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

DEBANJAN KAURI(Superset ID-6362388)

**NUnit-Handson**

* **Test Fixture & Test**

NUnit Test Case Instructions:

1. Create a Unit Test Project (.NET Framework) in the existing solution.
2. Add CalcLibrary as a project reference
3. Add a class named CalculatorTests for writing test cases.
4. Use [TestFixture], [SetUp], and [TearDown] for test class, initialization, and cleanup.
5. Write a test method to validate the Addition functionality.
6. Use [TestCase] to pass input values and expected results.
7. Use Assert.That to verify the result matches expectation.

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

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()

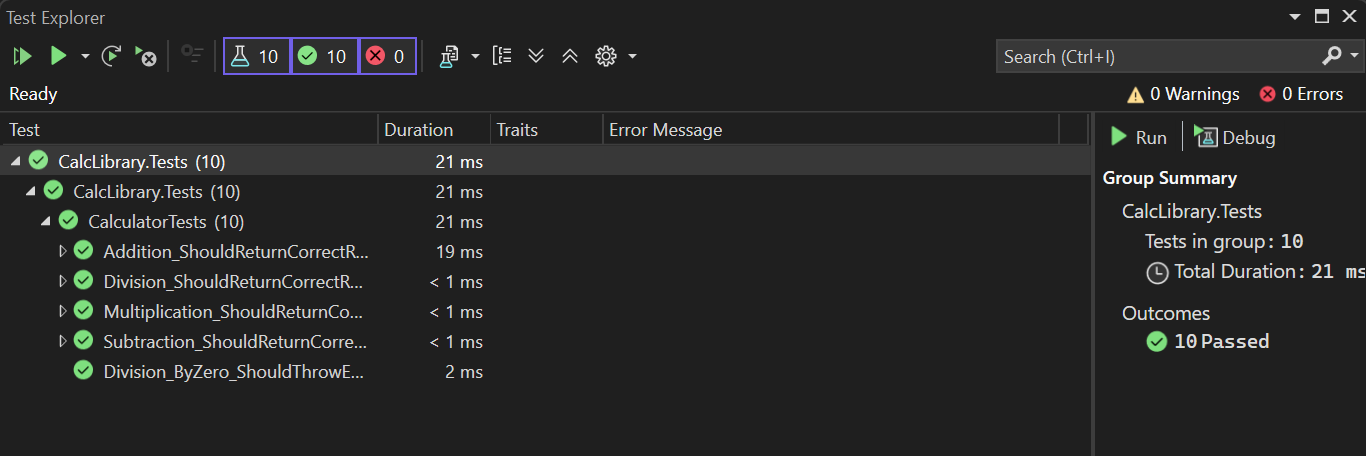
{

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

}

}

}

**OUTPUT:**

**Moq-Handson**

* **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:

public interface IMailSender

{

bool SendMail(string toAddress, string message);

}

* 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*

using System.Net;

using System.Net.Mail;

namespace CustomerCommLib

{

public interface IMailSender

{

bool SendMail(string toAddress, string message);

}

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;

}

}

}

*CustomerComm.cs*

namespace CustomerCommLib

{

public class CustomerComm

{

private readonly IMailSender \_mailSender;

public CustomerComm(IMailSender mailSender)

{

\_mailSender = mailSender;

}

public bool SendMailToCustomer()

{

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

return true;

}

}

}

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*

using NUnit.Framework;

using Moq;

using CustomerCommLib;

namespace CustomerComm.Tests

{

[TestFixture]

public class CustomerCommTests

{

private Mock<IMailSender> mockMailSender;

[OneTimeSetUp]

public void Setup()

{

mockMailSender = new Mock<IMailSender>();

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

}

[Test]

public void SendMailToCustomer\_ShouldReturnTrue\_WhenMailIsSent()

{

CustomerCommLib.CustomerComm customer = new CustomerCommLib.CustomerComm(mockMailSender.Object);

bool result = customer.SendMailToCustomer();

Assert.That(result, Is.True);

}

}

}

**OUTPUT:**