Unit Testing and TDD

Lesson 01 : Unit Testing using NUnit and Microsoft Test Framework

Capgemini

Lesson Objectives

- In this lesson, you will learn:
  - Unit Testing
  - Advantages of Unit Testing
  - Limitations of Unit Testing
  - NUnit Framework
  - Microsoft Test Framework
  - 3 A's of Testing
  - Working with Test Explorer
  - Code Coverage

What is Unit Testing?

- A unit test is a piece of code written by a developer that exercises a very small, specific area of functionality of the code being tested
- Usually a unit test exercises some particular method in a particular context

Why Unit Testing?

- It will make your designs better and drastically reduce the amount of time you spend debugging
- To avoid having to compile and run the entire application just to check a single function

Advantages of Unit Testing

- Unit testing helps eliminate uncertainty in the pieces themselves and can be used in a bottom-up testing style approach
- By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier
- Unit testing provides a sort of "living document".
- Clients and other developers looking to learn how to use the class can look at the unit tests to determine how to use the class to fit their needs and gain a basic understanding of the API

Limitations of Unit Testing

- Unit-testing will not catch every error in the program. By definition, it only tests the functionality of the units themselves
- Therefore, it will not catch integration errors, performance problems and any other system-wide issues
- In addition, it may not be easy to anticipate all special cases of input the program unit under study may receive in reality. Unit testing is only effective if it is used in conjunction with other software testing activities

Xunits Tools

- In order to run tests in the code, people typically use XUnit tools like JUnit for Java, NUnit for .NET (nunit.org)
- These tools often come in two flavors: command-line and GUI
- XUnit tools allow you to run all your tests and see which passed and which failed
- Best of all, these tools are free

Nunit Testing Framework

- NUnit is a unit-testing framework for all .Net languages
- NUnit brings xUnit to all .NET languages

Creating Tests

- Create a separate class library or simply create a new class in the same file as your production code
- Create the tests by using attributes to mark tests
- Write the test in VB or C#
- Use asserts in order to claim that certain things are true, are equal, and so forth

TestFixture Attribute

- The TestFixture attribute marks a class as containing tests
- Class must reference NUnit.Framework.DLL

```
namespace UnitTestingExamples
{
  using System;
  using NUnit.Framework;

  [TestFixture]
  public class SomeTests
  {
  }
}
```

Test Attribute

- The Test Attribute marks a method as a test
- The method must be:
  - Public
  - Parameterless
  - Void in C#, a Sub in VB .NET

Test Attribute

- The following code illustrates the use of this attribute

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

Assert Class

- The Assert class contains a number of static (shared) methods
- These methods are used to return a True or False from the test
- Methods include:
  - AreEqual
  - AreSame
  - IsTrue
  - IsFalse
  - IsNull
  - IsNotNull
  - Fail
  - Ignore

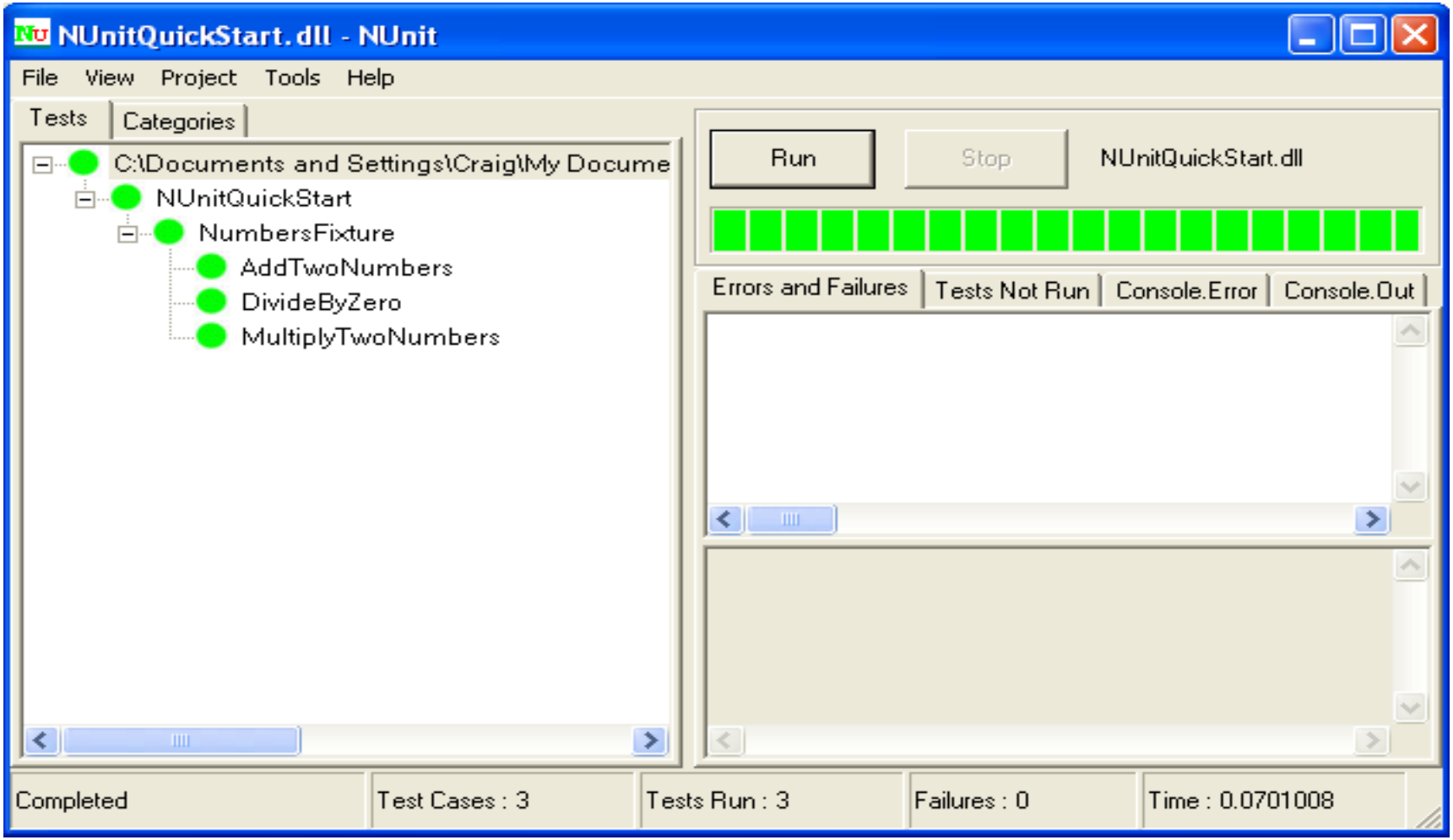
Example : A Simple Test

```
[TestFixture]
public class MyTests
{
  [Test]
  public void AddTest()
  {
    int sum=myMath.Add(2,3);
    Assert.AreEqual(5,sum);
  }
}
```

Running Tests

- NUnit comes with two different Test Runner applications: a Windows GUI app and a console XML app
- To use the GUI app, just run the application and tell it where your test assembly resides
- The test assembly is the class library (or executable) that contains the Test Fixtures
- The app will then show you a graphical view of each class and test that is in that assembly
- To run the entire suite of tests, simple click the Run button

Nunit GUI Screen



Red/Green/Refactor

- Red/Green/Refactor is the TDD mantra, and it describes the three states you go through when writing code using TDD methods
- Red
  - Write a failing test
- Green
  - Write the code to satisfy the test
- Refactor
  - Improve the code without changing its functionality

ExpectedException Attribute

- Use the ExpectedException attribute when a method should raise an exception on a particular failure condition

```
[Test]
[ExpectedException(typeof(InvalidOperationException))]
public void Foo()
{//…}
```

SetUp and TearDown

- SetUp and TearDown are used to mark methods that should be called before and after each test
  - You may have only one SetUp and one TearDown
- Used to reinitialize variables or objects for each test
  - Tests should not be dependent on each other!

TestFixtureSetUp & TestFixtureTearDown

- The TestFixtureSetUp and TestFixtureTearDown attributes mark methods that run before and after each test fixture is run
  - You may have only one TestFixtureSetUp and one TestFixtureTearDown
- Usually used to open and close database connections, files, or logging tools

Ignore Attribute

- The Ignore attribute is an optional attribute that can be added to a unit test method
- This attribute instructs the unit test engine to ignore the associated method
- Requires a string indicating the reason for ignoring the test to be provided

Ignore Attribute

```
[TestFixture]
  public class SomeTests
  {
    [Test]
    [Ignore("We're skipping this one for now.")]
    public void TestOne()
    {
      // Do something...
    }
  }
```

Demo

- Using NUnit Framework

Microsoft Test Framework

- Visual Studio provides an inbuilt framework for writing unit test cases.
- The framework includes a set of attributes, assertion classes and configuration files for manipulating the test environment.
- The test code is created by adding attributes to the classes and methods used for writing tests.
- The framework integrated with the IDE allows creation of test suites and addition of test cases to them with minimal effort.
- Class must reference Microsoft.VisualStudio.TestTools.UnitTesting

Creating Tests Using Visual Studio

- Create a separate class library or simply create a new class in the same project which contains the code to be tested.
- Create the tests by using predefined attributes.
- Write the test case code .
- Use appropriate Asserts.

Test Class

- The class which contains the methods used for unit testing is marked with an attribute called TestClassAttribute.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TestProject
{
    [TestClass]
    public class SampleTestClass
    {
            // unit test code written here
     }
}
```

Test Method

- The TestMethodAttribute is used to mark a method as a test method.
- A method marked with this attribute must be public, void and without parameters.

```
[TestClass]
public class SampleTestClass
{
    [TestMethod]
    public void TestOne()
    {
        //Unit Test logic written here
    }
}
```

TestInitialize Attribute

- Each test case might require a common initialization, like opening a data connection or simply instantiating an object.
- Such code must not be placed in each test case, but in a separate method.
- The TestInitialize attribute is used to mark the methods to run before any test.
- Used to allocate or configure any resources needed by all or many tests in a test class.

```
[TestInitialize]
 public void Init()
 {
        //Initialization logic written here
 }
```

TestCleanUp Attribute

- This attribute is used to run code after each test has completed.
- Used to deallocate resources used.

```
[TestCleanup]
 public void Destroy()
 {
     //DeInitialization logic goes here
 }
```

ExpectedException Attribute

- It is not sufficient to validate only the passing test cases.
- Error cases must be handled correctly.
- The ExpectedException attribute ensures that , the test case passes only if an exception which matches the type specified is thrown.

```
[ExpectedException(typeof(ArithmeticException))]
 public void TestException()
 {

 }
```

Using Test Windows

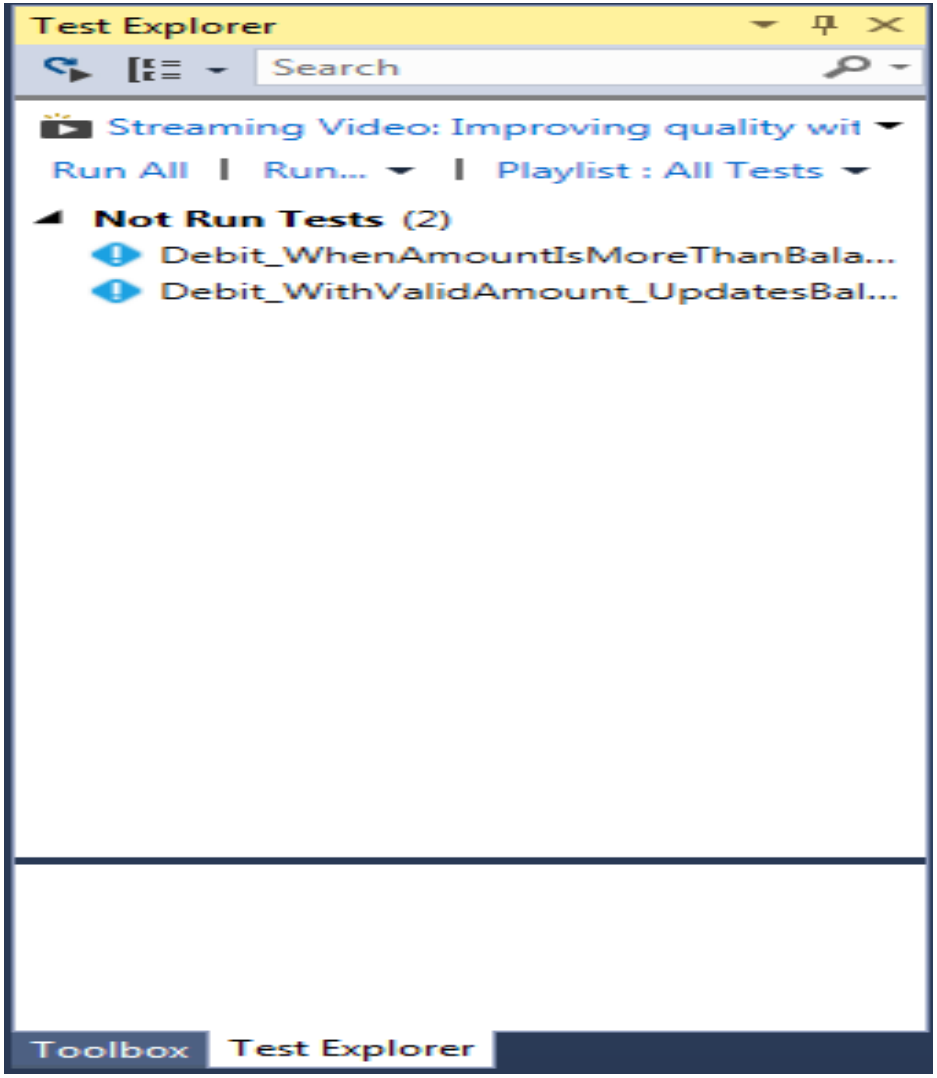
- Test View Window
- Test Manager Window
- Test Results
- Code Coverage

3 A’s of Testing

- Unit Testing is a 3-Step Process
- Arrange
  - Do necessary setup of the test
- Act
  - Execute the test
- Assert
  - Check and verify the returned result with expected results

Working with Test Explorer

- When you build the test project, the tests appear in Test Explorer.
- If Test Explorer is not visible, choose Test on the Visual Studio menu, choose Windows, and then choose Test Explorer.



Selectively Executing Test Cases

- You can run all the tests in the solution, all the tests in a group, or a set of tests that you select.
  - To run all the tests in a solution, choose the Run All icon.
  - To run all the tests in a default group, choose the Run icon and then choose the group on the menu.
  - Select the individual tests that you want to run, open the right-click menu for a selected test and then choose Run Selected Tests.
  - If individual tests have no dependencies that prevent them from being run in any order, turn on parallel test execution in the settings menu of the toolbar. This can noticeably reduce the time taken to run all the tests.

Best Practices for Tests

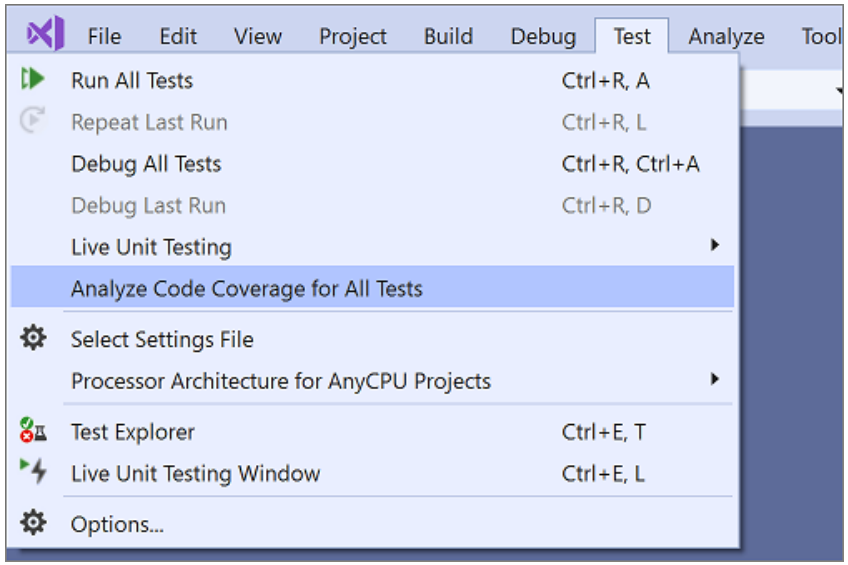
- The name of test should consist of three parts:
  - The name of the method being tested.
  - The scenario under which it's being tested.
  - The expected behaviour when the scenario is invoked.
- Standard naming convention of the test methods:
  - Test name should express a specific requirement
  - Test name should include the expected input or state and the expected result for that input or state
  - Test name should be presented as a statement or fact of life that expresses workflows and outputs
  - Test Name should only begin with Test if it is required by the testing framework or if it eases development and maintenance of the unit tests in some way.
  - Test name should include name of tested method or class
  - Bad Naming Convention
    - Test_Single()
  - Better Naming Convention
    - Add_SingleNumber_ReturnsSameNumber()

Code Coverage

- To determine what proportion of project's code is actually being tested by coded tests such as unit tests, you can use the code coverage feature of Visual Studio.
- To guard effectively against bugs, your tests should exercise or 'cover' a large proportion of your code.
- Code coverage analysis can be applied to both managed (CLI) and unmanaged (native) code.
- Code coverage is an option when you run test methods using Test Explorer.
- The results table shows the percentage of the code that was run in each assembly, class, and method.
- In addition, the source editor shows you which code has been tested.

Analyze Code Coverage

- On the Test menu, select Analyze Code Coverage for All Tests or you can also run code coverage from the Test Explorer tool window.



- After the tests have run, to see which lines have been run, choose Show Code Coverage Coloring icon Show Code Coverage Coloring in the Code Coverage Results window. By default, code that is covered by tests is highlighted in light blue.

Demo

- Creating Tests project using Microsoft Test Framework

Summary

- In this lesson, you have learnt about:
  - NUnit provides a powerful framework for unit testing
  - Developers assisted in implementing repeatable, best-practice processes
  - Microsoft test framework provides the unit testing environment for the .NET application
  - To determine what proportion of project's code is actually being tested by coded tests such as unit tests, you can use the code coverage feature of Visual Studio.