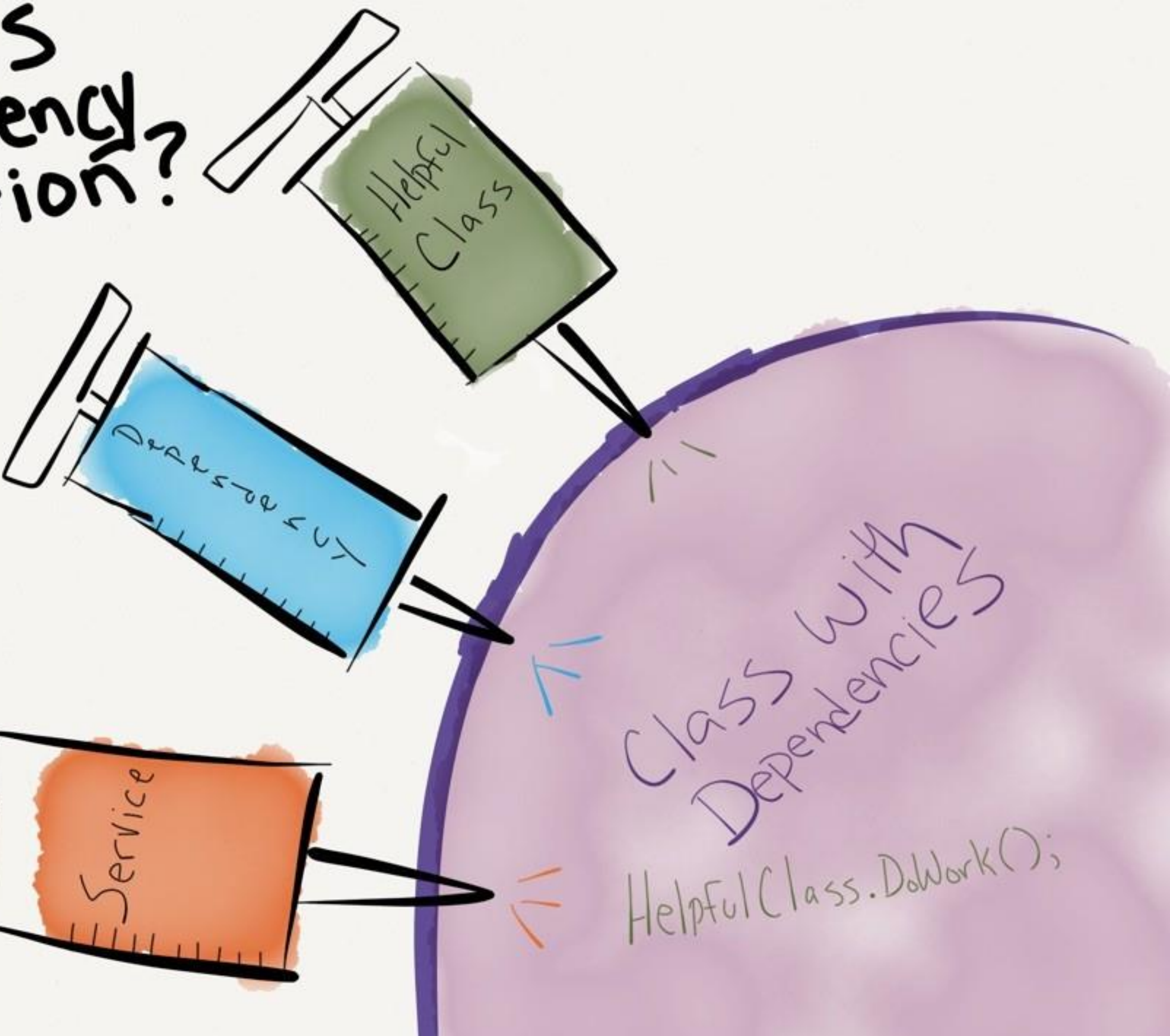
C♯ Dependency Injection Unit Testing EF

Rasmus Lystrøm
External Associate Professor
ITU

What is Dependency Injection?

¿Why is it useful?



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

Dependency Injection (DI)

Software design pattern which implements Inversion of Control (IoC)

Constructor Injection

Property (setter) Injection

Interface Injection

Structered readable code
Testable code
Dependency Inversion Principle
Separation of Concerns

Rock SOLID!!!

Pun intended

AWESOME!!

Programming to interface, not implementation…

```
public interface IFooService : IDisposable
{
   bool Update(Foo foo);
}
```

Constructor Injection

```
public class Worker : IDisposable
{
    private readonly IFooService _service;

    public Worker(IFooService service)
    {
        _service = service;
    }

    public bool DoWork(FooDto fooDto)
    {
        // Implementation
    }

    public void Dispose()
    {
        _service.Dispose();
    }
}
```

Private readonly field

Initialize from constructor

Remember to call Dispose…

Property Injection

Public setter

```
public class Worker : IDisposable
{
    public IFooService Service { private get; set; }

    public void DoWork(FooDto foo)
    {
        // Implementation
    }

    public void Dispose()
    {
        Service?.Dispose();
    }
}
```

Dispose with the King…

Interface Injection

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

Interface Injection II

Interface

```
public class Worker : IServiceSetter<IFooService>, IDisposable
{
    private IFooService _service;

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

    public void DoWork(FooDto fooDto)
    {
        // Implementation
    }

    public void Dispose()
    {
        _service?.Dispose();
    }
}
```

Implement interface

Interface Injection III

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

Interface Injection IV

Interface

```
public class Worker : IServiceSetter<IFooService>, IDisposable
{
    public IFooService Service { private get; set; }

    public bool DoWork(FooDto fooDto)
    {
        // Implementation
    }

    public void Dispose()
    {
        Service?.Dispose();
    }
}
```

Implement interface

Best practices

Use Adapter to enable interface if needed

Use constructor injection

Implement IDisposable

Use an IoC container

When interface not possible directly, use Adapter pattern

Pun intended

```
public sealed class FoolishService
{
    public bool Update(Foo foo)
    {
        // Implementation
    }
}
```

When interface not possible directly, use Adapter pattern II

```
public class FoolishServiceAdapter : IFooService
{
    private readonly FoolishService _service = new FoolishService();

    public bool Update(Foo foo)
    {
        return _service.Update(foo);
    }

    public void Dispose()
    {
    }
}
```

Unit Testing

Best Practices

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

Single Responsibility Principle

Only test the "System Under Test"

Use either mocks or stubs

Stub testing

Test stub

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

    public void Dispose()
    {
    }
}
```

Stub testing II

```
public class WorkerTests
{
    [Fact]
    public void DoWork_when_IFooService_Update_false_returns_false()
    {
        IFooService service = new FooServiceFalseStub();

        using (var worker = new Worker(service))
        {
            var result = worker.DoWork(new FooDto());

            Assert.False(result);
        }
    }
}
```

Mock testing

Mock using Moq

```
public class WorkerTests
{
    [Fact]
    public void DoWork_when_IFooService_Update  alse_returns_false()
    {
        var mock = new Mock<IFooService>();
        IFooService service = mock.Object;

        using (var worker = new Worker(service))
        {
            var result = worker.DoWork(new FooDto());

            Assert.False(result);
        }
    }
}
```

Mock testing II

Configure the mock

```
public class WorkerTests
{
    [Fact]
    public void DoWork_when_IFooService_Updat_true_returns_true()
    {
        var mock = new Mock<IFooService>();
        mock.Setup(m => m.Update(It.IsAny<Foo>())).Returns(true);

        using (var worker = new Worker(mock.Object))
        {
            var result = worker.DoWork(new FooDto());

            Assert.True(result);
        }
    }
}
```

Demo

Testing
Entity Framework

Best practices

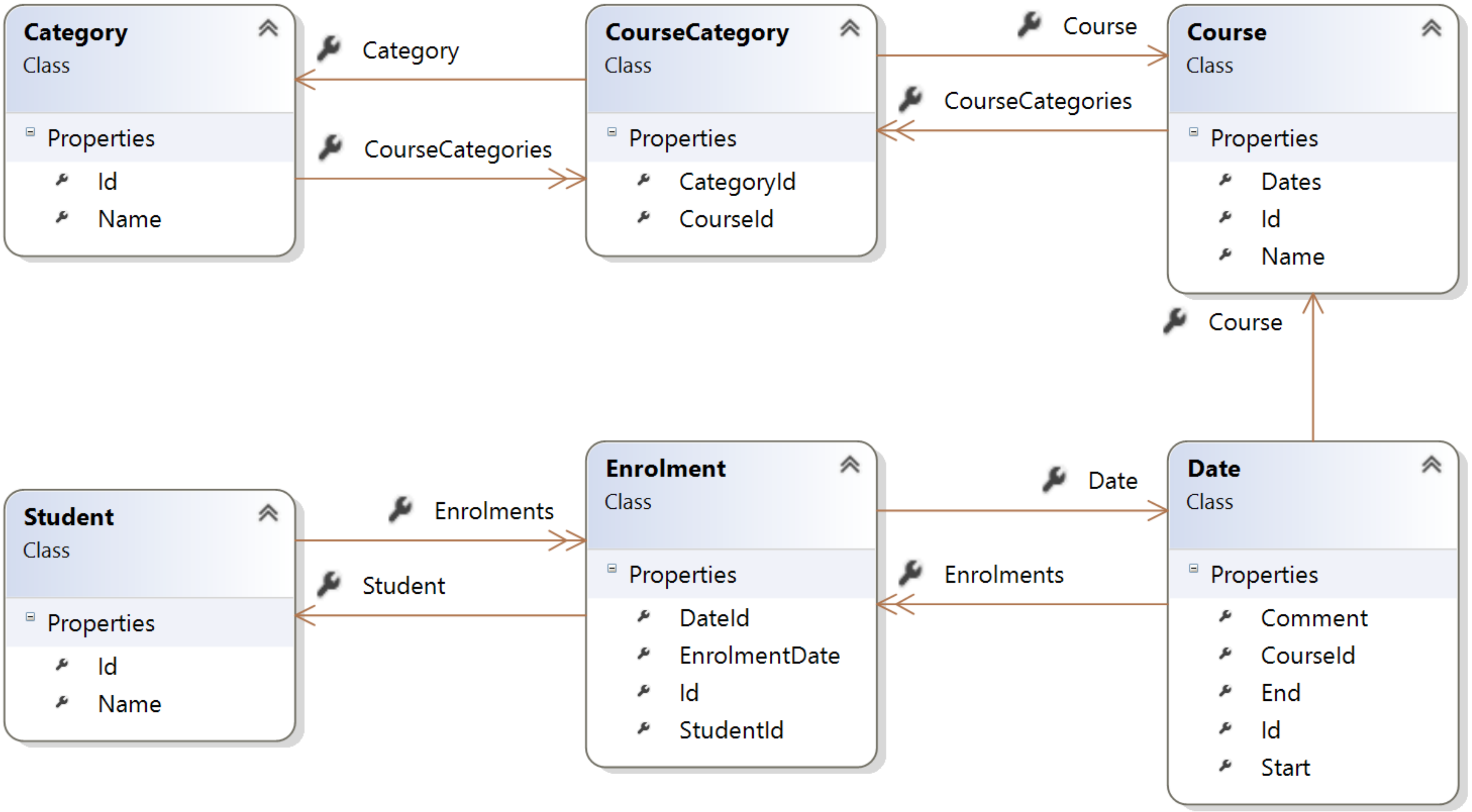
Implement IDisposable

Wrap in logical units/service classes/repositories

Don't test built in code…

Program to interface

Entity Model



IStudentRepository

Interface to enable Dependency Injection at the next layer

```
public interface IStudentRepository : IDisposable
{
    IEnumerable<StudentListDto> Read();

    IEnumerable<StudentListDto> Read(Status status);

    bool Enrol(int studentId, int dateId);

    int Create(StudentCrudDto student);

    bool Update(StudentCrudDto student);

    bool Delete(int studentId);
}
```

Enable InMemoryDatabase for DbContext class

```
public MyContext()
{
}

public MyContext(DbContextOptions<CourseBaseContext> options)
    : base(options)
{
}
```

```
"dependencies": {
    ""Microsoft.EntityFrameworkCore.InMemory": "1.0.1
},
```

In test library

https://docs.efproject.net/en/latest/miscellaneous/testing.html

Demo