2

INTRODUCTION TO DATABASE DESIGN

- What are the steps in designing a database?
- Why is the ER model used to create an initial design?
- What are the main concepts in the ER model?
- What are guidelines for using the ER model effectively?
- How does database design fit within the overall design framework for complex software within large enterprises?
- What is UML and how is it related to the ER model?
- **Key concepts:** database design, conceptual, logical, and physical design; entity-relationship (ER) model, entity set, relationship set, attribute, instance, key; integrity constraints, one-to-many and many-to-many relationships, participation constraints; weak entities, class hierarchies, aggregation; UML, class diagrams, database diagrams, component diagrams.

The great successful men of the world have used their imaginations. They think ahead and create their mental picture, and then go to work materializing that picture in all its details, filling in here, adding a little there, altering this bit and that bit, but steadily building, steadily building.

—Robert Collier

The *entity-relationship (ER) data model* allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design. It provides useful concepts that allow us to move from an informal description of what users want from

their database to a more detailed, precise description that can be implemented in a DBMS. In this chapter, we introduce the ER model and discuss how its features allow us to model a wide range of data faithfully.

We begin with an overview of database design in Section 2.1 in order to motivate our discussion of the ER model. Within the larger context of the overall design process, the ER model is used in a phase called *conceptual database design*. We then introduce the ER model in Sections 2.2, 2.3, and 2.4. In Section 2.5, we discuss database design issues involving the ER model. We briefly discuss conceptual database design for large enterprises in Section 2.6. In Section 2.7, we present an overview of UML, a design and modeling approach that is more general in its scope than the ER model.

In Section 2.8, we introduce a case study that is used as a running example throughout the book. The case study is an end-to-end database design for an Internet shop. We illustrate the first two steps in database design (requirements analysis and conceptual design) in Section 2.8. In later chapters, we extend this case study to cover the remaining steps in the design process.

We note that many variations of ER diagrams are in use and no widely accepted standards prevail. The presentation in this chapter is representative of the family of ER models and includes a selection of the most popular features.

2.1 DATABASE DESIGN AND ER DIAGRAMS

We begin our discussion of database design by observing that this is typically just one part, although a central part in data-intensive applications, of a larger software system design. Our primary focus is the design of the database, however, and we will not discuss other aspects of software design in any detail. We revisit this point in Section 2.7.

The database design process can be divided into six steps. The ER model is most relevant to the first three steps.

1. **Requirements Analysis:** The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. In other words, we must find out what the users want from the database. This is usually an informal process that involves discussions with user groups, a study of the current operating environment and how it is expected to change, analysis of any available documentation on existing applications that are expected to be replaced or complemented by the database, and so on.

> **Database Design Tools:** Design tools are available from RDBMS vendors as well as third-party vendors. For example, see the following link for details on design and analysis tools from Sybase:
> `http://www.sybase.com/products/application_tools`
> The following provides details on Oracle's tools:
> `http://www.oracle.com/tools`

Several methodologies have been proposed for organizing and presenting the information gathered in this step, and some automated tools have been developed to support this process.

2. **Conceptual Database Design:** The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints known to hold over this data. This step is often carried out using the ER model and is discussed in the rest of this chapter. The ER model is one of several high-level, or **semantic**, data models used in database design. The goal is to create a simple description of the data that closely matches how users and developers think of the data (and the people and processes to be represented in the data). This facilitates discussion among all the people involved in the design process, even those who have no technical background. At the same time, the initial design must be sufficiently precise to enable a straightforward translation into a data model supported by a commercial database system (which, in practice, means the relational model).

3. **Logical Database Design:** We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model of the chosen DBMS. We will consider only relational DBMSs, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema. We discuss this step in detail in Chapter 3; the result is a conceptual schema, sometimes called the **logical schema**, in the relational data model.

2.1.1 Beyond ER Design

The ER diagram is just an approximate description of the data, constructed through a subjective evaluation of the information collected during requirements analysis. A more careful analysis can often refine the logical schema obtained at the end of Step 3. Once we have a good logical schema, we must consider performance criteria and design the physical schema. Finally, we must address security issues and ensure that users are able to access the data they need, but not data that we wish to hide from them. The remaining three steps of database design are briefly described next:

4. **Schema Refinement:** The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems, and to refine it. In contrast to the requirements analysis and conceptual design steps, which are essentially subjective, schema refinement can be guided by some elegant and powerful theory. We discuss the theory of *normalizing* relations—restructuring them to ensure some desirable properties—in Chapter 19.

5. **Physical Database Design:** In this step, we consider typical expected workloads that our database must support and further refine the database design to ensure that it meets desired performance criteria. This step may simply involve building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema obtained from the earlier design steps. We discuss physical design and database tuning in Chapter 20.

6. **Application and Security Design:** Any software project that involves a DBMS must consider aspects of the application that go beyond the database itself. Design methodologies like UML (Section 2.7) try to address the complete software design and development cycle. Briefly, we must identify the entities (e.g., users, user groups, departments) and processes involved in the application. We must describe the role of each entity in every process that is reflected in some application task, as part of a complete workflow for that task. For each role, we must identify the parts of the database that must be accessible and the parts of the database that must *not* be accessible, and we must take steps to ensure that these access rules are enforced. A DBMS provides several mechanisms to assist in this step, and we discuss this in Chapter 21.

In the implementation phase, we must code each task in an application language (e.g., Java), using the DBMS to access data. We discuss application development in Chapters 6 and 7.

In general, our division of the design process into steps should be seen as a classification of the *kinds* of steps involved in design. Realistically, although we might begin with the six step process outlined here, a complete database design will probably require a subsequent **tuning** phase in which all six kinds of design steps are interleaved and repeated until the design is satisfactory.

2.2 ENTITIES, ATTRIBUTES, AND ENTITY SETS

An **entity** is an object in the real world that is distinguishable from other objects. Examples include the following: the Green Dragonzord toy, the toy department, the manager of the toy department, the home address of the manager of the toy department. It is often useful to identify a collection of similar entities. Such a collection is called an **entity set**. Note that entity sets need not be disjoint; the collection of toy department employees and the collection of appliance department employees may both contain employee John Doe (who happens to work in both departments). We could also define an entity set called Employees that contains both the toy and appliance department employee sets.

An entity is described using a set of **attributes**. All entities in a given entity set have the same attributes; this is what we mean by *similar*. (This statement is an oversimplification, as we will see when we discuss inheritance hierarchies in Section 2.4.4, but it suffices for now and highlights the main idea.) Our choice of attributes reflects the level of detail at which we wish to represent information about entities. For example, the Employees entity set could use name, social security number (ssn), and parking lot (lot) as attributes. In this case we will store the name, social security number, and lot number for each employee. However, we will not store, say, an employee's address (or gender or age).

For each attribute associated with an entity set, we must identify a **domain** of possible values. For example, the domain associated with the attribute *name* of Employees might be the set of 20-character strings.[1] As another example, if the company rates employees on a scale of 1 to 10 and stores ratings in a field called *rating*, the associated domain consists of integers 1 through 10. Further, for each entity set, we choose a *key*. A **key** is a minimal set of attributes whose values uniquely identify an entity in the set. There could be more than one **candidate** key; if so, we designate one of them as the **primary** key. For now we assume that each entity set contains at least one set of attributes that uniquely identifies an entity in the entity set; that is, the set of attributes contains a key. We revisit this point in Section 2.4.3.

The Employees entity set with attributes *ssn*, *name*, and *lot* is shown in Figure 2.1. An entity set is represented by a rectangle, and an attribute is represented by an oval. Each attribute in the primary key is underlined. The domain information could be listed along with the attribute name, but we omit this to keep the figures compact. The key is *ssn*.

2.3 RELATIONSHIPS AND RELATIONSHIP SETS

A **relationship** is an association among two or more entities. For example, we may have the relationship that Attishoo works in the pharmacy department.

[1] To avoid confusion, we assume that attribute names do not repeat across entity sets. This is not a real limitation because we can always use the entity set name to resolve ambiguities if the same attribute name is used in more than one entity set.

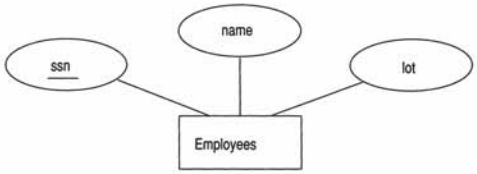

Figure 2.1 The Employees Entity Set

As with entities, we may wish to collect a set of similar relationships into a **relationship set**. A relationship set can be thought of as a set of n-tuples:

$$\{(e_1, \ldots, e_n) \mid e_1 \in E_1, \ldots, e_n \in E_n\}$$

Each n-tuple denotes a relationship involving n entities e_1 through e_n, where entity e_i is in entity set E_i. In Figure 2.2 we show the relationship set Works_In, in which each relationship indicates a department in which an employee works. Note that several relationship sets might involve the same entity sets. For example, we could also have a Manages relationship set involving Employees and Departments.

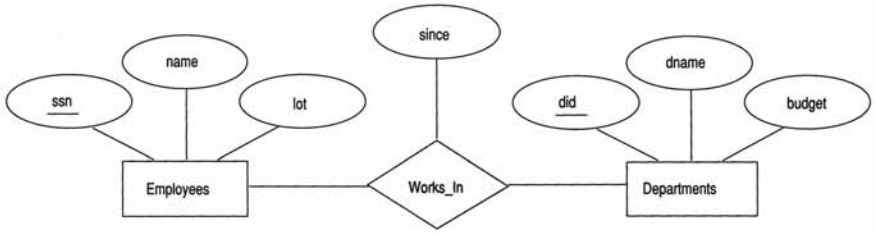

Figure 2.2 The Works_In Relationship Set

A relationship can also have **descriptive attributes**. Descriptive attributes are used to record information about the relationship, rather than about any one of the participating entities; for example, we may wish to record that Attishoo works in the pharmacy department as of January 1991. This information is captured in Figure 2.2 by adding an attribute, *since*, to Works_In. A relationship must be uniquely identified by the participating entities, without reference to the descriptive attributes. In the Works_In relationship set, for example, each Works_In relationship must be uniquely identified by the combination of employee *ssn* and department *did*. Thus, for a given employee-department pair, we cannot have more than one associated *since* value.

An **instance** of a relationship set is a set of relationships. Intuitively, an instance can be thought of as a 'snapshot' of the relationship set at some instant

in time. An instance of the Works_In relationship set is shown in Figure 2.3. Each Employees entity is denoted by its *ssn*, and each Departments entity is denoted by its *did*, for simplicity. The *since* value is shown beside each relationship. (The 'many-to-many' and 'total participation' comments in the figure are discussed later, when we discuss integrity constraints.)

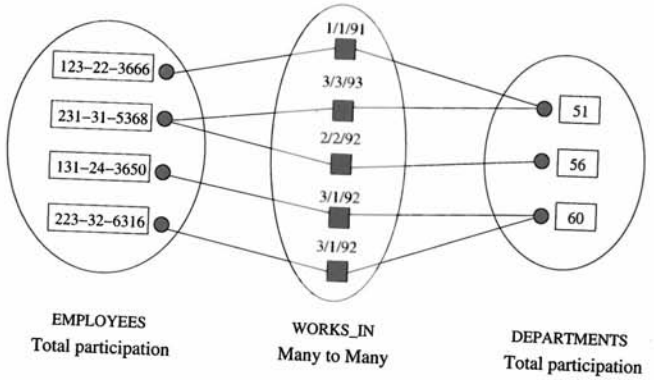

Figure 2.3 An Instance of the Works_In Relationship Set

As another example of an ER diagram, suppose that each department has offices in several locations and we want to record the locations at which each employee works. This relationship is **ternary** because we must record an association between an employee, a department, and a location. The ER diagram for this variant of Works_In, which we call Works_In2, is shown in Figure 2.4.

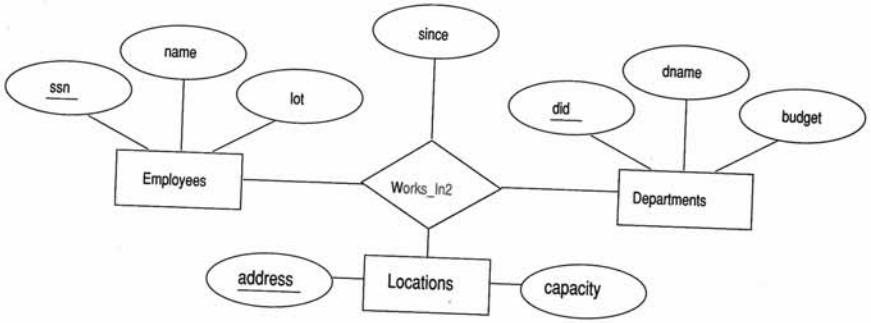

Figure 2.4 A Ternary Relationship Set

The entity sets that participate in a relationship set need not be distinct; sometimes a relationship might involve two entities in the same entity set. For example, consider the Reports_To relationship set shown in Figure 2.5. Since

employees report to other employees, every relationship in Reports_To is of the form (emp_1, emp_2), where both emp_1 and emp_2 are entities in Employees. However, they play different **roles**: emp_1 reports to the managing employee emp_2, which is reflected in the **role indicators** *supervisor* and *subordinate* in Figure 2.5. If an entity set plays more than one role, the role indicator concatenated with an attribute name from the entity set gives us a unique name for each attribute in the relationship set. For example, the Reports_To relationship set has attributes corresponding to the *ssn* of the supervisor and the *ssn* of the subordinate, and the names of these attributes are *supervisor_ssn* and *subordinate_ssn*.

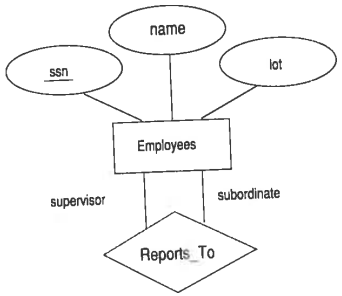

Figure 2.5 The Reports_To Relationship Set

2.4 ADDITIONAL FEATURES OF THE ER MODEL

We now look at some of the constructs in the ER model that allow us to describe some subtle properties of the data. The expressiveness of the ER model is a big reason for its widespread use.

2.4.1 Key Constraints

Consider the Works_In relationship shown in Figure 2.2. An employee can work in several departments, and a department can have several employees, as illustrated in the Works_In instance shown in Figure 2.3. Employee 231-31-5368 has worked in Department 51 since 3/3/93 and in Department 56 since 2/2/92. Department 51 has two employees.

Now consider another relationship set called Manages between the Employees and Departments entity sets such that each department has at most one manager, although a single employee is allowed to manage more than one department. The restriction that each department has at most one manager is

an example of a **key constraint**, and it implies that each Departments entity appears in at most one Manages relationship in any allowable instance of Manages. This restriction is indicated in the ER diagram of Figure 2.6 by using an arrow from Departments to Manages. Intuitively, the arrow states that given a Departments entity, we can uniquely determine the Manages relationship in which it appears.

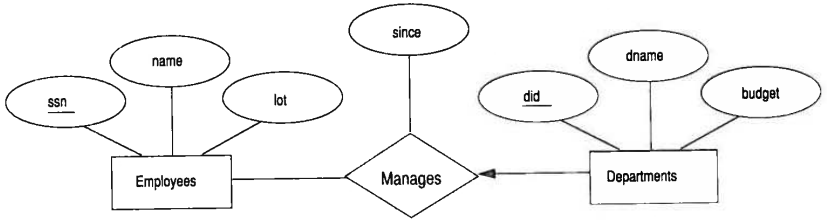

Figure 2.6 Key Constraint on Manages

An instance of the Manages relationship set is shown in Figure 2.7. While this is also a potential instance for the Works_In relationship set, the instance of Works_In shown in Figure 2.3 violates the key constraint on Manages.

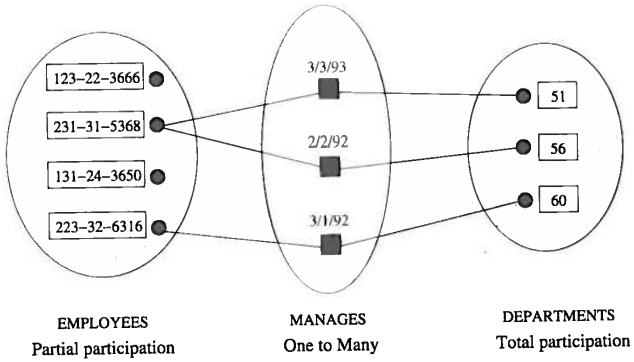

Figure 2.7 An Instance of the Manages Relationship Set

A relationship set like Manages is sometimes said to be **one-to-many**, to indicate that *one* employee can be associated with *many* departments (in the capacity of a manager), whereas each department can be associated with at most one employee as its manager. In contrast, the Works_In relationship set, in which an employee is allowed to work in several departments and a department is allowed to have several employees, is said to be **many-to-many**.

If we add the restriction that each employee can manage at most one department to the Manages relationship set, which would be indicated by adding an arrow from Employees to Manages in Figure 2.6, we have a **one-to-one** relationship set.

Key Constraints for Ternary Relationships

We can extend this convention—and the underlying key constraint concept—to relationship sets involving three or more entity sets: If an entity set E has a key constraint in a relationship set R, each entity in an instance of E appears in at most one relationship in (a corresponding instance of) R. To indicate a key constraint on entity set E in relationship set R, we draw an arrow from E to R.

In Figure 2.8, we show a ternary relationship with key constraints. Each employee works in at most one department and at a single location. An instance of the Works_In3 relationship set is shown in Figure 2.9. Note that each department can be associated with several employees and locations, and each location can be associated with several departments and employees; however, each employee is associated with a single department and location.

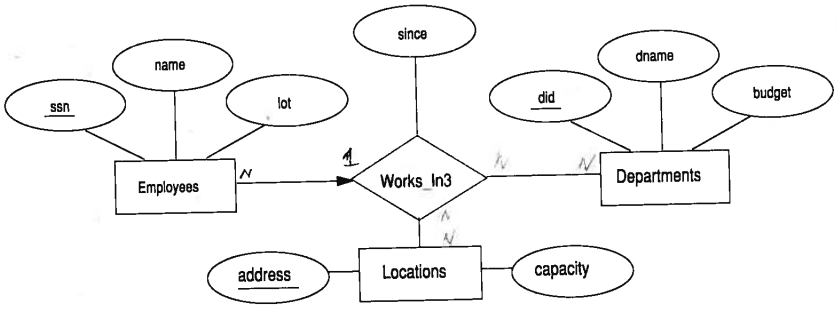

Figure 2.8 A Ternary Relationship Set with Key Constraints

2.4.2 Participation Constraints

The key constraint on Manages tells us that a department has at most one manager. A natural question to ask is whether every department has a manager. Let us say that every department is required to have a manager. This requirement is an example of a **participation constraint**; the participation of the entity set Departments in the relationship set Manages is said to be **total**. A participation that is not total is said to be **partial**. As an example, the

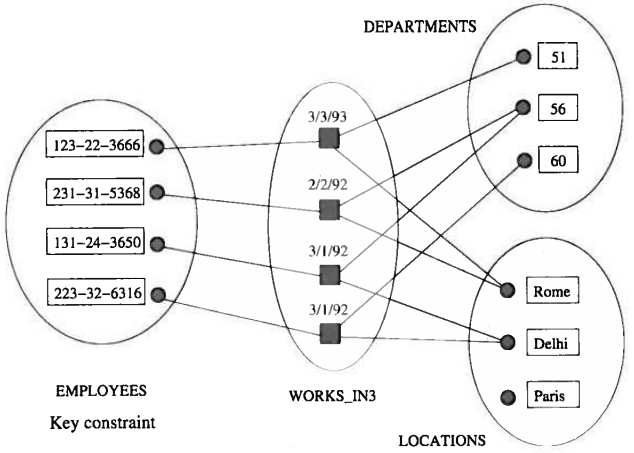

Figure 2.9 An Instance of Works_In3

participation of the entity set Employees in Manages is partial, since not every employee gets to manage a department.

Revisiting the Works_In relationship set, it is natural to expect that each employee works in at least one department and that each department has at least one employee. This means that the participation of both Employees and Departments in Works_In is total. The ER diagram in Figure 2.10 shows both the Manages and Works_In relationship sets and all the given constraints. If the participation of an entity set in a relationship set is total, the two are connected by a thick line; independently, the presence of an arrow indicates a key constraint. The instances of Works_In and Manages shown in Figures 2.3 and 2.7 satisfy all the constraints in Figure 2.10.

2.4.3 Weak Entities

Thus far, we have assumed that the attributes associated with an entity set include a key. This assumption does not always hold. For example, suppose that employees can purchase insurance policies to cover their dependents. We wish to record information about policies, including who is covered by each policy, but this information is really our only interest in the dependents of an employee. If an employee quits, any policy owned by the employee is terminated and we want to delete all the relevant policy and dependent information from the database.

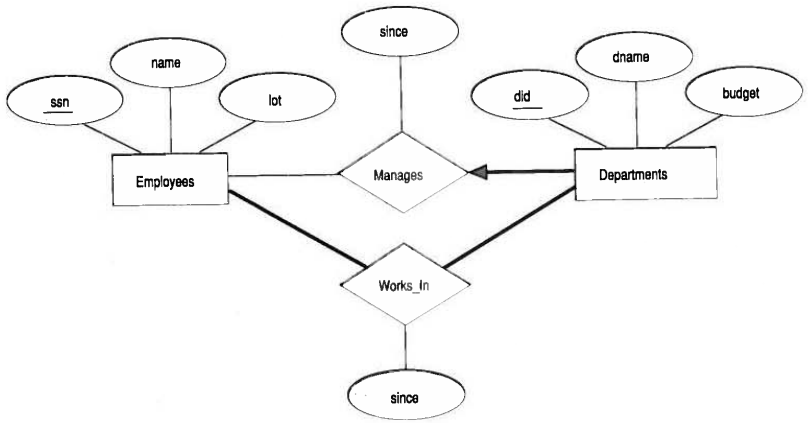

Figure 2.10 Manages and Works_In

We might choose to identify a dependent by name alone in this situation, since it is reasonable to expect that the dependents of a given employee have different names. Thus the attributes of the Dependents entity set might be *pname* and *age*. The attribute *pname* does *not* identify a dependent uniquely. Recall that the key for Employees is *ssn*; thus we might have two employees called Smethurst and each might have a son called Joe.

Dependents is an example of a **weak entity set**. A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity, which is called the **identifying owner**.

The following restrictions must hold:

- The owner entity set and the weak entity set must participate in a one-to-many relationship set (one owner entity is associated with one or more weak entities, but each weak entity has a single owner). This relationship set is called the **identifying relationship set** of the weak entity set.
- The weak entity set must have total participation in the identifying relationship set.

For example, a Dependents entity can be identified uniquely only if we take the key of the *owning* Employees entity and the *pname* of the Dependents entity. The set of attributes of a weak entity set that uniquely identify a weak entity for a given owner entity is called a *partial key* of the weak entity set. In our example, *pname* is a partial key for Dependents.

The Dependents weak entity set and its relationship to Employees is shown in Figure 2.11. The total participation of Dependents in Policy is indicated by linking them with a dark line. The arrow from Dependents to Policy indicates that each Dependents entity appears in at most one (indeed, exactly one, because of the participation constraint) Policy relationship. To underscore the fact that Dependents is a weak entity and Policy is its identifying relationship, we draw both with dark lines. To indicate that *pname* is a partial key for Dependents, we underline it using a broken line. This means that there may well be two dependents with the same *pname* value.

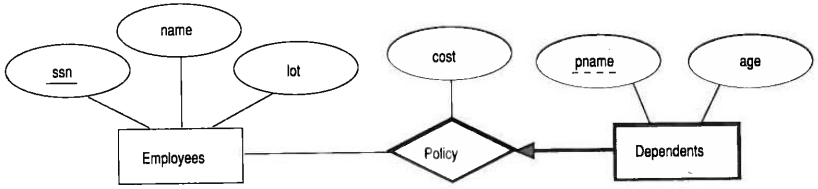

Figure 2.11 A Weak Entity Set

2.4.4 Class Hierarchies

Sometimes it is natural to classify the entities in an entity set into subclasses. For example, we might want to talk about an Hourly_Emps entity set and a Contract_Emps entity set to distinguish the basis on which they are paid. We might have attributes *hours_worked* and *hourly_wage* defined for Hourly_Emps and an attribute *contractid* defined for Contract_Emps.

We want the semantics that every entity in one of these sets is also an Employees entity and, as such, must have all the attributes of Employees defined. Therefore, the attributes defined for an Hourly_Emps entity are the attributes for Employees plus Hourly_Emps. We say that the attributes for the entity set Employees are **inherited** by the entity set Hourly_Emps and that Hourly_Emps **ISA** (read *is a*) Employees. In addition—and in contrast to class hierarchies in programming languages such as C++—there is a constraint on queries over instances of these entity sets: A query that asks for all Employees entities must consider all Hourly_Emps and Contract_Emps entities as well. Figure 2.12 illustrates the class hierarchy.

The entity set Employees may also be classified using a different criterion. For example, we might identify a subset of employees as Senior_Emps. We can modify Figure 2.12 to reflect this change by adding a second ISA node as a child of Employees and making Senior_Emps a child of this node. Each of these entity sets might be classified further, creating a multilevel ISA hierarchy.

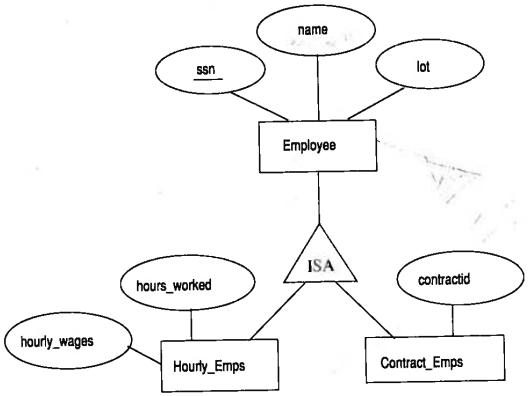

Figure 2.12 Class Hierarchy

A class hierarchy can be viewed in one of two ways:

- Employees is **specialized** into subclasses. Specialization is the process of identifying subsets of an entity set (the **superclass**) that share some distinguishing characteristic. Typically, the superclass is defined first, the subclasses are defined next, and subclass-specific attributes and relationship sets are then added.
- Hourly_Emps and Contract_Emps are **generalized** by Employees. As another example, two entity sets Motorboats and Cars may be generalized into an entity set Motor_Vehicles. Generalization consists of identifying some common characteristics of a collection of entity sets and creating a new entity set that contains entities possessing these common characteristics. Typically, the subclasses are defined first, the superclass is defined next, and any relationship sets that involve the superclass are then defined.

We can specify two kinds of constraints with respect to ISA hierarchies, namely, *overlap* and *covering* constraints. **Overlap constraints** determine whether two subclasses are allowed to contain the same entity. For example, can Attishoo be both an Hourly_Emps entity and a Contract_Emps entity? Intuitively, no. Can he be both a Contract_Emps entity and a Senior_Emps entity? Intuitively, yes. We denote this by writing 'Contract_Emps `OVERLAPS` Senior_Emps.' In the absence of such a statement, we assume by default that entity sets are constrained to have no overlap.

Covering constraints determine whether the entities in the subclasses collectively include all entities in the superclass. For example, does every Employees

entity have to belong to one of its subclasses? Intuitively, no. Does every Motor_Vehicles entity have to be either a Motorboats entity or a Cars entity? Intuitively, yes; a characteristic property of generalization hierarchies is that every instance of a superclass is an instance of a subclass. We denote this by writing 'Motorboats `AND` Cars `COVER` Motor_Vehicles.' In the absence of such a statement, we assume by default that there is no covering constraint; we can have motor vehicles that are not motorboats or cars.

There are two basic reasons for identifying subclasses (by specialization or generalization):

1. We might want to add descriptive attributes that make sense only for the entities in a subclass. For example, *hourly_wages* does not make sense for a Contract_Emps entity, whose pay is determined by an individual contract.
2. We might want to identify the set of entities that participate in some relationship. For example, we might wish to define the Manages relationship so that the participating entity sets are Senior_Emps and Departments, to ensure that only senior employees can be managers. As another example, Motorboats and Cars may have different descriptive attributes (say, tonnage and number of doors), but as Motor_Vehicles entities, they must be licensed. The licensing information can be captured by a Licensed_To relationship between Motor_Vehicles and an entity set called Owners.

2.4.5 Aggregation

As defined thus far, a relationship set is an association between entity sets. Sometimes, we have to model a relationship between a collection of entities and *relationships*. Suppose that we have an entity set called Projects and that each Projects entity is sponsored by one or more departments. The Sponsors relationship set captures this information. A department that sponsors a project might assign employees to monitor the sponsorship. Intuitively, Monitors should be a relationship set that associates a Sponsors relationship (rather than a Projects or Departments entity) with an Employees entity. However, we have defined relationships to associate two or more *entities*.

To define a relationship set such as Monitors, we introduce a new feature of the ER model, called *aggregation*. **Aggregation** allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship set. This is illustrated in Figure 2.13, with a dashed box around Sponsors (and its participating entity sets) used to denote aggregation. This effectively allows us to treat Sponsors as an entity set for purposes of defining the Monitors relationship set.

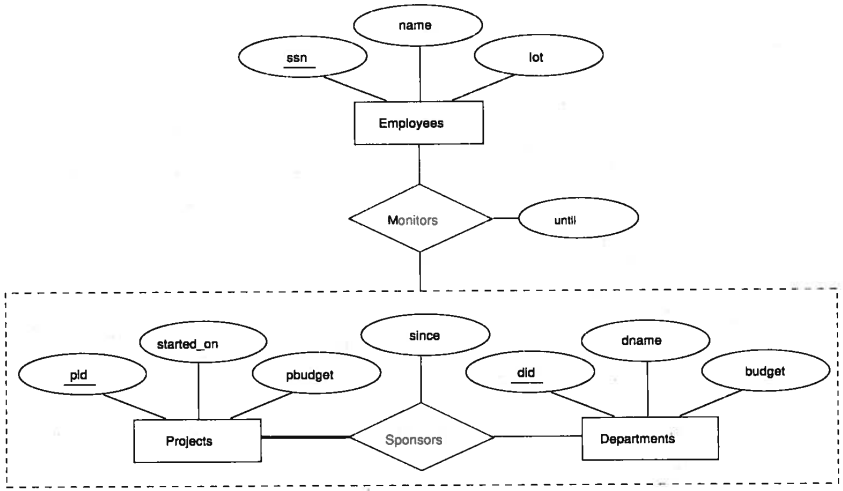

Figure 2.13 Aggregation

When should we use aggregation? Intuitively, we use it when we need to express a relationship among relationships. But can we not express relationships involving other relationships without using aggregation? In our example, why not make Sponsors a ternary relationship? The answer is that there are really two distinct relationships, Sponsors and Monitors, each possibly with attributes of its own. For instance, the Monitors relationship has an attribute *until* that records the date until when the employee is appointed as the sponsorship monitor. Compare this attribute with the attribute *since* of Sponsors, which is the date when the sponsorship took effect. The use of aggregation versus a ternary relationship may also be guided by certain integrity constraints, as explained in Section 2.5.4.

2.5 CONCEPTUAL DESIGN WITH THE ER MODEL

Developing an ER diagram presents several choices, including the following:

- Should a concept be modeled as an entity or an attribute?
- Should a concept be modeled as an entity or a relationship?
- What are the relationship sets and their participating entity sets? Should we use binary or ternary relationships?
- Should we use aggregation?

We now discuss the issues involved in making these choices.

2.5.1 Entity versus Attribute

While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modeled as an attribute or as an entity set (and related to the first entity set using a relationship set). For example, consider adding address information to the Employees entity set. One option is to use an attribute *address*. This option is appropriate if we need to record only one address per employee, and it suffices to think of an address as a string. An alternative is to create an entity set called Addresses and to record associations between employees and addresses using a relationship (say, Has_Address). This more complex alternative is necessary in two situations:

- We have to record more than one address for an employee.
- We want to capture the structure of an address in our ER diagram. For example, we might break down an address into city, state, country, and Zip code, in addition to a string for street information. By representing an address as an entity with these attributes, we can support queries such as "Find all employees with an address in Madison, WI."

For another example of when to model a concept as an entity set rather than an attribute, consider the relationship set (called Works_In4) shown in Figure 2.14.

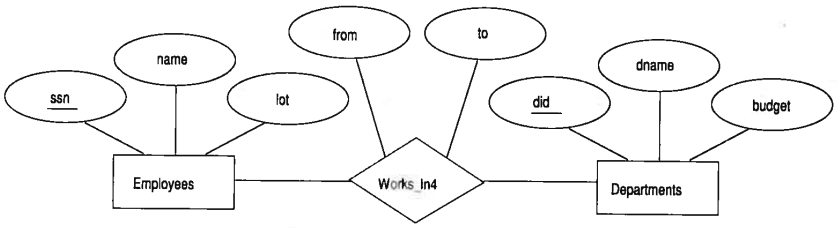

Figure 2.14 The Works_In4 Relationship Set

It differs from the Works_In relationship set of Figure 2.2 only in that it has attributes *from* and *to*, instead of *since*. Intuitively, it records the interval during which an employee works for a department. Now suppose that it is possible for an employee to work in a given department over more than one period.

This possibility is ruled out by the ER diagram's semantics, because a relationship is uniquely identified by the participating entities (recall from Section

2.3). The problem is that we want to record several values for the descriptive attributes for each instance of the Works_In2 relationship. (This situation is analogous to wanting to record several addresses for each employee.) We can address this problem by introducing an entity set called, say, Duration, with attributes *from* and *to*, as shown in Figure 2.15.

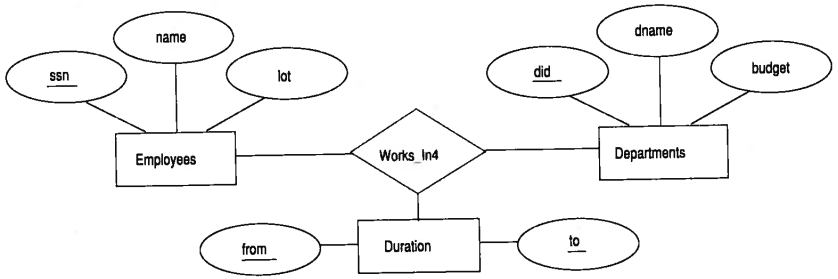

Figure 2.15 The Works_In4 Relationship Set

In some versions of the ER model, attributes are allowed to take on sets as values. Given this feature, we could make Duration an attribute of Works_In, rather than an entity set; associated with each Works_In relationship, we would have a set of intervals. This approach is perhaps more intuitive than modeling Duration as an entity set. Nonetheless, when such set-valued attributes are translated into the relational model, which does not support set-valued attributes, the resulting relational schema is very similar to what we get by regarding Duration as an entity set.

2.5.2 Entity versus Relationship

Consider the relationship set called Manages in Figure 2.6. Suppose that each department manager is given a discretionary budget (*dbudget*), as shown in Figure 2.16, in which we have also renamed the relationship set to Manages2.

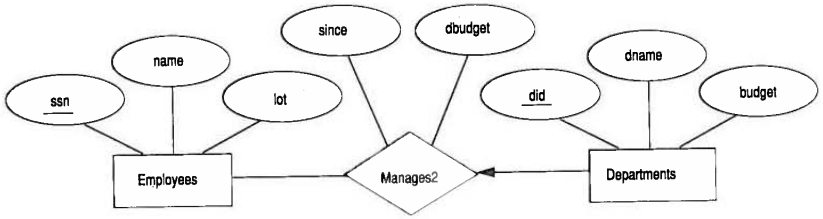

Figure 2.16 Entity versus Relationship

Given a department, we know the manager, as well as the manager's starting date and budget for that department. This approach is natural if we assume that a manager receives a separate discretionary budget for each department that he or she manages.

But what if the discretionary budget is a sum that covers *all* departments managed by that employee? In this case, each Manages2 relationship that involves a given employee will have the same value in the *dbudget* field, leading to redundant storage of the same information. Another problem with this design is that it is misleading; it suggests that the budget is associated with the relationship, when it is actually associated with the manager.

We can address these problems by introducing a new entity set called Managers (which can be placed below Employees in an ISA hierarchy, to show that every manager is also an employee). The attributes *since* and *dbudget* now describe a manager entity, as intended. As a variation, while every manager has a budget, each manager may have a different starting date (as manager) for each department. In this case *dbudget* is an attribute of Managers, but *since* is an attribute of the relationship set between managers and departments.

The imprecise nature of ER modeling can thus make it difficult to recognize underlying entities, and we might associate attributes with relationships rather than the appropriate entities. In general, such mistakes lead to redundant storage of the same information and can cause many problems. We discuss redundancy and its attendant problems in Chapter 19, and present a technique called *normalization* to eliminate redundancies from tables.

2.5.3 Binary versus Ternary Relationships

Consider the ER diagram shown in Figure 2.17. It models a situation in which an employee can own several policies, each policy can be owned by several employees, and each dependent can be covered by several policies.

Suppose that we have the following additional requirements:

- A policy cannot be owned jointly by two or more employees.
- Every policy must be owned by some employee.
- Dependents is a weak entity set, and each dependent entity is uniquely identified by taking *pname* in conjunction with the *policyid* of a policy entity (which, intuitively, covers the given dependent).

The first requirement suggests that we impose a key constraint on Policies with respect to Covers, but this constraint has the unintended side effect that a

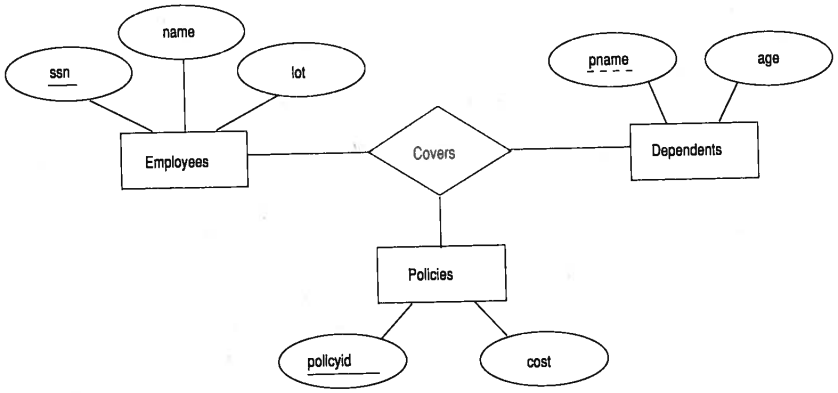

Figure 2.17 Policies as an Entity Set

policy can cover only one dependent. The second requirement suggests that we impose a total participation constraint on Policies. This solution is acceptable if each policy covers at least one dependent. The third requirement forces us to introduce an identifying relationship that is binary (in our version of ER diagrams, although there are versions in which this is not the case).

Even ignoring the third requirement, the best way to model this situation is to use two binary relationships, as shown in Figure 2.18.

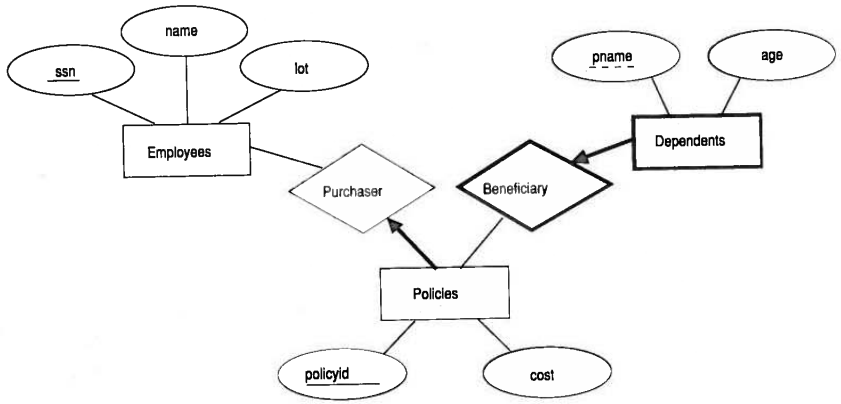

Figure 2.18 Policy Revisited

This example really has two relationships involving Policies, and our attempt to use a single ternary relationship (Figure 2.17) is inappropriate. There are situations, however, where a relationship inherently associates more than two entities. We have seen such an example in Figures 2.4 and 2.15.

As a typical example of a ternary relationship, consider entity sets Parts, Suppliers, and Departments, and a relationship set Contracts (with descriptive attribute *qty*) that involves all of them. A contract specifies that a supplier will supply (some quantity of) a part to a department. This relationship cannot be adequately captured by a collection of binary relationships (without the use of aggregation). With binary relationships, we can denote that a supplier 'can supply' certain parts, that a department 'needs' some parts, or that a department 'deals with' a certain supplier. No combination of these relationships expresses the meaning of a contract adequately, for at least two reasons:

- The facts that supplier S can supply part P, that department D needs part P, and that D will buy from S do not necessarily imply that department D indeed buys part P from supplier S!
- We cannot represent the *qty* attribute of a contract cleanly.

2.5.4 Aggregation versus Ternary Relationships

As we noted in Section 2.4.5, the choice between using aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a *relationship set* to an entity set (or second relationship set). The choice may also be guided by certain integrity constraints that we want to express. For example, consider the ER diagram shown in Figure 2.13. According to this diagram, a project can be sponsored by any number of departments, a department can sponsor one or more projects, and each sponsorship is monitored by one or more employees. If we don't need to record the *until* attribute of Monitors, then we might reasonably use a ternary relationship, say, Sponsors2, as shown in Figure 2.19.

Consider the constraint that each sponsorship (of a project by a department) be monitored by at most one employee. We cannot express this constraint in terms of the Sponsors2 relationship set. On the other hand, we can easily express the constraint by drawing an arrow from the aggregated relationship Sponsors to the relationship Monitors in Figure 2.13. Thus, the presence of such a constraint serves as another reason for using aggregation rather than a ternary relationship set.

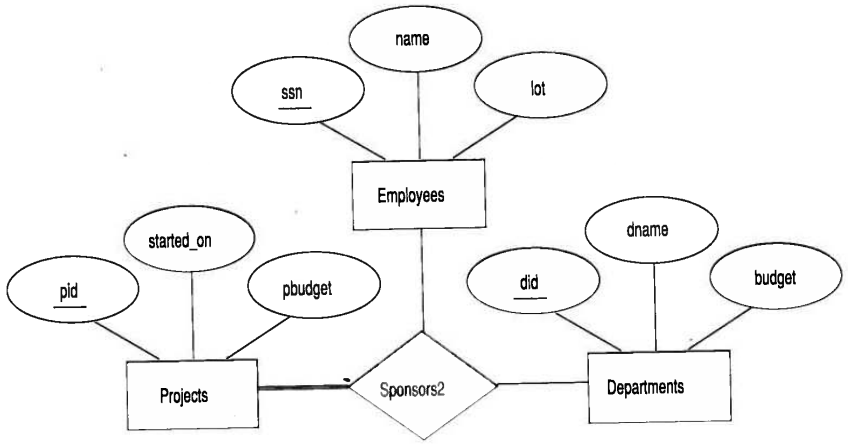

Figure 2.19 Using a Ternary Relationship instead of Aggregation

2.6 CONCEPTUAL DESIGN FOR LARGE ENTERPRISES

We have thus far concentrated on the constructs available in the ER model for describing various application concepts and relationships. The process of conceptual design consists of more than just describing small fragments of the application in terms of ER diagrams. For a large enterprise, the design may require the efforts of more than one designer and span data and application code used by a number of user groups. Using a high-level, semantic data model, such as ER diagrams, for conceptual design in such an environment offers the additional advantage that the high-level design can be diagrammatically represented and easily understood by the many people who must provide input to the design process.

An important aspect of the design process is the methodology used to structure the development of the overall design and ensure that the design takes into account all user requirements and is consistent. The usual approach is that the requirements of various user groups are considered, any conflicting requirements are somehow resolved, and a single set of global requirements is generated at the end of the requirements analysis phase. Generating a single set of global requirements is a difficult task, but it allows the conceptual design phase to proceed with the development of a logical schema that spans all the data and applications throughout the enterprise.

An alternative approach is to develop separate conceptual schemas for different user groups and then *integrate* these conceptual schemas. To integrate multiple conceptual schemas, we must establish correspondences between entities, relationships, and attributes, and we must resolve numerous kinds of conflicts (e.g., naming conflicts, domain mismatches, differences in measurement units). This task is difficult in its own right. In some situations, schema integration cannot be avoided; for example, when one organization merges with another, existing databases may have to be integrated. Schema integration is also increasing in importance as users demand access to *heterogeneous* data sources, often maintained by different organizations.

2.7 THE UNIFIED MODELING LANGUAGE

There are many approaches to end-to-end software system design, covering all the steps from identifying the business requirements to the final specifications for a complete application, including workflow, user interfaces, and many aspects of software systems that go well beyond databases and the data stored in them. In this section, we briefly discuss an approach that is becoming popular, called the **unified modeling language (UML) approach**.

UML, like the ER model, has the attractive feature that its constructs can be drawn as diagrams. It encompasses a broader spectrum of the software design process than the ER model:

- **Business Modeling:** In this phase, the goal is to describe the business processes involved in the software application being developed.
- **System Modeling:** The understanding of business processes is used to identify the requirements for the software application. One part of the requirements is the database requirements.
- **Conceptual Database Modeling:** This step corresponds to the creation of the ER design for the database. For this purpose, UML provides many constructs that parallel the ER constructs.
- **Physical Database Modeling:** UML also provides pictorial representations for physical database design choices, such as the creation of table spaces and indexes. (We discuss physical database design in later chapters, but not the corresponding UML constructs.)
- **Hardware System Modeling:** UML diagrams can be used to describe the hardware configuration used for the application.

There are many kinds of diagrams in UML. **Use case** diagrams describe the actions performed by the system in response to user requests, and the people involved in these actions. These diagrams specify the external functionality that the system is expected to support.

Activity diagrams show the flow of actions in a business process. **Statechart** diagrams describe dynamic interactions between system objects. These diagrams, used in business and system modeling, describe how the external functionality is to be implemented, consistent with the business rules and processes of the enterprise.

Class diagrams are similar to ER diagrams, although they are more general in that they are intended to model *application* entities (intuitively, important program components) and their logical relationships in addition to data entities and their relationships.

Both entity sets and relationship sets can be represented as classes in UML, together with key constraints, weak entities, and class hierarchies. The term *relationship* is used slightly differently in UML, and UML's relationships are binary. This sometimes leads to confusion over whether relationship sets in an ER diagram involving three or more entity sets can be directly represented in UML. The confusion disappears once we understand that all relationship sets (in the ER sense) are represented as classes in UML; the binary UML 'relationships' are essentially just the links shown in ER diagrams between entity sets and relationship sets.

Relationship sets with key constraints are usually omitted from UML diagrams, and the relationship is indicated by directly linking the entity sets involved. For example, consider Figure 2.6. A UML representation of this ER diagram would have a class for Employees, a class for Departments, and the relationship Manages is shown by linking these two classes. The link can be labeled with a name and cardinality information to show that a department can have only one manager.

As we will see in Chapter 3, ER diagrams are translated into the relational model by mapping each entity set into a table and each relationship set into a table. Further, as we will see in Section 3.5.3, the table corresponding to a one-to-many relationship set is typically omitted by including some additional information about the relationship in the table for one of the entity sets involved. Thus, UML class diagrams correspond closely to the tables created by mapping an ER diagram.

Indeed, every class in a UML class diagram is mapped into a table in the corresponding UML database diagram. UML's **database diagrams** show how classes are represented in the database and contain additional details about the structure of the database such as integrity constraints and indexes. Links (UML's 'relationships') between UML classes lead to various integrity constraints between the corresponding tables. Many details specific to the relational model (e.g., *views*, *foreign keys*, *null-allowed fields*) and that reflect

physical design choices (e.g., indexed fields) can be modeled in UML database diagrams.

UML's **component** diagrams describe storage aspects of the database, such as *tablespaces* and *database partitions*), as well as interfaces to applications that access the database. Finally, **deployment** diagrams show the hardware aspects of the system.

Our objective in this book is to concentrate on the data stored in a database and the related design issues. To this end, we deliberately take a simplified view of the other steps involved in software design and development. Beyond the specific discussion of UML, the material in this section is intended to place the design issues that we cover within the context of the larger software design process. We hope that this will assist readers interested in a more comprehensive discussion of software design to complement our discussion by referring to other material on their preferred approach to overall system design.

2.8 CASE STUDY: THE INTERNET SHOP

We now introduce an illustrative, 'cradle-to-grave' design case study that we use as a running example throughout this book. DBDudes Inc., a well-known database consulting firm, has been called in to help Barns and Nobble (B&N) with its database design and implementation. B&N is a large bookstore specializing in books on horse racing, and it has decided to go online. DBDudes first verifies that B&N is willing and able to pay its steep fees and then schedules a lunch meeting—billed to B&N, naturally—to do requirements analysis.

2.8.1 Requirements Analysis

The owner of B&N, unlike many people who need a database, has thought extensively about what he wants and offers a concise summary:

"I would like my customers to be able to browse my catalog of books and place orders over the Internet. Currently, I take orders over the phone. I have mostly corporate customers who call me and give me the ISBN number of a book and a quantity; they often pay by credit card. I then prepare a shipment that contains the books they ordered. If I don't have enough copies in stock, I order additional copies and delay the shipment until the new copies arrive; I want to ship a customer's entire order together. My catalog includes all the books I sell. For each book, the catalog contains its ISBN number, title, author, purchase price, sales price, and the year the book was published. Most of my customers are regulars, and I have records with their names and addresses.

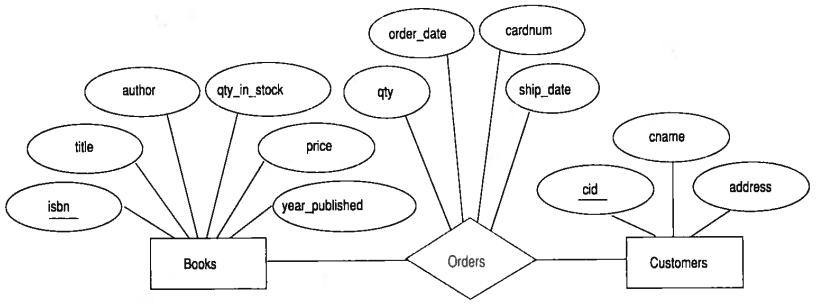

Figure 2.20 ER Diagram of the Initial Design

New customers have to call me first and establish an account before they can use my website.

On my new website, customers should first identify themselves by their unique customer identification number. Then they should be able to browse my catalog and to place orders online."

DBDudes's consultants are a little surprised by how quickly the requirements phase is completed—it usually takes weeks of discussions (and many lunches and dinners) to get this done—but return to their offices to analyze this information.

2.8.2 Conceptual Design

In the conceptual design step, DBDudes develops a high level description of the data in terms of the ER model. The initial design is shown in Figure 2.20. Books and customers are modeled as entities and related through orders that customers place. Orders is a relationship set connecting the Books and Customers entity sets. For each order, the following attributes are stored: quantity, order date, and ship date. As soon as an order is shipped, the ship date is set; until then the ship date is set to *null*, indicating that this order has not been shipped yet.

DBDudes has an internal design review at this point, and several questions are raised. To protect their identities, we will refer to the design team leader as Dude 1 and the design reviewer as Dude 2.

Dude 2: What if a customer places two orders for the same book in one day?
Dude 1: The first order is handled by creating a new Orders relationship and

the second order is handled by updating the value of the quantity attribute in this relationship.
Dude 2: What if a customer places two orders for different books in one day?
Dude 1: No problem. Each instance of the Orders relationship set relates the customer to a different book.
Dude 2: Ah, but what if a customer places two orders for the same book on different days?
Dude 1: We can use the attribute order date of the orders relationship to distinguish the two orders.
Dude 2: Oh no you can't. The attributes of Customers and Books must jointly contain a key for Orders. So this design does not allow a customer to place orders for the same book on different days.
Dude 1: Yikes, you're right. Oh well, B&N probably won't care; we'll see.

DBDudes decides to proceed with the next phase, logical database design; we rejoin them in Section 3.8.

2.9 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- Name the main steps in database design. What is the goal of each step? In which step is the ER model mainly used? **(Section 2.1)**
- Define these terms: *entity*, *entity set*, *attribute*, *key*. **(Section 2.2)**
- Define these terms: *relationship*, *relationship set*, *descriptive attributes*. **(Section 2.3)**
- Define the following kinds of constraints, and give an example of each: *key constraint*, *participation constraint*. What is a *weak entity*? What are *class hierarchies*? What is *aggregation*? Give an example scenario motivating the use of each of these ER model design constructs. **(Section 2.4)**
- What guidelines would you use for each of these choices when doing ER design: Whether to use an attribute or an entity set, an entity or a relationship set, a binary or ternary relationship, or aggregation. **(Section 2.5)**
- Why is designing a database for a large enterprise especially hard? **(Section 2.6)**
- What is UML? How does database design fit into the overall design of a data-intensive software system? How is UML related to ER diagrams? **(Section 2.7)**

EXERCISES

Exercise 2.1 Explain the following terms briefly: *attribute, domain, entity, relationship, entity set, relationship set, one-to-many relationship, many-to-many relationship, participation constraint, overlap constraint, covering constraint, weak entity set, aggregation,* and *role indicator.*

Exercise 2.2 A university database contains information about professors (identified by social security number, or SSN) and courses (identified by courseid). Professors teach courses; each of the following situations concerns the Teaches relationship set. For each situation, draw an ER diagram that describes it (assuming no further constraints hold).

1. Professors can teach the same course in several semesters, and each offering must be recorded.
2. Professors can teach the same course in several semesters, and only the most recent such offering needs to be recorded. (Assume this condition applies in all subsequent questions.)
3. Every professor must teach some course.
4. Every professor teaches exactly one course (no more, no less).
5. Every professor teaches exactly one course (no more, no less), and every course must be taught by some professor.
6. Now suppose that certain courses can be taught by a team of professors jointly, but it is possible that no one professor in a team can teach the course. Model this situation, introducing additional entity sets and relationship sets if necessary.

Exercise 2.3 Consider the following information about a university database:

- Professors have an SSN, a name, an age, a rank, and a research specialty.
- Projects have a project number, a sponsor name (e.g., NSF), a starting date, an ending date, and a budget.
- Graduate students have an SSN, a name, an age, and a degree program (e.g., M.S. or Ph.D.).
- Each project is managed by one professor (known as the project's principal investigator).
- Each project is worked on by one or more professors (known as the project's co-investigators).
- Professors can manage and/or work on multiple projects.
- Each project is worked on by one or more graduate students (known as the project's research assistants).
- When graduate students work on a project, a professor must supervise their work on the project. Graduate students can work on multiple projects, in which case they will have a (potentially different) supervisor for each one.
- Departments have a department number, a department name, and a main office.
- Departments have a professor (known as the chairman) who runs the department.
- Professors work in one or more departments, and for each department that they work in, a time percentage is associated with their job.
- Graduate students have one major department in which they are working on their degree.

- Each graduate student has another, more senior graduate student (known as a student advisor) who advises him or her on what courses to take.

Design and draw an ER diagram that captures the information about the university. Use only the basic ER model here; that is, entities, relationships, and attributes. Be sure to indicate any key and participation constraints.

Exercise 2.4 A company database needs to store information about employees (identified by *ssn*, with *salary* and *phone* as attributes), departments (identified by *dno*, with *dname* and *budget* as attributes), and children of employees (with *name* and *age* as attributes). Employees *work* in departments; each department is *managed by* an employee; a child must be identified uniquely by *name* when the parent (who is an employee; assume that only one parent works for the company) is known. We are not interested in information about a child once the parent leaves the company.

Draw an ER diagram that captures this information.

Exercise 2.5 Notown Records has decided to store information about musicians who perform on its albums (as well as other company data) in a database. The company has wisely chosen to hire you as a database designer (at your usual consulting fee of $2500/day).

- Each musician that records at Notown has an SSN, a name, an address, and a phone number. Poorly paid musicians often share the same address, and no address has more than one phone.
- Each instrument used in songs recorded at Notown has a name (e.g., guitar, synthesizer, flute) and a musical key (e.g., C, B-flat, E-flat).
- Each album recorded on the Notown label has a title, a copyright date, a format (e.g., CD or MC), and an album identifier.
- Each song recorded at Notown has a title and an author.
- Each musician may play several instruments, and a given instrument may be played by several musicians.
- Each album has a number of songs on it, but no song may appear on more than one album.
- Each song is performed by one or more musicians, and a musician may perform a number of songs.
- Each album has exactly one musician who acts as its producer. A musician may produce several albums, of course.

Design a conceptual schema for Notown and draw an ER diagram for your schema. The preceding information describes the situation that the Notown database must model. Be sure to indicate all key and cardinality constraints and any assumptions you make. Identify any constraints you are unable to capture in the ER diagram and briefly explain why you could not express them.

Exercise 2.6 Computer Sciences Department frequent fliers have been complaining to Dane County Airport officials about the poor organization at the airport. As a result, the officials decided that all information related to the airport should be organized using a DBMS, and you have been hired to design the database. Your first task is to organize the information about all the airplanes stationed and maintained at the airport. The relevant information is as follows:

- Every airplane has a registration number, and each airplane is of a specific model.
- The airport accommodates a number of airplane models, and each model is identified by a model number (e.g., DC-10) and has a capacity and a weight.
- A number of technicians work at the airport. You need to store the name, SSN, address, phone number, and salary of each technician.
- Each technician is an expert on one or more plane model(s), and his or her expertise may overlap with that of other technicians. This information about technicians must also be recorded.
- Traffic controllers must have an annual medical examination. For each traffic controller, you must store the date of the most recent exam.
- All airport employees (including technicians) belong to a union. You must store the union membership number of each employee. You can assume that each employee is uniquely identified by a social security number.
- The airport has a number of tests that are used periodically to ensure that airplanes are still airworthy. Each test has a Federal Aviation Administration (FAA) test number, a name, and a maximum possible score.
- The FAA requires the airport to keep track of each time a given airplane is tested by a given technician using a given test. For each testing event, the information needed is the date, the number of hours the technician spent doing the test, and the score the airplane received on the test.

1. Draw an ER diagram for the airport database. Be sure to indicate the various attributes of each entity and relationship set; also specify the key and participation constraints for each relationship set. Specify any necessary overlap and covering constraints as well (in English).
2. The FAA passes a regulation that tests on a plane must be conducted by a technician who is an expert on that model. How would you express this constraint in the ER diagram? If you cannot express it, explain briefly.

Exercise 2.7 The Prescriptions-R-X chain of pharmacies has offered to give you a free lifetime supply of medicine if you design its database. Given the rising cost of health care, you agree. Here's the information that you gather:

- Patients are identified by an SSN, and their names, addresses, and ages must be recorded.
- Doctors are identified by an SSN. For each doctor, the name, specialty, and years of experience must be recorded.
- Each pharmaceutical company is identified by name and has a phone number.
- For each drug, the trade name and formula must be recorded. Each drug is sold by a given pharmaceutical company, and the trade name identifies a drug uniquely from among the products of that company. If a pharmaceutical company is deleted, you need not keep track of its products any longer.
- Each pharmacy has a name, address, and phone number.
- Every patient has a primary physician. Every doctor has at least one patient.
- Each pharmacy sells several drugs and has a price for each. A drug could be sold at several pharmacies, and the price could vary from one pharmacy to another.

- Doctors prescribe drugs for patients. A doctor could prescribe one or more drugs for several patients, and a patient could obtain prescriptions from several doctors. Each prescription has a date and a quantity associated with it. You can assume that, if a doctor prescribes the same drug for the same patient more than once, only the last such prescription needs to be stored.
- Pharmaceutical companies have long-term contracts with pharmacies. A pharmaceutical company can contract with several pharmacies, and a pharmacy can contract with several pharmaceutical companies. For each contract, you have to store a start date, an end date, and the text of the contract.
- Pharmacies appoint a supervisor for each contract. There must always be a supervisor for each contract, but the contract supervisor can change over the lifetime of the contract.

1. Draw an ER diagram that captures the preceding information. Identify any constraints not captured by the ER diagram.
2. How would your design change if each drug must be sold at a fixed price by all pharmacies?
3. How would your design change if the design requirements change as follows: If a doctor prescribes the same drug for the same patient more than once, several such prescriptions may have to be stored.

Exercise 2.8 Although you always wanted to be an artist, you ended up being an expert on databases because you love to cook data and you somehow confused *database* with *data baste*. Your old love is still there, however, so you set up a database company, ArtBase, that builds a product for art galleries. The core of this product is a database with a schema that captures all the information that galleries need to maintain. Galleries keep information about artists, their names (which are unique), birthplaces, age, and style of art. For each piece of artwork, the artist, the year it was made, its unique title, its type of art (e.g., painting, lithograph, sculpture, photograph), and its price must be stored. Pieces of artwork are also classified into groups of various kinds, for example, portraits, still lifes, works by Picasso, or works of the 19th century; a given piece may belong to more than one group. Each group is identified by a name (like those just given) that describes the group. Finally, galleries keep information about customers. For each customer, galleries keep that person's unique name, address, total amount of dollars spent in the gallery (very important!), and the artists and groups of art that the customer tends to like.

Draw the ER diagram for the database.

Exercise 2.9 Answer the following questions.

- Explain the following terms briefly: *UML, use case diagrams, statechart diagrams, class diagrams, database diagrams, component diagrams*, and *deployment diagrams*.
- Explain the relationship between ER diagrams and UML.

BIBLIOGRAPHIC NOTES

Several books provide a good treatment of conceptual design; these include [63] (which also contains a survey of commercial database design tools) and [730].

The ER model was proposed by Chen [172], and extensions have been proposed in a number of subsequent papers. Generalization and aggregation were introduced in [693]. [390, 589]

contain good surveys of semantic data models. Dynamic and temporal aspects of semantic data models are discussed in [749].

[731] discusses a design methodology based on developing an ER diagram and then translating it to the relational model. Markowitz considers referential integrity in the context of ER to relational mapping and discusses the support provided in some commercial systems (as of that date) in [513, 514].

The entity-relationship conference proceedings contain numerous papers on conceptual design, with an emphasis on the ER model; for example, [698].

The OMG home page (`www.omg.org`) contains the specification for UML and related modeling standards. Numerous good books discuss UML; for example [105, 278, 640] and there is a yearly conference dedicated to the advancement of UML, the International Conference on the Unified Modeling Language.

View integration is discussed in several papers, including [97, 139, 184, 244, 535, 551, 550, 685, 697, 748]. [64] is a survey of several integration approaches.