CHAPTER 3

Comparing the Object and Relational Data Models

When people talk about databases, they almost always mean relational databases. This wasn't always the case, though, as databases existed before the relational data model was developed. Now, the case for considering alternatives has become stronger with the increasing dominance of object-oriented languages in a widening range of application areas, and with the emergence of native object databases such as db4o. It's vital to understand the benefits and limitations of today's data models, particularly in the context of object-oriented systems.

This chapter describes the evolution of data models, including the relational and the object-based "post-relational" models, and compares their characteristics, both good and bad. It discusses the strategies needed to make the relational model work with object-oriented systems. It then describes how object databases reflect the features expected in relational databases. The next chapter will examine the object data model in detail, and explain how this model is applied in db4o.

Data Models

Before the first DBMS was developed, programs accessed data from flat files. These did not allow representation of logical data relationships or enforcement of data integrity. Data modeling has developed over successive generations since the 1960s to provide applications with more powerful data storage features. In this chapter we will look at the differences between data models, concentrating on the relational model and the object model.

Generally speaking, data models have evolved in three generations. The early generation data models tend to refuse to completely go away, however. After all, companies often have made significant investments in databases, or have critical data dependent on them. Even some first-generation products are still in use and are still supported by their vendors.

Yet so far, the most commercially successful databases have certainly been those using the relational model, which is considered to be the second generation. Relational databases are definitely not about to disappear, and their dominance in the marketplace has made it very difficult for a new generation of databases to gain a foothold.

However, the application language world has changed. With the continuing evolution of Java and Microsoft's commitment to .NET, the developer's choice in many application areas is no longer between object-oriented and non-object-oriented; it is now a choice of which object-oriented platform (Java or .NET). The mismatch between the relational data model and the object-oriented application model puts new object databases like db4o in a strong position to offer a real alternative.

First Generation

The emergence of computer systems in the 1960s led to the development of the hierarchical and network data models, which are usually referred to as the first-generation data models. These models are described in this section.

Hierarchical Data Model

Most computer users are very familiar with a hierarchical way of storing information. The file system used by most personal computer operating systems is an example of a hierarchy, and accordingly this is known as the *hierarchical data model*. It represents directories and files as a system of trees. Typically a file system has the following characteristics:

- It allows one-to-one or one-to-many relationships between entities—a directory or folder may contain one file, or it may contain many files (or it may contain no files).
- An entity at a "many" end of a relationship can be related to only one entity at the "one" end—in other words, a file can only be in one directory.

This storage model is often described as *navigational*. This means that to find one particular item, you have to navigate your way down through the hierarchy using predefined relationships until you reach it. This can be very efficient if the searches you want to do follow these relationships closely. However, it can be very inefficient if you want to query your data in an ad hoc way—for instance looking at data from a user's point of view in ways that the database designer could not anticipate. If the information you want to find in a file system is contained in several files in different directories, then the process of gathering it can be very laborious.

Figure 3-1 shows an example of hierarchical data. Each `customer` can have many `orders`, and each order can contain many `items`. Each `item` belongs to a particular `order`, and only to that `order`.

Hierarchical databases, which were the first generation of databases, store their data pretty much like this. The best-known hierarchical database is IBM's Information Management System (IMS), which has been around since the 1960s and is still supported by IBM.

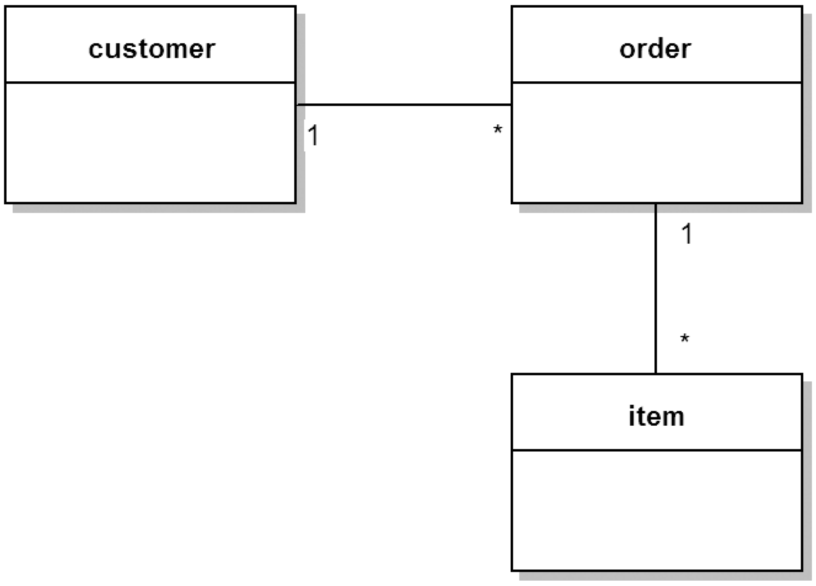

Figure 3-1. *Hierarchical data example*

Network Data Model

The network data model standard was developed in the late 1960s by the Committee on Data Systems Languages (CODASYL), the same organization that developed Cobol. It added one important feature for data modeling. Multiple parentage means that a single entity can be at the "many" ends of multiple relationships. This is a bit harder to imagine in terms of a file system: it would mean that a file could be in more than one directory at the same time.

In many scenarios, the network model allowed data to be modeled more realistically. In the example in Figure 3-1, it probably doesn't make sense to have every `item` as a separate entity that belongs to a specific `order`. What if there are several `orders` for the same type of `item`? A better way of modeling this data is shown in Figure 3.2. Now, the `order` is made up of `orderlines`, each of which could refer to an `item` and the required quantity of that `item`. Each `item` can appear in many different `orderlines`. `orderline` is at the "many" end of its relationships to `item` and `order`, which is allowed by the network model.

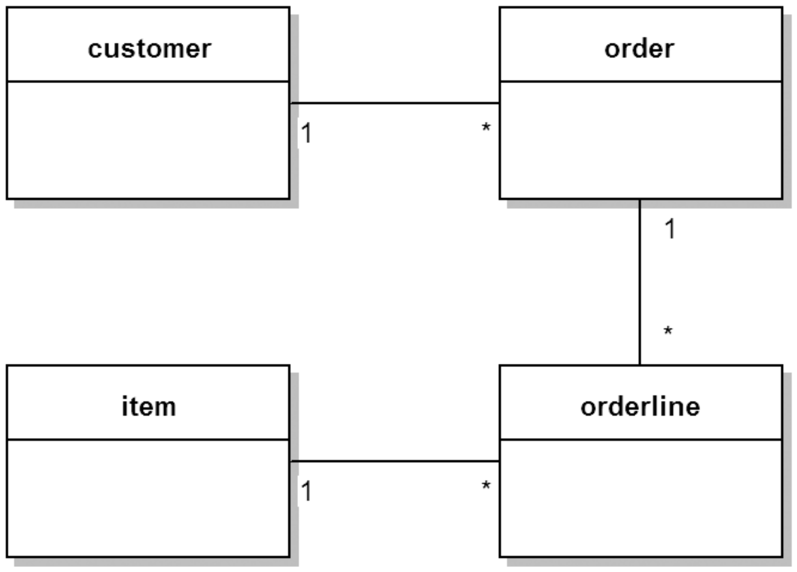

Figure 3-2. *Network data example*

The network model gave an extra degree of flexibility in data modeling, but it was still a navigational model. The network defines a set of relationships, and you have to follow them. Relationships are directional. For example, the relationship between `customer` and `order` in Figure 3-2 might have been defined to allow you to find the `item` for a particular `orderline`. If you needed to find all the `orderlines` that contain a particular `item`, you may have to change the data model itself to allow this new query to be performed.

The CODASYL query language had statements that allowed the user to jump from one data element to the next, through a graph of pointers among these elements. These queries were quite difficult to write, even for very simple queries. Listing 3-1 shows an example of a CODASYL query.

Listing 3-1. *CODASYL Query to Find How Much Wine Carl Has Ordered*

```
NAME := "Carl"
FIND CUSTOMERS RECORD USING CALC-KEY
LOOP: repeat forever
    FIND NEXT ORDERS RECORD IN CURRENT CUSTORD SET
    if FAIL then break LOOP
    FIND OWNER OF CURRENT ITEMORD SET
    GET ITEMS; ITYPE
    if ITEMS.ITYPE = "Wine" then do
        FIND CURRENT OF ORDERS RECORD
        GET ORDERS; QUANTITY
        print QUANTITY
        break LOOP
    end
end LOOP
```

As we have seen, the first-generation models were not suited to ad hoc queries, where you don't necessarily know how the data will need to be retrieved before you create the database. A criticism that is often made of object databases is that they are little more than a rehash of the old network databases. It is true that the object data model is also essentially navigational, and is also not well suited to ad hoc queries. However, as you will see, it is substantially different in other ways, and offers its own advantages, primarily the fact that it is a good match to the object model used in modern application design.

Second Generation: The Relational Model

The relational model has undoubtedly been the most widely used and commercially successful way to date of modeling data. Its characteristics are very different from the earlier models. To begin, data entities are represented by simple tabular structures, known as relations. Entity relationships and data integrity are defined by primary keys and foreign keys. The design of a relational database is based on the idea of normalization, the process of removing redundant data from your tables in order to improve storage efficiency, data integrity, and scalability.

■ **Note** The relational model was proposed in 1970 by Edgar Codd to represent the natural structure of data without the database user needing to know about the machine representation. It promoted the idea of end-user programming and interactive querying of a database. This was a big step forward in the usability of databases.

Data Access in the Relational Model

Data access uses a high-level nonprocedural language (SQL). This makes relational databases great for ad hoc querying. If the data you require is in the database, you will almost certainly be able to write a SQL query that retrieves it, though it may involve joining many tables to get the data. The query in Listing 3-1 becomes much more natural and understandable in SQL, as shown in Listing 3-2.

Listing 3-2. *SQL Query to Find How Much Wine Carl Has Ordered*

```
select QUANTITY
    from ORDERS
    where NAME='Carl' and ITEMTYPE='Wine'
```

SQL has evolved as a standard that is supported by most commercial database products, although those products usually add their own proprietary extensions to the language. The way queries are constructed is based on the branch of mathematics known as set theory, although you don't need to know the details of set theory to write SQL. To some people, the basis on formal mathematical principles is a major advantage of the relational model, although it is not entirely obvious why this should be of such great benefit. There are no particular foundations in mathematical theory for object-oriented programming languages, yet they are successful because they provide tools to get the job done well.

Relational queries are not navigational. You don't have to follow predefined paths. You can write a SQL query to select data from any table in a database. If you want to get data from more than one type of entity, you can write your queries to join tables. Usually, you join tables that have foreign key relationships or at least common field names, so that associated data from different tables can be assembled. There is no direction implied in a join, so there is no concept of navigating from one table to another.

Using the Relational Model with Procedural Programs

Relational databases are a great fit for procedural programs. You use entity relationship modeling to design your database schema, and then write procedural code that gets some data, does something with it, and stores the data. The application is heavily dependent on the database design. You can connect to that database from an application written in any language for which you can get a suitable database driver. The same database can be the target for many different applications.

An application will not be written in SQL itself. That's because SQL is a declarative language, not a programming language. It is not computationally complete, so you can't write a full program with it. Its job is to express queries and perform some manipulation of the data in the database. As a result, SQL is either used at an interactive prompt or, more commonly, appears as strings embedded within another language. Of course, many relational database systems have the ability to use stored procedures, which are program modules that exist within the DBMS. Even though these are within the DBMS, they still need to combine SQL with another language, such as Oracle's PL/SQL.

The use of embedded SQL strings in application code external to the database can present a problem to the programmer. The compiler simply interprets these as strings—if it's a valid string then it's okay as far as the compiler is concerned. It might be complete nonsense as far as the database is concerned: it may refer to tables or columns that don't exist, for example. You have to wait until runtime to find that out, and debugging can be difficult.

Using the Relational Model with Object-Oriented Programs

As you learned in Chapter 2, object-oriented systems try to model the problem domain in terms of objects. The entities in the system are described in the class diagram, rather than in the *entity relationship diagram*, or ERD, of the traditional system. Suddenly, the database is not the driver for the application—instead, its primary function is to provide a service to the application, namely the capability to persist objects.

This shift is significant because of the differences between the object model and the relational data model. A class diagram and an ERD may look superficially similar, but you can find many examples of relationships in the former that are hard to model in the latter. For example, inheritance is not directly supported in the relational model.

■ **Note** There are some significant variations between RDBMSs in their features, including their support for object orientation. The PostgreSQL RDBMS now features table inheritance, which supports polymorphism in queries. This is not, however, a standard RDBMS capability.

One result of this is often seen when object-oriented systems are created within a "data culture." The object model is reined in so that the class diagram matches a database designer's lovingly crafted ERD. This rarely produces a good object model.

The problem of embedded SQL strings also applies to object-oriented programs, perhaps more so as tools that support safe and easy refactoring cannot understand or modify them.

Third Generation: "Post-Relational" Models

The third-generation models, first proposed in the 1980s, are a response to the problems that often arise when marrying an object-oriented system to a relational database. They are sometimes described as "post-relational," although it is more realistic to consider them as coexisting with the relational model and providing additional options for developers. Unlike the second generation, where the relational model was pretty much universally adopted, with some proprietary variations, the object-oriented third generation has evolved in two distinct directions, which are described in this section.

The Object Data Model

The object data model is pretty much the same as the object-oriented paradigm described in Chapter 2, except that the objects are persistent. That is, they continue to exist after the program run finishes.

An object in an object database is analogous to an object in application memory. In most object databases, there are language bindings that allow you to use the persistent objects in applications. A Java application requires a Java language binding, and so on. The database schema itself is created using an object definition language, which defines the object classes that can be stored, and their relationships.

On the other hand, a native object database like db4o stores objects exactly as they are created in the application. Native object databases don't need an object definition language—the database schema is identical to the object domain model of the application. They are, therefore, closely tied to the applications that use them.

ORIGINS OF THE OBJECT DATABASE

The object database emerged from the "manifesto" written by Malcolm Atkinson and others in 1989, which specified a list of requirements that should be met by an object database, including some object-oriented features and some database-like features. The Object Database Management Group (ODMG) tried to establish standards for object databases, including ODL, an object definition language, and OQL, an object query language. These standards have been adopted to some extent, but have never achieved universal acceptance.

The final version of the ODMG standard, ODMG 3.0, was released in 2001, and ODMG disbanded after that. The ODMG Java language binding was the basis for Java Data Objects (JDO), an API for transparent persistence. JDO is not a database or data model: it is a persistence API that can be used with a variety of data stores, including relational databases.

db4o doesn't use the ODMG standards because it doesn't need to. Since it is a native database, it doesn't need an object definition language, and the native query capabilities it offers are more advanced than OQL. Similarly, although it is quite possible for a native object database to offer a JDO-compliant API for Java developers, db4o doesn't do this. The db4o API is simpler and more intuitive, and unlike JDO, supports .NET as well as Java.

The object data model is navigational—object access follows well-defined relationships as specified in the design model. It is not just the network model revisited, though. The key benefits come from the close match to application languages and the elimination of the impedance mismatch—fewer lines of code are needed for data access, and database configuration is reduced or eliminated. The result is greater developer productivity. Performance can also be greatly enhanced for queries that follow the defined relationships.

The Object-Relational Model

At roughly the same time that the object data model was being proposed, the problem of storing objects in databases was being approached from a different angle. The object-relational model extends the capabilities of the relational model to allow objects to be stored in the columns of a relational database. An object relational DBMS is sometimes referred to as a *hybrid* DBMS.

■ **Note** The hybrid DBMS approach was proposed by Michael Stonebraker in 1990, and has been implemented in some commercial RDBMSs, including Oracle. The motivation was a desire for databases that could store more complex entities and rules but that retained all the strengths of the second-generation databases.

The object-relational data model is an extension of the relational model, with the following features:

- A field may contain an object with attributes and operations.
- Complex objects can be stored in relational tables

For example, Oracle supports the ability to declare a new data type that is basically a class, and to set this new type as the data type for a table column.

The object-relational data approach is pretty much the opposite of what object databases, particularly native ones, are trying to achieve. Instead of giving persistence capability to an object-oriented system, it provides object-oriented capabilities within the database. This approach is certainly not suitable for the embedded applications for which db4o excels as you need a full-blown RDBMS to make use of it.

The use of terms like “third generation” and “post-relational” has not been particularly accurate—they implied that these types of databases would replace relational databases, the way that the second generation effectively supplanted the first. The third generation has not taken the place of relational databases, but instead exists in parallel to widen the options open to developers to use appropriate solutions for their own applications. db4o, with its small footprint and zero administration capability, further opens up the choices within the third generation to a new range of applications.

Fitting Objects into a Relational Database

If the design of an object-oriented system is application driven rather than data driven, then the database needs to provide a way to persist the objects in the system. If the database is relational, there needs to be a mapping of objects to database tables. This requirement is there regardless of the mechanism used to allow the application to communicate with the database (either hard-coded SQL within the application or a mapping layer such as Hibernate).

This can sometimes be a straightforward matter of mapping individual classes to separate database tables. However, if the class structure is more complex, then the mapping must be carefully considered to allow data to be represented and accessed as efficiently as possible. Looking at object-relational mapping strategies is a good way to understand the practical differences between the object and relational models. Let's take a look at some of the common object relationships that you might find in a class diagram, as described in Chapter 2. Remember that the data model that you would use in each case for an object database would be very similar, or identical in the case of a native object database, to the class diagram.

Aggregation

In an aggregation relationship, as shown in Figure 3-3, the owner object holds a reference to its owned objects, which are either single objects or collections. An owned relationship like this is implemented in the relational model using a foreign key column in the table on the many side of the relationship, as shown in Figure 3-4. You need to include foreign key columns in your data model that are not required in the object model.

■ **Note** Relational databases rely on primary keys to ensure that each row in a table is unique. Any unique field or combination of fields can be used as the primary key. In the object model, each object is unique already, and no key field is needed. In an object database, each object has a unique UID assigned to it automatically. This does mean that you can create objects that have identical field values but that are different objects—it is up to the application logic to enforce unique values if this is required.

Association is modeled in exactly the same way in the relational model, which does not distinguish between ownership and simple reference.

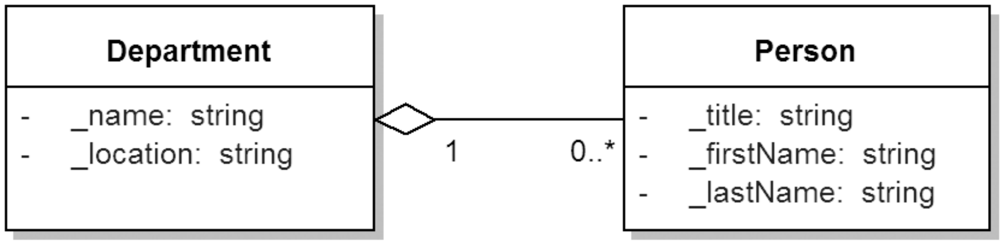

Figure 3-3. *Aggregation object model*

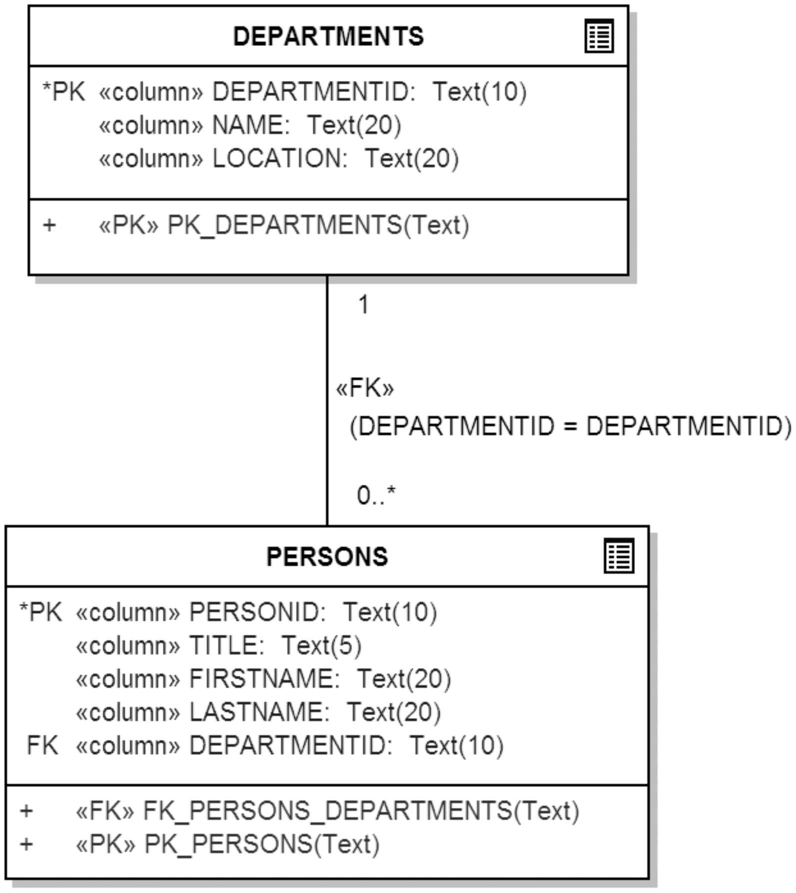
DEPARTMENTS
*PK «column» DEPARTMENTID: Text(10)
«column» NAME: Text(20)
«column» LOCATION: Text(20)
+ «PK» PK_DEPARTMENTS(Text)
1
«FK»
(DEPARTMENTID = DEPARTMENTID)
0..*
PERSONS
*PK «column» PERSONID: Text(10)
«column» TITLE: Text(5)
«column» FIRSTNAME: Text(20)
«column» LASTNAME: Text(20)
FK «column» DEPARTMENTID: Text(10)
+ «FK» FK_PERSONS_DEPARTMENTS(Text)
+ «PK» PK_PERSONS(Text)

Figure 3-4. *Aggregation relational model*

■**Note** Associations between objects are implemented by *object references*. The reference type is the class name of the associated object. A common beginner's mistake is to try to associate objects by giving them text or numeric fields with matching names, just like foreign keys in a relational database—very tempting if you have been brought up in the relational way, but not a good idea.

Inheritance

The relational model as used by most RDBMSs has no concept equivalent to inheritance. As a result, the mapping can become quite involved. There are several possible strategies, and there is no single best way to do it. The optimum strategy depends on the precise nature of the inheritance tree. To illustrate, we will map the tree shown in Figure 3-5, which has a simple two-layer hierarchy (things can become much more complicated for deeper hierarchies).

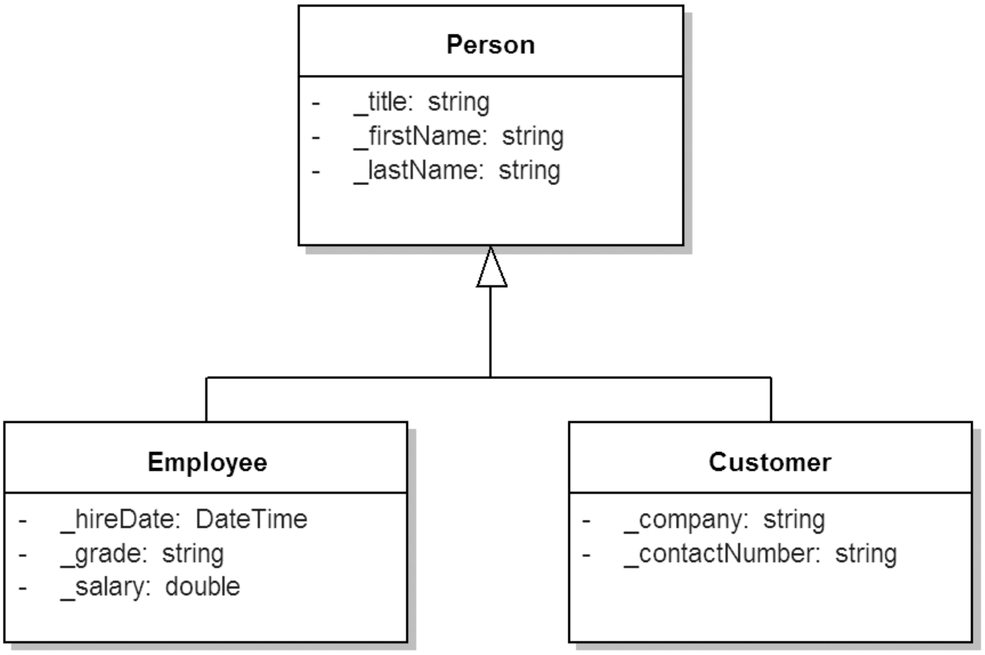

Figure 3-5. *A simple class inheritance tree*

This can be mapped to a relational database in three ways, illustrated by the following examples.

Vertical Mapping: One Table per Class

There is separate table for each class, including any abstract classes, as shown in Figure 3-6. Subclass tables are related by foreign keys to the superclass tables. You need an additional key field (`PersonID`) to create relationships. Creating an `Employee` object in the application involves joining `Person` and `Employee` tables in the database. This approach can result in complex queries in cases with deeper levels of inheritance than this example shows.

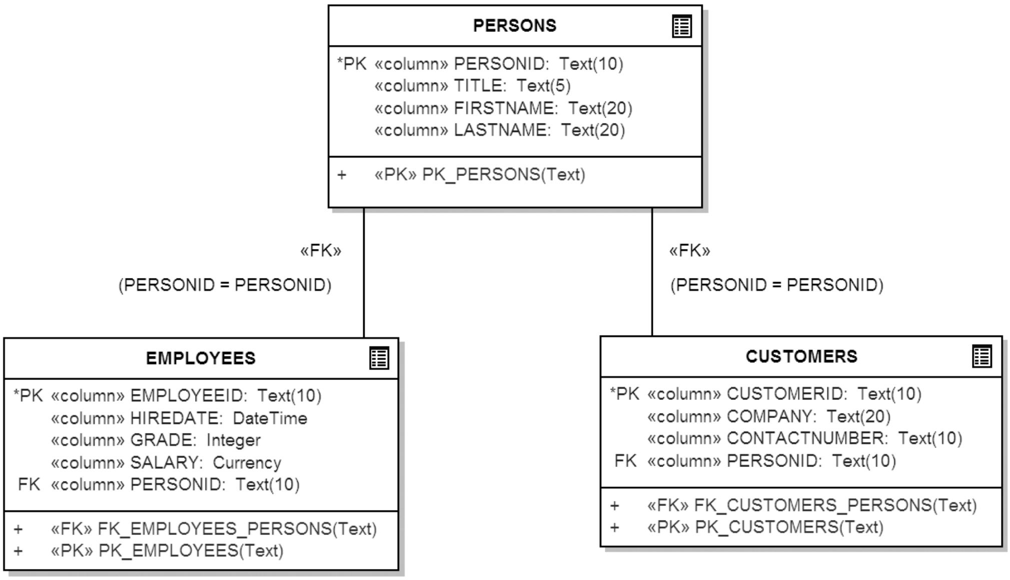
PERSONS
*PK «column» PERSONID: Text(10)
«column» TITLE: Text(5)
«column» FIRSTNAME: Text(20)
«column» LASTNAME: Text(20)
+ «PK» PK_PERSONS(Text)
«FK»
(PERSONID = PERSONID)
«FK»
(PERSONID = PERSONID)
EMPLOYEES
*PK «column» EMPLOYEEID: Text(10)
«column» HIREDATE: DateTime
«column» GRADE: Integer
«column» SALARY: Currency
FK «column» PERSONID: Text(10)
+ «FK» FK_EMPLOYEES_PERSONS(Text)
+ «PK» PK_EMPLOYEES(Text)
CUSTOMERS
*PK «column» CUSTOMERID: Text(10)
«column» COMPANY: Text(20)
«column» CONTACTNUMBER: Text(10)
FK «column» PERSONID: Text(10)
+ «FK» FK_CUSTOMERS_PERSONS(Text)
+ «PK» PK_CUSTOMERS(Text)

Figure 3-6. *Data model using vertical mapping*

Horizontal Mapping: One Table per Concrete Class

In this approach, each concrete class is mapped to a different table, and each table contains columns for all the attributes of its class, including inherited ones, as shown in Figure 3-7. This is the simplest to work with in your application, as every object you create will map onto one row of one table. It is not very resilient to schema changes, however. Changes in design of the class at the root of the inheritance tree require changes in *all* tables.

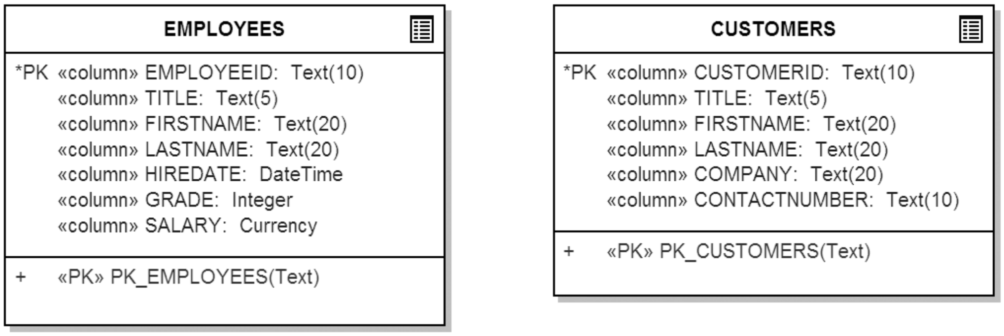
EMPLOYEES
*PK «column» EMPLOYEEID: Text(10)
«column» TITLE: Text(5)
«column» FIRSTNAME: Text(20)
«column» LASTNAME: Text(20)
«column» HIREDATE: DateTime
«column» GRADE: Integer
«column» SALARY: Currency
+ «PK» PK_EMPLOYEES(Text)
CUSTOMERS
*PK «column» CUSTOMERID: Text(10)
«column» TITLE: Text(5)
«column» FIRSTNAME: Text(20)
«column» LASTNAME: Text(20)
«column» COMPANY: Text(20)
«column» CONTACTNUMBER: Text(10)
+ «PK» PK_CUSTOMERS(Text)

Figure 3-7. *Data model using horizontal mapping*

Filtered Mapping: One Table per Tree

This ducks the issue of dealing with subclasses by lumping the whole tree together in a single table that has all the attributes of all the classes in the tree, as shown in Figure 3.8. A filter column (PERSONTYPE) is included to distinguish between subclasses. This approach manages to violate principles of both object and relational modeling at the same time!

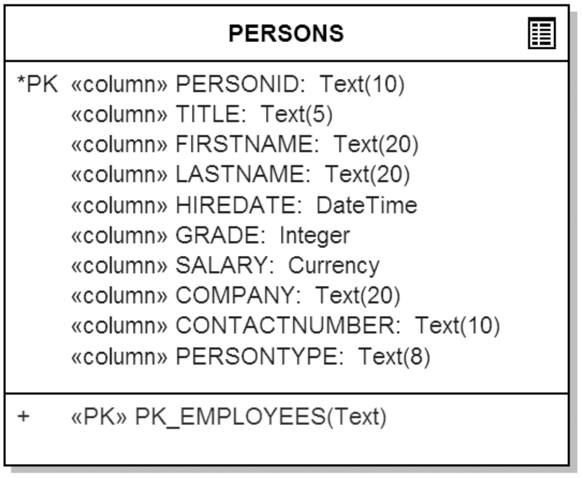
PERSONS
*PK «column» PERSONID: Text(10)
«column» TITLE: Text(5)
«column» FIRSTNAME: Text(20)
«column» LASTNAME: Text(20)
«column» HIREDATE: DateTime
«column» GRADE: Integer
«column» SALARY: Currency
«column» COMPANY: Text(20)
«column» CONTACTNUMBER: Text(10)
«column» PERSONTYPE: Text(8)
+ «PK» PK_EMPLOYEES(Text)

Figure 3-8. *Data model using filtered mapping*

Many-Many Relationships

The class diagram in Figure 3-9 shows a many-many relationship between the `Person` and `Project` classes. A `Person` can be assigned to more than one `Project`, while a single `Project` has many `Persons` assigned to it.

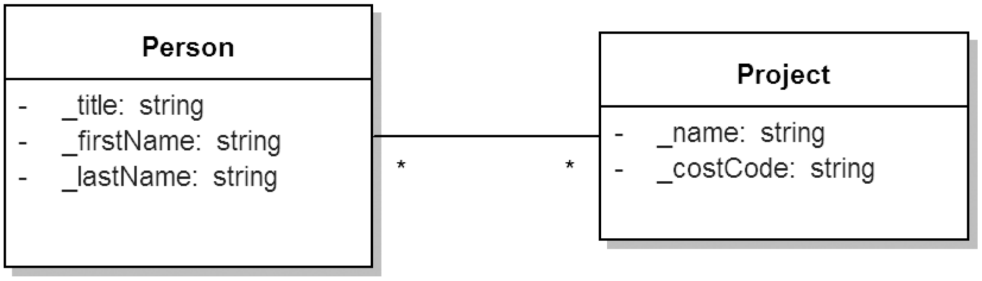

Figure 3-9. *A many-many relationship*

This is a common relationship, quite feasible in the object model, but it can't be represented directly by the relational model. This is a well-known problem: the workaround has been used since long before the object model appeared. In a relational database a join table is required to represent this relationship, introducing more key fields, as shown in Figure 3-10.

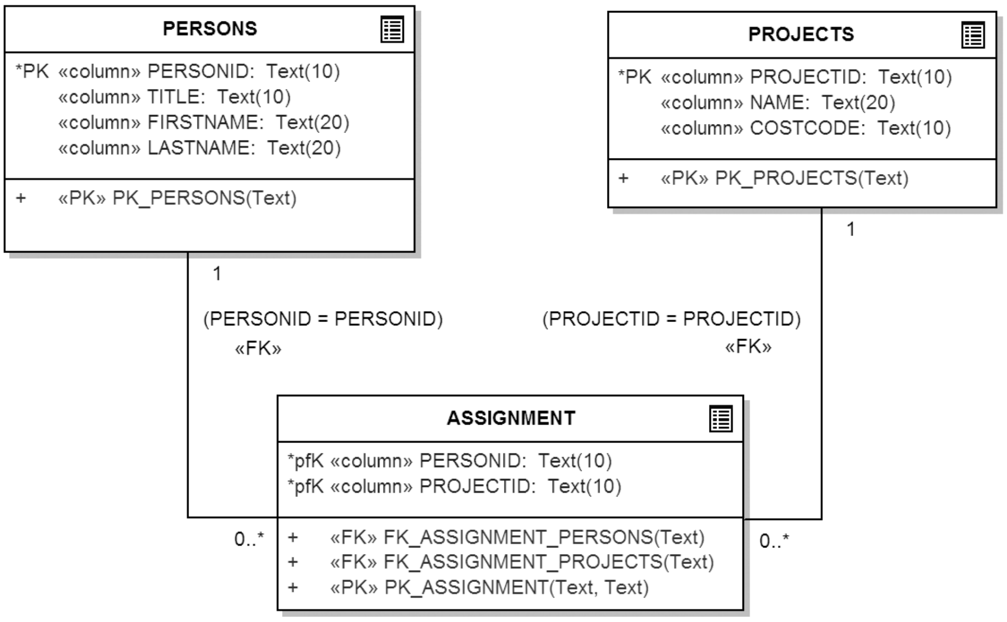
PERSONS
*PK «column» PERSONID: Text(10)
«column» TITLE: Text(10)
«column» FIRSTNAME: Text(20)
«column» LASTNAME: Text(20)
+ «PK» PK_PERSONS(Text)
PROJECTS
*PK «column» PROJECTID: Text(10)
«column» NAME: Text(20)
«column» COSTCODE: Text(10)
+ «PK» PK_PROJECTS(Text)
1
1
(PERSONID = PERSONID)
«FK»
(PROJECTID = PROJECTID)
«FK»
ASSIGNMENT
*pfK «column» PERSONID: Text(10)
*pfK «column» PROJECTID: Text(10)
+ «FK» FK_ASSIGNMENT_PERSONS(Text)
+ «FK» FK_ASSIGNMENT_PROJECTS(Text)
+ «PK» PK_ASSIGNMENT(Text, Text)
0..*
0..*

Figure 3-10. *Data model for many-many relationship*

Complex Relationships

Clearly, the relational model is different enough from the object model to make life complicated when you are trying to map anything but the simplest type of object relationship. Even in the simplest cases, you need to add key fields to make relationships work.

In Chapter 2, you saw that object relationships can become more complicated, and that many useful ones have been captured as design patterns. For example, the composite pattern example that we saw in Chapter 2 combines aggregation and inheritance to represent whole-part hierarchies, as illustrated in Figure 3-11.

Sorry, we're going to leave the mapping of this scenario as an exercise for the reader—have fun with this! Remember, in the object data model these relationships stay just as they are.

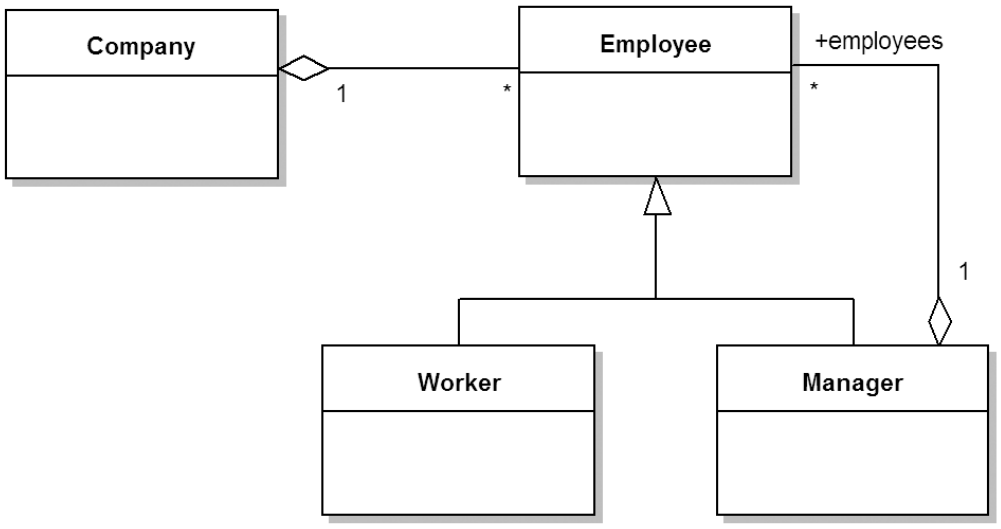

Figure 3-11. *Composite pattern example class diagram*

Referential Integrity and All That

Many people who are considering using db4o have a lot of experience with relational databases. Their first questions are often about whether an object database can provide the features they take for granted. In Chapter 13 we will look in detail at the issues of moving from RDBMS to db4o, but for now let's look at some common database features, and see how they relate to the object model.

Queries

Querying using SQL is a fundamental capability of a relational database. Atkinson's original manifesto for object databases (see the earlier sidebar, "Origins of the Object Database") stated that a query facility was a mandatory feature of an object database, and the ODMG standards defined a high-level query language, OQL. The nature of the data model dictates that object database queries be expressed somewhat differently from SQL queries, making use of object references rather than joins. db4o supports a new approach to querying, the type-safe native query, which is described in Chapter 6.

Referential Integrity

Referential integrity ensures, for example, that you couldn't create an `order` in the database for a `customer` who does not exist in the database.

In the relational model, referential integrity is enforced by foreign key relationships. In this example, the `ORDERS` table would have a foreign key that referred to the primary key of the `CUSTOMERS` table. Before a new `order` is added to an `ORDERS` table, the foreign key field is checked to make sure there is a matching value in the `CUSTOMERS` table.

In the object model, integrity is controlled by the application. Object references created in the application are maintained in the database. In the example, if the application is written so that all new `Order` instances are created as fields of `Customer` objects, then it is not possible to have an orphaned `Order` object. The database logic matches the application logic.

The way the previously discussed features work is strongly dependent on the data model. The following features are also required for a database to be considered to have ACID (Atomicity, Concurrency, Isolation and Durability) properties. As far as the user or developer is concerned, the way these are used depends not on what data model the database is using, but on the particular database product.

Transactions

Atomicity states that database modifications must follow an "all or nothing" rule. Each transaction is said to be "atomic." If one part of the transaction fails, the entire transaction fails. db4o's support for transactions is described in Chapter 9.

Isolation

Isolation requires that multiple transactions running at the same time should not affect each other's execution. For example, if one user runs a transaction against a database at the same time that another user runs a different transaction, both transactions should operate on the database in an isolated manner. db4o supports concurrent transactions in client/server mode, either as an embedded server or a network server. Client/server mode is described in Chapter 8.

Summary

In this chapter we looked at the evolution of data models and compared the object and relational data models. We looked at some of the difficulties involved in fitting object-oriented systems and relational databases together, and we looked at how some well-known database features are implemented in an object database. In the next chapter, we take a closer look at the object data model and how db4o stores its data.