**Week 2 - NUnit and Moq - Hands-On**

**NUnit\_Hands-On**

***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*

***Unit Test Code:***

using NUnit.Framework;

using CalcLibrary;

using System;

namespace CalcLibrary.Tests

{

[TestFixture]

public class CalculatorTests

{

private SimpleCalculator calc;

[SetUp]

public void Setup()

{

calc = new SimpleCalculator();

}

[TearDown]

public void Cleanup()

{

calc.AllClear();

}

[TestCase(5, 3, 8)]

[TestCase(-2, 2, 0)]

[TestCase(10.5, 4.5, 15.0)]

public void Addition\_ShouldReturnCorrectResult(double a, double b, double expected)

{

var result = calc.Addition(a, b);

Assert.AreEqual(expected, result);

}

[TestCase(10, 3, 7)]

[TestCase(0, 5, -5)]

public void Subtraction\_ShouldReturnCorrectResult(double a, double b, double expected)

{

var result = calc.Subtraction(a, b);

Assert.AreEqual(expected, result);

}

[TestCase(4, 5, 20)]

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

public void Multiplication\_ShouldReturnCorrectResult(double a, double b, double expected)

{

var result = calc.Multiplication(a, b);

Assert.AreEqual(expected, result);

}

[TestCase(20, 4, 5)]

[TestCase(9, 3, 3)]

public void Division\_ShouldReturnCorrectResult(double a, double b, double expected)

{

var result = calc.Division(a, b);

Assert.AreEqual(expected, result);

}

[Test]

public void Division\_ByZero\_ShouldThrowException()

{

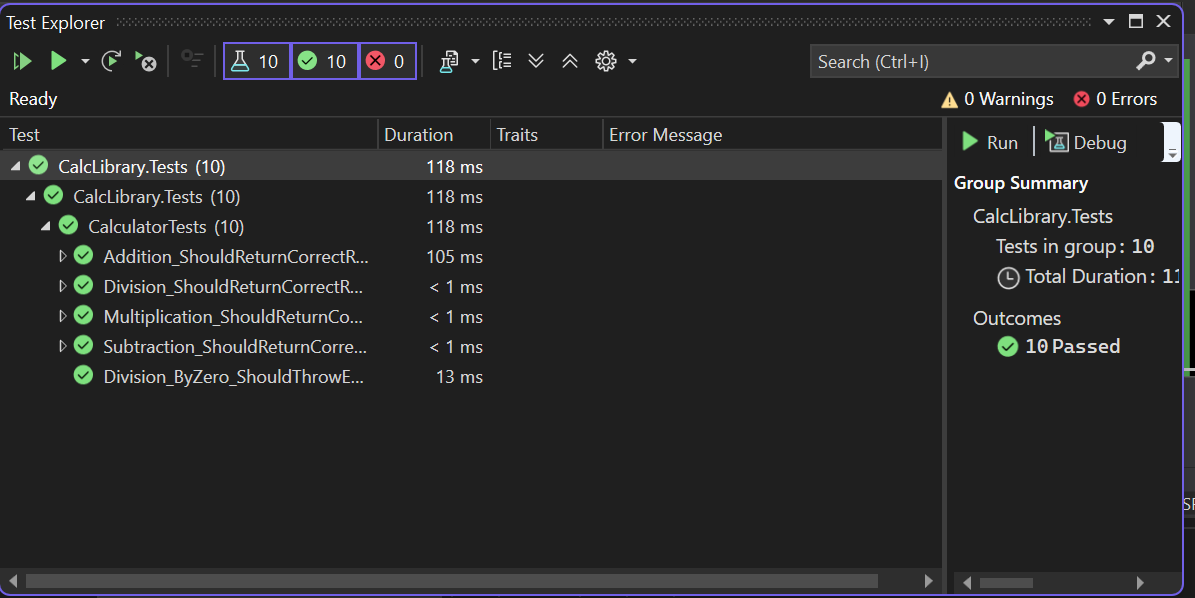
Assert.Throws<ArgumentException>(() => calc.Division(10, 0));

}

}

}

***Output:***

******

**Moq Hands-On**

***1. Write Testable Code with Moq***

Scenario:

You are assigned to unit test a module responsible for sending emails after each transaction.

To avoid sending actual emails during testing, you decide to mock the mail service

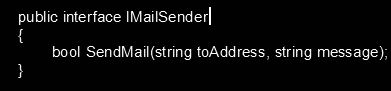
dependency. This approach improves test speed and ensures loose coupling within the

application.

**Task 1: Create the Mail Module for Testing**

● Create a C# Class Library project in Visual Studio named CustomerCommLib. Rename Class1 to MailSender, and include the namespaces System.Net and System.Net.Mail.

● Define the interface:

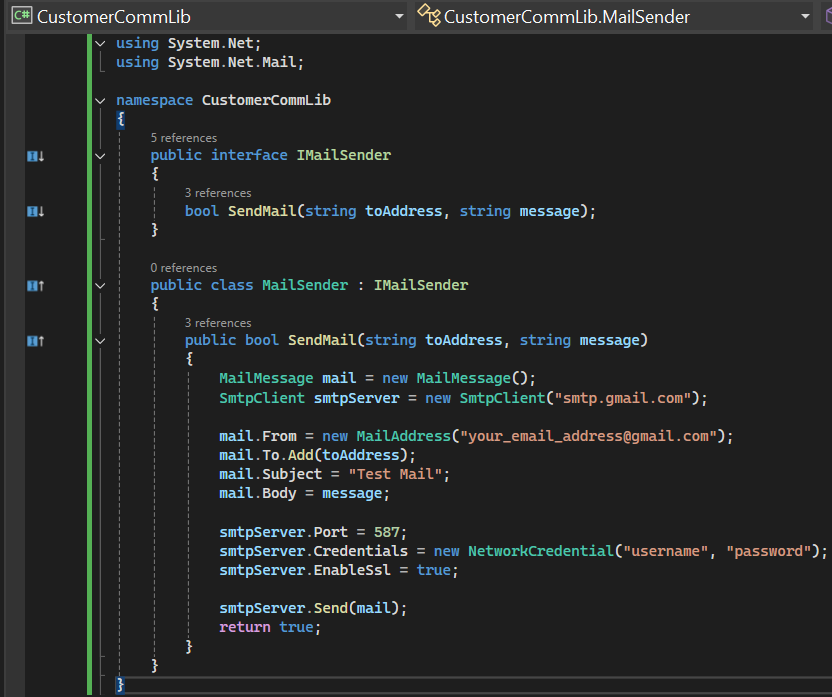


● Implement it in MailSender, which contains the SMTP logic to send emails.

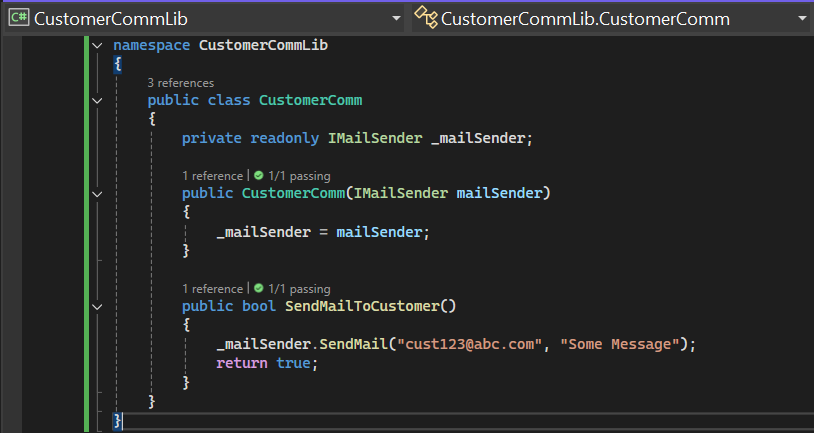
● Also, create a CustomerComm class that depends on IMailSender. Inject the dependency via the constructor and use it in SendMailToCustomer() to send a message. This design allows mocking IMailSender during unit testing, enabling testability without triggering actual emails.

● Build the project and prepare for unit testing with NUnit and Moq.

**Business Logic Codes:**

[***MailSender.cs***](http://mailsender.cs)******

***CustomerComm.cs***

******

**Task 2: Unit Testing with NUnit and Moq**

● Create a test project CustomerComm.Tests and install NUnit, NUnit Test Adapter, and Moq via NuGet. Add reference to CustomerCommLib.

● Use [TestFixture], [OneTimeSetUp], and [TestCase].

● Mock IMailSender to return true for any input.

● Test SendMailToCustomer() and assert the result with Assert.That.

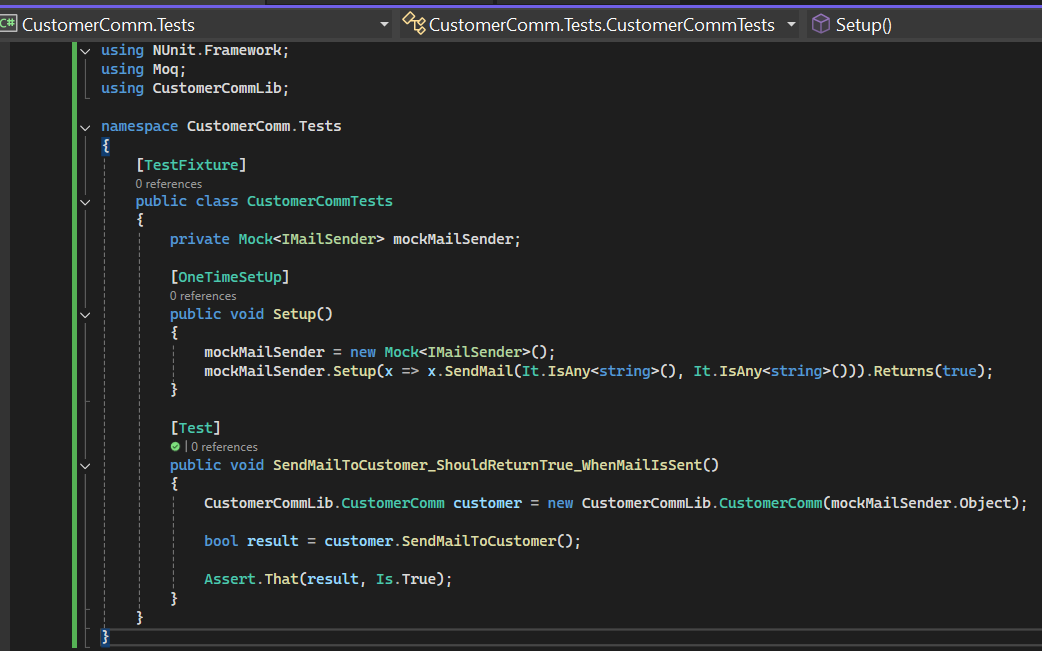
### 

### 

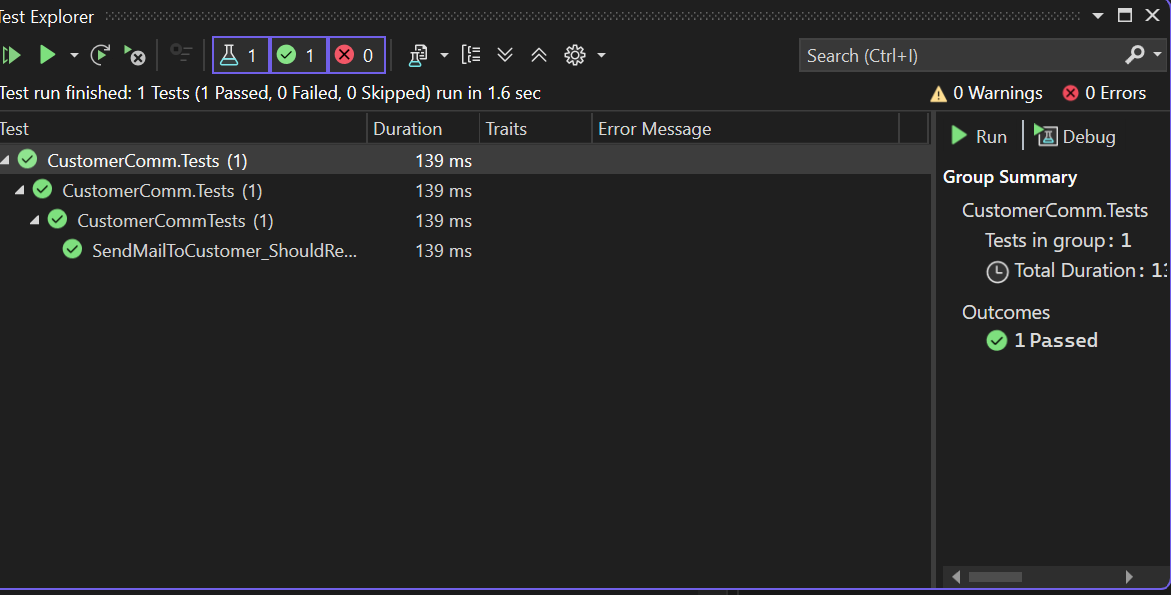
### 

### 

### **Unit Test Code: *CustomerCommTests.cs***



***Output:***

******

| **Ankita Pande**  **Superset ID: 6362381** |
| --- |