Mapping Objects To Relational Databases

Scott W. Ambler
President, Ronin International



Portions of this white paper have been modified from Scott W. Ambler's books, including

Building Object Applications That Work,
Process Patterns
and
The Object Primer 2nd Edition

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

This Version: October 21, 2000

Copyright 1998-2000 Scott W. Ambler

Change History

March/April 1998:
The following additions have been made to this document:

- A discussion of determining the value of object identifiers (OIDs) in a distributed environment
- A discussion of determining the value of OIDs in a multi-vendor database environment
- A discussion of replicating objects across persistence mechanisms
- An example showing the various ways to map inheritance.
- A discussion of the process patterns applicable to mapping objects to relational databases
- A discussion of the issues involved with performing an initial data load

Special thanks to Chris Roffler and Ben Bovee for pointing out needed improvements.

May 16th, 1998:
I removed portions of the original document and moved them into my new persistence layer design white paper, downloadable from *http://www.ambysoft.com/persistenceLayer.html*.

June 1st, 1998:
Minor updates to diagrams and pagination improvements

November, 1998:

- Fixes to inheritance section.
- Career advice for data modelers added.
- Process patterns for OO modeling added.
- Minor spelling and grammar updates.

February, 1999

- Updates to mistakes in diagrams
- Improved discussion of why using data models as the basis for your OO model isn't advisable

July and October 2000

- Updated with material from The Object Primer 2nd Edition
- Changed to reflect the fact that Ambysoft has partnered with Ronin International (www.ronin-intl.com)
- Inserted the "blatant advertising" which I realize is a little annoying but the reality is that you get a really solid white paper at the cost of having to endure advertising for some great books and a really good software consulting company.

Table Of Contents

1. THE OBJECT-RELATIONAL MISMATCH....1

2. THE IMPORTANCE OF OBJECT IDS....1

2.1 OIDs Should Have No Business Meaning....2
2.2 OID Uniqueness....2
2.3 Strategies for Assigning OIDs....3
2.3.1 Using MAX() on an Integer Column....3
2.3.2 Maintaining a Key-Values Table....3
2.3.3 GUIDs/UUIDs....3
2.3.4 Proprietary Persistence Mechanism Features....3
2.3.5 The HIGH/LOW Approach To OIDs....4
2.3.5.1 Implementing a HIGH/LOW OID....5
2.3.5.2 HIGH/LOW OIDs In a Distributed Environment....6
2.3.5.3 HIGH/LOW OIDs In A Multi-Vendor Environment....6
2.4 Replication of Objects....6

3. THE BASICS OF MAPPING....7

3.1 Mapping Attributes To Columns....7
3.2 Mapping Classes To Tables....8
3.2.1 Implementing Inheritance in a Relational Database....8
3.2.2 Mapping Several Classes To One Table....11
3.3 Mapping Relationships....12
3.3.1 The Difference Between Association and Aggregation....12
3.3.2 Implementing Relationships in Relational Databases....13
3.3.3 Implementing Many-To-Many Relationships....14
3.3.4 The Same Classes/Tables, Different Relationships....15

4. CONCURRENCY, OBJECTS, AND ROW LOCKING....16

4.1 Pessimistic Vs. Optimistic Locking....16

5. STORED PROCEDURES....16

6. TRIGGERS....17

7. PROCESS PATTERNS FOR MAPPING OBJECTS TO RDBS....18

7.1 Why is The Persistence Modeling Process Pattern Important?....21
7.2 The Implications?....23

8. THE REALITIES OF MAPPING OBJECTS TO RELATIONAL DATABASES....24

8.1 Objects and Relational Databases Are the Norm....24
8.2 ODBC and JDBC Classes Aren't Enough…....24
8.3 Therefore You Need a Persistence Layer....25
8.4 Hard-Coded SQL is an Incredibly Bad Idea....25
8.5 You Have to Map to Legacy Data…....26
8.6 …But The Data Model Doesn't Drive Your Class Diagram....26
8.7 Joins are Slow....26
8.8 Keys With Business Meaning Are a Bad Idea…....27
8.9 …And So Are Composite Keys....27
8.10 You Need Several Inheritance Strategies....27
8.11 Stored Procedures Are a Bad Idea....27

9. SO WHAT'S WITH THE ATTITUDE PROBLEM? 28

10. SUMMARY 28

11. REFERENCES AND RECOMMENDED READING 29

12. ABOUT THE AUTHOR 30

13. GLOSSARY OF TERMS 31

14. INDEX 35

This paper presents a practical look at the issues involved with mapping objects[1] to relational databases and should alleviate several common misconceptions prevalent in development circles today. Before you read any further, this paper is assumes that you want to develop object-oriented applications that are easy to extend and to maintain, that you are willing to invest the time during development to determine a persistence strategy that will achieve these aims. If your goal is to simply bang out a small application as quickly as you can, ignoring quality, then stop reading right now and just start hacking out some code. If your goal is to build something that will add long-term value to your organization, to build a quality object-oriented application, then read on. You have to make a conscious decision to do things right, and the first step is to take the time to understand what the right and wrong things are. This paper discusses many of the principles involved for successfully mapping objects to relational databases.

You must consciously choose to build a quality application, and that takes time and an understanding of the basics. This paper presents the basics of mapping objects to relational databases.

The material in this paper should be taken as a collection of strategies that you should follow whenever possible, and if you go against them then you should have a valid reason for doing so and know the implications of doing so. The strategies are based on my development experiences from small projects of several people to large projects of several hundred people, on projects in the financial, distribution, military, telecommunications, and outsourcing industries. I've applied these principles for applications written in C++, Smalltalk, Visual Basic, and Java. The bottom line is that the material in this white paper is based on real-world experiences on a wide variety of projects. I hope that you find this paper of use.

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. 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 duplicate data to join the rows in tables. 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 identify them. In relational terminology a unique identifier is called a key, in object terminology it is called an object identifier (OID) although perhaps persistent object identifier would be a better term. OIDs are typically implemented as full-fledged objects in your OO applications and as large integers, or several large integers for larger applications, in your relational schema. Figure 1 presents a diagram showing a possible implementation of an OID class and Figure 2 shows how an OID might be mapped to a column(s) in a table.

Object identifiers (OIDs) are used to uniquely identify objects in a relational database.

OIDs allow us to simplify our key strategy within a relational database. Although OIDs don't completely solve our navigation issue between objects they do make it easier. You still need to perform table joins,

[1] This white paper does not discuss the design of a persistence mechanism in detail, the topic http://www.ambysoft.com/persistenceLayer.html

assuming you don't intend to traverse, 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(s), in this case OIDs, it becomes very easy to write generic code to take advantage of this fact.

2.1 OIDs Should Have No Business Meaning

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. In the relational database world, this OID strategy is referred to as employing surrogate keys.

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.

2.2 OID Uniqueness

When assigning object IDs (OIDs) 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.

An OID should be unique within a class hierarchy, and ideally unique among all objects.

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.

2.3 Strategies for Assigning OIDs

The second issue, that of determining new OIDs, can greatly affect the runtime efficiency of your application.

2.3.1 Using MAX() on an Integer Column

In the past (Ambler, 1996) I suggested that one approach to assigning OIDs was to use the SQL MAX() function on an integer column being used as a primary key. The basic idea is that when you insert a new row into the table that you take the MAX() of that column, add one to it, and then use that as the value for your key. The problems with this approach is that you do a momentary table lock, although many databases are optimized for this approach, and that you don't have unique values for OIDs across all your objects, only for those stored within each table.

2.3.2 Maintaining a Key-Values Table

There are two flavors of this approach, the first where you maintain a single row with a single integer column in which a counter value is stored. When you insert into any table you first lock and increment this value and use it as the OID value. The advantage of this approach is that you avoid locking the other tables when you invoke MAX() and that you have a unique value across all objects. The disadvantages are that this table becomes a bottleneck (although you can cache this value in memory if needed). and that you quickly run through OID values.

The second flavor is to use a multi-row table where you have one row per table in your system, where each row has two columns, the first column is an identifier for the table name and the second is the integer value of the next key value for that table. Once again you avoid table locking with this approach but now have a greater range of OID values because each table now has it's own counter. Now you're back to losing uniqueness of OID values between objects and this table still potentially becomes a bottleneck (although you can cache these values in memory if needed).

2.3.3 GUIDs/UUIDs

Several years ago Digital came up with a strategy called UUIDs for determining generating unique key values based on hashing the value of the identification number of the Ethernet card within your computer and the current date and time, producing a 128-bit string that was guaranteed unique. For computers without Ethernet cards you could obtain a identification number stored online in a file. Microsoft has a similar strategy called GUIDs that also results in a 128-bit string.

Both of these strategies work well, although both strategies are proprietary and do not run on all platforms. There is also the philosophical issue to deal with, although I don't think it's a big deal, of not having your persistence mechanism(s) assign OID values for you.

2.3.4 Proprietary Persistence Mechanism Features

Some databases, for example Oracle, have built in features for generating unique values that can be used for OID values. Although these approaches work well, they are by definition proprietary and therefore out of your control. If you ever have to port to a new persistence mechanism, an event that is far more common and likely than your database/persistence community is willing to admit, then this can become a serious issue for you.

Scott's General Design Philosophy: Proprietary Approaches Will Always Burn You

I can't be more emphatic about this: you really need to think before accepting a proprietary technology within your environment. Yes, there are always performance reasons, and often ease-of-development reasons, but never forget that you also need to worry about "itty-bitty" issues such as portability, reliability, scalability, enhancability, and maintainability. I've seen a lot of organizations get themselves into serious trouble by using proprietary features that lock them into technologies that later prove to be insufficient for their futures needs. Later in this paper I'll discuss the evils of using stored procedures, amazingly enough still proprietary technology after all of these years.

2.3.5 The HIGH/LOW Approach To OIDs

The basic idea is that instead of using a large integer for the OID, requiring you to go to a single source (and therefore a bottleneck) to obtain the OID, you reorganize your OID into two logical components: A HIGH value that you obtain from a single source and a LOW value that the application assigns itself. 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 application first needs to create an OID. Because HIGH is obtained from a single source it is guaranteed to be unique. 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 the application that you're running requests a value for HIGH it will be assigned the value 1701 for HIGH, and all OIDs that the application assigns to objects will be combination of 1701 & 1, 1701 & 2, and so on. If my application requests a value for HIGH immediately after you it will given the 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. Yes, because it's possible that your value of LOW could be incremented to the maximum value that your design calls for, for example if LOW is four digits then the maximum value is 9999, therefore at this point you'll need to fetch a new value of HIGH and reset the value of LOW.

2.3.5.1 Implementing a HIGH/LOW OID

Take an OO approach to OIDs and encapsulate their behavior in a class.

Figure 1 presents one way to implement an OID class. The basic idea is that when a new persistent object is created it is assigned an OID which is created by the single instance of **Object Factory**. The sole responsibility of Object Factory is to create new OID objects. It does this by keeping track of the next HIGH and LOW values, having to fetch a HIGH value from the persistence mechanism (database) occasionally to do so. It creates an instance of **OID** based on the next values and returns it to be used as the unique OID for the new persistent object. The **asColumns** method returns a collection of data items that can be saved to a relational database to represent the instance of **OID**.

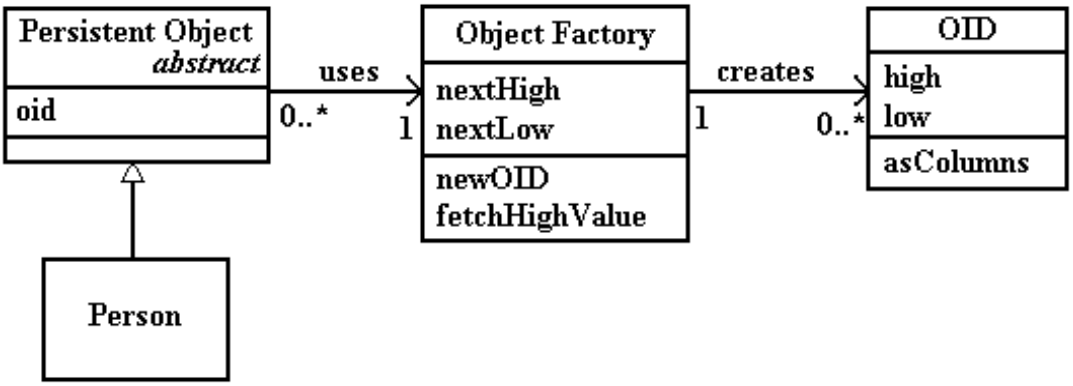


Figure 1. A class diagram showing a possible way to implement a HIGH/LOW OID.

Figure 2 shows three of the possible strategies for how a HIGH/LOW OID might be implemented in your relational database. Neither strategy is ideal, each has its advantages and disadvantages, the important thing is that you choose one strategy and use it consistently for all tables, making your persistence layer (Ambler, 1998c) that much easier to develop. The first strategy uses a single integer number for the key, simplifying your key strategy but putting a fairly small upper limit on the value of your OIDs (the largest integer number your database handles). The second strategy alleviates the size problem, you store the OID as a string of an arbitrary size that you choose, but increases the overhead of your key by using a large string (many databases are optimized to use single large integers as the primary keys but are slower for strings). The third strategy uses a composite key for your OID, the disadvantages of composite keys are discussed later in this paper, potentially reducing the storage overhead of your keys (as compared to the second strategy).

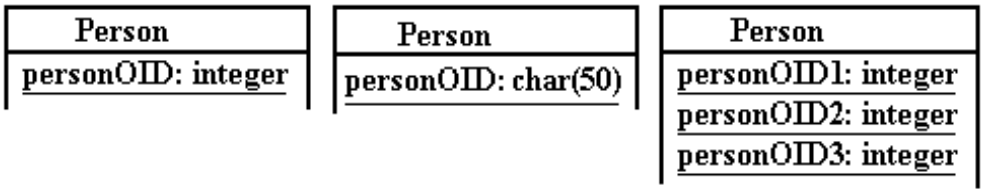


Figure 2. Mapping a HIGH/LOW OID to a relational table.

A fourth strategy, a combination of the second and third, involves hashing the OID object to a string. The basic idea is that you convert the OID into a collection of 8-bit numbers and then build a string from the numbers by first converting each number to a character and then concatenating the characters. For example, if your OID is built from a 32-bit HIGH number and an 8-bit LOW value then you would save them out as a 5-character string, the first four characters representing the high number (32/8 = 4) and the fifth representing the LOW value.

Scott's Preferred Approach

I typically use a 96-bit high number (often the combination of three 32-bit integers) for HIGH and a 32-bit integer for low. I then convert it to a 128-bit string taking the performance hit on using it as a key. The main disadvantage of this is that some databases have not yet been optimized for keys of fixed-length strings (instead they are optimized for Integers). This will change in time as developers demand support for more sophisticated approach to keys.

2.3.5.2 HIGH/LOW OIDs In a Distributed Environment

As mentioned previously, OIDs must be unique. But how do you guarantee uniqueness in a distributed environment where a client may obtain the HIGH value for its OID from a number of servers? The answer is simple: the server machines must also have a strategy for obtaining unique HIGH values between themselves. There are several strategies to doing so. First, the servers could use their own internal HIGH/LOW approach, obtaining a HIGH value from a single source. Second, the servers could obtain a block of HIGH values from a centralized source, obtaining a new block when they run out.

2.3.5.3 HIGH/LOW OIDs In A Multi-Vendor Environment

A related issue occurs when your organization uses persistence mechanisms produced by several vendors, a common occurrence in large organizations. Although several database vendors have strategies for producing surrogate keys, even in distribute environments, these strategies are typically applicable only for the products that they sell. The issue boils down to this: if the proprietary key value approach of Vendor A produces a HIGH value of 1701, and the proprietary key value approach of Vendor B also produces a HIGH value of 1701, then you're completely screwed. The implication is that you will need to roll your own strategy in such an environment, likely one of the strategies mentioned in the previous section.

2.4 Replication of Objects

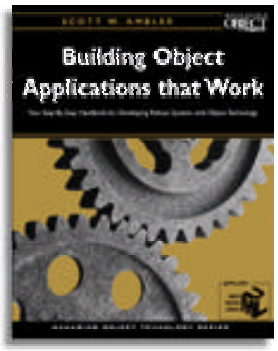
Replication is the issue of ensuring the information is kept up-to-date and synchronized when it is being stored and potentially updated at several physical locations. A primary consideration with the replication of data is the manner in which they are identified, and by having OIDs that are unique across all persistence mechanisms, regardless of vendor, ensures that you are able to determine uniquely identify an object no matter where and when it was created. There are significantly more issues involved with replication than just uniquely identifying objects, the point to be made is that OIDs are an enabler of replication.

3. The Basics of Mapping

In this section I will describe some of the basic issues of mapping (Ambler, 1998a; Ambler, 2001) objects into relational databases. These issues are:

- Mapping attributes to columns
- Mapping classes to tables
  - Implementing inheritance in an RDB
  - Mapping several classes to a single table
- Mapping relationships
  - One-to-one
  - One-to-many
  - Many-to-many
  - Association vs. aggregation
  - Same classes/tables, different relationships

Blatant Advertising – Purchase *Building Object Applications That Work* today!



Building Object Applications That Work is an intermediate-level book about object-oriented development. It covers a wide range of topics that few other books dare to consider, including: architecting your applications so that they're maintainable and extensible; OO analysis and design techniques; how to design software for stand-alone, client/server, and distributed environments; how to use both relational and object-oriented (OO) databases to make your objects persistent; OO metrics, analysis and design patterns; OO testing; OO user interface design; and a multitude of coding techniques to make your code robust. Visit www.ambysoft.com/buildingObjectApplications.html for more details.

3.1 Mapping Attributes To Columns

Attributes map to columns.

The attribute of a class will map to zero or more columns in a relational database. Remember, not all attributes are persistent. For example, an **Invoice** class may have a **grandTotal** attribute that is used by instances for calculation purposes but that isn't saved to the database. Furthermore, because some attributes of an objects are objects in their own right, a **Customer** object has an Address object as an attribute, sometimes a single OO attribute will map to several columns in the database (actually, chances are that the **Address** class will map to one or more tables in its own right). The important thing is that this is a recursive definition: At some point the attribute will be mapped to zero or more columns.

3.2 Mapping Classes To Tables

Classes map to tables, although often not directly. Except for very simple databases, you will never have a one-to-one mapping of classes to tables. In this section we will explore three different strategies for implementing inheritance structures to a relational database and see an example where dissimilar classes map to one table.

3.2.1 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 for mapping inheritance into 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). 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. 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 (it's possible, just harder than it needs to be).

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, there are many tables in the database, one for every class (plus tables to maintain relationships). 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, unless you add views to simulate the desired tables.

Let's consider an example. Figure 3 presents a UML (Rational, 1997) class diagram of a simple class hierarchy (the methods aren't shown) that I will map using each strategy, the resulting data models for which are shown in Figure 4. Notice how each strategy results in a different data model. Also notice how OIDs have been used for the keys. For the one table per class strategy the OID is used as both the primary key and as the foreign key to the **Person** table.

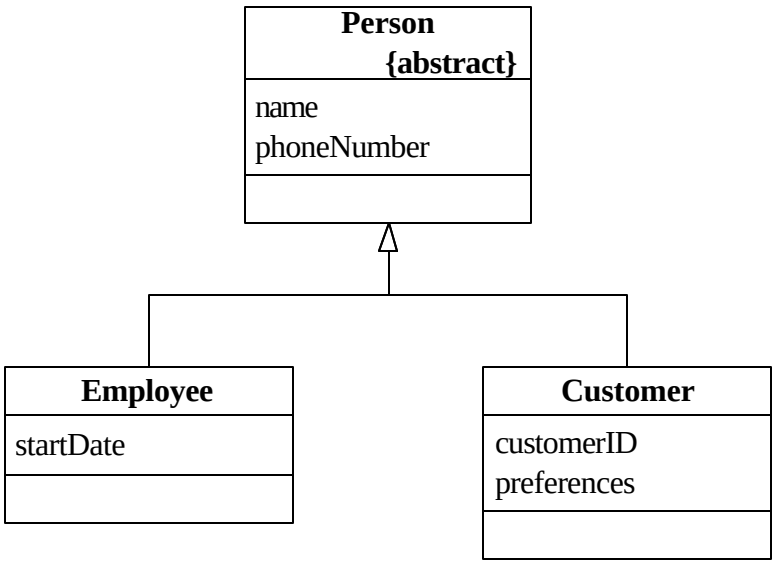


Figure 3. A UML class diagram of a simple class hierarchy.

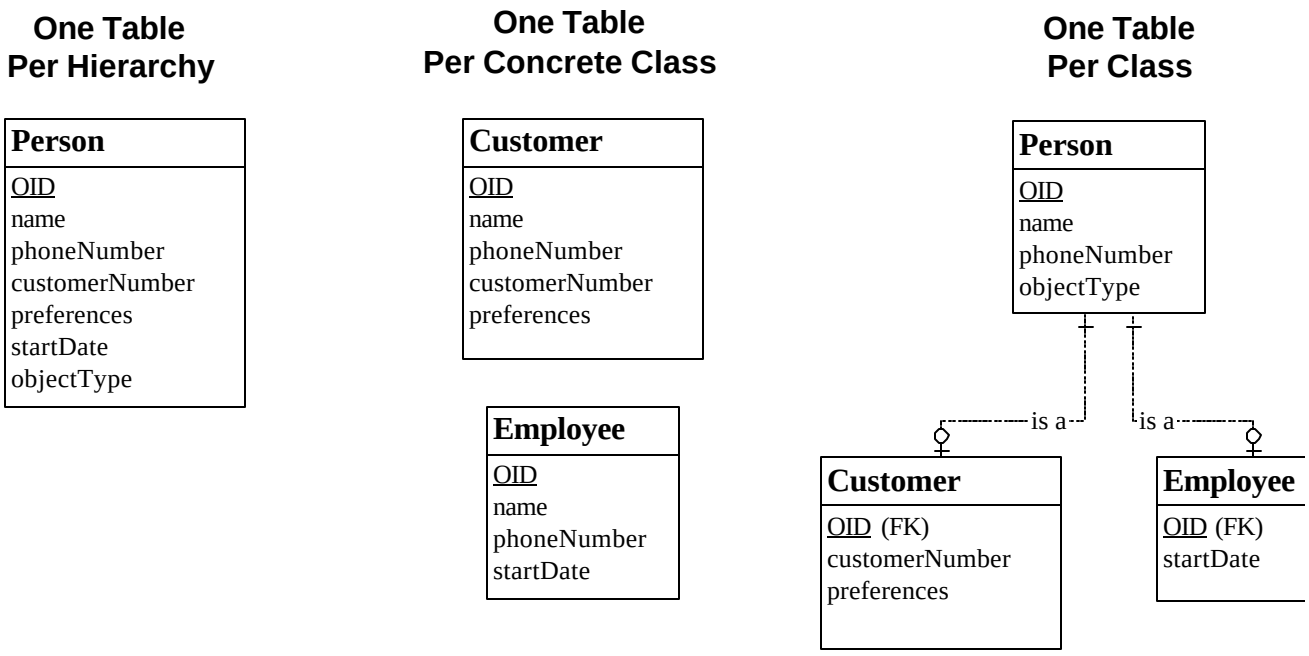


Figure 4. Mapping the class hierarchy using each strategy.

An interesting issue with the second mapping strategy, using one table per concrete class, is what do you do when you have someone who is both an employee and a customer[2]? The basic issue is what table is the authorative source for the name and phone number of the person? This is something that you will need to

[2] The use of the relationship "is a" in Figure 4 is not inheritance in this case, implied by the fact that the relationship is not mandatory. Sometimes an employee is a customer, sometimes they are not. Yet another example of the object/relational impedance mismatch.

decide, as well as rules for what to do when people (no longer) become an employee or customer, and then support it in your code.

To understand the design trade offs between the three strategies, consider the simple change to our class hierarchy presented in Figure 5: an **Executive** class has been added which inherits from **Employee**. Figure 6 presents the updated data models. Notice how very little effort was required to update the one table per hierarchy strategy, although the obvious problem of wasted space in the database has increased. With the one table per concrete class strategy we only needed to add a new table, although the issue of how do we handle objects that either change their relationship with us (customers become employees) has now become more complex because we've added the issue of promoting employees to become executives. With the third mapping strategy, mapping a single class to a single table, we needed to add a new table, one that included only the new attributes of Executive. The disadvantage of this approach is that it requires several database accesses to work with instances of Executive. The point to go away with is that none of the approaches are perfect, that each has its strengths and weaknesses (which are summarized in Table 1).

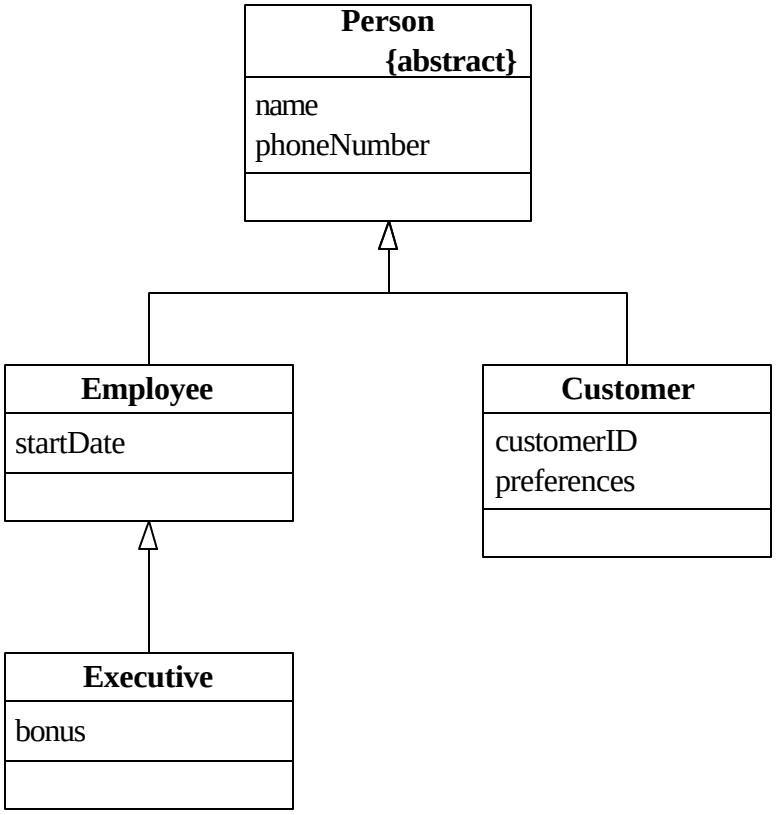


Figure 5. Extending the hierarchy.

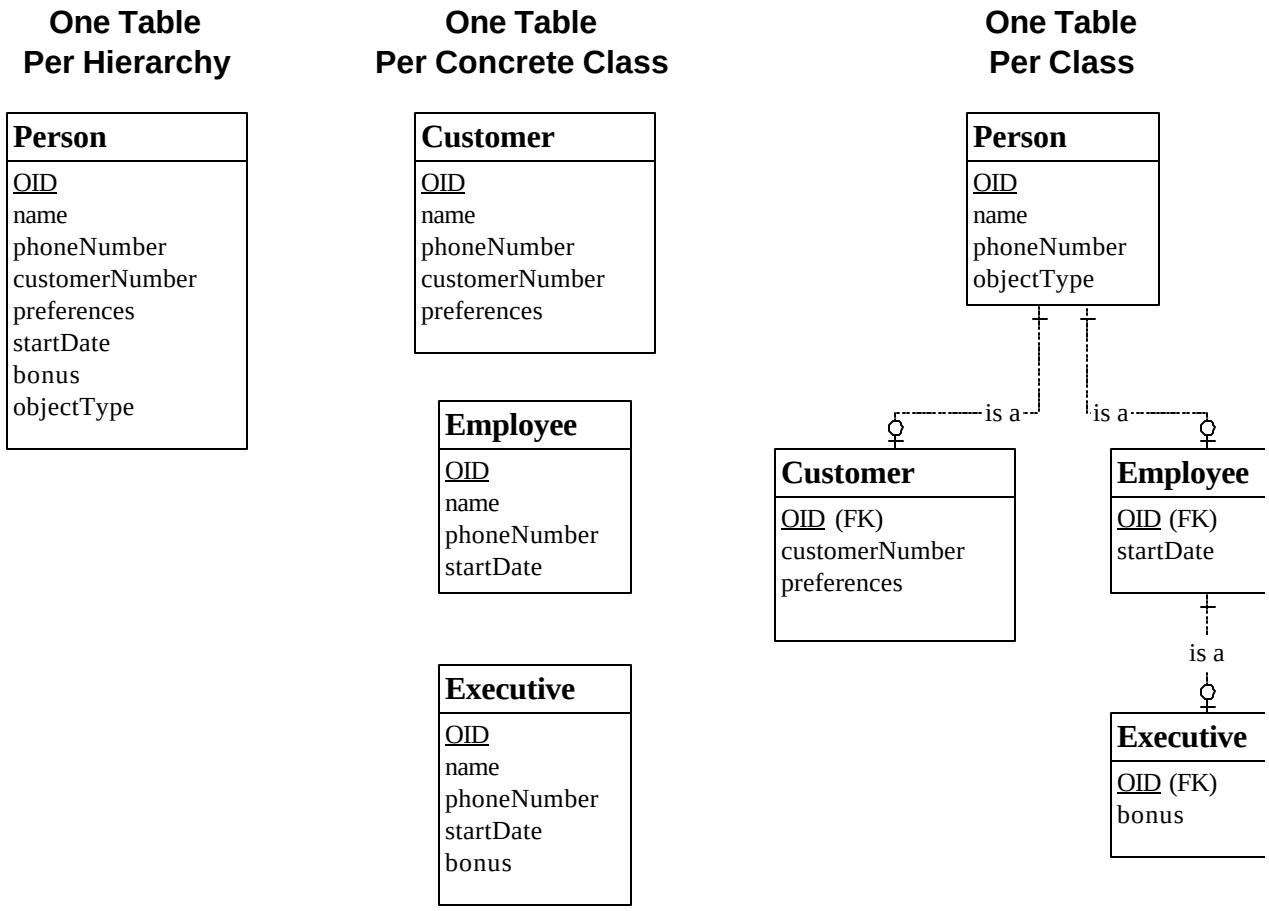


Figure 6. Extending the data models.

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

Table 1. Comparing the approaches to mapping inheritance.

3.2.2 Mapping Several Classes To One Table

Figure 7 shows a class diagram of the implementation of an address that uses two classes, **Address** and **Zip Code**. The **Zip Code** class was created to encapsulate the logic of validating a zip code number and formatting it appropriate for mailing labels. For example is the state correct (in the U.S. the first two digits of a zip code indicate the state) and hyphens should be inserted at appropriate places when the zip code is

output (the U.S. post office recently introduced a long version of a zip code with three sections to it). The bottom line is that the **Zip Code** class encapsulates cohesive behavior that is relevant to zip codes.

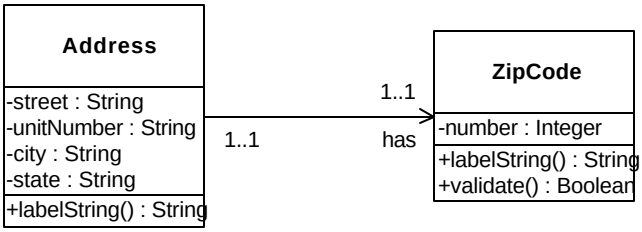


Figure 7. A UML class diagram representing a simple OO implementation of an address.

In a relational database, however, the behavior implemented by the **Zip Code** class isn't relevant, therefore a zip code can map to a single column in the **Address** table. Figure 8 shows the data model for **Address** for this particular situation (we'll revisit this example later in the paper for an alternative data model design based on new requirements). The lesson to be learned is that sometimes two dissimilar classes will map to one table. This occurs because class diagrams take a greater scope than data models – they show data and behavior whereas data models only show data.

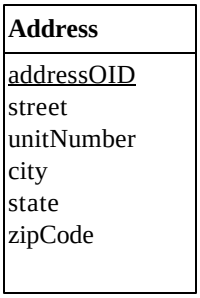


Figure 8. A data model showing a possible implementation for an address.

3.3 Mapping Relationships

Not only do we need to map objects into the database, we also need to map the relationships that the object is involved with so they can be restored at a later date. There are two types of relationship that an object can be involved with: association and aggregation. To map these relationships effectively we must understand the difference between them, how to implement relationships generally, and how to implement many-to-many relationships specifically.

3.3.1 The Difference Between Association and Aggregation

From a database perspective the only difference between association and aggregation relationships is how tightly the objects are bound to each other. With aggregation anything that you do to the whole in the database you almost always need to do to the parts, whereas with association that isn't the case.



Figure 9. The difference between instance and aggregation relationships.

In Figure 9 (Ambler, 1998a) we see three classes, two of which have a simple association between them and two which share an aggregation relationship. From a database point of view aggregation and association are different in the fact that with aggregation you usually want to read in the part when you read in the whole, whereas with an association it isn't always as obvious what you need to do. The same goes for saving objects to the database and deleting objects from the database. Granted this is usually specific to the business domain, but this rule of thumb seems to hold up in most circumstances. The differences between aggregation and association are discussed in further detail in my second book, *Building Object Applications That Work* (Ambler, 1998a).

3.3.2 Implementing Relationships in Relational Databases

Relationships in relational databases are maintained through the use of *foreign keys*. A foreign key is a data attribute(s) that appears in one table that may be part of or is coincidental with the key of another table. Foreign keys allow you to relate a record in one table with a record in another. To implement one-to-one and one-to-many relationships you merely have to include the key of one table in the other table. Let's look at an example.

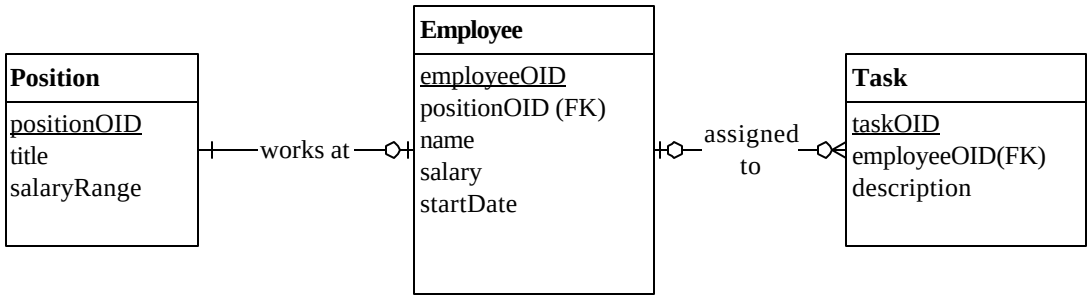


Figure 10. Implementing relationships in a relational database.

In Figure 10[3] (Ambler, 1998a) we see three tables, their keys (OIDs of course), and the foreign keys used to implement the relationships between them. First, we have a one-to-one relationship between **Position** and **Employee**. To implement this relationship we added the attribute **positionOID**, which is the key of **Position**, although we could just as easily have added a foreign key called **employeeOID** in **Position** instead. Second, we implement the many-to-one relationship (also referred to as a one-to-many relationship) between **Employee** and **Task** using the same sort of approach, the only difference being that we had to put the foreign key in **Task** because it was on the many side of the relationship.

3.3.3 Implementing Many-To-Many Relationships

To implement many-to-many relationships we need to introduce the concept of an associative table, a table whose sole purpose is to maintain the relationship between two or more tables in a relational database. In Figure 11 (Ambler, 1998a) we see that there is a many-to-many relationship between customers and accounts. In Figure 12 (Ambler, 1998a) we see how to use an associative table to implement a many-to-many relationship. In relational databases the attributes contained in an associative table are traditionally the combination of the keys in the tables involved in the relationship. It has been my experience, however, that it is easier to implement associative tables if you treat them as just another type of table – You assign them their own key field, in our case **OID**, and then add the necessary foreign keys to maintain the relationship.

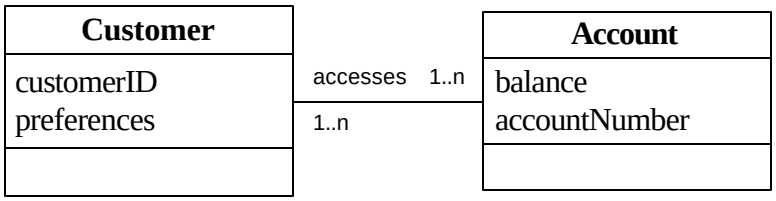


Figure 11. Two classes with a many-to-many relationship between them.

The advantage of this is that all tables are treated the same by your persistence layer, simplifying its implementation. Another advantage is one of run-time efficiency: some relational databases have problems joining tables that have many attributes in their keys. There is also the possibility that one or more columns may be added to the **Accesses** table to represent the security access rights for the customer on the account. Perhaps a given customer can deposit money into an account, but not withdraw, whereas another customer has full access to the account.

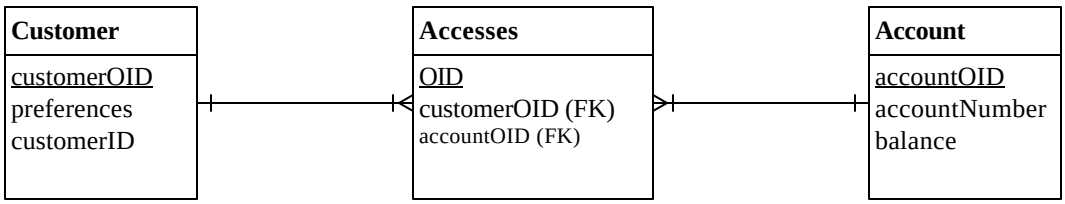


Figure 12. Implementing a many-to-many relationship in a relational database.

[3] All data models in this paper are shown using the Data Structure Diagram notation from the mid-1980s. All class models are shown using the Unified Modeling Language (UML). Please refer to my second book, *Building Object Applications That Work* (Ambler, 1998a), for an explanation of each notation.

3.3.4 The Same Classes/Tables, Different Relationships

Sometimes you can have a direct mapping between classes in your object application to tables in your relational database, yet the relationships between the classes/tables aren't the same. For example, Figure 13 shows an alternative implementation of the data model of Figure 8. In this case your organization has decided to purchase a list of all the zip codes in the countries that you do business in so that you can increase the quality of your address data. Unlike in the class diagram of Figure 7, which had a one-to-one relationship between the **Address** class and the **Zip Code** class, Figure 13 shows a many-to-one relationship. Assuming both models are correct, and for the purpose of this discussion they are, how can this be? The answer is that the needs are different.

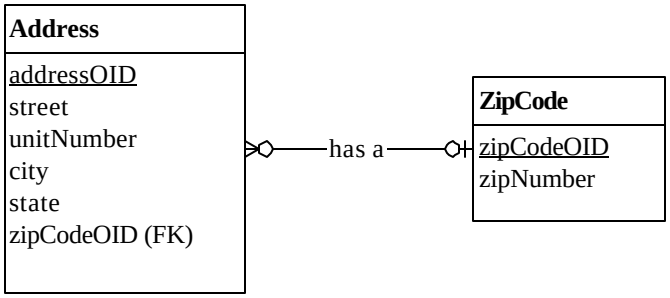


Figure 13. A data model showing an alternate implementation for an address.

The class diagram in Figure 7 shows that the OO application only needs a **Zip Code** class to encapsulate the validation rules of a zip code – is it in the right format, is it in the right state – it does not care about the fact that there are several addresses with the same zip code. There were no business requirements stating that a zip code has many addresses in it. The data model in Figure 13 shows that it is possible to have several addresses in the same zip code, which is an accurate way to model the data. Figure 13 also shows that sometimes you'll have an address with a zip code that isn't in the list that you bought (depending on the country you live in, even the post office does not have an accurate list of all zip codes). The zip code might be correct, it just isn't in your official list of zip codes. You might decide to have insertion triggers in your database to ensure that you don't insert an address without a proper zip code (perhaps if the zip code does not exist in the table you automatically add it to the list).

Scott's Soapbox – Data Models Don't Always Reflect Actual Requirements

The differences between the class model of Figure 7 and the data model of Figure 13 reveal a fundamental flaw in the approach taken by many data models. Don't get me wrong, the data model is in fact accurate, it just that it doesn't reflect the actual requirements of the organization that it was being modeled for. What is the impact of this mistake? First, the code written to support this data model, something that few data modelers take into consideration (very often they don't write any code at all) is much more complex than the code that would be written to support the class model. The association modeled in the data model is much more complex than that of the class model: it is bi-directional and optional. Second, there is now a difference (likely unbeknownst to the data modelers) between the software being built and the actual needs of the users of that software. Albeit this is a small difference, the example is very simple for the sake of discussion, but any difference will result in a greater training burden and cost of support for your organization.

Yes, you could rework your class diagram to take advantage of the fact that you now have this official list of zip codes, perhaps to do analysis of where your customers live, but until you have an actual requirement to do so you shouldn't – perhaps your users don't need such a thing.

4. Concurrency, Objects, and Row Locking

For this white paper concurrency deals with the issues involved with allowing multiple people simultaneous access to the same record in your relational database. Because it is possible, if you allow it, for several users to access the same database records, effectively the same objects, you need to determine a control strategy for allowing this. The control mechanism used by relational databases is locking, and in particular row locking.

4.1 Pessimistic Vs. Optimistic Locking

There are two main approaches to row locking: pessimistic and optimistic.

1. **Pessimistic locking**. An approach to concurrency in which an item is locked in the persistence mechanism for the entire time that it is in memory. For example, when a customer object is edited a lock is placed on the object in the persistence mechanism, the object is brought into memory and edited, and then eventually the object is written back to the persistence mechanism and the object is unlocked. This approach guarantees that an item won't be updated in the persistence mechanism while the item is in memory, but at the same time is disallows others to work with it while someone else does. Pessimistic locking is ideal for batch jobs that need to ensure consistency in the data that they write.

2. **Optimistic locking**. An approach to concurrency in which an item is locked in the persistence mechanism only for the time that it is accessed in the persistence mechanism. For example, if a customer object is edited a lock is placed on it in the persistence mechanism for the time that it takes to read it in memory and then it is immediately removed. The object is edited and then when it needs to be saved it is locked again, written out, then unlocked. This approach allows many people to work with an object simultaneously, but also presents the opportunity for people to overwrite the work of others. Optimistic locking is best for online processing.

Yes with optimistic locking you have an overhead of determining whether or not the record has been updated by someone else when you go to save it. This can be accomplished via the use of a common timestamp field in all tables: When you read a record you read in the timestamp. When you go to write the record you compare the timestamp in memory to the one in the database, if they're the same then you update the record (including the timestamp to the current time). If they're different then someone else has updated the record and you can't overwrite it (therefore displaying a message to the user).

5. Stored Procedures

A stored procedure is basically a function that runs on a relational database server. Although SQL code is usually a major component of a stored procedure most database vendors have their own proprietary programming language, each with its strengths and weaknesses. A stored procedure typically runs some SQL code, potentially massages the data, and then hands back a response in the form of zero or more records or as a database error message. Stored procedures are a very powerful feature of modern relational databases.

When mapping objects to relational databases there are two situations where using stored procedures make sense. First is when you're building a quick and dirty prototype that you intend to throw away, assuming that you don't have a solid persistence layer (Ambler, 1998c) already built, then this is most likely the quickest way to get your prototype up and running. The second situation is when you're mapping to a legacy database whose design is completely inappropriate for objects and you aren't able to rework it for

your specific needs. You can create stored procedures to read and write records that look like the objects that you want. Note that you don't need to write this code using stored procedures, instead you could do it in your language of choice and run it outside of your database (although perhaps still on your server machine to avoid unnecessary network traffic).

There are, however, several reasons why you don't want to use stored procedures when mapping objects to relational databases. First, the server can quickly become a bottleneck using this approach. You really need to have your act together when moving functionality onto your server – a simple stored procedure can bring the server to it knees if it is invoked often enough. Second, stored procedures are written in a proprietary language, and as anyone who has ever ported between database vendors, or even between database versions from the same vendor, this can be a show-stopper. The one thing that you can count on in this industry is change, and you can count on at least upgrading your database in time. Third, you dramatically increase the coupling within your database because stored procedures directly access tables, coupling the tables to the stored procedures. This increased coupling reduces the flexibility of your database administrators, when the want to reorganize the database they need to rewrite stored procedures, and increases the maintenance burden of developers because they have to deal with the stored procedure code.

The bottom line is that stored procedures are little better than a quick hack used to solve your short-term problems.

6. Triggers

A trigger is effectively a stored procedure that is automatically invoked for specific actions on a table. It is common to define insert triggers, update triggers, and deletion triggers for a table which will be invoked before an insertion, update, or deletion takes place on that table. The trigger must run successfully otherwise the action is aborted. Triggers are used to ensure referential integrity in your database.

Like stored procedures, triggers are written in a proprietary language for each database making them difficult to port between vendors and sometimes even versions of the same database. The good news is that many data modeling tools [4] will generate basic triggers for you based on the relationship information defined in your data model. As long as you don't modify the generated code, if you port between vendors/versions you can always regenerate your triggers from your data model.

[4] You can find these tools easily by doing a search on the Internet.

7. Process Patterns for Mapping Objects To RDBs

Figure 14 depicts the Persistence Modeling process pattern (Ambler, 1998b), in many ways a subset of the Detail Modeling process pattern (Ambler, 1998b) of **Figure 15**, which indicates the process for modeling the persistence aspects of your object-oriented application. In case process patterns are new to you, a process pattern describes an approach to developing software that is proven in practice to work effectively. The Persistence Modeling process pattern shows that your Object-Oriented Model, the key to which is your Class Model, should drive the development of your Logical Persistence Model which in turn drives the development of your Physical Persistence Model. The Persistence Modeling process pattern indicates that there exists two types of Persistence Model: a Logical Persistence Model and a Physical Persistence Model. A Logical Persistence Model is used to show what you want to build; in effect it is an analysis model. A Physical Persistence Model is used to show how you intend to build your persistence schema, in effect it is a design model. For the sake of our discussion, we're really talking about logical data models and physical data models.

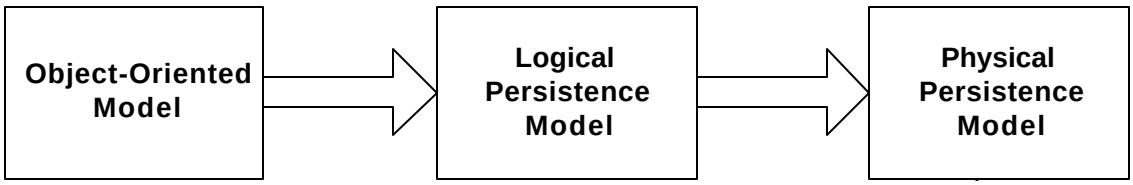


Figure 14. The Persistence Modeling process pattern.

Logical Persistence Models Offer Little Value

Although Figure 14 includes a logical persistence model, the reality is that most experienced object mappers go straight from their OO models to their physical persistence model. The extra information that logical persistence models contain such as domain values for attributes (something significantly more complex in the OO world considering many attributes are other objects) and candidate keys (a spectacularly bad idea as we saw previously) can actually be included in your class model if needed. On the other hand, the one advantage of modeling candidate keys is that they indicate potential ways that your users will access data -- important information for tuning your database. However, how your users will interact with your system should be reflected in your use cases so once again I'm not so sure we actually need to model candidate keys.

I typically suggest developing logical persistence models to organizations that are initially transitioning to object technology to support a comfort level that many experienced modelers can accept. This is the reason why I have included logical persistence modeling in the Persistence Modeling process pattern. What I've seen happen is that people, often the data modelers doing the work, quickly realize that logical persistence modeling is simply a waste of effort that can be cut out of the process.

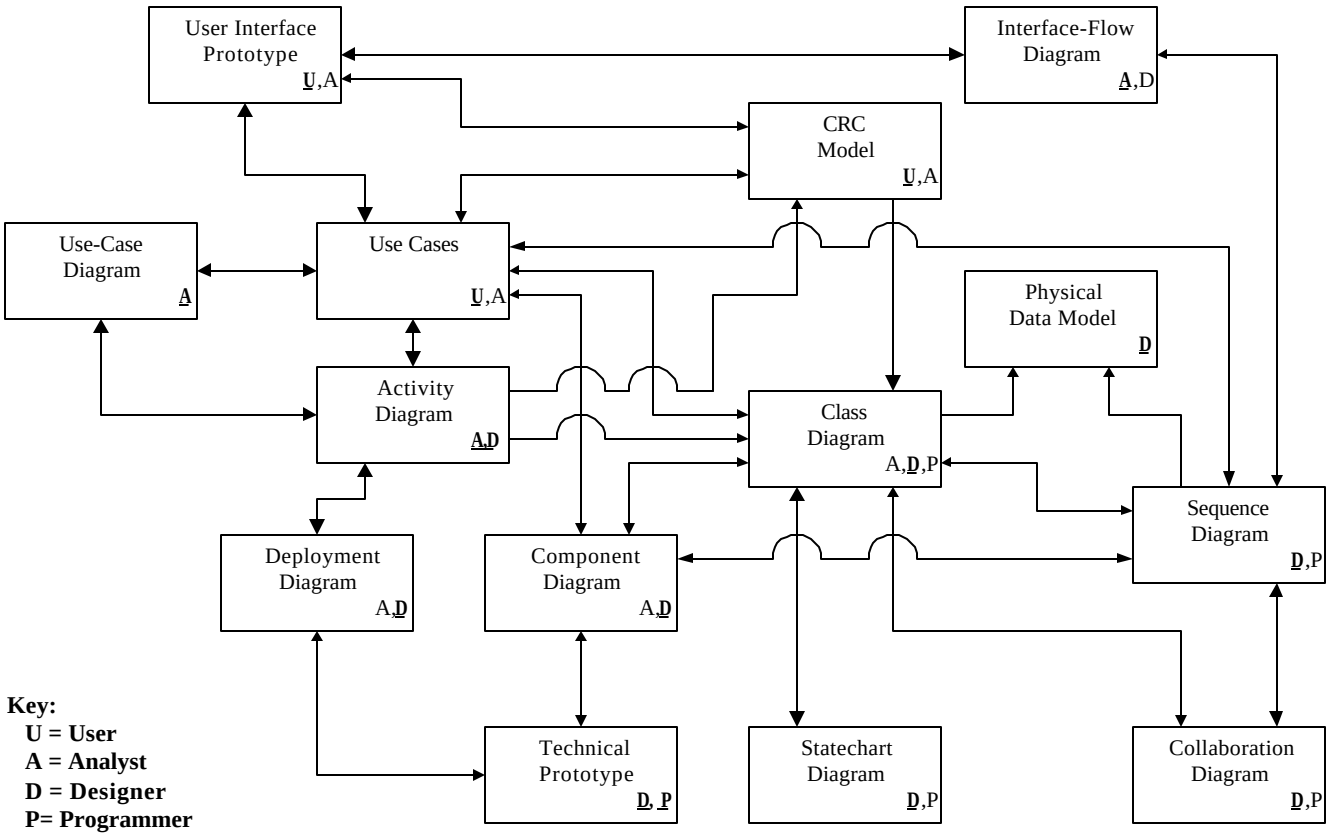


Figure 15. The Detailed Modeling Process pattern.

In Figure 15 the boxes represent the main techniques/diagrams of OO modeling and the arrows show the relationships between them, with the arrowheads indicating an "input into" relationship. For example, we see that a process model is an input into a class diagram. In the bottom right-hand corner of each box are letters which indicate who is typically involved in working on that technique/diagram. The key is straightforward: U=User, A=Analyst, D=Designer, and P=Programmer. The letter that is underlined indicates the group that performs the majority of the work for that diagram. For example, we see that users form the majority of the people involved in developing a CRC model and designers form the majority of those creating state diagrams.

An interesting feature of Figure 15 is that it illustrates that the object-oriented modeling process is both serial in the large and iterative in the small. The serial nature is exemplified when you look from the top-left corner to the bottom right corner: the techniques move from requirements gathering to analysis to design. You see the iterative nature of OO modeling from the fact that each technique drives, and is driven by, other techniques. In other words you iterate back and forth between models.

Figure 16 depicts an updated view of the solution to the Detailed Modeling process pattern as described in *The Object Primer 2nd Edition* (Ambler, 2001). The main difference is the inclusion of new artifacts (sorry about that, but development is hard after all) to reflect the real-world needs of complex application development.

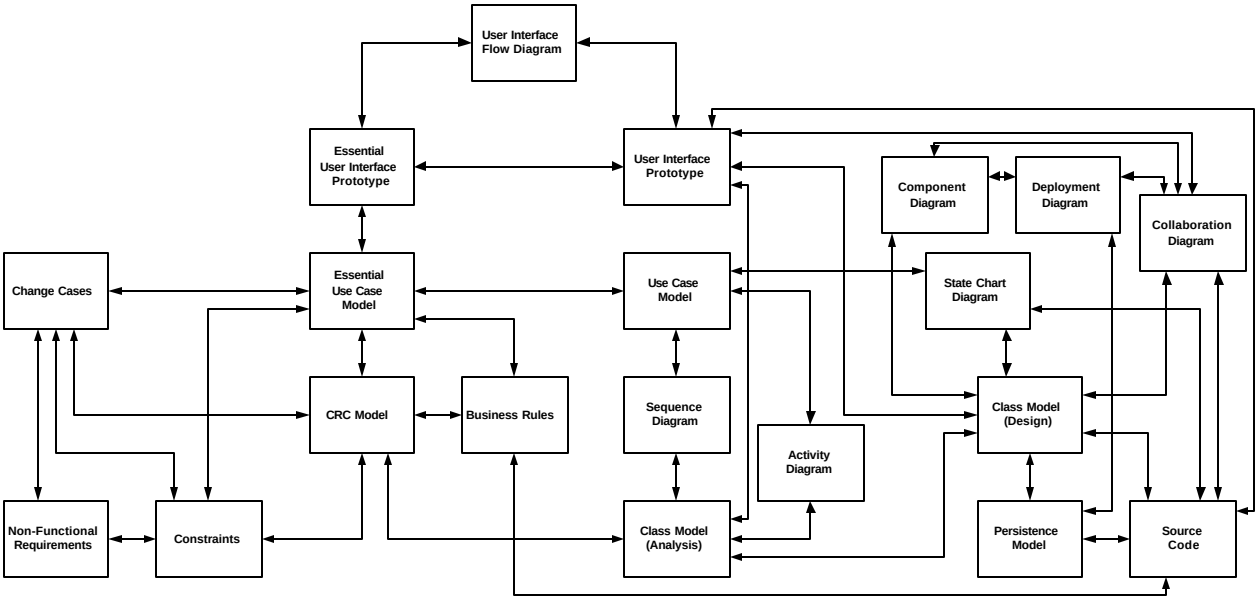


Figure 16. The artifacts of business software development.

Blatant Advertising – Purchase *The Object Primer, 2nd Edition*



The Object Primer 2nd Edition is a straightforward, easy to understand introduction to object-oriented concepts, requirements, analysis, and design techniques applying the techniques of the Unified Modeling Language (UML). The Object Primer goes further to show you how to move from object modeling to object-oriented programming, providing Java examples, and describes the techniques of the Full Lifecycle Object-Oriented Testing (FLOOT) methodology to enable you to test all of your development artifacts. It is the first book that describes how to develop a system using object-oriented and relational database technology end-to-end, from requirements all the way to Java code and a RDB design. It also puts this material in the context of the leading software processes, including the enhanced lifecycle for the Unified Process, the process patterns of the Object-Oriented Software Process (OOSP), and the best practices Extreme Programming (XP). Visit www.ambysoft.com/theObjectPrimer.html for more details.

7.1 Why is The Persistence Modeling Process Pattern Important?

A penny saved is a penny persisted. 😊

The answer to this question is simple: it provides a viable strategy for effectively modeling the persistence needs of your OO applications. Data models only take into account half of the picture (data) whereas object-oriented models take into account the entire picture (data and behavior). By using your OO models to drive the development of your data models you ensure that your database schema will actually support the needs of your application – after all, your OO application is built based on your OO models. This approach should make sense, by building both your application and your persistence mechanism from the same model you improve the chance that they will work together effectively.

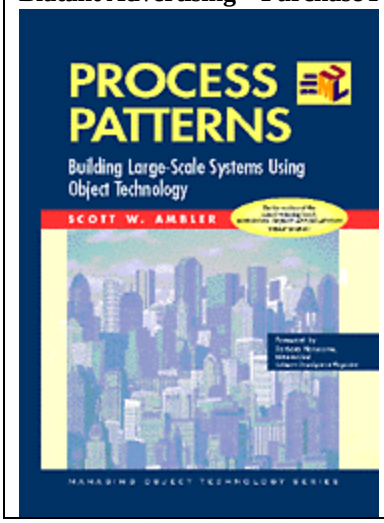
Using data models to drive the development of object-oriented models is a common process anti-pattern that many organizations follow. A process anti-pattern, as the name suggests, is an approach to developing software that is proven in practice to not be very effective. Many organizations have used their existing data models as input into the development of their class models, only to find later that resulting model results in a clumsy implementation of what needs to be built. Practice shows that data models are an insufficient basis from which to create a class model for the following reasons:

1. **Data models are too narrowly focused.** Few data models take behavior and object-oriented concepts such as inheritance into account in their design. In short, data models focus on a small portion of the overall picture, that of data.
2. **Many precepts of relational theory have proven disastrous in practice.** Data models often include many assumptions derived from classic relational theory (keys can have business meaning, composite keys are good idea, and centralized databases make sense) that prove to be undesirable from an object-oriented and/or software engineering point of view.
3. **Data models are rarely based on proven patterns common to the object world.** Data models will not take key object-oriented techniques such as design patterns (Gamma et. al. 1995) and analysis patterns (Fowler, 1997; Ambler, 1998a) into account. As anyone who has worked with analysis and design patterns knows, they provide fantastic opportunities for improving the extensibility and maintainability of your work.
4. **Data models may not reflect actual requirements.** Data models are often developed without taking the actual user requirements for the software into account, instead they are developed by focusing solely on the data.
5. **Your requirements have likely changed since the data model was developed.** Even if your legacy data model was developed based on the actual requirements, the requirements have likely changed anyway implying that your data model isn't accurate anyway.
6. **Data modeling naming conventions rarely make sense for OO development.** Naming conventions that made perfect sense in the data-modeling world are often inappropriate for the object-modeling world. Due to the high levels of consistency between and common paradigm supported by the models of the OO world (Ambler, 1998a; Ambler, 1998b) that is not exhibited by the models of the structured world naming conventions for classes and their attributes are applied to a wide range of models – class diagrams, sequence diagrams, collaboration diagrams, activity diagrams, and source code to name a few – instead of a single data model. The point to be made is that you need to take a wide range of issues into account when setting naming conventions, issues that would not have been considered when setting naming conventions for your data models. Remember, data models only take a very small portion of the overall design picture into account.
7. **There is more to OO modeling than class diagrams.** Even if you could magically derive a class model from a data model, you'd still need to do all the requirements gathering, analysis, and design work to develop these other models.

Following the Persistence Modeling process pattern you expend the majority of your modeling efforts getting your class model right – including appropriate normalization/denormalization issues, a topic that I cover in detail in *Building Object Applications That Work* (Ambler, 1998a). Once you are satisfied that your

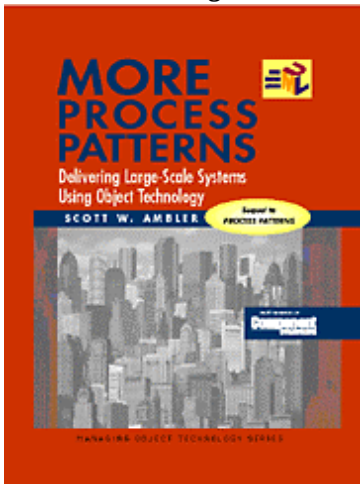
class model is stable, you generate your logical data model from it (many CASE tools support this with a press of a button) and then create your physical data model from there. It would be a serious mistake to assume that you are now simply in the old data-modeling world where all the logical to physical data modeling rules still apply. Yes, you need to take access and database performance considerations into account when developing the physical data model, but you also need to consider the fact any changes that you make in your physical data model also has performance impacts in your application. The more that the data model deviates from your class model, the greater the performance impact of your mapping. This poses an interesting problem: do you leave your class model unchanged (after all, it models what you actually need to build) and accept the performance hit from the greater mapping complexity, do you modify your class model to reduce the performance hit (but then bastardize your application as a result), or do you accept the performance hit in your database?

Blatant Advertising – Purchase *Process Patterns* today!



This book presents a collection of process patterns for successfully initiating a software project and taking it through the construction phase. It provides a wealth of advice for engineering requirements, modeling, programming, and testing. It puts these topics in the context of a proven software process for the development of large-scale, mission-critical software, covering topics that you typically don't find in other books about object-oriented development such as project management, quality assurance, risk management, and deliverables management. Object-oriented development is hard, particularly if you are building systems using n-tier technology such as Enterprise JavaBeans (EJB) or even the "simple" Java platform, and you need to understand the big picture to be successful. *Process Patterns*, and its sister book, *More Process Patterns*, give you that big picture. For more information, and to order online, visit www.ambysoft.com/processPatterns.html

Blatant Advertising – Purchase *More Process Patterns* today!



This book presents a collection of process patterns for successfully delivering a software project and then operating and supporting it once it is in production. It provides a wealth of advice for testing your object-oriented application, for reworking it, for preparing to transition it to your user community, and for supporting it once it is in production. It puts these topics in the context of a proven software process for the development of large-scale, mission-critical software, covering topics that you typically don't find in other books about object-oriented development such as project management, quality assurance, risk management, and deliverables management. Object-oriented development is hard, particularly if you are building systems using n-tier technology such as Enterprise JavaBeans (EJB) or even the "simple" Java platform, and you need to understand the big picture to be successful. *More Process Patterns*, and its sister book, *Process Patterns*, give you that big picture. For more information, and to order online, visit www.ambysoft.com/moreProcessPatterns.html

7.2 The Implications?

So what are the implications to you and your organization? First and foremost, you will need to train many of your existing data modelers in object modeling techniques. Yes, you will still need data modelers to aid in the development of a data model based on the needs of your OO models, but it is very obvious that you will need significantly fewer data modelers than you need now. Second, your data modeling community needs to come to grips with the fact that their role in software development is significantly reduced. In the past the data model was equal in importance to your process model, in the object world it plays a secondary role (at best) to your class model. It is a difficult fact for many data modelers to accept, but once they do they realize that the object-modeling world offers many opportunities for them. Third, many organizations will need to accept that their existing enterprise data model, although important in the past, is now little more than a source of entertainment for their OO modelers.

The majority of your organization's data modelers will need to be retrained in OO modeling techniques.

Your enterprise data model is little more than wall paper in the OO world.

Scott's Career Advice for Data Modelers

Logical data modelers. The reality is that there is no longer any need for logical data models, other than perhaps for interim hand-holding of people who haven't come up to speed on how to develop software following the OO paradigm. The implication is that logical data modelers either need to transition themselves into object modeling and/or physical data modeling or find employment elsewhere. The good news is that many of the core values of logical data modelers – to model, to take an enterprise view, to follow guidelines and conventions – are incredibly valuable. The bad news is that their chosen modeling mechanism, data models, are not sufficient and have been superceded by object-oriented modelers. I realize that this advice is spectacularly difficult for you to accept, but the sooner that you do the better you will be for it.

Physical data modelers. Because there is still a need for physical data models, regardless of what many self-proclaimed OO gurus will tell you, physical data modelers are still needed. You will need to learn how to map objects to RDBs and the increased performance complexities associated with doing so. The good news is that there is a very strong job market for people who understand how to do this.

Blatant Advertising – Hire Ronin International Consultants today!



Ronin International is a software consulting company that specializes in providing high-end object development services such as Enterprise JavaBeans (EJB) development, software process mentoring and tailoring, software architecture consulting (including reviews), and transitioning existing staff to object technologies and techniques. I am President of the company and work with clients around the world as a senior consultant. Visit www.ronin-intl.com for details.

8. The Realities of Mapping Objects To Relational Databases

This section describes and expands on several mapping issues that are described in the October 1997 issue of Software Development (Ambler, 1997e). The issues are:

- Objects and Relational Databases Are the Norm
- ODBC and JDBC Classes Aren't Enough
- You Need a Persistence Layer
- Hard-Coded SQL is an Incredibly Bad Idea
- You Have to Map to Legacy Data
- The Data Model Doesn't Drive Your Class Diagram
- Joins are Slow
- Keys With Business Meaning Are a Bad Idea
- Composite Keys Are a Bad Idea
- You Need Several Inheritance Strategies
- Stored Procedures Are a Bad Idea

8.1 Objects and Relational Databases Are the Norm

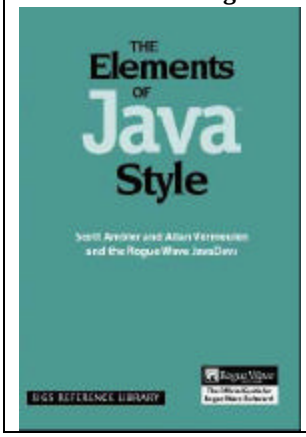
For years object gurus claimed that you shouldn't use relational databases to store objects because of the "object/relational impedance mismatch." Yes, the object paradigm is different from the relational paradigm, but for 99% of you the reality is that your development environment is object oriented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

8.2 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 (Ambler, 1997a). The bottom line is that the database access classes provided with your development environment are only a start, and a minimal one at that.

Blatant Advertising – Purchase *The Elements of Java Style* today!



This book (Vermeulen et. al., 2000) presents a collection of strategies for writing superior Java source code. This book presents a wider range of guidelines than what is presented here in this paper, and more importantly presents excellent source code examples. It covers many topics that are not covered in this paper, such as type safety issues, exception handling, assertions, and concurrency issues such as synchronization. This paper was combined with Rogue Wave's internal coding standards and then together were evolved to become *The Elements of Java Style*, so you should find the book to be an excellent next step in your Java learning process. Visit http://www.ambysoft.com/elementsJavaStyle.html for more details.

8.3 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.

Read the related whitepaper:

The Design of a Robust Persistence Layer for RDB

http://www.ambysoft.com/persistenceLayer.html

8.4 Hard-Coded SQL is an Incredibly Bad Idea

Save the whale objects. ☺

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.

8.5 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. Earlier I discussed the issues involved with loading data from legacy source data into an object-oriented application.

8.6 …But The Data Model Doesn't Drive Your Class Diagram

Just because you need to map to legacy data it does not 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.

For a better understanding of the object-oriented modeling process, I invite you to read my second and third books, *Building Object Applications That Work* (Ambler, 1998a) and *Process Patterns* (Ambler, 1998b), some of the few OO development books that actually shows how data modeling fits into the OO construction process.

8.7 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 is to traverse instead of join where it makes sense. Try it, it works really well.

8.8 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 does not make sense for a technical concept, a unique key, to be dependent on business rules.

The interesting thing is that data modelers call keys with business meaning 'smart keys,' yet experience has shown that a more appropriate term would have been 'incredibly stupid keys.' Goes to show what data modelers know!

8.9 …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 experiences 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).

8.10 You Need Several Inheritance Strategies

There are three fundamental solutions (Ambler, 1995b) 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.

8.11 Stored Procedures Are a Bad Idea

This issue was discussed previously, but the main conclusion was that stored procedures are little better than a quick hack used to solve your short-term problems.

9. So What's With The Attitude Problem?

In this section I want to address why we keep hearing that mapping does not 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.

10. 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.

Scott's Recommended Reading's

I maintain a persistence reading list at http://www.ambysoft.com/booksPersistence.html that you should find of value. There are also several other reading lists at http://www.ambysoft.com/books.html, including ones about object-orientation, patterns, component-based development, and Java.

11. References and Recommended Reading

Ambler, S.W. 1995a. *Complex Data Relationships: Bet on OODBMS*. Software Magazine, January 1995, p72-74.

Ambler, S.W. 1995b. *Mapping Objects to Relational Databases*. Software Development, October 1995, p63-72.

Ambler, S.W. 1996. *Object-Relational Mapping*. Software Development, October 1996, p47-50.

Ambler, S.W. 1997a. *Designing a Search Screen*. Software Development, January 1997, p79-82.

Ambler, S.W. 1997b. *Handling Object-Oriented Errors*. Software Development, February 1997, p71-73.

Ambler, S.W. 1997c. *Normalizing Classes*. Software Development, April 1997, p69-72.

Ambler, S.W. 1997d. *Implementing PickLists of Objects*. Software Development, June 1997, p73-76.

Ambler, S.W. 1997e. *The Realities of Mapping Objects To Relational Databases*. Software Development, October 1997, p71-74.

Ambler, S.W. 1998a. *Building Object Applications That Work: Your Step-By-Step Handbook for Developing Robust Systems with Object Technology*. SIGS Books/Cambridge University Press, 1998. http://www.ambysoft.com/buildingObjectApplications.html

Ambler, S.W. 1998b. *Process Patterns: Building Large-Scale Systems Using Object Technology*. New York: SIGS Books/Cambridge University Press 1998. http://www.ambysoft.com/processPatterns.html

Ambler, S.W. (1998c). *The Design of a Robust Persistence Layer For Relational Databases: An AmbySoft Inc. White Paper*. http://www.ambysoft.com/persistenceLayer.html.

Ambler, S.W. (2001). *The Object Primer 2nd Edition: The Application Developer's Guide to Object Orientation*. New York: Cambridge University Press. http://www.ambysoft.com/theObjectPrimer.html.

Ambler, S.W. & Constantine, L.L. (2000a). *The Unified Process Inception Phase*. Gilroy, CA: CMP Books. http://www.ambysoft.com/inceptionPhase.html.

Ambler, S.W. & Constantine, L.L. (2000b). *The Unified Process Elaboration Phase*. Gilroy, CA: CMP Books. http://www.ambysoft.com/elaborationPhase.html.

Ambler, S.W. & Constantine, L.L. (2000c). *The Unified Process Construction Phase*. Gilroy, CA: CMP Books. http://www.ambysoft.com/constructionPhase.html.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *A Systems of Patterns: Pattern-Oriented Software Architecture*. New York: John Wiley & Sons Ltd.

Rational (1997). *The Unified Modeling Language for Object-Oriented Development Documentation v1.1*. Rational Software Corporation, Monterey California.

Vermeulen, A., Ambler, S.W., Bumgardner, G., Metz, E., Misfeldt, T., Shur, J., & Thompson, P. (2000). *The Elements of Java Style*. New York: Cambridge University Press.

Important Note: The Software Development articles cannot be obtained online from Software Development nor from me. If you want to obtain copies you will have to get them from your local library or from a friend. Sorry.

12. About the Author

Scott W. Ambler is President of Denver-based Ronin International. Scott is the author of *The Object Primer 2nd Edition* (1995, 2000), *Building Object Applications That Work* (1998), *Process Patterns* (1998) and *More Process Patterns* (1999), and co-author of *The Elements of Java Style* (2000) all published by Cambridge University Press. He is also co-editor with Larry Constantine of the Unified Process book series from CMP books, including *The Unified Process Inception Phase* (Fall 2000), *The Unified Process Elaboration Phase* (Spring 2000), and *The Unified Process Construction Phase* (Summer 2000) all of which focus on best practices to enhance the Unified Process. He has worked with OO technology since 1990 in various roles: Process Mentor, Business Architect, System Analyst, System Designer, Project Manager, 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 is a contributing editor with *Software Development* (www.sdmagazine.com) and writes columns for *Computing Canada* (www.plesman.com). He can be reached via e-mail at scott.ambler@ronin-intl.com.

13. Glossary of Terms

This section describes the key terms that I have used throughout this document.

Aggregation -- Represents "is-part-of" relationships.

Anti-pattern – The description of a common approach to solving a common problem, an approach that in time proves to be wrong or highly ineffective.

Application server – A server on which business logic is deployed. Application servers are key to an n-tier client/server architecture.

Association -- Relationships, associations, exist between objects. For example, customers BUY products.

Associative table – A table in a relational database that is used to maintain a relationship between two or more other tables. Associative tables are typically used to resolve many-to-many relationships.

Client – A single-user PC or workstation that provides presentation services and appropriate computing, connectivity, and interfaces relevant to the business need. A client is also commonly referred to as a "front-end."

Client/server (C/S) architecture – A computing environment that satisfies the business need by appropriately allocating the application processing between the client and the server processes.

Concurrency – The issues involved with allowing multiple people simultaneous access to your persistence mechanism.

Coupling – A measure of how connected two items are.

CRUD – Acronym for create, retrieve, update, delete. The basic functionality that a persistence mechanism must support.

Data dictionary – A repository of information about the layout of a database, the layout of a flat file, the layout of a class, and any mappings among the three.

Database proxies – An object that represents a business object stored in a database. To every other object in the system the database proxy appears to be the object that it represents. When other objects send the proxy a message it immediately fetches the object from the database and replaces itself with the fetched object, passing the message onto it. See the Proxy pattern in chapter 4 for more details.

Database server – A server which has a database installed on it.

Distributed objects – An object-oriented architecture in which objects running in separate memory spaces (i.e. different computers) interact with one another transparently.

Domain/business classes – Domain/business classes model the business domain. Business classes are usually found during analysis, examples of which include the classes **Customer** and **Account**.

Fat-client – A two-tiered C/S architecture in which client machines implement both the user interface and the business logic of an application. Servers typically only supply data to client machines with little or no processing done to it.

Key – One or more columns in a relational data table that when combined form a unique identifier for each record in the table.

Lock – An indication that a table, record, class, object, ... is reserved so that work can be accomplished on the item being locked. A lock is established, the work is done, and the lock is removed.

n-Tier client/server – A client/server architecture in which client machines interact with application servers, which in turn interact with other application servers and/or database servers.

Object adapter – A mechanism that both converts objects to records that can be written to a persistence mechanism and converts records back into objects again. Object adapters can also be used to convert between objects and flat-file records.

Object identifiers (OIDs) – A unique identifier assigned to objects, typically a large integer number. OIDs are the object-oriented equivalent of keys in the relational world.

ODMG – Object Database Management Group, a consortium of most of the ODBMS vendors who together set standards for object databases.

OOCRUD – Object-oriented CRUD.

Optimistic locking – An approach to concurrency in which an item is locked only for the time that it is accessed in the persistence mechanism. For example, if a customer object is edited a lock is placed on it in the persistence mechanism for the time that it takes to read it in memory and then it is immediately removed. The object is edited and then when it needs to be saved it is locked again, written out, then unlocked. This approach allows many people to work with an object simultaneously, but also presents the opportunity for people to overwrite the work of others.

OQL – Object Query Languages, a standard proposed by the ODMG for the selection of objects. This is basically SQL with object-oriented extensions that provide the ability to work with classes and objects instead of tables and records.

Pattern – The description of a general solution to a common problem or issue from which a detailed solution to a specific problem may be determined. Software development patterns come in many flavors, including but not limited to analysis patterns, design patterns, and process patterns.

Persistence – The issue of how to store objects to permanent storage. Objects need to be persistent if they are to be available to you and/or to others the next time your application is run.

Persistence classes – Persistence classes provide the ability to permanently store objects. By encapsulating the storage and retrieval of objects via persistence classes you are able to use various storage technologies interchangeably without affecting your applications.

Persistence layer – The collection of classes that provide business objects the ability to be persistent. The persistence layer effectively wraps your persistence mechanism.

Persistence mechanism – The permanent storage facility used to make objects persistent. Examples include relational databases, object databases, flat files, and object/relational databases.

Pessimistic locking – An approach to concurrency in which an item is locked for the entire time that it is in memory. For example, when a customer object is edited a lock is placed on the object in the persistence mechanism, the object is brought into memory and edited, and then eventually the object is written back to the persistence mechanism and the object is unlocked. This approach guarantees that an item won't be

updated in the persistence mechanism whereas the item is in memory, but at the same time is disallows others to work with it while someone else does.

Process anti-pattern – An anti-pattern which describes an approach and/or series of actions for developing software that is proven to be ineffective and often detrimental to your organization.

Process pattern – A pattern which describes a proven, successful approach and/or series of actions for developing software.

Read lock – A type of lock indicating that a table, record, class, object,... is currently being read by someone else. Other people may also obtain read locks on the item, but no one may obtain a write lock until all read locks are cleared.

Reading into memory – When you obtain an object from the persistence mechanism but don't intend to update it.

Retrieving into memory – When you obtain an object from the persistence mechanism and will potentially update it.

Server – A server is one or more multiuser processors with shared memory that provides computing connectivity, database services, and interfaces relevant to the business need. A server is also commonly referred to as a "back-end."

SQL – Structured Query Language, a standard mechanism used to CRUD records in a relational database.

SQL statement – A piece of SQL code.

System layer – The collection of classes that provide operating-system-specific functionality for your applications, or that wrap functionality provided by non-OO applications, hardware devices, and/or non-OO code libraries.

Thin client – A two-tiered client/server architecture in which client machines implement only the user interface of an application.

Transaction – A transaction is a single unit of work performed in a persistence mechanism. A transaction may be one or more updates to a persistence mechanism, one or more reads, one or more deletes, or any combination thereof.

User-interface classes – User-interface classes provide the ability for users to interact with the system. User interface classes typically define a graphical user interface for an application, although other interface styles, such as voice command or handwritten input, are also implemented via user-interface classes.

Wrapping – Wrapping is the act of encapsulating non-OO functionality within a class making it look and feel like any other object within the system.

Write lock – A type of lock indicating that a table, record, class, object,... is currently being written to by someone else. No one may obtain either a read or a write lock until this lock is cleared.

14. Index

A

Ad-hoc reporting 8
Aggregation
  definition 31
  vs. association 12
Analysis patterns 7
Anti-pattern 31
Application server 31
Architecture 7
Assertions 25
Association
  vs. aggregation 12
Associative table 14
  definition 31
Attribute
  mapping 7
Author
  contacting 30

B

Book
  Building Object Applications That Work 7
  Elements of Java Style 25
  More Process Patterns 22
  Process Patterns 22
  The Object Primer 20
Business meaning 2, 27

C

C++ 28
Career advice 23
Client 31
Client/server (C/S) architecture 31
Composite keys 27
Concurrency 16, 25
  definition 31
Coupling 31
CRUD
  definition 31

D

Data dictionary
  definition 31
Data model 26
Data modelers
  career advice 23
Database administrators 25
Database proxy
  definition 31
Database server 31
Databases 7
Design patterns 7
Detailed modeling
  techniques 19
Distributed design 7
Distributed objects 31
Domain/business class 31

E

Enterprise data models 23
Example source code 25
Exception handling 25
Extreme Programming 20

F

Fat client 32
FLOOT 20
Foreign key 13
Full Lifecycle Object-Oriented Testing 20

G

GUIDs 3

I

Impedance mismatch ..... i, 24
Inheritance 27
  and relational databases 8
Instance relationship
  definition 31
Iterative in the small 19

J

JDBC 24
Joins 26

K

Key
  definition 32
  foriegn keys 13
Keys
  primary 2
Key-values table 3

L

Legacy databases 26
Lock

definition 32
optimistic locking 32
pessimistic locking 33
read lock 33
write lock 34
Locking 16
optimistic 16
pessimistic 16
Logical data modelers
career advice 23
Logical persistence model 18

M

Many-to-one relationship
implementation 14
Mapping
attributes 7
basics 7
different relationships 15
dissimilar classes to one table 11
inheritance 8
relationships 12
MAX() 3
Metrics 7

N

Naming conventions 21
n-Tier client/server 32

O

Object adapter
definition 32
Object database management group
definition 32
Object databases 7, 28
Object ID ..... i, 28
advantages i
and business meaning 2
assigning 2
vs. composite keys 27
Object identifier i
definition 32
Object paradigm i
Object query language
definition 32
Object-Oriented Software Process 20
ODBC 24
OID ..... *See* Object ID
One-to-many ..... *See* Many-to-one
One-to-one relationship
implementation 14
OOCRUD
definition 32
OOSP 20
Optimistic locking 16
definition 32

P

Pattern 32
Persistence
definition 32
Persistence class 32
Persistence layer 25
definition 32
white paper 25
Persistence mechanism 32
Pessimistic locking 16
definition 33
Physical data modelers
career advice 23
Physical persistence model 18
Polymorphism 2
Process anti-pattern 33
Process pattern 33
persistence modeling 18

R

Read lock
definition 33
Reading list 28
Referential integrity 17
Relational database
and inheritance 8
future of 28
Relational databases 7
Relational paradigm i
Relationships 2
Replication 6
Ronin International 23

S

Serial in the large 19
Server 33
Smart keys 27
SQL 25
definition 33
Stored procedures 16, 27
advantages 17
disadvantages 17
Surrogate keys 2
Synchronization 25
System layer 33

T

Task process pattern
persistence modeling 18

Thin client 33
Transaction
definition 33
Triggers 17
Type safety 25

U

Unified Modeling Language 20
Unified Process 20
User interface class 33
UUIDs 3

W

Wrapping 33
Write lock
definition 34

X

XP 20