Day 1: Testing Fundamentals & Microsoft Testing Ecosystem

1. Testing Foundations

What is Software Testing?

Software testing is a systematic process of evaluating a software application to:

- Identify defects and bugs
- Verify that the software meets specified requirements
- Validate that it works as expected for end users
- Ensure quality and reliability

Testing Types

1. Functional Testing

- **Definition**: Verifies that each function operates according to specifications
- **Focus Areas**:
  - Input handling
  - Business logic
  - Output validation
  - Error handling
- **Examples**:

```
// Example of a function to test
public decimal CalculateDiscount(decimal price, int customerType) {
    if (customerType == 1) return price * 0.1m;  // 10% discount
    if (customerType == 2) return price * 0.2m;  // 20% discount
    return 0;
}

// Corresponding test
[Test]
public void CalculateDiscount_PremiumCustomer_Returns20PercentDiscount() {
    var price = 100m;
    var customerType = 2;
    var expected = 20m;

    var actual = CalculateDiscount(price, customerType);

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

2. Non-functional Testing

- **Performance Testing**: Response time, scalability, stability
- **Security Testing**: Vulnerability assessment, penetration testing
- **Usability Testing**: User experience, accessibility
- **Load Testing**: System behavior under various load conditions

Testing Levels

1. **Unit Testing**
    - Tests individual units (functions, components, etc) in isolation
    - Fastest to execute and easiest to debug
    - Example:

```
public class Calculator {
    public int Add(int a, int b) => a + b;
}

[TestFixture]
public class CalculatorTests {
    [Test]
    public void Add_TwoPositiveNumbers_ReturnsSum() {
        // Arrange
        var calculator = new Calculator();

        // Act
        var result = calculator.Add(2, 3);

        // Assert
        Assert.That(result, Is.EqualTo(5));
    }
}
```

2. **Integration Testing**
    - Tests interaction between components
    - Verifies different parts work together
    - Example:

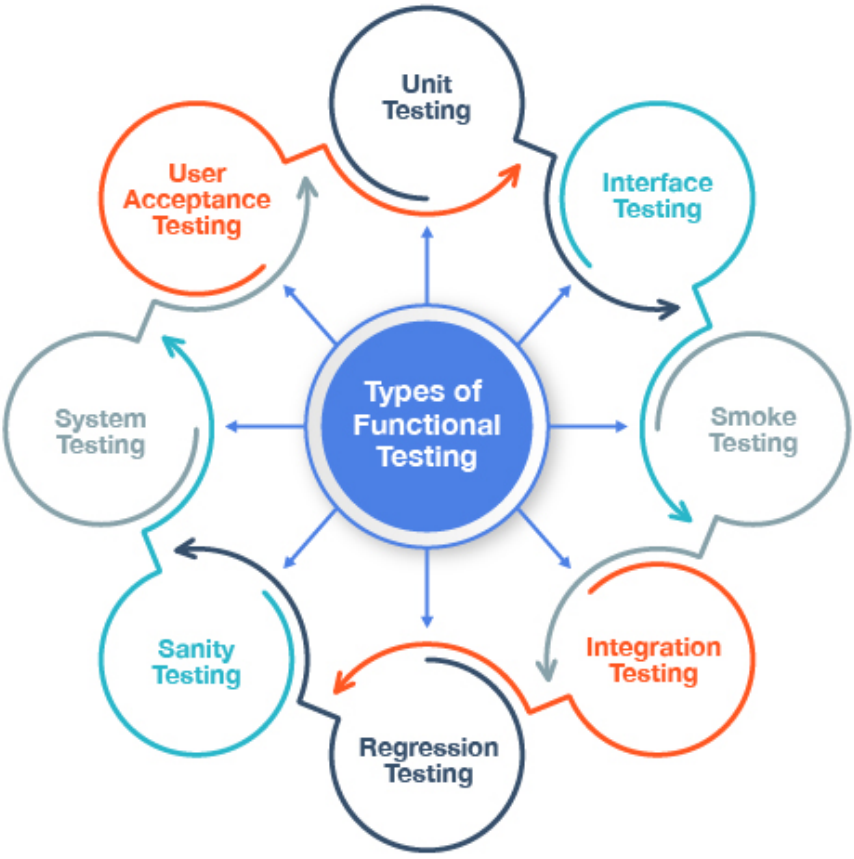
```
public class OrderProcessor {
    // our class depends on the work done by some other developer
    private readonly IInventoryService _inventory;
    private readonly IPaymentService _payment;

    public async Task<bool> ProcessOrder(Order order) {
        // our function should work assuming that the dependencies work
properly
        var inStock = await _inventory.CheckStock(order.ItemId);
        if (!inStock) return false;
```

```
            return await _payment.ProcessPayment(order.Amount);
        }
    }
```

3. **System Testing**
    - Tests the entire application as a whole
    - End-to-end scenarios
    - Example: Complete user journey from login to checkout
4. **Acceptance Testing**
    - Validates software meets business requirements
    - Often performed by end users or product owners

More details on types of testing



The image shows a diagram of different types of functional testing in software development. Here's a breakdown of each type with corresponding .NET tools, libraries, and frameworks:

1. **Unit Testing**:
    - Tests individual components or functions in isolation to verify they work correctly.
    - Developers typically write these tests to check if specific pieces of code perform as expected.
    - .NET tools: MSTest, NUnit, xUnit.net
    - Libraries: Moq, NSubstitute, FakeItEasy, FluentAssertions

2. **Interface Testing**:
    - Focuses on testing the communication between different software components or systems.
    - Ensures that data is correctly passed between interfaces and that components interact properly.
    - .NET tools: WCF Test Client, Swagger/OpenAPI, WireMock.NET
    - Libraries: RestSharp, Refit, WebApplicationFactory
3. **Smoke Testing**:
    - A preliminary test to verify the basic functionality of an application before more thorough testing.
    - Checks if the most critical features work without delving into details.
    - .NET tools: Selenium with C#, SpecFlow
    - Libraries: FluentAutomation, Atata
4. **Integration Testing**:
    - Tests how multiple components work together.
    - Verifies that different modules or services function correctly when combined.
    - .NET tools: xUnit.net (with WebApplicationFactory), NUnit
    - Libraries: TestServer, IntegrationFixture, Testcontainers for .NET
5. **Regression Testing**:
    - Ensures that new code changes don't adversely affect existing functionality.
    - Involves retesting previously validated features after modifications.
    - .NET tools: Azure DevOps Test Plans, BrowserStack with .NET
    - Libraries: Selenium WebDriver for .NET, Playwright for .NET
6. **Sanity Testing**:
    - A subset of regression testing that verifies specific functionality after changes.
    - Focuses on particular areas rather than the entire system.
    - .NET tools: Similar to regression testing tools
    - Libraries: SpecFlow, LightBDD, Fixie
7. **System Testing**:
    - Tests the complete, integrated software system to verify it meets requirements.
    - Evaluates the system's compliance with specified requirements.
    - .NET tools: Azure DevOps Test Plans, Visual Studio Test Professional
    - Libraries: Selenium Grid with .NET, Appium for .NET
8. **User Acceptance Testing**:
    - The final testing phase where actual users test the software.
    - Ensures it meets business requirements and is ready for deployment.
    - .NET tools: TestRail with .NET integration, Microsoft Team Foundation Server
    - Libraries: SpecFlow with Gherkin syntax, Behavior Driven Development (BDD) frameworks

Each of these testing types serves a specific purpose in the software testing lifecycle, helping to identify different types of issues at various development stages. Many of these tools integrate with the .NET ecosystem and can be used in combination with each other to create comprehensive testing strategies.

When to Automate vs Manual Testing

Automate When:

- Tests need to run frequently
- Repetitive tasks
- Performance testing
- Regression testing
- Complex calculations
- Data-driven scenarios

Manual Testing When:

- Exploratory testing
- Usability testing
- Ad-hoc testing
- One-time test cases
- Visual verification needed

2. The SDET Role

Role Overview

Software Development Engineer in Test (SDET) combines:

- Software development skills
- Testing expertise
- Quality assurance knowledge

Key Responsibilities

1. **Test Automation**
    - Design and implement test frameworks
    - Write automated tests
    - Maintain test suites
2. **Quality Processes**
    - Define testing strategies
    - Create test plans
    - Establish quality metrics
3. **Development Support**
    - Code reviews
    - Testing new features

    - Bug verification

Required Skills

Technical Skills

- C# programming
- Testing frameworks (MSTest, NUnit, xUnit)
- Version control (Git)
- CI/CD pipelines
- Database knowledge
- API testing

Soft Skills

- Problem-solving
- Communication
- Team collaboration
- Time management
- Attention to detail

Quality Metrics

Quality metrics are essential measurements that help teams assess the effectiveness of their testing efforts and the overall quality of their software. These metrics provide quantitative data for decision-making and process improvement.

1. Code Coverage

- **Definition**: Measures how much of your code is executed by tests
- **Types of Coverage**:
    - **Line Coverage**: Percentage of code lines executed
    - **Branch Coverage**: Percentage of code branches (if/else) executed
    - **Method Coverage**: Percentage of methods called
    - **Class Coverage**: Percentage of classes instantiated

Industry Standards:

- Minimum acceptable coverage: 70-80%
- Ideal coverage for critical systems: 90%+
- Coverage targets should be higher for:
    - Core business logic
    - Critical system components
    - Public APIs

2. Test Pass Rate

- **Definition**: Percentage of tests that pass out of total tests executed
- **Key Aspects**:

    - **Test Stability**: Consistent results across multiple runs
    - **Flaky Tests**: Tests that sometimes pass and sometimes fail
    - **Historical Trends**: Pass rate over time

Measurement Standards:

- Target pass rate: 95%+ for production deployments
- Acceptable flaky test rate: < 1%
- Historical trend analysis: Rolling 30-day window

3. Bug Detection Rate

- **Definition**: Number of defects found during testing vs. production
- **Metrics to Track**:
    - **Defect Density**: Bugs per 1000 lines of code
    - **Defect Leakage**: Bugs that escaped to production
    - **Defect Resolution Time**: Time to fix identified bugs
    - **Severity Distribution**: Critical vs. Minor bugs ratio

Industry Benchmarks:

- Acceptable defect density: < 1 bug per 1000 LOC
- Critical bug resolution time: < 24 hours
- Defect leakage target: < 5%

4. Automation Coverage

- **Definition**: Extent of test cases automated vs. manual
- **Key Metrics**:
    - **Automation Percentage**: (Automated Tests / Total Tests) × 100
    - **Automation ROI**: Time saved vs. automation development cost
    - **Automation Reliability**: Success rate of automated test runs
    - **Maintenance Cost**: Time spent maintaining automated tests

Target Metrics:

- Automation coverage target: 70%+ for regression tests
- ROI threshold: 3x (benefit vs. cost)
- Automation reliability: 98%+ success rate
- Maintenance overhead: < 20% of automation time

5. Performance Metrics

- **Test Execution Time**: Average time to run test suite
- **Resource Usage**: CPU, memory, network usage during tests
- **Test Parallelization**: Efficiency of parallel test execution
- **Test Environment Stability**: Uptime and reliability

Performance Targets:

- Unit test execution: < 1 second per test
- Full test suite: < 1 hour
- Resource usage: < 75% of available resources
- Environment stability: 99.9% uptime

3. Microsoft Technology Stack for Testing

.NET Testing Tools

1. **NUnit**
    - Popular open-source unit testing framework for .NET
    - Rich set of assertions and constraints
    - Attribute-based test decoration
    - Parameterized tests with TestCase attributes
    - Category-based test organization
    - Parallel test execution support
    - Custom test execution order
    - Extensive plugin ecosystem
2. **Visual Studio Testing Features**
    - Test Explorer
    - Live Unit Testing
    - Code Coverage
    - IntelliTest
3. **Azure DevOps Testing**
    - Test Plans
    - Test Cases
    - Test Suites
    - Automated Test Execution

Deep Dive into NUnit

Key Attributes

1. **Test Organization Attributes**
    - `[TestFixture]`: Marks a class containing tests
    - `[Test]`: Marks a method as a test
    - `[TestCase]`: Provides inline data for parameterized tests
    - `[TestCaseSource]`: Uses external method/property for test data
    - `[Category("CategoryName")]`: Categorizes tests for filtered execution

```
[TestFixture]
public class CalculatorTests {
    [Test]
    [Category("BasicMath")]
```

```
    public void Add_SimpleValues_ReturnsSum() { ... }

    [TestCase(1, 2, 3)]
    [TestCase(-1, -2, -3)]
    [TestCase(0, 0, 0)]
    public void Add_MultipleScenarios_ReturnsExpectedSum(int a, int b, int
expected) {
        var calculator = new Calculator();
        Assert.That(calculator.Add(a, b), Is.EqualTo(expected));
    }
}
```

2. **Setup and Teardown Attributes**
    - `[OneTimeSetUp]`: Runs once before all tests in a class
    - `[OneTimeTearDown]`: Runs once after all tests in a class
    - `[SetUp]`: Runs before each test
    - `[TearDown]`: Runs after each test

```
[TestFixture]
public class DatabaseTests {
    private DbConnection _connection;
    private TestData _testData;

    [OneTimeSetUp]
    public void InitializeTestSuite() {
        // Initialize resources for all tests
        _connection = new DbConnection(ConnectionString);
    }

    [SetUp]
    public void SetupTest() {
        // Setup for each test
        _testData = new TestData();
        _connection.BeginTransaction();
    }

    [TearDown]
    public void CleanupTest() {
        // Cleanup after each test
        _connection.RollbackTransaction();
    }

    [OneTimeTearDown]
    public void CleanupTestSuite() {
        // Cleanup all resources
        _connection.Dispose();
    }
}
```

3. **Constraint Model Assertions** NUnit uses a constraint model for assertions, making them more readable and flexible:

```
[TestFixture]
public class AssertionExamples {
    [Test]
    public void DemonstrateAssertions() {
        // Equality
        Assert.That(result, Is.EqualTo(expected));
        Assert.That(value, Is.Not.EqualTo(wrongValue));

        // Numeric Comparisons
        Assert.That(number, Is.GreaterThan(minimum));
        Assert.That(number, Is.LessThan(maximum));
        Assert.That(number, Is.InRange(1, 10));

        // String Comparisons
        Assert.That(text, Is.EqualTo("expected").IgnoreCase);
        Assert.That(text, Does.Contain("substring"));
        Assert.That(text, Does.StartWith("prefix"));
        Assert.That(text, Does.Match("regex pattern"));

        // Collection Assertions
        Assert.That(list, Is.Empty);
        Assert.That(list, Has.Count.EqualTo(5));
        Assert.That(list, Does.Contain("item"));
        Assert.That(list, Is.Ordered);
        Assert.That(list, Is.Unique);

        // Type Assertions
        Assert.That(obj, Is.InstanceOf<string>());
        Assert.That(obj, Is.Not.Null);

        // Exception Assertions
        Assert.That(() => methodThatThrows(),
            Throws.TypeOf<ArgumentException>()
                .With.Message.Contains("specific message"));

        // Composite Assertions
        Assert.That(number, Is.GreaterThan(0).And.LessThan(100));

        // Tolerance Assertions
        Assert.That(1.1, Is.EqualTo(1.15).Within(0.1));
        Assert.That(DateTime.Now,
Is.EqualTo(expected).Within(TimeSpan.FromSeconds(1)));
    }
}
```

4. **Advanced Attributes**
    - `[Combinatorial]`: Tests all combinations of values
    - `[Sequential]`: Tests values in sequence

- `[Random]`: Generates random test values
- `[Retry]`: Retries failed tests
- `[Timeout]`: Sets timeout for test execution
- `[Order]`: Specifies test execution order

```
[TestFixture]
public class AdvancedAttributeExamples {
    [Test]
    [Timeout(2000)] // 2 second timeout
    public async Task LongRunningOperation_CompletesInTime() {
        // Test implementation
    }

    [Test]
    [Retry(3)] // Retry up to 3 times
    public void UnreliableOperation_EventuallySucceeds() {
        // Test implementation
    }

    [Test]
    [Combinatorial]
    public void TestWithMultipleInputs(
        [Values(1, 2, 3)] int x,
        [Values("a", "b")] string y) {
        // Tests all combinations: (1,a), (1,b), (2,a), (2,b), (3,a), (3,b)
    }
}
```

5. **Parallel Test Execution**

```
[TestFixture]
[Parallelizable(ParallelScope.Children)]
public class ParallelTests {
    [Test]
    [Parallelizable]
    public async Task Test1() { ... }

    [Test]
    [Parallelizable]
    public async Task Test2() { ... }
}
```

This concludes the Day 1 material. Practice exercises and hands-on labs will help reinforce these concepts during the workshop.