Enterprise Application Frameworks

Roland Hediger

24. November 2013

Inhaltsverzeichnis

Teil I.

Theorie

1. Einführung in Spring Framework

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

Wann werden Abhängigkeiten nötig? Der Begriff Dependency Injection (DI) bezeichnet ein Umsetzungsparadigma, das in der objektorientierten Programmierung Anwendung findet. Dieses Paradigma beschreibt die Arbeitsweise von Frameworks: eine Funktion eines Anwendungsprogramms wird bei einer Standardbibliothek registriert und von dieser zu einem späteren Zeitpunkt aufgerufen. Das wird manchmal als eine Anwendung des Hollywood-Prinzips bezeichnet:don't call us, we'll call you (zu Deutsch: Rufen Sie uns nicht an, wir werden Sie anrufen): Statt dass die Anwendung den Kontrollfluss steuert und lediglich Standardfunktionen benutzt, wird die Steuerung der Ausführung bestimmter Unterprogramme an das Framework abgegeben

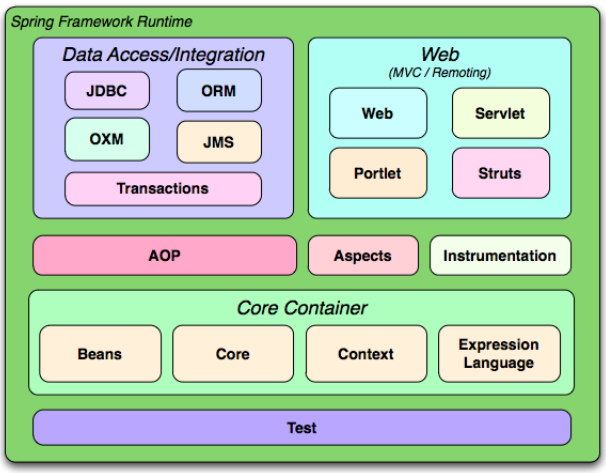
Aspect Orientated Programmierung (AOP): Dadurch kann man vor allem technische Aspekte wie Transaktionen oder Sicherheit isolieren und den eigentlichen Code davon frei halten.

Spring unterstützt die aspektorientierte Programmierung (AOP). AOP ist ein Programmierparadigma, um verschiedene logische Aspekte eines Anwendungsprogramms (kurz 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

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.

Framework Spring erlaubt es eine komplexe Applikation aus einfacheren Komponenten zusammenzubauen. Das Spring Framework stellt dabei System Services wie Transaktionsmanagement, Persistenzframework, etc. zur Verfügung, so dass sich der Entwickler möglichst auf die Business Logik konzentrieren kann.

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



1.1. Classpath (Spring HelloWorld)

- JRE + MAVEN + Projekt
- `src/main/java , src/main/res, src/test/java src/test/res`
- kann es auch Buildpath nennen.

```
localRepository setzen. Nutzt man in STS das Maven-Plugin müssen die entsprechenden
Einstellungen in der Konfiguration des Plugins vorgenommen werden.
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
<localRepository>d:/projects-dev/repository</localRepository>
</settings>
Listing 1: Eintrag in Maven File settings.xml um lokal den Ort des Repo zu setzen
```

1.2. Einfache Entkoppelung mittels Dependancy Injection

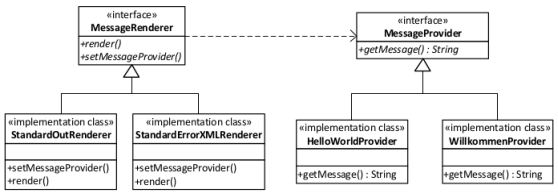


Abbildung 1.1.: Herleitung

1.2.1. Ausgangspunkt

Listing 1.1: Ausgangspunkt

```
1  public class HelloWorld {
    public static void main(String[] args) {
      System.out.println(  Hello   World!");
    }
  }
6   //Der erste Schritt kann folgendermassen aussehen:
   public class DecoupledHelloWorld {
    public static void main(String[] aargh) {
      MessageRenderer mr = new StandardOutRenderer();
      MessageProvider mp = new HelloWorldProvider();
11    mr.setMessageProvider(mp);
      mr.render();
      }
  }
```

1.2.2. Versuch 1

Das Design kann nun durch den Einsatz des Abstract Factory Design Pattern verbessert werden. Mit der Factory können die konkreten Implementationen "by nameërzeugt werden, wobei die Namen über ein Konfigurationsfile festgelegt werden. Hier ein Auszug aus einer möglichen Implementation:

Listing 1.2: Entkoppelung mit manuelles Setzten von Abhängigkeiten

```
1  public class DecoupledHelloWorldWithFactory {
    public static void main(String[] aargh) {
    MessageRenderer mr =
    MessageSupportFactory.getInstance().getMessageRenderer();
    MessageProvider mp =
```

```
6     MessageSupportFactory.getInstance().getMessageProvider();
      mr.setMessageProvider(mp);
      mr.render();
      }
  }
11
  private MessageSupportFactory() {
  ...
  // Read configuration file
  bundle=ResourceBundle.getBundle("ch.edu.msgConf");
16 // Get Renderer from the configuration
  String rendererClass=bundle.getString("renderer.class");
  // Get Provider from the configuration
  String providerClass=bundle.getString("provider.class");
  try {
21 renderer=(MessageRenderer)
  Class.forName(rendererClass).newInstance();
  provider=(MessageProvider)
  Class.forName(providerClass).newInstance();
  } catch (InstantiationException e) {
26 //exception handling code
  } catch (IllegalAccessException e) {
  //exception handling code
  } catch (ClassNotFoundException e) {
  //exception handling code
31 }
  ...
  }
```

1.2.3. Völlig entkoppelt

Grundsätzlich schöner wäre jedoch folgender Code: DecoupledHelloWorldWithSpring ist nun Spring Applikation, die über eine sogenannte BeanFactory das Bean mit dem Namen rendereräusliest, um anschliessend auf dieser Instanz die render() Methode auszuführen. In der Methode getBeanFactory() ist der Zugriff auf die Spring BeanFactory gekapselt. Sie wird in unserem Falle folgendermassen aussehen: In dieser Methode wird die Spring-XMLBeanFactory erzeugt. Diese Factory ist in der örg.springframework.beans-X.X"jar-Bibliothek abgelegt. Die Factory liest das Spring Konfigurationsfile "helloConfig.xmlünd wird die darin enthaltenen Spring Beans instanziieren und allfällige Abhängigkeiten über Dependency Injection auflösen. Das Spring Konfigurationsfile ist zentral und legt die konkreten Implementationen fest, die in der Applikation genutzt werden sollen. In diesem Beispiel kann es folgendermassen aussehen:

Listing 1.3: Depedancy Injection with Spring

```
  public class DecoupledHelloWorldWithSpring {
2     public static void main(String[] args) {
        BeanFactory factory = getBeanFactory();
        MessageRenderer mr = (MessageRenderer) factory.getBean("renderer");
        mr.render();
    }
7 }

  private static BeanFactory getBeanFactory() {
  XmlBeanFactory factory = new XmlBeanFactory(
  new ClassPathResource("/spring/helloConfig.xml"));
12 return factory;
  }
```

Listing 1.4: Spring Config Datei

```
  ?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
#1
5 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">
<bean id="renderer" class="edu.spring.domain.renderer.StandardOutRenderer"> #2
<property name="messageProvider" ref=    provider    /> #3
    </bean>
10 <bean id="provider" class="edu.spring.domain.provider.HelloWorldProvider" /> #4
</beans>
```

1.2.4. Besonderheiten

IDs sind optional im Bean file.

Es ist möglich Beans über Klassennamen zu hohlen. Aber muss eine konkrete Implementation nehmen.

Circular References a- b -c in xml file.Funktioniert. *Application Context empfohlen*. Instancen gemacht und dann gesetzt mit Setter. Was wenn wir eine Abhängigkeiten mit Injection im Kontruktor? Geht nicht - Arbeitsblatt 3. **Defaultmässig Injection per Setter gemacht**

2. Spring Konfiguration

2.1. Intro

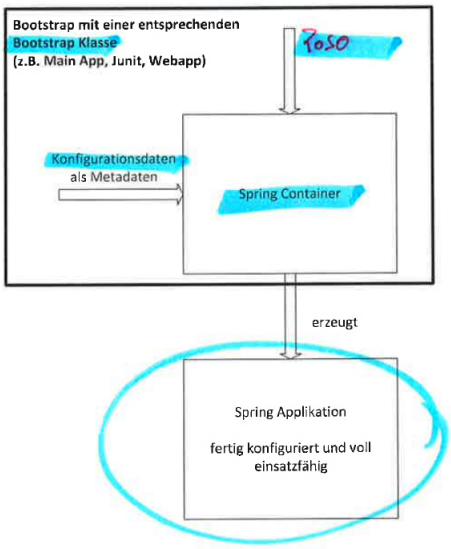


Abbildung 2.1.: Grundlagen der SpringConfig

Als Spring Container stehen zwei Ausprägungen im Zuentrum :

BeanFactory: ist der Container für die Spring Beans. Er stellt den grundlegenden Mechanismus für die Verwaltung der Beans zur Verfügung.

ApplicationContext: Erweitert die BeanFactory mit zusätzlicher Funktionalität:

- AOP Features
- Message Resource Handling (Internationalizierung)
- Even Publication

Wegen diese Features ist die ApplicationContext vvariante bevorzügt.

Die AplicationContext ist ein Interface. Es existiert folgende verschiedene Implementationen davon :

ClassPathXMLApplicationContext XML File über Classpath geladen.

FileSystemXMLApplicationContext XML File über Dateisystem geladen.

XmlWebApplicationContext XML Datei über Web Application Context geladen.

Werte von Bean Properties können in Property-Files ausgelagert werden für zusätzliche Flexibilität.[1] Properties sollen *nach Instanzierung von Beans* folgen. Deshalb wird dies in Spring Framework als **BeanFactoryPostProcessor** eingeführt. Implementationen davon sind : PropertyOverideConfigurer und PropertyPlaceHolderConfigurer. Wobei das zweite öfter zum Einsatz kommt.

Listing 2.1: property-placeholder Eintrag

```
<context:property_placeholder location="classpath:app.properties"/>
```

Element	Beschreibung
Context	Spring Schema
property-placeholder	Spring BeanFactory
location	Attribut
".."	Wert

[1] Ein Property-File ist einfach ein Key-Value Store : muster.eigenschaft=bla

Listing 2.2: Einsatzbeispiele von property-placeholder

```
    <bean id="provider"
     class="edu...HelloWorldMessageProvider">
     <property name="message" value="${helloworld.message}"/>
4
    <!-- JAVA DB Beispiel Property File --!>
    jdbc.driverClassName=org.hsqldb.jdbcDriver
   jdbc.local.url=jdbc:hsqldb:hsql://localhost/build/movierental
9  jdbc.memory.url=jdbc:hsqldb:mem:movierental
   jdbc.standalone.url=jdbc:hsqldb:file:build/movierental
   jdbc.url=${jdbc.memory.url}
   jdbc.username=sa
   jdbc.password=...
```

Bemerkung er Konstruktor eines ApplicationContext kann immer auch mehrere Config Locations verarbeiten. Dies ist wichtig, da damit auch auf der Konfigurationsseite eine Strukturierung z.B. entlang der Server Layers möglich wird. [2]

Die XML-Schema basierte Konfiguration wurde mit Spring 2.0 eingeführt und in den folgenden Versionen immer weiter ausgebaut. Es ersetzt die DTD Version.

Inzwischen gibt es verschiedene Schemas, um die Konfiguration zu vereinfachen. Es sind dies util, jee, lang, jms, tx, aop,context, tool, beans. Mehr zu diesen Schemas finden sie im Appendix C der Spring Reference Documentation.

2.2. Setter Injection vs Constructor Injection

Listing 2.3: Die zwei Arten von Injection

```
1   <bean id="renderer" class="edu.spring.domain.renderer.StandardOutRenderer">
   <property name="messageProvider" ref="provider"/>
   </bean>
   <bean id="provider" class="edu.spring.domain.provider.DummyMessageProvider">
   <constructor-arg name="message" value="Dummy Message"/>
6  </bean>
```

Variante	Vorteil	Nachteil
Setter-Injection	• Optionale Properties einfach gesetzt. • Leichte Handhabung der Properties.	• Pflicht Properties mussen mit Required annotiert werden.
Constructor-Injection	• Mandatory Properties über Konstruktor setzen • Keine Intanzen die nicht vollständig initializiert sind	• Unübersichtlichen Code mit viele Konstruktor-Argumente.

2.3. Konfiguration über XML hinaus

Traditionell wird ein ApplicationContext über eine XML Datei konfiguriert. Die XML-basierte Konfiguration ist aber nicht die einzige Möglichkeit den Spring Container zu konfigurieren. Da der Container komplett vom Format der Metadaten entkoppelt ist, stehen dem Programmierer weitere Möglichkeiten offen, wie:

- Annotationsbasiert : seit 2.5
- Javabasiert : seit 3.0

[2] Konstruktor nimmt String Array von xml Dateien

XML-basiert	• Keine Kompilation nach Konf Änderungen notwendig. • XML ist bekannt. • Tool Unterstützung ist vorhanden. • Mit Zentralem XML Datei sind die Beans und Äbhangigkeiten dazwischen einfach ersichtlich	• Überblick verloren bei grossen Projekten. (Mehrere verschiedene XML files) • Kann unübersichtlich sein mit Implementation und Konfiguration getrennt.
Annotationsbasiert	• Implementation und Konfiguration zusammen, besserer Überblick. • Keine "XML Hölle"	• Konfigurationen sind über alle Klassen vertraut. Ist schwierig den Überblick zu behalten. • Java Quellcode muss zur Verfügung stehen um Konfigurationsänderungen vorzunehmen
Javabasiert	• Kein XML notwendig. • Implementation und Konfiguration sauber getrennt. • Gute Toolunterstützung.	• Konfigurationsänderungen führen zu neue Kompilation der Klasse.

2.3.1. Implementation von Annotationsbasierte Konfiguration

Listing 2.4: Annotationsbasierte Konfiguration

```
@Component // 1
public class StandardOutRenderer implements MessageRenderer {
  @Autowired
  private MessageProvider messageProvider;
  @Override // 2
  public void setMessageProvider(MessageProvider mp) {
    this.messageProvider = mp;
  }
  @Override
  public void render() {
    System.out.println(messageProvider.getMessage());;
  }
}
```

- Klasse wird als SpringBean konfiguriert. Ich kann der Bean eine Id geben mit entweder @Qualifier oder @Component("name")
- Hier wird der BeanProperty gesucht vom Typ MessageProvider. Kann auch hier mehr spezifizieren mit @Qualifier

Bemerkung: BeanNameGenerator. Wenn keine Name spezifiziert ist wird automatisch die Klassenname genommen mit folgendem Format: MyClassName → myClassName (ist der Bean ID)

2.3.2. Scanning für Annotationen in den Klassen

Listing 2.5: Annotation Scanning aktivieren"

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

context:annotation-config Annotation unterstützung für Dependancy Injection aktivieren. (Erkennung von @Autowired)

context:component-scan Erweitert annotation-config sodass Beans über Annotations deklariert werden können.

2.4. Spring + Testing

Listing 2.6: Testing Example

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({"/spring/helloConfig.xml"})
public class HelloWorldMessageProviderAnnotationTest {
  @Autowired
  private MessageProvider messageProvider;
  @Test
  public void testGetMessage() {
  assertEquals("Hello World!", messageProvider.getMessage());
    }
}
```

RunWith Spring Spezifische JUnit Test Runner Class benutzt. Resourcen wie config file sind nur innerhalb des gegebenen Pakets zu finden.

ContextConfiguration Angabe des Spring Config files.

3. Datenbank

3.1. Data Acess Object Pattern

- DAO sind benutzt um Encapsulation zu erreichen.
- 1 DAO 1 Entity.
- Nicht für Transaktion, Session oder Verbindung verantwortlich.

Listing 3.1: GenericDAO

```
public interface GenericDao <T, PK extends Serializable> {
PK create(T newInstance);
  T getById(PK id);
  List<T> getAll();
  void update(T obj);
  void delete(T obj);
}
```

Separiert Persistenz von Business Logik.

Strategy Pattern Nicht 100% möglich. DAO beinflusst benutzung - nicht mehr POJO. Object Lifecycles (Managed), JDBC : Explicit Update Opartations.

3.1.1. Motivation

- Encapsulation
- Testability : DAO leichter bei Mock als hibernate Entity Manager
- Vendor unabhängigkeit : Abstrahiert verschiedene Implementationen von den gleichen DAO Typ (Hibernate, JDO)

Ausnahme: Falls Business Logik mehrheitlich aus DB Zugriff besteht, dann nutzt diese Trennung nichts.

3.2. Service Layer

Aufgaben:

- Core API für andere Schichten der Applikation : Facade Pattern
- Core Business Logik : Besteht aus mehrere Aufrufe an DAOs.

  Movie züruckgegeben oder Servicemethoden für DAO Methoden.
- Combines methods defined in the DAOs and assembles them to cohesive business methods that define an atomic unit of work

  Transaktionale Semantik (CRUD)

  Mit Spring AOP realistiert.

3.3. JDBC Problems

Wenn mann Verbindungen offen lässt, hat man evtl. keine Verbindungen mehr.

Listing 3.2: Verbindung offen Beispiel : JDBC

```
Connection conn = null;
    try {
      conn = ds.getConnection();
      ...
    } catch(SqlException e) { ... }
    } finally {
    if(conn != null){
    try {
      conn.close();
    } catch (SQLException e) {
    throw new RuntimeException(e);
    }
  }
}
```

3.4. JDBC Probleme

Gelöst mit Java 7:

Listing 3.3: Java7 JDBC solution

```
 Connection conn = null;
try (Connection conn = ds.getConnection()){
...
} catch(SqlException e) { ... }

public interface AutoCloseable {
void close() throws Exception;
}
```

Exceptions thrown while closing the resource may be shadowed by the exceptions thrown while using the resource.

3.4.1. JDBC pure DAO

Listing 3.4: JDBC DAO

```
public Movie getById(Long id) {
  Connection conn = null;
  try {
    conn = ds.getConnection();
    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery(
    "select * from MOVIES where MOVIE_ID = "+id);
    Movie m = null;
    if(rs.next()) {
    long priceCategory = rs.getLong("PRICECATEGORY_FK");
    m = new Movie(rs.getLong("MOVIE_ID"),
    rs.getString("MOVIE_TITLE"),
    rs.getDate("MOVIE_RELEASEDATE"),
    rs.getBoolean("MOVIE_RENTED"),
    priceCategoryDAO.getById(priceCategory));
    }
  return m;
  } catch (SQLException e) {
    throw new RuntimeException(e);
  }
  finally {
    if(conn != null){
  try {
    conn.close();
      } catch (SQLException e) {
    throw new RuntimeException(e);
  }
      }
    }
  }
```

Listing 3.5: PureJDBC Java 7

```
public Movie getById(Long id) {
   try (Connection conn = ds.getConnection()){
   Statement st = conn.createStatement();
   ResultSet rs = st.executeQuery(
   "select * from MOVIES where MOVIE_ID = "+id);
   if(rs.next()){
     long priceCategory = rs.getLong("PRICECATEGORY_FK");
     return new Movie(rs.getLong("MOVIE_ID"),
     rs.getString("MOVIE_TITLE"),
     rs.getDate("MOVIE_RELEASEDATE"),
     rs.getBoolean("MOVIE_RENTED"),
     priceCategoryDAO.getById(priceCategory));
     } else { return null;}
   } catch (SQLException e) {
     throw new RuntimeException(e);
   }
}
```

Probleme: Schreiben von Portable selbstdokumentierte Code der auf spezifischen Fehler reagiert ist problematisch.

Spring Support: Template um redundante Code zu reduzieren :

- DAOs mit DI
- Automatische Verbindungsverwaltung (Öffenen und Schliessen von Resourcen)

Konvertiert auch Exceptions von Frameworks zu eine Hirachie.

3.5. Spring Template Pattern

- Öffnen und Schliessen von Verbindungen
- Transaktionsverwaltung
- SQL Operationen in try catch Blöcke ausführen.
- Transaktion Commit und Rollback
- Resourcenverwaltung
- Exceptions fangen.

Template Typen:

JdbcTemplate Generics, Autoboxing, varargd

NamedParameterJdbcTemplate Namedparameter, statt Placeholderargumente. Lesbar, und leichter bei Wartung.

Template Erzeugung:

- JdbcTemplate(DataSource ds) constructor
  - Template may be stored in the DAO class (connections are created when needed)
- getJdbcTemplate() in extensions of JdbcDaoSupport
  - DataSource is injected by Spring
  - getJdbcTemplate is typically called in each DAO method
- getNamedParameterJdbcTemplate() in NamedParameterJdbcDaoSupport

Abbildung 3.1.: figure

Listing 3.6: JdbcDaoSupport

```
public abstract class JdbcDaoSupport extends DaoSupport {
  private JdbcTemplate jdbcTemplate;
  public final void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = createJdbcTemplate(dataSource);
```

```
    initTemplateConfig();
  }
  protected JdbcTemplate createJdbcTemplate(DataSource ds) {
    return new JdbcTemplate(ds);
  }
  public final JdbcTemplate getJdbcTemplate() {
    return this.jdbcTemplate;
  }
    protected void initTemplateConfig() { }
    ...
}
```

3.5.1. JDBC Template : Query Results

Listing 3.7: JDBC Query Results

```
public Movie getById(Long id){
    JdbcTemplate template = new JdbcTemplate(getDataSource());
    Map<String, Object> res = template.queryForMap(
    "select * from MOVIES where MOVIE_ID = ?", id);
    long priceCategory = (Long)res.get("PRICECATEGORY_FK");
    Movie m = new Movie(
    (Long)res.get("MOVIE_ID"),
    (String)res.get("MOVIE_TITLE"),
    (java.sql.Timestamp)res.get("MOVIE_RELEASEDATE"),
    ((Boolean)res.get("MOVIE_RENTED"),
    priceCategoryDAO.getById(priceCategory));
    return m;
  }
```

Listing 3.8: JDBC Query Results (Named Parameters

```
public Movie getById2(Long id) {
    NamedParameterJdbcTemplate template =
    new NamedParameterJdbcTemplate(getDataSource());
    String query = "select * from MOVIES where MOVIE_ID = :id";
    Map<String, Long> params = new HashMap<>();
    params.put("id", id);
    Map<String, Object> res=template.queryForMap(query, params);
    long priceCategory = (Long)res.get("PRICECATEGORY_FK");
    Movie m = new Movie(
    (Long)res.get("MOVIE_ID"),
    (String)res.get("MOVIE_TITLE"),
    (java.sql.Timestamp)res.get("MOVIE_RELEASEDATE"),
    ((Boolean)res.get("MOVIE_RENTED"),
    priceCategoryDAO.getById(priceCategory));
    return m;
  }
```

3.5.2. Query Results with Callbacks

Kopieren von Daten in Maps reicht nicht. Callback Methoden können übergegeben werden die von Template ausgeführt werden.

ResultSetExtractor Hohlt JDBC Result Set:

```
  interface ResultSetExtractor<T> {
public T extractData(ResultSet arg0)
throws SQLException, DataAccessException;
}
```

Implementationen sollen die ganze Resultset verarbeiten, und diese Resultset nicht schliessen.

RowCallBackHandler ProcessRow aufgerufen für jede Zeile , Resultät im Context gespeichert.

```
 interface RowCallbackHandler {
public void processRow(ResultSet arg0)
throws SQLException;
}
```

RowMapper gibt objekt zurück, Query gibt Liste zurück.

```
 interface RowMapper<T> {
public T mapRow(ResultSet rs, int rowNum)
throws SQLException
}
```

Listing 3.9: QueryResults Callback Example JDBC

```
public List<Movie> getByTitle(String name) {
 JdbcTemplate template = getJdbcTemplate();
 return template.query(
   "select * from MOVIES where MOVIE_TITLE = ?",
   new RowMapper<Movie>(){
 public Movie mapRow(ResultSet rs, int row)
 throws SQLException {
   long priceCategory=rs.getLong("PRICECATEGORY_FK");
 return new Movie(
     rs.getLong("MOVIE_ID"),
     rs.getString("MOVIE_TITLE"),
     rs.getTimestamp("MOVIE_RELEASEDATE"),
     rs.getBoolean("MOVIE_RENTED"),
     priceCategoryDAO.getById(priceCategory));
   }
 },
   name
 );
}
```

3.5.3. Exception Hirachie

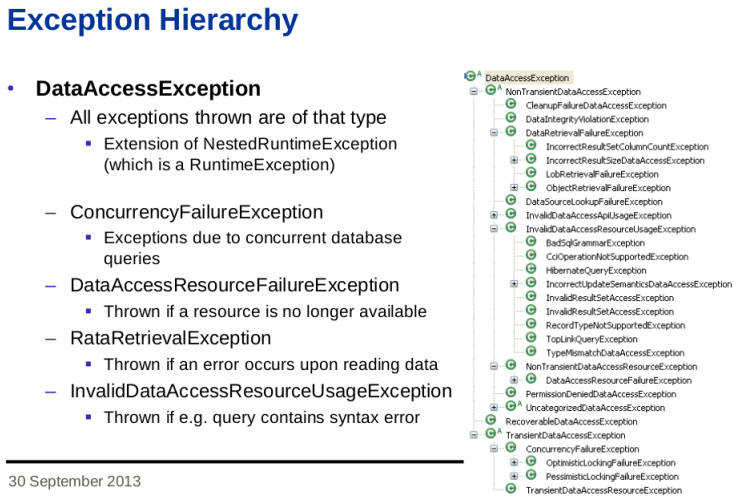


Abbildung 3.2.: figure

3.6. Resourcen in Spring

Data Sources javax.sql.Datasource : getConnection() + getConnection(String username,String password)

Listing 3.10: Sample JDBC Declaration von DataSource

```
<bean id="dataSource" class=
"org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName"
value="org.hsqldb.jdbcDriver"/>
<property name="url"
value="jdbc:hsqldb:hsql://localhost/lab-db"/>
<property name="username" value="sa"/>
<property name="password" value=""/>
</bean>
```

org.sfspringframework.jdbc.datasource.DriverManagerDataSource No connection pooling, ++unitTesting.

org.apache.commons.dbcp.BasicDataSource Jakarta, Connection Pooling von aussere JavaEE containers.

org.springframework.jndi.JndiObjectFactoryBean für JNDI Verbindungen.

Listing 3.11: Bean Resources JDBC

```
<bean id="userDAO"
class="ch.fhnw.edu.rental.daos.impl.JdbcTemplateUserDAO">
<property name="dataSource" ref="dataSource"/>
<property name="rentalDAO
ref="rentalDAO"/>
<property name="movieByIdSql">
<value>select * from MOVIES where MOVIE_ID = ?</value>
</property>
</bean>
<bean id="dataSource" class =
"org.springframework.jdbc.datasource.DriverManagerDataSource"
p:driverClassName="${jdbc.driverClassName}"
p:url="${jdbc.url}"
p:username="${jdbc.username}"
p:password="${jdbc.password}" />
```

3.7. Testing : DBUnit

Listing 3.12: DBUnit Test Data

```
<dataset>
<MOVIES
MOVIE_ID='1'
MOVIE_TITLE='Lord of the Rings'
MOVIE_RENTED='1'
PRICECATEGORY_FK='1'/>
<users user_id='1'
USER_NAME='Keller'
USER_FIRSTNAME ='Marc'/>
```

IDataSet representiert DataSet

```
FileInputStream stream = new FileInputStream ("dataset.xml");
FlatXmlDataSetBuilder builder = new FlatXmlDataSetBuilder();
IDataSet dataSet = builder.build(stream);
```

IDatabaseConnection ist für Datenbankverbindung

```
IDatabaseConnection connection = new DatabaseConnection(conn);
DatabaseConfig config = connection.getConfig();
config.setProperty(
DatabaseConfig.PROPERTY_DATATYPE_FACTORY,
new HsqldbDataTypeFactory()
);
```

DatabaseOperation.xxx.execute(connection, dataSet) xxx=UPDATE,INSERT,DELETE,DELETE_ALL,REFRESH,CLEAN_IN
= DELETE_ALL+INSERT

REFRESH Refresh oder Daten einfügen in DataSet. Existierende rows nicht geändert.

TestUtil Methode getSpringContext ist dann in SetUp methode für alle Tests.Ruft resetData auf ein DBInitializer Instanz.

Listing 3.13: TestUtil Example DBUnit

```
public class JdbcDbInitializer implements DbInitializer {
   public void resetData(ApplicationContext context)
   throws Exception {
   DataSource dataSource = (DataSource)context.getBean(
   "dataSource");
     Connection dbconn = dataSource.getConnection();
 IDatabaseConnection con = ...
 IDataSet ds = ...
     try {
       DatabaseOperation.CLEAN_INSERT.execute(conn, ds);
     } finally { connection.close(); }
   }
 }
```

DataSource Properties:

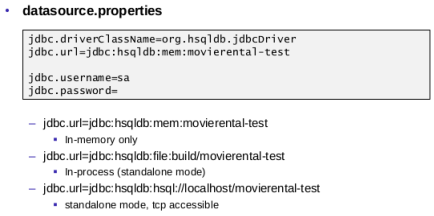


Abbildung 3.3.: DataSource Properties DBUnit

3.8. JPA

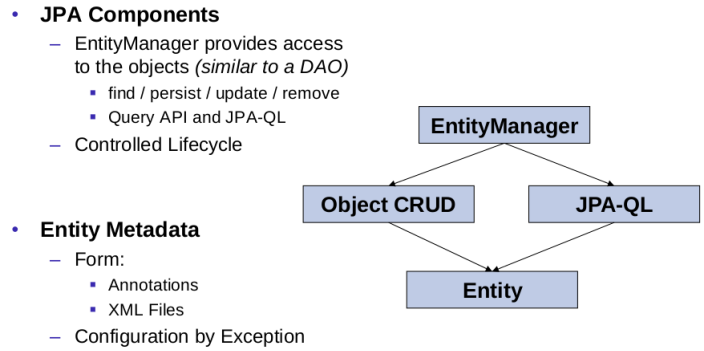


Abbildung 3.4.: JPA Intro

```
 @Entity
public class Movie {
@Id
private Long id;
private String title;
private boolean rented;
private Date releaseDate;
protected Movie(){}
public Movie(String title, Date releaseDate){ ... }
public String getTitle(){return title;}
public boolean isRented(){return rented;}
public void setRented(boolean rented){this.rented = rented;}
...
}
```

3.8.1. Entity Manager

Listing 3.14: Entity Manager JPA

```
 EntityManagerFactory emf =
Persistence.createEntityManagerFactory(persistenceUnit);
EntityManager em = emf.createEntityManager();
Movie m = new Movie("Wall-e",
new GregorianCalendar(2011, 9, 25).getTime());
em.persist(m);
m = em.find(Movie.class, 2211);
m.setRented(true);
```

3.8.2. Entity Annotations

Listing 3.15: JPA Annotations for Entities

```
 @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;
}
}
```

3.8.3. Entity Klasse Voraussetzungen

- Entity muss mit @Entity annotiert werden.
- Default Konstruktor ohne Argumente, kann auch noch andere haben, aber default muss public oder protected sein.
- Contraints:

  Entity Klasse und methoden mussen nicht final sein.

  Klasse muss eine "Top Level" Klasse sein.

  keine innere Klasse, keine Interface, keine Enum.

3.8.4. Annotation Definitionen

@Entity Klassifiziert Klasse als Entity :

```
                                                    @Target(TYPE) @Retention(RUNTIME)
public @interface Entity {
String name() default "";
}
```

Name: Entity nennen in Queries muss keine resevierte Wort aus JP-QL sein. Unqualified Name von Entity Klasse.

@Table Gibt die Tabelle für Entity an :

```
                                                    @Target(TYPE) @Retention(RUNTIME)
public @interface Table {
String name() default "";
String catalog() default "";
String schema() default "";
UniqueConstraint[] uniqueConstaints() default {};
}
```

name Name der Tabelle

Catalog Catalog der Tabelle

schema Schema der Tabelle]

uniqueConstaints Contraints apllied to generated DDL Tables.

@Column Kolumne für persistente Eigenschaft (Property)

```
 @Target({METHOD,FIELD}) @Retention(RUNTIME)
public @interface Column {
String name() default "";
// name of column
boolean unique() default false;
// is DB column unique
boolean nullable() default true; // is DB column nullable
boolean insertable() default true; // is manipulation with a
// sql insert allowed
boolean updatable() default true;
String columnDefinition() default "";
// e.g. CLOB / BLOB
String table() default "";
// table in which field
// is stored (sec. table)
int length() default 255;
// size for strings
int precision() default 0;
// decimal precsion
int scale() default 0;
// decimal scale
}
```

Unterstützte Typen: char, short, int, long, byte, float, double, boolean. Strings,Primitive Wrappers, Big Numericals, Java time types. JDBC time types. Byte und Char Arrays, Beliebige Serializable Types. Entity types & Collections of entity types

Single Valued Properities `{ T getProperty()`
`{ void setProperty(T t)`

Für Multi-valued persistente Eigenschaften muss T von Collection Set List oder Map sein.

@Enumerated
```
                                @Target({METHOD,FIELD}) @Retention(RUNTIME)
public @interface Enumerated {
EnumType value() default "ORDINAL";
}
```

Kann Ordinal oder String.

@Lob Large Object :

```
                                @Target({METHOD,FIELD}) @Retention(RUNTIME)
public @interface Lob {
}
```

The Lob type is inferred from the type of the persistent field or property, and (except for string and character-based types) defaults to Blob.

@Transient Nicht persistentes Feld.

@Id Typen für ID sind Primitive,Wrapper Typen oder Arrays davon, und String, Large numerics. Temporal Typen. Generierung :

Assigned keine key Generation.

Identity Auto Increment

Sequence DB Specific.

Table PK in Seperate Tabelle.

@GeneratedValue `Target({METHOD,FIELD}) @Retention(RUNTIME)`

```
public @interface GeneratedValue {
GenerationType strategy() default AUTO;
String generator() default "";
}
public enum GenerationType {TABLE, SEQUENCE, IDENTITY, AUTO};
```

Primary Key Declaration Simple : Single persistent field. Marked with @Id. Composite Primary Key : Single persistent field or set thereof. PK Klasse muss definiert werden. Legacy für ID aus mehrere Kolumnen : @EmbeddedId oder @IdClass

Listing 3.16: Annotationsbeispiel

```
 @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;
}
```

3.8.5. JPA Inheritance

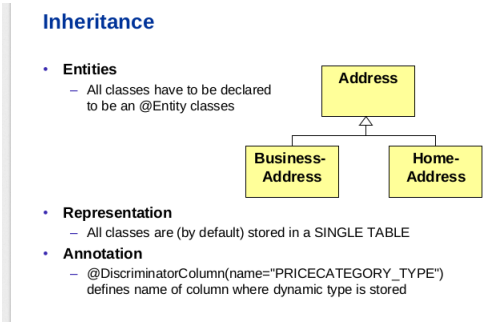


Abbildung 3.5.: figure

3.8.6. JPA Method and Field Annotations

3.8.7. JPA Entity Manager

Zweck Verwaltet Entity Instanz Lebenszyklus. Methoden um mit PersistenzKontext zu interagieren.

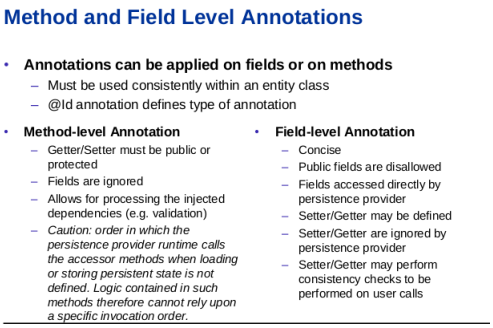


Abbildung 3.6.: figure

Managed Objects Returned oder an Entity Manager gegeben.

PersistenzKontext Set of Managed Objects. Nur eine Instanz von jedem Model.

Erzeugung Entity Manager Factory.

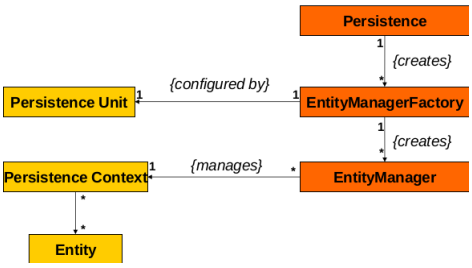


Abbildung 3.7.: figure

Listing 3.17: JPA 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>
```

Erzeugung von EM:

Listing 3.18: Erzeugung von EM

```
 EntityManagerFactory emf =
Persistence.createEntityManagerFactory("movierental");
EntityManager em = emf.createEntityManager();

@PersistenceUnit(name="movierental")
EntityManagerFactory emf = null;
@PersistenceContext(name="movierental")
private EntityManager em;
```

Listing 3.19: Entity Manager JPA

```
public interface EntityManager {
public void persist(Object entity);
public <T> T merge(T entity);
public void remove(Object entity);
public <T> T find(Class<T> entityClass, Object primaryKey);
...
public void flush();
public void setFlushMode(FlushModeType flushMode);
public FlushModeType getFlushMode();
public void refresh(Object entity);
public void clear();
public boolean contains(Object entity);
public Query createQuery(String ejbqlString);

}
```

merge Gibt neue einzigartige Instanz zurück. Insert Operationen.

refresh Zustand von Datenbank züruckgegeben. Änderungen ignoriert.

flush Persistence sync.

clear entities detatched, nicht flushed entities ignoriert.

createQuery

```
Query q = em.createQuery("from Movie m where m.title = :title");
q.setParameter("title", title);
List movies = q.getResultList();
```

3.8.8. Spring und Entity Manager

Listing 3.20: Spring und JPA

```
<bean id="entityManagerFactory" class =
"org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
<property name="persistenceUnitName" value="movierental" />
</bean>
<!--
Persistence unit is defined in persistence.xml that resides in the
META-INF directory on the class path
LocalContainerEntityManagerFactoryBean
    Allows to control the used datasource and the weaving process
Spring supports JPA annotations for persistence contexts / units if a
PersistenceAnnotationBeanPostProcessor is enabled
--!>
<bean class = "org.springframework.orm.jpa.support.
PersistenceAnnotationBeanPostProcessor" />
```

Listing 3.21: Spring Injected DAO JPA

```
public class JpaMovieDAO implements MovieDAO {
@PersistenceContext
private EntityManager em;
public Movie getById(Long id) {
return em.find(Movie.class, id);
}
...
```

3.8.9. Transactions mit JPA

em.transaction.begin, em.transaction.commit ohne spring, oder :

Listing 3.22: Spring Transactions AOP JPA

```
<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
<property name="entityManagerFactory"
ref="entityManagerFactory" />
</bean>
```

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

3.8.10. JPA Associations

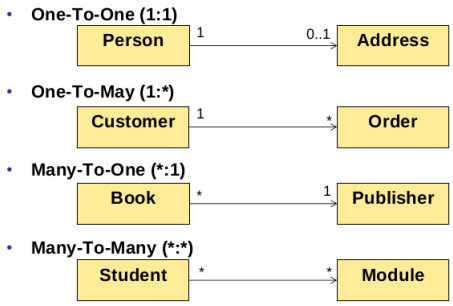


Abbildung 3.8.: Relationsmöglichkeiten JPA

Cascading

PERSIST Persist auf alle Objekte.

REMOVE Falls entity entfernt ist, dann sind alle damit asozierte Entitäten auch entfernt.Sollte nur für @OneToOne und @OneToMany gelten.

REFRESH Falls Entität "refreshed" ist, dann sind alle assozierte auch so gemacht.

MERGE Ist ein Entität managed geworden, (Merged mit Persistence Context) dann sind alle assozierte auch so.

DETACHED Alle Entitäten die damit assoziert sind weden auch als Detatched betrachtet.

ALL Selbsterklärend

FetchType

- **On associations, a fetch type can be defined**
  - Allows to specify when objects are loaded
  - Can be specified on
    - @OneToMany / @OneToOne / @ManyToMany / @ManyToOne
    - @Basic for regular fields
  - EAGER
    - Dependent objects are loaded with original object
    - Default for @OneToOne and @ManyToOne
    - Default for regular fields
  - LAZY
    - Dependent objects are loaded on demand
    - Original object must not be detached upon loading associated entities!
    - Default for @OneToMany and @ManyToMany

Abbildung 3.9.: JPA FetchType

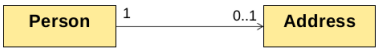


Abbildung 3.10.: figure

OneToOne Unidirectional

0..1 mehr flexibel als 1..1 - 1..1 = Konstruktorparam.

Listing 3.23: OnetoOne SQL

```
 CREATE MEMORY TABLE ADDRESS(
ID INTEGER NOT NULL PRIMARY KEY,
CITY VARCHAR(255), STREET VARCHAR(255))
CREATE MEMORY TABLE PERSON(
ID INTEGER NOT NULL PRIMARY KEY,
ADDRESS_ID INTEGER NOT NULL,
CONSTRAINT FK203A7330FF0EDE FOREIGN KEY(ADDRESS_ID)
REFERENCES ADDRESS(ID))
```

Listing 3.24: OnetoOne Code Java

```
 @Entity
public class Person {
@Id @GeneratedValue(strategy=GenerationType.AUTO)
private int id;
private String name;
@OneToOne
private Address address;
private Person(){}
public Person(String name) { this.name = name; }
public int getId() { return id; }
public Address getAddress() { return address; }
public void setAddress(Address address){
this.address = address;}

}
@Entity
public class Address {
@Id @GeneratedValue(strategy=GenerationType.AUTO)
private int id;
private String street, city;
private int zip;
private Address(){}
public Address(String street, int zip, String city) {
this.street = street; this.city = city; this.zip = zip;
}
public int getId() { return id; }
...
}
em.getTransaction().begin();
Person p = new Person("Dominik");
Address a = new Address("Steinackerstrasse", 5210, "Windisch");
p.setAddress(a);
em.persist(p);
em.persist(a); // Only if persist cascading not defined, order of persist operations not
    relevant.
em.getTransaction().commit();


@Entity
public class Person {
@Id @GeneratedValue(strategy=GenerationType.AUTO)
private int id;
private String name;
@OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
private Address address;
private Person(){}
public Person(String name) { this.name = name; }
public int getId() { return id; }
public Address getAddress() { return address; }
public void setAddress(Address address){this.address=address;}
```

Listing 3.25: OneToOne Bidirectional

```
@Entity
public class Address {
@Id @GeneratedValue(strategy=GenerationType.AUTO)
private int id;
private String street, city;
private int zip;
@OneToOne(mappedBy="address")
private Person person;
private Address(){}
public Address(String street, int zip, String city) {
this.street = street; this.city = city; this.zip = zip;
}
public Person getPerson() { return person; }
public void setPerson(Person person) { this.person = person; }
}

em.getTransaction().begin();
Person p = new Person("Dominik");
Address a = new Address("Steinackerstrasse", 5210, "Windisch");
p.setAddress(a);
em.persist(p);
em.flush();
// p = em.find(Person.class, 1);
System.out.println(p.getAddress().getStreet());
System.out.println(p.getAddress().getPerson().getName());
em.getTransaction().commit();

//MARKIERT AUF EINE SEITE NUR, MAPPEDBY
```

OneToMany biderectional

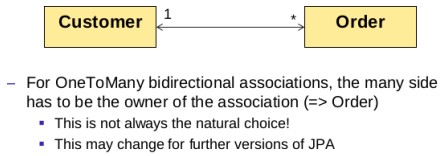


Abbildung 3.11.: JPA OnetoMany

Listing 3.26: JPA 1 to n

```
@Entity
public class Customer {
@Id @GeneratedValue(strategy=GenerationType.AUTO)
private int id;
// this is the inverse side of the relationship
@OneToMany(mappedBy="customer", cascade=CascadeType.ALL)
private Collection<Order> orders;
public Collection<Order> getOrders() {
return orders;
}
public void setOrders(Collection<Order> orders) {
this.orders = orders;
}
}

@Entity
public class Order {
@Id @GeneratedValue(strategy=GenerationType.AUTO)
private int id;
@ManyToOne
// Order is the owner
private Customer customer;
// of the relationship
public Order(){}
public Customer getCustomer() { return customer; }
public void setCustomer(Customer customer) {
this.customer = customer;
}
```

```
public int getId() { return id; }
}
```

Listing 3.27: JPA 1 to n sql

```
create table customer (
ID integer
FIRSTNAME varchar(32)
LASTNAME varchar(32)
);

create table address (
ID integer
CUSTOMER_ID integer
STATUS varchar(4)
PRICE decimal(10,2)
);
```

Listing 3.28: OneToMany Bidirectional persist JPA

```
 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();
```

- n Seite ist immer Besitzer.
- Nur referenzen von Order nach Customer sind persisted.
- 2 Orders sind im DB gespeichert (persist)
- Assoziationen sind nicht persisted.

Many to One

Listing 3.29: JPA Many to One Rental - User

```
 @Entity
public class Rental implements Serializable {
@Id
private int id;
@ManyToOne // Rental is the owner of the relationship
@JoinColumn(name="USER_FK") // optional
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; }
}

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;
}
}
```

Many to Many

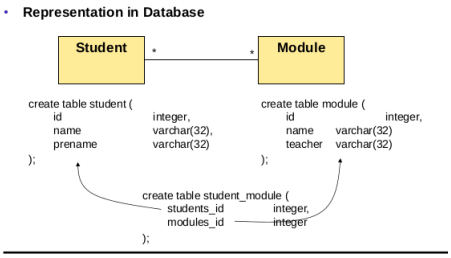


Abbildung 3.12.: figure

Listing 3.30: nton Test JPA

```
em.getTransaction().begin();
Module m1 = new Module("WebFrameworks", "Luthiger");
Module m2 = new Module("ConcurrentProg", "Gruntz");
Student s1 = new Student("Meier");
Student s2 = new Student("M ller");
em.persist(m1);
// all entities are persisted
em.persist(m2);
em.persist(s1);
em.persist(s2);
Collection<Module> modules = new LinkedList<Module>();
modules.add(m1); modules.add(m2);
Collection<Student> students = new LinkedList<Student>();
students.add(s1); students.add(s2);
s1.setModules(modules);
// (1)
m1.setStudents(students);
// (2)
em.getTransaction().commit();
```

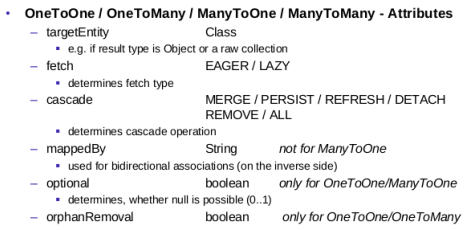


Abbildung 3.13.: n to n Attributen

Entity Relation Richtungen

Unidirectional Hat eine Besizer Seite.

Bidirectional Hat Besitzer- und Invers- Seite.

Besitzerseite 1 to 1 Besizer hat FK

n to 1 n hat FK auf 1.

n to n Beide Seiten möglich, Besitzer verwaltet Aktualisierungen der Beziehung im DB.

Mapped By Inverse gibt Besitzerseite an mittels diese Annotation. Nicht im ManyToOne angegeben. Many Seite auto Besitzer.

orphanRemoval

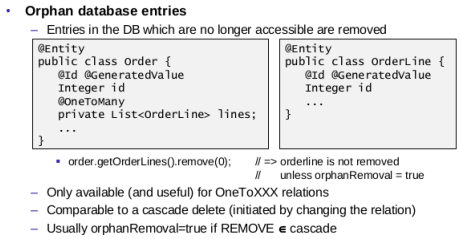


Abbildung 3.14.: figure

OrderBy

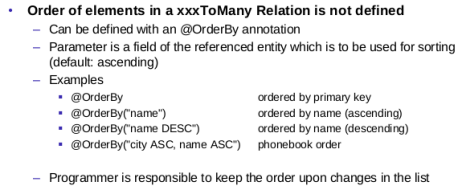


Abbildung 3.15.: figure

3.8.11. Inheritance with JPA

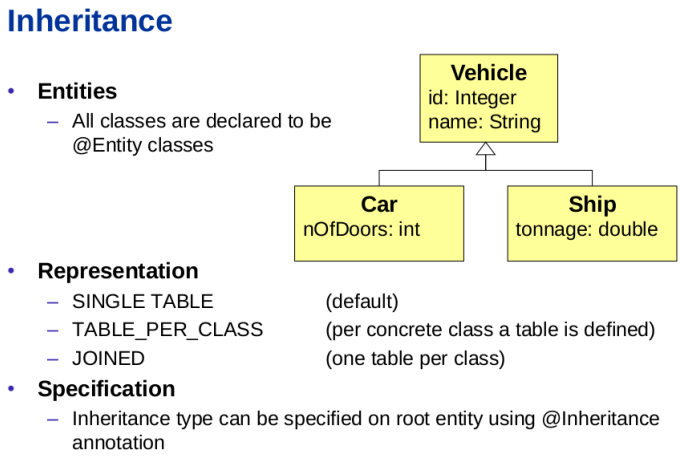


Abbildung 3.16.: figure

Single Table Strategy

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
```

- Alle Attributen in 1 Tabelle.
- Typ gespeichert in eine Discriminator Spalte (DType)

```
 @DiscriminatorColumn(String name, DiscriminatorType type)
name: Name of the column (default: DTYPE)
type: Type of the column (STRING, CHAR, INTEGER)
```

- Descriminator spezifizieren mit @DescriminatorValue annotation, default ist nicht qualifizierte Klassenname.

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

Abbildung 3.17.: figure

- **Alle felder die in subklassen addiert sind mussen nullable sein**
- **Foreign Keys can only refer to the base class**

Joined Strategy

ID	NAME
1	VW Sharan
2	Smart
3	Queen Mary

ID	NOFDOORS
1	5
2	2

ID	TONNAGE
3	76000

Abbildung 3.18.: figure

- Eine Tabelle ist für jede Klasse definiert, und PK ist verlinkt.
- PK ist eine FK die an Basis Tabelle zeigt.
- **Vorteile:** Normaliserte Schema, Alle Felder können mit NOT NULL definiert werden. FK beziehung zu konkrekte Subklassen möglich.
- **Nachteile:** Jede Entität muss über mehrere Tabellen gehen.

Table per Class Strategy

ID	NAME	NOFDOORS
1	VW Shara	5
2	Smart	2

ID	NAME	TONNAGE
3	Queen Mary	76000

Abbildung 3.19.: figure

- Eine Tabelle ist für jede nicht abstrakte Klasse definiert. Es beinhaltet alle Attributen von diese Klasse und die Basisklasse.
- **Vorteile:** Non Null Constraints definierbar. FK Referenzen an konkreten Subklassen möglich aber nicht zu Abstrakte Basisklassen.
- **Nachteile:** Polymoprphic Abfragen nötig, ID Generator nicht nutzbar. Nicht von Jpa2.0 vorrausgesetzt. Hibernate stellt es zur Verfügung.

Non Entity Base Class

Entitätsklassen können von nicht-domänen Klassen abgeleitet werden welche nicht zur TabellenMapping gehören.

Listing 3.31: Entität Basisklasse

```
  @MappedSuperclass
public abstract class UuidEntity {
@Id protected String id;
public UuidEntity() { this.id = UUID.randomUUID().toString(); }
public String getId() { return id; }
public boolean equals(Object x) {
return x instanceof UuidEntity
&& ((UuidEntity)x).id.equals(id);
}
public int hashCode() { return id.hashCode(); }
}
```

3.8.12. JPA Query Sprache (JPQL)

JPQL Mnipulieren von DB, von SQL inspiriert aber Abfragen sind direkt über Entitäten und ihre Felder.

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]
```

Typed Query

- Queries are created with the method createQuery on an entity manager

  em.createQuery(String q) returns a un-typed query

  em.createQuery(String q, Class¡T¿ c) returns a typed query

- Results:

  q.getResultList() static type of result is a list (un-typed or typed)

  q.getSingleResult() result of type Object or of the expected type

  NoResultException if no entry was found

  NonUniqueResultException if several entities were found

Listing 3.32: TypedQuery Example

```
 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();
```

Named Query

Listing 3.33: Query auf Entitätsklassen

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

Listing 3.34: Query auf EntitätsManager

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

Select und From

Where Klausel

Functions

Ordering and Paging

Order By Klasel Gibt Resultät eine Ordnung Asc or desc, desc ist default.

Range beschränken

- public Query setMaxResults(int maxResult)
- public Query setFirstResult(int startPosition)

  Start with 0

- **Select statement**
  - Select defines projection (structure of query result)
  - From declares entity variables

```
SELECT c FROM Customer c WHERE c.address.city = 'Basel'
```

```
SELECT c.name, c.prename FROM Customer c
```

  - Result is of type Object[] (of length 2) or List<Object[]>

```
SELECT DISTINCT c.address.city FROM Customer c
```

  - DISTINCT removes duplicates

```
SELECT NEW ch.fhnw.edu.Person(c.name,c.prename) FROM Customer c
```

  - Allows to create arbitrary objects (also non-Entity classes)

```
SELECT pk FROM PriceCategory pk
```

  - Polymorphic select statements are possible

Abbildung 3.20.: figure

Where Clause

- **Where restricts the set of entities**
- **Operators**
  - +, -, *, /
  - =, <=, >=, <, >, <>
  - [not] between … and … where c.nOfDoors between 4 and 5
  - [not] like … where c.name like 'A%' (% and _)
  - [not] in (….) where c.phonePrefix in ('079', '078', '077')
  - is [not] null where c.adr is not null
  - is [not] empty where c.rentals is not empty
  - [not] member of where 'JPA' member of c.skills
  - [not] exists *tests whether a subquery returns a result*
  - not, and, or

Abbildung 3.21.: figure

Listing 3.35: Order und Paging Beispiel

```
1  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();
```

Listing 3.36: Weitere Ordering + Paging

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

H2,Postgres mit Hibernate:

```
   SELECT * FROM Movie movie0_
  ORDER BY movie0_.name
  LIMIT 10
  OFFSET 20
```

Mysql mit Hibernate:

```
1  SELECT * from Movie movie0_
  ORDER BY movie0_.name
  LIMIT 20, 10
```

Listing 3.37: Weitere Beispiele Order + Paging

```
   TypedQuery<Movie> q = em.createQuery(
```

Where Clause: Parameters

- **Numbered Parameters**

```
"select m from Movie m where m.title = ?1"
```

  - Actual value is set with q.setParameter(int, Object)
  - Numbering starts with 1

```
q.setParameter(1, "Skyfall");
```

- **Named Parameters**

```
"select m from Movie m where m.title = :title"
```

  - Actual value is set with q.setParameter(String, Object)

```
q.setParameter("title", "Skyfall");
```

Abbildung 3.22.: figure

Functions

- **Functions can be used in select and where clauses**
  - Strings:
    - CONCAT, SUBSTRING, TRIM, LOWER, UPER, LENGTH, LOCATE
  - Math functions:
    - ABS, SQRT, MOD
  - Many Associations:
    - SIZE, INDEX
  - Temporal:
    - CURRENT_DATE(), CURRENT_TIME(), CURRENT_TIMESTAMP()
  - Conditional:
    - CASE
      WHEN … THEN …
      WHEN … THEN …
      ELSE …
      END

Abbildung 3.23.: figure

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

Oracle:

```
   SELECT * FROM
  (
  SELECT row_.*, rownum rownum_ from
  (
5 SELECT * FROM Movie movie0_ ORDER BY movie0_.name
  ) row_
  WHERE rownum <= 30
  )
  WHERE rownum_ > 10
```

Inner Joins

Outer Joins

Fetch Joins und Lazy Loading

Update and Delete

- With JPA update and delete operations can be performed without creating instances
- Can be applied on one entity only (no joins)
- Entities which are loaded in a entity context are not changed =¿ should be executed in a separate transaction

Aggregate Functions

- **Aggregate Functions can be defined in the select clause**
  - AVG
  - COUNT
  - MAX
  - MIN
  - SUM

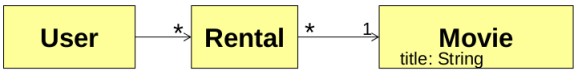
```
select max(c.age) from Customer c
```

```
select count(r) from Rental r where r.user.name = :name
```

```
select max(u.id) from User u
```

Abbildung 3.24.: figure

- **Associations across Many-Associations**



  - **SELECT u.rentals.movie.title FROM User u** does not work

```
SELECT r.movie.title from User u join u.rentals r
SELECT r.movie.title from User u inner join u.rentals r
SELECT r.movie.title from User u, in(u.rentals) r
```

  - Performs an inner join => entries may be duplicated (if different users refer to the same movie)
  - Inner join only returns entities which are referenced

Abbildung 3.25.: figure

Listing 3.38: Bulk update und Delete

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

Outer Joins

Inner vs Outer Joins

- Inner join only returns entities which are part of an association, whereas outer join returns objects which have no references / are not referenced
- JPA support only left outer joins
- Example
  - Inner Join

```
SELECT u.name, r from User u join u.rentals r
```

    - Returns name and rentals only from those users which have rented movies
    - Syntax: `[inner] join pathexpression variable`
  - Outer Join

```
SELECT u.name, r from User u left join u.rentals r
```

    - Returns all users (r may then be null)
    - Syntax: `left [outer] join pathexpression variable`

Abbildung 3.26.: figure

- **Fetch Joins and Lazy loading**
  - Allows to eagerly load dependent objects, i.e. allows to redefine the loading strategy for a query

```
SELECT u from User u
```

    - Rentals are not loaded (if not defined as eager loading)

```
SELECT u from User u left join fetch u.rentals
```

    - Query which loads the rentals and which returns ALL users (also those which do not have rentals) as it is an outer join
    - Inner join would be possible as well (but less useful)
  - Syntax:
    - `[ left [outer] | inner] join fetch pathexpression`

Abbildung 3.27.: figure

4. Data Transfer Objects

Detatched Entitäten als DTOs Problems

- Lazy load Exceptions werden geworfen falls Felder die noch nicht geladen sind zugegriffen worden sind.
- Accessors die keine Exceptions werfen verletzt Verträge.
- Reflection gibt proxies züruck die nicht serialisierbar sind.

Data Transfer Objects Werden benutzt um daten über Schichten der Applikation hinaus zu transportieren. Nur die benötigten Daten der Fordernde Layer werden übergegeben nicht alles. Keine Lazy Loading Überraschungen. Clienten sind dann ORM unabhängig.

Listing 4.1: DTO Beispiel

```
public class UserDTO implements Serializable {
private Long id;
private String name;
private String firstName;
private List<Long> rentalIds; // allows to access rentals
// on demand
public UserDTO(Long id, String name, String firstName,
List<Long> rentalIds) {
this.id = id;
this.name = name;
this.firstName = firstName;
this.rentalIds = rentalIds;
}
//Service impl JAVA muss innerhabl Transaktion ausgef hrt werden.
public UserDTO getUserDataById(Long id) throws ServiceException {
User u = getUserById(id);
List<Long> rentalIds = new ArrayList<Long>();
for(Rental r : u.getRentals()) rentalIds.add(r.getId());
return new UserDTO(u.getId(), u.getName(), u.getFirstName(),
rentalIds);
}
//Service IMPL JPA
public UserDTO getUserDataById(Long id) { // in JpaDAO
Query q = em.createNamedQuery("user.dataById");
q.setParameter("id", id);
UserDTO dto = (UserDTO)q.getSingleResult();
q = em.createNamedQuery("user.rentalsById");
q.setParameter("id", id);
dto.setRentalIDs((List<Long>)q.getResultList());
return dto;
}
@NamedQuery(name="user.dataById", query=
"SELECT NEW ch.fhnw.edu.services.userservice.model.UserDTO(
u.id, u.name, u.firstName) FROM User u WHERE u.id = :id"),
@NamedQuery(name="user.rentalsById", query=
"SELECT r.id FROM User u, IN(u.rentals) r WHERE u.id = :id")
```

4.1. Dozer

- Java Bean zu Bean Mapper. Kopiert Daten rekursiv. Kann benutzt werden um DTOs zu Kopieren.
- Unterstutzt Einfache PropertyMapping, Komplexe TypMappings, Direktionale Mappings, implicit explicit mapping und rekursiv Mapping. Mapping von Collections auch unterstutzt.

Listing 4.2: DTO Lösung mit Dozer

```
public UserDTO getUserDataById(Long id) throws ServiceException {
```

```
return (UserDTO)mapper.map(getUserById(id), UserDTO.class);
}
Mapper mapper; // to be injected by spring
public void setMapper(Mapper mapper) { this.mapper = mapper; }
```

```
<bean id="Mapper" class="org.dozer.DozerBeanMapper">
<property name="mappingFiles">
<list>
<value>dozer.xml</value>
</list>
</property>
</bean>
<mappings>
<configuration>
<custom-converters>
<converter type="ch.fhnw.edu.util.RentalConverter" >
<class-a>
ch.fhnw.edu.services.rentalservice.model.Rental</class-a>
<class-b>java.lang.Long</class-b>
</converter>
</custom-converters>
</configuration>
<mapping type="one-way">
<class-a>ch.fhnw.edu.services.userservice.model.User</class-a>
<class-b>ch.fhnw.edu.services.userservice.model.UserDTO</class-b>
<field>
<a>rentals</a>
<b>rentalIds</b>
</field>
</mapping>
</mappings>
```

4.2. Pros Cons

- Dtos haben gleiche Felder wie Domänenobjekten.

- Kopieren von Attributen hin und her.

+ Löst Lazy Loading Problemen.

+ Design - Müss gedanken machen über Remote Service Facades. Infos aus meherere Entitäten können in DTO kombiniert werden.

5. Remoting with Spring

Spring gibt Remoting Abstraktionen an. *Gibt unchecked Remote AccessException ersetzt rmi.RemoteException* **Bemerkungen:**

- Keine standard Unterstützung für security context Propagieren.
- Keine Remote Transactionspropagierung.
- **Keine Thread Safety** selbe Bean Instanz kann von mehrere Threads aufgerufen werden.
- Keine Session Verwaltung - alle Klienten teilen das selbe Bean Instanz.

5.1. Spring Service Export mit RMI

Listing 5.1: Spring Config fuer RMI

```
<bean class =
"org.springframework.remoting.rmi.RmiServiceExporter">
<property name="serviceName" value="AccountService"/>
<property name="service" ref="accountService"/>
<property name="serviceInterface"
value="ch.fhnw.edu.bank.AccountService"/>
</bean>
```

- Interface ist POJI = (Plain old Java Interface)
- RMIRegistry wird automatisch gestartet falls registryHost nicht explizit angegeben ist.

Listing 5.2: spring rmi configuration optionale Eigenschaften

```
<property name="registryPort" value="1099"/>
<property name="registryHost" value="147.86.3.30"/>
<property name="servicePort" value="7777"/>

default: 1099
default: localhost
default: 0
```

Listing 5.3: Server Code für RMI Export

```
public class Server {
public static void main(String[] args) throws Exception {
new ClassPathXmlApplicationContext("context.xml");
System.out.println("server started");
}
}
```

Log output :

```
            INFO: Looking for RMI registry at port '1099'
INFO: Could not detect RMI registry - creating new one
INFO: Binding service 'AccountService' to RMI registry:
RegistryImpl[UnicastServerRef [liveRef:
[endpoint:[10.211.1.33:1099](local),objID:[0:0:0, 0]]]]
server started
```

5.2. Benutzung ein RMI Dienst mittels Spring

Listing 5.4: Spring config File : benutzung

```
 <bean id = "accountService" class =
"org.springframework.remoting.rmi.RmiProxyFactoryBean">
<property name="serviceUrl"
value="rmi://localhost:1099/AccountService"/>
<property name="serviceInterface"
value="ch.fhnw.edu.bank.AccountService"/>
</bean>
```

5.3. Remoting Exceptions mit Spring

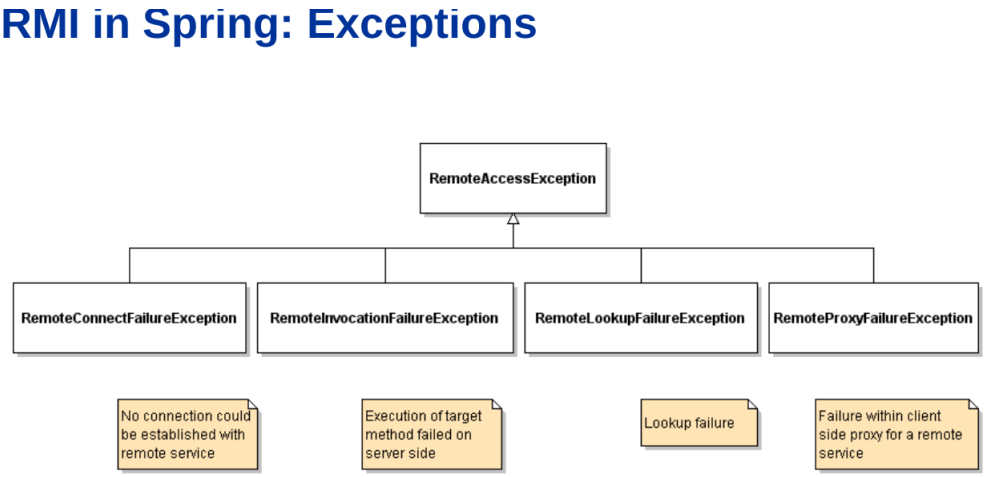


Abbildung 5.1.: Spring Remoting Exception Heirachy

Adapter Serverseitig wird ein Adapter angegeben dass RMI Massstäbe folgt. Leitet Requests weiter an Zielobjekt.

Proxy RemoteInvocationHandler Client leitet Requests an RMI Interface weiter und wird Exceptions mappen.

$C \rightarrow P \rightarrow A \rightarrow S$ C = Client P = Proxy A = Adapter S = Service.

5.4. Spring Remoting Client Seite (nicht Prüfungsrelevant)

Listing 5.5: Spring Remoting Client Code

```
 import java.rmi.Remote;
import java.rmi.RemoteException;
public interface RmiInvocationHandler extends Remote {
  public String getTargetInterfaceName()
  throws RemoteException;
    public Object invoke(RemoteInvocation invocation)
    throws RemoteException,
    NoSuchMethodException,
    IllegalAccessException,
    InvocationTargetException;
}

// The adapter which is registered by Spring in the RMI-registry implements this interface

public class RemoteInvocation implements Serializable {
  private String methodName;
  private Class[] parameterTypes;
  private Object[] arguments;
  // constructors, getters & setters ommitted
  public Object invoke(Object targetObject)
  throws NoSuchMethodException,
```

```
    IllegalAccessException,
23  InvocationTargetException {
      Method method = targetObject.getClass().getMethod(this.methodName, this.parameterTypes)
          ;
      return method.invoke(targetObject, this.arguments);
    }
  }
28 //Executed on the server by the adapter
```

5.5. Bemerkungen zum Remoting

Regelmässige Objekte
- Exportierte Objekten können nicht von regelmässige RMI Klienten benutzt werden.
- Exportierte Objekten sind vom RmiInvocationWrapper_Stub bzw RmiInvocationHandler.
- Falls Dienst ein Remote Interface anbietet + Service als ServiceInterface spezifiziert ist bei Export, dann ok für normalle Clients.

Client Interface Benutzt von Klientseite kann Remote sein, kann weniger Methoden enthalten als Interface angegeben von Server.?

5.6. HttpInvokers

HttpInvoker

- **Protocol**
  - HTTP based
  - Binary, using Java Serialization
- **Factories**
  - org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter
  - org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean

Abbildung 5.2.: figure

Hessian

- **Protocol**
  - HTTP based
  - Binary
- **Factories**
  - org.springframework.remoting.caucho.HessianServiceExporter
  - org.springframework.remoting.caucho.HessianProxyFactoryBean

Abbildung 5.3.: figure

Burlap

- **Protocol**
  - HTTP based
  - XML
- **Factories**
  - org.springframework.remoting.caucho.BurlapServiceExporter
  - org.springframework.remoting.caucho.BurlapProxyFactoryBean

Abbildung 5.4.: figure

5.6.1. Config

HTTP Based Protocols

- **web.xml**
  - remoting: Defines Dispatcher Servlet (pattern /*)
- **remoting-servlet.xml**

```
<beans>
   ...
   <bean name="/HttpInvoker" class = "org.springframework.
         remoting.httpinvoker.HttpInvokerServiceExporter">
      <property name="serviceInterface"
                value="ch.fhnw.edu.bank.AccountService" />
      <property name="service" ref="accountService" />
   </bean>
   ...
</beans>
```

Abbildung 5.5.: figure

- **Client Proxy HttpInvoker**

```
<bean id="accountService" class="org.springframework.
         remoting.httpinvoker.HttpInvokerProxyFactoryBean">
   <property name="serviceInterface"
             value="ch.fhnw.edu.bank.AccountService" />
   <property name="serviceUrl"
       value=
     "http://${web.host}:${web.port}/${web.app}/HttpInvoker"/>
</bean>
```

```
web.host=localhost
web.port=8080
web.app=spring-remote
```

Abbildung 5.6.: figure

6. Transaktionen

6.1. Key Design Principles

Separation of Concerns Teile die applikation in eigenartigen Features auf sodass die so disjunkt sind wie möglich.

Single Responsibility Principle Jede Komponent oder Modul sollte nur für ein gegebenes Feature oder Funktionalität verantwortlich sein.

Principle of Least Knowledge Eine Komponente sollte nicht über die interne Details von anderen Komponenten wissen (Law of Demeter) LOD.

Don't Repeat Yourself Nur eine Komponent per Funktionalität.

Avoid doing design upfront Vermeiden dass man zu viel von Anfang an in Design investiert, bleib lieber Flexibel.

Composition over Inheritance Inheritance vermehrt die Abhängigkeit und verhindert Reuse.

6.2. Key Architecture Principles

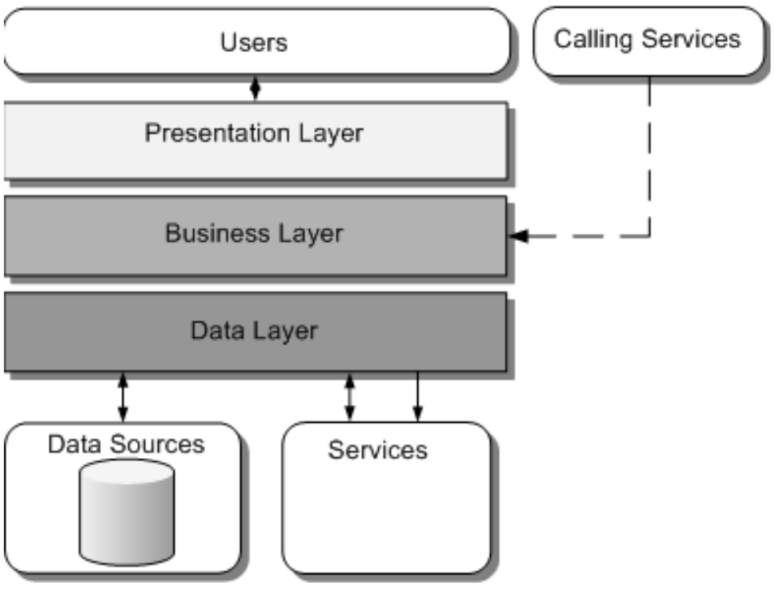
Build to change over Build to Last Design mit der Idee das man es später anpassen kann aufgrund von neuer Requirements.

Modellieren zu Analysieren und Risiken vermeiden Threat Modellen benutzen und UML.

Models und Views als Kollab Tool Effiziente Komm von design Prinzipien und Veränderungen sind Kritisch. Models und andere Visualisierungen Ausnutzen.

Wichtige Ingeneurentscheidungen identifizieren Entscheidungen von Architektur das erste mal richtig treffen.

6.3. SchichtenSystem



Splitting an application into separate layers that have distinct roles and functionalities helps you to maximize maintainability of the code, optimize the way that the application works when deployed in different ways, and provide a clear delineation between locations where certain technology or design decisions must be made.

from MS Application Architecture Guide

Abbildung 6.1.: figure

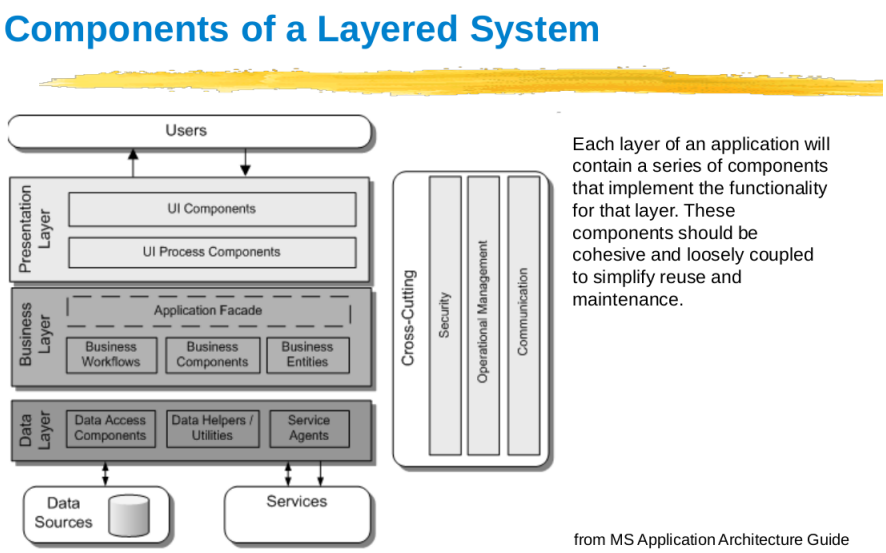


Abbildung 6.2.: figure

6.4. Enterprise App Design

Serviceschicht Komponenten bieten andere Klienten und Apps eine möglichkeit an, Businesslogik zuzugrefen und Appfunktionalität auszunutzen mittels Messagepassing. Applikationen müssen oftmals mehere arten von Clients unterstützen :

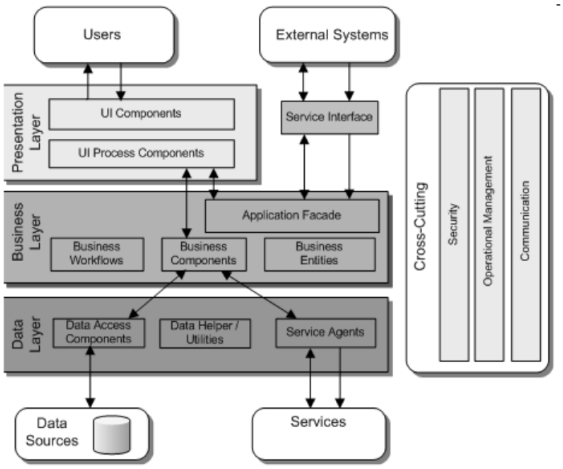


Abbildung 6.3.: figure

6.4.1. OO Design

Businesslogik soll eine Netzwerk von relativ kleinen Klassen sein und sollte mit die Konzepten der Domänenmodel übereinstimmen.

> ... Ä Domain Model creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form."

6.4.2. Enkapselung

Muss entschieden was Clients aufrufen können. - Interface von BL.

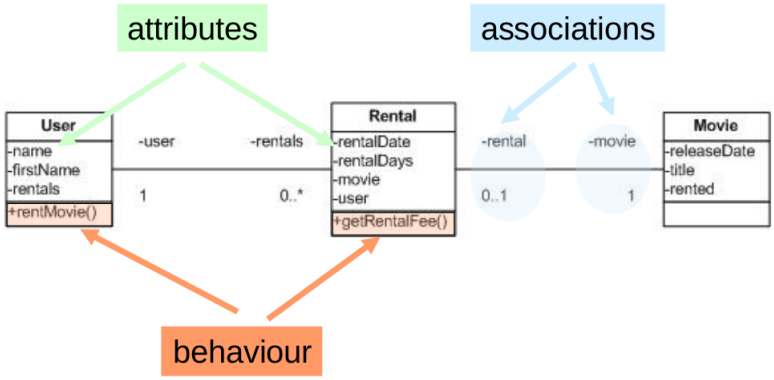


Abbildung 6.4.: figure

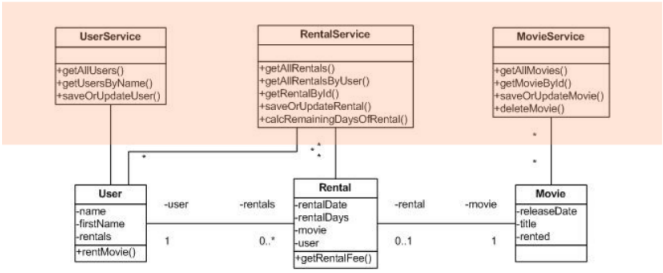


Abbildung 6.5.: Service Beispiel

\+ Maintainability : Implementierung muss nicht Presentation beeinflussen.

\- Viel Code.

Optionen Pojo Facade oder Domain Model öffentlich machen.

6.4.3. POJO Facade

\+ Entwicklung vereinfacht.

\+ Businesslogik kann auserhalb Container getestet werden.

\+ Deklarative Transaktionen und Sicherheit bevor Präsentationsschicht.

\+ Pojo Facade kann Domain Obj züruckgeben und nicht dtos.

\+ Dependency Injection.

\- Lazy Loading

\- Verteilte Transaktionen

6.4.4. Exposed Model Pattern

\+ Leicht zu verstehen.

\+ Leichte Navigation in Präsentationsschicht.

\- Code

\- LazyLoading Exception

\- DB Verwaltung in Präsentationsschicht.

\- Transaktionen im Präsentationsschicht.

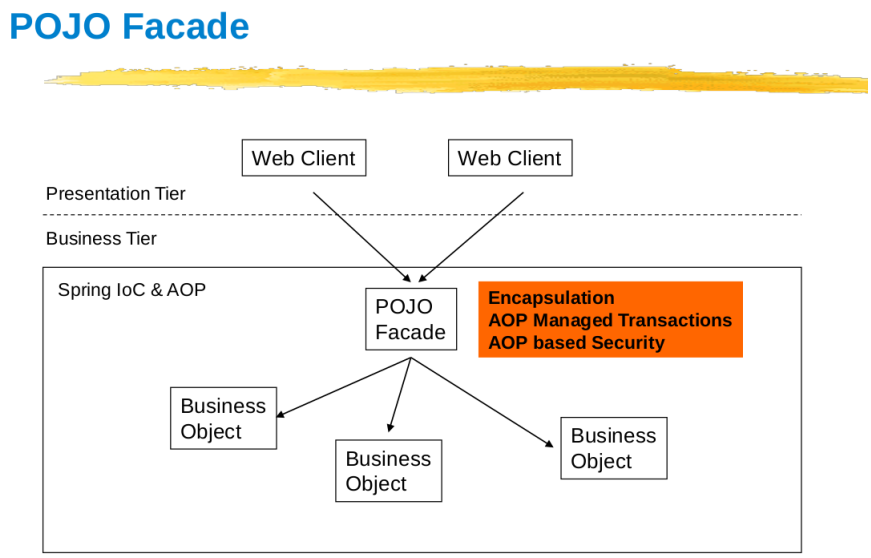


Abbildung 6.6.: figure

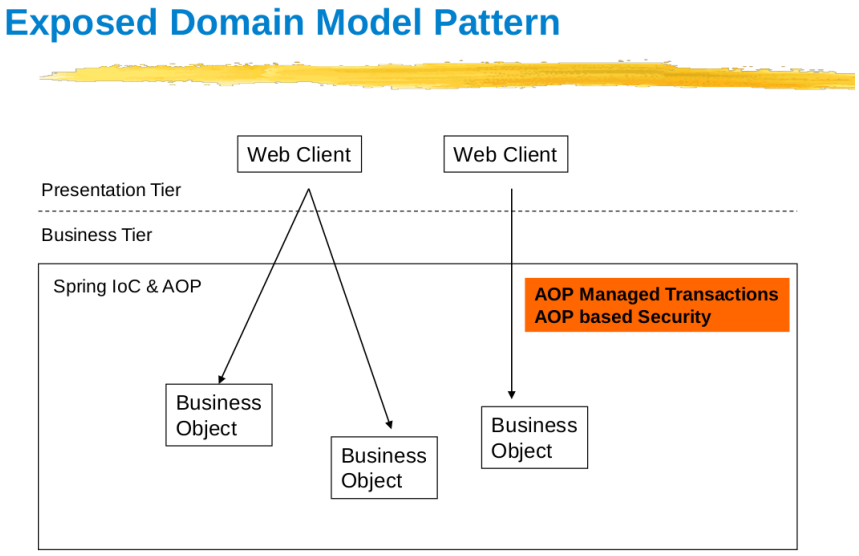


Abbildung 6.7.: figure

6.5. Transaktionengrenzen

Präsentationsschicht

- Servlet filters
- Konsistente ansicht der datenbank
- JTA und lokaltransaktionen unterstützt.
- Buffering overhead nötig für Rollback.
- Präsentationscode ist kompliziert
- **Programmatic Transaction Model**

BusinessSchicht

- Transaktion Interceptors
- Keine Konsistenz in transaktionen, zugriff auf Lazy geladene Objekten ausserhalb eine Transkaktion
- **Deklarative Transaktionsmodell**

6.6. Warum Transaktionen?

- Mehrere Datenbanksysteme sind zugegriffen worden für 1 Request.

- Mehere Datenbankzugriffe innerhalb 1 Request.
- Verschiedene verteilte Systeme werden zugegriffen für eine Request.

- Braucht Korrektheit und Recovery Guarantieen.
- Applikationsebene wird extrem kompliziert und Debug wird schwierig.

6.7. ACID

Atomicity means that a transaction is considered complete if and only if all of its operations were performed successfully. If any operation in a transaction fails, the transaction fails.

Consistency means that a transaction must transition data from one consistent state to another, preserving the data's semantic and referential integrity. While applications should always preserve data consistency, many databases provide ways to specify integrity and value constraints so that transactions that attempt to violate consistency will automatically fail.

Isolation means that any changes made to data by a transaction are invisible to other concurrent transactions until the transaction commits. Isolation requires that several concurrent transactions must produce the same results in the data as those same transactions executed serially, in some (unspecified) order.

Durability means that committed updates are permanent. Failures that occur after a commit cause no loss of data. Durability also implies that data for all committed transactions can be recovered after a system or media failure.

6.8. Transaktionsdefinitonen

Propagieren A transaction attribute controls the scope of a transaction – Example: When method-B executes, does it run within the scope of the transaction started by method-A, or does it execute with a new transaction?

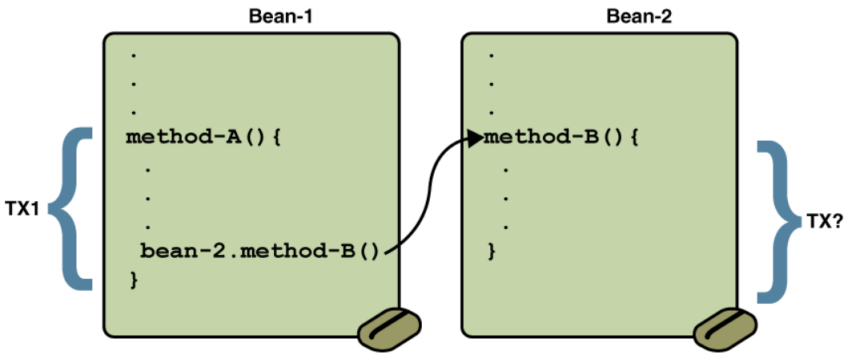


Abbildung 6.8.: figure

Isolation In database systems, isolation is a property that the changes made by an operation are not visible to other simultaneous operations on the system until its completion.

7. Transaktions Strategieen

7.1. Theorie

Konsistenz durch Transaktion Mit Transaktionen im engeren technischen Sinne ist gemeint, dass mehrere Arbeitsschritte zusammengefasst werden und entweder alle ausgeführt werden, oder alle nicht ausgeführt werden (Alles-oder-nichts-Prinzip). Wenn Geld von einem Konto auf ein anderes gebucht werden soll, dann darf es nicht vorkommen, dass die Abbuchung vom ersten Konto erfolgreich durchgeführt wird, aber die Gutschrift auf das zweite Konto fehlschlägt. Transaktionen stellen Konsistenz sicher.

Begin, Commit, Rollback, Transaktionsklammer, LUW
Transaktionen werden durch ein "BeginKommando gestartet. Konnten die einzelnen Arbeitsschritte erfolgreich durchgeführt werden, werden sie durch ein CommitKommando endgültig bestätigt (und für andere Prozesse sichtbar). Gab es einen Fehler, werden sie durch ein RollbackKommando zurückgesetzt. Das Begin-Startkommando und das beendende Commit- bzw. Rollback-Kommando stellt die so genannte Transaktionsklammer dar. Die Arbeitsschritte innerhalb der Transaktion werden auch schon mal als LUW (Logical Unit of Work) bezeichnet.

Dirty Read
Wenn Transaktionen dem ÄCIDPrinzip entsprechen sollen, müssen sie serialisiert nacheinander durchgeführt werden. Aus Performance-Gründen wird hiervon oft abgewichen und ein niedrigerer Transaction Isolation Leveleingestellt. Das kann zu folgenden Fehlern führen: Dirty Read: Es können von anderen Transaktionen geschriebene Daten gelesen werden, für die noch kein Commitërfolgte und die eventuell per Rollbackßurückgesetzt werden.

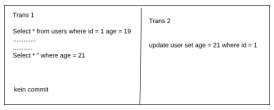


Abbildung 7.1.: figure

Non-repeatable Read (Stale Data): Während einer laufenden Transaktion können Daten von anderen Transaktionen geändert und committed werden, so dass in der ersten Transaktion ein zweites Auslesen zu anderen Daten führt.

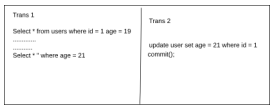


Abbildung 7.2.: figure

Phantom Read
Eine erste Transaktion liest über eine "Where-Klauseleine Liste von Datensätzen. Eine zweite Transaktion fügt weitere Datensätze hinzu (inkl. Commit). Wenn die erste Transaktion wieder über die gleiche "Where-Klausel"Datensätze liest oder bearbeitet, gibt es mehr Datensätze als vorher.

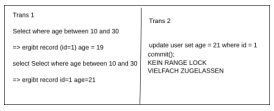


Abbildung 7.3.: figure

7.2. Transaction Isolation Levels

In Datenbanken können verschiedene so genannte Transaction Isolation Leveleingestellt werden, um den besten Kompromiss zwischen Isolation und Performance vorzugeben. Folgende Transaction Isolation Levelßind in ANSI-SQL2 definiert:

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

In Spring ist der Isolation Level auf DEFAULT gesetzt. Das bedeutet: Aufgabe der Datenbank - Read Committed in unsere Beispiel.

7.3. Transaktionen Propagieren

In einigen Transaktionsmanagern kann die Transaction Propagationpro 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.

7.4. Transaktionen mit Spring

Um das Transaktionshandling in einer Applikation einzuführen, gibt es im Spring-Umfeld mehrere Möglichkeiten. Die meistverwendete Variante nutzt die Möglichkeiten der Annotation.

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).

@Transactional Annotation anwenden Die @Transactional Annotation macht das Transaktionshandling in Spring sehr einfach. Es genügt die entsprechenden Methoden mit dieser Annotation zu kennzeichnen und das Transaktionshandling (Transaktion starten und am Ende der Methode commit aufrufen) wird automatisch durchgeführt.

Listing 7.1: Transaktional über eine Klasse

```
@Transactional
public interface DAO<T> {
T getById(Long id);
List<T> getAll();
void saveOrUpdate(T t);
void delete(T t);
}
```

Voraussetzungen für das korrekte Verhalten sind: 1. Die entsprechenenden Methoden müssen mit @Transactional annotiert und sie müssen publicßein. 2. Das Transaktionshandling muss in Spring konfiguriert sein.

7.5. Bemerkungen zu Transaktionsstrategie

Configuration 1. Braucht Klasse Annotation, 2. Braucht richtige Spring Konfiguration:

Listing 7.2: spring config transaction

```
<context:component-scan base-package="ch.fhnw.edu.rental"/>

 <tx:annotation-driven transaction-manager="txManager"/>

<bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>

<bean
  class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"
       />

  <bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
  p:dataSource-ref="dataSource" p:jpaVendorAdapter-ref="jpaAdapter">
  <property name="jpaProperties">
    <props>
      <prop key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>
      <prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
    </props>
  </property>
</bean>
```

Beispiel Propagation `@Transactional(propagation=Propagation.MANDATORY)`

Wo Transactional Überall nicht, aber sollte in service stattfinden und mit ein LUW passieren. **Default propagation = Required**

Lesende Transaktionen Bei DAO Methoden die nur lesen mach propagation=Supports.

Exceptionhandling Checked = kein Rollback, Entwickler, Runtime = User, rollback.

8. Aspektorientierte Programmierung (AOP)

Listing 8.1: AOP Config

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

Cross Cutting Code sollte soweit von der Applikation weg abstrahiert werden wie möglich. Cross cutting code ist code der sich auf Sicherheit, Komm oder Verwaltung bezieht. Integrierung in BL kann Design und Warbarkeit sehr schwierig machen. Frameworks helfen damit.

8.1. Definitionen

Cross Cutting Concern Verhalten mit Data die über die Schichten hinaus benutzt werden kann. Kann nicht lokalisiert/in einen Modul verschoben werden..Sicherheit,Transaktionsmanagement,Tracing.

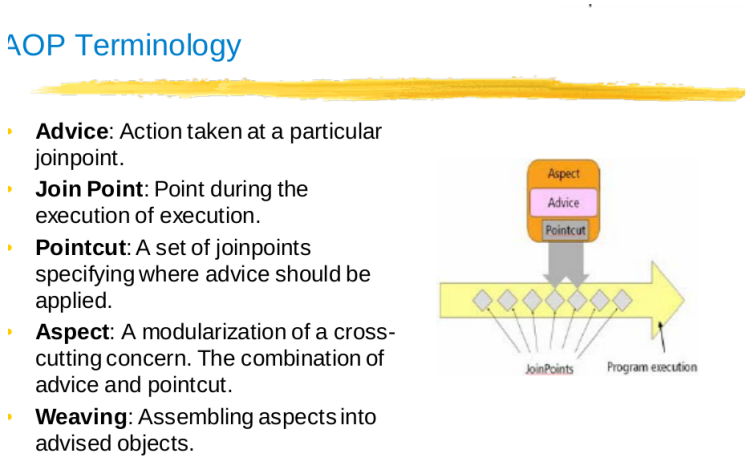


Abbildung 8.1.: AOP Terminologie

8.2. Advice Typen

Before : Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

After: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).

After Returning : Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception. An after returning advice has access to the return value (which it cannot modify), invoked method, methods arguments and target.

After Throwing : Advice to be executed if a method exits by throwing an exception.

Around : Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Spring benutzt Dynamische AOP Proxies um Cross cutting Unterstützung anzubieten. Field interception fehlt.

8.2.1. AOP Implementierungen

Compile Time Source Code während kompilieren modifiziert. AspectJ

Runtime - Byte Injection Class geladen, Subklasse genierert mit aspects - cgilib.

Runtime - JDK 1.3 Dynamic Proxy Proxy Object, same interface intercepts and wraps method calls.

8.2.2. Sprig AOP Architecture

- Proxybasiert
- ProxyFactory für Advised Instanzen.
- Proxyfactory nimmt object + aspects gibt Proxy züruck.
- Programmatische Ansatz.
- ProxyFactoryBean . Deklarative Proxy Erzeugung.
- Spring benutzt JDK Dyn Proxies / cgilib um Proxies zu erzeugen.

  1 Interface oder mehr JDK Dyn Proxy

  Keine Interfaces Cgilib Proxy.

8.3. AOP Calls

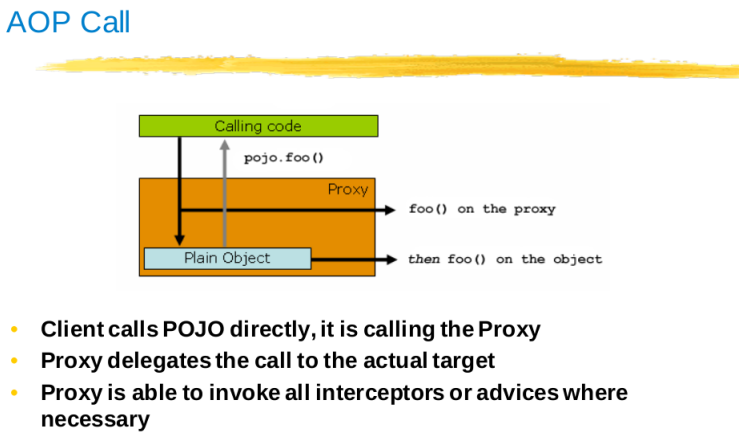


Abbildung 8.2.: figure

Once the call has finally reached the target all subsequent calls on the object itself are going to be invoked against the actual target, and not the proxy.

8.3.1. Programmatische Proxies

Listing 8.2: JDKProxy

```
ProxyFactory factory = new ProxyFactory(new SimplePojo());
factory.addAdvice(new AfterAdvice());
factory.addInterface(Pojo.class);
Pojo pojo = (Pojo) factory.getProxy();
pojo.foo();
```

Listing 8.3: CGIlib Proxy

```
ProxyFactory factory = new ProxyFactory(new SimplePojo());
factory.addAdvice(new AfterAdvice());
factory.setProxyTargetClass(true);
Pojo pojo = (Pojo) factory.getProxy();
pojo.foo();
```

8.4. AOP Development

1. AOP Model : XML / Annotation - AspectJ Support
2. Advice : advice logik + type.
3. Pointcut : Definieren mit Expr Language.

8.4.1. Aspect J Expression Language

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

8.4.2. AspectJ Support Beispiel

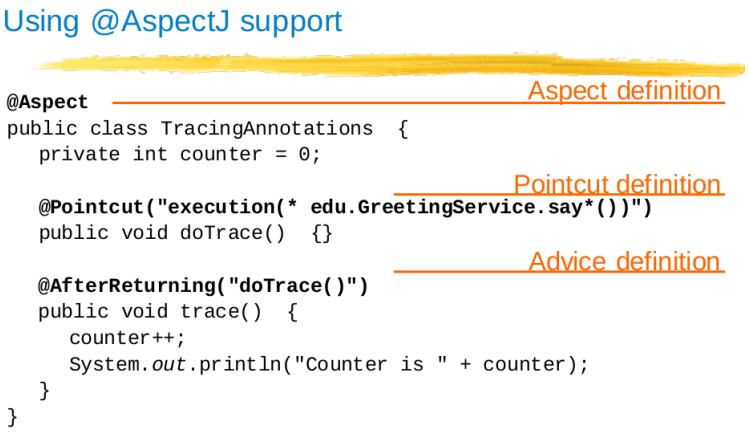


Abbildung 8.3.: figure

8.4.3. Enable Aspect J

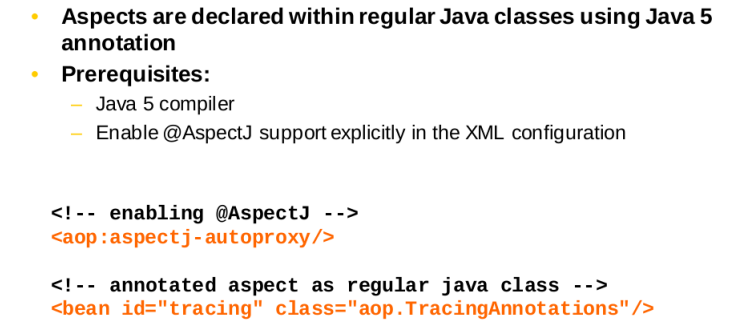


Abbildung 8.4.: figure

8.4.4. AspectJ DI

Listing 8.4: Aspect J Depedency Injection

```
 @Aspect
@Component
// add <component-scan>
public class TracingAnnotations {
@Autowired
private Statistic statistic;
@Before("execution(* edu.GreetingService.say*())")
public void trace() {
// do something with the statistic bean
}
}
```

8.5. Vorteile Nachteile

+ reduces code.
+ Wartbarkeit
+ Flexibel Definitionen - XML / Annotationen.

- Tool Support, Ausbildung
- Warbarkeit nicht richtig, unvorhersehrbare Verhalten (änderung Methodenname).
- Entwickler – unvorhersehrbare Verhalten - Upgrade von Legacy.
- Sicherheitsproblemen - AOP unterstutzt keine Weaving.

8.6. Spring configuration

```
<aop:aspectj-autoproxy>
```

Teil II.

Code, Beispiele Übungen

9. JPA Complete

Listing 9.1: Movie

```
package ch.fhnw.edu.rental.model;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table(name = "movies")
@NamedQueries({
  @NamedQuery(name="movie.all", query="from Movie"),
  @NamedQuery(name="movie.byTitle", query="from Movie m where m.title = :title")
})
public class Movie implements Serializable {
  @Id
  @GeneratedValue
  @Column(name = "movie_id")
  private Long id;

  @Column(name = "movie_title")
  private String title;

  @Column(name = "movie_rented")
  private boolean rented;

  @Column(name = "movie_releasedate")
  private Date releaseDate;

  @ManyToOne
  @JoinColumn(name = "pricecategory_fk")
  private PriceCategory priceCategory;

  Movie() {
    // JPA requires that a default constructor is available
  }

  public Movie(String title, Date releaseDate, PriceCategory priceCategory) throws
      NullPointerException {
    if ((title == null) || (releaseDate == null) || (priceCategory == null)) {
      throw new NullPointerException("not all input parameters are set!");
    }
    this.title = title;
    this.releaseDate = releaseDate;
    this.priceCategory = priceCategory;
    this.rented = false;
  }

  public PriceCategory getPriceCategory() {
    return priceCategory;
  }

  public void setPriceCategory(PriceCategory priceCategory) {
    this.priceCategory = priceCategory;
  }

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

```
    }

    public Date getReleaseDate() {
      return releaseDate;
    }

    public boolean isRented() {
      return rented;
    }

    public void setRented(boolean rented) {
      this.rented = rented;
    }

    public Long getId() {
      return id;
    }

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

  }
```

Listing 9.2: Price Category

```
  package ch.fhnw.edu.rental.model;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.DiscriminatorColumn;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table(name = "pricecategories")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorValue("PriceCategory")
@DiscriminatorColumn(name = "pricecategory_type")
@NamedQueries({
  @NamedQuery(name="pricecategory.all", query="from PriceCategory")
})
public abstract class PriceCategory implements Serializable {
  @Id
  @GeneratedValue
  @Column(name = "pricecategory_id")
  private Long id;

  public Long getId() {
    return id;
  }

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

  public abstract double getCharge(int daysRented);

  public int getFrequentRenterPoints(int daysRented) {
    return 1;
  }
}
```

Listing 9.3: PriceCategory Type Children

```
  package ch.fhnw.edu.rental.model;
```

```
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorValue("ChildrenPriceCategory")
public class PriceCategoryChildren extends PriceCategory {

  @Override
  public double getCharge(int daysRented) {
    double result = 1.5;
    if (daysRented > 3) {
      result += (daysRented - 3) * 1.5;
    }
    return result;
  }

  @Override
  public String toString() {
    return "Children";
  }
}
```

Listing 9.4: Rental

```
package ch.fhnw.edu.rental.model;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorValue("ChildrenPriceCategory")
public class PriceCategoryChildren extends PriceCategory {

  @Override
  public double getCharge(int daysRented) {
    double result = 1.5;
    if (daysRented > 3) {
      result += (daysRented - 3) * 1.5;
    }
    return result;
  }

  @Override
  public String toString() {
    return "Children";
  }
}
```

Listing 9.5: Role

```
package ch.fhnw.edu.rental.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "roles")
public class Role {
  @Id
  @GeneratedValue
  @Column(name = "role_id")
  private Long id;
```

```
    @Column(name = "role_roleName", unique=true)
18  private String roleName;

    public Role() {
    }

23  public String getRoleName() {
      return roleName;
    }

    public void setRoleName(String roleName) {
28    this.roleName = roleName;
    }

    public Long getId() {
      return id;
33  }

    protected void setId(Long id) {
      this.id = id;
    }
38 }
```

Listing 9.6: User

```
1  package ch.fhnw.edu.rental.model;

  import javax.persistence.Column;
  import javax.persistence.Entity;
  import javax.persistence.GeneratedValue;
6 import javax.persistence.Id;
  import javax.persistence.Table;

  @Entity
  @Table(name = "roles")
11 public class Role {
    @Id
    @GeneratedValue
    @Column(name = "role_id")
    private Long id;
16
    @Column(name = "role_roleName", unique=true)
    private String roleName;

    public Role() {
21  }

    public String getRoleName() {
      return roleName;
    }
26
    public void setRoleName(String roleName) {
      this.roleName = roleName;
    }

31  public Long getId() {
      return id;
    }

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

9.1. Services

Listing 9.7: MovieService

```
1  package ch.fhnw.edu.rental.services.impl;

  import java.util.List;

  import org.apache.commons.logging.Log;
```

```
import org.apache.commons.logging.LogFactory;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import ch.fhnw.edu.rental.daos.MovieDAO;
import ch.fhnw.edu.rental.daos.PriceCategoryDAO;
import ch.fhnw.edu.rental.daos.impl.ManagedDAO;
import ch.fhnw.edu.rental.model.Movie;
import ch.fhnw.edu.rental.model.PriceCategory;
import ch.fhnw.edu.rental.service.RentalServiceException;
import ch.fhnw.edu.rental.services.MovieService;

@Transactional
public class MovieServiceImpl implements MovieService {
  private Log log = LogFactory.getLog(this.getClass());

  private MovieDAO movieDAO;
  private PriceCategoryDAO priceCategoryDAO;


  public void setPriceCategoryDAO(PriceCategoryDAO priceCategoryDAO) {
    this.priceCategoryDAO = priceCategoryDAO;
  }

  public void setMovieDAO(MovieDAO movieDAO) {
    this.movieDAO = movieDAO;
  }

  @Transactional(propagation = Propagation.SUPPORTS)
  public Movie getMovieById(Long id) throws RentalServiceException {
    return movieDAO.getById(id);
  }

  @Transactional(propagation = Propagation.SUPPORTS)
  public List<Movie> getAllMovies() throws RentalServiceException {
    List<Movie> movies = movieDAO.getAll();
    if (log.isDebugEnabled()) {
      log.debug("getAllMovies() done");
    }
    return movies;
  }

  @Transactional(propagation = Propagation.SUPPORTS)
  public List<Movie> getMoviesByTitle(String title) throws RentalServiceException {
    return movieDAO.getByTitle(title);
  }

  public void saveOrUpdateMovie(Movie movie) throws RentalServiceException {
    if (movie == null) {
      throw new RentalServiceException("'movie' parameter is not set!");
    }
    movieDAO.saveOrUpdate(movie);
//    throw new Exception("just for testing");
    if (log.isDebugEnabled()) {
      log.debug("saved or updated movie[" + movie.getId() + "]");
    }
  }


  @SuppressWarnings("unchecked")
  public void deleteMovie(Movie movie) throws RentalServiceException {
    if (movie == null) {
      throw new RentalServiceException("'movie' parameter is not set!");
    }
    if (movie.isRented()) {
      throw new RentalServiceException("movie is still used");
    }

    if(movieDAO instanceof ManagedDAO<?>) {
      movie = ((ManagedDAO<Movie>)movieDAO).manage(movie);
    }

    movieDAO.delete(movie);

    if (log.isDebugEnabled()) {
      log.debug("movie[" + movie.getId() + "] deleted");
```

```
    }
  }

  public List<PriceCategory> getAllPriceCategories() throws RentalServiceException {
    return priceCategoryDAO.getAll();
  }

}
```

Listing 9.8: RentalOverdueService

```
package ch.fhnw.edu.rental.services.impl;

import java.util.Calendar;
import java.util.Date;
import java.util.List;

import javax.mail.internet.MimeMessage;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import ch.fhnw.edu.rental.model.Rental;
import ch.fhnw.edu.rental.service.RentalServiceException;
import ch.fhnw.edu.rental.services.MailService;
import ch.fhnw.edu.rental.services.RentalOverdueService;
import ch.fhnw.edu.rental.services.RentalService;

@Component("rentalOverdueService")
public class RentalOverdueServiceImpl implements RentalOverdueService  {
  private Log log = LogFactory.getLog(RentalOverdueServiceImpl.class);

  @Autowired
  private RentalService rentalService;

  @Autowired
  private MailService mailService;

  @Value("${jobs.simulation}")
  private boolean simulation;

  @Value("${mail.reminder.template}")
  private String template;

  @Override
  @Scheduled(fixedRate = 10000)
  public void checkRentals() throws RentalServiceException {
    Date today;
    List<Rental> rentals = rentalService.getAllRentals();
    Calendar cal = Calendar.getInstance();
    today = cal.getTime();
    for (Rental rental : rentals) {
      cal.setTime(rental.getRentalDate());
      cal.add(Calendar.DAY_OF_YEAR, rental.getRentalDays());
      Date dueDate = cal.getTime();
      if (today.after(dueDate)) {
        if (! simulation) {
          MimeMessage message = mailService.prepareMailMessage(rental, template);
          mailService.sendMailMessage(message);
          log.info("Reminder for user '" + rental.getUser().getLastName()
              + " " + rental.getUser().getFirstName() + "' sent");
        } else {
          log.debug("Simulation: Reminder for user '" + rental.getUser().getLastName()
              + " " + rental.getUser().getFirstName() + "' sent");
        }
      }
    }
  }
}
```

Listing 9.9: MailService

```
package ch.fhnw.edu.rental.services.impl;

import java.util.Map;

import javax.mail.MessagingException;
import javax.mail.internet.MimeMessage;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;
import org.springframework.stereotype.Component;

import ch.fhnw.edu.rental.model.Rental;
import ch.fhnw.edu.rental.service.RentalServiceException;
import ch.fhnw.edu.rental.services.MailService;
import ch.fhnw.edu.rental.services.impl.util.MailModelHelper;
import ch.fhnw.edu.rental.services.impl.util.MailTemplateHelper;

@Component
public class MailServiceImpl implements MailService {
  private Log log = LogFactory.getLog(MailServiceImpl.class);

  private JavaMailSender mailSender;
  private String mailFromAddress;
  private String mailSubject;
  private MailModelHelper mailModelHelper;
  private MailTemplateHelper mailTemplateHelper;

  public void setMailSubject(String mailSubject) {
    this.mailSubject = mailSubject;
  }

  public void setMailFromAddress(String mailFromAddress) {
    this.mailFromAddress = mailFromAddress;
  }

  public void setMailSender(JavaMailSender mailSender) {
    this.mailSender = mailSender;
  }

  public void setMailModelHelper(MailModelHelper mailModelHelper) {
    this.mailModelHelper = mailModelHelper;
  }

  public void setMailTemplateHelper(MailTemplateHelper mailTemplateHelper) {
    this.mailTemplateHelper = mailTemplateHelper;
  }

  @Override
  public MimeMessage prepareMailMessage(final Rental rental, final String template) throws
      RentalServiceException
{
    try {
      MimeMessage mimeMessage = mailSender.createMimeMessage();
      MimeMessageHelper helper = new MimeMessageHelper(mimeMessage, "UTF-8");
      helper.setTo(rental.getUser().getEmail());
      helper.setFrom(mailFromAddress);
      helper.setSubject(mailSubject);
      Map<String, Object> model = mailModelHelper.fillModel(rental);
      String textMessage = mailTemplateHelper.mergeTemplate(model, template);
      helper.setText(textMessage);
      log.debug("Mail prepared for '" + rental.getUser().getLastName() + " " +
rental.getUser().getFirstName());
      return helper.getMimeMessage();
    } catch (MessagingException e) {
      throw new RentalServiceException(e.getMessage());
    }
  }

  @Override
  public void sendMailMessage(MimeMessage message) {
    mailSender.send(message);
    log.debug("Mail sent successfully");
```

```
  }
}
```

Listing 9.10: RentalService

```
package ch.fhnw.edu.rental.services.impl;

import java.util.Date;
import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import ch.fhnw.edu.rental.daos.MovieDAO;
import ch.fhnw.edu.rental.daos.RentalDAO;
import ch.fhnw.edu.rental.daos.impl.ManagedDAO;
import ch.fhnw.edu.rental.model.Rental;
import ch.fhnw.edu.rental.service.RentalServiceException;
import ch.fhnw.edu.rental.services.RentalService;

@Transactional
public class RentalServiceImpl implements RentalService {
  private Log log = LogFactory.getLog(this.getClass());

  private RentalDAO rentalDAO;

  public void setRentalDAO(RentalDAO rentalDAO) {
    this.rentalDAO = rentalDAO;
  }

  private MovieDAO movieDAO;

  public void setMovieDAO(MovieDAO movieDAO) {
    this.movieDAO = movieDAO;
  }

  @Override
  @Transactional(propagation = Propagation.SUPPORTS)
  public List<Rental> getAllRentals() throws RentalServiceException {
    List<Rental> rentals = rentalDAO.getAll();
    if (log.isDebugEnabled()) {
      log.debug("getAllRentals() done");
    }
    return rentals;
  }

  @Override
  @Transactional(propagation = Propagation.SUPPORTS)
  public int calcRemainingDaysOfRental(Rental rental, Date date) {
    return rental.calcRemainingDaysOfRental(date);
  }

  @Override
  @Transactional(propagation = Propagation.SUPPORTS)
  public Rental getRentalById(Long id) {
    return rentalDAO.getById(id);
  }

  @SuppressWarnings("unchecked")
  @Override
  public void deleteRental(Rental rental) throws RentalServiceException {
    if (rental == null) {
      throw new RentalServiceException("'rental' parameter is not set!");
    }
    if(rentalDAO instanceof ManagedDAO<?>) {
      rental = ((ManagedDAO<Rental>)rentalDAO).manage(rental);
    }

    rental.getUser().getRentals().remove(rental);
    rental.getMovie().setRented(false);

    rentalDAO.delete(rental);
```

```
      if( !(movieDAO instanceof ManagedDAO<?>)){
        movieDAO.saveOrUpdate(rental.getMovie());
      }

      if (log.isDebugEnabled()) {
        log.debug("rental[" + rental.getId() + "] deleted");
      }
    }
}
```

Listing 9.11: UserService

```
package ch.fhnw.edu.rental.services.impl;

import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import ch.fhnw.edu.rental.daos.MovieDAO;
import ch.fhnw.edu.rental.daos.RentalDAO;
import ch.fhnw.edu.rental.daos.UserDAO;
import ch.fhnw.edu.rental.daos.impl.ManagedDAO;
import ch.fhnw.edu.rental.model.Movie;
import ch.fhnw.edu.rental.model.Rental;
import ch.fhnw.edu.rental.model.User;
import ch.fhnw.edu.rental.service.RentalServiceException;
import ch.fhnw.edu.rental.services.UserService;

@Transactional
public class UserServiceImpl implements UserService {
  private Log log = LogFactory.getLog(this.getClass());

  private UserDAO userDAO;
  private RentalDAO rentalDAO;
  private MovieDAO movieDAO;

  public void setUserDAO(UserDAO userDAO) {
    this.userDAO = userDAO;
  }

  public void setRentalDAO(RentalDAO rentalDAO) {
    this.rentalDAO = rentalDAO;
  }

  public void setMovieDAO(MovieDAO movieDAO) {
    this.movieDAO = movieDAO;
  }

  @Override
  @Transactional(propagation = Propagation.SUPPORTS)
  public User getUserById(Long id) throws RentalServiceException {
    User user = this.userDAO.getById(id);
    return user;
  }

  @Override
  @Transactional(propagation = Propagation.SUPPORTS)
  public List<User> getAllUsers() throws RentalServiceException {
    List<User> users = userDAO.getAll();
    if (log.isDebugEnabled()) {
      log.debug("getAllUsers() done");
    }
    return users;
  }

  @Override
  public void saveOrUpdateUser(User user) throws RentalServiceException {
    userDAO.saveOrUpdate(user);
    if (log.isDebugEnabled()) {
      log.debug("saved or updated user[" + user.getId() + "]");
    }
  }
```

```
65    @Override
      @SuppressWarnings("unchecked")
      public void deleteUser(User user) throws RentalServiceException {
        if (user == null) {
          throw new RentalServiceException("'user' parameter is not set!");
70      }
        if(userDAO instanceof ManagedDAO<?>) {
          user = ((ManagedDAO<User>)userDAO).manage(user);
        }

75      userDAO.delete(user); // if (user.getRentals().size()>0) associated rentals
                    // have to be deleted by userDAO.delete(user) as well
        if (log.isDebugEnabled()) {
          log.debug("user[" + user.getId() + "] deleted");
        }
80    }

      @Override
      @Transactional(propagation = Propagation.SUPPORTS)
      public List<User> getUsersByName(String name) throws RentalServiceException {
85      List<User> users = userDAO.getByName(name);
        return users;
      }

      @Override
90    @SuppressWarnings("unchecked")
      public Rental rentMovie(User user, Movie movie, int days)
          throws RentalServiceException {
        if (user == null)
          throw new IllegalArgumentException("parameter 'user' is null!");
95      if (movie == null)
          throw new IllegalArgumentException("parameter 'movie' is null!");
        if (days < 1)
          throw new IllegalArgumentException("parameter 'days' must be > 0");

100     // In case of managed daos, the detached entities are merged first.
        if(movieDAO instanceof ManagedDAO<?>) {
          movie = ((ManagedDAO<Movie>)movieDAO).manage(movie);
        }
        if(userDAO instanceof ManagedDAO<?>) {
105       user = ((ManagedDAO<User>)userDAO).manage(user);
        }

        Rental rental = new Rental(user, movie, days);
        rentalDAO.saveOrUpdate(rental);
110     // the constructor of rental changed the movie to rented, this change must
        // be persisted in case of non managed DAOs.
        if(! (movieDAO instanceof ManagedDAO)){
          movieDAO.saveOrUpdate(movie);
        }
115     // Similarly, the user has to be saved as he refers to an additional rental
        // in case that the user is not managed and in case that the user is the
        // owner of the association.
        // if(! (userDAO instanceof ManagedDAO)) {
        //  userDAO.saveOrUpdate(user);
120     // }

        return rental;
      }

125   @Override
      @SuppressWarnings("unchecked")
      @Transactional(propagation = Propagation.SUPPORTS)
      public void returnMovie(User user, Movie movie)
          throws RentalServiceException {
130     if (user == null)
          throw new IllegalArgumentException("parameter 'user' is null!");
        if (movie == null)
          throw new IllegalArgumentException("parameter 'movie' is null!");
        if(userDAO instanceof ManagedDAO<?>) {
135       user = ((ManagedDAO<User>)userDAO).manage(user);
        }
        if(movieDAO instanceof ManagedDAO<?>) {
          movie = ((ManagedDAO<Movie>)movieDAO).manage(movie);
        }
```

```
140
        Rental rentalToRemove = null;
        for (Rental rental : user.getRentals()) {
          if (rental.getMovie().equals(movie)) {
            rentalToRemove = rental;
145         break;
          }
        }

        user.getRentals().remove(rentalToRemove);
150     rentalToRemove.getMovie().setRented(false);
        rentalDAO.delete(rentalToRemove);
      }
    }
```

10. Discussions JPA

10.1. JDBC

Movie: getAll

```
@Override
public List<Movie> getAll() {
   JdbcTemplate template = getJdbcTemplate();
   return template.query("select * from MOVIES",
      new RowMapper<Movie>(){
         @Override
         public Movie mapRow(ResultSet rs, int row)
                                    throws SQLException {
            return createMovie(rs);
         }
      }
   );
}
```

- MovieDAO extends JdbcDaoSupport
  - getJdbcTemplate() returns the template

Movie: createMovie

```
private Movie createMovie(ResultSet rs) throws SQLException {
   long priceCategory = rs.getLong("PRICECATEGORY_FK");
   Movie m = new Movie(
         rs.getString("MOVIE_TITLE"),
         rs.getDate("MOVIE_RELEASEDATE"),
         priceCategoryDAO.getById(priceCategory));
   m.setId(rs.getLong("MOVIE_ID"));
   m.setRented(rs.getBoolean("MOVIE_RENTED"));
   return m;
}
```

- Movie constructor does not accept a primary key
  - User code never provides a PK

Movie: getById

```
@Override
public Movie getById(Long id){
   JdbcTemplate template = getJdbcTemplate();
   return template.queryForObject(
      "select * from MOVIES where MOVIE_ID = ?",
      new RowMapper<Movie>(){
         @Override
         public Movie mapRow(ResultSet rs, int row)
                                    throws SQLException {
            return createMovie(rs);
         }
      },
      id
   );
}
```


Create / Update / Delete

- **SaveOrUpdate**
  - Decide which operation is needed depending on the PK
    - getId() == null => not saved in database => INSERT
    - getId() != null => stored in database => UPDATE

```
@Override
public void saveOrUpdate(Movie movie) {
   JdbcTemplate template = getJdbcTemplate();
   if (movie.getId() == null) {
      // insert in DB
      long id = ...;
      movie.setId(id);
   }
   else {
      // update in DB
   }
}
```

Create / Update / Delete

- **Update**

```
...
else {
   // update
   template.update(
      "UPDATE MOVIES SET MOVIE_TITLE = ?, MOVIE_RENTED = ?,
           MOVIE_RELEASEDATE = ?,PRICECATEGORY_FK = ?
           WHERE MOVIE_ID = ?",
      movie.getTitle(),
      movie.isRented(),
      movie.getReleaseDate(),
      movie.getPriceCategory().getId(),
      movie.getId());
}
...
```


Create / Update / Delete (cont)

- **Delete**

```
@Override
public void delete(Movie movie) {
   JdbcTemplate template = getJdbcTemplate();
   template.update(
      "delete from MOVIES where MOVIE_ID = ?", movie.getId());
   movie.setId(null);
}
```

  - Upon deletion instance has to be marked as fresh, otherwise a subsequent call to saveOrUpdate would not succeed
  - For the deletion of a movie one could check whether the rented-flag is false, otherwise a foreign key constraint exception will be thrown

```
if(movie.isRented()) throw new IllegalStateException();
```

    - But such a test is already performed in the service implementation

PrimaryKey Generation

- **Problem with auto-increment**

```
CREATE MEMORY TABLE MOVIES(MOVIE_ID BIGINT
   GENERATED BY DEFAULT AS IDENTITY(START WITH 1)
   NOT NULL PRIMARY KEY,
```

  - PK has to be read before it can be assigned to the instance
  - How to access the instance if PK is not known?
- **Determine new PK first**
  - Sequence
    - Read next PK from a sequence
  - Read next value

```
template.queryForLong("select max(MOVIE_ID) from MOVIES")+1;
```

    - Not thread-safe
  - Use a singleton PK factory
    - May use a table where the PKs are stored

PrimaryKey Generation

- **Solution: JDBC 3.0 getGeneratedKey functionality**
  - Works with auto-increment

```
SimpleJdbcInsert insert = new SimpleJdbcInsert(getDataSource())
   .withTableName("MOVIES")
   .usingGeneratedKeyColumns("MOVIE_ID");

Map<String,Object> parameters = new HashMap<String, Object>();
parameters.put("MOVIE_TITLE", movie.getTitle());
parameters.put("MOVIE_RELEASEDATE", movie.getReleaseDate());
parameters.put("MOVIE_RENTED", movie.isRented());
parameters.put("PRICECATEGORY_FK",
                     movie.getPriceCategory().getId());

Number id = insert.executeAndReturnKey(parameters);
movie.setId((Long)id);
```

PrimaryKey Generation

- **Solution: call identity() [HSQL specific]**

```
Connection conn = getDataSource().getConnection();

PreparedStatement pst;
pst = conn.prepareStatement("INSERT INTO MOVIES"
 + "(MOVIE_TITLE, MOVIE_RELEASEDATE, MOVIE_RENTED, "
 + " PRICECATEGORY_FK) VALUES (?,?,?,?)");
pst.setString(1, movie.getTitle());
pst.setDate(2, new Date(movie.getReleaseDate().getTime()));
pst.setBoolean(3, movie.isRented());
pst.setLong(4, movie.getPriceCategory().getId());
pst.execute();

Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("call identity()");
rs.next();
movie.setId(res.getLong(1));
```


equals / hashCode

- **Comparison**
  - Same entity may exist in several instances, not found with contains => equals / hashCode must be overridden
- **Implementation of equals & hashCode**
  - Primary Key is unique, but set after creation
    - Entity must not be added to a collection before it is saved or
    - Primary key must not be used in implementation of equals/hashCode
  - hashCode
    - Use final & immutable attributes only (hashCode must not change)
  - equals
    - Compare PKs if both are available (!= null), compare fields otherwise

=> equals/hashCode can be generated by Eclipse

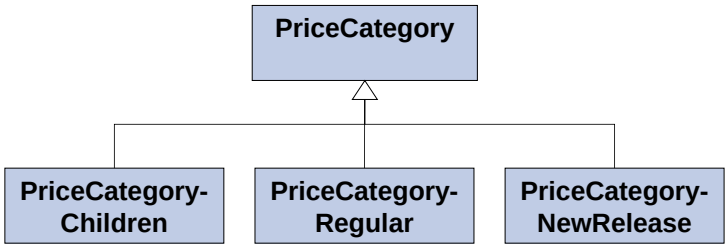
hashCode

```
// Rental.hashCode
public int hashCode() {
   int result = 1;
   …
   result = 31*result + ((user == null) ? 0 : user.hashCode());
   return result;
}
```

```
// User.hashCode
public int hashCode() {
   int result = 1;
   ...
   result = 31*result + ((rentals==null) ? 0 :rentals.hashCode());
   return result;
}
```

  - Problem: Cyclic dependency => StackOverflowException

Inheritance



- **Mapping to database?**

Inheritance

```
private PriceCategory createPriceCategory(ResultSet rs)
                                          throws SQLException {
   String type = rs.getString("PRICECATEGORY_TYPE");
   PriceCategory c = null;
   if("Regular".equals(type)) {
      c = new PriceCategoryRegular();
   }
   else if("Children".equals(type)) {
      c = new PriceCategoryChildren();
   }
   else if("NewRelease".equals(type)) {
      c = new PriceCategoryNewRelease();
   }
   c.setId(rs.getLong("PRICECATEGORY_ID"));
   return c;
}
```

- Java7: case statement on type (but type must not be null)

Inheritance

```
private PriceCategory createPriceCategory(ResultSet rs)
                                          throws SQLException {
   String type = rs.getString("PRICECATEGORY_TYPE");
   PriceCategory c = null;
   switch(type) {
   case "Regular" : c = new PriceCategoryRegular();
   case "Children" :  c = new PriceCategoryChildren();
   case "NewRelease" : c = new PriceCategoryNewRelease();
   }
   c.setId(rs.getLong("PRICECATEGORY_ID"));
   return c;
}
```


Cascade Delete

- **Upon deletion of a user, all associated rentals should be deleted**

```
@Override
public void delete(User user) {
   JdbcTemplate template = getJdbcTemplate();
   for(Rental r : user.getRentals()){
      rentalDAO.delete(r);
   }
   template.update("delete from USERS where USER_ID = ?",
                   user.getId());
   user.setId(null);
}
```

- Do we have to remove the rentals from the deleted instance as well?
  => user.setRentals(new LinkedList<Rentals>());
- Design decision:
  does such cascading delete belong to the DAO or to the UserService?

Cascade Delete

- **Variant**

```
@Override
public void delete(User user) {
   JdbcTemplate template = getJdbcTemplate();
   template.update("delete from RENTALS WHERE USER_ID=?",
                   user.getId());
   template.update("delete from USERS WHERE USER_ID=?",
                   user.getId());
}
```

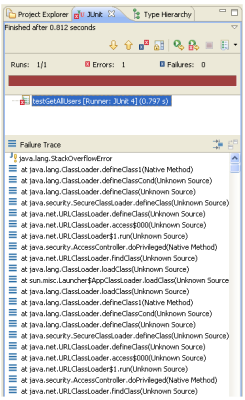
- This way, UserDAO also accesses RENTALS table

Cyclic Dependencies

- **userDao.getById**
  - u.setRentals(rentalDao.getRentalsByUser(u))
- **rentalDao.getRentalsByUser**
  - createRental (for each rental object)
    - movieDao.getById(rs.getLong("MOVIE_ID"))
    - userDao.getById(rs.getLong("USER_ID"))
- **…**

`java.lang.StackOverflowError`

=> general problem of bidirectional associations



Solution 1: Do not load known user

```
@Override
public List<Rental> getRentalsByUser(final User user) {
   JdbcTemplate template = getJdbcTemplate();
   return template.query(
      "select * from RENTALS where USER_ID = ?",
      new RowMapper<Rental>() {
         @Override
         public Rental mapRow(ResultSet rs, int row)
                                          throws SQLException {
            return createRental(rs, user);
         }
      }, user.getId());
}
```

- Disadvantages
  - Has to be implemented individually for each case
  - If a rental is loaded, rental may be added several times to rentals of user

Solution 1: Do not load known user

```
private Rental createRental(ResultSet rs) throws SQLException {
   Long id = rs.getLong("RENTAL_ID");
   User user = userDAO.getById(rs.getLong("USER_ID"));

   //return createRental(rs, user); // do not call this!
   for(Rental r : user.getRentals()){
      if(r.getId().equals(id)) return r;
   }
   throw new RuntimeException("inconsistent user");
}
```


Solution 2: Unification

- **Cyclic Dependencies can be solved with unification**

```
public class ObjectUnifier<T> {
   private Map<Long, WeakReference<T>> cache =
      new HashMap<Long, WeakReference<T>>();
   public T getObject(Long id) {
      if (cache.get(id) != null){
         return cache.get(id).get(); // may be null
      } else {
         return null;
      }
   }
   public void putObject(Long id, T obj) {
      cache.put(id, new WeakReference<T>(obj));
   }
   public void remove(Long id) { cache.remove(id); }
   public void clear() { cache.clear(); }
}
```

Solution 2: Unification

- **createUser**
  - Calls rentalDAO.getRentalsByUser

```
private User createUser(ResultSet rs) throws SQLException {
   Long id = rs.getLong("USER_ID");
   User u = cache.getObject(id);
   if (u == null) {
      u = new User(rs.getString("USER_NAME"),
                   rs.getString("USER_FIRSTNAME"));
      u.setId(id);
      u.setEmail(rs.getString("USER_EMAIL"));
      cache.putObject(id, u);
      u.setRentals(rentalDAO.getRentalsByUser(u));
   }
   return u;
}
```


Solution 2: Unification

- **createRental**
  - Calls userDAO.getById

```
private Rental createRental(ResultSet rs) throws SQLException {
   Long id = rs.getLong("RENTAL_ID");
   Rental r = cache.getObject(id);
   if (r == null) {
      r = new Rental();
      r.setId(id);
      cache.putObject(id, r);
      r.setMovie(movieDAO.getById(rs.getLong("MOVIE_ID")));
      r.setUser(userDAO.getById(rs.getLong("USER_ID")));
      r.setRentalDays(rs.getInt("RENTAL_RENTALDAYS"));
   }
   return r;
}
```


Solution 2: Unification

- **Consequences**
  - equals / hashCode must not necessarily be overwritten
  - Cache has to be cleared, e.g. with an aspect

```
@Aspect
public class CacheAspect {
   private int level = 0;

   @Before("execution(* *..*Service.*(..))")
   public void enterService(){ level++; }

   @After("execution(* *..*Service.*(..))")
   public void exitService() { level--;
      if (level == 0) { cleanup(); }
   }

   private void cleanup(){ ... }
}
```


Summary: Problems which had to be solved

- **Primary Key Generation**
- **Mapping of inheritance**
- **Cyclic Structures**
- **Equals & hashCode**
- **Dependent objects**
  - Deletion of dependent objects
  - Update of dependent objects ???
  - Insert of dependent objects ???

=> JPA addresses all these aspects

10.2. JPA

Fachhochschule Nordwestschweiz

1) Primary Key

a) HSQL AUTO Strategy => IDENTITY

- lab-jpa\target\databases\lab-jpa-db.script

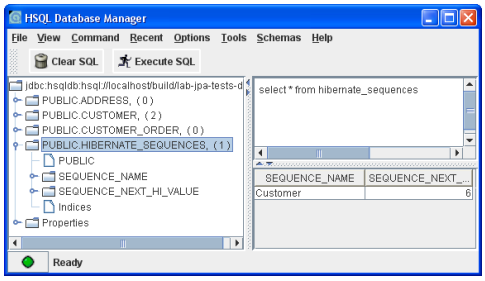
```
CREATE MEMORY TABLE CUSTOMER(
   ID INTEGER
      GENERATED BY DEFAULT AS IDENTITY(START WITH 1)
      NOT NULL
      PRIMARY KEY,
   AGE INTEGER
      NOT NULL,
   NAME VARCHAR(255),
   ADDRESS_ID INTEGER,
   CONSTRAINT FK27FBE3FEAAEE95A3
      FOREIGN KEY(ADDRESS_ID) REFERENCES ADDRESS(ID))
```


Fachhochschule Nordwestschweiz

1) Primary Key

b) TableGeneratedValue(strategy=GenerationType.TABLE)

- PKs: 32768 * SEQUENCE_NEXT_HI_VALUE



Fachhochschule Nordwestschweiz

1) Primary Key

c) Table Generator Annotation

```
@TableGenerator(
   name="generator",                    // name of the generator
   table="ID_GEN",                      // table name
   pkColumnName="GEN_KEY",              // name of key column
   valueColumnName="GEN_VALUE",         // name of value column
   pkColumnValue="CUSTOMER_ID",         // key entry
   allocationSize=10)                   // size of block
@Id
@GeneratedValue(
       strategy=GenerationType.TABLE, generator="generator")
```


Fachhochschule Nordwestschweiz

1) Primary Key

- **Performance comparison**
  - 10'000 insert statements

– AUTO	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) EntityManager Cache

- **Same Entity Manager**

```
Customer c1 = em.find(Customer.class, 1);
Customer c2 = em.find(Customer.class, 1);
```

  - Identical references due to entity cache

- **Different Entity Managers**

```
Customer c1 = emf.createEntityManager().find(Customer.class,1);
Customer c2 = emf.createEntityManager().find(Customer.class,1);
```

  - Different references/instances

3) Lazy Loading

```
@Entity
public class Customer {
   @Id
   private int id;

   @OneToOne
   private Address address;
   ...
```

```
Customer c = em.find(Customer.class, 1);
System.out.println(c.getAddress().getClass().getName());

ch.fhnw.edu.model.Address
```


3) Lazy Loading

```
@Entity
public class Customer {
   @Id
   private int id;

   @OneToOne(fetch=FetchType.LAZY)
   private Address address;
   ...
```

```
Customer c = em.find(Customer.class, 1);
System.out.println(c.getAddress().getClass().getName());

ch.fhnw.edu.model.Address_$$_javassist_2
```

=> Address is a proxy which knows how to load the data

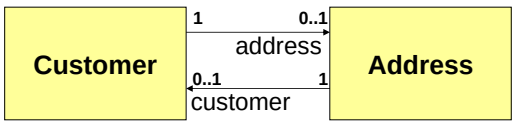
3) Lazy Loading

- **Byte-Code manipulation engine can be specified**
  - hibernate.bytecode.provider = javaassist [default]
    - ch.fhnw.edu.model.Address_$$_javassist_2
  - hibernate.bytecode.provider = cglib [deprecated]
    - ch.fhnw.edu.model.Address$$EnhancerByCGLIB$$8cbef091

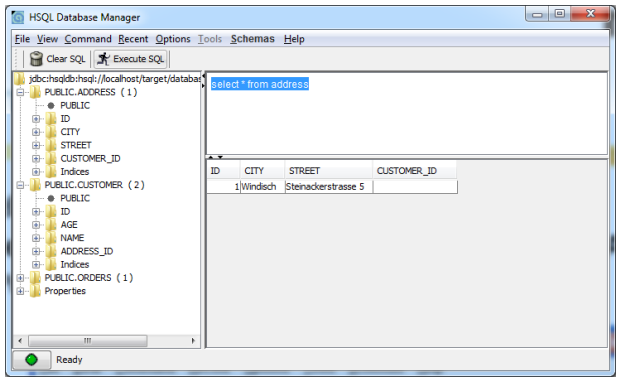
- **Specification**
  - persistence.xml
    - <property name=*"hibernate.bytecode.provider" value="cglib"*/>
  - hibernate.properties [overrides definitions in persistence.xml]
    - hibernate.bytecode.provider=javassist

4) OneToOne

```
@Entity                          @Entity
public class Customer {          public class Address {
   @Id                              @Id
   private int id;                  private int id;

   @OneToOne                        @OneToOne
   private Address address;         private Customer customer;
   ...                              ...
```



4) OneToOne



5) Bidirectional OneToOne

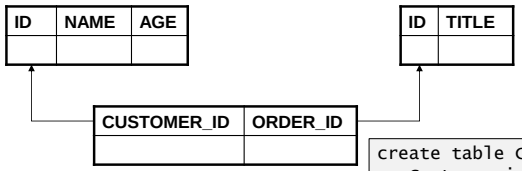
```
Customer c = new Customer("Gosling", 44);
Address a = new Address("Infinite Loop 1", "Cupertino");
c.setAddress(a);

em.persist(a); // necessary ???
em.persist(c);
```

- **em.persist(a)**
  - not necessary if *cascade=CascadeType.PERSIST*
  - Otherwise, if ***a*** is not persisted, an exception is thrown

```
object references an unsaved transient instance - save the
transient instance before flushing:
ch.fhnw.edu.model.Customer.address -> ch.fhnw.edu.model.Address
```


6) Unidirectional OneToMany

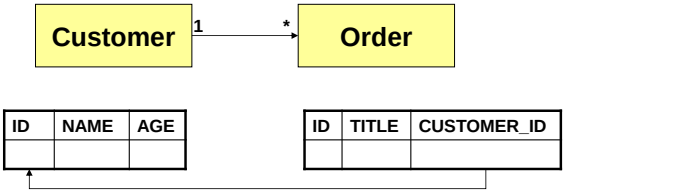
```
@OneToMany
private List<Order> orders = new ArrayList<Order>();
```




```
create table CUSTOMER_ORDERS (
   Customer_id integer not null,
   orders_id integer not null,
   unique (orders_id))
```

6) Unidirectional OneToMany

```
@OneToMany
@JoinColumn(name="CUSTOMER_ID")
private List<Order> orders = new ArrayList<Order>();
```



6) Unidirectional OneToMany

- **Inconsistent Model: what happens?**

```
c1.addOrder(o1);
c1.addOrder(o2);

c2.addOrder(o1);
c2.addOrder(o3);
```

  - Intermediate Table:
    - Exception in thread "main" java.sql.SQLException: Integrity constraint violation FKE0BB9646C73D8EAC table: CUSTOMER_ORDERS
  - Foreign Key:
    - Last Insert wins

7) Flush Mode

```
Customer c = em.find(Customer.class, 1);
c.getAddress().setCity("Basel");

Query q = em.createQuery("select a.city from Address a");
List<?> cities = q.getResultList();
for(Object city : cities)
       System.out.println(city);
```

- **em.setFlushMode(FlushModeType.*COMMIT*);**
  - Windisch
  - *Pending changes are synchronized with the database upon commit*
- **em.setFlushMode(FlushModeType.*AUTO*); [default]**
  - Basel
  - *All pending changes in persistence context are synchronized with database before a query is executed*

8) Persistence Context & Database

```
em.getTransaction().begin();
Customer c = em.find(Customer.class, 1);
c.getAddress().setCity("Zuerich");
// flush not necessary as default flush-mode = AUTO
Query q = em.createQuery("select a.city from Address a");
List<?> cities = q.getResultList();
for(Object city : cities)
       System.out.println(city);            Zürich
System.in.read();
em.getTransaction().rollback();
                                            Wettingen
```

- **Synchronizing PersistenceContext with Database # COMMIT**
- **Whether uncommitted changes are visible depends on TX isolation level**

Fachhochschule Nordwestschweiz

Assignment 5

- **Bidirectional @OneToOne**
- **Bidirectional @OneToMany**

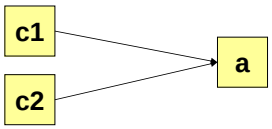
Fachhochschule Nordwestschweiz

1) Bidirectional OneToOne

```
Customer c1 = new Customer("Lee", 44);
Customer c2 = new Customer("Gosling", 55);
Address a = new Address("Infinite Loop 1", "Cupertino");
c1.setAddress(a);
c2.setAddress(a);

em.persist(a);
em.persist(c1);
em.persist(c2);
```

- em.persist(a) is not necessary if *cascade=CascadeType.PERSIST* is defined on address association
- Not a 1:1 association
- Inconsistency in DB as well

ID	AGE	NAME	ADDRESS_ID
7	44	Lee	4
8	55	Gosling	4



Fachhochschule Nordwestschweiz

1) Bidirectional OneToOne

```
public void setAddress(Address address) {
   if(this.address != null){
      this.address.setCustomer(null);
   }
   this.address = address;
   if(this.address != null){
      this.address.setCustomer(this);
   }
}
```

```
public void setCustomer(Customer c) {
   if(this.customer != null){
      this.customer.setAddress(null);
   }
   this.customer = c;
   if(this.customer != null){
      this.customer.setAddress(this);
   }
}
```

- **c1.setAddress(a);**
  => Leads to an infinite loop

Fachhochschule Nordwestschweiz

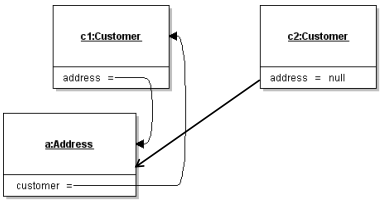
1) Bidirectional OneToOne

```
public void setAddress(Address address) {
   if(this.address != address){
      if(this.address != null){
         this.address.setCustomer(null);
      }
      this.address = address;
      if(this.address != null){
         this.address.setCustomer(this);
      }
   }
}
```

```
public void setCustomer(Customer customer) {
   if(this.customer != customer){
      if(this.customer != null){
         this.customer.setAddress(null);
      }
      this.customer = customer;
      if(this.customer != null){
         this.customer.setAddress(this);
      }
   }
}
```

1) Bidirectional OneToOne

- **c1.setAddress(a);**
- **c2.setAddress(a);**
  - c2.address = a
  - a.setCustomer(c2)
    - c1.setAddress(null)
      - a.setCustomer(null)
        - c1.setAddress(null)
          - …



=> Leads to an infinite loop with the calls
c1.setAddress(null)
a.setCustomer(null)

1) Bidirectional OneToOne

```
public void setAddress(Address address) {
   if(this.address != address){
      Address oldAddress = this.address;
      this.address = address;
      if(oldAddress != null){
         oldAddress.setCustomer(null);
      }
      if(this.address != null){
         this.address.setCustomer(this);
      }
   }
}
```

- **c1.setAddress(a);**
  c2.setAddress(a);
  => All references are set to null

1) Bidirectional OneToOne

```
public void setAddress(Address address) {
   if(this.address != address){
      Address oldAddress = this.address;
      this.address = address;
      if(oldAddress != null){
         oldAddress.setCustomer(null);
      }
      if(address != null){
         address.setCustomer(this);
      }
   }
}
```

- **Seems to work...**
  - until someone finds a counter example

1) Bidirectional OneToOne

```
private transient boolean setting = false;
public void setAddress(Address address) {
   if( !setting ){
      setting = true;
      if(this.address != address ){ // optimization only
         if(this.address != null)
            this.address.setCustomer(null);
         this.address = address;
         if(this.address != null)
            this.address.setCustomer(this);
      }
      setting = false;
   }
}
```

- **Avoids reentrant calls**
  - Problem: is not thead-safe!
    => Variant: use ThreadLocal to store the *setting* flag

Fachhochschule Nordwestschweiz

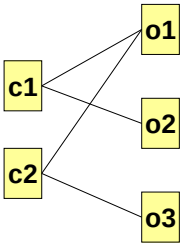
Assignment 5

- **Bidirectional @OneToOne**
- **Bidirectional @OneToMany**

Fachhochschule Nordwestschweiz

2) Bidirectional OneToMany

```
Customer c1 = new Customer("Haller", 52);
Customer c2 = new Customer("Schneider", 66);
Order o1 = new Order(),
Order o2 = new Order();
Order o3 = new Order();

c1.addOrder(o1);
c1.addOrder(o2);

c2.addOrder(o1);
c2.addOrder(o3);

em.persist(c1);
em.persist(c2);
```



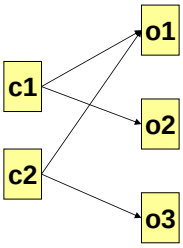
- Assumption: cascade=CascadeType.*PERSIST*

Fachhochschule Nordwestschweiz

2) Bidirectional OneToMany

```
Customer:
   @OneToMany(mappedBy = "customer",
              cascade=CascadeType.PERSIST)
   private List<Order> orders
              = new ArrayList<Order>();

   public void addOrder(Order o){
      orders.add(o);
   }
```

```
Order:
   @ManyToOne
   private Customer customer
```



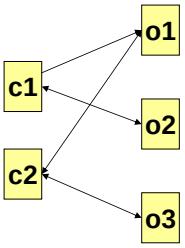
ID	CUSTOMER_ID
4	
5	
6	

Fachhochschule Nordwestschweiz

2) Bidirectional OneToMany

```
Customer:
   @OneToMany(mappedBy = "customer",
              cascade=CascadeType.PERSIST)
   private List<Order> orders
              = new ArrayList<Order>();

   public void addOrder(Order o){
      orders.add(o);
      o.setCustomer(this);
   }
```

```
Order:
   @ManyToOne
   private Customer customer;

   public void setCustomer(Customer c){
      this.customer = c;
   }
```



ID	CUSTOMER_ID
10	20
11	19
12	20

2) Bidirectional OneToMany

- **Customer**

```
public void addOrder(Order order) {
   if(orders.add(order)) order.setCustomer(this);
}
public void removeOrder(Order o){
   if(orders.remove(o)) o.setCustomer(null);
}
```

Breaks recursion if it is a set and not a list, otherwise not!

- **Order**

```
public void setCustomer(Customer customer){
   if(this.customer != null)
         this.customer.removeOrder(this);
   this.customer = customer;
   if(this.customer != null)
         this.customer.addOrder(this);
}
```


2) Bidirectional OneToMany

- **Customer**

```
public void addOrder(Order o){
   o.setCustomer(this);
}
public void removeOrder(Order o){
   o.setCustomer(null);
}
```

- **Order**

```
public void setCustomer(Customer customer){
   if(this.customer != null)
         this.customer.getOrders().remove(this);
   this.customer = customer;
   if(this.customer != null)
         this.customer.getOrders().add(this);
}
```

Provided that getOrders does not return null

Provided that getOrders does not return a copy or an immutable collection.

2) Bidirectional OneToMany

- **Customer**

```
public Collection<Order> getOrders() {
  return Collections.unmodifiableList(this.orders);
}
void addOrder(Order o){ this.orders.add(o); }
void removeOrder(Order o){ this.orders.remove(o); }
```

Prevents that orders are added over the getter

Not public!

- **Order**

```
public void setCustomer(Customer customer){
   if(this.customer != null)
         this.customer.removeOrder(this);
   this.customer = customer;
   if(this.customer != null)
         this.customer.addOrder(this);
}
```


Automatic Consistency

- **Advantage**
  - Consistent objects, and as a consequence consistency in the DB
  - Easy to use
- **Disadvantage**
  - Runtime-overhead, even if consistency is not needed (e.g. as transaction commits)
  - Setters do more than what is expected from a setter
  - JPA: Field access is needed

11. AOP

Listing 11.1: Aspekt Beispiele

```
 @Aspect
@Component
public class MovieProgressAspect {
  private static final Logger LOG = LoggerFactory
      .getLogger(MovieProgressAspect.class);

  @Autowired
  private MovieProgress movieProgress;

  @AfterReturning(pointcut = "execution(* ch.fhnw.edu.rental.services.MovieService.
      getAllMovies())", returning =
"movielist")
  public void checkMovieList(List<Movie> movielist) {
    if (movieProgress.checkForUpdateProgress(movielist)) {
      LOG.debug(movieProgress.toString());
    }
  }
}

@Aspect
@Component
public class MovieStatisticAspect {
    private static final Logger LOG = LoggerFactory.getLogger(MovieStatisticAspect.class);
    @Autowired
    private MovieStatistic statistic;

    @Around("execution(* *..*.MovieService.saveOrUpdateMovie(..)) && args(movie)")
    public void checkSave(ProceedingJoinPoint pjp, Movie movie) throws Throwable  {
      if (movie.getId() == null) {
        pjp.proceed();
        statistic.movieAdded();
        LOG.debug("Actual # of movies are {}", statistic.getNrOfMovieInstance());
      } else {
        pjp.proceed();
      }
    }


    @After("execution(* *..*.MovieService.deleteMovie(..))")
    public void checkDelete()  {
      statistic.movieDeleted();
      LOG.debug("Actual # of movies are {}", statistic.getNrOfMovieInstance());
    }
}

@Aspect
@Component
public class MovieValidatorAspect {
  private static final Logger LOG = LoggerFactory
      .getLogger(MovieValidatorAspect.class);
  @Autowired
  private MovieValidator movieValidator;

  @Around("execution(* *..*.MovieService.saveOrUpdateMovie(..)) && args(movie)")
  public void checkMovieEntity(ProceedingJoinPoint pjp, Movie movie) throws Throwable {
    if (movieValidator.isValid(movie)) {
      LOG.debug("Proceeding for movie '{}'", movie.getTitle());
      pjp.proceed();
    } else {
      LOG.debug("Movie '{}' is not valid", movie.getTitle());
      throw new RuntimeException("Movie Bean not valid");
    }
  }
}
```