Datenbanksysteme 2, 5. Übung
Hinweise zur Lösung

In diesem Dokument sind typische Fragestellungen und Beispiele für Fehler zusammengestellt, die bei der Bearbeitung der ersten 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:

Fehlende Ressourcenfreigabe

Ein typischer Fehler ist, dass Ressourcenfreigabe vergessen wird. Als Beispiel dient die folgende Factory-Methode, die keinerlei Ressourcenfreigabe beinhaltet.

Negativ-Beispiel ohne Ressourcenfreigabe:

```
public static List<Movie> findByTitle(String title) throws SQLException {
	String sql =
		"SELECT movieid, title, year, type FROM dbs2_movie WHERE upper(title) like ?";
	List<Movie> res = new ArrayList<>();
	PreparedStatement stmt = DBConnection.getConnection().prepareStatement(sql);
	stmt.setString(1, "%" + title.toUpperCase() + "%");
	ResultSet rs = stmt.executeQuery();
	while (rs.next()) {
		Movie movie =
			new Movie(
				rs.getLong("movieid"),
				rs.getString("title"),
				rs.getInt("year"),
				rs.getString("type"));
		res.add(movie);
	}
	return res;
}
```

Zunächst funktioniert diese Methode, bei vielfachen Aufrufen (längere Programmlaufzeit) tritt allerdings folgender Fehler auf:

```
java.sql.SQLException: ORA-01000: Maximale Anzahl offener Cursor überschritten
```

Um das zu vermeiden, sollte das AutoCloseable-Interface genutzt werden. Eine passende Korrektur sähe also folgendermaßen aus:

```
public static List<Movie> findByTitleOk(String title) throws SQLException {
	String sql =
		"SELECT movieid, title, year, type FROM dbs2_movie WHERE upper(title) like ?";
	List<Movie> res = new ArrayList<>();
	try (PreparedStatement stmt = DBConnection.getConnection().prepareStatement(sql)) {
		stmt.setString(1, "%" + title.toUpperCase() + "%");
		try (ResultSet rs = stmt.executeQuery()) {
			while (rs.next()) {
				Movie movie =
					new Movie(
						rs.getLong("movieid"),
						rs.getString("title"),
						rs.getInt("year"),
						rs.getString("type"));
					res.add(movie);
			}
		}
	}
	return res;
}
```

Keine Verwendung von PreparedStatements

Es sollte unbedingt vermieden werden, Parameter von SQL-Statements per String-Konkatenation in ein SQL-Statement einzubauen. Dies kann ggf. zu einer Sicherheitslücke (SQL Injection, siehe Kapitel 4 der Vorlesung) führen. Unabhängig von der Sicherheitslücke können in der falschen Lösung bestimmte Nutzereingaben zu syntaktisch falschem SQL führen, so dass eine Exception ausgelöst wird, die mit korrekter Programmierung vermeidbar wäre.

Negativ-Beispiel mit String-Konkatenation:

```
String sql =
        "SELECT movieid, title, year, type FROM dbs2_movie WHERE upper(title) like '%"
                + title.toUpperCase() + "%'";
Statement stmt = DBConnection.getConnection().createStatement();
try (ResultSet rs = stmt.executeQuery(sql)) {
        // ...
```

Hier würde z.B. die Sucheingabe `test'` zu folgender Fehlermeldung führen:

```
java.sql.SQLSyntaxErrorException: ORA-00911: Ungültiges Zeichen
```

Um das zu vermeiden, sollten Sie immer PreparedStatements verwenden. Die richtige Lösung sieht also folgendermaßen aus:

```
String sql =
        "SELECT movieid, title, year, type FROM dbs2_movie WHERE upper(title) like ?";
try (PreparedStatement stmt = DBConnection.getConnection().prepareStatement(sql)) {
        stmt.setString(1, "%" + title.toUpperCase() + "%");
        try (ResultSet rs = stmt.executeQuery()) {
                // ...
```

Unterdrücken von Exceptions

Mit Hilfe von Exceptions werden Fehler im Programm gemeldet und andere Programmteile können darauf reagieren. Daher sollten Exceptions niemals unterdrückt werden. Dies führt dazu, dass ein Fehler unbemerkt bleibt. Damit kann z.B. eine Transaktion, in deren Ablauf der Fehler entstanden ist, nicht korrekt auf den Fehler reagieren (d.h. ein Rollback auslösen).

Das Unterdrücken von Exceptions passiert besonders leicht, wenn die Entwicklungsumgebung (Eclipse z.B.) eine entsprechende Code-Anpassung automatisch anbietet.

Negativ-Beispiel für das Unterdrücken von Exceptions:

```
public void update() {
	// update an existing movie
	String sql = "UPDATE dbs2_movie SET type = ?, title = ?, year = ? WHERE movieid = ?";
	try (PreparedStatement stmt = DBConnection.getConnection().prepareStatement(sql)) {
		stmt.setString(1, type);
		stmt.setString(2, title);
		stmt.setInt(3, year);
		stmt.setLong(4, movieId);
		stmt.executeUpdate();
	} catch (SQLException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
}
```

Der Catch-Block wurde in diesem Beispiel durch Eclipse automatisch eingefügt, nachdem die entsprechende Nachfrage von Eclipse positiv beantwortet wurde. Der Code hat allerdings zur Folge, dass die aufrufende Methode einen Fehler nicht erkennt. Wenn also z.B. diese Methode durch `insertUpdateMovie` aufgerufen wird und das Update-Statement fehlschlägt, wird `insertUpdateMovie` die Transaktion trotzdem zu Ende führen und mit Commit bestätigen. In der GUI würde der Benutzer keine Rückmeldung bekommen und vermuten, dass seine Änderungen korrekt eingetragen sind. Tatsächlich sind aber die Änderungen in der Movie-Tabelle aufgrund des Fehlers nicht übernommen worden. Einziger Hinweis auf den Fehler wäre in diesem Fall die Konsolen-Ausgabe, die allerdings bei einer GUI vmtl. gar nicht wahrgenommen wird.

Eine passende Implementierung würde also in dieser Methode die Exception nicht abfangen, sondern an die aufrufende Methode weitergeben.

Positiv-Beispiel: Exception wird in ActiveRecord-Klasse nicht abgefangen:

```
public void update() throws SQLException {
	// update an existing movie
	String sql = "UPDATE dbs2_movie SET type = ?, title = ?, year = ? WHERE movieid = ?";
	try (PreparedStatement stmt = DBConnection.getConnection().prepareStatement(sql)) {
		stmt.setString(1, type);
		stmt.setString(2, title);
		stmt.setInt(3, year);
		stmt.setLong(4, movieId);
		stmt.executeUpdate();
	}
}
```

Keine Prüfung des Rückgabewerts von executeUpdate

Der Rückgabewert von `executeUpdate` ist die Anzahl der tatsächlich veränderten Datensätze. In vielen Fällen ist der erwartete Wert hierbei eins, da z.B. in den update-Methoden der ActiveRecord-Klassen jeweils nur ein Datensatz verändert wird. Daher ist es sinnvoll, zu prüfen, ob der Rückgabewert auch tatsächlich dem erwarteten Wert entspricht und anderenfalls eine Exception auszulösen. Dies bewirkt, dass ein entsprechender Fehler entdeckt werden kann und die umgebende Transaktion ein Rollback auslösen kann.

Positiv-Beispiel für die Prüfung des Rückgabewertes von `executeUpdate`:

```
public void update() throws SQLException {
	// update an existing movie
	String sql = "UPDATE dbs2_movie SET type = ?, title = ?, year = ? WHERE movieid = ?";
	try (PreparedStatement stmt = DBConnection.getConnection().prepareStatement(sql)) {
		stmt.setString(1, type);
		stmt.setString(2, title);
		stmt.setInt(3, year);
		stmt.setLong(4, movieId);
		if (stmt.executeUpdate() != 1) {
			throw new SQLException("Could not update movie with id " + movieId);
		}
	}
}
```

Transaktionssteuerung in ActiveRecord-Klassen

Die Transaktionssteuerung muss in den Manager-Klassen implementiert werden, da hier die logischen Einheiten zu finden sind, die atomar und isoliert ausgeführt werden sollen. Daher würde eine Transaktionssteuerung in den ActiveRecord-Klassen zu einem vorzeitigen Ende einer Transaktion führen. Als Beispiel sei hier wieder die `insertUpdateMovie`-Methode genannt. Diese führt mehrere Schritte aus, im Insert-Fall das Einfügen des Movie-Datensatzes, das Einfügen der Genres sowie das Einfügen der Charaktere. Wenn in diesem letzten Schritt ein Fehler auftritt (z.B. ist ein Schauspieler nicht vorhanden, der referenziert wurde), sollte die gesamte Änderung des Films rückgängig gemacht werden (durch Rollback). Damit wird vermieden, dass die Datenbank in einem undefinierten Zustand ist, in dem ein Teil des Films eingefügt ist, ein anderer Teil aber nicht. Wenn die insert-Methode der Movie-ActiveRecord-Klasse Methoden zur Transaktionssteuerung enthält, ist dies nicht mehr möglich.

Negativ-Beispiel für Transaktionssteuerung in den ActiveRecord-Klassen

```
public void insert() throws SQLException {
	Long id = getNewId();

	String sql = "INSERT INTO dbs2_movie(movieid, title, year, type) VALUES (?, ?, ?, ?)";
	boolean ok = false;
	try (PreparedStatement stmt = DBConnection.getConnection().prepareStatement(sql)) {
		stmt.setLong(1, id);
		stmt.setString(2, title);
		stmt.setInt(3, year);
		stmt.setString(4, type);
		if (stmt.executeUpdate() != 1) {
			throw new SQLException("Error during insert of new movie");
		}
		DBConnection.getConnection().commit();
		ok = true;
	} finally {
		if (!ok)
			DBConnection.getConnection().rollback();
	}

	this.movieId = id;
}
```

Richtig ist es hier, gar keine Transaktionssteuerung in den ActiveRecord-Klassen zu verwenden. Die obige Methode sollte also folgenermaßen aussehen:

```
public void insert() throws SQLException {
	Long id = getNewId();

	String sql = "INSERT INTO dbs2_movie(movieid, title, year, type) VALUES (?, ?, ?, ?)";
	try (PreparedStatement stmt = DBConnection.getConnection().prepareStatement(sql)) {
		stmt.setLong(1, id);
		stmt.setString(2, title);
		stmt.setInt(3, year);
		stmt.setString(4, type);
		if (stmt.executeUpdate() != 1) {
			throw new SQLException("Error during insert of new movie");
		}
	}

	this.movieId = id;
}
```

Keine oder falsche Transaktionssteuerung in den Manager-Klassen

Die Transaktionssteuerung muss in den Manager-Klassen eingebaut werden. Dabei ist zu beachten, dass jeweils die ganze Logik, die atomar ausgeführt wird, zu einer Transaktion wird. Dies soll hier am Beispiel der `insertUpdateMovie`-Methode verdeutlicht werden:

Positiv-Beispiel für die Transaktionssteuerung in `insertUpdateMovie`:

```
public void insertUpdateMovie(MovieDTO movieDTO) throws Exception {
	boolean ok = false;
	try {
		// Eintrag in DBS2_MOVIE eintragen oder aktualisieren
		// ...

		// Genres zu dem Movie eintragen oder aktualisieren
		// ...

		// Charaktere zu dem Movie eintragen oder aktualisieren
		// ...

		DBConnection.getConnection().commit();
		ok = true;
	} finally {
		if (!ok)
			DBConnection.getConnection().rollback();
	}
}
```

Wichtig ist dabei, dass im eigentlichen Hauptblock, in dem die Logik der Methode ausgeführt wird, keine weiteren Aufrufe von Commit und/oder Rollback enthalten sind (siehe vorheriger Punkt zum Thema Transaktionssteuerung in den ActiveRecord-Klassen).

Weiterhin sollten Sie sich angewöhnen, Transaktionssteuerung auch bei nur lesenden Transaktionen zu implementieren (also z.B. die `getMovie`-Methode). Dies verhindert, dass nach der Methode noch eine aktive Transaktion existiert, die dann mit der nächsten Transaktion zusammengelegt wird. Dies kann je nach verwendeter Datenbank und genauen Einstellungen in der Datenbank ggf. zu unnötigen Synchronisationsoperationen in der Datenbank führen, die ggf. das Programm verlangsamen oder zu einem sogenannten Dead-Lock führen (siehe Kapitel 7 der Vorlesung).