Mapping objects to relational databases

What you need to know and why

Level: Introductory

Scott W. Ambler (scott_ambler@ca.ibm.com), Practice Leader, Agile Development, Rational Methods Group, IBM 01 Jul 2000

Why is mapping objects to relational databases an issue for modern developers? For one thing, object technology, such as Java technology, is the most common environment applied for the development of new software systems. Also, relational databases are still the preferred approach for storage of persistent information and are likely to remain so for quite some time. Read on to see how you'll put this skill to use.

Why a paper about mapping objects to relational databases? Because of the "impedance mismatch" between the object paradigm and the relational paradigm. The object paradigm is based on software engineering principles such as coupling, cohesion, and encapsulation, whereas the relational paradigm is based on mathematical principles, particularly those of set theory. The two different theoretical foundations lead to different strengths and weaknesses. Furthermore, the object paradigm is focused on building applications out of objects that have both data and behavior, whereas the relational paradigm is focused 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 less-than-ideal combination of the two paradigms, but then, a few hitches are to be expected. One of the secrets of success for mapping objects to relational databases is to understand both paradigms and their differences, and then make intelligent trade-offs based on that knowledge.

This paper should alleviate several common misconceptions prevalent in development circles today, presenting a practical look at the issues involved with mapping objects to relational databases. The strategies are based on my development experiences, ranging from small to large projects in the financial, distribution, military, telecommunications, and outsourcing industries. I've applied these principles for applications written in C++, Smalltalk, Visual Basic, and the Java language.

How to map objects to relational databases

In this section I will describe the fundamental techniques required to successfully map objects into relational databases:

1. Mapping attributes to columns
2. Implementing inheritance in a relational database
3. Mapping classes to tables
4. Mapping associations, aggregation, and composition
5. Implementing relationships in relational databases

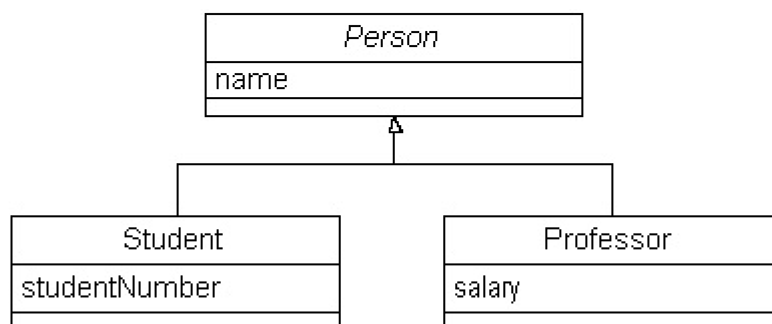
1. Mapping attributes to columns

A class attribute will map to either zero or a number of columns in a relational database. It is important to remember that not all attributes are persistent. For example, an `Invoice` class may have a `grandTotal` attribute that is used by its instances for calculation purposes, but that is not saved to the database. Furthermore, some object attributes are objects in their own right; for example, a `Course` object has an instance of `TextBook` as an attribute, which maps to several columns in the database (actually, chances are that the `TextBook` 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. It is also possible that several attributes could map to a single column in a table. For example, a class representing a U.S. zip code may have three numeric attributes, one representing each of the sections in a full zip code, whereas the zip code may be stored as a single column in an address table.

2. Implementing inheritance in a relational database

The concept of inheritance throws in several interesting twists when saving objects into a relational database. (See "Building Object Applications That Work" in Resources.) The issue basically boils down to figuring out how to organize the inherited attributes within your persistence model. The way in which you resolve this challenge can have a major impact on your system design. There are three fundamental solutions for mapping inheritance into a relational database, and to understand them I will discuss the trade-offs of mapping the class diagram presented in Figure 1. To keep the issues simple, I have not modeled all of the attributes of the classes; nor have I modeled their full signatures or any of the methods of the classes.

Figure 1. A UML class diagram of a simple class hierarchy



3. Mapping classes to tables

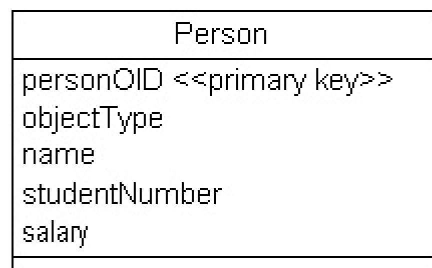
Classes map to tables, although often not directly. Except with very simple databases, you will never have a one-to-one mapping of classes to tables. In the following sections I will discuss three strategies for implementing inheritance structures to a relational database:

3.1. Using one data entity for an entire class hierarchy
3.2. Using one data entity per concrete class
3.3. Using one data entity per class

3.1. Using one data entity for an entire class hierarchy

With this approach, you map an entire class hierarchy into one data entity, where all the attributes of all the classes in the hierarchy are stored. Figure 2 depicts the persistence model for the class hierarchy of Figure 1 when this approach is taken. Notice that a `personOID` column was introduced for the primary key of the table, I will use OIDs (identifiers with no business meaning, also known as surrogate keys) in all of the solutions, just to be consistent and to take the best approach that I know of for assigning keys to data entities.

Figure 2. Mapping the class hierarchy to one single data entity



The advantages of this approach are that it is simple, that polymorphism is supported when a person changes roles, and that ad hoc reporting (reporting performed for the specific purposes of a small group of users, who commonly write the reports themselves) is also very easy with this approach because all of the personal data you need is found in one table. The disadvantages are that every time a new attribute is added anywhere in the class hierarchy a new attribute must be added to the table. This increases the coupling within the class hierarchy -- if a mistake is made while adding a single attribute, it could affect all the classes within the hierarchy in addition to the subclasses of whatever class got the new attribute. It also potentially wastes a lot of space in the database. I also had to add the

`objectType` column to indicate whether the row represents a student, a professor, or another type of person. This works well when someone has a single role, but quickly breaks down if they have multiple roles (for example, the person is both a student and a professor).

3.2. Using one data entity per concrete class

With this approach each data entity includes both the attributes and the inherited attributes of the class that it represents. Figure 3 depicts the persistence model for the class hierarchy of Figure 1 when this approach is taken. There are data entities corresponding to both the `Student` class and the `Professor` class because they are concrete, but not the `Person` class because it is abstract (indicated by the fact that its name is depicted in italics). Each of the data entities was assigned its own primary key, `studentOID` and `professorOID` respectively.

Figure 3. Mapping each concrete class to a single data entity

Student	Professor
studentOID <<primary key>> name studentNumber	professorOID <<primary key>> name salary

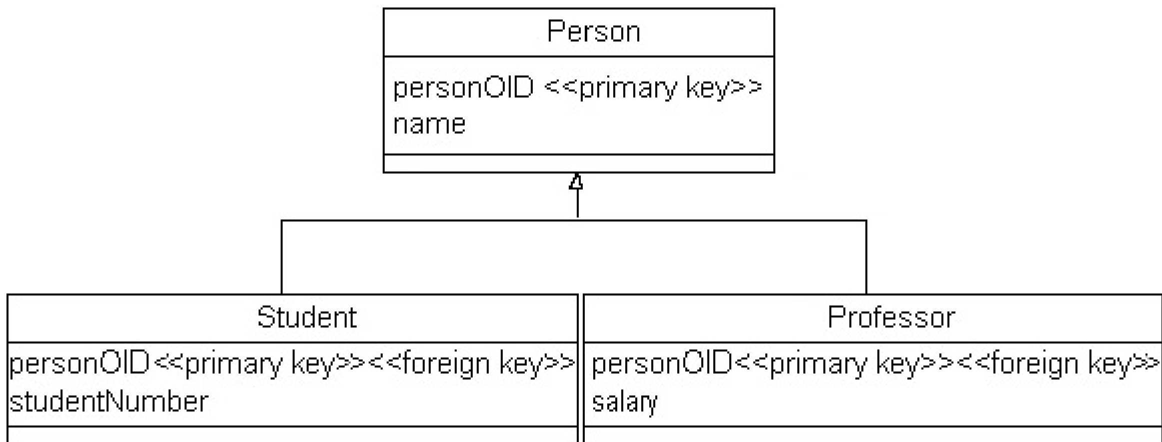
The biggest advantage to this approach is that it is still fairly easy to perform ad hoc reporting, given that all the data you need about a single class is stored in only one table. There are several disadvantages, however. One is that when you modify a class you have to modify its table and the table of any of its subclasses. For example, if you were to add height and weight to the `Person` class, you would need to update both tables, which is a lot of work. Second, whenever an object changes its role -- perhaps you hire one of your graduating students to become a professor -- you 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 is possible; just harder than it should be.) For instance, where would you store the name of someone who is both a student and a professor?

3.3. Using one data entity per class

With this approach you create one table per class, the attributes of which are the OID and the attributes that are specific to that class. Figure 4 depicts the persistence model for the class hierarchy of Figure 1 when this approach is taken. Notice that `personOID` is used as the primary key for all three data entities. An interesting feature of Figure 4 is that the `personOID` column in both `Professor` and `Student` is assigned two stereotypes, something that is not allowed in the Unified Modeling Language (UML). My opinion is that this is an issue that will have to be addressed by the UML persistence modeling profile and may even necessitate a change in this

modeling rule. (See "Towards a UML Profile for a Relational Persistence Model" in Resources for more information about persistence models.)

Figure 4. Mapping each class to its own data entity



The main advantage of this approach is that it conforms best to object-oriented concepts. 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 because you merely need to modify or add one table. There are several disadvantages to this approach. First, there are many tables in the database -- one for every class, in fact (plus tables to maintain relationships). Second, it takes longer to read and write data using this technique because you have 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 each of the disk-drive heads operate independently). Third, ad hoc reporting on your database is difficult, unless you add views to simulate the desired tables.

Comparing the mapping strategies

Notice, now, how each mapping strategy results in a different model. To understand the design trade-offs between the three strategies, consider the simple change to our class hierarchy presented in Figure 5: a `TenuredProfessor` class has been added, which inherits from `Professor`.

Figure 5. Extending the initial class hierarchy

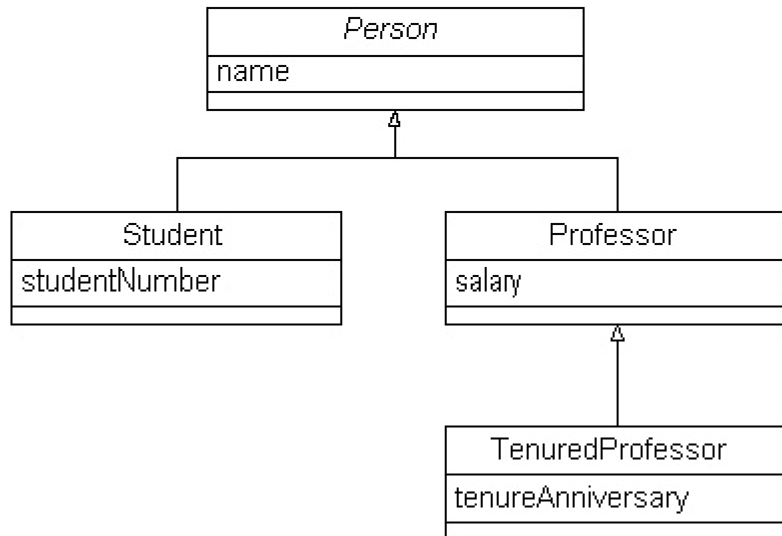


Figure 6 presents the updated persistence model for mapping the entire class hierarchy into one data entity. Notice how very little effort was required to update the model following this strategy, although the obvious problem of wasted space in the database has increased.

Figure 6. Mapping the extended hierarchy to a single data entity

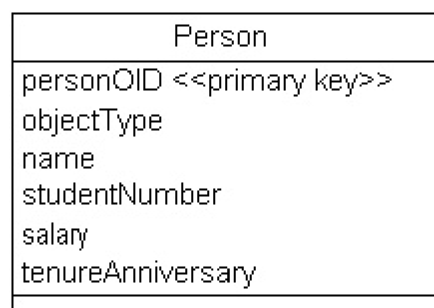


Figure 7 presents the persistence model when each concrete class is mapped to a data entity. With this strategy I only had to add a new table, although the issue of

how to handle objects that change their relationship with us (students become professors) has now become more complex because we have added the promotion of professors to tenured professors.

Figure 7. Mapping the concrete classes of the extended hierarchy to data entities

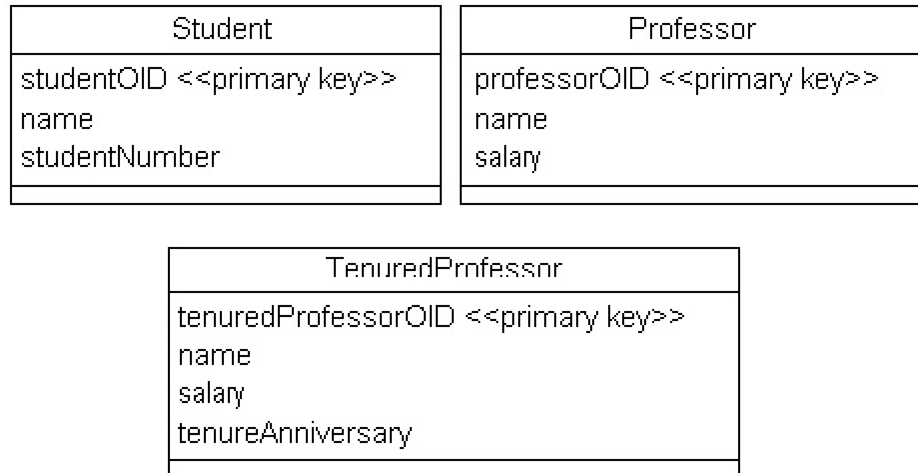
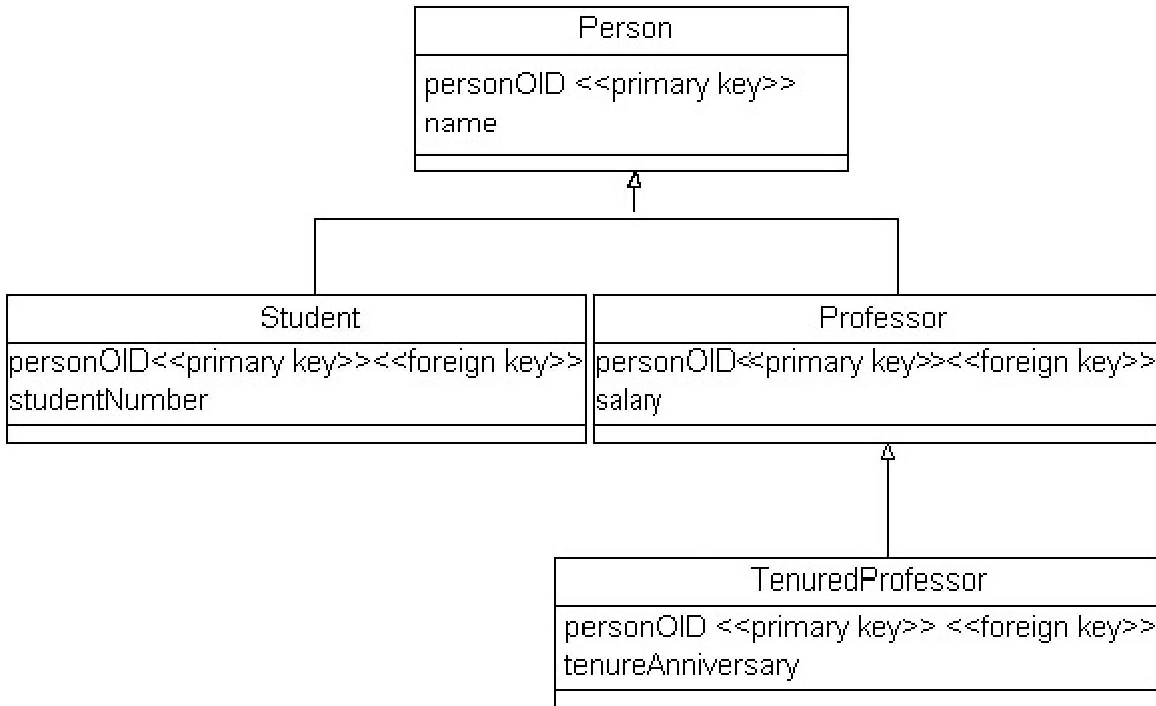


Figure 8 presents the solution for the third mapping strategy -- mapping a single class to a single data entity. This required me to add a new table that included only the new attributes of the `TenuredProfessor` class. The disadvantage of this approach is that it requires several database accesses to work with instances of the new class.

Figure 8. Mapping all classes of the extended hierarchy to data entities



The point to go away with is that none of these approaches is perfect; each has its strengths and weaknesses. They are compared here in Table 1.

Table 1. Comparing the approaches to mapping inheritance

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

4. Mapping associations, aggregation, and composition

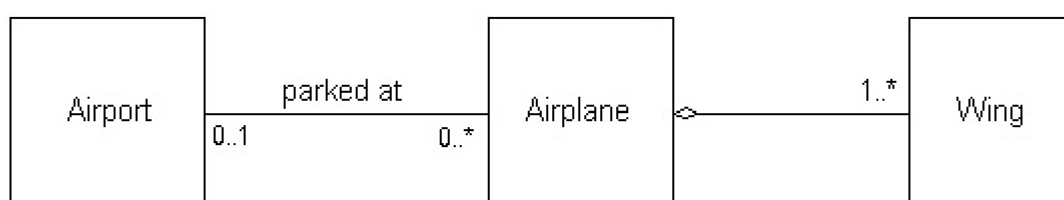
Not only must you map objects into the database, you must also map the relationships that the object is involved with so they can be restored at a later date. There are four types of relationships that an object can be involved with: inheritance, association, aggregation, and composition. To map these relationships effectively, we must understand the differences between them, how to implement

relationships generally, and how to implement many-to-many relationships specifically.

The difference between association, aggregation, and composition

From a database perspective, the only difference between association and aggregation/composition relationships is how tightly the objects are bound to each other. With aggregation and composition, anything you do to the whole in the database you almost always need to do to the parts, whereas with association that is not the case.
In Figure 9 you see three classes, two of which have a simple association between them and two that share an aggregation relationship (actually, composition would likely have been a more accurate way to model this). (See "Building Object Applications That Work" in Resources for more information about relationships.)
From a database point of view, aggregation/composition 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 is not 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.

Figure 9. The difference between association and aggregation/composition

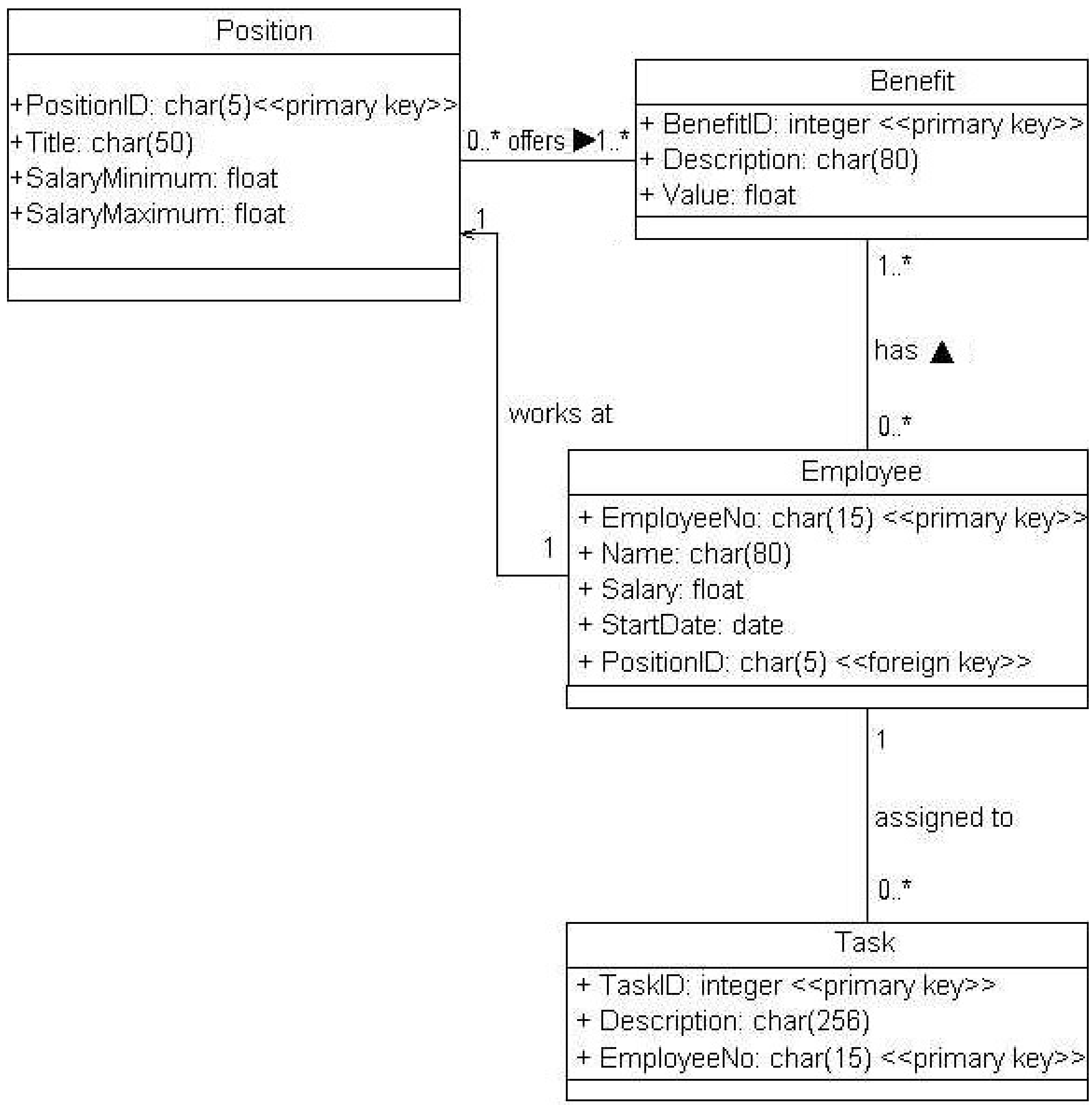


5. Implementing relationships in relational databases

Relationships in relational databases are maintained through the use of foreign keys. A foreign key is one or more data attributes that appear in one table that may be part of, or is coincidental with, the key of another table. Foreign keys allow you to relate a row in one table with a row 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.
In Figure 10 you see three tables, their keys (OIDs), and the foreign keys used to implement the relationships between them. First, there is a one-to-one association between the `Position` and `Employee` data entities. A one-to-one association is one

in which the maximums of each of its multiplicities are one. To implement this relationship I used the attribute `positionOID`, the key of the `Position` data entity, in the `Employee` data entity. I was forced to do it this way because the association is uni-directional -- employee rows know about their position rows but not the other way around. Had this been a bi-directional association, I would have had to add a foreign key called `employeeOID` in `Position` as well. Second, I implemented the many-to-one association (also referred to as a one-to-many association) between `Employee` and `Task` using the same sort of approach, the only difference being that I had to put the foreign key in `Task` because it was on the 'many' side of the relationship.

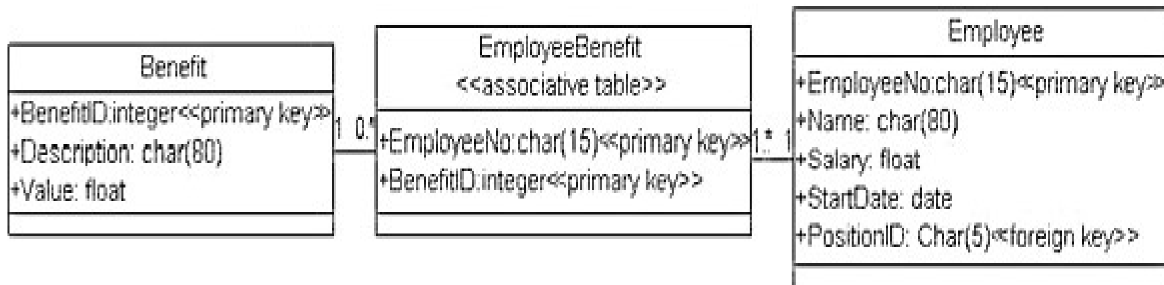
Figure 10. A persistence model for a simple human resources database.



Implementing many-to-many associations

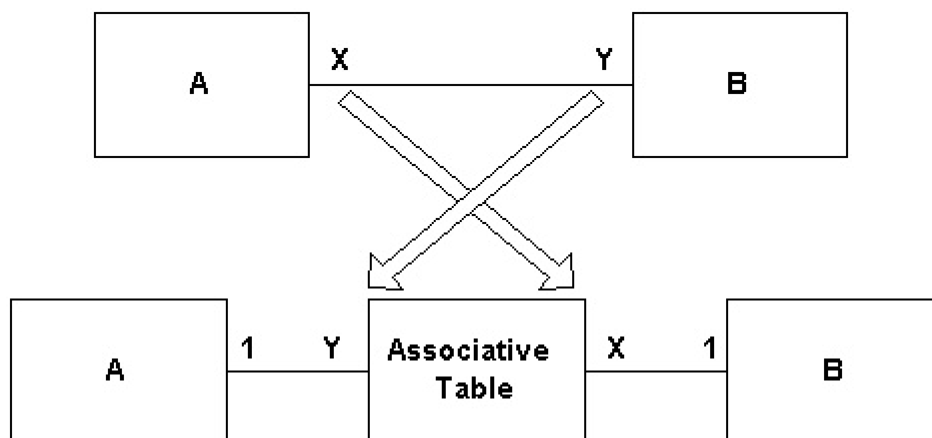
To implement many-to-many relationships, you need the concept of an associative table, a data entity whose sole purpose is to maintain the association between two or more tables in a relational database. In Figure 10 you see that there is a many-to-many relationship between `Employee` and `Benefit`. In Figure 11 you 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. The name of an associative table is typically either the combination of the names of the tables that it associates or the name of the association that it implements. In this case I chose `EmployeeBenefit` over `BenefitEmployee` and `has` because I felt it reflected the nature of the association better.

Figure 11. Implementing a many-to-many relationship in a relational database



Notice the application of multiplicities in Figure 11. The rule is that the multiplicities "cross over" once the associative table is introduced, as indicated in Figure 12. A multiplicity of '1' is always introduced on the outside edges, as you can see in Figures 11 and 12, to preserve the overall multiplicity of the original association. The original association indicated that an employee has one or more benefits and that any given benefit is given to zero or more employees. In Figure 11 you see that this is still true even with the associative table in place to maintain the association.

Figure 12. Introducing an associative table



It is important to note that I choose to apply the stereotype "<<associative table>>" rather than the notation for associative classes -- a dashed line connecting the associative class to the association that it describes -- for two reasons. First, the purpose of an associative table is to implement an association, whereas the purpose of an associative class is to describe an association. Second, the approach taken in Figure 11 reflects the actual implementation strategy that you would need to take using relational technology.

Summary

In this paper you discovered the basics of mapping objects to relational databases. It is possible to successfully and easily store objects in relational databases if you follow the steps described in this paper. If you have any questions, please feel free to e-mail me at scott.ambler@ronin-intl.com

Resources

- *Building Object Applications That Work: Your Step-By-Step Handbook for Developing Robust Systems with Object Technology* by Scott W. Ambler (New York:SIGS Books/Cambridge University Press)
- *Process Patterns: Building Large-Scale Systems Using Object Technology* by Scott W. Ambler (New York:SIGS Books/Cambridge University Press)
- *The Object Primer 2 nd Edition -- The Application Developer's Guide to Object-Orientation* by Scott W. Ambler (New York:Cambridge University Press)
- The Process Patterns Resource Page by Scott W. Ambler
- "Enhancing the Unified Process" by Scott W. Ambler
- Towards a UML Profile for a Relational Persistence Model: Working Page by Scott W. Ambler

About the author

Scott W. Ambler is President of Ronin International, a consulting firm specializing in object-oriented software process mentoring, architectural modeling, and Enterprise JavaBeans (EJB) development. He has authored or co-authored several books about object-oriented development, including the recently released *The Object Primer 2nd Edition*, which covers, in detail, the subjects summarized in this article. He can be reached at scott.ambler@ronin-intl.com.