Adria IUG Udruga Infomix korisnika Adria

Informix

IDS & Hibernate From Design to Implementation

Gary Ben-Israel

IIUG Board of Directors

gary@iiug.org

Agenda

- Object Relational Mapping (ORM)
- Entity Beans
- HQL – Hibernate Query Language
- Named queries
- Using native SQL
- Executing a stored procedure
- The Informix dialect
- Hibernate embedded objects
- Inheritance

Object Relational Mapping (ORM) Solving the Paradigm mismatch

- The problem of granularity
- The problem of subtypes
- The problem of Identity
- Problems related to associations
- Problem of object graph navigation

The problem of granularity

Granularity refers to the relative size of the objects you're working with.

As an example let's look at the address fields in a RDBMS table user. In Java we will have an address class. We can use UDTs but they are poorly supported by SQL and not portable between different databases.

The standard solution will be to map the address class to:

```
create table user(
   name  varchar(50),
   address_street varchar(50),
   adress_city varchar(15),
   adress_state varchar(15),
   adress_zipcode varchar(15) )
```

The problem of subtypes

In Java we implement inheritance using a super and sub class.
In an SQL database we can't declare that credit_card_details
table is a subtype of billing_details by:
create table credit_card_details extends billing_details.
Hence it is not surprising that SQL will not support polymorphisem

The problem of Identity

Java objects define two different notations of sameness:

Object identity (Checked with a==b)

Equality as determined by the implementation of the equals() method.

On the other hand identity of a database row is expressed as the primary key value.

Neither equals() nor == is naturally equivalent to the primary key value.

Problems related to associations

Object-oriented languages represent association using object references and collections of object references.
In the relational world, an association is represented by a foreign key column.
Object references are inherently directional, from one object to another. If you need both directions you must define the association twice. Once in each class.
Java associations can be many to many.
If you have a many to many relationship in a relational database using a link table, this table doesn't appear anywhere in the object model.

Problem of object graph navigation

In Java when you access the billing information of a user you call aUser.getBillingDetails().getAccountNumber()
This is often described as walking the object graph.
Unfortunately this is not an efficient way to retrieve data from an SQL database.
In the database you would want to use a join to minimize the number of SQL queries.
This mismatch in the way we access objects in Java and in a relational database is perhaps the single most common source of **performance problems** in Java applications.

Udruga Infomix korisnika Adria
Adria IUG

Entity Beans

Informix

Entity beans in Java Persistence 1.0 specification are available only as plain old java objects (POJOs). To implement an entity bean you need to define a bean class and decide what field you will use as the identifier (primary key) of that class.

In Java persistence, entity beans are not components like their EJB 2.1 counterparts. Application code works directly with the bean.

So how is an entity queried? How is it stored? Is some magic involved?

NO. Interaction with entity beans is done via a new service called the EntityManager.

No Magic. No byte code manipulation. No special proxies.

Just Plain Java.

Udruga Infomix korisnika Adria
Adria IUG

Informix

Entity bean Declarations
Table definition

Mappings using Hibernate-Annotations

<u>City.java</u>

```
@Entity
@Table(name = "city")
```

Entity bean Declarations Column declarations

```
public class City implements Serializable{
    @Id
    @Column(name = "city_ik")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    @Column(name="city_code", nullable=false)
    private int code;

    @Column(name="city_name", nullable=false, length=30)
    private String name;

    @Column(name=”latitude”)
    private Double latitude;
```

Entity bean Declarations Relationship

```
@OneToMany(fetch = FetchType.LAZY , mappedBy = "city")
private List<Street> streets;

public City() {
}

public String getName() {
        return name;
}
public void setName(String cityName) {
        this.name = cityName;
}
//equals
//hashCode  (Don't use a lazy object in the hashCode)
}
```

Udruga Infomix korisnika Adria
Adria IUG

Entity bean Declarations Column declarations

Informix

Street.java

```
@Entity
@Table(name = "street")
public class Street implements Serializable {

@Id
 @Column(name = "street_ik")
 @GeneratedValue(strategy = GenerationType.AUTO)
private int id;

@Column(name="street_code", nallable =false)
private int code;

@Column(name="street_name", length=20, nallable =false)
private String name;
```

Entity bean Declarations Relationship

```
@ManyToOne(optional = false)
@JoinColumn(name = "city_ik", referencedColumnName = "city_ik")
private City city;

public Street() {
}

public String getName() {
        return name;
}
public void setName(String streetName) {
        this.name = streetName;
}
…
}
```

Simple HQL query

from Streert s where s.code > :code

NOTES:

For each object that Hibernate loads, it needs to do one or more extra SQL queries to load associated objects. This is generally regarded as a bad thing. In this case, loading Street will require fetching of its related City.

I.e. this query will generate (at least) one SQL query for the initial select,

select street_ik, street_code, street_name...from street...

and then a sequence of selects from the associated tables :

select city_ik...from city...
select city_ik...from city...
…

Adria IUG Udruga Infomix korisnika Adria

Informix

HQL using a left join

Improving performance can be done using left join fetch:

The associated objects will be loaded in the initial query

```
from Street s
where s.code > : code
left join fetch s.city
```

Multi-criteria search functionalities

Complex multi-criteria search functionalities are frequently found in web-enabled business applications. Multi-criteria search screens typically have a large number of search criteria, many of which are optional.

The traditional approach to implementing a multi-criteria search query with Hibernate involves building an HQL query on-the-fly, based on the search criteria entered by the user.

Udruga Infomix korisnika Adria
Adria IUG

Informix

Multi-criteria Example

```
Map parameters = new HashMap();
StringBuffer queryBuf = new StringBuffer("from City c ");
boolean firstClause = true;

if (minLatitude!= null) {
        queryBuf.append(firstClause ? " where " : " and ");
        queryBuf.append("c.latitude >= :minLatitude ");
        parameters.put("minLatitude", minLatitude);
        firstClause = false;
}
if (maxLatitude  != null) {
        queryBuf.append(firstClause ? " where " : " and ");
        queryBuf.append("c.latitude <= :maxLatitude ");
        parameters.put("maxLatitude ", maxLatitude);
        firstClause = false;
}
// And so on for all the query criteria...
```

Multi-criteria Example
(Continued)

```
String hqlQuery = queryBuf.toString();
Query query = session.createQuery(hqlQuery);

// Set query parameter values
Iterator iter = parameters.keySet().iterator();
while (iter.hasNext()) {
        String name = (String) iter.next();
        Object value = map.get(name);
        query.setParameter(name,value);
}

// Execute the query
List results = query.list();
```

This approach is cumbersome and error-prone !!

Using Hibernate Criteria

A Criteria object is created using the createCriteria() method in the Hibernate session object :

```
Criteria criteria = session.createCriteria(City.class);
```

Once created, you add Criterion objects (generally obtained from static methods of the Expression class) to build the query. Methods such as setFirstResult(), setMaxResults(), and setCacheable() may be used to customize the query behavior.

Hibernate Criteria example

Finally, to execute the query, the list() (or, if appropriate uniqueResult()) method is invoked :

```
List cities = session.createCriteria(City.class)
        .add(Expression.ge("latitude", minLatitude  );
        .add(Expression.le("latitude", maxLatitude  );
        .addOrder( Order.asc("name") )
        .setFirstResult(0)
        .setMaxResults(10)
        .list();
```

The Hibernate Criteria API supports a rich set of comparison operators.
The standard SQL operators (=, <, <=, >, >=) are supported respectively by the following methods in the Expression class : eq(), lt(), le(), gt(), ge()

Named Queries

To execute a named-query you simply do the following:

```
session.getNamedQuery("findCityByName")
      .setString("name", citySearchName)
      .list();
```

The getNamedQuery() method obtains a Query instance for a named query.
The named-query should be defined in the Entity class and should be named 'findCityByName'.

```
@NamedQueries({
@NamedQuery(name = “findCityByName",
                    query ="from City c where c.name = :name")
})
```

Native SQL

```
session.createSQLQuery("SELECT * FROM STREET").list();
```

These will return a List of Object arrays (Object[]) with scalar values for each column in the STREET table.
Hibernate will use ResultSetMetadata to deduce the actual order and types of the returned scalar values. To avoid the overhead of using ResultSetMetadata or simply to be more explicit in what is returned one Can use addScalar().

Native SQL (continued)


```
session.createSQLQuery("SELECT * FROM STREET ")
.addScalar("ID", Hibernate.INT)
.addScalar("NAME", Hibernate.STRING)
.addScalar("CODE", Hibernate.LONG)
…
```

This will still return Object arrays, but now it will not use ResultSetMetdata
Instead it will explicitly get the ID, NAME and CODE column as
respectively a Int, String and a Long from the underlying resultset.
This also means that only these three columns will be returned, even
Though the query is using * and could return more than the three listed
columns.

Executing a stored procedure

Mapping in the entity:

```
@NamedNativeQuery(name = "X",
                  query = "{? = call XYZ(?, ?)}",
                  resultSetMapping = "XM",
                  hints = { @QueryHint(name = "org.hibernate.callable",
                                       value = "true") })
@SqlResultSetMapping(name = XM",
                     columns = { @ColumnResult(name = "x") })
```

Adria IUG Udruga Infomix korisnika Adria

Informix

Executing a stored procedure

(continued)

Execution (in DAO):

```
Query q = session.createNamedQuery("X");
q.setParameter(1, "whatever");
q.setParameter(2, "whatever");
String x = (String) q.getSingleResult();
```

In NITE we are using Spring to execute stored procedures:

org.springframework.jdbc.object.StoredProcedure

```
private NiteStoredProcedureExecuter executer;

    NiteStoredProcedure storedProcedure= executer.createStoreProcedure(SP);
    addSqlParams(storedProcedure);

    // Return result
    Map result = executer.execute(storedProcedure, getInputParamers(p1, p2,..,Pn));
    returnValue = (Integer) result.values().iterator().next();
```

Udruga Infomix korisnika Adria
Adria IUG

Informix

Lazy Vs. Eager

- By default Hibernate will apply an eager fetch on all relations
- In a large database tables may refer to huge collections of other tables.
- Unless lazy fetching is explicitly requested all referred tables will be fetched
- **NOTE**: When applying one to one relation hibernate will ignore the lazy request and perform an eager fetch.

Adria IUG Udruga Infomix korisnika Adria

Informix

The Informix dialect

public class InformixDialect extends org.hibernate.dialect.Dialect

It contains the specific SQL definitions for Informix

Unfortunately the Informix dialect supplied by Hibernate is not sufficient

To get an up to date copy you will need to create your own local dialect

Local Informix Dialect

public class MyInformixDialect extends InformixDialect

```
registerColumnType(Types.DOUBLE, "float");

registerFunction("second", new SQLFunctionTemplate(Hibernate.INTEGER,
    "second(?1)"));

public String getLimitString(String querySelect, int offset, int limit) {
    StringBuffer limitString = new StringBuffer(20);
    limitString.append((offset > 0) ? " skip " + offset : "");
    limitString.append((limit > 0) ? " first " + limit : "");
    return new StringBuffer(querySelect.length() + 8).append(querySelect).insert(
        querySelect.toLowerCase().indexOf("select") + 6,
        limitString.toString()).toString();
}
```

Adria IUG
Udruga Infomix korisnika Adria

Informix

Hibernate Embedded objects:

By Using @Embeddable annotation

Example:

```
@Entity
public class Person implements Serializable {

    // Persistent component using defaults
     private Address homeAddress;
…
}

@Embeddable
public class Address implements Serializable {
    private String city;
    private String nationality;
}
```

Using Embeddable Objects

Note that Embeddable is currently
not supported in the EJB3 spec, however,
Hibernate Annotations supports it.

You can use them but when building an HQL query
Hibernate will regard the embedded alias as a table.
For instance f you have an embeddable object address within the
customer class and you are using:
Customer.Address.getZipCode()
Hibernate will regard Address as a table and try to perform a join.
To avoid it you will need to define the aliases manually.

Inheritance

A simple strategy for mapping classes to database tables might be “one table for every class.” This approach sounds simple, and it works well until you encounter inheritance.

Inheritance is the most visible feature of the structural mismatch between the object-oriented and relational worlds.

Object-oriented systems model both **“is a”** and **“has a”** relationships.
SQL-based models provide only **“has a”** relationships between entities.

Inheritance Types

There are three different approaches to representing an Inheritance hierarchy.

- **Table per concrete class** - Discard polymorphism and inheritance relationships completely from the relational model
- **Table per class hierarchy** - Enable polymorphism by de-normalizing the relational model and using a type discriminator column to hold type information
- **Table per subclass** - Represent “is a” (inheritance) relationships as “has a” (foreign key) relationships

Table per concrete class

We could use exactly one table for each (non-abstract) class.

The main problem with this approach is that it doesn't support polymorphic associations very well.

In the database, associations are usually represented as foreign key relationships.

Hence a query against the superclass must be executed as several SQL SELECTs, one for each concrete subclass.

Note that Hibernate currently doesn't support the use of unions and will always use multiple SQL queries.

A further conceptual problem with this mapping strategy is that several Different columns of different tables share the same semantics. This makes schema evolution more complex.

Table per class hierarchy

Alternatively, an entire class hierarchy could be mapped to a single table. This table would include columns for all properties of all classes in the hierarchy. The concrete subclass represented by a particular row is identified by the value of a *type discriminator column.*
This mapping strategy is a winner in terms of both performance and simplicity.

There is one major problem: Columns for properties declared by subclasses must be declared to be nullable.

Table per subclass

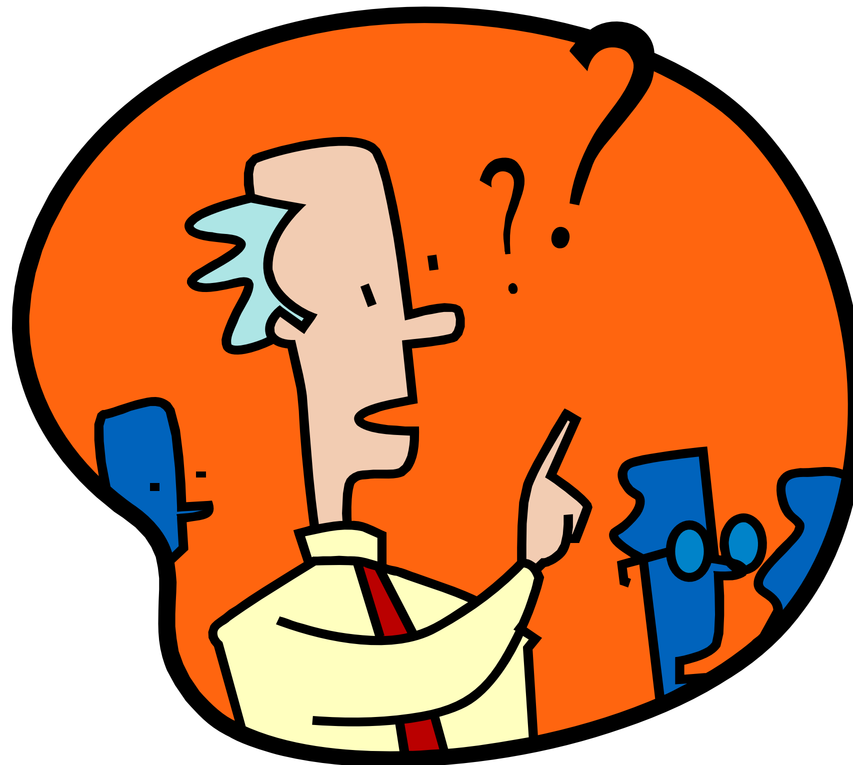
The third option is to represent inheritance relationships as relational foreign Key associations. Every subclass that declares persistent properties its own table.

Unlike the strategy that uses a table per concrete class, the table here contains columns only for each non-inherited property (each property declared by the subclass itself) along with a primary key that is also a foreign key of the superclass table.

The primary advantage of this strategy is that the relational model is completely normalized. Schema evolution and integrity constraint definition are straightforward.

A polymorphic association to a particular subclass may be represented as a foreign key pointing to the table of that subclass.

Questions



Udruga Infomix korisnika Adria

Adria IUG

Informix

IDS & Hibernate From Design to Implementation

Gary Ben-Israel

IIUG Board of Directors

gary@iiug.org