Hands-On Exercise: Unit Testing with NUnit in C#

Submitted by- Richa Goyal (6365275)

1. Objective Overview

This hands-on exercise is designed to introduce and implement unit testing using the NUnit framework in C#. Through this we:

- Understand what Unit testing is and how it differs from Functional testing.
- Learn the importance of Automated testing.
- Write a unit test to validate a calculator's addition method.
- Explore key NUnit attributes: [TestFixture], [SetUp], [TearDown], [Test], [TestCase], and [Ignore].
- Practice writing testable and loosely coupled code.

2. Key Concepts

Unit Testing vs Functional Testing

- **Unit Testing**: Testing the smallest parts (units) of an application (e.g., a single method). It often involves mocking dependencies.
- **Functional Testing**: Validates that the complete functionality of the application behaves as expected (end-to-end).

Types of Testing

- Unit Testing
- Functional Testing
- Automated Testing
- Performance Testing

Benefits of Automated Testing

- Saves time on repeated manual testing.
- Improves accuracy and consistency.
- Enables early bug detection.
- Facilitates continuous integration.

Loosely Coupled & Testable Design

- Code should not tightly depend on specific implementations.
- Use interfaces or abstractions (like IMathLibrary) to make testing easier.
- Promotes reusability, maintainability, and testing flexibility.

3. Writing Unit Tests with NUnit

Step-by-Step Implementation in VS Code

Step 1: Set Up the Environment

- Make sure .NET SDK is installed.
- Open the solution in VS Code.

Step 2: Create a Unit Test Project

```
dotnet new nunit -n CalcLibraryTests
dotnet sln add CalcLibraryTests/CalcLibraryTests.csproj
```

Step 3: Add Reference to Main Project

```
dotnet add CalcLibraryTests reference CalcLibrary/CalcLibrary.csproj
```

Step 4: Write Test Class-

```
using NUnit.Framework;
using CalcLibrary;

namespace CalcLibraryTests
{
    [TestFixture]
    public class CalculatorTests
    {
        SimpleCalculator? calculator;

        [SetUp]
        public void Init()
        {
            calculator = new SimpleCalculator();
        }

        [TearDown]
        public void Cleanup()
        {
            calculator = null;
        }

        [TestCase(2.0, 3.0, 5.0)]
        [TestCase(-1.5, -1.5, -3.0)]
        [TestCase(0.0, 0.0, 0.0)]
        public void TestAddition(double a, double b, double expected)
        {
            double result = calculator!.Addition(a, b);
            Assert.That(result, Is.EqualTo(expected).Within(0.001));
        }

        [Test, Ignore("Demo of ignored test")]
        public void IgnoredTest()
        {
            Assert.That(1 + 1, Is.EqualTo(3));
        }
    }
}
```

Step 5: Run the Tests

```
dotnet test
```

4. NUnit Attributes Explained

Attribute	Purpose
[TestFixture]	Declares the class as a container for NUnit tests.
[SetUp]	Runs before each test to initialize objects or state.
[TearDown]	Runs after each test to clean up resources.
[Test]	Marks a method as a test case.
[TestCase]	Allows multiple inputs to be tested in a single method.
[Ignore]	Temporarily skips a test.

5. Benefits of Parameterized Tests (``)

- Eliminates code duplication.
- Makes test cases cleaner and easier to understand.
- Allows testing multiple input-output combinations in a single method.

6. Conclusion

This exercise covered:

- Setting up NUnit in a .NET project using VS Code.
- Writing unit tests using best practices.
- Understanding how unit testing improves software quality.

You now have hands-on experience with writing and running NUnit test cases in C#. This forms the foundation for robust and maintainable automated testing.