Konzepte der Datenbanktechnologie

Prof. Dr. U. Hoffmann
FH Wedel

Hibernate 1

Gliederung

Hibernate

- Eigenschaften
- Persistenz
- Konfiguration
- Mapping
- Vererbung
- Änderungsmanagement
- Laden und Zwischenspeichern
- HQL–Anfragen
- Criteria–Anfragen

Wesentliche Eigenschaften von Hibernate

- Transparente Persistenz
- Transitive Persistenz (Persistenz per Erreichbarkeit)
- Detached Object Support
- Inheritence mapping strategies
- Intelligent fetching and caching
- Automatic dirty object checking
- Unterschiedliche Anfragekonzepte: Queries und Criteria

Transparente Persistenz

- **Jede Klasse darf persistent sein.**
  Es gibt keine Anforderungen an Superklassen, von denen die persistente Klasse erben muss oder an Interfaces, welche sie implementieren muss.
- **Persistente Klassen dürfen auch in anderen Umgebungen benutzt werden**, z.B. in Testumgebungen.
- **Persistente Objekte haben keine für den Anwender sichtbare Zustände.**
  Sie benötigen in der Anwendungsschicht keine besonderen Behandlungen. Alle persistenzspezifischen Aspekte werden in den Interfaces `Session` oder `Query` verwaltet.

Fazit:

Der Anwendungsprogrammierer kann Hibernate als Black Box betrachten: Der konkrete Persistenzmechanismus wird verborgen.

Transitive Persistenz

Verallgemeinerung der Persistenz durch Erreichbarkeit:

Für alle Felder können individuell folgende Optionen eingestellt werden (in den XML-Metadaten):

- Feld soll überhaupt nicht in der Datenbank verändert werden, wenn eine Datenbankänderung im referenzierenden Objekt vorgenommen wird.
- Feld soll genau dann in der Datenbank aktualisiert werden, wenn eine Aktualisierung im referenzierenden Objekt vorgenommen wird.
- Feld soll genau dann aus der Datenbank gelöscht werden, wenn eine Löschung des referenzierenden Objekts vorgenommen wird.
- Feld soll genau dann in der Datenbank verändert werden, wenn eine Datenbankänderung im referenzierendenObjekt vorgenommen wird.

In JDO entspricht das der Unterteilung in First-Class und Second-Class-Objects, dort allerdings wesentlich grober.

Hibernate–Konfiguration

Die Hibernate–Konfiguration

- erfolgt durch ein Property–File (hibernate.properties) oder
- durch ein XML–File (hibernate.properties.xml)
- Festlegung der Datenbankbverbindung, des SQL–Dialekts
- Angabe des JDBC–Treibers
- weitere Angaben
- Konfigurationsfile muss im CLASSPATH stehen.

Hibernate–Konfiguration: Beispiel

```
hibernate.properties
hibernate.dialect=net.sf.hibernate.dialect.HSQLDialect
hibernate.connection.driver_class=org.hsqldb.jdbcDriver
hibernate.connection.url=jdbc:hsqldb:data/events
hibernate.connection.username=sa
hibernate.connection.password=
transaction.factory_class=
       net.sf.hibernate.transaction.JDBCTransactionFactory

hibernate.cache.provider_class=
       net.sf.hibernate.cache.HashtableCacheProvider
hibernate.hbm2ddl.auto=update
hibernate.show.sql=true
```

Konfigurations–Beispiel

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

  <session-factory>

    <!--  MySQL connection -->
    <property name="connection.url">jdbc:mysql://host/database</property>
...
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="dialect">org.hibernate.dialect.MySQLMyISAMDialect</property>
...
    <property name="show_sql">true</property>
...
    <!-- this will create the database tables for us -->
    <property name="hibernate.hbm2ddl.auto">create</property>
    <mapping resource="Mappingfile.hbm.xml" />

  </session-factory>

</hibernate-configuration>
```

Objekt–relationales Mapping

- Die Metadaten für das objekt–relationale Mapping werden in Mapping–Dokumenten in XML notiert.
- Endung: `.hbm.xml`
- Inhalt:
  - Beschreibung der persistenten Daten einer Klasse
  - Beziehung der Klasse zu anderen Klassen.

Mapping–Beispiel

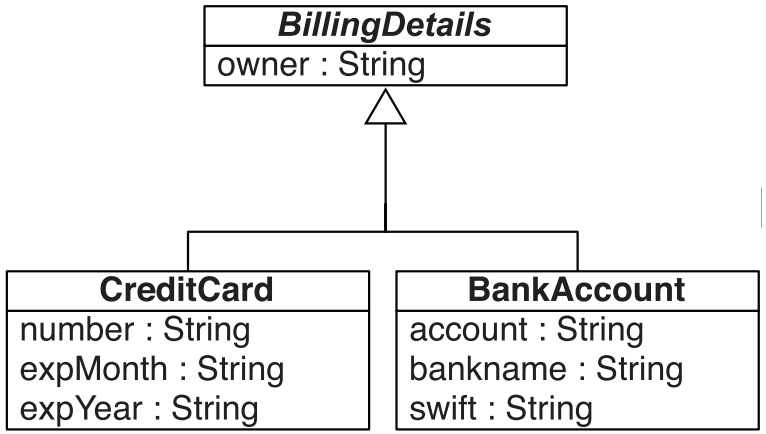
```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sourceforge.net/
                        hibernate-mapping-2.0.dtd">
<hibernate-mapping>
<class name="Eggs" table="EGGS">
<id name="id" type="int">
      <meta attribute="scope-set">protected</meta>
      <generator class="native"></generator>
</id>
<property name="spam" type="string" not-null="true"/>
</class>
</hibernate-mapping>
```

Strategien zum Mapping von Vererbung

Hibernate ermöglicht es, zwischen folgenden Abbildungsmechanismen für die Vererbung zu wählen (einstellbar für jede Klasse in den Metadaten):

- **Table per concreteclass**
  Die Klasse und all ihre Unterklassen bekommen jeweils eine eigene Tabelle.
- **Table per classhierarchy**
  Es gibt eine einzige Tabelle für die gesamte Klassenhierarchie. Die Zugehörigkeit zu einer bestimmten Klasse wird in einer separaten Spalte abgespeichert.
- **Table per subclass**
  Es gibt gesonderte Tabellen für die Unterklassen.
  Dort werden aber nur die Felder abgespeichert, die nicht geerbt werden.

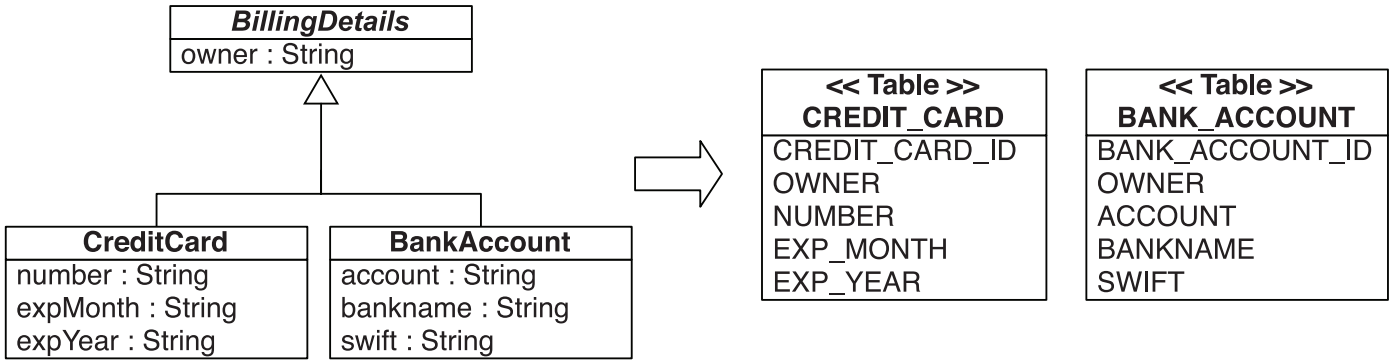
Mapping von Vererbung



Eine Vererbungsstruktur kann auf unterschiedliche Weisen relational abgebildet werden:

- **Table per concrete class**
- **Table per class hierarchy**
- **Table per subclass**

Table per concrete class (impliziter Polymorphismus)



- Konkrete Klassen werden *normal* abgebildet.
- Hibernate erkennt Vererbungshierarchie.
- Problem polymorphe Assoziationen:
  kann nicht durch einfache Fremdschlüssel-Beziehung
  ausgedrückt werden (Schlüssel in unterschiedlichen Tabellen).
- Problem polymorphe Abfragen:
  resultieren in mehrere SQL–Abfragen:

```
select CREDIT_CARD_ID, OWNER, NUMBER, EXP_MONTH, ...
from CREDIT_CARD

select BANK_ACCOUNT_ID, OWNER, ACCOUNT, ...
from BANK_ACCOUNT
```

Table per concrete class (explizit als Union)

Beispiel (Table per concrete class: Union)

```
<hibernate-mapping>
  <class name="BillingDetails" abstract="true">
   <id name="id"
      column="BILLING_DETAILS_ID"
      type="long">
      <generator class="native"/>
   </id>
   <property name="name" column="OWNER" type="string"/>
       ...
   <union-subclass
       name="CreditCard" table="CREDIT_CARD">
       <property name="number" column="NUMBER"/>
       <property name="expMonth" column="EXP_MONTH"/>
       <property name="expYear" column="EXP_YEAR"/>
   </union-subclass>
   <union-subclass
      name="BankAccount" table="BANK_ACCOUNT">
   ...
  </class>
</hibernate-mapping>
```

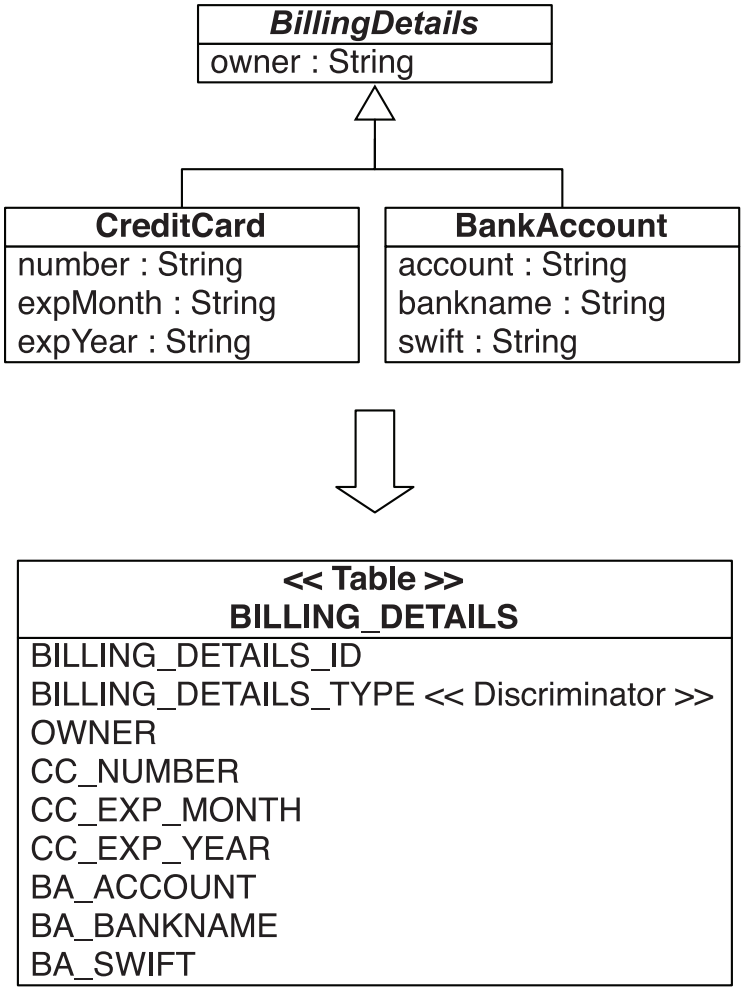
Table per conrete class (explizit als Union)

- Polymorphe Abfrage (nach `BillingDetails`): SQL

```
select
    BILLING_DETAILS_ID, OWNER,
    NUMBER, EXP_MONTH, EXP_YEAR,
    ACCOUNT, BANKNAME, SWIFT
    CLAZZ_
from
  ( select
      BILLING_DETAILS_ID, OWNER,
      NUMBER, EXP_MONTH, EXP_YEAR,
      null as ACCOUNT, null as BANKNAME, null as SWIFT,
      1 as CLAZZ_
    from
      CREDIT_CARD
    union
    select
        BILLING_DETAILS_ID, OWNER,
        null as NUMBER, null as EXP_MONTH, null as EXP_YEAR, ...
        ACCOUNT, BANKNAME, SWIFT,
        2 as CLAZZ_
      from
        BANK_ACCOUNT
   )
```

- Polymorphe Assoziationen auch über Union–Abfrage.

Table per class hierarchy



- Die gesamte Klassenhierarchie wird auf eine Tabelle abgebildet.

Table per class hierarchy

Beispiel (Table per class hierarchy)

```
<hibernate-mapping>
 <class name="BillingDetails" table="BILLING_DETAILS">
   <id name="id" column="BILLING_DETAILS_ID"
       type="long">
       <generator class="native"/>
   </id>
   <discriminator column="BILLING_DETAILS_TYPE"
                  type="string"/>
   <property name="owner" column="OWNER" type="string"/>
       ...
   <subclass name="CreditCard"
             discriminator-value="CC">
     <property name="number" column="CC_NUMBER"/>
     <property name="expMonth" column="CC_EXP_MONTH"/>
     <property name="expYear" column="CC_EXP_YEAR"/>
   </subclass>
   <subclass name="BankAccount" discriminator-value="BA">
      ...
 </class>
</hibernate-mapping>
```

Table per class hierarchy

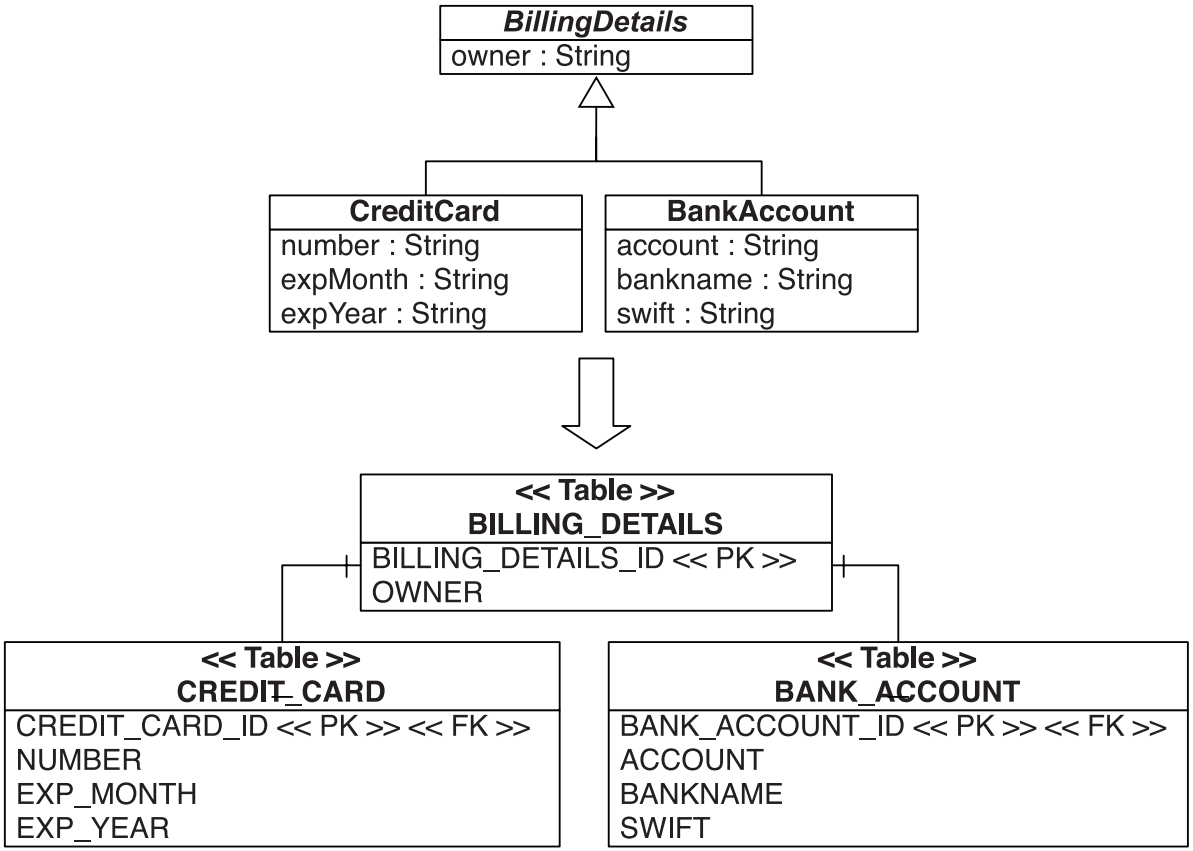
- Polymorphe Abfrage nach `BillingDetails`:

```
select
    BILLING_DETAILS_ID, BILLING_DETAILS_TYPE, OWNER,
    CC_NUMBER, CC_EXP_MONTH, ..., BA_ACCOUNT, BA_BANKNAME, ...
from BILLING_DETAILS
```

- Abfrage nach konkreter Unterklasse `CreditCard`:

```
select BILLING_DETAILS_ID, OWNER, CC_NUMBER, CC_EXP_MONTH, ...
from BILLING_DETAILS
where BILLING_DETAILS_TYPE='CC'
```

Table per subclass



- Jede Klasse wird in einer eigenen Tabelle abgelegt.

Table per subclass

Beispiel (Table per subclass)

```
<hibernate-mapping>
 <class name="BillingDetails" table="BILLING_DETAILS">
   <id name="id"
       column="BILLING_DETAILS_ID"
       type="long">
     <generator class="native"/>
   </id>
   <property name="owner" column="OWNER" type="string"/>
   ...
   <joined-subclass name="CreditCard" table="CREDIT_CARD">
    <key column="CREDIT_CARD_ID"/>
    <property name="number" column="NUMBER"/>
    <property name="expMonth" column="EXP_MONTH"/>
    <property name="expYear" column="EXP_YEAR"/>
   </joined-subclass>
   <joined-subclass name="BankAccount" table="BANK_ACCOUNT">
       ...
 </class>
</hibernate-mapping>
```

Table per subclass

- Polymorphe Abfrage nach `BillingDetails`:

```
select BD.BILLING_DETAILS_ID, BD.OWNER,
       CC.NUMBER, CC.EXP_MONTH, ..., BA.ACCOUNT,
       BA.BANKNAME, ...
    case
        when CC.CREDIT_CARD_ID is not null then 1
        when BA.BANK_ACCOUNT_ID is not null then 2
        when BD.BILLING_DETAILS_ID is not null then 0
    end as CLAZZ_
from BILLING_DETAILS BD
    left join CREDIT_CARD CC
      on BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
    left join BANK_ACCOUNT BA
      on BD.BILLING_DETAILS_ID = BA.BANK_ACCOUNT_ID
```

- Abfrage nach konkreter Unterklasse `CreditCard`:

```
select BD.BILLING_DETAILS_ID, BD.OWNER, CC.NUMBER, ...
from CREDIT_CARD CC
    inner join BILLING_DETAILS BD
      on BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
```

Welche Startegie für das Mappen?

- **Table per concrete class**
  Modellierung benötigt keine polymorphe Abfragen und Assoziationen.
- **Table per class hierarchy**
  Polymorphe Assoziationen werden benötigt. NULL-Werte akzeptabel. Subklassen definieren wenige Attribute.
- **Table per subclass**
  Polymorphe Assoziationen werden benötigt. Klassen unterscheiden sich stark.

Automatic Dirty Object Checking

- **Neue Werte für persistente Felder werden automatisch in die Datenbank geschrieben.**
  Mit SQL-update-Mechanismus für Wahrung der Objektidentität.
- **Unnötige Aktualisierungen der Datenbank werden vermieden.**
  Bei Zuweisung eines neuen Objekts wird zuerst nachgesehen, ob es tatsächlich unterschiedliche Werte enthält, bevor eine Aktualisierung gemacht wird.

Intelligent fetching and caching

Strategien für das Laden

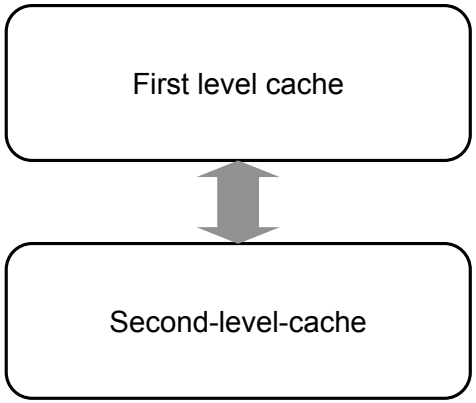
- **Immediate**
  Alles im cache wird sofort mit der Datenbank aktualisiert.
- **Lazy**
  Aktualisiert wird erst beim ersten Zugriff.
- **Eager (OuterJoin)**
  Aktualisiert werden alle Daten, die zu einem Objekt gehören, das aktualisiert wird (Klassen und Navigationstiefe einstellbar).
- **Batch**
  wie eager, aber erst nach erstem Zugriff
  und auch nur bis zu einer einstellbaren *Anzahl von Objekten*

Intelligent fetching and caching

Caching:

Aufgaben:

- Verringerung der Häufigkeit des tatsächlichen Zugriffs auf die Datenbank
- Arbeiten mit identischen Objekten für gleiche Werte



Session

Session factory

Hibernate-Cache-Architektur:

First Level cache:

- obligatorisch
- nicht konfigurierbar

Second Level cache:

- optional
- vielfältig konfigurierbar: Art der Objekte, Vorhaltungszeit, ...

Anfragekonzepte

Was gab es vor Hibernate?

- SQL-Anfragen — JDBC
- SQL-ähnliche Anfragen mit Objekten statt Tabellen (ODMG) — OQL
- Objektfilter — JDOQL (JDO)

Hibernate bietet alle drei Anfrageformen:

- SQL
- HQL
- Criteria

Hibernate Query Language (HQL)

Make SQL be object oriented:

- Classes and properties instead of tables and columns
- Polymorphism
- Associations
- Muchless verbose than SQL

Full support for relational operations:

- Inner/outer/full joins, cartesianproducts
- Projection
- Aggregation (max, avg) and grouping
- Ordering
- Subqueries
- SQL function calls

nach Gavin King: *Object/Relational Mapping with Hibernate*

Hibernate Query Language (HQL)

HQL is a language for talking about "sets of objects":
It unifies *relational* operations with *object models*

Simplest HQL Query:

```
from Student
```

i.e. get all the students:

```
List allStudents =
  session.createQuery("from Student").list();
```

nach Gavin King: *Object/Relational Mapping with Hibernate*

Hibernate Query Language (HQL)

HQL is a language for talking about "sets of objects":
It unifies *relational* operations with *object models*

More realistic example:

```
select item
from AuctionItem item
  join item.bids bid
where item.description like 'hib%'
  and bid.amount > 100
```

i.e.get all the AuctionItems with a Bidworth > 100 and description that begins with *hib*

nach Gavin King: *Object/Relational Mapping with Hibernate*

Hibernate Query Language (HQL)

HQL is a language for talking about "sets of objects":
It unifies *relational* operations with *object models*

Projection:

```
select item.description, bid.amount
from AuctionItem item
  join item.bids bid
where bid.amount > 100
order by bid.amount desc
```

i.e.get the description and amount for all the AuctionItems with a Bidworth > 100

nach Gavin King: *Object/Relational Mapping with Hibernate*

Hibernate Query Language (HQL)

HQL is a language for talking about "sets of objects":
It unifies *relational* operations with *object models*

Sorting functions:

```
select max(bid.amount), count(bid)
from AuctionItem item
  left join item.bids bid
  group by item.type
  order by max(bid.amount)
```

nach Gavin King: *Object/Relational Mapping with Hibernate*

Hibernate Query Language (HQL)

HQL is a language for talking about "sets of objects":
It unifies *relational* operations with *object models*

Runtime fetch strategies (outer join option, eager fetching):

```
from AuctionItem item
  left join fetch item.bids
  join fetch item.successfulBid
where item.id = 12
```

nach Gavin King: *Object/Relational Mapping with Hibernate*

Criteria–Anfragen

HQL query:

```
session.createQuery("fromStudent").list();
```

Criteria query:

```
session.createCriteria(Student.class).list();
```

Criteria Queries

HQL query:

```
session.createQuery
   (“from Professor professor
      join professor.liest vorlesung
      where vorlesung.titel like ‘Grundlagen%’
          and vorlesung.sws > 1”).list();
```

Criteria query:

```
session.createCriteria(Professor.class)
  .setAlias (“liest”, vorlesung)
  add(Expression.like(vorlesung.titel, “Grundlagen”))
  add(Expression.gt (vorlesung.sws, new Integer (1))).list();
```

Criteria Queries with Examples

```
AuctionItem item = new AuctionItem();
item.setDescription("hib");
Bid bid = new Bid();
bid.setAmount(1.0);
List auctionItems =
  session.createCriteria(AuctionItem.class)
    .add( Example.create(item).enableLike(MatchMode.START) )
    .createCriteria("bids")
      .add( Example.create(bid) )
    .list();
```

Äquivalente HQL–Abfrage

```
from AuctionItem item
    join item.bids bid
where item.description like 'hib%'
    and bid.amount > 1.0
```

nach Gavin King: *Object/Relational Mapping with Hibernate*

Zusammenfassung

Hibernate

- Eigenschaften
- Persistenz
- Konfiguration
- Mapping
- Vererbung
- Änderungsmanagement
- Laden und Zwischenspeichern
- HQL–Anfragen
- Criteria–Anfragen