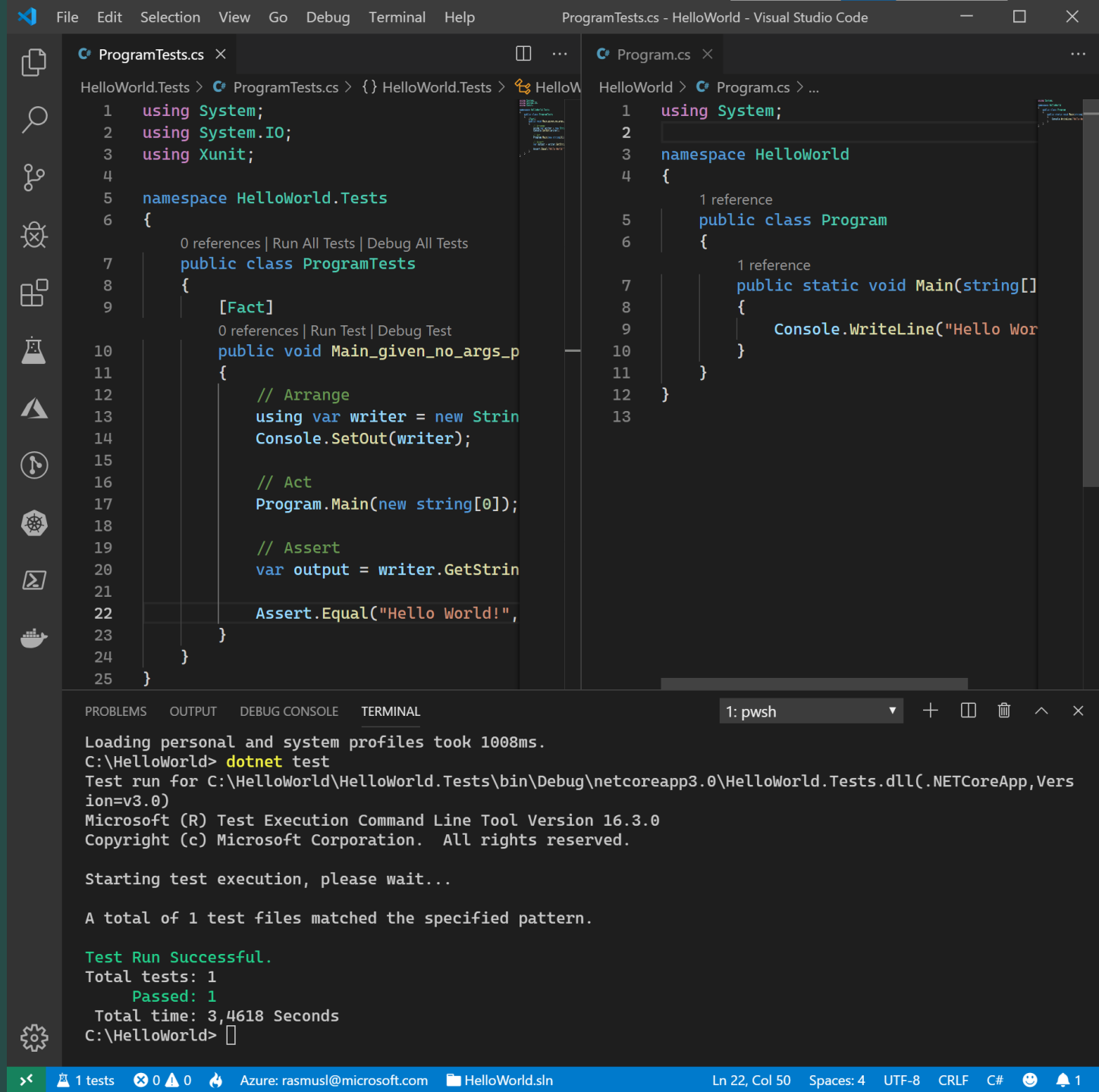
C#
Dependency Injection
Testing Entity Framework

Rasmus Lystrøm
Associate Professor
ITU
rnie@itu.dk



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

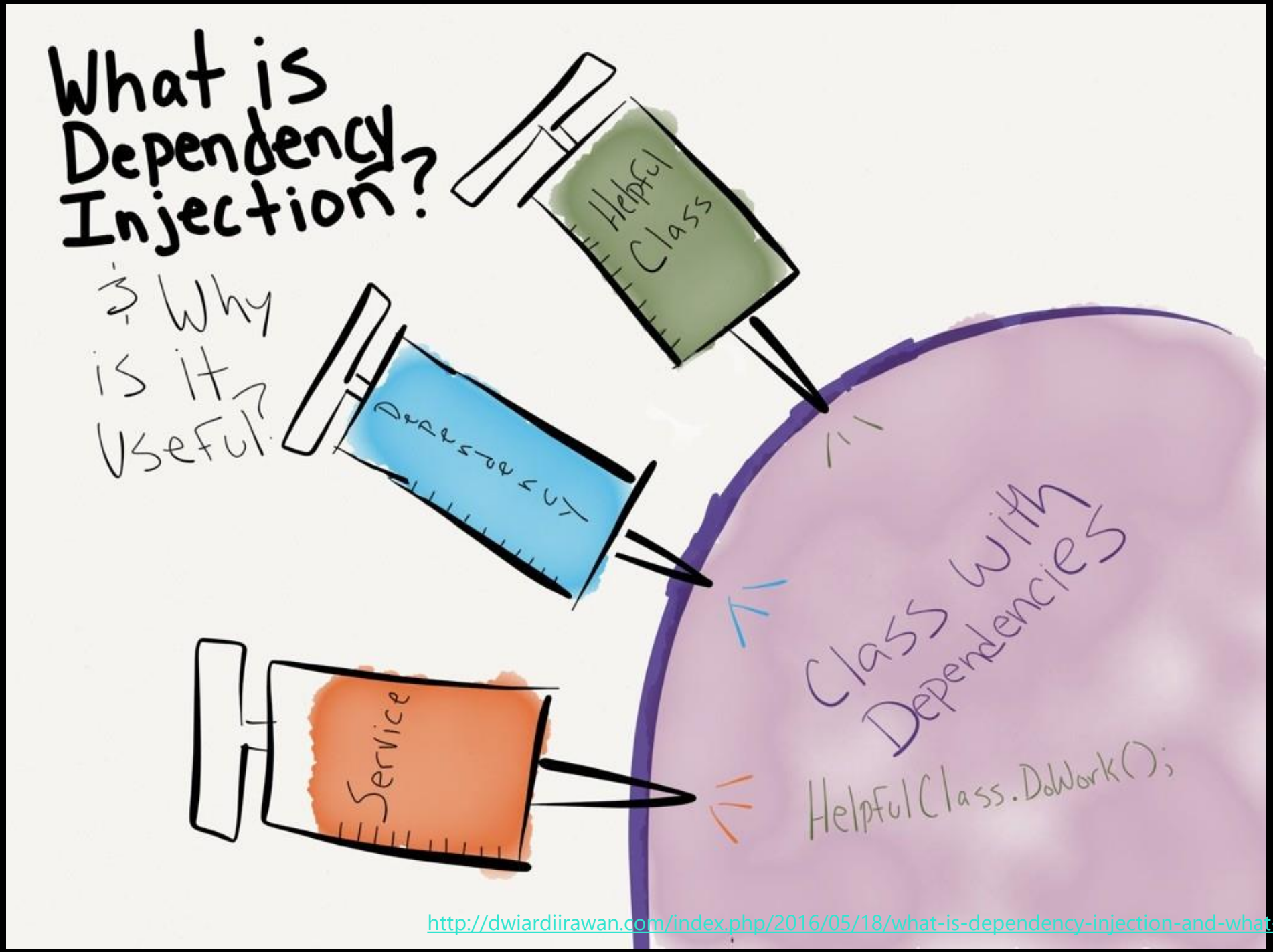
Do not allow lazy loading of Powers

Testing …

Testing live databases is hard

Testing live full systems is hard

By transitivity: Testing … is hard…



http://dwiardiirawan.com/index.php/2016/05/18/what-is-dependency-injection-and-what-are-the-advantage/

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!!!

AWESOME!!

Pun intended

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 IFooServi

```
public class Baz
{
    public bool Graul
    {
        IFooMapper ma
        var foo = map

        IFooService s

        return servic
    }
}
```

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

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

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

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

Interface

Implement interface

Interface Injection III

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

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

Interface

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 SqliteConnection("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