tgm
Die Schule der Technik

JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

Systemtechnik Labor

4AHIT 2017/18

Komponentenbasierte Programmierung

Laborprotokoll

Marc Rousavy

24. April 2018

Bewertung:		Version:	1.1
Betreuer:	Michael Borko	Begonnen:	12. April 2018
		Beendet:	20. April 2018

Inhaltsverzeichnis

1 Einführung **3**
1.1 Ziele . . . 3
1.2 Voraussetzungen . . . 3
1.3 Aufgabenstellung . . . 3
1.4 Bewertung . . . 5
1.5 Quellen . . . 6

2 Lösung **7**
2.1 Vorbereitung . . . 7
2.2 Datenbank . . . 9
2.3 JPA Konfiguration . . . 10
2.4 Entities . . . 11
2.4.1 Astah . . . 11
2.4.2 Java Klassen . . . 11
2.5 EntityManager . . . 14
2.5.1 Erstellen . . . 14
2.5.2 Create . . . 14
2.5.3 Update . . . 14
2.5.4 Remove . . . 14

Literaturverzeichnis **16**

1 Einführung

Diese Übung zeigt die Anwendung von komponentenbasierter Programmierung mittels Webframeworks.

1.1 Ziele

Das Ziel dieser Übung ist die automatisierte Persistierung und Verwendung von Objekten eines vorgegebenen Domänenmodells mittels eines Frameworks. Dabei sollen die CRUD-Operationen der verwendeten API zur Anwendung kommen.

Die Persistierung soll mittels der Java Persistence API (JPA) realisiert werden.

1.2 Voraussetzungen

- Grundlagen zu Java und das Anwenden neuer Application Programming Interfaces (APIs)
- Verständnis über relationale Datenbanken und dessen Anbindung mittels höherer Programmiersprachen (JDBC/ODBC)
- Verständnis von UML und Build-Tools

1.3 Aufgabenstellung

Erstellen Sie von folgendem Modell Persistenzklassen und implementieren Sie diese mittels JPA:

Suche Die Suche nach Zügen muss auf jeden Fall die Auswahl des Abfahrts- und Ankunftsortes (nur folgende Bahnhöfe sind möglich: Wien Westbhf, Wien Hütteldorf, St. Pölten, Amstetten, Linz, Wels, Attnang-Puchheim, Salzburg) ermöglichen. Dies führt zur Anzeige der möglichen Abfahrten, die zur Vereinfachung an jedem Tag zur selben Zeit stattfinden. Des weiteren wird auch die Dauer der Fahrt angezeigt.

In dieser Liste kann nun eine gewünschte Abfahrtszeit ausgewählt werden. Die Auswahl der Zeit führt zu einer automatischen Weiterleitung zum Ticketshop.

Um sich die Auslastung der reservierten Sitzplätze anzusehen, muss bei dem Suchlisting noch das Datum ausgewählt werden. Dieses Service steht jedoch nur registrierten Benutzern zur Verfügung.

Ticketshop Man kann Einzeltickets kaufen, Reservierungen für bestimmte Züge durchführen und Zeitkarten erwerben. Dabei sind folgende Angaben notwendig:

Einzeltickets: Strecke (Abfahrt/Ankunft), Anzahl der Tickets, Optionen (Fahrrad, Großgepäck) Reservierung: Strecke (Abfahrt/Ankunft), Art der Reservierung (Sitzplatz, Fahrrad, Rollstuhlstellplatz), Reisetag und Zug (Datum/Uhrzeit) Zeitkarte: Strecke, Zeitraum (Wochen- und Monatskarte)

Um einen Überblick zu erhalten, kann der Warenkorb beliebig befüllt und jederzeit angezeigt werden. Es sind keine änderungen erlaubt, jedoch können einzelne Posten wieder gelöscht werden.

Die Funktion „Zur Kassa gehen“ soll die Bezahlung und den Ausdruck der Tickets sowie die Zusendung per eMail/SMS ermöglichen. Dabei ist für die Bezahlung nur ein Schein-Service zu verwenden um zum Beispiel eine Kreditkarten- bzw. Maestrotransaktion zu simulieren.

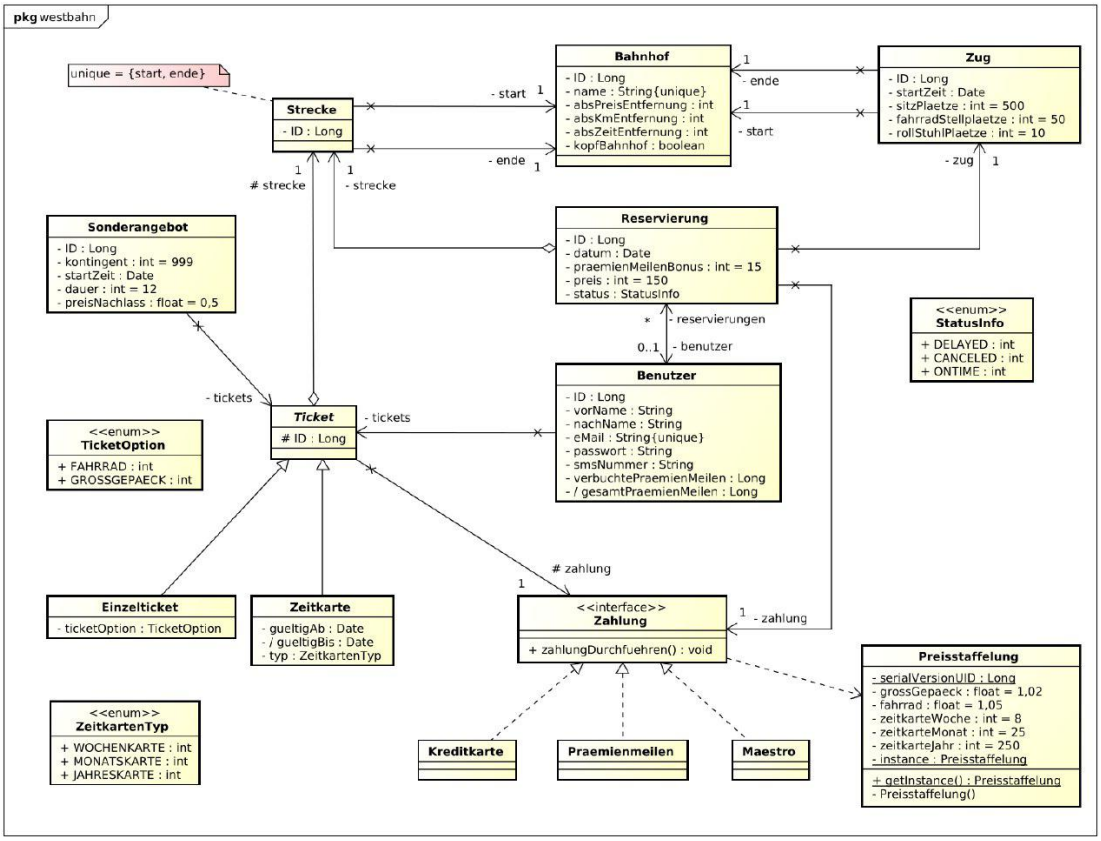

Abbildung 1: Die Westbahn Datenstruktur (UML)

Prämienmeilen Benutzer können sich am System registrieren um getätigte Käufe und Reservierungen einzusehen. Diese führen nämlich zu Prämienmeilen, die weitere Vergünstigungen ermöglichen. Um diese beim nächsten Einkauf nützen zu können, muss sich der Benutzer einloggen und wird beim „Zur Kassa gehen“ gefragt, ob er die Prämienmeilen für diesen Kauf einlösen möchte.

Instant Notification System der Warteliste Der Kunde soll über änderungen bezüglich seiner Reservierung (Verspätung bzw. Stornierung) mittels ausgesuchtem Service (eMail bzw. SMS) benachrichtigt werden. Bei ausgelasteten Zügen soll auch die Möglichkeit einer Anfrage an reservierte Plätze möglich sein. Dabei kann ein Zuggast um einen Platz ansuchen, bei entsprechender änderung einer schon getätigten Reservierung wird der ansuchende Kunde informiert und es wird automatisch seine Reservierung angenommen.

Sonderangebote Für festzulegende Fahrtstrecken soll es ermöglicht werden, dass ein fixes Kontingent von Tickets (z.b.: 999) zu einem verbilligten Preis (z.b.: 50

Task 1 - Mapping Schreiben Sie für alle oben definierten Klassen und Relationen entsprechende Hibernate JPA Implementierungen (javax.persistence.*). Bis auf die Klasse Reservierung sollen dafür

die Annotationen verwendet werden. Die Klasse Reservierung soll mittels XML Mapping definiert werden.

Task 2 - Named Queries Schreiben Sie folgende NamedQueries (kein plain SQL und auch keine Inline-Queries) für das Domänenmodell aus Task1. Die Queries sollen die entsprechenden Parameter akzeptieren und die gewünschten Typen zurückliefern:

1. Finde alle Reservierungen für einen bestimmten Benutzer, der durch die eMail-Adresse definiert wird.
2. Liste alle Benutzer auf, die eine Monatskarte besitzen.
3. Liste alle Tickets für eine bestimmte Strecke aus (durch Anfangs- und Endbahnhof definiert), wo keine Reservierungen durchgeführt wurden.

Task 3 - Validierung Alle Constraints der einzelnen Entitäten sollen verifiziert werden. Hierfür soll die Bean Validation API verwendet werden. Folgende Einschränkungen sollen überprüft werden:

1. Zug und Strecke können nicht denselben Start- und Endbahnhof besitzen.
2. Die eMail des Benutzers soll ein gängiges eMail-Pattern befolgen.
3. Die Startzeit eines Sonderangebotes kann nicht in der Vergangenheit liegen.
4. Der Name eines Bahnhofs darf nicht kürzer als zwei und nicht länger als 150 Zeichen sein. Sonderzeichen sind bis auf den Bindestrich zu unterbinden.

1.4 Bewertung

- Gruppengrösse: 1 Person
- Anforderungen "überwiegend erfüllt"
  - Dokumentation und Beschreibung der angewendeten Schnittstelle
  - Task 1
  - Task 2
- Anforderungen ßur Gänze erfüllt"
  - Task 3
  - Ausreichende Testobjekte zur Validierung der Persistierung
  - Überprüfung der funktionalen Anforderungen mittels Regressionstests

1.5 Quellen

"The Java EE Tutorial - Persistence"; Oracle; online: https://docs.oracle.com/javaee/7/tutorial/partpersist.htm#BNBPY

"HTML5 - A vocabulary and associated APIs for HTML and XHTML"; W3C; 17.12.2012; online: https://www.w3.org/TR/2012/CR-html5-20121217/forms.html#valid-e-mail-address

"Hibernate ORM Documentation"; JBoss; online: http://hibernate.org/orm/documentation/5.2/l

"Hibernate ORM 5.2.13 Final User Guide"; JBoss; 25.01.2018; online: https://docs.jboss.org/hibernate/orm/5.2/userg

2 Lösung

2.1 Vorbereitung

Als Projekt wird eine **Java EE 8** Application mit folgenden Frameworks erstellt:

- **Maven** - Package/Library Management System
- **Java Persistence API** - Annotation Declarations für Entities und das ORM System
- **Hibernate** - Die Implementierung von Java Persistence API (JPA)
- **Java Database Connectivity (JDBC)** - Java Datenbank Interface
- **log4j** - Java Logger API

IntelliJ lädt automatisch benötigte Libraries, andernfalls kann man dies manuell hinzufügen.
Maven wird bei dem Menüpunkt **Project Dependencies** ausgewählt.
Mittels Maven können nun alle Libraries hinzugefügt werden:

- MySQL:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>6.0.6</version>
</dependency>
```

- Hibernate (inklusive JPA):

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.16.Final</version>
</dependency>
```

- log4j:

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

- XML Bind:

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
</dependency>
```

Das **Maven** Project Object Model (POM) sieht dementsprechend folgendermaßen aus:

```
<!--pom.xml-->
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" >
    <modelVersion>4.0.0</modelVersion>

    <groupId>mrousavy</groupId>
    <artifactId>Westbahnhof</artifactId>
    <version>1.0</version>

    <dependencies>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>6.0.6</version>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>5.2.16.Final</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
        </dependency>
        <dependency>
            <groupId>javax.xml.bind</groupId>
            <artifactId>jaxb-api</artifactId>
            <version>2.3.0</version>
        </dependency>
    </dependencies>
</project>
```

2.2 Datenbank

Als Datenbank habe ich mich für eine MySQL 5 Datenbank entschieden, welche in einem Docker [1] Container laufen wird.

Dazu habe ich mir eine Dockerfile [3] geschrieben, welche diesen angepassten MySQL Container zusammenbaut.

Das vollständige Dockerfile schaut dementsprechend folgendermaßen aus:

```
FROM mysql:latest

ENV MYSQL_ROOT_PASSWORD=root

ENV MYSQL_DATA_DIR=/var/lib/mysql \
    MYSQL_RUN_DIR=/run/mysqld \
    MYSQL_LOG_DIR=/var/log/mysql

ADD ["db_dump.sql", "/tmp/dump.sql"]
COPY ./db_dump.sql /docker-entrypoint-initdb.d/

RUN /etc/init.d/mysql start && mysql -u root -p${MYSQL_ROOT_PASSWORD} <
↪  /tmp/dump.sql

EXPOSE 3306
```

Wobei das `db_dump.sql` Create-Script nur die Datenbank erstellt:

```
DROP DATABASE IF EXISTS westbahn;
CREATE DATABASE westbahn;
USE westbahn;
```

Um den Container zu kompilieren verwende ich folgendes script:

```
#!/bin/bash

docker stop mysql
docker rm mysql

docker build -t mysqlimg .
sleep 2
docker run -d --name mysql -v /home/mrousavy/Dockerfiles/data:/var/lib/mysql
↪  mysqlimg
```

Und gestartet kann er mit

```
docker start mysql
```

werden. In meinem Fall ist die IP Addresse des Containers `172.17.0.2`.

2.3 JPA Konfiguration

Die JPA (bzw. Hibernate) muss korrekt konfiguriert werden. Unter anderem muss die SQL Verbindung und der Typ festgelegt werden.

Die Konfigurations-Datei ist eine XML Datei, welche in den App-Resources gespeichert werden muss.

In meinem Fall ist das der relative Pfad: `src/main/resources/META-INF/persistence.xml`.

Der root tag dieser XML Datei ist ein `persistence` tag, in welchem die `persistence-unit` (mit dem `name` Attribut) definiert wird.

Diese `persistence-unit` hat verschiedene `property` tags, welche bestimmte Attribute in der JPA konfigurieren.

Die vollständige `persistence.xml` Datei wird folgendermaßen definiert:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4                  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                   ↪  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
5                  version="2.0" >
6
7      <persistence-unit name="westbahn" >
8          <description>Westbahn Persistence Configuration XML file</description>
9          <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
10
11         <properties>
12             <!-- MySQL -->
13             <property name="hibernate.connection.url"
               ↪  value="jdbc:mysql://172.17.0.2:3306/westbahn" />
14             <property name="hibernate.connection.username" value="root" />
15             <property name="hibernate.connection.password" value="root" />
16             <property name="hibernate.connection.driver"
               ↪  value="com.mysql.jdbc.Driver" />
17             <property name="hibernate.dialect"
               ↪  value="org.hibernate.dialect.MySQL5Dialect" />
18
19             <property name="hibernate.hbm2ddl.auto" value="create"/>
20             <property name="hibernate.show_sql" value="true"/>
21             <property name="hibernate.format_sql" value="true"/>
22         </properties>
23     </persistence-unit>
24 </persistence>
```

2.4 Entities

Entities sind einfache Klassen, bzw. Plain Old Java Object (POJO), welche durch die JPA auf die Datenbank *gemapped* [2] werden.

Für die Westbahn, müssen alle Entities in dem UML (Siehe: Figur **??**) implementiert werden.
Je Entity muss ein POJO erstellt werden, welches eine von der JPA verwaltete Schnittstelle zwischen dem Java Code und der Datenbank repräsentiert.

2.4.1 Astah

Die Entities habe ich aus dem Astah UML Diagramm exportiert: **Tool -> Java -> Export Java**
Astah hat nun alle Klassen aus dem UML Diagramm exportiert, welche aber noch alle JPA **Annotations** erhalten werden.

2.4.2 Java Klassen

Annotations sind Metadata informationen, welche zur Kompilierung (bei Ausnahmefällen: Auch zur Laufzeit) Klassen, Attributen oder Methoden hinzugefügt werden können.
Die wichtigste **Annotation** ist:

```
1 @Entity
2 public class Bahnhof {}
```

welche auf jedem *gemappten* [2] POJO zu finden ist.

Außerdem sollte jede Entity eine eindeutige Identifikation haben, oft durch eine Ganzzahl mit dem namen `ID`.
Beispielsweise könnte so die ID von einem Bahnhof ausschauen:

```
1 @Id
2 @GeneratedValue(strategy = GenerationType.IDENTITY)
3 private long ID;
```

wobei die **Annotation** `@GeneratedValue` der JPA mitteilt, es handelt sich um ein generiertes Attribut welches im Idealfall nicht vom Benutzer modifiziert wird. Beispielsweise wäre dies eine automatisch mitzählende Identifikation.

Alle anderen Attribute bzw. **Properties** können zusätzliche **Annotations** erhalten, sind jedoch optional. Beispielsweise kann ein name auch einzigartig sein:

```
1 @Column(unique = true)
2 private String name;
```

Mit etwas Integrated Development Environment (IDE) Zauberei können auch getter und setter Methoden generiert werden, womit das POJO - und nun auch die Entity - nun so ausschaut:

```java
package BusinessObjects;

import javax.persistence.*;
import java.io.Serializable;

@Entity
public class Bahnhof {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long ID;

    @Column(unique = true)
    private String name;

    private int absPreisEntfernung;

    private int absKmEntfernung;

    private int absZeitEntfernung;

    private boolean kopfBahnhof;

    public long getID() {
        return ID;
    }

    public void setID(long ID) {
        this.ID = ID;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAbsPreisEntfernung() {
        return absPreisEntfernung;
    }

    public void setAbsPreisEntfernung(int absPreisEntfernung) {
        this.absPreisEntfernung = absPreisEntfernung;
    }

    public int getAbsKmEntfernung() {
        return absKmEntfernung;
    }
```

```
50
51      public void setAbsKmEntfernung(int absKmEntfernung) {
52          this.absKmEntfernung = absKmEntfernung;
53      }
54
55      public int getAbsZeitEntfernung() {
56          return absZeitEntfernung;
57      }
58
59      public void setAbsZeitEntfernung(int absZeitEntfernung) {
60          this.absZeitEntfernung = absZeitEntfernung;
61      }
62
63      public boolean isKopfBahnhof() {
64          return kopfBahnhof;
65      }
66
67      public void setKopfBahnhof(boolean kopfBahnhof) {
68          this.kopfBahnhof = kopfBahnhof;
69      }
70  }
```

Dementsprechend müssen nun alle POJOs zu **Entities** gemacht werden.

Ein Sonderfall war die Entity Strecke (`Strecke.java`), da diese *unique constraints* benötigte.
Dazu gefunden habe ich die Dokumentation in den Java Docs: Unique Constraints, welche zu folgendem Code für die Entity **Strecke** führte:

```
1   @Table(
2           uniqueConstraints = @UniqueConstraint(columnNames = {"start_id", "ende_id"})
3   )
```

Die Spalten heißen `start_id`/`end_id`, da es sich nicht um Scalar-Properties, welche standardisierte Datentypen sind, sondern um Navigation-Properties, welche im Hintergrund von der JPA zu Foreign Keys verarbeitet werden, hält.

Es wird also kein tatsächliches Bahnhof Objekt auf der Datenbank gespeichert (da dies nicht möglich ist), sondern nur der Foreign Key zu einem Eintrag in dem Bahnhof Table. (Siehe: Mapping [2])

2.5 EntityManager

2.5.1 Erstellen

Sobald alle Entities erstellt wurden, kann man die `EntityManagerFactory` [5] verwenden um eine Verbindung zur Datenbank aufzubauen und den `EntityManager` [4] erstellen, um mit den JPA *gemappten* Entities arbeiten.

```
1 EntityManagerFactory emfactory = Persistence.createEntityManagerFactory("westbahn");
2 EntityManager entitymanager = emfactory.createEntityManager();
3 // ...
4 entitymanager.close();
5 emfactory.close();
```

2.5.2 Create

Entities können erstellt werden, indem das POJO mittels `new` erstellt wird, und dem `EntityManager` *angehängt* wird:

```
1 entityManager.getTransaction().begin();
2 Bahnhof bahnhof = new Bahnhof();
3 bahnhof.setName("Wien Spittelau");
4 entitymanager.persist(bahnhof);
5 entityManager.getTransaction().commit();
```

Mittels `persist` wird diese Entity dem `PersistenceContext` hinzugefügt, es besteht jedoch auch die Möglichkeit die Entity mittels `merge` hinzuzufügen. Allerdings wird bei `merge` eine neue Entity mit kopierten Werten erstellt, was ein unnötiger Overhead ist.

2.5.3 Update

Um bestimmte Werte einer Entity zu ändern, muss dies an einer Entity welche bereits im `PersistenceContext` registriert ist getan werden.

Falls diese Entity nur lokal existiert, muss diese zuerst mittels ID über den `EntityManager` gefunden werden.

```
1 entityManager.getTransaction().begin();
2 Bahnhof bahnhof = entityManager.find(Bahnhof.class, 0);
3 bahnhof.setName("Wien Heiligenstadt");
4 entityManager.flush();
5 entityManager.getTransaction().commit();
```

2.5.4 Remove

Um eine Entity zu entfernen muss sie zuerst gefunden werden (oder bereits im `PersistenceContext` vorhanden sein) und dann mittels `remove` entfernt werden:

```
1  entityManager.getTransaction().begin();
2  Bahnhof bahnhof = entityManager.find(Bahnhof.class, 0);
3  entityManager.remove(bahnhof);
4  entityManager.flush();
5  entityManager.getTransaction().commit();
```

Glossar

Entity „Entities sind einfache Klassen, bzw POJO, welche durch die JPA auf die Datenbank *gemapped* werden." [**wiki-entity**]. 11, 13

Unique Constraints „Unique Constraints sind Constraints welche die Einzigartigkeit dieses Attributes versichern." [**jdoc:uniqueConstraints**]. 13

Akronyme

IDE Integrated Development Environment. 11

JDBC Java Database Connectivity. 7

JPA Java Persistence API. 7, 10, 11, 13, 14

POJO Plain Old Java Object. 11, 13, 14

POM Project Object Model. 8

Literaturverzeichnis

[1] Wikipedia Authoren. *Docker (software)*. URL: https://en.wikipedia.org/wiki/Docker_(software) (besucht am 24. 04. 2018).

[2] Wikipedia Authoren. *Java Persistence/Mapping*. URL: https://en.wikibooks.org/wiki/Java_Persistence/Mapping (besucht am 24. 04. 2018).

[3] Docker. *Dockerfile reference*. URL: https://docs.docker.com/engine/reference/builder/ (besucht am 24. 04. 2018).

[4] Oracle. *Interface EntityManager*. URL: https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html (besucht am 24. 04. 2018).

[5] Oracle. *Interface EntityManagerFactory*. URL: https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManagerFactory.html (besucht am 24. 04. 2018).

Abbildungsverzeichnis

1 Die Westbahn Datenstruktur (UML) 4