

# **Mapping Objects To Relational Databases**

**An AmbySoft Inc. White Paper**

**Scott W. Ambler**  
Object-Oriented Consultant  
AmbySoft Inc.

Material for this White Paper has been modified from  
*Building Object Applications That Work*  
by Scott W. Ambler  
SIGS Books, 1997

<http://www.AmbySoft.com/mappingObjects.pdf>

**This Version: August 17, 1997**

**Copyright 1997 Scott W. Ambler**

## Table Of Contents

<b>1. THE OBJECT-RELATIONAL MISMATCH .....</b>	<b>1</b>
<b>2. THE IMPORTANCE OF OBJECT IDS .....</b>	<b>1</b>
<b>3. IMPLEMENTING INHERITANCE IN A RELATIONAL DATABASE.....</b>	<b>3</b>
<b>4. USE A DATA DICTIONARY.....</b>	<b>4</b>
<b>5. THE REALITIES OF MAPPING OBJECTS TO RELATIONAL DATABASES.....</b>	<b>4</b>
<b>6. SO WHAT'S WITH THE ATTITUDE PROBLEM? .....</b>	<b>6</b>
<b>7. SUMMARY .....</b>	<b>7</b>
<b>8. ABOUT THE AUTHOR.....</b>	<b>8</b>

This paper presents a practical look at the issues involved with mapping and should alleviate several common misconceptions prevalent in development circles today.

Issues that are addressed in this paper:

- The object-relational mismatch
- The importance of OIDs
- Implementing inheritance
- Use a data dictionary
- The realities of mapping objects to relational databases
- So what's with the attitude problem?

## 1. The Object-Relational Mismatch

The object paradigm is based on building applications out of objects that have both data and behavior, whereas the relational paradigm is based on storing data in the rows of tables. The “impedance mismatch” comes into play when you look at the preferred approach to access: With the object paradigm you traverse objects via their relationships whereas with the relational paradigm you join the rows in tables via their relationships. This fundamental difference results in a non-ideal combination of the two paradigms, although when have you ever used two different things together without a few hitches? One of the secrets of success for mapping objects to relational databases is to understand both paradigms, and their differences, and then make intelligent tradeoffs based on that knowledge.

## 2. The Importance of Object IDs

We need to assign unique identifiers to our objects so that we can distinguish between them. A very good way to accomplish this is to assign each object an attribute called an object identifier (OID). OIDs are typically implemented as very large integers that are used as the key values in relational database tables. The main advantage of OIDs is that your keys are as simple as possible – a single numeric field.

OIDs allow us to simplify our key strategy within a relational database. Although OIDs don't completely solve our navigation issue between objects (fundamentally relational databases simply aren't set up this way) they do make it easier. You still need to perform table joins to read in an aggregate of objects, such as an invoice and all of its line items, but at least it's doable.

Another advantage is that the use of OIDs also puts you into a position in which it is fairly easy to automate the maintenance of relationships between objects. When all of your tables are keyed on the same type of column, in this case OIDs, it becomes very easy to write generic code to take advantage of this fact.

A very critical issue that needs to be pointed out is that OIDs should have absolutely no business meaning whatsoever. Nada. Zip. Zilch. Zero. Any column with a business meaning can potentially change, and if there's one thing that we learned over the years in the relational world it's that it's a fatal mistake to give your keys meaning. If your users decide to change the business meaning, perhaps they want to add some digits or make the number alphanumeric, you need to make changes to your database in every single spot where you use that information. Anything that is used as a primary key in one table is virtually guaranteed to be used in other tables as a foreign key. What should be a simple change, adding a digit to your customer number, can be a huge maintenance nightmare. Yuck.

To give you an example, consider telephone numbers. Because phone numbers are unique many companies use them as keys for their customers. Although this sounds like a good idea, you're actually asking for trouble. I live near Toronto, Canada and because of the increased use of cellular phones, modems, and fax machines the local phone company was recently forced to divide the phone numbers of

the 416 area code between 416 and 905. What's less work, changing the phone numbers in a few tables that had them as non key columns or changing lots of tables that used them as either primary or foreign keys, not to mention the changes needed to your indexes? Moral of the story – OIDs should have no business meaning.

### **A Strategy for Assigning Object IDs**

When assigning object ids (OIDs) to objects there are two main issues that you need to address: The level of uniqueness of the OID and how to calculate it. The importance of the first issue, uniqueness, isn't always obvious to developers who are new to object orientation. There are three levels of uniqueness that you need to consider: uniqueness within the class, uniqueness within the class hierarchy, and uniqueness across all classes. For example, will the OID for a customer object be unique only for instances of customer, to people in general, or to all objects. Given the OID value 74656 will it be assigned to a customer object, an employee object, and an order object, will it be assigned to a customer but not an employee (because Customer and Employee are in the same class hierarchy), or will it only be assigned to a customer object and that's it. The real issue is one of polymorphism: It is probable that a customer object may one day become an employee object, but likely not an order object. To avoid this issue of reassigning OIDs when an object changes type, you at least want uniqueness at the class hierarchy level, although uniqueness across all classes completely avoids this issue.

The second issue, that of assigning OIDs, can greatly affect the runtime efficiency of your application. In the past (SD October 1996) I suggested that there are three basic approaches to assigning OIDs: Using the SQL MAX() function on a single table, maintain a single row table which stores the value of an incremental counter, and use MAX() with a quick write. Each of these techniques had a serious drawback: The MAX() approach requires a table lock to ensure uniqueness, the single row table quickly becomes a bottleneck when you use it for every OID in the application, and the MAX() with a quick write doesn't guarantee uniqueness. Since I wrote that article, however, I've come up with a fourth technique that avoids all of these problems.

The basic idea is that instead of using a large integer for the OID use a string in the format "HIGH:LOW" where HIGH and LOW are integer numbers concatenated by a colon. The value HIGH is obtained from a single row table in the database (or from a built in key value function for some databases) when the user first logs onto the system. HIGH is guaranteed to be unique for that user for that session. At this point the value for LOW is set at zero and is incremented every time an OID is needed for that user's session. For example, if you log onto to an application right now you'll be assigned the value 1701 for HIGH, and all OIDs that the application assigns to objects will be the strings '1701:1', '1701:2', and so on. If I log on immediately after you, I'll be assigned the HIGH value of 1702, and the OIDs that will be assigned to objects that I create will be '1702:1', '1702:2', and so on. The advantage of this approach is that the single row table is no longer as big of a bottle neck, there is very little network traffic needed to get the value for an OID (one database hit per session), and OIDs can be assigned uniquely across all classes. In other words, this approach is as simple and efficient as you're ever going to get it.

### 3. Implementing inheritance in a relational database

By using OIDs to uniquely identify our objects in the database we greatly simplify our strategy for database keys (table columns that uniquely identify records) making it easier to implement inheritance, aggregation, and instance relationships. First let's consider inheritance, the relationship that throws in the most interesting twists when saving objects into a relational DB. The problem basically boils down to "How do you organize the inherited attributes within the database?" The way in which you answer this question can have a major impact on your system design.

There are three fundamental solutions to implementing inheritance in a relational database:

1. **Use one table for an entire class hierarchy.** Map an entire class hierarchy into one table, where all the attributes of all the classes in the hierarchy are stored in it. The advantages of this approach are that it is simple – polymorphism is supported when a person either changes roles or has multiple roles (i.e., the person is both a customer and an employee) and it is easy to assign OIDs to objects because all of the objects are stored in one table. Ad hoc reporting is also very easy with this approach because all of the data you need about a person is found in one table. The disadvantages are that every time a new attribute is added anywhere in the class hierarchy a new attribute needs to be added to the table. This increases the coupling within the class hierarchy – If a mistake is made when adding a single attribute it could affect all the classes within the hierarchy and not just the subclasses of whatever class got the new attribute. It also wastes a lot of space in the database.
2. **Use one table per concrete class.** Each table includes both the attributes and the inherited attributes of the class that it represents. The main advantage of this approach is that it is still fairly easy to do ad hoc reporting as all the data you need about a single class is stored in only one table. It is still easy to assign OIDs to objects as long as polymorphism, the ability of objects to change their type, isn't an issue. There are several disadvantages however. First, when we modify a class we need to modify its table and the table of any of its subclasses. For example if we were to add height and weight to the person class we would need to add it in all three of our tables, a lot of work. Second, whenever an object changes its role, perhaps we hire one of our customers, we need to copy the data into the appropriate table and assign it a new OID, once again a lot of work. Third, it is difficult to support multiple roles and still maintain data integrity.
3. **Use one table per class.** Create one table per class, the attributes of which are the OID and the attributes that are specific to that class. The main advantage of this approach is that it conforms to object-oriented concepts the best. It supports polymorphism very well as you merely have records in the appropriate tables for each role that an object might have. It is also very easy to modify superclasses and add new subclasses as you merely need to modify/add one table. There are several disadvantages to this approach. First of all you end up with many tables in the database, one for every class. Second, it takes longer to read and write data using this technique because you need to access multiple tables. This problem can be alleviated if you organize your database intelligently by putting each table within a class hierarchy on different physical disk-drive platters (this assumes that the disk-drive heads all operate independently). Third, ad hoc reporting on your database is difficult if you don't add views on your data tables to make it easier.

Factors to Consider	One table per hierarchy	One table per concrete class	One table per class
Ease of implementation	Simple	Medium	Difficult
Ease of data access	Simple	Simple	Medium/Simple
Ease of assigning OIDs	Simple	Medium	Medium
Coupling	Very high	High	Low
Speed of data access	Fast	Fast	Medium/Fast
Support for polymorphism	Medium	Low	High

## 4. Use a Data Dictionary

A persistence layer only becomes useful when you combine it with a data dictionary because it reduces the coupling between your OO application and your RDB. You can implement your data dictionary as either a flat file or a table in your database.

The Information Stored in Your Data Dictionary
<p>At a minimum each row of your data dictionary should include the following fields:</p> <ul style="list-style-type: none"> <li>• The name of the class being mapped</li> <li>• The name of the attribute being mapped</li> <li>• The name of the column that the attribute is being mapped to</li> <li>• The name of the table that contains the column</li> </ul> <p>You should also consider including:</p> <ul style="list-style-type: none"> <li>• The name of the database the table is in (if your applications accesses several databases)</li> <li>• An indication if a column is being used as a key (in case you haven't standardized on OID as the key field)</li> <li>• An indication if a column is being used as a foreign key (in case you intend to automate the maintenance of relationships) and for what table it is a foreign key</li> </ul>

## 5. The Realities of Mapping Objects To Relational Databases

### Reality #1: Objects and Relational Databases Are the Norm

For years object gurus claimed that you shouldn't use relational databases to store objects because of the "object/relational impedance mismatch" (see sidebar). Yes, the object paradigm is different from the relational paradigm, but for 99% of you the reality is that your development environment is object oriented and your persistence mechanism is a relational database. Deal with it.

### Reality #2: ODBC and JDBC Classes Aren't Enough...

Although most development environments come with rudimentary access mechanisms to relational databases, they are at best a good start. Common "generic" mechanisms include Microsoft's Open Database Connectivity (ODBC) and Java's Java Database Connectivity (JDBC) – Most object development environments include class libraries that wrap one of these standard approaches.

The fundamental problem with these class libraries, as well as those that wrap access to native database drivers, are that they are too complex. In a well-designed library I should only have to send objects messages like **delete**, **save**, and **retrieve** to handle basic persistence functionality. The interface for working with multiple objects in the database isn't much more complicated (Thinking Objectively, SD January 1997). The bottom line is that the database access classes provided with your development environment are only a start, and a minimal one at that.

### **Reality #3: ...Therefore You Need a Persistence Layer**

A persistence layer encapsulates access to databases, allowing application programmers to focus on the business problem itself. This means that the database access classes are encapsulated providing a simple yet complete interface for application programmers. Furthermore, the database design should be encapsulated so that programmers don't need to know the intimate details of the database layout: that's what database administrators (DBAs) are for. A persistence layer completely encapsulates your permanent storage mechanism(s), sheltering you from changes.

The implication is that your persistence layer needs to use a data dictionary that provides the information needed to map objects to tables. When the business domain changes, and it always does, you shouldn't have to change any code in your persistence layer. Furthermore, if the database changes, perhaps a new version is installed or the DBA rearranges some tables, the only thing that should change is the information in the data dictionary. Simple database changes should not require changes to your application code, and data dictionaries are critical if you want to have a maintainable persistence approach..

### **Reality #4: Hard-Coded SQL is an Incredibly Bad Idea**

A related issue is one of including SQL (structured query language) code in your object application. By doing so you effectively couple your application to the database design, which reduces both maintainability and enhanceability. The problem is that whenever basic changes are made in the database, perhaps tables or columns are moved or renamed, you have to make corresponding changes in your application code. Yuck! A better approach is for the persistence layer to generate dynamic SQL based on the information in the data dictionary. Yes, dynamic SQL is a little slower but the increased maintainability more than makes up for it.

### **Reality #5: You Have to Map to Legacy Data...**

Although the design of legacy databases rarely meet the needs of an object-oriented application, the reality is that your legacy databases are there to stay. The push for centralized databases in the 1980s has now left us with a centralized disaster: Database schemas that are difficult to modify because of the multitude of applications coupled to them. The implication is that few developers can truly start fresh with a relational database design that reflects their object-oriented design, instead they must make do with a legacy database.

### **Reality #6: ...But The Data Model Doesn't Drive Your Class Diagram**

Just because you need to map to legacy data it doesn't mean that you should bastardize your object design. I've seen several projects crash in flames because a legacy data model was used as the basis for the class diagram. The original database designers didn't use concepts like inheritance or polymorphism in their design, nor did they consider improved relational design techniques (see below) that become apparent when mapping objects. Successful projects model the business using object-oriented techniques, model the legacy database with a data model, and then introduce a "legacy mapping layer" that encapsulates the logic needed to map your current object design to your ancient data design. You'll sometimes find it

easier to rework portions of your database than to write the corresponding mapping code, code that is convoluted because of either poor or outdated decisions made during data modeling.

### **Reality #7: Joins are Slow**

You often need to obtain data from several tables to build a complex object, or set of objects. Relational theory tells you to join tables to get the data that you need, an approach that often proves to be slow and untenable for live applications. Therefore don't do joins! Because several small accesses are usually more efficient than one big join you should instead traverse tables to get the data. Part of overcoming the object/relational impedance mismatch (see sidebar) is to traverse instead of join where it makes sense. Try it, it works really well.

### **Reality #8: Keys With Business Meaning Are a Bad Idea...**

Experience with mapping objects to relational databases leads to the observation that keys shouldn't have business meaning, which goes directly against one of the basic tenets of relational theory. The basic idea is that any field that has business meaning is out of the scope of your control and therefore you risk having its value or its layout change. Trivial changes in your business environment, perhaps customer numbers increase in length, can be expensive to change in the database because the customer number attribute is used in many places as a foreign key. Yes, many relational databases now include administration tools to automate this sort of change, but even so it's still a lot of error-prone work. In the end I believe that it simply doesn't make sense for a technical concept, a unique key, to be dependent on business rules.

### **Reality #9: ...And So Are Composite Keys**

While I'm attacking the sacred values of DBAs everywhere, composite keys (keys made up of more than one column) are also a bad idea. Composite keys increase the overhead in your database as foreign keys, increase the complexity of your database design, and often incur additional processing requirements when many fields are involved. My experience is that an object id (OID), a single column attribute that has no business meaning and which uniquely identifies the object, is the best kind of key. Ideally OIDs are unique within the scope of your enterprise-wide database(s), in other words any given row in any given table has a unique key value. OIDs are simple and efficient, their only downside is that experienced relational DBAs often have problems accepting them at first (although fall in love with them over time).

### **Reality #10: You Need Several Inheritance Strategies**

There are three fundamental solutions (SD October 1995) for implementing inheritance in a relational database: use one table for an entire class hierarchy; use one table per concrete class; or use one table per class. Although all three approaches work well, none of them are ideal for all situations. The end result is that your persistence layer will need to support all three approaches at some point, although implementing one table per concrete class at first is the easiest way to start.

## **6. So What's With The Attitude Problem?**

In this section I want to address why we keep hearing that mapping doesn't work. First, let's go for an easy kill – employees of object database companies. I shouldn't have to point out that OODB people have a stake in shooting down relational databases. Don't get me wrong, I really like OODB, but I am a realist – I'll listen to what OODB people have to say when they're talking about object databases, but when they're talking about relational databases I listen with a grain of salt.

The second problem are the articles that talk about C++ projects that ran aground when they used relational technology. When you actually read these articles in detail, especially from the eye of someone with experience in more than C++, you quickly realize that most of their problems lie with C++ and its inherent difficulties, and not with mapping objects to RDBs. You really need to read between the lines with a lot of these articles.

The third problem lies in not understanding all of the realities mentioned earlier. You need to encapsulate your database. You need to use OIDs effectively. You need to let your class diagram drive your database design. You shouldn't wrap a legacy database. If you ignore this advice you risk running into serious trouble.

I challenge you to go back and re-read any anti mapping articles you have read in the past. I challenge you to question the advice of so-called object gurus who claim that mapping isn't a good idea. When you think for yourself I believe that you will see that mapping objects to relational databases is a very viable approach to object persistence.

## 7. Summary

Considering the investment in legacy data that exists today, and the reluctance of organizations to move away from it, I suspect that organizations will be mapping objects to relational databases for years to come. I also believe that relational databases will evolve in time. Evolution, not revolution, will be the name of the game for the vast majority of organizations. Whether or not this will be the best strategy only time will tell.

In this paper I discussed the realities of mapping objects to relational databases. Regardless of what the object gurus tell you relational databases are the norm, not the exception, for storing objects. Yes, the object/relational impedance mismatch means that you need to rethink a couple of relational tenets, but that's not a big deal. The material in this paper is based on my real-world experiences, it is not academic musings, and I hope that I've shattered a few of your misconceptions about this topic. You really can map objects successfully.

## 8. About the Author

Scott W. Ambler is a object development consultant living in the village of Sharon, Ontario, which is 60 km north of Toronto, Canada. He has worked with OO technology since 1990 in various roles: Business Architect, System Analyst, System Designer, Smalltalk programmer, Java programmer, and C++ programmer. He has also been active in education and training as both a formal trainer and as an object mentor.

Scott has a Master of Information Science and a Bachelor of Computer Science from the University of Toronto and is the author of the best-selling books *The Object Primer* and *Building Object Applications That Work*, both published by SIGS Books (<http://www.sigs.com>). Scott has published several white papers about OO development, including the *AmblySoft Inc. Java Coding Standards* which can be downloaded free of charge from his personal web site (<http://www.amblysoft.com>). Scott is a contributing editor with *Software Development* (<http://www.sdmagazine.com>), writes a column for *Computing Canada* (<http://www.plesman.com>), and has had feature articles appear in *Object Magazine* and *Client/Server Computing*.

He can be reached via e-mail at:

**[ambler@hookup.net](mailto:ambler@hookup.net)**

and you can visit his personal web site:

**<http://www.amblysoft.com>**

### About The Object Primer

*The Object Primer* is a straightforward, easy to understand introduction to object-oriented analysis and design techniques. Object-orientation is the most important change to system development since the advent of structured methods. While OO is often used to develop complex systems, OO itself does not need to be complicated. This book is different than any other book ever written about object-orientation (OO) – It's written from the point of view of a real-world developer, somebody who has lived through the difficulty of learning this exciting new approach. Readers of *The Object Primer* have found it to be one of the easiest introductory books in OO development on the market today, many of whom have shared their comments and kudos with me. Topics include **CRC modeling**, **use cases**, **use-case scenario testing**, and **class diagramming**.

### About Building Object Applications That Work

*Building Object Applications That Work* is about: **architecting** your applications so that they're maintainable and extensible; analysis and design techniques using the **Unified Modeling Language (UML)**; creating applications for stand-alone, **client/server**, and **distributed** environments; using both **relational** and object-oriented (OO) databases for persistence; OO **metrics**; applying OO **patterns** to improve the quality of your applications; OO **testing** (it's harder, not easier); **user interface design** so your users can actually work with the systems that you build; and **coding** applications in a way that makes them **maintainable** and **extensible**.

Uses the  
  
 Unified  
 Modeling  
 Language

### About the AmbySoft Inc. Java Coding Standards

The *AmblySoft Inc. Java Coding Standards* summarizes in one place the common coding standards for Java, as well as presents several guidelines for improving the quality of your code. It is in Adobe PDF format and can be **downloaded free of charge** from <http://www.amblysoft.com>.