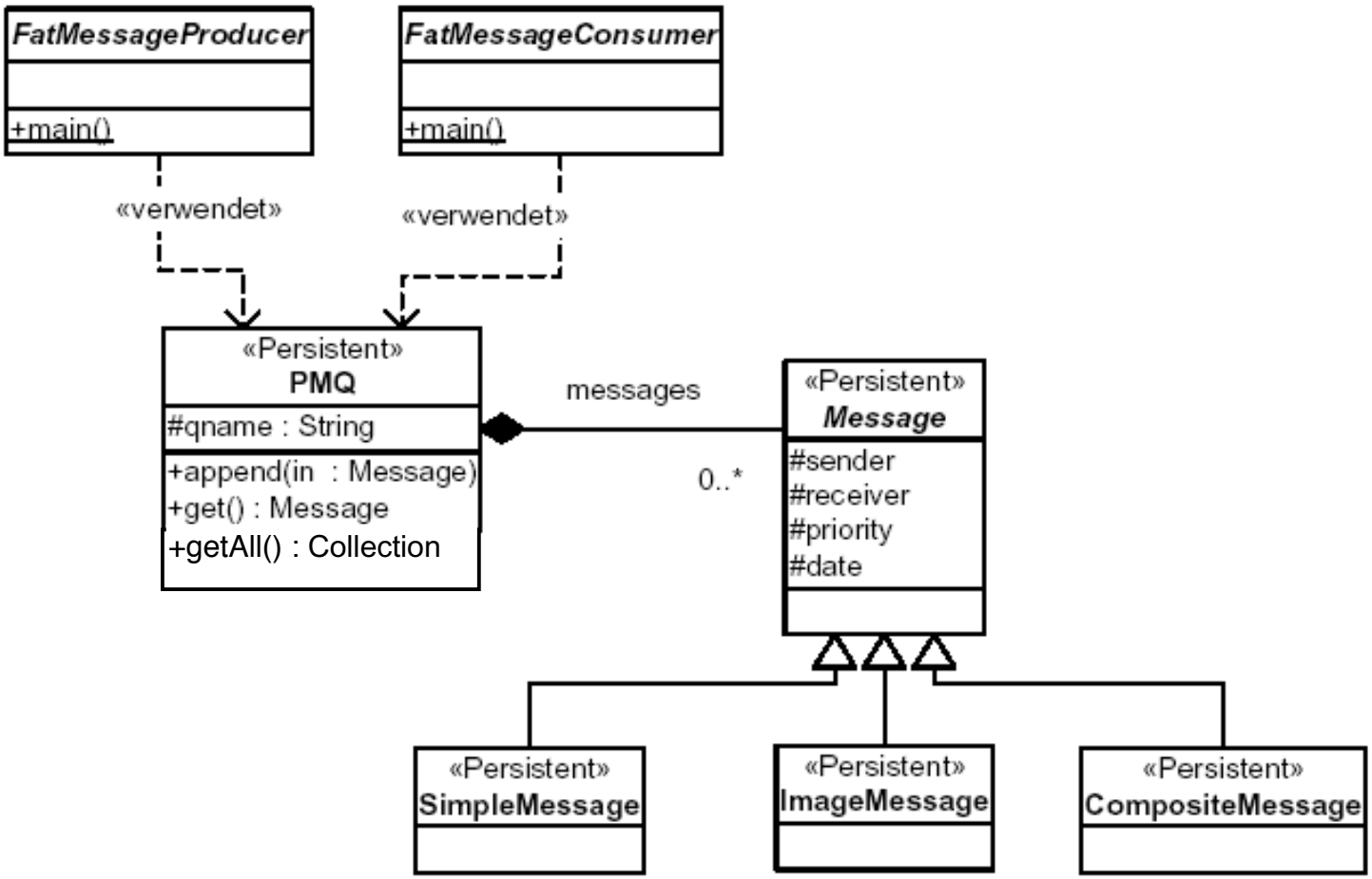
# Objektorientierte Datenbanken

Ralf Möller, FH-Wedel

- Beim vorigen Mal:
  - Java Data Objects Teil 2, Queries
- Heute:
  - Java Data Objects Teil 3, Objektidentität
- Lernziele:
  - Grundlagen der Programmierung persistenter Objekte
  - Java Data Objects

# Beispiel


FatMessageProducer
+main()
FatMessageConsumer
+main()
«verwendet»
«verwendet»
«Persistent»
PMQ
#qname : String
+append(in : Message)
+get() : Message
+getAll() : Collection
messages
0..*
«Persistent»
Message
#sender
#receiver
#priority
#date
«Persistent»
SimpleMessage
«Persistent»
ImageMessage
«Persistent»
CompositeMessage

# JDO Identities

1. Jedes persistente Objekt muss dauerhaft und eindeutig identifizierbar sein. Es besitzt eine Objekt-ID.

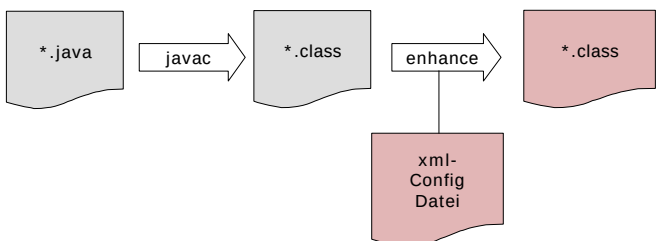
2. Aufgrund seiner Identität kann ein persistentes Objekt in der Datenbank aufgefunden werden, z.B. mit `PersistenceManager.getObjectById(Object oid)`

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

# JDO Enhancer

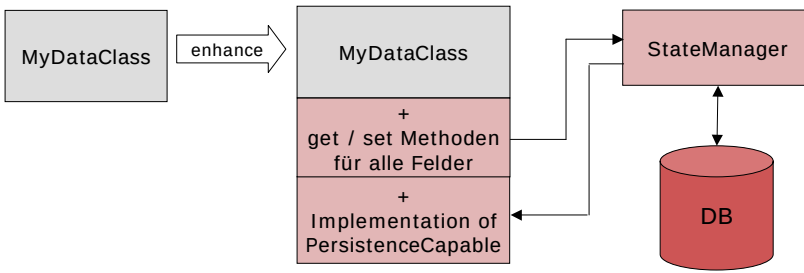
- Die Umprogrammierung des Java-Codes (Enhancement) findet auf Ebene Bytecode, also mit den Class-Files statt.



# JDO Enhancer

- In Java-Klassen, welche mit persistenten Objekten *arbeiten*, werden die Operatoren, welche diese Objekte verändern, in Methoden*aufrufe* umprogrammiert.

- In Java-Klassen, welche *selber persistente Objekte* repräsentierten, werden die entsprechenden Methoden durch den Enhancer *implementiert*.



# XML Konfigurationsdatei

- Das Konfiguartionsfile beschreibt die persistenten Klassen.
- Wird für den Postprocessor (Enhancer) und zur Laufzeit der Applikation benötigt.

Beispiel: Datei pmq.jdo

```
<jdo>
  <package name="pmq">
    <class name="PMQ"/>
    <class name="Message" identity-type="datastore"/>
    <class name="ImageMessage"
            persistence-capable-superclass="pmq.Message"/>
      <class name="SimpleMessage"
            persistence-capable-superclass="pmq.Message"/>
      <class name="CompositeMessage"
            persistence-capable-superclass="pmq.Message"/>
 </package>
</jdo>
```

# XML Konfigurationsdatei

1. In der Konfigurationsdatei *können* für die einzelnen Felder einer Klasse weitere logische Angaben gemacht werden:
   1. Gehört ein Feld zum Primärschlüssel (nur wenn `identity-type = "application"` )
   2. ist das Feld `persistent`, `transactional`, none
   3. von welchem Datentyp (Klasse) sind die Elemente, wenn es sich um ein Collection- oder Map-Feld handelt

2. Speicherbezogene Angaben:
   1. gehört das Feld zur `default-fetch-group`
   2. soll es in der Datenbank als Teil des umgebenden Objektes gespeichert werden (`embedded`)

# 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 )
```

# 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() );

// später: hole Objekt aus der Datenbank
PMQ pmq = (PMQ) pm.getObjectById( "oidstring", 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.

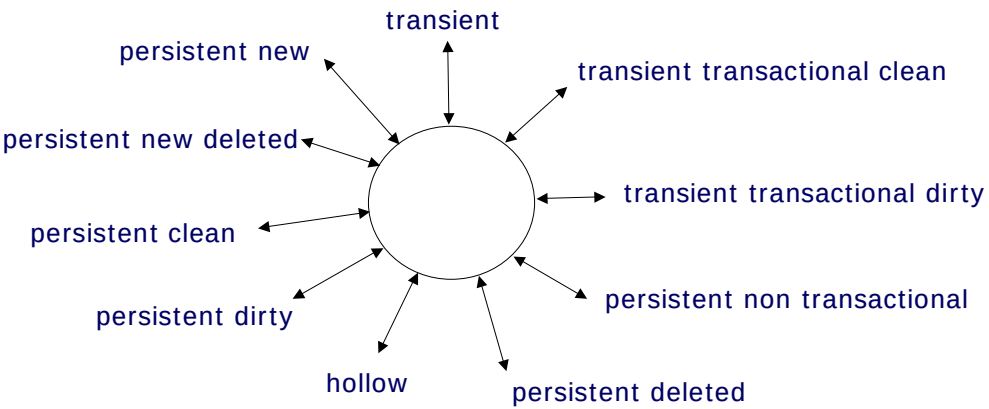
# Transaktionskontrolle

# Transaktionsmodi

- JDO stellt mehrere Transaktionsmodi zur Verfügung
  1. Persistent Transactional
  2. Persistent Nontransactional
  3. Transient Transactional
  4. Optimistic Transactions
- JDO definiert sehr genau die Zustände und Zustands-übergänge von Objekten in und zwischen den verschiedenen Transaktionsmodi

# Zustände

- Ein Objekt in JDO kann folgende Zustände einnehmen



# Explizite 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)
```

# JDO Persistenzmodell

- Grundsatz / Anforderungen der Datenbank
- Enhancer
- Konfigurationsdate
- 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, festschreiben oder zurücksetzen
  - Objekte gezielt aus der Datenbank holen via Query oder Objekt-ID
- Für den programmiersprachlichen Umgang mit Objekten gilt:

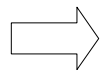
  Arbeiten mit Datenbankobjekten = Arbeiten mit Java-Objekten

# 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 Code von JDO-Applikationen wird modifziert: Dereferenzierungs-, Zuweisungsoperatoren usw. 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.)

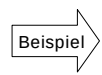
# First Class / Second Class Objects

- First Class Objekt (FCO)
  - hat eine ObjektID
  - ist shareable
  - ist Transfereinheit



als FCO deklarierte und benutzerdefinierte Klasse

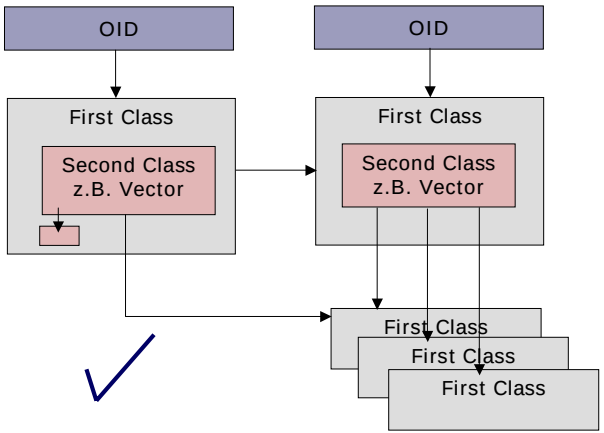
- Second Class Objekt (SCO)
  - hat keine ObjektID
  - ist non-shareable
  - gehört immer zu einem First Class Objekt
  - hat Wertsemantik

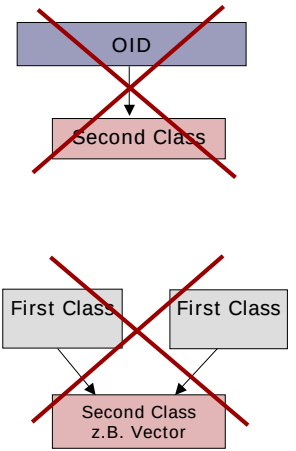



```
String
Date
Vector
Array
...
```

weitere, nicht als FCO deklarierte, serialisierbare Klassen

# First Class / Second Class Objects ff

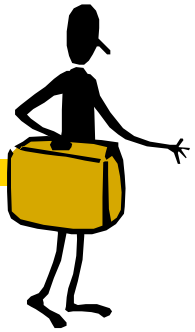


# `static`,`transient`, `final`

- 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 initalisiert werden dürfen, kann sich dadurch ein unerwartetes Verhalten ergeben, beim Laden von Objekten aus der Datebank.
- Ein `final static` Felder zeigt dasselbe Verhalten wie in einer allgemeinen Java-Applikation

# Zusammenfassung, Kernpunkte



- Persistenz
- Einführung Java Data Objects (Teil 3)
- Objektidentität, Transaktionen, Zustände, ...

# Was kommt beim nächsten Mal?

- ODMG-Standard