Design patterns to increase performance when using an object-relational mapping and their categorization

Gerhard Wanner[1], Oriel Maute[2]

[1] Stuttgart University of Applied Sciences, Schellingstr. 24,
70174 Stuttgart, Germany
wanner.fbm@fht-stuttgart.de
[2] plenum Systems GmbH, Max-Lang-Str. 24,
70771 Leinfelden-Echterdingen, Germany
info@oriel.de

Abstract. Today's sophisticated software applications are developed using object-oriented techniques. For database access the object-oriented approach hasn't established – the usage of relational databases is still prevailing. There are a couple of tools that bridge this so called "impedance mismatch". These tools allow the usage of relational database systems with an object-oriented approach. This is achieved in extending the object-oriented language with an object-oriented database model, often called persistency model. But the transparent usage of such a persistency model often leads to unacceptable slow systems. This paper describes how, and why this can occur. It will then introduce different categories of design patterns to prevent certain performance problems when using an object-relational mapping. Finally, it gives examples for each of these categories, pointing out their application area and their advantages and disadvantages.

1 Introduction

Most of today's software applications are developed with object-oriented methods, but use relational database systems for data management. This is common for web-applications with Java Servers Pages and Servlets, fat-client applications with direct database access, as well as application-server based business-applications using Enterprise Java Beans and a host database. Independently of these application-areas a persistence framework is used which bridges the object-oriented and the relational world [1]. A good introduction to persistence frameworks can be found in [2].

With Java Data Objects (JDO) [3] we now have a standard in the Java world, which specifies the integration of the persistence model into the programming language. This standard was long overdue – it offers a possibility to change the persistency framework with manageable costs. In the meantime all important manufacturers offer JDO-compliant products or are at least on the way to adapt them [4].

Having this standard allows one to focus on another problem of using a persistence framework. Since the SQL-commands are generated by the framework, the developer looses the feeling for the costs of object operations on the database. When the amount

of data increases, a navigation along an awkwardly mapped association is sufficient to overload the database system or the persistency framework.

In the following section a set of patterns is specified, in order to improve the performance of an application with the smallest possible interferences with the application code. The patterns describe recommendations for the modelling, the mapping, and the usage of the persistence framework. Since the improvement in one area, like the performance, often involves a degradation in another area, side-effects are described and meaningful application areas are named.

Except for a few exceptions, the described patterns are realisable with each JDO-compliant persistency framework. The exceptions concern only the area of the mapping possibilities, which are not standardised in JDO [5]. As the presented patterns are quite straightforward, they should be adaptable to most of the tools on the market.

2 Patterns

Since the appearance of the well known book of the "Gang of Four" [6] software patterns are one of the main topics in software development. They are not describing the usage of more or less prefabricated components, but the knowledge about available constructional possibilities for the solution of a problem.

This is exactly what our patterns do: they describe the possibilities to improve the performance of an object-oriented application that uses a relational database. The patterns are classified into three areas:

- Modelling patterns: they refer to the modelling of the object-oriented part of the application. The application of such a pattern usually entails the modification of the class model.
- Mapping patterns: they refer to the mapping of the classes to the relations, hence the tables and views. Mapping patterns therefore are based on the change of the mapping rules. Basic patterns in this area can be found in [7].
- Application patterns: they are based on a change of the application code. With these patterns the interface of the persistence model is used differently or in another sequence.

For some of the patterns specified below we give sample code in Java using the programming interface of JDO. The specified patterns, however, are not bound to a specific programming language. Also the samples do not presuppose a certain relational database system, since proprietary extensions, like stored procedures or triggers, are not used.

2.1 Preload associations

A recurring situation in programming user interfaces is that after a query a quantity of objects is displayed. However, to display them, more information from associated objects is needed. Without further precautions this leads to the fact that with n objects $1+n$ SQL-commands are executed to display the data - a phenomenon that is also

known under the name "n+1 database call problem". The following example is to clarify this: There is a class Person which has one or no Address (Fig. 1).

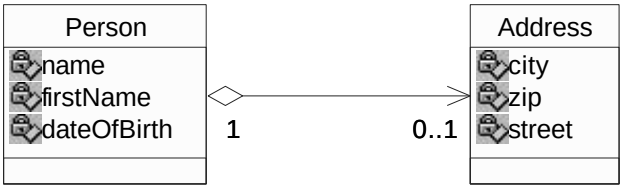


Fig. 1. Class model of the example

To display the query result, name and first name of the Person as well as the zip code of the Address should be used. Therefore first the Person objects are determined according to a given search criteria. Then, to display the result, one navigates from each Person object to its Address object, for example by calling the method 'getAddress()'. Each navigation causes a new database query in the persistency framework.

For this problem there are two solutions. The first one uses the extended mapping features of the persistency frameworks, works only with aggregations and is based on denormalisation. This solution is presented in section 2.2.

The second uses a mechanism based on preloading all related objects and reduces thereby the number of SQL-commands, in the above quoted example to exactly 2.

Basis of this pattern are the "Fetch Groups" or "Default Fetch Groups" available under JDO. In the mapping rules the Default Fetch Group defines for each attribute whether it is to be loaded with the instantiation of the object from the database (default-fetch-group is true) or only at the first access (default-fetch-group is false). According to the standard, all primitive Java types are loaded directly, and all referencing attributes at the first access.

One possibility is to set the Default Fetch Group for the involved referencing attributes in such a way, that the related objects are already loaded with the instantiation of the original object. In the example the solution is to include the referencing attribute 'address' into the Default Fetch Group.

This pattern belongs to the mapping patterns, because the specification of the Default Fetch Group takes place in a declarative manner in the persistence descriptor. However, from our point view this has two disadvantages:

- Firstly it is not sure that the used JDO implementation implements an optimised strategy for loading the related objects. When selecting a JDO framework it is important to pay attention to such details.
- The second drawback is that related objects are always loaded with the Default Fetch Group, even if they are not needed at all. The declaration of a Default Fetch Group thus only makes sense with referencing attributes which are used always or nearly always.

One solution to this problem is the declaration of a Fetch Group in the query. Beside the Default Fetch Groups it is possible to declare further Fetch Groups in the persistence descriptor. There a Fetch Group can be declared, which includes a referencing attribute. Now this Fetch Group can be used in a query and the referencing attribute is loaded for all selected objects (Prog. example 1). This application pattern works actu-

ally exactly as described above, only it is adjustable finer - the references can be loaded where it is necessary because of performance reasons.

```
Extent lExtent = lPM.getExtent(Person.class, true);
Query lQuery = lPM.newQuery(lExtent);
lQuery.declareParameters("String jdoGenieOptions");
Collection lResult = (Collection)
    lQuery.execute("fetchGroup=load_addresses");
```

Prog. example 1. Specification of a Fetch Group in a query

But no advantage without disadvantage: Fetch Groups are not a JDO standard! There are some JDO implementations that offer this mechanism and support it with adequate optimizing procedures. An example is JDO Genie [8]: there the Fetch Group can be declared inside a query. Nevertheless this can be done in a way, which does not prevent a change to another JDO framework.

2.2 Map classes with to-1-aggregations into one table

As already indicated in the preceding section there is a second possibility for improving the performance in the case of to-1-aggregations. This mapping pattern uses mapping possibilities of the persistency framework, namely the mapping of two classes in one table. Since the persistency framework hides this special mapping from the developer - the class model is not changed - this is a possibility to increase the performance without any changes in the application code

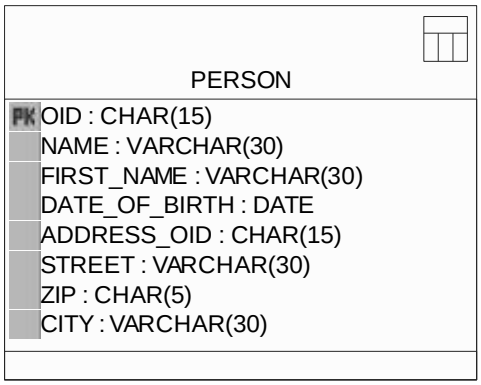


Fig. 2. Relational mapping for the example

As Fig. 2 shows all attributes of the class Address are moved to the table of the class Person. Since the association is optional (cardinality 0..1) also the OID of the Address is included as ADDRESS_OID, in order to distinguish between an empty Address and a non-existing relationship. The persistency framework now ensures that the Address data is loaded while loading the Person objects; it therefore does not need a further SQL command. In the next step all Person objects get instantiated, the Address objects

get instantiated and the relationships between the objects are created. The application developer notices nothing of all that, since the class model remains unchanged. From the point of view of the application development this is a very good solution, because necessary changes remain limited to the mapping.

Unfavourable at this solution is that the Address data is loaded even if the Addresses are not accessed at all. That means that when loading objects of the class Person the associated Address object is always instantiated and the relationship between the two objects is established. There is another disadvantage if, as given in the example, the relationship is optional. In this case there are empty fields in the database. The most critical disadvantage is, however, that this procedure is not standardised. Not all manufacturers of persistence frameworks offer this mapping variant, therefore their usage creates a dependency on certain products.

Why does this sample only work with aggregations and not also with regular associations? The main reason is, that with regular associations the related object (in the example the Address) could be stored independently of the original object (in the example the Person). In the database however, there would be a table entry without primary key, since the OID of the Person object is not determined yet.

2.3 Use queries instead of multi-level object navigations

As mentioned before the goal of an object-relational persistence framework is to hide the relational view from the application developer. For many developers this means that knowledge about relational database systems and the basic internal functionality of the persistence framework is not necessary, since the framework represents an abstraction layer, which lets the relational view appear as a black box. Unfortunately an object-relational persistence framework cannot be regarded as an independent layer without restrictions.

Let's consider an example: a vendor has customers with possibly several open orders. An open order consists of one or more articles.

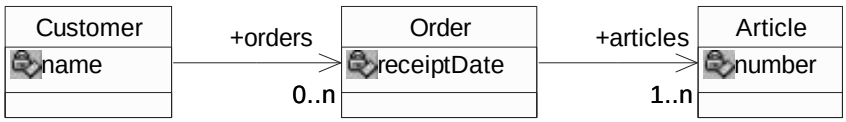


Fig. 3. Class model of the example

Let's suppose we are interested in all customers named 'Miller'. As additional search criterion, each resulting customer must have at least one open order, in which at least one article with the number '4711' appears. This task could be solved in the following way:

```
Iterator lCustList = lPM.getExtent(
    Customer.class, false).iterator();
while(lCustList.hasNext()) {
    Customer lCustomer = (Customer)lCustList.next();
    if("Miller".equals(lCustomer.getName())) {
```

```
            Iterator lOrderList = lCustomer.
                getOrders().iterator();
            while(lOrderList.hasNext()) {
                Order lOrder = (Order)lOrderList.next();
                lIterator lArticleList =
                    lOrder.getArticles().iterator();
                while(lArticleList.hasNext()) {
                    Article lArticle = (Article)
                        lArticleList.next();
                    if(lArticle.getNumber() == 4711) {
                        // Customer lCustomer has the name
                        // 'Miller' and at least one open
                        // order with the article 4711

                    }
                }
            }
        }
    }
```

Prog. example 2. Inefficient search of customers using multi-level navigation

The above program meets the business requirements. Unfortunately the implementation is almost catastrophic regarding performance and memory consumption. Let's regard the disadvantages in detail: For the determination of all customers one SQL command is needed. For each customer object with the name 'Miller' a further query is necessary to get its open orders. To retrieve the articles of every found open order, another query is required. Altogether approximately $1 + n + n * m$ SQL instructions are thus set off, whereby n stands for the number of average open orders per customer and m for the average number of articles of an open order.

Unfortunately not only the number of SQL commands, which are indirectly executed by the program code above, are unfavourable for the performance, but also the number of materialized objects, although these objects are not needed at all in our example.

These disadvantages are caused by the object navigations, which are triggered by calling the harmless looking get-methods. The developer should always be conscious that the call of a get-method of a persistent object causes a SQL instruction, if the result of the get-method is not already cached. Besides this, calling a get-method on a persistent object entails that objects of the destination class will be at least partially materialized.

An experienced database programmer will immediately recognize that the business requirement above can be mapped with only one SQL command. Knowing that objectoriented retrieval queries in object-relational persistence frameworks are normally converted into one single SQL command, an efficient solution of the problem results: Only one query in an objectoriented query language is to be executed, to cover the entire business requirement. Using JDO, the query can be expressed in the following way:

```
Class lCustomerClass = Customer.class;
Extent lCustomerExtent =
    lPM.getExtent(lCustomerClass, false);
String lVariables = "Order O; Article A";
String lFilter = "this.name == paramName " +
    "&& this.orders.contains(O) " +
    "&& O.article.contains(A) " +
    "&& A.number == 4711 ";
String lParameter = "String paramName";
Query lQuery = lPersistenceManager.newQuery(
    lCustomerExtent, lFilter);
lQuery.declareParameters(lParameter);
lQuery.declareVariables(lVariables);
Collection lCustomer = (Collection)
    lQuery.excecute("Miller");
```

Prog. example 3. Efficient search of customers using JDOQL

Using this JDOQL query, the business requirements are met and only one SQL command is executed. Besides this, unnecessary objects won't be materialized.

The example shows that multi-level object navigations should be replaced by JDOQL queries. Thus the number of SQL commands will be reduced and the memory consumption will be kept small.

The design pattern of this section can be verbalized as follows: Multi-level object navigations should be replaced by JDOQL queries, if the objects of the intermediate classes are not part of the search result and thus are not explicitly required.

The design pattern of this section is considered an application pattern. For its implementation existing application code must be changed. How do you recognize whether to use the shown design pattern or not? A first hint is a large number of SQL commands executed by the code-section – this information can be easily extracted from the SQL log.

After that the code must be analyzed to determine whether objects of intermediate classes of the navigation are needed for the further processing. If objects of intermediate classes are needed, then the pattern 'preloading of associations' should be preferred (section 2.1). If objects of intermediate classes are not needed, then the multi-level object navigations should be replaced by a single JDOQL query, corresponding to this design pattern.

2.4 Limit the attributes to be loaded during object instantiation

From an object with many attributes frequently only a small number of attributes is needed for graphical display, e.g. in a listbox. Therefore the object needs not be loaded completely.

Many object-relational mapping frameworks support the possibility to specify which attributes are to be filled in the result objects of each query. The remaining attributes are filled with proxies for on-demand loading of the whole object.

An auction system serves as example for this design pattern. The class Auction (Fig. 4) consists of more than 10 attributes, from which at least the attribute 'description' can contain a long text, possibly HTML code.

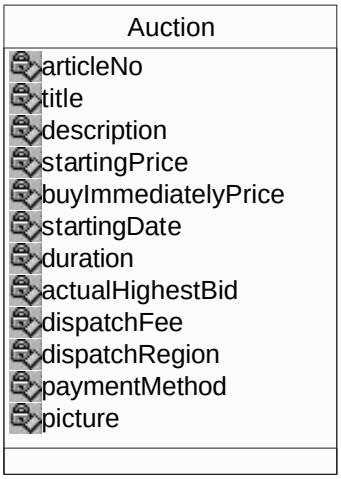


Fig. 4. Class model of the example

Probably only the attributes 'articleNo', 'title', 'startingDate', 'actualHighestBid' and 'duration' - for the calculation of the auction end - are visible to the user. So it is sufficient to load these 5 attributes for each Auction object.

The possible increase of performance by this pattern depends on the ratio of loaded attributes. Even though the number of SQL instructions remains unchanged and all objects must still be instantiated, the partial loading has significant effects. This applies even more, if large data or text fields are affected: fewer data must be selected from the database, transferred and filled into the objects. This accelerates the expensive object instantiation and additionally reduces the memory consumption of the application.

Another advantage of this design pattern is that neither the class model nor the object-relational mapping has to be modified. However the application code must be changed slightly: for each object-query one has to decide about the attributes to be loaded. Here one has to be extremely careful: if you forget to load an attribute which you actually need, then the object will be completely loaded by the proxy-mechanismn. This leads to one additional SQL command for each object! This may particulary happen during maintenance, if users wish the display of a further attribute…

2.5 Avoid bi-directional associations

The last design pattern, a modelling sample, discusses the type of association between two classes. While modeling the class diagram, associations will be identified by reviewing the use cases. Thereby nearly always unidirectional associations are

sufficient. But nevertheless often these associations will be modelled bi-directional. The reason for this is usually carelessness or comfort. And this, although bi-directional associations are the large exception; not without reason the uni-directional association meanwhile is the default in many modelling tools.

First of all, a bi-directional association may be helpful, since thereby it can be navigated in both directions and another use case may possibly exist (or added later), which needs the other navigation direction. But this is bad for the performance – not only, but in particular for object-relational mapping frameworks.

Example: A class Person has an association to a class Country. It is important for the use cases, that the corresponding Country to a given Person can be queried, e.g. to retrieve the country-code. The other direction of the association may also be reasonable - thus all inhabitants of a Country could be determined.

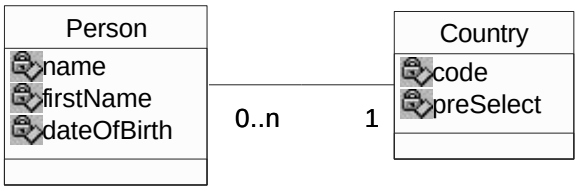


Fig. 5. Class model of the example

The problem with the bi-directional association is, that there must be a container in each object of the class Country, containing all inhabitants of the specific Country. It is obvious that in such a container hundreds of thousands of Person objects can be contained at runtime, and these objects first must be materialized from the database.

Why does the container include these objects, although the navigation direction from a Country object to its inhabitants (objects of class Person) was not yet used at all? Different mechanisms of object-relational mapping frameworks must load these objects, for example for integrity checks. Apart from this, the problematic object navigation from a Country object to its inhabitants can also be used inadvertently without explicitly using the Person objects – e.g. by calling the 'size()' method on the container object. In each case the developers should be protected against such technical pitfalls. Further arguments against the use of bi-directional associations are possibly additional memory consumption, performance loss for the management of the association by the persistence framework and problems with the garbage collector.

For example, the Country objects belong to the static data, which are durably stored in the memory of the application. After an object navigation along the association from a Country object to its Person objects, the container of the Person objects will be created and durably stored as a strong reference in the corresponding Country object. Thus the Person objects are durably ignored by the garbage collector!

In summary each bi-directional association must be scrutinized. As long as all use cases can be implemented this is even more true, because an uni-directional association does not have any disadvantages.

3 Conclusion and further work

The design patterns discussed serve as a good starting point to increase the performance when using a persistence framework. They represent one possibility to comply with non-functional requirements, especially regarding the response-times.

Of course the shown design patterns cannot eliminate the negative effects of a totally botched architecture. But caused by the transparency of the object-relational mapping, developers are tempted to write code, that won't reach the required performance, despite an otherwise good architecture.

Some of the design patterns, e.g. the pattern 'avoid bi-directional associations', should be an integral part of the used modelling guidelines.

The discussed design patterns are also useful when selecting a JDO-compliant persistency-framework. Simply apply the patterns and examine the SQL-trace produced by the frameworks to separate the wheat from the chaff.

The next stage in our research is to find ways to integrate the mentioned patterns into an abstraction layer of the persistence model. The idea is still to have an JDO-compliant interface but it's the framework and not the developer that does the optimizations. This is not easy to achieve because, as you see while discussing the patterns, it's often necessary to consider the application as a whole. A local performance optimization could lead to an overall decrease of performance. Our research follows the direction of building a framework that collects statistical data on how the application uses the persistence model and then optimizes the database access step by step during runtime. Overall this leads to a learning application that's getting faster and faster and even optimizes itself when the system is changed by the developer, for example through the modification of the access and navigation paths.

References

1. Scott W. Ambler. The Fundamentals of Mapping Objects to Relational Databases. See http://www.agiledata.org/essays/mappingObjects.html
2. Wolfgang Keller, Jens Coldeway. Relational Database Access Layers: A Pattern Language, Collected Papers from the PLoP’96 und EuroPLoP’96 Conferences Washington University, Department of Computer Science, Technical Report WUCS 97-07, February 1997
3. Java™ Data Objects Specification ("Specification"), Version: 1.0 FCS, March 2002, Sun Microsystems
4. JDOCentral.com. Common Page around JDO with tutorials and links to manufactures. See http://www.jdocentral.com
5. Robin M. Roos. Java Data Objects. Addison-Wesley Professional, October 2002, ISBN 0-3211-2380-8
6. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns. Addison-Wesley, 1st edition, January 1995, ISBN 0-2016-3361-2
7. Wolfgang Keller. Mapping Objects To Tables: A Pattern Language, Proc. Of European Conference on Pattern Languages of Programming Conference (EuroPLoP)’97, Bushman, F. and Riehle, D.; (eds), Irsee, Germany, 1997
8. Hemisphere Technologies - JDO Genie Manual. Version 1.2.4 (31. January 2003). See http://downloads.hemtech.co.za/jdo/manual/manual.html