WEEK-2 UNIT TESTING

1. Explain the meaning of Unit Testing and its difference from Functional Testing

**Unit Testing**:  
 Testing individual components/methods (smallest testable parts) in isolation.

Testing Addition(double a, double b) in SimpleCalculator.

**Functional Testing**:  
 Testing the whole system or a use-case scenario to ensure the software behaves as expected.

Not isolated — it may involve UI, database, etc.

Smallest Unit to Test & Mocking Dependencies

* **Smallest Unit**: A **method** is the smallest testable part.  
   Example: Addition(double a, double b)
* **Mocking Dependencies**:  
   When a class depends on others (like database or API), we **mock** them to isolate the logic.  
   Your SimpleCalculator doesn’t need mocks yet (no external dependencies), but using IMathLibrary shows a *testable design*.

2 . List of Various Testing Types

* **Unit Testing** – Test methods/functions (e.g. Addition).
* **Functional Testing** – Check overall functionality.
* **Automated Testing** – Any test run by scripts/tools.
* **Performance Testing** – Tests speed, load, scalability.

3. Benefits of Automated Testing

* Faster feedback
* Reusable test cases
* Easier to detect regressions
* Saves manual effort
* Enhances code quality

4. Explain Loosely Coupled & Testable Design

* **Loosely Coupled**: Classes interact via **interfaces** or **abstractions**, not tightly to other concrete classes.
* **Testable Design**: The logic can be tested independently.

5. Code Not Dependent on Class for Data

public class CalculatorClient

{

private readonly IMathLibrary \_calculator;

public CalculatorClient(IMathLibrary calculator)

{

\_calculator = calculator;

}

public double Add(double a, double b)

{

return \_calculator.Addition(a, b);

}

}

6. Write Your First Test Program for Addition

using NUnit.Framework;

using CalcLibrary;

namespace CalcLibrary.Tests

{

[TestFixture]

public class CalculatorTests

{

private SimpleCalculator calc;

[SetUp]

public void Setup()

{

calc = new SimpleCalculator();

}

[TearDown]

public void TearDown()

{

calc.AllClear();

}

[Test]

[TestCase(2, 3, 5)]

[TestCase(0, 0, 0)]

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

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

{

var result = calc.Addition(a, b);

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

}

[Test]

[Ignore("This test is ignored as an example.")]

public void IgnoredTest()

{

Assert.Fail("This will not run");

}

}

}

### **7.Use of Attributes**

|  |
| --- |
| setup |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Runs **before every test** – useful for object creation/setup   |  | | --- | | TearDown |  |  | | --- | | Runs **after every test** – useful for cleanup |  |  | | --- | | Ignore |  |  | | --- | | Skips the test (temporarily disable) | |

### **8. Explain the Benefit of Writing Parameterized Test Cases**

* Reduces duplication
* Covers multiple input combinations
* Easier to maintain

Create the NUnit Test Cases

using NUnit.Framework;

using CalcLibrary;

namespace CalcLibrary.Tests

{

[TestFixture]

public class CalculatorTests

{

private SimpleCalculator calc;

[SetUp]

public void Setup()

{

calc = new SimpleCalculator();

}

[TearDown]

public void TearDown()

{

calc.AllClear();

}

[Test]

[TestCase(2, 3, 5)]

[TestCase(0, 0, 0)]

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

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

{

var result = calc.Addition(a, b);

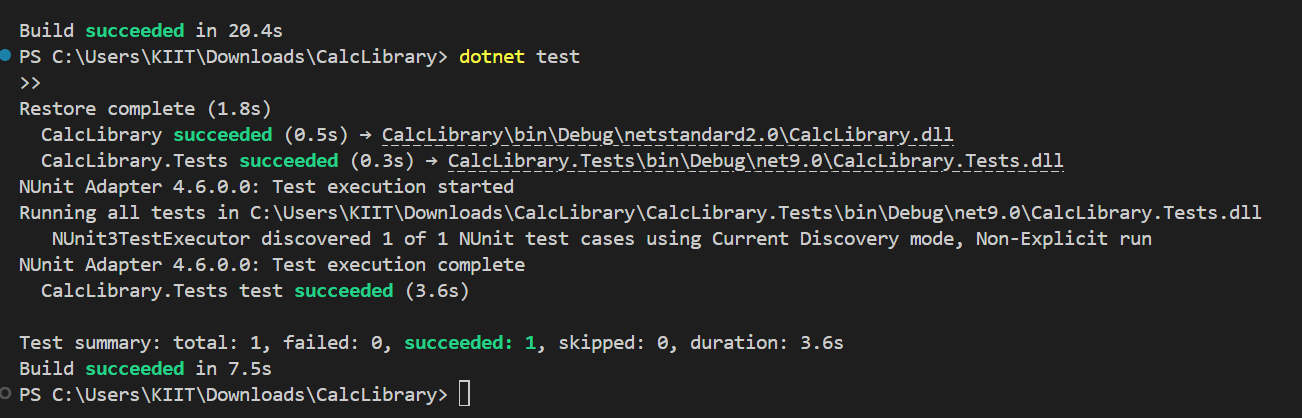
Assert.That(result, Is.EqualTo(expected));

}

}

}

OUTPUT



Write Testable Code with Moq

## **Task1**

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](mailto: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;

}

}

}

namespace CustomerCommLib

{

public class CustomerComm

{

private IMailSender \_mailSender;

public CustomerComm(IMailSender mailSender)

{

\_mailSender = mailSender;

}

public bool SendMailToCustomer()

{

// Here, we pass hardcoded values just for the example.

[\_mailSender.SendMail("cust123@abc.com](mailto:_mailSender.SendMail("cust123@abc.com)", "Some Message");

return true;

}

}

}

TASK 2

using NUnit.Framework;

using Moq;

using CustomerCommLib;

namespace CustomerComm.Tests

{

[TestFixture]

public class CustomerCommTests

{

private Mock<IMailSender> \_mockMailSender;

private CustomerComm \_customerComm;

[OneTimeSetUp]

public void Init()

{

// Arrange: Create mock of IMailSender

\_mockMailSender = new Mock<IMailSender>();

// Setup: Any string arguments => return true

\_mockMailSender

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

.Returns(true);

// Inject the mock into CustomerComm

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

}

[Test]

public void SendMailToCustomer\_ShouldReturnTrue\_WhenMailIsSent()

{

// Act

bool result = \_customerComm.SendMailToCustomer();

// Assert

Assert.IsTrue(result);

}

}

}

