**Nunit Handson**

**Superset id- 6363535**

### **Explain the meaning of Unit Testing and its difference on comparison with Functional Testing**

Unit Testing refers to testing the smallest testable part of an application, typically individual methods or functions, in isolation. The goal is to ensure each unit behaves correctly.

Difference from Functional Testing:

| Criteria | Unit Testing | Functional Testing |
| --- | --- | --- |
| Scope | Smallest part – method/function | Whole system or module |
| Dependency | Mock dependencies (isolated) | Real dependencies (e.g., DB, APIs) |
| Performed by | Developers | QA/testers |
| Example | Testing Add(a, b) method | Testing login functionality as a whole |

### **Smallest unit to test & mocking dependencies**

The smallest unit is usually a method or function. To test this in isolation, we use mocking to replace dependencies (like databases or external services) so the test remains focused on the unit itself.

**List various types of testing**

* Unit Testing – Tests individual components.
* Functional Testing – Validates the application against requirements.
* Automated Testing – Uses scripts/tools to run tests automatically.
* Performance Testing – Measures speed, scalability, and stability.

### **Understand the benefit of Automated Testing**

* Faster execution than manual testing.
* Repeatability – Tests can run any time during development.
* Early bug detection, reducing cost.
* Continuous Integration (CI) support – Run tests automatically on code changes.

### **Explain what is Loosely Coupled & Testable Design**

A loosely coupled design means components do not depend heavily on each other. They interact via interfaces or abstractions.

Benefits:

* Easier to mock dependencies in tests.
* Easier to maintain and reuse code.

Example:  
 Instead of: Calculator calc = new Calculator(); // tightly coupled

Use: IMathLibrary calc = new SimpleCalculator(); // loosely coupled

### **Understand the need of [SetUp], [TearDown] & [Ignore] attributes**

* [SetUp] – Runs before each test. Used to initialize objects.
* [TearDown] – Runs after each test. Used for cleanup.
* [Ignore] – Skips a test (e.g., when not implemented or temporarily disabled).

### **Explain the benefit of writing Parameterized Test Cases**

Parameterized tests reduce duplication and test various inputs with one method using the [TestCase] attribute.

Benefits:

* Test multiple scenarios quickly.
* Easier to maintain and expand test coverage.

**Nunit Handson:**

* Create a Unit test project(.Net Framework) in the solution provided.
* Add the CalcLibrary project as reference
* Create a class “CalculatorTests” to write all the test cases for the methods in the solution
* Use the ‘TestFixture’, ‘SetUp’ and ‘TearDown’ attributes, to declare, initialize and cleanup activities respectively
* Create a Test method to check the addition functionality
* Use the ‘TestCase’ attribute to send the inputs and the expected result
* Use Assert.That to check the actual and expected result match

[**SimpleCalculatorTests.cs**](http://simplecalculatortests.cs)

**Code**

using CalcLibrary;

using NUnit.Framework;

using System;

namespace CalcLibrary.Tests

{

[TestFixture]

public class SimpleCalculatorTests

{

SimpleCalculator calc;

[SetUp]

public void Setup()

{

calc = new SimpleCalculator();

}

[TearDown]

public void TearDown()

{

calc = null;

}

[Test]

[TestCase(3.5, 4.5, 8.0)]

[TestCase(-2.0, 2.0, 0.0)]

public void Addition\_ReturnsCorrectResult(double a, double b, double expected)

{

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

Assert.That(result, Is.EqualTo(expected).Within(0.0001));

}

[Test]

[TestCase(5.5, 2.0, 3.5)]

[TestCase(10.0, 10.0, 0.0)]

public void Subtraction\_ReturnsCorrectResult(double a, double b, double expected)

{

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

Assert.That(result, Is.EqualTo(expected).Within(0.0001));

}

[Test]

[TestCase(2.0, 3.0, 6.0)]

[TestCase(-1.0, 5.0, -5.0)]

public void Multiplication\_ReturnsCorrectResult(double a, double b, double expected)

{

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

Assert.That(result, Is.EqualTo(expected).Within(0.0001));

}

[Test]

[TestCase(10.0, 2.0, 5.0)]

[TestCase(9.0, 3.0, 3.0)]

public void Division\_ReturnsCorrectResult(double a, double b, double expected)

{

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

Assert.That(result, Is.EqualTo(expected).Within(0.0001));

}

[Test]

public void Division\_ByZero\_ThrowsArgumentException()

{

Assert.Throws<ArgumentException>(() => calc.Division(5.0, 0.0));

}

}

}

[**MathLibrary.cs**](http://mathlibrary.cs)

**Code**

using System;

namespace CalcLibrary

{

interface IMathLibrary

{

double Addition(double a, double b);

double Subtraction(double a, double b);

double Multiplication(double a, double b);

double Division(double a, double b);

}

public class SimpleCalculator : IMathLibrary

{

double result = 0;

public double Addition(double a, double b)

{

result = a + b;

return result;

}

public double Subtraction(double a, double b)

{

result = a - b;

return result;

}

public double Multiplication(double a, double b)

{

result = a \* b;

return result;

}

public double Division(double a, double b)

{

if (b == 0)

throw new ArgumentException("Second Parameter Can't be Zero");

result = a / b;

return result;

}

public void AllClear()

{

result = 0;

}

public double GetResult

{

get { return result; }

}

}

}

