COMENIUS UNIVERSITY, BRATISLAVA, SLOVAKIA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
DEPARTMENT OF COMPUTER SCIENCE

OBJECT-RELATIONAL MAPPING

Diploma Thesis

Diplomant:
JAROSLAV ORSÁG

Diploma thesis supervisor:
ING. ARCH. AUGUSTÍN MRÁZIK

BRATISLAVA 2006

Object-relational Mapping

DIPLOMA THESIS

Jaroslav Orság

COMENIUS UNIVERSITY, BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
DEPARTMENT OF COMPUTER SCIENCE

Computer science

Diploma thesis supervisor
Ing. Arch. Augustín Mrázik

BRATISLAVA 2006

I would like to say thank You to my diploma thesis supervisor Augustín Mrázik for his help, books as well as professional and valuable advices while I was writing this diploma thesis.

Hereby I honestly declare that I have worked on this diploma thesis independently, using only the listed literature.

Bratislava, May 2006

……………………...........

Abstract

Storing of objects to relational databases is actual topic today. Object-orientation is mainstream approach that is used for application development today and applications usually have to store their data. If we talk about enterprise applications, we can say that it is usual choice to use relational databases as a data store. As there is a gap between object-orientation and relational paradigm – we call this semantic gap – this task is not straightforward. Object-relational mapping is in place especially when we deal with large information systems that have to be open and easy to maintain. One option how to deal with this issue is to develop own layer that manages the mapping between objects and relational databases' entities in our application. Another option is to use third party product which can automate majority of mapping process when used properly. In first part of this work we will look at ways how such product can be designed and implemented as well as patterns for mapping of object-oriented constructs to entities of relational databases will be presented. We will examine advances as well as drawbacks of specific approaches or patterns. Then in the second part we will examine how these products look like, categorize them by different criteria and compare them. We will also look at architecture of these solutions and compare their architecture from different aspects. When categorizing and comparing of persistence frameworks we will use knowledge from first part of this work.

The goal of this diploma thesis is to compare persistence frameworks and to provide deeper look into internals of these frameworks. This thesis provides comparsion table with selected persistence frameworks, provides a basic orientation in the variety of ORM tools and emphasizes some characteristics of ORM tools that are worth of our attention. We will see that there exist varieties of tools which are suitable for varieties of scenarios.

Keywords: Persistence, Mapping, Object-relational mapping, ORM, Persistence framework

Table of contents

List of Figures

List of Tables

1 Introduction

Nowadays, the object-oriented programming is already widely accepted approach for the development of business applications. Object-oriented development provides more suitable methods and facilities for modeling of the real world objects than e.g. the paradigm of structured programming.

We speak about facilities such as class, which is a unit of definition of data and behavior, or object, which is a run-time instance of class. If we are developing an application in an object-oriented language we will probably use encapsulation as means for hiding data and behavior (methods). Another very useful concept of object-oriented programming is the concept of inheritance. With inheritance we can extend existing classes by new data and/or behavior. In a specialized subclass we can use all inherited data and behavior and we can change (override) the inherited behavior. Say we have class A, and classes B and C inherited from class A. Concept of abstraction enables us to work with instances of B and C as if they were instances of A. Very powerful feature is polymorphism, where the behavior of an object depends on its actual type. These words about OOP are just very introductory and reader can get more information about basic concepts of OOP for example in [ŠEŠERA, MIČOVSKÝ].

A widely spread language for writing in object-oriented style today is Java with its frameworks, technologies, APIs and specifications. Java became something like mainstream during its more than decade lasting availability for community. Another platform for the development of object-oriented applications is Microsoft's .net platform. This platform is younger than Java and not so spread as Java. Anyway it records popularity among lots of developers. We would say that if there are not any special requirements for the product, in our case Enterprise application, choice would be one of the above mentioned technologies. In the past Smalltalk or C++ based solutions were conventional, if we speak about object-oriented development.

In this diploma thesis we will mainly deal with Enterprise information systems (EIS), or Enterprise applications. We will outline Enterprise application as application for corporations or institutions, where more than one user works with this application. Components of this application are usually distributed over a computer network. Enterprise applications are usually working with big amount of data, which have to be persistent. Lot of users work with this application concurrently and usually there are groups of users that have different view on application and thus have different user interfaces. Enterprise application usually has to be integrated with other systems.

We have mentioned that data with which Enterprise application works usually have to be persistent. Persistence of data / objects is usually provided by databases systems. As we are working with objects, straightforward solution would be to store objects directly in an object-oriented database (OODBMS). Many people believed this idea in the ‘90s. OODBMS vendors promised this would become the mainstream for storing objects. There are two main reasons why object-oriented databases failed to become mainstream:

- Heterogeneous environments. Object-oriented development and deployment environments were totally different technologically and conceptually. These environments were based primarily on languages such as C++, Smalltalk or Java. By some experts C++ is not even considered to be an object-oriented language. In contrast, Smalltalk is considered to be a fully object-oriented environment with full support for reflection. Java is based on same principles as Smalltalk, albeit with conventional C++-like syntax. There used to be a joke that Java is a crippled Smalltalk with a C++ syntax.

- Dependence on legacy systems. In the ‘90s there were great expectations for a switch to object-oriented databases. We speak about Enterprise applications of large banks, insurance companies and similar. These times, there have been many legacy database systems based e.g. on the network model specified by CODASYL with COBOL as a host language or they had their data in a relational databases that were also already widespread that time. For large companies it was very difficult to switch to this new platform that time since they had all their important business data stored in these legacy systems.

Instead of object-oriented databases becoming the mainstream, relational databases remained in this role. In late ‘60s Codd published his first paper about relational databases. From this time relational databases evolved a lot. Relational databases are strictly data-oriented. Even relations between data are data. SQL standard as a query language for relational databases evolved also and lots of vendors are extending it by new capabilities. Entity-relationship diagrams are useful tool for high-level modeling of real world for needs of database design. Entity-relationship diagram consists of entities, their attributes and relations. Entity is object of the real world, e.g. a car. Attribute of Car is engineType for example. We can also define relationships between entities. These relationships can be of multiplicity 1:1, 1:n, m:n. It is also possible to have n-ary relationships, where n > 2. When designing logical data model (tables), we model these relationships as attributes of corresponding tables.

For completeness we want to mention object-relational databases, as well. These are relational databases that enable to apply some principles of OOP. E.g. in Oracle since its version 8 there offers a possibility to define abstract data types and pointers. More of today’s database servers are considered to be object-relational - for example DB2, PostgreSQL and already mentioned Oracle. For more information about object-relational databases see [WIKI1].

Usage of relational databases for storing of data is typical for today's situation in software engineering. Enterprise applications are developed in object-oriented environments on the other hand. Resolving of this mismatch (between relational data store and object-oriented software) is necessary for ensuring of persistence of objects. Note that we want to define this persistence for full-featured objects. We don't want to define model where we can work with objects in kind of restrictive way. When we want to make objects persistent, we have to define procedures how to map objects to strictly data-oriented structures of relational databases.

There was a joke in '90s about a car and a garage. Usual situation is that when we come home, we park our car in a garage, turn off the engine and go away. If we want to use the car again, we start the engine and get out of the garage. This approach was shown as an example of storing objects in an object-oriented database. In contrary, storing a car in a relational database would be as follows: we park our car in a garage, and then we take our car apart to pieces and store them to shelves. When we want to use our car again, we "simply" assemble the car from the pieces and start using it. This joke spoke in favor of object-oriented databases, however as we have seen, relational databases remained in the mainstream.

Anyway, the story of disassembling and assembling the car represents the situation with object-relational mapping very well. This is the subject of this diploma thesis. In other words, it is about the layer in our Enterprise application which maps objects to the underlying relational database and enables to store and retrieve them efficiently. In the thesis we will analyze problems and techniques associated with object-relational mapping. We will also describe some widely accepted tools/frameworks for object-relational mapping and we will provide simple benchmark of these tools. One of the frameworks will have our special attention – to make the reader familiar with this kind of tools. For each framework we will also examine and categorize it from the view of patterns, principle of work and supported scenarios of mapping.

2 Terms and abbreviations

In this short part we will introduce some terms and abbreviations that will be used in this work and with which the reader should be familiar.

We will mean the same by terms object-relational mapping and abbreviation of this term – ORM.

By ORM tool, persistence framework, object-relational mapper or O-R mapper we will mean a tool, or component that enables mapping of objects to relational databases. This tool usually adds special tier to application and typically realizes CRUD (see later) behavior.

Terms relation and table will mean the same in some situations. By table we mean table in a relational database. Although relation is more general term, it will be clear from context if we mean database table or relation among entities. We will also mean the same by column and attribute of database table. Same situation is with row and tuple of database table.

DAL (Data Access Logic) or DALC (Data Access Logic Component) is component, or set of components which work above data access API. This API can be JDBC, ODBC, ADO, or similar. DAL or DALC are layers that should separate business logic code from database access code. It is important to note, that data access API (such as JDBC, ADO) is not persistence framework at all. On the other hand DAL or DALC can evolve into persistence framework/layer.

We will use terms as Domain model and Business logic layer for layer in our Enterprise application, which contains core objects and behavior of our application. We can say that this is the heart of our application. For more information about Domain model and layering, see [FOWLER]. Sometimes we will call Business entity object from Business logic layer (Domain model).

CRUD (Create, Read, Update, Delete) is kind of behavior of e.g. component. DAL components usually have CRUD behavior. They have to call insert, select, update and delete statements on database. For example if we have CarDalc component, this one's responsibility will be creation of new cars in database, as well as updating attributes of existing ones. Functionality for selecting of existing cars from database and for deleting is also embedded in CarDalc.

3 Impedance mismatch

In the introduction we have outlined what object-relational mapping is and why it is important to deal with it. Here we will introduce some problems with mapping and introduce the term of “impedance mismatch”. Impedance mismatch is more general term and it also appears for example when mapping Domain model to User interface.

In our application we will probably have a Domain model, which will contain core objects with behavior, which will need to get persistent. ORM Impedance mismatch is set of technical and conceptual difficulties when developing applications in which Domain model is object-oriented and the underlying data store is a relational database. Role of persistence framework is to cope with these difficulties. This mismatch is given by differences between object-oriented and relational paradigms.

If we wanted to translate some fundamental concepts of OOP to relational terminology we would say that term class roughly fits to database table. Next, object as an instance of a class could be table record.

Let’s speak about encapsulation as a mean for hiding of implementation details. In OOP, class developer provides client developer just with public interface, to avoid creating dependencies. OOP language enables him to do this by language constructs. Speaking about relational databases, there is no similar concept. First, no behavior is associated with tables (which roughly fit to classes) and second, no access modifiers can be defined for stored procedures or functions. We do not consider execute permissions for stored procedures, or functions.

Inheritance, abstraction and polymorphism – there is no native equivalent for these concepts in original relational theory and technology of RDBMS, although nowadays most of the RDBMS vendors implement some object-oriented features into their systems. We have already mentioned these systems – they are object-relational databases. For example PostgreSQL has support for table inheritance.

Anyway, the role of RDBMS is, roughly said, to efficiently store data, and to enable fast access to these data. That’s why we don’t require some of the concepts as much as other ones. For example simulating of inheritance, abstraction, or polymorphism is not as crucial as for example existence of an equivalent for a class.

Besides these fundamental differences there are some other difficulties, as well:

- Declarative vs. imperative approach. Object-oriented languages are imperative languages. They have methods in which we can say how to do something by sequence of statements. In contrast, SQL is a declarative language, where we just say what to do.

- Normalization and modeling. Creating of database schema is different process from modeling of object model. Usually we try to get normalized database schema without redundancy and without having ambiguous data. On the other hand, when modeling object model we use different techniques. We can use concepts such as inheritance, polymorphism and abstraction. Object-oriented design is usually influenced by object-oriented analysis. In the ideal case, we compose design patterns together.

4 Basic classification of persistence frameworks

This chapter introduces the basic classification of OR mappers. At first we give basic assumptions about design and architecture of our application. Then we will explain where mapping layer fits into this architecture and provide some ways how this can be designed. We will provide just basic and high-level image of this architecture in this part.

4.1 Design considerations

In this work we will consider multi-tiered application which uses a relational database as a data store. Multi-tiered architectural style is a widely spread approach for the development of Enterprise information systems nowadays. We will describe a generic EIS in bottom-up direction. EIS that will be described is simple system and there are lots of more complex (and more simple) systems different from this one, but with similar architecture. This is just an example of how such system could look.

As we can see in the Figure 1, at the bottom there is RDBMS with the data. We access RDBMS by data access API, which could be JDBC, ODBC or ADO.

Above it there are data-access logic components – DALC. This layer passes calls such as SQL selects, stored procedure calls, or function calls to data access API. Classes in this layer fill Business entities with data, update data in database depending on changes in Business entities, insert and delete data from database. This is frequently referred to as CRUD behavior.

Above DALC layer there is Business logic layer with Business entities. This layer is independent of data store and no SQL code is used within it, if properly designed. In ideal case only a few calls on DALC are needed.

At the top, there is User interface designed according to Model-View-Controller architectural pattern. We will not deal with User interface or Impedance mismatch between User interface and Business logic layer. For more information see [MICHALEC].

Our goal is to automate creation of DALC layer. Instead of DALC we will have ORM framework, which will supply persistence logic. There are several ways how to supply this persistence logic. Later we will see that we can divide mappers to code generators and reflection-based. Now we will examine how the persistent logic can look from the view of programmer/designer of Business logic layer. From his view we can divide mappers into three main groups:

- mappers that use tuples approach,

- entity approach,
- domain model approach.

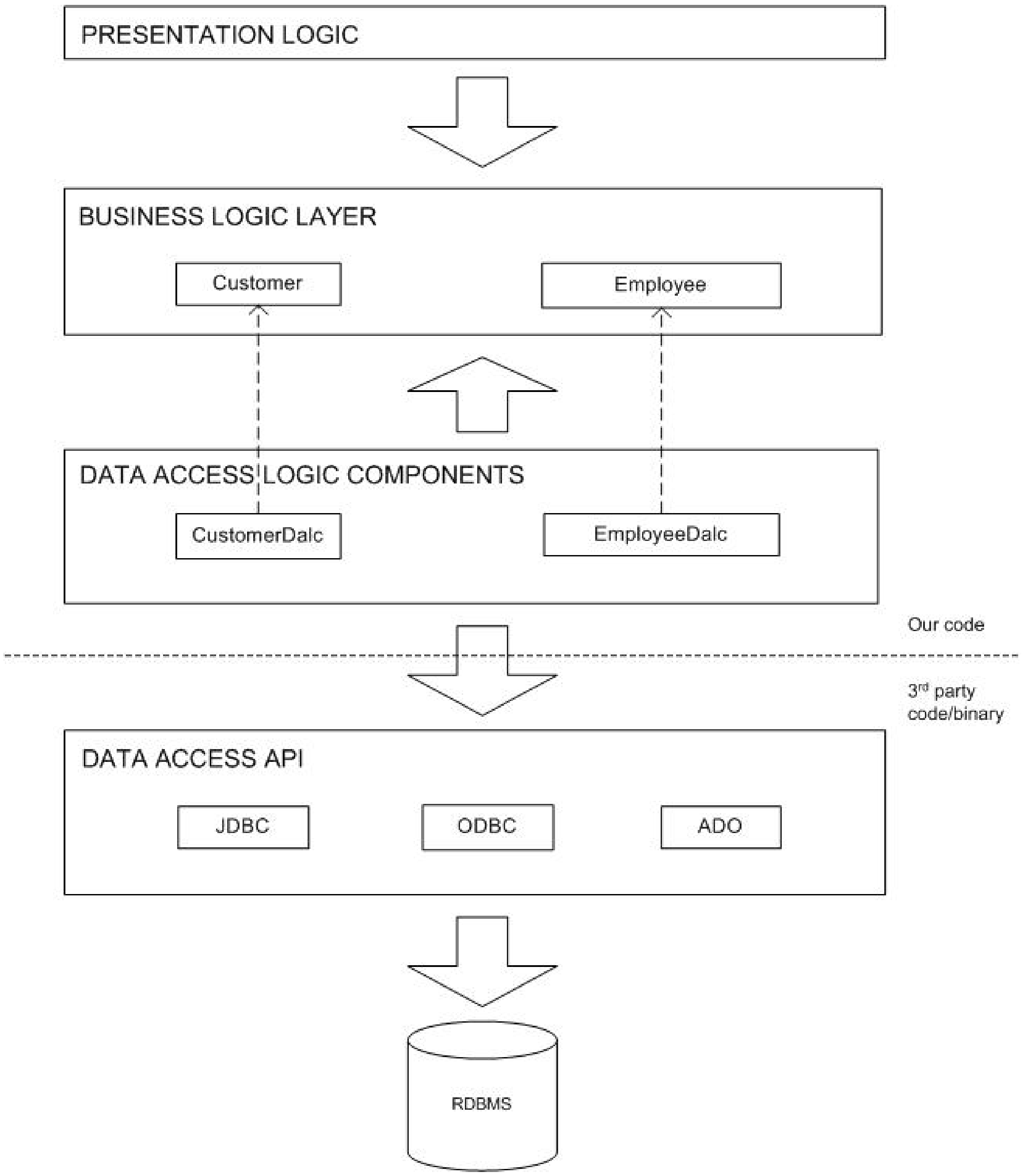

Figure 1 - Architecture of typical EIS

4.2 Types of mappers

Here we will present basic types of mappers by design of originated Business entities. We will begin by simple approach used in very simple applications such as guest books on www pages. Then we will present more sophisticated approaches used by real OR mappers.

4.2.1 Tuples approach

When this approach is used, it is a question whether we speak about object-relational mapping at all. Application simply reads records from RDBMS to record sets and then it works with these record sets. Data aren't read to objects (Business entities). This approach is not appropriate for the development of larger systems, because Business logic is very tightly coupled with the format of the data in the data store. When e.g. attribute moves from one table to another table (e.g. because of refactoring), all places in application source code that works with this attribute must be changed.

4.2.2 Entity approach

As in the previous case, application reads data to record set at first. After reading into record set, Business entities are filled with data from this record set. These Business entities have form of data structures, which are very similar to tables from relational databases. For example DataSets, or DataTables used in ADO.NET can be used for this purpose. ADO.NET DataTable is tabular structure, which can have PK, or FK constraints defined. DataSet is set of DataTables, among which relations can be defined, with support of cascade update or delete. We can watch similar functionality in today's RDBMSs and we can say that DataSet is an offline mirror of part of the database.

Drawback of this approach is that our business entities have no methods – no behavior. Only simple methods associated with low level processing of data can be observed. There are methods such as `CheckConstraints()`.

Business logic is then placed in so-called management classes. Business entities without behavior are arguments of Business logic management classes' methods.

This kind of design preserves relational thinking at the level of objects.

4.2.3 Domain model approach

This approach is similar to entity approach. The difference is in Business entities. In this approach Business entities are real objects with behavior. Objects are not just containers for images of database tables. We can use inheritance, polymorphism and other concepts of object-orientation. Business logic is placed in methods of Business entities. There is no need of management classes, as mentioned above.

5 Patterns & Techniques

Until now we have looked at object-relational mapping or at ORM tools from outside. We have said what ORM framework is and how we can categorize ORM frameworks from the view of application developer/designer, but we haven't said how they can be designed. In this chapter we will start looking at persistence frameworks from inside. This means we will look at structure (design and architecture) of these products and at their behavior and we will categorize them from this point of view.

At first we will look at structural patterns. In our case this means how structures from Domain model are mapped to structures in the relational database. We will examine several ways how inheritance structures can be mapped to tables. With each approach we will review advantages as well as drawbacks. In similar way we will examine mapping of relationships and aggregations.

Then we will look at the performance a bit closer. We will examine how performance of persistence tool can be improved by a good design. Few patterns introducing concepts of denormalization, caching and lazy loading will be presented.

Finally we will deal with configuration. We mean configuration of mapping. That is, how client developer can force persistence framework to behave in a specific way. For example he might want to tell that class Car from Domain model has to be mapped on database table tblCar. We will introduce concept of attribute-oriented programming and compare this approach with use of usual configuration files. We will also deal with support (respectively workarounds when no support exists) of frameworks such as Java and MS.net for attribute-orientation in short.

Note that this chapter can not be considered as complete list of patterns for ORM. In each part references to suitable resources will be listed. These references will help the reader to get more information about this topic.

5.1 Structural patterns

Here we will look at structural patterns that can be used by mappers. There are patterns for mapping of object-oriented structures, such as inheritance hierarchies, aggregations, relations. While coping with relations we will consider different kinds of relations by cardinality and orientation.

We will present several solutions (patterns) here. OR mapper designer can decide which solution he will adopt according to requirements he wants to meet.

For more detailed discussion about benefits and drawbacks of patterns described in this section see [AMBER] or [KELLER]. For more information about implementation of these patterns see [FOWLER].

5.1.1 Strategies for mapping of inheritance

As we have mentioned earlier, inheritance is not inherently supported by relational databases. That's why there are specialized patterns for translation of inheritance structures. It is kind of serialization, where inheritance tree is stored to database and, of course, it is possible to restore it. Note that we won't consider case of multiple inheritance. Reader can discuss problems of mapping of multiple inheritance with [AMBER].

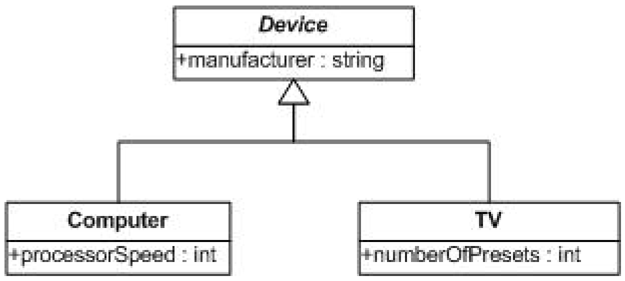

Figure 2 - Example of an inheritance tree

While presenting patterns for mapping of inheritance, example would be surely convenient tool for explanation. Our example is depicted in Figure 2, where we can see simple structure with three classes. Name of the base class is Device and names of its descendants are Computer and TV. Device has attribute manufacturer, which is common for both descendants. Attribute specific for Computer is processorSpeed and attribute specific for TV is numberOfPresets. Device class is abstract, so it can not be instantiated. To be honest, this example is probably too simple, but here it serves only as an example of a simple inheritance structure, where every class has its specific attributes.

Note that it is possible to combine strategies presented below within an object model.

5.1.1.1 Hierarchy to a single table

This is probably the simplest approach for mapping of inheritance structures. Whole inheritance hierarchy is mapped to a single table, which should be named after most general class in the hierarchy (Device in our example). As we can see in Figure 3, set of attributes of Device table is equal to union of attributes of all classes in a hierarchy plus deviceType attribute.

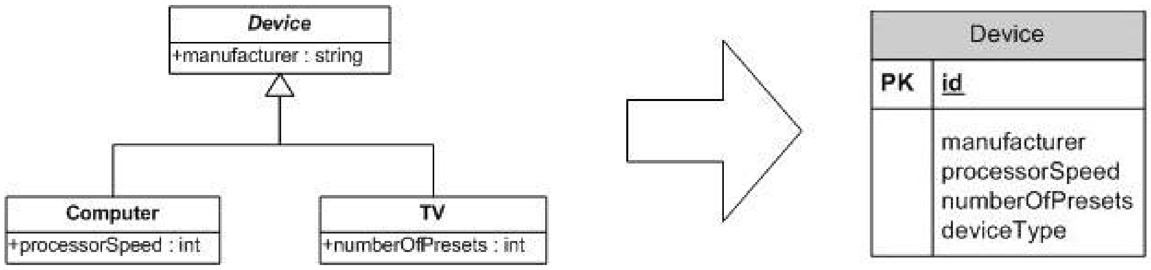

Figure 3 - Hierarchy to a single table

This attribute indicates type of Device in specific row in the table. It can be implemented as varchar. If we wanted to be correct with normalization rules, we would make it to be foreign key to table with Device types. If Device could have several types (which probably isn't our case), we have an option to add boolean attributes to Device table. One boolean attribute would specify, whether specific row is of appropriate type. Decision to make type assignment based on NULL values isn't the correct one. Imagine numberOfPresets attribute of TV is nullable and processorSpeed attribute of Computer is also nullable. Then we could have row with values <'Manufacturer', <NULL>, <NULL>>. In this case we don't know whether this is Computer, or TV.

Drawback of resulting database schema is its space consumption. There are lots of NULLs and we can observe that with bigger amount of classes we have bigger amount of wasted space. On the other hand this structure has good performance because we need only one select to get particular object with all its properties. We need no additional joins.

Another drawback is that all traffic associated with arbitrary class from inheritance tree is concentrated to only one table.

Maintenance of resulting scheme is very easy.

5.1.1.2 Each class to its own table

As we can see in Figure 4, every class in inheritance hierarchy is mapped to its own table. Note that also abstract classes have their tables. Every table has attributes of its corresponding class. There are no attributes inherited from super classes here.

Let's look at ID attribute of tables and its meaning. In the case of inheritance tree root class's table, this attribute has different meaning as in others tables. In the root class's table it is primary key. In descendant classes' tables it's primary key as well as foreign key. It points to parent class's table and in appropriate row there are values of inherited attributes from this parent. Similarly we can traverse up to the root of the hierarchy and get all attributes inherited from all the ancestor classes with help of this ID attribute.

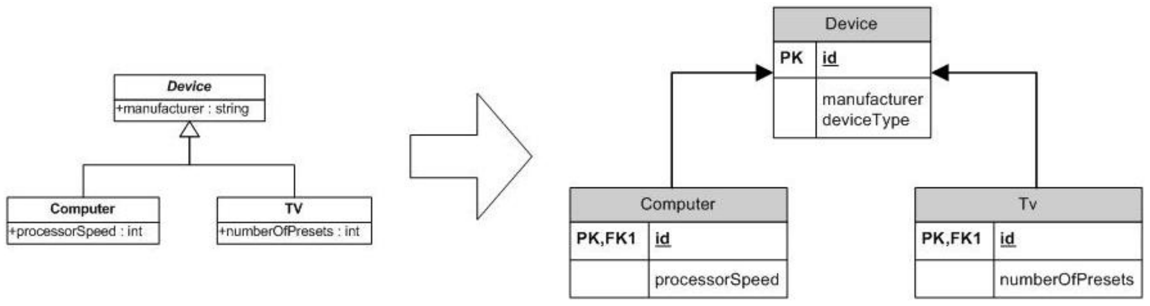

Figure 4 - Each class to its own table

For better performance we will maybe need identification of type in base class's table, as we have already mentioned earlier. Use of boolean attributes is an option.

If we wanted to see tables with all inherited attributes, ideal solution would be use of views defined at database server. Simply said, views are stored and named SQL selects on server. We can have one view for each class in our inheritance hierarchy. Each of these views performs as many joins as deep is our class in inheritance hierarchy.

Drawback of this pattern is poor performance, because number of joins we need to get full information about an object equals approximately to depth of our class in inheritance hierarchy. On the other hand resulting relational model is highly flexible (thus maintainable) and its space consumption is almost ideal.

We can observe heavy load on the root class's table.

Use of this pattern is convenient in situations, where there is significant overlap between types.

5.1.1.3 Concrete class to its own table

By this pattern we map every concrete class from inheritance hierarchy to its own table. Attributes of this class as well as attributes of all parent classes are mapped to this table. There are no links (e.g. in the form of foreign keys) between tables of parent and child classes. Situation is depicted in Figure 5. Note, that abstract classes are not mapped to database tables. Their attributes are mapped to tables of child concrete classes.

We can observe good performance of accessing single object's data, as well as of writing and updating attributes of single object. This is because we have to access only one table.

Space consumption is optimal, because we have no redundant attributes for linking to tables of parent objects and we don't waste space with attributes of child objects in tables of parent objects. Another positive aspect is, that load is balanced on all tables – there is not bigger load on root tables (tables on which root object of the inheritance hierarchy is mapped).

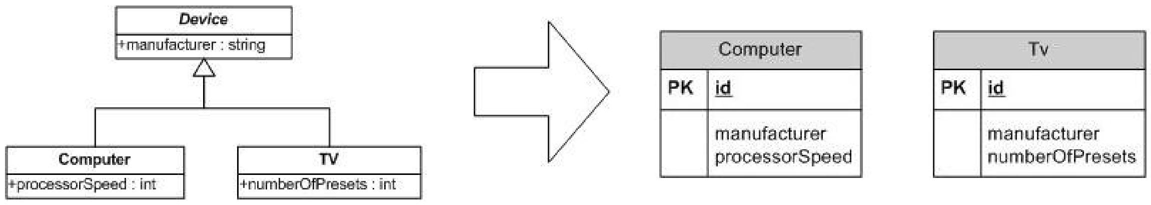

Figure 5 - Concrete class to its own table

Drawback of this pattern is efficiency of polymorphic reads, because we have to visit all the tables. We can also observe that even small change of object model causes changes in all tables to which child objects of changed object are mapped. Similar situation is e.g. inserting a new subclass into a class hierarchy. We can cope with some of these problems by use of code generators. If object model changes database table definitions as well as (polymorphic) queries are re-generated.

Use of this pattern is suitable in situations where we assume that object model will not change very often.

5.1.1.4 Generic table structure

This approach is also known as meta-data driven. In the Figure 6 we can see that we have generic structure for modeling of arbitrary inheritance hierarchy where we can model attributes of classes also. Meta-part of Figure 6 is part where we have classes such as Class, Attribute, Inheritance and AttributeType. Value is table, where attribute values are stored. AttributeType table us used for converting of attribute values stored as varchar to and from its real types.

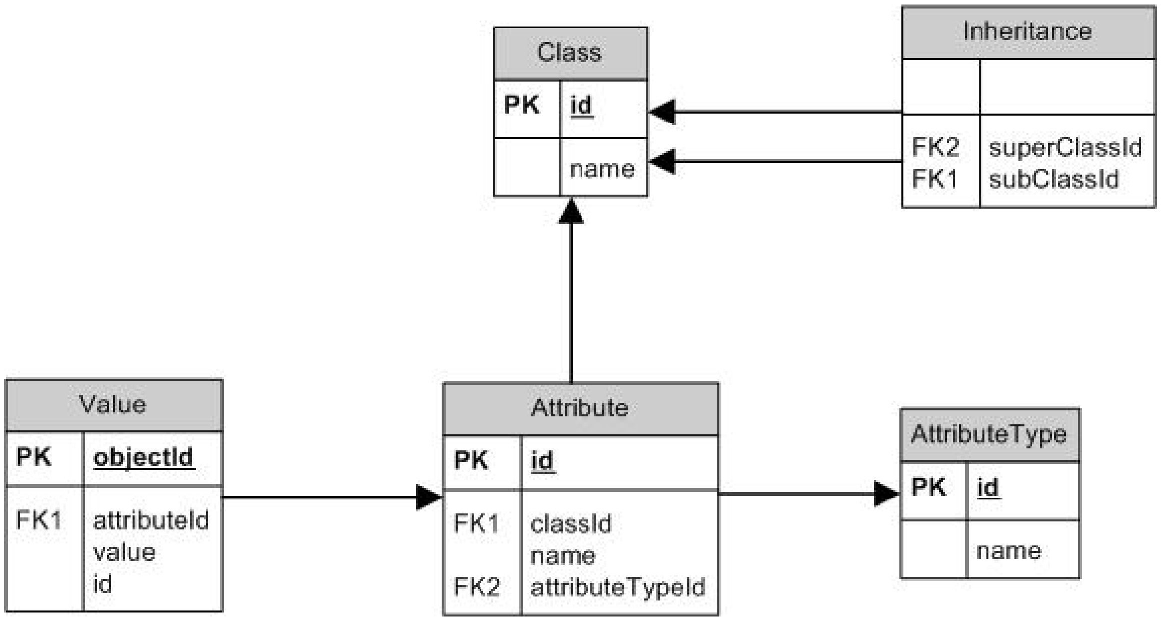

Figure 6 - Generic table structure

This pattern provides great flexibility. We can easily augment our schema to support mapping of object relationships. On the other hand this approach is very slow, because lots of tables have to be accessed in order to read or modify an object. Thus it is suitable for applications that use small amount of complex data. For more information about this approach see [AMBER].

5.1.1.5 Objects in LOBs

Idea of this approach is to save state of object to Large Object database data type. After serialization (binary or XML) we store serialized state to database table with ID of our object. Objects from whole inheritance hierarchy are stored in the same table.

Although reading or updating require accessing of just one database table, reading of LOBs is usually very slow as well as their serialization into objects. In addition, LOB isnot meant primarily for structured data – as objects are. It is not possible to query objects stored in LOBs by their attributes (using a SQL query) since the attributes are “inside” od the binary LOBs. Hence objects can be retrieved solely using their direct pointer and they can be queried only after being restored to full-featured objects. This pattern is suitable to use with graphs of small objects. This whole structure is serialized to database as a single field. For more information about this approach see [KELLER] or [FOWLER].

5.1.2 Strategies for mapping of Object Relationships

Objects can be connected together and can reference each other. These connections are called associations. Special cases of association are composition and aggregation. Associations have some attributes. We will examine these attributes and explore association types by these attributes.

We can divide object associations into groups by criteria such as multiplicity, orientation or order. We will examine all of these criteria from the view of object-relational mapping. We will not consider deeply how multiplicity, orientation or order could be implemented in object-oriented environments. Instead, we will look how we can map these to relational database.

For a moment we will deal with multiplicity of associations. Consider, we have 1:1 object relation e.g. between TV and RemoteController entities. Mapping will look as follows: we will have one table for TV and one table for RemoteController. Database table called TV will have RemoteControllerId column, which will point to id column of RemoteController database table. We will get the same result if we create tvId column in RemoteController database table. The tvId column will point to id column of TV database table. RemoteControllerId or tvId will be foreign keys. This situation is depicted at Figure 7.

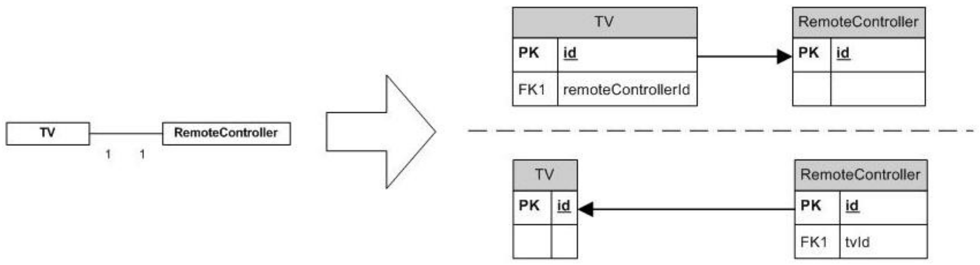

Figure 7 - 1:1 association mapping

Similar situation is 1:n multiplicity. Difference is, that foreign key column must be implemented in database table of entity, which has multiplicity of n. Suitable example could be 1:n relation between Car and Color (we suppose one car can have only one color). Car database table will have ColorId column which will point to id column of Color database table (Figure 8).

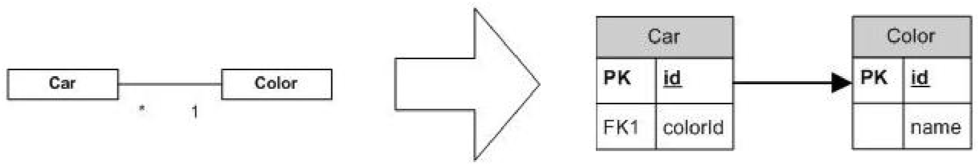

Figure 8 - 1:n association mapping

Quite different is situation with m:n multiplicity. We will need third database table to map m:n relation. This table is called association table. It has two columns: IDs of entities between which is relation we want to model by association table. One row of association table represents objects which are associated. Association table could have also other columns – if we wanted to have attributes of association defined. An example of m:n association is association between TV and Channel where name of association is ChannelPresetOnTV. Attribute of this association could be presetNumber (Figure 9).

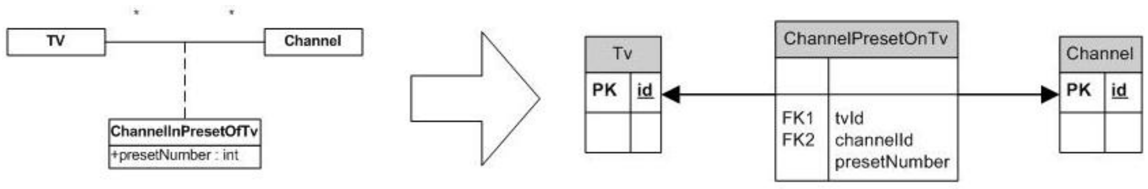

Figure 9 - m:n association mapping

Object relations can be oriented or not oriented. If relation is not oriented, object A has reference to object B, as well as object B has reference to object A, where object A and object B are related. Because placement of foreign key does not imply orientation, we have to store information about orientation in database explicitly.

Similarly if we want to map ordered collection, we will have to store information about order explicitly in database. This will require some additional costs with administration of this information. E.g. we would have to cope with situations where we have numbered records and we will have to add record between third and fourth record. One solution would be to renumber all records with sequence number bigger than three and give sequence number four to a new record. Another issue is coping with gaps in numbering that evolve after record deletions.

Mapping of aggregation

Single table aggregation

This approach is also known as Dependent mapping. As we can see on Figure 10, all attributes of aggregated objects are in database table of aggregating object. Multiplicity of aggregation relation is 1:1. Let's speak about situation where we have multiplicity c:1. This means we have one aggregating object and c aggregated objects (where c is constant). Database table of aggregating object would have c sets of attributes of aggregated object – one set for one aggregated object.

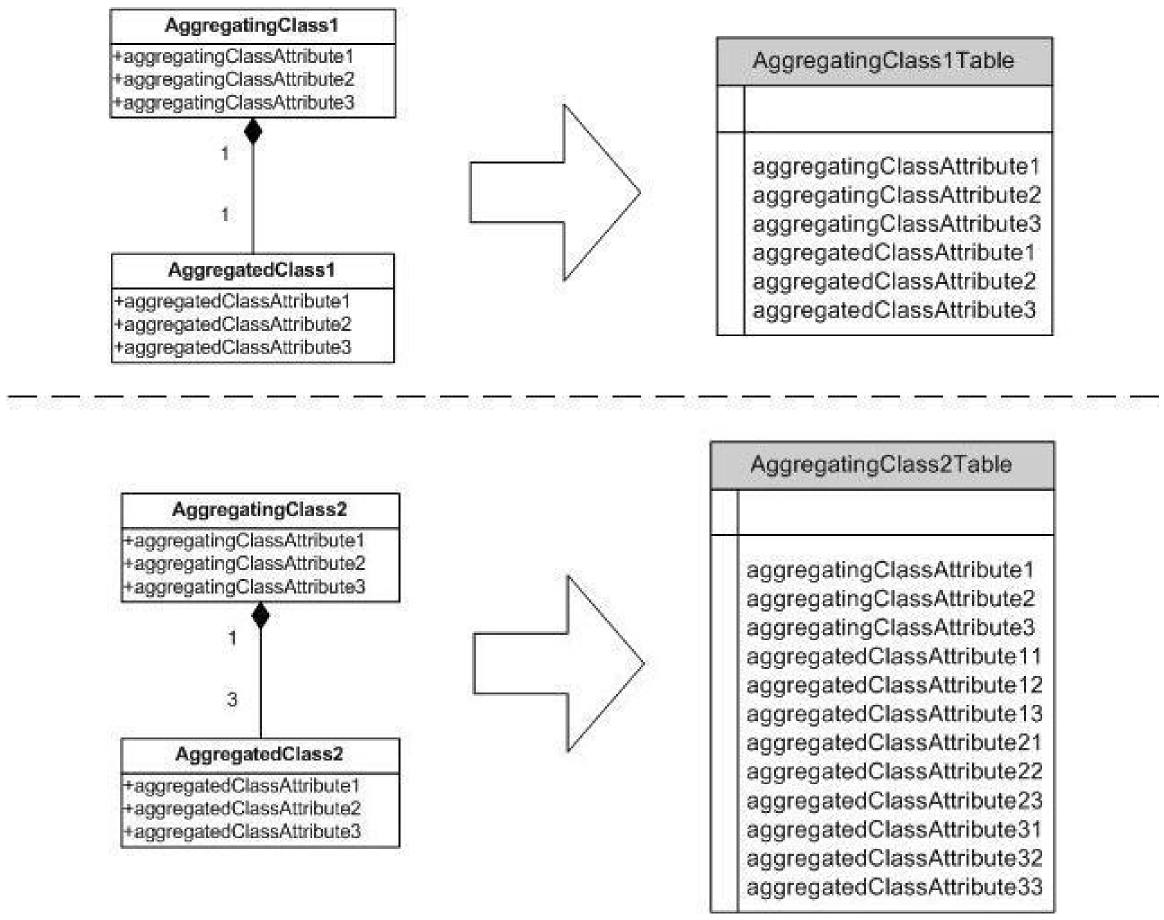
AggregatingClass1
+aggregatingClassAttribute1
+aggregatingClassAttribute2
+aggregatingClassAttribute3
1
1
AggregatedClass1
+aggregatedClassAttribute1
+aggregatedClassAttribute2
+aggregatedClassAttribute3
AggregatingClass1Table
aggregatingClassAttribute1
aggregatingClassAttribute2
aggregatingClassAttribute3
aggregatedClassAttribute1
aggregatedClassAttribute2
aggregatedClassAttribute3
AggregatingClass2
+aggregatingClassAttribute1
+aggregatingClassAttribute2
+aggregatingClassAttribute3
1
3
AggregatedClass2
+aggregatedClassAttribute1
+aggregatedClassAttribute2
+aggregatedClassAttribute3
AggregatingClass2Table
aggregatingClassAttribute1
aggregatingClassAttribute2
aggregatingClassAttribute3
aggregatedClassAttribute11
aggregatedClassAttribute12
aggregatedClassAttribute13
aggregatedClassAttribute21
aggregatedClassAttribute22
aggregatedClassAttribute23
aggregatedClassAttribute31
aggregatedClassAttribute32
aggregatedClassAttribute33

Figure 10 - Single table aggregation and c:1 aggregation

As we need no database join operations, this approach brings good performance. On the other hand, maintainability of such solution is very poor. Imagine that value of above mentioned c changes. We would have to change database schema – add new columns to table of aggregating object. If structure of aggregated objects changes, we will have to change all c sets of its attributes. Pattern presented below brings more flexibility into this situation.

Foreign key aggregation

When using the Foreign key aggregation pattern, we create separate database table for aggregated object type. Of course we have database table for aggregating object type. We add foreign key attribute to this table. This attribute will point to table of aggregated object (Figure 11).

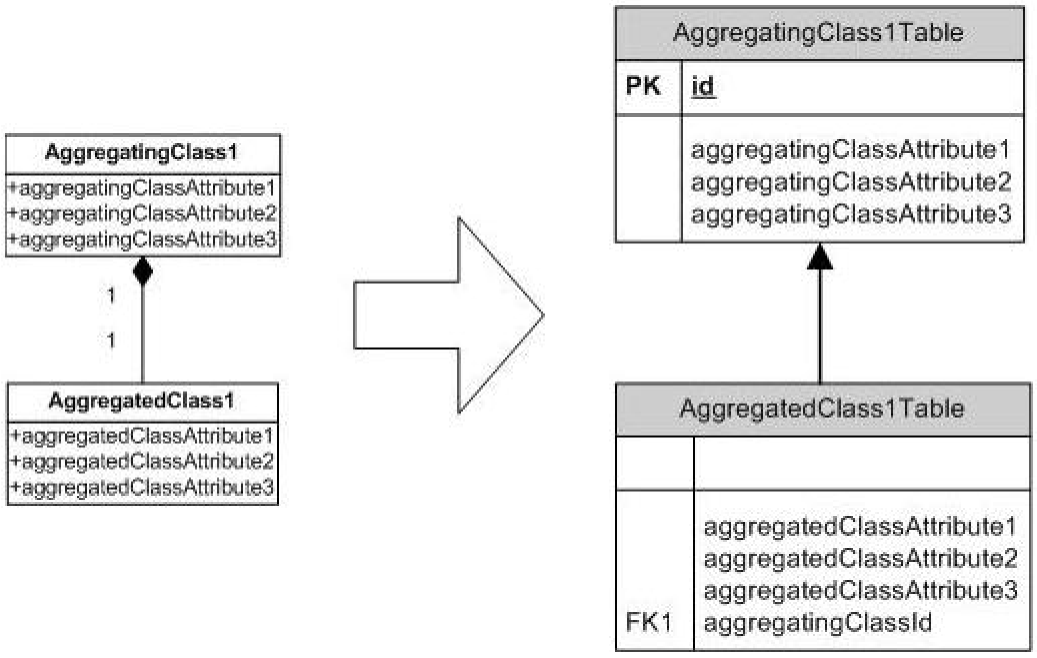
AggregatingClass1
+aggregatingClassAttribute1
+aggregatingClassAttribute2
+aggregatingClassAttribute3
1
1
AggregatedClass1
+aggregatedClassAttribute1
+aggregatedClassAttribute2
+aggregatedClassAttribute3
AggregatingClass1Table
PK
id
aggregatingClassAttribute1
aggregatingClassAttribute2
aggregatingClassAttribute3
AggregatedClass1Table
aggregatedClassAttribute1
aggregatedClassAttribute2
aggregatedClassAttribute3
FK1
aggregatingClassId

Figure 11 - Foreign key aggregation

Foreign key implies need of join if we want to get aggregating and aggregated object, but entities are more maintainable. We don't have to change aggregating object's database table when aggregated object's structure changes. Another advancement is that we can reference aggregated objects from other tables that from those to which aggregating object is mapped.

5.1.3 Strategies for mapping of static fields

In this part we will deal in short with mapping of static (also known as class-scope) fields. These fields are common for all instances of class, so handling of mapping is quite different from handling of instance fields (which is straightforward). There are few approaches how to cope with this problem.

One of the approaches is to create database table, where columns are named by names of static fields and are of the types of appropriate static fields. Another approach could be to create generic table with columns like these: ClassName, VariableName, Value. After accesing of field value we will have to cast, because all the values are stored as varchar.

5.2 Performance related patterns

While trying to design and implement an object-relational mapper with use of mentioned patterns, the designer will probably have to cope with performance issues. This is significant especially when target database maintains huge amount of data which are highly structured (so much joins is needed to get full information). If we talk about relational databases, huge amount of data is common situation. We will use principles of known approaches to cope with performance issues. These principles are little adjusted to fit needs of ORM-like applications.

At first we will look at denormalization – technique which can rapidly speed-up selecting of data. Other techniques for speed-up of reading we will deal with are caching and lazy loading. Then we will look how we can optimize updating of data.

5.2.1 Normalization and denormalization

When designing schema of relational database designers usually want to remove redundancy and enforce integrity of data. This can be accomplished by normalization of database schema to normal forms (3NF or BCNF). Normalized schema has some nice attributes when we talk about redundancy, or integrity. But when we talk about performance, its behavior isn't as good as its not-normalized equivalent could have.

Imagine two tables: Invoice (with columns ID, description, name, date, …) and InvoiceDetail (with columns ID, InvoiceID, description, name, ...), as is shown in Figure 12. If we wanted to see invoices with all their details, we will have to join these two tables. Not-normal equivalent of this simple schema could contain one table named e.g. InvoiceWithDetails. Table InvoiceWithDetails will have columns such as InvoiceName, InvoiceDescription, InvoiceDetailName, InvoiceDetailDescription., …. Maintenance of the second schema is difficult and error-prone. Inconsistent data can be created if table is not updated properly. On the other hand, second table has much better performance, because we don't have to do any additional joins.

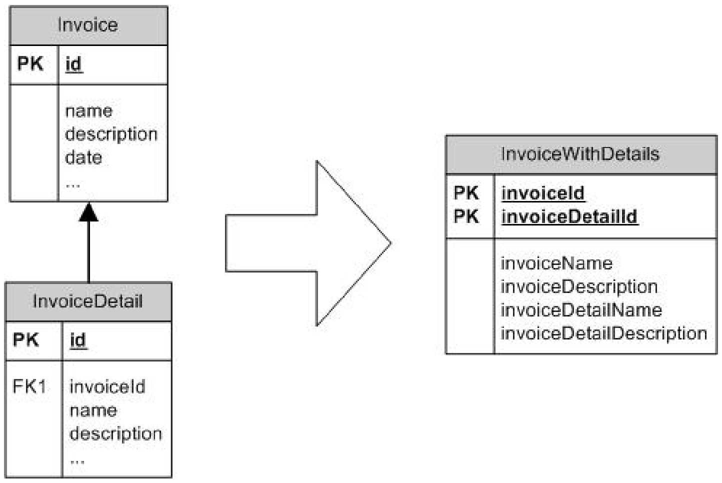

Figure 12 - Denormalization

Another example of situation where denormalization is suitable are aggregations. Instead of counting of complex aggregations at runtime, these aggregations can be counted once for a period of time and stored to special table. This is relatively easy to do, when data don't change very often.

By denormalization we will understand adjustment of database schema, where adjusted schema will contain all tables that normalized schema contains plus tables with structure that doesn't meet normalization criteria. These added tables are adjusted for needs of our application. There are for example result rows of join performed on multiple tables stored here. Goal is to prevent making joins from application at run-time. These tables are updated by synchronization procedures and not directly by application. When application wants to update any database row, it updates row of table from original database schema. In our example when application wants to show all invoices with their details, it will make a SQL select from adjusted InvoiceWithDetails table without any joins. To update InvoiceDetailName attribute of InvoiceWithDetails table, our application will update Name attribute of appropriate row in InvoiceDetail database table.

We can choose from multiple synchronization models. One of them is scheduled job at RDBMS server, which periodically updates denormalized table. Another is to make a SQL trigger for table which we are about to update (InvoiceDetail). This trigger fires an update on denormalized table (InvoiceWithDetails), after update of table which is part of normalized database schema (InvoiceDetail).

It is role of database schema designer to consider use of denormalization. If we talk about avoiding of making joins or aggregations at run-time it is very important to know how often affected data are being updated.

5.2.2 Caching

We can imagine caching in context of object-relational mapping as a possibility to store objects read from database for next use. We can speak of local cache and global cache. As local cache we can imagine a cache, where objects read from database are stored in cache after reading. Let application developer throw away reference to these objects. Later he might want to get them from database again. If some conditions are met, no database call is needed. He simply reads object from the cache. Global cache is application server-scope cache. This means that if one user has read some data from database, these data are cached. When another user requests the same data, these are simply read from cache and no database call is needed. It is important to note, that decision whether to read from cache or from database as well as whole processing of caching is transparent for application developer. Another examination of caching is out of scope of this work.

5.2.3 Lazy loading

Lazy loading is a technique, which avoids reading of sets of data, which are not necessarily needed at the given moment.

We will explain this principle on situation where we are designing the way how our Business object is going to be filled with data from relational database. Suppose a trivial situation, where our Business object has just some fields and holds no references to another Business objects. Suppose that class, our object is instance of, has name Person. It has fields such as Name, Surname, DateOfBirth and Photo, where first three fields occupy only small space in memory. Photo can be potentially huge and it is not very efficient to read everytime object state is loaded from database. Instead, just small-sized data type fields are loaded. Huge fields are loaded on demand, where on demand means that client programmer wants to access this field via property. When the property is read for the first time, it takes a moment, of course. Next reads are quick, as field is already in memory.

Although the previous example was maybe nice, huge fields are not right problem. Problem can be big amount of database calls. When we are filling our Business entity with data it will probably cost one call to database. But sometimes we might want to fill related business entities as well. Some Business entities have lots of related entities which we do not need maybe. Techniques of lazy loading can help us to resolve this situation. By correct combination of lazy loading and caching we can achieve very nice performance.

Several techniques can be used for implementation of lazy loading: lazy instantiation, virtual proxy, value holder and ghost. Lazy instantiation is the simplest approach where we simply set value of object that has to be processed lazy to null when instantiaing it. When trying to access this object and its fields we check if it is null and if so, we load its state from database. We have to resolve the situation where null is a legal value. Problem of this approach is that Domain objects are tightly coupled with data access logic – when object is null, we simply read its state from database. Virtual proxy approach solves this situation. Instead of our business object we have object that have same interface and holds reference to our business object. For more information about virtual proxy design pattern, see [GoF]. As we can have more than one proxy for one business object, we can get some nasty object identity problems. Value holder approach solves this issue. With ghost approach we store just id of the object in it and load other information on demand. When client programmer request any concrete field we load all the fields, or we can create groups of fields that are used together and load whole group when one field from the group was requested. For more detailed information about lazy loading see [FOWLER].

5.2.4 Delayed Updates

Until now we have investigated ways how to optimize reading from database in this part. Here we will deal with problem of updating/inserting. As we have said, number of calls to database is very critical issue and thus it would be great to find way how to minimize it.

Unit of work is pattern for object-relational mapping that solves this problem. It is a class which holds references to business objects and remembers their state. It maintains three lists. In first it remembers changed objects, in second inserted objects and in third deleted objects during business transaction. Important is, that all changes can be accomplished in one call to database. Unit of work simply traverses all its lists, generates appropriate SQL inserts, updates and deletes and send it as a batch to database. We won’t deal with ways how SQL batch can be sent to database here.

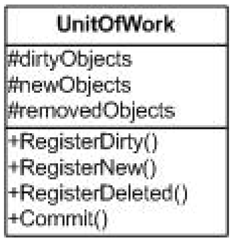

Figure 13 - Unit of work

As we can see at Figure 13, Unit of work has methods for registration of objects to its lists and method `Commit()` which generates SQL statements and send them to database.

5.3 Configuration

Although transparency and automation are features required from ORM frameworks, certain flexibility is also required. In certain situations developers/designers aren’t satisfied with black-box mapper that creates database schema from Domain model. For example, when application have to meet special performance requirements, application developer/designer may want to affect how objects are mapped to relational database tables. On the other hand, laborious configuring of routine and trivial mapping tasks is often tedious. That’s why adequate proportion of flexibility and automation should be met.

Usual form of configuring of mappings is declarative configuration. It is not usual for OR mappers to have ability to configure mapping imperatively at run-time. It is a question, whether functionality like this could be needed. Another question is, whether it is a good practice to change mapping at runtime. As database schema and object model should be stable during application life, mappings should be also stable. We don't consider database, or application components updates, which is a special case. That's why we will consider just declarative forms of mapping definition/configuration.

Pattern related to this topic is Metadata mapping. Reader can get more information about this pattern in [FOWLER]. We will also deal with this pattern and its implementation in next part.

Now we will deal with ways, how to store configuration information. We have two basic possibilities. We can store mapping information to external configuration file. Favorite choice is XML file today. Another option is to store mapping information directly in the source code by means of declarative constructs of language. Third option is to store mapping information in database, but we won't deal with this option.

5.3.1 Attribute-oriented programming and equivalent approaches

Attributes are a convenient way, how to declaratively attach meta-data information to a class respectively modify behavior of a class, method, assembly, or another construct. Attributes are used both in .net framework and in Java. .net framework's C# language syntax of using attributes can look as follows:

```
[PersistentClass("User")]
public class Person{
      [PersistentProperty("Name")]
      public string Name
      {
            get
            {
                  // property implementation here
            }
            set
            {
                  // property implementation here
            }
      }
```

```
        [PersistentProperty(“DateOfBirth”)]
        public DateTime DateOfBirth
        {
              get
              {
                    // property implementation here
              }
              set
              {
                    // property implementation here
              }
        }

        [TransientProperty]
        public int Age
        {
              get
              {
                    // calculate age from DateOfBirth
              }
        }
        // rest of class implementation here
}
```

When persistence framework tries to find out, to which database table class Person has to be mapped, it checks attributes of Person class. It finds out, that this class is marked by `PersistentClass(“User”)` attribute. Value of parameter of this attribute is “User”, so our persistence framework knows now that class Person has to be mapped to database table User. Also some properties are marked by attributes. Public properties Name and DateOfBirth are marked by `PersistentProperty` attribute with appropriate parameter. This parameter says, to which column given property will be mapped. Age property is marked as `TransientProperty`. This means that this property is not mapped to database. At run-time, age is computed from DateOfBirth property when client programmer wants to know this information. This property is implied and thus not stored in RDBMS.

Also other information about classes/properties can be passed to persistence framework through attributes.

Drawback of using attributes is, that every time we want to change mapping, we have to change source files, thus recompile them and redeploy/redistribute them. This can be significant handicap in environments, where mapping changes often.

DataObjects.NET is an example of persistence framework that uses attributes for declaring of Meta data.

Although concepts of attribute-oriented programming weren't supported in Java, e.g. Hibernate framework has ability to declare mapping information directly in the source code. XDoclet tool was used to perform this. XDoclet is a code generator that uses JavaDoc tags and other Meta information to generate code (in our case XML configuration file). So application developer declared mapping information directly in the code with use of JavaDoc tags. Now, Java has support for attributes from version 5.0 of JDK. Attributes are called annotations in Java.

5.3.2 Configuration files

Configuration files containing mapping Meta data are also popular and widely-spread technique. Usual format for configuration files is XML with appropriate XML schema or DTD again that configuration document can be validated.

Configuration file approach doesn't suffer from lacks of attribute approach. Configuration file is not part of assembly, so we can make changes on it independently from assembly. At worst, we will have to restart application when the configurations file changes.

Simple XML configuration file can look as follows:

```
<mappings>
      <classMapping className=”Person” dbTableName=”User”>
            <propertyMapping propertyName=”Name”
                             dbAttributeName=”Name” />
            <propertyMapping propertyName=”DateOfBirth”
                             dbAttributeName=”DateOfBirth” />
      </classMapping >
</mappings>
```

This example shows one of the XML configuration files equivalent to example in part about configuration attributes.

Example of persistence framework that uses configuration files for mapping is Hibernate.

5.3.3 Combination

By suitable combination of two above approaches, we could get flexible framework. When using this hybrid approach, we will put configuration information that is supposed to change when object model changes to attributes. Configuration settings that are changing often and independently from object model will be placed to XML configuration file.

6 Benchmarking

In next chapter we will try to compare some of the well known persistence frameworks. We will show benchmarking techniques here, as well as mention and explain comparison criteria here. As it is difficult (if not impossible) to judge which persistent framework is the best, we will just compare their behavior from (more or less) distinct aspects. Resulting information could be that framework A is good at one aspect and framework B at other aspect. Anyway we will not tell that one framework is better than another in general. Developers searching for suitable framework can choose according to their (and project) requirements on such framework.

For each framework we will provide a brief description of it at first. This description will contain basic information about this framework such as which platform it is suitable for or which database servers are supported. Very important will be information about licensing and its flexibility. We can also mention some special features of this product, if any. Reader should get basic picture of given framework from this summary.

In the next part we will look at architecture of this product. We will be interested in how this framework works and how it is designed. For more information about how we will explore architecture of frameworks, see part Architecture later in this chapter.

At the end, when we describe each framework and compare architecture of these frameworks, we will provide summary, where each framework is ranked from different aspects. This summary will have form of table where on the horizontal axis names of frameworks will be. Feature / Criterion will take place on vertical axis. In conjunction of column with framework A and row with Criterion B will be number expressing measure of how framework A implements feature B. Meaning of this number will be expressed later in this chapter. This table represents summary of characteristics of given frameworks and will be part of conclusion. For more information about this table and meaning on numbers in it, see part Benchmarking criteria later in this chapter.

Note that one framework is special in our benchmarking. DataObjects.NET is described in very detailed way. This is because reader should have better understanding of at least one framework. He should also understand benchmarking criteria. That’s why we discussed each criterion for this framework.

6.1 Architecture

In this part we will try to investigate how the architecture of selected ORM frameworks looks. This is a difficult task, because not all of them provide source code and even they do, analyzing source code is not a trivial objective. Anyway, it is still possible to infer how architecture looks from behavior of the framework and from its API.

We will be interested in several aspects of ORM tools. We will investigate how structures from Domain model can be mapped to relational databases or which patterns can be used for mapping. For example we could have a choice if we use Hierarchy to a single table or Each class to its own table or other pattern. We will be interested in how associations can be mapped. This means, if Foreign key mapping or Association table mapping can be used. For more information about these patterns see part 5.1 or [FOWLER].

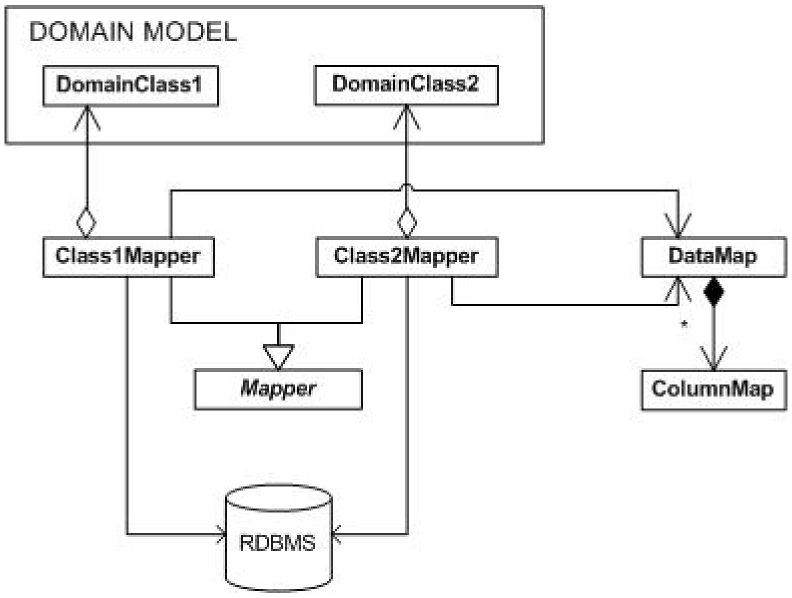

Figure 14 - Architecture of EIS with mapper classes

Now we will spend few words about Mapper classes. We will explain this on Figure 14. On right we can see two classes: DataMap and ColumnMap. These classes contain meta-data information about mapping: which class has to be mapped to which table and which field to which column. Note, that this is very simple example. Approximately in the middle we can see the mapper classes. These classes are synchronizing state of objects with state of database. There are two possibilities how association between mapper and its domain class can be realized: reflection and code generation. Reflection is functionality available in Java as well as in .net. With reflection we are able to obtain information about loaded assemblies and types defined in these assemblies. You can create instances at run-time and call methods on these instances. Drawback of this approach is that it is slow and hard to debug. Reflection is used in methods of mappers that work with instances of Domain objects. Mapper at first reads information about its

DomainClass from DataMap and ColumnMap classes. Then it knows names of fields and everything it needs about its DomainClass. An alternative for reflection is code generation. Framework at first looks at meta-data information to DataMap and ColumnMap and simply generates methods for working with Domain objects. This code has to be compiled and deployed with rest of application. We will be also interested, if framework uses configuration files, or attributes in source code for gathering meta-data information about mapping.

We will also examine how given framework copes with Lazy loading and Delayed updating (Unit of work pattern).

We can see that there exist different sources of information about the model. We can start with object model and generate database model and mapping info from it. Also we can start with a database table and generate object model and mapping info from it. Third option is to create (e.g. XML) configuration file, which holds information about mapping. With this approach database and Domain model can be relatively independent.

It should be clear, that first option is good for applications that are written from scratch. We simply create our object model and let database schema be generated. This approach is not suitable, when database and object model are maintained by different teams. Thus team that is maintaining object model doesn't have full control over database model and vice-versa. In situations like this it would be great to have possibility to generate object model from database. Another situation where this functionality would be welcomed is integrating of legacy systems, e.g. building a new interface over database backend of legacy system.

6.2 Benchmarking Criteria

In this part we will describe criteria from the summary table from the last part of this work. If dealing with any criteria will not be relevant for any reason, we won't consider it. For each criteria we will score how given framework fulfills given criteria. We will score each framework for each criterion from zero to three points with following meaning:

- 0 points: feature not supported
- 1 point: feature supported, but considerably limited
- 2 points: feature implemented
- 3 points: given framework fully supports feature following from criterion

6.2.1 Transparency

By our informal definition, transparency is amount of effort of software developer to deal with persistence logic. Because the goal of each ORM framework is to minimize development effort on DALs, we will consider this criterion as crucial. For more information about Transparent Persistency, see part 4.

6.2.2 Documentation & Support

In this part we will examine how given framework is documented. We will also examine how difficult is it to find solution for not very common problem. This means that we haven't found solution in primary documentation. We will try to search for FAQs or similar resources for community, respectively from community www pages. Another aspect taken into consideration could be response times on reports of problems. As contacting of framework-developing organization is out of scope of this work we will acquire information from supplier guidelines supplied to public, if available.

6.2.3 Dependencies, Supported platforms, Portability

While deciding which ORM framework will be used for development project crucial question is, whether ORM framework can be used on desired platform. By platform we will mean operating system or development platform. We will consider just Windows and Linux/Unix operating systems. Widely spread development platforms of today are Java and Microsoft .net. That's why we will concentrate on this two. If the framework needs any third-party component, we will consider this situation, too.

Another aspect considered is how many and which database server given framework supports.

Basic principles of assessment are:

- Framework can not have 3 points if it doesn't support any one of these RDBMSs: MS SQL, Oracle, IBM DB2
- Framework can not have 3 points if it doesn't cope with persistence of either Java, or .net objects

Motivation for these two rules is that above mentioned technologies are widely spread. Although e.g. MySql is very popular, it is not suitable for middle-sized, not even huge projects. And large projects are those that need support of persistence frameworks.

6.2.4 Maintainability of model

By maintainability we will mean maintainability of Domain model, from which relational database schema can be generated. Fewer operations are required when Domain model changes, more maintainable model is. E.g. maintenance of model where only operation needed when model changes is to change class definition is better than of model, where we have to change class definition and then change meta-data xml configuration mapping file. Maintainability falls off with every additional operation needed for change to take place.

6.2.5 Usability

By usability we will understand easiness of using framework and ability of medium-skilled development team to use given framework with assumption they have never used this framework before. This means how long does it take for team members to get familiar with given environment and to effectively use it. Of course, more features usually mean more time to get familiar with given product. On the other hand users (also users of persistence frameworks) just might want to use subset of whole feature set. That’s why it is important for frameworks to be usable (by our understanding) even they have huge feature-set.

6.2.6 Configuration vs. automation

In part Configuration vs. automation we will look at manner, how mapping is configurable, respectively automated. In this situation, mapping means set of rules for translation from Object model to relational database model. More automated generation of relational schema is less configurable mapping is. In extreme case, where process is fully automated, software developer has no mean to change resulting database schema. Thus we require from this schema to be efficient, normal (in terms of normal forms) and readable. By readable we mean that another software developer team should be able to build an application on this database. Reporting from this database might be required, too.

We will also deal with techniques of configuring the mapping process (if this process is configurable). One possibility is using of custom attributes, another is using of configuration files. Advantages and drawbacks of this approaches are described in part 5.3.

6.2.7 Constraints on design of Domain model

Many persistence frameworks have specific requirements on design of persistent objects in Business Logic Layer. Examples of requirements like these are those of DataObjects.NET: every persistent class must be defined as abstract and must be derived from DataObject class. In this part we will examine, if these requirements aren’t very restrictive or redundant.

7 Comparison of selected ORM frameworks

Here we will spend a few words about each framework we will deal with. At first we will look at DataObjects.NET. It is relatively new tool for .net framework. We will deal with this framework in a more detailed way than with other frameworks. Next framework will be Hibernate which has been (and still is) very popular in Java world. It expands also in .net world now. Then we will look at TopLink framework which is kind of cult and it is part of Oracle fusion middleware suite now. Last one will be well known standard called Java Data Objects (JDO), which is known in Java world.

7.1 DataObjects.NET

DataObjects.NET is a persistence framework written in .net framework designed for software projects written in .net framework.

When working with DataObjects.NET we will create our Domain model at first. In our model persistent classes have to be abstract – this is design constraint required by DataObhjects.NET. If we want to define real abstract class we have to mark our class with DataObject.NET's [Abstract] attribute. Abstract proxy for this class will be created. We will speak more about proxies at the moment. We will mark our classes/properties with other appropriate attributes as well – to enforce DataObjects.NET framework to persist given objects in desired way. If our object model changes or we create new one, we will want underlying database to change with respect to changes in object model, respectively changes in meta-data. Figure 15 depicts how DataObjects.NET works when it is requested to "refresh" database structure. When `Domain.Build()` method is called it builds its representation of object model at first. From this object model it builds proxy classes (developer actually works with these classes). Another artifact built from object model is database model. This database model is compared to database model extracted from existing database. Update scripts folowing from differences between these two database models are generated then. At the end, database is updated.

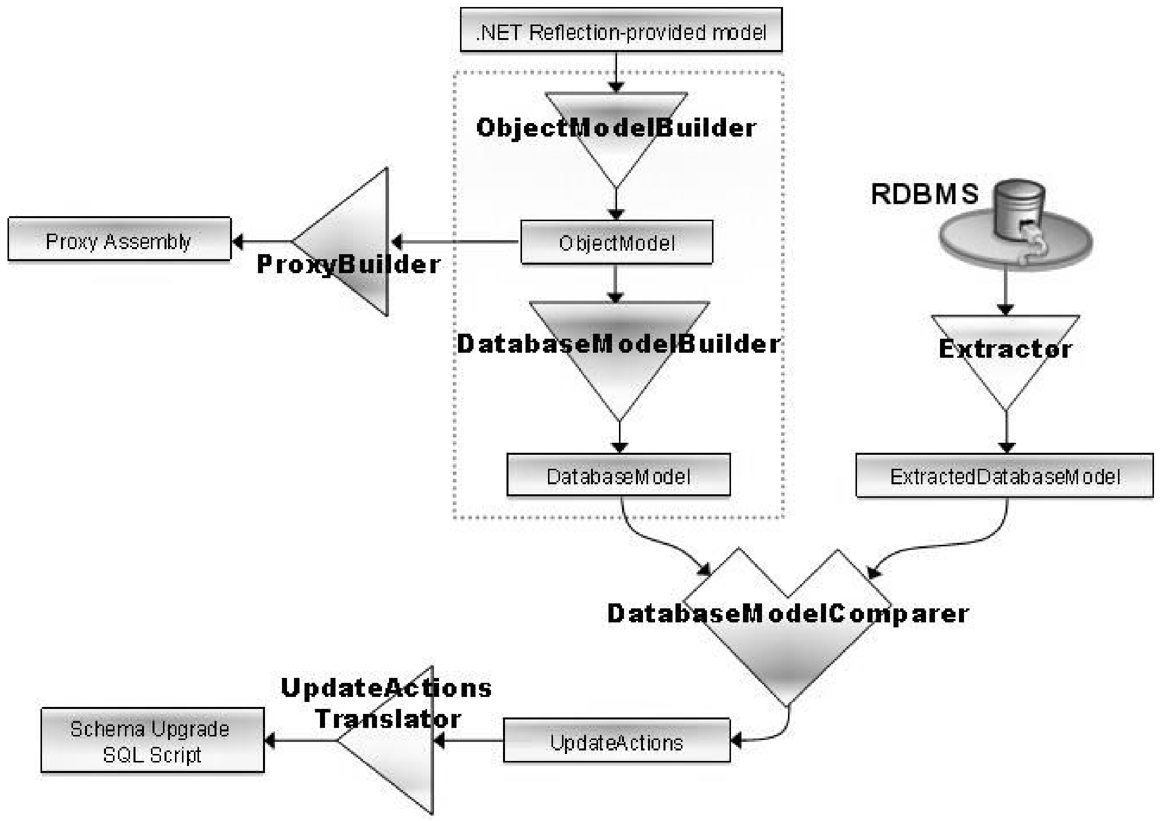

Figure 15 - DataObjects.NET architecture

In Figure 16 we can see our Business Logic Layer in context of services provided by DataObjects.NET.

DataObjects.NET provides just one-way mapping. This means that it is able to map objects to tables, not vice-versa. If we wanted to integrate database of legacy system, we should use another mapper.

Vendor of this framework provides more editions of product (Express, Standard, Advanced, Professional, Enterprise, Unlimited). These editions differ in attributes such as these:

- Count of persistent types
- Count of database tables
- Size of cache
- Periodicity of updates
- Forms of support
- Source code availability
- Number of developers allowed to use product
- Price

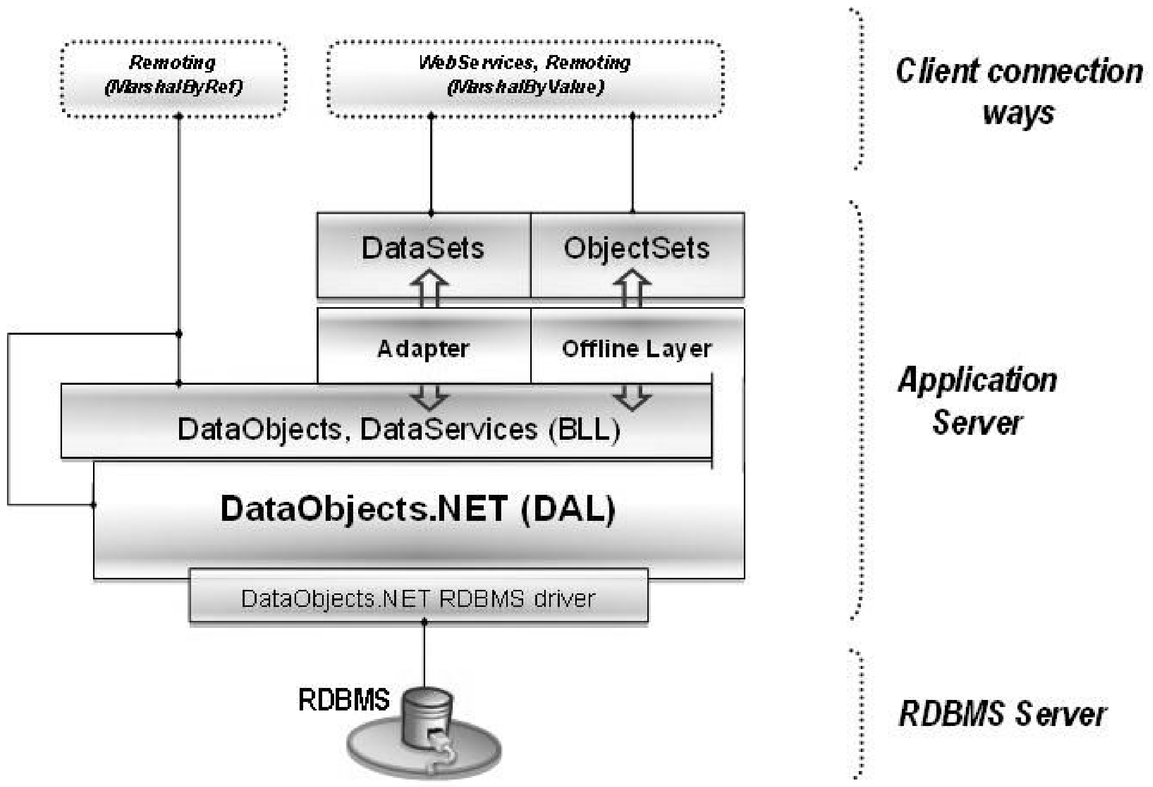

Figure 16 - DataObjects.NET in context of EIS

7.1.1 Code Snippet

7.1.1.1 Persistent type definition

```
public abstract class Person: DataObject
{
      public abstract string Name {get; set;}
      public abstract string SecondName {get; set;}
      public abstract string Surname {get; set;}
      public abstract int    Age {get; set;}
      public abstract string Info {get; set;}

      [NotPersistent]
      public virtual string FullName {
            get { return Name+" "+SecondName+" "+Surname; }
      }
}
```

We can see definition of persistent type person with some persistent and one not persistent property. Notice, that persistent classes and properties must be declared as abstract from architectural reasons.

Class that inherits from DataObject and all its properties are considered as persistent by default. As can be seen in snippet, not persistent property is marked by `[NotPersistent]` attribute. This property is implied by Name, SecondName and Surname persistent properties. It is redundant and thus not persistent to database.

7.1.1.2 Persistent type usage

```
long instanceID = 0;
using (Session session = new Session(domain)) {
      session.BeginTransaction();
      // Let’s create our persistent instance
      Person p =  (Person)session.CreateObject(typeof(Person));
      // Instance is already persistent to the storage now
      // Let's set the persistent property
      p.Age = 2;
      // And read the instance ID
      instanceID = a.ID;
      // And finally commit our work
      session.Commit();
}
```

To say something about this code snippet it is needed to mention some information about design of this framework.

Domain type represents connection to one concrete database on one database server of its type. For example it could be database Pets on database server CorporateDataStore running on Oracle.

Another important type is Session. We can have multiple sessions for one Domain. Session represents one connection of one client application to database. Session provides instantiation of persistent objects, transaction processing and many more functionalities.

In our example we create session object from domain object at first. By using of session we manage transactions during whole example. Session is also used for creating of persistent object of type Person. Session serves as a factory here and actually creates Person’s concrete descendant's instance. Then age is assigned to our new instance of person and object is saved to database by `session.Commit()` call.

7.1.2 Benchmarking

7.1.2.1 Transparency

Additional code needed to manage persistence in DataObjects.NET can be summarised into these groups:

- Attributes for configuration of mapping
- Domain management (connection to database, management of models and proxies, DataObjects.NET's runtime services)
- Session management (transactions, caching, querying, ...)

These kinds of statements are necessary for basic management of persistence. If developer doesn't want to use advanced features of DataObjects.NET they are also sufficient for management of persistence. As management of connections and sessions is necessary condition while working with database we consider this as a big amount of transparency. This product doesn't force any not needed code.

3 Points

7.1.2.2 Documentation & Support

DataObjects.NET framework is very well documented. Following documents are available with product:

- Large pps presentation with almost all important features explained
- DataObjects.NET class library reference – html help 2 version (HxS)
- DataObjects.NET class library reference – (Chm) version
- Manual
- FAQ
- Other documentation such as ROI Calculation, benefits and features list, Revision History, license agreement
- Forum (depending on purchased edition with response time depending on purchased edition)
- E-mail support (depending on purchased edition with response time depending on purchased edition)

3 Points

7.1.2.3 Dependencies, Supported platforms, Portability

DataObjects.NET is a .net assembly. Thus it requires .net framework installed on the target machine. Supported versions are 1.1, 2.0, or Mono. Mono support is in alpha stage currently. Mono support enables DataObjects.NET to be used on other than Microsoft platforms.

List of supported database servers with their additional requirements (if any) follows:

- MS SQL 2000/2005, MSDE 2000
- MS Access
- Oracle; Oracle client should be installed
- Firebird; FireBird Client and FireBird .net data provider should be installed
- MaxDB/SapDB; MaxDB/SapDB ODBC driver should be installed

We can evaluate the fact that IBM DB2 RDBMS is not among supported servers as a drawback, because DB2 is very popular database server and many customers require using this server in software solution for them.

2 Points

7.1.2.4 Maintainability of model

If we want to change object model, all we need is to change definition of persistent abstract class in source code and/or change its attributes (when we want to persist specific property as indexed e.g.) in a desired way.

To make changes to take effect it is needed to rebuild our domain by calling instance `Build()` method on our domain object. This call will cause generation of object model and generation of database model from object model. Then framework compares newly generated database model with existing database model and generates update script to be run on target database.

Another important step in `Domain.Build()` method is generation of assembly with new proxy classes. These classes are descendands of abstract persistent types with which we actually work.

3 Points

7.1.2.5 Usability

DataObjects.NET is an example of persistent framework that is usable (by our definition – see part 6.2.5) although its features set is rather huge. Our experience is that we were able to work with this framework while using just basic features. As new requirements (given by project) appear, there is no problem to study features (if available) of this framework and use them in development project.

3 Points

7.1.2.6 Configuration vs. Automation

We can configure mapping by use of .net attributes. We can use attributes like these:

- `[Persistent]` \ `[NotPersistent]`
- `[Nullable]`
- `[SqlType]`
- `[Length]`
- `[Indexed]`, `[Index]` – for setting properties as indexed in persistent data store. Index can be Unique or Clustered.
- `[ShareAncestorTable]`, `[ShareDescendantTable]` class attributes – by use of these attributes we are able to force framework to implement Hierarchy to a single table pattern (see part 5.1.1.1) or Each concrete class to its own table pattern (see part 5.1.1.3) for mapping of inheritance. Note that default behavior (when we don't use this attributes) is Each class to its own table (see part 5.1.1.2).

We can observe absence of possibility to define mapping by business layer independent configuration file. Every time we want to change mapping, we also have to change and recompile class library with persistent class definitions.

0 Points

7.1.2.7 Constraints on design of object model

When designing an object model, we have to keep some conventions. In the code snippet provided earlier we can see that each persistent class has to be derived from DataObject class and has to be abstract. We aren't constrained in considerable way, because we are still able to make our inheritance hierarchies. Everything we have to do is to make roots of inheritance trees in our inheritance forest derived from DataObject. Actually variable with which we work is declared as this abstract type, but it is instantiated by factory. Actual type of our variable is concrete descendant of mentioned abstract type. This descendant is proxy – it means that it is looking after communication with our database backend via persistence layer.

2 Points

7.1.3 Special Features

DataObjects.NET persistence tool provides lots of specific features. For example it enables persistent objects to be serialized. When object is serialized, developer is able to expose this object via e.g. .net Remoting. .net Remoting provides similar functionality in .net, as RMI in Java.

Another interesting feature DataObjects.NET provides is NTFS-like security system. It enables you to define allowed\denied permissions to access objects.

7.1.4 Architecture

DataObjects.NET uses runtime reflection, but just while running a `Domain.Build()` method. We have spoken about this method earlier. When this method is called, internal representation of object model is created. Then proxy classes are generated. Proxy classes are concrete classes that inherit from abstract persistent classes developer defines. As these classes are generated, working with them is fast (if we compare it to pure-reflection approach).

In this framework we can define which of the mentioned patterns to use when mapping inheritance. We configure this with use of .net attributes. We can use `[ShareDescendantTable]` attribute, `[ShareAncestorTable]` attribute or no attribute on any class. If we use no attribute, Each class to its own table pattern will be used. We can adjust this behavior on any subclass with use of `[ShareDescendantTable]` attribute. Then, Each concrete class to its own table pattern will be used for mapping. Finally, with use of `[ShareAncestorTable]` attribute we will force framework to use Hierarchy to a single table pattern.

DataObjects.NET supports one-way, as well as mutual associations, where association ends can have 1:1, 1:n or m:n multiplicities. We can use `[Contained]` attribute to say, that aggregation is Composition – aggregated object makes sense just in context with aggregating object.

This framework as well as lot of other frameworks has ability to delay updates by use of transactions. By use of Session object we fetch persistent objects from database. If we are about to change objects, we can start transaction. After we have changed, deleted or created new persistent objects we can commit or rollback this transaction with use of Session object – Session has methods for starting, committing and rollbacking a transaction.

DataObjects.NET fetches objects from database on first attempt to access them. If object is already in cache it is not fetched from database at all. We can define `[LoadOnDemand]` attribute on persistent properties, which says that property should be loaded on first attempt to access it.

Drawback of DataObject.NET framework is that it is suitable just for software project written from scratch, not for software projects that want to integrate with existing (legacy) database system. Reason is that it generates its own relational database schema. This schema can not be maintained like developer will change it and then change mapping, because when mapping is changed, `Domain.Build()` method find this out, generates new database schema and replaces old schema with new one.

7.2 Hibernate

Hibernate is very popular open-source persistent framework. It is very popular in Java community and it expands to .net world as well. .net clone of Hibernate is called NHibernate. In Java, managed as well as not managed environments are supported by Hibernate. By managed enviroment we mean environment where Domain model is hosted by e.g. EJBs. Here we have built in facilities such as transaction management or JNDI for definition of datasource.

Unlike DataObjects.NET, this framework doesn't have requirement that all Domain model classes have to inherit from one abstract class. Persistent classes also don't have to implement any special interface. Testing is less complicated with this approach. Also it is more simple to switch to other mapper when we have no mapper-dependent intrusions in source code.

Unlike other open source tools, this product is well documented. It is widely-spread among the comunity and thus lot of tutorials and articles about various aspects of using of this framework exist. Also lots of trainings are available for developers. Hibernate, as lot of others similar tools, have its own query language. It is called HQL. Hibernate supports variety of database servers: Oracle, DB2, MS SQL, Sybase, MySQL, PostgreSQL, TimesTen, HypersonicSQL, SAP DB.

Architecture

Hibernate uses XML configuration files for storing information about mapping. There is also possibility to configure mapping in standard Java .properties file, but this option is rarely used. Usually, we split configuration to more XML configuration files by rule one class – one configuration file. This is easier to maintain and it is easier to find errors in configuration when using this approach. It is also possible to configure mapping directly in the source code. Before Java annotations exists XDoclet generator was used to generate configuration files from JavaDoc tags.

Let's deal with mapping of inheritance and supported patterns for this mapping. Hibernate supports all three patterns we have presented in part 5.1.1. In order to how we configure mapping, specific pattern is used. If we wanted to use Hierarchy to a single table pattern, our configuration will have these specifics. Base class will be declared with `<class>` element. Within this element we will have to define `<discriminator>` element with `<column>` element with name attribute – name of discriminator. Discriminator is column in database table which tells about which subclass actual row is instance of. Then we have to define `<subclass>` elements within mentioned `<class>` element and configure value of discriminator for instances of these classes. If we want to use Each class to its own table approach we don't have to define discriminator in configuration file. We just have to define subclasses as `<joined-subclass>` elements within `<class>` element. For Concrete class to its own table we just define mapping for concrete classes separately – as they were not related. For more information about configuration of mapping see [HIBERNATE_REF].

Hibernate also supports lots of variants of association mapping. It supports one-to-many, many-to-one, as well as many-to-many (association table) associations. For all of previous it supports uni- and bi-directional relations.

Runtime reflection is used for accessing right fields of persistent objects. In latest version of Hibernate default method to access persistent fields is quite different. Hibernate uses byte code enhancement with use of CGLIB byte code generation library. Distinction to standard code generation frameworks is that Hibernate doesn't generate source code, but Java byte code.

Hibernate uses Transaction type for delaying of updates (Unit of work pattern). Session type is related to Transaction type. Session represents interaction with database. We are asking Session for persistent object as well as we use it for inserting, updating and deleting of objects. If we want to embed some adjustments on one or more objects and delay their update – update them with one database call, we wrap them to hibernate Transaction. At the end of interaction we simply call commit method on Transaction object.

Hibernate has very rich support for lazy processing. It is easy to configure in XML mapping file which attribute or which association has to be processed as lazy. Usually XML element corresponding to class of association has boolean attribute named lazy.

We can use Hibernate as a mapping bridge between existing Object model and relational database. Of course these models have to be similar to make sense to establish a mapping between them. Important fact is that database model is not absolutely dependent on Object model – we can change it easily and then change mapping. This is important when integrating legacy systems, too. Anyway Hibernate has also possibility to generate database schema. This feature is available in Hibernate Tools – member of Hibernate suite. It is not present in Hibernate core. It is also possible to generate (reverse engineer) Object model and mapping from database model. This reverse engineering is out of scope of this work.

7.3 TopLink

TopLink is a product that is on the market for more than ten years and evolved to number one during that time. It has lot of big enterprise clients and after some acquisitions is now part of Oracle Fusion Middleware suite.

TopLink is suitable for Java projects. It persists plain Java objects as well as EJBs. Interesting feature of TopLink is that it is able to transparently persist objects to XML. It is able to work with variety of J2EE application servers and supports every database for that JDBC driver exists. Specific features and extensions of following database servers are also supported: Oracle, DB2, MS SQL, MySQL.

This framework is suitable for large projects, because it is very complicated and thus not suitable for little and agile projects. Thought lot of tools exists for working with TopLink it is still complicated and some time invested to study is necessary. Anyway it has very good performance.

Several ways how to query for objects exists here. We can use QBE-like syntax, java expressions, SQL or stored procedures – predefined named queries which are recommended way to query for data.

TopLink is not open source and not for free. On the other hand support is in price.

Architecture

At Figure 17 we can see one of the scenarios where TopLink can be used. Our Business object is Entity Enterprise Java Bean deployed in Container that manages persistence (we call it Container Managed Persistence – CMP). In this scenario J2EE Container manages mapping, querying and other tasks of TopLink automatically. TopLink is able to provide persistence in many more scenarios. For example it is able to persist POJOs (Plain Old Java Objects), where it can not use services of application server such as transaction services (JTA).

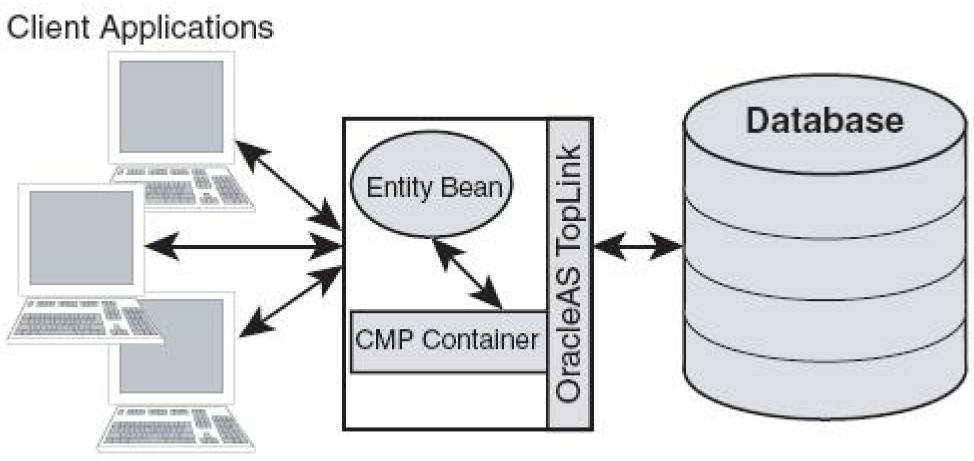

Figure 17 - Use of TopLink with EJB CMP

Although TopLink uses reflection to access fields of persistent objects, its performance is very good. As TopLink has very rich support for lot of mapping scenarios we won't deal with how we can map inheritance or association. TopLink has very rich support for delayed updates and transactions. Lazy loading is called indirection in TopLink. TopLink supports this kinds of indirection: ValueHolder indirection, Proxy indirection and Transparent indirection.

TopLink uses metadata mapping information, of course. Besides standard ways for developers to provide mapping metadata TopLink provides API for working with metadata and changing them. Anyway, this way is not recommended. Recommended way is to let IDE generate metadata information. This IDE is called OracleAS TopLink Mapping Workbench. This IDE supports several approaches. We can

- Import existing classes and database tables for mapping
- Import existing classes and generate database tables and mapping
- Import database tables and generate classes and mapping
- Create classes, database tables and mapping

So we can see that TopLink is flexible enough to support different scenarios. We can for example start new project, define classes and let database be generated or we can let classes be generated when we are integrating legacy system.

7.4 LLBLGen Pro

This persistence framework is representative framework, which uses code generation and is suitable for persisting of objects in .net environment. It is suitable for generating of data-access tier and for creation of business façade – interface for Domain model. LLBLGen Pro contains designer studio and TemplateStudio. In designer studio we can define Entities, EntityLists, Views and stored procedure calls. These are four artifacts LLBLGen Pro works with. Entities are classes from Domain model, EntityLists are typed lists of these classes (in fact they are read only .net typed DataTables), Views are direct mappings of database views and by defining stored procedure you are able to call stored procedure with one line of code. By use of this designer you are able to define mappings, associations and inheritance in very convenient way. After you configure mapping and define all entities you can generate DAL. If you want to adjust generation process, you can do it in template editor. Here you can edit templates used for generation or add custom code parts that are preserved after code re-generation.

LLBLGenPro's entities are custom classes – not tabular .net DataTables or DataSets. Generated code is pattern-based and thus readable for pattern-aware software developer. Used patterns are for example Strategy, Factory, Data Access Object or Data Transfer object.

LLBLGen Pro has also some tracing capabilities. You are able to trace, which sql statements have been sent to database or which methods of DAL have been called. Of course, we have possibility to query for objects. We can also use group by and having statements and aggregate functions also.

This framework is designed to be used in .net environment and generates C# or VB.net code. Works with all currently used versions of Microsoft.net. Supported databases are: MS SQL, Oracle, Firebird, MS Access, IBM DB2, MySql and SQLServerCE.

Interesting feature is validation objects. We can define objects that will validate value of field of an entity or state of whole entity. Very interesting is also fact that we can use more than one database in one project.

This software is not for free and is not open source. If you pay, you will get a license for whole department/division. You will also get access to customer area of llblgen's www page.

Forum, knowledgebase, faq and possibility of asking questions via web interface is available for users of this framework. Interesting is fact, that forum and faq contain some equivalent questions. Also rich documentation in pdf and chm format is available.

Architecture

As we have already said, this framework uses code generation. Code that is accessing fields of business objects is generated from metadata. These metadata are gathered via designer studio we have also spoken about. Developer simply draws mappings without knowing anything about configuration files. From diagrams and configuration data, DAL is generated.

Now we will look closer how we can configure mapping of inheritance in designer studio. LLBLGen Pro supports two of three patterns for mapping of inheritance. It supports Hierarchy to a single table pattern which is called TargetPerEntityHierarchy here and Each class to its own table pattern which is called TargetPerEntity in LLBLGen Pro. If we wanted to configure Hierarchy to a single table, we will select option Create sub type for this entity from the context menu that appears after right click on desired parent entity. After that dialog appears in which we can define discriminator field (which discriminates among entity types) as well as sub type with its discriminator value. We have two possibilities how to create TargetPerEntity mapping. One possibility is to let designer to find all hierarchies in entities in the project and construct them. Designer will create mapping with desired pattern also. Another option is to define hierarchy per entity. By right click on desired sub entity and selecting Make subtipe of option from the context menu we can choose super type of our entity. Note that it is also possible to view and adjust hierarchies in graphical form in designer. Note that it is not possible to mix patterns in one inheritance hierarchy in LLBLGen Pro.

LLBLGen Pro is mapper which uses existing database. When talking about object associations, this frameworks dynamically creates them from underlying database schema. It searches for foreign key based database relations and maps them to Domain model. Developer is then able to alter them with use of designer. He is for example able to set up orientation of association, change name of field that accesses another object via association or hide association following from relational database relation. Besides adjusting of implicit associations developer is able to define his own object associations with desired multiplicity and so on. All of this can be done conveniently with use of designer studio.

Developers can use UnitOfWork class in client code. UnitOfWork holds objects that have to be changed, inserted or deleted. For example at the end of scope of interaction with user we can process all changes on registered objects with one database call. We maintain this container by calling methods like `AddForDelete()`.

As we have outlined, this framework works with existing database. When we are developing project from scratch and want to use this product it is suitable to design database model at first. Then we can adjust implied Domain model by use of designer studio and adjust generation of code by use of TemplateStudio. This step can be treated as configuration of mapping. It should be clear that there should be no problem when integrating legacy relational databases with use of this framework.

7.5 Java Data Objects (JDO)

JDO is a persistence standard defined by Sun. Lot of 3rd party vendors exists that implement this standard and sell their implementations, for example SolarMetric Kodo or TJDO. TJDO is an open source implementation of JDO standard. JDO is also used by vendors of Enterprise Appliocation Systems that want to make their system extendible. They provide APIs for their systems which can be based on JDO. This means they create a JDO implementation and then mapping for their system. SAP is an example of such a system.

JDO is standard that enables objects to persist not just to relational databases, but to XML or file system as well. There also exists another specification similar to Sun JDO and concurrent to Sun JDO which is called Castor JDO. This specification isn't transparent to data source – it supports persistence to relational databases only. It is incompatibile with Sun JDO specification, with poor support and documentation and we will not deal with it any more.

JDO is intended for managing of persistence of Plain Old Java Objects, but it can be used to ensure persistence of Entity Enterprise Java Beans, or can be used as a persistence mechanism in J2EE application servers for EJB CMP.

Architecture

When we want our class to be persistent, it have to implement interface called `PersistenceCapable`. It is interface defined by JDO standard. This interface contains methods which helps JDO implementation to manage fields of persistent object. Methods of this interface can be very complex. In practice it is not required to implement these methods by Domain model developer. Usually, byte code enhancer tool is used to adjust our byte code to implement PersistenceCapable interface as we can see on Figure 18. This byte code enhancer is provided by JDO implementation vendor. During enhancement of byte code it uses some meta data supplied by developer also. Usually line numbers of source code and similar details are preserved, so there is no problem to debug enhanced code. Requirement of JDO specification is that code enhanced by different JDO implementation vendor have to be binary compatibile. Thus persistent class should be portable among distinct JDO implementations and thus this class is independent of one concrete implementation.

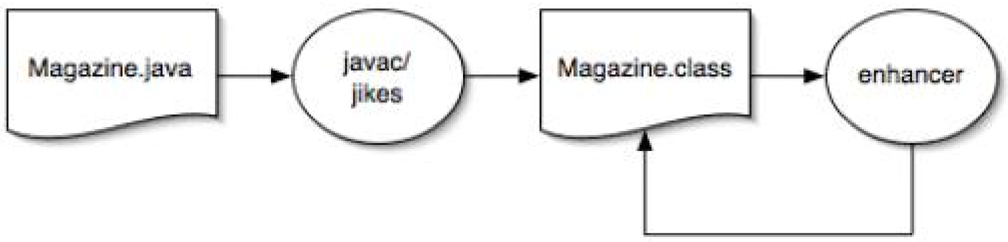

Figure 18 - JDO byte code enhancement process

Specification of configuration file for definition of mapping is defined very briefly in JDO standard and this is the space for JDO implementation vendors. For example Solarmetric Kodo enables all three types of inheritance mapping. This implementation also offers possibility to declare mapping information in source code and generate configuration file by use of XDoclet.

JDO Standard defines set of states that class can be in as well as possible lifecycles of an object. States defined by standard are Transient, Persistent-new, Persistent-dirty, Hollow, Persistent-clean, Persistent-deleted and Persistent-new-deleted. Note that some JDO implementations augument this set of states by new states. For more information about states and possible transitions between them, see [JDO_SPEC].

JDO byte code enhancer approach tells us that JDO approach is closer to (byte) code generation than to reflection. As we have said, input for this generation (adjustment or enhancement) are configuration data, which contain mapping information.

We can see, that any JDO implementation is similar to frameworks like Hibernate when we talk about supported scenarios. We have Domain model, database model and configuration file. It is a choice of concrete vendor and his JDO implementation whether he provides tools for generation of database schema from Domain model, generation of Domain model from database schema and similar.

8 Conclusion

In the first part of this work we have dealt with patterns and categories of ORM frameworks. This part served as a basis for later comparison of these frameworks. We have spoken about frameworks that are suitable for development model where database schema is generated from Domain model (DataObjects.NET). We have also dealt with situation, where Domain model is given by database schema (LLBLGen Pro). More flexible are frameworks that enable both directions (Hibernate, TopLink and some implementations of JDO). We have also spoken about way how framework manipulates Business objects – it can use runtime reflection (DataObjects.NET, Hibernate, TopLink), code generation (LLBLGen Pro) or byte code enhancement (Hibernate, JDO). Most of mentioned frameworks have ability to configure mappings both by attributes in the source code of Domain model and by configuration files. Exceptions are DataObjects.NET where we can configure mapping only by attributes in source code and LLBLGen Pro where we configure mapping with use of specialized IDE.

In Table 1 we can see comparison of frameworks we have dealt with by our benchmarking criteria. This table can serve as a tool when deciding which persistent framework to use in our project.

	DataObjects.NET	Hibernate	TopLink	LLBLGen Pro	Java Data Objects
Transparency	3	3	3	3	3
Documentation & Support	3	3	3	3	3
Dependencies, Supported platforms,	2	3	3	3	3
Maintainability of model	3	3	3	3	3
Usability	3	3	2	3	2
Configuration vs. automation	0	3	3	2	3
Constraints on design of Domain	2	3	3	1	2

Table 1 - Comparison of selected ORM frameworks

Every software project has its specifics and thus there are types of projects that have different requirements on ORM tool from other projects. We have to decide if we want to use robust framework with lots of features, but possibly difficult to use, or small simple framework developers adapt very quickly. Of course, it is probable that we will use simple framework for small, agile project.

It should be clear that in most cases it is less expensive to use framework provided by third party instead of developing new one. In many projects there are just few DALC classes that manage persistence at the beginning. As project grows, number of DALC classes grows and often special library is created for them. Costs which are connected with development, debugging and testing of these classes can grow very high.

9 References

[AMBER]
Amber S. W.: Agile database techniques, *http://www.agiledata.org/essays/mappingObjects.html*, Wiley, 2003

[ARTBASE]
Mrázik A.: ArtBASE, ArtInAppleS, 1994
User guide for Object-Oriented database system ArtBASE

[BALLARD]
Ballard P.: ORM Requirements Recommendations,
http://www.theserverside.net/discussions/thread.tss?thread_id=29953
Discussion about criteria for benchmarking

[BROWN]
Brown K.: Crossing Chasms: A Pattern Language for Object-RDBMS Integration,
http://portal.acm.org/citation.cfm?id=232951&jmp=cit&coll=portal&dl=ACM, Adisson-Wesley, 1996

[DATA_OBJECTS_DOT_NET]
http://www.x-tensive.com/Data/Downloads/DataObjects.NET/DataObjects.NET%20Manual.pdf
DataObjects.NET persistence framework manual

[FOWLER]
Fowler M.: Patterns of Enterprise Application Architecture, Adisson-Wesley, 2002

[FUSSELL]
Fussel M.L.: Foundations of Object-relational Mapping,
http://www.chimu.com/publications/objectRelational/objectRelational.pdf, 1997

[GoF]
Gamma E, Helm R., Johnson R., Vlissides J.: Design Patterns – Elements of Reusable Object-Oriented Software, Adisson-Wesley, 1995

[HIBERNATE_REF]
http://www.hibernate.org/hib_docs/v3/reference/en/pdf/hibernate_reference.pdf
Hibernate framework reference

[JDO_SPEC]
http://jcp.org/aboutJava/communityprocess/final/jsr012/index2.html
Specification of JDO standard

[KELLER]
Keller W.: Mapping objects to tables,
http://www.objectarchitects.de/ObjectArchitects/papers/Published/ZippedPapers/mappings04.pdf, 1997

[LLBL_GEN_PRO]
http://www.llblgen.com/pages/documentation.aspx
Documentation for LLBLGen Pro framework

[MARGUERIE]
Marguerie F.: Choosing an object-relational mapping tool,
http://madgeek.com/Articles/ORMapping/EN/mapping.htm
Information about criteria for benchmarking

[MICHALEC]
Michalec V.: Deklaratívny objektový grafický užívateľský interfejs, 2002

[ŠEŠERA, MIČOVSKÝ]
Šešera Ľ., Mičovský A.: Objektovo - orientovaná tvorba systémov a jazyk C++, Alfa, 1993

[TOP_LINK]
http://www.oracle.com/technology/products/ias/toplink/index.html
Oracle TopLink framework webpage with documentation

[WIKI1]
http://en.wikipedia.org/wiki/Object-relational_database
Information about Object-relational databases

[WIKI2]
http://c2.com/cgi/wiki?ObjectRelationalToolComparisonDotNet
Information about criteria for benchmarking

Abstrakt

Ukladanie objektov do relačných databáz je dnes aktuálna téma. Čo sa týka prístupov k vývoju aplikácií, objektovo-orientované programovanie dnes predstavuje hlavný prúd. Tieto aplikácie si väčšinou svoje dáta potrebujú niekde uložiť. Ak hovoríme o podnikových aplikáciách, ako úložisko dát býva väčšinou použitá relačná databáza. Keďže je medzi objektovou a relačnou paradigmou priepasť - voláme ju sémantická priepasť - splnenie tejto úlohy nebýva priamočiare. O objektovo-relačnom mapovaní má zmysel hovoriť hlavne v súvislosti s veľkými aplikáciami, ktoré by mali byť dostatočne otvorené a udržiavateľné. Jedna z možností ako riešiť tento problém je vytvoriť si v našej aplikácii vlastnú vrstvu, ktorá sa stará o mapovanie medzi objektami a entitami relačných databáz. Ďalšou možnosťou je použitie produktu tretej strany, pomocou ktorého môžeme zautomatizovať väčšiu časť procesu súvisiaceho s mapovaním, ak daný produkt použijeme správne. V prvej časti práce poukazujeme na spôsoby, akými môžu byť takéto produkty navrhnuté a implementované. Tiež prezentujeme vzory pre mapovanie objektových konštrukcií na entity relačných databáz. Preskúmame výhody a nevýhody použitia jednotlivých prístupov a vzorov. V ďaľšej časti práce sa pozrieme na to, ako takéto produkty vyzerajú, kategorizujeme ich podľa rôznych kriterií a porovnáme. Tiež sa zameriame na architektúru jednotlivých riešení a porovnáme ich architektúru z rôznych pohľadov. Pri kategorizovaní a porovnávaní budeme využívať vedomosti z prvej časti práce.

Cieľom diplomovej práce je porovnať frameworky zabezpečujúce perzistenciu a poskytnúť pohľad do vnútra týchto frameworkov. Práca prináša porovnávaciu tabuľku s vybranými frameworkami pre zabezpečenie perzistencie, prináša základnú orientáciu na súčasnom rozsiahlom trhu s ORM nástrojmi a zvýrazňuje niektoré charakteristiky ORM nástrojov, ktoré sú hodné našej pozornosti. Zistíme, že existuje množstvo nástrojov vhodných pre množstvo scenárov.