NUnit and Moq

NUnit-Handson:

1. What is Unit Testing?

Definition: Unit testing involves testing individual components or functions of your code (like a method in a class) in isolation to verify they work correctly.

Focus: Smallest testable part — e.g., a single method.

Dependencies: Often uses mocking to replace dependent classes/objects.

### ****Unit Testing vs Functional Testing****

| **Aspect** | **Unit Testing** | **Functional Testing** |
| --- | --- | --- |
| Scope | Smallest units (methods/functions) | Entire features or modules |
| Isolation | Tested in isolation (often with mocks) | Integrated with other parts of system |
| Speed | Fast | Slower (more complex setups) |
| Tools | NUnit, xUnit, MSTest | Selenium, Postman, REST Assured |

2. Types of Testing

Unit Testing – Testing individual units of code in isolation.

Functional Testing – Testing if the software works as expected from the user’s perspective.

Automated Testing – Any testing done via scripts or frameworks.

Performance Testing – Checks responsiveness, stability under load.

3. Benefits of Automated Testing

Reduces human error.

Speeds up regression testing.

Makes refactoring safer.

Supports CI/CD workflows.

Documents code behavior.

4. Loosely Coupled & Testable Design

Loosely Coupled: Classes should depend on abstractions (interfaces), not concrete classes.

Testable Code: Avoid hard dependencies; use dependency injection so logic can be tested independently.

Bad:

var db = new Database();

Good:

public Calculator(IDatabase db) { \_db = db; }

using NUnit.Framework;

using CalcLibrary;

namespace CalcTests

{

[TestFixture]

public class CalculatorTests

{

private Calculator \_calc;

[SetUp]

public void Setup()

{

\_calc = new Calculator();

}

[TearDown]

public void TearDown()

{

}

[Test]

public void Add\_TwoPositiveNumbers\_ReturnsCorrectSum()

{

int result = \_calc.Add(2, 3);

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

}

[TestCase(1, 2, 3)]

[TestCase(-1, -2, -3)]

[TestCase(0, 0, 0)]

public void Add\_MultipleCases\_ReturnsExpectedResult(int a, int b, int expected)

{

int result = \_calc.Add(a, b);

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

}

[Test, Ignore("This test is under development")]

public void Add\_TestIgnored()

{

Assert.Fail("This should be ignored.");

}

}

}

Moq-Handson:

1. Write Testable Code with Moq

### **Step 1: Set Up Your Test Project**

Create a new **Unit Test Project (.NET Framework)**.

Add **references**:

CustomerCommLib

Install **NUnit** and **Moq** via NuGet:

mathematica

CopyEdit

Install-Package NUnitInstall-Package Moq

Step 2: Create the Unit Test for CustomerComm

using NUnit.Framework;

using Moq;

using CustomerCommLib;

namespace CustomerCommTests

{

[TestFixture]

public class CustomerCommTests

{

private Mock<IMailSender> \_mockMailSender;

private CustomerComm \_customerComm;

[SetUp]

public void Setup()

{

\_mockMailSender = new Mock<IMailSender>();

\_mockMailSender.Setup(x => x.SendMail(It.IsAny<string>(), It.IsAny<string>()))

.Returns(true);

\_customerComm = new CustomerComm(\_mockMailSender.Object);

}

[Test]

public void SendMailToCustomer\_ShouldReturnTrue\_WhenMailIsSentSuccessfully()

{

var result = \_customerComm.SendMailToCustomer();

Assert.That(result, Is.True);

\_mockMailSender.Verify(x => x.SendMail(It.IsAny<string>(), It.IsAny<string>()), Times.Once);

}

}

}