```
PERMIT(PermitId, LicensePlate, CarModel, StudentId)
STUDENT(SId, SName, GradYear, MajorId)
DEPT(DId, DName)
ENROLL(EId, Grade, StudentId, SectionId)
SECTION(SectId, YearOffered, Prof, CourseId)
COURSE(CId, Title, DeptId)
```

(a)

```
PERMIT(StudentId, LicensePlate, CarModel)
STUDENT(SId, SName, GradYear, MajorName)
DEPT(DName)
ENROLL(StudentId, SectionId, Grade)
SECTION(SectId, YearOffered, Prof, Title)
COURSE(Title, DeptName)
```

(b)

Figure 3-5
Two relational schemas generated from Figure 3-1

3.4 The Design Process

Now that we understand how to transform a class diagram into a relational schema, we can turn to the most critical part of all—how to create a well-designed class diagram. In this section we shall introduce a design process that has six steps.

Six Steps in the Design of a Class Diagram

Step 1. Create a requirements specification.
Step 2. Create a preliminary class diagram from the nouns and verbs of the specification.
Step 3. Check for inadequate relationships in the diagram.
Step 4. Remove redundant relationships from the diagram.
Step 5. Revise weak-weak and strong-strong relationships.
Step 6. Identify the attributes for each class.

The following subsections will discuss these steps in detail. As an example, we shall apply our design process to the university database. We shall see which design decisions led to the class diagram of Figure 3-1, and examine other design decisions that would have resulted in other class diagrams.

We should point out that this design process (like most design processes) is flexible. For example, Steps 3 through 6 need not be applied in a strict order. Moreover, the

activity of one step may produce an insight that causes the designer to revisit previous steps. It is not uncommon for a designer to repeat the design process several times, with each iteration resulting in a more accurate and more detailed diagram.

3.4.1 Requirements Analysis

The first step in the design of a database is to determine the data that it should hold. This step is called *requirements analysis*. There are several ways to obtain this information:

- you can ask the potential users of the database how they expect to use it;
- you can examine the data-entry forms that people will use;
- you can determine the ways that people intend to query the database;
- you can examine the various reports that will get generated from the data in the database.

The end product of this analysis is a text document that summarizes the requirements of the database; this document is called the *requirements specification*.

In our example, we assume that the university has asked us to design a database to support their student enrollment data. The result of our requirements analysis is the requirements specification of Figure 3-6.

The university is composed of departments. Academic departments (such as the Mathematics and Drama departments) are responsible for offering courses. Non-academic departments (such as the Admissions and Dining Hall departments) are responsible for the other tasks that keep the university running.

Each student in the university has a graduation year and majors in a particular department. Each year, the students who have not yet graduated enroll in zero or more courses. A course may not be offered in a given year; but if it is offered, it can have one or more sections, each of which is taught by a professor. A student enrolls in a particular section of each desired course.

Each student is allowed to have one car on campus. In order to park on campus, the student must request a parking permit from the Campus Security department. To avoid misuse, a parking permit lists the license plate and model of the car.

The database should:

- allow students to declare and change their major department;
- keep track of parking permits;
- allow departments, at the beginning of each year, to specify how many sections of each course it will offer for that year, and who will teach each section;
- allow current students to enroll in sections each year;
- allow professors to assign grades to students in their sections.

Figure 3-6
The requirements specification for the university database

3.4.2 The Preliminary Class Diagram

The second step in database design is to create a preliminary class diagram, by extracting relevant concepts from the requirements specification. We are primarily interested in nouns, which denote the real-world entities that the database will be modeling, and verbs, which denote the relationships between those entities. Figure 3-7 lists the nouns and verbs from Figure 3-6.

The requirements specification defines the *scope* of the database. For example, it is clear from the requirements specification that the database is intended to hold data related only to student enrollments—there is no mention of employee issues (such as salaries, work schedules, or job descriptions), financial issues (such as departmental budgets, expenses, or student financial aid), or resource issues (such as classroom allocation, or the tracking of library loans).

Once we understand the scope of the database, we can cast a more critical eye on the nouns and verbs used in the requirements specification. Consider again Figure 3-7. The nouns *university* and *campus* are irrelevant to the database, because its scope includes only the one university and its campus. Similarly, the verb *composed of* is also irrelevant. The noun *non-academic department* is also irrelevant, because the database contains only academic information. The noun *car* is irrelevant, because the database holds information about the permit, not the car. And finally, the verb *requests* is irrelevant, because the database only needs to store the data for each issued permit, and not the data related to a requested permit.

Of course, a designer does not make decisions such as these by fiat. Typically, the designer takes the time to discuss and clarify the issues with the database users, so that everyone has the same understanding of the database's scope.

Nouns	Verbs		
university	university	*composed of*	departments
academic department	academic dept	*offers*	course
non-academic department	course	*has*	sections
course	student	*graduates in*	year
student	student	*majors in*	department
year	student	*enrolls in*	section
grade	student	*receives*	grades
section	professor	*assigns*	grades
professor	professor	*teaches*	section
car	section	*offered in*	year
campus	student	*requests*	permit
permit	student	*is issued*	permit
license plate	permit	*lists*	license plate
car model	permit	*lists*	car model

Figure 3-7
Extracting nouns and verbs from the requirements specification

The designer uses the revised noun and verb list to construct a preliminary class diagram. Each noun maps to a class, and each verb maps to a relationship. There are no attributes yet. Figure 3-8 depicts the class diagram implied by the relevant nouns and verbs of Figure 3-7.

In order to make the class diagram more readable, we labeled each relationship with the verb that it denotes. You should convince yourself that the listed cardinalities correspond to your understanding of each relationship.

3.4.3 Inadequate Relationships

The relationships in a class diagram depict how the entities in the database connect to each other. For example, suppose we are given a STUDENT entity. Then by following the *majors in* relationship, we can find the DEPT entity corresponding to that student's major. Or by following the *enrolls in* relationship, we can find all of the sections that the student has enrolled in. Relationships can also be composed. Starting with a STUDENT entity, we can find the courses that the student has enrolled in, by following the relationships *enrolls in* and *has*.

The designer must check relationships carefully to see if they convey their intended meaning. For example, suppose that we had mistakenly assumed that students enroll in courses instead of sections, and had therefore written the *enrolls in* relationship so that it went between STUDENT and COURSE (instead of between STUDENT and SECTION). See Figure 3-9.

That class diagram has a problem. Given a student, we can determine the courses that the student enrolled in, and for each course, we can determine its sections. But we do not know which of those sections the student was in. We say that the *enrolls in* relationship in this diagram is *inadequate*.

Given a class diagram, there is no surefire way to tell if it contains an inadequate relationship. The only thing that the designer can do is to carefully examine each path through the class diagram to ensure that it expresses the desired information.

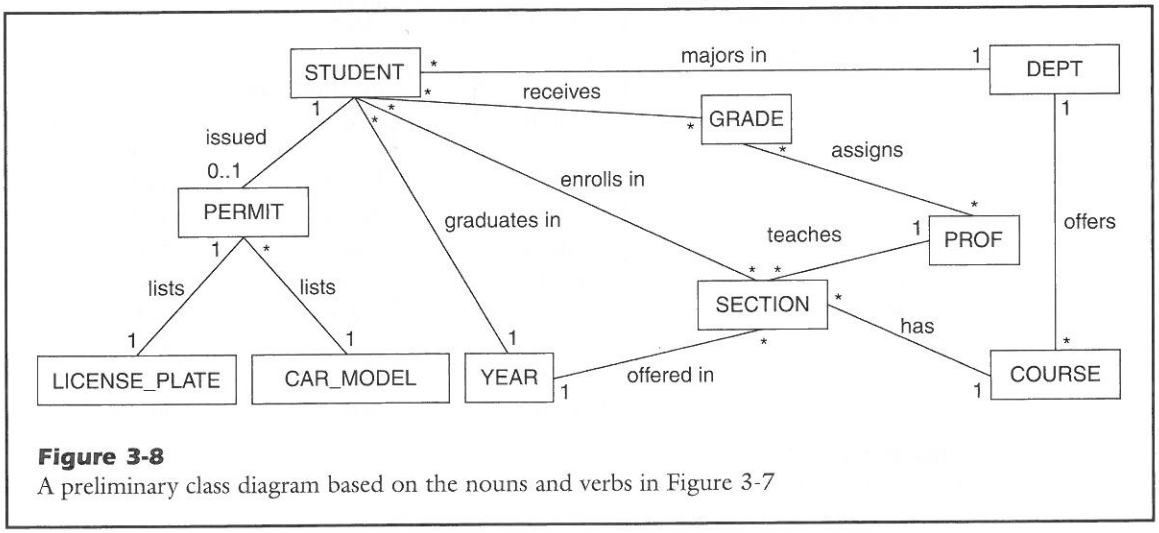


Figure 3-8
A preliminary class diagram based on the nouns and verbs in Figure 3-7

Figure 3-9
A class diagram containing an inadequate relationship

The diagram of Figure 3-8 has two inadequate relationships. Stop for a moment, look back at the figure, and see if you can tell what they are.

The two inadequate relationships are *receives* and *assigns*. Given a STUDENT entity, the *receives* relationship tells us what grades the student received, but there is no way to determine which section each grade was for. The *assigns* relationship is similarly problematic: It tells us which grades a professor gave out, but does not tell us to which student and in which section.

The cause of these two inadequate relationships was the simplistic way that we extracted them from the requirements specification. For example, it isn't enough to say "student receives grades." Instead, we need to say "student receives a grade for an enrolled section." Similarly, the *assigns* verb needs to be stated as "professor assigns grade to student in section."

These revised relationships are more complex than the other relationships we have seen, because they involve more than two nouns. They are often called *multi-way relationships*.

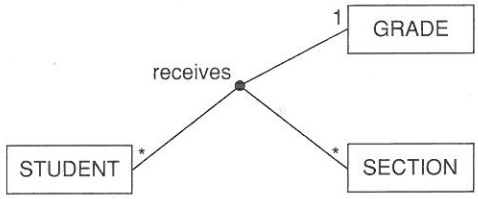
A *multi-way relationship* involves three or more classes.

There are two basic ways to represent a multi-way relationship in a class diagram. One way is to draw a line that connects more than two boxes. The other way is to turn the multi-way relationship into a class that has relationships to the participating classes. These two ways are depicted in Figure 3-10 for the *receives* relationship.

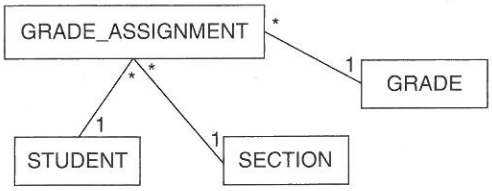
The annotations on the *receives* relationship in Figure 3-10(a) deserve explanation. The annotation next to GRADE is "1" because there will be exactly one grade for a student in a section. The annotation next to STUDENT is "*" because there can be many students in a section having the same grade. And the annotation next to SECTION is "*" because there can be many sections for which a student has the same grade.

In Figure 3-10(b), we replaced the relationship *receives* with the class GRADE_ASSIGNMENT. There will be a GRADE_ASSIGNMENT entity for each time a grade is assigned. (In terms of Figure 3-1, this class is the same as ENROLL, but we don't know that yet.)

These two approaches are notationally similar. For example, you can imagine the GRADE_ASSIGNMENT class "growing" out of the dot in the *receives* relationship. The approaches are also conceptually similar. The ability to turn a relationship into a class corresponds to the linguistic ability to turn a verb into a noun. For example, the



(a) Use a line that connects the three boxes



(b) Rewrite the relationship as a class

Figure 3-10
Two ways to handle a multi-way relationship

verb "receives" can become "reception" or the gerund "receiving." The technical term for this process is *reification.*

> *Reification* is the process of turning a relationship into a class.

In this chapter we shall adopt the second approach of Figure 3-10; that is, we choose to reify each multi-way relationship. There are two reasons. The first reason is that reification makes the class diagram easier to read, because multi-way relationships tend to be more awkward to decipher than regular binary relationships. The second reason is that the reified class provides more flexibility in later stages of the design process; for example, it can participate in other relationships and have its own attributes.

Returning to our running example, we also use reification to rewrite *assigns,* which is the other inadequate relationship in Figure 3-8. Here we need to create a new class that is related to STUDENT, SECTION, GRADE, and PROF. This new class will have one record for each grade assignment. Interestingly enough, we already have such a class, namely GRADE_ASSIGNMENT. We certainly don't need both classes in our diagram, so we combine them. The resulting class diagram appears in Figure 3-11.

3.4.4 Redundant Relationships

Consider Figure 3-11 and suppose we want to know the sections that a particular student has enrolled in. The most obvious tack is to follow the *enrolls in* relationship. However, with a bit of thought we can convince ourselves that the path from STUDENT

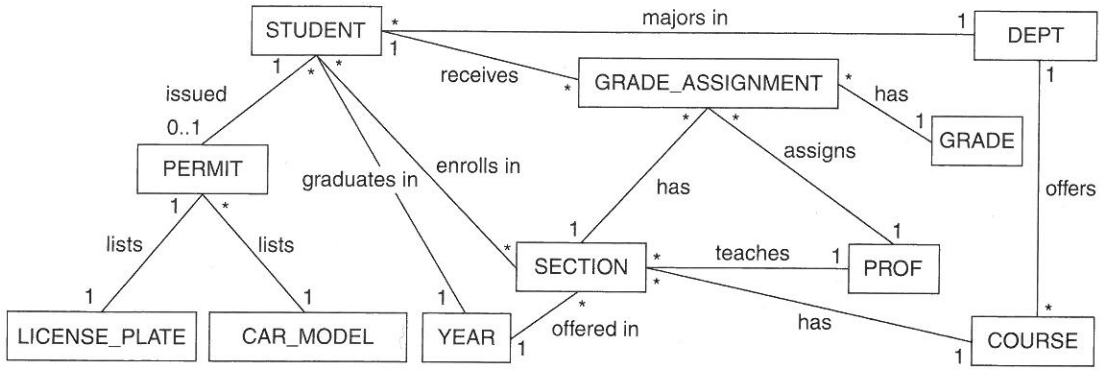


Figure 3-11
The class diagram that results from reifying the inadequate relationships

to GRADE_ ASSIGNMENT to SECTION has the same effect. In other words, the *enrolls in* relationship is *redundant*. The database doesn't need it. Removing the relationship from the class diagram will not change the content of the database.

> A relationship is *redundant* if removing it does not change the information content of the class diagram.

Redundant relationships are not desirable in a class diagram, for several reasons:

- *They are inelegant.* Ideally, each concept in a class diagram should have its own unique, proper spot.
- *They obfuscate the design.* A class diagram that is littered with redundant relationships is more difficult to understand.
- *They cause implementation problems.* If a user updates the database by modifying the contents of a redundant relationship without also modifying the non-redundant portion of the database (or vice versa), then there will be two different versions of the same relationship. In other words, the database will become inconsistent.

We express this concern as the following design rule:

> Strive to eliminate redundant relationships from the class diagram.

The relationship *assigns* (between PROF and GRADE_ASSIGNMENT) is another example of a redundant relationship. This relationship tells us, for each grade assignment, who assigned the grade. But if we look back at the requirements specification, we see that the grades for a section are always assigned by the professor of that section. Consequently,

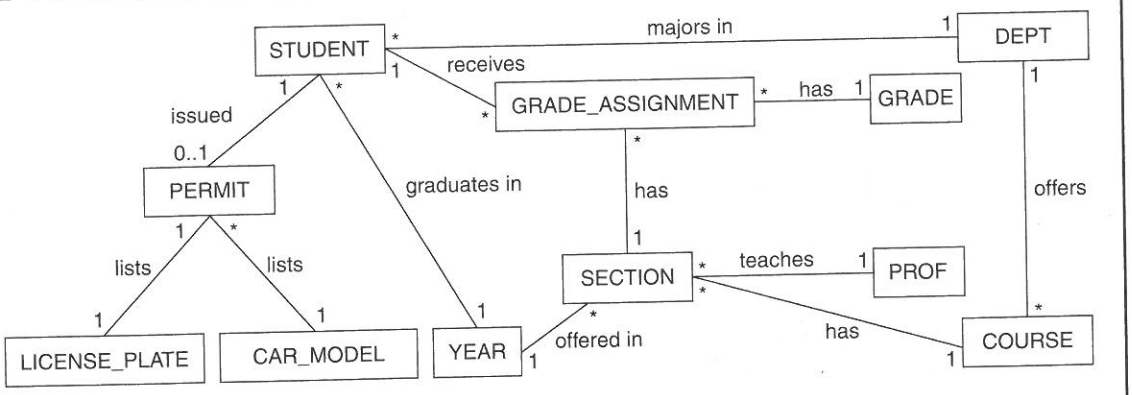


Figure 3-12
The class diagram that results from removing redundant relationships

we can get the same information by following the path from GRADE_ASSIGNMENT to SECTION to PROF. Therefore, the *assigns* relationship is redundant.

The designer of a class diagram must check each relationship for redundancy. For example, consider the *majors in* relationship between STUDENT and DEPT. It will be redundant if it is equivalent to the path from STUDENT to GRADE_ASSIGNMENT to SECTION to COURSE to DEPT. That path, however, denotes the departments that offer the courses taken by a student, which is quite different from the department that the student is majoring in. Thus we can infer that *majors in* is not redundant.

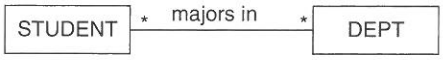
Figure 3-12 depicts the class diagram that results from removing the two redundant relationships. Exercise 3.1 asks you to verify that none of the relationships in this figure are redundant.

3.4.5 Handling General Relationships

The transformation algorithm of Figure 3-4 only handles weak-strong relationships. A class diagram, however, may contain relationships with two strong sides or two weak sides. In this section we consider these possibilities, and show how they can be rewritten in a way that uses only weak-strong relationships.

Weak-Weak Relationships Consider the *majors in* relationship between STUDENT and DEPT. This relationship is many-one in Figure 3-12, and asserts that each student must have exactly one major. Suppose instead that students at the university can have any number of majors; in this case, the annotation next to DEPT should be "*", resulting in the many-many relationship of Figure 3-13(a).

Figure 3-13(b) shows what happens when we reify that many-many relationship. The new class STUDENT_MAJOR has a relationship with each of STUDENT and DEPT. There will be one STUDENT_MAJOR entity for each major declared by each



(a) A many-many version of the *majors in* relationship

(b) Reifying the many-many relationship into two many-one relationships

Figure 3-13
Reifying a many-many relationship

student. That is, if Joe has two majors, and Amy and Max each have one major, then there will be four instances of STUDENT_MAJOR. Each student can thus have any number of related STUDENT_MAJOR entities, but each STUDENT_MAJOR entity corresponds to exactly one student. Thus the relationship between STUDENT and STUDENT_MAJOR is many-one. Similarly, the relationship between DEPT and STUDENT_MAJOR is also many-one.

In other words, reifying a many-many relationship creates an equivalent class diagram that no longer contains the many-many relationship. This useful trick allows us to get rid of unwanted many-many relationships. But it is more than just a trick. Most of the time, reifying a many-many relationship actually improves the quality of the class diagram.

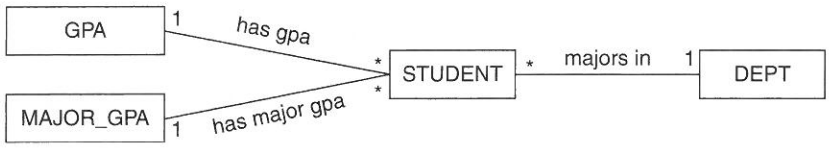
For example, suppose that our class diagram needs to store both the GPA and the major GPA for each student (where the "major GPA" is the GPA of courses taken in the student's major). Figure 3-14(a) shows the class diagram in the case when a student has exactly one major. This diagram has no problems. But suppose now that a student can have several majors. Then the student will also have several major GPAs, which means that the *has major gpa* relationship must also be many-many. Figure 3-14(b) depicts this class diagram.

The problem with Figure 3-14(b) is that the *has major gpa* relationship is now inadequate. Given a student, we can determine the various major GPAs, but we do not know which GPA corresponds to which major. The solution is to combine the *has major gpa* and *majors in* relationships into a single multi-way relationship, as shown in Figure 3-14(c). But now that we have created a multi-way relationship, we need to reify it anyway. The result is the new class STUDENT_MAJOR, as shown in Figure 3-14(d). Note how the MAJOR_GPA class winds up in a many-one relationship with STUDENT_MAJOR.

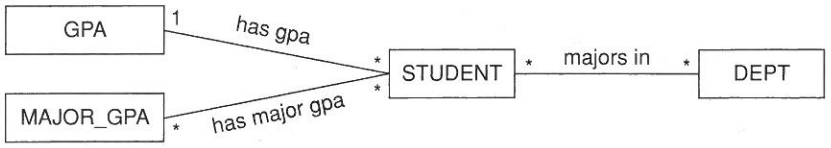
Reification can be used to remove all types of weak-weak relationships, not just many-many relationships.

For example, consider Figure 3-15(a), which shows the same diagram as before, except that we now suppose that students can have at most one major. The difference is that the annotation next to DEPT and MAJOR_GPA are now both "0..1".

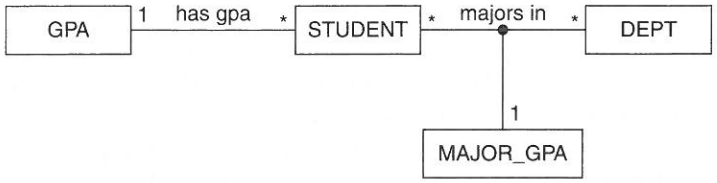
In this case, the relationship *has major gpa* is not inadequate (as it was in Figure 3-14), but it is problematic. The problem is that there is a dependency between *has major gpa* and *majors in*—namely, that a student should have a major GPA if and only if the



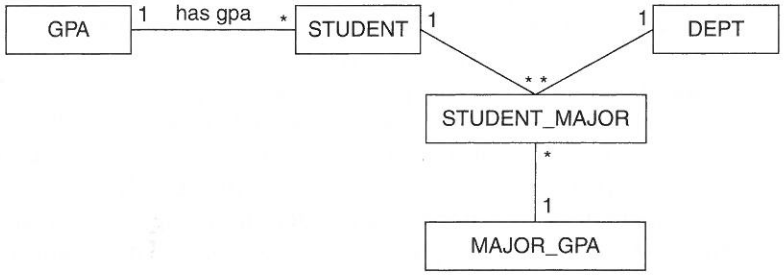
(a) Adding GPA information to the class diagram



(b) Modifying the diagram so that a student can have several majors



(c) Creating a multi-way relationship to fix the inadequacies of part (b)

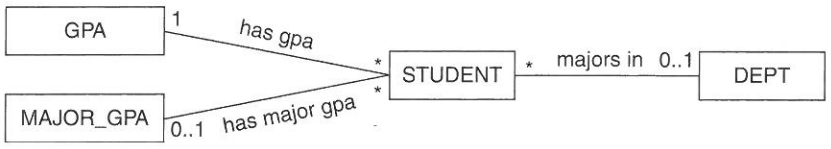


(d) The result of reifying the multi-way relationship

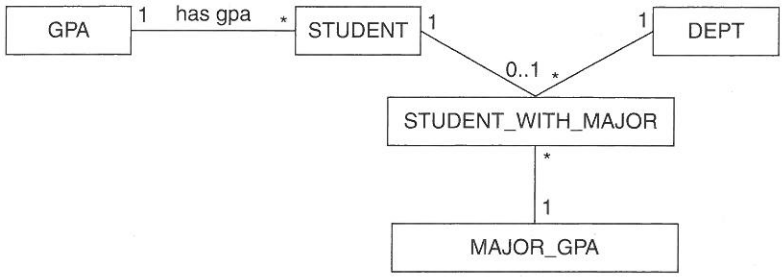
Figure 3-14
The need to reify a many-many relationship

student has a major. This dependency is inelegant, and indicates that they are really two aspects of the same multi-way relationship.

It thus makes sense to transform the class diagram in exactly the same way as before. We first merge the *majors in* and *has major gpa* relationships into a single multi-way relationship, and then reify that relationship. The result appears in Figure 3-15(b). Note the similarity of this class diagram with the analogous diagram of Figure 3-14(d); the difference



(a) Students can have at most one major



(b) The result of reifying the weak-weak relationship

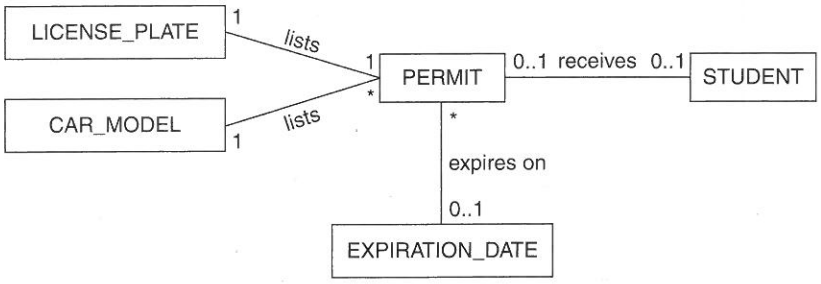
Figure 3-15
Reifying a weak-weak many-one relationship

is that the new class, which we call STUDENT_WITH_MAJOR, now has a "0..1" annotation in its relationship with STUDENT. This annotation reflects the fact that a student may have zero or one related STUDENT_WITH_MAJOR entity, but if there is one, then there is exactly one DEPT and MAJOR_GPA entity related to it.

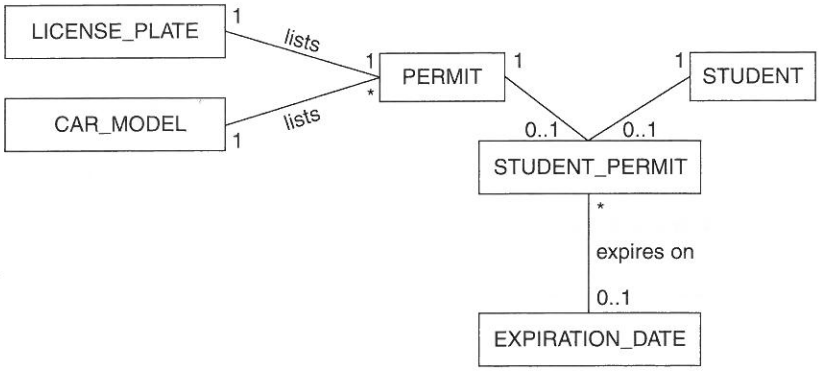
The preceding two examples showed how to reify many-many and weak-weak many-one relationships. The final type of weak-weak relationship to consider is when both sides of the relationship have the "0..1" annotation. For an example of this type of relationship, consider the relationship between PERMIT and STUDENT in Figure 3-12, and suppose that the university gives out parking permits to staff as well as students. Moreover, suppose that student permits are "crippled" in the sense that they have an expiration date, whereas staff permits do not. In this relationship, each student can have at most one permit, but only some of the students will have a permit; in addition, each permit can be issued to at most one student, but only some of the permits will be issued to students. This relationship is symmetrical, in that both sides are annotated with "0..1". It is a weak-weak one-one relationship. The class diagram is shown in Figure 3-16(a).

The problem with this diagram is the same as in the previous example. The relationship *expires on* is dependent on the relationship *receives*, and so they need to be combined into a single multi-way relationship. Figure 3-16(b) depicts the result of reifying that relationship.

The similarity of these three examples points out the essential nature of a weak-weak relationship, no matter what form it is in. A weak-weak relationship needs to be reified for two reasons. The first reason is a practical one: We do it because our transformation



(a) Student permits have an expiration date



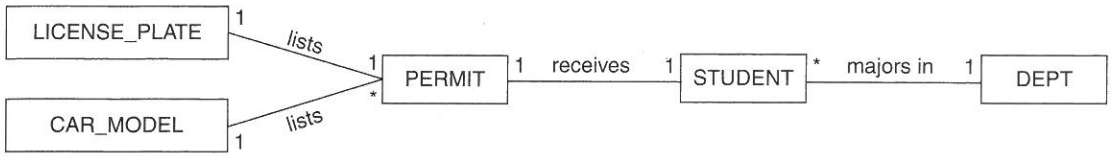
(b) The result of reifying the weak-weak relationship

Figure 3-16
Reifying a weak-weak one-one relationship

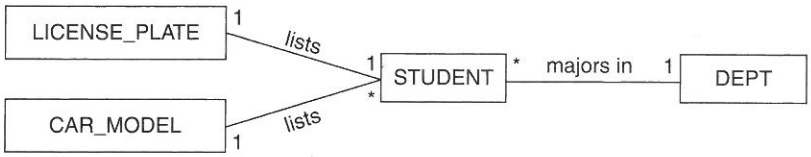
algorithm requires it. The second reason is conceptual: A weak-weak relationship often turns out to be a part of a multi-way relationship, and therefore will get reified anyway.

Strong-Strong Relationships Recall that a strong-strong relationship has "1" annotations on both sides. In a strong-strong relationship, the entities in the two classes are in complete one-to-one correspondence. For example, the relationship between STUDENT and PERMIT would be strong-strong if the university required every student to get a parking permit. See Figure 3-17(a).

One way to deal with a strong-strong relationship is to merge the two classes. Since the entities are in a one-one correspondence, they can be treated as different aspects of the same entity. Another way is to simply leave the relationship alone; when the time comes to transform the class diagram into a relational schema, we just treat one side of the relationship as if it were weak. This second way is important when one of the classes will be treated as an attribute of the other, as we shall see in the next section. Figure 3-17(b) illustrates both approaches. The classes PERMIT and STUDENT are merged, whereas the class LICENSE_PLATE is left alone.



(a) Each student must get a parking permit



(b) Merging the PERMIT and STUDENT classes

Figure 3-17
Dealing with strong-strong relationships

3.4.6 Adding Attributes to Classes

Classes vs. Attributes Up to this point, we have treated each noun as *a class*. It is now time to differentiate between two kinds of noun:

- nouns that correspond to an entity in the world, such as *student;*
- nouns that correspond to a value, such as *year*.

The first kind of noun corresponds to a class, and the second kind of noun corresponds to an *attribute*.

> A *class* denotes an entity in the world.
> An *attribute* denotes a value that describes an entity.

Actually, the distinction between *entity* and *value* is not as clear-cut as the above definition implies. For example, consider license plates. A license plate certainly is an entity in the real world. There are many things that a database could store about a license plate: its state, its design, its physical condition, and so on. However, our requirements specification expresses a noninterest in these things—to it, a license plate is just a string of characters that is written on a permit. Consequently, it is reasonable to represent a license plate as an attribute in that database. A different database, such as a database maintained by the registry of motor vehicles, might choose to represent license plates as a class.

Now consider professors. The requirement specification doesn't say much about professors, so let's assume that we only need to store the name of each professor. But that does not necessarily mean that professors should be represented using an attribute. The question is whether the database should treat a professor's name as a random character string, or

whether the database should maintain an explicit, "official" set of professor names. If the former case is true, then we want to represent professors using an attribute. If the latter case is true, then we need to keep PROF as a class.

Departments are in exactly the same situation as professors. We can choose to model departments as an attribute or as a class, depending on whether we want the database to explicitly keep track of the current departments.

The point is that the decision to model a noun as a class or as an attribute is not inherent in the real world. Instead, it is a decision about the relative importance of the noun in the database. A simple rule of thumb is this:

> We should represent a noun as a class if we want the database to keep an explicit list of its entities.

Transforming Classes into Attributes Given a class diagram, the algorithm of Figure 3-18 shows how to transform one of its classes into an attribute.

For example, consider again the class diagram of Figure 3-12, which appears in Figure 3-19(a) (although for consistency with Figure 3-1, we have now renamed the class GRADE_ASSIGNMENT as ENROLL). We would like to turn GRADE, PROF, YEAR, LICENSE_PLATE, and CAR_MODEL into attributes.

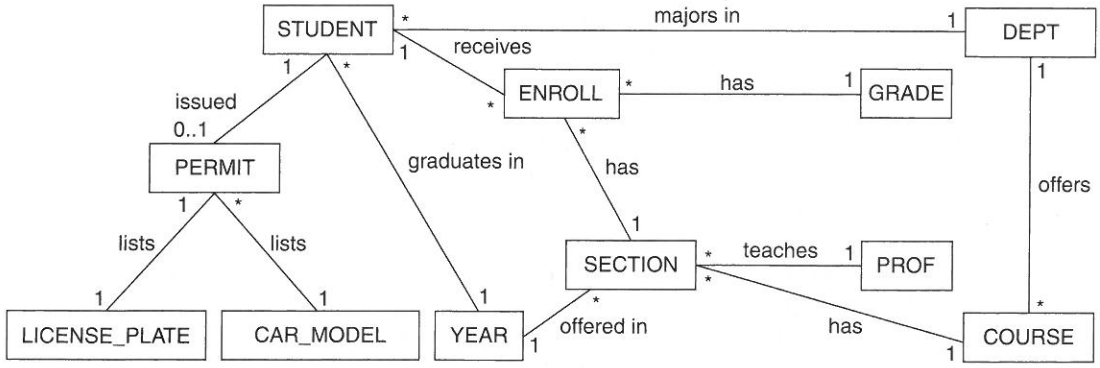
The class GRADE is related only to ENROLL. Thus we simply need to add an attribute *Grade* to that class, and remove GRADE from the diagram. Similarly, an attribute *Prof* can be added to the class SECTION, and attributes *LicensePlate* and *CarModel* can be added to class PERMIT. The class YEAR is related to both STUDENT and SECTION. Thus we must add an attribute to each of these classes. Figure 3-19(b) depicts the resulting class diagram.

Adding Additional Attributes to the Diagram The designer should also be on the lookout for useful attributes that are not mentioned in the requirements specification. Very often, the writer of a requirements specification assumes that the reader understands the basic concepts in the world to be modeled, such as the facts that students have names and courses have titles. Such concepts are typically not mentioned in the specification because they are too obvious to mention.

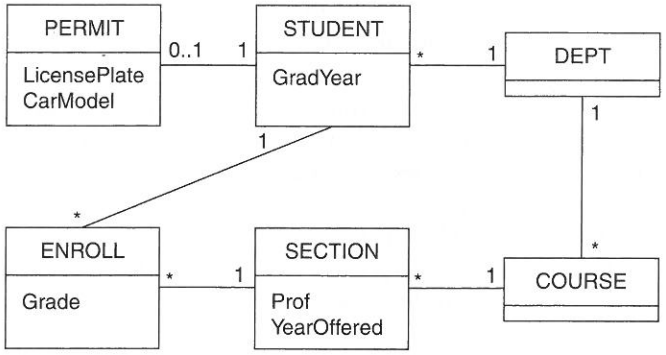
Once the designer has determined what classes will be in the class diagram, the designer can flesh out those classes with unstated, "obvious" attributes. In the example of Figure 3-19, we chose to add *SName* to STUDENT, *DName* to DEPT, and *Title* to COURSE. The resulting class diagram appears in Figure 3-19(c). You should note that this diagram is identical to that of Figure 3-1.

1. Let C be the class that we want to turn into an attribute.
2. Add an attribute to each class that is directly related to C.
3. Remove C (and its relationships) from the class diagram.

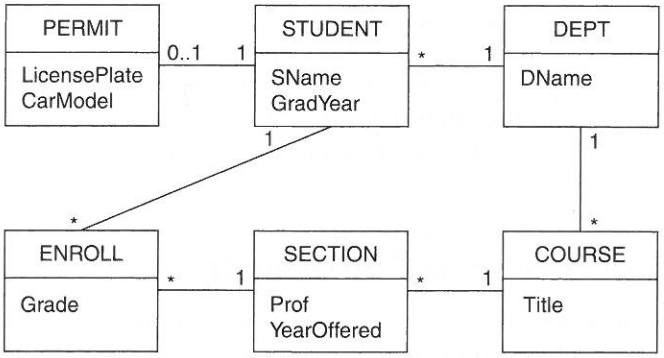
Figure 3-18
An algorithm to turn a class into an attribute



(a) The class diagram of Figure 3-12



(b) Turning the classes LICENSE_PLATE, CAR_MODEL, YEAR, GRADE, and PROF into attributes



(c) Adding new attributes *SName*, *DName*, and *Title*

Figure 3-19
Adding attributes to a class diagram

Attribute Cardinality Attributes have cardinality, similar to relationships. An attribute can be *single-valued* or *multi-valued*. As the name implies, an entity will have one value for each single-valued attribute, and can have several values for each multi-valued attribute.

For example, consider again the class PROF in Figure 3-19(a). The *teaches* relationship is many-one, meaning that each section has a single professor. Consequently, the attribute *Prof* in Figure 3-19(b) will be single-valued. If *teaches* had been many-many, then *Prof* would be multi-valued.

Single-valued attributes are easy to implement in a relational database—each attribute corresponds to a field of a table. A multi-valued attribute can be implemented as a collection type (such as a list or array). Many relational systems support such collection types. If your target system does not, then it would be best to avoid multi-valued attributes—for example, by first reifying the many-many relationship.

Compound Attributes In our discussion of the attribute-creation algorithm of Figure 3-18, we implicitly assumed that the transformed class had no attributes of its own. However, the algorithm actually works fine on classes that have attributes.

Consider for example Figure 3-19(c), and suppose that we decide that PERMIT should not be a class. The algorithm says that we should create an attribute *Permit* in STUDENT, but it is vague on what the value of that attribute should be. Clearly, the value should involve the values of *LicensePlate* and *CarModel* somehow. One possibility is to combine these two values into a single value; for example, the value of *Permit* could be a string of the form "Corolla with plate #ABC321." Another possibility is to create a structured attribute value that contains the two underlying values. Such an attribute is called a *compound attribute,* because its value is a structure that consists of multiple subvalues.

For another example of a compound attribute, suppose that each student has an address, which consists of a street number, a city, and a postal code. We could give STUDENT three additional attributes, or we could create a compound attribute, *Address,* that is composed of these three sub-values. Compound attributes are often useful in queries. For example, a user can refer to an address either in its entirety, as a single unit; or the user can refer to the individual pieces separately.

Most commercial database systems support structured values, which means that a compound attribute can be implemented as a single field of a table. If your database system does not, then it is best to avoid compound attributes.

3.5 Relationships as Constraints

Each many-one relationship in a class diagram embeds a constraint. For example, the *major of* relationship implies that a student must have exactly one major department. When we transform a class diagram into a relational schema, we must be sure that the database system will be able to enforce the constraints implied by its many-one relationships.

As we have seen, a many-one relationship is implemented by a foreign key in the weak-side table, and referential integrity ensures that each weak-side record will have exactly one related strong-side record. So in general, there is no problem—referential

integrity guarantees that the transformed relational database will satisfy each many-one constraint implied by the class diagram.

Actually, there is one area where a problem can arise. This problem area concerns the way that we reify multi-way relationships. Consider again Figure 3-10. The multi-way relationship *receives* is many-one, because there is exactly one grade for a given student and section. However, note that this constraint is missing from the reified class GRADE_ASSIGNMENT. Its relationships state that each grade assignment must have a student, section, and grade, but there is nothing that prohibits a student from having two grade assignments for the same section. This constraint is also missing from the transformed relational schema of Figure 3-5(a), because several ENROLL records could have the same value for *StudentId* and *SectionId*.

The solution to this dilemma is to declare {*StudentId, SectionId*} to be the key of ENROLL, as in Figure 3-5(b). This key ensures that a student cannot enroll in a section more than once, and therefore cannot have more than one grade for a given section.

Generalizing this solution results in the following principle:

> Let T be a table corresponding to a reified relationship.
> The key of T should consist of the foreign keys that correspond to the "many" sides of that relationship.

This principle assumes, of course, that table T contains all of the necessary foreign keys, which implies that the class diagram must contain all of the relationships created by the reification. Unfortunately, this assumption will not be true if some of these relationships are redundant, and thus get eliminated from the class diagram. Consider the following example.

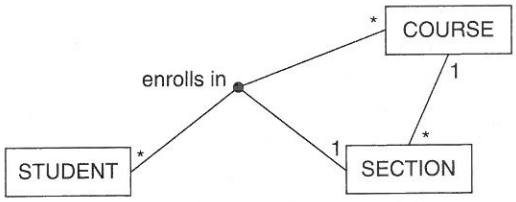
Suppose that the requirements specification for the university database contained the requirement shown in Figure 3-20(a). This requirement extends the relationship *enrolls in* of Figure 3-2. Whereas the relationship was originally just between STUDENT and SECTION, this new requirement forces the relationship to be multi-way, between STUDENT, SECTION, and COURSE. This relationship is depicted in Figure 3-20(b), along with the many-one relationship between SECTION and COURSE. Figure 3-20(c) shows what happens when we reify *enrolls in*, creating the new class ENROLL. And Figure 3-20(d) shows the tables that would be generated from these classes. We chose the key of ENROLL according to the above principle; that is, the fields {*StudentId, CourseId*} need to be the key of ENROLL in order to enforce the many-one constraint of the *enrolls in* relationship.

Note that the relationship between ENROLL and COURSE in Figure 3-20(c) is redundant, because the same information can be obtained by going from ENROLL to SECTION to COURSE. However, removing that relationship from the class diagram causes the field *CourseId* to not be part of the ENROLL table, which means that we will not be able to enforce the many-one constraint.

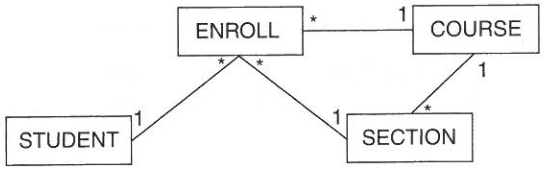
This example illustrates a serious conundrum. How should we deal with the relationship between ENROLL and COURSE? On one hand, if we leave the relationship in the class diagram then the resulting ENROLL table will contain the redundant field *CourseId*, and the potential for inconsistent data. For example, consider Figure 3-20(d);

A student can enroll in a section of a course only if the student has not taken that course before.

(a) The additional requirement



(b) Revising *enrolls in* to be a 3-way relationship



(c) The corresponding reified class ENROLL

```
ENROLL(StudentId, CourseId, SectionId)
SECTION(SectId, CourseId)
STUDENT(SId)
COURSE(CId)
```

(d) The relational schema generated from the class diagram

Figure 3-20
Adding a requirement that cannot be easily enforced

there is nothing to stop a user from inserting a record into ENROLL that has the wrong course ID for its section. On the other hand, if we remove the relationship from the class diagram, then the resulting ENROLL table will not be able to enforce its key constraint. That is, there would be nothing to stop a user from inserting multiple records into ENROLL for the same student and course.

As far as anyone knows, this conundrum has no satisfactory resolution. The designer must choose whether to keep the redundant relationship and live with the potential for inconsistency, or to remove the relationship and deal with a possible constraint violation. For example, if we choose to keep the redundant relationship, then we can avoid inconsistency by forcing users to update ENROLL via a customized data-entry form; this form would not let the user specify a value for *CourseId*, but would automatically determine its proper value from the SECTION table. Or if we choose to remove the redundant relationship, then we can enforce the key constraint via an explicit assertion (as we will see in Figure 5-3).

3.6 Functional Dependencies and Normalization

3.6.1 Functional Dependencies

In order to create a well-designed class diagram, we must analyze each relationship carefully to determine if any are redundant. As we have seen, the problem with redundant relationships is that they introduce redundant fields in the generated tables, which can lead to inconsistent data.

However, it is not enough to focus on the class diagram. As we shall see, redundancy can creep into the generated tables in other ways, such as via overzealous attribute choices and overlooked constraints. In this section, we show how *functional dependencies* can be used to check for redundancy in the generated tables.

We introduce functional dependencies with an example: Suppose the university has a rule that says that every professor must teach exactly one course. This rule constrains the SECTION table, because two records having the same value for the *Prof* field must also have the same value for the *CourseId* field. Therefore, the *Prof* field *functionally determines* the *CourseId* field. We call this constraint a *functional dependency* (or *FD*).

We write this FD using the notation `Prof→CourseId`. The field *Prof* is the left-hand side (or LHS) of the FD, and *CourseId* is the right-hand side (or RHS).

The left-hand side of a functional dependency may have more than one field. For example, suppose the university sets a rule stating that all sections of a course during the year must be taught by the same professor. This rule implies that if two SECTION records have the same values for *CourseId* and *YearOffered*, then they must also have the same value for *Prof*; that is, the FD `CourseId,YearOffered→Prof` holds.

> The functional dependency $A_1A_2...A_n \rightarrow B$ asserts that two records having the same values for the A_i fields must also have the same value for the field B.

Since FDs are constraints, you cannot discover them by looking at the current contents of a table. Instead, you have to consider the rules governing the contents of the table. For example, here are some potential FDs for the SECTION table in Figure 1-1 (we have already seen the first two):

1. If a professor always teaches the same course every year, then the FD `Prof→CourseId` holds.
2. If all sections of a course during a year are taught by the same professor then the FD `CourseId,YearOffered→Prof` holds.
3. If all of the sections that a professor teaches in a given year are for the same course, then the FD `Prof,YearOffered→CourseId` holds.
4. If courses are taught at most once a year, then the FD `CourseId,YearOffered→SectId` holds.

An FD is a generalization of a superkey. In Chapter 2, we saw that if *K* is a superkey of a table then there cannot be two records having the same values for *K*. It follows that the FD *K→B* will be valid for every other field *B* in the table.

We can use the set of FDs for a table to infer its superkeys. For example, assume that *SectId* is a key of the SECTION table, and suppose that FD #4 above holds. We can infer that SECTION cannot have two records having the same values for {*CourseId, YearOffered*}, because the FD implies that those two records have the same value for *SectId*, which is impossible since *SectId* is a key. It follows that {*Course, YearOffered*} must be a superkey of SECTION.

In general, suppose that $\{K_1,...,K_n\}$ is known to be a superkey and let X be some set of fields. Exercise 3.15 asks you to show that if the FD $X \rightarrow K_i$ holds for each i, then X must also be a superkey.

Section 2.3 showed how a superkey can have superfluous fields. An FD can also contain superfluous fields for similar reasons. If you add a field to the left side of an FD, then you get another FD whose constraint is less restrictive; so if the first FD holds, then so does the second one. For example, suppose that `Prof→Course` holds, meaning that a professor always teaches the same course. Then the FD `Prof, YearOffered→Course` must also hold, because it asserts that a professor teaches just one course in any given year. We say that an FD has a *superfluous field* in its LHS if removing that field results in another FD.

FDs having superfluous fields are not interesting. We therefore restrict our attention to *full FDs:*

> A *full FD* is an FD that does not have superfluous fields.

3.6.2 Non-Key-Based FDs

Consider the LHS of a full FD. If these fields form a key, then we say that the FD is *key-based.*

> A full FD is *key-based* if the fields in its LHS form a key.

Key-based FDs are essentially key specifications, and thus are a natural part of a relational schema. Non-key-based FDs, however, are quite different. In fact, they almost certainly result from a flaw in the database design.

> A table that has a non-key-based FD contains redundancy.

For example, suppose that the FD `Prof→Course` holds for the table SECTION. Since the FD is not key-based, the table can contain several records having the same *Prof* value; these records will all have the same redundant value for *Course.*

There are several ways that non-key-based FDs can arise in a relational schema. We shall consider three common ones.

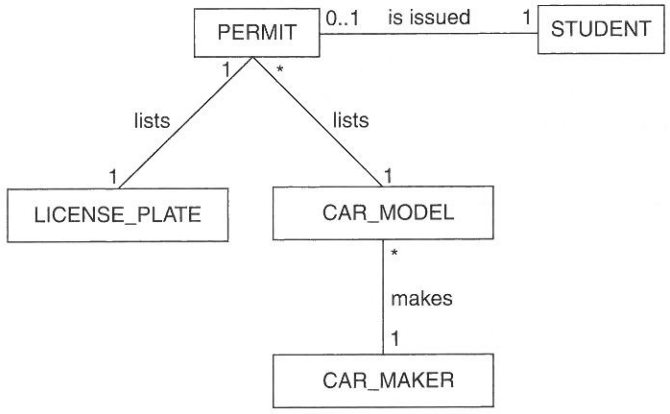
Choosing Inappropriate Attributes A non-key-based FD can occur when a designer turns a class into an attribute. For an example, suppose that the requirements specification says that a parking permit should not only list the license plate and model

or the car, but also its manufacturer. Figure 3-21(a) shows the revised class diagram. Note that the new class CAR_MAKER has a many-one relationship to CAR_MODEL, because every model has a specific manufacturer.

Now suppose that when the time comes to select attributes, we choose to create attributes for the license plate, car model, and car manufacturer, and add them to class PERMIT; see Figure 3-21(b). These three attributes seem perfectly reasonable; after all, the PERMIT table is almost the same as it was in Figure 3-5(a), and the new attribute *CarMaker* seems fairly innocuous.

However, the attribute *CarMaker* is actually problematic, because it causes the PERMIT table to have the non-key-based FD `CarModel→CarMaker`. And the presence of this FD means that the table will contain redundancy—namely, any two permits for the same model will always have the same value for the field *CarMaker*. This redundancy, as always, can lead to an inconsistent database; for example, there is no way to stop Campus Security from issuing one permit for a "toyota corolla" and another permit for a "ford corolla."

A designer has two ways to deal with this redundancy. The first way is to ignore it. The rationale would be that although the FD is valid in the real world, it is totally



(a) The revised class diagram

```
PERMIT(PermitId, LicensePlate, CarModel,
       CarMaker, StudentId)
```

(b) Choosing *LicensePlate*, *CarModel*, and *CarMaker* to be attributes of PERMIT

```
PERMIT(PermitId, LicensePlate, CarModelId, StudentId)
CAR_MODEL(CarModelId, ModelName, Maker)
```

(c) Keeping CAR_MODEL as a class

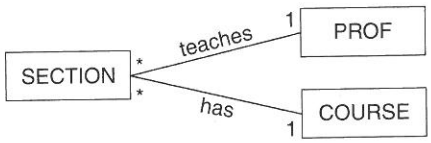
Figure 3-21
Adding the car manufacturer to the database

uninteresting in the database. That is, the database really doesn't care about what models each car manufacturer makes. The purpose of the *CarMaker* field is simply to help identify a car, and an inconsistent value is not a big deal.

The second way is to enforce the FD via a foreign key. In order to do this, we need CAR_MODEL to be a class, not an attribute. The corresponding CAR_MODEL table will have a record for each car model, listing the model's name and manufacturer; see Figure 3-21(c). In this case, the CAR_MODEL table acts as the "official" list of all models; if a student has a car whose model is not on the list, then the list will need to be updated before the permit can be issued.

Omitted Relationships A non-key-based FD can also arise when the designer omits a many-one relationship from the class diagram. Suppose for example that the university has the constraint that every professor teaches exactly one course. This constraint may not sound like a relationship, and so we may forget to include a relationship between PROF and COURSE in our class diagram. Figure 3-22(a) depicts our class diagram, and Figure 3-22(b) depicts its corresponding relational schema. It is not until we examine the schema for the table SECTION that we realize that the constraint applies and the non-key-based FD *Prof→CourseId* holds.

The solution to this problem is to fix the class diagram and then regenerate the relational schema. In this case, we need to add a many-one relationship between PROF and COURSE. This new relationship causes the relationship between SECTION and COURSE to become redundant, and so we omit it. The resulting relational schema appears in Figure 3-22(c), and contains no non-key-based FDs.



(a) The original class diagram

```
SECTION(SectId, Prof, CourseId)
COURSE(CourseId, Title)
```

(b) The generated relational schema

```
SECTION(SectId, ProfId)
PROF(ProfId, PName, CourseId)
COURSE(CourseId, Title)
```

(c) The generated schema after adding the omitted relationship and removing the redundant relationship

Figure 3-22
Dealing with an omitted relationship between PROF and COURSE

Note how the relationship between PROF and COURSE means that we can no longer choose to turn PROF into an attribute; it must remain a class, in order to enforce the many-one constraint.

Undetected Redundant Relationships It is not easy to detect redundant relationships in a class diagram, especially when it is large. Consider again the class diagram of Figure 3-20(c) and its corresponding relational schema of Figure 3-20(d). In retrospect, we know that the relationship between ENROLL and COURSE is redundant, and can be omitted. But even if we miss detecting this redundant relationship in the class diagram, we still have the chance to discover it in the relational schema. In particular, the ENROLL table of Figure 3-20(d) has the non-key-based FD `SectionId→CourseId`. This FD identifies the redundant relationship for us. The solution to the problem is therefore to go back to the class diagram and remove the redundant relationship, if we desire.

3.6.3 Normal Forms

A well-designed relational schema is said to be *normalized*. There are two relevant normal forms.

> A relational schema is in *Boyce-Codd Normal Form (BCNF)* if all full FDs are key-based.

> A relational schema is in *Third Normal Form (3NF)* if the RHS field of all non-key-based FDs belongs to some key.

The definition of 3NF is less restrictive than BCNF, because it allows a table to contain certain non-key-based FDs. In particular, suppose that the right side of a non-key-based FD f contains a key field. Then that FD would be allowed under 3NF, but not under BCNF.

This non-key-based FD f causes redundancy. However, if we remove the redundancy (say, by moving the fields of f to another table), then we will not be able to enforce the key. We saw an example of this situation in Figure 3-20, where f is the non-key-based FD `SectionId→CourseId`. The generated table ENROLL is not in BCNF, because of the FD. However, the table is in 3NF because the right-hand side of that FD, *CourseId*, is part of the key of ENROLL.

That example demonstrates that neither of the above definitions is perfect. The 3NF schema is "under-normalized," in the sense that the FD `SectionId→CourseId` is part of ENROLL and causes redundancy. On the other hand, the BCNF schema that results from removing the redundant relationship is "over-normalized," in the sense that the FD `StudentId,CourseId→SectionId` has been split between two tables, and must be enforced via a multi-table constraint.

The choice of which normal form to use basically comes down to a tradeoff between the conflicting FDs. How important is each FD? Will the potential inconsis-

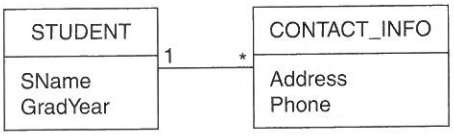
tencies of the 3NF schema be significant? How much do we care about enforcing the multi-table constraint in the BCNF schema? Fortunately, experience has shown that this dilemma occurs infrequently, and so the difference between 3NF and BCNF is usually irrelevant.

3.6.4 Normalized Does Not Mean Well Designed

Given a relational schema, we examine it for non-key-based FDs in order to discover and fix flaws in its class diagram. Each non-key-based FD denotes a source of undesirable redundancy. However, a schema in normal form is not guaranteed to be well-designed, because not all instances of redundancy can be represented by FDs.

For example, suppose that we want our database to hold student email addresses and phone numbers. We decide to create a class CONTACT_INFO that contains the fields *Address* and *Phone;* see Figure 3-23(a). If a student can have several email addresses and several phone numbers, then it is not immediately obvious how these values should be stored in CONTACT_INFO records. We decide to use the following strategy: Each student will have a record for every combination of *Address* and *Phone* values. For example, if Joe has two email addresses and three phone numbers, then we will store 2x3=6 records for him in CONTACT_INFO; see Figure 3-23(b).

This design is amazingly bad. There is a lot of redundancy: Each email address is repeated as many times as the student has phone numbers, and vice versa. But this redundancy cannot be identified using FDs. The CONTACT_INFO table is in BCNF.



(a)

CONTACT_INFO	InfoId	Address	Phone	StudentId
	91	joe@yahoo.com	6-1234	1
	92	joe@gmail.com	6-1234	1
	93	joe@yahoo.com	3-5555	1
	94	joe@gmail.com	3-5555	1
	95	joe@yahoo.com	9-9998	1
	96	joe@gmail.com	9-9998	1
	97	amy@gmail.com	6-4622	2
	98	amy@gmail.com	8-2323	2

(b)

Figure 3-23
Redundancy that FDs cannot catch

It turns out that this kind of redundancy corresponds to another kind of constraint, called a *MultiValued Dependency* (or *MVD*). If we desired, we could revise our normalization criteria to deal with MVDs. The resulting schema would be in a form called *Fourth Normal Form* (*4NF*).

Unfortunately, there are other examples of redundancy that are not caught by MVDs. In the early years of the relational model, researchers developed increasingly more complex dependencies and normal forms, with each new normal form handling a situation that the previous normal forms could not handle. The goal was to be able to have a definition of "normal form" that always produced well-designed schemas.

That goal has been largely abandoned. Database design is still more of an art than a science. In the end, it is up to the database designer to craft an appropriate class diagram. Constructs such as FDs can help, but they cannot substitute for intelligent design and careful analysis.

3.7 Chapter Summary

- The information in a database is often vital to the functioning of an organization. A poorly designed database can be difficult to use, run slowly, and contain the wrong information.
- A good way to design a database is to construct a *class diagram*. A class diagram is a high-level representation of the database that focuses on what information the tables should hold and how they should relate to each other.
- A class diagram is built out of *classes* and *relationships*. A class denotes a table, and is represented in the diagram by a box. A relationship denotes (to some extent) a foreign key connection between two tables, and is represented by a line connecting two boxes.
- Each side of a relationship has a *cardinality* annotation that indicates, for a given record, how many other records can be related to it. An annotation may be a "1", which indicates exactly one related record; a "*", which indicates any number of related records; or a "0..1", which indicates at most one related record. A relationship is said to be *many-many*, *many-one*, or *one-one*, depending on the annotations on each side.
- The "1" annotation is called a *strong annotation*, because it asserts that an entity cannot exist without a related entity. The other annotations are *weak annotations*. A foreign key corresponds to a weak-strong relationship.
- The algorithm of Figure 3-4 generates tables from a class diagram. This algorithm generates one table per class. The table contains a field for each of the class's fields, and a foreign key field for each relationship where the class is on the weak side.
- A relationship with two weak sides is called a *weak-weak* relationship. One example of such a relationship is a many-many relationship. The transformation algorithm does not generate tables for weak-weak relationships. Instead, we must first *reify* the relationship. We replace the relationship by a new class, which has weak-strong rela-