Datenbanksysteme 2, 8. Übung
Hinweise zur Lösung des praktischen Teils

In diesem Dokument sind typische Fragestellungen und Beispiele für Fehler zusammengestellt, die bei der Implementierung der zweiten Bonusaufgabe aufgetreten sind. Bitte gehen Sie die genannten Punkte sorgfältig durch und prüfen Sie Ihre eigene Lösung in Bezug auf die genannten Punkte.

Folgende Themen werden diskutiert:

Anwendung mit klarer Paketstruktur verwenden und diese auch beachten

Es handelt sich um eine klassische Drei-Schichten-Anwendung (vgl. Folie 5-5). Dies sollte sich auch in der entsprechenden Java-Paketstruktur wiederfinden. Es sollte also (mindestens) ein Paket gui für die gegebenen GUI-Elemente, ein Paket logic für die Manager- und DTO-Klassen sowie ein Paket persistence für Entity-Klassen und Factories geben; letztere Arten kann man auch in zwei getrennte Pakete einordnen. Diese Aufteilung muss auch logisch im Java-Code entsprechend eingehalten werden.
Es ist z.B. sinnvoll, in den Manager-Klassen keine JP-QL Anweisungen aufzunehmen (diese sollten ggfs. über Factory-Klassen aufgerufen werden). Die Manager-Klassen für diese Anwendung sollten recht kurz sein, da die Anwendungslogik hier nicht besonders komplex ist. In meiner Lösung sind fast alle Methoden in den Manager-Klassen zwischen 1 und 15 Zeilen lang. Einzige Ausnahme ist insertUpdateMovie in MovieManager, die allerdings auch nur ca. 30 Zeilen umfasst und damit deutlich kürzer als in der ersten Projektaufgabe ist.
Ebenso sollten in einer MovieFactory nur Anfragen an die Movie Entitäten erfolgen und bspw. keine Anfragen an Person Entitäten. Methoden in den Factory-Klassen sind typsicherweise statisch und liefern Entitäten der zugehörigen Typen anhand bestimmter Kriterien zurück.

Beispiele (zur Übersichtlichkeit hier ohne Exception-Handling, dazu s. unten):

```
// PersonManager
public List<String> getPersonList(String text) {
    return PersonFactory.getPersonNamesByPattern(text);
}

// MovieManager
public MovieDTO getMovie(long movieId) {
    EntityManager em = Emf.getEM();
    em.getTransaction().begin();
    MovieDTO ret = Mapper.movieToDTO(MovieFactory.findById(movieId));
    em.getTransaction().commit();
    return ret;
}

// Mapper
public static MovieDTO movieToDTO(Movie movie) {
    MovieDTO movieDTO = new MovieDTO();
    movieDTO.setId(movie.getId());
    movieDTO.setTitle(movie.getTitle());
    movieDTO.setYear(movie.getYear());
    movieDTO.setType(Character.toString(movie.getType()));
    for (Genre g : movie.getGenres()) {
        movieDTO.addGenre(g.getGenre());
    }
    for (MovieCharacter mC : movie.getMc()) {
        movieDTO.addCharacter(new CharacterDTO(mC.getCharacter(), mC.getAlias(),
mC.getPerson().getName()));
    }
    return movieDTO;
}

// PersonFactory
public static List<String> getPersonNamesByPattern(String pattern) {
    EntityManager em = Emf.getEM();
    em.getTransaction().begin();
    List<String> list = em.createQuery("SELECT p.name FROM Person p WHERE
LOWER(p.name) LIKE LOWER('%" + pattern + "%')").getResultList();
    em.getTransaction().commit();
```

```
    return list;
}
```

EntityManagerFactory ist ein Singleton

Die EntityManagerFactory sollte nur einmal erzeugt werden. Das Objekt ist schwergewichtig, d.h. das Erzeugen des Objektes ist sehr langsam. Beim Erzeugen wird die Datenbankverbindung aufgebaut, und es wird geprüft, ob das Schema in der Datenbank zu den Entities passt, und (je nach Einstellung) das Schema in der Datenbank ggf. auch aktualisiert oder ganz neu angelegt. Eine EntityManagerFactory sollte also zu Beginn des Programms einmalig erzeugt werden und überall im Programm sollte nur dieses eine Objekt verwendet werden. Dazu kann das Entwurfsmuster "Singleton" zum Einsatz kommen.

Umgang mit dem EntityManager und Transaktionen

1. Immer eine Transaktion verwenden?
Wenn mit einem EntityManager gearbeitet wird, sollte **immer** eine Transaktion verwendet werden, die über `em.getTransaction().begin()` gestartet wurde. Dies gilt für lesenden wie für schreibenden Datenbankzugriff.
Wenn man dies nicht macht, ist das Verhalten des EntityManagers insbesondere bei schreibenden Zugriffen nicht klar und hängt ggf. von der verwendeten JPA-Bibliothek ab (Hibernate, EclipseLink, ...). In der von mir verwendeten Hibernate-Version wird z.B. ein persist-Aufruf ohne Transaktion ignoriert, d.h. der Datensatz wird nicht gespeichert, aber es wird auch kein Fehler erzeugt.
Lesende Zugriffe funktionieren zumeist, aber es ist nicht garantiert, dass zusammenhängende Operationen auch einen konsistenten Datenstand zurückliefern. Daher sollten Sie auch bei lesenden Operationen immer mit Transaktionen arbeiten.

2. Mehrere EntityManager pro Transaktion?
Im Verlauf einer Transaktion sollte auch immer der gleiche EntityManager verwendet werden. Der EntityManager ist die zentrale Schnittstelle zwischen Datenbank und den Objekten im Speicher; daher sollte diese Schnittstelle nur einmal existieren. Wenn in einer Untermethode ein Objekt über einen EntityManager geladen wird, und dieser in der Untermethode wieder geschlossen wird, geht das Objekt in den Zustand "detached" über und muss ggf. mit "merge" wieder mit einem anderen EntityManager verknüpft werden. Daher sollten Sie es vermeiden, in kleineren Hilfsmethoden einen neuen EntityManager zu erstellen. Zusätzlich ermöglicht dies dem EntityManager das Cachen von Objekten.

3. Ein EntityManager für mehrere Transaktionen?
Umgekehrt stellt sich die Frage, ob ein EntityManager für das ganze Programm ausreicht oder ob pro Transaktion ein neuer EntityManager erzeugt werden sollte. Hier sollten Sie sich klar machen, dass ein Objekt ggf. aus dem Cache gelesen wird, wenn es über den EntityManager angefordert wird. Dies kann man sich ansehen, wenn man auf die an die Datenbank geschickten SQL-Statements bspw. im Log achtet. Hier kommt ggf. eine find-Methode ohne SQL-Befehl aus, wenn das Ergebnis aus dem Cache geladen wurde. Das Caching hat zur Folge, dass Änderungen, die zwischenzeitlich von einem anderen Programm an der Datenbank vorgenommen wurden, nicht zu aktualisierten Objekten im Speicher führen. Konkret in der GUI führt dies dazu, dass in der Detailansicht eines Films immer die Werte zu sehen sind, die das MovieDTO beim ursprünglichen Laden aus der Datenbank hatte (die also auch in der Filmliste zu sehen sind). Wenn Sie für jede Transaktion einen frischen EntityManager verwenden, wird beim Laden der Daten für die

Detailansicht das MovieDTO noch einmal neu aus der DB gelesen mit den jetzt aktuellen Werten.

Beispielcode dazu:

```
public void insertUpdateMovie(MovieDTO movieDTO) {
   // Diese Methode entspricht einer Transaktion
   EntityManager em = Emf.getEM();
        // neuen em erzeugen oder existierenden zurückgeben (je nach Variante, s. oben)
   try {
      em.getTransaction().begin();  // und die Transaktion starten

      // ... hier kommt der eigentliche Code

      // Untermethoden erhalten em als Parameter und müssen sich nicht
      // um das Erzeugen oder Schliessen oder um die Transaktionssteuerung
      // kümmern:
      untermethode(em, ...);

      em.getTransaction().commit();   // erfolgreiches Ende
   } finally {
      // Dieser finally-Block kann auch in eine Hilfsmethode ausgelagert
      // werden, da er für alle Transaktionen gleich aussieht
      try {
         // wenn die Transaktion noch aktiv ist, gab es keinen Commit
         // d.h. es ist ein Fehler aufgetreten. Dann wird Rollback
         // aufgerufen
         if (em.getTransaction().isActive())
            em.getTransaction().rollback();
      } finally {
         // close im finally stellt sicher, dass auch bei Fehlern beim Rollback
         // der em geschlossen wird (nur bei einem em pro Transaktion nötig)
         em.close();
      }
   }
}
```

Die Anmerkungen bzgl. Exceptions zur ersten Bonusaufgabe gelten hier genauso.

Optimale Ausnutzung der Funktionen des EntityManager

Nutzen Sie die Funktionalitäten eines EntityManager und verzichten Sie auf unnötige manuelle Kontrolle der Persistenz der Objekte. So benötigen Sie bei Änderungen an den Objekten keine Anweisungen wie persist oder merge (evtl. abhängig von der Nutzung des EntityManager bei detached-Objekten, s. oben). Sie müssen lediglich die Attribute der Objekte korrekt belegen, den Rest erledigt der EntityManager für Sie. Explizite Zuordnungen benötigen Sie meist nur bei Neuanlegen (persist) sowie Löschen (remove) von Entitäten. Ebenso sollten Sie auf keinen Fall explizite JP-QL Statements zum Aktualisieren der Objekte nutzen. Bei so einfachen Entity-Modellen wie hier ist die Bearbeitung der entsprechenden Attribute (innerhalb der Kontrolle des EntityManagers, bspw. durch eine Transaktion) auf Java-Ebene absolut ausreichend. Alles Weitere erledigt das Framework automatisch. Das ist gerade einer der wesentlichen Mehrwerte der Nutzung eines solchen Frameworks.

Gehören die Annotationen an die Getter oder an die Attribute der Entities?

Die JPA-Annotationen wie z.B. @Id oder @OneToMany können sowohl an die Attribute als auch an die Getter-Methoden geschrieben werden. Beispiele für beide Versionen:

```
@Id @GeneratedValue
private Long id;

...
```

```
// keine Annotationen hier
public Long getId() {
   return id;
}
```

oder

```
// keine Annotationen hier
private Long id;

...

@Id @GeneratedValue
public Long getId() {
   return id;
}
```

Im Prinzip ist beides möglich. Der Unterschied ist, dass beim Einlesen des Objektes aus der Datenbank oder beim Speichern in die Datenbank JPA im ersten Fall direkt die Werte der Attribute liest bzw. schreibt. Im zweiten Fall hingegen greift JPA über die Getter- und Setter-Methoden auf die Attribute zu, wenn das Objekt aus der Datenbank eingelesen bzw. in die Datenbank geschrieben wird. Dies bedeutet, dass ggf. weitere Verarbeitungsschritte oder komplexere Logik, die in den Getter- oder Setter-Methoden enthalten sein könnte, für den Transfer zwischen Datenbank und Hauptspeicher relevant sind. Meistens ist es einfacher und klarer, die Attribute zu annotieren. Dann werden die Objekte immer 1:1 in die Datenbank geschrieben bzw. aus der Datenbank gelesen, und Logik innerhalb der Getter-Methoden findet nur bei Aufrufen innerhalb des Java-Programmes Anwendung.

Ein paar weitere Argumente dafür, die Attribute zu Annotieren, finden sich hier:
https://thorben-janssen.com/access-strategies-in-jpa-and-hibernate/

Abbildung von Beziehungen in den Entities

Eine Beziehung aus dem Klassenmodell (Assoziation) kann im Java-Code wahlweise nur auf einer Seite oder auf beiden Seiten als Attribut abgebildet werden. Als Beispiel sei hier die Beziehung zwischen Movie und Genre genannt. Hier gibt es also drei Realisierungsvarianten:

1. Nur ein Attribut "List<Genre> genres" in Movie
2. Nur ein Attribute "List<Movie> movies" in Genre
3. Beide Attribute, Verknüpfung der Attribute über "mappedBy" in den Annotationen.

Welche Lösung hier verwendet werden sollte, hängt davon ab, welche Attribute im Programmcode auch tatsächlich verwendet werden. Im Beispielprogramm wird häufig zu einem Movie die Liste seiner Genres benötigt. Die Liste aller Movies, die zu einem Genre gehören, wird aber in diesem Programm niemals benötigt. Daher wäre es hier sinnvoll, Variante 1 zu verwenden, da nur dieses Attribut auch tatsächlich benötigt wird. Falls doch einmal die Liste der Movies zu einem Genre benötigt würde, könnte diese Liste auch per JP-QL-Query ermittelt werden. Beachten Sie aber, dass dies eigentlich eine Schwäche in der Modellierung ist, denn für diesen Fall hätte man die Beziehung im Modell als gerichtete Beziehung von Movie zu Genre modellieren sollen. Insofern entspricht die Variante 3 genauer dem gegebenen Klassenmodell aus der Aufgabe. Bei Variante 3 müssen Sie unbedingt „mappedBy“ nutzen. Wird dies vergessen, wird für jede Richtung der Beziehung eine eigene (und damit eine redundante) Tabelle in der DB angelegt.

Eine Ausnahme sind 1:N-Beziehungen. Wenn man hier nur die @OneToMany-Seite in den Klassen einfügt, führt dies bei manchen JPA-Providern zu einem komplexeren Datenmodell, da dann die Beziehung nicht über einen Fremdschlüssel, sondern über eine Join-Tabelle umgesetzt wird.

Verwendung von Queries statt Navigation im Objektmodell?

Wenn ein Attribut wie eben beschrieben zur Verfügung steht, sollte es auch verwendet werden. D.h. wenn das Attribut "genres" in der Klasse Movie existiert, sollte die Liste der Genres zu diesem Movie über dieses Attribut abgefragt werden. Ein neuer JP-QL-Query, um die Liste Genres zu einem Movie zu erfragen, wäre in diesem Fall überflüssig und macht den Code komplexer und schlechter lesbar:

Negativ-Beispiel:

```
Movie movie = ...
List<Genre> genres =
   em.createQuery(
      "SELECT g FROM Movie m JOIN m.genres g WHERE m = :movie",
      Genre.class).
      setParameter("movie", movie).
      getResultList();
for (Genre genre : genres)
   System.out.println(genre.getGenre());
```

Bessere Lösung mit dem Attribut "genres":

```
Movie movie = ...
for (Genre genre : movie.getGenres())
   System.out.println(genre.getGenre());
```

Lazy oder Eager Loading?

Lazy Loading bedeutet, dass Objekte, die mit einem Objekt verbunden sind, nicht direkt beim Laden des Objekts mitgeladen werden. Bei Eager geschieht dies direkt. Eager Loading kann eine Vielzahl von Ladevorgängen provozieren, die ggf. unnötig sind und das Programm stark ausbremsen können. Bei Lazy Loading werden die abhängigen Objekte vom Entity-Manger automatisch nachgeladen, wenn Sie referenziert werden. Daher ist in den meisten Fällen Lazy Loading zu bevorzugen. Dabei ist allerdings zu beachten, dass das automatische Nachladen nicht mehr funktioniert, wenn das Objekt von dem EntityManager detached ist (z.B. weil der EntityManager geschlossen wurde). In diesem Fall muss vor dem Detachen das Nachladen des abhängigen Objekts forciert werden. Im Beispiel-Programm können alle Beziehungen mit Lazy Loading umgesetzt werden.

Aktualisierung der Charaktere zu einem Movie

Eine schwierige Stelle der Aufgabe ist das Aktualisieren der Charaktere zu einem Movie. Hier schlagen wir wie in der letzten Bonusaufgabe vor, keinen detaillierten Abgleich vorzunehmen, sondern die alten Charaktere eines Movies komplett zu löschen und wieder neu einzutragen. Dies gilt umso mehr, da unklar ist, anhand welcher Information ein Charakter zu identifizieren ist und darüber evtl. Änderungen erkannt werden könnten. Wenn Sie aber das Löschen der alten Charaktere vergessen und nur die neuen hinzufügen, gibt es nachher in der DB zu viele Charaktere (nämlich alle vorher existenten doppelt zzgl. der neuen).

Wir nehmen an, dass die Beziehung folgendermaßen annotiert ist:

```
// in Movie:
@OneToMany(mappedBy = "movie", cascade = CascadeType.ALL)
private Set<MovieCharacter> character = new HashSet<MovieCharacter>();

// in MovieCharacter:
@ManyToOne
private Movie movie;
```

Ein erster Versuch könnte also in etwa so aussehen:

```
movie.getCharacter().clear();
for (CharacterDTO cdto : cDTOs) {
   MovieCharacter mc = characterFromDTO(cdto);
   movie.getCharacter().add(mc);
   mc.setMovie(movie);
}
```

Dies funktioniert aber nicht ganz, da die alten MovieCharacter-Einträge nicht gelöscht werden. Woran liegt das? Es wird durch das Löschen der Kollektion "characters" in Movie nur die Beziehung zwischen dem Movie und dem MovieCharacter, nicht aber der MovieCharacter selbst gelöscht.
Zur Lösung gibt es unterschiedliche Ansätze. Entweder werden die alten MovieCharacter-Objekte explizit gelöscht (Objekt für Objekt oder alle zusammen über eine JP-QL DELETE-Query), oder es wird (unter Nutzung über das Material aus der Vorlesung hinausgehender Annotationseigenschaften) das Attribut "orphanRemoval" in der @OneToMany-Annotation verwendet:

```
// in Movie:
@OneToMany(mappedBy = "movie", cascade = CascadeType.ALL, orphanRemoval = true)
private Set<MovieCharacter> character = new HashSet<MovieCharacter>();
```

Wie wird die Reihenfolge der Charaktere gespeichert?

Wie in der ersten Bonusaufgabe wird die Reihenfolge der Charaktere in der Datenbank über eine Position gespeichert, wohingegen sie im Java-Code über die Reihenfolge einer Java-Liste dargestellt ist. Die Umsetzung muss beim Laden bzw. Speichern erfolgen. Dies kann entweder von Hand oder (unter Nutzung über das Material aus der Vorlesung hinausgehender Annotationen) mit JPA-Unterstützung erfolgen.
Die manuelle Lösung ist es, die Liste der Charaktere nach dem Einlesen im Speicher zu sortieren, und beim Schreiben das Positions-Attribut hochzuzählen.
Eine elegantere Variante ist die Verwendung der Annotation @OrderColumn. Diese bestimmt eine Spalte in der Tabelle, über die die Kollektion automatisch durch JPA sortiert wird. Die Spalte bzw. das Attribut muss dann auch in MovieCharacter nicht mehr aufgenommen werden. Die Position ist also nicht mehr Teil des Klassenmodells, sondern nur noch Teil des relationalen Modells.

```
// in Movie:
@OneToMany(mappedBy = "movie", cascade = CascadeType.ALL, orphanRemoval = true)
@OrderColumn(name="POS")
private List<MovieCharacter> mc = new ArrayList<MovieCharacter>();
```