

Java Data Objects

© Dr. Arno Schmidhauser
Letzte Revision: Juni 2004
Email: arno.schmidhauser@sws.bfh.ch
Webseite: <http://www.sws.bfh.ch/db>

Dieses Skript stützt sich wesentlich auf die Spezifikation
Java Data Objects, Version 1.0.1, 31. Mai 2003

Inhalt

I

Übersicht

II

Arbeiten mit JDO

III

Persistenzmodell

IV

O-R Mapping

V

Transaktionsmanagement

VI

Umfeld

VII

OO-Datenbanken

I Übersicht

Was ist JDO

1. Eine Allzweck-Architektur um Java-Objekte in einer Datenbank abzulegen
 1. unabhängig vom Datenmodell (relational, objekt-orientiert, ...)
 2. managed oder non-managed Gebrauch möglich (Applikationsserver oder Standalone)
2. Durch verschiedenste Interessengruppen abgeseignete Spezifikation.

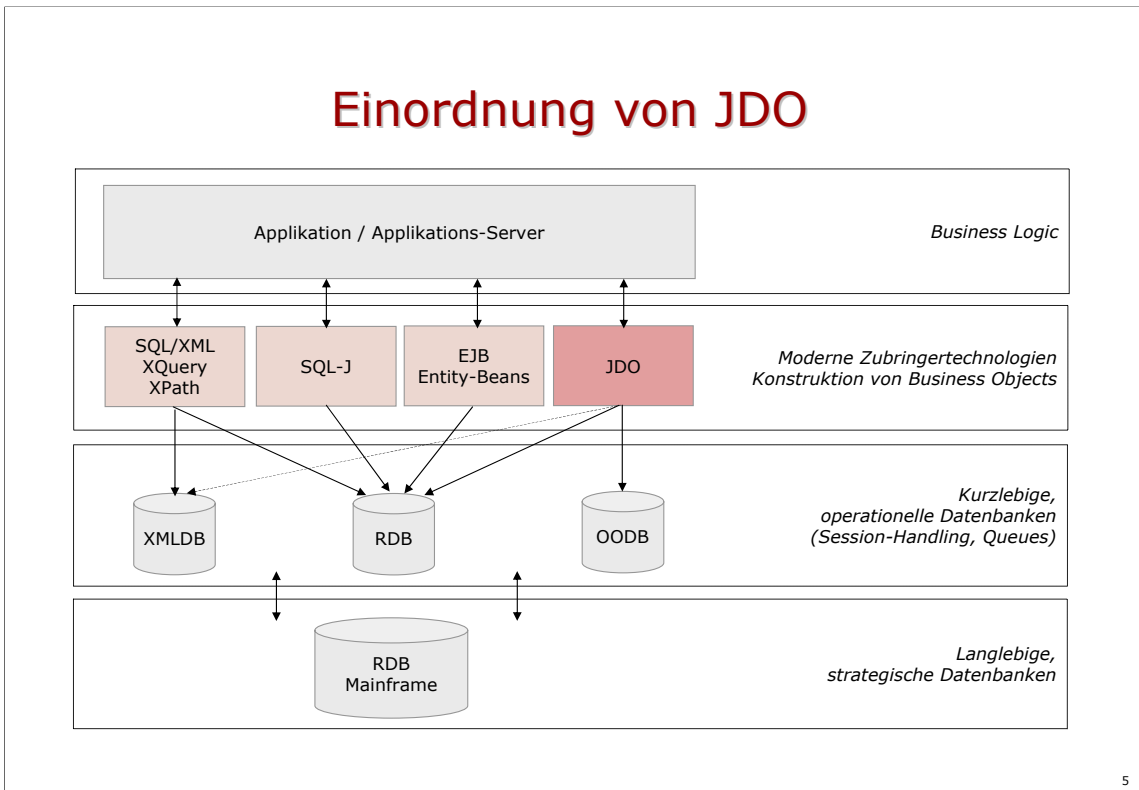
3

1. Mit JDO wird eine streng objektorientierte und javabasierte Sicht beibehalten beim Arbeiten mit einer Datenbank. Enthält eine Klasse Komponentenobjekte, sollen diese java-like angesprochen werden können. Jede Java-Klasse ist ohne Änderungen speicherbar (Basistypen, Arrays, Collection-Klassen, Benutzer-Klassen). Es gibt keine zusätzlichen Datentypen (Wie zum Beispiel DCollection im ODMG-Standard). Eine Beziehungen von der Klasse A zur Klasse B ist gegeben, indem Klasse A ein Objekt von B als Member enthält, oder indem A einen Vektor von Objekten der Klasse B enthält. Es sollen keine zusätzlichen Deklarationen notwendig sein, wie zum Beispiel Relationships in EJB notwendig sein.
 2. Bei der Spezifikation haben verschiedenste Personen und Hersteller mitgewirkt, u.a. IBM, Informix, Objectivity, Oracle, SAP, Sun, Software AG, Versant.

JDO Anwendungskonzept

1. Typsystem von Java, keine speziellen Datentypen für Persistenz -> Arrays, Collections, Utility-Klassen, Benutzer-Klassen.
2. Modifikation von Objekten mit Java-Befehlen (Zuweisung), keine speziellen Operatoren wie in SQL.
3. Von der Applikation kontrolliert werden lediglich der Lebenszyklus von Objekten (persistent, transient) und die Transaktionsgrenzen (begin, rollback, commit).
4. Zugriff auf Objekte in der Datenbank via Extent einer Klasse, via bekannte Objekt-ID, via Abfragesprache.

Einordnung von JDO



II Arbeiten mit JDO

JDO API

Packages

`javax.jdo`

`javax.jdo.spi`

Interfaces

`PersistenceManagerFactory`

`PersistenceManager`

`Transaction`

`Extent`

`Query`

`InstanceCallbacks`

Runtime Exceptions

`JDOException`

`JDOCanRetryException`

`JDODataStoreException`

`JDOUserException`

`JDOFatalException`

`...`

Classes

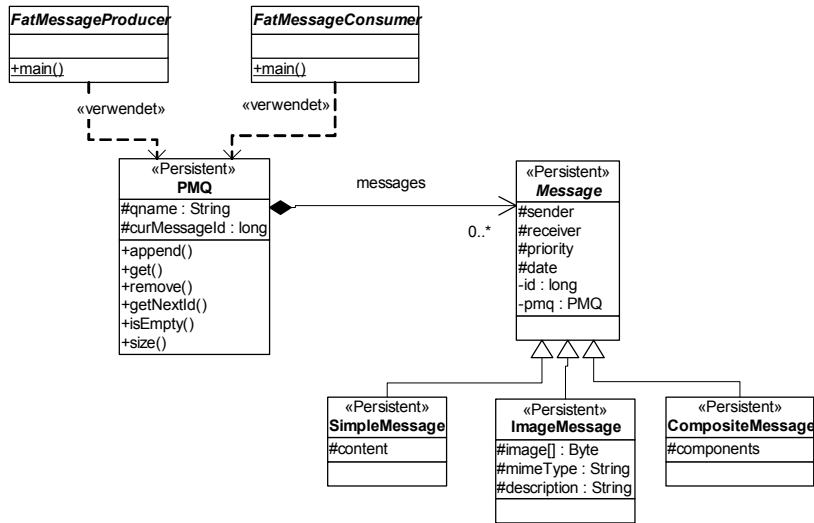
`JDOHelper`

7

`javax.jdo` enthält das API für den Benutzer (Anwendungsentwickler) von JDO

`javax.jdo.spi` enthält das API für Entwickler von JDO-Technologien (z.B. Solarmetric)

Beispiel Persistent Message Queue



8

Message Queue mit persistenten Messages haben vielfältige Einsatzmöglichkeiten:

Diagnosesysteme, Call Center Lösungen, Workflow Systeme, Auftragsabwicklung, Email-Systeme, technisches Queueing für Datenbank-Replikation usw.

Eine Message Queue hat in der Regel eine einfache Semantik (enqueue- und dequeue-Methode), tendiert aber auch dazu, gewisse Zusatzfunktionen anbieten zu müssen. Beispielsweise Filterung nach Absender, Empfänger, Priorität oder die Multiplikation einer Meldung in mehrere Queues usw.

Das Beispiel Persistent Message Queue eignet sich aus folgenden Gründen gut für die Untersuchung neuer Datenbank-Technologien:

- Es beinhaltet reine Datenobjekte (Messages) wie auch funktionsorientierte Objekte (Queues).
- Eine Vererbungshierarchie muss abgebildet werden (Vergleich relationale und objektorientierte Varianten)
- Der Collection-Typ List (und nicht nur Set) kommt zur Anwendung. Relationale Datenbanken kennen nur Sets.
- Es wird mit einer UML-Komposition gearbeitet.
- Performance-Überlegungen kommen ins Spiel bei sehr schnellen Queues.

Erzeugen persistenter Objekte

```

Properties p = new Properties();
...
PersistenceManagerFactory pmf;
pmf = JDOHelper.getPersistenceManagerFactory( p );
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tra = pm.currentTransaction();
tra.begin();
PMQ pmq = new PMQ( name );
pm.makePersistent( pmq );
tra.commit();
...

```

→ PMQInit.java

9

Die `PersistenceManagerFactory` repräsentiert eine Datenquelle (jedoch nicht eine Verbindung zu ihr). Eine `PersistenceManagerFactory` kann gemäss Spezifikation serialisiert und an einen JNDI-Namen gebunden werden. Über eine `PersistenceManagerFactory` erhält man eine Instanz von `PersistenceManager`.

Der `PersistenceManagerFactory` können über das Property-Objekt verschiedene Parameter mitgegeben werden (siehe weiter unten)

Ein `PersistenceManagerFactory`-Objekt kann über die Klasse `JDOHelper` mit `getPersistenceManagerFactory(p)` erzeugt werden.

Ein `PersistenceManager`-Objekt beinhaltet eine Verbindung zur Datenbank, ein Transaktionsobjekt und verschiedene Methode zur Verwaltung des Lebenszyklus von Daten-Objekten, beispielsweise `makePersistent()` und `deletePersistent()`. Über `PersistenceManager` können auch alle Objekte einer bestimmten Klasse (`Extent`) abgeholt oder Queries definiert werden.

Die Methode `makePersistent()` führt dazu, dass das genannte Objekt, und alle vom ihm aus erreichbaren weiteren Objekte, als persistent markiert werden. Sie erhalten eine ObjektID und werden zum Commit-Zeitpunkt in die Datenbank übertragen. Sollte ein Rollback stattfinden, verliert das Objekt und alle von ihm abhängigen wieder die Persistenz. Der Objektzustand wird auf den Wert vor dem Aufruf von `makePersistent()` zurückgesetzt.

Pro `Persistence Manager` wird *ein* Objekt-Cache geführt. Wenn also beispielsweise zweimal mit `getObjectById()` ein Objekt in der Datenbank gesucht wird, liefern die zwei Aufrufe dasselbe identische Objekt zurück, wenn die Abfragen über denselben `Persistence Manager` ausgeführt wurden.

Standard-Properties:

- javax.jdo.PersistenceManagerFactoryClass
- javax.jdo.option.Optimistic
- javax.jdo.option.RetainValues
- javax.jdo.option.RestoreValues
- javax.jdo.option.IgnoreCache
- javax.jdo.option.NontransactionalRead
- javax.jdo.option.NontransactionalWrite
- javax.jdo.option.Multithreaded
- javax.jdo.option.ConnectionUserName
- javax.jdo.option.ConnectionPassword
- javax.jdo.option.ConnectionURL
- javax.jdo.option.ConnectionFactoryName
- javax.jdo.option.ConnectionFactory2Name

Weitere Implementationsabhängige Properties dürfen gesetzt werden. Ist ein Property gesetzt, jedoch nicht unterstützt von der Implementation, wird beim Erzeugen der Factory eine `JDOFatalUserException` geworfen.

Die Methode `PersistenceManagerFactory.supportedOptions()` liefert eine Liste aller unterstützten Optionen.

Automatische Persistenz

1. Jedes transiente Objekt, welches an ein persistentes Objekt angehängt wird, ist automatisch persistent.

```

...
Message m = new SimpleMessage( ... );
pmq.append( m );
...

```

←———— m wird persistent !

2. Ein persistentes Objekt wird nur durch `PersistenceManager.deletePersistent()` aus der Datenbank entfernt.
3. Die Persistenzerstellung ist *rekursiv*, die Persistenzaufhebung ist *nicht rekursiv*.

→ `FatMessageConsumer.java` : 59-60

11

Zu 1: Kann ein Objekt über eine oder mehrere Stufen via `'.'`-Operator angesprochen werden oder ist es direkt oder indirekt Element eines Vektors, eines Arrays oder einer anderen Datenstruktur des übergeordneten Objektes, so wird es persistent. Das Message-Objekt `m` wird erst zum Commit-Zeitpunkt persistent und bekommt erst dann eine ObjektID zugewiesen. Wird die Transaktion mit `rollback` abgebrochen, so geht `m` verloren.

Im ODMG-Standard gibt es ein ähnliches Konzept, das 'Persistence by reachability' heisst: Ist ein Objekt `o` von einem persistenten Objekt `p` aus erreichbar (über direkte Dereferenzierung oder über Operationen welche eine Dereferenzierung beinhalten) so ist `o` persistent. Ist umgekehrt das Objekt `o` von `p` aus nicht mehr erreichbar, verliert es seine Persistenz. Das Objekt `p` ist im ODMG-Standard insofern ausgezeichnet, dass es einen Namen besitzt (mit `bind()` zugewiesen) und dadurch eine so genannte Persistenzwurzel ist. Das Konzept der Persistenzwurzel wurde in JDO nicht übernommen: Einmal persistente Objekte können nur explizit wieder aus der Datenbank gelöscht werden.

Zu 2: Ein Objekt, das mit `deletePersistent()` gelöscht wurde, ist anschliessend weder in der Applikation, noch in der Datenbank zugreifbar (Es wird allenfalls eine `JDOUserException` ausgeworfen).

Die Methode `PersistenceManager.makeTransient(Object o)` führt nicht zum Löschen in der Datenbank, das Objekt wird lediglich für die gerade laufende Applikation zu einem transienten Objekt. Sein Zustand wird zum Commit-Zeitpunkt nicht in die Datenbank zurückübertragen, Modifikationen führen nicht zu einem Lock auf der Datenbank. Nach Applikationsende geht das Objekt verloren, wie jedes andere transiente Objekt.

Aufsuchen persistenter Objekte

1. Der wichtigste Schritt ist das Holen von Objekten aus der Datenbank. Es gibt drei grundsätzliche Möglichkeiten, welche eine JDO-Implementation anbieten muss:
 1. Via Extent einer Klasse
 2. Via Query über eine beliebige Collection von Objekten
 3. Via ObjektID
2. Die Modifikation von persistenten Objekten geschieht 100% wie bei transienten Objekten ohne Verwendung einer Datenbank.
3. Zum Commit-Zeitpunkt werden modifizierte Objekte in die Datenbank zurückgeschrieben. Welche das sind, weiss das Datenbank-Framework, der Entwickler ist von der Buchführung darüber vollständig entlastet.

Extent

1. Der Extent ist ein *logisches* Gefäss für die Menge aller Objekte einer bestimmten Klasse, allenfalls auch ihrer Unterklassen.
2. Das Konstruieren des Extends beinhaltet noch nicht das Holen der Objekte aus der Datenbank.
3. Der Extent stellt einen Iterator zur Verfügung. Der Iterator bestimmt den Algorithmus für das Abholen der Objekte aus der Datenbank.
4. Queries können über einen Extent oder eine Collection durchgeführt werden. Queries über einen Extent werden *serverseitig* abgewickelt, Queries über eine Collection *clientseitig*.

13

Ein Extent wird via `PersistenceManager.getExtent(Class c, boolean subclasses)` konstruiert. Die wesentliche Methode von Extent ist `getIterator()` und liefert einen `java.util.Iterator` zurück.

Ob der Iterator neu erzeugte und persistente, aber noch nicht committete Objekte bereits berücksichtigt, wird durch `PersistenceManager.setIgnoreCache()` bestimmt. Das technische Problem im Hintergrund ist, dass sich persistente Objekte einer Klasse datenbankseitig befinden, neue persistente Objekte hingegen erst zum Commit-Zeitpunkt dorthin gespeichert werden. Logisch geht eine Applikation aber davon aus, dass ein neu erzeugtes Objekt zum Extent gehört, unabhängig vom Status der Transaktion.

Extent, Beispiel

```
...
PersistenceManager pm = pmf.getPersistenceManager();
tra.begin();
Extent ext = pm.getExtent( PMQ.class, true );
Iterator it = ext.iterator();
while ( it.hasNext() ) {
    PMQ pmq = (PMQ) it.next();
    if ( pmq.getName().equals( "myQueue" ) ) {
        SimpleMessage sm = new SimpleMessage( ... );
        pmq.append( sm );
    }
}
tra.commit();
...
```

14

Das Arbeiten mit einem Iterator über den Extent ist dann vernünftig, wenn der Grossteil der Objekte in der Applikation benötigt und behandelt wird. Andernfalls ist es einfacher, (-zusätzlich-) mit Queries zu arbeiten. Die Optimierung der Abfrage ist dann der Datenbank überlassen.

Queries

1. Queries über Extent
2. Queries über beliebige Collections
3. Parameter (von der Applikation in die Abfrage)
4. Variablen(innerhalb einer Abfrage)
5. Sortierung

6. Kapselung (protected, private) wird nicht beachtet
7. Keine Funktionsaufrufe (wie bei SQL-J oder OQL)
8. Kompilierung (Prepare) möglich

15

Zu 1: Queries über einen Extent werden serverseitig ausgeführt. Ob latente Objekte im Cache berücksichtigt werden ist durch `Query.setIgnoreCache()` einstellbar.

Zu 2: Queries über Collections werden clientseitig ausgeführt. Die Collection muss Objekte einer bestimmten Klasse enthalten. Die Klasse ist bei `PersistenceManager.newQuery()` oder mit `Query.setClass()` anzugeben. Queries dürfen gemäss JDO Spezifikation über Collections mit *transienten* Objekten abgesetzt werden, wenn die Implementation dies zulässt.

Zu 8: Queries können kompiliert werden (Methode `Query.compile()`). Das kommt insbesondere den relationalen Systemen entgegen, welche sehr oft die Möglichkeit besitzen, SQL-Abfragen vorzubereiten (kompilieren, Rechte prüfen, Ausführungsplan erstellen) und dann wesentlich schneller auszuführen.

Queries sind rein für Abfragen vorgesehen. Modifizierende Queries (analog zu den SQL-Befehlen update, delete, insert) existieren nicht.

Query, Beispiel 1

```

...
Extent ext = pm.getExtent( PMQ.class, false );
Query query = pm.newQuery();
query.setCandidates( ext );
query.setFilter( "qname == p_qname" );
query.declareParameters( "String p_qname" );
String param1 = args[1];
Collection result = (Collection) query.execute( param1 );
Iterator it = result.iterator();
if ( it.hasNext() ) { pmq = (PMQ) it.next(); }
// ... use pmq to add or get messages ...

```

→ **FatMessageConsumer.java: 36-51, FatMessageSelector: 85-98**

16

Dieses Query soll eine Message Queue mit dem Namen in args[1] auffinden. pm ist eine Instanz von PersistenceManager.

Die Methode setCandidates() legt fest, auf welche Objekte das Query wirken soll. Hier ist es der Extent der Klasse PMQ.

Die Methode setClass() legt fest, von welcher Klasse die abgefragten Objekte sind.

Die Methode setFilter() beinhaltet das eigentliche Query und resultiert für jedes Candidate-Object in einem booleschen Ausdruck. Der Filter darf Parameternamen enthalten. Der Filter kann die Attribute der Klasse direkt benutzen und auch weiter dereferenzieren. String- und Datums-Attribute und natürlich Zahlenwerte können direkt mit ==, !=, <, >, <=, >= verglichen werden. Boolesche Verknüpfungoperatoren sind &&, || und !. Zahlen-Ausdrücke können innerhalb von Filtern mit +, -, *, / gebildet werden. Strings können auch mit + konkateniert werden. Die einzig unterstützten Methodenaufrufe in Filtern sind String.startsWith(String s), String.endsWith(String s), Collection.isEmpty() und Collection.contains(Object o).

Die Methode declareParameters() legt Name und Typ der Parameter fest. Mehrere Parameter können durch Komma getrennt angegeben werden, beispielsweise query.setParameters("String p_qname, int p_priority"). Werte von Basistypen können als Basistypen (int, double) oder mit den entsprechenden Wrapper-Klassen (Integer, Double) deklariert sein.

Für die Methode execute() gibt es Varianten mit einem, zwei oder drei Parametern. Ausserdem gibt es die Methoden executeWithArray(Object[] params) und executeWithMap(Map params).

Query, Beispiel 2

```
Query query = pm.newQuery();
query.setCandidates( pmq.getAll() );
query.setClass( Message.class );
query.setFilter("priority <= p_prio && sender == p_sender" );
query.declareParameters( "int p_prio, String p_sender" );
Integer param1 = new Integer( args[2] );
String param2 = args[3];
Collection result = (Collection) query.execute(param1, param2);
```

→ **FatMessageSelector: 69-82**

17

Dieses Query stellt ein Collection Query dar. Es sucht nach allen Meldungen einer bestimmten Message Queue, welche eine Priorität < args[2] und den Absender args[3] haben. Im Gegensatz zum Extent Query werden nicht alle Instanzen von Message durchsucht, sondern nur diejenigen in der Vorauswahl von setCandidates().

Query, Beispiel 3

```
Extent ext = pm.getExtent( PMQ.class, false );
Query query = pm.newQuery();
query.setCandidates( ext );
query.setFilter("messages.contains(m)
                && m.priority <= p_prio" );
query.declareVariables( "Message m" );
query.declareParameters( "int p_prio" );
Integer param1 = new Integer( 3 );
query.setOrdering( "qname" );
Collection result = (Collection) query.execute( param1 );
```

18

Dieses Query sucht nach PMQ Instanzen, welche mindestens eine Meldung mit einer Priorität kleiner gleich 3 besitzen. Es benutzt die Variable m und die Operation contains(). Die resultierenden Instanzen werden nach Name sortiert.

Der Unterschied zwischen einem Parameter und einer Variablen ist, dass der Wert eines Parameters vom umgebenden Programm in das Query hineingegeben wird. Eine Variable wird nur durch ihren Namen und ihren Typ definiert, nicht durch ihren Wert. Der Wert einer Variable ergibt sich aus den Objekten von setCandidates().

Variablen-Deklaration werden mit Strichpunkt abgetrennt. Parameter-Deklaration werden mit Komma abgetrennt.

Objekt ID

```
// hole Objekt ID
Object id = pm.getObjectId( pmq );
String strId = id.toString();
// Verwende Objekt ID, z.B. in HTML-Form

// später: hole Objekt aus der Datenbank via ID.
Object id = pm.newObjectIdInstance( PMQ.class, strId );
PMQ pmq = (PMQ) pm.getObjectById( id, true );
```

19

Das Auffinden einzelner Objekte über die Objekt-ID ist sehr effizient.

Wenn der zweite Parameter von `getObjectId()` `true` ist, muss die gesuchte ID in der Datenbank existieren, ansonsten wird eine `JDODataStoreException` ausgeworfen. Wenn der zweite Parameter `false` ist, wird ein leeres Objekt erzeugt, wenn die gesuchte ID in der Datenbank nicht existiert.

Zu obigem Beispiel: Die OID eines Objektes wird vom Programm als String ausgegeben. Später holt der Benutzer das Objekt wieder aus der DB in die Applikation durch Eingabe der OID in `args[0]`.

Die ObjektID im String-Format ist beispielsweise für Web-Applikationen mit HTML sehr praktisch.

Instance Callbacks

- Instance Callbacks ermöglichen Verarbeitungsoperationen bevor ein Objekt in die Datenbank geschrieben oder nachdem es daraus gelesen oder gelöscht wurde.

```
Interface InstanceCallbacks {
    public void jdoPostLoad();
    public void jdoPreStore();
    public void jdoPreDelete();
    public void jdoPreClear();
}
```

20

`jdoPostLoad()` wird nach dem Laden eines Objektes aus der Datenbank durchgeführt (im Rahmen einer Lese-Operation oder beim refresh). Die Funktionen sollte nicht auf andere persistente Objekte zugreifen oder den eigenen Zustand persistenter Felder ändern. Sie eignet sich für die Initialisierung transienter Felder des eigenen Objektes, zum Beispiel Entschlüsselung des Inhaltes oder Dekompression.

`jdoPreStore()` wird vor dem Speichern eines Objektes in die Datenbank durchgeführt (Objekte im Zustand 'Persistent Dirty' oder 'Persistent-New'). Die Funktionen kann auf andere persistente Objekte zugreifen oder den eigenen Zustand persistenter Felder ändern. Sie eignet sich für die Übertragung transienter Felder in persistente Felder des eigenen Objektes, zum Beispiel Verschlüsselung des Inhaltes oder Kompression, Setzen benutzerdefinierter Zeitstempel oder Verwaltungsattribute.

`jdoPreDelete()` wird aufgerufen, wenn ein Objekt in den Zustand 'Persistent deleted' oder 'Persistent new deleted' eintritt. Dies ist im Rahmen eines Aufrufes von `deletePersistent()` der Fall. Die Funktionen kann auf andere persistente Objekte zugreifen oder den eigenen Zustand persistenter Felder ändern. Sie kann beispielsweise auch zum Löschen weiterer Objekte benutzt werden, die vom gelöschten Objekt abhängig sind ("kaskadierte Löschung").

`jdoPreClear()` wird beim Entleeren eines Objektes aufgerufen d.h. beim Übergang in den Zustand 'Hollow' aus den Zuständen 'Persistent deleted', 'Persistent new deleted', 'Persistent clean' und 'Persistent dirty'.

Multithreading

1. Eine JDO Implementation *darf* Multithreading zulassen auf den Klassen `Transaction`, `Extent`, `Query`, `PersistenceManager`, sowie auf den eigentlichen Datenklassen.
2. Das Multithreading kann ein- oder ausgeschaltet werden:
`PersistenceManagerFactory.setMultithreaded(boolean)`
`boolean PersistenceManagerFactory.getMultithreaded()`
3. Applikationskontrollierte Synchronisation ist jederzeit möglich.

21

Zu beachten: Bei JDO ist ein `PersistenceManager` und eine Transaktion 1:1 gekoppelt. Es ist daher zu jedem Zeitpunkt eindeutig, im Rahmen welcher Transaktion ein Objekt gelesen, modifiziert oder gelöscht wird. Die Erzeugung eines Objektes erfolgt durch die Klasse `PersistenceManager`, die Zuordnung zu einer Transaktion und einer Datenbank ist damit über das Objekt selber eindeutig gegeben. Die Anzahl Threads spielt dabei keine Rolle.

Im Rahmen von ODMG können zu einem Datenbankobjekt (Klasse `Database`) *mehrere* Transaktionsobjekte (Klasse `Transaction`) instanziiert werden. Persistente Objekte werden aber aus der Klasse `Database` heraus erzeugt. Damit ist bei mehreren laufenden Transaktionen nicht mehr eindeutig, im Rahmen welcher dieser Transaktionen ein Objekt erzeugt, gelesen, modifiziert oder gelöscht wird. Die Klasse `Transaction` besitzt daher die Methoden `join()` und `leave()`, mit denen das Transaktionsobjekt einem bestimmten Thread zugeordnet werden kann. Bei JDO ist diese Zuordnung und damit die beiden Methoden überflüssig.

Die Klasse `PersistenceManager` kennt die Methoden `setUserObject()` und `getUserObject()`. Damit kann ein beliebiges Objekt an den `PersistenceManager` gekoppelt werden, beispielsweise um bestimmte Parameter oder Zustände festzuhalten, oder als Synchronisationsobjekt für Multithreading in Java.

Multithreading, Beispiel

Producer Thread (n mal)

```
public class Producer extends Thread {
    protected Transaction tra;
    public Producer( Transaction tra )
    {
        this.tra = tra;
    };

    public void run()
    {
        while (getDataFromSomewhere() ) {
            synchronized ( tra ) {
                // database access...
            }
        }
    }
}
```

Controller Thread (1 mal)

```
public class Controller {
    public static void main( String[] args )
    {
        // create and start producer threads
        Transaction tra = new Transaction();
        tra.begin();
        Producer prd1 = new Producer( tra );
        prd1.start();
        Producer prd2 = new Producer( tra );
        prd2.start();
        // commit every 100 msec
        while ( true ) {
            Thread.sleep( 100 );
            synchronized ( tra ) {
                tra.commit();
                tra.begin();
            }
        }
    }
}
```

22

Mehrere Producer-Threads erzeugen Messages über dieselbe Verbindung/Transaktion in eine OODB.

Ein Commit-Befehl wird *periodisch* für alle Threads durchgeführt.

Weil das Erzeugen der Messages lokal stattfindet, und erst beim Commit die Messages übertragen werden, ergeben sich massive Performance-Gewinne bis zu Faktor 100.

Objektorientierte Datenbanken haben wesentliche Vorteile, wenn eine hohe Dichte von Änderungs- oder Einfügeoperationen auf persistenten Objekte verlangt ist. Da alle Operationen im lokalen Cache stattfinden und zum Commit-Zeitpunkt nur noch die *Daten* als Ganzes übertragen werden, ist ein Grossteil des Transfer-Overheads nur einmal zu erledigen. Bei Relationalen Datenbanken (SQL) wird jede Änderungs- oder Einfügeoperation *einzel*n übertragen, oder allenfalls als Block von einzelnen *Befehlen*. Ausserdem ist das Führen der Log-Files aufwändiger, weil sowohl Before- wie After-Images abgelegt werden müssen. Es ist ja zum Änderungs- oder Einfügezeitpunkt noch nicht bekannt, ob die Transaktion mit einem Commit oder Rollback enden wird. Bei Objektdatenbanken werden Objekte erst transferiert, wenn ein Commit stattfinden soll. Für Rollback-Operationen gibt es keinen Aufwand datenbankseitig, weil lediglich der lokale Cache geflushed wird. Ausserdem entfällt der Aufwand für das Parsen der Einfüge- und Änderungs-Befehle, da bei Objektdatenbanken nur *Daten*, keine *Befehle* transferiert werden.

Objektidentität

JDO Identities

1. Jedes persistente Objekt muss dauerhaft und eindeutig identifizierbar sein. Es besitzt eine Objekt-ID.
2. Aufgrund seiner Identität kann ein persistentes Objekt in der Datenbank aufgefunden werden, z.B. mit `PersistenceManager.getObjectById(Object oid)`
3. Aufgrund seiner Identität kann ein persistentes Objekt von anderen persistenten Objekten referenziert werden. Damit bleiben Beziehungen zwischen Objekten in der Datenbank erhalten.
4. In JDO gibt es 3 Arten von Objekt-Identität
 - Application defined Identity
 - Data Store defined Identity
 - Nondurable Identity

→ **FatMessageSelector: 45**

24

Im Rahmen einer gewöhnlichen Java-Applikation ist die Objekt-ID im wesentlichen die Speicheradresse eines Objektes. Für persistente Objekte ist diese ID aus offensichtlichen Gründen ungenügend. Weil JDO beansprucht, mit verschiedensten Datenbank-Typen umgehen zu können (relational, objektorientiert, flat files), sind verschiedene Arten von Objekt-ID's denkbar:

- Application defined Identity: Diese entspricht dem relationalen Primärschlüssel, im allgemeinen Fall also einer Kombination verschiedener Werte aus den Nutzdaten. Definiert werden die Schlüssel-Felder durch die Applikation, resp. deren Entwickler.
- Data Store defined Identity: Dies sind von den Nutzdaten unabhängige, generierte Werte. Sie werden nach einem vorgegebenen Algorithmus von der Datenbank erzeugt und einem Daten-Objekt zugewiesen. Objektorientierte Datenbank arbeiten typischerweise mit solchen ID's. Besitzt ein Java-Objekt eine (Speicher-)Referenz auf ein anderes Java-Objekt, so muss diese Referenz beim Übertragen des ersten Objektes in die Datenbank in eine Objekt-ID auf das referenzierte Objekt übersetzt werden.
- Nondurable Identity: In der Datenbank hat ein Objekt keine ID. Die ID wird beim Laden in die Applikation erzeugt. Typischerweise könnte das bei serialisierten Objekten der Fall sein oder einfachen Datensätzen ohne Primärschlüssel. Die Objekte können nicht via ID, aber via Queries aus der Datenbank in die Applikation geladen werden. Dasselbe Datenbankobjekt kann mehrmals nacheinander geladen werden und bekommt jeweils eine neue nondurable ID's. Das Auffinden in der Datenbank beim Zurückschreiben ist wertbasiert oder einfach kopierend.

Die ObjektID wird als Java-Klasse implementiert. Bei allen Methoden im JDO-API, wo eine ObjektID übergeben oder zurückgegeben wird, ist ein allgemeines Java-Objekt in der Signatur verwendet. Das ermöglicht grösstmögliche Generizität.

Die Art der ObjektID wird pro Datenklasse im JDO-Konfigurationsfile festgelegt.

Application Identity, Beispiel

```
// ObjectID Klasse für Datenklasse PMQ
class PMQId implements java.io.Serializable {
    public String qname;
    public PMQId( String qname ) { this.qname = qname; }
    public String toString() { return qname; }
    public int hashCode() { return qname.hashCode(); }
    public boolean equals( Object o ) {
        return qname.equals(((PMQId)o).toString() );
    }
}

// Objekt mit einer bestimmten ID suchen
PMQId pmqId = new PMQId( "myPMQ" );
PMQ pmq = (PMQ) pm.getObjectById( pmqId )
```

25

Für die ObjectID-Klasse muss gelten:

1. Sie ist öffentlich und serialisierbar.
2. Alle nicht-statischen Felder sind serialisierbar und öffentlich.
3. Sie hat nur öffentliche Felder. Diese sind Basistypen, Instanzen von numerischen Wrapper-Klassen, der `String`- oder der `Date`-Klasse.
4. Die Feldnamen und -Typen, welche den Primärschlüssel definieren, sind in der Datenklasse und in der ObjectID-Klasse identisch.
5. Die `equals()` und die `hashCode()`-Methode sind von allen Schlüsselfeldern abhängig.
6. Es gibt einen öffentlichen Konstruktor ohne Argumente und einen öffentlichen Konstruktor mit einem String-Argument.
7. Es gibt eine `toString()`-Methode, deren return-Wert als Parameter für einen Konstruktor von ObjectID verwendbar ist. Wird eine ObjektID-Instanz x mit einem String aus der `toString()`-Methode einer ObjektID-Instanz y konstruiert, so soll der Vergleich `x.equals(y)` wahr sein.
8. Alle Daten-Klassen in einer Ableitungshierarchie verwenden dieselbe ObjectID-Klasse.

Die ObjectID-Klasse für eine bestimmte Datenklasse wird von der Applikation implementiert und im JDO-Konfigurationsfile als solche deklariert, beispielsweise:

```
<class name="PMQ" identity-type="application" objectidclass="PMQId">
  <field name="qname" primary-key="true"/>
</class>
```

Die `public`-Felder einer `ObjectId`-Instanz sollen keinesfalls modifiziert werden. Änderungen an der `ObjectId`-Instanz werden nicht an das Datenobjekt weiterpropagiert. Umgekehrt jedoch schon. Bei Änderungen an einem Feld des Datenobjektes, das zur Objekt-ID gehört, kann mit `PersistenceManager.getObjectId()` eine neue `ObjectId`-Instanz abgeholt werden, welche die Änderungen am Datenobjekt reflektiert. Besteht die Objekt-ID bereits (Verletzung der Primärschlüssel-Integrität) wird eine `JDOUserException` ausgeworfen und die Änderung wird rückgängig gemacht.

Zu beachten: Wird mit Application Identity gearbeitet, sollte die `equals()` und die `hashCode()`-Methode in den Datenobjekten so implementiert werden, dass der Vergleich von zwei Datenobjekten dasselbe Resultat liefert, wie der Vergleich ihrer `ObjectId` Instanzen. Ansonsten kann es zu Problemen beim Einfügen von Datenobjekten in persistente Mengen (`Set`) kommen, welche sicherstellen, dass ein Objekt nur einmal vorkommt. Ebenso kann es zu Problemen in der Abfragesprache beim Vergleich von Objekten mit `'=='` kommen.

Data Store Identity, Beispiel

```
// erzeuge persistentes Objekt und merke OID
PMQ pmq = new PMQ( "myPMQ" );
pm.makePersistent( pmq );
System.out.println( pm.getObjectId( pmq ).toString() );
// merke OID.
// später: hole Objekt aus der Datenbank via ID.
Object id = pm.newObjectIdInstance( PMQ.class, strId );
PMQ pmq = (PMQ) pm.getObjectById( id, true );
```

- Die Objekt-ID wird zum Zeitpunkt des Aufrufs von `PersistenceManager.makePersistent()` zugewiesen.
- Die DB-Implementation garantiert für die Eindeutigkeit.
- Die Objekt-ID kann nicht geändert werden.

27

Data Store Identity ist der Default-Mechanismus, wenn keine anderen Angaben im JDO-Konfigurationsfile vorhanden sind .

Wenn der zweite Parameter von `getObjectId()` `true` ist, muss die gesuchte ID in der Datenbank existieren, ansonsten wird eine `JDODataStoreException` ausgeworfen. Wenn der zweite Parameter `false` ist, wird ein leeres Objekt erzeugt, wenn die gesuchte ID in der Datenbank nicht existiert.

Zu obigem Beispiel: Die OID eines Objektes wird vom Programm als String ausgegeben. Später holt der Benutzer das Objekt wieder aus der DB in die Applikation durch Eingabe der OID in `args[0]`.

Für die `ObjectId`-Klasse gilt:

1. Sie ist öffentlich und serialisierbar
2. Alle nicht-statischen Felder sind serialisierbar und öffentlich.
3. Es gibt einen öffentlichen Konstruktor ohne Argumente und einen öffentlichen Konstruktor mit einem String-Argument.
4. Es gibt eine `toString()`-Methode, deren return-Wert als Parameter für einen Konstruktor von `ObjectId` verwendbar ist. Wird eine `ObjectId`-Instanz `x` mit einem String aus der `toString()`-Methode einer `ObjectId`-Instanz `y` konstruiert, so soll der Vergleich `x.equals(y)` wahr sein.

Das Auffinden einzelner Objekte über die (Data Store) Objekt-ID ist wesentlich effizienter als über Queries oder Extents.

III Persistenzmodell

- Grundsatz / Anforderungen der Datenbank
- Enhancer
- Konfigurationsdatei
- First Class / Second Class Objekte
- spezielle Felder

Grundsatz

- Für den Entwickler sollen sich das Arbeiten mit der Datenbank auf gewisse *logische* Aufgaben beschränken:
 - Objekte als persistent markieren oder löschen
 - Transaktion starten, comitten oder rollbacken
 - Objekte gezielt aus der Datenbank holen via Query oder Objekt-ID
- Für den programmiersprachlichen Umgang mit Objekten gilt:

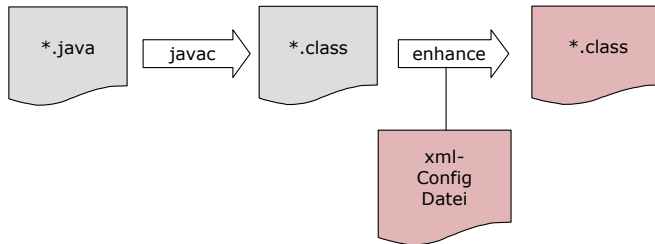
Arbeiten mit Datenbankobjekten = Arbeiten mit Java-Objekten

Anforderung der Datenbank

- Die Datenbank muss applikationsseitig informiert sein über:
 - Anforderung eines Objektes aus der Datenbank durch eine Dereferenzierungsoperation im Applikationscode mit . oder [] Operation
 - Änderungen am Objekt durch Zuweisung, Increment / Decrement Operationen etc.
 - Den Objektzustand (hollow, clean, dirty, deleted usw.)
- Der Byte-Code von JDO-Applikationen wird modifiziert: Dereferenzierungs- und Zuweisungsoperatoren werden "umprogrammiert" in Methodenaufrufe.
- In den Methodenaufrufen wird das Datenbanksystem aktiviert für das Holen von Objekten aus der Datenbank, das Setzen von Flags (hollow, clean, dirty, deleted usw.)

JDO Enhancer

- Die Umprogrammierung des Java-Codes (Enhancement) findet auf Ebene Bytecode, also mit den Class-Files statt.



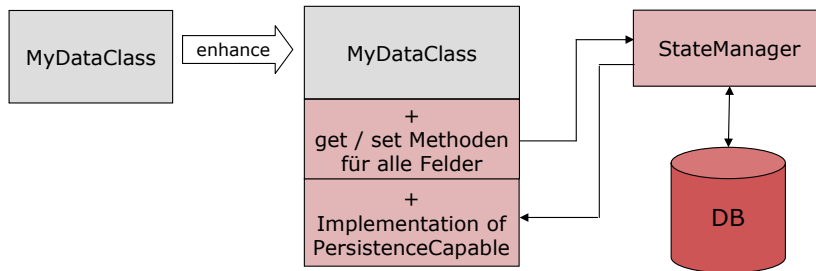
- Der Enhancer kann ein separates Java-Programm oder ein spezieller Class-Loader sein, welcher das Enhancing on the fly durchführt.

31

Der Enhancer ist in der Regel selber ein Java-Programm. Der Vorgang heisst auch Postprocessing, im Gegensatz zu Preprocessing. Letzteres würde heissen, es wird der Java-Sourcecode und nicht der Java Bytecode umprogrammiert. Bei C++ Anwendungen mit OO-Datenbanken wird meist mit Preprocessing gearbeitet.

JDO Enhancer

- In Java-Klassen, welche mit persistenten Objekten *arbeiten*, werden die Operatoren, welche diese Objekte verändern, in *Methodenaufrufe* umprogrammiert.
- In Java-Klassen, welche *selber persistente Objekte* repräsentieren, werden die entsprechenden Methoden durch den Enhancer *implementiert*.



32

Der Enhancer ist in der Regel selber ein Java-Programm. Der Vorgang heisst auch Postprocessing, im Gegensatz zu Preprocessing. Letzteres würde heissen, es wird der Java-Sourcecode und nicht der Java Bytecode umprogrammiert. Bei C++ Anwendungen mit OO-Datenbanken wird meist mit Preprocessing gearbeitet.

Gegenüber dem ODMG Standard ist bei JDO definiert, wie eine Klasse nach dem Postprocessing aussehen muss. Beispielsweise ist vorgeschrieben, dass eine Datenklasse das Interface `PersistenceCapable` implementieren und eine Änderung an einem Objekt via `StateManager` an die Datenbank propagiert werden muss.

Die ganze Abbildungs-Logik auf ein Datenbanksystem liegt im `StateManager`.

Beispiel Klassen-Code vor Enhance

```
public class Person
{
    public String name;
    public Person partner;
}
```

Beispiel Applikations-Code vor Enhance

```
public abstract class PersonApplication
{
    ...
    public static void main(String args[])
    {
        ...
        Transaction tra = pm.currentTransaction();
        tra.begin();

        // make person persistent
        Person person = new Person();
        pm.makePersistent( person );

        // get name of person
        String s = person.name;

        // set name of person
        person.name = "Arthur";

        // get partner
        Person person2 = person.partner;

        tra.commit();
        ...
    }
}
```

Beispiel Klassen-Code nach Enhance

```

public class Person implements PersistenceCapable
{
    public String name;
    public Person partner;
    protected transient StateManager jdoStateManager;
    protected transient byte jdoFlags;
    ...
    public static final void jdoSetName(Person person, String s)
    {
        if(person.jdoFlags == 0) {
            person.name = s;
            return;
        }
        statemanager.setStringField(
            person, jdoInheritedFieldCount + 0, person.name, s);
        return;
    }
    ...
}

```

Beispiel Applikations-Code nach Enhance

```

public abstract class PersonApplication
{
    ...
    public static void main(String args[])
    {
        try {
            Transaction tra = pm.currentTransaction();
            transaction.begin();
            Person person = new Person();
            pm.makePersistent(person);
            String s = Person.jdoGetName(person);
            Person.jdoSetName(person, "Arthur");
            Person person2 = Person.jdoGetPartner(person);
            transaction.commit();
        }
    }
}

```

Konfigurationsdatei

- Die Konfigurationsdatei hat drei Zwecke:
 - Definition, welche Klassen in der Datenbank gespeichert werden sollen.
 - Verändern des Default-Verhaltens.
 - Zusatzinformation liefern, welche die JDO-Implementation nicht aus der Klassendefinition ableiten kann.
- Die Konfigurationsdatei ist in XML erstellt und die Struktur durch einen DTD definiert.

35

Siehe Kapitel 18 JDO-Spezifikation.

XML Konfigurationsdatei

- Beispiel `pmq.jdo` für eine Objektdatenbank

```
<?xml version="1.0" encoding="UTF-8"?>
<jdo>
  <package name="pmq">
    <class name="Message"/>
    <class name="ImageMessage"
      persistence-capable-superclass="Message"/>
    <class name="SimpleMessage"
      persistence-capable-superclass="Message"/>
    <class name="CompositeMessage"
      persistence-capable-superclass="Message"/>
    <class name="PMQ">
      <field name="messages" embedded="true">
        <collection element-type="Message"/>
      </field>
    </class>
  </package>
</jdo>
```

36

Das obige Konfigurationsfile enthält folgende Aussagen:

Die Klassen `Message`, `ImageMessage`, `SimpleMessage` und `CompositeMessage` haben persistente Objekte. Die Klasse `PMQ` hat ebenfalls persistente Objekte und enthält eine *typisierte* Collection von `Message`-Objekten angelegt. Das Collectionobjekt selbst soll ein Second Class Objekt sein (`embedded="true"`), die von der Collection referenzierten Objekte sind jedoch First Class Objekte (Wenn die `Message`-Objekte ebenfalls Second Class Objekte sein sollen müsste `<collection embedded-elements="true">` spezifiziert sein). Gemäss JDO-Spezifikation ist der Default für alle Basistypen und ihre Wrapperklassen, sowie `Date` und die meisten Collection-Klassen `embedded="true"`.

Das Konfigurationsfile beschreibt nur die *persistenten* Klassen. Alle Objekte von Klassen, welche in der Datei `pmq.jdo` vorkommen, werden als persistenzfähig und als First Class Objekte betrachtet. Klassen, die als Member von persistenten Klassen verwendet werden, aber im Konfigurationsfile nicht genannt sind, gelten als Second Class Objekte. Der Aufruf des Enhancers erfolgt beispielsweise wie folgt:

```
java kodo.enhance.JDOEnhancer -properties kodo.properties pmq\pmq.jdo
```

Diejenigen Klassen und Methoden, welche persistente Objekte nur *bearbeiten*, benötigen keine spezielle Konfiguration, müssen aber vom Enhancer auch bearbeitet werden, beispielsweise wie folgt:

```
java kodo.enhance.JDOEnhancer -properties kodo.properties *.class
```

Wenn eine Klasse mit application defined identity arbeiten soll, muss beispielsweise folgendes angegeben werden:

```
<class name="PMQ" identity-type = "application" objectid-class="PMQid">
  Defaultwert für den identity-type ist datastore.
```

Die Angabe `persistence-capable-superclass` ist nur notwendig, wenn die Klasse von einer anderen, persistenten Klasse abgeleitet ist.

Der Ausschnitt aus dem DTD für die Konfiguration von Klassen sieht wie folgt aus:

```
<!ELEMENT class (field|extension)*>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class identity-type (application|datastore|nondurable) #IMPLIED>
<!ATTLIST class objectid-class CDATA #IMPLIED>
<!ATTLIST class requires-extent (true|false) 'true'>
<!ATTLIST class persistence-capable-superclass CDATA #IMPLIED>
```

Der Ausschnitt aus dem DTD für die Konfiguration von Felder sieht wie folgt aus:

```
<!ELEMENT field ((collection|map|array)?, (extension)*)?>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field persistence-modifier
    (persistent|transactional|none) #IMPLIED>
<!ATTLIST field primary-key (true|false) 'false'>
<!ATTLIST field null-value (exception|default|none) 'none'>
<!ATTLIST field default-fetch-group (true|false) #IMPLIED>
<!ATTLIST field embedded (true|false) #IMPLIED>

<!ELEMENT collection (extension)*>
<!ATTLIST collection element-type CDATA #IMPLIED>
<!ATTLIST collection embedded-element (true|false) #IMPLIED>

<!ELEMENT map (extension)*>
<!ATTLIST map key-type CDATA #IMPLIED>
<!ATTLIST map value-type CDATA #IMPLIED>
<!ATTLIST map embedded-key (true|false) #IMPLIED>
<!ATTLIST map embedded-value (true|false) #IMPLIED>

<!ELEMENT array (extension)*>
<!ATTLIST array embedded-element (true|false) #IMPLIED>

<!ELEMENT extension (extension)*>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>
```

Die default-fetch-group ist diejenige Gruppe von Feldern, die beim Laden des Objektes von der Datenbank in die Applikation übertragen wird. Andere Felder werden erst bei Bedarf nachgeladen. Primitive Typen, sowie String und Date gehören per default zur default-fetch-group.

extension ist ein Element, um produktspezifische Erweiterungen einzubringen. ObjectDB definiert beispielsweise damit, welche Attribute indexiert werden sollen:

```
<class name="A">
  <extension vendor-name="objectdb" key="index" value="f0" />
  <extension vendor-name="objectdb" key="unique-index" value="f1" />
</class>
```

First Class / Second Class Objects

- First Class Objekt (FCO)
 - hat eine ObjektID
 - ist shareable
 - ist Transfereinheit
 - ist abfragbar
- Second Class Objekt (SCO)
 - hat keine ObjektID
 - ist non-shareable
 - gehört immer zu einem First Class Objekt
 - ist nicht abfragbar



als FCO deklarierte und benutzerdefinierte Klasse



String
Date
Vector
Array
...
weitere, nicht als FCO deklarierte, serialisierbare Klassen

38

Der Lebenszyklus von First Class Objekten kann mit `makePersistent()` / `deletePersistent()` explizit kontrolliert werden. First Class Objekte sind die abfragbare Einheit in einem Query (Eine Abfrage "Suche alle Objekte welche ..." bezieht sich immer auf First Class Objekte). Die Felder von First Class Objekten werden durch einen StateManager einzeln behandelt.

Second Class Objekte haben, wie die Basistypen `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, `double`, die Funktion von Typen. Im technischen Sinne von Java sind es jedoch Objekte. In der Datenbank haben Second Class Objekte keine eigene ID, sie sind physikalisch in die umgebenden First Class Objekte eingebettet und sind daher eher *Werte*, statt Objekte. Second Class Objekte werden meist als Ganzes durch Serialisierung von und zur Datenbank übertragen. Benutzerdefinierte SCO-Klassen müssen daher das Interface `Serializable` implementieren.

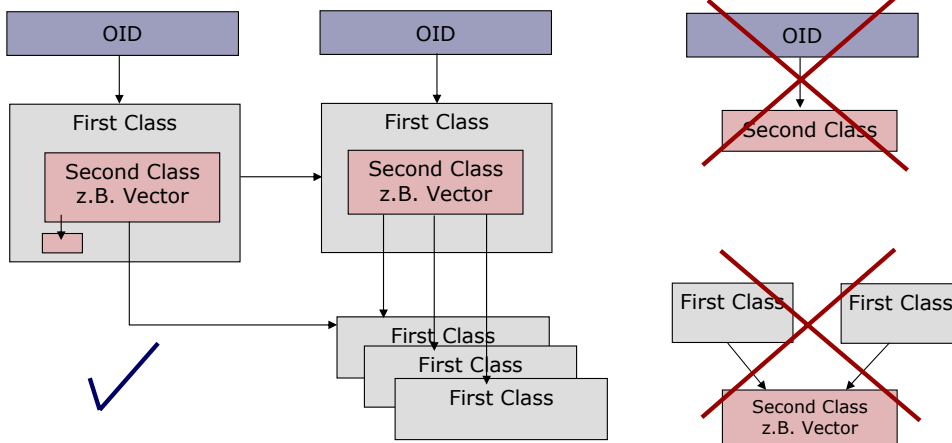
JDO schreibt vor, dass folgende Java-Klassen, ohne weitere Deklaration durch den Benutzer speicherbar sein müssen (wahlweise als FCO's oder SCO's): `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `String`, `Date`, `Locale`, `HashSet`, `ArrayList`, `HashMap`, `Hashtable`, `LinkedList`, `TreeMap`, `TreeSet` und `Vector`. Auch die Interfaces `Collection`, `Set`, `Map` und `List` sollen verwendet werden dürfen. Arrays müssen nur optional unterstützt werden, was in Implementationen basierend auf OO-Datenbanken meist der Fall ist.

Objektdatenbanken stellen teilweise gleichzeitig eine SCO- und eine FCO-Variante einer Klasse bereit. Beispielsweise sind in der OODB Versant die Klassen `VVector` und `LargeVector` FCO's, die Klassen `DVector` und `Vector` jedoch SCO's.

In relationalen Datenbanken wird ein FCO in einen oder mehrere Datensätze einer Tabelle abgebildet (mit Primärschlüssel), während ein SCO zu einem einzelnen Wert eines Attributes wird (keine eigene Identität).

Eine benutzerdefinierte Klasse, welche das Interface `Serializable` implementiert, gilt als Second Class, wenn sie nicht als First Class in der XML-Konfigurationsdatei deklariert ist.

First Class / Second Class Objects ff



39

Aus Sicht der Applikation macht sich der Unterschied zwischen FCO's und SCO's einzig dadurch bemerkbar, dass Second Class Objekte non-shareable sind. Zwar ist es möglich, in der Applikation eine Situation zu konstruieren, dass beispielsweise zwei First Class Objekte auf dasselbe `Vector`-Object zeigen. Beim Zurückschreiben in die Datenbank (`commit`) findet allerdings eine Spaltung dieses `Vector`-Objektes in zwei Second Class Objekte statt. Jedes davon wird einem seiner First Class Objekte zugeteilt. Beim nächsten Laden in die Applikation hat jedes First Class Objekt eine eigene Instanz des `Vector` Objektes.

Second Class Objekte können Referenzen auf First Class Objekte enthalten, oder auch auf andere Second Class Objekte (beispielsweise Strings).

Second Class Objekte haben keine ObjektID, sie können also beispielsweise nicht für sich alleine persistent gemacht werden mit `makePersistent()`.

Spezielle Felder – `static,transient, final`

- Der Lebenszyklus eines `static` Feldes wird von JDO nicht kontrolliert. Es gelten die allgemein Java Regeln.
- Der Lebenszyklus eines `transient` Feldes wird von JDO nicht kontrolliert. Es gelten die allgemeinen Java Regeln.
- Ein `final` Feld wird nicht einer JDO-Datenbank gespeichert. Da sie nur vom Konstruktor initialisiert werden dürfen, kann sich dadurch ein unerwartetes Verhalten ergeben, beim Laden von Objekten aus der Datenbank.
- Ein `final static` Felder zeigt dasselbe Verhalten wie in einer allgemeinen Java-Applikation

IV O-R Mapping

41

Das OR-Mapping ist in der JDO-Spezifikation 1.0.1 nur ansatzweise definiert.
Nichtsdestoweniger ist die Produktwelt schon fortgeschritten.
Siehe Technologie Memo zu Kodo/JDO Relational Mapping.

V Transaktionsmanagement

Transaktionsmodi

- JDO stellt mehrere Transaktionsmodi zur Verfügung
 1. Persistent Transactional (Normalfall)
 2. Persistent Nontransactional
 3. Transient Transactional
 4. Optimistic Transactions

43

JDO definiert sehr genau die Zustände und Zustands-übergänge von Objekten in und zwischen den verschiedenen Transaktionsmodi. Diese wesentliche Leistung von JDO ermöglicht es den Technologie-Herstellern ein einheitliches Transaktionsverhalten zu implementieren.

Zu 1: *Persistent Transactional* ist der Standard-Fall für eine Datenbank. Objekte werden im Rahmen einer Transaktion gelesen und geschrieben. Dabei werden auf den Objekten in der Datenbank entsprechende Sperren gesetzt. Zum Commit-Zeitpunkt werden veränderte Objekte in die Datenbank zurückgeschrieben und der lokale Objekt-Cache geleert. Letzteres deshalb, weil die Übereinstimmung von Cache und Datenbank nicht mehr gesichert ist, es bestehen ja keine Lese- oder Schreibsperren mehr. Ein Objekt, das im Rahmen einer Transaktion direkt oder indirekt mit `makePersistent()` persistent gemacht oder aus der Datenbank zwecks lesendem oder schreibendem Zugriff entnommen wurde, besitzt den Zustand *Persistent Transactional*. Folgende Properties sollten der *PersistenceManagerFactory* mitgegeben werden, um ausschliesslich im Modus *Persistent Transactional* zu arbeiten:

```
javax.jdo.option.RetainValues = false
javax.jdo.option.NontransactionalRead = false
javax.jdo.option.NontransactionalWrite = false
```

Zu 2: *Persistent Nontransactional* bedeutet, es dürfen Objekte auch ausserhalb einer Transaktion geschrieben und gelesen werden. Zu unterscheiden für die praktische Bedeutung ist dabei 'Nontransactional Read' und 'Nontransactional Write'. 'Nontransactional Read' ist häufig eine praktische und effiziente Lösung für das Lesen von Daten, die sicher nicht modifiziert und zurückgeschrieben werden. Es werden keine Sperren gesetzt, die Daten stehen damit jederzeit zur Verfügung. Daten, die nicht im Rahmen einer Transaktion gelesen werden, müssen auch nicht zum Commit-Zeitpunkt freigegeben werden. Die Daten können im Cache verweilen und dürfen lesend verwendet werden, solange die Applikation die Daten benötigt. Folgende Properties müssen der *PersistenceManagerFactory* mitgegeben werden, um vollumfänglich im Modus *Persistent Nontransactional* arbeiten zu können:

```
javax.jdo.option.RetainValues = true
javax.jdo.option.NontransactionalRead = true
javax.jdo.option.NontransactionalWrite = true
```

'Nontransactional Write' bedeutet, Daten können *ausserhalb* einer Transaktion gelesen und lokal modifiziert werden. Der Vorteil liegt darin, dass während der Bearbeitung keine Sperren in der Datenbank gesetzt sind. Ein modifiziertes Objekt kann nicht direkt in die Datenbank zurückgeschrieben werden. Es muss eine Transaktion eröffnet werden und die Änderungen sind nochmals durchzuführen. Ein Anwendung von 'Nontransactional Write' sind "Was wäre wenn"-Programme: Von der Datenbank gelesene Daten können direkt geändert und dargestellt werden, es besteht jedoch meist nicht der Anspruch, sie in die Datenbank zurückzuschreiben. Ein persistentes Objekt im Zustand 'Nontransactional' kann *innerhalb* einer Transaktion modifiziert werden: Es wird damit automatisch in die Transaktion eingekoppelt und beim commit in die Datenbank zurückgeschrieben. Dies gilt allerdings nur für die Änderungen, die innerhalb der Transaktion durchgeführt wurden. Die Änderungen vor der Transaktion gehen verloren.

Zu 3: *Transient Transactional* bedeutet, dass der Zustand *transienter* Objekte im Rahmen eines Rollback-Befehls zurückgesetzt werden kann. Der Zustand eines Objektes, das als transient transactional markiert ist, wird zu Beginn einer Transaktion festgehalten. Erfolgt ein Commit, bleibt der Zustand des Objektes unverändert. Erfolgt ein Rollback, wird der Zustand des Objektes wie zu Beginn der Transaktion gesetzt. Der Vorteil von Objekten im Zustand Transient Transactional liegt darin, dass sie mit persistenten Objekten "synchron" gehalten werden können (Beispiel GUI-Objekte), auch wenn sie nicht unbedingt in der Datenbank aufbewahrt werden sollen. Folgende Properties sollten der PersistenceManagerFactory mitgegeben werden, um mit im Modus Transient transactional arbeiten zu können:

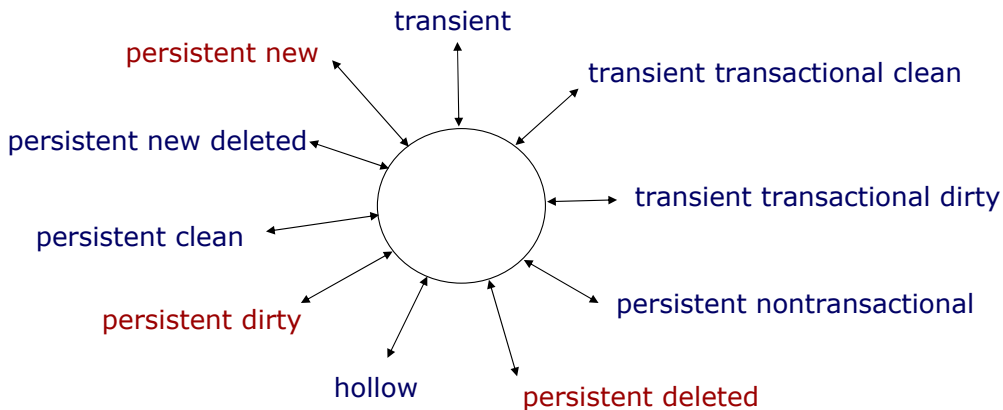
```
javax.jdo.option.RestoreValues = true
```

Zu 4: *Optimistic Transactions* heisst, ein Objekt wird zusammen mit dem Zeitstempel der letzten Modifikation aus der Datenbank gelesen. Es wird keine dauerhafte Lesesperre in der Datenbank gesetzt. Andere Transaktionen können also das Objekt ebenfalls lesen und verändern. Zum Commit-Zeitpunkt wird anhand des Zeitstempels geprüft, ob das Objekt zwischenzeitlich von einer anderen Transaktion modifiziert wurde. Wenn nein, wird das Objekt in die Datenbank geschrieben und der Zeitstempel angepasst. Wenn ja, wird die Transaktion abgebrochen und ein Rollback durchgeführt (geänderte Objekte werden auf Zustand wie beim letzten Lesen zurückgesetzt). Nach dem erfolgreichen Commit bleiben die Daten im Cache erhalten (im Gegensatz zu einer nicht-optimistischen Transaktion). Mit Beginn der nächsten Transaktion können die Daten wieder geändert und wieder committed werden. Es gibt kein automatisches Refresh der Daten im Cache. Ist ein Refresh gewünscht, muss dieser mit `refresh()` ausgelöst werden. Folgende Properties sollten der PersistenceManagerFactory mitgegeben werden, um mit Optimistischen Transaktionen zu arbeiten:

```
javax.jdo.option.Optimistic = true
javax.jdo.option.RetainValues = true
```

Zustände

- Ein Objekt in JDO kann folgende Zustände einnehmen



45

Die sehr genauen Zustandsdefinitionen durch die JDO Spezifikation ermöglichen den Herstellern Implementierungen mit genau vorhersagbarem Verhalten anzubieten.

transient: Nur der Applikation bekanntes Objekt ohne Objekt-ID.

persistent new: Das Objekt bekommt eine ID. Das Objekt ist beim PersistenceManager eingetragen. Dieser ist verantwortlich für die Übertragung des Objektes in die Datenbank.

persistent new deleted: Ein Objekt, das soeben persistent gemacht wurde, in derselben Transaktion aber wieder gelöscht wurde. Beim commit oder rollback wird das Objekt wieder transient.

persistent clean: Ein von der Datenbank gelesenes, aber noch nicht modifiziertes Objekt.

persistent dirty: Ein von der Datenbank gelesenes und modifiziertes Objekt.

hollow: Ein persistentes Objekt, dessen Werte von der Datenbank noch nicht gelesen, oder bei Aufräumen des Cache-Speichers wieder freigegeben wurden. Ein Objekt im Zustand hollow ist eindeutig identifiziert, weil seine ID definiert und geladen ist. Das Resultat einer Abfrage darf eine Collection von Objekten im Zustand hollow sein. Es kann ja bei Bedarf eindeutig auf den Zustand des Objektes zugegriffen werden.

persistent deleted: ein persistentes Objekt, das aus der Datenbank gelöscht werden soll beim commit. Ein Zugriff auf den Objektzustand durch die Applikation ist verboten.

persistent non transactional: Das Objekt ist an sich persistent. Es kann aber auch ausserhalb einer Transaktion gelesen oder geschrieben werden. Das Lesen und Schreiben ist dabei nicht mit dem Zustand in der Datenbank korreliert.

transient transactional clean: Ein transientes Objekt, das einer Transaktionskontrolle unterliegt, innerhalb der Transaktion aber noch nicht geändert wurde.

persistent transactional dirty: Ein transientes Objekt, das einer Transaktionskontrolle unterliegt und innerhalb der Transaktion geändert wurde.

Die Übergänge zwischen den Zuständen sind häufig implizit, zum Beispiel durch Operationen wie commit oder rollback, Zuweisungsoperationen in Java usw.

Die rot markierten Zustände erfordern zum Commit-Zeitpunkt eine Aktion gegenüber der Datenbank.

Zustandsübergänge

- Viele Zustandsübergänge geschehen implizit, beispielsweise durch das Lesen oder Schreiben von Feldern eines Objektes, innerhalb (oder ausserhalb) einer Transaktion.
- Einige Zustandsübergänge werden sinnvollerweise explizit ausgelöst, weil sie Teil der Applikationslogik sind. Dazu gehören beispielsweise:

```

1. PersistenceManager.makePersistent(Object o)
2. PersistenceManager.deletePersistent(Object o)
3. PersistenceManager.makeTransactional(Object o)
4. PersistenceManager.makeNonTransactional(Object o)
5. Transaction.retainValues()
6. PersistenceManager.makeTransient(Object o)
7. PersistenceManager.evict()
8. PersistenceManager.refresh(Object o)

```

46

1. mit `makePersistent()` wird ein transientes Objekt persistent.
2. Mit `deletePersistent()` wird ein persistentes Objekt in der Datenbank gelöscht und in der Applikation ungültig.
3. mit `makeTransactional()` können transiente Objekte dem commit/rollback Mechanismus unterworfen werden. Persistente Objekte im Zustand 'Persistent Nontransactional' können wieder in eine laufende Transaktion "eingekoppelt" werden.
4. mit `makeNonTransactional()` können von der Datenbank gelesene (aber noch nicht geänderte) persistente Objekte aus der Transaktion "ausgekoppelt" werden. Sie können weiterhin gelesen oder modifiziert werden, je nachdem, ob die Datenbank-Implementation 'Persistent Nontransactional' unterstützt oder nicht. Über die `PersistenceManagerFactory` kann die Unterstützung für das NonTransactional-Verhalten durch Aufruf der Methoden `getNonTransactionalRead()` oder `getNonTransactionalWrite()` in Erfahrung gebracht werden.
5. mit `setRetainValues(true)` können persistente Objekte nach dem Commit-Befehl veranlasst werden, im Cache zu bleiben und in den Zustand 'Persistent Nontransactional' überzugehen. Sie können dann ausserhalb einer Transaktion weiterhin gelesen oder geändert werden.
6. mit `makeTransient()` verliert ein persistentes Objekt seine Identität und seine Verbindung zum `PersistenceManager`. Es wird zu einem lokalen Objekt. Die originale Instanz in der Datenbank verbleibt.
7. mit `evict()` können unveränderte, persistente Objekte aus dem Objekt-Cache in der Applikation gelöscht werden.
8. mit `refresh()` wird der Zustand eines Objektes von der Datenbank in die Applikation geladen. Dies kann für

Für obige Funktionen existieren verschiedene Varianten mit verschiedenen Parametern: Ein Objekt, eine Menge von Objekten, alle Objekte usw.

Das genaue Verhalten und die genau möglichen Zustandsübergänge sind dem Zustandsdiagramm auf den folgenden Seiten zu entnehmen.

Zustandsdiagramm für JDO-Objekte: Das Zustandsdiagramm auf den folgenden zwei Seiten gibt sämtliche möglichen Übergänge und Lebenszustände von JDO-Objekten an. Zu beachten ist, dass nur Objekte mit dem Zustand P-dirty oder P-new zum Commit-Zeitpunkt in die Datenbank zurückgeschrieben werden können. Objekte im Zustand P-del werden beim Commit in der Datenbank gelöscht.

- P-new ein bisher transientes, jetzt als persistent markiertes Objekt
- P-clean ein aus der Datenbank gelesenes, aber nicht modifiziertes Objekt
- P-dirty ein aus der Datenbank gelesenes und modifiziertes Objekt
- Hollow ein (noch) leeres Objekt mit gegebener OID und ev. Primärschlüssel, jedoch ohne Daten. Die Daten können jederzeit anhand der OID aus der DB gelesen werden.

current state	Transient	P-new	P-clean	P-dirty	Hollow
method					
makePersistent	P-new	unchanged	unchanged	unchanged	unchanged
deletePersistent	error	P-new-del	P-del	P-del	P-del
makeTransactional	T-clean	unchanged	unchanged	unchanged	P-clean
makeNontransactional	error	error	P-nontrans	error	unchanged
makeTransient	unchanged	error	Transient	error	Transient
commit retainValues=false	unchanged	Hollow	Hollow	Hollow	unchanged
commit retainValues=true	unchanged	P-nontrans	P-nontrans	P-nontrans	unchanged
rollback restoreValues=false	unchanged	Transient	Hollow	Hollow	unchanged
rollback restoreValues=true	unchanged	Transient	P-nontrans	P-nontrans	unchanged
refresh with active datastore transaction	unchanged	unchanged	unchanged	P-clean	unchanged
refresh with active optimistic transaction	unchanged	unchanged	unchanged	P-nontrans	unchanged
evict	n/a	unchanged	Hollow	unchanged	unchanged
read field outside transaction	unchanged	impossible	impossible	impossible	P-nontrans
read field with active Optimistic transaction	unchanged	unchanged	unchanged	unchanged	P-nontrans
read field with active Datastore transaction	unchanged	unchanged	unchanged	unchanged	P-clean
write field or makeDirty outside transaction	unchanged	impossible	impossible	impossible	P-nontrans
write field or makeDirty with active transaction	unchanged	unchanged	P-dirty	unchanged	P-dirty
retrieve outside or with active optimistic transaction	unchanged	unchanged	unchanged	unchanged	P-nontrans
retrieve with active datastore transaction	unchanged	unchanged	unchanged	unchanged	P-clean

- T-clean ein transientes, nicht modifiziertes Objekt unter Transaktionskontrolle.
- T-dirty ein transientes, modifiziertes Objekt unter Transaktionskontrolle.
- P-new-del in der gleichen Transaktion als persistent, dann wieder transient markiertes Objekt. Unterschied zu P-del: Beim Rollback wird P-new-del wieder transient mit exakt dem Zustand wie vor P-new. P-del wird beim Rollback Hollow (ungültig).
- P-del ein persistentes, als gelöscht markiertes Objekt.
- P-nontrans ein persistentes Objekt, das nicht unter Transaktionskontrolle steht. Es kann lokal gelesen und geändert werden, ohne Auswirkung auf die Datenbank.

current state	T-clean	T-dirty	P-new-del	P-del	P-nontrans
method					
makePersistent	P-new	P-new	unchanged	unchanged	unchanged
deletePersistent	error	error	unchanged	unchanged	P-del
makeTransactional	unchanged	unchanged	unchanged	unchanged	P-clean
makeNontransactional	Transient	error	error	error	unchanged
makeTransient	unchanged	unchanged	error	error	Transient
commit retainValues=false	unchanged	T-clean	Transient	Transient	unchanged
commit retainValues=true	unchanged	T-clean	Transient	Transient	unchanged
rollback restoreValues=false	unchanged	T-clean	Transient	Hollow	unchanged
rollback restoreValues=true	unchanged	T-clean	Transient	P-nontrans	unchanged
refresh	unchanged	unchanged	unchanged	unchanged	unchanged
evict	unchanged	unchanged	unchanged	unchanged	Hollow
read field outside transaction	unchanged	impossible	impossible	impossible	unchanged
read field with optimistic transaction	unchanged	unchanged	error	error	unchanged
read field with active datastore transaction	unchanged	unchanged	error	error	P-clean
write field or makeDirty outside transaction	unchanged	impossible	impossible	impossible	unchanged
write field or makeDirty with active transaction	T-dirty	unchanged	error	error	P-dirty
retrieve¹ outside or with active optimistic transaction	unchanged	unchanged	unchanged	unchanged	unchanged
retrieve¹ with active datastore transaction	unchanged	unchanged	unchanged	unchanged	P-clean

¹ganzes Objekt

retainValues: Zurückhalten des Objektzustandes im Cache nach dem Commit als P-nontransactional.

restoreValues: Alten Objektzustand im Cache nach dem Rollback als P-nontransactional behalten.

Transaction Isolation Levels

- *Es existieren keine API-Funktion zum Setzen des Isolation Levels.*
- Die Spezifikation empfiehlt, dass sich eine portable Applikation nur auf den Isolation Level READ COMMITTED abstützt:

Portable applications must not depend on isolation levels stronger than read-committed provided by the underlying datastore. Some fields might be read at different times by the JDO implementation, and there is no guarantee as to read consistency compared to previously read data. A JDO persistence-capable instance might contain fields instantiated by multiple datastore accesses, with no guarantees of consistency (read-committed isolation level).

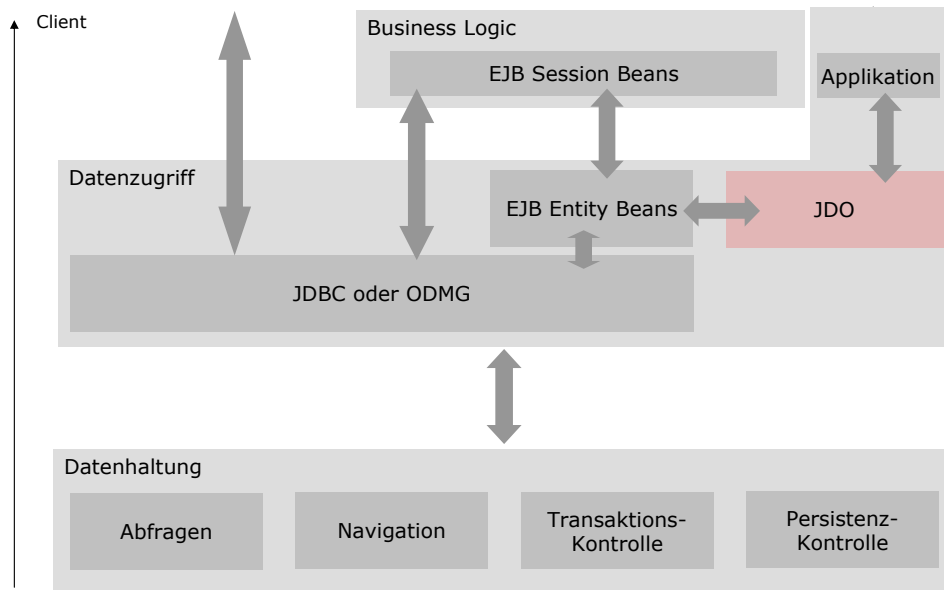
49

Der Isolation Level READ COMMITTED ist problematisch für bestimmte Anwendungen. Es können Lost Updates auftreten, wenn mehrere unabhängige JDO- oder andere Anwendungen auf die Datenbank zugreifen. Siehe Technologie-Memo über Kodo/JDO Isolation Levels.

VI Umfeld

- EJB Integration
- Produkte
- Schlussbeurteilung

Einordnung von JDO im J2EE Umfeld



51

JDO lässt sich als Datenabstraktion verstehen, welche hauptsächlich den Transportmechanismus von Objekten von und zur Datenbank versteckt. Persistente und transiente Objekten verhalten sich bezüglich Navigation und Zugriff mit Java-Methoden (Zuweisungen, get/set Methoden auf den Standardklassen) völlig identisch. Ausserdem werden Klassen für das Durchführen von Abfragen (JDOQL), die Transaktionskontrolle (z.B. beginn, commit, rollback), und die Persistenzkontrolle (z.B. makePersistent(), deletePersistent()) bereitgestellt.

Bei JDBC ist die Applikation für den Transport von Daten und die Umwandlung von Datensätzen in Objekte verantwortlich.

Bei EJB-Entity Beans ist zusätzlich ein Abstraktion des Ortes möglich (Remote-Zugriff), bei JDO befindet sich der benützende Client im selben Prozess wie die JDO-Dienste.

Integration von JDO und EJB

- JDO eignet sich für den Datenzugriff in
 - Stateless und Statefull Session Beans mit CMT
 - Stateless und Statefull Session Beans mit BMT
 - Entity Beans mit BMP
 - Entity Beans mit CMP (für Middleware-Hersteller)
- JDO Objekte eignen sich nicht für Remote-References, wie EJB's.
- Der optimale Einsatz von JDO spielt sich dort ab, wo bei EJB's mit den Local-Interfaces gearbeitet wird.

Integration von JDO und EJB

- Die Schnittstelle zwischen Bean-Entwickler und Container sind im Wesentlichen `PersistenceManagerFactory` und `PersistenceManager`
- Ein Session Bean soll via JNDI auf eine vom Container bereitgestellte `PersistenceManagerFactory` zugreifen können, beispielsweise über `"java:comp/env/jdo/myPMF"`
- Aus der `PersistenceManagerFactory` holt das Session Bean in jeder Business-Methode einen `PersistenceManager` ab.
- Die Factory muss so implementiert sein, dass auf die Transaktionsattribute im Deployment Descriptor Rücksicht genommen wird (`Required`, `RequiresNew`, `Mandatory`, ...)

Session Beans

```
public class MySessionBean implements javax.ejb.SessionBean
{
    protected PersistenceManagerFactory pmf;
    public void myBusinessMethod {
        PersistenceManager pm = pmf.getPersistenceManager();
        // mit pm arbeiten, bei BMT auch Transaktionsmethoden
        pm.close();
    }
    public void ejbActivate()
    {
        javax.naming.Context cntx = new InitialContext();
        pmf = (PersistenceManagerFactory)
            cntx.lookup("java:comp/env/jdo/myPMF");
    }
    ...
}
```

54

Kapitel 16.1, JDO-Spezifikation: Jedes Session Bean soll mit einer PersistenceManagerFactory assoziiert sein. Diese Assoziation ist bei der Aktivierung des Beans zu erstellen.

Die Methode `pmf.getPersistenceManager()` muss innerhalb jeder Business-Methode aufgerufen werden, damit die `PersistenceManagerFactory` entscheiden kann, inwieweit eine bestehende Transaktion für dieses Bean verwendet und an den `PersistenceManager` übergeben werden kann. Je nach gesetztem Transaktionsattribut im Deployment-Deskriptor muss die `PersistenceManagerFactory` dem `PersistenceManager` jedesmal eine neue Transaktion mitgegeben oder kann auf eine bestehende zurückgreifen.

Container Managed Transactions CMT

Der Transaktions-Manager innerhalb des Applikationsservers führt selbstständig `begin-` und `commit-`Methoden für die im `PersistenceManager` eingebettete Transaktion durch. Das explizite Aufrufen von `PersistenceManager.commit()`, resp. den anderen Transaktionsmethoden wäre in diesem Fall ein Fehler.

Bean Managed Transactions BMT

Über den `PersistenceManager` kann das Transaktionsobjekt mit `currentTransaction()` abgeholt werden. Dieses ermöglicht dann den Aufruf von `begin()`, `commit()`, `rollback()`. Im Fall von CMT führt der Aufruf einer dieser Methoden zu Auswerfen einer `JDOUserException`.

Verteilte Transaktionen

Verteilte Transaktionen werden über die `PersistenceManagerFactory` verwaltet. Die Factory besitzt die notwendige Information, mit welchen anderen Factory-Objekten verteilte Transaktionen zu realisieren sind. Die zu erzeugenden `PersistenceManager`-Objekte (respektive ihre Transaktions-Objekte) sind dann über die Factorys an die verteilte Transaktion gebunden. Das 2PC Protokoll wird vom Container (bei CMT) oder vom Entwickler innerhalb der `SessionBean-Businessmethoden` (bei BMT) ausgelöst.

Entity Beans, BMP

Notwendige Instanzvariablen

```
PersistenceManagerFactory pmf
PersistenceManager pm
Object jdoInstance jdoInst
```

Methode

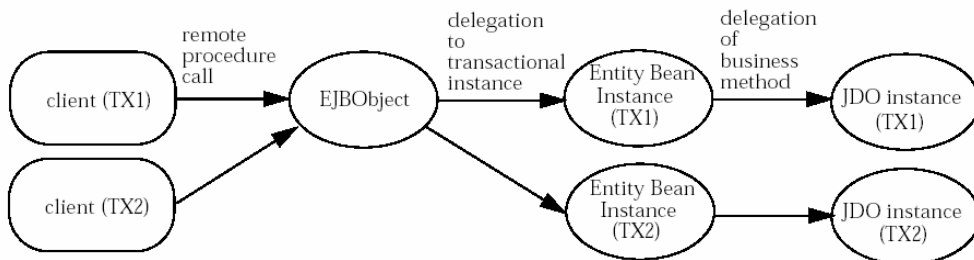
Wichtigste Aufgabe

setEntityContext()	Lookup für pmf
ejbCreate()	erzeugt jdoInst, pm.makePersistent(jdoInst)
ejbRemove()	pm.deletePersistent(jdoInst)
ejbActivate()	pm = pmf.getPersistenceManager(), jdoInst = getObjectById(...)
ejbPassivate()	jdoInst = null
ejbLoad()	-
ejbStore()	-
ejbFind()	JDO-Objekt suchen, ObjectID zurückliefern

55

Alle Methoden, welche den `PersistenceManager` benötigen, müssen einen solchen unmittelbar vor Gebrauch von der `PersistenceManagerFactory` abholen.

Entity Beans



- Bei Container Managed Persistence CMP arbeitet die Referenzimplementation von J2EE 1.4 mit JDO

Schlussbeurteilung

- Vermeidet auf Ebene Applikationsentwicklung den Strukturbruch zwischen relationaler und objektorientierter Welt. 100% objektorientiert bezüglich Persistenz. 100% Java orientiert.
- Kapselt gleichberechtigt zwei ausgereifte DB-Technologien (OODB, RDB).
- Produkte und Mapping-Tools sind vorhanden und werden weiterentwickelt.
- Alternatives oder integriertes Konzept für EJB Entity Beans.

VII OO-Datenbanken

- Konzepte und Eigenschaften
- Serverarchitekturen
- Der ODMG-Standard
- Produkte

58

OO-Datenbanken entstanden Anfang der 90er Jahre und sind zu einer ausgereiften Technologie herangewachsen. Bis heute haben OODMBS im Vergleich zur relationalen Welt nur eine kleine Position auf dem Markt. Möglicherweise waren viele Konzepte (zum Beispiel OQL) zu ambitiös. Heute ist eine ausgeprägte Tendenz zur Abstraktion des Data-Layers im Bereich Business Logic/Middle Tier erkennbar. OO-Datenbanken gewinnen dadurch wieder wesentlich an Attraktivität, besonders wenn man an das extrem aufwendige Mapping von relationalen Strukturen zu Objekten im Rahmen von Applikationsservern denkt.

Eigenschaften, Datenmodell

- Datenmodell richtet sich an OO-Konzepten aus
 - Vererbung
 - Kapselung
 - Komposition, Aggregation
 - Objektidentität
- Im Idealfall sind die Objekte in der Datenbank über verschiedene Programmiersprachen zugreifbar.
- In der Praxis wird mit Vorteil nur *eine* Programmiersprache verwendet. Standardisiert sind C++ / Smalltalk / Java

Eigenschaften, Persistenz / 1

- Der Entwickler muss sich nicht um den Transport von Objekten von und zur Datenbank kümmern.
- Beim Beginn der Zusammenarbeit mit der Datenbank wird ein Wurzelobjekt in die Applikation geholt, z.B. ein `Vector`.
- Ausgehend vom Wurzelobjekt sind alle anderen Objekte, wie bei einem normalen Programm, erreichbar.
- Neue Objekte, die an das Wurzelobjekt gebunden werden, gehören automatisch zur Datenbank.

Eigenschaften, Persistenz / 2

- Jeder Art von Objekt kann einen Namen bekommen.
- Jedes Objekt, das selbst einen Namen hat oder über beliebig viele Navigationsschritte erreichbar ist, gehört zur Datenbank.
- Wichtiger Begriff: *Persistence bei Reachability*.
- Eine spezielle Löschoption ist nicht erforderlich: Jedes nicht mehr erreichbare kann vom Garbage Collector eingesammelt werden (Gleiches Prinzip wie Java selbst). Eine spezielle Löschoption ist allerdings im Standard vorhanden.

61

Bei JDO ist man von diesem Prinzip wieder etwas abgewichen, indem Objekt explizit persistent gemacht werden und auch explizit wieder gelöscht werden müssen.

Persistenz, Beispiel 1 in Java

- Erzeugen und Benennen eines Set-Objektes:

```
public static void main( String[] args ) {  
    Database db;  
    Transaction tra;  
    try {  
        // Verbindung zur DB herstellen  
        db = Database.open( "basis", Database.openReadWrite );  
        // Transaktion starten  
        tra = new Transaction();  
        tra.begin();  
        // Set erzeugen und mit Namen versehen  
        DSet personen = new SetOfObject();  
        db.bind( personen, "AllePersonen" );  
        tra.commit();  
        db.close();  
    }  
}
```

Damit wird eine Menge von Objekten (Personen) in der Datenbank abgelegt.

Persistenz, Beispiel 2 in Java

- Abfragen und Verwalten von Objekten

```
public static void main( String[] args ) {  
    Database db;  
    Transaction tra;  
  
    try  
        // Verbindung zur DB herstellen  
        db = Database.open( "basis", Database.openReadWrite );  
  
        // Transaktion starten  
        tra = new Transaction();  
        tra.begin();  
        // Persistente Objektmenge referenzieren  
        DSet personen = (DSet) db.lookup( "AllePersonen" );  
        ...  
    }  
}
```

Persistente und benannte
Objektmenge

```
// Neue Person erzeugen
Person p;
p = new Person( name );
p.setName( "Joggeli" );
personen.add( p );

// Personen abfragen
// Personen abfragen
Iterator it = personen.iterator();
while ( it.hasNext() ) {
    p = (Person) it.nextElement();
    System.out.println( p.getName() );
}

// Transaktion abschliessen
tra.commit();
db.close();
}
```

Persistenz, Beispiel 3 in Java

- Entfernen von Objekten

```
public static void main( String[] args ) {
    Database db;
    Transaction tra;
    try {
        // Persistente Objektmenge referenzieren
        DSet personen = (DSet) db.lookup( "AllePersonen" );
        // Personen abfragen
        Iterator it = personen.iterator();
        while ( it.hasNext() ) {
            it.remove( p );
        }
        // Transaktion abschliessen
        tra.commit();
        db.close();
    }
    catch ( Exception e ) {
        tra.abort();
    }
}
```

65

Eigenschaften, Arbeiten mit Objekten

- Im Gegensatz zu relationalen Datenbanken werden Objekte über die Mittel der Programmiersprache *modifiziert*.
- Eine Abfragesprache (OQL) existiert, dient jedoch hauptsächlich dem *Suchen* von Objekten.
- Die Konzepte Trigger, Stored Procedure, Constraints, unique key rücken bei OODB in den Hintergrund, weil der Zugriff auf Objekte über Methoden gekapselt ist.

Object Identifiers OID / 1

- Das Arbeiten mit Object Identifiers ist eine Kernaufgabe von OODBMS
 - Jedes Objekt besitzt eine eindeutige Identifikation, den *Object Identifier* (OID).
 - Der OID stellt die Beziehung zwischen einem Objekt seitens Applikation/Cache und seitens Datenbank her.
 - OIDs müssen systemweit eindeutig, unveränderlich, orts- und zustandsunabhängig sein (LOID).
 - Einmal vergebene OID's dürfen auch nach dem Löschen des Objektes nie mehr wiederverwendet werden dürfen, damit eine bereits beim Client abgelegte Objektreferenz nicht plötzlich auf ein neues Objekt zeigt.

Object Identifiers OID / 2

- Kernstück der Arbeit mit OID ist eine serverseitige Übersetzungstabelle, genannt Persistent Object Table POT:
 - Memory-resident
 - Recovery-fähig
 - Unterliegt Locking
 - Grösse ~ Anzahl Objekte in der DB
 - Muss hocheffizient verwaltet werden

Persistent Object Table (POT)

OID	Storage Adress

Enhancement, Prinzip

- Einbau der Datenbank-Funktionalität
- Modifikation des Java-Byte Codes:
 - Einfügen von Methoden anstelle von Feld-Zugriffen
 - Ergänzung der Klasse um OID, OID von Komponenten etc.
- Konfigurationsfile steuert das Postprocessing
- Einen genaueren Einblick erhält man durch Disassemblieren des generierten Java-Codes mit `javap -c`

Enhancement, Code-Beispiele

```
String s = p1.name;           // vor enhance  
String s = p1.getName();    // nach enhance
```

```
p1.name = "Arthur";         // vor enhance  
p1.setName( "Arthur" );    // nach enhance
```

```
p2 = p1.partner;           // vor enhance  
p2 = p1.getPartner();     // nach enhance
```

(Das Feld partner sei ein Komponentenobjekt von Person)

Enhancement, Was passiert beim Zugriff auf Objekte

```
p1 = db.lookup( "Person" );
```

Anhand des Namens die OID in der DB suchen und leeres Personenobjekt erzeugen

```
p1.getName();
```

Lesesperre setzen, String aus DB holen, ausliefern

```
p1.setName( "Arthur" );
```

Schreibsperre setzen, dirty flag setzen

```
p2 = p1.getPartner();
```

Anhand der OID des Partnerobjektes ein leeres Personenobjekt erzeugen und in der Objekttable im Cache eintragen

```
tra.commit();
```

Modifizierte Objekte suchen und zurückschreiben.

Object Query Language OQL

- Ähnlich zu SQL
- Aufruf aus Programmiersprache heraus
- Im Gegensatz zu SQL keine Modifikations-Befehle
- Methodenaufruf möglich
- Navigation, Join, Gruppierung, Subqueries
- Optimierungsmöglichkeiten mit Indices

- Aus heutiger Sicht: Ähnlichkeit mit SQL-J

OQL - Beispiele

```
select k
from k in privatkunden
where k.name = "Meyer"
and "Peter" in k.vornamen
```

```
select distinct k.name
from k in privatkunden, b in k.bestellungen
where b.betrag > 1000000
order by k.adresse.strasse
```

```
select k.name, k.vornamen
from k in privatkunden
where k.hasCredit( 10000 )
```

Transaktionsverarbeitung

- ACID-Regel gilt auch für OODBMS
- Caching von Objektzuständen beim Client
- Recovery durch reines Redo-Logging
- Meist optimistisches Concurrency Control möglich
- Häufig auch nicht transaktioneller Zugriff auf Objekte möglich
- Verteilte Transaktionen, Replikation, Event-Mechanismen, Lange Transaktionen, Versionierung von Objekten und Schema, Fail-Over Server, Clustering, Monitoring Tools gehören häufig zum Produktangebot.

74

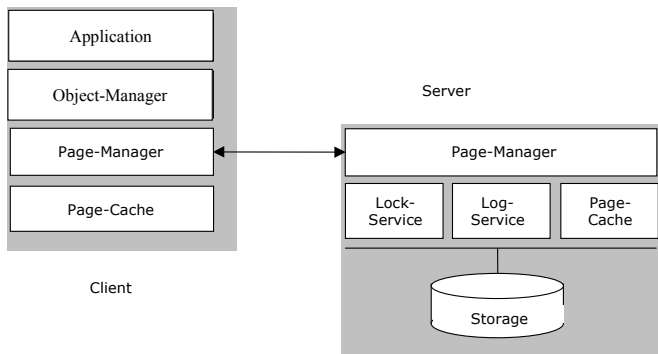
Versant stellt beispielsweise alle im letzten Punkt genannten Eigenschaften zur Verfügung.

Server-Architekturen

- Im Gegensatz zur relationalen Datenbank, sind objektorientierte Datenbanken in zwei grundsätzlich verschiedenen Architekturen realisiert:
- *Page-Server* "verstehen" nichts von Objekten. Sie liefern lediglich I/O Pages (2ⁿ KB) an die Clients aus und sind zusätzlich für die **A**tomarität, **I**solation und **D**auerhaftigkeit von Transaktionen, basierend auf I/O-Pages, zuständig.
- *Object-Server* liefern aufgrund einer Client-Anfrage einzelne Objekte aus. Daneben sind sie für die Einhaltung der **ACID**-Transaktionseigenschaften und allenfalls für die Durchführung von OQL-Abfragen und Objektmethoden verantwortlich.

Page-Server / 1

- Verbindungs- und Transaktionskontrolle
- Sperren setzen und freigeben auf angeforderte Pages
- Ausliefern von Pages aufgrund einer Page-Number
- Pages empfangen und im Logfile protokollieren
- Page-Cache bewirtschaften



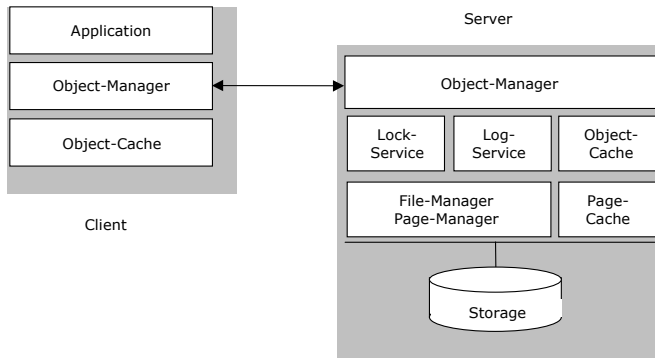
Produkte
 Objectivity/DB
 ObjectStore

Page-Server / 2

- Tendenz zu folgenden Eigenschaften
 - Prozessorabhängiges Page-Format für Daten und Code
 - effizienter Transfer von vielen kleinen Objekten
 - Page Level Locking
 - Fat Clients, I/O bounded Server
 - Es werden mehr Objekte als notwendig transferiert
 - Ausführung von OQL und Methoden im Client
- Einsatz für
 - grosse Datenmengen
 - Kleiner Concurrent Use gemeinsamer Daten
 - Applikationen mit rechenintensiven Aufgaben

Object-Server / 1

- Verbindungs- und Transaktionskontrolle
- Sperren auf Objekten verwalten
- Übersetzen von OID in physische Adressen
- Objekte empfangen und im Logfile protokollieren
- Page- und Object-Cache bewirtschaften
- OQL Abfragen und Objektmethoden durchführen



Produkte
Versant
POET

Object-Server / 2

- Tendenz zu folgenden Eigenschaften
 - Abstraktes Datenformat
 - häufiges Kopieren zwischen Speichermedium und Applikation
 - Object Level Locking
 - Thin Clients, I/O- und CPU-bounded Server
 - inkonsistente Daten zwischen Applikationscache und DB
 - Ausführung von Abfragen beim Server
- Einsatz für
 - kleine und mittlere Datenmengen
 - Hoher Concurrent Use gemeinsamer Daten

Der ODMG Standards

- Object Database Management Group
- ODMG 1.0 1993, ODMG 2.0 1997, ODMG 3.0 2000
- ODMG-Gruppe wurde 2004 aufgelöst
- Volle Kraft Richtung JDO!
- OODB-Technologie bleibt bestehen
- Komponenten des ODMG-Standards

- Object Model OM
- Object Definition Language ODL
- Object Query Language OQL
- Bindings für C++ / Java / Smalltalk

Language Bindings

- Definieren das exakte Aussehen (Name, Typ, Parameter) von Klassen, Methoden und Datentypen, die für den Umgang mit persistenten Objekten und Objektmengen, sowie für das Öffnen von Datenbanken und das Durchführen von Transaktionen notwendig sind.
- Die Language Bindings sind Voraussetzung für die automatische Übersetzung von ODL in konkrete Programmiersprachen.

Language Binding Java, Collections

- ODMG Collection Interfaces

```
interface DCollection { ... }
interface DSet extends DCollection, java.util.Set { ... }
interface DList extends DCollection, java.util.List{ ... }
interface DArray extends DCollection, java.util.List{ ... }
interface DBag extends DCollection { ... }
interface DMap extends java.util.List{ ... }
```

- Die **Java Collection Interfaces** unterstützen Methoden für das Einfügen, Löschen und Konvertieren von Elementen.
- Die **ODMG Collection Interfaces** enthalten Methoden für Abfragen (`select()`, `query()`) nach Elementen und Mengenoperationen (`intersect()`, `difference()` `union()`, `subsetOf()`)
- Rückbeziehungen, Extents und Keys werden im Java-Binding nicht unterstützt.

Language Binding Java, Datenbank

- Klassen für die Datenbank- und Transaktionsverwaltung:

```
public class Database {  
    public static Database open( String name, int mode ) throws ODMGException;  
    public void close() throws ODMGException;  
    public void bind( Object o, String name );  
    public void unbind( String name ) throws ObjectNameNotFoundException;  
    public Object lookup( String name ) throws ObjectNameNotFoundException;  
}
```

```
public class Transaction {  
    public void begin();  
    public void commit();  
    public void abort();  
    public void checkpoint();  
    public void join();  
    public void leave();  
    public static Transaction current();  
    public boolean isOpen();  
    public void lock( Object o, int mode );  
}
```

Language Binding Java, Queries

- Klasse für das Durchführen von OQL Queries:

```
interface OQLQuery {  
    public void create( String query ) throws InvalidQueryException;  
    public void bind( Object param ) throws QueryParameterCountInvalidException;  
    public Object execute() throws QueryException  
}
```

- Zu Beachten: Die Syntax des Query-Strings ist ebenfalls im ODMG Standard als BNF definiert.

Produkte

- Die Grossen, welche den Namen OODB verdienen
 - Versant
 - POET
 - Objectivity
 - ObjectStore

Unterschiede JDO / ODMG

- + definierbare ID-Typen (Data Store oder Application)
- + Klar definierte Objektgefäße: Extents
- + Nur Java Standard-Collections (keine DSet, DList, ...)
- + Gleichzeitiger Gebrauch verschiedener DB-Systeme
- + Runtime Exceptions statt Checked Exceptions
- + Instance Callbacks im Standard
- + Sehr genaue Zustände und Übergänge für Objekte
- + Gleichermassen für OODB und RDB als Persistenz-Sockel
- Keine 'Persistence by Reachability' mit Root-Elementen
- Isolation Levels und explizites Locking fehlen
- Query Language JDOQL semantisch schwach
- Proprietär bezügl. Programmiersprache: Java