**1. What is Unit Testing and how is it different from Functional Testing?**

* **Unit Testing** tests the **smallest piece of code** (like a method or class) in isolation. It often **mocks dependencies** to isolate behaviours.
* **Functional Testing** tests the **end-to-end behaviour** of a system or module to ensure it meets business requirements.

**2. What is the smallest unit to test and how do mocks help?**

* The smallest unit is a **method or function**.
* **Mocking** simulates external dependencies (like databases or APIs), allowing isolated testing of only the code under test.

**3. What are the various types of testing?**

* **Unit Testing** – Tests individual functions/methods.
* **Functional Testing** – Validates application features against requirements.
* **Automated Testing** – Runs tests automatically using tools/scripts.
* **Performance Testing** – Measures speed, scalability, and stability under load.

**4. What are the benefits of Automated Testing?**

* Saves time with **repeatable and fast** test execution.
* Increases **reliability** and reduces human error.
* Enables **continuous integration (CI/CD)** pipelines.
* Provides quick feedback during development.

**5. What is a Loosely Coupled & Testable Design?**

* A **loosely coupled design** minimizes dependency between components (e.g., using interfaces).
* It’s **testable** because classes can be tested **independently** using mocks or stubs.

Example:

public interface IDataProvider {

int GetData();

}

public class Calculator {

private readonly IDataProvider \_provider;

public Calculator(IDataProvider provider) {

\_provider = provider;

}

public int AddToData(int value) => \_provider.GetData() + value;

}

This code does **not depend directly** on any concrete data class.

**6. Write your first testing program to validate Calculator Add method using NUnit**

using NUnit.Framework;

[TestFixture]

public class CalculatorTests

{

private Calculator \_calculator;

[SetUp]

public void Setup() => \_calculator = new Calculator();

[Test]

public void Add\_TwoNumbers\_ReturnsSum()

{

Assert.That(\_calculator.Add(3, 2), Is.EqualTo(5));

}

}

**7. What are [SetUp], [TearDown], and [Ignore] attributes in NUnit?**

* **[SetUp]** – Runs **before each test** to initialize resources.
* **[TearDown]** – Runs **after each test** to clean up resources.
* **[Ignore]** – Skips a test **temporarily**, useful when a test is broken or not ready.

**8. What are the benefits of writing parameterized test cases?**

* Avoids **code duplication** by running the same test with different inputs.
* Ensures better **coverage** and fewer errors.

🧪 Exampleusing TestCase:

[TestCase(1, 2, 3)]

[TestCase(-1, 5, 4)]

[TestCase(0, 0, 0)]

public void Add\_WithVariousInputs\_ReturnsCorrectResult(int a, int b, int expected)

{

Assert.That(\_calculator.Add(a, b), Is.EqualTo(expected));

}

using System;

public class Calculator

{

public int Add(int a, int b) => a + b;

}

public class Program

{

public static void Main()

{

Calculator calc = new Calculator();

TestAdd(calc, 5, 10, 15);

TestAdd(calc, -2, 3, 1);

TestAdd(calc, 0, 0, 0);

}

static void TestAdd(Calculator calc, int a, int b, int expected)

{

int result = calc.Add(a, b);

if (result == expected)

{

Console.WriteLine($"PASS: Add({a}, {b}) = {result}");

}

else

{

Console.WriteLine($"FAIL: Add({a}, {b}) = {result}, expected {expected}");

}

}

}



