1. **NUnit Hands-on:**

Objectives

· Explain the meaning of Unit testing and its difference on comparison with Functional testing

o Smallest unit to test mocking dependencies

· List various types of testing

o Unit testing, Functional testing, Automated testing, Performance testing

· Understand the benefit of automated testing

· Explain what is loosly coupled & testable design

o Write code that is NOT dependent on the class for data.

· Write your first testing program to validate a calculator addition operation

o TestFixture, Test

· Understand the need of [SetUp], [TearDown] & [Ignore] attributes.

· Explain the benefit of writing parameterised test cases.

o TestCase

TestFixture & Test Please download the application available here. 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.

**SOLUTION:**

## ****Objectives****

## 1. Explain the Meaning of Unit Testing & Its Difference from Functional Testing

**Unit Testing** is the process of testing individual units or components (e.g., methods, functions) in isolation to ensure they work as expected.

It focuses on the smallest part of an application, like a method

Dependencies are mocked/stubbed

**Functional Testing** tests the system as a whole to ensure the functionality behaves as expected.

| **Unit Testing** | **Functional Testing** |
| --- | --- |
| Tests single components/methods | Tests whole modules or features |
| Done by developers | Done by QA/testers |
| Fast, isolated | Slower, broader |
| Uses mocking | Uses real integrations |

### 2. Smallest Unit to Test – Mocking Dependencies

A **unit** is the smallest testable piece of code (e.g., a method).

When testing, we **mock** external dependencies (like database calls, services) so we can test just the unit in isolation.

### 3. List of Testing Types

| **Type** | **Description** |
| --- | --- |
| **Unit Testing** | Tests individual methods or classes |
| **Functional Testing** | Tests full features from end-user point of view |
| **Automated Testing** | Uses tools to automatically run test cases |
| **Performance Testing** | Checks system speed and scalability under load |

### Benefits of Automated Testing

### Fast and repeatable

### Catches bugs early

### Encourages clean, testable code

### Enables Continuous Integration / Continuous Deployment (CI/CD)

### 5. Explain Loosely Coupled & Testable Design

**Loosely Coupled Code**: Classes and methods are not tightly connected. You can replace or mock dependencies easily.

It promotes:

Easier unit testing

Reusability

Maintainability

### Example of Loosely Coupled Design:

A method that doesn’t directly depend on another class — it uses an interface or accepts data as parameters.

Create Calculator Logic (MyCalcLibrary):

namespace MyCalcLibrary

{

public class MyCalculator

{

public int Add(int a, int b)

{

return a + b;

}

}

}

### Create Unit Test Project (MyCalcLibraryTests)

Installed packages:

NUnit

NUnit3TestAdapter

Microsoft.NET.Test.Sdk

### NUnit Test Class

**File:** MyCalculatorTests.cs

using NUnit.Framework;

using MyCalcLibrary;

namespace MyCalcLibraryTests

{

[TestFixture]

public class CalculatorTests

{

private Calculator calculator;

[SetUp]

public void SetUp()

{

calculator = new Calculator();

}

[TearDown]

public void TearDown()

{

calculator = null;

}

[Test]

[TestCase(1, 2, 3)]

[TestCase(10, 5, 15)]

[TestCase(-1, 1, 0)]

[TestCase(100, 200, 300)]

public void Add\_WhenCalled\_ReturnsCorrectSum(int a, int b, int expected)

{

var result = calculator.Add(a, b);

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

}

[Test]

[Ignore("This test is skipped")]

public void ThisWillBeIgnored()

{

Assert.Fail("This should be skipped.");

}

}

}

## NUnit Attributes Explained

| **Attribute** | **Purpose** |
| --- | --- |
| [TestFixture] | Declares a test class |
|  |  |
| [SetUp] | Runs before each test (like constructor) |
| [TearDown] | Runs after each test (like destructor) |
| [Ignore] | Skips the test |
| [TestCase] | Runs the test with different sets of inputs |

**OUTPUT:**

