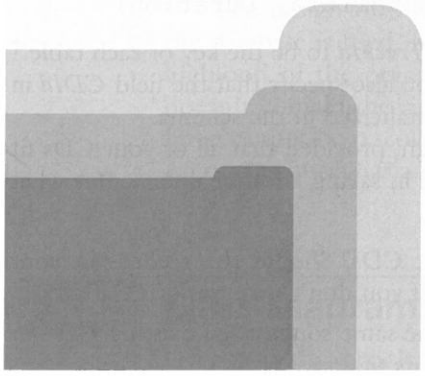
3
Data Design



Designing a database is an important task, and is much more difficult than it might appear. This chapter presents a methodology for designing relational databases. In this methodology, we begin by analyzing the data and structuring it into *entities* and *relationships*. We use a *class diagram* to express this structure.

A class diagram eventually needs to be turned into a relational schema. We will examine an algorithm that performs this transformation. This algorithm can handle only certain class diagrams (namely, those diagrams where the relationships correspond to foreign keys). We therefore consider what to do about other, more general relationships between classes. We show that the presence of such relationships indicates an incomplete design; we also show how to restructure the class diagram so that it not only is better, but it also no longer needs the general relationships.

We also address the question of what it means to be a good design. We examine two kinds of problematic relationships—*inadequate relationships* and *redundant relationships*—and show how to eliminate them from a class diagram. We then introduce *functional dependencies*, and show how they can be used as a tool to identify and fix redundancies in a class diagram.

3.1 Designing Tables Is Difficult

A relational database is just a set of tables. If you want to create a new database, all you have to do is figure out what data you need to keep and design some tables to hold that data. What is so difficult about that?

Let's try a simple example. Suppose that you want to use a database to catalog your CD music collection. Each CD has some data you want to store, such as its title, release year, band, and genre. Each individual track on the CD also has data, such as its track number, song title, and duration. So you create two tables: one to hold CD data, and one to hold track data. The tables have the following schema.

```
CD(CDId, Title, ReleaseYear, Band, Genre)
TRACK(TrackId, CDId, Track#, SongTitle, Duration)
```

You decide to create artificial fields *CDId* and *TrackId* to be the key of each table.† These fields are underlined in the above schema. You also specify that the field *CDId* in TRACK is a foreign key of CD; this foreign key is italicized in the schema.

Frankly, there is nothing wrong with this design, provided that all of your CDs fit this simple structure and that you aren't interested in saving additional data. But what would you do about the following issues?

- A "best of" CD contains tracks from different CDs. Should the records in your TRACK table reference the original CD? What if you don't own the original CD?
- What if you burn your own CD that contains the same songs as an existing CD but in a different order? Is it the same CD? What is its title?
- Some CDs, such as "Best of 90s Salsa," contain tracks from multiple bands. Do you have to store the band name with each track individually?
- Suppose you burn a CD that contains music from different genres. Do you save the genre with the track, or do you create a new genre for the CD, such as "party mix"?
- Do you want to keep data on the individual performers in the band? If so, do you want to be able to know about band members who create solo albums or move to different bands? A track may have appearances by "guest artists" who perform only on that track. How will you represent this?
- Do you want to keep data on each song, such as who wrote it and when? Often a track will be a cover of a song that was written and performed by another band. How will you represent that? Do you want to be able to create a playlist of every version of a song in the order of its performance?
- Do you want to save song lyrics? What about the author of those lyrics? What will you do with songs that do not have lyrics?

We could go on, but the point is this: As you widen the scope of what you want the database to contain, complexity tends to creep in. Choices need to be made about how specific the data will be (e.g., should *genre* be stored with a band, a CD, a track of a CD,

†A track ID uniquely identifies a track of any CD, as opposed to a track#, which is the location of the track on a particular CD.

or a song?), how extensive the data will be (e.g., do we really want to know who wrote a song?), and how interrelated the data will be (e.g., should we be able to compare different versions of the same song?).

Try for yourself to come up with a good set of tables to hold all of this data. How do you decide what tables to create, and what fields those tables should have? What should the foreign keys be? And most importantly, how do you *know* that your design is a good one? What started out as a simple, easily-understood, two-table database has turned into something much less simple and quite likely much less easy to understand.

Many corporate databases are even more complex. For example, peruse the *amazon.com* or *imdb.com* websites and note the different kinds of data they manage. A hit-or-miss design strategy just isn't going to work for these databases. We need a methodology that can guide us through the design process.

The design of a database has consequences that can last far into the future. As the database is used, new data is collected and stored in it. This data can become the lifeblood of the organization using the database. If it turns out that the database doesn't contain the right data, the organization will suffer. And because missing data often cannot be re-collected, it can be very difficult to restructure an existing database. An organization has a lot of incentive to get it right the first time.

3.2 Class Diagrams

Over the years, people have discovered that the best way to design a relational database is to *not* think about tables. The reason is that a database is a model of some portion of the world, and the world is just not organized as tables. Instead, the world tends to be organized as a network of related entities. Thus the best approach is to first construct a *class diagram,* which is a model of the world in terms of entities and relationships, and to then translate that diagram to a relational schema. This section introduces class diagrams and their terminology. The following sections show how to translate a class diagram to a relational schema, and how to create a well-designed diagram.

3.2.1 Classes, Relationships, and Attributes

People have developed many different notational systems for depicting entities and their relationships. The pictures we shall draw are called *class diagrams,* and are taken from the design language *UML.*

> A *class diagram* describes the entities that the database should hold and how they should relate to each other.

A class diagram uses three constructs: *class, relationship,* and *attribute.* A class denotes a type of entity, and is represented in the diagram by a box. A relationship denotes a meaningful correspondence between the entities of two or more classes, and is represented by a line connecting those classes. An attribute is used to describe an entity.

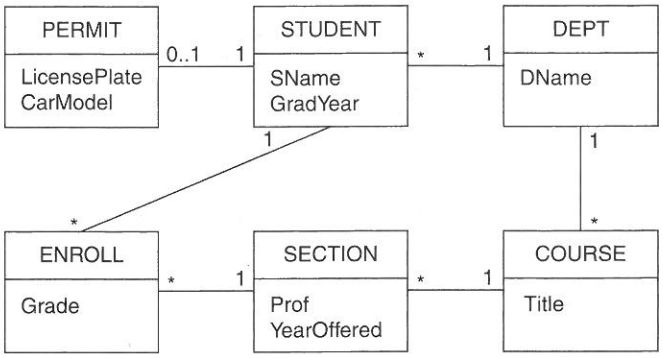


Figure 3-1
A class diagram for the university database

Each class will have zero or more attributes, which are written inside its box. An entity will have a value for each attribute of its class.

Figure 3-1 depicts the class diagram for our example university database, with the addition of a class PERMIT that holds information about parking permits. In this diagram, the class STUDENT participates in three relationships; the classes DEPT, ENROLL, SECTION, and COURSE participate in two relationships each; and the class PERMIT participates in one relationship.

Most concepts in a class diagram have a corresponding concept in the relational model. A class corresponds to a table, and its attributes correspond to the fields of the table. An entity corresponds to a record. A relationship is the one concept that does not have a direct analog in the relational model, although a foreign key comes close. These correspondences will form the core of the algorithm that translates class diagrams to relational schemas, as we shall see in Section 3.3.

3.2.2 Relationship Cardinality

Each side of a relationship has a *cardinality* annotation.

> The *cardinality* annotations indicate, for a given entity, how many other entities can be related to it.

Consider the relationship between STUDENT and DEPT in Figure 3-1. The "*" next to STUDENT means that a department can have any number of (including zero) student majors, and the "1" next to DEPT means that each STUDENT record must have exactly one associated major department. This relationship is called a *many-one* relationship, because a department can have many students, but a student can have only one department. Many-one relationships are the most common form of

relationship in a database. In fact, all but one of the relationships in Figure 3-1 are many-one.

Now suppose that we changed the relationship between STUDENT and DEPT so that the annotation next to DEPT was a "*" instead of a "1". This annotation would denote that a student could declare several (including zero) major departments, and each department could have several student majors. Such a relationship is called a *many-many* relationship.

The relationship between STUDENT and PERMIT is an example of a *one-one* relationship, meaning that each student can have one parking permit, and each permit is for a single student. The relationship is not exactly symmetrical, because a permit must have exactly one related student, whereas a student can have at most one (and possibly zero) related permits. We denote this latter condition by the "0..1" annotation next to PERMIT in the class diagram.

In general, UML allows annotations to be of the form "N..M," which denotes that an entity must have at least N and at most M related entities. The annotation "1" is a shorthand for "1..1", and "*" is a shorthand for "0..*". The annotations "1", "*", and "0..1" are the most common in practice, and are also the most important for database design. We shall consider them exclusively in this chapter.

3.2.3 Relationship Strength

Each side of a relationship has a *strength,* which is determined by its annotation. The "1" annotation is called a *strong annotation,* because it requires the existence of a related entity. Conversely, the "0..1" and "*" annotations are called *weak annotations,* because there is no such requirement. For example, in Figure 3.1 the relationship from STUDENT to DEPT is strong, because a STUDENT entity cannot exist without a related DEPT entity. On the other hand, the relationship from DEPT to STUDENT is weak, because a DEPT entity can exist without any related STUDENT entities.

We can classify a relationship according to the strength of its two sides. For example, all of the relationships in Figure 3-1 are *weak-strong,* because they have one weak side and one strong side. All many-many relationships are *weak-weak,* as are one-one relationships that are annotated by "0..1" on both sides. And a relationship that has "1" on both sides is called *strong-strong.*

3.2.4 Reading a Class Diagram

Because of its visual nature, a class diagram provides an overall picture of what data is available in the database. For example, by looking at Figure 3-1 we can see that *students* are *enrolled* in *sections* of *courses.* We can therefore assume that it should be possible to find, for example, all courses taken by a given student, all students taught by a given professor in a given year, and the average section size of each math course. We can also assume from the diagram that it will *not* be possible to determine which students used to be math majors (because only the current major is stored), or which professor is the advisor of a given student (because there is no advisor-advisee relationship).

The purpose of a class diagram is to express the design of the database as clearly as possible. Anything that the designer can do to improve the readability of the diagram is

a good thing. For example, sometimes the meaning of a relationship is obvious by looking at the diagram, and sometimes it is not. In Figure 3-1, the relationship between STUDENT and ENROLL is obvious, as it denotes a student enrollment. However, the relationship between STUDENT and DEPT is less obvious. In such cases, designers often annotate the relationship by writing a description (such as "major") on the relationship's line in the class diagram.

3.3 Transforming Class Diagrams to Tables

3.3.1 Relationships and Foreign Keys

A class diagram consists of classes, attributes, and relationships. Classes correspond to tables, and attributes correspond to fields. But what do relationships correspond to? The insight is this:

> A foreign key corresponds to a weak-strong relationship.

For example, consider again the schema of the CD database that appeared at the beginning of this chapter:

```
CD(CDId, Title, ReleaseYear, Band, Genre)
TRACK(TrackId, CDId, Track#, SongTitle, Duration)
LYRICS(TrackId, Lyrics, Author)
```

Here, we have added a third table, LYRICS, which holds the lyrics and author of each track that has lyrics. We use the field *TrackId* as the key of LYRICS, because each track can have at most one set of lyrics.

The field *CDId* in TRACK is a foreign key of CD, and the field *TrackId* in LYRICS is a foreign key of TRACK. The referential integrity constraints associated with these foreign keys assert that every TRACK record must have exactly one corresponding CD record, and every LYRICS record must have exactly one corresponding TRACK record. However, referential integrity says nothing about the other side of those relationships—a CD record can have any number of TRACK records that refer to it (even zero), and a TRACK record can have any number of LYRICS records.

In other words, the foreign key expresses a many-one relationship between CD and TRACK, and a many-one relationship between TRACK and LYRICS. (We can infer that the relationship between TRACK and LYRICS is actually one-one because the foreign key field is also the key.) We can depict this relational schema as the class diagram of Figure 3-2.

Note that the attributes of these classes are not identical to the fields of their corresponding tables. In particular, the key and foreign key fields are missing. In general, a class diagram should not have such fields. Foreign keys are not needed because their sole purpose is to implement a relationship, and relationships are already expressed directly in the diagram. And keys are not necessary in a class diagram because there are no foreign keys to refer to them.

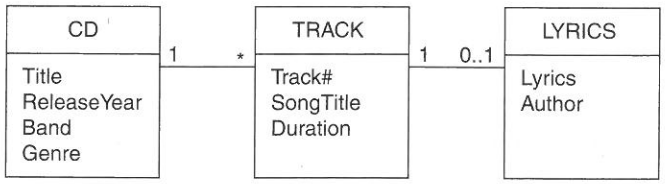


Figure 3-2
A class diagram for the example CD database

> A class should not have attributes whose sole purpose is to form a key or be a foreign key.

It is perfectly legitimate (and not uncommon) for a class to have no attributes.

3.3.2 Transforming a Relational Schema

We can transform a relational schema to a class diagram by making use of the correspondence between foreign keys and relationships. The algorithm is straightforward and appears in Figure 3-3.

We just saw one example of this algorithm, where we transformed the 3-table CD schema to the class diagram of Figure 3-2. For another example, consider the relational schema for the university database given in Figure 2-1, with the foreign key specifications of Figure 2-2. Applying this algorithm results in the class diagram of Figure 3-1 (minus the class PERMIT).

Although this algorithm is simple and straightforward, it turns out to be quite useful. The reason is that class diagrams tend to be easier to understand than relational schemas. For example, compare the relational schema of Figure 2-1 to the class diagram

1. Create a class for each table in the schema.
2. If table T_1 contains a foreign key of table T_2, then create a relationship between classes T_1 and T_2. The annotation next to T_2 will be "1". The annotation next to T_1 will be "0..1" if the foreign key is also a key; otherwise, the annotation will be "*".
3. Add attributes to each class. The attributes should include all fields of its table, except for the foreign keys and any artificial key fields.

Figure 3-3
The algorithm to transform a relational schema to a class diagram

of Figure 3-1. The class diagram represents relationships visually, whereas the tables represent relationships via foreign keys. Most users find it easier to understand a picture than a linear list of table schemas.

Class diagrams also tend to be easier to revise than relational schemas. People have discovered that the best way to revise a relational schema is to transform its tables to a class diagram, revise the diagram, and then transform the diagram back to tables.

Knowing the class diagram for a relational schema also makes it easier to write queries. As we shall see in Chapter 4, often the hard part of writing a query is determining which tables are involved and how they get joined. The visual nature of a class diagram and its explicit relationships helps tremendously. In fact, if you showed the relational schema of Figure 2-1 to an experienced database user, that user would very likely translate these tables back into a class diagram, just to be able to picture how the tables are connected. And this is but a simple example. In a more realistic database, class diagrams become indispensable.

3.3.3 Transforming a Class Diagram

Figure 3-4 presents an algorithm to transform a class diagram into a relational schema. This algorithm works by creating a foreign key in the schema for each relationship in the diagram.

For an example of this transformation, consider the class diagram of Figure 3-1. Suppose for simplicity that we choose to use an artificial key for each table. Then the resulting database would have the schema of Figure 3-5(a) (where primary keys are underlined and foreign keys are in italics).

A variation of this algorithm is to intersperse the choice of primary keys and foreign keys, so that a foreign key field for a table could also be part of its primary key. The schema of Figure 3-5(b) uses this variation to generate as many non-artificial keys as possible. For example, the foreign key fields {*StudentId*, *SectionId*} are used as the primary key for ENROLL. This compound key is a good choice here because ENROLL does not have other tables referencing it, and so a corresponding multi-field foreign key will not get generated. In addition, the foreign key *StudentId* in PERMIT can be used as its primary key, because of the one-one relationship to STUDENT.

The algorithm of Figure 3-4 is only able to transform weak-strong relationships. The designer therefore has the responsibility to ensure that the class diagram has no weak-weak or strong-strong relationships. This issue will be addressed in Section 3.4.5.

1. Create a table for each class, whose fields are the attributes of that class.
2. Choose a primary key for each table. If there is no natural key, then add a field to the table to serve as an artificial key.
3. For each weak-strong relationship, add a foreign key field to its weak-side table to correspond to the key of the strong-side table.

Figure 3-4
The algorithm to transform a class diagram to a relational schema

```
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