# Mapping Objects to Relational Databases: O/R Mapping In Detail




Most modern business application development projects use object technology such as Java or C# to build the application software and **relational databases** to store the data. This isn't to say that you don't have other options, there are many applications built with procedural languages such as COBOL and many systems will use object databases or XML databases to store data. However, because object and relational technologies are by far the norm that's what I assume you're working with in this article. If you're working with different storage technologies then many of the concepts are still applicable, albeit with modification (don't worry, **Realistic XML** overviews mapping issues pertaining to objects and XML).



Unfortunately we need to deal with the **object relational (O/R) impedance mismatch**, and to do so you need to understand two things: the process of mapping objects to relational databases and how to implement those mappings. in this article the term "mapping" will be used to refer to how objects and their relationships are mapped to the tables and relationships between them in a database. As you'll soon find out it isn't quite as straightforward as it sounds although it isn't too bad either.

## Table of Contents



## 1. Basic Concepts

When learning how to map objects to relational databases the place to start is with the data attributes of a class. An attribute will map to zero or more columns in a relational database. Remember, not all attributes are persistent, some are used for temporary calculations. For example, a *Student* object may have an *averageMark* attribute that is needed within your application but isn't saved to the database because it is calculated by the application. Because some attributes of an objects are objects in their own right, a *Customer* object has an *Address* object as an attribute – this really reflects an association between the two classes that would likely need to be mapped, and the attributes of the *Address* class itself will need to be mapped. The important thing is that this is a recursive definition: At some point the attribute will be mapped to zero or more columns.

The easiest mapping you will ever have is a property mapping of a single attribute to a single column. It is even simpler when the each have the same basic types, e.g. they're both dates, the attribute is a string and the column is a char, or the attribute is a number and the column is a float.

### Mapping Terminology

**Mapping (v)**. The act of determining how objects and their relationships are persisted in permanent data storage, in this case relational databases.

**Mapping (n)**. The definition of how an object's property or a relationship is persisted in permanent storage.

**Property**. A data attribute, either implemented as a physical attribute such as the string *firstName* or as a virtual attribute implemented via an operation such as *getTotal()* which returns the total of an order.

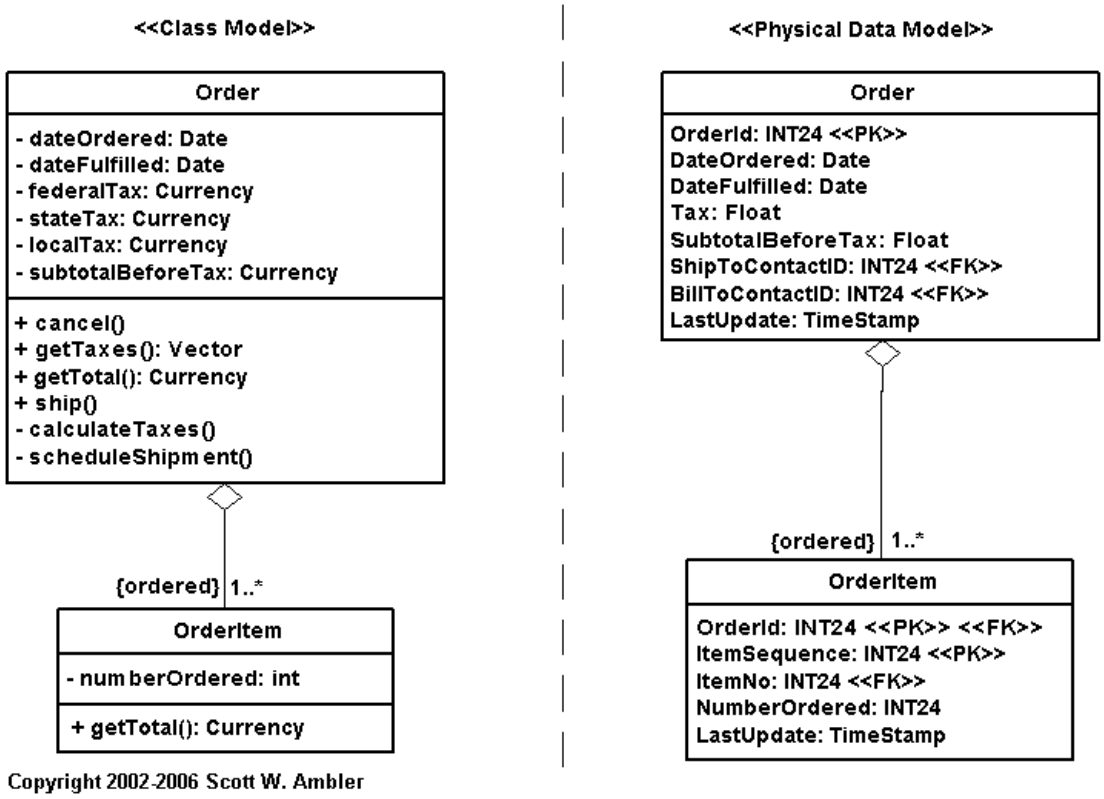
**Property mapping**. A mapping that describes how to persist an object's property.

**Relationship mapping**. A mapping that describes how to persist a relationship (association, aggregation, or composition) between two or more objects.

It can make it easier to think that classes map to tables, and in a way they do, but not always directly. Except for very simple databases you will never have a one-to-one mapping of classes to tables, something you will see later in this article with regards to **inheritance mapping**. However, a common theme that you will see throughout this article is that a one class to one table mapping is preferable for your initial mapping (**performance tuning** may motivate you to refactor your mappings).

For now, let's keep things simple. **Figure 1** depicts two models, a UML class diagram and a physical data model which follows the **UML data modeling profile**. Both diagrams depict a portion of a simple schema for an order system. You can see how the attributes of the classes could be mapped to the columns of the database. For example, it appears that the *dateFulfilled* attribute of the *Order* class maps to the *DataFulfilled* column of the *Order* table and that the *numberOrdered* attribute of the *OrderItem* class maps to the *NumberOrdered* column of the *OrderItem* table.

**Figure 1. Simple mapping example.**






Note that these initial property mappings were easy to determine for several reasons. First, similar naming standards were used in both models, an aspect of **Agile Modeling (AM)'s Apply Modeling Standards practice**. Second, it is very likely that the same people created both models. When people work in separate teams it is quite common for their solutions to vary, even when the teams do a very good job, because they make different design decisions along the way. Third, one model very likely drove the development of the other model. In **Different Projects Require Different Strategies** I argued that when you are building a new system that your **object schema should drive the development of your database schema**.

The easiest mapping you will ever have is a property mapping of a single attribute to a single column. It is even simpler when the each have the same basic types, e.g. they're both dates, the attribute is a string and the column is a char, or the attribute is a number and the column is a float.

Even though the two schemas depicted in **Figure 1** are very similar there are differences. These differences mean that the mapping isn't going to be perfect. The differences between the two schemas are:

- There are several attributes for tax in the object schema yet only one in the data schema. The three attributes for tax in the *Order* class presumably should be added up and stored in the *tax* column of the *Order* table when the object is saved. When the object is read into memory, however, the three attributes would need to be calculated (or a lazy initialization approach would need to be taken and each attribute would be calculated when it is first accessed). A schema difference such as this is a good indication that the database schema needs to be refactored to split the tax column into three.
- The data schema indicates **keys** whereas the object schema does not. Rows in tables are uniquely identified by **primary keys** and relationships between rows are maintained through the use of foreign keys. Relationships to objects, on the other hand, are implemented via references to those objects not through foreign keys. The implication is that in order to fully persist the object data, and the relationships which the objects are involved in, that the objects need to know about the key values used in the database to identify them. This additional information is called "**shadow information**".
- Different types are used in each schema. The *subTotalBeforeTax* attribute of *Order* is of the type *Currency* whereas the *SubTotalBeforeTax* column of the *Order* table is a float. When you implement this mapping you will need to be able to convert back and forth between these two representations without loss of information.

## 1.1 Shadow Information and Scaffolding

Shadow information is any data that objects need to maintain, above and beyond their normal domain data, to persist themselves. This typically includes primary key information, particularly when the primary key is a surrogate key that has no business meaning, **concurrency control** markings such as timestamps or incremental counters, and versioning numbers. For example, in **Figure 1** you see that the *Order* table has an *OrderID* column used as a primary key and a *LastUpdate* column that is used for optimistic concurrency control that the *Order* class does not have. To persist an order object properly the Order class would need to implement shadow attributes that maintain these values.

**Figure 2** shows a detailed design class model for the *Order* and *OrderItem* classes. There are several changes from **Figure 1**. First, the new

diagram shows the shadow attributes that the classes require to properly persist themselves. Shadow attributes have an implementation visibility, there is a space in front of the name instead of a minus sign, and are assigned the stereotype <<persistence>> (this is not a UML standard). Second, it shows the scaffolding attributes required to implement the relationship the two classes. Scaffolding attributes, such as the *orderItems* vector in *Order*, also have an implementation visibility. Third, a *getTotalTax()* operation was added to the *Order* class to calculate the value required for the *tax* column of the *Order* table. This is why I use the term property mapping instead of attribute mapping – what you really want to do is map the properties of a class, which sometimes are implemented as simple attributes and other times as one or more operations, to the columns of a database.

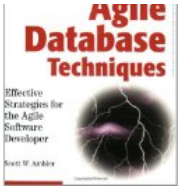
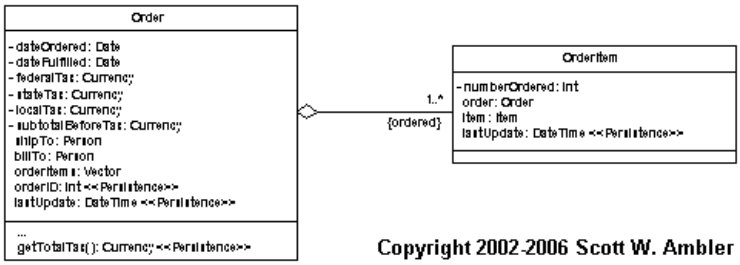



**Figure 2. Including "shadow information" on a class diagram.**




One type of shadow information that I have not discussed yet is a boolean flag to indicate whether an object currently exists in the database. The problem is that when you save data to a relational database you need to use a SQL update statement if the object was previously retrieved from the database and a SQL insert statement if the data does not already exist. A common practice is for each class to implement an *isPersistent* boolean flag, not shown in **Figure 2**, that is set to true when the data is read in from the database and set to false when the object is newly created.

It is a common **style convention** in the UML community to not show shadow information, such as keys and concurrency markings, on class diagrams. Similarly, the common convention is to not model scaffolding code either. The idea is that everyone knows you need to do this sort of thing, so why waste your time modeling the obvious?

Shadow information doesn't necessarily need to be implemented by the business objects, although your application will need to take care of it somehow. For example, with **Enterprise JavaBeans (EJBs)** you store primary key information outside of EJBs in primary key classes, the individual object references a corresponding primary key object. The Java Data Object (JDO) approach goes one step further and implement shadow information in the JDOs and not the business objects.

## 1.2 Mapping Meta Data

**Figure 3** depicts the meta data representing the property mappings required to persist the *Order* and *OrderItem* classes of **Figure 2**. Meta data is information about data. **Figure 3** is important for several reasons. First, we need some way to represent mappings. We could put two schemas side by side, as you see in **Figure 1**, and then draw lines between them but that gets complicated very quickly. Another option is a tabular representation that you see in **Figure 3**. Second, the concept of mapping meta data is critical to the functioning of **persistence frameworks** which are a **database encapsulation strategy** that can enable agile database techniques.

**Figure 3. Meta data representing the property maps.**

| Property | Column |
|---|---|
| Order.orderID | Order.OrderID |
| Order.dateOrdered | Order.DateOrdered |
| Order.dateFulfilled | Order.DateFulfilled |
| Order.getTotalTax() | Order.Tax |
| Order.subtotalBeforeTax | Order.SubtotalBeforeTax |
| Order.shipTo.personID | Order.ShipToContactID |
| Order.billTo.personID | Order.BillToContactID |
| Order.lastUpdate | Order.LastUpdate |
| OrderItem.ordered | OrderItem.OrderID |
| Order.orderItems.position(orderItem) | OrderItem.ItemSequence |
| OrderItem.item.number | OrderItem.ItemNo |
| OrderItem.numberOrdered | OrderItem.NumberOrdered |
| OrderItem.lastUpdate | OrderItem.LastUpdate |

The naming convention that I'm using is reasonably straightforward: *Order.dateOrdered* refers to the *dateOrdered* attribute of the *Order* class. Similarly *Order.DateOrdered* refers to the *DateOrdered* column of the *Order* table. *Order.getTotalTax()* refers to the *getTotalTax()* operation of *Order* and *Order.billTo.personID* is the *personID* attribute of the *Person* object referenced by the *Order.billTo* attribute. Likely the most difficult property to understand is *Order.orderItems.position(orderItem)* which refers to the position within the *Order.orderItems* vector of the instance of *OrderItem* that is being saved.

**Figure 3** hints at an important part of the O/R **impedance mismatch** between object technology and relational technology. Classes implement both behavior and data whereas relational database tables just implement data. The end result is that when you're mapping the properties of classes into a relational database you end up mapping operations such as *getTotalTax()* and *position()* to columns. Although it didn't happen in this example, you often need to map two operations that represent a single property to a column – one operation to set the value, e.g. *setFirstName()*, and one operation to retrieve the value, e.g. *getFirstName()*. These operations are typically called setters and getters respectively, or sometimes mutators and accessors.

Whenever a key column is mapped to a property of a class, such as the mapping between *OrderItem.ItemSequence* and *Order.orderItems.position(orderItem)*, this is really part of the effort of relationship mapping, discussed later in this article. This is because keys implement relationships in relational databases.

## 1.3 How Mapping Fits Into The Overall Process

See the essay **Evolutionary Development**.

## 2. Mapping Inheritance Structures

Relational databases do not natively support inheritance, forcing you to map the inheritance structures within your object schema to your data schema. Although there is somewhat of a backlash against inheritance within the object community, due in most part to the fragile base class problem, my experience is that this problem is mostly due to poor encapsulation practices among object developers than with the concept of inheritance. What I'm saying is that the fact you need to do a little bit of work to map an inheritance hierarchy into a relational database shouldn't dissuade you from using inheritance where appropriate.

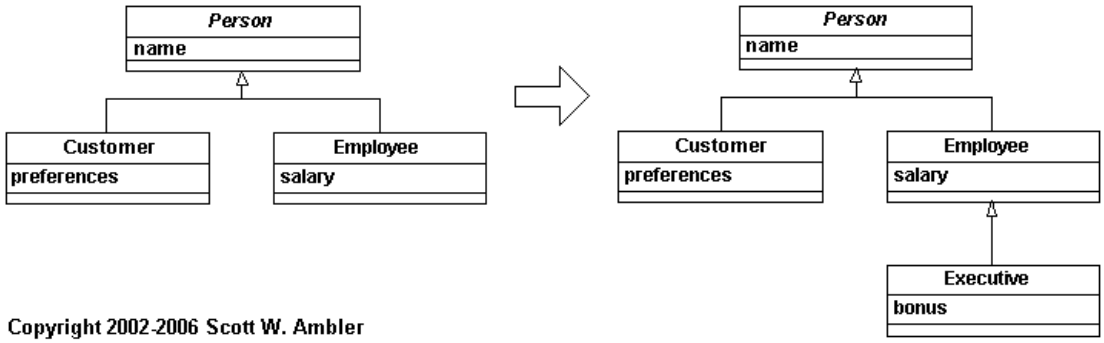
The concept of inheritance throws in several interesting twists when saving objects into a relational DB. How do you organize the inherited attributes within your data model? In this section you'll see that there are three primary solutions for mapping inheritance into a relational database, and a fourth supplementary technique that goes beyond inheritance mapping. These techniques are:

- **Map the entire class hierarchy to a single table**
- **Map each concrete class to its own table**
- **Map each class to its own table**
- **Map the classes into a generic table structure**

To explore each technique I will discuss how to map the two versions of the class hierarchy presented in **Figure 4**. The first version depicts three classes – *Person*, an abstact class, and two concrete classes, *Employee* and *Customer*. You know that *Person* is abstract because its name is shown in italics. In older versions of the UML the constraint "{abstract}" would have been used instead. The second version of the hierarchy adds a fourth concrete class to the hierarchy, *Executive*. The idea is that you have implemented the first class hierarchy and are now presented with a new requirement to support giving executives, but not non-executive employees, fixed annual bonuses. The *Executive* class was added to support this new functionality.

For the sake of simplicity I have not modeled all of the attributes of the classes, nor have I modeled their full signatures, nor have I modeled any of the operations. This diagram is just barely good enough for my purpose, in other words it is an agile model. Furthermore these hierarchies could be approved by applying the **Party analysis pattern** or the **Business Entity** analysis pattern. I haven't done this because I need a simple example to explain mapping inheritance hierarchies, not to explain the effective application of analysis patterns – I always follow **AM's Model With A Purpose principle**.

**Figure 4. Two versions of a simple class hierarchy.**



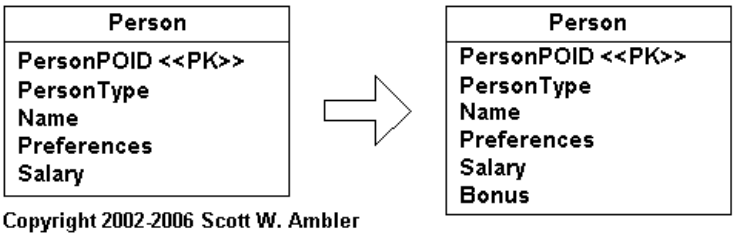



Inheritance can also be a problem when it's misapplied – for example, the hierarchy in **Figure 4** could be better modeled via the Party (**Hay 1996**, **Fowler 1997**) or the Business Entity (**Ambler 1997**) patterns. For example, if someone can be both a customer and an employee you would have to objects in memory for them, which may be problematic for your application. I've chosen this example because I needed a simple, easy to understand class hierarchy to map.

### 2.1 Map Hierarchy To A Single Table

Following this strategy you store all the attributes of the classes in one table. **Figure 5** depicts the data model for the class hierarchies of **Figure 4** when this approach is taken. The attributes of each the classes are stored in the table *Person*, a good table naming strategy is to use the name of the hierarchy's root class, in a very straightforward manner.

**Figure 5. Mapping to a single table.**





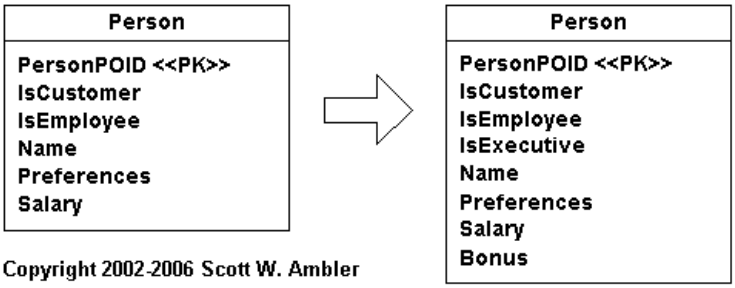

Two columns have been added to the table – *PersonPOID* and *PersonType*. The first column is the primary key for the table, you know this because of the <<PK>> stereotype, and the second is a code indicating whether the person is a customer, an employee, or perhaps both. *PersonPOID* is a persistent object identifier (POID), often simply called an object identifier (OID), which is a surrogate key. I could have used the optional stereotype of <<Surrogate>> to indicate this but chose not to as POID implies this, therefore indicating the stereotype would only serve to complicate the diagram (follow the **AM practice Depict Models Simply**). **Data Modeling 101** discusses **surrogate keys** in greater detail.

The *PersonType* column is required to identify the type of object that can be instantiated from a given row. For example the value of *E* would indicate the person is an employee, *C* would indicate customer, and *B* would indicate both. Although this approach is straightforward it tends to break down as the number of types and combinations begin to grow. For example, when you add the concept of executives you need to add a code value, perhaps *X*, to represent this. Now the value of *B*,

representing both, is sort of goofy. Furthermore you might have combinations involving executives now, for example it seems reasonable that someone can be both an executive and a customer so you'd need a code for this. When you discover that combinations are possible you should consider applying the ***Replace Type Code With Booleans*** database refactoring, as you see in **Figure 6**.

For the sake of simplicity I did not include columns for concurrency control, such as the time stamp column included in the tables of **Figure 2**, nor did I include columns for data versioning.

**Figure 6. A refactored approach.**

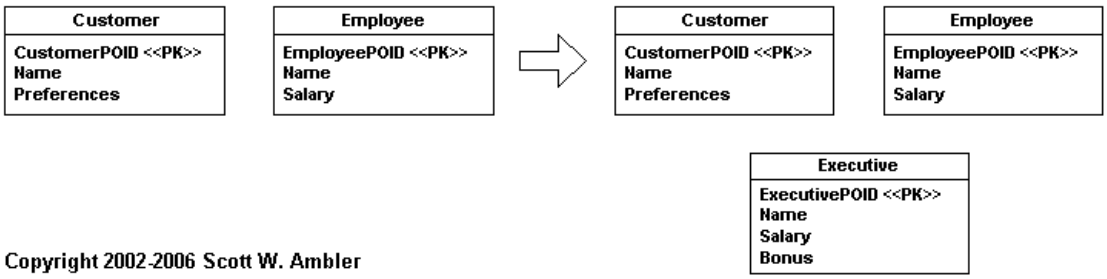





## 2.2 Map Each Concrete Class To Its Own Table

With this approach a table is created for each concrete class, each table including both the attributes implemented by the class and its inherited attributes. **Figure 7** depicts the physical data model for the class hierarchy of **Figure 4** when this approach is taken. There are tables corresponding to each of the *Customer* and *Employee* classes because they are concrete, objects are instantiated from them, but not *Person* because it is abstract. Each table was assigned its own primary key, *customerPOID* and *employeePOID* respectively. To support the addition of *Executive* all I needed to do was add a corresponding table with all of the attributes required by executive objects.

**Figure 7. Mapping concrete classes to tables.**



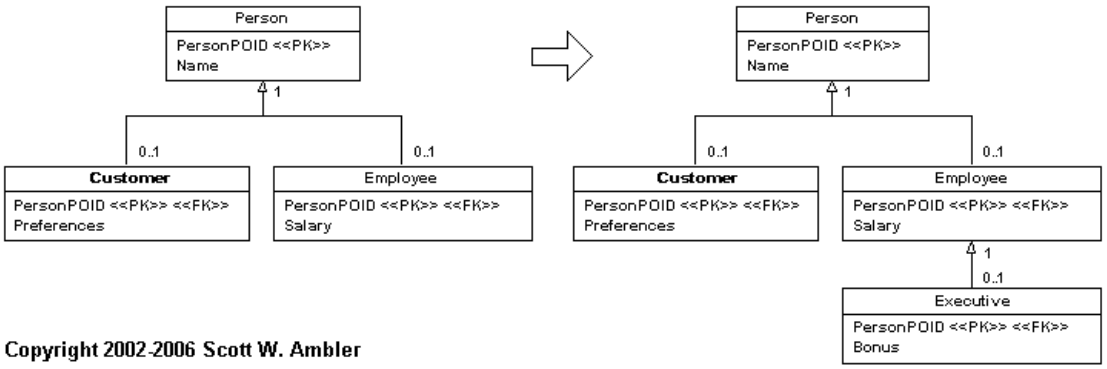



## 2.3 Map Each Class To Its Own Table

Following this strategy you create one table per class, with one column per business attributes and any necessary identification information (as well as other columns required for concurrency control and versioning). **Figure 8** depicts the physical data model for the class hierarchy of **Figure 4** when each class is mapped to a single table. The data for the *Customer* class is stored in two tables, *Customer* and *Person*, therefore to retrieve this data you would need to join the two tables (or do two separate reads, one to each table).

The application of keys is interesting. Notice how *personPOID* is used as the primary key for all of the tables. For the *Customer*, *Employee*, and *Executive* tables the *personPOID* is both a primary key and a foreign key. In the case of *Customer*, *personPOID* is its primary key and a foreign key used to maintain the relationship to the *Person* table. This is indicated by application of two stereotypes, <<PK>> and <<FK>>. In older versions of the **UML** it wasn't permissible to assign several stereotypes to a single model element but this restriction was lifted in UML version 1.4.

**Figure 8. Mapping each class to its own table.**



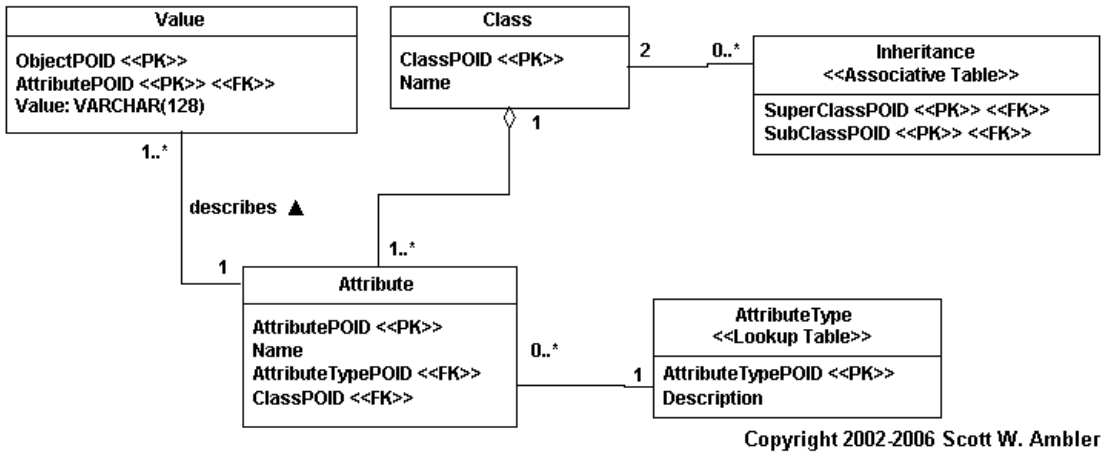



A common modification that you may want to consider is the addition of a type column, or boolean columns as the case may be, in the *Person* table to indicate the applicable subtypes of the person. Although this is additional overhead it makes some types of queries easier. The addition of views is also an option in many cases, an approach that I prefer over the addition of type or boolean columns because they are easier to maintain.

## 2.4 Map Classes To A Generic Table Structure

A fourth option for mapping inheritance structures into a relational database is to take a generic, sometimes called meta-data driven approach, to mapping your classes. This approach isn't specific to inheritance structures, it supports all forms of mapping. In **Figure 9** you see a data schema for storing the value of attributes and for traversing inheritance structures. The schema isn't complete, it could be extended to map associations for example, but it's sufficient for our purposes. The value of a single attribute is stored in the *Value* table, therefore to store an object with ten business attributes there would be ten records, one for each attribute. The *Value.ObjectPOID* column stores the unique identifier for the specific object (this approach assumes a common key strategy across all objects, when this isn't the case you'll need to extend this table appropriately). The *AttributeType* table contains rows for basic data types such as data, string, money, integer and so on. This information is required to convert the value of the object attribute into the varchar stored in *Value.Value*.

**Figure 9. A generic data schema for storing objects.**




Let's work through an example of mapping a single class to this schema. To store the *OrderItem* class in **Figure 2** there would be three records in the *Value* table. One to store the value for the number of items ordered, one to store the value of the *OrderPOID* that this order item is part of, and one to store the value of the *ItemPOID* that describes the order item. You may decide to have a fourth row to store the value of the *lastUpdated* shadow attribute if you're taking an optimistic locking approach to **concurrency control**. The *Class* table would include a row for the *OrderItem* class and the *Attribute* table would include one row for each attribute stored in the database (in this case either 3 or 4 rows).

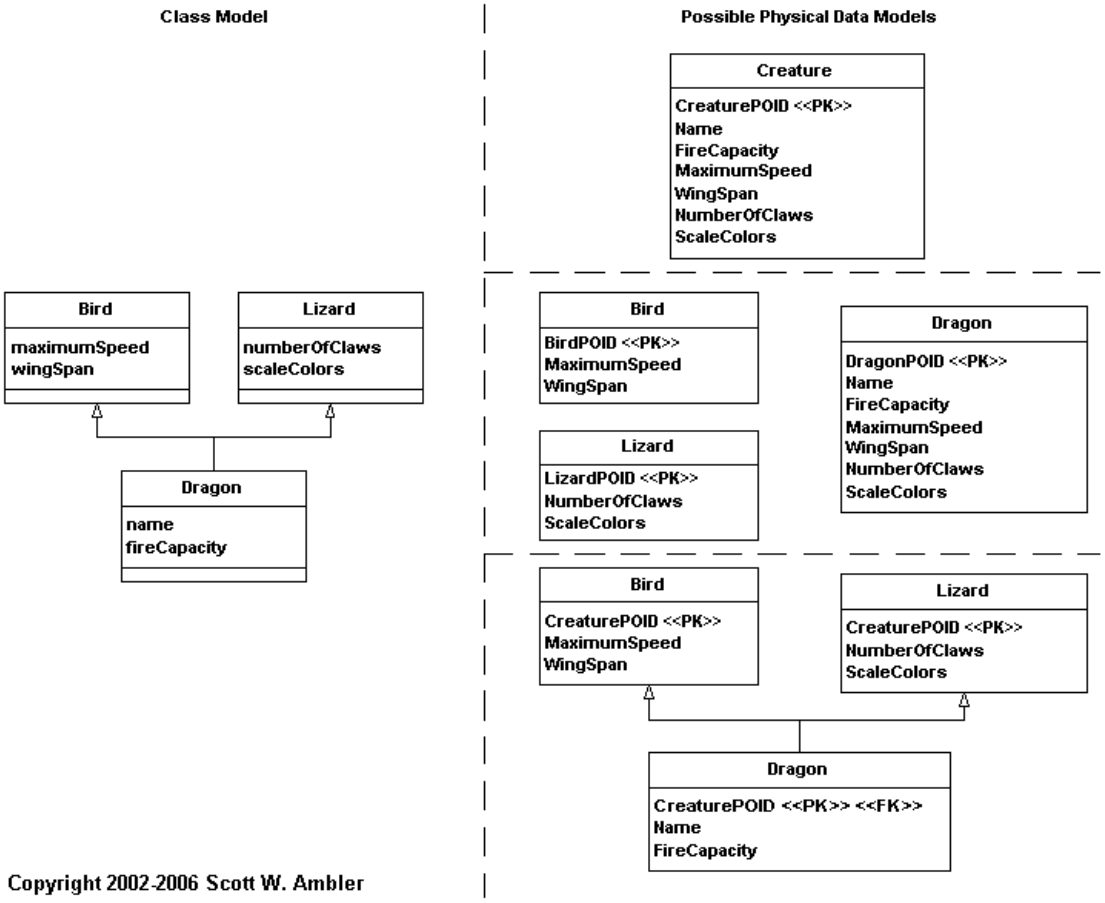
Now let's map the inheritance structure between *Person* and *Customer*, shown in **Figure 4**, into this schema. The *Inheritance* table is the key to inheritance mapping. Each class would be represented by a row in the *Class* table. There would also be a row in the *Inheritance* table, the value of *Inheritance.SuperClassPOID* would refer to the row in *Class* representing *Person* and *Inheritance.SubClassPOID* would refer to the row in *Class* representing *Customer*. To map the rest of the hierarchy you require one row in *Inheritance* for each inheritance relationship.

## 2.5 Mapping Multiple Inheritance

Until this point I have focused on mapping single inheritance hierarchies, single inheritance occurs when a subclass such as *Customer* inherits directly from a single parent class such as *Person*. Multiple inheritance occurs when a subclass has two or more direct superclasses, such as *Dragon* directly inheriting from both *Bird* and *Lizard* in **Figure 10**. Multiple inheritance is generally seen as a questionable feature of an object-oriented language, since 1990 I have only seen one domain problem where multiple inheritance made sense, and as a result most languages choose not to support it. However, languages such as C++ and Eiffel do support it so you may find yourself in a situation where you need to map a multiple inheritance hierarchy to a relational database.

**Figure 10** shows the three data schemas that would result from applying each of the three inheritance mapping strategies. As you can see mapping multiple inheritance is fairly straightforward, there aren't any surprises in **Figure 10**. The greatest challenge in my experience is to identify a reasonable table name when mapping the hierarchy into a single table, in this case *Creature* made the most sense.

**Figure 10. Mapping multiple inheritance.**




## 2.6 Comparing The Strategies

None of these mapping strategies are ideal for all situations, as you can see in **Table 1**. My experience is that the easiest strategy to work with is to have one table per hierarchy at first, then if you need to refactor your schema according. Sometimes I'll start by applying the one table per class strategy whenever my team is motivated to work with a "pure design approach". I stay away from using one table per concrete class because it typically results in the need to copy data back and forth between tables, forcing me to refactor it reasonably early in the life of the project anyway. I rarely use the generic schema approach because it simply doesn't scale very well.

It is important to understand that you can combine the first three strategies – one table per hierarchy, one table per concrete class, and one table per class – in any given application. You can even combine these strategies in a single, large hierarchy.

**Table 1. Comparing the inheritance mapping strategies.**

| Strategy | Advantages | Disadvantages | When to Use |
|---|---|---|---|
| **One table per hierarchy** | Simple approach.<br><br>Easy to add new classes, you just need to add new columns for the additional data.<br><br>Supports polymorphism by simply changing the type of the row.<br><br>Data access is fast because the data is in one table.<br><br>Ad-hoc **reporting** is very easy because all of the data is found in one table. | Coupling within the class hierarchy is increased because all classes are directly coupled to the same table. A change in one class can affect the table which can then affect the other classes in the hierarchy.<br><br>Space potentially wasted in the database.<br><br>Indicating the type becomes complex when significant overlap between types exists.<br><br>Table can grow quickly for large hierarchies. | This is a good strategy for simple and/or shallow class hierarchies where there is little or no overlap between the types within the hierarchy. |
| **One table per concrete class** | Easy to do ad-hoc **reporting** as all the data you need about a single class is stored in only one table.<br><br>Good performance to access a single object's data. | When you modify a class you need 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 add columns to the *Customer*, *Employee*, and *Executive* tables.<br><br>Whenever an object changes its role, perhaps you hire one of your customers, you need to copy the data into the appropriate table and assign it a new POID value (or perhaps you could reuse the existing | When changing types and/or overlap between types is rare. |

| | | | |
|---|---|---|---|
| | | POID value).<br><br>It is difficult to support multiple roles and still maintain data integrity. For example, where would you store the name of someone who is both a customer and an employee? | |
| **One table per class** | Easy to understand because of the one-to-one mapping.<br><br>Supports polymorphism very well as you merely have records in the appropriate tables for each type.<br><br>Very easy to modify superclasses and add new subclasses as you merely need to modify/add one table.<br><br>Data size grows in direct proportion to growth in the number of objects. | There are many tables in the database, one for every class (plus tables to maintain relationships).<br><br>Potentially 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).<br><br>Ad-hoc **reporting** on your database is difficult, unless you add views to simulate the desired tables. | When there is significant overlap between types or when changing types is common. |
| **Generic schema** | Works very well when database access is encapsulated by a robust **persistence framework**.<br><br>It can be extended to provide meta data to support a wide range of mappings, including relationship mappings. In short, it is the start at a mapping meta data engine.<br><br>It is incredibly flexible, enabling you to quickly change the way that you store objects because you merely need to update the meta data stored in the *Class*, *Inheritance*, *Attribute*, and *AttributeType* tables accordingly. | Very advanced technique that can be difficult to implement at first.<br><br>It only works for small amounts of data because you need to access many database rows to build a single object.<br><br>You will likely want to build a small administration application to maintain the meta data.<br><br>**Reporting** against this data can be very difficult due to the need to access several rows to obtain the data for a single object. | For complex applications that work with small amounts of data, or for applications where you data access isn't very common or you can pre-load data into caches. |

## 3. Mapping Object Relationships

In addition to property and inheritance mapping you need to understand the art of relationship mapping. There are three types of object relationships that you need to map: association, aggregation, and composition. For now, I'm going to treat these three types of relationship the same – they are mapped the same way although there are interesting nuances when it comes to **referential integrity**.

### 3.1 Types of Relationships

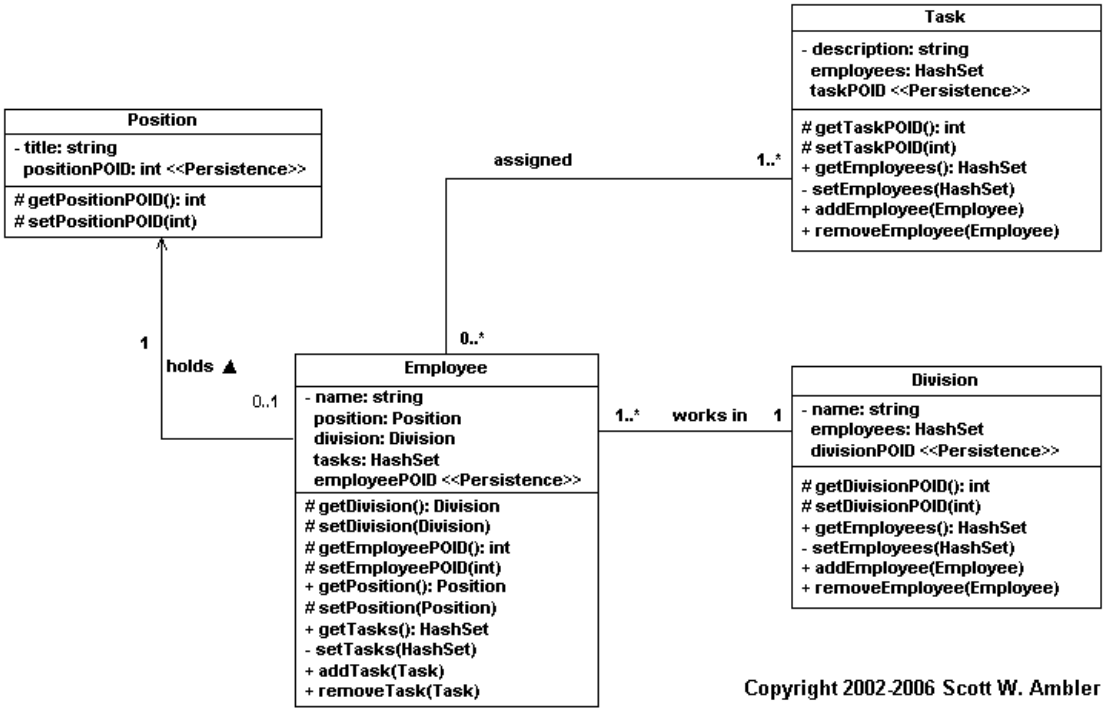
There are two categories of object relationships that you need to be concerned with when mapping. The first category is based on multiplicity and it includes three types:

- **One-to-one relationships**. This is a relationship where the maximums of each of its multiplicities is one, an example of which is *holds* relationship between *Employee* and *Position* in **Figure 11**. An employee holds one and only one position and a position may be held by one employee (some positions go unfilled).
- **One-to-many relationships**. Also known as a many-to-one relationship, this occurs when the maximum of one multiplicity is one and the other is greater than one. An example is the *works in* relationship between *Employee* and *Division*. An employee works in one division and any given division has one or more employees working in it.
- **Many-to-many relationships**. This is a relationship where the maximum of both multiplicities is greater than one, an example of which is the *assigned* relationship between *Employee* and *Task*. An employee is assigned one or more tasks and each task is assigned to zero or more employees.

The second category is based on directionality and it contains two types, uni-directional relationships and bi-directional relationships.

- **Uni-directional relationships**. A uni-directional relationship when an object knows about the object(s) it is related to but the other object(s) do not know of the original object. An example of which is the *holds* relationship between *Employee* and *Position* in **Figure 11**, indicated by the line with an open arrowhead on it. *Employee* objects know about the position that they hold, but *Position* objects do not know which employee holds it (there was no requirement to do so). As you will soon see, uni-directional relationships are easier to implement than bi-directional relationships.
- **Bi-directional relationships**. A bi-directional relationship exists when the objects on both end of the relationship know of each other, an example of which is the *works in* relationship between *Employee* and *Division*. *Employee* objects know what division they work in and *Division* objects know what employees work in them.

**Figure 11. Relationships between objects.**




It is possible to have all six combinations of relationship in object schemas. However one aspect of the **impedance mismatch** between object technology and relational technology is that relational technology does not support the concept of uni-directional relationships – in relational databases all associations are bi-directional (relationships are implemented via foreign keys, which can be joined/traversed in either direction).

## 3.2 How Object Relationships Are Implemented

Relationships in object schemas are implemented by a combination of references to objects and operations. When the multiplicity is one (e.g. 0..1 or 1) the relationship is implemented with a reference to an object, a getter operation, and a setter operation. For example in **Figure 11** the fact that an employee works in a single division is implemented by the *Employee* class via the combination of the attribute *division*, the *getDivision()* operation which returns the value of *division*, and the *setDivision()* operation which sets the value of the *division* attribute. The attribute(s) and operations required to implement a relationship are often referred to as scaffolding.

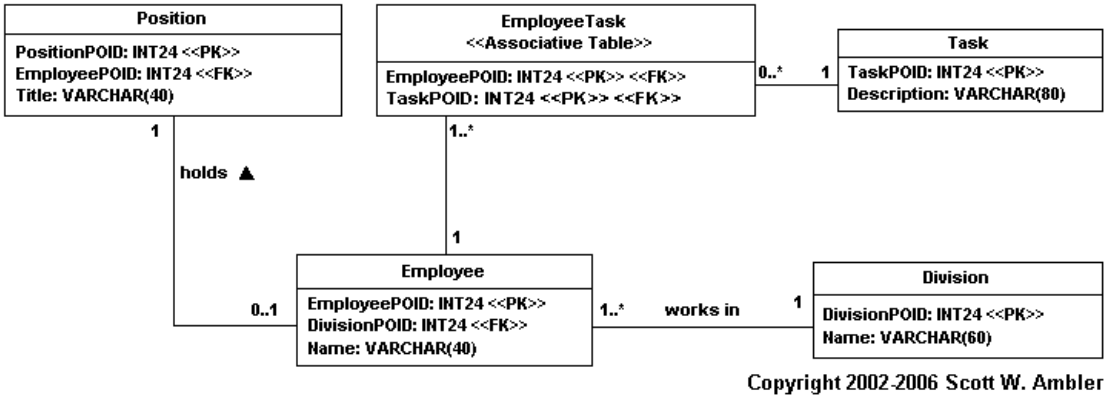
When the multiplicity is many (e.g. N, 0..*, 1..*) the relationship is implemented via a collection attribute, such as an *Array* or a *HashSet* in Java, and operations to manipulate that array. For example the *Division* class implements a *HashSet* attribute named *employees*, *getEmployees()* to get the value, *setEmployees()* to set the value, *addEmployee()* to add an employee into the *HashSet*, and *removeEmployee()* to remove an employee from the *HashSet*.

When a relationship is uni-directional the code is implemented only by the object that knows about the other object(s). For example, in the uni-directional relationship between *Employee* and *Position* only the *Employee* class implements the association. Bi-directional associations, on the other hand, are implemented by both classes, as you can see with the many-to-many relationship between *Employee* and *Task*.

## 3.3 How Relational Database Relationships Are Implemented

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. With a one-to-one relationship the foreign key needs to be implemented by one of the tables. In **Figure 12** you see that the *Position* table includes *EmployeePOID*, a foreign key to the *Employee* table, to implement the association. I could easily have implemented a *PositionPOID* column in *Employee* instead.

**Figure 12. Relationships in a relational database.**




To implement a one-to-many relationship you implement a foreign key from the "one table" to the "many table". For example *Employee* includes a *DivisionPOID* column

to implement the *works in* relationship to *Division*. You could also choose to overbuild your database schema and implement a one-to-many relationship via an associative table, effectively making it a many-to-many relationship.

There are two ways to implement many-to-many associations in a relational database. The first one is to implement in each table the foreign key column(s) to the other table several times. For example to implement the many-to-many relationship between *Employee* and *Task* you could have five *TaskPOID* columns in *Employee* and the *Task* table could include seven *EmployeePOID* columns. Unfortunately you run into a problem with this approach when you assign more than five tasks to an employee or more than seven employees to a single task. A better approach is to implement what is called an associative table, an example of which is *EmployeeTask* in **Figure 12**, which includes the combination of the primary keys of the tables that it associates. With this approach you could have fifty people assigned to the same task, or twenty tasks assigned to the same person, and it wouldn't matter. The basic "trick" is that the many-to-many relationship is converted into two one-to-many relationships, both of which involve the associative table.

Because foreign keys are used to join tables, all relationships in a relational database are effectively bi-directional. This is why it doesn't matter in which table you implement a one-to-one relationship, the code to join the two tables is virtually the same. For example, with the existing schema in **Figure 12** the SQL code to join across the holds relationship would be

```
SELECT * FROM Position, Employee

WHERE Position.EmployeePOID = Employee.EmployeePOID
```

Had the foreign key been implemented in the Employee table the SQL code would be

```
SELECT * FROM Position, Employee

WHERE Position.PositionPOID = Employee.PositionPOID
```

A consistent key strategy within your database can greatly simplify your relationship mapping efforts. The first step is to prefer single-column keys. The next step is to use a globally unique surrogate key, perhaps following the **GUID or HIGH-LOW** strategies, so you are always mapping to the same type of key column.

Now that we understand how to implement relationships in each technology, let's see how you map them. I will describe the mappings from the point of view of mapping the object relationships into the relational database. An interesting thing to remember is that in some cases you have design choices to make. Once again beware of the "magic CASE tool button" that supposedly automates everything for you.

## 3.4 Relationship Mappings

A general rule of thumb with relationship mapping is that you should keep the multiplicities the same. Therefore a **one-to-one** object relationship maps to a one-to-one data relationship, a **one-to-many** maps to a one-to-many, and a **many-to-many** maps to a many-to-many. The fact is that this doesn't have to be the case, you can implement a one-to-one object relationship with to a one-to-many or even a many-to-many data relationship. This is because a one-to-one data relationship is a subset of a one-to-many data relationship and a one-to-many relationship is a subset of a many-to-many relationship.

**Figure 13** depicts the property mappings between the object schema of **Figure 11** and the data schema of **Figure 12**. Note how I have only had to map the business properties and the **shadow information** of the objects, but not **scaffolding attributes** such as *Employee.position* and *Employee.tasks*. These scaffolding attributes are represented via the shadow information that is mapped into the database. When the relationship is read into memory the values of stored in the primary key columns will be stored in the corresponding shadow attributes within the objects. At the same time the relationship that the primary key columns represent will be defined between the corresponding objects by setting the appropriate values in their scaffolding attributes.

**Figure 13. Property mappings.**

| Property | Column |
|---|---|
| Position.title | Position.Title |
| Position.positionPOID | Position.PositionPOID |
| Employee.name | Employee.Name |
| Employee.employeePOID | Employee.EmployeePOID |
| Employee.employeePOID | EmployeeTask.EmployeePOID |
| Division.name | Division.Name |
| Division.divisionPOID | Division.DivisionPOID |
| Task.description | Task.Description |
| Task.taskPOID | Task.TaskPOID |
| Task.taskPOID | EmployeeTask.TaskPOID |

### 3.4.1 One-To-One Mappings

Consider the one-to-one object relationship between *Employee* and *Position*. Let's assume that whenever a *Position* or an *Employee* object is read into memory that the application will automatically traverse the *holds* relationship and automatically read in the corresponding object. The other option would be to manually traverse the relationship in the code, taking a lazy read approach where the other object is read at the time it is required by the application. The trade-offs of these two approaches are discussed in **Implementing Referential Integrity**. **Figure 14** shows how the object relationships are mapped.

**Figure 14. Mapping the relationships.**

| Object Relationship | From | To | Cardinality | Automatic Read | Column(s) | Scaffolding Property |
|---|---|---|---|---|---|---|
| holds | Employee | Position | One | Yes | Position.EmployeePOID | Employee.position |
| held by | Position | Employee | One | Yes | Position.EmployeePOID | Employee.position |
| works in | Employee | Division | One | Yes | Employee.DivisionPOID | Employee.division |
| has working in it | Division | Employee | Many | No | Employee.DivisionPOID | Division.employees |
| assigned | Employee | Task | Many | No | Employee.EmployeePOID | Employee.tasks |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | EmployeeTask.EmployeePOID | |
| assigned to | Task | Employee | Many | No | Task.TaskPOID<br><br>EmployeeTask.TaskPOID | Task.employees |

Let's work through the logic of retrieving a single *Position* object one step at a time:

1. The *Position* object is read into memory.
2. The *holds* relationship is automatically traversed.
3. The value held by the *Position.EmployeePOID* column is used to identify the single employee that needs to be read into memory.
4. The *Employee* table is searched for a record with that value of *EmployeePOID*.
5. The *Employee* object (if any) is read in and instantiated (due to the automatic read indicated in the held by row of Figure 14).
6. The value of the *Employee.position* attribute is set to reference the *Position* object.

Now let's work through the logic of retrieving a single *Employee* object one step at a time:

1. The *Employee* object is read into memory.
2. The *holds* relationship is automatically traversed.
3. The value held by the *Employee.EmployeePOID* column is used to identify the single position that needs to be read into memory.
4. The *Position* table is searched for a row with that value of *EmployeePOID*.
5. The *Position* object is read in and instantiated (due to the automatic read indicated in the holds row).
6. The value of the *Employee.position* attribute is set to reference the *Position* object.

Now let's consider how the objects would be saved to the database. Because the relationship is to be automatically traversed, and to maintain referential integrity, a **transaction** is created. The next step is to add update statements for each object to the transaction. Each update statement includes both the business attributes and the key values mapped in **Figure 13**. Because relationships are implemented via foreign keys, and because those values are being updated, the relationship is effectively being persisted. The transaction is submitted to the database and run (see **Introduction to Transaction Control** for details).

There is one annoyance with the way the holds relationship has been mapped into the database. Although the direction of this relationship is from *Employee* to *Position* within the object schema, it's been implemented from *Position* to *Employee* in the database. This isn't a big deal, but it is annoying. In the data schema you can implement the foreign key in either table and it wouldn't make a difference, so from a data point of view when everything else is equal you could toss a coin. Had there been a potential requirement for the holds relationship to turn into a one-to-many relationship, something that a **change case** would indicate, then you would be motivated to implement the foreign key to reflect this potential requirement. For example, the existing data model would support an employee holding many positions. However, had the object schema been taken into account, and if there were no future requirements motivating you to model it other wise, it would have been cleaner to implement the foreign key in the *Employee* table instead.

### 3.4.2 One-To-Many Mappings

Now let's consider the *works in* relationship between *Employee* and *Division* in **Figure 11**. This is a one-to-many relationship – an employee works in one division and a single division has many employees working in it. As you can see in **Figure 13** an interesting thing about this relationship is that it should be automatically traversed from *Employee* to *Division*, something often referred to as a cascading read, but not in the other direction. Cascading saves and cascading deletes are also possible, something covered in the discussion of **referential integrity**.

When an employee is read into memory the relationship is automatically traversed to read in the division that they work in. Because you don't want several copies of the same division, for example if you have ten employee objects that all work for the IT division you want them to refer to the same IT division object in memory. The implication is that you will need to implement a strategy for doing this, one option is to implement a cache that ensures only one copy of an object exists in memory or to simply have the *Division* class implement it's own collection of instances in memory (effectively a mini-cache). If the application needs to it will read the *Division* object into memory, then it will set the value of *Employee.division* to reference the appropriate *Division* object. Similarly the *Division.addEmployee()* operation will be invoked to add the employee object into its collection.

Saving the relationship works in the same way as it does for one-to-one relationships – when the objects are saved so are their primary and foreign key values so therefore the relationship is automatically saved.

Every example in this article uses foreign keys, such as *Employee.DivisionPOID*, pointing to the primary keys of other tables, in this case *Division.DivisionPOID*. This doesn't have to be the case, sometimes a foreign key can refer to an alternate key. For example, if the *Employee* table of **Figure 12** were to include a *SocialSecurityNumber* column then that would be an alternate key for that table (assuming all employees are American citizens). If this where the case you would have the option to replace the *Position.EmployeePOID* column with *Position.SocialSecurityNumber*.

### 3.4.3 Many-To-Many Mappings

To implement many-to-many relationships you need the concept of an associative table, a data entity whose sole purpose is to maintain the relationship between two or more tables in a relational database. In **Figure 11** there is a many-to-many relationship between *Employee* and *Task*. In the data schema of **Figure 12** I needed to introduce the associative table *EmployeeTask* to implement a many-to-many relationship the *Employee* and *Task* tables. In relational databases the attributes contained in an associative table are traditionally the combination of the keys in the tables involved in the relationship, in the case *EmployeePOID* and *TaskPOID*. 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 *EmployeeTask* over *Assigned*.

Notice the 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 of the relationship within the data schema to preserve overall multiplicity of the original relationship. The original relationship indicated that an employee is assigned to one or more tasks and that a task has zero or more employees assigned to it. In the data schema you see that this is still true even with the associative table in place to maintain the relationship.

Assume that an employee object is in memory and we need a list of all the tasks they have been assigned. The steps that the application would need to go through are:

1. Create a SQL Select statement that joins the *EmployeeTask* and *Task* tables together, choosing all *EmployeeTask* records with the an *EmployeePOID* value the same as the employee we are putting the task list together.
2. The Select statement is run against the database.
3. The data records representing these tasks are marshaled into *Task* objects. Part of this effort includes checking to see if the *Task* object is already in memory. If it is then we may choose to refresh the object with the new data values (this is a **concurrency** issue).
4. The *Employee.addTask()* operation is invoked for each *Task* object to build the collection up.

A similar process would have been followed to read in the employees involved in a given task. To save the relationship, still from the point of view of the *Employee* object, the steps would be:

1. Start a transaction.
2. Add Update statements for any task objects that have changed.
3. Add Insert statements for the *Task* table for any new tasks that you have created.
4. Add Insert statements for the *EmployeeTask* table for the new tasks.
5. Add Delete statements for the *Task* table any tasks that have been deleted. This may not be necessary if the individual object deletions have already occurred.
6. Add Delete statements for the *EmployeeTask* table for any tasks that have been deleted, a step that may not be needed if the individual deletions have already occurred.
7. Add Delete statements for the *EmployeeTask* table for any tasks that are no longer assigned to the employee.
8. Run the transaction.

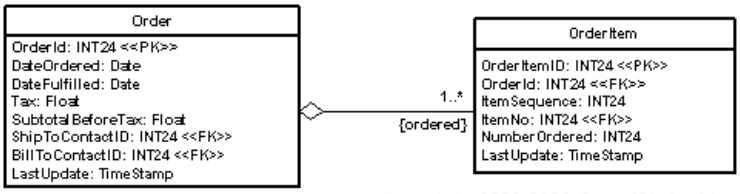
Many-to-many relationships are interesting because of the addition of the associative table. Two business classes are being mapped to three data tables to support this relationship, so there is extra work to do as a result.

## 3.5 Mapping Ordered Collections

**Figure 1** depicted a classic *Order* and *OrderItem* model with an aggregation association between the two classes. An interesting twist is the {ordered} constraint placed on the relationship – users care about the order in which items appear on an order. When mapping this to a relational database you need to add an addition column to track this information. The database schema, also depicted in **Figure 1**, includes the column *OrderItem.ItemSequence* to persist this information. Although this mapping seems straightforward on the surface, there are several issues that you need take into consideration. These issues become apparent when you consider basic persistence functionality for the aggregate:

- **Read the data in the proper sequence**. The scaffolding attribute that implements this relationship must be a collection that enables sequential ordering of references and it must be able to grow as new *OrderItems* are added to the *Order*. In **Figure 2** you see that a Vector is used, a Java collection class that meets these requirements. As you read the order and order items into memory the Vector must be filled in the proper sequence. If the values of the *OrderItem.ItemSequence* column start from 1 and increase by 1 then you can simply use the value of the column as the position to insert order items into the collection. When this isn't the case you must include an ORDER BY clause in the SQL statement submitted to the database to ensure that the rows appear in order in the result set.
- **Don't include the sequence number in the key**. You have an order with five order items in memory and they have been saved into the database. You now insert a new order item in between the second and third order items, giving you a total of six order items. With the current data schema of **Figure 1** you have to renumber the sequence numbers for every order item that appears after the new order item and then write out all them even though nothing has changed other than the sequence number in the other order items. Because the sequence number is part of the primary key of the *OrderItem* table this could be problematic if other tables, not shown in **Figure 1**, refer to rows in *OrderItem* via foreign keys that include *ItemSequence*. A better approach is shown in **Figure 15** where the *OrderItemID* column is used as the primary key.
- **When do you update sequence numbers after rearranging the order items?** Whenever you rearrange order items on an order, perhaps you moved the fourth order item to be the second one on the order, you need to update the sequence numbers within the database. You may decide to cache these changes in memory until you decide to write out the entire order, although this runs the risk that the proper sequence won't be saved in the event of a power outage.
- **Do you update sequence numbers after deleting an order item?** If you delete the fifth of six order items do you want to update the sequence number for what is now the fifth item or do you want to leave it as it. The sequence numbers still work – the values are 1, 2, 3, 4, 6 – but you can no longer use them as the position indicators within your collection without leaving a hole in the fifth position.
- **Consider sequence number gaps greater than one.** Instead of assigning sequence numbers along the lines of 1, 2, 3, … instead assign numbers such as 10, 20, 30 and so on. That way you don't need to update the values of the *OrderItem.ItemSequence* column every time you rearrange order items because you can assign a sequence number of 15 when you move something between 10 and 20. You will need to change the values every so often, for example after several rearrangements you may find yourself in the position of trying to insert something between 17 and 18. Larger gaps help to avoid this (e.g. 50, 100, 150, …) but you'll never completely avoid this problem.

**Figure 15. Improved data schema for persisting Order and OrderItem.**

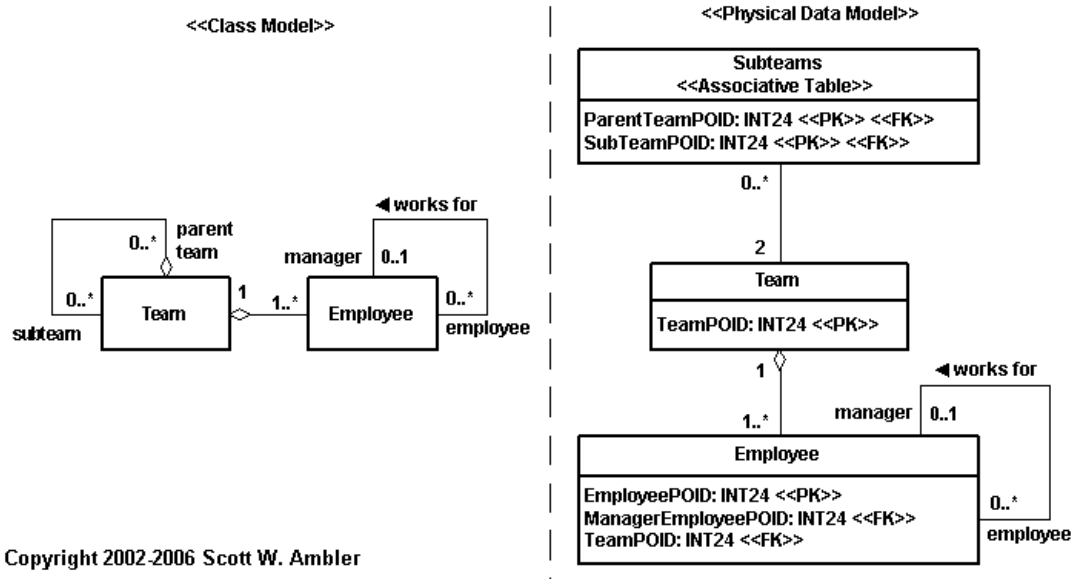


### 3.6 Mapping Recursive Relationships

A recursive relationship, also called reflexive relationships (**Reed 2002**; **Larman 2002**), is one where the same entity (class, data entity, table, …) is involved with both ends of the relationship. For example the *manages* relationship in **Figure 16** is recursive, representing the concept that an employee may manage several other employees. The aggregate relationship that the *Team* class has with itself is recursive – a team may be a part of one or more other teams.

**Figure 16** depicts a class model that includes two recursive relationships and the resulting data model that it would be mapped to. For the sake of simplicity the class model includes only the classes and their relationships and the data model includes only the keys. The **many-to-many** recursive aggregation is mapped to the Subteams associative table in the same way that you would map a normal many-to-many relationship – the only difference is that both columns are foreign keys into the same table. Similarly the **one-to-many** *manages* association is mapped in the same way that you would map a normal one-to-many relationship, the *ManagerEmployeePOID* column refers to another row in the *Employee* table where the manager's data is stored.

**Figure 16. Mapping recursive relationships.**




## 4. Mapping Class-Scope Properties

Sometimes a class will implement a property that is applicable to all of its instances and not just single instances. The *Customer* class of **Figure 17** implements *nextCustomerNumber*, a class attribute (you know this because it's underlined) which stores the value of the next customer number to be assigned to a new customer object. Because there is one value for this attribute for the class, not one value per object, we need to map it in a different manner. **Table 2** summarizes the four basic strategies for mapping class scope properties.

**Figure 17. Mapping class scope attributes.**

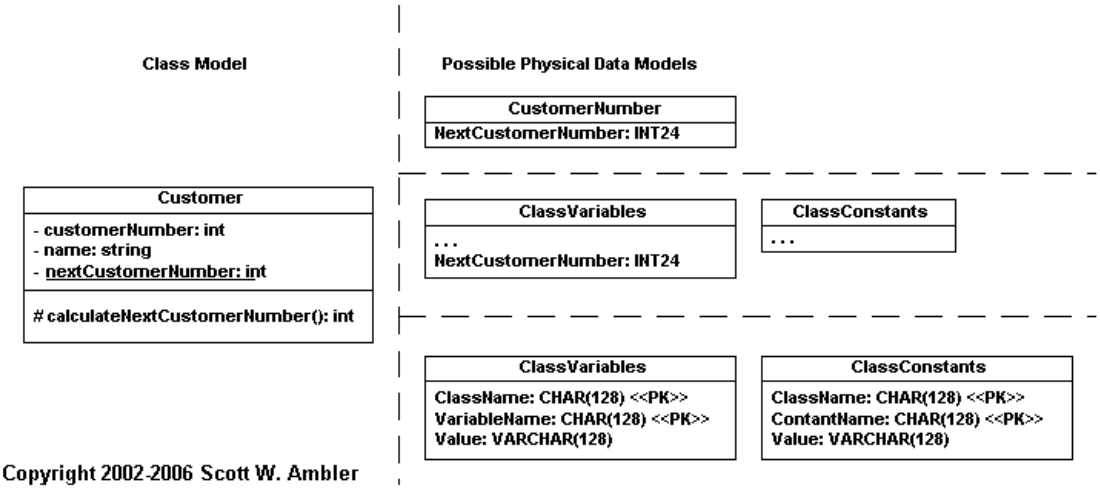



**Table 2. Strategies for mapping class scope properties.**

| Strategy | Example | Advantages | Disadvantages |
|---|---|---|---|
| Single Column, Single-Row Table | The *CustomerNumber* table of **Figure 17** implements this strategy. | Simple<br><br>Fast access | Could result in many small tables |
| Multi-Column, Single-Row Table for a Single Class | If *Customer* implemented a second class scope attribute then a *CustomerValues* table could be introduced with one | Simple<br><br>Fast access | Could result in many small tables, although fewer than the single column approach |

| | | | |
|---|---|---|---|
| | column for each attribute. | | |
| Multi-Column, Single-Row Table for all Classes | The topmost version of the *ClassVariables* table in **Figure 17**. This table contains one column for each class attribute within your application, so if the *Employee* class had a *nextEmployeeNumber* class attribute then there would be a column for this as well. | Minimal number of tables introduced to your data schema. | Potential for concurrency problems if many classes need to access the data at once. One solution is to introduce a *ClassConstants* table, as shown in **Figure 17**, to separate attributes that are read only from those that can be updated. |
| Multi-Row Generic Schema for all Classes | The bottommost version of the *ClassVariables* and *ClassConstants* tables of **Figure 17**. The table contains one row for each class scope property in your system. | Minimal number of tables introduced to your data schema.<br><br>Reduces concurrency problems (assuming your database supports row-based locking). | Need to convert between types (e.g. *CustomerNumber* is an integer but is stored as character data).<br><br>The data schema is coupled to the names of your classes and their class scope properties. You could avoid this with an even more generic schema along the lines of **Figure 9**. |

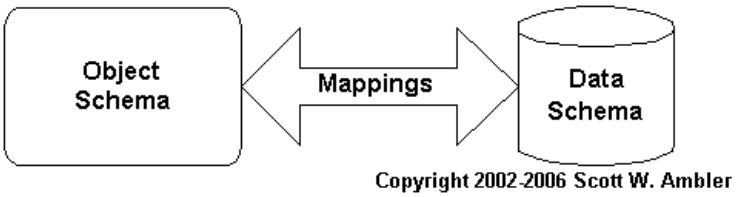
## 5. Performance Tuning

One of the most valuable services that an Agile DBA can perform on a development team is performance tuning. A very good book is ***Database Tuning*** by Shasha and Bonnet (2003). When working with structured technology most of the performance tuning effort was database-oriented, generally falling into one of two categories:

1. **Database performance tuning**. This effort focuses on changing the database schema itself, often by **denormalizing** portions of it. Other techniques include changing the types of key columns, for example an index is typically more effective when it is based on numeric columns instead of character columns; reducing the number of columns that make up a composite key; or introducing indices on a table to support common joins.
2. **Data access performance tuning**. This effort focuses on improving the way that data is accessed. Common techniques include the introduction of stored procedures to "crunch" data in the database server to reduce the result set transmitted across the network; reworking SQL queries to reflect database features; clustering data to reflect common access needs; and caching data within your application to reduce the number of accesses. In fact, although I haven't presented an example in this article, a common strategy is to map an attribute of a class to a stored function. For example, you could map the *Customer.totalPortfolio* to the *calculateCustomerPortfolio()* stored procedure. Granted, this may introduce performance problems itself (do you really want this stored function to be invoked each time you read in a customer object?) and instead you might want to map *Customer.totalPortfolio* attribute to the *Customer.TotalPortfolio* column which would be calculated via a trigger (or in batch).

Neither of these needs go away with object technology, although as **Figure 18** implies the situation is a little more complicated. An important thing to remember is that your object schema also has structure to it, therefore changes to your object schema can affect the database access code that is generated based on the mappings to your database. For example, assume that the *Employee* class has a *homePhoneNumber* attribute. A new feature requires you to implement phone number specific behavior (e.g. your application can call people at home). You decide to refactor *homePhoneNumber* into its class, and example of **third normal object form (3ONF)**, and therefore update your mappings to reflect this change. Performance degrades as a result of this change, motivating you to change either your mappings which the data access paths or the database schema itself. The implication is that a change to your object source code could motivate a change to your database schema. Sometimes the reverse happens as well. This is perfectly fine, because as an agile software developer you are used to working in an evolutionary manner.

**Figure 18. Performance tuning opportunities.**




There are two main additions to performance tuning that you need to be aware of: **mapping tuning** and object schema tuning. Mapping tuning is described below. When it comes to object schema tuning most changes to your schema will be covered by common **refactorings**. However, a technique called **lazy reading** can help dramatically.

### 5.1 Tuning Your Mappings

Throughout this article you have seen that there is more than one way to map object schemas to data schemas – there are four ways to map **inheritance structures**, two ways to map a **one-to-one relationship** (depending on where you put the foreign key), and four ways to map **class-scope properties**. Because you have mapping choices, and because each mapping choice has its advantages and disadvantages, there are opportunities to improve the data access performance of your application by changing your choice of mapping. Perhaps you implemented the **one table per class** approach to mapping inheritance only to discover that it's too slow, motivating you to refactor it to use the **one table per hierarchy** approach.

It is important to understand that whenever you change a mapping strategy that it will require you to change either your object schema, your data schema, or both.

### 5.2 Lazy Reads

An important performance consideration is whether the attribute should be automatically read in when the object is retrieved. When an attribute is very large, for example the picture of a person could be 100k whereas the rest of the attributes are less than 1k, and rarely accessed you may want to consider taking a lazy read approach. The basic idea is that instead of automatically bringing the attribute across the network when the object is read you instead retrieve it only when the attribute is actually needed. This can be accomplished by a getter method, an operation whose purpose is to provide the value of a single attribute, that checks to see if the attribute has been initialized and if not retrieves it from the database at that point.

Other common uses for lazy read is **reporting** and for retrieving objects as the results of **searches** where you only need a small subset of the data of an object.

## 6. Implementation Impact On Your Objects

The **O/R impedance mismatch** forces you to map your object schema to your data schema. To implement these mappings you will need to add code to your business objects, code that impacts your application. These impacts are the primary fodder for the argument that object purists make against using object and relational technology together. Although I wish the situation were different, the reality is that we're using object and relational technology together and very likely will for many years to come. Like it or not we need to accept this fact.

I think that there is significant value in summarizing how mapping impacts your objects. Some of this material you have seen in this article and some you will see in other chapters. The impacts on your code include the need to:

- Maintain **shadow information**.
- **Refactor** it to improve overall performance.
- Work with **legacy data**. It is common to work with legacy databases and that there are often significant data quality, design, and architectural problems associated with them. The implication is that you often need to map your objects to legacy databases and that your objects may need to implement integration and data cleansing code to do so.
- **Encapsulate database access**. Your strategy for encapsulating database access determines how you will implement your mappings. Your objects will be impacted by your chosen strategy, anywhere from including embedded SQL code to implementing a common interface that a persistence framework requires.
- Implement **concurrency control**. Because most applications are multi-user, and because most databases are accessed by several applications, you run the risk that two different processes will try to modify the same data simultaneously. Therefore your objects need to implement concurrency control strategies that overcome these challenges.
- **Retrieve objects from a relational database**. You will want to work with collections of the same types of objects at once, perhaps you want to list all of the employees in a single division.
- **Implement referential integrity**. There are several strategies for implementing referential integrity between objects and within databases. Although referential integrity is a business issue, and therefore should be implemented within your business objects, the reality is that many if not all referential integrity rules are implemented in the database instead.
- **Implement security access control**. Different people have different access to information. As a result you need to implement security access control logic within your objects and your database.
- **Implement reporting**. Do your business objects implement basic reporting functionality or do you leave this effort solely to reporting tools that go directly against your database. Or do you use a combination.
- **Implement object caches**. Object caches can be used to improve application performance and to ensure that objects are unique within memory.

## 7. Implications for Model Driven Architecture (MDA)

The **Model-Driven Architecture (MDA)** defines an approach to modeling that separates the specification of system functionality from the specification of its implementation on a specific technology platform. In short, it defines guidelines for structuring specifications expressed as models. The MDA promotes an approach where the same model specifying system functionality can be realized on multiple platforms through auxiliary mapping standards, or through point mappings to specific platforms. It also supports the concept of explicitly relating the models of different applications, enabling integration, interoperability and supporting system evolution as platform technologies come and go.

Although the MDA is based on the Unified Modeling Language (UML), and the **UML does not yet officially support a data model**, my expectation is that object to relational mapping will prove to be one of the most important features that MDA-compliant CASE tools will support. My hope is that the members of the OMG find a way to overcome the **cultural impedance mismatch** and start to work with data professionals to bring issues such as UML data modeling and object-to-relational mapping into account. Time will tell.

## 8. Patternizing What You Have Learned

In this article you learned the basics of mapping objects to relational databases (RDBs), including some basic implementation techniques that will be expanded on in following chapters. You saw that there are several strategies for mapping inheritance structures to RDBs and that mapping object relationships into RDBs is straightforward once you understand the differences between the two technologies. Techniques for mapping both instance attributes and class attributes were presented, providing you with strategies to complete map a class's attributes into an RDB.

This article included some methodology discussions that described how mapping is one task in the iterative and incremental approach that is typical of agile software development. A related concept is that it is a fundamental mistake to allow your existing database schemas or **data models to drive the development of your object models**. Look at them, treat them as constraints, but don't let them negatively impact your design if you can avoid it.

Throughout this article I have described mapping techniques in common prose, some authors choose to write patterns instead. The first such effort was the **Crossing Chasms pattern language** and the latest effort is captured in the book **Patterns of Enterprise Application Architecture**. **Table 3** summarizes the critical material presented in this article as patterns, using the names suggested by other authors wherever possible.

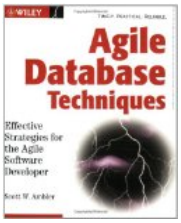
**Table 3. Mapping patterns.**

| Pattern | Description |
|---|---|
| **Class Table** | Map each individual class within an inheritance hierarchy to its own table. |

| | |
|---|---|
| **Inheritance** | |
| **Concrete Table Inheritance** | Map the concrete classes of an inheritance hierarchy to its own table. |
| **Foreign Key Mapping** | A relationship between objects is implemented in a relational database as foreign keys in tables. |
| **Identity Field** | Maintain the primary key of an object as an attribute. This is an example of Shadow Information. |
| **Lazy Initialization** | Read a high-overhead attribute, such as a picture, into memory when you first access it, not when you initially read the object into memory. |
| **Lazy Read** | Read an object into memory only when you require it. |
| **Legacy Data Constraint** | Legacy data sources are a constraint on your object schema but they should not drive its definition. |
| **Map Similar Types** | Use similar types in your classes and tables. For example it is easier to map an integer to an numeric column than it is to map it to a character-based column. |
| **Map Simple Property to Single Column** | Prefer to map the property of an object, such as the total of an order or the first name of an employee, to a single database column. |
| **Mapping-Based Performance Tuning** | To improve overall data access performance you can change your object schema, your data schema, or the mappings in between the two. |
| **Recursive Relationships Are Nothing Special** | Map a recursive relationship exactly the same way that you would map a non-recursive relationship. |
| **Representing Objects as Tables** | Prefer to map a single class to a single table but be prepared to evolve your design based to improve performance. |
| **Separate Tables for Class-Scope Properties** | Introduce separate tables to store class scope properties. |
| **Shadow Information** | Classes will need to maintain attributes to store the values of database keys (see Identity Field) and concurrency columns to persist themselves. |
| **Single Column Surrogate Keys** | The easiest key strategy that you can adopt within your database is to give all tables a single column, surrogate key that has a globally unique value. |
| **Single Table Inheritance** | Map all the classes of an inheritance hierarchy to a single table. |
| **Table Design Time** | Let your object schema form the basis from which you develop your data schema but be prepared to iterate your design in an evolutionary manner. |
| **Uni-directional Key Choice** | When a one-to-one unidirectional association exists from class A to class B, put the foreign key that maintains the relationship in the table corresponding to class A. |

## 9. References and Suggested Online Readings

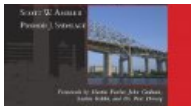

- At **www.ambysoft.com/essays/mappingObjects.html** I maintain a list of links to mapping white papers posted on the web.
- **Adopting Agile/Evolutionary Database Techniques**
- **Agile Database Best Practices**
- **Agile/Evolutionary Data Modeling**
- **Choosing a Primary Key: Natural or Composite?**
- **A Classification of Object-Relational Impedance Mismatch** (Ireland, Bowers, Newton & Waugh)
- **The Cultural Impedance Mismatch Between Data Professionals and Application Developers**
- **Introduction to Data Normalization**
- Mapping Objects to Relational Databases
- **On Relational Theory**
- **SQL Tutorial**
- **Survey Results (Agile and Data Management)**
- **When is Enough Modeling Enough?**
- **Why Data Models Don't Drive Object Models (And Vice Versa)**

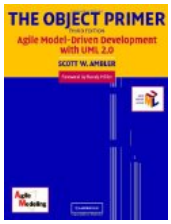

This book describes the philosophies and skills required for developers and database administrators to work together effectively on project teams following evolutionary software processes such as Extreme Programming (XP), the **Rational Unified Process (RUP)**, the **Agile Unified Process (AUP)**, Feature Driven Development (FDD), Dynamic System Development Method (DSDM), or **The Enterprise Unified Process (EUP)**. In March 2004 it won a Jolt Productivity award.


This book presents a collection of architectural patterns, many of which hit on persistence-related issues. I highly suggest this book as a complement to the material presented in this article.


This book describes, in detail, how to **refactor a database schema** to improve its design. The first section of the book overviews the fundamentals evolutionary database techniques in general and of database refactoring in detail. More importantly it presents strategies for implementing and deploying database refactorings, in the context of both "simple" single application databases and in "complex" multi-application databases. The second section, the majority of the book, is a **database refactoring reference catalog**. It describes over 60 database refactorings,



presenting data models overviewing each refactoring and the code to implement it.




This book presents a full-lifecycle, **agile model driven development (AMDD)** approach to software development. It is one of the few books which covers both object-oriented and data-oriented development in a comprehensive and coherent manner. Techniques the book covers include **Agile Modeling (AM)**, **Full Lifecycle Object-Oriented Testing (FLOOT)**, over 30 **modeling techniques**, **agile database techniques**, **refactoring**, and **test driven development (TDD)**. If you want to gain the skills required to build mission-critical applications in an agile manner, this is the book for you.

### Let Us Help

We actively work with clients around the world to improve their information technology (IT) practices, typically in the role of mentor/coach, team lead, or trainer. A full description of what we do, and how to contact us, can be found at **Scott W. Ambler + Associates**.

---

Disciplined Agile Delivery | Agile Modeling | Agile Data | Enterprise Unified Process | Agile Unified Process | Ambysoft | The Software Development Practice Advisor

SCOTT AMBLER + Associates | @scottwambler

---



This site owned by **Ambysoft Inc.**