NUnit is most popular tool for doing Unit Testing. Before start, we need to learn what is Unit Testing and why NUnit is a popular tool for doing unit testing.

**Unit Testing**

Every software is composed of various modules. Each module is composed of various classes. Classes composed of various functions. Function is the smallest unit of code in the application.

When we test individual function behavior without touching any other functions and determine whether it works exactly as per the requirements or not that is called Unit Testing.

Some of the advantages of Unit Testing:

1. Defects found early in development life cycle
2. Reliable Code
3. Maintainable code
4. Faster testing by only single click of action

## NUnit

[NUnit](https://www.nunit.org/index.php) is a unit testing framework for .NET. It is the most used framework for writing unit test cases.

We can write testing code in either C# or VB.NET. It is suggested to write testing code in different assemblies called **Test Assemblies**. These assemblies only contain testing code nothing else. We need to run these test assemblies to check whether all test cases are passed or failed. For that we required Test Runner.

**Test Runners** are UI tool which actually run NUnit test cases and show the result of test cases whether they are passed or failed. We'll learn about test runners in Environment Setup in next post.

NUnit is very easy to use. It only provides some custom attributes and some static Assert classes. With the combination of custom attributes and static classes, we can write unit test cases easily.

Custom attributes provides hint to NUnit test runners that these classes or functions contains unit testing code. Assert classes is used to test the conditions whether system under test (SUT) satisfy a condition or not. If condition is satisfied then test is pass else fail.

Some of the custom attributes are:

* TestFixture
* Setup
* TearDown
* Test
* Category
* Ignore
* TestCase
* Repeat
* MaxTime

There are two steps in configure NUnit project environment:

1. Configure Project with NUnit assemblies
2. Setup TestRunners which show the results of NUnit test cases

**Configure Project with NUnit assemblies**

We always creates separate project when creating project for NUnit. According to naming conventions test project name should be [Project Under Test].[Tests]. For example, if we are testing the project name "CustomerOrderService" then test project name should be "CustomerOrderService.Tests".

1. Visual Studio -> New Project -> Class Library -> Name: CustomerOrderService
2. Right click on solution -> Add New Project -> Nunit Test Project

Now we have two projects in our solution. In CustomerOrderService project, we write code for business logic and in second project CustomerOrderServiceTests we write test cases for CustomerOrderService project.

using NUnit.Framework;

namespace CustomerOrderService.Tests

{

[TestFixture]

public class Class1

{

[Test]

public void Test1()

{

Assert.That(1 == 1);

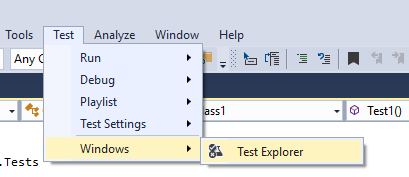
}

}

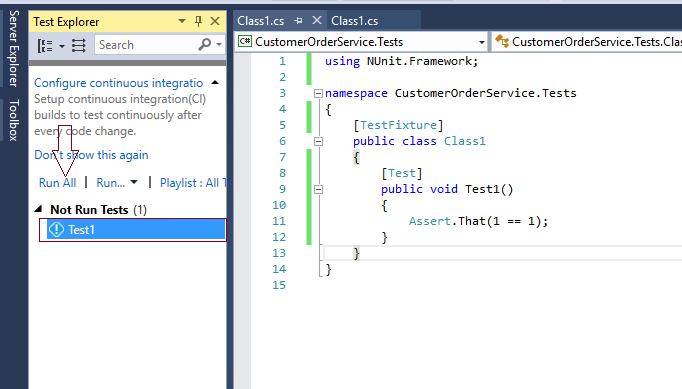
}

Include namespace NUnit.Framework in namespaces section. Write the above code exactly in Class. We'll learn more about TestFixture attribute, Test attribute and Assert class in our next posts. Now Build the solution.

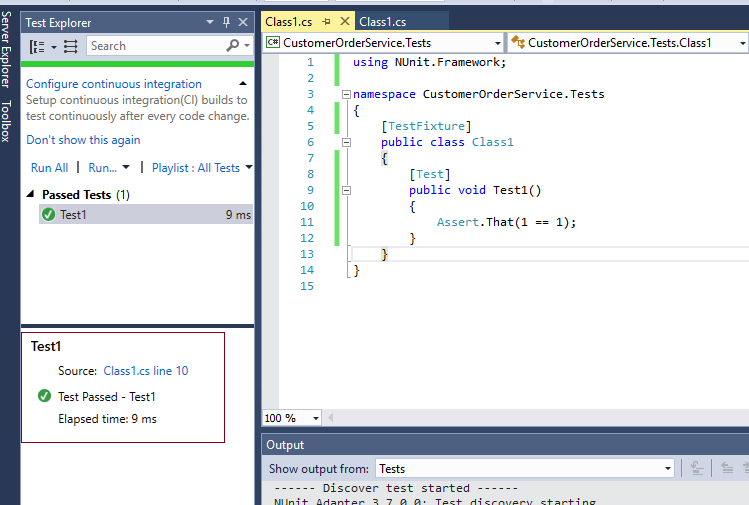
Choose visual studio Test Menu -> Windows -> Test Explorer.



Text Explorer shows Test function in Not Run Tests section. Choose Run All button to execute test cases.



In below of Test Explorer, it will show the result of Test1 test result. In the below screenshot, it is showing the result 'Test Passed - Test1'.



Now our Test project and TestRunner is configured properly. In next posts, we'll learn more about Nunit attributes and classes.

Example:

We have two projects CustomerOrderService project which is a class library and CustomerOrderService.Tests project which is a NUnit test project.

First create two classes Customer and Order and one enum CustomerType in CustomerOrderService project.

namespace CustomerOrderService

{

public enum CustomerType

{

Basic,

Premium,

SpecialCustomer

}

}

namespace CustomerOrderService

{

public class Customer

{

public int CustomerId { get; set; }

public string CustomerName { get; set; }

public CustomerType CustomerType { get; set; }

}

}

namespace CustomerOrderService

{

public class Order

{

public int OrderId { get; set; }

public int ProductId { get; set; }

public int ProductQuantity { get; set; }

public decimal Amount { get; set; }

}

}

CustomerType enum is used for differentiate customers. Some customers are Basic customers and some are Premium or Special Customer.

In the next class CustomerOrderService, we will write our Business Logic to give some discount to Premium or Special Customer types. In the NUnit project, we will write unit test cases for validating the discount logic.

namespace CustomerOrderService

{

public class CustomerOrderService

{

public void ApplyDiscount(Customer customer, Order order)

{

if (customer.CustomerType == CustomerType.Premium)

{

order.Amount = order.Amount - ((order.Amount \* 10) / 100);

}

else if(customer.CustomerType == CustomerType.SpecialCustomer)

{

order.Amount = order.Amount - ((order.Amount \* 20) / 100);

}

}

}

}

In the above method, we are giving 10% discount to Premium customers and 20% discount to Special Customers. We are not giving any discount to Basic customers.

Now, come under CustomerOrderService.Tests project. Create a new class "CustomerOrderServiceTests". Add namespace "using NUnit.Framework" in the namespaces section.

Add [TestFixture] attribute to CustomerOrderTests class. [TestFixture] attribute marks the class that this class contains test methods. Only classes with [TestFixture] attribute can add test case methods.

using NUnit.Framework;

namespace CustomerOrderService.Tests

{

[TestFixture]

public class CustomerOrderServiceTests

{

}

}

Add a new method name "When\_PremiumCustomer\_Expect\_10PercentDiscount" and add [TestCase] attribute.

using NUnit.Framework;

namespace CustomerOrderService.Tests

{

[TestFixture]

public class CustomerOrderServiceTests

{

[TestCase]

public void When\_PremiumCustomer\_Expect\_10PercentDiscount()

{

}

}

}

All test cases method are public and void return because in the test cases we should not return any value.

We should write NUnit test method name in special naming convention. Below is the naming convention I am using in the above test case name.

When\_StateUnderTest\_Expect\_ExpectedBehavior

First part starts with fixed 'When'. Second part specify the state which we want to test. Third part is also fixed 'Expect' and forth part specify the expected behavior of method under test.

Now, we write our first NUnit test case example method code. A test case body is divided into three sections "AAA".

"AAA" denotes the Arrange, Act, and Assert.

Arrange: In Arrange section, we will initialize everything which we are required to run the test case. It includes any dependencies and data needed.

Act: In Act section, we called the business logic method which behavior we want to test.

Assert: Specify the criteria for passing the test case. If these criteria passed, that means test case is passed else failed.

First NUnit Test Case Example

[TestCase]

public void When\_PremiumCustomer\_Expect\_10PercentDiscount()

{

//Arrange

Customer premiumCustomer = new Customer

{

CustomerId = 1,

CustomerName = "George",

CustomerType = CustomerType.Premium

};

Order order = new Order

{

OrderId = 1,

ProductId = 212,

ProductQuantity = 1,

Amount = 150

};

CustomerOrderService customerOrderService = new CustomerOrderService();

//Act

customerOrderService.ApplyDiscount(premiumCustomer, order);

//Assert

Assert.AreEqual(order.Amount, 135);

}

In Arrange section, we initialize two variables premiumCustomer and order. In the next line, we create an instance of our CustomerOrderService class.

In Act section, we called actual method of CustomerOrderService class with variables initialized in Arrange section.

In Assert section, we check our expected response. Is amount equals to 135 or not?

Assert Class

Assert is NUnit framework class which has static methods to verify the expected behavior. In the above example, AreEqual method verifies that two values are equal or not.

If the Assert condition is passed then the NUnit test case is passed else failed.

Now we will run our first NUnit test case.

Steps:

Build the full Solution

Click on Test Menu -> Windows -> Test Explorer

Click on Run All link.

'Run All' link search entire solution for methods which has [TestCase] attribute and run all those methods.

After Run All, if our test case method name color change to Green, that means our test case is passed, and if it turns into red color that means our test is failed.

Below is our test result.

NUnit test case example

Now if I change the expected result to 130 and again build the project and click on "Run All" link. Then our test case fails.

[TestCase]

public void When\_PremiumCustomer\_Expect\_10PercentDiscount()

{

//Arrange

Customer premiumCustomer = new Customer

{

CustomerId = 1,

CustomerName = "George",

CustomerType = CustomerType.Premium

};

Order order = new Order

{

OrderId = 1,

ProductId = 212,

ProductQuantity = 1,

Amount = 150

};

CustomerOrderService customerOrderService = new CustomerOrderService();

//Act

customerOrderService.ApplyDiscount(premiumCustomer, order);

//Assert

Assert.AreEqual(order.Amount, 130);

}

NUnit TestFixture attribute is a class level attribute and it indicates that this class contains NUnit Test Methods.

Below are the topics we covered in this tutorial:

TestFixture Example and Usage

Parameterized TestFixtures

TestFixture Inheritance

Generic TestFixture

TestFixture Restrictions

NUnit TestFixture Example and Usage

Below is the example usage of TestFixture NUnit attribute.

using NUnit.Framework;

namespace CustomerOrderService.Tests

{

[TestFixture]

public class CustomerOrderServiceTests

{

}

}

Remember: TestFixture can only be placed on class not on methods. You can learn more about writing test case methods here.

Parameterized / Arguments TestFixtures

Sometimes our NUnit class needs arguments. We can pass arguments to TestFixture class through constructors. We can pass arguments in TestFixture attribute like shown below:

[TestFixture(CustomerType.Basic)]

public class CustomerOrderServiceTests

{

private CustomerType customerType;

public CustomerOrderServiceTests(CustomerType customerType)

{

this.customerType = customerType;

}

}

In the above example, our class needs CustomerType as parameter so we create a constructor and specify the CustomerType as parameter. We provide value to parameters in TestFixture as CustomerType.Basic.

If the NUnit framework did not found any matching TestFixture attribute for example, if we did not pass CustomerType.Basic as argument then NUnit framework give below error.

Message: OneTimeSetup: No suitable constructor was found

We can create multiple constructors and pass multiple parameters through TestFixture. Below is the example:

[TestFixture(CustomerType.Premium, 100.00)]

[TestFixture(CustomerType.Basic)]

public class CustomerOrderServiceTests

{

private CustomerType customerType;

private double minOrder;

public CustomerOrderServiceTests(CustomerType customerType, double minOrder)

{

this.customerType = customerType;

this.minOrder = minOrder;

}

public CustomerOrderServiceTests(CustomerType customerType) : this(customerType, 0)

{

}

[TestCase]

public void TestMethod()

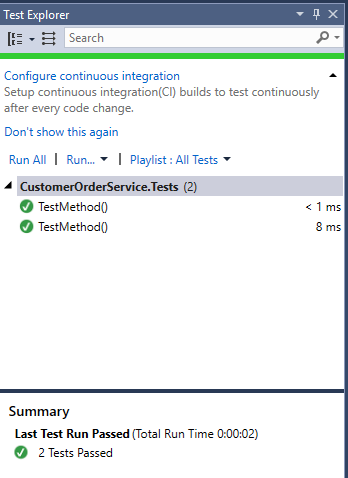
{

Assert.IsTrue((customerType == CustomerType.Basic && minOrder == 0 || customerType == CustomerType.Premium && minOrder > 0));

}

}

When we create multiple constructors NUnit will create separate objects using each constructor. For example in the above example, NUnit will create two separate test methods using each constructor parameters.



NUnit TestFixture Inheritance

A TestFixture attribute supports inheritance that means we can apply TestFixture attribute on base class and inherit from derived Test Classes. A base class can be an Abstract class.

Abstract Fixture Pattern

This pattern is used when we have to validating the logic of base class and make sure that derived class does not violate the base class implementation.

For example, we have a base Employee class and two inherit Manager and DeliveryManager classes. We have some validations in Employee class and we write some test cases for Employee class and we need to make sure these validations must verify by the derived classes.

public abstract class Employee

{

public string Name { get; set; }

public bool ContainsIllegalChars()

{

if (this.Name.Contains("$"))

{

return true;

}

return false;

}

}

public class Manager : Employee {}

public class DeliveryManager : Employee {}

Test Cases

using NUnit.Framework;

namespace EmployeeService.Tests

{

[TestFixture]

public class EmployeeTests

{

public virtual Employee CreateEmployee()

{

return new Employee();

}

[TestCase]

public void When\_NameContainsIllegalChars\_Expect\_ContainsIllegalChars\_ReturnsTrue()

{

Employee employee = CreateEmployee();

employee.Name = "Da$ya";

var result = employee.ContainsIllegalChars();

Assert.IsTrue(result);

}

}

public class ManagerTests : EmployeeTests

{

public override Employee CreateEmployee()

{

return new Manager();

}

}

public class VicePresidentTests : EmployeeTests

{

public override Employee CreateEmployee()

{

return new DeliveryManager();

}

}

}

We have created a EmployeeTests class in C#. Created a virtual method CreateEmployee which will create a new instance of Employee class and can override by derived classes. Write a test method which test the ContainsIllegalChars method of Emplyee class.

Create two other Tests Classes ManagerTests and VicePresidentTests which inherits from EmployeeTests classes. Note we have not used TestFixture attribute on derived classes. It is automatically derived by inherited classes.

We override the CreateEmployee method and return derived classes of Employee class. NUnit framework will create three different test methods for all three classes. Below is the screenshot:

TestFixture Inheritance

Generic TestFixture

In addition to parameters, we can also give indication to NUnit which data types are passing into the TestFixture attribute. Below is the example.

[TestFixture(CustomerType.Premium, 100.00, TypeArgs = new Type[] { typeof(CustomerType), typeof(double) })]

public class CustomerOrderServiceTests<T1, T2>

{

private T1 customerType;

private T2 minOrder;

public CustomerOrderServiceTests(T1 customerType, T2 minOrder)

{

this.customerType = customerType;

this.minOrder = minOrder;

}

[TestCase]

public void TestMethod()

{

Assert.That(customerType, Is.TypeOf<CustomerType>());

Assert.That(minOrder, Is.TypeOf<double>());

}

}

In the above example, we specify the parameters and then using TypeArgs specify the typeof arguments. In the TestMethod, we are passing correct Type as generic arguments or not.

TestFixture Restrictions

Below are some restrictions/points about TestFixture attribute.

It can only place on class.

If no arguments is provided in TestFixture attribute then class must have default constructor.

If arguments is provided in TestFixture attribute then class must have matching constructor.

We can place multiple TestFixture attributes on a single class.

TestFixture attribute can be inherited

We can provide generic arguments to TestFixture class.

We can apply TestFixture attribute on abstract class.

NUnit TestCase Arguments / Parameters

TestCase arguments are used when we have to use same test case with different data.

For example, in the above case, we fixed the age of employee as 60. For different ages we have to write different test cases. But by using the TestCase parameters we can use same test method for different ages.

[TestCase(60)]

[TestCase(80)]

[TestCase(90)]

public void When\_AgeGreaterAndEqualTo60\_Expects\_IsSeniorCitizeAsTrue(int age)

{

Employee emp = new Employee();

emp.Age = age;

bool result = emp.IsSeniorCitizen();

Assert.That(result == true);

}

In this example, we have use three TestCase attributes on same method with different parameters. NUnit framework will create three different test cases using these three parameters.

NUnit TestCase Arguments

NUnit TestCase ExpectedResult

In the above example, we have fixed the result to true that means we can only check the above test case with positive parameters. But by using ExpectedResult property, we can also specify different results for different parameters. Below is the example:

[TestCase(29, ExpectedResult = false)]

[TestCase(0, ExpectedResult = false)]

[TestCase(60, ExpectedResult = true)]

[TestCase(80, ExpectedResult = true)]

[TestCase(90, ExpectedResult = true)]

public bool When\_AgeGreaterAndEqualTo60\_Expects\_IsSeniorCitizeAsTrue(int age)

{

Employee emp = new Employee();

emp.Age = age;

bool result = emp.IsSeniorCitizen();

return result;

}

In this example, we change the return type of method to bool data type and also change the last line of test case method to return statement. In the parameters, we specify the ExpectedResult as bool data type matching return type of test method.

Author Property

We can specify author name in the test method who has written the test case. Below is the example:

[TestCase(Author = "Michael")]

public void When\_AgeGreaterAndEqualTo60\_Expects\_IsSeniorCitizeAsTrue()

{

...

}

[TestCase(Author = "George")]

public void When\_AgeGreaterAndEqualTo100\_Expects\_IsSeniorCitizeAsTrue()

{

...

}

For executing tests, right click on any test method and choose GroupBy -> Traits.

NUnit GroupBy Traits

By choosing this option, test explorer categorized test methods according to different properties of TestCase. Below is the example of Author property group by:

NUnit Trait Example

TestName property

TestName property is used when we have to use different name than the specified test method name. Below is the example:

[TestCase(TestName = "EmployeeAgeGreaterAndEqualTo60\_Expects\_IsSCitizenAsTrue")]

public void When\_AgeGreaterAndEqualTo60\_Expects\_IsSeniorCitizeAsTrue()

{

...

}

[TestCase(TestName = "EmployeeAgeGreaterThan100\_Expects\_IsSCitizenAsTrue")]

public void When\_AgeGreaterThan100\_Expects\_IsSeniorCitizeAsTrue()

{

...

}

Ignore TestCase

Sometimes we need to ignore our test case reason may be code is not yet complete. So we can use Ignore property to mark test case as ignore. This still show test method in Test Explorer but test explorer will not execute it.

[TestCase(Ignore = "Code is not complete yet.")]

public void When\_AgeGreaterAndEqualTo60\_Expects\_IsSeniorCitizeAsTrue()

{

....

} NUnit TestCase Array

Below is the example of passing array to a test method.

[TestCase(new int[] { 2, 4, 6 })]

public void When\_AllNumberAreEven\_Expects\_AreAllNumbersEvenAsTrue(int[] numbers)

{

Number number = new Number();

bool result = number.AreAllNumbersEven(numbers);

Assert.That(result == true);

}

There is one restriction on array type. Array type must be a constant expression. Array types are limited to below types:

bool

byte

char

short

int

long

float

double

Enum

object

For passing other data types like string, we can use either object type or can use NUnit TestCaseSource. Below is the example of passing strings as object array.

[TestCase(new object[] { "1", "2", "3" })]

public void When\_AllNumberAreEven\_Expects\_AreAllNumbersEvenAsTrue(object[] numbers)

{

....

}

TestCaseSource Attribute

TestCaseSource indicates that pass source parameter can be used as a parameter. Below is the syntax of TestCaseSource.

[TestCaseSource(Type sourceType, string sourceName)]

In the source type, we can define the parameter type. In sourceName we provide the name of the data source. In source name, we can provide below names:

Static Field / Property / Method Name

Property Name

Field Name

Custom Type implements IEnumerable

Below is the example of passing string array using Custom Type in TestCaseSource attribute.

public class StringArrayTestDataSource : IEnumerable

{

public IEnumerator GetEnumerator()

{

yield return new string[] { "2", "4", "6" };

yield return new string[] { "3", "4", "5" };

yield return new string[] { "6", "8", "10" };

}

}

[TestFixture]

public class EmployeeTests

{

[TestCaseSource(typeof(StringArrayTestDataSource))]

public void When\_StringArrayAreEvenNumbers\_Expects\_IsStringArrayOfEvenNumbersAsTrue(string[] numbers)

{

Number number = new Number();

bool result = number.IsStringArrayOfEvenNumbers(numbers);

Assert.That(result == true);

}

}

In the above example, we have create a new class StringArrayTestDataSource that implements interface IEnumerable. In the GetEnumerator method, we returns our string arrays. We have applied TestCaseSource attribute and pass typeof(StringArrayTestDataSource) as parameter. In the test method parameter we have used string array parameter.

NUnit TestRunner will pick a string array from GetEnumerator method and pass one by one into the test method.

NUnit TestCase Execution Order

Sometimes we need to execute test methods in a particular order. These test method are dependent on each other. For that, NUnit provides the Order attribute. Below is the example.

[TestCase]

[Order(1)]

public void When\_AgeGreaterAndEqualTo60\_Expects\_IsSeniorCitizeAsTrue()

{

....

}

[TestCase]

[Order(2)]

public void When\_AgeLessThan60\_Expects\_IsSeniorCitizeAsFalse()

{

....

}