Part III

Relational Database Design Practice

In this part of the book, you will read about some of the practical techniques we use when working with relational database designs. You will be introduced to the SQL language statements needed to create relational schemes and their contents. You will also see how a CASE tool can help design and document a database. In addition, this part contains three complete relational design case studies to provide further examples of the database design process.

Page left intentionally blank

Chapter 10 Introduction to SQL

SQL[1] is a database definition and database manipulation language that has been implemented by virtually every relational database management system (DBMS) intended for multiple users, partly because it has been accepted by ANSI (the American National Standards Institute) and ISO (International Standards Organization) as a standard query language for relational databases.

The chapter presents an overview of the environment in which SQL exists. We will begin with a bit of SQL history so you will know where it came from and where it is heading. Then, you will read about the way in which SQL commands are processed and the software environments in which they function.

A Bit of SQL History

SQL was developed by IBM at its San Jose Research Laboratory in the early 1970s. Presented at an ACM conference in 1974, the language was originally named SEQUEL (Structured English Query Language) and pronounced "sequel." The language's name was later shortened to SQL.

Although IBM authored SQL, the first SQL implementation was provided by Oracle Corporation (then called Relational Software Inc.). Early commercial implementations were concentrated on midsized UNIX-based DBMSs, such as Oracle, Ingres, and Informix. IBM followed in 1981 with SQL/DS, the forerunner to DB2, which debuted in 1983.

[1]Whether you say "sequel" or "S-Q-L" depends on how long you've been working with SQL. Those of us who have been working in this field for longer than we'd like to admit often say "sequel," which is what I do. When I started using SQL, there was no other pronunciation. That is why you'll see "a SQL" (a sequel) rather than "an SQL" (an es-que-el) throughout this book. Old habits die hard! However, many people do prefer the acronym.

ANSI published the first SQL standard (SQL-86) in 1986. An international version of the standard issued by ISO appeared in 1987. A significant update to SQL-86 was released in 1989 (SQL-89). Virtually all relational DBMSs that you encounter today support most of the 1989 standard.

In 1992, the standard was revised again (SQL-92), adding more capabilities to the language. Because SQL-92 was a superset of SQL-89, older database application programs ran under the new standard with minimal modifications. In fact, until October 1996, DBMS vendors could submit their products to NIST (National Institute for Standards and Technology) for verification of SQL standard compliance. This testing and certification process provided significant motivation for DBMS vendors to adhere to the SQL standard. Although discontinuing standard compliance testing saves the vendors money, it also makes it easier for products to diverge from the standard.

The SQL-92 standard was superseded by SQL:1999, which was once again a superset of the preceding standard. The primary new features of SQL:1999 supported the object-relational data model, which is discussed in Chapter 27 of this book.

The SQL:1999 standard also adds extensions to SQL to allow scripts or program modules to be written in SQL or to be written in another programming language, such as C++ or Java and then invoked from within another SQL statement. As a result, SQL becomes less "relational," a trend decried by some relational purists.

Note: Regardless of where you come down on the relational theory argument, you will need to live with the fact that the major commercial DBMSs, such as Oracle and DB/2, have provided support for the object-relational data model for some time now. The object-relational data model is a fact of life, although there certainly is no rule that says that you must use those features, should you choose not to do so.

Even the full SQL:1999 standard does not turn SQL into a complete, standalone programming language. In particular, SQL lacks I/O statements. This makes perfect sense, since SQL should be implementation and operating system independent. However, the full SQL:1999 standard does include operations such as selection and iteration that make it *computationally complete*. These language features, which are more typical of general-purpose programming languages, are used when writing stored procedures and triggers:

- Triggers: A *trigger* is a script that runs when a specific database action occurs. For example, a trigger might be written to execute when data are inserted or deleted.

- Stored procedures: A *stored procedure* is a script that runs when it is called by an application program written in a general-purpose programming language or another SQL language module.

Triggers and stored procedures are stored in the database itself, rather than being a part of an application program. The scripts are, therefore, available to all application programs written to interact with the database.

The SQL standard has been updated four times since the appearance of SQL:1999, in versions named SQL:2003, SQL:2006, SQL:2008, and SQL:2011. As well as fleshing out the capabilities of the core relational capabilities and extending object-relational support, these revisions have added support for XML (Extensible Markup Language). XML is a platform-independent method for representing data using text files. SQL's XML features are introduced in Chapter 26.

Conformance Levels

The SQL in this book is based on the more recent versions of the SQL standard (SQL:2003 through SQL:2011). However, keep in mind that SQL:2011 (or whatever version of the language you are considering) is simply a standard, not a mandate. Various DBMSs exhibit different levels of conformance to the standard. In addition, the implementation of language features usually lags behind the standard. Therefore, although SQL:2011 may be the latest version of the standard, no DBMS meets the entire standard and most are based on earlier versions.

Note: In one sense, the SQL standard is a moving target. Just as DBMSs look like they're going to catch up to the most recent standard, the standard is updated. DBMS developers scurry to implement new features and, as soon as they get close, the standard changes again.

Conformance to early versions of the standard (SQL-92 and earlier) was measured by determining whether the portion of the language required for a specific level of conformance were supported. Each feature in the standard was identified by a *leveling rule*, indicating at which conformance level it was required. At the time, there were three conformance levels:

- Full SQL-92 conformance: all features in the SQL-92 standard are supported.
- Intermediate SQL-92 conformance: all features required for intermediate conformance are supported.
- Entry SQL-92: conformance: all features required for entry level conformance are supported.

In truth, most DBMSs were only entry level compliant, and some supported a few of the features at higher conformance levels. The later standards define conformance in a different way, however.

The standard itself is documented in nine parts (parts 1, 2, 3, 4, 9, 10, 11, 13, 14). Core conformance is defined as supporting the basic SQL features (Part 2, Core/Foundation) as well as features for definition and information schemas (Part 11, SQL/Schemata). A DBMS can claim conformance to any of the remaining parts individually, as long as the product meets the conformance rules presented in the standard.

In addition to language features specified in the standard, there are some features from earlier standards that, although not mentioned in the 2006, 2008, and 2011 standards, are widely implemented. This includes, for example, support for indexes.

SQL Environments

There are two general ways in which you can issue a SQL command to a database:

- Interactive SQL, in which a user types a single command and sends it immediately to the database: The result of an interactive query is a table in main memory (a *virtual table*). In mainframe environments, each user has one result table at a time, which is replaced each time a new query is executed; PC environments sometimes allow several. As you read in Chapter 6, result tables may not be legal relations—because of nulls, they may have no primary key—but that is not a problem because they are not part of the database, but exist only in main memory.
- Embedded SQL, in which SQL statements are placed in an application program: The interface presented to the user may be form-based or command-line based. Embedded SQL may be *static*, in which case the entire command is specified at the time the program is written. Alternatively, it may be *dynamic*, in which case the program builds the statement using user input and then submits it to the database.

In addition to the two methods for writing SQL syntax, there are also a number of graphic query builders. These provide a way for a user who may not know the SQL language to "draw" the elements of a query. Some of these programs are report writers (for example, Crystal Reports[2]) and are not intended for data modification or for maintaining the structure of a database.

[2]For more information, see www.crystalreports.com.

Interactive SQL Command Processors

At the most general level, we can describe working with an interactive SQL command processor in the following way:

- Type the SQL command.
- Send the command to the database and wait for the result.

In this era of the graphic user interface (GUI), command line environments like that in Figure 10.1 seem rather primitive. Nonetheless, the SQL command line continues to provide basic access to relational databases and is used extensively when developing a database.

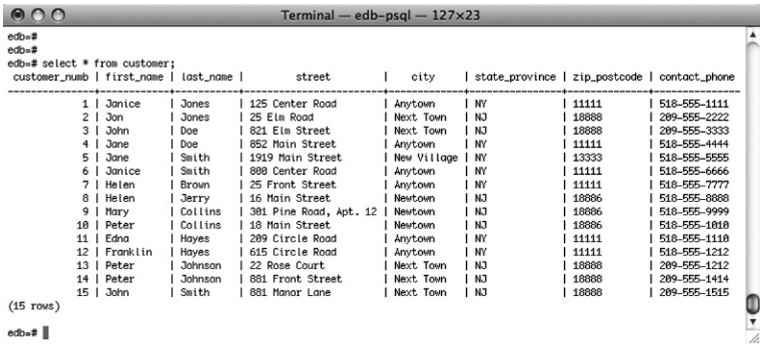


FIGURE 10.1 A typical SQL command line environment.

A command line environment also provides support for *ad hoc queries*, queries that arise at the spur of the moment, and are not likely to be issued with any frequency. Experienced SQL users can usually work faster at the command line than with any other type of SQL command processor.

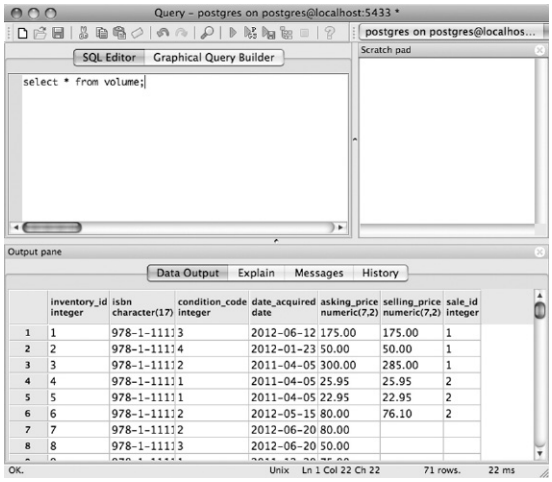
The down side to the traditional command line environment is that it is relatively unforgiving. If you make a typing error or an error in the construction of a command, it may be difficult to get the processor to recall the command so that it can be edited and resubmitted to the database. In fact, you may have no other editing capabilities except the backspace key.

The SQL command examples that you will see throughout this book were all tested in a command line environment. As you are learning to create your own queries, this is, in most cases, the environment in which you will be working.

GUI Environments

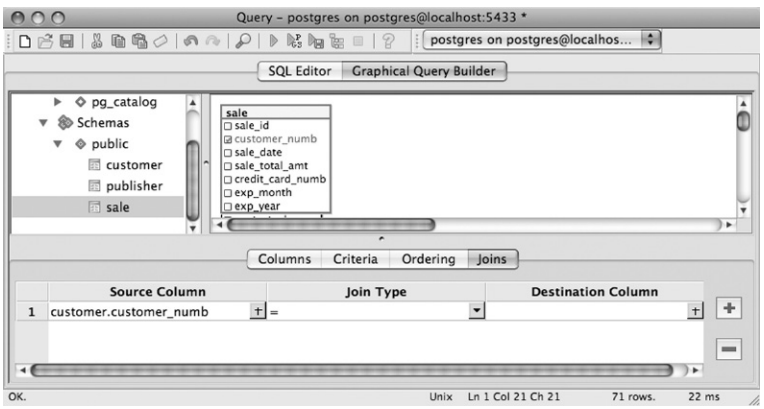
There are actually two strategies used by GUI environments to provide access to a SQL database. The first is to simply provide a window into which you can type a command, just as you would do from the command line (for

example, Figure 10.2, which shows a query tool for Postgres). Such environments usually make it easier to edit the command, supporting recall of the command and full-screen editing features.



■ **FIGURE 10.2 Typing a SQL command into a window.**

The other strategy is to provide a "query builder," an environment in which the user is guided through the construction of the query (for example, Figure 10.3). The query builder presents the user with lists of the legal command elements. Those lists change as the query is built so that the user also constructs legal syntax. The query builder type of SQL command



■ **FIGURE 10.3 A "query builder" environment.**

environment makes it much easier for many users to construct correct SQL statements, but it is also slower than working directly at the command line.

The Embedded SQL Dilemma

Embedding SQL in a general-purpose programming language presents an interesting challenge. The host languages (for example, Java, C++, JavaScript) have compilers or interpreters that don't recognize SQL statements as legal parts of the language. The solution is to provide SQL support through an application library that can be linked to a program. Program source code is passed through a precompiler that changes SQL commands into calls to library routines. The modified source code will then be acceptable to a host language compiler or interpreter.

In addition to the problem of actually compiling an embedded SQL program, there is a fundamental mismatch between SQL and a general-purpose programming language: Programming languages are designed to process data one row at a time, while SQL is designed to handle many rows at a time. SQL therefore includes some special elements so that it can process one row at a time when a query has returned multiple rows.

Elements of a SQL Statement

There are certainly many options for creating a SQL command. However, they are all made up of the same elements:

- Keywords: Each SQL command begins with a keyword—such as SELECT, INSERT, or UPDATE—that tells the command processor the type of operation that is to be performed. The remainder of the keywords precede the tables from which data are to be taken, indicate specific operations that are to be performed on the data, and so on.
- Tables: A SQL command includes the names of the tables on which the command is to operate.
- Columns: A SQL command includes the names of the columns that the command is to affect.
- Functions: A *function* is a small program that is built into the SQL language. Each function does one thing. For example, the AVG function computes the average of numeric data values. You will see a number of SQL functions discussed throughout this book.

Keywords and tables are required for all SQL commands. Columns may be optional, depending on the type of operation being performed. Functions are never required for a legal SQL statement, but, in some cases may be essential to obtaining a desired result.

For Further Reading

Chamberlin, DD, Raymond FB. SEQUEL: a structured english query language. 1974. http://www.almaden.ibm.com/cs/people/chamberlin/sequel-1974.pdf

Programmers Stack Exchange. Sequel vs. S-Q-L. http://programmers.stackexchange.com/questions/108518/sequel-vs-s-q-l

O'Reilly Media. History of the SQL Standard from SQL in a nutshell, 2nd ed. (entire book: http://users.atw.hu/sqlnut/index.html). http://users.atw.hu/sqlnut/sqlnut2-chp-1-sect-2.html

Chapter 11

Using SQL to Implement a Relational Design

As a complete data manipulation language, SQL contains statements that allow you to insert, modify, delete, and retrieve data. However, to a database designer, the portions of SQL that support the creation of database structural elements are of utmost importance. In this chapter, you will be introduced to the SQL commands that you will use to create and maintain the tables that make up a relational database.

Some of the structural elements in a relational database are created with SQL retrieval commands embedded in them. We will therefore defer a discussion of those elements until Chapter 21.

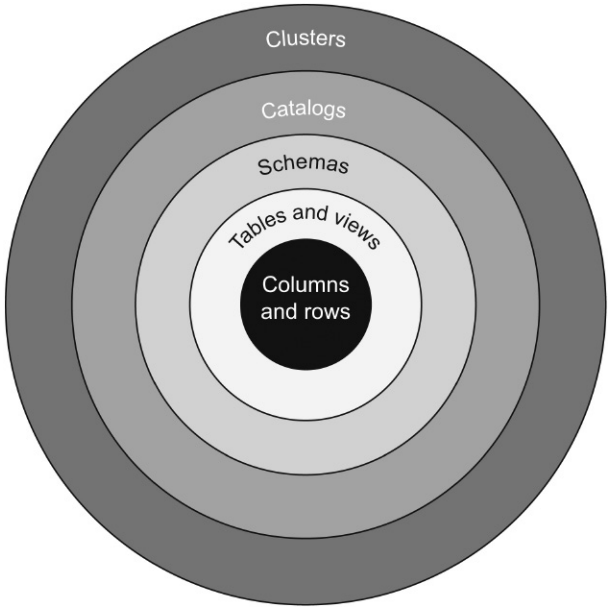
The actual file structure of a database is implementation dependent, as is the procedure needed to create database files. Therefore, the discussion in this chapter assumes that the necessary database files are already in place.

Note: You will see extensive examples of the use of the syntax presented in this chapter at the end of each of the three case studies that follow in this book.

Database Structure Hierarchy

The elements that describe the structure of a SQL:2011-compliant database are arranged in a hierarchy, diagrammed in Figure 11.1. The smallest units with which the DBMS works—columns and rows—appear in the center. These, in turn, are grouped into tables and views.

The tables and views that comprise a single logical database are collected into a schema. Multiple schemas are grouped into catalogs, which can then be grouped into clusters. A catalog usually contains information describing



■ **FIGURE 11.1 The SQL:2011 database structure hierarchy.**

all the schemas handled by one DBMS. Catalog creation is implementation dependent and, therefore, not part of the SQL standard.

Prior to SQL-92, clusters often represented database files, and the clustering of database elements into files was a way to increase database performance by placing data accessed together in the same physical file. The SQL-92 and beyond concept of a cluster, however, is a group of catalogs that are accessible using the same connection to a database server.

Note: Don't forget that SQL is a dynamic language. The standard has been updated in 1999, 2003, 2006, and 2011.

In current versions of SQL, none of the groupings of database elements are related to physical storage structures. If you are working with a centralized mainframe DBMS, you may find multiple catalogs stored in the same database file. However, on smaller or distributed systems, you are likely to find one catalog or schema per database file or to find a catalog or schema split between multiple files.

Clusters, catalogs, and schemas are not required elements of a database environment. In a small installation where there is one collection of tables serving a single purpose, for example, it may not even be necessary to create a schema to hold them.

Naming and Identifying Structural Elements

The way in which you name and identify database structural elements is, in some measure, dictated by the structure hierarchy:

- Column names must be unique within the table.
- Table names must be unique within the schema.
- Schema names must be unique within their catalog.
- Catalog names must be unique within their cluster.

When a column name appears in more than one table in a SQL statement—as it often does, when there are primary key–foreign key references—the user must specify the table from which a column should be taken (even if it makes no difference which table is used). The general form for qualifying duplicate column names is:

```
table_name.column_name
```

If an installation has more than one schema, then a user must also indicate the schema in which a table resides:

```
schema_name.table_name.column_name
```

This naming convention means that two different schemas can include tables with the same name.

By the same token, if an installation has multiple catalogs, a user will need to indicate the catalog from which a database element comes:

```
catalog_name.schema_name.table_name.column_name
```

The names that you assign to database elements can include the following:

- Letters
- Numbers
- Underscores (_)

SQL names can be up to 128 characters long. According to the SQL standard, they should not be case sensitive. In fact, many SQL command processors convert names to all upper- or lowercase before submitting a SQL statement to a DBMS for processing. However, there are always exceptions, so you should check your DBMS's naming rules before creating any structural elements.

Note: Some DBMSs also allow pound signs (#) and dollar signs ($) in element names, but neither is recognized by SQL statements, so their use should be avoided.

Schemas

To a database designer, a schema represents the overall, logical design of a complete database. As far as SQL is concerned, however, a schema is nothing more than a container for tables, views, and other structural elements. It is up to the database designer to place a meaningful group of elements within each schema.

A schema is not required to create tables and views. In fact, if you are installing a database for an environment in which there is likely to be only one logical database, then you can just as easily do without one. However, if more than one database will be sharing the same DBMS and the same server, then organizing database elements into schemas can greatly simplify the maintenance of the individual databases.

Creating a Schema

To create a schema, you use the CREATE SCHEMA statement. In its simplest form, it has the syntax

```
CREATE SCHEMA schema_name
```

as in

```
CREATE SCHEMA antique_opticals;
```

Note: The majority of SQL command processors require a semicolon at the end of each statement. However, that end-of-statement marker is not a part of the SQL standard and you may encounter DBMSs that do not use it. We will use it in this book at the end of statements that could be submitted to a DBMS. It does not appear at the end of command "templates" that show the elements of a command.

By default, a schema belongs to the user who created it (the user ID under which the schema was created). The owner of the schema is the only user ID that can modify the schema, unless the owner grants that ability to other users.

To assign a different owner to a schema, you add an AUTHORIZATION clause:

```
CREATE SCHEMA schema_name AUTHORIZATION owner_user_ID
```

For example, to assign the *Antique Opticals* schema to the user ID DBA, someone could use:

```
CREATE SCHEMA antique_opticals AUTHORIZATION dba;
```

When creating a schema, you can also create additional elements at the same time. To do so, you use braces to group the CREATE statements for the other elements, as in:

```
CREATE SCHEMA schema_name AUTHORIZATION owner_user_id
{
// other CREATE statements go here
}
```

This automatically assigns the elements within the braces to the schema.

Identifying the Schema You Want to Use

One of the nicest things about a relational database is that you can add or delete database structural elements at any time. There must therefore be a way to specify a current schema for new database elements after the schema has been created initially with the CREATE SCHEMA statement.

One way to do this is with the SET SCHEMA statement:

```
SET SCHEMA schema_name
```

To use SET SCHEMA, the user ID under which you are working must have authorization to work with that schema.

Alternatively, you can qualify the name of a database element with the name of the schema. For example, if you are creating a table, then you would use something like

```
CREATE TABLE schema_name.table_name
```

For those DBMSs that do not support SET SCHEMA, this is the only way to attach new database elements to a schema after the schema has been created.

Domains

As you know, a domain is an expression of the permitted values for a column in a relation. When you define a table, you assign each column a data type (example, character, or integer) that provides a broad domain. A DBMS will not store data that violate that constraint.

The SQL-92 standard introduced the concept of user-defined domains, which can be viewed as user-defined data types that can be applied to columns in tables. (This means you have to create a domain before you can assign it to a column!)

Domains can be created as part of a CREATE SCHEMA statement, which has the following syntax:

```
CREATE DOMAIN domain_name data_type
CHECK (expression_to_validate_values)
```

The CHECK clause is actually a generic way of expressing a condition that data must meet. It can include a retrieval query to validate data against other data stored in the database, or it can include a simple logical expression. In that expression, the keyword VALUE represents the data being checked.

For example, if *Antique Opticals* wanted to validate the price of a disc, someone might create the following domain:

```
CREATE DOMAIN price numeric (6,2)
CHECK (VALUE >= 19.95);
```

After creating this domain, a column in a table can be given the data type of price. The DBMS will then check to be certain that the value in that column is always greater than, or equal to, 19.95. (We will leave a discussion of the data type used in the preceding SQL statement until we cover creating tables in the next section of this chapter.)

The domain mechanism is very flexible. Assume, for example, that you want to ensure that telephone numbers are always stored in the format XXX-XXX-XXXX. A domain to validate that format might be created as:

```
CREATE DOMAIN telephone char (12)
CHECK (SUBSTRING (VALUE FROM 4 FOR 1 = '-') AND
     SUBSTRING (VALUE FROM 8 FOR 1 = '-'));[1]
```

When a user attempts to store a value in a column to which the *telephone* domain has been assigned, the DBMS performs two SUBSTRING functions. The first looks at the fourth character in the string to determine whether it is a -. The second examines the character at position 8. The data will be accepted only if both conditions are met.

You can use the CREATE DOMAIN statement to give a column a default value. For example, the following statement sets up a domain that holds either Y or N, and defaults to Y:

```
CREATE DOMAIN boolean char (1)
DEFAULT = 'Y'
CHECK (UPPER(VALUE) = 'Y' OR UPPER(VALUE) = 'N');[2]
```

[1]The SUBSTRING function extracts characters from text data, beginning at the character following FROM. The number of characters to extract appears after the FOR.
[2]As you might guess, the UPPER function converts characters to upper case. Although SQL names may not be case sensitive, data are!

Tables

The most important structure within a relational database is the table. As you know, tables contain just about everything, including business data and the data dictionary.

SQL divides tables into three categories:

- *Permanent base tables*: Permanent base tables are tables whose contents are stored in the database, and remain permanently in the database, unless they are explicitly deleted.
- *Global temporary tables*: Global temporary tables are tables for working storage that are destroyed at the end of a SQL session. The definitions of the tables are stored in the data dictionary, but their data are not. The tables must be loaded with data each time they are going to be used. Global temporary tables can be used only by the current user, but they are visible to an entire SQL session (either an application program or a user working with an interactive query facility).
- *Local temporary tables*: Local temporary tables are similar to global temporary tables. However, they are visible only to the specific program module in which they are created.

Temporary base tables are subtly different from views, which assemble their data by executing a SQL query. You will read more about this difference and how temporary tables are created and used later in this chapter.

Most of the tables in a relational database are permanent base tables. You create them with the CREATE TABLE statement:

```
CREATE TABLE table_name
(   column1_name column1_data_type
    column1_constraints,
    column2_name column2_data_type
    column2_constraints, …
    table_constraints)
```

The constraints on a table include declarations of primary and foreign keys. The constraints on a column include whether values in the column are mandatory, as well as other constraints you may decide to include in a CHECK clause.

Column Data Types

Each column in a table must be given a data type (or a user-defined domain). Although data types are somewhat implementation dependent, most DBMSs that support SQL include the following predefined data types:

- INTEGER (abbreviated INT): a positive or negative whole number. The number of bits occupied by the value is implementation dependent. In most cases, integers are either 32 or 64 bits.

- SMALLINT: a positive or negative whole number. A small integer is usually half the size of a standard integer. Using small integers when you know you will need to store only small values can save space in the database.
- NUMERIC: a fixed-point positive or negative number. A numeric value has a whole number portion and a fractional portion. When you create it, you must specify the total length of the number (including the decimal point), and how many of those digits will be to the right of the decimal point (its *precision*). For example

  ```
  NUMERIC (6,2)
  ```

  creates a number in the format XXX.XX. The DBMS will store exactly two digits to the right of the decimal point.
- DECIMAL: a fixed-point positive or negative number. A decimal number is similar to a numeric value. However, the DBMS may store more digits to the right of the decimal point than you specify. Although there is no guarantee that you will get the extra precision, its use can provide more accurate results in computations.
- REAL: a "single-precision" floating point value. A floating point number is expressed in the format:

  ```
  XX,XXXXX * 10YY
  ```

  where YY is the power to which 10 is raised. Because of the way in which computers store floating point numbers, a real number may not be an exact representation of a value, but only a close approximation. The range of values that can be stored is implementation dependent, as is the precision. You therefore cannot specify a size for a real number column.
- DOUBLE PRECISION (abbreviated DOUBLE): a "double-precision" floating point number. The range and precision of double precision values are implementation dependent, but generally both will be greater than single-precision real numbers.
- FLOAT: a floating point number for which you can specify the precision. The DBMS will maintain at least the precision that you specify. (It may be more.)
- BOOLEAN: a true/false value. The BOOLEAN data type was introduced in the SQL:1999 standard, but is not supported by all DBMSs. If your DBMS does not include BOOLEAN, the best alternative is usually a one-character text column with a CHECK clause restricting values to Y and N.
- BIT: storage for a fixed number of individual bits. You must indicate the number of bits, as in

  ```
  BIT (n)
  ```

where *n* is the number of bits. (If you do not, you will have room for only one bit.)

- BIT VARYING: storage for a varying number of bits up to a specified maximum, as in

  ```
  BIT VARYING (n)
  ```

  where *n* is the maximum number of bits. In some DBMSs, columns of this type can be used to store graphic images.
- DATE: a date.
- TIME: a time.
- TIMESTAMP: the combination of a date and a time.
- CHARACTER (abbreviated CHAR): a fixed-length space to hold a string of characters. When declaring a CHAR column, you need to indicate the width of the column:

  ```
  CHAR (n)
  ```

  where *n* is the amount of space that will be allocated for the column in every row. Even if you store less than *n* characters, the column will always take up *n* bytes—or n × 2 bytes, if you are storing UNICODE characters—and the column will be padded with blanks to fill up empty space. The maximum number of characters allowed is implementation dependent (usually 256 or more).
- CHARACTER VARYING (abbreviated VARCHAR): a variable length space to hold a string of characters. You must indicate the maximum width of the column—

  ```
  VARCHAR (n)
  ```

  —but the DBMS stores only as many characters as you insert, up to the maximum *n*. The overall maximum number of characters allowed is implementation dependent (usually 256 or more).
- INTERVAL: a date or time interval. An interval data type is followed by a qualifier that specifies the size of the interval and, optionally, the number of digits. For example:

  ```
  INTERVAL YEAR
  INTERVAL YEAR (n)
  INTERVAL MONTH
  INTERVAL MONTH (n)
  INTERVAL YEAR TO MONTH
  INTERVAL YEAR (n) TO MONTH
  INTERVAL DAY
  INTERVAL DAY (n)
  INTERVAL DAY TO HOUR
  INTERVAL DAY (n) TO HOUR
  INTERVAL DAY TO MINUTE
  INTERVAL DAY (n) TO MINUTE
  INTERVAL MINUTE
  INTERVAL MINUTE (n)
  ```

In the preceding examples, *n* specifies the number of digits. When the interval covers more than one date and/or time unit, such as YEAR TO MONTH, you can specify a size for only the first unit. Year–month intervals can include days, hours, minutes, and/or seconds.

- BLOB (Binary Large Object): a block of binary code (often a graphic) stored as a unit and retrievable only as a unit. In many cases, the DBMS cannot interpret the contents of a BLOB (although the application that created the BLOB can do so). Because BLOB data are stored as undifferentiated binary, BLOB columns cannot be searched directly. Identifying information about the contents of a BLOB must be contained in other columns of the table, using data types that can be searched.

In Figure 11.2 you will find bare-bones CREATE TABLE statements for the *Antique Opticals* database. These statements include only column names and data types. SQL will create tables from statements in this format, but because the tables have no primary keys, many DBMSs will not allow you to enter data.

Default Values

As you are defining columns, you can designate a default value for individual columns. To indicate a default value, you add a DEFAULT keyword to the column definition, followed by the default value. For example, in the *orders* relation the order date column defaults to the current system date. The column declaration is therefore written:

```
order_date date DEFAULT CURRENT_DATE;
```

Notice that this particular declaration is using the SQL value CURRENT_DATE. However, you can place any value after DEFAULT that is a valid instance of the column's data type.

NOT NULL Constraints

The values in primary key columns must be unique and not null. In addition, there may be other columns for which you want to require a value. You can specify such columns by adding NOT NULL after the column declaration. Since *Antique Opticals* wants to ensure that an order date is always entered, the complete declaration for that column in the *orders* table is

```
order_date date NOT NULL DEFAULT CURRENT_DATE;
```

Primary Keys

There are two ways to specify a primary key:

- Add a PRIMARY KEY clause to a CREATE TABLE statement. The keywords PRIMARY KEY are followed by the names of the primary key column or columns, surrounded by parentheses.

```
CREATE TABLE customer
      (customer_numb int,
      customer_first_name varchar (15),
      customer_last_name varchar (15),
      customer_street varchar (30),
      customer_city varchar (15),
      customer_state char (2),
      customer_zip char (10),
      customer_phone char (12));

CREATE TABLE distributor
      (distributor_numb int,
      distributor_name varchar (15),
      distributor_street varchar (30),
      distributor_city varchar (15),
      distributor_state char (2),
      distributor_zip char (10),
      distributor_phone char (12),
      distributor_contact_person varchar (30),
      contact_person_ext char (5));

CREATE TABLE item
      (item_numb int,
      item_type varchar (15),
      title varchar (60),
      distributor_numb int,
      retail_price numeric (6,2),
      relase_date date,
      genre varchar (20),
      quant_in_stock int);

CREATE TABLE order
      (order_numb int,
      customer_numb int,
      order_date date,
      credit_card_numb char (16),
      credit_card_exp_date char (5),
      order_complete boolean,
      pickup_or_ship char (1));

CREATE TABLE order_line
      (order_numb int,
      item_numb int,
      quantity int,
      discount_percent int,
      selling_price numeric (6,2),
      line_cost numeric (7,2),
      shipped boolean,
      shipping_date date);
```

■ **FIGURE 11.2 Initial CREATE TABLE statements for the *Antique Opticals* database (continues).**

```
CREATE TABLE purchase
       (purchase_date date,
       customer_numb int,
       items_received boolean),
       customer_paid boolean);

CREATE TABLE purchase_item
       (purchase_date date,
       customer_numb int,
       item_numb int,
       condition char (15),
       price_paid numeric (6,2));

CREATE TABLE actor
       (actor_numb int,
       actor_name varchar (60));

CREATE TABLE performance
       (actor_numb int,
       item_numb int,
       role varchar (60));

CREATE TABLE producer
       (producer_name varchar (60),
       studio varchar (40));

CREATE TABLE production
       (producer_name varchar (60),
       item_numb int);
```

■ **FIGURE 11.2 (*cont.*)**

- Add the keywords PRIMARY KEY to the declaration of each column that is part of the primary key. Use a CONSTRAINT clause if you want to name the primary key constraint.

In Figure 11.3, you will find the CREATE TABLE statement for the *Antique Opticals* database, including both PRIMARY KEY and CONSTRAINT clauses. Notice that in those tables that have concatenated primary keys, all the primary key columns have been included in a PRIMARY KEY clause.

Foreign Keys

As you know, a foreign key is a column (or combination of columns) that is exactly the same as the primary of some table. When a foreign key value matches a primary key value, we know that there is a logical relationship between the database objects represented by the matching rows.

```
CREATE TABLE customer
      (customer_numb int PRIMARY KEY,
      customer_first_name varchar (15),
      customer_last_name varchar (15),
      customer_street varchar (30),
      customer_city varchar (15),
      customer_state char (2),
      customer_zip char (10),
      customer_phone char (12));

CREATE TABLE distributor
      (distributor_numb int PRIMARY KEY,
      distributor_name varchar (15),
      distributor_street varchar (30),
      distributor_city varchar (15),
      distributor_state char (2),
      distributor_zip char (10),
      distributor_phone char (12),
      distributor_contact_person varchar (30),
      contact_person_ext char (5));

CREATE TABLE item
      (item_numb int CONSTRAINT item_pk PRIMARY KEY,
      item_type varchar (15),
      title varchar (60),
      distributor_numb int,
      retail_price numeric (6,2),
      relase_date date,
      genre varchar (20),
      quant_in_stock int);

CREATE TABLE order
      (order_numb int,
      customer_numb int,
      order_date date,
      credit_card_numb char (16),
      credit_card_exp_date char (5),
      order_complete boolean,
      pickup_or_ship char (1)
      PRIMARY KEY (order_numb));

CREATE TABLE order_line
      (order_numb int,
      item_numb int,
      quantity int,
      discount_percent int,
```

■ **FIGURE 11.3 CREATE TABLE statements for the *Antique Opticals* database including primary key declarations (continues).**

```
      selling_price numeric (6,2),
      line_cost numeric (7,2),
      shipped boolean,
      shipping_date date
      PRIMARY KEY (order_numb, item_numb));

CREATE TABLE purchase
      (purchase_date date,
      customer_numb int,
      items_received boolean,
      customer_paid boolean
      PRIMARY KEY (purchase_date, customer_numb));

CREATE TABLE purchase_item
      (purchase_date date,
      customer_numb int,
      item_numb int,
      condition char (15),
      price_paid numeric (6,2)
      PRIMARY KEY (purchase_date, customer_numb, item_numb));

CREATE TABLE actor
      (actor_numb int PRIMARY KEY,
      actor_name varchar (60));

CREATE TABLE performance
      (actor_numb int,
      item_numb int,
      role varchar (60)
      PRIMARY KEY (actor_numb, item_numb));

CREATE TABLE producer
      (producer_name varchar (60) CONSTRAINT producer_pk PRIMARY KEY,
      studio varchar (40));

CREATE TABLE production
      (producer_name varchar (60),
      item_numb int
      PRIMARY KEY (producer_name, item_numb));
```

■ **FIGURE 11.3 (*cont.*)**

One of the major constraints on a relation is referential integrity, which states that every nonnull foreign key must reference an existing primary key value. To maintain the integrity of the database, it is vital that foreign key constraints be stored within the database's data dictionary so that the DBMS can be responsible for enforcing those constraints.

To specify a foreign key for a table, you add a FOREIGN KEY clause:

```
FOREIGN KEY foreign_key_name (foreign_key_columns)
REFERENCES primary_key_table (primary_key_columns)
    ON UPDATE update_option
    ON DELETE delete_option
```

Each foreign key–primary key reference is given a name. This makes it possible to identify the reference at a later time, in particular, so you can remove the reference if necessary.

Note: Some DBMSs, such as Oracle, do not support the naming of foreign keys, in which case you would use preceding syntax without the name.

The names of the foreign key columns follow the name of the foreign key. The REFERENCES clause contains the name of the primary key table being referenced. If the primary key columns are named in the PRIMARY KEY clause of their table, then you don't need to list the primary key columns. However, if the columns aren't part of a PRIMARY KEY clause, you must list the primary key columns in the REFERENCES clause.

The final part of the FOREIGN KEY specification indicates what should happen when a primary key value being referenced by a foreign key value is updated or deleted. There are three options that apply to both updates and deletions and one additional option for each:

- SET NULL: Replace the foreign key value with null. This isn't possible when the foreign key is part of the primary key of its table.
- SET DEFAULT: Replace the foreign key value with the column's default value.
- CASCADE: Delete or update all foreign key rows.
- NO ACTION: On update, make no modifications of foreign key values.
- RESTICT: Do not allow deletions of primary key rows.

The complete declarations for the *Antique Opticals* database tables, which include foreign key constraints, can be found in Figure 11.4. Notice that, although there are no restrictions on how to name foreign keys, the foreign keys in this database have been named to indicate the tables involved. This makes them easier to identify, if you need to delete or modify a foreign key at a later date.

Additional Column Constraints

There are additional constraints that you can place on columns in a table beyond primary and foreign key constraints. These include requiring unique values and predicates in CHECK clauses.

```
CREATE TABLE customer
      (customer_numb int PRIMARY KEY,
      customer_first_name varchar (15),
      customer_last_name varchar (15),
      customer_street varchar (30),
      customer_city varchar (15),
      customer_state char (2),
      customer_zip char (10),
      customer_phone char (12));

CREATE TABLE distributor
      (distributor_numb int PRIMARY KEY,
      distributor_name varchar (15),
      distributor_street varchar (30),
      distributor_city varchar (15),
      distributor_state char (2),
      distributor_zip char (10),
      distributor_phone char (12),
      distributor_contact_person varchar (30),
      contact_person_ext char (5));

CREATE TABLE item
      (item_numb int  CONSTRAINT item_pk PRIMARY KEY,
      item_type varchar (15),
      title varchar (60),
      distributor_numb int,
      retail_price numeric (6,2),
      relase_date date,
      genre varchar (20),
      quant_in_stock int);

CREATE TABLE order
      (order_numb int,
      customer_numb int,
      order_date date,
      credit_card_numb char (16),
      credit_card_exp_date char (5),
      order_complete boolean,
      pickup_or_ship char (1)
      PRIMARY KEY (order_numb)
      FOREIGN KEY order2customer (customer_numb)
      REFERENCES customer
            ON UPDATE CASCADE
            ON DELETE RESTRICT);
```

■ **FIGURE 11.4 The complete CREATE TABLE statements for the *Antique Opticals* database (continues).**

```
CREATE TABLE order_line
      (order_numb int,
      item_numb int,
      quantity int,
      discount_percent int,
      selling_price numeric (6,2),
      line_cost numeric (7,2),
      shipped boolean,
      shipping_date date
      PRIMARY KEY (order_numb, item_numb)
      FOREIGN KEY order_line2item (item_numb)
      REFERENCES item
            ON UPDATE CASCADE
            ON DELETE RESTRICT
      FOREIGN KEY order_line2order (order_numb)
      REFERENCES order
            ON UPDATE CASCADE
            ON DELETE CASCADE);

CREATE TABLE purchase
      (purchase_date date,
      customer_numb int,
      items_received boolean,
      customer_paid boolean
      PRIMARY KEY (purchase_date, customer_numb)
      FOREIGN KEY purchase2customer (customer_numb)
      REFERENCES customer
            ON UPDATE CASCADE)
            ON DELETE RESTRICT);

CREATE TABLE purchase_item
      (purchase_date date,
      customer_numb int,
      item_numb int,
      condition char (15),
      price_paid numeric (6,2)
      PRIMARY KEY (purchase_date, customer_numb, item_numb)
      FOREIGN KEY purchase_item2purchase (purchase_date, customer_numb
            ON UPDATE CASCADE
            ON DELETE CASCADE
      FOREIGN KEY purchase_item2item (item_numb)
      REFERENCES item
            ON UPDATE CASCADE
            ON DELETE RESTRICT);

CREATE TABLE actor
      (actor_numb int PRIMARY KEY,
      actor_name varchar (60))
```

■ **FIGURE 11.4 (*cont.*)**

```
CREATE TABLE performance
      (actor_numb int,
      item_numb int,
      role varchar (60)
      PRIMARY KEY (actor_numb, item_numb)
      FOREIGN KEY performance2actor (actor_numb)
      REFERENCES actor
            ON UPDATE CASCADE
            ON DELETE CASCADE
      FOREIGN KEY performance2item (item_numb)
      REFERENCES item
            ON UPDATE CASCADE
            ON DELETE CASCADE);

CREATE TABLE producer
      (producer_name varchar (60) CONSTRAINT producer_pk PRIMARY KEY,
      studio varchar (40));

CREATE TABLE production
      (producer_name varchar (60),
      item_numb int
      PRIMARY KEY (producer_name, item_numb)
      FOREIGN KEY production2producer (producer_name)
      REFERENCES producer
            ON UPDATE CASCADE
            ON DELETE CASCADE
      FOREIGN KEY production2item
      REFERENCES item
            ON UPDATE CASCADE
            ON DELETE CASCADE);
```

■ **FIGURE 11.4 (*cont.*)**

Requiring Unique Values

If you want to ensure that the values in a non-primary key column are unique, then you can use the UNIQUE keyword. UNIQUE verifies that all non-null values are unique. For example, if you were storing social security numbers in an employee table that used an employee ID as the primary key, you could also enforce unique social security numbers with

```
ssn char (11) UNIQUE
```

The UNIQUE clause can also be placed at the end of the CREATE TABLE statement, along with the primary key and foreign key specifications. In that case, it takes the form

```
UNIQUE (column_names)
```

Check Clauses

The CHECK clause to which you were introduced earlier in the chapter, in the "Domains" section, can also be used with individual columns to declare column-specific constraints. To add a constraint, you place a CHECK clause after the column declaration, using the keyword VALUE in a predicate to indicate the value being checked.

For example, to verify that a column used to hold true-false values is limited to T and F, you could write a CHECK clause as

```
CHECK (UPPER(VALUE) = 'T' OR UPPER(VALUE) = 'F')
```

Modifying Database Elements

With the exception of tables, database elements are largely unchangeable. When you want to modify them, you must delete them from the database and create them from scratch. In contrast, just about every characteristic of a table can be modified without deleting the table, using the ALTER TABLE statement.

Adding Columns

To add a new column to a table, use the ALTER TABLE statement with the following syntax:

```
ALTER TABLE table_name
ADD column_name column_data_type column_constraints
```

For example, if *Antique Opticals* wanted to add a telephone number column to the producer table, they would use

```
ALTER TABLE producer
ADD producer_phone char (12);
```

To add more than one column at the same time, simply separate the clauses with commas:

```
ALTER TABLE producer
ADD producer_phone char (12),
ADD studio_street char (30),
ADD studio_city char (15),
ADD studio_state char (2),
ADD studio_zip char (10);
```

Adding Table Constraints

You can add table constraints, such as foreign keys, at any time. To do so, include the new constraint in an ALTER TABLE statement:

```
ALTER TABLE table_name
ADD table_constraint
```

Assume, for example, that *Antique Opticals* created a new table named *states*, and included in it all the two-character US state abbreviations. The company would then need to add a foreign key reference to that table from the customer, distributor, and producer tables:

```
ALTER TABLE customer
ADD FOREIGN KEY customer2states (customer_state)
    REFERENCES states (state_name);

ALTER TABLE distributor
ADD FOREIGN KEY distributor2states (distributor_state)
    REFERENCES states (state_name);

ALTER TABLE producer
ADD FOREIGN KEY producer2states (studio_state)
    REFERENCES states (state_name);
```

When you add a foreign key constraint to a table, the DBMS verifies that all existing data in the table meet that constraint. If they do not, the ALTER TABLE statement will fail.

Modifying Columns

You can modify columns by changing any characteristic of the column, including the data type, size, and constraints.

Changing Column Definitions

To replace a complete column definition, use the MODIFY command with the current column name and the new column characteristics. For example, to change the customer number in *Antique Opticals' customer* table from an integer to a character column, they would use

```
ALTER TABLE customer
MODIFY customer_numb char (4);
```

When you change the data type of a column, the DBMS will attempt to convert any existing values to the new data type. If the current values cannot be converted, then the table modification will not be performed. In general, most columns can be converted to character. However, conversions from a character data type to numbers, dates, and/or times require that existing data represent legal values in the new data type.

Given that the DBMS converts values whenever it can, changing a column data type may seem like a simple change, but it isn't. In this particular example, the customer number is referenced by foreign keys and therefore the foreign key columns must be modified as well. You need to remove the foreign key constraints, change the foreign key columns, change the

primary key column, and then add the foreign key constraints back to the tables containing the foreign keys. Omitting the changes to the foreign keys will make it impossible to add any rows to those foreign key tables because integer customer numbers will never match character customer numbers. Moral to the story: before changing column characteristics, consider the effect of those changes on other tables in the database.

Changing Default Values

To add or change a default value only (without changing the data type or size of the column), include the DEFAULT keyword:

```
ALTER TABLE order_line
MODIFY discount_percent DEFAULT 0;
```

Changing Null Status

To switch between allowing nulls and not allowing nulls, without changing any other characteristics, add NULL or NOT NULL as appropriate:

```
ALTER TABLE customer
MODIFY customer_zip NOT NULL;
```

or

```
ALTER TABLE customer
MODIFY customer_zip NULL;
```

Changing Column Constraints

To modify a column constraint without changing any other column characteristics, include a CHECK clause:

```
ALTER TABLE item
MODIFY retail_price
    CHECK (VALUE >= 12.95);
```

Deleting Table Elements

You can also delete structural elements from a table as needed, without deleting the entire table:

- To delete a column:

  ```
  ALTER TABLE order_line
  DELETE line_cost;
  ```

- To delete a CHECK table constraint (a CHECK that has been applied to an entire table, rather than to a specific column):

  ```
  ALTER TABLE customer
  DELETE CHECK;
  ```

- To remove the UNIQUE constraint from one or more columns:

```
ALTER TABLE item
DELETE UNIQUE (title);
```

- To remove a table's primary key:

```
ALTER TABLE customer
DELETE PRIMARY KEY;
```

  Although you can delete a table's primary key, keep in mind that if you do not add a new one, you will not be able to modify any data in that table.

- To delete a foreign key:

```
ALTER TABLE item
DELETE FOREIGN KEY item2distributor;
```

Renaming Table Elements

You can rename both tables and columns:

- To rename a table, place the new table name after the RENAME keyword:

```
ALTER TABLE order_line
RENAME line_item;
```

- To rename a column, include both the old and new column names, separated by the keyword TO:

```
ALTER TABLE item
RENAME title TO item_title;
```

Deleting Database Elements

To delete a structural element from a database, you "drop" the element. For example, to delete a table you would type:

```
DROP TABLE table_name
```

Dropping a table (or any database structural element, for that matter) is irrevocable. In most cases, the DBMS will not bother to ask you "are you sure?" but will immediately delete the structure of the table and all of its data, if it can. A table deletion will fail, for example, if it has foreign keys referencing it and one or more of the foreign key constraints contain ON DELETE RESTRICT. Dropping a table or view will also fail if the element being dropped is currently in use by another user.

Note: There is one exception to the irrevocability of a delete. If an element is deleted during a program-controlled transaction and the transaction is rolled back, the deletion will be undone. Undoing transactions is covered in Chapter 22.

You can remove the following elements from a database with the DROP statement:

- Tables
- Views

```
DROP VIEW view_name
```

- Indexes

```
DROP INDEX index_name
```

- Domains

```
DROP DOMAIN domain_name
```

For Further Reading

The Web sites in the following citations provide extensive SQL tutorials that you can use as references when needed.

SQLcourse.com. Creating Tables. http://www.sqlcourse.com/create.html

tutorialspoint. SQL – Constraints. http://www.tutorialspoint.com/sql/sql-constraints.htm

tutorialspoint. SQL – DROP or DELETE Table. http://www.tutorialspoint.com/sql/sql-drop-table.htm

w3schools.com. The SQL CREATE DATABASE Statement. http://www.w3schools.com/sql/sql_create_db.asp

w3schools.com. The SQL CREATE TABLE Statement. http://www.w3schools.com/sql/sql_create_table.asp

Page left intentionally blank

Chapter 12 Using CASE Tools for Database Design

A *CASE* (computer-aided software engineering) tool is a software package that provides support for the design and implementation of information systems. By integrating many of the techniques used to document a system design—including the data dictionary, data flows, and entity relationships—CASE software can increase the consistency and accuracy of a database design. They can also ease the task of creating the diagrams that accompany a system design.

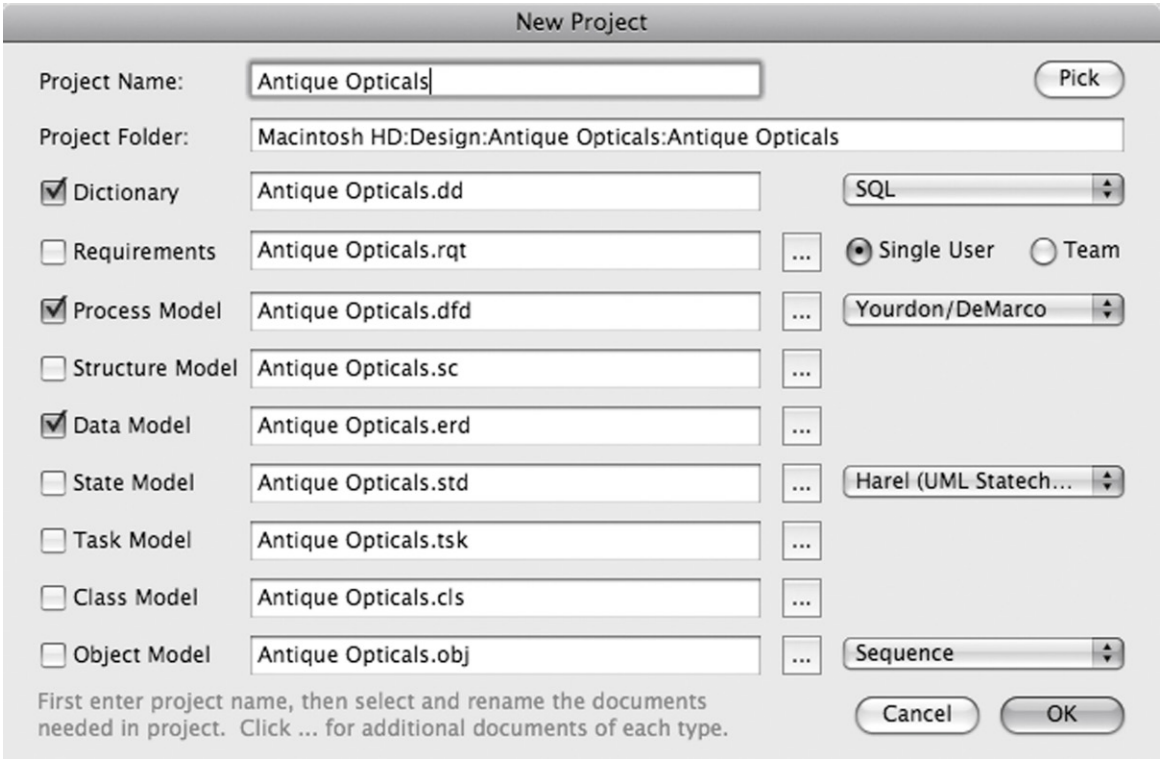
There are many CASE tools on the market. The actual "look" of the diagrams is specific to each particular package. However, the examples presented in this chapter are typical of the capabilities of most CASE tools.

Note: The specific CASE software used in Chapters 13–15 is MacA&D by Excel Software (www.excelsoftware.com). (There's also a Windows version.) Other such packages that are well suited to database design include Visio (www.microsoft.com) and Visible Analyst (www.visible.com).

A word of warning is in order about CASE tools before we proceed any further. A CASE tool is exactly that—a tool. It can document a database design and it can provide invaluable help in maintaining the consistency of a design. Although some current CASE tools can verify the integrity of a data model, they cannot design the database for you. There is no software in the world that can examine a database environment and identify the entities, attributes, and relationships that should be represented in a database. The model created with CASE software is therefore only as good as the analysis of the database environment provided by the people using the tool.

CASE Capabilities

Most CASE tools organize the documents pertaining to a single system into a "project." As you can see in Figure 12.1, by default, a typical project supports the following types of documents:



■ **FIGURE 12.1 CASE software project documents.**

- Data dictionary: In most CASE tools, the data dictionary forms the backbone of the project, providing a single repository for all processes, entities, attributes, and domains used anywhere throughout the project.
- Requirements: CASE tool requirements documents store the text descriptions of product specifications. They also make it possible to arrange requirements in a hierarchy, typically from general to specific.
- Data flow diagrams (DFD): As you read in Chapter 4, DFDs document the way in which data travel throughout an organization, indicating who handles the data. Although it isn't necessary to create a DFD, if your only goal with the project is to document a database design DFDs

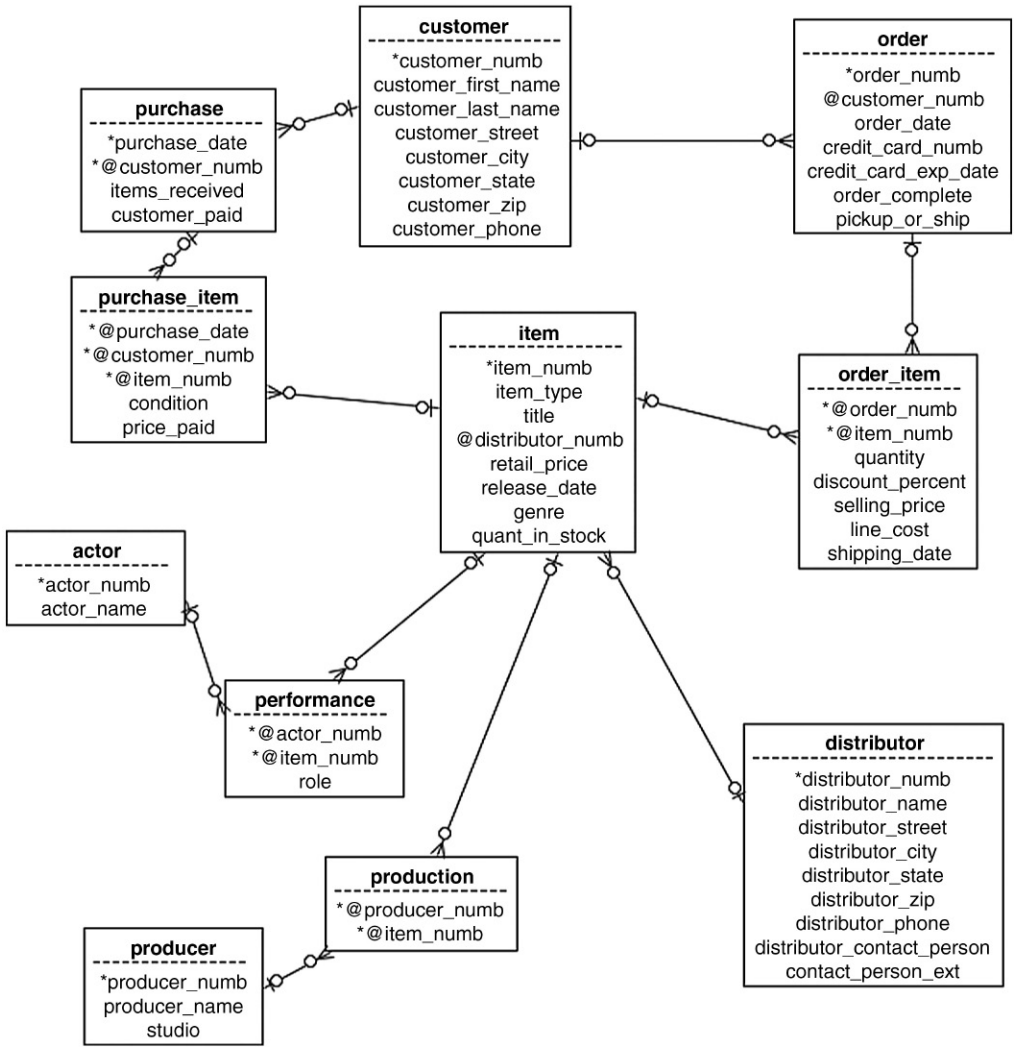
can often be useful in documenting the relationships between multiple organization units and the data they handle. Data flow diagrams can, for example, help you determine whether an organization needs a single database or a combination of databases.

- Structure charts: Structure charts are used to model the structure of application programs that will be developed using structured programming techniques. The charts show the relationship between program modules.
- Data models: Data models are the ER diagrams about which you have been reading. The ER diagram on which the examples in this chapter are based can be found in Figure 12.2.
- Screen prototypes: Drawings of sample screen layouts are typically most useful for documenting the user interface of application programs. However, they can also act as a crosscheck to ensure that a database design is complete by allowing you to verify that everything needed to generate the sample screen designs is present in the database.
- State models: State models, documented in state transition diagrams, indicate the ways in which data change as they move through the information system.
- Task diagrams: Task diagrams are used to help plan application programs in which multiple operations (tasks) occur at the same time. They are therefore not particularly relevant to the database design process.
- Class diagrams: Class diagrams are used when performing object-oriented, rather than structured, analysis and design. They can also be used to document an object-oriented database design.
- Object diagrams: Object diagrams are used during object-oriented analysis to indicate how objects communicate with one another by passing messages.

Many of the diagrams and reports that a CASE tool can provide are designed to follow a single theoretical model. For example, the ER diagrams that you have seen earlier in this book might be based on the Chen model, or the Information Engineering model. Any given CASE tool will support some selection of diagramming models. You must therefore examine what a particular product supports before you purchase it to ensure that it provides exactly what you need.

ER Diagram Reports

In addition to providing tools for simplifying the creation of ER diagrams, many CASE tools can generate reports that document the contents of an ERD. For example, in Figure 12.3, you can see a portion of a report that



■ **FIGURE 12.2 ER diagram created with the sample CASE tool for *Antique Opticals.***

provides a description of each entity and its attributes, including the attribute's data type. The "physical" line contains the name that the database element will have in SQL CREATE TABLE statements; it can be different than the element's data dictionary entry name.[1] For many designers, this type of report actually constitutes a paper-based data dictionary.

[1]The ## at the end of each Physical line is a terminator used by the CASE tool; it isn't part of the name of the entity or attribute.

```
************************************************

Entity: actor

Language: SQL
Physical: actor##

Attributes:

...*actor_numb
Language: SQL
DataType: INTEGER
Physical: actor_numb##

...actor_name
Language: SQL
DataType: long_name
Physical: actor_name##


************************************************
```

■ **FIGURE 12.3 Part of an entity specification report.**

A CASE tool can also translate the relationships in an ER diagram into a report, such as that in Figure 12.4. The text in the report describes the *cardinality* of each relationship in the ERD (whether the relationship is one-to-one, one-to-many, or many-to-many) and can therefore be very useful for pinpointing errors that may have crept into the graphic version of the diagram.

Data Flow Diagrams

There are two widely used styles for DFDs: Yourdon/DeMarco (which has been used throughout this book), and Gene & Sarson.

The Yourdon/DeMarco style, which you can see in Figure 12.5, uses circles for processes. (This particular example is for a small taxi company that rents its cabs to drivers.) Data stores are represented by parallel lines. Data flows are curved or straight lines, with labels that indicate the data that are moving along that pathway. External sources of data are represented by rectangles.

In concept, the Gene & Sarson style is very similar: It varies primarily in style. As you can see in Figure 12.6, the processes are round-cornered rectangles as opposed to circles. Data stores are open-ended rectangles rather than parallel lines. External sources of data remain as rectangles and data flows use only straight lines. However, the concepts of numbering the processing and exploding each process with a child diagram that shows further detail is the same, regardless of which diagramming style you use.

actor is associated with zero or more instances of performance.
performance is associated with zero or one instance of actor.

customer is associated with zero or more instances of order.
order is associated with zero or one instance of customer.

customer is associated with zero or more instances of purchase.
purchase is associated with zero or one instance of customer.

distributor is associated with zero or more instances of item.
item is associated with zero or one instance of distributor.

item is associated with zero or more instances of order_item.
order_item is associated with zero or one instance of item.

item is associated with zero or more instances of performance.
performance is associated with zero or one instance of item.

item is associated with zero or more instances of production.
production is associated with zero or one instance of item.

item is associated with zero or more instances of purchase_item.
purchase_item is associated with zero or one instance of item.

order is associated with zero or more instances of order_item.
order_item is associated with zero or one instance of order.

producer is associated with zero or more instances of production.
production is associated with zero or one instance of producer.

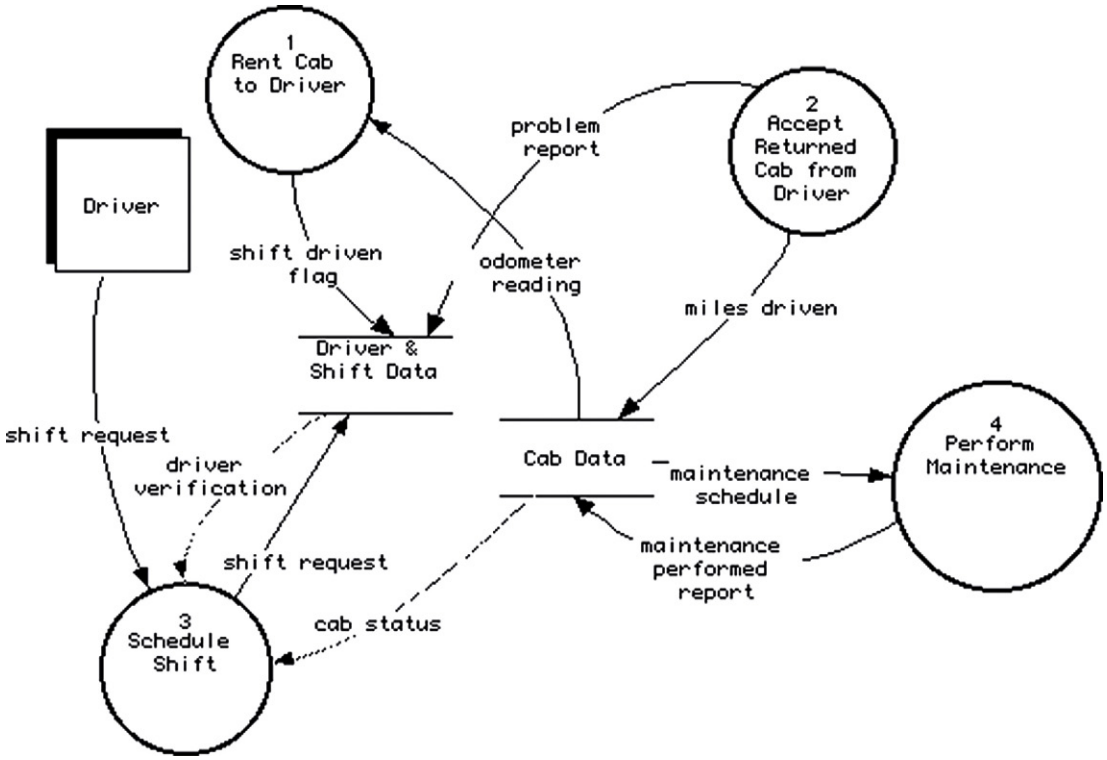
purchase is associated with zero or more instances of purchase_item.
purchase_item is associated with zero or one instance of purchase.

■ **FIGURE 12.4 A relation specification report.**

As mentioned earlier, DFDs are very useful in the database design process for helping a designer to determine whether an organization needs a single, integrated database or a collection of independent databases. For example, it is clear from the taxi company's DFDs that an integrated database is required. Of the four processes shown in the diagram, three use data from both the cab data store and the drive and shift data store. (Only the maintenance process uses just one data store.) You will see examples of using DFDs in this way in one of the case studies in the following three chapters.

The Data Dictionary

From a database designer's point of view, the ER diagram and its associated data dictionary are the two most important parts of CASE software. Since you were introduced to several types of ER diagrams in Chapter 4, we will not repeat them here, but instead focus on the interaction of the diagrams and the data dictionary.

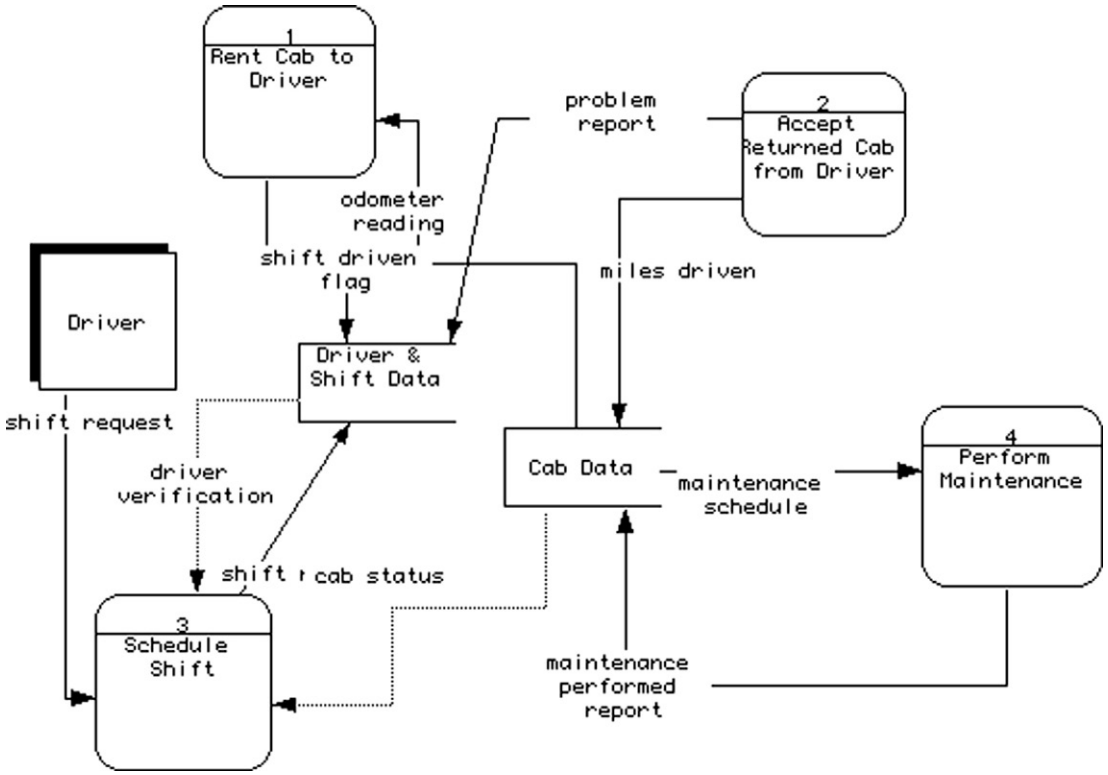


■ **FIGURE 12.5 Yourdon/Marco style DFD.**

A data dictionary provides a central repository for documenting entities, attributes, and domains. In addition, by linking entries in the ER diagram to the data dictionary you can provide enough information for the CASE tool to generate the SQL CREATE statements needed to define the structure of the database.

The layout of a data dictionary varies with the specific CASE tool, as does the way in which entries are configured. In the CASE tool used for examples in this chapter, entities are organized alphabetically, with the attributes following the entity name. Entity names are red; attributes are blue. (Of course, you can't see the colors in this black-and-white book, so you'll have to take my word for it.) Domain names appear alphabetically among the entities. Each relationship in the related ERD also has an entry. Because each item name begins with "Relation," all relationship entries sort together in the data dictionary.

When you select an entity name, the display shows the entity's name, composition (the attributes in the entity), definition (details needed to generate

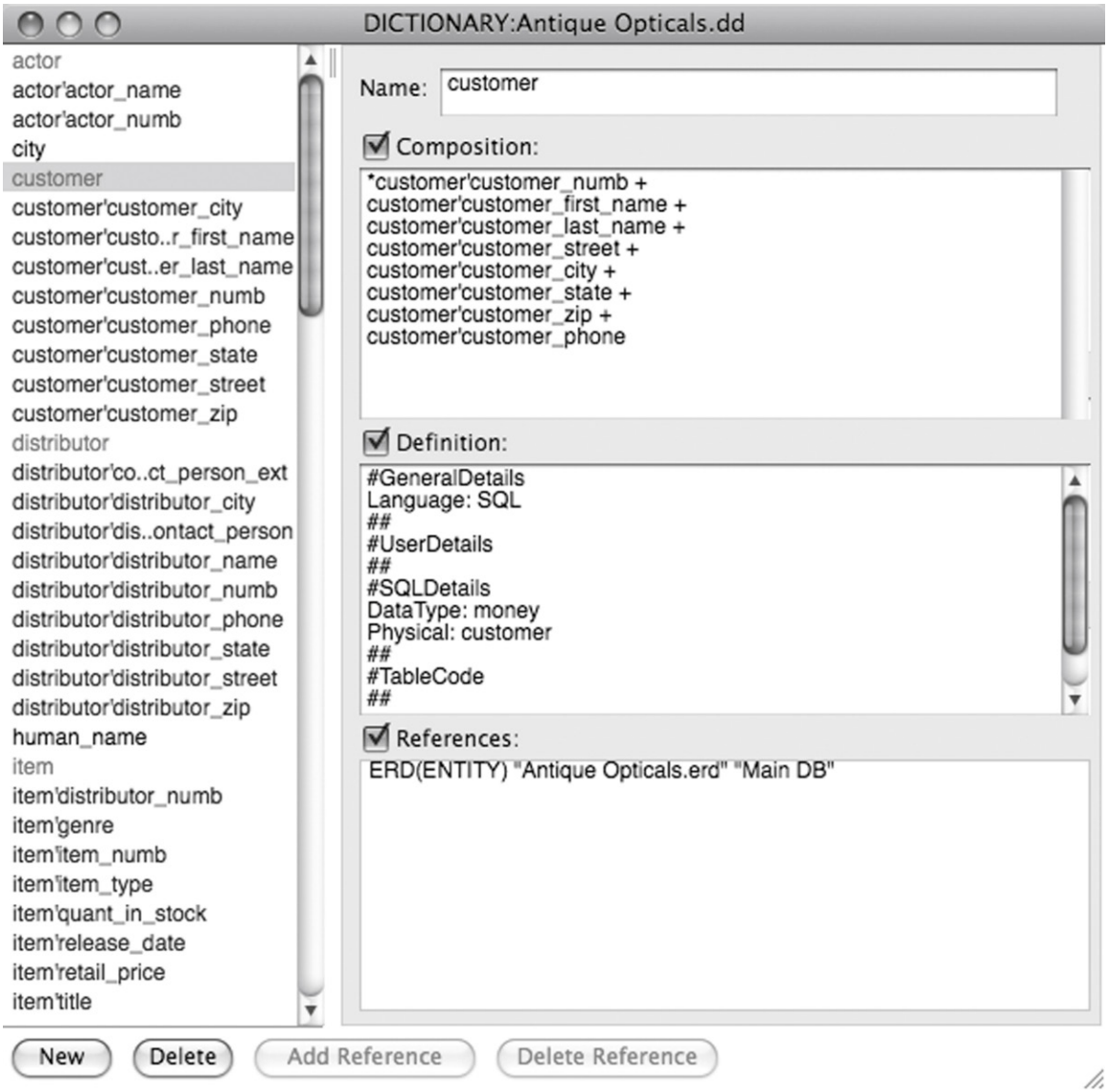


■ **FIGURE 12.6 Gene & Sarson style DFD.**

SQL, and so on), and type of database element (in the References section). Figure 12.7, for example, shows the information stored in the data dictionary for *Antique Opticals' customer* relation. All of the information about the entity (and all other entries, for that matter) is editable, but because the format is specific to the CASE tool, be careful when making changes unless you know exactly how entries should appear.

Attribute entries (Figure 12.8) are similar to entity entries, but have no data in the composition section. Attribute definitions can include the attribute's data type, a default value, and any constraints that have been placed on that attribute. In most cases, these details are entered through a dialog box, relieving the designer of worrying about specific SQL syntax.

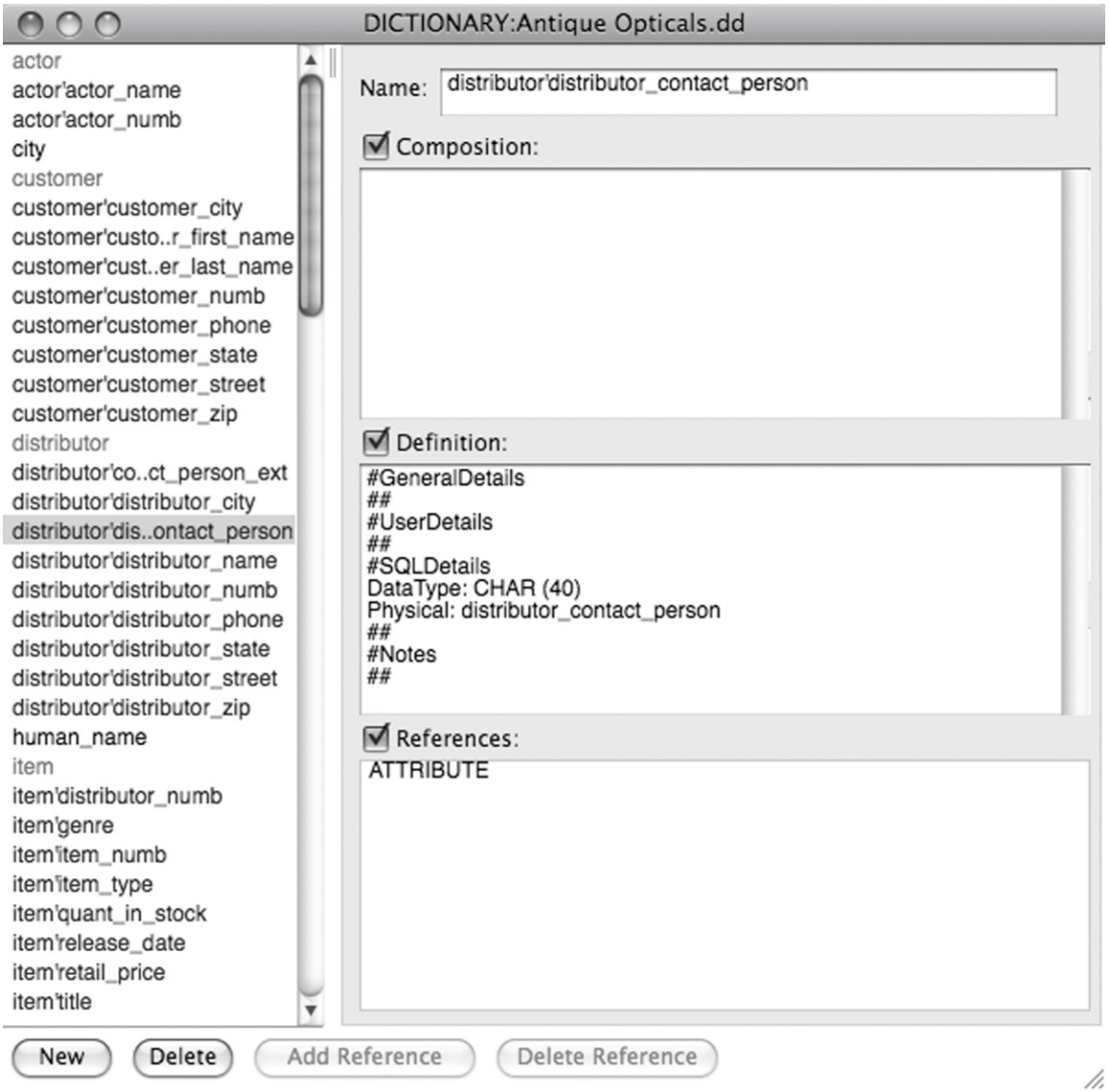
Relationships (Figure 12.9) are named by the CASE tool. Notice that the definition indicates which entities the relationship relates, as well as which



■ **FIGURE 12.7 Definition of an entity in a data dictionary window.**

is at the "many" end of the relationship (the child) and which is at the "one" end (the parent).

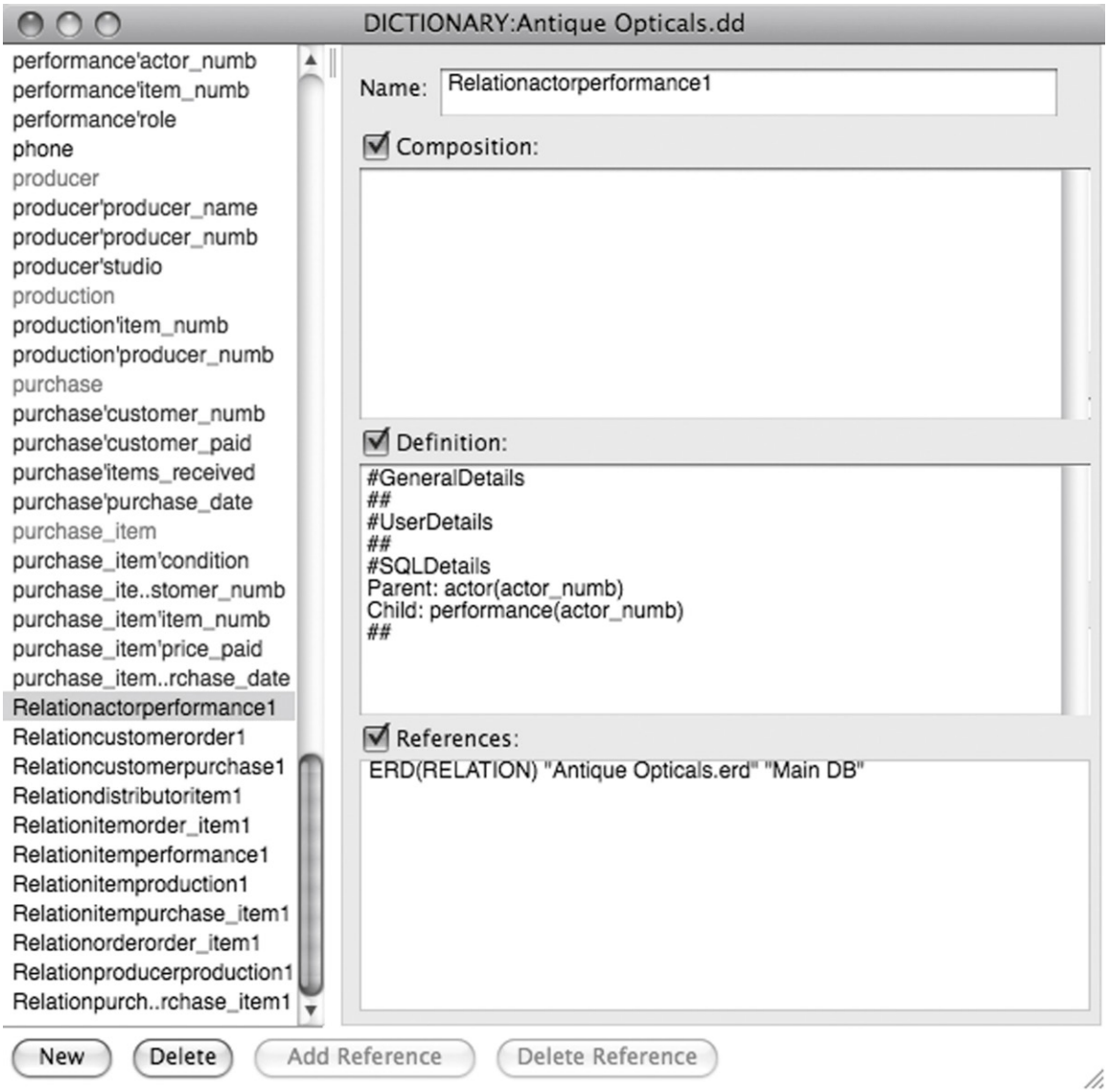
Many relational DBMSs now support the definition of custom domains. Such domains are stored in the data dictionary (Figure 12.10), along with their definitions. Once a domain has been created and is part of the data



■ **FIGURE 12.8 Definition of an attribute in a data dictionary window.**

dictionary, it can be assigned to attributes. If a database administrator needs to change a domain, it can be changed once in the data dictionary and propagated automatically to all attributes entries that use it.

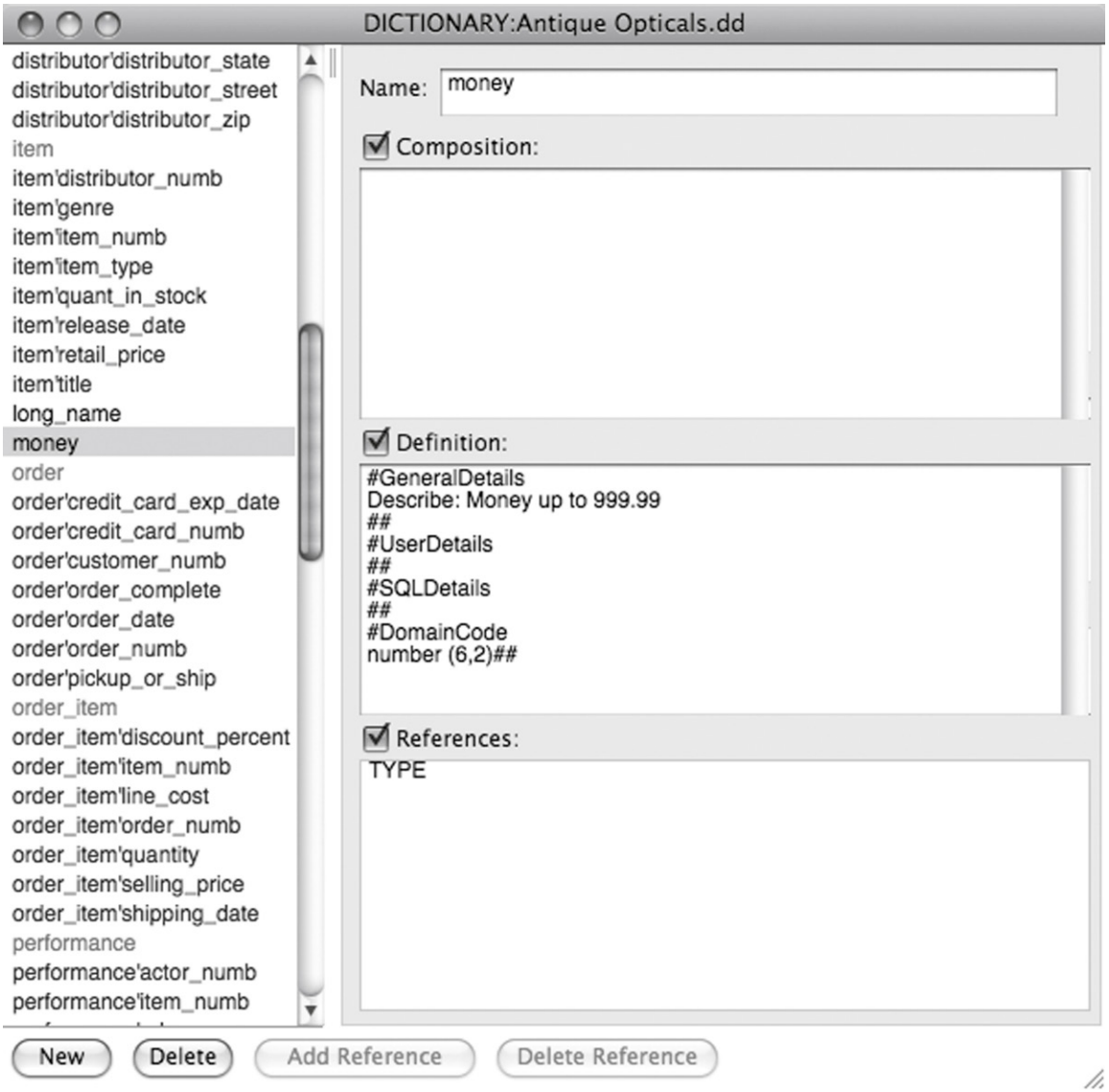
The linking of data dictionary entries to an ER diagram has another major benefit: The data dictionary can examine its entries and automatically identify



■ **FIGURE 12.9 Data dictionary entry for a relationship between two entities in an ERD.**

foreign keys. This is yet another way in which the consistency of attribute definitions enforced by a CASE tool's data dictionary can support the database design process.

Note: Mac A&D is good enough at identifying foreign keys to pick up concatenated foreign keys.



■ **FIGURE 12.10 Data dictionary entry for custom domain.**

Keep in mind that a CASE tool is not linked dynamically with a DBMS. Although data definitions in the data dictionary are linked to diagrams, changes made to the CASE tool's project will not affect the DBMS. It is up to the database administrator to make the actual changes to the database.

Code Generation

The end product of most database design efforts is a set of SQL CREATE TABLE commands. If you are using CASE software, and the software contains a complete data dictionary, then the software can generate the SQL for you. You will typically find that a given CASE tool can tailor the SQL syntax to a range of specific DBMSs. In most cases, the code will be saved in a text file, which you can then use as input to a DBMS.

Note: Most of today's CASE tools will also generate XML for you. XML provides a template for interpreting the contents of files containing data and therefore isparticularly useful when you need to transfer schemas and data between DBMSs with different SQL implementations, or between DBMSs that do not use SQL at all. XML has become so important for data exchange that it is covered in Chapter 26.

The effectiveness of the SQL that a CASE tool can produce, as you might expect, depends on the completeness of the data dictionary entries. To get truly usable SQL, the data dictionary must contain:

- Domains for every attribute.
- Primary key definitions (created as attributes, are added to entities in the ER diagram).
- Foreign key definitions (created as attributes, are added to entities in ER diagram or by the CASE tool after the ER diagram is complete).
- Any additional constraints that are to be placed on individual attributes or on the entity as a whole.

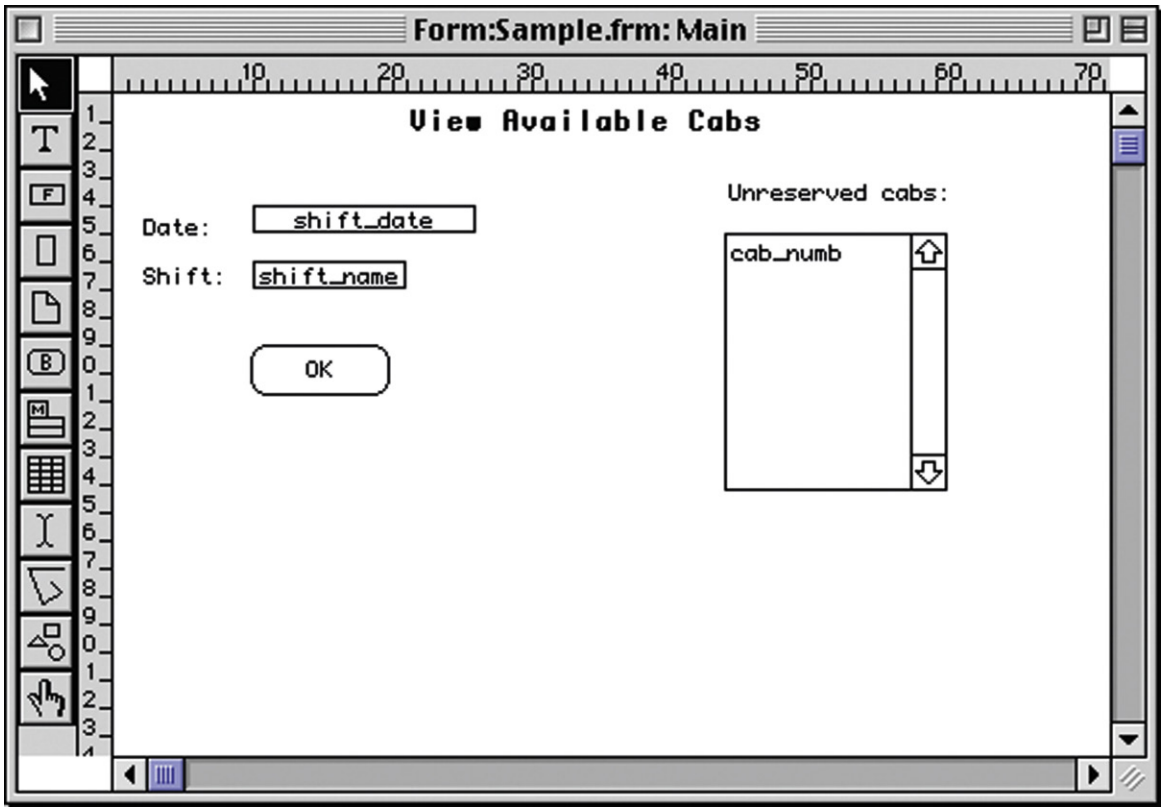
Sample Input and Output Designs

Sample input and output designs form part of the system documentation, especially in that they help document requirements. They can also support the database designer by providing a way to double-check that the database can provide all the data needed by application programs. Many CASE tools therefore provide a way to draw and label sample screen and report layouts.

Most of today's CASE tools allow multiple users to interact with the same project. This means that interface designers can work with the same data dictionary that the systems analysts and database designers are building, ensuring that all the necessary data elements have been handled.

For example, one of the most important things that the person scheduling cab reservations for the taxi company needs to know is which cabs are not reserved for a given date and shift. A sample screen, such as that

in Figure 12.11, will do the trick.[2] The diagram shows what data the user needs to enter (the shift date and the shift name). It also shows the output (cab numbers). The names of the fields on the sample screen design can be linked to the data dictionary.



■ **FIGURE 12.11 Sample screen design.**

A CASE tool can be used to model an entire application program. The "browse" tool at the very bottom of the tool bar in Figure 12.11 switches into browse mode, in which buttons and menus become active. Users can make choices from pull-down menus that can be linked to other forms. Buttons can also trigger the opening of other forms. Users can click into data entry

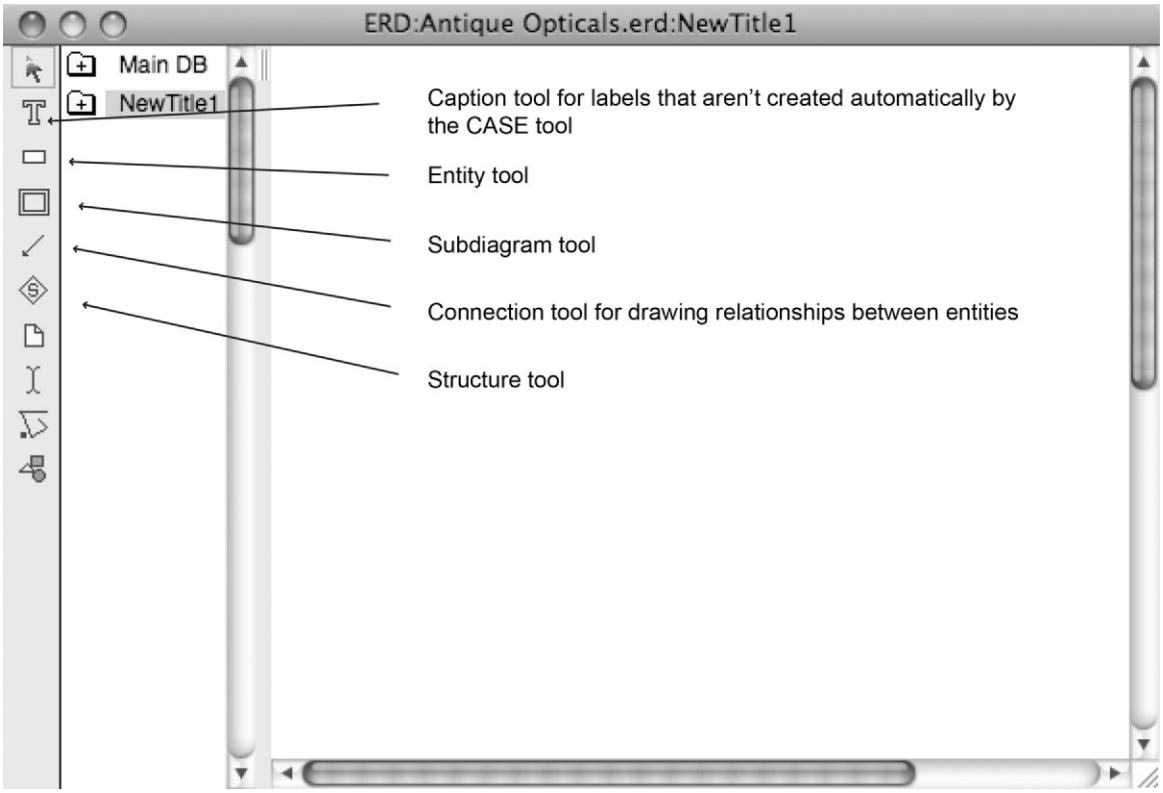
[2]In the interest of complete disclosure, you should know that when Mac A&D was ported from Mac OS 9 to Mac OS X, the screen and report design module wasn't included. (It is now available as a stand-alone product.) Therefore, the sample screen designs that you will see in this chapter and in Chapter 13 are from an older version of the product.

fields and tab between fields. Users can therefore not only see the layout and output screen and documents, but also navigate through an application.

The Drawing Environment

To this point, you've been reading about the way in which the functions provided by CASE software can support the database design effort. In this last section we will briefly examine the tools you can expect to find as part of CASE software, tools with which you can create the types of documents you need.

Because many of the documents you create with CASE software are diagrams, the working environment of a CASE tool includes a specialized drawing environment. For example, in Figure 12.12, you can see the drawing tools provided by the sample CASE tool for creating ER diagrams. (Keep in mind that each CASE tool will differ somewhat in the precise layout of its drawing tool bars, but the basic capabilities will be similar.)



■ **FIGURE 12.12 Example CASE tool drawing environment for ER diagrams.**

The important thing to note is that the major shapes needed for the diagrams—for ER diagrams, typically just the entity and relationship line—are provided as individual tools. You therefore simply click the tool you want to use in the tool bar and draw the shape in the diagram, much like you would if you were working with a general-purpose object graphics program.

For Further Reading

To learn more about the Yourdon/DeMarco method of structure analysis using DFDs, see either of the following:

DeMarco, T., Flauger, P.J., 1985. Structured analysis and system specification. Prentice-Hall.

Yourdon, E., 2000. Modern structured analysis. Prentice Hall PTR.

Chapter 13 Database Design Case Study #1: Mighty-Mite Motors

It is not unusual for a database designer to be employed to reengineer the information systems of an established corporation. As you will see from the company described in this chapter, information systems in older companies have often grown haphazardly, with almost no planning and integration. The result is a hodgepodge of data repositories that cannot provide the information needed for the corporation to function because they are isolated from one another. In such a situation, it is the job of the database designer to examine the environment as a whole and to focus on the integration of data access across the corporation, as well as the design of one or more databases that will meet individual department needs.

On the bright side, an organization such as Mighty-Mite Motors, which has a history of data processing of some kind, knows quite well what it needs in information systems, even if the employees are unable to articulate those needs immediately. There will almost certainly be a collection of paper forms and reports that the organization uses regularly to provide significant input to the systems design process.

Corporate Overview

Might-Mite Motors, Inc. (MMM) is a closely held corporation, established in 1980, that manufactures and markets miniature rideable motor vehicles for children. Products include several models of cars, trucks, all-terrain vehicles, and trains (see Figure 13.1). Vehicles are powered by car batteries and achieve speed of about 5 mph.

At this time, MMM is organized into three divisions: Product Development, Manufacturing, and Marketing and Sales. Each division is headed by a

Mighty-Mite Motors

Product Catalog

Winter Holiday Season 2020

■ **FIGURE 13.1 Might-Mite Motors product catalog (continues).**

Model #001

001

All Terrain Vehicle: Accelerator in the handgrip lets young riders reach speeds of up to 5 mph. Vehicle stops immediately when child removes his or her hand from the handlegrips. Can carry one passenger up to 65 lbs. **Suggested retail price: $124.95**

Model #002

002

4-Wheel Drive Cruiser: Two-pedal drive system lets vehcile move forward at 2 1/2 mph on hard surfaces, plus reverse. Electronic speed reduction for beginners. Includes one 6v battery and one recharger. Ages 3–7 (can carry two passengers up to 40 lbs each). **Suggested retail price: $249.99**

■ **FIGURE 13.1** (*cont.*)

Model #003

Classic roadster: Sounds include engine start-up, rev, shifting gears, and idle. Two forward speeds—2 1/2 mph and 5 mph; reverses at 2 1/2 mph. High-speed lockout. On/off power pedal. Power-Lock electric brake. Includes two 6v batteries and recharger. Ages 3–7 (carries two passengers up to 60 lbs each). **Suggested retail price: $189.95**

003

Model #004

Sports car #1: Two-forward speeds, 2 1/2 and 5 mph. Reverses at 2 1/2 mph. High-speed lockout. Power-Lock electric brake. Includes two 6v batteries and one recharger. Ages 3–6 (carries two passengers up to 90 lbs. total).
Suggested retail price: $249.95

004

Model #005

Sports car #2: Phone lets child pretend to talk while he or she drives. Two forward speeds—2 1/2 mph and 5 mph; reverses at 2 1/2 mph. High-speed lockout. Power-Lock electric brake. Includes two 6v batteries and one recharger. Ages 3–6 (carries two passengers up to 90 lbs. total).
Suggested retail price: $249.95

005

■ **FIGURE 13.1** (***cont.***)

Model #008

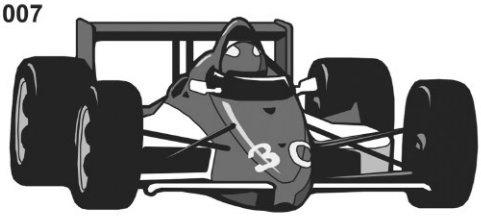
006

Turbo-Injected Porche: Working stick shift—3 mph and 5 mph froward; 3 mph reverse. High-speed lockout. Adjustable seat. Doors, trunk, and hood open. Simulated car phone. Includes one 18v battery and recharger. Ages 3–8 (carries two passengers up to 120 lbs. total)
Suggested retail price: $299.95

Model #007

007

Indy car: Dual motors for cruising on a variety of surfaces, even up hills. Two forward speeds (2 1/2 and 5 mph), plus reverse (2 1/2 mph). Adjustable seat. Includes two 6v batteries and recharger. Ages 3–7 (carries on passenger up to 80 lbs.)
Suggested retail price: $269.95



Model #008

008

2-Ton Pickup: Metallic teal color. Simulated chrome engine covers and headlight with over-sized wheels. 2 1/2 mph forward speed. Includes on 6v battery and recharger. Ages 3–7 (carries one passenger up to 65 lbs).
Suggested retail price: $189.95

■ **FIGURE 13.1** (***cont.***)

Model #008

009

Santa Fe Train: Soundly engineered for a little guy or gal. A hand-operated on/off button controls the 6v battery-operated motor. Reaches speeds to 5 mph. Includes a battery powered "whoo whoo" whistle to greet passersby. Ride on 76" x 168" oval track (sold separately) or on carpet or sidewalk, indoors or outdoors. Plastic body and flooboard; steel axles and coupling pins. Brigh red, blue and yellow body featrues a large lift-up seat and trailing car for storage. Includes battery and chargers. Ages 3−6.
Suggested retail price: $159.95

Model #010

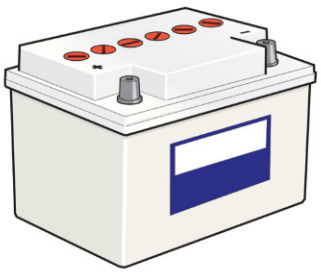
Oval track: Measures 76" × 168."
Suggested retail price: $39.95

Model #011

6 Pieces Straight track: Six straight track section 19" each (total 109")..
Suggested retail price: $19.95

Model #012

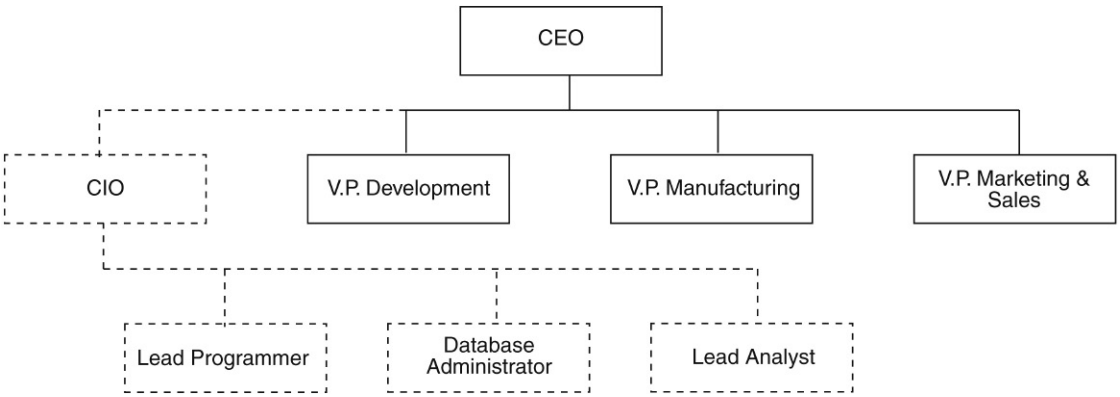
012



Rechargable battery (6v): For use with 6v or 12v vehicles. For 12v vehicles, use two. To charge, use charger included with vehicle.
Suggested retail price: $27.95

■ **FIGURE 13.1** (***cont.***)

vice president, who reports directly to the CEO. (An organization chart appears in Figure 13.2. The solid lines represent existing positions; dotted lines represent positions that will be added as part of the reengineering project.) All these divisions are housed in a single location that the corporation owns outright.



■ **FIGURE 13.2 Might-Mite Motors organization chart.**

Product Development Division

The Product Development division is responsible for designing and testing new and redesigned products. The division employs design engineers who use computer-aided design (CAD) software to prepare initial designs for new or redesigned vehicles. Once a design is complete, between one and 10 prototypes are built. The prototypes are first tested in-house using robotic drivers/passengers. After refinement, the prototypes are tested by children in a variety of settings. Feedback from the testers is used to refine product designs and to make decisions about which designs should actually be manufactured for mass marketing.

Manufacturing Division

The Manufacturing division is responsible for producing product for mass market sales. Manufacturing procures its own raw materials and manages its own operations, including personnel (hiring, firing, scheduling) and assembly line management. Manufacturing maintains the inventory of products ready for sale. It also handles shipping of products to resellers, based on sales information received from Marketing and Sales.

Marketing and Sales Division

MMM sells directly to toy stores and catalog houses; the corporation has never used distributors. Marketing and Sales employs a staff of 25 sales

people who make personal contacts with resellers. Salespeople are responsible for distributing catalogs in their territories, visiting and/or calling potential resellers, and taking reseller orders. Order accounting is handled by Marketing and Sales. As noted earlier, Marketing and Sales transmits shipping information to Manufacturing, which takes care of actual product delivery.

Current Information Systems

MMM's information systems are a hodgepodge of computers and applications that have grown up with little corporate planning. The Product Development division relies primarily on stand-alone CAD workstations. In contrast to the sophistication of the CAD machines, testing records are kept and analyzed manually. Product Development employs product designers (some of whom function as project leaders) and clerical support staff, but no information systems personnel. Attempts to have clerical staff develop simple database applications to store data about children who test new and redesigned products and the results of those tests have proved futile. It has become evident that Product Development needs information systems professionals, and although the division is willing to hire IT staff, corporate management has decided to centralize the IT staff, rather than add to a decentralized model.

Manufacturing uses a stand-alone server to track purchases and inventory levels of raw materials, personnel scheduling, manufacturing line scheduling, and finished product inventory. Each of the applications running on the server were custom-written by outside consultants in COBOL, many years ago; the most significant maintenance they have had was when they were ported from the department's original minicomputer to the server, about 15 years ago. The data used by a Manufacturing application are contained in files that do not share information with any of the other applications. Manufacturing employs one COBOL programmer, who will be retiring shortly, and a system administrator. Although the programmer is talented, the most he can do is fix superficial user interface issues and repair corrupted data files; he was not part of the original program development and does not understand the functioning of much of the application code, which was poorly written and even more poorly documented. The applications no longer meet the needs of the Manufacturing division and management has determined that it isn't cost effective to write new applications to access the existing data files.

Marketing and Sales, which wasn't computerized until 1987, has a local area network consisting of one server and 15 workstations. The server

provides shared applications, such as word processing and spreadsheets. It also maintains a marketing and sales database that has been developed using a PC-based product. The database suffers from several problems, including a limit of 10 users at one time and concurrency control problems that lead to severe data inconsistencies. The marketing and sales database was developed by the division's two IT employees at the time, both of whom have since left the company. No current staff understands the software. Regardless of the amount of time spent trying to maintain the database, inaccurate data continue to be introduced.

The Marketing and Sales network is not connected to the Internet. Sales people must therefore transmit hard copies of their orders to the central office, where the orders are manually keyed into the existing database. Some of the sales people do have laptop computers, but because the network has no Internet connection, the sales people cannot connect to it when they are out of the office.

Reengineering Project

Because MMM seems to have lost its strategic advantage in the marketplace, the CEO has decided to undertake a major systems reengineering project. The overall thrust of the project is to provide an information system that will support better evaluation of product testing, better analysis of sales patterns, better control of the manufacturing process, and enhanced communications options throughout the corporation. New information systems will be based on a client/server model and include one or more databases running on an Internet-connected network of servers, workstations, and PCs. The ultimate goal is to create a Web site for the company so resellers can place orders online.

New Information Systems Division

The first step in the reengineering project is to establish an information technology division. This new division will also be housed in the corporate headquarters, along with the three existing divisions. To accommodate the new division, MMM will be constructing a 10,000 square foot addition to its building.

MMM is in the process of searching for a Chief Information Officer (CIO). This individual, who will report directly to the CEO, will manage the new division and be responsible for overseeing the reengineering of information systems that will handle all of the corporation's operations, as well as the creation, implementation, and maintenance of the company's Web presence.

All current IT personnel (those who work for the Manufacturing, and Marketing and Sales divisions) will be transferred to the new IT division. The division will hire (either internally or externally) three management-level professionals: a Lead Programmer (responsible for overseeing application development), a Database Administrator (responsible for database design and management), and a Lead Analyst (responsible for overseeing systems analysis and design efforts). The company will investigate various solutions for the development of the Web site, with an eye to determining whether it should be done in-house or outsourced to a consulting firm.

Basic System Goals

The CEO has defined the following goals for the reengineering project:

- Develop a corporation-wide data administration plan that includes a requirements document detailing organizational functions that require technology support and the functions that the reengineered system will provide.
- Provide an application roadmap that documents all application programs that will be needed to support corporate operations.
- Investigate alternatives for developing and hosting a Web site that will allow online orders. Conduct a cost-benefit analysis of those alternatives before beginning development.
- Document all databases to be developed for the corporation. This documentation will include ER diagrams and data dictionaries.
- Create a timeline for the development of applications and their supporting databases.
- Specify hardware changes and/or acquisitions that will be necessary to support the reengineered information systems.
- Plan and execute a security strategy for an expanded corporate network that will include both internal and external users.
- Implement the planned systems.

Note: It is important to keep in mind that the implementation of a Web presence for the company is relatively independent of the database design. The Web application will require the same data as all the other types of data entry. Concerns about hosting and security are rarely the job of the database designer.

Current Business Processes

To aid the systems analysts in their assessment of MMM's information systems needs, the CEO of MMM asked all existing division heads to document the way in which information is currently processed. This

documentation, which also includes some information about what an improved system should do, provides a starting point for the redesign of both business and IT processes.

Sales and Ordering Processes

MMM receives orders at its plant in two ways: either by telephone directly from customers or from members of the sales staff who have visited customers in person. Orders from the remote sales staff usually arrive by fax or overnight courier.

Each order is taken on a standard order form (Figure 13.3). If the order arrives by fax, it will already be on the correct form. Telephone orders are written directly onto the form. Several times a day, a clerk enters the orders into the existing database. Unfortunately, if the sales office is particularly busy, order entry may be delayed. This backup has a major impact on production line scheduling and thus on the company's ability to fill orders. The new information system must streamline the order entry process, including online order entry, electronic transmission of order data from the field, and the direct entry of in-house orders.

The in-house sales staff has no access to the files that show the current finished-goods inventory. They are therefore unable to tell customers when their orders will ship. They can, however, tell customers how many orders are ahead of theirs to be filled and, based on general manufacturing timetables, come up with an approximation of how long it will take to ship a given order. Therefore, another goal of the information systems reengineering project is to provide better company-wide knowledge of how long it will take to fill customer orders.

Manufacturing, Inventory, and Shipping Processes

The MMM Manufacturing division occupies a large portion of the MMM facility. The division controls the actual manufacturing lines (three assembly lines), a storage area for finished goods, a storage area for raw materials, and several offices for supervisory and clerical staff.

The manufacturing process is triggered when a batch of order forms is received each morning by the manufacturing office. The batch consists of all orders that were entered into the sales database the previous working day. A secretary takes the individual order forms, and completes a report summarizing the number ordered by model (Figure 13.4). This report is then given to the Manufacturing Supervisor, whose responsibility it is to schedule which model will be produced on each manufacturing line, each day.

Mighty-Mite Motors

Customer Order Form

Customer #:

Order date:

Name:

Street:

City: State: Zip:

Voice phone #: Fax:

Contact person

First name: Last name:

Item #	Quantity	Unit Price	Line Total
		.	.
		.	.
		.	.
		.	.
		.	.
		.	.

Order total: .

■ **FIGURE 13.3 Mighty-Mite Motors order form.**

Mighty-Mite Motors
Order Summary

MM/DD/YYYY

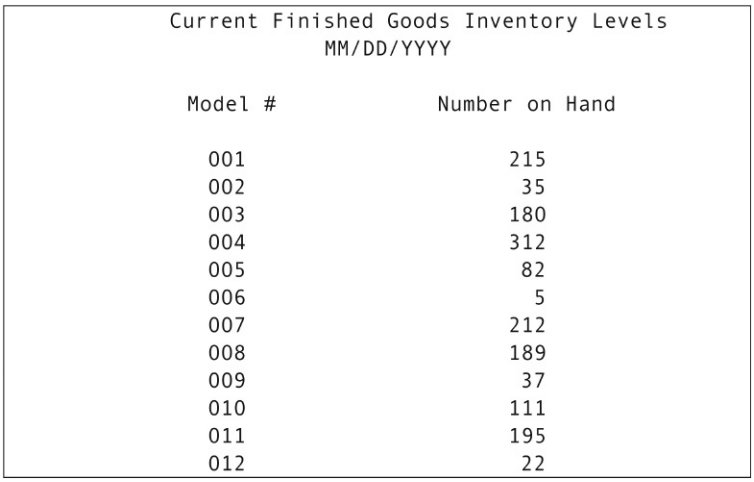
Model #	Quantity Ordered
001	75
002	150
004	80
005	35
008	115
009	25
010	25
011	15

■ **FIGURE 13.4 Mighty-Mite Motors order summary report format.**

The scheduling process is somewhat complex, because the Manufacturing Supervisor must take into account previously placed orders, which have determined the current manufacturing schedule, and current inventory levels, as well as the new orders. The availability of raw materials and the time it takes to modify a manufacturing line to produce a different model also enter into the scheduling decision. This is one function that MMM's management understands will be almost impossible to automate; there is just too much human expertise involved to translate into an automatic process. However, it is vital that the Manufacturing Supervisor have access to accurate, up-to-date information about orders, inventory, and the current time schedule so that judgments can be made based on as much hard data as possible.

As finished vehicles come off the assembly line, they are packed for shipping, labeled, and sent to finished goods storage. Each shipping carton contains one vehicle, which is marked with its model number, serial number, and date of manufacturing. The Shipping Manager, who oversees finished goods storage and shipping, ensures that newly manufactured items are entered into the shipping inventory files.

The Shipping Manager receives customer order forms after the order report has been completed. (Photocopies of the order forms are kept in the Marketing and Sales office as backup.) The orders are placed in a box in reverse chronological order so that the oldest orders can be filled first. The Shipping Manager checks orders against inventory levels by looking at the inventory level output screen (Figure 13.5). If the manager sees that there is enough inventory to fill an order, then the order is given to a shipping clerk for processing. If there isn't enough inventory, then the order is put back in the box, where it will be checked again the following day. Under this system, no partial orders are filled, because they would be extremely difficult



Current Finished Goods Inventory Levels
MM/DD/YYYY

Model #	Number on Hand
001	215
002	35
003	180
004	312
005	82
006	5
007	212
008	189
009	37
010	111
011	195
012	22

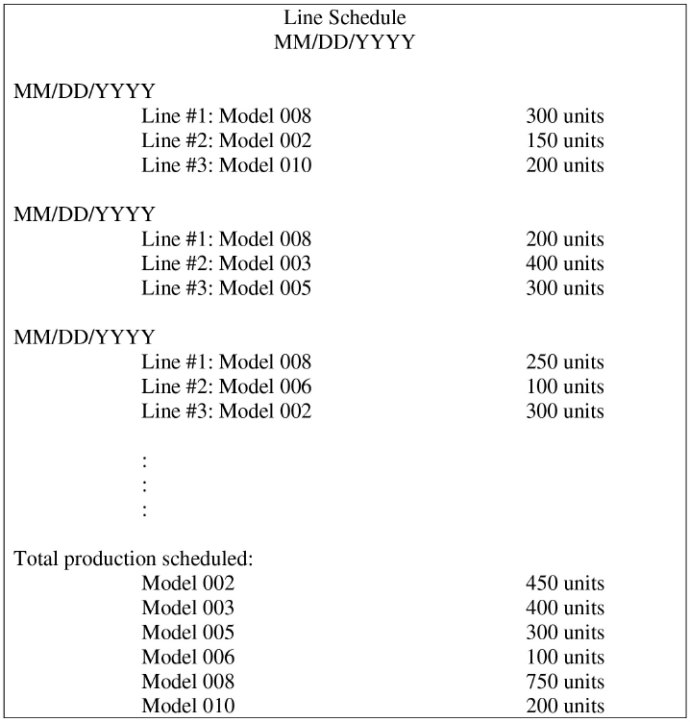
■ **FIGURE 13.5 Mite-Mite Motors inventory screen layout.**

to track. (The reengineered information system should allow handling of partial shipments.)

Shipping clerks are given orders to fill. They create shipping labels for all vehicles that are part of a shipment. The cartons are labeled and set aside for pickup by UPS. The shipping clerks create UPS manifests, ensure that the items being shipped are removed from the inventory file, and return the filled orders to the Shipping Manager. The orders are then marked as filled and returned to Marketing and Sales. The reengineered information system should automate the generation of pick-lists, packing slips, and updating of finished-goods inventory.

MMM's raw materials inventory is maintained on a just-in-time basis. The Manufacturing Supervisor checks the line schedule (Figure 13.6) and the current raw materials inventory (Figure 13.7) daily to determine what raw materials need to be ordered. This process relies heavily on the Manufacturing Supervisor's knowledge of which materials are needed for which model vehicle. MMM's CEO is very concerned about this process because the Manufacturing Supervisor, while accurate in scheduling the manufacturing line, is nowhere near as accurate in judging raw materials needs. The result is that occasionally manufacturing must stop because raw materials have run out. The CEO would therefore like to see ordering of raw materials triggered automatically. The new information system should keep track of the raw materials needed to produce each model and, based on the line schedule and a reorder point established for each item, generate orders for items when needed.

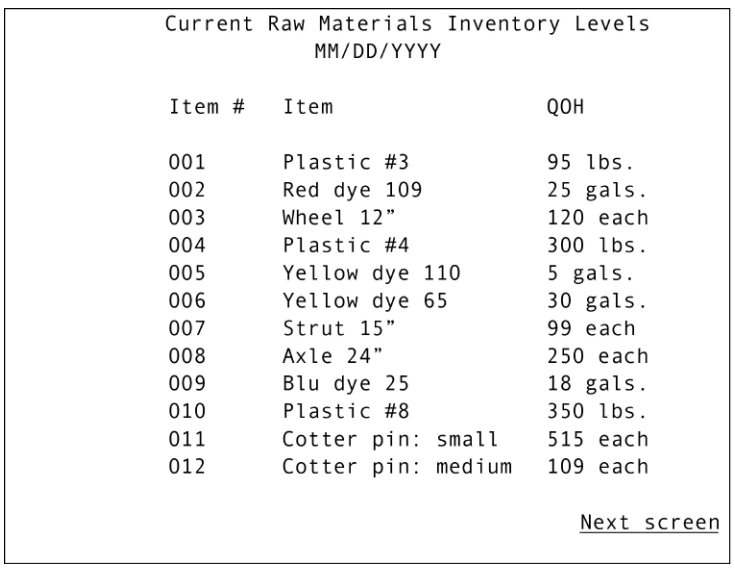
Raw materials are taken from inventory each morning as each manufacturing line is set up for the day's production run. The inventory files are



Line Schedule
MM/DD/YYYY

MM/DD/YYYY	
Line #1: Model 008	300 units
Line #2: Model 002	150 units
Line #3: Model 010	200 units
MM/DD/YYYY	
Line #1: Model 008	200 units
Line #2: Model 003	400 units
Line #3: Model 005	300 units
MM/DD/YYYY	
Line #1: Model 008	250 units
Line #2: Model 006	100 units
Line #3: Model 002	300 units
:	
:	
:	
Total production scheduled:	
Model 002	450 units
Model 003	400 units
Model 005	300 units
Model 006	100 units
Model 008	750 units
Model 010	200 units

■ **FIGURE 13.6 Mighty-Mite Motors line schedule report format.**



Current Raw Materials Inventory Levels
MM/DD/YYYY

Item #	Item	QOH
001	Plastic #3	95 lbs.
002	Red dye 109	25 gals.
003	Wheel 12"	120 each
004	Plastic #4	300 lbs.
005	Yellow dye 110	5 gals.
006	Yellow dye 65	30 gals.
007	Strut 15"	99 each
008	Axle 24"	250 each
009	Blu dye 25	18 gals.
010	Plastic #8	350 lbs.
011	Cotter pin: small	515 each
012	Cotter pin: medium	109 each

Next screen

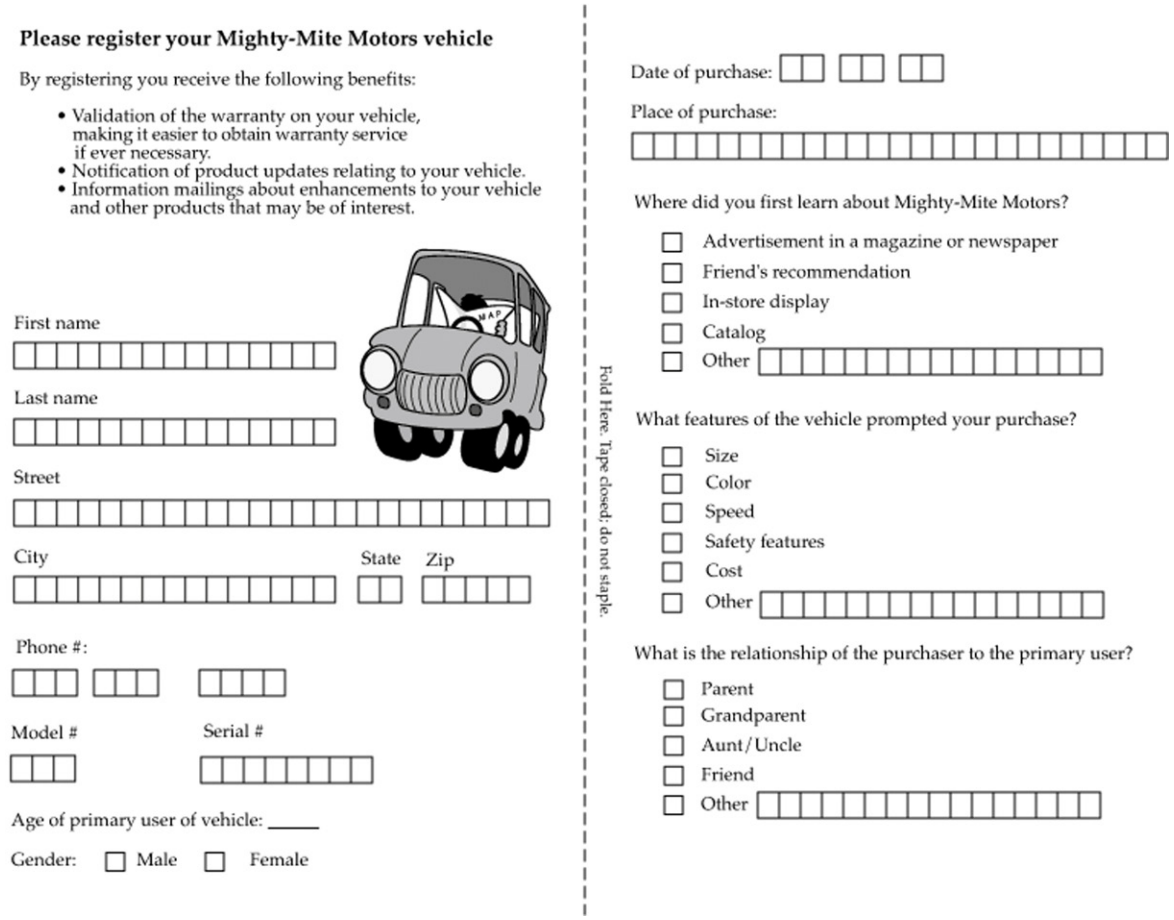
■ **FIGURE 13.7 Mighty-Mite Motors raw materials inventory screen layout.**

modified immediately after all raw materials have been removed from storage for a given manufacturing line. There is no way to automate the reduction of inventory; however, the new information system should make it very easy for nontechnical users to update inventory levels.

Product Testing and Support Function

MMM's top management makes decisions about which model vehicles to produce, based on data from three sources: product testing, customer registrations, and problem reports.

Customer registrations are received on cards packaged with sold vehicles (Figure 13.8). Currently, the registration cards are filed by customer name.



■ **FIGURE 13.8 Mighty-Mite Motors purchase registration form.**

However, MMM would also like access to these data by model and serial number to make it easier to notify customers if a recall occurs. Management would also like summaries of the data by model purchased, age of primary user, gender of primary user, and who purchased the vehicle for the child.

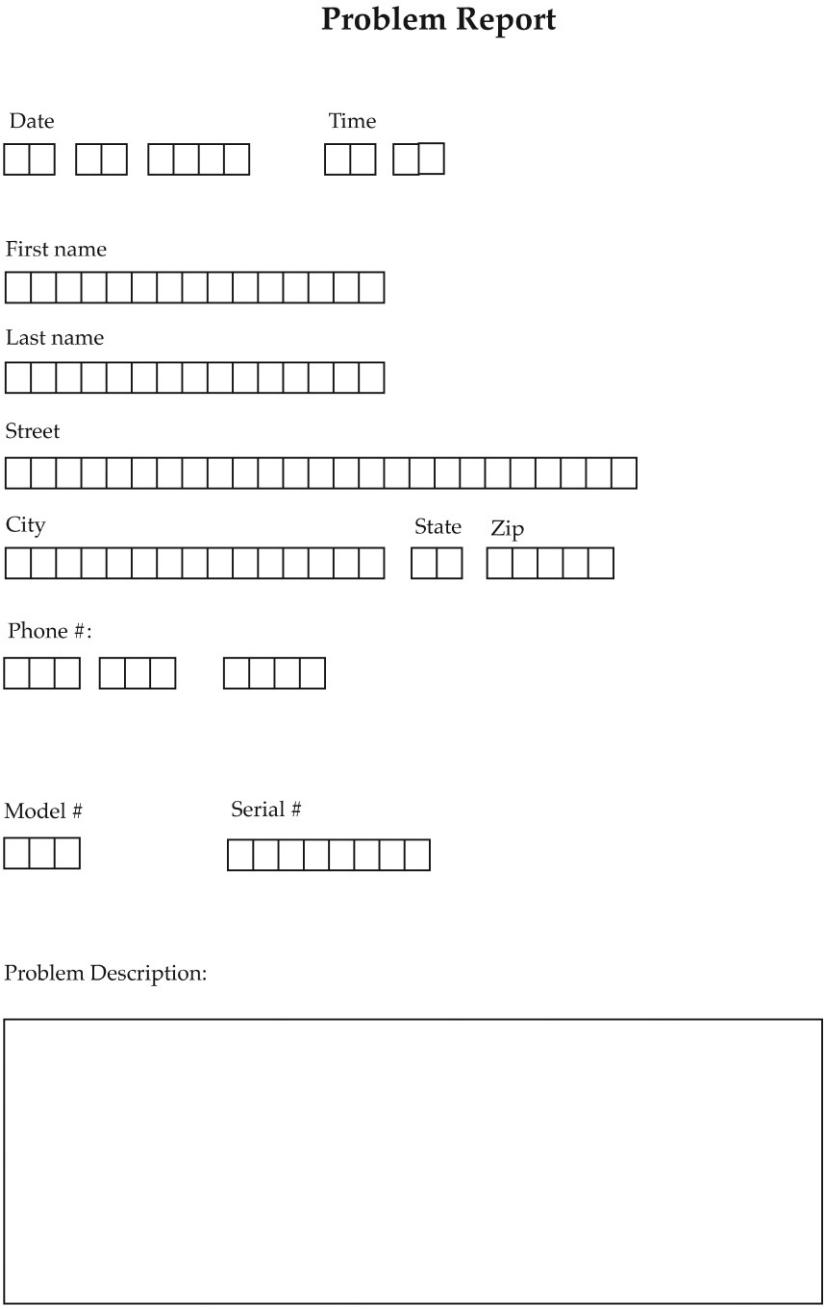
Problem reports (Figure 13.9) are taken by customer support representatives who work within the product testing division. These reports include the serial number and model experiencing problems, along with the date and type of problem. Currently, the problem descriptions are nonstandard, made up of whatever terms the customer support representative happens to use. It is therefore difficult to summarize problem reports to get an accurate picture of which models are experiencing design problems that should be corrected. MMM would therefore like to introduce a standardized method for describing problems, probably through a set of problem codes. The result should be regular reports on the problems reported for each model that can be used to help make decisions about which models to continue, which to discontinue, which to redesign, and which to recall.

MMM does not repair its own products. When a problem report is received, the customer is either directed to return the product to the store where it was purchased for an exchange (during the first 30 days after purchase) or directed to an authorized repair center in the customer's area. In the latter case, the problem report is faxed to the repair center, so that it is waiting when the customer arrives. MMM does not plan to change this procedure because it currently provides quick, excellent service to customers and alleviates the need for MMM to stock replacement parts. However, the fax transmissions could be replaced with electronic data sent over the Internet.

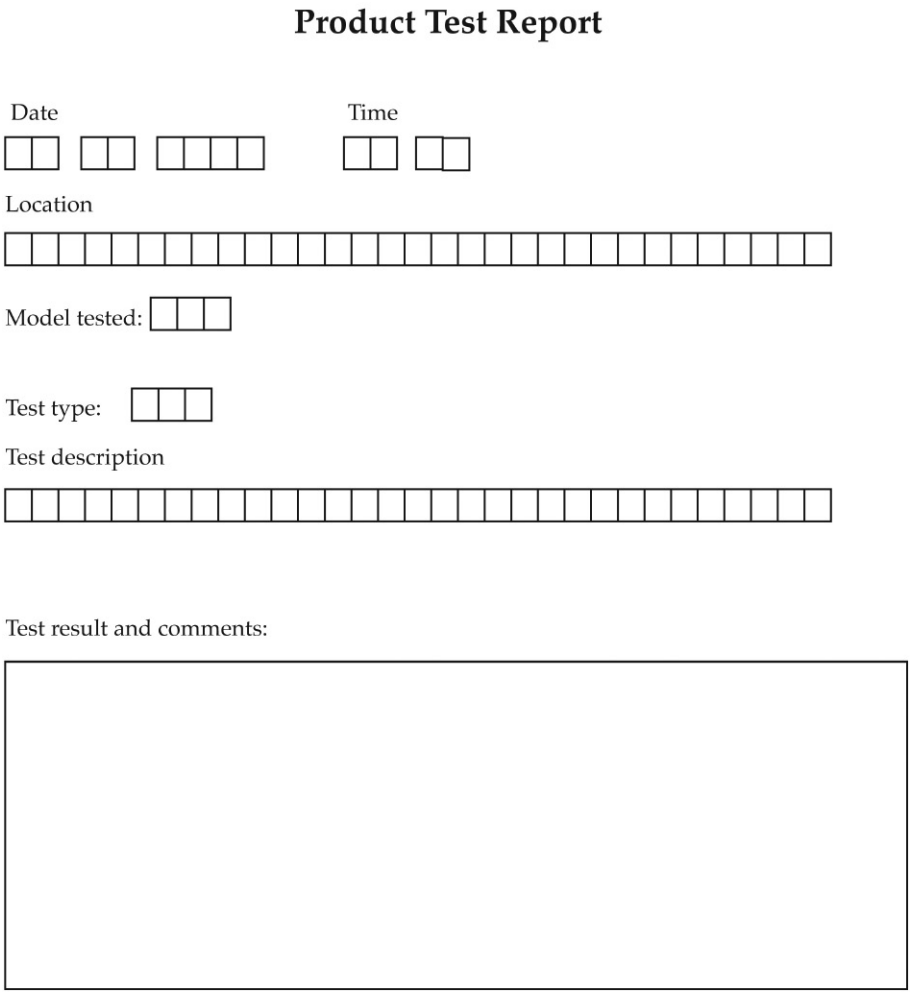
Product test results are recorded on paper forms (Figure 13.10). After a testing period is completed, the forms are collated manually to produce a summary of how well a new product performed. MMM would like the test results stored within an information system so that the testing report can be produced automatically, saving time and effort. Such a report will be used to help decide which new models should be placed in production.

Designing the Database

The most effective approach to the design of a database (or collection of databases) for an environment as diverse as that presented by Mighty-Mite Motors usually involves breaking the design into components indicated by the organization of the company. As the design evolves, the designer can examine the entities and the relationships to determine where parts of the organization will need to share data. Working on one portion of the



■ **FIGURE 13.9 Mighty-Mite Motors problem report.**



■ **FIGURE 13.10 Mighty-Mite Motors product test report.**

design at a time also simplifies dealing with what might at first seem to be an overwhelmingly large database environment. Paying special attention for the needs for shared data helps ensure that shared data are consistent and suitable for all required uses.

A systems analysis indicates that the MMM database environment falls into the following areas:

- Manufacturing (including finished goods inventory and raw materials ordering).

- Sales to toy stores, and shipping of products ordered.
- Purchase registrations.
- Testing.
- Problem handling.

Examining the Data Flows

During the systems analysis, a data flow diagram can be of enormous use in identifying where data are shared by various parts of an organization. The top-level DFD (the *context diagram* in Figure 13.11) actually tells us very little. It indicates that there are three sources outside the company provide data: customers (the stores to which the company sells), purchasers (the individuals who purchase products from the stores), and raw materials suppliers. Somewhere, all those data are used by a general process named "Manufacture and Sell Products" to keep the company in business.

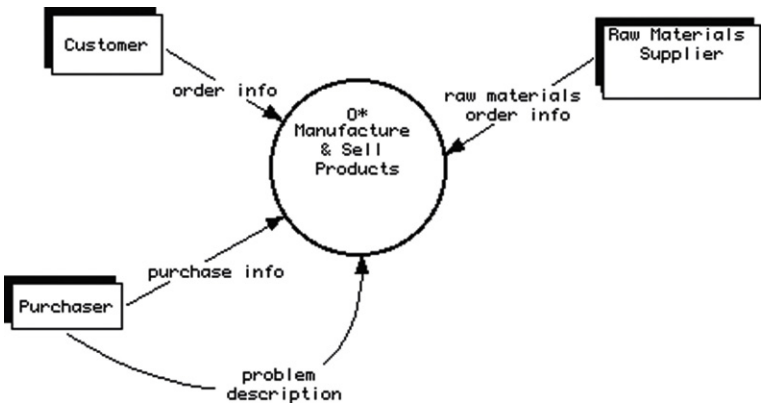
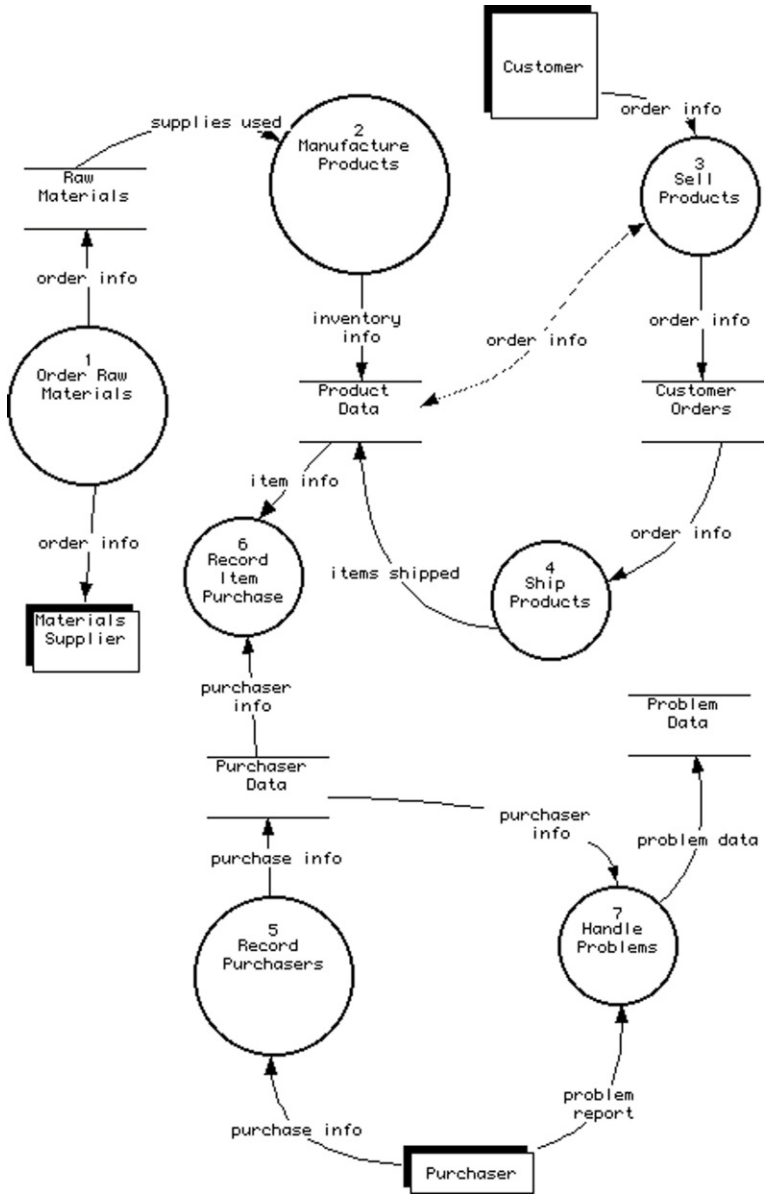


FIGURE 13.11 Context DFD for Mighty-Mite Motors.

However, the level 1 DFD (Figure 13.12) is much more telling. As the data handling processes are broken down, five data stores emerge:

- Raw materials: This data store holds both the raw materials inventory and the orders for raw materials.
- Product data: The product data store contains data about the products being manufactured, product testing results, and the finished goods inventory.
- Customer orders: This data store contains customer information, as well as order data.



■ **FIGURE 13.12 Level 1 DFD for Mighty-Mite Motors.**

- Purchaser data: The purchaser data store contains information about the individuals who purchase products and the products they have purchased.
- Problem data: This final data store contains problem reports.

As you examine the processes that interact with these five data stores, you will find a number of processes that manipulate data in more than one data store, as well as data stores that are used by more than one process:

- The raw materials data store is used by the raw materials ordering and the manufacturing processes.
- Product data are used by manufacturing, sales, shipping, and product registration.
- Customer order data are used by sales and shipping.
- The purchases data store is used by purchaser registration and problem handling.
- The problem data store, used only by problem handling, is the only data store not shared by multiple processes.

The raw materials ordering process is the only process that uses only a single data store. Nonetheless, the level 1 DFD makes it very clear that there is no instance in which a single process uses a single data store without interaction with other data stores and processes. Given that each process in the DFD probably represents all or part of an application program, this suggests that the database designer should consider either a single database or a set of small databases, along with software to facilitate the interchange of data.

The DFD makes it very clear that the need for the integration of the various data stores is very strong. In addition, Mighty-Mite Motors is a relatively small business and therefore a single database that manages all needed aspects of the company will not grow unreasonably large. It will also be more cost effective and perform better than multiple databases that use some type of middleware to exchange data. Ultimately, the database designer may decide to distribute the database onto multiple servers, placing portions of it that are used most frequently in the division where that use occurs. The database design, however, will be the same, regardless of whether the final implementation is centralized or distributed. The essential decision is to create a single database rather than several smaller, interrelated databases that must exchange data.

The ER Diagram

The systems analyst preparing the requirements document for the Mighty-Mite Motors reengineering project has had two very good sources of information

about exactly what needs to be stored in the database: the employees of the company and the paper documents that the company has been using. The document that is given to the database designer is therefore quite complete.

The design needs to capture all the information on the paper documents. Some documents are used only for input (for example, the product registration form or the order form). Others represent reports that an application program must be able to generate (for example, the line schedule report). Although the current documents do not necessarily represent all the outputs application programs running against the database will eventually prepare, they do provide a good starting place for the design. Whenever the designer has questions, he or she can then turn to Might-Mite's employees for clarification.

Working from the requirements document prepared by the systems analyst, along with the paper input and output documents, the database designer puts together the ER diagram. Because there are so many entities, all of which interconnect, the diagram is very wide. It has therefore been split into three pieces so you can see it. As you look at each piece, keep in mind that entities that appear on more than one piece represent the connection between the three illustrations.

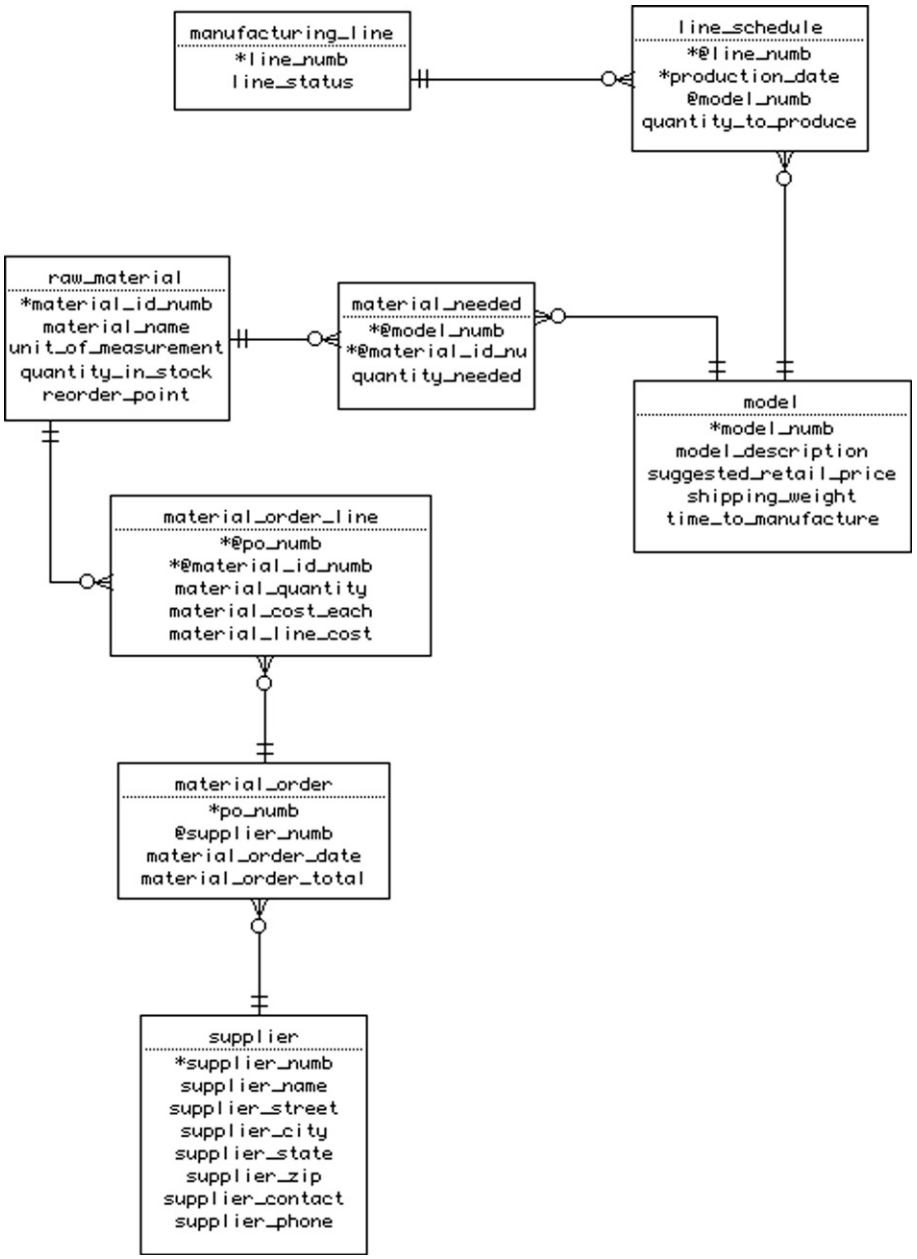
The first part (found in Figure 13.13) contains the entities for raw materials and manufacturing. This portion of the data model is dealing with three many-to-many relationships:

- *material_order* to *raw_material* (resolved by the composite entity *material_order_line*),
- *raw_material* to *model* (resolved by the composite entity *material_needed*),
- *manufacturing_line* to *model* (resolved by the composite entity *line_schedule*).

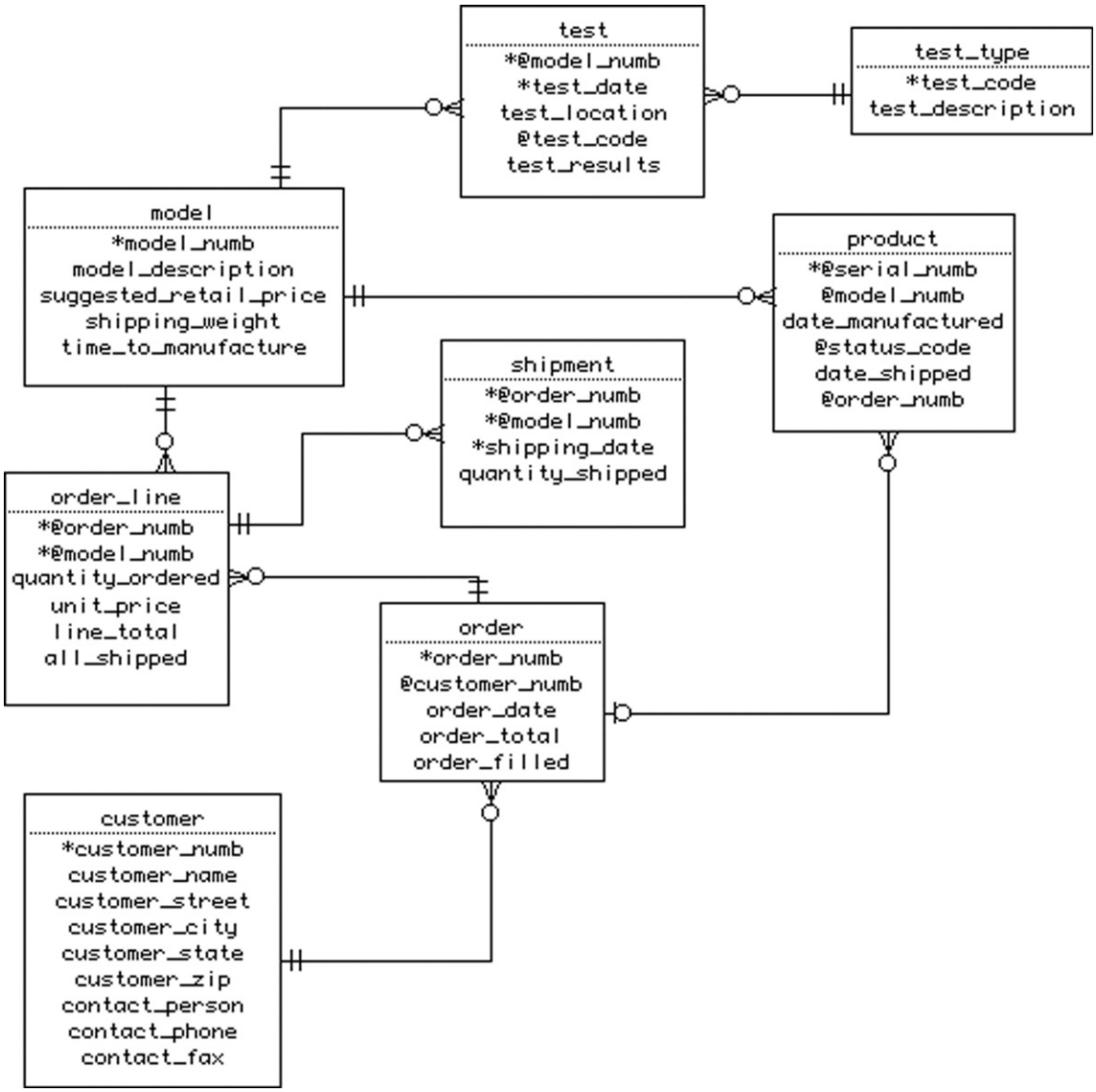
The second portion of the ERD (Figure 13.14) contains entities for product testing and sales. (Remember that in this instance, the customers are toy stores rather than individual purchasers.) There are two many-to-many relationships:

- *test_type* to *model* (resolved by the *test* entity),
- *order* to *model* (resolved by the *order_line* composite entity).

The test entity is somewhat unusual for a composite entity. It is an activity that someone performs and, as such, has an existence outside the database. It is not an entity created just to resolve a many-to-many relationship.



■ **FIGURE 13.13 Might-Mite Motors ERD (part 1).**



■ **FIGURE 13.14 Might-Mite Motors ERD (part II).**

At this point, the diagrams become a bit unusual because of the need to keep track of individual products rather than simply groups of products of the same model. The *model* entity, which you first saw in Figure 13.13, represents a type of vehicle manufactured by Mighty-Mite Motors. However, the *product* entity, which first appears in Figure 13.14, represents a

single vehicle that is uniquely identified by a serial number. This means that the relationships between an order, the line items on an order, and the models and products are more complex than for most other sales database designs.

The *order* and *line_item* entities are fairly typical. They indicate how many of a given model are required to fill a given order. The *shipment* entity then indicates how many of a specific model are shipped on a specific date. However, the database must also track the order in which individual products are shipped. As a result, there is a direct relationship between the *product* entity and the *order* entity in addition to the relationships between *order_line* and *model*. In this way, Mighty-Mite Motors will know exactly where each product has gone. At the same time, the company will be able to track the status of orders (in particular, how many units of each model have yet to ship).

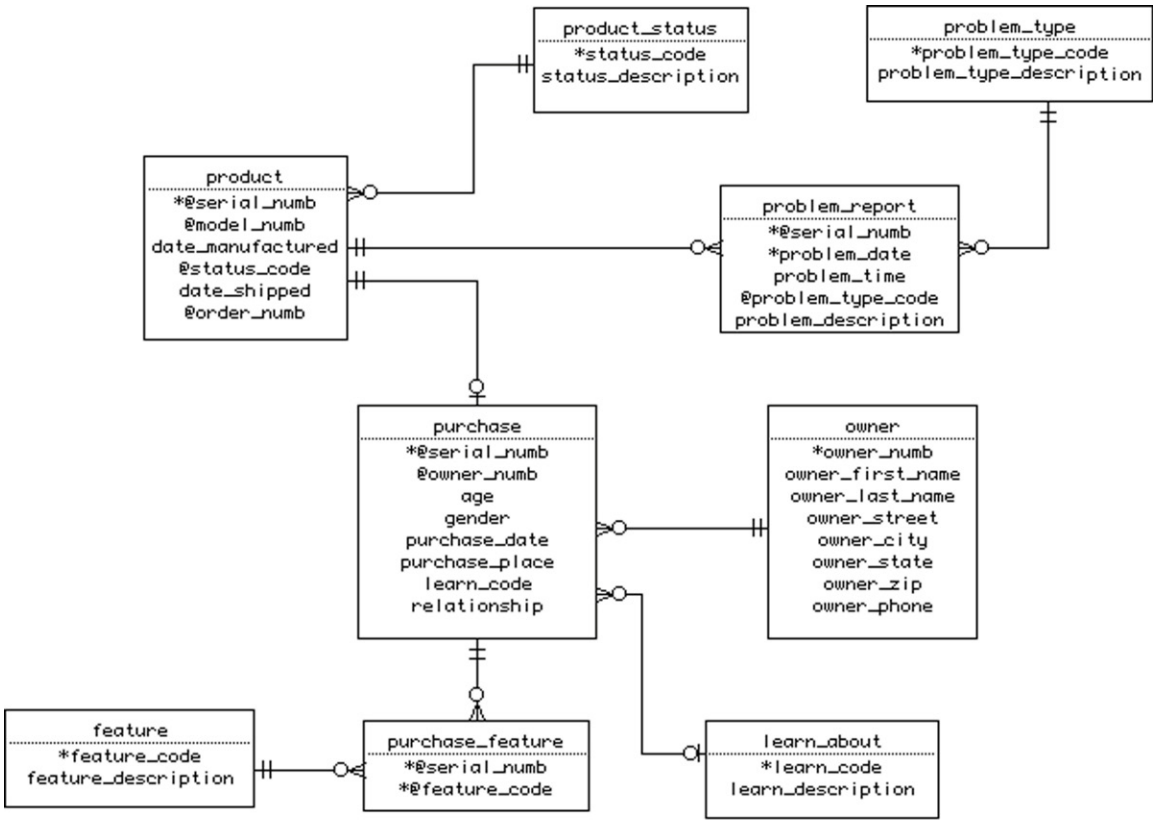
The final portion of the ERD (Figure 13.15) deals with the purchasers and problem reports. There are two many-to-many relationships:

- *problem_type* to *product* (resolved with the entity *problem_report*),
- *purchase* to *feature* (resolved with the composite entity *purchase_feature*).

As with the test entity that you saw earlier, the *problem_report* entity acts like a composite entity to resolve a many-to-many relationship, but is really a simple entity. It is an entity that has an existence outside the database and was not created simply to take care of the M:N relationship.

Note: Calling an entity "problem_report" can be a bit misleading. In this case, the word "report" does not refer to a piece of paper, but to the action of reporting a problem. A "problem_report" is therefore an activity rather than a document. In fact, the printed documentation of a problem report will probably include data from several entities, including the product, problem_report, purchase, and owner entities.

If you look closely at the diagram, you'll notice that there is a one-to-one relationship between the product and purchase entities. The handling of the data supplied by a purchaser on the product registration card presents an interesting dilemma for a database designer. Each product will be registered by only one purchaser. (Even if the product is later sold or given to someone else, the new owner will not have a registration card to send in.) There will be only one set of registration data for each product, at first thought suggesting that all the registration data should be part of the *product* entity.



■ **FIGURE 13.15 Mighty-Mite Motors ERD (part III).**

However, there is a lot of registration data—including one repeating group (the features for which the purchaser chose the product, represented by the *feature* and *purchase_feature* entities)—and the product is involved in a number of relationships that have nothing to do with product registration. If the DBMS has to retrieve the registration data along with the rest of the product data, database performance will suffer. It therefore makes sense in this case to keep the purchase data separate and to retrieve it only when absolutely needed.

Note: One common mistake made by novice database designers is to create an entity called "registration card." It is important to remember that the card itself is merely an input document. What is crucial is the data the card contains and the entity that the data describe, rather than the medium on which the data are supplied.

Creating the Tables

The tables for the Mighty-Mite Motors database can come directly from the ER diagram. They are as follows:

```
model (model_numb, model_description, suggested_retail_price,
     shipping_weight, time_to_manufacture)

test (model_numb, test_date, test_location, test_code,
     test_results)

test_types (test_code, test_description)

customers (customer_numb, customer_name, customer_street,
     customer_city, customer_state, customer_zip,
     contact_person, contact_phone, contact_fax)

orders (order_numb, customer_numb, order_date, order_total,
     order_filled)

order_line (order_numbzu, model_numb, quantity_ordered,
     unit_price, line_total, all_shipped)

shipments (order_numb, model_numb, quantity_shipped)

product (serial_numb, model_numb, date_manufactured,
     status_code, order_numb, date_shipped)

raw_material (material_id_numb, material_name,
     unit_of_measurement, quantity_in_stock, reorder_point)

supplier (supplier_numb, supplier_name, supplier_street,
     supplier_city, supplier_state, supplier_zip,
     supplier_contact, supplier_phone)

material_order (po_numb, supplier_numb, material_order_date,
     material_order_total)

material_order_line (po_numb, material_id_numb,
     material_quantity, material_cost_each, material_line_cost)

manufacturing_line (line_numb, line_status)

line_schedule (line_numb, production_date, model_numb,
     quantity_to_produce)

owner (owner_numb, owner_first_name, owner_last_name,
     owner_street, owner_city, owner_state, owner_zip,
     owner_phone)

purchase (serial_numb, owner_numb, age, gender, purchase_date,
     purchase_price, learn_code, relationship)

purchase_feature (serial_numb, feature_code)

learn_about (learn_code, learn_description)

feature (feature_code, feature_description)

problem_report (serial_numb, problem_date, problem_time,
     problem_type_code, problem_details)

problem_type (problem_type_code, problem_type_description)
```

Generating the SQL

Assuming that the designers of the Mighty-Mite Motors database are working with a CASE tool, then generating SQL statements to create the database can be automated. For example, in Figure 13.16 you will find the SQL generated by Mac A&D from the ER diagram you saw earlier in this chapter.

```
CREATE TABLE model
(
      model_numb INTEGER,
      model_description VARCHAR (40),
      suggested_retail_price NUMBER (6,2),
      shipping_weight NUMBER(6,2),
      time_to_manufacture TIME,
      PRIMARY KEY (model_numb)
);
CREATE TABLE test_type
(
      test_code INTEGER,
      test_description VARCHAR (40),
      PRIMARY KEY (test_code)
);
CREATE TABLE test
(
      test_date DATE,
      test_location VARCHAR (40),
      test_code INTEGER,
      test_results VARCHAR (256),
      PRIMARY KEY (model_numb, test_date),
      FOREIGN KEY (model_numb) REFERENCES model,
      FOREIGN KEY (test_code) REFERENCES test_type
);
CREATE TABLE customer
(
      customer_numb INTEGER,
      customer_name VARCHAR (40),
      customer_street VARCHAR (50),
      customer_city VARCHAR (50),
      customer_state CHAR (2),
      customer_zip CHAR (10),
      contact_person VARCHAR (30),
      contact_phone CHAR (12),
      contact_fax CHAR (12),
      PRIMARY KEY (customer_numb)
);
```

■ **FIGURE 13.16 SQL statements needed to create the Mighty-Mite Motors database (continues).**

```
CREATE TABLE order
(
      order_numb INTEGER,
      customer_numb INTEGER,
      order_date DATE,
      order_total NUMBER (8,2),
      order_filled BOOLEAN,
      PRIMARY KEY (order_numb),
      FOREIGN KEY (customer_numb) REFERENCES customer
);

CREATE TABLE order_line
(
      order_numb INTEGER,
      model_numb INTEGER,
      quantity_ordered INTEGER,
      unit_price NUMBER (6,2),
      line_total NUMBER (8,2),
      all_shipped BOOLEAN,
      PRIMARY KEY (order_numb, model_numb),
      FOREIGN KEY (order_numb) REFERENCES order,
      FOREIGN KEY (model_numb) REFERENCES model
);

CREATE TABLE shipment
(
      order_numb INTEGER,
      model_numb INTEGER,
      shipping_date DATE,
      quantity_shipped INTEGER,
      PRIMARY KEY (order_numb, model_numb, shipping_date),
      FOREIGN KEY (order_numb, model_numb) REFERENCES order_line
);

CREATE TABLE product
(
      serial_numb INTEGER,
      model_numb INTEGER,
      date_manufactured DATE,
      status_code INTEGER,
      date_shipped DATE,
      order_numb INTEGER,
      PRIMARY KEY (serial_numb),
      FOREIGN KEY (model_numb) REFERENCES model,
      FOREIGN KEY (status_code) REFERENCES product_status,
      FOREIGN KEY (order_numb) REFERENCES order
);

CREATE TABLE product_status
(
      status_code INTEGER,
      status_description VARCHAR (40),
      PRIMARY KEY (status_code)
);
```

■ **FIGURE 13.16 (*cont.*)**

```
CREATE TABLE raw_material
(
      material_id_numb INTEGER,
      material_name VARCHAR (40),
      unit_of_measurement CHAR (12),
      quantity_in_stock INTEGER,
      reorder_point INTEGER,
      PRIMARY KEY (material_id_numb)
);

CREATE TABLE material_needed
(
      model_numb INTEGER,
      material_id_numb INTEGER,
      quantity_needed INTEGER,
      PRIMARY KEY (model_numb, material_id_numb),
      FOREIGN KEY (model_numb) REFERENCES model,
      FOREIGN KEY (material_id_numb) REFERENCES raw_material
};

CREATE TABLE supplier
(
      supplier_numb INTEGER,
      supplier_name VARCHAR (40),
      supplier_street VARCHAR (50),
      supplier_city VARCHAR (50),
      supplier_state CHAR (2),
      supplier_zip CHAR (10),
      supplier_phone CHAR (12),
      PRIMARY KEY (supplier_numb)
};

CREATE TABLE material_order
(
      po_numb INTEGER,
      supplier_numb INTEGER,
      material_order_date DATE,
      material_order_total NUMBER (8,2),
      PRIMARY KEY (po_numb),
      FOREIGN KEY (supplier_numb) REFERENCES supplier
);

CREATE TABLE material_order_line
(
      po_numb INTEGER,
      material_id_numb INTEGER,
      material_quantity INTEGER,
      material_cost_each NUMBER (6,2),
      material_line_cost NUMBER (8,2),
      PRIMARY KEY (po_numb, material_id_numb),
      FOREIGN KEY (po_numb) REFERENCES material_order,
      FOREIGN KEY (material_id_numb) REFERENCES raw_material
}
```

■ **FIGURE 13.16** (***cont.***)

```
CREATE TABLE manufacturing_line
(
      line_numb INTEGER,
      line_status CHAR (12),
      PRIMARY KEY (line_numb)
);

CREATE TABLE line_schedule
(
      line_numb INTEGER,
      production_date DATE,
      model_numb INTEGER,
      quantity_to_product INTEGER
      PRIMARY KEY (line_numb, production_date),
      FOREIGN KEY (lne_numb) REFERENCES manufacturing_line,
      FOREIGN KEY (model_numb) REFERENCES model
);

CREATE TABLE owner
(
      owner_numb INTEGER,
      owner_street VARCHAR (50),
      owner_city VARCHAR (50),
      owner_state CHAR (2),
      owner_zip CHAR (10),
      owner_phone CHAR (10),
      PRIMARY KEY (owner_numb)
};

CREATE TABLE purchase
(
      serial_numb INTEGER,
      owner_numb INTEGER,
      age INTEGER,
      gender CHAR (1),
      purchase_date DATE,
      purchase_place VARCHAR (50),
      learn_code INTEGER,
      relationship CHAR (10),
      PRIMARY KEY (serial_numb),
      FOREIGN KEY (serial_numb) REFERENCES product,
      FOREIGN KEY (owner_numb) REFERENCES owner
      FOREIGN KEY (learn_code) REFERENCES learn_about
);

CREATE TABLE feature
(
      feature_code INTEGER,
      feature_description VARCHAR (40),
      PRIMARY KEY (feature_code)
);
```

■ **FIGURE 13.16** (***cont.***)

```
CREATE TABLE purchase_feature
(
      serial_numb INTEGER,
      feature_code INTEGER,
      PRIMARY KEY (serial_numb, feature_code),
      FOREIGN KEY (serial_numb) REFERENCES product,
      FOREIGN KEY (feature_code) REFERENCES feature
};

CREATE TABLE learn_about
(
      learn_code INTEGER,
      learn_description VARCHAR (50),
      PRIMARY KEY (learn_code)
};

CREATE TABLE problem_type
(
      problem_type_code INTEGER,
      problem_type_description VARCHAR (50),
      PRIMARY KEY (problem_type_code)
);

CREATE TABLE problem_report
(
      serial_numb INTEGER,
      problem_date DATE,
      problem_time TIME,
      problem_type_code INTEGER,
      problem_details VARCHAR (50),
      PRIMARY KEY (serial_numb, problem_date),
      FOREIGN KEY (serial_numb) REFERENCES product,
      FOREIGN KEY (product_type_code) REFERENCES problem_type
};
```

■ **FIGURE 13.16** (***cont.***)

Page left intentionally blank

Chapter 14 Database Design Case Study #2: East Coast Aquarium

Many-to-many relationships are often the bane of the relational database designer. Sometimes, it is not completely clear that you are dealing with that type of relationship. However, failure to recognize the many-to-many can result in serious data integrity problems.

The organization described in this chapter actually needs two databases that don't share data, the larger of which is replete with many-to-many relationships. In some cases, it will be necessary to create additional entities for composite entities to reference merely to ensure data integrity.

Perhaps the biggest challenge facing a database design working for East Coast Aquarium is the lack of complete specifications. As you will read, the people who will be using the application programs created to manipulate the aquarium's two new databases have only a general idea of what they need the programs to do. Unlike Mighty-Mite Motors—which had the history of working from a large collection of existing forms, documents, and procedures—East Coast Aquarium has nothing of that sort.

Organizational Overview

The East Coast Aquarium is a nonprofit organization dedicated to the study and preservation of marine life. Located on the Atlantic coast in the heart of a major northeastern US city, it provides a wide variety of educational services to the surrounding area. The aquarium is supported by donations, memberships, charges for private functions, gift shop revenues, class fees, and the small admission fees it charges to the public. Research activities are funded by federal and private grants. To help keep costs down, many of the public service jobs (leading tours, staffing the admissions counter, running the gift shop) are handled by volunteers.

The aquarium grounds consist of three buildings: the main facility, a dolphin house, and a marina where the aquarium's research barge is docked.

The centerpiece of the main building is a three-story center tank that is surrounded by a spiral walkway. The sides of the tank are transparent so that visitors can walk around the tank, observing the residents at various depths.

Note: If you happen to recognize the layout of this aquarium, please keep in mind that only the physical structure of the environment is modeled after anything that really exists. The way in which the organization functions is purely a product of my imagination, and no comment, either positive or negative, is intended with regard to the real-world aquarium.

The height of the tank makes it possible to simulate the way in which habitats change as the ocean depth changes. Species that dwell on the ocean floor, coral reef fish, and sand bar dwellers therefore are all housed in the same tank, interacting in much the same way as they would in the ocean.

The remaining space on the first floor of the main building (Figure 14.1) includes the gift shop and a quarantine area for newly arrived animals. The latter area is not accessible to visitors.

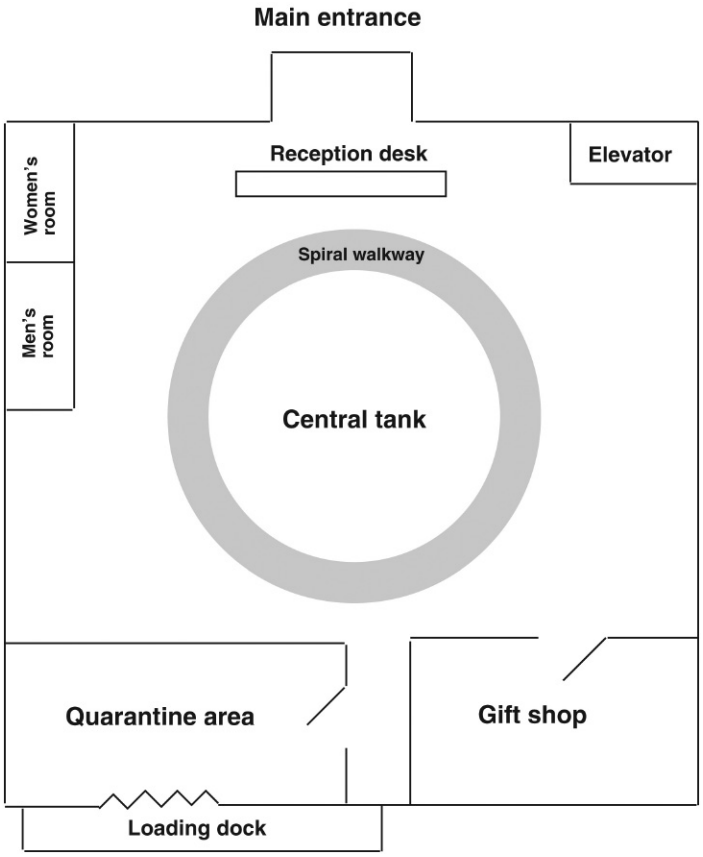
The second floor (Figure 14.2) contains a classroom and the volunteer's office. Small tanks containing single-habitat exhibits are installed in the outside walls. These provide places to house species that have special habitat requirements or that don't coexist well with other species.

The third floor (Figure 14.3) provides wall space for additional small exhibits. It also houses the aquarium's administrative offices.

East Coast Aquarium has two very different areas in which it needs data management. The first is in the handling of its animals—where they are housed in the aquarium, the source and location from where they came, what they are to be fed, problems that occur in the tanks, and so on. The second area concerns the volunteers, including who they are, what they have been trained to do, and when they are scheduled to work. For this particular organization, the two data environments are completely separate: They share no data. A database designer who volunteers to work with the aquarium staff will therefore prepare two database designs, one to be used by the volunteer staff in the volunteer's office and another to be used by the administrative and animal-care staff through the aquarium grounds.

Animal Tracking Needs

Currently, East Coast Aquarium uses a general-purpose PC accounting package to handle its data processing needs. The software takes care of

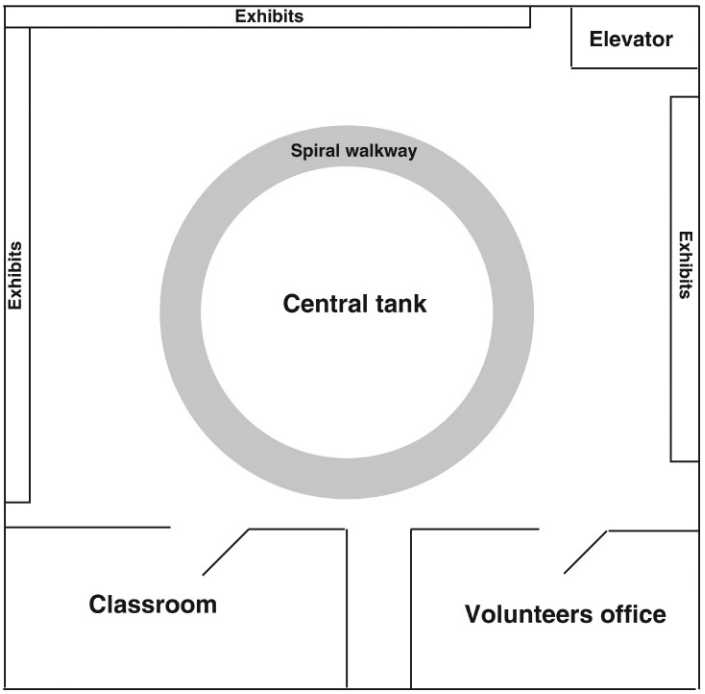


■ **FIGURE 14.1 The first floor of East Coast Aquarium's main building.**

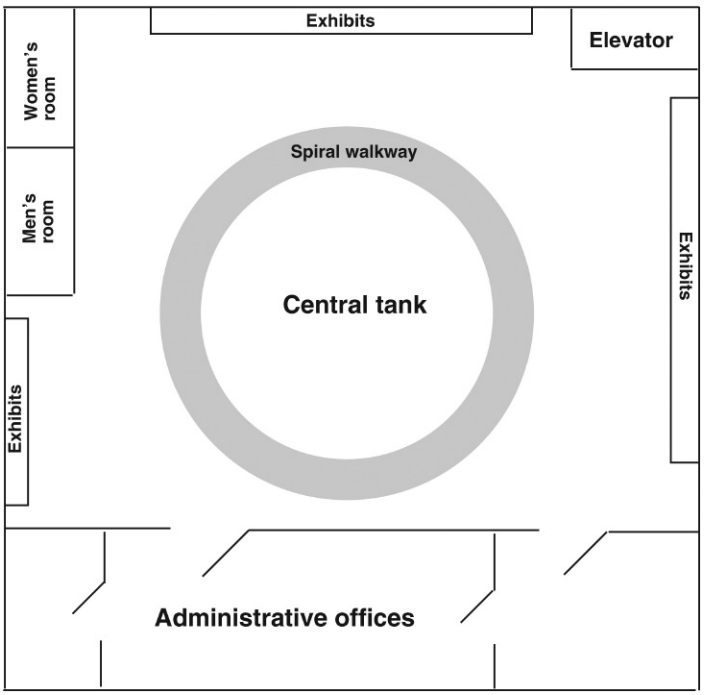
payroll as well as purchasing, accounts payable, and accounts receivable. Grant funds are managed by special-purpose software designed to monitor grant awards and how they are spent.

Although the accounting and grant management packages adequately handle the aquarium's finances, there is no data processing that tracks the actual animals housed in the aquarium. The three people in charge of the animals have expressed a need for the following:

- An "inventory" of which species are living in which locations in the aquarium. Some species can be found in more than one tank and several tanks in addition to the central tank contain more than one species. For larger animals, such as sharks and dolphins, the head animal keeper would like a precise count. However, for small fish that



■ FIGURE 14.2 The second floor of East Coast Aquarium's main building.



■ FIGURE 14.3 The third floor of East Coast Aquarium's main building.

are often eaten by large fish and that breed in large numbers, only an estimate is possible. The animal handling staff would like to be able to search for information about animals using either the animal's English name or its Latin name.

- Data about the foods each species eats, including how much should be fed at what interval. The head animal keeper would like to be able to print out a feeding instruction list every morning to give to staff. In addition, the animal-feeding staff would like to store information about their food inventory. Although purchasing of food is handled by the administrative office, the head animal keeper would like an application program to decrement the food inventory automatically by the amount fed each day and to generate a tickle request whenever the stock level of a type of food drops below the reorder point. This will make it much easier to ensure that the aquarium does not run short of animal food.
- Data about the sizes, locations, and habitats of the tanks on the aquarium grounds. Some tanks, such as the main tank, contain more than one habitat and the same habitat can be found in more than one tank.
- Data about tank maintenance. Although the main tank is fed directly from the ocean, the smaller tanks around the walls of the main building are closed environments, much like a saltwater aquarium might be at home. This means that the pH and salinity of the tanks must be monitored closely. The head animal keeper therefore would like to print out a maintenance schedule each day as well as be able to keep track of what maintenance is actually performed.
- Data about the habitats in which a given species can live. When a new species arrives at the aquarium, the staff can use this information to determine which locations could possibly house that species.
- Data about where species can be obtained. If the aquarium wants to increase the population of a particular species and the increase cannot be generated through in-house breeding, then the staff would like to know which external supplier can be contacted. Some of the suppliers sell animals; others, such as zoos or other aquariums, will trade or donate animals.
- Problems that arise in the tanks. When animals become ill, the veterinarian wants to be able to view a history of both the animal and the tank in which it is currently living.
- Data about orders placed for animals, in particular, the shipments in which animals arrive. Since any financial arrangements involved in securing animals are handled by the administrative office, these data indicate only how many individuals of each species are included on a given order or shipment.

The shipment and problem data are particularly important to the aquarium. When animals first arrive, they are not placed immediately into the general population. Instead, they are held in special tanks in the quarantine area at the rear of the aquarium's first floor. The length of the quarantine is determined by the species.

After the quarantine period has passed and the animals are declared disease free, they can be placed on exhibit in the main portion of the aquarium. Nonetheless, animals do become ill after they have been released from quarantine. It is therefore essential that records are kept of the sources of animals so that patterns of illness can be tracked back to specific suppliers. By the same token, patterns of illnesses in various species housed in the same tank can be an indication of serious problems with the environment in the tank.

The Volunteer Organization

The volunteer organization (the Friends of the Aquarium) is totally separate from the financial and animal-handling areas of the aquarium. Volunteers perform tasks that do not involve direct contact with animals, such as leading tours, manning the admissions desk, and running the gift shop. The aquarium has provided office space and a telephone line for the volunteer coordinator and her staff. Beyond that, the Friends of the Aquarium organization has been on its own to secure office furniture and equipment.

The recent donation of a PC now makes it possible for the volunteers to input some of the volunteer data online, although the scheduling is still largely manual. Currently, the scheduling processing works in the following way:

- The person on duty in the volunteer's office receives requests for volunteer services from the aquarium's administrative office. Some of the jobs are regularly scheduled (for example, staffing the gift shop and the admissions desk). Others are ad hoc, such as the request by a schoolteacher to bring a class of children for a tour.
- The volunteer doing the scheduling checks the list of volunteers to see who is trained to do the job requested. Each volunteer's information is recorded in a data file that contains the volunteer's contact data, along with the volunteer's skills. A skill is a general expression of something the volunteer knows how to do, such as lead a tour for elementary school children. The volunteer's information also includes an indication of when that person is available to work.
- The volunteer doing the scheduling searches the file for those people who have the required skill and have indicated that they are available at the required time. Most volunteer's work on a regularly scheduled

basis, either at the admissions desk or in the gift shop. However, for ad hoc jobs, the person doing the scheduling must start making telephone calls until someone who is willing and able to do the job is found.

- The volunteer is scheduled for the job by writing in the master scheduling notebook. As far as the volunteer coordinator is concerned, a job is an application of a skill. Therefore, a skill is knowing how to lead a tour for elementary school students, while a job that applies that skill is leading a tour for Mrs Brown's third graders at 10 am on Thursday.

One of the things that is very difficult to do with the current scheduling process is to keep track of the work record of each individual volunteer. The aquarium holds a volunteer recognition luncheon once a year and the volunteer organization would like to find an easy way to identify volunteers who have put in an extra effort so that they can be recognized at that event. In contrast, the volunteer organization would also like to be able to identify volunteers who rarely participate—the people who stay on the volunteer rolls only to get free admission to the aquarium—as well as people who make commitments to work but do not show up. (The latter are actually far more of a problem than the former.)

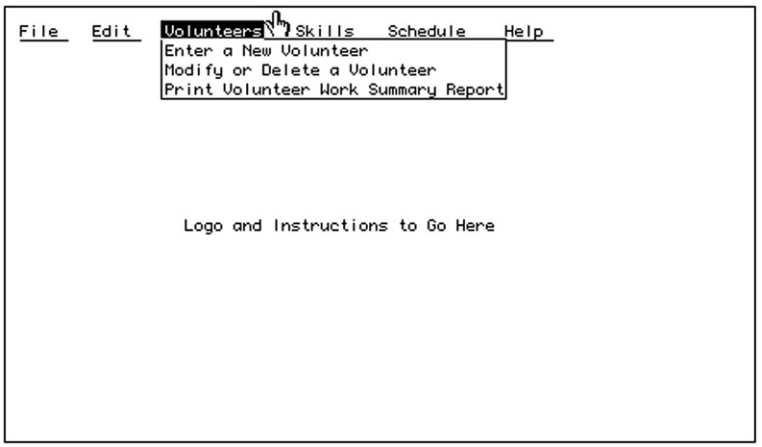
The Volunteers Database

In terms of scope, the volunteers database is considerably smaller than the animal tracking database. It therefore makes sense to tackle that smaller project first. The database designer will create an application prototype and review it with the users. When the users are satisfied and the designers feel they have detailed information to actually design a database, they will move on to the more traditional steps of creating an ER diagram, tables, and SQL statements.

Note: As you will see, there is a lot involved in creating a prototype. It requires very detailed, intensive work, and produces a number of diagrams and/or application program shells. We will therefore look at the volunteers prototype in full, but, in the interest of length, we will look at only selected aspects of the animal tracking prototype.

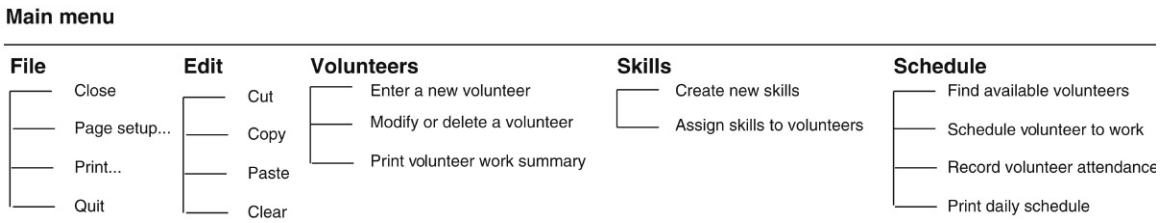
Creating the Application Prototype

Given that the specifications of the database are rather general, the first step is to create a prototype of an application program interface. It begins with the opening screen and its main menu bar (Figure 14.4). As you can see, when in browse mode, the CASE tool allows users and designers to pull down the menus in the menu bar.



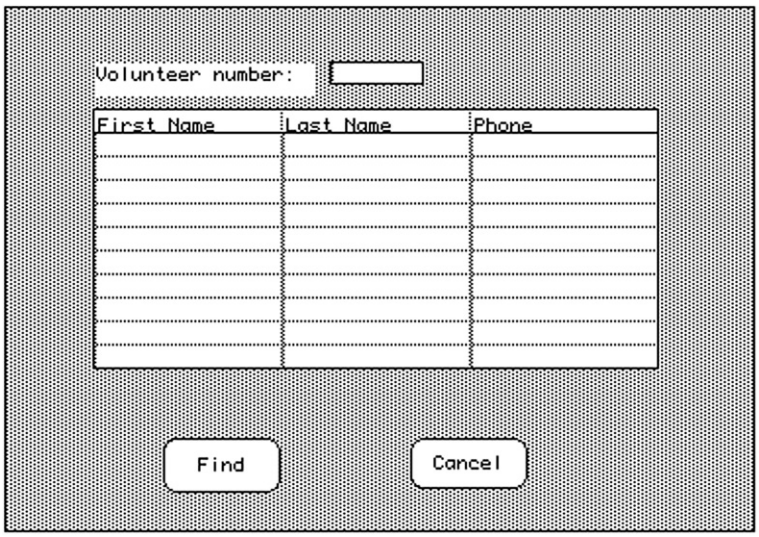
■ **FIGURE 14.4 Main menu prototype for the volunteers application.**

The complete menu tree (with the exception of the Help menu, whose contents are determined by the user interface guidelines of the operating system on which the application is running) can be found in Figure 14.5. Looking at the menu options, users can see that their basic requirements have been fulfilled. The details, however, must be specified by providing users with specific input and output designs.

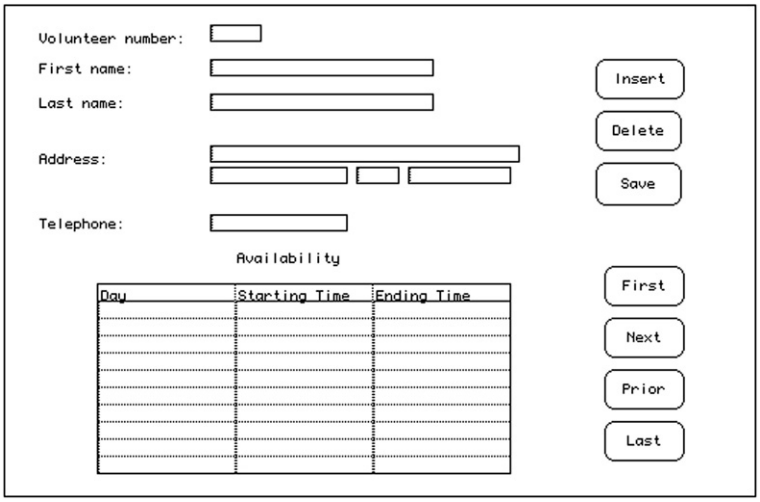


■ **FIGURE 14.5 Menu tree of the volunteers database prototype application.**

Each menu option in the prototype's main menu has therefore been linked to a screen form. For example, to modify or delete a volunteer, a user must first *find* the volunteer's data. Therefore, the Modify or Delete a Volunteer menu option leads to a dialog box that lets the user either enter a volunteer number or select a volunteer by name and phone number from a list (Figure 14.6). With the prototype, clicking the Find button opens the modify/delete form (Figure 14.7). Users can click in the data entry fields and tab between them, but the buttons at the right of the window are not functional.



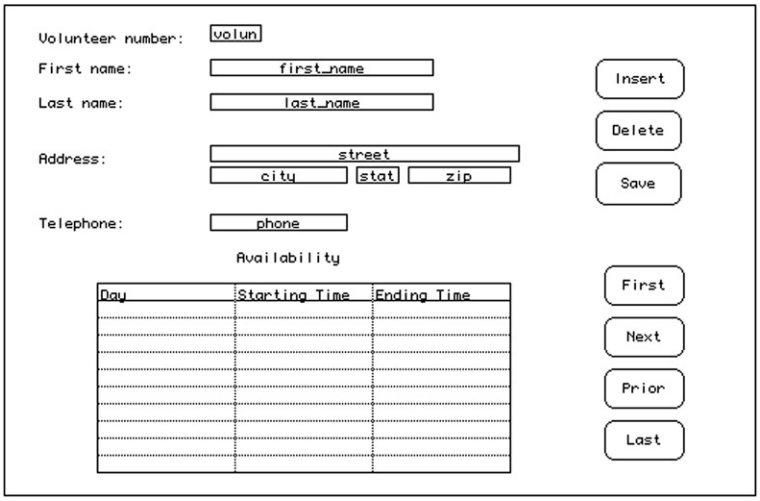
■ **FIGURE 14.6 Prototype of a dialog box for finding a volunteer for modifications.**



■ **FIGURE 14.7 Prototype of form for modifying and deleting a volunteer.**

While in browse mode, the CASE tool presents a form as it would appear to the user. However, in design mode, a database designer can see the names of the fields on the form (for example, Figure 14.8). These field names suggest attributes that will be needed in the database.

In the case of the volunteer data, it is apparent to the designers that there are at least two entities (and perhaps three) involved with the data that describe a



■ **FIGURE 14.8 Prototype data modification form showing field names.**

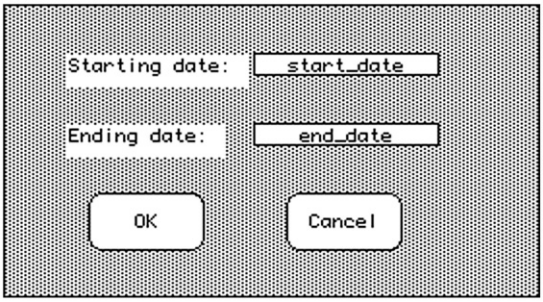
volunteer. The first entity is represented by the single-valued fields occupying the top half of the form (volunteer number, first name, last name, city, state, zip, and phone). However, the availability data—day of the week, starting time, and ending time—are multi-valued and therefore must be given an entity of their own. This also implies that there will be a one-to-many relationship between a volunteer and a period of time during which he or she is available.

Note: Should you choose, the field names of a screen prototype can become part of the data dictionary. However, if the field names do not ultimately correspond to column names, their inclusion may add unnecessary complexity to the data dictionary.

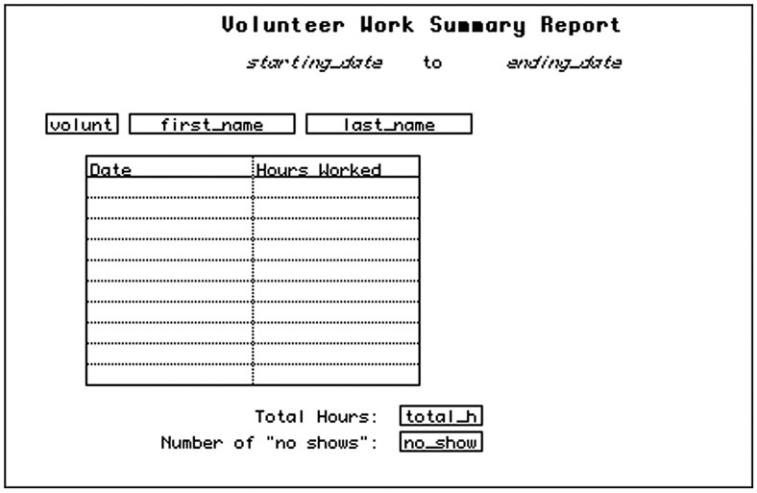
The remainder of the prototype application and its forms are designed and analyzed in a similar way:

- The volunteer work summary report has been designed to let the user enter a range of dates that the report will cover (see Figure 14.9). The report itself (Figure 14.10) is a control-break report that displays the work performed by each volunteer, along with the total hours worked and the number of times the volunteer was a "no show." The latter number was included because the volunteer coordinator had indicated that it was extremely important to know which volunteers consistently signed up to work and then didn't report when scheduled.

The need to report the "no shows" tells the designers that the schedule table needs to include a Boolean column that indicates whether a person



■ **FIGURE 14.9 A dialog box layout for entering dates for the work summary report.**



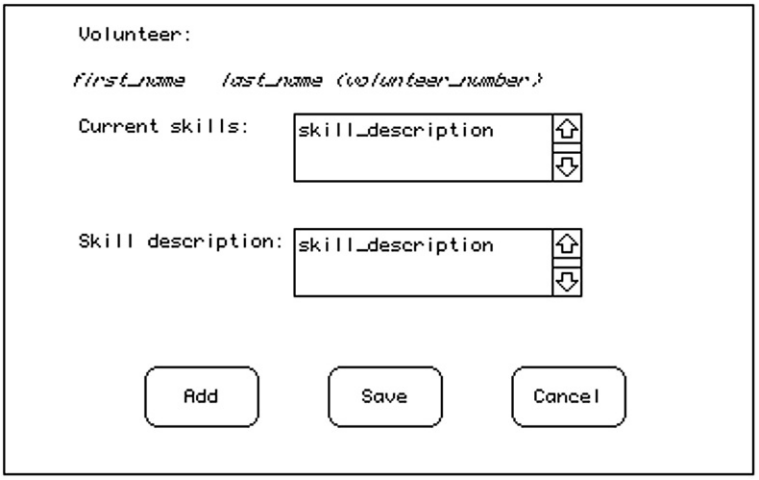
■ **FIGURE 14.10 Prototype layout for the work summary report.**

showed up for a scheduled shift. The report layout also includes some computed fields (total hours worked and number of no shows) that contain data that do not need to be stored, but can be generated when the report is displayed.

- Entering a new skill into the master list requires only a simple form (Figure 14.11). The end user sees only the description of a skill. However, the database designers know that the best way to handle unstructured blocks of text is to assign each description a skill number, which can then be used as a foreign key throughout the database. Users, however, do not necessarily need to know that a skill number is being used; they will always see just the text descriptions.

■ **FIGURE 14.11 Entering a new skill.**

- To assign skills to a volunteer, the end user must first find the volunteer. The application can therefore use a copy of the dialog box in Figure 14.6. In this case, however, the Find button leads to the form in Figure 14.12.



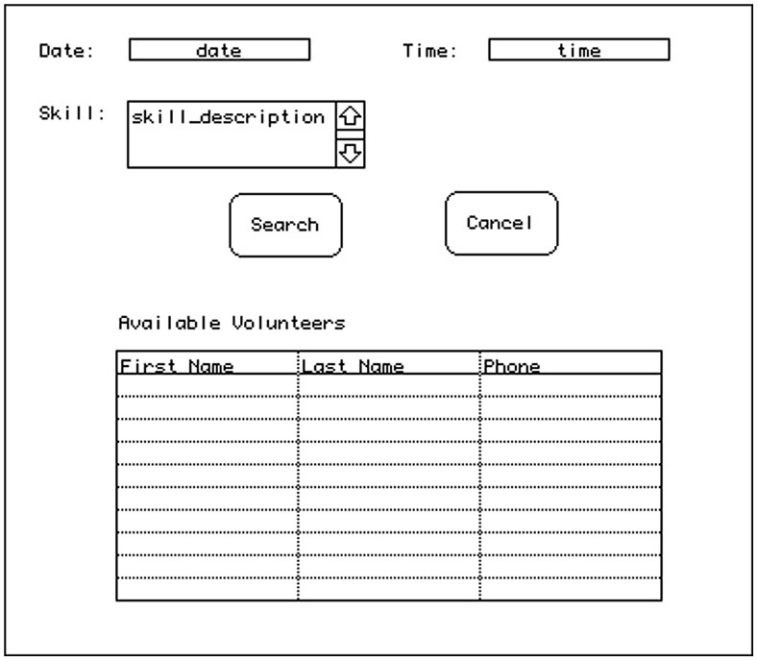
■ **FIGURE 14.12 Assigning skills to a volunteer.**

A database designer will recognize quickly that there is a many-to-many relationship between a skill and a volunteer. There are actually three entities behind Figure 14.12: the skill, the volunteer, and the composite entity that represents the relationship between the two. The skill entry form displays data from the volunteer entity at the top, data from the composite entity in the current skills list, and all skills not assigned from the skills table in the skill description list. Of course, the actually foreign key used in the composite entity is a skill number, but the user sees only the result of a join back to the skills table that retrieves the skill description.

Note: Database integrity constraints will certainly prevent anyone from assigning the same skill twice to the same volunteer. However, it is easier if

the user can see currently assigned skills. Then, the application can restrict what appears in the skill description list to all skills not assigned to that volunteer. In this case, it is a matter of user interface design, rather than database design.

- To find the volunteers available to perform a specific job, the volunteer's application needs a form something like Figure 14.13. The end user enters the date and time of the job and chooses the skill required by the job. Clicking the Search button fills in the table at the bottom of the form with the names and phone numbers of volunteers who are theoretically available.



■ **FIGURE 14.13 Finding available volunteers.**

Of all the outputs produced by this application, finding available volunteers is probably the most difficult to implement. The application program must not only work with overlapping intervals of time, but also consider both when a volunteer indicates he or she will be available and when a volunteer is already scheduled to work. In most cases, however, a database designer does not have to write the application program code. The designer needs only to ensure that the data necessary to produce the output are present in the database.

Note: A smart database designer, however, would discuss any output that involves something as difficult as evaluating overlapping time intervals with application programmers to ensure that the output is feasible, not only in terms of data manipulation, but in terms of performance as well. There is no point in specifying infeasible output in a design.

- Once the person doing the volunteer scheduling has located a volunteer to fill a specific job, then the volunteer's commitment to work needs to become a part of the database. The process begins by presenting the user with a Find Volunteer dialog box like that in Figure 14.6. In this case, the Find button is linked to the Schedule Volunteer window (Figure 14.14). A database designer will recognize that this is not all the data the needs to be stored about a job, however. In particular, someone will need to record whether the volunteer actually appeared to do the scheduled job on the day of the job; this cannot be done when the job is scheduled initially.

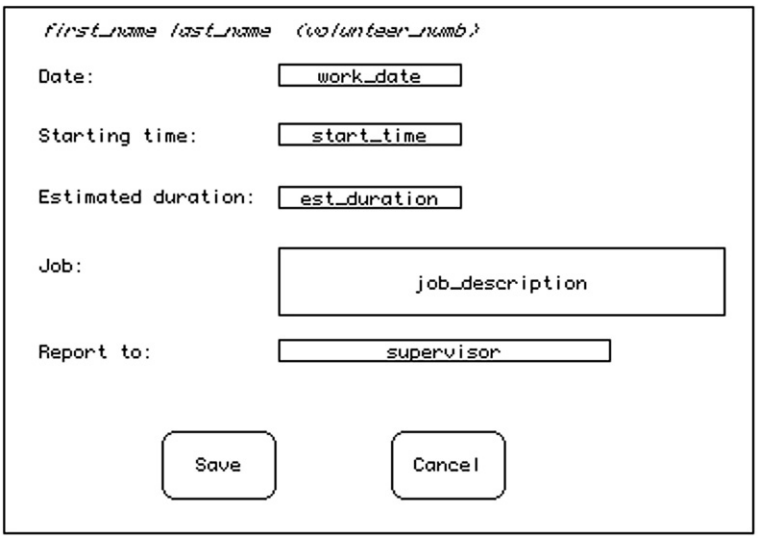
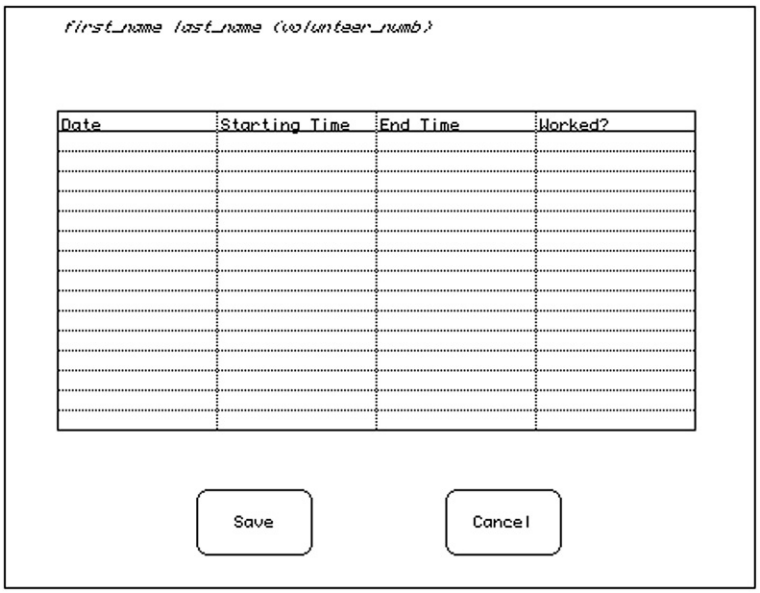


FIGURE 14.14 Scheduling a volunteer to perform a job.

- To record attendance, an end user first locates the volunteer using a Find Volunteer dialog box (Figure 14.6), which then leads to a display of the jobs the volunteer has been scheduled to work in reverse chronological order (see Figure 14.15). For those jobs that have not been worked, the End Time and Worked? columns will be empty. The user can then scroll the list to find the job to be modified and enter



■ **FIGURE 14.15 Recording jobs worked.**

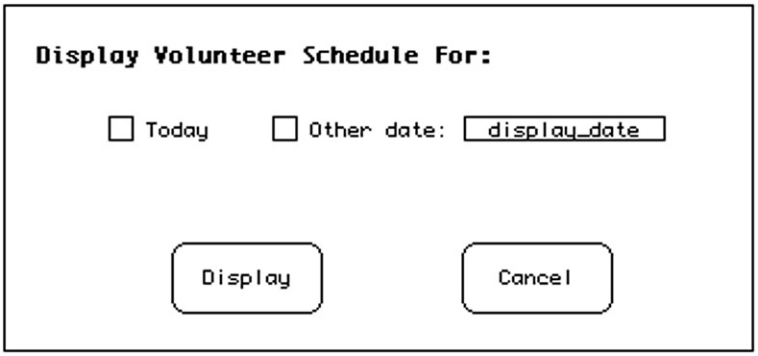
values for the two empty columns. The fields on this form, plus those on the job scheduling form, represent the attributes that will describe the job entity.

- To print a daily schedule, an end user first uses a dialog box to indicate the date for which a schedule should be displayed (Figure 14.16). The application program then assembles the report (Figure 14.17). To simplify working with the program, the application developers should probably allow users to double-click on any line in the listing to open the form in Figure 14.15 for the scheduled volunteer. However, this capability will have no impact on the database design.

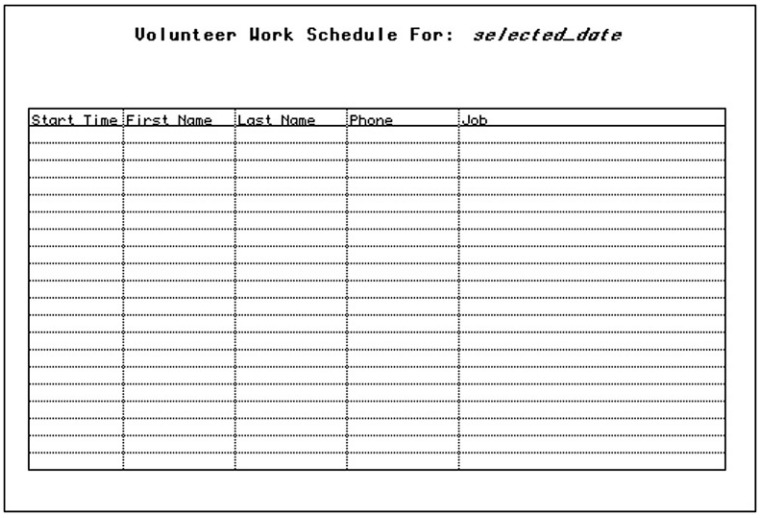
Creating the ER Diagram

From the approved prototype of the application design and conversations with the volunteers who do volunteer scheduling, the database designers can gather enough information to create a basic ER diagram for the volunteers organization. The designers examine each screen form carefully to ensure that the database design provides the attributes and relationships necessary to generate the output.

At first, the ER diagram (Figure 14.18) may seem overly complex. However, two of the entities—*state* and *day*—are present for referential integrity



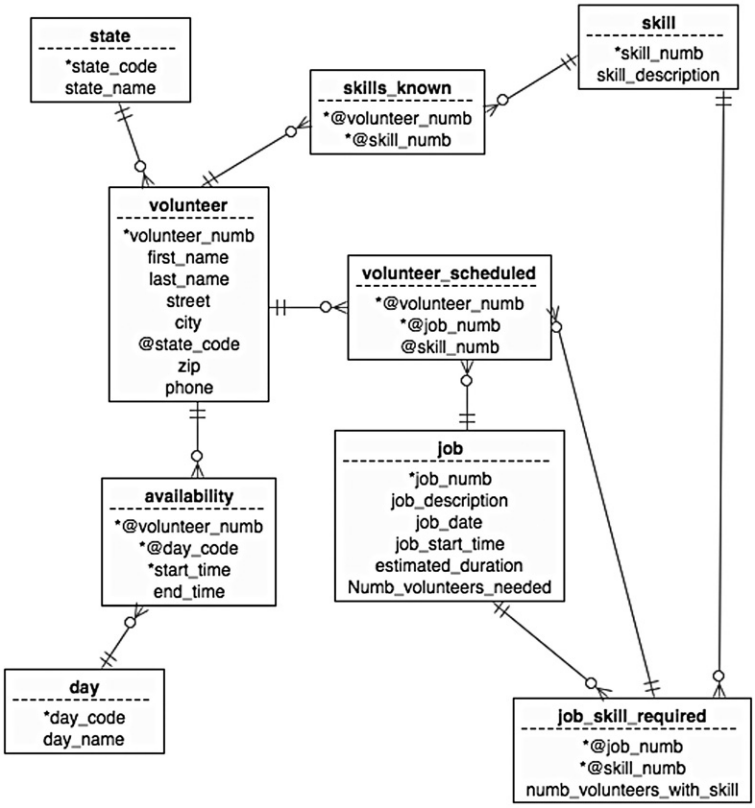
■ **FIGURE 14.16 Choosing a date for schedule display.**



■ **FIGURE 14.17 Volunteer work schedule.**

purposes, ensuring that abbreviations for state and day names are entered consistently. The relationships between a volunteer, jobs, and skills also aren't quite as simple as they might seem at first, in part because there are several many-to-many relationships:

- There is a many-to-many relationship between volunteers and skills, which is handled by the *skills_known* entity.
- Because a job may require more than on volunteer, there is a many-to-many relationship between volunteers and jobs that is handled by the *volunteer_scheduled* entity.



■ **FIGURE 14.18 ER diagram for the volunteers database.**

- A job may require more than one skill and a skill is used on many jobs. The many-to-many relationship between a job and a skill is therefore handled by the *job_skill_required* entity.

Making the situation a bit more complex is that the meaning of the M:M relationship between *job* and *skill* (through *job_skill_required*) is used differently than the relationship between *volunteer_scheduled* and *job_skill_required*. A row is added to *job_skill_required* for each skill required by a job; these data are available when the job is requested. As volunteers are scheduled for the job, rows are added to *volunteer_scheduled*, creating the relationship between that entity and *job_skill_required*. (This is essential for determining the correct volunteers still needed to be scheduled for specific skills for a specific job.) The foreign key in *volunteer_scheduled* uses one column from the table's primary key (*job_numb*) and one non-key attribute (*skill_numb*). Nonetheless, that concatenation is exactly the same as the primary key of the *job_skill_required* table (same columns with the same meaning).

Designing the Tables

The ER diagram in Figure 14.18 produces the following tables:

```
volunteer (volunteer_numb, first_name, last_name, street,
     city, state_code, zip, phone)

state (state_code, state_name)

availability (volunteer_numb, day_code, start_time, end_time)

day (day_code, day_name)

skill (skill_numb, skill_description)

skills_known (volunteer_numb, skill_numb)

job (job_numb, job_description, job_date, job_start_time,
     estimated_duration, numb_volunteers_needed)

job_skill_required (job_numb, skill_numb,
     numb_volunteers_with_skill)

volunteer_scheduled (volunteer_numb, job_numb, skill_numb)
```

Generating the SQL

The nine tables that make up the volunteers database can be created with the SQL in Figure 14.19. Notice that some of the attributes in the volunteers table have been specified as NOT NULL. This constraint ensures that at least a name and phone number are available for each volunteer.

```
CREATE TABLE skill
(
  skill_numb integer,
  skill_description char (50),
  CONSTRAINT PK_skill PRIMARY KEY (skill_numb)
);

CREATE TABLE job
(
  job_numb integer,
  job_description varchar (256),
  job_date date,
  job_start_time time,
  estimated_duration interval,
  numb_volunteers_needed integer,
  CONSTRAINT PK_job PRIMARY KEY (job_numb)
);
```

■ **FIGURE 14.19 SQL statements needed to create the tables for the volunteers database (continues).**

```
CREATE TABLE job_skill_required
(
  job_numb integer,
  skill_numb integer,
  numb_volunteers_with_skill integer,
  CONSTRAINT PK_job_skill_required PRIMARY KEY (job_numb,skill_numb),
  CONSTRAINT Relationjobjob_skill_required1 FOREIGN KEY ()
     REFERENCES job,
  CONSTRAINT Relationskilljob_skill_required1 FOREIGN KEY ()
     REFERENCES skill
);

CREATE TABLE state
(
  state_code char (2),
  state_name char (15),
  CONSTRAINT PK_state PRIMARY KEY (state_code)
);

CREATE TABLE volunteer
(
  volunteer_numb integer,
  first_name char (15) NOT NULL,
  last_name char (15) NOT NULL,
  street char (50),
  city char (30),
  state_code char (2),
  zip char (10),
  phone char (10) NOT NULL,
  CONSTRAINT PK_volunteer PRIMARY KEY (volunteer_numb),
  CONSTRAINT Relationstatevolunteer1 FOREIGN KEY (state_code)
      REFERENCES state
);

CREATE TABLE volunteer_scheduled
(
  volunteer_numb integer,
  job_numb integer,
  skill_numb integer,
  CONSTRAINT PK_volunteer_scheduled PRIMARY KEY
      (volunteer_numb,job_numb),
  CONSTRAINT Relationvolunteervolunteer_scheduled1 FOREIGN KEY ()
      REFERENCES volunteer,
  CONSTRAINT Relationjobvolunteer_scheduled1 FOREIGN KEY ()
     REFERENCES job,
  CONSTRAINT Relationjob_skill_requiredvolunteer_scheduled1
     FOREIGN KEY () REFERENCES job_skill_required
);

CREATE TABLE skills_known
(
  volunteer_numb integer,
  skill_numb integer,
  CONSTRAINT PK_skills_known PRIMARY KEY (volunteer_numb,skill_numb),
  CONSTRAINT Relationvolunteerskills_known1 FOREIGN KEY
     (VOLUNTEER_NUMB) REFERENCES volunteer,
  CONSTRAINT Relationskillskills_known1 FOREIGN KEY (skill_numb)
     REFERENCES skill
);
```

FIGURE 14.19 (*cont.*)

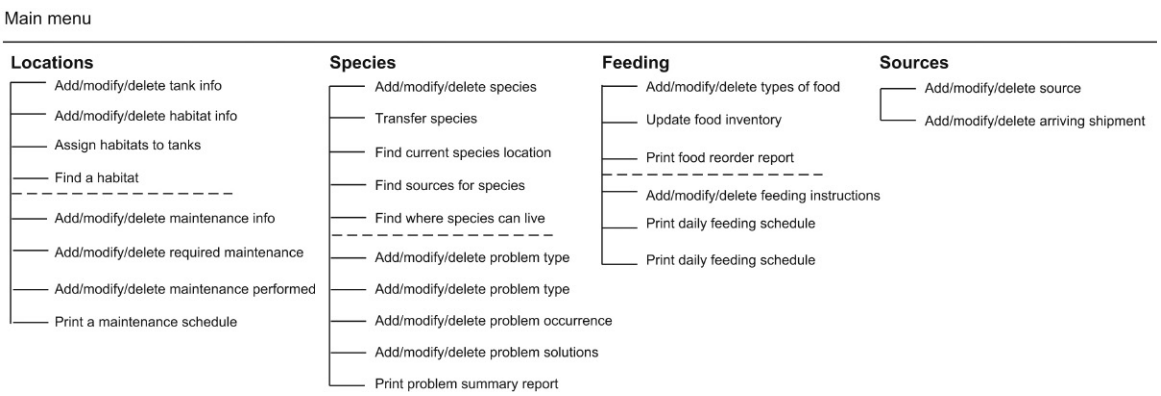
```
CREATE TABLE day
(
  day_code char (3),
  day_name char (10),
  CONSTRAINT PK_day PRIMARY KEY (day_code)
);

CREATE TABLE availability
(
  volunteer_numb integer,
  day_code char (3),
  start_time time,
  end_time time,
  CONSTRAINT PK_availability PRIMARY KEY
     (volunteer_numb,day_code,start_time),
  CONSTRAINT Relationvolunteeravailability1 FOREIGN KEY
     (volunteer_numb) REFERENCES volunteer,
  CONSTRAINT Relationdayavailability1 FOREIGN KEY (day_code)
     REFERENCES day
);
```

■ **FIGURE 14.19** (***cont.***)

The Animal Tracking Database

The animal tracking database is considerably bigger than the volunteers database. The application that will manipulate that database therefore is concomitantly larger, as demonstrated by the menu tree in Figure 14.20. (The File and Edit menus have been left off so the diagram will fit across the width of the page. However, they are intended to be the first and second menus from the left, respectively. A Help menu can also be added along the right side.)

The functionality requested by the animal handlers falls generally into four categories: the locations (the tanks) and their habitats, the species, the food,



■ **FIGURE 14.20 Menu tree for the animal tracking application.**

and the sources of animals. The organization of the application interface, therefore, was guided by those groups.

Highlights of the Application Prototype

The screen and report layouts designed for the animal tracking application provide a good starting place for the database designers to identify the entities and attributes needed in the database. As with the volunteers application, there is not necessarily a one-to-one correspondence between an entity and an output.

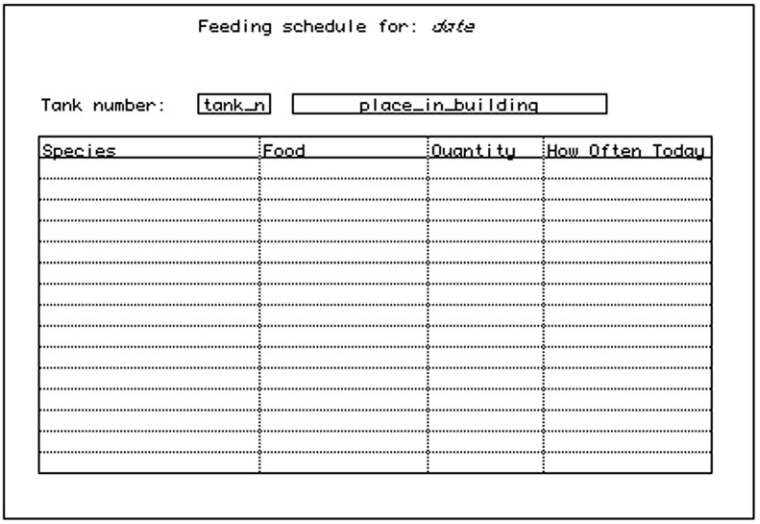
Note: One of the common mistakes made when designing the interface of database application programs is to use one data entry form per table. Users do not look at their environments in the same way as a database designer, however, and often the organization imposed by tables does not make sense to the users. Another benefit of prototyping is therefore that it forces database and application designers to adapt to what the users really need, rather than the other way around.

Food Management

One of the important functions mentioned by the aquarium's animal handlers was management of the aquarium feeding schedule (including what *should* be fed and what *was* fed), and the food inventory. First, they wanted a daily feeding schedule that could be printed and carried with them as they worked (for example, Figure 14.21). They also wanted to be able to record that feeding had occurred so that an updated feeding schedule could take prior feedings into account. Knowing that each species may eat more than one type of food and that each type of food can be eaten by many species, a database designer realizes that there are a number of entities required to implement what the users need:

- An entity that describes each species.
- An entity that describes each tank in the aquarium.
- An entity that describes a type of food.
- A composite entity between the *species* and *location* entities to record where a specific species can be found.
- A composite entity between a species and a type of food, recording which food a species eats, how much it eats, and how often it is fed.
- A composite entity between a species and a type of food, recording which food was fed to an animal, when it was fed, and how much it was fed.

Food inventory management—although it sounds like a separate application to the animal handlers—actually requires nothing more than the food entity. The food entity needs to store data about how much food is currently in stock (modified by data from the entity that describes what was fed and by manual entries made when shipments of food arrive) and a reorder point.

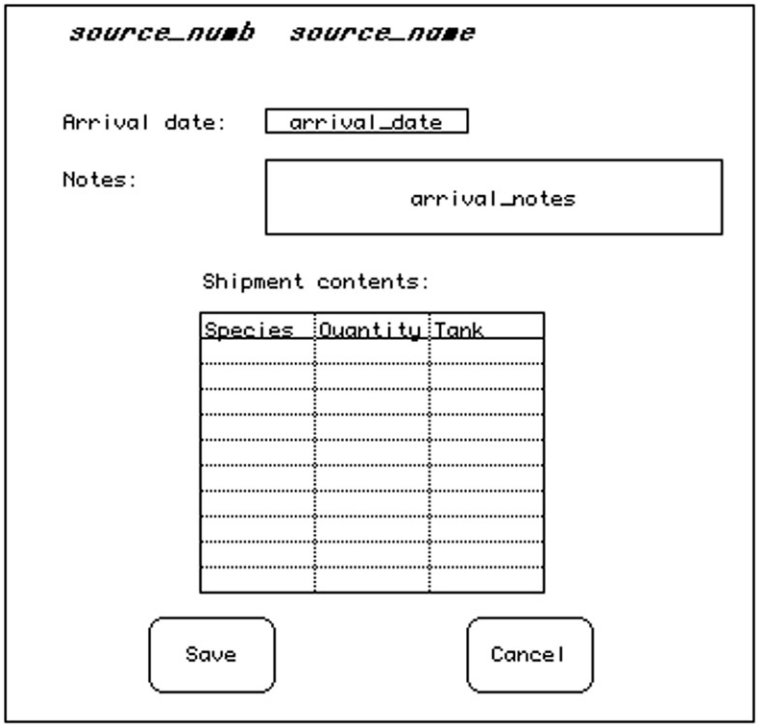


■ **FIGURE 14.21 Daily feeding schedule.**

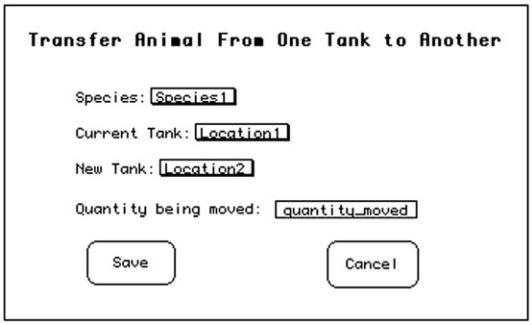
Handling Arriving Animals

When a shipment of animals arrives at the aquarium, animal handlers first check the contents of the shipment against the shipment's paperwork. They then take the animals and place them in the aquarium's quarantine area. The data entry form that the animal handlers will use to store data about arrivals therefore includes a place for entering an identifier for the tank in which the new animals have been placed (Figure 14.22). Given that the aquarium staff needs to be able to locate animals at any time, this suggests that the quarantine tanks should be handled no differently from the exhibit tanks and that there is only one entity for a tank.

After the quarantine period has expired and the animals are certified as healthy, they can be transferred to another location in the building. This means an application program must delete the species from their current tank (regardless of whether it is a quarantine tank or an exhibit tank) and insert data for the new tank. The screen form (Figure 14.23) therefore lets the user identify the species and its current location using popup menus. The user also uses a popup menu to identify the new location. To the database designer, this translates into the modification of one row (if the species is new to the exhibit tank) or the modification of one row and the deletion of another (if some of the species already live in the exhibit tank) in the table that represents the relationship between a species and a tank. All the database designer needs to do, however, is to provide the table; the application program will take care of managing the data modification.



■ **FIGURE 14.22 Recording the arrival of a shipment of animals.**



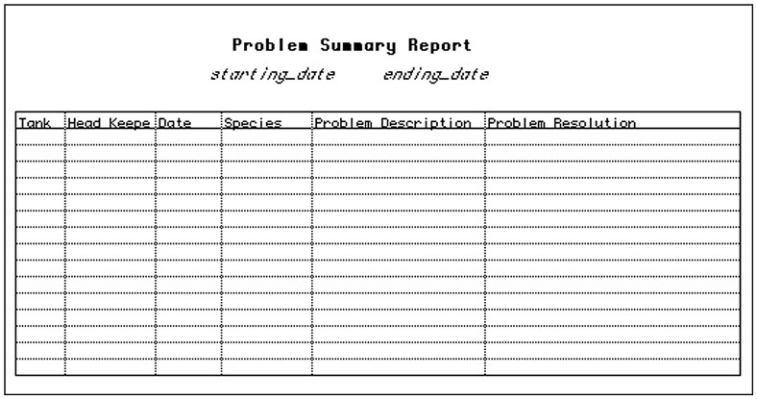
■ **FIGURE 14.23 Moving a species between tanks.**

Problem Analysis

The health of the animals in the aquarium is a primary concern of the animal handlers. They are therefore anxious to be able to analyze the problems that occur in the tanks for patterns. Perhaps a single species is experiencing more problems than any other; perhaps an animal handler is not

paying as much attention to the condition of the tanks for which he or she is responsible.

The animal handlers want the information in Figure 14.24 included in the problem summary report. What cannot be seen from the summary screen created by the CASE tool is that the data will appear as a control-break layout. For example, each tank number will appear only once; each species will appear once for each tank in which it was the victim of a problem. By the same token, each type of problem will appear once for each tank and species it affected. Each type of problem also will appear once for each tank and species it affected. Only the problem solutions will contain data for every row in the sample output table.

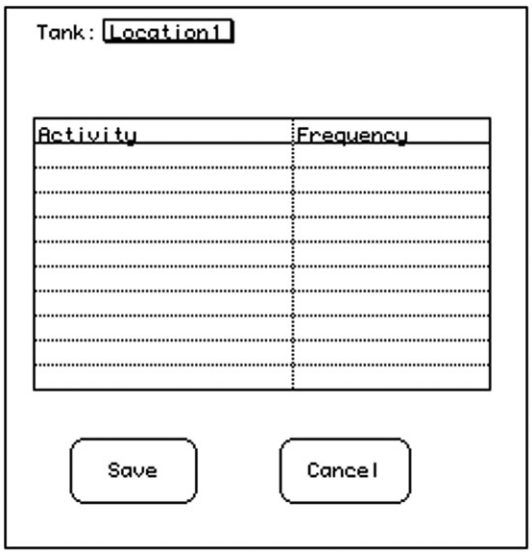


■ **FIGURE 14.24 Problem summary report.**

To a database designer, the form in Figure 14.24 suggests the need for five entities:

- The species.
- The tank.
- The type of problem.
- A problem occurrence (a type of problem occurring in one tank and involving one species).
- Problem solutions (one or more solutions tried for one problem occurrence). There may be many solutions to a single problem occurrence.

One of the best ways to handle problems is to avoid them. For this reason, the animal handlers also want to include maintenance data in their database. To move data entry simpler for the end users, the form for entering required maintenance (Figure 14.25) lets a user select a tank and then enter as many maintenance activities as needed.



■ **FIGURE 14.25 Entering required maintenance.**

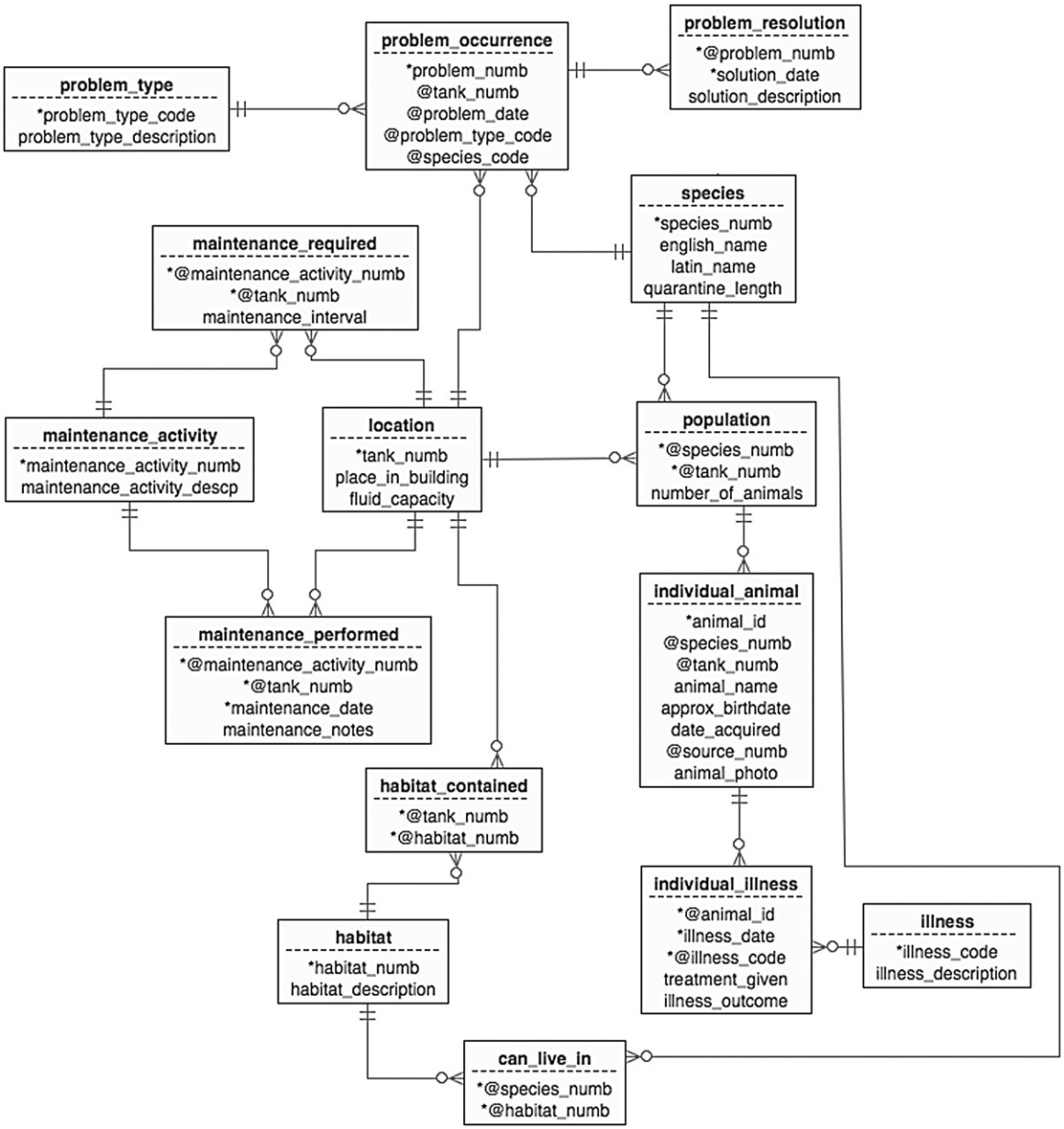
A database designer views such a form as requiring three entities: the tank, the maintenance activity, and the maintenance required for this tank (a composite entity between the tank and maintenance activity entities).

Creating the ER Diagram

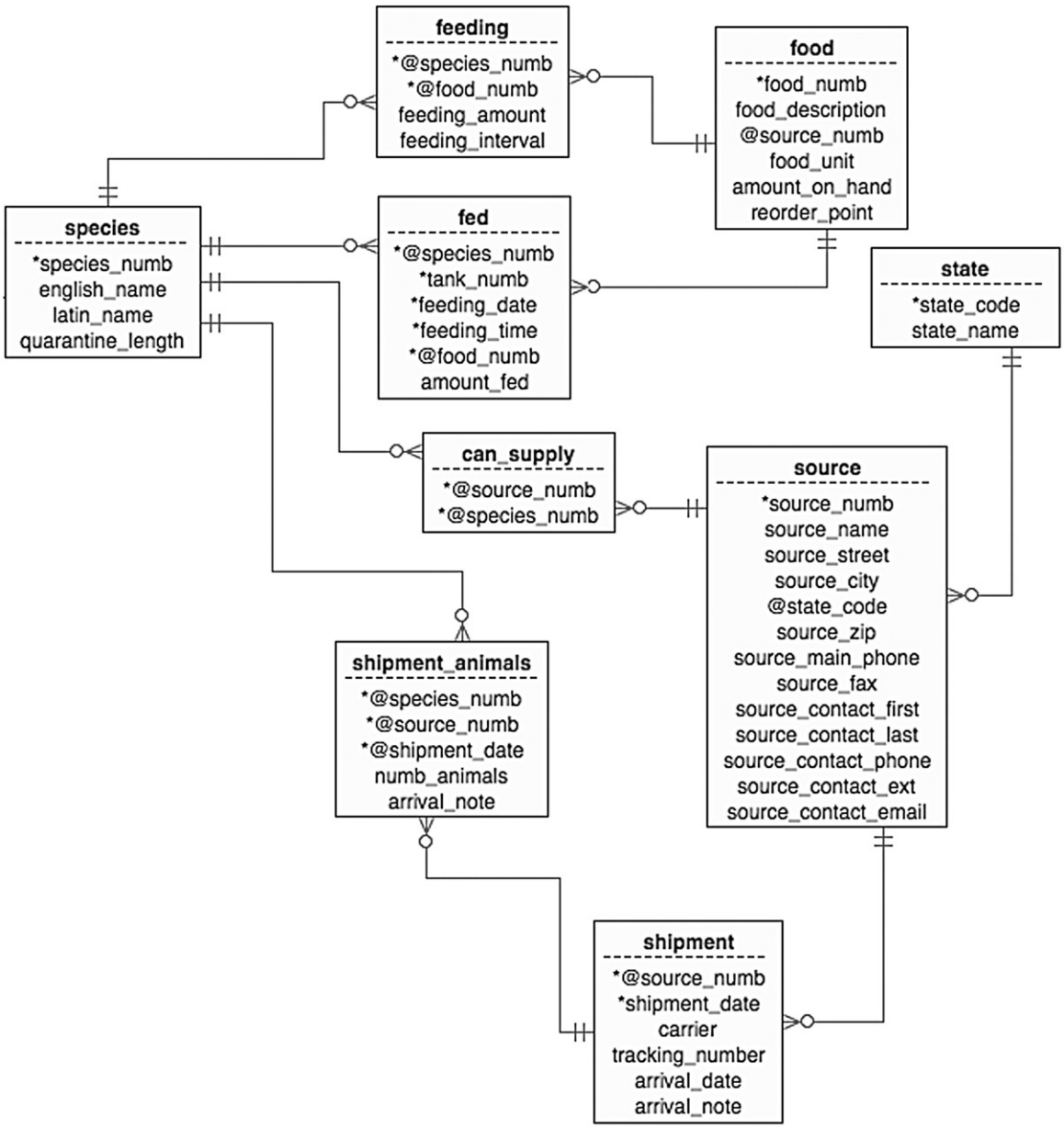
After refining the entire application prototype (and giving it a far more attractive design than the rather dull output produced by the CASE tool), the database designers for the East Coast Aquarium generate a large interconnected ER diagram. (Part I can be found in Figure 14.26; part II appears in Figure 14.27.) As you can see, when examining both diagrams, the centerpiece is the *species* entity, which participates in seven different relationships.

There are at least 11 many-to-many relationships represented by this design:

- Species to location
- Location to habitat
- Species to habitat
- Location to maintenance activity for required maintenance
- Location to maintenance activity for maintenance performed
- Location to problem
- Species to problem
- Species to food



■ **FIGURE 14.26 Animal handling ERD (part I).**



■ **FIGURE 14.27 Animal handling ERD (part II).**

- Species to source for ability of source to supply the species
- Shipment to species
- Illness to individual animal for tracking the condition of mammals and other large animals

The relationships involving location, problem, and species are particularly interesting. On the surface, there appears to be a many-to-many relationship between a tank and a type of problem. By the same token, there appears to be another many-to-many relationship between a species and a type of problem. The problem is that if the database maintained the two separate relationships, each with its own individual composite entity, then it will be impossible to determine which species was affected by which problem, in which tank. To resolve the issue, the designer uses a three-way composite entity—*problem_occurrence*— that relates three parent entities (*location*, *problem*, and *species*) rather than the traditional two. Semantically, a problem occurrence is one type of problem affecting one species in one location and therefore identifying it in the database requires all three parent entities.

In contrast, why is there no three-way composite entity between species, location, and habitat? As with the preceding example, there is a many-to-many relationship between species and location, and a many-to-many relationship between habitat and location. The answer, once again, lies in the meaning of the relationships. Were we to create a single composite entity relating all three entities, we would be asserting that a given species lives in a given habitat, in a given location. However, the animal handlers at the aquarium know that this type of data is not valid, particularly because if an animal lives in a tank with many habitats, it may move among multiple habitats. Instead, the relationship between species and habitat indicates all habitats in which a species can live successfully; the relationship between location and habitat indicates the habitats present in a tank.

The remainder of the many-to-many relationships are the typical two-parent relationships that you have been seeing throughout this book. The only aspect of these relationships that is the least bit unusual is the two relationships between maintenance activity and location. Each relationship has a different meaning (scheduled maintenance versus maintenance actually performed). The design therefore must include two composite entities, one to represent the meaning of each individual relationship.

Note: There is no theoretical restriction to the number of relationships that can exist between the same parent entities. As long as each relationship has a different meaning, there is usually justification for including all of them in a database design.

Creating the Tables

The ER diagrams translate to the following tables:

```
species (species_numb, English_name, latin_name,
      quarantine_length)

location (tank_numb, place_in_building, fluid_capacity)

population (species_numb, tank_numb, number_of_animals)

individual_animal (animal_id, species_numb, tank_numb,
      animal_name, approx_birthdate, source_numb, animal_photo)

illness (illness_code, illness_description)

individual_illness (animal_id, illness_date, illness_code,
      treatment_given, illness_outcome)

habitat (habitat_numb, habitat_description)

habitat_contained (tank_numb,habitat_numb)

can_live_in (species_numb, habitat_numb)

problem_type (problem_type_code, problem_type_description)

problem_occurrence (problem_numb, tank_numb, problem_date,
      problem_type_code, species_code)

problem_resolution (problem_numb, solution_date,
      solution_description)

maintenance_activity (maintenance_activity_numb,
      maintenance_activity_desc)

maintenance_required (maintenance_activity_numb, tank_numb,
      maintenance_interval)

maintenance_performed (maintenance_activity_numb, tank_numb,
      maintenance_date, maintenance_notes)

food (food_numb, food_description, source_numb, food_unit,
      amount_on_hand, reorder_point)

feeding (species_numb, food_numb, feeding_amount,
      feeding_interval)

fed (species_numb, tank_numb, feeding_date, feeding_time,
      food_numb, amount_fed)
```

```
state (state_code, state_name)

source (source_numb, source_name, source_street, source_city,
     state_code, source_zip, source_main_phone, source_fax,
     source_contact_first, source_contact_last,
     source_contact_phone, source_contact_ext,
     source_contact_email)

can_supply (source_numb, species_numb)

shipment (source_numb, shipment_nate, carrier,
     tracking_number, arrival_date, arrival_note)

shipment_animals (species_numb, source_numb, shipment_date,
     numb_animals, arrival_note)
```

Choosing a primary key for the *problem_occurrence* table presents a bit of a dilemma. Given that a problem occurrence represents a relationship between a problem type, tank, and species, the theoretically appropriate primary key is a concatenation of the problem type, tank number, species number, and problem date. However, this is an extremely awkward primary key to use as a foreign key in the *problem_resolution* table. Although it is unusual to give composite entities arbitrary unique keys, in this case it makes good practical sense.

There are several tables in this design that are "all key" (made up of nothing but the primary key). According to the CASE tool used to draw the ER diagram, this represents an error in the design. However, there is nothing in relational database theory that states that all-key relations are not allowed. In fact, they are rather common when they are needed to represent a many-to-many relationship that has no accompanying relationship data.

Generating the SQL

The SQL CREATE statements that generate the animal tracking database for East Coast Aquarium can be found in Figure 14.28. Because of the large number of composite entities, there are also a large number of foreign keys. Other than that, the SQL presents no unusual features.

```
CREATE TABLE state
(
  state_code char (2),
  state_name varchar (20),
  CONSTRAINT PK_STATE PRIMARY KEY (state_code)
);

CREATE TABLE source
(
  source_numb integer,
  source_name char (15),
  source_street varchar (500,
  source_city varchar (50),
  state_code char (2),
  source_zip char (10),
  source_main_phone char (10),
  source_fax char (10),
  source_contact_first char (15),
  source_contact_last char (15),
  source_contact_phone char (10),
  source_contact_ext char (5),
  source_contact_email varchar (256),
  CONSTRAINT PK_SOURCE PRIMARY KEY (source_numb),
  CONSTRAINT Relationstatesource1 FOREIGN KEY () REFERENCES STATE
);

CREATE TABLE shipment
(
  source_numb integer,
  shipment_date date,
  carrier varchar (30),
  tracking_number char (20),
  arrival_date date,
  arrival_note varchar (50,
  CONSTRAINT PK_SHIPMENT PRIMARY KEY (source_numb,shipment_date)
);

CREATE TABLE species
(
  species_numb integer,
  engish_name varchar (256),
  latin_name varchar (256),
  quarantine_length integer,
  CONSTRAINT PK_species PRIMARY KEY (species_numb)
);
```

■ **FIGURE 14.28 SQL statements prepared by a CASE tool for the animal tracking database (continues).**

```
CREATE TABLE shipment_animals
(
  species_numb integer,
  source_numb integer,
  shipment_date date,
  numb_animals integer,
  arrival_note varchar (256),
  CONSTRAINT PK_SHIPMENTANIMALS PRIMARY KEY (species_numb,source_numb,shipment_date)
);

CREATE TABLE can_supply
(
  source_numb integer,
  species_numb integer,
  CONSTRAINT PK_CAN_SUPPLY PRIMARY KEY (source_numb,species_numb),
  CONSTRAINT Relationspeciescan_supply1 FOREIGN KEY ()
      REFERENCES species,
  CONSTRAINT Relationsourcecan_supply1 FOREIGN KEY () REFERENCES SOURCE
);

CREATE TABLE food
(
  food_numb integer,
  food_description varchar (256),
  source_numb integer,
  food_unit char (10),
  amount_on_hand integer,
  reorder_point integer,
  CONSTRAINT PK_food PRIMARY KEY (food_numb)
);

CREATE TABLE fed
(
  species_numb integer,
  feeding_date date,
  feeding_time time,
  tank_numb integer,
  food_numb integer,
  amount_fed integer,
  CONSTRAINT PK_fed PRIMARY KEY
      (species_numb,tank_numb,feeding_date,feeding_time,food_numb),
  CONSTRAINT Relationspeciesfed1 FOREIGN KEY () REFERENCES species,
  CONSTRAINT Relationfoodfed1 FOREIGN KEY () REFERENCES food
);
```

■ **FIGURE 14.28 (*cont.*)**

```
CREATE TABLE feeding
(
  species_numb integer,
  food_numb integer,
  feeding_amount integer,
  feeding_interval interval,
  CONSTRAINT PK_feeding PRIMARY KEY (species_numb,food_numb),
  CONSTRAINT Relationspeciesfeeding1 FOREIGN KEY () REFERENCES species,
  CONSTRAINT Relationfoodfeeding1 FOREIGN KEY () REFERENCES food
);

CREATE TABLE location
(
  tank_numb integer,
  place_in_building char (6),
  fluid_capacity integer,
  CONSTRAINT PK_location PRIMARY KEY (tank_numb)
);

CREATE TABLE problem_type
(
  problem_type_code integer,
  problem_type_description varchar(256),
  CONSTRAINT PK_problem_type PRIMARY KEY (problem_type_code)
);

CREATE TABLE problem_occurrence
(
  problem_numb integer,
  tank_numb integer,
  problem_date date,
  problem_type_code integer,
  species_code integer,
  CONSTRAINT PK_problem_occurrence PRIMARY KEY (problem_numb),
  CONSTRAINT Relationproblem_typeproblem_occurrence1 FOREIGN KEY ()
      REFERENCES problem_type,
  CONSTRAINT Relationproblem_occurrencelocation1 FOREIGN KEY ()
      REFERENCES location,
  CONSTRAINT Relationspeciesproblem_occurrence1 FOREIGN KEY ()
      REFERENCES species
);

CREATE TABLE problem_resolution
(
  problem_numb integer,
  solution_date date,
  solution_description varchar (256),
  CONSTRAINT PK_problem_resolution## PRIMARY KEY
      (problem_numb,solution_date),
  CONSTRAINT Relationproblem_occurrenceproblem_resolution1
      FOREIGN KEY () REFERENCES problem_occurrence
);
```

■ **FIGURE 14.28** (***cont.***)

```
CREATE TABLE habitat
(
  habitat_numb integer,
  habitat_description varchar (256),
  CONSTRAINT PK_habitat PRIMARY KEY (habitat_numb)
);

CREATE TABLE can_live_in
(
  species_numb integer,
  habitat_numb integer,
  CONSTRAINT PK_can_live_in PRIMARY KEY (species_numb,habitat_numb),
  CONSTRAINT Relationhabitatcan_live_in1 FOREIGN KEY ()
       REFERENCES habitat
);

CREATE TABLE habitat_contained
(
  tank_numb integer,
  habitat_numb integer,
  CONSTRAINT PK_habitat_contained PRIMARY KEY (tank_numb,habitat_numb),
  CONSTRAINT Relationlocationhabitat_contained1 FOREIGN KEY ()
      REFERENCES location,
  CONSTRAINT Relationhabitathabitat_contained1 FOREIGN KEY ()
      REFERENCES habitat
);

         CREATE TABLE maintenance_activity
(
  maintenance_activity_numb integer,
  maintenance_activity varchar (256),
  CONSTRAINT PK_maintenance_activity
         PRIMARY KEY (maintenance_activity_numb)
);

CREATE TABLE maintenance_performed
(
  maintenance_activity_numb integer,
  tank_numb integer,
  maintenance_date date,
  maintenance_notes varchar (256),
  CONSTRAINT PK_maintenance_performed PRIMARY KEY
         (maintenance_activity_numb,tank_numb,maintenance_date),
  CONSTRAINT Relationmaintenance_activitymaintenance_performed1
         FOREIGN KEY () REFERENCES maintenance_activity,
  CONSTRAINT Relationlocationmaintenance_performed1 FOREIGN KEY ()
         REFERENCES location
);
```

■ **FIGURE 14.28** (***cont.***)

```
CREATE TABLE maintenance_required
(
  maintenance_activity_numb integer,
  tank_numb integer,
  maintenance_interval interval,
  CONSTRAINT PK_maintenance_required PRIMARY KEY
         (maintenance_activity_numb,tank_numb),
  CONSTRAINT Relationmaintenance_requiredmaintenance_activity1
         FOREIGN KEY () REFERENCES maintenance_activity,
  CONSTRAINT Relationlocationmaintenance_required1 FOREIGN KEY ()
         REFERENCES location
);

CREATE TABLE illness
(
  illness_code integer,
  illness_description varchar (256),
  CONSTRAINT PK_illness PRIMARY KEY (illness_code)
);

CREATE TABLE population
(
  species_numb integer,
  tank_numb integer,
  number_of_animals integer,
  CONSTRAINT PK_population PRIMARY KEY (species_numb,tank_numb),
  CONSTRAINT Relationspeciespopulation1 FOREIGN KEY ()
         REFERENCES species,
  CONSTRAINT Relationlocationpopulation1 FOREIGN KEY ()
         REFERENCES location
);

CREATE TABLE individual_animal
(
  animal_id integer,
  species_numb integer,
  tank_numb integer,
  animal_name varchar (50),
  approx_birthdate char (10),
  date_acquired date,
  source_numb integer,
  animal_photo blob,
  CONSTRAINT PK_individual_animal PRIMARY KEY (animal_id),
  CONSTRAINT Relationpopulationindividual_animal1 FOREIGN KEY ()
         REFERENCES population
);
```

■ **FIGURE 14.28** (***cont.***)

```
CREATE TABLE individual_illness
(
  animal_id integer,
  illness_date date,
  illness_code integer,
  treatment_given varchar (256),
  illness_outcome varchar (256),
  CONSTRAINT PK_individual_illness
         PRIMARY KEY (animal_id,illness_date,illness_code),
  CONSTRAINT Relationindividual_animalindividual_illness1
         FOREIGN KEY () REFERENCES individual_animal,
  CONSTRAINT Relationillnessindividual_illness1
         FOREIGN KEY () REFERENCES illness
);
```

■ **FIGURE 14.28** (***cont.***)

Chapter 15 Database Design Case Study #3: SmartMart

Many retail chains today maintain both a Web and a brick-and-mortar presence in the marketplace. Doing so presents a special challenge for inventory control because the inventory is shared between physical stores and Web sales. The need to allow multiple shipping addresses and multiple payment methods within a single order also adds complexity to Web selling. Online shopping systems also commonly allow users to store information about multiple credit cards.

To familiarize you with what is necessary to maintain the data for such a business, we'll be looking at a database for SmartMart, a long-established retailer with 325 stores across North America that has expanded into Web sales. SmartMart began as a local grocery store, but over the years has expanded to carry clothing, sundries, home furnishings, hardware, and electronics. Some stores still have grocery departments; others carry just "dry" goods.

In addition to the retail stores, SmartMart maintains four regional warehouses that supply the stores as well as ship products ordered over the Web.

The Merchandising Environment

SmartMart has three major areas for which it wants an integrated database: in-store sales, Web sales, and some limited Human Resources needs. The sales data must be compatible with accounting systems to simplify data transfer. In addition, both the in-store sales and Web sales applications must use the same data about products.

Product Requirements

The products that SmartMart sells are stocked throughout the company's stores, although every store does not carry every product. The database must therefore include data about the following:

- Products
- Stores
- Departments within stores
- Products stocked within a specific department
- Current sales promotions for a specific product

The store and department data will need to be integrated with the database's Human Resources data.

In-Store Sales Requirements

The data describing in-store sales serve two purposes: accounting and inventory control. Each sale (whether paid with cash or credit) must decrement inventory and provide an audit trail for accounting.

Retaining data about in-store sales is also essential to SmartMart's customer service reputation. The company offers a 15-day return period, during which a customer can return any product with which he or she is not satisfied. Store receipts therefore need to identify entire sales transactions, in particular which products were purchased during a specific transaction.

Because the company operates in multiple US states, there is a wide variety of sales tax requirements. Which products are taxed varies among states, as well as the sales tax rates. The database must therefore include sales tax where necessary as a part of an in-store transaction.

The database must distinguish between cash and credit transactions. The database will not store customer data about cash transactions, but must retain card numbers, expiration dates, and customer names on credit sales.

Web Sales Requirements

Web sales add another layer of complexity to the SmartMart data environment. The Web application must certainly have access to the same product data as the in-store sales, but must also integrate with a shopping cart application.

To provide the most flexibility, SmartMart wants to allow customers to store multiple shipping addresses, to ship to multiple addresses on the same order,

and to store multiple credit card data from which a customer can choose when checking out. In addition, customers are to be given a choice as to whether to pick up their order or have it shipped. The Web application must therefore have access to data about which stores are in a customer's area and which products are available at each store, the same inventory information used by in-store applications.

Finally, the Web application must account for backorders and partial shipments. This means that a shipment is not the same as a Web order, whereas an in-store sale delivers its items at one time. (Should an in-store customer want to purchase an item that is not in stock, the item will be handled as if it were a Web order.)

Personnel Requirements

Although a complete personnel database is beyond the scope of this case, SmartMart's management does want to be able to integrate some basic HR functions into the database, especially the scheduling of "sales associates" to specific departments in specific stores. The intent is to eventually be able use an expert system to analyze sales and promotion data to determine staffing levels and to move employees among the stores in a region as needed.

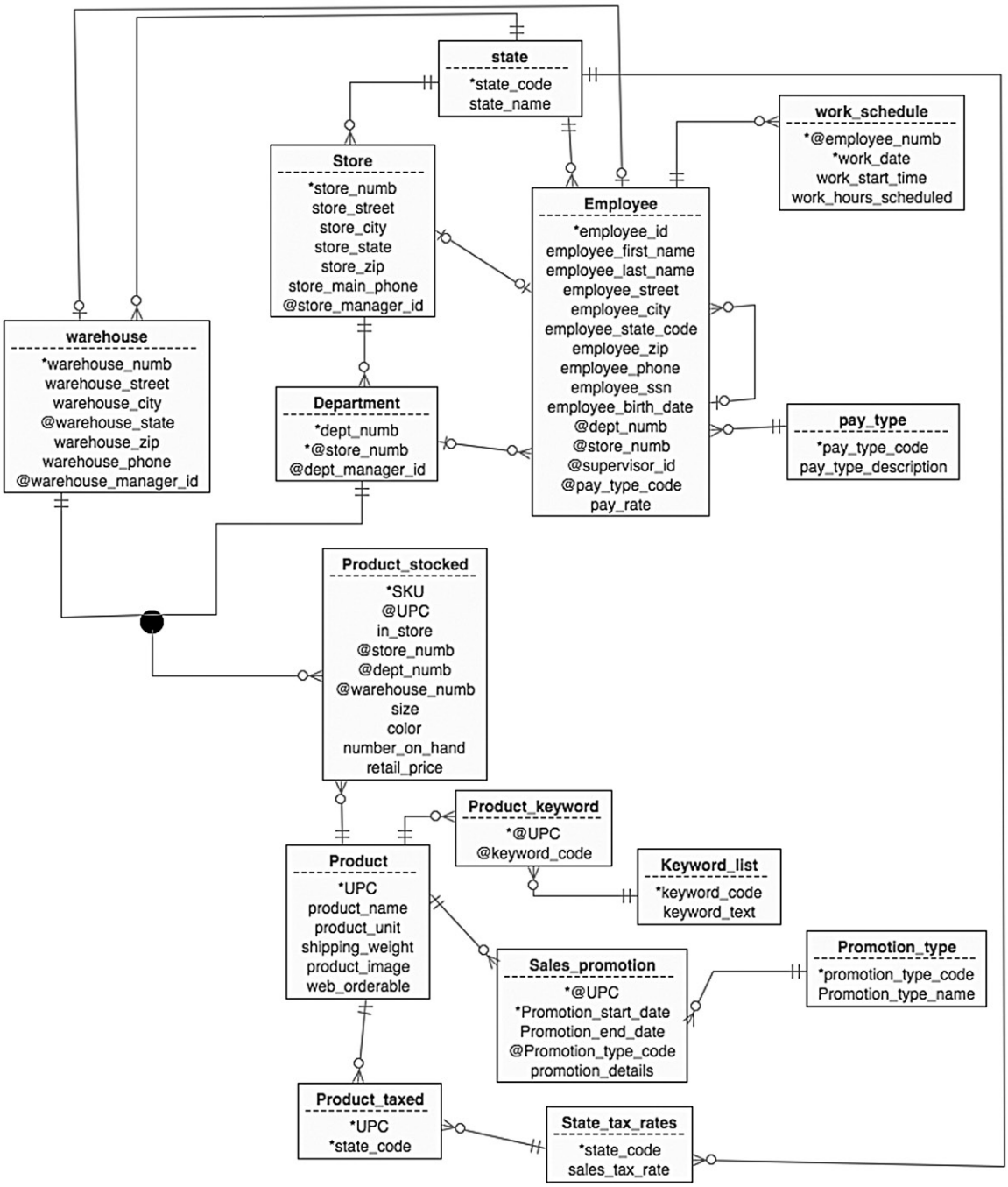
Putting Together an ERD

As you might expect, the SmartMart ERD is fairly large. It therefore has been broken into three parts to make it easier to see and understand.

Stores, Products, and Employees

As you can see in Figure 15.1 (the first third of the ERD), the SmartMart database begins with four "foundation" entities (entities that are only at the "one" end of 1:M relationships): *employee*, *store*, *warehouse*, and *product*.

The *store* and *warehouse* entities, at least at this time, have exactly the same attributes. It certainly would be possible to use a single entity representing any place products were kept. This would remove some of the complexity that arises when locating a product. However, there is no way to be certain that the data stored about a store and a warehouse will remain the same over the life of the database. Separating them into two entities after the database has been in use for some time would be very difficult and time



■ **FIGURE 15.1 The SmartMart ERD (part 1).**

consuming. Therefore, the choice was made to handle them as two distinct entities from the beginning.

Reference Entities

There are also several entities that are included for referential integrity purposes (*state*, *keyword_list*, *pay_type*, *promotion_type*). These entities become necessary because the design attempts to standardize text descriptions by developing collections of acceptable text descriptions and then using a numbering scheme to link the descriptions to places where they are used. There are two major benefits to doing this.

First, when the text descriptions, such as a type of pay (for example, hourly versus salaried), are standardized, searches will be more accurate. Assume, for example, that the pay types haven't been standardized. An employee who is paid hourly might have a pay type of "hourly," "HOURLY," "hrly," and so on. A search that retrieves all rows with a pay type of "hourly" will miss any rows with "HOURLY" or "hrly," however.

Second, using integers to represent values from the standardized list of text descriptions saves space in the database. Given the relative low price of disk storage, this usually isn't a major consideration.

The drawback, of course, is that when you need to search on or display the text description, the relation containing the standardized list and the relation using the integers representing the terms must be joined. Joins are relatively slow activities, but, in this case, the reference relation containing the master list of text descriptions will be relatively small; the join uses integer columns, which are quick to match. Therefore, unless for some reason the reference relation becomes extremely large, the overhead introduced by the join is minimal.

Circular Relationships

If you look closely at the *employee* entity in Figure 15.1, you'll see a relationship that seems to relate the entity to itself. In fact, this is exactly what that circular relationship does. It represents the idea that a person who supervises other employees is also an employee: The *supervisor_id* attribute is drawn from the same domain as *employee_id*. Each supervisor is related to many employees and each employee has only one supervisor.

There is always a temptation to create a separate *supervisor* entity. Given that a supervisor must also be an employee, however, the *supervisor* entity would contain data duplicated from the *employee* entity. This means that we introduce unnecessary duplicated data into the database and run a major risk of data inconsistencies.

Note: To retrieve a list of supervisors and who they supervise, someone using SQL would join a copy of the employee table to itself, matching the supervisor_id column in one table to the employee_id column in the other. The result table would contain data for two employees in each row (the employee and the employee's supervisor) that could be manipulated—in particular, sorted—for output, as needed.

Mutually Exclusive Relationships

There is one symbol on the ERD in Figure 15.1 that has not been used before in this book: the small circle that sits in the middle of the relationships between a stocked product, a department (in a store), and a warehouse. This type of structure indicates a *mutually exclusive* relationship. A given product can be stocked in a store or in a warehouse, but not both. (This holds true for this particular data environment because a product stocked represents physical items to be sold.)

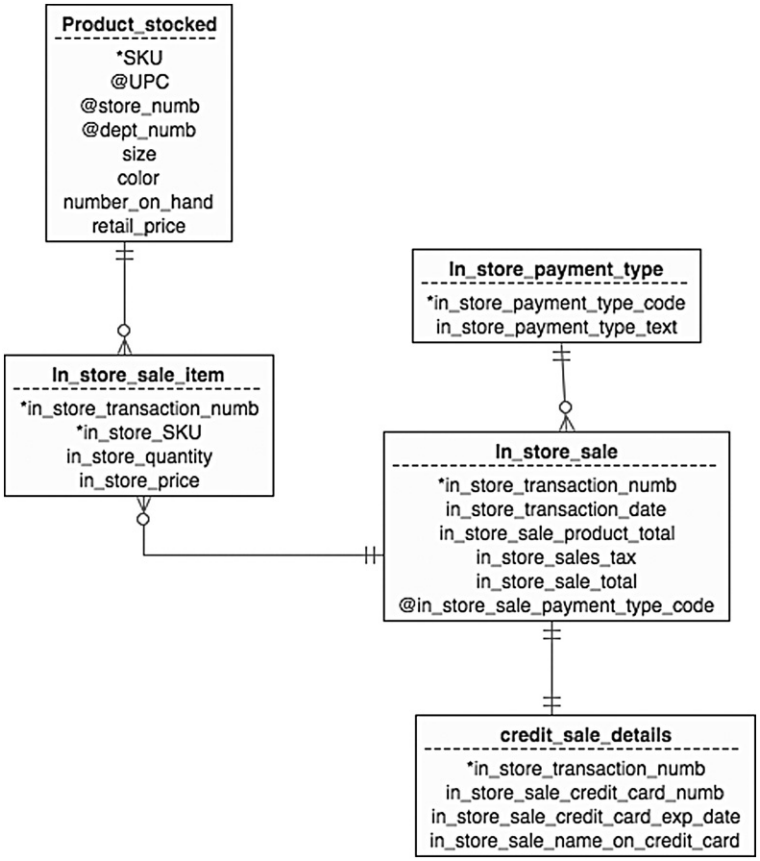
The structure of the *product_stocked* entity reflects its participation in this type of relationship. In particular, it contains a Boolean column (*in_store*) that holds a value of true if the product is stocked in a store; a value of false indicates that the product is stocked in a warehouse. The value of the *in_store* attribute will then tell a user whether to use the *warehouse_numb* column or the concatenation of the *store_numb* column with the *dept_numb* attribute to find the actual location of an item.

One-to-one Relationships

Earlier in this book, you read that true one-to-one relationships are relatively rare. There are, however, three of them visible in Figure 15.1. All involve employees that manage something: a store, a department within a store, or a warehouse. A corporate unit may have one manager at a time, or it may have no manager; an employee may be the manager of one corporate unit or the manager of none. It is the rules of this particular database environment that make the one-to-one relationships valid.

In-store Sales

The second part of the ERD (Figure 15.2) deals with in-store sales. The data that are common to cash and credit sales are stored in the *in_store_sale* entity. These data are all that are needed for a cash sale. Credit sales, however, require data about the credit card used (the *credit_sale_details* entity). Notice that there is therefore a one-to-one relationship between *in_store_sale* and *credit_sale_details*. The two-entity design is not



■ **FIGURE 15.2 The SmartMart ERD (part 2).**

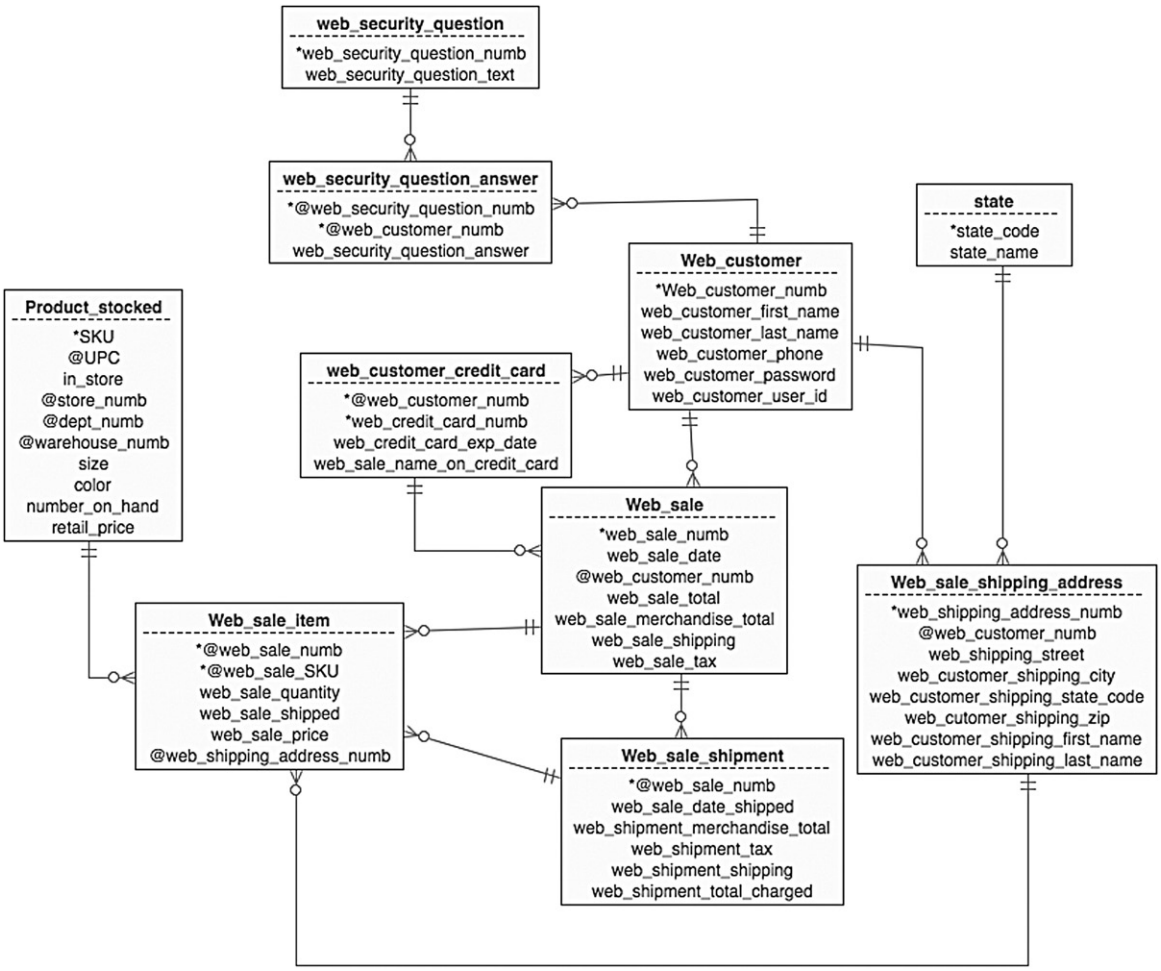
required for a good database design, but has been chosen for performance reasons with the assumption that although currently there are more credit/debit than cash transactions, the data retrieval is very skewed, requiring customer data far more frequently than credit card details. It therefore is a way of providing horizontal partitioning for an access pattern that is "unbalanced." In other words, a small proportion of the queries of sales data will need credit data.

After SmartMart's database has been in production for some time, the database administrator can look at the actual proportion of retrievals that need the credit card details. If a large proportion of the retrievals include the credit/debit data, then it may make sense to combine *in_store_sale* and *credit_sale_details* into a single entity and simply leave the credit detail

columns as null for cash sales. Although some space will be wasted, the combined design avoids the need to perform a lengthy join when retrieving data about a credit sale.

Web Sales

The third portion of the ERD (Figure 15.3) deals with Web sales. Each Web sale uses only one credit card, but a Web customer may keep more than



■ **FIGURE 15.3 The SmartMart ERD (part 3).**

one credit card number within SmartMart's database. A customer may also keep more than one shipping address; multiple shipping addresses can be used within the same order. (Multiple credit cards for a single order are not supported.)

At first glance, it might appear that the three relationships linking *web_customer*, *web_sale*, and *web_customer_credit_card* form a circular relationship. However, the meaning of the relationships is such that the circle does not exist:

- The direct relationship between *web_customer* and *web_customer_credit_card* represents the cards that a web customer has allowed to be stored in the SmartMart database. The credit card data are retrieved when the customer is completing a purchase. He or she chooses one for the current order.
- The relationship between *web_customer* and *web_sale* connects a purchase to a customer.
- The relationship between *web_customer_credit_card* and *web_sale* represents the credit card used for a specific order.

It can be difficult to discern such subtle differences in meaning from a simple ERD. There are two solutions: define the meaning of the relationships in the data dictionary or add relationship names to the ERD.

The *web_sale_shipping_address* entity is used to display addresses from which a user can choose. However, because items within the same shipment can be sent to different addresses, there is a many-to-many relationship between *web_sale* and *web_sale_shipping_address*. The *web_sale_item* resolves that many-to-many relationship into two one-to-many relationships.

The relationship between *web_sale*, *web_sale_item*, and *web_sale_shipment* is ultimately circular, as it appears in the ERD. However, at the time the order is placed, there are no instances of the *web_sale_shipment* entity because no items have shipped. The circle is closed when items actually ship. SmartMart's database can handle multiple shipments for a single order. However, each item on an order is sent to the customer in a single shipment. How do we know this? Because here is a 1:M relationship between a web sale and a web sale shipment (a sale may have many shipments) and a 1:M relationship between a shipment and an item on the order (each item is shipped as part of only one shipment).

Creating the Tables

The ERDs you have just seen produce the following tables, listed in alphabetical order:

```
credit_sale_details (in_store_transaction_numb,
     in_store_sale_credit_card_numb, in_store_exp_date,
     in_store_sale_name_on_credit_card)

department (dept_numb, store_numb, dept_manager_id)

employee (employee_id, employee_first_name, employee_last_name,
     employee_street, employee_city, employee_state_code,
     employee_zip, employee_phone, employee_ssn,
     employee_birth_date, dept_id, store_numb, supervisor_id,
     pay_type_code, pay_rate)

in_store_payment_type (in_store_payment_type_code,
     in_store_payment_type_text)

in_store_sale (in_store_transaction_numb,
     in_store_transaction_date, in_store_sale_product_total,
     in_store_sales_tax,in_store_total,
     in_store_payment_type_code)

in_store_sale_item (in_store_sale_transaction_numb,
     in_store_SKU, in_store_quantity, in_store_price)

keyword_list (keyword_code, keyword_text)

pay_type (pay_type_code, pay_type_description)

product (UPC, product_name, product_unit, shipping_weight,
     product_image, web_orderable)

product_keyword (UPC, keyword_code)

product_stocked (SKU, UPC, in_store, store_numb, dept_numb,
     warehouse_numb, size, color, number_on_hand, retail_price)

product_taxed (UPC, state_code)

promotion_type (promotion_type_code, promotion_type_name)

sales_promotion (UPC, promotion_start_date, promotion_end_date,
     promotion_type_code, promotion_details)

state (state_code, state_name)

state_tax_rates (state_code, sales_tax_rate)

store (store_numb, store_street, store_city, store_state_code,
     store_zip, store_main_phone, store_manager_id)
```

```
warehouse (warehouse_id, warehouse_street, warehouse_city,
     ware_house_state_code, warehouse_zip, warehouse_phone,
     warehouse_manager_id)

web_customer (web_customer_numb, web_customer_first_name,
     web_customer_last_name, web_customer_phone,
     web_customer_password, web_customer_user_id)

web_customer_credit_card (web_customer_numb,
     web_credit_card_numb, web_credit_card_exp_date,
     web_sale_name_on_credit_card)

web_sale (web_sale_numb, web_sale_date, web_customer_numb,
     web_sale_total, web_sale_merchandise_total,
     web_sale_shipping,web_sale_tax)

web_sale_item (web_sale_numb, web_sale_SKU,
     web_sale_quantity, web_sale_shipped, web_sale_price,
     web_shipping_address_numb)

web_sale_shipment (web_sale_numb, web_sale_date_shipped,
     web_shipment_merchandise_total, web_shipment_tax,
     web_shipment_shipping, web_shipment_total_charged)

web_sale_shipping_address (web_shipping_address_numb,
     web_customer_numb, web_shipping_street,
     web_shipping_city, web_shipping_state_code,
     web_customer_zip, web_customer_shipping_first_name,
     web_customer_shipping_last_name)

web_security_question (web_security_question_numb,
     web_security_question_text)

web_security_question_answer (web_security_question_numb,
     web_customer_numb, web_security_answer_text)

work_schedule (employee_id, work_date, work_start_time,
     work_hours_scheduled)
```

Because of the circulation relationship between a supervisor, who must be an employee, and an employee being supervised, the employee table contains a foreign key that references the primary key of its own table: *supervisor_id* is a foreign key referencing *employee_id*. There is nothing wrong with this type of design. The definition of a foreign key states only that the foreign key is the same as the primary key of some table; it does not rule out the foreign key referencing the primary key of its own table.

Generating the SQL

The case tool that generated the SQL misses some very important foreign keys—the relationships between employees and various other entities—because the two columns don't have the same name. Therefore, the database designer must add the constraints manually to the foreign key tables' CREATE TABLE statements:

```
employee table: FOREIGN KEY (supervisor_id)
     REFERENCES employee (employee_id)

warehouse table: FOREIGN KEY (warehouse_manager_id)
     REFERENCES employee (employee_id)

department table: FOREIGN KEY (department_manager_id)
     REFERENCES employee (employee_id)

store table: FOREIGN KEY (store_manager_id)
     REFERENCES employee (employee_id)
```

The manual foreign key insertions could have been avoided, had the manager IDs in the warehouse, department, and store tables been given the name *employee_id*. However, it is more important in the long-run to have the column names reflect the meaning of the columns.

There is also one foreign key missing of the two needed to handle the mutually exclusive relationship between a product stocked and either a department (in a store) or a warehouse. The foreign key from *product_stocked* to department is present, but not the reference to the warehouse table. The database designer must, therefore, add the following to the *product_stocked* table:

```
FOREIGN KEY (warehouse_numb)
     REFERENCES warehouse (warehouse_numb)
```

Won't there be a problem with including the two constraints, if only one can be valid for any single row in *product_stocked*? Not at all. Remember that referential integrity constraints are only used when the foreign key is not null. Therefore, a product stocked in a warehouse will have a value in its *warehouse_numb* column, but null in the *store_numb* and *dept_numb* columns. The reverse is also true: A product stocked in a store will have values in its *store_numb* and *dept_numb* columns, but null in the *warehouse_numb* column.

The foreign key relationships to the *state* reference relation must also be added manually, because the foreign key columns do not have the same name as the primary key column in the *state* table. Whether to use the same name throughout the database (*state_name*) is a design decision. If all tables that contain a state use the same name, foreign keys will be added automatically by the CASE tool. However, depending on the way in which the states are used, it may be difficult to distinguish among them, if the column names are all the same (Figure 15.4).

```
CREATE TABLE state
(
  state_code char (2),
  state_name char (15),
  CONSTRAINT PK_state PRIMARY KEY (state_code),
  CONSTRAINT RelationstateState_tax_rates1 FOREIGN KEY ()
      REFERENCES state_tax_rates
);

CREATE TABLE warehouse
(
  warehouse_id integer,
  warehouse_street char (50),
  warehouse_city char (3),
  wharehouse_state_code char (2),
  warehouse_zip char (10),
  warehouse_phone char (12),
  warehouse_manager_id integer,
  CONSTRAINT PK_warehouse PRIMARY KEY (warehouse_id)
);

CREATE TABLE state_tax_rates
(
  state_code char (2),
  sales_tax_rate number (5,20),
  CONSTRAINT PK_state_tax_rates PRIMARY KEY (state_code),
  CONSTRAINT RelationstateState_tax_rates1 FOREIGN KEY ()
      REFERENCES state
);

CREATE TABLE product
(
  UPC char (15),
  product_name varchar (256),
  product_unit char (10),
  shipping_weight integer,
  product_image blob,
  web_orderable boolean,
  CONSTRAINT PK_product PRIMARY KEY (UPC)
);

CREATE TABLE product_taxed
(
  UPC integer,
  state_code integer,
  CONSTRAINT PK_product_taxed PRIMARY KEY (UPC,state_code),
  CONSTRAINT RelationProductProduct_taxed1 FOREIGN KEY ()
      REFERENCES product,
  CONSTRAINT RelationState_tax_ratesProduct_taxed1 FOREIGN KEY ()
      REFERENCES state_tax_rates
);
```

■ **FIGURE 15.4 SQL CREATE statements for the SmartMart database (continues).**

```
CREATE TABLE In_store_payment_type
(
  in_store_payment_type_code integer,
  in_store_payment_type_text char (10),
  CONSTRAINT PK_In_store_payment_type PRIMARY KEY
      (in_store_payment_type_code)
);

CREATE TABLE web_customer
(
  web_customer_numb integer,
  web_customer_first_name char (15),
  web_customer_last_name INTEGER,
  web_customer_phone char (12),
  web_customer_password char (15),
  web_customer_user_id char (15),
  CONSTRAINT PK_web_customer PRIMARY KEY (web_customer_number)
);

CREATE TABLE web_sale_shipping_address
(
  web_shipping_address_numb integer,
  web_customer_numb integer,
  web_shipping_street char (50),
  web_customer_shipping_city char (50),
  web_customer_shipping_state_code char (2),
  eb_cutomer_shipping_zip char (10),
  web_customer_shipping_first_name char (15),
  web_customer_shipping_last_name char (15),
  CONSTRAINT PK_web_sale_shipping_address
      PRIMARY KEY (web_shipping_address_numb),
  CONSTRAINT RelationWeb_customerWeb_sale_shipping_address1
      FOREIGN KEY () REFERENCES web_customer,
  CONSTRAINT RelationstateWeb_sale_shipping_address1
      FOREIGN KEY () REFERENCES state
);

CREATE TABLE web_customer_credit_card
(
  web_customer_numb integer,
  web_credit_card_numb char (16),
  web_credit_card_exp_date date,
  web_sale_name_on_credit_card varchar (50),
  CONSTRAINT PK_web_customer_credit_card
      PRIMARY KEY (web_customer_numb,web_credit_card_numb),
  CONSTRAINT RelationWeb_customerweb_customer_credit_card1
      FOREIGN KEY () REFERENCES web_customer
);
```

■ **FIGURE 15.4** (***cont.***)

```
CREATE TABLE web_sale
(
  web_sale_numb integer,
  web_sale_date date,
  web_customer_numb integer,
  web_sale_total number (7,2),
  web_sale_merchandise_total number (6,2),
  web_sale_shipping number (6,2),
  web_sale_tax number (6,2),
  CONSTRAINT PK_web_sale PRIMARY KEY (web_sale_numb),
  CONSTRAINT RelationWeb_customerWeb_sale1 FOREIGN KEY ()
      REFERENCES web_customer,
  CONSTRAINT Relationweb_customer_credit_cardWeb_sale1
      FOREIGN KEY () REFERENCES web_customer_credit_card
);

CREATE TABLE web_sale_shipment
(
  web_sale_numb inter,
  web_sale_date_shipped date,
  web_shipment_merchandise_total number(7,2),
  web_shipment_tax number (6,2),
  web_shipment_shipping number (6,2),
  web_shipment_total_charged number (6,2),
  CONSTRAINT PK_web_sale_shipment PRIMARY KEY (web_sale_numb),
  CONSTRAINT RelationWeb_saleWeb_sale_shipment1
      FOREIGN KEY () REFERENCES web_sale
);

CREATE TABLE pay_type
(
  pay_type_code integer,
  pay_type_description varhar (10),
  CONSTRAINT PK_pay_type PRIMARY KEY (pay_type_code)
);
```

■ **FIGURE 15.4** (*cont.*)

```
CREATE TABLE employee
(
  employee_id integer,
  employee_first_name varchar (15),
  employee_last_name integer,
  employee_street varchar (50),
  employee_city varchar (50),
  employee_state_code char (2),
  employee_zip char (10),
  employee_phone char (12),
  employee_ssn char (11),
  employee_birth_date date,
  dept_id integer,
  store_numb integer,
  supervisor_id integer,
  pay_type_code integer,
  pay_rate number (10,2),
  CONSTRAINT PK_employee PRIMARY KEY (employee_id),
  CONSTRAINT RelationEmployeeDepartment1
      FOREIGN KEY () REFERENCES department,
  CONSTRAINT Relationpay_typeEmployee1
      FOREIGN KEY () REFERENCES pay_type,
  CONSTRAINT RelationEmployeestate1
      FOREIGN KEY () REFERENCES state
);

CREATE TABLE department
(
  dept_numb integer,
  store_numnb integer,
  dept_manager_id integer,
  CONSTRAINT PK_department PRIMARY KEY (dept_numb,store_numnb),
  CONSTRAINT RelationStoreDepartment1 FOREIGN KEY () REFERENCES Store
);

CREATE TABLE store
(
  store_numb integer,
  store_street char (50),
  store_city char (50),
  store_state_code char (2),
  store_zip char (10),
  store_main_phone char (12),
  store_manager_id integer,
  CONSTRAINT PK_Store PRIMARY KEY (store_numb)
);
```

■ **FIGURE 15.4** (***cont.***)

```
CREATE TABLE work_schedule
(
  employee_numb integer,
  work_date date,
  work_start_time time,
  work_hours_scheduled integer,
  CONSTRAINT PK_work_schedule PRIMARY KEY (employee_numb,work_date),
  CONSTRAINT RelationEmployeeWork_schedule2
      FOREIGN KEY () REFERENCES employee
);

CREATE TABLE credit_sale_details
(
  in_store_transaction_numb integer,
  in_store_sale_credit_card_numb char (16),
  in_store_exp_date date,
  in_store_sale_name_on_credit_card varchar (50),
  CONSTRAINT PK_credit_sale_details
      PRIMARY KEY (in_store_transaction_numb)
);

CREATE TABLE promotion_type
(
  promotion_type_code integer,
  promotion_type_name INTEGER,
  CONSTRAINT PK_Promotion_type PRIMARY KEY (promotion_type_code)
);

CREATE TABLE keyword_list
(
  keyword__code integer,
  keyword_text varchar (50),
  CONSTRAINT PK_keyword_list PRIMARY KEY (keyword__code)
);

CREATE TABLE product_keyword
(
  UPC char (15),
  keyword_code integer,
  CONSTRAINT PK_product_keyword PRIMARY KEY (UPC),
  CONSTRAINT RelationKeyword_listProduct_keyword1
      FOREIGN KEY () REFERENCES keyword_list
);
```

■ **FIGURE 15.4** (*cont.*)

```
CREATE TABLE sales_promotion
(
  UPC char (15),
  promotion_start_date date,
  promotion_end_date date,
  promotion_type_code integer,
  promotion_details char (50),
  CONSTRAINT PK_sales_promotion PRIMARY KEY (UPC,promotion_start_date),
  CONSTRAINT RelationProductSales_promotion1
      FOREIGN KEY () REFERENCES product
);

CREATE TABLE In_store_sale
(
  in_store_transaction_numb integer,
  in_store_transaction_date date,
  in_store_sale_product_total number (8,2),
  in_store_sales_tax number (6,2),
  in_store_sale_total number (8,2),
  in_store_sale_payment_type_code integer,
  CONSTRAINT PK_In_store_sale PRIMARY KEY (in_store_transaction_numb),
  CONSTRAINT RelationIn_store_salecredit_sale_details1
      FOREIGN KEY () REFERENCES credit_sale_details,
  CONSTRAINT RelationIn_store_payment_typeIn_store_sale1
      FOREIGN KEY () REFERENCES In_store_payment_type
);

CREATE TABLE product_stocked
(
  SKU char (15),
  UPC char (15),
  in_store boolean,
  store_ numb integer,
  dept_numb integer,
  warehouse_numb integer,
  size char (10),
  color char (15),
  number_on_hand integer,
  retail_price number (7,2),
  CONSTRAINT PK_product_stocked PRIMARY KEY (SKU),
  CONSTRAINT RelationProductProduct_stocked1
      FOREIGN KEY () REFERENCES product,
  CONSTRAINT RelationDepartmentProduct_stocked1
      FOREIGN KEY () REFERENCES department
);
```

■ **FIGURE 15.4** (***cont.***)

```
CREATE TABLE In_store_sale_item
(
  in_store_transaction_numb integer,
  in_store_SKU integer,
  in_store_quantity integer,
  in_store_price number (7,2)O,
  CONSTRAINT PK_In_store_sale_item
      PRIMARY KEY (in_store_transaction_numb,in_store_SKU),
  CONSTRAINT RelationProduct_stockedIn_store_sale_item1
      FOREIGN KEY () REFERENCES product_stocked,
  CONSTRAINT RelationIn_store_saleIn_store_sale_item1
      FOREIGN KEY () REFERENCES In_store_sale
);

CREATE TABLE web_sale_item
(
  web_sale_numb integer,
  web_sale_SKU char (15),
  web_sale_quantity integer,
  web_sale_shipped boolean,
  web_sale_price number (6,2),
  web_shipping_address_numb integer,
  CONSTRAINT PK_web_sale_item PRIMARY KEY (web_sale_numb,web_sale_SKU),
  CONSTRAINT RelationProduct_stockedWeb_sale_item1
      FOREIGN KEY () REFERENCES product_stocked,
  CONSTRAINT RelationWeb_saleWeb_sale_item1
      FOREIGN KEY () REFERENCES web_sale,
  CONSTRAINT RelationWeb_sale_shipmentWeb_sale_item1
      FOREIGN KEY () REFERENCES web_sale_shipment,
  CONSTRAINT RelationWeb_sale_shipping_addressWeb_sale_item1
      FOREIGN KEY () REFERENCES web_sale_shipping_address
);

CREATE TABLE web_security_question
(
  web_security_question_numb integer,
  web_security_question_text varchar (256),
  CONSTRAINT PK_web_security_question PRIMARY KEY
      (web_security_question_numb)
);

CREATE TABLE web_security_question_answer
(
  web_security_question_numb integer,
  web_customer_numb integer,
  web_security_question_answer_text varchar (256),
  CONSTRAINT PK_web_security_question_answer PRIMARY KEY
      (web_security_question_numb,web_customer_numb)
  CONSTRAINT RelationWeb_security_questionWeb_security_question_answer1
      FOREIGN KEY () REFERENCES web_security_question
  CONSTRAINT RelationWeb_customerWeb_security_question_answer1
      FOREIGN KEY () REFERENCES web_customer
);
```

■ **FIGURE 15.4** (***cont.***)

Page left intentionally blank