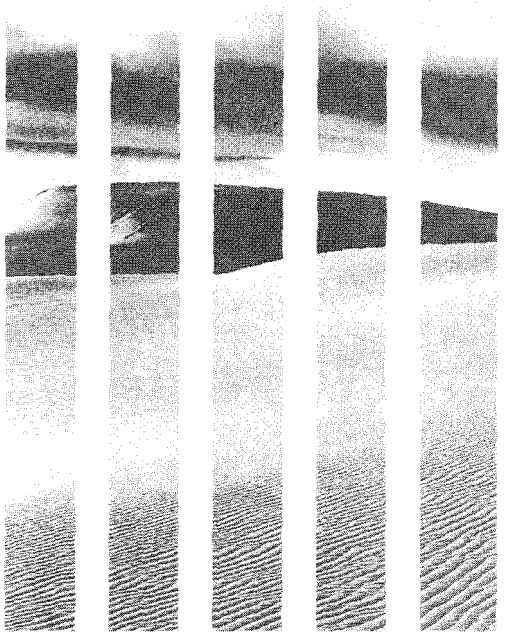

1

INTRODUCTION AND CONCEPTUAL MODELING

1

Databases and Database Users

Databases and database systems have become an essential component of everyday life in modern society. In the course of a day, most of us encounter several activities that involve some interaction with a database. For example, if we go to the bank to deposit or withdraw funds, if we make a hotel or airline reservation, if we access a computerized library catalog to search for a bibliographic item, or if we buy some item—such as a book, toy, or computer—from an Internet vendor through its Web page, chances are that our activities will involve someone or some computer program accessing a database. Even purchasing items from a supermarket nowadays in many cases involves an automatic update of the database that keeps the inventory of supermarket items.

These interactions are examples of what we may call **traditional database applications,** in which most of the information that is stored and accessed is either textual or numeric. In the past few years, advances in technology have been leading to exciting new applications of database systems. **Multimedia databases** can now store pictures, video clips, and sound messages. **Geographic information systems** (GIS) can store and analyze maps, weather data, and satellite images. **Data warehouses** and **online analytical processing** (OLAP) systems are used in many companies to extract and analyze useful information from very large databases for decision making. **Real-time** and **active database technology** is used in controlling industrial and manufacturing processes. And database search techniques are being applied to the World Wide Web to improve the search for information that is needed by users browsing the Internet.

To understand the fundamentals of database technology, however, we must start from the basics of traditional database applications. So, in Section 1.1 of this chapter we define what a database is, and then we give definitions of other basic terms. In Section 1.2, we provide a simple UNIVERSITY database example to illustrate our discussion. Section 1.3 describes some of the main characteristics of database systems, and Sections 1.4 and 1.5 categorize the types of personnel whose jobs involve using and interacting with database systems. Sections 1.6, 1.7, and 1.8 offer a more thorough discussion of the various capabilities provided by database systems and discuss some typical database applications. Section 1.9 summarizes the chapter.

The reader who desires only a quick introduction to database systems can study Sections 1.1 through 1.5, then skip or browse through Sections 1.6 through 1.8 and go on to Chapter 2.

1.1 INTRODUCTION

Databases and database technology are having a major impact on the growing use of computers. It is fair to say that databases play a critical role in almost all areas where computers are used, including business, electronic commerce, engineering, medicine, law, education, and library science, to name a few. The word *database* is in such common use that we must begin by defining what a database is. Our initial definition is quite general.

A **database** is a collection of related data.[1] By **data,** we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. You may have recorded this data in an indexed address book, or you may have stored it on a hard drive, using a personal computer and software such as Microsoft Access, or Excel. This is a collection of related data with an implicit meaning and hence is a database.

The preceding definition of database is quite general; for example, we may consider the collection of words that make up this page of text to be related data and hence to constitute a database. However, the common use of the term *database* is usually more restricted. A database has the following implicit properties:

- A database represents some aspect of the real world, sometimes called the **miniworld** or the **universe of discourse (UoD).** Changes to the miniworld are reflected in the database.
- A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.
- A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

1. We will use the word *data* as both singular and plural, as is common in database literature; context will determine whether it is singular or plural. In standard English, *data* is used only for plural; *datum* is used for singular.

In other words, a database has some source from which data is derived, some degree of interaction with events in the real world, and an audience that is actively interested in the contents of the database.

A database can be of any size and of varying complexity. For example, the list of names and addresses referred to earlier may consist of only a few hundred records, each with a simple structure. On the other hand, the computerized catalog of a large library may contain half a million entries organized under different categories—by primary author's last name, by subject, by book title—with each category organized in alphabetic order. A database of even greater size and complexity is maintained by the Internal Revenue Service to keep track of the tax forms filed by U.S. taxpayers. If we assume that there are 100 million taxpayers and if each taxpayer files an average of five forms with approximately 400 characters of information per form, we would get a database of $100 \times 10^6 \times 400 \times 5$ characters (bytes) of information. If the IRS keeps the past three returns for each taxpayer in addition to the current return, we would get a database of 8×10^{11} bytes (800 gigabytes). This huge amount of information must be organized and managed so that users can search for, retrieve, and update the data as needed.

A database may be generated and maintained manually or it may be computerized. For example, a library card catalog is a database that may be created and maintained manually. A computerized database may be created and maintained either by a group of application programs written specifically for that task or by a database management system. Of course, we are only concerned with computerized databases in this book.

A **database management system (DBMS)** is a collection of programs that enables users to create and maintain a database. The DBMS is hence a *general-purpose software system* that facilitates the processes of *defining*, *constructing*, *manipulating*, and *sharing* databases among various users and applications. **Defining** a database involves specifying the data types, structures, and constraints for the data to be stored in the database. **Constructing** the database is the process of storing the data itself on some storage medium that is controlled by the DBMS. **Manipulating** a database includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data. **Sharing** a database allows multiple users and programs to access the database concurrently.

Other important functions provided by the DBMS include *protecting* the database and *maintaining* it over a long period of time. **Protection** includes both *system protection* against hardware or software malfunction (or crashes), and *security protection* against unauthorized or malicious access. A typical large database may have a life cycle of many years, so the DBMS must be able to **maintain** the database system by allowing the system to evolve as requirements change over time.

It is not necessary to use general-purpose DBMS software to implement a computerized database. We could write our own set of programs to create and maintain the database, in effect creating our own *special-purpose* DBMS software. In either case—whether we use a general-purpose DBMS or not—we usually have to deploy a considerable amount of complex software. In fact, most DBMSs are very complex software systems.

To complete our initial definitions, we will call the database and DBMS software together a **database system.** Figure 1.1 illustrates some of the concepts we discussed so far.

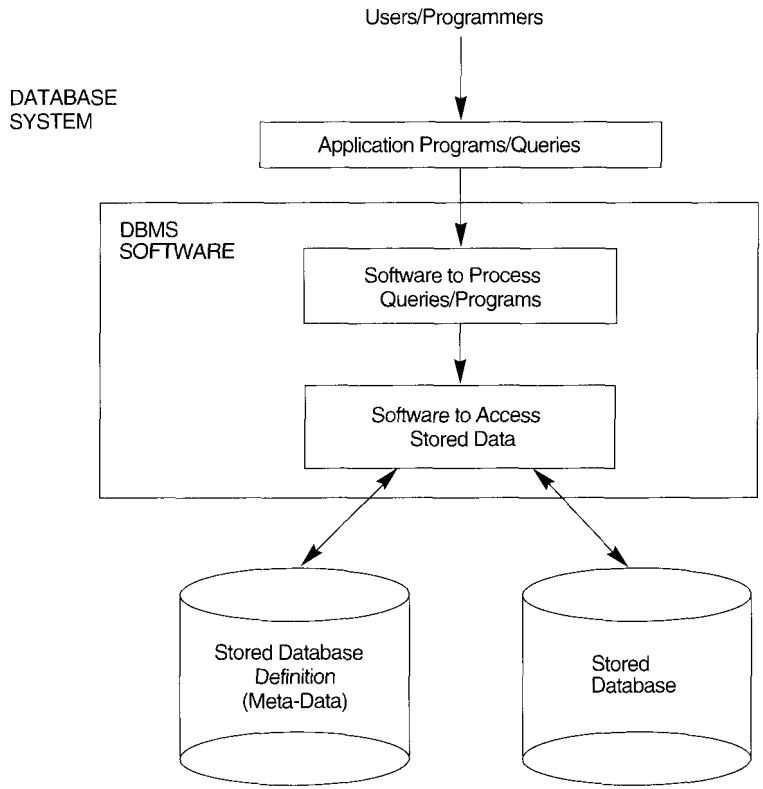

FIGURE 1.1 A simplified database system environment.

1.2 An Example

Let us consider a simple example that most readers may be familiar with: a UNIVERSITY database for maintaining information concerning students, courses, and grades in a university environment. Figure 1.2 shows the database structure and a few sample data for such a database. The database is organized as five files, each of which stores data records of the same type.[2] The STUDENT file stores data on each student, the COURSE file stores data on each course, the SECTION file stores data on each section of a course, the GRADE_REPORT file stores the grades that students receive in the various sections they have completed, and the PREREQUISITE file stores the prerequisites of each course.

To *define* this database, we must specify the structure of the records of each file by specifying the different types of **data elements** to be stored in each record. In Figure 1.2, each STUDENT record includes data to represent the student's Name, StudentNumber, Class

2. We use the term *file* informally here. At a conceptual level, a *file* is a *collection* of records that may or may not be ordered.

STUDENT	Name	StudentNumber	Class	Major
	Smith	17	1	CS
	Brown	8	2	CS

COURSE	CourseName	CourseNumber	CreditHours	Department
	Intro to Computer Science	CS1310	4	CS
	Data Structures	CS3320	4	CS
	Discrete Mathematics	MATH2410	3	MATH
	Database	CS3380	3	CS

SECTION	SectionIdentifier	CourseNumber	Semester	Year	Instructor
	85	MATH2410	Fall	98	King
	92	CS1310	Fall	98	Anderson
	102	CS3320	Spring	99	Knuth
	112	MATH2410	Fall	99	Chang
	119	CS1310	Fall	99	Anderson
	135	CS3380	Fall	99	Stone

GRADE_REPORT	StudentNumber	SectionIdentifier	Grade
	17	112	B
	17	119	C
	8	85	A
	8	92	A
	8	102	B
	8	135	A

PREREQUISITE	CourseNumber	PrerequisiteNumber
	CS3380	CS3320
	CS3380	MATH2410
	CS3320	CS1310

FIGURE 1.2 A database that stores student and course information.

(freshman or 1, sophomore or 2, . . .), and Major (mathematics or math, computer science or CS, . . .); each COURSE record includes data to represent the CourseName, CourseNumber, CreditHours, and Department (the department that offers the course); and so on. We must also specify a **data type** for each data element within a record. For example, we can specify that Name of STUDENT is a string of alphabetic characters, StudentNumber of STUDENT is an integer, and Grade of GRADE_REPORT is a single character from the set {A, B, C, D, F, I}. We may also use a coding scheme to represent the values of

a data item. For example, in Figure 1.2 we represent the Class of a STUDENT as 1 for freshman, 2 for sophomore, 3 for junior, 4 for senior, and 5 for graduate student.

To *construct* the UNIVERSITY database, we store data to represent each student, course, section, grade report, and prerequisite as a record in the appropriate file. Notice that records in the various files may be related. For example, the record for "Smith" in the STUDENT file is related to two records in the GRADE_REPORT file that specify Smith's grades in two sections. Similarly, each record in the PREREQUISITE file relates two course records: one representing the course and the other representing the prerequisite. Most medium-size and large databases include many types of records and have *many relationships* among the records.

Database *manipulation* involves querying and updating. Examples of queries are "retrieve the transcript—a list of all courses and grades—of Smith," "list the names of students who took the section of the Database course offered in fall 1999 and their grades in that section," and "what are the prerequisites of the Database course?" Examples of updates are "change the class of Smith to Sophomore," "create a new section for the Database course for this semester," and "enter a grade of A for Smith in the Database section of last semester." These informal queries and updates must be specified precisely in the query language of the DBMS before they can be processed.

1.3 Characteristics of the Database Approach

A number of characteristics distinguish the database approach from the traditional approach of programming with files. In traditional **file processing,** each user defines and implements the files needed for a specific software application as part of programming the application. For example, one user, the *grade reporting office*, may keep a file on students and their grades. Programs to print a student's transcript and to enter new grades into the file are implemented as part of the application. A second user, the *accounting office*, may keep track of students' fees and their payments. Although both users are interested in data about students, each user maintains separate files—and programs to manipulate these files—because each requires some data not available from the other user's files. This redundancy in defining and storing data results in wasted storage space and in redundant efforts to maintain common data up to date.

In the database approach, a single repository of data is maintained that is defined once and then is accessed by various users. The main characteristics of the database approach versus the file-processing approach are the following:

- Self-describing nature of a database system
- Insulation between programs and data, and data abstraction
- Support of multiple views of the data
- Sharing of data and multiuser transaction procesing

We next describe each of these characteristics in a separate section. Additional characteristics of database systems are discussed in Sections 1.6 through 1.8.

1.3.1 Self-Describing Nature of a Database System

A fundamental characteristic of the database approach is that the database system contains not only the database itself but also a complete definition or description of the database structure and constraints. This definition is stored in the DBMS **catalog,** which contains information such as the structure of each file, the type and storage format of each data item, and various constraints on the data. The information stored in the catalog is called **meta-data,** and it describes the structure of the primary database (Figure 1.1).

The catalog is used by the DBMS software and also by database users who need information about the database structure. A general-purpose DBMS software package is not written for a specific database application, and hence it must refer to the catalog to know the structure of the files in a specific database, such as the type and format of data it will access. The DBMS software must work equally well with *any number of database applications*—for example, a university database, a banking database, or a company database—as long as the database definition is stored in the catalog.

In traditional file processing, data definition is typically part of the application programs themselves. Hence, these programs are constrained to work with only *one specific database,* whose structure is declared in the application programs. For example, an application program written in C++ may have struct or class declarations, and a COBOL program has Data Division statements to define its files. Whereas file-processing software can access only specific databases, DBMS software can access diverse databases by extracting the database definitions from the catalog and then using these definitions.

In the example shown in Figure 1.2, the DBMS catalog will store the definitions of all the files shown. These definitions are specified by the database designer prior to creating the actual database and are stored in the catalog. Whenever a request is made to access, say, the Name of a STUDENT record, the DBMS software refers to the catalog to determine the structure of the STUDENT file and the position and size of the Name data item within a STUDENT record. By contrast, in a typical file-processing application, the file structure and, in the extreme case, the exact location of Name within a STUDENT record are already coded within each program that accesses this data item.

1.3.2 Insulation between Programs and Data, and Data Abstraction

In traditional file processing, the structure of data files is embedded in the application programs, so any changes to the structure of a file may require *changing all programs* that access this file. By contrast, DBMS access programs do not require such changes in most cases. The structure of data files is stored in the DBMS catalog separately from the access programs. We call this property **program-data independence.** For example, a file access program may be written in such a way that it can access only STUDENT records of the structure shown in Figure 1.3. If we want to add another piece of data to each STUDENT record, say the BirthDate, such a program will no longer work and must be changed. By contrast, in a DBMS environment, we just need to change the description of STUDENT records in the catalog to reflect the inclusion of the new data item BirthDate; no programs are changed. The next time a DBMS program refers to the catalog, the new structure of STUDENT records will be accessed and used.

Data Item Name	Starting Position in Record	Length in Characters (bytes)
Name	1	30
StudentNumber	31	4
Class	35	4
Major	39	4

FIGURE 1.3 Internal storage format for a STUDENT record.

In some types of database systems, such as object-oriented and object-relational systems (see Chapters 20 to 22), users can define operations on data as part of the database definitions. An **operation** (also called a *function* or *method*) is specified in two parts. The *interface* (or *signature*) of an operation includes the operation name and the data types of its arguments (or parameters). The *implementation* (or *method*) of the operation is specified separately and can be changed without affecting the interface. User application programs can operate on the data by invoking these operations through their names and arguments, regardless of how the operations are implemented. This may be termed **program-operation independence.**

The characteristic that allows program-data independence and program-operation independence is called **data abstraction.** A DBMS provides users with a **conceptual representation** of data that does not include many of the details of how the data is stored or how the operations are implemented. Informally, a **data model** is a type of data abstraction that is used to provide this conceptual representation. The data model uses logical concepts, such as objects, their properties, and their interrelationships, that may be easier for most users to understand than computer storage concepts. Hence, the data model *hides* storage and implementation details that are not of interest to most database users.

For example, consider again Figure 1.2. The internal implementation of a file may be defined by its record length—the number of characters (bytes) in each record—and each data item may be specified by its starting byte within a record and its length in bytes. The STUDENT record would thus be represented as shown in Figure 1.3. But a typical database user is not concerned with the location of each data item within a record or its length; rather, the concern is that when a reference is made to Name of STUDENT, the correct value is returned. A conceptual representation of the STUDENT records is shown in Figure 1.2. Many other details of file storage organization—such as the access paths specified on a file—can be hidden from database users by the DBMS; we discuss storage details in Chapters 13 and 14.

In the database approach, the detailed structure and organization of each file are stored in the catalog. Database users and application programs refer to the conceptual representation of the files, and the DBMS extracts the details of file storage from the catalog when these are needed by the DBMS file access modules. Many data models can be used to provide this data abstraction to database users. A major part of this book is devoted to presenting various data models and the concepts they use to abstract the representation of data.

In object-oriented and object-relational databases, the abstraction process includes not only the data structure but also the operations on the data. These operations provide an abstraction of miniworld activities commonly understood by the users. For example,

(a)

TRANSCRIPT	StudentName	Student Transcript				
		CourseNumber	Grade	Semester	Year	SectionId
	Smith	CS1310	C	Fall	99	119
		MATH2410	B	Fall	99	112
	Brown	MATH2410	A	Fall	98	85
		CS1310	A	Fall	98	92
		CS3320	B	Spring	99	102
		CS3380	A	Fall	99	135

(b)

PREREQUISITES	CourseName	CourseNumber	Prerequisites
	Database	CS3380	CS3320
			MATH2410
	Data Structures	CS3320	CS1310

FIGURE 1.4 Two views derived from the database in Figure 1.2. (a) The STUDENT TRANSCRIPT view. (b) The COURSE PREREQUISITES view.

an operation CALCULATE_GPA can be applied to a STUDENT object to calculate the grade point average. Such operations can be invoked by the user queries or application programs without having to know the details of how the operations are implemented. In that sense, an abstraction of the miniworld activity is made available to the user as an **abstract operation.**

1.3.3 Support of Multiple Views of the Data

A database typically has many users, each of whom may require a different perspective or **view** of the database. A view may be a subset of the database or it may contain **virtual data** that is derived from the database files but is not explicitly stored. Some users may not need to be aware of whether the data they refer to is stored or derived. A multiuser DBMS whose users have a variety of distinct applications must provide facilities for defining multiple views. For example, one user of the database of Figure 1.2 may be interested only in accessing and printing the transcript of each student; the view for this user is shown in Figure 1.4a. A second user, who is interested only in checking that students have taken all the prerequisites of each course for which they register, may require the view shown in Figure 1.4b.

1.3.4 Sharing of Data and Multiuser Transaction Processing

A multiuser DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and

maintained in a single database. The DBMS must include **concurrency control** software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct. For example, when several reservation clerks try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one clerk at a time for assignment to a passenger. These types of applications are generally called **online transaction processing (OLTP)** applications. A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly.

The concept of a **transaction** has become central to many database applications. A transaction is an *executing program* or *process* that includes one or more database accesses, such as reading or updating of database records. Each transaction is supposed to execute a logically correct database access if executed in its entirety without interference from other transactions. The DBMS must enforce several transaction properties. The **isolation** property ensures that each transaction appears to execute in isolation from other transactions, even though hundreds of transactions may be executing concurrently. The **atomicity** property ensures that either all the database operations in a transaction are executed or none are. We discuss transactions in detail in Part V of the textbook.

The preceding characteristics are most important in distinguishing a DBMS from traditional file-processing software. In Section 1.6 we discuss additional features that characterize a DBMS. First, however, we categorize the different types of persons who work in a database system environment.

1.4 Actors on the Scene

For a small personal database, such as the list of addresses discussed in Section 1.1, one person typically defines, constructs, and manipulates the database, and there is no sharing. However, many persons are involved in the design, use, and maintenance of a large database with hundreds of users. In this section we identify the people whose jobs involve the day-to-day use of a large database; we call them the "actors on the scene." In Section 1.5 we consider people who may be called "workers behind the scene"—those who work to maintain the database system environment but who are not actively interested in the database itself.

1.4.1 Database Administrators

In any organization where many persons use the same resources, there is a need for a chief administrator to oversee and manage these resources. In a database environment, the primary resource is the database itself, and the secondary resource is the DBMS and related software. Administering these resources is the responsibility of the **database administrator (DBA)**. The DBA is responsible for authorizing access to the database, for coordinating and monitoring its use, and for acquiring software and hardware resources as needed. The DBA is accountable for problems such as breach of security or poor system response time. In large organizations, the DBA is assisted by a staff that helps carry out these functions.

1.4.2 Database Designers

Database designers are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data. These tasks are mostly undertaken before the database is actually implemented and populated with data. It is the responsibility of database designers to communicate with all prospective database users in order to understand their requirements, and to come up with a design that meets these requirements. In many cases, the designers are on the staff of the DBA and may be assigned other staff responsibilities after the database design is completed. Database designers typically interact with each potential group of users and develop **views** of the database that meet the data and processing requirements of these groups. Each view is then analyzed and *integrated* with the views of other user groups. The final database design must be capable of supporting the requirements of all user groups.

1.4.3 End Users

End users are the people whose jobs require access to the database for querying, updating, and generating reports; the database primarily exists for their use. There are several categories of end users:

- **Casual end users** occasionally access the database, but they may need different information each time. They use a sophisticated database query language to specify their requests and are typically middle- or high-level managers or other occasional browsers.
- **Naive** or **parametric end users** make up a sizable portion of database end users. Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates—called **canned transactions**—that have been carefully programmed and tested. The tasks that such users perform are varied:

  Bank tellers check account balances and post withdrawals and deposits.

  Reservation clerks for airlines, hotels, and car rental companies check availability for a given request and make reservations.

  Clerks at receiving stations for courier mail enter package identifications via bar codes and descriptive information through buttons to update a central database of received and in-transit packages.
- **Sophisticated end users** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS so as to implement their applications to meet their complex requirements.
- **Stand-alone users** maintain personal databases by using ready-made program packages that provide easy-to-use menu-based or graphics-based interfaces. An example is the user of a tax package that stores a variety of personal financial data for tax purposes.

A typical DBMS provides multiple facilities to access a database. Naive end users need to learn very little about the facilities provided by the DBMS; they have to understand only the user interfaces of the standard transactions designed and implemented for their

use. Casual users learn only a few facilities that they may use repeatedly. Sophisticated users try to learn most of the DBMS facilities in order to achieve their complex requirements. Stand-alone users typically become very proficient in using a specific software package.

1.4.4 System Analysts and Application Programmers (Software Engineers)

System analysts determine the requirements of end users, especially naive and parametric end users, and develop specifications for canned transactions that meet these requirements. **Application programmers** implement these specifications as programs; then they test, debug, document, and maintain these canned transactions. Such analysts and programmers—commonly referred to as **software engineers**—should be familiar with the full range of capabilities provided by the DBMS to accomplish their tasks.

1.5 Workers behind the Scene

In addition to those who design, use, and administer a database, others are associated with the design, development, and operation of the DBMS *software and system environment*. These persons are typically not interested in the database itself. We call them the "workers behind the scene," and they include the following categories.

- DBMS **system designers and implementers** are persons who design and implement the DBMS modules and interfaces as a software package. A DBMS is a very complex software system that consists of many components, or **modules,** including modules for implementing the catalog, processing query language, processing the interface, accessing and buffering data, controlling concurrency, and handling data recovery and security. The DBMS must interface with other system software, such as the operating system and compilers for various programming languages.
- **Tool developers** include persons who design and implement **tools**—the software packages that facilitate database system design and use and that help improve performance. Tools are optional packages that are often purchased separately. They include packages for database design, performance monitoring, natural language or graphical interfaces, prototyping, simulation, and test data generation. In many cases, independent software vendors develop and market these tools.
- **Operators and maintenance personnel** are the system administration personnel who are responsible for the actual running and maintenance of the hardware and software environment for the database system.

Although these categories of workers behind the scene are instrumental in making the database system available to end users, they typically do not use the database for their own purposes.

1.6 Advantages of Using the DBMS Approach

In this section we discuss some of the advantages of using a DBMS and the capabilities that a good DBMS should possess. These capabilities are in addition to the four main characteristics discussed in Section 1.3. The DBA must utilize these capabilities to accomplish a variety of objectives related to the design, administration, and use of a large multiuser database.

1.6.1 Controlling Redundancy

In traditional software development utilizing file processing, every user group maintains its own files for handling its data-processing applications. For example, consider the UNIVERSITY database example of Section 1.2; here, two groups of users might be the course registration personnel and the accounting office. In the traditional approach, each group independently keeps files on students. The accounting office also keeps data on registration and related billing information, whereas the registration office keeps track of student courses and grades. Much of the data is stored twice: once in the files of each user group. Additional user groups may further duplicate some or all of the same data in their own files.

This **redundancy** in storing the same data multiple times leads to several problems. First, there is the need to perform a single logical update—such as entering data on a new student—multiple times: once for each file where student data is recorded. This leads to *duplication of effort*. Second, *storage space is wasted* when the same data is stored repeatedly, and this problem may be serious for large databases. Third, files that represent the same data may become *inconsistent*. This may happen because an update is applied to some of the files but not to others. Even if an update—such as adding a new student—is applied to all the appropriate files, the data concerning the student may still be *inconsistent* because the updates are applied independently by each user group. For example, one user group may enter a student's birthdate erroneously as JAN-19-1984, whereas the other user groups may enter the correct value of JAN-29-1984.

In the database approach, the views of different user groups are integrated during database design. Ideally, we should have a database design that stores each logical data item—such as a student's name or birth date—in *only one place* in the database. This ensures consistency, and it saves storage space. However, in practice, it is sometimes necessary to use **controlled redundancy** for improving the performance of queries. For example, we may store StudentName and CourseNumber redundantly in a GRADE_REPORT file (Figure 1.5a) because whenever we retrieve a GRADE_REPORT record, we want to retrieve the student name and course number along with the grade, student number, and section identifier. By placing all the data together, we do not have to search multiple files to collect this data. In such cases, the DBMS should have the capability to *control* this redundancy so as to prohibit inconsistencies among the files. This may be done by automatically checking that the StudentName-StudentNumber values in any GRADE_REPORT record in Figure 1.5a match one of the Name-StudentNumber values of a STUDENT record (Figure 1.2). Similarly, the SectionIdentifier-CourseNumber values in

(a)

GRADE_REPORT	StudentNumber	StudentName	SectionIdentifier	CourseNumber	Grade
	17	Smith	112	MATH2410	B
	17	Smith	119	CS1310	C
	8	Brown	85	MATH2410	A
	8	Brown	92	CS1310	A
	8	Brown	102	CS3320	B
	8	Brown	135	CS3380	A

(b)

GRADE_REPORT	StudentNumber	StudentName	SectionIdentifier	CourseNumber	Grade
	17	Brown	112	MATH2410	B

FIGURE 1.5 Redundant storage of StudentName and CourseNumber in GRADE_REPORT. (a) Consistent data. (b) Inconsistent record.

GRADE_REPORT can be checked against SECTION records. Such checks can be specified to the DBMS during database design and automatically enforced by the DBMS whenever the GRADE_REPORT file is updated. Figure 1.5b shows a GRADE_REPORT record that is inconsistent with the STUDENT file of Figure 1.2, which may be entered erroneously if the redundancy is *not controlled*.

1.6.2 Restricting Unauthorized Access

When multiple users share a large database, it is likely that most users will not be authorized to access all information in the database. For example, financial data is often considered confidential, and hence only authorized persons are allowed to access such data. In addition, some users may be permitted only to retrieve data, whereas others are allowed both to retrieve and to update. Hence, the type of access operation—retrieval or update—must also be controlled. Typically, users or user groups are given account numbers protected by passwords, which they can use to gain access to the database. A DBMS should provide a **security and authorization subsystem,** which the DBA uses to create accounts and to specify account restrictions. The DBMS should then enforce these restrictions automatically. Notice that we can apply similar controls to the DBMS software. For example, only the DBA's staff may be allowed to use certain **privileged software,** such as the software for creating new accounts. Similarly, parametric users may be allowed to access the database only through the canned transactions developed for their use.

1.6.3 Providing Persistent Storage for Program Objects

Databases can be used to provide **persistent storage** for program objects and data structures. This is one of the main reasons for **object-oriented database systems.** Programming languages typically have complex data structures, such as record types in Pascal or class

definitions in C++ or Java. The values of program variables are discarded once a program terminates, unless the programmer explicitly stores them in permanent files, which often involves converting these complex structures into a format suitable for file storage. When the need arises to read this data once more, the programmer must convert from the file format to the program variable structure. Object-oriented database systems are compatible with programming languages such as C++ and Java, and the DBMS software automatically performs any necessary conversions. Hence, a complex object in C++ can be stored permanently in an object-oriented DBMS. Such an object is said to be **persistent,** since it survives the termination of program execution and can later be directly retrieved by another C++ program.

The persistent storage of program objects and data structures is an important function of database systems. Traditional database systems often suffered from the so-called **impedance mismatch problem,** since the data structures provided by the DBMS were incompatible with the programming language's data structures. Object-oriented database systems typically offer data structure **compatibility** with one or more object-oriented programming languages.

1.6.4 Providing Storage Structures for Efficient Query Processing

Database systems must provide capabilities for *efficiently executing queries and updates*. Because the database is typically stored on disk, the DBMS must provide specialized data structures to speed up disk search for the desired records. Auxiliary files called **indexes** are used for this purpose. Indexes are typically based on tree data structures or hash data structures, suitably modified for disk search. In order to process the database records needed by a particular query, those records must be copied from disk to memory. Hence, the DBMS often has a **buffering** module that maintains parts of the database in main memory buffers. In other cases, the DBMS may use the operating system to do the buffering of disk data.

The **query processing and optimization** module of the DBMS is responsible for choosing an efficient query execution plan for each query based on the existing storage structures. The choice of which indexes to create and maintain is part of *physical database design and tuning*, which is one of the responsibilities of the DBA staff.

1.6.5 Providing Backup and Recovery

A DBMS must provide facilities for recovering from hardware or software failures. The **backup and recovery subsystem** of the DBMS is responsible for recovery. For example, if the computer system fails in the middle of a complex update transaction, the recovery subsystem is responsible for making sure that the database is restored to the state it was in before the transaction started executing. Alternatively, the recovery subsystem could ensure that the transaction is resumed from the point at which it was interrupted so that its full effect is recorded in the database.

1.6.6 Providing Multiple User Interfaces

Because many types of users with varying levels of technical knowledge use a database, a DBMS should provide a variety of user interfaces. These include query languages for casual users, programming language interfaces for application programmers, forms and command codes for parametric users, and menu-driven interfaces and natural language interfaces for stand-alone users. Both forms-style interfaces and menu-driven interfaces are commonly known as **graphical user interfaces (GUIs).** Many specialized languages and environments exist for specifying GUIs. Capabilities for providing Web GUI interfaces to a database—or Web-enabling a database—are also quite common.

1.6.7 Representing Complex Relationships among Data

A database may include numerous varieties of data that are interrelated in many ways. Consider the example shown in Figure 1.2. The record for Brown in the STUDENT file is related to four records in the GRADE_REPORT file. Similarly, each section record is related to one course record as well as to a number of GRADE_REPORT records—one for each student who completed that section. A DBMS must have the capability to represent a variety of complex relationships among the data as well as to retrieve and update related data easily and efficiently.

1.6.8 Enforcing Integrity Constraints

Most database applications have certain **integrity constraints** that must hold for the data. A DBMS should provide capabilities for defining and enforcing these constraints. The simplest type of integrity constraint involves specifying a data type for each data item. For example, in Figure 1.2, we may specify that the value of the Class data item within each STUDENT record must be an integer between 1 and 5 and that the value of Name must be a string of no more than 30 alphabetic characters. A more complex type of constraint that frequently occurs involves specifying that a record in one file must be related to records in other files. For example, in Figure 1.2, we can specify that "every section record must be related to a course record." Another type of constraint specifies uniqueness on data item values, such as "every course record must have a unique value for CourseNumber." These constraints are derived from the meaning or **semantics** of the data and of the miniworld it represents. It is the database designers' responsibility to identify integrity constraints during database design. Some constraints can be specified to the DBMS and automatically enforced. Other constraints may have to be checked by update programs or at the time of data entry.

A data item may be entered erroneously and still satisfy the specified integrity constraints. For example, if a student receives a grade of A but a grade of C is entered in the database, the DBMS *cannot* discover this error automatically, because C is a valid value for the Grade data type. Such data entry errors can only be discovered manually (when the student receives the grade and complains) and corrected later by updating the database. However, a grade of Z can be rejected automatically by the DBMS, because Z is not a valid value for the Grade data type.

1.6.9 Permitting Inferencing and Actions Using Rules

Some database systems provide capabilities for defining *deduction rules* for *inferencing* new information from the stored database facts. Such systems are called **deductive database systems.** For example, there may be complex rules in the miniworld application for determining when a student is on probation. These can be specified *declaratively* as **rules,** which when compiled and maintained by the DBMS can determine all students on probation. In a traditional DBMS, an explicit *procedural program code* would have to be written to support such applications. But if the miniworld rules change, it is generally more convenient to change the declared deduction rules than to recode procedural programs. More powerful functionality is provided by **active database systems,** which provide active rules that can automatically initiate actions when certain events and conditions occur.

1.6.10 Additional Implications of Using the Database Approach

This section discusses some additional implications of using the database approach that can benefit most organizations.

Potential for Enforcing Standards. The database approach permits the DBA to define and enforce standards among database users in a large organization. This facilitates communication and cooperation among various departments, projects, and users within the organization. Standards can be defined for names and formats of data elements, display formats, report structures, terminology, and so on. The DBA can enforce standards in a centralized database environment more easily than in an environment where each user group has control of its own files and software.

Reduced Application Development Time. A prime selling feature of the database approach is that developing a new application—such as the retrieval of certain data from the database for printing a new report—takes very little time. Designing and implementing a new database from scratch may take more time than writing a single specialized file application. However, once a database is up and running, substantially less time is generally required to create new applications using DBMS facilities. Development time using a DBMS is estimated to be one-sixth to one-fourth of that for a traditional file system.

Flexibility. It may be necessary to change the structure of a database as requirements change. For example, a new user group may emerge that needs information not currently in the database. In response, it may be necessary to add a file to the database or to extend the data elements in an existing file. Modern DBMSs allow certain types of evolutionary changes to the structure of the database without affecting the stored data and the existing application programs.

Availability of Up-to-Date Information. A DBMS makes the database available to all users. As soon as one user's update is applied to the database, all other users can

immediately see this update. This availability of up-to-date information is essential for many transaction-processing applications, such as reservation systems or banking databases, and it is made possible by the concurrency control and recovery subsystems of a DBMS.

Economies of Scale. The DBMS approach permits consolidation of data and applications, thus reducing the amount of wasteful overlap between activities of data-processing personnel in different projects or departments. This enables the whole organization to invest in more powerful processors, storage devices, or communication gear, rather than having each department purchase its own (weaker) equipment. This reduces overall costs of operation and management.

1.7 A Brief History of Database Applications

We now give a brief historical overview of the applications that use DBMSs, and how these applications provided the impetus for new types of database systems.

1.7.1 Early Database Applications Using Hierarchical and Network Systems

Many early database applications maintained records in large organzations, such as corporations, universities, hospitals, and banks. In many of these applications, there were large numbers of records of similar structure. For example, in a university application, similar information would be kept for each student, each course, each grade record, and so on. There were also many types of records and many interrelationships among them.

One of the main problems with early database systems was the intermixing of conceptual relationships with the physical storage and placement of records on disk. For example, the grade records of a particular student could be physically stored next to the student record. Although this provided very efficient access for the original queries and transactions that the database was designed to handle, it did not provide enough flexibility to access records efficiently when new queries and transactions were identified. In particular, new queries that required a different storage organization for efficient processing were quite difficult to implement efficiently. It was also quite difficult to reorganize the database when changes were made to the requirements of the application.

Another shortcoming of early systems was that they provided only programming language interfaces. This made it time-consuming and expensive to implement new queries and transactions, since new programs had to be written, tested, and debugged. Most of these database systems were implemented on large and expensive mainframe computers starting in the mid-1960s and through the 1970s and 1980s. The main types of early systems were based on three main paradigms: hierarchical systems, network model based systems, and inverted file systems.

1.7.2 Providing Application Flexibility with Relational Databases

Relational databases were originally proposed to separate the physical storage of data from its conceptual representation and to provide a mathematical foundation for databases. The relational data model also introduced high-level query languages that provided an alternative to programming language interfaces; hence, it was a lot quicker to write new queries. Relational representation of data somewhat resembles the example we presented in Figure 1.2. Relational systems were initially targeted to the same applications as earlier systems, but were meant to provide flexibility to quickly develop new queries and to reorganize the database as requirements changed.

Early experimental relational systems developed in the late 1970s and the commercial RDBMSs (relational database management systems) introduced in the early 1980s were quite slow, since they did not use physical storage pointers or record placement to access related data records. With the development of new storage and indexing techniques and better query processing and optimization, their performance improved. Eventually, relational databases became the dominant type of database systems for traditional database applications. Relational databases now exist on almost all types of computers, from small personal computers to large servers.

1.7.3 Object-Oriented Applications and the Need for More Complex Databases

The emergence of object-oriented programming languages in the 1980s and the need to store and share complex-structured objects led to the development of object-oriented databases. Initially, they were considered a competitor to relational databases, since they provided more general data structures. They also incorporated many of the useful object-oriented paradigms, such as abstract data types, encapsulation of operations, inheritance, and object identity. However, the complexity of the model and the lack of an early standard contributed to their limited use. They are now mainly used in specialized applications, such as engineering design, multimedia publishing, and manufacturing systems.

1.7.4 Interchanging Data on the Web for E-Commerce

The World Wide Web provided a large network of interconnected computers. Users can create documents using a Web publishing language, such as HTML (HyperText Markup Language), and store these documents on Web servers where other users (clients) can access them. Documents can be linked together through **hyperlinks,** which are pointers to other documents. In the 1990s, electronic commerce (e-commerce) emerged as a major application on the Web. It quickly became apparent that parts of the information on e-commerce Web pages were often dynamically extracted data from DBMSs. A variety of techniques were developed to allow the interchange of data on the

Web. Currently, XML (eXtended Markup Language) is considered to be the primary standard for interchanging data among various types of databases and Web pages. XML combines concepts from the models used in document systems with database modeling concepts.

1.7.5 Extending Database Capabilities for New Applications

The success of database systems in traditional applications encouraged developers of other types of applications to attempt to use them. Such applications traditionally used their own specialized file and data structures. The following are examples of these applications:

- **Scientific** applications that store large amounts of data resulting from scientific experiments in areas such as high-energy physics or the mapping of the human genome.
- Storage and retrieval of **images,** from scanned news or personal photographs to satellite photograph images and images from medical procedures such as X-rays or MRI (magnetic resonance imaging).
- Storage and retrieval of **videos,** such as movies, or **video clips** from news or personal digital cameras.
- **Data mining** applications that analyze large amounts of data searching for the occurrences of specific patterns or relationships.
- **Spatial** applications that store spatial locations of data such as weather information or maps used in geographical information systems.
- **Time series** applications that store information such as economic data at regular points in time, for example, daily sales or monthly gross national product figures.

It was quickly apparent that basic relational systems were not very suitable for many of these applications, usually for one or more of the following reasons:

- More complex data structures were needed for modeling the application than the simple relational representation.
- New data types were needed in addition to the basic numeric and character string types.
- New operations and query language constructs were necessary to manipulate the new data types.
- New storage and indexing structures were needed.

This led DBMS developers to add functionality to their systems. Some functionality was general purpose, such as incorporating concepts from object-oriented databases into relational systems. Other functionality was special purpose, in the form of optional modules that could be used for specific applications. For example, users could buy a time series module to use with their relational DBMS for their time series application.

1.8 When Not to Use a DBMS

In spite of the advantages of using a DBMS, there are a few situations in which such a system may involve unnecessary overhead costs that would not be incurred in traditional file processing. The overhead costs of using a DBMS are due to the following:

- High initial investment in hardware, software, and training
- The generality that a DBMS provides for defining and processing data
- Overhead for providing security, concurrency control, recovery, and integrity functions

Additional problems may arise if the database designers and DBA do not properly design the database or if the database systems applications are not implemented properly. Hence, it may be more desirable to use regular files under the following circumstances:

- The database and applications are simple, well defined, and not expected to change.
- There are stringent real-time requirements for some programs that may not be met because of DBMS overhead.
- Multiple-user access to data is not required.

1.9 Summary

In this chapter we defined a database as a collection of related data, where *data* means recorded facts. A typical database represents some aspect of the real world and is used for specific purposes by one or more groups of users. A DBMS is a generalized software package for implementing and maintaining a computerized database. The database and software together form a database system. We identified several characteristics that distinguish the database approach from traditional file-processing applications. We then discussed the main categories of database users, or the "actors on the scene." We noted that, in addition to database users, there are several categories of support personnel, or "workers behind the scene," in a database environment.

We then presented a list of capabilities that should be provided by the DBMS software to the DBA, database designers, and users to help them design, administer, and use a database. Following this, we gave a brief historical perspective on the evolution of database applications. Finally, we discussed the overhead costs of using a DBMS and discussed some situations in which it may not be advantageous to use a DBMS.

Review Questions

1.1. Define the following terms: *data*, *database*, *DBMS*, *database system*, *database catalog*, *program-data independence*, *user view*, *DBA*, *end user*, *canned transaction*, *deductive database system*, *persistent object*, *meta-data*, *transaction-processing application*.

1.2. What three main types of actions involve databases? Briefly discuss each.

1.3. Discuss the main characteristics of the database approach and how it differs from traditional file systems.
1.4. What are the responsibilities of the DBA and the database designers?
1.5. What are the different types of database end users? Discuss the main activities of each.
1.6. Discuss the capabilities that should be provided by a DBMS.

Exercises

1.7. Identify some informal queries and update operations that you would expect to apply to the database shown in Figure 1.2.
1.8. What is the difference between controlled and uncontrolled redundancy? Illustrate with examples.
1.9. Name all the relationships among the records of the database shown in Figure 1.2.
1.10. Give some additional views that may be needed by other user groups for the database shown in Figure 1.2.
1.11. Cite some examples of integrity constraints that you think should hold on the database shown in Figure 1.2.

Selected Bibliography

The October 1991 issue of *Communications of the* ACM and Kim (1995) include several articles describing next-generation DBMSs; many of the database features discussed in the former are now commercially available. The March 1976 issue of ACM *Computing Surveys* offers an early introduction to database systems and may provide a historical perspective for the interested reader.

2

Database System Concepts and Architecture

The architecture of DBMS packages has evolved from the early monolithic systems, where the whole DBMS software package was one tightly integrated system, to the modern DBMS packages that are modular in design, with a client/server system architecture. This evolution mirrors the trends in computing, where large centralized mainframe computers are being replaced by hundreds of distributed workstations and personal computers connected via communications networks to various types of server machines—Web servers, database servers, file servers, application servers, and so on.

In a basic client/server DBMS architecture, the system functionality is distributed between two types of modules.[1] A **client module** is typically designed so that it will run on a user workstation or personal computer. Typically, application programs and user interfaces that access the database run in the client module. Hence, the client module handles user interaction and provides the user-friendly interfaces such as forms- or menu-based GUIs (Graphical User Interfaces). The other kind of module, called a **server module,** typically handles data storage, access, search, and other functions. We discuss client/server architectures in more detail in Section 2.5. First, we must study more basic concepts that will give us a better understanding of modern database architectures.

In this chapter we present the terminology and basic concepts that will be used throughout the book. We start, in Section 2.1, by discussing data models and defining the

1. As we shall see in Section 2.5, there are variations on this simple *two-tier* client/server architecture.

concepts of schemas and instances, which are fundamental to the study of database systems. We then discuss the three-schema DBMS architecture and data independence in Section 2.2; this provides a user's perspective on what a DBMS is supposed to do. In Section 2.3, we describe the types of interfaces and languages that are typically provided by a DBMS. Section 2.4 discusses the database system software environment. Section 2.5 gives an overview of various types of client/server architectures. Finally, Section 2.6 presents a classification of the types of DBMS packages. Section 2.7 summarizes the chapter.

The material in Sections 2.4 through 2.6 provides more detailed concepts that may be looked upon as a supplement to the basic introductory material.

2.1 Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of data abstraction by hiding details of data storage that are not needed by most database users. A **data model**—a collection of concepts that can be used to describe the structure of a database—provides the necessary means to achieve this abstraction.[2] By *structure of a database*, we mean the data types, relationships, and constraints that should hold for the data. Most data models also include a set of **basic operations** for specifying retrievals and updates on the database.

In addition to the basic operations provided by the data model, it is becoming more common to include concepts in the data model to specify the **dynamic aspect** or **behavior** of a database application. This allows the database designer to specify a set of valid **user-defined operations** that are allowed on the database objects.[3] An example of a user-defined operation could be COMPUTE_GPA, which can be applied to a STUDENT object. On the other hand, generic operations to insert, delete, modify, or retrieve any kind of object are often included in the *basic data model operations*. Concepts to specify behavior are fundamental to object-oriented data models (see Chapters 20 and 21) but are also being incorporated in more traditional data models. For example, object-relational models (see Chapter 22) extend the traditional relational model to include such concepts, among others.

2.1.1 Categories of Data Models

Many data models have been proposed, which we can categorize according to the types of concepts they use to describe the database structure. **High-level** or **conceptual data models** provide concepts that are close to the way many users perceive data, whereas **low-level** or **physical data models** provide concepts that describe the details of how data is stored in

2. Sometimes the word *model* is used to denote a specific database description, or schema—for example, "the marketing data model." We will not use this interpretation.

3. The inclusion of concepts to describe behavior reflects a trend whereby database design and software design activities are increasingly being combined into a single activity. Traditionally, specifying behavior is associated with software design.

the computer. Concepts provided by low-level data models are generally meant for computer specialists, not for typical end users. Between these two extremes is a class of **representational (or implementation) data models,** which provide concepts that may be understood by end users but that are not too far removed from the way data is organized within the computer. Representational data models hide some details of data storage but can be implemented on a computer system in a direct way.

Conceptual data models use concepts such as entities, attributes, and relationships. An **entity** represents a real-world object or concept, such as an employee or a project, that is described in the database. An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary. A **relationship** among two or more entities represents an association among two or more entities, for example, a works-on relationship between an employee and a project. Chapter 3 presents the entity-relationship model—a popular high-level conceptual data model. Chapter 4 describes additional conceptual data modeling concepts, such as generalization, specialization, and categories.

Representational or implementation data models are the models used most frequently in traditional commercial DBMSs. These include the widely used **relational data model,** as well as the so-called legacy data models—the **network** and **hierarchical models**—that have been widely used in the past. Part II of this book is devoted to the relational data model, its operations and languages, and some of the techniques for programming relational database applications.[4] The SQL standard for relational databases is described in Chapters 8 and 9. Representational data models represent data by using record structures and hence are sometimes called **record-based data models.**

We can regard **object data models** as a new family of higher-level implementation data models that are closer to conceptual data models. We describe the general characteristics of object databases and the ODMG proposed standard in Chapters 20 and 21. Object data models are also frequently utilized as high-level conceptual models, particularly in the software engineering domain.

Physical data models describe how data is stored as files in the computer by representing information such as record formats, record orderings, and access paths. An **access path** is a structure that makes the search for particular database records efficient. We discuss physical storage techniques and access structures in Chapters 13 and 14.

2.1.2 Schemas, Instances, and Database State

In any data model, it is important to distinguish between the *description* of the database and the *database itself*. The description of a database is called the **database schema,** which is specified during database design and is not expected to change frequently.[5] Most data

4. A summary of the network and hierarchical data models is included in Appendices E and F. The full chapters from the second edition of this book are accessible from the Web site.

5. Schema changes are usually needed as the requirements of the database applications change. Newer database systems include operations for allowing schema changes, although the schema change process is more involved than simple database updates.

models have certain conventions for displaying schemas as diagrams.[6] A displayed schema is called a **schema diagram.** Figure 2.1 shows a schema diagram for the database shown in Figure 1.2; the diagram displays the structure of each record type but not the actual instances of records. We call each object in the schema—such as STUDENT or COURSE—a **schema construct.**

A schema diagram displays only *some aspects* of a schema, such as the names of record types and data items, and some types of constraints. Other aspects are not specified in the schema diagram; for example, Figure 2.1 shows neither the data type of each data item nor the relationships among the various files. Many types of constraints are not represented in schema diagrams. A constraint such as "students majoring in computer science must take CS1310 before the end of their sophomore year" is quite difficult to represent.

The actual data in a database may change quite frequently. For example, the database shown in Figure 1.2 changes every time we add a student or enter a new grade for a student. The data in the database at a particular moment in time is called a **database state** or **snapshot.** It is also called the *current* set of **occurrences** or **instances** in the database. In a given database state, each schema construct has its own *current set* of instances; for example, the STUDENT construct will contain the set of individual student entities (records) as its instances. Many database states can be constructed to correspond to a particular database schema. Every time we insert or delete a record or change the value of a data item in a record, we change one state of the database into another state.

The distinction between database schema and database state is very important. When we **define** a new database, we specify its database schema only to the DBMS. At this

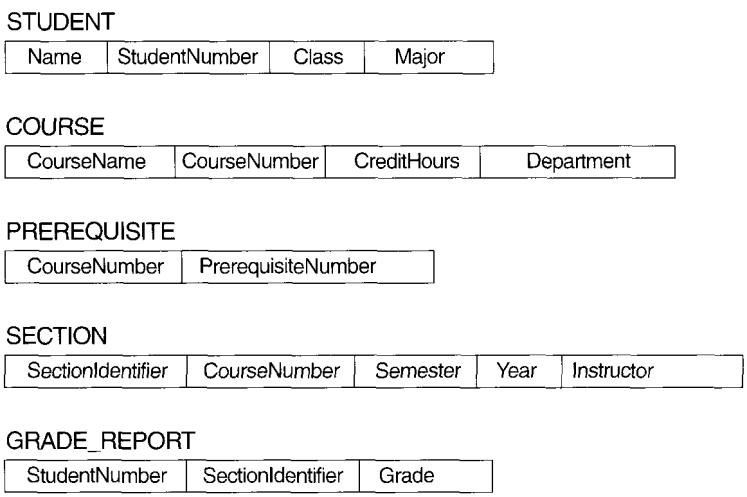

FIGURE 2.1 Schema diagram for the database in Figure 1.2.

6. It is customary in database parlance to use *schemas* as the plural for *schema*, even though *schemata* is the proper plural form. The word *scheme* is sometimes used for a schema.

point, the corresponding database state is the *empty state* with no data. We get the *initial state* of the database when the database is first **populated** or **loaded** with the initial data. From then on, every time an update operation is applied to the database, we get another database state. At any point in time, the database has a *current state*.[7] The DBMS is partly responsible for ensuring that *every* state of the database is a **valid state**—that is, a state that satisfies the structure and constraints specified in the schema. Hence, specifying a correct schema to the DBMS is extremely important, and the schema must be designed with the utmost care. The DBMS stores the descriptions of the schema constructs and constraints—also called the **meta-data**—in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to. The schema is sometimes called the **intension,** and a database state an **extension** of the schema.

Although, as mentioned earlier, the schema is not supposed to change frequently, it is not uncommon that changes need to be occasionally applied to the schema as the application requirements change. For example, we may decide that another data item needs to be stored for each record in a file, such as adding the DateOfBirth to the STUDENT schema in Figure 2.1. This is known as **schema evolution.** Most modern DBMSs include some operations for schema evolution that can be applied while the database is operational.

2.2 Three-Schema Architecture and Data Independence

Three of the four important characteristics of the database approach, listed in Section 1.3, are (1) insulation of programs and data (program-data and program-operation independence), (2) support of multiple user views, and (3) use of a catalog to store the database description (schema). In this section we specify an architecture for database systems, called the **three-schema architecture,**[8] that was proposed to help achieve and visualize these characteristics. We then further discuss the concept of data independence.

2.2.1 The Three-Schema Architecture

The goal of the three-schema architecture, illustrated in Figure 2.2, is to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels:

1. The **internal level** has an **internal schema,** which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.

7. The current state is also called the *current snapshot* of the database.

8. This is also known as the ANSI/SPARC architecture, after the committee that proposed it (Tsichritzis and Klug 1978).

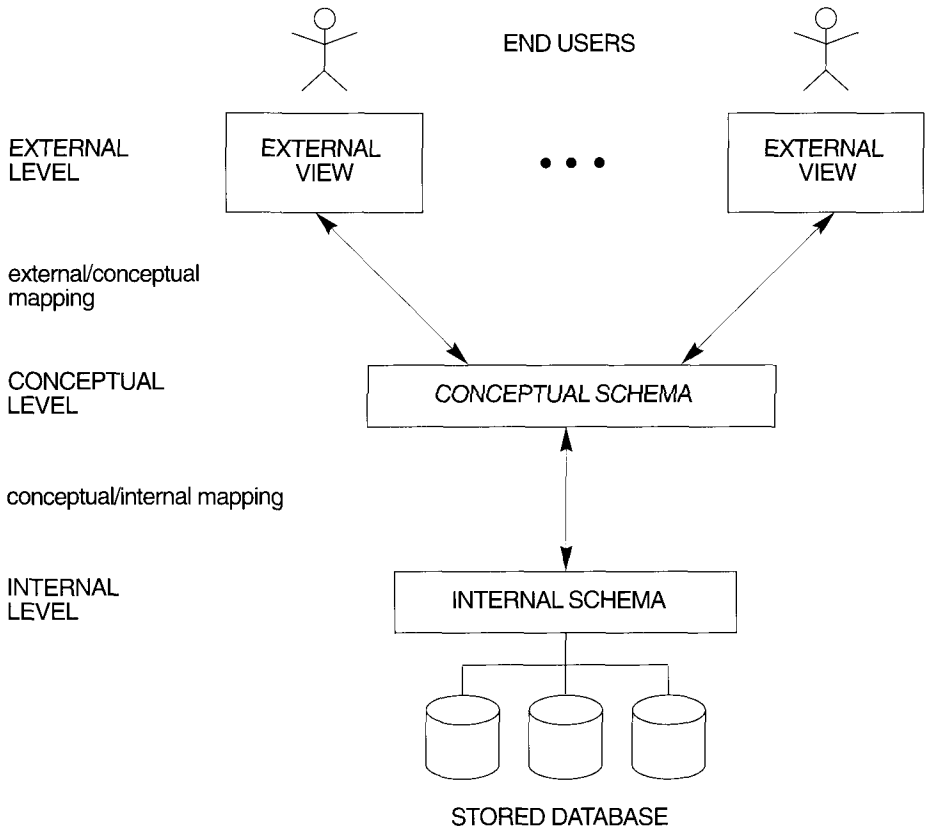

FIGURE 2.2 The three-schema architecture.

2. The **conceptual level** has a **conceptual schema,** which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. Usually, a representational data model is used to describe the conceptual schema when a database system is implemented. This *implementation conceptual schema* is often based on a *conceptual schema design* in a high-level data model.
3. The **external** or **view level** includes a number of **external schemas** or **user views.** Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. As in the previous case, each external schema is typically implemented using a representational data model, possibly based on an external schema design in a high-level data model.

The three-schema architecture is a convenient tool with which the user can visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely, but support the three-schema architecture to some extent. Some DBMSs may

include physical-level details in the conceptual schema. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information. Some DBMSs allow different data models to be used at the conceptual and external levels.

Notice that the three schemas are only *descriptions* of data; the only data that *actually* exists is at the physical level. In a DBMS based on the three-schema architecture, each user group refers only to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called **mappings.** These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

2.2.2 Data Independence

The three-schema architecture can be used to further explain the concept of **data independence,** which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item). In the last case, external schemas that refer only to the remaining data should not be affected. For example, the external schema of Figure 1.4a should not be affected by changing the GRADE_REPORT file shown in Figure 1.2 into the one shown in Figure 1.5a. Only the view definition and the mappings need be changed in a DBMS that supports logical data independence. After the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before. Changes to constraints can be applied to the conceptual schema without affecting the external schemas or application programs.
2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well. Changes to the internal schema may be needed because some physical files had to be reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema. For example, providing an access path to improve retrieval speed of SECTION records (Figure 1.2) by Semester and Year should not require a query such as "list all sections offered in fall 1998" to be changed, although the query would be executed more efficiently by the DBMS by utilizing the new access path.

Whenever we have a multiple-level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. The DBMS uses additional software to accomplish these mappings by referring to the mapping information in the catalog. Data independence occurs because when the schema is changed at some level, the schema at the next higher level remains unchanged; only the *mapping* between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed.

The three-schema architecture can make it easier to achieve true data independence, both physical and logical. However, the two levels of mappings create an overhead during compilation or execution of a query or program, leading to inefficiencies in the DBMS. Because of this, few DBMSs have implemented the full three-schema architecture.

2.3 Database Languages and Interfaces

In Section 1.4 we discussed the variety of users supported by a DBMS. The DBMS must provide appropriate languages and interfaces for each category of users. In this section we discuss the types of languages and interfaces provided by a DBMS and the user categories targeted by each interface.

2.3.1 DBMS Languages

Once the design of a database is completed and a DBMS is chosen to implement the database, the first order of the day is to specify conceptual and internal schemas for the database and any mappings between the two. In many DBMSs where no strict separation of levels is maintained, one language, called the **data definition language** (DDL), is used by the DBA and by database designers to define both schemas. The DBMS will have a DDL compiler whose function is to process DDL statements in order to identify descriptions of the schema constructs and to store the schema description in the DBMS catalog.

In DBMSs where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only. Another language, the **storage definition language** (SDL), is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages. For a true three-schema architecture, we would need a third language, the **view definition language** (VDL), to specify user views and their mappings to the conceptual schema, but in most DBMSs the DDL is used to define both conceptual and external schemas.

Once the database schemas are compiled and the database is populated with data, users must have some means to manipulate the database. Typical manipulations include retrieval, insertion, deletion, and modification of the data. The DBMS provides a set of operations or a language called the **data manipulation language** (DML) for these purposes.

In current DBMSs, the preceding types of languages are usually *not considered distinct languages;* rather, a comprehensive integrated language is used that includes constructs for conceptual schema definition, view definition, and data manipulation. Storage definition is typically kept separate, since it is used for defining physical storage structures to fine-

tune the performance of the database system, which is usually done by the DBA staff. A typical example of a comprehensive database language is the SQL relational database language (see Chapters 8 and 9), which represents a combination of DDL, VDL, and DML, as well as statements for constraint specification, schema evolution, and other features. The SDL was a component in early versions of SQL but has been removed from the language to keep it at the conceptual and external levels only.

There are two main types of DMLs. A **high-level** or **nonprocedural** DML can be used on its own to specify complex database operations in a concise manner. Many DBMSs allow high-level DML statements either to be entered interactively from a display monitor or terminal or to be embedded in a general-purpose programming language. In the latter case, DML statements must be identified within the program so that they can be extracted by a precompiler and processed by the DBMS. A **low-level** or **procedural** DML *must* be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately. Hence, it needs to use programming language constructs, such as looping, to retrieve and process each record from a set of records. Low-level DMLs are also called **record-at-a-time** DMLs because of this property. High-level DMLs, such as SQL, can specify and retrieve many records in a single DML statement and are hence called **set-at-a-time** or **set-oriented** DMLs. A query in a high-level DML often specifies *which* data to retrieve rather than *how* to retrieve it; hence, such languages are also called **declarative.**

Whenever DML commands, whether high level or low level, are embedded in a general-purpose programming language, that language is called the **host language** and the DML is called the **data sublanguage.**[9] On the other hand, a high-level DML used in a stand-alone interactive manner is called a **query language.** In general, both retrieval and update commands of a high-level DML may be used interactively and are hence considered part of the query language.[10]

Casual end users typically use a high-level query language to specify their requests, whereas programmers use the DML in its embedded form. For naive and parametric users, there usually are **user-friendly interfaces** for interacting with the database; these can also be used by casual users or others who do not want to learn the details of a high-level query language. We discuss these types of interfaces next.

2.3.2 DBMS Interfaces

User-friendly interfaces provided by a DBMS may include the following.

Menu-Based Interfaces for Web Clients or Browsing. These interfaces present the user with lists of options, called **menus,** that lead the user through the formulation of

9. In object databases, the host and data sublanguages typically form one integrated language—for example, C++ with some extensions to support database functionality. Some relational systems also provide integrated languages—for example, Oracle's PL/SQL.

10. According to the meaning of the word *query* in English, it should really be used to describe only retrievals, not updates.

a request. Menus do away with the need to memorize the specific commands and syntax of a query language; rather, the query is composed step by step by picking options from a menu that is displayed by the system. Pull-down menus are a very popular technique in **Web-based user interfaces**. They are also often used in **browsing interfaces,** which allow a user to look through the contents of a database in an exploratory and unstructured manner.

Forms-Based Interfaces. A forms-based interface displays a **form** to each user. Users can fill out all of the form entries to insert new data, or they fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions. Many DBMSs have **forms specification languages,** which are special languages that help programmers specify such forms. Some systems have utilities that define a form by letting the end user interactively construct a sample form on the screen.

Graphical User Interfaces. A graphical interface (GUI) typically displays a schema to the user in diagrammatic form. The user can then specify a query by manipulating the diagram. In many cases, GUIs utilize both menus and forms. Most GUIs use a **pointing device,** such as a mouse, to pick certain parts of the displayed schema diagram.

Natural Language Interfaces. These interfaces accept requests written in English or some other language and attempt to "understand" them. A natural language interface usually has its own "schema," which is similar to the database conceptual schema, as well as a dictionary of important words. The natural language interface refers to the words in its schema, as well as to the set of standard words in its dictionary, to interpret the request. If the interpretation is successful, the interface generates a high-level query corresponding to the natural language request and submits it to the DBMS for processing; otherwise, a dialogue is started with the user to clarify the request.

Interfaces for Parametric Users. Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. Systems analysts and programmers design and implement a special interface for each known class of naive users. Usually, a small set of abbreviated commands is included, with the goal of minimizing the number of keystrokes required for each request. For example, function keys in a terminal can be programmed to initiate the various commands. This allows the parametric user to proceed with a minimal number of keystrokes.

Interfaces for the DBA. Most database systems contain privileged commands that can be used only by the DBA's staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

2.4 The Database System Environment

A DBMS is a complex software system. In this section we discuss the types of software components that constitute a DBMS and the types of computer system software with which the DBMS interacts.

2.4.1 DBMS Component Modules

Figure 2.3 illustrates, in a simplified form, the typical DBMS components. The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the **operating system** (OS), which schedules disk input/output. A higher-level **stored data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog. The dotted lines and circles marked

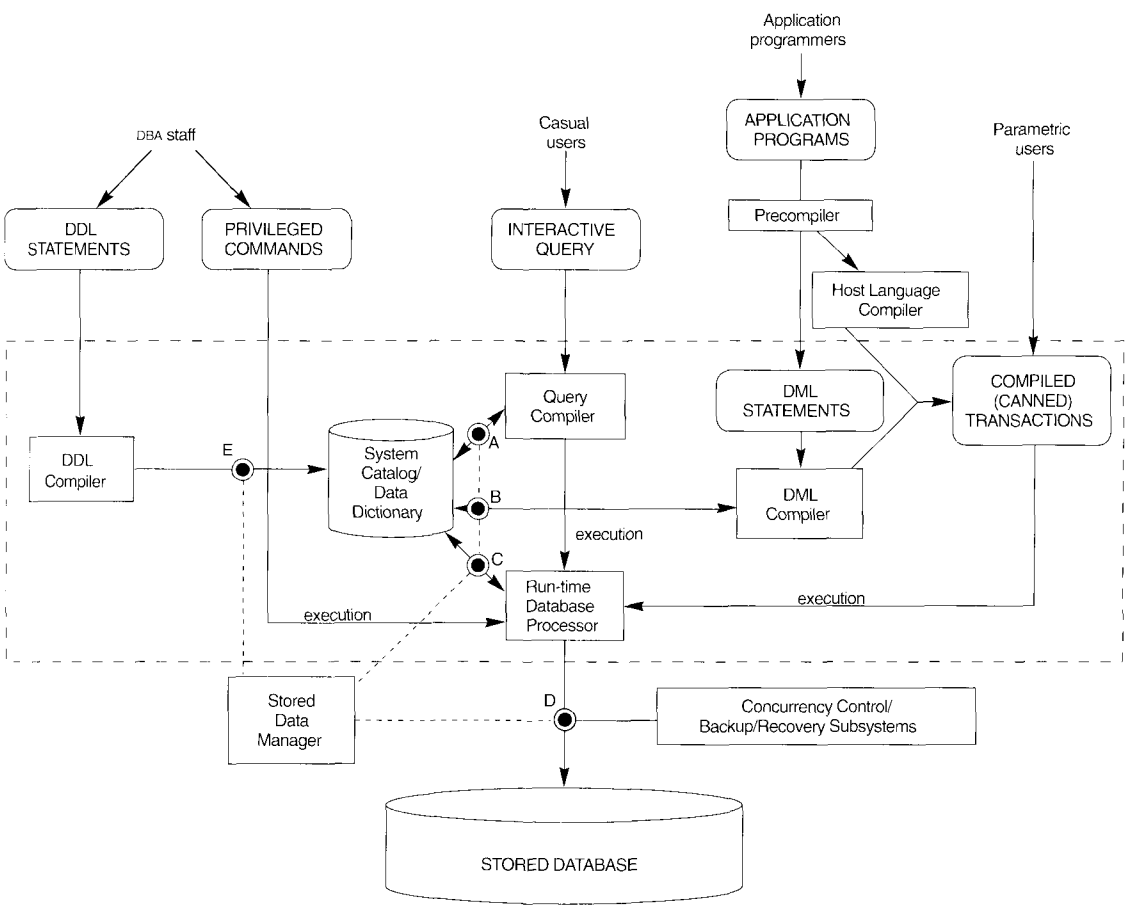

FIGURE 2.3 Component modules of a DBMS and their interactions.

A, B, C, D, and E in Figure 2.3 illustrate accesses that are under the control of this stored data manager. The stored data manager may use basic OS services for carrying out low-level data transfer between the disk and computer main storage, but it controls other aspects of data transfer, such as handling buffers in main memory. Once the data is in main memory buffers, it can be processed by other DBMS modules, as well as by application programs. Some DBMSs have their own **buffer manager module,** while others use the OS for handling the buffering of disk pages.

The **DDL** compiler processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names and sizes of files, names and data types of data items, storage details of each file, mapping information among schemas, and constraints, in addition to many other types of information that are needed by the DBMS modules. DBMS software modules then look up the catalog information as needed.

The **runtime database processor** handles database accesses at runtime; it receives retrieval or update operations and carries them out on the database. Access to disk goes through the stored data manager, and the buffer manager keeps track of the database pages in memory. The **query compiler** handles high-level queries that are entered interactively. It parses, analyzes, and compiles or interprets a query by creating database access code, and then generates calls to the runtime processor for executing the code.

The **precompiler** extracts DML commands from an application program written in a

- *Loading:* A loading utility is used to load existing data files—such as text files or sequential files—into the database. Usually, the current (source) format of the data file and the desired (target) database file structure are specified to the utility, which then automatically reformats the data and stores it in the database. With the proliferation of DBMSs, transferring data from one DBMS to another is becoming common in many organizations. Some vendors are offering products that generate the appropriate loading programs, given the existing source and target database storage descriptions (internal schemas). Such tools are also called **conversion tools.**
- *Backup:* A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape. The backup copy can be used to restore the database in case of catastrophic failure. Incremental backups are also often used, where only changes since the previous backup are recorded. Incremental backup is more complex but saves space.
- *File reorganization:* This utility can be used to reorganize a database file into a different file organization to improve performance.
- *Performance monitoring:* Such a utility monitors database usage and provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files to improve performance.

Other utilities may be available for sorting files, handling data compression, monitoring access by users, interfacing with the network, and performing other functions.

2.4.3 Tools, Application Environments, and Communications Facilities

Other tools are often available to database designers, users, and DBAs. **CASE** tools[11] are used in the design phase of database systems. Another tool that can be quite useful in large organizations is an expanded **data dictionary** (or **data repository**) **system.** In addition to storing catalog information about schemas and constraints, the data dictionary stores other information, such as design decisions, usage standards, application program descriptions, and user information. Such a system is also called an **information repository.** This information can be accessed *directly* by users or the DBA when needed. A data dictionary utility is similar to the DBMS catalog, but it includes a wider variety of information and is accessed mainly by users rather than by the DBMS software.

Application development environments, such as the PowerBuilder (Sybase) or JBuilder (Borland) system, are becoming quite popular. These systems provide an environment for developing database applications and include facilities that help in many facets of database systems, including database design, GUI development, querying and updating, and application program development.

11. Although CASE stands for computer-aided software engineering, many CASE tools are used primarily for database design.

The DBMS also needs to interface with **communications software,** whose function is to allow users at locations remote from the database system site to access the database through computer terminals, workstations, or their local personal computers. These are connected to the database site through data communications hardware such as phone lines, long-haul networks, local area networks, or satellite communication devices. Many commercial database systems have communication packages that work with the DBMS. The integrated DBMS and data communications system is called a **DB/DC** system. In addition, some distributed DBMSs are physically distributed over multiple machines. In this case, communications networks are needed to connect the machines. These are often **local area networks (LANs)**, but they can also be other types of networks.

2.5 Centralized and Client/Server Architectures for DBMSs

2.5.1 Centralized DBMSs Architecture

Architectures for DBMSs have followed trends similar to those for general computer system architectures. Earlier architectures used mainframe computers to provide the main processing for all functions of the system, including user application programs and user interface programs, as well as all the DBMS functionality. The reason was that most users accessed such systems via computer terminals that did not have processing power and only provided display capabilities. So, all processing was performed remotely on the computer system, and only display information and controls were sent from the computer to the display terminals, which were connected to the central computer via various types of communications networks.

As prices of hardware declined, most users replaced their terminals with personal computers (PCs) and workstations. At first, database systems used these computers in the same way as they had used display terminals, so that the DBMS itself was still a **centralized** DBMS in which all the DBMS functionality, application program execution, and user interface processing were carried out on one machine. Figure 2.4 illustrates the physical components in a centralized architecture. Gradually, DBMS systems started to exploit the available processing power at the user side, which led to client/server DBMS architectures.

2.5.2 Basic Client/Server Architectures

We first discuss client/server architecture in general, then see how it is applied to DBMSs. The **client/server architecture** was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, database servers, Web servers, and other equipment are connected via a network. The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a **file server** that maintains the files of the client

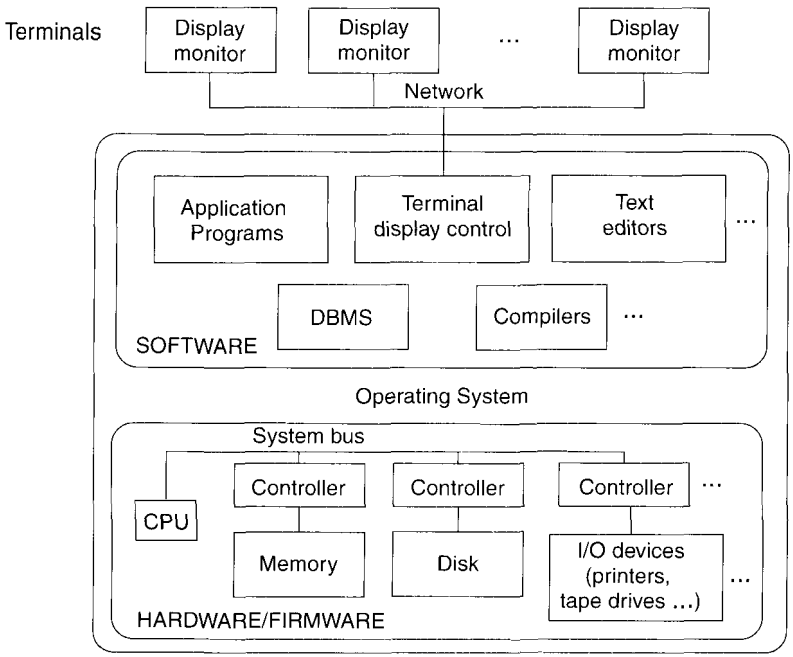

FIGURE 2.4 A physical centralized architecture.

machines. Another machine could be designated as a **printer server** by being connected to various printers; thereafter, all print requests by the clients are forwarded to this machine. **Web servers** or **e-mail servers** also fall into the specialized server category. In this way, the resources provided by specialized servers can be accessed by many client machines. The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications. This concept can be carried over to software, with specialized software—such as a DBMS or a CAD (computer-aided design) package—being stored on specific server machines and being made accessible to multiple clients. Figure 2.5 illustrates client/server architecture at the logical level, and Figure 2.6 is a simplified diagram that shows how the physical

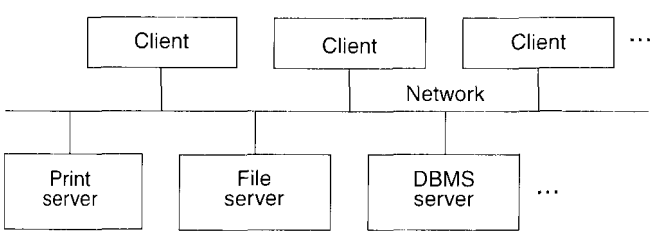

FIGURE 2.5 Logical two-tier client/server architecture.

architecture would look. Some machines would be only client sites (for example, diskless workstations or workstations/PCs with disks that have only client software installed). Other machines would be dedicated servers. Still other machines would have both client and server functionality.

The concept of client/server architecture assumes an underlying framework that consists of many PCs and workstations as well as a smaller number of mainframe machines, connected via local area networks and other types of computer networks. A **client** in this framework is typically a user machine that provides user interface capabilities and local processing. When a client requires access to additional functionality—such as database access—that does not exist at that machine, it connects to a server that provides the needed functionality. A **server** is a machine that can provide services to the client machines, such as file access, printing, archiving, or database access. In the general case, some machines install only client software, others only server software, and still others may include both client and server software, as illustrated in Figure 2.6. However, it is more common that client and server software usually run on separate machines. Two main types of basic DBMS architectures were created on this underlying client/server framework: two-tier and three-tier.[12] We discuss those next.

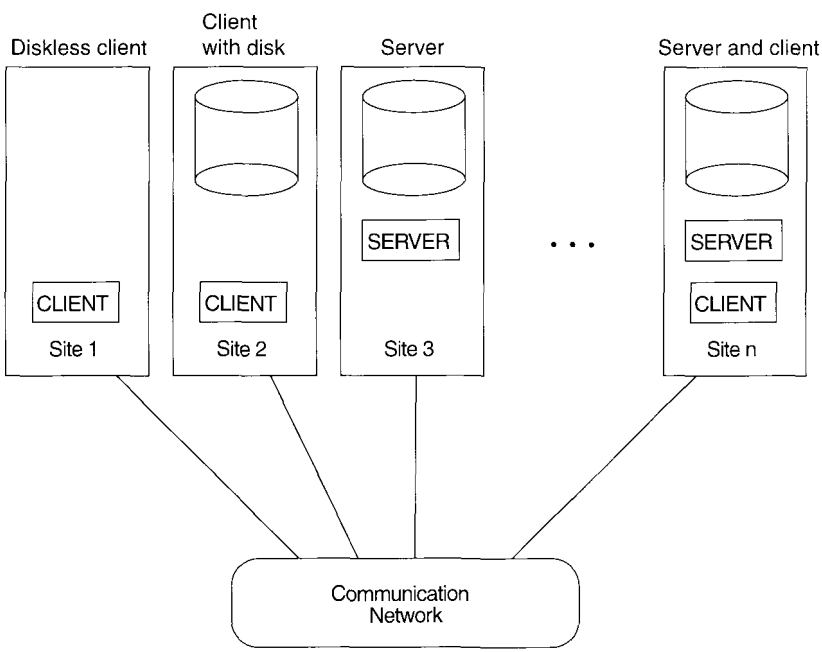

FIGURE 2.6 Physical two-tier client-server architecture.

12. There are many other variations of client/server architectures. We only discuss the two most basic ones here. In Chapter 25, we discuss additional client/server and distributed architectures.

2.5.3 Two-Tier Client/Server Architectures for DBMSs

The client/server architecture is increasingly being incorporated into commercial DBMS packages. In relational DBMSs (RDBMSs), many of which started as centralized systems, the system components that were first moved to the client side were the user interface and application programs. Because SQL (see Chapters 8 and 9) provided a standard language for RDBMSs, this created a logical dividing point between client and server. Hence, the query and transaction functionality remained on the server side. In such an architecture, the server is often called a **query server** or **transaction server,** because it provides these two functionalities. In RDBMSs, the server is also often called an **SQL server,** since most RDBMS servers are based on the SQL language and standard.

In such a client/server architecture, the user interface programs and application programs can run on the client side. When DBMS access is required, the program establishes a connection to the DBMS (which is on the server side); once the connection is created, the client program can communicate with the DBMS. A standard called **Open Database Connectivity (ODBC)** provides an **application programming interface (API),** which allows client-side programs to call the DBMS, as long as both client and server machines have the necessary software installed. Most DBMS vendors provide ODBC drivers for their systems. Hence, a client program can actually connect to several RDBMSs and send query and transaction requests using the ODBC API, which are then processed at the server sites. Any query results are sent back to the client program, which can process or display the results as needed. A related standard for the Java programming language, called **JDBC,** has also been defined. This allows Java client programs to access the DBMS through a standard interface.

The second approach to client/server architecture was taken by some object-oriented DBMSs. Because many of these systems were developed in the era of client/server architecture, the approach taken was to divide the software modules of the DBMS between client and server in a more integrated way. For example, the **server level** may include the part of the DBMS software responsible for handling data storage on disk pages, local concurrency control and recovery, buffering and caching of disk pages, and other such functions. Meanwhile, the **client level** may handle the user interface; data dictionary functions; DBMS interactions with programming language compilers; global query optimization, concurrency control, and recovery across multiple servers; structuring of complex objects from the data in the buffers; and other such functions. In this approach, the client/server interaction is more tightly coupled and is done internally by the DBMS modules—some of which reside on the client and some on the server—rather than by the users. The exact division of functionality varies from system to system. In such a client/server architecture, the server has been called a **data server,** because it provides data in disk pages to the client. This data can then be structured into objects for the client programs by the client-side DBMS software itself.

The architectures described here are called **two-tier architectures** because the software components are distributed over two systems: client and server. The advantages of this architecture are its simplicity and seamless compatibility with existing systems. The emergence of the World Wide Web changed the roles of clients and server, leading to the three-tier architecture.

2.5.4 Three-Tier Client/Server Architectures for Web Applications

Many Web applications use an architecture called the **three-tier architecture,** which adds an intermediate layer between the client and the database server, as illustrated in Figure 2.7. This intermediate layer or **middle tier** is sometimes called the **application server** and sometimes the **Web server,** depending on the application. This server plays an intermediary role by storing business rules (procedures or constraints) that are used to access data from the database server. It can also improve database security by checking a client's credentials before forwarding a request to the database server. Clients contain GUI interfaces and some additional application-specific business rules. The intermediate server accepts requests from the client, processes the request and sends database commands to the database server, and then acts as a conduit for passing (partially) processed data from the database server to the clients, where it may be processed further and filtered to be presented to users in GUI format. Thus, the *user interface*, *application rules*, and *data access* act as the three tiers.

Advances in encryption and decryption technology make it safer to transfer sensitive data from server to client in encrypted form, where it will be decrypted. The latter can be done by the hardware or by advanced software. This technology gives higher levels of data security, but the network security issues remain a major concern. Various technologies for data compression are also helping in transferring large amounts of data from servers to clients over wired and wireless networks.

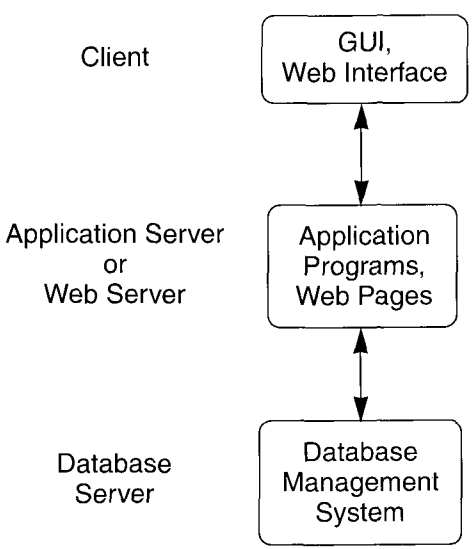

FIGURE 2.7 Logical three-tier client/server architecture.

2.6 Classification of Database Management Systems

Several criteria are normally used to classify DBMSs. The first is the **data model** on which the DBMS is based. The main data model used in many current commercial DBMSs is the **relational data model.** The **object data model** was implemented in some commercial systems but has not had widespread use. Many legacy (older) applications still run on database systems based on the **hierarchical** and **network data models.** The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed in object databases. This has led to a new class of DBMSs called **object-relational** DBMSs. We can hence categorize DBMSs based on the data model: relational, object, object-relational, hierarchical, network, and other.

The second criterion used to classify DBMSs is the **number of users** supported by the system. **Single-user systems** support only one user at a time and are mostly used with personal computers. **Multiuser systems,** which include the majority of DBMSs, support multiple users concurrently.

A third criterion is the **number of sites** over which the database is distributed. A DBMS is **centralized** if the data is stored at a single computer site. A centralized DBMS can support multiple users, but the DBMS and the database themselves reside totally at a single computer site. A **distributed** DBMS (DDBMS) can have the actual database and DBMS software distributed over many sites, connected by a computer network. **Homogeneous** DDBMSs use the same DBMS software at multiple sites. A recent trend is to develop software to access several autonomous preexisting databases stored under **heterogeneous** DBMSs. This leads to a **federated** DBMS (or **multidatabase system**), in which the participating DBMSs are loosely coupled and have a degree of local autonomy. Many DDBMSs use a client-server architecture.

A fourth criterion is the **cost** of the DBMS. The majority of DBMS packages cost between $10,000 and $100,000. Single-user low-end systems that work with microcomputers cost between $100 and $3000. At the other end of the scale, a few elaborate packages cost more than $100,000.

We can also classify a DBMS on the basis of the **types of access path** options for storing files. One well-known family of DBMSs is based on inverted file structures. Finally, a DBMS can be **general purpose** or **special purpose.** When performance is a primary consideration, a special-purpose DBMS can be designed and built for a specific application; such a system cannot be used for other applications without major changes. Many airline reservations and telephone directory systems developed in the past are special purpose DBMSs. These fall into the category of **online transaction processing (OLTP)** systems, which must support a large number of concurrent transactions without imposing excessive delays.

Let us briefly elaborate on the main criterion for classifying DBMSs: the data model. The basic relational data model represents a database as a collection of tables, where each table can be stored as a separate file. The database in Figure 1.2 is shown in a manner very similar to a relational representation. Most relational databases use the high-level query language called SQL and support a limited form of user views. We discuss the relational

model, its languages and operations, and techniques for programming relational applications in Chapters 5 through 9.

The object data model defines a database in terms of objects, their properties, and their operations. Objects with the same structure and behavior belong to a **class,** and classes are organized into **hierarchies** (or **acyclic graphs**). The operations of each class are specified in terms of predefined procedures called **methods.** Relational DBMSs have been extending their models to incorporate object database concepts and other capabilities; these systems are referred to as **object-relational** or **extended relational systems.** We discuss object databases and object-relational systems in Chapters 20 to 22.

Two older, historically important data models, now known as legacy data models, are the network and hierarchical models. The **network model** represents data as record types and also represents a limited type of 1:N relationship, called a **set type.** Figure 2.8 shows a network schema diagram for the database of Figure 1.2, where record types are shown as rectangles and set types are shown as labeled directed arrows. The network model, also known as the CODASYL DBTG model,[13] has an associated record-at-a-time language that must be embedded in a host programming language. The **hierarchical model** represents data as hierarchical tree structures. Each hierarchy represents a number of related records. There is no standard language for the hierarchical model, although most hierarchical DBMSs have record-at-a-time languages. We give a brief overview of the network and hierarchical models in Appendices E and F.[14]

The **XML** (eXtended Markup Language) **model,** now considered the standard for data interchange over the Internet, also uses hierarchical tree structures. It combines database concepts with concepts from document representation models. Data is represented as elements, which can be nested to create complex hierarchical structures. This model

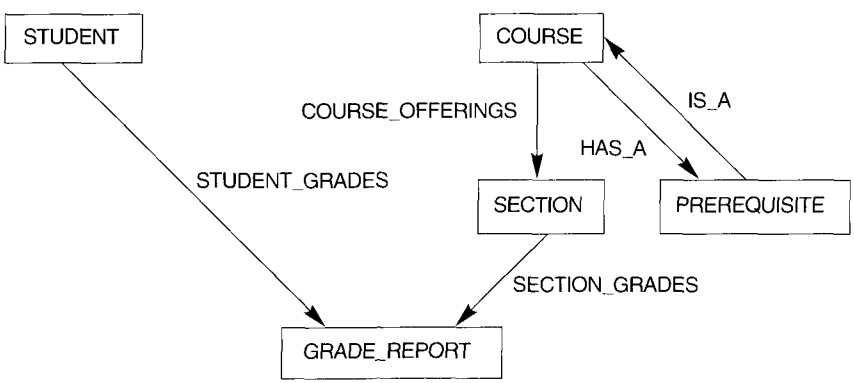

FIGURE 2.8 The schema of Figure 2.1 in network model notation

13. CODASYL DBTG stands for Conference on Data Systems Languages Data Base Task Group, which is the committee that specified the network model and its language.

14. The full chapters on the network and hierarchical models from the second edition of this book are available over the Internet from the Web site.

conceptually resembles the object model, but uses different terminology. We discuss XML and how it is related to databases in Chapter 26.

2.7 SUMMARY

In this chapter we introduced the main concepts used in database systems. We defined a data model, and we distinguished three main categories of data models:

- High-level or conceptual data models (based on entities and relationships)
- Low-level or physical data models
- Representational or implementation data models (record-based, object-oriented)

We distinguished the schema, or description of a database, from the database itself. The schema does not change very often, whereas the database state changes every time data is inserted, deleted, or modified. We then described the three-schema DBMS architecture, which allows three schema levels:

- An internal schema describes the physical storage structure of the database.
- A conceptual schema is a high-level description of the whole database.
- External schemas describe the views of different user groups.

A DBMS that cleanly separates the three levels must have mappings between the schemas to transform requests and results from one level to the next. Most DBMSs do not separate the three levels completely. We used the three-schema architecture to define the concepts of logical and physical data independence.

We then discussed the main types of languages and interfaces that DBMSs support. A data definition language (DDL) is used to define the database conceptual schema. In most DBMSs, the DDL also defines user views and, sometimes, storage structures; in other DBMSs, separate languages (VDL, SDL) may exist for specifying views and storage structures. The DBMS compiles all schema definitions and stores their descriptions in the DBMS catalog. A data manipulation language (DML) is used for specifying database retrievals and updates. DMLs can be high level (set-oriented, nonprocedural) or low level (record-oriented, procedural). A high-level DML can be embedded in a host programming language, or it can be used as a stand-alone language; in the latter case it is often called a query language.

We discussed different types of interfaces provided by DBMSs, and the types of DBMS users with which each interface is associated. We then discussed the database system environment, typical DBMS software modules, and DBMS utilities for helping users and the DBA perform their tasks. We then gave an overview of the two-tier and three-tier architectures for database applications, which are now very common in most modern applications, particularly Web database applications.

In the final section, we classified DBMSs according to several criteria: data model, number of users, number of sites, cost, types of access paths, and generality. The main classification of DBMSs is based on the data model. We briefly discussed the main data models used in current commercial DBMSs.

Review Questions

2.1. Define the following terms: *data model*, *database schema*, *database state*, *internal schema*, *conceptual schema*, *external schema*, *data independence*, *DDL*, *DML*, *SDL*, *VDL*, *query language*, *host language*, *data sublanguage*, *database utility*, *catalog*, *client/server architecture*.

2.2. Discuss the main categories of data models.

2.3. What is the difference between a database schema and a database state?

2.4. Describe the three-schema architecture. Why do we need mappings between schema levels? How do different schema definition languages support this architecture?

2.5. What is the difference between logical data independence and physical data independence?

2.6. What is the difference between procedural and nonprocedural DMLs?

2.7. Discuss the different types of user-friendly interfaces and the types of users who typically use each.

2.8. With what other computer system software does a DBMS interact?

2.9. What is the difference between the two-tier and three-tier client/server architectures?

2.10. Discuss some types of database utilities and tools and their functions.

Exercises

2.11. Think of different users for the database of Figure 1.2. What types of applications would each user need? To which user category would each belong, and what type of interface would each need?

2.12. Choose a database application with which you are familiar. Design a schema and show a sample database for that application, using the notation of Figures 2.1 and 1.2. What types of additional information and constraints would you like to represent in the schema? Think of several users for your database, and design a view for each.

Selected Bibliography

Many database textbooks, including Date (2001), Silberschatz et al. (2001), Ramakrishnan and Gehrke (2002), Garcia-Molina et al (1999, 2001), and Abiteboul et al. (1995), provide a discussion of the various database concepts presented here. Tsichritzis and Lochovsky (1982) is an early textbook on data models. Tsichritzis and Klug (1978) and Jardine (1977) present the three-schema architecture, which was first suggested in the DBTG CODASYL report (1971) and later in an American National Standards Institute (ANSI) report (1975). An in-depth analysis of the relational data model and some of its possible extensions is given in Codd (1992). The proposed standard for object-oriented databases is described in Cattell (1997). Many documents describing XML are available on the Web, such as XML (2003).

Examples of database utilities are the ETI Extract Toolkit (www.eti.com) and the database administration tool DB Artisan from Embarcadero Technologies (www.embarcadero.com).

3

Data Modeling Using the Entity-Relationship Model

Conceptual modeling is a very important phase in designing a successful database application. Generally, the term **database application** refers to a particular database and the associated programs that implement the database queries and updates. For example, a BANK database application that keeps track of customer accounts would include programs that implement database updates corresponding to customers making deposits and withdrawals. These programs provide user-friendly graphical user interfaces (GUIs) utilizing forms and menus for the end users of the application—the bank tellers, in this example. Hence, part of the database application will require the design, implementation, and testing of these **application programs.** Traditionally, the design and testing of application programs has been considered to be more in the realm of the software engineering domain than in the database domain. As database design methodologies include more of the concepts for specifying operations on database objects, and as software engineering methodologies specify in more detail the structure of the databases that software programs will use and access, it is clear that these activities are strongly related. We briefly discuss some of the concepts for specifying database operations in Chapter 4, and again when we discuss database design methodology with example applications in Chapter 12 of this book.

In this chapter, we follow the traditional approach of concentrating on the database structures and constraints during database design. We present the modeling concepts of the **Entity-Relationship (ER) model,** which is a popular high-level conceptual data model. This model and its variations are frequently used for the conceptual design of database applications, and many database design tools employ its concepts. We describe

the basic data-structuring concepts and constraints of the ER model and discuss their use in the design of conceptual schemas for database applications. We also present the diagrammatic notation associated with the ER model, known as **ER diagrams.**

Object modeling methodologies such as UML **(Universal Modeling Language)** are becoming increasingly popular in software design and engineering. These methodologies go beyond database design to specify detailed design of software modules and their interactions using various types of diagrams. An important part of these methodologies—namely, *class diagrams*[1]—are similar in many ways to the ER diagrams. In class diagrams, *operations* on objects are specified, in addition to specifying the database schema structure. Operations can be used to specify the *functional requirements* during database design, as discussed in Section 3.1. We present some of the UML notation and concepts for class diagrams that are particularly relevant to database design in Section 3.8, and briefly compare these to ER notation and concepts. Additional UML notation and concepts are presented in Section 4.6 and in Chapter 12.

This chapter is organized as follows. Section 3.1 discusses the role of high-level conceptual data models in database design. We introduce the requirements for an example database application in Section 3.2 to illustrate the use of concepts from the ER model. This example database is also used in subsequent chapters. In Section 3.3 we present the concepts of entities and attributes, and we gradually introduce the diagrammatic technique for displaying an ER schema. In Section 3.4 we introduce the concepts of binary relationships and their roles and structural constraints. Section 3.5 introduces weak entity types. Section 3.6 shows how a schema design is refined to include relationships. Section 3.7 reviews the notation for ER diagrams, summarizes the issues that arise in schema design, and discusses how to choose the names for database schema constructs. Section 3.8 introduces some UML class diagram concepts, compares them to ER model concepts, and applies them to the same database example. Section 3.9 summarizes the chapter.

The material in Sections 3.8 may be left out of an introductory course if desired. On the other hand, if more thorough coverage of data modeling concepts and conceptual database design is desired, the reader should continue on to the material in Chapter 4 after concluding Chapter 3. Chapter 4 describes extensions to the ER model that lead to the Enhanced-ER (EER) model, which includes concepts such as specialization, generalization, inheritance, and union types (categories). We also introduce some additional UML concepts and notation in Chapter 4.

3.1 Using High-Level Conceptual Data Models for Database Design

Figure 3.1 shows a simplified description of the database design process. The first step shown is **requirements collection and analysis.** During this step, the database designers interview prospective database users to understand and document their **data requirements.** The result of this

1. A **class** is similar to an *entity type* in many ways.

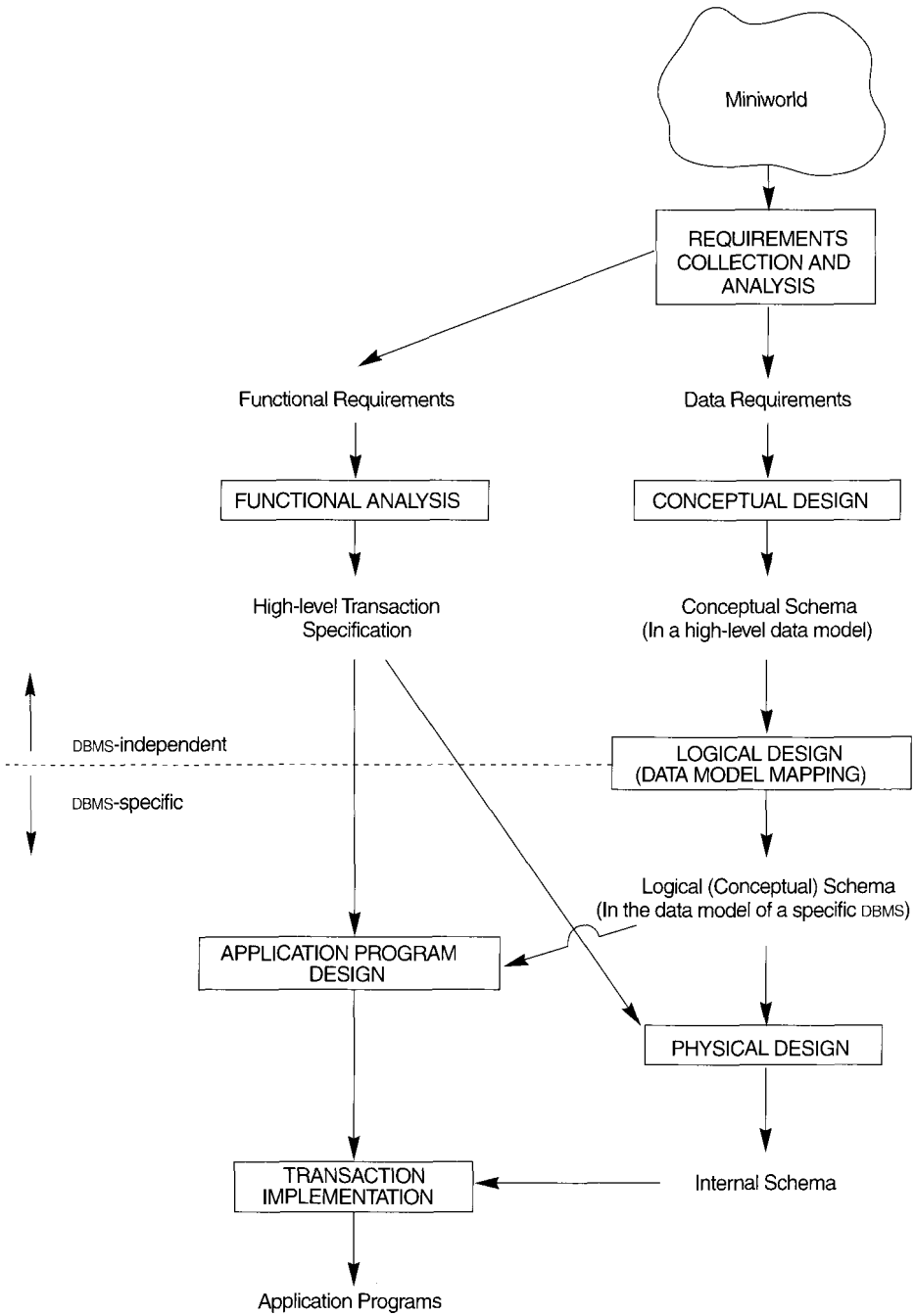

FIGURE 3.1 A simplified diagram to illustrate the main phases of database design.

step is a concisely written set of users' requirements. These requirements should be specified in as detailed and complete a form as possible. In parallel with specifying the data requirements, it is useful to specify the known **functional requirements** of the application. These consist of the user-defined **operations** (or **transactions**) that will be applied to the database, including both retrievals and updates. In software design, it is common to use *data flow diagrams*, *sequence diagrams*, *scenarios*, and other techniques for specifying functional requirements. We will not discuss any of these techniques here because they are usually described in detail in software engineering texts. We give an overview of some of these techniques in Chapter 12.

Once all the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called **conceptual design.** The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Because these concepts do not include implementation details, they are usually easier to understand and can be used to communicate with nontechnical users. The high-level conceptual schema can also be used as a reference to ensure that all users' data requirements are met and that the requirements do not conflict. This approach enables the database designers to concentrate on specifying the properties of the data, without being concerned with storage details. Consequently, it is easier for them to come up with a good conceptual database design.

During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user operations identified during functional analysis. This also serves to confirm that the conceptual schema meets all the identified functional requirements. Modifications to the conceptual schema can be introduced if some functional requirements cannot be specified using the initial schema.

The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational or the object-relational database model—so the conceptual schema is transformed from the high-level data model into the implementation data model. This step is called **logical design** or **data model mapping,** and its result is a database schema in the implementation data model of the DBMS.

The last step is the **physical design** phase, during which the internal storage structures, indexes, access paths, and file organizations for the database files are specified. In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications. We discuss the database design process in more detail in Chapter 12.

We present only the basic ER model concepts for conceptual schema design in this chapter. Additional modeling concepts are discussed in Chapter 4, when we introduce the EER model.

3.2 An Example Database Application

In this section we describe an example database application, called COMPANY, that serves to illustrate the basic ER model concepts and their use in schema design. We list the data requirements for the database here, and then create its conceptual schema step by step as

we introduce the modeling concepts of the ER model. The COMPANY database keeps track of a company's employees, departments, and projects. Suppose that after the requirements collection and analysis phase, the database designers provided the following description of the "miniworld"—the part of the company to be represented in the database:

1. The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.
2. A department controls a number of projects, each of which has a unique name, a unique number, and a single location.
3. We store each employee's name, social security number,[2] address, salary, sex, and birth date. An employee is assigned to one department but may work on several projects, which are not necessarily controlled by the same department. We keep track of the number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee.
4. We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's first name, sex, birth date, and relationship to the employee.

Figure 3.2 shows how the schema for this database application can be displayed by means of the graphical notation known as **ER diagrams.** We describe the step-by-step process of deriving this schema from the stated requirements—and explain the ER diagrammatic notation—as we introduce the ER model concepts in the following section.

3.3 Entity Types, Entity Sets, Attributes, and Keys

The ER model describes data as *entities*, *relationships*, and *attributes*. In Section 3.3.1 we introduce the concepts of entities and their attributes. We discuss entity types and key attributes in Section 3.3.2. Then, in Section 3.3.3, we specify the initial conceptual design of the entity types for the COMPANY database. Relationships are described in Section 3.4.

3.3.1 Entities and Attributes

Entities and Their Attributes. The basic object that the ER model represents is an **entity,** which is a "thing" in the real world with an independent existence. An entity may be an object with a physical existence (for example, a particular person, car, house, or

2. The social security number, or SSN, is a unique nine-digit identifier assigned to each individual in the United States to keep track of his or her employment, benefits, and taxes. Other countries may have similar identification schemes, such as personal identification card numbers.

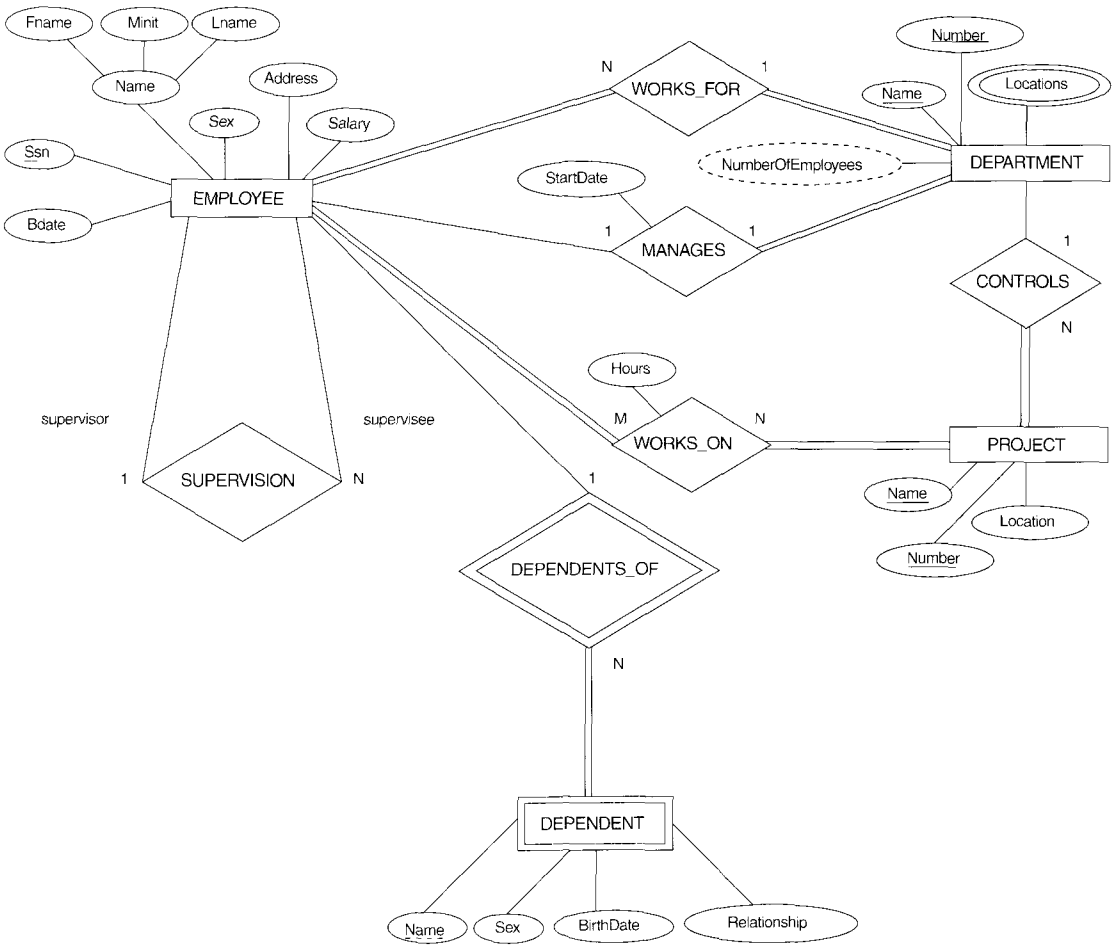

FIGURE 3.2 An ER schema diagram for the COMPANY database.

employee) or it may be an object with a conceptual existence (for example, a company, a job, or a university course). Each entity has **attributes**—the particular properties that describe it. For example, an employee entity may be described by the employee's name, age, address, salary, and job. A particular entity will have a value for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.

Figure 3.3 shows two entities and the values of their attributes. The employee entity e_1 has four attributes: Name, Address, Age, and HomePhone; their values are "John Smith," "2311 Kirby, Houston, Texas 77001," "55," and "713-749-2630," respectively. The company entity c_1 has three attributes: Name, Headquarters, and President; their values are "Sunco Oil," "Houston," and "John Smith," respectively.

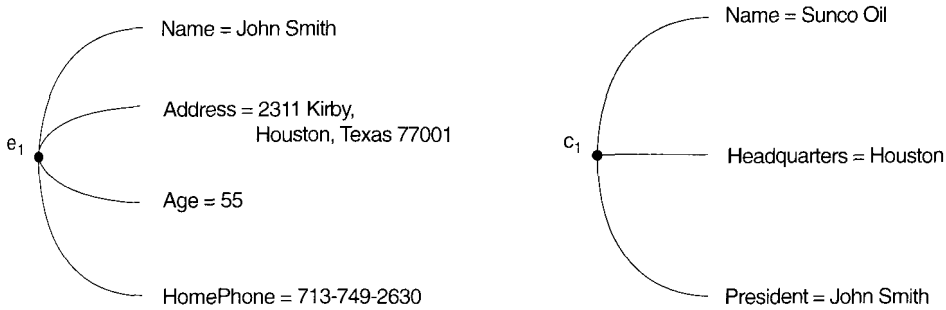

FIGURE 3.3 Two entities, employee e_1 and company c_1, and their attributes.

Several types of attributes occur in the ER model: *simple* versus *composite*, *single-valued* versus *multivalued*, and *stored* versus *derived*. We first define these attribute types and illustrate their use via examples. We then introduce the concept of a *null value* for an attribute.

Composite versus Simple (Atomic) Attributes. **Composite attributes** can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the employee entity shown in Figure 3.3 can be subdivided into StreetAddress, City, State, and Zip,[3] with the values "2311 Kirby," "Houston," "Texas," and "77001." Attributes that are not divisible are called **simple** or **atomic attributes.** Composite attributes can form a hierarchy; for example, StreetAddress can be further subdivided into three simple attributes: Number, Street, and ApartmentNumber, as shown in Figure 3.4. The value of a composite attribute is the concatenation of the values of its constituent simple attributes.

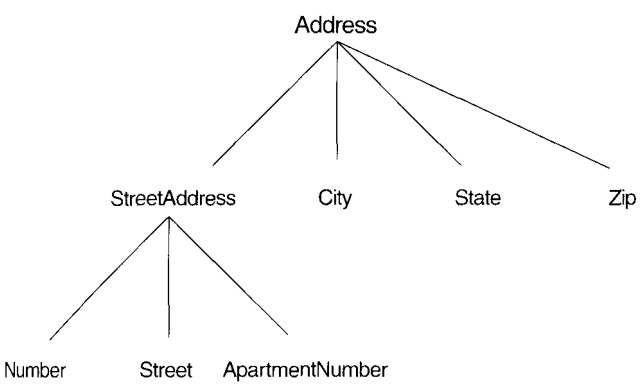

FIGURE 3.4 A hierarchy of composite attributes.

3. The zip code is the name used in the United States for a 5-digit postal code.

Composite attributes are useful to model situations in which a user sometimes refers to the composite attribute as a unit but at other times refers specifically to its components. If the composite attribute is referenced only as a whole, there is no need to subdivide it into component attributes. For example, if there is no need to refer to the individual components of an address (zip code, street, and so on), then the whole address can be designated as a simple attribute.

Single-Valued versus Multivalued Attributes. Most attributes have a single value for a particular entity; such attributes are called **single-valued.** For example, Age is a single-valued attribute of a person. In some cases an attribute can have a set of values for the same entity—for example, a Colors attribute for a car, or a CollegeDegrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two values for Colors. Similarly, one person may not have a college degree, another person may have one, and a third person may have two or more degrees; therefore, different persons can have different *numbers of values* for the CollegeDegrees attribute. Such attributes are called **multivalued.** A multivalued attribute may have lower and upper bounds to constrain the number of values allowed for each individual entity. For example, the Colors attribute of a car may have between one and three values, if we assume that a car can have at most three colors.

Stored versus Derived Attributes. In some cases, two (or more) attribute values are related—for example, the Age and BirthDate attributes of a person. For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's BirthDate. The Age attribute is hence called a **derived attribute** and is said to be **derivable from** the BirthDate attribute, which is called a **stored attribute.** Some attribute values can be derived from *related entities;* for example, an attribute NumberOfEmployees of a department entity can be derived by counting the number of employees related to (working for) that department.

Null Values. In some cases a particular entity may not have an applicable value for an attribute. For example, the ApartmentNumber attribute of an address applies only to addresses that are in apartment buildings and not to other types of residences, such as single-family homes. Similarly, a CollegeDegrees attribute applies only to persons with college degrees. For such situations, a special value called **null** is created. An address of a single-family home would have null for its ApartmentNumber attribute, and a person with no college degree would have null for CollegeDegrees. Null can also be used if we do not know the value of an attribute for a particular entity—for example, if we do not know the home phone of "John Smith" in Figure 3.3. The meaning of the former type of null is *not applicable*, whereas the meaning of the latter is *unknown*. The "unknown" category of null can be further classified into two cases. The first case arises when it is known that the attribute value exists but is *missing*—for example, if the Height attribute of a person is listed as null. The second case arises when it is *not known* whether the attribute value exists—for example, if the HomePhone attribute of a person is null.

Complex Attributes. Notice that composite and multivalued attributes can be nested in an arbitrary way. We can represent arbitrary nesting by grouping components of

```
{AddressPhone( {Phone(AreaCode,PhoneNumber)},
Address(StreetAddress(Number,Street,ApartmentNumber),
                City,State,Zip) ) }
```

FIGURE 3.5 A complex attribute: AddressPhone.

a composite attribute between parentheses () and separating the components with commas, and by displaying multivalued attributes between braces {}. Such attributes are called **complex attributes.** For example, if a person can have more than one residence and each residence can have multiple phones, an attribute AddressPhone for a person can be specified as shown in Figure 3.5.[4]

3.3.2 Entity Types, Entity Sets, Keys, and Value Sets

Entity Types and Entity Sets. A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of the employees. These employee entities share the same attributes, but each entity has *its own value(s)* for each attribute. An **entity type** defines a *collection* (or *set*) of entities that have the same attributes. Each entity type in the database is described by its name and attributes. Figure 3.6 shows two entity types, named EMPLOYEE and COMPANY, and a list of attributes for each. A few individual entities of each type are also illustrated, along with the values of their attributes. The collection of all entities of a particular entity type in the database at any point in time is called an **entity set;** the entity set is usually referred to using the same name as the entity type. For example, EMPLOYEE refers to both a *type of entity* as well as the current *set of all employee entities* in the database.

An entity type is represented in ER diagrams[5] (see Figure 3.2) as a rectangular box enclosing the entity type name. Attribute names are enclosed in ovals and are attached to their entity type by straight lines. Composite attributes are attached to their component attributes by straight lines. Multivalued attributes are displayed in double ovals.

An entity type describes the **schema** or **intension** for a *set of entities* that share the same structure. The collection of entities of a particular entity type are grouped into an entity set, which is also called the **extension** of the entity type.

Key Attributes of an Entity Type. An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes. An entity type usually has an attribute whose values are distinct for each individual entity in the entity set. Such an attribute is called a **key attribute,** and its values can be used to identify each entity

4. For those familiar with XML, we should note here that complex attributes are similar to complex elements in XML (see Chapter 26).

5. We are using a notation for ER diagrams that is close to the original proposed notation (Chen 1976). Unfortunately, many other notations are in use. We illustrate some of the other notations in Appendix A and later in this chapter when we present UML class diagrams.

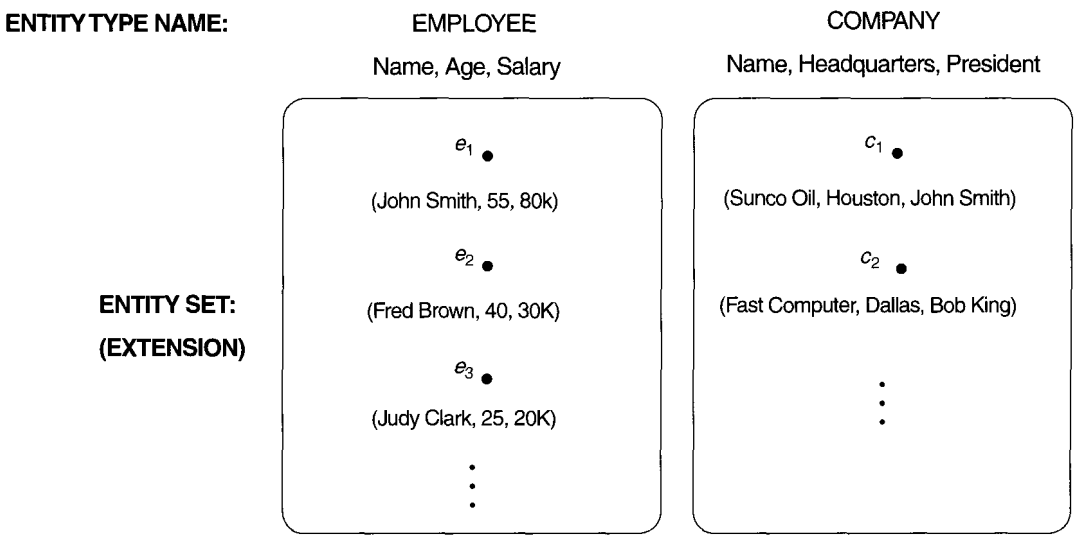

FIGURE 3.6 Two entity types, EMPLOYEE and COMPANY, and some member entities of each.

uniquely. For example, the Name attribute is a key of the COMPANY entity type in Figure 3.6, because no two companies are allowed to have the same name. For the PERSON entity type, a typical key attribute is SocialSecurityNumber. Sometimes, several attributes together form a key, meaning that the *combination* of the attribute values must be distinct for each entity. If a set of attributes possesses this property, the proper way to represent this in the ER model that we describe here is to define a *composite attribute* and designate it as a key attribute of the entity type. Notice that such a composite key must be *minimal;* that is, all component attributes must be included in the composite attribute to have the uniqueness property.[6] In ER diagrammatic notation, each key attribute has its name **underlined** inside the oval, as illustrated in Figure 3.2.

Specifying that an attribute is a key of an entity type means that the preceding uniqueness property must hold for *every entity set* of the entity type. Hence, it is a constraint that prohibits any two entities from having the same value for the key attribute at the same time. It is not the property of a particular extension; rather, it is a constraint on *all extensions* of the entity type. This key constraint (and other constraints we discuss later) is derived from the constraints of the miniworld that the database represents.

Some entity types have *more than one* key attribute. For example, each of the VehicleID and Registration attributes of the entity type CAR (Figure 3.7) is a key in its own right. The Registration attribute is an example of a composite key formed from two simple component attributes, RegistrationNumber and State, neither of which is a key on its own. An entity type may also have *no key,* in which case it is called a *weak entity type* (see Section 3.5).

6. Superfluous attributes must not be included in a key; however, a **superkey** may include superfluous attributes, as explained in Chapter 5.

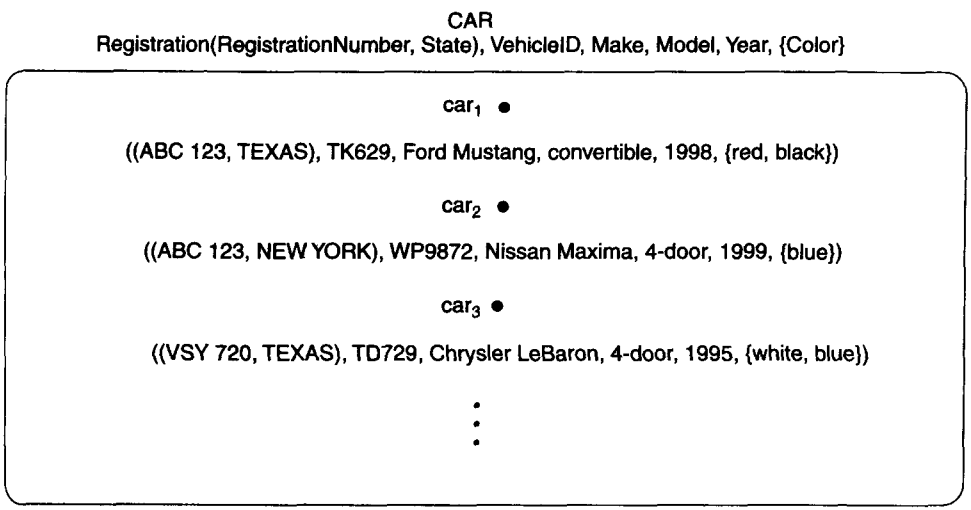

FIGURE 3.7 The CAR entity type with two key attributes, Registration and VehicleID.

Value Sets (Domains) of Attributes. Each simple attribute of an entity type is associated with a **value set** (or **domain** of values), which specifies the set of values that may be assigned to that attribute for each individual entity. In Figure 3.6, if the range of ages allowed for employees is between 16 and 70, we can specify the value set of the Age attribute of EMPLOYEE to be the set of integer numbers between 16 and 70. Similarly, we can specify the value set for the Name attribute as being the set of strings of alphabetic characters separated by blank characters, and so on. Value sets are not displayed in ER diagrams. Value sets are typically specified using the basic **data types** available in most programming languages, such as integer, string, boolean, float, enumerated type, subrange, and so on. Additional data types to represent date, time, and other concepts are also employed.

Mathematically, an attribute A of entity type E whose value set is V can be defined as a **function** from E to the power set[7] $P(V)$ of V:

$$A : E \rightarrow P(V)$$

We refer to the value of attribute A for entity e as $A(e)$. The previous definition covers both single-valued and multivalued attributes, as well as nulls. A null value is represented by the *empty set*. For single-valued attributes, $A(e)$ is restricted to being a *singleton set* for each entity e in E, whereas there is no restriction on multivalued attributes.[8] For a composite attribute A, the value set V is the Cartesian product of $P(V_1)$,

7. The **power set** $P(V)$ of a set V is the set of all subsets of V.

8. A **singleton set** is a set with only one element (value).

$P(V_2), \ldots, P(V_n)$, where $V_1, V_2, \ldots, V_n$ are the value sets of the simple component attributes that form A:

$$V = P(V_1) \times P(V_2) \times \ldots \times P(V_n)$$

3.3.3 Initial Conceptual Design of the COMPANY Database

We can now define the entity types for the COMPANY database, based on the requirements described in Section 3.2. After defining several entity types and their attributes here, we *refine* our design in Section 3.4 after we introduce the concept of a relationship. According to the requirements listed in Section 3.2, we can identify four entity types—one corresponding to each of the four items in the specification (see Figure 3.8):

1. An entity type DEPARTMENT with attributes Name, Number, Locations, Manager, and ManagerStartDate. Locations is the only multivalued attribute. We can specify that both Name and Number are (separate) key attributes, because each was specified to be unique.
2. An entity type PROJECT with attributes Name, Number, Location, and ControllingDepartment. Both Name and Number are (separate) key attributes.
3. An entity type EMPLOYEE with attributes Name, SSN (for social security number), Sex, Address, Salary, BirthDate, Department, and Supervisor. Both Name and Address may be composite attributes; however, this was not specified in the requirements. We must go back to the users to see if any of them will refer to the individual components of Name—FirstName, MiddleInitial, LastName—or of Address.
4. An entity type DEPENDENT with attributes Employee, DependentName, Sex, BirthDate, and Relationship (to the employee).

DEPARTMENT
Name, Number, {Locations}, Manager, ManagerStartDate

PROJECT
Name, Number, Location, ControllingDepartment

EMPLOYEE
Name (FName, MInit, LName), SSN, Sex, Address, Salary,
BirthDate, Department, Supervisor, {WorksOn (Project, Hours)}

DEPENDENT
Employee, DependentName, Sex, BirthDate, Relationship

FIGURE 3.8 Preliminary design of entity types for the COMPANY database.

So far, we have not represented the fact that an employee can work on several projects, nor have we represented the number of hours per week an employee works on each project. This characteristic is listed as part of requirement 3 in Section 3.2, and it can be represented by a multivalued composite attribute of EMPLOYEE called WorksOn with the simple components (Project, Hours). Alternatively, it can be represented as a multivalued composite attribute of PROJECT called Workers with the simple components (Employee, Hours). We choose the first alternative in Figure 3.8, which shows each of the entity types just described. The Name attribute of EMPLOYEE is shown as a composite attribute, presumably after consultation with the users.

3.4 Relationship Types, Relationship Sets, Roles, and Structural Constraints

In Figure 3.8 there are several *implicit relationships* among the various entity types. In fact, whenever an attribute of one entity type refers to another entity type, some relationship exists. For example, the attribute Manager of DEPARTMENT refers to an employee who manages the department; the attribute ControllingDepartment of PROJECT refers to the department that controls the project; the attribute Supervisor of EMPLOYEE refers to another employee (the one who supervises this employee); the attribute Department of EMPLOYEE refers to the department for which the employee works; and so on. In the ER model, these references should not be represented as attributes but as **relationships,** which are discussed in this section. The COMPANY database schema will be refined in Section 3.6 to represent relationships explicitly. In the initial design of entity types, relationships are typically captured in the form of attributes. As the design is refined, these attributes get converted into relationships between entity types.

This section is organized as follows. Section 3.4.1 introduces the concepts of relationship types, relationship sets, and relationship instances. We then define the concepts of relationship degree, role names, and recursive relationships in Section 3.4.2, and discuss structural constraints on relationships—such as cardinality ratios and existence dependencies—in Section 3.4.3. Section 3.4.4 shows how relationship types can also have attributes.

3.4.1 Relationship Types, Sets, and Instances

A **relationship type** R among n entity types $E_1, E_2, \ldots, E_n$ defines a set of associations—or a **relationship set**—among entities from these entity types. As for the case of entity types and entity sets, a relationship type and its corresponding relationship set are customarily referred to by the *same name*, R. Mathematically, the relationship set R is a set of **relationship instances** r_i, where each r_i associates n individual entities $(e_1, e_2, \ldots, e_n)$, and each entity e_j in r_i is a member of entity type E_j, $1 \leq j \leq n$. Hence, a relationship type is a mathematical relation on $E_1, E_2, \ldots, E_n$; alternatively, it can be defined as a subset of the Cartesian product $E_1 \times E_2 \times \ldots \times E_n$. Each of the entity types $E_1, E_2, \ldots, E_n$ is said to

participate in the relationship type R; similarly, each of the individual entities $e_1, e_2, \ldots, e_n$ is said to participate in the relationship instance $r_i = (e_1, e_2, \ldots, e_n)$.

Informally, each relationship instance r_i in R is an association of entities, where the association includes exactly one entity from each participating entity type. Each such relationship instance r_i represents the fact that the entities participating in r_i are related in some way in the corresponding miniworld situation. For example, consider a relationship type WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department for which the employee works. Each relationship instance in the relationship set WORKS_FOR associates one employee entity and one department entity. Figure 3.9 illustrates this example, where each relationship instance r_i is shown connected to the employee and department entities that participate in r_i. In the miniworld represented by Figure 3.9, employees e_1, e_3, and e_6 work for department d_1; e_2 and e_4 work for d_2; and e_5 and e_7 work for d_3.

In ER diagrams, relationship types are displayed as diamond-shaped boxes, which are connected by straight lines to the rectangular boxes representing the participating entity types. The relationship name is displayed in the diamond-shaped box (see Figure 3.2).

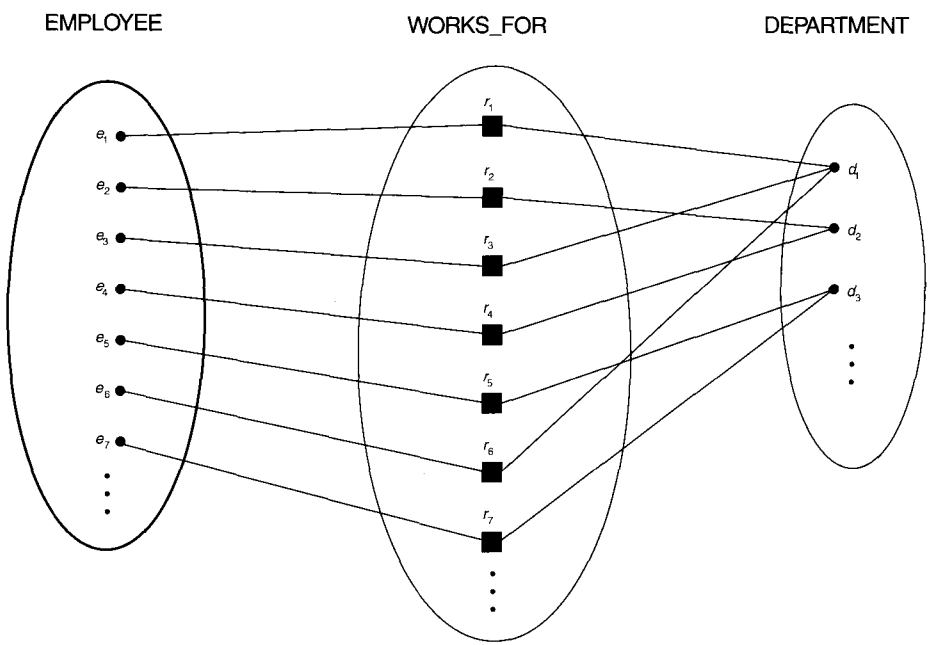

FIGURE 3.9 Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.

3.4.2 Relationship Degree, Role Names, and Recursive Relationships

Degree of a Relationship Type. The **degree** of a relationship type is the number of participating entity types. Hence, the WORKS_FOR relationship is of degree two. A relationship type of degree two is called **binary,** and one of degree three is called **ternary.** An example of a ternary relationship is SUPPLY, shown in Figure 3.10, where each relationship instance r_i associates three entities—a supplier s, a part p, and a project j—whenever s supplies part p to project j. Relationships can generally be of any degree, but the ones most common are binary relationships. Higher-degree relationships are generally more complex than binary relationships; we characterize them further in Section 4.7.

Relationships as Attributes. It is sometimes convenient to think of a relationship type in terms of attributes, as we discussed in Section 3.3.3. Consider the WORKS_FOR relationship type of Figure 3.9. One can think of an attribute called Department of the EMPLOYEE entity type whose value for each employee entity is (a reference to) the *department entity* that the employee works for. Hence, the value set for this Department attribute is the *set of all* DEPARTMENT *entities*, which is the DEPARTMENT entity set. This is what we did in Figure 3.8 when we specified the initial design of the entity type EMPLOYEE for the COMPANY database. However, when we think of a binary relationship as an attribute, we always have two

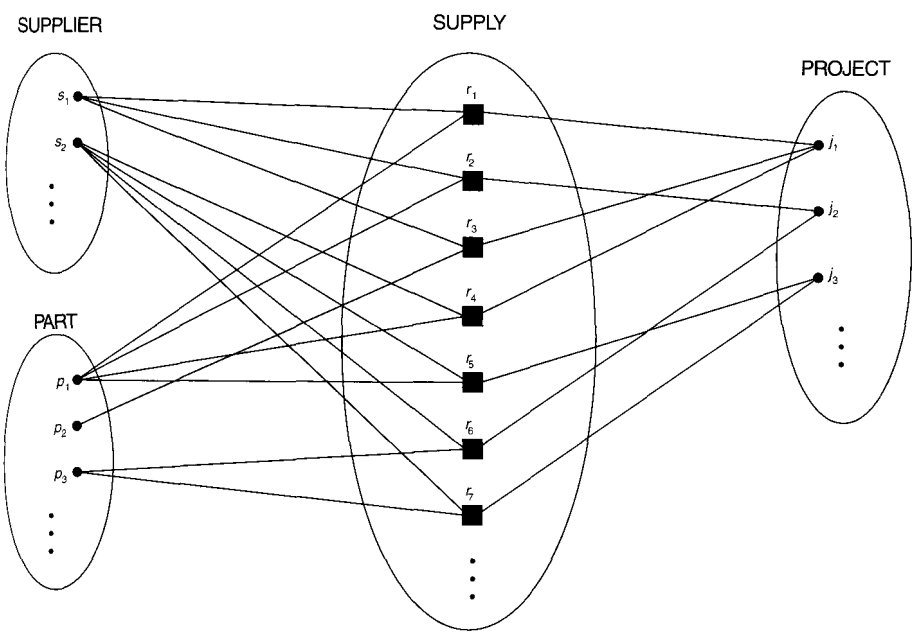

FIGURE 3.10 Some relationship instances in the SUPPLY ternary relationship set.

options. In this example, the alternative is to think of a multivalued attribute Employees of the entity type DEPARTMENT whose values for each department entity is the *set of employee entities* who work for that department. The value set of this Employees attribute is the power set of the EMPLOYEE entity set. Either of these two attributes—Department of EMPLOYEE or Employees of DEPARTMENT—can represent the WORKS_FOR relationship type. If both are represented, they are constrained to be inverses of each other.[9]

Role Names and Recursive Relationships. Each entity type that participates in a relationship type plays a particular **role** in the relationship. The **role name** signifies the role that a participating entity from the entity type plays in each relationship instance, and helps to explain what the relationship means. For example, in the WORKS_FOR relationship type, EMPLOYEE plays the role of *employee* or *worker* and DEPARTMENT plays the role of *department* or *employer*.

Role names are not technically necessary in relationship types where all the participating entity types are distinct, since each participating entity type name can be used as the role name. However, in some cases the *same* entity type participates more than once in a relationship type in *different roles*. In such cases the role name becomes essential for distinguishing the meaning of each participation. Such relationship types are called **recursive relationships.** Figure 3.11 shows an example. The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity type. Hence, the EMPLOYEE entity type *participates twice* in SUPERVISION: once in the role of *supervisor* (or *boss*), and once in the role of *supervisee* (or *subordinate*). Each relationship instance r_i in SUPERVISION associates two employee entities e_j and e_k, one of which plays the role of supervisor and the other the role of supervisee. In Figure 3.11, the lines marked "1" represent the supervisor role, and those marked "2" represent the supervisee role; hence, e_1 supervises e_2 and e_3, e_4 supervises e_6 and e_7, and e_5 supervises e_1 and e_4.

3.4.3 Constraints on Relationship Types

Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. These constraints are determined from the miniworld situation that the relationships represent. For example, in Figure 3.9, if the company has a rule that each employee must work for exactly one department, then we would like to describe this constraint in the schema. We can distinguish two main types of relationship constraints: *cardinality ratio* and *participation*.

9. This concept of representing relationship types as attributes is used in a class of data models called **functional data models.** In object databases (see Chapter 20), relationships can be represented by reference attributes, either in one direction or in both directions as inverses. In relational databases (see Chapter 5), foreign keys are a type of reference attribute used to represent relationships.

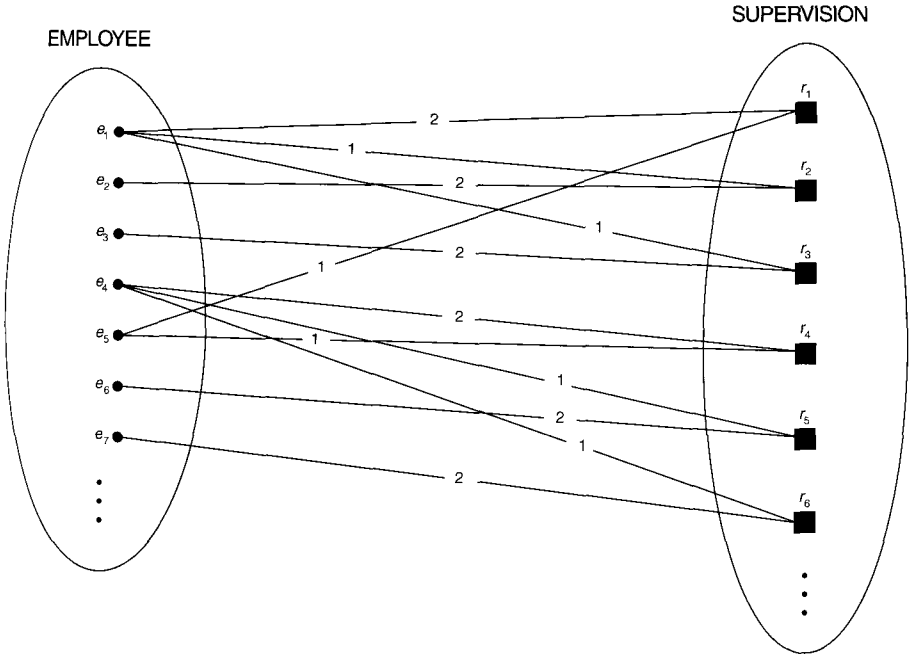

FIGURE 3.11 A recursive relationship SUPERVISION between EMPLOYEE in the *supervisor* role (1) and EMPLOYEE in the *subordinate* role (2).

Cardinality Ratios for Binary Relationships. The **cardinality ratio** for a binary relationship specifies the *maximum* number of relationship instances that an entity can participate in. For example, in the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEE is of cardinality ratio 1:N, meaning that each department can be related to (that is, employs) any number of employees,[10] but an employee can be related to (work for) only one department. The possible cardinality ratios for binary relationship types are 1:1, 1:N, N:1, and M:N.

An example of a 1:1 binary relationship is MANAGES (Figure 3.12), which relates a department entity to the employee who manages that department. This represents the miniworld constraints that—at any point in time—an employee can manage only one department and a department has only one manager. The relationship type WORKS_ON (Figure 3.13) is of cardinality ratio M:N, because the miniworld rule is that an employee can work on several projects and a project can have several employees.

Cardinality ratios for binary relationships are represented on ER diagrams by displaying 1, M, and N on the diamonds as shown in Figure 3.2.

10. N stands for *any number* of related entities (zero or more).

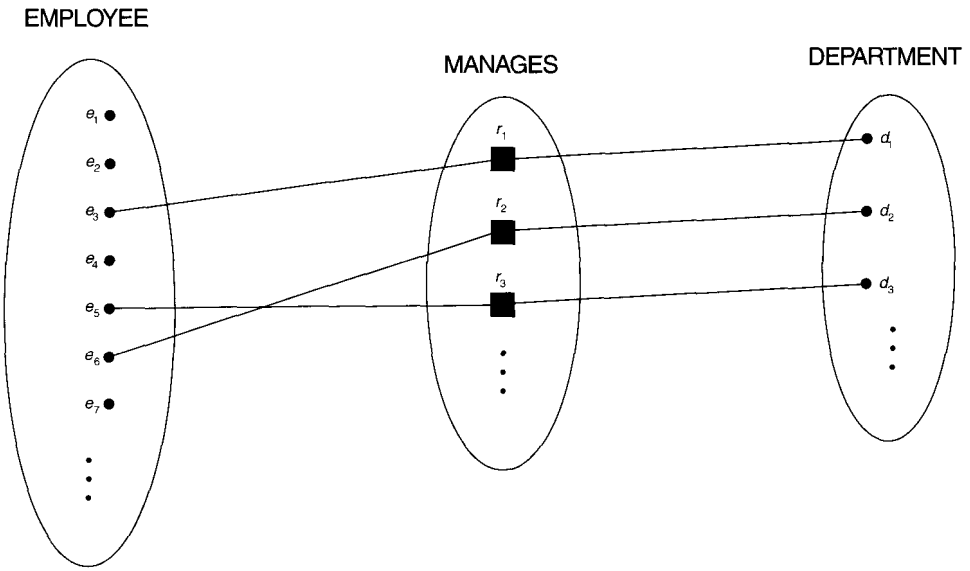

FIGURE 3.12 A 1:1 relationship, MANAGES.

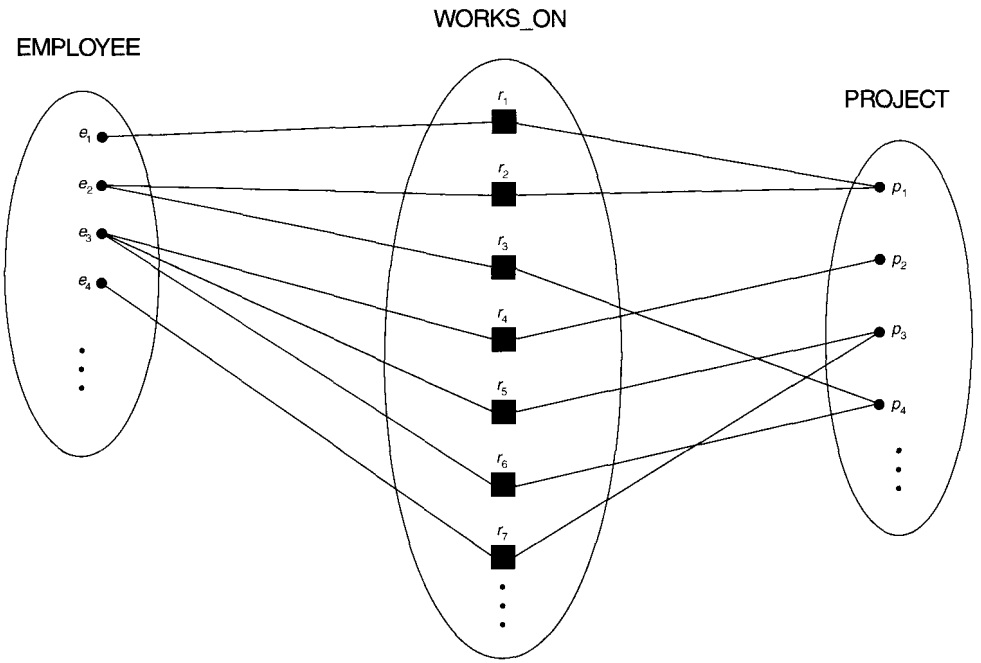

FIGURE 3.13 An M:N relationship, WORKS_ON.

Participation Constraints and Existence Dependencies. The **participation constraint** specifies whether the existence of an entity depends on its being related to another entity via the relationship type. This constraint specifies the *minimum* number of relationship instances that each entity can participate in, and is sometimes called the **minimum cardinality constraint.** There are two types of participation constraints—total and partial—which we illustrate by example. If a company policy states that *every* employee must work for a department, then an employee entity can exist only if it participates in at least one WORKS_FOR relationship instance (Figure 3.9). Thus, the participation of EMPLOYEE in WORKS_FOR is called **total participation,** meaning that every entity in "the total set" of employee entities must be related to a department entity via WORKS_FOR. Total participation is also called **existence dependency.** In Figure 3.12 we do not expect every employee to manage a department, so the participation of EMPLOYEE in the MANAGES relationship type is **partial,** meaning that *some* or "part of the set of" employee entities are related to some department entity via MANAGES, but not necessarily all. We will refer to the cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.

In ER diagrams, total participation (or existence dependency) is displayed as a *double line* connecting the participating entity type to the relationship, whereas partial participation is represented by a *single line* (see Figure 3.2).

3.4.4 Attributes of Relationship Types

Relationship types can also have attributes, similar to those of entity types. For example, to record the number of hours per week that an employee works on a particular project, we can include an attribute Hours for the WORKS_ON relationship type of Figure 3.13. Another example is to include the date on which a manager started managing a department via an attribute StartDate for the MANAGES relationship type of Figure 3.12.

Notice that attributes of 1:1 or 1:N relationship types can be migrated to one of the participating entity types. For example, the StartDate attribute for the MANAGES relationship can be an attribute of either EMPLOYEE or DEPARTMENT, although conceptually it belongs to MANAGES. This is because MANAGES is a 1:1 relationship, so every department or employee entity participates in *at most one* relationship instance. Hence, the value of the StartDate attribute can be determined separately, either by the participating department entity or by the participating employee (manager) entity.

For a 1:N relationship type, a relationship attribute can be migrated *only* to the entity type on the N-side of the relationship. For example, in Figure 3.9, if the WORKS_FOR relationship also has an attribute StartDate that indicates when an employee started working for a department, this attribute can be included as an attribute of EMPLOYEE. This is because each employee works for only one department, and hence participates in at most one relationship instance in WORKS_FOR. In both 1:1 and 1:N relationship types, the decision as to where a relationship attribute should be placed—as a relationship type attribute or as an attribute of a participating entity type—is determined subjectively by the schema designer.

For M:N relationship types, some attributes may be determined by the *combination of participating entities* in a relationship instance, not by any single entity. Such attributes

must be specified as relationship attributes. An example is the Hours attribute of the M:N relationship WORKS_ON (Figure 3.13); the number of hours an employee works on a project is determined by an employee-project combination and not separately by either entity.

3.5 Weak Entity Types

Entity types that do not have key attributes of their own are called **weak entity types.** In contrast, **regular entity types** that do have a key attribute—which include all the examples we discussed so far—are called **strong entity types.** Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with one of their attribute values. We call this other entity type the **identifying** or **owner entity type,**[11] and we call the relationship type that relates a weak entity type to its owner the **identifying relationship** of the weak entity type.[12] A weak entity type always has a *total participation constraint* (existence dependency) with respect to its identifying relationship, because a weak entity cannot be identified without an owner entity. However, not every existence dependency results in a weak entity type. For example, a DRIVER_LICENSE entity cannot exist unless it is related to a PERSON entity, even though it has its own key (LicenseNumber) and hence is not a weak entity.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship (Figure 3.2). The attributes of DEPENDENT are Name (the first name of the dependent), BirthDate, Sex, and Relationship (to the employee). Two dependents *of two distinct employees* may, by chance, have the same values for Name, BirthDate, Sex, and Relationship, but they are still distinct entities. They are identified as distinct entities only after determining the *particular employee entity* to which each dependent is related. Each employee entity is said to **own** the dependent entities that are related to it.

A weak entity type normally has a **partial key,** which is the set of attributes that can uniquely identify weak entities that are *related to the same owner entity*.[13] In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute Name of DEPENDENT is the partial key. In the worst case, a composite attribute of *all the weak entity's attributes* will be the partial key.

In ER diagrams, both a weak entity type and its identifying relationship are distinguished by surrounding their boxes and diamonds with double lines (see Figure 3.2). The partial key attribute is underlined with a dashed or dotted line.

Weak entity types can sometimes be represented as complex (composite, multivalued) attributes. In the preceding example, we could specify a multivalued attribute Dependents for EMPLOYEE, which is a composite attribute with component attributes Name, BirthDate,

11. The identifying entity type is also sometimes called the **parent entity type** or the **dominant entity type.**

12. The weak entity type is also sometimes called the **child entity type** or the **subordinate entity type.**

13. The partial key is sometimes called the **discriminator.**

Sex, and Relationship. The choice of which representation to use is made by the database designer. One criterion that may be used is to choose the weak entity type representation if there are many attributes. If the weak entity participates independently in relationship types other than its identifying relationship type, then it should *not* be modeled as a complex attribute.

In general, any number of levels of weak entity types can be defined; an owner entity type may itself be a weak entity type. In addition, a weak entity type may have more than one identifying entity type and an identifying relationship type of degree higher than two, as we illustrate in Section 4.7.

3.6 Refining the ER Design for the COMPANY Database

We can now refine the database design of Figure 3.8 by changing the attributes that represent relationships into relationship types. The cardinality ratio and participation constraint of each relationship type are determined from the requirements listed in Section 3.2. If some cardinality ratio or dependency cannot be determined from the requirements, the users must be questioned further to determine these structural constraints.

In our example, we specify the following relationship types:

1. MANAGES, a 1:1 relationship type between EMPLOYEE and DEPARTMENT. EMPLOYEE participation is partial. DEPARTMENT participation is not clear from the requirements. We question the users, who say that a department must have a manager at all times, which implies total participation.[14] The attribute StartDate is assigned to this relationship type.
2. WORKS_FOR, a 1:N relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.
3. CONTROLS, a 1:N relationship type between DEPARTMENT and PROJECT. The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial, after consultation with the users indicates that some departments may control no projects.
4. SUPERVISION, a 1:N relationship type between EMPLOYEE (in the supervisor role) and EMPLOYEE (in the supervisee role). Both participations are determined to be partial, after the users indicate that not every employee is a supervisor and not every employee has a supervisor.
5. WORKS_ON, determined to be an M:N relationship type with attribute Hours, after the users indicate that a project can have several employees working on it. Both participations are determined to be total.

14. The rules in the miniworld that determine the constraints are sometimes called the *business rules*, since they are determined by the "business" or organization that will utilize the database.

6. DEPENDENTS_OF, a 1:N relationship type between EMPLOYEE and DEPENDENT, which is also the identifying relationship for the weak entity type DEPENDENT. The participation of EMPLOYEE is partial, whereas that of DEPENDENT is total.

After specifying the above six relationship types, we remove from the entity types in Figure 3.8 all attributes that have been refined into relationships. These include Manager and ManagerStartDate from DEPARTMENT; ControllingDepartment from PROJECT; Department, Supervisor, and WorksOn from EMPLOYEE; and Employee from DEPENDENT. It is important to have the least possible redundancy when we design the conceptual schema of a database. If some redundancy is desired at the storage level or at the user view level, it can be introduced later, as discussed in Section 1.6.1.

3.7 ER DIAGRAMS, NAMING CONVENTIONS, AND DESIGN ISSUES

3.7.1 Summary of Notation for ER Diagrams

Figures 3.9 through 3.13 illustrate examples of the participation of entity types in relationship types by displaying their extensions—the individual entity instances and relationship instances in the entity sets and relationship sets. In ER diagrams the emphasis is on representing the schemas rather than the instances. This is more useful in database design because a database schema changes rarely, whereas the contents of the entity sets change frequently. In addition, the schema is usually easier to display than the extension of a database, because it is much smaller.

Figure 3.2 displays the COMPANY **ER database schema** as an **ER diagram.** We now review the full ER diagram notation. Entity types such as EMPLOYEE, DEPARTMENT, and PROJECT are shown in rectangular boxes. Relationship types such as WORKS_FOR, MANAGES, CONTROLS, and WORKS_ON are shown in diamond-shaped boxes attached to the participating entity types with straight lines. Attributes are shown in ovals, and each attribute is attached by a straight line to its entity type or relationship type. Component attributes of a composite attribute are attached to the oval representing the composite attribute, as illustrated by the Name attribute of EMPLOYEE. Multivalued attributes are shown in double ovals, as illustrated by the Locations attribute of DEPARTMENT. Key attributes have their names underlined. Derived attributes are shown in dotted ovals, as illustrated by the NumberOfEmployees attribute of DEPARTMENT.

Weak entity types are distinguished by being placed in double rectangles and by having their identifying relationship placed in double diamonds, as illustrated by the DEPENDENT entity type and the DEPENDENTS_OF identifying relationship type. The partial key of the weak entity type is underlined with a dotted line.

In Figure 3.2 the cardinality ratio of each *binary* relationship type is specified by attaching a 1, M, or N on each participating edge. The cardinality ratio of DEPARTMENT : EMPLOYEE in MANAGES is 1:1, whereas it is 1:N for DEPARTMENT : EMPLOYEE in WORKS_FOR, and M:N for WORKS_ON. The

participation constraint is specified by a single line for partial participation and by double lines for total participation (existence dependency).

In Figure 3.2 we show the role names for the SUPERVISION relationship type because the EMPLOYEE entity type plays both roles in that relationship. Notice that the cardinality is 1:N from supervisor to supervisee because each employee in the role of supervisee has at most one direct supervisor, whereas an employee in the role of supervisor can supervise zero or more employees.

Figure 3.14 summarizes the conventions for ER diagrams.

3.7.2 Proper Naming of Schema Constructs

When designing a database schema, the choice of names for entity types, attributes, relationship types, and (particularly) roles is not always straightforward. One should choose names that convey, as much as possible, the meanings attached to the different constructs in the schema. We choose to use *singular names* for entity types, rather than plural ones, because the entity type name applies to each individual entity belonging to that entity type. In our ER diagrams, we will use the convention that entity type and relationship type names are in uppercase letters, attribute names are capitalized, and role names are in lowercase letters. We have already used this convention in Figure 3.2.

As a general practice, given a narrative description of the database requirements, the *nouns* appearing in the narrative tend to give rise to entity type names, and the *verbs* tend to indicate names of relationship types. Attribute names generally arise from additional nouns that describe the nouns corresponding to entity types.

Another naming consideration involves choosing binary relationship names to make the ER diagram of the schema readable from left to right and from top to bottom. We have generally followed this guideline in Figure 3.2. To explain this naming convention further, we have one exception to the convention in Figure 3.2—the DEPENDENTS_OF relationship type, which reads from bottom to top. When we describe this relationship, we can say that the DEPENDENT entities (bottom entity type) are DEPENDENTS_OF (relationship name) an EMPLOYEE (top entity type). To change this to read from top to bottom, we could rename the relationship type to HAS_DEPENDENTS, which would then read as follows: An EMPLOYEE entity (top entity type) HAS_DEPENDENTS (relationship name) of type DEPENDENT (bottom entity type). Notice that this issue arises because each binary relationship can be described starting from either of the two participating entity types, as discussed in the beginning of Section 3.4.

3.7.3 Design Choices for ER Conceptual Design

It is occasionally difficult to decide whether a particular concept in the miniworld should be modeled as an entity type, an attribute, or a relationship type. In this section, we give some brief guidelines as to which construct should be chosen in particular situations.

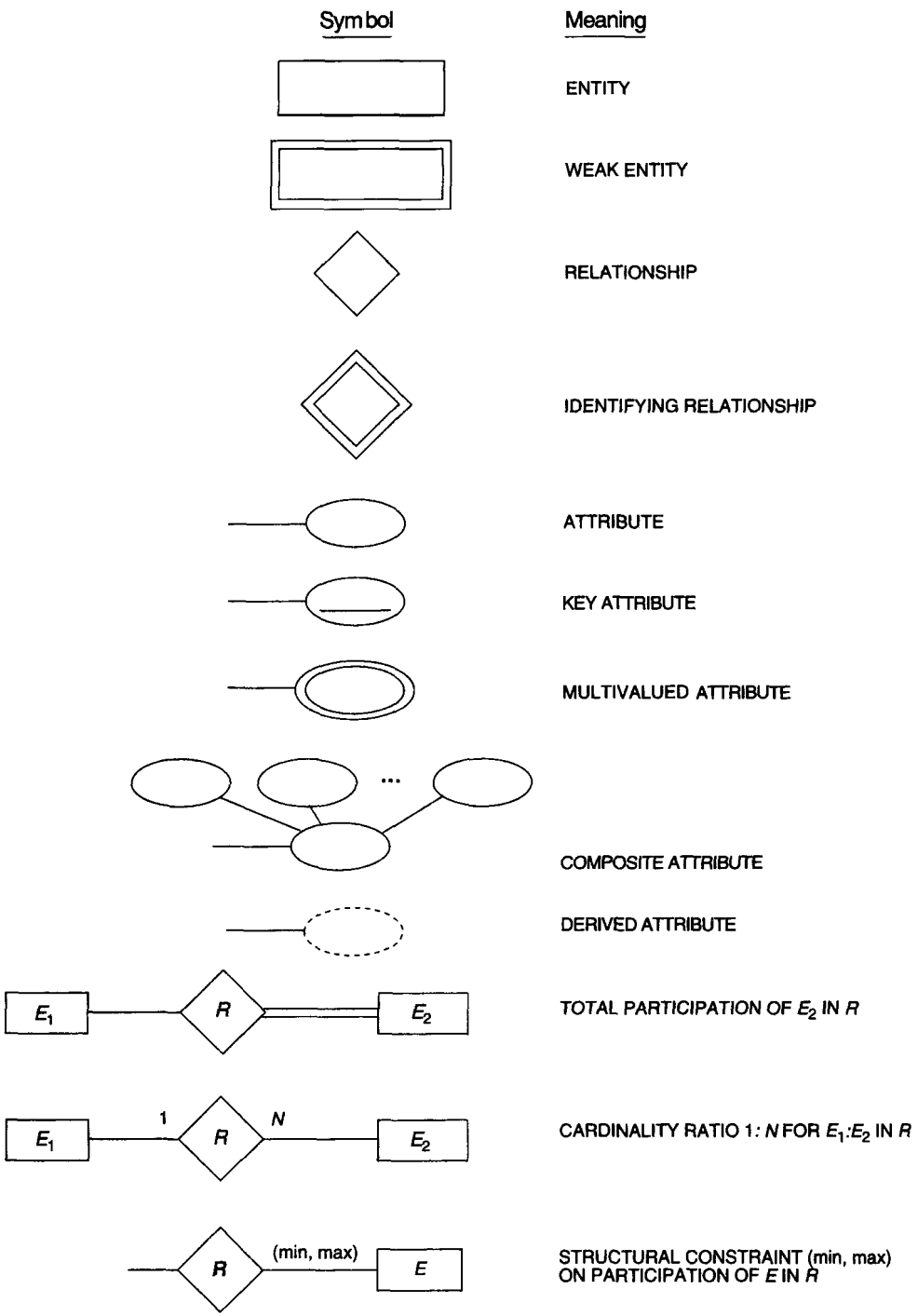

FIGURE 3.14 Summary of the notation for ER diagrams.

In general, the schema design process should be considered an iterative refinement process, where an initial design is created and then iteratively refined until the most suitable design is reached. Some of the refinements that are often used include the following:

- A concept may be first modeled as an attribute and then refined into a relationship because it is determined that the attribute is a reference to another entity type. It is often the case that a pair of such attributes that are inverses of one another are refined into a binary relationship. We discussed this type of refinement in detail in Section 3.6.
- Similarly, an attribute that exists in several entity types may be elevated or promoted to an independent entity type. For example, suppose that several entity types in a UNIVERSITY database, such as STUDENT, INSTRUCTOR, and COURSE, each has an attribute Department in the initial design; the designer may then choose to create an entity type DEPARTMENT with a single attribute DeptName and relate it to the three entity types (STUDENT, INSTRUCTOR, and COURSE) via appropriate relationships. Other attributes/relationships of DEPARTMENT may be discovered later.
- An inverse refinement to the previous case may be applied—for example, if an entity type DEPARTMENT exists in the initial design with a single attribute DeptName and is related to only one other entity type, STUDENT. In this case, DEPARTMENT may be reduced or demoted to an attribute of STUDENT.
- In Chapter 4, we discuss other refinements concerning specialization/generalization and relationships of higher degree. Chapter 12 discusses additional top-down and bottom-up refinements that are common in large-scale conceptual schema design.

3.7.4 Alternative Notations for ER Diagrams

There are many alternative diagrammatic notations for displaying ER diagrams. Appendix A gives some of the more popular notations. In Section 3.8, we introduce the Universal Modeling Language (UML) notation for class diagrams, which has been proposed as a standard for conceptual object modeling.

In this section, we describe one alternative ER notation for specifying structural constraints on relationships. This notation involves associating a pair of integer numbers (min, max) with each *participation* of an entity type E in a relationship type R, where $0 \leq \text{min} \leq \text{max}$ and $\text{max} \geq 1$. The numbers mean that for each entity e in E, e must participate in at least min and at most max relationship instances in R *at any point in time*. In this method, $\text{min} = 0$ implies partial participation, whereas $\text{min} > 0$ implies total participation.

Figure 3.15 displays the COMPANY database schema using the (min, max) notation.[15] Usually, one uses either the cardinality ratio/single-line/double-line notation *or* the (min,

15. In some notations, particularly those used in object modeling methodologies such as UML, the (min, max) is placed on the *opposite sides* to the ones we have shown. For example, for the WORKS_FOR relationship in Figure 3.15, the (1,1) would be on the DEPARTMENT side, and the (4,N) would be on the EMPLOYEE side. Here we used the original notation from Abrial (1974).

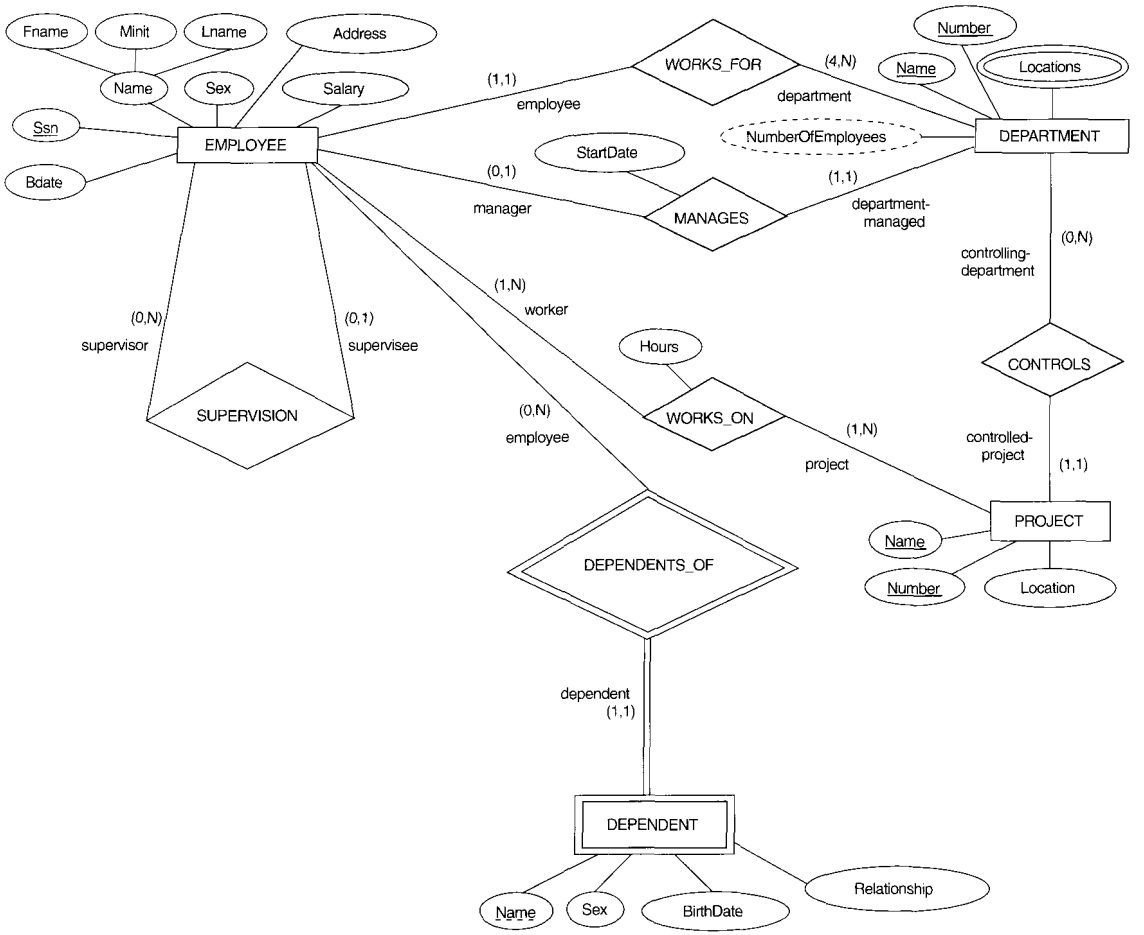

FIGURE 3.15 ER diagrams for the COMPANY schema, with structural constraints specified using (min, max) notation.

max) notation. The (min, max) notation is more precise, and we can use it easily to specify structural constraints for relationship types of *any degree*. However, it is not sufficient for specifying some key constraints on higher-degree relationships, as discussed in Section 4.7.

Figure 3.15 also displays all the role names for the COMPANY database schema.

3.8 NOTATION FOR UML CLASS DIAGRAMS

The UML methodology is being used extensively in software design and has many types of diagrams for various software design purposes. We only briefly present the basics of UML

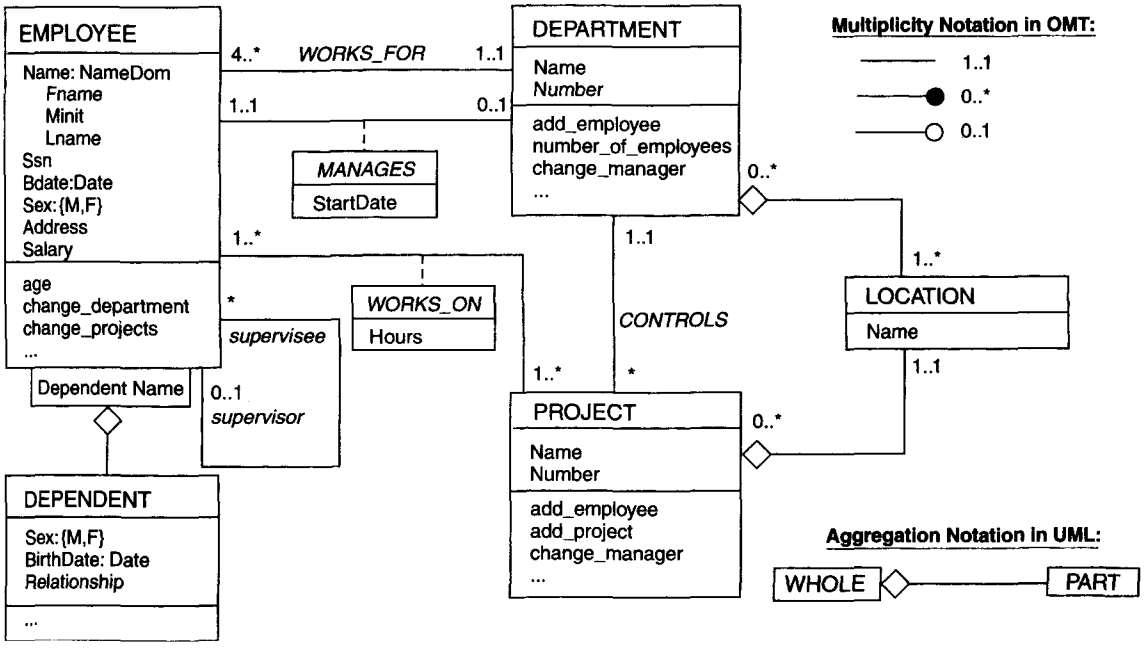

FIGURE 3.16 The COMPANY conceptual schema in UML class diagram notation.

class diagrams here, and compare them with ER diagrams. In some ways, class diagrams can be considered as an alternative notation to ER diagrams. Additional UML notation and concepts are presented in Section 4.6, and in Chapter 12. Figure 3.16 shows how the COMPANY ER database schema of Figure 3.15 can be displayed using UML class diagram notation. The *entity types* in Figure 3.15 are modeled as *classes* in Figure 3.16. An *entity* in ER corresponds to an *object* in UML.

In UML class diagrams, a **class** is displayed as a box (see Figure 3.16) that includes three sections: The top section gives the **class name,** the middle section includes the **attributes** for individual objects of the class; and the last section includes **operations** that can be applied to these objects. Operations are *not* specified in ER diagrams. Consider the EMPLOYEE class in Figure 3.16. Its attributes are Name, Ssn, Bdate, Sex, Address, and Salary. The designer can optionally specify the **domain** of an attribute if desired, by placing a colon (:) followed by the domain name or description, as illustrated by the Name, Sex, and Bdate attributes of EMPLOYEE in Figure 3.16. A composite attribute is modeled as a **structured domain,** as illustrated by the Name attribute of EMPLOYEE. A multivalued attribute will generally be modeled as a separate class, as illustrated by the LOCATION class in Figure 3.16.

Relationship types are called **associations** in UML terminology, and relationship instances are called **links.** A **binary association** (binary relationship type) is represented as a line connecting the participating classes (entity types), and may optionally have a

name. A relationship attribute, called a **link attribute,** is placed in a box that is connected to the association's line by a dashed line. The (min, max) notation described in Section 3.7.4 is used to specify relationship constraints, which are called **multiplicities** in UML terminology. Multiplicities are specified in the form *min..max*, and an asterisk (*) indicates no maximum limit on participation. However, the multiplicities are placed *on the opposite ends of the relationship* when compared with the notation discussed in Section 3.7.4 (compare Figures 3.16 and 3.15). In UML, a single asterisk indicates a multiplicity of 0..*, and a single 1 indicates a multiplicity of 1..1. A recursive relationship (see Section 3.4.2) is called a **reflexive association** in UML, and the role names—like the multiplicities—are placed at the opposite ends of an association when compared with the placing of role names in Figure 3.15.

In UML, there are two types of relationships: association and aggregation. **Aggregation** is meant to represent a relationship between a whole object and its component parts, and it has a distinct diagrammatic notation. In Figure 3.16, we modeled the locations of a department and the single location of a project as aggregations. However, aggregation and association do not have different structural properties, and the choice as to which type of relationship to use is somewhat subjective. In the ER model, both are represented as relationships.

UML also distinguishes between **unidirectional** and **bidirectional** associations (or aggregations). In the unidirectional case, the line connecting the classes is displayed with an arrow to indicate that only one direction for accessing related objects is needed. If no arrow is displayed, the bidirectional case is assumed, which is the default. For example, if we always expect to access the manager of a department starting from a DEPARTMENT object, we would draw the association line representing the MANAGES association with an arrow from DEPARTMENT to EMPLOYEE. In addition, relationship instances may be specified to be **ordered.** For example, we could specify that the employee objects related to each department through the WORKS_FOR association (relationship) should be ordered by their Bdate attribute value. Association (relationship) names are *optional* in UML, and relationship attributes are displayed in a box attached with a dashed line to the line representing the association/aggregation (see StartDate and Hours in Figure 3.16).

The operations given in each class are derived from the functional requirements of the application, as we discussed in Section 3.1. It is generally sufficient to specify the operation names initially for the logical operations that are expected to be applied to individual objects of a class, as shown in Figure 3.16. As the design is refined, more details are added, such as the exact argument types (parameters) for each operation, plus a functional description of each operation. UML has *function descriptions* and *sequence diagrams* to specify some of the operation details, but these are beyond the scope of our discussion. Chapter 12 will introduce some of these diagrams.

Weak entities can be modeled using the construct called **qualified association** (or **qualified aggregation**) in UML; this can represent both the identifying relationship and the partial key, which is placed in a box attached to the owner class. This is illustrated by the DEPENDENT class and its qualified aggregation to EMPLOYEE in Figure 3.16. The partial key DependentName is called the **discriminator** in UML terminology, since its value distinguishes the objects associated with (related to) the same EMPLOYEE. Qualified associations are not restricted to modeling weak entities, and they can be used to model other situations in UML.

3.9 SUMMARY

In this chapter we presented the modeling concepts of a high-level conceptual data model, the Entity-Relationship (ER) model. We started by discussing the role that a high-level data model plays in the database design process, and then we presented an example set of database requirements for the COMPANY database, which is one of the examples that is used throughout this book. We then defined the basic ER model concepts of entities and their attributes. We discussed null values and presented the various types of attributes, which can be nested arbitrarily to produce complex attributes:

- Simple or atomic
- Composite
- Multivalued

We also briefly discussed stored versus derived attributes. We then discussed the ER model concepts at the schema or "intension" level:

- Entity types and their corresponding entity sets
- Key attributes of entity types
- Value sets (domains) of attributes
- Relationship types and their corresponding relationship sets
- Participation roles of entity types in relationship types

We presented two methods for specifying the structural constraints on relationship types. The first method distinguished two types of structural constraints:

- Cardinality ratios (1:1, 1:N, M:N for binary relationships)
- Participation constraints (total, partial)

We noted that, alternatively, another method of specifying structural constraints is to specify minimum and maximum numbers (min, max) on the participation of each entity type in a relationship type. We discussed weak entity types and the related concepts of owner entity types, identifying relationship types, and partial key attributes.

Entity-Relationship schemas can be represented diagrammatically as ER diagrams. We showed how to design an ER schema for the COMPANY database by first defining the entity types and their attributes and then refining the design to include relationship types. We displayed the ER diagram for the COMPANY database schema. Finally, we discussed some of the basic concepts of UML class diagrams and how they relate to ER model concepts.

The ER modeling concepts we have presented thus far—entity types, relationship types, attributes, keys, and structural constraints—can model traditional business data-processing database applications. However, many newer, more complex applications—such as engineering design, medical information systems, or telecommunications—require additional concepts if we want to model them with greater accuracy. We discuss these advanced modeling concepts in Chapter 4. We also describe ternary and higher-degree relationship types in more detail in Chapter 4, and discuss the circumstances under which they are distinguished from binary relationships.

Review Questions

3.1. Discuss the role of a high-level data model in the database design process.
3.2. List the various cases where use of a null value would be appropriate.
3.3. Define the following terms: *entity*, *attribute*, *attribute value*, *relationship instance*, *composite attribute*, *multivalued attribute*, *derived attribute*, *complex attribute*, *key attribute*, *value set (domain)*.
3.4. What is an entity type? What is an entity set? Explain the differences among an entity, an entity type, and an entity set.
3.5. Explain the difference between an attribute and a value set.
3.6. What is a relationship type? Explain the differences among a relationship instance, a relationship type, and a relationship set.
3.7. What is a participation role? When is it necessary to use role names in the description of relationship types?
3.8. Describe the two alternatives for specifying structural constraints on relationship types. What are the advantages and disadvantages of each?
3.9. Under what conditions can an attribute of a binary relationship type be migrated to become an attribute of one of the participating entity types?
3.10. When we think of relationships as attributes, what are the value sets of these attributes? What class of data models is based on this concept?
3.11. What is meant by a recursive relationship type? Give some examples of recursive relationship types.
3.12. When is the concept of a weak entity used in data modeling? Define the terms *owner entity type*, *weak entity type*, *identifying relationship type*, and *partial key*.
3.13. Can an identifying relationship of a weak entity type be of a degree greater than two? Give examples to illustrate your answer.
3.14. Discuss the conventions for displaying an ER schema as an ER diagram.
3.15. Discuss the naming conventions used for ER schema diagrams.

Exercises

3.16. Consider the following set of requirements for a university database that is used to keep track of students' transcripts. This is similar but not identical to the database shown in Figure 1.2:
   a. The university keeps track of each student's name, student number, social security number, current address and phone, permanent address and phone, birthdate, sex, class (freshman, sophomore, . . . , graduate), major department, minor department (if any), and degree program (B.A., B.S., . . . , Ph.D.). Some user applications need to refer to the city, state, and zip code of the student's permanent address and to the student's last name. Both social security number and student number have unique values for each student.
   b. Each department is described by a name, department code, office number, office phone, and college. Both name and code have unique values for each department.

c. Each course has a course name, description, course number, number of semester hours, level, and offering department. The value of the course number is unique for each course.
d. Each section has an instructor, semester, year, course, and section number. The section number distinguishes sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, . . . , up to the number of sections taught during each semester.
e. A grade report has a student, section, letter grade, and numeric grade (0, 1, 2, 3, or 4).

Design an ER schema for this application, and draw an ER diagram for that schema. Specify key attributes of each entity type, and structural constraints on each relationship type. Note any unspecified requirements, and make appropriate assumptions to make the specification complete.

3.17. Composite and multivalued attributes can be nested to any number of levels. Suppose we want to design an attribute for a STUDENT entity type to keep track of previous college education. Such an attribute will have one entry for each college previously attended, and each such entry will be composed of college name, start and end dates, degree entries (degrees awarded at that college, if any), and transcript entries (courses completed at that college, if any). Each degree entry contains the degree name and the month and year the degree was awarded, and each transcript entry contains a course name, semester, year, and grade. Design an attribute to hold this information. Use the conventions of Figure 3.5.

3.18. Show an alternative design for the attribute described in Exercise 3.17 that uses only entity types (including weak entity types, if needed) and relationship types.

3.19. Consider the ER diagram of Figure 3.17, which shows a simplified schema for an airline reservations system. Extract from the ER diagram the requirements and constraints that produced this schema. Try to be as precise as possible in your requirements and constraints specification.

3.20. In Chapters 1 and 2, we discussed the database environment and database users. We can consider many entity types to describe such an environment, such as DBMS, stored database, DBA, and catalog/data dictionary. Try to specify all the entity types that can fully describe a database system and its environment; then specify the relationship types among them, and draw an ER diagram to describe such a general database environment.

3.21. Design an ER schema for keeping track of information about votes taken in the U.S. House of Representatives during the current two-year congressional session. The database needs to keep track of each U.S. STATE's Name (e.g., Texas, New York, California) and include the Region of the state (whose domain is {Northeast, Midwest, Southeast, Southwest, West}). Each CONGRESSPERSON in the House of Representatives is described by his or her Name, plus the District represented, the StartDate when the congressperson was first elected, and the political Party to which he or she belongs (whose domain is {Republican, Democrat, Independent, Other}). The database keeps track of each BILL (i.e., proposed law), including the BillName, the DateOfVote on the bill, whether the bill PassedOrFailed (whose domain is {Yes, No}), and the Sponsor (the congressperson(s) who sponsored—

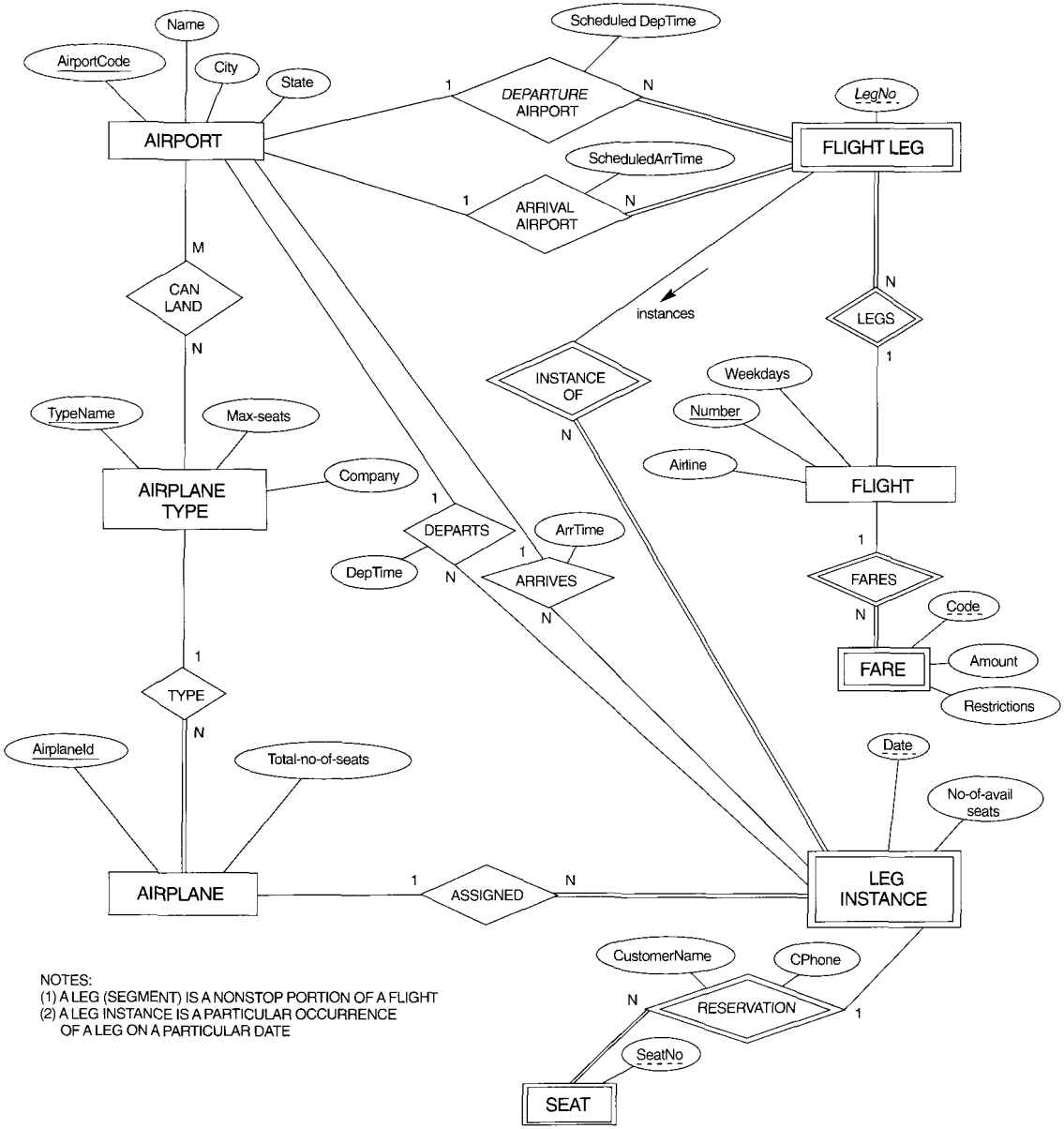

FIGURE 3.17 An ER diagram for an AIRLINE database schema.

that is, proposed—the bill). The database keeps track of how each congressperson voted on each bill (domain of vote attribute is {Yes, No, Abstain, Absent}). Draw an ER schema diagram for this application. State clearly any assumptions you make.

3.22. A database is being constructed to keep track of the teams and games of a sports league. A team has a number of players, not all of whom participate in each game. It is desired to keep track of the players participating in each game for each team, the positions they played in that game, and the result of the game. Design an ER schema diagram for this application, stating any assumptions you make. Choose your favorite sport (e.g., soccer, baseball, football).

3.23. Consider the ER diagram shown in Figure 3.18 for part of a BANK database. Each bank can have multiple branches, and each branch can have multiple accounts and loans.

a. List the (nonweak) entity types in the ER diagram.

b. Is there a weak entity type? If so, give its name, partial key, and identifying relationship.

c. What constraints do the partial key and the identifying relationship of the weak entity type specify in this diagram?

d. List the names of all relationship types, and specify the (min, max) constraint on each participation of an entity type in a relationship type. Justify your choices.

e. List concisely the user requirements that led to this ER schema design.

f. Suppose that every customer must have at least one account but is restricted to at most two loans at a time, and that a bank branch cannot have more than 1000 loans. How does this show up on the (min, max) constraints?

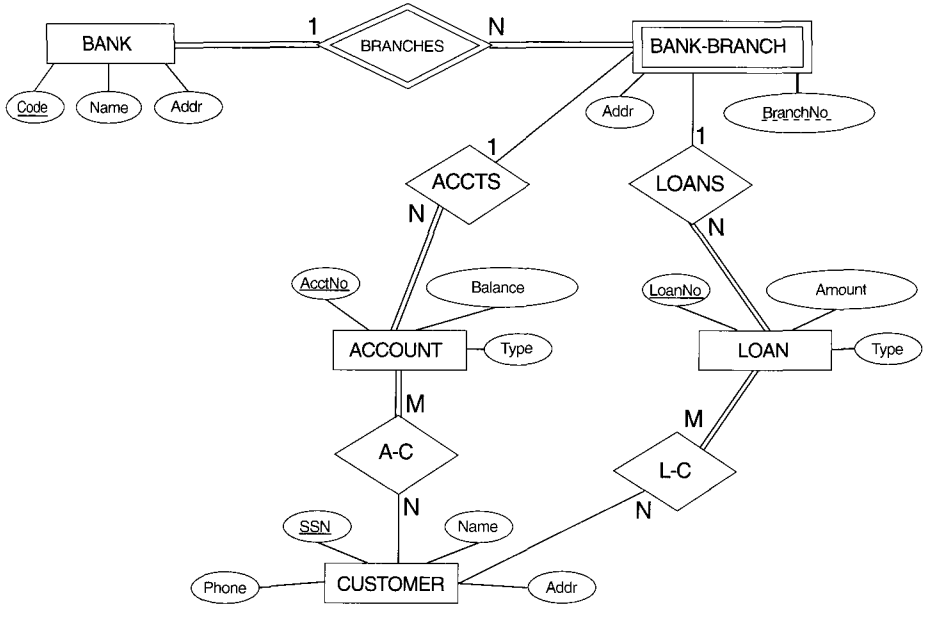

FIGURE 3.18 An ER diagram for a BANK database schema.

3.24. Consider the ER diagram in Figure 3.19. Assume that an employee may work in up to two departments or may not be assigned to any department. Assume that each department must have one and may have up to three phone numbers. Supply (min, max) constraints on this diagram. *State clearly any additional assumptions you make.* Under what conditions would the relationship HAS_PHONE be redundant in this example?

3.25. Consider the ER diagram in Figure 3.20. Assume that a course may or may not use a textbook, but that a text by definition is a book that is used in some course. A course may not use more than five books. Instructors teach from two to four courses. Supply (min, max) constraints on this diagram. *State clearly any additional assumptions you make.* If we add the relationship ADOPTS between INSTRUCTOR and TEXT, what (min, max) constraints would you put on it? Why?

3.26. Consider an entity type SECTION in a UNIVERSITY database, which describes the section offerings of courses. The attributes of SECTION are SectionNumber, Semester, Year, CourseNumber, Instructor, RoomNo (where section is taught), Building (where section is taught), Weekdays (domain is the possible combinations of weekdays in which a section can be offered {MWF, MW, TT, etc.}), and Hours (domain is all possible time periods during which sections are offered {9–9:50 A.M., 10–10:50 A.M., . . . , 3:30–4:50 P.M., 5:30–6:20 P.M., etc.}). Assume that Section-

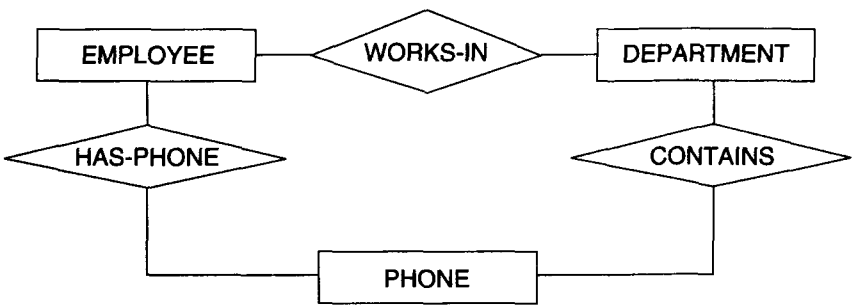

FIGURE 3.19 Part of an ER diagram for a COMPANY database.

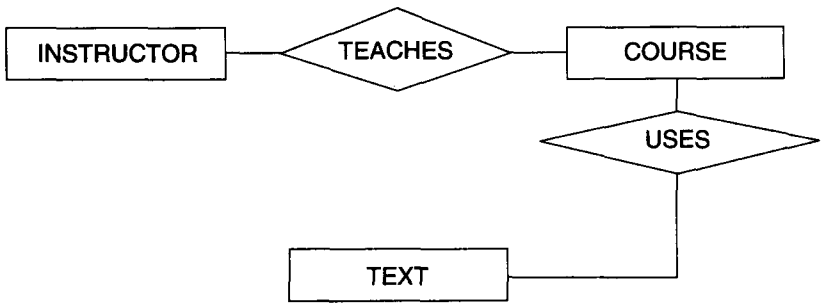

FIGURE 3.20 Part of an ER diagram for a COURSES database.

Number is unique for each course within a particular semester/year combination (that is, if a course is offered multiple times during a particular semester, its section offerings are numbered 1, 2, 3, etc.). There are several composite keys for SECTION, and some attributes are components of more than one key. Identify three composite keys, and show how they can be represented in an ER schema diagram.

Selected Bibliography

The Entity-Relationship model was introduced by Chen (1976), and related work appears in Schmidt and Swenson (1975), Wiederhold and Elmasri (1979), and Senko (1975). Since then, numerous modifications to the ER model have been suggested. We have incorporated some of these in our presentation. Structural constraints on relationships are discussed in Abrial (1974), Elmasri and Wiederhold (1980), and Lenzerini and Santucci (1983). Multivalued and composite attributes are incorporated in the ER model in Elmasri et al. (1985). Although we did not discuss languages for the entity-relationship model and its extensions, there have been several proposals for such languages. Elmasri and Wiederhold (1981) proposed the GORDAS query language for the ER model. Another ER query language was proposed by Markowitz and Raz (1983). Senko (1980) presented a query language for Senko's DIAM model. A formal set of operations called the ER algebra was presented by Parent and Spaccapietra (1985). Gogolla and Hohenstein (1991) presented another formal language for the ER model. Campbell et al. (1985) presented a set of ER operations and showed that they are relationally complete. A conference for the dissemination of research results related to the ER model has been held regularly since 1979. The conference, now known as the International Conference on Conceptual Modeling, has been held in Los Angeles (ER 1979, ER 1983, ER 1997), Washington, D.C. (ER 1981), Chicago (ER 1985), Dijon, France (ER 1986), New York City (ER 1987), Rome (ER 1988), Toronto (ER 1989), Lausanne, Switzerland (ER 1990), San Mateo, California (ER 1991), Karlsruhe, Germany (ER 1992), Arlington, Texas (ER 1993), Manchester, England (ER 1994), Brisbane, Australia (ER 1995), Cottbus, Germany (ER 1996), Singapore (ER 1998), Salt Lake City, Utah (ER 1999), Yokohama, Japan (ER 2001), and Tampere, Finland (ER 2002). The next conference is scheduled for Chicago in October 2003.

4

Enhanced Entity-Relationship and UML Modeling

The ER modeling concepts discussed in Chapter 3 are sufficient for representing many database schemas for "traditional" database applications, which mainly include data-processing applications in business and industry. Since the late 1970s, however, designers of database applications have tried to design more accurate database schemas that reflect the data properties and constraints more precisely. This was particularly important for newer applications of database technology, such as databases for engineering design and manufacturing (CAD/CAM[1]), telecommunications, complex software systems, and Geographic Information Systems (GIS), among many other applications. These types of databases have more complex requirements than do the more traditional applications. This led to the development of additional *semantic data modeling* concepts that were incorporated into conceptual data models such as the ER model. Various semantic data models have been proposed in the literature. Many of these concepts were also developed independently in related areas of computer science, such as the **knowledge representation** area of artificial intelligence and the **object modeling** area in software engineering.

In this chapter, we describe features that have been proposed for semantic data models, and show how the ER model can be enhanced to include these concepts, leading to the **enhanced ER,** or **EER,** model.[2] We start in Section 4.1 by incorporating the

1. CAD/CAM stands for computer-aided design/computer-aided manufacturing.

2. EER has also been used to stand for *Extended* ER model.

concepts of *class/subclass relationships* and *type inheritance* into the ER model. Then, in Section 4.2, we add the concepts of *specialization* and *generalization*. Section 4.3 discusses the various types of *constraints* on specialization/generalization, and Section 4.4 shows how the UNION construct can be modeled by including the concept of *category* in the EER model. Section 4.5 gives an example UNIVERSITY database schema in the EER model and summarizes the EER model concepts by giving formal definitions.

We then present the UML class diagram notation and concepts for representing specialization and generalization in Section 4.6, and briefly compare these with EER notation and concepts. This is a continuation of Section 3.8, which presented basic UML class diagram notation.

Section 4.7 discusses some of the more complex issues involved in modeling of ternary and higher-degree relationships. In Section 4.8, we discuss the fundamental abstractions that are used as the basis of many semantic data models. Section 4.9 summarizes the chapter.

For a detailed introduction to conceptual modeling, Chapter 4 should be considered a continuation of Chapter 3. However, if only a basic introduction to ER modeling is desired, this chapter may be omitted. Alternatively, the reader may choose to skip some or all of the later sections of this chapter (Sections 4.4 through 4.8).

4.1 Subclasses, Superclasses, and Inheritance

The EER (Enhanced ER) model includes all the modeling concepts of the ER model that were presented in Chapter 3. In addition, it includes the concepts of **subclass** and **superclass** and the related concepts of **specialization** and **generalization** (see Sections 4.2 and 4.3). Another concept included in the EER model is that of a **category** or **union type** (see Section 4.4), which is used to represent a collection of objects that is the *union* of objects of different entity types. Associated with these concepts is the important mechanism of **attribute and relationship inheritance.** Unfortunately, no standard terminology exists for these concepts, so we use the most common terminology. Alternative terminology is given in footnotes. We also describe a diagrammatic technique for displaying these concepts when they arise in an EER schema. We call the resulting schema diagrams **enhanced ER** or **EER diagrams.**

The first EER model concept we take up is that of a **subclass** of an entity type. As we discussed in Chapter 3, an entity type is used to represent both a *type of entity* and the *entity set* or *collection of entities of that type* that exist in the database. For example, the entity type EMPLOYEE describes the type (that is, the attributes and relationships) of each employee entity, and also refers to the current set of EMPLOYEE entities in the COMPANY database. In many cases an entity type has numerous subgroupings of its entities that are meaningful and need to be represented explicitly because of their significance to the database application. For example, the entities that are members of the EMPLOYEE entity type may be grouped further into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE, and so on. The set of entities in each of the latter groupings is a subset of

the entities that belong to the EMPLOYEE entity set, meaning that every entity that is a member of one of these subgroupings is also an employee. We call each of these subgroupings a **subclass** of the EMPLOYEE entity type, and the EMPLOYEE entity type is called the **superclass** for each of these subclasses. Figure 4.1 shows how to diagramatically represent these concepts in EER diagrams.

We call the relationship between a superclass and any one of its subclasses a **superclass/subclass** or simply **class/subclass relationship.**[3] In our previous example, EMPLOYEE/SECRETARY and EMPLOYEE/TECHNICIAN are two class/subclass relationships. Notice that a member entity of the subclass represents the *same real-world entity* as some member of the superclass; for example, a SECRETARY entity 'Joan Logano' is also the EMPLOYEE 'Joan Logano'. Hence, the subclass member is the same as the entity in the superclass, but in a distinct *specific role*. When we implement a superclass/subclass relationship in the

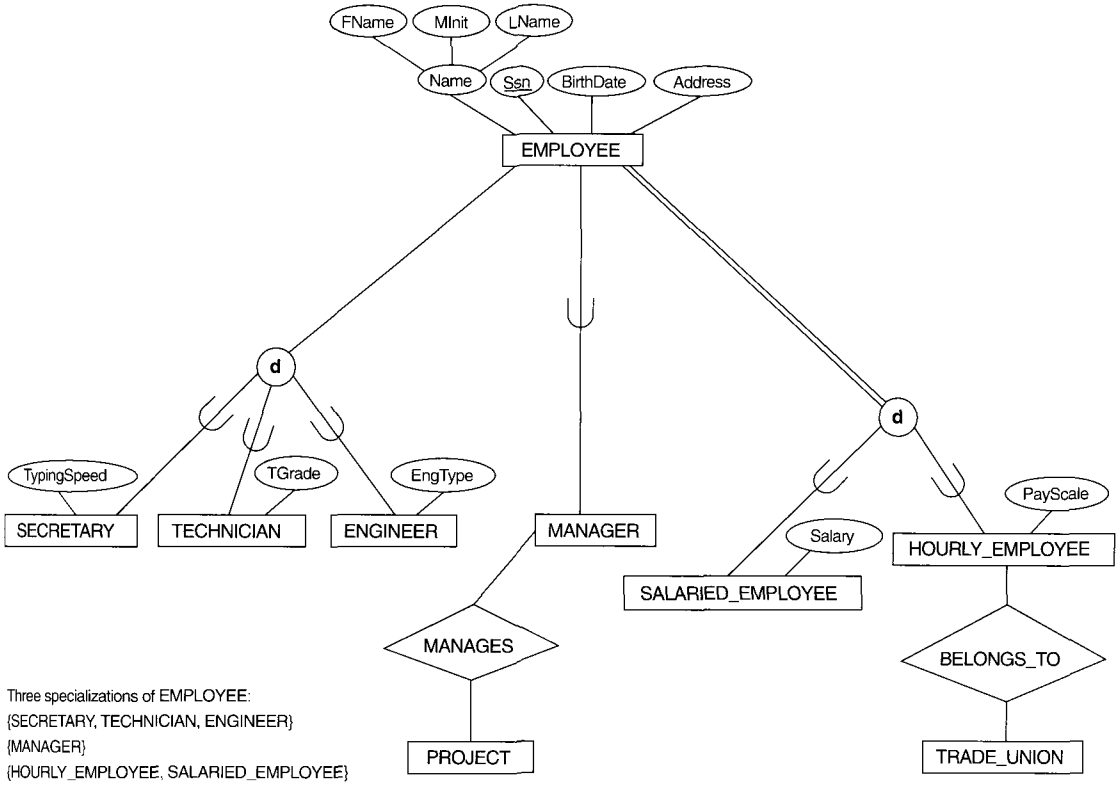

FIGURE 4.1 EER diagram notation to represent subclasses and specialization.

3. A class/subclass relationship is often called an **IS-A** (or **IS-AN**) **relationship** because of the way we refer to the concept. We say "a SECRETARY is an EMPLOYEE," "a TECHNICIAN is an EMPLOYEE," and so on.

database system, however, we may represent a member of the subclass as a distinct database object—say, a distinct record that is related via the key attribute to its superclass entity. In Section 7.2, we discuss various options for representing superclass/subclass relationships in relational databases.

An entity cannot exist in the database merely by being a member of a subclass; it must also be a member of the superclass. Such an entity can be included optionally as a member of any number of subclasses. For example, a salaried employee who is also an engineer belongs to the two subclasses ENGINEER and SALARIED_EMPLOYEE of the EMPLOYEE entity type. However, it is not necessary that every entity in a superclass be a member of some subclass.

An important concept associated with subclasses is that of **type inheritance.** Recall that the *type* of an entity is defined by the attributes it possesses and the relationship types in which it participates. Because an entity in the subclass represents the same real-world entity from the superclass, it should possess values for its specific attributes *as well as* values of its attributes as a member of the superclass. We say that an entity that is a member of a subclass **inherits** all the attributes of the entity as a member of the superclass. The entity also inherits all the relationships in which the superclass participates. Notice that a subclass, with its own specific (or local) attributes and relationships together with all the attributes and relationships it inherits from the superclass, can be considered an *entity type* in its own right.[4]

4.2 Specialization and Generalization

4.2.1 Specialization

Specialization is the process of defining a *set of subclasses* of an entity type; this entity type is called the **superclass** of the specialization. The set of subclasses that form a specialization is defined on the basis of some distinguishing characteristic of the entities in the superclass. For example, the set of subclasses {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of the superclass EMPLOYEE that distinguishes among employee entities based on the *job type* of each employee entity. We may have several specializations of the same entity type based on different distinguishing characteristics. For example, another specialization of the EMPLOYEE entity type may yield the set of subclasses {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}; this specialization distinguishes among employees based on the *method of pay.*

Figure 4.1 shows how we represent a specialization diagrammatically in an EER diagram. The subclasses that define a specialization are attached by lines to a circle that represents the specialization, which is connected to the superclass. The *subset symbol* on each line connecting a subclass to the circle indicates the direction of the superclass/subclass relationship.[5] Attributes that apply only to entities of a particular subclass—such

4. In some object-oriented programming languages, a common restriction is that an entity (or object) has *only one type*. This is generally too restrictive for conceptual database modeling.

5. There are many alternative notations for specialization; we present the UML notation in Section 4.6 and other proposed notations in Appendix A.

as TypingSpeed of SECRETARY–are attached to the rectangle representing that subclass. These are called **specific attributes** (or **local attributes**) of the subclass. Similarly, a subclass can participate in **specific relationship types,** such as the HOURLY_EMPLOYEE subclass participating in the BELONGS_TO relationship in Figure 4.1. We will explain the d symbol in the circles of Figure 4.1 and additional EER diagram notation shortly.

Figure 4.2 shows a few entity instances that belong to subclasses of the {SECRETARY, ENGINEER, TECHNICIAN} specialization. Again, notice that an entity that belongs to a subclass represents *the same real-world entity* as the entity connected to it in the EMPLOYEE superclass, even though the same entity is shown twice; for example, e_1 is shown in both EMPLOYEE and SECRETARY in Figure 4.2. As this figure suggests, a superclass/subclass relationship such as

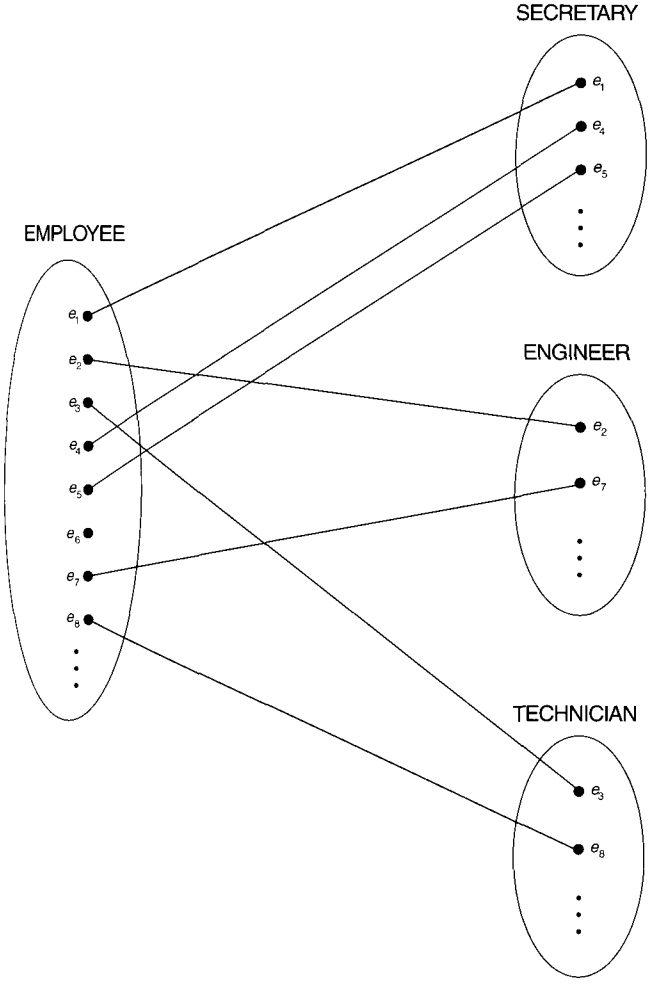

FIGURE 4.2 Instances of a specialization.

EMPLOYEE/SECRETARY somewhat resembles a 1:1 relationship *at the instance level* (see Figure 3.12). The main difference is that in a 1:1 relationship two *distinct entities* are related, whereas in a superclass/subclass relationship the entity in the subclass is the same real-world entity as the entity in the superclass but is playing a *specialized role*—for example, an EMPLOYEE specialized in the role of SECRETARY, or an EMPLOYEE specialized in the role of TECHNICIAN.

There are two main reasons for including class/subclass relationships and specializations in a data model. The first is that certain attributes may apply to some but not all entities of the superclass. A subclass is defined in order to group the entities to which these attributes apply. The members of the subclass may still share the majority of their attributes with the other members of the superclass. For example, in Figure 4.1 the SECRETARY subclass has the specific attribute TypingSpeed, whereas the ENGINEER subclass has the specific attribute EngType, but SECRETARY and ENGINEER share their other inherited attributes from the EMPLOYEE entity type.

The second reason for using subclasses is that some relationship types may be participated in only by entities that are members of the subclass. For example, if only HOURLY_EMPLOYEES can belong to a trade union, we can represent that fact by creating the subclass HOURLY_EMPLOYEE of EMPLOYEE and relating the subclass to an entity type TRADE_UNION via the BELONGS_TO relationship type, as illustrated in Figure 4.1.

In summary, the specialization process allows us to do the following:

- Define a set of subclasses of an entity type
- Establish additional specific attributes with each subclass
- Establish additional specific relationship types between each subclass and other entity types or other subclasses

4.2.2 Generalization

We can think of a *reverse process* of abstraction in which we suppress the differences among several entity types, identify their common features, and **generalize** them into a single **superclass** of which the original entity types are special **subclasses.** For example, consider the entity types CAR and TRUCK shown in Figure 4.3a. Because they have several common attributes, they can be generalized into the entity type VEHICLE, as shown in Figure 4.3b. Both CAR and TRUCK are now subclasses of the **generalized superclass** VEHICLE. We use the term **generalization** to refer to the process of defining a generalized entity type from the given entity types.

Notice that the generalization process can be viewed as being functionally the inverse of the specialization process. Hence, in Figure 4.3 we can view {CAR, TRUCK} as a specialization of VEHICLE, rather than viewing VEHICLE as a generalization of CAR and TRUCK. Similarly, in Figure 4.1 we can view EMPLOYEE as a generalization of SECRETARY, TECHNICIAN, and ENGINEER. A diagrammatic notation to distinguish between generalization and specialization is used in some design methodologies. An arrow pointing to the generalized superclass represents a generalization, whereas arrows pointing to the specialized subclasses represent a specialization. We will *not* use this notation, because the decision as to which process is more appropriate in a particular situation is often subjective. Appendix A gives some of the suggested alternative diagrammatic notations for schema diagrams and class diagrams.

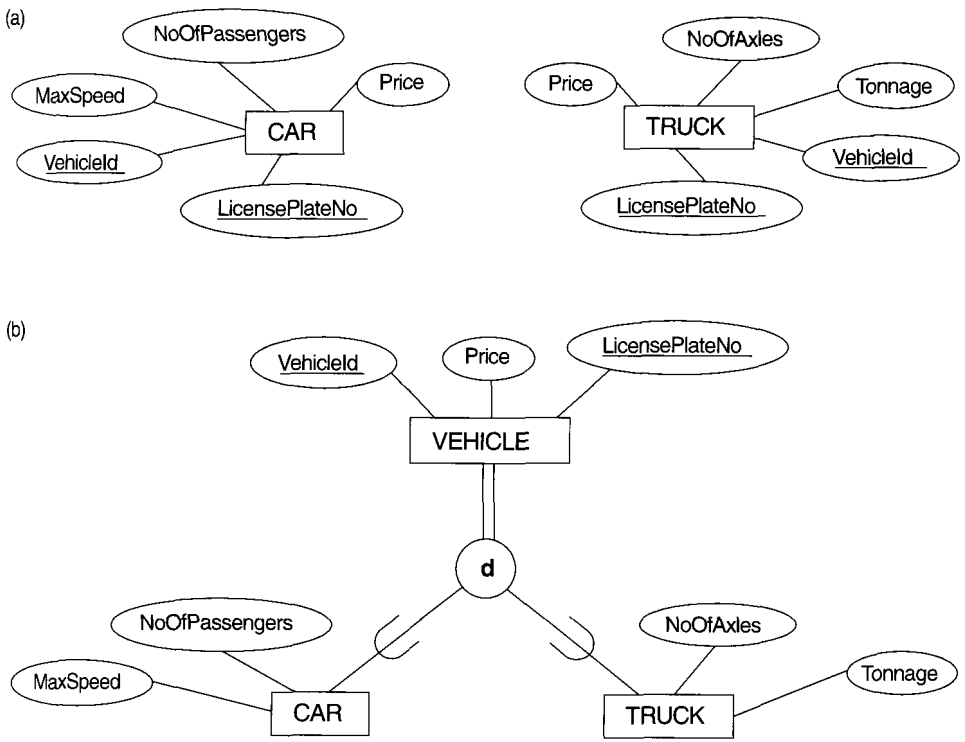

FIGURE 4.3 Generalization. (a) Two entity types, CAR and TRUCK. (b) Generalizing CAR and TRUCK into the superclass VEHICLE.

So far we have introduced the concepts of subclasses and superclass/subclass relationships, as well as the specialization and generalization processes. In general, a superclass or subclass represents a collection of entities of the same type and hence also describes an *entity type*; that is why superclasses and subclasses are shown in rectangles in EER diagrams, like entity types. We next discuss in more detail the properties of specializations and generalizations.

4.3 Constraints and Characteristics of Specialization and Generalization

We first discuss constraints that apply to a single specialization or a single generalization. For brevity, our discussion refers only to *specialization* even though it applies to *both* specialization and generalization. We then discuss differences between specialization/generalization *lattices* (*multiple inheritance*) and *hierarchies* (*single inheritance*), and elaborate on the differences between the specialization and generalization processes during conceptual database schema design.

4.3.1 Constraints on Specialization and Generalization

In general, we may have several specializations defined on the same entity type (or superclass), as shown in Figure 4.1. In such a case, entities may belong to subclasses in each of the specializations. However, a specialization may also consist of a *single* subclass only, such as the {MANAGER} specialization in Figure 4.1; in such a case, we do not use the circle notation.

In some specializations we can determine exactly the entities that will become members of each subclass by placing a condition on the value of some attribute of the superclass. Such subclasses are called **predicate-defined** (or **condition-defined**) **subclasses.** For example, if the EMPLOYEE entity type has an attribute JobType, as shown in Figure 4.4, we can specify the condition of membership in the SECRETARY subclass by the condition (JobType = 'Secretary'), which we call the **defining predicate** of the subclass. This condition is a *constraint* specifying that exactly those entities of the EMPLOYEE entity type whose attribute value for JobType is 'Secretary' belong to the subclass. We display a predicate-defined subclass by writing the predicate condition next to the line that connects the subclass to the specialization circle.

If *all* subclasses in a specialization have their membership condition on the *same* attribute of the superclass, the specialization itself is called an **attribute-defined specialization,** and the attribute is called the **defining attribute** of the specialization.[6] We display an attribute-defined specialization by placing the defining attribute name next to the arc from the circle to the superclass, as shown in Figure 4.4.

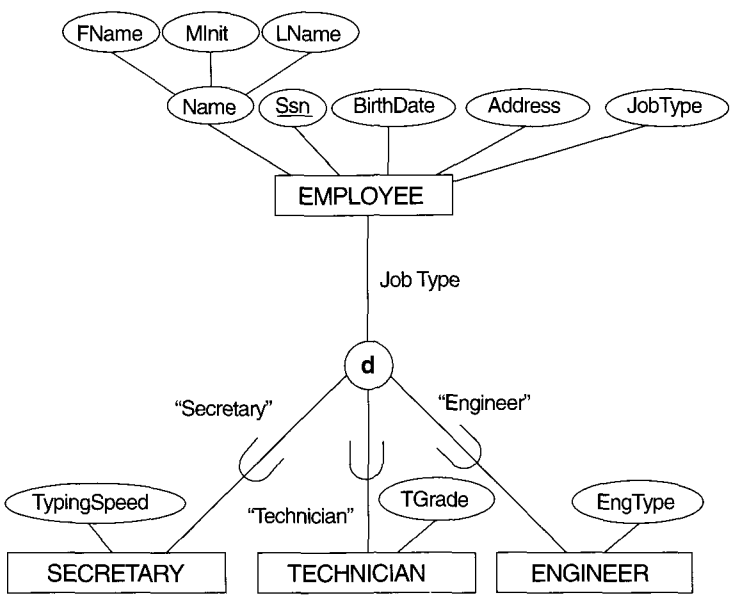

FIGURE 4.4 EER diagram notation for an attribute-defined specialization on JobType.

6. Such an attribute is called a *discriminator* in UML terminology.

When we do not have a condition for determining membership in a subclass, the subclass is called **user-defined.** Membership in such a subclass is determined by the database users when they apply the operation to add an entity to the subclass; hence, membership is *specified individually for each entity by the user*, not by any condition that may be evaluated automatically.

Two other constraints may apply to a specialization. The first is the **disjointness constraint,** which specifies that the subclasses of the specialization must be disjoint. This means that an entity can be a member of *at most one* of the subclasses of the specialization. A specialization that is attribute-defined implies the disjointness constraint if the attribute used to define the membership predicate is single-valued. Figure 4.4 illustrates this case, where the **d** in the circle stands for *disjoint*. We also use the **d** notation to specify the constraint that user-defined subclasses of a specialization must be disjoint, as illustrated by the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} in Figure 4.1. If the subclasses are not constrained to be disjoint, their sets of entities may **overlap;** that is, the same (real-world) entity may be a member of more than one subclass of the specialization. This case, which is the default, is displayed by placing an **o** in the circle, as shown in Figure 4.5.

The second constraint on specialization is called the **completeness constraint,** which may be total or partial. A **total specialization** constraint specifies that *every* entity in the superclass must be a member of at least one subclass in the specialization. For example, if every EMPLOYEE must be either an HOURLY_EMPLOYEE or a SALARIED_EMPLOYEE, then the specialization {HOURLY_EMPLOYEE, SALARIED_EMPLOYEE} of Figure 4.1 is a total specialization of EMPLOYEE. This is shown in EER diagrams by using a double line to connect the superclass to the circle. A single line is used to display a **partial specialization,** which allows an entity not to belong to any of the subclasses. For example, if some EMPLOYEE entities do not belong

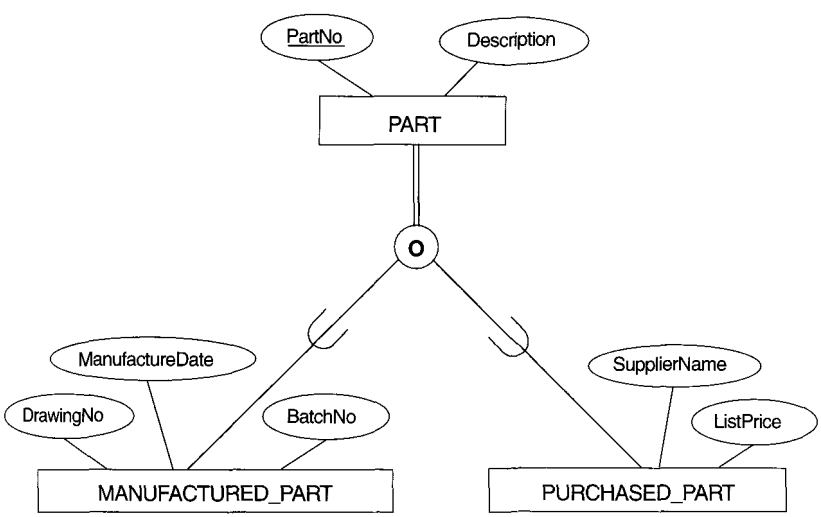

FIGURE 4.5 EER diagram notation for an overlapping (nondisjoint) specialization.

to any of the subclasses {SECRETARY, ENGINEER, TECHNICIAN} of Figures 4.1 and 4.4, then that specialization is partial.[7]

Notice that the disjointness and completeness constraints are *independent*. Hence, we have the following four possible constraints on specialization:

- Disjoint, total
- Disjoint, partial
- Overlapping, total
- Overlapping, partial

Of course, the correct constraint is determined from the real-world meaning that applies to each specialization. In general, a superclass that was identified through the *generalization* process usually is **total,** because the superclass is *derived from* the subclasses and hence contains only the entities that are in the subclasses.

Certain insertion and deletion rules apply to specialization (and generalization) as a consequence of the constraints specified earlier. Some of these rules are as follows:

- Deleting an entity from a superclass implies that it is automatically deleted from all the subclasses to which it belongs.
- Inserting an entity in a superclass implies that the entity is mandatorily inserted in all *predicate-defined* (or *attribute-defined*) subclasses for which the entity satisfies the defining predicate.
- Inserting an entity in a superclass of a *total specialization* implies that the entity is mandatorily inserted in at least one of the subclasses of the specialization.

The reader is encouraged to make a complete list of rules for insertions and deletions for the various types of specializations.

4.3.2 Specialization and Generalization Hierarchies and Lattices

A subclass itself may have further subclasses specified on it, forming a hierarchy or a lattice of specializations. For example, in Figure 4.6 ENGINEER is a subclass of EMPLOYEE and is also a superclass of ENGINEERING_MANAGER; this represents the real-world constraint that every engineering manager is required to be an engineer. A **specialization hierarchy** has the constraint that every subclass participates *as a subclass* in *only one* class/subclass relationship; that is, each subclass has only one parent, which results in a tree structure. In contrast, for a **specialization lattice,** a subclass can be a subclass in *more than one* class/subclass relationship. Hence, Figure 4.6 is a lattice.

Figure 4.7 shows another specialization lattice of more than one level. This may be part of a conceptual schema for a UNIVERSITY database. Notice that this arrangement would

7. The notation of using single or double lines is similar to that for partial or total participation of an entity type in a relationship type, as described in Chapter 3.

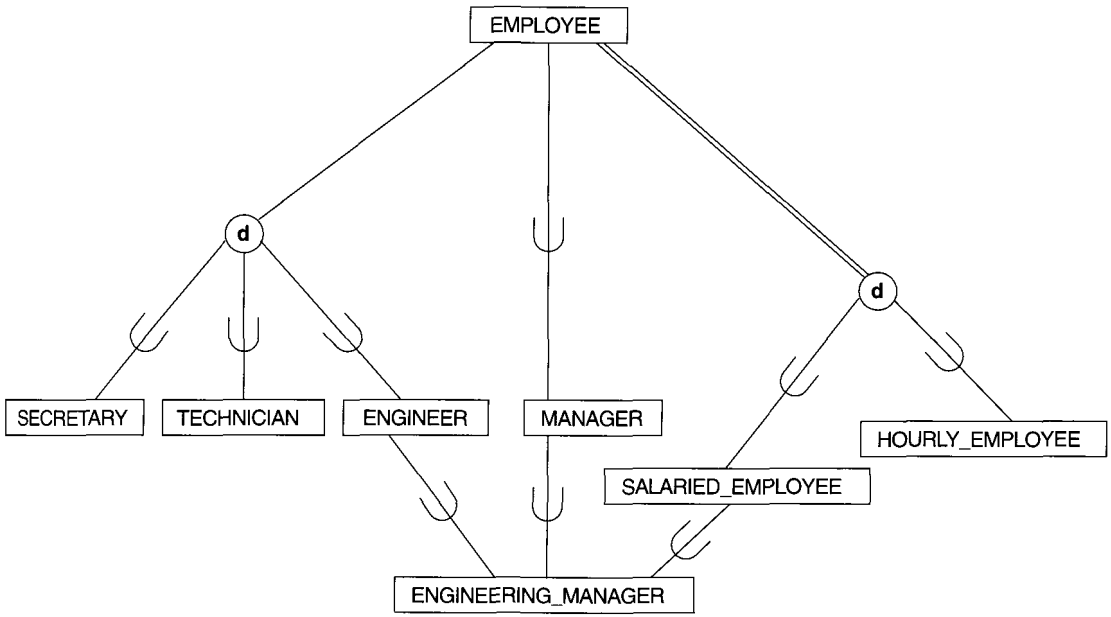

FIGURE 4.6 A specialization lattice with shared subclass ENGINEERING_MANAGER.

have been a hierarchy except for the STUDENT_ASSISTANT subclass, which is a subclass in two distinct class/subclass relationships. In Figure 4.7, all person entities represented in the database are members of the PERSON entity type, which is specialized into the subclasses {EMPLOYEE, ALUMNUS, STUDENT}. This specialization is overlapping; for example, an alumnus may also be an employee and may also be a student pursuing an advanced degree. The subclass STUDENT is the superclass for the specialization {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT}, while EMPLOYEE is the superclass for the specialization {STUDENT_ASSISTANT, FACULTY, STAFF}. Notice that STUDENT_ASSISTANT is also a subclass of STUDENT. Finally, STUDENT_ASSISTANT is the superclass for the specialization into {RESEARCH_ASSISTANT, TEACHING_ASSISTANT}.

In such a specialization lattice or hierarchy, a subclass inherits the attributes not only of its direct superclass but also of all its predecessor superclasses *all the way to the root* of the hierarchy or lattice. For example, an entity in GRADUATE_STUDENT inherits all the attributes of that entity as a STUDENT *and* as a PERSON. Notice that an entity may exist in several *leaf nodes* of the hierarchy, where a **leaf node** is a class that has *no subclasses of its own*. For example, a member of GRADUATE_STUDENT may also be a member of RESEARCH_ASSISTANT.

A subclass with *more than one* superclass is called a **shared subclass,** such as ENGINEERING_MANAGER in Figure 4.6. This leads to the concept known as **multiple inheritance,** where the shared subclass ENGINEERING_MANAGER directly inherits attributes and relationships from multiple classes. Notice that the existence of at least one shared subclass leads to a lattice (and hence to *multiple inheritance*); if no shared subclasses existed, we would have a hierarchy rather than a lattice. An important rule related to multiple inheritance can be illustrated by the example of the shared subclass STUDENT_ASSISTANT in Figure 4.7, which

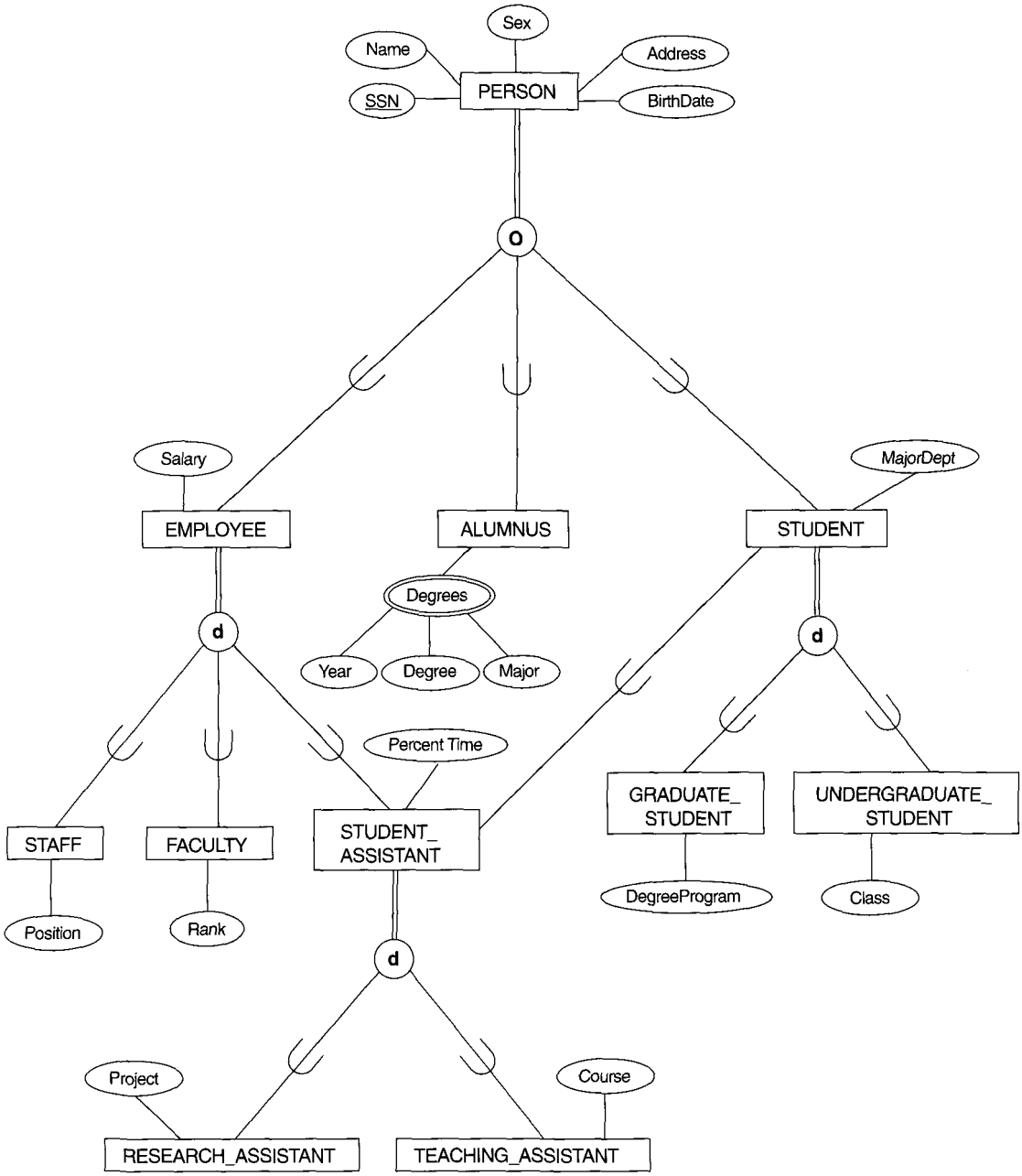

FIGURE 4.7 A specialization lattice with multiple inheritance for a UNIVERSITY database.

inherits attributes from both EMPLOYEE and STUDENT. Here, both EMPLOYEE and STUDENT inherit *the same attributes* from PERSON. The rule states that if an attribute (or relationship) originating in the *same superclass* (PERSON) is inherited more than once via different paths (EMPLOYEE and STUDENT) in the lattice, then it should be included only once in the shared subclass (STUDENT_ASSISTANT). Hence, the attributes of PERSON are inherited *only once* in the STUDENT_ASSISTANT subclass of Figure 4.7.

It is important to note here that some models and languages *do not allow* multiple inheritance (shared subclasses). In such a model, it is necessary to create additional subclasses to cover all possible combinations of classes that may have some entity belong to all these classes simultaneously. Hence, any *overlapping* specialization would require multiple additional subclasses. For example, in the overlapping specialization of PERSON into {EMPLOYEE, ALUMNUS, STUDENT} (or {E, A, S} for short), it would be necessary to create seven subclasses of PERSON in order to cover all possible types of entities: E, A, S, E_A, E_S, A_S, and E_A_S. Obviously, this can lead to extra complexity.

It is also important to note that some inheritance mechanisms that allow multiple inheritance do not allow an entity to have multiple types, and hence an entity can be a member *of only one class*.[8] In such a model, it is also necessary to create additional shared subclasses as leaf nodes to cover all possible combinations of classes that may have some entity belong to all these classes simultaneously. Hence, we would require the same seven subclasses of PERSON.

Although we have used specialization to illustrate our discussion, similar concepts *apply equally* to generalization, as we mentioned at the beginning of this section. Hence, we can also speak of **generalization hierarchies** and **generalization lattices.**

4.3.3 Utilizing Specialization and Generalization in Refining Conceptual Schemas

We now elaborate on the differences between the specialization and generalization processes, and how they are used to refine conceptual schemas during conceptual database design. In the specialization process, we typically start with an entity type and then define subclasses of the entity type by successive specialization; that is, we repeatedly define more specific groupings of the entity type. For example, when designing the specialization lattice in Figure 4.7, we may first specify an entity type PERSON for a university database. Then we discover that three types of persons will be represented in the database: university employees, alumni, and students. We create the specialization {EMPLOYEE, ALUMNUS, STUDENT} for this purpose and choose the overlapping constraint because a person may belong to more than one of the subclasses. We then specialize EMPLOYEE further into {STAFF, FACULTY, STUDENT_ASSISTANT}, and specialize STUDENT into {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT}. Finally, we specialize STUDENT_ASSISTANT into {RESEARCH_ASSISTANT, TEACHING_ASSISTANT}. This successive specialization corresponds to a **top-down conceptual refinement process** during concep-

8. In some models, the class is further restricted to be a *leaf node* in the hierarchy or lattice.

tual schema design. So far, we have a hierarchy; we then realize that STUDENT_ASSISTANT is a shared subclass, since it is also a subclass of STUDENT, leading to the lattice.

It is possible to arrive at the same hierarchy or lattice from the other direction. In such a case, the process involves generalization rather than specialization and corresponds to a **bottom-up conceptual synthesis.** In this case, designers may first discover entity types such as STAFF, FACULTY, ALUMNUS, GRADUATE_STUDENT, UNDERGRADUATE_STUDENT, RESEARCH_ASSISTANT, TEACHING_ASSISTANT, and so on; then they generalize {GRADUATE_STUDENT, UNDERGRADUATE_STUDENT} into STUDENT; then they generalize {RESEARCH_ASSISTANT, TEACHING_ASSISTANT} into STUDENT_ASSISTANT; then they generalize {STAFF, FACULTY, STUDENT_ASSISTANT} into EMPLOYEE; and finally they generalize {EMPLOYEE, ALUMNUS, STUDENT} into PERSON.

In structural terms, hierarchies or lattices resulting from either process may be identical; the only difference relates to the manner or order in which the schema superclasses and subclasses were specified. In practice, it is likely that neither the generalization process nor the specialization process is followed strictly, but that a combination of the two processes is employed. In this case, new classes are continually incorporated into a hierarchy or lattice as they become apparent to users and designers. Notice that the notion of representing data and knowledge by using superclass/subclass hierarchies and lattices is quite common in knowledge-based systems and expert systems, which combine database technology with artificial intelligence techniques. For example, frame-based knowledge representation schemes closely resemble class hierarchies. Specialization is also common in software engineering design methodologies that are based on the object-oriented paradigm.

4.4 Modeling of UNION Types Using Categories

All of the superclass/subclass relationships we have seen thus far have a *single superclass*. A shared subclass such as ENGINEERING_MANAGER in the lattice of Figure 4.6 is the subclass in three *distinct* superclass/subclass relationships, where each of the three relationships has a *single* superclass. It is not uncommon, however, that the need arises for modeling a single superclass/subclass relationship with *more than one* superclass, where the superclasses represent different entity types. In this case, the subclass will represent a collection of objects that is a subset of the UNION of distinct entity types; we call such a *subclass* a **union type** or a **category.**[9]

For example, suppose that we have three entity types: PERSON, BANK, and COMPANY. In a database for vehicle registration, an owner of a vehicle can be a person, a bank (holding a lien on a vehicle), or a company. We need to create a class (collection of entities) that includes entities of all three types to play the role of *vehicle owner*. A category OWNER that is a *subclass of the* UNION of the three entity sets of COMPANY, BANK, and PERSON is created for this purpose. We display categories in an EER diagram as shown in Figure 4.8. The superclasses

9. Our use of the term *category* is based on the ECR (Entity-Category-Relationship) model (Elmasri et al. 1985).

COMPANY, BANK, and PERSON are connected to the circle with the U symbol, which stands for the *set union operation*. An arc with the subset symbol connects the circle to the (subclass) OWNER category. If a defining predicate is needed, it is displayed next to the line from the

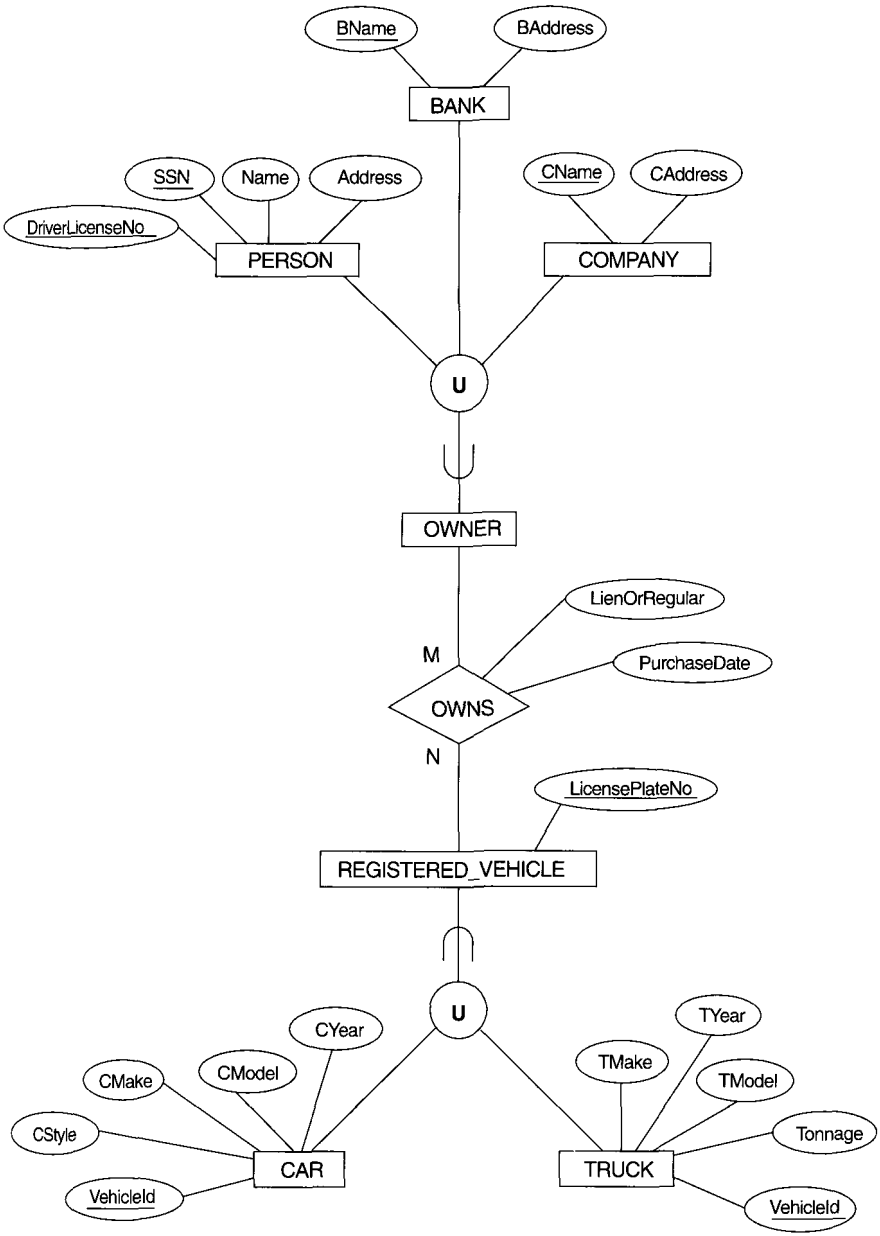

FIGURE 4.8 Two categories (union types): OWNER and REGISTERED_VEHICLE.

superclass to which the predicate applies. In Figure 4.8 we have two categories: OWNER, which is a subclass of the union of PERSON, BANK, and COMPANY; and REGISTERED_VEHICLE, which is a subclass of the union of CAR and TRUCK.

A category has two or more superclasses that may represent *distinct entity types*, whereas other superclass/subclass relationships always have a single superclass. We can compare a category, such as OWNER in Figure 4.8, with the ENGINEERING_MANAGER shared subclass of Figure 4.6. The latter is a subclass of *each of* the three superclasses ENGINEER, MANAGER, and SALARIED_EMPLOYEE, so an entity that is a member of ENGINEERING_MANAGER must exist in *all three*. This represents the constraint that an engineering manager must be an ENGINEER, a MANAGER, *and* a SALARIED_EMPLOYEE; that is, ENGINEERING_MANAGER is a subset of the *intersection* of the three subclasses (sets of entities). On the other hand, a category is a subset of the *union* of its superclasses. Hence, an entity that is a member of OWNER must exist in *only one* of the superclasses. This represents the constraint that an OWNER may be a COMPANY, a BANK, *or* a PERSON in Figure 4.8.

Attribute inheritance works more selectively in the case of categories. For example, in Figure 4.8 each OWNER entity inherits the attributes of a COMPANY, a PERSON, or a BANK, depending on the superclass to which the entity belongs. On the other hand, a shared subclass such as ENGINEERING_MANAGER (Figure 4.6) inherits *all* the attributes of its superclasses SALARIED_EMPLOYEE, ENGINEER, and MANAGER.

It is interesting to note the difference between the category REGISTERED_VEHICLE (Figure 4.8) and the generalized superclass VEHICLE (Figure 4.3b). In Figure 4.3b, every car and every truck is a VEHICLE; but in Figure 4.8, the REGISTERED_VEHICLE category includes some cars and some trucks but not necessarily all of them (for example, some cars or trucks may not be registered). In general, a specialization or generalization such as that in Figure 4.3b, if it were *partial*, would not preclude VEHICLE from containing other types of entities, such as motorcycles. However, a category such as REGISTERED_VEHICLE in Figure 4.8 implies that only cars and trucks, but not other types of entities, can be members of REGISTERED_VEHICLE.

A category can be **total** or **partial.** A total category holds the *union* of all entities in its superclasses, whereas a partial category can hold a *subset of the union*. A total category is represented by a double line connecting the category and the circle, whereas partial categories are indicated by a single line.

The superclasses of a category may have different key attributes, as demonstrated by the OWNER category of Figure 4.8, or they may have the same key attribute, as demonstrated by the REGISTERED_VEHICLE category. Notice that if a category is total (not partial), it may be represented alternatively as a total specialization (or a total generalization). In this case the choice of which representation to use is subjective. If the two classes represent the same type of entities and share numerous attributes, including the same key attributes, specialization/generalization is preferred; otherwise, categorization (union type) is more appropriate.

4.5 An Example UNIVERSITY EER Schema and Formal Definitions for the EER Model

In this section, we first give an example of a database schema in the EER model to illustrate the use of the various concepts discussed here and in Chapter 3. Then, we summarize the EER model concepts and define them formally in the same manner in which we formally defined the concepts of the basic ER model in Chapter 3.

4.5.1 The UNIVERSITY Database Example

For our example database application, consider a UNIVERSITY database that keeps track of students and their majors, transcripts, and registration as well as of the university's course offerings. The database also keeps track of the sponsored research projects of faculty and graduate students. This schema is shown in Figure 4.9. A discussion of the requirements that led to this schema follows.

For each person, the database maintains information on the person's Name [Name], social security number [Ssn], address [Address], sex [Sex], and birth date [BDate]. Two subclasses of the PERSON entity type were identified: FACULTY and STUDENT. Specific attributes of FACULTY are rank [Rank] (assistant, associate, adjunct, research, visiting, etc.), office [FOffice], office phone [FPhone], and salary [Salary]. All faculty members are related to the academic department(s) with which they are affiliated [BELONGS] (a faculty member can be associated with several departments, so the relationship is M:N). A specific attribute of STUDENT is [Class] (freshman = 1, sophomore = 2, . . . , graduate student = 5). Each student is also related to his or her major and minor departments, if known ([MAJOR] and [MINOR]), to the course sections he or she is currently attending [REGISTERED], and to the courses completed [TRANSCRIPT]. Each transcript instance includes the grade the student received [Grade] in the course section.

GRAD_STUDENT is a subclass of STUDENT, with the defining predicate Class = 5. For each graduate student, we keep a list of previous degrees in a composite, multivalued attribute [Degrees]. We also relate the graduate student to a faculty advisor [ADVISOR] and to a thesis committee [COMMITTEE], if one exists.

An academic department has the attributes name [DName], telephone [DPhone], and office number [Office] and is related to the faculty member who is its chairperson [CHAIRS] and to the college to which it belongs [CD]. Each college has attributes college name [CName], office number [COffice], and the name of its dean [Dean].

A course has attributes course number [C#], course name [Cname], and course description [CDesc]. Several sections of each course are offered, with each section having the attributes section number [Sec#] and the year and quarter in which the section was offered ([Year] and [Qtr]).[10] Section numbers uniquely identify each section. The sections being offered during the current quarter are in a subclass CURRENT_SECTION of SECTION, with

10. We assume that the *quarter* system rather than the *semester* system is used in this university.

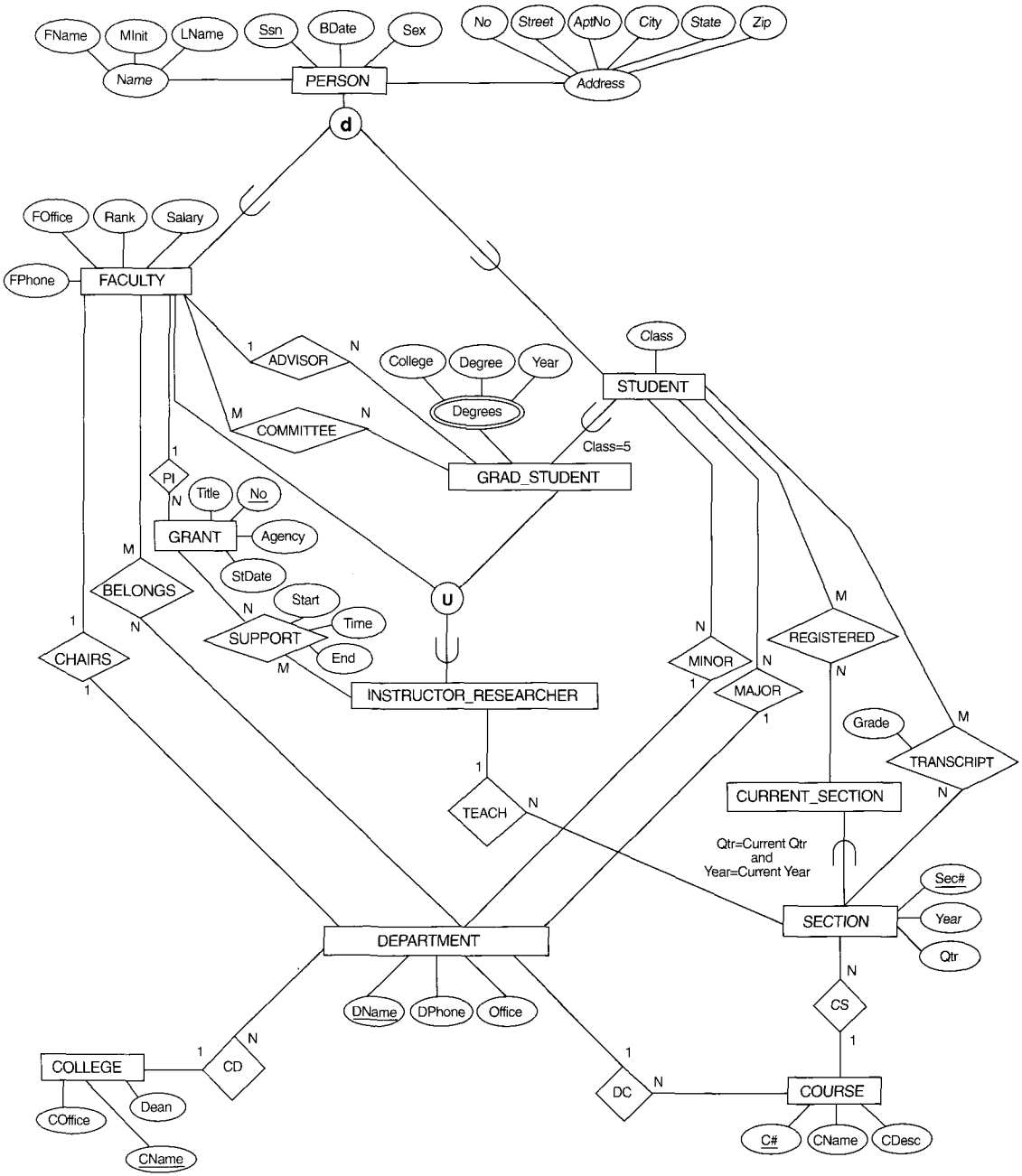

FIGURE 4.9 An EER conceptual schema for a UNIVERSITY database.

the defining predicate Qtr = CurrentQtr and Year = CurrentYear. Each section is related to the instructor who taught or is teaching it ([TEACH]), if that instructor is in the database.

The category INSTRUCTOR_RESEARCHER is a subset of the union of FACULTY and GRAD_STUDENT and includes all faculty, as well as graduate students who are supported by teaching or research. Finally, the entity type GRANT keeps track of research grants and contracts awarded to the university. Each grant has attributes grant title [Title], grant number [No], the awarding agency [Agency], and the starting date [StDate]. A grant is related to one principal investigator [PI] and to all researchers it supports [SUPPORT]. Each instance of support has as attributes the starting date of support [Start], the ending date of the support (if known) [End], and the percentage of time being spent on the project [Time] by the researcher being supported.

4.5.2 Formal Definitions for the EER Model Concepts

We now summarize the EER model concepts and give formal definitions. A **class**[11] is a set or collection of entities; this includes any of the EER schema constructs that group entities, such as entity types, subclasses, superclasses, and categories. A **subclass** S is a class whose entities must always be a subset of the entities in another class, called the **superclass** C of the **superclass/subclass** (or **IS-A**) **relationship.** We denote such a relationship by C/S. For such a superclass/subclass relationship, we must always have

$$S \subseteq C$$

A **specialization** $Z = \{S_1, S_2, \ldots, S_n\}$ is a set of subclasses that have the same superclass G; that is, G/S_i is a superclass/subclass relationship for $i = 1, 2, \ldots, n$. G is called a **generalized entity type** (or the **superclass** of the specialization, or a **generalization** of the subclasses $\{S_1, S_2, \ldots, S_n\}$). Z is said to be **total** if we always (at any point in time) have

$$\bigcup_{i=1}^{n} S_i = G$$

Otherwise, Z is said to be **partial.** Z is said to be **disjoint** if we always have

$$S_i \cap S_j = \varnothing \text{ (empty set) for } i \neq j$$

Otherwise, Z is said to be **overlapping.**

A subclass S of C is said to be **predicate-defined** if a predicate p on the attributes of C is used to specify which entities in C are members of S; that is, $S = C[p]$, where $C[p]$ is the set of entities in C that satisfy p. A subclass that is not defined by a predicate is called **user-defined.**

11. The use of the word *class* here differs from its more common use in object-oriented programming languages such as C++. In C++, a class is a structured type definition along with its applicable functions (operations).

A specialization Z (or generalization G) is said to be **attribute-defined** if a predicate $(A = c_i)$, where A is an attribute of G and c_i is a constant value from the domain of A, is used to specify membership in each subclass S_i in Z. Notice that if $c_i \neq c_j$ for $i \neq j$, and A is a single-valued attribute, then the specialization will be disjoint.

A **category** T is a class that is a subset of the union of n defining superclasses $D_1, D_2, \ldots, D_n$, $n > 1$, and is formally specified as follows:

$$T \subseteq (D_1 \cup D_2 \ldots \cup D_n)$$

A predicate p_i on the attributes of D_i can be used to specify the members of each D_i that are members of T. If a predicate is specified on every D_i, we get

$$T = (D_1[p_1] \cup D_2[p_2] \ldots \cup D_n[p_n])$$

We should now extend the definition of **relationship type** given in Chapter 3 by allowing any class—not only any entity type—to participate in a relationship. Hence, we should replace the words *entity type* with *class* in that definition. The graphical notation of EER is consistent with ER because all classes are represented by rectangles.

4.6 Representing Specialization/ Generalization and Inheritance in UML Class Diagrams

We now discuss the UML notation for generalization/specialization and inheritance. We already presented basic UML class diagram notation and terminology in Section 3.8. Figure 4.10 illustrates a possible UML class diagram corresponding to the EER diagram in Figure 4.7. The basic notation for generalization is to connect the subclasses by vertical lines to a horizontal line, which has a triangle connecting the horizontal line through another vertical line to the superclass (see Figure 4.10). A blank triangle indicates a specialization/generalization with the *disjoint* constraint, and a filled triangle indicates an *overlapping* constraint. The root superclass is called the **base class,** and leaf nodes are called **leaf classes.** Both single and multiple inheritance are permitted.

The above discussion and example (and Section 3.8) give a brief overview of UML class diagrams and terminology. There are many details that we have not discussed because they are outside the scope of this book and are mainly relevant to software engineering. For example, classes can be of various types:

- Abstract classes define attributes and operations but do not have objects corresponding to those classes. These are mainly used to specify a set of attributes and operations that can be inherited.
- Concrete classes can have objects (entities) instantiated to belong to the class.
- Template classes specify a template that can be further used to define other classes.

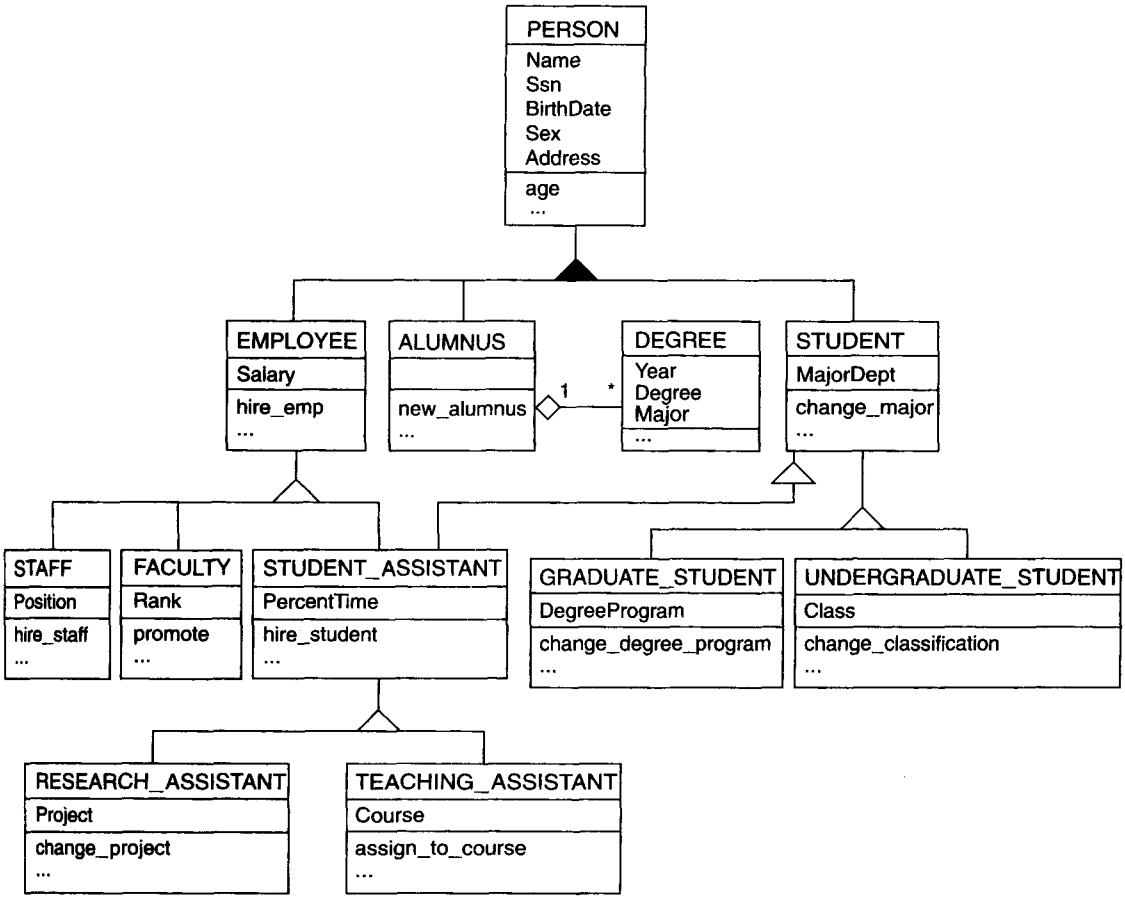

FIGURE 4.10 A UML class diagram corresponding to the EER diagram in Figure 4.7, illustrating UML notation for specialization/generalization.

In database design, we are mainly concerned with specifying concrete classes whose collections of objects are permanently (or persistently) stored in the database. The bibliographic notes at the end of this chapter give some references to books that describe complete details of UML. Additional material related to UML is covered in Chapter 12, and object modeling in general is further discussed in Chapter 20.

4.7 Relationship Types of Degree Higher Than Two

In Section 3.4.2 we defined the **degree** of a relationship type as the number of participating entity types and called a relationship type of degree two *binary* and a relationship type of degree three *ternary*. In this section, we elaborate on the differences between binary

and higher-degree relationships, when to choose higher-degree or binary relationships, and constraints on higher-degree relationships.

4.7.1 Choosing between Binary and Ternary (or Higher-Degree) Relationships

The ER diagram notation for a ternary relationship type is shown in Figure 4.11a, which displays the schema for the SUPPLY relationship type that was displayed at the instance level in Figure 3.10. Recall that the relationship set of SUPPLY is a set of relationship instances (s, j, p), where s is a SUPPLIER who is currently supplying a PART p to a PROJECT j. In general, a relationship type R of degree n will have n edges in an ER diagram, one connecting R to each participating entity type.

Figure 4.11b shows an ER diagram for the three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. In general, a ternary relationship type represents different information than do three binary relationship types. Consider the three binary relationship types CAN_SUPPLY, USES, and SUPPLIES. Suppose that CAN_SUPPLY, between SUPPLIER and PART, includes an instance (s, p) whenever supplier s *can supply* part p (to any project); USES, between PROJECT and PART, includes an instance (j, p) whenever project j *uses* part p; and SUPPLIES, between SUPPLIER and PROJECT, includes an instance (s, j) whenever supplier s *supplies some part* to project j. The existence of three relationship instances (s, p), (j, p), and (s, j) in CAN_SUPPLY, USES, and SUPPLIES, respectively, does not necessarily imply that an instance (s, j, p) exists in the ternary relationship SUPPLY, because the *meaning is different.* It is often tricky to decide whether a particular relationship should be represented as a relationship type of degree n or should be broken down into several relationship types of smaller degrees. The designer must base this decision on the semantics or meaning of the particular situation being represented. The typical solution is to include the ternary relationship *plus* one or more of the binary relationships, if they represent different meanings and if all are needed by the application.

Some database design tools are based on variations of the ER model that permit only binary relationships. In this case, a ternary relationship such as SUPPLY must be represented as a weak entity type, with no partial key and with three identifying relationships. The three participating entity types SUPPLIER, PART, and PROJECT are together the owner entity types (see Figure 4.11c). Hence, an entity in the weak entity type SUPPLY of Figure 4.11c is identified by the combination of its three owner entities from SUPPLIER, PART, and PROJECT.

Another example is shown in Figure 4.12. The ternary relationship type OFFERS represents information on instructors offering courses during particular semesters; hence it includes a relationship instance (i, s, c) whenever INSTRUCTOR i offers COURSE c during SEMESTER s. The three binary relationship types shown in Figure 4.12 have the following meanings: CAN_TEACH relates a course to the instructors who *can teach* that course, TAUGHT_DURING relates a semester to the instructors who *taught some course* during that semester, and OFFERED_DURING relates a semester to the courses offered during that semester *by any instructor.* These ternary and binary relationships represent different information, but certain constraints should hold among the relationships. For example, a relationship instance (i, s, c) should not exist in OFFERS *unless* an instance (i, s) exists in TAUGHT_DURING,

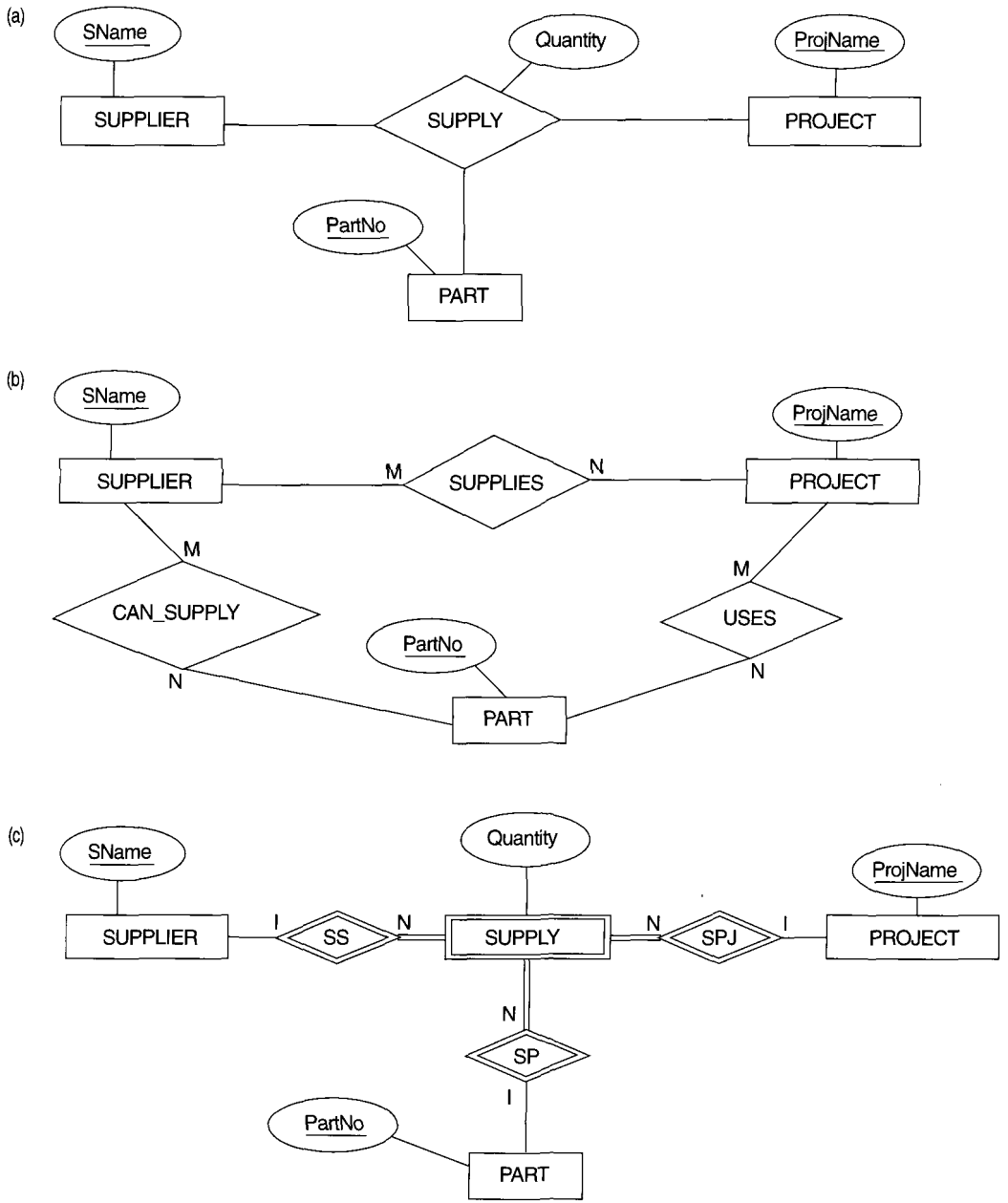

FIGURE 4.11 Ternary relationship types. (a) The SUPPLY relationship. (b) Three binary relationships not equivalent to SUPPLY. (c) SUPPLY represented as a weak entity type.

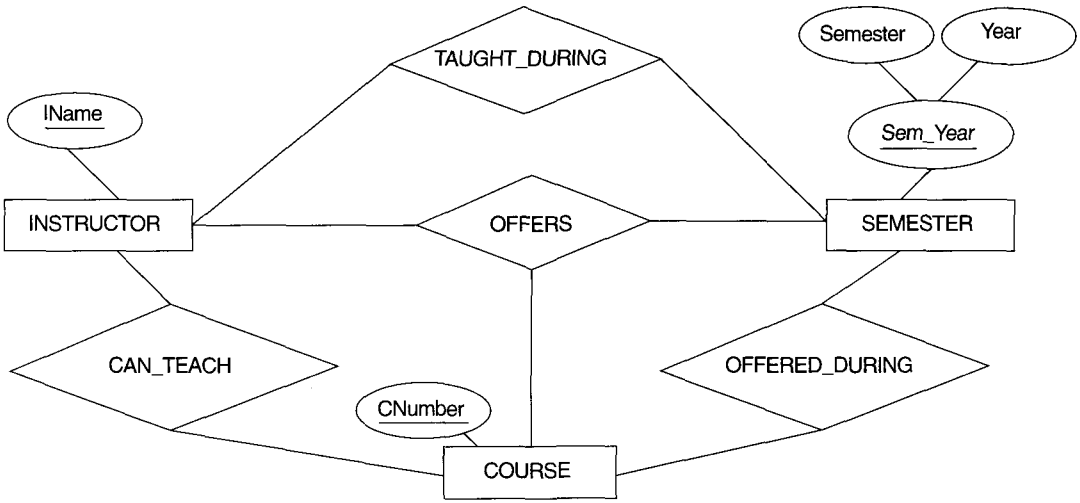

FIGURE 4.12 Another example of ternary versus binary relationship types.

an instance (s, c) exists in OFFERED_DURING, and an instance (i, c) exists in CAN_TEACH. However, the reverse is not always true; we may have instances (i, s), (s, c), and (i, c) in the three binary relationship types with no corresponding instance (i, s, c) in OFFERS. Note that in this example, based on the meanings of the relationships, we can infer the instances of TAUGHT_DURING and OFFERED_DURING from the instances in OFFERS, but we cannot infer the instances of CAN_TEACH; therefore, TAUGHT_DURING and OFFERED_DURING are redundant and can be left out.

Although in general three binary relationships cannot replace a ternary relationship, they may do so under certain *additional constraints*. In our example, if the CAN_TEACH relationship is 1:1 (an instructor can teach one course, and a course can be taught by only one instructor), then the ternary relationship OFFERS can be left out because it can be inferred from the three binary relationships CAN_TEACH, TAUGHT_DURING, and OFFERED_DURING. The schema designer must analyze the meaning of each specific situation to decide which of the binary and ternary relationship types are needed.

Notice that it is possible to have a weak entity type with a ternary (or n-ary) identifying relationship type. In this case, the weak entity type can have *several* owner entity types. An example is shown in Figure 4.13.

4.7.2 Constraints on Ternary (or Higher-Degree) Relationships

There are two notations for specifying structural constraints on n-ary relationships, and they specify different constraints. They should thus *both be used* if it is important to fully specify the structural constraints on a ternary or higher-degree relationship. The first

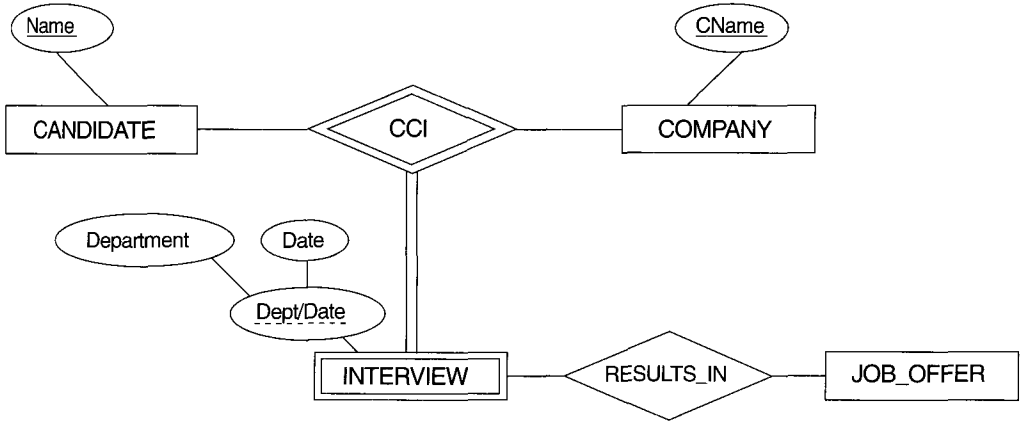

FIGURE 4.13 A weak entity type INTERVIEW with a ternary identifying relationship type.

notation is based on the cardinality ratio notation of binary relationships displayed in Figure 3.2. Here, a 1, M, or N is specified on each participation arc (both M and N symbols stand for *many* or *any number*).[12] Let us illustrate this constraint using the SUPPLY relationship in Figure 4.11.

Recall that the relationship set of SUPPLY is a set of relationship instances (s, j, p), where s is a SUPPLIER, j is a PROJECT, and p is a PART. Suppose that the constraint exists that for a particular project-part combination, only one supplier will be used (only one supplier supplies a particular part to a particular project). In this case, we place 1 on the SUPPLIER participation, and M, N on the PROJECT, PART participations in Figure 4.11. This specifies the constraint that a particular (j, p) combination can appear *at most once* in the relationship set because each such (project, part) combination uniquely determines a single supplier. Hence, any relationship instance (s, j, p) is *uniquely identified* in the relationship set by its (j, p) combination, which makes (j, p) a key for the relationship set. In this notation, the participations that have a one specified on them are not required to be part of the identifying key for the relationship set.[13]

The second notation is based on the (min, max) notation displayed in Figure 3.15 for binary relationships. A (min, max) on a participation here specifies that each entity is related to at least *min* and at most *max* relationship instances in the relationship set. These constraints have no bearing on determining the key of an n-ary relationship, where $n > 2$,[14] but specify a different type of constraint that places restrictions on how many relationship instances each entity can participate in.

12. This notation allows us to determine the key of the *relationship relation*, as we discuss in Chapter 7.

13. This is also true for cardinality ratios of binary relationships.

14. The (min, max) constraints can determine the keys for binary relationships, though.

4.8 Data Abstraction, Knowledge Representation, and Ontology Concepts

In this section we discuss in abstract terms some of the modeling concepts that we described quite specifically in our presentation of the ER and EER models in Chapter 3 and earlier in this chapter. This terminology is used both in conceptual data modeling and in artificial intelligence literature when discussing **knowledge representation** (abbreviated as **KR**). The goal of KR techniques is to develop concepts for accurately modeling some **domain of knowledge** by creating an **ontology**[15] that describes the concepts of the domain. This is then used to store and manipulate knowledge for drawing inferences, making decisions, or just answering questions. The goals of KR are similar to those of semantic data models, but there are some important similarities and differences between the two disciplines:

- Both disciplines use an abstraction process to identify common properties and important aspects of objects in the miniworld (domain of discourse) while suppressing insignificant differences and unimportant details.
- Both disciplines provide concepts, constraints, operations, and languages for defining data and representing knowledge.
- KR is generally broader in scope than semantic data models. Different forms of knowledge, such as rules (used in inference, deduction, and search), incomplete and default knowledge, and temporal and spatial knowledge, are represented in KR schemes. Database models are being expanded to include some of these concepts (see Chapter 24).
- KR schemes include **reasoning mechanisms** that deduce additional facts from the facts stored in a database. Hence, whereas most current database systems are limited to answering direct queries, knowledge-based systems using KR schemes can answer queries that involve **inferences** over the stored data. Database technology is being extended with inference mechanisms (see Section 24.4).
- Whereas most data models concentrate on the representation of database schemas, or meta-knowledge, KR schemes often mix up the schemas with the instances themselves in order to provide flexibility in representing exceptions. This often results in inefficiencies when these KR schemes are implemented, especially when compared with databases and when a large amount of data (or facts) needs to be stored.

In this section we discuss four **abstraction concepts** that are used in both semantic data models, such as the EER model, and KR schemes: (1) classification and instantiation, (2) identification, (3) specialization and generalization, and (4) aggregation and association. The paired concepts of classification and instantiation are inverses of one another, as are generalization and specialization. The concepts of aggregation and association are also related. We discuss these abstract concepts and their relation to the concrete representations used in the EER model to clarify the data abstraction process and

15. An *ontology* is somewhat similar to a conceptual schema, but with more knowledge, rules, and exceptions.

to improve our understanding of the related process of conceptual schema design. We close the section with a brief discussion of the term *ontology*, which is being used widely in recent knowledge representation research.

4.8.1 Classification and Instantiation

The process of **classification** involves systematically assigning similar objects/entities to object classes/entity types. We can now describe (in DB) or reason about (in KR) the classes rather than the individual objects. Collections of objects share the same types of attributes, relationships, and constraints, and by classifying objects we simplify the process of discovering their properties. **Instantiation** is the inverse of classification and refers to the generation and specific examination of distinct objects of a class. Hence, an object instance is related to its object class by the IS-AN-INSTANCE-OF or IS-A-MEMBER-OF relationship. Although UML diagrams do not display instances, the UML diagrams allow a form of instantiation by permitting the display of individual objects. We *did not* describe this feature in our introduction to UML.

In general, the objects of a class should have a similar type structure. However, some objects may display properties that differ in some respects from the other objects of the class; these **exception objects** also need to be modeled, and KR schemes allow more varied exceptions than do database models. In addition, certain properties apply to the class as a whole and not to the individual objects; KR schemes allow such **class properties.** UML diagrams also allow specification of class properties.

In the EER model, entities are classified into entity types according to their basic attributes and relationships. Entities are further classified into subclasses and categories based on additional similarities and differences (exceptions) among them. Relationship instances are classified into relationship types. Hence, entity types, subclasses, categories, and relationship types are the different types of classes in the EER model. The EER model does not provide explicitly for class properties, but it may be extended to do so. In UML, objects are classified into classes, and it is possible to display both class properties and individual objects.

Knowledge representation models allow multiple classification schemes in which one class is an *instance* of another class (called a **meta-class**). Notice that this *cannot* be represented directly in the EER model, because we have only two levels—classes and instances. The only relationship among classes in the EER model is a superclass/subclass relationship, whereas in some KR schemes an additional class/instance relationship can be represented directly in a class hierarchy. An instance may itself be another class, allowing multiple-level classification schemes.

4.8.2 Identification

Identification is the abstraction process whereby classes and objects are made uniquely identifiable by means of some **identifier.** For example, a class name uniquely identifies a whole class. An additional mechanism is necessary for telling distinct object instances

apart by means of object identifiers. Moreover, it is necessary to identify multiple manifestations in the database of the same real-world object. For example, we may have a tuple <Matthew Clarke, 610618, 376-9821> in a PERSON relation and another tuple <301-54-0836, CS, 3.8> in a STUDENT relation that happen to represent the same real-world entity. There is no way to identify the fact that these two database objects (tuples) represent the same real-world entity unless we make a provision *at design time* for appropriate cross-referencing to supply this identification. Hence, identification is needed at two levels:

- To distinguish among database objects and classes
- To identify database objects and to relate them to their real-world counterparts

In the EER model, identification of schema constructs is based on a system of unique names for the constructs. For example, every class in an EER schema—whether it is an entity type, a subclass, a category, or a relationship type—must have a distinct name. The names of attributes of a given class must also be distinct. Rules for unambiguously identifying attribute name references in a specialization or generalization lattice or hierarchy are needed as well.

At the object level, the values of key attributes are used to distinguish among entities of a particular entity type. For weak entity types, entities are identified by a combination of their own partial key values and the entities they are related to in the owner entity type(s). Relationship instances are identified by some combination of the entities that they relate, depending on the cardinality ratio specified.

4.8.3 Specialization and Generalization

Specialization is the process of classifying a class of objects into more specialized subclasses. Generalization is the inverse process of generalizing several classes into a higher-level abstract class that includes the objects in all these classes. Specialization is conceptual refinement, whereas generalization is conceptual synthesis. Subclasses are used in the EER model to represent specialization and generalization. We call the relationship between a subclass and its superclass an **IS-A-SUBCLASS-OF** relationship, or simply an **IS-A** relationship.

4.8.4 Aggregation and Association

Aggregation is an abstraction concept for building composite objects from their component objects. There are three cases where this concept can be related to the EER model. The first case is the situation in which we aggregate attribute values of an object to form the whole object. The second case is when we represent an aggregation relationship as an ordinary relationship. The third case, which the EER model does not provide for explicitly, involves the possibility of combining objects that are related by a particular relationship instance into a *higher-level aggregate object*. This is sometimes useful when the higher-level aggregate object is itself to be related to another object. We call the relation-

ship between the primitive objects and their aggregate object **IS-A-PART-OF;** the inverse is called **IS-A-COMPONENT-OF.** UML provides for all three types of aggregation.

The abstraction of **association** is used to associate objects from several *independent classes*. Hence, it is somewhat similar to the second use of aggregation. It is represented in the EER model by relationship types, and in UML by associations. This abstract relationship is called **IS-ASSOCIATED-WITH.**

In order to understand the different uses of aggregation better, consider the ER schema shown in Figure 4.14a, which stores information about interviews by job applicants to various companies. The class COMPANY is an aggregation of the attributes (or component objects) CName (company name) and CAddress (company address), whereas JOB_APPLICANT is an aggregate of Ssn, Name, Address, and Phone. The relationship attributes ContactName and ContactPhone represent the name and phone number of the person in the company who is responsible for the interview. Suppose that some interviews result in job offers, whereas others do not. We would like to treat INTERVIEW as a class to associate it with JOB_OFFER. The schema shown in Figure 4.14b is *incorrect* because it requires each interview relationship instance to have a job offer. The schema shown in Figure 4.14c is not allowed, because the ER model does not allow relationships among relationships (although UML does).

One way to represent this situation is to create a higher-level aggregate class composed of COMPANY, JOB_APPLICANT, and INTERVIEW and to relate this class to JOB_OFFER, as shown in Figure 4.14d. Although the EER model as described in this book does not have this facility, some semantic data models do allow it and call the resulting object a **composite** or **molecular object.** Other models treat entity types and relationship types uniformly and hence permit relationships among relationships, as illustrated in Figure 4.14c.

To represent this situation correctly in the ER model as described here, we need to create a new weak entity type INTERVIEW, as shown in Figure 4.14e, and relate it to JOB_OFFER. Hence, we can always represent these situations correctly in the ER model by creating additional entity types, although it may be conceptually more desirable to allow direct representation of aggregation, as in Figure 4.14d, or to allow relationships among relationships, as in Figure 4.14c.

The main structural distinction between aggregation and association is that when an association instance is deleted, the participating objects may continue to exist. However, if we support the notion of an aggregate object—for example, a CAR that is made up of objects ENGINE, CHASSIS, and TIRES—then deleting the aggregate CAR object amounts to deleting all its component objects.

4.8.5 Ontologies and the Semantic Web

In recent years, the amount of computerized data and information available on the Web has spiraled out of control. Many different models and formats are used. In addition to the database models that we present in this book, much information is stored in the form of **documents,** which have considerably less structure than database information does. One research project that is attempting to allow information exchange among computers on the Web is called the **Semantic Web,** which attempts to create knowledge representation

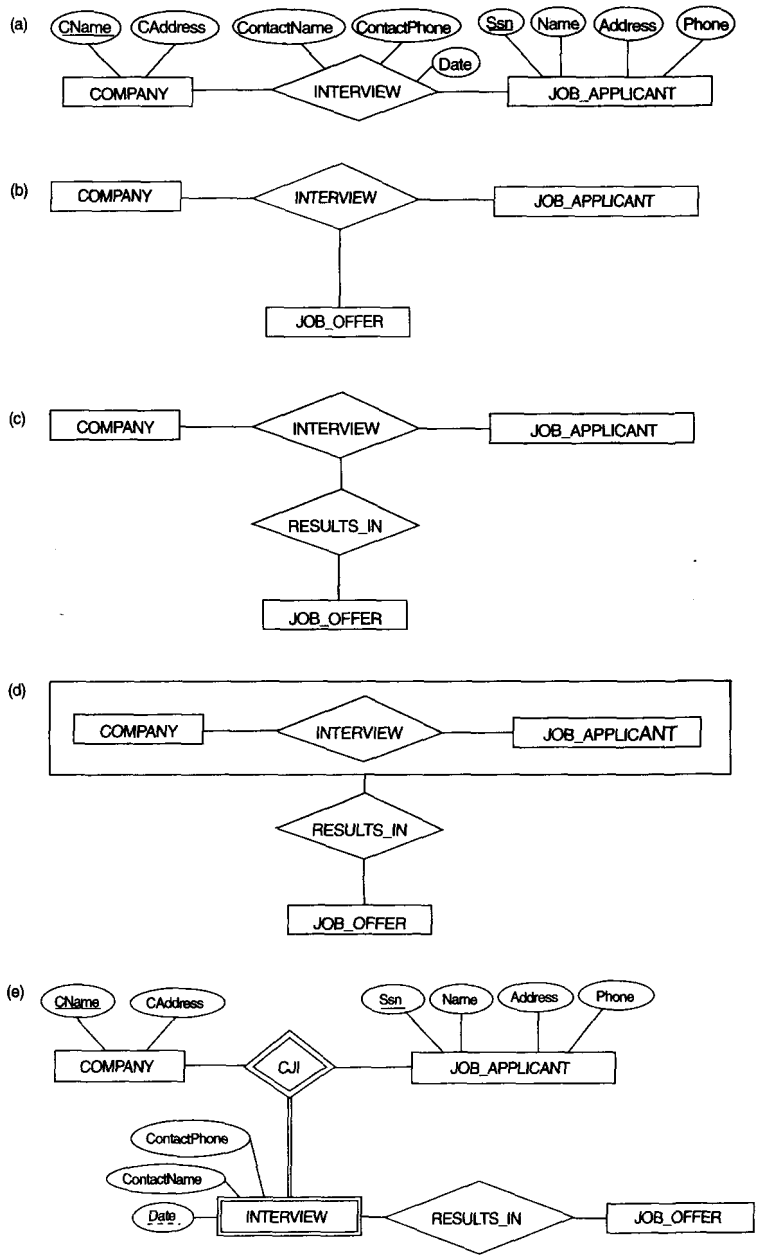

FIGURE 4.14 Aggregation. (a) The relationship type INTERVIEW. (b) Including JOB_OFFER in a ternary relationship type (incorrect). (c) Having the RESULTS_IN relationship participate in other relationships (generally not allowed in ER). (d) Using aggregation and a composite (molecular) object (generally not allowed in ER). (e) Correct representation in ER.

models that are quite general in order to to allow meaningful information exchange and search among machines. The concept of *ontology* is considered to be the most promising basis for achieving the goals of the Semantic Web, and is closely related to knowledge representation. In this section, we give a brief introduction to what an ontology is and how it can be used as a basis to automate information understanding, search, and exchange.

The study of ontologies attempts to describe the structures and relationships that are possible in reality through some common vocabulary, and so it can be considered as a way to describe the knowledge of a certain community about reality. Ontology originated in the fields of philosophy and metaphysics. One commonly used definition of **ontology** is "a *specification* of a *conceptualization*."[16]

In this definition, a **conceptualization** is the set of concepts that are used to represent the part of reality or knowledge that is of interest to a community of users. **Specification** refers to the language and vocabulary terms that are used to specify the conceptualization. The ontology includes both *specification* and *conceptualization*. For example, the same conceptualization may be specified in two different languages, giving two separate ontologies. Based on this quite general definition, there is no consensus on what exactly an ontology is. Some possible techniques to describe ontologies that have been mentioned are as follows:

- A **thesaurus** (or even a **dictionary** or a **glossary** of terms) describes the relationships between words (vocabulary) that represent various concepts.
- A **taxonomy** describes how concepts of a particular area of knowledge are related using structures similar to those used in a specialization or generalization.
- A detailed **database schema** is considered by some to be an ontology that describes the concepts (entities and attributes) and relationships of a miniworld from reality.
- A **logical theory** uses concepts from mathematical logic to try to define concepts and their interrelationships.

Usually the concepts used to describe ontologies are quite similar to the concepts we discussed in conceptual modeling, such as entities, attributes, relationships, specializations, and so on. The main difference between an ontology and, say, a database schema is that the schema is usually limited to describing a small subset of a miniworld from reality in order to store and manage data. An ontology is usually considered to be more general in that it should attempt to describe a part of reality as completely as possible.

4.9 SUMMARY

In this chapter we first discussed extensions to the ER model that improve its representational capabilities. We called the resulting model the enhanced ER or EER model. The concept of a subclass and its superclass and the related mechanism of attribute/relationship inheritance were presented. We saw how it is sometimes necessary to create additional

16. This definition is given in Gruber (1995).

classes of entities, either because of additional specific attributes or because of specific relationship types. We discussed two main processes for defining superclass/subclass hierarchies and lattices: specialization and generalization.

We then showed how to display these new constructs in an EER diagram. We also discussed the various types of constraints that may apply to specialization or generalization. The two main constraints are total/partial and disjoint/overlapping. In addition, a defining predicate for a subclass or a defining attribute for a specialization may be specified. We discussed the differences between user-defined and predicate-defined subclasses and between user-defined and attribute-defined specializations. Finally, we discussed the concept of a category or union type, which is a subset of the union of two or more classes, and we gave formal definitions of all the concepts presented.

We then introduced some of the notation and terminology of UML for representing specialization and generalization. We also discussed some of the issues concerning the difference between binary and higher-degree relationships, under which circumstances each should be used when designing a conceptual schema, and how different types of constraints on n-ary relationships may be specified. In Section 4.8 we discussed briefly the discipline of knowledge representation and how it is related to semantic data modeling. We also gave an overview and summary of the types of abstract data representation concepts: classification and instantiation, identification, specialization and generalization, and aggregation and association. We saw how EER and UML concepts are related to each of these.

Review Questions

4.1. What is a subclass? When is a subclass needed in data modeling?

4.2. Define the following terms: *superclass of a subclass*, *superclass/subclass relationship*, is-a *relationship*, *specialization*, *generalization*, *category*, *specific (local) attributes*, *specific relationships*.

4.3. Discuss the mechanism of attribute/relationship inheritance. Why is it useful?

4.4. Discuss user-defined and predicate-defined subclasses, and identify the differences between the two.

4.5. Discuss user-defined and attribute-defined specializations, and identify the differences between the two.

4.6. Discuss the two main types of constraints on specializations and generalizations.

4.7. What is the difference between a specialization hierarchy and a specialization lattice?

4.8. What is the difference between specialization and generalization? Why do we not display this difference in schema diagrams?

4.9. How does a category differ from a regular shared subclass? What is a category used for? Illustrate your answer with examples.

4.10. For each of the following UML terms (see Sections 3.8 and 4.6), discuss the corresponding term in the EER model, if any: *object*, *class*, *association*, *aggregation*, *generalization*, *multiplicity*, *attributes*, *discriminator*, *link*, *link attribute*, *reflexive association*, *qualified association*.

4.11. Discuss the main differences between the notation for EER schema diagrams and UML class diagrams by comparing how common concepts are represented in each.

4.12. Discuss the two notations for specifying constraints on n-ary relationships, and what each can be used for.
4.13. List the various data abstraction concepts and the corresponding modeling concepts in the EER model.
4.14. What aggregation feature is missing from the EER model? How can the EER model be further enhanced to support it?
4.15. What are the main similarities and differences between conceptual database modeling techniques and knowledge representation techniques?
4.16. Discuss the similarities and differences between an ontology and a database schema.

Exercises

4.17. Design an EER schema for a database application that you are interested in. Specify all constraints that should hold on the database. Make sure that the schema has at least five entity types, four relationship types, a weak entity type, a superclass/subclass relationship, a category, and an n-ary ($n > 2$) relationship type.
4.18. Consider the BANK ER schema of Figure 3.18, and suppose that it is necessary to keep track of different types of ACCOUNTS (SAVINGS_ACCTS, CHECKING_ACCTS, . . .) and LOANS (CAR_LOANS, HOME_LOANS, . . .). Suppose that it is also desirable to keep track of each account's TRANSACTIONS (deposits, withdrawals, checks, . . .) and each loan's PAYMENTS; both of these include the amount, date, and time. Modify the BANK schema, using ER and EER concepts of specialization and generalization. State any assumptions you make about the additional requirements.
4.19. The following narrative describes a simplified version of the organization of Olympic facilities planned for the summer Olympics. Draw an EER diagram that shows the entity types, attributes, relationships, and specializations for this application. State any assumptions you make. The Olympic facilities are divided into sports complexes. Sports complexes are divided into *one-sport* and *multisport* types. Multisport complexes have areas of the complex designated for each sport with a location indicator (e.g., center, NE corner, etc.). A complex has a location, chief organizing individual, total occupied area, and so on. Each complex holds a series of events (e.g., the track stadium may hold many different races). For each event there is a planned date, duration, number of participants, number of officials, and so on. A roster of all officials will be maintained together with the list of events each official will be involved in. Different equipment is needed for the events (e.g., goal posts, poles, parallel bars) as well as for maintenance. The two types of facilities (one-sport and multisport) will have different types of information. For each type, the number of facilities needed is kept, together with an approximate budget.
4.20. Identify all the important concepts represented in the library database case study described here. In particular, identify the abstractions of classification (entity types and relationship types), aggregation, identification, and specialization/generalization. Specify (min, max) cardinality constraints whenever possible. List

details that will affect the eventual design but have no bearing on the conceptual design. List the semantic constraints separately. Draw an EER diagram of the library database.

Case Study: The Georgia Tech Library (GTL) has approximately 16,000 members, 100,000 titles, and 250,000 volumes (or an average of 2.5 copies per book). About 10 percent of the volumes are out on loan at any one time. The librarians ensure that the books that members want to borrow are available when the members want to borrow them. Also, the librarians must know how many copies of each book are in the library or out on loan at any given time. A catalog of books is available online that lists books by author, title, and subject area. For each title in the library, a book description is kept in the catalog that ranges from one sentence to several pages. The reference librarians want to be able to access this description when members request information about a book. Library staff is divided into chief librarian, departmental associate librarians, reference librarians, check-out staff, and library assistants.

Books can be checked out for 21 days. Members are allowed to have only five books out at a time. Members usually return books within three to four weeks. Most members know that they have one week of grace before a notice is sent to them, so they try to get the book returned before the grace period ends. About 5 percent of the members have to be sent reminders to return a book. Most overdue books are returned within a month of the due date. Approximately 5 percent of the overdue books are either kept or never returned. The most active members of the library are defined as those who borrow at least ten times during the year. The top 1 percent of membership does 15 percent of the borrowing, and the top 10 percent of the membership does 40 percent of the borrowing. About 20 percent of the members are totally inactive in that they are members but never borrow.

To become a member of the library, applicants fill out a form including their SSN, campus and home mailing addresses, and phone numbers. The librarians then issue a numbered, machine-readable card with the member's photo on it. This card is good for four years. A month before a card expires, a notice is sent to a member for renewal. Professors at the institute are considered automatic members. When a new faculty member joins the institute, his or her information is pulled from the employee records and a library card is mailed to his or her campus address. Professors are allowed to check out books for three-month intervals and have a two-week grace period. Renewal notices to professors are sent to the campus address.

The library does not lend some books, such as reference books, rare books, and maps. The librarians must differentiate between books that can be lent and those that cannot be lent. In addition, the librarians have a list of some books they are interested in acquiring but cannot obtain, such as rare or out-of-print books and books that were lost or destroyed but have not been replaced. The librarians must have a system that keeps track of books that cannot be lent as well as books that they are interested in acquiring. Some books may have the same title; therefore, the title cannot be used as a means of identification. Every book is identified by its International Standard Book Number (ISBN), a unique interna-

tional code assigned to all books. Two books with the same title can have different ISBNs if they are in different languages or have different bindings (hard cover or soft cover). Editions of the same book have different ISBNs.

The proposed database system must be designed to keep track of the members, the books, the catalog, and the borrowing activity.

4.21. Design a database to keep track of information for an art museum. Assume that the following requirements were collected:

- The museum has a collection of ART_OBJECTS. Each ART_OBJECT has a unique IdNo, an Artist (if known), a Year (when it was created, if known), a Title, and a Description. The art objects are categorized in several ways, as discussed below.
- ART_OBJECTS are categorized based on their type. There are three main types: PAINTING, SCULPTURE, and STATUE, plus another type called OTHER to accommodate objects that do not fall into one of the three main types.
- A PAINTING has a PaintType (oil, watercolor, etc.), material on which it is DrawnOn (paper, canvas, wood, etc.), and Style (modern, abstract, etc.).
- A SCULPTURE or a STATUE has a Material from which it was created (wood, stone, etc.), Height, Weight, and Style.
- An art object in the OTHER category has a Type (print, photo, etc.) and Style.
- ART_OBJECTS are also categorized as PERMANENT_COLLECTION, which are owned by the museum (these have information on the DateAcquired, whether it is OnDisplay or stored, and Cost) or BORROWED, which has information on the Collection (from which it was borrowed), DateBorrowed, and DateReturned.
- ART_OBJECTS also have information describing their country/culture using information on country/culture of Origin (Italian, Egyptian, American, Indian, etc.) and Epoch (Renaissance, Modern, Ancient, etc.).
- The museum keeps track of ARTIST's information, if known: Name, DateBorn (if known), DateDied (if not living), CountryOfOrigin, Epoch, MainStyle, and Description. The Name is assumed to be unique.
- Different EXHIBITIONS occur, each having a Name, StartDate, and EndDate. EXHIBITIONS are related to all the art objects that were on display during the exhibition.
- Information is kept on other COLLECTIONS with which the museum interacts, including Name (unique), Type (museum, personal, etc.), Description, Address, Phone, and current ContactPerson.

Draw an EER schema diagram for this application. Discuss any assumptions you made, and that justify your EER design choices.

4.22. Figure 4.15 shows an example of an EER diagram for a small private airport database that is used to keep track of airplanes, their owners, airport employees, and pilots. From the requirements for this database, the following information was collected: Each AIRPLANE has a registration number [Reg#], is of a particular plane type [OF_TYPE], and is stored in a particular hangar [STORED_IN]. Each PLANE_TYPE has a model number [Model], a capacity [Capacity], and a weight [Weight]. Each HANGAR has a number [Number], a capacity [Capacity], and a location [Location]. The database also keeps track of the OWNERS of each plane [OWNS] and the EMPLOYEES who

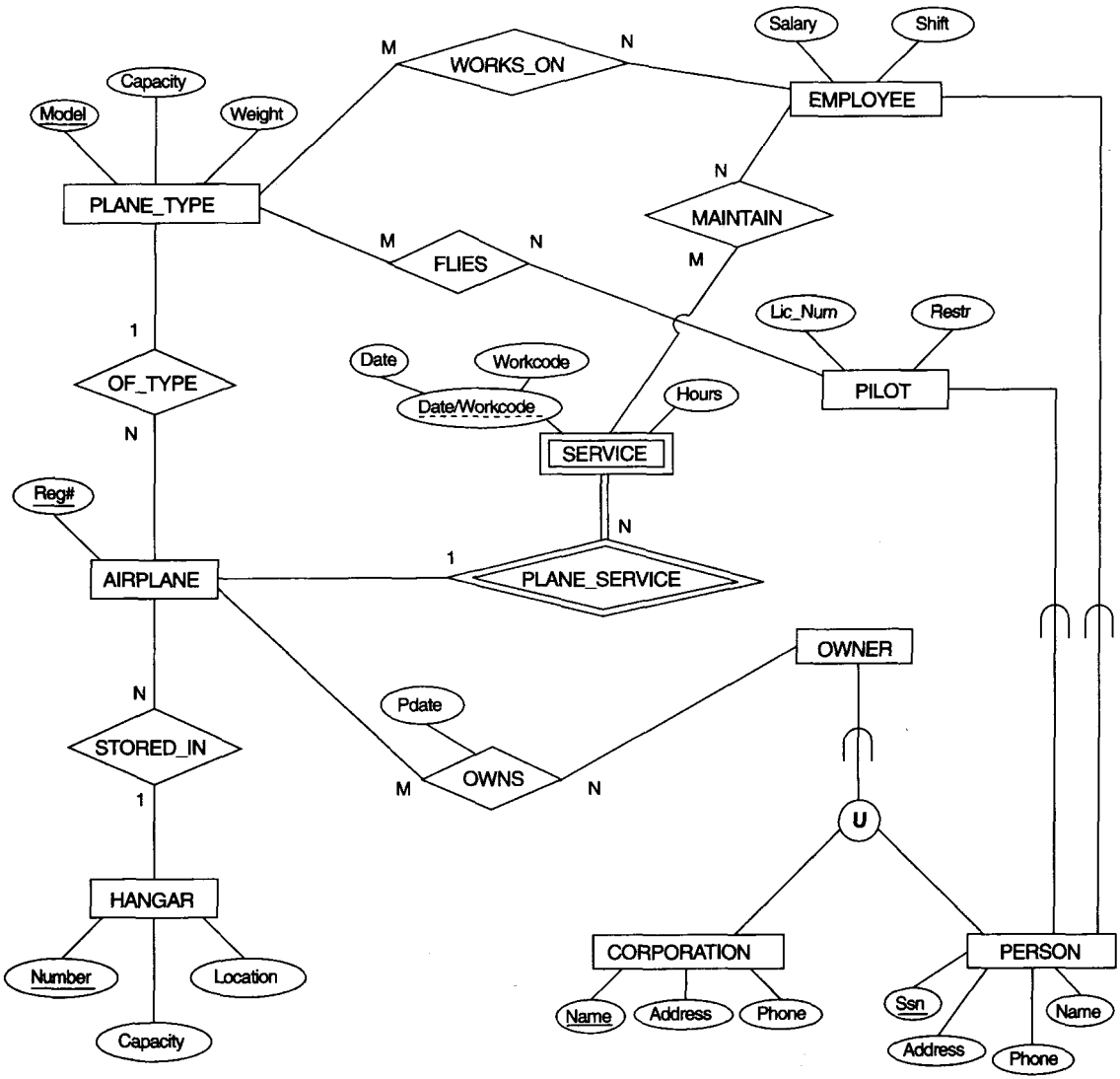

FIGURE 4.15 EER schema for a SMALL AIRPORT database.

have maintained the plane [MAINTAIN]. Each relationship instance in OWNS relates an airplane to an owner and includes the purchase date [Pdate]. Each relationship instance in MAINTAIN relates an employee to a service record [SERVICE]. Each plane undergoes service many times; hence, it is related by [PLANE_SERVICE] to a number of service records. A service record includes as attributes the date of maintenance [Date], the number of hours spent on the work [Hours], and the type of work done [Workcode]. We use a weak entity type [SERVICE] to represent airplane service,

because the airplane registration number is used to identify a service record. An owner is either a person or a corporation. Hence, we use a union type (category) [OWNER] that is a subset of the union of corporation [CORPORATION] and person [PERSON] entity types. Both pilots [PILOT] and employees [EMPLOYEE] are subclasses of PERSON. Each pilot has specific attributes license number [Lic_Num] and restrictions [Restr]; each employee has specific attributes salary [Salary] and shift worked [Shift]. All PERSON entities in the database have data kept on their social security number [Ssn], name [Name], address [Address], and telephone number [Phone]. For CORPORATION entities, the data kept includes name [Name], address [Address], and telephone number [Phone]. The database also keeps track of the types of planes each pilot is authorized to fly [FLIES] and the types of planes each employee can do maintenance work on [WORKS_ON]. Show how the SMALL AIRPORT EER schema of Figure 4.15 may be represented in UML notation. (*Note:* We have not discussed how to represent categories (union types) in UML, so you do not have to map the categories in this and the following question.)

4.23. Show how the UNIVERSITY EER schema of Figure 4.9 may be represented in UML notation.

Selected Bibliography

Many papers have proposed conceptual or semantic data models. We give a representative list here. One group of papers, including Abrial (1974), Senko's DIAM model (1975), the NIAM method (Verheijen and VanBekkum 1982), and Bracchi et al. (1976), presents semantic models that are based on the concept of binary relationships. Another group of early papers discusses methods for extending the relational model to enhance its modeling capabilities. This includes the papers by Schmid and Swenson (1975), Navathe and Schkolnick (1978), Codd's RM/T model (1979), Furtado (1978), and the structural model of Wiederhold and Elmasri (1979).

The ER model was proposed originally by Chen (1976) and is formalized in Ng (1981). Since then, numerous extensions of its modeling capabilities have been proposed, as in Scheuermann et al. (1979), Dos Santos et al. (1979), Teorey et al. (1986), Gogolla and Hohenstein (1991), and the entity-category-relationship (ECR) model of Elmasri et al. (1985). Smith and Smith (1977) present the concepts of generalization and aggregation. The semantic data model of Hammer and McLeod (1981) introduced the concepts of class/subclass lattices, as well as other advanced modeling concepts.

A survey of semantic data modeling appears in Hull and King (1987). Eick (1991) discusses design and transformations of conceptual schemas. Analysis of constraints for *n*-ary relationships is given in Soutou (1998). UML is described in detail in Booch, Rumbaugh, and Jacobson (1999). Fowler and Scott (2000) and Stevens and Pooley (2000) give concise introductions to UML concepts.

Fensel (2000) is a good reference on Semantic Web. Uschold and Gruninger (1996) and Gruber (1995) discuss ontologies. A recent entire issue of Communications of the ACM is devoted to ontology concepts and applications.