Objektorientierte Datenbanken

Vorlesung 10

Sebastian Iwanowski
FH Wedel

Wesentliche Eigenschaften von Hibernate

- **Transparente Persistenz**
- **Transitive Persistenz (Persistenz per Erreichbarkeit)**
- **Detached Object Support**
- **Inheritence mapping strategies**
- **Intelligent fetching and caching**
- **Automatic dirty object checking**
- **Unterschiedliche Anfragekonzepte: Queries und Criteria**

Transparente Persistenz

- **Jede Klasse darf persistent sein.**
  Es gibt keine Anforderungen an Superklassen, von denen die persistente Klasse erben muss oder an Interfaces, welche sie implementieren muss.

  JDO: einige Systemklassen sind verboten, alle persistenten Klassen müssen serialisierbar sein

- **Persistente Klassen dürfen auch in anderen Umgebungen benutzt werden, z.B. in Testumgebungen.**

  JDO: nur wenn sie explizit detached werden!

- **Persistente Objekte haben keine für den Anwender sichtbare Zustände.**
  Sie benötigen in der Anwendungsschicht keine besonderen Behandlungen.
  Alle persistenzspezifischen Aspekte werden in den Interfaces `Session` oder `Query` verwaltet.

  JDO: umfangreiches Lebenszyklusmanagement, nicht transparent

Transparente Persistenz

Fazit:

Der Anwendungsprogrammierer kann Hibernate als Black Box betrachten: Der konkrete Persistenzmechanismus wird verborgen.

JDO:

Der Anwendungsprogrammierer kann nicht alles mit persistenten Objekten machen, was mit Java-Objekten erlaubt ist

Offene Frage:

Welchen Preis bezahlt man bei Hibernate ?

Anm. 1: Die JDO-Einschränkungen sind nicht sehr groß, es ist mehr eine Frage des Komforts!

Anm. 2: Auch in Hibernate ist nicht alles möglich, siehe Kap. 6.2 (S. 209) POJO-Buch

Transitive Persistenz

Verallgemeinerung der Persistenz durch Erreichbarkeit:

Für alle Felder können individuell folgende Optionen eingestellt werden (in den XML-Metadaten):

none • **Feld soll überhaupt nicht in der Datenbank verändert werden, wenn eine Datenbankänderung im referenzierenden Objekt vorgenommen wird.**

save-update • **Feld soll genau dann in der Datenbank aktualisiert werden, wenn eine Aktualisierung im referenzierenden Objekt vorgenommen wird.**

delete • **Feld soll genau dann aus der Datenbank gelöscht werden, wenn eine Löschung des referenzierenden Objekts vorgenommen wird.**

all • **Feld soll genau dann in der Datenbank verändert werden, wenn eine Datenbankänderung im referenzierenden Objekt vorgenommen wird.**

all-delete-orphan

delete-orphan

Transitive Persistenz im OR-Mapping bei JDO

Verallgemeinerung der Persistenz durch Erreichbarkeit:

- **First class: eigene Tabelle** — *entity parameter*
- **Second class: Spalte in fremder Tabelle** — *value parameter*

JDO definiert action-Attribute für den Zugriff auf Tabellenzeilen:

none
- **legt keinen Fremdschlüssel für Tabellenzeile an:**
  Aktualisierung dieser Zeile hat keine Auswirkungen auf andere Objekte.

cascade
- **legt Fremdschlüssel für Tabellenzeile an:**
  Beim Löschen dieser Zeile werden alle Zeilen gelöscht, die auf diese zeigen.

default
- **Löschen richtet sich nach defaultmäßiger JDO-Spezifikation**

weitere Attribute möglich

Offene Frage:

Hängt das mit den entity associations von Hibernate zusammen ?

Detached Object Support

Vorteile von Hibernate (POJOs) gegenüber EJB 2:

Vorteile gelten auch für JDO-2, nicht für JDO-1

For applications using servlets + session beans:

- You don't need to `select` a row when you only want to `update` it!
- You don't need DTOs anymore!
- You may serialize objects to the web tier, then serialize them back to the EJB tier in the next request
- Hibernate lets you *selectively* reassociate a subgraph! (essential for performance)

Das geht wohl auch nicht in JDO-2

aus Gavin King: *Object / Relational Mapping with Hibernate*

Detached Object Support

Step 1: **Retrieve some objects in a session bean:**

```
session
    .createQuery("from Student student where student.semester > 1 ")
    .list();
```

Step 2: **Collect user input in a servlet / action:**

```
student.setSemester(student.getSemester()-1);
```

Step 3: **Make the changes persistent, back in the session bean:**

```
session.update(student);
```

nach Gavin King: *Object / Relational Mapping with Hibernate*

Detached Object Support

Even transitive persistence!

```
Session session = sf.openSession();
Transaction tx = session.beginTransaction();
Student student =
   (Student) session.get(Student.class, itemId);
tx.commit();
session.close();

Prüfung prüfung = student.getPrüfungen.getFirst();
prüfung.setNote(1);

Session session2 = sf.openSession();
Transaction tx = session2.beginTransaction();
session2.update(student);
tx.commit();
session2.close();
```

JDO: explicit detach

JDO: makePersistent (student)

nach Gavin King: *Object / Relational Mapping with Hibernate*

Detached Object Support

The Big Problem only for Hibernate?

Detached objects + Transitive persistence:

- How do we distinguish between newly instantiated objects and detached objects that are already persistent in the database?

Solution (Hibernate)

- Version property (if there is one)
- Identifier value e.g. `unsaved-value="0"` (only works for generated surrogate keys, not for natural keys in legacy data)
- Write your own strategy, implement `Interceptor.isUnsaved()`

Solution (JDO)

- via *detached* state (still enhanced information: ObjectId)

aus Gavin King: *Object / Relational Mapping with Hibernate*

Inheritance mapping strategies

Hibernate ermöglicht es, zwischen folgenden Abbildungsschemata für die Vererbung zu wählen (einstellbar für jede Klasse in den Metadaten):

- **Table per concrete class**
  Die Klasse und all ihre Unterklassen bekommen jeweils eine eigene Tabelle.

- **Table per class hierarchy**
  Es gibt eine einzige Tabelle für die gesamte Klassenhierarchie.
  Die Zugehörigkeit zu einer bestimmten Klasse wird in einer separaten Spalte abgespeichert.

- **Table per subclass**
  Es gibt gesonderte Tabellen für die Unterklassen.
  Dort werden aber nur die Felder abgespeichert, die nicht geerbt werden.

Einmal eingestellt, gilt das Abbildungsschema für die gesamte Hierarchie!

Inheritance mapping strategies in JDO

In JDO wird das Abbildungsschema für jede Klasse getrennt festgelegt: Sie gilt nicht in der gesamten Hierarchie.

- **new-table**
  Die Klasse bekommt eigene Spalten für alle in ihr definierten Felder.
- **superclass-table**
  Die Felder dieser Klasse werden als Spalten in der Superklassentabelle angelegt.
- **subclass-table**
  Die Felder dieser Klasse werden als Spalten in der Subklasse noch einmal angelegt.

Realisierung der Hibernate-Schemata:

- **Table per concrete class**
  subclass-table for class; new-table for all subclasses
- **Table per class hierarchy**
  new-table for highest class in hierarchy; superclass-table for all other classes
- **Table per subclass**
  new-table for all classes

In JDO ist die Realisierung weiterer Schemata möglich!

Intelligent fetching and caching

Fetching strategies:

- **Immediate**
  Alles im cache wird sofort mit der Datenbank aktualisiert.

  JDO: über fetch groups

- **Lazy**
  Aktualisiert wird erst beim ersten Zugriff.

  JDO: Zustand hollow

- **Eager (Outer Join)**
  Aktualisiert werden alle Daten, die zu einem Objekt gehören, das aktualisiert wird (Klassen und Navigationstiefe einstellbar).

  JDO: über fetch groups

- **Batch**
  wie eager, aber erst nach erstem Zugriff
  und auch nur bis zu einer einstellbaren Anzahl von Objekten

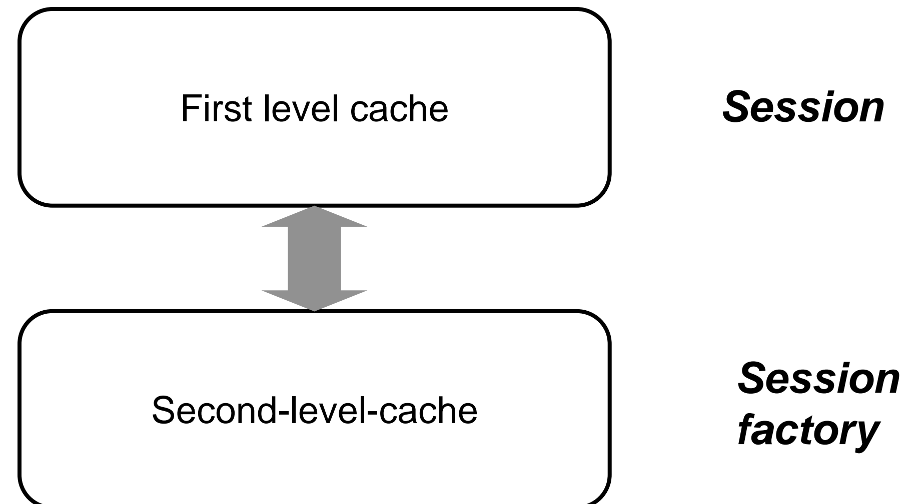
Intelligent fetching and caching

Caching:

Aufgaben:

- Verringerung der Häufigkeit des tatsächlichen Zugriffs auf die Datenbank
- Arbeiten mit identischen Objekten für gleiche Werte

Hibernate-Cache-Architektur:



First Level cache (Query cache):

- obligatorisch
- nicht konfigurierbar

Second Level cache:

- optional
- vielfältig konfigurierbar: Art der Objekte, Vorhaltungszeit, ...

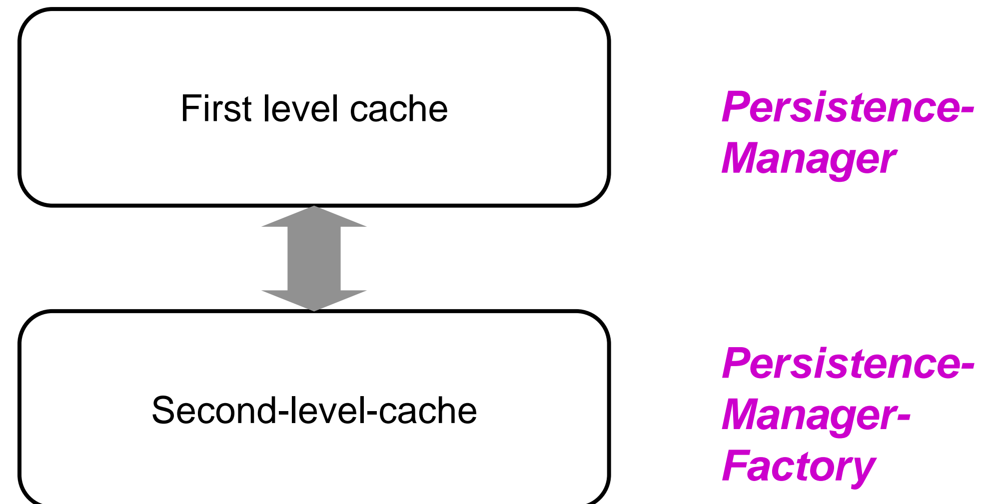
Intelligent fetching and caching

Caching in JDO-2:

Aufgaben:

- **Verringerung der Häufigkeit des tatsächlichen Zugriffs auf die Datenbank**
- **Arbeiten mit identischen Objekten für gleiche Werte**

JDO-2-Cache-Architektur:



First Level cache:

- **wird nach jeder Transaktion geleert**

Second Level cache:

- **Nur für optimistische Transaktionen**

Automatic Dirty Object Checking

- **Neue Werte für persistente Felder werden automatisch in die Datenbank geschrieben.**
  Im Prinzip wie bei JDO, aber mit SQL-update-Mechanismus für Wahrung der Objektidentität.

- **Unnötige Aktualisierungen der Datenbank werden vermieden.**
  Bei Zuweisung eines neuen Objekts wird zuerst nachgesehen, ob es tatsächlich unterschiedliche Werte enthält, bevor eine Aktualisierung gemacht wird.

nicht ganz durchgängig !

in JDO: produktspezifisch