Minutes of Session: Unit Testing & Test-Driven Development (TDD)

Date: 28th July 2025
Topic: Testing frameworks and TDD approach
Duration: Full Day

Participants : Batch C1 = C3
Focus Areas:

Unit Testing & Test-Driven Development	
Introduction to Unit Testing	What is unit testing? (Isolation testing of software modules)
	Benefits of unit testing Bugs detection, Code behavior, code Refactoring, Overall helps in improving quality of code. Ex public int Subtract(int a, int b) => a - b; Unit test can verify if Subtract(5,3) returns 2.
	Basic concepts and terminology **Test Case:** A Single scenario to validate piece or functionality **Test Suite: A Collection of test cases** **Test Fixure : Setup code executed before or after tests.(ex Boilerplate)** **Test Runner: A tool that runs test cases.**
	Testing Frameworks : **Nunit** - Open source and very popular for .NET **MSTest** - Microsoft's own testing framework **xUnit** - Modern unit test framework for .NET Core.
Introduction to Testing Frameworks	Introduction to popular testing frameworks **(NUnit, MSTest, XTest)**
	Setting up a testing framework in a project
	Understanding test projects and test classes
Writing Test Cases **(Arrange: setup test data** **Act: call the method under test** **Assert: Verify the output)**	Anatomy of a unit test
	Creating test methods
	Organizing test classes and test suites
Assertions and Test Data	Using assertions to validate test results
	Common assertion methods (e.g., Assert.AreEqual, Assert.IsTrue)
	Test data setup and teardown
Testing Techniques	Test-driven development (TDD)
	Testing different scenarios (positive, negative, edge cases)
	Mocking dependencies using frameworks like Moq

Test Execution and Reporting	Running tests using the testing framework
	Analyzing test results and understanding test reports
	Handling failures and debugging failing tests

	NUNit	MS Test
Open Source	Excellent	Moderate
Attribute Syntax	[Test], [Setup]	[TestMethod],[TestInitilialize]
INtegration with VS	Requires Adapter(Nuget)	Native to Visual Studio
Popularity	Widely used in open source	Popular in Microsoft Proejcts

1. Introduction to Unit Testing

Definition

Unit testing is a software testing technique that isolates individual components (units) of a program to verify they function as intended.

Benefits

- Detects bugs early in development
- Validates code behavior
- Simplifies code refactoring
- Improves maintainability and reliability

Use Case

```
public int Subtract(int a, int b) => a - b;
// Unit test: Assert.AreEqual(2, Subtract(5, 3));
```

---

2. Basic Concepts and Terminology

Term	Description
Test Case	A single test scenario for a function/module
Test Suite	A collection of related test cases
Test Fixture	Setup/teardown routines before or after test cases
Test Runner	Tool to execute tests and report results

Use Case

Group tests for calculator operations (Add, Subtract, Multiply, Divide) in a suite to validate behavior together.

---

3. Testing Frameworks

Framework	Description	Syntax	Integration
NUnit	Open-source, popular in .NET ecosystem	`[Test]`, `[SetUp]`	Requires NuGet adapter
MSTest	Native to Visual Studio	`[TestMethod]`, `[TestInitialize]`	Built-in
xUnit	Modern, used with .NET Core	`[Fact]`, `[Theory]`	Needs NuGet

---

4. Setting Up a Testing Framework

Steps

1. Create a Unit Test Project in Visual Studio
2. Add reference to the main project
3. Install a test framework via NuGet

4. Create test classes with annotated test methods

---

5. Writing Test Cases

Structure

- **Arrange**: Prepare inputs or dependencies
- **Act**: Call the method under test
- **Assert**: Validate the result/output

Example

```
[Test]
public void Add_ShouldReturnCorrectSum()
{
    int result = calculator.Add(2, 3);
    Assert.AreEqual(5, result);
}
```

---

6. Organizing Test Classes & Suites

Best Practices

- Group tests by class/module
- Use descriptive test method names
- Use `[SetUp]` or `[TestInitialize]` for common test prep

Use Case

```
[TestClass]
```

```
public class EmailValidatorTests
{
    [TestMethod]
    public void IsEmailValid_ShouldReturnFalseForInvalidEmail()
    {
        // logic
    }
}
```

---

7. Assertions and Test Data

Definition

Assertions check whether actual output equals expected output.

Common Assertions

- `Assert.AreEqual(expected, actual)`
- `Assert.IsTrue(condition)`
- `Assert.IsNull(object)`

Use Case

```
Assert.AreEqual(5, Divide(10, 2));
```

---

8. Setup and Teardown

Definition

Used to initialize or cleanup shared objects for tests.

- NUnit: `[SetUp]`, `[TearDown]`

- MSTest: `[TestInitialize]`, `[TestCleanup]`

Use Case

Open DB connection in `[SetUp]`, close in `[TearDown]`.

---

9. Testing Techniques

TDD (Test-Driven Development)

Write test cases before writing the functional code.

Scenario Testing

Write positive, negative, and edge case tests.

Mocking

Use libraries like Moq to simulate dependencies.

Use Case

Use Moq to simulate `IUserRepository` when testing `UserService`.

---

10. Test Execution & Reporting

Execution

- Use Visual Studio Test Explorer
- Use CLI (`dotnet test`) for automation

Best Practices

- Run tests on every code change

- Use CI/CD pipelines
- Analyze coverage reports and failures

Demo Summary

Activity	Tool/Framework Used
Creating a test project	NUnit / MSTest
Writing test cases for calculator	NUnit
Mocking repository using Moq	Moq + NUnit
Running tests and viewing results	Test Explorer

Scenario-Based FAQs – Interview & Viva

Q1. What if your unit test fails in CI/CD but passes locally?
A: Possible reasons include environment differences, missing test data, or flaky tests. Use consistent configurations and isolate test dependencies.

Q2. How would you test a method that calls an external API?
A: Use mocking to simulate the API response (using Moq or a fake service).

Q3. Why should you avoid testing private methods directly?
A: Private methods should be tested indirectly via public methods to ensure encapsulation.

Q4. What's the difference between `[TestInitialize]` in MSTest and `[SetUp]` in NUnit?
A: Both serve the same purpose—setup logic before each test. The difference lies in the framework syntax.

Q5. How would you write unit tests for async methods?
A: Use `async Task` return type and `await` in test methods.

```
[TestMethod]
public async Task GetUserAsync_ShouldReturnUser() {
   var user = await service.GetUserAsync(1);
   Assert.IsNotNull(user);
}
```

Q6. What is the role of mocking in unit testing?
A: Mocking isolates the system under test from external dependencies like databases or web services, ensuring deterministic behavior.

Q7. How can unit testing help during refactoring?
A: If tests are already in place, you can refactor safely—tests act as a safety net for regressions.

Q8. When is it not ideal to write unit tests?
A: For trivial methods (like property getters/setters) or prototypes not yet intended for production.

User Story (for Practice)

Story Title: Email Validation in Registration Module

As a backend developer,
I want to validate user email input,
So that only valid emails are processed during registration.

Acceptance Criteria:

- Invalid emails like `abc@`, `abc.com`, or `@xyz.com` should be rejected.
- Valid emails like `test@example.com` should be accepted.

Task:

1. Write a method `bool IsEmailValid(string email)` using regex
2. Write unit tests using MSTest and NUnit
   - Positive test for valid emails
   - Negative test for malformed emails
3. Mock user input in the registration service (use Moq if necessary)
4. Run tests and generate report