

```

P_RatNumForName (SKYPLATNAME);
// If no world levels, no secret exit
if ( ! (gamemode == commercial) )
    && (W_CheckNumForName("map31") < 0)
    secretexit = false;
else
    secretexit = true;
gamemode == ga_completed;

// skip the description field
memset (check, 0, sizeof (check));
sprintf (check, "version %d", VERSION);
if (strcmp (save_p, check))
    return; // bad version
save_p += VERSIONSIZE;

// This was quite messy with SPECIAL and commented parts,
// supposedly backs to make the latest edition work.
// It might not work properly.
if (episode < 1)
    episode = 1;

if (gamemode == retail )
{
    if (episode < 4)
        episode = 4;
    else if ( ! gamemode == shareware )
    {
        if (episode > 3)
            episode = 1; // only start episode 1 on shareware
        else
        {
            if (episode > 3)
                episode = 3;
        }
    }
}

if (map < 1)
    map = 1;

if (map > 9)
    && ( ! gamemode != commercial )
    map = 9;

M_ClearRandom ();

if (skill == sk_nightmare || respawnparm)
    respawnmonsters = true;
else
    respawnmonsters = false;

// fastparm || skill == sk_nightmare && gameskill != sk_nightmare
for (i=S_SARG_RUN1; i<=S_SARG_PAIN2; i++)
    mobinfo[MT_BRUISERSHOT].speed = 20*FRACUNIT;
mobinfo[MT_TROOPSHOT].speed = 20*FRACUNIT;

// force players to be initialized upon first level load
for (i=0; i<MAXPLAYERS; i++)
    players[i].playerstate = PST_REBORN;

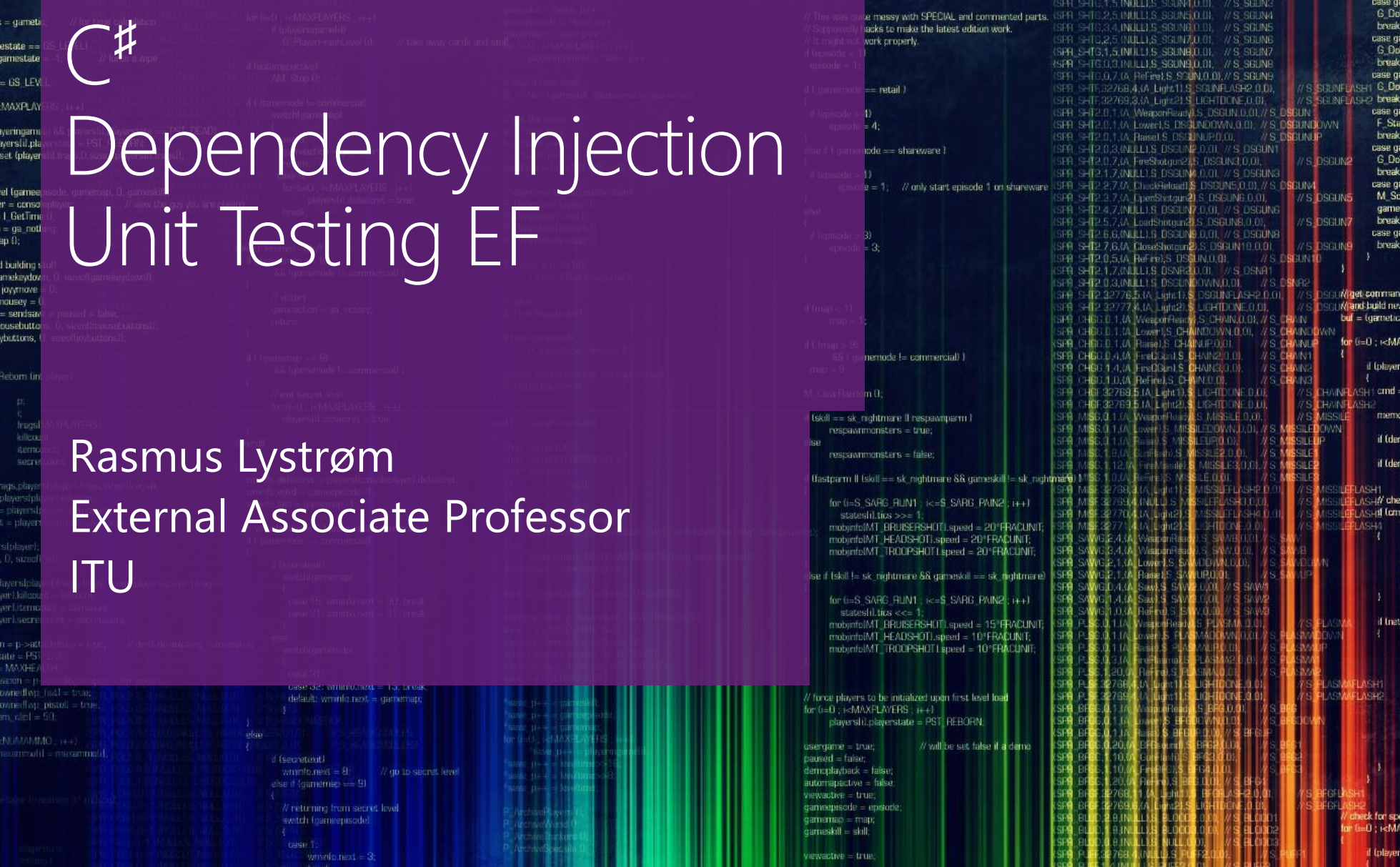
usergame = true; // will be set false if a demo
paused = false;
demoPlayback = false;
autoMapview = false;
viewactive = true;
gamemode = episode;
gamemap = map;
gameskill = skill;
viewactive = true;

```

C#

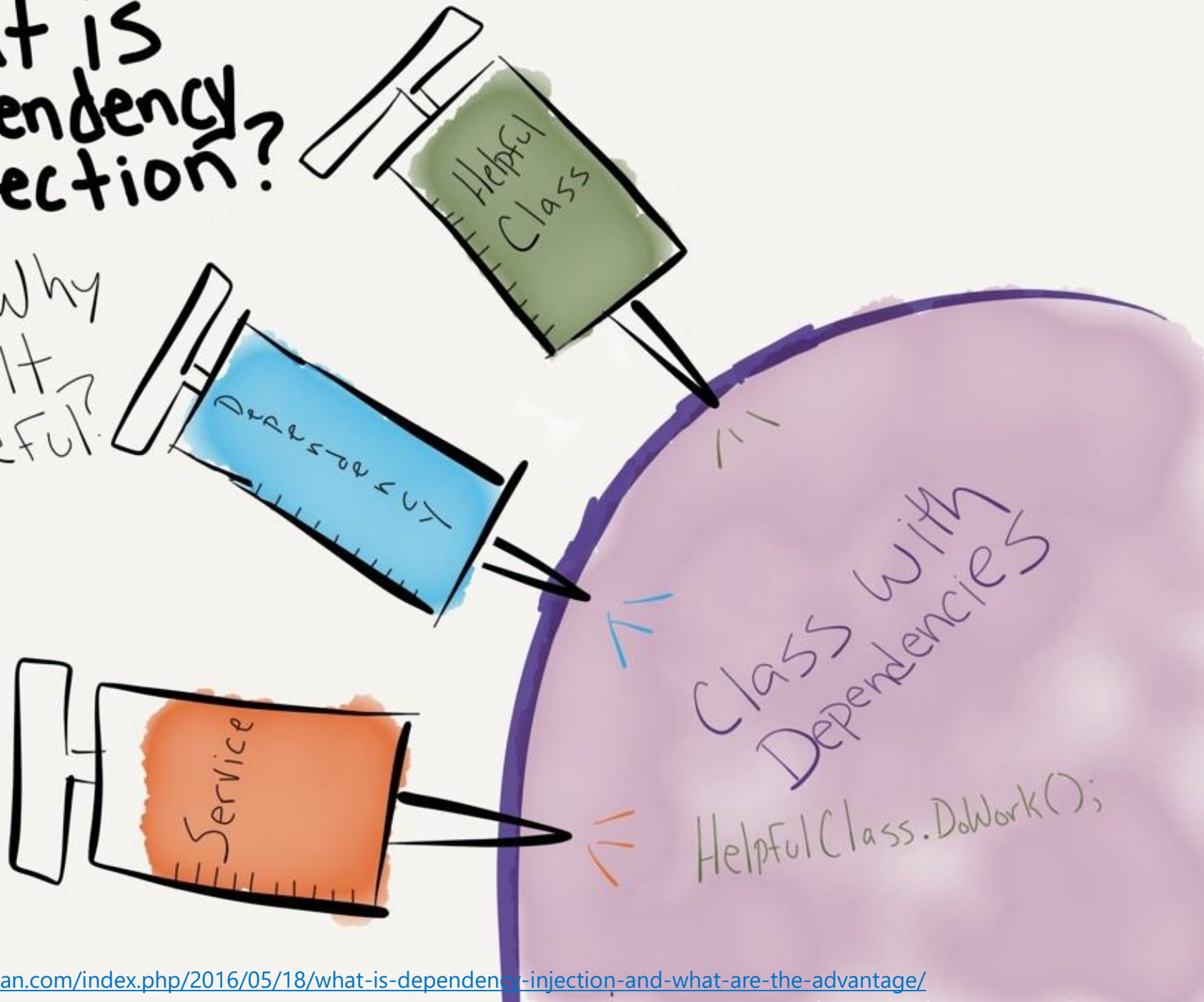
Dependency Injection Unit Testing EF

Rasmus Lystrøm
External Associate Professor
ITU



What is Dependency Injection?

Why is it useful?



Dependency Injection (DI)

Software design pattern which implements Inversion of Control (IoC)

Constructor
Injection

Property (setter)
Injection

Interface
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 : 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

Use an IoC
container

Implement
IDisposable

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_returns_false_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_Update_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

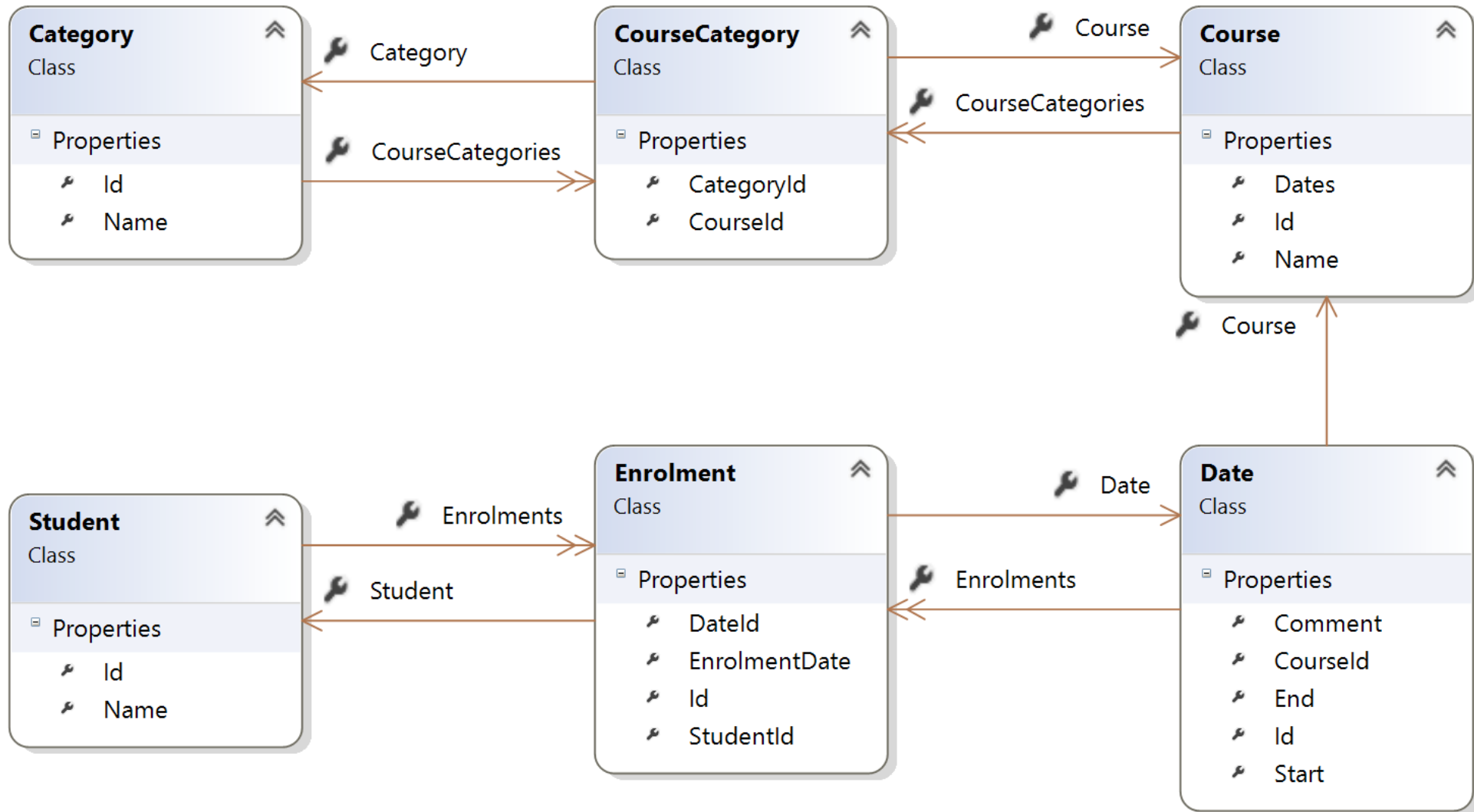
Implement
IDisposable

Wrap in logical
units/service
classes/repositories

Program to
interface

Don't test built
in code...

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

Demo