**Q1. EXPLAIN THE MEANING OF UNIT TESTING AND ITS DIFFERENCE ON COMPARISON WITH FUNCTIONAL TESTING**

1. ****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.

1. **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

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Unit Testing** | **Functional Testing** |
| Focus | Single unit (method/class) | System as a whole |
| Dependencies | Mocked/Isolated | Real components |
| Scope | Narrow | Broad |
| Tools (examples) | NUnit, xUnit, JUnit | Selenium, Postman, QTP |

**Q2. LIST VARIOUS TYPES OF TESTING**

· **Unit Testing** – Tests individual methods or components.

· **Functional Testing** – Ensures the software meets functional requirements.

· **Automated Testing** – Uses scripts/tools to automatically run tests (can be unit or functional).

· **Performance Testing** – Checks how the system performs under load, stress, or scalability scenarios.

**Q3. UNDERSTAND THE BENEFIT OF AUTOMATED TESTING**

### ****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.

**Q4. EXPLAIN WHAT IS LOOSELY COUPLED & TESTABLE DESIGN**

**Loosely Coupled Design**:

* Classes/components are **independent** and interact via **interfaces or abstractions**, not directly relying on specific implementations.
* Makes code **easier to test, maintain, and extend**.

**Testable Design**:

* Code written with **testability in mind** allows for easy injection of test doubles (mocks/stubs).

**CODE:**

using CalcLibrary;

namespace StudentGrades.nUnitTests

{

[TestFixture]

public class GradeCalculatorTests

{

private SimpleCalculator calculator;

[SetUp]

public void Setup()

{

calculator = new SimpleCalculator();

}

public void TearDown()

{

calculator = null;

}

[TestCase(90, 14,104)]

[TestCase(90, 54, 144)]

[TestCase(50, 30, 90)]

[TestCase(10, 30, 20)]

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

{

var actual = calculator.Addition(a, b);

Assert.AreEqual(expected, actual);

}

[TestCase(50,30,90)]

[TestCase(10, 30, 90)]

[TestCase(120, 1, 121)]

[TestCase(50, 80, 130)]

public void Addition\_NotEqualTest(double a, double b, double Notexpected)

{

var actual = calculator.Addition(a, b);

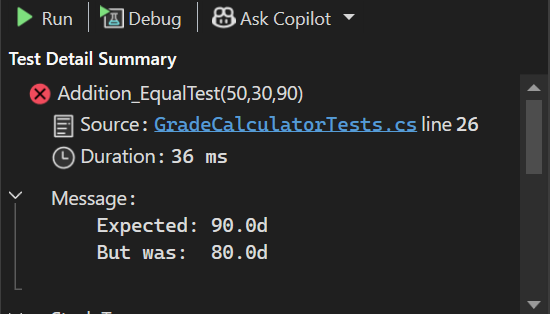
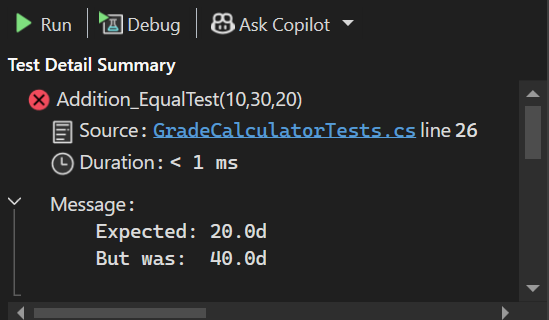
Assert.AreNotEqual(Notexpected, actual);

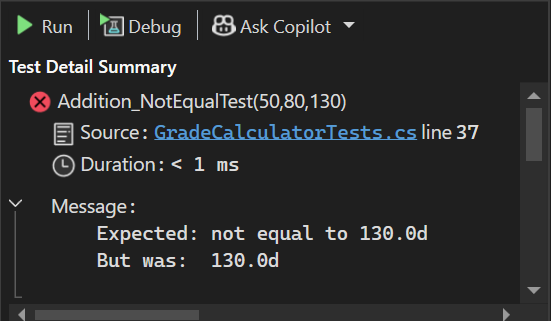
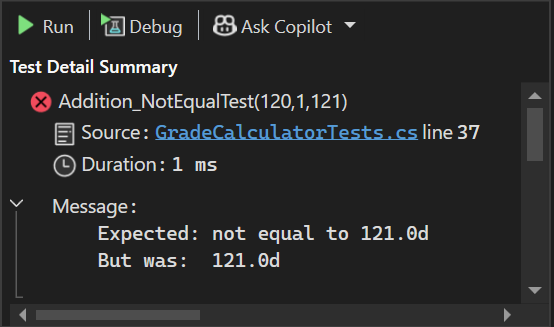
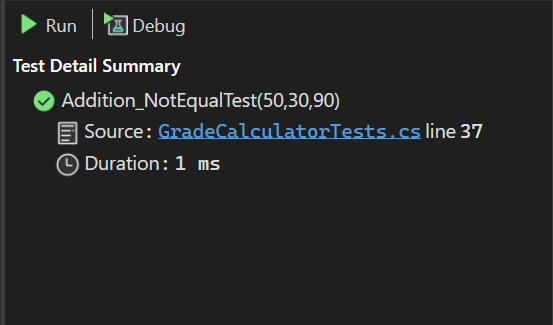
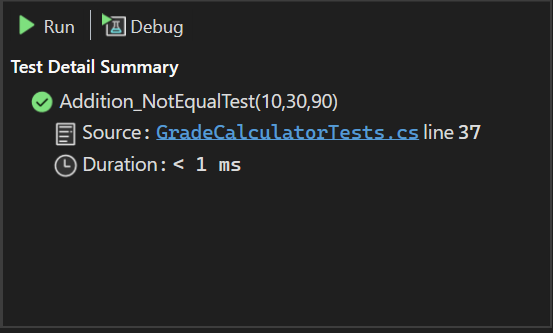
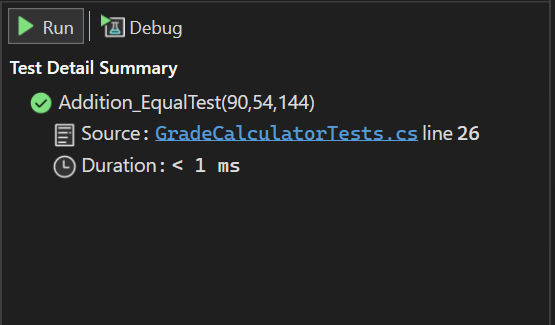
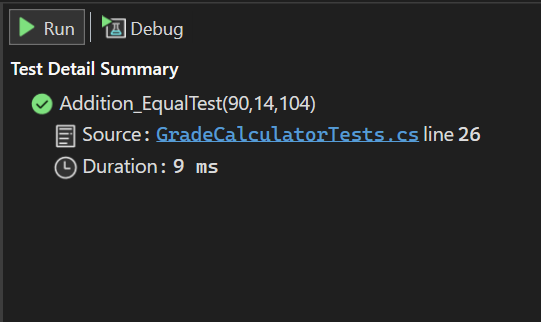
}

}

}

**OUTPUT:**

****

****