**Objectives**

* Explain the meaning of Unit testing and its difference on comparison with Functional testing
  + Smallest unit to test mocking dependencies
* List various types of testing
  + Unit testing, Functional testing, Automated testing, Performance testing
* Understand the benefit of automated testing
* Explain what is loosly coupled & testable design
  + Write code that is NOT dependent on the class for data.
* Write wer first testing program to validate a calculator addition operation
  + TestFixture, Test
* Understand the need of [SetUp], [TearDown] & [Ignore] attributes.
* Explain the benefit of writing parameterised test cases.
  + TestCase

**TestFixture & Test**

Please download the application available [here](https://cognizantonline.sharepoint.com/:u:/r/sites/GTP-Solutions/Gencsharepath/Shared%20Documents/Internship2020/FSE/DotNet/02%20-%20NUnit,%20C%23%204.5,%20ASP.Net%20Core/Handson/CalcLibrary.zip?csf=1&e=aLxB66). This will be used to write Unit test cases  
  
Follow the steps listed below to write the NUnit test cases for the application.

* Create a Unit test project(.Net Framework) in the solution provided.
* Add the CalcLibrary project as reference
* Create a class “CalculatorTests” to write all the test cases for the methods in the solution
* Use the ‘TestFixture’, ‘SetUp’ and ‘TearDown’ attributes, to declare, initialize and cleanup activities respectively
* Create a Test method to check the addition functionality
* Use the ‘TestCase’ attribute to send the inputs and the expected result
* Use Assert.That to check the actual and expected result match

1. Unit testing is a term to describe testing individual functions or components of a software application in isolation. It validates that a given piece of code is doing the right thing. The intent is to test the smallest unit of logic—typically methods or classes—without touching external systems. This is generally accomplished by making use of frameworks such as NUnit, MSTest, or xUnit.

In contrast, functional testing is aimed at ensuring that the software behaves as specified. It is done by testing entire features or modules, usually involving integration with systems outside the software, such as databases or APIs. Unit testing is developer-oriented and performed early in the development process, whereas functional testing is more comprehensive and tends to be performed by testers or QA teams at a later stage in the process.

2. There are a number of different testing activities, each of which addresses a particular area of software quality. Unit testing verifies individual components of code. Functional testing ensures business requirements and feature behavior. Automated testing is the use of scripts and tools to run tests without human intervention so that tests can be run repeatedly and regularly. Performance testing measures how the application performs when under load, testing responsiveness, stability, and scalability. Collectively, these types of testing lead to strong, reliable software.

3. Automated testing has huge benefits in the development process. Automated tests can be run repeatedly, fast, and reliably with ease and can easily pinpoint regressions or errors early in the development process. Automated tests are faster than manual tests, particularly when the application becomes large and complex. They also increase confidence when refactoring or adding new features, guaranteeing that the existing functionality does not break.

4. A loosely coupled design is where various components of a system are not tightly connected and can be tested or changed independently without impacting one another. This is done with interfaces, dependency injection, and abstraction. A testable design adheres to the above principle to enable individual components to be tested in isolation. For instance, rather than having a class create its dependencies explicitly, the dependencies should be passed in so that mock or fake implementations can be utilized during unit testing. This produces more maintainable, scalable, and healthy code.

5. In order to make code testable and loosely coupled, it should never depend on hard-coded data or direct object references. Data should be passed through parameters or interfaces instead. For example, instead of creating a database connection within a class, a class should take an interface of the connection into its constructor. This makes unit tests able to pass in a mock implementation of the interface, allowing the logic to be tested without going through real data sources.

6. With NUnit, we can create a basic unit test to check the accuracy of an addition method of a calculator. This is done by attributing the test class with `[TestFixture]` and the test method with `[Test]`. In the test method, the `Addition()` method is invoked with test input values, and the output is verified using assertions such as `Assert.That()` to confirm it is equal to the expected output. This is the starting point for unit testing.

7. In NUnit, the `[SetUp]` attribute is employed for setting the test environment prior to every test execution, e.g., object initialization or setting preconditions. The `[TearDown]` attribute is employed for cleanup after every test, e.g., releasing resources or resetting states. The `[Ignore]` attribute is invoked to bypass a test temporarily—this is helpful when a test is being developed or shouldn't run because of known problems. These lifecycle methods ensure that tests are decoupled and stable.

8. Parameterized tests with `[TestCase]` in NUnit enable us to execute the same test logic with varying input and output values. This prevents code duplication and enhances test coverage. For instance, rather than creating three tests for the addition method, we can create one test method with multiple `[TestCase]` attributes. This renders tests efficient, neat, and easy to maintain.

Code:  
using NUnit.Framework;

using CalcLibrary;

using System;

namespace TestProject1

{

[TestFixture]

public class SimpleCalculatorTests

{

private SimpleCalculator? calculator;

[SetUp]

public void Init()

{

calculator = new SimpleCalculator();

}

[TearDown]

public void Cleanup()

{

calculator = null;

}

[Test]

[TestCase(10, 5, 15)]

[TestCase(0, 0, 0)]

[TestCase(-2, -3, -5)]

public void Addition\_ShouldReturnCorrectResult(double a, double b, double expected)

{

double result = calculator!.Addition(a, b);

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

}

[Test]

[TestCase(10, 5, 5)]

[TestCase(0, 0, 0)]

[TestCase(-2, -3, 1)]

public void Subtraction\_ShouldReturnCorrectResult(double a, double b, double expected)

{

double result = calculator!.Subtraction(a, b);

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

}

[Test]

[TestCase(2, 3, 6)]

[TestCase(0, 5, 0)]

[TestCase(-2, -3, 6)]

public void Multiplication\_ShouldReturnCorrectResult(double a, double b, double expected)

{

double result = calculator!.Multiplication(a, b);

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

}

[Test]

[TestCase(10, 2, 5)]

[TestCase(5, 2, 2.5)]

public void Division\_ShouldReturnCorrectResult(double a, double b, double expected)

{

double result = calculator!.Division(a, b);

Assert.That(result, Is.EqualTo(expected).Within(0.0001));

}

[Test]

public void Division\_ByZero\_ShouldThrowException()

{

Assert.Throws<ArgumentException>(() => calculator!.Division(10, 0));

}

[Test]

public void AllClear\_ShouldResetResultToZero()

{

calculator!.Addition(5, 5);

calculator.AllClear();

Assert.That(calculator.GetResult, Is.EqualTo(0));

}

}

}

