Objektorientierte Datenbanken

Vorlesung 8 (Lebenszykluszustände)

Sebastian Iwanowski
FH Wedel

Lebenszykluszustände eines Objekts

- **Jedes Objekt einer persistenzfähigen Klasse hat Zustand, der die Persistenz„geschichte“ beschreibt**
- **Die Zustände werden intern als flags abgespeichert**
- **Zustände werden durch bestimmte Aktionen implizit gesetzt**
- **Einige Zustände können abgefragt oder explizit gesetzt werden**
- **Gegenwärtige Zustände beeinflussen die Ausführung anderer Aktionen**

Lebenszykluszustände eines Objekts

Obligatorische Zustände:

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

- **detached-clean**
- **detached-dirty**

} **nur in JDO 2.0**

Lebenszykluszustände eines Objekts

Zusammenhang zwischen den obligatorischen Zuständen (JDO 1.0.1) :

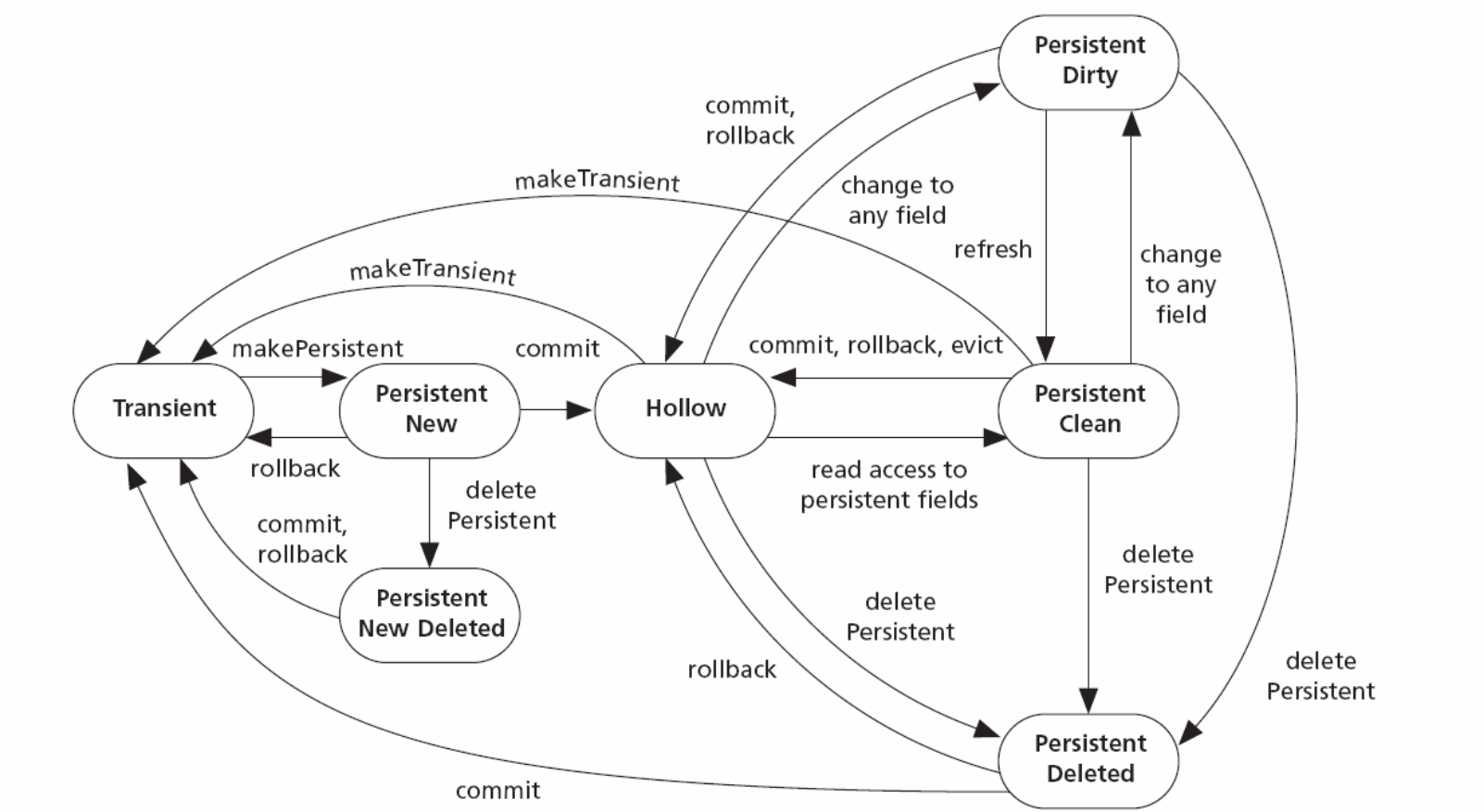


Figure 4.9 All required state transitions

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**

(könnte durch Garbage Collector aufgeräumt werden!)

Konsequenz ?

- **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 !

nicht in bei der Defaulteinstellung von Versant !!

Lebenszykluszustände eines Objekts

Kann man die Zustände abfragen ?

Zustände von JDO 1.0.1:

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*
`isDetached(Object)`	*F*	*F*	*F*	*F*	*F*	*F*	*F*

Lebenszykluszustände eines Objekts

Detach / Attach:

In JDO 1.0.1 existierendes Problem:

Wenn Objekt an einer Stelle geändert werden soll, die keine Verbindung zum Persistenzmanager hat, dann muss es kopiert werden, geändert und die Änderung in das Originalobjekt übertragen werden.

Anwendungsbeispiel: Server hat Verbindung zu Datenbank, Client führt Änderung durch.

PersistenceManager ist nicht serialisierbar!

Methoden von PersistenceManager:

- `Object detachCopy (Object obj)`
- `Object attachCopy (Object obj, boolean makeTransactional)`

Vorteile:

mehr Anwendungskomfort

Lebenszykluszustände eines Objekts

Detach / Attach:

3 Möglichkeiten, ein detach vorzunehmen:

- **explizit über Methoden des PersistenceManagers**
- **Automatisch über PM-Methode setDetachAllOnCommit**
- **implizit über Serialisierung eines Objekts**

Lebenszykluszustände eines Objekts

Zustände von JDO 2.0:

Methoden von JDOHelper:	detached-clean	detached-dirty
`isPersistent(Object)`	*F*	*F*
`isTransactional(Object)`	*F*	*F*
`isDirty(Object)`	*F*	*T*
`isNew(Object)`	*F*	*F*
`isDeleted(Object)`	*F*	*F*
`isDetached(Object)`	*T*	*T*

Optionale Lebenszykluszustände

☺ wird von Versant größtenteils unterstützt ☺

Lebenszykluszustände eines Objekts

Optionale Zustände:

- **transient-clean**
- **transient-dirty**
- **persistent-nontransactional**
- **persistent-nontransactional-dirty** **nur in JDO 2.0, nicht in Versant**

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

- **transactional** → `transient-clean`, `transient-dirty`

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

- **transient** → `transient`

  Defaultzustand

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

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)`

`makeNontransactional(obj)`

für den Zugriff außerhalb von Transaktionen: wechselt in Zustand **`transient`**

`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)

Zugriff auf persistente Objekte außerhalb von Transaktionen

Den in einer Transaktionen zugegriffenen Objekten können von dieser Transaktion aus folgende Eigenschaften (properties) zugewiesen werden:

- **nontransactionalRead**

  zum Lesen von Feldern außerhalb von Transaktionen

- **nontransactionalWrite**

  zum Beschreiben von Feldern außerhalb von Transaktionen

Methoden von `Transaction`:

`setNontransactionalRead(bool)`

`setRetainValues(bool)`

behält alle Werte im Cache des PersistenceManagers nach Ende der Transaktion

`setRestoreValues(bool)`

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

`setNontransactionalWrite(bool)`

☹ nicht in Versant ☹

Zugriff auf persistente Objekte außerhalb von Transaktionen

Zugriffe außerhalb von Transaktion sind für folgende Objekte möglich:

- **transiente Objekte** → `transient`

  nicht persistente Objekte außerhalb der Kontrolle des Persistenzmanagers

- **detached Objekte** → `detached-clean, detached-dirty`

  persistente Objekte außerhalb der Kontrolle des Persistenzmanagers

- **transaktionale Objekte** → `transient`

  nicht persistente Objekte unter der Kontrolle des Persistenzmanagers
  geht erst nach Ausführung der PM-Methode **`makeNontransactional`**

- **persistente Objekte** → `persistent-nontransactional`

  persistente Objekte unter der Kontrolle des Persistenzmanagers

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`**

Zugriff auf persistente Objekte außerhalb von Transaktionen

Anmerkungen zum Zustand `persistent-nontransactional` :

- **makeNontransactional** **! Besonderheiten !**

  Der Zustand **`persistent-nontransactional`** wird auch durch Anwendung der PM-Methode **`makeNontransactional`** auf persistente Objekte erreicht

- **Optimistische Transaktionen**

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

- **Versant-Defaults**

  Persistente Objekte bei Versant wechseln außerhalb von Transaktionen bei Lesezugriffen per default in den Zustand **`persistent-nontransactional`**.

- **nontransactionalWrite** **☹ bei Versant nicht möglich ☹**

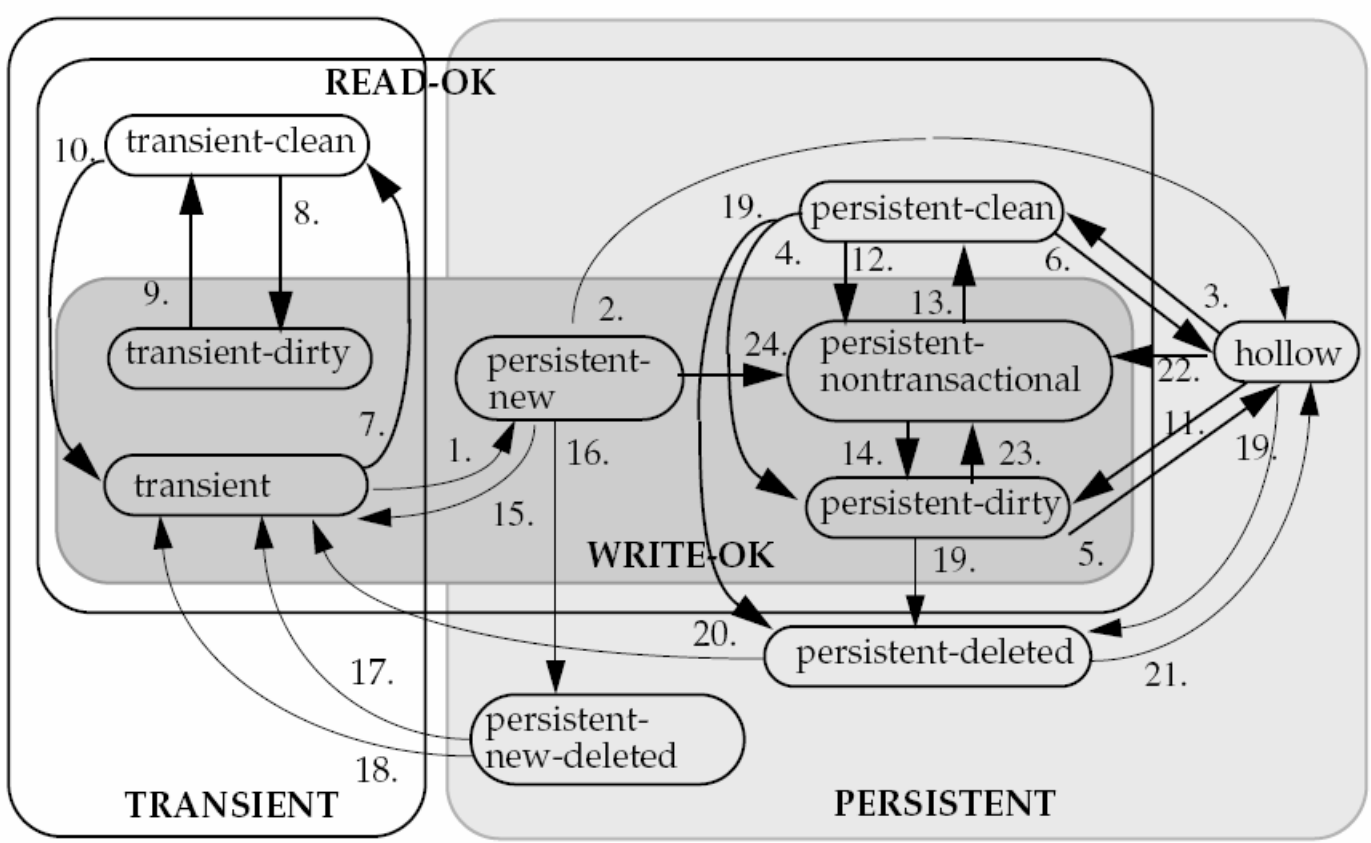
  Objekte des Zustands **`persistent-nontransactional`** wechseln bei Schreibzugriffen in den Zustand **`persistent-nontransactional-dirty`**.

Optionale Lebenszykluszustände eines Objekts

Abfragemöglichkeiten für die optionalen Zustände:

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

Zusammenfassung: Lebenszykluszustände



(aus Spec. 2.0, Fig. 14.0, S. 66, final release)

Was fehlt ?