Unit Test med xUnit og Moq

Programmering - 2. semester

SOLID (D) - Dependency Inversion & Testbarhed

Agenda

1. Hvad er Unit Testing?
2. Test Pyramiden
3. xUnit v3 - Framework
   [Fact], [Theory], Assert
4. Arrange-Act-Assert mønstret
5. Mocking med Moq
   Mock, Setup, Verify
6. Dependency Inversion & testbarhed
7. Adapter Pattern for testbarhed

Hvad er Unit Testing?

> En unit test tester en enkelt "enhed" af kode isoleret fra dens afhængigheder.

En "enhed" er typisk en metode eller en klasse

Testen verificerer at koden opfører sig korrekt

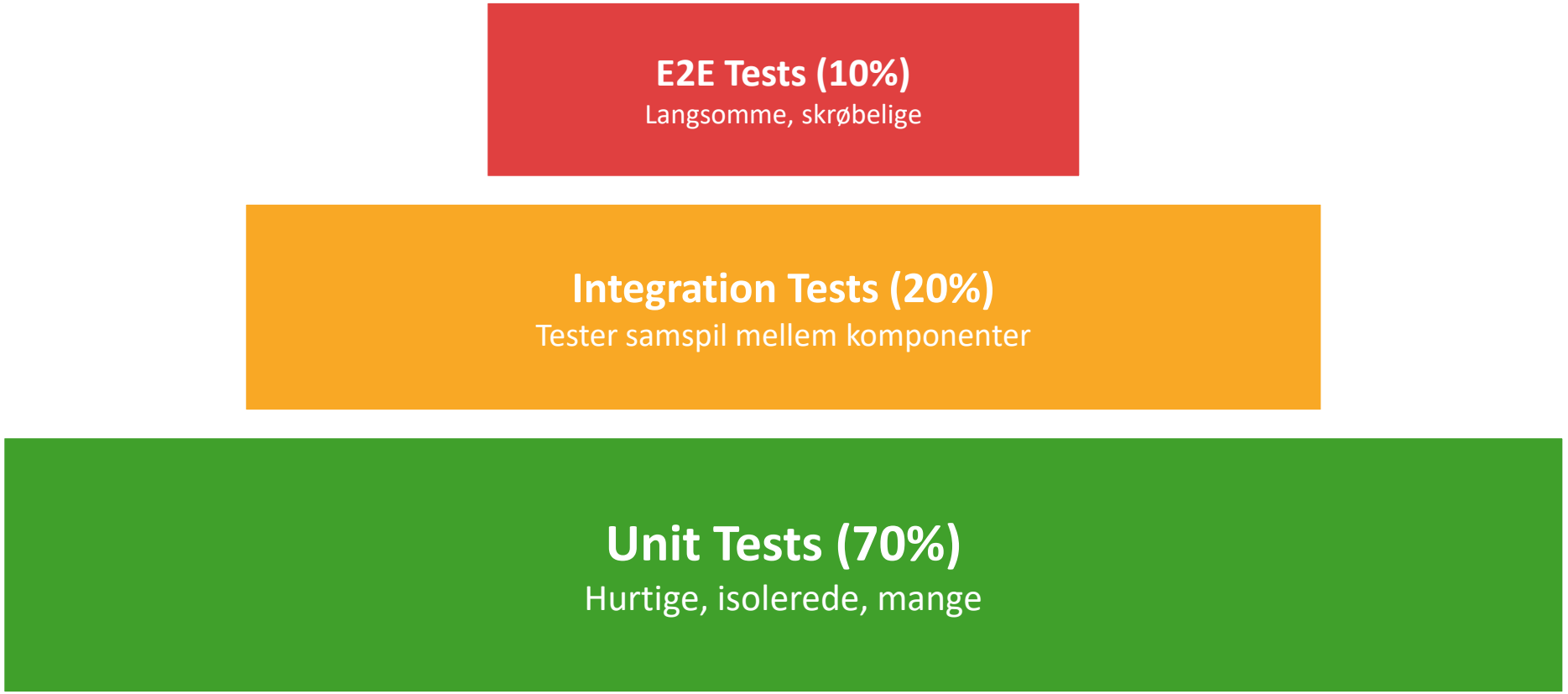
Afhængigheder erstattes med test doubles (mocks/stubs)

Unit tests er hurtige - kører på millisekunder

Giver øjeblikkelig feedback under udvikling

Fanger fejl tidligt i udviklingsprocessen

Test Pyramiden



xUnit v3 - Introduktion

xUnit er et open source test framework til .NET

Bruges sammen med Microsoft.NET.Test.Sdk

Vigtige koncepter:

- [Fact] - en enkelt test case
- [Theory] - parameteriserede tests
- Assert klassen - verificering af resultater

NuGet pakker til test projektet:

```
<!-  .csproj -->
<PackageReference Include="Microsoft.NET.Test.Sdk" Version="18.0.1" />
<PackageReference Include="Moq" Version="4.20.72" />
<PackageReference Include="xunit.v3.mtp-v2" Version="3.2.2" />
```

[Fact] - En enkelt test case

[Fact] markerer en metode som en test

Testen har ingen parametre

Tester ét specifikt scenarie

```
[Fact]
public void Addition_ReturnsCorrectSum()
{
    // Arrange
    var calculator = new Calculator();

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

    // Assert
    Assert.Equal(5, result);
}
```

[Theory] og [InlineData]

[Theory] kører samme test med forskellige data

[InlineData] leverer testdata som parametre

Perfekt til at teste edge cases og grænseværdier

```
[Theory]
[InlineData(2, 3, 5)]
[InlineData(0, 0, 0)]
[InlineData(-1, 1, 0)]
[InlineData(100, -50, 50)]
public void Addition_WithVariousInputs_ReturnsCorrectSum(
    int a, int b, int expected)
{
    // Arrange
    var calculator = new Calculator();

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

    // Assert
    Assert.Equal(expected, result);
}
```

Assert - Vigtige metoder

```
// Sammenligning
Assert.Equal(expected, actual);
Assert.NotEqual(unexpected, actual);

// Boolean
Assert.True(condition);
Assert.False(condition);

// Null check
Assert.Null(object);
Assert.NotNull(object);
```

```
// Collections
Assert.Empty(collection);
Assert.Single(collection);
Assert.Contains(item, collection);

// Exceptions
Assert.Throws<Exception>(
    () ⇒ dangerousMethod()
);

// Type check
Assert.IsType<MyClass>(object);
```

Arrange - Act - Assert (AAA)

Arrange
Opsæt test data
og afhængigheder
(mocks, objekter)

Act
Kald metoden
der testes
(system under test)

Assert
Verificér at
resultatet er
som forventet

```
[Fact]
public void Method_Scenario_ExpectedResult()
{
    // Arrange - opsæt data og afhængigheder
    var mock = new Mock<IDependency>();
    var sut = new MyService(mock.Object);

    // Act - kald metoden
    var result = sut.DoSomething();

    // Assert - verificér resultat
    Assert.Equal(expected, result);
}
```

Test Doubles - Mocking

Test doubles erstatter rigtige afhængigheder med kontrollerede implementeringer

Stub
Returnerer foruddefinerede værdier. Bruges når vi har brug for data.

Mock
Verificerer at specifikke metoder blev kaldt. Interaktions-test.

Fake
Forenklet implementering. Fx in-memory database i stedet for SQL Server.

Vi bruger Moq biblioteket til at oprette mocks i C#

NuGet pakke: Moq version 4.20+

Moq - Grundlæggende brug

```
// 1. Opret en mock af et interface
var mockRepo = new Mock<ILejemaalRepository>();

// 2. Setup - definer hvad mock'en skal returnere
mockRepo.Setup(repo ⇒ repo.HentLejemaal())
        .Returns(new List<Lejemaal> { ... });

// 3. Brug mock'ens Object som den rigtige dependency
var service = new EjendomBeregnerService(mockRepo.Object);

// 4. Kald metoden (Act)
var result = (service as IEjendomBeregnerService).BeregnKvadratmeter();

// 5. Verificér at metoden blev kaldt (valgfrit)
mockRepo.Verify(repo ⇒ repo.HentLejemaal(), Times.Once);
```

Mock<T> opretter en mock-instans af interface T

Setup() definerer returværdi for en metode

.Object giver den mockede instans til brug i koden

Verify() checker at en metode blev kaldt

Moq - Komplet eksempel

```
public class CalculatorServiceTest
{
    [Fact]
    public void Calculate_Returns_Sum_Of_Repository_Values()
    {
        // Arrange
        var data = new List<Lejemaal>
        {
            new Lejemaal { Lejlighednummer = 1, Kvadratmeter = 50.0, AntalRum = 2 },
            new Lejemaal { Lejlighednummer = 2, Kvadratmeter = 75.0, AntalRum = 3 }
        };
        var expected = 125.0;

        var mockRepo = new Mock<ILejemaalRepository>();
        mockRepo.Setup(r ⇒ r.HentLejemaal()).Returns(data);

        var sut = new EjendomBeregnerService(mockRepo.Object)
                      as IEjendomBeregnerService;

        // Act
        var actual = sut.BeregnKvadratmeter();

        // Assert
        Assert.Equal(expected, actual);
    }
}
```

Dependency Inversion & Testbarhed

High-level moduler skal ikke afhænge af low-level moduler.
Begge skal afhænge af abstraktioner (interfaces).

Uden DIP (svær at teste):

```
public class EjendomBeregnerService {
    public double BeregnKvadratmeter() {
        // Direkte afhængighed af filsystemet!
        string[] data =
File.ReadAllLines("data.csv");
        // ... beregning
    }
}
```

Med DIP (testbar):

```
public class EjendomBeregnerService {
    private readonly ILejemaalRepository _repo;

    public EjendomBeregnerService(
        ILejemaalRepository repo) {
        _repo = repo;  // Interface dependency
    }
}
```

Interfaces gør det muligt at erstatte afhængigheder med mocks

Constructor injection giver eksplicitte afhængigheder

Koden bliver løst koblet og nemmere at vedligeholde

Adapter Pattern - Testbar fil-adgang

Problem: File.ReadAllLines() kan ikke mockes direkte

Løsning: Indkapsle fil-adgang bag et interface (Adapter Pattern)

Interface:

```
public interface IFile
{
    string[] ReadAllLines();
}
```

Adapter (produktion):

```
public class FileAdapter : IFile
{
    private readonly string _path;
    public FileAdapter(string path) {
        _path = path;
    }
    string[] IFile.ReadAllLines() {
        return File.ReadAllLines(_path);
    }
}
```

I test - mock af IFile:

```
var mockFile = new Mock<IFile>();
mockFile.Setup(f ⇒ f.ReadAllLines())
        .Returns(new[] { "header", "\"101\"; \"755\"; \"3\"" });
var repo = new LejemaalFraFilRepository(mockFile.Object);
```

Opsummering

Unit tests tester kode isoleret fra afhængigheder

xUnit v3: [Fact] for enkelt-tests, [Theory] for parameteriserede tests

AAA-mønstret: Arrange - Act - Assert

Moq bruges til at oprette mocks af interfaces

- Mock<T>, Setup(), .Object, Verify()

Dependency Inversion Principle muliggør testbarhed

- Afhæng af abstraktioner (interfaces), ikke konkrete klasser

Adapter Pattern indkapsler eksterne afhængigheder (fx filsystem)

Næste skridt: Prøv det selv med EjendomBeregner opgaven!

Links og ressourcer

xUnit: https://xunit.net
Moq: https://github.com/devlooped/moq

Video: Master xUnit Like A Pro in Under 10 Minutes!
Video: xUnit tutorial - How to run unit testing in C#
Video: How to use Moq to mock xUnit tests for a .NET project

Bog: Domain-Centric Architecture with C#
- Kapitel 13: Testing Domain-Centric Applications
- Sektion 14.1: The Testing Pyramid
- Sektion 14.2: Unit Testing the Domain Layer
- Sektion 14.4: Mocking and Test Doubles