Dependency Injection mit dem Unity Container

Vom Problem zur Lösung – unter Anwendung des Unity Containers

W3L AG
info@W3L.de

2015

Inhalt

- Problemstellung
- Einführung Dependency Injection
- Manuelle Objektkomposition
- Objektkomposition mit dem Unity Container
- Dependency Injection Container

Problemstellung – Enge Kopplung

```
public class ProductOrderService
{

    private readonly SqlCustomerRepository _customerRepository;
    private readonly SqlProductRepository _productRepository;
    private readonly EmailSender _emailSender;

    public ProductOrderService()
    {
        _customerRepository = new SqlCustomerRepository();
        _productRepository = new SqlProductRepository();
        _emailSender = new EmailSender();
    }

}
```

- **Nachteile**
  - Implementierungen können nicht ausgetauscht werden
  - Objekte können nicht geteilt werden
  - Isoliertes Testen des Services nicht möglich
  - Benötigte Abhängigkeiten des Services sind von außen nicht erkennbar
  - Service-Klasse muss abhängige Klassen kennen und referenzieren können

Lösung – Dependency Injection

- **Constructor Injection**

```
public class ProductOrderService
{

    private readonly ICustomerRepository _customerRepository;
    private readonly IProductRepository _productRepository;
    private readonly IEmailSender _emailSender;

    public ProductOrderService(ICustomerRepository customerRepository,
          IProductRepository productRepository, IEmailSender emailSender)
    {
        _customerRepository = customerRepository;
        _productRepository = productRepository;
        _emailSender = emailSender;
    }

}
```

- **Behebt die Nachteile der vorherigen Lösung**
- **Weitere Varianten**
  - Setter Injection
  - Interface Injection
- **Neue Problemstellung**
  - Wer erstellt nun initial die Objekte und setzt sie zusammen?

Composition Root

- **Definition**
  - “A Composition Root is a (preferably) unique location in an application where modules are composed together.” – [1]
- **Liegt möglichst nah am Einstiegspunkt einer Anwendung**
- **Sollte möglichst die einzige Stelle sein, an der ein Dependency Injection Container verwendet wird**

- **Verwendungsmuster**
  - **Register**
    - **Registriere alle Komponenten**
  - **Resolve**
    - **Erstelle initialen Objektgraph**
  - **Release**
    - **Gebe initialen Objektgraph frei**

Manuelle Objektkomposition

```
public class PoorMansBootstrapper
{
    public void Register() { /* Nothing to do */ }

    public Application Resolve()
    {
        ICustomerRepository singletonCustomerRepository = new SqlCustomerRepository();
        IProductRepository singletonProductRepository = new SqlProductRepository();
        return new Application(
            new SecuredProductOrderService(
                new ProductOrderService(
                    singletonCustomerRepository,
                    singletonProductRepository,
                    new EmailSender())));
    }

    public void Release(Application application)
    {
        application.Dispose();
    }
}
```

Objektkomposition mit dem Unity Container

```
public class UnityBootstrapper : IDisposable
{
    private readonly UnityContainer _container = new UnityContainer();

    public void Register()
    {
        // Singleton.
        _container.RegisterType<ICustomerRepository, SqlCustomerRepository>(
            new ContainerControlledLifetimeManager());
        _container.RegisterType<IProductRepository, SqlProductRepository>(
            new ContainerControlledLifetimeManager());

        // Transient.
        _container.RegisterType<IEmailSender, EmailSender>();

        // Decorator.
        _container.RegisterType<IProductOrderService>(new InjectionFactory(x =>
            new SecuredProductOrderService(x.Resolve<ProductOrderService>())));
    }

    public Application Resolve()
    {
        return _container.Resolve<Application>();
    }

    public void Release(Application application)
    {
        _container.Teardown(application);
        application.Dispose();
    }

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

Dependency Injection Container

- Framework zur Unterstützung von Objektgraphkompositionen und -verwaltung
- Features
  - Unterschiedliche Konfigurationsmöglichkeiten
    - Register by Code
    - Register by Convention
    - Register by XML
  - Auto-Wiring
  - Lifetime-Management
    - Transient
    - Singleton
    - Per-Thread
    - Benutzerdefiniert
  - Interception
  - ...

Dependency Injection Container für .NET

- Autofac
- Castle.Windsor
- Ninject
- Simple Injector
- Spring.NET
- StructureMap
- Unity Container
- ...

Weitere Informationen

Seemann, Mark
Allgemeine Informationen
http://blog.ploeh.dk/tags.html#Dependency%20Injection-ref

Von Deursen, Steven
Problembeispiele und Lösungsansätze
http://www.cuttingedge.it/blogs/steven/

Fowler, Martin
Ausführliche Einführung mit Beispielen
http://martinfowler.com/articles/injection.html

Microsoft Prism
Praktische Anwendung von Composition Root
https://msdn.microsoft.com/en-us/library/ff648465.aspx

Quellen

[1] Seemann, Mark
http://blog.ploeh.dk/2011/07/28/CompositionRoot/

[2] Seemann, Mark
Dependency Injection in .NET
ISBN-13: 978-1935182504

[3] Microsoft
Unity Container
https://msdn.microsoft.com/en-us/library/dn170416.aspx

Inhouse-Schulungen

Wir bieten Inhouse-Schulungen und Beratung durch unsere IT-Experten und –Berater.

Schulungsthemen

- Softwarearchitektur (OOD)
- Requirements Engineering (OOA)
- Nebenläufige & verteilte Programmierung

Gerne konzipieren wir auch eine individuelle Schulung zu Ihren Fragestellungen.

Sprechen Sie uns an!
Tel. 0231/61 804-0, info@W3L.de

W3L-Akademie

In Zusammenarbeit mit der Fachhochschule Dortmund bieten wir

zwei Online-Studiengänge

- B.Sc. Web- und Medieninformatik
- B.Sc. Wirtschaftsinformatik

und 7 Weiterbildungen im IT-Bereich an.

Besuchen Sie unsere Akademie!
http://Akademie.W3L.de