



# **Objekt-Relationales Mapping**

## **Erfahrungen mit Hibernate aus der Praxis**

**Gerhard Müller, [Gerhard.Mueller@tngtech.com](mailto:Gerhard.Mueller@tngtech.com)**

**TNG Technology Consulting GmbH**  
**<http://www.tngtech.com>**

# Ankündigung



- Thema: Objekt-Relationales Mapping
- Abstract:

Das Problem, Objekte aus der Objekt-orientierten Welt in relationalen Datenbanken zu speichern, ist so alt wie die Objektorientierung selbst. Diverse Ansätze wurden schon mit wechselndem Erfolg untersucht. Insbesondere, nach dem sich Entity EJBs als steiniger Weg erwiesen haben, hat sich der Fokus auf Container-unabhängige, dedizierte Mapping-Techniken verschoben. Hierbei werden Plain Old Java Objects, einzeln oder auch als sogar geschachtelte Collections, über XML-Mappingfiles automatisiert auf relationale Datenbanken abgebildet. Die Lösungen sind erstaunlich elegant und im praktischen Einsatz schnell produktiv zu nehmen. Wir berichten in diesem Vortrag insbesondere über den Open Source O/R-Mapper "Hibernate".
- Vortragender: Dipl. Inf. Gerhard Müller, Managing Partner bei der TNG Technology Consulting GmbH, <http://www.tngtech.com>

# Hintergrund



- TNG Technology Consulting GmbH
- Viel Erfahrung bei J2EE-Anwendungen
- Mein Schwerpunkt: Architektur, Design, Implementierung, Betrieb

# Ziel des Vortrages



- Problematik Objektorientierung / Persistenz in Relationalen Datenbanken erläutern
- Vorstellung von Hibernate als ein Beispiel für OR- Mapper
- Beispiele 😊

# Agenda



- Einführung
- Vorstellung von Hibernate
  - Einführung
  - Überblick
  - Hello, World!
  - Konfiguration & Mapping
  - Arbeiten mit Hibernate
  - Hibernate Query Language
- Bewertung

# Objektorientierung: was heißt das noch mal?



- Objekte haben eine eindeutige, von ihren Attributen unabhängige (implizite) Objektidentität und Lebenszyklus
  - Java: Garbage Collection, Existenz durch Erreichbarkeit
- Objekte können Attribute haben
  - Attribute sind meist nicht direkt zugreifbar ("information hiding", setter/getter)
- Objekte können Beziehungen haben
  - 1:1 (Referenz, gerichtet und ungerichtet)
  - 1:n (Referenzen, z.B. Array, gerichtet und ungerichtet)
  - n:m (1:n und 1:m)
- Objekte haben ggf. Typ-Beziehungen untereinander
  - Vererbung, instanceof, Implementierung von Interfaces
- Collections von Objekten
  - Arrays, Sets, Lists,... (Collection API)
- Java: Sprache der 3. Generation (prozedural)

# Persistenz von Java-Objekten



- Java-Objekte sind normalerweise "transient"
- Speicherungsmöglichkeiten:
  - Objektserialisierung
    - Serializable...
  - Manuelles Objekt-Relationales Mapping
    - Handgeschriebener JDBC-Code
  - OR-Mapping-Tools
    - Java Data Objects (JDO)
    - Container Managed Persistence (EJB 2.x)
    - **Proprietäre Tools und Frameworks**
  - Objektorientierte Datenbanksysteme (OODMBS)

# Wünschenswerte Persistenzeigenschaften



- **Transparenz**  
Benutzer arbeiten in gleicher Weise mit transienten und persistenten Objekten. Persistenz erfordert keine Sonderbehandlung bei der Programmierung
- **Interoperabilität**  
Persistente Objekte können auch in anderen als der Erstellungsumgebung verwendet werden UND das Festschreiben ist von konkreten Persistenzsystemen unabhängig.  
→ Laufzeitumgebung und persistenter Speicher sind austauschbar
- **Skalierbare Wiederauffindbarkeit**  
Das Auffinden von persistenten Objekten erfolgt ohne vollständiges Durchsuchen des Objektpools
- **Mehrbenutzer, Konflikterkennung, Verteilung**

Quelle: <http://www.michaelklein.net/obj2/material/2003-03-31-Objektpersistenz.pdf>

# Relationale Datenbanken



- Relationen: **Mengen** von (einfachen) Attributen
- **Schlüssel**: minimale, identifizierende Attributkombination; Fremdschlüssel
- Strikte Trennung zwischen Daten & Operationen
  - Alle Daten sind "sichtbar"
- Paralleles Arbeiten auf Daten die Regel
  - Transaktionen
- Große Datenmengen kein Problem
- SQL: Sprache der 4. Generation (deskriptive Sprache; das Ergebnis wird definiert, aber nicht der Algorithmus, wie es berechnet wird)

# Problem: O/R-"Mismatch"



- Das relationale und das objektorientierte Modell lassen sich nicht perfekt aufeinander abbilden
  - Beziehungen (Relationships)
    - Schlüssel und Tabellen vs. explizite Objekt-Identität
  - Theorie
    - Prozedural vs. Mengentheorie
  - Organisation
    - Tabellen vs. Klassendefinitionen
  - Identität
    - Eindeutig vs. nicht eindeutig

# Mapping von Objekten auf relationale Datenbanken

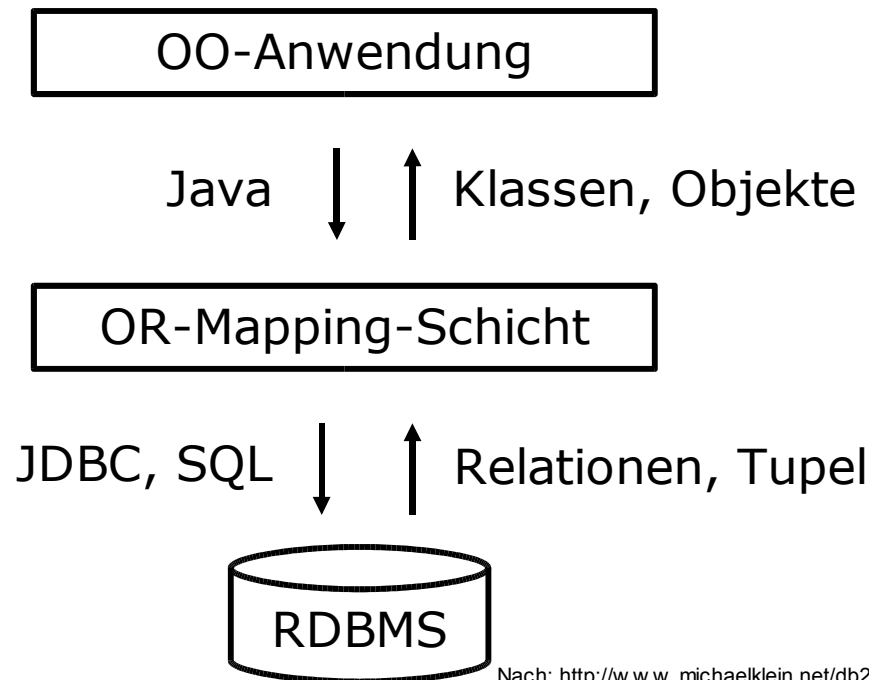


- Klasse → Tabelle(n)
- Objektidentität → Surrogat (id, Sequenz, ...)
- Strukturen:
  - Auflösen der Strukturen bis auf elementare Attributtypen (meist schneller)
  - Eigene Tabellen (joins nötig, Verbindung über ids)
- Assoziationen:
  - Referenzen → Primär/Fremdschlüsselbeziehungen
  - Ggf. in eigenen Tabellen (n:m-Assoziationen)

# Objekt-Relationale Mapping-Tools



- **Idee:** Füge zwischen Anwendung und RDBMS eine zusätzliche Softwareschicht ein, die das OR-Mapping automatisch und transparent durchführt:



Nach: <http://www.michaelklein.net/db2/material/2003-03-31-Objektpersistenz.pdf>

# Performance: Lazy Loads sind notwendig



- Ein Objekt (mit assoziierten Objekten) kann ziemlich groß sein, z.B.
  - Eine Organisation
  - Ein Auto
- Das Laden aller Daten auf Einmal ist häufig nicht möglich
  - Die Daten müssen geladen werden, während man sich durch die Objektbeziehungen "durchhangelt"
  - Dabei müssen "Stellvertreterobjekte" (Proxies) durch die "echten" Daten on-the-fly ersetzt werden können

# Objekt-Relational Mapping Tools



- **Open Source:**
  - Castor/JDO  
<http://castor.exolab.org/>
  - JRelationalFramework  
<http://jrf.sourceforge.net/>
  - Turbine/Torque  
<http://jakarta.apache.org/turbine/tor>
  - Cayenne  
<http://objectstyle.org/cayenne/>
  - ObjectRelationalBridge  
<http://db.apache.org/obj/>
  - **Hibernate**  
<http://www.hibernate.org/>
  - ...
  
- **Kommerzielle Tools:**
  - TopLink / Oracle
  - CocoBase / Thought, Inc.
  - ...
  
- **Siehe auch:**  
[http://www.service-architecture.com/products/object-relational\\_mapping.html](http://www.service-architecture.com/products/object-relational_mapping.html)



# HIBERNATE



- Open-Source (LGPL) Java-OR-Mapper
- Von der Homepage <http://www.hibernate.org/>:
  - "Relational Persistence For Idiomatic Java"
  - "Hibernate is a powerful, ultra-high performance object/relational persistence and query service for Java. Hibernate lets you develop persistent objects following common Java idiom - including association, inheritance, polymorphism, composition and the Java collections framework. Extremely fine-grained, richly typed object models are possible. The *Hibernate Query Language*, designed as a "minimal" object-oriented extension to SQL, provides an elegant bridge between the object and relational worlds. Hibernate is now the most popular ORM solution for Java."

# Hibernate - Überblick



- Versionen:
  - 1.2.5: alte, stabile Version
  - 2.0.3: neue, stabile Version (empfohlen)
  - 2.1 beta3b: neueste Entwicklerversion
- Unterstützte Datenbanken:
  - DB2, MySQL, SAP DB, Oracle, Oracle 9, Sybase, Sybase Anywhere, Progress, Mckoi SQL, Interbase, Pointbase, PostgreSQL, HypersonicSQL, Microsoft SQL Server, Ingres, Informix, FrontBase

# Hibernate - Download



- Hibernate
  - Hibernate 2.1 beta 3 (7.09.2003)
  - Hibernate 2.0.3 (27.08.2003) **empfohlen**
    - Benötigte Jars, Source, umfangreiche Doku
- Hibernate Extensions 2.0 (13.06.2003)
  - hbm2java, class2hbm, ddl2hbm
  - Avalon-Wrapper für Hibernate
- Hibernate Examples 20030725 (30.07.2003)
  - **"Quickstart: A tutorial for Hibernate beginners, using Tomcat 4.1 and Hibernate2. Includes example configuration files, a persistent Java class and mapping"**

# Hibernate - Überblick



- "Einfaches" Persistenz-Framework
  - Arbeitet mit Reflection
  - Kein Byte-Code-Enhancement  
(zumindest von dem der Entwickler etwas mitbekommt)
- Unterstützt viele Datenbanken mit JDBC-Treibern
- Ziemlich transparent für Anwendungsentwickler
- Vielfältige Mapping-Unterstützungen
- Verwendungsmöglichkeiten:
  - Standalone
  - in Applikationsservern
  - in EJBs
  - ...

# Hibernate - Features



- Kann über JMX gemanaged werden
- Unterstützt austauschbare Implementierungen für
  - Datenbank-Connection-Providern
  - Transaktionen
  - JNDI-Provider
- JCA Connector-Unterstützung
- Outer Join Fetching-Unterstützung
- Unterstützung verschiedener Schemata in Datenbanken
- Unterstützung von mehr als einer Datenbank (z.B. Oracle & MySQL)
- JCA Connector-Unterstützung
- Ausführliches Logging mittels log4j oder JDK 1.4 möglich
- Versionierte Daten für optimistisches Locking
- Lazy Loads von Collections
- Persistent Lifecycle Callbacks
- Validationsunterstützung
- Gute eigene Query-Sprache
- Oracle CLOBs werden unterstützt

# Aus was besteht eine Hibernate-Anwendung?



Eine Hibernate-Anwendung besteht normalerweise aus 4 unterschiedlichen Teilen:

1. Hibernate-Konfigurationsdatei (bzw. XML-Konfigurationsdatei)
2. Pro persistenter Klasse einer Hibernate Mapping XML-Datei
3. Der Hibernate Java Library und anderer benötigter JAR-Dateien
4. Der HQL (Hibernate Query Language), um Abfragen gegen die Datenbank zu machen

Sowie natürlich

1. Den Java-Klassen
2. Der Datenbank mit dem Datenbank Schema

# Hibernate – Hello, World!



- Installation eines DB-Treibers
  - Kopieren eines JARs
- Installation von Hibernate
  - Kopieren einiger JARs
  - Ggf. +Log4j & andere Utility-Libs
- Konfiguration von Hibernate
- Ggf. Erstellung von DB-Tabellen
- Erstellen von Java-Klassen
- Beispiel läuft 😊 (hoffentlich)

# Anforderungen an zu persistierende Klassen



- Default-Konstruktor muss verfügbar sein
  - Darf auch private sein... Besser ist package-lokal
- zu persistierende Attribute
  - werden mittels Reflection ermittelt:  
getFoo(), isFoo(), setFoo()
    - diese müssen NICHT public sein!
  - praktisch: id, am besten vom Typ Long oder String
- Klasse idealer Weise nicht "final"
  
- Anforderungen an abgeleitete Klassen
  - normale Vererbung
  - benötigt werden:
    - Default-Konstruktor
    - set/get-Methoden

# Einfaches Beispiel für eine zu persistierende Klasse



```
package com.tngtech.examples.hibernate.demo;
import org.apache.log4j.Logger;
```

```
public class Message {
    private static final Logger log = Logger.getLogger(Message.class);
    private Long id;
    private String text;
    private Message nextMessage;

    Message() { }
    public Message(String text) { this.text = text; }

    public Long getId() { return id; }
    private void setId(Long id) { this.id = id; }

    public String getText() { return text; }
    public void setText(String text) { this.text = text; }

    public Message getNextMessage() { return nextMessage; }
    public void setNextMessage(Message nextMessage) { this.nextMessage = nextMessage; }
}
```

# Hibernate-Konfiguration



- Entweder über Properties
  - Hibernate.properties
- Oder über ein XML-Konfigurationsfile
- Minimalbeispiel für Oracle 9i:

hibernate.query.substitutions	true 1, false 0, yes 'Y', no 'N'
hibernate.dialect	net.sf.hibernate.dialect.Oracle9Dialect
hibernate.connection.driver_class	oracle.jdbc.driver.OracleDriver
hibernate.connection.username	system
hibernate.connection.password	system
hibernate.connection.url	jdbc:oracle:thin:@localhost:1521:orcl9
hibernate.connection.pool_size	1
hibernate.statement_cache.size	25
hibernate.show_sql	true

# Klassen-Mapping-Dateien



- Pro persistenter Klasse typischerweise eine Datei
  - *Klassenname*.hbm.xml
  - Sinnvollerweise im Paket (src/jar) der Klasse
- Wesentliche Bestandteile:
  - <hibernate-mapping>
    - <class ...
      - <id ...>
        - <generator .../>
      - </id>
      - <discriminator .../>
      - <version .../>
      - <property .../>
      - <property .../>
      - *Assoziationen*
    - </class>
  - </hibernate-mapping>

# Klassen-Mapping-Dateien: Beispiel



```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sf.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
  <class
    name="com.tngtech.examples.hibernate.demo.Message" table="GMTEST_MESSAGES">
    <id
      name="id" column="MESSAGE_ID">
      <generator class="native"/>
    </id>
    <property
      name="text" column="MESSAGE_TEXT"/>
    <many-to-one
      name="nextMessage"
      cascade="all"
      column="NEXT_MESSAGE_ID"/>
    </class>
  </hibernate-mapping>
```

# Wie, wann und womit werden Mapping-Dateien erstellt?



- Datenbank → .hbm.xml (Middlegen)
- Class → .hbm.xml (MapGenerator)

Und dann ggf.:

- hbm.xml → Java (hbm2java)
- hbm.xml → Datenbank (via SchemaExport/SchemaUpdate)

# Klassen-Mapping: class



- `<class`
  - `name="ClassName"` (Klassenname, kann auch Interface sein. Dann muss Implementierung als `<subclass>`-Element definiert werden)
  - `table="tableName"` (Tabellenname in der Datenbank)
  - `discriminator-value="discriminator_value"` (optional, default: Klassenname)
  - `mutable="true|false"` (optional, default: true)
  - `schema="owner"` (optional, default: siehe hibernate-mapping-Attribut)
  - `proxy="ProxyInterface"` (optional, hier kann man Proxy-Klasse angeben für Lazy Initializing)
  - `dynamic-update="true|false"` (optional, default: false, nur geänderte Felder sollen per Update gespeichert werden)
  - `dynamic-insert="true|false"` (optional, default: false, nur "nicht-null"-Felder sollen gespeichert werden)
  - `select-before-update="true|false"` (optional, default: false)
  - `polymorphism="implicit|explicit"` (optional, default: implicit; definiert ob bei Queries die Subklassen mit ermittelt werden sollen oder nicht)
  - `where="arbitrary sql where condition"` (optional, kann verwendet werden um die durch Hibernate gefundenen Informationen zu reduzieren)
  - `persistor="PersistorClass"` (optional, definiert eine Klasse, die beschreibt, wie die  
zu persistierende Klasse zu persistieren ist; z.B. in LDAP)
  - `batch-size="N"` (optional, default: 1, definiert die Batch-Size für JDBC)
- `</class>`

# Klassen-Mapping: id & co



- `<id`
  - `name="propertyName"` (optional, wenn nicht angegeben, wird davon ausgegangen, dass Klasse keine "id" hat)
  - `type="typename"` (optional)
  - `column="column_name"` (optional, default: propertyName)
  - `unsaved-value="any|none|null|id_value">` (optional, default: null)
  - `<generator class="generatorClass"/>`
    - (increment, identity, sequence, hilo, seqhilo, uuid.hex, uuid.string, native, assigned, foreign; eigene sind möglich. Achtung: Verfügbarkeit je nach Datenbank, Sinnhaftigkeit je nach Einsatzszenario!)
- `</id>`
- `<composite-id>`, nur falls nötig, wird auch unterstützt
- `<discriminator`
  - `column="discriminator_column"` (optional, default: class)
  - `type="discriminator_type"` (optional, default: string, erlaubt: string, character, integer, byte, short, boolean, yes\_no, true\_false)
  - `force="true|false"` (optional, default: false)
- `/>`
- `<version`
  - `column="version_column"` (optional, default: property name)
  - `name="propertyName"` (der Name der Version-Property in der persistenten Klasse)
  - `type="typename"` (optional, default: integer, der Typ für die Versionsnummer, erlaubt sind: long, integer, short, timestamp, calendar)
- `/>`

# Klassen-Mapping: Attribute



- `<property`
  - **name**="propertyName" (Java Property)
  - **column**="column\_name" (optional, default: propertyName)
  - **type**="typename" (optional, definiert Hibernate-Typ, u.a. )
  - **update**="true|false" (optional, default: true, definiert, ob diese Property mit in Updates gespeichert werden soll)
  - **insert**="true|false" (optional, default: true, definiert, ob diese Property mit in Inserts gespeichert werden soll)
  - **formula**="arbitrary SQL expression" (optional, ermöglicht "berechnete" Properties)
- `>`

Unterstützte Typen:

1. Der Name von Hibernate-Basistypen  
(wie integer, string, character, date, timestamp, float, binary, serializable, object, blob).
2. Der Name einer Java-Klasse mit einem Default-Basistyp  
(wie int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob).
3. Der Name einer Subklasse von PersistentEnum
4. Der Name einer serialisierbaren Java-Klasse
5. Der Klassenname eines Custom Types (z.B.. com.illflow.type.MyCustomType).

# Klassen-Mapping: Beziehungen



- one-to-one
- many-to-one
- one-to-many
- many-to-many
- components
- subclass
- joined-subclass
- set
- map
- composite-id

*Genauere Erklärung siehe <http://www.xylax.net/hibernate/>*

# Collections



- Collections werden durch `<set>`, `<list>`, `<map>`, `<bag>`, `<array>` und `<primitive-array>` definiert.
- Beispiel:
- `<map`
  - `name="propertyName"`
  - `table="table_name"`
  - `schema="schema_name"`
  - `lazy="true|false"`
  - `inverse="true|false"`
  - `cascade="all|none|save-update|delete|all-delete-orphan"`
  - `sort="unsorted|natural|comparatorClass"`
  - `order-by="column_name asc|desc"`
  - `where="arbitrary sql where condition"`
  - `outer-join="true|false|auto"`
  - `batch-size="N">`
  - `<key .... />`
  - `<index .... />`
  - `<element .... />`
- `</map>`

# Abgeleitete Klassen



- Empfohlen: "table-per-class-hierarchy" Mapping-Strategie mittels "subclass"-Elementen (eine DB-Tabelle für alle abgeleiteten Klassen)
- Auch möglich: "table-per-subclass" Mapping-Strategie mittels "joined-subclass"-Elementen (eine eigene DB-Tabelle für jede Klasse)
- `<subclass`
  - `name="ClassName"`
  - `discriminator-value="discriminator_value"` (optional, default: Klassenname)
  - `proxy="ProxyInterface"` (optional, Klasse oder Interface für lazy-Initialisierungs-Proxies)
  - `dynamic-update="true|false"` (optional, default: true, siehe class-Element)
  - `dynamic-insert="true|false">` (optional, default: true, siehe class-Element)
  - `<property .... />`
  - .....
- `</subclass>`
- `<joined-subclass`
  - ...
  - `<key .... >`
  - `<property .... />`
  - .....
- `</joined-subclass>`

# Wichtige Hibernate-Klassen und Interfaces



- Configuration -- Initialisierung
- SessionFactory -- Erzeugt Sessions
- TransactionFactory, ConnectionProvider: interne Abstraktionsklassen
- Session -- repräsentiert das "Gespräch" der Anwendung mit dem persistenten Speicher; entspricht einem Cache auf transaktionaler Ebene
- Auf Session- und Session-Factory-Ebene sind caches möglich
- Transaktion – durch Sessions erzeugte atomare "Gesprächs"teile

# Arbeiten mit Hibernate - Speichern einer Nachricht



```
Session session = getSessionFactory().openSession();  
Transaction trans = session.beginTransaction();
```

```
Message message = new Message("Hello, World!");  
session.save(message);
```

```
trans.commit();  
session.close();
```

SQL:

```
insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT,  
NEXT_MESSAGE_ID) values (1, 'Hello, World!', null)
```

# Arbeiten mit Hibernate - Auslesen von Nachrichten



```
Session session = sessionFactory().openSession();  
Transaction trans = session.beginTransaction();
```

```
List messages = session.find("from Message message order by message.text asc");  
System.out.println( messages.size() + " message(s) found:" );
```

```
for ( Iterator iter= messages.iterator();iter.hasNext(); ) {  
    Message message = (Message)iter.next();  
    System.out.println( message.getText() );  
}
```

```
trans.commit();  
session.close();
```

SQL:

```
select message.MESSAGE_ID, message.MESSAGE_TEXT, message.NEXT_MESSAGE_ID  
from MESSAGES message order by message.MESSAGE_TEXT asc
```

# Arbeiten mit Hibernate - Aktualisieren einer Nachricht



```
Session session = getSessionFactory().openSession();  
Transaction trans = session.beginTransaction();
```

```
// 1 is the generated id of the first message
```

```
Message message = (Message) session.load( Message.class, new Long(1) );  
message.setText("Greetings Earthling");  
message.setNextMessage( new Message("Take me to your leader (please)" );
```

```
trans.commit();  
session.close();
```

```
SQL:
```

```
select message.MESSAGE_ID, message.MESSAGE_TEXT, message.NEXT_MESSAGE_ID from MESSAGES  
message where message.MESSAGE_ID = 1
```

```
insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID) values (2, 'Take me to your  
leader (please)', null)
```

```
update MESSAGES set MESSAGE_TEXT = 'Greetings Earthling', NEXT_MESSAGE_ID = 2 where MESSAGE_ID  
= 1
```

# Arbeiten mit Hibernate - Löschen einer Nachricht



```
Session session = getSessionFactory().openSession();  
Transaction trans = session.beginTransaction();
```

```
Message message = (Message) session.load( Message.class, new Long(1) );
```

```
// method 1 – deleting the loaded message
```

```
session.delete(message);
```

```
// method 2 – delete all Messages after the tenth message
```

```
// session.delete(“from player in class example.message where player.id > 10”);
```

```
trans.commit();
```

```
session.close();
```

SQL:

```
delete from MESSAGES message where message.MESSAGE_ID = 1
```

# Braucht man immer eine Session?

## Nein! → optimistische Nebenläufigkeit



// läuft so durch

```
Session sessionOne = getSession();
Transaction transOne = sessionOne.beginTransaction();
AdvancedMessage wellKnownEntity1 =
(AdvancedMessage) sessionOne.load
(AdvancedMessage.class, wellKnownKey);
transOne.commit();
sessionOne.close();
```

```
// modify without a session...
wellKnownEntity1.setText("modified without a session...");
```

// ok, and finally save object in another session...

```
Session sessionTwo = getSession();
Transaction transTwo = sessionTwo.beginTransaction();
sessionTwo.update(wellKnownEntity1);
transTwo.commit();
sessionTwo.close();
```

// Endet mit Exception

```
Session sessionOne = getSession();
Transaction transOne = sessionOne.beginTransaction();
AdvancedMessage wellKnownEntity1 = (AdvancedMessage)
sessionOne.load(AdvancedMessage.class, wellKnownKey);
transOne.commit();
sessionOne.close();
```

// modify without a session...

```
wellKnownEntity1.setText("modified without a session...");
```

```
Session sessionThree = getSession();
Transaction transThree = sessionThree.beginTransaction();
AdvancedMessage wellKnownEntity2 = (AdvancedMessage)
sessionThree.load(AdvancedMessage.class, wellKnownKey);
wellKnownEntity2.setText("modified inside a session");
transThree.commit();
sessionThree.close();
```

// ok, and finally save object in another session...

```
Session sessionTwo = getSession();
Transaction transTwo = sessionTwo.beginTransaction();
sessionTwo.update(wellKnownEntity1);
```

**// HERE this will fail with an EXCEPTION**

```
transTwo.commit();
```

```
sessionTwo.close();
```

# Hibernate Query Language



- Hibernate hat eigene Query-Language: Hibernate Query Language
  - sieht ähnlich aus wie SQL
  - aber: voll objektorientiert, versteht Vererbung, Polymorphie und Assoziationen
  - Hibernate Query Language (HQL) hat deutlich mehr Features als z.B. die EJB QL
    - from
    - select
    - where
    - having
    - 'order by'
    - 'group by'
    - Aggregationsfunktionen

# HQL - Beispiele



- alle Foos:
  - from com.tngtech.Foo
- alle persistenten Objekte:
  - from java.lang.Object o
- from com.tngtech.Foo as foo where foo.name='XXX'
- from bank.Account account where
  - account.owner.id.country = 'AU'
  - and account.owner.id.medicareNumber = 123456
- from com.tngtech.Foo foo where foo.class = com.tngtech.FooBar

# HQL – Finden von Objekten



- `List list = session.find(...)`, um Objekte zu finden
- `Iterator it = session.iterate(...)`, wenn nicht alle Ergebnisse benötigt werden
- `Query q = session.createQuery(...)`, um noch feinere Kontrolle über die Ergebnisse zu bekommen
- Anzahl von Objekten, ohne diese zu instantiieren:  
`( (Integer) session.iterate("select count(*) from ...").next() ).intValue()`

# Ausgewähltes Query-Beispiel



## **HQL:**

```
select cust
from Product prod, Store store
inner join store.customers cust
where prod.name = 'widget'
      and store.location.name in
      ( 'Melbourne', 'Sydney' )
      and prod = all elements
(cust.currentOrder.lineItems)
```

## **SQL-Äquivalent:**

```
SELECT cust.name, cust.address, cust.phone,
       cust.id, cust.current_order
FROM customers cust,
     stores store,
     locations loc,
     store_customers sc,
     product prod
WHERE prod.name = 'widget'
      AND store.loc_id = loc.id
      AND loc.name IN ( 'Melbourne', 'Sydney' )
      AND sc.store_id = store.id
      AND sc.cust_id = cust.id
      AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
              AND cust.current_order = o.id
      )
```

# Advanced Stuff (wenn die Zeit reicht...)



- Probleme beim Laden von XML-Dateien
  - Lösung: Entity-Resolver
- Versionierung
- Audit-Interceptor
- Eigener Connection Provider
- Speicherung/Zugriff von Sessions
- Vermeidung von Fehlerbehandlungscode:  
Transaktionsblöcke als anonyme innere Klassen

# Hibernate – Integrationsmöglichkeiten & Tools



- Entwicklung:
  - XDoclet (<http://xdoclet.sourceforge.net/>)
  - Middlegen (<http://sourceforge.net/projects/middlegen>)
  - AndromDA (<http://www.andromda.org/>)
  - IDE Plugin-Unterstützung: Eclipse, IDEA
  - Ant
- Betrieb:
  - Diverse JDBC-Connection-Pools, u.a.
    - C3P0 (<http://sourceforge.net/projects/c3p0>)
    - Apache DBCP (<http://jakarta.apache.org/commons/dbcp/>)
    - Proxool (<http://proxool.sourceforge.net/>)
  - Diverse Caching-Implementierungen, u.a.:
    - Tangosol Coherence Cache

# Bekannte Einschränkungen



- Einige features der Query-Sprache werden nicht bei Datenbanken unterstützt, die keine Subquery-Unterstützung haben
- Die "table-per-subclass" Mapping-Strategie wird nur bei Datenbanken unterstützt mit ANSI-entsprechendem CASE...WHEN-Support oder (Oracle) DECODE-Funktion
  - NICHT bei HSQLDB und MySQL

# Hibernate – Bewertung



- **Transparenz**
  - Eingeschränkt gegeben. Nutzer muss sich über manchmal leicht veränderte Semantik im Klaren sein.
- **Interoperabilität**
  - Teilweise gegeben. Anwendung ist fest an Hibernate gebunden.
- **Skalierbare Wiederauffindbarkeit**
  - Gegeben. Spezielle Anfragesprache (HQL) wird auf SQL gemappt und direkt im RDBMS ausgeführt.

# Hibernate-Bewertung: Vorteile



- Open Source
- Relativ transparent
- Lässt sich Clustern
- Geringer Overhead
  - Mission: mache etwas, und dann richtig
  - Keine Standard-Unterstützung, nur das eigene Interface
- Gute Mapping-Unterstützung
  - 1-1; many-1; many-many
- Offenes Caching-Interface
- Keine Basisklasse für zu persistierende Klassen, von der man erben muss, kein Interface, was erfüllt werden muß
  - damit keine starke Kopplung von persistenten Objekten an Hibernate (POJ-Objekte)
- Starke Query-Sprache (je nach Mächtigkeit der zugrundeliegenden Datenbank)
- Gute Doku

# Hibernate-Bewertung: Nachteile



- Mapping-Granularität
  - Kann nicht eine Klasse auf 3 Tabellen abbilden
- Implikationen aus Lazy-Laden nicht immer leicht zu verstehen
- Kein Standard
- Keine graphischen Tools

# Referenzen



- Hibernate
  - <http://www.hibernate.org/>
  - <http://sourceforge.net/projects/hibernate/>
- Hibernate-Einführung
  - <http://www.theserverside.com/resources/articles/Hiberr>
- Mapping-Diagramme
  - <http://www.xylax.net/hibernate/>
- Scott Ambler
  - <http://www.agiledata.org/essays/mappingObjects.html>
  - <http://www.agiledata.org/essays/impedanceMismatch.html>



**ENDE**