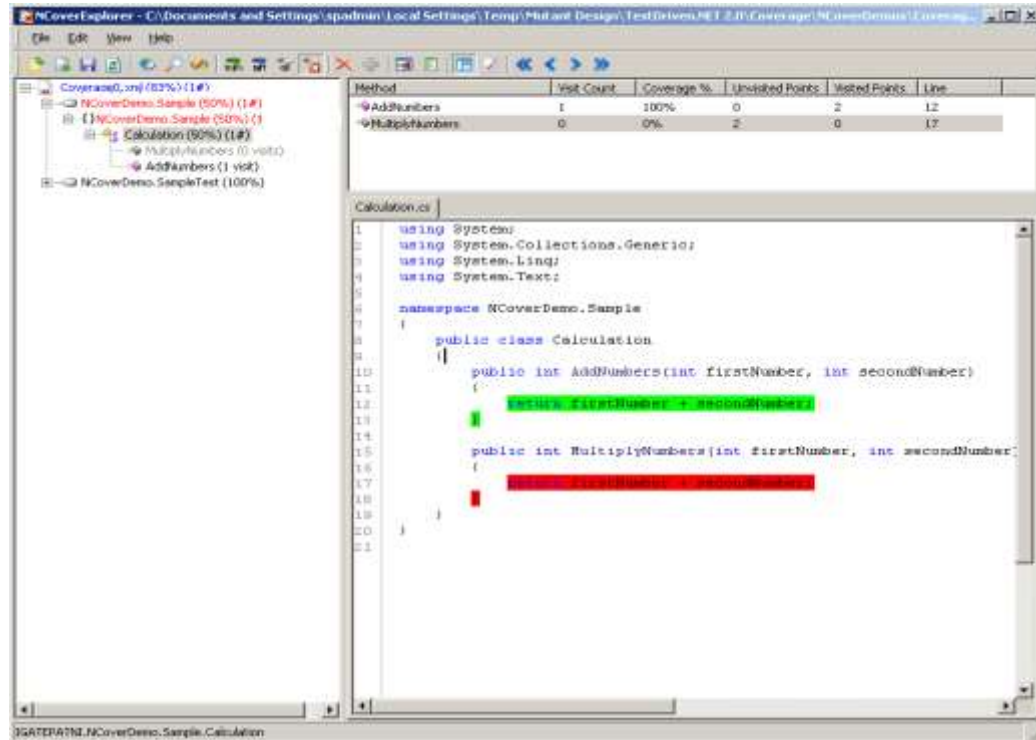
```
      [Test]
      public void TestAddNumbers()
      {
         int actual, expected;
         expected = 11;
         actual = calc.AddNumbers(5, 6);
         Assert.AreEqual(expected, actual);

      }
      [TestFixtureTearDown]
      public void TearDown()
      {
         calc = null;
      }
   }
}
```

Step 3: Right click the TestFixture attribute and test for code coverage using NCover

```
namespace NCoverDemo.SampleTest
{
    [TestFixture]
    public class                    et
    {
        Calculati
        [TestFixt
        public vo
        {
            calc               tion();
        }

        [Test]
        public vo               bers()
        {
            int a              ed;
            expec
            actua              umbers(5, 6);
            Asser              pected, actual);
        }
    }
```

Context menu items:
- Build
- Run Test(s)
- Test With ▶
  - ▶ Debugger
  - Coverage
  - .NET 1.1
  - In-Proc
- Repeat Test Run
- Refactor ▶
- Organize Usings ▶
- Create Unit Tests...
- Insert Snippet...
- Surround With...
- Go To Definition
- Find All References
- Breakpoint ▶
- Run To Cursor
- Go To Reflector
- Cut
- Copy
- Paste

Capgemini Internal

NCoverExplorer shows only 50% of code coverage,highlighting the visited code in green background and unvisited code in red background

Step 4: To create the code coverage report Press F6

The information contained in this document is proprietary and confidential. For Capgemini only. | **18** / 21

Capgemini Internal
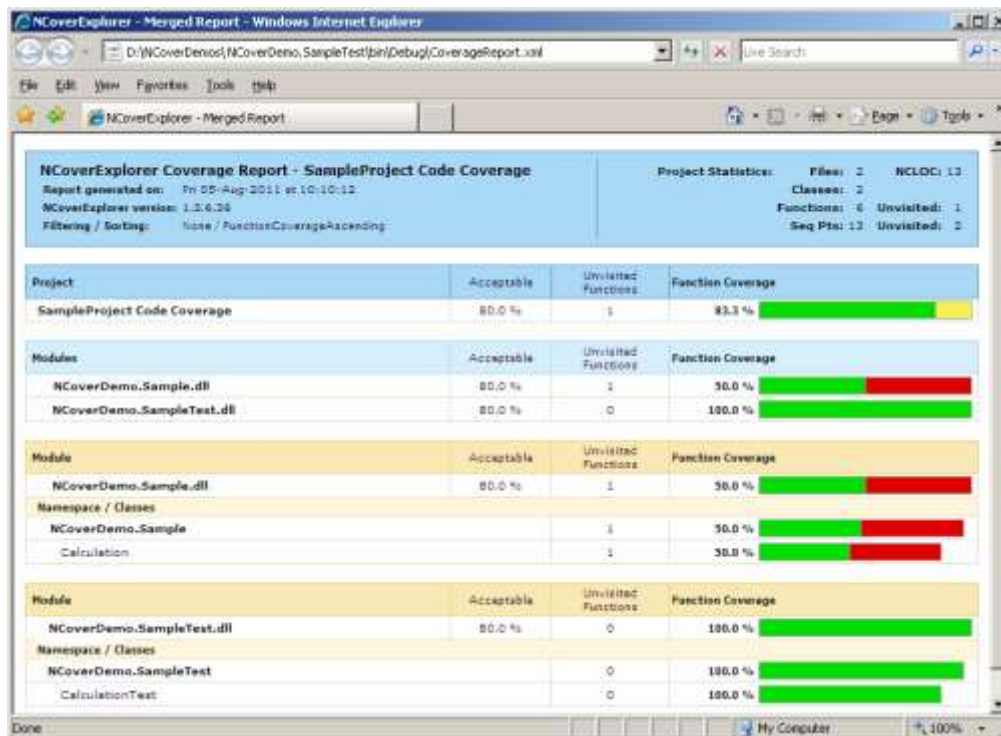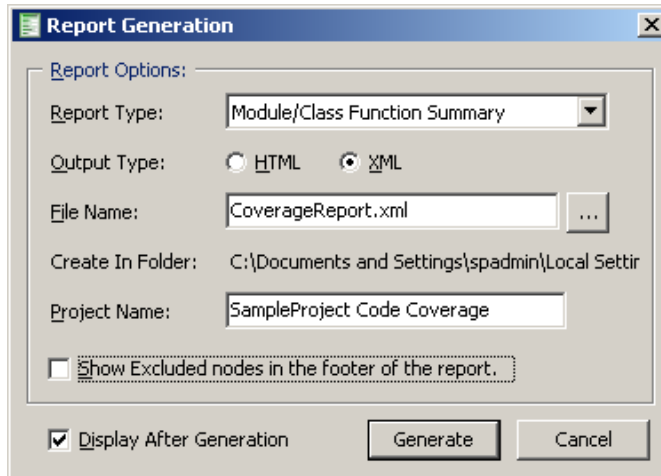
<<TODO> ADD the test case for MultiplyNumbers method and generate a code coverage report achieving 100% code coverage

|

Lab 3: Using Log4Net for logging service

Goals	• Understand working of Log4Net logging service • Understand the steps involved in creating logs using Log4Net.
Time	60 Minutes
Lab Setup	VS.NET 2013, Log4net 1.x

Lab 3 A. Problem statement

Write the production code & add logging service to it.

Solution:

Step 1: Add a reference of Log4net.dll to the project

Step 2: In the App.config file, under Configuration->Configsections, add the following section

```
<section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler, log4net" />
```

Step 3: In app.config, add a new section "<log4net>". This section will contain all the settings related to the Log4net configuration.

Step 4: In app.config, under "log4net" section, add the required appender (output target).

```
<appender name="ConsoleAppender" type="log4net.Appender.ConsoleAppender" >
        <layout type="log4net.Layout.PatternLayout">
<param name="ConversionPattern" value="%p - %d{dd-MM-yyyy hh:mm:ss tt} – %m%n"/>
        </layout>
</appender>

<logger name="ConsoleLogger">
    <level value="ALL" />
    <appender-ref ref="ConsoleAppender" />
 </logger>
```

Step 5: In the app.config file, under "log4net" section, for each appender, add logger and the level (DEBUG, INFO …)

Step 6: To log any information/error/warning, call the appropriate method in the following manner:

```
namespace Log4NetApp.Demo01
{
   public class Sample
   {
      private static readonly ILog log =
LogManager.GetLogger("ConsoleLogger");

      static void Main(string[] args)
      {
         //This calls the default configurator for log4net
         XmlConfigurator.Configure();

         log.Info("Entering application.");

         for (int counter = 1; counter <= 10; counter++)
         {
            log.DebugFormat("Inside of the loop (Counter = {0})", counter);
         }

         try
         {
            throw new NotImplementedException("Testing log4net");

         }
         catch (NotImplementedException ex)
         {
            log.Fatal(ex.Message);
         }

         log.Info("Exiting application.");
      }
   }
}
```

<<todo>
Change the level, in such a way that the console appender should log only fatal messages