

Frühlingsgefühle

Einführung in Spring.NET

Thomas Haug
thomas.haug@mathema.de

Spring.NET – In eigener Sache

- Thomas Haug
 - Software Entwickler seit 1984 ;-)
 - Entwicklung im Enterprise Umfeld seit 1998
 - Java und .Net Entwicklung als Senior Developer, Software Architekt und Teamleiter
 - Seit Oktober 2007 als Senior Consultant, Trainer und Architekt für MATHEMA Software GmbH tätig

Spring.NET - Agenda

- Motivation
- Überblick
- Spring.NET Grundlagen
 - Objekte erzeugen
 - Dependency Injection
 - Lebenszyklus
- Fortgeschrittene Spring.NET Programmierung
 - Persistenz und Transaktionen - Grundlagen
 - NHibernate Beispiel

Motivation

Spring.NET – Motivation

- Die Entwicklung von (Enterprise) Anwendungen ist eine anspruchsvolle Tätigkeit
 - Persistenzhandhabung
 - Transaktionsmanagement
 - Verteilungstechnologien
 - ...
- Je größer das zu entstehende System, desto weniger ist eine „ad hoc“ Verdrahtung von Software-Entitäten akzeptabel
 - Stattdessen Einsatz von Komponentenframeworks
 - Aber wie „invasiv“ ist das Komponentenframework?
 - Binden an ein Framework (bzw. Vendor-lockin)

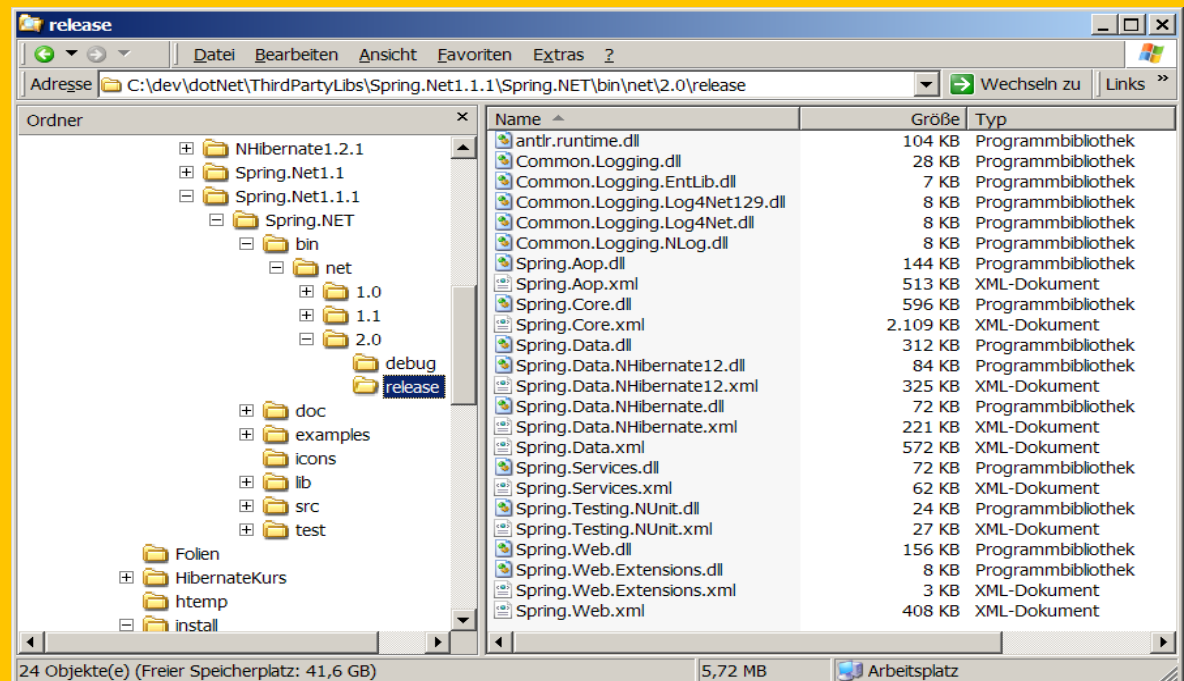
Überblick

Spring.NET – Überblick (1/8)

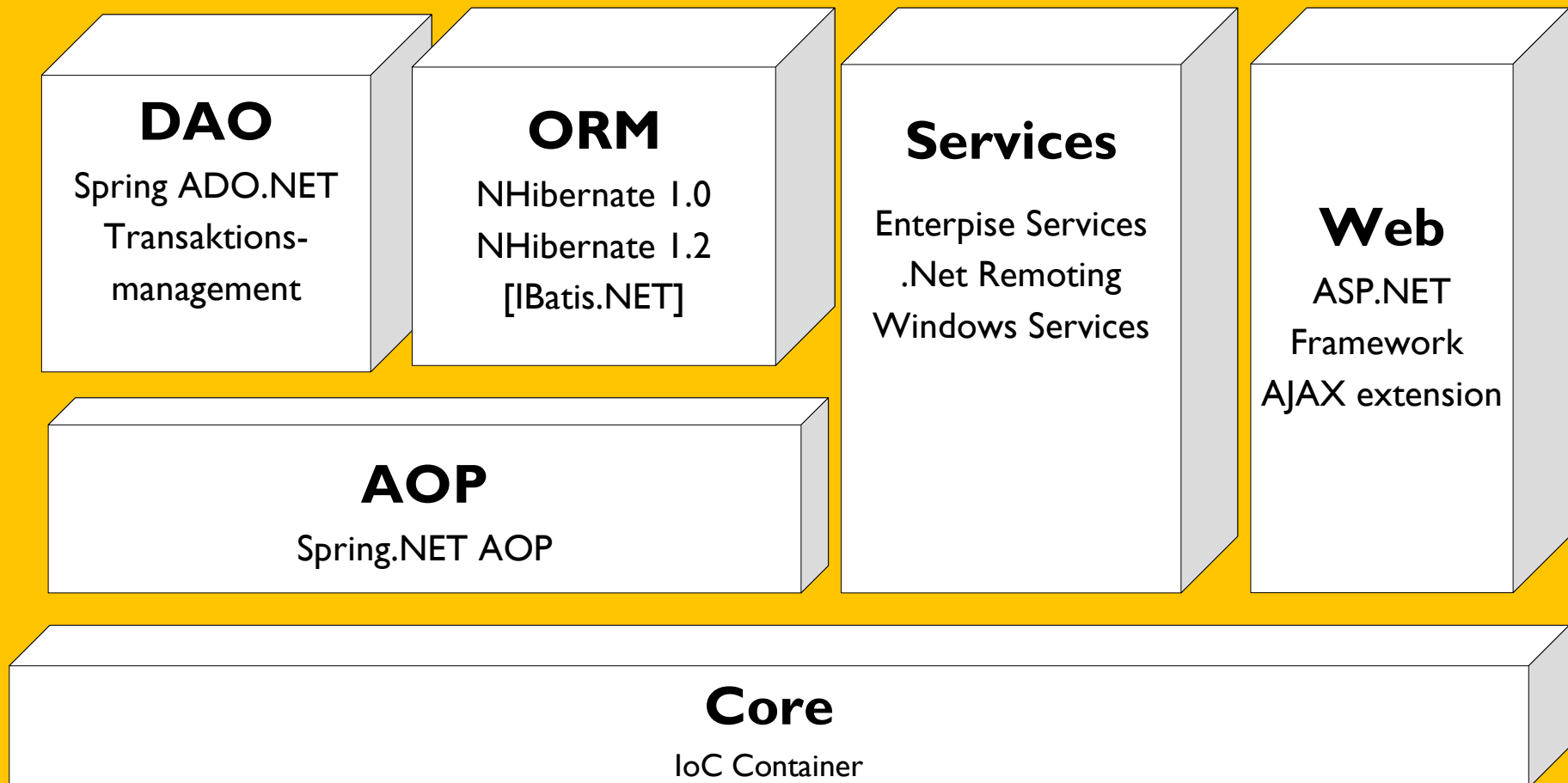
- Was ist Spring.NET
 - Ein „leichtgewichtiges“ Framework zum Erstellen von Enterprise .NET Anwendungen
 - Stellt einen so genannten Inversion of Control (IoC) Container zur Verfügung,
 - „non-invasive“, Code soll keine bzw. wenige Spring.NET Anteile verwenden
 - Stellt das Programmieren gegen Schnittstellen in den Vordergrund, wird aber nicht erzwungen
 - Abstrahiert von eingesetzten (Enterprise) Technologien (wie Persistenz und Verteilung), um
 - den Umgang mit diesen Technologien zu erleichtern
 - die Austauschbarkeit der Technologien zu ermöglichen

Spring.NET – Überblick (2/8)

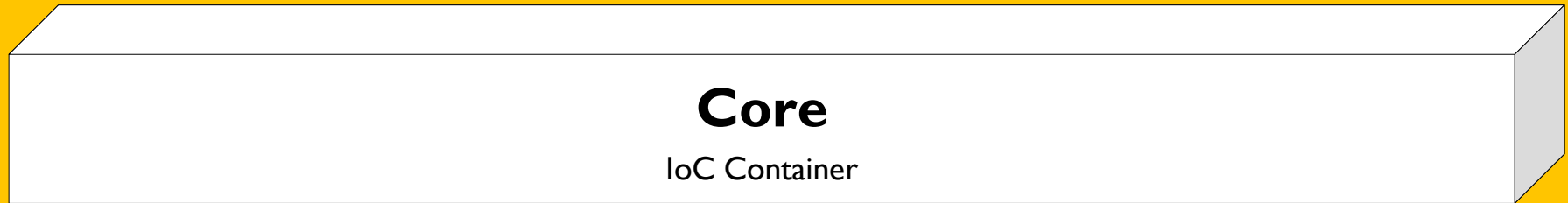
- Ist eine Portierung des Java Spring Frameworks
- <http://www.springframework.net/>
- Aktuelle Version 1.1.1
 - Installationsstruktur



Spring.NET – Überblick (3/8)

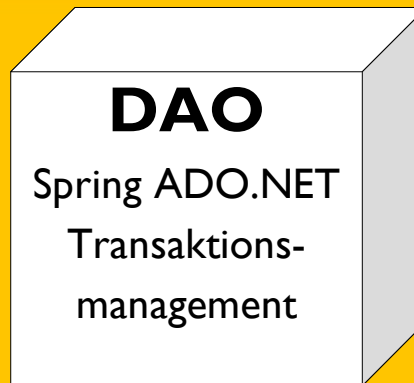


Spring.NET – Überblick (4/8)



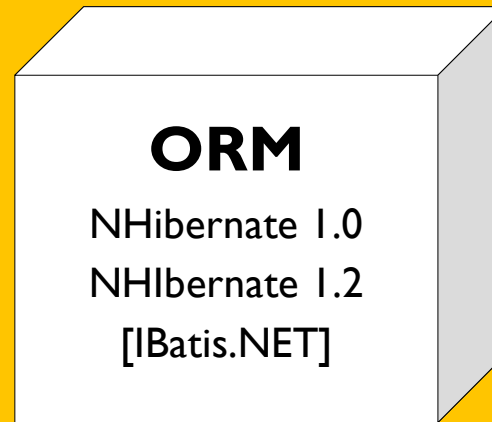
- Das Fundament des Spring.NET Frameworks
- Stellt den Komponenten Container und die Dependency Injection bereit
- Das grundlegende Konzept ist die `IObjectFactory` zum Erzeugen von Komponenten
- Das `IApplicationContext` erweitert die `IObjectFactory`
 - „Lokalisierte“ `MessageResources`
 - Unterstützung für Messaging (eigentlich Eventsystem)
 - Transparente Erstellung von kaskadierten Kontexten

Spring.NET – Überblick (5/8)



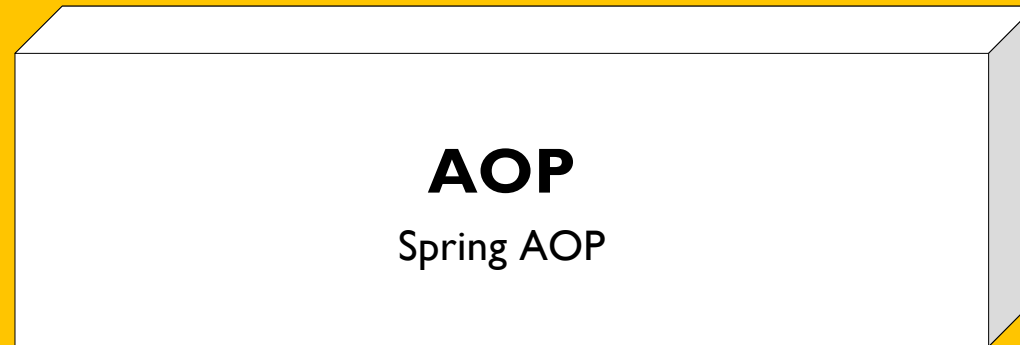
- Es erleichtert deutlich die Arbeit mit ADO.NET und Transaktionen
- Das DAO (Data Access Objects) Paket bietet eine Abstraktionschicht für ADO.NET
- Unterstützung für programmatisches als auch deklaratives Transaktionsmanagement

Spring.NET – Überblick (6/8)



- Integrationsbaustein für unterschiedliche Objekt-Relationale Mapping Werkzeuge:
 - NHibernate 1.0
 - NHibernate 1.2 (Achtung NHibernate 1.2.1 wird nicht unterstützt)
 - IBatis.NET (bisher nicht offiziell)

Spring.NET – Überblick (7/8)



- Aspect Oriented Programming (AOP) wird immer wichtiger
- Man benutzt AOP, um einzelne Teil der Programmlogik zu modularisieren
 - sog. **Cross cutting concerns (Querschnittsdienste)**
- Diese concerns werden an vielen Stellen einer Applikation verwendet, Beispiele sind Logging und Tracing

Spring.NET – Überblick (8/8)



- Vereinfacht die Entwicklung von
 - Enterprise Services
 - .Net Remoting Services und Web Services
 - Windows Services
- Spring.NET stellt entsprechende Exporter und Proxies zur Verfügung

Spring.NET Grundlagen

- Objekte erzeugen
- Dependency Injection
 - Lebenszyklus

Objekte erzeugen

- `IObjectFactory`
- `IApplicationContext`
- **Objekt Definition**

Spring.NET – ObjectFactory (I/2)

- Spring.NET instanziiert, managed und konfiguriert Objekte
 - Diese spiegeln sich in den Konfigurationsdaten der ObjectFactory wieder
 - Aber: nicht alle Abhängigkeiten sind als Konfigurationsdaten sichtbar, da manche aus programmatischen Interaktionen zur Laufzeit entstehen
- Eine ObjectFactory wird durch das Interface `Spring.Objects.Factory.IObjectFactory` repräsentiert

Spring.NET – ObjectFactory (2/2)

- Das Interface `IObjectFactory` stellt folgende Methoden bereit

```
namespace Spring.Objects.Factory
{
    public interface IObjectFactory : IDisposable
    {
        object this[string name] { get; }
        object ConfigureObject(object target, string name);
        bool ContainsObject(string name);
        string[] GetAliases(string name);
        object GetObject(string name);
        object GetObject(string name, object[] arguments);
        object GetObject(string name, Type requiredType);
        object GetObject(string name, Type requiredType,
            object[] arguments);
        Type GetType(string name);
        bool IsPrototype(string name);
        bool IsSingleton(string name);
    }
}
```

Spring.NET – IApplicationContext (1/3)

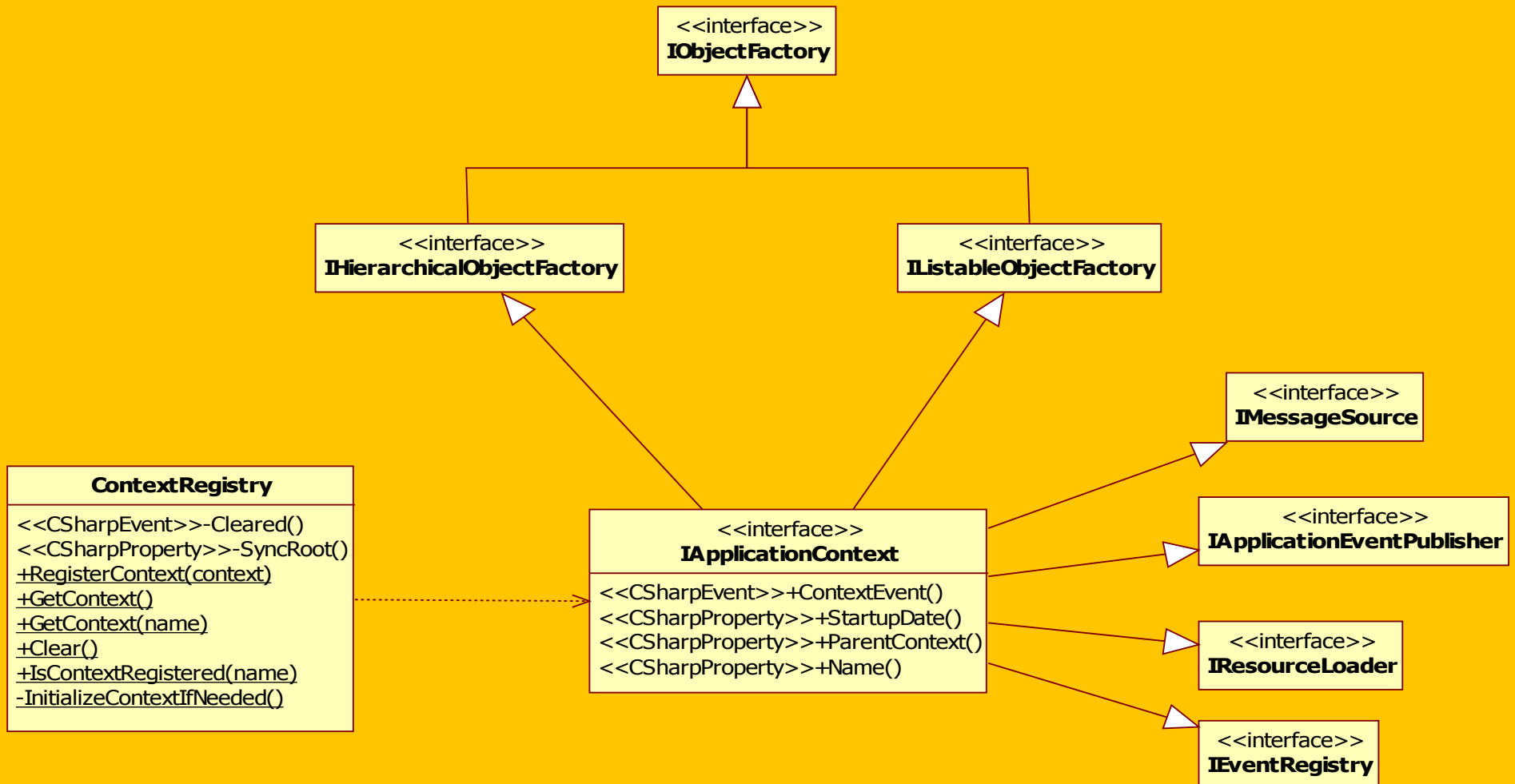
- Das Interface `IApplicationContext` ist eine Erweiterung der `IObjectFactory`
- Es unterstützt neben den Aufgaben der `IObjectFactory`:
 - Zugriff auf internationalisierbare Messages
 - Das Verteilen von Application-Events
 - Das generische Laden von Ressourcen

Spring.NET – IApplicationContext (2/3)

- IObjectFactory vs IApplicationContext
 - Die meisten Anwendungen nutzen den IApplicationContext
 - Nur 'kleinen' Applikationen dürfte die ObjectFactory ausreichen.
- Die `Spring.Context.SupportContextRegistry` kann als Factory von IApplicationContext genutzt werden

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler,
        Spring.Core" />
    </sectionGroup>
  </configSections>
  <spring>
    <context>
      <resource uri="file://config/services.xml"/>
    </context>
  </spring>
</configuration>
```

Spring.NET – IApplicationContext (3/3)



Spring.NET – XmlObjectFactory und XmlApplicationContext

- XmlObjectFactory und XmlApplicationContext sind die gängigste Realisierungen der IObjectFactory Schnittstelle
- Ist über eine XML-Datei konfigurierbar
- XML Datei muss dem Schema entsprechen

```
<?xml version="1.0" encoding="utf-8" ?>
<objects xmlns="http://www.springframework.net"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.springframework.net
         http://www.springframework.net/xsd/spring-objects.xsd">
  <description>meine erste Konfiguration</description>

  ...

</objects>
```

Spring.NET – Objekte definieren (I/4)

- XML Datei bestimmt wie eine Komponente mittels der `IObjectFactory` (`IApplicationContext`) erzeugt wird
- Unter Verwendung der `XmlObjectFactory` oder eines `XmlApplicationContext` lässt sich eine Spring.NET Objekt wie folgt erzeugen

```
<?xml version="1.0" encoding="utf-8" ?>
<objects xmlns="http://www.springframework.net"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.net
        http://www.springframework.net/xsd/spring-objects.xsd">
  <description>Beispiel 2</description>
  <object id="trainer" type="SpringBeispiel.Entities.Person,
                        SpringBeispiel" >
    <property name="nachname" value="Haug" />
  </object>
</objects>
```

Spring.NET – Objekte definieren (2/4)

- Implementierungsklasse muss einen Default-Konstruktor bereitstellen
- Das folgende Beispiel einer Objekt Definition spezifiziert, dass das Objekt durch Aufrufen einer Factory-Methode erzeugt werden soll:

```
</objects>  
  <object id="teilnehmer" type="SpringBeispiel.Entities.Person,  
        SpringBeispiel" factory-method="Create">  
    <property name="vorname" value="Ernie" />  
  </object>  
</objects>
```

- Das Attribut `factory-method` spezifiziert den Namen der statischen (!) Factory Methode
 - Spring.NET wird diese Methode bei `getObject()` aufrufen

Spring.NET – Objekte definieren (3/4)

- Verwendung von bestehenden Factory-Klassen in Spring.NET
 - Das `type` Attribut darf nicht gesetzt sein
 - `factory-object` und `factory-method` Attribut muss gesetzt sein

```
<objects>
  <object id="personFactory"
         type="SpringBeispiel.Entities.PersonFactory,
             SpringBeispiel">
  </object>

  <object id="teilnehmer2" factory-object="personFactory"
         factory-method="CreatePerson">
  </object>
</objects>
```

Spring.NET – Objekte definieren (4/4)

- Jedes Spring.NET Object hat eine oder mehrere Id's
 - Diese werden Identifier oder Namen genannt, wobei diese beiden Ausdrücke auf den selben Sachverhalt (nämlich die Identifikation des Objekts) verweisen
- Diese Id's müssen innerhalb der `IObjectFactory` oder des `IApplicationContext` **eindeutig sein**
- Ein Objekt hat in aller Regel nur eine Id
 - Hat es allerdings mehr als eine, so können die zusätzlichen Id's als Aliase verstanden werden

Spring.NET – Geltungsbereiche für Objekte (1/3)

■ Singleton

- Alle `getObject()` Anfragen an die `IObjectFactory` bzw. den `IApplicationContext` zu einem Objekt mit `singleton` Geltungsbereich haben zur Folge, dass ein und dasselbe Objekt zurückgeliefert wird
- Aber kein echtes Singleton, sondern pro `IObjectFactory`

■ Prototype

- Ist eine Objekt im `prototype` Geltungsbereich erstellt, so wird bei jedem Aufruf von `getObject()` an der `IObjectFactory` oder am `IApplicationContext` eine neue Objekt Instanz erzeugt und zurückgeliefert
- Objekte werden per Default im Singleton Geltungsbereich erzeugt

Spring.NET – Geltungsbereiche für Objekte (2/3)

■ Beispiel:

```
<object id="singletonObject"  
        type="Class1, SpringBeispiel" singleton="true" />  
<object id="prototypeObject"  
        type="Class1, SpringBeispiel" singleton="false" />
```

■ Achtung:

- Wenn ein Objekt als Prototyp deklariert wurde, ändert sich der Lebenszyklus geringfügig
- Spring.NET kann in diesem Fall nicht den gesamten Lebenszyklus des Objekts verwalten, da es nach ihrer Erstellung an den Client übergeben wird
- Der Container hat ab diesem Moment keinerlei Zugriff mehr auf das Objekt und hat somit keine Möglichkeit zu erkennen, wann dieses nicht mehr benötigt wird

Spring.NET – Geltungsbereiche für Objekte (3/3)

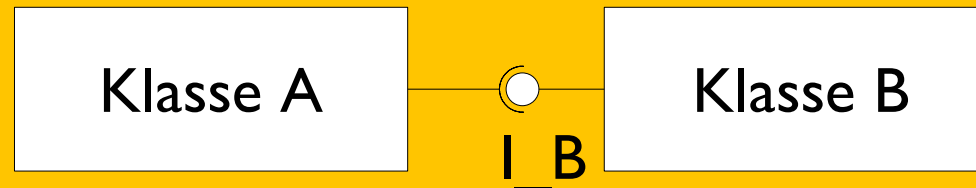
- Request (nur bei Webapplikationen)
 - Ist ein Objekt im `request` Geltungsbereich erstellt, so wird innerhalb einer HTTP Anfrage bei jedem `getObject()` Aufruf diese Instanz zurückgeliefert.
- Session (nur bei Webapplikationen)
 - Ist ein Objekt im `session` Geltungsbereich erstellt, so wird innerhalb einer HTTP Session bei jedem `getObject()` Aufruf diese Instanz zurückgeliefert.
- Application(nur bei Webapplikationen)
 - Ist ein Objekt im `application` Geltungsbereich erstellt, so wird innerhalb einer Webapplikation bei jedem `getObject()` Aufruf diese Instanz zurückgeliefert.
- Hinweis: Web Geltungsbereiche können nicht über die `IObjectFactory` oder den `IApplicationContext` abgefragt werden, es gibt lediglich `isSingleton()` und `isPrototype()`

Dependency Injection

- Warum DI
- Grundlagen
- DI in Spring
- Setter und Konstruktor-basierte DI

Spring.NET – Dependency Injection (1/9)

- In „traditionellen“ Programmen wird das Objektgeflecht im Programmcode zusammengefügt.
- Beispiel
 - Klasse A nutzt die Schnittstelle I_B, die von der Klasse B realisiert wird:



- Code:

```
// direkte Erzeugung
public class A {
    I_B iB = new B();
}
```

```
// Per Factory
public class A {
    I_B iB =
        I_B_Factory.getInstance();
}
```

- **Problem: A muss wissen wie I_B Instanzen erzeugt werden**

Spring.NET – Dependency Injection (2/9)

- Spring.NET reduziert durch Dependency Injection (DI) die Kopplung zwischen Komponenten (Objekten) und Ressourcen
- Dependency Injection wird häufig auch als Inversion of Control (IoC) tituliert und andersherum
 - Tatsächlich sind die beiden Ausdrücke aus der Sicht von Spring(.NET) als gleichbedeutend zu sehen
 - *IHMO zu kurz gegriffen: DI ist nur eine spezielle Form von IoC*
- Inversion of Control ist ein Grundprinzip von allen Frameworks

„Hollywood-Prinzip“

Don't call us, we call you!

Spring.NET – Dependency Injection (3/9)

- Das IoC-Pattern kontrolliert die Abhängigkeiten
 - Es übernimmt die Rolle des Hauptprogramms und koordiniert das Vernetzen der Objekte
 - IoC vereinfacht den Code damit drastisch
 - Der Code ist quasi „unverschmutzt“
- Man muss zwischen zwei IoC-Patterns differenzieren:
 - **Dependency Lookup**
 - Beispiel: EJB Container und CCM Container
 - **Dependency Injection**

Spring.NET – Dependency Injection (4/9)

■ Dependency Injection

- Abhängigkeiten werden von außen den Komponenten hineingereicht („injected“), anstatt sie von diesen selbst auffinden zu lassen
- Die Komponenten bringen ihre Abhängigkeit durch die Bereitstellung von bestimmten Methoden zum Ausdruck
 - setter-Methoden (Setter Injection)
 - spezifische Konstruktoren (Constructor Injection)
 - Field Injection – nicht mit Spring.NET

Spring.NET – Dependency Injection (5/9)

- Vorteile der Dependency Injection:
 - Abhängigkeiten werden durch Interfaces definiert
 - Auflösung der Abhängigkeiten zur Laufzeit
 - kein komplexer, rechenintensiver Lookup
 - „Non-invasive“, d.h. nahezu keine Änderungen an bestehenden Klassen
- Was kann DI zusätzlich bringen?
 - Verbessertes Design durch vermehrte Verwendung von Interfaces
 - Vereinfachte Testbarkeit
 - Vereinfachte Wartbarkeit
 - Wiederverwendbarkeit wird möglich

Spring.NET – Dependency Injection (6/9)

- Spring(.NET) empfiehlt zwar Setter basierte DI, bietet jedoch auch Unterstützung für Konstruktor basierter DI
 - falls man bereits existierende Klassen verwenden möchte, die keine entsprechenden Setter Properties unterstützen
- *Dies ist ein kontrovers diskutiertes Thema siehe*
 - <http://martinfowler.com/articles/injection.html>
 - „Best Practice“: möglichst viel zur Erzeugung des Objekts (Komponente) bereits initialisieren, deshalb Konstruktor basierte DI verwenden
 - Weiterer Vorteil keine Setter für unveränderbare Attribute

Spring.NET – Dependency Injection (7/9)

■ Setter-basierte Injection

```
class Person {  
    private Adresse adresse;  
    public Adresse Adresse {  
        get { return this.adresse; }  
        set { this.adresse = value; }  
    }  
}
```

```
<objects >  
    <object id="trainer" type="..." >  
        <property name="nachname" value="Bu" />  
        <property name="vorname" value="Hui" />  
        <property name="adresse" ref="adr" />  
    </object>  
    <object id="adr" type="..." >  
        <property name="strasse" value="Am Spuk 1" />  
        <property name="stadt" value="Gruselstadt" />  
        <property name="plz" value="12345" />  
    </object>  
</objects>
```

Spring.NET – Dependency Injection (8/9)

■ Konstruktor-basierte Injection

```
class Person {  
    public Person(string vorname, string nachname)  
        this.vorname = vorname;  
        this.nachname = nachname;  
    }  
}
```

```
<objects >  
    <object id="teilnehmer" type="..." >  
        <constructor-arg index="0" value="Thomas" />  
        <constructor-arg name="nachname" value="Haug" />  
        <property name="adresse" ref="adr" />  
    </object>  
  
    <object id="adr" type="..." >  
        <property name="strasse" value="Am Spuk 1" />  
        <property name="stadt" value="Gruselstadt" />  
        <property name="plz" value="12345" />  
    </object>  
</objects>
```

Spring.NET – Dependency Injection (9/9)

- Die Abhängigkeiten zwischen den Objekten werden wie folgt aufgelöst:
 - Konstruktor Argumente
 - Argumente einer Factory Methode
 - Properties, die in der Instanz gesetzt werden, nachdem diese erzeugt wurde
- Es ist die Aufgabe des Containers diese Abhängigkeiten zu „injizieren“, sobald der Container ein Objekt erzeugt
- Spring.NET Container kümmert sich um zirkuläre Abhängigkeiten, bei Konstruktor DI

Spring.NET – DI und leere Strings

- Spring.NET behandelt leere Argumente als Eigenschaften und desgleichen als leere Strings
- Beispiel:

```
<object type="...">  
  <property name="email">  
    <value></value>  
  </property>  
</object>
```

bzw.

```
<object type="...">  
  <property name="email"  
    value="" />  
</object>
```

- entspricht

```
exampleObject.setEmail( "" );
```

Spring.NET – DI und Null Werte

- Das `null` Element wird benutzt um Null-Werte zu behandeln
- Beispiel:

```
<object class="ExampleObject, ...">  
  <property name="email">  
    <null/>  
  </property>  
</object>
```

- entspricht dem C# Code:

```
exampleObject.setEmail(null);
```

Spring.NET – DI für Collections (I/2)

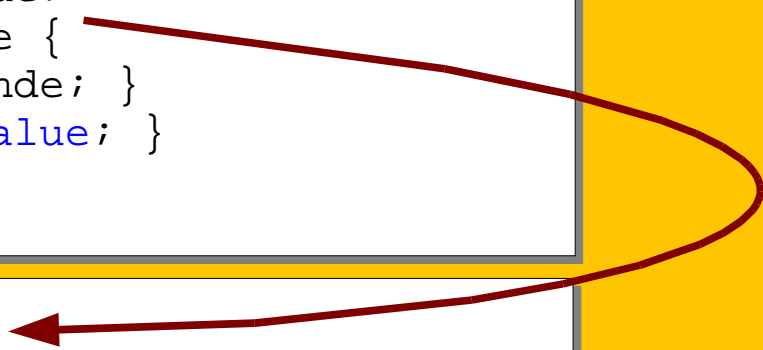
```
class Person {  
    private IList<string> hobbys;  
    public IList<string> Hobbys {  
        get { return this.hobbys; }  
        set { this.hobbys = value; }  
    }  
}
```

```
<object id="trainer" type="..." >  
    <property name="hobbys">  
        <!-- Angabe des Typs ermöglicht Generics einsatz-->  
        <list element-type="string">  
            <value>lesen</value>  
            <value>spuken</value>  
            <value>nix tun</value>  
        </list>  
    </property>  
</object>
```

Spring.NET – DI für Collections (2/2)

```
class Person {  
    private IDictionary besitzstaende;  
    public IDictionary Besitzstaende {  
        get { return this.besitzstaende; }  
        set { this.besitzstaende = value; }  
    }  
}
```

```
<object id="trainer" type="..." >  
    <property name="besitzstaende">  
        <dictionary>  
            <entry key="Auto" value="Porsche 356" />  
            <entry key="Autoersatzteile">  
                <dictionary>  
                    <entry key="Teil 1" value="Motor" />  
                </dictionary>  
            </entry>  
            <entry key="haus" value="Burg Ruine" />  
        </dictionary>  
    </property>  
</object>
```



Spring.NET – Anonyme Objekte

- Ein `object`-Element innerhalb eines `property` Elements definiert einen inneres Objekt

```
<object id="trainer" type="..." >  
  <property name="vorname" value="Hui" />  
  <property name="adresse">  
    <object type="..." >  
      <property name="strasse" value="Am Spukschloss 3" />  
      <property name="stadt" value="Grusselstadt" />  
      <property name="postleitzahl" value="12345" />  
    </object>  
  </property>  
</object>
```

- Zu beachten ist, dass das `singleton` und jegliches `id` Flag ignoriert werden (anonyme Prototypen)

Lebenszyklus steuern

Spring.NET – Den Lebenszyklus steuert (1/7)

- Es gibt zwei Spring.NET Schnittstellen, mittels denen ein Objekt über den Lebenszyklus informiert werden kann:
 - `IInitializingObject`
 - `IDisposable`
- Implementiert man `IInitializingObject`, so hat dies zur Folge das am Objekt die Methode `AfterPropertiesSet()` aufgerufen wird.
- Implementiert man `IDisposable`, so wird `Dispose()` aufgerufen, sobald das zugehörige `IApplicationContext` Objekt bereinigt (`Clear()`) wird

Spring.NET – Den Lebenszyklus steuert (2/7)

```
<<interface>>  
Factory::IInitializingObject  
+AfterPropertiesSet()
```

- Wenn nach der Erzeugung eines Objekts und dem Setzen dessen Properties spezifische Initialisierungen notwendig sind
- Zusätzlich kann diese Phase genutzt werden, um die gesetzten Properties zu prüfen (z.B. zum Gewährleisten, dass das Objekt einsetzbar ist)
- Zu beachten: die Verwendung von `IInitializingObject` führt zu einer Abhängigkeit zum Spring.NET Framework
- Im Falle der `XmlObjectFactory/XmlApplicationContext` kann durch die `init-method` Konfiguration diese Abhängigkeit vermieden werden

Spring.NET – Den Lebenszyklus steuert (3/7)

■ Beispiel mit `init-method` Attribut

```
<object id="exampleObject" type="..." init-method="Init"/>

public class ExampleObject {
    public void Init() {
        // do some initialization work
    }
}
```

■ Das obige Beispiel ist gleichbedeutend mit:

```
<object id="exampleObject" type="..."/>

public class ExampleObject : IInitializingObject {
    public void AfterPropertiesSet() {
        // do some initialization work
    }
}
```

Spring.NET – Den Lebenszyklus steuert (4/7)

```
<<interface>>  
IDisposable  
+Dispose()
```

- Die `Dispose` Methode am Ende des Lebenszyklus eines Objekts aufgerufen.
- Kann zum Bereinigen von benötigten Ressourcen genutzt werden
- Im Falle der `XmlObjectFactory/XmlApplicationContext` kann alternative die `destroy-method` Konfiguration genutzt werden
- Gilt nur für Singleton Objekte und `Clean()` muss am `IApplicationContext` ausgeführt werden

Spring.NET – Den Lebenszyklus steuert (5/7)

■ Beispiel mit destroy-method Attribut

```
<object id="exampleObject" type="..." destroy-method="Cleanup"/>

public class ExampleObject {
    public void Cleanup() {
        // do some cleanup work
    }
}
```

■ Das obige Beispiel ist gleichbedeutend mit

```
<object id="exampleObject" type="..."/>

public class ExampleObject : IDisposable {
    public void Disposit() {
        // do some cleanup work
    }
}
```

Spring.NET – Den Lebenszyklus steuert (6/7)

- Bei der Durchmischung der verschiedenen Init und Destroy Varianten gelten folgende Ausführungsreihenfolgen
- Initialisierung
 - 1) `AfterPropertiesSet()` der `IInitializingObject` Schnittstelle
 - 2) die mittels `init-method` konfigurierte Methode
- Zerstören
 - 1) `Dispose()` Methode der `IDisposable` Schnittstelle
 - 2) die mittels `destroy-method` konfigurierte Methode

Spring.NET – Den Lebenszyklus steuert (7/7)

- Zugriff auf Spring.NET Umgebung durch ein Spring.NET Objekt
 - `ObjectFactoryAware`

```
<<interface>>  
IObjectFactoryAware  
-<<CSharpProperty>>+ObjectFactory()
```

- `IObjectNameAware`

```
<<interface>>  
IObjectNameAware  
-<<CSharpProperty>>+ObjectName()
```

Fortgeschrittene Spring.NET Programmierung

Persistenz und Transaktionen

Grundlagen

Spring.NET – Persistenz (1/4)

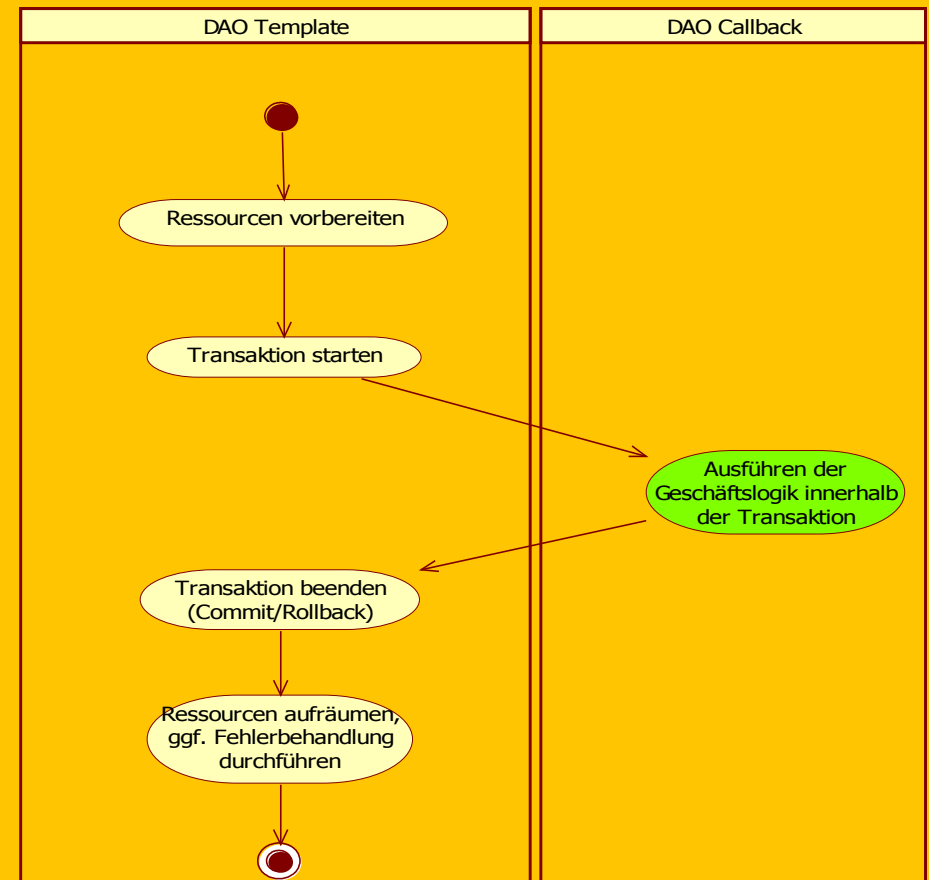
- Applikationen müssen oftmals Daten dauerhaft speichern
- Hierfür haben sich relationale Datenbank Managementsysteme (RDMS) durchgesetzt
 - XML Datenbanken und OO Datenbanken sind Nischenprodukte
- .Net bietet eine Vielzahl von unterschiedlichen Varianten, um relationale Datenbanken anzusprechen
 - ADO.NET
 - Object-Relational Mapping Produkte
 - LINQ

Spring.NET – Persistenz (2/4)

- Spring.NET versucht durch Vorgabe eines einfachen Programmiermodells den Anteil an technischen Code in einer Applikation zu reduzieren
- Hierzu bedient es sich dem Data Access Object (DAO) Pragadigma (Entwurfsmuster)
- Spring.NET DAO Templates unterstützen ADO.NET und NHibernate
- Exceptions der eingesetzten Persistenzschicht werden durch Spring.NET Exceptions gekapselt, um Unabhängigkeit zu ermöglichen

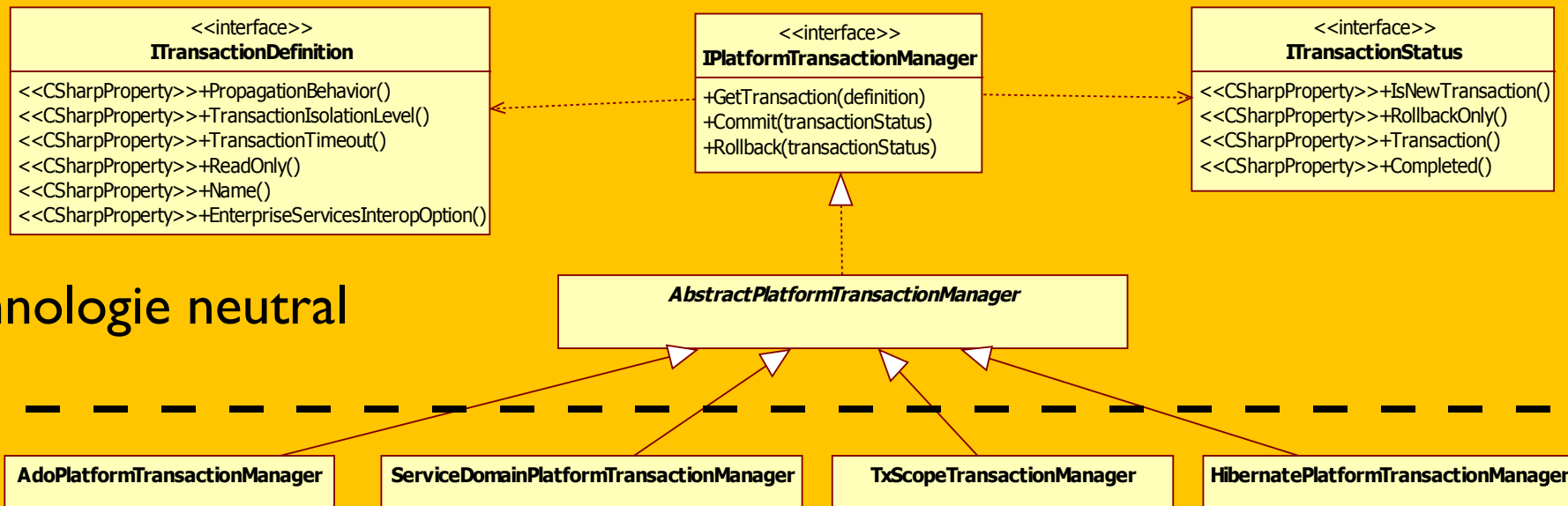
Spring.NET – Persistenz (3/4)

- Generelles Verhalten der DAO Templates basiert auf dem Template Entwurfsmuster (GOF Pattern)
- Spring.NET DAO Template kümmert sich um Verbindungsaufbau, -abbau, Transaktionssteuerung und Exception Handlung
- Spring delegiert den Applikations-spezifischen Anteil an einen sog. DAO Callback, der durch Entwickler bereitgestellt wird



Spring.NET – Transaktionen (I/3)

- Im Rahmen der Unabhängigkeit von Technologien (wie ADO.NET, NHibernate), bietet Spring.NET eine Transaktionsabstraktionsschicht, die eine einheitliche Programmierschnittstelle bereitstellt:

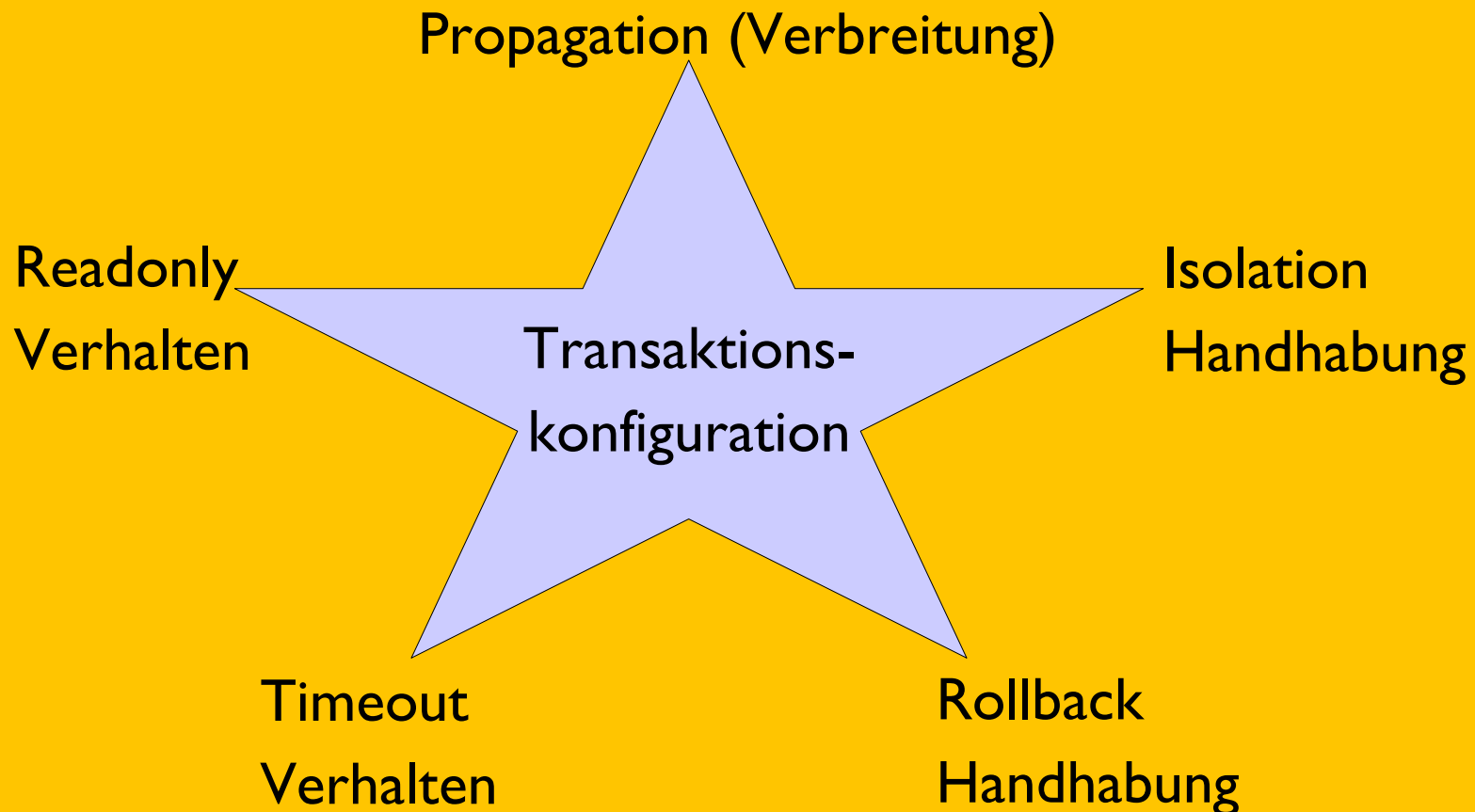


Technologie neutral

Technologie abhängig

Spring.NET – Transaktionen (2/4)

■ Konfigurationsdimensionen



Spring.NET – Transaktionen (2/3)

- Nur die Konfiguration eines Transaktionsmanagers ist abhängig von der eingesetzten Technologien, nicht der Programmcode
- Beispiel ADO.Net Konfiguration

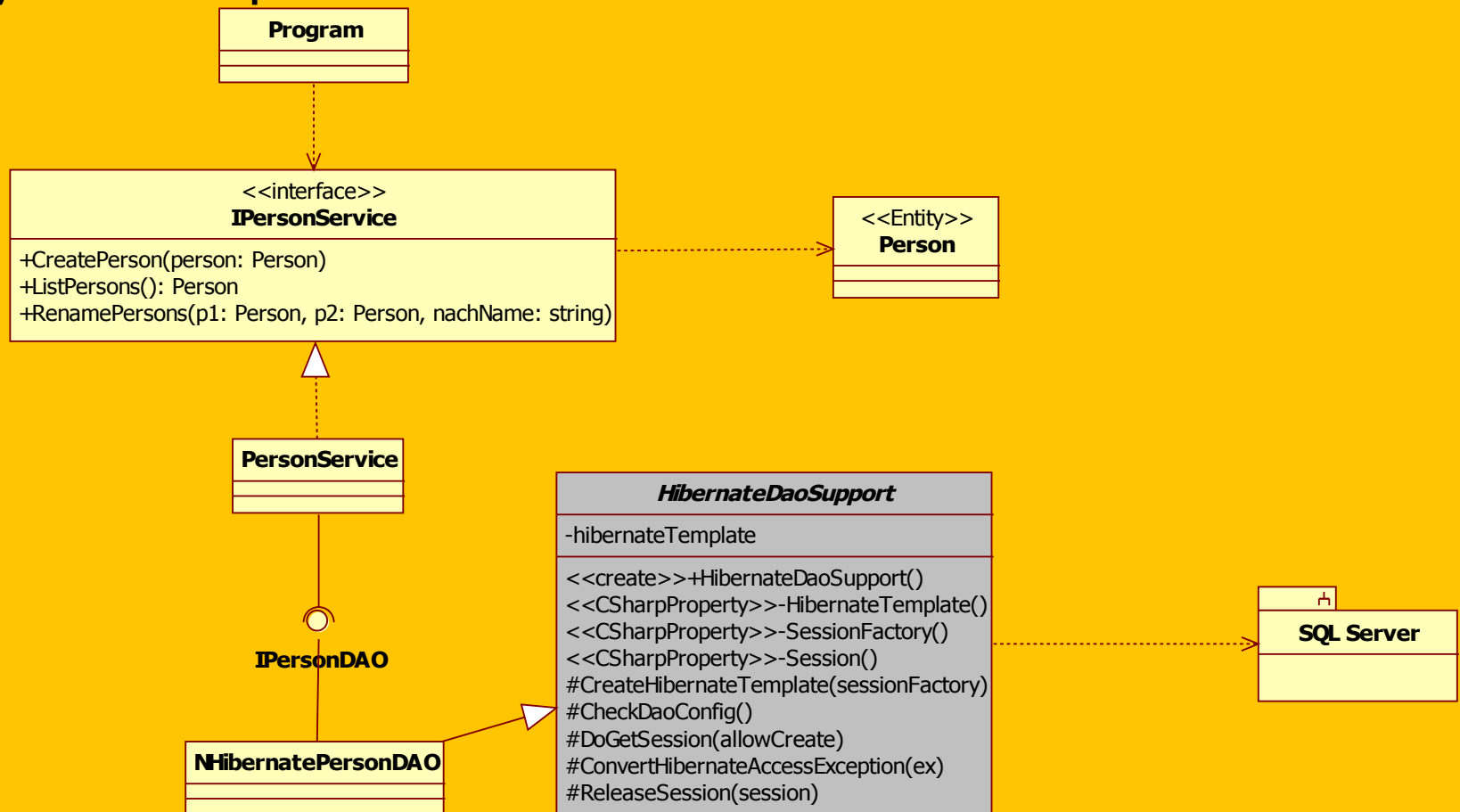
```
<objects xmlns="http://www.springframework.net "  
    xmlns:db="http://www.springframework.net/database">  
  <db:provider id="DbProvider"  
    provider="System.Data.SqlClient"  
    connectionString="server=Haug\SQLEXPRESS;  
                    Integrated Security=SSPI;  
                    database=DEMO_DB" />  
  <object id="AdoTransactionManager"  
    type="Spring.Data.AdoPlatformTransactionManager,  
        Spring.Data">  
    <property name="DbProvider" ref="DbProvider" />  
  </object>  
</objects>
```

Spring.NET – Transaktionen (3/3)

- Spring.NET unterstützt programmatische als auch deklarative Transaktionshandhabung
- Deklarative Transaktionssteuerung kann über
 - Methoden Attribute (Annotationen) als auch
 - XML Konfiguration durchgeführt werden
 - Nutzt generell Spring AOP Mechanismen (AutoProxying)

Spring.NET – Persistenz Beispiel (1/7)

■ System Setup



Spring.NET – Persistenz Beispiel (2/7)

■ Konfiguration (services.xml)

Achtung nur ein \"

```
<objects xmlns="http://www.springframework.net"
  xmlns:db="http://www.springframework.net/database">
  <db:provider id="DbProvider"
    provider="System.Data.SqlClient"
    connectionString="server=haug\sqlexpress;Integrated
      Security=SSPI; database=DEMO_DB"/>
  <object id="SessionFactory"
    type="Spring.Data.NHibernate.LocalSessionFactoryObject,
      Spring.Data.NHibernate12">
    <property name="DbProvider" ref="DbProvider" />
    <property name="MappingAssemblies">
      <list>
        <value>SpringBeispiel.Entities</value>
      </list>
    </property>
    <property name="HibernateProperties">
      <dictionary>
        <entry key="hibernate.connection.provider"
          value="NHibernate.Connection.DriverConnectionProvider" />
        <entry key="hibernate.dialect"
          value="NHibernate.Dialect.MsSql2000Dialect" />
        <entry key="hibernate.connection.driver_class"
          value="NHibernate.Driver.SqlClientDriver" />
      </dictionary>
    </property>
  </object>
```

Spring.NET – Persistenz Beispiel (3/7)

■ Konfiguration (services.xml, Fortsetzung)

```
<object id="HibernateTransactionManager"
    type="Spring.Data.NHibernate.HibernateTransactionManager,
        Spring.Data.NHibernate12">
    <property name="DbProvider" ref="DbProvider" />
    <property name="SessionFactory" ref="SessionFactory" />
</object>

<object id="HibernateTemplate"
    type="Spring.Data.NHibernate.HibernateTemplate,
        Spring.Data.NHibernate12">
    <property name="SessionFactory" ref="SessionFactory" />
    <property name="TemplateFlushMode" value="Auto" />
</object>

<object id="PersonDao"
    type="...">
    <property name="HibernateTemplate" ref="HibernateTemplate" />
</object>

<object id="PersonService" type="...">
    <property name="PersonDAO" ref="PersonDao" />
</object>

<tx:attribute-driven transaction-manager="transactionManager"/>
</objects>
```

Spring.NET – Persistenz Beispiel (4/7)

■ NHibernatePersonDAO Implementierung

```
using Spring.Data.NHibernate.Generic.Support;

...

class NHibernatePersonDAO : HibernateDaoSupport, IPersonDAO {
    public void SavePerson(SpringBeispiel.Entities.Person person) {
        //wirft eine Spring.Dao.DataAccessException
        //falls ein Fehler auftritt
        this.HibernateTemplate.Save(person);
    }

    public IList<Person> ListPersons() {
        return HibernateTemplate.LoadAll<Person>;
    }

    public void UpdatePerson(Person person) {
        this.HibernateTemplate.Update(person);
    }
}
```

Spring.NET – Persistenz Beispiel (5/7)

■ PersonServiceImpl Implementierung (Auszug)

```
using Spring.Transaction.Interceptor;
...
class PersonServiceImpl : IPersonService {
    ...

    [Transaction(Timeout=1000)]
    public void RenameLastName(Person p1, Person p2,
                               string lastName) {
        p1.Nachname = lastName;
        p2.Nachname = lastName;
        this.personDAO.UpdatePerson(p1);
        this.personDAO.UpdatePerson(p2);
    }

    [Transaction(ReadOnly=true)]
    public IList<Person> ListPersons()
    {
        return this.personDAO.ListPersons();
    }
}
```

Spring.NET – Persistenz Beispiel (6/7)

■ Hauptprogramm

```
class Program {
    static void Main(string[] args) {
        IApplicationContext applCtx = ContextRegistry.GetContext();
        IPersonService personService = (IPersonService)
            applCtx.GetObject("PersonService");

        Person ernie = new Person("Ernie", "Blabla");
        Person bert = new Person("Bert", "Blabla");

        personService.CreatePerson(ernie);
        personService.CreatePerson(bert);

        personService.RenameLastName(ernie, bert, "Hui");

        IList<Person> persons= personService.ListPersons();
        foreach (var item in persons) {
            Console.WriteLine("Person :" + item);
        }
    }
}
```

Spring.NET – Persistenz Beispiel (7/7)

■ Konfiguration Main Programm (App.config)

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name="parsers"
        type="Spring.Context.Support.NamespaceParsers-
          SectionHandler, Spring.Core" />
      <section name="context"
        type="Spring.Context.Support.ContextHandler,
          Spring.Core" />
      <section name="objects"
        type="Spring.Context.Support.DefaultSectionHandler,
          Spring.Core" />
    </sectionGroup>
  </configSections>

  <spring>
    <parsers>
      <parser type="Spring.Data.Config.DatabaseNamespaceParser,
        Spring.Data" />
      <parser type="Spring.Transaction.Config.TxNamespaceParser, ..." />
      <parser type="Spring.Aop.Config.AopNamespaceParser, Spring.Aop" />
    </parsers>
    <context name="Beispiel">
      <resource uri="file://config/services.xml"/>
    </context>
  </spring>
</configuration>
```

Fazit

Spring.NET – Fazit

- Interessantes Framework (nicht schon wieder einen eigenen Container implementieren ...)
- Einstieg ist leicht (, sofern man bereits mit Spring (Java) gearbeitet hat)
- XML Konfiguration kann schnell unübersichtlich werden
 - Konfigurationen in unterschiedliche Dateien auslagern (z. B. an den Layern des zu entwerfenden Systems orientieren)
- In diesem Vortrag wurde nur kleiner Ausschnitt betrachtet, es gibt noch viel mehr ...
- Arbeiten mit Spring.NET macht Spaß

Spring.NET – Fragen

Vielen Dank!

thomas.haug@mathema.de