tgm
Die Schule der Technik

JUST DO IT
HÖHERE ABTEILUNG FÜR
INFORMATIONSTECHNOLOGIE

Systemtechnik Labor

4BHIT 2017/18, Gruppe 2

Komponentenbasierte Programmierung

Laborprotokoll

Peter Fuchs

17. Mai 2018

Bewertung:		Version:	1.0
Betreuer:	Michael Borko	Begonnen:	15.3.18
		Beendet:	3.5.18

Inhaltsverzeichnis

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:

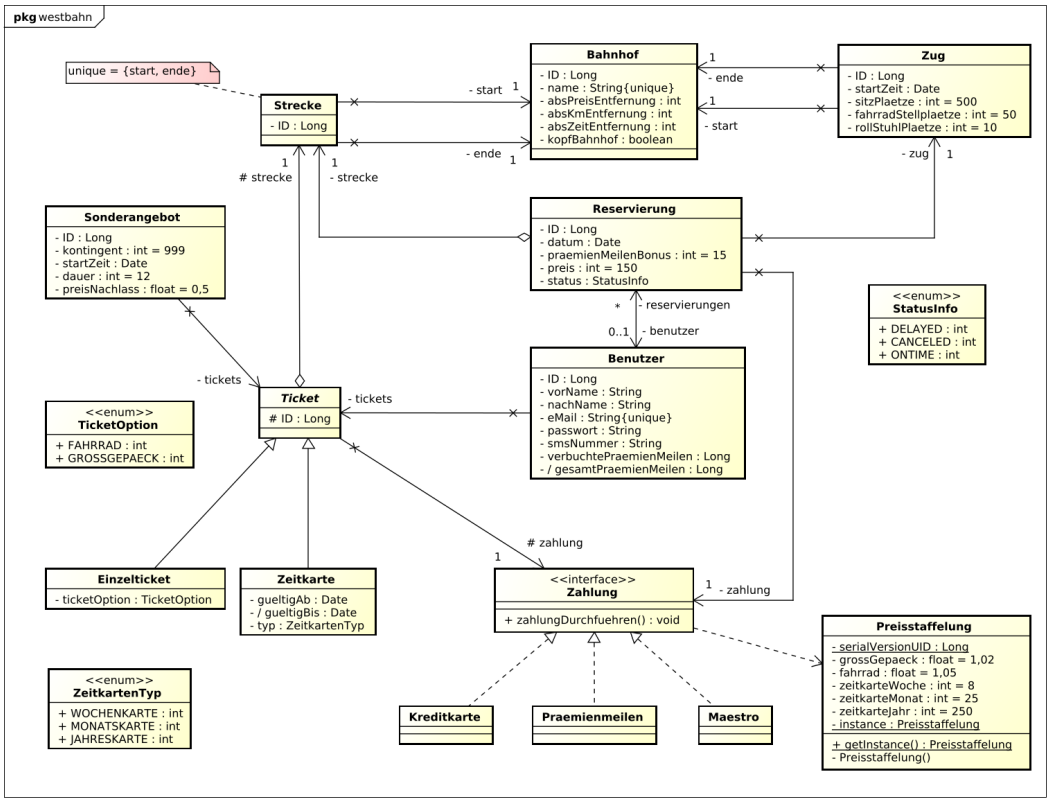

Abbildung 1: UML-Klassendiagramm

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.

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.

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% Reduktion) angeboten wird. Diese Angebote haben neben dem Kontingent auch eine zeitliche Beschränkung. Der Start wird mit Datum und Uhrzeit

festgelegt. Die Dauer wird in Stunden angegeben. Diese Angebote werden automatisch durch Ablauf der Dauer beendet.

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:

a) Finde alle Reservierungen für einen bestimmten Benutzer, der durch die eMail-Adresse definiert wird.

b) Liste alle Benutzer auf, die eine Monatskarte besitzen.

c) 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:

a) Zug und Strecke können nicht denselben Start- und Endbahnhof besitzen.

b) Die eMail des Benutzers soll ein gängiges eMail-Pattern befolgen.

c) Die Startzeit eines Sonderangebotes kann nicht in der Vergangenheit liegen.

d) 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 "**zur Gänze erfüllt**"
  - Task 3
  - Ausreichende Testobjekte zur Validierung der Persistierung
  - Überprüfung der funktionalen Anforderungen mittels Regressionstests

2 Durchführung

2.1 Installation Hibernate

Als Vorlage wurde das westbahn-Package von Professor Borko verwendet. Dieses beinhaltete alle benötigten externen JAR-Files sowie das benötigte UML-Diagramm in Form eines Astah-Files.
Um Hibernate aufzusetzen musste zuerst ein Projekt erstellt werden. In diesem werden im Build-Path alle benötigten externen JARs hinzugefügt.

2.2 Konfiguration

Danach muss eine Konfigurationsdatei für Hibernate erstellt werden (diese basiert auf einer Konfigurationsdatei aus diesem Tutorial). Angepasst werden musste hier vor allem die Verbindung zur Datenbank. Zusätzlich wurden in diesem File alle Klassen, sowie das zu erstellende XML-Mapping-File (Reservierung) angegeben. Im Endeffekt sollte die Konfigurationsdatei ungefähr so ausssehen:

```
<?xml version="1.0" encoding="utf-8"?>
<hibernate-configuration>
	<session-factory>
		<property name="hbm2ddl.auto">create</property>
		<property name="dialect">org.hibernate.dialect.MySQLDialect</property>
		<property name="hibernate.connection.driver_class">
			com.mysql.jdbc.Driver
		</property>
		<property name="connection.url">
			jdbc:mysql://localhost:3306/westbahn?useSSL=false
		</property>
		<mapping class="westbahn.Strecke" />
		<mapping class="westbahn.Bahnhof" />
		[...]
		<mapping resource="reservierung.hbm.xml"/>
	</session-factory>
</hibernate-configuration>
```

Auflistung 1: hibernate.cfg.xml

Wichtig: Beim Mapping wird für die Klassen standardmäßig das Attribut class verwendet, im Falle von XML-Mapping ist es das Attribut "resource".

2.3 Task 1 - Annotationen und Mapping

2.3.1 Klassen

Um die Klassen für Hibernate zu "taggen", muss vor die Klasse selbst der Tag `@Entity` gesetzt werden. Damit die Klassen jedoch auch in die Datenbank geschrieben werden können, müssen vor die verschiedenen Attribute ebenfalls Tags gesetzt werden. Die benötigten Attribute müssen daher Annotationen aus folgender Tabelle inkludieren:

Annotation	Ursache
@Id	Wenn das Attribut die Id für das Objekt darstellt und daher eindeutig identifizierbar ist.
@GeneratedValue	Wenn das Attribut automatisch generiert werden soll. Entspricht dem SQL-Befehl "auto increment".
@OneToOne	Sollten die Objekte eine Referenz auf andere Objekte besitzen, die im Entity Relationship Diagram (ERD) einer 1:1-Beziehung entsprechen, so muss diese Annotation verwendet werden.
@ManyToOne **@OneToMany**	Sollten die Objekte eine Referenz auf andere Objekte aufweisen, die im ERD einer 1:N-Beziehung entsprechen, so muss diese Annotation verwendet werden.
@Column	Wenn einem Attribut besondere Eigenschaften zugewiesen werden sollen (z.B. unique), dann wird diese Annotation verwendet. Der Syntax hierbei lautet @Column(variable = value), z.B. @Column(unique = true).
@NotNull	Darf ein Attribut nie den Wert null besitzen, so muss dies mittels dieser Annotation gekennzeichnet werden.

Tabelle 1: Annotationen der Objekte

(Definition Entity Relationship Diagram)

2.3.2 Interfaces

Interfaces und Enums sollten nicht in die Datenbank geschrieben werden, da sie nicht eindeutig identifizierbar sind. Daher werden sie nur über Annotationen zu den in die Datenbank geschriebenen Klassen hinzugefügt.
Die Tags zur Erfüllung dieser Aufgaben waren `@Embeddable` beim Interface und allen implementierenden Child-Klassen, die nicht in die Datenbank geschrieben werden können (aufgrund fehlender Primary Keys), sowie `@Embedded` bei den zugehörigen Attributen.

2.3.3 XML-Mapping

Für das XML-Mapping musste ein extra .hbm.xml-File erstellt werden. Dieses besitzt als root-Element **<hibernate-mapping>**, in welchem sich ein **<class>**-Element befindet. Dieses referenziert durch den Parameter name auf die gemappte Klasse (in diesem Fall westbahn.Reservierung) und hat als Child-Elemente alle Attribute. Hierbei wird in **<id>**, **<property>** (Standard-Datentypen), **<component>** (Interfaces/Enums) und wie bei den Annotationen **<many-to-one>**, **<one-to-one>** oder **<one-to-many>** (für Referenzen auf andere Tabellen) unterschieden. Bei allen diesen Elementen findet sich ein Parameter "name", der dem Attribut-Namen entspricht. Zusätzlich ist mittels type/class der Datentyp angegeben. "access=field" sorgt dafür, dass das Attribut selbst und nicht die Getters und Setters referenziert werden. Das vollständige XML-Dokument sollte dann so ähnlich aussehen:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
	<class name="westbahn.Reservierung">
		<id name="ID" access="field">
			<generator class="increment"></generator>
		</id>
		<property name="datum" type="java.util.Date" access="field" />
		<property name="praemienMeilenBonus" type="int" access="field" />
		<property name="preis" type="int" access="field" />
		<component name="status" class="westbahn.StatusInfo" access="field" />
		<component name="zahlung" class="westbahn.Zahlung" access="field" />
		<one-to-one name="zug" class="westbahn.Zug" access="field" />
		<many-to-one name="strecke" class="westbahn.Strecke" access="field" />
		<many-to-one name="benutzer" class="westbahn.Benutzer" access="field" />
	</class>
</hibernate-mapping>
```

Auflistung 2: reservierung.hbm.xml

Anmerkung: Da dieses File auf einem Guide basiert, ist ein .dtd-File also eine Vorlage für den Aufbau des XML-Files integriert. Dies ist nicht unbedingt notwendig, aber durchaus sinnvoll um die Korrektheit des Syntaxes zu überprüfen.

2.4 Task 2 - Named Queries

Für die Durchführung dieser Aufgabe wurde ein Guide von JBoss als Vorlage verwendet.
Die Named Queries basierten fast alle auf outer Joins und waren wie folgt aufgebaut:

a) Es werden zuerst alle Reservierungen ausgewählt und zu diesen werden alle Benutzer gejoint, bei der die ID der Reservierung jener des Benutzers entspricht. Am Ende werden aus dieser Gesamtmenge die Benutzer ausgewählt, die die angegebene E-Mail-Adresse besitzen:

```
SELECT * FROM reservierung LEFT OUTER JOIN benutzer ON reservierung.benutzer =
↪ benutzer.ID WHERE benutzer.eMail = :eMail
```

b) Es werden zuerst alle Reservierungen ausgewählt und mithilfe eines natural Joins wird die Tabelle "ticket"mit und danach alle Daten mit dem DTYPE (also der Art des Tickets) "Zeitkarte"ausgewählt. Von dieser werden nun nur noch die Typen verwendet, die einem angegebenen Wert entsprechen:

```
SELECT * FROM benutzer NATURAL JOIN ticket WHERE ticket.DTYPE = 'Zeitkarte' AND
↪ ticket.typ = :typ
```

c) Zuerst werden alle Reservierungen ausgewählt und zu diesen werden alle Benutzer gejoint, die eine Reservierung besitzen. Nun werden alle Tickets hinzugefügt und diese Menge wird mithilfe der Variablen von End- und Startbahnhof auf eine Strecke beschränkt. Zusätzlich werden am Ende nur jene Daten ausgewählt, die keine Reservierung haben:

```
SELECT ticket.DTYPE, ticket.ID, ticket.ticketOption, ticket.gueltigAb, ticket.typ,
↪ ticket.strecke_ID, reservierung.ID FROM reservierung LEFT OUTER JOIN benutzer
↪ ON benutzer.ID = benutzer RIGHT OUTER JOIN ticket ON ticket.ID =
↪ benutzer.tickets_ID LEFT OUTER JOIN strecke ON strecke.ID = ticket.strecke_ID
↪ WHERE strecke.ende_ID = :ende AND strecke.start_ID = :start HAVING
↪ reservierung.ID IS NULL;
```

2.5 Task 3 - Validierung

Die Validierung sollte auf der JBean-Validation basieren. Hierfür musste zuerst das hibernate-validator.jar-File eingebunden werden. Nun konnten die einzelnen Tasks abgearbeitet werden:

a) Zug und Strecke dürfen nicht denselben Start- und Endbahnhof besitzen:
Für die Umsetzung dieser Validierung wurde ein neues Attribut bei Zug und Strecke hinzugefügt. Dieses ist ein boolean mit dem Namen "endeIsStart" und es wird bei Aufruf des Konstruktors auf die Gleichheit der Namen von Start- und Endbahnhof gesetzt. Dieses Attribut wird zusätzlich mit der Annotation `@AssertFalse` versehen, um eine Differenz von Start- und Endbahnhof zu erfordern.

b) Die eMail des Benutzers soll ein gängiges eMail-Pattern befolgen:
Hierfür wurde beim Attribut "eMail" die Annotation `@Pattern` mit dem Parameter `(regexp="/^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\\.[a-zA-Z0-9-]+)*$/")` hinzugefügt. Dieses Pattern wurde von w3 entwickelt und ist hier zu finden.

c) Die Startzeit eines Sonderangebotes kann nicht in der Vergangenheit liegen:
Hierfür wurde die Annotation `@Future` verwendet. Dies löste die Aufgabenstellung allerdings nicht vollständig. Das Problem bei dieser Annotation ist, dass aktuelle Daten - offensichtlich - nicht als zukünftige gezählt werden und daher ein Sonderangebot nicht zum aktuellen Zeitpunkt gestartet werden kann. Es gibt mehrere mögliche Lösungen zu diesem Problem. Umgesetzt wurde schlussendlich jene mit einem boolean, welcher auf false gesetzt wurde, wenn das Datum vor dem jetzigen Datum lag. Eine Alternative wäre gewesen, die JBean-Version und die Hibernate-Version auf den neuesten Stand zu bringen, allerdings gab es in diesem Umfeld einige Komplikationen mit dem Code und der Umsetzung, weswegen diese Lösung wieder verworfen wurde.

d) Der Name eines Bahnhofs darf nicht kürzer als zwei und nicht länger als 150 Zeichen sein. Sonderzeichen sind bis auf den Bindestrich zu unterbinden: Um diese Vorgabe einhalten zu können, mussten zum Attribut "name" der Klasse "Bahnhof" zwei Annotationen hinzugefügt werden. Die erste war für die Länge verantwortlich und war eine simple `@Size`-Annotation mit den Parametern `(min=2, max=150)` verwendet. Die Sonderzeichen wurden wieder durch eine `@Pattern`-Annotation überprüft, die hier verwendete Regex war `"^[a-z A-Z0-9äöüÄÖÜ-]*\$"`.

2.6 Testing

Für das Testing wurde JUnit 5.0 verwendet. Auch hier musste zuerst die API eingebunden werden (Java Build Bath -> Libraries -> Add Library -> JUnit -> JUnit 5). Nachdem das Testing-Tool erfolgreich installiert wurde, konnten die Prerequesits für die Tests geschrieben werden:
Hierfür wurde die Datenbank mit einigen Testwerten gefüllt, die danach persistiert und committed wurden.

2.6.1 Persistency

Diese Persistierung wurde auch direkt durch die Methoden `checkPersistency[_<2-4>]` getestet. Hierfür wurden ein Testbahnhof und ein Testnutzer erstellt, welche beide zur Datenbank hinzugefügt wurden und danach wieder aus dieser Ausgelesen wurden. Danach wurden die Objekte jeweils auf Gleichheit überprüft. Zusätzlich wurden noch Methoden geschrieben, welche die Anzahl an Objekten in der Datenbank überprüfen sollten. Hierfür wurden jeweils Anfragen bezüglich der Anzahl von Bahnhöfen, Strecken, Zügen, Tickets, Benutzern und Reservierungen ausgeführt und auf Gleichheit mit der Anzahl an eingefügten Testwerten überprüft.
Als nächstes wurde noch ein Bahnhof zur Datenbank hinzugefügt, dann verändert und persistiert. Hier wurde nach jedem Eintrag in die Datenbank überprüft, ob dieses Objekt auch jenem in der Datenbank entspricht. Zusätzlich wurde überprüft, ob nur ein Element in die Datenbank eingeschrieben wird, oder ob es mehrere Objekte sind.
Zuletzt wurde noch überprüft, ob ein neuer Bahnhof mit exakt gleichen Daten zusätzlich gespeichert, oder persistiert wird. Theoretisch müsste die Datenbank diesen Typen als gleichwertig erkennen, da es allerdings unterschiedliche Java-Objekte sind, war die Erwartung dementsprechend auch, dass mehrere Einträge mit gleichen Daten in die Datenbank gespeichert werden. Diese Erwartungen wurden durch das Testen der Größe aller Objekte, die den verwendeten Bahnhof-Namen besitzen, (erwarteter Wert: 2, tatsächlicher Wert: 2) erfüllt.

2.6.2 Named Queries

Als nächstes wurden die Named Queries aus Task 2 überprüft. Dazu wurde jeweils ein Query-Objekt durch die Methode `getNamedQuery(String query_name)` erstellt, welches die jeweilige Query durchführen sollte. Hier wurden die einzelnen Queries mit verschiedenen gültigen/vorhandenen und nicht vorhandenen Werten durchgeführt und die Richtigkeit mit manueller Überprüfung über die Datenbank festgestellt. Zusätzlich wurden bei manchen Tests Veränderungen in der Datenbank vorgenommen, um die Richtigkeit nochmals zu überprüfen.

2.6.3 Validation

Als letztes wurden Tests zu Task 3 geschrieben. Diese beinhalteten vor allem einige korrekte und nicht korrekte Werte zu den Bahnhöfen, der E-Mail, des Datums und des Namens der Bahnhöfe. Wichtig war bei diesen Tests, dass wirklich alle Tests durchgeführt wurden, das heißt neben zu langen Namen und ungültigen E-Mail-Adressen auch Mischungen aus ungültigen Zeichen und zu langen Namen und Ähnlichem.

3 Fehler und Probleme

3.1 Astah formartiert Enums nicht sinnvoll

Eines der ersten Probleme war die fehlerhafte Übertragung des Astah-Files zu Java. Die Formatierung der Enums war syntaktisch nicht korrekt und musste manuell angepasst werden.

3.2 MySQL-JAR nicht implementiert

Dieses Problem sorgte anfangs für eine nicht funktionale Verbindung zur Datenbank. Nachdem das jar-File inkludiert wurde, konnte allerdings problemlos weitergearbeitet werden.

3.3 Wie sollen Interfaces/Enums implementiert werden?

Hier war vor allem das Problem, dass keine IDs vorhanden waren und nicht ganz klar war, wie diese in die Datenbank geschrieben werden sollten. Nach einiger Nachforschung wurden die Annotationen `@Embeddable/@Embedded` gefunden, mit denen dieses Problem gelöst werden konnte.

3.4 JUnit zum Build Path hinzufügen

Da JUnit nicht als JAR-File geliefert wurde, war am Anfang unklar, wie die Library in Eclipse eingebunden werden sollte. Hierbei war stackoverflow hilfreich, welche die Lösung des Problems bereitgestellt hatten.

3.5 Datenbank immer neu erstellt - nicht persistent

Ein weiteres Problem war die Tatsache, dass Objekte in der Datenbank immer neu erstellt wurden, das heißt, dass keine Persistenz getestet werden konnte. Dieses Problem wurde behoben, indem im "hibernate.cfg.xml"-File die Eigenschaft hbm2ddl.auto von update zu create geändert wurde.

3.6 Reihenfolge der Tests

Das Problem mit der Reihenfolge der Tests bestand vor allem deswegen, weil am Anfang in den Tests eine Veränderung der Daten ausgeführt wurde, auf welche sich andere Tests bezogen. Dieses Problem wurde umgangen, indem diese Veränderung vor die Tests gezogen bzw. ganz aus dem Programm entfernt wurde.

3.7 SQL-Abfrage fehlerhaft

Es kam immer wieder vor, dass SQL-Abfragen keine Objekte zurückgeben. Dies passierte vor allem bei Abfragen, die eine Menge an Objekten zurückgaben (z.B. `SELECT * FROM Benutzer`). Um dieses Problem zu umgehen, wurde an die SQL-Abfrage die Methode `addEntity(Class arg0)` angehängt, welche den Typ der Rückgabe ohne eine Named Query angibt.

4 Quellen

- Testing (JUnit):
  https://oss.sonatype.org/content/repositories/snapshots/org/junit/jupiter/junit-jupiter-api/5.0.0-SNAPSHOT/
- Add JUnit5 to Build Path:
  https://stackoverflow.com/questions/38402155/eclipse-junit-5-support
- SQL returns no objects:
  http://docs.jboss.org/hibernate/core/3.3/reference/en/html_single/#d0e13696
- Validation:
  http://hibernate.org/validator/documentation/getting-started/
- Named-Queries:
  http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#hql
- Enums:
  http://www.codejava.net/frameworks/hibernate/hibernate-enum-type-mapping-example
- Interfaces:
  https://stackoverflow.com/questions/925818/hibernate-persistence-without-id
- XML-Mapping:
  https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/mapping.html#mapping-declaration-proper
- Hibernate ORM Documentation:
  http://hibernate.org/orm/documentation/5.2/l
- E-Mail-Validation:
  https://www.w3.org/TR/2012/CR-html5-20121217/forms.html#valid-e-mail-address

Abbildungsverzeichnis

Tabellenverzeichnis

Auflistungsverzeichnis