

**Programmieren mit Spring**  
**Begleitbuch zum Seminar bei DSV**  
**vom 11.-13.12.2017**  
**Rainer Sawitzki**



<b>1</b>	<b>Rekapitulation der JPA Grundlagen.....</b>	<b>1-7</b>
1.1	O/R-Mapping .....	1-7
1.1.1	Die Kluft zwischen OO und relationalen Datenbanken .....	1-7
1.1.2	Aufgaben eines Objekt-relationalen Mappers .....	1-8
1.1.3	EJB 3 .....	1-16
1.2	Übersicht .....	1-17
1.3	Entities.....	1-20
1.3.1	FlatBook.....	1-20
1.3.2	ExtendedFlatBook .....	1-23
1.3.3	GeneratedKeyFlatBook.....	1-25
1.3.4	CompositeKeyFlatBook .....	1-26
1.3.5	EmbeddedKeyFlatBook .....	1-31
1.3.6	Vererbung.....	1-35
1.3.7	Mehrere Tabellen pro Entity.....	1-42
1.3.8	Relationen.....	1-43
1.4	Die EJB Query Language .....	1-57
1.4.1	Datenbestand .....	1-57
1.4.2	Programm .....	1-61
1.4.3	Suche alle Verleger.....	1-61
1.4.4	Suche einen bestimmten Verleger .....	1-62
1.4.5	Parametrisierte Suche nach einem Verleger .....	1-62
1.4.6	Bedingungen unter Verwendung der Relationen .....	1-63
1.4.7	Verknüpfte Bedingungen .....	1-63
1.4.8	Joins .....	1-64
1.4.9	Subqueries.....	1-64
<b>2</b>	<b>Spring – Eine Übersicht.....</b>	<b>2-66</b>
2.1	Was ist das Spring-Framework?.....	2-66
2.2	Der POJO-Container .....	2-70
2.3	Value Properties .....	2-75
2.3.1	Die Bean-Definition.....	2-75
2.3.2	Default-Typen von Properties und deren Konfiguration ...	2-75
2.3.3	Nullwerte von Properties.....	2-77
2.3.4	<code>init</code> - und <code>destroy</code> -Methoden.....	2-78
2.3.5	Setzen von Properties im Konstruktor.....	2-79
2.3.6	Registrierung spezieller Properties-Editoren.....	2-80
2.3.7	Verwendung von externen Properties-Dateien .....	2-82
2.4	Referenzen auf andere Beans.....	2-84
2.4.1	Identifikation über die ID .....	2-84
2.4.2	Definition einer Inner Bean .....	2-85

---

2.5	Abstrakte Beans und Vererbung an Kinder .....	2-86
2.6	Zugriff auf Ressourcen .....	2-87
2.6.1	Die <code>Resource</code> -Schnittstelle .....	2-87
2.6.2	Internationalisierung mit der <code>MessageSource</code> -Schnittstelle .....	2-89
2.7	Factory Beans .....	2-90
2.8	Konfiguration mit Annotations .....	2-97
<b>3</b>	<b>Datenzugriffe mit Spring .....</b>	<b>3-103</b>
3.1	Inversion of Control mit Method Template .....	3-103
3.2	Direkter JDBC-Zugriff .....	3-103
3.3	Transaktions-Management .....	3-107
3.3.1	Übersicht .....	3-107
3.3.2	<code>TransactionCallback</code> .....	3-108
3.3.3	<code>TransactionProxyFactoryBean</code> .....	3-109
3.3.4	tx-Namensraum .....	3-110
3.3.5	Annotations-basierte Transaktionssteuerung .....	3-110
3.4	Hibernate-Unterstützung .....	3-114
<b>4</b>	<b>Das Spring Web Framework .....</b>	<b>4-115</b>
4.1	Das Dispatcher-Servlet .....	4-116
4.2	Spezielle Beans für Web Anwendungen .....	4-118
4.3	Controller .....	4-121
4.3.1	Ein einfacher Controller .....	4-121
4.3.2	Weitere Controller-Implementierungen .....	4-124
4.4	Handler .....	4-130
4.5	Views .....	4-132
4.6	Web-Anwendungen und Form Controller .....	4-133
4.6.1	Form Controller .....	4-133
4.6.2	Formulare .....	4-133
<b>5</b>	<b>Spring Web Flow .....</b>	<b>5-139</b>
5.1	Graphen-orientierte Programmierung .....	5-139
5.1.1	Übersicht .....	5-139
5.1.2	Eigenschaften der Graphen-orientierte Programmierung ....	5-140
5.1.3	Prinzipielle Realisierung einer Graphen-orientierten Sprache .....	5-142
5.2	Business Prozesse und Web Flows .....	5-144
5.3	Übersicht .....	5-146
5.4	Konfiguration .....	5-146

5.4.1	Die Flow Registry.....	5-147
5.4.2	Der Flow-Executor .....	5-149
5.5	Elementare Flow-Definition.....	5-151
5.5.1	Das Flow-Schema.....	5-151
5.5.2	Der View State .....	5-151
5.5.3	Transitionen .....	5-152
5.5.4	End State .....	5-152
5.5.5	Flow-Parameter und Ergebnisse .....	5-153
5.5.6	Variablen.....	5-153
5.5.7	Sub Flows .....	5-154
5.5.8	Vererbung .....	5-154
5.6	Die Unified Expression Language .....	5-155
5.6.1	Objektzugriff.....	5-155
5.6.2	Typen von Expressions.....	5-155
5.6.3	Spezielle Variablen .....	5-156
5.6.4	Such-Algorithmus für Objekte .....	5-157
5.7	Actions.....	5-157
5.7.1	Events.....	5-157
5.7.2	Evaluate.....	5-159
5.7.3	action-state und decision-state .....	5-159
5.8	Views.....	5-160
5.8.1	view-state.....	5-160
5.8.2	Globale Transaktionen.....	5-160
5.8.3	View Template .....	5-160
5.8.4	Partielles Rendern .....	5-161
5.8.5	Binding.....	5-161
5.8.6	View History.....	5-164
5.9	Testen von Flows .....	5-165
5.10	Erste Beispiele.....	5-167
5.10.1	Simple Flow .....	5-167
5.10.2	Flow Variablen .....	5-172
5.11	Ausnahme-Verarbeitung.....	5-174
5.12	Dienste .....	5-175
5.12.1	Flow Managed Persistence.....	5-175
5.12.2	Security.....	5-176
5.13	Integration mit JavaServer Faces .....	5-180
5.13.1	Flow-Konfiguration .....	5-180
5.13.2	Änderungen in der web.xml .....	5-181
5.13.3	Spring-Beans-Konfiguration.....	5-184
5.13.4	Ein Beispiel-Flow .....	5-185

5.13.5	Die Seiten .....	5-189
5.14	Schema-Dateien .....	5-192
5.14.1	Flow-Definition .....	5-192
5.14.2	Die Flow-Konfiguration.....	5-193

# 1 Rekapitulation der JPA Grundlagen

## 1.1 O/R-Mapping

### 1.1.1 Die Kluft zwischen OO und relationalen Datenbanken

Die objektorientierte und die relationale Welt passen aus vielerlei Gründen nicht so recht zusammen:

- Die Objekte der Objektorientierung leben im Hauptspeicher und sind somit "flüchtig". Die "Objekte" von Datenbanken sind persistent.
- Die Objektorientierung kennt "intelligente" Objekte – Objekte, die ihren Zustand kapseln. Die Klassen dieser Objekte enthalten Methoden, mittels derer der Zustand der Objekte abfragbar und manipulierbar ist. Relationale Datenbanken dagegen enthalten nur "dumme" Daten.
- Relationale Datenbanken kennen andere Typen als objektorientierte Sprachen. Die Datenbank kennt z.B. die Typen `CHAR` und `VARCHAR`; Java dagegen kennt den Typ `String`.
- Objektorientierte Sprachen sind imperative Sprachen; die typische Datenbanksprache SQL aber ist deklarativ.
- In der objektorientierten Welt sind Objekte miteinander über Referenzen (also Pointer) verbunden. In relationalen Datenbanken werden die "Objekte" über Fremdschlüssel-Beziehungen miteinander verbunden. Die Objektorientierung kennt aber keine Fremdschlüssel (und auch keine Primärschlüssel).
- Die Objektorientierung fokussiert individuelle Objekte; zwischen diesen Objekten kann navigiert werden. (Natürlich lassen sich solche individuellen Objekte auch in Collections zusammenfassen.) Die typische Zugriffsweise von Datenbanken ist dagegen der `SELECT` in Verbindung mit dem `JOIN` – eine Zugriffsweise, die grundsätzlich Mengen von Zeilen liefert. Im Gegensatz zur Objektorientierung operiert die Datenbank also mengenorientiert.
- Die Objektorientierung kennt das Vererbungskonzept. Relationale Datenbanken dagegen kennen mit wenigen Ausnahmen keine Vererbung.
- Relationale Datenbanken beruhen wesentlich auf dem Konzept der referenziellen Integrität; in der Objektorientierung ist dieses Konzept von Natur aus unbekannt.

### 1.1.2 Aufgaben eines Objekt-relationalen Mappers

Aus diesen Unterschieden zwischen der objektorientierten und der Datenbank-Welt leiten sich die Hauptaufgaben eines objekt-relationalen Mappers ab.

Im Folgenden wird vorausgesetzt, dass ein O/R-Mapper JDBC benutzt, um auf die Datenbanken zuzugreifen.

#### 1.1.2.1 Abbildung von Tabellenzeilen auf Objekte und umgekehrt

Ein O/R-Mapper muss eine Zeile einer relationalen Tabelle auf ein Objekt abbilden können – und zwar in beide Richtungen: er muss die Spaltenwerte einer Tabellenzeile lesen und dieses dann den Attributen des Objekts zuweisen können; und er muss umgekehrt die Attributwerte eines Objekts auslesen können und diese den Spalten einer Tabelle zuweisen können.

Diese Abbildung muss "generisch" erfolgen. Auf der JDBC-Seite werden dabei die Methoden `ResultSet.getObject` und `PreparedStatement.setObject` genutzt. Mittels der `ResultSet`-Methode `getObject` kann ein beliebiger Spaltenwert aus einer Ergebnistabelle gelesen werden; `getObject` erzeugt ein zu dem Spaltentyp passendes Objekt (z.B. einen `String`, ein `Integer`- oder ein `Double`-Objekt etc.) und liefert dieses Objekt in der allgemeinen Form eines `Objects` zurück. Umgekehrt verlangt die `setObject`-Methode der Klasse `PreparedStatement` eine allgemeine `Object`-Referenz als Parameter.

```
public interface ResultSet {  
    ...  
    public Object getObject (int columnIndex)  
    public Object getObject (String columnName)  
}
```

```
public interface PreparedStatement {  
    ...  
    public void setObject (int columnIndex, Object value)  
    public void setObject (String columnName, Object value)  
}
```

Auf der Seite der Java-Objekte werden meistens Beans (bzw. POJOs: Plain Old Java Objects) vorausgesetzt. Eine Bean ist ein Objekt einer Klasse, die erstens das Interface `Serializable` implementiert, die zweitens einen parameterlosen Konstruktor besitzt und die drittens über setter- und getter-Methoden verfügt, welche den Zugriff auf die Attribute eines Objekts dieser Klasse gestatten.



Aufgrund der Existenz eines parameterlosen Konstruktors können Objekte dann mittels `Class.newInstance` "generisch" erzeugt werden:

```
public class Class {  
    ...  
    public Object newInstance ()  
}
```

Ein `Book`-Objekt könnte etwa wie folgt erzeugt werden:

```
String clsName = "Book";  
Class cls = Class.forName (clsName);  
Object obj = cls.newInstance ();
```

(Um ein Objekt einer Klasse zu erzeugen, muss also nur der Name der Klasse in Form eines `Strings` (!) bekannt sein.)

Und aufgrund der Existenz von getter- und setter-Methoden können dann die Attribute eines Objekts per Reflection gelesen bzw. gesetzt werden. Hier ein kleines Beispiel (wobei von der Fehlerbehandlung abgesehen wird):

```
Class cls = Class.forName ("Book");  
Object obj = cls.newInstance ();  
....  
Method m = cls.getMethod ("getTitle", new Class [] { });  
Object value = m.invoke (obj, new Object [] { });
```

Hier wird die Methode `setTitle` auf das zuvor erzeugte `Book`-Objekt aufgerufen. (Man beachte auch hier, dass nur der Name der Methode in Form eines `Strings` bekannt sein muss, um die Methode dann per `Method.invoke` aufrufen lassen zu können.)

Und so würde der Titel des Buches neu gesetzt werden können:

```
Method m = cls.getMethod ("setTitle", new Class [] { String.class });  
m.invoke (obj, new Object [] { "Design Patterns" });
```

Seien in der Klasse `Book` also z.B. folgende Methoden gegeben:

```
public class Book ... {  
    ...  
    void setIsbn (String isbn)  
    String getIsbn ()  
  
    void setTitle (String title)  
    String getTitle ()  
  
    void setPrice (double price)  
    double getPrice ()  
}
```

Dann bezeichnet man die jeweilige setter/getter-Kombination auch als Property. Die Klasse `Book` enthält also die Properties `"isbn"`, `"title"` und `"price"`. Der Begriff Property darf dabei nicht verwechselt werden mit dem Begriff Attribut bzw. Instanzvariable. Die Attribute, auf denen die obigen drei Methoden operieren, könnten `theke`, `antitheke` und `syntheke` heißen!

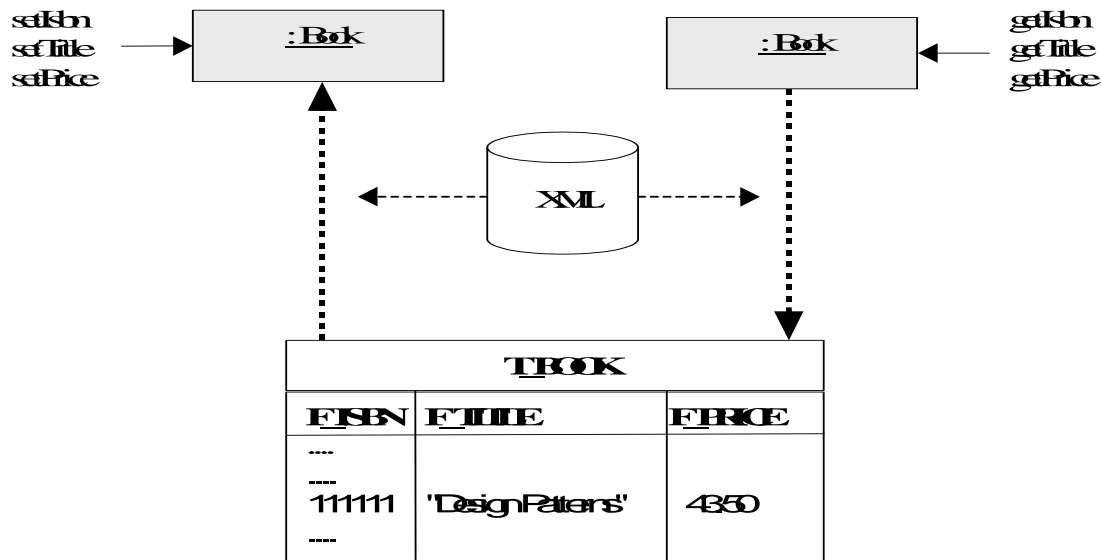
Mittels Reflection können also die Properties von Objekten gelesen und gesetzt werden.

Und schließlich muss bei der Abbildung von Tabellenzeilen auf Objekte geklärt werden, welche Tabellenspalte auf welche Property abgebildet werden soll. Im einfachsten Falle könnten die Namen der Tabellenspalten identisch sein mit den Namen der Properties der Objektklasse. Ansonsten muss diese Abbildung in irgendeiner Form beschrieben sein – z.B. in einer XML-Datei. Diese könnte etwa wie folgt aussehen:

```
<mapping class="Book" table="T_BOOK">  
    <property name="isbn" column="F_ISBN" />  
    <property name="name" column="F_NAME" />  
    <property name="price" column="F_PRICE" />  
    ...  
</mapping>
```

Eine Alternative ist es, das Mapping über Annotations durchzuführen.

Das folgende Schaubild soll das erforderliche Mapping verdeutlichen:



#### 1.1.2.2 Abbildung von Primär-/Fremdschlüssel-Beziehungen auf Referenzen

Die in der Datenbank enthaltenen Primär-/Fremdschlüsselbeziehungen müssen auf referenzielle Beziehungen der Objekte abgebildet werden.

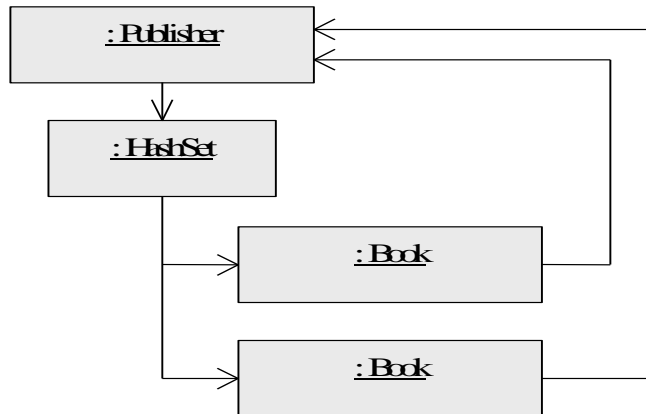
Sei z.B. neben der Tabelle T\_BOOK noch die Tabelle T\_PUBLISHER gegeben. Die T\_PUBLISHER-Tabelle habe den Primärschlüssel F\_ID. Dann würde die T\_BOOK-Tabelle eine Fremdschlüssel-Spalte F\_PUBLISHER\_ID enthalten.

Hier einige beispielhafte Einträge dieser Tabellen:

TPUBLISHER	
FID	FNAME
....	
5	"Addison Wesley"
....	

TBOOK			
ISBN	TITLE	PRICE	F_PUBLISHER_ID
....			
11111	'Design Patterns'	450	5
22222	'Oreil'	400	5
....			

Wird nun auf den Publisher mit der F\_ID 5 zugegriffen, so sollten drei Objekte erzeugt werden: ein Publisher- und zwei Book-Objekte. Diese müssten dann etwa wie folgt miteinander verknüpft sein:



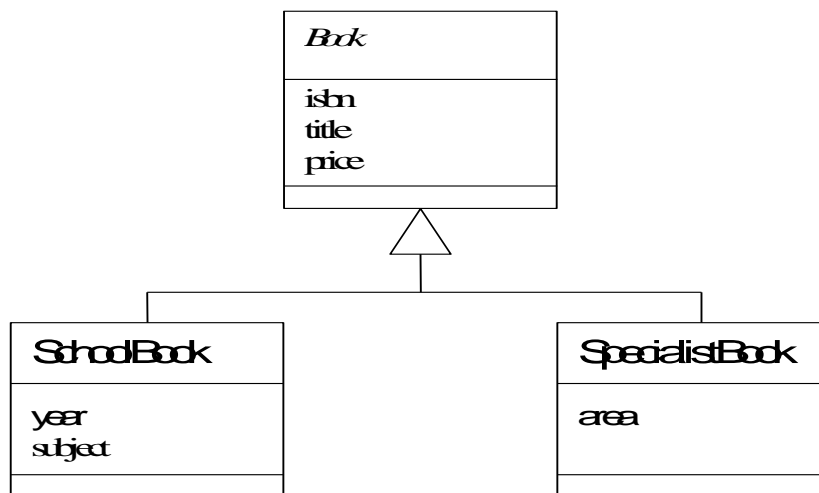
Ein `Publisher` wird ein Collection-Objekt (z.B. ein `HashSet`) besitzen, in welchem die Referenzen auf die `Books` dieses Publishers gespeichert werden; und jedes `Book`-Objekt wird eine Referenz auf den `Publisher` besitzen, der dieses `Book` verlegt.

Der OR-Mapper sollte diese Objekte automatisch miteinander verbinden (aus Sicht der Java-Anwendung sollte die Transformation zwischen Primär-/Fremdschlüssel-Beziehungen und den referenziellen Beziehungen der Objekte also vollständig transparent sein.)

### 1.1.2.3 Abbildung von Vererbungsbeziehungen

Ein OR-Mapper muss Vererbungsbeziehungen transparent auf Tabellen abbilden können.

Sei z.B. folgende Klassenhierarchie gegeben:



Dann könnten Schulbücher und Fachbücher z.B. auf drei Tabellen abgebildet werden: eine Tabelle enthält die Basisdaten aller Bücher (einschließlich des Primary keys `F_ISBN`); für jede abgeleitete Klasse existiert eine weitere Tabelle, in welcher jeweils die isbn-Nummern und die Spezialdaten der Bücher gespeichert werden.

Seien z.B. zwei Fachbücher und ein Schulbuch gegeben. Diese könnten in den Tabellen wie folgt gespeichert sein:

<b>T_BOOK</b>			
<b>F_ISBN</b>	<b>F_TITLE</b>	<b>F_AUTHOR</b>	<b>F_PRICE</b>
11111	'Spick'	'Langlaas'	450
22222	'Spick'	'Gad'	450
33333	'Spick'	'Langlaas'	500

<b>T_SPECIALIST_BOOK</b>	
<b>F_ISBN</b>	<b>F_AREA</b>
11111	"OO-Design"
22222	"OO-Programming"

<b>T_SCHOOL_BOOK</b>		
<b>F_ISBN</b>	<b>F_YEAR</b>	<b>F_SUBJECT</b>
333333	10	"English"

Um auf Grundlage dieser Tabellenstruktur ein `SpecialistBook` (oder ein `SchoolBook`) zu erzeugen, müssen dann die Daten aus zwei Tabellen eingelesen werden: aus der Basistabelle `T_BOOK` und der Tabelle `T_SPECIALISTBOOK` (bzw. `T_SCHOOLBOOK`).

Die oben gezeigte Abbildung ist nur eine von mehreren Möglichkeiten. Gleichgültig aber, welche konkrete Abbildung verwendet wird, für die Java-Anwendung sollte die Abbildung transparent sein. Die konkrete Form der Abbildung sollte auch geändert werden können, ohne dass die Java-Anwendung von einer solchen Änderung tangiert wird.

Die konkrete Abbildung dann wieder z.B. in einer XML-Datei oder Annotations beschrieben werden.

#### 1.1.2.4 Generierung von SQL-Statements

Wenn dem OR-Mapping die Abbildung zwischen dem Objektmodell und dem Datenbank-Schema über XML-Dateien bekannt gemacht wird, dann ist es natürlich auch möglich, die für das `INSERT`, das `UPDATE` und das `DELETE` erforderlichen SQL-Statements automatisch zur Laufzeit zu generieren.

Ist z.B. das bereits oben erwähnte Mapping vorgegeben:

```
<mapping class="Book" table="T_BOOK">
  <property name="isbn" column="F_ISBN" />
  <property name="name" column="F_NAME" />
  <property name="price" column="F_PRICE" />
</mapping>
```

Dann kann aufgrund dieses Mapping z.B. folgender `INSERT` automatisch generiert werden:

```
INSERT INTO T_BOOK (F_ISBN, F_NAME, F_PRICE) VALUES (?, ?, ?)
```

Zum Zwecke der Generierung des `DELETE`-Statements müsste aus dem obigen Mapping allerdings auch noch hervorgehen, dass `F_ISBN` der Primary key ist. Dann könnte folgendes `DELETE`-Statement generiert werden:

```
DELETE FROM T_BOOK WHERE F_ISBN = ?
```

#### 1.1.2.5 Objektorientierte Abfragesprache

Für Abfragen sollte ein OR-Mapper eine eigene, objektorientierte Sprache anbieten.

In der Java-Anwendung sollten die Tabellennamen und die Spaltennamen nicht bekannt sein. Die Java-Anwendung kennt nur die Properties der entsprechenden Klasse. Also sollten auch Abfragen derart formuliert werden können, dass nur die Kenntnis der Namen der Java-Klassen und die Namen der Properties dieser Klassen bekannt sein müssen.

Statt also etwa folgenden `SQL-SELECT` formulieren zu müssen:

```
SELECT F_ISBN, F_TITLE, F_PRICE FROM T_BOOK WHERE F_ISBN = ?
```

sollte einfach formuliert werden können:

```
from Book where isbn = ?
```

Wobei `Book` der Name der Klasse und `isbn` eine der `Book`-Properties ist. Man beachte, dass die Aufzählung der Spalten unnötig ist – denn es soll ja ein `Book`-Objekt erzeugt werden und komplett mit den Daten der entsprechenden Tabellenzeile initialisiert werden. Also muss die Projektion naturgemäß alle Spalten umfassen.

Für die Formulierung von Joins darf dann natürlich auch nicht mehr der Name der Fremdschlüssel-Spalte verwendet werden. Statt also zu formulieren:

```
SELECT B.ISBN, B.PRICE P.ID P.NAME  
FROM BOOK B, PUBLISHER P  
WHERE B.PUBLISHER_ID = P.PUBLISHER_ID
```

sollte etwa die folgende Zeile notiert werden können:

```
"from Book b, Publisher p where b.publisher = p";
```

(Wobei vorausgesetzt wird, dass die Klasse `Book` die Methoden `getPublisher` und `setPublisher` besitzt – also die Property `"publisher"`.)

Die objektorientierte Abfragesprache muss dann natürlich vom OR-Mapper in ein geeignetes SQL-Statement transformiert werden. Hierbei sollte beachtet werden, dass es natürlich "das" SQL überhaupt nicht gibt – vielmehr existieren unterschiedliche SQL-Dialekte. Und diese Dialekte sollten bei der Transformation berücksichtigt werden – um möglichst performante SQL-Statements zu generieren.

#### 1.1.2.6 Identität von Objekten

Wird mittels eines OR-Mappers eine Tabellenzeile gelesen und in ein Objekt transformiert, so sollte dieses Objekt das einzige Objekt sein, welches die entsprechende Zeile im Hauptspeicher repräsentiert. Eine nochmalige Aufforderung, die entsprechende Zeile zu lesen, sollte also dasselbe Objekt zurückliefern wie die erste Aufforderung. Hierzu muss ein OR-Mapper die von ihm erzeugten und bereitgestellten Objekte geeignet cachen. Die Frage lautet dann natürlich, wie lange ein solcher Cache "gültig" bleibt.

#### 1.1.2.7 Natives SQL

Für bestimmte Zwecke mag es sinnvoll oder gar notwendig sein, Abfragen oder DML-Befehle direkt in SQL zu formulieren. Der OR-Mapper sollte solche "workarounds" zulassen. Der OR-Mapper sollte sich also nicht einbilden, **alles** besser zu können... Insbesondere sollte natürlich der Aufruf von Stored Procedures möglich sein.

#### 1.1.2.8 J SE und J EE

Ein O/R-Mapper sollte sowohl in einer einfachen J SE-Anwendung als auch in einer J EE-Anwendung genutzt werden können. Diese Anforderung steht z.B. in direktem Widerspruch zu den sog. CMP-Entity-Beans (Container Managed Persistence) von EJB. Solche "Beans" lassen sich nur einem EJB-Container nutzen – sind also in einer einfachen J SE-Anwendung völlig unbrauchbar. M.a.W.: ein O/R-Mapper sollte (in seinem Kern) keinerlei Annahmen über die Umgebung machen, in welcher er verwendet wird.

#### 1.1.2.9 Keine "Verdopplung" von Datenbank-Constraints

Die Datenbank garantiert die Einhaltung bestimmter Constraints – insbesondere des `FOREIGN KEY`-Constraints (also referenzielle Integrität). Solche Constraints, die bereits die Datenbank selbst garantiert, brauchen natürlich vom OR-Mapper nicht noch zusätzlich garantiert werden. Der OR-Mapper muss also die Datenbank nicht "neu erfinden".

#### 1.1.3 EJB 3

Mit dem neuen Java Persistence API (JPA) wurde für die EJB 3 ein komplette Überarbeitung der EntityBean-Spezifikation übermittelt. Eigentlich gibt es seit EJB 3 keinen eigenen Bean-Typen mehr, der speziell einem persistenten Datenzustand entspricht. Stattdessen wurde in der Java Enterprise Edition ein Standard für O/R-Mapping definiert, der sich in vielerlei Hinsicht an populären Werkzeugen wie Hibernate, iBatis oder TopLink orientiert. Diese Ähnlichkeit führte dazu, dass beispielsweise der JBoss-Applikationsserver die geforderte Implementierung des JPA durch ein geringfügig modifiziertes Hibernate umgesetzt hat.

JPA setzt fast vollständig auf den Annotations-basierten Ansatz.



## 1.2 Übersicht

Die Java Enterprise Edition 5 hat mit der Integration des Java Persistence API das Komponentenmodell der EntityBeans komplett überarbeitet. Das Programmiermodell weist nur noch marginale Ähnlichkeiten zum bisherigen auf. Stattdessen wird eine Kombination aus Annotations in Verbindung mit POJOs und einem Entity-Manager eingeführt. Diese Art des O/R-Mappings ist aus anderen Frameworks wie Oracle TopLink und Hibernate bestens bekannt.

Der `javax.persistence.EntityManager` stellt die folgenden Funktionalitäten zur Verfügung:

- `persist(Object p)` speichert eine Entität
- Suchen von Entitäten mit `find(Class entityClass, Object primaryKey)`
- Erzeugen von Query-Objekten
- Cache-API und Synchronisation mit der Datenbank (`refresh`, `merge`, `flush`)
- Zugriff auf die aktuell laufende Transaktion

Ein Applikationsserver muss eine Implementierung des EntityManagers zur Verfügung stellen. Dies folgt über die Dependency Injection eines `PersistenceContext`. Solch ein Kontext definiert im Wesentlichen, welche Implementierung verwendet werden soll und wie diese konfiguriert ist.

Eine `Query` ist ein komfortables API um Datenbank-Abfragen zu definieren.

Der weitaus größte Teil des neuen APIs besteht jedoch aus Annotations, die dem POJO, das als Entity aufgefasst werden soll, hinzugefügt werden:

- `AttributeOverride`
- `AttributeOverrides`
- `Basic`
- `Column`
- `ColumnResult`
- `DiscriminatorColumn`
- `Embeddable`
- `EmbeddableSuperclass`
- `Embedded`
- `EmbeddedId`

- Entity
- EntityListener
- EntityResult
- Enumerated
- FieldResult
- FlushMode
- GeneratedValue
- Id
- IdClass
- Inheritance
- JoinColumn
- JoinColumns
- JoinTable
- Lob
- ManyToMany
- ManyToOne
- MapKey
- NamedNativeQueries
- NamedNativeQuery
- NamedQueries
- NamedQuery
- OneToMany
- OneToOne
- OrderBy
- PersistenceContext
- PersistenceContexts
- PersistenceUnit
- PersistenceUnits
- PostLoad
- PostPersist
- PostRemove

- PostUpdate
- PrePersist
- PreRemove
- PreUpdate
- PrimaryKeyJoinColumn
- PrimaryKeyJoinColumns
- QueryHint
- SecondaryTable
- SecondaryTables
- SequenceGenerator
- SqlResultSetMapping
- Table
- TableGenerator
- Temporal
- Transient
- UniqueConstraint
- Version

Die wichtigsten Annotations werden im folgenden Unterkapitel beschrieben.

## 1.3 Entities

Nachdem für die Beispiele der `PersistenceContext` von einem Applikationsserver zur Verfügung gestellt wird, werden für die folgenden Beispiele jeweils eine `SessionBean` benutzt, in denen die jeweiligen Entities angesprochen werden. Die angegebenen JUnit-Tests rufen jeweils korrespondierende Methoden innerhalb der `SessionBean` auf, in denen die eigentliche Test-Logik ausgeführt wird.

### 1.3.1 FlatBook

Zur Definition einer simplen Entity werden nur zwei Annotations benötigt:

- `@Entity` deklariert die Klasse als Entity-Klasse
- `@Id` definiert die Spalte, die als Primärschlüssel benutzt werden wird

```
@Entity
public class FlatBook implements Serializable {
    private static final long serialVersionUID = 1L;

    private String isbn;

    private String title;

    private int pages;

    private double price;

    private String description;

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @Id
    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public int getPages() {
        return pages;
    }

    public void setPages(int pages) {
        this.pages = pages;
    }
}
```

```
}

public double getPrice() {
    return price;
}

public void setPrice(double price) {
    this.price = price;
}

public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}
}
```

Der Quellcode der SessionBean lautet:

```
@Stateless
@Remote
public class PersistenceTestBean implements
RemotePersistenceTest {

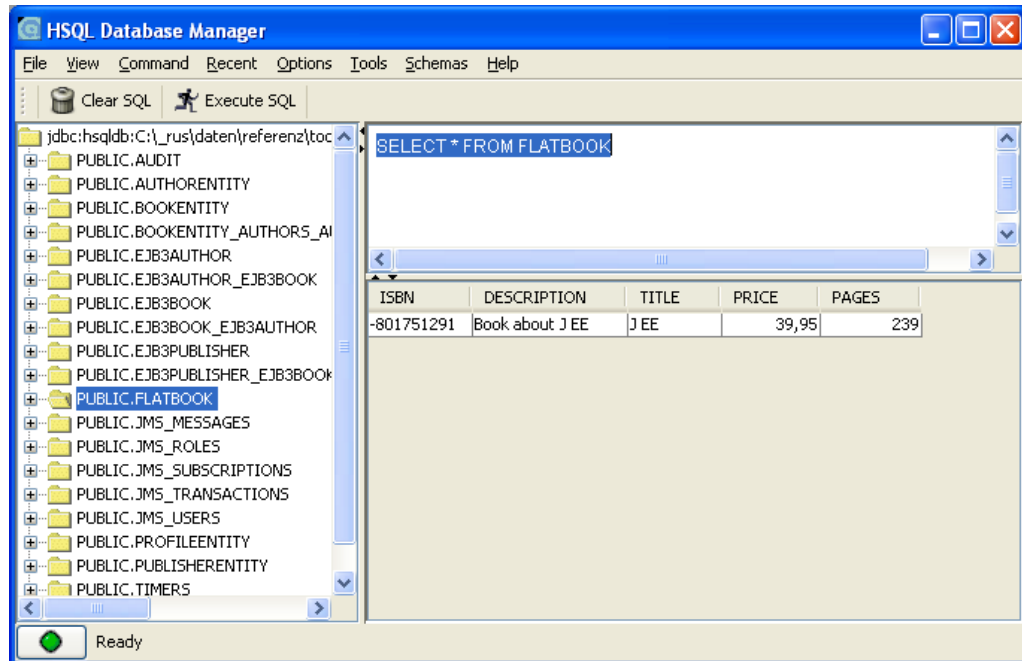
    @PersistenceContext
    private EntityManager entityManager;

    private Random random;

    {
        random = new Random();
    }

    public void testFlatEntity() {
        FlatBook flatBook = new FlatBook();
        flatBook.setIsbn(Integer.toString(random.nextInt()));
        flatBook.setTitle("J EE");
        flatBook.setDescription("Book about J EE");
        flatBook.setPages(239);
        flatBook.setPrice(39.95);
        entityManager.persist(flatBook);
    }
}
```

Nach Ausführen dieser Methode wird der Datensatz in die im `PersistenceContext` definierten Datenbank abgelegt. Beim JBoss ist dies die Hypersonic-Datenbank, die mit einer einfachen Administrationsoberfläche ausgeliefert wird:



Die erzeugte Tabelle hat folgende Definition:

```
create table FlatBook (isbn varchar(255) not null, description
varchar(255), title varchar(255), price double not null, pages
integer not null, primary key (isbn))
```

Die Struktur der Tabelle wurde offensichtlich aus dem Namen der Entity-Klasse gebildet. Alle Attribute werden als persistent angenommen und die zugehörigen SQL-Typen aus dem Java-Typ abgeleitet.

### 1.3.2 ExtendedFlatBook

Mit den Annotations `@Table`, `@Basic`, `@Transient` und `@Column` werden zusätzlich Metadaten eingeführt:

```
@Entity
@Table(name = "EXTENDED_FLAT_BOOK", uniqueConstraints =
@UniqueConstraint(columnNames = { "COL_TITLE" }))
public class ExtendedFlatBook implements Serializable {
    private static final long serialVersionUID = 1L;

    private String isbn;

    private String title;

    private int pages;

    private double price;

    private String description;

    private boolean available;

    @Transient
    public boolean isAvailable() {
        return available;
    }

    public void setAvailable(boolean available) {
        this.available = available;
    }

    @Basic(optional=false)
    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @Id
    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public int getPages() {
        return pages;
    }

    public void setPages(int pages) {
        this.pages = pages;
    }
}
```

```

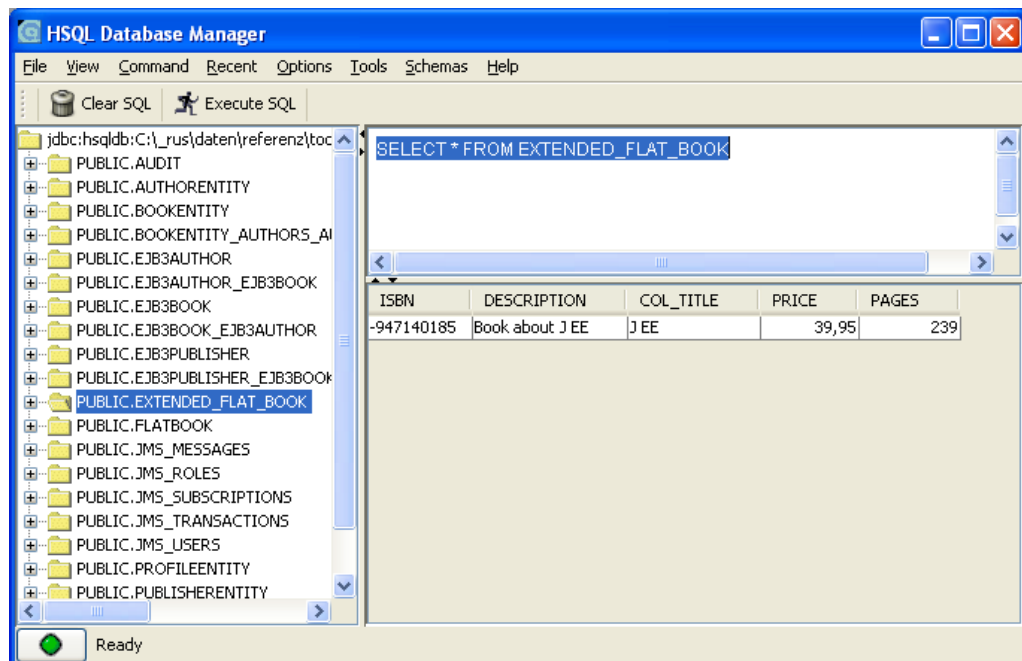
    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    @Column(length=100, name="COL_TITLE")
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}

```



Die Tabellendefinition lautet nun:

```

create table EXTENDED_FLAT_BOOK (isbn varchar(255) not null,
description varchar(255) not null, COL_TITLE varchar(100),
price double not null, pages integer not null, primary key
(isbn), unique (COL_TITLE))

```

- Der Name der Tabelle sowie der Unique-Constraint wird aus @Table übernommen
- Name Länge der description-Spalte werden aus @Column gelesen
- Das @Transient-Attribut available ist nicht innerhalb der Tabelle abgelegt



### 1.3.3 GeneratedKeyFlatBook

Der Primärschlüssel des Buches kann auch automatisch erzeugt werden. Dazu dient die Annotation `@GeneratedValue`:

```
@Entity
public class GeneratedKeyFlatBook implements Serializable {
    private static final long serialVersionUID = 1L;

    private Long isbn;

    private String title;

    private int pages;

    private double price;

    private String description;

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE)
    public Long getIsbn() {
        return isbn;
    }

    public void setIsbn(Long isbn) {
        this.isbn = isbn;
    }

    public int getPages() {
        return pages;
    }

    public void setPages(int pages) {
        this.pages = pages;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

Der Typ der isbn wird für die gewählte Strategie der Schlüsselerzeugung (fortlaufende Sequenz) auf Long geändert.

Die Tabellendefinition lautet nun:

```
create table GeneratedKeyFlatBook (isbn bigint not null,
description varchar(255), title varchar(255), price double not
null, pages integer not null, primary key (isbn))
```

Der Primärschlüssel wird dabei aus einer separat angelegten Sequenz-Tabelle entnommen.

Wird die Strategy auf GenerationType.AUTO geändert, so lautet die Tabellendefinition:

```
create table GeneratedKeyFlatBook (isbn bigint generated by
default as identity (start with 1), description varchar(255),
title varchar(255), price double not null, pages integer not
null, primary key (isbn))
```

### 1.3.4 CompositeKeyFlatBook

Wird ein zusammengesetzter Schlüssel benötigt, so wird dafür eine eigene Klasse geschrieben, die alle Attribute des Composite Keys besitzt sowie die Semantik eines Value Objects (equals, hashCode...) besitzt:

```
final public class Isbn implements java.io.Serializable {

    private static final long serialVersionUID = 1L;

    private String languageCode;

    private String publisherNumber;

    private String bookNumber;

    private String checkNumber;

    public Isbn() {
        super();
    }

    public Isbn(String language, String publisherNumber, String
bookNumber,
        String check) {
        this.languageCode = language;
        this.publisherNumber = publisherNumber;
        this.bookNumber = bookNumber;
        this.checkNumber = check;
        if (this.getStringValue().length() != 13)
            throw new IllegalArgumentException();
    }
}
```

```
}

public Isbn(String isbn) {
    String[] tokens = isbn.split("-");
    this.languageCode = tokens[0];
    this.publisherNumber = tokens[1];
    this.bookNumber = tokens[2];
    this.checkNumber = tokens[3];
    if (this.getStringValue().length() != 13)
        throw new IllegalArgumentException();
}

public String getLanguageCode() {
    return this.languageCode;
}

public String getPublisherNumber() {
    return this.publisherNumber;
}

public String getBookNumber() {
    return this.bookNumber;
}

public String getCheckNumber() {
    return this.checkNumber;
}

@Transient
public String getStringValue() {
    return this.languageCode + "-" + this.publisherNumber + "-"
        + this.bookNumber + "-" + this.checkNumber;
}

@Override
public boolean equals(Object other) {
    if (!(other instanceof Isbn))
        return false;
    Isbn isbn = (Isbn) other;
    return this.languageCode.equals(isbn.languageCode)
        && this.publisherNumber.equals(isbn.publisherNumber)
        && this.bookNumber.equals(isbn.bookNumber)
        && this.checkNumber.equals(isbn.checkNumber);
}

@Override
public int hashCode() {
```

```
        return this.languageCode.hashCode() +
this.publisherNumber.hashCode()
            + this.bookNumber.hashCode() +
this.checkNumber.hashCode();
    }

    @Override
    public String toString() {
        return this.getClass().getName() + " [" +
this.getStringValue() + "];"
    }

    public void setBookNumber(String bookNumber) {
        this.bookNumber = bookNumber;
    }

    public void setCheckNumber(String check) {
        this.checkNumber = check;
    }

    public void setLanguageCode(String language) {
        this.languageCode = language;
    }

    public void setPublisherNumber(String publisherNumber) {
        this.publisherNumber = publisherNumber;
    }
}
```

Zu beachten hier ist die Verwendung der `@Transient`-Annotation: Diese ist hier notwendig da sonst in der Entity-Klasse nach einem Attribut "stringValue" als Bestandteil des Schlüssels gesucht werden würde.

Die zugehörige Entity definiert die Primary-Key-Klasse durch @IdClass:

```
@IdClass(Isbn.class)
@Entity
public class CompositeKeyFlatBook implements Serializable {
    private static final long serialVersionUID = 1L;

    private Isbn isbn;

    private String title;

    private String language;

    private String publisherNumber;

    private String bookNumber;

    private String check;

    @Transient
    public Isbn getIsbn() {
        if (isbn == null) {
            isbn = new Isbn(getLanguageCode(), getPublisherNumber(),
                getBookNumber(), getCheckNumber());
        }
        return isbn;
    }

    public void setIsbn(Isbn isbn) {
        this.isbn = isbn;
        setPublisherNumber(isbn.getPublisherNumber());
        setBookNumber(isbn.getBookNumber());
        setCheckNumber(isbn.getCheckNumber());
        setLanguageCode(isbn.getLanguageCode());
    }

    @Id
    public String getBookNumber() {
        return bookNumber;
    }

    public void setBookNumber(String bookNumber) {
        this.bookNumber = bookNumber;
    }

    @Id
```

```
public String getCheckNumber() {
    return check;
}

public void setCheckNumber(String check) {
    this.check = check;
}

@Id
public String getLanguageCode() {
    return language;
}

public void setLanguageCode(String language) {
    this.language = language;
}

@Id
public String getPublisherNumber() {
    return publisherNumber;
}

public void setPublisherNumber(String publisherNumber) {
    this.publisherNumber = publisherNumber;
}

public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}
}
```

Auch hier wird `@Transient` verwendet, um die nur aus der Objektorientierten Modellierung sinnvollen Methode `getIsbn()` nicht als persistente Property zu interpretieren und in der Datenbank abzulegen.

Die zugehörige Datenbanktabelle wird durch das folgende Statement definiert:

```
create table CompositeKeyFlatBook (publisherNumber varchar(255)
not null, bookNumber varchar(255) not null, languageCode
varchar(255) not null, checkNumber varchar(255) not null, title
varchar(255), primary key (publisherNumber, bookNumber,
languageCode, checkNumber))
```

### 1.3.5 EmbeddedKeyFlatBook

Der Primärschlüssel kann aber auch als Beispiel für einen "Embedded" Typ verwendet werden. Dabei werden Attribute einer Entity im Objektmodell zu einer eigenen Klasse zusammengefasst, die keine eigene Entity ist, deshalb so im Datenmodell nicht auftritt. Embedded Typen sind somit klar von Relationen zu trennen.

Dazu dienen die Annotations `@Embeddable`, `@Embedded` und `@EmbeddedId`.

`@Embeddable` sind in diesem Beispiel die `Isbn` und eine Buch-Beschreibung:

```
@Embeddable
final public class IsbnPk implements java.io.Serializable {

    private static final long serialVersionUID = 1L;

    private String languageCode;

    private String publisherNumber;

    private String bookNumber;

    private String checkNumber;

    public IsbnPk() {
        super();
    }

    public IsbnPk(String language, String publisherNumber, String
bookNumber,
        String check) {
        this.languageCode = language;
        this.publisherNumber = publisherNumber;
        this.bookNumber = bookNumber;
        this.checkNumber = check;
        if (this.getStringValue().length() != 13)
            throw new IllegalArgumentException();
    }

    public IsbnPk(String isbn) {
        String[] tokens = isbn.split("-");
        this.languageCode = tokens[0];
        this.publisherNumber = tokens[1];
        this.bookNumber = tokens[2];
        this.checkNumber = tokens[3];
        if (this.getStringValue().length() != 13)
            throw new IllegalArgumentException();
    }

    public String getLanguageCode() {
        return this.languageCode;
    }

    public String getPublisherNumber() {
        return this.publisherNumber;
    }
}
```

```
}

public String getBookNumber() {
    return this.bookNumber;
}

public String getCheckNumber() {
    return this.checkNumber;
}

@Transient
public String getStringValue() {
    return this.languageCode + "-" + this.publisherNumber + "-"
        + this.bookNumber + "-" + this.checkNumber;
}

@Override
public boolean equals(Object other) {
    if (!(other instanceof IsbnPk))
        return false;
    IsbnPk isbn = (IsbnPk) other;
    return this.languageCode.equals(isbn.languageCode)
        && this.publisherNumber.equals(isbn.publisherNumber)
        && this.bookNumber.equals(isbn.bookNumber)
        && this.checkNumber.equals(isbn.checkNumber);
}

@Override
public int hashCode() {
    return this.languageCode.hashCode() +
this.publisherNumber.hashCode()
        + this.bookNumber.hashCode() +
this.checkNumber.hashCode();
}

@Override
public String toString() {
    return this.getClass().getName() + " [" +
this.getStringValue() + "];"
}

public void setBookNumber(String bookNumber) {
    this.bookNumber = bookNumber;
}

public void setCheckNumber(String check) {
    this.checkNumber = check;
}

public void setLanguageCode(String language) {
    this.languageCode = language;
}

public void setPublisherNumber(String publisherNumber) {
    this.publisherNumber = publisherNumber;
}
}

@Embeddable
public class BookInfo implements Serializable {
```



```
private static final long serialVersionUID = 1L;

private String description;

private int pages;

public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = description;
}

public int getPages() {
    return pages;
}

public void setPages(int pages) {
    this.pages = pages;
}

@Override
public boolean equals(Object other) {
    if (!(other instanceof BookInfo))
        return false;
    BookInfo bookInfo = (BookInfo) other;
    return this.description.equals(bookInfo.description)
        && (this.pages == bookInfo.pages);
}

@Override
public int hashCode() {
    return this.description.hashCode() + pages;
}
}
```

Die Entity-Klasse verwendet diese Klassen als Primärschlüssel (`@EmbeddedId`) bzw. als normales Attribut(`@EmbeddedId`):

```
@Entity
public class EmbeddedKeyFlatBook implements Serializable {
    private static final long serialVersionUID = 1L;

    private IsbnPk isbn;

    private String title;

    private BookInfo bookInfo;

    @EmbeddedId
    public IsbnPk getIsbn() {
        return isbn;
    }

    public void setIsbn(IsbnPk isbn) {
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @Embedded
    public BookInfo getBookInfo() {
        return bookInfo;
    }

    public void setBookInfo(BookInfo bookInfo) {
        this.bookInfo = bookInfo;
    }
}
```

Die Tabellenstruktur enthält alle Attribute der eingebetteten Typen in einzelnen Spalten:

```
create table EmbeddedKeyFlatBook (publisherNumber varchar(255)
not null, bookNumber varchar(255) not null, languageCode
varchar(255) not null, checkNumber varchar(255) not null, title
varchar(255), description varchar(255), pages integer not null,
primary key (publisherNumber, bookNumber, languageCode,
checkNumber))
```

### 1.3.6 Vererbung

Entities können im Objektmodell in einer polymorphen Vererbungshierarchie stehen. Im Datenmodell kann dieses durch verschiedene Strategien nachgebildet werden:

- Eine Typspalte innerhalb einer Tabelle, die alle Attribute aller Subklassen gemeinsam enthält
- Pro Typ wird eine eigene Tabelle verwendet
- Basisklassen und Subklassen werden durch eine Relation miteinander verknüpft

#### 1.3.6.1 Single Table Inheritance

Book, SchoolBook und SpecialistBook stehen in einer einfachen Vererbungshierarchie, die durch eine einzige Tabelle abgebildet werden soll:

Book ist eine abstrakte Klasse, die den Primärschlüssel und die Basis-Attribute eines Buches definiert:

```
@Entity
@Table(name="SingleTableInheritanceBook")
public abstract class Book implements Serializable {
    private static final long serialVersionUID = 1L;

    private String isbn;

    private String title;

    @Id
    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

Um eine abgeleitete Klasse, z.B. `SchoolBook` definieren zu können, werden die folgenden Annotations benutzt:

- `@Inheritance` definiert die Strategie der Umsetzung in das Datenmodell durch die Enumeration `javax.persistence.InheritanceStrategy`. Für das hier gewählte Beispiel wird `SINGLE_TABLE` gewählt:
- Die zur Typ-Unterscheidung benutzte Spalte wird mit `@DiscriminatorColumn` definiert. Hier erfolgt die Angabe des Spaltennamens sowie des zu verwendende Typs
- Die Typinformation erfolgt durch die Angabe eines Wertes der Spalte. Dieser wird mit dem `@DiscriminatorValue` angegeben

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="book_type",
discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("School")
public class SchoolBook extends Book{

    private static final long serialVersionUID = 1L;
    private byte year;
    private String subject;
    public String getSubject() {
        return subject;
    }
    public void setSubject(String subject) {
        this.subject = subject;
    }
    public byte getYear() {
        return year;
    }
    public void setYear(byte year) {
        this.year = year;
    }
}
```

Die weitere Subklasse `SpecialistBook` wird analog definiert. Allerdings wird hier nun natürlich ein anderes `@DiscriminatorValue` benutzt:

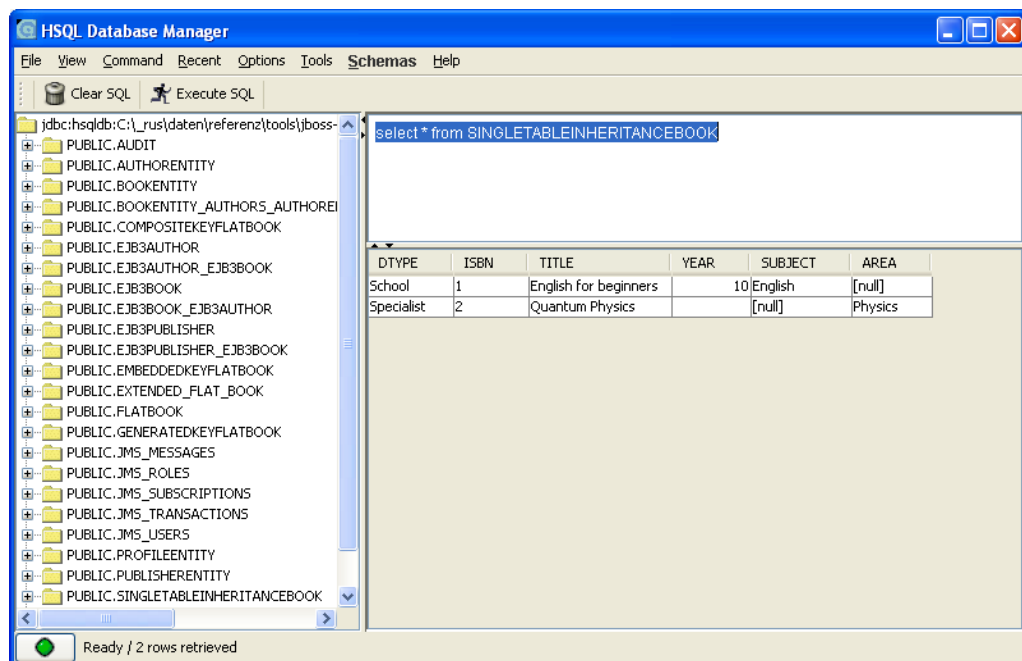
```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="book_typ",
discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("Specialist")
public class SpecialistBook extends Book{

    private static final long serialVersionUID = 1L;
    private String area;
    public String getArea() {
        return area;
    }
    public void setArea(String subject) {
        this.area = subject;
    }
}
```

Die zugehörige Tabellendefinition erfordert für die Attribute der Subklassen Spaltendefinitionen mit möglichen null-Werten:

```
create table SingleTableInheritanceBook (DTYPE varchar(31) not
null, isbn varchar(255) not null, title varchar(255), year
tinyint, subject varchar(255), area varchar(255), primary key
(isbn))
```

Die exemplarisch angelegten Datensätze für ein `SchoolBook` und ein `SpecialistBook` füllen die Tabelle mit den jeweils vorhandenen Attributen:



### 1.3.6.2 Table per Class

Mit dieser Strategie wird pro nicht-abstrakte Klasse der Vererbungshierarchie eine eigene Tabelle angelegt:

```
@Entity(name="TablePerClassBook")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Book implements Serializable {
    private static final long serialVersionUID = 1L;

    private String isbn;

    private String title;

    @Id
    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

```
@Entity(name="TablePerClassSchoolBook")
@Table(name="TablePerClassSchoolBook")
public class SchoolBook extends Book {

    private static final long serialVersionUID = 1L;

    private byte year;

    private String subject;

    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public byte getYear() {
        return year;
    }

    public void setYear(byte year) {
        this.year = year;
    }
}
```

```
@Entity(name="TablePerClassSpecialistBook")
@Table(name="TablePerClassSpecialistBook")
public class SpecialistBook extends Book{

    private static final long serialVersionUID = 1L;
    private String area;
    public String getArea() {
        return area;
    }
    public void setArea(String subject) {
        this.area = subject;
    }
}
```

Die beiden Tabellen enthalten jeweils alle Attribute:

```
create table TablePerClassSchoolBook (isbn varchar(255) not
null, title varchar(255), year tinyint not null, subject
varchar(255), primary key (isbn))
create table TablePerClassSpecialistBook (isbn varchar(255) not
null, title varchar(255), area varchar(255), primary key
(isbn))
```

### 1.3.6.3 Vererbung durch Relationen

Hier wird die Vererbung durch relational verknüpfte Tabellen aufgebaut.

```
@Entity(name="JoinTableBook")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Book implements Serializable {
    private static final long serialVersionUID = 1L;

    private String isbn;

    private String title;

    @Id
    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

```
@Entity(name="JoinTableSchoolBook")
@Table(name="JoinTableSchoolBook")
public class SchoolBook extends Book {

    private static final long serialVersionUID = 1L;

    private byte year;

    private String subject;

    public String getSubject() {
        return subject;
    }

    public void setSubject(String subject) {
        this.subject = subject;
    }

    public byte getYear() {
        return year;
    }

    public void setYear(byte year) {
        this.year = year;
    }
}
```

```
@Entity(name="JoinTableSpecialistBook")
@Table(name="JoinTableSpecialistBook")
public class SpecialistBook extends Book{

    private static final long serialVersionUID = 1L;
    private String area;
    public String getArea() {
        return area;
    }
    public void setArea(String subject) {
        this.area = subject;
    }
}
```

Jetzt wird auch für die abstrakte Basisklasse eine Tabelle angelegt. Dadurch werden redundante Spaltendefinitionen vermieden:

```
create table JoinTableBook (isbn varchar(255) not null, title
varchar(255), primary key (isbn))
```

```
create table JoinTableSchoolBook (isbn varchar(255) not null,
year tinyint not null, subject varchar(255), primary key
(isbn))
```

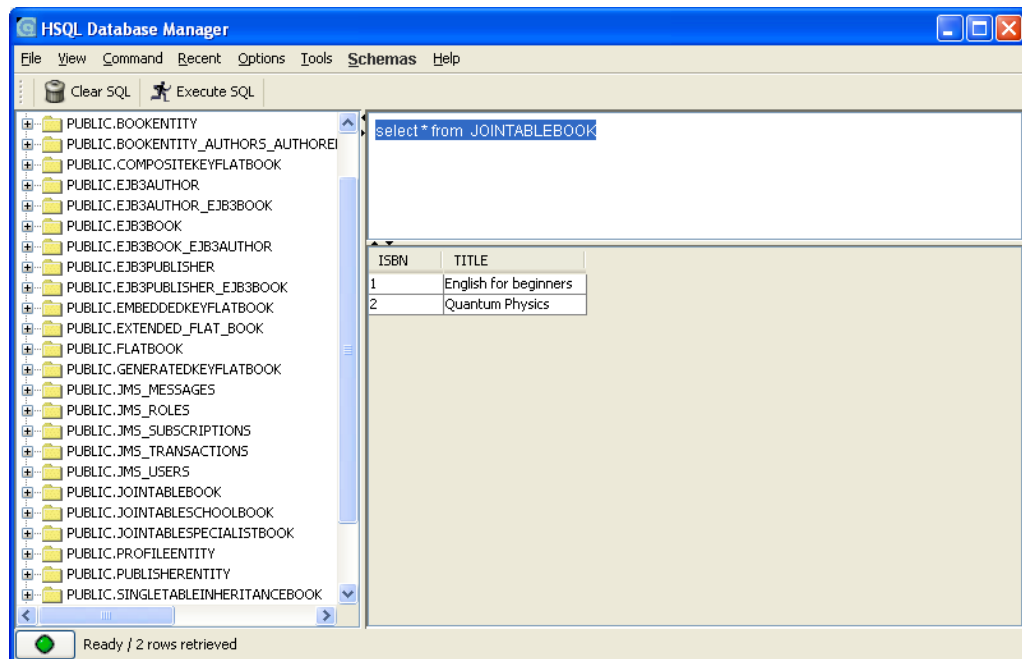
```
create table JoinTableSpecialistBook (isbn varchar(255) not
null, area varchar(255), primary key (isbn))
```

Dazu kommen noch die beiden Constraints, die die Relation aufbauen:

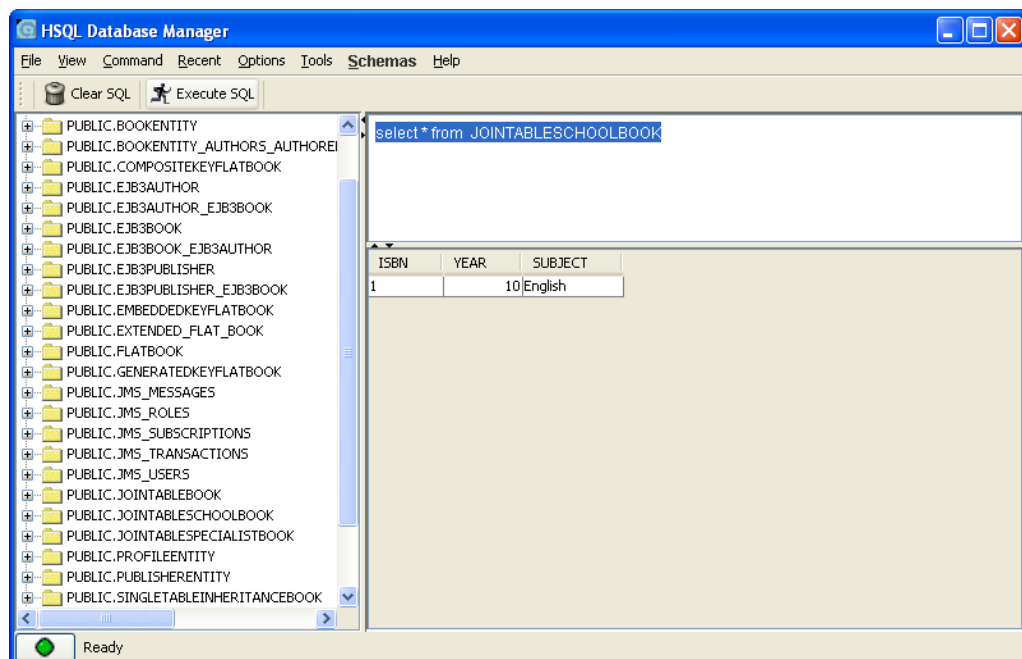


```
alter table JoinTableSchoolBook add constraint  
FK7E47C841D158C7C8 foreign key (isbn) references JoinTableBook  
alter table JoinTableSpecialistBook add constraint  
FKCFE6357ED158C7C8 foreign key (isbn) references JoinTableBook
```

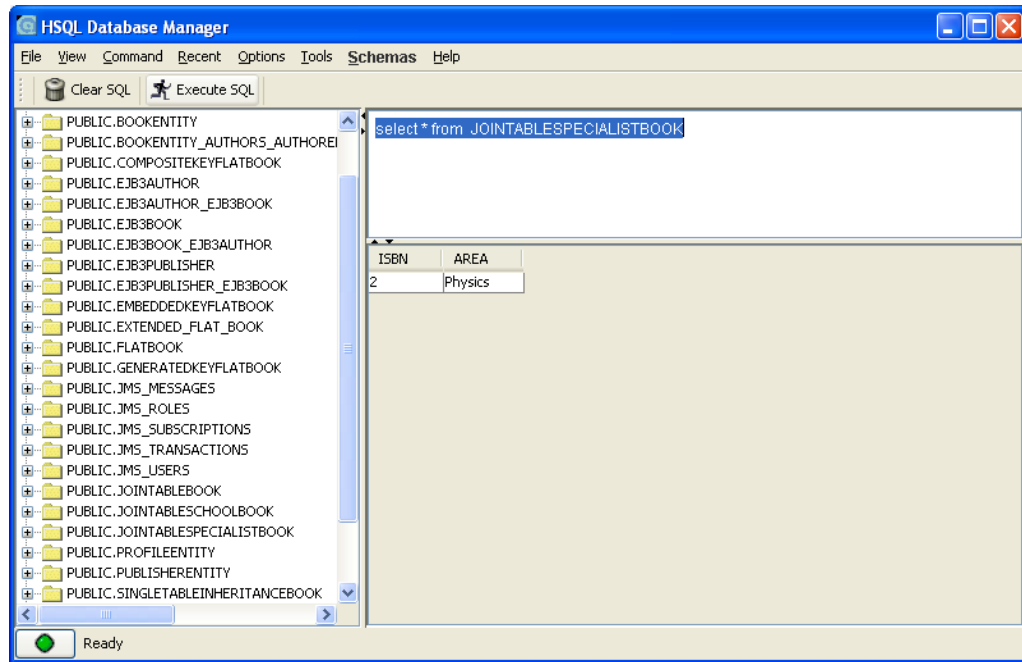
Die Einträge in der Haupttabelle lauten:



Die Schulbuch-Tabelle enthält den folgende Eintrag:



Das Fachbuch analog:



### 1.3.7 Mehrere Tabellen pro Entity

Eine Entity kann auch auf mehrere Tabellen umgesetzt werden. Dazu dient die Annotation `@SecondaryTable`. Im folgenden Beispiel hat jedes Buch eine Verfügbarkeit, die in einer anderen Tabelle (`BOOK_STOCK`) abgelegt ist:

```
@Entity
@SecondaryTable(name="BOOK_STOCK",
pkJoinColumns={@PrimaryKeyJoinColumn(name="isbn")})
public class MultiTableBook implements Serializable {
    private static final long serialVersionUID = 1L;

    private String isbn;

    private String title;

    private boolean available;

    @Column(name="COL_AVAILABLE", table="BOOK_STOCK")
    public boolean isAvailable() {
        return available;
    }

    public void setAvailable(boolean available) {
        this.available = available;
    }

    @Id
    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
}
```

```
public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}
}
```

Die Einträge für ein `MultiTableBook` sind auf zwei Tabellen verteilt:

```
create table MultiTableBook (isbn varchar(255) not null, title
varchar(255), primary key (isbn))

create table BOOK_STOCK (COL_AVAILABLE bit, isbn varchar(255)
not null, primary key (isbn))
```

### 1.3.8 Relationen

#### 1.3.8.1 One-to-One

Ein Mapping einer Entity auf mehrere Tabellen ist eine Vorstufe zu einer One-to-One-Relation. Allerdings werden für eine echte Relation zwei Entitäten benötigt. So hat im Folgenden jedes Buch eine Verkaufsstatistik:

```
@Entity(name="OneToOneRelationBook")
public class Book implements Serializable {
    private static final long serialVersionUID = 1L;

    private String isbn;

    private String title;

    private BookStatistics bookStatistics;

    @Id
    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @OneToOne(optional=false, cascade={CascadeType.ALL})
    @JoinColumn()
```

```
        name="BOOK_STATISTICS_ID", unique=true, nullable=false,
updatable=false)
    public BookStatistics getBookStatistics() {
        return bookStatistics;
    }

    public void setBookStatistics(BookStatistics bookStatistics)
    {
        this.bookStatistics = bookStatistics;
    }
}
```

```
@Entity(name="OneToOneBookStatistics")
public class BookStatistics implements Serializable{

    private Long id;

    private int sold;

    public int getSold() {
        return sold;
    }

    public void setSold(int sold) {
        this.sold = sold;
    }

    private Book book;

    @OneToOne(optional=false, mappedBy="bookStatistics")
    public Book getBook() {
        return book;
    }

    public void setBook(Book book) {
        this.book = book;
    }

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
}
```

Die erzeugten Tabellen und Constraints lauten:

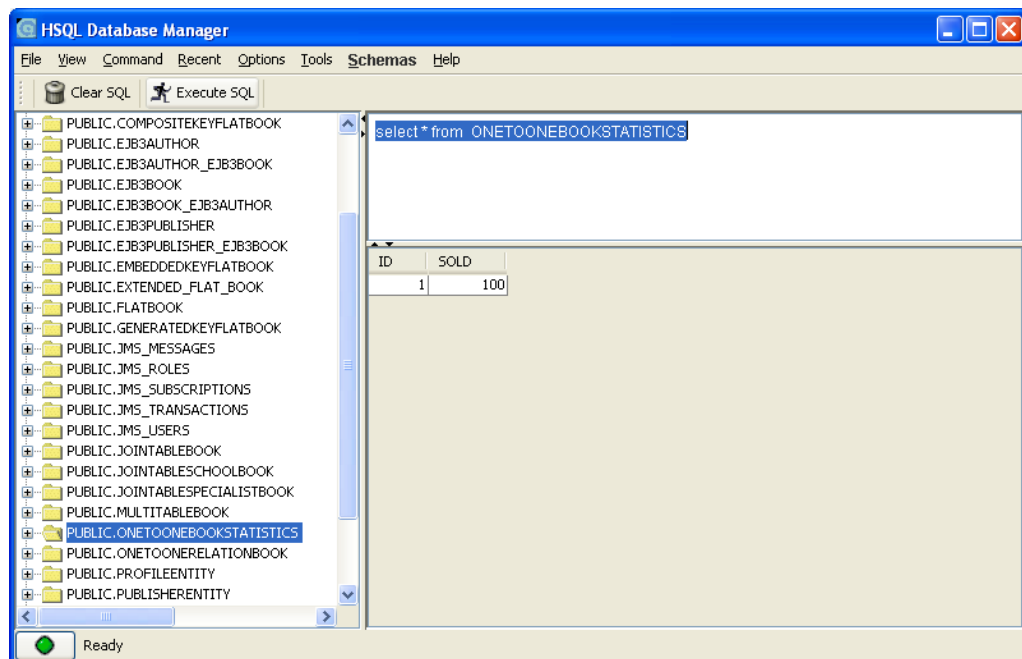
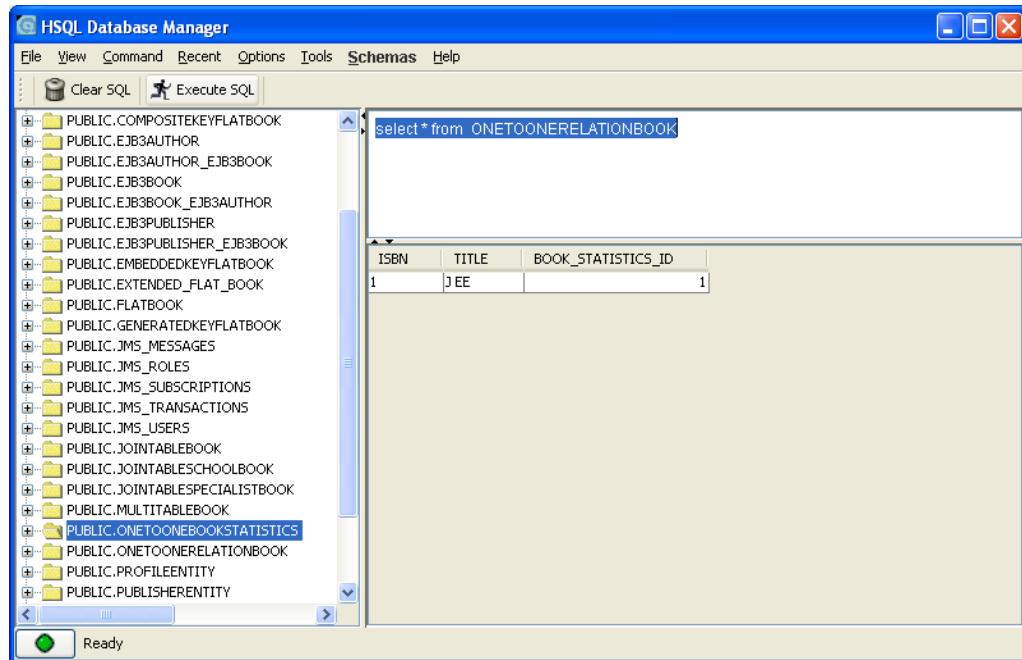
```
create table OneToOneRelationBook (isbn varchar(255) not null,  
title varchar(255), BOOK_STATISTICS_ID bigint not null, primary  
key (isbn), unique (BOOK_STATISTICS_ID))  
  
create table OneToOneBookStatistics (id bigint generated by  
default as identity (start with 1), sold integer not null,  
primary key (id))  
  
alter table OneToOneRelationBook add constraint  
FK897F06EAF23BD925 foreign key (BOOK_STATISTICS_ID) references  
OneToOneBookStatistics
```

Für die Definition der hier verwendeten bidirektionalen Relation sind die folgenden Informationen vorhanden:

- Der jeweilige Typ der Entity ist durch den Typ der Property bekannt
- Durch die `@OneToOne`-Annotation auf Seite der `Book`-Entity wird die Fremdschlüssel-Spalte angegeben. Damit bei der Neuanlage eines Buches auch die Statistik angelegt wird, wird hier die Kaskadierung aktiviert (`cascade=CascadTyp.ALL`)
- Der Fremdschlüssel wird durch `@JoinColumn` bestimmt. Nullwerte sind bei der One-to-One-Relation nicht zulässig, der Wert wird durch die Relation bestimmt und ist deshalb bei Aktualisierungen nicht zu berücksichtigen. Die Spalte enthält exakt einen Verweis auf die `OneToOneBookStatistics`-Tabelle und muss deshalb eindeutig sein
- Auf Seiten der `BookStatistic` wird in der `@OneToOne` angegeben, dass jedes Buch eine Statistik haben muss (`optional = false`). Alle weiteren Informationen die Relation betreffend können aus der Annotation auf der `Book`-Seite gelesen werden. Deshalb wird der Name der Property noch benötigt (`mappedBy="bookStatistics"`)

Das Einfügen von Daten erfolgt exemplarisch durch die folgende Testmethode:

```
public void testOneToOneEntity() {  
    Book book = new Book();  
    book.setIsbn("1");  
    BookStatistics bookStatistics = new BookStatistics();  
    bookStatistics.setSold(100);  
    book.setTitle("J EE");  
    book.setBookStatistics(bookStatistics);  
    bookStatistics.setBook(book);  
    entityManager.persist(book);  
}
```



### 1.3.8.2 One-to-Many

Die Definition einer One-to-Many-Relation benutzt in der Objektorientierten Welt das Collection-API. Im folgenden Beispiel hat ein Verleger eine Menge von Büchern:

```
@Entity(name="OneToManyRelationBook")
public class Book implements Serializable {
    private static final long serialVersionUID = 1L;

    private Long isbn;

    private String title;

    private Publisher publisher;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Long getIsbn() {
        return isbn;
    }

    public void setIsbn(Long isbn) {
        this.isbn = isbn;
    }

    @ManyToOne
    @JoinColumn(name="PUBLISHER_ID", nullable=false)
    public Publisher getPublisher() {
        return publisher;
    }

    public void setPublisher(Publisher publisher) {
        this.publisher = publisher;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public boolean equals(Object obj) {
        return EqualsBuilder.reflectionEquals(this, obj);
    }
    @Override
    public int hashCode() {
        return title.hashCode();
    }
    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this,
        ToStringStyle.MULTI_LINE_STYLE);
    }
}
@Entity
public class Publisher implements Serializable{
```

```

private static final long serialVersionUID = 1L;
private Long publisherId;
private String publisherName;
private Set<Book> books;

@OneToMany(cascade=CascadeType.ALL, mappedBy="publisher")
public Set<Book> getBooks() {
    return books;
}
public void setBooks(Set<Book> books) {
    this.books = books;
}

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
public Long getPublisherId() {
    return publisherId;
}
public void setPublisherId(Long publisherId) {
    this.publisherId = publisherId;
}
public String getPublisherName() {
    return publisherName;
}
public void setPublisherName(String publisherName) {
    this.publisherName = publisherName;
}
@Override
public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj);
}
@Override
public int hashCode() {
    return publisherName.hashCode();
}
@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this,
ToStringStyle.MULTI_LINE_STYLE);
}
}

```

Die Tabellendefinitionen und Constraints lauten:

```

create table Publisher (publisherId bigint generated by default
as identity (start with 1), publisherName varchar(255), primary
key (publisherId))

create table OneToManyRelationBook (isbn bigint generated by
default as identity (start with 1), title varchar(255),
PUBLISHER_ID bigint not null, primary key (isbn))

alter table OneToManyRelationBook add constraint
FKB0D3388598F3FFB foreign key (PUBLISHER_ID) references
Publisher

```



Das Hinzufügen von Büchern zu einem Verleger erfolgt, wie bereits erwähnt, durch das Collection-API. Es ist deshalb absolut notwendig, für korrekte Implementierungen von `equals` und `hashCode` zu sorgen, da ansonsten Einträge mehrfach im Set vorhanden sein könnten, was dann zu Fehlern bei der Umsetzung in das relationale Datenmodell führen würde. Hier werden zur Vereinfachung einige Klassen des Apache Commons Language-Paktes verwendet.

Das Einfügen von Daten erfolgt exemplarisch durch die folgende Testmethode:

```
public void testOneToManyEntity() {
    Publisher publisher = new Publisher();
    publisher.setPublisherName("Addison");

    org.javacream.demo.ejb3.persistence.entity.relation.one_to_many.Book book1 = new
org.javacream.demo.ejb3.persistence.entity.relation.one_to_many
.Book();
    book1.setTitle("Book1");
    book1.setPublisher(publisher);

    org.javacream.demo.ejb3.persistence.entity.relation.one_to_many.Book book2 = new
org.javacream.demo.ejb3.persistence.entity.relation.one_to_many
.Book();
    book2.setTitle("Book2");
    book2.setPublisher(publisher);

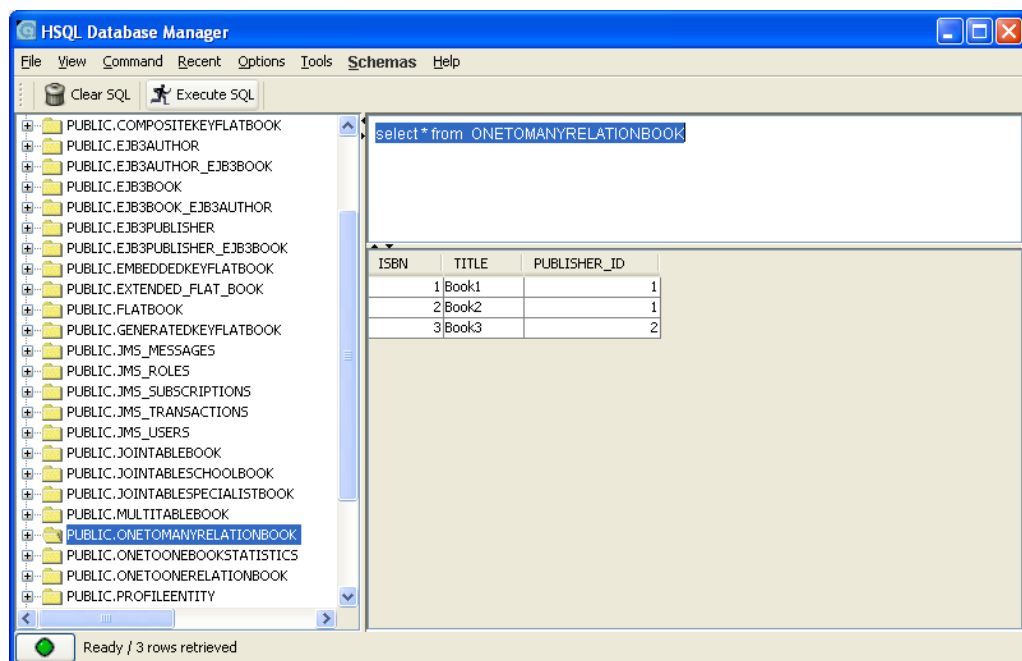
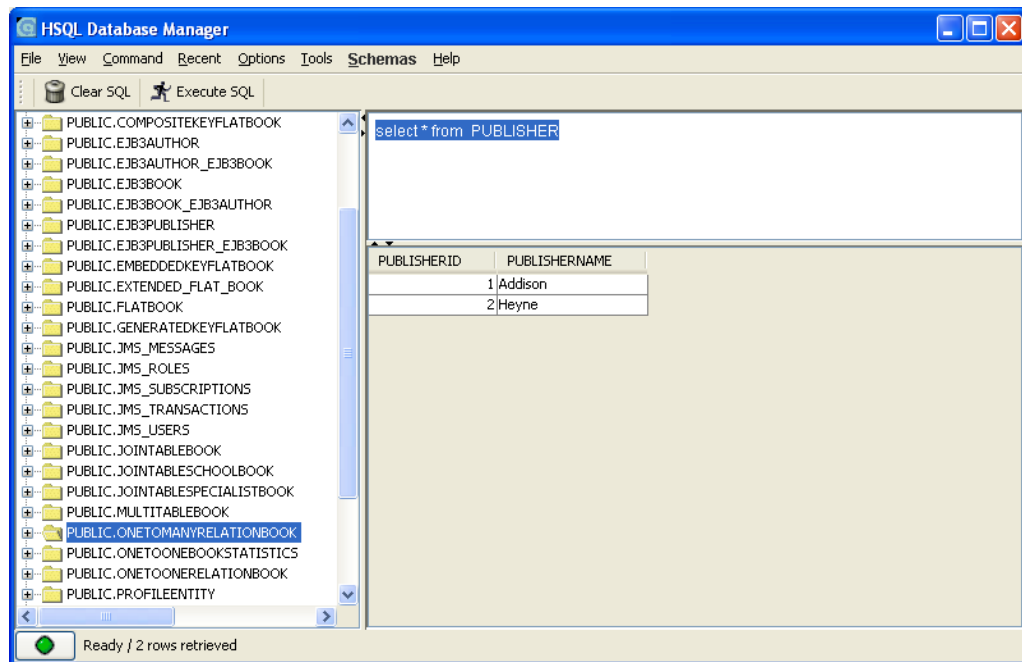
    HashSet<org.javacream.demo.ejb3.persistence.entity.relation.o
ne_to_many.Book> books = new
HashSet<org.javacream.demo.ejb3.persistence.entity.relation.one
_to_many.Book>();
    books.add(book1);
    books.add(book2);
    publisher.setBooks(books);

    Publisher publisher2 = new Publisher();
    publisher2.setPublisherName("Heyne");

    org.javacream.demo.ejb3.persistence.entity.relation.one_to_ma
ny.Book book3 = new
org.javacream.demo.ejb3.persistence.entity.relation.one_to_many
.Book();
    book3.setTitle("Book3");
    book3.setPublisher(publisher2);

    HashSet<org.javacream.demo.ejb3.persistence.entity.relation.o
ne_to_many.Book> books2 = new
HashSet<org.javacream.demo.ejb3.persistence.entity.relation.one
_to_many.Book>();
    books2.add(book3);
    publisher.setBooks(books);
    publisher2.setBooks(books2);

    entityManager.persist(publisher);
    entityManager.persist(publisher2);
}
```



### 1.3.8.3 Many-to-Many

Auch Many-to-Many-Relationen benutzen in der Objektorientierten Welt das Collection-API. Im folgenden Beispiel hat jedes Buch eine Liste von Büchern:

```
@Entity(name = "ManyToManyRelationBook")
public class Book implements Serializable {
    private static final long serialVersionUID = 1L;

    private Long isbn;

    private String title;

    private List<Author> authors;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Long getIsbn() {
        return isbn;
    }

    public void setIsbn(Long isbn) {
        this.isbn = isbn;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public boolean equals(Object obj) {
        return EqualsBuilder.reflectionEquals(this, obj);
    }

    @Override
    public int hashCode() {
        return title.hashCode();
    }

    @Override
    public String toString() {
        return ToStringBuilder.reflectionToString(this,
            ToStringStyle.MULTI_LINE_STYLE);
    }

    @ManyToMany(cascade = { CascadeType.ALL })
    @JoinTable(name = "BOOKS_AUTHORS", joinColumns = {
    @JoinColumn(name = "ISBN") }, inverseJoinColumns = {
    @JoinColumn(name = "AUTHOR_ID") })
    public List<Author> getAuthors() {
        return authors;
    }

    public void setAuthors(List<Author> authors) {
```

```
        this.authors = authors;
    }
}
```

```
@Entity
public class Author implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    private Long authorId;

    private String authorName;

    private Set<Book> books;

    @ManyToMany(cascade = { CascadeType.ALL }, mappedBy =
"authors")
    public Set<Book> getBooks() {
        return books;
    }

    public void setBooks(Set<Book> books) {
        this.books = books;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Long getAuthorId() {
        return authorId;
    }

    public void setAuthorId(Long publisherId) {
        this.authorId = publisherId;
    }

    public String getAuthorName() {
        return authorName;
    }

    public void setAuthorName(String publisherName) {
        this.authorName = publisherName;
    }

    @Override
    public boolean equals(Object obj) {
        return EqualsBuilder.reflectionEquals(this, obj);
    }

    @Override
    public int hashCode() {
        return authorName.hashCode();
    }

    @Override
```

```
public String toString() {  
    return ToStringBuilder.reflectionToString(this,  
        ToStringStyle.MULTI_LINE_STYLE);  
}  
}
```

Die Tabellendefinitionen und Constraints lauten:

```
create table Author (authorId bigint generated by default as  
identity (start with 1), authorName varchar(255), primary key  
(authorId))  
  
create table ManyToManyRelationBook (isbn bigint generated by  
default as identity (start with 1), title varchar(255), primary  
key (isbn))  
  
create table BOOKS_AUTHORS (ISBN bigint not null, AUTHOR_ID  
bigint not null)  
  
alter table BOOKS_AUTHORS add constraint FK4B34373F524F3CF  
foreign key (ISBN) references ManyToManyRelationBook  
  
alter table BOOKS_AUTHORS add constraint FK4B34373DC9D87CA  
foreign key (AUTHOR_ID) references Author
```

Das Einfügen von Daten erfolgt exemplarisch durch die folgende Testmethode:

```
public void testManyToManyEntity() {  
  
    org.javacream.demo.ejb3.persistence.entity.relation.many_to_  
many.Book book1 = new  
org.javacream.demo.ejb3.persistence.entity.relation.many_to_  
many.Book();  
    book1.setTitle("Book1");  
  
    org.javacream.demo.ejb3.persistence.entity.relation.many_to_  
many.Book book2 = new  
org.javacream.demo.ejb3.persistence.entity.relation.many_to_  
many.Book();  
    book2.setTitle("Book2");  
  
    org.javacream.demo.ejb3.persistence.entity.relation.many_to_  
many.Book book3 = new  
org.javacream.demo.ejb3.persistence.entity.relation.many_to_  
many.Book();  
    book3.setTitle("Book3");  
  
    Author author = new Author();  
    author.setAuthorName("Author1");  
  
    Author author2 = new Author();  
    author2.setAuthorName("Author2");  
  
    HashSet<org.javacream.demo.ejb3.persistence.entity.relation.  
many_to_many.Book> booksOfAuthor1 = new
```

```

HashSet<org.javacream.demo.ejb3.persistence.entity.relation.
many_to_many.Book>();
    booksOfAuthor1.add(book1);
    booksOfAuthor1.add(book2);

    HashSet<org.javacream.demo.ejb3.persistence.entity.relation.
many_to_many.Book> booksOfAuthor2 = new
HashSet<org.javacream.demo.ejb3.persistence.entity.relation.
many_to_many.Book>();
    booksOfAuthor2.add(book2);
    booksOfAuthor2.add(book3);

    author.setBooks(booksOfAuthor1);
    author2.setBooks(booksOfAuthor2);

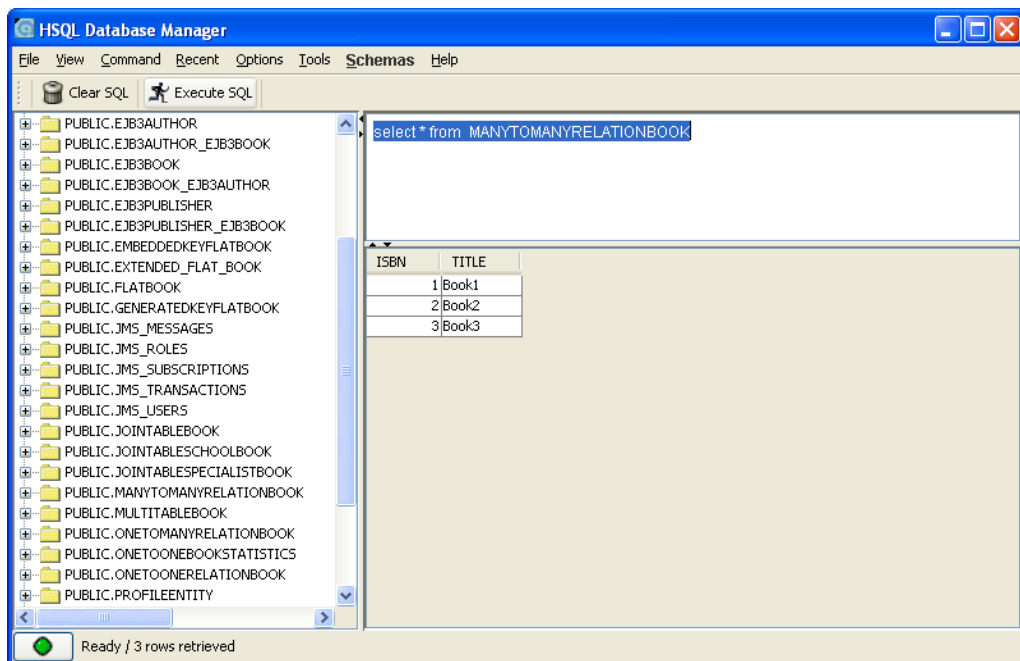
    ArrayList<Author> authorsOfBook1 = new ArrayList<Author>();
    authorsOfBook1.add(author);
    book1.setAuthors(authorsOfBook1);

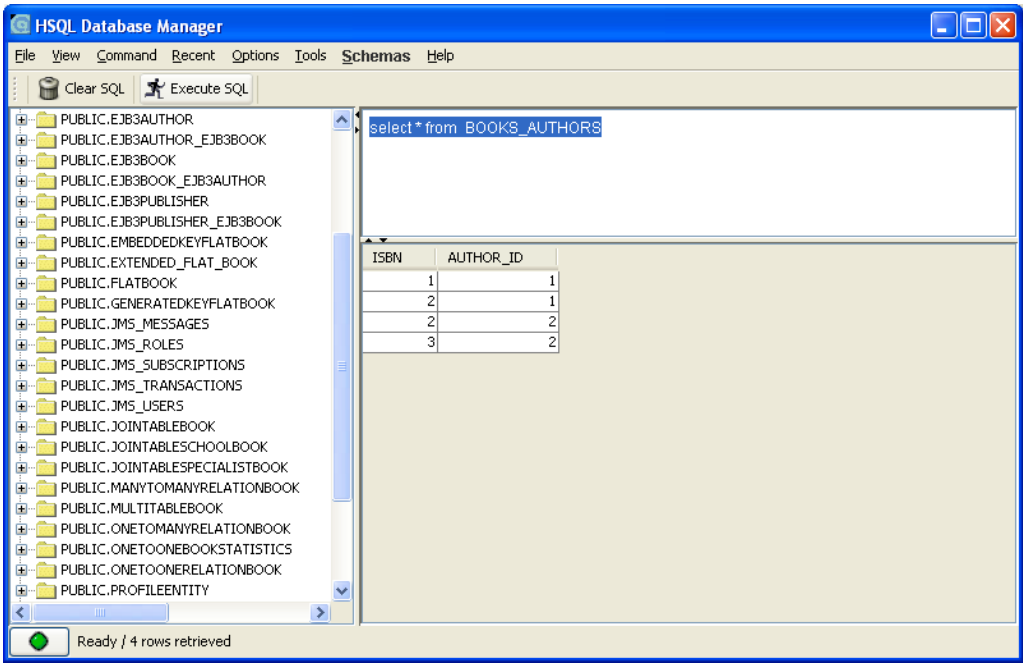
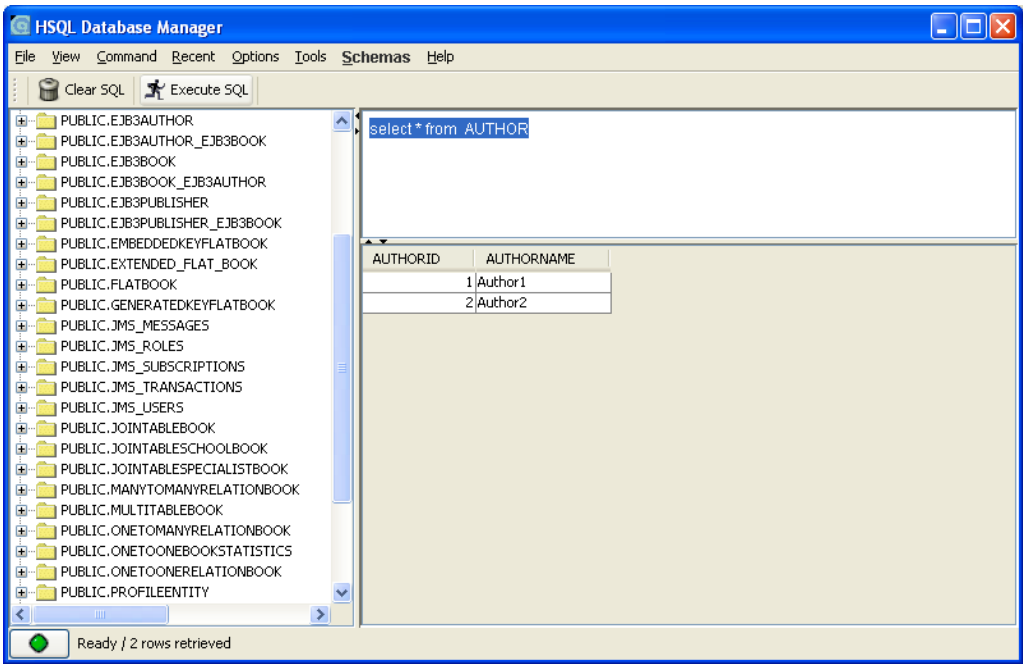
    ArrayList<Author> authorsOfBook2 = new ArrayList<Author>();
    authorsOfBook2.add(author);
    authorsOfBook2.add(author2);
    book2.setAuthors(authorsOfBook2);

    ArrayList<Author> authorsOfBook3 = new ArrayList<Author>();
    authorsOfBook3.add(author2);
    book3.setAuthors(authorsOfBook3);

    entityManager.persist(book1);
    entityManager.persist(book2);
}

```





#### 1.3.8.4 Ausblick

Die Beispiele in den vorherigen Kapiteln zeigen den Datenzugriff mit dem neuen Persistenz-API. Allerdings sind die Konfigurationen noch nicht optimiert eingestellt. Begriffe wie

- Sortierung
- Kaskadierung
- Lazy Loading
- Fetching-Strategien
- Massen-Updates und -Deletes
- Natives SQL

wurden noch nicht ausführlich gezeigt. Diese Themen werden im Uni-log-Seminar 33014: "e-Business-Anwendungen mit J EE" vertieft.



## 1.4 Die EJB Query Language

### 1.4.1 Datenbestand

So wie die meisten O/R-Mapper definiert auch das Java Persistence API eine eigene Query-Sprache. Die Syntax dieser Sprache ist zwar angelehnt an Standard-SQL, enthält aber auch Elemente der Objektorientierung:

- Objektorientierte Formulierung der Abfragen
- Polymorphe Abfragen

Die Beispiele orientieren sich nach den in den vorherigen Kapiteln eingeführten Entitäten und benutzen den folgenden Datenbestand:

Es gibt 4 Verleger:

Tabelle QUERYPUBLISHER

PUBLISHERID	PUBLISHERNAME
1	Publisher1
2	Publisher2
3	Publisher3
4	Publisher4

10 Autoren schreiben Bücher:

Tabelle QUERYAUTHOR

AUTHORID	AUTHORNAME
1	Author1
2	Author2
3	Author3
4	Author4
5	Author5
6	Author6
7	Author7
8	Author8
9	Author9
10	Author10

Diese schreiben insgesamt 40 Bücher, die von den 10 Verlegern aufgelegt werden. Jedes Buch hat weiterhin seine Verkaufsstatistik.

Tabelle QEURYBOOKS

ISBN	TITLE	PUBLISHER_ID	BOOK_STATISTIC_ID
1	Book1	1	1
2	Book2	1	2
3	Book3	1	3
4	Book4	1	4
5	Book5	1	5
6	Book6	1	6
7	Book7	1	7
8	Book8	1	8
9	Book9	1	9
10	Book10	1	10
11	Book11	2	11
12	Book12	2	12
13	Book13	2	13
14	Book14	2	14
15	Book15	2	15
16	Book16	2	16
17	Book17	2	17
18	Book18	2	18
19	Book19	2	19
20	Book20	2	20
21	Book21	3	21
22	Book22	3	22
23	Book23	3	23
24	Book24	3	24
25	Book25	3	25
26	Book26	3	26
27	Book27	3	27
28	Book28	3	28
29	Book29	3	29
30	Book30	3	30
31	Book31	4	31
32	Book32	4	32
33	Book33	4	33
34	Book34	4	34

35	Book35	4	35
36	Book36	4	36
37	Book37	4	37
38	Book38	4	38
39	Book39	4	39
40	Book40	4	40

Jeweils 10 Bücher sind somit einem Verleger zugeordnet.

Tabelle QUERYBOOKSTATISTICS

ID	SOLD	ID	SOLD	ID	SOLD	ID	SOLD
1	0	2	1	3	2	4	3
5	4	6	5	7	6	8	7
9	8	10	9	11	10	12	11
13	12	14	13	15	14	16	15
17	16	18	17	19	18	20	19
21	20	22	21	23	22	24	23
25	24	26	25	27	26	28	27
29	28	30	29	31	30	32	31
33	32	34	33	35	34	36	35
37	36	38	37	39	38	40	39

Die Autoren schreiben die verschiedenen Bücher, wobei die Autoren mit der Id 4, 5 und 8 jeweils auch als Co-Autoren auftreten.

Tabelle QUERYBOOKS\_QUERY\_AUTHORS

ISBN	AUTHOR_ID	ISBN	AUTHOR_ID	ISBN	AUTHOR_ID	ISBN	AUTHOR_ID
1	1	1	4	1	5	1	8
2	1	3	1	4	1	4	4
5	2	5	5	6	2	7	2
7	4	8	2	8	8	9	3
9	5	10	3	10	4	11	3
12	3	13	4	13	5		
14	4	15	4	15	8		
16	4	17	5	18	5		
19	5	19	4	20	5	21	6
21	5	22	6	22	4	22	8
23	6	24	6	25	7	25	4
25	5	26	7	27	7	28	7
28	4	29	8	29	5		
30	8	31	8	31	4	32	8
33	9	33	5	34	9	34	4
35	9	36	9	36	8	37	10
37	4	37	5	38	10	39	10
40	10	40	4				

Für die folgenden Beispiele sind die Entitäten so konfiguriert, dass sämtliche Relationen automatisch geladen werden, der so genannte "Eager"-Fetch-Typ. Diese Einstellungen sind so für den Produktionsbetrieb bei größeren Datenmengen nicht sinnvoll bzw. führen schnell zu unerträglicher Performance bzw. Speicherproblemen. Die Optimierung der Zugriffe ist Bestandteil des Unilog-Seminars 33014: "e-Business-Anwendungen mit J EE".

### 1.4.2 Programm

Um die Abfragen von einem entfernten JUnit-Test ausführen lassen zu können wird die folgende SessionBean benutzt:

```
@Stateless
@Remote
public class QueryTestBean implements RemoteQueryTest {

    @PersistenceContext
    private EntityManager entityManager;

    public Object executeEjbql(String ejbql) {
        return entityManager.createQuery(ejbql).getResultList();
    }

    public Object executeParemetrizedEjbql(String ejbql, Object...
params) {
        Query query = entityManager.createQuery(ejbql);
        int paramCounter = 0;
        for (Object param : params) {
            paramCounter++;
            query.setParameter(paramCounter, param);
        }
        return query.getResultList();
    }
}
```

### 1.4.3 Suche alle Verleger

Die folgende Testmethode sucht alle Verleger:

```
public void searchAllPublishersQuery() {
    Object result = queryTest
        .executeEjbql("Select publisher from QueryPublisher as
publisher");
    Collection<Publisher> publishers = (Collection<Publisher>)
result;
    assertSame("There must be 4 Publishers!", 4,
publishers.size());
    for (Publisher publisher : publishers) {
        System.out.println(publisher.getPublisherName() + " has
"
            + publisher.getBooks().size() + " books");
    }
}
```

mit der Ausgabe:

```
Publisher1 has 10 books
Publisher2 has 10 books
Publisher3 has 10 books
Publisher4 has 10 books
```

#### 1.4.4 Suche einen bestimmten Verleger

Die folgende Testmethode sucht den Verleger mit der Id 1:

```
public void searchOnePublisherQuery() {
    Object result = queryTest
        .executeEjbql("Select publisher from QueryPublisher as
publisher where publisher.publisherId = 1");
    Collection<Publisher> publishers = (Collection<Publisher>)
result;
    assertEquals("There must be 1 Publishers!", 1,
publishers.size());
    for (Publisher publisher : publishers) {
        System.out.println(publisher.getPublisherName() + " has "
            + publisher.getBooks().size() + " books");
    }
}
```

mit der Ausgabe:

```
Publisher1 has 10 books
```

#### 1.4.5 Parametrisierte Suche nach einem Verleger

Die folgende Testmethode sucht den Verleger mit der Id 2. Dies erfolgt durch die Angabe eines nummerierten Platzhalters. Statt Nummern wären auch Namen zulässig (z.B. :id), was dann aber eine kompliziertere Methodensignatur der Test-SessionBean erfordert hätte. Dies ist hier aus Gründen der Vereinfachung nicht durchgeführt worden.

```
public void searchOnePublisherParameterizedQuery() {
    Object result = queryTest
        .executeParametrizedEjbql("Select publisher from
QueryPublisher as publisher where publisher.publisherId = ?1",
new Long(2));
    Collection<Publisher> publishers = (Collection<Publisher>)
result;
    assertEquals("There must be 1 Publishers!", 1,
publishers.size());
    for (Publisher publisher : publishers) {
        System.out.println(publisher.getPublisherName() + " has "
            + publisher.getBooks().size() + " books");
    }
}
```

mit der Ausgabe:

```
Publisher2 has 10 books
```

#### 1.4.6 Bedingungen unter Verwendung der Relationen

Die folgende Testmethode sucht alle Bücher, die mehr als 35-mal verkauft wurden:

```
public void searchBestsellers() {
    Object result = queryTest
        .executeParemetrizedEjbQl("Select book from QueryBook
as book where book.bookStatistics.sold > 35");
    Collection<Book> books = (Collection<Book>) result;
    assertEquals("There must be 4 matching Books!", 4,
books.size());
    for (Book book : books) {
        System.out.println(book.getTitle() + " is sold "
            + book.getBookStatistics().getSold() + " times");
    }
}
```

Mit der Ausgabe:

```
Book37 is sold 36 times
Book38 is sold 37 times
Book39 is sold 38 times
Book40 is sold 39 times
```

#### 1.4.7 Verknüpfte Bedingungen

Die folgende Testmethode sucht alle Bücher deren Titel mit "7" endet, die mehr als 35-mal verkauft wurden:

```
public void searchBestsellersWithTitleEndingWith7() {
    Object result = queryTest
        .executeParemetrizedEjbQl("Select book from QueryBook
as book where book.bookStatistics.sold > 35 and book.title like
'%7'");
    Collection<Book> books = (Collection<Book>) result;
    assertEquals("There must be 1 matching Book!", 1,
books.size());
    for (Book book : books) {
        System.out.println(book.getTitle() + " is sold "
            + book.getBookStatistics().getSold() + " times");
    }
}
```

mit der Ausgabe:

```
Book37 is sold 36 times
```

### 1.4.8 Joins

Die folgende Testmethode sucht alle Bücher des Autors "4", die mehr als 35-mal verkauft wurden:

```
public void searchBestsellersByAuthor4() {
    Object result = queryTest
        .executeParemetrizedEjbQl("Select book from QueryBook
as book join book.authors as author where
book.bookStatistics.sold > 35 and author.authorId=4");
    Collection<Book> books = (Collection<Book>) result;
    assertEquals("There must be 2 matching Books!", 2,
books.size());
    for (Book book : books) {
        System.out.println(book.getTitle() + " by " +
book.getAuthors() + " is sold "
            + book.getBookStatistics().getSold() + " times");
    }
}
```

mit der Ausgabe:

```
Book37 by [Author: id=10, name=Author10, Author: id=4,
name=Author4, Author: id=5, name=Author5] is sold 36 times
Book40 by [Author: id=10, name=Author10, Author: id=4,
name=Author4] is sold 39 times
```

### 1.4.9 Subqueries

Die folgende Testmethode sucht alle Bücher, die mehr als 3 Autoren haben:

```
public void searchBooksWihMoreThan3Authors() {
    Object result = queryTest
        .executeParemetrizedEjbQl("Select book from QueryBook
as book where (select count(authors) from book.authors as
authors) >= 3");
    Collection<Book> books = (Collection<Book>) result;
    assertEquals("There must be 4 matching Books!", 4,
books.size());
    for (Book book : books) {
        System.out.println(book.getTitle() + " has 3 or more
authors: " + book.getAuthors());
    }
}
```

mit der Ausgabe:

```
Book22 has 3 or more authors: [Author: id=6, name=Author6,
Author: id=4, name=Author4, Author: id=8, name=Author8]
Book25 has 3 or more authors: [Author: id=7, name=Author7,
Author: id=4, name=Author4, Author: id=5, name=Author5]
Book37 has 3 or more authors: [Author: id=10, name=Author10,
Author: id=4, name=Author4, Author: id=5, name=Author5]
```





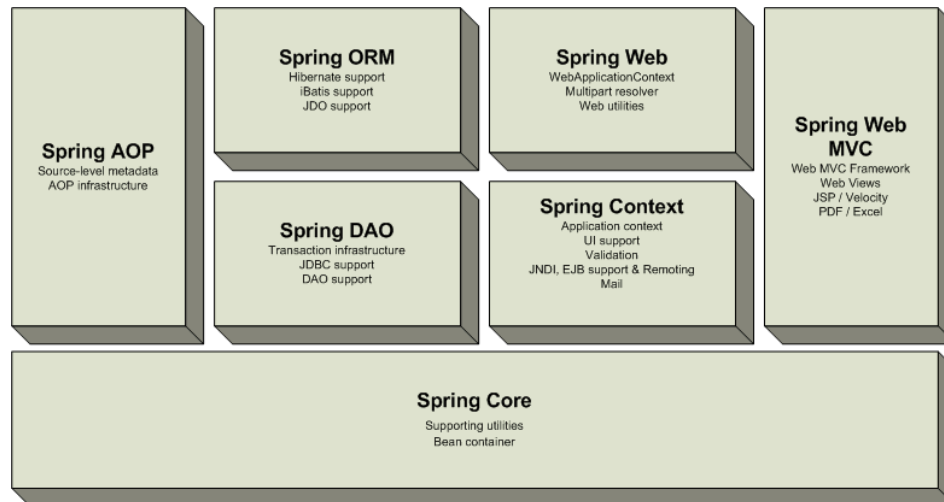
## 2 Spring – Eine Übersicht

Im vorherigen Abschnitt wurde dargelegt, dass in einem Service-orientierten Ansatz Abhängigkeiten durch eine externe Konfiguration beschrieben und als Implementierungen leichtgewichtige POJO-Klassen benutzt werden. Prinzipiell ist dies nichts wirklich Neues, sondern nur eine konsequente Umsetzung des Design Patterns „Strategy“. Benötigt werden nun eine Kontext-Implementierung sowie ein vernünftiges Format für die Konfigurationsdatei, bevorzugt mit Validierungs- und Visualisierungs-Funktionen.

### 2.1 Was ist das Spring-Framework?

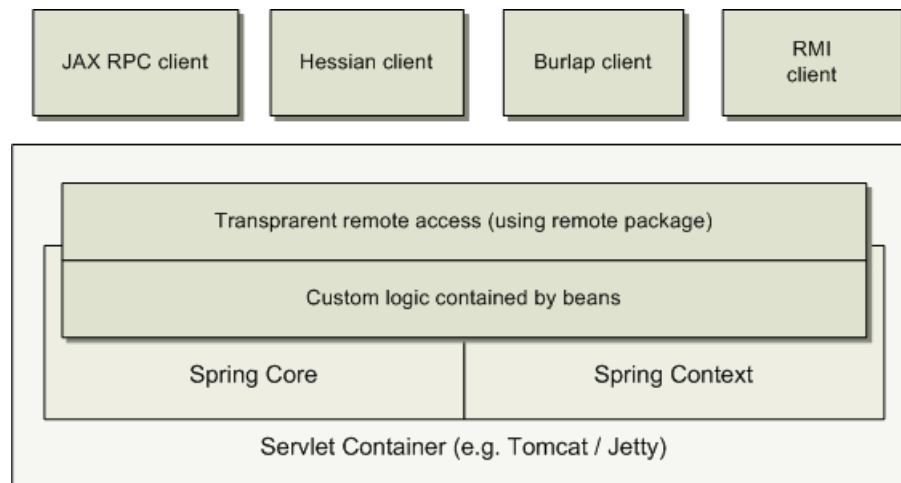
Die Antwort auf die in der Kapitelüberschrift gestellte Frage ist vielschichtig. Spring ist:

- Eine Bibliothek nützlicher Klassen:
  - Konfiguration
  - Internationalisierung
  - Validierung
  - Erstellung von Web-basierten Anwendungen (
  - Verteilte Anwendungen mit verschiedenen Kommunikationsprotokollen (RMI, http, SOAP, JMS)
  - Unterstützung eines Aspektorientierten Ansatzes
- Eine Container für POJOs, deren Properties und Abhängigkeiten über eine Konfigurationsdatei definiert werden können und durch Dependency Injection gesetzt werden
- Ein Meta-Framework für Enterprise Services, das alle Technologie-abhängigen Dienste durch ein verallgemeinertes API kapselt:
  - Transaktions-Management
  - O/R-Mapper bzw. Abstraktion von JDBC-Zugriffen
  - Authentifizierung
  - Java Message Service Provider
  - Mail-Server
- Container für J EE-Komponenten
  - Stateless SessionBeans
  - MessageDrivenBeans
  - Outbound Connectors

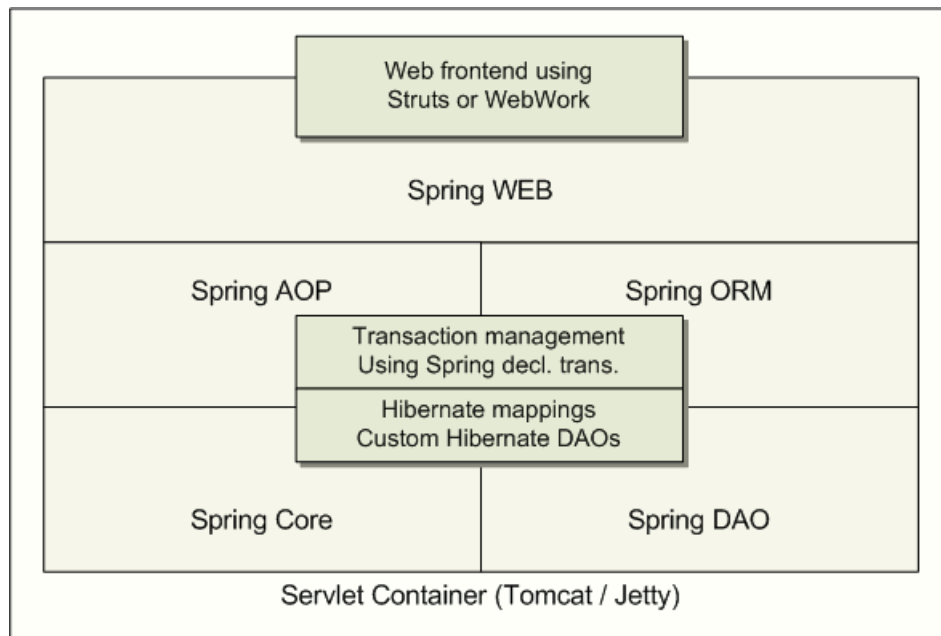


Spring deckt ein breites Spektrum an potenziellen Einsatzbereichen ab:

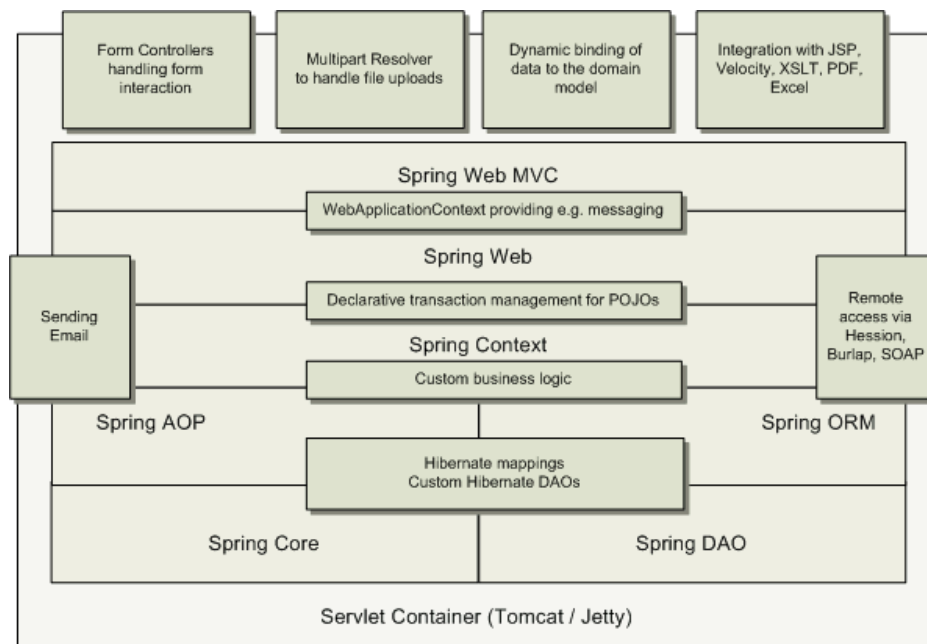
- Einsatz als komfortabler Container für POJOs in Standalone-Applikationen aber auch in einer Middleware-Schicht
- Verteilung von POJOs mit verschiedenen Netzwerk-Protokollen:



- Realisierung der Business- und Datenzugriffsschicht für ein Web Frontend:



- Realisierung einer kompletten Web-basierten Applikation:



Dies wird in der Organisation der Spring-Archive deutlich, die eine Aufteilung in die Spring-Basiskomponenten und zusätzliche Module vorsieht.

 spring-aop.jar	275 KB
 spring-beans.jar	310 KB
 spring-context.jar	145 KB
 spring-core.jar	152 KB
 spring-dao.jar	106 KB
 spring-jdbc.jar	223 KB
 spring-remoting.jar	178 KB
 spring-support.jar	191 KB
 spring-web.jar	160 KB
 spring-webmvc.jar	256 KB

## 2.2 Der POJO-Container

Ein zentrales Feature des Spring-Frameworks ist die externe Konfiguration der die Anwendung ausmachenden JavaBeans durch eine XML-basierte Definitionsdatei. Das Spring-Framework benutzt diese Datei konsequent dazu, alle Abhängigkeiten der Anwendung zu deklarieren und diese durch das Prinzip der „Dependency Injection“ zu setzen.

Das Design Pattern Dependency Injection bedeutet, dass eine Komponente alle Abhängigkeiten zu anderen Komponenten nicht selber auflöst, sondern von einem Container „von Außen“ setzen lässt. In Java ist die Umsetzung dieses Prinzips für eine Komponente trivial: Die Abhängigkeit ist ein `private` Attribut mit einer öffentlichen `set`-Methode.

Ein ähnlich gebrauchter und motivierter Begriff lautet „Inversion of Control“. Allerdings sind hier mögliche Implementierungs-Strategien nicht so evident und klar:

- Das „Method Template“ Pattern definiert Sequenzen in einer Superklasse und verlangt die Implementierung einer abstrakten Methode.
- Mit Hilfe des Command Patterns werden Sequenzen in ein Objekt verpackt und von einem Command Interpreter ausgeführt.
- Eine Template-Klasse führt eine Implementierung einer Schnittstelle, die eine Callback-Methode deklariert, aus und ruft intern den Callback auf.

Zentrales Element von Spring ist die Spring-Konfigurationsdatei, in der die „Spring Beans<sup>1</sup>“ definiert werden. Eine Spring Bean ist eine Kombination einer Java-Klasse, die keinerlei Abhängigkeiten zum Spring-Framework hat (also ein „plain old Java Object“, ein POJO ist) und ein Eintrag in einer Konfigurationsdatei, der die Properties und damit die Abhängigkeiten des POJOs definiert.

Eine einzelne Spring Bean wird definiert durch:

- eine eindeutige ID bzw. einen Namen
- ihren voll qualifizierten Klassen-Namen
- eine Liste von optionalen Namen, die als Alias benutzt werden können
- falls notwendig: Angabe einer `init`- oder `destroy`-Methode
- Setzen von Properties
  - mit `setter`-Methoden
  - mit Konstruktor-Argumenten

---

<sup>1</sup> „SpringBean“ ist ein Begriff, der als passende Analogie zu den „Enterprise JavaBeans“ geprägt wurde.

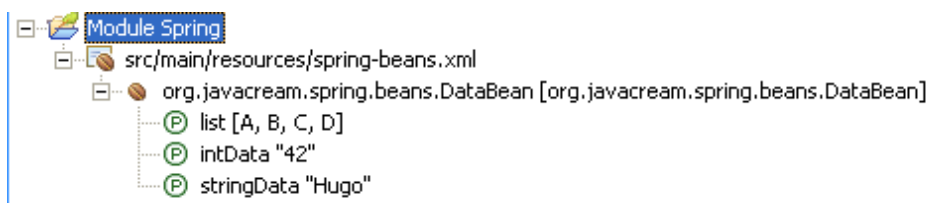
- Gültige Property-Typen sind:
  - Einfache Datentypen
  - String
  - Listen und Maps
  - Referenzen auf andere Spring-Beans
  - `java.util.Properties`
  - Beliebige Typen, für die ein `java.beans.PropertyEditor` registriert wurde
- die Spring-Container-Konfiguration:
  - Anzahl der zu erzeugenden Instanzen (`scope = <singleton|prototype|...>`). Falls der Nicht-Singleton-Scope gewählt wurde, wird jede Aufforderung einer Anwendung, eine Spring Bean bereit zustellen, von einer neuen Instanz.
  - Zeitpunkt der Bean-Erzeugung: Beim Initialisierung des Spring-Frameworks oder beim ersten Zugriff (`lazy-init = <true|false>`).
- Konfiguration des Spring-Frameworks
  - Automatisches Prüfen, ob alle Dependencies oder Properties von Spring Beans, die durch set-Methoden definiert sind, gesetzt wurden (`dependency-check = <all|simple|object|none>`).
  - Versuch, alle Dependencies durch Analyse und Typprüfung der konfigurierten Spring Beans aufzulösen (`autowire = <all|byName|byType|none>`). Hier wird versucht, alle Properties aller Spring Beans an Hand der Namen/Typen mit anderen Spring Beans zu verknüpfen.

Beispiel: Konfiguration einer einfachen „DataBean“:

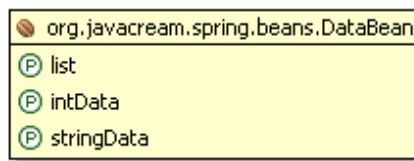
DataBean
<ul style="list-style-type: none"><li>- <code>intData: int</code></li><li>- <code>list: List &lt;E&gt;</code></li><li>- <code>stringData: String</code></li></ul>
<ul style="list-style-type: none"><li>+ <code>equals(Object): boolean</code></li><li>+ <code>hashCode(): int</code></li><li>+ <code>toString(): String</code></li></ul>

```
<beans
  default-autowire="no"
  default-lazy-init="false"
  default-dependency-check="none"
>
  <bean
    id="org.javacream.spring.beans.DataBean"
    class="org.javacream.spring.beans.DataBean"
    scope="singleton"
  >
    <property name="list">
      <list>
        <value>A</value>
        <value>B</value>
        <value>C</value>
        <value>D</value>
      </list>
    </property>
    <property name="intData">
      <value>42</value>
    </property>
    <property name="stringData">
      <value>Hugo</value>
    </property>
  </bean>
</beans>
```

Durch das XML-Format ist die Visualisierung dieser Date recht einfach möglich, was hier am Beispiel der Spring-IDE, ein Eclipse-Plugin, gezeigt wird:







Um diese Konfigurationsdatei zur Laufzeit in einen Baum von Java-Objekten zu verwandeln wird entweder eine `BeanFactory` oder aber der `ApplicationContext` benutzt<sup>2</sup>. Dieser benötigt als Eingabeparameter die Spring-Konfigurationsdatei, in der die Spring Bean-Definitionen stehen.

Die Methoden der `BeanFactory` bzw. des `ApplicationContext` lassen sich, wie dem unten dargestellten Klassendiagramm zu entnehmen ist, in verschiedene Gruppen unterteilen.

---

<sup>2</sup> Der `ApplicationContext` bietet im Vergleich zur `BeanFactory` eine erweiterte Funktionalität und sollte deshalb im Normalfall verwendet werden.



## 2.3 Value Properties

### 2.3.1 Die Bean-Definition

Jede Spring Bean hat

- eine innerhalb der Konfigurationsdatei eindeutige Id
  - einen Klassennamen
  - optional eine Liste von (Komma-getrennten) Namen
  - Die Möglichkeit, die Standard-Konfiguration, beispielsweise den Scope zu überschreiben<sup>3</sup>.

### 2.3.2 Default-Typen von Properties und deren Konfiguration

Wie weiter unten noch näher erläutert werden wird verwendet Spring intern die der JavaBean-Spezifikation entnommenen

`java.beans.PropertyEditor`-Schnittstelle bzw. geeigneten Implementierungen. Die folgenden Beispiele zeigen deren Konfiguration.

#### 2.3.2.1 Einfache Datentypen und String

Alle einfachen Java-Datentypen werden automatisch in den richtigen Wert umgewandelt. Die Typ-Information entnimmt Spring durch Inspektion der zu verwendenden Java-Klasse. Die folgenden Schreibweisen sind möglich:

```
<bean id="Demo" class="org.javacream.spring.beans.DemoBean">
    <property name="intData">
        <value>42</value>
    </property>
</bean>
```

oder alternativ:

```
<bean id="Demo" class="org.javacream.spring.beans.DemoBean">
    <property name="intData" value="42" />
</bean>
```

---

<sup>3</sup> Ab Spring 2 erweitert der scope das singleton-Attribut. Für Web-Anwendungen wurden die zusätzlichen Scopes "Request" und "Session" eingeführt.

### 2.3.2.2 Listen

Eine Liste wird durch das `list`-Tag beschrieben:

```
<bean id="Demo" class="org.javacream.spring.beans.DemoBean">
  <property name="list">
    <list>
      <value>A</value>
      <value>B</value>
      <value>C</value>
      <value>D</value>
    </list>
  </property>
</bean>
```

Bei Bedarf können diese Elemente (und auch die der folgenden Abschnitte) beliebig verschachtelt werden:

```
<bean id="Demo" class="org.javacream.spring.beans.DemoBean">
  <property name="list">
    <list>
      <value>A</value>
      <value>B</value>
      <value>C</value>
      <list>
        <value>1</value>
        <value>2</value>
        <value>3</value>
      </list>
    </list>
  </property>
</bean>
```

Hier enthält die Liste an vierter Stelle eine andere Liste.

### 2.3.2.3 Maps

Diese werden durch das `map`-Tag beschrieben, in dem intern `entry` und darin dann die Tags `key` und `value` zugelassen sind.

```
<bean id="Demo" class="org.javacream.spring.beans.DemoBean">
  <property name="map">
    <map>
      <entry>
        <key>
          <value>keyObject</value>
        </key>
        <value>42</value>
      </entry>
      <entry>
        <key>
          <value>keyObject2</value>
        </key>
        <value>4711</value>
      </entry>
    </map>
  </property>
</bean>
```

### 2.3.3 Nullwerte von Properties

Soll als Wert einer Property die `null` gesetzt werden, so geschieht dies durch das `<null/>`-Element:

```
<property name="stringProperty"><null/></property>
```

Im Unterschied dazu hätte die Property in der folgenden Konfiguration den Wert „“:

```
<property name="stringProperty"><value></value></property>
```

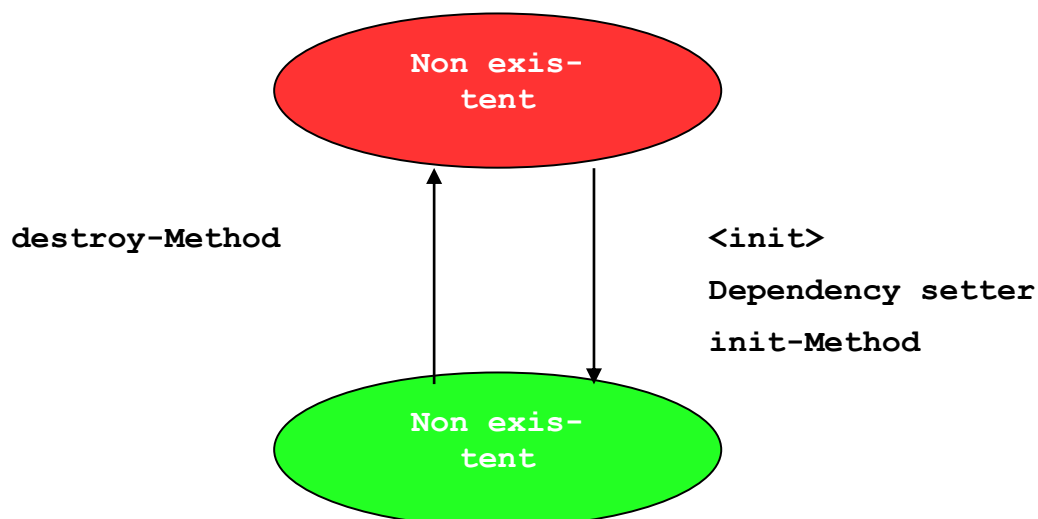
Dieses Element ist notwendig, falls der automatische Dependency check benutzt werden soll.

### 2.3.4 init- und destroy-Methoden

Benötigt eine Spring Bean eine Initialisierung und/oder eine Aufräum-Methode, so kann der `ApplicationContext` diese Methoden via Reflection aufrufen, falls sie in der Konfigurationsdatei eingetragen sind:

```
<bean
  id="org.javacream.spring.beans.LifecycleBean"
  class="org.javacream.spring.beans.LifecycleBean"
  init-method="initBean"
  destroy-method="destroyBean"
>
</bean>
```

Der Lebenszyklus einer Spring Bean ist so definiert, dass die Initialisierungs-Methoden erst nach dem Konstruktoraufruf (was natürlich klar ist) und nach dem Aufruf aller setter-Methoden erfolgt.



In den Initialisierungsmethoden sind damit alle Dependencies garantiert gesetzt.

### 2.3.5 Setzen von Properties im Konstruktor

Besitzt die Spring Bean einen Konstruktor mit Parametern,

```
/**
 * @spring.constructor-arg value="Set By Constructor"
 * @spring.constructor-arg value="177"
 * @spring.constructor-arg list="Z,Y,X,W"
 * @param stringData
 * @param intData
 * @param list
 */
public DataWithConstructorBean(String stringData, int intData,
List list) {
    this.stringData = stringData;
    this.intData = intData;
    this.list = list;
}
```

so können diese ebenfalls in der Konfigurationsdatei eingetragen werden:

```
<bean
    id="org.javacream.spring.beans.DataWithConstructorBean"
class="org.javacream.spring.beans.DataWithConstructorBean"
    singleton="true"
>

    <constructor-arg>
        <value>Set By Constructor</value>
    </constructor-arg>
    <constructor-arg>
        <value>177</value>
    </constructor-arg>
    <constructor-arg>
        <list>
            <value>Z</value>
            <value>Y</value>
            <value>X</value>
            <value>W</value>
        </list>
    </constructor-arg>

</bean>
```

### 2.3.6 Registrierung spezieller Properties-Editoren

Die folgende Bean definiert Properties, für deren Typen Standardmäßig keine Property-Editoren registriert sind, nämlich Datumswerte und eine Referenz auf eine andere Bean:

```
public class CustomPropertyDataBean {  
  
    private Date date;  
    private DataBean dataBean;  
    //...  
}
```

Die folgende Bean-Konfiguration führt ohne weitere Konfigurationen zu Fehlern, da die für die Property „dataBean“, die ja vom Typ `DataBean` ist, kein `PropertyEditor` registriert ist. Ebenso ist für Datumswerte standardmäßig keine Umwandlung möglich, da eine spezielle Konfiguration für das zu verwendende Datumsformat erfolgen muss.

```
<bean  
    id="org.javacream.spring.beans.CustomPropertyDataBean"  
    class="org.javacream.spring.beans.CustomPropertyDataBean"  
>  
    <property name="dataBean">  
        <value>Fritz,15</value>  
    </property>  
    <property name="date">  
        <value>03.10.1964</value>  
    </property>  
</bean>
```

Sowohl das Datum als auch die Property vom Typ `DataBean` können von den Standardmäßig registrierten Property-Editoren nicht aus der übergebenen Zeichenkette aufgebaut werden.



Während es für das `java.util.Date` in der Klassenbibliothek von Spring bereits einen `PropertyEditor` gibt (`org.springframework.beans.propertyeditors.CustomDateEditor`), muss für die `DataBean` eine eigene Implementierung erstellt werden:

```
public class DataBeanPropertyEditor extends PropertyEditorSupport {

    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        DataBean dataBean = new DataBean();
        int commaPosition = text.indexOf(',');
        String stringData = text.substring(0, commaPosition);
        int intData = Integer.parseInt(text.substring(commaPosition + 1));
        dataBean.setIntData(intData);
        dataBean.setStringData(stringData);
        setValue(dataBean);
    }

}
```

Diese Editoren werden nun unter Verwendung des `org.springframework.beans.factory.config.CustomEditorConfigurer` beim `ApplicationContext` registriert<sup>4</sup>:

```
<bean id="customEditorConfigurer"
      class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="java.util.Date">
                <bean
                    class="org.springframework.beans.propertyeditors.CustomDateEditor">
                    <constructor-arg type="java.text.DateFormat">
                        <bean class="java.text.SimpleDateFormat">
                            <constructor-arg type="java.lang.String">
                                <value>dd.MM.yyyy</value>
                            </constructor-arg>
                        </bean>
                    </constructor-arg>
                    <constructor-arg type="boolean">
```

---

<sup>4</sup> Diese Registrierung könnte für bestimmte `Context`-Implementierungen auch über das Programm gesteuert erfolgen. Die angegebene Variante über die Konfigurationsdatei ist jedoch die empfohlene Variante.

```
        <value>true</value>
      </constructor-arg>
    </bean>
  </entry>
  <entry key="org.javacream.spring.beans.DataBean">
    <bean
      class="org.javacream.spring.beans.propertyeditor.DataBeanPropertyEditor">
    </bean>
  </entry>
</map>
</property>
</bean>
```

### 2.3.7 Verwendung von externen Properties-Dateien

Sollen innerhalb der Spring-Konfigurationsdatei an mehreren Stellen dieselben Werte verwendet werden, so können diese einfach in eine Properties-Datei ausgelagert werden. Innerhalb der Konfiguration wird dann über eine Expression der Form `${propertyName}` auf die Property referenziert<sup>5</sup>.

Das Auslesen der Properties-Datei übernimmt hierbei der `org.springframework.beans.factory.config.PropertyPlaceholderConfigurer`:

```
<bean id="externalProperties"
  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location"
    value="ExternalProperties.properties">
  </property>
</bean>

<bean id="externalPropertiesDataBean"
  class="org.javacream.spring.beans.DataBean">
  <property name="stringData" value="${externalProperty1}">
  </property>
</bean>
```

---

<sup>5</sup> Diese Syntax ist aus den diversen Expression Languages (z.B. JavaServer Pages) oder aber Apache Ant wohl bekannt.

Die Properties-Datei („ExternalProperties.properties“):

```
externalProperty1=testExternalProperty1  
externalProperty2=testExternalProperty2
```

Sollen die Properties einer Bean mit der ID „beanId“ aus einer Properties-Datei bezogen werden, so kann der `org.springframework.beans.factory.config.PropertyOverrideConfigurer` benutzt werden:

```
<bean id="PropertyOverride"  
  class="org.springframework.beans.factory.config.PropertyOverrideConfigurer">  
  <property name="location" value="PropertyOverride.properties">  
  </property>  
</bean>
```

Die Properties in der angegebenen Datei müssen die Form „beanId.propertyName“ haben:

```
propertyOverrideDataBean.stringData=overrideStringData  
propertyOverrideDataBean.intData=42
```

## 2.4 Referenzen auf andere Beans

### 2.4.1 Identifikation über die ID

Soll der Wert einer Property einer Bean auf eine Instanz einer anderen Spring Bean gesetzt werden, so geschieht dies durch das `<ref>`-Tag:

```
<beans>
  <bean>
    id="org.javacream.spring.beans.DataBean"
    class="org.javacream.spring.beans.DataBean"
    singleton="true"
  >

  <property name="list">

    <list>
      <value>A</value>
      <value>B</value>
      <value>C</value>
      <value>D</value>
    </list>
  </property>
  <property name="intData">

    <value>42</value>
  </property>
  <property name="stringData">

    <value>Hugo</value>
  </property>
</bean>

  <bean>
    id="org.javacream.spring.beans.DataBeanAggregateBean"
    class="org.javacream.spring.beans.DataBeanAggregateBean"
    singleton="true"
  >
    <property name="dataBean">
      <ref bean="org.javacream.spring.beans.DataBean"/>
    </property>
    <property name="name">
      <value>DataBeanAggregateBean</value>
    </property>
  </bean>
</beans>
```

### 2.4.2 Definition einer Inner Bean

Ist die zu setzende Referenz nur für eine Bean-Definition relevant, so kann auch eine Spring Bean ohne Referenz als „Inner Bean“ definiert werden:

```
<bean
id="org.javacream.spring.beans.DataBeanAggregateBeanWithInnerBe
an"

class="org.javacream.spring.beans.DataBeanAggregateBean"
singleton="true">

<property name="dataBean">
  <bean class="org.javacream.spring.beans.DataBean">
    <property name="list">

      <list>
        <value>1</value>
        <value>2</value>
        <value>3</value>
        <value>4</value>
      </list>
    </property>
    <property name="intData">

      <value>42</value>
    </property>
    <property name="stringData">

      <value>Hugo</value>
    </property>
  </bean>
</property>
<property name="name">
  <value>DataBeanAggregateBeanWithInnerBean</value>
</property>

</bean>
```

## 2.5 Abstrakte Beans und Vererbung an Kinder

Die Konfiguration von Beans kann durch einen Template-Mechanismus vereinfacht werden:

- Durch das `parent`-Attribut des `bean`-Tags werden alle Properties der Eltern-Spring Bean geerbt und können bei Bedarf überschrieben werden
  - Die Eltern-Spring Bean selber ist entweder eine vollständige Bean-Definition (die dann auch erzeugt wird) oder abstrakt:
    - Setzen des `abstract`-Attributs im `bean`-Tag auf „true“

```
<bean id="abstractKeyGenerator" abstract="true">
  <property name="prefix" value="${prefix}"></property>
  <property name="suffix" value="${suffix}"></property>
</bean>

<bean id="randomKeyGenerator"
class="org.javacream.keygeneration.business.RandomKeyGenerator"
  parent="abstractKeyGenerator">
</bean>

<bean id="counterKeyGenerator"
class="org.javacream.keygeneration.business.CounterKeyGenerato
r"
  parent="abstractKeyGenerator" dependency-check="all">
  <property name="counter" value="42"></property>
</bean>
```

## 2.6 Zugriff auf Ressourcen

### 2.6.1 Die Resource-Schnittstelle

Das Auslesen von Ressourcen wird komplett von Hilfsklassen des Spring-Frameworks durchgeführt:

- URL und damit auch der Dateizugriff
- Classpath

Am einfachsten ist es, zum Lesen den `ApplicationContext` zu verwenden:

```
public void testLoadResource() {  
    ApplicationContext context = SpringUtil.getApplicationContext("Spring Beans");  
    Resource resource = context.getResource("file:src/test/resources/TestResource.properties");  
    assertTrue(resource.exists());  
    resource = context.getResource("classpath:TestResource.properties");  
    assertTrue(resource.exists());  
}
```

Benötigt eine Spring Bean beispielsweise Konfigurationsinformationen, so bietet Spring einen `PropertyEditor`, der eine Zeichenkette in eine Ressource umwandelt. Dazu ist es jedoch notwendig, dass die Bean eine Property vom Typ

`org.springframework.core.io.Resource` definiert:

```
<bean  
    id="org.javacream.spring.beans.resource.ResourceBean"  
    class="org.javacream.spring.beans.resource.ResourceBean"  
>  
    <property name="resource">  
        <value>classpath:TestResource.properties</value>  
    </property>  
</bean>
```

Dieser Ansatz ist jedoch etwas problematisch, da nun die Spring Bean durch die `Resource`-Schnittstelle eine direkte Kopplung an das Spring-Framework hat.

Soll dies vermieden werden, so kann auch eine Property vom Typ `java.io.InputStream` verwendet werden, da Spring dafür einen eigenen

PropertyEditor  
(org.springframework.beans.propertyeditors.InputStreamEditor) zur Verfügung stellt.

Ein eigener Properties-Editor ist zwar Bestandteil der Spring-Bibliothek, ist aber einfach zu implementieren und damit didaktisch wertvoll<sup>6</sup>:

```
public class PropertiesPropertyEditor extends PropertyEditorSupport implements ApplicationContextAware{

    private ApplicationContext applicationContext;

    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
    }

    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        Resource resource = applicationContext.getResource(text);
        Properties props = new Properties();
        try {
            props.load(resource.getInputStream());
        } catch (IOException e) {
            e.printStackTrace();
            throw new IllegalArgumentException(e.getMessage());
        }
        setValue(props);
    }

}
```

---

<sup>6</sup> Die hier erstmalig benutzte Schnittstelle

org.springframework.context.ApplicationContextAware ist eine von Spring intern benutzte Schnittstelle, die vom ApplicationContext für Callback-Aufrufe benutzt wird. Die Spring-Bibliothek enthält einige Schnittstellen, die solche Callbacks definieren. Properties werden auch durch das spezielle <props> Tag definiert.



### 2.6.2 Internationalisierung mit der `MessageSource`-Schnittstelle

Neben dem Zugriff auf Ressourcen, die von der Konzeption her fachliche Konfigurationen enthalten stellt Spring auch eine Zugriffsschicht für `ResourceBundles` zur Internationalisierung zur Verfügung.

Der Zugriff erfolgt über eine konfigurierte Bean des Typs `org.springframework.context.support.ResourceBundleMessageSource`:

```
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
      <list>
        <value>messages</value>
        <value>exceptions</value>
      </list>
    </property>
</bean>
```

## 2.7 Factory Beans

Eine Factory Bean übernimmt für den Application Context die Aufgabe der konkreten Objekterzeugung. Die weitere Kontrolle über den Lebenszyklus und damit insbesondere über den Scope der Spring Bean bleibt davon unbeeinflusst.

Das Spring-Framework selber definiert eine ganze Reihe von Factory Beans, insbesondere:

- Erzeugung von Proxy-Objekten für die Netzwerkkommunikation.
- Umhüllen von Fachobjekten mit Aspekten

In der Klassenbibliothek ist ebenfalls eine abstrakte Klasse enthalten, die für eigene Factory Beans einfach als Superklasse benutzt werden kann:

```
package org.javacream.demo.spring.factory;

import java.util.List;

import org.springframework.beans.factory.config.AbstractFactoryBean;

public class DecoratorFactoryBean extends AbstractFactoryBean {

    private Object product;

    private List<Decoratable> decoratorList;

    @Override

    protected Object createInstance() throws Exception {

        if (!decoratorList.isEmpty()) {

            Object first = decoratorList.get(0);

            Decoratable next = null;

            for (Decoratable decoratable : decoratorList) {

                if (next != null) {

                    next.setDelegate(decoratable);

                }

            }

        }

    }

}
```

```
        next = decoratable;

    }

    next.setDelegate(product);

    return first;

} else {

    return product;

}

}

@Override

public Class<?> getObjectType() {

    return product.getClass();

}

public void setProduct(Object product) {

    this.product = product;

}

public void setDecoratorList(List<Decoratable> proxyList) {

    this.decoratorList = proxyList;

}

}
```

Diese Factory Bean kann eine Instanz einer Klasse (das `product`) mit einer Liste von `Decoratable`-Implementierungen dekorieren. `Decoratable` ist hierbei eine eigene Hilfs-Schnittstelle:

```
package org.javacream.demo.spring.factory;

public interface Decoratable{
    public void setDelegate(Object delegate);
}
```

Für den angesprochenen `StoreService` werden die beiden folgenden Decorators benutzt:

Ein simpler Validator:

```
package org.javacream.demo.spring.factory;

import org.javacream.store.StoreService;

public class StoreServiceValidator implements StoreService, Decoratable {

    private StoreService storeService;

    public int getStock(String category, String id) {
        if (category != null && id != null && !category.contains(" ")) {
            return storeService.getStock(category, id);
        } else {
            return 0;
        }
    }

    public void setDelegate(Object delegate) {
        storeService = (StoreService) delegate;
    }
}
```

und ein Cache:

```
package org.javacream.demo.spring.factory;

import java.util.HashMap;

import org.javacream.store.StoreService;

public class StoreServiceCache implements StoreService, Decor-
atable {

    private StoreService storeService;
    private HashMap<StoreServiceKey, Integer> cache = new
HashMap<StoreServiceKey, Integer>();

    public int getStock(String category, String id) {
        StoreServiceKey key = new StoreServiceKey(category, id);
        Integer stock = cache.get(key);
        if (stock == null) {
            System.out.println("Cache miss");
            stock = storeService.getStock(category, id);
            cache.put(key, stock);
        }else{
            System.out.println("Cache hit");
        }
        return stock;
    }

    public void setDelegate(Object delegate) {
        storeService = (StoreService) delegate;
    }

    private class StoreServiceKey {
        private String category, id;

        public StoreServiceKey(String category, String id) {
            this.category = category;
            this.id = id;
        }

        @Override
        public int hashCode() {
            final int prime = 31;
            int result = 1;
            result = prime * result + getOuterType().hashCode();
            result = prime * result
                + ((category == null) ? 0 : category.hashCode());
        }
    }
}
```

```
        result = prime * result + ((id == null) ? 0 :
id.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        StoreServiceKey other = (StoreServiceKey) obj;
        if (!getOuterType().equals(other.getOuterType()))
            return false;
        if (category == null) {
            if (other.category != null)
                return false;
        } else if (!category.equals(other.category))
            return false;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }

    private StoreServiceCache getOuterType() {
        return StoreServiceCache.this;
    }

}
```

Die folgende Spring-Konfiguration zeigt die Verwendung der Factory Bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="decoratorFactory"
class="org.javacream.demo.spring.factory.DecoratorFactoryBean"
        name="storeServiceFromFactory">
        <property name="product">
            <bean
class="org.javacream.store.business.StoreServiceImpl" />
        </property>
        <property name="decoratorList">
            <list>
                <bean
class="org.javacream.demo.spring.factory.StoreServiceValidator"
></bean>
                <bean
class="org.javacream.demo.spring.factory.StoreServiceCache"></b
ean>
            </list>
        </property>
    </bean>
</beans>
```

Diese Formulierung ist eleganter und nachvollziehbarer als eine (ebenfalls mögliche) Definition der Decorators als eigene Beans und referenzieren der `delegate-Property`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <bean id="storeServiceValidator"
class="org.javacream.demo.spring.factory.StoreServiceValidator"
        name="storeServiceWithoutFactory">
        <property name="delegate">
            <bean
class="org.javacream.demo.spring.factory.StoreServiceCache">
                <property name="delegate">
                    <bean
class="org.javacream.store.business.StoreServiceImpl" />
                </property>
            </bean>
        </property>
    </bean>
</beans>
```

„Dekoration“ ist übrigens nicht zu verwechseln mit dem Hinzufügen von Aspekten: Aspekte sind in der Regel recht allgemein gehalten und „wissen nicht“, welches Business-Interface sie anreichern. Ein Decorator kennt die genaue Signatur des umhüllte Objekts und kann somit deutlich spezieller implementiert werden.



## 2.8 Konfiguration mit Annotations

Neben der Definition von Spring Beans mit Hilfe der XML-Dateien enthält das Spring API ab der Version 2.5 auch einen Satz von Annotations:

- `org.springframework.stereotype.Component` ist die „Basis“-Annotation. Jede Klasse mit dieser Annotation wird als Spring Bean interpretiert. Als Parameter benötigt diese Annotation den Namen der Bean.
- `Service`, `Repository` und `Controller` sind ebenfalls gültige Annotations. Alle sind selber ebenfalls `Components`, die aber verschiedene Ausprägungen signalisieren, die von weiteren Spring-Komponenten berücksichtigt werden können. So könnten beispielsweise alle `Services` in einem Server registriert werden oder aber alle `Repositories` eine Referenz auf eine `DataSource` injiziert bekommen.
- `org.springframework.context.annotation.Scope` definiert den Scope der Bean.
- `org.springframework.beans.factory.annotation.Autowired` definiert das Autowire-Verhalten.
- `org.springframework.beans.factory.annotation.Required` verlangt das Setzen einer Property.

Daneben werden die Annotations der JEE benutzt, die nicht direkt den Enterprise Beans zugeordnet sind. Dies sind aus dem Paket `javax.annotation`:

- `@PostConstruct` für eine init-Methode
- `@PreDestroy` für eine destroy-Methode
- `@Resource` für eine Referenz auf eine andere Bean.

Das folgende Beispiel zeigt den annotierten MapBooksService:

```
package org.javacream.books.warehouse.business;

import java.util.ArrayList;

import java.util.Collection;

import java.util.HashMap;

import java.util.Map;

import javax.annotation.PostConstruct;

import javax.annotation.PreDestroy;

import javax.annotation.Resource;

import org.javacream.books.warehouse.BookException;

import org.javacream.books.warehouse.BooksService;

import org.javacream.books.warehouse.value.BookValue;

import org.javacream.keygeneration.KeyGenerator;

import org.javacream.store.StoreService;

import org.springframework.context.annotation.Scope;

import org.springframework.stereotype.Service;

@Service("booksService")

@Scope("prototype")

public class MapBooksService implements BooksService {

    private KeyGenerator keyGenerator;

    private Map<String, BookValue> books;

    private StoreService storeService;

    {

        books = new HashMap<String, BookValue>();
    }
}
```

```
}

Map<String, BookValue> getBooksMap() {

    return books;

}

@Resource(name="storeServiceImpl")

public void setStoreService(StoreService storeService) {

    this.storeService = storeService;

}

@PostConstruct

public void initTheBean(){

    System.out.println("Initializing...");

}

public String newBook(String title) throws BookException {

    String isbn = keyGenerator.next();

    BookValue detail = new BookValue();

    detail.setIsbn(isbn);

    detail.setTitle(title);

    books.put(isbn, detail);

    return isbn;

}

public BookValue findBookByIsbn(String isbn) throws BookException {

    BookValue result = (BookValue) books.get(isbn);

    if (result == null) {
```

```
        throw new BookException(BookExceptionType.NOT_FOUND, isbn);
    }

    result.setAvailable(storeService.getStock("books", isbn) > 0);

    return result;
}

public BookValue updateBook(BookValue bookDetailValue) throws BookException {
    // Potenzielle Validierung:

    if (bookDetailValue.getPrice() <= 0) {
        throw new BookException(BookExceptionType.CONSTRAINT, "price <= 0");
    }

    BookValue value = findBookByIsbn(bookDetailValue.getIsbn());
    value.setTitle(bookDetailValue.getTitle());
    value.setPrice(bookDetailValue.getPrice());
    value.setAvailable(bookDetailValue.isAvailable());

    return value;
}

public void deleteBookByIsbn(String isbn) throws BookException {
    Object result = books.remove(isbn);

    if (result == null) {
        throw new BookException(
            BookExceptionType.NOT_DELETED, isbn);
    }
}
```

```
    }

    }

    public KeyGenerator getKeyGenerator() {

        return keyGenerator;

    }

    @Resource (name="randomKeyGenerator")

    public void setKeyGenerator(KeyGenerator keyGenerator) {

        this.keyGenerator = keyGenerator;

    }

    public StoreService getStoreService() {

        return storeService;

    }

    public Collection<BookValue> findAllBooks() {

        return new ArrayList<BookValue>(books.values());

    }

    @PreDestroy

    public void destroyTheBean(){

        System.out.println("Destroying...");

    }

}
```

Um die Annotations-basierten Beans vom `ApplicationContext` verwaltet zu bekommen, ist folgende Konfiguration notwendig:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
                           beans-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-
                           2.5.xsd">

    <context:component-scan base-package="org.javacream"/>

</beans>
```

Damit der `ApplicationContext` nicht alle Klassen laden und nach Annotations durchforsten muss, kann ein `base-package` angegeben werden. Weiterhin können inclusion- und exclusion-Filter angegeben werden.

## 3 Datenzugriffe mit Spring

### 3.1 Inversion of Control mit Method Template

Bei den Datenzugriffen verwendet Spring den „Inversion of Control“-Ansatz, um Dienste wie Transaktionssteuerung oder eine vereinfachte, zentrale Ausnahmebehandlung einzuführen.

Die Grundidee dieses Ansatzes ist eine Template-Klasse, die eine Aktion ausführen kann, und eine Callback-Klasse, deren Methode innerhalb des Templates ausgeführt wird. Dies entspricht im Wesentlichen dem Design Pattern „Method Template“, nur dass hier die abstrakte Methode in das Callback-Interface ausgelagert wird.

### 3.2 Direkter JDBC-Zugriff

Für den direkten Zugriff mit JDBC gibt es das `JdbcTemplate`, das in einer Spring-Konfiguration eingetragen werden kann. Als zentrale Property wird hierfür eine `DataSource` benötigt, die wiederum als Spring Bean definiert werden kann. Das Spring-Framework enthält dafür einen ganzen Satz von Implementierungen:

- Eine eigene Implementierung, die direkt mit dem `DriverManager` zusammenarbeitet.
- Einbindung des Apache DB-Projekts.
- Zugriff auf eine JNDI-`DataSource`.

Die folgende Konfiguration definiert ein `JdbcTemplate` für die H2-Datenbank:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans default-autowire="no" default-lazy-init="false"
    default-dependency-check="none">

    <bean id="HsqlDataSource"

        class="org.springframework.jdbc.datasource.DriverManager
DataSource">
```

```
<property name="driverClassName" value="org.hsqldb.jdbcDriver"></property>
<property name="password" value=""></property>
<property name="username" value="SA"></property>
<property name="url" value="jdbc:hsqldb:hsqldb://raum10srv01/unilogdb"></property>
<!--
<property name="url" value="jdbc:hsqldb:/data"></property>
-->
<!--
<property name="url" value="jdbc:hsqldb:."></property>
-->
</bean>

<bean id="JdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource"
ref="HsqlDataSource"></property>
</bean>
</beans>
```

Hier wird als `DataSource` die `DriverManagerDataSource` benutzt, der die `Connections` direkt vom `java.sql.DriverManager` holt.

In der Praxis wird hierfür wahrscheinlich ein `Connection Pool` benutzt, beispielsweise innerhalb eines Applikationsservers. Auch dafür stellt die Spring-Klassenbibliothek geeignete Implementierungen zur Verfügung.

Die eigentliche Datenzugriffsklasse ist nun sehr einfach zu schreiben:

- Das `JdbcTemplate` wird als `Property` definiert und über `Dependency Injection` gesetzt



- Implementierungen geeigneter Callback-Schnittstellen, wahrscheinlich definiert als Innere oder sogar Anonyme Klassen, enthalten die eigentliche Datenzugriffslogik:

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;
import java.util.ArrayList;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.StatementCallback;

public class JdbcDemo{

    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void log(final String logMessage) {
        return (String) jdbcTemplate.execute(new StatementCall-
back() {
            public Object doInStatement(Statement statement)
                throws SQLException, DataAccessException {

                int rows = statement.executeUpdate("insert into
LOG_TABLE values('" + logMessage + "')");
                return null;
            }

        });
    }

    public List getLogMessages() {
        return (Integer) jdbcTemplate.execute(new StatementCa-
llback() {
            public Object doInStatement(Statement statement)
                throws SQLException, DataAccessException {
                ResultSet rs = statement
                    .executeQuery("select * from LOG_TABLE"s;
                ArrayList result = new ArrayList();

                while (rs.next()) {
                    result.add(rs.getObject(1));
                }
                return result;
            }
        });
    }
}
```

Als Callback-Schnittstellen existieren:

- Das oben verwendete einfache `StatementCallback`. Bei der Ausführung genügt hier als Parameter eine Instanz einer geeigneten Implementierung.
- Das `PreparedStatementCallback` benötigt als Parameter das für die Erzeugung notwendige Template. Dieses wird als String-Parameter der `execute`-Methode des `JdbcTemplate` übergeben.
- Das `CallableStatementCallback` benötigt als Parameter ebenfalls eine Zeichenkette, die die aufzurufende Stored Procedure definiert.

### 3.3 Transaktions-Management

#### 3.3.1 Übersicht

Spring stellt für die Abstraktion der verwendeten Transaktions-Management-Technologie ein eigenes API zur Verfügung:

- `org.springframework.transaction` mit Exception-Klassen, die den typischen Transaktionsfehlern entsprechen. Wie in Spring üblich sind dies `RuntimeExceptions`
- Unterstützung deklarativer Transaktionssteuerung unter Verwendung des Spring AOP-Frameworks (`org.springframework.transaction.interceptor`) bzw. mit Java 5 Annotations (`org.springframework.transaction.annotation`)
- Verwendung einer JTA-konformen Transaktions-Managers, der von einem externen System, wahrscheinlich einem Applikationsserver<sup>7</sup>, zur Verfügung gestellt wird (`org.springframework.transaction.support` und `org.springframework.transaction.jta`).

Für die Verwendung des Transaktions-APIs gibt es mehrere unterschiedliche Ansätze:

- Deklarative Transaktionssteuerung
- Programmatische Transaktionssteuerung
  - Verwendung des JTA-APIs, wobei der Programmierer die volle Verantwortung für das korrekte commit/rollback, insbesondere beim Exception-Handling, ist
  - Verwendung des `TransactionTemplates`. Hier wird durch eine Callback-Mechanismus die Verantwortung an die Superklasse delegiert<sup>8</sup>.
- Verwendung eines „höherwertigen“ Frameworks für den Datenzugriff, das die Transaktionssteuerung intern übernimmt.

---

<sup>7</sup> Es gibt aber auch eine Reihe unabhängiger JTA-Implementierungen

<sup>8</sup> Dies wird durch das Design Pattern „Method Template“ erreicht

Für alle Möglichkeiten ist es notwendig, eine Implementierung eines `TransactionManagers` zur Verfügung zu stellen. Dies ist selbstverständlich wieder eine Spring Bean, die in verschiedenen Ausprägungen (eigene Spring-Implementierung, Anbindung vorhandener Produkte, insbesondere durch JNDI-Lookup) existieren:

```
<bean
class="org.springframework.jdbc.datasource.DataSourceTransaction
Manager">
    <property name="dataSource" ref="dataSource"></property>
</bean>
```

### 3.3.2 TransactionCallback

Beispiel: Ausführung einer Transaktion innerhalb eines Callback:

```
public interface TransactionCallback {
    Object doInTransaction(TransactionStatus status);
}

public class TransactionTemplate extends DefaultTransactionDe-
finition implements InitializingBean {
    //...
    public Object execute(TransactionCallback action) throws
    TransactionException {
        TransactionStatus status =
        this.transactionManager.getTransaction(this);
        Object result = null;
        try {
            result = action.doInTransaction(status);
        }
        catch (RuntimeException ex) {
            // Transactional code threw application exception -> roll-
            back
            rollbackOnException(status, ex);
            throw ex;
        }
        catch (Error err) {
            // Transactional code threw error -> rollback
            rollbackOnException(status, err);
            throw err;
        }
        this.transactionManager.commit(status);
        return result;
    }
    //...
}
```

Der Sinn dieser Konstruktion ist evident: Jeder Aufruf der Callback-Methode wird innerhalb eines Transaktionskontextes ausgeführt. Normale Methodenterminierung und die Fehlerbehandlung führen zu korrektem commit/rollback-Verhalten.

### 3.3.3 TransactionProxyFactoryBean

Das folgende Beispiel zeigt die deklarative Transaktionssteuerung unter Verwendung der `TransactionProxyFactoryBean`:

```
<bean id="transactionalBooksService"
class="org.springframework.transaction.interceptor.TransactionP
roxyFactoryBean"
    name="org.javacream.books.warehouse.BooksService">
    <property name="proxyInterfaces">
        <valu-
e>org.javacream.books.warehouse.BooksService</value>
    </property>
    <property name="target" ref="booksService"></property>
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
    <property name="transactionManager">
        <bean
class="org.springframework.jdbc.datasource.DataSourceTransactio
nManager">
            <property name="dataSource"
ref="dataSource"></property>
        </bean>
    </property>
</bean>
```

Hier wird der Zugriff auf den `BooksService` transaktionell gesteuert. Der `DataSourceTransactionManager` kontrolliert alle von der konfigurierten `DataSource` erzeugten `Connections`. Alternativ könnte hier selbstverständlich auch eine Anbindung an den Transaktionsmanager eines Applikationsservers erfolgen. Die `TransactionProxyFactoryBean` erzeugt den entsprechenden Interceptor, wobei die Transaktionsattribute über eine Property-Konfiguration erfolgen. Die oben angegebenen Einstellungen bedeuten, dass jede Methode („\*“) einen vorhandenen Transaktionskontext propagieren soll („PROPAGATION\_REQUIRED“). Ist kein Transaktionskontext vorhanden wird ein neuer eröffnet. Die mögliche Attribute sind innerhalb der

### Schnittstelle

`org.springframework.transaction.TransactionDefinition` definiert<sup>9</sup>:

- `PROPAGATION_REQUIRES`
- `PROPAGATION_REQUIRES_NEW`
- `PROPAGATION_MANDATORY`
- `PROPAGATION_SUPPORTS`
- `PROPAGATION_NOT_SUPPORTED`
- `PROPAGATION_NEVER`
- `PROPAGATION_NESTED`

Zusätzlich kann, durch ein Komma separiert, `readOnly` angegeben werden.

### 3.3.4 tx-Namensraum

Die Verwendung dieser Factory Bean ist ab Spring 2.0 in dieser Form nicht mehr notwendig. Die Transaktionskonfiguration erfolgt durch die Verwendung des XML-Schemas

`xmlns:tx="http://www.springframework.org/schema/tx."`

```
<aop:config>
  <aop:pointcut id="booksServiceMethods"
    expression="execution(* *..IBatisBooksService.*(..))"
  />
  <aop:advisor advice-ref="txAdvice" pointcut-
    ref="booksServiceMethods" />
</aop:config>
<tx:advice id="txAdvice" transaction-
  manager="txManager">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED" />
  </tx:attributes>
</tx:advice>
```

### 3.3.5 Annotations-basierte Transaktionssteuerung

Die folgende Klasse deklariert den Schlüsselgenerator und dessen Transaktionsverhalten mit Annotations:

<sup>9</sup> Die Namen und die Interpretation dieser Werte entsprechen den allgemein verwendeten Transaktionsattributen, wie sie beispielsweise auch innerhalb der JEE benutzt werden.

```
package org.javacream.keygeneration.business;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.StatementCallback;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;
@Repository("keyGenerator")
public class AnnotatedJdbcKeyGenerator{
    private JdbcTemplate jdbcTemplate;
    @Resource(name="jdbcTemplate")
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    @PostConstruct
    public void init() {
        jdbcTemplate.execute(new StatementCallback() {
            public Object doInStatement(Statement statement)
                throws SQLException, DataAccessException {
                statement.execute("drop table keys if exists");
                statement.execute("create table keys (col_key integer)");
                statement.execute("insert into keys values (42)");
                return null;
            }
        });
    }
    @Transactional(propagation=Propagation.REQUIRED)
    public int next(boolean rollback) {
        int key = (Integer) jdbcTemplate.execute(new StatementCallback() {
            public Object doInStatement(Statement statement)
                throws SQLException, DataAccessException {
```

```
        ResultSet rs = statement.executeQuery("select * from
keys");
        rs.next();
        int key = rs.getInt(1);
        int newKey = key + 1;
        statement.execute("update keys set col_key=" +
newKey);
        return newKey;
    }
});
if (rollback){
    throw new KeyGeneratorException("rollback!", key);
}else{
    return key;
}
}
```

Die Konfiguration der Spring Bean ebenfalls mit Annotations ist nicht zwingend, aber konsequent.



Für die Aktivierung der Transaktionen wird das Tag `<tx:annotation-driven/>` benötigt, ebenso natürlich eine `DataSource` und der `TransactionManager`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd"
       ">
    <context:component-scan base-package="org.javacream" />
    <tx:annotation-driven />
    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource"
            ref="dataSource"></property>
    </bean>
    <bean id="jdbcTemplate"
          class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource"
            ref="dataSource"></property>
    </bean>
    <bean id="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="org.hsqldb.jdbcDriver">
        </property>
        <property name="url" value="jdbc:hsqldb:."></property>
        <property name="username" value="SA"></property>
        <property name="password" value=""></property>
    </bean>
</beans>
```

### 3.4 Hibernate-Unterstützung

Der Open Source O/R-Mapper Hibernate und Spring passen von den Grundkonzepten her hervorragend zusammen:

- Dependency Injection -Ansatz zur Konfiguration seiner Komponenten durch austauschbare Dienst-Implementierungen (`DataSourcees`, `TransactionManager`)
- Konfiguration über Properties- bzw. XML-Dateien
- Annotations sowie Unterstützung Aspektorientierter Ansätze<sup>10</sup>

Hibernate3 bietet bereits eine saubere Unterstützung für Transaktionen: Die Methode `SessionFactory.getCurrentSession()` verwendet den für die Factory gesetzten Transaktions-Manager, um die dem aktuellen Kontext (lokaler Methodenaufruf, `HttpRequest`, EJB-Aufruf) assoziierte Transaktion zuordnen zu können.

Dazu muss wird der Transaktions-Manager in der Spring-Konfiguration definiert und der `SessionFactory` zugeordnet. Für eine Singlethreaded-Anwendung genügt dafür der `org.springframework.orm.hibernate3.HibernateTransactionManager`:

```
<bean id="TxManager"
class="org.springframework.orm.hibernate3.HibernateTransactionM
anager">
<property name="sessionFactory" ref="HibernateSessionFactory"/>
</bean>
```

Alternativ hierzu kann eine der Spring-Implementierungen in `org.springframework.transaction.jta` für die `SessionFactory` gesetzt werden.

## 4 Das Spring Web Framework

Das Spring Web Framework ist nach dem Model-View-Controller Design Pattern aufgebaut. Die Rolle des Datenmodells übernehmen hierbei SpringBeans, die von Controller-Implementierungen aufgenommen werden. Teil des Controllers ist das bereits bekannte DispatcherServlet, das dem einkommenden Request das auszuführende Kommando entnimmt und dann den Aufruf an den dazu konfigurierten Controller delegiert. Die Rolle der View-Komponenten übernehmen häufig JavaServer Pages, Spring unterstützt aber auch andere Template-Engines (z.B. Apache Velocity) oder Dateiformate (Tabellenkalkulationen, PDF).

Die Erstellung von Web Anwendungen mit Hilfe von Spring ist durch ein sauber designtes Framework realisiert worden. Das Basis-Design entspricht dem MVC-Design-Pattern:

- Die Model-Komponenten sind normale SpringBeans
- Die Controller sind durch mehrere Komponenten abgebildet:
  - Ein Dispatcher liest aus dem Http-Request ein Aktionskommando aus und entscheidet, welcher
  - Handler aufgerufen wird. Dieser delegiert schließlich noch an den eigentlichen
  - Controller, der die Aufrufe an das Modell weiter delegiert.
- Die View-Komponenten sind zweigeteilt:
  - HTML-Formulare, die vom Browser auf dem Client dargestellt werden. Alternativ können auch andere Formate (Tabellenkalkulation, PDF) erzeugt werden.
  - Auf dem Server übernimmt eine Template Engine die Erzeugung der Dokumenten-Formate, die der Client angefordert hat.

Das Spring-Framework steht in seiner Komplexität ähnlichen Ansätzen wie Apache Spring oder WebWorks in nichts nach. Beispiele für vollständige Web Applikationen sprengen den Rahmen der Präsentation, hier sei auf die der Spring-Distribution beiliegenden Beispiele (petclinic, jpetstore) verwiesen.

## 4.1 Das Dispatcher-Servlet

Dieses Servlet, eine Instanz der Klasse

`org.springframework.web.servlet.DispatcherServlet`, wurde bereits bei den entfernten Methodenaufrufen über http eingeführt.

Dieses zentrale Servlet, das häufig auch nach einem Design Pattern für verteilte Anwendungen „Front Controller“ benannt wird, wird in der `web.xml` konfiguriert und als Web-Archiv in einem Servlet-Engine installiert.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    <servlet>
        <servlet-name>FrontController</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/web-spring-beans.xml</param-
value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>FrontController</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>FrontController</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>

    <taglib>
        <taglib-uri>/spring</taglib-uri>
        <taglib-location>/WEB-INF/spring.tld</taglib-location>
    </taglib>
</web-app>
```

Das Servlet liest noch zusätzlich bei der Initialisierung eine Konfigurationsdatei mit beliebigen SpringBean-Definitionen. Wie üblich wird dadurch ein `ApplicationContext` aufgebaut. Dieser ist eine Erweiterung, die den Zugriff auf den `ServletContext` ermöglicht:

```
org.springframework.web.context.WebApplicationContext
```

Der Zugriff auf eine Implementierung dieser Schnittstelle wird im `HttpRequest` abgelegt. Als Schlüssel wird `DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE` verwendet.

## 4.2 Spezielle Beans für Web Anwendungen

Eine Spring-Web Anwendung definiert Bean-Typen, die für die speziellen Anforderungen konzipiert sind:

Begriff	Erläuterung
<code>controller(s)</code>	Der eigentliche Controller ruft die Methoden des Models auf.
<code>handler mapping(s)</code>	Ein Handler ist ein Pre- oder Postprozessor, der an Hand bestimmter Request-Kriterien ausgeführt werden. Handler sind auch als „Intercepting Filter“ bekannt.
<code>view resolver</code>	Das Ergebnis eines Model-Aufrufs wird von einem bestimmten Template für den Client aufbereitet. Der View Resolver (auch „Application Controller“) entscheidet durch einen Alias-Namen, welche View verwendet werden soll.
<code>locale resolver</code>	Wird für internationalisierte Darstellungen verwendet.
<code>theme resolver</code>	Personalisierte Layouts für die angezeigten Views.
<code>multipart resolver</code>	File-Upload aus HTML-Seiten.
<code>handlerexception resolver</code>	Ausnahmebehandlung, beispielsweise die Darstellung einer Fehler-View.

Die Resolver werden wie der `WebApplicationContext` als Parameter des Requests gesetzt. Als Schlüssel werden wieder statische Attribute des `DispatcherServlets` benutzt.

Die Initialisierungs-Parameter des `DispatcherServlets` sind:

Parameter	Erläuterung
<code>contextClass</code>	Implementierung der <code>WebApplication-Context-Schnittstelle</code> , die vom <code>DispatcherServlet</code> benutzt wird. Standard: <code>XmlWebApplicationContext</code> .
<code>contextConfigLocation</code>	Name der Konfigurationsdateien (Kommagetrennte Liste), die gelesen werden sollen
<code>namespace</code>	Der Namensraum der Web-Applikation. Standard: <code>[server-name]-servlet</code> .

Eine zentrale Klasse des MVC-Webframeworks ist `ModelAndView`: Im MVC-Design übernimmt der Controller die Ansteuerung des Datenmodells und bekommt in der Regel ein zugehöriges Ergebnis als Datensnapshot (ein Value Object) oder eine Live-Entität. Für eine Web-Anwendung muss dieses Ergebnis aber noch in eine vom Browser darstellbare Form, eine Serverseitig erstellte View, umgewandelt werden. Und genau das macht `ModelAndView`: Diese Klasse hält eine beliebige Menge von Objekten, zugreifbar über einen eindeutigen Schlüssel sowie den Namen einer View.

Was Spring jetzt noch wissen muss ist, wie die Server-seitige View erstellt werden soll. Dafür sind verschiedenste Technologien möglich und etabliert:

- JavaServer Pages sowie Tag Libraries, insbesondere mit dem Standard der JavaServer Pages Standard Tag Libraries (JSTL)
- Expression Languages wie das beliebte Apache Velocity ([apache.org](http://apache.org)), dem jedoch in letzter Zeit von Freemarker ([freemarker.org](http://freemarker.org)) der Rang abgelaufen wird. Beide Projekte sind Open Source.
- XSL-Transformationen sind insbesondere dann beliebt, wenn das Model bereits Daten im XML-Format liefert.

Welche Technologie benutzt wird hängt natürlich von der Spring-Konfiguration ab, in der ein View-Resolver eingetragen werden muss:

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="viewClass"><value>org.springframework.web.servlet.view.JstlView</value>
</property>
<property name="prefix"><value>/WEB-INF/pages/books/</value></property>
<property name="suffix"><value>.jsp</value></property>
</bean>
```

Hier werden JSP-Templates aufgelöst, die sich im Verzeichnis `WEB-INF/pages/books/` befinden und die Standard-Endung `".jsp"` haben.



## 4.3 Controller

### 4.3.1 Ein einfacher Controller

Ein Controller bekommt als Parameter einen `HttpServletRequest` und liefert als Ergebnis eine Instanz der Klasse `ModelAndView`, die dann die Darstellung übernimmt. Alternativ kann der Controller auch direkt in den `HttpServletResponse` schreiben.

```
package org.springframework.web.servlet.mvc;

public interface Controller {

    /**
     * Process the request and return a ModelAndView Object which
     * the DispatcherServlet
     * will render.
     */
    ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception;
}
```

Spring selber liefert bereits eine Reihe von (abstrakten) Basisimplementierungen, die die Implementierung eigener Controller-Klassen stark erleichtern:

Der `AbstractController` liefert eine `SpringBean`-konforme Basis-Klasse, die einige Details des HTTP-Protokolls konfigurierbar macht:

- Soll eine `HttpSession` aufgebaut werden?
- Welche Methoden (HTTP-GET, HTTP-POST,...) werden unterstützt?
- Konfiguration von Caching-Parametern

### Eine einfache Implementierung:

```
public class DemoController extends AbstractController {

    private static final String DEMO_PAGE;

    static{
        DEMO_PAGE = "<HTML>" +
            "<TITLE>Demo Page</TITLE>" +
            "<BODY>A demo page created by a DemoController</BODY>"
+
            "<HTML>";
    }

    @Override
    protected ModelAndView handleRequestInternal(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        response.getWriter().print(DEMO_PAGE);
        return null;
    }
}
```

benötigt nun nur noch eine Definition als SpringBean und ein Mapping zwischen der aufrufenden URL und der Bean-ID<sup>11</sup>.

---

<sup>11</sup> Es sind auch andere Mapping-Strategien denkbar und möglich.

Beides ist Bestandteil der Spring-Konfiguration:

```
<beans>

    <bean id="DemoController"
        class="org.javacream.spring.beans.web.DemoController">
        <property name="cacheSeconds" value="120" />
    </bean>

    <bean id="handlerMapping"

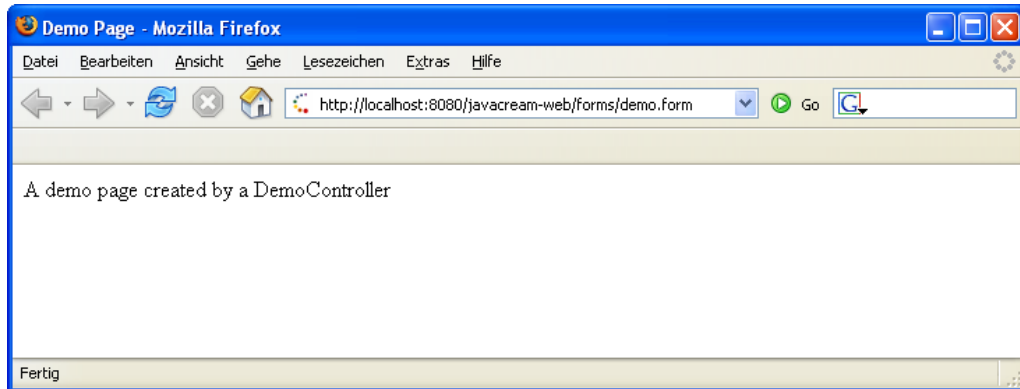
        class="org.springframework.web.servlet.handler.SimpleUrl
HandlerMapping">
        <property name="mappings">
            <props>
                <prop key="demo.form">
                    DemoController
                </prop>
            </props>
        </property>
    </bean>

</beans>
```

Das Ganze wird über ein Ant-Skript in ein Web-Archiv verpackt und kann dann im Server installiert werden:

```
<target name="web-war" description="javacream-web.war">
    <jar destfile="${target.dir}/javacream-web.war">
        <zipfileset dir="${web-web-inf.dir}" prefix="WEB-INF">
            <include name="web.xml" />
        </zipfileset>
        <zipfileset dir="${spring-beans.dir}" prefix="WEB-INF">
            <include name="web-spring-beans.xml" />
        </zipfileset>
        <zipfileset dir="${classes.dir}" prefix="WEB-
INF/classes"/>
        <zipfileset dir="${spring.dir}" prefix="WEB-INF/lib">
            <include name="spring.jar" />
        </zipfileset>
        <zipfileset dir="${classes.dir}">
            <include name="*.properties" />
        </zipfileset>
    </jar>
</target>
```

```
</jar>
</target>
```



### 4.3.2 Weitere Controller-Implementierungen

Der `ParameterizableViewController` erlaubt in seiner Konfiguration die Angabe des View-Namens, der im `WebApplicationContext` gebunden wird.

Der `UrlFilenameViewController` wandelt einen der URL angehängten Dateinamen in den Namen einer View um.

Eine weitere sehr praktische Implementierung ergibt sich aus folgender Überlegung: Wie viele Kommandos müssen die Controller in einer realen Anwendung verstehen? Je nach Komplexität kommt hier sehr schnell eine unangenehm hohe Zahl heraus, die entweder eine große Zahl von Controller-Klassen oder aber Dispatching-Logik Kommando – Model-Methode erfordert. Genau diese Logik steht nun aber bereits mit dem `MultiActionController` bereit.

Dieser benutzt den `HttpRequest` dazu, über einen `MethodNameResolver` eine Methode der Signatur

```
ModelAndView <name>(HttpServletRequest, HttpServletResponse);
```

oder, für die Fehlerbehandlung einen `ExceptionHandler`

```
ModelAndView <name>(HttpServletRequest, HttpServletResponse,
<ExceptionKlasse>);
```

aufzurufen.

Property	Beschreibung
delegate	Die <code>delegate</code> -Property wird benutzt, wenn dieser Controller direkt (ohne Ableitung) verwendet werden soll. Er braucht dann diese Property, um an die „eigentliche“ Controller-Implementierung delegieren zu können
methodNameResolver	<p>Zur Verfügung stehen:</p> <ul style="list-style-type: none"><li>• <code>InternalPathMethodNameResolver</code>, der einen Pfad der URL als Methode interpretiert und</li><li>• <code>ParameterMethodNameResolver</code> der einen konfigurierbaren URL-Parameter als Methodenaufruf interpretiert,</li><li>• <code>PropertiesMethodNameResolver</code>: Ein Mapping zwischen URLs und Methoden in Form eines <code>Properties</code>-Objekts</li></ul>

Beispiel:

Ein einfaches Datenmodell:

```
public class SimpleDemoModel {  
  
    private static List<SimpleDataValue> simpleDataValues;  
    static {  
        simpleDataValues = new ArrayList<SimpleDataValue>();  
        for (int i = 0; i < 10; i++) {  
            SimpleDataValue value = new SimpleDataValue();  
            value.setName("SimpleDataValue:name" + i);  
            value.setDescription("SimpleDataValue:description" +  
i);  
            ArrayList<SimpleDataDetailValue> detailValues = new Ar-  
rrayList<SimpleDataDetailValue>(  
                5);  
            for (int j = 0; j < 5; j++) {  
                SimpleDataDetailValue detailValue = new SimpleDataDe-  
tailValue();  
                detailValue.setDetail1("SimpleDataDetailValue:detail1"  
+ j);  
                detailValue.setDetail2("SimpleDataDetailValue:detail2"  
+ j);  
            }  
            detailValues.add(detailValue);  
        }  
        simpleDataValues.addAll(detailValues);  
    }  
}
```

```

        detailValue.setDetail3("SimpleDataDetailValue:detail3"
+ j);
        detailValue.setDetail4("SimpleDataDetailValue:detail4"
+ j);
    }
    value.setSimpleDataDetailValues(detailValues);
    simpleDataValues.add(value);
    }
}

    public SimpleDataValue getSimpleDataValue(int index) {
        return simpleDataValues.get(index);
    }

    public int insertSimpleDataValue(SimpleDataValue simple-
DataValue) {
        simpleDataValues.add(simpleDataValue);
        return simpleDataValues.size() - 1;
    }

    public void removeSimpleDataValue(SimpleDataValue simp-
leDataValue) {
        simpleDataValues.remove(simpleDataValue);
    }

    public List<SimpleDataValue> getSimpleDataValueList() {
        return Collections.unmodifiableList(simpleDataValues);
    }
}

```

wird von dem folgenden Controller benutzt:

```

public class SimpleDemoController {

    private SimpleDemoModel simpleDemoModel;
    public SimpleDemoModel getSimpleDemoModel() {
        return simpleDemoModel;
    }
    public void setSimpleDemoModel(SimpleDemoModel simpleDe-
moModel) {
        this.simpleDemoModel = simpleDemoModel;
    }
}

```

```
    }

    public ModelAndView insert(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException{

        SimpleDataValue simpleDataValue = new SimpleDataValue();

        simpleDataValue.setName("Name");

        simpleDataValue.setName(Long.toString(System.currentTimeMillis()));

        simpleDemoModel.insertSimpleDataValue(simpleDataValue);

        response.getWriter().write("Created SimpleDataValue " +
            simpleDataValue);

        return null;
    }

    public ModelAndView getSimpleDataValues(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException{

        List<SimpleDataValue> simpleDataValues = simpleDemoModel.getSimpleDataValueList();

        for (SimpleDataValue value: simpleDataValues){

            response.getWriter().write("<p>Found SimpleDataValue " +
                value + "</p>");

        }

        return null;
    }

}
```

Die zugehörige Spring-Konfiguration enthält die folgenden SpringBeans:

```
<bean id="handlerMapping"

    class="org.springframework.web.servlet.handler.SimpleUrl
HandlerMapping">

    <property name="mappings">

        <props>

            <prop key="/*.form">

                DemoController

            </prop>

            <prop key="/*.method">

                MultiActionController

            </prop>

        </props>

    </property>

</bean>
```

```

        </prop>
    </props>
</property>
</bean>

    <bean
id="org.javacream.spring.beans.web.mvc.model.SimpleDemoModel"
class="org.javacream.spring.beans.web.mvc.model.SimpleDemoModel"
">
    </bean>

    <bean
id="org.javacream.spring.beans.web.mvc.controller.SimpleDemoCon
troller"
class="org.javacream.spring.beans.web.mvc.controller.SimpleDemo
Controller">
        <property name="simpleDemoModel"
ref="org.javacream.spring.beans.web.mvc.model.SimpleDemoModel"/
>
    </bean>

    <bean id="MultiActionController"
class="org.springframework.web.servlet.mvc.multiaction.MultiAct
ionController">
        <property name="delegate"
ref="org.javacream.spring.beans.web.mvc.controller.SimpleDemoCo
ntroller"/>
        <property name="methodNameResolver">
            <bean
class="org.springframework.web.servlet.mvc.multiaction.Paramete
rMethodNameResolver">
                <property name="paramName" value="method"/>
            </bean>
        </property>
    </bean>

```

Das DispatcherServlet reagiert die URLs mit der Endung „\*.method“ durch ein neues Mapping in der web.xml:

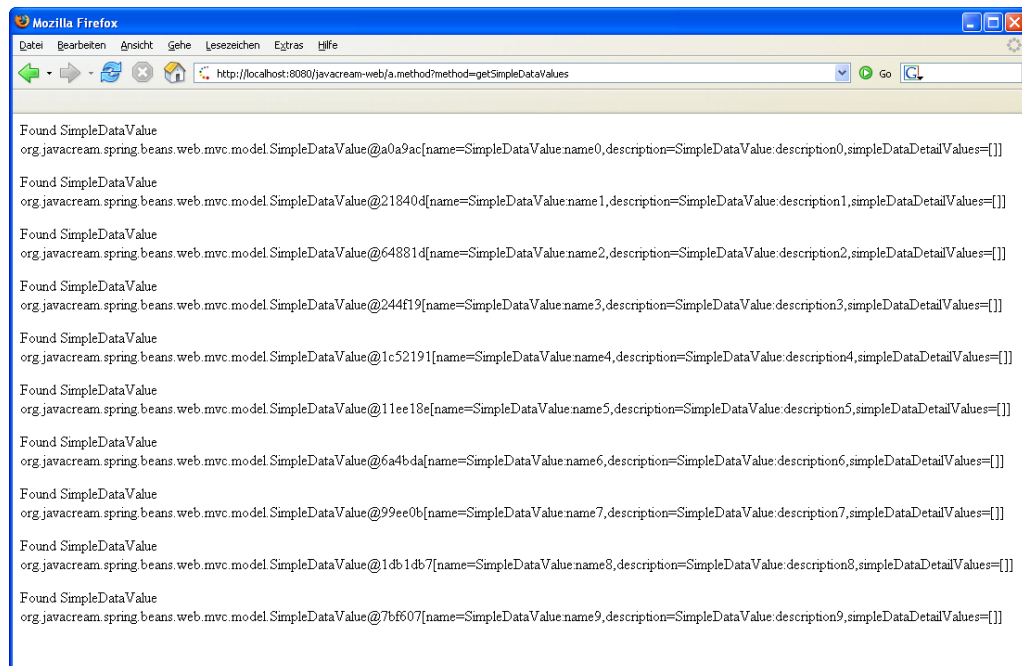
```

<servlet-mapping>
    <servlet-name>FrontController</servlet-name>
    <url-pattern>*.method</url-pattern>
</servlet-mapping>

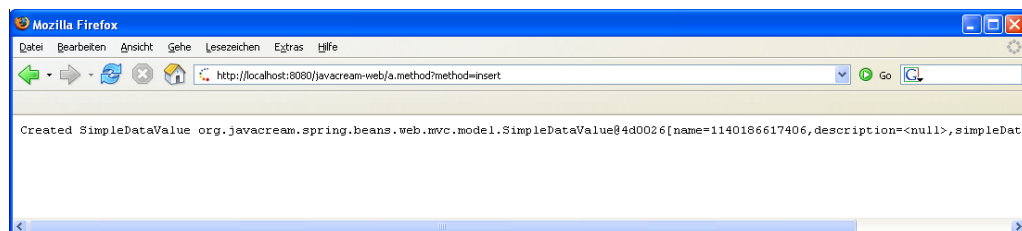
```



```
http://localhost:8080/javacream-  
web/a.method?method=getSimpleDataValues
```



```
http://localhost:8080/javacream-web/a.method?method=insert
```



## 4.4 Handler

Handler sind Implementierungen, die zwischen das Dispatcher-Servlet und den zugehörigen Controller geschaltet werden können. Ein Beispiel dafür sind die bereits verwendeten Handler-Mappings, die die Zuordnung zum Controller durchführen. Daneben können auch Interceptors eingefügt werden.

Diese Interceptors unterscheiden sich von den Servlet-Filtern des Servlet API dadurch, dass beim Aufruf eines Spring-Interceptors im Request bereits der Spring-Kontext zur Verfügung steht. Weiterhin werden diese Interceptors natürlich in der Spring-Konfigurationsdatei definiert und nicht in der `web.xml`.

Beispiel:

```
public class TracingInterceptor extends HandlerInterceptorAdapter {

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        Logger.getAnonymousLogger().log(Level.INFO, request.toString() + ", handler=" + handler + "modelAndView=" + modelAndView);
        super.postHandle(request, response, handler, modelAndView);
    }

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        Logger.getAnonymousLogger().log(Level.INFO, request.toString() + ", handler=" + handler);
        return super.preHandle(request, response, handler);
    }
}
```

## Spring-Konfiguration:

```
<bean id="handlerMapping"

    class="org.springframework.web.servlet.handler.SimpleUrl
HandlerMapping">

    <property name="interceptors">
        <list>
            <ref
an="org.javacream.spring.beans.web.mvc.interceptor.TracingInter
ceptor"/>
        </list>
    </property>
    <property name="mappings">
        <props>
            <prop key="/*.form">
                DemoController
            </prop>
            <prop key="/*.method">
                MultiActionController
            </prop>
        </props>
    </property>
</bean>

<bean
id="org.javacream.spring.beans.web.mvc.interceptor.TracingInter
ceptor"

    class="org.javacream.spring.beans.web.mvc.interceptor.Tr
acingInterceptor"/>
```

## 4.5 Views

In den vorherigen Beispielen war es Aufgabe des Controllers, die Antwort in den Servlet-Response zu schreiben. Diese Implementierungs-Strategie ist atypisch und sehr kontraproduktiv für größere Anwendungen.

Konform zum Framework ist die Rückgabe einer `ModelAndView`-Instanz: Diese Klasse definiert sowohl Elemente der Datenhaltung (also entweder des Modells oder von Ergebnis-Objekten, die in einer Map gehalten werden), als auch die Information darüber, welche View die Daten präsentieren soll.

Ist die View `null`, so ist es Aufgabe des Dispatcher-Servlets, die geeignete View zu finden. Setzt der Controller aber explizit eine View, so wird diese verwendet.

## 4.6 Web-Anwendungen und Form Controller

### 4.6.1 Form Controller

Mit den bisherigen Informationen könnten prinzipiell bereits komplexe Web-Anwendungen erstellt werden. Der Komfort des Spring-Frameworks ist jedoch deutlich größer, da eine Kopplung zwischen den im Browser dargestellten HTML-Formularen und den Controllern auf Server-Seite hergestellt wird. Diese Kopplung sind die `CommandController`.

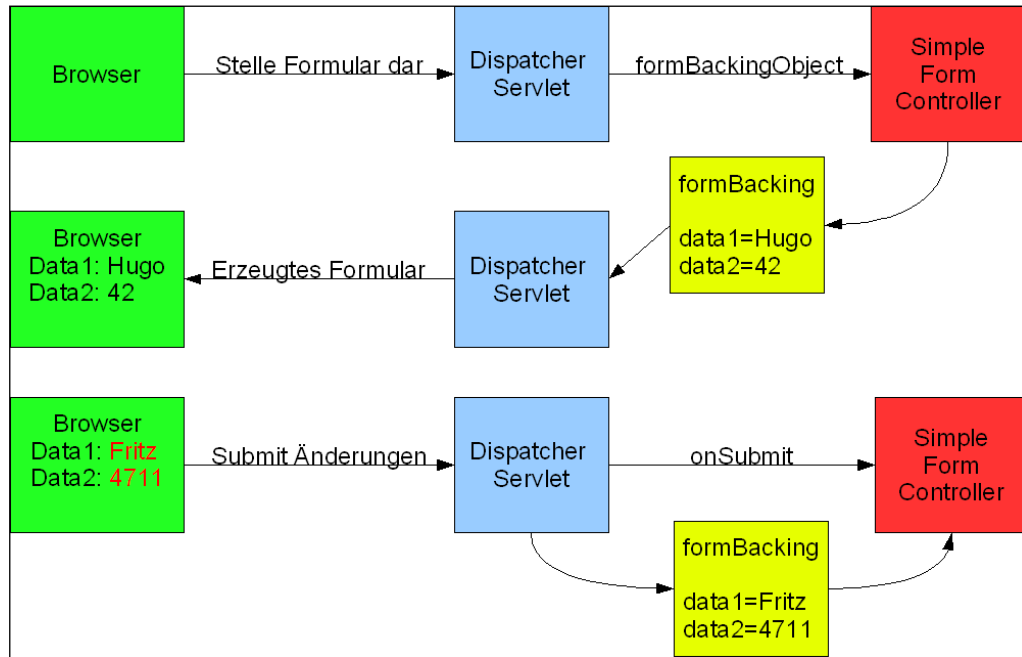
- `AbstractCommandController`: Eine Klasse, die automatisch aus dem `Http-Request` ein Datenobjekt erzeugt, eventuell mit vorhergehender Validierung.
- `AbstractFormController`: Diese Klasse macht die oben bereits angesprochene Kopplung zwischen Formular, Datenobjekt und Controller.
- `SimpleFormController`: Hier wird ein vollkonfigurativer Ansatz unterstützt: Der Controller, das Datenobjekt und die zu verwenden Views werden für diesen Controller in der Spring-Konfigurationsdatei eingetragen. Ein Beispiel dafür ist weiter unten gegeben.
- `AbstractWizardController`: Diese Klasse unterstützt nun Funktionalitäten wie Blättern etc., wie sie in typischen Dialoggesteuerten Web-Anwendungen benötigt werden.

### 4.6.2 Formulare

Für die Erstellung von Formularen ist eine eigene, vollständige Controller-Implementierung vorhanden: Der `SimpleFormController`.

Zur Implementierung eigener Funktionalität müssen in jedem Falle zwei Methoden überschrieben werden:

- `Object formBackingObject(HttpServletRequest request)`. Diese Methode wird aufgerufen, bevor das zugehörige Formular dargestellt wird. Der Rückgabewert dieses Aufrufs wird einesteils zur initialen Darstellung benutzt, andererseits wird er aber auch typischerweise in die Session gebunden. Das form backing object dient somit als das Daten-Modell der Web-Anwendung.
- `ModelAndView onSubmit(Object command)`. Diese Methode wird in einem zweiten Schritt (und damit als Ergebnis eines zweiten `Http-Requests`) aufgerufen. Der Übergabeparameter ist vom Typ des form backing objects und ist mit den Werten gefüllt, die der Anwender im `Html-Formular` eingegeben hat.



Das folgende Beispiel zeigt einen SimpleFormController für die BooksService-Anwendung:

```

public class BooksServiceFormController extends SimpleFormController {

    private BooksService booksService;

    public BooksService getBooksService() {
        return booksService;
    }

    public void setBooksService(BooksService booksService) {
        this.booksService = booksService;
    }

    @Override
    protected ModelAndView onSubmit(Object command) throws
Exception {
        System.out.println(command);
        ModelAndView mav = new ModelAndView(getSuccessView());
        BookValue bookValue = booksService.findBookByIsbn(((BookValue) command)
            .getIsbn());
        mav.addObject("book", bookValue);
        return mav;
    }
}

```

```

    }

    @Override
    protected Object formBackingObject(HttpServletRequest request)
        throws Exception {
        System.out.println("Calling formBackingObject");
        BookValue bv = new BookValue();
        bv.setIsbn("0815");
        return bv;
    }
}

```

Die zugehörige Spring-Konfiguration konfiguriert das zugehörige Mapping sowie den Controller:

```

<beans>
    <import resource="booksservice-spring-beans.xml" />
    <bean id="handlerMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandler
Mapping">
        <property name="mappings">
            <props>
                <prop key="/books.html">booksFormController</prop>
            </props>
        </property>
    </bean>

    <bean id="booksFormController"

        class="org.javacream.books.warehouse.web.BooksServiceFor
mController">
        <property name="sessionForm" value="true"></property>
        <property name="formView" valu-
e="books/findBooks"></property>
        <property name="successView" valu-
e="books/showBookResult"></property>
        <property name="commandName" value="book"></property>
        <property name="commandClass"
            value="org.javacream.books.warehouse.value.BookValue">
</property>
        <property name="bindOnNewForm" value="true" />
        <property name="booksService"
ref="mapBooksService"></property>
    </bean>

```

```

        <bean id="viewResolver"

            class="org.springframework.web.servlet.view.InternalResourceViewResolver">
                <property name="viewClass">
                    <value>org.springframework.web.servlet.view.JstlView</value>
                </property>
                <property name="prefix" value="/WEB-INF/jsp/" />
                <property name="suffix" value=".jsp" />
            </bean>

</beans>

```

Daneben wird für den View-Resolver noch eine JSP-Seite mit JSTL-Unterstützung angegeben.

Bei der Konfiguration des Controllers sind die folgenden Einstellungen wichtig:

- **formView:** Der Controller weiß, für welche Formularseite er zuständig ist.
- **commandName:** Unter diesem Namen wird das form backing object gebunden.
- **sessionForm:** Gültigkeitsbereich für das form backing object
- **commandClass:** Klassenname des form backing objects und des Kommando-Objekts in der `onSubmit`-Methode.
- **successView:** Name der nächsten darzustellenden Seite

Die zugehörigen JSP-Seiten haben für eine äußerst einfache Seite das folgende Aussehen:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@ taglib prefix="spring" uri="/spring"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Find Book</title>
</head>
<body>

<form method=post><spring:bind path="book.isbn">
    <input type="text" name="isbn" value="<c:out value=
"${status.value}"/>">
        <input type="submit" value="findBook">

```



```
</spring:bind></form>
</body>
</html>
```

Mit Hilfe des `<spring:bind>`-Tags wird eine Property des form backing objects in den aktuellen Status der JSP-Seite gelegt.

Auch die Success-View benutzt diesen Binding-Mechanismus:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<%@ taglib prefix="spring" uri="/spring"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Find Book</title>
</head>
<body>
<table>
  <tr>
    <td>ISBN</td>
    <td><spring:bind path="book.isbn">
      <c:out value="${status.value}" />
    </spring:bind></td>
  </tr>
  <tr>
    <td>Title</td>
    <td><spring:bind path="book.title">
      <c:out value="${status.value}" />
    </spring:bind></td>
  </tr>
  <tr>
    <td>Price</td>
    <td><spring:bind path="book.price">
      <c:out value="${status.value}" />
    </spring:bind></td>
  </tr>
  <tr>
    <td>Available</td>
    <td><spring:bind path="book.available">
      <c:out value="${status.value}" />
    </spring:bind></td>
  </tr>
</table>
</body>
```

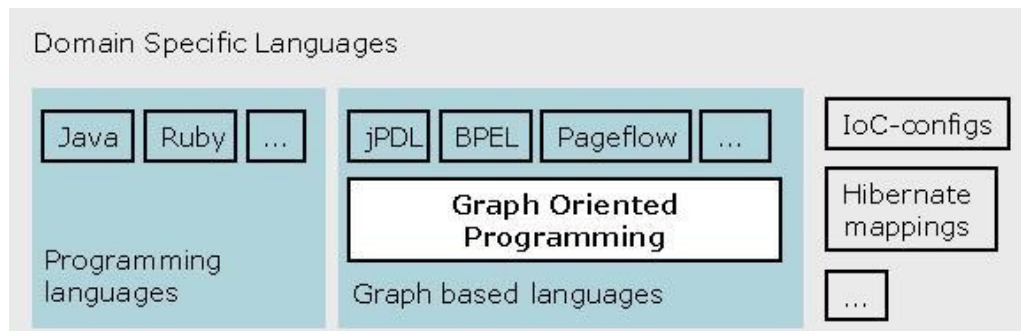
```
</html>
```

## 5 Spring Web Flow

### 5.1 Graphen-orientierte Programmierung<sup>12</sup>

#### 5.1.1 Übersicht

Die Graphen-orientierte Programmierung bildet das Rückgrat der Modellierung eines Business Prozesses. Dieser Ansatz ist sicherlich nicht für alle Aufgabenbereiche sinnvoll einzusetzen, aber genau dies hat sich in letzter Zeit immer mehr herauskristallisiert: Es gibt nicht **die** Programmiersprache, die alle Problembereiche innerhalb einer Unternehmensdomäne abdecken kann! Viel natürlicher ist eine Vielzahl von Domänensprachen, die jeweils ein Werkzeug für die optimale Lösung eines bestimmten Problembereichs bieten<sup>13</sup>. So bestehen moderne Applikationen in der Regel aus einem Mischmasch von Domänenspezifischen Sprachen:



Graphen-orientierte Sprachen sind konzipiert für die Umsetzung von Workflows, Orchestrierung bzw. Pageflows. Das Basiselement hierfür ist ein gerichteter Graph aus verschiedenen Zuständen und Transitionen.

<sup>12</sup> Grundlage dieses Abschnitts ist der jPGL-Userguide von JBoss. Auch ein Großteil der verwendeten Grafiken entstammt dieser hilfreichen Übersicht.

<sup>13</sup> Die daraus resultierende Heterogenität in den notwendigen Qualifikationen der IT-Mitarbeiter ist in der Praxis natürlich nicht vernachlässigbar. So kann nicht jedes Teammitglied gleichermaßen in allen Sprachen ausgebildet sein.

### 5.1.2 Eigenschaften der Graphen-orientierte Programmierung

Objektorientierte Sprache wie Java sind für die Umsetzung eher kurzlebiger Aufgaben konzipiert: Innerhalb einer laufenden Virtuellen Java Maschine werden bestimmte Sequenzen innerhalb einer Methode abgearbeitet. Innerhalb der Methode existiert ein Satz lokaler Variabler, Parameter und Attribute. Die Methoden können selbstverständlich Nebeneffekte produzieren, die beispielsweise innerhalb einer Datenbank abgespeichert werden können. Was jedoch nicht vorgesehen ist die Unterbrechung einer Methode wobei der aktuelle Zustand der Variablen automatisch abgespeichert wird, so dass der aktuelle oder aber auch ein komplett anderer Prozess zu einem völlig unabhängigen Zeitpunkt den Business Process wieder aufnehmen kann. Dieser so genannte „Wait State“ ist ein Grundelement der Graphen-orientierten Programmierung. Der gesamte Zustand des Methodenaufrufs innerhalb der Virtuellen Maschine muss jederzeit persistiert werden können.<sup>14</sup>

Ein weiterer Unterschied bezieht sich in der Möglichkeit, die Sequenzen zu visualisieren. Die Aktivitäts- bzw. Sequenzdiagramme der UML bieten zwar eine prinzipielle Möglichkeit, dies darzustellen. Allerdings sind die Menge von Schlüsselwörtern und die Anzahl der potenziell verwendeten so enorm, dass eine übersichtliche und standardisierte Darstellung extrem feingranular und damit komplett unübersichtlich wird. So kann zu Übungszwecken versucht werden, für die folgende doch recht triviale Logik ein Sequenzdiagramm zu erstellen<sup>15</sup>:

```
public void accumulate(String category, String id, String counterName,

    long add) {

    NamedCounter counter = null;

    synchronized (counterMaps) {

        HashMap<String, HashMap<String, NamedCounter>> counterMapForCategory = counterMaps

            .get(category);

        if (counterMapForCategory == null) {
```

---

<sup>14</sup> Selbstverständlich kann diese Anforderung durch eine Objekt-orientierte Programmiersprache implementiert werden. Bestes Beispiel hierfür ist jBPM von JBoss. Eine Graphen-orientierte Sprache stellt diese Funktionalität jedoch dem Entwickler bereits fertig zur Verfügung.

<sup>15</sup> Hinweis: Diese Anwendung erstellt eine Aufrufstatistik gruppiert nach Kategorien, einer Id und einem Namen.

```
        System.out.println("no counter for category " +
category);

        counterMapForCategory = new HashMap<String,
HashMap<String, NamedCounter>>();

        counterMaps.put(category, counterMapForCategory);
    }

    HashMap<String, NamedCounter> counterMapForId = counterMapForCategory

        .get(id);

    if (counterMapForId == null) {

        System.out.println("no counter for category " +
category + ", id " + id);

        counterMapForId = new HashMap<String,
NamedCounter>();

        counterMapForCategory.put(id, counterMapForId);
    }

    counter = counterMapForId.get(counterName);

    if (counter == null) {

        System.out.println("no counter for category " +
category + ", id " + id + ", name " + counterName);

        counter = new NamedCounter(id, counterName);

        counterMapForId.put(counterName, counter);
    }
}

counter.addTimeStep(add);
}
```

Die Essenz einer Graphen-orientierte Programmiersprache ist jedoch genau eine klare, übersichtliche visuelle Darstellung. Die Lösung ist nun fast unverschämt einfach: Statt eine Vielzahl von Sprachkonstrukten zu verwenden wird eine eher knapp kalkulierte Menge von grafischen Symbole definiert, die eine semantische Bedeutung besitzen. Natürlich ist damit die Granularität der Problembereiche eher grob, aber genau das ist ja gerade der Sinn verschiedener Domänensprache: Der Business Prozess wird grob mit Hilfe einer Graphen-orientierten Sprache

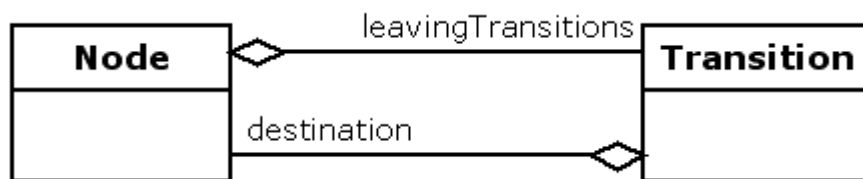
modelliert. Feingranulierte Sequenzen sind außerhalb des Verwendungsbereichs und bleiben bei klassischen Sprachen wie Java oder C#.

### 5.1.3 Prinzipielle Realisierung einer Graphen-orientierten Sprache

Dieser Abschnitt beschreibt, wie eine Graphen-orientierte Sprache mit Hilfe einer OOP-Sprache umgesetzt werden kann.

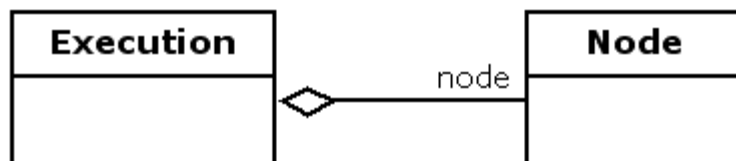
#### 5.1.3.1 Knoten und Transitionen

Die Struktur eines Graphen wird beschrieben durch `Nodes` und `Transitions`. Der Graph ist gerichtet, der `Node` besitzt somit ankommende und ausgehende `Transitions`:

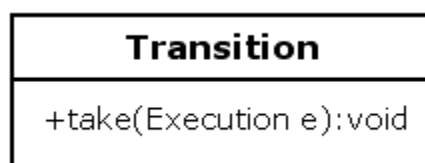


#### 5.1.3.2 Die Ausführung

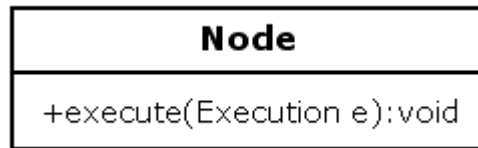
Eine `Execution` bzw. ein `Token` referenziert den aktuelle Knoten:



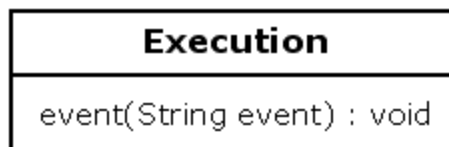
Die Aufgabe der `Transition` ist es, eine `Execution` entgegen zu nehmen. Dabei erfolgt ein Wechsel zwischen den `Nodes`:



Der Ziel-Node veranlasst die Ausführung der `Execution`:



Falls der `Node` die `Execution` nicht propagiert bleibt diese innerhalb des `Nodes`. Dann verhält sich der `Node` als ein Wait State. Damit die `Execution` irgendwann wieder aufgenommen werden kann wird ein Benachrichtigungsmechanismus über Events eingeführt:



## 5.2 Business Prozesse und Web Flows

Langlaufende Business Prozesse sind die klassischen Anwendungsfälle von Geschäftsanwendungen. Sowohl eine Business-To-Consumer Anwendung wie ein Online-Warenhaus oder der Komplex „Kontoführung“ innerhalb einer Bank weisen im Wesentlichen identische Merkmale auf. Diese führten in den letzten Jahren in zunehmendem Maße zur Entwicklung spezieller Laufzeitumgebungen und auch zur Spezifikation neuer Programmier-Modelle und -Sprachen. Beispiele hierfür sind die „Process Manager“ von IBM und Oracle, die als Programmiersprache BPEL (die Business Process Execution Language) favorisieren. Im reinen Java-Umfeld stellt jBPM von JBoss ein stabiles und etabliertes Open Source-Produkt dar.

Auch in Web Anwendungen werden Business Prozesse sinnvoll eingesetzt. **Web Flows** werden bei der Beschreibung der Seitennavigation nützlich eingesetzt: Jeder State repräsentiert dabei meistens eine Seite, die dem Benutzer im Browser zur Verfügung gestellt wird.

Im Gegensatz zu den Sequenzen einer klassischen Applikationslogik innerhalb einer Methode besitzt ein Flow eine andere Ausrichtung:

- Der Flow hat einen Zustand, seinen **State**, in Form von Flow-Variablen, der persistiert werden kann. Im Gegensatz dazu stehen die lokalen Variablen einer Methode einer traditionellen Programmiersprache.
- Der Aufruf kann aus verschiedenen Aufrufkontexten heraus erfolgen. Der Zustand eines Flows kann von jeder Anwendung zu beliebiger Zeit auf einer beliebigen Plattform restauriert werden. Im Gegensatz dazu wird eine Methode stets innerhalb eines einzelnen Threads einer Java Virtual Machine ausgeführt. Eine Restaurierung der Variablen erfordert die Wiederherstellung der Register-Inhalte der VM, was nach Beenden des Programms nicht mehr möglich ist.
- Flows können parallelisiert und auch wiederum zusammengefasst werden. Diese Eigenschaft ist nicht mit dem Multithreading von Java gleichzusetzen, da prinzipiell die einzelnen Flow-Abschnitte auch in verschiedenen Anwendungen ablaufen können.
- Flows können durch Diagramme beschrieben werden. Diese bestehen aus einem zumindest teilweise standardisierten Satz von Symbolen. Beispiele hierfür sind die Garrett-Diagramme bzw. die BPM.org.

Flows definieren Anwendungslogik, die in Programmen abgebildet werden müssen. Auf Grund der besonderen Anforderungen ist Java alleine als Programmiersprache jedoch ungeeignet: Insbesondere die simple Umsetzung von Wartezuständen kann nicht durch eine Virtuelle Maschine realisiert werden. Es ist absurd anzunehmen, dies durch blockierende Threads zu realisieren. Dafür ist in jedem Falle eine Persistierung innerhalb einer Datenbank vorzusehen.



Weiterhin sind Flows als Diagramme darstellbar und bestehen damit aus einer hierarchischen Ablaufstruktur. Dafür bietet sich XML natürlich als Sprache an.

Flows werden somit im Java-Umfeld durch eine Kombination aus

- XML für die Prozessbeschreibung,
- Java für kurz laufende Sequenzen, die bei einem State-Wechsel ablaufen können und
- einem Persistenz-Mechanismus für die Ablage der Flow-Variablen umgesetzt.

Die Persistierung erfolgt bei Web-Anwendungen typischerweise durch den Servlet-Container im Rahmen der Replikation/Speicherung der `HttpSession`.

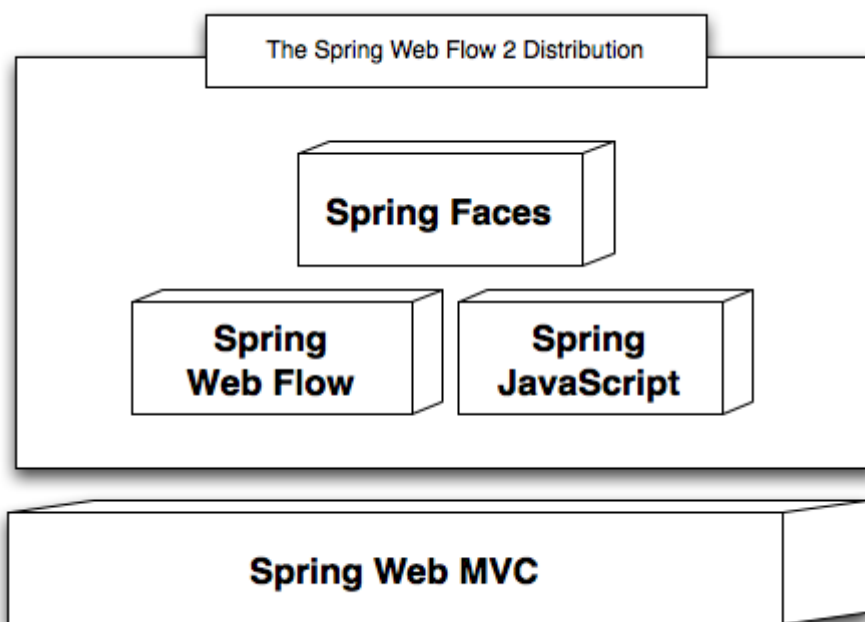
Durch die Kombination verschiedener Programmiermodelle und Laufzeitumgebungen ist es notwendig, den Datenaustausch über einen Kontext zu definieren. Insbesondere muss

- aus der XML-Flow-Definition auf Java-Properties und Methoden zugegriffen werden. Dies wird durch eine Expression Language ermöglicht werden. Im Java-Umfeld ist die JBoss-EL die Standard-Implementierung der Unified EL, die im Rahmen der JEE spezifiziert ist.
- Ebenso benötigt die Java-Sequenz Zugriff auf den aktuellen Zustand des Flows. Dies wird in der Regel durch eine Callback-Methode realisiert, die als Parameter eine Referenz auf ein Kontext- oder Event-Objekt bekommt.
- Die Persistierung der State-Variablen kann direkt durch ein Speichern in einer Datenbank, beispielsweise unter Verwendung eines O/R-Mappers wie Hibernate erfolgen. Bei Web Flows kann dies auch indirekt durch den Session Dienst des Servlet-Containers erfolgen. Dieser bietet laut JEE-Spezifikation in jedem Falle eine Unterstützung für den Request-, Session- und Application-Scope. Insgesamt benötigen Web Flows die folgenden Scopes, die selber wiederum den Gültigkeitsbereich von Flow-Variablen definieren:
  - **Flow Scope:** Dieser beginnt mit dem Starten des Flows und endet mit diesem. Flow Scoped-Variable müssen serialisiert werden können.
  - **View Scope:** Beginnt mit dem Eintritt in eine View und endet mit dem Austritt. View Scoped-Variable müssen serialisiert werden können, da ein View State mehrere Seitenaufrufe umfassen kann.
  - **Request Scope:** Für jeden Aufruf wird ein neuer Satz von Variablen bereitgestellt, die ab dem Ende des Aufrufs komplett entfernt werden.

- **Flash Scope:** Eine Mischung aus Flow und Request Scope: Die Variablen werden beim Start des Flows angelegt, aber nach jedem Rendern einer View zurückgesetzt. Auch diese Variablen müssen serialisierbar sein.
- **Conversational Scope:** Umfasst alle Flows inklusive sämtlicher eventuell aufgerufener Sub-Flows.

### 5.3 Übersicht

Spring Web Flow integriert sich auf folgende Art in Spring Enterprise:



- Spring Web MVC: Das normale Web Framework mit Controller, View Resolver und Templating.
- Spring JavaScript: Eine kapselnde Schicht um Client-seitiges JavaScript, insbesondere für die AJAX-Unterstützung.
- Spring Faces: Zur Anbindung der JavaServer Faces Technologie.

### 5.4 Konfiguration

Spring Web Flow benutzt eine normale Spring-Beans-Konfiguration. Zur vereinfachten Definition der notwendigen Spring Beans werden weitere Namensräume zur Verfügung gestellt:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xmlns:webflow="http://www.springframework.org/schema/webflow-
config"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-
beans-2.5.xsd
        http://www.springframework.org/schema/webflow-config
        http://www.springframework.org/schema/webflow-
config/spring-webflow-config-2.0.xsd">

    <!-- Setup Web Flow here -->

</beans>
```

Damit können die für den Spring Web Flow notwendigen Beans einfach konfiguriert werden.

#### 5.4.1 Die Flow Registry

Sämtliche Flow-Definitionsdateien werden in einer zentralen Registry angemeldet:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xmlns:webflow="http://www.springframework.org/schema/webflow-
config"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-
beans-2.5.xsd
        http://www.springframework.org/schema/webflow-config
        http://www.springframework.org/schema/webflow-
config/spring-webflow-config-2.0.xsd">

    <webflow:flow-registry id="flowRegistry">
        <webflow:flow-location path="/WEB-INF/flows/main.xml" />
    </webflow:flow-registry>

</beans>
```

Die Registry selber kann durch die Angabe eines `FlowBuilderServices` weiter konfiguriert werden. Ein `org.springframework.webflow.engine.builder.FlowBuilder` ist hierbei eine Implementierung dieser Schnittstelle, die selber wiederum als Fabrik für die folgenden Services dient:

- Der `conversion-service` dient zur Umwandlung von Zeichenketten in die entsprechenden Java-Typen. Die Default-Implementierung

stellt die gängigen Konversions-Algorithmen für Zahlen, logisch Werte etc. zur Verfügung.

- Der `expression-parser` implementiert die geforderte Anbindung der XML-Ausdrücke nach Java-Properties/-Methoden. Die Standard-Implementierung verwendet JBoss-EL bzw. ONGL.
- Der `view-factory-creator` veranlasst das Rendern einer View aus einem Template heraus. Die Standard-Implementierung benutzt hierbei das Spring MVC, das intern dann wieder die gängigen Template Engines (JavaServer Pages, Apache Velocity und Freemarker) unterstützt.

Beispiel:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

       xmlns:webflow="http://www.springframework.org/schema/webflow-
       config"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-
           beans-2.5.xsd
           http://www.springframework.org/schema/webflow-config
           http://www.springframework.org/schema/webflow-
           config/spring-webflow-config-2.0.xsd">

<webflow:flow-registry id="flowRegistry" flow-builder-
services="flowBuilderServices">
    <webflow:flow-location path="/WEB-INF/flows/main.xml" />
</webflow:flow-registry>

<webflow:flow-builder-services id="flowBuilderServices"
    conversion-service="conversionService"
    expression-parser="expressionParser"
    view-factory-creator="viewFactoryCreator" />

<bean id="conversionService" class="..." />
<bean id="expressionParser" class="..." />
<bean id="viewFactoryCreator" class="..." />
</beans>
```

Flow Registries können in einer hierarchischen Eltern-Kind-Beziehung stehen. Dabei delegiert die Kind-Registry die Zugriffe an die Eltern-Registry, falls sie eine Flow-Definition nicht selbst finden kann:

```
<!-- my-system-config.xml -->

<webflow:flow-registry id="flowRegistry" parent="sharedFlowRegistry">
```

```
<webflow:flow-location path="/WEB-INF/flows/booking/booking.xml" />
</webflow:flow-registry>
```

```
<!-- shared-config.xml -->
<webflow:flow-registry id="sharedFlowRegistry">
    <!-- Global flows shared by several applications -->
</webflow:flow-registry>
```

Welche Flows in der Registry angemeldet sind, wird durch verschiedene `webflow:flow-location`-Elemente angegeben. Diese unterstützen:

- Eine Pfadangabe

```
<webflow:flow-location
path="/WEB-INF/flows/booking/booking.xml"
```

- In der Pfadangabe sind Patterns möglich:

```
<webflow:flow-location-pattern value="/WEB-INF/flows/**/*-
flow.xml" />
```

- eine Id. Ohne Angabe ist die Id gleich dem Namen der Flow-Datei ohne Pfad und Endung:

```
<webflow:flow-location
path="/WEB-INF/flows/booking/booking.xml" id="bookHotel" />
```

- Zusätzliche Flow-Metadaten, die von der Flow-Laufzeitumgebung ausgelesen werden können und damit zu einem modifizierten Verhalten führen können. Welche Attributnamen dabei gültig sind wird später gezeigt.

```
<webflow:flow-location path="/WEB-INF/flows/booking/booking.xml">
    <flow-definition-attributes>
        <attribute name="caption" value="Books a hotel" />
        <attribute name="persistence-context" value="true" type="boolean" />
    </flow-definition-attributes>
</webflow:flow-location>
```

#### 5.4.2 Der Flow-Executor

Dies ist neben der Flow Registry der zweite zentrale Dienst, den jede Spring Web Flow Anwendung benötigt:

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xmlns:webflow="http://www.springframework.org/schema/webflow-
config"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-
beans-2.5.xsd
        http://www.springframework.org/schema/webflow-config
        http://www.springframework.org/schema/webflow-
config/spring-webflow-config-2.0.xsd">

<webflow:flow-executor id="flowExecutor"/>

</beans>

```

Der Flow Executor kann Listener registrieren, wobei jeder Listener optional als `criteria` die Ids der Flows bekommen kann, für die er angewendet werden soll:

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xmlns:webflow="http://www.springframework.org/schema/webflow-
config"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-
beans-2.5.xsd
        http://www.springframework.org/schema/webflow-config
        http://www.springframework.org/schema/webflow-
config/spring-webflow-config-2.0.xsd">

<flow-execution-listeners>
    <listener ref="securityListener" crite-
ria="securedFlow1,securedFlow2"/>
    <listener ref="persistenceListener"/>
</flow-execution-listeners>

</beans>

```

Der Executor selbst kann über zwei Attribute getuned werden:

- `max-executions`: maximale Anzahl von Flow-Instanzen pro User Session.

- `max-execution-snapshots`: Als Snapshot wird hierbei die History-Funktion bezeichnet. Um dieses Feature zu deaktivieren, wird der Wert 0 gesetzt, -1 steht für eine beliebige Menge von Snapshots.

## 5.5 Elementare Flow-Definition

### 5.5.1 Das Flow-Schema

Jede Spring Web Flow-Definition erfolgt in einem Schema-basierten XML-Dokument, das dann bei der Flow Registry angemeldet werden kann.

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      xsi:schemaLocation="http://www.springframework.org/schema/webflow
http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

</flow>
```

Innerhalb des Flows kann eine beliebige Menge von States angegeben werden. Der erste angegebene State ist automatisch der Start-Zustand.

### 5.5.2 Der View State

Ein View State repräsentiert einen Zustand, der als View gerendered werden kann:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      xsi:schemaLocation="http://www.springframework.org/schema/webflow
http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

<view-state id="enterBookingDetails" />

</flow>
```

Die Id des View States wird im Standard-Fall als Dateiname (ohne `xhtml`-Endung) interpretiert. Dabei liegt die Datei im selben Verzeichnis wie die zugehörige Flow-Definition. Explizit kann der Dateiname über das `view`-Attribut gesetzt werden.

Jede View kann über das `model`-Attribut ein zugehöriges Datenmodell zugeordnet bekommen. Dies wird im Kapitel über das Binding näher erläutert.

### 5.5.3 Transitionen

Innerhalb eines `view-state` können eine beliebige Menge von Transitionen definiert werden:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      xsi:schemaLocation="http://www.springframework.org/schema/webflow
http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

  <view-state id="enterBookingDetails" >
    <transition on="confirm" to="bookingConfirmed" />
    <transition on="revise" to="enterBookingDetails" />
    <transition on="cancel" to="bookingCancelled" />
  </view-state>

</flow>
```

### 5.5.4 End State

Jede Flow-Definition kann eine beliebige Menge von End States definieren:

```
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      xsi:schemaLocation="http://www.springframework.org/schema/webflow
http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

  <view-state id="enterBookingDetails">
    <transition on="submit" to="reviewBooking" />
  </view-state>

  <view-state id="reviewBooking">
    <transition on="confirm" to="bookingConfirmed" />
    <transition on="revise" to="enterBookingDetails" />
    <transition on="cancel" to="bookingCancelled" />
  </view-state>

  <end-state id="bookingConfirmed" />

</flow>
```



```
<end-state id="bookingCancelled" />

</flow>
```

## 5.5.5 Flow-Parameter und Ergebnisse

### 5.5.5.1 Input

Jeder Flow kann Parameter definieren. Dazu dient das `input`-Tag:

```
<input name="hotelId" />
```

Ohne weitere Konfiguration wird dieser Parameter im Flow Scope unter dem angegebenen Namen abgelegt.

Weitere Konfiguration sind möglich:

- Angabe des Typs:

```
<input name="hotelId" type="long" />
```

- Angabe von Scope und Name:

```
<input name="hotelId" value="flowScope.myParameterObject.hotelId" />
```

- Required Parameter:

```
<input name="hotelId" type="long" value="flowScope.hotelId" required="true" />
```

- **Output**

Der End State kann ein Ergebnis liefern:

```
<end-state id="bookingConfirmed">
    <output name="bookingId" />
</end-state>
```

Hier wird als Ergebnis der Wert der Variablen `bookingId` zurückgegeben. Genauer kann die Variable durch die Angabe eines `value` definiert werden:

```
<output name="confirmationNumber" value="booking.confirmationNumber" />
```

## 5.5.6 Variablen

Jeder Flow kann eine beliebige Menge von Variablen deklarieren, die beim Start des Flows angelegt werden.

```
<var name="searchCriteria"
class="com.mycompany.myapp.hotels.search.SearchCriteria"/>
```

Die Variablen werden von der Flow-Laufzeitumgebung im Flow Scope

abgelegt. Enthält die implementierende Klasse Spring Beans-Annotations werden diese interpretiert. Insbesondere werden alle `@Autowired`-Elemente korrekt verdrahtet.

### 5.5.7 Sub Flows

Jeder Flow kann einen anderen Flow aufrufen. Dazu dient der `sub-flow-state`:

```
<subflow-state id="addGuest" subflow="createGuest">
    <input name="booking" />
    <transition to="reviewBooking" />
</subflow-state>
```

Eventuelle Parameter werden durch `input`-Tags gesetzt.

### 5.5.8 Vererbung

#### 5.5.8.1 Der Vererbungs-Mechanismus

Im Gegensatz zum Konzept der Vererbung in Objekt-orientierten Sprachen werden die Eigenschaften der jeweiligen Parent-Elemente mit den im Child definierten Elementen zusammengeführt.

#### 5.5.8.2 Flow-Vererbung

Jeder Flow kann von einer Liste von Parent-Flows erben:

```
<flow parent="common-transitions, common-states">
```

Die hier angegebenen Parent-Flows müssen wie alle anderen Flows auch in die Registry eingetragen werden.

Sind Parent-Flows nicht zur Ausführung gedacht kann das Tag `abstract` benutzt werden:

```
<flow abstract="true">
```

#### 5.5.8.3 State-Vererbung

Ein State kann von einem State eines Parent-Flows erben, falls dieser vom selben Typ ist:

```
<view-state id="child-state" parent="parent-flow#parent-view-state">
```

## 5.6 Die Unified Expression Language

### 5.6.1 Objektzugriff

Der Zugriff auf die Java-Objekte innerhalb des Flows erfolgt über eine Expression Language. Die Anbindung an Java erfolgt dabei recht einfach:

- Jedes Objekt besitzt einen Alias-Namen und kann darüber referenziert werden.
- Der Zugriff auf Attribute erfolgt ohne den Umweg über getter- und setter-Methoden durch `alias.attribut`.
- Analog erfolgt der Zugriff auf Methoden, wobei nun ein rundes Klammernpaar optionale Parameterlisten aufnehmen kann:  
`alias.method(..)`
- Durch Objektbäume wird durch einfache Verkettung mit Hilfe des Punktoperator navigiert.
- Der Zugriff auf Maps erfolgt über ein paar eckiger Klammern, in denen der Schlüssel angegeben wird: `mappedValue[key]`

### 5.6.2 Typen von Expressions

#### 5.6.2.1 Eval Expressions

Eine Eval-Expression wird dynamisch zur Laufzeit ausgewertet.

```
searchCriteria.nextPage()
```

#### 5.6.2.2 Template Expressions

Diese erlauben ein Mischen von Fixtext mit variablen Ausdrücken. Template Expressions müssen durch den `${}`-Ausdruck eingegrenzt werden.

```
<view-state id="error" view="error-  
${externalContext.locale}.xhtml"
```

### 5.6.3 Spezielle Variablen

Die folgenden Variablen sind implizit innerhalb von Flow-Elementen vorhanden:

Name	Beschreibung
<code>flowScope</code>	Der Flow Scope
<code>viewScope</code>	Der View Scope
<code>requestScope</code>	Der Request Scope
<code>flashScope</code>	Der Flash Scope
<code>conversationScope</code>	Der Conversation Scope
<code>requestParameters</code>	Zugriff auf die Request Parameter eines Client-Aufrufs
<code>currentEvent</code>	Zugriff auf das aktuelle Event-Objekt. Jeder Event hat eine Quelle ( <code>source</code> ), eine <code>id</code> und eine Map von Attributen, die <code>attributes</code> .
<code>currentUser</code>	Der authentifizierte Benutzer als <code>Principal</code> mit einem <code>name</code> .
<code>messageContext</code>	Zugriff auf den aktuellen <code>MessageContext</code> . Dies ist im Wesentlichen ein Container für Messages.
<code>resourceBundle</code>	Zugriff auf die Properties eines Resource Bundels. Dieses wird im Standard-Fall unter dem Namen <code>messages</code> im Flow-Verzeichnis gesucht.
<code>externalContext</code>	Zugriff auf aktuelle Client-Informationen und damit auch auf dessen User Session.
<code>flowRequestContext</code>	Zugriff auf den <code>RequestContext</code> . Dieser repräsentiert den aktuellen Thread des Aufrufs und enthält somit beispielsweise Informationen über die aktuell ausgeführte Aktion.
<code>flowExecutionContext</code>	Zugriff auf den <code>FlowRequestContext</code> . Dieser enthält den aktuellen Flow und liefert beispielsweise Zugriff auf den Flash und Conversation Scope.
<code>flowExecutionUrl</code>	Relative URL des aktuellen Aufrufs.

### 5.6.4 Such-Algorithmus für Objekte

Soll ein Objekt einem Scope zugeordnet werden, ist die Angabe der entsprechenden speziellen Variable unerlässlich. Beim Zugriff ist die Angabe der Scope-Variable optional, die Flow-Laufzeitumgebung durchsucht die Scopes in der Reihenfolge `request`, `flash`, `view`, `flow`, und `conversation`.

## 5.7 Actions

### 5.7.1 Events

Die eben eingeführte Expression Language ermöglicht es, Zugriff auf Java-Objekte zu erhalten. Actions definieren die Stelle, an der dieser erfolgt. Dazu definiert Spring Web Flow die folgenden Events:

#### 5.7.1.1 on-start

```
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      xsi:schemaLocation="http://www.springframework.org/schema/webflow
http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <input name="hotelId" />

    <on-start>
        <evaluate expression="bookingService.createBooking(hotelId, currentUser.name) "
            result="flowScope.booking" />
    </on-start>

</flow>
```

#### 5.7.1.2 on-entry

```
<view-state id="changeSearchCriteria"
view="enterSearchCriteria.xhtml" popup="true">
    <on-entry>
        <render fragments="hotelSearchForm" />
    </on-entry>
</view-state>
```

### 5.7.1.3 on-render

```
<view-state id="reviewHotels">
    <on-render>
        <evaluate expression=
on="bookingService.findHotels(searchCriteria)" re-
sult="viewScope.hotels" result-type="dataModel" />
    </on-render>
</view-state>
```

### 5.7.1.4 on-transition

```
<transition on="guestCreated" to="reviewBooking">
<evaluate expression=
"booking.guestList.add(currentEvent.attributes.newGuest)" />
</transition>
```

Liefert eine Action ein Error-Ergebnis bzw. `false`, so wird die Transition nicht durchgeführt.

Eine Action ohne `to`-Attribut verbleibt im aufrufenden State und wird als Event-handler bezeichnet.

### 5.7.1.5 on-exit

```
<view-state id="editOrder">
    <on-exit>
        <evaluate expression="orderService.releaseLock(order,
currentUser)" />
    </on-exit>
</view-state>
```

### 5.7.1.6 on-end

```
<flow xmlns="http://www.springframework.org/schema/webflow"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/webfl
ow

http://www.springframework.org/schema/webflow/spring-webflow-
2.0.xsd">

    <on-end>
        <evaluate expression="orderService.releaseLock(order,
currentUser)" />
    </on-end>

</flow>
```

### 5.7.2 Evaluate

Mit Hilfe von `evaluate` wird die Aktion ausgeführt. Optional können Ergebnisse Variablen zugeordnet werden:

```
<evaluate expression="bookingService.findHotels(searchCriteria)" result="flowScope.hotels" result-type="dataModel"/>
```

### 5.7.3 action-state und decision-state

Ein `action-state` ermöglicht die Ausführung einer Aktion. Dieser State wird zu einem anderen State, wohl einem `view-state` weiterleiten:

```
<action-state id="moreAnswersNeeded">
    <evaluate expression="interview.moreAnswersNeeded()" />
    <transition on="yes" to="answerQuestions" />
    <transition on="no" to="finish" />
</action-state>
```

Ein `decision-state` verzweigt in Abhängigkeit von einer boolschen Variable:

```
<decision-state id="moreAnswersNeeded">
    <if test="interview.moreAnswersNeeded"
    then="answerQuestions" else="finish"/>
</decision-state>
```

Nachdem Transitionen immer ein Event-Objekt als Parameter bekommen, müssen die Rückgabewerte der in `action-state` oder `decision-state` aufgerufenen Methoden speziell interpretiert werden:

Rückgabotyp	Ausdruck des erzeugten Events
<code>java.lang.Boolean</code>	yes (für <code>true</code> ), no (für <code>false</code> )
<code>java.lang.Enum</code>	Der Name der Enumeration
<code>java.lang.String</code>	Der Wert der Zeichenkette
Alle anderen	„success“

## 5.8 Views

### 5.8.1 view-state

Views werden durch den bereits bekannten `view-state` definiert.

Der Flow bleibt so lange in diesem State, bis eine Benutzeraktion registriert wurde, beispielsweise ein Submit. Dann kann der State über eine Transition verlassen werden:

```
<view-state id="enterBookingDetails">
    <transition on="submit" to="reviewBooking" />
</view-state>
```

Transitionen ohne das `to`-Attribut verbleiben auf derselben View, diese wird dann aber neu gerendered. Auf diese Art und Weise kann beispielsweise eine View Scope-Variable über mehrere Aufrufe hinweg benutzt werden.

Views können durch Angabe des `popup`-Attributs als Popup-Fenster dargestellt werden:

```
<view-state id="changeSearchCriteria"
view="enterSearchCriteria.xhtml" popup="true">
```

### 5.8.2 Globale Transaktionen

Sollen alle Views einen gemeinsamen Satz von Transaktionen enthalten, so können diese global auf der Ebene der Flow-Definition eingetragen werden:

```
<global-transitions>
    <transition on="login" to="login">
    <transition on="logout" to="logout">
</global-transitions>
```

### 5.8.3 View Template

Zur Angabe des zu verwendenden Templates sind mehrere Möglichkeiten vorhanden:

- Ohne Angabe des `view`-Attributs wird davon ausgegangen, dass eine im Verzeichnis des Flows eine Datei vorhanden ist, die genauso heißt wie die `id` des `view-state` und die Endung `.xhtml` hat.
- Das `view`-Attribut kann eine relative oder absolute Dateiangabe sein:

```
<view-state id="enterBookingDetails"
view="bookingDetails.xhtml">
<view-state id="enterBookingDetails" view="/WEB-INF/hotels/booking/bookingDetails.xhtml">
```



- Wird als Hintergrundtechnologie das Spring MVC-Framework benutzt, so wird `view` als logischer Name interpretiert, der dann vom `ViewResolver` konkret aufgelöst wird.

#### 5.8.4 Partielles Rendern

Bei einer Transition kann angegeben werden, dass als Ergebnis nur ein Teil der Seite zurück gesendet wird:

```
<transition on="next">
    <evaluate expression="searchCriteria.nextPage()" />
    <render fragments="searchResultsFragment" />
</transition>
```

Dies ist insbesondere für AJAX-Anwendungen interessant.

#### 5.8.5 Binding

##### 5.8.5.1 Definition und Arbeitsweise

Jede View kann an ein Datenmodell gebunden werden. Das zugehörige Java-Objekt muss dazu Properties gemäß den Bean-Konventionen definieren.

```
<view-state id="enterBookingDetails" model="booking">
</view-state>
```

Definiert der `view-state` das Attribut `model`, so passiert bei einem Request folgendes:

- Die Parameter des Requests werden dem Java-Objekt durch Aufruf der setter-Methoden gesetzt.
- Danach wird eine Validierung durchgeführt.
- Ist diese erfolgreich, so wird eine definierte Transition ausgeführt.
- Sonst wird die View erneut gerendered.

Für bestimmte Transitionen kann das Binding aufgehoben werden. Sinnvoll ist dies beispielsweise bei Cancel-Transitionen:

```
<view-state id="enterBookingDetails" model="booking">
    <transition on="proceed" to="reviewBooking">
    <transition on="cancel" to="bookingCancelled" bind="false"
/>
</view-state>
```

### 5.8.5.2 Binder-Konfiguration

Durch die Angabe eines `binder`-Tags kann das potenzielle Binding auf einen festgelegten Satz von Bean-Properties beschränkt werden:

```
<view-state id="enterBookingDetails" model="booking">
  <binder>
    <binding property="creditCard" />
    <binding property="creditCardName" />
    <binding property="creditCardExpiryMonth" />
    <binding property="creditCardExpiryYear" />
  </binder>
  <transition on="proceed" to="reviewBooking" />
  <transition on="cancel" to="cancel" bind="false" />
</view-state>
```

Sinnvoll ist dies insbesondere aus Security-Gründen: Werden einfach alle Properties der JavaBean wie im vorigen Abschnitt über Request-Parameter setzbar gemacht, so kann versucht werden, auch kritische Properties durch eine Manipulation des Http-Requests zu setzen. Dies wird durch dieses explizite Binding verhindert.

Zusätzlich liefert das Binding auch noch die Möglichkeit, spezielle Konverter zu verwenden:

```
<binding property="checkinDate" converter="shortDate" />
```

Ebenso kann eine Property als `required` gekennzeichnet werden:

```
<binding property="checkinDate" converter="shortDate" required="true" />
```

### 5.8.5.3 Validierung

Zur Validierung existieren zwei Ansätze:

- Erweiterung der als Model verwendeten Klasse um eine Validierungs-Methode.

```
public void validateEnterBookingDetails(MessageContext context)
{
    Calendar calendar = Calendar.getInstance();
    if (checkinDate.before(today())) {
        context.addMessage(new MessageBuilder().error().source("checkinDate").defaultText(
            "Check in date must be a future date").build());
    } else if (!checkinDate.before(checkoutDate)) {
        context.addMessage(new MessageBuilder().error().source("checkoutDate").defaultText(
            "Check out date must be later than check in
            date").build());
    }
}
```

- Schreiben einer eigenen Validator-Klasse:

```
@Component
public class BookingValidator {
    public void validateEnterBookingDetails(Booking booking,
        MessageContext context) {
        if (booking.getCheckinDate().before(today())) {
            context.addMessage(new MessageBuilder().error().source("checkinDate").defaultText(
                "Check in date must be a future
                date").build());
        } else if (!booking.getCheckinDate().before(checkoutDate)) {
            context.addMessage(new MessageBuilder().error().source("checkoutDate").defaultText(
                "Check out date must be later than check in
                date").build());
        }
    }
}
```

Die Zuordnung des Validators zu einem Modell ist nicht ganz trivial:

- Der Validator selber muss eine Spring Bean sein. Dies wird in obigem Beispiel durch die `@Component`-Annotation gesichert.
- Die Id der Spring Bean muss der Konvention `<model-name>validator` genügen. In obigem Beispiel wird Spring die Komponente per Default unter dem Namen `booking-`

Validator registrieren. Diese wird damit als Validator für das booking-Modell benutzt.

Ebenso wie das Binding selber kann eine Transition auch die Validierung umgehen:

```
<view-state id="chooseAmenities" model="booking">
    <transition on="proceed" to="reviewBooking">
        <transition on="back" to="enterBookingDetails" validate="false" />
    </transition>
</view-state>
```

### 5.8.6 View History

Ein Anwender kann mit Hilfe des Back-Buttons des Browsers eine vorher dargestellte Seite darstellen. Dieses Feature ist im Spring Web Flow im Standard-Fall unterstützt.

Soll dieses Verhalten geändert werden, sind zwei Möglichkeiten vorhanden.

- **discard:** Eine Zurückgehen zur Quelle einer Transition wird verhindert:

```
<transition on="cancel" to="bookingCancelled" history="discard">
```

- **invalidate:** Nun kann keine im Laufe der Flow-Verarbeitung angezeigten Views benutzt werden:

```
<transition on="confirm" to="bookingConfirmed" history="invalidate">
```

## 5.9 Testen von Flows

Zum Testen von Flows stellt Spring Web Flow eine abstrakte Basis-Klasse zur Verfügung:

```
org.springframework.webflow.test.execution.AbstractXmlFlowExecutionTests
```

Diese Klasse definiert eine Reihe von Methoden, die bei Bedarf überschrieben werden können:

```
protected FlowDefinitionResource getResource(  
    FlowDefinitionResourceFactory resourceFactory) {  
    return resourceFactory.createFileResource("WebContent/WEB-INF/flows/main.xml");  
}
```

Diese Methode lädt den zu testenden Flow.

```
protected void configureFlowBuilder-  
Context(MockFlowBuilderContext builderContext) {  
    builderContext.registerBean("bookingService", new StubBookingService());  
}
```

Hier können Objekte in den Flow-Context gelegt werden. Dies können die echten Business-Objekte sein, aber natürlich auch Dummies oder Mock-Objekte.

Flows können von anderen Flows erben<sup>16</sup>, diese weiteren Definitionen werden von der folgenden Methode geladen:

```
protected FlowDefinitionResource[] getModel-  
Resources(FlowDefinitionResourceFactory resourceFactory) {  
    return new FlowDefinitionResource[] {  
        resourceFactory.createFileResource("src/main/webapp/WEB-INF/flows/common/common.xml")  
    };  
}
```

---

<sup>16</sup> Dies wird weiter unten behandelt.

Eine typische Test-Methode sieht folgendermaßen aus:

```
public void testMainFlow(){
    MockExternalContext context = new MockExternal-
Context();
    startFlow(context);
    assertEquals("page1", context.getCurrentState());
    context.setEventId("go");
    resumeFlow(context);
    assertEquals("page2", context.getCurrentState());
    context.setEventId("go");
    resumeFlow(context);
    assertEquals("page3", context.getCurrentState());
    context.setEventId("go");
    resumeFlow(context);
    assertEquals("page1", context.getCurrentState());
}
```

## 5.10 Erste Beispiele

### 5.10.1 Simple Flow

#### 5.10.1.1 Web Seiten

Der folgende einfache Flow besteht aus drei trivialen Seiten, die im Wesentlichen denselben Aufbau haben:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; char-
set=UTF-8" />
<title>Page <i> Title</title>
</head>
<body>
<h1>
Page <i>
</h1>
<form method="post">
<input type="submit" name="_eventId_forward" value="go for-
ward"/>
<input type="submit" name="_eventId_backward" value="go back-
ward"/>
</form>
</body>
</html>
```

### 5.10.1.2 Spring Beans

Die Spring Beans der Anwendung sind sehr einfach: Sie enthalten die elementare Konfiguration der Flow-Engine, den `FlowController` von Spring Web Flow sowie die Anbindung an das `SimpleUrlHandlerMapping` von Spring MVC. Letzteres lädt die Web Seiten der Anwendung über eine simple Namensauflösung unter Verwendung der JSP-Technologie. Ersterer stellt sicher, dass die Server-seitige Flow-Instanz sowie jeder Aufruf eine eindeutige Id bekommen. Damit können die verschiedenen eingeführten Scopes unterstützt werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:webflow="http://www.springframework.org/schema/webflow-config"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/webflow-config
           http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.0.xsd"
       ">

    <webflow:flow-registry id="flowRegistry">
        <webflow:flow-location path="/WEB-INF/flows/main.xml"
            id="main" />
    </webflow:flow-registry>
    <webflow:flow-executor id="flowExecutor" />
    <bean id="flowController"
class="org.springframework.webflow.mvc.servlet.FlowController">
        <property name="flowExecutor" ref="flowExecutor" />
    </bean>
    <bean id="flowUrlMappings"

        class="org.springframework.web.servlet.handler.SimpleUrl
HandlerMapping">
        <property name="mappings">
            <value>
                /main=flowController
            </value>
        </property>
    </bean>
</beans>
```

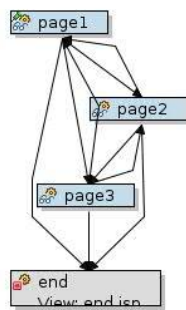


### 5.10.1.3 Die Flow-Definition

Die folgende Flow-Definition verlinkt die Seiten miteinander:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">
  <view-state id="page1">
    <transition on="forward" to="page2"></transition>
    <transition on="backward" to="page3"></transition>
    <transition on="end" to="end"></transition>
  </view-state>
  <view-state id="page2">
    <transition on="forward" to="page3"></transition>
    <transition on="backward" to="page1"></transition>
    <transition on="end" to="end"></transition>
  </view-state>
  <view-state id="page3">
    <transition on="forward" to="page1"></transition>
    <transition on="backward" to="page2"></transition>
    <transition on="end" to="end"></transition>
  </view-state>
  <end-state id="end" view="end.jsp"></end-state>
</flow>
```

oder Visualisiert:

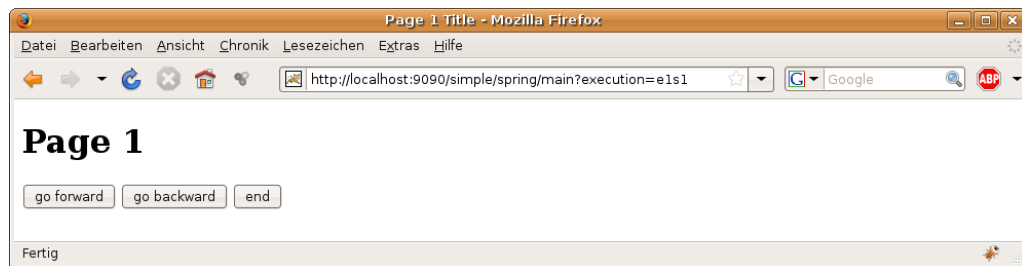


Damit ist die Seitennavigation und damit bereits die gesamte einfache Anwendung fertig gestellt.

#### 5.10.1.4 Laufzeit

Die folgende Einstiegsseite startet einen neuen Flow:

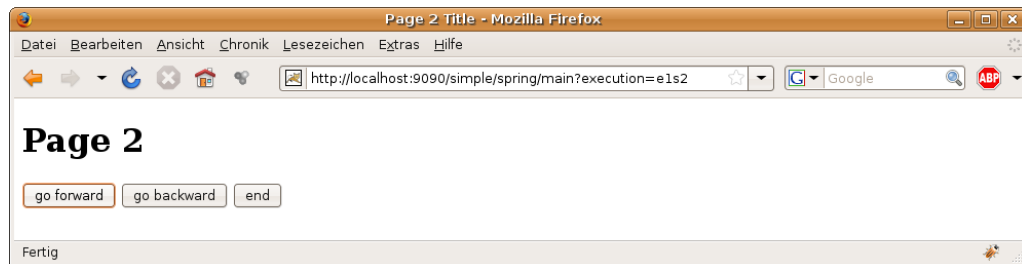
```
<html>
<head>
</head>
<body>
<a href="spring/main">Enter Simple Flow</a>
</body>
</html>
```



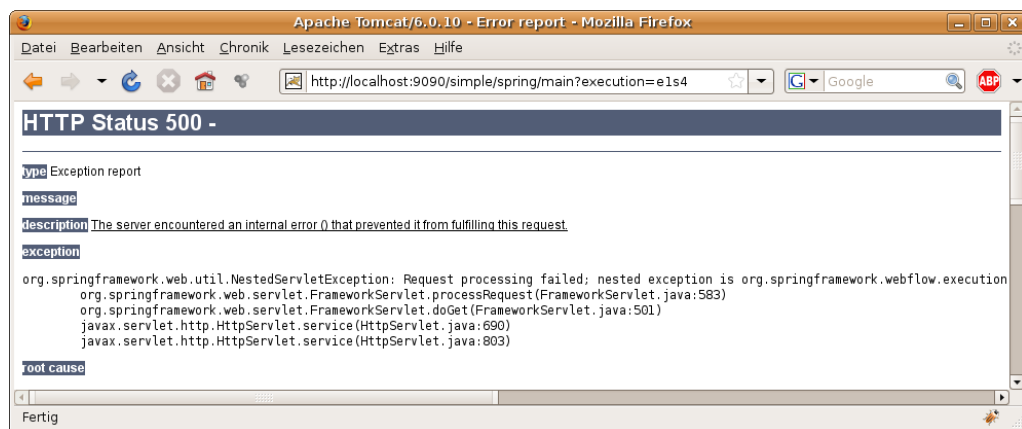
Zu beachten ist insbesondere die Url des Aufrufs, die eine execution-Id bekommen hat:

```
http://localhost:9090/simple/spring/main?execution=e1s1
```

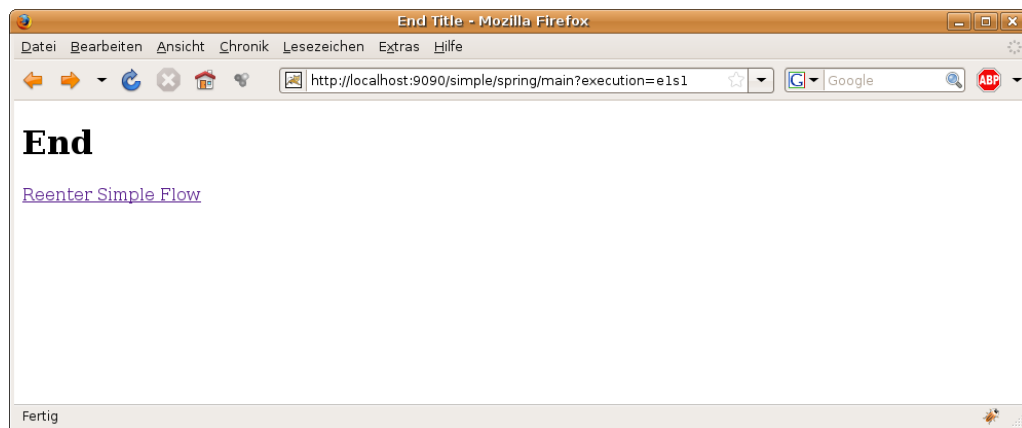
Jede Navigation wird an Hand der Flow-Definition abgehandelt und bekommt eine eigene Id:



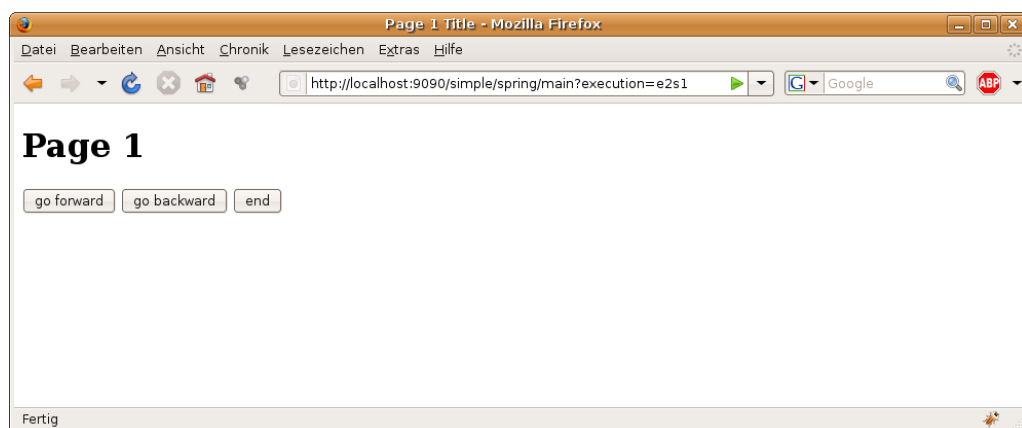
Durch den Aufbau der Historie ist ein zurückgehen möglich<sup>17</sup>. Unbekannte Ids führen jedoch natürlich zu einer Fehlermeldung:



Das Ende des Flows bekommt keine eigene Id:



Der Versuch, nun auf eine vorhandene Execution zu verweisen führt zum Starten eines neuen Flows:



<sup>17</sup> Die daraus resultierende Sicherheitsproblematik wird später behandelt.

### 5.10.2 Flow Variablen

Der folgende Flow definiert zwei Variablen vom Typ einer Benutzerdefinierten Klasse:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <var name="flowScopedDataBean"
        class="org.javacream.demo.DataBean" />

    <on-start>
        <evaluate expression="flowScopedDataBean.setData('Demo Data') "></evaluate>
        <evaluate expression="flowScopedDataBean.setCounter(42) "></evaluate>
    </on-start>

    <view-state id="inputPage">
        <var name="viewScopedDataBean"
            class="org.javacream.demo.DataBean" />
        <on-render>
            <evaluate expression="viewScopedDataBean.setData('View Scoped Demo Data') "></evaluate>
            <evaluate expression="viewScopedDataBean.setCounter(4711) "></evaluate>
        </on-render>
        <transition on="forward" to="outputPage" />
    </view-state>

    <view-state id="outputPage">
        <transition on="end" to="end"></transition>
    </view-state>

    <end-state id="end" view="end.jsp"></end-state>
</flow>
```

Der Zugriff auf diese Variablen aus einer JSP-Seite heraus erfolgt so:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; char-
set=UTF-8" />
<title>Page 1 Title</title>
</head>
<body>
<h1>
Input:
</h1>
<table>
<tr><td>Data (Flow):
</td><td>${flowScopedDataBean.data}</td></tr>
<tr><td>Counter (Flow):
</td><td>${flowScopedDataBean.counter}</td></tr>
<tr><td>Data
(View):</td><td>${viewScopedDataBean.data}</td></tr>
<tr><td>Counter (View):
</td><td>${viewScopedDataBean.counter}</td></tr>
</table>
<form method="post">
<input type="submit" name="_eventId_forward" value="go for-
ward"/>
</form>
</body>
</html>
```

## 5.11 Ausnahme-Verarbeitung

Sowohl dem Flow als auch jedem State kann ein `exception-handler` hinzugefügt werden. Dieses Tag referenziert eine Spring-Bean, deren Klasse die Schnittstelle `org.springframework.webflow.engine.FlowExecutionExceptionHandler` implementiert.

Darin sind zwei Methoden enthalten:

```
boolean canHandle(FlowExecutionException exception)
```

Hier wird bestimmt, ob dieser Handler diese Exception bzw. den Typ der darin transportierten `cause-Exception` behandeln kann.

```
void handle(FlowExecutionException exception, RequestContextContext context)
```

Hier erfolgt die eigentliche Verarbeitung. Zentrales Element hierbei ist der `RequestControlContext`, mit dessen Hilfe die verschiedensten Flow-Aktionen durchgeführt werden können. Das folgende Beispiel führt exemplarisch eine Transition in einen Zustand namens `handled` aus:

```
public void handle(FlowExecutionException exception,
    RequestContextContext requestControlContext) {
    System.out.println("Exception: " + exception);
    Transition transition = new Transition(new DefaultTargetStateResolver(
        "handled"));
    requestControlContext.execute(transition);
}
```

## 5.12 Dienste

### 5.12.1 Flow Managed Persistence

Innerhalb einer Spring Beans-Konfiguration kann eine Anbindung an einen O/R-Mapper wie Hibernate oder aber das JPA erfolgen. Dazu wird für den zweiten Fall eine `EntityManagerFactory` sowie sinnvollerweise ein `TransactionManager` definiert.

Wird der JPA-Persistenzkontext ausschließlich innerhalb des Service-Layers benutzt, so werden dem Web Flow ausschließlich detached Value Objects zur Verfügung gestellt. Ein „lazy load“ von Relationen bzw. eine automatische Synchronisation von Zustandsänderungen ist somit innerhalb der Web Schicht nicht möglich.

Spring Web Flow ermöglicht es nun aber, den Entity-Charakter der Objekte vorzuziehen. Dazu ist es nur notwendig, den Flow mit einem `persistence-context`-Element zu versehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

      xsi:schemaLocation="http://www.springframework.org/schema/webflow
http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <persistence-context>

</flow>
```

Die Webflow-Konfiguration benutzt einen speziellen Listener:

```
<webflow:flow-executor id="flowExecutor" flow-
registry="flowRegistry">
    <webflow:flow-execution-listeners>
        <webflow:listener ref="jpaFlowExecutionListener" />
    </webflow:flow-execution-listeners>
</webflow:flow-executor>

<bean id="jpaFlowExecutionListener"

class="org.springframework.webflow.persistence.JpaFlowExecution
Listener">
    <constructor-arg ref="entityManagerFactory" />
    <constructor-arg ref="transactionManager" />
</bean>
```

Diese, so genannte Flow Managed Persistence muss natürlich auch über einen Transaktionsrahmen gesteuert werden können. Das Element `end-state` enthält das `commit`-Attribut:

```
<end-state id="bookingConfirmed" commit="true" />
```

### 5.12.2 Security

Zur Absicherung des Flows sind mehrere Dinge notwendig:

- Erstellen einer Spring-Bean-Konfiguration mit Beans aus dem Spring-Security-Projekt:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:security="http://www.springframework.org/schema/security"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-
beans-2.5.xsd
           http://www.springframework.org/schema/security
           http://www.springframework.org/schema/security/spring-security-
2.0.2.xsd">

    <!-- Configure Spring Security -->
    <security:http auto-config="true">
        <security:form-login login-page="/spring/login" login-
processing-url="/spring/loginProcess" default-target-
url="/spring/main" authentication-failure-
url="/spring/login?login_error=1" />
        <security:logout logout-url="/spring/logout" logout-
success-url="/spring/logoutSuccess" />
    </security:http>

    <!--
        Define local authentication provider, a real app would
        use an external provider (JDBC, LDAP, CAS, etc)

        usernames/passwords are:
            keith/melbourne
            erwin/leuven
            jeremy/atlanta
            scott/rochester
    -->
    <security:authentication-provider>
        <security:password-encoder hash="md5" />
```



```

        <security:user-service>
            <security:user name="keith" pass-
word="417c7382b16c395bc25b5da1398cf076" authorities="ROLE_USER,
ROLE_SUPERVISOR" />
            <security:user name="erwin" pass-
word="12430911a8af075c6f41c6976af22b09" authorities="ROLE_USER,
ROLE_SUPERVISOR" />
            <security:user name="jeremy" pass-
word="57c6cbff0d421449be820763f03139eb" authorities="ROLE_USER"
/>
            <security:user name="scott" pass-
word="942f2339bf50796de535a384f0d1af3e" authorities="ROLE_USER"
/>
        </security:user-service>
    </security:authentication-provider>

</beans>

```

Hier wird als Security-Mechanismus BASIC eingestellt, das heißt, der Web Server schickt eine Aufforderung an den Browser, den Anmelde-dialog für Benutzername und Passwort zu öffnen. Als Security Provider wird zu Testzwecken eine einfache Implementierung benutzt, die die Daten aus der Konfiguration selber ausliest.

Die `web.xml` muss einen Filter für die Security definieren. Dieser benötigt bereits die geladene Spring-Konfiguration, die über einen weiteren Filter zur Verfügung gestellt wird:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">
    <display-name>Simple Spring Web Flow</display-name>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/web-application-config.xml
        </param-value>
    </context-param>
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-
class>org.springframework.web.filter.DelegatingFilterProxy</fil
ter-class>

```

```
        </filter>
        <filter-mapping>
            <filter-name>springSecurityFilterChain</filter-name>
            <url-pattern>/*</url-pattern>
        </filter-mapping>
        <listener>
            <listener-
class>org.springframework.web.context.ContextLoaderListener</li
stener-class>
        </listener>
        <servlet>
            <servlet-name>Spring Dispatcher Servlet</servlet-name>
            <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servle
t-class>
            <init-param>
                <param-name>contextConfigLocation</param-name>
                <param-value></param-value>
            </init-param>
        </servlet>
        <servlet-mapping>
            <servlet-name>Spring Dispatcher Servlet</servlet-name>
            <url-pattern>/spring/*</url-pattern>
        </servlet-mapping>
        <welcome-file-list>
            <welcome-file>index.html</welcome-file>
        </welcome-file-list>
    </web-app>
```

Das Dispatcher-Servlet muss nun keine Bean-Konfigurationsdatei mehr laden, der bisher vorhandene Initialisierungsparameter bleibt deshalb leer.

Die Flow-Definition benutzt nun das `secured-Element`. Dieses ist in verschiedenen Umgebungen (`flow`, `view-state`, `transition`) gültig und definiert die zur Ausführung notwendigen Rollen:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">
  <secured attributes="ROLE_USER"/>
  <view-state id="page1">
    <transition on="forward" to="page2"></transition>
    <transition on="backward" to="page3"></transition>
    <transition on="end" to="end"></transition>
  </view-state>
  <view-state id="page2">
    <transition on="forward" to="page3"></transition>
    <transition on="backward" to="page1"></transition>
    <transition on="end" to="end"></transition>
  </view-state>
  <view-state id="page3">
    <transition on="forward" to="page1"></transition>
    <transition on="backward" to="page2"></transition>
    <transition on="end" to="end"></transition>
  </view-state>
  <end-state id="end" view="end.jsp"></end-state>
</flow>
```

## 5.13 Integration mit JavaServer Faces

### 5.13.1 Flow-Konfiguration

Für die Integration mit JavaServer Faces wird die Web Flow-Registry mit einem speziellen `FlowBuilderService` konfiguriert. Zur Vereinfachung wird dafür ein eigener Namensraum definiert. Weiterhin werden ein `FlowController` und ein `HandlerMapping` benötigt:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:webflow="http://www.springframework.org/schema/webflow-
       config"
       xmlns:faces="http://www.springframework.org/schema/faces"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-
           beans-2.5.xsd
           http://www.springframework.org/schema/webflow-config
           http://www.springframework.org/schema/webflow-
           config/spring-webflow-config-2.0.xsd
           http://www.springframework.org/schema/faces
           http://www.springframework.org/schema/faces/spring-
           faces-2.0.xsd"
       ">
    <import re-
    source="config/services/booksservice/booksservice-spring-
    beans.xml" />
    <import resource="config/common-beans.xml" />
    <webflow:flow-registry id="flowRegistry"
        flow-builder-services="facesFlowBuilderServices">
        <webflow:flow-location path="/WEB-
        INF/flows/commonBookFlow.xml"
            id="commonBook" />
        <webflow:flow-location path="/WEB-
        INF/flows/newBookFlow.xml"
            id="newBook" />
        <webflow:flow-location path="/WEB-
        INF/flows/manageBooksFlow.xml"
            id="manageBooks" />
    </webflow:flow-registry>
    <webflow:flow-executor id="flowExecutor" />

    <!-- Configures the Spring Web Flow JSF integration -->
    <faces:flow-builder-services
    id="facesFlowBuilderServices" />
```

```

        <bean id="flowController"
class="org.springframework.webflow.mvc.servlet.FlowController">
        <property name="flowExecutor" ref="flowExecutor" />
        </bean>
        <bean id="flowUrlMappings"

        class="org.springframework.web.servlet.handler.SimpleUrl
HandlerMapping">
        <property name="mappings">
        <props>
                <prop key="/newBook">flowController</prop>
                <prop key="/manageBooks">flowController</prop>
        </props>
        </property>
        </bean>

</beans>

```

### 5.13.2 Änderungen in der web.xml

Obwohl zur Laufzeit nicht benutzt wird zum Hochfahren der Faces-Laufzeitumgebung das Faces-Servlet konfiguriert. Weiterhin werden die JSF-Tag-Libraries eingebunden:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        id="WebApp_ID" version="2.5">
        <display-name>Simple Spring Web Flow</display-name>
        <servlet>
        <servlet-name>Spring Dispatcher Servlet</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servle
t-class>
        <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/web-application-
config.xml</param-value>
        </init-param>
        </servlet>
        <servlet-mapping>
        <servlet-name>Spring Dispatcher Servlet</servlet-name>
        <url-pattern>/spring/*</url-pattern>
        </servlet-mapping>

```

```

        <!--
            Just here so the JSF implementation can initialize,
*not* used at
            runtime
        -->
        <servlet>
            <servlet-name>Faces Servlet</servlet-name>
            <servlet-
class>javax.faces.webapp.FacesServlet</servlet-class>
            <load-on-startup>1</load-on-startup>
        </servlet>

        <!-- Just here so the JSF implementation can initialize
    -->

        <servlet-mapping>
            <servlet-name>Faces Servlet</servlet-name>
            <url-pattern>*.faces</url-pattern>
        </servlet-mapping>

        <welcome-file-list>
            <welcome-file>index.html</welcome-file>
        </welcome-file-list>
        <jsp-config>
            <!--
                JSP Taglib Eintrag fuer JSTL
=====
            -->
            <taglib>
uri>                <taglib-uri>http://java.sun.com/jstl/core</taglib-
uri>
                <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
            </taglib>

            <taglib>
                <taglib-uri>http://java.sun.com/jstl/xml</taglib-uri>
                <taglib-location>/WEB-INF/tld/x.tld</taglib-location>
            </taglib>

            <taglib>
                <taglib-uri>http://java.sun.com/jstl/fmt</taglib-uri>
                <taglib-location>/WEB-INF/tld/fmt.tld</taglib-
location>
            </taglib>

            <taglib>

```

```
        <taglib-uri>http://java.sun.com/jstl/sql</taglib-uri>
        <taglib-location>/WEB-INF/tld/sql.tld</taglib-
location>
        </taglib>
        <!--
            JSP Taglib Eintrag
=====
        -->
        </jsp-config>
</web-app>
```

Wird eine Ajax-Unterstützung mit dem Spring Javascript-Projekt bzw. den Spring JSF-Komponenten gewünscht, so wird das ResourceServlet benötigt:

```
        <!-- Serves static resource content from .jar files such
as spring-faces.jar -->
        <servlet>
            <servlet-name>Resources Servlet</servlet-name>
            <servlet-
class>org.springframework.js.resource.ResourceServlet</servlet-
class>
            <load-on-startup>0</load-on-startup>
        </servlet>

        <!-- Map all /resources requests to the Resource Servlet
for handling -->
        <servlet-mapping>
            <servlet-name>Resources Servlet</servlet-name>
            <url-pattern>/resources/*</url-pattern>
        </servlet-mapping>
```

Die Faces-Konfiguration kann trivial bleiben, muss aber vorhanden sein:

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
    "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config
    1.0//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
</faces-config>
```

### 5.13.3 Spring-Beans-Konfiguration

Die Spring-Bean-Konfiguration bleibt im Wesentlichen unverändert. Im folgenden Beispiel soll für die Benutzung einer Bean das AOP-Framework benutzt werden, deshalb werden in der Flow-Definition keine Variablen deklariert, sondern innerhalb dieser Konfiguration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schem
a/beans http://www.springframework.org/schema/beans/spring-
beans-2.5.xsd">
    <import
        re-
source="classpath:/services/keygeneration/keygenerator-spring-
beans.xml" />
    <import
        re-
source="classpath:/services/storeservice/storeservice-spring-
beans.xml" />

    <bean id="mapBooksService"
class="org.javacream.books.model.business.MapBooksService"
        scope="flow" init-method="prepare">
        <property name="keyGenerator"
ref="randomKeyGenerator"></property>
        <property name="storeService"
ref="storeServiceImpl"></property>
    </bean>

    <bean id="tracer"
class="org.javacream.spring.util.SystemTraceInterceptor">
    </bean>

    <bean id="tracingProxy"
class="org.springframework.aop.framework.ProxyFactoryBean"
        scope="flow" name="booksService">
```



```

        <property name="target"
ref="mapBooksService"></property>
        <property name="interceptorNames">
            <list>
                <value>tracer</value>
            </list>
        </property>
    </bean>
</beans>

```

#### 5.13.4 Ein Beispiel-Flow

Das folgende Beispiel benutzt die Bücher-Applikation. Zur Rekapitulation:

- Publisher haben eine Liste von Book (1:n-Relation)
- Authors haben Books und umgekehrt (m:n-Relation)
- Ein Book hat seine BookStatistics mit Verkaufszahlen
- Der BooksService stellt Methoden zum Anlegen/Suchen/Löschen und Datenobjekten zur Verfügung.

Der folgende Page Flow definiert einige elementare Abläufe<sup>18</sup>:

Der Superflow importiert die notwendigen SpringBeans, deklariert Variablen und enthält einen Super-View-State:

```

<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/webflow http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd"
    abstract="true">
    <var name="book"
class="org.javacream.books.model.value.Book" />
    <var name="helper"
class="org.javacream.spring.webflow.util.Helper" />

    <view-state id="commonViewState">
        <on-entry>
            <evaluate expression="helper.saveSourceView(flowRequestContext)"></evaluate>

```

<sup>18</sup> Ob dieses Beispiel den Einsatz des Web Flows notwendig macht ist hier nicht Sinn der Erläuterung. Es wird eine einfache Seitennavigation benötigt, um das Beispiel nicht unnötig komplex zu machen.

```
        </on-entry>
        <on-exit>
            <set name="flowScope.lastView" value="requestScope['webflow.OriginatingViewState'].id"></set>
        </on-exit>
    </view-state>
    <view-state id="error" view="/WEB-INF/pages/error.jsp">
        <transition on="back" to="{source}">
        </transition>
        <transition on="end" to="end"></transition>
    </view-state>
    <end-state id="end" view="/WEB-INF/pages/end.jsp"></end-state>

    <global-transitions>
        <transition on="end" to="end"></transition>
        <transition on-exception="java.lang.Exception" to="error"></transition>
        <transition on="back" to="{flowScope.lastView}"></transition>
    </global-transitions>

    <bean-import
        resource="../../config/services/booksservice/booksservice-spring-beans.xml" />
</flow>
```

Die Super-State-Definition speichert die letzte Seite in einer allgemeinen Variable ab, die über eine globale Transaktion angesprungen werden können.

Zwei Sub-Flows sind vorhanden:

- Anlegen eines neuen Buches:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd"
      parent="commonBook">
  <var name="book"
class="org.javacream.books.model.value.Book" />
  <view-state id="titleInput" view="/WEB-INF/pages/newBookTitleInput.jsp"
    parent="commonBook#commonViewState">
    <on-entry>
      <evaluate expression="helper.dump(resourceBundle.titleInput)"></evaluate>
    </on-entry>
    <transition on="newBook" to="success">
      <evaluate expression="booksService.newBook(flowScope.book.title)"
        result="flowScope.book"></evaluate>
    </transition>
  </view-state>
  <view-state id="success" view="/WEB-INF/pages/newBookSuccess.jsp"
    parent="commonBook#commonViewState">
  </view-state>
</flow>
```

- oder etwas komplexer: Management von Büchern:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd"
      parent="commonBook">
  <view-state id="booksList" view="/WEB-INF/pages/booksList.jsp"
    parent="commonBook#commonViewState">
  </view-state>
</flow>
```

```

        <on-render>
            <evaluate expression="booksService.getBooks()" re-
result="viewScope.booksList"
            result-type="dataModel"></evaluate>
        </on-render>

        <transition on="detail" to="showBookDetail">
            <set name="book" valu-
e="viewScope.booksList.selectedRow"></set>
        </transition>

        <transition on="publisherDetail"
to="showPublisherDetail">
            <set name="flowScope.publisher" valu-
e="viewScope.booksList.selectedRow.publisher"></set>
        </transition>

        <transition on="delete">
            <evaluate
expressi-
on="booksService.deleteBookByIsbn(viewScope.booksList.selectedR
ow.isbn)"></evaluate>
        </transition>
    </view-state>

    <view-state id="showBookDetail" model="book"
        view="/WEB-INF/pages/bookDetail.jsp" pa-
rent="commonBook#commonViewState">
        <on-entry>
            <evaluate expression="book.getAuthors()" re-
result="viewScope.authors"
            result-type="dataModel"></evaluate>
        </on-entry>
        <transition on="update">
            <evaluate expressi-
on="booksService.updateBook(book)"></evaluate>
        </transition>
        <transition on="authorDetail" to="showAuthorDetail">
            <set name="flowScope.author" valu-
e="viewScope.authors.selectedRow"></set>
        </transition>
        <transition on="booksList" to="booksList">
        </transition>
    </view-state>

    <view-state id="showPublisherDetail" model="publisher"

```

```

        view="/WEB-INF/pages/publisherDetail.jsp" parent="commonBook#commonViewState">
        <transition on="updatePublisher">
        <evaluate expression="booksService.updatePublisher(publisher)"></evaluate>
        </transition>
        <transition on="booksList" to="booksList">
        </transition>
        </view-state>
        <view-state id="showAuthorDetail" model="author"
        view="/WEB-INF/pages/authorDetail.jsp" parent="commonBook#commonViewState">
        <transition on="updateAuthor">
        <evaluate expression="booksService.updateAuthor(author)"></evaluate>
        </transition>
        <transition on="bookDetail" to="showBookDetail">
        </transition>
        </view-state>
    </flow>

```

Zu beachten ist hier insbesondere der Datentyp bei der Evaluierung der Listen: hier wird `dataModel` benutzt. Dies ist eine Erweiterung des `Faces-DataModels` und bietet zusätzlich die Eigenschaft `selectedRow` an. Dies ist für die Ermittlung der innerhalb eines `dataGrids` ausgewählten Zeile nützlich.

### 5.13.5 Die Seiten

Die Seiten sind normale JSF-Seiten, die als Aktionen die Ids von Transitionen angeben. Weiterhin werden alle Spring Beans sowie die Flow-Variablen zur Verfügung gestellt.

Die folgende Seite zeigt beispielsweise die Bücherliste mit einer Verlinkung auf die Detailansichten von Verlegern und Büchern:

```

<%@ page language="java" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Books list</title>
</head>

```

```
<body>
<f:view>
    <h:form>
        <h:dataTable id="books" value="#{booksList}"
var="theBook">
            <h:column>
                <f:facet name="header">
                    <h:outputText value="ISBN"/>
                </f:facet>
                <h:outputText value="#{theBook.isbn}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Title"/>
                </f:facet>
                <h:outputText value="#{theBook.title}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Available"/>
                </f:facet>
                <h:outputText value="#{theBook.available}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Sold"/>
                </f:facet>
                <h:outputText value="#{theBook.bookStatistics.sold}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Publisher"/>
                </f:facet>
                <h:commandLink action="publisherDetail">
                    <h:outputText valu-
e="#{theBook.publisher.publisherName}"/>
                </h:commandLink>
            </h:column>
            <h:column>
```

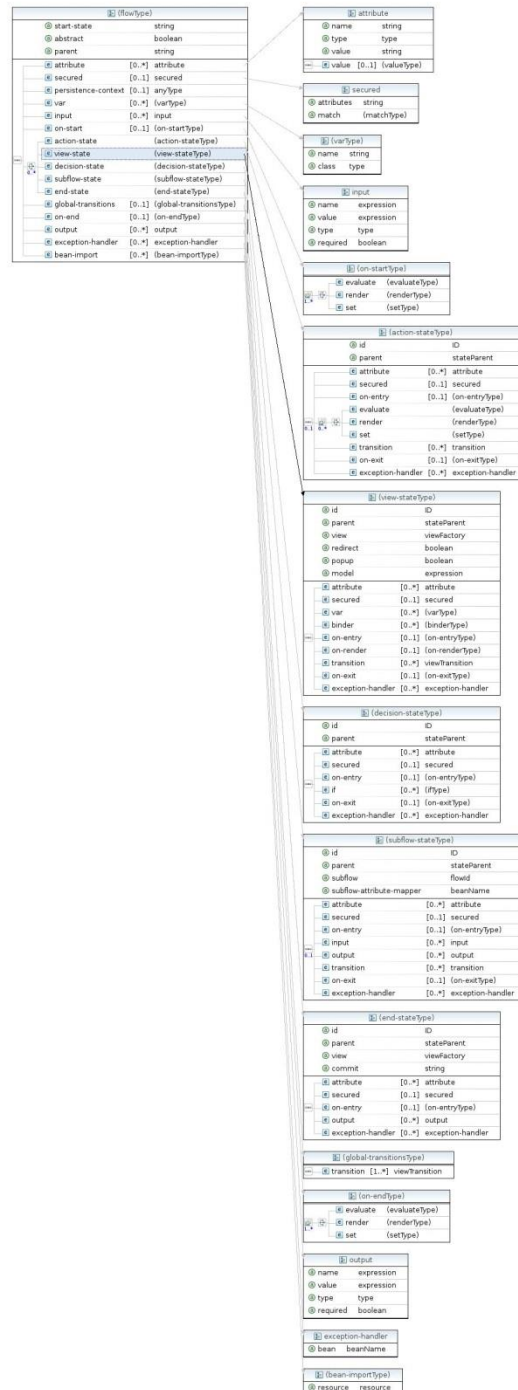
```
<f:facet name="header">
    <h:outputText value=""/>
</f:facet>
<h:commandButton value="Delete" action="delete"/>
<h:commandButton value="Detail" action="detail"/>
</h:column>
</h:dataTable>
    <h:commandButton value="#{resourceBundle.back}" ac-
tion="back"/>
    <h:commandButton value="#{resourceBundle.end}" ac-
tion="end"/>

</h:form>
</f:view>
</body>
</html>
```

## 5.14 Schema-Dateien

### 5.14.1 Flow-Definition

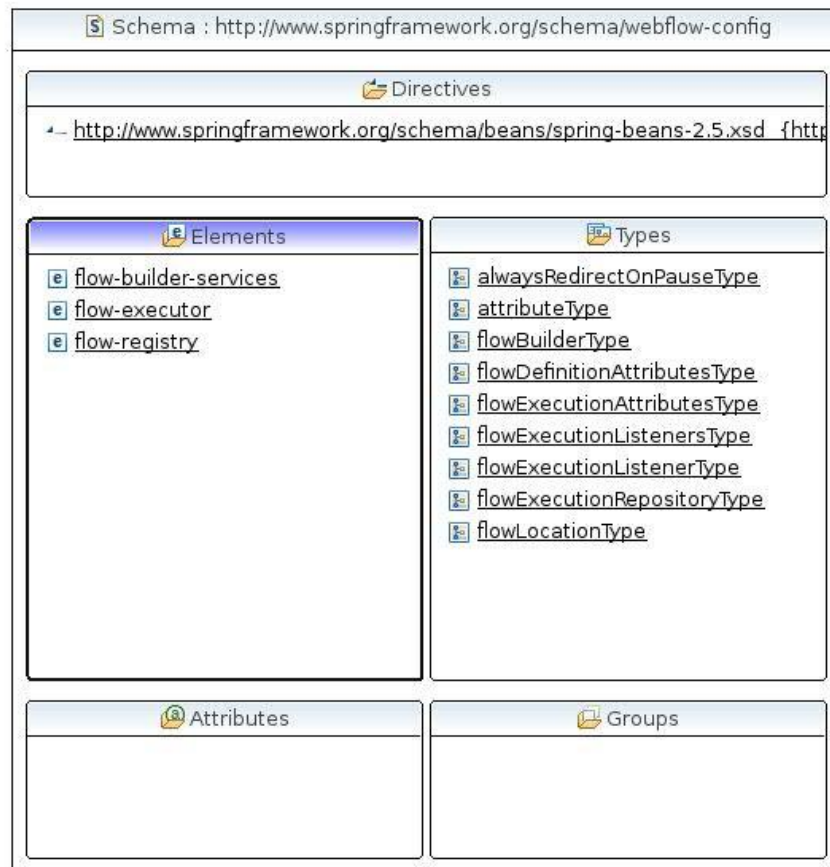
Die folgende Abbildung zeigt das Schema für die Flow-Definition:



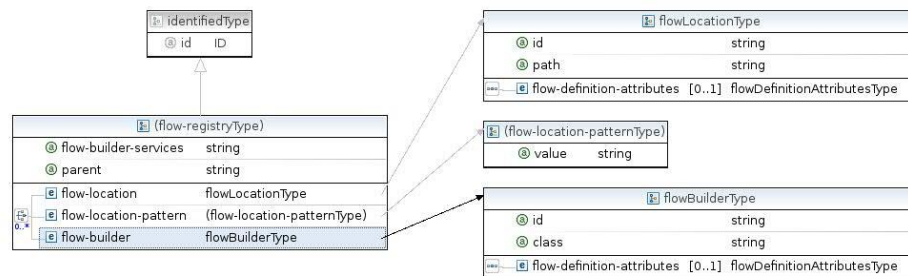


## 5.14.2 Die Flow-Konfiguration

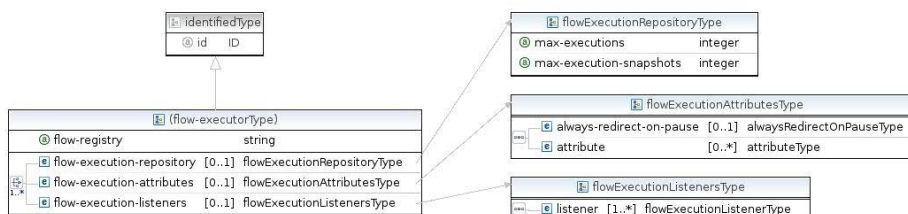
### 5.14.2.1 Übersicht



### 5.14.2.2 Flow Registry



### 5.14.2.3 Flow Executor



### 5.14.2.4 Flow Builder Service

