Das Spring Framework – eine Einführung

Sebastian Schelter
Softwareengineering
Freie Universität Berlin
Arnimallee 14
14195 Berlin
sebastian@alombra.de

Abstract: Eine Einführung in die grundlegenden Konzepte des Spring Frameworks mit Schwerpunkt auf Dependency Injection und der Darstellung einiger exemplarischer Anwendungsbeispiele

Inhaltsverzeichnis **Seite**

I. Überblick und Geschichte
1. Überblick 2
2. Geschichte 2
II. Grundlegene Konzepte
1. Module des Spring Frameworks 3
2. POJOs 3
3. Dependency Injection 4
4. AOP 8
5. Templates 9
III. Beispiele
1. Unit Testing 10
2. JMS Integration 11
3. Transparentes Caching mit AOP 13
IV. Quellen 14

I. Überblick und Geschichte

1. Überblick

Spring ist ein OpenSource Applikationsframework für die Java Plattform, insbesondere für J2EE. Seine Entstehung ist auf die Frustration mit der oft unnötigen Komplexität bestehender J2EE Architekturen zurückzuführen, insbesondere auf die von Enterprise JavaBeans und Application Servern. Spring stellt in seiner Gesamtheit eine leichtgewichtige Alternative zu diesen dar.

Laut seinen Entwicklern bietet Spring die folgenden Vorteile bei der Anwendungsentwicklung:

- effektive Verwaltung der Objekte in der Mittelschicht
- es müssen keinerlei Singletons mehr per Hand geschrieben werden
- es müssen keinerlei eigene Konfigurationsformate mehr erdacht und implementiert werden
- der Aufwand gegen Interfaces zu programmieren wird fast auf null reduziert
- die Anwendung selbst wird so gut wie nichts von der Spring API implementieren müssen
- Applications built using Spring are very easy to test.
- die entstehende Infrastruktur wird die leichtgewichtigste sein, die möglich ist

Mit welchen Mitteln dies erreicht werden soll, werde ich im zweiten Abschnitt ausführlich erläutern.

2. Geschichte

In seinem im Oktober 2002 erschienen Buch "Expert One-on-One J2EE Design and Development" stellt Rod Johnson die Grundlagen und Prinzipien eines alternativen J2EE Frameworks vor. Ein Grundgerüst des Frameworks ist dem Buch bereits beigelegt. Das Interesse ist so groß, dass das Framework als OpenSource Projekt weiterentwickelt wird und daraufhin im Juni 2003 die Version 1.0 von Spring erscheinen kann.

Im Juni 2006 erscheint dann die Version 2.0, Spring gewinnt in diesem Jahr den "JAX Innovation Award" und ist bis zu diesem Zeitpunkt bereits mehr als 1 Million mal aus dem Internet geladen worden.

Der letzte grundlegende Release war im November 2007, als Spring 2.5 erschien, die aktuelle Version.

II. Grundlegende Konzepte

1. Module des Spring Frameworks

Spring ist streng modular aufgebaut mit möglichst wenig Abhängigkeiten zwischen den einzelnen Modulen. Das hat zur Folge, das der Entwickler nur die Teile des Frameworks einzubinden hat, die er wirklich benutzen möchte.

Die Module im Einzelnen:

Core	der Dependency Injection Container (grundlegend für alle anderen Module)
AOP	Unterstützung für aspektorientierte Programmierung
JEE	Untersützung für diverse JEE APIs, wie beispielsweise JMS, EJB, Email
Web	Spring eigenes MVC Framework, Untersützung für diverse andere Web Frameworks und Technologien wie Struts, Tapestry, JSF, Unterstützung für Templating Technologien wie JSP, Velocity, Freemarker
DAO	Datenzugriff per JDBC und Transaktionsmanagement
ORM	Einbindung etablierter Bibliotheken für objektrelationes Mapping wie Hibernate, iBatis oder TopLink

2. POJOs

Nun zur Frage, wie die Klassen unserer springbasierten Anwendung aussehen müssen. Die Anwort ist einfach, es können simple POJOs sein.

Unter "POJOs" versteht man Java Klassen, die von keiner vorgebenen Elternklasse erben, kein vorgegebenes Interface implementieren und keine bestimmten Annotations besitzen müssen. Sie sind somit die einfachste Form objektorientierten Designs.

Wegen ihrer Unabhängigkeit von der Einbettung in bestehende Laufzeitsysteme stellen sie einen radikalen Gegenentwurf zu den Enterprise Java Beans aus der J2EE Spezifikation 2.1 dar, die aus sehr vielen komplexen Einzelteilen zu bestehen hatten.

Mithilfe des Spring Containers können aus POJOs bestehende Anwendungen in beliebigen Umgebungen lauffähig gemacht werden, beispielsweise in J2EE Containern, Servlet Containern, Unit Tests und Shell Anwendungen.

Die Technik, die dies ermöglicht wird nun im nächsten Abschnitt vorgestellt.

3. Dependency Injection

Anwendungsbeispiel

Die Technik und die Problemstellung, die sie löst, soll an einem Beispiel erläutert werden:

Wir stellen uns vor, wir hätten eine Anwendun zu entwickeln, die Filme eines bestimmte Regisseurs aus einer Datenbank oder von einem Webservice liest. Sie soll aus 2 Komponenten bestehen, dem **MovieFinder**, der alle vorhandenen Filme liest und dem **MovieLister**, der dann die zu einem Regisseur gehörigen auswählt.

```
public class Movie implements Serializable {
     private String name;
     private Director director;
...
}

public interface MovieFinder {
     public Movie[] findMovies();
}

public class MovieFinderImpl implements MovieFinder {
     public Movie[] findMovies() {
          // Lesen der Filme aus einer Datenbank oder von einem Webservice
     }
}
```

Die entscheidende Frage lautet nun: *Wie bekommt der MovieLister die Implementierung des MovieFinders?* Allgemein ausgedrückt: wie wird die Abhängigkeit zwischen den Komponenten aufgelöst?

Naive Implementierung

Eine erste, naive Implementierung könnte wie folgt aussehen:

```
public class MovieLister {

     public List<Movie> listMovies( String directorName ) {

          List<Movie> movies = new LinkedList<Movie>();
          MovieFinder movieFinder = new MovieFinderImpl();

          for( Movie movie : movieFinder.findMovies() ) {
               if ( movie.getDirector().getName().equals( directorName ) ) {
                    movies.add( movie );
               }
          }

          return movies;
     }
}
```

Problematisch hierbei ist jedoch die Tatsache, dass die beiden Komponenten nun direkt im Code gekoppelt sind (schlechtes Softwaredesign) und dass die Klasse auch nur sehr schwer testbar ist, da man die Implementierung des MovieLister nicht gegen eine einfache Testimplementierung austauschen kann.

Klassische J2EE Lösung: Service Locator Pattern

Die klassische Lösung für dieses Problem ist der sogenannte ServiceLocator Pattern, der wie folgt implementiert werden würde:

```
public class ServiceLocator {
     ...
     public Object getService( String serviceName ) {
          if ( "movieFinder".equals( serviceName ) ) {
               // Holen per JNDI
          }
     }
}
public class MovieLister {
     public List<Movie> listMovies( String directorName ) {
          ServiceLocator locator = ServiceLocator.getInstance();
          MovieFinder finder = (MovieFinder)locator.getService("movieFinder");
          ...
     }
}
```

Damit erreicht man zwar eine Entkopplung der beiden Klassen, jedoch hat auch diese Vorgehensweise diverse Nachteile.

Es ist nun eine zusätzliche Klasse notwendig, die nichts mit unserem eigentlichen Anwendungsproblem zu tun hat, sondern nur der Architektur dient und unser MovieLister muss nun den ServiceLocator kennen und benutzen können.

Außerdem ist die Testbarkeit nun zwar theoretisch gewährleistet, jedoch muss der Entwickler dafür sorgen, dass die vom ServiceLocator zurückgelieferten Implementierungen auch gegen einfache Testklassen austauschbar sind.

Dependency Injection

Eine elegante Lösung der Problematik der Auflösung von Abhängigkeiten bietet die sogenannte Dependency Injection. Grundlage ist das sogenannte Hollywoodprinzip: „Don't call us, we call you“. Das bedeutet, dass unsere Klasse die benötigten Komponenten von außen injiziert bekommt, sie muss also nur noch wissen, was sie braucht und sich nicht mehr selbst darum kümmern, wie sie die benötigten Komponenten bekommt. Der Dependency Injection Container garantiert, dass die Abhängigkeiten zur Laufzeit gesetzt werden und entkoppelt dadurch die Klasse von ihrer Laufzeitumgebung.

Technisch funktioniert Dependency Injection so, dass die Klasse eine Membervariable für die benötigte Komponente beinhaltet und ein Konstruktorargument oder eine Setter-Methode bereitstellt, über die die Komponente dann injiziert werden kann:

```
public class MovieLister {
     private MovieFinder movieFinder;
     ...
}
```

Übergabe durch den Konstruktor (Constructor Injection):

```
public MovieLister( MovieFinder finder ) { this.movieFinder = finder; }
```

Übergabe durch eine Setter-Methode (Setter Injection):

```
public void setMovieFinder( MovieFinder finder ) { this.movieFinder = finder; }
```

Vergleich verschiedener DI-Techniken

Beide Techniken werden von fast allen DI-Containern unterstützt und haben folgende Vor- und Nachteile:

Typ	Vorteile	Nachteile
Constructor Injection	• Klasse immer in gültigem Zustand • wenig Code notwendig	• nicht kompatibel zum Java Beans Standard • viele Argumente bei vielen Abhängigkeiten • Konstruktorargumente müssen in Subklassen übertragen werden • für Tests müssen alle Abhängigkeiten gesetzt werden
Setter Injection	• kompatibel zum JavaBeans Standard • für Tests können einzelne Abhängigkeiten gesetzt werden	• Klasse kann theoretisch in ungültigem Zustand sein • viel Code notwendig

Konfiguration des Beispiels mit Spring

In Spring erfolgt die Konfiguration der Komponenten über eine XML-Datei, in der die „Beans“ genannten Komponenten und ihre Abhängigkeiten zueinander deklariert werden. Spring liest dann beim Anwendungsstart diese Datei, erzeugt die „Beans“ (standardmäßig als Singletons) und verknüpft sie miteinander.

Deklaration des MovieFinders:

```
<bean id="movieFinder" class="de.fuberlin.spring.MovieFinderImpl"/>
```

Deklaration des MovieListers mit Setter-Injection

```
<bean id="movieLister" class="de.fuberlin.spring.MovieLister">
     <property name="movieFinder" ref="movieFinder"/>
</bean>
```

Deklaration des MovieListers mit Constructor-Injection

```
<bean id="movieLister" class="de.fuberlin.spring.MovieLister">
     <constructor-arg ref="movieFinder"/>
</bean>
```

Allgemeine Konfigurationsmöglichkeiten mit Spring

Mithilfe des XML Formates von Spring können auch komplexe Datenstrukturen wie Enums, Properties, Lists und Maps abgebildet werden, im Allgemeinen kann man sagen, dass man damit beliebige JavaBeans erzeugen lassen kann.

```
<bean id="moreComplexObject" class="example.ComplexObject">
  <property name="adminEmails">
    <props>
        <prop key="administrator">administrator@example.org</prop>
        <prop key="support">support@example.org</prop>
    </props>
  </property>
  <property name="someMap">
    <map>
        <entry>
            <key><value>a ref</value></key>
            <ref bean="myDataSource" />
        </entry>
    </map>
  </property>
</bean>
```

Da das XML Format schnell unübersichtlich werden kann, bietet Spring auch noch die Möglichkeit der Konfiguration über Annotations, die jedoch weit weniger Datenstrukturen unterstützt als die herkömmliche XML-Konfiguration:

@Component / @Service / @Repository
markiert Klasse als Spring-Bean

@Autowired
verlangt Injection

@Qualifier
trifft Auswahl bei mehreren Injection-Kandidaten

```
@Service
public class MovieLister {

     @Autowired
     @Qualifier("movieFinder")
     private MovieFinder movieFinder;
}
```

4. AOP

Wir haben nun gesehen, dass sich mithilfe von Spring Anwendungen aus POJOs unabhängig von der Laufzeitumgebung bauen und durch Dependency Injection konfigurieren lassen können, jedoch gibt es Problemfelder moderner betrieblicher Informationssysteme, die sich mit diesen beiden Techniken noch nicht elegant genügend abdecken lassen.

Als Beispiel hierfür soll die Transaktionalität einer Anwendung angeführt werden. Diese erstreckt sich über mehrere Schichten, da die Transaktionen in der An-wendungsschicht koordiniert werden müssen und die Operationen in der Persistenz-schicht müssen sich an diesen beteiligen. Das heißt ohne weitere Hilfsmittel müsste diese Funktionalität hart an den betreffenden Stellen in der Anwendung kodiert werden, wünschenswert wäre jedoch ein Mechanismus, mit dem man die Transaktionalität nach Fertigstellung der Anwendung konfigurieren kann.

Es handelt sich bei dieser Anforderung um einen sogenannten "Cross Cutting Concern", einen querschnittlichen Belang, der nicht in einer Schicht modular gekapselt werden kann.

Die technische Lösung dafür nennt sich AOP, aspektorientierte Programmierung, mithilfe derer man diese Belange gekapselt modellieren und deklarativ verschiedenen Codefragmenten zuordnen kann.

Das Spring Framework bietet Unterstützung für AOP an, diese ist nur auf Methodenaufrufe anwendbar und funktioniert folgendermaßen:

Die Beans werden im Spring Container noch vor den Injektion mit "Proxies" versehen, die die zusätzliche Funktionalität beinhalten und vor den eigentlichen Methodenaufrufen ausführen (dieser Ansatz ist dem "Decorator"-Pattern sehr ähnlich). Für den benutzenden Code geschieht dies weitgehend transparent.

Die Konfiguration erfolgt über Annotations oder per Spring XML.

5. Templates

"Templates" sind einfache Hilfsklassen, wie sie sich jeder Entwickler, der über einen längeren Zeitraum mit einer bestimmten Technologie arbeitet, schon selbst ge-schrieben hat.

Das Spring Framework bietet solche Hilfsklassen für eine Vielzahl von APIs wie beispielsweise JMS, JDBC oder Hibernate. Diese "Templates" bieten automatisches Ressourcenmanagement, einheitliche Fehlerbehandlung und meist auch eine Verein-fachung der benutzten Schnittstelle.

Als Beispiel sei eine Methode aus dem JMSTemplate aufgeführt:

```
public void convertAndSend( String destinationName, Object message );
```

Diese Methode schickt eine JMS Nachricht an die unter dem Namen angegebene Warteschlange, das Nachrichtenobjekt wird automatisch konvertiert und die Verbindung zum JMS Broker ohne Zutun des aufrufenden Codes verwaltet.

III. Beispiele

Im Folgenden sollen nun an einigen kurzen Beispielen gezeigt werden, wie die erläuterten Techniken in der Praxis umgesetzt werden können.

1. Unit Testing

POJO Modellierung und Dependency Injection ermöglichen den einfachen Austausch von Komponenten für Testzwecke. Anstatt der realen Implementierung des MovieFinders (der auf eine externe Datenbank zugreift, die verfügbar sein muss), die für unsere Tests ungeeignet wäre, da sie auf die reale Infrastruktur der Anwendung zurückgreift, bauen wir uns eine kleine Testimplementierung, die nur 2 Filme zurückliefert.

```
public class MockMovieFinder implements MovieFinder {
     public Movie[] findMovies() {
          return new Movie[] {
               new Movie( "Lord of the Rings", new Director( "Peter Jackson" ) ),
               new Movie( "Batman Begins", new Director( "Christopher Nolan" ) )
          };
     }
}
```

In unserem Test können wir nun die Implementierung des MovieListers erzeugen und per Hand unsere Testklasse injizieren. Dadurch bekommen wir einen schnell und ohne zusätzliche externe Infrastrukrur ausführbaren Test.

```
@Test
public void some() {

     MovieLister movieLister = new MovieLister();

     movieLister.setMovieFinder( new MockMovieFinder() );

     List<Movie> movies = movieLister.listMovies( "Christopher Nolan" );

     assertEquals( 1, movies.size() );
     assertEquals( "Christopher Nolan", movies.get(0).getDirector().getName() );
}
```

2. JMS Integration

In diesem Beispiel sollen nun die Movie Objekte asynchron über eine JMS Queue verschickt werden. Als JMS Implementierung wird Apache ActiveMQ genutzt und wir gehen davon aus, dass lokal eine Brokerinstanz läuft.

In der Spring Konfiguration müssen der Broker und die Queue folgendermaßen bekannt gemacht werden:

```
<amq:connectionFactory id="cFactory" brokerURL="tcp://localhost:61616"/>

<amq:queue id="movieQueue" physicalName="movieQueue"/>
```

Desweiteren sollte man das Spring eigene JMS Template benutzen, da es einem die Arbeit mit der eigentlichen JMS API abnimmt:

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
     <constructor-arg ref="cFactory" />
</bean>
```

Das Verschicken per JMS kann nun mit wenigen Zeilen Code implementiert werden, man braucht nur ein Exemplar des gerade konfigurierten JMS Templates und muss darauf die Methode convertAndSend mit dem Namen der Queue und dem Movie Objekt als Parameter aufrufen. Da die Movie Klasse das Interface Serializable implementiert, benutzt das JmsTemplate automatisch ObjectMessages und konvertiert das Movie Exemplar automatisch.

```
@Component
public class MoviePublisher {
...
     @Autowired
     private JmsTemplate jmsTemplate;

     public void publish() {
          for ( Movie movie : movieLister.listMovies( "Peter Jackson" ) ) {
               jmsTemplate.convertAndSend( "movieQueue", movie );
          }
     }
}
```

Der Empfang der JMS Nachrichten gestaltet sich noch viel einfacher, man benötigt eine Klasse, die eine Methode besitzt, die nur das Movie Objekt als Parameter bekommt:

```
@Component
public class MovieConsumer {
     public void consume( Movie movie ) {
          ...
     }
}
```

Diese kann nun per Spring zu einem sogenannten „Message Driven POJO“ gemacht werden, also einem POJO, dass auch JMS Messages verarbeiten kann, indem man einen MessageListerContainer dafür konfiguriert. Dies ist ein dynamischer Adapter, der die JMS Nachricht aus der Queue holt, bestätigt, zurückkonvertiert und dann per Reflection unsere Methode aus dem POJO aufruft.

```
<jms:listener-container>
     <jms:listener destination="movieQueue"
                   ref="movieConsumer" method="consume"/>
</jms:listener-container>
```

3. Transparentes Caching mit AOP

Abschließend sollen kurz die Möglichkeiten von Spring AOP gezeigt werden. Man stelle sich vor, dass sich die Filme, die der MovieFinder zurückliefert, nie oder nur in sehr großen Zeitabständen ändern. Das macht sie zu idealen Caching-Kandidaten, man könnte sie einmalig lesen und dann im Speicher behalten.

Ein solches Verhalten stellt jedoch wieder einen querschnittlichen Belang dar, den man nicht an jeder Stelle im Code expliziert implementieren müssen, sondern nachträglich konfigurierbar ein- und ausschalten können möchte. Im Beispiel soll eine Annotation für Methoden gebaut werden, deren Ergebnis dann automatisch gecacht wird.

Mit Spring AOP lässt sich dies mit wenig Aufwand realisieren:

Wir definieren uns die Annotation `public @interface Cachable {}` und annotieren die Lesemethode des MovieFinders `@Cachable public Movie[] findMovies() { ... }`

Anschließend schalten wir Spring AOP mit Konfiguration über AspectJ Annotationen ein und definieren eine Spring Bean namens CachingAspect, in der das Cachingverhalten gekapselt werden soll.

```
<aop:aspectj-autoproxy/>
<bean id="cachingAspect" class="...CachingAspect"/>
```

Mithilfe der @Around Annotation können wir festlegen, dass dieser Aspekt um jeden Methodenaufruf einer mit @Cachable annotierten Methode aufgerufen wird. Der Code selbst identifiziert den Methodenaufruf, schaut im Speicher (in einer einfachen Implementierung könnte man eine HashMap nehmen) nach, ob das Ergebnis dieser Methode bereits vorliegt und liefert es im Erfolgsfall zurück ohne die Methode auszuführen. Sollte dies nicht der Fall sein, wird die Methode einmalig ausgeführt und das Ergebnis gespeichert.

```
@Aspect
public class CachingAspect {
     ...
     @Around( "@annotation( de.fuberlin.spring.Cachable )" )
     public Object cache( ProceedingJoinPoint joinPoint ) throws Throwable {

          String key = ...

          if ( !cache.containsKey( key ) ) {
               cache.put( key, joinPoint.proceed() );
          }
          return cache.get( key );
     }
}
```

IV. Quellen

Spring Dokumentation
http://static.springframework.org/spring/docs/2.5.x/reference/index.html

Artikel von Rod Johnson zum Release von Spring 2.5
http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25

Martin Fowler:
Inversion of Control Containers and the Dependency Injection pattern
http://martinfowler.com/articles/injection.html

Diskussion über Dependency Injection
http://www.theserverside.com/news/thread.tss?thread_id=23358

Spring Framework bei Wikipedia
http://de.wikipedia.org/wiki/Spring_(Framework)

AOP bei Wikipedia
http://de.wikipedia.org/wiki/AOP