SWE 4x	**Übung zu Softwareentwicklung mit modernen Plattformen 4**	**SS 2015, Übung 7**

Abgabetermin: SA in der KW 26

Gr. 1, E. Pitzer
Gr. 2, F. Gruber-Leitner

Name _____ **Aufwand in h** _____

Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

Ausbaustufe 2: CaaS-DB (24 Punkte)

Nachdem der Restaurantbetreiber von der Präsentation Ihrer ersten Ausbaustufe restlos begeistert war, steht einem Ausbau bis hin zur Online-Plattform nichts mehr im Wege.

Im nächsten Schritt sollen Sie die Anforderung umsetzen, dass mehrere Mitarbeiter des Restaurantbetreibers die Benutzeroberfläche gleichzeitig benutzen können (im Restaurant und auch von zu Hause aus). Dafür ist es notwendig, die Benutzeroberfläche und die Datenverwaltung zu entkoppeln und die Kommunikation der beiden Komponenten über das Netzwerk durchzuführen.

In dieser Ausbaustufe sollen Sie weiters dafür sorgen, dass die Daten dauerhaft in einer Datenbank gespeichert werden.

Im Detail sollte Ihre Anwendung folgende Anforderungen erfüllen:

- Überlegen Sie sich eine geeignete Repräsentation für die Daten in Ihrem Programm. Im konkreten Anwendungsfall werden Sie dafür Klassen zur Speicherung der Benutzer-, Menü- und Bestelldaten benötigen. Diese Klassen werden häufig als Domänenmodell bezeichnet.
- Implementieren Sie einen RMI-Server, der die Daten des Menü-Bestellsystems zentral verwaltet. Stellen Sie sicher, dass die Serverkomponente parallel beliebig viele Clients mit Daten versorgen kann.
- Bauen Sie die in Übung 6 entwickelte JavaFX-basierte Benutzeroberfläche zu einem RMI-Client aus, der mit der Serverkomponente kommuniziert. Entwerfen Sie ein Datenmodell, das alle in CaaS anfallenden Daten abbildet. Berücksichtigen Sie nicht nur die Daten der Menüverwaltung sondern auch die Daten für die Online-Bestellung.
- Stellen Sie Klassen zum Zugriff auf die Datenbank auf Basis von JDBC zur Verfügung. Bereiten Sie auch bereits Methoden für das Hinzufügen und Laden von Bestellungen vor. Achten Sie darauf, dass die Datenbankzugriffschicht möglichst weitgehend von den anderen Komponenten der Anwendung getrennt ist. Stellen Sie dazu die gesamte Funktionalität der Datenbankzugriffsschicht über Interfaces zur Verfügung.
- Entwickeln Sie eine Testsuite, mit der Sie die Datenzugriffsschicht unabhängig von den anderen Systemkomponenten testen können.

1 Campina as a Service

1.1 Lösungsidee

Folgend ist die Dokumentation der Aufgabenstellung Erweiterung CAAS angeführt.
Da hierbei mehrere Komponenten wie:

1. RMI Server
2. Domänen Modell
3. DAOs für Domänen Modell
4. UI mit JavaFx

verwendet werden, soll die Gesamtapplikation in mehrere Projekte aufgeteilt werden.
Diese Projekte sollen wie folgt definiert werden:

1. *swe-campina-data-model-api*:
   Die API für das Domänen Modell und die Schnittstellen für den Datenzugriff. Es enthält keinen Datenbank spezifischen Source.
2. *swe-campina-data-model-impl*:
   Die Implementierung der API spezifiziert in *swe-campina-data-model-api*. Stellt eine MySql spezifische Implementierung dar.
3. *swe-campina-dao-api*:
   Die API der DAOs für das Domänen Modell. Sind bereits hier Remote Schnittstellen.
4. *swe-campina-dao-impl*:
   Die Implementierung der API spezifiziert in *swe-campina-dao-api*. Stellt eine MySql spezifische Implementierung dar.
5. *swe-campina-rmi-api*:
   Die API des RMI Servers, der die DAOs hostet.
6. *swe-campina-rmi-impl*:
   Die Implementierung der API spezifiziert in *swe-campina-rmi-api*. Es hat Abhängigkeiten auf *swe-campina-dao-api* und *swe-campina-dao-impl*, da es diese Beans hostet.
7. *swe-campina-fx*:
   Die UI für CAAS in JavaFX. Hier sollen nur die Nötigen Änderungen vorgenommen werden um die Persistenz der Daten zu gewährleisten.

Folgend ein paar Informationen bezüglich dem Setup.

1.1.1 RMI Server

Im Projekt *swe-campina-rmi-impl* gibt es eine Klasse names *MainServer.java*. Diese startet den RMI Server und registriert die DAOFactory in der Registry. Diese Factory wird von den Clients genutzt um die DAOs zu produzieren, wobei pro Instanz maximal 10 Clients gebunden werden. Es nutzen jedoch alle dieselbe `ConnectionManager` Instanz.

1.1.2 DAOs

Im Projekt *swe-campina-data-model-impl/src/main/resources* befindet sich das DDL *create-campina-schema.sql*, die ausgeführt werden muss bevor das erste Mal eine der Applikationen gestartet wird. Des Weiteren ist dort eine Konfigurationsdatei namens *db-config-properties* enthalten, welche die Konfigurationsparameter für den Verbindungsaufbau enthält, die im `ConnectionManager` genutzt werden. Diese Parameter werden über eine Enumeration `DbConfigParam` abgebildet, die auch die Properties Datei lädt. Diese muss auf Korrektheit geprüft werden bevor die Applikationen gestartet werden, um eine Datenbankverbindung gewährleisten zu können.

1.1.3 UI

Im Projekt *swe-campina-fx* gibt es eine Klasse namens *MainUI.java* welche die UI startet. Es werden Testdaten angelegt, sofern der Benutzer mit der Email *thomas.herzog@students.fh-hagenberg.at* order mit dem Benutzernamen *cchet* noch nicht auf der Datenbank existiert. Es werden hierbei auch die Orders für diesen Benutzer angelegt.

1.1.4 JUnit Tests

Für die DAOs wurde eine abstrakte Basisklasse eingeführt, welche eine Methode zur verfügung stellt welche die Datenbank neu erstellen kann. Sie nutzt dieselben Verbindungsdaten wie der Rest der Applikationen, daher ist anzumerken, das die Datenbank neu angelegt wird und daher alle Daten verloren gehen, was aber bei diesem Prototypen keine Rolle spielt, da Testdaten angelegt werden.

1.2 Source swe-campina-data-model-api

Folgend ist der DSource des Projekts *swe-campina-data-model-api* angeführt, welches die Spezifikation des Datenmodells darstellt, wobei hierbei keine Datenbank spezifischen Anteile enthalten sind.
Für das Mapping der Domänenmodelle gegen die Datenbanl wurden die JEE Annotations `@Table()`, `@Column` verwendet, da diese sich hierfür gut geeignet haben. Es wird hierbei lediglich auf die Neuerungen eingegangen und der bestehende Source nicht nochmals dokumentiert.

1.2.1 AbstractEntity.java

Diese Klasse stellt die Wurzelklasse alle Entitäten dar. Sie enthält bereits eine `int hash()`, `boolean equals(Object other)` Implementierung.

Listing 1: AbstractEntity.java

```
1  package at.fh.ooe.swe4.campina.persistence.api;
2
3  import java.io.Serializable;
4
5  import javax.persistence.Column;
6  import javax.persistence.Id;
7
8  /**
9   * This class represents the root of each entity with an integer id. Should be
10  * generic in the future.
11  *
12  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
13  * @date Jun 21, 2015
14  */
15 public abstract class AbstractEntity implements Serializable {
16
17   private static final long  serialVersionUID  = 1095329951571671581L;
18
19   private Integer        id;
20   private Long        version;
21
22   public AbstractEntity() {
23   }
24
25   public AbstractEntity(Integer id) {
26     super();
27     this.id = id;
28   }
29
30   @Id
31   @Column(name = "ID")
32   public Integer getId() {
33     return id;
34   }
35
36   public void setId(Integer id) {
37     this.id = id;
38   }
39
40   @Column(name = "version")
41   public Long getVersion() {
42     return version;
43   }
44
45   public void setVersion(Long version) {
46     this.version = version;
47   }
```

```
48
49    @Override
50    public int hashCode() {
51      final int prime = 31;
52      int result = 1;
53      result = prime * result + ((id == null) ? 0 : id.hashCode());
54      return result;
55    }
56
57    @Override
58    public boolean equals(Object obj) {
59      if (this == obj) {
60        return true;
61      }
62      if (obj == null) {
63        return false;
64      }
65      if (getClass() != obj.getClass()) {
66        return false;
67      }
68      AbstractEntity other = (AbstractEntity) obj;
69      if (id == null) {
70        if (other.id != null) {
71          return false;
72        }
73      } else if (!id.equals(other.id)) {
74        return false;
75      }
76      return true;
77    }
78
79  }
```

1.2.2 ConnectionManager.java

Dieses Interface spezifiziert den Verbindungsmanager der die Verbindungen verwaltet.

Listing 2: ConnectionManager.java

```
1  package at.fh.ooe.swe4.campina.persistence.api;
2
3  import java.sql.Connection;
4
5  /**
6   * This interface specifies an conection manager which provides connections for
7   * services. The implementation decides if the connections are either cached or
8   * created new each time {@link ConnectionManager#getClass()} is called.
9   *
10  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
11  * @date Jun 18, 2015
12  */
13 public interface ConnectionManager {
14
15   /**
16    * Returns a jdbc connection.
17    *
18    * @param batchStatements
19    *            true if multiple statements per transaction shall be supported
20    * @return the jdbc connection
21    */
22   public Connection getConnection(boolean batchStatements);
23 }
```

1.2.3 EntityManager.java

Dieses Interface spezifiziert den EntityManager, wobei jeweils eine Instanz für einen Typ eines Domänenmodells zuständig ist. Diese Spezifikation solles erleichtern mit den Entitäten und JDBC umzugehen.

Listing 3: EntityManager.java

```
1  package at.fh.ooe.swe4.campina.persistence.api;
2
3  import java.sql.Connection;
4  import java.sql.ResultSet;
5  import java.sql.SQLException;
6  import java.util.List;
7  import java.util.Objects;
8
9  import javax.persistence.Column;
10
11 /**
12  * This interface specifies an entity manager which is used for entity types .
13  *
14  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
15  * @date Jun 19, 2015
16  * @param <E>
17  *            the entity type the isntance is for
18  */
19 public interface EntityManager<E extends AbstractEntity> {
20   /**
21    * This is a helper class which hold the metadata from the @Column annotated
22    * methods which represent the dataabse columns
23    *
24    * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
25    * @date Jun 19, 2015
26    */
27   public static final class ColumnMetadata {
28
29     public final Column    column;
30     public final String    getter;
31     public final String    setter;
32     public final Class<?>  typeClass;
33
34     /**
35      * @param column
36      * @param methodName
37      * @param typeClass
38      */
39     public ColumnMetadata(Column column, String methodName, Class<?> typeClass) {
40       super();
41       Objects.requireNonNull(column);
42       Objects.requireNonNull(methodName);
43       Objects.requireNonNull(typeClass);
44
45       this.column = column;
46       this.getter = methodName;
47       this.setter = methodName.replace("get", "set");
48       this.typeClass = typeClass;
49     }
50
51   }
52
53   /**
54    * Saves or updates the given entity depending on set id or not.
55    *
```

```
56      * @param con
57      *            the underlying connection
58      * @param entity
59      *            the entity to save or update
60      * @return the saved or updated entity
61      * @throws SQLException
62      *             if the save or update fails due an sql error.
63      * @throws NullPointerException
64      *             if the entity is null
65      */
66     public E saveOrUpdate(Connection con, E entity) throws SQLException;
67
68     /**
69      * Deletes the given entity.
70      *
71      * @param con
72      *            the underlying connection
73      * @param entity
74      *            the entity to be deleted
75      * @throws SQLException
76      *             if the deletion fails
77      * @throws NullPointerException
78      *             if con or entity are null
79      */
80     public void delete(Connection con, E entity) throws SQLException;
81
82     /**
83      * Gets all entries from the backed entity type.
84      *
85      * @param con
86      *            the underlying connection
87      * @return the found entity
88      * @throws SQLException
89      *             if the entity could not be found
90      */
91     public List<E> byType(Connection con) throws SQLException;
92
93     /**
94      * Gets the entity by its id.
95      *
96      * @param con
97      *            the underlying connection
98      * @param id
99      *            TODO
100     * @return the given instance with filled result
101     * @throws SQLException
102     *             if the fetch fails
103     * @throws NullPointerException
104     *             if con or entity are null
105     */
106    public E byId(Connection con, Integer id) throws SQLException;
107
108    /**
109     * Gets the entities by custom query.
110     *
111     * @param con
112     *            the underlying connection
113     * @param query
114     *            the query to execute
115     * @param args
116     *            the arguments for the query
117     * @return the entity list
118     * @throws SQLException
```

```
119     *             if the query execution fails
120     */
121    public List<E> byQuery(Connection con, String query, Object... args) throws SQLException;
122
123    /**
124     * Fills the entity with the values of the result set. It is assumed that
125     * the columns in the same order as this entity managers relys on and that
126     * all of the columns are present.
127     *
128     * @param result
129     *             the result set
130     * @param entity
131     *             the entity to fill
132     * @param offset
133     *             the offset of the column index
134     * @throws SQLException
135     *              if the filling fails
136     */
137    public void fillEntity(ResultSet result, AbstractEntity entity, int offset) throws SQLException;
138
139    /**
140     * Gets the table name with schema (if set) of the backed entity type.
141     *
142     * @return the table name
143     */
144    public String getTableName();
145
146    /**
147     * Gets the column names of the backed entity in the form of name-1, name-2,
148     * ... , name-n
149     *
150     * @param prefix
151     *             TODO
152     *
153     * @return the backed entity column names
154     */
155    public String getColumnNames(String prefix);
156
157    /**
158     * Returns an unmodifiable list containing the column metadata.
159     *
160     * @return the column metadata
161     */
162    public List<ColumnMetadata> getColumnMeta();
163 }
```

1.2.4 User.java

Diese Klasse stellt den Benutzer auf der Datenbank da.

Listing 4: User.java

```
1  package at.fh.ooe.swe4.campina.persistence.api.entity;
2
3  import javax.persistence.Column;
4  import javax.persistence.Table;
5
6  import at.fh.ooe.swe4.campina.persistence.api.AbstractEntity;
7
8  /**
9   * The campina user.
10  *
11  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
12  * @date Jun 5, 2015
13  */
14 @Table(name = "USER", schema = "CAMPINA")
15 public class User extends AbstractEntity {
16
17   private static final long  serialVersionUID  = 94057306870207307L;
18
19   private String        firstName;
20   private String        lastName;
21   private String        email;
22   private String        username;
23   private String        password;
24   private Boolean        adminFlag      = Boolean.FALSE;
25   private Boolean        blockedFlag      = Boolean.FALSE;
26
27   /**
28    *
29    */
30   public User() {
31   }
32
33   /**
34    * @param id
35    * @param firstName
36    * @param lastName
37    * @param email
38    */
39   public User(Integer id, String firstName, String lastName, String email) {
40     super(id);
41     this.firstName = firstName;
42     this.lastName = lastName;
43     this.email = email;
44   }
45
46   @Column(name = "FIRST_NAME")
47   public String getFirstName() {
48     return firstName;
49   }
50
51   public void setFirstName(String firstName) {
52     this.firstName = firstName;
53   }
54
55   @Column(name = "LAST_NAME")
56   public String getLastName() {
57     return lastName;
58   }
```

```
59
60    public void setLastName(String lastName) {
61      this.lastName = lastName;
62    }
63
64    @Column(name = "EMAIL")
65    public String getEmail() {
66      return email;
67    }
68
69    public void setEmail(String email) {
70      this.email = email;
71    }
72
73    @Column(name = "USERNAME")
74    public String getUsername() {
75      return username;
76    }
77
78    public void setUsername(String username) {
79      this.username = username;
80    }
81
82    @Column(name = "PASSWORD")
83    public String getPassword() {
84      return password;
85    }
86
87    public void setPassword(String password) {
88      this.password = password;
89    }
90
91    @Column(name = "ADMIN_FLAG")
92    public Boolean getAdminFlag() {
93      return adminFlag;
94    }
95
96    public void setAdminFlag(Boolean adminFlag) {
97      this.adminFlag = adminFlag;
98    }
99
100   @Column(name = "BLOCKED_FLAG")
101   public Boolean getBlockedFlag() {
102     return blockedFlag;
103   }
104
105   public void setBlockedFlag(Boolean blockedFlag) {
106     this.blockedFlag = blockedFlag;
107   }
108 }
```

1.2.5 Menu.java

Diese Klasse stellt die Menukarte auf der Datenbank da.

Listing 5: User.java

```
1  package at.fh.ooe.swe4.campina.persistence.api.entity;
2
3  import javax.persistence.Column;
4  import javax.persistence.Table;
5
6  import at.fh.ooe.swe4.campina.persistence.api.AbstractEntity;
7  import at.fh.ooe.swe4.campina.persistence.api.entity.constants.Day;
8
9  /**
10  * The menu.
11  *
12  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
13  * @date Jun 5, 2015
14  */
15 @Table(name = "MENU", schema = "CAMPINA")
16 public class Menu extends AbstractEntity implements Comparable<Menu> {
17
18   private static final long  serialVersionUID  = 7569033478037865818L;
19
20   private Day          day;
21   private String        label;
22
23   /**
24    *
25    */
26   public Menu() {
27     super();
28   }
29
30   /**
31    *
32    * @param id
33    */
34   public Menu(Integer id) {
35     super(id);
36   }
37
38   /**
39    * @param id
40    * @param day
41    * @param label
42    */
43   public Menu(Integer id, Day day, String label) {
44     super(id);
45     this.day = day;
46     this.label = label;
47   }
48
49   @Column(name = "DAY")
50   public Day getDay() {
51     return day;
52   }
53
54   public void setDay(Day day) {
55     this.day = day;
56   }
57
58   @Column(name = "LABEL")
```

```
59    public String getLabel() {
60      return label;
61    }
62
63    public void setLabel(String label) {
64      this.label = label;
65    }
66
67    @Override
68    public int compareTo(Menu o) {
69      return Integer.valueOf(day.ordinal)
70              .compareTo(o.getDay().ordinal);
71    }
72  }
```

1.2.6 MenuEntry.java

Diese Klasse stellt den Menu Eintrag auf der Datenbank da.

Listing 6: MenuEntry.java

```
1  package at.fh.ooe.swe4.campina.persistence.api.entity;
2
3  import java.math.BigDecimal;
4  import java.util.HashSet;
5  import java.util.Set;
6
7  import javax.persistence.Column;
8  import javax.persistence.Table;
9
10 import at.fh.ooe.swe4.campina.persistence.api.AbstractEntity;
11
12 /**
13  * The menu entry
14  *
15  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
16  * @date Jun 5, 2015
17  */
18 @Table(name = "MENU_ENTRY", schema = "CAMPINA")
19 public class MenuEntry extends AbstractEntity implements Comparable<MenuEntry> {
20
21   private static final long  serialVersionUID  = -1771596592654083392L;
22
23   private Integer        ordinal;
24   private String         label;
25   private BigDecimal      price;
26   private Menu        menu;
27
28   /**
29    *
30    */
31   public MenuEntry() {
32     super();
33   }
34
35   /**
36    * @param id
37    */
38   public MenuEntry(Integer id) {
39     super(id);
40   }
41
42   /**
43    * @param id
44    * @param ordinal
45    * @param label
46    * @param price
47    * @param menu
48    */
49   public MenuEntry(Integer id, Integer ordinal, String label, BigDecimal price, Menu menu) {
50     this(id);
51     this.ordinal = ordinal;
52     this.label = label;
53     this.price = price;
54     this.menu = menu;
55   }
56
57   @Column(name = "ORDINAL")
58   public Integer getOrdinal() {
```

```
59      return ordinal;
60    }
61
62    public void setOrdinal(Integer ordinal) {
63      this.ordinal = ordinal;
64    }
65
66    @Column(name = "LABEL")
67    public String getLabel() {
68      return label;
69    }
70
71    public void setLabel(String label) {
72      this.label = label;
73    }
74
75    @Column(name = "PRICE")
76    public BigDecimal getPrice() {
77      return price;
78    }
79
80    public void setPrice(BigDecimal price) {
81      this.price = price;
82    }
83
84    @Column(name = "MENU_ID")
85    public Menu getMenu() {
86      return menu;
87    }
88
89    public void setMenu(Menu menu) {
90      this.menu = menu;
91    }
92
93    @Override
94    public int compareTo(MenuEntry o) {
95      if ((getId() == null) && (o.getId() == null)) {
96        return 0;
97      } else if (getId() == null) {
98        return -1;
99      } else if (o.getId() == null) {
100       return 1;
101     } else {
102       return getId().compareTo(o.getId());
103       // return day.compareTo(o.getDay());
104     }
105   }
106 }
```

1.2.7 Order.java

Diese Klasse stellt die Bestellung auf der Datenbank da.

Listing 7: Order.java

```java
package at.fh.ooe.swe4.campina.persistence.api.entity;

import java.util.Calendar;

import javax.persistence.Column;
import javax.persistence.Table;

import at.fh.ooe.swe4.campina.persistence.api.AbstractEntity;

/**
 * The order on the database.
 *
 * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
 * @date Jun 5, 2015
 */
@Table(name = "ORDER", schema = "CAMPINA")
public class Order extends AbstractEntity {

  private static final long  serialVersionUID  = -4218683353334647328L;

  private User        user;
  private MenuEntry      menuEntry;
  private Calendar      orderDate;
  private Calendar      collectDate;

  /**
   * @param id
   * @param user
   * @param menuEntry
   * @param orderDate
   * @param collectDate
   */
  public Order(Integer id, User user, MenuEntry menuEntry, Calendar orderDate, Calendar
   ↪  collectDate) {
    super(id);
    this.user = user;
    this.menuEntry = menuEntry;
    this.orderDate = orderDate;
    this.collectDate = collectDate;
  }

  /**
   *
   */
  public Order() {
  }

  /**
   * @param id
   */
  public Order(Integer id) {
    super(id);
  }

  @Column(name = "USER_ID")
  public User getUser() {
    return user;
  }
```

```
58
59    public void setUser(User user) {
60      this.user = user;
61    }
62
63    @Column(name = "MENU_ENTRY_ID")
64    public MenuEntry getMenuEntry() {
65      return menuEntry;
66    }
67
68    public void setMenuEntry(MenuEntry menuEntry) {
69      this.menuEntry = menuEntry;
70    }
71
72    @Column(name = "ORDER_DATE")
73    public Calendar getOrderDate() {
74      return orderDate;
75    }
76
77    public void setOrderDate(Calendar orderDate) {
78      this.orderDate = orderDate;
79    }
80
81    @Column(name = "COLLECT_DATE")
82    public Calendar getCollectDate() {
83      return collectDate;
84    }
85
86    public void setCollectDate(Calendar collectDate) {
87      this.collectDate = collectDate;
88    }
89  }
```

1.3 Source swe-campina-data-model-impl

Folgend ist die Implementierung der Spezifikation *swe-campina-data-model-api* angeführt. Sie wurde auf eine MySql Datenbank ausgelegt.

1.3.1 ConnectionManagerImpl.java

Diese Klasse ist die Implementierung der Spezifikation `ConnectionManager` dar. Sie verwaltet die Verbindungen zu einer Datenbank.

Listing 8: ConnectionManagerImpl.java

```
1  package at.fh.ooe.swe4.campina.persistence.impl;
2
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.util.Objects;
6
7  import org.apache.log4j.Logger;
8
9  import at.fh.ooe.swe4.campina.persistence.api.ConnectionManager;
10
11 /**
12  * This is the default implementation for the {@link ConnectionManager}
13  * interface. Each time the method
14  * {@link ConnectionManager#getConnection(boolean)} is called a new connection
15  * will be created with the provided metadata.
16  *
17  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
18  * @date Jun 18, 2015
19  */
20 public class ConnectionManagerImpl implements ConnectionManager {
21
22   private final DbMetadata  metadata;
23
24   private static final Logger  log  = Logger.getLogger(ConnectionManagerImpl.class);
25
26   /**
27    * This class holds the metadata information for the used database.
28    *
29    * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
30    * @date Jun 18, 2015
31    */
32   public static final class DbMetadata {
33     public final String    driverName;
34     public final String    url;
35     public final String    username;
36     public final String    password;
37     public final int    isolationLevel;
38     public final boolean  useCredentials;
39
40     /**
41      * For user connections
42      *
43      * @param driverName
44      *            the driver class name
45      * @param url
46      *            the db location
47      * @param username
48      *            the username
49      * @param password
50      *            the users password
```

```
51        * @param isolationLevel
52        *            the isolation level
53        * @see Connection for the isolation levels
54        */
55       public DbMetadata(String driverName, String url, String username, String password, int
          ↪  isolationLevel) {
56         super();
57         Objects.requireNonNull(driverName);
58         Objects.requireNonNull(url);
59         Objects.requireNonNull(username);
60         Objects.requireNonNull(password);
61
62         this.driverName = driverName;
63         this.url = url;
64         this.username = username;
65         this.password = password;
66         this.isolationLevel = isolationLevel;
67         this.useCredentials = Boolean.TRUE;
68       }
69
70       /**
71        * For anonymous connections
72        *
73        * @param driverName
74        *            the dirver class name
75        * @param url
76        *            the db location
77        */
78       public DbMetadata(String driverName, String url, int isolationLevel) {
79         super();
80         this.driverName = driverName;
81         this.url = url;
82         this.username = null;
83         this.password = null;
84         this.isolationLevel = isolationLevel;
85         this.useCredentials = Boolean.FALSE;
86       }
87
88       @Override
89       public String toString() {
90         final StringBuilder sb = new StringBuilder(100);
91         sb.append(String.format("%1$10s", "driver: "))
92           .append(driverName)
93           .append(System.lineSeparator())
94           .append(String.format("%1$10s", "url: "))
95           .append(url)
96           .append(System.lineSeparator())
97           .append(String.format("%1$10s", "ISO.: "))
98           .append(isolationLevel)
99           .append(System.lineSeparator());
100        if (useCredentials) {
101          sb.append(String.format("%1$10s", "username: "))
102            .append(username)
103            .append(System.lineSeparator())
104            .append(String.format("%1$10s", "password: "))
105            .append(password)
106            .append(System.lineSeparator());
107        }
108        return sb.toString();
109      }
110    }
111
112    /**
```

```
113     * @param metadata
114     *            the database metadata for creating the connections
115     */
116    public ConnectionManagerImpl(final DbMetadata metadata) {
117      Objects.requireNonNull(metadata);
118
119      setup(metadata);
120
121      this.metadata = metadata;
122    }
123
124    @Override
125    public Connection getConnection(boolean batchStatements) {
126      try {
127        final Connection con;
128        if (metadata.useCredentials) {
129          con = DriverManager.getConnection(metadata.url, metadata.username, metadata.password);
130        } else {
131          con = DriverManager.getConnection(metadata.url);
132        }
133        con.setTransactionIsolation(metadata.isolationLevel);
134        con.setAutoCommit(!batchStatements);
135        return con;
136      } catch (Exception e) {
137        throw new IllegalStateException("Should not happen :(");
138      }
139    }
140
141    // ########################################################################
142    // Private Section
143    // ########################################################################
144    /**
145     * Setups the connection manager with the provide config and tries to
146     * connect to the backing database.
147     *
148     * @param metadata
149     *            the provide database metadata
150     * @throws IllegalArgumentException
151     *             if the metadata is invalid
152     */
153    private void setup(final DbMetadata metadata) throws IllegalArgumentException {
154      final String ln = System.lineSeparator();
155      log.info("------------------------------------------------");
156      log.info("Beginn Setup Connection manager");
157      log.info("------------------------------------------------");
158      log.info("Provided DbMetadata:"
159          + ln
160          + ln
161          + metadata.toString()
162        );
163
164      try {
165        Class.forName(metadata.driverName);
166      } catch (ClassNotFoundException e) {
167        log.error("driver class not found", e);
168        throw new IllegalArgumentException("Driver class '" + metadata.driverName + "' not found");
169      }
170
171      try {
172        log.info("Trying to connect to database: " + metadata.url);
173        final Connection con;
174        if (metadata.useCredentials) {
175          con = DriverManager.getConnection(metadata.url, metadata.username, metadata.password);
```

```
176       } else {
177         con = DriverManager.getConnection(metadata.url);
178       }
179       log.info("Trying to set isolation level: " + metadata.isolationLevel);
180       con.setTransactionIsolation(metadata.isolationLevel);
181       log.info("Trying to close connection");
182       con.close();
183     } catch (Exception e) {
184       log.error("Setup failed for: ", e);
185       throw new IllegalArgumentException("Exception occured during initial setup", e);
186     }
187 
188     log.info("-----------------------------------------------");
189     log.info("End Setup Connection manager");
190     log.info("-----------------------------------------------");
191   }
192 }
```

1.3.2 DbConfigParam.java

Diese Enumeration spezifiziert die Parameter für die Konfigurationsdatei *db-config.properties* und lädt diese auch.

Listing 9: DbConfigParam.java

```
1  package at.fh.ooe.swe4.campina.persistence.impl;
2
3  import java.util.ResourceBundle;
4
5  /**
6   * This enum specifies the db-config.properties used parameters
7   *
8   * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
9   * @date Jun 18, 2015
10  */
11 public enum DbConfigParam {
12   DRIVER,
13   URL,
14   USER,
15   PASSWORD,
16   ISOLATION;
17
18   /**
19    * Gets the provided configuration
20    */
21   private static final ResourceBundle  dbConfig  = ResourceBundle.getBundle("db-config");
22
23   /**
24    * Returns the String representation of the parameter value
25    *
26    * @return the parameter value
27    */
28   public String val() {
29     return dbConfig.getString(this.name());
30   }
31 }
```

1.3.3 EntityManagerImpl.java

Diese Klasse stellt die Implementierung der SPezifikation `EntityManager` dar. Sie soll den Übergang von Domänen Modell auf die Datenbank erleichtern, wobei sich hier sehr an JPA + Hibernate orientiert wurde.

Listing 10: EntityManagerImpl.java

```
1  package at.fh.ooe.swe4.campina.persistence.impl;
2
3  import java.lang.reflect.Method;
4  import java.math.BigDecimal;
5  import java.sql.Connection;
6  import java.sql.PreparedStatement;
7  import java.sql.ResultSet;
8  import java.sql.SQLException;
9  import java.sql.Timestamp;
10 import java.sql.Types;
11 import java.util.ArrayList;
12 import java.util.Arrays;
13 import java.util.Calendar;
14 import java.util.Collections;
15 import java.util.HashMap;
16 import java.util.LinkedList;
17 import java.util.List;
18 import java.util.Map;
19 import java.util.Objects;
20 import java.util.concurrent.ConcurrentHashMap;
21
22 import javax.persistence.Column;
23 import javax.persistence.Table;
24
25 import org.apache.commons.lang.StringUtils;
26 import org.apache.log4j.Logger;
27
28 import at.fh.ooe.swe4.campina.persistence.api.AbstractEntity;
29 import at.fh.ooe.swe4.campina.persistence.api.EntityManager;
30
31 /**
32  * This is a light weight entity manager which allows to perform JPA like
33  * operations on the backed entity.
34  *
35  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
36  * @date Jun 18, 2015
37  * @param <E>
38  */
39 public class EntityManagerImpl<E extends AbstractEntity> implements EntityManager<E> {
40
41   private final Class<E>                                clazz;
42   private final Table                                   table;
43   private List<ColumnMetadata>                             columnMetataDataList;
44
45   private static final Map<Class<? extends AbstractEntity>, Map<Statement,
      ↪  String>>  cache          = new ConcurrentHashMap<Class<? extends AbstractEntity>,
      ↪  Map<Statement, String>>();
46   private static final String                           INSERT_TEMPLATE      = "INSERT INTO %s
      ↪  (%s, version) VALUES (%s, 1);";
47   private static final String                           UPDATE_TEMPLATE      = "UPDATE %s SET
      ↪  %s, version=version+1 WHERE id=?;";
48   private static final String                           DELETE_TEMPLATE      = "DELETE FROM %s
      ↪  WHERE id=?;";
49   private static final String                           SELECT_BY_ID_TEMPLATE  = "SELECT id,
      ↪  %s, version FROM %s WHERE id=?";
```

```
50  private static final String                                SELECT_ID_BY_ID_VERSION  = "SELECT id
     ↪  FROM %s WHERE id=? AND version=?";
51  private static final String                                SELECT_BY_TYPE      = "SELECT id, %s,
     ↪  version FROM %s";
52  private static final String                                SELECT_LAST_INSERT_ID  = "SELECT
     ↪  LAST_INSERT_ID();";
53  private static final Logger                                log            =
     ↪  Logger.getLogger(EntityManagerImpl.class);
54  private static final List<String>                            EXCLUDE_FIELDS      =
     ↪  Arrays.asList(new String[] {
55                                                                        "id",
56                                                                        "version"
57                                                          });
58
59  /**
60   * This enumeration specifies the supported statements.
61   *
62   * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
63   * @date Jun 19, 2015
64   */
65  private static enum Statement {
66    INSERT,
67    UPDATE,
68    DELETE,
69    SELECT_BY_ID,
70    SELECT_ID_BY_ID_VERSION,
71    SELECT_BY_TYPE,
72    SELECT_LAST_INSERT_ID;
73  }
74
75  /**
76   * @param clazz
77   *            the backed entity class
78   * @throws NullPointerException
79   *             <ul>
80   *             <li>clazz is null</li>
81   *             <li>clazz has no @Table annotation</li>
82   *             </ul>
83   */
84  public EntityManagerImpl(final Class<E> clazz) {
85    Objects.requireNonNull(clazz);
86    Objects.requireNonNull(clazz.getAnnotation(Table.class), "Given entity class has no Table
     ↪  annotation");
87
88    this.clazz = clazz;
89    this.table = clazz.getAnnotation(Table.class);
90
91    setup();
92  }
93
94  @Override
95  public E saveOrUpdate(final Connection con, final E entity) throws SQLException {
96    Objects.requireNonNull(con);
97    Objects.requireNonNull(entity);
98
99    final Statement stmt = (entity.getId() == null) ? Statement.INSERT : Statement.UPDATE;
100   String sql = cache.get(clazz)
101             .get(stmt);
102   try (final PreparedStatement pstmt = con.prepareStatement(sql);) {
103     final List<Object> values = getValues(entity);
104
105     for (int i = 0; i < columnMetataDataList.size(); i++) {
106       final ColumnMetadata col = columnMetataDataList.get(i);
```

```
107         final Object value = values.get(i);
108         final int sqlType = converttoSqlType(col.typeClass);
109
110         if (value == null) {
111           pstmt.setNull(i + 1, sqlType);
112         } else {
113           pstmt.setObject(i + 1, convertValueFromEntity(value), sqlType);
114         }
115       }
116
117       if (entity.getId() != null) {
118         pstmt.setInt(columnMetataDataList.size() + 1, entity.getId());
119       }
120
121       pstmt.executeUpdate();
122
123       // set id
124       if (entity.getId() == null) {
125         sql = SELECT_LAST_INSERT_ID;
126         final PreparedStatement lastInsertIdPstmt = con.prepareStatement(sql);
127         final ResultSet resultId = lastInsertIdPstmt.executeQuery();
128         resultId.next();
129         entity.setId(resultId.getInt(1));
130         entity.setVersion(Long.valueOf(1));
131       } else {
132         entity.setVersion(entity.getVersion() + 1);
133       }
134     } catch (Throwable e) {
135       log.error("Error on saveOrUpdate");
136       log.error(sql);
137       throw e;
138     }
139     return entity;
140   }
141
142   @Override
143   public void delete(final Connection con, final E entity) throws SQLException {
144     Objects.requireNonNull(con);
145     Objects.requireNonNull(entity);
146     Objects.requireNonNull(entity.getId(), "Cannot delete entity with null id");
147
148     // check for existing entity with set id
149     checkForExistingWithVersion(con, entity);
150
151     // delete entity
152     final String sql = cache.get(clazz)
153                 .get(Statement.DELETE);
154     try (
155         final PreparedStatement pstmt = con.prepareStatement(sql);) {
156       System.out.println(cache.get(clazz)
157                   .get(Statement.DELETE));
158       pstmt.setInt(1, entity.getId());
159       pstmt.executeUpdate();
160     } catch (Throwable e) {
161       log.error("Error on delete");
162       log.error(sql);
163       throw e;
164     }
165   }
166
167   @Override
168   public E byId(final Connection con, Integer id) throws SQLException {
169     Objects.requireNonNull(con);
```

```
170     Objects.requireNonNull(id);
171
172     // full by id
173     final E entity = newEntity();
174     final String sql = cache.get(clazz)
175                 .get(Statement.SELECT_BY_ID);
176     try (
177         final PreparedStatement pstmt = con.prepareStatement(sql);) {
178       pstmt.setInt(1, id);
179       final ResultSet result = pstmt.executeQuery();
180       if (result.next()) {
181         entity.setId(result.getInt(1));
182         fillEntity(result, entity, 1);
183         entity.setVersion(result.getLong(columnMetataDataList.size() + 2));
184       } else {
185         throw new SQLException("Entity not found for id");
186       }
187     } catch (Throwable e) {
188       log.error("Error on byId");
189       log.error(sql);
190       throw e;
191     }
192     return entity;
193   }
194
195   @Override
196   public List<E> byType(final Connection con) throws SQLException {
197     Objects.requireNonNull(con);
198
199     final List<E> entities = new ArrayList<>();
200     try (
201         final PreparedStatement pstmt = con.prepareStatement(cache.get(clazz)
202                                       .get(Statement.SELECT_BY_TYPE));) {
203       final ResultSet result = pstmt.executeQuery();
204
205       while (result.next()) {
206         final E entity = newEntity();
207         entity.setId(result.getInt(1));
208         fillEntity(result, entity, 1);
209         entity.setVersion(result.getLong(columnMetataDataList.size() + 2));
210         entities.add(entity);
211       }
212     }
213     return entities;
214   }
215
216   @Override
217   public List<E> byQuery(final Connection con, final String query, Object... args) throws
      ↪  SQLException {
218     Objects.requireNonNull(query);
219     Objects.requireNonNull(args);
220
221     final List<E> entities = new ArrayList<>();
222     try (final PreparedStatement pstmt = con.prepareStatement(query)) {
223       for (int i = 0; i < args.length; i++) {
224         pstmt.setObject(i + 1, args[i]);
225       }
226       final ResultSet result = pstmt.executeQuery();
227       while (result.next()) {
228         final E entity = newEntity();
229         entity.setId(result.getInt(1));
230         fillEntity(result, entity, 1);
231         entity.setVersion(result.getLong(columnMetataDataList.size() + 2));
```

```
232         entities.add(entity);
233       }
234     } catch (SQLException e) {
235       log.error("Error on bybyQuery");
236       log.error(query);
237       throw new SQLException("Could not execute custom query", e);
238     }
239
240     return entities;
241   }
242
243   @Override
244   public List<ColumnMetadata> getColumnMeta() {
245     return Collections.unmodifiableList(columnMetataDataList);
246   }
247
248   @Override
249   public String getTableName() {
250     return table.name();
251   }
252
253   @Override
254   public String getColumnNames(String prefix) {
255     final List<String> names = new ArrayList<>(columnMetataDataList.size());
256     for (ColumnMetadata colMeta : columnMetataDataList) {
257       final String name;
258       if (prefix != null) {
259         name = prefix + "." + colMeta.column.name()
260                             .toLowerCase();
261       } else {
262         name = colMeta.column.name()
263                     .toLowerCase();
264       }
265       names.add(name);
266     }
267     return StringUtils.join(names, ",");
268   }
269
270   /**
271    * Files the given entity instance with the in the {@link ResultSet}
272    * contained values. The result set ust hold all columns including the id.
273    *
274    * @param result
275    * @param entity
276    * @throws SQLException
277    */
278   @Override
279   public void fillEntity(ResultSet result, AbstractEntity entity, int offset) throws SQLException
    ↪  {
280     Objects.requireNonNull(result);
281     Objects.requireNonNull(entity);
282
283     ColumnMetadata colMeta = null;
284     Object value = null;
285     try {
286       for (int i = 0; i < columnMetataDataList.size(); i++) {
287         colMeta = columnMetataDataList.get(i);
288         value = result.getObject(offset + i + 1);
289         final Object convertedValue;
290         if (value != null) {
291           convertedValue = convertValueToEntity(colMeta, value);
292         } else {
293           convertedValue = value;
```

```
294         }
295         clazz.getMethod(colMeta.setter, colMeta.typeClass)
296             .invoke(entity, convertedValue);
297       }
298     } catch (Throwable e) {
299       if (value != null) {
300
301       }
302       throw new java.lang.IllegalStateException("Could not fill entity: " + ((colMeta != null) ?
          ↪  colMeta.setter : ""), e);
303     }
304   }
305
306   /**
307    * Setup this service util by creating all supported statements for the
308    * backed entity class if and only if no other instance which backs this
309    * entity type has already created the statements.
310    */
311   public void setup() {
312     log.info("-------------------------------------------");
313     log.info("Started Setup for " + this.getClass()
314                     .getSimpleName() + "<" + clazz.getSimpleName() + ">");
315     log.info("-------------------------------------------");
316
317     Map<Statement, String> newCache = new HashMap<Statement, String>();
318     this.columnMetataDataList = getColumnMetataDataList();
319     Map<Statement, String> statementCache = cache.putIfAbsent(clazz, newCache);
320
321     // only if no other instance has initialized cache for this class
322     if (statementCache == null) {
323       final List<String> columnNames = new ArrayList<>(columnMetataDataList.size());
324       final List<String> parameters = new ArrayList<>(columnMetataDataList.size());
325       final List<String> updateParameters = new ArrayList<>(columnMetataDataList.size());
326       for (ColumnMetadata col : columnMetataDataList) {
327         final String name = col.column.name()
328                         .toLowerCase();
329         columnNames.add(name);
330         parameters.add("?");
331         updateParameters.add(name + "=?");
332       }
333
334       // -- prepare statements --
335       final String cols = StringUtils.join(columnNames, ", ");
336       final String params = StringUtils.join(parameters, ", ");
337       final String updateParams = StringUtils.join(updateParameters, ", ");
338       final String tableName = (table.schema().isEmpty()) ? table.name()
339                                     .toLowerCase() : (new
                                      ↪  StringBuilder(table.schema()).append(".")
340                                                     .append(table.name())
341                                                     .toString().toLowerCase());
342
343       log.info("Creating " + StringUtils.join(Statement.values(), ", ") + " Statements");
344       cache.get(clazz)
345           .put(Statement.INSERT, String.format(INSERT_TEMPLATE, tableName, cols, params));
346       cache.get(clazz)
347           .put(Statement.UPDATE, String.format(UPDATE_TEMPLATE, tableName, updateParams));
348       cache.get(clazz)
349           .put(Statement.DELETE, String.format(DELETE_TEMPLATE, tableName));
350       cache.get(clazz)
351           .put(Statement.SELECT_BY_ID, String.format(SELECT_BY_ID_TEMPLATE, cols, tableName));
352       cache.get(clazz)
353           .put(Statement.SELECT_ID_BY_ID_VERSION, String.format(SELECT_ID_BY_ID_VERSION,
              ↪  tableName));
```

```
354         cache.get(clazz)
355             .put(Statement.SELECT_BY_TYPE, String.format(SELECT_BY_TYPE, cols, tableName));
356
357         log.info(cache.get(clazz)
358                 .get(Statement.INSERT));
359         log.info(cache.get(clazz)
360                 .get(Statement.UPDATE));
361         log.info(cache.get(clazz)
362                 .get(Statement.SELECT_BY_ID));
363         log.info(cache.get(clazz)
364                 .get(Statement.SELECT_ID_BY_ID_VERSION));
365         log.info(cache.get(clazz)
366                 .get(Statement.SELECT_BY_TYPE));
367         log.info(SELECT_LAST_INSERT_ID);
368       }
369       log.info("-----------------------------------------");
370       log.info("Finished Setup for " + this.getClass()
371                           .getSimpleName() + "<" + clazz.getSimpleName() + ">");
372       log.info("-----------------------------------------");
373     }
374
375     // ########################################################################
376     // Private section
377     // ########################################################################
378
379     /**
380      * Checks if the given entity exists on the database with its set id and
381      * version.
382      *
383      * @param con
384      *            the underlying connection
385      * @param entity
386      *            the entity to check
387      * @throws SQLException
388      *             if the entity is not found with its id and version on the
389      *             database
390      * @throws NullPointerException
391      *             <ul>
392      *             <li>con is null</li>
393      *             <li>entity is null</li>
394      *             <li>entity id is null</li>
395      *             </ul>
396      */
397     private void checkForExistingWithVersion(final Connection con, final E entity) throws
      ↪  SQLException {
398       Objects.requireNonNull(con);
399       Objects.requireNonNull(entity);
400       Objects.requireNonNull(entity.getId());
401
402       // entity does exist with its set version
403       final PreparedStatement existPstmt = con.prepareStatement(cache.get(clazz)
404                                       .get(Statement.SELECT_ID_BY_ID_VERSION));
405       existPstmt.setInt(1, entity.getId());
406       existPstmt.setLong(2, entity.getVersion());
407       if (!existPstmt.executeQuery()
408               .next()) {
409         throw new SQLException("Entity not found for set id and version !!! id: " + entity.getId()
410                                                 .toString() + " | version: " + entity.getVersion()
411                                                                 .toString());
412       }
413     }
414
415     /**
```

```
416     * Gets the {@link Column} annotations of the backed entity class.
417     *
418     * @return the list of column annotations
419     */
420    private List<ColumnMetadata> getColumnMetataDataList() {
421      try {
422        final List<ColumnMetadata> columns = new LinkedList<>();
423
424        final Method[] methods = clazz.getMethods();
425        for (Method method : methods) {
426          final Column col = method.getAnnotation(Column.class);
427          if ((col != null) && (!EXCLUDE_FIELDS.contains(col.name()
428                                  .toLowerCase()))) {
429            columns.add(new ColumnMetadata(col, method.getName(), method.getReturnType()));
430          }
431        }
432        return Collections.unmodifiableList(columns);
433      } catch (Throwable e) {
434        throw new IllegalArgumentException("Could not get column information", e);
435      }
436    }
437
438    /**
439     * Gets a list of values from the given entity
440     *
441     * @param entity
442     *            the entity to retrieve values from
443     * @return the list of the retrieved values
444     * @throws IllegalStateException
445     *             if the value could not be retrieved by reflection
446     * @throws NullPointerException
447     *             if the entity is null
448     */
449    private List<Object> getValues(final E entity) throws IllegalStateException {
450      Objects.requireNonNull(entity);
451
452      final List<Object> values = new LinkedList<>();
453
454      for (ColumnMetadata colMeta : columnMetataDataList) {
455        try {
456          values.add(clazz.getMethod(colMeta.getter)
457                  .invoke(entity));
458        } catch (Throwable e) {
459          throw new IllegalArgumentException("Could not invoke '" + colMeta.getter + "'", e);
460        }
461      }
462      return values;
463    }
464
465    /**
466     * Converts the sql result returned value to the proper entity type.
467     *
468     * @param colMeta
469     *            the current column
470     * @param value
471     *            the current column value
472     * @return the converted value
473     * @throws IllegalStateException
474     *             if the many to one relation entity could not be instantiated
475     */
476    private Object convertValueToEntity(ColumnMetadata colMeta, Object value) {
477      if (value == null) {
478        return value;
```

```
479     } else if (Calendar.class.isAssignableFrom(colMeta.typeClass)) {
480       final Calendar cal = Calendar.getInstance();
481       cal.setTimeInMillis(((Timestamp) value).getTime());
482       return cal;
483     } else if (BigDecimal.class.isAssignableFrom(colMeta.typeClass)) {
484       return ((BigDecimal) value);
485     } else if (Enum.class.isAssignableFrom(colMeta.typeClass)) {
486       return Enum.valueOf((Class<Enum>) colMeta.typeClass, (String) value);
487     }
488     if ((AbstractEntity.class.isAssignableFrom(colMeta.typeClass))) {
489       try {
490         final AbstractEntity fk = (AbstractEntity) colMeta.typeClass.newInstance();
491         fk.setId((Integer) value);
492         return fk;
493       } catch (Exception e) {
494         throw new IllegalStateException("Could not create many-to-one relation entity '" +
          ↪  colMeta.typeClass.getName() + "'");
495       }
496 
497     } else if (Boolean.class.isAssignableFrom(colMeta.typeClass)) {
498       return (((Integer) value).equals(1)) ? Boolean.TRUE : Boolean.FALSE;
499     }
500     return value;
501   }
502 
503   /**
504    * Converts the given value to the proper sql type.
505    *
506    * @param value
507    *            the value ot be converted
508    * @return the converted value
509    * @throws IllegalStateException
510    *             if the value type cannot be converted
511    */
512   private Object convertValueFromEntity(final Object value) {
513     Objects.requireNonNull(value);
514 
515     if (value instanceof Calendar) {
516       return new Timestamp(((Calendar) value).getTimeInMillis());
517     } else if (value instanceof AbstractEntity) {
518       return ((AbstractEntity) value).getId();
519     } else if (value instanceof BigDecimal) {
520       return ((BigDecimal) value).doubleValue();
521     } else if (value instanceof Enum) {
522       return ((Enum) value).name();
523     } else {
524       return value;
525     }
526   }
527 
528   /**
529    * Converts the entity value type to the proper sql type.
530    *
531    * @param clazz
532    *            the return type class of the entity getter
533    * @return the integer representing the sql type
534    */
535   private int converttoSqlType(final Class<?> clazz) {
536     Objects.requireNonNull(clazz);
537 
538     if (String.class.isAssignableFrom(clazz)) {
539       return Types.VARCHAR;
540     } else if (Integer.class.isAssignableFrom(clazz)) {
```

```
541         return Types.INTEGER;
542       } else if (BigDecimal.class.isAssignableFrom(clazz)) {
543         return Types.DOUBLE;
544       } else if (Boolean.class.isAssignableFrom(clazz)) {
545         return Types.SMALLINT;
546       } else if (AbstractEntity.class.isAssignableFrom(clazz)) {
547         return Types.INTEGER;
548       } else if (Enum.class.isAssignableFrom(clazz)) {
549         return Types.VARCHAR;
550       } else if (Calendar.class.isAssignableFrom(clazz)) {
551         return Types.TIMESTAMP;
552       }
553       return 0;
554     }
555
556     /**
557      * Creates a new entity instance of the backed tpye
558      *
559      * @return the new instance
560      * @throws IllegalStateException
561      *             if the instantiation fails
562      */
563     private E newEntity() {
564       final E entity;
565       try {
566         entity = clazz.newInstance();
567       } catch (Throwable e) {
568         throw new IllegalStateException("Could not create new isntance");
569       }
570       return entity;
571     }
572 }
```

1.3.4 create-campina-schema.sql

Diese Datei enthält die DDL Befehle zum Anlegen des Campina Schemas.

Listing 11: create-campina-schema.sql

```
CREATE SCHEMA if not exists campina;
USE campina;
DROP TABLE IF EXISTS campina.order;
DROP TABLE IF EXISTS campina.menu_entry;
DROP TABLE IF EXISTS campina.menu;
DROP TABLE IF EXISTS campina.user;

CREATE TABLE campina.user (
  id INTEGER NOT NULL AUTO_INCREMENT,
  version INTEGER NOT NULL default 1,
  first_name VARCHAR(255) NOT NULL,
  last_name VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL,
  username VARCHAR(255) NOT NULL,
  password VARCHAR(255) NOT NULL,
  admin_flag smallint NOT NULL default 0,
  blocked_flag smallint NOT NULL default 0,
  PRIMARY KEY(id)
);

CREATE TABLE campina.order(
  id INTEGER NOT NULL auto_increment,
  version INTEGER NOT NULL default 1,
  order_date TIMESTAMP NOT NULL default CURRENT_TIMESTAMP,
  collect_date TIMESTAMP NOT NULL,
  user_id INTEGER NOT NULL,
  menu_entry_id INTEGER NOT NULL,
  PRIMARY KEY(id)
);

CREATE TABLE campina.menu(
  id INTEGER NOT NULL auto_increment,
  version INTEGER NOT NULL default 1,
  day VARCHAR(255) NOT NULL,
  label VARCHAR(255) NOT NULL,
  PRIMARY KEY (id)
);

CREATE TABLE campina.menu_entry(
  id INTEGER NOT NULL auto_increment,
  version INTEGER NOT NULL default 1,
  ordinal INTEGER NOT NULL,
  label VARCHAR(255) NOT NULL,
  price DECIMAL(4,2) NOT NULL,
  menu_id INTEGER NOT NULL,
  PRIMARY KEY (id)
);

ALTER TABLE campina.order
ADD CONSTRAINT fk_menu_entry_id
FOREIGN KEY (menu_entry_id)
REFERENCES campina.menu_entry(id);

ALTER TABLE campina.order
ADD CONSTRAINT fk_user_id
FOREIGN KEY (user_id)
REFERENCES campina.user(id);

```

```
59 ALTER TABLE campina.menu_entry
60 ADD CONSTRAINT fk_menu_id
61 FOREIGN KEY (menu_id)
62 REFERENCES campina.menu(id);
```

1.3.5 db-config.properties

Diese Datei enthält die Konfiguration für JDBC.

Listing 12: db-config.properties

```
1  ######################################################################
2  # This properties file provides the ddb connection information #
3  #                                                                    #
4  # created: 2015-06-18                                                #
5  # author: Thomas Herzog                                              #
6  ######################################################################
7  DRIVER=com.mysql.jdbc.Driver
8  URL=jdbc:mysql://localhost:3307/campina
9  USER=root
10 PASSWORD=root
11 ISOLATION=2
```

1.4 Source swe-campina-dao-api

Folgend ist der Source des Projekts *swe-campina-dao-api* angeführt, welcher die Sources für den Datenbankzugriff des Domänen Modells enthält.

1.4.1 AbstractRemoteDao.java

Diese Klasse stellt die Wurzelklasse dar, von der alle Remote DAOs erben.

Listing 13: AbstractRemoteDao.java

```
1  package at.fh.ooe.swe4.campina.dao.api;
2
3  import java.rmi.RemoteException;
4  import java.rmi.server.UnicastRemoteObject;
5  import java.util.Objects;
6
7  import at.fh.ooe.swe4.campina.persistence.api.ConnectionManager;
8
9  /**
10  * This class is the base class for entity DAOs which are accessible via RMI and
11  * therefore remote DAOs.
12  *
13  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
14  * @date Jun 19, 2015
15  */
16 public class AbstractRemoteDao extends UnicastRemoteObject {
17
18   private static final long      serialVersionUID  = -6317106515178653903L;
19
20   protected final ConnectionManager  connectionManager;
21
22   /**
23    * @param connectionManager
24    * @throws RemoteException
25    * @throws NullPointerException
26    *             if the connection manager is null
27    */
28   public AbstractRemoteDao(ConnectionManager connectionManager) throws RemoteException {
29     super();
30     Objects.requireNonNull(connectionManager);
31
32     this.connectionManager = connectionManager;
33   }
34
35 }
```

1.4.2 RemoteDao.java

Dieses Interface spezifiziert die Mindestanfordernungen an ein DAO. Es enthält alle Operationen, die für einen Entity anwendbar sein müssen.

Listing 14: RemoteDao.java

```
1  package at.fh.ooe.swe4.campina.dao.api;
2
3  import java.rmi.Remote;
4  import java.rmi.RemoteException;
5  import java.util.List;
6
7  import at.fh.ooe.swe4.campina.persistence.api.AbstractEntity;
8
9  /**
10  * This interface marks an interface as an service which needs to provide at
11  * least these basic database operations.
12  *
13  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
14  * @date Jun 17, 2015
15  * @param <T>
16  *            the type of the entity
17  */
18 public interface RemoteDao<T extends AbstractEntity> extends Remote {
19
20   /**
21    * Saves or updates the entity
22    *
23    * @param entity
24    *            the entity to be saved or updated
25    * @return the saved or updated entity
26    * @throws RemoteException
27    *             if the remote invocation fails
28    */
29   public T save(T entity) throws RemoteException;
30
31   /**
32    * Deletes the entity
33    *
34    * @param entity
35    *            the entity to be saved or deleted
36    * @throws RemoteException
37    *             if the remote invocation fails
38    */
39   public void delete(T entity) throws RemoteException;
40
41   /**
42    * Gets the entity by its id
43    *
44    * @param id
45    *            the entity id
46    * @return the found entity
47    * @throws RemoteException
48    *             if the load fails
49    */
50   public T byId(Integer id) throws RemoteException;
51
52   /**
53    * Gets all entities.
54    *
55    * @return the list of entities
56    * @throws RemoteException
```

```
57      *             if the load fails
58      */
59     public List<T> getAll() throws RemoteException;
60
61 }
```

1.4.3 UserDao.java

Diese Interface spezifiziert die DAO Methoden für den Entitätentyp User.

Listing 15: UserDao.java

```
1  package at.fh.ooe.swe4.campina.dao.api;
2
3  import at.fh.ooe.swe4.campina.persistence.api.entity.User;
4
5  /**
6   * The DAO for the {@link User} entity type.
7   *
8   * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
9   * @date Jun 21, 2015
10  */
11 public interface UserDao extends RemoteDao<User> {
12 }
```

1.4.4 MenuDao.java

Diese Interface spezifiziert die DAO Methoden für den Entitätentyp Menu.

Listing 16: MenuDao.java

```
1  package at.fh.ooe.swe4.campina.dao.api;
2
3  import at.fh.ooe.swe4.campina.persistence.api.entity.Menu;
4
5  /**
6   * The DAO for the {@link Menu} entity type.
7   *
8   * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
9   * @date Jun 20, 2015
10  */
11 public interface MenuDao extends RemoteDao<Menu> {
12
13 }
```

1.4.5 MenuEntryDao.java

Diese Interface spezifiziert die DAO Methoden für den Entitätentyp MenuEntry.

Listing 17: MenuEntryDao.java

```java
package at.fh.ooe.swe4.campina.dao.api;

import java.rmi.RemoteException;
import java.util.List;

import at.fh.ooe.swe4.campina.persistence.api.entity.Menu;
import at.fh.ooe.swe4.campina.persistence.api.entity.MenuEntry;

/**
 * The DAO for the {@link MenuEntry} entity type.
 *
 * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
 * @date Jun 21, 2015
 */
public interface MenuEntryDao extends RemoteDao<MenuEntry> {

  /**
   * Gets all {@link MenuEntry} for given {@link Menu} id.
   *
   * @param id
   *            the menu id
   * @return the found menu entries
   * @throws RemoteException
   *             if the loading fails. Will contain a cause.
   */
  public List<MenuEntry> getAllForMenu(int id) throws RemoteException;

}
```

1.4.6 OrderDao.java

Diese Interface spezifiziert die DAO Methoden für den Entitätentyp Order.

Listing 18: OrderDao.java

```java
package at.fh.ooe.swe4.campina.dao.api;

import at.fh.ooe.swe4.campina.persistence.api.entity.Order;

/**
 * The DAO for the {@link Order} entity type.
 *
 * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
 * @date Jun 21, 2015
 */
public interface OrderDao extends RemoteDao<Order> {
}
```

1.4.7 EmailAlreadyUsedException.java

Diese Exception zeigt dem Client an, dass eine Email bereits verwendet wurde.

Listing 19: EmailAlreadyUsedException.java

```
1  package at.fh.ooe.swe4.campina.dao.exception;
2
3
4  /**
5   * This exception indicates than an email address is already used by another
6   * user.
7   *
8   * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
9   * @date Jun 21, 2015
10  */
11 public class EmailAlreadyUsedException extends RuntimeException {
12
13   private static final long  serialVersionUID  = -7433764767593804037L;
14
15   public EmailAlreadyUsedException() {
16     // TODO Auto-generated constructor stub
17   }
18
19   public EmailAlreadyUsedException(String s) {
20     super(s);
21     // TODO Auto-generated constructor stub
22   }
23
24   public EmailAlreadyUsedException(String s, Throwable cause) {
25     super(s, cause);
26     // TODO Auto-generated constructor stub
27   }
28
29 }
```

1.4.8 UsernameAlreadyUsedException.java

Diese Exception zeigt dem Client an, dass ein Benutzername bereits verwendet wurde.

Listing 20: UsernameAlreadyUsedException.java

```
1  package at.fh.ooe.swe4.campina.dao.exception;
2
3  /**
4   * This exception indicates that an username is already used by another user.
5   *
6   * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
7   * @date Jun 21, 2015
8   */
9  public class UsernameAlreadyUsedException extends RuntimeException {
10
11   private static final long  serialVersionUID  = 2301658696430408670L;
12
13   public UsernameAlreadyUsedException() {
14     // TODO Auto-generated constructor stub
15   }
16
17   public UsernameAlreadyUsedException(String s) {
18     super(s);
19     // TODO Auto-generated constructor stub
20   }
21
22   public UsernameAlreadyUsedException(String s, Throwable cause) {
23     super(s, cause);
24     // TODO Auto-generated constructor stub
25   }
26
27 }
```

1.5 Source swe-campina-dao-impl

Folgend ist der Source des Projekts *swe-campina-dao-impl* angeführt, welches die Implementierung der Spezifikation *swe-campina-dao-api* darstellt.

1.5.1 UserDaoImpl.java

Diese Klasse stellt die Implementierung der Spezifikation `UserDao` dar.

Listing 21: UserDaoImpl.java

```
1  package at.fh.ooe.swe4.campina.dao.impl;
2
3  import java.rmi.RemoteException;
4  import java.sql.Connection;
5  import java.sql.PreparedStatement;
6  import java.sql.SQLException;
7  import java.util.Collections;
8  import java.util.Comparator;
9  import java.util.List;
10
11 import at.fh.ooe.swe4.campina.dao.api.AbstractRemoteDao;
12 import at.fh.ooe.swe4.campina.dao.api.UserDao;
13 import at.fh.ooe.swe4.campina.dao.exception.EmailAlreadyUsedException;
14 import at.fh.ooe.swe4.campina.dao.exception.UsernameAlreadyUsedException;
15 import at.fh.ooe.swe4.campina.persistence.api.ConnectionManager;
16 import at.fh.ooe.swe4.campina.persistence.api.EntityManager;
17 import at.fh.ooe.swe4.campina.persistence.api.entity.User;
18 import at.fh.ooe.swe4.campina.persistence.impl.EntityManagerImpl;
19
20 /**
21  * This is the implementation of the {@link UserDao} specification.
22  *
23  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
24  * @date Jun 20, 2015
25  */
26 public class UserDaoImpl extends AbstractRemoteDao implements UserDao {
27
28   private static final long      serialVersionUID  = 8350405718897091714L;
29
30   private final EntityManager<User>  userEm        = new EntityManagerImpl<>(User.class);
31
32   private static final String      USER_ID_BY_EMAIL  = "SELECT id FROM campina.user WHERE email=?
     ↪  AND id<>?";
33   private static final String      USER_ID_BY_USERNAME  = "SELECT id FROM campina.user WHERE
     ↪  username=? AND id<>?";
34
35   /**
36    * @param connectionManager
37    * @throws RemoteException
38    */
39   public UserDaoImpl(ConnectionManager connectionManager) throws RemoteException {
40     super(connectionManager);
41   }
42
43   @Override
44   public User save(User user) throws RemoteException {
45     if (user == null) {
46       throw new RemoteException("Cannot save or update null entity", new NullPointerException());
47     }
48
49     Integer id = (user.getId() == null) ? -1 : user.getId();
50     try (Connection con = connectionManager.getConnection(Boolean.TRUE);) {
```

```
51        PreparedStatement pstmt;
52        // Email already used
53        pstmt = con.prepareStatement(USER_ID_BY_EMAIL);
54        pstmt.setString(1, user.getEmail());
55        pstmt.setInt(2, id);
56        if (pstmt.executeQuery()
57              .next()) {
58          throw new EmailAlreadyUsedException();
59        }
60        // Username already used
61        pstmt = con.prepareStatement(USER_ID_BY_USERNAME);
62        pstmt.setString(1, user.getUsername());
63        pstmt.setInt(2, id);
64        if (pstmt.executeQuery()
65              .next()) {
66          throw new UsernameAlreadyUsedException();
67        }
68
69        user = userEm.saveOrUpdate(con, user);
70        con.commit();
71        return user;
72      } catch (Throwable e) {
73        throw new RemoteException("Could not save user", e);
74      }
75    }
76
77    @Override
78    public void delete(User user) throws RemoteException {
79      if ((user == null) || (user.getId() == null)) {
80        throw new RemoteException("Cannot delete null entity or entity with null id", new
          ↪  NullPointerException());
81      }
82
83      try (Connection con = connectionManager.getConnection(Boolean.TRUE);) {
84        userEm.delete(con, user);
85        con.commit();
86      } catch (SQLException e) {
87        throw new RemoteException("Could not delete entity", e);
88      }
89    }
90
91    @Override
92    public User byId(final Integer id) throws RemoteException {
93      if (id == null) {
94        throw new RemoteException("Cannot find entity with null id", new NullPointerException());
95      }
96
97      try (Connection con = connectionManager.getConnection(Boolean.TRUE);) {
98        final User user = userEm.byId(con, id);
99        return user;
100     } catch (SQLException e) {
101       throw new RemoteException("Could not get user by id", e);
102     }
103   }
104
105   @Override
106   public List<User> getAll() throws RemoteException {
107     try (Connection con = connectionManager.getConnection(Boolean.TRUE);) {
108       final List<User> users = userEm.byType(con);
109       Collections.sort(users, new Comparator<User>() {
110         @Override
111         public int compare(User o1, User o2) {
112           return o1.getLastName()
```

```
113                 .compareTo(o2.getLastName());
114           }
115         });
116         return users;
117       } catch (SQLException e) {
118         throw new RemoteException("Could not get all users", e);
119       }
120     }
121 }
```

1.5.2 MenuDaoImpl.java

Diese Klasse stellt die Implementierung der Spezifikation `MenuDao` dar.

Listing 22: MenuDaoImpl.java

```
1  package at.fh.ooe.swe4.campina.dao.impl;
2
3  import java.rmi.RemoteException;
4  import java.sql.Connection;
5  import java.sql.SQLException;
6  import java.util.Collections;
7  import java.util.Comparator;
8  import java.util.List;
9
10 import at.fh.ooe.swe4.campina.dao.api.AbstractRemoteDao;
11 import at.fh.ooe.swe4.campina.dao.api.MenuDao;
12 import at.fh.ooe.swe4.campina.persistence.api.ConnectionManager;
13 import at.fh.ooe.swe4.campina.persistence.api.EntityManager;
14 import at.fh.ooe.swe4.campina.persistence.api.entity.Menu;
15 import at.fh.ooe.swe4.campina.persistence.impl.EntityManagerImpl;
16
17 /**
18  * This is the implementation of the {@link MenuDao} specification.
19  *
20  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
21  * @date Jun 20, 2015
22  */
23 public class MenuDaoImpl extends AbstractRemoteDao implements MenuDao {
24
25   private static final long  serialVersionUID  = 2017517222109172291L;
26
27   final EntityManager<Menu>  menuEm        = new EntityManagerImpl<>(Menu.class);
28
29   /**
30    * @param connectionManager
31    * @throws RemoteException
32    */
33   public MenuDaoImpl(ConnectionManager connectionManager) throws RemoteException {
34     super(connectionManager);
35   }
36
37   @Override
38   public Menu save(Menu menu) throws RemoteException {
39     if (menu == null) {
40       throw new RemoteException("Cannot save or update null entity", new NullPointerException());
41     }
42
43     try (final Connection con = connectionManager.getConnection(Boolean.TRUE)) {
44       menu = menuEm.saveOrUpdate(con, menu);
45       con.commit();
46       return menu;
47     } catch (Throwable e) {
48       throw new RemoteException("Could not save or update menu", e);
49     }
50   }
51
52   @Override
53   public void delete(Menu menu) throws RemoteException {
54     if (menu == null) {
55       throw new RemoteException("Cannot save or update null entity", new NullPointerException());
56     }
57
58     if ((menu == null) || (menu.getId() == null)) {
```

```
59        throw new RemoteException("Cannot delete null entity or entity with null id", new
          ↪  NullPointerException());
60      }
61
62      try (Connection con = connectionManager.getConnection(Boolean.TRUE);) {
63        menuEm.delete(con, menu);
64        con.commit();
65      } catch (SQLException e) {
66        throw new RemoteException("Could not delete menu", e);
67      }
68    }
69
70    @Override
71    public Menu byId(Integer id) throws RemoteException {
72      if (id == null) {
73        throw new RemoteException("Cannot fetch entity with null id", new NullPointerException());
74      }
75
76      try (Connection con = connectionManager.getConnection(Boolean.TRUE);) {
77        final Menu menu = menuEm.byId(con, id);
78        return menu;
79      } catch (SQLException e) {
80        throw new RemoteException("Could not get menu by id", e);
81      }
82    }
83
84    @Override
85    public List<Menu> getAll() throws RemoteException {
86      try (Connection con = connectionManager.getConnection(Boolean.TRUE);) {
87        final List<Menu> menus = menuEm.byType(con);
88        Collections.sort(menus, new Comparator<Menu>() {
89          @Override
90          public int compare(Menu o1, Menu o2) {
91            return o1.getLabel()
92                  .compareTo(o2.getLabel());
93          }
94        });
95        return menus;
96      } catch (SQLException e) {
97        throw new RemoteException("Could not get all menus", e);
98      }
99    }
100
101 }
```

1.5.3 MenuEntryDaoImpl.java

Diese Klasse stellt die Implementierung der Spezifikation `MenuEntryDao` dar.

Listing 23: MenuEntryDaoImpl.java

```
1  package at.fh.ooe.swe4.campina.dao.impl;
2
3  import java.rmi.RemoteException;
4  import java.sql.Connection;
5  import java.sql.SQLException;
6  import java.util.Collections;
7  import java.util.Comparator;
8  import java.util.List;
9
10 import at.fh.ooe.swe4.campina.dao.api.AbstractRemoteDao;
11 import at.fh.ooe.swe4.campina.dao.api.MenuEntryDao;
12 import at.fh.ooe.swe4.campina.persistence.api.ConnectionManager;
13 import at.fh.ooe.swe4.campina.persistence.api.EntityManager;
14 import at.fh.ooe.swe4.campina.persistence.api.entity.MenuEntry;
15 import at.fh.ooe.swe4.campina.persistence.impl.EntityManagerImpl;
16
17 /**
18  * This is the implementation of the {@link MenuEntryDao} specification.
19  *
20  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
21  * @date Jun 20, 2015
22  */
23 public class MenuEntryDaoImpl extends AbstractRemoteDao implements MenuEntryDao {
24
25   private static final long        serialVersionUID  = 2017517222109172291L;
26
27   private final EntityManager<MenuEntry>  menuEntryEm      = new
     ↪  EntityManagerImpl<>(MenuEntry.class);
28   private static final String        SELECT_FOR_MENU    = "SELECT id, %s, version FROM %s WHERE
     ↪  menu_id=?";
29
30   /**
31    * @param connectionManager
32    * @throws RemoteException
33    */
34   public MenuEntryDaoImpl(ConnectionManager connectionManager) throws RemoteException {
35     super(connectionManager);
36   }
37
38   @Override
39   public MenuEntry save(MenuEntry menuEntry) throws RemoteException {
40     if (menuEntry == null) {
41       throw new RemoteException("Cannot save or update null entity", new NullPointerException());
42     }
43
44     try (final Connection con = connectionManager.getConnection(Boolean.TRUE)) {
45       menuEntry = menuEntryEm.saveOrUpdate(con, menuEntry);
46       con.commit();
47       return menuEntry;
48     } catch (Throwable e) {
49       throw new RemoteException("Could not save or update menu", e);
50     }
51   }
52
53   @Override
54   public void delete(MenuEntry menuEntry) throws RemoteException {
55     if (menuEntry == null) {
56       throw new RemoteException("Cannot save or update null entity", new NullPointerException());
```

```
57      }
58
59      if ((menuEntry == null) || (menuEntry.getId() == null)) {
60        throw new RemoteException("Cannot delete null entity or entity with null id", new
          ↪  NullPointerException());
61      }
62
63      try (Connection con = connectionManager.getConnection(Boolean.TRUE);) {
64        menuEntryEm.delete(con, menuEntry);
65        con.commit();
66      } catch (SQLException e) {
67        throw new RemoteException("Could not delete entity", e);
68      }
69    }
70
71    @Override
72    public MenuEntry byId(Integer id) throws RemoteException {
73      if (id == null) {
74        throw new RemoteException("Cannot fetch entity with null id", new NullPointerException());
75      }
76
77      try (Connection con = connectionManager.getConnection(Boolean.TRUE);) {
78        final MenuEntry menuEntry = menuEntryEm.byId(con, id);
79        return menuEntry;
80      } catch (SQLException e) {
81        throw new RemoteException("Could not get menu entry by id", e);
82      }
83    }
84
85    @Override
86    public List<MenuEntry> getAll() throws RemoteException {
87      try (Connection con = connectionManager.getConnection(Boolean.TRUE);) {
88        final List<MenuEntry> users = menuEntryEm.byType(con);
89        Collections.sort(users, new Comparator<MenuEntry>() {
90          @Override
91          public int compare(MenuEntry o1, MenuEntry o2) {
92            return o1.getOrdinal()
93                  .compareTo(o2.getOrdinal());
94          }
95        });
96        return users;
97      } catch (SQLException e) {
98        throw new RemoteException("Could not get all menu entries", e);
99      }
100   }
101
102   @Override
103   public List<MenuEntry> getAllForMenu(final int id) throws RemoteException {
104     try (Connection con = connectionManager.getConnection(Boolean.TRUE);) {
105       final List<MenuEntry> menuEntries = menuEntryEm.byQuery(con,
106                                   String.format(SELECT_FOR_MENU, menuEntryEm.getColumnNames(null),
                                      ↪  menuEntryEm.getTableName()),
107                                   Integer.valueOf(id));
108       Collections.sort(menuEntries);
109       return menuEntries;
110     } catch (SQLException e) {
111       throw new RemoteException("Could not get all menu entries for menu", e);
112     }
113   }
114 }
```

1.5.4 OrderDaoImpl.java

Diese Klasse stellt die Implementierung der Spezifikation `OrderDao` dar.

Listing 24: OrderDaoImpl.java

```
1  package at.fh.ooe.swe4.campina.dao.impl;
2
3  import java.rmi.RemoteException;
4  import java.sql.Connection;
5  import java.sql.PreparedStatement;
6  import java.sql.ResultSet;
7  import java.util.ArrayList;
8  import java.util.List;
9
10 import at.fh.ooe.swe4.campina.dao.api.AbstractRemoteDao;
11 import at.fh.ooe.swe4.campina.dao.api.OrderDao;
12 import at.fh.ooe.swe4.campina.persistence.api.ConnectionManager;
13 import at.fh.ooe.swe4.campina.persistence.api.EntityManager;
14 import at.fh.ooe.swe4.campina.persistence.api.entity.Menu;
15 import at.fh.ooe.swe4.campina.persistence.api.entity.MenuEntry;
16 import at.fh.ooe.swe4.campina.persistence.api.entity.Order;
17 import at.fh.ooe.swe4.campina.persistence.api.entity.User;
18 import at.fh.ooe.swe4.campina.persistence.impl.EntityManagerImpl;
19
20 /**
21  * This is the implementation of the {@link OrderDao} specification.
22  *
23  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
24  * @date Jun 21, 2015
25  */
26 public class OrderDaoImpl extends AbstractRemoteDao implements OrderDao {
27
28   private final EntityManager<Order>    orderEm;
29   private final EntityManager<User>     userEm;
30   private final EntityManager<MenuEntry>  menuEntryEm;
31   private final EntityManager<Menu>     menuEm;
32
33   private static final String         SELECT_QUERY    = "SELECT %s FROM campina.order o "
34                                     + "INNER JOIN campina.user u ON u.id = o.user_id "
35                                     + "INNER JOIN campina.menu_entry me ON me.id = o.menu_entry_id
                                        ↪   "
36                                     + "INNER JOIN campina.menu m ON m.id = me.menu_id "
37                                     + "ORDER BY order_date DESC ";
38   private static final long         serialVersionUID  = -247051344150973512L;
39
40   /**
41    * @param connectionManager
42    * @throws RemoteException
43    */
44   public OrderDaoImpl(ConnectionManager connectionManager) throws RemoteException {
45     super(connectionManager);
46     this.orderEm = new EntityManagerImpl<>(Order.class);
47     this.userEm = new EntityManagerImpl<>(User.class);
48     this.menuEm = new EntityManagerImpl<>(Menu.class);
49     this.menuEntryEm = new EntityManagerImpl<>(MenuEntry.class);
50   }
51
52   @Override
53   public Order save(Order order) throws RemoteException {
54     if (order == null) {
55       throw new RemoteException("Cannot save or update null entity", new NullPointerException());
56     }
57
```

```
58      try (final Connection con = connectionManager.getConnection(Boolean.TRUE)) {
59        order = orderEm.saveOrUpdate(con, order);
60        con.commit();
61        return order;
62      } catch (Throwable e) {
63        throw new RemoteException("Could not save or update menu", e);
64      }
65    }
66
67    @Override
68    public void delete(Order entity) throws RemoteException {
69      throw new UnsupportedOperationException();
70    }
71
72    @Override
73    public Order byId(Integer id) throws RemoteException {
74      throw new UnsupportedOperationException();
75    }
76
77    @Override
78    public List<Order> getAll() throws RemoteException {
79      String names = "o.id, " + orderEm.getColumnNames("o") + ", o.version, ";
80      names += "me.id, " + menuEntryEm.getColumnNames("me") + ", me.version, ";
81      names += "m.id, " + menuEm.getColumnNames("m") + ", m.version, ";
82      names += "u.id, " + userEm.getColumnNames("u") + ", u.version";
83
84      System.out.println(String.format(SELECT_QUERY, names));
85      final List<Order> orders = new ArrayList<>();
86      try (final Connection con = connectionManager.getConnection(Boolean.FALSE)) {
87        try (final PreparedStatement pstmt = con.prepareStatement(String.format(SELECT_QUERY,
          ↪  names))) {
88          final ResultSet result = pstmt.executeQuery();
89
90          while (result.next()) {
91            final Order order = new Order();
92            final MenuEntry menuEntry = new MenuEntry();
93            final Menu menu = new Menu();
94            final User user = new User();
95            int colIdx = 1;
96
97            order.setId(result.getInt(colIdx));
98            orderEm.fillEntity(result, order, colIdx);
99            order.setVersion(result.getLong((colIdx = orderEm.getColumnMeta()
100                                 .size() + 2)));
101
102           menuEntry.setId(result.getInt((colIdx += 1)));
103           menuEntryEm.fillEntity(result, menuEntry, colIdx);
104           menuEntry.setVersion(result.getLong((colIdx = colIdx + menuEntryEm.getColumnMeta()
105                                       .size() + 1)));
106
107           menu.setId(result.getInt((colIdx += 1)));
108           menuEm.fillEntity(result, menu, colIdx);
109           menu.setVersion(result.getLong((colIdx = colIdx + menuEm.getColumnMeta()
110                                   .size() + 1)));
111
112           user.setId(result.getInt((colIdx += 1)));
113           userEm.fillEntity(result, user, colIdx);
114           user.setVersion(result.getLong((colIdx = colIdx + userEm.getColumnMeta()
115                                   .size() + 1)));
116
117           menuEntry.setMenu(menu);
118           order.setUser(user);
119           order.setMenuEntry(menuEntry);
```

```
120
121             orders.add(order);
122           }
123         }
124       } catch (Throwable e) {
125         throw new RemoteException("Could not load all orders", e);
126       }
127
128       return orders;
129     }
130   }
```

1.5.5 AbstractDaoTest.java

Diese abstrakte Klasse stellt die Wurzelklasse aller DAO Test Implementierungen dar. Sie stellt Methoden für das Anlegen einer Datenbank zu Verfügung, was in den folgenden Tests vor und nachdem Test passiert.

Listing 25: AbstractDaoTest.java

```
1  package at.fh.ooe.swe4.campina.test.dao.api;
2
3  import java.nio.charset.Charset;
4  import java.nio.file.Files;
5  import java.nio.file.Paths;
6  import java.sql.Connection;
7  import java.sql.PreparedStatement;
8  import java.sql.SQLException;
9  import java.util.ArrayList;
10 import java.util.List;
11 import java.util.Objects;
12
13 import org.apache.log4j.Level;
14 import org.apache.log4j.Logger;
15 import org.junit.ClassRule;
16 import org.junit.Rule;
17 import org.junit.rules.ExpectedException;
18 import org.junit.runner.RunWith;
19 import org.junit.runners.JUnit4;
20
21 import at.fh.ooe.swe4.campina.persistence.api.AbstractEntity;
22 import at.fh.ooe.swe4.campina.persistence.api.ConnectionManager;
23 import at.fh.ooe.swe4.campina.persistence.impl.ConnectionManagerImpl;
24 import at.fh.ooe.swe4.campina.persistence.impl.ConnectionManagerImpl.DbMetadata;
25 import at.fh.ooe.swe4.campina.persistence.impl.DbConfigParam;
26 import at.fh.ooe.swe4.campina.persistence.impl.EntityManagerImpl;
27 import at.fh.ooe.swe4.junit.test.suite.watcher.LoggingTestClassWatcher;
28 import at.fh.ooe.swe4.junit.test.suite.watcher.LoggingTestInvocationWatcher;
29
30 /**
31  * This is the base class for all dao test.<br>
32  * It provides the connection manager for the underlying test database.<br>
33  * It provides utility methods for saving all types of entities and an
34  * entitymanager for the tested entity dao.
35  *
36  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
37  * @date Jun 21, 2015
38  * @param <T>
39  */
40 @RunWith(JUnit4.class)
41 public abstract class AbstractDaoTest<T extends AbstractEntity> {
42
43   protected final DbMetadata          dbMeta;
44   protected final ConnectionManager      conManager;
45   protected final EntityManagerImpl<T>    em;
46
47   private final Logger            log;
48   @ClassRule
49   public static final LoggingTestClassWatcher  testClassWatcher    = new
     ↪  LoggingTestClassWatcher(Level.INFO);
50   @Rule
51   public final LoggingTestInvocationWatcher  testWatcher        = new
     ↪  LoggingTestInvocationWatcher(Level.INFO);
52   @Rule
53   public RemoteExceptionLogger        remoteExceptionLogger  = new RemoteExceptionLogger();
```

```
54  @Rule
55  public ExpectedException          expectedException    = ExpectedException.none();
56
57  /**
58   * @param entityClazz
59   */
60  public AbstractDaoTest(final Class<T> entityClazz) {
61    super();
62    log = Logger.getLogger(this.getClass());
63    log.info("----------------------------------------");
64    log.info("Initialization started");
65    log.info("----------------------------------------");
66    dbMeta = new DbMetadata(DbConfigParam.DRIVER.val(),
67        DbConfigParam.URL.val(),
68        DbConfigParam.USER.val(),
69        DbConfigParam.PASSWORD.val(),
70        2);
71    this.conManager = new ConnectionManagerImpl(dbMeta);
72    this.em = new EntityManagerImpl<>(entityClazz);
73    setupDB();
74    log.info("----------------------------------------");
75    log.info("Initialization finished");
76    log.info("----------------------------------------");
77  }
78
79  /**
80   * Drops and recreates the test database. Should be called before each test
81   * method invocation.
82   */
83  protected void setupDB() {
84    log.info("----------------------------------------");
85    log.info("Initialization database started");
86    log.info("ddl: db-config.xml");
87    log.info("----------------------------------------");
88    try {
89      final List<String> lines = Files.readAllLines(Paths.get(this.getClass()
90                                    .getClassLoader()
91                                    .getResource("create-campina-schema.sql")
92                                    .toURI()), Charset.forName("UTF-8"));
93
94      final List<String> statements = new ArrayList<>(lines.size());
95      String statement = "";
96      for (String string : lines) {
97        statement += " " + string;
98        if (string.endsWith(";")) {
99          statements.add(statement);
100         statement = "";
101       }
102     }
103     try (final Connection con = conManager.getConnection(Boolean.FALSE)) {
104       try (final PreparedStatement pstmt = con.prepareStatement("DROP SCHEMA IF EXISTS
          ↪  campina")) {
105         pstmt.executeUpdate();
106       }
107       for (String string : statements) {
108         log.info("Executing ddl: " + string);
109         try (final PreparedStatement pstmt = con.prepareStatement(string)) {
110           pstmt.executeUpdate();
111         }
112       }
113     } catch (SQLException e) {
114       throw new IllegalStateException("Could not setup db", e);
115     }
```

```
116       } catch (Throwable e) {
117         log.error("Could not setup database", e);
118         throw new IllegalStateException("ddl file load failed", e);
119       }
120       log.info("-----------------------------------------");
121       log.info("Initialization database finished");
122       log.info("-----------------------------------------");
123     }
124
125     /**
126      * Saves the collection of entities.
127      *
128      * @param entities
129      *            the entities to be saved
130      * @return the saved entities
131      */
132     protected <E extends AbstractEntity> List<E> saveEntities(final List<E> entities) {
133       Objects.requireNonNull(entities);
134       final List<E> saved = new ArrayList<>(entities.size());
135
136       for (E entity : entities) {
137         saved.add(saveEntity(entity));
138       }
139
140       return saved;
141     }
142
143     /**
144      * Saves the given entity.
145      *
146      * @param entity
147      *            the entity to be saved.
148      * @return the saved entity
149      */
150     @SuppressWarnings({
151               "rawtypes", "unchecked" })
152     protected <E extends AbstractEntity> E saveEntity(final E entity) {
153       Objects.requireNonNull(entity);
154
155       E entityDB;
156       try (final Connection con = conManager.getConnection(Boolean.FALSE)) {
157         entityDB = (E) (new EntityManagerImpl(entity.getClass())).saveOrUpdate(con, entity);
158       } catch (SQLException e) {
159         throw new IllegalStateException("Entity save failed", e);
160       }
161
162       return entityDB;
163     }
164 }
```

1.5.6 RemoteDetailMatcher.java

Diese Klasse ist eine Hilfsklasse, die über eine JUnit Rule auf Excptions reagiert, damit Causes getestet werden können, da die DAOs immer `RemoteException` werfen.

Listing 26: RemoteDetailMatcher.java

```java
package at.fh.ooe.swe4.campina.test.dao.api;

import java.rmi.RemoteException;
import java.util.Objects;

import org.hamcrest.BaseMatcher;
import org.hamcrest.Description;

public class RemoteDetailMatcher extends BaseMatcher<Throwable> {

  private final Class<? extends Throwable>  clazz;

  /**
   * @param clazz
   */
  public RemoteDetailMatcher(Class<? extends Throwable> clazz) {
    super();
    Objects.requireNonNull(clazz);

    this.clazz = clazz;
  }

  @Override
  public boolean matches(Object item) {
    return ((item != null)
        && ((((RemoteException) item).detail) != null))
        && (clazz.isInstance(((RemoteException) item).detail));
  }

  @Override
  public void describeTo(Description description) {
    description.appendText("Detail of RemoteException is not equal to '" + clazz.getName() + "'");

  }
}
```

1.5.7 RemoteExceptionLogger.java

Diese Klasse ist eine Hilfsklasse, die über eine JUnit Rule auf Excptions reagiert, damit Causes geloggt werden können, da die DAOs immer `RemoteException` werfen, wobei dessen Member `remoteExc.detail` die eigentliche Cause enthält.

Listing 27: RemoteExceptionLogger.java

```
1  package at.fh.ooe.swe4.campina.test.dao.api;
2
3  import java.rmi.RemoteException;
4
5  import org.apache.log4j.Logger;
6  import org.junit.rules.TestWatcher;
7  import org.junit.runner.Description;
8
9  /**
10  * This class logs the detail of the thrown remoteException which the client
11  * would also do.
12  *
13  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
14  * @date Jun 21, 2015
15  */
16 public class RemoteExceptionLogger extends TestWatcher {
17
18   private static final Logger  log  = Logger.getLogger(RemoteExceptionLogger.class);
19
20   public RemoteExceptionLogger() {
21   }
22
23   @Override
24   protected void failed(Throwable e, Description description) {
25     super.failed(e, description);
26     if (e != null) {
27       if (e instanceof RemoteException) {
28         final RemoteException re = (RemoteException) e;
29         if (re.detail != null) {
30           log.error("RemoteException with detail: ", e);
31         }
32       }
33     }
34   }
35 }
```

1.5.8 UserDaoTest.java

Diese Testklasse testet das DAO `UserDao`

Listing 28: UserDaoTest.java

```
1  package at.fh.ooe.swe4.campina.test.dao.impl;
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.assertTrue;
5  import static org.junit.Assert.fail;
6
7  import java.rmi.RemoteException;
8  import java.sql.Connection;
9  import java.sql.SQLException;
10 import java.util.ArrayList;
11 import java.util.List;
12
13 import org.junit.Before;
14 import org.junit.Test;
15
16 import at.fh.ooe.swe4.campina.dao.api.UserDao;
17 import at.fh.ooe.swe4.campina.dao.exception.EmailAlreadyUsedException;
18 import at.fh.ooe.swe4.campina.dao.exception.UsernameAlreadyUsedException;
19 import at.fh.ooe.swe4.campina.dao.impl.UserDaoImpl;
20 import at.fh.ooe.swe4.campina.persistence.api.entity.User;
21 import at.fh.ooe.swe4.campina.test.dao.api.AbstractDaoTest;
22 import at.fh.ooe.swe4.campina.test.dao.api.RemoteDetailMatcher;
23
24 /**
25  * This test class represents the test class for the {@link UserDao}
26  * implementation.
27  *
28  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
29  * @date Jun 20, 2015
30  */
31 public class UserDaoTest extends AbstractDaoTest<User> {
32
33   private final UserDao  dao;
34
35   public UserDaoTest() throws RemoteException {
36     super(User.class);
37     this.dao = new UserDaoImpl(conManager);
38   }
39
40   @Before
41   public void beforeTest() {
42     setupDB();
43   }
44
45   @Test
46   public void saveNull() throws RemoteException {
47     // -- Given --
48     final User user = null;
49     expectedException.expect(new RemoteDetailMatcher(NullPointerException.class));
50
51     // -- When | Then --
52     dao.save(user);
53   }
54
55   @Test
56   public void save() throws RemoteException {
57     // -- Given --
58     User user = new User();
```

```
59      user.setFirstName("Thomas");
60      user.setLastName("Herzog");
61      user.setUsername("cchet");
62      user.setEmail("t.t@t.at");
63      user.setPassword("xxxxxxx");
64      user.setAdminFlag(Boolean.TRUE);
65      user.setBlockedFlag(Boolean.FALSE);
66
67      // -- When --
68      user = dao.save(user);
69
70      // -- Then --
71      try (final Connection con = conManager.getConnection(Boolean.TRUE);) {
72        final User userDB = em.byId(con, user.getId());
73        assertEquals(user, userDB);
74      } catch (SQLException e) {
75        fail("Could not obtain connection");
76      }
77    }
78
79    @Test
80    public void saveDuplicateEmail() throws RemoteException {
81      // -- Given --
82      User user = new User();
83      user.setFirstName("Thomas");
84      user.setLastName("Herzog");
85      user.setUsername("cchet");
86      user.setEmail("t.t@t.at");
87      user.setPassword("xxxxxxx");
88      user.setAdminFlag(Boolean.TRUE);
89      user.setBlockedFlag(Boolean.FALSE);
90      user = saveEntity(user);
91      user.setId(null);
92      expectedException.expect(new RemoteDetailMatcher(EmailAlreadyUsedException.class));
93
94      // -- When --
95      user = dao.save(user);
96    }
97
98    @Test
99    public void saveDuplicateUsername() throws RemoteException {
100     // -- Given --
101     User user = new User();
102     user.setFirstName("Thomas");
103     user.setLastName("Herzog");
104     user.setUsername("cchet");
105     user.setEmail("t.t@t.at");
106     user.setPassword("xxxxxxx");
107     user.setAdminFlag(Boolean.TRUE);
108     user.setBlockedFlag(Boolean.FALSE);
109     user = saveEntity(user);
110     user.setId(null);
111     user.setEmail(user.getEmail() + ".com");
112     expectedException.expect(new RemoteDetailMatcher(UsernameAlreadyUsedException.class));
113
114     // -- When | Then --
115     user = dao.save(user);
116   }
117
118   @Test
119   public void getAllEmpty() throws RemoteException {
120     // -- Given | When --
121     final List<User> fetchedUsers = dao.getAll();
```

```
122
123     // -- Then --
124     assertTrue(fetchedUsers.isEmpty());
125   }
126
127   @Test
128   public void getAll() throws RemoteException {
129     // -- Given --
130     List<User> users = new ArrayList<>(5);
131     for (int i = 0; i < 5; i++) {
132       User user = new User();
133       user.setFirstName("Thomas-" + i);
134       user.setLastName("Herzog-" + i);
135       user.setUsername("cchet-" + i);
136       user.setEmail("t-" + i + ".t@t.at");
137       user.setPassword("xxxxxxx");
138       user.setAdminFlag(Boolean.TRUE);
139       user.setBlockedFlag(Boolean.FALSE);
140       users.add(user);
141     }
142     users = saveEntities(users);
143
144     // -- When --
145     final List<User> fetchedUsers = dao.getAll();
146
147     // -- Then --
148     assertEquals(users.size(), fetchedUsers.size());
149     assertEquals(users, fetchedUsers);
150   }
151
152   @Test
153   public void byIdNull() throws RemoteException {
154     // -- Given --
155     final Integer id = null;
156     expectedException.expect(new RemoteDetailMatcher(NullPointerException.class));
157
158     // -- When | Then --
159     dao.byId(id);
160   }
161
162   @Test
163   public void byId() throws RemoteException {
164     // -- Given --
165     User user = new User();
166     user.setFirstName("Thomas");
167     user.setLastName("Herzog");
168     user.setUsername("cchet");
169     user.setEmail("t.t@t.at");
170     user.setPassword("xxxxxxx");
171     user.setAdminFlag(Boolean.TRUE);
172     user.setBlockedFlag(Boolean.FALSE);
173     user = saveEntity(user);
174
175     // -- When --
176     final User userDB = dao.byId(user.getId());
177
178     // -- Then --
179     assertEquals(user, userDB);
180   }
181 }
```

1.5.9 MenuDaoTest.java

Diese Testklasse testet das DAO `MenuDao`

Listing 29: MenuDaoTest.java

```
1  package at.fh.ooe.swe4.campina.test.dao.impl;
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.assertTrue;
5  import static org.junit.Assert.fail;
6
7  import java.rmi.RemoteException;
8  import java.sql.Connection;
9  import java.sql.SQLException;
10 import java.util.ArrayList;
11 import java.util.List;
12
13 import org.junit.Before;
14 import org.junit.Test;
15
16 import at.fh.ooe.swe4.campina.dao.api.MenuDao;
17 import at.fh.ooe.swe4.campina.dao.impl.MenuDaoImpl;
18 import at.fh.ooe.swe4.campina.persistence.api.entity.Menu;
19 import at.fh.ooe.swe4.campina.persistence.api.entity.constants.Day;
20 import at.fh.ooe.swe4.campina.test.dao.api.AbstractDaoTest;
21 import at.fh.ooe.swe4.campina.test.dao.api.RemoteDetailMatcher;
22
23 /**
24  * This test class represents the test class for the {@link MenuDao}
25  * implementation.
26  *
27  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
28  * @date Jun 20, 2015
29  */
30 public class MenuDaoTest extends AbstractDaoTest<Menu> {
31
32   private final MenuDao  dao;
33
34   public MenuDaoTest() throws RemoteException {
35     super(Menu.class);
36     this.dao = new MenuDaoImpl(conManager);
37   }
38
39   @Before
40   public void beforeTest() {
41     setupDB();
42   }
43
44   // -- Then --
45   @Test
46   public void saveNull() throws RemoteException {
47     // -- Given --
48     final Menu menu = null;
49     expectedException.expect(new RemoteDetailMatcher(NullPointerException.class));
50
51     // -- When --
52     dao.save(menu);
53   }
54
55   @Test
56   public void save() throws RemoteException {
57     // -- Given --
58     Menu menu = new Menu();
```

```
59      menu.setDay(Day.MONDAY);
60      menu.setLabel("menu-1");
61
62      // -- When --
63      dao.save(menu);
64
65      // -- Then --
66      try (final Connection con = conManager.getConnection(Boolean.TRUE);) {
67        final Menu menuDB = em.byId(con, menu.getId());
68        assertEquals(menu, menuDB);
69      } catch (SQLException e) {
70        fail("Could not obtain connection");
71      }
72    }
73
74    @Test
75    public void getAllEmpty() throws RemoteException {
76      // -- Given | When --
77      final List<Menu> fetchedMenus = dao.getAll();
78
79      // -- Then --
80      assertTrue(fetchedMenus.isEmpty());
81    }
82
83    @Test
84    public void getAll() throws RemoteException {
85      // -- Given --
86      List<Menu> menus = new ArrayList<>(5);
87      for (int i = 0; i < 5; i++) {
88        Menu menu = new Menu();
89        menu.setDay(Day.MONDAY);
90        menu.setLabel("menu-1-" + i);
91        menus.add(menu);
92      }
93      menus = saveEntities(menus);
94
95      // -- When --
96      final List<Menu> fetchedMenus = dao.getAll();
97
98      // -- Then --
99      assertEquals(menus.size(), fetchedMenus.size());
100     assertEquals(menus, fetchedMenus);
101   }
102
103   @Test
104   public void byIdNull() throws RemoteException {
105     // -- Given --
106     final Integer id = null;
107     expectedException.expect(new RemoteDetailMatcher(NullPointerException.class));
108
109     // -- When | Then --
110     dao.byId(id);
111   }
112
113   @Test
114   public void byId() throws RemoteException {
115     // -- Given --
116     Menu menu = new Menu();
117     menu.setDay(Day.MONDAY);
118     menu.setLabel("menu-1");
119     menu = saveEntity(menu);
120
121     // -- When --
```

```
122     final Menu menuDB = dao.byId(menu.getId());
123
124     // -- Then --
125     assertEquals(menu, menuDB);
126   }
127 }
```

1.5.10 MenuEntryDaoTest.java

Diese Testklasse testet das DAO `MenuEntryDao`

Listing 30: MenuEntryDaoTest.java

```
1  package at.fh.ooe.swe4.campina.test.dao.impl;
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.assertTrue;
5  import static org.junit.Assert.fail;
6
7  import java.math.BigDecimal;
8  import java.rmi.RemoteException;
9  import java.sql.Connection;
10 import java.sql.SQLException;
11 import java.util.ArrayList;
12 import java.util.List;
13
14 import org.junit.Before;
15 import org.junit.Test;
16
17 import at.fh.ooe.swe4.campina.dao.api.MenuEntryDao;
18 import at.fh.ooe.swe4.campina.dao.impl.MenuEntryDaoImpl;
19 import at.fh.ooe.swe4.campina.persistence.api.entity.Menu;
20 import at.fh.ooe.swe4.campina.persistence.api.entity.MenuEntry;
21 import at.fh.ooe.swe4.campina.persistence.api.entity.constants.Day;
22 import at.fh.ooe.swe4.campina.test.dao.api.AbstractDaoTest;
23 import at.fh.ooe.swe4.campina.test.dao.api.RemoteDetailMatcher;
24
25 /**
26  * This test class represents the test class for the {@link MenuEntryDao}
27  * implementation.
28  *
29  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
30  * @date Jun 20, 2015
31  */
32 public class MenuEntryDaoTest extends AbstractDaoTest<MenuEntry> {
33
34   private final MenuEntryDao  dao;
35   private Menu        menu;
36
37   public MenuEntryDaoTest() throws RemoteException {
38     super(MenuEntry.class);
39     this.dao = new MenuEntryDaoImpl(conManager);
40   }
41
42   @Before
43   public void beforeTest() {
44     setupDB();
45     this.menu = new Menu();
46     menu.setDay(Day.MONDAY);
47     menu.setLabel("menu-1");
48     menu = saveEntity(menu);
49   }
50
51   @Test
52   public void saveNull() throws RemoteException {
53     // -- Given --
54     final MenuEntry menuEntry = null;
55     expectedException.expect(new RemoteDetailMatcher(NullPointerException.class));
56
57     // -- When | Then --
58     dao.save(menuEntry);
```

```
59    }
60
61    @Test
62    public void save() throws RemoteException {
63      // -- Given --
64      MenuEntry menuEntry = new MenuEntry();
65      menuEntry.setLabel("menu-1");
66      menuEntry.setMenu(menu);
67      menuEntry.setOrdinal(0);
68      menuEntry.setPrice(BigDecimal.ONE);
69      menuEntry = saveEntity(menuEntry);
70
71      // -- When --
72      dao.save(menuEntry);
73
74      // -- Then --
75      try (final Connection con = conManager.getConnection(Boolean.TRUE);) {
76        final MenuEntry menuEntryDB = em.byId(con, menuEntry.getId());
77        assertEquals(menuEntry, menuEntryDB);
78      } catch (SQLException e) {
79        fail("Could not obtain connection");
80      }
81    }
82
83    @Test
84    public void getAllEmpty() throws RemoteException {
85      // -- Given | When --
86      final List<MenuEntry> fetchedMenus = dao.getAll();
87
88      // -- Then --
89      assertTrue(fetchedMenus.isEmpty());
90    }
91
92    @Test
93    public void getAll() throws RemoteException {
94      // -- Given --
95      List<MenuEntry> menuEntries = new ArrayList<>(5);
96      for (int i = 0; i < 5; i++) {
97        MenuEntry menuEntry = new MenuEntry();
98        menuEntry.setLabel("menu-entry-1-" + i);
99        menuEntry.setMenu(menu);
100       menuEntry.setOrdinal(i);
101       menuEntry.setPrice(BigDecimal.ONE);
102       menuEntries.add(menuEntry);
103     }
104     menuEntries = saveEntities(menuEntries);
105
106     // -- When --
107     final List<MenuEntry> fetchedMenuEntries = dao.getAll();
108
109     // -- Then --
110     assertEquals(menuEntries.size(), fetchedMenuEntries.size());
111     assertEquals(menuEntries, fetchedMenuEntries);
112   }
113
114   @Test
115   public void getAllForMenuEmpty() throws RemoteException {
116     // -- Given | When --
117     final List<MenuEntry> fetchedMenus = dao.getAllForMenu(menu.getId());
118
119     // -- Then --
120     assertTrue(fetchedMenus.isEmpty());
121   }
```

```
122
123   @Test
124   public void getAllForMenu() throws RemoteException {
125     // -- Given --
126     List<MenuEntry> menuEntries = new ArrayList<>(5);
127     for (int i = 0; i < 5; i++) {
128       MenuEntry menuEntry = new MenuEntry();
129       menuEntry.setLabel("menu-entry-1-" + i);
130       menuEntry.setMenu(menu);
131       menuEntry.setOrdinal(i);
132       menuEntry.setPrice(BigDecimal.ONE);
133       menuEntries.add(menuEntry);
134     }
135     menuEntries = saveEntities(menuEntries);
136
137     // -- When --
138     final List<MenuEntry> fetchedMenuEntries = dao.getAllForMenu(menu.getId());
139
140     // -- Then --
141     assertEquals(menuEntries.size(), fetchedMenuEntries.size());
142     assertEquals(menuEntries, fetchedMenuEntries);
143   }
144
145   @Test
146   public void byIdNull() throws RemoteException {
147     // -- Given --
148     final Integer id = null;
149     expectedException.expect(new RemoteDetailMatcher(NullPointerException.class));
150
151     // -- When --
152     dao.byId(id);
153   }
154
155   @Test
156   public void byId() throws RemoteException {
157     // -- Given --
158     MenuEntry menuEntry = new MenuEntry();
159     menuEntry.setLabel("menu-entry-1");
160     menuEntry.setMenu(menu);
161     menuEntry.setOrdinal(0);
162     menuEntry.setPrice(BigDecimal.ONE);
163     menuEntry = saveEntity(menuEntry);
164
165     // -- When --
166     final MenuEntry menuEntryDB = dao.byId(menuEntry.getId());
167
168     // -- Then --
169     assertEquals(menu.getId(), menuEntryDB.getId());
170   }
171 }
```

1.5.11 OrderDaoTest.java

Diese Testklasse testet das DAO `OrderDaoTest`

Listing 31: OrderDaoTest.java

```
1  package at.fh.ooe.swe4.campina.test.dao.impl;
2
3  import static org.junit.Assert.assertEquals;
4  import static org.junit.Assert.assertTrue;
5  import static org.junit.Assert.fail;
6
7  import java.math.BigDecimal;
8  import java.rmi.RemoteException;
9  import java.sql.Connection;
10 import java.sql.SQLException;
11 import java.util.ArrayList;
12 import java.util.Calendar;
13 import java.util.List;
14
15 import org.junit.After;
16 import org.junit.Before;
17 import org.junit.Test;
18
19 import at.fh.ooe.swe4.campina.dao.api.OrderDao;
20 import at.fh.ooe.swe4.campina.dao.impl.OrderDaoImpl;
21 import at.fh.ooe.swe4.campina.persistence.api.entity.Menu;
22 import at.fh.ooe.swe4.campina.persistence.api.entity.MenuEntry;
23 import at.fh.ooe.swe4.campina.persistence.api.entity.Order;
24 import at.fh.ooe.swe4.campina.persistence.api.entity.User;
25 import at.fh.ooe.swe4.campina.persistence.api.entity.constants.Day;
26 import at.fh.ooe.swe4.campina.test.dao.api.AbstractDaoTest;
27 import at.fh.ooe.swe4.campina.test.dao.api.RemoteDetailMatcher;
28
29 /**
30  * This test class represents the test class for the {@link OrderDao}
31  * implementation.
32  *
33  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
34  * @date Jun 20, 2015
35  */
36 public class OrderDaoTest extends AbstractDaoTest<Order> {
37
38   private final OrderDao  dao;
39
40   public OrderDaoTest() throws RemoteException {
41     super(Order.class);
42     this.dao = new OrderDaoImpl(conManager);
43   }
44
45   @Before
46   public void beforeTest() {
47     setupDB();
48   }
49
50   @After
51   public void afterTest() {
52     setupDB();
53   }
54
55   @Test
56   public void saveNull() throws RemoteException {
57     // -- Given --
58     final Order order = null;
```

```
59      expectedException.expect(new RemoteDetailMatcher(NullPointerException.class));
60
61      // -- When | Then --
62      dao.save(order);
63    }
64
65    @Test
66    public void save() throws RemoteException {
67      // -- Given --
68      User user = new User();
69      user.setFirstName("Thomas-");
70      user.setLastName("Herzog-");
71      user.setUsername("cchet-");
72      user.setEmail("t.t@t.at");
73      user.setPassword("xxxxxxx");
74      user.setAdminFlag(Boolean.TRUE);
75      user.setBlockedFlag(Boolean.FALSE);
76      user = saveEntity(user);
77
78      Menu menu = new Menu();
79      menu.setDay(Day.MONDAY);
80      menu.setLabel("menu-1");
81      menu = saveEntity(menu);
82
83      MenuEntry menuEntry = new MenuEntry();
84      menuEntry.setLabel("menu--entry-1");
85      menuEntry.setMenu(menu);
86      menuEntry.setOrdinal(1);
87      menuEntry.setPrice(BigDecimal.ONE);
88      menuEntry = saveEntity(menuEntry);
89
90      Order order = new Order();
91      order.setMenuEntry(menuEntry);
92      order.setOrderDate(Calendar.getInstance());
93      order.setCollectDate(Calendar.getInstance());
94      order.setUser(user);
95      order = saveEntity(order);
96
97      // -- When --
98      order = dao.save(order);
99
100     // -- Then --
101     try (final Connection con = conManager.getConnection(Boolean.TRUE);) {
102       final Order orderDB = em.byId(con, user.getId());
103       assertEquals(order, orderDB);
104     } catch (SQLException e) {
105       fail("Could not obtain connection");
106     }
107   }
108
109   @Test
110   public void getAllEmpty() throws RemoteException {
111     // -- Given | When --
112     final List<Order> fetchedOrders = dao.getAll();
113
114     // -- Then --
115     assertTrue(fetchedOrders.isEmpty());
116   }
117
118   @Test
119   public void getAll() throws RemoteException {
120     // -- Given --
121     List<Order> orders = new ArrayList<>(5);
```

```
122     User user = new User();
123     user.setFirstName("Thomas-");
124     user.setLastName("Herzog-");
125     user.setUsername("cchet-");
126     user.setEmail("t.t@t.at");
127     user.setPassword("xxxxxxx");
128     user.setAdminFlag(Boolean.TRUE);
129     user.setBlockedFlag(Boolean.FALSE);
130     user = saveEntity(user);
131
132     Menu menu = new Menu();
133     menu.setDay(Day.MONDAY);
134     menu.setLabel("menu-1");
135     menu = saveEntity(menu);
136
137     MenuEntry menuEntry = new MenuEntry();
138     menuEntry.setLabel("menu--entry-1");
139     menuEntry.setMenu(menu);
140     menuEntry.setOrdinal(1);
141     menuEntry.setPrice(BigDecimal.ONE);
142     menuEntry = saveEntity(menuEntry);
143
144     for (int i = 0; i < 10; i++) {
145       final Order order = new Order();
146       order.setMenuEntry(menuEntry);
147       order.setOrderDate(Calendar.getInstance());
148       order.setCollectDate(Calendar.getInstance());
149       order.setUser(user);
150     }
151     orders = saveEntities(orders);
152
153     // -- When --
154     final List<Order> fetchedOrders = dao.getAll();
155
156     // -- Then --
157     assertEquals(orders.size(), fetchedOrders.size());
158     assertEquals(orders, fetchedOrders);
159     for (Order order : fetchedOrders) {
160       assertEquals(order.getUser(), user);
161       assertEquals(order.getMenuEntry(), menuEntry);
162       assertEquals(order.getMenuEntry()
163                 .getMenu(), menu);
164     }
165   }
166 }
```

1.6 Source swe-campina-rmi-api

Folgend ist der Source des Projekts *swe-campina-rmi-api* angeführt, welches die Spezifikation für den RMI Server darstellt.

1.6.1 RmiServer.java

Dieses Interface spezifiziert den RMI Server.

Listing 32: RmiServer.java

```
1  package at.fh.ooe.swe4.campina.rmi.api.rmi;
2
3  import java.io.Serializable;
4  import java.rmi.Remote;
5  import java.rmi.RemoteException;
6
7  /**
8   * This interface is the specification for the RMI server.
9   *
10  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
11  * @date Jun 15, 2015
12  */
13 public interface RmiServer extends Serializable {
14
15   /**
16    * Registers an bean. The RMI server must have been started before.
17    *
18    * @param bean
19    *            the bean instance to register
20    * @param interfaceClazz
21    *            the class of the current instance which will be used for name
22    *            resolving.
23    * @throws RemoteException
24    *             if the bean could not be registered
25    * @throws NullPointerException
26    *             if the bean or the interface are null
27    * @throws IllegalStateException
28    *             if the RMI hasn't been started before
29    */
30   public <T extends Remote> void bindBean(T bean, Class<T> interfaceClazz) throws RemoteException;
31
32   /**
33    * Unbinds a registered bean if the service is bind to the backed RMI
34    * server.
35    *
36    * @param interfaceClazz
37    *            the registered bean interface class
38    * @throws RemoteException
39    *             if the RMI server is not started
40    * @throws NullPointerException
41    *             if the interface class is null
42    */
43   public <T extends Remote> void unbindBean(Class<T> interfaceClazz) throws RemoteException;
44
45   /**
46    * SWtarts the RMI server.
47    *
48    * @throws RemoteException
49    *             if the RMI server could not be started
50    * @throws NullPointerException
51    *             if the URL is null
52    */
```

```
53    public void start() throws RemoteException;
54
55    /**
56     * Stops the RMI server and ends all connections, therefore client could
57     * experience {@link RemoteException}.
58     *
59     * @throws RemoteException
60     *             if the rmi server could not be started
61     */
62    public void stop() throws RemoteException;
63  }
```

1.6.2 RmiDaoFactory.java

Dieses Interface spezifiziert die RMI DAO Factory.

Listing 33: RmiDaoFactory.java

```
1   package at.fh.ooe.swe4.campina.rmi.api.factory;
2
3   import java.rmi.Remote;
4   import java.rmi.RemoteException;
5
6   /**
7    * This interface specifies a RMI DAO factory which creates instances for the
8    * given name.
9    *
10   * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
11   * @date Jun 15, 2015
12   */
13  public interface RmiDaoFactory extends Remote {
14
15    /**
16     * Creates a DAO for the given name.
17     *
18     * @param interfaze
19     *             the DAO name which must be exactly the Interface name of the
20     *             DAO.
21     * @return the DAO instance
22     * @throws IllegalArgumentException
23     *              if the name does not map to a DAO interface
24     */
25    public <T extends Remote> T createDao(Class<T> interfaze) throws RemoteException;
26  }
```

1.7 Source swe-campina-rmi-impl

Folgend ist der Source des Projekts *swe-campina-rmi-impl* angeführt, welches die Implementierung der Spezifikation *swe-campina-rmi-impl*.

1.7.1 RmiServerImpl.java

Diese Klasse stellt die Implementierung der Spezifikation *RmiServer* dar.

Listing 34: RmiServerImpl.java

```
1  package at.fh.ooe.swe4.campina.rmi.impl;
2
3  import java.rmi.AlreadyBoundException;
4  import java.rmi.NotBoundException;
5  import java.rmi.Remote;
6  import java.rmi.RemoteException;
7  import java.rmi.registry.LocateRegistry;
8  import java.rmi.registry.Registry;
9  import java.rmi.server.UnicastRemoteObject;
10 import java.util.HashSet;
11 import java.util.Objects;
12 import java.util.Set;
13
14 import org.apache.log4j.Logger;
15
16 import at.fh.ooe.swe4.campina.rmi.api.rmi.RmiServer;
17
18 /**
19  * This is the RMI server implementation.
20  *
21  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
22  * @date Jun 15, 2015
23  */
24 public class RmiServerImpl implements RmiServer {
25
26   private static final long  serialVersionUID    = 164219497553939223L;
27
28   private Registry      serviceRegistry      = null;
29   private int           port;
30   private Set<String>       registeredServiceNames  = new HashSet<>();
31   private static final Logger  log             = Logger.getLogger(RmiServerImpl.class);
32
33   /**
34    * @param port
35    *            the prot to host the beans on.
36    */
37   public RmiServerImpl(int port) {
38     if ((port <= 1024) || (port >= 65535)) {
39       throw new IllegalArgumentException("port is invalid: " + port);
40     }
41
42     this.port = port;
43   }
44
45   @Override
46   public void start() throws RemoteException {
47     if (serviceRegistry != null) {
48       throw new IllegalStateException("The rmi server is already started");
49     }
50     log.info("Starting service regsitry on: 'rmi://localhost:" + port + "'");
51     serviceRegistry = LocateRegistry.createRegistry(port);
52     log.info("Service regsitry registered");
```

```
53    }
54
55    @Override
56    public void stop() throws RemoteException {
57      if (serviceRegistry == null) {
58        throw new IllegalStateException("The rmi server is not started");
59      }
60      try {
61        for (String name : registeredServiceNames) {
62          serviceRegistry.unbind(name);
63          log.info("Service '" + name + "' unbound");
64        }
65      } catch (NotBoundException e) {
66        throw new IllegalStateException("Should not happen :(");
67      }
68      serviceRegistry = null;
69      port = -1;
70      registeredServiceNames.clear();
71    }
72
73    @Override
74    public <T extends Remote> void bindBean(T service, Class<T> interfaceClazz) throws
      ↪  RemoteException {
75      Objects.requireNonNull(service);
76      Objects.requireNonNull(interfaceClazz);
77
78      final String name = interfaceClazz.getSimpleName();
79      if (registeredServiceNames.contains(name)) {
80        log.info("Service '" + name + "' will get bound");
81      }
82      try {
83        Remote rmiService = service;
84        if (!(service instanceof UnicastRemoteObject)) {
85          rmiService = UnicastRemoteObject.exportObject(service, port);
86        }
87        serviceRegistry.bind(name, rmiService);
88        registeredServiceNames.add(name);
89      } catch (AlreadyBoundException e) {
90        throw new IllegalStateException("Should not happen :(");
91      }
92
93    }
94
95    @Override
96    public <T extends Remote> void unbindBean(final Class<T> interfaceClazz) throws RemoteException
      ↪  {
97      Objects.requireNonNull(interfaceClazz);
98
99      if (registeredServiceNames.contains(interfaceClazz.getSimpleName())) {
100       log.info("Unbind Service: " + interfaceClazz.getName() + "");
101       try {
102         serviceRegistry.unbind(interfaceClazz.getSimpleName());
103       } catch (NotBoundException e) {
104         throw new IllegalStateException("Should not happen :(");
105       }
106     }
107   }
108 }
```

1.7.2 RmiDaoFactoryImpl.java

Diese Klasse stellt die Implementierung der Spezifikation *RmiDaoFactory* dar.

Listing 35: RmiDaoFactoryImpl.java

```
1  package at.fh.ooe.swe4.campina.rmi.impl;
2
3  import java.rmi.Remote;
4  import java.rmi.RemoteException;
5  import java.rmi.server.UnicastRemoteObject;
6  import java.util.HashMap;
7  import java.util.Iterator;
8  import java.util.Map;
9  import java.util.Map.Entry;
10 import java.util.Objects;
11 import java.util.SortedSet;
12 import java.util.Timer;
13 import java.util.TimerTask;
14 import java.util.TreeSet;
15
16 import org.apache.log4j.Logger;
17
18 import at.fh.ooe.swe4.campina.persistence.api.ConnectionManager;
19 import at.fh.ooe.swe4.campina.persistence.impl.ConnectionManagerImpl;
20 import at.fh.ooe.swe4.campina.persistence.impl.ConnectionManagerImpl.DbMetadata;
21 import at.fh.ooe.swe4.campina.rmi.api.factory.RmiDaoFactory;
22
23 /**
24  * This class creates the remote object for the client so that multiple instance
25  * can be used on the client site. Therefore the produced beans are considered
26  * to be stateless implemented beans, because the client is not supposed to
27  * expect anything more but an stateless bean.
28  *
29  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
30  * @date Jun 17, 2015
31  */
32 public class RmiDaofactoryImpl extends UnicastRemoteObject implements RmiDaoFactory {
33
34   private static final long                serialVersionUID  = 9162336931859659503L;
35
36   private Timer                        cleanupTimer;
37   private final Map<Class<Remote>, SortedSet<DaoWrapper>>  daoCache      = new HashMap<>(100,
      ↪  (float) 0.75);
38   private final DbMetadata                databaseMeta;
39   private final ConnectionManager              connectionManager;
40   private final Object                   lockObject      = new Object();
41
42   private static final Logger                log          =
      ↪  Logger.getLogger(RmiDaofactoryImpl.class);
43   private static final String                IMPL_NAME_SPACE    =
      ↪  "at.fh.ooe.swe4.campina.dao.impl.";
44
45   private static final int                CLIENT_COUNT    = 10;
46
47   /**
48    * Helper class for wrapping the cached DAO for caching purposes.
49    *
50    * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
51    * @date Jun 17, 2015
52    */
53   private static final class DaoWrapper implements Comparable<DaoWrapper> {
54
55     public int      clientCount;
```

```
56      public final Remote  instance;
57
58      /**
59       * @param instance
60       */
61      public DaoWrapper(Remote instance) {
62        super();
63        this.instance = instance;
64        this.clientCount = 0;
65      }
66
67      /**
68       * Ensures that the lowest client count DAO is the first element of the
69       * sorted container
70       */
71      @Override
72      public int compareTo(DaoWrapper o) {
73        return Integer.valueOf(clientCount)
74                .compareTo(o.clientCount);
75      }
76    }
77
78    /**
79     * Cleanup time task which ensures that not more than 10 references are
80     * bound to one DAO. If the service has the client count breached then the
81     * reference to this DAO will be released.
82     *
83     * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
84     * @date Jun 17, 2015
85     */
86    private static final class CleanupTask extends TimerTask {
87
88      private final RmiDaofactoryImpl  factory;
89
90      /**
91       * @param factory
92       */
93      public CleanupTask(RmiDaofactoryImpl factory) {
94        super();
95        this.factory = factory;
96      }
97
98      @Override
99      public void run() {
100       // ensures that clients have to wait for cleanup finished
101       synchronized (factory.lockObject) {
102         int count = 0;
103         for (Entry<Class<Remote>, SortedSet<DaoWrapper>> entry : factory.daoCache.entrySet()) {
104           final Iterator<DaoWrapper> it = entry.getValue()
105                                .iterator();
106           while (it.hasNext()) {
107             final DaoWrapper service = it.next();
108             if (service.clientCount >= CLIENT_COUNT) {
109               it.remove();
110               count++;
111             }
112           }
113         }
114         log.info("Finished DAO cache cleanup (" + count + " removed)");
115       }
116     }
117   }
118
```

```
119   /**
120    * @throws RemoteException
121    *             if remote object could not be created
122    */
123   public RmiDaofactoryImpl(final DbMetadata databaseMeta) throws RemoteException {
124     Objects.requireNonNull(databaseMeta);
125
126     this.cleanupTimer = new Timer();
127     this.cleanupTimer.schedule(new CleanupTask(this), 0, (int) (10 * 1000));
128     this.databaseMeta = databaseMeta;
129
130     // constructor tries to establish a connection and therefore validates
131     // the provided metadata
132     this.connectionManager = new ConnectionManagerImpl(databaseMeta);
133   }
134
135   @Override
136   public <T extends Remote> T createDao(Class<T> interfaze) throws RemoteException {
137     synchronized (this.lockObject) {
138
139       DaoWrapper dao;
140       SortedSet<DaoWrapper> typedDaoCache = daoCache.get(interfaze);
141       // -- No instance cached --
142       if (typedDaoCache == null) {
143         log.info("Init DAO cache: '" + interfaze.getSimpleName() + "'");
144         typedDaoCache = new TreeSet<DaoWrapper>();
145         daoCache.put((Class<Remote>) interfaze, typedDaoCache);
146       }
147       // -- empty cache --
148       if ((typedDaoCache.isEmpty()) || (typedDaoCache.first().clientCount >= CLIENT_COUNT)) {
149         typedDaoCache.add(new DaoWrapper(newDaoInstance(interfaze)));
150         log.info("Caching new DAO: '" + interfaze.getSimpleName() + "'");
151       }
152
153       // -- get first bean with lowest client count --
154       dao = typedDaoCache.first();
155
156       dao.clientCount++;
157       log.info("Retrieved cached DAO: '" + interfaze.getSimpleName() + "'");
158
159       if (dao.instance == null) {
160         throw new RemoteException("Creation of DAO: '" + interfaze.getName() + "' failed");
161       }
162
163       return (T) dao.instance;
164     }
165   }
166
167   /**
168    * Creates a new DAO instance.
169    *
170    * @param interfaze
171    *            the DAO interface class
172    * @return the DAO instance or null if the creation failed
173    */
174   private <T extends Remote> T newDaoInstance(final Class<T> interfaze) {
175     Objects.requireNonNull(interfaze);
176
177     T instance = null;
178     try {
179       Class<T> clazz = (Class<T>) Class.forName(IMPL_NAME_SPACE + interfaze.getSimpleName() +
        ↪  "Impl");
180       instance = clazz.getConstructor(ConnectionManager.class)
```

```
181                 .newInstance(this.connectionManager);
182       } catch (Throwable e) {
183         e.printStackTrace();
184         log.error("Could create service instance", e);
185       }
186
187       return instance;
188     }
189   }
```

1.7.3 MainServer.java

Diese Klasse stellt die Main Klasse für den RMI Server dar.

Listing 36: MainServer.java

```
1  package at.fh.ooe.swe4.campina.rmi.impl;
2
3  import at.fh.ooe.swe4.campina.persistence.impl.ConnectionManagerImpl.DbMetadata;
4  import at.fh.ooe.swe4.campina.persistence.impl.DbConfigParam;
5  import at.fh.ooe.swe4.campina.rmi.api.factory.RmiDaoFactory;
6  import at.fh.ooe.swe4.campina.rmi.api.rmi.RmiServer;
7
8  /**
9   * This is the main RMI server which hosts the {@link RmiDaoFactory} for the
10  * client applications.
11  *
12  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
13  * @date Jun 21, 2015
14  */
15 public class MainServer {
16
17   public MainServer() {
18     // TODO Auto-generated constructor stub
19   }
20
21   public static void main(String[] args) throws Throwable {
22     if (System.getSecurityManager() == null) {
23       System.setSecurityManager(new SecurityManager());
24     }
25     final DbMetadata metadata = new DbMetadata(DbConfigParam.DRIVER.val(),
26         DbConfigParam.URL.val(),
27         DbConfigParam.USER.val(),
28         DbConfigParam.PASSWORD.val(),
29         Integer.valueOf(DbConfigParam.ISOLATION.val()));
30     final RmiDaoFactory serviceFactory = new RmiDaofactoryImpl(metadata);
31     final RmiServer rmiServer = new RmiServerImpl(50555);
32     rmiServer.start();
33     rmiServer.bindBean(serviceFactory, RmiDaoFactory.class);
34   }
35 }
```

1.7.4 log4j.properties

Die Konfigurationsdatei für Log4j für die RMI Server hosting JVM.

Listing 37: log4j.properties

```
1 log4j.rootLogger=INFO,STDOUT
2
3 log4j.appender.STDOUT=org.apache.log4j.ConsoleAppender
4 log4j.appender.STDOUT.layout=org.apache.log4j.PatternLayout
5 log4j.appender.STDOUT.layout.ConversionPattern=%p\t%d{HH:mm:ss,SSS}\t- %m%n
```

1.8 Source swe-campina-fx

Folgend ist der Source des Projekts *swe-campina-fx* angeführt, wobei hier nur die Klassen, in dennen Änderungen vorgenommen wurden angeführt sind.
Die Domänen Modelle, die bereits vorhanden waren, wurden in das Projekt *swe-campina-data-model-api* verschoben. Die Simulation der Datenbank wurde entfernt.

1.8.1 UserEventControl.java

Diese Klasse für die Events bezüglich der Entität `User`.

Listing 38: UserEventControl.java

```
1  package at.fh.ooe.swe4.campina.fx.view.admin.user.control;
2
3  import java.rmi.RemoteException;
4  import java.util.List;
5  import java.util.Objects;
6
7  import javafx.collections.ObservableList;
8  import javafx.event.ActionEvent;
9  import javafx.scene.Node;
10 import javafx.scene.control.Button;
11 import javafx.scene.control.ChoiceBox;
12 import javafx.scene.text.Text;
13 import javafx.scene.text.TextFlow;
14
15 import org.apache.commons.lang.StringUtils;
16 import org.apache.log4j.Logger;
17
18 import at.fh.ooe.swe4.campina.dao.api.UserDao;
19 import at.fh.ooe.swe4.campina.dao.exception.EmailAlreadyUsedException;
20 import at.fh.ooe.swe4.campina.dao.exception.UsernameAlreadyUsedException;
21 import at.fh.ooe.swe4.campina.fx.rmi.service.locator.DaoLocator;
22 import at.fh.ooe.swe4.campina.fx.view.admin.user.model.UserModel;
23 import at.fh.ooe.swe4.campina.fx.view.admin.user.part.UserTabviewHandler;
24 import at.fh.ooe.swe4.campina.fx.view.api.FormContext;
25 import at.fh.ooe.swe4.campina.persistence.api.entity.User;
26
27 /**
28  * The control bean for the user tab.
29  *
30  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
31  * @date Jun 3, 2015
32  */
33 public class UserEventControl {
34
35   private final UserDao    dao;
36   private static final Logger  log  = Logger.getLogger(UserEventControl.class);
37
38   /**
39    * Creates test data since we have no back-end yet
40    */
41   public UserEventControl() {
42     dao = DaoLocator.getDao(UserDao.class);
43   }
44
45   // ##############################################################
46   // Button controls
47   // ##############################################################
48   /**
49    * Handles the new action of the form.
```

```
50      *
51      * @param event
52      *            the {@link ActionEvent}
53      */
54    public void handleNewAction(final ActionEvent event) {
55      final FormContext<UserModel> ctx = (FormContext<UserModel>) ((Node)
          ↪  event.getSource()).getUserData();
56      // clear former set message
57      populateFormMessage(null, ctx);
58      // reset form
59      ctx.formHandler.resetForm(ctx);
60      // create new user model with new user entity
61      ctx.model.reset();
62      // hide buttons
63      setButtonVisibility(ctx, Boolean.FALSE);
64      // reload users
65      handleUserReload(ctx);
66    }
67
68    /**
69     * Handles the save action of the form.
70     *
71     * @param event
72     *            the {@link ActionEvent}
73     */
74    public void handleSaveAction(final ActionEvent event) {
75      final FormContext<UserModel> ctx = (FormContext<UserModel>) ((Node)
          ↪  event.getSource()).getUserData();
76      // validate form
77      ctx.formHandler.validateForm(ctx);
78      // is valid
79      if (ctx.valid) {
80        // fill model with form data
81        ctx.formHandler.fillModel(ctx);
82
83        // TODO: Persist entity here
84        User user = ctx.model.getEntity();
85        try {
86          user = dao.save(user);
87          user = dao.byId(user.getId());
88          populateFormMessage(null, ctx);
89        } catch (RemoteException e) {
90          log.error("Could not save user", e);
91          if (e.detail != null) {
92            if (e.detail instanceof EmailAlreadyUsedException) {
93              populateFormMessage("Email bereits vergeben", ctx);
94            } else if (e.detail instanceof UsernameAlreadyUsedException) {
95              populateFormMessage("Bentuezrname bereits vergeben", ctx);
96            }
97          }
98        }
99        // init model with new saved user
100       ctx.model.prepare(user);
101       // enable buttons
102       setButtonVisibility(ctx, Boolean.TRUE);
103       // reload data from db (now backing list)
104       handleUserReload(ctx);
105     } else {
106       populateFormMessage("Formular ungültig !!! Bitte Eingaben prüfen", ctx);
107     }
108     event.consume();
109
110   }
```

```
111
112   /**
113    * Handles the delete action of the form.
114    *
115    * @param event
116    *            the {@link ActionEvent}
117    */
118   public void handleDeleteAction(final ActionEvent event) {
119     final FormContext<UserModel> ctx = (FormContext<UserModel>) ((Node)
       ↪  event.getSource()).getUserData();
120     // clear former set message
121     populateFormMessage(null, ctx);
122     // reset the form
123     ctx.formHandler.resetForm(ctx);
124
125     final UserModel model = ((ChoiceBox<UserModel>)
       ↪  ctx.getNode(UserTabviewHandler.USER_SELECTION_KEY)).getSelectionModel()
126                                                           .getSelectedItem();
127
128     // TODO: Delete entity from db here
129
130     // existing user gets deleted
131     if (model.getId() != null) {
132       try {
133         dao.delete(model.getEntity());
134       } catch (RemoteException e) {
135         log.error("Could not delete user", e);
136         populateFormMessage("Benutzer konnte nicht gelöscht werden", ctx);
137       }
138     }
139
140     // reset model
141     ctx.model = new UserModel();
142     // disable buttons
143     setButtonVisibility(ctx, Boolean.FALSE);
144     // reload users
145     handleUserReload(ctx);
146   }
147
148   /**
149    * Handles the block action of the form.
150    *
151    * @param event
152    *            the {@link ActionEvent}
153    */
154   public void handleBlockAction(final ActionEvent event) {
155     final FormContext<UserModel> ctx = (FormContext<UserModel>) ((Node)
       ↪  event.getSource()).getUserData();
156     // clear old set message
157     // selected user model
158     final UserModel model = ((ChoiceBox<UserModel>)
       ↪  ctx.getNode(UserTabviewHandler.USER_SELECTION_KEY)).getSelectionModel()
159                                                           .getSelectedItem();
160
161     final Button blockButton = (Button) ctx.getNode(UserTabviewHandler.BLOCK_BUTTON_ID);
162     User user = model.getEntity();
163
164     // invert user blocked state
165     user.setBlockedFlag(!model.getEntity()
166                   .getBlockedFlag());
167     try {
168       user = dao.save(user);
169       user = dao.byId(user.getId());
```

```
170       populateFormMessage(null, ctx);
171     } catch (RemoteException e) {
172       log.error("Could not block user", e);
173       populateFormMessage("Konnte Benutzer nicht sperren", ctx);
174     }
175
176     ctx.model.prepare(user);
177     ctx.formHandler.fillForm(ctx);
178
179     // TODO: Update blocked flag on db
180
181     // got blocked
182     if (model.getEntity()
183           .getBlockedFlag()) {
184       blockButton.setText("Freigeben");
185     }
186     // got freed
187     else {
188       blockButton.setText("Blockieren");
189     }
190     // user reload
191     handleUserReload(ctx);
192   }
193
194   // ################################################################
195   // Selection controls
196   // ################################################################
197   public void handleUserSelection(final FormContext<UserModel> ctx, final UserModel user) {
198     // clear former set message if new user
199     if (!ctx.model.equals(user)) {
200       populateFormMessage(null, ctx);
201     }
202     // Selection present
203     if (user.getId() != null) {
204       try {
205         final User userDB = dao.byId(user.getId());
206         user.prepare(userDB);
207         ctx.model.prepare(user.getEntity());
208         ctx.formHandler.fillForm(ctx);
209         setButtonVisibility(ctx, Boolean.TRUE);
210       } catch (RemoteException e) {
211         log.error("Could not load selected user", e);
212         ctx.model.reset();
213         ctx.formHandler.fillForm(ctx);
214         setButtonVisibility(ctx, Boolean.FALSE);
215       }
216     }
217     // No selection present
218     else {
219       ctx.model.reset();
220       ctx.formHandler.fillForm(ctx);
221       setButtonVisibility(ctx, Boolean.FALSE);
222     }
223   }
224
225   // ################################################################
226   // Load controls
227   // ################################################################
228   /**
229    * Handles the load of the user for the selection
230    *
231    * @param userList
232    *            the {@link ObservableList} to add users to
```

```
233     */
234   public void handleUserReload(final FormContext<UserModel> ctx) {
235     Objects.requireNonNull(ctx);
236
237     final ObservableList<UserModel> userList = (ObservableList<UserModel>)
        ↪  ctx.getObserable(UserTabviewHandler.USER_SELECTION_KEY);
238     userList.clear();
239     userList.add(new UserModel());
240     boolean found = Boolean.FALSE;
241     try {
242       final List<User> users = dao.getAll();
243       for (User user : users) {
244         if (user.equals(ctx.model.getEntity())) {
245           found = Boolean.TRUE;
246         }
247         userList.add(new UserModel(user));
248       }
249     } catch (RemoteException e) {
250       log.error("Coul not load users", e);
251       // TODO: handle exception
252     }
253
254     if (!found) {
255       ctx.model.reset();
256       ctx.formHandler.fillForm(ctx);
257       setButtonVisibility(ctx, Boolean.FALSE);
258     }
259     userList.set(userList.indexOf(ctx.model), ctx.model);
260
261     ((ChoiceBox<UserModel>)
        ↪  ctx.getNode(UserTabviewHandler.USER_SELECTION_KEY)).getSelectionModel()
262                                             .select(ctx.model);
263   }
264
265   /**
266    * Sets the button visibility of these buttons which required persistent
267    * user.
268    *
269    * @param ctx
270    *            the form context
271    * @param visible
272    *            the new visible flag
273    */
274   private void setButtonVisibility(final FormContext<UserModel> ctx, final boolean visible) {
275     Objects.requireNonNull(ctx);
276
277     ctx.getNode(UserTabviewHandler.DELETE_BUTTON_ID)
278       .setVisible(visible);
279     ctx.getNode(UserTabviewHandler.BLOCK_BUTTON_ID)
280       .setVisible(visible);
281   }
282
283   /**
284    * Populates a message to the message box. <br>
285    * If message is null the actual set message will be cleared
286    *
287    * @param message
288    *            the message to populate
289    * @param ctx
290    *            the form context
291    */
292   private void populateFormMessage(final String message, final FormContext<UserModel> ctx) {
293     final TextFlow flow = ((TextFlow) ctx.getNode(UserTabviewHandler.FORM_MESSAGE));
```

```
294       flow.getChildren()
295         .clear();
296       flow.setPrefHeight(0);
297       if (!StringUtils.isEmpty(message)) {
298         flow.getChildren()
299           .add(new Text(message));
300         flow.setPrefHeight(30);
301       }
302     }
303   }
```

1.8.2 MenuEventControl.java

Diese Klasse für die Events bezüglich der Entität `Menu`.

Listing 39: MenuEventControl.java

```
1  package at.fh.ooe.swe4.campina.fx.view.admin.menu.control;
2
3  import java.rmi.RemoteException;
4  import java.util.List;
5  import java.util.Objects;
6
7  import javafx.collections.ObservableList;
8  import javafx.event.ActionEvent;
9  import javafx.scene.Node;
10 import javafx.scene.control.ChoiceBox;
11 import javafx.scene.text.Text;
12 import javafx.scene.text.TextFlow;
13
14 import org.apache.commons.lang.StringUtils;
15 import org.apache.log4j.Logger;
16
17 import at.fh.ooe.swe4.campina.dao.api.MenuDao;
18 import at.fh.ooe.swe4.campina.fx.rmi.service.locator.DaoLocator;
19 import at.fh.ooe.swe4.campina.fx.view.admin.menu.model.MenuModel;
20 import at.fh.ooe.swe4.campina.fx.view.admin.menu.part.MenuTabViewHandler;
21 import at.fh.ooe.swe4.campina.fx.view.api.FormContext;
22 import at.fh.ooe.swe4.campina.persistence.api.entity.Menu;
23
24 /**
25  * The event control for the {@link Menu} entity.
26  *
27  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
28  * @date Jun 5, 2015
29  */
30 public class MenuEventControl {
31
32   private final MenuDao    dao  = DaoLocator.getDao(MenuDao.class);
33   private static final Logger  log  = Logger.getLogger(MenuEventControl.class);
34
35   public MenuEventControl() {
36   }
37
38   /**
39    * Handles the new action of the form.
40    *
41    * @param event
42    *            the {@link ActionEvent}
43    */
44   public void newAction(final ActionEvent event) {
45     final FormContext<MenuModel> ctx = (FormContext<MenuModel>) ((Node)
       ↪  event.getSource()).getUserData();
46     // clear former set message
47     populateFormMessage(null, ctx);
48     // reset form
49     ctx.formHandler.resetForm(ctx);
50     // create new user model with new user entity
51     ctx.model.reset();
52     // hide buttons
53     ctx.getNode(MenuTabViewHandler.MENU_DELETE_BUTTON_ID)
54       .setVisible(Boolean.FALSE);
55     // reload users
56     handleMenuReload(ctx);
57   }
```

```
58
59    /**
60     * Saves a menu
61     *
62     * @param event
63     *            the {@link ActionEvent}
64     */
65    public void saveMenu(final ActionEvent event) {
66      final FormContext<MenuModel> ctx = (FormContext<MenuModel>) ((Node)
        ↪  event.getSource()).getUserData();
67      ctx.getNode(MenuTabViewHandler.MENU_DELETE_BUTTON_ID)
68        .setVisible(Boolean.TRUE);
69      ctx.formHandler.validateForm(ctx);
70      ctx.formHandler.fillModel(ctx);
71
72      if (ctx.valid) {
73        try {
74          final Menu menu = dao.save(ctx.model.getEntity());
75          ctx.model.prepare(dao.byId(menu.getId()));
76          ctx.formHandler.fillForm(ctx);
77        } catch (RemoteException e) {
78          log.error("Could not save Menu", e);
79        }
80        handleMenuReload(ctx);
81      } else {
82        populateFormMessage("Formular ungültig !! Bitte Eingaben prüfen", ctx);
83        handleMenuReload(ctx);
84      }
85    }
86
87    /**
88     * Deletes an menu
89     *
90     * @param event
91     *            the {@link ActionEvent}
92     */
93    public void deleteMenu(final ActionEvent event) {
94      final FormContext<MenuModel> ctx = (FormContext<MenuModel>) ((Node)
        ↪  event.getSource()).getUserData();
95
96      if (ctx.model.getId() != null) {
97        try {
98          dao.delete(ctx.model.getEntity());
99        } catch (RemoteException e) {
100         log.error("Could not delete menu", e);
101       }
102       ctx.model.reset();
103       ctx.formHandler.fillForm(ctx);
104     }
105     handleMenuReload(ctx);
106   }
107
108   /**
109    * The handles the load of the menus. Resets the current ctx.model in the
110    * observed list and set this model as selected
111    *
112    * @param ctx
113    *            the form context
114    */
115   public void handleMenuReload(final FormContext<MenuModel> ctx) {
116     Objects.requireNonNull(ctx);
117
118     final ObservableList<MenuModel> list = (ObservableList<MenuModel>)
        ↪  ctx.getObserable(MenuTabViewHandler.MENU_SELECTION_KEY);
```

```
119     list.clear();
120     list.add(new MenuModel());
121
122     boolean found = Boolean.FALSE;
123     try {
124       final List<Menu> menus = dao.getAll();
125       for (Menu menu : menus) {
126         if (menu.equals(ctx.model.getEntity())) {
127           found = Boolean.TRUE;
128         }
129         final MenuModel model = new MenuModel();
130         model.prepare(menu);
131         list.add(model);
132       }
133     } catch (RemoteException e) {
134       log.error("Could not load menus", e);
135     }
136
137     if (!found) {
138       ctx.model.reset();
139       ctx.formHandler.fillForm(ctx);
140     }
141
142     // need to replace observed instance
143     list.set(list.indexOf(ctx.model), ctx.model);
144
145     // need to select current context hold model
146     ((ChoiceBox<MenuModel>)
      ↪   ctx.getNode(MenuTabViewHandler.MENU_SELECTION_KEY)).getSelectionModel()
147                                             .select(ctx.model);
148   }
149
150   /**
151    * Populates a message to the message box. <br>
152    * If message is null the actual set message will be cleared
153    *
154    * @param message
155    *            the message to populate
156    * @param ctx
157    *            the form context
158    */
159   private void populateFormMessage(final String message, final FormContext<MenuModel> ctx) {
160     final TextFlow flow = ((TextFlow) ctx.getNode(MenuTabViewHandler.MENU_FORM_MESSAGE));
161     flow.getChildren()
162       .clear();
163     flow.setPrefHeight(0);
164     if (!StringUtils.isEmpty(message)) {
165       flow.getChildren()
166         .add(new Text(message));
167       flow.setPrefHeight(30);
168     }
169   }
170 }
```

1.8.3 MenuEntryEventControl.java

Diese Klasse für die Events bezüglich der Entität `MenuEntry`.

Listing 40: MenuEntryEventControl.java

```
1  package at.fh.ooe.swe4.campina.fx.view.admin.menu.control;
2
3  import java.rmi.RemoteException;
4  import java.util.ArrayList;
5  import java.util.List;
6  import java.util.Objects;
7
8  import javafx.collections.ObservableList;
9  import javafx.event.ActionEvent;
10 import javafx.scene.Node;
11 import javafx.scene.control.ChoiceBox;
12 import javafx.scene.text.Text;
13 import javafx.scene.text.TextFlow;
14
15 import org.apache.commons.lang.StringUtils;
16 import org.apache.log4j.Logger;
17
18 import at.fh.ooe.swe4.campina.dao.api.MenuDao;
19 import at.fh.ooe.swe4.campina.dao.api.MenuEntryDao;
20 import at.fh.ooe.swe4.campina.fx.rmi.service.locator.DaoLocator;
21 import at.fh.ooe.swe4.campina.fx.view.admin.menu.model.MenuEntryModel;
22 import at.fh.ooe.swe4.campina.fx.view.admin.menu.part.MenuTabViewHandler;
23 import at.fh.ooe.swe4.campina.fx.view.api.FormContext;
24 import at.fh.ooe.swe4.campina.persistence.api.entity.Menu;
25 import at.fh.ooe.swe4.campina.persistence.api.entity.MenuEntry;
26
27 /**
28  * The event handler for the {@link MenuEntry} entity.
29  *
30  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
31  * @date Jun 5, 2015
32  */
33 public class MenuEntryEventControl {
34
35   private final MenuEntryDao  dao    = DaoLocator.getDao(MenuEntryDao.class);
36   private final MenuDao    menuDao  = DaoLocator.getDao(MenuDao.class);
37   private static final Logger  log    = Logger.getLogger(MenuEntryEventControl.class);
38
39   public MenuEntryEventControl() {
40   }
41
42   /**
43    * Handles the new action of the form.
44    *
45    * @param event
46    *            the {@link ActionEvent}
47    */
48   public void newAction(final ActionEvent event) {
49     final FormContext<MenuEntryModel> ctx = (FormContext<MenuEntryModel>) ((Node)
       ↪  event.getSource()).getUserData();
50     // clear former set message
51     populateFormMessage(null, ctx);
52     // reset form
53     ctx.formHandler.resetForm(ctx);
54     // create new user model with new user entity
55     ctx.model.reset();
56     // hide buttons
57     ctx.getNode(MenuTabViewHandler.MENU_ENTRY_DELETE_BUTTON_ID)
```

```
58        .setVisible(Boolean.FALSE);
59      // reload users
60      handleMenuEntryReload(ctx);
61    }
62
63    /**
64     * Saves a {@link MenuEntry}
65     *
66     * @param event
67     *            the {@link ActionEvent}
68     */
69    public void saveMenuEntry(final ActionEvent event) {
70      final FormContext<MenuEntryModel> ctx = (FormContext<MenuEntryModel>) ((Node)
        ↪  event.getSource()).getUserData();
71      ctx.getNode(MenuTabViewHandler.MENU_ENTRY_DELETE_BUTTON_ID)
72        .setVisible(Boolean.TRUE);
73      ctx.formHandler.validateForm(ctx);
74      ctx.formHandler.fillModel(ctx);
75
76      if (ctx.valid) {
77        try {
78          final MenuEntry menuEntry = dao.save(ctx.model.getEntity());
79          ctx.model.prepare(dao.byId(menuEntry.getId()));
80          ctx.formHandler.fillForm(ctx);
81        } catch (RemoteException e) {
82          log.error("Menu entry saving failed", e);
83          populateFormMessage("Menu Eintrag konnte nicht gespeichert werden", ctx);
84        }
85      } else {
86        populateFormMessage("Formular ungültig !! Bitte Eingaben prüfen", ctx);
87      }
88      handleMenuEntryReload(ctx);
89    }
90
91    /**
92     * Deletes a {@link MenuEntry}
93     *
94     * @param event
95     *            the {@link ActionEvent}
96     */
97    public void deleteMenuEntry(final ActionEvent event) {
98      final FormContext<MenuEntryModel> ctx = (FormContext<MenuEntryModel>) ((Node)
        ↪  event.getSource()).getUserData();
99
100     if (ctx.model.getId() != null) {
101       try {
102         dao.delete(ctx.model.getEntity());
103         ctx.model.prepare(new MenuEntry());
104         // handleMenuLoad(ctx);
105         ctx.formHandler.fillForm(ctx);
106       } catch (RemoteException e) {
107         log.error("Could not delete menu entry", e);
108         populateFormMessage("Konnte Menu Enintrag nicht löschen", ctx);
109       }
110     }
111     handleMenuEntryReload(ctx);
112   }
113
114   /**
115    * Handles the {@link MenuEntry} reload
116    */
117   public void handleMenuEntryReload(final FormContext<MenuEntryModel> ctx) {
118     Objects.requireNonNull(ctx);
```

```
119
120     final ObservableList<MenuEntryModel> list = (ObservableList<MenuEntryModel>)
         ↪  ctx.getObserable(MenuTabViewHandler.MENU_ENTRY_SELECTION_KEY);
121     list.clear();
122     list.add(new MenuEntryModel());
123     List<MenuEntry> entries = new ArrayList<>();
124     boolean found = Boolean.FALSE;
125     try {
126       entries = dao.getAll();
127       for (MenuEntry menuEntry : entries) {
128         if (menuEntry.equals(ctx.model.getEntity())) {
129           found = Boolean.TRUE;
130         }
131         final MenuEntryModel model = new MenuEntryModel();
132         model.prepare(menuEntry);
133         list.add(model);
134       }
135     } catch (RemoteException e) {
136       log.error("Could ot load all menu entries", e);
137     }
138
139     if (!found) {
140       ctx.model.reset();
141       ctx.formHandler.fillForm(ctx);
142     }
143
144     // need to replace observed instance
145     list.set(list.indexOf(ctx.model), ctx.model);
146
147     // need to select current context hold model
148     ((ChoiceBox<MenuEntryModel>)
         ↪  ctx.getNode(MenuTabViewHandler.MENU_ENTRY_SELECTION_KEY)).getSelectionModel()
149                                                 .select(ctx.model);
150   }
151
152   /**
153    * Handles the {@link Menu} reload.
154    *
155    * @param ctx
156    *            teh form context
157    */
158   public void handleMenuLoad(final FormContext<MenuEntryModel> ctx) {
159     Objects.requireNonNull(ctx);
160
161     ctx.model.getMenus()
162           .clear();
163     ctx.model.getMenus()
164           .add(null);
165     try {
166       ctx.model.getMenus()
167             .addAll(menuDao.getAll());
168       int idx = 0;
169       if ((idx = ctx.model.getMenus()
170                 .indexOf(ctx.model.getMenu())) != -1) {
171         ctx.model.getEntity()
172               .setMenu(ctx.model.getMenus()
173                           .get(idx));
174       }
175     } catch (RemoteException e) {
176       log.error("Cannot load referencing menu entry menus", e);
177     }
178   }
179
```

```
180   /**
181    * Populates a message to the message box. <br>
182    * If message is null the actual set message will be cleared
183    *
184    * @param message
185    *            the message to populate
186    * @param ctx
187    *            the form context
188    */
189   private void populateFormMessage(final String message, final FormContext<MenuEntryModel> ctx) {
190     final TextFlow flow = ((TextFlow) ctx.getNode(MenuTabViewHandler.MENU_ENTRY_FORM_MESSAGE));
191     flow.getChildren()
192       .clear();
193     flow.setPrefHeight(0);
194     if (!StringUtils.isEmpty(message)) {
195       flow.getChildren()
196         .add(new Text(message));
197       flow.setPrefHeight(30);
198     }
199   }
200 }
```

1.8.4 MainFX.java

Diese Klasse stellt die Main Klasse für die JavaFX Applikation dar.

Listing 41: MainFX.java

```
1  package at.fh.ooe.swe4.campina.fx.main;
2
3  import java.math.BigDecimal;
4  import java.rmi.RemoteException;
5  import java.util.Calendar;
6  import java.util.Random;
7
8  import javafx.application.Application;
9  import javafx.scene.Scene;
10 import javafx.stage.Stage;
11 import at.fh.ooe.swe4.campina.dao.api.MenuDao;
12 import at.fh.ooe.swe4.campina.dao.api.MenuEntryDao;
13 import at.fh.ooe.swe4.campina.dao.api.OrderDao;
14 import at.fh.ooe.swe4.campina.dao.api.UserDao;
15 import at.fh.ooe.swe4.campina.fx.rmi.service.locator.DaoLocator;
16 import at.fh.ooe.swe4.campina.fx.view.scene.MainSceneViewHandler;
17 import at.fh.ooe.swe4.campina.persistence.api.entity.Menu;
18 import at.fh.ooe.swe4.campina.persistence.api.entity.MenuEntry;
19 import at.fh.ooe.swe4.campina.persistence.api.entity.Order;
20 import at.fh.ooe.swe4.campina.persistence.api.entity.User;
21 import at.fh.ooe.swe4.campina.persistence.api.entity.constants.Day;
22
23 /**
24  * Main class which starts the JavaFX application.<br>
25  * It creates test data if the user who is tried to be create does not exists.
26  * Otherwise it is assumed that the test data already exist.
27  *
28  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
29  * @date Jun 21, 2015
30  */
31 public class MainFX extends Application {
32
33   /**
34    *
35    */
36   public MainFX() {
37   }
38
39   @Override
40   public void start(Stage primaryStage) throws Exception {
41     final MainSceneViewHandler def = new MainSceneViewHandler();
42     final Scene scene = def.createNode();
43     def.initHandler();
44     primaryStage.setScene(scene);
45     primaryStage.setMinWidth(700);
46     primaryStage.setMinHeight(300);
47     primaryStage.setTitle("Campina");
48
49     primaryStage.show();
50   }
51
52   /**
53    * Prepares the test data for the application.
54    */
55   private static void prepareData() {
56     User user = new User();
57     user.setFirstName("THomas");
58     user.setLastName("Herzog");
```

```
59      user.setAdminFlag(Boolean.TRUE);
60      user.setBlockedFlag(Boolean.FALSE);
61      user.setEmail("thomas.herzog@students.fh-hangeberg.at");
62      user.setPassword("x");
63      user.setUsername("cchet");
64      try {
65        user = DaoLocator.getDao(UserDao.class)
66                  .save(user);
67
68        Menu menu = new Menu();
69        menu.setDay(Day.FRIDAY);
70        menu.setLabel("Freitagsmenu");
71        menu = DaoLocator.getDao(MenuDao.class)
72                  .save(menu);
73
74        MenuEntry menuEntry = new MenuEntry();
75        menuEntry.setLabel("Fisch");
76        menuEntry.setMenu(menu);
77        menuEntry.setOrdinal(1);
78        menuEntry.setPrice(BigDecimal.ONE);
79        menuEntry = DaoLocator.getDao(MenuEntryDao.class)
80                    .save(menuEntry);
81
82        final Random r = new Random(System.currentTimeMillis());
83        for (int i = 0; i < 10; i++) {
84          Order order = new Order();
85          order.setMenuEntry(menuEntry);
86          final Calendar orderDate = Calendar.getInstance();
87          orderDate.add(Calendar.DAY_OF_YEAR, (r.nextInt(30) + 1));
88          final Calendar collectDate = (Calendar) orderDate.clone();
89          collectDate.add(Calendar.DAY_OF_YEAR, (r.nextInt(10) + 1));
90          order.setOrderDate(orderDate);
91          order.setCollectDate(collectDate);
92          order.setUser(user);
93          order = DaoLocator.getDao(OrderDao.class)
94                      .save(order);
95        }
96      } catch (RemoteException e) {
97
98      }
99
100   }
101
102   /**
103    * @param args
104    * @throws Throwable
105    */
106   public static void main(String args[]) throws Throwable {
107     prepareData();
108     launch(args);
109   }
110
111 }
```

1.9 Test swe-campina-dao-impl

Folgend sind die Tests für das Projekt *swe-campina.dao-impl* angeführt, wobei hier lediglich die JUnit Tests aus Eclipse angeführt sind.
Die DAOs können ohne weitere Infrastruktur getestet werden, und die Test haben zwar Abhängigkeiten auf RMI spezifische API jedoch ist kein registrieren der DAOs über RMI erforderlich.

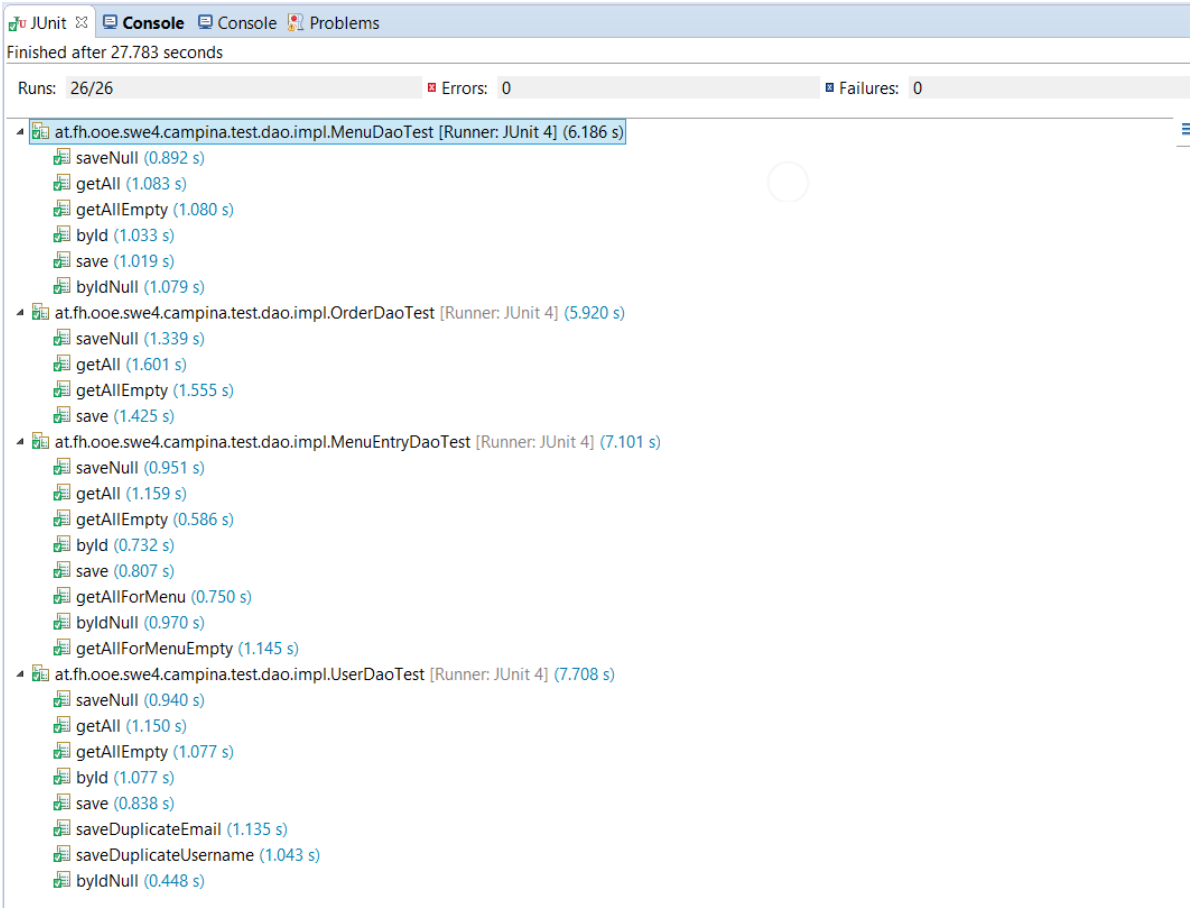

Abbildung 1: Die JUnit Test aus Eclipse