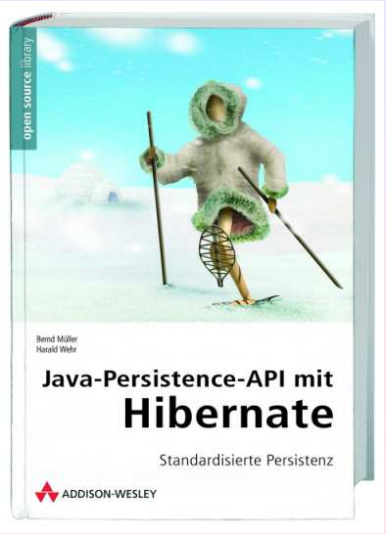
JPA und/mit Hibernate

Bernd Müller

Fachhochschule Braunschweig/Wolfenbüttel
Fachbereich Informatik

Sommersemester 2008

Grundlage und ausführliche Erläuterungen



Bernd Müller
Java-Persistence-API mit Hibernate
Standardisierte Persistenz
Addison-Wesley
ISBN 978-3-8273-2537-2
39,95 Euro
Bei Amazon kaufen

Agenda

Motivation und Einführung

Motivation

Motivation: Die Aufgabe

- Verwaltung persistenter Daten ein Standardproblem der Software-Entwicklung
- lange Zeit Diskussionen, ob CRUD-Operationen händisch codiert oder automatisiert werden sollten
- mittlerweile Konsens: automatisiert (Motivation nächste Seite)
- Frage: wie? (es gibt mindestens 20 Alternativsysteme)

Konsens?

- händische JDBC-Konsistenz ist immer schneller/effizienter als eine automatisierte Lösung !
- ja, genauso wie Assembler-Code immer schneller ist als compilierter Code, oder gar Java. Wollen Sie Assembler schreiben?
- ja, aber nur, wenn Sie beliebig viel Ressourcen voraussetzen. Können Sie 50 % Ihrer Projekt-Ressourcen in die Persistenz stecken?
- wissen Sie, welche guten Cache-Alternativen existieren?
- wissen Sie, welche Unterschiede zw. Oracle, DB2, MS SQL, ... existieren?
- wissen Sie, ...

Historie

- 2001 empfand Gavin King die Standardpersistenzlösung (CMP Entity Beans) als unbefriedigend und begann eigene Entwicklung einer Persistenzlösung
- Hibernate versuchte dabei Anforderungen von Entwicklern zu erfüllen
- Diese Anforderungen nahmen überhand (gerechnet auf ein sehr kleines Entwickler-Team)
- Gavin King übergab Hibernate JBoss Inc. und entwickelte fortan full-time
- ... EJB3 (JPA) maßgeblich mit beeinflusst
-

Hibernate

Charakteristika

- Unterstützte DBMS: Oracle, DB2, MS SQL Server, Sybase, PostgreSQL, MySQL, HSQLDB, SAP DB, Informix, Interbase, Pointbase, Ingres
- konform zu JPA und EJB 3.0
- Lizenz: Lesser GNU Public License
- sehr weit verbreitet: 2 Millionen Downloads in 5 Jahren

Funktionsweise

- Meta-Daten
  - welche Klasse auf welche Tabelle(n) (fine-grained)
  - Identität
  - Assoziationen
  - Vererbung
  - ...

HSQLDB

- Open-Source DBMS
- Home: www.hsqldb.org
- in Java implementiert
- ist die Datenbank für Open-Office
- ist die (Default-) Datenbank für JBoss AS (Seam, ...)
- In-Process-Mode (selbe JVM), Server-Mode (TCP/IP), ...
- für Schulungen ausreichender Teil von SQL realisiert
- Daten als Text-File (super für Dozenten)
- ...
- übrigens: Java-DB (Derby) ab Java 6.

kleine Demo

Weiterführende Literatur

siehe Kapitel Literatur

Mapping einfacher Klassen

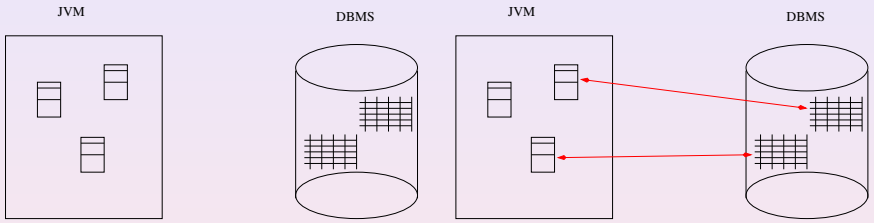
- Mit XML-Mapping-Datei
- Mit Annotationen (Java 5)

XML-Mapping-Dateien

Meta-Daten

- Meta-Daten werden immer benötigt
- XML-Mapping-Dateien *der* Hibernate-Standard
- größter Funktionsumfang
- funktioniert auch mit Java 1.3 und 1.4

Zu beantwortende Fragen



- Welche Klassen sind persistent?
- Welche Tabelle für welche Klasse?
- Vererbung?
- Assoziationen?
- ...

Grundsätzliches

- Hibernate-Objekte sind normale Java-Objekte (POJOs)
- müssen kein Interface implementieren
- müssen von keiner Oberklasse erben
- Sie sehen der Klasse nicht an, dass sie persistent ist

Wie kommen Java, SQL und Mapping-Definition zusammen?

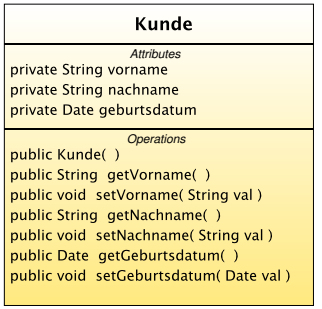
Top down Beginne mit Business-Modell in Java, erzeuge Mapping-Meta-Daten, erzeuge (generiere) DB-Schema

Bottom up Existierendes DB-Schema. Reverse-Engineering (händisch/automatisch), um Mapping-Meta-Daten zu erzeugen. Erzeuge (generiere) Java-Business-Modell

Middle out Mapping-Meta-Daten reichen aus, um sowohl Java, als auch DB-Schema zu generieren. Machen wir gleich!

Meet in the middle Existierendes Java-Modell *und* DB-Schema. Wir schreiben Mapping-Meta-Daten. Schwierigster Fall und in der Regel nicht ohne Änderung am Business-Modell und/oder DB-Schema möglich

Beispiel Klasse Kunde



- POJO = JavaBean
- also Default-Konstruktor
- Getter und Setter
- Problem: Java-Identität und DB-Identität sind etwas ganz anderes

Identität und Gleichheit

Java-Identität: == Die JVM unterscheidet Objekte anhand ihrer Referenzen (für C-Programmierer: Zeiger auf den Heap)

Gleichheit Java definiert die Gleichheit von Objekten über die `equals()`-Methode, die Sie definieren (müssen). Definiert in `java.lang.Object`, d.h. Sie erben die Methode immer.
Anmerkung: Die geerbte Methode ist die Identität.
Bitte merken: Wenn Sie `equals()` überschreiben, müssen Sie auch `hashCode()` überschreiben.

DB-Identität Über den Primärschlüssel einer Tabelle. Nicht abbildbar auf Java, daher neues Property.
Gehört aber nicht in UML, sondern ist ein Implementierungsdetail der Persistenz.

Mapping der Klasse Kunde

```
<!DOCTYPE ...
<hibernate-mapping
  package="de.pmst.hibernate.intro.model">
  <class name="Kunde" table="kunde">
    <id name="id" type="integer">
      <column name="id" />
      <generator class="identity" />
    </id>
    <property name="vorname" type="string">
      <column name="vorname" length="30" />
    </property>
    <property name="nachname" type="string">
      <column name="nachname" length="30" />
    </property>
    <property name="geburtsdatum" type="date">
      <column name="geburtsdatum" />
    </property>
  </class>
</hibernate-mapping>
```

Erläuterung der XML-Elemente

hibernate-mapping Klar, das Wurzelelement. Mit optionalem Java-Package.

class Die Java-Klasse mit optionalem Tabellennamen

id Der Primärschlüssel vom Typ integer

column Der optionale Spaltenname

generator Generator des Primärschlüssels

property Java-Property

Java und SQL

- man muss jetzt die Java-Klasse und die Tabelle erzeugen
- wir wollen den *Middle-Out*-Ansatz verfolgen, d.h. die Java-Klasse und das DDL-Statement erzeugen lassen
- die Generatoren sind als Ant-Tool erhältlich:
  - Klasse: `org.hibernate.tool.ant.HibernateToolTask`
  - Tool: `<hbm2java>`
  - Tool: `<hbm2ddl>`
- die Erzeugnisse sehen wir gleich als Demo

Wie wird das Ganze eingesetzt?

- Zur Erinnerung: Die Java-Klasse ist ein JavaBean: Default-Konstruktor und Properties für `id`, `vorname`, `nachname` und `geburtsdatum` (Property, Getter, Setter)
- Die Hibernate-Session ist das zentrale Objekt. Es repräsentiert die „Unit-of-Work“ mit der Datenbank
- Die Hibernate-Session merkt sich die von ihr persistent zu haltenden Objekte und verwaltet dazu eine Liste von SQL-Statements, die „irgendwann einmal“ an die Datenbank zu schicken sind.
- Das Transaction-Objekt verwaltet Transaktionen. Alternatven sind JTA oder container-managed Transaktionen bei EJBs.
- Das Beispiel liest sich ganz einfach

Verwendung der Klasse Kunde, Teil 1

```
public static void main(String[] args) throws ParseException{
    SessionFactory sf;
    Transaction tx = null;
    Session session = null;

    sf = new Configuration().configure().buildSessionFactory();

    Kunde neu = new Kunde();
    neu.setVorname("Heidi");
    neu.setNachname("Mustermann");
    neu.setGeburtsdatum(new SimpleDateFormat("dd.MM.yyyy")
                        .parse("11.07.1960"));

    session = sf.openSession();
    tx = session.beginTransaction();
    session.save(neu);
    tx.commit();
```

Verwendung der Klasse Kunde, Teil 2

```
...
    tx = session.beginTransaction();
    Kunde kunde =
        (Kunde) session.load(Kunde.class, new Integer(1));
    System.out.println("gelesener Kunde: "
                       + kunde.getNachname());
    tx.commit();

    session.close();
    sf.close();
}
```


Konfiguration

Hibernate muss noch wissen

- welches DBMS
- welchen JDBC-Treiber
- welchen Server (Server-Instanz)
- Authentifizierungsdaten
- ...

Konfigurationdatei hibernate.cfg.xml

```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">
      org.hsqldb.jdbcDriver
    </property>
    <property name="hibernate.connection.url">
      jdbc:hsqldb:hsql://localhost/bank
    </property>
    <property name="hibernate.connection.username">
      sa
    </property>
    <property name="hibernate.dialect">
      org.hibernate.dialect.HSQLDialect
    </property>
    <property name="show_sql">true</property>
  </session-factory>
</hibernate-configuration>
```

Übung

Öffnen Sie das Eclipse-Projekt.
Laden Sie das Ant-Build-File im Verzeichnis hsqldb.
Laden Sie das Ant-Build-File im Wurzelverzeichnis des Projekts.
Starten Sie die Datenbank (Target start).
Starten Sie den HSQL-Database-Manager (Target GUI).

Übung

Führen Sie eine Abfrage durch:

1. *Command → Select*
2. *Kunde einfügen*
3. *Execute SQL*

Führen Sie das Target „insert data“ aus und wiederholen Sie die Abfrage.
Beenden Sie das Datenbanksystem (Command → Shutdown → Execute SQL)

Übung

Führen Sie das Target „run KundeTest“ im Projekt-Build-File aus und überprüfen Sie die Funktionsweise.

Übung

Hibernate definiert eine Reihe sinnvoller Defaults für bestimmte Werte. Entfernen Sie aus obigem Mapping die Angabe des Tabellennamens und die Angabe der Spaltennamen. Welche Änderungen ergeben sich?

Übung

Anstatt einen Kunden über den Primärschlüssel zu laden, sollen alle Kunden geladen werden. Die Hibernate-Query-Language (HQL) ist eine SQL-ähnliche Sprache, die dies leicht ermöglicht. Erstellen Sie eine neue Klasse. Die Anweisung

```
session.createQuery("from Kunde k").list()
```

liefert die Liste (java.util.List) aller Kunden zurück. Iterieren Sie über die Liste und geben Vor- und Nachnamen aus.

Annotationen

Warum Annotationen?

- EJB 3.0 enthält JPA
- Alles an einer Stelle
- Nachteile:
  - geht nur mit Java 5 aufwärts
  - mehr Java-Code
  - man sieht es der Klasse an (schadet aber nichts)
  - für die „alten“ Hasen: neue Syntax

Was ändert sich?

- Angabe, *welche* Klasse persistent ist. Durch die `@Entity`-Annotation, nicht durch Mapping
- Mindestangabe des Primärschlüssels. Durch die `@Id`-Annotation.
- Defaults:
  - alle Properties der Klasse, die nicht mit `@Transient` annotiert sind
  - Tabelle wie Klasse
  - Spalten wie Properties

Klasse Kunde

```
@Entity
public class Kunde implements Serializable {

  private Integer id;
  private String vorname;
  private String nachname;
  private Date geburtsdatum;

  public Kunde() {}


  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  public Integer getId() { return this.id; }

  public void setId(Integer id) { this.id = id; }

  public String getVorname() { return this.vorname; }
  public void setVorname(String vorname) { this.vorname = vorname; }
  ..
```


Verwendung der Klasse Kunde

- keine Änderungen
- außer andere Konfiguration

```
...
sf = new AnnotationConfiguration().configure().buildSessionFactory();

Kunde neu = new Kunde();
...
session = sf.openSession();
tx = session.beginTransaction();
session.save(neu);
tx.commit();
...
```


Konfiguration

- ändert sich auch nicht wesentlich
- als Mapping wird jetzt die annotierte Klasse angegeben

```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class"> ...

    <mapping class="de.pdbm.hibernate.intro.model.Kunde" />

  </session-factory>
</hibernate-configuration>
```

Generieren der DDL

- auch mit Hibernate-Tools
- jetzt aber mit `annotationconfiguration`

```
<hibernatetool >
  ...
  <annotationconfiguration
     configurationfile="hibernate.cfg.xml" />
  <hbm2ddl outputfilename="schema.sql"
           drop="true" format="true" />

</hibernatetool>
```


Übung

Im Projekt `intro-anno` starten Sie das Datenbanksystem und führen das Target `KundeTest` aus.

Übung

Generieren Sie die Datenbank neu (Target „generate DDL“). Beobachten Sie das Logging.

Übung

Frau Mustermann hat geheiratet und nimmt den Namen ihres Mannes an. Kopieren Sie die Klasse `KundeTest` und editieren Sie sie. Ihr Code sollte dem unten angegebenen Code-Ausschnitt ähneln.

```
session = sf.openSession();
tx = session.beginTransaction();
Kunde kunde =
  (Kunde) session.load(Kunde.class, new Integer(1));
System.out.println("gelesener Kunde: "
                   + kunde.getNachname());
kunde.setNachname("Mueller");
tx.commit();
session.close();
```


Ist Ihnen etwas aufgefallen?

- es wurde keine Speicheroperation ausgeführt
- nachdem das Objekt an die Session gebunden wurde, wurde ein Setter aufgerufen
- am Ende der Transaktion werden alle Änderungen persistent gemacht
- man nennt dies *Automatic Dirty Checking*

Java Persistence API

Überblick

- für das Entity-Mapping hatten wir bereits JPA verwendet (`javax.persistence. ...`)
- jetzt auch für Main-Methode:
  - aus `SessionFactory` wird `EntityManagerFactory`
  - aus `Session` wird `EntityManager`
  - aus `Transaction` wird `EntityTransaction`
- bitte nur als „Daumenregel“
- konzeptionell bleibt alles beim Alten
- Anzeichen, dass Hibernate JPA *sehr* beeinflusst hat
- zusätzlich noch Datei `persistence.xml` im Verzeichnis `META-INF`

Verwendung des Entity-Managers

```
public static void main(String[] args) throws ParseException {
  EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("kundentest");
  EntityManager em;
  EntityTransaction tx;

  Kunde neu = new Kunde();
  neu.setVorname("Heidi");
  neu.setNachname("Mustermann");
  neu.setGeburtsdatum(new SimpleDateFormat("dd.MM.yyyy")
                      .parse("11.07.1960"));

  em = emf.createEntityManager();
  tx = em.getTransaction();
  tx.begin();
  em.persist(neu);
  tx.commit();

  ...
```

Verwendung des Entity-Managers II

```
  ...

  tx = em.getTransaction();
  tx.begin();
  Kunde kunde = em.find(Kunde.class, new Integer(1));
  System.out.println("gelesener Kunde: " + kunde.getNachname());
  tx.commit();

  em.close();
  emf.close();
}
```

persistence.xml

```
<persistence ...>
  <persistence-unit name="kundentest">

  <!-- wird von JPA verlangt, Hibernate sucht aber nach Entities -->
  <!--class>de.pdbm.hibernate.intro.model.Kunde</class-->

    <properties>
      <property name="hibernate.connection.driver_class"
                value="org.hsqldb.jdbcDriver" />
      <property name="hibernate.connection.url"
                value="jdbc:hsqldb:hsql://localhost/bank" />
      <property name="hibernate.connection.username" value="sa" />
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.HSQLDialect" />
      <property name="hibernate.show_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```


Übung

Führen Sie die Klasse `KundeTestJPA` aus. Versuchen Sie die Unterschiede zu Hibernate nachzuvollziehen. Betrachten Sie die Datei `persistence.xml` und vergleichen Sie mit `hibernate.cfg.xml`.

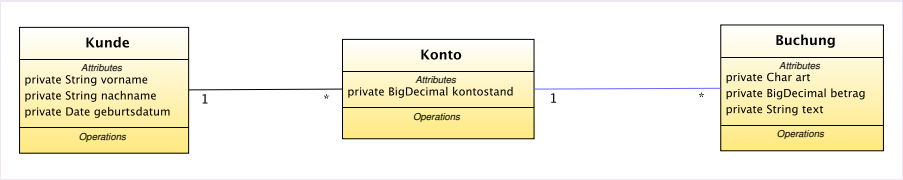
Tipps fürs Nachvollziehen

- Verwenden Sie ein Eclipse in ZIP-Form
  (bei exe wissen Sie nie, wer wo etwas hinschreibt)
- rufen Sie das Exe auf mit
  - -vm <SDK-Java> (KEIN JRE-Java!)
  - -data <Ihr Workspace>
- die kompletten Hibernate-Bibliotheken hibernate-libs.tgz können Sie als Eclipse-Projekt herunterladen.
- das Beispielprojekt in XML genauso: intro-xml.tgz
- und auch das Beispielprojekt mit Annotationen: intro-anno.tgz

Assoziationen

Beispielanwendung Bank

- Wir erweitern das Beispiel um Assoziationen



- Klassen Kunde, Konto, Buchung
- Konto mit Kontostand
- Buchung mit Art ('H','S'), Betrag und Text

Assoziationen mit XML-Mapping

Realisierung von Assoziationen

- Assoziationen werden wie ganz normale Properties behandelt
- d.h. Instanzvariable plus Getter und Setter
- entsprechend ihrer Kardinalität (Objekt oder Collection)
- die Collection-Art in Java muss zur Collection-Art im Mapping passen
- „zu-n-Assoziationen“ haben typischerweise Mengensemantik, also:
  `private Set<Konto> konten = new HashSet<Konto>();`
- und `<set>` im Mapping

Problem:

- `hbm2java` generiert nur
  `private Set konten = new HashSet();`
  im Default
- setzte `jdk5="true"` in `hbm2java`-Task

1-zu-n Kunde-Konto

```
<hibernate-mapping package="de.pmst.hibernate.assoc.model">
  <class name="Kunde" table="kunde">
    ...

    <set name="konten">
      <key>
        <column name="kunde" not-null="true" />
      </key>
      <one-to-many class="Konto" />
    </set>

    </class>
</hibernate-mapping>
```

Erläuterungen

```
<set name="konten">
```

Java-Property `Set<Konto>` `konten` mit Getter und Setter

```
<key>
  <column name="kunde" not-null="true" />
</key>
```

Fremdschlüssel in anderer Tabelle, not null

```
<one-to-many class="Konto" />
```

1-zu-n-Assoziation zur Klasse Konto

Verbesserungen

- um in beiden Richtungen navigieren zu können und die 1-zu-n- und die n-zu-1-Assoziation nicht als zwei unabhängige Assoziationen erscheinen zu lassen, wird das Attribut `inverse` gesetzt (`inverse="true"`)
- um nicht Kunde und Konto separat speichern zu müssen, werden die Konten automatisch gespeichert (`cascade="save-update"`)
- aus
  `session.save(kunde)`
  `session.save(konto)`
  wird dann
  `session.save(kunde)`

1-zu-n Kunde-Konto neu

```
<hibernate-mapping package="de.pmst.hibernate.assoc.model">
  <class name="Kunde" table="kunde">
    ...

    <set name="konten" inverse="true" cascade="save-update">
      <key>
        <column name="kunde" not-null="true" />
      </key>
      <one-to-many class="Konto" />
    </set>

    </class>
</hibernate-mapping>
```


Verwendung der Assoziation

- bisher: nur Getter und Setter der Assoziation
- evtl. schöner: eine Business-Methode, die die beiden Aufrufe zusammenfasst
- in der Klasse `Kunde`:

```
public void addKonto(Konto konto) {
    konto.setKunde(this);
    konten.add(konto);
}
```

- problematisch: dies ist ein Code-Zusatz zu generiertem Code
- im Projekt als extra Datei

Verwendung der Assoziation II

```
tx = session.beginTransaction();

Kunde neu = new Kunde();
neu.setVorname("Heidi");
neu.setNachname("Mustermann");
neu.setGeburtsdatum(new SimpleDateFormat("dd.MM.yyyy")
                    .parse("11.07.1960"));

Konto kto = new Konto();
kto.setKontostand(new BigDecimal(0));
neu.addKonto(kto);

session.save(neu);
tx.commit();
```


Verwendung der Assoziation III

```
Kunde kunde =
  (Kunde) session.load(Kunde.class, new Integer(1));
System.out.println("gelesener Kunde: "
                   + kunde.getNachname());
System.out.println("gelesene Konten: "
                   + kunde.getKonten());
for (Iterator iter = kunde.getKonten().iterator();
     iter.hasNext();) {
  Konto element = (Konto) iter.next();
  System.out.println("Konto: " + element.getId() + ",
                     Kontostand: "
                     + element.getKontostand());
}
```


Und jetzt im Schnelldurchgang noch den Rest:
die Klassen `Konto` und `Buchung`


```
<hibernate-mapping package="de.pmst.hibernate.assoc.model">
  <class name="Konto" table="konto">
    <id name="id" type="integer">
      <column name="id" />
      <generator class="identity" />
    </id>
    <property name="kontostand" type="big_decimal">
      <column name="kontostand" />
    </property>
    <many-to-one name="kunde" class="Kunde" >
      <column name="kunde" not-null="true" />
    </many-to-one>
    <set name="buchungen" inverse="true" cascade="save-update">
      <key>
        <column name="konto" not-null="true" />
      </key>
      <one-to-many class="Buchung" />
    </set>
  </class>
</hibernate-mapping>
```

```
<hibernate-mapping package="de.pmst.hibernate.assoc.model">
  <class name="Buchung" table="buchung">
    <id name="id" type="integer">
      <column name="id" />
      <generator class="identity" />
    </id>
    <property name="betrag" type="big_decimal">
      <column name="betrag" />
    </property>
    <property name="art" type="character">
      <column name="art" length="1" />
    </property>
    <property name="text" type="string">
      <column name="text" length="100" />
    </property>
    <many-to-one name="konto" class="Konto">
      <column name="konto" not-null="true" />
    </many-to-one>
  </class>
</hibernate-mapping>
```


Was bleibt noch?

- 1-zu-1, 1-zu-n, n-zu-1, n-zu-m ?
- 1-zu-1:
  mit `<one-to-one>` und einem spez. Primärschlüsselgenerator
  oder
  mit `<many-to-one>` und Fremdschlüsselspalte mit `unique`
- 1-zu-n: hatten wir
- n-zu-1: dasselbe, nur andere Richtung
- n-zu-m:
  geht mit `<many-to-many>` ...
  Erfahrung zeigt aber, dass in der Praxis meist noch einige Attribute an der Assoziation hängen (wie z.B. der Zeitstempel der Entstehung), so dass besser eine zusätzliche Assoziationsklasse dazwischengeschaltet wird. Dies entspricht dann der Tabellenstruktur.

Übung

1. *Öffnen Sie das Projekt „assoc-xml“*
2. *Starten Sie die Datenbank*
3. *Laden Sie beide Ant-Build-Files (im Projektverzeichnis und im Verzeichnis `hibernate`)*
4. *Generieren Sie die Datenbank und die POJOs*
5. *Kopieren Sie die Mapping-Dateien und die generierten POJOs in das Modell-Package*
6. *Fügen Sie die Dateien `...add` Ihren entsprechenden POJOs hinzu*
7. *Schauen Sie sich die beiden Klassen `Populate` und `LinkTest` im Test-Package an und führen Sie sie über Ant aus*

Übung

Erweitern Sie die Klasse Konto derart, dass eine Buchung mit einer (!) Methode vorgenommen werden kann.
Diese Übung ist optional

Assoziationen mit Annotationen

Änderungen

- es müssen „lediglich“ die Properties für die Assoziationen mit den entsprechenden Annotationen versehen werden
- es kann dazu kommen, dass dies relativ unübersichtlich wird
- nach meiner persönlichen Meinung aber immer noch besser, als über zwei Dateien verteilt

Kunde (Assoziation)

```
@Entity
public class Kunde implements Serializable {

    private Integer id;
    private Set<Konto> konten = new HashSet<Konto>();

    ...
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Integer getId() {
        return this.id;
    }

    ...

    @OneToMany(mappedBy="kunde", cascade=CascadeType.ALL)
    public Set<Konto> getKonten() {
        return this.konten;
    }
    public void setKonten(Set<Konto> konten) {
        this.konten = konten;
    }
}
```

Konto (Assoziation)

```
@Entity
public class Konto implements Serializable {

    private Kunde kunde;
    private Set<Buchung> buchungen = new HashSet<Buchung>();

    @ManyToOne
    @JoinColumn(name="kunde")
    public Kunde getKunde() {
        return this.kunde;
    }
    public void setKunde(Kunde kunde) {
        this.kunde = kunde;
    }

    @OneToMany(mappedBy="konto", cascade=CascadeType.ALL)
    public Set<Buchung> getBuchungen() {
        return this.buchungen;
    }
    public void setBuchungen(Set<Buchung> buchungen) {
        this.buchungen = buchungen;
    }
}
```

Buchung (Assoziation)

```
@Entity
public class Buchung implements Serializable {

    private Konto konto;

    ...

    @ManyToOne
    @JoinColumn(name="konto")
    public Konto getKonto() {
        return this.konto;
    }

    public void setKonto(Konto konto) {
        this.konto = konto;
    }
}
```


Verwendung

- bei der Verwendung bleibt alles beim Alten
- beim Code haben wir den Management-Code der Assoziationen unterschlagen, z.B. in der Klasse `Kunde`:

```
public void addKonto(Konto konto) {
    konto.setKunde(this);
    konten.add(konto);
}
```

- zur Wiederholung: Das ist normaler Java-Code, den Sie auch ohne Hibernate schreiben müssten

Übung

Um schon ein wenig in Richtung „anspruchsvolles“ Codieren zu kommen, haben wir in Projekt „assoc-anno“ die Klasse `HibernateUtil` eingeführt, die auf der nächsten Seite angegeben ist. Mit ihrer Hilfe ist die `SessionFactory` aus dem Anwendungs-Code verbannt.

Öffnen Sie das Projekt „assoc-anno“ und erweitern Sie die Klasse `KundeTest` um ein Konto und eine Buchung.

Kunde (Assoziation)

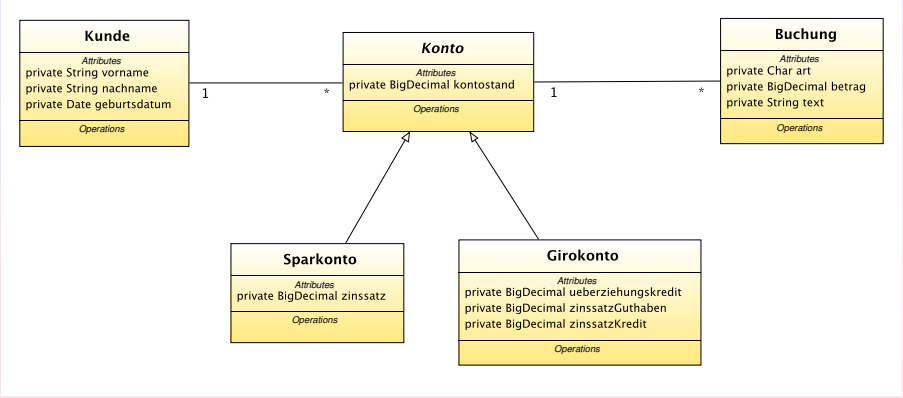
```
public class HibernateUtil {

  private static final SessionFactory sessionFactory;

  static {
    try {
      sessionFactory =
        new AnnotationConfiguration().configure()
                                     .buildSessionFactory();
    } catch (Throwable ex) {
      // im Ernstfall hier etwas loggen
      throw new ExceptionInInitializerError(ex);
    }
  }

  public static Session getSession() throws HibernateException {
    return sessionFactory.openSession();
  }
}
```

Vererbung

Das Problem

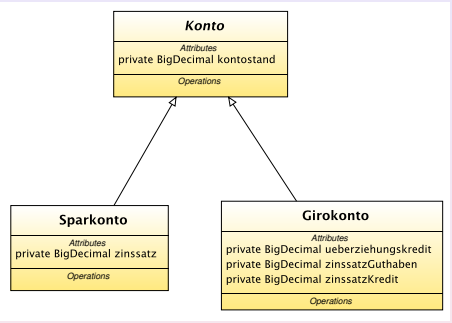
- bisheriger Ansatz für Persistenz: eine Tabelle pro Klasse
- bei objektorientierter Vererbung wird das komplizierter
- Problem: In der Objektorientierung gibt es eine *is-a-* und eine *has-a*-Beziehung. In Datenbanken nur die *has-a*-Beziehung.
- prinzipiell gibt es drei Möglichkeiten der Realisierung von Vererbung:
  - eine Tabelle je konkreter Klasse
  - eine Tabelle für gesamte Hierachie
  - eine Tabelle je Unterklasse

Das erweiterte Beispiel



4.1 (a) Eine Tabelle je konkreter Klasse

Idee



- eine Tabelle für jede nicht abstrakte Klasse
- alle Properties, auch die geerbten, werden Spalten dieser Tabelle

Sparkonto:

ID	SPARKONTOSTAND	ZINSSATZ
...		

Girokonto:

ID	KONTOSTAND	UEB_Kredit	ZINS_HABEN	ZINS_SOLL
...				

Vorteile

- Queries sind einfach und performant

Nachteile

- keine Polymorphie
- sehr problematisch bei Assoziationen (wie in diesem Fall)
- alle Konten $\Rightarrow$ zwei Selects

Eclipse-Projekt *vererbung-1*

Die Syntax

Die Oberklasse:

```
@MappedSuperclass
public abstract class Konto implements Serializable {

    private Integer id;
    private BigDecimal kontostand;
...
```

Die geerbten Properties können bei Bedarf umbenannt werden

```
@Entity
@AttributeOverride(name="kontostand",
    column=@Column(name="sparkontostand"))
public class Sparkonto extends Konto {

    private BigDecimal zinssatz;
```


4.1 (b) Eine Tabelle je konkreter Klasse mit Union

- selbe Tabellenstrukturen
- existiert (optional) auch bei JPA/EJBs (nächsten beiden Alternativen Standard)
- lässt teilweise Polymorphie zu (n-zu-1)
- Eclipse-Projekt *vererbung-2*

Die Syntax

Die Oberklasse:

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Konto implements Serializable {

    private Integer id;
    private BigDecimal kontostand;
...
```

Die Unterklasse (nichts zu tun):

```
@Entity
public class Sparkonto extends Konto {

    private BigDecimal zinssatz;
...
```


Verwendung Polymorphie

Ausschnitt Klasse KontoTest (vererbung-2)

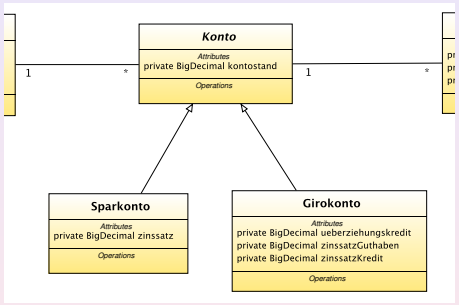
```
kunde = (Kunde) session.load(Kunde.class, 1);

// Polymorphie geht!
for (Iterator iter = kunde.getKonten().iterator();
     iter.hasNext();) {
  Konto element = (Konto) iter.next();
  System.out.println("Konto: " + element.getId() + " "
                     + element.getClass());
}
```

Ausgabe

```
[java] Konto: 1 class de.pmst.hibernate.assoc.model.Sparkonto
[java] Konto: 2 class de.pmst.hibernate.assoc.model.Girokonto
```


4.2 Eine Tabelle für eine ganze Hierarchie

Idee



- Oberklasse und *alle* Unterklassen werden in *eine* Tabelle geschrieben
- der konkrete Typ, der durch eine Tabellenzeile repräsentiert wird, steht in einer *Diskriminatorspalte*
- dies ist die beste Alternative bzgl. Performanz und Einfachheit!!!

Konto:

ID	KONTO_ART	KONTOSTAND	ZINSSATZ	UEB_KREDIT	ZINS_GUT	ZINS_SOLL
...						

Analyse

Vorteile

- performant
- einfach
- änderungsfreundlich

Nachteile

- die Spalten für die Unterklassenproperties können nicht „`not null`“ sein
- damit je nach Projektanforderungen evtl. nicht verwendbar !
- nicht normalisiert

Beispiel (vererbung-3)

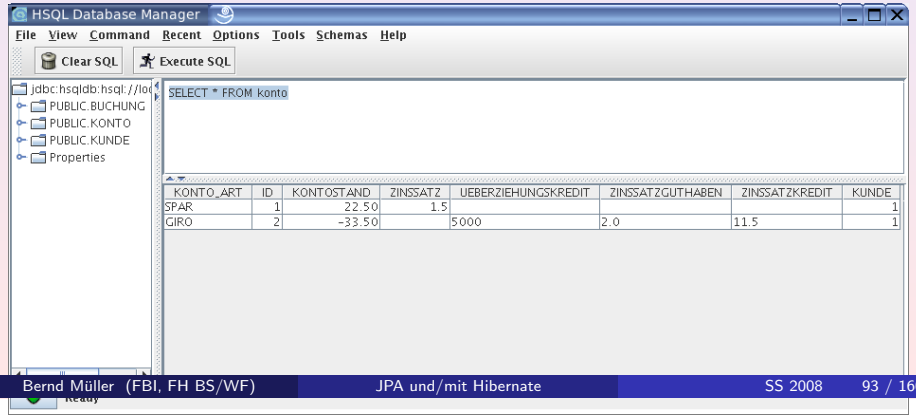
```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "KONTO_ART",
            discriminatorType = DiscriminatorType.STRING)
public class Konto implements Serializable {

    private Integer id;
    private BigDecimal kontostand;
    private Kunde kunde;
    private Set<Buchung> buchungen = new HashSet<Buchung>();
...
```


Beispiel II

```
@Entity
@DiscriminatorValue("SPAR")
public class Sparkonto extends Konto {

    private BigDecimal zinssatz;
```

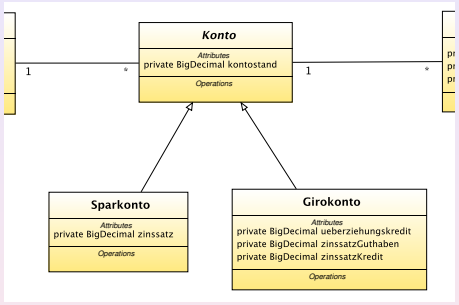
Beispiel III

- ganz normale Java-Programmierung mit Polymorphie (ohne Code, aber im Projekt vererbung-3):
  - Kunde erzeugen
  - Spar- und Girokonto erzeugen
  - je eine Buchung für Spar- und Girokonto



4.3 Eine Tabelle je Unterklasse

Idee



- Vererbung wird durch Fremdschlüsselbeziehungen repräsentiert
- jede Klasse (auch abstrakte), die Properties hat, erhält eine eigene Tabelle

Konto:

ID	KONTOSTAND
...	

Sparkonto:

ID	ZINSSATZ
...	

Girokonto:

ID	UEBERZIEHUNGSKREDIT	ZINSSATZGUTHABEN	ZINSSATZKREDIT
...			

Beispiel I

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Konto implements Serializable {

    private Integer id;
    private BigDecimal kontostand;
    private Kunde kunde;
    private Set<Buchung> buchungen = new HashSet<Buchung>();

    ....
```

Beispiel II

```
@Entity
public class Sparkonto extends Konto {

    private BigDecimal zinssatz;

    ...
```


Analyse

Vorteile

- Not-Null-Constraints erlaubt
- normalisiert

Nachteile

- Joins nötig (Performanz)

Fazit

- es gibt 3 (4) Realiserungsalternativen für Vererbung
- Sie müssen sich je nach Anforderungen entscheiden
- d.h. es kann sein, dass Sie es in einem Projekt so, in einem anderen Projekt anders machen
- es kann sogar in einem Projekt unterschiedlich gemacht werden

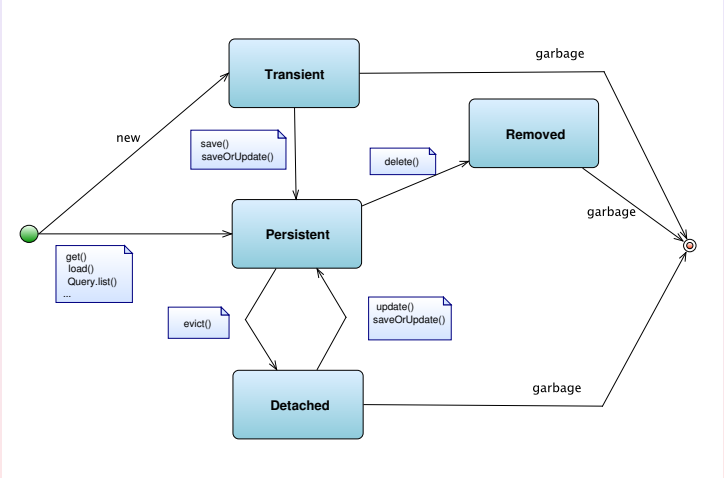
Lebenszyklus, Identität, Cache, ...

Lebenszyklus

Motivation

- bisher nur strukturelle/statische Aspekte angeschaut: Objektstruktur, Assoziationen, Vererbung
- jetzt *dynamische* Aspekte anschauen
- Verständnis unbedingt notwendig für effiziente Persistenzlösungen
- Lebenszyklus:
  - wie wird ein Objekt persistent?
  - wie verliert es seine Persistenz?
- zur Wiederholung: Hibernate ist eine transparente Persistenzlösung, d.h. ein Objekt weiß nichts von seiner Persistenz (ist POJO)

Der Lebenszyklus im Überblick



Objektzustände

Transient Nach Aufruf des `new`-Operators ist ein Objekt zunächst transient. Es ist nicht mit einer Tabellenzeile verbunden. Ein transientes Objekt kann durch den Aufruf eines Persistenzmanagers persistent werden oder indem es von einem persisteten Objekt referenziert wird.

Persistent Ein persistentes Objekt hat eine Datenbankidentität (PK, siehe nächsten Abschnitt). Jedes persistente Objekt hat einen Persistenzkontext, mit dem Änderungen am Objekt festgestellt werden können.

Removed Ein Objekt kann explizit gelöscht werden. Es wird am Ende der Unit-of-Work in der Datenbank gelöscht. Solange ist es nur als „zu löschen“ gekennzeichnet.

Detached Wenn der Persistenzkontext eines Objekts geschlossen wird, existiert das Objekt als Java-Objekt weiter, ist aber von der Datenbank *losgelöst*. Hibernate garantiert dann nicht mehr, dass das Objekt mit der Datenbank synchronisiert wird.

Der Persistenzkontext

- eine `Session` hat einen Persistenzkontext
- alle persistenten Objekte (dieser Session) werden in diesem Kontext verwaltet
- automatic Dirty-Checking
- transactional Write-Behind
- first-level Cache
- Garantie der Java-Objektidentität

Dirty-Checking und transactional Write-Behind

- der Zustand eines persistenten Objekts wird am Ende einer Unit-of-Work mit der Datenbank synchronisiert
- es werden *nicht* für alle persistenten Objekte INSERT/UPDATE/DELETE-Statements abgesetzt, sondern nur für Objekte, die dirty sind
- dirty: Objekt hat Änderungen, die noch nicht in der Datenbank stehen
- das Schreiben in die Datenbank wird möglichst lange herausgezögert
- das nennt man *transparent transaction-level write-behind*
- dadurch können Locks in der Datenbank zeitlich minimiert werden
- Bemerkung: Es werden immer alle Properties aktualisiert. Hibernate kennt aber die Properties, die geändert wurden und kann ein UPDATE-Statement erzeugen, das dies berücksichtigt. Das lohnt sich in der Regel nur bei Tabellen mit sehr vielen Spalten. Wird durch `dynamic-update="true"` bzw. `dynamicUpdate = true` eingeschaltet.

Übung

Überprüfen Sie den dynamischen Update. Dazu führen Sie z.B. im Projekt intro-anno/intro-xml (Kopie!) eine Änderung durch Aufruf eines Setters aus und überprüfen die durch Hibernate erzeugte UPDATE-Anweisung. Dann fügen Sie, falls Sie Annotationen verwendet haben

```
@org.hibernate.annotations.Entity(dynamicUpdate = true)
```

ein. Falls Sie mit dem XML-Mapping arbeiten, fügen Sie

```
dynamic-update="true"
```

im `class`-Element hinzu.

Identität

Wiederholung Java

- Identität (==)
  - bei Werttypen: selber Wert
  - bei Referenztypen: selbe Referenz
- Gleichheit (`equals()`)
  - Anwendungsdefiniert
  - Default in `java.lang.Object` ist Identität, d.h. Sie erben diese Gleichheitsdefinition
- benötigen Sie eine andere Gleichheit, müssen Sie `equals()` überschreiben und, ganz wichtig, `hashCode()` ebenfalls (gleiche Objekte müssen gleichen Hash-Code haben!)

Identität mit Hibernate

- Hibernate kann Identität nur im Persistenzkontext (Session) garantieren

```
session1 = ...openSession();
Kunde k1 = session1.get(Kunde.class, 4711);
Kunde k2 = session1.get(Kunde.class, 4711);
...
... k1 == k2 // ergibt true
session1.close();
session2 = ...openSession();
Kunde k3 = session2.get(Kunde.class, 4711);
...
... k1 == k3 // ergibt false
session2.close();
```


Identität mit Hibernate II

- Probleme mit detached Objekten

```
Set alle = new HashSet();
alle.add(k1);
alle.add(k2);
alle.add(k3);
```

- Frage: Wieviele Elemente enthält die Menge?
- Antwort: (API-Doc!)

Identität mit Hibernate III

- Gleichheit über Primärschlüssel: `k1.getId().equals(k2.getId())`
- dies geht allerdings nur bei persistenten Objekten
- d.h. falls Sie transiente Objekte (nach new-Aufruf) einer Menge hinzufügen, funktioniert dies nicht
- andere Alternative ist über Anwendungs-OB
- Sie definieren einen anwendungsspezifischen Schlüsselkandidaten, der als ID dient

Übungen zu Lebenszyklus, Session, Identität

Übung

Führen Sie die beiden Ladebefehle in einer Session aus und danach in zwei Sessions. Überzeugen Sie sich davon, dass ein bzw. zwei SELECTs abgesetzt werden.

Übung

Laden mit load() erzeugt ein Proxy. Ein Zugriff außerhalb der Session führt zu einer LazyInitializationException. Überprüfen Sie diese Aussage.

Übung

Laden mit get() lädt das Objekt. Ein Zugriff außerhalb der Session führt zu keiner LazyInitializationException. Überprüfen Sie diese Aussage.

Übung

Assoziationen werden im Default immer lazy geladen. Ein Zugriff außerhalb der Session führt zu einer LazyInitializationException. Verlangen

Übungen zu Lebenszyklus, Session, Identität II

Übung

Assoziationen werden im Default immer lazy geladen. Ein Zugriff außerhalb der Session führt zu einer LazyInitializationException. Verlangen Sie Eager-Loading. Machen Sie sich die Auswirkungen klar!!!

Cache

Wozu Caches?

- Was ist ein Cache?
  - Ein Cache hält eine Repräsentation eines *(möglichst) aktuellen* Datenbankzustands in Anwendungsnähe
  - d.h. im Speicher oder Platte des Anwendungs-Servers
- Hilft nur bei vielen Lesezugriffen
- ... und verhindert dort *unnötige* Datenbankzugriffe
- Ist meist geschichtet

First-level Cache

- wird durch die Session realisiert
- verhindert Lesezugriff für ein Objekt, das schon in der Session ist (Beispiel hatten wir schon)

Second-level Cache

- Ist pluggable in Hibernate
- Für mehrere Threads, JVMs, Cluster
- Mögliche (eingebaute) Alternativen (alle open-source):
  - EHCache
  - OpenSynphony OSCache
  - SwarmCache
  - JBoss Cache

Tipps

- Man muss sehr genau wissen, was man tut
- In der Regel sollte man zuerst nichts tun
- Gefahren:
  - Caching beeinflusst Transaction-Isolation negativ
  - Caching führt zu korrupten Daten
  - Caching verlangsamt die Anwendung
  - ...

Logging

Hibernate-Konfiguration (hibernate.cfg.xml)

`hibernate.show_sql` Gibt Logging aller generierten SQL-Statements frei

`hibernate.format_sql` Formatiert diese (für mich nicht sinnvoll)

`hibernate.use_sql_comments` Gibt den Ursprung des SQLs an

Log4j

- Hibernate logt über Apache Commons-Logging
- ... ein kleiner Layer über Apache Log4j oder JDK 1.4 Logging
- Default ist Log4j
- Datei `log4j.properties` im Quellverzeichnis
- `log4j.logger.org.hibernate.type=debug`
  logt Bind-Parameter von JDBC-Queries
  (INSERT INTO ... VALUES(... ?, ?, ?))

Übung

Probieren Sie alle genannten Optionen aus.

Queries

Überblick

- Query-Möglichkeiten sehr wichtig für guten Datenzugriffs-Code
- ebenfalls für die Performanz
- falls Sie schon sehr viel SQL geschrieben haben, werden Sie beim Einsatz eines OR-Mappers fürchten, diese Möglichkeit zu verlieren
- dies ist bei Hibernate nicht der Fall, es gibt genügend Alternativen
- bis hin zu nativen SQL-Queries (wenn's ganz schlimm kommt)
- neben JPA QL (Query Language) und HQL (Hibernate Query Language)
- gibt es auch die Möglichkeit *Query by Criteria* (native Hibernate)
- und *Query by Example* (native Hibernate)
- und wie gesagt SQL

Grundsätzliches Vorgehen

- Erzeugen der Query (mit Projektion und Restriktion)
- (in der Regel) Binden von Laufzeitargumenten an Query-Parameter
- Ausführen der Query und Verwendung der Ergebnisdaten

JPA QL und HQL

Die ersten Schritte

- HQL erinnert stark an ein „objektorientiertes“ SQL
- Beispiele:

```
from Buchung
FROM Buchung b
from Buchung b where b.betrag < 100
from Buchung b where b.betrag between 50 and 200
```

- nicht case-sensitive
- *Buchung* ist die Klasse, nicht die Tabelle

Der Code

- zuerst die Session nach einem Query-Objekt fragen
- optional die Query mit Parametern versorgen
- dann durch die Ergebnismenge iterieren

```
// alle Buchungen holen:
Query query = session.createQuery("from Buchung");
for (Iterator iter = query.list().iterator(); iter.hasNext();) {
    Buchung b = (Buchung) iter.next();
    System.out.println(b.getText() + " " + b.getBetrag());
}
```


Parameter

- zwei Arten von Parametern
  - Parameter mit Bezeichnern
  - Positionsparameter (mit ?, wie PreparedStatements in JDBC)

```
Query query = session.createQuery(
   "from Buchung b where b.betrag between :von and :bis");
query.setInteger("von", 50);  // realistischer: Variable
query.setInteger("bis", 150);
for (Iterator iter = query.list().iterator(); iter.hasNext();) {
  Buchung b = (Buchung) iter.next();
  System.out.println(b.getText() + " " + b.getBetrag());
}
```

- Alternative ist `setParameter()`
- Typ wird über Reflection gefunden
- im Beispiel dann:
  `query.setParameter("von", new BigDecimal("50"));`

Ein einziges Ergebnis

- bei manchen Queries weiß man, dass es nur ein Ergebnis (Zeile) gibt
- größter/kleinster Wert
- Primärschlüssel / unique

```
Query query = session.createQuery(
      "from Buchung b order by b.betrag desc");
Buchung b = (Buchung) query.setMaxResults(1).uniqueResult();
System.out.println(b.getText() + " " + b.getBetrag());
```


Blättern

- häufige Anforderung: seitenweises Blättern
- setFirstResult() / setMaxResults()

```
Query query = session.createQuery(
        "from Kunde k order by b.nachname");
query.setFirstResult(120);
query.setMaxResults(20);
List l = query.list();
```

Der Rest

- fast alles, was Sie sonst noch kennen, ist vorhanden
- Aggregatsfunktionen: `count()`, `min()`, `max()`, `sum()`, `avg()`
  - select count(*) from Buchung b
  - select avg(b.betrag) from Buchung b
- Expressions:
  - Mathematik: +, -, *, /
  - Vergleiche: =, >=, >, ...
  - Logik: and, or, not
  - Klammerung
  - in, not in, between, is null, is not null, is empty, ...
  - Funktionen: `substring()`, `trim()`, `lower()`, `upper()`, `length()`

Der Rest II

- order by (hatten wir schon)
- group by

```
select avg(b.betrag), b.art
from Buchung b group by b.art
```

- Subqueries

```
from Buchung buch
where buch.betrag >=
      (select avg(b.betrag) from Buchung b)
```


Eigentlich nicht erwähnenswert

- Hibernate ist voll objektorientiert
- daher sind Queries polymorph
- `from Konto k` liefert *alle* Konten, insbesondere auch Giro- und Sparkonten

Übung

Holen Sie alle Kunden aus der Datebank.

Übung

Holen Sie alle Kunden aus der Datebank, deren Nachname einem bestimmten Muster entspricht.

Übung

Realisieren Sie folgende Abfragen:

- *alle Buchungen, deren Betrag zwischen a und b liegt*
- *alle Haben-Buchungen*

Übung

Man kann auch alle persistenten Objekte holen, nämlich mit
`from java.lang.Object o`
Es werden dann alle Objekte aus der Datenbank geholt, für die ein Mapping existiert.
Schreiben Sie ein Programm, das ausgibt wieviele Kunden, Konten und Buchungen es gibt und dazu nur eine (die obige) HQL-Query benutzt.
In einer realen Anwendung würden Sie dies natürlich nicht tun ;-)

Query by Criteria (QBC)

Die Idee

- man möchte ohne SQL-artige Syntax auskommen
- für den nicht geübten SQL-Entwickler häufig einfacher
- Queries werden in Java programmiert und zwar durch das Erzeugen und Kombinieren von Objekten (in der richtigen Reihenfolge)
- dazu wird zunächst ein `Criteria`-Objekt erzeugt, an das dann `Criterions` gehängt werden (wird gleich klarer)

QBC Beispiele

- Session auch Factory für `Criteria`
- Methode: `createCriteria()`
- einfaches Beispiel: alle Buchungen

```
Criteria crit = session.createCriteria(Buchung.class);
for (Iterator iter = crit.list().iterator(); iter.hasNext();) {
  Buchung b = (Buchung) iter.next();
  ...
}
```

QBC Beispiele II

- Kriterien werden mit der Factory `org.hibernate.criterion.Restriction` erzeugt
- es gibt die üblichen Verdächtigen: eq(), ge(), gt(), in(), like(), not(), ...

```
Criteria crit = session.createCriteria(Buchung.class);
Criterion restriction =
    Restrictions.eq("betrag", new BigDecimal("55.55"));
crit.add(restriction);
...
List l = crit.list();
```


QBC Beispiele III

- häufig hintereinander gehängt:

```
Criteria crit = session.createCriteria(Buchung.class);
crit.add(Restrictions.between("betrag",
                              new BigDecimal("0"),
                              new BigDecimal("200")))
    .add(Restrictions.eq("art", 'S'))
    .add(Restrictions.like("text", "%Rechnung%"));
...
List l = crit.list();
```

- also alle Buchungen
  - deren Betrag zwischen 0 und 200 ist
  - deren Art 'S' ist
  - deren Text *Rechnung* enthält
- bei vielen Fabrikmethoden ist erster Parameter ein Property-Name
- also Vorsicht: nicht typsicher und nicht refaktorisierungssicher
- ist ein SQL-String aber auch nicht

Query by Example (QBE)

Die Idee

- für Suchanfragen, die z.B. optional vom Benutzer auszuwählende Anfragekriterien nutzen, mit anderen Worten „die *sehr* dynamisch sind, ist HQL wenig geeignet
- QBC ist etwas besser geeignet, da kein SQL-String zusammengebaut werden muss
- mit QBE definiert man über das Criteria-Interface ein Objekt, das bestimmte Eigenschaften (Property-Werte) hat und sucht dann nach allen Objekten, die in diesen Eigenschaften übereinstimmen

QBE Beispiel I

- bei einem „leeren“ Objekt ist das Beispiel „am allgemeinsten“ und liefert alle Buchungen
- `org.hibernate.criterion.Example` ist eine Factory für Beispiele

```
Buchung beispielbuchung = new Buchung();

session = HibernateUtil.getSession();
tx = session.beginTransaction();
Criteria crit = session.createCriteria(Buchung.class);
crit.add(Example.create(beispielbuchung));
...
   crit.list()
```


QBE Beispiel II

- jetzt im Beispiel die Buchungsart 'S', also alle Sollbuchungen

```
Buchung beispielbuchung = new Buchung();
beispielbuchung.setArt('S');

session = HibernateUtil.getSession();
tx = session.beginTransaction();
Criteria crit = session.createCriteria(Buchung.class);
crit.add(Example.create(beispielbuchung));
...
   crit.list()
```


QBE Beispiel III

- das Setzen des Buchungstexts auf „Rechung“ würde nur Buchungen mit *exakt* diesem Text finden
- die Methode `enableLike(<matchcode>)` vergleicht mit 'like', allerdings für alle String-Properties

```
Buchung beispielbuchung = new Buchung();
beispielbuchung.setArt('S');
beispielbuchung.setText("Rechnung");

session = HibernateUtil.getSession();
tx = session.beginTransaction();
Criteria crit = session.createCriteria(Buchung.class);
crit.add(Example.create(beispielbuchung)
                .enableLike(MatchMode.ANYWHERE));
...
   crit.list()
```


QBE Beispiel III

- da die Klasse `Example` das Interface `Criterion` implementiert, können Sie Query by Example und Query by Criteria mischen

```
Buchung beispielbuchung = new Buchung();
beispielbuchung.setArt('S');
beispielbuchung.setText("Rechnung");

session = HibernateUtil.getSession();
tx = session.beginTransaction();
Criteria crit = session.createCriteria(Buchung.class);
crit.add(Example.create(beispielbuchung)
                .enableLike(MatchMode.ANYWHERE))
    .add(Restrictions.between("betrag",
                              new BigDecimal("100"),
                              new BigDecimal("200")));

...
   crit.list()
```

Native SQL

Automatisches ResultSet-Handling

- Sie können direkt über JDBC (und ohne Hibernate) SQLs gegen die Datenbank schicken
- Sie verzichten dann aber auf das automatische ResultSet-Handling
- d.h. ein Ergebnis kann auf ein Objekt bzw. eine Liste von Objekten gemapped werden
- die Session dient wieder als Query-Fabrik

SQL Beispiel I

- ein Select liefert eine Liste von Objekten zurück
- diese ist damit eigenlich unbrauchbar (später machen wir sie brauchbar)

```
SQLQuery query = session.createSQLQuery("select * from Buchung");
List ergebnis = query.list();
```

- wenn Hibernate weiß, welches Objekt erwartet wird, kann über das Mapping das Property und zugehörige Spalte bestimmt werden
- es kommen dann richtige Entities zurück

```
SQLQuery query =
    session.createSQLQuery("select * from Buchung");
List ergebnis = query.addEntity(Buchung.class).list();
for (Iterator iter = ergebnis.iterator(); iter.hasNext();) {
    Buchung b = (Buchung) iter.next();
```


SQL Beispiel II

- das ganze funktioniert auch für komplexere Queries
- und Parameter sind auch zugelassen

```
SQLQuery query = session.createSQLQuery(
    "select ko.* from Konto ko, Kunde ku"
    + " where ko.kunde = ku.id and ku.nachname = :nachname");

List ergebnis = query.addEntity(Konto.class)
                     .setParameter("nachname", "Mustermann").list(
for (Iterator iter = ergebnis.iterator(); iter.hasNext();) {
  Konto k = (Konto) iter.next();
  ...
```

Skalare Werte

- SQL-Queries ohne Entity-Mapping sind nicht ganz sinnlos
- das Ergebnis ist eine Liste von Object-Arrays mit skalaren Typen (String, Zahl, Zeitstempel)
- mit `addScalar()` wird ein SQL-Alias als ein skalarer Wert definiert, dessen Typ automatisch bestimmt wird

```
SQLQuery query = session.createSQLQuery(
    "select k.vorname as vorname from Kunde k");
List ergebnis = query.addScalar("vorname").list();
```

Bei mir funktioniert das folgende identisch

```
SQLQuery query = session.createSQLQuery(
    "select k.vorname from Kunde k");
List ergebnis = query.list();
```

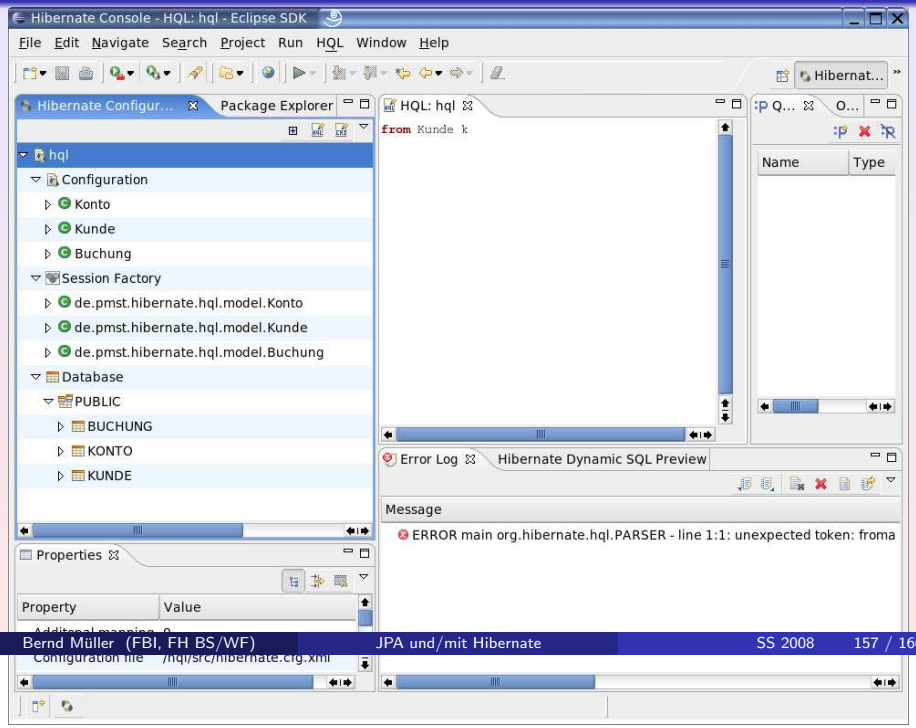

Übung

Testen Sie die vorgestellten Anfragen.

JBoss-IDE

IDE und Tools

- die JBoss-IDE enthält die Hibernate-Tools, die wir bereits als Ant-Tasks verwendet haben
- und baut eine graphische Oberfläche darüber
- man kann also
  - aus XML-Mapping das DDL und Java generieren
  - aus Annotationen das DDL generieren
- die richtig guten Features sind allerdings
  - das Reverse-Engineering
  - und ein syntaxgestützter HQL-Editor und Ausführungsumgebung



Übung

Praktische Übungen mit der JBoss-IDE

Literatur

- „Die Bibel“: Christian Bauer, Gavin King. *Java Persistence with Hibernate*. Manning, 2006.
- Christian Bauer, Gavin King. *Hibernate in Action*. Manning, 2004. (Vorgängerversion, ohne Annotationen)
- JPA (EJB 3.0)
- Hibernate Reference
- Hibernate Annotations
- Hibernate Tools
- www.hibernate.org