**Cognizant - DN 4.0 I Deep Skilling**

**WEEK-2**

NUnit and Moq

**EXERCISE 1: 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.

**SOLUTION :**

**CODE -:**

**PROJECT CalcLibrary :**

namespace CalcLibrary

{

public class Calculator

{

public int Add(int a, int b)

{

return a + b;

}

}

}

**Test method CalculatorTests :**

using NUnit.Framework;

using CalcLibrary; // Make sure this namespace matches your actual CalcLibrary namespace

namespace UnitTestProject

{

[TestFixture]

public class CalculatorTests

{

private Calculator \_calculator;

[SetUp]

public void Setup()

{

\_calculator = new Calculator();

}

[TestCase(3, 4, 7)]

[TestCase(10, 5, 15)]

[TestCase(-2, -3, -5)]

public void Add\_ShouldReturnCorrectSum(int a, int b, int expected)

{

var result = \_calculator.Add(a, b);

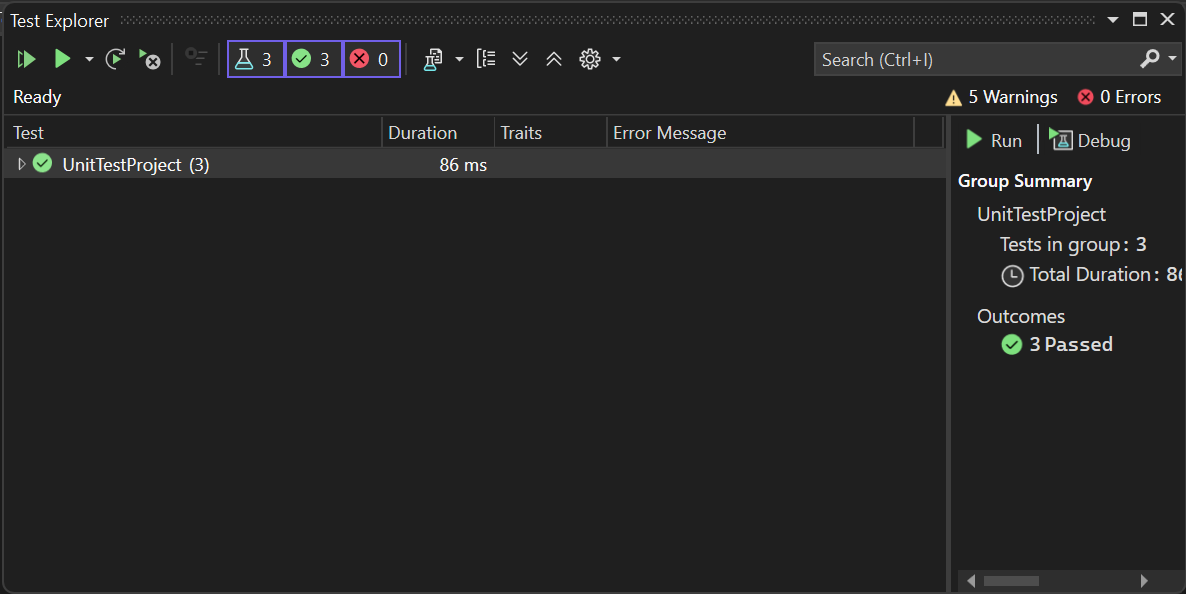
Assert.That(result, Is.EqualTo(expected));

}

}

}

**OUTPUT -:**



**Exercise 2:**

You are tasked to write a unit test code for the below scenario.

The application in which you are teamed up with, deals with a mail server communication in which your application tries to send mail to its users upon every transaction. Your role is to write unit testing the module that contains send mail functionality. You wanted to perform testing the module without sending any email.

After investigating the problem scenario, you found a solution and that is creating **mock** objects of these external dependencies in the unit testing project so that you can achieve speedier test execution and loose coupling of code.

## **Task1: Create a Testable Mail Sender Module**

You need to create a class library that encapsulates the mail-sending logic and enables mocking for unit testing.

#### ****Steps:****

#### ****Create a Class Library Project:****

Use **Visual Studio IDE**.

Name the project **CustomerCommLib**.

1. **Rename the Default Class:**

Rename the default Class1 to MailSender.

1. **Add Necessary Namespaces:**

Add references to the **System.Net** and **System.Net.Mail** namespaces.

1. **Design an Interface:**

Define an interface named IMailSender with a method for sending emails.This abstraction allows you to mock the mail-sending behavior during testing.

1. **Implement the Interface in MailSender Class:**

Implement the actual logic to send an email using SMTP in the MailSender class. Note that this implementation directly interacts with a real SMTP server and is **not suitable for unit testing** without mocking.

1. **Create the Class Under Test – CustomerComm:**

Define a class named CustomerComm which accepts an IMailSender dependency via its constructor. Implement a method in this class that utilizes the IMailSender to send a message to a hardcoded customer address.

### ****Unit Testing Considerations:****

Since the actual implementation of MailSender communicates with a real SMTP server, it's not suitable for unit testing.

To properly unit test the CustomerComm class:

Use a mocking framework like **Moq** to create a mock of IMailSender.

This allows you to test the behavior of the CustomerComm class **without sending real emails**.This approach enables **faster test execution** and promotes **loose coupling** in the code.

**SOLUTION :**

**CODE -:**

**Class MailSender :**

using System.Net;

using System.Net.Mail;

namespace CustomerCommLib

{

public class MailSender : IMailSender

{

public bool SendMail(string toAddress, string message)

{

try

{

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;

}

catch

{

return false;

}

}

}

}

**Class IMailSender :**

namespace CustomerCommLib

{

public interface IMailSender

{

bool SendMail(string toAddress, string message);

}

}

**Class CustomerComm :**

namespace CustomerCommLib

{

public class CustomerComm

{

private readonly IMailSender \_mailSender;

public CustomerComm(IMailSender mailSender)

{

\_mailSender = mailSender;

}

public bool SendMailToCustomer()

{

string message = "Some Message";

string toAddress = "cust123@abc.com";

return \_mailSender.SendMail(toAddress, message);

}

}

}

**Test method CustomerCommTests :**

using NUnit.Framework;

using Moq;

using CustomerCommLib;

namespace UnitTestProject

{

[TestFixture]

public class CustomerCommTests

{

private Mock<IMailSender> \_mockMailSender;

private CustomerComm \_customerComm;

[SetUp]

public void Setup()

{

\_mockMailSender = new Mock<IMailSender>();

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

}

[Test]

public void SendMailToCustomer\_ReturnsTrue\_WhenMailSentSuccessfully()

{

// Arrange

\_mockMailSender

.Setup(x => x.SendMail(It.IsAny<string>(), It.IsAny<string>()))

.Returns(true);

// Act

var result = \_customerComm.SendMailToCustomer();

// Assert

Assert.That(result, Is.True);

\_mockMailSender.Verify(x => x.SendMail("cust123@abc.com", "Some Message"), Times.Once);

}

}

}

**OUTPUT -:**

