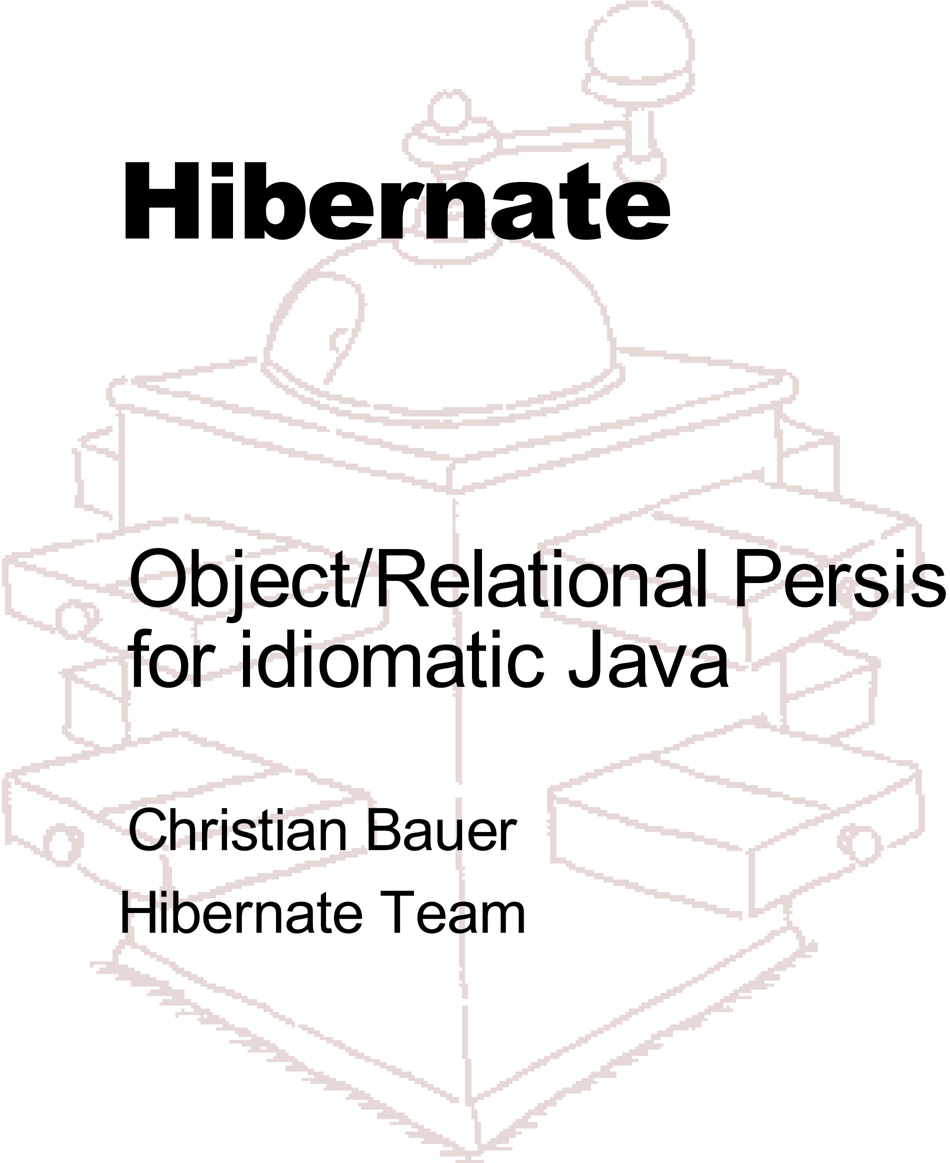
BeJUG

Hibernate



Object/Relational Persistence for idiomatic Java

Christian Bauer

Hibernate Team

Why we need object / relational mapping and why Hibernate is the best solution.

Key Topics

- Why object/relational mapping?
- Solving the mismatch with tools
- Basic Hibernate features
- Hibernate Query Options
- *Detached Objects*

The structural mismatch

- Java types vs. SQL datatypes
  - user-defined types (UDT) are in SQL:1999
  - current products are proprietary
- Type inheritance
  - no common inheritance model
- Entity relationships
- Collection semantics

Behavioral aspects

- Java object identity, equality, primary keys

```
a == b
a.equals(b)
?
```

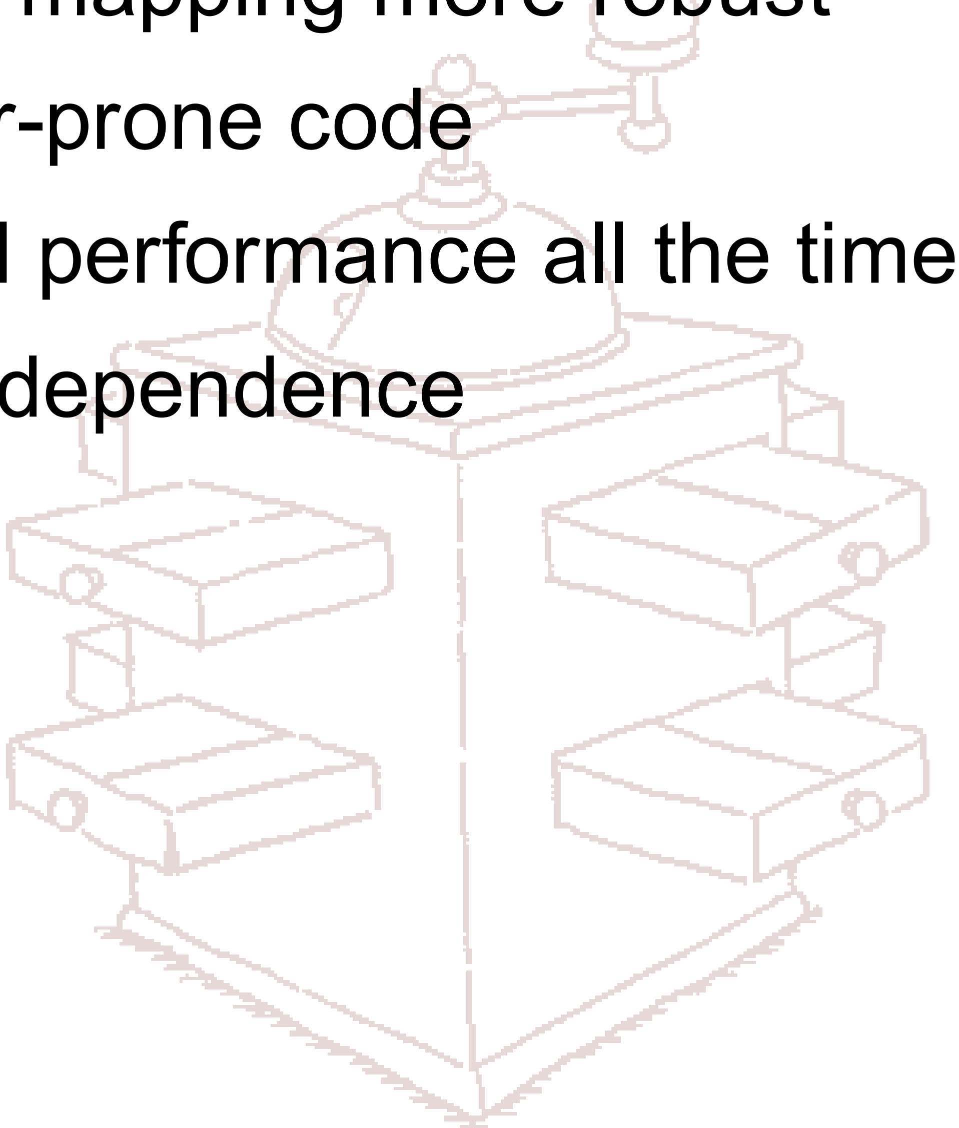
- Polymorphism
- Joining tables vs. navigating associations

“Modern” ORM Solutions

- Transparent Persistence
- Automatic dirty object checking
- Transitive Persistence
- Inheritance mapping strategies
- Smart fetching and caching
- Development tools

Why ORM?

- Structural mapping more robust
- Less error-prone code
- Optimized performance all the time
- Vendor independence



Defining Transparent Persistence

- Any class can be a *persistent* class
- No interfaces have to be implemented
- No persistent superclass has to be extended

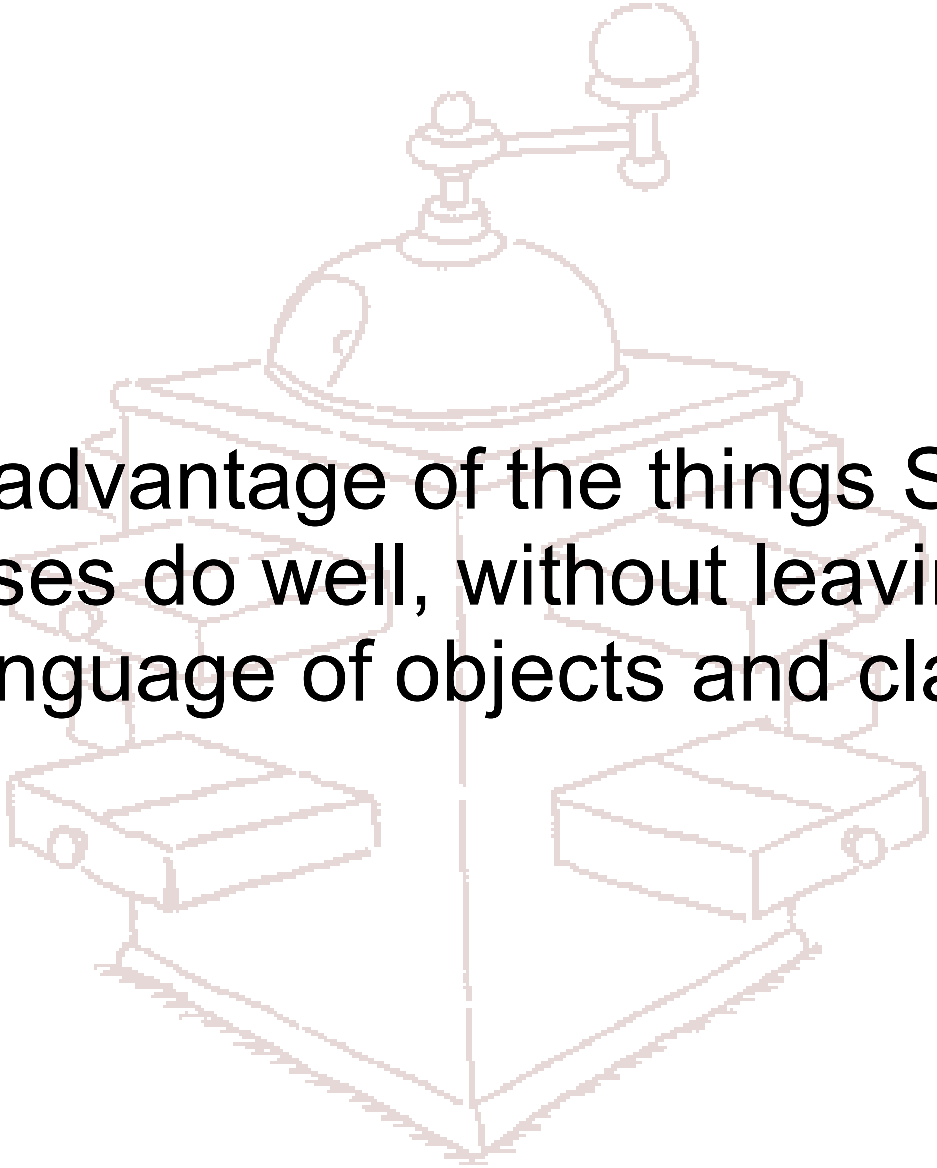
→ Persistent classes can be used outside of the “persistence” context (Unit Tests)

→ Full portability without any dependency

Data integrity is the first rule

- Even so, the relational model is important
- Current implementations are the problem
- Always ensure data integrity using the database
- The data in your SQL database will be around much longer than your application!

The Goal



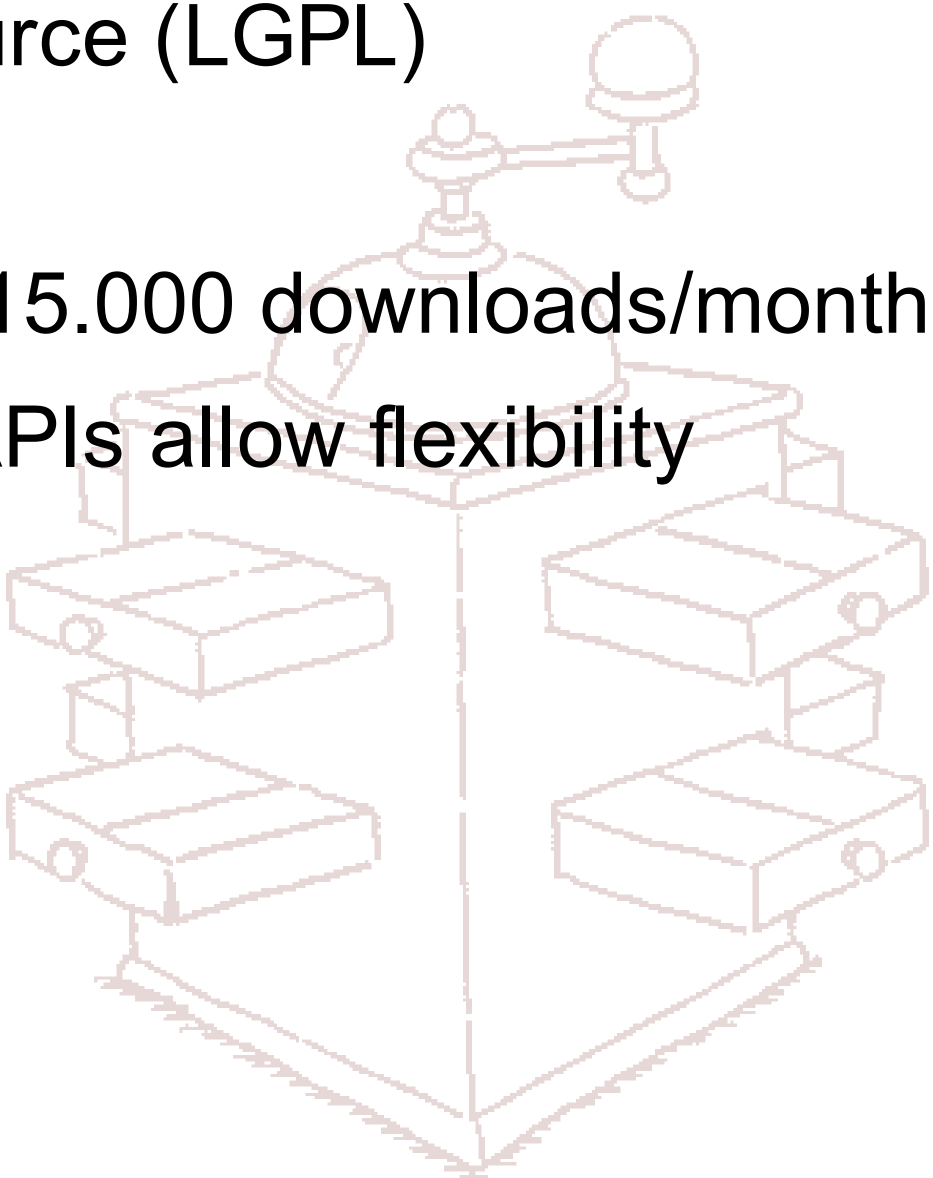
Take advantage of the things SQL databases do well, without leaving the Java language of objects and classes.

The Real Goal

Do less work and have a happy DBA.

Hibernate

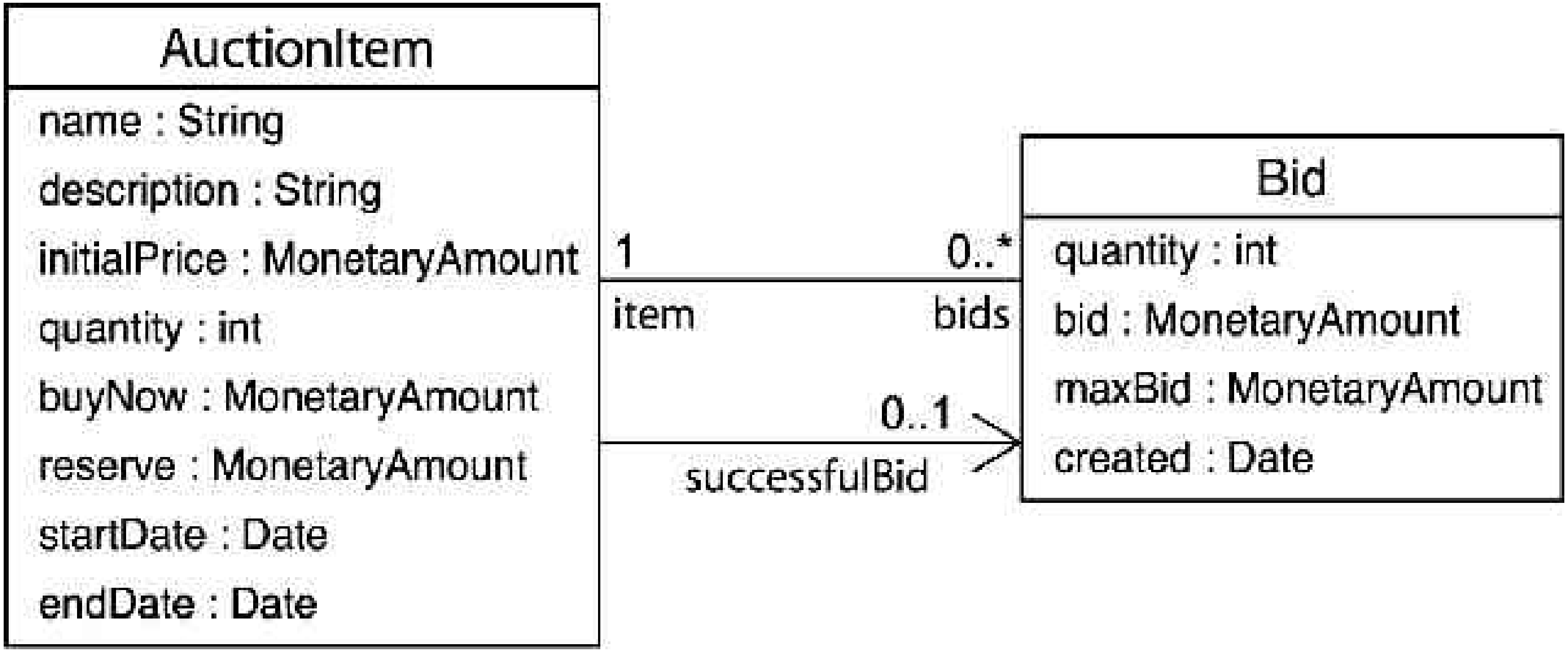
- Open Source (LGPL)
- Mature
- Popular (15.000 downloads/month)
- Custom APIs allow flexibility



Features

- Persistence for POJOs (JavaBeans)
- Flexible and intuitive mapping
- Support for fine-grained object models
- Powerful, high performance queries
- Dual-Layer Caching Architecture (HDLCA)
- Toolset for roundtrip development
- Support for *detached* persistent objects

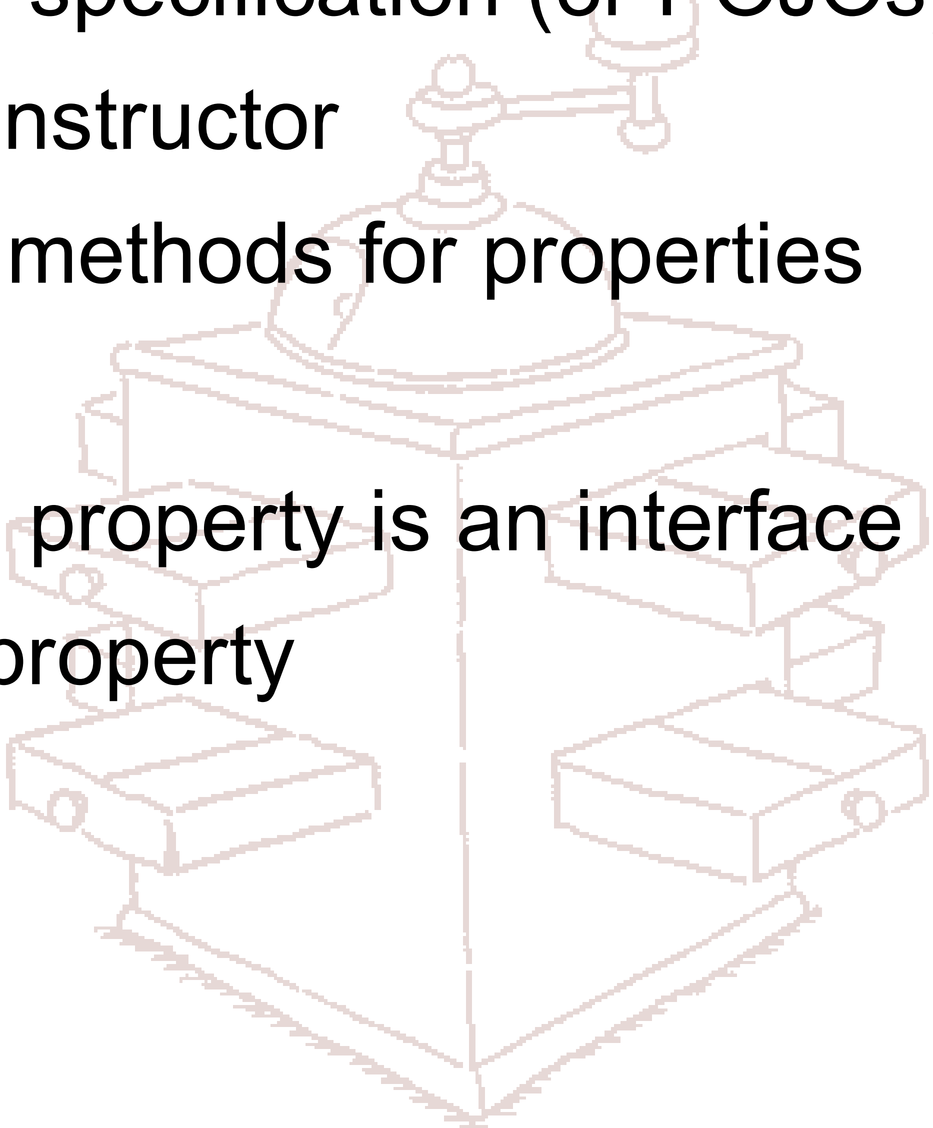
An example object model



Persistent classes

- JavaBean specification (or POJOs)
- No-arg constructor
- Accessor methods for properties

- Collection property is an interface
- Identifier property



XML Mapping Metadata

```
<class name=“AuctionItem” table=“AUCTION_ITEM”>

  <id name=“id” column=“ITEM_ID”>
    <generator class=“native”/>
  </id>

  <property name=“description” column=“DESCR”/>

  <many-to-one name=“successfulBid”
                column=“SUCCESSFUL_BID_ID”/>

  <set name=“bids” cascade=“all” lazy=“true”>
    <key column=“ITEM_ID”/>
    <one-to-many class=“Bid”/>
  </set>

</class>
```

Automatic dirty object checking

Retrieve an `AuctionItem` and change the description:

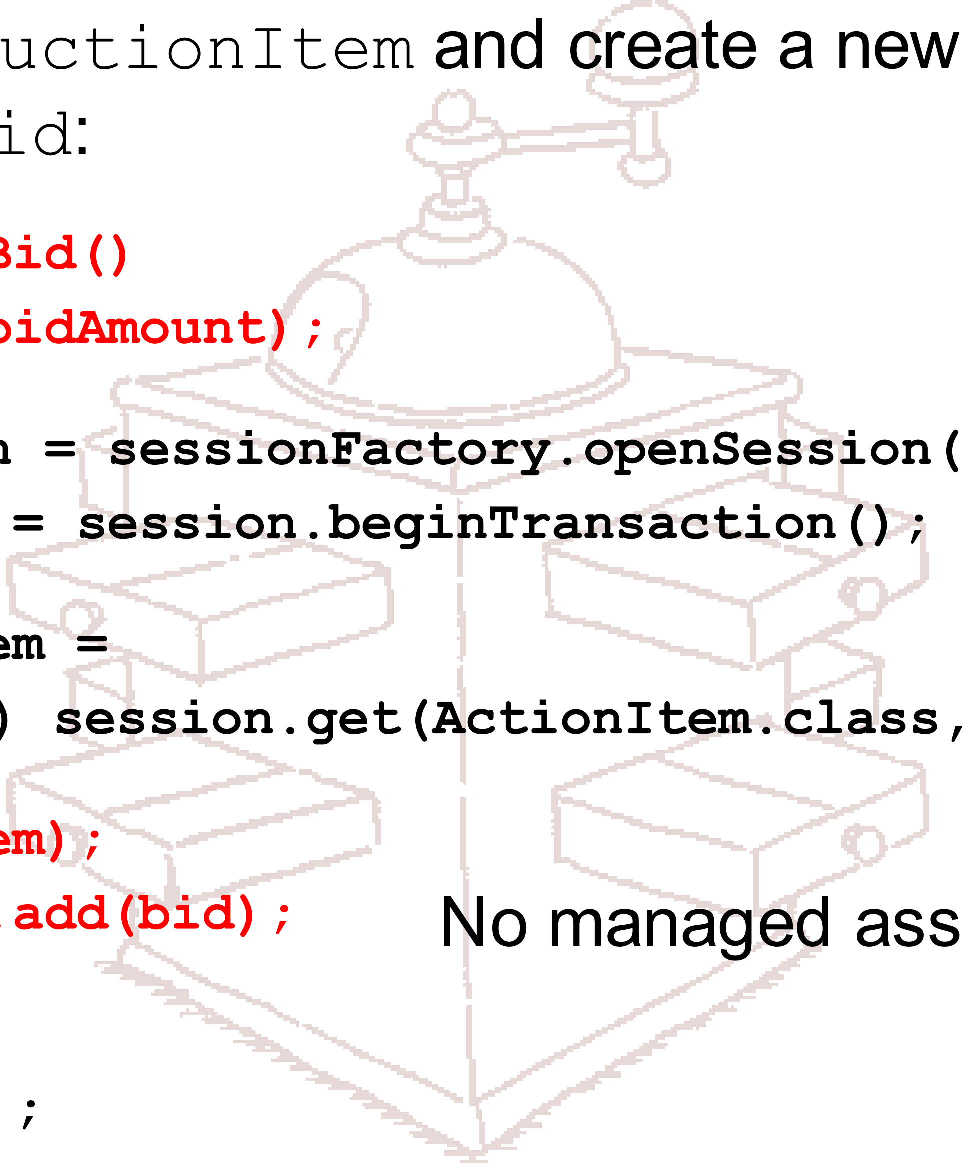
```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

AuctionItem item =
   (AuctionItem) session.get(ActionItem.class, itemId);
item.setDescription(newDescription);

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

Transitive Persistence

Retrieve an `AuctionItem` and create a new persistent `Bid`:



```
Bid bid = new Bid()
bid.setAmount(bidAmount);

Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

AuctionItem item =
   (AuctionItem) session.get(ActionItem.class, itemId);

bid.setItem(item);
item.getBids().add(bid);

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

No managed associations!

Detached objects

Retrieve an `AuctionItem` and change the description:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
AuctionItem item =
   (AuctionItem) session.get(ActionItem.class, itemId);
tx.commit();
session.close();

item.setDescription(newDescription);

Session session2 = sessionFactory.openSession();
Transaction tx = session2.beginTransaction();
session2.update(item);
tx.commit();
session2.close();
```

Hibernate query options

- Hibernate Query Language (HQL)
  - “minimal” object-oriented dialect of ANSI SQL
- Criteria Queries (QBC)
  - extensible framework for query objects
  - includes Query By Example (QBC)
- Native SQL queries
  - direct passthrough with automatic mapping
  - named SQL queries in metadata

Hibernate Query Language

- Make SQL “object-oriented”
  - Classes and properties instead of tables and columns
  - supports Polymorphism
  - automatic association joining
  - *much* less verbose than SQL
- Full support for relational operations
  - inner/outer/full joins, cartesian product
  - projection, ordering, aggregation and grouping
  - subqueries and SQL functions

Simplest HQL query

```
from AuctionItem;
```

i.e. get all the **`AuctionItems`**:

```
List allAuctions =
     session.createQuery("from AuctionItem").list();
```

A more realistic HQL example

```
select item
    from AuctionItem item
    join item.bids as bid
where item.description like “Hibernate%”
    and bid.amount > 100
```

i.e. get all the **`AuctionItem`**s with a **`Bid`** worth more than 100 and an item description that begings with “Hibernate”.

Criteria queries

```
List auctionItems =
 session.createCriteria(AuctionItem.class)
        .setFetchMode(“bids”, FetchMode.EAGER)
        .add( Expression.like(“description”, desc) )
        .createCriteria(“successfulBid”)
          .add( Expression.gt(“amount”, minAmount) )
        .list();
```

Equivalent HQL:

named query parameters

```
from AuctionItem item
  left join fetch item.bids
where item.description like :description
  and item.successfulbid.amount > :minAmount
```

Example queries

```
Bid exampleBid = new Bid();
exampleBid.setAmount(100);

List auctionItems =
   session.createCriteria(AuctionItem.class)
            .add( Example.create(exampleBid) )
            .createCriteria(“bid”)
              .add( Expression(“created”, yesterday)
            .list();
```

Equivalent HQL:

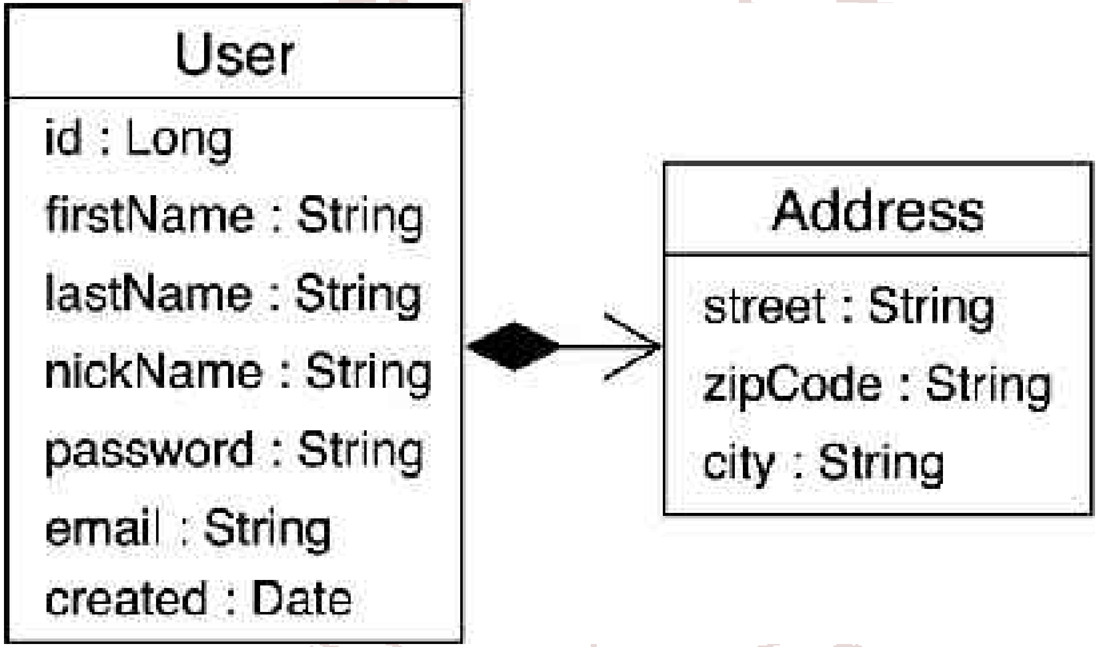
```
  from AuctionItem item
      join item.bids bid
  where bid.amount = 100
      and bid.created = :yesterday
```

Fine-grained persistence

- “More classes than tables”
- Fine-grained object models are good
  - greater code reuse
  - easier to understand
- Hibernate defines
  - Entities (lifecycle and relationships)
  - Values (no identity, “embedded” state)

Composing objects

- `Address` of a `User`
- `Address` depends on `User`



Mapping components

In the mapping metadata of the containing class:

```
<class name=“User” table=“USER”>
  …
  <component name=“address”>
      <property name=“street” column=“STREET”/>
      <property name=“zipCode” column=“ZIPCODE”/>
      <property name=“city” column=“CITY”/>
  </component>

</class>
```

Custom types

- Extend the built-in mapping types
- i.e. `MonetaryAmount` class
- Maps to `AMOUNT` and `CURRENCY` columns

➔ Not limited to “one property = one column”

➔ More flexible than Component

➔ Low-level manipulation/type conversion

Mapping custom types

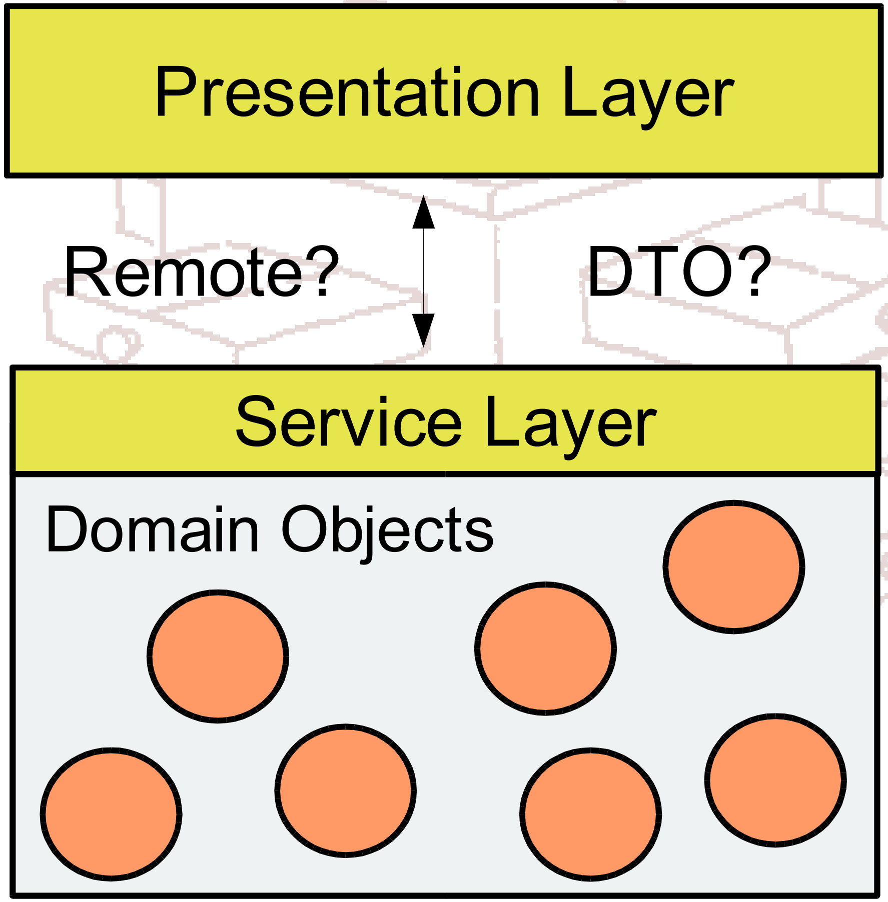
```
<class name=“Bid” table=“BID”>
  …
  <property name=“amount”
           type=“MonetoryAmountUserType”>
      <column name=“AMOUNT”/>
      <column name=“CURRENCY”/>
  </property>

</class>
```

We still have to write the `MonetaryAmountUserType` class!

Layer communication

The presentation layer is decoupled from the service layer and business logic:



DTOs are Evil

- "Useless" extra LoC
- Only state, no behavior
- Results in parallel class hierarchies
- Shotgun changes are bad

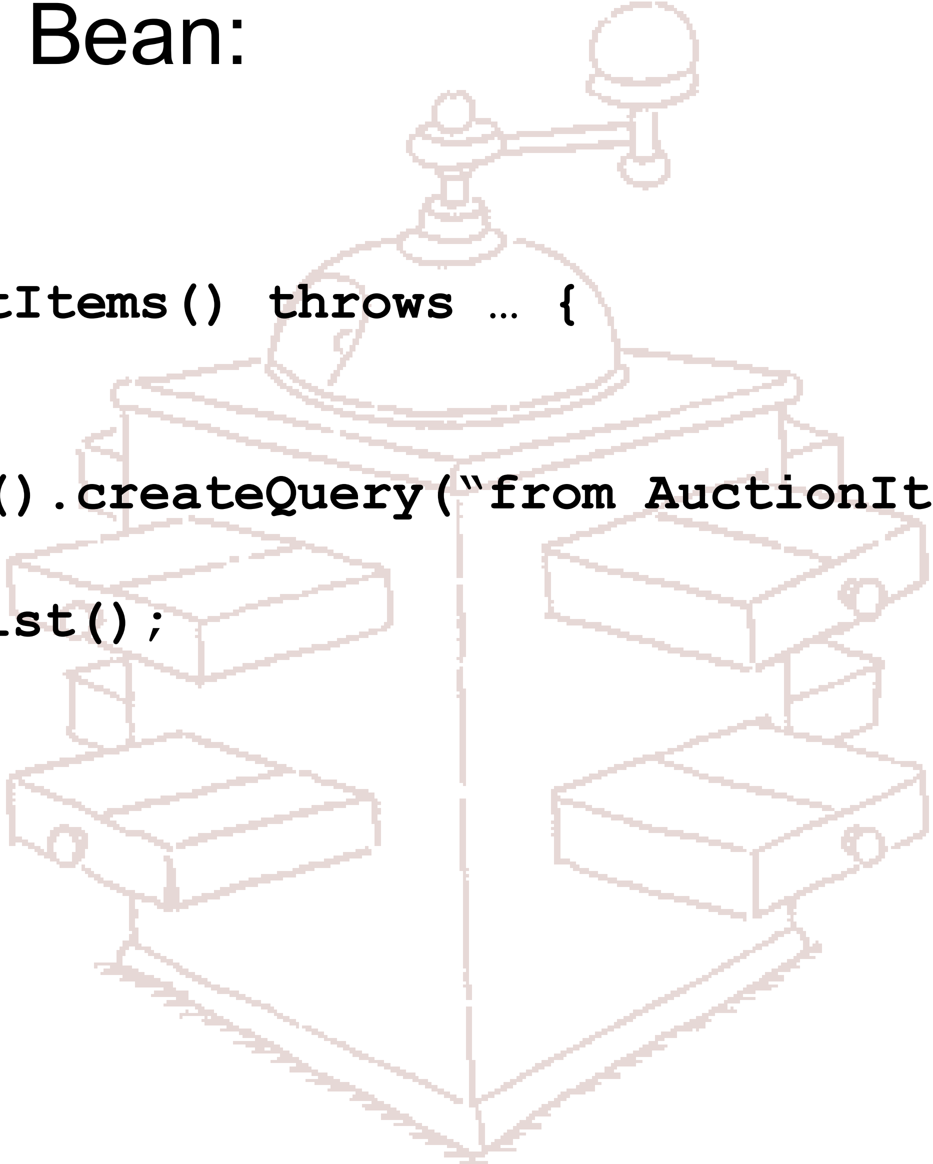
Solution: Detached Object Support

Detached Object Support

- For applications using Servlets and Session Beans
- You don't need DTOs anymore
- You may serialize objects to the web tier, then serialize them back to the EJB tier in the next request
- Hibernate lets you *selectively* reattach a subgraph!

Step 1: Retrieve objects

in a Session Bean:



```
public List getItems() throws … {
 Query q =
  getSession().createQuery(“from AuctionItem”);
  return q.list();
}
```

Step 2: Manipulate objects

in a Servlet, set user information:

```
item.setDescription(newDescription);
```

Step 3: Make changes persistent

back in the Session Bean:

```
public void updateItem(AuctionItem item) throws … {
  getSession().update(item);
}
```

Even with transitive persistence!

```
Session session = sf.openSession();
Transaction tx = session.beginTransaction();
AuctionItem item =
  (AuctionItem) session.get(ActionItem.class, itemId);
tx.commit();
session.close();

Bid bid = new Bid();
bid.setAmount(bidAmount);
bid.setItem(item);
item.getBids().add(bid);

Session session2 = sf.openSession();
Transaction tx = session2.beginTransaction();
session2.update(item);
tx.commit();
session2.close();
```

The Big Problem

- Detached Objects & Transitive Persistence
- How do we distinguish between newly instantiated objects and detached objects that are already persistent?

Solution

- Hibernate uses the “version” property, if there is one
- Hibernate uses the identifier value
  - no identifier value means a new object
  - doesn't work for natural keys, only for Hibernate managed surrogate values!
- Write your own strategy with

```
Interceptor.isUnsaved()
```

Other approaches

- CarrierWave

  http://carrierwave.sourceforge.net

- Service Data Objects (SDO)

CarrierWave

- “Automatic DTO” implementation
- Code generation of DTOs (Images)
- Graph Plans describe closure
- Finders refine the query (by example)
- Graphs of Images tracks changes
- Actions can be invoked on Images
- Transport over RMI, Local, XML?
- Integration with persistence mechanism

CarrierWave on the Client

```
ClientSession clientSession =
    ClientSession.createClientSession( null );

QueryClient queryClient =
    clientSession.getQueryClient();

// A GraphPlan of depth 1
GraphPlan graphPlan = new GraphPlan(1, true);

UserFinderImage finder =
    new UserFinderImage("scott");

UserImage user = (UserImage)
 queryClient.selectImageGraphWith(graphPlan, finder);
```

The **UserImage** implements the automatically generated **User** interface for clients. **UserImage** is the wrapper, tracking changes. The finder is custom written by server-side developers.

CarrierWave on the Server

Markup business classes with interfaces and meta tags for DTO (Image) code generation:

```
/**
 *
 * @image-field address auction.model.Address
 * @image-read-only password
 */
public class User implements ImageableIdentifiable {

  String handle;
  String password;
  Address address;

  public User() {}
  ...
}
```

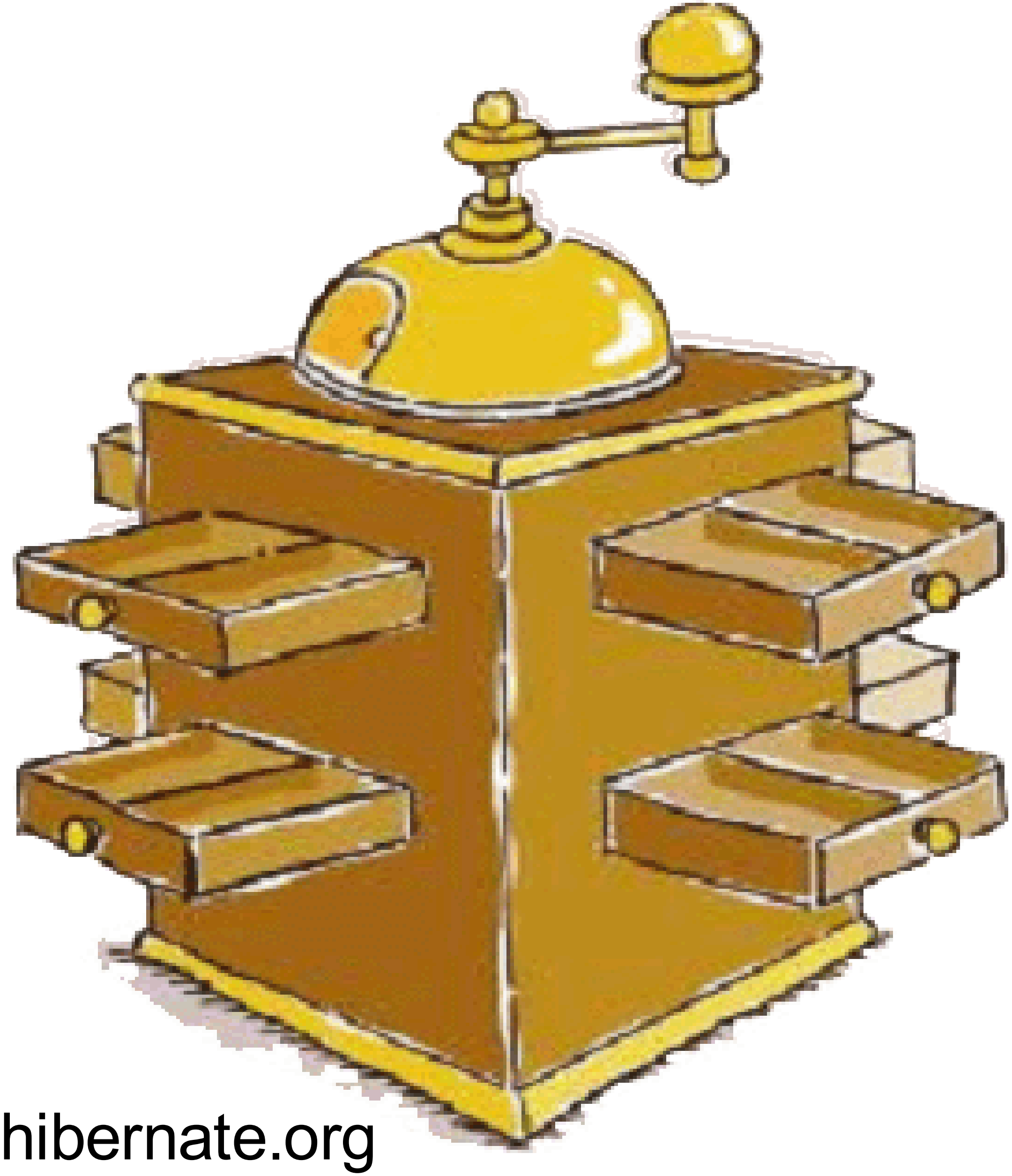
Client-Server communication

- Using the verbs **select**, **modify**, **delete**, **invoke**
- Fault for lazy loading
- Independent of transport mechanism

Is CarrierWave ready?

- Not mature
- Over-designed in some aspects
- Bad choice of class/interface names
- Documentation lacking
- Pro: Fundamentally useful pattern

JavaPolis 2003



http://www.hibernate.org