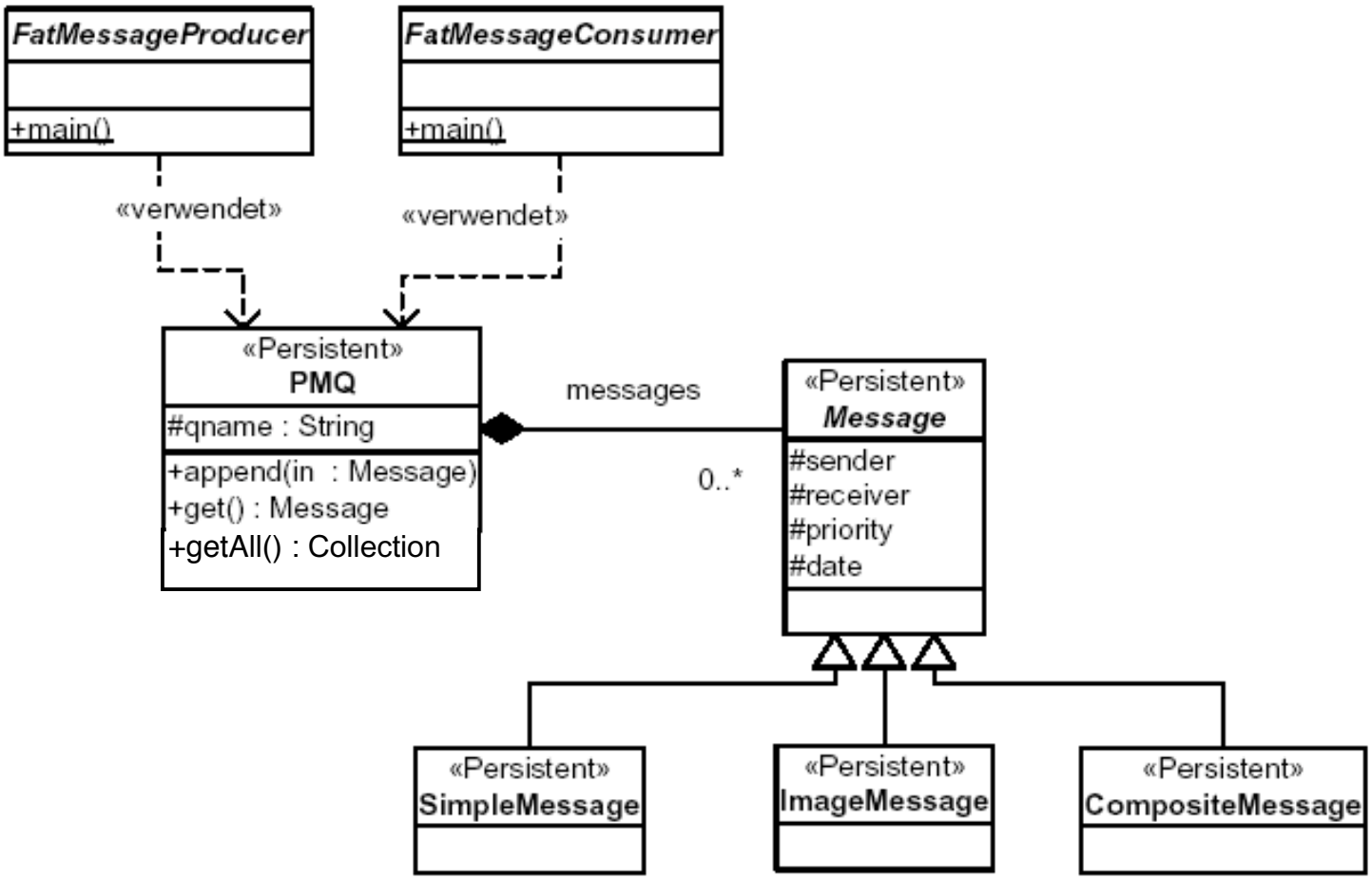
Objektorientierte Datenbanken

Ralf Möller, FH-Wedel

- Beim vorigen Mal:
  - Java Data Objects Teil 1
- Heute:
  - Java Data Objects Teil 2
- Lernziele:
  - Grundlagen der Programmierung persistenter Objekte
  - Java Data Objects

Arbeiten mit JDO

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

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();
...
```

Automatische Persistenz

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

```
...
PMQ pmq = new PMQ(  name );
pm.makePersistent( pmq );
Message m = new SimpleMessage( ... );
pmq.append( m );
...
```


m wird persistent !

2. Ein persistentes Objekt wird nur durch `PersistenceManager.deletePersistent()` wieder aus der Datenbank entfernt

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 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, gegebenenfalls auch ihrer Unterklassen.
2. Das Konstruieren des Extents beinhaltet noch nicht das Holen der Objekte aus der Datenbank.
3. Der Extent stellt einen Iterator zur Vefügung. 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*.

Extents

- Enthalten alle Objekte einer Klasse (d.h. auch Objekte von Unterklassen)
- Werden automatisch vom DBMS gepflegt
- Extent PersistenceManager.getExtent(Class, boolean)
- Iterator Extent.iterator()

Extent, Beispiel

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

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)

Query, Beispiel 1

```
...
Extent e = pm.getExtent( PMQ.class, false );
Query query = pm.newQuery();
query.setCandidates( e );
query.setClass( PMQ.class );
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 ...
```

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);
```

Query, Beispiel 3

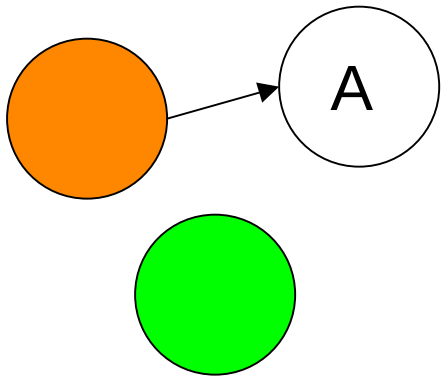
```
Extent e = pm.getExtent( PMQ.class, false );
Query query = pm.newQuery();
query.setCandidates( e );
query.setClass( PMQ.class );
query.setFilter("messages.contains(m)
                && m.priority <= p_prio" );
query.declareVariables( "Message m" );
query.declareParameters( "int p_prio" );
Integer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

JDO intern - Zustände von Objekten

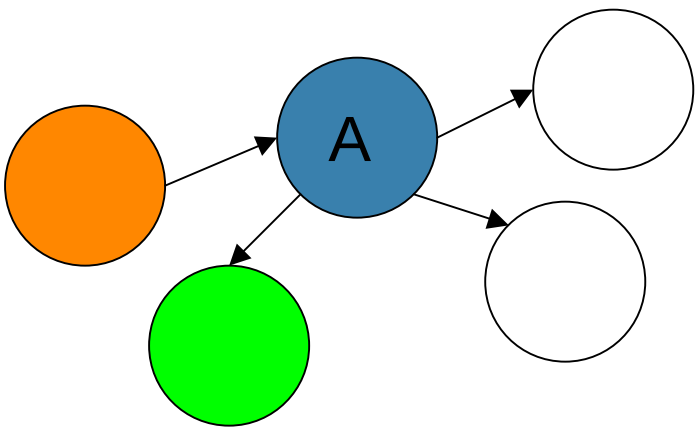
- Jedes Objekt im Speicher hat einen Zustand
  - transient, persistent-clean, persistent-dirty, hollow, …
- Zustandsübergänge entweder explizit (makePersistent()) oder implizit (Lese-/Schreibe-Zugriff)
  - exakte Definition in der JDO Spezifikation
- Hollow-Objekte sind praktisch hohl, ermöglichen das Laden von Objekten aus der DB erst bei Bedarf

FAST OBJECTS™

Hollow Objekte illustriert



Objekt A ist hollow



Nach Feldzugriff ist Objekt A nicht mehr hollow

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();
}
```

Multithreading

1. Eine JDO Implementation *darf* Multithreading zulassen auf den Klassen `Transaction`, `Extent`, `Query`, `PersistenceManager`, sowie auf den eigentlichen Datenklassen.

2. Das Multihreading kann, sofern überhaupt erlaubt, ein- oder ausgeschaltet werden:

   ```
   PersistenceManagerFactory.setMultithreaded(boolean)
   boolean PersistenceManagerFactory.getMultithreaded()
   ```

3. Applikationskontrollierte Synchronisation ist natürlich jederzeit möglich.

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

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 )
```

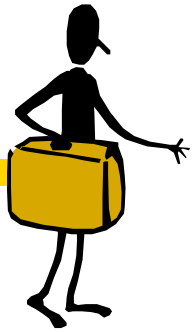
Data Store Identity, Beispiel

```
// 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.

Zusammenfassung, Kernpunkte



- Persistenz
- Einführung Java Data Objects (Teil 2)

Was kommt beim nächsten Mal?

- Java Data Objects dritter Teil