**WEEK – 2 NUnit HandsOn**

**1. NUnit HandsOn**

**Given :**

**Objectives**

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

**What is Unit Testing?**

Unit Testing is a software testing technique where individual components or functions (called units) of the code are tested in isolation to ensure they work as expected.

* It focuses on a single method or class (smallest unit).
* Typically written by developers during development.
* Uses test frameworks like NUnit.
* Ensures that internal logic (e.g., calculations, return values) is correct.

Example: Testing if a method Add(2, 3) returns 5.

**What is Functional Testing?**

Functional Testing checks if the overall system or a specific feature behaves correctly from the user’s perspective.

* Focuses on complete features or workflows (like submitting a form or logging in).
* Usually performed by testers or QA teams.
* Uses tools like Selenium, Postman, etc.
* Verifies whether the application meets business requirements.

Example: Testing if clicking “Submit” on a login page logs in the user correctly.

**Unit Testing & Mocking Dependencies**

* When a method depends on external systems (like a database or file), it can be mocked using tools like Moq.
* This makes tests faster, isolated, and predictable.

Example: Instead of calling a real database, we mock a repository to return fake data for testing logic.

* + **Smallest unit to test mocking dependencies**

When writing unit tests, the goal is to test the smallest piece of functionality—usually a single method or class—in isolation.

However, some methods depend on external components like:

* A database
* A file system
* An email sender
* A web service

You don’t want to actually call those during unit tests.  
Instead, you mock those dependencies using a mocking framework like Moq.

* **List various types of testing**
  + **Unit testing, Functional testing, Automated testing, Performance testing**

**1. Unit Testing**

* Definition: Testing the smallest individual unit of code, such as a method or function, in isolation.
* Purpose: Ensures that each part of the code behaves correctly.
* Tools: NUnit, xUnit, Moq
* Who does it? Developers

Example: Testing if Add(2, 3) returns 5.

**2. Functional Testing**

* Definition: Testing a system or feature to ensure it behaves as expected based on business requirements.
* Purpose: Validates whether the application delivers the required functionality.
* Tools: Selenium, Postman, QTP
* Who does it? QA/Testers

Example: Testing the login process from the UI.

**3. Automated Testing**

* Definition: Testing that is performed automatically by scripts or tools rather than manually.
* Purpose: Speeds up testing and ensures repeatability.
* Tools: NUnit, Selenium, JUnit, Appium
* Who does it? Developers + Testers

Example: Automatically running 50 test cases every time code changes.

**4. Performance Testing**

* Definition: Testing how the system performs under load, stress, or high traffic.
* Purpose: Ensures the system is responsive and stable under different conditions.
* Tools: JMeter, LoadRunner, Apache Benchmark
* Who does it? QA/Performance Testers

Example: Testing how a website handles 1,000 users logging in at once.

* **Understand the benefit of automated testing**

**1. Speed and Efficiency**

* Automated tests run much faster than manual testing.
* A large number of tests can be executed in minutes.

Example: Run hundreds of test cases every time code changes — instantly.

**2. Improved Accuracy**

* Manual testing is prone to human errors.
* Automated tests perform exact, repeatable checks every time.

**3. Reusability**

* Once written, automated test scripts can be reused across projects, versions, and builds.

**4. Continuous Integration Support**

* Automated testing integrates with tools like GitHub Actions, Azure DevOps, or Jenkins for CI/CD pipelines.
* Every code change is automatically tested, ensuring quick feedback.

**5. Cost-Effective in the Long Run**

* While writing automated tests requires initial effort, it saves time and cost over repeated manual testing.

**6. Better Test Coverage**

* Automated testing makes it feasible to test more scenarios (including edge cases), increasing code quality and coverage.

**7. Fast Feedback for Developers**

* Developers can catch bugs early, immediately after writing code, instead of waiting for QA.
* **Explain what is loosly coupled & testable design**
  + **Write code that is NOT dependent on the class for data.**

**What is Loosely Coupled Design?**

A loosely coupled design is when classes or components in your code are not tightly dependent on the internal details of other classes.

* Each class knows only what it needs to know — not the full implementation of others.
* This makes code easier to test, extend, and maintain.

Loose coupling = more flexibility, testability  
Tight coupling = hard to test, rigid design

**Why Is It Important for Testing?**

When classes are loosely coupled, you can replace dependencies with mocks (using Moq or similar libraries) during unit testing.

This helps in:

* Testing just one class at a time (unit)
* Avoiding reliance on real databases, files, etc.
* **Write your first testing program to validate a calculator addition operation**
  + **TestFixture, Test**

**1. Calculator Logic :**

namespace CalcLibrary

{

public class Calc

{

public int Add(int a, int b)

{

return a + b;

}

}

}

**2. NUnit Test Code :**

using NUnit.Framework;

using CalcLibrary;

namespace CalcTests

{

[TestFixture] // Marks this class as a test suite

public class CalcTests

{

private Calc calc;

[SetUp]

public void Setup()

{

calc = new Calc(); // Create object before each test

}

[Test] // Basic test to check addition

public void Test\_Addition\_ReturnsCorrectResult()

{

int result = calc.Add(2, 3);

Assert.That(result, Is.EqualTo(5));

}

}

}

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

**1. [SetUp] – Run Before Every Test**

Purpose:

* Prepares a fresh environment before each test case runs.
* Often used to initialize objects, open database connections, etc.

**2. [TearDown] – Run After Every Test**

Purpose:

* Used to clean up after each test.
* Helps release resources, close files/connections, or reset data.

**3. [Ignore] – Skip a Test**

Purpose:

* Temporarily disable a test case.
* Useful when the test is incomplete, failing due to an external reason, or not relevant for the current run.
* **Explain the benefit of writing parameterised test cases.**
  + **TestCase**

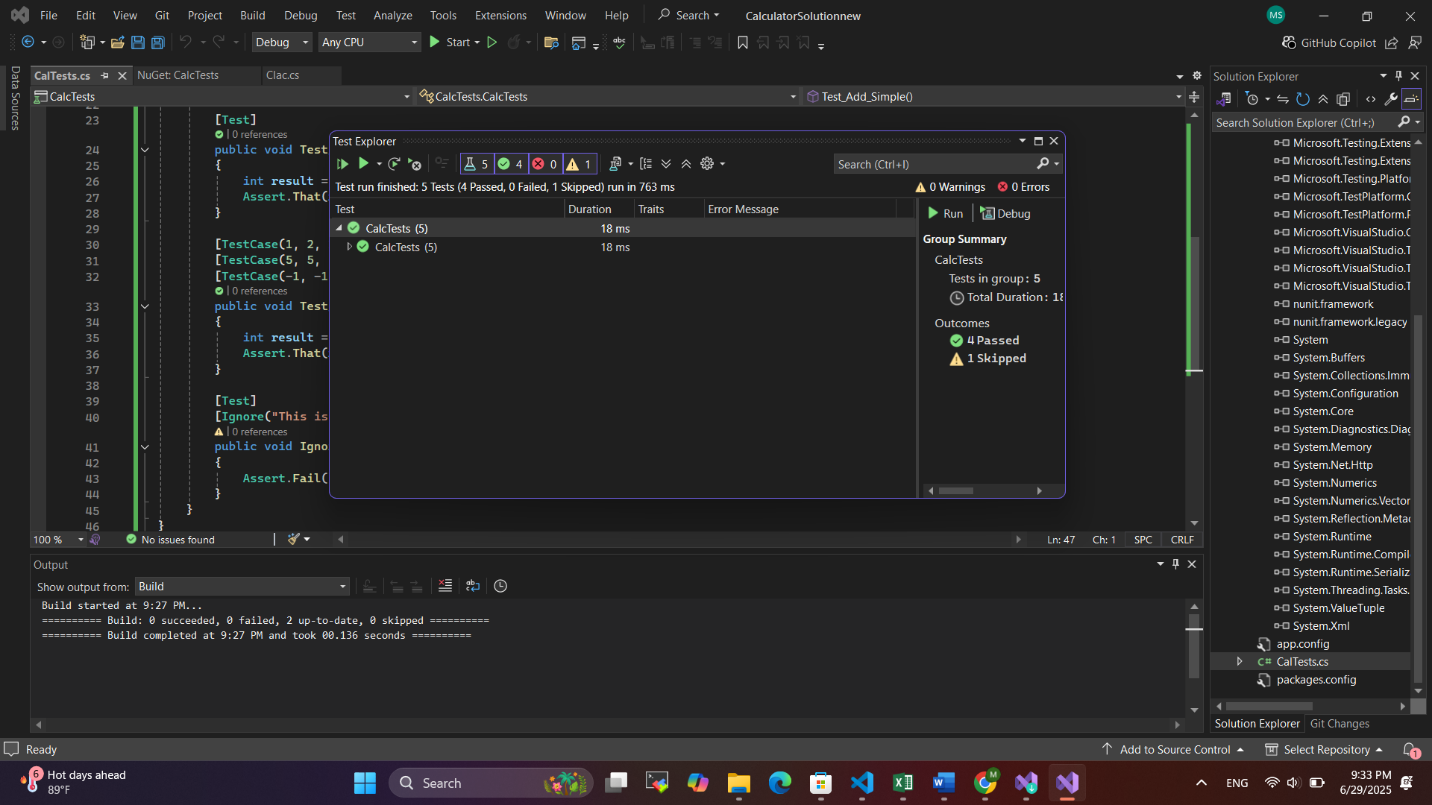
Using [TestCase] in NUnit allows you to write parameterised test cases, which means a single test method can run multiple times with different sets of input and expected output values. This avoids code duplication and makes the test code cleaner and easier to maintain. Instead of writing separate test methods for each input combination, you define multiple [TestCase] attributes above one method. This improves test coverage, saves time, and helps in quickly validating a method against various scenarios, including edge cases. It also makes test results more organized in the test explorer, showing each case separately.

**TestFixture & Test**

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

**OUTPUT :**

****