**NUnit and Moq Testing**

**1. NUnit-Handson:**

**Objectives**

* Explain the meaning of Unit testing and its difference on comparison with Functional testing
  + Smallest unit to test mocking dependencies
* List various types of testing
  + Unit testing, Functional testing, Automated testing, Performance testing
* Understand the benefit of automated testing
* Explain what is loosly coupled & testable design
  + Write code that is NOT dependent on the class for data.
* Write your first testing program to validate a calculator addition operation
  + TestFixture, Test
* Understand the need of [SetUp], [TearDown] & [Ignore] attributes.
* Explain the benefit of writing parameterised test cases.
  + TestCase

**TestFixture & Test**

Please download the application available [here](https://cognizantonline.sharepoint.com/:u:/r/sites/GTP-Solutions/Gencsharepath/Shared%20Documents/Internship2020/FSE/DotNet/02%20-%20NUnit,%20C%23%204.5,%20ASP.Net%20Core/Handson/CalcLibrary.zip?csf=1&e=aLxB66). This will be used to write Unit test cases  
  
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

Implementation :

using NUnit.Framework;

using CalcLibrary;

namespace CalculatorUnitTests

{

    [TestFixture]

    public class CalculatorTests

    {

        private SimpleCalculator calculator;

        [SetUp]

        public void SetUp()

        {

            calculator = new SimpleCalculator();

        }

        [TearDown]

        public void TearDown()

        {

            calculator.AllClear();

        }

        [Test]

        [TestCase(2, 3, 5)]

        [TestCase(-1, -4, -5)]

        [TestCase(1.5, 2.5, 4.0)]

        public void TestAddition(double a, double b, double expected)

        {

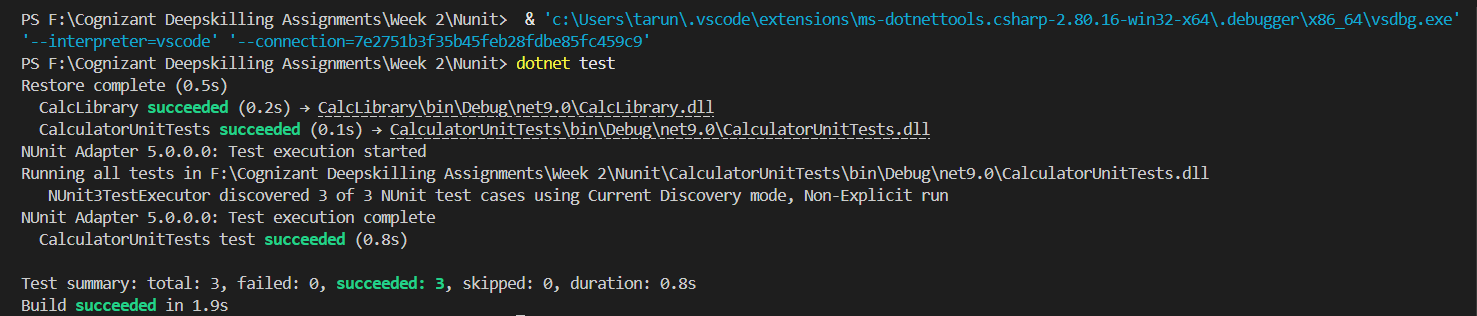
            var actual = calculator.Addition(a, b);

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

        }

    }

}



**Explanation:**

* **What is Unit Testing? How is it Different from Functional Testing?**
* Unit testing focuses on checking individual components or small parts (units) of a program, like a single method or function, to ensure it works as expected. The goal is to isolate one piece of logic and verify it independently.  
    
  Functional testing, on the other hand, verifies that the system or application behaves correctly as a whole. It checks the output based on input, without concern for internal code.  
    
  Key Difference:  
  - Unit testing = Small, isolated code blocks  
  - Functional testing = End-to-end functionality of features  
    
  Mocking dependencies helps unit testing by replacing real services (like a database or email server) with fake objects so you can test the logic in isolation.
* **List of Various Testing Types**
  + Unit Testing: Testing individual units (methods/functions) of code.  
    - Functional Testing: Testing application behavior against requirements.  
    - Automated Testing: Running tests automatically via tools/scripts.  
    - Performance Testing: Checking how fast or stable the system is under load.
* **Benefits of Automated Testing**
  + Saves time by running tests quickly and repeatedly  
    - Detects bugs early in development  
    - Helps ensure code changes don’t break existing functionality  
    - Enables Continuous Integration / Delivery (CI/CD)  
    - Increases confidence in code reliability
* **Loosely Coupled & Testable Design**
* A loosely coupled design means that parts of the code are independent. For example, a class should not create or depend directly on another class—it should depend on an interface. This makes it easier to test or replace dependencies.  
    
  Testable design uses Dependency Injection (DI) so mock objects can be passed during testing instead of real ones.  
    
  Example:  
  Bad (tight coupling):  
   var mailer = new MailSender();  
    
  Good (loose coupling via interface):  
   IMailSender mailer = new MockMailSender();
* **First Testing Program: Calculator Addition (NUnit Example)**
* using NUnit.Framework;  
    
  [TestFixture]  
  public class CalculatorTests  
  {  
   [Test]  
   public void Add\_WhenGivenTwoNumbers\_ReturnsSum()  
   {  
   var result = 2 + 3;  
   Assert.AreEqual(5, result);  
   }  
  }
* **Understanding [SetUp], [TearDown], and [Ignore] Attributes**
  + [SetUp]: Code inside this method runs before each test — used to initialize test objects.  
    - [TearDown]: Runs after each test — used for cleanup.  
    - [Ignore]: Skips a test during execution (e.g., not ready or temporarily disabled).
* **Benefits of Parameterized Test Cases ([TestCase])**
* - Allows you to run the same test method with multiple input values  
  - Reduces code duplication  
  - Makes tests cleaner and more maintainable  
    
  Example:  
  [TestCase(2, 3, 5)]  
  [TestCase(-1, -2, -3)]  
  [TestCase(0, 10, 10)]  
  public void Add\_ValidInputs\_ReturnsExpectedResult(int a, int b, int expected)  
  {  
   Assert.AreEqual(expected, a + b);  
  }

**Moq-Handson**

Objectives:

* Understand how Mocking can enhance Test-Driven Development (TDD)
  + Mocking, Isolation, Test doubles, Mock Vs Fake Vs Stub, Key advantages of TDD
* Explain the meaning of Mocking in Unit Testing and why use mocks in Unit Testing
  + Mocking and Isolation in Unit Testing, Isolating dependencies in Tests using Mocks and Stubs
* Understand the basics of DI (Dependency Injection) and how dependency injection helps unit testing in applications
  + Dependency Injection, Constructor Injection, Method Injection
* Demonstrate on how to create a testable code with Moq.
  + Testable code
* Demonstrate on how to create a mock object that access database for unit tests
  + Mock database for Unit Tests
* Demonstrate on mock object that access the file system for unit tests
  + Mock files for Unit Tests

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.

## **Task2**

In this task, you will create unit test project which make use of NUnit framework and Moq.

* Create a new class library project called **CustomerComm.Tests** and add the following external dependencies to it using **NuGet Package Manager.**
  + NUnit
  + NUnit Test Adapter
  + Moq
* Add the references of assemblies as appropriate including **CustomerCommLib.**
* Write unit test code and **mock** the **MailSender (IMailSender)** class.
* Use **TestFixture**, **OneTimeSetUp** and **TestCase** attribute classes on top of test class, init method and test method respectively.
* **Configure** the mock object in such away that **SendMail()** method will accept any two string arguments and always return true when **SendMailToCustomer()** gets invoked.
* Finally **assert** the return value to “true”.

Implementation:

using NUnit.Framework;

using Moq;

using CustomerCommLib;

namespace CustomerCommTests

{

    [TestFixture]

    public class CustomerCommTests

    {

        private Mock<IMailSender> mockMailSender;

        private CustomerComm customerComm;

        [OneTimeSetUp]

        public void Init()

        {

            mockMailSender = new Mock<IMailSender>();

            mockMailSender

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

                .Returns(true);

            customerComm = new CustomerComm(mockMailSender.Object);

        }

        [Test]

        [TestCase]

        public void SendMailToCustomer\_WhenCalled\_ReturnsTrue()

        {

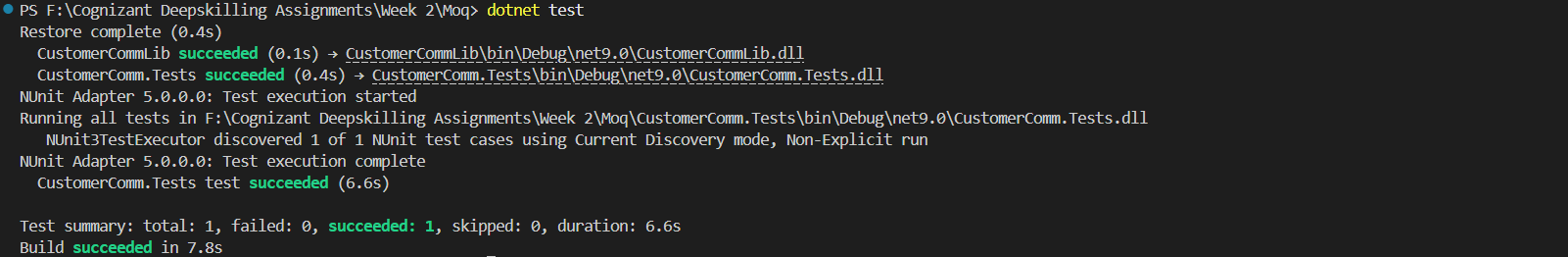
            var result = customerComm.SendMailToCustomer();

            Assert.That(result, Is.True);

        }

    }

}



**Mocking, Dependency Injection, and Testable Code in TDD**

* **How Mocking Enhances Test-Driven Development (TDD)**
* Mocking helps in isolating the part of the code being tested by replacing real dependencies (e.g., a database or a network) with fake objects that simulate behavior. This isolation improves speed and accuracy of tests and allows testing in early stages of development.  
    
  Key concepts:  
  - Mocking: Creating fake objects to simulate real ones.  
  - Isolation: Running the test without relying on external systems.  
  - Test Doubles: General term for objects like mocks, stubs, fakes, and spies.  
  - Mock vs Fake vs Stub:  
   - Mock: Verifies behavior (e.g., method called or not).  
   - Stub: Returns predefined data but doesn’t verify usage.  
   - Fake: A simplified real implementation, often in-memory.  
  - TDD Advantages:  
   - Catches issues early  
   - Leads to better design  
   - Encourages modular and testable code
* **Meaning of Mocking in Unit Testing and Its Purpose**
* Mocking in unit testing means replacing real dependencies with controllable and lightweight mock objects. These help isolate the logic being tested and avoid unintended effects like network calls, database hits, or file writes.  
    
  Mocking allows:  
  - Testing only the business logic  
  - Avoiding slow or unstable external services  
  - Ensuring repeatability and speed of tests  
    
  Isolation ensures the test fails only when the unit under test is broken — not because of a network or I/O error.
* **Basics of Dependency Injection (DI) for Unit Testing**
* Dependency Injection is a design pattern that passes dependencies from the outside instead of creating them inside the class.  
    
  Types of DI:  
  - \*\*Constructor Injection\*\*: Injecting dependencies through the constructor (most common)  
  - \*\*Method Injection\*\*: Passing dependencies through a method parameter  
  - \*\*Property Injection\*\*: Setting dependency via a public property (less common)  
    
  Benefit in testing: You can inject mock versions of services during unit testing, improving testability and reducing coupling.
* **Creating Testable Code with Moq**
* To create testable code, design classes to depend on interfaces, and inject dependencies (e.g., services or clients). Then in your unit test, use the Moq library to provide mock implementations.  
    
  Example:  
  ```csharp  
  var mockService = new Mock<IService>();  
  mockService.Setup(s => s.GetData()).Returns("Fake Data");  
    
  var controller = new MyController(mockService.Object);  
  Assert.AreEqual("Fake Data", controller.Fetch());  
  ```  
    
  This ensures the test checks your logic and not the real service.
* **Mocking a Database in Unit Tests**
* Databases are slow and hard to control in tests. To mock a database:  
  - Use an interface like `IRepository` or `IDbService`  
  - Replace it with a Moq object during test  
  - Stub methods like `GetUser()` or `Save()`  
    
  Example:  
  ```csharp  
  var mockRepo = new Mock<IUserRepository>();  
  mockRepo.Setup(r => r.GetUser(1)).Returns(new User { Id = 1, Name = "Test" });  
  ```  
    
  This avoids hitting the actual database and improves speed and reliability.
* **Mocking File System Access in Unit Tests**
* Interacting with the file system is slow and environment-specific. Use interfaces like `IFileService` and replace real implementations with mocks.  
    
  Example:  
  ```csharp  
  var mockFileService = new Mock<IFileService>();  
  mockFileService.Setup(f => f.Read("data.txt")).Returns("file content");  
  ```  
    
  This lets you test how your code reacts to file inputs without depending on actual files.