### 1. ****What is Unit Testing?****

* Unit testing involves testing the **smallest testable part** of an application — typically a **function, method, or class** — in isolation.
* It uses techniques like **mocking** to isolate the unit and **simulate dependencies** like databases, APIs, or services.
* **Example**: Testing a Calculator.Add(a, b) method without involving UI or DB.

**Functional Testing**:

* Functional testing verifies the **end-to-end behavior** of the system against the business requirements.
* It typically doesn't mock dependencies and may involve interaction between components.
* **Example**: Testing whether clicking a "Calculate" button in the UI returns the correct

#### Difference from Functional Testing:

| **Unit Testing** | **Functional Testing** |  |
| --- | --- | --- |
| Tests **individual methods** | Tests **complete features/workflows** |  |
| Done by **developers** | Done by **QA/testers** |  |
| Usually uses **mocked data** | Uses **real or simulated environments** |  |
| Fast and isolated | Slower and more integrated |  |

**2. Smallest Unit & Mocking Dependencies**

* The smallest unit is a method or function.
* Mocking means replacing real dependencies (e.g., database, APIs) with fake/stub versions so we only test the logic in one class/method.

**Example:**

If your CalculatorService uses Logger, you can mock Logger so you're not testing it along with your calculator logic.

1. **Types of Testing**

| **Type of Testing** | **Purpose** |
| --- | --- |
| **Unit Testing** | Test individual units in isolation |
| **Functional Testing** | Validate that the system works as expected |
| **Automated Testing** | Uses code/scripts to run tests automatically |
| **Performance Testing** | Tests how the system performs under load |

### ****Benefits of Automated Testing****

* **Repeatability**: Run tests frequently and consistently.
* **Speed**: Faster than manual testing.
* **Reliability**: Reduces human error.
* **Coverage**: Helps test more scenarios in less time.
* **Cost-effective**: Saves effort and cost in long-term maintenance.
* **CI/CD Support**: Essential for DevOps and continuous delivery pipelines.

**5. Loosely Coupled & Testable Design**

**✅ Loose Coupling:**

* Components are independent and can work without depending tightly on other components.
* Allows for easier mocking and reusability.

**✅ Testable Code:**

* Easy to test in isolation
* Uses interfaces or dependency injection
* Avoids hard-coded dependencies

**BAD:**

var db = new RealDatabase(); // tightly coupled

**GOOD:**

public class MyService(IDatabase db) { } // loosely coupled via interface

1. ****Understanding:** [SetUp]**,** [TearDown]**,** [Ignore]**

| **Attribute** | **Purpose** |
| --- | --- |
| [SetUp] | Runs **before each test** (e.g., initialize calculator) |
| [TearDown] | Runs **after each test** (e.g., cleanup memory) |
| [Ignore] | Skips the test temporarily (e.g., not ready to test yet) |

1. ****Benefits of Parameterized Tests:****

* Covers **multiple scenarios** in one test method
* Cleaner and less repetitive code
* Easy to **extend** with new test input

**CODE:**

using NUnit.Framework;

using CalcLibrary;

using System;

namespace CalculatorTests

{

[TestFixture]

public class CalculatorTests

{

private SimpleCalculator \_calculator;

[SetUp]

public void Setup()

{

\_calculator = new SimpleCalculator();

}

[TearDown]

public void Teardown()

{

\_calculator.AllClear();

}

[TestCase(5, 3, 8)]

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

[TestCase(0, 0, 0)]

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

{

double result = \_calculator.Addition(a, b);

Assert.AreEqual(expected, result);

}

[TestCase(5, 3, 2)]

[TestCase(-5, -2, -3)]

[TestCase(0, 0, 0)]

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

{

double result = \_calculator.Subtraction(a, b);

Assert.AreEqual(expected, result);

}

[TestCase(5, 3, 15)]

[TestCase(-2, -4, 8)]

[TestCase(2.5, 4, 10)]

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

{

double result = \_calculator.Multiplication(a, b);

Assert.AreEqual(expected, result);

}

[TestCase(10, 2, 5)]

[TestCase(9, 3, 3)]

[TestCase(5.5, 2.2, 2.5)]

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

{

double result = \_calculator.Division(a, b);

Assert.AreEqual(expected, result, 0.0001); // Tolerance for floating-point comparison

}

[Test]

public void Division\_ByZero\_ShouldThrowArgumentException()

{

var ex = Assert.Throws<ArgumentException>(() => \_calculator.Division(10, 0));

Assert.That(ex.Message, Is.EqualTo("Second Parameter Can't be Zero"));

}

[Test]

public void GetResult\_ShouldReturnLatestResult()

{

\_calculator.Addition(5, 5);

Assert.AreEqual(10, \_calculator.GetResult);

}

[Test]

public void AllClear\_ShouldResetResultToZero()

{

\_calculator.Addition(2, 3);

\_calculator.AllClear();

Assert.AreEqual(0, \_calculator.GetResult);

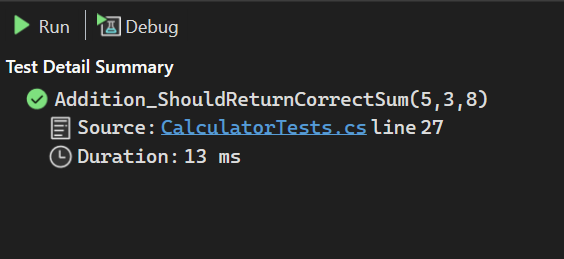
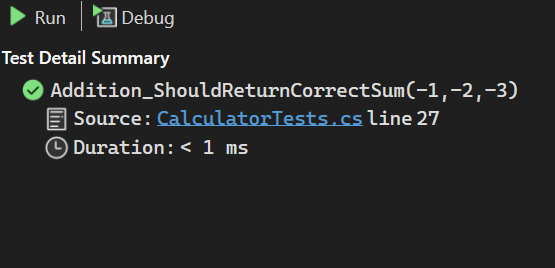
}

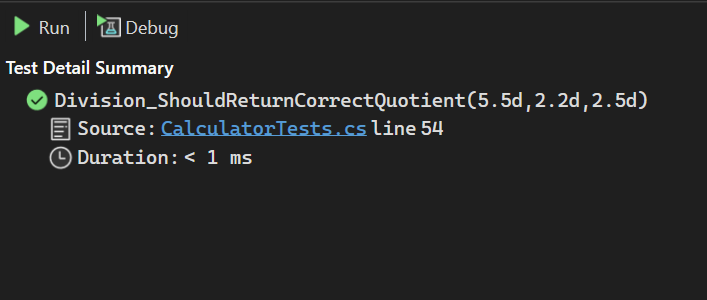
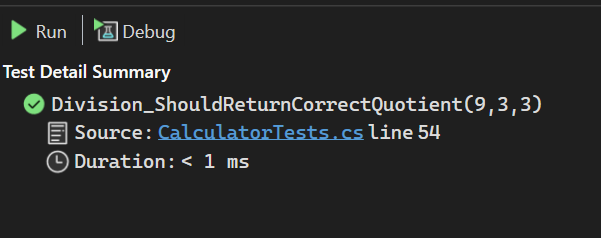
}

}

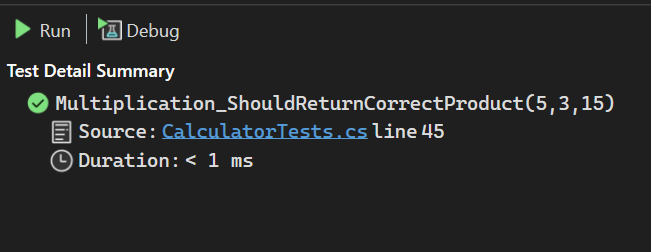
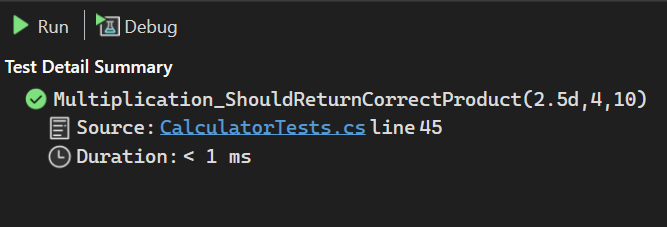
OUTPUT:

**ADDITION**



**DIVISION**

**MULTIPLICATION**



**SUBTRACTION**

