WEEK - 2 (Module - 2)

Yash Raj

SuperSet ID - (6364442)

Nunit And Moq - Handson:

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

namespace CalcLibrary

{

public class Calculator

{

public int Add(int a, int b)

{

return a + b;

}

}

}

using NUnit.Framework; using CalcLibrary;

namespace Nunit

{

[TestFixture]

public class CalculatorTests

{

private Calculator calc;

[SetUp]

public void SetUp()

{

calc = new Calculator(); }

[TearDown]

public void TearDown()

{

}

[Test]

public void Add\_TwoNumbers\_ReturnsSum()

{

int result = calc.Add(2, 3);

Assert.That(result, Is.EqualTo(5));

}

[TestCase(1, 2, 3)]

[TestCase(5, 7, 12)]

[TestCase(-2, 3, 1)]

[TestCase(0, 0, 0)]

public void Add\_MultipleTestCases\_ReturnsExpected(int a, int b, int expected)

{

int result = calc.Add(a, b);

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

}

[Test, Ignore("This test is intentionally ignored")]

public void Subtract\_ThisIsPlaceholder()

{

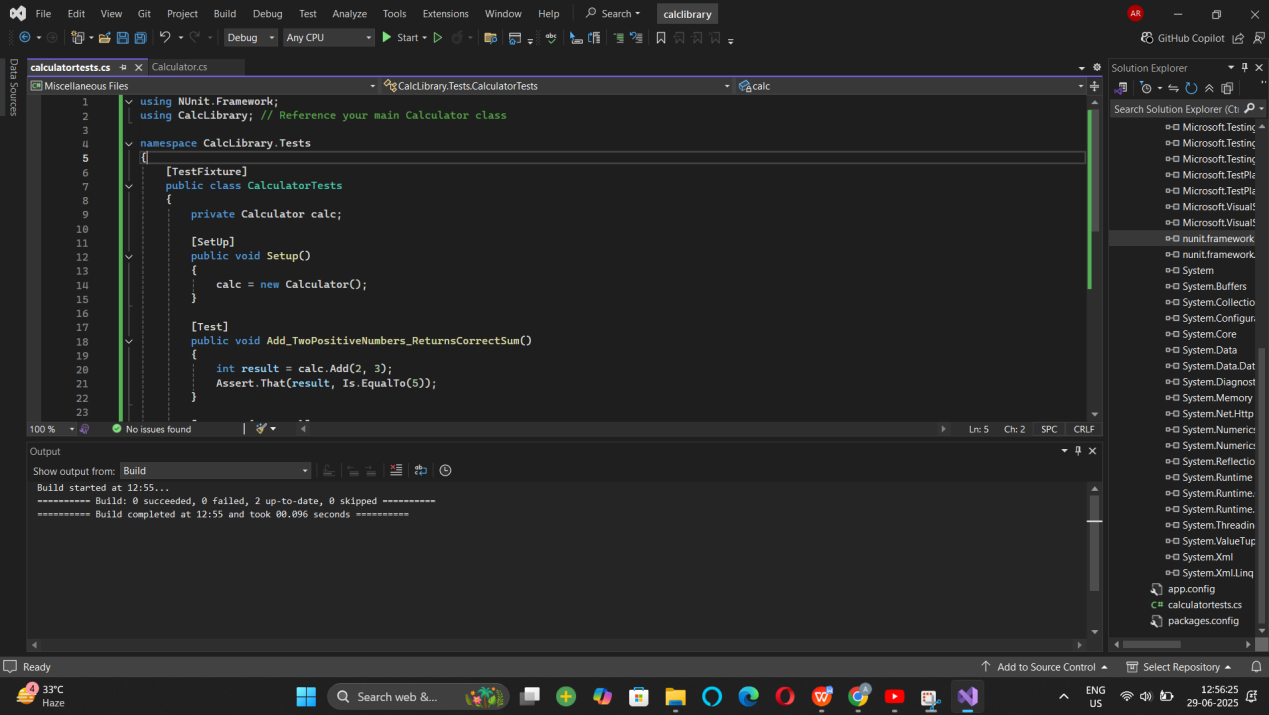
Assert.Fail("This is just a placeholder test.");

}

}

}

Output:



Exercise 2:

**Parameterized test cases**

1. Create test case to verify the subtraction feature of the calculator with various input types.

* Create test cases with ‘TestCase’ attribute to send in input parameters and the expected result.
* Add more than 1 ‘TestCase’ attributes to check various combinations for subtractions.
* Use Assert.Equal to check the actual and expected results

1. Create a test case to verify the multiplication concepts of calculator

* Create test cases with ‘TestCase’ attribute to send in input parameters and the expected result.
* Add more than 1 ‘TestCase’ attributes to check various combinations for subtractions.
* Use Assert.Equal to check the actual and expected results

1. Create a test case to verify the division logic of the calculator

* Create test cases with ‘TestCase’ attribute to send in input parameters and the expected result.
* Add more than 1 ‘TestCase’ attributes to check various combinations for subtractions.
* Use Assert.Equal to check the actual and expected results
* In one of the inputs, provide the divisor value to be 0
  + Use Try Catch block to catch the ArgumentException
  + Use Assert.Fail to notify the user that the test case has failed. Give the message “Division by zero” in the Assert.Fail, which will be notified to the user. This message will be seen in the test explorer.

**Test void methods**

In the MathLibrary class there is a property “GetResult”. The result of every operation is stored in a variable ‘result’. This value is accessed by the property.

The class also has a method “AllClear” that sets the value of the result variable to 0.

* Create a test method ‘TestAddAndClear’
* Invoke the Addition method of the math class library
* Verify if the expected and Actual results match using Assert.AreEqual
* Invoke the ‘AllClear’ method
* Use Assert.AreEqual to check if the result is 0 or not

using NUnit.Framework;

using CalcLibrary;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace CalcLibrary.Tests

{

[TestFixture]

public class CalculatorTests

{

[Test]

public void Add\_TwoNumbers\_ReturnsCorrectSum()

{

var calc = new Calculator();

int result = calc.Add(5, 3);

Assert.That(result, Is.EqualTo(8));

; }

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace CalcLibrary

{

public class Calculator

{

private int result;

public int GetResult => result;

public int Add(int a, int b)

{

result = a + b;

return result;

}

public int Subtract(int a, int b)

{

result = a - b;

return result;

}

public int Multiply(int a, int b)

{

result = a \* b;

return result;

}

public int Divide(int a, int b)

{

if (b == 0)

throw new ArgumentException("Cannot divide by zero");

result = a / b;

return result;

}

public void AllClear()

{

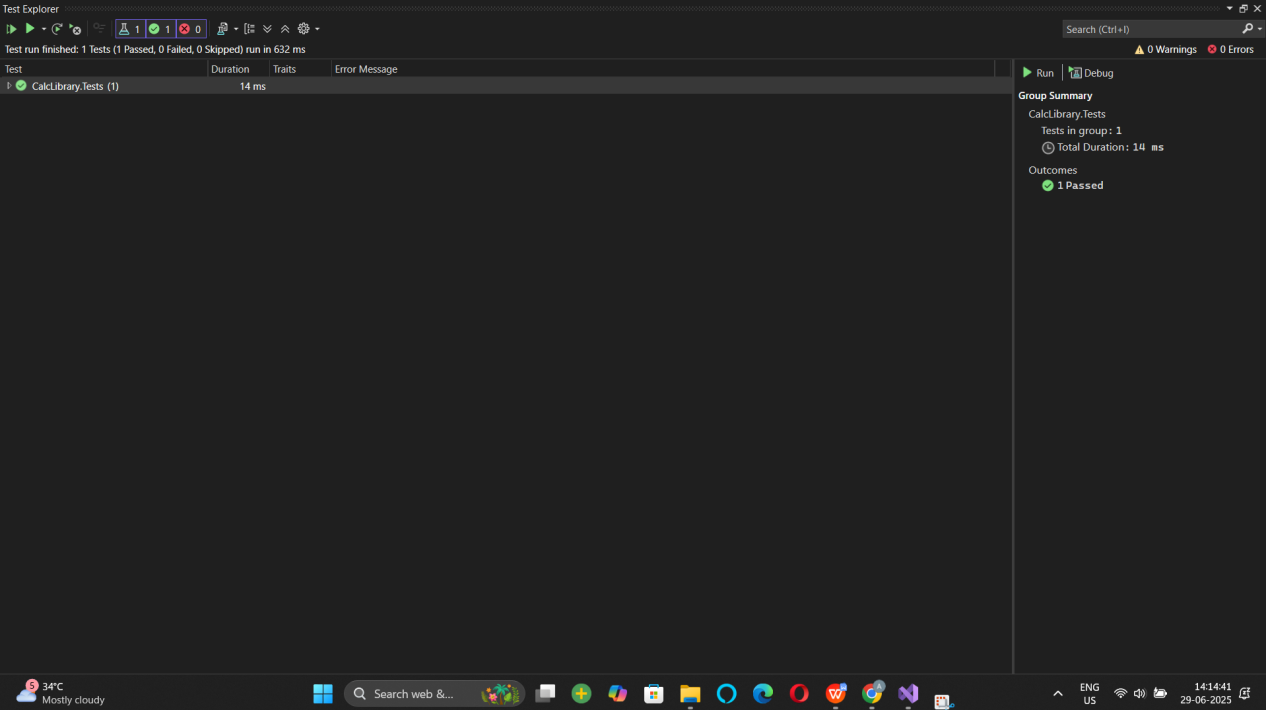
result = 0;

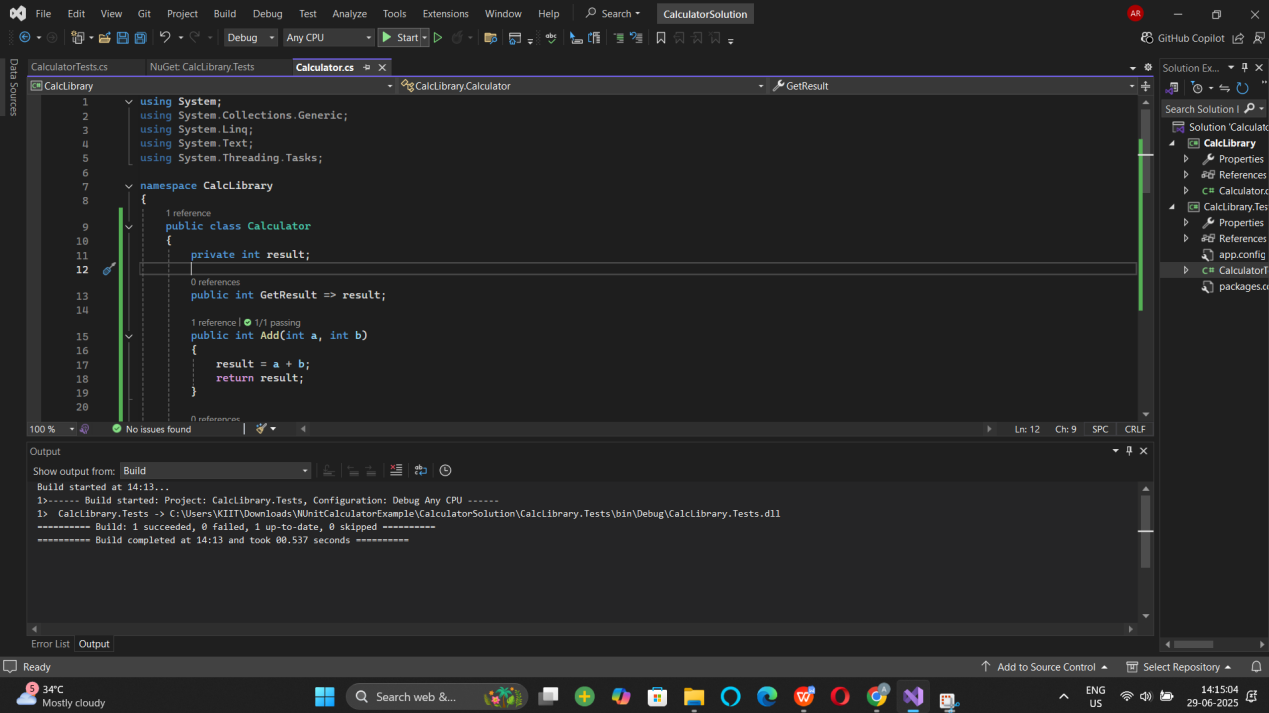
}

}

}

Output :





Moq handson:

1. Write Testable Code with Moq

## **Scenario**

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.

**Note:** Duration to complete this exercise is **30 min**.

## **Task1**

In this task, you will create a class library that will be used for unit testing.

* Create a **Class Library (Language C#)** project using Visual Studio IDE, and name it as **CustomerCommLib.**
* Rename the default **Class1** class name as **MailSender.**
* Include the following namespaces with ‘using’ directive.
  + **System.Net**
  + **System.Net.Mail**
* Define an interface as follow.

public interface IMailSender

{

        bool SendMail(string toAddress, string message);

}

* And provide implementation of **IMailSender** in the **MailSender** class as seen below.

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);

}

}

}

The above class can’t be unit testing since the code access the STMP mail server.

* Create another class called **CustomeComm** which is the **class under test** in the given scenario.

namespace CustomerCommLib

{

public class CustomerComm

{

IMailSender \_mailSender;

public CustomerComm(IMailSender mailSender)

{

\_mailSender=mailSender;

}

public bool SendMailToCustomer()

{

//Actual logic goes here

//define message and mail address

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

return true;

}

}

}

In the above code we **injected the dependency** (IMailSender) through **constructor** of **CustomerComm** class so that we can **pass the mock object** of the dependency wherever it is necessary.

We have successfully created a class that's written in such a way that we can run a unit test against it and an exception won't be thrown. We achieve this by mocking the call to IMailSender.SendMail() and adding a mocked return value of true to it.

* Finally **build** your project and be ready for the unit testing with NUnit and Moq.

using Moq;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace CustomerCommLib.Tests

{

public class CustomerCommTests

{

public void SendMailToCustomer\_ShouldReturnTrue\_WhenMailIsSentSuccessfully()

{

var mockMailSender = new Mock<IMailSender>();

mockMailSender.Setup(x => x.SendMail(It.IsAny<string>(), It.IsAny<string>())).Returns(true);

var customerComm = new CustomerComm(mockMailSender.Object);

bool result = customerComm.SendMailToCustomer();

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace CustomerCommLib

{

public class CustomerComm

{

IMailSender \_mailSender;

public CustomerComm(IMailSender mailSender)

{

\_mailSender = mailSender;

}

public bool SendMailToCustomer()

{

string to = "cust123@abc.com";

string message = "Some Message";

return \_mailSender.SendMail(to, message);

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace CustomerCommLib

{

public interface IMailSender

{

bool SendMail(string toAddress, string message);

}

}

Output:-

