Lecture: Introduction to Unit Testing in C#

1. Introduction

- **Definition of Unit Testing**: Testing individual components of the software in isolation.
- **Importance of Unit Testing**: Ensures code quality, helps find bugs early, facilitates refactoring, and supports documentation.

2. Unit Testing Frameworks in C#

- **Popular Frameworks**:
  - MSTest: Microsoft's test framework.
  - NUnit: A widely used open-source testing framework.
  - xUnit: Modern, flexible, and extensible testing framework.

3. Writing Your First Unit Test

- **Anatomy of a Unit Test**:
  - Arrange: Set up any required variables, objects, or dependencies.
  - Act: Execute the method or functionality being tested.
  - Assert: Verify that the outcome is as expected.

4. Writing and Running Your First Unit Test

Example Class to Test:

```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

NUnit Test Example

```
using NUnit.Framework;

[TestFixture]
public class CalculatorTests
{
```

```
    private Calculator calculator;

    [SetUp]
    public void Setup()
    {
        calculator = new Calculator();
    }

    [Test]
    public void Add_ReturnsCorrectSum()
    {
        // Arrange
        int a = 2;
        int b = 3;

        // Act
        int result = calculator.Add(a, b);

        // Assert
        Assert.AreEqual(5, result);
    }
}
```

Explanation

- **Setup (`[SetUp]`)**: This attribute marks a method that is executed before each test method (`[Test]`). Here, we initialize the `Calculator` instance.
- **Test Method (`[Test]`)**: This attribute marks a method as a test method. In this example:
    - **Arrange**: We set up the input values (`a` and `b`).
    - **Act**: We call the `Add` method of the `Calculator` instance with the arranged values.
    - **Assert**: We verify that the result of `Add(2, 3)` is equal to `5`.

Running NUnit Tests

To run NUnit tests:

- Ensure NUnit framework is installed and integrated into your Visual Studio project.

This example demonstrates how to write and execute unit tests using NUnit in C#.

Adding parameterize test:

```
using NUnit.Framework;

[TestFixture]
public class CalculatorTests
{
    private Calculator calculator;

    [SetUp]
    public void Setup()
    {
        calculator = new Calculator();
    }

    [TestCase(2, 3, 5)]
    [TestCase(0, 0, 0)]
    [TestCase(-1, 1, 0)]
    [TestCase(-2, -3, -5)]
    [TestCase(100, 200, 300)]
    public void Add_ReturnsCorrectSum(int a, int b, int expectedSum)
    {
        // Act
        int result = calculator.Add(a, b);

        // Assert
        Assert.AreEqual(expectedSum, result);
    }
}
```

Testing Exceptions

- **ExpectedException Attribute**:

```
[Test]
public void Divide_ByZeroThrowsException()
{
    // Arrange
    int a = 10;
    int b = 0;

    // Act & Assert
    Assert.Throws<DivideByZeroException>(() => calculator.Divide(a, b));
}
```

Lecture 2 Unit Testing with NUnit:

5. Mocking Dependencies with Moq

Let's go through a complete example of how to use Moq to mock dependencies in unit tests with NUnit.

Scenario Recap

We have an `OrderProcessor` class that processes orders. Part of its job is to send a confirmation email using an `IEmailService`. We want to test that when an order is processed, the `OrderProcessor` correctly uses the `IEmailService` to send a confirmation email.

What We Are Testing

We are testing that the `OrderProcessor`:

1. Sends an email to the correct recipient.
2. Uses the correct subject line for the email.
3. Uses the correct body content for the email.

We do this by verifying that the `SendEmail` method of the `IEmailService` is called with the correct parameters when `OrderProcessor.ProcessOrder` is executed.

Purpose of the Test

The purpose of this test is to ensure that when an order is processed, the `OrderProcessor` correctly uses the `IEmailService` to send an email with the expected details. By using a mock `IEmailService`, we isolate the test from the actual email sending logic and focus on verifying the interaction between `OrderProcessor` and `IEmailService`.

Moq N-Tier Application

In this example, we will mock the Data Access Layer (DAL) to test the Business Logic Layer (BLL) using Moq.

Scenario

We have a simple application that manages customers. The `CustomerService` in the BLL depends on a `ICustomerRepository` in the DAL to retrieve and save customer data.

Code Example

ICustomerRepository Interface (DAL)

This interface defines the methods for data access operations.

```
public interface ICustomerRepository
{
    Customer GetCustomerById(int id);
    void SaveCustomer(Customer customer);
}
```

Customer Class

A simple POCO class representing a customer.

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}
```

CustomerService Class (BLL)

This class contains business logic and depends on `ICustomerRepository`.

```
public class CustomerService
{
    private readonly ICustomerRepository _customerRepository;

    public CustomerService(ICustomerRepository customerRepository)
    {
        _customerRepository = customerRepository;
    }

    public Customer GetCustomerById(int id)
    {
        return _customerRepository.GetCustomerById(id);
    }

    public void SaveCustomer(Customer customer)
    {
        if (string.IsNullOrEmpty(customer.Name))
        {
            throw new ArgumentException("Customer name cannot be empty");
        }

        if (string.IsNullOrEmpty(customer.Email))
        {
            throw new ArgumentException("Customer email cannot be empty");
        }

        _customerRepository.SaveCustomer(customer);
    }
}
```

Unit Test with Moq (BLL)

We will write a unit test to verify the `CustomerService` class using Moq to mock the `ICustomerRepository`.

```
using NUnit.Framework;
using Moq;

[TestFixture]
public class CustomerServiceTests
{
    private Mock<ICustomerRepository> _mockCustomerRepository;
    private CustomerService _customerService;
```

```
    [SetUp]
    public void Setup()
    {
        _mockCustomerRepository = new Mock<ICustomerRepository>();
        _customerService = new CustomerService(_mockCustomerRepository.Object);
    }

    [Test]
    public void GetCustomerById_ReturnsCustomer()
    {
        // Arrange
        var customer = new Customer { Id = 1, Name = "John Doe", Email =
"john.doe@example.com" };
        _mockCustomerRepository.Setup(repo =>
repo.GetCustomerById(1)).Returns(customer);

        // Act
        var result = _customerService.GetCustomerById(1);

        // Assert
        Assert.AreEqual(customer, result);
    }

    [Test]
    public void SaveCustomer_WithValidCustomer_CallsSaveCustomerOnRepository()
    {
        // Arrange
        var customer = new Customer { Id = 1, Name = "John Doe", Email =
"john.doe@example.com" };

        // Act
        _customerService.SaveCustomer(customer);

        // Assert
        _mockCustomerRepository.Verify(repo => repo.SaveCustomer(customer), Times.Once);
    }

    [Test]
    public void SaveCustomer_WithEmptyName_ThrowsArgumentException()
    {
        // Arrange
        var customer = new Customer { Id = 1, Name = "", Email = "john.doe@example.com" };

        // Act & Assert
        var ex = Assert.Throws<ArgumentException>(() =>
_customerService.SaveCustomer(customer));
        Assert.AreEqual("Customer name cannot be empty", ex.Message);
    }
```

```
    [Test]
    public void SaveCustomer_WithEmptyEmail_ThrowsArgumentException()
    {
        // Arrange
        var customer = new Customer { Id = 1, Name = "John Doe", Email = "" };

        // Act & Assert
        var ex = Assert.Throws<ArgumentException>(() =>
_customerService.SaveCustomer(customer));
        Assert.AreEqual("Customer email cannot be empty", ex.Message);
    }
}
```

Explanation

1. **Mocking the Dependency**:
    - `Mock<ICustomerRepository>`: Creates a mock object for the `ICustomerRepository` interface.
    - `_mockCustomerRepository.Object`: Provides the mocked instance to the `CustomerService`.
2. **Setup Method (`[SetUp]`)**:
    - Initializes the mock object and the `CustomerService` instance before each test.
3. **Test Methods (`[Test]`)**:
    - **GetCustomerById_ReturnsCustomer**:
        - **Arrange**: Sets up the mock to return a specific `Customer` object when `GetCustomerById(1)` is called.
        - **Act**: Calls `GetCustomerById` on the `CustomerService`.
        - **Assert**: Verifies that the returned customer matches the expected customer.
    - **SaveCustomer_WithValidCustomer_CallsSaveCustomerOnRepository**:
        - **Arrange**: Creates a valid `Customer` object.
        - **Act**: Calls `SaveCustomer` on the `CustomerService`.
        - **Assert**: Verifies that `SaveCustomer` was called exactly once on the `ICustomerRepository`.
    - **SaveCustomer_WithEmptyName_ThrowsArgumentException**:
        - **Arrange**: Creates a `Customer` object with an empty name.
        - **Act & Assert**: Asserts that calling `SaveCustomer` throws an `ArgumentException` with the expected message.
    - **SaveCustomer_WithEmptyEmail_ThrowsArgumentException**:
        - **Arrange**: Creates a `Customer` object with an empty email.

    - **Act & Assert**: Asserts that calling `SaveCustomer` throws an `ArgumentException` with the expected message.

Purpose of the Tests

The purpose of these tests is to ensure that the `CustomerService` class behaves correctly:

- Retrieves customers correctly from the repository.
- Validates customer data before saving.
- Calls the appropriate methods on the repository with the correct data.

By using Moq, we isolate the `CustomerService` class from the actual data access implementation, allowing us to focus on testing its business logic.

This example demonstrates how to use Moq in an N-Tier application to mock dependencies in the Data Access Layer (DAL) and test the Business Logic Layer (BLL).