CHAPTER 4

Learning from the Data Model

In the previous chapter, we attempted to extract the essential tasks involved in a real-world problem and express them with use cases. We also made a first attempt at determining the data that are necessary to support those tasks and formed an initial data model, which we depicted with a class diagram. In this chapter, we look more closely at the data model to see how it can further our understanding of a database system.

A data model is a precise description of the data stored for a real–world problem, in much the same way that a mathematical equation describes a real-world physical event, or an architectural drawing describes the plan of a building. However, like a mathematical equation or an architectural plan, the data model is neither a complete nor an exact description of a real situation. It will always be based on definitions and assumptions, and it has a finite scope. For example, a high school student's simple mathematical equation to describe the path of a ball tossed into the air will probably make assumptions about the constancy of the gravitational force and the absence of air resistance, and will likely assume low speeds where relativistic effects can be ignored. The equation is precise and correct for the assumptions that have been made, but it does not reflect the real problem exactly. It is, however, a good, pragmatic, and extremely useful description that captures the essentials of the real physical event.

A data model has similar benefits and limitations to a mathematical equation. It is a model of the relationships among the *data items* that are being *stored* about a problem, but it is not a complete model of the real problem itself. Constraints on money, time, and expertise will always mean that problems will need to be scoped and assumptions made in order to extract the essential elements. It is crucial that the definitions and assumptions are clearly expressed so that the client and the analyst are not talking at cross–purposes.

In the early stages of the analysis, as client and developer are trying to understand the problem (and each other), the details will necessarily be vague. In this chapter, we look at how the initial data model can be used to discover where definitions and scope may need to be more rigorously expressed.

Review of Data Models

The essential aspects of a data model were defined in Chapter 2. We will revisit these by way of an example that will highlight some additional features. Think about a small hostel that provides a number of single rooms for school groups visiting a national park. The hostel has a small database to keep track of its rooms and the people currently in residence. Because the hostel primarily deals with groups of students with a single point of contact, the idea of a group is central to their business model. It is still important to know which rooms particular students or teachers have been allotted. An initial data model to capture this information is shown in Figure 4-1.

You can see that there is a 1–Many relationship between the `Group` and `Guest` classes. Reading from left to right in Figure 4-1, we have that a particular group is related to one or more guests, and from right to left that a particular guest is associated with exactly one group. Figure 4-1 also depicts a 1–1 relationship between `Guest` and `Room`. Reading left to right, we have that each guest must be associated with one room and in the other

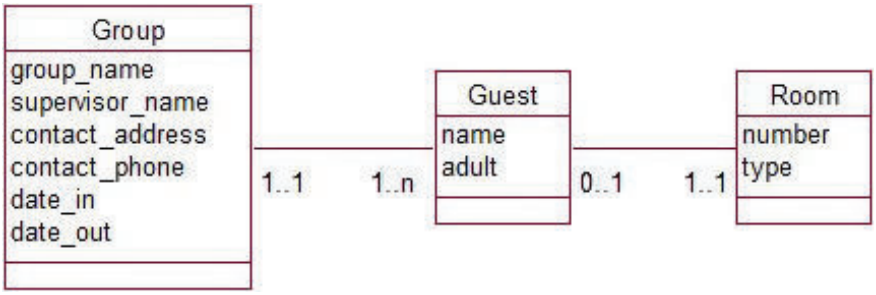

Figure 4-1. *Initial data model for the current occupancy of a small hostel*

direction that a room can be associated with at most one guest but maybe none. In normal speak, we have that groups consist of a number of guests, and each guest has a room. Rooms are for one guest only, and they may not all be full. Some possible instances of these objects and relationships are shown in Figure 4-2. We have

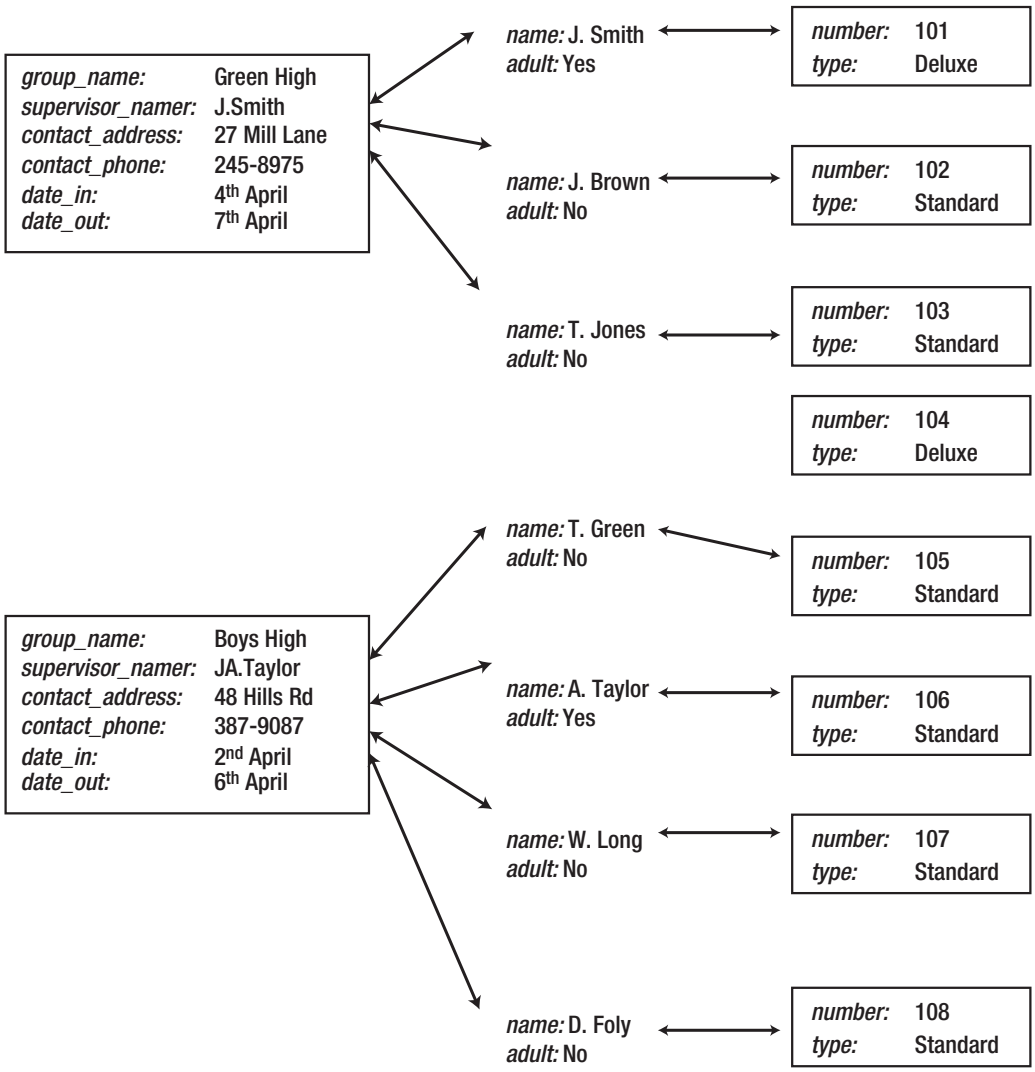

Figure 4-2. *Objects and relationship instances consistent with Figure 4-1*

two groups: Green High with three associated guests, and Boys High with four. Each of the guests is associated with one room (and some rooms are empty). Take a little time to convince yourself of how the class diagram in Figure 4-1 represents the situation.

Notice that room 104 is empty as is allowed by the data model (a room does not have to be associated with a guest). Now that we have read what the data model tells us in a mechanical way, let's think a little bit more deeply about what the model is telling us about the real problem and how it is being handled.

What is the definition of a *group* for this hostel? We see that in the example data in Figure 4-2 the Boys High group consists of four people. There is only one arrival date and departure date for the entire group and just the one set of contact information that applies to all the group members. In this respect, our definition of *group* is a little different from what we might expect in normal conversation. It is not a set of people who all know each other and feel as though they belong together, but a set of people with the same arrival and departure dates and common contact information. For the most part this model will work well for the hostel, but there will be the odd exceptional case. What would we do if A. Taylor and W. Long need to leave the Boys High group a day early? How could we record this information? There is no place to store dates with Taylor's or Long's `Guest` object, and there is only room for one departure date to be stored with the Boys High `Group` object. Within this model, we could capture this information by creating another `Group` object for Taylor and Long (Boys High Early Leavers, say) with a different set of dates. That would cope with knowing who is coming and going when. If it is essential, however, that the system needs to record that these two groups of Boys High people are somehow "together," the data would need to be modeled differently.

The data model also tells us that a guest *must* belong to a group. What else does this tell us about the definition of a group? What about a lone traveler wishing to stay at the hostel? Given that the hostel is primarily set up for groups of people this is unlikely, but it can be accommodated within the model. We can have a group with just one guest. In this respect, the definition of *group* for this database problem is once again different from the way the word is used in normal conversation. We would not generally refer to a *group of one person*; however, for this data model that is a possibility that might eventuate.

So our original data model, which at first glance looked quite simple, has told us quite a bit about how the problem is being dealt with. It has led us to a precise definition for a *group*:

> *A group is a set of guests with common contact information and with identical arrival and departure dates. A separate group will need to be formed for each different set of arrival and departure dates. A group can have one or more guests associated with it.*

By being careful with the definition of the `Group` class, we have avoided needing special cases for groups with more than one set of dates or for guests traveling alone. This keeps the problem and its solution simple. Of course, if the majority of guests were lone travelers, we would rethink the problem and model it in an entirely different way.

In the rest of this chapter, we will look at questions we can ask about small pieces of a data model in order to learn more about the problem at hand. The questions we will look at only apply to relationships between two classes, but they can open up a great deal of discussion about the problem. As more is understood about a problem, what we learn from the data model can be reflected in the use cases. The questions we will consider are as follows:

- **Optionality:** Should it be 0 or 1?
- **Cardinality of 1:** Might it occasionally be 2?
- **Cardinality of 1:** What about historical data?
- **Many–Many:** Are we missing anything?

Optionality: Should It Be 0 or 1?

As described in Chapter 2, the optionality of one end of a relationship is the smallest number of objects that can be associated with an object at the other end. This is usually 0 or 1. For example, in Figure 4-1, reading the relationship between guest and room from left to right, we have that a particular guest *must* be associated with a room (optionality 1), whereas reading the relationship from right to left, we see that a particular room may have no related guest (optionality 0).

Optionalities can provide a great deal of information about the definitions of classes and the scope of the problem. We will look at a few small examples, each of which illustrates some aspect of deciding on the appropriate optionality.

Student Course Example

Consider the data model in Figure 4-3, which shows a relationship between students and courses in which they enroll.

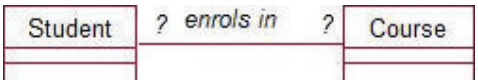

Figure 4-3. *Data model for students enrolling in courses*

On first sight this is quite trivial: a student can enroll in many courses, and a course can have many students enrolled in it. What about the optionalities? Can a student be enrolled in no courses? Our normal conversational definition of a student is someone who is studying or, more accurately, is formally enrolled in a course (which is quite different really!). What is our definition of student for this database? It has been a long time since I've been able to be described as a student in normal conversation, but I am quite sure I still feature in the student database at my former university. For the purpose of this example then, we might define a student as someone who is, or has been, enrolled in a course.

Does it make any sense to have a "student" in our database that is not and has never been enrolled in any courses? What about a person who has been accepted into a university but has not yet made final decisions about any specific courses? Is this person a student? The university would certainly want to keep information about such a person (her ID, name, address, and so on). We can accommodate this situation by expanding our definition of a student to include people accepted by and/or registered with the university.

What about a person who has contacted the university and asked to be sent information about enrollment? Any typically cash–strapped institution will want to keep information about such a person. Asking this question starts to involve issues about the scope of the problem as well as the definition of a student. It is important that questions such as "Exactly who are these people you call students?" are considered right at the start of the analysis process. Is the system to include contact details for everyone who has ever expressed an interest in attending the university, or is the scope to be restricted (at least for the time being) to records of current and former students?

Clearly, only the client can answer these questions. It is useful to see how careful consideration of the details of even the most simple data model can lead to important questions about much wider aspects of the problem. Asking whether a student must be enrolled in a course may seem pedantic at first, but until we can answer that question clearly, we have not even begun to understand the problem we are trying to solve.

Reading the relationship from right to left and questioning whether a course must have a student enrolled in it leads to a similar debate about how we define a course. What data might we want to keep about a course? Think about all the different situations we might need to deal with. We might need to consider former, current, or proposed courses; popular courses offered more than once concurrently (two streams); and unpopular courses

that are on the books but lack students. You cannot come up with absolute answers without being able to discuss the situation with a client, but you can come up with some possible definitions for consideration.

Customer Order Example

Here is an easier example. (Or is it?) We keep information on customers and the orders they place. Our first instinct is to say that customers can place many orders and each order is placed by one customer. This can be represented as in Figure 4-4.

Figure 4-4. *Data model for customers placing orders*

What about the optionalities? Consider the relationship from left to right. Can a customer be associated with no orders? This depends on the definition of a customer. For the purposes of many businesses, it might be *anyone I am hopeful of selling something to*. A working definition such as *anyone who has ever placed an order and other people who are to be sent catalogs* seems reasonable and suggests an optionality of 0 (i.e., customers in our database have not necessarily placed an order). However, this definition should probably spark a few questions such as, "Do you want to be able to identify people who have previously placed orders but who are now fed up with being sent catalogs?"

Reading the relationship from right to left, we want to know whether each order *must* have an associated customer. This seems trivial. What is the point of an order if we don't know who it is for? If an order arrives in the mail with no name or address, it would be reasonable to say that it should not be entered in the database, and so from this perspective we can insist that every order must have a customer (optionality 1).

However, there is a subtle difference between knowing for whom an order has been placed and relating it to a customer object in the database. A written order may come in the mail from Mrs. Smith of Riccarton Road. While we know who placed the order, that is different from associating it with a customer. We may have to create a new object if Mrs. Smith is a new customer, or we may be faced with deciding which of the existing three Mrs. Smiths this order is from. The problem of distinguishing customers with similar details or deciding whether two or more entries in the customer database actually refer to the same real person can be difficult. Once again, we are not trying to solve any of these issues just now. We are simply using the data model to make us think clearly about some of the issues we will have to confront.

Insect Example

Here is another example of how investigating the optionalities of a relationship can lead to questions about the scope of the problem. Figure 4-5 shows part of a possible data model from Example 1-3 in which farms were visited and several samples of insects were collected. A `Visit` object would contain information about the date and conditions of a particular visit and would be associated with several `Sample` objects. Each sample object would contain information about the number of insects collected.

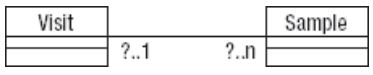

Figure 4-5. *Data model for collecting samples*

Asking whether a sample *must* be associated with a visit is like the question in the previous section about whether an order *must* have a customer. If for this research project our samples only come from farms, it is reasonable that we had to visit a farm to collect them, and so each sample should always be associated with a visit. However, if the scope of the database is broader, with records of samples that have been stored for years and whose origin is uncertain, we may have to reconsider. If we insist that a sample is associated with a visit then we will not easily be able to record information about these historical samples.

Asking whether each visit must have an associated sample (should the optionality at the sample end be 0 or 1?) leads to an interesting question. Is it possible that at some time we may want to record visits to farms for reasons other than collecting samples? These questions may seem trivial, but the broad understanding of the larger problem can only be improved.

A Cardinality of 1: Might It Occasionally Be Two?

Every part of a problem is susceptible to exceptional occurrences. During the analysis of a situation, it is important to think carefully about different scenarios to ensure that the database will be able to cope adequately with all the data that may eventuate. Some "exceptions" are really complications that have been overlooked. Real life and real problems are always complicated. Even something as simple as *write down your usual address* can have hidden difficulties, as many children in shared custody discover when they have to fill out an address on a school form. It might seem picky to insist on asking "Might a person have more than one usual address?" but thousands of modern–day families cannot be shrugged off as exceptional.

In this section, we will look at how to deal with "exceptions" that do not warrant a complete overhaul of the problem but nevertheless are likely to turn up during the lifetime of the database. We have already seen an example of a likely exception earlier in this chapter in the hostel data model. There we considered the case in which some members of a group might want to leave before the others. In the hostel data model, rather than complicating the problem by allowing each group to have several dates, we redefined what we meant by *group* for the purposes of storing the data (i.e., a set of people arriving and leaving on the same dates).

The following sections provide some other examples where a different definition can help cope with some foreseeable, but unusual, events.

Insect Example

In the previous section we looked at the example of a scientist visiting a farm to collect insect samples. Some insects might behave differently if it is fine or raining so it may be important to record information about the weather when the sample was collected. To record the weather conditions consistently, the scientist may decide to choose from one of a number of categories. Introducing a `Weather` class with objects for the different conditions (e.g. fine, overcast, raining) can ensure that this information is recorded consistently. Part of a possible class diagram to represent the data is shown in Figure 4-6.

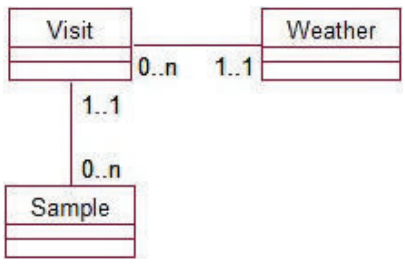

Figure 4-6. *Associating a weather category with a visit*

Reading the relationship between weather category and visit from left to right, it is reasonable that a visit will have one weather type that describes it, but there might also be occasions when a thunderstorm arrives while the last few samples are being collected. If so, do we care? The answer will, of course, depend on the client, but it is up to the analyst to ask the question and propose some possibilities.

At one extreme, the conditions under which each individual sample is collected may be vital. In this case, it might be more sensible to associate each sample with its own weather condition, as shown in Figure 4-7.

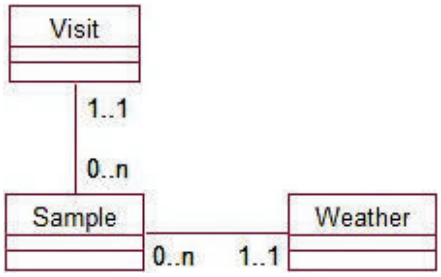

Figure 4-7. *Associating each sample with a weather category*

This latter solution may be overkill when the majority of visits have stable weather conditions. It seems pointless to record the same weather condition for each of 50 samples. A compromise solution may be to say that, if the weather changes markedly, we will create another visit. This way all visits have a single associated weather type, and we can cope with the "exceptional" case by redefining what we mean by a visit. For example:

> *A visit is a time spent on a farm during constant weather conditions on a single day. It is possible to have more than one visit to a farm per day.*

This compromised solution is similar to our redefinition of a group for people with different departure and arrival dates in the hostel example. The data model remains unchanged, but our revised definition of a visit is in place for the inevitable day when lightning strikes, so to speak.

Sports Club Example

Here is another little snippet of a database problem. A local sports club may want to keep a list of its membership and the team for which each member currently plays (SeniorB, JuniorA, Veteran, etc.). One way to model this data is shown in Figure 4-8.

Figure 4-8. *Members and their current teams*

The data model as it stands does not require all members to be associated with a team (optionality 0 at the team end). This means members may be purely social or may miss out on being selected for a team. However, we should still ask questions about the maximum number of teams with which a member might be associated. For example, "Can a member play for more than one team and, if so, do we care?" The data model clearly does not allow for historical records to be kept. If a player is promoted from one team to another, he will simply be

associated with the new team, and we will lose information about his association with his previous team. If the scope of the database is simply to record current affiliations of members with teams, then that is OK. (If not, just wait a few moments until the next section.)

Even if we are only keeping information on current team membership, we are always going to come across the situation where injury or sickness necessitates a member of one team filling in for another team for a particular match. How will this affect the data model? This is a question of scope. Why are we keeping this data and what information do we want to be able to extract from the database? If we want to keep track of which players played in particular matches, our data model is woefully inadequate. We will need to introduce a `Match` class and consider other complications (see Chapter 5).

However, the scope of the problem may simply be to record a person's *main* team. This may be to enable team members to be on a list to be phoned if a match is canceled or if there is to be a rescheduled practice or a social outing. If this is the case, the cardinality of 1 in the data model in Figure 4-8 is fine, so long as it is understood that the relationship *plays for* means a player's *main* team rather than just any team they may play for.

A Cardinality of 1: What About Historical Data?

We have had a number of examples of relationships with a cardinality of 1 at one end. A room has one guest; a club member plays for one team. In both these cases, we have been careful to add the word *currently* because over time a room will have many guests and a player many teams. An important question is, "Do we want our system to keep track of previous guests or previous team affiliations?" This is often overlooked during the analysis, and sometimes the oversight does not become evident for some time. A sports club will find its system just fine for the first season but may get a surprise when the next year's teams replace the previous ones, which are then lost forever. In this section, we will look at a few different examples to illustrate how we can manage historical data.

Sports Club Example

To illustrate how the sports club might lose its historical data, let's look at some simple data as it might be kept in a database table. If each member is associated with just one team, that team becomes a characteristic of the member, and the relationship can be represented as an attribute in the `Member` class as in Figure 4-9.

member_no	last_name	first_name	team
152	Abell	Walt	SeniorB
103	Anderson	James	JuniorA
276	Avery	Graeme	JuniorA
287	Brown	Bill	JuniorA
298	Burns	Lance	Veteran

Figure 4-9. *Members and their current teams*

The following season when Bill Brown graduates to the SeniorB team, his previous association with the JuniorA team will be lost. If the historical data are important, the problem must be remodeled to reflect the fact that members will be associated with many teams over time. A revised model is shown in Figure 4-10.

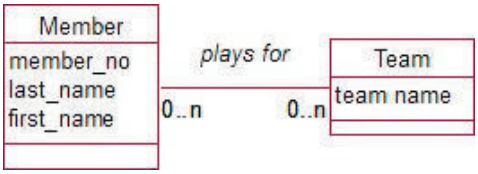

Figure 4-10. *Members and the teams for which they play*

Departments Example

Figure 4-11 is an example that often appears in textbooks. Reading from left to right, we have that each department has one employee as its manager. But clearly this means one at a time. Over time, the department will have several different managers.

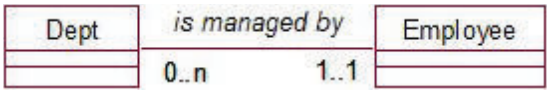

Figure 4-11. *Each department has a manager.*

The important question for this situation is, "Do we want to keep track of former managers?" Why are we keeping information about managers at all? If it is just to have someone to call when something goes wrong, probably the current manager is all that is required. However, if we want to know who was in charge when something went wrong last year, we will need to keep a history. The data model will need to change so that a department can be associated with several managers as in Figure 4-12.

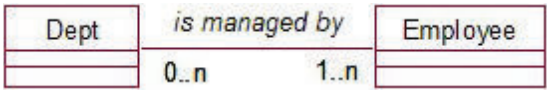

Figure 4-12. *A department has several managers over time.*

We will see in the next section how the introduction of an intermediate class will allow us to keep the dates for each manager.

Insect Example

Here is a real example of a problem arising in our scientific database of insect samples. To put the data in perspective, we need to know that the main objective of this long–term project was to see how the numbers of insects change as farming methods evolve over the years. The farms selected represented different farming types (organic, cropping, etc.). Throughout the duration of the project, each farm was visited several times to collect samples. Figure 4-13 shows part of an early attempt at a data model.

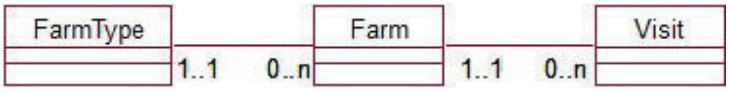

Figure 4-13. *Visits to farms of different types*

At first the data model in Figure 4-13 seemed to be serving its purpose adequately, but this was only because the farming types had not changed during the time the project had been running. However, real trouble was in store. A farm can only be associated with one farm type in this model. When a farm did eventually change, say from a conventional cropping farm to an organic farm, the previous farming type would be lost if the database was set up this way.

A farm can only be associated with one farming type *at a time*. The important question to ask is, "Might the type change over time, and is it important for the system to record that historical data?" In this case, it was critical to the whole experiment to keep information about the history of the farm types, but no one had noticed the problem because the time frames for change were very long.

A Many–Many: Are We Missing Anything?

We have come across quite a few Many–Many relationships in our examples so far. For example, a student can enroll in many courses, and a course can have many students enrolled in it. If we widen the scope of some of the examples to include historical data, as in the previous section, a number of 1–Many relationships will become Many–Many relationships (i.e., departments may have many managers, members many teams, and farms many types over a length of time).

Often we find that we need to keep some additional information about a Many–Many relationship. In the sports team example, we altered the model of members and teams to allow a member to be associated with more than one team. However, if we look at the model in Figure 4-10, we have no idea *when* those associations occurred. The historical data will not be of much use without a date attached somewhere. But where will the date go? In Figure 4-10, we have two classes: `Member` and `Team`. The date does not belong as an attribute of `Member` because it will be dependent on which team we are interested in. Similarly, the date cannot be an attribute of the `Team` class because there will be different dates for each of the players. This problem occurs often and is usually remedied by the introduction of a new class.

We need to ask the question:

> *Is there any data that we need to record that depend on particular instances of each of the classes in our Many–Many relationship?*

In this example, the question would be:

> *Is there any data that depend on a particular player and a particular team?*

And the answer is:

> *Yes—the dates that player played for that team.*

Figure 4-14 shows how an intermediate class can be incorporated into the Many–Many relationship so that data that depend on a particular pairing of objects from each class can be included.

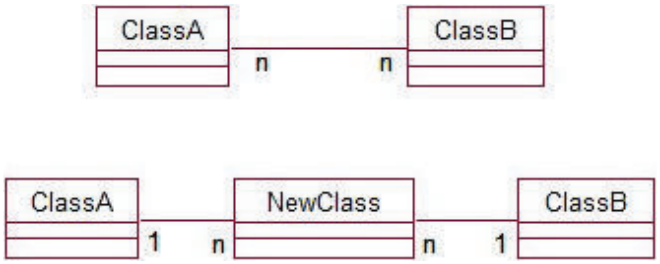

Figure 4-14. *Introducing a new class in a Many–Many relationship*

In situations where we have data that depend on instances of both classes in a Many–Many relationship, the Many–Many relationship is replaced by a new class and two 1–Many relationships. The many ends of the new relationships are always attached to the new intermediate class. We will see what this means for some of the examples we have already examined.

Sports Club Example

Let's reconsider the member and team problem. We'll put some attributes in the classes to make it clearer what information each is maintaining. The model is shown in Figure 4-15.

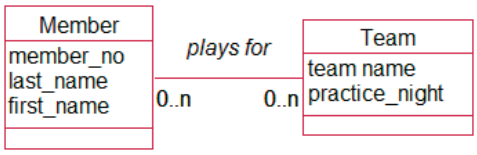

Figure 4-15. *Many–Many relationship between members and teams*

As we have already mentioned, the date that a particular member plays for a particular team cannot live in the `Member` class (because a member will play for many different teams over time) nor can it live in the `Team` class. Figure 4-16 introduces a new intermediate class, `Contract`, in the same way as was done in Figure 4-14.

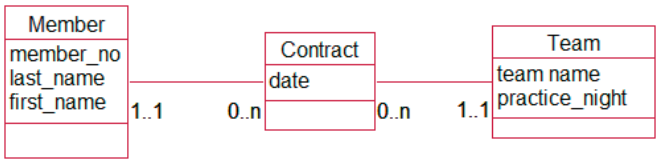

Figure 4-16. *Intermediate class, Contract, to accommodate the date a member played for a team*

Reading from the middle class outward, the model tells us that each contract is for exactly one team and exactly one member. Reading from the outside inward, we see that each member can have many contracts as can each team. Figure 4-17 shows some objects that might occur in such a data model.

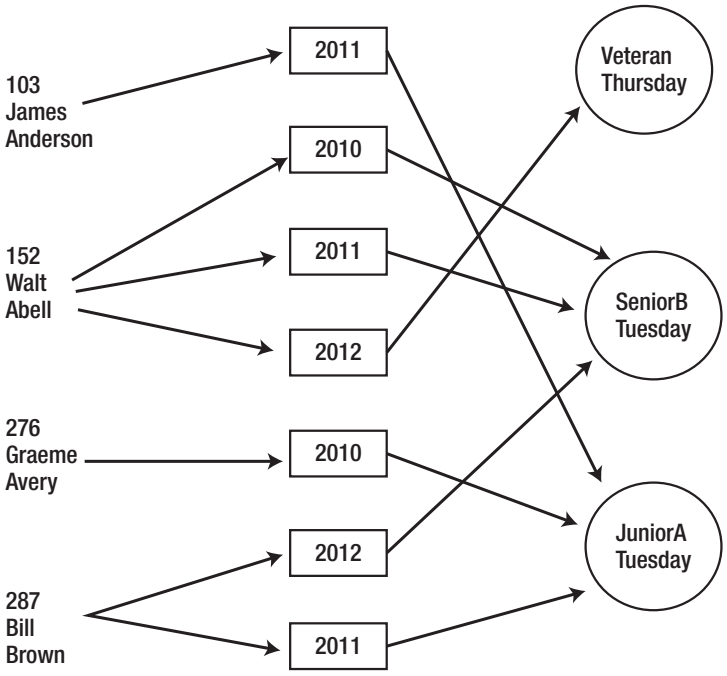

Figure 4-17. *Some possible objects of the Member, Contract, and Team classes*

We can now see what years members played for particular teams. We can see that Bill Brown (287) played for the JuniorA team in 2004 and for the SeniorB team in 2005. These data would be stored in database tables as shown in Figure 4-18.

member_no	last_name	first_name
152	Abell	Walt
103	Anderson	James
276	Avery	Graeme
287	Brown	Bill
298	Burns	Lance

Member

team_name	practice_night
JuniorA	Tuesday
SeniorB	Tuesday
Veteran	Thursday
Under 18	Monday

Team

member	team	year
103	JuniorA	2011
276	JuniorA	2010
287	JuniorA	2011
287	SeniorB	2012
152	SeniorB	2010
152	Veteran	2012
152	SeniorB	2011

Contract

Figure 4-18. *Data for players, contracts, and teams*

Student Course Example

Let's now return to the Many–Many relationship of students enrolling in courses (Figure 4-3). This isn't just a historical problem, although we clearly will want to know when the student completed the course. But even if we were only keeping student enrollments for a single year or semester, we should still look to see whether there is missing information that might require an extra class. The question that needs to be asked is:

> *Are there any data that I want to keep that are specific to a particular student and his or her enrollment in a particular course?*

One obvious piece of data that fits the preceding criteria is the result or grade. Once again, we cannot keep the grade with the `Student` class (because it requires knowledge of which course) nor with `Course` class (because the grade depends on which student). In the same way as we dealt with this situation in Figure 4-14, we can introduce a new class, `Enrollment`, between the `Student` and `Course` classes as shown in Figure 4-19.

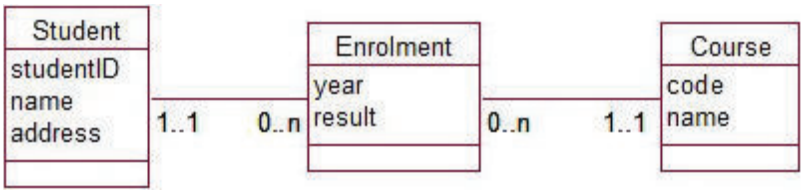

Figure 4-19. *Intermediate class to accommodate the result (and the year)*

A student and a course can each have many enrollments, and a particular enrollment is for exactly one student and one course. If we were to draw some objects, we would get a picture very like that in Figure 4-17 with students, enrollments, and courses replacing players, contracts, and teams.

Meal Delivery Example

As a final example of when we might need an additional class to keep information about a Many–Many relationship, let's look again at the meal delivery problem (Example 3-1) from the previous chapter. The initial data model had a Many–Many relationship between types of meal and orders. A particular type of meal (a chicken vindaloo, say) might appear on many orders, and a particular order may include many different meal types, as shown in Figure 4-20.

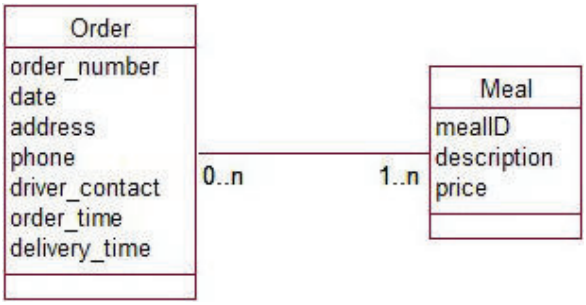

Figure 4-20. *Orders for different meal types*

What happens if a family orders three chicken vindaloos, one hamburger, and one pork fried rice? Where do we put these quantities? The quantity cannot be an attribute in the `Order` class (for this order there are three

quantities and they each depend on the particular meal) nor in the `Meal` class (for there will potentially be hundreds of orders involving a particular type of meal, each with different quantities).

Once again, our problem of where to put the additional data is solved by including a new class as shown in Figure 4-21.

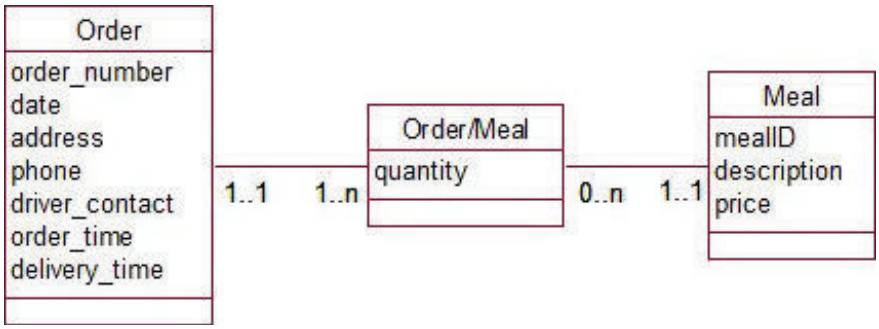

Figure 4-21. *Orders for different types of meal–with additional class to store quantities*

For some problems, it can be difficult to come up with a meaningful name for the intermediate class. In such a case, it is always possible to use a concatenation of the two original class names as we have done here with `Order/Meal`. We could maybe have called the class `Orderline`, in this example, as it represents each line in the order (i.e., a meal and the quantity). You might find it helpful to sketch some objects of the three classes in Figure 4-21 to clarify what is happening.

We can also use this new intermediate class to solve one of the other problems we deferred in Chapter 3. This was the problem of coping with the price of a meal changing over time. In the `Meal` class in Figure 4-21, we can define the `price` attribute as being the current price for that type of meal. An order placed for that meal today will be at that price. How do we know what was charged for this type of meal on an order several months ago? To deal with the problem of changing prices, we can include an attribute, `price`, in the intermediate class `Order/Meal`. This will be the price charged for a particular meal on a particular order and will not change when the current `price` changes in the `Meal` class. This way we have a complete history of the prices for each meal on each order. A `price` attribute in this intermediate class can allow us to keep historical data and also to deal with "unusual" situations such as specials or discounts. We are always keeping the price that was actually charged for that type of meal on that particular order.

The question that needed to be asked about the original Many–Many relationship in Figure 4-20 was:

> *Are there any data we need to store about a particular meal type on a particular order?*

And the answer is:

> *Yes, the quantity of that meal type ordered and the price being charged for that meal type on the order.*

When a Many–Many Doesn't Need an Intermediate Class

A few Many–Many relationships contain complete information for a problem without the need for an intermediate class in the data model. Problems that involve categories as part of the data often do not require an additional class. Example 1-1, "The Plant Database," involved plants and uses to which they could be put. The original data model is repeated in Figure 4-22.

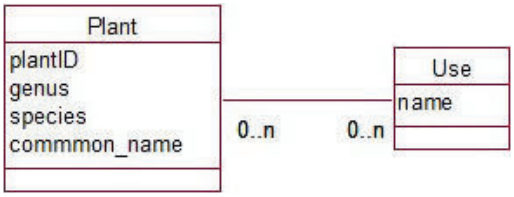

Figure 4-22. *Plants and their uses*

We can ask the question, "Is there any information we want to keep about a particular species and a particular use?"

In this case, the answer is probably, "No." A Many–Many relationship that doesn't require any additional information often occurs when we have something that belongs to a number of different categories; for example, a plant has many different uses and all we want to know is what they are.

It is possible, however, that in a different situation we might want to record whether a particular plant is excellent or just reasonable at hedging. Or we may want to note how many of a particular species are needed to be sufficient for attracting bees. In both these cases, we might need an intermediate class. Try sketching a new model for these situations.

Summary

Even at the very early stages of analysis, a simple data model can provide us with a number of questions. The answers to these questions will help us to understand a problem better. The resulting clarifications to the problem should eventually be reflected in the use cases and may affect the final model and the eventual implementation.

In this chapter, we have suggested some questions about a single relationship between two classes. Some of the questions we have discussed are reviewed here.

Optionality: Should it be 0 or 1? Considering whether an optionality should be 0 or 1 might affect definitions of our classes; for example, "Would a student who was not enrolled in any courses still be considered a student for the purposes of our database?"

A cardinality of 1*:* Might it occasionally be 2? We need to consider whether there might be exceptional cases in which we might want to squeeze two numbers or categories into a box designed for one; for example, "What happens if the weather changes during a visit?" Redefining a class might help out for the exceptional cases, as in, "If the weather changes, we will call it two visits."

A cardinality of 1: What about historical data? Always consider whether the 1 in a relationship really means "just one at a time." For example, "A department has one manager. Do we want to know who the previous managers of the department were?" If so, the relationship should be Many–Many.

Many–Many: Are we missing anything? Consider whether there is information we need to record about a particular pairing of objects from each class; for example, "What might we want to know about a particular student and a particular course?" If there is such information (e.g., the grade), introduce a new intermediate class.

TESTING YOUR UNDERSTANDING

Exercise 4-1.

Figure 4-23 shows a first draft of modeling the situation where a publishing company wants to keep information about authors and books. Consider the possible optionalities at each end of the relationship `writes` and so determine some possible definitions for a book and an author.

Figure 4-23. *Consider possible optionalities for authors writing books.*

Exercise 4-2.

Figure 4-24 shows a possible data model for cocktail recipes. The Many–Many relationship `uses` can be navigated in either direction. To find out the ingredients in a Manhattan or to discover the possible uses for that bottle of Vermouth. What is missing?

Figure 4-24. *Cocktails and their ingredients. What is missing?*

Exercise 4-3.

Part of the data model about guests at a hostel is shown in Figure 4-25. How could the model be amended to keep historical information about room occupancy?

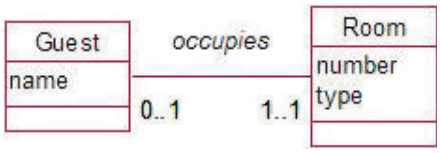

Figure 4-25. *How could this be amended to keep historical information about room occupancy?*

CHAPTER 5

Developing a Data Model

In the previous chapters, you've seen how to determine the requirements of a database problem by considering the tasks users of the system need to carry out. Tasks were represented with use cases, and a simple data model was developed to represent the required data. In Chapter 4, you saw that a great deal can be learned about a problem by questioning some of the details of simple relationships, particularly the number of objects involved at each end of a relationship. In this chapter, you'll be introduced to a few problems that frequently occur in order to enlarge your armory for attacking tricky situations.

Attribute, Class, or Relationship?

It is never possible to say that a given data model is *the* correct one. We can only say that it meets the requirements of a problem within a given scope, and subject to certain assumptions or approximations. If we have a piece of data describing some person or thing or event, it is possible that there may be different ways of representing that information. In this section, we look at a simple problem, described in Example 5-1, for which various pieces of data may be represented as an attribute, class, or relationship depending on the overall requirements of the problem.

EXAMPLE 5-1: SPORTS CLUB

Let's say we are keeping information about current teams for a sports club. The club wishes to keep very simple records of the team name, its grade, and the captain. As a start we could have a class to contain this information as shown in Figure 5-1.

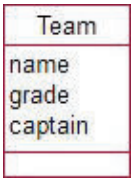

Figure 5-1. *Simple class for Team*

In Figure 5-1, each of the pieces of information we are capturing about a team, the name, grade, and the name of the captain is represented by an attribute. With this model, we can find the values of the attributes for any given team, but that is about all we can do. Of course, that may be all we want to do!

In previous chapters, we saw that it is important to consider how the data being stored might be used in the future. With the data in Figure 5-1, it is quite likely we may want to find all the teams in a given grade. Will the simple data model allow this? It is certainly possible to find all the **`Team`** objects with a given value for the **`grade`** attribute; however, to obtain reliable data, we would require the data entry to be exact. We would not get an accurate list of all teams in senior grade if the value of the **`grade`** attribute for different objects was variously recorded as "Senior," "snr," "Sen Grade," "Senior Grd," and so on. We saw a similar problem in Example 2-1 where we wanted to ensure plant genus information (like *Eucalyptus*) was always spelled correctly. If reliable recording and extracting of data about grades is important for our sports club, we need a data model that will ensure grades are recorded consistently. This can be done by representing the grade of a team as a class as in Figure 5-2. Each possible grade becomes an object of the **`Grade`** class, and each team is related to the appropriate **`Grade`** object.

Figure 5-2. *Representing a team's grade as a class*

Therefore, depending on the requirements of the project, we might choose to represent the grade as an attribute of **`Team`** (if the consistency of the spelling is not important) or as a class of its own (if we think we may want to find all the teams belonging to the same grade, for example).

Now consider the **`captain`** attribute in Figure 5-1. It's unlikely that a person will captain more than one team at a time, so a query analogous to the one in the previous section (find all the teams Jenny currently captains) is unlikely to be a high priority. However, there may be some additional data about a captain that we might like to keep: perhaps her phone number and address. In the context of a sports club, it is highly likely that this information already exists in some membership list. We very possibly have another class, **`Member`**, that keeps contact information about all the members of the sports club. If so, we can represent a team's captain as a relationship between the **`Team`** and **`Member`** classes as shown in Figure 5-3. A particular object of the **`Member`** class is the captain of an object of the **`Team`** class.

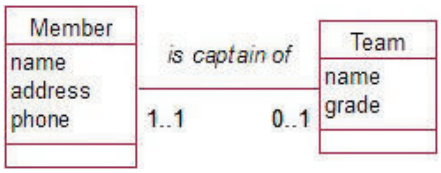

Figure 5-3. *Representing captain as a relationship*

Once again, depending on the problem, we have different ways to represent a team's captain. We might choose to represent the captain as an attribute of **`Team`** or as a relationship between **`Member`** and **`Team`**. The determining factor here will be whether the problem requires information about members in general.

Some useful questions to ask when considering whether to represent information as an attribute, class, or relationship are summarized here:

- "Am I likely to want to summarize, group, or select using a given piece of information?" For example, might you want to select teams based on grade? If so, consider making the piece of information into a class.
- "Am I likely now or in the future to store other data about this piece of information?" For example, might you want to keep information such as phone and address about a captain? Does (or should) this information already exist in another class? If so, consider representing the piece of information as a relationship between the classes.

Two or More Relationships Between Classes

How would the model in Figure 5-3 change if we also wanted to keep information about the people playing for a team? We may need to know their names and phone numbers. Keeping all this information as attributes of the **`Team`** class will rapidly become unwieldy. We would need attributes such as **`player1Name`**, **`player1Phone`**, and so on. Once again, we probably have the information we require about players already in a **`Member`** class. The fact that particular members play for a particular team can therefore be represented as a relationship between the classes. This is very similar to the situation in Example 2-1 where plant objects were related to particular use objects. Figure 5-4 shows the addition of this relationship between **`Member`** and **`Team`** in our data model.

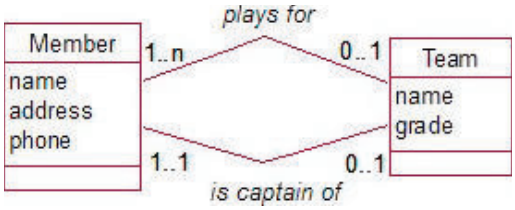

Figure 5-4. *Different relationships between Member and Team*

We now have two relationships between our **`Member`** and **`Team`** classes. One is about which members play for the team (could be many). The other is about which member is the captain of the team (just one). The model in Figure 5-4 also allows for members who do not play for or captain any teams (they may be social members of the club). Such members would simply not be linked to a team.

You may be wondering whether a captain of the team should always be one of the players in that team. The model as drawn in Figure 5-4 does not have anything to tell us about such a constraint. There are a number of ways to represent constraints such as this. It is possible to make the constraints part of the relevant use case. For the use case describing entering information about a team, we would say that the captain has to be one of the players. The *Object Constraint Language (OCL)*, developed by the Object Management Group,[1] to supplement the Unified Modeling Language, UML, provides a formal specification for constraints. I will not delve into formal methods in this book, preferring to draw attention to these additional constraints in the use case text.

Another situation in which it is possible to consider two relationships between classes is when we have historical data. Example 5-2 revisits the **`Rooms`**, **`Groups`**, and **`Guests`** example that we first examined in Chapter 4.

[1] http://www.omg.org/

EXAMPLE 5-2: SMALL HOSTEL

A small hostel consists of single-occupancy rooms. Typically, groups of people (e.g. school classes) stay at the hostel. We will expand the problem from Chapter 4 by keeping information about previous guests as well as current guests. A room will have many guests over time.(For simplification, we will assume that a guest only stays once and in one room.) The revised model is shown in Figure 5-5.

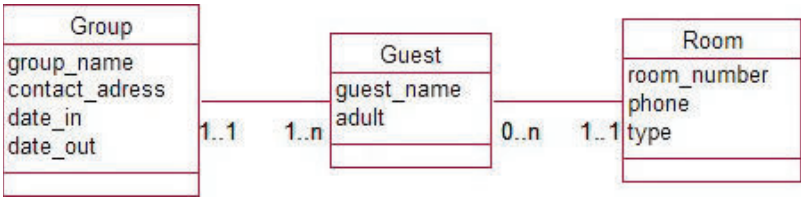

Figure 5-5. *Model for a single occupancy room having many guests over time*

The hostel primarily caters to groups of visitors and so the check–in and check–out dates belong with the group rather than each individual guest.

What can we find out from the model in Figure 5-5? If we have some query about a guest, we can easily find his room number, and we can also find the length of the stay by checking the dates of the related group object. Things become a little more complicated if we want to find the name of the guest currently occupying a room (say there has been a complaint about noise). There are an increasing number of guests associated with each room over time, so how do we go about finding the current one? One way would be to search through all the guests associated with the room and check their associated group information to find one with a **`date_out`** value in the future. Another likely task is to retrieve a list of empty rooms. To do this, we would have to find those rooms without a guest belonging to a group with a **`date_out`** in the future.

These solutions are quite feasible, but for tasks that are likely to be required regularly, they are complicated and tedious. A different option is to consider having a second relationship between **`Room`** and **`Guest`** for the *current* guest. All guests will be associated with a room as in Figure 5-5, but we add an additional relationship between the current guest and the room. This is shown in Figure 5-6.

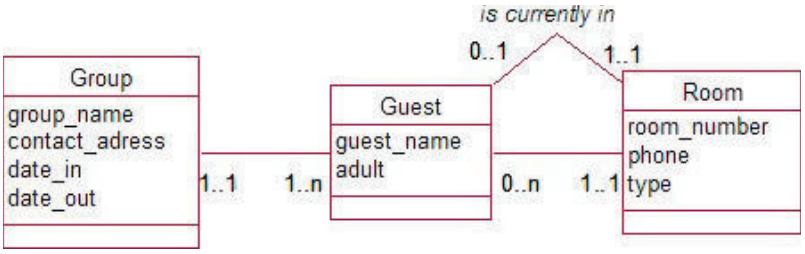

Figure 5-6. *Alternative model for a room having many guests over time*

With the data model in Figure 5-6, we can find the current guest with reference to objects of only two classes (**Room** and **Guest**). With the model in Figure 5-5 we needed to inspect date attribute values of the **Group** object as well. To find empty rooms, we can now simply look for all rooms that have no current guest.

There are a few problems with modeling the data in this way as some extra updating is required to keep the data consistent. For example, when a group checks out, we will have to update the **date_out** in **Group**, and we will also have to remove each *is currently in* relationship instance to reflect that the room is now empty. This extra maintenance step is caused because we are in effect storing the same piece of information in more than one way. While the *retrieval* of information about empty rooms and current guests is simpler for the model in Figure 5-6, the *updating* of data is more complex. Table 5-1 shows possible use cases for checking out a group, and a report for listing all currently empty rooms for each data model.

Table 5-1. *Some Possible Use Cases for the Alternative Guest and Room Models*

	With Data Model in Figure 5-5	With Data Model in Figure 5-6
Check out a group	Update **date_out** for appropriate **Group** object.	Update **date_out** for appropriate **Group** object. Find all associated **Guest** objects and remove the *currently in* association with the **Room** object.
List all currently empty rooms	First find the occupied rooms: find all **Group** objects with **date_out** in the future. Find all the associated **Guest** objects for these groups, and the set of all **Room** objects associated with these guests. List the **room_number** for all **Room** objects *not* included in this set.	Find all **Room** objects that do not have an associated current **Guest** object.

It is clear that the reporting is simpler for a model such as that in Figure 5-6, while the maintenance is simpler for one like Figure 5-5. The problem with the model in Figure 5-6 is that if the updating required when checking out a group is not done correctly, we will end up with a database that has inconsistent information. While the model in Figure 5-6 appears easier to query, it is so at the expense of making the maintenance more difficult and therefore the reliability more likely to be compromised. As you shall see in the next section, it is best to avoid the situation where we have information stored more than once.

At this point we realize that neither of the models in Figures 5-5 or 5-6 is particularly good for the hostel problem if we want to keep historical data about room occupancy. The solution discussed in Chapter 4 allowed us to keep information about what groups had been to the hostel, who the guests were, and the current occupant of a room. That all worked well. Once we try to add the historical data about room occupancy, the model becomes difficult to manage. An alternative solution is suggested in the Testing Your Understanding section for Chapter 4.

Different Routes Between Classes

Using the model in Figure 5-6, we can find the current guest in a room by two routes: via the relationship *currently in* or by checking the **date_out** for each guest who has occupied the room. The problem here is that if the data are not carefully maintained, we might find that we come up with two different answers. For example, if a group is checked out but we did not remove all the *is currently in* associations (as in the use case in Table 5-1), the first route will give us the previous guest in the room, while the second route will show an empty room.

As argued in the previous section, the advantages in easy retrieval may appear to outweigh the associated data maintenance complications. What we should avoid at all costs is having alternative routes for a piece of information when there is no associated reduction in complexity.

Redundant Information

Having what should be the same piece of information available by two different routes can be referred to as *redundant information*. In the previous section, we had redundant information about the current occupant of a room. We could find the current occupant by inspecting the *is currently in* relation, or we could deduce the current occupant by looking at the check–out dates of the groups.

Let's have a look at Example 5-3, which is another case of redundant information.

EXAMPLE 5-3: STARTUP INCUBATOR

A startup incubator has employees who each work for one of a number of different small project groups. Each group and all its employees are housed in one particular room, with larger rooms housing several groups. We may require information such as where each employee is located, a particular employee's phone number, where to find a particular group, which employees work in each group, who is in each room, and so on. One possible data model is shown in Figure 5-7. Take a moment to understand the data model and the information it contains about the number of groups in a room and so on for this particular problem. The model has redundant information. Can you see what it is?

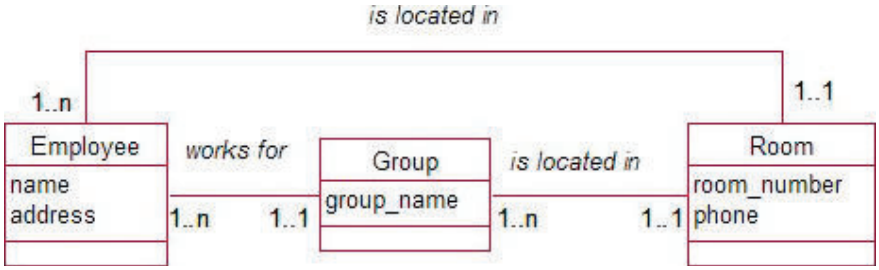

Figure 5-7. *Employee, Group, and Room with a redundant relationship*

With respect to Example 5-3, if we regularly want to find an employee's phone number, we might think that the top relationship in Figure 5-7 between **`Employee`** and **`Room`** would be a useful direct route. However, this same information is very easily available by an alternative route through **`Group`**. We can find the employee's (one only) group and then find that group's (one only) room. This is a very simple retrieval (it does not involve all the complications with dates that plagued the small hostel in Example 5-2).

However, the extra relationship is not just unnecessary, it is dangerous. With two routes for the same information, we risk getting two different answers unless the data is very carefully maintained. Whenever an employee changes groups or a group shifts rooms, there will be two relationship instances to update. Without very careful updating procedures, we could end up having that Jim is in Group A, which is in Room 12, while the other route may have Jim associated directly with Room 15. Redundant information is prone to inconsistencies and should always be removed.

■ **Note** Whenever there is a closed path in a data model (as in Figure 5-7), it is worth checking carefully to ensure that none of the relationships are redundant.

Routes Providing Different Information

Not all closed paths necessarily mean redundant data. One of the routes may contain different information. Alter the problem in Example 5-3 slightly to allow an employee to work for more than one of the small project groups. This is shown in Figure 5-8. Can you deduce which room an employee is in now?

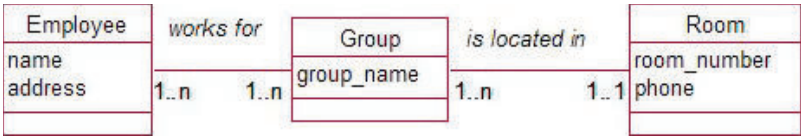

Figure 5-8. *Employees working for more than one project group*

In the model in Figure 5-8, there is no certain clear route between an employee and a particular room. For example, Group A may be in Room 12, Group B in Room 16, and Jim may work for both groups. Thus, Jim could be in either Room 12 or Room 16. Just narrowing the possibilities like that may be all the problem requires. If, however, each employee has a home room and we wish to record that information, we will need an additional relationship between employee and room as in Figure 5-9.

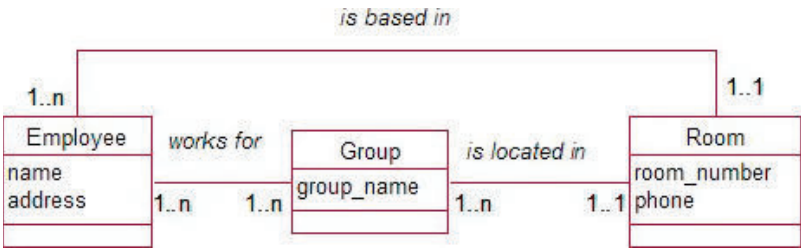

Figure 5-9. *Different routes are providing different information.*

It might seem that we have introduced another path that will give different answers to a question such as, "What room is Jim in?" Figure 5-9 allows us to have Jim based in a room different from that of any of the groups he works for. For real–life problems, this may be exactly what is required. The size of a room and the number of employees in a group are unlikely to always match. The important thing is to ensure that two routes do not contain what should be identical information so we do not introduce avoidable inconsistencies.

False Information from a Route (Fan Trap)

Not being able to deduce an employee's room from Figure 5-8 is an example of a more general problem. Take a look at Example 5-4.

EXAMPLE 5-4: LARGER ORGANIZATION

An organization has several divisions. Each of these divisions has many employees and is broken down into a number of groups. We might model this as in Figure 5-10. Have a look at the model. What can we deduce about which group or groups a particular employee is associated with?

Figure 5-10. *One (dangerous) way to model an organization*

Figure 5-10 represents a very common problem often referred to as a *fan trap*. The danger here is to take a route between employee and group and infer something that was not intended. Figure 5-11 shows some possible objects consistent with the model in Figure 5-10.

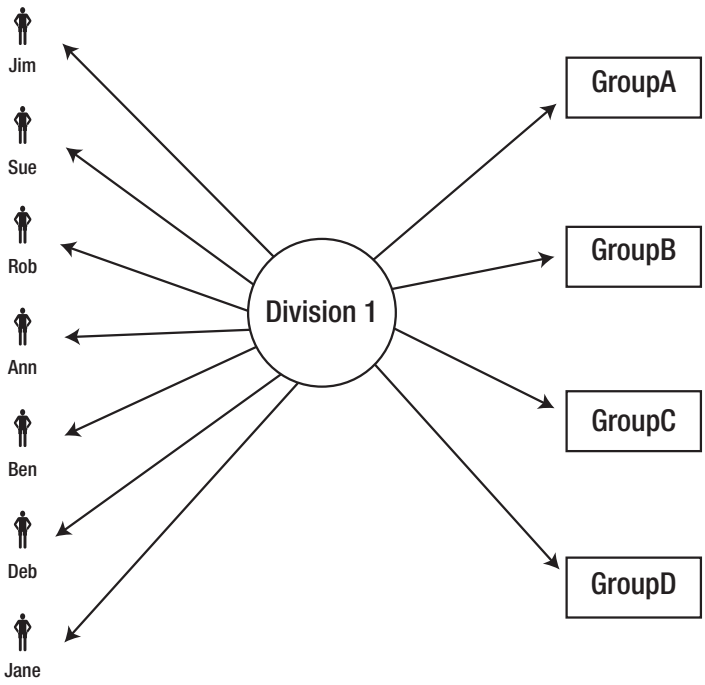

Figure 5-11. *A fan trap*

Consider employees Jim and Sue. It is not possible to infer anything about which groups Jim or Sue work for. It is only possible to get many combinations of a **`Group`** object and an **`Employee`** object that have a **`Division`** in common: Jim A, Jim B, Sue B, Jane D, and so on.[2] We must not mistake these combinations for the information we require—for example, which group or groups does Jane belong to?

The feature that alerts us to a fan trap is a class with two relationships with a Many cardinality at the outside ends. This leads to the fan shape in Figure 5-11.

[2] This situation is sometimes also referred to as a *lossy join*.

What can we do about it? If it is important for our system to be able to show the groups for which an employee works, we will need another relationship between **`Group`** and **`Employee`**, or we may need to model the problem quite differently (as shown in the next section).

Gaps in a Route Between Classes (Chasm Trap)

We might choose to model the relationships between divisions, groups, and employees in a hierarchical way as in Figure 5-12 (i.e., a division has groups and groups have employees). The optionality at one end of the employee–group relationship has not been specified. Think about the different possibilities.

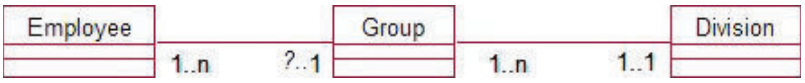

Figure 5-12. *Another way of modeling an organization*

Figure 5-13 shows some example objects. We have a direct connection between an employee and a single group (Jim works for Group A) and another between a group and its one division (Group A is in Division 1). We can therefore make a confident and unique connection between Jim and Division 1.

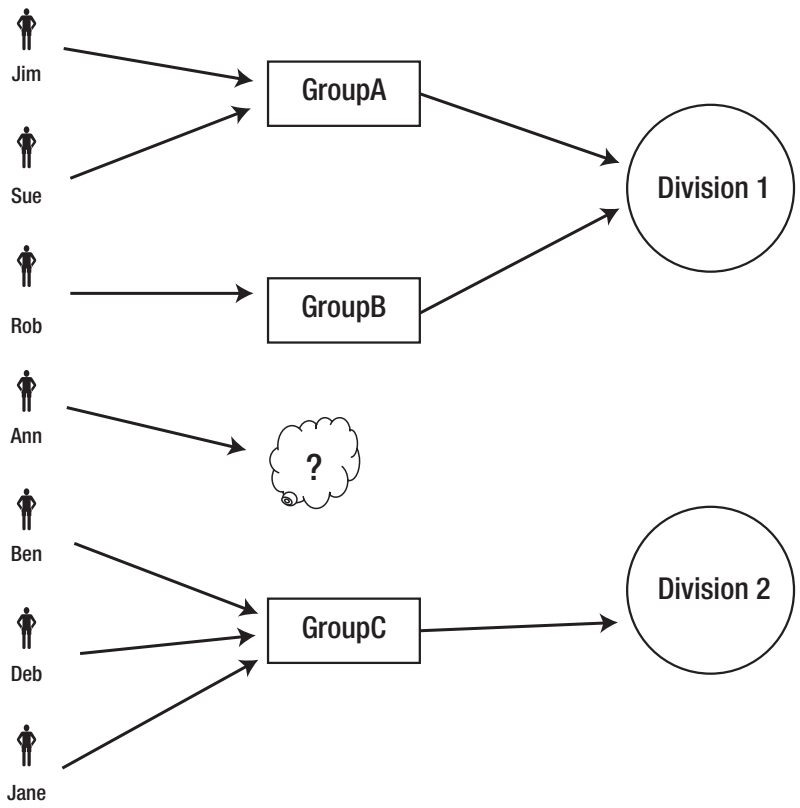

Figure 5-13. *A chasm trap*

So far, so good. However, in situations such as this, it is always useful to check that that connection is always there. What if Ann is not attached to a specific group? Maybe she is a general administrator for Division 1 and serves all groups. If an employee does not necessarily belong to a group, the model in Figure 5-12 does not provide a link between Ann and her division. To find the appropriate **`Division`** object, we need to know the **`Group`**, and Ann has no related **`Group`** object. If we need to know this information, we have a problem. This is sometimes referred to as a *chasm trap* (we can't get there from here).

This is yet another case where careful study of the data model provides quite interesting questions about the problem. For a model such as the one in Figure 5-12, we should always check for the exceptional case of an employee who may not be attached to any group.

How we solve the problem of a chasm trap depends on the situation we are modeling. One possibility is to add another relationship between division and employee so we can always make that connection. However, this extra relationship is going to cause redundant information. For many employees, we will have two routes for connecting them with a division: directly and via their group. This is the situation we had in Example 5-3 and can lead to inconsistent results for connecting employees and divisions. This is not recommended.

A different way to get around the problem in Example 5-4 is to introduce another group object (administration or ancillary staff). Ann could belong to this group, and we can then insist that every employee *must* be in a group. However, it may be that the problem needs to be remodeled entirely. It is often best to go back to the use cases and reconsider what information is the most important for the problem. It is never possible to capture every detail in a project with finite resources, so pragmatism becomes very important.

Relationships Between Objects of the Same Class

Let's return to our sports club from Example 5-1. Many clubs require a new member to be introduced or sponsored by an existing member. If it is necessary to store sponsorship information, a first attempt at a data model might be as shown in Figure 5-14.

Figure 5-14. *Modeling members and sponsors (not correct)*

The problem with the model in Figure 5-14 is that (by definition) a sponsor is a member. The model will mean that if Jim sponsors a new member of the club, we will be storing two objects for him (one in the **`Member`** class and one in the **`Sponsor`** class), both probably containing the same information (until it inevitably becomes inconsistent). What is really happening here is that members sponsor each other. This can be represented by a *self relationship* as shown in Figure 5-15.

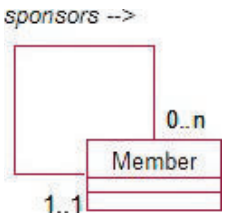

Figure 5-15. *Members sponsor other members*

The relationship in Figure 5-15 is read exactly the same as a relationship between two different classes. Reading clockwise, we have *a particular member may sponsor many members*, while counterclockwise we have *a particular member is sponsored by exactly one member*. As with all relationships, we have to change the verb depending on the direction (i.e., *sponsors* and *is sponsored by*). I have annotated Figure 5-15 to dispel any confusion about which way around we are going.

Nothing in this data model prevents members from sponsoring themselves. Such constraints need to be noted, most usefully by mentioning them in the appropriate use case (e.g., adding a member).

Self relationships appear in many situations. This is certainly true for data pertaining to genealogy or animal breeding. Consider the case in Figure 5-16 in which we record information about animals and their mothers (I am only leaving out fathers to keep the example simple!).

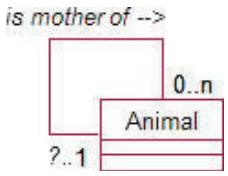

Figure 5-16. *Genealogical data about animals*

Reading clockwise, we have that *one animal may be the mother of several other animals*, and counterclockwise, that *each animal has at most one mother*. Why not *exactly* one mother? Every animal has to have a mum, does it not? This is where we have to be quite sure about the definitions of our classes. The class **`Animal`** represents those animals about which we are keeping data, not all animals. If we trace back the ancestry of a pure–bred dog for example, we may find his mother in our database, as well as her mother, but eventually we will come to a blank. You might argue that the additional generations should be added for completeness, but this could mean tracing back to the primeval slime. Our data model does not say that *some animals do not have mothers*, but merely that *some animals do not have mothers that are recorded in our database*.

As an aside, note that our **`Animal`** class will presumably contain animals of both sexes. Clearly if we establish an *is mother of* relationship between two **`Animal`** objects, the mother must be female. As it stands, there is nothing in the model to prevent male animals being recorded as mothers. This constraint could be expressed in the use case, but if this is a serious genealogical database, we may wish to treat males and females slightly differently. We will discuss ways to use techniques called *generalization* and *specialization* for situations such as this in Chapter 6.

Relationships Involving More Than Two Classes

In the examples so far, the information in which we have been interested generally relates to relationships involving two classes (e.g., which members are on which team, or which employee is in which group). Sometimes we have data that depend on objects of more than two classes. Let's reconsider a sports club. As well as keeping data about members and their current team, we might also want to keep information about games or matches between teams. Ignoring, for now, complications such as byes,[3] we can say that exactly two teams play in a match. A possible data model is shown in Figure 5-17.

[3] A *bye* sometimes occurs in a competition with an odd number of teams.

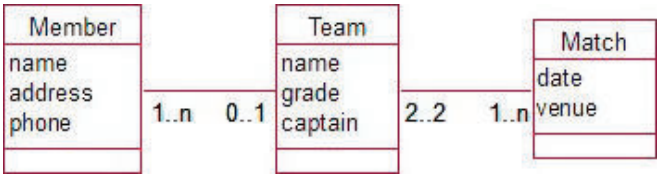

Figure 5-17. *Possible data model for members, teams, and matches*

The model in Figure 5-17 allows us to record a player's current or main team, the current members of a particular team, and the matches in which teams are involved. However, we cannot deduce that a particular player played in any given match (he may have been sick or injured). This is an example of the fan trap described earlier in the chapter. A team has many players and is involved in many matches, but we cannot say any more about which players were involved in particular matches. We could attempt to address this by adding a relationship between **`Member`** and **`Match`** as shown in Figure 5-18. Look carefully at the new data model. Can you see where there is a possibility that the data might become inconsistent?

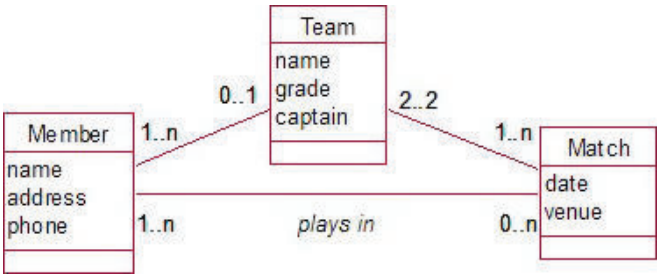

Figure 5-18. *Another model to represent members, teams, and matches*

From Figure 5-18, it is possible to have the following relationship instances:

- John plays for Team A.
- John plays in the match on Tuesday.
- The match on Tuesday is between Teams B and C.

If John plays for only one team (as the model indicates), then something weird is going on here.

Let's think through this problem with members, teams, and matches a bit further. If we want to keep track of who plays in which matches, our problem has some intricacies that the model does not adequately represent. If we are allowing for people being injured and not taking part, we also need to account for the situation in which someone from another team may need to fill in as a replacement. For example, John normally plays for Team A, but filled in for Team B on Tuesday because Scott was injured. Our scenario in the previous paragraph is not so weird—just a bit more complicated than we originally thought.

We still have a problem, however. We are happy that John normally plays for Team A and that he just happened to play in the match between Team B and Team C on Tuesday, but the model doesn't tell us which team he was playing for.

We need to step back, revisit the use cases, and figure out exactly what it is we want to know. If we want to know exactly which players played on each team for each match, then no combination of the relationships in Figure 5-18 will tell us that. The crucial point is that who played for which team in which match requires simultaneous knowledge of objects from three classes: which **`Member`**, which **`Team`**, and which **`Match`**. This is sometimes referred to as a *ternary relationship* (and, similarly, quaternary for four classes and so on).

When we have a case where the information we need requires simultaneous information from objects of three (or more) classes, we introduce a new class connected to all three classes as shown in Figure 5-19.

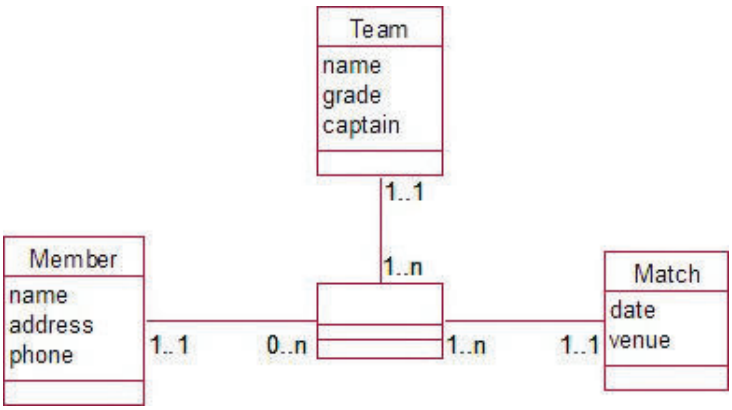

Figure 5-19. *Members and the team for which they played in a particular match*

We might be able to think of an appropriate name for this class; in this case **`Appearance`** would be sensible. If not, concatenating the other three class names will suffice (e.g., **`Team/Member/Match`**). This is not unlike introducing a new class in the Many–Many relationships we considered in Chapter 4. As in that case, the cardinality at each of the outer classes is 1.

Reading this model, we have something like this: each appearance involves one member, one team, and one match (e.g., Jim appeared for Team A in the match on Saturday the 12th); each member may have many appearances (Jim can appear in many matches, possibly for different teams); a team will have many players appearing in different matches; and a match will have many players appearing for each of the teams.

The new class may or may not have attributes. It may just be a holding place for valid combinations of **`Member`**, **`Team`**, and **`Match`** objects. If there are attributes for the new class, they must be something that involves all three classes. For example, what do we need to know about a particular player playing for a particular team in a particular match? Possibly the position. If we wanted to know that Jim played fullback for Team A in the match on Saturday the 12th, our new class is the place to record that information.

Figure 5-19 clearly has additional information that is impossible to deduce from Figure 5-18. What about vice versa? Can we re–create all the information in Figure 5-18 from 5-19? By looking at Figure 5-19, we can deduce all the teams a player played for, all the matches in which a player played, and the teams involved in each match. We do not need to add extra relationships between each pair of classes to figure out this information. In fact, it would be dangerous to do so as we would then have two routes for finding a piece of information and, as we have seen, that redundancy can lead to inconsistencies. However, there may be other information about each pair of classes that we would like to keep. For instance, in Figure 5-19 we know all the teams Jim played for but we don't know the team with which he regularly trains. Some binary relationships between each of the three classes may be required in addition to the relationships with the new class.

Whenever we have a pattern such as that in Figure 5-19, we should check whether the other binary relationships are necessary. If we have classes A, B, and C connected to a third class, for each pair of classes we should ask a question like, "Is there something I need to know about a relationship between A and B that is

independent of C?" For the preceding example, we could ask, "Is there something I need to know about player and team that is independent of the match?" The answer here would be, "Yes. I want to know the player's main team." We would therefore add a binary relationship to represent that information, as shown in Figure 5-20.

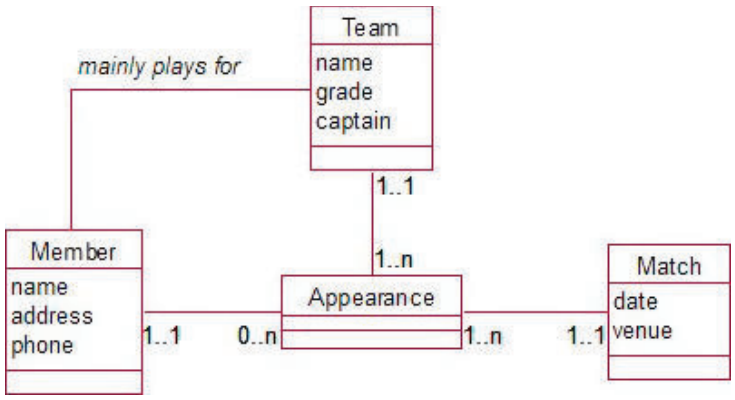

Figure 5-20. *Including a binary relationship for information independent of one class*

We need to ask a similar question for each of the other combinations; for example, "What do I need to know about a particular team and a particular match independent of the members?" The winning team maybe. "What do I need to know about a particular member and particular match independent of the teams?" Perhaps who was refereeing.

Summary

This chapter has described a miscellany of common modeling situations. Investigating these leads to a more precise understanding and representation of the real–life problem. These situations are summarized as follows:

- **Attribute, class, or relationship?**
  - Here are some examples of questions to help you decide. *Might I want to select objects based on the value of an attribute*? For example, might you want to select teams based on grade? If the answer is yes, consider introducing a class for that information (create a grade class).
  - *Am I likely now or in the future to store other data about this information?* For example, might I want to keep additional information about the captain: phone, address, and so on? If yes, consider introducing a class.
  - *Am I storing (or should I be storing) such information already?* For example, the information about a captain is the same or similar to information about members. Consider a relationship between existing classes.
- **More than one relationship between two classes:**
  - Consider more than one relationship between two classes if there is different information involving both classes. For example, a member might *play* for a team, *captain* a team, *manage* a team, and so on.

- **Consider self relationships:**
  - Objects of a class can be related to each other. For example, members *sponsor* other members, people *are parents of* other people.
- **Different routes between classes:**
  - Check wherever there is a closed loop to see whether the same information is being stored more than once.
  - Check to ensure you are not inferring more than you should from a route; that is, look out for fan traps where a class is related to two other classes and there is a cardinality of Many at both outer ends.
  - Check to ensure a path is available for all objects; that is, look out for chasm traps (are there optional relationships along the route?).
- **Information dependent on objects of more than two classes:**
  - Consider introducing a new class where you need to know about combinations of objects from three or more classes simultaneously; for example, which member played for which team in which match?
  - Any attributes in the new class must depend on a particular combination of objects from *each* of the participating classes; such as, what do I need to know about a particular *member* playing on a particular *team* in a particular *match*?
  - Consider what information might be pertinent to two objects from *pairs* of the contributing classes; for example, what do I need to know about a particular member and a particular team *independent* of any match?

TESTING YOUR UNDERSTANDING

Exercise 5-1.

The class in Figure 5-21 records information about a department and the manager's name. What other options are there for modeling information about the manager and location of a department?

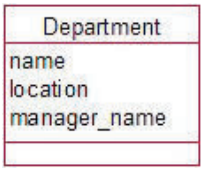

Figure 5-21. *Initial attempt at modeling the information about a department*

Exercise 5-2.

A university wants to model information about the teaching of courses. A number of staff members may contribute to providing lectures, and one staff member is denoted as the course supervisor. Suggest an initial data model.

Exercise 5-3.

How would you model information about marriages- who marries whom and when? Think about all the different situations that could eventuate (for simplicity, do not worry at this stage about the gender of the participants).

Exercise 5-4.

An orchestra keeps information about its musicians, its repertoire and concerts. A partial data model is shown in Figure 5-22. The relationships store information such as that Joe Smith is required for Saturday's concert and that Beethoven's violin sonata is to be performed at Saturday's concert.

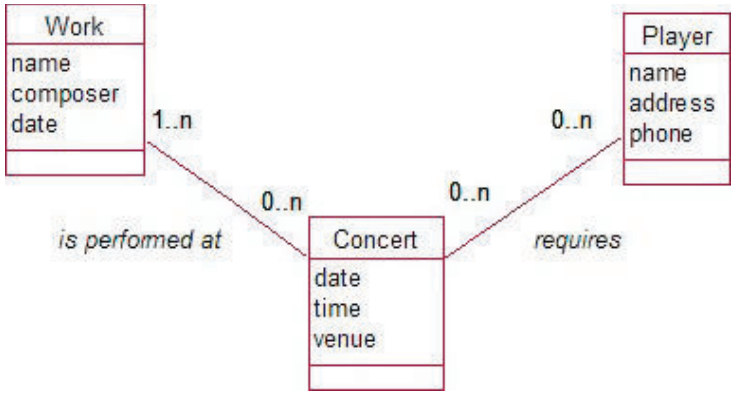

Figure 5-22. *A partial data model for an orchestra's repertoire and concerts*

What false information could be deduced from this initial model?
Amend the model so that it can maintain the following information correctly:

- Which players are involved in particular works in a concert
- The works being presented at a concert
- The fee a player receives for appearing in a particular concert