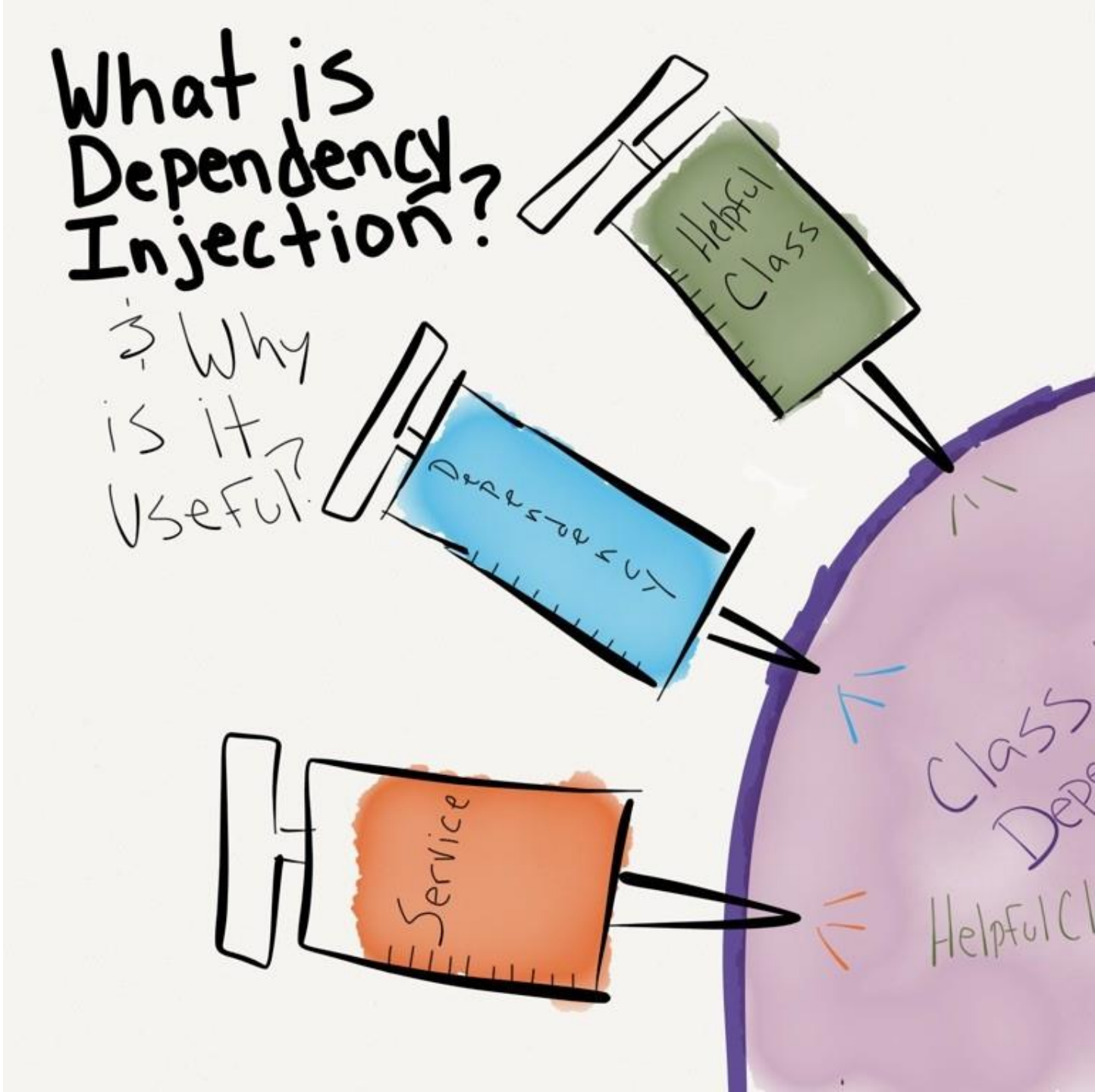
C♯ Dependency Injection Testing Entity Framework

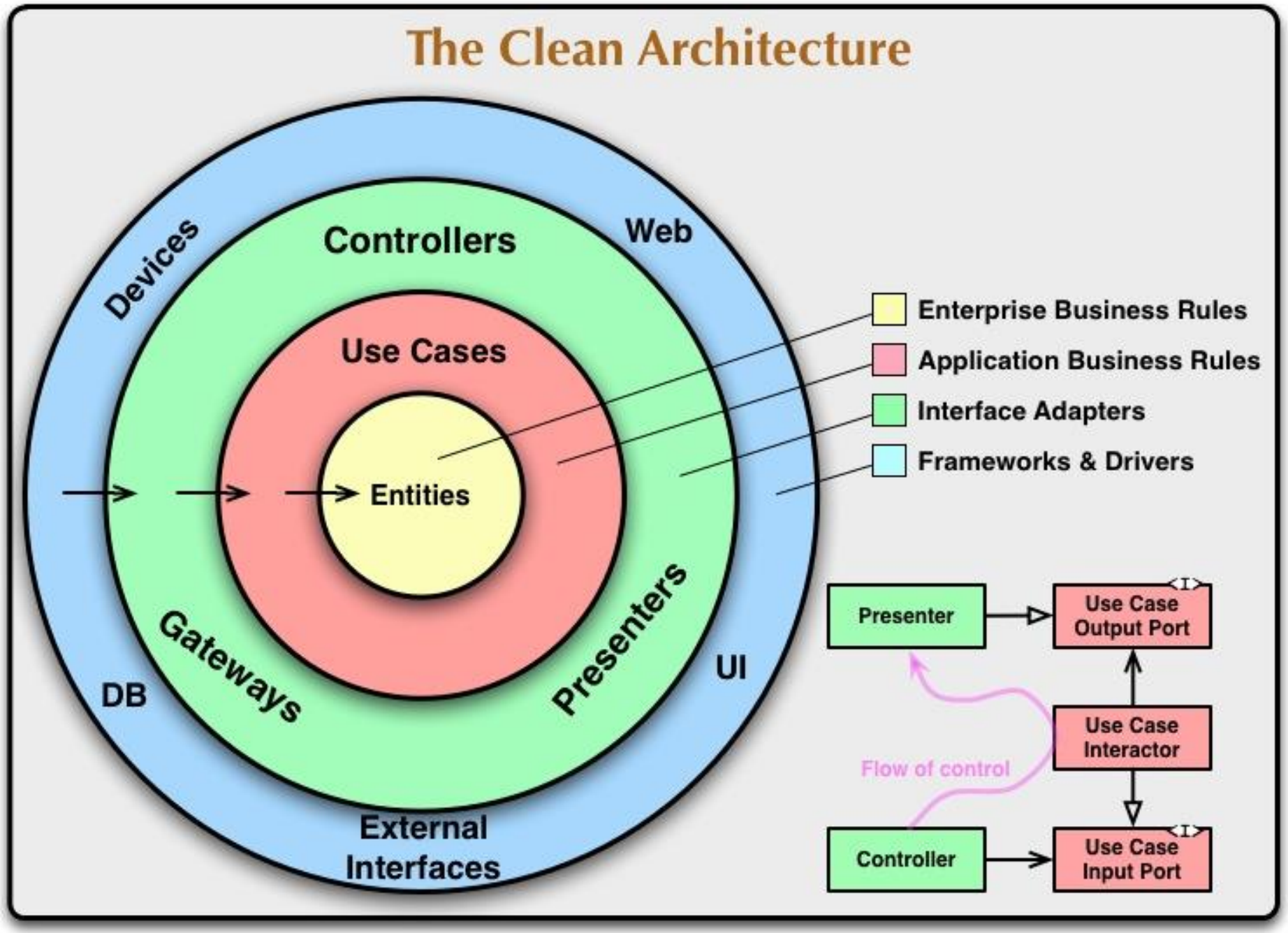
Rasmus Lystrøm
Associate Professor
ITU



Agenda

Clean Onions
Repository pattern
Testing database code
Dependency Injection
Testing Entity Framework

Onion Architecture


The Clean Architecture
Devices
Web
Controllers
Use Cases
Entities
Gateways
Presenters
DB
UI
External
Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case
Output Port
Use Case
Interactor
Flow of control
Controller
Use Case
Input Port

The Repository Pattern

The Repository Pattern

Enable CRUD on domain objects (entities)

Usually: one repository per entity

Debatable: has a *Save()* method

Generic Repository

```
public interface Repository<T, K>
{
    T Create(T entity);
    IReadOnlyCollection<T> Read();
    T Read(K id);
    void Update(T entity);
    void Delete(K id);
}
```

The Repository Pattern

… but wait … Entity Framework already does not for me!?

Recommended Repository: Per entity e.g., Character

```
public interface ICharacterRepository : IDisposable
{
    int Create(CharacterDetailsDTO character);
    CharacterDetailsDTO Read(int characterId);
    IReadOnlyCollection<CharacterDTO> Read();
    void Update(CharacterDetailsDTO character);
    void Delete(int characterId);
}
```

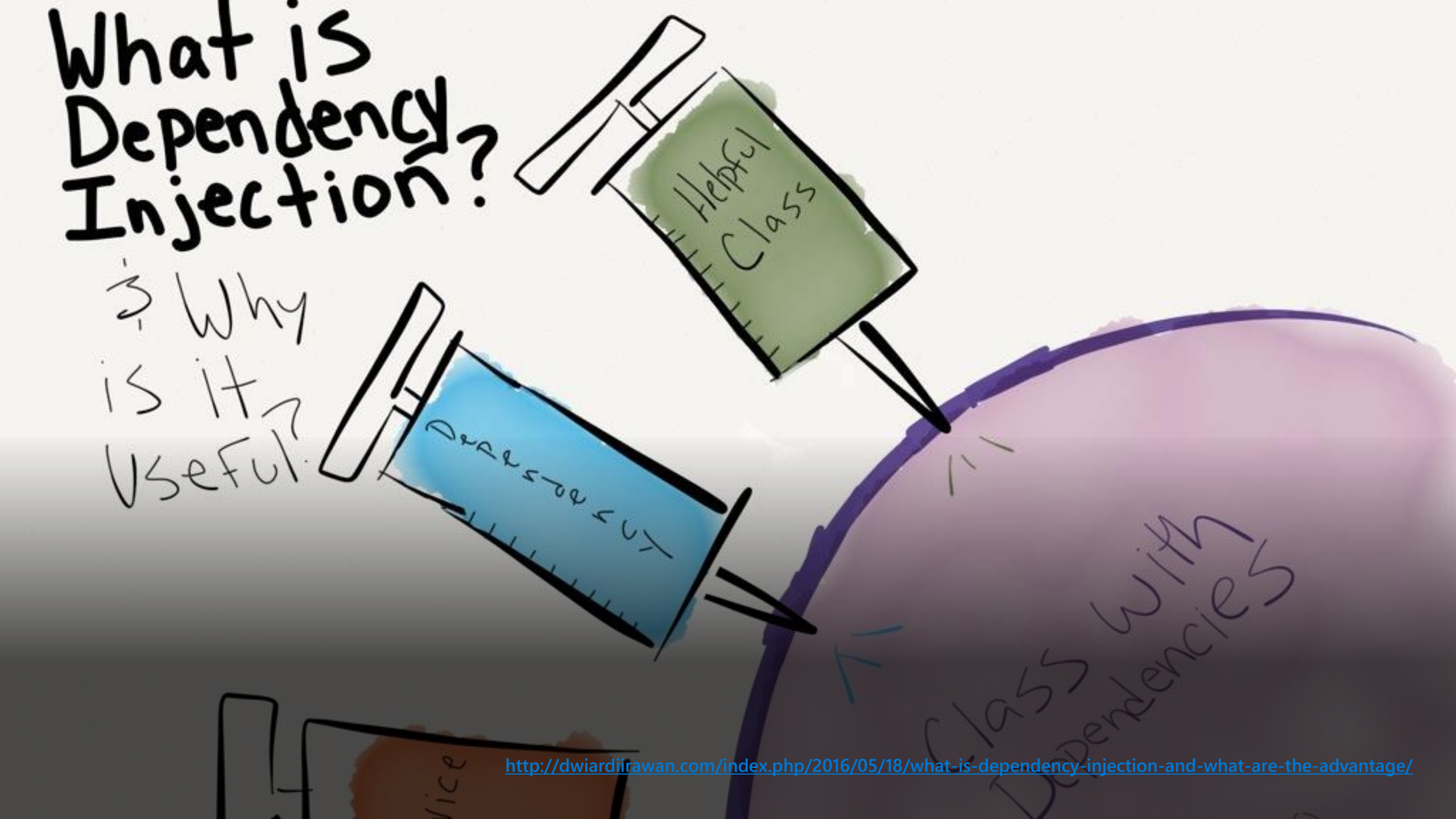
…or something similar…

Testing …

Testing live databases is hard

Testing live full systems is hard

By transitivity: Testing … is hard…



http://dwiardirawan.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	Use	Program	Use
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...

Testing Entity Framework

SQLite in-memory database

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

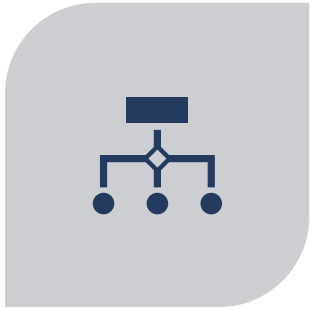
```
using var connection = new SqliteConnection("Filename=:memory:");
connection.Open();

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

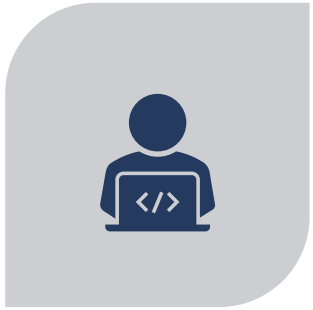
Demo

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

Best practices

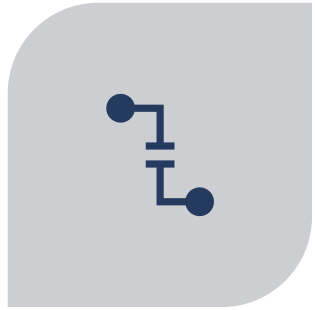


WRAP IN LOGICAL UNITS/SERVICE CLASSES/REPOSITORIES



DON’T TEST BUILT-IN CODE…

PROGRAM TO INTERFACE



REPOSITORIES SHOULD NOT DEPEND ON OTHER REPOSITORIES

Thank you