**NUnitTesting**

In this IT world a unit refers to simply a smallest piece of code which takes an input, does certain operation, and gives an output.

And testing this small piece of code is called Unit Testing.

A lot of unit test frameworks are available for .Net nowadays, if we check in Visual Studio we have MSTest from Microsoft integrated in Visual Studio.

Some 3rd party frameworks are:

NUnit

MbUnit

Out of all these Nunit is the most-used testing Framework.

Required DLL

1. NUnit.dll
2. NUnit3TestAdapter (Transient is a single instance per code request.)

**[TestFixture]**

The class that is to be tested using Nunit should be decorated with TextFixture.

**[Test]**

This attribute identifies the method to be tested. If we do not write this attribute then we can't identify the test in Test Explorer. We have an Assert class with the following methods for validating different conditions in the TestFixture.

**[SetUp]**

***Method decorated with SetUp will be execute first before each test execution.***

This attribute is used when you want to execute a piece of code in each test case. It identifies a method to be executed each time before a TestMethod/Test is executed.

*If two SetUp classes are there the class that was written first will be executed first and then after that, the class that is written next to it will execute. (Top to bottom approach).*

**[TearDown]**

***The method decorated with TearDown will be execute after each test are executed.***

*Similarly, in TearDown the order is reversed -- it will follow bottom to top approach.*

**[ExpectedException]**

At times, we may want **to test that a method should throw an exception in a particular scenario**. We can test this using this attribute.

**[Ignore] or [Ignore(“Reason”)]**

To ignore the test case

**[TestCase(2240)]**

Marks a method as parametrized test suit and provides arguments for each test case.

public void PromostandardDataServiceTest(int companyId, TestCaseConfig config)

{ }

**What is mocking?**

Let's think that one application is being develop and many developers are working in this project and each one is assigned to develop a function. Let's think that I am developing a function that will insert one employee information into the DB; if it is not present in the DB then fine and one of my fellow developer is developing the function to check the existence.

And I have completed my function but this guy has not, as he has a little bit of a workload, haha.. Now, as I completed my task, I wanted to test my function but for that I need to depend on the checking function that is still not developed.

So, how I will do that? I need to create mock object that will bypass the checking function. The point to make here is that there are many mocking frameworks to implement the mock object. In this article we will use MOQ as a mocking framework.

Here is our code that we will test using the unit test application.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace TestProjectLibrary

{

public class checkEmployee

{

public virtual Boolean checkEmp()

{

throw new NotImplementedException();

}

}

public class processEmployee

{

public Boolean insertEmployee(checkEmployee objtmp)

{

objtmp.checkEmp();

return true;

}

}

}

Now, see the implementation, the checkEmployee class contains a checkEmp() function that is still not implemented. And we are sending an object of the checkEmployee class to the insertEmployee() function to check whether the employee already exists before it is inserted into the DB.

So, the concept is that since the checkEmployee class is not fully implemented, we will send a mock object of the checkEmployee class as an argument of the insertEmployee() function. Here is sample code of the implementation.

using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using TestProjectLibrary;

**using Moq;**

namespace UnitTest

{

[TestClass]

public class UnitTest

{

[TestMethod]

public void TestMethod2()

{

Mock<checkEmployee> chk = new Mock<checkEmployee>();

chk.Setup(x => x.checkEmp()).Returns(true);

processEmployee obje = new processEmployee();

Assert.AreEqual(obje.insertEmployee(chk.Object), true);

}

}

}

Have a look at the first two lines of TestMethod2(). We are defining a mock object associated with checkCmployee class and in the next line we are setting the mock object.

chk.Setup(x => x.checkEmp()).Returns(true);

The preceding line is a bit interesting. Moq has a Setup() function by which we can set up the mock object. We need to use a lambda expression to point to a specific function. Here we are referring to the checkEmp() function and the Returns parameter value is true.

This means that whenever the unit test application encounters the checkEmp() function it will always return true without executing it's code. So, ultimately, it will not execute at all and the result will be always true.

**Mocking**

[TestCase(100, "Declined")]

[TestCase(549, "Declined")]

[TestCase(550, "Maybe")]

[TestCase(674, "Maybe")]

[TestCase(675, "We look forward to doing business with you!")]

public void MakeCreditDecision\_Always\_ReturnsExpectedResult(int creditScore, string expectedResult)

{

**mockCreditDecisionService = new Mock<ICreditDecisionService>(MockBehavior.Strict);**

**mockCreditDecisionService.Setup(p => p.GetDecision(creditScore)).Returns(expectedResult);**

systemUnderTest = new CreditDecision(mockCreditDecisionService.Object);

var result = systemUnderTest.MakeCreditDecision(creditScore);

Assert.That(result, Is.EqualTo(expectedResult));

mockCreditDecisionService.VerifyAll();

}

We'll cover the MockBehavior.Strict bit in a minute, but let's focus on that second line - this is where we configure the mock for this test. In this case, we're telling the mock, "Hey Mock, if your GetDecision method is invoked with this specific number (creditScore), return this response (expectedResult). If it gets invoked with any other number, fail the test immediately". (that's part of MockBehavior.Strict)

Next up, we execute the MakeCreditDecision method just like we did before, the only remaining step is to ask our Mock instance if all of its expectations were fulfilled using mockCreditDecisionService.VerifyAll()