**Unit Testing: Objectives, Concepts, and Examples**

**1. Unit Testing vs Functional Testing**

* **Unit Testing**
  + Tests the **smallest parts of an application** (e.g. methods or functions) in isolation.
  + Written primarily by developers using unit testing frameworks such as **NUnit** or **xUnit**.
  + Fast, precise, and code-focused.
* **Functional Testing**
  + Tests the **entire system or specific features** against business requirements.
  + Focuses on user interactions, workflows, and feature correctness.
* **Key Differences**

| **Aspect** | **Unit Testing** | **Functional Testing** |
| --- | --- | --- |
| Scope | Smallest units (methods, classes) | Entire features or workflows |
| Speed | Fast | Slower |
| Focus | Code logic and correctness | Business logic and user scenarios |
| Who Writes | Developers | Testers / QA team |

**2. Types of Testing**

* **Unit Testing:** Testing individual units or components in isolation.
* **Functional Testing:** Validating that features work as intended for the user.
* **Automated Testing:** Using scripts/tools to run tests automatically.
* **Performance Testing:** Evaluating how the system performs under expected or peak loads.

**3. Benefits of Automated Testing**

* Provides **quick feedback** during development.
* Detects bugs **early** in the development cycle.
* Supports **Continuous Integration (CI)** pipelines efficiently.
* Saves time compared to repetitive manual testing.

**4. Loosely Coupled and Testable Design**

* **Loosely Coupled Design**
  + Separates concerns to make components **independent and easier to test**.
  + Promotes dependency injection for flexibility.
* **Example**

public interface IDataService

{

string GetData();

}

public class BusinessLogic

{

private readonly IDataService \_service;

public BusinessLogic(IDataService service)

{

\_service = service;

}

public string Process() => \_service.GetData();

}

**5. Writing a Unit Test: Calculator Addition Example (NUnit)**

* **Test Code**

csharp

CopyEdit

[TestFixture]

public class CalculatorTests

{

[Test]

public void Add\_TwoNumbers\_ReturnsSum()

{

var calc = new Calculator();

var result = calc.Add(2, 3);

Assert.AreEqual(5, result);

}

}

**6. [SetUp], [TearDown], and [Ignore] Attributes**

* **[SetUp]:** Runs before each test method (e.g. initialize test objects).
* **[TearDown]:** Runs after each test method (e.g. clean up resources).
* **[Ignore]:** Skips a test (e.g. temporarily disable failing tests).
* **Example**

[TestFixture]

public class SampleTests

{

private Calculator calc;

[SetUp]

public void Init() => calc = new Calculator();

[TearDown]

public void Cleanup() => calc = null;

[Test]

[Ignore("This test is temporarily disabled")]

public void TestToIgnore() { }

}

**7. Parameterized Tests with [TestCase]**

* **Purpose:** Run the same test logic with multiple input values.
* **Example**

[TestFixture]

public class CalculatorTests

{

[TestCase(2, 3, 5)]

[TestCase(0, 0, 0)]

[TestCase(-1, 1, 0)]

public void Add\_TestCases(int a, int b, int expected)

{

var calc = new Calculator();

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

Assert.AreEqual(expected, result);

}

}