

# C# Dependency Injection Testing Entity Framework

Rasmus Lystrøm  
Associate Professor  
ITU  
[rne@itu.dk](mailto:rne@itu.dk)

```
File Edit Selection View Go Debug Terminal Help
ProgramTests.cs x
HelloWorld.Tests > ProgramTests.cs > {} HelloWorld.Tests > HelloWorld
1 using System;
2 using System.IO;
3 using Xunit;
4
5 namespace HelloWorld.Tests
6 {
7     0 references | Run All Tests | Debug All Tests
8     public class ProgramTests
9     {
10         [Fact]
11         0 references | Run Test | Debug Test
12         public void Main_given_no_args_p
13         {
14             // Arrange
15             using var writer = new String
16             Console.SetOut(writer);
17
18             // Act
19             Program.Main(new string[0]);
20
21             // Assert
22             var output = writer.GetStringBuilder().ToString();
23             Assert.Equal("Hello World!", output);
24         }
25     }

```

```
Program.cs x
HelloWorld > Program.cs > ...
1 using System;
2
3 namespace HelloWorld
4 {
5     1 reference
6     public class Program
7     {
8         1 reference
9         public static void Main(string[] args)
10         {
11             Console.WriteLine("Hello World!");
12         }
13     }

```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: pwsh
Loading personal and system profiles took 1008ms.
C:\HelloWorld> dotnet test
Test run for C:\HelloWorld\HelloWorld.Tests\bin\Debug\netcoreapp3.0\HelloWorld.Tests.dll (.NETCoreApp,Version=v3.0)
Microsoft (R) Test Execution Command Line Tool Version 16.3.0
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...

A total of 1 test files matched the specified pattern.

Test Run Successful.
Total tests: 1
Passed: 1
Total time: 3,4618 Seconds
C:\HelloWorld>

```

1 tests 0 0 Azure: rasmusl@microsoft.com HelloWorld.sln Ln 22, Col 50 Spaces: 4 UTF-8 CRLF C# 1

# Agenda

Note on *virtual*

Testing ...

Dependency Injection

Stubbing and mocking

Testing Entity Framework

# Virtual

Autogenerated

```
public partial class FuturamaContext : DbContext
{
    public virtual DbSet<Character> Characters { get; set; }
}
```

```
public class SuperheroContext : DbContext
{
    public DbSet<Superhero> Superheroes { get; set; }
}
```

# Virtual

```
public class Superhero  
{  
    public int Id { get; set; }  
  
    public virtual City City { get; set; }  
  
    public ICollection<SuperheroPower> Powers { get; set; }  
}
```

Allow lazy loading of City

Do not allow lazy loading of Powers

Testing ...

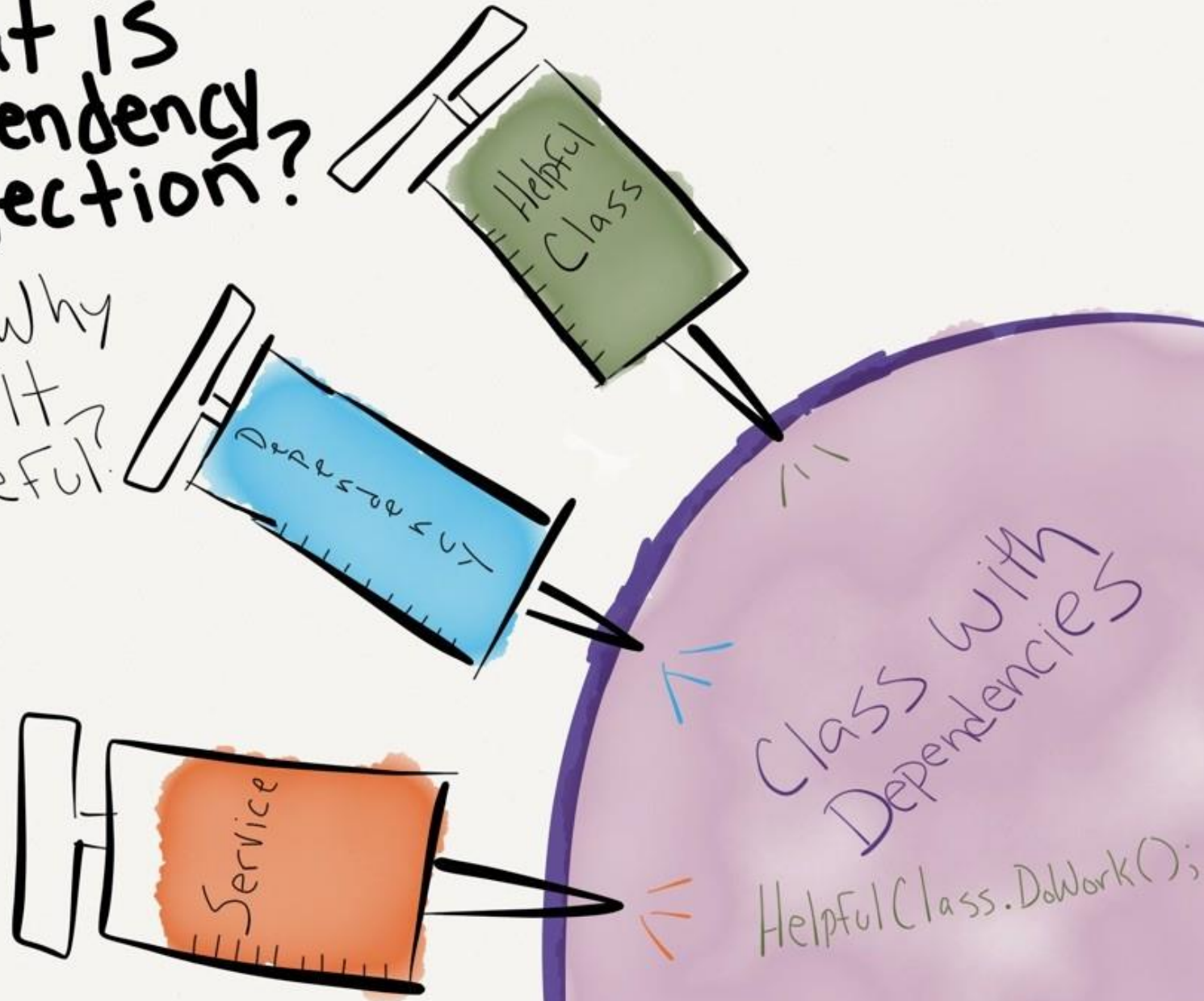
Testing live databases is hard

Testing live full systems is hard

By transitivity: Testing ... is hard...

# What is Dependency Injection?

Why is it useful?



# Dependency Injection

Software design pattern which  
implements Inversion of Control (IoC)

# Dependency Injection (DI)



Constructor Injection



Property (Setter) Injection



Interface Injection



# Dependency Injection

Structured readable code

Testable code

Dependency Inversion Principle

Separation of Concerns

Rock SOLID!!!

Pun intended

AWESOME!!

# Programming to interface, not implementation...

```
public interface IFooService
{
    bool Bar(Foo foo);
}
```

```
public class FooService : IFooService
{
    bool Bar(Foo foo)
    {
        // Implementation
    }
}
```

```
public interface IFooMapper
{
    Foo Map(Qux qux);
}
```

# Using IFooService

```
public class Baz
{
    public bool Grauhl
    {
        IFooMapper map
        var foo = map

        IFooService s

        return service
    }
}
```



# Constructor Injection (preferred)

```
public class Baz
{
    private readonly IFooMapper _mapper;
    private readonly IFooService _service;

    public Baz(IFooMapper mapper, IFooService service)
    {
        _mapper = mapper;
        _service = service;
    }

    public bool Grault(Qux qux)
    {
        var foo = _mapper.Map(qux);

        return _service.Bar(foo);
    }
}
```

Private readonly  
fields

Initialize from  
constructor

# Property Injection

```
public class Baz
{
    public IFooService Service { private get; set; }

    public bool Grault(Qux qux)
    {
        ...

        return Service?.Update(foo);
    }
}
```

Public setter

Is this King?

# Interface Injection

```
public interface IServiceSetter<T>
{
    void SetService(T service);
}
```

```
public interface IServiceSetter<T>
{
    T Service { set; }
}
```

# Interface Injection II

Interface



```
public class Baz : IServiceSetter<IFooService>
{
    private IFooService _service;

    public void SetService(IFooService service)
    {
        _service = service;
    }

    public bool Grault(Qux qux)
    {
        // Implementation
    }
}
```

Implement  
interface



# Interface Injection III

Interface

```
public class Baz : IServiceSetter<IFooService>
{
    public IFooService Service { private get; set; }

    public bool Grault(Qux qux)
    {
        // Implementation
    }
}
```

Implement  
interface



# Best practices

Use Adapter to  
enable interface  
if needed

Use constructor  
injection

Program to  
interface

Use an IoC  
container

More on this in a  
couple of weeks...

# IoC Container

```
dotnet add package Microsoft.Extensions.DependencyInjection
```

```
IServiceCollection services = new ServiceCollection();
```

```
services.AddScoped<IService, Service>();
```

```
var provider = services.BuildServiceProvider();
```

```
var service = provider.GetRequiredService<IService>();
```

# Unit Testing Best Practices (recap)

01

Never test against a live database, file, or web service

02

Single Responsibility Principle

03

Only test the "System Under Test"

04

Atomic tests

05

Use either mocks or stubs

# Stubbing and mocking

# Stubs

```
public class FooServiceFalseStub : IFooService
{
    public bool Bar(Foo foo)
    {
        return false;
    }
}
```

# Stubs

```
[Fact]
public void Exec_when_IFooService_Update_returns_false_returns_false()
{
    IFooService service = new FooServiceTrueStub();

    var baz = new Baz(service);

    var result = baz.Exec(new Foo());

    Assert.False(result);
}
```

# Mocks

[Fact]

```
public void Exec_when_IFooService_Update_returns_false_returns_false()
{
    var mock = new Mock<IFooService>();
    IFooService service = mock.Object;

    var baz = new Baz(service);

    var result = baz.Exec(new Foo());

    Assert.False(result);
}
```

# Mocks

[Fact]

```
public void Exec_when_IFooService_Update_true_returns_true()
{
    var mock = new Mock<IFooService>();
    mock.Setup(m => m.Update(It.IsAny<Foo>()))).Returns(true);

    var baz = new Baz(mock.Object);

    var result = baz.Exec(new Foo());

    Assert.True(result);
}
```



# Demo

White-box testing with ***Moq***

# Testing Entity Framework

# SQLite in-memory database

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

```
using var connection = new SqlConnection("Filename=:memory:");  
connection.Open();
```

```
var builder = new DbContextOptionsBuilder<MyContext>().UseSqlite(connection);  
using var context = new MyContext(builder.Options);
```

# Demo

Black box testing with ***SQLite in-memory***

# Best practices



Wrap in logical units/service classes/repositories



Don't test built-in code...



Program to interface