

Module 5:

"DI Containers"



TEKNOLOGISK
INSTITUT

Agenda

- ▶ **Pure DI**
- ▶ Dependency Injection Containers
- ▶ Pattern: Register-Resolve-Release
- ▶ Workshop C.1: Adding Unity DI to Services
- ▶ DI: Interception
- ▶ Workshop C.2: Interception
- ▶ DI: Lifetime Management
- ▶ Workshop D: If Time Permits

Pattern: Pure DI (a.k.a. Poor Man's DI)

- ▶ *Pure DI is the practice of applying DI without a DI Container.*
- ▶ Outline
 - This is essentially what we have been doing throughout the course
 - Compose object graphs manually at Composition Root
- ▶ See:
"Dependency Injection Principles, Practices, and Patterns"
Steven van Deursen and Mark Seemann (2019)

Agenda

- ▶ Pure DI
- ▶ **Dependency Injection Containers**
- ▶ Pattern: Register-Resolve-Release
- ▶ Workshop C.1: Adding Unity DI to Services
- ▶ DI: Interception
- ▶ Workshop C.2: Interception
- ▶ DI: Lifetime Management
- ▶ Workshop D: If Time Permits

Dependency Injection Containers

- ▶ Unity
- ▶ Autofac
- ▶ Simple Injector
- ▶ Microsoft.Extensions.DependencyInjection

- ▶ Ninject
- ▶ Castle Windsor
- ▶ Spring.NET
- ▶ Simpleloc
- ▶ ...

- ▶ Don't create your own! 😊

Unity Container

- ▶ Microsoft's "classic" DI container implementation
 - Available in **Unity** nuget package

```
IUnityContainer unity = new UnityContainer();

unity.RegisterType<ICreateUserService, CreateUserService>()
    .RegisterType<IMessageTemplateRepository, SqlMessageTemplateRepository>()
    .RegisterType<MessageTemplateContext>()
    .RegisterType<CreateUserViewModel>()
    .RegisterInstance<IDependency>(new DependencyImplementation())
    ;
```

```
ICreateUserService service = unity.Resolve<ICreateUserService>();
```

Agenda

- ▶ Pure DI
- ▶ Dependency Injection Containers
- ▶ **Pattern: Register-Resolve-Release**
- ▶ Workshop C.1: Adding Unity DI to Services
- ▶ DI: Interception
- ▶ Workshop C.2: Interception
- ▶ DI: Lifetime Management
- ▶ Workshop D: If Time Permits

Pattern: Register-Resolve-Release

- ▶ *Always do a sequence of three things with a container:*
 - *Register components with the container*
 - *Resolve root components*
 - *Release components from the container.*
- ▶ Outline
 - RRR captures the best practice of container use in the Composition Root **only!**
- ▶ See:
<https://blog.ploeh.dk/2010/09/29/TheRegisterResolveReleasepattern/>
Mark Seemann (2010)

Agenda

- ▶ Pure DI
- ▶ Dependency Injection Containers
- ▶ Pattern: Register-Resolve-Release
- ▶ **Workshop C.1: Adding Unity DI to Services**
- ▶ DI: Interception
- ▶ Workshop C.2: Interception
- ▶ DI: Lifetime Management
- ▶ Workshop D: If Time Permits



Workshop C.1: Adding Unity DI to Services



Agenda

- ▶ Pure DI
- ▶ Dependency Injection Containers
- ▶ Pattern: Register-Resolve-Release
- ▶ Workshop C.1: Adding Unity DI to Services
- ▶ **DI: Interception**
- ▶ Workshop C.2: Interception
- ▶ DI: Lifetime Management
- ▶ Workshop D: If Time Permits

Dependency Injection Topic Areas

- ▶ Object Composition
- ▶ Interception
- ▶ Lifetime Management

Definition: Interception

- ▶ *Interception is the ability to intercept calls between two collaborating components in such a way that you can enrich or change the behavior of the dependency without the need to change the two collaborators themselves.*
- ▶ Examples
 - Security, Caching, Logging, Composites, Decorators, Proxy, ...
 - Test-specific behavior
- ▶ See:
"Dependency Injection Principles, Practices, and Patterns"
Steven van Deursen and Mark Seemann (2019)

Example: Composite Behavior

- ▶ Sending messages via multiple channels

```
class CompositeMessageTransmissionStrategy : IMessageTransmissionStrategy
{
    readonly IEnumerable<IMessageTransmissionStrategy> _strategies;

    public CompositeMessageTransmissionStrategy(
        params IMessageTransmissionStrategy[] strategies)
    {
        _strategies = strategies;
    }

    async public Task TransmitAsync(...) { ...}
}
```

Example: Composite Behavior

- Note: This is where the various container frameworks have (quirky) specific behaviors...

```
_container
    .RegisterType<IMessageTransmissionStrategy,
        SendGridEmailTransmissionStrategy>()
    .RegisterType<IMessageTransmissionStrategy,
        TwilioSmsTransmissionStrategy>()
    .RegisterType<IMessageTransmissionStrategy,
        CompositeMessageTransmissionStrategy>(  
        new InjectionConstructor(  
            new ResolvedArrayParameter<IMessageTransmissionStrategy>(  
                new ResolvedParameter<TwilioSmsTransmissionStrategy>(),  
                new ResolvedParameter<SendGridEmailTransmissionStrategy>()  
            )  
        )  
    );
```

Agenda

- ▶ Pure DI
- ▶ Dependency Injection Containers
- ▶ Pattern: Register-Resolve-Release
- ▶ Workshop C.1: Adding Unity DI to Services
- ▶ DI: Interception
- ▶ **Workshop C.2: Interception**
- ▶ DI: Lifetime Management
- ▶ Workshop D: If Time Permits

Workshop C.2: Interception



Agenda

- ▶ Pure DI
- ▶ Dependency Injection Containers
- ▶ Pattern: Register-Resolve-Release
- ▶ Workshop C.1: Adding Unity DI to Services
- ▶ DI: Interception
- ▶ Workshop C.2: Interception
- ▶ **DI: Lifetime Management**
- ▶ Workshop D: If Time Permits

Definition: Lifestyles

- ▶ *A Lifestyle is a formalized way of describing the intended lifetime of a dependency.*
- ▶ Transient ~ New instance at every resolve
- ▶ Singleton ~ Only one instance exists per container
- ▶ Scoped ~ New instance at every scope
- ▶ Unity has **LifetimeManager** associated with registrations

Transient Lifestyle

- ▶ Default lifestyle for Unity and most other containers
 - Instances are not disposed by container

```
container.RegisterType<ILogger, ConsoleLogger>();
```

```
container.RegisterType<ILogger, ConsoleLogger>(
    new TransientLifetimeManager()
);
```

Singleton Lifestyle

- ▶ Can be selected for Unity and most other containers
 - Instances are tracked and disposed by container

```
container.RegisterSingleton<ILogger, ConsoleLogger>();
```

```
container.RegisterType<ILogger, ConsoleLogger>(
    new ContainedControlledLifetimeManager()
);
```

```
container.RegisterInstance<ILogger>( new ConsoleLogger() );
```

Scoped Lifestyle

- ▶ Can be selected for Unity and most other containers
 - Instances are reused within the each child container

```
IDependencyScope scope = container.CreateChildContainer();
```

- ▶ Usually not selected explicitly, but chosen by plugging into some inversion of control framework,
 - E.g. per request in ASP.NET

Agenda

- ▶ Pure DI
- ▶ Dependency Injection Containers
- ▶ Pattern: Register-Resolve-Release
- ▶ Workshop C.1: Adding Unity DI to Services
- ▶ DI: Interception
- ▶ Workshop C.2: Interception
- ▶ DI: Lifetime Management
- ▶ **Workshop D: If Time Permits**



Workshop D: If Time Permits...



Summary

- ▶ Pure DI
- ▶ Dependency Injection Containers
- ▶ Pattern: Register-Resolve-Release
- ▶ Workshop C.1: Adding Unity DI to Services
- ▶ DI: Interception
- ▶ Workshop C.2: Interception
- ▶ DI: Lifetime Management
- ▶ Workshop D: If Time Permits



WINCUBATE

Jesper Gulmann Henriksen

PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31

Email : jgh@wincubate.net

WWW : <http://www.wincubate.net>

Hasselvangelen 243

8355 Solbjerg

Denmark