Objektorientierte Datenbanken

Vorlesung 9
Sebastian Iwanowski
FH Wedel

Inhalt heute: JDO – letzter Teil

Vertiefung vom letzten Mal: Lebenszykluszustände

Optionale Lebenszykluszustände
(mit FastObjects-Implementierungen)

Zusammenfassung: JDO

Vertiefung vom letzten Mal: Lebenszykluszustände

Lebenszykluszustände eines Objekts

Obligatorische Zustände:

- **transient**
- **persistent-new**
- **hollow**
- **persistent-clean**
- **persistent-dirty**
- **persistent-deleted**
- **persistent-new-deleted**

Optionale Zustände: → **☺ gibt es alle in FastObjects ☺**

aber mit einigen Besonderheiten

- **transient-clean**
- **transient-dirty**
- **persistent-nontransactional**

Lebenszykluszustände eines Objekts

Welche Auswirkungen haben die Zustände ?

Transiente Objekte:

- **keine Verbindung zu irgendeinem Datenbankobjekt**
- **`makePersistent` erzeugt immer neues Datenbankobjekt**

„hohle“ (hollow) Objekte:

- **Verbindung zu konkretem Datenbankobjekt**
- **keine Speicherbelegung für Datenfelder (aus Effizienzgründen)**
- **keine feste Bindung vom PersistenceManager mehr**

Konsequenz ? **(könnte durch Garbage Collector aufgeräumt werden!)**

- **Außerhalb einer Transaktion sind die Datenfelder nicht zugänglich !**

Lebenszykluszustände eines Objekts

Erreichen des „Spezialzustands“ `hollow`:

- **durch Beendigung einer Transaktion (gilt für alle Objekte des PersistenceManagers)**
- **durch direkten Objektzugriff aus Datenbank (über ObjektId bzw. Objektnamen)**
- **durch Iterieren eines Extents**
- **durch Ergebnis einer Query**
- **durch Navigieren von anderem persistenten Objekt aus**

➔ `hollow` ist der „Normalzustand“ eines Objekts !

Lebenszykluszustände eines Objekts

Kann man die Zustände abfragen ?

Methoden von JDOHelper:

	transient	hollow	p.-clean	p.-dirty	p.-new	p.-del.	p.-new-del.
`isPersistent(Object)`	*F*	*T*	*T*	*T*	*T*	*T*	*T*
`isTransactional(Object)`	*F*	*F*	*T*	*T*	*T*	*T*	*T*
`isDirty(Object)`	*F*	*F*	*F*	*T*	*T*	*T*	*T*
`isNew(Object)`	*F*	*F*	*F*	*F*	*T*	*F*	*T*
`isDeleted(Object)`	*F*	*F*	*F*	*F*	*F*	*T*	*T*

für FastObjects:

`PersistenceManagers.isHollow (PersistenceManager, Object)`

Optionale Lebenszykluszustände

☺ wird von FastObjects unterstützt ☺

Wiederholung: Persistenzkonzept bei Attributen

Drei Typen für Attribute persistenter Objekte:

- **persistent**

  normale Persistenz durch Erreichbarkeit, **`rollback()`** bei Transaktionen wirksam

- **transactional**

  Daten sind immer transient, aber **`rollback()`** bei Transaktionen wirksam

- **none**

  Daten sind immer transient, **`rollback()`** bei Transaktionen setzt sie nicht zurück

Analogon: Lebenszykluszustände der Objekte

Drei Grundtypen für Objekte persistenzfähiger Klassen:

Lebenszykluszustände:

- **persistent**
  - `persistent-clean`
  - `persistent-dirty`

  Objekte sind in der Datenbank und in der Cacheverwaltung des PersistenceManagers, **`rollback()`** bei Transaktionen wirksam

  FastObjects-Besonderheiten

- **transactional**
  - `transient-clean`
  - `transient-dirty`

  Objekte sind nicht in der Datenbank, aber in der Cacheverwaltung des PersistenceManagers, **`rollback()`** bei Transaktionen wirksam

  ☺ auch in FastObjects ☺
  aber mit Besonderheiten

- **transient**
  - `transient`

  Objekte sind weder in der Datenbank noch in der Cacheverwaltung des PersistenceManagers, **`rollback()`** bei Transaktionen unwirksam

  Defaultzustand

Analogon: Lebenszykluszustände der Objekte

Drei Grundtypen für Objekte persistenzfähiger Klassen:

- **persistent**

  Objekte sind in der Datenbank und in der Cacheverwaltung des PersistenceManagers, **`rollback()`** bei Transaktionen wirksam

- **transactional**

  Objekte sind nicht in der Datenbank, aber in der Cacheverwaltung des PersistenceManagers, **`rollback()`** bei Transaktionen wirksam

- **transient**

  Objekte sind weder in der Datenbank noch in der Cacheverwaltung des PersistenceManagers, **`rollback()`** bei Transaktionen unwirksam

Methoden von `PersistenceManager`:

`makePersistent(obj)`

`makeTransactional(obj)`
☺ auch in FastObjects ☺

`makeNontransactional(obj)`
☺ auch in FastObjects ☺
aber mit Besonderheiten

`makeTransient(obj)`

Wiederholung: Transaktionskonzept

3 Transaktionsstrategien:

- **Normale (pessimistische) Transaktionen**

  Bei Zugriff wird eine Sperre auf das betreffende Datenbankobjekt gelegt.

  Erst bei Transaktionsende wird die Sperre aufgehoben.

- **Optimistische Transaktionen**

  Es gibt während der Transaktion die meiste Zeit keine Sperre.

  Es wird vor dem Commit nachgeprüft, ob die benutzten Daten sich während der Transaktion geändert haben (dabei gibt es eine Sperre).

  Falls sich die Daten während der Transaktion geändert haben,
  erfolgt eine Nachricht an die Transaktion als Exception und kein Commit.

- **Datenbankzugriff außerhalb von Transaktionen**

  für Objekte in bestimmten Zuständen (wird jetzt besprochen)

Datenbankzugriff außerhalb von Transaktionen

Die in einer Transaktionen zugegriffenen Daten können von dieser Transaktion aus in folgende Zustände gebracht werden:

- **nontransactionalRead**

  zum Lesen von Feldern außerhalb von Transaktionen

- **nontransactionalWrite**

  zum Beschreiben von Feldern außerhalb von Transaktionen

Methoden von `Transaction`:

`setNontransactionalRead(bool)`

☹ **nicht in FastObjects** ☹

`setRetainValues(bool)`

behält alle Werte im Cache des PersistenceManagers nach Ende der Transaktion

☺ **auch in FastObjects** ☺

`setRestoreValues(bool)`

stellt alte Werte im Cache des PersistenceManagers her nach rollback der Transaktion

☺ **auch in FastObjects** ☺

`setNontransactionalWrite(bool)`

☹ **nicht in FastObjects** ☹

Datenbankzugriff außerhalb von Transaktionen

Zugriffe sind für alle Daten möglich, die sich noch im Cache vom `PersistenceManager` befinden:

- **transaktionale Objekte**

  nicht persistente Objekte unter der Kontrolle des Persistenzmanagers

- **persistente Objekte**

  Problem:
  Diese Objekte wechseln im Normalfall nach Ende einer Transaktion in den Zustand **`hollow`** und fallen aus der Kontrolle des Persistenzmanagers

  Lösung:
  Für nichttransaktionalen Zugriff wechseln diese Objekte in den neuen Zustand **`persistent-nontransactional`**

  Anmerkung:
  Im Zustand **`persistent-nontransactional`** befinden sich auch die persistenten Objekte von optimistischen Transaktionen, bevor **`commit`** gemacht wird.

Optionale Lebenszykluszustände eines Objekts

Abfragemöglichkeiten für die optionalen Zustände:

Methoden von JDOHelper:	transient	hollow	p.-nontransactional	t.-clean	t.-dirty
`isPersistent(Object)`	*F*	*T*	*T*	*F*	*F*
`isTransactional(Object)`	*F*	*F*	*F*	*T*	*T*
`isDirty(Object)`	*F*	*F*	*F*	*F*	*T*
`isNew(Object)`	*F*	*F*	*F*	*F*	*F*
`isDeleted(Object)`	*F*	*F*	*F*	*F*	*F*

für FastObjects:

```
PersistenceManagers.isHollow (PersistenceManager, Object)
PersistenceManagers.isPersistentNonTransactional
(PersistenceManager, Object)
```

Praktische Erfahrungen mit FastObjects

- **Lesezugriff außerhalb von Transaktionen:**

  nur nach **`setRetainValues(true)`** oder **`setRestoreValues(true)`** erlaubt

- **Schreibzugriff außerhalb von Transaktionen:**

  nicht zulässig

Praktische Erfahrungen mit FastObjects

- **transaktionale Objekte:**

  Transaktionale Objekte wechseln nach **`commit`** der Transaktion, in der sie entstanden sind, in den Zustand **`transient-clean`**.
  Wenn sie danach in einer Transaktion geändert werden, also in den Zustand **`transient-dirty`** wechseln, so darf diese Transaktion nur mit **`rollback`** abgeschlossen werden, d.h. **`commit`** ist nicht zulässig.
  Im Lesezugriff wird also stets der Wert gelesen, der in der ersten oder in der gegenwärtigen Transaktion gesetzt wurde.

  Ein Wechsel in den Zustand transient mit **`makeNontransactional`** ist im Zustand transient-dirty nicht zulässig.

  nach Auskunft von Versant Europa ein FastObjects-Bug

Praktische Erfahrungen mit FastObjects

- **persistente Objekte:**

  Persistente Objekte wechseln nach **`commit`** der Transaktion, in der auf sie zugegriffen wurde und in der **`setRetainValues (true)`** oder **`setRestoreValues (true)`** ausgeführt wurde, in den Zustand **`persistent-nontransactional`** (nicht **`hollow`**!). Wenn sie danach in einer Transaktion geändert werden, die mit **`rollback`** abgeschlossen wird, dann behalten sie im Cache dennoch die neuen Werte.

  Ein Wechsel in den Zustand persistent-nontransactional mit **`makeNontransactional`** ist im Zustand persistent-dirty nicht zulässig.

  Ein Wechsel in den Zustand persistent-nontransactional mit **`makeNontransactional`** ist im Zustand persistent-clean zwar zulässig, hat aber keine Auswirkungen: Innerhalb von Transaktionen wird bei Feldänderungen automatisch in den Zustand persistent-dirty zurückgekehrt.

  Außerhalb von Transaktionen ist Lesezugriff nur erlaubt, wenn **`setRetainValues(true)`** oder **`setRestoreValues (true)`** ausgeführt wurde. Anderenfalls geht Objekt nach Beendigung der Transaktion in den Zustand **`hollow`**, auch wenn die PersistenceManagers-Methoden etwas anderes anzeigen !

Zusammenfassung: JDO

Zusammenfassung: JDO

Vorlesung 4: Der `PersistenceManager` als Dreh- und Angelpunkt

alle Lese- und Schreibzugriffe, Transaktionsmanagement

Vorlesung 5: Persistenz- und Transaktionskonzept

Attributmanagement via XML-Metadaten, transiente / transaktionale und persistente Attribute, First-Class- und Second-Class-Objekte

Transaktionseigenschaften, Isolationsniveaus, Transaktionsstrategien (pessimistisch und optimistisch)

Vorlesung 6 / 7: JDOQL

Filter in Java-Syntax, Parameter- und Variablendeklarationen, Anfragen mit collection-wertigen Attributen, Vergleich zu OQL

Zusammenfassung: JDO

Vorlesung 8: Datenidentitätskonzept und Lebenszykluszustände (Anfang)

ID-Klassen, Datastore- und Application-Identity,
verschiedene Zugriffsmöglichkeiten auf ein Datenbankobjekt

Vorlesung 9: Lebenszykluszustände (Vertiefung)

obligatorische Lebenszykluszustände, Wirkung und Zweck (besonders **`hollow`**),
Zustandsabfrage

transaktionale Objekte, nichttransaktionaler Datenzugriff

Beim nächsten Mal:

Hibernate

Konzepte im Vergleich: Vorlesung
Code: Übung