Unit Testing with NUnit

1. Objectives

- Explain the meaning of Unit Testing and its difference from Functional Testing.
- List various types of testing: Unit, Functional, Automated, and Performance Testing.
- Understand the benefit of automated testing.
- Explain the concept of loosely coupled and testable design.
- Write a testing program to validate a calculator's addition operation.
- Understand the need for `[SetUp]`, `[TearDown]`, and `[Ignore]` attributes.
- Explain the benefits of writing parameterized test cases using `[TestCase]`.

2. Unit Testing vs Functional Testing

2.1. Unit Testing

Unit testing is the process of testing individual units or components of a software system in isolation. It typically tests a single function, method, or class to ensure correctness.

2.2. Functional Testing

Functional testing validates the complete functionality of a system according to specified requirements. It often involves integration and interaction between components.

2.3. Comparison

Aspect	**Unit Testing**	**Functional Testing**
Scope	Individual method/class	Full application functionality
Dependencies	Mocked/Isolated	Real systems and components
Speed	Fast	Slower
Tooling	NUnit, xUnit	Selenium, Postman, etc.

3. Types of Testing

- **Unit Testing** – Tests individual code units.
- **Functional Testing** – Verifies application behavior against requirements.

- **Automated Testing** – Testing using code to validate functionality.
- **Performance Testing** – Measures system speed and scalability.

4. Benefits of Automated Testing

- Speeds up testing with continuous integration.
- Reduces manual effort and human error.
- Improves confidence during refactoring.
- Supports regression testing and coverage.

5. Loosely Coupled and Testable Design

A loosely coupled system reduces dependencies between classes by using interfaces or abstractions. This allows components to be tested independently using mocks.

Example:

```
public class UserManager
{
    private readonly IEmailService _service;

    public UserManager(IEmailService service)
    {
        _service = service;
    }
}
```

This makes the code testable by injecting mock objects in unit tests.

6. Calculator Implementation

6.1. Class Code: CalcLibrary/Calculator.cs

```
namespace CalcLibrary
{
    public class Calculator
    {
        public int Add(int a, int b)
        {
            return a + b;
        }
    }
}
```

7. NUnit Test Code

7.1. Test Class: CalculatorTests.cs

```
using NUnit.Framework;
using CalcLibrary;

namespace CalculatorTests
{
    [TestFixture]
    public class CalculatorTests
    {
        private Calculator calc;

        [SetUp]
        public void Init() => calc = new Calculator();

        [TearDown]
        public void Cleanup() => calc = null;

        [TestCase(2, 3, 5)]
        [TestCase(-1, 1, 0)]
        [TestCase(0, 0, 0)]
        public void Add_ReturnsExpectedResult(int a, int b, int expected)
        {
            int result = calc.Add(a, b);
            Assert.That(result, Is.EqualTo(expected));
        }

        [Test]
        [Ignore("Subtraction not implemented yet.")]
        public void Subtract_Placeholder() { }
    }
}
```

8. Explanation of Attributes

- `[TestFixture]` – Marks the test class.
- `[SetUp]` – Code to execute before each test.
- `[TearDown]` – Code to clean up after each test.
- `[Test]` – Denotes a test method.
- `[TestCase]` – Parameterized inputs and expected outputs.
- `[Ignore]` – Skips a test that is not ready.

9. Simulated Test Output

```
Running CalculatorTests...

Test Run Summary:
-------------------------
 Add_ReturnsExpectedResult (3 test cases passed)
 Subtract_Placeholder - Ignored

Result: Passed 3, Ignored 1
```

10. Conclusion

This report demonstrates the process of writing unit tests using NUnit in C. It includes the creation of a testable Calculator class, usage of setup and teardown methods, parameterized tests, and handling unimplemented logic gracefully with `[Ignore]`.

The test code validates the addition method using multiple input-output combinations and provides a robust example of test-driven development.