FoAM – A Framework for Application-Oriented O/R Mapping

Christian Rehn
Institut für Informatik, Technische Universität München, Germany
rehn@in.tum.de

Abstract *This paper enumerates the typical O/R mapping problems, analyses the most critical ones and shows a way to overcome them by using the mapping framework FoAM. FoAM supports aplication oriented mapping of classes to database tables and has succesfully been used in real world Java/J2EE and C# projects together with Oracle and Microsoft SQL-Server databases. In one project it has been used to replace TopLink[1] which increased the performance of the productive system by a factor of 2.5.*

Keywords: object-relational mapping, ORM, object-orientation, database systems

1 Introduction

Today most business application software is developed using object-oriented technologies and relational databases to store the data. All these applications share the same problem: how to map objects and their relationsships to tables and relationships between them in the database. As shown in section 1.1, object and relational technologies are quite different, also known as object-relational impedance mismatch. So conceptual and technical difficulties are often encountered when a relational database management system is being used by a program written in an object-oriented programming language or style.

The paper at hand discusses the differences between relational and object-oriented concepts in section 1.1 and analyses the main reason for O/R mapping difficulties in section 2. Section 2.6 focuses on the basic cause why non-application-oriented mappings will never achive good performance for large business applications. To overcome the mapping difficulties section 3 presents the framework *FoAM* which addresses most of the problems identified in section 2. Finally section 4 summarizes this paper.

1.1 Concept Distinctions

The following two subsections take a short look at object-oriented and relational concepts to lay the foundation for the comparison and analysis in section 2.

1.1.1 Object-Oriented Concepts

Object systems are typically characterized by the following basic concepts:

class: A *class* describes the rules by which objects (instances of a class) behave.

state: The values of the internal *fields* of an object determine its *state*.

identity: Objects have unique *identities* which are distinct from their states. Two objects with the same state are still separate and distinct objects, even if they are bit-for-bit mirrors of each other.

behavior: The operations that clients can invoke to interact with an object define the *behavior* of the object. The behavior distinguishes objects from data structures in a procedural language.

encapsulation: *Encapsulation* hides the internal representation of an object from outside parties. Access to objects is performed via interfaces that together provide the only access to the internals of an object.

inheritance, polymorphism: *Inheritance* is a way to form new classes using classes that have already been defined. The new (derived) classes inherit attributes and behaviour of the pre-existing (base) classes. The ability for a derived object to implement the base functionality of a parent object in a new way is called *polymorphism*.

1.1.2 Relational Concepts

Relational systems allow the storage and retrival of data based on predicate logic and truth statements. The pricipals of the relational model were originally laid down in 1969-1970 by E.F. Codd[2]. The relational model is characterized by:

relation, attribute (column): A *relation* is a truth predicate about the world, a statement

of facts (*attributes*) that provide meaning to the predicate. In a relation, attribute ordering is entirely unspecified.

tuple (row): A *tuple* is a truth statement within the context of a relation, a set of attribute values that match the required set of attributes in the relation. Note that two tuples are considered identical if their relation and attribute values are also identical.

relation value, relation variable (table): A *relation value*, is a combination of a relation and a set of tuples that match that relation, and a *relation variable* is a placeholder for a given relation, but can change value over time.

A collection of relation variables is commonly referred to as a *database*.

These basic element types can be combined using a set of operators (*restrict, project, product, join, divide, union, intersection* and *difference*). These operators form the basis of SQL, the universally-acceptance language for interacting with a relational system.

2 Mapping Challenges

If you compare the concepts described above, it becomes fairly clear that there are distinct differences between a relational and an object-oriented system design. If you try to combine both worlds in modern applications, a number of problems occure (see [3]). These problems are the subject of this section.

2.1 Mapping Classes to Tables

The most obvious challenge wehn using objects as a front-end to a relational data store is that of how to map classes to tables. A first and simple approach would be to map tables to classes and columns to fields. The field types line up almost directly against the relational column types (Strings to VARCHARs, ints to INTEGERs, ...). In a first approach for any given class defined in a system, a corresponding table can be defined to go with it.

2.1.1 Inheritance

Unfortunately, the relational model does not support any sort of inheritance or IS-A kind of relation. Therefore we have three possible options to map inheritance into the relational world:

one-table-per-class:
Each class in the inheritance hierarchy gets its own relational table, and objects of derived types are stitched together from relational JOIN operations across the various inheritance-based tables. Considering that JOINs are among the most expensive expressions in RDBMS queries, usually one of the following approaches is used to map inheritance.

one-table-per-concrete-class or one-table-per-class-family:
In the first case one table for each concrete (most-derived) class in the second case for the entire class hierarchy is created. The drawback of the one-table-per-concrete-class solution is denormalization. Unfortunately, the denormalization costs are often significant for a large volume of data, and/or the table(s) will contain significant amounts of empty columns, which will need NULLability constraints on all columns, eliminating the powerful integrity constraints offered by an RDBMS.

None of these mapping strategies are ideal for all situations. Therefore the best sollution is to combine the three approaches in a given application. Even mapping multiple inheritance is simple by applying the three inheritance mapping approaches above. As you can see, the mapping of classes to tables is simple and staight forword.

2.2 Mapping Relationships

There are three types of object relationships that have to be mapped: *association*, *aggregation* and *composition*. For the O/R mapping the three types of relationships can be treated the same way although we can find slight nuances when it comes to referential integrity.

There are two characteristics of object relationships that have to be considered for mapping: multiplicity and directionality. For multiplicity we can distinguish three types:

- One-to-one relationships
- One-to-many relationships
- Many-to-many relationships

In relational systems an association is actually reversed, from the associatee to the associator via foreign key columns. This means that for m:n associations, a third table must be used to store the actual relationship between associator and associatee, and even for the simpler 1:n relationships the associator has no inherent knowledge of the relations to which it associates–discovering that data requires a JOIN against any or all associated tables at some point.

For directionality there exist two types:

- Uni-directional relationships
- Bi-directional relationships

Relational technology does not support the concept of uni-directional relationships, all associations are bi-directional. Relationships are implemented by foreign keys which can be traversed (JOINed) in either direction. This is completely different in the object-oriented world. Here all associations are unidirectional, i.e. associated objects have no idea they are in fact associated unless an explicit bidirectional association is established.

Despite the differences, mapping relationsships is quite simple and can be done straight forword using foreign keys and additional tablels for n:m relationships.

2.3 Identity Issues

Object systems use an implicit sense of identity based on the object's location in memory. By contrast in a relational model identity is implicit in the state itself. Two rows with the exact same state are identical. Usually this is considered as a relational data corruption because of redundancy and often explicitly disallowed by explicit relational constraints, such as primary key constraints. There is no implicit sense of identity to a relation except that offered by its attributes. The same is not true for object systems, where two objects that contain precisely identical bit patterns in two different locations of memory are in fact separate objects. Therefore the relational system must offer some kind of unique identity concept to match that of the notion of object identity.

2.4 Data Retrieval

One of the most important questions for O/R mappings is: How can an entity be retrieved once it has been stored within a database? An object-oriented approach would make use of constructor-style syntax identifying the object(s) desired, but unfortunately constructor syntax isn't generic enough to allow for something that flexible; in particular, it lacks the ability to initialize a collection of objects, and queries frequently need to return a collection, rather than just a single entity as multiple trips to the database to fetch entities individually is generally considered too wasteful (see section 2.6). As a result O/R mapping tools usually end up with one of Query-By-Example (QBE), Query-By-API (QBA), or Query-By-Language (QBL) approaches.

The problem with the QBE approach is that while it is perfectly sufficient for simple queries, it is not expressive enough to support more complex queries that are frequently needed. Therefore many O/R mapping tools support the QBA approach, in which queries are constructed by query objects. Unfortunately QBA is usually much more verbose than the traditional SQL approach, and certain styles of queries (i.e. outer joins) are more difficult to represent. In addition developers have the problem that table and column names are strings and there is no validity-checking at compile time. So misspelling can only be detected at runtime which complicates software development.

The QBL approach includes a new SQL-like language (i.e. OQL and HQL) which supports the complex and powerful queries normally supported by SQL. Very often those languages are only a subset of SQL and thus do not offer the full power of SQL. In addition QBL leads to a loss in transparency. Developers need not to be aware of the physical schema of the data model but of how object associations and properties are represented within the language and the subset of the object's capabilities within the query language.

2.5 Shadow Information

Shadow information is any data that objects need to maintain, above and beyond their normal domain data, to persist themselves. Examples are primary key information (particularly when the primary key is a surrogate key that has no business meaning), concurrency control markings such as timestamps or incremental counters, locking information and versioning numbers. To persist an object properly the corresponding class has to implement shadow attributes that maintain these values. Shadow information disrupts transparency a little bit, which is an important point for the application of O/R mapping tools, but can be handled by mapping tools and developers very easily.

2.6 Partial Objects

The problem with partial objects is the major problem for O/R mapping tools that do not support application-orientation. It is the main mapping challenge and therefore discussed at the end of this section.

It is generally known, that the number and complexity of queries and the size of the result sets have a major impact on performance of a software system. Especially in a client server environment where data/objects have to be sent over networks the size and number of messages is critical. Therefore developers look for ways to minimize this cost by optimizing the number of round trips and data retrieved. In SQL, this optimization is achieved by carefully structuring the request, making sure to retrieve only the columns and/or tables desired, rather than entire tables or sets of tables. The notion of being able to return a part of a table is fundamental to the ability to optimize queries. Usually application only require a portion of the complete relation in some situation while they need different

portions or the whole table in other situations.

This presents the central issue for most object/relational mapping layers as their goal is to enable the developer to see nothing but objects. Developers cannot tell the O/R layer how the objects returned by the query will be used. To relieve the performance problem most O/R layers support lazy-loading of objects and/or fields. This is especially helpful if you have a network of related objects. Without lazy-loading you would always get all those related objects even if your are only interested in a single one. Referenced objects are loaded on demand when they are accessed for the first time. This reduces the amount of data but also increases the number of queries an/or network messages when referenced objects have to be loaded in an additional step. It might be helpful in some situations whereas it can be counterproductive in others. Without application-orientation some compile-time per class decisions have to be made about when to retrieve associated objects and there will always be common use-cases where the decision made will be exactly the wrong thing to do. These decisions cannot be changed on a situational basis.

3 The Solution: FoAM

In order to solve many of the O/R mapping problems described above we have developed the *FOAM*-framework to support application oriented mapping of objects and their relationships. Up to now *FoAM* has been used in several real world C# and Java/J2EE projects together with Microsoft SQL-Server and Oracle databases. With little effort it could also be used together with other OO languages or relational databases.

As you have seen in section 2.6, application-orientation is indispensable if you want to achieve good performance. Most available mapping tools and libraries provide universal solutions to be flexible enough to be applied to all kinds of OO-applications. This generalization normally leads to unadjusted mapping decisions, a high number of queries (i.e. multiple SELECTs instead of one JOIN) and to moderate performance. Because database access is the most critical part in modern applications, those mapping tools and libraries often try to conceal these performance problems by caches or tuning parameters that can be manipulated by the application developer. But caches can only solve one part of the partial object problem, the can reduce the number of queries executed on the database. What still remains is the inability to provide situation dependent filling levels of objects. Apart from lazy-loading which is not situation dependent but has to be used for all instances of a class, no finer grained adaptions to application needs can be made.

When using *FoAM* the developer can make class and object specific and even situation dependent decisions. An example for a class specific decision is: all instances of a class can be optimistically locked (see section 3.6). In addition to class specific decision developers have the ability to make situation specific decisions, i.e. in this part of the code an object is needed without its related objects and elsewhere in the code only the attributes *id* and *name* have to be filled (see section 3.5). Anytime a partially filled object can be completed or even reduced. The reduction might for example be used to remove read-only attributes/related objects before sending an object back from a client to the server in order to reduce network traffic.

3.1 Technical Aspects

FoAM consists of a number of abstract base classes, interfaces and abstract controllers and makes intensive use of inheritance. A class X of persistable objects $\{x_1, x_2, \ldots\}$ simply has to be derived from one of the *FoAM*-provided base classes and a controller C_X has to be implemented. To make the controller implementation very easy it can be derived from one of the abstract controllers provided by *FoAM*, so C_X only has to add class-X-specific knowledge. *FoAM* defines standard mappings and in addition allows developer to adopt or overwrite the default behavior where needed. For the interaction between an OO-application and a relational database *FoAM* uses SQL and any API that supports the execution of SQL statements as for example JDBC or ODBC. To use *FoAM* the application developer has to have knowledge about both, the application and the database schema which, in our opinion, is indispensable for writing good and performant applications. If the database schema is completely hidden from the programmer by a mapping tool or library only universal solutions can be applied an no application or schema specific knowledge can be utilized.

3.2 Mapping Classes and Relations

The default mappings of classes to tables with *FoAM* is straight forward as described in section 2.1: each class, concrete class or class family (depending on application and performance needs) has a corresponding table. Attributes map to columns of the adequate type. If needed, attributes can be added that are not persisted (do not have corresponding columns), for example to cache some derived values. By default all persistable objects have some attributes (*id, name, changedBy, changedOn, ...*) and a basic behavior in common. The classes of these objects have to be derived from

the abstract class *PersistentObject (PO)*. The default attributes can be modified if needed by simply adjusting the PO class. The *id* (primary key) attribute of class *PO* is also used to indicate whether an object currently exists in the database or not (in this case it has a null value).

The lifecycle of persistable objects is controlled by class-specific object controllers. It forms the bridge between the object oriented and the database world and is responsible to create, store, retrieve and delete objects. All controllers should be derived from the abstract class *PersistentObject-Controller (C_{PO})* which defines a default behavior (see figure 1). Derived controllers need not over-

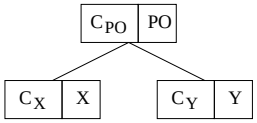

Figure 1: Classes and controllers

ride this behavior. Only in special cases where class specific optimizations are needed (mostly for performance reasons) application-specific implementations might override the default implementations. Everything that is needed to control the lifecycle of objects and map classes/objects to tables/rows is encapsulated in the controllers. For example, a controller knows the name of the corresponding database table and the SQL statements to insert, update or select a tuple to store, modify or retrieve an object. Part of this knowledge can be found in the methods *sqlSelect*(), *sqlInto*(), *sqlValues*() and *sqlSET*(). Each controller C_X uses the method of its base class and adds knowledge about the class X it is responsible for as you can see in figure 2. Each controller in the inheritance hierarchy is only responsible for the attributes of the corresponding class in the inheritance hierarchy.

```
public String sqlInto() {
  return super.sqlInto()
    + ", attribute1ofX " + ", attribute2ofX ";
}
```

Figure 2: Adding class specific knowledge

Controller C_{PO} is responsible for the default attributes of class *PO*, controller C_X only for additional attributes of class X. For example to store an instance x of class X in table T_X the *save* method implemented by C_{PO} is called. If x has not been stored in the database before, the *save* method uses the methods *sqlInto*() and *sqlValues*() to execute an INSERT statement. If there is a tuple for x in table T_X the *save*() method uses *sqlSet*() to execute an UPDATE statement. To retrieve objects C_{PO} implements some operations as for example *findAll*(), *findFirst*(), *findByPrimaryKey*(), *findByWhere*(), ...which are all based on the *sqlSelect*() method. In order to make objects from query results (resultsets) the controller C_{PO} calls the constructor of the according class using reflection. Therefore each persistable class must implement a constructor that has a resultset parameter, calls super(resultset) and finally fills its own attributes from it (see figure 3).

All types of relationships discussed in section 2.2 are handled by *FoAM* in the same way. Here we explain the mapping of a one-to-many relationship which can be adapted to all other relationship types. In order to store a one-to-many relationship a foreign key is used in the database table that represents the many-side. To save an object x that contains a collection object $V_{x-y} = y_1, y_2, \ldots, y_n$ and is an instance of class X, the controller C_X of X first writes a tuple $t_x = (a_1, a_2, \ldots)$ to table T_X executing an INSERT or UPDATE statement ($a_1, a_2, \ldots$ are attribute values of x). After that the primary key pk_x of t_x must have a value. This primary key and the collection V_{x-y} is used by C_X to call method $fillDBFromVector()$ of controller C_Y. This method is inherited from and implemented in C_{PO}. In a loop over the elements of V_{x-y} all objects $y_i, i \in \{1, \ldots, n\}$ are now saved in table T_Y executing INSERT or UPDATE statements. All the tuples t_y get a foreign key $fk_y := pk_x$.

Retrieving objects with one-to-many relationships is described below in section 3.4.

3.3 Entity Identity

If we take a look at a distributed system we find the same identity problems that we have for O/R mappings. Identical copies of the same object can reside in different address spaces on different nodes of the system. The objects are identical but the identity cannot be based on the location in memory. In this case we can no longer check the identity of objects by checking reference equivalence. Instead only object equivalence can be used. As all persistable *FoAM* objects get an id value based on the primary key the application and the database agree on the 'identity' of stored persistable objects. Objects with different ids are not equivalent whereas objects with identical ids have been filled from the same tuple.

To check object identity the abstract *FoAM* class *PO* has two implementations with different semantics. The first implementation only compares the id-value of two objects and should be used for read-only objects. The second implementation uses reflection to compare all non-immutable attributes and can be used by all derived classes. If necessary (i.e. for better performance by not using reflection) it can of course be overridden.

3.4 Object Retrieval

Controllers are responsible for data retrieval and object initialization. They provide a number of methods to get an object by name or id, a vector of all objects for a certain class and so on. To do this, controllers and classes of persistable objects share the task of object retrieval. Controllers are responsible to get the required data from database tables. Exploiting inheritance controllers create appropriate select statements. The execution of a query returns a resultset that represent all matching tuples. Class *PO* and all derived classes implement a Constructor to initialize an object from an element of a resultset (see figure 3).

```
public X(ResultSet rs) throws SQLException {
  super(getController().getTableName(), rs);
  attribute1ofX = rs.getLong("column4attribute1ofX");
  attribute2ofX = rs.getString("column4attribute2ofX");
}
```

Figure 3: Constructor of class X

To retrieve objects with relationships *FoAM* provides two options:

- using a single query with JOIN statements or
- using one SELECT statement for each object

In the first case each constructor uses the parts it needs to initialize its own attributes and forwards the resultset to associated objects by calling their constructors. An additional postfix is used in the query to distinguish attributes with identical names. The JOIN statement best suits to one-to-one relationships (see figure 4).

```
public Y(ResultSet rs) throws SQLException {
  super(getController().getTableName(), rs);
  attribute1ofY = rs.getLong("column4attribute1ofY"
                             + POSTFIX_Y);
  attribute2ofY = new Z(rs);
}
```

Figure 4: Constructor of class Y

In the second case where multiple selects are executed the constructor of Y has to call the *findByPrimaryKey()* method of the Controller C_Z of class Z for one-to-one or the *findByForeignKey()* method for one-to-many relationships instead of the constructor as shown in figure 5.

```
public Y(ResultSet rs) throws SQLException {
  super(getController().getTableName(), rs);
  attribute1ofY = rs.getLong("column4attribute1ofY");
  attribute2ofY = ZController.findByPrimaryKey(
    rs.getLong("foreignKeyForZ"));
  attribute3ofY = ZController.findByForeignKey(
    rs.getLong("primaryKeyOfY"));
}
```

Figure 5: Alternative constructor of class Y

3.5 Lightweight and Partial Objects

In section 2.6 the partial-object problem has been identified the be the main source of O/R mapping performance problems. To increase performance and reduce network traffic *FoAM* has the ability to support different *filling levels* for objects. With *FoAM* an application developer has all the benefits he is familiar with when using SQL by retrieving situation specific data instead of whole objects.

For example as already stated in the abstract we increase the performance in a project by replacing TopLink[1] with *FoAM* by a factor of 2.5. The application of lightweight and partial objects added an additional factor of 10 in situations, where large objects had been exchanged between client and server. Without partial objects sometimes 40 MB had to be sent over the network. Compressing these objects did not improve the situation, as the time for compressing and decompressing outweighed the savings of reduced network traffic. Some parts of the objects within those 40MB were even not visible on most client forms and mainly accessed on the server. Other parts could be lazily loaded later when a user really wanted to access the data. Parts of the lazy-loaded data were read-only and could be removed before sending the objects back to the server for computations. After introducing *partial objects* initially only 500 kB of data had to be retrieved from the database and exchanged between client and server.

In order to support different filling levels a class has to implements the interface *IDifferentFillingLevels*. All instances of this class can be in one of three filling states[1]:

- *incomplete, id and name only*: only the two attributes *id* and *name* that are defined in *PO* are retrieved by the controller and filled in the constructor (see also section 3.4).
- *incomplete, all attributes except relationships*: all attributes that can be filled by a single SELECT query without any JOINs get values.
- *completely filled*: the level that all persistable objects support innately.

Partial classes and their controllers support an additional parameter for their constructor and some of the lifecycle methods to indicate the required filling level (see figure 6).

To change or assert filling levels *FoAM* provides the method *changeFillingLevel()*. The method has three parameters: the object, the filling level and an additional value to control if the given level is the minimum (at least) or maximum (at most) level or if it is exactly the level the object has to adopt.

[1]For our real-world applications three filling levels had been enough up to now. If additional levels are needed, *FoAM* can be extended accordingly.

```
public ZP(ResultSet rs, FillingLevel fl)
throws SQLException {
  super(getController().getTableName(), rs, fl);
  if (fl.isPartial() || fl.isComplete())
    attribute1ofZP = rs.getLong("column4a1ofZP");
}
```

Figure 6: Constructor of partial class ZP

Without modification the framework supports completion only on the server (where the database directly is accessible). Reduction can also be used on the client computer (i.e. to reduce network traffic). In one of our projects we extended the behavior of method *changeFillingLevel()* so it could even be executed on client hosts. The implementation of completion of the client strongly depends on the application-specific environment (sockets, EJBs, RMI, .NET, ...) and is not part of the *FoAM* framework.

Different filling levels are only one step towards application orientation. If we have a network of related objects we need the ability to specify the filling level of all those partial objects within the network. Therefore *FoAM* provides the concept of *lightweight* classes. Lightweight classes implement the interface *ILightweight* and provide a constructor that takes an additional argument, an Array that describes the filling levels of partial objects within the network (see figure 7). Lightweight

```
public ZL(ResultSet rs, Lightweight[] lw)
throws SQLException {
  super(getController().getTableName(), rs, lw);
  attribute1ofZL = ZPController.findByPrimaryKey(
    rs.getLong("foreignKeyForZP"),
    Lightweight.RESULTS_INCOMPLETE.isElementOf(lw)
      ? FillingLevel.ID_NAME_ONLY
      : FillingLevel.COMPLETE);
}
```

Figure 7: Constructor of lightweight class ZL

classes do either forward the lightweight status to lightweight classes of extract a filling level and forward it to partial classes. This situation is visualized in figure 8 where AL is the root of a tree of related classes. When a new instance *al* of the lightweight class AL is created the additional parameter to describe the lightweight status is forwarded to all directly related lightweight classes (BL and CL). Those in turn forward the lightweight status until a partial class is reached. Next the filling level for this partial class is extracted (in AL, FL and CL) and forwarded to the incomplete object (DP, GP and IP).

3.6 Locking Support

In order to support optimistic and pessimistic locking, the additional classes *OptimisticLockablePersistentObject (OLPO)* and *PessimisticLockablePersistentObject (PLPO)* together with their con-

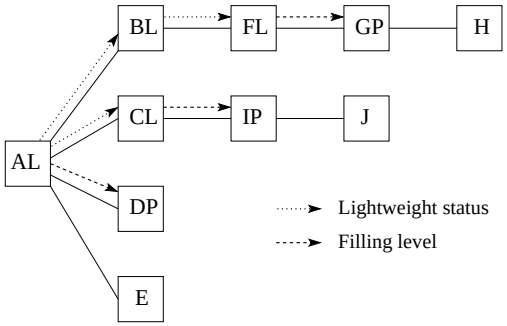

Figure 8: Lightweight status and filling level forwarding.

trollers C_{OLPO} and C_{PLPO} are available in the *FoAM* framework. A class Z and its controller C_Z just have to inherit from one of those classes and the corresponding table T_Z gets a new column *version* for optimistic locking or *lockedBy* for pessimistic locking.

4 Conclusion

In the paper at hand the simple and lightweight O/R mapping framework *FoAM* has been presented. The framework overcomes the difficulties of universal O/R mapping tools and libraries that try to be the transparent and simple-to-use swiss-mapping-knives for all application domains. *FoAM* implements default mappings that can be used without modification or extended for application-specific needs. Unlike most mapping frameworks mapping exceptions are possible, that means you can even ignore the framework completely without any conflicts to gain maximum performance for special classes and situations. The introduction of classes that support different filling levels for their instances and lightweight classes that allow the control of filling levels of related objects enable an application to get objects according to the situation specific needs. *FoAM* also provides a way to adapt the filling levels of objects to changing needs over the time.

The next step to further improve the framework will be the implementation of class-specific caching to reduce the number of database queries for infrequently changing objects.

References

[1] Oracle Toplink. www.oracle.com/technology/products/ias/toplink.

[2] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[3] The Vietnam of Computer Science. blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx.