Day 2: Advanced Testing Concepts & Practical Implementation

This guide provides in-depth coverage of the topics for Day 2 of the Microsoft Fullstack Testing Workshop. We'll explore parameterized testing with NUnit and Test-Driven Development (TDD) with practical examples.

Table of Contents

Parameterised Tests in NUnit

Overview

Parameterized testing allows you to run the same test multiple times with different input data. This reduces code duplication and makes tests more maintainable.

Basic Example

Here's a simple example of a parameterized test:

```
public class CalculatorTests
{
    [Test]
    [TestCase(1, 1, 2)]
    [TestCase(0, 5, 5)]
    [TestCase(-1, 1, 0)]
    [TestCase(10, -5, 5)]
    public void Add_WhenCalled_ReturnsSum(int a, int b, int expected)
    {
        // Arrange
        var calculator = new Calculator();

        // Act
        var result = calculator.Add(a, b);

        // Assert
        Assert.That(result, Is.EqualTo(expected));
    }
}
```

TestCaseSource Attribute

For more complex scenarios, use TestCaseSource:

```
public class UserValidatorTests
{
    private static IEnumerable<TestCaseData> UserTestCases()
    {
        yield return new TestCaseData("john.doe", "Password123!")
            .Returns(true)
            .SetName("Valid username and password");

        yield return new TestCaseData("j", "Password123!")
            .Returns(false)
            .SetName("Username too short");

        yield return new TestCaseData("john.doe", "pass")
            .Returns(false)
            .SetName("Password too short");
    }

    [Test]
    [TestCaseSource(nameof(UserTestCases))]
    public bool ValidateUser(string username, string password)
    {
        var validator = new UserValidator();
        return validator.IsValid(username, password);
    }
}
```

Working with External Data

Example using CSV data:

```
public class ProductTests
{
    public static IEnumerable<TestCaseData> GetProductTestData()
    {
        var csvLines = File.ReadAllLines("testdata.csv").Skip(1); // Skip header
        foreach (var line in csvLines)
        {
            var values = line.Split(',');
            yield return new TestCaseData(
                values[0],                    // Product Name
                decimal.Parse(values[1]),     // Price
                bool.Parse(values[2])         // Expected IsValid result
            ).SetName($"Product_{values[0]}");
        }
    }

    [Test]
    [TestCaseSource(nameof(GetProductTestData))]
    public void ValidateProduct_WithTestData_ReturnsExpectedResult(
        string name, decimal price, bool expectedIsValid)
    {
```

```
        var product = new Product(name, price);
        var validator = new ProductValidator();

        var result = validator.IsValid(product);

        Assert.That(result, Is.EqualTo(expectedIsValid));
    }
}
```

Test-Driven Development

The TDD Cycle

1. Write a failing test (Red)
2. Write minimal code to make the test pass (Green)
3. Refactor the code (Refactor)

Practical TDD Example

Let's develop a simple string calculator using TDD:

Step 1: First Test

```
[TestFixture]
public class StringCalculatorTests
{
    [Test]
    public void Add_EmptyString_ReturnsZero()
    {
        // Arrange
        var calculator = new StringCalculator();

        // Act
        var result = calculator.Add("");

        // Assert
        Assert.That(result, Is.EqualTo(0));
    }
}
```

Step 2: Initial Implementation

```
public class StringCalculator
{
    public int Add(string numbers)
    {
        if (string.IsNullOrEmpty(numbers)) return 0;
        return -1; // Placeholder
```

```
    }
}
```

Step 3: Add More Tests

```
[Test]
public void Add_SingleNumber_ReturnsNumber()
{
    var calculator = new StringCalculator();
    var result = calculator.Add("1");
    Assert.That(result, Is.EqualTo(1));
}

[Test]
public void Add_TwoNumbers_ReturnsSum()
{
    var calculator = new StringCalculator();
    var result = calculator.Add("1,2");
    Assert.That(result, Is.EqualTo(3));
}
```

Step 4: Complete Implementation

```
public class StringCalculator
{
    public int Add(string numbers)
    {
        if (string.IsNullOrEmpty(numbers)) return 0;

        return numbers.Split(',')
            .Select(int.Parse)
            .Sum();
    }
}
```

TDD Best Practices

1. **Keep Tests Simple**
    - One assertion per test
    - Clear naming convention
    - Follow Arrange-Act-Assert pattern
2. **Write Minimal Code**

```
// Bad - Over-implementation
public int Add(string numbers)
{
    if (string.IsNullOrEmpty(numbers)) return 0;
    var result = 0;
    var numberList = new List<int>();
    foreach(var num in numbers.Split(','))
    {
        numberList.Add(int.Parse(num));
    }
    result = numberList.Sum();
    return result;
}

// Good - Minimal implementation
public int Add(string numbers)
{
    return string.IsNullOrEmpty(numbers) ? 0 :
        numbers.Split(',').Sum(n => int.Parse(n));
}
```

3. **Refactoring Phase**
    - Improve code readability
    - Remove duplication
    - Maintain test coverage

Conclusion

This guide covered the essential concepts of parameterized testing and TDD. Remember:

- Use parameterized tests to reduce test code duplication
- Follow the Red-Green-Refactor cycle in TDD
- Keep tests simple and focused
- Refactor continuously while maintaining test coverage

Additional Resources

- NUnit Documentation
- Clean Code by Robert C. Martin
- Test Driven Development by Kent Beck