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. Mocking is a central technique in Test-Driven Development (TDD) to enable unit under test isolation through mocking out external dependencies. In TDD, we first write the test and then write actual production code. With the use of mocks, we are able to test units even when their dependencies such as databases, APIs, or email servers have not been implemented yet. This technique promotes writing loosely coupled code and accelerates development because it enables directed, speedy, and consistent tests. Most commonly used test doubles in TDD include mocks, fakes, and stubs. Mocks are programmable objects that verify interactions, stubs return fixed results, and fakes are working implementations with simplified behavior. By using these techniques, TDD becomes more effective and maintainable.

2. Mocking in unit testing refers to creating substitute objects that mimic the behavior of real dependencies. The goal is to encapsulate the unit under test so that test results are not affected by the behavior or state of the outside system. For instance, when testing a method that sends an email, we may mock the email sender class to prevent actual emails from being sent. Mocks allow us to assert that some methods were called with given parameters. They also avoid side effects like writing to a database or issuing network requests. This gives tests that are faster and more consistent and helps find problems at the unit level.

3. Dependency Injection (DI) is a design pattern applied to accomplish Inversion of Control (IoC) between dependencies and classes. As opposed to establishing dependencies internally, they are supplied from the outside—either through constructor injection, method injection, or property injection. In unit testing, DI facilitates easy replacement of real dependencies with mock objects. For instance, if a class relies on an email sender, injecting the dependency enables passing in a mock object during testing. This method isolates classes from each other, making them more modular and testable in isolation.

4. When we write testable code with Moq, we declare interfaces for the components that will be faked and inject them into the classes to be tested. For instance, if we have a class that sends an email using `IMailSender`, we can use Moq to mock out `IMailSender` and set it up to return a certain value. This enables us to test if the class properly calls the `SendMail` method without actually sending an email. Moq has methods such as `.Setup()`, `.Returns()`, and `.Verify()` for specifying behavior and validating method calls, making it possible to thoroughly test interaction logic.

5. To make a mock object for a database, we abstract the database logic behind an interface such as `IPlayerMapper`. You can then mock this interface with Moq to emulate database behavior. For example, we can set `IsPlayerNameExistsInDb()` to return false in a unit test that states a player name is unique. In the same way, `AddNewPlayerIntoDb()` can be mocked to ensure it was called with the proper arguments. This enables testing the logic in classes such as `Player.RegisterNewPlayer()` without a real database connection, which enhances test speed and autonomy.

6. Mocking file system access requires encapsulating static classes such as `Directory` into interfaces such as `IDirectoryExplorer`. You can then mock this interface to return pre-defined lists of files during unit testing. For instance, rather than calling `Directory.GetFiles()`, in our production code we would call `GetFiles()` on `IDirectoryExplorer`. In our unit test, we use Moq to arrange the `GetFiles()` method to return a list such as `["file1.txt", "file2.txt"]`. This helps ensure our test no longer relies on real file contents in the file system, thus being more portable and easier to execute in any environment.

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.

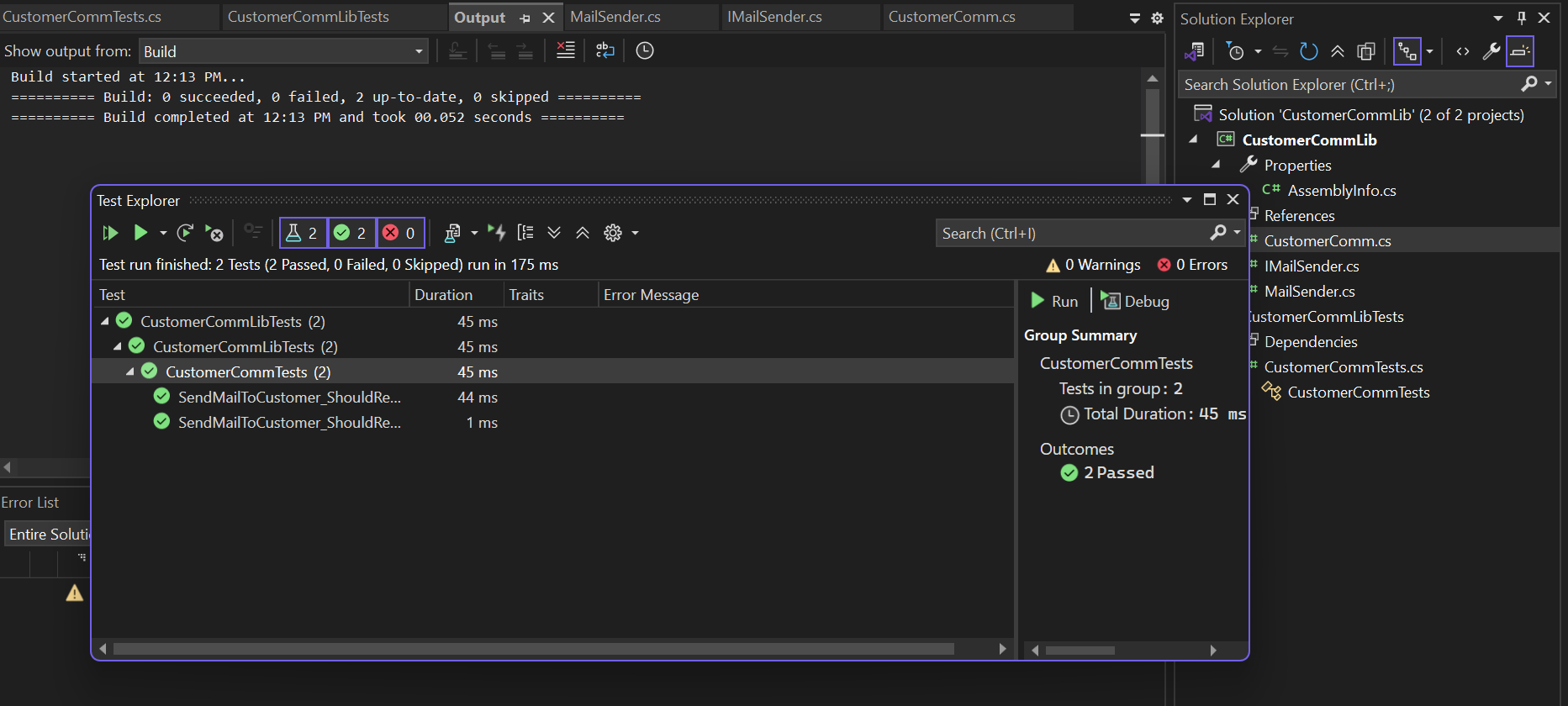
We have successfully created a class that's written in such a way that we can run a unit test against it and an exception won't be thrown. We achieve this by mocking the call to IMailSender.SendMail() and adding a mocked return value of true to it.

* 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”.



1. Mock file object for Unit Tests

## **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 the file system and it searches for files and retrieves files under the specified path. In the existing system, **Directory.GetFiles()** method has been used. You found that it’s not good idea to use Directory.GetFiles from the System.IO being its **static** and **unable to unit tes**t such methods.

After investigating the problem scenario, you found a solution and that is refactoring the code. Instead of using directly the static method Directory.GetFiles, you decided to create your own implementation to the method so that be able to **mock** files in the Unit Tests.

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

## **Task1**

* Create a **Class Library (Language C#)** project using Visual Studio IDE, and name it as **MagicFilesLib.**
* Rename the default **Class1** class name as **DirectoryExplorer** and include the following code snippet into it.
* Include the following namespaces with ‘using’ directive.
  + **System.Collections.Generic**
  + **System.IO**
* Define an interface as follow.

public interface IDirectoryExplorer

     {

         ICollection<string> GetFiles(string path);

     }

* And provide implementation of **IDirectoryExplorer** in the **DirectoryExplorer** class as seen below.

namespace MagicFilesLib

{

public class DirectoryExplorer: IDirectoryExplorer

{

public ICollection<string> GetFiles(string path)

{

string[] files = Directory.GetFiles(path);

return files;

}

}

}

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

## **Task2**

* Create a new class library project called **DirectoryExplorer.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 **MagicFilesLib.**
* Write unit test code and **mock the DirectoryExplorer (IDirectoryExplorer),** which is the class under test, with some hard coded file names.
* Use **TestFixture**, **OneTimeSetUp** and **TestCase** attribute classes on top of test class, init method and test method respectively.
* Add the following declarations in the test class.

private readonly string \_file1 = "file.txt";

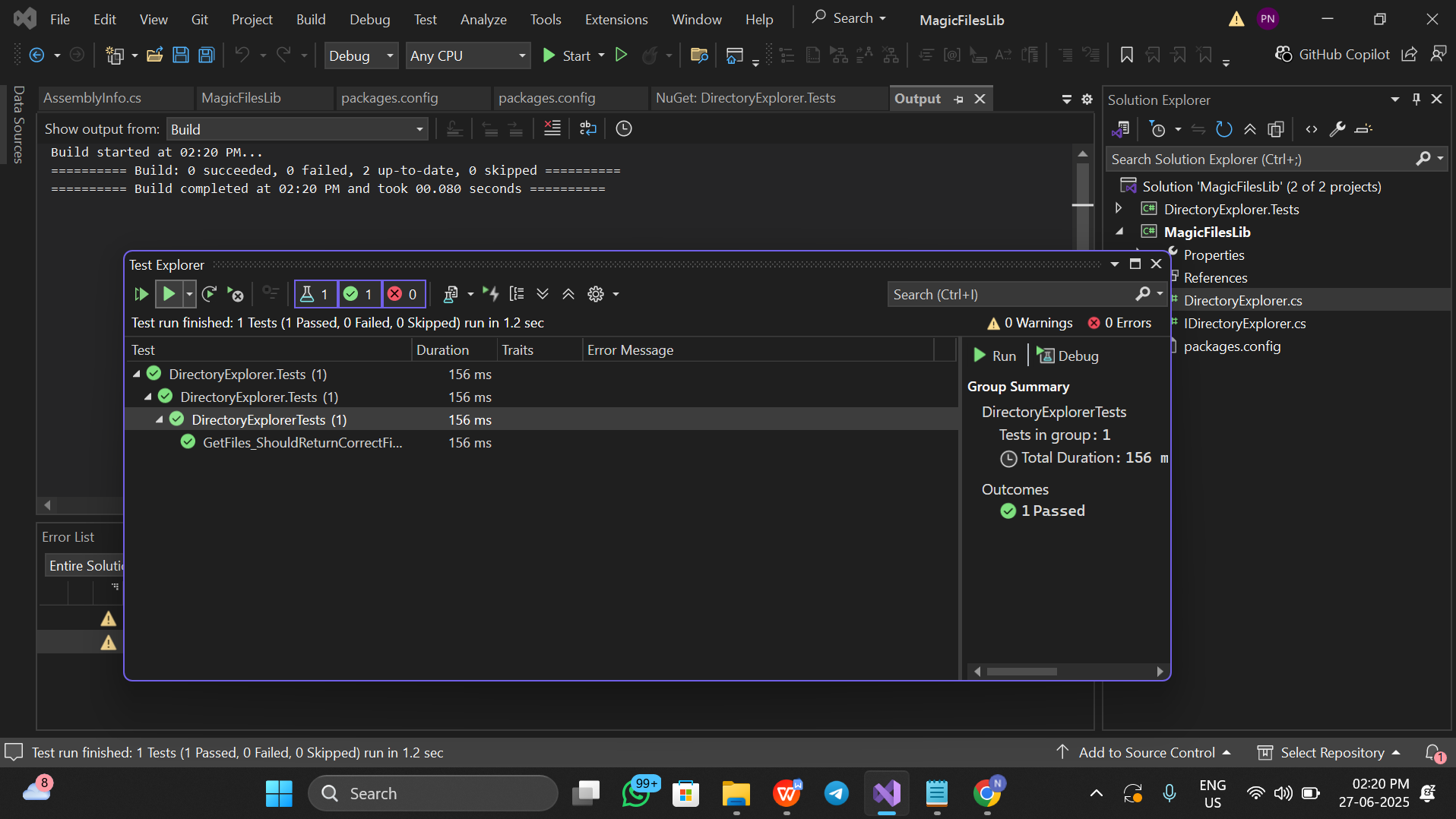
private readonly string \_file2 = "file2.txt";

* In the test method, **assert** the following so that,

*the collection is not null*

*the collection count is equal to 2*

*the collection contains \_file1*



1. Mock database for Unit Tests

## **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 network database in which your application stores the record or certain players. It involves storing and retrieval of player details. Your role is to write unit testing the player module which involves an external dependency. You can’t proceed with unit testing.

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 **60 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 **PlayersManagerLib.**
* Rename the default **Class1** class name as **PlayerManager.**
* Include the following namespaces with ‘using’ directive.
  + **System.Data**
  + **System.Data.SqlClient**
* Define an interface as follow.

public interface IPlayerMapper

{

        bool IsPlayerNameExistsInDb(string name);

        Void AddNewPlayerIntoDb(string name);

}

* And provide implementation of **IPlayerMapper** in the **PlayerMapper** class as seen below.

namespace PlayersManagerLib

{

public class PlayerMapper: IPlayerMapper

{

private readonly string \_connectionString =

"Data Source=(local);Initial Catalog=GameDB;Integrated Security=True";

public bool IsPlayerNameExistsInDb(string name)

{

using(SqlConnection connection = new SqlConnection(\_connectionString))

{

connection.Open();

using(SqlCommand command = connection.CreateCommand())

{

command.CommandText = "SELECT count(\*) FROM Player WHERE 'Name' = @name";

command.Parameters.AddWithValue("@name", name);

// Get the number of player with this name

var existingPlayersCount = (int) command.ExecuteScalar();

// Result is 0, if no player exists, or 1, if a player already exists

return existingPlayersCount > 0;

}

}

}

public void AddNewPlayerIntoDb(string name)

{

using(SqlConnection connection = new SqlConnection(\_connectionString))

{

connection.Open();

using(SqlCommand command = connection.CreateCommand())

{

command.CommandText = "INSERT INTO Player ([Name]) VALUES (@name)";

command.Parameters.AddWithValue("@name", name);

command.ExecuteNonQuery();

}

}

}

}

}

The above class can’t be unit testing since the code access the database.

* Create another class called **Player** and add the following codes.

public class Player

{

public string Name { get; private set; }

public int Age { get; private set; }

public string Country { get; private set; }

public int NoOfMatches {get; private set;}

public Player(string name, int age, string country, int noOfMatches)

{

Name = name;

Age=age;

Country= country;

NoOfMatches = noOfMatches;

}

public static Player RegisterNewPlayer(string name, IPlayerMapper playerMapper = null)

{

// If a PlayerMapper was not passed in, use a real one.

// This allows us to pass in a "mock" PlayerMapper (for testing),

// but use a real PlayerMapper, when running the program.

if(playerMapper == null)

{

playerMapper = new PlayerMapper();

}

if(string.IsNullOrWhiteSpace(name))

{

throw new ArgumentException("Player name can’t be empty.");

}

// Throw an exception if there is already a player with this name in the database.

if(playerMapper.IsPlayerNameExistsInDb (name))

{

throw new ArgumentException("Player name already exists.");

}

// Add the player to the database.

playerMapper. AddNewPlayerIntoDb (name);

return new Player(name, 23, “India”,30);

}

}

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 **PlayerManager.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 **PlayersManagerLib.**
* Write unit test code and **mock** the **PlayerMapper (IPlayerMapper)** class.
* Use **TestFixture**, **OneTimeSetUp** and **TestCase** attribute classes on top of test class, init method and test method respectively.
* Use **ExpectedException** attributeto specify that the execution of a test will throw an exception.
* When the **RegisterNewPlayer** function calls **IsPlayerNameExistsInDb**, you need to make sure that the mock object to return **“false”.**
* In the test method, **assert** various player attributes.

