**NUnit HandsOn:**

1. **Meaning of Unit Testing and Its Difference from Functional Testing**

**Unit Testing**:

* Unit testing is a type of software testing where individual components or functions of a software are tested in isolation.
* It focuses on the **smallest testable part** of the code like a method or function.
* Mocking is often used to **simulate dependencies**, so the unit being tested is isolated from external systems like databases or APIs.

**Functional Testing**:

* Functional testing is a **higher-level** test that validates the software system against the functional requirements/specifications.
* It tests **end-to-end scenarios** including integration between components.

1. **Types of Testing**

· **Unit Testing**: Tests individual methods or components.

· **Functional Testing**: Validates the system works as intended (business logic).

· **Automated Testing**: Uses tools or scripts to run tests automatically.

· **Performance Testing**: Checks system performance under load (e.g., response time, throughput).

1. **Understand the benefit of automated testing**

· **Speed**: Tests run faster than manual tests.

· **Repeatability**: Can run frequently and consistently.

· **Coverage**: Tests can cover a large codebase efficiently.

· **Early Bug Detection**: Catches regressions early in development.

· **Cost-Effective**: Saves time and resources in the long term.

1. **Explain what is loosly coupled & testable design**

**Loosely Coupled**:

* Components do **not heavily depend** on one another.
* We can **swap dependencies** (e.g., with mock objects) easily.

**Testable Design**:

* Code is structured in a way that makes it easy to test.
* **Interfaces**, **dependency injection**, and **mocking frameworks are used**.

1. **Write your first testing program to validate a calculator addition operation**

using NUnit.Framework;

[TestFixture]

public class CalculatorTests

{

[Test]

public void Add\_TwoNumbers\_ReturnsSum()

{

var calc = new SimpleCalculator();

double result = calc.Addition(3, 4);

Assert.AreEqual(7, result);

}

}

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

· [SetUp]: Runs **before** each test. Good for initializing resources.

· [TearDown]: Runs **after** each test. Good for cleanup.

· [Ignore]: Temporarily skip a test (e.g., if the feature isn’t ready).

1. **Explain the benefit of writing parameterised test cases.**

· Reduces **code duplication**.

· Allows testing **multiple input-output combinations** easily.

· Increases **test coverage** with minimal code.

**Test for Calculator**

using CalcLibrary;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using NUnit.Framework;

using System;

namespace CalcLibrary.Tests

{

[TestFixture]

public class CalculatorTests

{

private SimpleCalculator \_calculator;

[SetUp]

public void Setup()

{

\_calculator = new SimpleCalculator();

}

[TearDown]

public void TearDown()

{

\_calculator = null;

}

[TestCase(10, 5, 15)]

[TestCase(0, 0, 0)]

[TestCase(-3, -6, -9)]

[TestCase(8, 12, 20)]

public void Addition\_ValidInputs\_ReturnsCorrectResult(double a, double b, double expected)

{

double result = \_calculator.Addition(a, b);

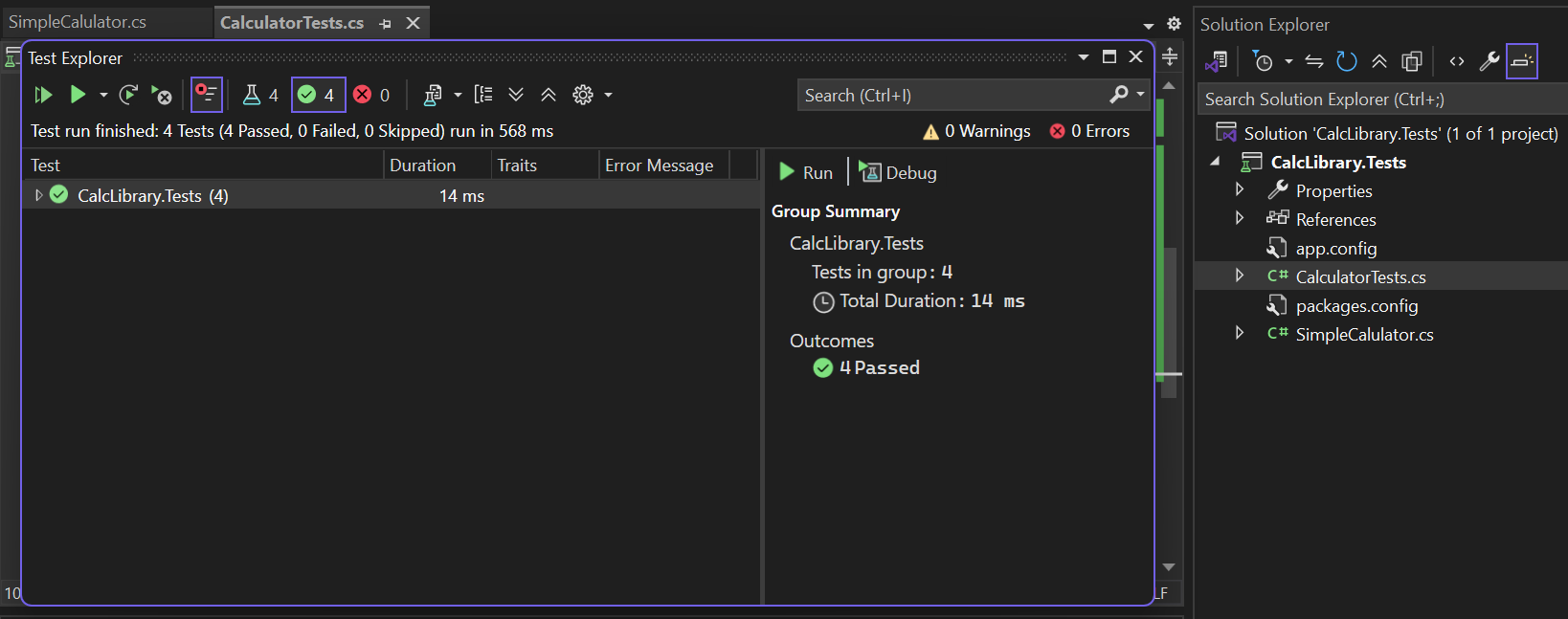
NUnit.Framework.Assert.That(result, Is.EqualTo(expected));

}

}

}

**Test Results:**



**Moq HandsOn**

#### 1. ****File: IMailSender.cs****

namespace CustomerCommLib

{

public interface IMailSender

{

bool SendMail(string toAddress, string message);

}

}

#### 2. ****File: MailSender.cs****

using System.Net;

using System.Net.Mail;

namespace CustomerCommLib

{

public class MailSender : IMailSender

{

public bool SendMail(string toAddress, string message)

{

MailMessage mail = new MailMessage();

SmtpClient smtpServer = new SmtpClient("smtp.gmail.com");

mail.From = new MailAddress("your\_email\_address@gmail.com");

mail.To.Add(toAddress);

mail.Subject = "Test Mail";

mail.Body = message;

smtpServer.Port = 587;

smtpServer.Credentials = new NetworkCredential("username", "password");

smtpServer.EnableSsl = true;

smtpServer.Send(mail);

return true;

}

}

}

#### 3. ****File: CustomerComm.cs****

namespace CustomerCommLib

{

public class CustomerComm

{

private readonly IMailSender \_mailSender;

public CustomerComm(IMailSender mailSender)

{

\_mailSender = mailSender;

}

public bool SendMailToCustomer()

{

return \_mailSender.SendMail("cust123@abc.com", "Some Message");

}

}

}

**Project 2: CustomerComm.Tests**

1. **File: CustomerCommTests.cs**

using NUnit.Framework;

using Moq;

using CustomerCommLib;

namespace CustomerComm.Tests

{

[TestFixture]

public class CustomerCommTests

{

private Mock<IMailSender> \_mockMailSender;

private CustomerCommLib.CustomerComm \_customerComm;

[OneTimeSetUp]

public void Init()

{

\_mockMailSender = new Mock<IMailSender>();

\_mockMailSender.Setup(m => m.SendMail(It.IsAny<string>(), It.IsAny<string>())).Returns(true);

\_customerComm = new CustomerCommLib.CustomerComm(\_mockMailSender.Object);

}

[Test]

public void SendMailToCustomer\_ShouldReturnTrue\_WhenMailIsSent()

{

bool result = \_customerComm.SendMailToCustomer();

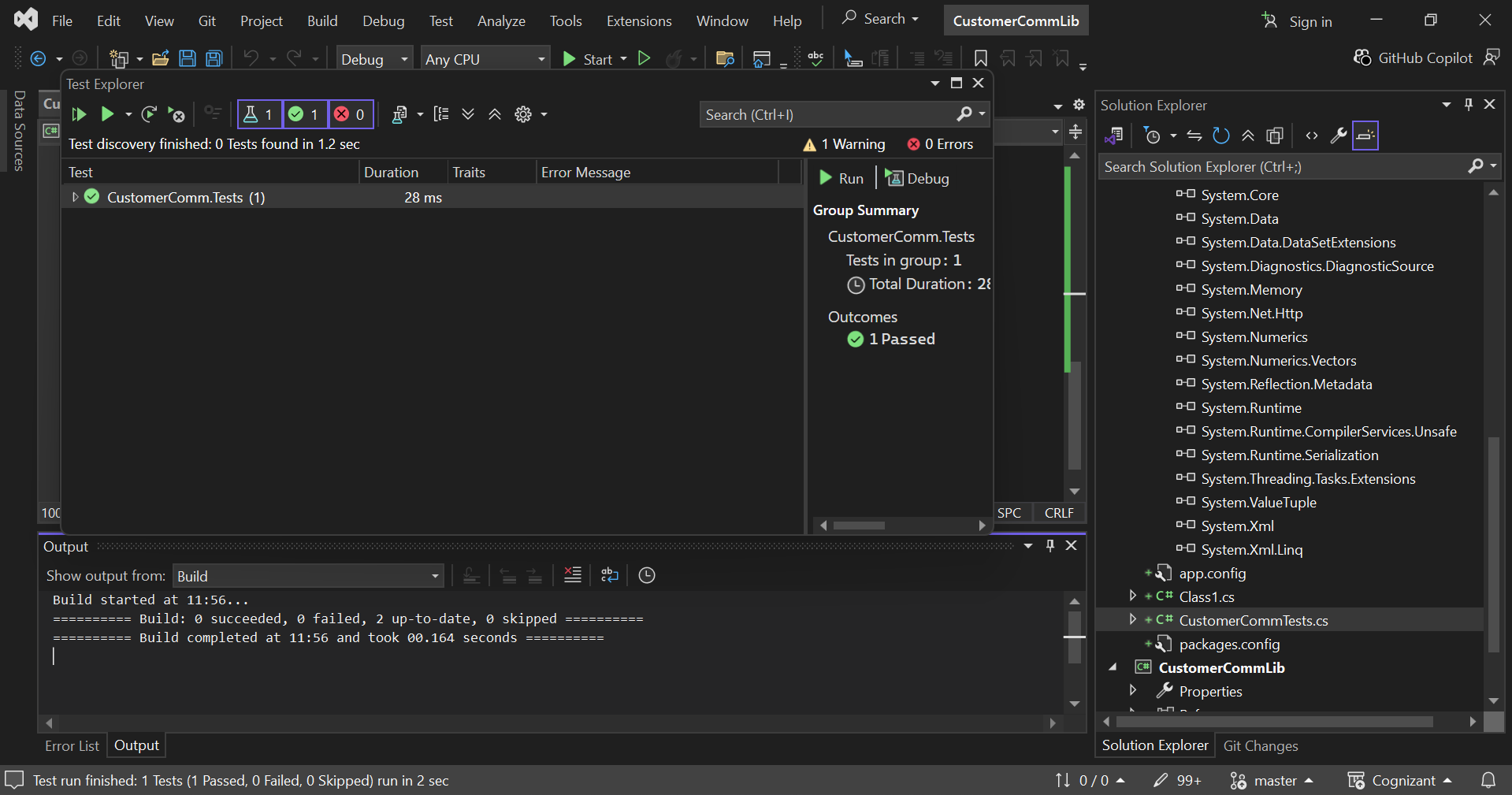
Assert.That(result, Is.True);

}

}

}

**Output**

****