University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

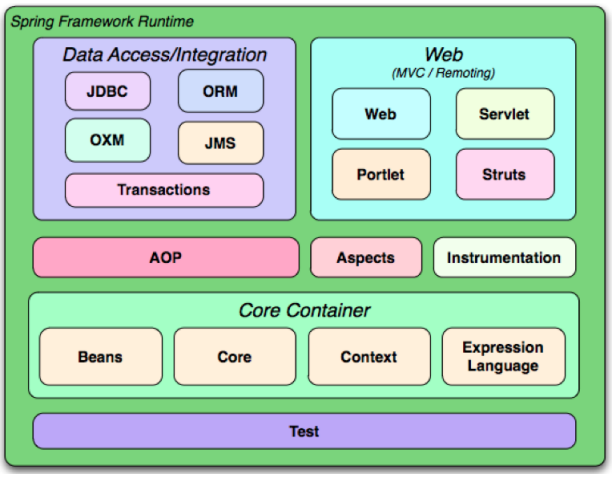
# EAF Zusammenfassung

Jonas Schwammberger, Fabio Oesch & Jan Fässler

24. November 2013

## Inhaltsverzeichnis

# 1 Spring Framework




## 1.1 Grundsätze

**Dependency Injection**
Den Objekten werden die benötigten Ressourcen und Objekte zugewiesen. Sie müssen sie nicht selbst suchen.

**Aspekt-orientierte Programmierung**
Dadurch kann man vor allem technische Aspekte wie Transaktionen oder Sicherheit isolieren und den eigentlichen Code davon frei halten.

**Templates**
Vorlagen dienen dazu, die Arbeit mit einigen Programmierschnittstellen (APIs) zu vereinfachen, indem Ressourcen automatisch aufgeräumt sowie Fehlersituationen einheitlich behandelt werden.

### 1.1.1 Dependency Injection

Dieses Paradigma beschreibt die Arbeitsweise von Frameworks: eine Funktion eines Anwendungsprogramms wird bei einer Standardbibliothek registriert und von dieser zu einem späteren Zeitpunkt aufgerufen. Statt dass die Anwendung den Kontrollfluss steuert und lediglich Standardfunktionen benutzt, wird die Steuerung der Ausführung bestimmter Unterprogramme an das Framework abgegeben.

### 1.1.2 Aspect-oriented

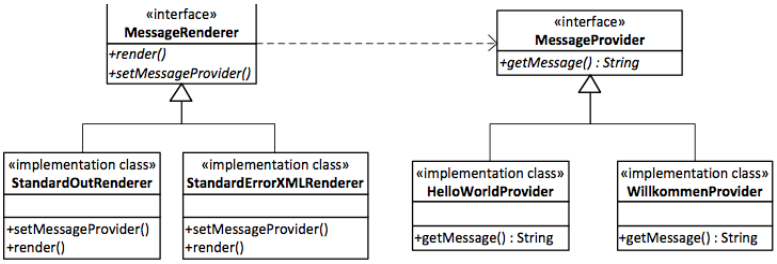
AOP ist ein Programmierparadigma, um verschiedene logische Aspekte einer Anwendung getrennt voneinander zu entwerfen, zu entwickeln und zu testen. Die getrennt entwickelten Aspekte werden dann zur endgültigen Anwendung zusammengefügt. In Enterprise Applikationen werden mit AOP vor allem System Services, wie Transaktion, Sicherheit, ... von der Business Logik entkoppelt, so dass sich der Programmierer beim Erstellen des Business Objekte vollkommen auf die Geschäftslogik und dadurch auf den eigentlichen Verwendungszweck der Applikation konzentrieren kann.

### 1.1.3 Container

Spring ist ein Container, d.h. er enthält und verwaltet den Lebenszyklus und die Konfiguration von Java-Objekten. Die Java-Objekte sind sogenannte POJOs (Plain-Old-Java- Objects). Der Spring Container kann schnell herunter- und hinaufgefahren werden. Das ermöglicht eine effiziente Entwicklung und einen gezielten Einsatz für Unit. Tests. Der Spring Container kann beispielsweise explizit für einen Unit-Test hochgefahren werden. Der Container kann problemlos im Java EE, Java SE und in Webapplikationen integriert werden. Auch setzt er keine spezielle Architektur voraus und benötigt

keine umgebungsspezifischen Konfigurationsdateien. Spring wird einfach mit einer entsprechenden Applikation mitgeliefert.

## 1.2 Configuration




### 1.2.1 XML Based

Vor/Nachteile:

+ Keine Kompilation nach Konfigurationsänderung
+ XML ist bekannt
+ Tool unterstützung vorhanden
+ Zentrales XML-File, dadurch abhängigkeiten gut ersichtlich
- Kann bei grossen Projekten unübersichtlich werden
- Implementation und Konfiguration sind von einander getrennt

Listing 1: SimpleSpringApp

```
public class DecoupledHelloWorldWithSpring {
  public static void main(String[] args) {
    BeanFactory factory = getBeanFactory();
    MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
    mr.render();
  }
  private static BeanFactory getBeanFactory() {
    return new XmlBeanFactory(new ClassPathResource("helloConfig.xml"));
  }
}
public class HelloWorldProvider implements MessageProvider {
  @Override public String getMessage() { return "Hello World"; }
}
public class StandardOutRenderer implements MessageRenderer {
  private MessageProvider provider = null;
    public void render() {
    System.out.println(provider.getMessage());
  }
  public void setMessageProvider(MessageProvider mp) {
    provider = mp;
  }
}
```

Listing 2: helloConfig.xml

```
<?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="provider" class="edu.spring.domain.provider.HelloWorldProvider" />
```

```
    <bean id="renderer" class="edu.spring.domain.renderer.StandardOutRenderer">
      <property name="messageProvider" ref="provider"/>
    </bean>
  </beans>
```

### 1.2.2 Annotation Based

Vor/Nachteile:

- \+ Implementation und Konfiguration sind zusammen
- \+ keine riesen XML
- \- Bei grossen Projekten ist es schwierig den Überblick zu behalten
- \- Der Java Quellcode muss zur Verfügung stehen

Listing 3: Annotation based

```
public class DecoupledHelloWorldWithSpring {
  public static void main(String[] args) {
    BeanFactory factory = getBeanFactory();
    MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
    mr.render();
  }
  private static BeanFactory getBeanFactory() {
    return new XmlBeanFactory(new ClassPathResource("helloConfig.xml"));
  }
}
@Component
public class HelloWorldProvider implements MessageProvider {
  public String getMessage() { return "Hello World"; }
}
@Component
public class StandardOutRenderer implements MessageRenderer {
  @Autowired
  private MessageProvider provider = null;
    public void render() {
    System.out.println(provider.getMessage());
  }
  public void setMessageProvider(MessageProvider mp) {
    provider = mp;
  }
}
```

Listing 4: helloConfig.xml

```
<?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">
  <context:annotation-config />
  <context:component-scan base-package="edu.spring" />
</beans>
```

### 1.2.3 Java Based

- \+ Implementation und Konfiguration sind zusammen
- \+ Alles in Java, kein XML
- \+ Gute Tool Unterstützung

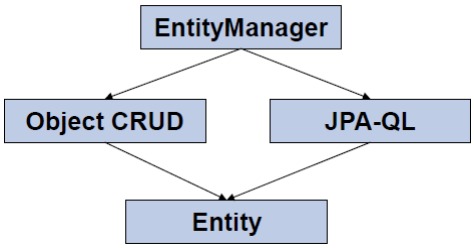
- Konfigurationsänderungen führen zur Komplilation der Konfigurationsklasse

Listing 5: JavaConfig

```
@Configuration
@ComponentScan
public class Application {
    @Bean
    MessageProvider provider() {
        return new HelloWorldProvider();
    }
  public static void main(String[] args) {
      ApplicationContext context = new AnnotationConfigApplicationContext(Application.class
          );
      MessageRenderer m = context.getBean(StandardOutRenderer.class);
      m.render();
  }
}
```

# 2 Java Persistence API (JPA)

## 2.1 General Info




**JPA Components**

- EntityManager provides access to the objects (similar to a DAO)
  - find / persist / update / remove
  - Query API and JPA-QL
- Controlled Lifecycle

**Entity Metadata**

- Form:
  - Annotations
  - XML Files
- Configuration by Exception

## 2.2 Entity Annotations

Listing 6: Entity

```
@Entity
@Table(name = "CUSTOMERS")
public class Customer implements Serializable {
        @Id
        @GeneratedValue(strategy=GenerationType.IDENTITY)
        private int id;
        private String firstName;
        @Column(name="NAME")
        private String lastName;
        protected Customer(){}
        public Customer(String firstName, String lastName){
                this.firstName = firstName;
                this.lastName = lastName;
                // id is not set!
        }
        public int getId() { return this.id; } // read only
        public String getFirstName() { return this.firstName; }
        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }
        public String getLastName() { return this.lastName; }
        public void setLastName(String lastName) {
                this.lastName = lastName;
        }
}
```

Folie 10 - 19, Foliensatz JPA1.pdf sind Spezifikationen und Anforderungen an Entity Klasse

### 2.2.1 Primary Keys: Generation

- Assigned
  - Primary keys may be assigned by application, i.e. no key generation
    - E.g. language table: Primary Key is the ISO country code
- Identity
  - Auto increment supported by some DBs
- Sequence
  - Some DBs support sequences which generate unique values (e.g. Oracle, PostgreSql)
- Table
  - Primary keys are stored in a separate PK table

**Performance Comparison**

- 10'000 insert statements
- AUTO (Identity) 7534 msec
- TABLE (allocationSize = 32768) 2244 msec
- TABLE (allocationSize = 1) 9612 msec
- TABLE (allocationSize = 2) 7429 msec
- TABLE (allocationSize = 4) 5856 msec
- ASSIGNED (user defined) 1959 msec

### 2.2.2 Associations

- OneToOne, owning side contains the foreign key
- OneToMany
- ManyToOne
- ManyToMany, either side may be the owning side
- Owning side determines the updates to the relationships in the database

**Relationships can be:**

- Unidirectional
  - Has an owning side
- Bidirectional
  - Has an owning side
  - Has an inverse side

**ManyToOne: User - Rental: bidirectional**
Bei bidirektionalen Beziehungen sind die Many-Side die owning Side.
Example:

Listing 7: Example

```
@Entity
public class Rental implements Serializable { @Id
        private int id;
        @ManyToOne // Rental is the owner of the relationship
        @JoinColumn(name="USER_FK") // optional
        // JPA macht bei OneToMany und ManyToOne eine Zwischentable.
```

```
        // Mit Keyword JoinColumn wird dies unterbunden. => Foreign Key = Owning side
        private User user;
        public Rental(){}
        public user getUser() { return user; }
        public void setUser (Customer user) {
                this.user = user;
        }
        public int getId() { return id; }
        public void setId(int id) { this.id = id; }
}
```

Inverse Side Example:

Listing 8: InverseSideExample

```
@Entity
public class User implements Serializable {
        ...
        @OneToMany(mappedBy="user") private Collection<Rental> rentals;
        // this is the inverse side of the relationship
        public Collection<Rental> getRentals() {
                return rentals;
        }
        public void setRentals(Collection<Rental> rentals) {
                this.rentals = rentals;
        }
}
```

**Only references from $n$ to $1$ are persisted!**

Listing 9: OneToMany bidirectional

```
em.getTransaction().begin();
Customer c = new Customer();
Order o1 = new Order();
Order o2 = new Order();
List<Order> orders = new LinkedList<Order>();
orders.add(o1); orders.add(o2);
c.setOrders(orders);
em.persist(c);
em.getTransaction().commit();
```

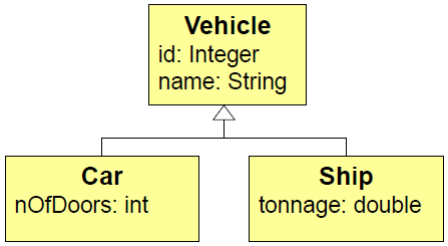
- the two orders are stored in the DB (due to the cascade=PERSIST)
- the associations are NOT persisted!!!

**OneToOne / OneToMany / ManyToOne / ManyToMany - Attributes**

- fetch EAGER / LAZY
  - determines fetch type
- cascade MERGE / PERSIST / REFRESH / DETACH REMOVE / ALL
  - determines cascade operation
- mappedBy String, not for ManyToOne
  - used for bidirectional associations (on the inverse side)
- optional boolean, only for OneToOne/ManyToOne
  - determines, whether null is possible (0..1)

- orphanRemoval boolean, only for OneToOne/OneToMany

**Example:**
@OneToOne(cascade=CascadeType.PERSIST, CascadeType.REMOVE)
Slides 20 & 21, Foliensatz JPA2_ Slides.pdf Cascading und Fetch Types werden erklärt
Im Zusammenhang mit Fetch Types ist Lazy Loading Problem erklärt im Foliensatz 00_ JPA2_ Arbeitsblatt_ Besprechung.pdf **Inheritance**




### 2.2.3 Examples

Listing 10: Example

```
@Entity @Table(name="EMP")
public class Employee {
        public enum Type {FULL, PART_TIME};
        protected Employee(){}
        public Employee(String name, Type type){
                this.name = name; this.type = type;
        }
        @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
        long id;
        @Enumerated(EnumType.STRING)
        @Column(name="EMP_TYPE", nullable=false)
        Type type;
        @Lob byte[] picture;
        String name;
}

create table EMP (
        id bigint generated by default as identity (start with 1),
        picture longvarbinary,
        EMP_TYPE varchar(255) not null,
        primary key (id)
)
```

- Representation
  - SINGLE TABLE (default)
  - TABLE_ PER_ CLASS (per concrete class a table is defined)
  - JOINED (one table per class)
- Specification
  - Inheritance type can be specified on root entity using @Inheritance annotation

**Single Table Example (@Inheritance(strategy=InheritanceType.SINGLE_TABLE))**

| DTYPE | ID | NAME | NOFDOORS | TONNAGE |
|---|---|---|---|---|
| Car | 1 | VW Sharan | 5 | (null) |
| Car | 2 | Smart | 2 | (null) |
| Ship | 3 | Queen Mary | (null) | 76000 |

**Disadvantages**
- All fields added in subclasses must be nullable

• Foreign keys can only refer to the base class

**Joined Table Example (@Inheritance(strategy=InheritanceType.JOINED))**

| ID | NAME |
|---|---|
| 1 | VW Sharan |
| 2 | Smart |
| 3 | Queen Mary |

| ID | NOFDOORS |
|---|---|
| 1 | 5 |
| 2 | 2 |

| ID | TONNAGE |
|---|---|
| 3 | 76000 |

- Advantages:
  - •normalized schema, a database table for each class
  - All fields can be defined with not null conditions
  - Foreign-key references to concrete subclasses are possible
- Disadvantages:
  - Each entity access has to go over several tables

**TABLE_ PER_ CLASS Table Example (@Inheritance(strategy=InheritanceType.TABLE_ PER_ CLASS))**

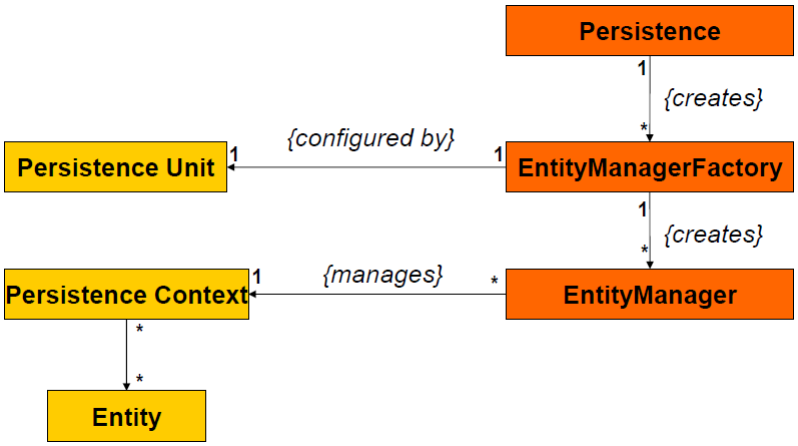
| ID | NAME | NOFDOORS |
|---|---|---|
| 1 | VW Sharan | 5 |
| 2 | Smart | 2 |

| ID | NAME | TONNAGE |
|---|---|---|
| 3 | Queen Mary | 76000 |

- Advantages:
  - Non-null constraints can be defined
  - Foreign-key references to concrete subclasses are possible (but not to abstract base classes)
- Disadvantages:
  - Polymorphic queries need to access several tables
  - Identity generator cannot be used
  - Not required by JPA 2.0-Spec (but provided by Hibernate)

## 2.3 Entity Manager

Entity Manager API auf Seite 8-11, Foliensatz JPA2_ Slides.pdf




**Only one Java instance with the same persistent identity may exist in a Persistence Context**

Listing 11: Access to Entity Manager

```
// J2SE: using factory
EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("movierental");
EntityManager em = emf.createEntityManager();

// J2EE: injected by container
@PersistenceUnit(name="movierental")
EntityManagerFactory emf = null;
```

```
@PersistenceContext(unitName="movierental")
private EntityManager em;
// => container managed entity manager
```

#### 2.3.1 Queries

- JPQL
  - Used to query/manipulate database
  - Inspired by SQL, but it operates directly on the entities and its fields
- Statements
  - select_ statement ::= select_ clause from_ clause [where_ clause] [groupby_ clause] [having_ clause] [orderby_ clause]
  - update_ statement ::= update_ clause [where_ clause]
  - delete_ statement ::= delete_ clause [where_ clause]

### 2.4 Transactions

**Access to the EntityManager must run within a transaction**

### 2.5 Data Transfer Object

- Detached Entity objects as DTOs
  - Hibernate developers say that you can use hibernate entity or domain objects as result types in service methods
  - Problems:
    - Lazy load exceptions are thrown if "not-loaded"fields are accessed
    - Having an accessor which does throw an exception is contract violating
    - Accessing the type of a result using reflection returns a proxy type (which is not serializable)
- Data Transfer Objects
  - Are used to transfer data across layers of your application
  - Only the data needed by the requesting layer are passed, i.e. not all properties need to be
  - No Lazy Loading Exception surprises
  - Clients are independent of ORM technology used

#### 2.5.1 Service Method (Dozer)

- Dozer is a Java Bean to Java Bean mapper that recursively copies data from one object to another $\Rightarrow$ can be used to copy DTO
- Dozer supports simple property mapping, complex type mapping, bi-directional mapping, implicit-explicit mapping, as well as recursive mapping. This includes mapping collection attributes that also need mapping at the element level

#### 2.5.2 Con

- Code Duplication
  - In particular when DTOs have the same fields as domain objects
- Code to copy attributes back and forth
  - Dozer / Spring BeanUtils / JPA

### 2.5.3 Pro

- Lazy Loading Problem
  - You are not catched by a Lazy Loading Exception
    - neither on client side
    - nor upon serialization
- Triggers Design
  - Forces you to think about the interface of the remote service façades
  - Information from multiple domain objects can be combined in one DTO

# 3 Spring Remoting

## 3.1 Prüfung

**Facade:** Zuständig für abstraktion von Service Layer und benutzt am besten DTO's

**Service Layer:** Service Schnittstellen stellt Business Cases dar

**Domain Model:** Entities

**Database Access:** Wird von JPA Implementation übernommen

# 4 Transaktionen

## 4.1 Transaction Isolation Strategies

- Read Uncommitted: Geringste Isolation, höchste Performance, es können Dirty Reads, Non-repeatable Reads und Phantom Reads auftreten
- Read Committed: Es gibt keine Dirty Reads mehr, aber es gibt weiterhin Non-repeatable Reads und Phantom Reads
- Repeatable Read:Keine Dirty Reads und keine Non-repeatable Reads, aber weiterhin Phantom Reads
- Serializable: Keine Dirty Reads, keine Non-repeatable Reads und keine Phantom Reads, höchste Isolation, geringste Performance

Default Isolation level????

## 4.2 Transaction PropagationStartegies

n einigen Transaktionsmanagern kann die ”Transaction Propagation”pro Methode unterschiedlich eingestellt werden. Es sind nicht immer alle Einstellungen möglich. Die unterschiedlichen Einstellungen zur Transaction Propagation bewirken beim Eintritt in die jeweilige Methode Folgendes:

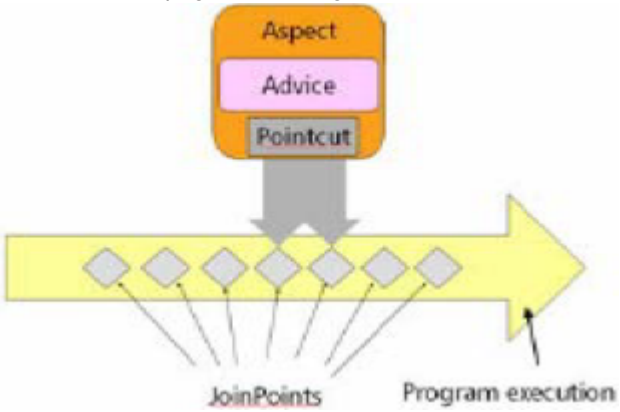
- Required: Falls bereits vorher eine Transaktion begonnen wurde, wird sie fortgesetzt. Falls noch keine Transaktion aktiv ist, wird eine neue gestartet.
- RequiresNew: Unabhängig davon, ob bereits eine Transaktion aktiv ist, wird immer eine neue eigene Transaktion gestartet. Diese Transaktion benötigt ein eigenes Commit bzw. Rollback. Ein Commit bzw. Rollback in dieser neuen Transaktion führt nicht zum Commit bzw. Rollback in einer eventuell vorher begonnenen Transaktion. RequiresNew bedingt keine 'Nested Transaction', sondern ein 'Suspend' der laufenden Transaktion und Einschieben der Untertransaktion. Unabhängig vom Ergebnis dieser Untertransaktion wird anschließend mit der übergeordneten Transaktion fortgefahren.
- Supports: Falls bereits eine Transaktion aktiv ist, wird sie verwendet. Ansonsten wird keine Transaktion verwendet.
- NotSupported: Die Methode wird immer ohne Transaktion ausgeführt, auch wenn bereits vorher eine Transaktion gestartet wurde.
- Mandatory: Es muss bereits eine Transaktion aktiv sein. Sonst wird eine Exception geworfen.
- Never: Es darf keine Transaktion aktiv sein. Sonst wird eine Exception geworfen.
- Nested: Eine geschachtelte Transaktion wird gestartet. Diese Option wird meistens nicht unterstützt.

WICHTIG: In JPA müssen die meisten Manipulationen an den Entitäten eines PersistenceContext von einer JPA-Transaktion eingehüllt werden. Das betrifft praktisch alle ßchreibendenËntityManager-Methoden (persist, flush, remove, merge, lock und refresh).

# 5 Aspect-Oriented Programming

Aspekte sind Filter, welche vor und nach dem Aufruf anderer Methoden eingehängt wird.

Transaction Propagation Strategie wird im Movierental ebenfalls über einen Aspect implementiert.




- Advice: Action taken at a particular joinpoint.
- Join Point: Point during the execution of execution.
- Pointcut: A set of joinpoints specifying where advice should be applied.
- Aspect: A modularization of a cross-cutting concern. The combination of advice and pointcut.
- Weaving: Assembling aspects into advised objects.

## 5.1 Implementationssmöglichkeiten

- Compile time – modify the source code during compilation
- Run time – byte injection, benutzt Library cglib
- Run time – using the JDK 1.3 Dynamic Proxy. Instead of getting an object instance, the application receives a proxy object.The proxy implements the same interface.

Spring benutzt Proxies.

## 5.2 AOP in Spring

Listing 12: Enable annotated Aspects

```
<!-- enabling @AspectJ -->
<aop:aspectj-autoproxy/>
<!-- annotated aspect as regular java class -->
<bean id="tracing" class="aop.TracingAnnotations"/>
```

Listing 13: Enable annotated Aspects

```
@Aspect
@Component // add <component-scan>
public class TracingAnnotations {
        @Autowired
        private Statistic statistic;

        //execution(...) = point cut. Or simply what method name pattern
```

```
        //it should be used upon.
        @Before("execution(* edu.GreetingService.say*())")
        public void trace() {
                // do something with the statistic bean
        }
}
```

**Types of Advice**, wie dieser Aspect eine Methode umhüllt.

- Before: Vor einem Methodenaufruf
- After: Nach einem Methodenaufruf, auch wenn eine Exception geworfen wurde.
- After Returning: Nach einem Methodenaufruf normal zurückkommt.
- Around: Alles

**Pointcut**, wann eine Methode umhüllt wird. Execution ist das wichtigste (arbeitet mit einem Pattern Matching im Methodennamen).

- execution - for matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP
- within - limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)
- this - limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type
- target - limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type
- args - limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types
- @target - limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type
- @args - limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given type(s)
- @within - limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
- @annotation- limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation

## 5.3 Pros

- ...reduces the amount of codes
- ...enhancing maintainability
- ...flexible; aspects, point-cuts and advices can be defined either at source code level, or in XML metadata.

## 5.4 Cons

- AOP suffers from a lack of tool support and widespread education
- easy to introduce unpredictable behavior
- There are security concerns with code weaving, and hence some AOP implementation does not support bytecode weaving, including Spring (Spring only support runtime weaving).

# 6 Anhang

Listing 14: Persistence Unit

```
<persistence>
        <persistence-unit name="movierental"
                        transaction-type="RESOURCE_LOCAL">
                <class>ch.fhnw.edu.rental.model.Movie</class>
                ...
                <properties>
                <property name="hibernate.connection.driver_class"
                        value="org.hsqldb.jdbcDriver" />
                <property name="hibernate.connection.url"
                        value="jdbc:hsqldb:hsql://localhost/lab-db" />
                <property name="hibernate.connection.username"
                        value="sa" />
                <property name="hibernate.connection.password"
                        value="" />
                </properties>
        </persistence-unit>
</persistence>
```

Listing 15: Query Examples

```
TypedQuery<Movie> q = em.createQuery(
        "select m from Movie m where m.title = :title",
Movie.class);
q.setParameter("title", title);
List<Movie> movies = q.getResultList();

@NamedQueries({
        @NamedQuery(name="movie.all", query="from Movie"),
        @NamedQuery(name="movie.byTitle",
                query="select m from Movie m where m.title = :title")
})
class Movie {...}

TypedQuery<Movie> q = em.createNamedQuery(
        "movie.byTitle", Movie.class);
q.setParameter("title", title);
List<Movie> movies = q.getResultList();

SELECT c FROM Customer c WHERE c.address.city = 'Basel'
SELECT c.name, c.prename FROM Customer c
SELECT DISTINCT c.address.city FROM Customer c
SELECT NEW ch.fhnw.edu.Person(c.name,c.prename) FROM Customer c
SELECT pk FROM PriceCategory pk

TypedQuery<Movie> q = em.createQuery(
        "select m from Movie m order by m.name", Movie.class);
q.setFirstResult(20);
q.setMaxResults(10);
List<Movie> movies = q.getResultList();

Query q = em.createQuery(
        "delete from Movie m where m.id > 1000");
int result = q.executeUpdate();
```