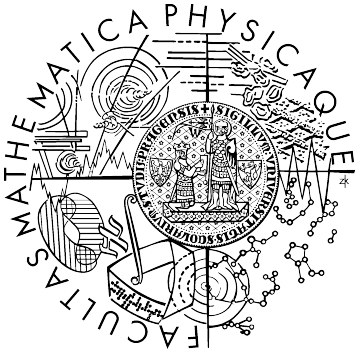
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Martin Hlavatý

Porovnání použitelnosti Java O/R frameworku

Feasibility analysis of Java O/R framework

Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Jan Lánský

Studijní program: Informatika, Softwarové systémy

2008

I would like to thank to my diploma thesis supervisor Mgr. Jan Lánský for his help and for many valuable advices he gave me, while I was writing this thesis.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 5.8.2008 Martin Hlavatý

Contents

Název práce: Analýza použitelnosti Java O/R frameworku
Autor: Martin Hlavatý
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: Mgr. Jan Lánský
e-mail vedoucího: Jan.Lansky@mff.cuni.cz

Abstrakt: Tato diplomová práce analyzuje použitelnost jednoho z nejpopulárnějších nástrojů pro objektově-relační mapování (Hibernate). Zkoumá, jakým způsobem ovlivňuje použití tohoto nástroje architekturu a výkon aplikace a ukazuje, jak lze Hibernate využít k implementaci některých typických požadavků kladených na "enterprise" systémy (například zaznamenávání historie změn). Závěry jsou demonstrováný na strukturálně složitém doménovém modelu, který byl vytvořen pro účely této práce, ale zároveň je součástí reálné aplikace.
Teoretická část se zabývá dotazovacím jazykem HQL a jeho vyjadřovací silou. Obsahuje formální důkaz o převoditelnosti speciální podmnožiny relační algebry (relační algebra bez sjednocení, průniku a rozdílu) na HQL.

Klíčová slova: ORM, Hibernate, HQL, Persistence

Title: Feasibility analysis of JAVA O/R framework
Author: Martin Hlavatý
Department: Department of Software Engineering
Supervisor: Mgr. Jan Lánský
Supervisor's e-mail address: Jan.Lansky@mff.cuni.cz

Abstract: The aim of this thesis is to analyze the usability of one of the most popular O/R mapping frameworks (Hibernate). It examines, whether Hibernate somehow influences an architecture or a performance of the system, which uses Hibernate for data persistence. This thesis also shows, how Hibernate can be used to implement some typical requirements for the enterprise systems (for example audit logging). Findings are demonstrated on the complex domain model, which was created for the purpose of this thesis, but it is also a part of a real-world application.
Theoretical part of this thesis examines the power of HQL. It contains the formal proof of translatability of a subset of relational algebra (relational algebra without union, intersection and difference) into HQL.

Keywords: ORM, Hibernate, HQL, Persistence

1. Introduction

1.1. Preface

When object oriented programming conquered the software world, everybody thought that the era of relational databases is over. But later on it proved that Object databases were more a hype than the „silver bullet“. Relational databases are mature technology, they are well-formalized and used in wide spectrum of applications. But if we want to use them together with the object oriented world, we need to solve several problems. The basic entity in relational theory is a relation, while in OOP it is an class (and its instance – object). Relationships between two relations are modeled using foreign keys, while relationships between classes (objects) can by modeled using pointers. This difference in basics concepts is called „Impedance mismatch“ and will be discussed in detail in section 1.5.

One way, how to solve „Impedance mismatch“ problem, is using so called object-relation mapping software. This tools help you to overcome the gap between relational and object oriented data models. There are plenty of such tools and it is not possible to speak about all of them in this thesis, so we will focus on the most widely used one – the Hibernate (or Nhibernate, which is its .NET implementation). The reasons for choosing Hibernate will be discussed in section 1.8.2.

But this topic is still too wide, to be handled by reasonably long thesis, and there are too many books, articles, bachelor and master thesis describing the more or less basic usage of Hibernate ([1], [2], [3], [4], [5]). That is why we will focus on the research into „Economy of Hibernate usage in enterprise application development“. This thesis describes some typical problems, which an enterprise software architects have to deal with, and tries to find their solution using Hibernate.

1.2. Structure of the Thesis and How To Read

This thesis consists of two relatively independent parts – theoretical and practical (+ introduction and conclusion).

First chapter (the one you are reading right now) provides an introduction to problematic of an object-relational mapping and describes some well known ORM frameworks.

Theoretical part of this thesis deals with expressive power of HQL and contains a proof of theorem saying that „In HQL can be expressed each query, which can be expressed in relational algebra without union, intersection and difference“. You can find it in chapter 2.

Practical part is divided into two chapters. Chapter 3 examines, whether Hibernate somehow limits the design of a domain model and how Hibernate affects the performance of a system, which uses it for data persistence.

Chapter 4 contains description of a typical problems, which needs to be solved by an enterprise applications and discuss, whether it is possible and suitable to use Hibernate to solve them.

The last chapter summarizes the main ideas discussed in this thesis..

1.2.1. How To Read

If you are familiar with Hibernate skip sections 1.5 to 1.8 and section 2.2.

Developers and architects should read chapter 4. People interested in mapping of relational algebra into HQL should read chapter 2.

If you want to decide, whether this thesis contains information, which you are searching for, read the last chapter (5), because it summarize all main ideas discussed in this thesis.

1.3. Evolution of the Topic

The original task from the submitter of this thesis (Profinit company) was to take some huge legacy system and try to rewrite it using Hibernate. After more than four month of intensive effort (more than 500 hours spent on analysis and prototyping) it was clear, that reimplementation of such a big system is not possible. Therefore the topic slightly changed – the reference system is used as an source of typical problems, which needs to be solved in enterprise systems.

While trying to reimplement system mentioned in previous paragraph, author of this thesis discovered some limitations of a query language used by Hibernate and therefore decided to examine them more in depth. Topic of this thesis was therefore expanded to cover formal proof of translatability of relational algebra into HQL.

1.4. Goals

1. Figure out, whether relational algebra can be translated into HQL. If yes, provide a formal proof. If not, define the greatest subset of relational algebra, which can be translated into HQL (and provide a formal proof).
2. Decide whether Hibernate can be used in systems with complex domain model. If yes, implement a complex domain model using Hibernate.
3. Analyze performance of a Hibernate-based application and try to find ways, how to increase it.
4. Decide whether it is possible to use Hibernate to solve some typical enterprise application problems.

1.5. Impedance Mismatch

Impedance mismatch is a set of logical and technical problems which are often encountered when a program written in an object-oriented programming language uses relational database management system.

As was mentioned in section 1.1, the basic entity in the relational world is a relation (database table) and its instance (row in a table). In OOP it is a class and its instance - an object. Natural idea (and basically the only possibility) is to try to map a relation to a class and its instance to an object. But this concept has several problems (caused by differences in basic paradigms):

Declarative vs. Imperative Approach

The most common object oriented languages (such as Java or C++) are imperative – programmer must explicitly specify, how to achieve his goal. But SQL (or relational calculus) is a declarative language – programmer only needs to specify, what he wants to achieve.

On the other side, each program written in declarative language have to be translated to some imperative language to be executed (all commonly used low-level languages are imperative). That is the reason, why this problem is not so big as it seems.

Encapsulation

Attributes in class can have different visibility, which determines, who can use them. Class usually contains also some behavior provided by its methods. There is no similar concept in the relational theory. All attributes of a relation are accessible for anybody. A relation also don't contain any behavior.
Modern RDBMS supports so called triggers – pieces of code associated with some table. They are executed, when some predefined condition is fulfilled (new row is inserted into table …) and provide some behavior. Triggers roughly fit to methods, but they are not contained in the original relational theory.

Identity

Two objects, which are in the same state (have same values of all attributes) in the same time do not have to be identical (in the most common languages is the identity defined in terms of memory addresses). Two instances of the same relation are identical, if they have same values of attributes contained in some candidate key (primary key).

Solution to this problem is to introduce an alternative definition of identity (e.g. `equals` in Java)

Inheritance and Polymorphism.

Although inheritance can be added into relational theory, there is no way how to introduce polymorphism.

Normalization

Classes are usually designed without respect to the theory of normal forms. Therefore they might contain a lot of redundant information. In the relational world, it is a good practice to obey at least third normal form (but sometimes schema might be denormalized to increase performance).

1.6. Accessing Database (in Java)

Before we start to describe various ORM tools, it is a good idea to look at the more traditional way of accessing the database – JDBC. Understanding of this concept is essential, because ORM tools use JDBC to access database.

JDBC is an abbreviation of "Java Database Connectivity". Roughly said, JDBC is an API for manipulation with data stored in a relational database. Each RDBMS has its own implementation, which is usually distributed as a one jar[1] file.

First step, when trying to access database, is to establish a connection. This connection is represented by the instance of `Connection` class and it is a valuable resource. Because establishing connection takes a lot of time (compared to other operations), majority of Java database applications use so called "connection pools"[2] to obtain connections. This service establishes predefined number of connections at

1 JAR = Java archive. Zipped Java classes and other resources together with additional metadata (manifest file). It is the most common way of distribution of Java applications and libraries.

2 More details about connection pooling you can find in online JDBC tutorial at address http://java.sun.com/developer/onlineTraining/Programming/JDCBook/conpool.html

the beginning and lends them to the application on demand. When application no more needs the connection, it simply returns it back to connection pool.

After connection has been established (or obtained from connection pool service) application can execute all CRUD operations using for example methods `executeUpdate`, `createStatement` and `executeQuery`.

Lets look at the example of JDBC usage. Imagine, that you have the following schema: `customer(id, first_name, last_name, telephone)` and you want to retrieve all customers:

```
List result = new ArrayList();
try {
  Statement stmt = getConnection().createStatement();
  ResultSet rs = stmt.executeQuery(
    "SELECT id, first_name, last_name, telephone "
    + "FROM customer");

  // Fetch each row from the result set
  while (rs.next()) {
    Customer customer = new Customer();
    customer.setId(rs.getString("id"));
    customer.setFirstName(rs.getString("first_name"));
    customer.setLastName(rs.getString("last_name"));
    customer.setTelephone(rs.getString("telephone"));

    result.add(customer);
  }
} catch (SQLException e) { // handle error }
```

If you look at the example above, you may notice that there is a lot of boilerplate code. Whenever you want to retrieve some data, you have to handle exceptions and provide mapping between your domain objects and query results. Of course that experienced developer would encapsulate the mapping code into methods and use them wherever possible – this is the first step towards usage of ORM tool.

This section was just a very brief introduction to the JDBC. This area is very complex and is outside the scope of this thesis. For more details about JDBC and its usage, please see [6].

1.7. ORM Frameworks (in Java)

There are many ORM tools in Java. We can mention TopLink, Hibernate, JDO, JPOX, Kodo, iBATIS and Cayenne[3].

Some ORM tools supports only basic mapping and some provide such advanced features as the inheritance mapping, polymorphic relations support or their own query language.

As we have mentioned above, there are many ORM tools and therefore it is not possible to speak about all of them in this chapter. We will choose one representative

3 This list includes only the most popular and best known standards, frameworks and tools.

from each group of ORM tools – simple (Spring JDBCTemplate) and complex (JPA).

1.7.1. Spring JDBCTemplate

`JDBCTemplate` is a part of the Spring framework[4] and it is the simplest ORM tool possible. In fact, it is not an ORM tool at all, because all mappings have to be specified in JDBC manner by developer. But this feature will help us to explain, what typical ORM tool is supposed to do.

But if `JDBCTemplate` doesn't help you with creating the mapping between domain objects and database data, what is it good for? The answer is simple – when developing applications with pure JDBC there is a lot of boilerplate code (like exception handling, scrolling trough results …). `JDBCTemplate` provides template methods[5], which contain this boilerplate code and allow developers to focus only on the core logic.

Lets look at the example of `JDBCTemplate` usage. Imagine, that you have the same schema as in section 1.6: `customer(id, first_name, last_name, telephone)` and you want to retrieve all customers:

```
Collection customers = getJdbcTemplate().query(
  "SELECT id, first_name, last_name, telephone "
    + "FROM customer",
  new RowMapper() {
    public Object mapRow(ResultSet rs, int rowNum)
      throws SQLException {
        Customer customer = new Customer();
        customer.setId(rs.getString("id"));
        customer.setFirstName(rs.getString("first_name"));
        customer.setLastName(rs.getString("last_name"));
        customer.setTelephone(rs.getString("telephone"));

        return customer;
      }
});
```

Lets look, what actually happens in the code above. First, we have obtained a `JDBCTemplate` instance and used it to execute a query. Method `query` takes two parameters – the most interesting one is the second, which contains code for mapping query result to a domain object. Because Java doesn't support pointers to functions, the mapping code has to be passed to the `query` method as an anonymous inner class. Please note, that there is no exception handling – all checked exceptions are converted to unchecked, which can be handled in one place. This helps to fulfill the idea of separation of concerns.

Because mapping between `customer` table and `Customer` domain object will be probably needed more than once, it is not a good idea to use anonymous inner class. Better solution is to implement `RowMapper` interface in some public or package protected class, which can be then reused in multiple queries. This concept

4 http://www.springframework.org

5 Template method is a desing pattern described for example in [7] or at address http://en.wikipedia.org/wiki/Design_pattern_(computer_science)

is very similar to the one, which is used in more advanced ORM tools (the only difference is that here you have to write the mapping code by hand).

Unfortunately, further description of `JDBCTemplate's` functionality goes beyond the scope of this thesis. More information you can find in [8] or [9].

1.7.2. JPA

JPA is an acronym for "Java Persistence API". It is a standard[6] created by Sun Microsystems company. It is a replacement of the older standard JDO and therefore Sun Microsystems recommends to use JPA for all new projects. JPA is based on Hibernate, TopLink and several other technologies for data persistence.

JPA uses annotations to describe mapping metadata (XML is also supported, but it is used very rarely). Configuration is typically held in XML, although it is possible to create it programatically (which makes integration with frameworks such as Spring much easier).

JPA is very similar to Hibernate, which will be described in the next section, therefore in this section we will focus only on the differences.

A JPA-based application can use more JPA implementations at the same time – they can be configured via so called persistence units (defined in configuration file `persistence.xml`). This means that Hibernate can meet TopLink in one application and they will be able to cooperate (share the same database). On the other hand, this is not a common scenario. Majority of JPA-based applications use only one persistence unit and only one JPA implementation.

JPA has its own query language – JPQL, which is based on HQL. Other query languages, such as Criteria API, are not supported. This will change in version 2.0, which should contain support for pure Java query language (similar to Criteria API).

Lets look at the example of JPA usage:

```
// Create EntityManagerFactory for persistent unit "unit1"
emf = Persistence.createEntityManagerFactory("unit1");

// Create new EntityManager
em = emf.createEntityManager();

em.getTransaction().begin();   // Begin transaction

// Business logic
Query query = em.createQuery("SELECT c FROM Customer c
                              WHERE c.name = :name");
query.setParameter("name", "Charles Clarke");
Customer c = (Customer) query.getSingleResult();

em.getTransaction().commit();  // Commit the transaction

// Close this EntityManager
em.close();
```

6 JPA 1.0 is a part of a EJB 3.0 standard defined in JSR 220 (http://jcp.org/en/jsr/detail?id=220)

The code above loads customer named “Charles Clarke” assuming that there is only one person of that name in the database (otherwise an exception would be thrown). Query is executed within a transaction, but it is not necessary in this case (we do not save anything and neither do we access any lazily fetched property). But there are not many situations, where explicite transaction handling is necessary. Therefore declarative transactions are used more frequently.

1.8. Introduction to Hibernate

1.8.1. What Is Hibernate

Hibernate is the most popular open source Java persistence framework. It fully implements JPA 1.0 standard (described in section 1.7.2). Hibernate is currently being developed in Red Hat company[7] by Gavin King and his team.

It is a mature technology (in comparison to other JPA implementations) with a big community around it. Many good books have been written about Hibernate (for example [11], [12], [13] and [14]), but probably the best is [2][8], because it was written by Gavin King, the author of Hibernate. But it is not necessary to buy any book, because Hibernate has a very good reference documentation ([1]).

Hibernate provides an abstraction layer between relational database and application, which uses it. It can completely remove the need to write any SQL statement and highly reduces the amount of code needed for data persistence.

On the other hand, because Hibernate is a complex technology, it is sometimes difficult to understand, why it behaves the way it behaves (it is a “black box”). Also the learning curve is quite long (compared to the simpler ORM tools such as iBatis).

1.8.2. Why Hibernate

As we have explained at the beginning (in the section 1.1), it is not possible to focus on more than one ORM tool in this thesis. Thus we had to choose one representative from the very large group of ORM tools. From the name of this section is clear, that Hibernate was chosen, but it is necessary to explain why.

Reasons for choosing Hibernate:

- It was preferred by the submitter of this thesis (Profinit company).
- It is free of charge.
- It is published under the open source license (LGPL). Access to the source codes makes it easier to learn and understand it.
- It works with POJOs. There is no need to extend any particular class or implement any particular interface, unless you want to customize the object-relational mapping process.
- It is a mature technology. Thus it is highly probable that it doesn't contain any critical error.
- It is very popular and commonly used nowadays.

7 Hibernate used to be a product of the JBoss company. But JBoss has been recently acquired by Red Hat.

8 “Java Persistence with Hibernate“ is the newer version of „Hibernate in Action“ ([10]). All code examples are in two versions – one uses XML for specifying mapping metadata and the other annotations. This is the reason, why many people preffer [2] over [10]. The same publisher has published “Hibernate Quickly” ([15]), which is one of the worst books about Hibernate (it can help you to start using Hibernate, but you won't know how Hibernate works).

The last reason is the most important one. On the other hand, it is difficult to make any estimation about the popularity of Hibernate compared to other ORM tools. There are no global statistics (companies usually do not publish such information), but we can use the service *www.indeed.com* to compare the number of a job opportunities for developers with knowledge of Hibernate, TopLink, iBatis or pure JDBC as shown on the Diagram 1. Idea behind this comparison is simple – when some company uses some particular technology a lot, it needs many developers with knowledge of this technology. This will affect the number of a job opportunities for such developers (and this number is monitored by *www.indeed.com*). Thus the popularity of a ORM tool is positively correlated with the number of a job opportunities for developers knowing this technology.

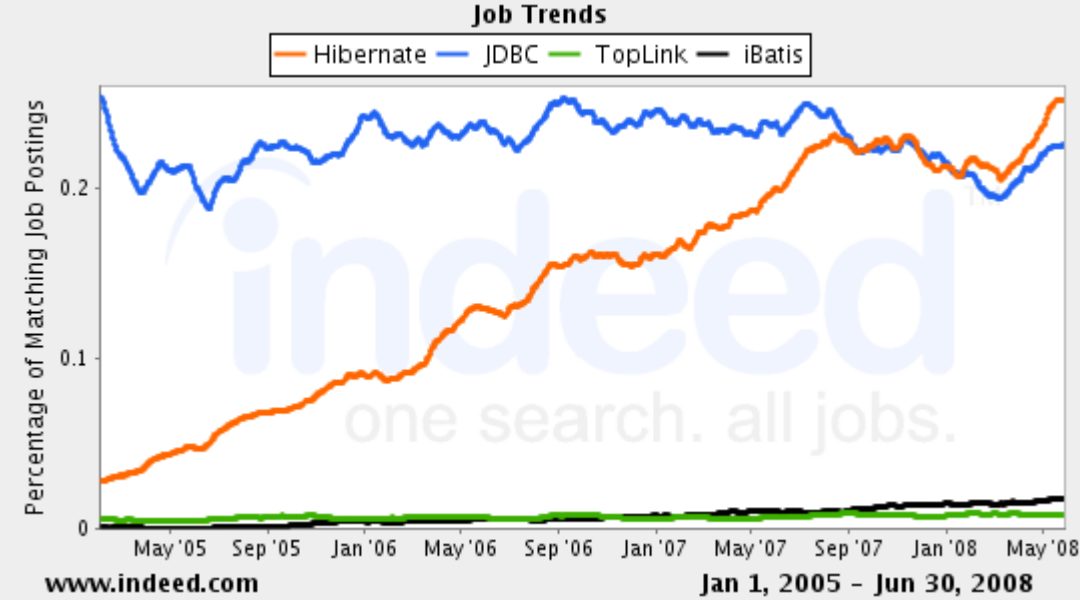


Diagram 1: Number of job opportunities for developers with knowledge of Hibernate or TopLink or iBatis or JDBC (source: www.indeed.com)

As you can see on the Diagram 1, the popularity of Hibernate has growing trend. A few months ago it even left behind the JDBC. Knowledge of other ORM tools is demanded much less often.

After we have explained the reasons for choosing Hibernate, lets shortly describe, how Hibernate can be used in a project.

1.8.3. Basic Usage

When you want to use Hibernate in your project, you have to do following things:

- Put required libraries into classpath.
- Configure it using XML file, property file or Java (these ways can be combined). Configuration tells Hibernate how to connect to the database, which entity classes should be used, whether to use secondary cache ...
- Specify the mapping between domain model and database schema – this can be done using annotations or XML (both ways can be used at the same time).

Lets start with the configuration. The most common way of configuring Hibernate is via XML file. For newly started projects it is a good idea to use Hibernate in a JPA way, because it is a standard. In the section about JPA, we have intentionally omitted the example of configuration. Here it is:

```
<persistence>
 <persistence-unit name="unit1"
   transaction-type="RESOURCE_LOCAL">
  <properties>
    <property name="hibernate.dialect"
      value="org.hibernate.dialect.DerbyDialect"/>
    <property name="hibernate.connection.driver_class"
      value="org.apache.derby.jdbc.EmbeddedDriver"/>
    <property name="hibernate.connection.username"
      value="app"/>
    <property name="hibernate.connection.password"
      value="app"/>
    <property name="hibernate.connection.url"
      value="jdbc:derby:example"/>
  </properties>
 </persistence-unit>
</persistence>
```

This sample `persitence.xml` file defines one persistence unit called *unit1*, which uses database *example* running on Apache Derby in embedded mode[9].

In case we do not explicitly list classes from the domain model (which contains mapping metadata), Hibernate will search the classpath for all classes annotated with `@Entity` annotation.

Creating mapping metadata is also very simple. You can use either XML or annotations. Annotations have following advantages over XML:

- Metadata are in the same file as code => less error-prone
- IDE and compiler can assist you.
- They are much shorter. XML requires a lot of boilerplate text (tags, ...)

But annotations have also disadvantages:

- They make source code less readable. For this situation is sometimes used the term “annotation hell”.
- Source codes are tied to JPA (or to Hibernate, when you need to use some Hibernate-specific annotation).
- You can have only one mapping.
- Some older tools don't work with them very well or not at all.

Despite the disadvantages above, annotations are currently the preferred way of expressing metadata in Java.

Lets look at the simple example of an entity class annotated with JPA annotations. Example below contains one entity class mapped to the table `customer`. Please notice, that it implements interface `Serializable`. This is the only restriction,

9 Embedded mode means that database server shares the JVM with the client application. It is not possible to connect to the server from outside the application.

which Hibernate puts on the domain model (each entity class has to implement interface `Serializable`).

```
@Entity  @Table(name = "customer")
public class Customer implements Serializable {
  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long dbid;

  @Column(name = „cust_name", length = 20)
  private String name;

  @OneToMany(mappedBy = „customer")
  private Set<Order> orders;
  … // Getters and setters
}
```

Attribute `dbid` is mapped as a primary key. It is automatically generated by the underlaying database, when a transient object is saved.

Semantics of other annotations and their attributes is probably clear - except `mappedBy` attribute of `OneToMany` annotation. This attribute specifies name of the attribute in the referenced class, which contains back reference (in this case reference to the instance of `Customer`). Again, it is the name of an attribute, not a column!

The aim of this thesis is not to replace Hibernate's reference documentation and therefore we won't go more in depth. For the more detailed description of Hibernate annotations please see [16].

When we have configured Hibernate and created mapping metadata, we can start writing persistence-related code. To increase maintainability of the application, this should be put into one place - the DAO layer. Enterprise applications often have layered design, because it helps to separate the concerns and increases testability and maintainability (and it is easier to distribute the work among developers).

There is one design pattern, which can help us to create DAO layer – it is called `GenericDAO` pattern. The basic idea is simple - create one interface and its implementation containing basic operations for manipulation with domain objects (`GenericDAO`) and then create one interface and its implementation for each domain object. This concrete DAOs extend the `GenericDAO` as shown on the Diagram 2.

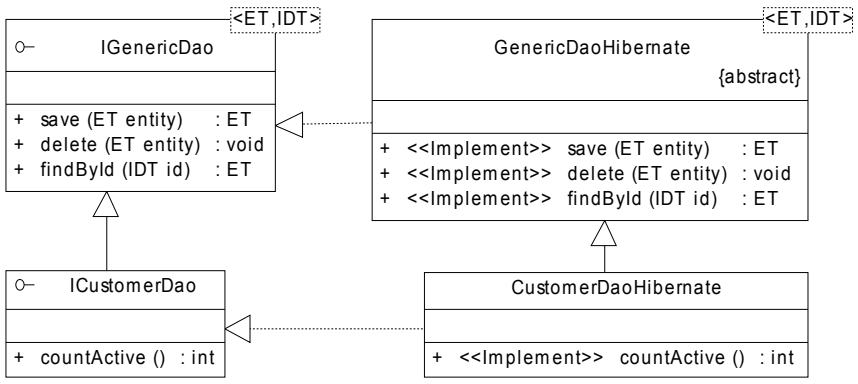


Diagram 2: Example of the GenericDAO design pattern

This may seem as a lot of work, but when Java 1.5 is used, it can be implemented via generic methods, which results in the significant reduction of the amount of code.

Lets look at the sample implementation of the `IGenericDao` interface. It uses dependency injection to initialize reference to an instance of the `EntityManager` class. This class is the main connection point between Hibernate and the rest of the application.

```
public class GenericDaoHibernate<ET, IDT>
  implements IGenericDao<ET, IDT> {

  @PersistenceContext(name = "manager1")
  private EntityManager entityManager;

  //initialized from generic type
  private Class<ET> entityType;

  public ET save(ET entity) {
    return entityManager.merge(entity);
  }

  public void delete(ET entity) {
    entityManager.remove(entity);
  }

  public ET findById(IDT id) {
    return entityManager.find(entityType, id);
  }
}
```

Please notice, that method `save` doesn't distinguish between persistent and transient objects. It lets Hibernate to decide, whether use INSERT or UPDATE statement.

Second interesting method is `findById`. By a single line of code and without any SQL, you are able to load entity with given identifier from the database. This method is particularly useful in web applications (request often contains identifier of the entity holding data, which should be displayed).

This was a really brief introduction into Hibernate usage via `EntityManager` class and `GenerficDao` design pattern. There are plenty of great books covering this topic (see section 1.8.1 for their list), but we suggest the official documentation ([17]), because it contains the most relevant information (this part of Hibernate changes quite quickly).

2. Expressive Power of HQL

2.1. What We Want to Achieve

In this chapter, we will try to formally prove, that HQL has at least same expressive power as the relational algebra. For understanding this proof (and the majority of this chapter), at least basic knowledge of the relational algebra is required. Please see [18] or [19] for the basic introduction to this topic. In the rest of this chapter, we will use acronym RA, when speaking about relational algebra.

Why do we need to know, whether HQL is as strong as RA? The answer is simple – it strongly influences the feasibility of Hibernate (which is the topic of this thesis). Imagine for example, that we would prove that HQL doesn't support restrictions. This would mean that Hibernate can't be used in systems with larger database tables (for performance reasons – all rows would have to be fetched).

Very natural question is: "Can't we use the fact that HQL is translated into SQL?". The power of SQL is well-researched, therefore it makes sense to try to base our proof on this fact. Unfortunately, from the fact that HQL is translated into SQL can be deduced only that SQL is as strong as HQL and nothing more. Therefore this fact would help us only when we proved that HQL is as strong as SQL. But this is more difficult than proving that HQL is as strong as RA.

We have explained why it is not possible to base our proof on the fact that HQL is translated into SQL. Basic idea behind the proof introduced in section 2.3 is to define translations of all language constructs of RA into HQL. This translation will be based on the definition of a relation-to-class mapping.

Unless otherwise mentioned, we mean SQL-92 ([20]) when referring to SQL. This version was chosen for the following reasons:

- It is implemented by nearly all RDBMS (at least majority of this standard)
- It doesn't contain object extensions.
- It is mature.

2.2. HQL Basics

HQL is similar to SQL, but instead of database tables and columns it uses classes and their attributes. It is case-insensitive, except for names of Java classes and attributes.

We will briefly describe the basic language constructs. For more details please refer to [1] (Chapter 14) or [2] (Chapter 14).

2.2.1. Example Schema

```
public class Customer implements Serializable {
     private Long dbid;  // primary key
     private String name;
     private Set<Order> orders;
     … // Getters and setters
}
```

```
public class Order implements Serializable {
     private Long dbid;  // primary key
     private Date date;
     private BigDecimal total;
     private Customer customer;
     private List<Item> items;
     … // Getters and setters
}

public class Item implements Serializable {
  private Long dbid;     // primary key
  private BigDecimal price;
  private Product product;
  private Order order;
  … // Getters and setters
}

public class Product implements Serializable {
  private Long dbid;     // primary key
  private String name;
  private BigDecimal price;
  … // Getters and setters
}
```

2.2.2. The FROM Clause

Unlike SQL, in HQL is SELECT clause optional. The simplest (correct) query is:

```
FROM xxx
```

(xxx stands for the name of a java class, which is being queried). In our example we can write either:

```
FROM model.Customer
```

or simply (thanks to *auto-import*):

```
FROM Customer
```

Aliases work exactly like in SQL (using optional keyword *AS*):

```
FROM Customer AS cust
```

If you specify more than one class in the FROM clause, the result will be Cartesian product (or "cross" join):

```
FROM Customer AS cust, Order AS o
```

You do not have to use only entities (i.e. classes, which were mapped as persistent), but also their supertypes. Queries, which use supertype of more than one entity, are called "polymorphic queries". Following example returns all persistent objects:

```
FROM java.lang.Object
```

2.2.3. Associations and Joins

Hibernate supports the same join types as SQL (with similar semantics):

- inner join
- left outer join
- right outer join
- full join (rarely used)

There is also one type, which is not in SQL (it doesn't make sense here), but in HQL is very common. It is called “fetch join“ and allows user to declare that some lazily fetched association should be initialized (data should be fetched).

Here are some examples:

`FROM Order AS order FETCH JOIN order.items` (returns all orders with all their items)

`FROM Customer As cust INNER JOIN cust.orders` (returns all customers with at least one order)

If you need to supply some join condition, use the WITH keyword (usage is similar to WHERE, which will be described later). Following example will return all customers and each customer will have reference to the collection of his/her orders with total price greater than 10:

```
FROM Customer AS cust LEFT JOIN cust.orders AS order
WITH order.total > 10
```

2.2.4. The SELECT Clause

HQL SELECT clause works exactly the same way as SQL SELECT does (if we ignore the difference between classes and relations). However there are some improvements:

- Possibility to use constructors
- Implicit joins
- Virtual attributes – id, size …

Possibility to use constructors

By default, query with multiple projections returns a list of tuples, but you can transform it into the list of domain objects or the list of smaller tuples. The only prerequisite is that object being returned has a constructor with arguments corresponding to the objects in tuple (same types and order).

Following example returns list of all customers (we assume that `Customer` has constructor with two arguments – one of type `Long`, second of type `String`):

```
SELECT new Customer(c.did, c.name) FROM Customer c
```

Please note, that you can use an arbitrary class – not only those from domain model.

Implicit joins

This means the possibility to create a projection to the collection-typed attributes:

```
SELECT c.orders FROM Customer c WHERE c.dbid = 1
```

Query above will return orders of the customer with primary key equal to 1. It is equal to the following query:

```
SELECT o FROM Customer c
      LEFT JOIN c.orders o
WHERE c.dbid = 1
```

Virtual attributes

Each domain object has "virtual attribute" called `id`, which is an alias for the actual attribute representing primary key (simple or composite). Therefore query:

```
SELECT c.id FROM Customer c
```

is equal to the query:

```
SELECT c.dbid FROM Customer c
```

Another "virtual attribute" is called `size`. It is applicable only to the collections, because it serves as an alias to the function `size()`. Therefore query, which returns all customers with more than ten orders, can be expressed as:

```
FROM Customer c WHERE c.orders.size > 10
```

2.2.5. The WHERE Clause

Again, WHERE Clause in HQL is similar to the SQL one. You can use there all "virtual attributes" described in the previous section.

Implicit joins are even more powerful, because they allow you to put restrictions on transitively accessed attributes without need to write complex query:

```
FROM Customer c WHERE c.orders.items.product.name = "PC"
```

Without implicit joins, this query would look like:

```
FROM Customer c JOIN c.orders o
                JOIN o.items i
                JOIN i.product p
WHERE p.name = "PC"
```

This HQL construct is really useful, but you have to use it carefully, because it can lead to superfluous joins (and join is an expensive operation, therefore when abusing implicit joins, you would be soon dealing with serious performance issues).

2.2.6. Expressions

Following expressions are allowed in WHERE clause (the list is not complete, it contains only the most important or interesting expressions. Complete list can be found in [18], Chapter 14):

- Operators
  - Mathematical: `+, -, *, /`
  - Comparison: `=, >=, <=, <>, !=, LIKE`
  - Logical: `AND, OR, NOT`
- Parentheses: `( )`
- `IN, BETWEEN, IS NULL, IS EMPTY, MEMBER OF` (and their negations such as `NOT IN, IS NOT NULL` ...)

- Any function or operator defined by EJB-QL 3.0: `abs()`, `bit_length()`, `mod()`, `length()`, `lower()`, `locate()`, `sqrt()`, `substring()`, `trim()`, `upper()`
- `CAST(? AS ?)` – the second argument is the name of a Hibernate type.
- `index()` - function, that applies to aliases of a joined indexed collection
- Collection-valued path expressions: `size()`, `elements()`, `minelement()`, `maxelement()`, `indices()`, `minindex()`, `maxindex()`
- Any database-supported SQL scalar function.
- ? - JDBC-style positional parameters
- Named parameters (for example `:name`)
- SQL literals - `'string'`, `42`, `0.01E+10`, `'1984-04-22 11:59:59.0'`
- Java public static final constants (`Math.PI`) and enums

The most important part of a list above are the named parameters and therefore we will look at them more in depth.

Imagine complex query with a lot of parameters:

```
FROM Customer c JOIN c.orders o JOIN o.items i
  WHERE upper(c.name) LIKE ?
          AND  o.total < ?
          AND o.total > ?
          AND ( o.date BETWEEN ? AND ? )
          AND i.price < ?
          AND i.price > ?
```

It is really easy to make a mistake when counting question marks (when you want to specify parameter's value). With named parameters, query would look like this:

```
FROM Customer c JOIN c.orders o JOIN o.items i
  WHERE upper(c.name) LIKE :name
          AND  o.total < :maxTotal
          AND o.total > :minTotal
          AND (o.date BETWEEN :start AND :end)
          AND i.price < :maxPrice
          AND i.price > :minPrice
```

This query is more transparent and less error-prone. But that is not all. Named parameters have another significant advantage – you can use one parameter multiple times. Please look at the following query:

```
FROM Customer c JOIN c.orders o JOIN o.items i
  WHERE o.total = :price AND i.price = :price
```

When you specify the value of the `price` parameter, Hibernate will replace (more precisely said: bind) both occurrences of this parameter with supplied value. This functionality can not be implemented using positional parameters (i.e. question marks).

2.2.7. The ORDER BY Clause

Is exactly the same as in SQL (the only difference is that ORDER BY in HQL works with attributes instead of columns). Example:

```
FROM Product p ORDER BY p.price DESC, p.name ASC
```

2.2.8. The GROUP BY Clause

Again, the basic syntax is the same as in SQL. When you want to put restriction on some attribute used in GROUP BY clause, use the HAVING keyword:

```
SELECT sum(o.total) FROM Order o
     GROUP BY o.date
          HAVING o.date > '2008-08-08'
```

Please note, that you can use aggregate functions (and other SQL functions) in the HAVING clause – but only if the underlying database system supports them. Although this requirement is natural, it reduces the portability (between different RDBMS).

2.2.9. Subqueries

HQL supports subqueries, but only in case that underlying database supports subselects. Subqeries can be correlated (i.e. can contain reference(s) to the alias(es) from the parent query) and can be used in WHERE or SELECT clauses.

Each subquery have to be surrounded by parentheses:

```
FROM Customer c
     WHERE NOT EXISTS
          (FROM Customer c2 JOIN c2.orders
               WHERE c2.id=c.id)
```

2.3. Translation of Relational Algebra into HQL

In this chapter, we will show that HQL has at least same expressive power as relational algebra (RA). More precisely said, we will try to proof that each query, which can be expressed in RA, can be expressed also in HQL.

We will provide a „mapping“ from each RA's language constructs into HQL.

2.3.1. Definitions

Please note, that following definitions may not cover all aspects of the object theory (mainly behavioral ones - such as polymorphism). It would bring undesirable complexity to the proof discussed in this chapter.

Definition 1 (according [19]): Domain

A finite set of atomic elements (values).

Definition 2 (according [19]): Relation

Relation is a set of attribute-domain pairs. „All the attributes of a relation must be different. An attribute uniquely identifies the role played by a domain in a relation. The domains referenced in a relation are not necessarily all different. The collection of attribute-domain pairs of a relation is not ordered.

A relation named R with n attribute-domain pairs is noted as $R(A_1:D_1,...,A_n:D_n)$, where the A_i's are the attributes and the D_i's are the domains (more precisely, domain names). Formally, the correspondence between attributes and domains in a relation

can be noted as a function $\{(A_1 \rightarrow D_1) ..... (A_i \rightarrow D_i) ..... (A_n \rightarrow D_n)\}$.“[19] (Section 3.2.1)

Definition 3 (according [19]): Relation Value

„Relation value (or simply value) of a relation $R(A_1:D_1, ..., A_n:D_n)$ is a set of n-tuples $<d_1, ..., d_n>$ of elementary values, where d_i belongs to D_i for all i between 1 and n. More exactly, the value of R is a set of labeled n-tuples $<A_1:d_1, ..., A_n:d_n>$ of values, where each value is explicitly associated with an attribute. Thus, a labeled n-tuple is an unordered set of attribute-value pairs.

Formally, at every moment, the value of a relation R is a subset of the generalized Cartesian product of its domains indexed by its attributes. The indexing is specified by the correspondence between attributes and domains, described as a function in the relation structure.“ [19] (Section 3.2.1)

Relation Instance

Relation instance of a relation $R(A_1:D_1, ..., A_n:D_n)$ is a n-tuple $<d_1, ..., d_n>$ of elementary values, where d_i belongs to D_i for all i between 1 and n. Thus, a relation value is a set of relation instances.

Class

Set of attribute-type pairs and operations. All attributes must be different. Operations are called methods. A class named C with n attribute-type pairs is noted as $C(A_1:T_1, ..., A_n:T_n, O_1, …, O_m)$, where the A_i's are the attributes, T_i's are the domains (more precisely, domain names), the O_j's are operations (methods) and:

$$1 <= i <= n \in N$$
$$1 <= j <= m \in N$$
$$0 < m + n$$

Type

A domain or a finite set of instances of the same class.

Object

Instance of a class. Instance of a class $C(A_1:T_1, ..., A_n:T_n, O_1, …, O_m)$ is n-tuple $<t_1, ..., t_n>$, where d_i belongs to T_i for all i between 1 and n.

2.3.2. Basic Concepts

In this section, we will provide a mapping between type and domain, class and a relation and between object and relation instance.

Please notice, that type is an extension to domain. Therefore it is possible to express all domains using types. This implies, that relation $R(A_1:D_1,..., A_n:D_n)$ is equal to the relation $R(A_1:T_1,..., A_n:T_n)$, where T_i is a type representing domain D_i (for all i between 0 and n).

Relation $R(A_1:T_1,..., A_k:T_k)$ complies with the definition of a class, where:

$$k = n$$
$$m = 0$$

Therefore a relation $R(A_1:T_1, ..., A_n:T_n)$ is equal to the class $C(A_1:T_1, ..., A_n:T_n)$.

Relation instance of a relation $R(A_1:T_1,..., A_n:T_n)$ is n-tuple $<t_1,...,t_n>$ (by the definition), which also complies with the definition of instance of the class $C(A_1:T_1,..., A_n:T_n)$.

Conclusion is, that each relation can be represented as a class and each relation instance can be represented as an object.

2.3.3. Projection

Definition 4 (according [19]): Projection

Given a tuple t: $A \rightarrow D$ of a relation R(A, B), the projection of t on attributes B is a tuple t' defined as follows:

$$t': B \rightarrow D \text{ such that } \exists b \in B \; t'(b) = t(b)$$

The tuple t' thus obtained will also be noted t[B] .

Given a relation R(A:DA, B:DB), the projection of R on attributes B is a relation $R_1(B:DB)$ denoted R[B] (i.e. $R[B_1 , \ldots, B_m]$ if $B = \{B_1, ..., B_m\}$) whose value is given by:

$$\{s : B \rightarrow DB \mid \exists t \in R \;\; \forall b \in B \; t(b) = s(b) \}$$

or, equivalently by:

$$\{t[B] \mid t \in R\}$$

Lemma 1

Given a relation R(A:DA,B:DB), the result of the projection of R on attributes B (R[B]) is equal to the result of the following HQL query:

```
SELECT new C1(b1, ..., bn) FROM C
```

Where C is defined as $C(A_1:TA_1, \ldots , A_m:TA_m, B_1:TB_1, \ldots B_n:TB_n)$ and C_1 is defined as $C_1(B_1:TB_1, \ldots B_n:TB_n)$[10].

Proof

First, we will use mapping defined in section 2.3.2. Thus:

(1) $R(A_1:DA_1,\ldots, A_m:DA_m, B_1:DB_1, \ldots, B_n:DB_n)$

can be expressed as

$C(A_1:TA_1, \ldots, A_m:TA_m, B_1:TB_1, \ldots B_n:TB_n)$

and

(2) $R_1(B_1:DB_1, \ldots, B_n:DB_n)$

can be expressed as

$C_1(B_1:TB_1, \ldots B_n:TB_n)$.

Assuming that a result of the query:

```
FROM C
```

is the set S of objects {o | o instance of C} implies that a result of the query:

```
SELECT new C1(b1, ..., bn) FROM C
```

is (by the definition of SELECT operation) a set S_1 of objects

10 As you may have noticed, we need one class for each combination of attributes (each projection list). This is possible, because each class have to have finite number of attributes. Although this is theoretically correct, in practice it would be better to use the same class (with multiple constructors) or array of objects (tuple).

$$\{o_1 \mid o_1 \text{ instance of } C_1 \ \& \ \exists o \in S \text{ such that } \forall j \in \{1, \ldots, n\}\ o_1(b_j) = o(b_j)\}$$

Now, we can rewrite the definition of projection using (1) and (2):

Given a relation R represented as a class C(A:TA,B:TB), the "projection" of R on attributes B is a relation R_1 represented as a class C_1(B:TB) whose value is given by:

$$\{o : B \rightarrow TB \mid \exists p \text{ instance of } C \text{ such that } \forall b \in B\ o(b) = p(b)\}$$
$$=$$
$$\{o \mid o \text{ instance of } C_1 \ \& \ \exists p \text{ instance of } C \text{ such that } \forall b \in B\ o(b) = p(b)\}$$
$$=$$
$$\{o \mid o \text{ instance of } C_1 \ \& \ \exists p \in S \text{ such that } \forall b \in B\ o(b) = p(b)\}$$
$$=$$
$$\{o \mid o \text{ instance of } C_1 \ \& \ \exists p \in S \text{ such that } \forall j \in \{1, \ldots, n\}\ o(b_j) = p(b_j)\}$$
$$=$$
$$\{o_1 \mid o_1 \text{ instance of } C_1 \ \& \ \exists o \in S \text{ such that } \forall j \in \{1, \ldots, n\}\ o_1(b_j) = o(b_j)\}$$
$$= \text{result of the query: } \texttt{SELECT new } C_1\texttt{(}b_1\texttt{, ..., }b_n\texttt{) FROM C}$$

Q.E.D

2.3.4. Rename

Definition 5 (according [19]): Rename

„An operation that renames the attributes of a relation can be presented as a function "rename(R,M)": its arguments are an expression denoting a relation R and a correspondence of attributes M, and its value is the relation with renamed attributes.

Formally, let $A = \{A_1,...,A_n\}$ be the attributes of R and $B = \{B_1,...,B_n\}$ be the desired attributes for the renamed relation. M can be described as a functional mapping $\{(A_i \rightarrow B_j)\}$ which actually is bijective or one-to-one, that is, functional in both directions. M can be a partial renaming which affects only a subset of the attributes of R. In particular, the renaming operation leaves its relation argument unchanged when this argument has no attribute subject to renaming."

Note

Definition above holds for classes too (it is clear from 2.3.2 and definitions of a class and a relation). Therefore it is not necessary to define a mapping between RA and HQL.

The correspondence function M can be defined in HQL using keyword "AS". For example:

```
SELECT b1 AS a1, b2 AS a2 FROM C
```

But this query returns a tuple (not a class).

2.3.5. Restriction (Selection)

Definition 6 (according [19]): Restriction

Given a relation R, the restriction of R is the subset of tuples of R which satisfy a specified condition.

Formally, let R(A:DA) be a relation, with $A = \{A_l, ..., A_n\}$. Then $R[A_i\ \theta\ c]$, where A_i is an attribute in A, is a restriction of R, if c is a constant of the domain D_i

associated with A_i and θ is any binary comparison operation (such as EQ, NE, GT, GE, LT, LE) defined for values of that domain.

The relation value R_l of $R[A_i \theta c]$ is given by:

$$\{t \mid t \in R \text{ and } t(A_i)\ \theta\ c\}$$

Another form of restriction is written $R[A_i \theta A_j]$, where A_i and A_j must be associated with the same domain in relation R, and θ is a binary comparison operation defined for values of that domain. The value of $R[A_i \theta A_j]$ is given by:

$$\{t \mid t \in R \text{ and } t(A_i)\ \theta\ t(A_j))$$

Lemma 2

Let $R(A_l{:}DA_l, A_n{:}DA_n)$ be a relation. Then the result of a restriction $R[A_i \theta c]$, where c is a constant of the domain D_i associated with A_i and θ is a binary comparison operation ∈ {EQ, NE, GT, GE, LT, LE}, is equal to the result of the following HQL query:

`FROM C WHERE` $A_i\ \theta\ c$

Where C is defined as $C(A_1{:}TA_1, \ldots, A_n{:}TA_n)$ and $c \in TA_i$.

Proof

In section 2.3.2 we have proven that:

(1) $R(A_l{:}DA_l, A_n{:}DA_n)$

can be expressed as

$$C(A_1{:}TA_1, \ldots, A_n{:}TA_n)$$

The result of a query „`FROM C WHERE` $A_i\ \theta\ c$“ is (by definition) a set S =

$$\{o \mid o \text{ instance of C such that } o(A_i)\ \theta\ c\}$$

Definition of a restriction can be rewritten using (1):

Given relation $R(A_l{:}DA_l, A_n{:}DA_n)$ represented as class $C(A_1{:}TA_1, \ldots, A_n{:}TA_n)$, $R[A_i \theta c] = \{o \mid o \text{ instance of C such that } o(A_i)\ \theta\ c\}$

Q.E.D

Note

Proof for a second form of a restriction ($R[A_i \theta A_j]$) is almost the same as the proof above (simply replace “c“ with “A_j“ or “$o(A_j)$“).

Lemma 3

Let $R(A_l{:}DA_l, A_n{:}DA_n)$ be a relation. Then the result of a restriction

$$R[A_1\ \theta_1\ c_1][A_2\ \theta_2\ c_2]...[A_p\ \theta_p\ c_p]$$

where p<=n and ∀i∈ {1,...,p} c_i is a constant of the domain D_i associated with A_i and θ_i is a binary comparison operation ∈ {EQ, NE, GT, GE, LT, LE}, is equal to the result of a following HQL query:

(*) `FROM C WHERE` $A_1\ \theta_1\ c_1$ `AND` $A_2\ \theta_2\ c_2$ `AND ... AND` $A_p\ \theta_p\ c_p$

Where C is defined as $C(A_1{:}TA_1, \ldots, A_n{:}TA_n)$ and $\forall i \in \{1,...,p\}\ c_i \in T_i$.

Proof

In section 2.3.2 we have proven that:

(1) $R(A_1:DA_1, \ldots, A_n:DA_n)$

can be expressed as

$$C(A_1:TA_1, \ldots, A_n:TA_n)$$

The result of a query (*) is (by definition) a set S =

$$\{o \mid o \text{ instance of C such that } \forall i \in \{1,...,p\}\ o(A_i)\ \theta\ c_i\}$$

Definition of a restriction can be rewritten using (1):

Given relation $R(A_1:DA_1, A_n:DA_n)$ represented as class $C(A_1:TA_1, \ldots, A_n:TA_n)$,

$$R[A_1\ \theta_1\ c_1][A_2\ \theta_2\ c_2]...[A_p\ \theta_p\ c_p]$$

$$=$$

$$\{o \mid o \text{ instance of C such that } \forall i \in \{1,...,p\}\ o(A_i)\ \theta_i\ c_i\}$$

Q.E.D

2.3.6. Cartesian Product

Definition 7 (according [19]): Cartesian Product

The Cartesian product of two relations R_1 and R_2 is the set of all tuples t such that t is the concatenation of a tuple t_1 belonging to R_1 and a tuple t_2 belonging to R_2. Formally, given $R_1(A:DA)$ and $R_2(B:DB)$ with A and B disjoint, the Cartesian product $R_1 \times R_2$ of R_1 and R_2 is a relation R(A:DA,B:DB) whose value is given by:

$$\{t : A \cup B \rightarrow DA \cup DB \mid t[A] \in R_1 \text{ and } t[B] \in R_2\}$$

or, equivalently:

$$\{t_1 + t_2 \mid t_1 \in R_1 \text{ and } t_2 \in R_2\}$$

Lemma 4

Given a relation $R_1(A:DA)$ and $R_2(B:DB)$ with A and B disjoint, the result of Cartesian product of R_1 and R_2 (i.e. relation R(A:DA, B:DB)) is equal to the result of the following HQL query:

```
SELECT new C(a1, ..., am, b1, ..., bn) FROM C1, C2
```

Where C is defined as $C(A_1:TA_1, \ldots, A_m:TA_m, B_1:TB_1, \ldots, B_n:TB_n)$, C_1 is defined as $C_1(A_1:TA_1, \ldots, A_m:TA_m)$ and C_2 is defined as $C_2(B_1:TB_1, \ldots, B_n:TB_n)$.

Proof

Section 2.3.2 implies that:

(1) $R(A_1:DA_1,\ldots, A_m:DA_m, B_1:DB_1, \ldots, B_n:DB_n)$

can be expressed as

$$C(A_1:TA_1, \ldots, A_m:TA_m, B_1:TB_1, \ldots B_n:TB_n)$$

and

(2) $R_1(A_1:DA_1, \ldots, A_m:DA_m)$

can be expressed as

$$C_1(A_1:TA_1, \ldots A_m:TA_m).$$

and

(3) $R_2(B_1:DB_1, \ldots, B_n:DB_n)$

can be expressed as

$$C_2(B_1:TB_1, \ldots B_n:TB_n).$$

By the definition, a result of the query:

(4) `FROM C1, C2`

is a set K of 2-tuples $\{o_1 + o_2 \mid o_1$ instance of C_1 and o_2 instance of $C_2\}$. Therefore result of the following query:

`SELECT new C(a1, ..., am, b1, ..., bn) FROM C1, C2`

is (by the definition of SELECT operation and (4)) a set S of objects

$\{o \mid o$ instance of $C \ \& \exists o_1$ instance of C_1 such that $\forall a \in A\ o_1(a) = o(a)$
$\& \exists o_2$ instance of C_2 such that $\forall b \in B\ o_2(b) = o(b)\}$

Now, we can rewrite the definition of Cartesian product using (1), (2) and (3): Given $R_1(A:DA)$ represented as a class $C_1(A:TA)$ and $R_2(B:DB)$ represented as a class $C_2(B:TB)$ with A and B disjoint, the Cartesian product $R_1 \times R_2$ of R_1 and R_2 is a relation R(A:DA,B:DB) represented as a class C(A:TA,B:TB) whose value is given by:

$$\{o : A \cup B \rightarrow TA \cup TB \mid \exists o_1 \text{ instance of } C_1 \text{ such that } \forall a \in A\ o_1(a) = o(a)$$
$$\& \exists o_2 \text{ instance of } C_2 \text{ such that } \forall b \in B\ o_2(b) = o(b)\}$$
$$=$$
$$\{o \mid o \text{ instance of } C \ \& \exists o_1 \text{ instance of } C_1 \text{ such that } \forall a \in A\ o_1(a) = o(a)$$
$$\& \exists o_2 \text{ instance of } C_2 \text{ such that } \forall b \in B\ o_2(b) = o(b)\}$$

Q.E.D

2.3.7. θ–Join

Definition 8 (according [19]): θ–Join

Given two relations R_1 and R_2, and a condition $A_1 \theta B_1$ where A_1 is an attribute of R_1 and B_1 is an attribute of R_2, the θ-join of R_1 and R_2 is the set of all tuples t such that t is the concatenation of a tuple t_1 of R_1 and a tuple t_2 of R_2 such that the condition $t_1(A_1)\ \theta\ t_2(B_1)$ is satisfied.

Formally, let $R_1(A:DA)$ and $R_2(B:DB)$ be two relations and assume that their attributes have been renamed so that they have no attribute in common. Then "$R_1[Ai\ \theta\ Bj]R_2$" is a θ-join of R_1 and R_2, if A_i is an attribute of R_1 and B_j is an attribute of R_2, which are associated in R_1 and R_2 with the same domain, and if θ is a binary operation (like EQ, NE, GT, GE, LT, LE) defined for values of that domain.

The relation value of $R_1[A_i\ \theta\ B_j]R_2$ is defined to be:

$$\{t : A \cup B \rightarrow DA \cup DB \mid t[A] \in R_1 \text{ and } t[B] \in R_2 \text{ and } t(A_i)\ \theta\ t(B_j)\}$$

or, equivalently:

$$\{t_1 + t_2 \mid tl \in R_1 \text{ and } t_2 \in R_2 \text{ and } t_1(A_i)\ \theta\ t_2(B_j)\}$$

Note

From the definition of a θ-join is clear[11] that:

$$R_1[A_i\ \theta\ B_j]R_2 = R_1 \times R_2[A_i\ \theta\ B_j]$$

11 Value of Cartesian product is defined as $\{t : A \cup B \rightarrow DA \cup DB \mid t[A] \in R_1$ and $t[B] \in R_2\}$ and value of a restriction is defined as $\{t \mid t \in R$ and $t(A_i)\ \theta\ t(B_j)\}$. If we put these two conditions together we get $\{t : A \cup B \rightarrow DA \cup DB \mid t[A] \in R_1$ and $t[B] \in R_2$ and $t(A_i)\ \theta\ t(B_j)\}$, which is the definition of the θ-join.

Therefore it is natural to try to map the θ-join to the following query (because we can use 2.3.5 and 2.3.6 to prove it):

```
SELECT C(A1,..., Am, B1, ..., Bn)
      FROM (SELECT new C(A1,..., Am, B1, ..., Bn)
            FROM C1 AS c1, C2 AS c2 )
WHERE Ai θ Bj
```

Unfortunately, this approach is wrong, because HQL currently doesn't support subqueries in FROM clause.

Lemma 5

Let R_l(A:DA) and R_2(B:DB) be two relations and assume that their attributes have been renamed so that they have no attribute in common. Relation value of their θ- join ($R_l[A_i\ \theta\ B_j]R_2$) is equal to the result of the following query:

(*) `SELECT new C(A1, ..., Am, B1, ..., Bn) FROM C1 AS c1, C2 AS c2 WHERE c1.Ai θ c2.Bj`

Proof

Result of the (*) is a set S =

$\{o \mid o \text{ instance of } C\ \&\ o(A_i)\ \theta\ o(B_j)$
$\&\ \exists o_1$ instance of C_1 such that $\forall k \in \{1,...,m\}\ o_1(A_k) = o(A_k)$
$\&\ \exists o_2$ instance of C_2 such that $\forall l \in \{1,...,n\}\ o_2(B_1) = o(B_1)\ \}$

In section 2.3.5 we have proven that set $S_l = \{o \mid o \text{ instance of } C\ \&\ o(A_i)\ \theta\ o(B_j)\}$ corresponds to the value of a restriction $R[A_i\ \theta\ B_j]$.

In section 2.3.6 we have proved that a set S_2 =

$\{o \mid o \text{ instance of } C\ \&\ \exists o_1$ instance of C_1 such that $\forall k \in \{1,...,m\}\ o_1(A_k) = o(A_k)$
$\&\ \exists o_2$ instance of C_2 such that $\forall l \in \{1,...,n\}\ o_2(B_l) = o(B_l)\ \}$

corresponds to the value of a Cartesian product $R_1 \times R_2$.

Therefore (and because of a relation nesting[12]) set S is a relation value of a relation $R_1 \times R_2[A_i\ \theta\ B_j] = R_l[A_i\ \theta\ B_j]R_2$

Q.E.D

2.3.8. Natural Join

Definition 9 (according [19]): Natural join

Informally, the natural join of two relations R_1 and R_2 is the equi-join (θ-join with "θ" being "=") of the two relations on their corresponding attributes, with only one of each pair of corresponding attributes being retained in the result.

Formally, let R_1(A:DA, B:DB) and R_2(B:DB, C:DC) be two relations such that, after suitable renamings:

(a) Each attribute in B is associated with the same domain in both R_1 and R_2

(b) Attributes in A are all different from attributes in C.

12 For the precise definition of this term please see [19], section 4.3.1

The natural join $R_1 * R_2$ of R_1 and R_2 is a relation R(A:DA, B:DB, C:DC) whose value is given by:

$$\{t : A \cup B \cup C \rightarrow DA \cup DB \cup DC \mid t[A \cup B] \in R_1 \text{ and } t[B \cup C] \in R_2\}$$

or, equivalently:

$$\{t_1 + t_2[C] \mid t_1 \in R_1 \text{and } t_2 \in R_2 \text{ and } t_1[B] = t_2[B]\}$$

Lemma 6

Let R_1(A:DA, B:DB) and R_2(B:DB, C:DC) be two relations satisfying (a) and (b). Let R(A:DA, B:DB, C:DC) = $R_1 * R_2$. Then a value of the relation R is equal to a value of the following query:

(*)
```
SELECT new C(A1, ..., Am, B1, ..., Bn, C1, ..., Cp)
      FROM C1 AS c1, C2 AS c2
      WHERE  c1.B1 = c2.B1 AND
             c1.B2 = c2.B2 AND … AND
             c1.Bn = c2.Bn
```

Proof

Section 2.3.2 implies that:

(1) $R(A_1:DA_1,\ldots, A_m:DA_m, B_1:DB_1, \ldots, B_n:DB_n, C_1:DC_1, \ldots, C_p:DC_p)$
can be expressed as
$C(A_1:TA_1, \ldots, A_m:TA_m, B_1:TB_1, \ldots B_n:TB_n, C_1:TC_1, \ldots, C_p:TC_p)$

and

(2) $R_1(A_1:DA_1, \ldots, Am:DAm, B_1:DB_1, \ldots, B_n:DB_n)$
can be expressed as
$C_1(A_1:TA_1, \ldots , Am:TAm, B_1:TB_1, \ldots, B_n:TB_n).$

and

(3) $R_2(B_1:DB_1, \ldots, B_n:DB_n, C_1:DC_1, \ldots, C_p:DC_p)$
can be expressed as
$C_2(B_1:TB_1, \ldots, B_n:TB_n, C_1:TC_1, \ldots, C_p:TC_p).$

We can rewrite the definition of a natural join using (1), (2) and (3):

Let relation R_1(A:DA, B:DB) represented as a class C_1(A:TA, B:TB) and R_2(B:DB, C:DC) represented as C_2(B:TB, C:TC) satisfy (a) and (b). The natural join $R_1 * R_2$ of R_1 and R_2 is a relation R(A:DA, B:DB, C:DC) represented as a class C(A:TA, B:TB, C:TC) whose value is given by:

$\{o \mid o \text{ instance of } C \;\&\; \exists o_1 \text{ instance of } C_1 \text{ such that } \forall a \in A\ o_1(a) = o(a)$
$\;\&\; \exists o_2 \text{ instance of } C_2 \text{ such that } \forall c \in C\ o_2(c) = o(c)$
$\;\&\; \forall b \in C\ o_1(b) = o_2(b) = o(b)\}$

$=$

$\{o \mid o \text{ instance of } C \;\&\; \exists o_1 \text{ instance of } C_1 \text{ such that } \forall d \in A \cup B\ o_1(d) = o(d)$
$\;\&\; \exists o_2 \text{ instance of } C_2 \text{ such that } \forall d \in B \cup C\ o_2(d) = o(d)$
$\;\&\; \forall i \in \{1,\ldots,n\}\ o_1(B_i) = o_2(B_i)\}$

= result of the query (*)

Q.E.D

2.3.9. Union

Unfortunately HQL doesn't support unions and there is no way, how to simulate it. Intersection and difference are not supported either. Therefore the statement „HQL is at least as strong as RA“ is false.

Definition 10: RA_{hql}

RA_{hql} is a relational algebra without following operations:

- Intersection
- Union
- Difference.

Lemma 7

RA_{hql} is closed under composition.

Proof

[19] (Section 4.3). This proof doesn't use neither of excluded operations and therefore it can be applied to RA_{hql} too.

Note

Excluded operations are not used in any previous section (rename, projection, restriction, Cartesian product, θ-join and natural join). Thus all previously proved lemmas holds for RA_{hql} too.

2.3.10. Theorem

Theorem

Each query, which can be expressed in RA_{hql}, can be expressed in HQL. Thus HQL is at least as strong as RA_{hql}.

Proof

It is clear from the Lemma 1, 2, 3, 4, 5 and 6 and from sections 2.3.2 and 2.3.4.

2.3.11. Conclusion

We have proved that each query, which can be expressed in relational algebra without intersection, union and difference, can be expressed also in HQL.

Because HQL doesn't support these operations, it can't be stronger than RA. Fortunately, this is not a big limitation, because in a real world these operations are used rarely (usually there is another way, how to express the same query without using these operations).

This knowledge can be used for construction of a HQL query optimizer based on relational optimizations, but it goes beyond the scope of this thesis.

3. Using Hibernate in Systems with Complex Domain Model

3.1. Reference Systems

3.1.1. OpenTCM

OpenTCM is a system supporting QA process in a middle or large-sized IT company. It is a middle-sized web application written in Java language and it uses Hibernate as its persistence technology.

OpenTCM was chosen because it is a system, which was designed for use with ORM (Hibernate) from the very beginning.

It is the only system (used in this thesis), which has not been developed by Profinit company[13]. OpenTCM is an open source project developed by group of students from Charles University (Faculty of Math and Physics).

More information about this project you can find at address: www.opentcm.org

3.1.2. PPR Offline

PPR Offline is the representative of rich client applications. It is a large-sized application for maintaining insurance contracts (PPR is an acronym for: Insurance of an Entrepreneurial Risks). It is being developed by the Profinit company and therefore it is not possible to publish its source codes in this thesis.

PPR Offline was chosen because it is one of a few desktop applications, which use Hibernate.

3.1.3. Balicky

Balicky system is a large-sized enterprise application for maintaining insurance contracts. It is a web application. It uses plain JDBC, because in the time of its creation, there were no modern ORM available (it is more than eight years old). It has a very large data model (hundreds of tables) and contains hundreds of stored procedures. It was developed and is maintained by Profinit company.

Balicky was chosen, because it is a typical representative of systems, where using Hibernate doesn't bring any significant advantage.

3.2. Limitations

In this section we will try to answer, whether it is possible to use Hibernate in a systems with complex domain model and whether it somehow limits the design of the domain or data model. We will assume that domain and data model are fully under our control.

13 „Profinit is an independent provider of information technology services and solutions. It provides IT consulting within the full scope of software engineering disciplines, develop custom - tailored systems and offer solutions for Data Quality, Data and Application Integration, Identity Management and Enterprise Content Management“ (citation from: www.profinit.eu)

3.2.1. Data Types And Number of Attributes

Java and relational databases do not have the same data type sets. Hibernate therefore have to convert types between domain and data model. But is it able to convert all primitive and basic Java data types to their equivalents in relational database? The answer is yes (according to [2], Table 5.1).

And what about date represenation? Relational databases usually distinguish between date, time and date with time (usually called timestamp), but in Java there is only one type – `java.util.Date`. Hibernate solves this problem by introducing the annotation `Temporal`, which allows you to specify, whether attribute should be stored as `date`, `time` or `timestamp`.

Another potential problem resides in text strings. Relational databases use many string representations such as `char array`, `varchar` and `CLOB`, but Java has only one – `java.lang.String`. By default, Hibernate maps Java `Strings` to `varchars` ([2], Table 5.1), but you can force mapping to `CLOB` by using annotation `Lob` ([2], Table 5.3).

Annotation `Lob` can be used also for mapping byte arrays to `VARBINARY` or `BLOB`. You can also use `java.sql.Clob` and `java.sql.Blob` instead of `String` and `byte[]`.

If you are not satisfied with the default mapping strategy, you can define your own by implementing interface `UserType`. This way, you can map for example Oracle `XMLType` to `org.w3c.dom.Document`.

From the paragraphs above it is clear, that using Hibernate doesn't lay down any restrictions on data types used in a domain model.

Another potential restriction for the domain model design is the number of attributes, which can be mapped in a single entity class. Bad new is that such restriction really exists. It is not Hibernate's restriction, but a restriction of the underlying RDBMS (each database system somehow limits the number of columns per table, but in some systems this limit is so high, that it is not an actual problem). Hibernate allows you to map one entity into multiple tables ([16], Section 2.2.7) and therefore it can help you to go around the limitation of your RDBMS. On the other hand, large number of attributes (table columns) usually signals a bad design.

Conclusion is that Hibernate has the same requirements on the number of attributes per entity as the pure JDBC and it supports the same type set.

3.2.2. Identifiers (Primary Keys)

Lets look, how Hibernate solves identity problem[14]. Authors of the Hibernate suggest using an artificial primary key wherever it is possible ([2], Section 4.2.3). This approach has several advantages:

- Its value never changes (there is no reason for changing it).
- It can always be non-null, because Hibernate can generate it.
- It can be easily indexed (it is a number).

14 Identity problem is a part of the Impedance Mismatch problem, which was described in section 1.5

However there are some situations, when using artificial primary key is not the best solution. Artificial primary key is unsuitable for:

- Relationship tables.
- Tables containing reference data.

Artificial primary key stored in a single column is just a recommendation, not a restriction. Hibernate supports composite primary keys and it doesn't lay down any restrictions on the complexity of the primary key ([2], Section 8.1.1).

The real problem lies in the implementation of `equals` and `hashCode` methods in your entity classes. Problem occurs when you implement this methods using attribute (primary key), which is generated during database insert. This will cause that transient instances of such entities will be equal to each other.

There are two possible solutions – either use different generation strategy (for example "`hilo`") or implement `equals` and `hashCode` using attribute(s), which contains business key.

Another common mistake in implementation of `equals`, `hashCode` and `toString` methods is accessing attributes directly instead of using getters. Hibernate often creates dynamic proxies of your entity classes (for example to implement lazy fetching) and it doesn't initialize attributes from the superclass, but it overrides their getters to return the correct data.

Severity of this problem is illustrated by the effort spent on fixing errors caused by wrong implementation of `equals` and `hashCode` methods in OpenTCM (3 MD[15]) and PPR Offline (5 MD).

Lets summarize suggestions for the implementation of the `equals` and `hashCode` methods in entity classes:

- Always use getters and setters (everywhere).
- Never use attributes initialized during database insert.
- Use `instanceof` instead of comparison of results of `getClass` calls (this implementation is generated by Eclipse IDE by default).

3.2.3. Associations (Relationships)

We can divide binary associations based on navigability:

- Uni-directional
- Bi-directional

Hibernate supports both types, but working with bi-directional relationships is much more comfortable.

Another possible division is based on multiplicity:

- 1:1
- 1:0 and 0:1
- 1:N and N:1
- 0:N and N:0
- M:N

Hibernate uses following annotations to express these relationships:

- `OneToMany` (or `CollectionOfComponents`)

15 MD = Man-Day. Unit of production equal to the work one developer can produce in a day.

- `ManyToOne`
- `ManyToMany`
- `OneToOne` (or `Embedded`)

This annotations allows you to express all types of associations described above:

Association	Annotation
1:1	OneToOne
1:0	OneToOne with optional=true
0:1	OneToOne with optional=true (defined in opposite direction)
1:N	OneToMany
N:1	ManyToOne
0:N	ManyToOne with optional=true (defined in opposite direction)
N:0	ManyToOne with optional=true
M:N	ManyToMany

Table 1: Association types (based on multiplicity) and corresponing annotations

Table above proves that using Hibernate does not limit the types of associations, which can be used in the domain model. However, when your data and domain model do not correspond very well (this often happens in legacy systems), it might be difficult to find appropriate mapping. Hibernate provides additional mapping options, which goes beyond the scope of this thesis (for more information see [2], Chapter 6).

Composition

Composition is a variant of association. It represents strong "has a" relationship. Because it is a special type of association, you can use Hibernate's `OneToOne` and `OneToMany` annotations, but this assumes that dependent class is an entity (and entities can be used independently – by definition).

Hibernate provides an alternative mapping of composition – a component[16]. Component is a class annotated with `Embeddable` annotation. It is not mapped to a separate table, but shares the table with the parent entity (in case of one-to-one relationship). This reduces the number of joins needed, when querying for the parent entity, resulting in a higher performance.

When the relationship is not one-to-one, but one-to-many, you have to use `CollectionOfElements` annotation. In this case, it is not possible to share the table with the parent entity, because it would lead to denormalization of the data model. Therefore the component is mapped to a separate table. There are two main differences between regular entity and `CollectionOfElements`:

1. All operations on the parent entity are automatically cascaded to the component (insert, update and delete). When using regular entities, this behaviour have to be explicitly turned on.

16 The term component is not used here in its usual meaning.

2. Component do not have to have the identifier. Hibernate automatically creates primary key from all columns with "NOT NULL" constraint.
3. You can't use component in a FROM clause of any HQL query.

Components are ideal for implementing association classes. But there is one significant disadvantage – such relationships are always uni-directional. If this is a problem, you can always use a regular entity, but you need to take care of the primary key (composite primary key have to be used).

Ternary Associations

Ternary associations are much less common than the binary ones. It is a good practise to try to avoid them, because they make your data model difficult to understand. But sometimes they are necessary.

Ternary (in general n-ary) associations can be implemented via components. The implementation is basically the same as the implementation of an association class, but you need to add additional association to the third entity. If you mark the join column as "NOT NULL", Hibernate will include it into the primary key (it will be the foreign key at the same time).

Lets look at the example of a component represening ternary association:

```
@Embeddable
public class OrderItem {
  @Parent
  private Customer customer;

  @ManyToOne @JoinColumn(name = "product_id")
  private Product product;

  @ManyToOne @JoinColumn(name = "price_id")
  private Price price;

  … // Getters and setters
}
```

Conclusion

Hibernate does not limit the types of associations, which can be used in the domain model. It also supports composition, association classes and ternary relationships. This means, that you are able to map arbitrary object graph to the set of tables in a relational database.

3.2.4. Reference Implementation

To prove that it is possible to use Hibernate in systems with complex domain model, author of this thesis in cooperation with Profinit company have created reference implementation. It is a part of the PPR Offline application (described in section 3.1.2).

This domain model contains over ninety classes, which form weakly connected graph. It can be divided into three main groups:

- Entities for reference data

- Entities for configuration
- Entities for data – this part forms strongly connected component

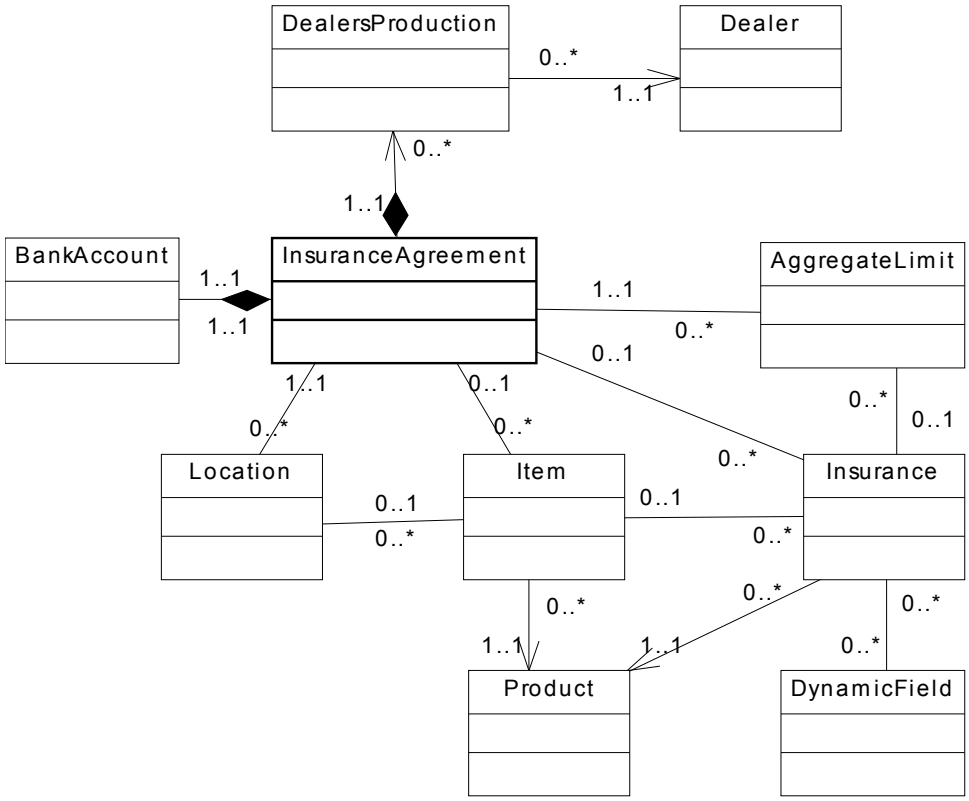


Diagram 3: Reference domain model core classes

Diagram 3 shows the most important part of the domain model. You may notice, that all types of associations are present – uni-directional, bi-directional, composition, 1:1, 1:N, 0:N and M:N.

The most important entity class is `InsuranceAgreement`, which represents insurance agreement or amendment agreement (if agreement is signed and needs to be changed, new version is created. Differences between the original and the new version are described in so called "amendment agreement").

Diagram 3 shows, that `Item` can be associated with `InsuranceAgreement` and `Location`. It can be associated with either `InsuranceAgreement` or a `Location`, to be more precise (but this constraint is not drawn in the diagram). For `Insurance` holds the simmilar constraint – it can be associated either with an `InsuranceAgreement` or with an `Item`.

Another noticable thing is, that `AggregateLimit` closes a circle(s) (`InsuranceAgreement`, `Location`, `Item`, `Insurance`, `AggregateLimit` or `InsuranceAgreement`, `Location`, `Insurance`, `AggregateLimit` or `InsuranceAgreement`, `Insurance`, `AggregateLimit`). This brings several problems, which will be discussed later in this chapter.

The way, how this object graph is manipulated is truly unique, because it fully hides the relational database used for data storage. Lets describe a simple data manipulation use case:

1. User selects the insurance agreement, he / she wants to work with. New session (instance of the class `Session`) is created, `InsuranceAgreement` and all associated entities are loaded into memory (one call to Hibernate's method `get`) and session is closed.
2. User modifies attributes of the insurance agreement (or referenced entities), adds references to the new (transient) entities and removes references to some persistent entities.
3. User saves the insurance agreement. New session is created, the whole object graph is reattached and then persisted by one call to Hibernate's method `saveOrUpdate`.

As you can see from the previous paragraphs, application loads and saves only one entity – `InsuranceAgreement`. Associated entities are loaded (saved) by the cascade. The way, how inserts, updates and deletes are cascaded is probably clear (this feature is available in the majority of todays RDBMS), but one thing might be confusing: "What happens when user removes the reference to some associated object?" Such object needs to be deleted from the database, but it is not possible to use traditional delete cascade, because these objects are not referenced from the parent object.

There are basically two ways, how to solve this problem:

1. Load the original version of the object from the database and compare it with the new version. Delete all objects, which are referenced from the old version and not from the new one. This is really bad solution – it needs a complex logic for detection of changes and it is very slow.
2. Store references to the objects, which should be deleted. This requires either creation of proxies of the entity classes or their bytecode instrumentation. Hibernate supports both ways, but proxying is used more often. This solution is much faster than previous one and gives developers finer control over the whole process (for example, when you have collection of twenty elements and you want to delete nineteen of them, it is faster to delete all elements and then insert the one, you want to keep in collection. You can force Hibernate to do so by clearing the collection and inserting one element).

Reference implementation uses the second solution via so called `DELETE_ORPHAN` cascade. Even though it is a great feature, it has some drawbacks:

- You cannot change reference to the collection, which has a `DELETE_ORPHAN` cascade. When you try to do so, Hibernate will throw an exception.
- It is not possible to have two paths to one entity, where one path have defined `INSERT` cascade and the other `DELTE` or `DELETE_ORPHAN` cascade. Lets use part of our reference domain model to describe, why this is not possible. Imagine that we want to delete an instance of the `Insurance` associated with `AggregateLimit`. We remove it from the limit's list of associated insurances and save the `InsuranceAgreement` – save will be propagated to the `AggregateLimit` due to the cascade. Hibernate will try to remove `Insurance` from the database, because it is no longer referenced from the `AggregateLimit`. But in the same time, it will try to insert it, because it is referenced from `InsuranceAgreement`. Because Hibernate

is not able to decide, whether it should be deleted or not, it will throw an exception.
The reference implementation solves this problem by removing all cascades from the `AggregateLimit – Insurance` association.

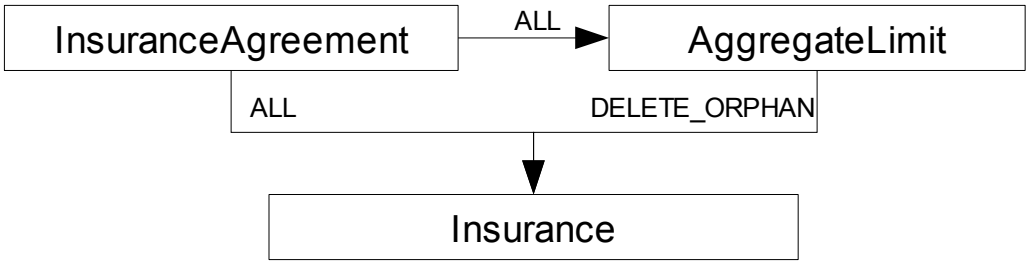


Diagram 4: Example of invalid DELETE_ORPHAN cascade.

Another very important thing is maintaining correct references in the object graph. Consider following situation: You have `InsuranceAgreement` A, which has `Insurance` 1 in its insurance collection, but `Insurance` 1 references `InsuranceAgreement` B (instead of A). What happens, when `InsuranceAgreement` A is saved? Well, there are two possible scenarios:

1. `InsuranceAgreement` B is persistent. In this case, relation between `Insurance` 1 and `InsuranceAgreement` B is saved.
2. `InsuranceAgreement` B is transient. Assuming that there is no INSERT cascade defined on the `Insurance` to `InsuranceAgreement` association, Hibernate will throw an exception.

In the reference implementation this problem do not occur, because it ensures reference integrity by special set of `addXXX` and `removeXXX` methods (`XXX` stands for name of entity, which is added or removed). These methods set back references to the correct values. They also contains some additional logic related to amendments (which is not related to Hibernate and therefore it won't be described more in depth).

Next problem is similar (and related) to the previous two. As we have written in the beginning of this section, reference domain model contains all types of associations. The most problematic is the one between the `Insurance` and the `DynamicField`, because it is M:N association. This type of association doesn't work well with DELETE and DELETE_ORPHAN cascades, because Hibernate can not detect, that object being deleted by cascade is referenced from some other object. Thus the delete operation might fail on a foreign key constraint (that is the better case. In the worst case, delete operation succeeds and database is left in inconsistent state).

Please note, that there is no way, how Hibernate can help you to solve this problem. The reference implementation uses following tricks to overcome this problem:

- Only `Insurance` instances associated with the same `Item` can share references to the instance(s) of the `DynamicField`. Therefore the DELETE cascade doesn't make any harm.
- Method `removeInsurance` defined in the class `Item` checks, whether `Insurance` being removed shares some `DynamicFields` with other instances of `Insurance` associated with the same `Item`. If such

`DynamicField` exists, method `removeInsurance` deletes the reference to it from the `Insurance` being removed. Thus DELETE_ORPHAN cascade works properly.

As you can see, solving this problem requires a large effort and therefore it is not a good idea to use "Detached objects" pattern based on cascades, when your domain model contains larger number of M:N associations.

Comparison with JDBC

PPR Offline was originally designed to be used with JDBC. Therefore we can compare the estimated effort for the JDBC solution[17] with the real effort spent on Hibernate-based implementation. Unfortunately the concrete numbers are confidential, but their comparison can be published. Lets stop beating about the bush, Hibernate-based solution required approximately the same effort as the JDBC-based estimate. This might be quite surprising, but it has a very good reason – Hibernate was used for the fist time on such a big project (in Profinit company).

According to [21] (Section 6.2), Hibernate can increase productivity by hundreds percent in comparison with JDBC. Natural question is: "Why there is such a big difference in the results between [21] and this thesis?" The answer is simple, reference implementation in [21] was created for the purpose of a measurement, whereas our reference implementation is a part of the real system.

Lets take another reference system – OpenTCM (described in section 3.1.1). The effort spent on development of the DAO layer and the domain model was approximately sixty man-days. Estimated effort for a JDBC-based solution was eighty man-days. In this case, using Hibernate saved 25% of the effort spent on the implementation of the DAO layer and the domain model.

Conclusion

Reference implementation proved, that Hibernate can be used in enterprise applications. Even though the initial effort spent on the data persistence was greater than expected, in the long run will this solution pay off, because of easier maintainability.

3.3. Performance

We have shown that Hibernate doesn't limit the design of a domain model, but what about the performance? Hibernate creates an abstraction layer over the relational databases and this brings some overhead. Is it large or small? How can performance be tuned, when you do not have the full control over SQL queries generated by Hibernate? And what about scalability, can Hibernate based applications be clustered? We will try to find answers to these question in this section.

To answer the first question, lets divide the overhead into three groups:

- Build-time
- Start-time
- Run-time

17 Developers from the Profinit company have a lot of experiences with the development of JDBC-based systems. Therefore their estimations are very accurate.

Using Hibernate doesn't affect your build cycle (in case you are not using bytecode instrumentation), thus there is no build-time overhead. To be more precise, there is no build-time overhead for the core functionality. If you are using schema generation task (hbm2dll) or some code generation tool (such as xDoclet), your build process is affected by Hibernate.

Start-time overhead is a bigger problem. It depends on the number of entities in your domain model, on the log level and the Hibernate's configuration (whether schema update or validation is turned on). Hibernate performs a lot of tasks during the startup. The most time consuming are:

- Searching for the entity classes and analysing the domain model (mapping definitions).
- Schema update (or validation). This ensures that domain model is compatible with database schema.
- Generation of static SQL statements for CRUD operations.
- Parsing named queries.

Startup of the reference implementation (see section 3.2.4) takes approximately 40 seconds on the average machine (2 GB RAM, dual core CPU) with logging level set to TRACE. With logging level set to WARN, startup takes 20 seconds.

By the runtime overhead we mean CPU time consumed by Hibernate. Memory consumption is not considered.

Hibernate uses reflection massively and it is a known fact, that reflection is slow (actually the newest versions of JVM implement reflection very efficiently, but it still brings some overhead). Good new is, that the latest versions of Hibernate use CGLIB runtime bytecode generation library, which reduces the usage of reflection.

What about scalability? Answer to this question can be found in Hibernate's documentation (citation from [22]):

“Hibernate implements an extremely high-concurrency architecture with no resource-contention issues (apart from the obvious - contention for access to the database). This architecture scales extremely well as concurrency increases in a cluster or on a single machine.”

Neither of our reference systems can prove this statement, because PPR Offline is a single user application, OpenTCM is designed for at most hundred concurrent users and Balicky doesn't use Hibernate.

Because of its architecture (sessions do not share any data), Hibernate can be clustered quite easily. One of the most popular clustering solutions, which are used together with Hibernate, is Terracotta (see [23]).

The bottleneck of nearly all modern enterprise applications is the database manipulation. SQL allows you to express one idea in many ways, but even though the resulting set of data is the same, the performance might differ a lot (in hundreds of percent). Thus the feasibility of any ORM tool highly depends on its ability to produce optimal queries.

Very good comparison of the performance of the pure JDBC and Hibernate can be found in [21] (Section 6.3). Conclusion is that Hibernate is faster than JDBC in all examined scenarios except one – loading the whole contents of the database. Unfortunately this comparison suffers from the same problem as the one described in the section 3.2.4 - it doesn't use real-world application.

Lets use reference implementation described in section 3.2.4 to examine the Hibernate's performance in the real world application.

The largest impact on the performance has the decision, whether lazy or eager fetching will be used (and where). Eager fetching prevents your code from throwing the `LazyInitializationException`, but it is much slower (performance depends on the number of eagerly fetched associations).

The reference implementation uses eager fetching (because of "Detached object" pattern). Please see the Appendix A for an example of SQL query produced by Hibernate, when it was trying to eagerly load one instance of `AggregateLimit`. As you may notice, the query contains join of more than seventy tables! It is clear that size of result set grows exponentially with the number of a join tables (except tables, which contains reference data). Author of this thesis wasn't patient enough to measure the time needed for this query to complete (it run over three days before being forcibly stopped).

Hibernate by default loads all associations lazily (they are fetched, when association is accessed for the first time). When you choose eager fetching, outer join is used by default. For large object graphs, this is not a best solution (as you can see in Appendix A). Fortunately Hibernate can be configured to use additional query to fetch required data. This increases the communication overhead, but also decrease the size of a `ResultSet` and makes it possible to use the second level cache.

Table 2 shows, how fetch type influences the time needed for the load of a single instance of `InsuranceAgreement`. Time was measured on the computer with single core CPU (Centrino), 2 GB RAM and with HDD ATA133 / 5400. As RDBMS was used Derby (version 10.3.2.1) in embedded mode. Load was repeated ten times for each configuration except the first one (fetching using outer joins). Resulting time is the arithmetic mean computed from all measured times for a concrete configuration.

Optimization	**Time - Cache turned on**	**Time - Cache turned off**
Fetching using outer joins	NA (> 3 days)	NA (> 3 days)
Collection fetched using SUBSELECT strategy	10 s	10 s
Collections and back references fetched using SUBSELECT strategy	8 s	8 s
Collections and back references fetched using SUBSELECTs, cached reference data fetched using SELECT strategy	7 s	8 s

Table 2: Time needed to load fully initialized instance of `InsuranceAgreenment`

From the Table 2 is clear that solving Cartesian product problem brings the largest performance gain. Another 20% of time can be saved by using primary cache to initialize back references.

Please note that performance gain of the secondary cache is highly influenced by the fact, that the same instance has been loaded ten times. Using the second level

cache for the reference data brings significant performance gain only when following conditions are met:

- Associations are initialized using neither outer nor inner joins, but using separate SELECTs.
- Set of values is very limited (thus it is highly probable, that cache will contain the required data).
- Cached data are in memory (not swapped to the hard drive). Although in some cases it might be advantageous to allow cache overflow to the HDD.

Unfortunately our reference implementation can't help us to measure the performance gain resulting from the batching (we always save at most one detached instance of the `InsuranceAgreement` in a single session). Therefore we can not verify the results published in [21] (Section 6.3).

3.4. Conclusion

We have shown that Hibernate doesn't put any restriction on the design of the domain model. It is ideal for the newly started projects, when there are no restrictions on the design of the domian and data model (they can be adapted to allow effective usage of Hibernate). In such cases, Hibernate can save up to 25 % of the effort spent on the implementation of a DAO layer.

By creating the reference implemenation we have proven, that it is possible to use Hibernate in systems with a complex domain model.

Hibernate allows fine performance tuning, but even the default configuration provides a satisfactory performance. When using eager fetching, it is important to carefuly pick the fetching strategy. With the wrong strategy (such as using outer joins to initialize collections) even a simple query can run for an extremly long time.

4. Common Problems And Their Solution Using Hibernate

In this chapter, we will look at the typical problems, which need to be solved in enterprise applications. We will discuss the possible solutions, trying to answer, whether using Hibernate will pay off (and under what circumstances).

Problems described in this chapter come from the real world, because this thesis was created in cooperation with Profinit company, which provided access to the reference systems (see section 3.1).

4.1. Audit Logging

Common requirement for the enterprise application is to track all changes made by users. These logs usually contain three main attributes:

- Who changed the data
- When it has been changed
- What data were changed (sometimes it is required to store the old and the new version of the data)

This problem is usually solved using database triggers. The main advantage of this solution is high performance, but there are some serious disadvantages:

- Very limited portability – triggers are usually written in the vendor-specific scripting languages (such as Oracle PLSQL)
- There must be one-to-one correspondence between application users and database users, otherwise it wouldn't be possible to save the information about a user, who changed the data. This is a huge problem mainly for web-based systems with a large number of users.
- It is not possible to define one trigger for all tables => when you need to track changes in all tables, you have to create (and maintain) a large number of triggers.

Please note, that some database systems are licensed per user and therefore the one-to-one correspondence between application users and database users highly increases TCO[18].

Here is an example of the simple trigger written in Oracle PLSQL:

```
CREATE TABLE item_log(who VARCHAR2(40), when DATE);

CREATE TRIGGER biud_item
     BEFORE INSERT OR UPDATE OR DELETE ON item
BEGIN
     INSERT INTO item_log(who, when)
          VALUES(user,sysdate);
END;
/
```

18 TCO = Total Cost of Ownership

Is it possible to implement the audit logging using Hibernate? Yes and there are several ways:

1. Explicit call to the audit logging method(s) in each DAO method (which inserts, updates or deletes any data).
2. Event listener
3. Interceptor

Lets look at these solutions more in depth.

4.1.1. Explicit Call to Logging Methods

This is probably the most intuitive solution. Its greatest advantage is that you have a full control over the logging process, but this also means a lot of repetitive code. Another big advantage is that all code can run within one transaction and therefore you do not have to take care of the audit log's and database's consistency. Main disadvantages are:

- Coupling of the DAOs and the audit logging. Changes in the logging API are difficult (or even impossible)
- Requires a lot of code.
- It is error prone. When adding a new DAO method, you have to add also call to the logging method.

Example:

```
public class ItemDao extends GenericDao<Item>
     implements IItemDao {

     private IAuditLogger auditLogger;
     private Session session;

     public void saveNewItem(Item i) {
          auditLogger.logInsert(i, getLoggedUser());
          session.save(i);
     }
}

public class AuditLogger implements IAuditLogger {

     private Session session;

     public void logInsert(DomainObject o, User u) {
          AuditLog log =
               new AuditLog(o, u, AuditLog.INSERT);
          session.save(log);
     }
}
```

4.1.2. Event Listener

This approach uses Hibernate's events system. All CRUD methods, which are defined in the class `Session`, generate one or more events. This events are passed

to the registered listeners. Listeners should be stateless, because they are shared between all sessions (for a particular session factory).

We will listen for the following events:

- PostInsertEvent
- PreUpdateEvent
- PreDeleteEvent

Please note that some identifier generating strategies (such as identity) require a database access to work properly and therefore we are listening for `PostInsertEvent` instead of `PreInsertEvent`. This will ensure, that all objects will have their identifiers initialized.

Lets look at the sample implementation of the listener for `PreInsertEvent`. To make it easier to understand, some unimportant parts were omitted.

```
public class HibernateAuditLogListener
  implements PostInsertEventListener {

  public boolean onPostInsert(PostInsertEvent event) {
    try {
      final String entityName =
        event.getEntity().getClass().toString();

      StatelessSession session = event.getPersister()
                                      .getFactory()
                                   .openStatelessSession();
      session.beginTransaction();

      AuditLog logEntry = new AuditLog(getLoggedUser(),
        entityName, AuditLog.INSERT , new Date());

      session.insert(logEntry);

      session.getTransaction().commit();
      session.close();
    } catch (HibernateException e) {...}
    return false;
  }
```

There are two importants thing to notice. First is, that we use a new session. This is necessary because event is fired during the flush, when Hibernate's data structures might be in an inconsistent state. Using a new session is not a big problem itself, but it has one very troublesome impact - audit logging code runs in a new transaction (this was the second thing, you should have noticed in the code above).

Imagine that user saves two new objects in a transaction and the second save fails. Naturally, transaction is rollbacked and the database is left untouched. Well, not this time, because first insert fired `PostInsertEvent` and therefore our listener has inserted a new row to the audit log. Because listener doesn't run in the same transaction as the code inserting the data, rollback doesn't influence it. This means that our log contains wrong data.

How to solve this problem? The simplest solution is to use the Ostrich algorithm. We won't worry about the inconsistency between the log and the database. Even though this seems as a bad solution, it is perfectly reasonable for the systems with low percentage of rollbacked transactions. We will simply let users to filter out the invalid data.

More sophisticated solution is to slightly change the transaction manager, which is being used. It should be able to rollback transaction covering logging code, whenever the main transaction (i.e. transaction in which runs your DAO method) was rollbacked. But this brings another problem – because listener is stateless (it is defacto singleton), it can be wrapped in at most one transaction. If there are two threads calling two DAO methods (in two independent transactions) and one transaction is rollbacked and the second committed, it causes that the audit logging transaction will be rollbacked. Thus our log will not contain information about changes made by the committed transaction.

Problem described in the previous paragraph can be solved by using the `ThreadLocal` variable holding the audit log transaction (this way we can add state to our stateless listeners).

Last, but not least problem is with the update of detached objects. If the domain object is modified and saved when the session is oppened, Hibernate is able to detect all changes (and tell us, which attributes have changed). But this information is lost when the session is closed. Thus when we need to log the old and the new value of the object being updated, we need to load the old value from the database. Alternative solution is to merge detached instances to the current session before updating them (this have to be done in each DAO method, which updates the data). Merging should be preferred because it doesn't require additional SELECTs and therefore it has not such a big impact on the performance.

Lets summarize advantages and disadvantages of the event listener approach. Advantages are:

- Number SLOC[19] needed to implement this functionality is not correlated with size of a domain or data model.
- Logging is orthogonal to the rest of the application. All audit logging code is in one place.
- It is not error-prone (once it has been debugged)

Disadvantages:

- A lot of problems needs to be solved
- Requires modification of a transaction manager
- Requires a new session, which implies a new database connection.

4.1.3. Interceptor

An alternative to the observer pattern (event listeners) used in the previous solution is the AOP approach. Hibernate allows you to register code, which is called whenever some data-related event occurs. Interceptors are usually used for validation of entities being persisted (or updated).

19 SLOC = Source Line Of Code

Interceptors can be either session-coped or session factory scoped (i.e. global). Global interceptors have to be thread-safe and can't store session-specific state, since multiple sessions will use this interceptor concurrently.

We will use a session-scoped interceptor to implement the audit logging, because it can be statefull and we do not have to take care of the thread safety. Below, you can find an example of how it could be implemented (some unimportant parts of the code were omitted):

```
public class HibernateAuditLogInterceptor
  extends EmptyInterceptor {

  private Set inserts = new HashSet();

  public boolean onSave(Object obj, Serializable id,
    Object[] newValues, String[] properties, Type[] types)
    throws CallbackException {

    if (obj instanceof Auditable) {
      try {
        Class objectClass = obj.Class();
        String className = objectClass.getSimpleName();

        AuditLog logEntry = new AuditLog(getLoggedUser(),
          entityName, AuditLog.INSERT , new Date());

        inserts.add(logEntry);
      } catch (Exception e) { ... }
    }
  return false;
  }

  public void afterTransactionCompletion(Transaction tx) {
    if ( tx.wasCommitted() ) {
      Session session = sessionFactory.openSession();
      try {
        Iterator iter = inserts.iterator();
        while (iter.hasNext()) {
          AuditLog logEntry = (AuditLog) iter.next();
          session.save(logRecord);
        }
      } catch (HibernateException e) {
        throw new CallbackException(e);
      } finally {
        inserts.clear();
        session.flush();
        session.close();
      }
    }
  }
}
```

This code is even more flexible than the listener described in the previous section, because it allows you to define, which domain objects should be logged (they have to implement `Auditable` interface, which is empty and thus servers only as a marker). On the other hand, the previous solution can be easily extended to support this functionality too. If you are using Java 1.5 and higher, you may use annotations instead of a marker interface.

Another important thing to notice is that we do not save the log entry immediately after we receive information about the log event (call of a callback method), but we wait until the transaction commits. This is an elegant way, how to ensure consistency of the audit log and the database (it goes around the problem with transactions, which was described in the previous section). It also slightly increases performance, because database updates can be batched.

The only problem occurs, when system crashes after the original transaction commits, but before all audit log entries are saved. This will mean that the log will not contain all required data. This situation is rare, but if you are creating a highly reliable system, you have to solve this problem. We can reuse the solution from the previous section – log entries will be saved immediately. Logging code will run in a separate transaction, which will be handled by the interceptor. This means that the new transaction will be started in the `afterTransactionBegin` method and committed or rollbacked in the `afterTransactionCompletion` method. Because each session has its own instance of the interceptor, we do not have to use a `ThreadLocal` variable for storing the reference to the active transaction.

The problem with detecting update of detached objects (as it has been described in the previous section) applies to the interceptor approach too. You can use the same solution (i.e. load the old value from the database or use merge operation).

Lets summarize advantages and disadvantages of the interceptor approach. Advantages are:

- Number SLOC needed to implement this functionality is not correlated with size of a domain or data model.
- Logging is orthogonal to the rest of the application. All audit logging code is in one place.
- It is not error-prone (once it has been debugged)
- You do not have to modify your transaction manager.

Disadvantages:

- Many problems needs to be solved
- Requires a new session, which implies a new database connection.

4.1.4. Conclusion

	Explicit call	Event listener	Interceptor
Number of SLOC	Large	Small	Small
Requires second Session	No	Yes	Yes
Error-prone	Yes	No	No
Hibernate-specific	No	Yes	No (JPA Callback)

Table 3: Comparison of the audit log implemenations

The interceptor approach seems to be the best solution to the audit logging problem, because it doesn't require so much coding as explicite call to the logging methods and it allows you to solve the problem with consistency in a more elegant way than the event listener approach. But for some applications it might be better to use some other approach (such as “explicite call to the logging methods” in systems with only a few DAO methods).

Using Hibernate instead of triggers brings following benefits:

- Log can contain information about the application user, not the database user.
- System can have only one (or a very limited number) database user. When database system is licensed per user, this will mean significant reduction of costs.
- When using interceptor or event listener approach, the amount of code needed to implement this functionality doesn't depend on the number of tables or entities being logged.
- It can be used with database systems, which do not support triggers (some embedded databases).
- Higher portability.

The only significant disadvantage is the performance. There is a high communication overhead and in some cases, the number of executed statements is also higher (for example when you use additional SELECT when logging information about update).

4.2. Temporal Data

Another common requirement for an enterprise system is the ability to work with temporal data. Imagine an insurance company, which decides to change the price of its products. This change shouldn't affect current customers (even if they want to change the set or risks, covered by their insurance) and therefore the information system in this company have to compute insurance payments for new and old customers differently.

This problem is usually solved by adding additional condition to each query, which uses tables that store temporal data. But this is a lot of work and it is quite

error prone (it is easy to forget to add this condition to some SQL query). Another problem occurs, when you need queries without restriction on temporal columns. It is easy to solve this problem (simply create two versions of a query – with and without restriction on the temporal column), but it doubles the number of queries, which you need to write and maintain.

The biggest advantage of this solution is performance. Each query can be carefully tuned.

Alternative solution is based on loading all data and filtering them by hand. If all temporal domain objects implements some particular interface (for example `ITemporal`), you will need only one simple peace of code to implement filtering logic:

```
public class TemporalFilter {
  public void filter(Collection<ITemporal> collection,
                     Date date) {

    Iterator<ITemporal> iter = collection.iterator();
    while (iter.hasNext()) {
      ITemporal elem = iter.next();
      if (!(date.after(elem.getFrom()) &&
            date.before(elem.getTo() )) {
        iter.remove();
      }
    }
  }
}
```

The disadvantage of this approach is obvious – performance. A lot of unnecessary data might be loaded. This solution therefore makes sense only when performance is not an issue or when it is highly probable, that there won't be much expired data.

Can Hibernate help us to solve this problem? It is clear that both solutions described above works with Hibernate too, but it doesn't bring any significant benefit. Lets describe another solution, which uses Hibernate-specific feature called “Filters“.

Filter is a named restriction, which can be applied when querying for specified entities. Filters can be dynamically turned on and off (for the current Session), which allows you to have more filters for one entity. Filtering of associations is also possible. For the basic introduction to the filter usage please see [1] (Chapter 17).

Even though Hibernate has its own query language, which supports restrictions, Hibernate filters use pure SQL. But you can use named parameters in the same way, as they are used in HQL.

Hibernate-based implementation of a temporal filter is really straightforward, as you can see in the following example:

```
@FilterDef(name="pricing_filter",
  parameters=@ParamDef( name="when", type="date" ) )
@Filter(name="pricing_filter",
  condition="date_from <= :when and :when <= date_to")
public class Pricing implements Serializable {
  private Long dbid;     // primary key

  @Temporal(TemporalType.DATE)
  @Column(name = "date_to")
  private Date dateFrom;

  @Temporal(TemporalType.DATE)
  @Column(name = "date_from")
  private Date dateTo;

  private BigDecimal price;
}
```

Please notice that for each filter you need two annotations. `FilterDef` specifies name(s) and type(s) of the filter's parameter(s). `Filter` contains definition of a restriction.

Big disadvantage of Hibernate filters is the fact, that they affect only one entity. Therefore you have to define one filter for each temporal entity and association in your domain model.

If you want to increase maintainability and readability of your source codes, you should obey this rules (whenever it is possible):

- Temporal columns (start and end date) should have same name in all tables. This can be ensured by defining superclass of all entities containing temporal data (use `@MappedSuperclass` annotation).
- Naming of filters should be consistent. Good idea is to prefix the name of each filter with the name of an entity, for which is filter defined. But do not use dots in filter name (Hibernate doesn't like them)!
- Try to keep `FilterDef` and `Filter` annotations together (it is not possible, when you are using collection filters). `FilterDef` can be defined at package level, but do not use this feature.
- Move restriction string into a constant (`public static final String` attribute). Use this constant in all temporal filter definitions. Please note, that this constant can be used only when temporal columns in all tables have the same name.
- Use named parameters.

Compared to the previously described solutions of the temporal data problem, Hibernate filters have following advantages:

- Higher performance than `TemporalFilter` approach.
- Requires less coding than the first approach.
- High maintainability.
- They are flexible (you can dynamically turn filters on and off)

It has no significant disadvantage and therefore we can declare it as a best solution (from the described ones) to the temporal data problem.

5. Conclusion

5.1. Future Work

This thesis have touched many interesting topic – both practical and theoretical, but some of them went beyond its scope.

In the theoretical part, it would be nice to define a subset HQL_{RA} of the Hibernate Query Language and to prove that relational algebra without union, intersection and difference has the same expressive power as HQL_{RA}. With this knowledge it would be possible to examine, whether it is feasible to use relational optimizations together with HQL and whether it increases the performance.

Hibernate contains another query language - Criteria API. It can be interesting to compare expressive power of HQL, Criteria API and RA.

Criteria API can be also compared with HQL in the term of usability and performance.

The original topic of this thesis (trying to use Hibernate in huge enterprise system) is also interesting, but is very work-intensive.

Last, but not least, the chapter 4 describes only a very limited set of problems. It would be useful to examine others.

5.2. Evaluation of Goals

In the first chapter we have defined four goals. Now it is time to evaluate, whether this goals were accomplished.

Chapter one shortly described, which problems needs to be solved by any object-relational mapping tool (“Impedance mismatch“ problem). It provided a brief introduction to the Hibernate framework and explained, why Hibernate has been chosen. It also shortly described several other ORM tools. Even though first chapter doesn't help us to accomplish any goal, it provided a necessary introduction to the problematic of the object-relational mapping.

Second chapter provided a brief introduction to HQL and discussed, whether the relational algebra can be translated into HQL. It defined a relational algebra without union, intersection and difference and formally proved, that it can be translated into HQL. Thus the first goal is accomplished.

Third chapter has shown, that Hibernate doesn't put any restriction on the design of the domain model. It described the reference implementation of the complex domain model, which was created to demonstrate this finding. Thus the second goal is accomplished.

Third chapter also discussed, how Hibernate influences application's performance. It has shown, that using Hibernate results in a longer startup time. Reference implementation was used to demonstrate the negative impacts of the eager fetching on the performance and it has shown several ways, how these impacts can be minimalised. Third goal has been therefore accomplished.

Chapter four described two real world problems, which Hibernate can help you to solve. Several solutions were provided for each problem together with their advantages and disadvantages. It has been shown, that Hibernate-based solution

usually requires less coding than the traditional one and it is easier to maintain. Chapter four helped us to accomplish the last (fourth) goal.

All goals were accomplished.

5.3. Recap

- HQL is not stronger than RA, because it doesn't contain union, intersection and difference.
- HQL is stronger than RA without union, intersection and difference.
- Hibernate doesn't put any restriction on the design of the domain model. Although adapting the design to Hibernate can decrease the total effort.
- It is possible to use Hibernate in systems with a complex data model.
- Eager fetching can significantly decrease performance. Selecting right fetching strategy can reduce negative impacts on the performance.
- Fetching strategy influences performance more than second level cache.
- Audit logging can be implemented via Hibernate.

Literature

[1] Hibernate documentation http://www.hibernate.org/hib_docs/v3/reference/en/
[2] Christian Bauer, Gavin King: **Java Persistence With Hibernate**, Manning 2007
[3] Jaroslav Orság: Object-Relational Mapping, (2006)
[4] Lubor Šubčík: Moderní způsoby zajištění persistence dat u J2EE aplikací, (2006)
[5] Arnošt valíček: Objektově-relační mapování v Javě, (2007)
[6] Introduction to the JDBC
http://java.sun.com/javase/technologies/database/index.jsp
[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison-Wesley 1995
[8] Johnson, R., et al.: **Professional Java development with the Spring Framework**, Wrox 2005
[9] Spring Reference Documentation
http://www.springframework.org/docs/reference/jdbc.html
[10] Christian Bauer, Gavin King: **Hibernate in Action**, Manning 2004
[11] James Elliott: **Hibernate: A Developer's Notebook**, O'Reilly 2004
[12] E. Pugh, J. D. Gradecki: **Professional Hibernate**, Wrox 2004
[13] Dave Minter, Jeff Linwood: **Pro Hibernate 3**, Apress 2005
[14] Dave Minter, Jeff Linwood: **Beginning Hibernate: From Novice to Professional**, Apress 2006
[15] Patrick Peak, Nick Heudecker: **Hibernate Quickly**, Manning 2005
[16] Hibernate Annotations http://www.hibernate.org/hib_docs/annotations/
[17] Hibernate EntityManager
http://www.hibernate.org/hib_docs/entitymanager/reference/en/html_single/
[18] Wikipedia: Relational algebra http://en.wikipedia.org/wiki/Relational_algebra
[19] Alain Pirrote:**A Precise Definition of Basic Relational Notions and of The Relational Agebra**,1982
[20] SQL92 Standard http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1
[21] Michal Pravda: Výhody a nevýhody použití perzistence objektů v jazyce Java, MFF UK (2007)
[22] Performance Q&A http://www.hibernate.org/15.html
[23] Clustering Hibernate with Terracotta
http://www.terracotta.org/confluence/display/orgsite/Hibernate

Terms and abbreviations

CRUD	– Create, Retrieve, Update, Delete
DI	– Depency Injection
HQL	– Hibernate Query Language
JDBC	– Java Database Access.
JVM	– Java Virtual Machine
MD	– Man-Day
OOP	– Object Oriented Programming
ORM	– Object-Relational Mapping
POJO	– Plain Old Java Object
RA	– Relational Algebra
RDBMS	– Relational DataBase Management System
SLOC	– Source Line Of Code
TOC	– Total Cost of Ownership

Diagrams

Tables

Definitions

Appendix A: Example of a SQL query generated by Hibernate

```
select souhrnnyli0_.id as id9_62_, souhrnnyli0_.datum_pocatku_platnosti as
datum2_9_62_, souhrnnyli0_.datum_storna as datum3_9_62_, souhrnnyli0_.id_limitu as
id4_9_62_, souhrnnyli0_.nazev as nazev9_62_, souhrnnyli0_.smlouva as smlouva9_62_,
nazevlimit1_.klic as klic23_0_, nazevlimit1_.hodnota as hodnota23_0_,
pojisteni2_.souhrnny_limit as souhrnny8_64_, pojisteni2_.id as id64_, pojisteni2_.id
as id4_1_, pojisteni2_.cislo_dodatku_storna as cislo2_4_1_, pojisteni2_.datum_storna
as datum3_4_1_, pojisteni2_.platnost_od as platnost4_4_1_, pojisteni2_.id_pojisteni
as id5_4_1_, pojisteni2_.max_pojistne_plneni as max6_4_1_, pojisteni2_.nazev_id as
nazev14_4_1_, pojisteni2_.predmet_id as predmet11_4_1_, pojisteni2_.sazebnik_id as
sazebnik13_4_1_, pojisteni2_.sleva as sleva4_1_, pojisteni2_.smlouva_id as
smlouva10_4_1_, pojisteni2_.souhrnny_limit as souhrnny8_4_1_,
pojisteni2_.spoluucast_id as spoluucast12_4_1_, pojisteni2_.typ as typ4_1_,
nazevpojis3_.klic as klic25_2_, nazevpojis3_.hodnota as hodnota25_2_,
policka4_.pojisteni_id as pojisteni1_65_, policko5_.id as policko2_65_, policko5_.id
as id6_3_, policko5_.ciselna_hodnota as ciselna2_6_3_, policko5_.policko as
policko6_3_, policko5_.textova_hodnota as textova3_6_3_, policko5_.vyctova_hodnota as
vyctova4_6_3_, cpolicko6_.id as id48_4_, cpolicko6_.typ as typ48_4_,
hodnoty7_.policko as policko66_, hodnoty7_.klic as klic66_, hodnoty7_.klic as
klic43_5_, hodnoty7_.hodnota as hodnota43_5_, hodnoty7_.policko as policko43_5_,
chodnotapo8_.klic as klic43_6_, chodnotapo8_.hodnota as hodnota43_6_,
chodnotapo8_.policko as policko43_6_, predmet9_.id as id7_7_,
predmet9_.cislo_dodatku_storna as cislo2_7_7_, predmet9_.datum_storna as datum3_7_7_,
predmet9_.platnost_od as platnost4_7_7_, predmet9_.predmet_id as predmet5_7_7_,
predmet9_.misto_id as misto9_7_7_, predmet9_.nazev_predmetu as nazev11_7_7_,
predmet9_.sazebnik as sazebnik7_7_, predmet9_.smlouva_id as smlouva7_7_7_,
predmet9_.specifikace_predmetu as specifik6_7_7_, predmet9_.typ_pojistne_hodnoty as
typ10_7_7_, predmet9_.typ_predmetu as typ8_7_7_, predmet9_.vlastnictvi_predmetu as
vlastni12_7_7_, mistopojis10_.id as id1_8_, mistopojis10_.cislo_dodatku_storna as
cislo2_1_8_, mistopojis10_.datum_storna as datum3_1_8_, mistopojis10_.platnost_od as
platnost4_1_8_, mistopojis10_.misto_id as misto5_1_8_, mistopojis10_.cinnost_id as
cinnost9_1_8_, mistopojis10_.popis as popis1_8_, mistopojis10_.zona_id as zona7_1_8_,
mistopojis10_.smlouva_id as smlouva8_1_8_, adresy11_.misto_pojisteni as misto6_67_,
adresy11_.id as id67_, adresy11_.id as id0_9_, adresy11_.psc as psc0_9_,
adresy11_.ulice as ulice0_9_, adresy11_.adresa_id as adresa5_0_9_,
adresy11_.misto_pojisteni as misto6_0_9_, podnikatel12_.klic as klic21_10_,
podnikatel12_.nazev_cinnosti as nazev2_21_10_, podnikatel12_.odvetvi as
odvetvi21_10_, podnikatel13_.klic as klic22_11_, podnikatel13_.nazev_odvetvi as
nazev2_22_11_, rizikovapo14_.klic as klic26_12_, rizikovapo14_.hodnota as
hodnota26_12_, rizikovapo14_.platnost_do as platnost3_26_12_,
rizikovapo14_.platnost_od as platnost4_26_12_, pojistnasm15_.id as id5_13_,
pojistnasm15_.cetnost_placeni as cetnost14_5_13_, pojistnasm15_.cislo_dodatku as
cislo2_5_13_, pojistnasm15_.cislo_navrhu as cislo3_5_13_, pojistnasm15_.cislo_ps as
cislo4_5_13_, pojistnasm15_.konec_platnosti as konec5_5_13_,
pojistnasm15_.pocatek_platnosti as pocatek6_5_13_, pojistnasm15_.datum_uzavreni as
datum7_5_13_, pojistnasm15_.korespondence as korespo18_5_13_,
pojistnasm15_.pobocka_produkce as pobocka16_5_13_, pojistnasm15_.popis as popis5_13_,
pojistnasm15_.sleva as sleva5_13_, pojistnasm15_.stav as stav5_13_,
pojistnasm15_.typ_pojistneho as typ17_5_13_, pojistnasm15_.cislo_uctu_pojistnika as
cislo10_5_13_, pojistnasm15_.kod_banky as kod15_5_13_,
pojistnasm15_.predcisli_uctu_pojistnika as predcisli11_5_13_,
pojistnasm15_.specificky_symbol_uctu_pojistnika as specificky12_5_13_,
pojistnasm15_.vysledna_pml as vysledna13_5_13_, cetnostpla16_.klic as klic15_14_,
cetnostpla16_.hodnota as hodnota15_14_, koresponde17_.klic as klic20_15_,
koresponde17_.hodnota as hodnota20_15_, osoby18_.smlouva_id as smlouva11_68_,
osoby18_.id as id68_, osoby18_.id as id3_16_, osoby18_.adresa_id as adresa10_3_16_,
osoby18_.koresp_adresa_id as koresp14_3_16_, osoby18_.cislo_pasu as cislo2_3_16_,
osoby18_.evidencni_vypis as evidencni3_3_16_, osoby18_.funkce as funkce3_16_,
osoby18_.ico as ico3_16_, osoby18_.jmeno as jmeno3_16_, osoby18_.nazev_firmy as
nazev7_3_16_, osoby18_.prijmeni as prijmeni3_16_, osoby18_.rodne_cislo as
rodne9_3_16_, osoby18_.role as role3_16_, osoby18_.smlouva_id as smlouva11_3_16_,
osoby18_.statni_prislusnost as statni15_3_16_, osoby18_.titul_id as titul13_3_16_,
osoby18_.typ as typ3_16_, adresa19_.id as id0_17_, adresa19_.psc as psc0_17_,
adresa19_.ulice as ulice0_17_, adresa19_.adresa_id as adresa5_0_17_,
adresa19_.misto_pojisteni as misto6_0_17_, adresa19_.typ as typ0_17_, adresa20_.id as
id0_18_, adresa20_.psc as psc0_18_, adresa20_.ulice as ulice0_18_,
adresa20_.adresa_id as adresa5_0_18_, adresa20_.misto_pojisteni as misto6_0_18_,
adresa20_.typ as typ0_18_, roleosoby21_.klic as klic27_19_, roleosoby21_.hodnota as
hodnota27_19_, statnipris22_.klic as klic29_20_, statnipris22_.hodnota as
hodnota29_20_, titul23_.klic as klic31_21_, titul23_.hodnota as hodnota31_21_,
typosoby24_.klic as klic33_22_, typosoby24_.hodnota as hodnota33_22_,
cislopoboc25_.klic as klic16_23_, cislopoboc25_.hodnota as hodnota16_23_,
```

```
pojisteni26_.smlouva_id as smlouva10_69_, pojisteni26_.id as id69_, pojisteni26_.id
as id4_24_, pojisteni26_.cislo_dodatku_storna as cislo2_4_24_,
pojisteni26_.datum_storna as datum3_4_24_, pojisteni26_.platnost_od as
platnost4_4_24_, pojisteni26_.id_pojisteni as id5_4_24_,
pojisteni26_.max_pojistne_plneni as max6_4_24_, pojisteni26_.nazev_id as
nazev14_4_24_, pojisteni26_.predmet_id as predmet11_4_24_, pojisteni26_.sazebnik_id
as sazebnik13_4_24_, pojisteni26_.sleva as sleva4_24_, pojisteni26_.smlouva_id as
smlouva10_4_24_, pojisteni26_.souhrnny_limit as souhrnny8_4_24_,
pojisteni26_.spoluucast_id as spoluucast12_4_24_, pojisteni26_.typ as typ4_24_,
csazebnik27_.id as id50_25_, csazebnik27_.nazev as nazev50_25_,
csazebnik27_.platnost_do as platnost3_50_25_, csazebnik27_.platnost_od as
platnost4_50_25_, predmetyap28_.sazebnik as sazebnik70_, predmetyap28_.platnost_do as
platnost2_70_, predmetyap28_.platnost_od as platnost3_70_, predmetyap28_.pojisteni as
pojisteni70_, predmetyap28_.predmet as predmet70_, cpojisteni29_.id as id47_26_,
cpojisteni29_.nazev as nazev47_26_, konverze30_.pojisteni_id as pojisteni5_71_,
konverze30_.id as id71_, konverze30_.id as id44_27_, konverze30_.id_oj as id2_44_27_,
konverze30_.platnost_do as platnost3_44_27_, konverze30_.platnost_od as
platnost4_44_27_, konverze30_.pojisteni_id as pojisteni5_44_27_,
okamzikyaa31_.pojisteni_id as pojisteni1_72_, okamzikyaa31_.akce_id as akce2_72_,
okamzikyaa31_.okamzik_id as okamzik3_72_, okamzikyaa31_.poradi as poradi72_,
okamzikyaa31_.platnost_do as platnost5_72_, okamzikyaa31_.platnost_od as
platnost6_72_, cakce32_.id as id41_28_, cakce32_.java_program as java2_41_28_,
cokamzik33_.id as id45_29_, cokamzik33_.nazev as nazev45_29_, policka34_.pojisteni_id
as pojisteni1_73_, policka34_.platnost_do as platnost2_73_, policka34_.platnost_od as
platnost3_73_, policka34_.policko_id as policko4_73_, cpolicko35_.id as id48_30_,
cpolicko35_.typ as typ48_30_, souhrnneli36_.pojisteni_id as pojisteni1_74_,
csouhrnnyl37_.id as souhrnne2_74_, csouhrnnyl37_.id as id51_31_, csouhrnnyl37_.nazev
as nazev51_31_, cpredmet38_.id as id49_32_, cpredmet38_.misto_poj_povinne as
misto2_49_32_, cpredmet38_.nazev as nazev49_32_, cpredmet38_.typ as typ49_32_,
typpredmet39_.klic as klic37_33_, typpredmet39_.hodnota as hodnota37_33_,
spoluucast40_.klic as klic28_34_, spoluucast40_.hodnota as hodnota28_34_,
ctyppojist41_.klic as klic34_35_, ctyppojist41_.hodnota as hodnota34_35_,
otazky42_.pojisteni_id as pojisteni1_75_, otazky42_.otazka_id as otazka2_75_,
otazky42_.platnost_do as platnost3_75_, otazky42_.platnost_od as platnost4_75_,
cotazka43_.id as id46_36_, cotazka43_.poradi as poradi46_36_, cotazka43_.typ as
typ46_36_, zavisina44_.pojisteni_id as pojisteni1_76_, zavisina44_.master_id as
master2_76_, zavisina44_.pozadovany_vysledek as pozadovany3_76_, cotazka45_.id as
id46_37_, cotazka45_.poradi as poradi46_37_, cotazka45_.typ as typ46_37_,
predmety46_.smlouva_id as smlouva7_77_, predmety46_.id as id77_, predmety46_.id as
id7_38_, predmety46_.cislo_dodatku_storna as cislo2_7_38_, predmety46_.datum_storna
as datum3_7_38_, predmety46_.platnost_od as platnost4_7_38_, predmety46_.predmet_id
as predmet5_7_38_, predmety46_.misto_id as misto9_7_38_, predmety46_.nazev_predmetu
as nazev11_7_38_, predmety46_.sazebnik as sazebnik7_38_, predmety46_.smlouva_id as
smlouva7_7_38_, predmety46_.specifikace_predmetu as specifik6_7_38_,
predmety46_.typ_pojistne_hodnoty as typ10_7_38_, predmety46_.typ_predmetu as
typ8_7_38_, predmety46_.vlastnictvi_predmetu as vlastni12_7_38_, nazevpredm47_.klic
as klic24_39_, nazevpredm47_.hodnota as hodnota24_39_, nemovitost48_.predmet as
predmet78_, nemovitost48_.id as id78_, nemovitost48_.id as id2_40_,
nemovitost48_.cena as cena2_40_, nemovitost48_.index as index2_40_,
nemovitost48_.predmet as predmet2_40_, nemovitost48_.psc as psc2_40_,
nemovitost48_.ulice as ulice2_40_, nemovitost48_.specifikace as specifik7_2_40_,
policka49_.predmet_id as predmet1_79_, policko50_.id as policko2_79_, policko50_.id
as id6_41_, policko50_.ciselna_hodnota as ciselna2_6_41_, policko50_.policko as
policko6_41_, policko50_.textova_hodnota as textova3_6_41_,
policko50_.vyctova_hodnota as vyctova4_6_41_, csazebnik51_.id as id50_42_,
csazebnik51_.nazev as nazev50_42_, csazebnik51_.platnost_do as platnost3_50_42_,
csazebnik51_.platnost_od as platnost4_50_42_, typpojistn52_.klic as klic36_43_,
typpojistn52_.hodnota as hodnota36_43_, typpredmet53_.klic as klic37_44_,
typpredmet53_.hodnota as hodnota37_44_, vlastnictv54_.klic as klic39_45_,
vlastnictv54_.hodnota as hodnota39_45_, vozidla55_.predmet as predmet80_,
vozidla55_.vozidlo_id as vozidlo1_80_, vozidla55_.vozidlo_id as vozidlo1_12_46_,
vozidla55_.cena as cena12_46_, vozidla55_.index as index12_46_, vozidla55_.predmet as
predmet12_46_, vozidla55_.cena_nova as cena4_12_46_, vozidla55_.cislo_karoserie as
cislo5_12_46_, vozidla55_.druh as druh12_46_, vozidla55_.registracni_znacka as
registra6_12_46_, vozidla55_.rok_vyroby as rok7_12_46_, vozidla55_.typ_provedeni as
typ9_12_46_, vozidla55_.znacka as znacka12_46_, druhvozidl56_.klic as klic18_47_,
druhvozidl56_.hodnota as hodnota18_47_, typprovede57_.klic as klic38_48_,
typprovede57_.hodnota as hodnota38_48_, znackavozi58_.klic as klic40_49_,
znackavozi58_.hodnota as hodnota40_49_, zarizeni59_.predmet as predmet81_,
zarizeni59_.zarizeni_id as zarizeni1_81_, zarizeni59_.zarizeni_id as
zarizeni1_13_50_, zarizeni59_.cena as cena13_50_, zarizeni59_.index as index13_50_,
zarizeni59_.predmet as predmet13_50_, zarizeni59_.rok_vyroby as rok4_13_50_,
zarizeni59_.specifikace as specifik5_13_50_, zarizeni59_.typ as typ13_50_,
zarizeni59_.vyrobni_cislo as vyrobni7_13_50_, prilohy60_.smlouva_id as smlouva4_82_,
prilohy60_.id as id82_, prilohy60_.id as id8_51_, prilohy60_.druh as druh8_51_,
prilohy60_.id_prilohy as id2_8_51_, prilohy60_.pocet_stran as pocet3_8_51_,
prilohy60_.smlouva_id as smlouva4_8_51_, druhpriloh61_.klic as klic17_52_,
druhpriloh61_.hodnota as hodnota17_52_, souhrnneli62_.smlouva as smlouva83_,
```

```
souhrnneli62_.id as id83_, souhrnneli62_.id as id9_53_,
souhrnneli62_.datum_pocatku_platnosti as datum2_9_53_, souhrnneli62_.datum_storna as
datum3_9_53_, souhrnneli62_.id_limitu as id4_9_53_, souhrnneli62_.nazev as
nazev9_53_, souhrnneli62_.smlouva as smlouva9_53_, specialniu63_.smlouva_id as
smlouva3_84_, specialniu63_.id as id84_, specialniu63_.id as id10_54_,
specialniu63_.smlouva_id as smlouva3_10_54_, specialniu63_.text as text10_54_,
spravci64_.smlouva_id as smlouva1_85_, spravce65_.id as spravce2_85_, spravce65_.id
as id11_55_, spravce65_.cislo_pobocky as cislo8_11_55_, spravce65_.cislo_spravce as
cislo2_11_55_, spravce65_.email as email11_55_, spravce65_.jmeno as jmeno11_55_,
spravce65_.osobni_cislo as osobni5_11_55_, spravce65_.prijmeni as prijmeni11_55_,
spravce65_.telefon as telefon11_55_, cislopoboc66_.klic as klic16_56_,
cislopoboc66_.hodnota as hodnota16_56_, stavsmlouv67_.klic as klic30_57_,
stavsmlouv67_.hodnota as hodnota30_57_, typpojistn68_.klic as klic35_58_,
typpojistn68_.hodnota as hodnota35_58_, kodbanky69_.klic as klic19_59_,
kodbanky69_.kod_banky as kod2_19_59_, kodbanky69_.nazev_banky as nazev3_19_59_,
ziskatele70_.smlouva_id as smlouva10_86_, ziskatele70_.id as id86_, ziskatele70_.id
as id14_60_, ziskatele70_.cislo_pobocky as cislo9_14_60_,
ziskatele70_.cislo_ziskatele as cislo2_14_60_, ziskatele70_.jmeno as jmeno14_60_,
ziskatele70_.nazev as nazev14_60_, ziskatele70_.osobni_cislo as osobni5_14_60_,
ziskatele70_.podil_produkce as podi16_14_60_, ziskatele70_.podil_provize as
podil7_14_60_, ziskatele70_.prijmeni as prijmeni14_60_, ziskatele70_.smlouva_id as
smlouva10_14_60_, cislopoboc71_.klic as klic16_61_, cislopoboc71_.hodnota as
hodnota16_61_
   from d_souhrnny_limit_ppr souhrnnyli0_ left outer join c_nazev_limitu_ppr
nazevlimit1_ on souhrnnyli0_.nazev=nazevlimit1_.klic left outer join d_pojisteni_ppr
pojisteni2_ on souhrnnyli0_.id=pojisteni2_.souhrnny_limit left outer join
c_nazev_pojisteni_ppr nazevpojis3_ on pojisteni2_.nazev_id=nazevpojis3_.klic left
outer join d_pojisteni_policko_ppr policka4_ on pojisteni2_.id=policka4_.pojisteni_id
left outer join d_policko_ppr policko5_ on policka4_.policko_id=policko5_.id left
outer join c_policko_ppr cpolicko6_ on policko5_.policko=cpolicko6_.id left outer
join c_policko_hodnota_ppr hodnoty7_ on cpolicko6_.id=hodnoty7_.policko left outer
join c_policko_hodnota_ppr chodnotapo8_ on
policko5_.vyctova_hodnota=chodnotapo8_.klic left outer join d_predmet_ppr predmet9_
on pojisteni2_.predmet_id=predmet9_.id left outer join d_misto_pojisteni_ppr
mistopojis10_ on predmet9_.misto_id=mistopojis10_.id left outer join d_adresa_ppr
adresy11_ on mistopojis10_.id=adresy11_.misto_pojisteni left outer join
c_podnikatelska_cinnost_ppr podnikatel12_ on
mistopojis10_.cinnost_id=podnikatel12_.klic left outer join
c_podnikatelske_odvetvi_ppr podnikatel13_ on podnikatel12_.odvetvi=podnikatel13_.klic
left outer join c_rizikova_povodnova_zona_ppr rizikovapo14_ on
mistopojis10_.zona_id=rizikovapo14_.klic left outer join d_pojistna_smlouva_ppr
pojistnasm15_ on mistopojis10_.smlouva_id=pojistnasm15_.id left outer join
c_cetnost_placeni_ppr cetnostpla16_ on
pojistnasm15_.cetnost_placeni=cetnostpla16_.klic left outer join c_korespondence_ppr
koresponde17_ on pojistnasm15_.korespondence=koresponde17_.klic left outer join
d_osoba_ppr osoby18_ on pojistnasm15_.id=osoby18_.smlouva_id left outer join
d_adresa_ppr adresa19_ on osoby18_.adresa_id=adresa19_.id left outer join
d_adresa_ppr adresa20_ on osoby18_.koresp_adresa_id=adresa20_.id left outer join
c_role_osoby_ppr roleosoby21_ on osoby18_.role=roleosoby21_.klic left outer join
c_statni_prislusnost_ppr statnipris22_ on
osoby18_.statni_prislusnost=statnipris22_.klic left outer join c_titul_ppr titul23_
on osoby18_.titul_id=titul23_.klic left outer join c_typ_osoby_ppr typosoby24_ on
osoby18_.typ=typosoby24_.klic left outer join c_cislo_pobocky_ppr cislopoboc25_ on
pojistnasm15_.pobocka_produkce=cislopoboc25_.klic left outer join d_pojisteni_ppr
pojisteni26_ on pojistnasm15_.id=pojisteni26_.smlouva_id left outer join
c_sazebnik_ppr csazebnik27_ on pojisteni26_.sazebnik_id=csazebnik27_.id left outer
join c_sazebnik_predmet_pojisteni_ppr predmetyap28_ on
csazebnik27_.id=predmetyap28_.sazebnik left outer join c_pojisteni_ppr cpojisteni29_
on predmetyap28_.pojisteni=cpojisteni29_.id left outer join c_konverze_ppr
konverze30_ on cpojisteni29_.id=konverze30_.pojisteni_id left outer join
c_pojisteni_okamzik_akce_ppr okamzikyaa31_ on
cpojisteni29_.id=okamzikyaa31_.pojisteni_id left outer join c_akce_ppr cakce32_ on
okamzikyaa31_.akce_id=cakce32_.id left outer join c_okamzik_ppr cokamzik33_ on
okamzikyaa31_.okamzik_id=cokamzik33_.id left outer join c_pojisteni_policko_ppr
policka34_ on cpojisteni29_.id=policka34_.pojisteni_id left outer join c_policko_ppr
cpolicko35_ on policka34_.policko_id=cpolicko35_.id left outer join
c_pojisteni_souhrnnylimit_ppr souhrnneli36_ on
cpojisteni29_.id=souhrnneli36_.pojisteni_id left outer join c_souhrnny_limit_ppr
csouhrnnyl37_ on souhrnneli36_.souhrnneLimity_id=csouhrnnyl37_.id left outer join
c_predmet_ppr cpredmet38_ on predmetyap28_.predmet=cpredmet38_.id left outer join
c_typ_predmetu_ppr typpredmet39_ on cpredmet38_.typ=typpredmet39_.klic left outer
join c_spoluucast_ppr spoluucast40_ on pojisteni26_.spoluucast_id=spoluucast40_.klic
left outer join c_typ_pojisteni_ppr ctyppojist41_ on
pojisteni26_.typ=ctyppojist41_.klic left outer join c_typpojisteni_otazka_ppr
otazky42_ on ctyppojist41_.klic=otazky42_.pojisteni_id left outer join c_otazka_ppr
cotazka43_ on otazky42_.otazka_id=cotazka43_.id left outer join
c_otazka_zavislost_ppr zavisina44_ on cotazka43_.id=zavisina44_.pojisteni_id left
outer join c_otazka_ppr cotazka45_ on zavisina44_.master_id=cotazka45_.id left outer
```

```
join d_predmet_ppr predmety46_ on pojistnasm15_.id=predmety46_.smlouva_id left outer
join c_nazev_predmetu_ppr nazevpredm47_ on
predmety46_.nazev_predmetu=nazevpredm47_.klic left outer join d_nemovitost_ppr
nemovitost48_ on predmety46_.id=nemovitost48_.predmet left outer join
d_predmet_policko_ppr policka49_ on predmety46_.id=policka49_.predmet_id left outer
join d_policko_ppr policko50_ on policka49_.policko_id=policko50_.id left outer join
c_sazebnik_ppr csazebnik51_ on predmety46_.sazebnik=csazebnik51_.id left outer join
c_typ_pojistne_hodnoty_ppr typpojistn52_ on
predmety46_.typ_pojistne_hodnoty=typpojistn52_.klic left outer join
c_typ_predmetu_ppr typpredmet53_ on predmety46_.typ_predmetu=typpredmet53_.klic left
outer join c_vlastnictvi_predmetu_ppr vlastnictv54_ on
predmety46_.vlastnictvi_predmetu=vlastnictv54_.klic left outer join d_vozidlo_ppr
vozidla55_ on predmety46_.id=vozidla55_.predmet left outer join c_druh_vozidla_ppr
druhvozidl56_ on vozidla55_.druh=druhvozidl56_.klic left outer join
c_typ_provedeni_vozidla_ppr typprovede57_ on
vozidla55_.typ_provedeni=typprovede57_.klic left outer join c_znacka_vozidla_ppr
znackavozi58_ on vozidla55_.znacka=znackavozi58_.klic left outer join d_zarizeni_ppr
zarizeni59_ on predmety46_.id=zarizeni59_.predmet left outer join d_priloha_ppr
prilohy60_ on pojistnasm15_.id=prilohy60_.smlouva_id left outer join
c_druh_prilohy_ppr druhpriloh61_ on prilohy60_.druh=druhpriloh61_.klic left outer
join d_souhrnny_limit_ppr souhrnneli62_ on pojistnasm15_.id=souhrnneli62_.smlouva
left outer join d_specialni_ujednani_ppr specialniu63_ on
pojistnasm15_.id=specialniu63_.smlouva_id left outer join d_smlouva_spravce_ppr
spravci64_ on pojistnasm15_.id=spravci64_.smlouva_id left outer join d_spravce_ppr
spravce65_ on spravci64_.spravce_id=spravce65_.id left outer join c_cislo_pobocky_ppr
cislopoboc66_ on spravce65_.cislo_pobocky=cislopoboc66_.klic left outer join
c_stav_smlouvy_ppr stavsmlouv67_ on pojistnasm15_.stav=stavsmlouv67_.klic left outer
join c_typ_pojistneho_ppr typpojistn68_ on
pojistnasm15_.typ_pojistneho=typpojistn68_.klic left outer join c_kod_banky_ppr
kodbanky69_ on pojistnasm15_.kod_banky=kodbanky69_.klic left outer join
d_ziskatel_ppr ziskatele70_ on pojistnasm15_.id=ziskatele70_.smlouva_id left outer
join c_cislo_pobocky_ppr cislopoboc71_ on
ziskatele70_.cislo_pobocky=cislopoboc71_.klic where souhrnnyli0_.id=?
```