Mapping Associations from OODBs to RDBMs

Prepared for discussion on the Internet

This paper represents personal opinion and is not a statement by sd&m.

Wolfgang Keller

sd&m München March 10, 1995

sd&m GmbH & Co. KG
Finanz-Informationssysteme
Thomas-Dehler-Straße 18
81737 München
Telefon (089) 6 27 02 - 0
Telefax (089) 6 27 02 50

Table of Contents

1 2

1 Abstract **1**

2 The Problem **1**

3 Architecture **2**

4 Mapping Associations **3**

4.1 First Example - 1:n Association 3

4.2 Various Solutions 5

4.3 Second Example - n:m Association 6

4.4 Our Proposed Solution 7

4.5 The Design of our Solution 9

4.5.1 Lazy Notification 12

4.5.2 Implementing Event Notification 13

4.5.3 Storing Associations 14

4.6 Discussion of our Solution 14

5 References **14**

1 Abstract

This paper describes problems that you will be confronted with, when trying to build an Object Oriented Data Access Layer for relational legacy data without any alterations of old tables - the relational data reengineering problem[1]. Mapping associations is one central topic that has to be handled for such an access layer.

We present our current solution, the solutions we have abandoned in our discussion and we invite others to join our discussion with the purpose to find an elegant solution for the handling of relational legacy data in OODBMS without breaking ODMG'93 user code patterns.

This paper does not discuss the overal architecture of an access layer - it only discusses the special problems around relations.

If you have comments or ideas, please send email to `wk@sdm.de` or `100655.566@compuserve.com`

We have posted other discussion papers on other partial aspects of our problem - so parts of this paper might be redundant.

2 The Problem

Our goal is to offer an Object Oriented Access Layer for relational data. The environment we develop our solution for is a large bank with thousands of legacy applications and thousands of relational legacy tables. We have decided to use an access layer instead of an OODBMS because we have to support about 80% old relational legacy data and only about 20% new data in the new applications, we are going to develop. We also have to support access to data in systems like IMS and TABEX.

Our special situation (DB2, IMS, OS/2, MVS, 3 Layer C/S) does not allow us to use Persistence, Objectivity or ONTOS as frontends for various reasons:

- Persistence does not yet support DB2.

[1]see also [Pre 94]

- ONTOS and Objectivity have hooks for replacing their storage management by custom routines but no way of dealing with transaction monitors like IMS/DC - we have to access our data through IMS/DC.

- We don’t know if anyone of them has a reengineering concept that works without changing tables and allows coexistence of old and new applications.

3 Architecture

To facilitate the following discussion, we will briefly describe our overall architecture for an OO Data Access layer (see also Picture 1):

- The Object Layer contains persistent objects from a common protocol class PObject. It also contains an object manager, transaction objects, the database object and some additional classes like filters and catalogs.

- The Entity Object Layer contains Objects that reflect the physical relations but use intelligent data type classes for base types and structures.

- The Distribution Layer is a set of abstract queries that are used to hide the physical location of data an the database system used (IMS, DB2, TABEX). Queries are static (with respect to DB2 and others) and are selected by a query manager at runtime.

Object Layer
Persistent Objects, Filters, Catalogs, Object Management, Downcasting, ...

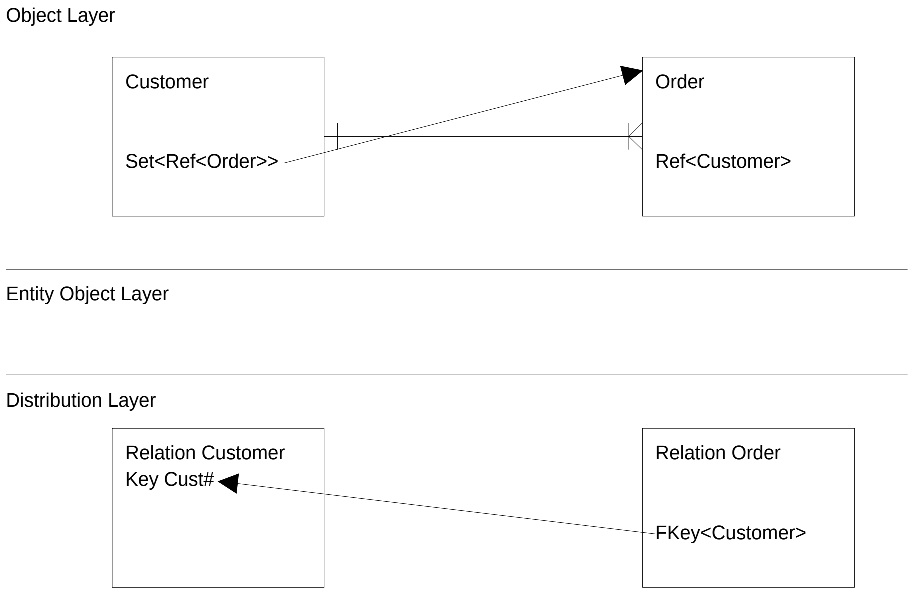
Entity Object Layer
Casting, Intelligent Data Types

Distribution Layer
Queries, Hides Data Location, Hides Physical Storage System

Picture 1: Architecture

4 Mapping Associations

Mapping associations between objects to existing relational databases is one of the harder tasks in our overall task list.



Picture 2: The object oriented way of thinking will invert the way associations are represented - it is a complex task to hide this completely from application level programmers

The problem stems from the fact that OODBMS will invert the associations as represented by foreign key relations in relational database systems (see *Picture 2*). Our goal is to provide users (= application level programmers) with a transparent solution for this problem. Reelational constructs should be totally hidden and encapsulated by classes.

4.1 First Example - 1:n Association

In **Picture 2** we have a typical 1:n association between two objects, customer and order. In an OODBMS this association will be represented by one of the following class definitions:

```
class Customer {
....
   set<Ref<Order>>Orders;
}
```

or

```
class Customer {
....
   set<Ref<Order>> Orders inverse Customer;
}
```

if the OODBMS is supposed to store a bidirectional association. The relational schema will be implemented as shown in **Picture 2**: The Relation Order will include the customer relations primary key as a foreign key.

The user code we want to write when we want to insert a new customer and further want to define an order for this customer will be something like:

```
Ref<Customer> MyCustomer;
Ref<Order> MyOrder;
MyCustomer = new Customer(some arguments ....);
MyOrder = new Order(some arguments);
MyCustomer.insert(MyOrder)
```

The call to MyCustomer.insert should be implemented something like:

```
VOID Customer :: insert (const Ref<Order> & ord) {
  ....
    Orders.insert(ord) // inserts referenced object into container
    markModified() // requests storage of object from object manager
}
```

For the application programmer this looks easy and is the way things should be implemented. For the application layer design problems begin exactly here. Note that we have manipulated only the customer object - but in the relational schema we will have to store a new image of the order relation.

The problem is, how do we notify the order object that it must change, e.g. register the customer objects OID (object identifier)?

4.2 Various Solutions

Simple OODBMS

Simple OODBMS that are just object oriented data storage systems do not have these problems. Their relations are implemented exactly the forward way the association is declared in the class definition.

An examples for such a system is POET (by Poet Inc.). POET will not even allow to specify inverse associations that can be used both ways round.

Full featured OODBMS

Other OODBMS like e.g. Objectivity will allow the programmer to specify inverse associations plus relational integrity constraints (like delete dependencies).

They will treat the programmers declarations with a preprocessor and will automatically generate instance variables for containers and methods for handling insertions, deletions and such.

Programming Conventions

The cheapest way to implement the above behavior is to have it implemented by the application programmer instead of a preprocessor. The method will then look like:

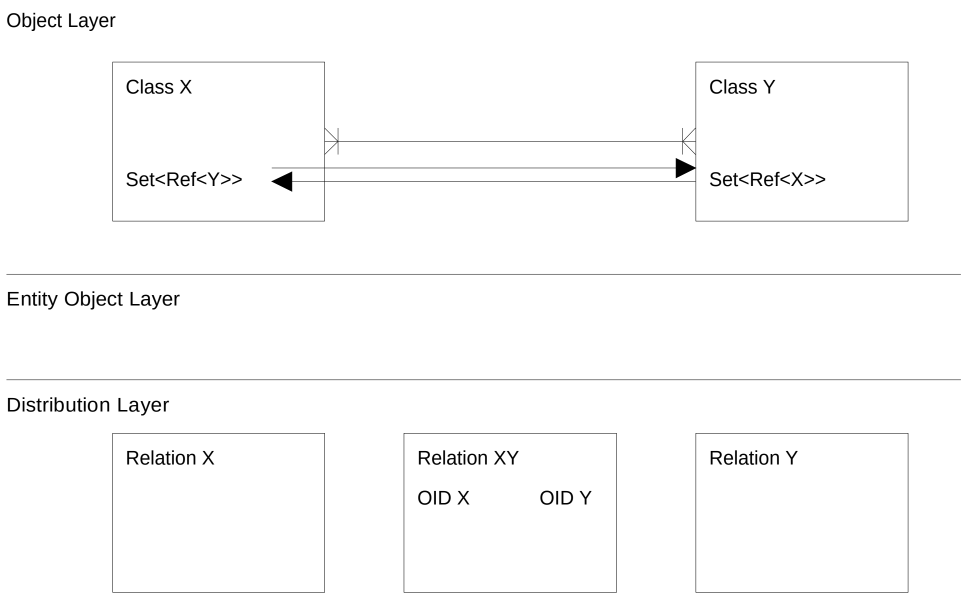
```
VOID Customer :: insert (Ref<Order> & ord) {
  ....
    Orders.insert(ord) // inserts referenced object into container
    ord->SetCustomer(This->getSmart-Pointer()); // stores inverse
                                               // association
    ord->markModified(); // requests Order to be stored
}
```

This solution can live without any precautions by the object oriented access layer. We have a normal standard container class (set) which can be used in any context, not only the access layer.

Before we come to our solution, we have to discuss the problem of many to many (= n:m) associations.

4.3 Second Example - n:m Association

The solution from the section above will no longer work for the more challenging example of many to many associations. *Picture 3* shows an example.



Picture 3: Many to many association - the object layer does not contain a relation object

As we can see in *Picture 3* there is no equivalent for the relation table on the level of the object layer. We thus have no possibility to escape from the problem by storing the inverse association and setting the associated object modified, like in the case of 1:n associations.

The problem occurs for inserts into the set of associations and also for deletes from the set of associations.

We therefore need another mechanism to deal with this kind of problem

4.4 Our Proposed Solution

For our solution we will use a derived container class that will register all manipulations concerning the associations. So the class declaration will be

```
class X {
....
   Association<X,Y> sXY;
}
```

and the code for inserting an X to Y association should be

```
VOID X :: insert (Ref<Y> & y) {
  ....
    sXY.insert(y) // inserts referenced object into container
    ord->markModified(); // requests Order to be stored
}
```

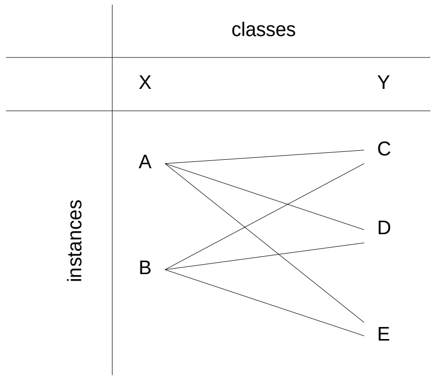
So far nothing has happened on the database level. When a write2Database is invoked by the object manager on commit, all association containers are checked for registered changes. The changes will then be executed on the database level. The code to do this is handwritten. It could alse be generated by a preprocessor, but we have stepped back from a preprocessor solution for reasons of cost.

Implementation

To provide correct semantics the Association implementation has to guarantee, that

- deleted associations are invisible for calls after deletion
- inserted relationships are made visible before a transaction commits

n:m Relationships must be treated as **symetric twosided relationships** for reasons of the underlying database design which uses one table for both directions and will therefore have duplicate key problems when an object layer should try to insert two rows, one for each direction. Consider the following situation with object instances A, B, C, D and E of types X and Y.



Picture 4: twosided symetric n:m associations

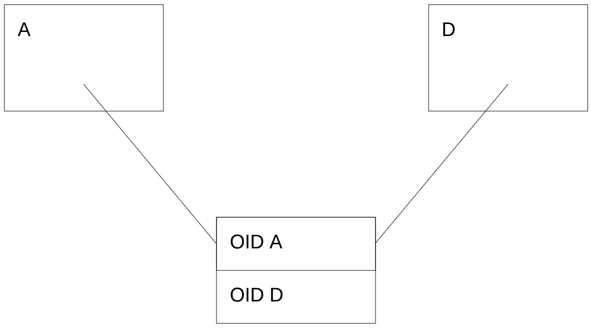
If you delete association A <---> D the association has to be deleted by both sides.

There’s two possibilities to do this:

(1) Instance A of class X will notify instance D of class Y that it should remove the relation from ist association set.

(2) The association container will consist of association objects. Those association record objects. These consist of two OIDs (both derived from a common class OIDBase. These association objects can themselves be treated as persistent objects that have their own generic write2DB methods.

The following picture will illustrate the idea for pattern 2:



The association objects will make deletion of associations an easy job, because both A or D cann see if the association object is invalid for some reasons -

inserting is still hard job because the association object has to be inserted in both sets.

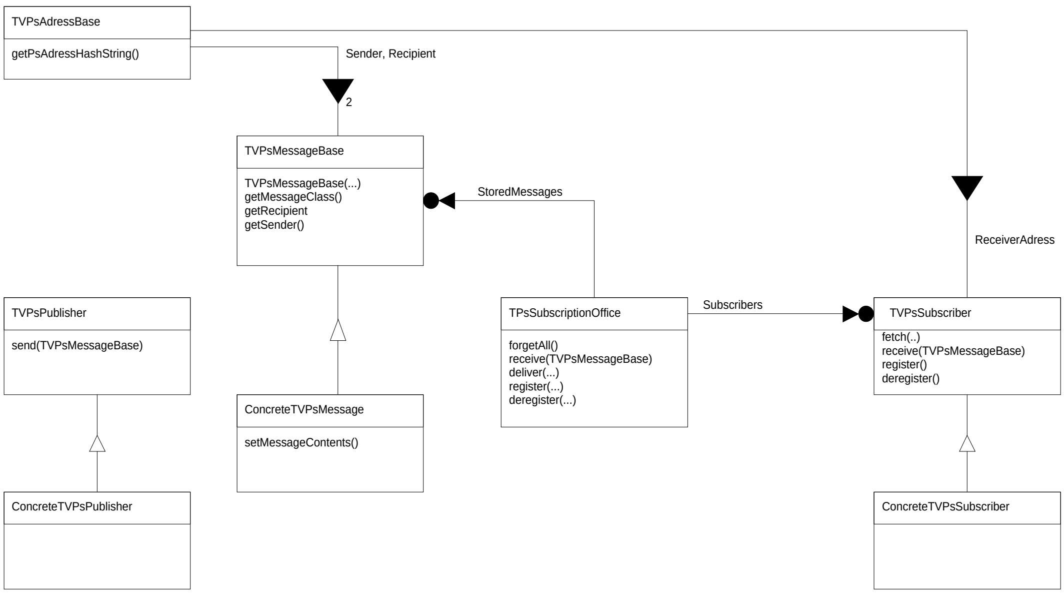
So if we want to simplify the task, pattern (1) looks smarter.

But in pattern (1) D needs code to process the messages from object A. This code can be part of a clever template. Otherwise we would have to generate code with a preprocessor - which is very expensive in terms of implementation cost.

If we use one of the two above patterns, we can treat 1:n associations as a special case of n:m associations.

4.5 The Design of our Solution

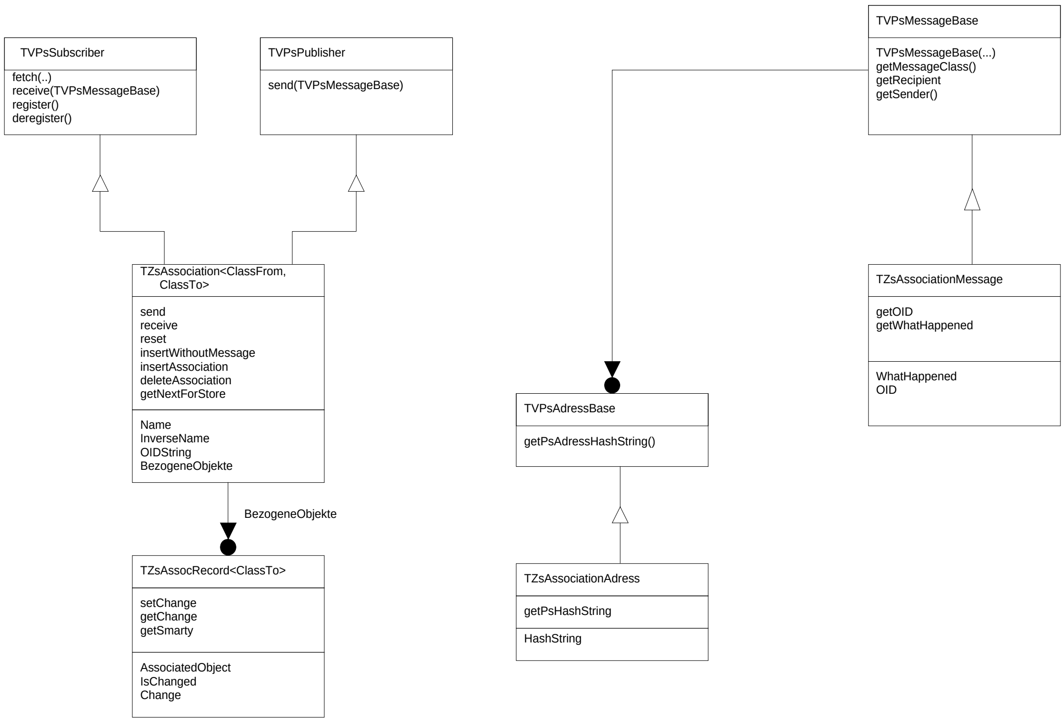
For our solution for n:m associations between arbitrary classes we will use a derivate of the Observer Pattern [Gam 94]. The observer pattern is related to the model-view-controller pattern and also known as publish-subscribe mechanism. This pattern is used to propagate changes via a unified protocol. The observer has to react when notified of a change of the observed subject.



Picture 5: Publish/Subscribe Mechanism

For our case associations can occur in both roles as observers and subjects. An association has to notify the other side of the association of any inserted, deleted or stored associations. On the other hand it must be notified of identical events on the other side of the association.

Associations are encapsulated in a template class TZsAssociation<ClassFrom, ClassTo>. This template class replaces code generation. Each symetric association is represented as two member variables of this template class on each side of the association.



Picture 6: Solution Sketch

In our example we have two classes (X and Y) that have at least one n:m association.

For the side of class X the template has to behave like

- a set<Ref<Y>> to insert, delete and access associated objects of class Y,
- a subject that notifies the observing association in class Y of any insert, delete or store events,
- an observer that follows any insert, delete or store events from the other side of the association,

  an iterator that allows browsing through the actual set of related objects.

Some facts make life easier, some make life more complicated than the standard pattern.

- There can be several associations between two classes X and Y. These associations must be named with a unique-per-class identifier and must also have a reverse name for reasons of symetry. This implies that it is not sufficient to dispatch a notify-message from an instance of class X to an instance of class Y as there is more than one possible association member that may be the recipient of this message. As an example see the following declaration:

  ```
  class Professor {
        Ref<Student> Students inverse Professors;
        Ref<Student> Tutors inverse Bosses;
  }
  ```

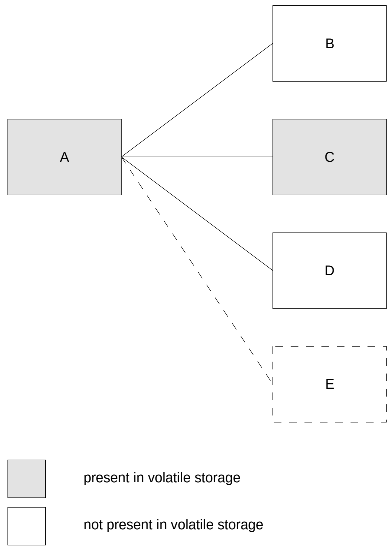
  A single association object like *Students* will only know the OID and inverse name of the object it has to notify - therefore we need a dispatcher at the level of the enclosing object (*Student*) that will forward the events to the correct instance of an association.

- One subject association (e.g. Tutors) has several subscribing object instances of class *Student.* These are known implicitly through object identity of the associated object instances of class *Student*.

  Hence it is not necessary to implement methods like attach or detach that are used to take care of more than one observer for a discrete event.

4.5.1 Lazy Notification

A naive implementation of the above design will not lead to a good performing system. The reason for this is not evident at first glance. But see the szenario in Picture 7



Picture 7: Lazy Notification

Instance A of class X is related to instances B,C and D of class Y. But only instance C of class Y is actually loaded in volatile storage. It would be very undesired, if a simple insertion of an instance E of class Y into the association or the deletion of e.g. instance B would result in loading B or E into volatile storage.

On the other hand instance E must be notified of it's association to A as it will not find this association in any database table as long as the insert is not commitet.

The result is a tradeoff:

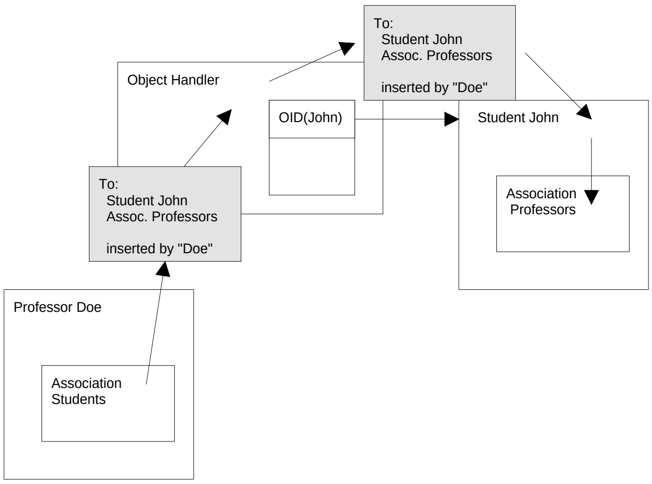
- we can have an optimum performing system with a complex central defered notification mechanism in which the messages that cannot be delivered to a volatile instance will be delivered lazy on loading an object from the relational database. This must be done by the object handler or some similar agent.

- we can have a system that is easier to understand with instant message delivery by accepting the fact that an object might be loaded from the database just for receiving the event that it has been inserted or deleted from an association.

- we should not allow that a store event on commiting the associations will wake up all related objects - in this case we might end up reading all objects from the database. This is a strong argument for the lazy propagation mechanism.

4.5.2 Implementing Event Notification

The proposed lazy event notification mechanism can be implemented as shown in Picture 8.



Picture 8: Lazy Event Notification

The association object *Students* in object instance *Professor Doe* will send a message via the object handler (which is also a derived class of the post office class TPsSubscriptionOffice that is addressed to the associated object instances (*Student John's*) association object *Professors*.

The message is delivered if and only if the referenced student is loaded in volatile storage. The object handler and also the post office can check that easily.

In case the Student object is not present in volatile storage the message will be queued and propagated on load of the student object. This solution is maximum lazy and nevertheless save.

4.5.3 Storing Associations

Only the mapper part of each persistent class will know how to store associations. It can therefore consult the association objects via an iterator. The association objects themselves will not know how to write themselves to a database. This knowledge is posession of the write2database methods. These are handwritten parts of the access layer.

The lazy event notification mechanism works fine here as well. The associations are usually stored in one place only in the relational database. Therefore it is sufficient if one of the objects, that are part of the association, writes them to the database. If the other object, that is part of the association, is not loaded into volatile storage, it will be notified *lazy* - meaning the message will be dumped after commiting the transaction.

4.6 Discussion of our Solution

The above solution is based on a design pattern for event notification, it is maximum lazy with regards to good performance and uses templates for encapsulting all common aspects that must otherwise be generated by a code generator.

As we have shown above handling associations in an object oriented access layer for relational databases is a very challenging task. Any solutions that are less complex than ours are welcome but we do not see a less complex solution within easy reach.

5 References

[ODMG 93] Rick G. G. Cattell (Ed.) et. al.: *Object Database Standard (ODMG 93);* Morgan Kaufmann Publishers, 1993.

[Pre 94] William Premerlani, Michal R. Blaha: *An approach for reverse engineering of relational databases;* Communications of the ACM, May 1994, p42(9).

[Gam 94] Erich Gamma et al., *Design Patterns,* Addison Wesley 1994.