Let's recap the story so far, in our endeavors to design a database. We started with use cases to describe the basic requirements of a problem and developed an initial data model. By looking carefully at the details of the model, we were able to develop questions to help understand further subtleties and complexities of the real-world problem. We then looked at a number of situations that occur in many models in the hope that these would be useful when difficult situations arose in other contexts.

The goal is not to attempt to get a perfect or complete model. The outcome we are seeking is agreement on a model that accurately reflects the essential requirements of the real-world problem. This will involve numerous iterations as the use cases adjust to reflect the improved understanding and the changing scope. Having arrived at a set of use cases and a data model with which everyone is comfortable, we can now move on to the third phase of the development process, as shown in the bottom-right square of Figure 7-1.

In this and the following chapters, we will look at how to design a database that could be implemented in a relational database product (e.g., MySQL, Microsoft Access, SQL Server, Oracle, etc.).

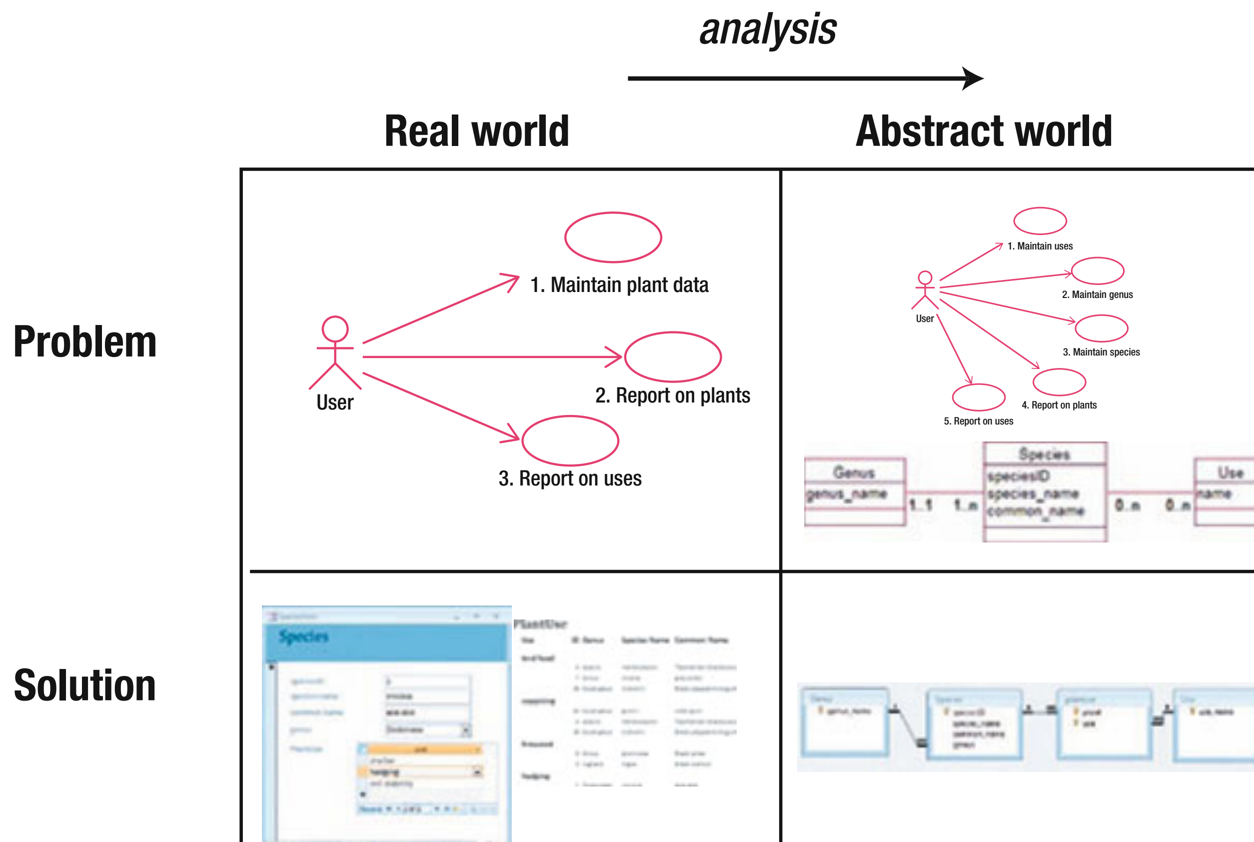


Figure 7-1. *Database development process*

Representing the Model

We have gone to a great deal of trouble to capture as much detail as possible in the data model. Much of this detail can be represented and enforced by standard techniques built into relational database management software. A good model, implemented using the standard techniques, allows us to capture many of the constraints implied by the relationships between classes without recourse to programming or complex interface design.

In this chapter, I will show you how many of the aspects of the data model can be captured by standard database functionality. To give you an idea of what is coming up, I have summarized the techniques in Table 7-1.

Table 7-1. *Techniques to Represent Aspects of the Data Model*

Feature in Model	Technique Used in Relational Database
Class	Add a table with a primary key.
Attribute	Add a field with an appropriate data type to the table.
Object	Add a row of data to the table.
1–Many relationship	Use a foreign key, i.e., a reference to a particular row (or object) in the table at the 1 end of the relationship.
Many–Many relationship	Add a new table with two foreign keys.
Optionality of 1 at the 1 end of a relationship	Make the value of the foreign key required.
Parent and child classes (inheritance)	Add a table for the parent class. Add tables for each child class with a primary key that is also a foreign key referencing the parent table (not an exact representation but OK).

All of the techniques described in Table 7-1 can be carried out in most database management products as part of the specification of the tables. More complex constraints may require some additional procedures or checking at data input time, but with a good model this can be minimized. By using the built–in facilities of the database product, the time required for implementation, maintenance, and expansion of the application is greatly reduced.

Representing Classes and Attributes

Consider Figure 7-2, which represents a small part of a data model for members and teams.

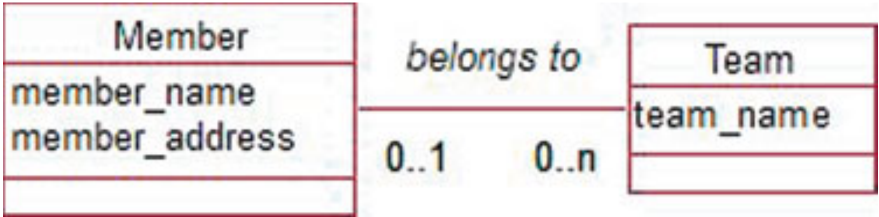


Figure 7-2. *Part of data model for members and teams*

The first step is to design a database table for each class. The attributes of the class will become the field or column names of the table, and when the data is added, each row, or record, in the table will represent an object For example, as a start we would create a table called `Member` for the `Member` class in Figure 7-2. The table would have two fields or columns, one for a member's name and one for the address. We would then add a row to the table for each member object (e.g., John Smith, 83 SomePlace, Christchurch).

Creating a Table

All relational databases allow you to create a table using SQL, which is a language for creating, updating, and querying databases. First we need a database that stores all our tables. To create a table, we need to provide a name for the table and a name and domain for each of the columns. The domain specifies the set of values that is allowed for that particular column. We will talk a little more about domains later in this chapter. For many purposes it is sufficient to just specify a data type, for example a date (8/4/12), a piece of text or a set of characters ("Mary Smith"), an integer (467), or some other type of number (3.57). Particular database products provide different data types, and we will talk about suitable choices a little later on in the section "Choosing Data Types."

In addition to providing SQL as a way of creating a table, many databases provide a more graphical front end through which the user can provide information about the columns and their data types. Equivalent ways of creating a very simple `Member` table are shown in Listing 7-1 and Figures 7-3 and 7-4. Figure 7-3 shows the MySQLWorkbench front end, while Figure 7-4 shows the equivalent in MS Access. Both programs will generate SQL statements similar to those shown in Listing 7-1. Note that the data type for the two fields is called *Text* in Access and *VarChar* in SQL. These both just mean that the user will be able to enter any number of characters up to the maximum number stated (25 for name and 40 for address).

Listing 7-1. *Standard SQL Command to Create a Customer Table with Two Fields*

```
CREATE TABLE Member (
member_name VARCHAR(25),
member_address VARCHAR(40)
)
```

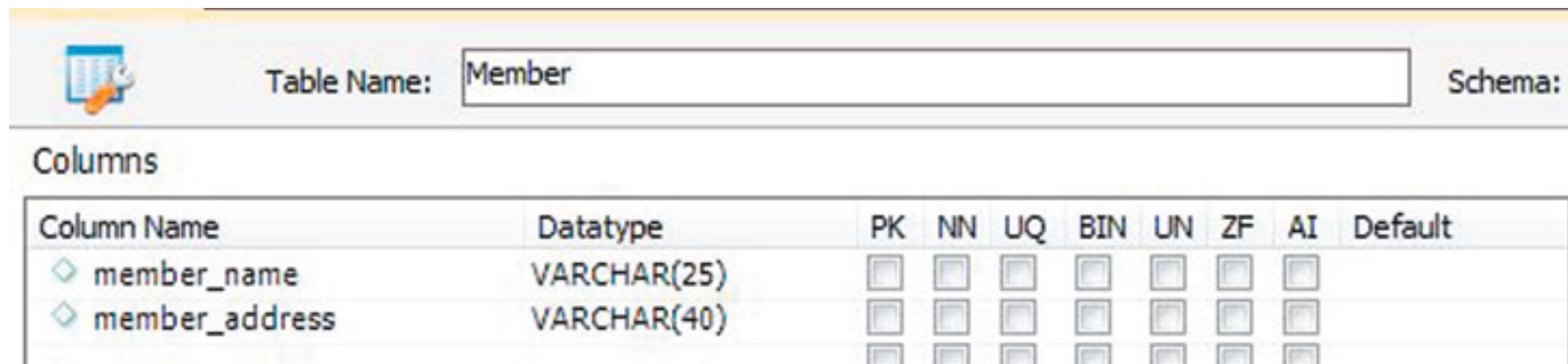


Figure 7-3. *Creating a Member table in MySQL Workbench*

Member

Field Name	Data Type
member_name	Text
member_address	Text

Text
Memo
Number
Date/Time
Currency
AutoNumber
Yes/No
OLE Object
Hyperlink
Lookup Wizard...

General | Lookup

Field Size	40
Format	
Input Mask	
Caption	
Default Value	
Validation Rule	
Validation Text	
Required	No
Allow Zero Length	Yes
Indexed	No
Unicode Compression	Yes
IME Mode	No Control
IME Sentence Mode	None
Smart Tags	

7-4. *Creating a Member table in Microsoft Access*

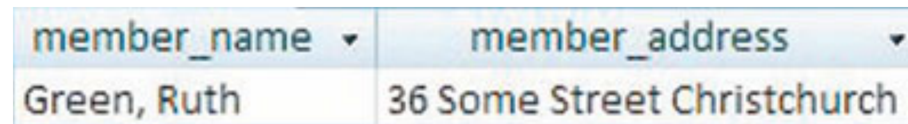
Indexed	No
Unicode Compression	Yes
IME Mode	No Control
IME Sentence Mode	None
Smart Tags	

Figure 7-4. *Creating a Member table in Microsoft Access*

As you can see in Figures 7-3 and 7-4, there are many additional possible specifications for columns. We will come to some of those later. For now we have just declared the minimum requirements of a name and data type for each of the columns. Once the table has been created, we can enter data. In the case of the `Member` table, we would enter a row for each member object. Once again, this can be done with SQL commands as in Listing 7-2 or for many products through a table-like front end as shown for Access in Figure 7-5.

Listing 7-2. *SQL Command for Entering a Record into the Member Table*

```
INSERT INTO Member (name, address)
VALUES (‘Green, Ruth’ ‘36 Some Street, Christchurch’ )
```



member_name	member_address
Green, Ruth	36 Some Street Christchurch

Figure 7-5. *Entering data through the MS Access interface*

Choosing Data Types

Each attribute in a class becomes a *field* or a column in a table. When we create the table, we need to provide a name for the field (e.g., `name`, `address`) and specify the type of data that will be stored in that field. Database products often offer a bewildering number of different data types, but they basically fall into the following groups:

Character types: These allow you to enter any combination of characters–numbers, letters, and punctuation. They are used for names, addresses, descriptions, and so on. You usually need to provide a maximum length for the data going into the field. In SQL, a type of VARCHAR(60) would allow you to enter any number of characters up to 60. In Access, the equivalent type is called Text. If you have very large amounts of text (notes, discussions, and so on), you might like to look at other types (e.g., Text in SQL Server or Memo in Access).

Integer types: These types are for entering numbers with no fractional part. They are great for ID numbers such as customer numbers and for anything that you can count. Database systems often provide different–sized integer types (long, short, byte, etc.) that have different maximum numbers that can be entered. Unless you have particular performance problems or extremely large amounts of data, you will probably be fine if you use the ordinary integer type (INT in SQL). Just check that the biggest number it can handle is large enough for your data.

Numbers with a fractional part: These are used for things that you measure (heights, weights, etc.) and also for numbers that result from calculations such as averages. Most of the time you will be just fine with what is called a *float* or *single* (depending on the product you are using). Other types exist if you need particularly accurate measurements or calculations. One situation when a float may not be suitable is when you need to record rather large amounts of money accurately. Many products now provide *money* or *currency* types for this situation, or you may find the type is called something like *fixed–length decimal*. These types enable you to have many significant figures so that you can accurately keep track of your billions, down to a fraction of a cent!

Dates: No prizes for guessing the type of data you can put in fields with these types. If your product has different date types, some may allow you to include times and others may allow you to access dates further into the past or future.

Why is it important to get the correct data type? You could argue that since you can put anything in a character field, you can have character fields for everything (and I've seen it done!). There are three main re why it is important to choose an appropriate data type for each of your fields:

Constraints on the values: A character field type has no constraints on what you can enter; however, most other fields do. Number fields won't allow you to accidentally mistype a number, say, by putting in an extra decimal point or a letter "O" instead of the number 0. Dates won't allow February 29 unless it is in a leap year, and so on. For this reason, phone numbers, which are likely to have extra symbols like ( ) for area codes, need to be stored in character rather than number fields.

Ordering: Different types of fields have different ways for comparing or ordering values. For example, character fields can be sorted alphabetically (A to Z), number fields numerically (small to large), and dates chronologically (older first). If you store numbers in character fields and then ask your product to sort them for you, you might get something like this: 10, 12, 123, 2, 200, 36. Dates in a character field might be sorted like this: August 1, 2012; February 1, 2012; May 4, 2012. Can you see why?

Calculations: Your database product can do arithmetic and perform other functions on your data, but only if it is the correct type. For example, it will be able to add, multiply, and average numbers; figure out how many days fall between two dates; and look for particular characters in a piece of text. You need to have the correct types in order to take advantage of this functionality. And getting back to phone numbers, you never want to subtract them, average them, or order them numerically, so they can and should generally be stored in a character field type.

Domains and Constraints

A domain is a set of values allowed for an attribute or field. For something like a product description it might be sufficient for the domain to be any set of characters up to some specified length. Other attributes might have more specific domains. For example, a gender attribute might only allow the values “M” or “F”; a day of the week might be constrained to the characters “Mon,” “Tue,” “Wed,” “Thu,” “Fri,” “Sat,” and “Sun”; possible marks for an exam might be whole numbers between 0 and 100.

Some database products (e.g., SQL Server) allow users to create their own domains or data types. For example, you might define the type ExamMark as an integer between 0 and 100. This user defined domain or type can then be used in all the tables in the database. Other products (e.g., Access) do not permit the creation of domains, but all products allow constraints to be declared on individual columns. For instance, we could declare gender as being a character type of length 1 with the constraint that it can only accept the values “M” or “F.” The difference between a constraint and a domain is that the former has to be specified in every table whereas the latter only needs to be declared once in the database.

The SQL code for creating a table with a constraint on the values for a gender field is shown in Listing 7-3.

Listing 7-3. *SQL for Creating a Table with a Constraint*

```
CREATE TABLE Member (
member_name VARCHAR(25),
member_address VARCHAR(40),
gender VARCHAR(1) CHECK gender IN (‘M’, ‘F’)
)
```

One very important constraint is specifying whether a value is required or can be left empty. A field with nothing in it is said to be *null,* and when a table is created we can specify which fields are not allowed to have nulls. This is shown in SQL in Listing 7-4.

Listing 7-4. *SQL for Specifying That the Name Field Must Have Values*

```
CREATE TABLE Member (
member_name VARCHAR(20) NOT NULL,
member_address VARCHAR(45),
gender VARCHAR(1) CHECK gender IN (‘M’, ‘F’)
)
```

Looking at the code in Listing 7-4, it is reasonable to ask why we haven't insisted gender must always have a value as well. All members have a gender, after all. In general, there are two main reasons why we might need to put a null in a field: either the field doesn't apply for a particular record (a person may or may not have a driver's license number) or the field does apply, but at the moment we don't know the actual value. For the situation with gender then, clearly the value applies, but there could be situations where we do not know what it is. If we force a value to always be entered, we risk not being able to enter the record or having a distressed data entry operator taking a guess at a likely value.

Consider a university administrator entering details from a stack of student applications, a couple of which have left the box for gender empty. The university would much rather have the student's information entered incompletely than not at all. At least then they can extract some fees and contact the person about the gender at a later time. What about name–should that be allowed to be null? It is always a judgment call, but personally I think recording details about a nameless student is probably going to result in trouble somewhere down the line.

Even if you think a value is going to be essential for the accuracy of your data, do not underestimate the likelihood that disallowing nulls might cause an incorrect value to be entered. I find myself doing this all the time when filling in web forms for US sites that demand a value for a state. I live in New Zealand. We don't have states, so I just make something up. Some sites accept "XXX," while others demand a real US state; in those cases I use "Virginia." I don't know why. What I do know is that this situation drives me crazy, and that any statistics being gathered about the states of site visitors are going to be hopelessly inaccurate.

Checking Character Fields

Character fields are a bit different from other field types. With a character field, we can enter anything we like (if there are no other constraints), and so it is possible to enter several values into a field. Other fields such as numbers and dates only ever allow one value to be entered.

You have seen examples of storing several values in one character field in Figure 7-5. The `member_name` field as it stands could contain data about the first name, last name, possibly other names, initials, and titles. To find the record shown in Figure 7-5 is going to be very difficult. Will the user know whether to search for Mrs. Green, Mrs. Rose Green, Rose Green, Mrs. R. Green, or Ms. Green? It is also going to be difficult to sort the records sensibly. We usually want to order people by last names, and this is not going to be possible with the way we are recording the data in Figure 7-5. The way the address data is being recorded is going to make it difficult to select records by city or print nicely formatted address labels easily. Separating the data into fields as in Figure 7-6 makes the data much more useful. A good rule of thumb is that any data that you are likely to want to search for, sort by, or extract in some way should be in a field all by itself.

last_name	first_name	title	street_address	city	post_code
Turner	John	Mr	Flat 1, 6 Moa Street	Christchurch	8033
Green	Ruth	Mrs	36 Some Street	Christchurch	8041

Figure 7-6. *Improved fields to describe a customer*

If the accuracy of values in a field is really crucial to the project, maybe that particular piece of information should actually be in a class of its own. You might recall that we separated `genus` out of our `Plant` class (refer to the discussion of Figure 2-12) because its accuracy was important and we didn't want any misspellings. In the table in Figure 7-6 we might ask how important it is for the values in the `city` field to be accurately recorded. If accuracy is essential (e.g., we regularly want to target advertising to customers in a particular city), we may need two classes, `City` and `Customer`, with a 1–Many relationship between them. If we only want the address for sending general mail, accuracy isn't so important, as the postman can probably cope with the odd misspelling.

Primary Key

We have taken our model and for each class we have created a table. Each attribute in the class is represented by a field with a particular data type, and we can apply some constraints to the values we allow into a field. We have thus far overlooked one constraint that is so important that it gets a section all to itself. This involves choosing a *primary key* for the table. It is imperative that we can always find a particular object (or row or record in a table). This means that all records must be unique; otherwise, you can't distinguish between two that are identical.

Consider the consequences of two identical records: when a member pays his annual fee, we need to connect that payment to the member somehow. What if we have two identical rows in our `Member` table for Ruth Smith? How will we know which row is associated with a payment of fees? If we get it wrong, then one Mrs. Smith will be pretty upset when she receives another invoice. Every member needs to be able to be uniquely identified. There must never be two identical records in any of our tables.

Determining a Primary Key

A key is a field, or combination of fields, that is guaranteed to have a unique value for every record in the table. It is possible to learn quite a bit about a problem by considering which fields are likely to be possible keys. We will see later that there can be more than one set of fields that can have unique values in a table. We choose one of these to be the primary key and then use that to enable us to identify records uniquely.

Consider which fields could be keys for the following table where the names of the fields are given in parentheses after the table name:

```
Member (name, address, phone, birth_date)
```

How about name for a key? No; it is entirely possible that we may have two members with the same name, and we will need to be able to distinguish them. What about the combination (name, address)? This is more promising, but then dads have been known to name their sons after themselves, and it is not improbable that they may at various times of their lives share the same address and be members of the same club. Many organizations sometimes key their clients on the combination (name, birth_date), feeling this is unlikely to be duplicated. However, there are regular horror stories in the press of people who suddenly discover they have a namesake twin as they struggle to fend off bailiffs and police.

A potential key must be guaranteed to be unique for every possible record. In cases like our Member table, there is not much choice but to add a new attribute or field such as member_number and then assign all members their own unique numbers so we can distinguish them. This is sometimes called a *surrogate key*. In real life, we can always distinguish individuals, but when we look at the data we are storing about them, we may not be able to find a unique set of values. In many cases, privacy laws prevent information such as social security or tax numbers being used to identify people, so each business or organization is often compelled to provide its own personal identification number.

When we create a table in our database, we can specify the field which is to be the primary key of the table. The SQL to do this is shown in Listing 7-5. Most database products also usually provide you with an interface to help create a table and select the field(s) that make up the primary key.

Listing 7-5. *Specifying a Primary Key*

```
CREATE TABLE Member (
member_number INT PRIMARY KEY,
member_name VARCHAR (25)
)
```

With the primary key field specified, a constraint is put on the table that will ensure that every record must have a unique value for member_number. The user will never be able to put in two records with the same value for

member_number, and so every member in our table can be uniquely distinguished. The constraint also ensures that the primary key field always has a value, so every record is certain to have a value for member_number.

It is possible to get the database to automatically generate unique values for fields like member_number. Depending on the product you are using, you will find a field type called identity, auto_increment, autonumber, or something similar. You can then specify some starting number and a step size, and every new row entered into the table will automatically be assigned the next available number.

Concatenated Keys

It isn't always necessary or even advisable to introduce a new automatically incrementing number field into a table to act as a primary key. With the case of members, there was no other way to ensure a unique field for every record, but often a unique field or combination of fields already exists in the table. When we have a combination of fields that can uniquely identify a record, this is referred to as a *concatenated* or *composite* key.Thinking about which combinations of fields are possible keys can help you discover and understand subtleties of the problem. Here is an example.

What is a possible key for the table in Figure 7-7, which is keeping information about students enrolling in courses?

student	course	year	grade
13887	COMP101	2011	B
17625	COMP101	2011	E
17625	COMP102	2012	A
18574	COMP102	2012	B

Figure 7-7. *Enrollments table*

17625	COMP102	2012	A
18574	COMP102	2012	B

Figure 7-7. *Enrollments table*

`student` will not be suitable as a key, as a student will have a record for each of the courses in which he enrolls (we can see that the value 17625 appears in at least two records). Similarly, `course` will not do, as a course will have many enrollments each with its own record (the value COMP102 is duplicated).In fact, every column has duplicated values, so no single field is suitable as a key.

What about the combination (`student`, `course`)? In the few records shown in Figure 7-7, this combination is always unique, but we have to be sure this will *always* be the case for every record we may need to enter. We need to find out a bit more about the problem. Consider this dialog:

> *Analyst:* Could student 17625 enroll in COMP101 a second time to try and improve his grade?
>
> *Client:* Yes.

Now we see that `student` and `course` will *not* be a suitable key. As soon as the student tries to enroll in the course again, we will have another row with the same values for student number and course.

Let's try the combination (`studentID`, `course`, `year`):

> *Analyst:* Is it possible for a student to enroll in the same course again in the same year (say during the summer)?
>
> *Client:* It is for some subjects.
>
> *Analyst:* If a student did reenroll in the same subject in the summer, would you want to keep both her previous and her new grade?
>
> *Client:* Of course!

The combination (`studentID`, `course`, `year`) will not do as a key either because we will have to repeat the values of the three fields when a student enrolls in the same course later in the same year. Clearly, we need an additional attribute (`semester` maybe) to differentiate these enrollments. Thinking about a possible key has revealed a little more of the complexity of this problem and helped us spot a missing attribute or field.

Whenever we are checking the suitability of a combination of fields as a key, we need to find a question that checks that the combination will always be unique. In this case, we needed to ask questions such as the following:

> *Is a student ever likely to enroll in a course more than once?*
>
> *Yes. (*`student`*,* `course`*) is not a suitable key.*
>
> *Is it possible that a student will enroll in the same course more than once in a single year?*
>
> *Yes. (*`student`*,* `course`*,* `year`*) is not a suitable key.*
>
> *Is it possible that a student will enroll in the same course more than once in a single semester of a given year?*
>
> *No. (*`student`*,* `course`*,* `year`*,* `semester`*) is a possible key.*

Now look at the other fields in the table:

> *Is it possible that a student might need to have more than one grade for a given enrollment (e.g., an initial grade and a revised grade)?*
>
> *Maybe. (In that case, the problem is much more complicated than we thought.)*

Would it all have been easier if we had just abandoned looking for a concatenated key and added an automatically generated `enrollment_number` field that could be guaranteed to be always unique?

Enrollment (`enrollment_number`, `student`, `course`, `semester`, `year`, `grade`)

Consider the case where a student can only enroll in a course once a semester. The enrollment table now has two possible keys: `enrollment_number` and the combination (`studentID`, `course`, `semester`, `year`). What are the pros and cons of choosing one key over the other?

`enrollment_number` is shorter than the concatenated key, and you will see in Chapter 9 that this may be a consideration. However, if we make the combination of fields (`student`, `course`, `semester`, `year`) the key, the database will ensure that we never enter duplicate values (i.e., it will impose the constraint that a student cannot enroll in the course more than once a semester). This will effectively ensure enrollments do not get entered twice accidentally. If we choose the `enrollment_number` as the key, we will need to find another way to prevent such duplications.

An automatically generated number may be a sensible key, but it should not be included in a table because we can't be bothered thinking about alternatives. If we don't think about what values are suitable keys, we may miss discovering some subtleties of the problem.

Representing Relationships

So far we have represented each class as a table, each attribute as a field with a particular type, and decided on a field, or combination of fields with unique values, to be a primary key. We can now use this primary key to help us represent relationships between the classes in our model.

Let's consider our sports club example. A simple data model is shown in Figure 7-8 with some possible objects in Figure 7-9. A member may have *one* team that he currently plays for, and each team has exactly *one* captain.

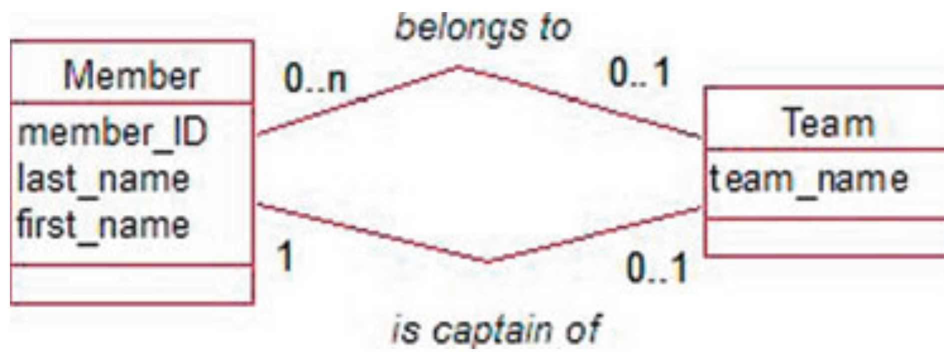


Figure 7-8. *Sports club data model*

105
Smith
Jim

118
Jones
Bob

198
Green
Linda

235
Brown
Grant

216
Brown
Sue

199
Norton
Harvey

265
King
Jane

SeniorB

JuniorA

is captain of

plays for

Figure 7-9. *Members, teams, and instances of the relationships between them*

Figure 7-9. Members, teams, and instances of the relationships between them

First we design two tables to represent the classes and choose a primary key for each. We will adopt the convention of underlining the primary key fields.

- `Member`: (<u>`member_ID`</u>, `last_name`, `first_name`)
- `Team`: (<u>`team_name`</u>)

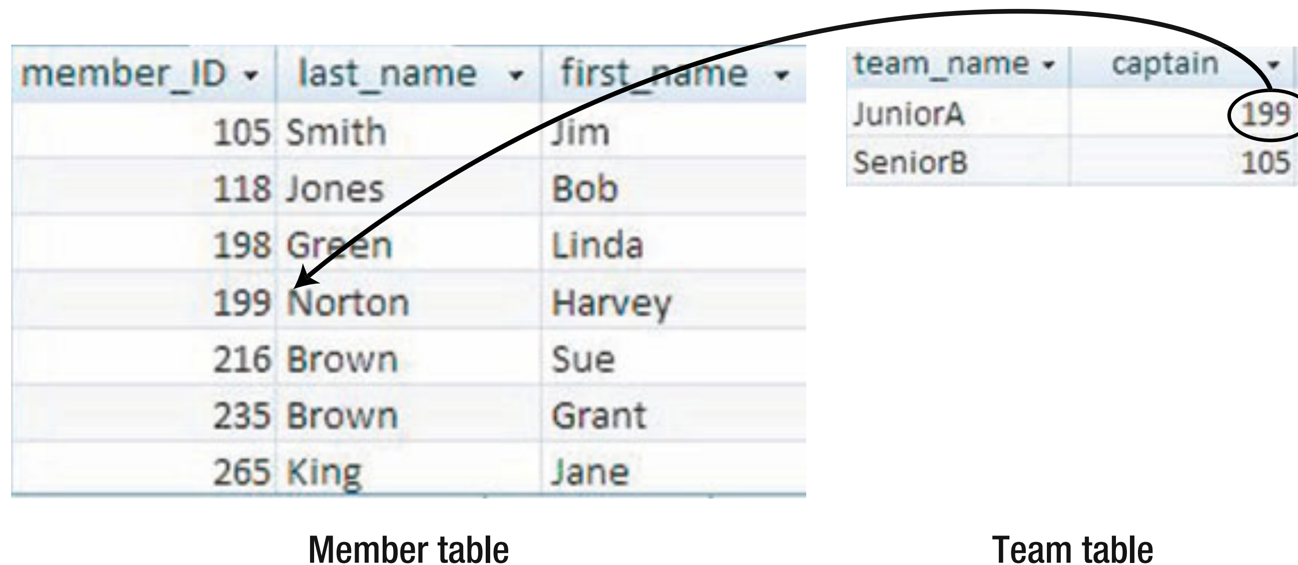
Each of the objects in Figure 7-9 will be a row in the appropriate table.

To represent the relationships *plays for* and *is captain of*, we need a way of specifying each of the lines between the objects in Figure 7-9. For example, we need to show that Bob Jones plays for SeniorB, and the captain of JuniorA is Harvey Norton.

As we have primary keys established, we can easily identify the row associated with each object (e.g., Harvey Norton is the row in the `Member` table where the primary key field `member_ID` has the value 199). To represent the relationship between the objects, we use these key values by way of a *foreign key* as described in the next section.

Foreign Keys

Figure 7-10 shows the two tables `Member` and `Team` again, but now we have added a field to show who is the captain of each team. What we have done is put a new field in the `Team` table (`captain`) that will contain the key value of the member who is its captain. This is a foreign key. A foreign key is a field(s) (in this case `captain`) that refers to the primary key field(s) in some other table (in this case it contains a value of the key field `member_ID` from the table `Member`). In this way, we establish the relationships between objects of different classes.



member_ID	last_name	first_name
105	Smith	Jim
118	Jones	Bob
198	Green	Linda
199	Norton	Harvey
216	Brown	Sue
235	Brown	Grant
265	King	Jane

Member table

team_name	captain
JuniorA	199
SeniorB	105

Team table

Figure 7-10. *The Team table has a foreign key field (captain) referring to the Member table.*

The SQL statement for creating the table `Team` with a foreign key referring to the `Member` table is shown in Listing 7-6. Many products also provide a diagrammatic interface for specifying foreign keys. The interface for setting up a foreign key in Access is shown in Figure 7-11.

Listing 7-6. *SQL to Create a Team Table with a Foreign Key*

```
CREATE TABLE Team (
team_name VARCHAR(10) PRIMARY KEY,
captain INT FOREIGN KEY REFERENCES Member
)
```

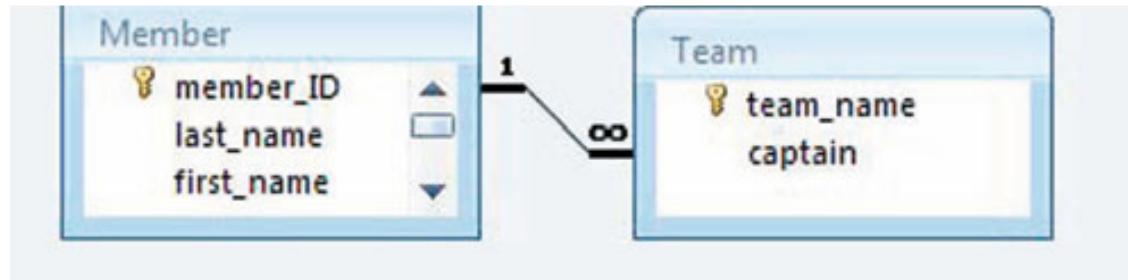


Figure 7-11. *Access interface for specifying captain is a foreign key referring to the Member table*

The two fields member_ID and captain will both have values from the same domain, that is, the set of MemberIDs. Formally, a foreign key and the primary key of the table it references must have the same domain. In most database products this is softened to requiring them to be the same data type or a compatible data type. The types of data that are regarded as compatible will depend on the database software being used (e.g., character fields of different lengths are compatible in some products, but not in others).

Referential Integrity

Arm-in-arm with the idea of a foreign key is the concept of *referential integrity*. This is a constraint that says that each *value* in a foreign key field (i.e., 199 and 105 in the Team table in Figure 7-10) must exist as values in the primary key field of the table being referred to (i.e., 199 and 105 must exist as values in the member_ID field in Member). This prevents us putting a nonexistent member (say, 765) as the captain of a team. It also means that we cannot remove members 199 and 105 from our member table while they are captains of teams. As soon as you set up a foreign key, this referential integrity constraint is automatically taken care of for you.

Representing 1–Many Relationships

In the previous sections, you have seen how it is possible to represent instances of a relationship in the data model by using a foreign key. In general, the process for a 1–Many relationship is as follows:

> *For a 1–Many relationship, the key field from the table representing the class at the 1 end is added as a foreign key in the table representing the class at the Many end.*

We have already represented the relationship *is captain of* in Figure 7-8. Let's now use our general guideline to do the same thing for the relationship *plays for* between `Member` and `Team`.

The class at the 1 end is `Team`, so we take the primary key field from the `Team` table and add it as a new foreign key attribute in the `Member` table. We can give the field any name we like, but it should clearly indicate the relationship it is representing, for example `current_team`. This is shown in Figure 7-12.

member_ID	last_name	first_name	current_team
118	Jones	Bob	SeniorB
198	Green	Linda	JuniorA
199	Norton	Harvey	JuniorA
216	Brown	Sue	SeniorB
235	Brown	Grant	
265	King	Jane	JuniorA

Member table

current_team is a foreign key referencing Team

team_name	captain
JuniorA	199
SeniorB	105

Team table

captain is a foreign key referencing Member

Figure 7-12. *Both relationships in sports club model represented by foreign keys*

Figure 7-12. *Both relationships in sports club model represented by foreign keys*

Referential integrity, which is a result of making the current_team field a foreign key, will ensure that the value entered in current_team can be found in the primary column (team_name) of the Team table. This ensures that members can only play for teams that already exist in the Team table.

Note that a foreign key field can be null. Grant Brown does not belong to a team, so there is no value in the foreign key field in his record. This is consistent with the optionality of the *plays for* relationship in the class diagram back in Figure 7-8. If the relationship was not optional, we would have to impose an additional constraint on the field to say that nulls were not permitted. If we wanted to ensure that every team had a captain (as the data model suggests), then as well as making the captain field in the Team table a foreign key, we would also specify that it cannot ever be null.

Let's look at another example of a 1–Many relationship—this time a self relationship. We will consider the case where a member sponsors other members who wish to obtain membership with the club. The relevant part of the data model is shown in Figure 7-13, and some objects and their relationships are shown in Figure 7-14.

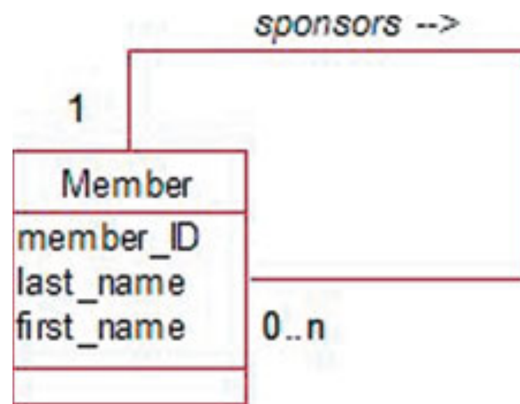


Figure 7-13. *Self relationship: Member sponsors other members.*

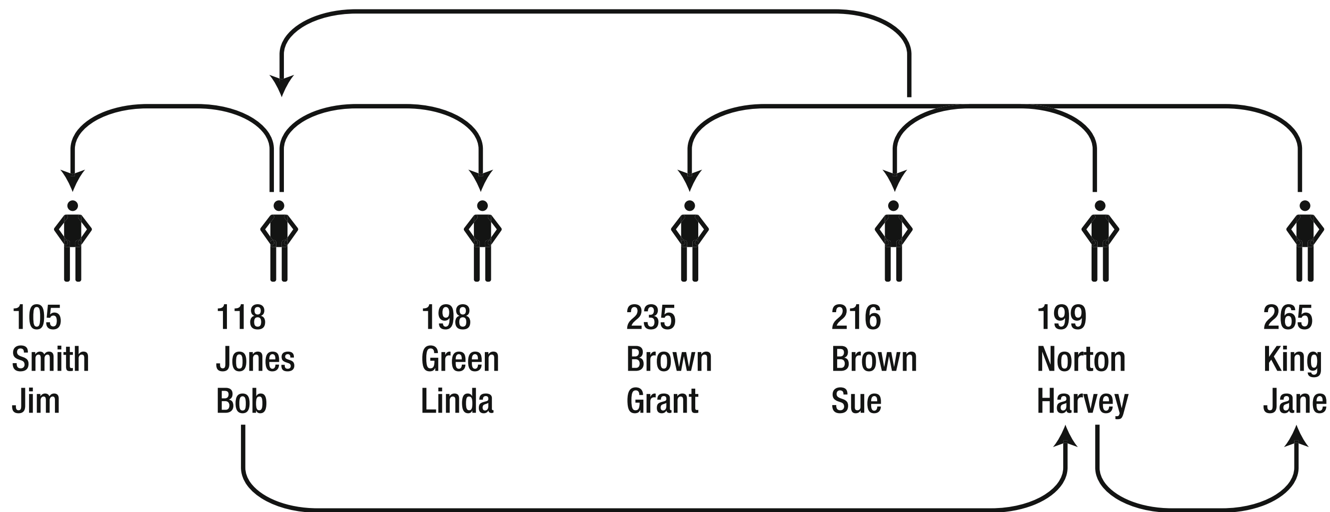


Figure 7-14. *Instances of members who sponsor each other*

This self relationship is a 1–Many relationship, and we do exactly the same as we do for any other 1–Many relationship. We take the key from the table representing the class at the 1 end (`member_ID`) and add it as a foreign key to the table representing the class at the Many end (`Member`). It makes no difference that it is the same table. We give the new foreign key field a name that describes the relationship, say *sponsor*, and the table will look like that in Figure 7-15.

member_ID	last_name	first_name	current_team	sponsor
105	Smith	Jim	SeniorB	118
118	Jones	Bob	SeniorB	216
198	Green	Linda	JuniorA	118
199	Norton	Harvey	JuniorA	118
216	Brown	Sue	SeniorB	265
235	Brown	Grant		199
265	King	Jane	JuniorA	199

Figure 7-15. *A foreign key (sponsor) representing a self relationship*

198	Green	Linda	JuniorA	118
199	Norton	Harvey	JuniorA	118
216	Brown	Sue	SeniorB	265
235	Brown	Grant		199
265	King	Jane	JuniorA	199

Figure 7-15. *A foreign key (sponsor) representing a self relationship*

The table `Member` has a foreign key, `sponsor`, referencing its own table. Jim Smith is sponsored by member 118, who is Bob Jones. Unlike a primary key, there is no restriction that a foreign key such as `sponsor` must be unique. In Figure 7-15 we can see that 118 (Bob Jones) is sponsoring several members. Referential integrity ensures that a member can only be sponsored by someone who is already a member. There is a bit of a problem if the relationship is compulsory, which means we add a constraint not to allow nulls in the sponsor field. How do you ever get the first member into the database when there is no existing member to sponsor her? This isn't just a database problem, it is actually part of our problem description. All new members need a sponsor, but what about the founding members? Making the sponsor field required is probably not a good idea.

Representing Many–Many Relationships

You may remember from Chapter 4 that Many–Many relationships are not as common as you might at first expect. Often they are a sign that some information about the problem has been initially overlooked, and an intermediate class is required to store that information. They do, however, genuinely occur where we have objects that simultaneously belong in many categories. Figures 7-16 and 7-17 review the plant database from Chapter 2 where we had species of plants that were suitable for a variety of uses.

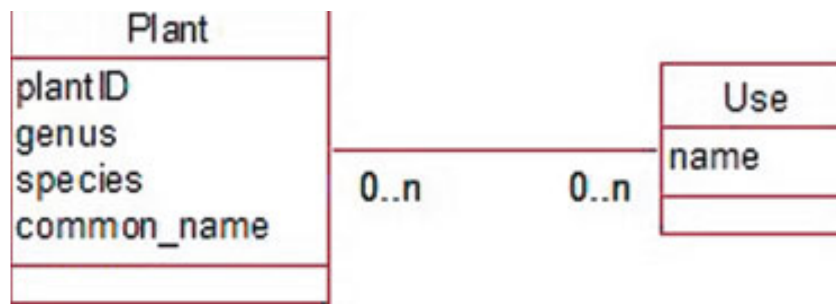


Figure 7-16. *Data model for plant database*

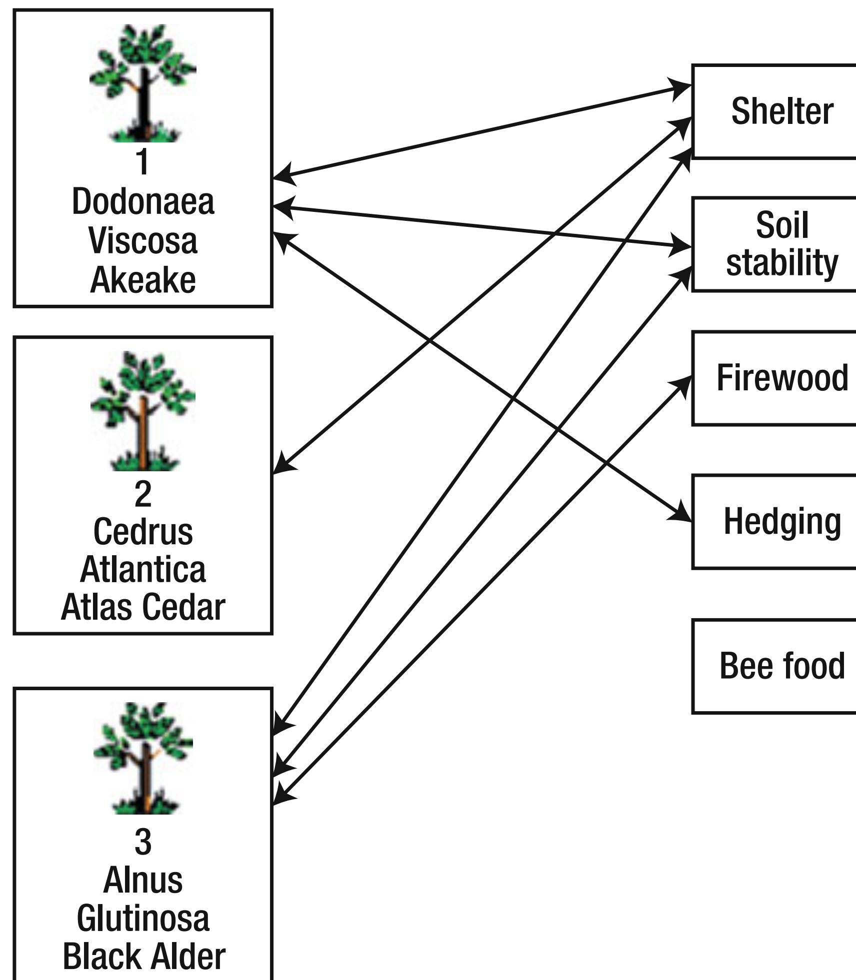


Figure 7-17. *Some examples of species and their uses*

Figure 7-17. Some examples of species and their uses

How are we to represent all the instances of this relationship? Foreign keys will no longer do the trick, as we will never know how many uses a particular species will have, nor how many species will be related to a particular use. To deal with this in a relational database, we have to introduce a new intermediate class in our data model. You saw how to do this in Chapter 4 when we had some additional information that required a new class. In this Many–Many situation, the new intermediate class will not have any attributes, as there is nothing we wish to know about a particular combination of `Species` and `Use`. We use the new class (`Species_Use`) simply to store all the relevant pairings of `Use` and `Species`. As in Chapter 4, the new class connects to the existing classes with two 1–Many relationships as shown in Figure 7-18. We can interpret the diagram as: "Each `Species_Use` object (or pairing) consists of exactly one species and exactly one use."

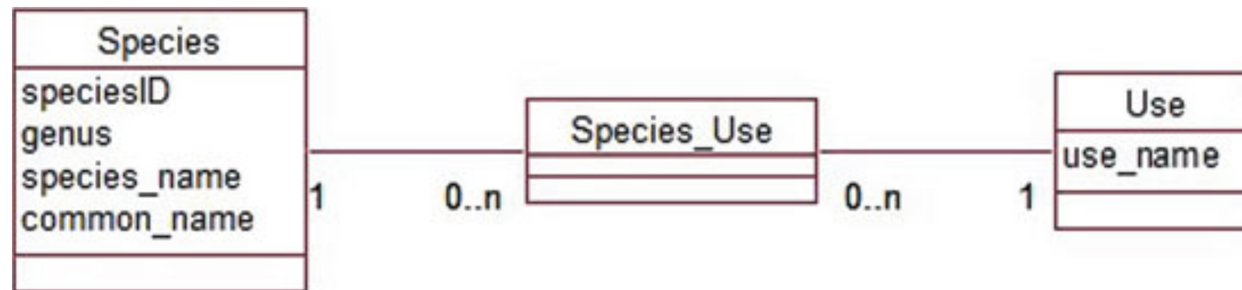


Figure 7-18. *Adding another class to represent a Many–Many relationship in a relational database*

The two 1–Many relationships can now be dealt with like any other 1–Many relationship. First we need to create a table, `Species_Use`, for our new class. Then for each of the 1–Many relationships, we add the primary key field from the table representing the class at the 1 end as a foreign key in the table representing the class at the Many end. This means adding two new foreign key attributes, `species` and `use`, to the `Species_Use` table. These foreign keys will reference the `Species` and `Use` tables, respectively. The resulting tables with some data are shown in Figure 7-19.

speciesID	species_name	common_name	genus
1	viscosa	ake-ake	Dodonaea
2	atlantica	atlas cedar	Cedrus
3	nigra	black walnut	Juglans
4	melanoxylon	Tasmanian blackwood	Acacia
5	hippocastanum	Horse Chestnut	Aesculus
6	glutinosa	Black alder	Alnus

Species table

species	use
1	hedging
1	shelter
1	soil stability
2	shelter
3	firewood
3	shelter
3	soil stability

Species_Use table

use
bee food
bird food
coppicing
firewood
hedging
shelter
soil stability
timber

Use table

Figure 7-19. *Representing a Many–Many relationship with an additional table with two foreign keys*

3 soil stability

Figure 7-19. Representing a Many–Many relationship with an additional table with two foreign keys

We now have to decide on a primary key for the new `Species_Use` table. The combination of the two foreign key fields (`speciesID`, `use`) will do the trick. This combining of foreign keys to form a primary key is often the case in the situation where an intermediate table has been introduced in a Many–Many relationship.

Representing 1–1 Relationships

In all of the previous sections, we have always ended up taking the primary key field at the 1 end of the relationship and using it as a foreign key in the table at the other end. If both ends of the relationship have a cardinality of 1, which way around should we do this?

Our example of members and teams had a 1–1 relationship: *is captain of*. That part of the data model is shown in Figure 7-20.

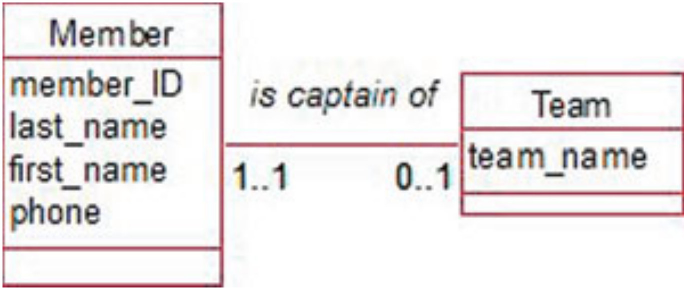


Figure 7-20. Is captain of is a 1–1 relationship.

Representing Inheritance

Relational databases do not have the concept of inheritance built into them; however, it is possible to approximate the idea of inheritance.

As discussed in Chapter 6, inheritance is very useful to model tricky problems, but it should only be used when other more simple patterns cannot fully represent some essential complications. Figure 7-22 shows a simple case of inheritance in which lecturers and students inherit the attributes of a person and also have some specialized attributes of their own.

One way to capture the main aspects of inheritance in a relational database is to set up classes for each parent class and subclass and include a 1–1 relationship between each subclass and its parent as shown in Figure 7-23. The relationships (reading upward) say that a lecturer *is a* person and a student *is a* person, which is a natural way to think about the model.

The relationship between `Student` and `Person` in Figure 7-23 is compulsory at the top end because every student is a person, but optional at the bottom end because a person does not have to be a student. We can now set up tables as we did for the 1–1 relationship in the previous section. We choose to put the foreign key (`personID`) in the `Student` table (because a student has to be a person) and similarly for the `Lecturer` table. `personID` will also be the primary key of the `Lecturer` and the `Student` tables. We will end up with three tables as shown in Figure 7-24.

The question is whether to put `member_ID` as a foreign key in the `Team` table or `team_name` as a foreign key in the `Member` table. The resulting tables for these alternatives are shown in Figure 7-21.

member_ID	last_name	first_name	is_captain_of
105	Smith	Jim	SeniorB
118	Jones	Bob	
198	Green	Linda	
199	Norton	Harvey	JuniorA
216	Brown	Sue	
235	Brown	Grant	
265	King	Jane	

Foreign key in Member table

OR

team_name	captain
JuniorA	199
SeniorB	105

Foreign key in Team table

Figure 7-21. *Alternative ways to represent the 1–1 relationship*

The same information is represented in both tables. We mustn't do both simultaneously, as we might end up with inconsistent data. For example, we could end up with Bob being captain of SeniorB according to the `Member` table, but Jim being the captain according to the `Team` table.

In the `Member` table in Figure 7-21, we have many empty values for the field `is_captain_of` because that end of the relationship is optional (a member doesn't have to be a captain and most members won't be). In general, you should put the foreign key in the table that has the compulsory association if there is one. A team *must* have a captain, so put the foreign key in the `Team` table.

Putting the foreign key in the `Team` table ensures that each team only has one captain, however there is nothing to prevent a member being captain of more than one team (e.g., there is nothing currently in the design of the `Team` table to prevent us putting 199 on more than one row). With the foreign key in the `Member` table we have the opposite situation: a person can only captain one team but we could have JuniorA appearing on several rows, and so effectively having several captains. We will see how to enforce both constraints with unique indexes in Chapter 9.

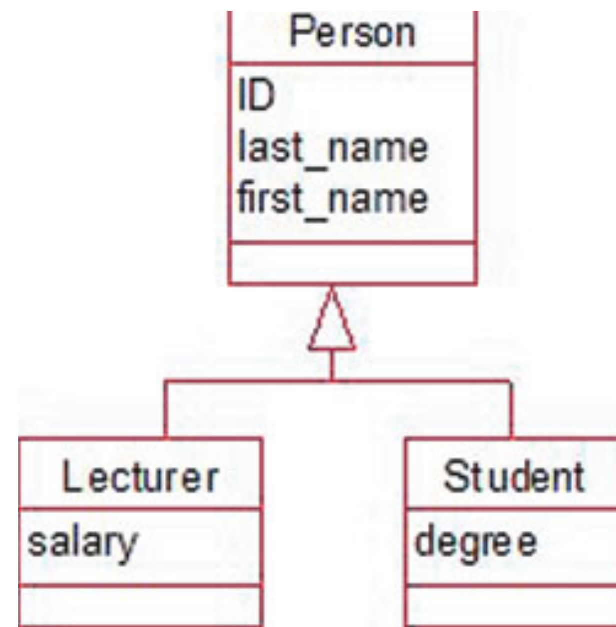


Figure 7-22. *Simple model with inheritance*

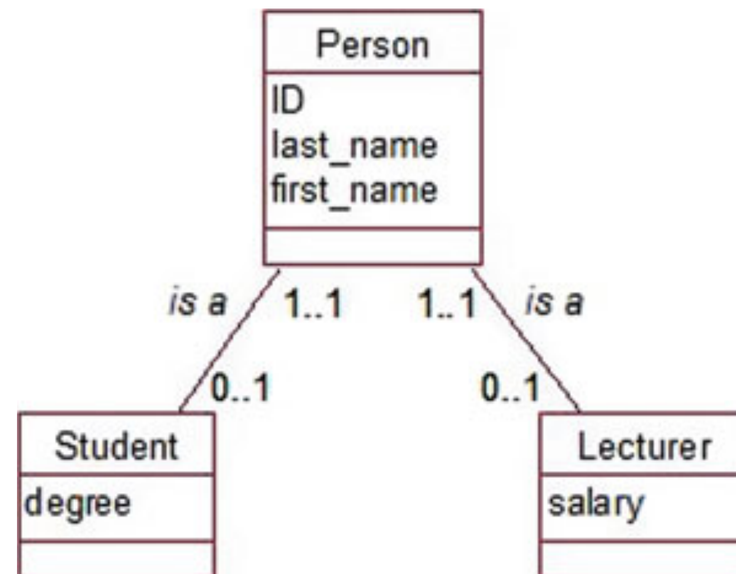


Figure 7-23. *Inheritance approximated with 1–1 is a relationship*

0..1 0..1

Student
degree

Lecturer
salary

Figure 7-23. *Inheritance approximated with 1–1 is a relationship*

ID	last_name	first_name
101	Jones	Sue
108	Brown	Lin
110	Li	Bo
112	Green	Mike

Person table

personID	salary
101	75000
108	90000
110	12000

Lecturer table

personID	degree
108	Arts
110	Science
112	Arts

Student table

Figure 7-24. *Tables representing the model in Figure 7-23*

What elements of the inheritance have we captured? Well, we have the contact details of each person in just one place–the Person table. We know who are lecturers and who are students, and we have the specialist attributes of each role neatly stored in the appropriate tables. As a special bonus, we have also managed to capture multiple inheritance! John and Linda feature in both the lecturer and the student tables.

What is different from true inheritance? In the model in Figure 7-23, we have just *one* object for Sue (a Lecturer object). In Figure 7-24, we have *two* rows (a row in the Person table and a row in the Lecturer table) with a relationship between them. Extending the model is also different in the two cases. If, at a later date, we require an additional subclass (e.g., Administrator), this can be added quite simply to the hierarchy in Figure 7-22. In Figure 7-23, we would need to add another class but in addition also create and maintain another relationship.

Summary

We have taken a data model and represented the main features using the functionality available in relational database products. Following is a summary of the steps:

1. For each class, create a table.
2. For each attribute, create a field and choose an appropriate data type. Consider whether some attributes (e.g., address) should be split into several fields.
3. Think about which fields should be required to have a value.
4. Consider what constraints need to be placed on the values of fields. Possibly create a new domain if your database product supports this.
5. Choose a field or combination of fields as the primary key. Ask careful questions to ensure that the key fields will always have unique values.
6. For each Many–Many relationship, insert a new intermediary class and two 1–Many relationships.
7. For each 1–Many relationship, take the primary key field(s) from the table representing the class at the 1 end and add this field(s) as a foreign key in the table representing the class at the Many end.
8. For a 1–1 relationship, put the foreign key in the table where it is most likely to have a value or where the attribute is most important.
9. For compulsory relationships, add a constraint to the foreign key fields that they must not be null.
10. For inheritance (as an approximation), alter the model to have a 1–1 *is a* relationship between the parent and each child class. Create tables and foreign keys as in point seven.

TESTING YOUR UNDERSTANDING

Exercise 7-1.

Figure 7-25 shows an initial data model for a small library. It is incomplete, so as you answer the questions below consider what else might need to be included.

a) Explain to the librarian what the initial data model means.

b) Design tables for a relational database which would capture the information represented by the model. Include primary and foreign keys and other appropriate constraints.

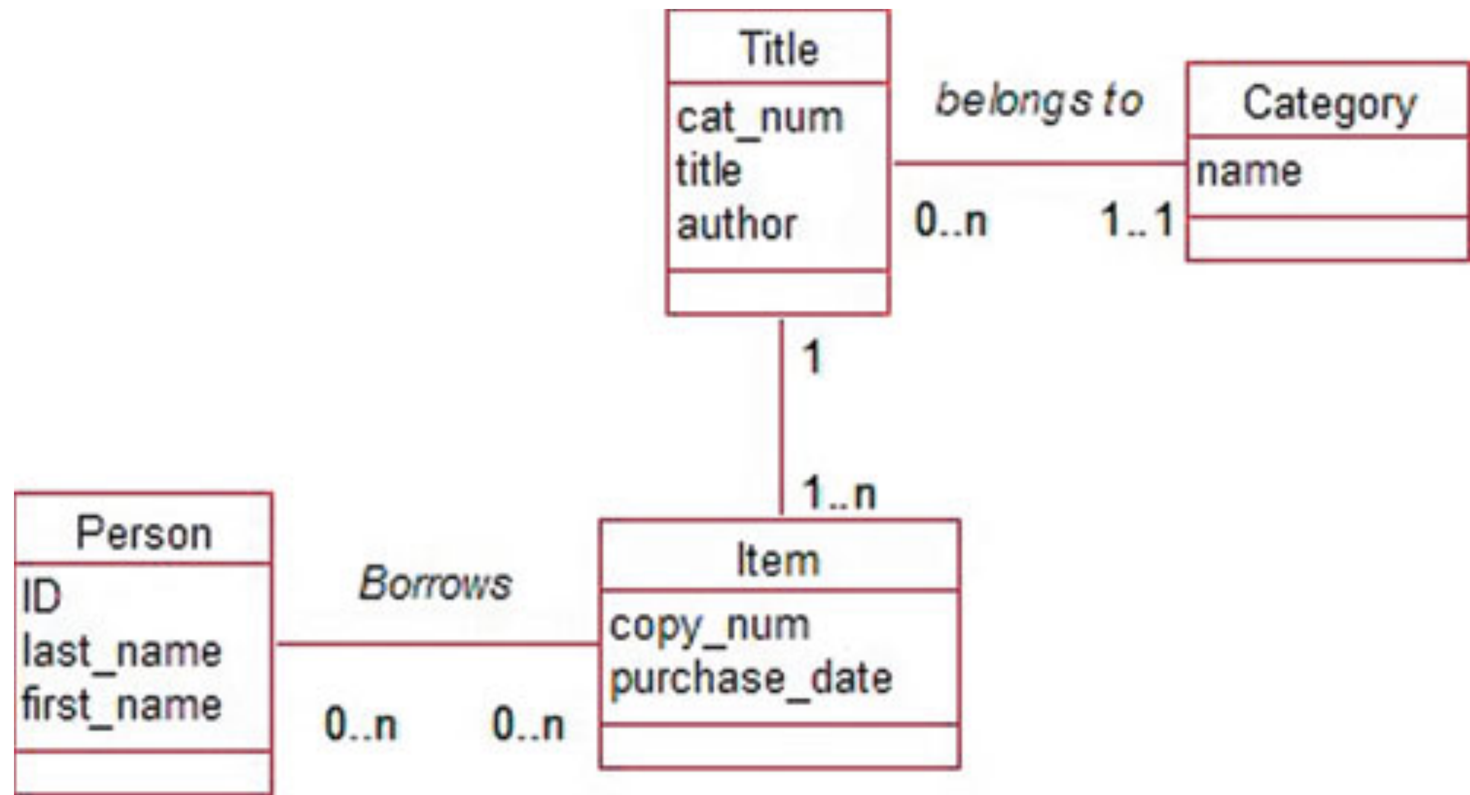


Figure 7-25. *Draft data model for a small library*