NUnit With C#

**NUnit Testing Framework**

NUnit is a unit-testing framework for .NET applications in which the entire application is isolated into diverse modules. Each module is tested independently to ensure that the objective is met. The NUnit Framework caters to a range of attributes that are used during unit tests. They are used to define Test -Fixtures, Test methods, ExpectedException, and Ignore methods.

**TextFixture Attribute**

The TestFixture attribute is an indication that a class contains test methods. When you mention this attribute to a class in your project, the Test Runner application will scan it for test methods. The following code illustrates the usage of this attribute as in the following

**using System;**

**using NUnit.Framework;**

**using System.Text;**

**namespace UNitTesting {**

**[TestFixture]**

**public class Program {}**

**}**

**Test Attribute**

The Test attribute indicates that a method within a test fixture should be run by the Test Runner application. The method must be public, return void, and will not be run when the Test Fixture is run. The following code depicts the use of this attribute as in the following

[TestFixture]

**public** **class** Program {

    [Test]

**public** **void** Test() { ...

    }

}

**Assert Class**

the Assert class is used to confirm whether the test cases are producing the expected result or not using its auxiliary methods such as AreEqual() or AreNotEqual().

**ExpectedException Attribute**

You could fortify your code to handle various exceptions by using try...Catch blocks. But sometimes you have some circumstances where you actually want to ensure that an exception occurs. To overcome such a problem you should use the ExpectedException attribute as in the following

[TestFixture]

**public** **class** Program {

    [Test]

    [ExpectedException(**typeof**(DivideByZeroException))]

**public** **void** Test() {

**int** i = 10, j = 0, x;

        x = i / j;

    }

}

In the previous code, we are intentionally committing a divide by zero exception that is to be detected during the NUnit Test execution.

**Ignore Attribute**

The Ignore attribute is required to indicate that a test should not be run on a particular method. Use the Ignore attribute as in the following:

[TestFixture]

**public** **class** Program

{

[Test]

[Ignore("This method is skipping")]

**public** **void** Test()

{

...

}

}

**Simple Business Logic Project**

In the following, we are implementing Discounts on specific sales made by customers such as 5% discount on purchases of $1000 to $1999, 10% on $2000 to $4999, and so on. Such a discount will be deducted automatically in the final payment based on some implicit calculation stated in the business logic.

1. Create a C# .Net Class Library project called UnitTest to produce a Dll.
2. Rename the class1 to UtilityLib and use the following business logic

using System;

using System.Text;

namespace UnitTest {

**public** classUtilityLib {

**public** **double** calculateDiscount(**double** SalesAmnt) {

**double** DiscountPrice = 0.0;

**if** (SalesAmnt == 0 || SalesAmnt < 0) {

                thrownew ArgumentException(" Sales Amount should not be 'Zero/Negative'");

            } **else** **if** (SalesAmnt >= 1000 && SalesAmnt < 2000) {

                // 5% Discount

                DiscountPrice = SalesAmnt - (SalesAmnt \* 0.05);

            } **else** **if** (SalesAmnt >= 2000 && SalesAmnt < 5000) {

                // 10% Discount

                DiscountPrice = SalesAmnt - (SalesAmnt \* 0.1);

            } **else** **if** (SalesAmnt >= 5000 && SalesAmnt < 20000) {

                // 50% Discount

                DiscountPrice = SalesAmnt - (SalesAmnt \* 0.5);

            } **else** {

                // No Discount

                DiscountPrice = SalesAmnt - 0.0;

            }

**return** DiscountPrice;

        }

    }

}

1. Now build the project and note that the UnitTest.dll file is created in the solution Bin/Debug folder that would be referenced in the Unit Test project later.

**Test-Driven Project Development**

There are a few steps to execute the unit test using NUnit as follows,

1. Create another C# .Net Class Library project called UtilityLibtesting.
2. Then add a reference for the UnitTest.dll and NUnit framework DLL files that reside in the "\Program Files\NUnit 2.6.2\bin\framework\nunit.framework.dll" directory.

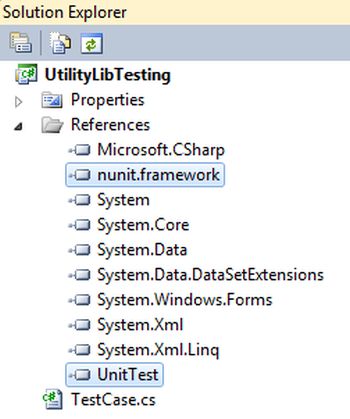


Figure 1.3 Add references of NUnit framework and UnitTest.dll

1. Now add the namespace definition to the class library file as follows:

**using** NUnit.Framework;

**using** UnitTest;

1. Now rename the class1.cs to TestCase.cs with TextFixture attribute.
2. It is time to write test case methods to verify whether the business logic stated in the DLL file is producing the expected result or not.
3. So we write another independent method for each corresponding function given in the DLL file with the [test] attribute and we also use the assert class to AreEqual() method in order to confirm the desired result. The AreEqual() method would take two arguments; the first is the expected result and the second specific method with an argument defined in the business logic class.

// Test Case#1: Sales Amount is Greater or Equal than 1000 : Verification

[Test]

**public** **void** OneThousand\_G\_E() {

    UtilityLib obj = **new** UtilityLib();

    Assert.AreEqual(950, obj.calculateDiscount(1000));

}

1. We also put the condition for actual test cases by using the assert class AreNotEqual() method in which we pass any value as the first argument as in the following

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_Test for Actual Results\_\_\_\_\_\_\_\_\_\_\_\_

// Test Case#6: Sales Amount is not producing expected Result : Verification

[Test]

**public** **void** OneThousand() {

    UtilityLib obj = **new** UtilityLib();

    Assert.AreNotEqual(930, obj.calculateDiscount(1000));

}

Thereafter we specify the rest of the code for each condition as in the following

using System;

using NUnit.Framework;

using UnitTest;

namespace UtilityLibTesting {

    [TestFixture]

**public** classTestCase {

        //\_\_\_\_\_\_\_\_\_\_Test for Expected Results\_\_\_\_\_\_\_\_\_\_

        // Test Case#1: Sales Amount is Greater or Equal than 1000 : Verification

        [Test]

**public** **void** OneThousand\_G\_E() {

            UtilityLib obj = **new** UtilityLib();

            Assert.AreEqual(950, obj.calculateDiscount(1000));

        }

        // Test Case#2: Sales Amount is Greater or Equal than 2000 : Verification

        [Test]

**public** **void** TwoThousand\_G\_E() {

            UtilityLib obj = **new** UtilityLib();

            Assert.AreEqual(1800, obj.calculateDiscount(2000));

        }

        // Test Case#3: Sales Amount is Greater or Equal than 5000 : Verification

        [Test]

**public** **void** FiveThousand\_G\_E() {

            UtilityLib obj = **new** UtilityLib();

            Assert.AreEqual(2500, obj.calculateDiscount(5000));

        }

        // Test Case#4: Sales Amount is 0 : Verification

        [Test]

**public** **void** ZeroAmount() {

            UtilityLib obj = **new** UtilityLib();

**try** {

                obj.calculateDiscount(0);

            } **catch** (Exception e) {}

        }

        // Test Case#5: Sales Amount is below 1000 : Verification

        [Test]

**public** **void** OneThousand\_Below() {

            UtilityLib obj = **new** UtilityLib();

            Assert.AreEqual(999, obj.calculateDiscount(999));

        }

        //\_\_\_\_\_\_\_\_\_\_Test for Actual Results\_\_\_\_\_\_\_\_\_\_

        // Test Case#6: Sales Amount is not producing expected Result : Verification

        [Test]

**public** **void** OneThousand() {

            UtilityLib obj = **new** UtilityLib();

            Assert.AreNotEqual(930, obj.calculateDiscount(1000));

        }

        // Test Case#7: Sales Amount is not producing expected Result : Verification

        [Test]

**public** **void** TwoThousand() {

            UtilityLib obj = **new** UtilityLib();

            Assert.AreNotEqual(1900, obj.calculateDiscount(2000));

        }

    }

}

1. Finally build the project and you will notice that the UtilityLibTesting.dll file is created in the project solution directory to be used later during the unit testing in the NUnit application.

**Running Test Case**

Until now we have seen the basics of the code, now it is time how to run your tests using the NUnit GUI. To use the GUI application, just simply run the software and target (open the project of) your test assembly where it resides. The test assembly is the class library (or executable) that contains the Test Fixtures. The application will then show you a graphical view of each class. To run the entire suite of tests, simply click the Run button. If you want to run only one Test Fixture or even just a single test, you can double-click it in the tree. After opening the project file, all the methods stated in the DLL file will show up in the left pane as in the following;

Figure 1.5 Test Method Loading

Now hit the Run button on the right side to start the Unit Automotive testing. It will take some time to verify all the conditions given in the test cases. If all the methods function properly as specified by the test condition then the GUI will produce the results in the Green Progress bar as in the following:

Figure 1.6 NUnit Test Report with No Error

In the scenario, if the code has some built-in programming glitches, then they are easily rectified during the NUnit Testing process. The following picture depicts a problem in the TwoThousand() method with the cross sign. Here the report indicates that all the method implementations are working properly except one method:

Figure 1.7 NUnit Test Fail output

We can go deeper into the error by using the Errors and Failure tab in the right bottom. This will display the exact error in the source code as in the following: