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

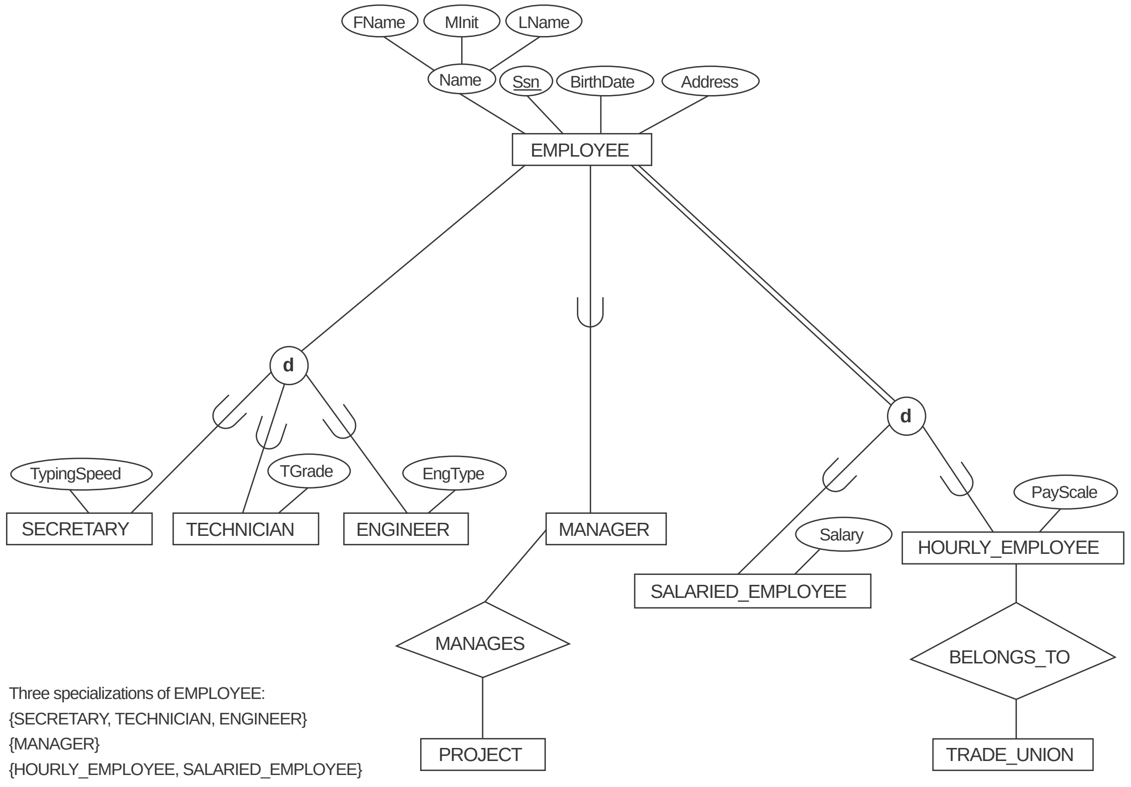

FIGURE 4.1 EER diagram notation to represent subclasses and specialization

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 EMPLOYEE/SECRETARY

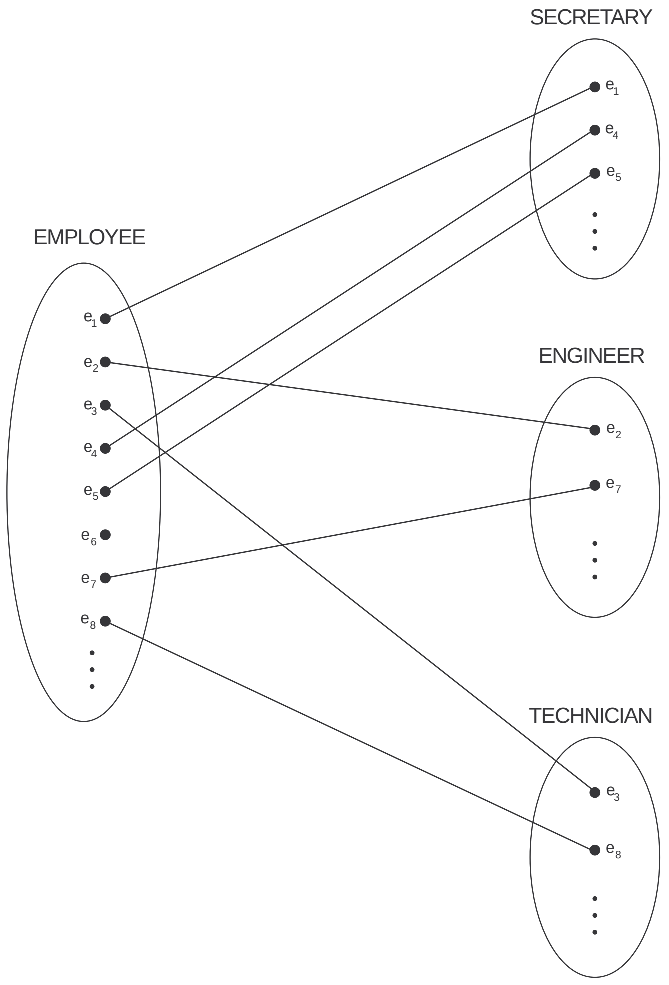

FIGURE 4.2 Instances of a specialization

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.

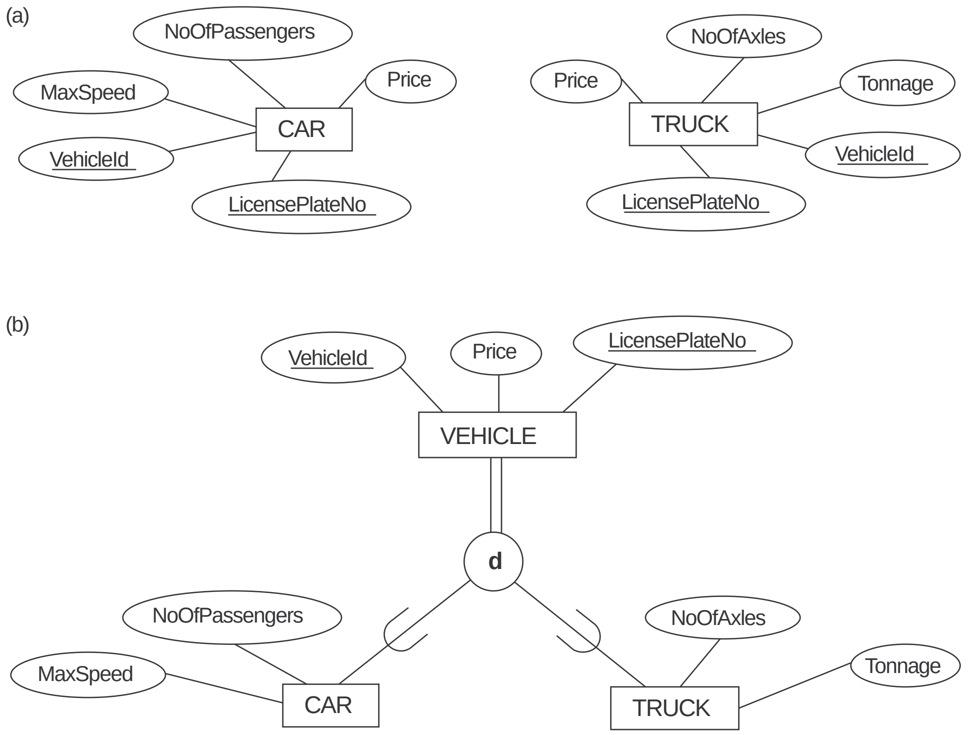

FIGURE 4.3 Generalization. (a) Two entity types, CAR and TRUCK. (b) Generalizing CAR and TRUCK into the superclass VEHICLE

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.

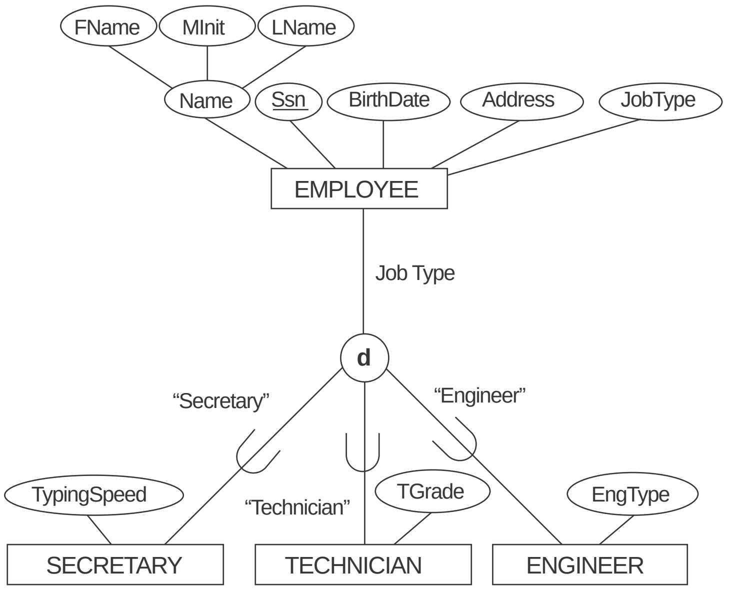

FIGURE 4.4 EER diagram notation for an attribute-defined specialization on JobType

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

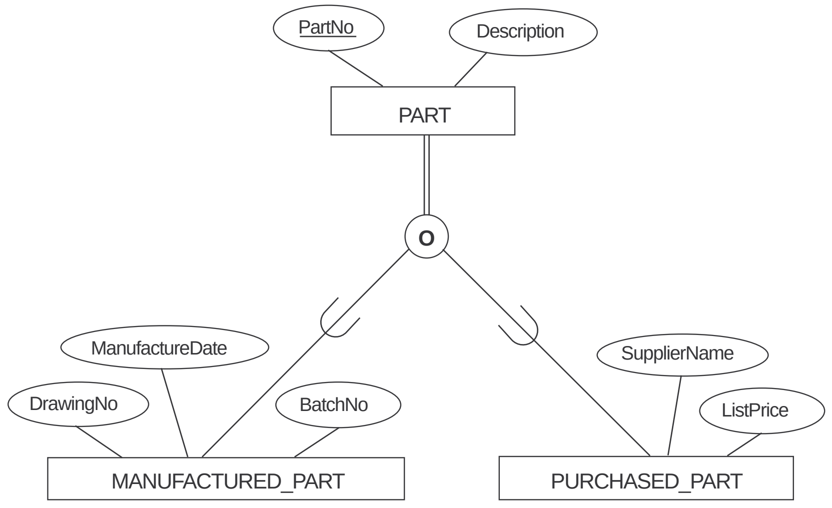

FIGURE 4.5 EER diagram notation for an overlapping (nondisjoint) specialization

belong 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.

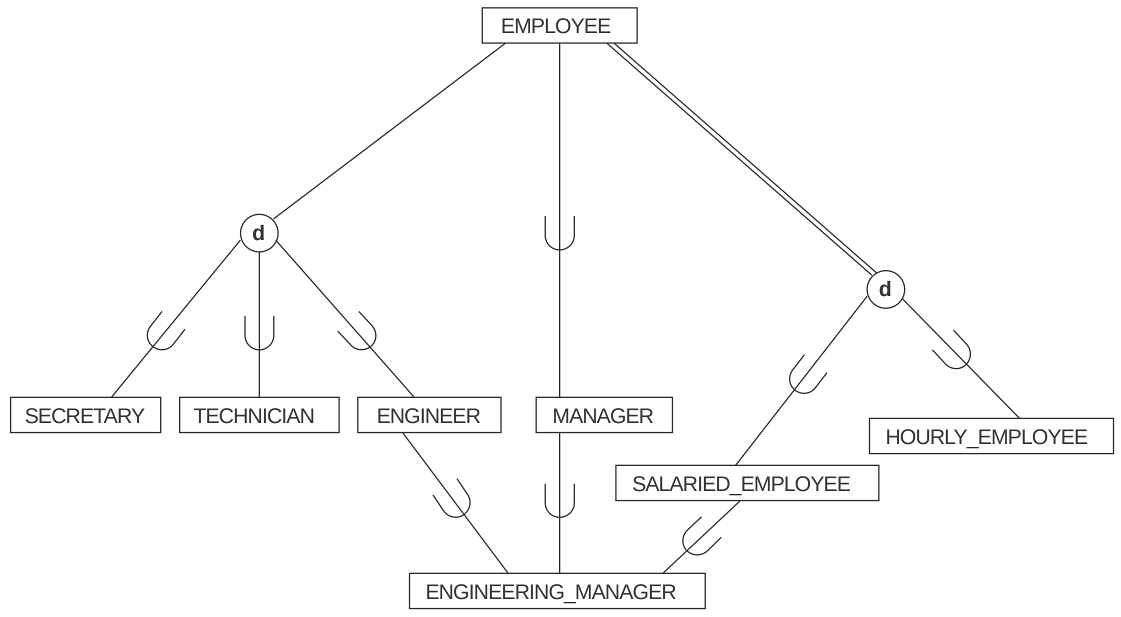

FIGURE 4.6 A specialization lattice with shared subclass ENGINEERING_MANAGER

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

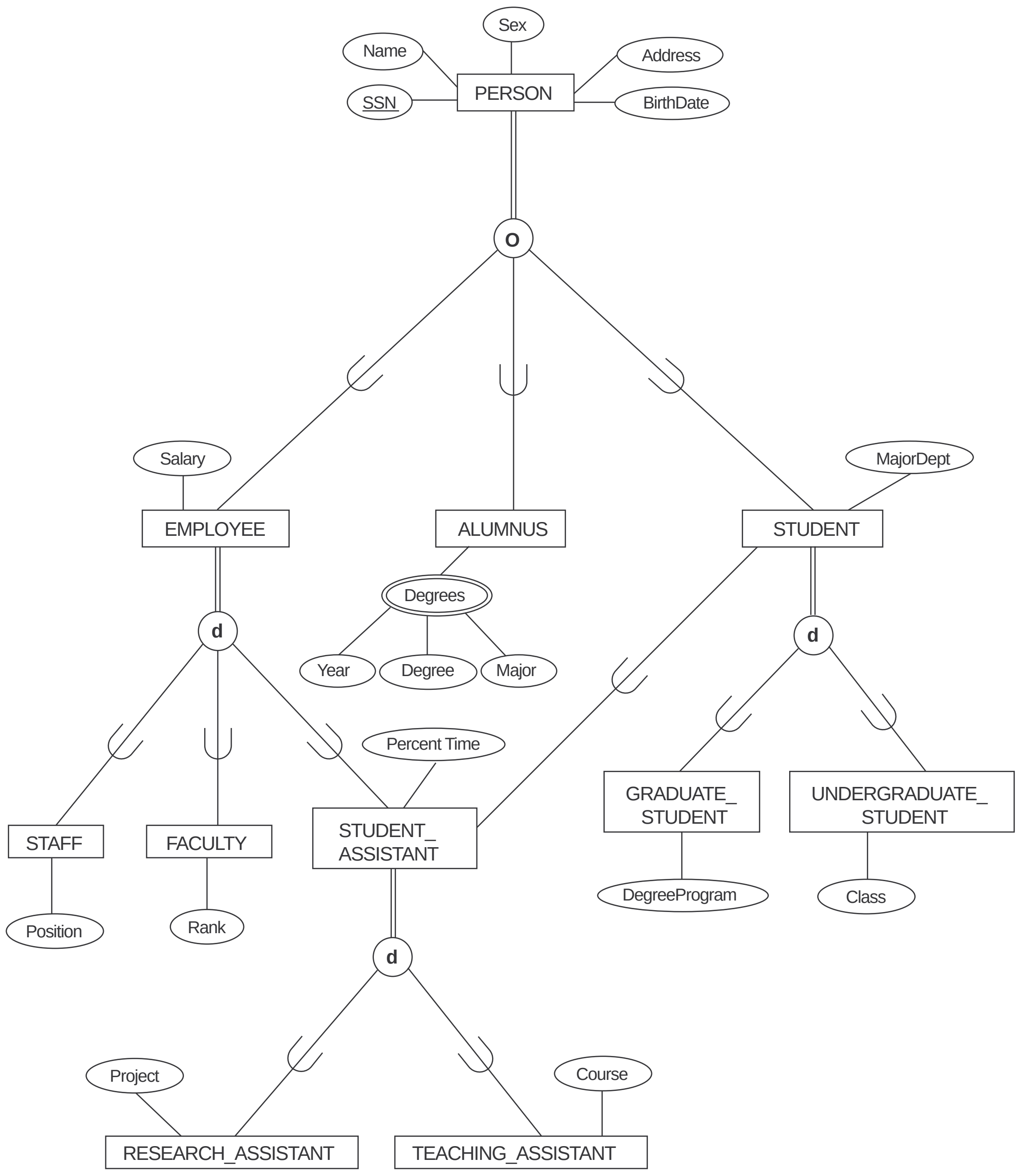

FIGURE 4.7 A specialization lattice with multiple inheritance for a UNIVERSITY database

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 conceptual

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

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

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

superclasses COMPANY, BANK, and PERSON are connected to the circle with the ∪ 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

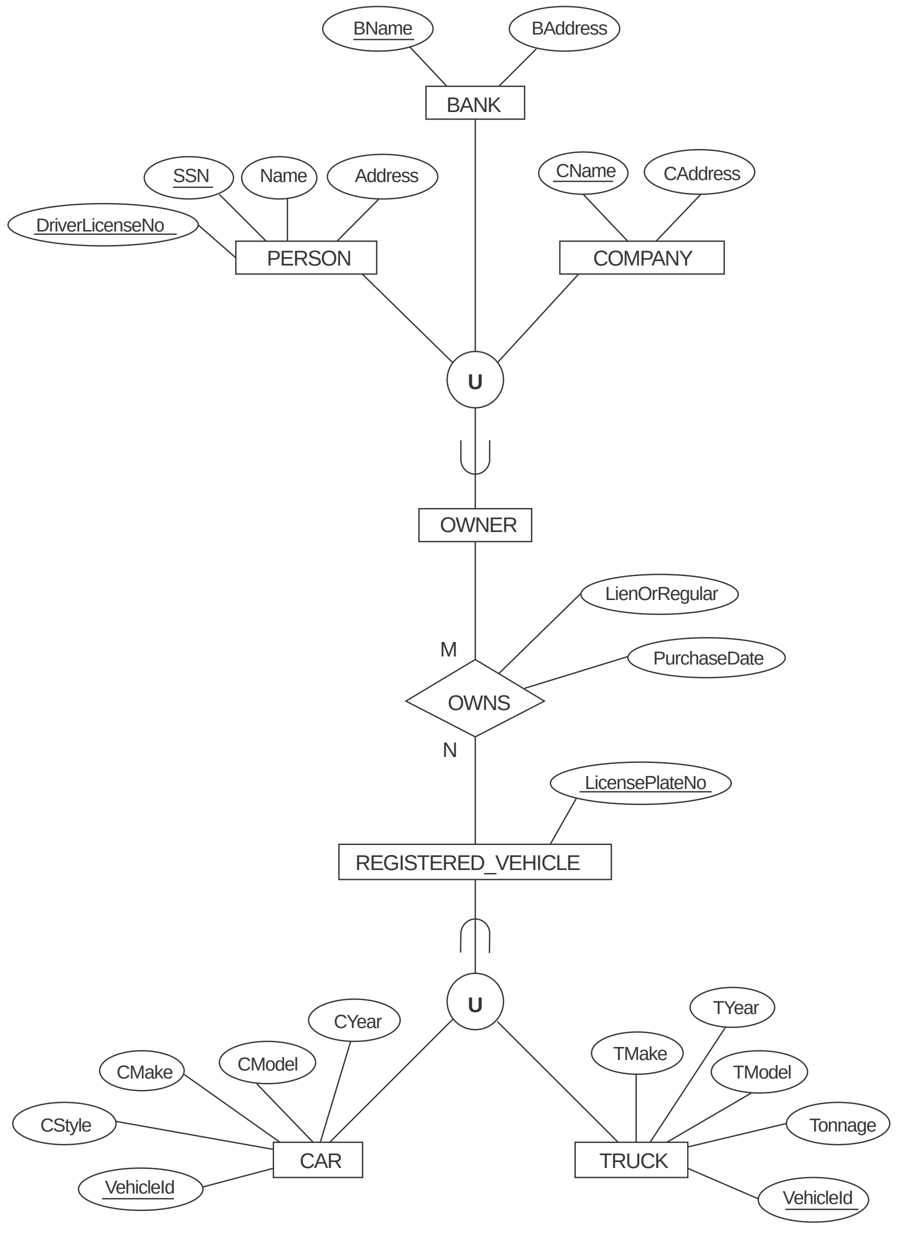

FIGURE 4.8 Two categories (union types): OWNER and REGISTERED_VEHICLE

the line from the 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.

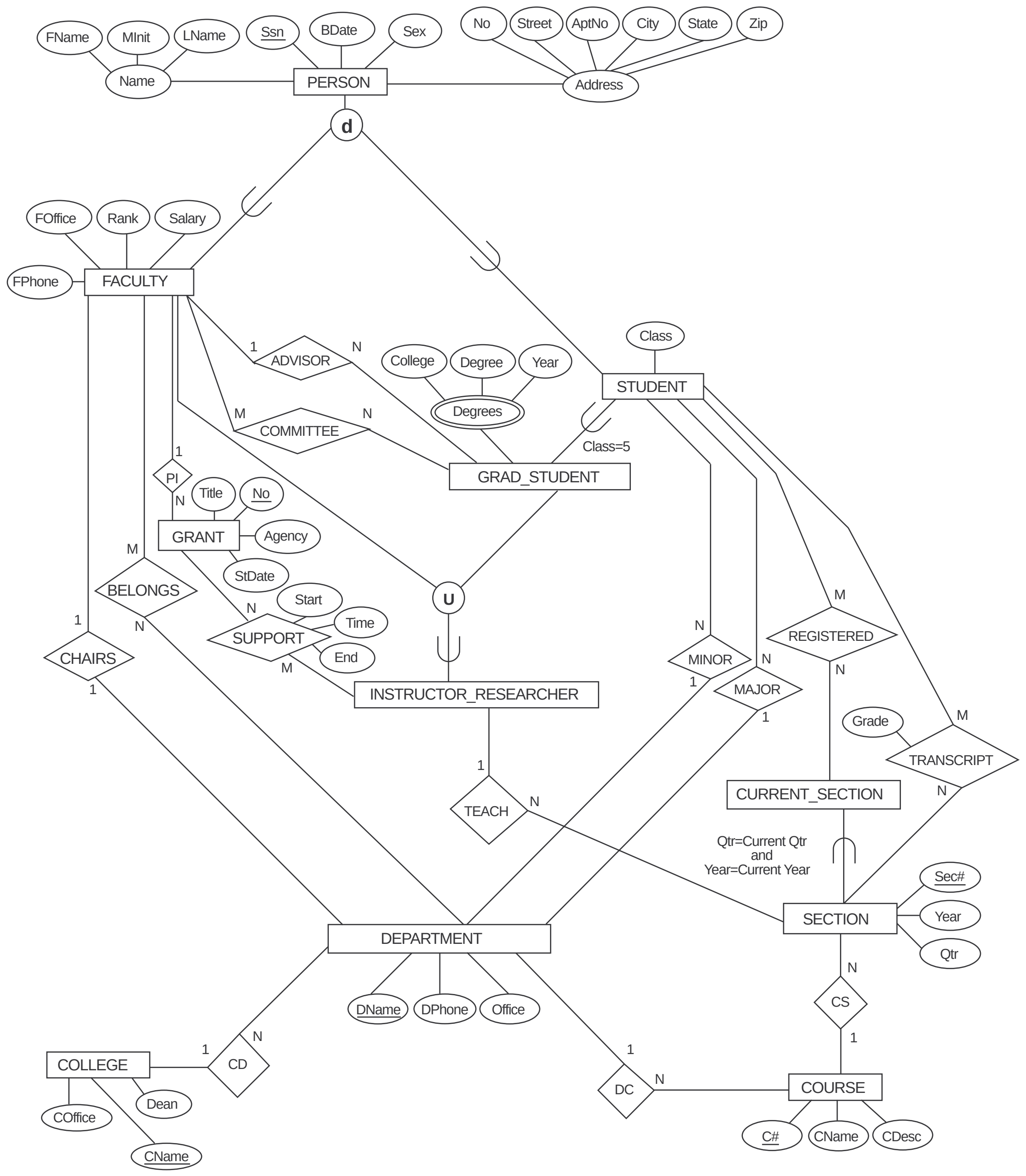

FIGURE 4.9 An EER conceptual schema for a UNIVERSITY database

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

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

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 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.

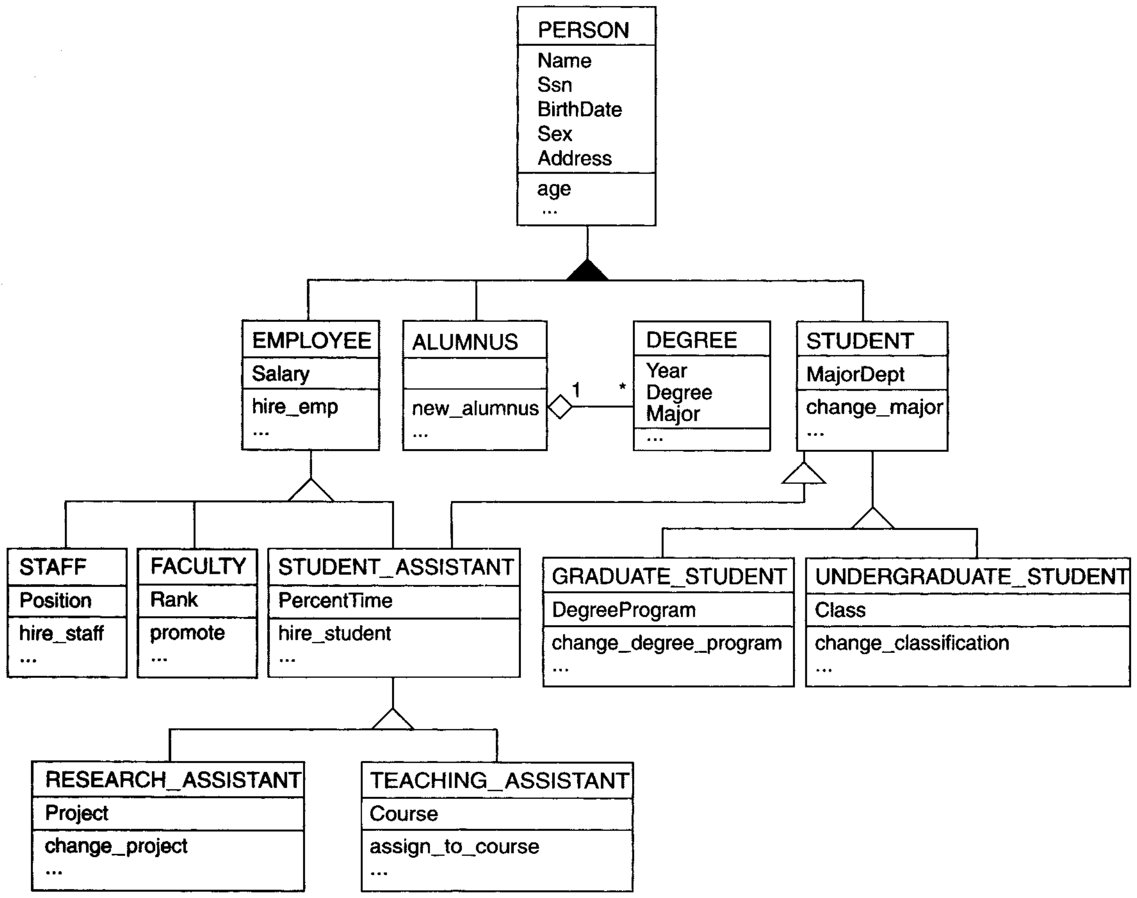

FIGURE 4.10 A UML class diagram corresponding to the EER diagram in Figure 4.7, illustrating UML notation for specialization/generalization

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,

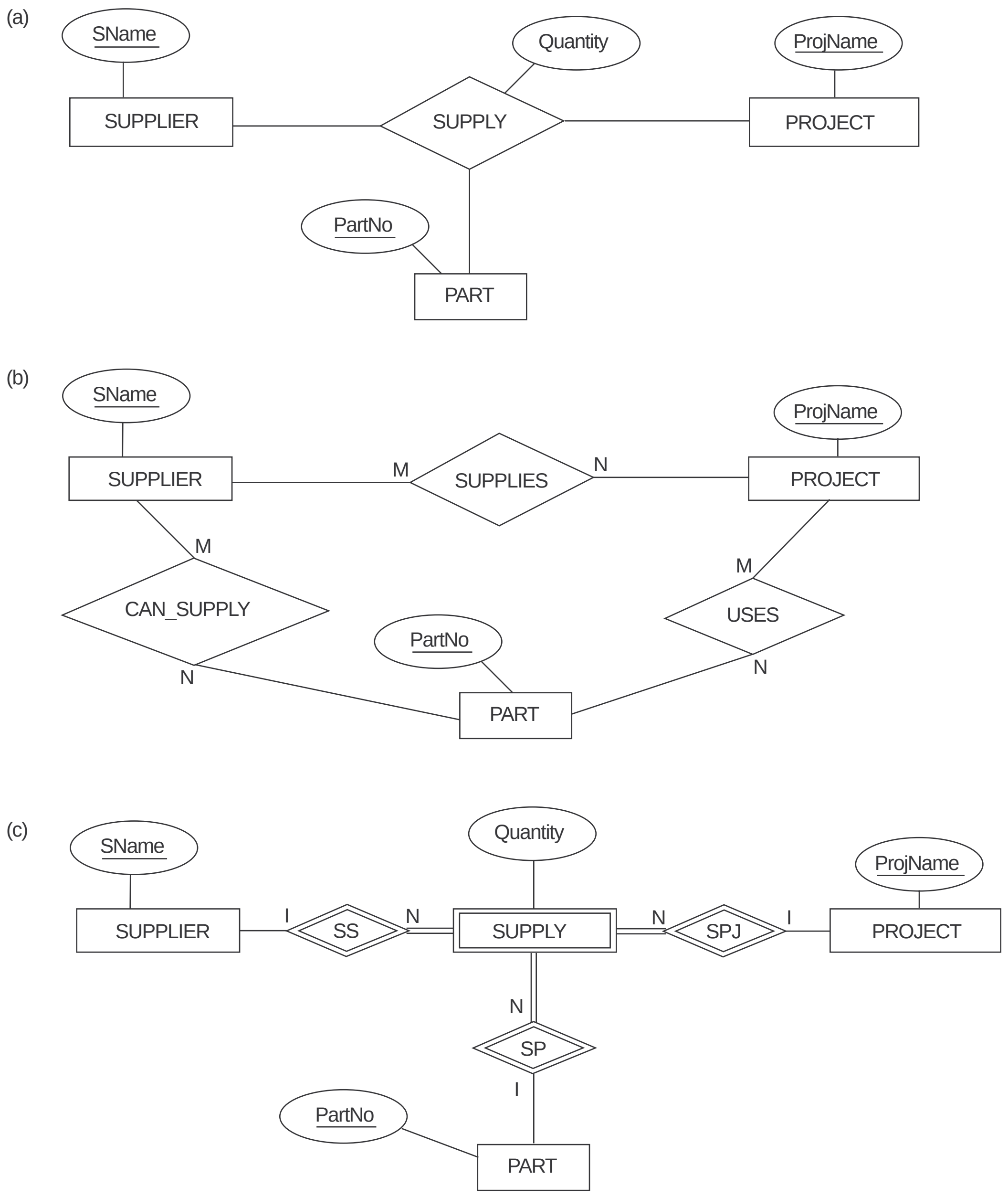

FIGURE 4.11 Ternary relationship types. (a) The SUPPLY relationship. (b) Three binary relationships not equivalent to SUPPLY. (c) SUPPLY represented as a weak entity type.

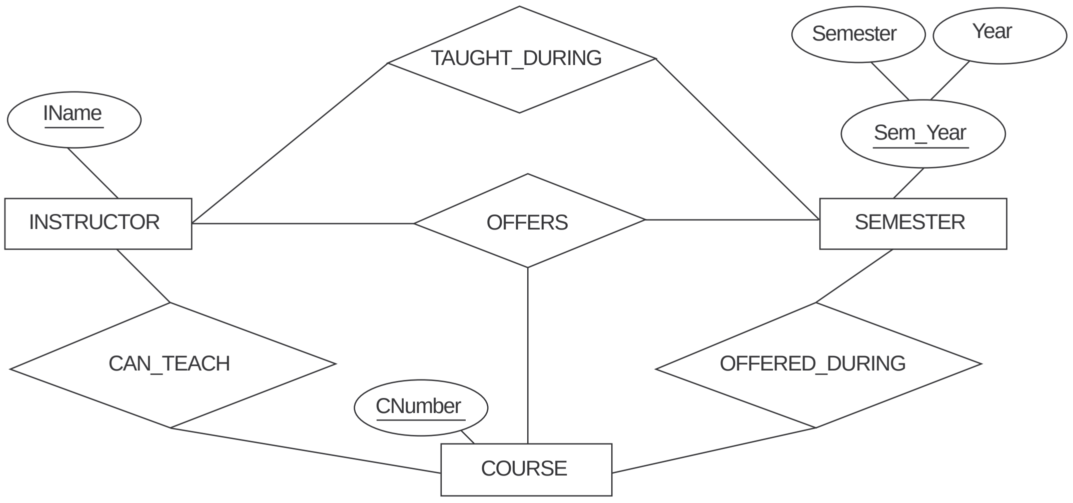

FIGURE 4.12 Another example of ternary versus binary relationship types

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

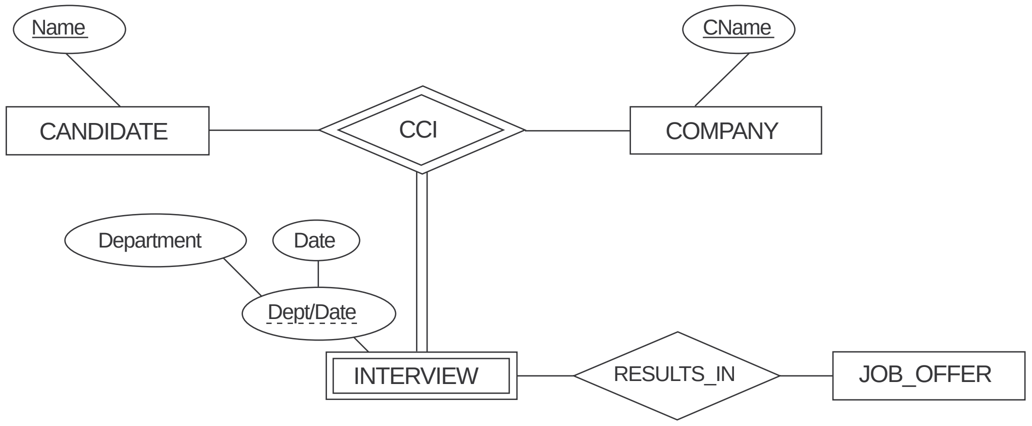

FIGURE 4.13 A weak entity type INTERVIEW with a ternary identifying relationship type

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 Chapter 24).
- 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-AN-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 relationship 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

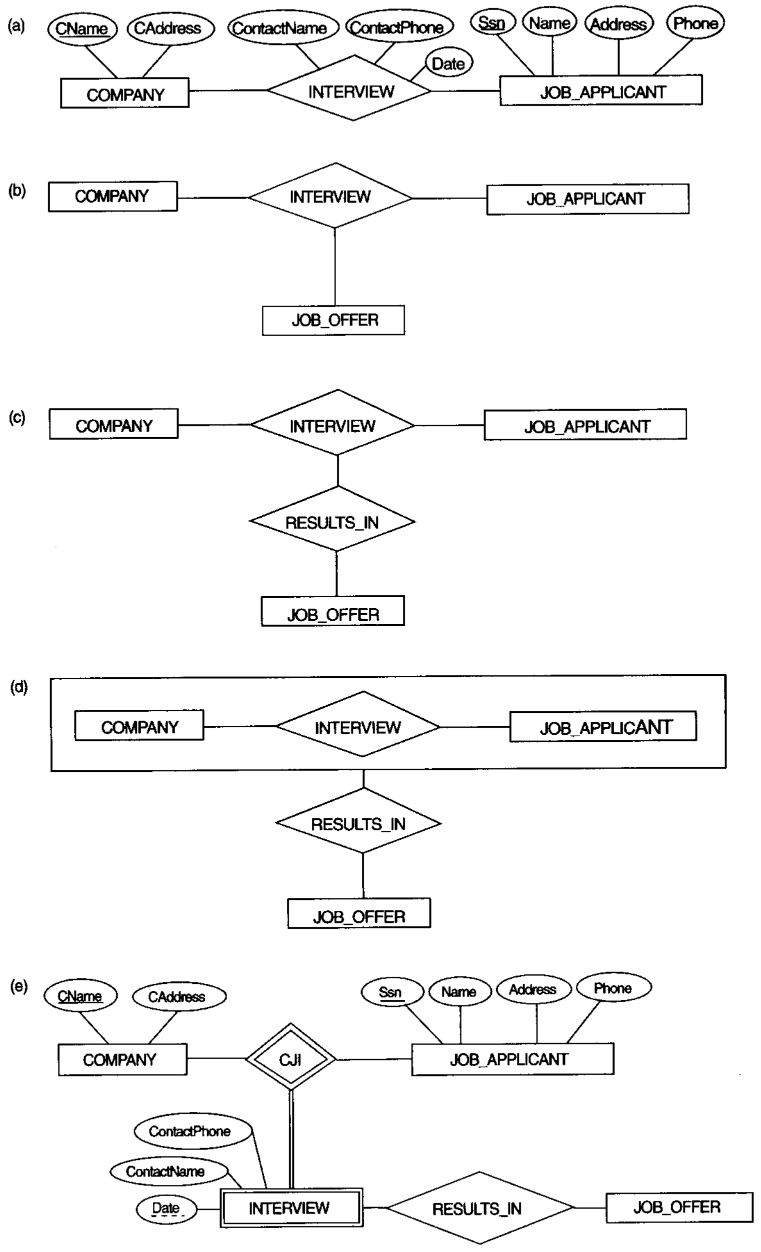

FIGURE 4.14 Aggregation. (a) The relationship type INTERVIEW. (b) Including JOB_OFFER in a ternary relationship type (incorrect). (c) Having the RESULTS_IN relationship participate in other relationships (generally not allowed in ER). (d) Using aggregation and a composite (molecular) object (generally 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. dentify 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 international 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 have maintained the plane [MAINTAIN]. Each relationship instance in OWNS

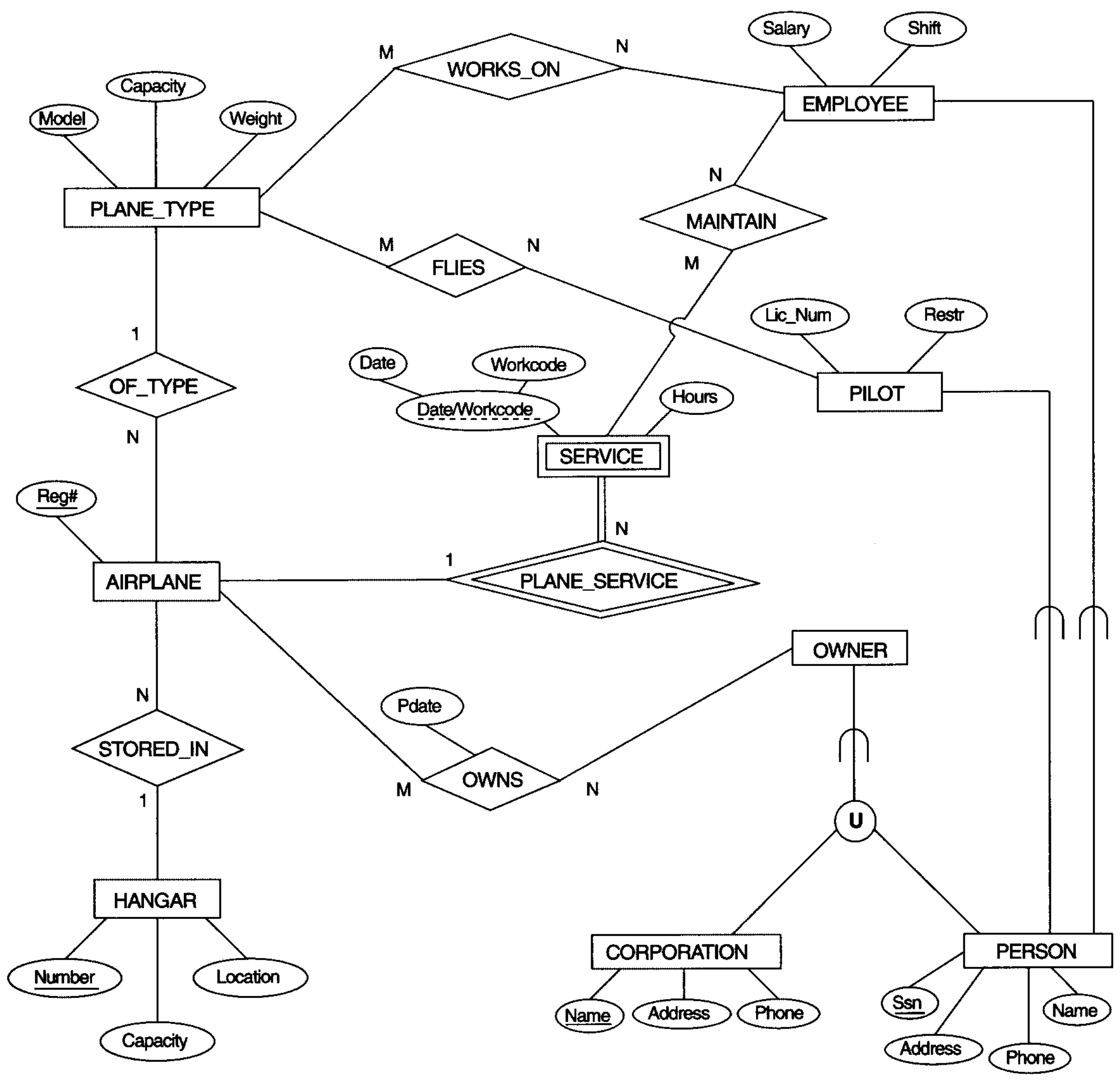

FIGURE 4.15 EER schema for a SMALL AIRPORT database

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). Another survey of conceptual modeling is Pillalamarri et al. (1988). 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).