14

Normalization and Refinement

Chapters 11 through 13 walked through the steps of designing a preliminary database for The Pampered Pet. They showed how to gather requirements, build semantic object and entity-relationship models, and convert those into a relational model. Chapter 13 showed how to identify rules that should be built into the database and more complex or changeable rules that should be isolated as business rules.

Even after all of this work, the database isn't perfect. This chapter puts the finishing touches on the database by normalizing it appropriately.

In this chapter you see examples of:

❑ Improving the design to make the database more flexible.

❑ Identifying tables that are insufficiently normalized.

❑ Normalizing tables to prevent data anomalies.

❑ Not normalizing where normalization would be more trouble than it's worth.

Improving Flexibility

Figure 14-1 shows the relational design built in Chapter 13.

This design is fairly reasonable, and I've seen worse designs in working databases, but it can use a couple of improvements. Later sections in this chapter discuss normalization, but first there's a big flaw to fix.

If you think about the design long enough and you walk through the use cases, you'll notice that there's a problem with the course data. Currently the design allows many customers to take a course and it allows a course to hold many customers. However, the design allows only one instance of any given course. If you run a Puppy Socialization course in April, you cannot run the same course again in May because the course's dates would have already passed.

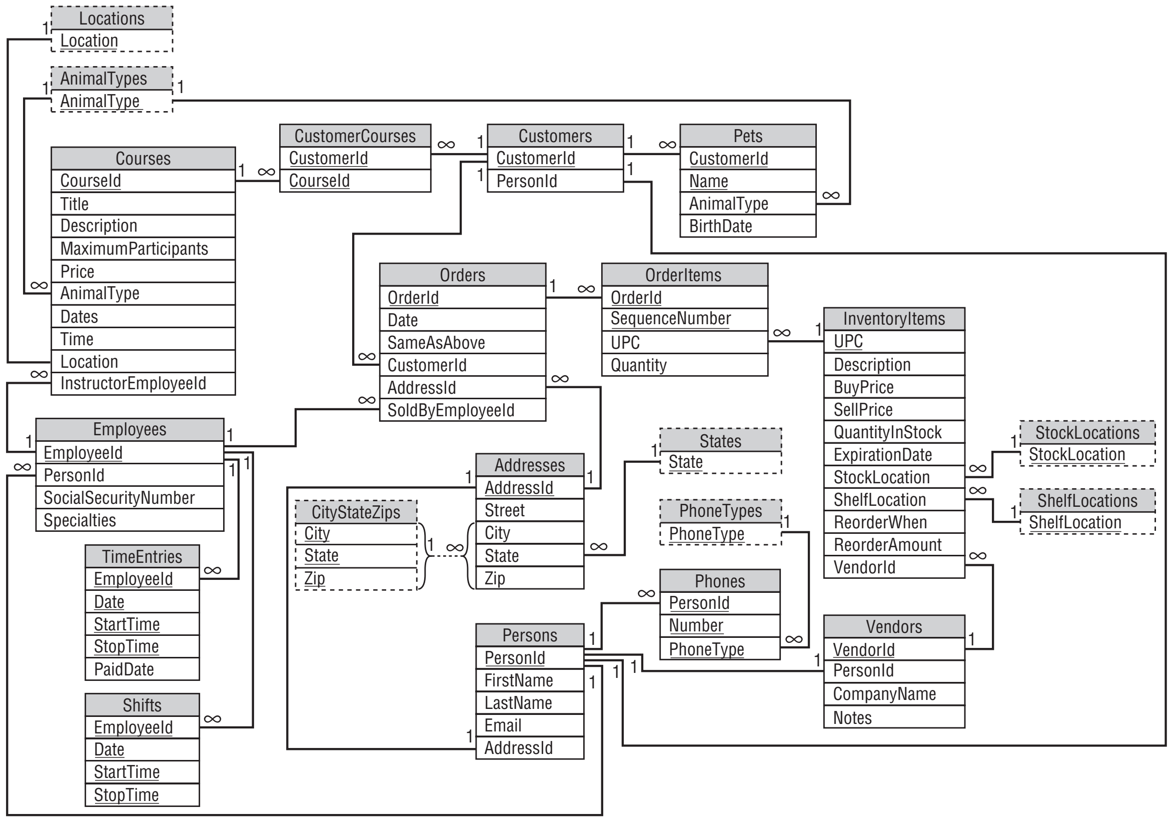

Figure 14-1

Furthermore, the same customer couldn't take both the April and May courses (some dogs are slow learners) because that would require identical records in the CustomerCourses table.

Instead you would need to make a completely new Courses record for the May course. That wouldn't be the end of the world, and some applications would do exactly that, but now the table contains multiple records that really represent different offerings of the same thing. You can tell that there's a problem because the records would have so many duplicated fields: Title, Description, Price, and AnimalType. The fields that would change between offerings are MaximumParticipants, Dates, Time, Location, and InstructorEmployeeId.

The customers taking the course would also change (except for those who fail the first time and retake the course), so the course offering would need some kind of new ID value to link it to the CustomerCourses table.

The database needs a new type of object to represent a course offering. The course offerings will link to a Courses record that provides all of their shared data.

One course can have many offerings, but an offering corresponds to only one course, so this is a one-to-many relationship, and you can add it to the database the way you always build a one-to-many relationship. First, add an ID field to the ''one'' table. Then refer to the ID field in the ''many'' table as a foreign key.

Now the CustomerCourses table should link to the new CourseOfferings table instead of the Courses table (because the customer takes an offering of a course not the abstract course description).

Figure 14-2 shows the design with the new table added.

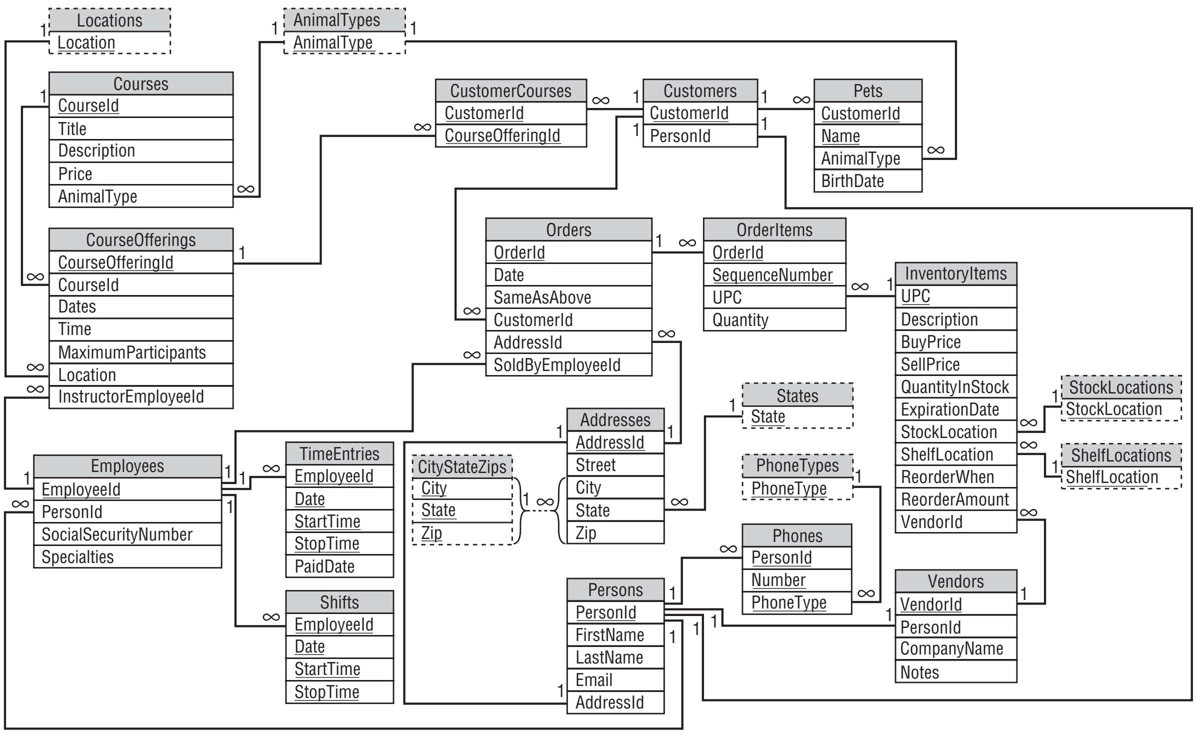

Figure 14-2

Verifying First Normal Form

The previous chapters were pretty careful about building their tables so they're almost certainly in 1NF, right? Not always. With some experience and attention to detail, you should be able to build tables in 1NF almost all of the time and 3NF most of the time, but occasionally a few normalization bugs slip through.

Recall the rules for 1NF:

1. Each column must have a unique name.
2. The order of the rows and columns doesn't matter.
3. Each column must have a single data type.
4. No two rows can contain identical values. (The table has a unique primary key.)
5. Each column must contain a single value.

You can easily verify the first four rules. For example, the Courses table's columns all have different names, the order of rows and columns doesn't matter, each column in the Courses table has a single data type, and the CourseId field is the primary key so no two records can have the same CourseId value and therefore, they are different.

The rule that usually catches people is rule 5: Each column must contain a single value. Sometimes the data in a field contains more than one logical piece of data. In that case, the field should be broken into pieces.

If you know how many pieces there will be, you can use multiple fields in the same table. For example, if you have a Name field that should be broken into FirstName and LastName, you know there are only two pieces to the field and you can just replace it with two new fields.

If the field's data could contain any number of values, you should move the values into a new table and link back to the original table. For example, suppose the Customers table contains a Children field that lists the customer's children's names separated by commas. In that case you can't just add a bunch of Child fields to the Customers table. Instead you need to add a CustomerChildren table that uses CustomerId to find the Customer associated with a particular record.

If you look carefully at each of the fields shown in Figure 14-2, you'll find a couple that might contain multiple data values. You can ignore simple compound values such as the Pets table's BirthDate field and the Phones table's Number field. Though you can think of a birth date as containing a day, month, and year, The Pampered Pet will probably never need to look at those values separately. I suppose someone might want to make a list of all pets born on the 13th of every month, but that would be pretty strange. Similarly, unless the requirements call for you to be able to list customers in a given area code, it isn't worth breaking up the phone number field.

The first field that truly holds more than one value is the CourseOfferings table's Dates field. This single field is supposed to hold a list of the dates when a course offering takes place. For example, a particular offering might occur every Wednesday for six weeks.

If every course is only offered on a weekly schedule, such as every Wednesday, you could encode that in a single field by simply giving the day of the week. If there might be exceptions (for example, six Mondays, skipping Labor Day), that system doesn't work as well.

To solve this problem, you can break the Dates field into pieces. If The Pampered Pet doesn't require every course to have the same number of sessions, you need to move the values into a new CourseOfferingDates table that refers back to its course offering.

Figure 14-3 shows the design with the new CourseOfferingDates table.

One hint that a field might contain multiple values is that its name is plural. If the field represents a number, such as the CourseOfferings table's MaximumParticipants field, it probably represents a single value. If the field represents a group of values, however, it should probably be broken apart.

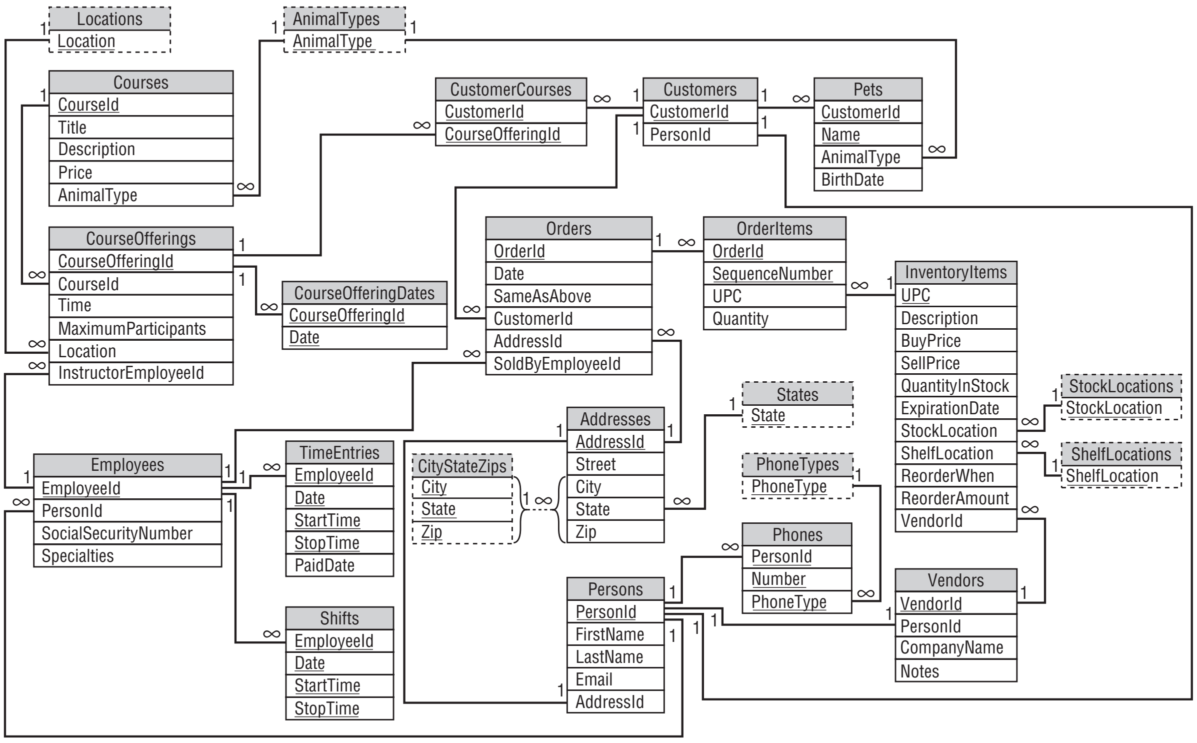

Figure 14-3

Try It Out Normal

There's one other table in Figure 14-3 that isn't in 1NF. Find and fix it.

1. Look for fields that don't represent a single value.
2. Decide whether the field contains a fixed, known number of values or an unknown number of values. If the field contains a fixed number of values, split it into the required number of fields. If the field contains an unknown number of values, move the values into a new table.
3. Perform other modifications to the design if this change requires them. (Hint: In this case, you'll need to create a new lookup table.)

How It Works

1. All of the fields shown in Figure 14-3 represent a single value except the Employees table's Specialties field. This field lists the employee's areas of expertise. They might include animal types, products, problems, and so forth. A typical value might be, ''Cat, Dog, Parasites.'' (This is also the only field in Figure 14-3 that has a plural name other than MaximumParticipants, which represents a single number.)

2. The Specialties field could contain any number of values so it cannot be broken into new fields within the Employees table. Instead the model needs a new Specialties table that refers back to Employees. The new table will have fields EmployeeId and Specialty.
3. The new table's Specialty field can contain only specific values, such as Cat, Dog, and Penguin, so the model should validate the field by making it a foreign key that refers to a lookup table. In this case, the new table should be called Specialties and will have a single field Specialty. Figure 14-4 shows the new design.

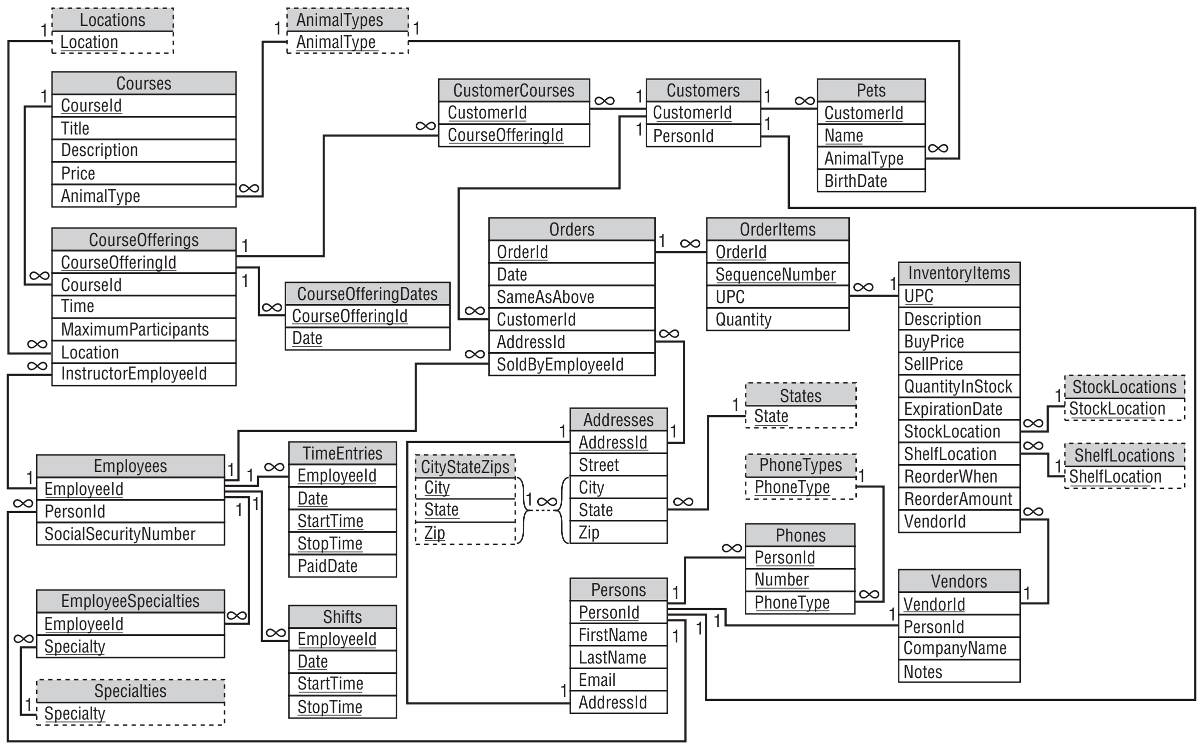

Figure 14-4

Verifying Second Normal Form

Recall the rules for 2NF:

1. The table is in 1NF.
2. All of the non-key fields depend on all of the key fields.

Many of the tables have a one-field primary key, so every other field must depend on the entire primary key. In the intermediate tables representing many-to-many relationships and the lookup tables, every field is part of the primary key, so rule 2 doesn't apply.

The only tables remaining to consider are Pets, OrderItems, and TimeEntries.

Pets

The Pets table's primary key contains the combination CustomerId/Name. Its other fields are AnimalType and BirthDate. AnimalType depends on both CustomerId and Name because you need to know both CustomerId and Name to deduce the pet's AnimalType.

Another way to think of this is to notice that if you know the CustomerId alone, you cannot guess the pet's AnimalType because the customer might have a cat and a fish. Similarly, if you know the pet's Name, you cannot determine the AnimalType because different customers might have different kinds of pets with the same name. You can use similar arguments to show that BirthDate depends on both CustomerId and Name.

A third way to look at this, which may be less intuitive but which is easy to apply mechanically, is to ask yourself whether the database could contain any record with the same CustomerId, a different Name, and a different AnimalType. If CustomerId alone determines animal type, then every record with the same CustomerId must have the same AnimalType.

In this case, the database could hold a record with the same CustomerId, a different Name, and a different AnimalType (someone could own a fish named Phred and a dog named Pheidaux) so you know that AnimalType depends on CustomerId.

Similarly, you can ask whether another record could have the same Name, different CustomerId, and different AnimalType. I have a dog named Snortimer and I've met two other people with cats named Snortimer, so that situation is possible. That means AnimalType also depends on CustomerId.

You can use similar arguments to show that BirthDate depends on both CustomerId and Name.

(This all assumes a customer doesn't give multiple pets the same name. There's a silly Sandra Boynton song named ''Fifteen Animals'' about a guy who has 15 pets all named Bob, except his turtle, which he named Simon James Alexander Ragsdale III. For this customer, you'll probably have to assign the pets serial numbers or something: Bob-1, Bob-2, Bob-3, and so forth.)

Try It Out OrderItems

Verify that the OrderItems table is in 2NF. For each of the table's primary keys (OrderId and SequenceNumber) and each of the non-key fields (UPC and Quantity), see if another record could have a different value for the key field and a different value for the non-key field. Consider all four combinations:

1. OrderId and UPC
2. OrderId and Quantity
3. SequenceNumber and UPC
4. SequenceNumber and Quantity

How It Works

1. Could there be two records with the same SequenceNumber, different OrderId, and different UPC? Yes. Two orders could contain different items listed first. Then the OrderItems records will

have the same SequenceNumber (because the items are listed first), different OrderId (because they're two separate orders), and different UPC (because the orders are for different things).

2. Could there be two records with the same SequenceNumber, different OrderId, and different Quantity? Yes. Two orders could contain different quantities of the same item. For example, one customer could order one toy mouse and a second customer could order two toy mice. Then the OrderItem records will have the same SequenceNumber (because they are the first items for each order), different OrderId (because they are different orders), and different Quantity (because the first customer ordered one toy mouse and the second customer ordered two).
3. Could there be two records with the same OrderId, different SequenceNumber, and different UPC? Yes. Suppose an order contains two different items. Then the OrderItems records will have the same OrderId (because they're part of the same order), different SequenceNumber (because they're different line items in the order), and different UPC (because the two items are different).
4. Could there be two records with the same OrderId, different SequenceNumber, and different Quantity? Yes. Suppose an order contains two items with different quantities (for example, one hamster wheel and two rawhide bones). Then the OrderItems table will contain two records for this order with the same OrderId, different SequenceNumber (1 and 2), and different Quantity (1 and 2).

Because the table can hold all of these combinations, all of the non-key fields depend on every primary key field, so the table is in 2NF.

TimeEntries

The TimeEntries table's primary key includes the fields EmployeeId, Date, StartTime, and StopTime. Its only remaining field is PaidDate. To see that PaidDate depends on all of the primary key fields, ask yourself whether you could deduce the PaidDate value if you are missing one of the key values.

If EmployeeId is missing, another employee might have worked the same shift and may or may not be paid.

If Date is missing, the employee might have worked similar hours on another date and may or may not have been paid.

The StartTime and StopTime fields are a bit trickier. If StartTime is missing, the table could contain another record for the same employee on this date with the same StopTime but a different StartTime. However, the business rules require that this table cannot have two records for the same employee on the same date with overlapping times, so this record would not be allowed.

Does that mean the PaidDate field doesn't depend on StopTime, so the table isn't in 2NF? Not really. Though the table isn't allowed to hold a record with overlapping times, the table's structure doesn't prevent it; a business rule does.

To see this in another way, consider the wrestling match schedule described in the ''Second Normal Form (2NF)'' section of Chapter 7. The following table shows part of a schedule of matches. The table's primary key is the Time/Wrestler combination.

Time	Wrestler	Class	Rank
1:30	Annette Cart	Pro	3
2:00	Sydney Dart	Amateur	1
3:45	Annette Cart	Pro	3

The problem with this table is that Class and Rank depend only on Wrestler and not on Time. Annette Cart is ranked 3rd professionally no matter when she wrestles. That makes the table vulnerable to data anomalies. For example, if you change the first entry's Class to Amateur, it contradicts the third entry.

Now consider again the TimeEntries table. Because of the ''no overlapping time'' business rule, this table cannot hold two records with the same EmployeeId, Date, and StopTime. It cannot hold two records that correspond to the two wrestling schedule records for Annette Cart and that means it cannot suffer from the same kind of modification anomaly as the wrestling schedule table.

Similarly, the table is safe from update anomalies because you cannot add two records with the same EmployeeId, Date, and StopTime.

Verifying Third Normal Form

Recall the rules for 3NF:

1. The table is in 2NF.
2. It contains no transitive dependencies.

A *transitive dependency* is when one non-key field's value depends on another non-key field's value. It takes a bit more work to find transitive dependencies than it does to detect other errors.

Because transitive dependencies occur when two non-key fields are related, you only need to consider tables that have at least two non-key fields. In this example, those tables are Courses, CourseOfferings, Pets, Orders, OrderItems, InventoryItems, Employees, Vendors, Addresses, and Persons.

Most of these tables are easy to check. For example, consider OrderItems. Its non-key fields are UPC and Quantity. Are these fields related? Of course not. The type of item a customer is buying does not determine the number of items bought or vice versa. (If the store has only one Jack Russell Terrier, you can only buy one, but that's an inventory issue, not a database design issue. Of course Jack Russells are so energetic it might be insane to buy more than one but again, that's not a database design issue.)

In the Courses table, the Title, Description, Price, and AnimalType fields don't depend on each other. If the business requirements stated that all Dog courses had the same price, then things would be different, but in this example there are no such restrictions.

The Orders table might give you pause because the CustomerId might be related in some sense to the AddressId. Remember that an Orders record has an AddressId field only if the customer wants the order shipped to an address other than the customer's usual address, however, so the relationship is not really there. If the order included the customer's home address every time, there would be a relationship between the CustomerId and the AddressId.

The Addresses table also contains the standard weird relationship between City, State, and Zip. Chapter 13 already considered this relationship (see the section ''Try It Out, Address Constraints'') and decided to live with a lookup table for local addresses rather than building an enormous lookup table for every City/State/Zip combination.

The last tricky table is CourseOfferings. The maximum number of participants for a course is determined by the location where it is taught. For example, the store's back conference room can only hold 20 people so that's the maximum size for any course taught there.

This transitive dependency means that any course in a particular location will have the same MaximumParticipants value. To remove this dependency, you should create a new table that lists the MaximumParticipants value for each location and then remove MaximumParticipants from the CourseOfferings table. However, the design already contains a Locations lookup table that holds location names. You can use that table if you add the MaximumParticipants field to it.

Note that it's not always a good idea to combine tables in this manner. In this case, however, the new version of the table holds data for a single, clearly defined purpose: to describe locations. Because the table's fields both fit this purpose, it makes sense to put them in the same table.

Figure 14-5 shows the new design. The new version of the Locations table is no longer simply a lookup table, so it's not drawn with a dashed rectangle in Figure 14-5.

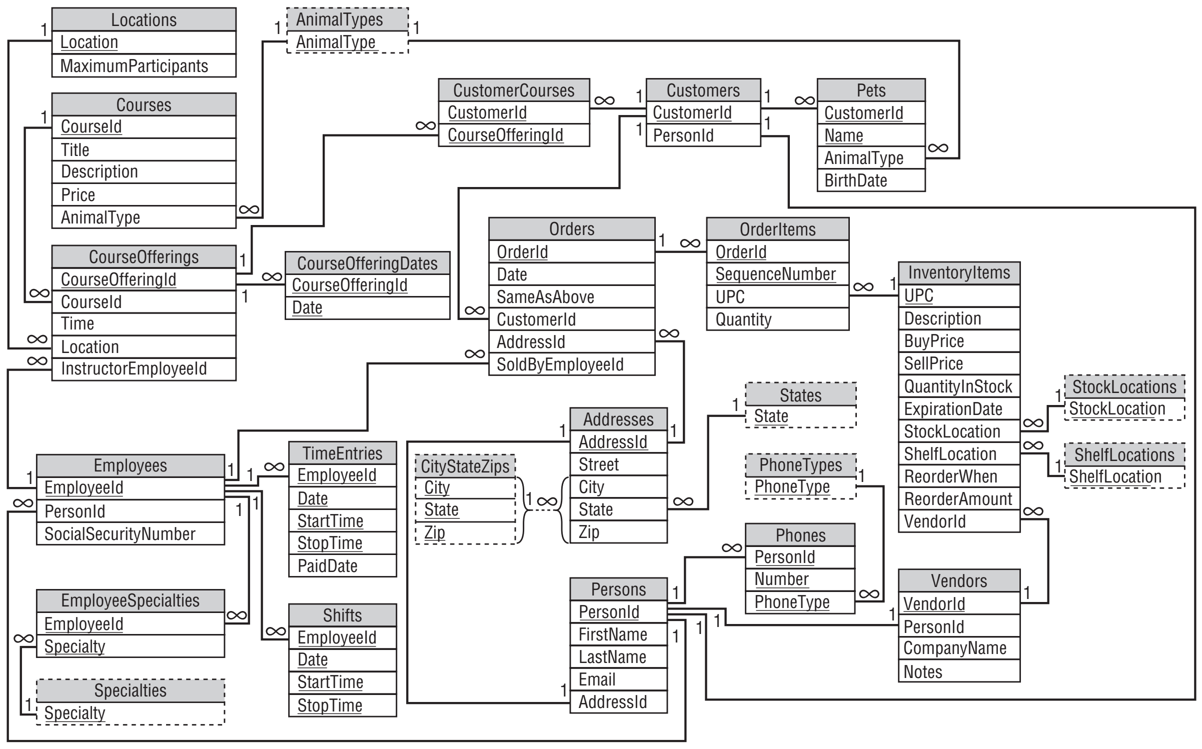

Figure 14-5

Notice that the Locations table acts as a lookup table for the CourseOfferings table's Location field. The Locations table is not only a lookup table, however, so it's not drawn with a dashed rectangle. You could remove the pure lookup tables from the database and the database would still function, although you would need to implement some field-level check constraints. If you removed the Locations table, you would lose all of the MaximumParticipants data.

Note that the business rules could have indicated that different courses using a particular location might be able to have different numbers of participants. For example, a seminar on piranha feeding takes less room than a hands-on elephant training workshop, so more people will fit in the room. You could model that situation by making the Locations table use Location and AnimalType as its primary key, but this example seems complicated enough already.

Summary

This chapter refined The Pampered Pet database to increase its flexibility. It also normalized the database's tables to make the database more resistant to data anomalies.

This chapter showed examples of:

- Increasing the database's flexibility by allowing multiple offerings of a particular course.
- Putting the CourseOfferings table into 1NF by moving course dates into a new CourseOffering-Dates table.
- Putting the Employees table into 1NF by moving employee specialty information into a new EmployeeSpecialties table.
- Putting the CourseOfferings table into 3NF by moving information about the maximum number of participants at a location into a new Locations table.

(Of course you may have seen these problems earlier and been muttering under your breath about how silly the design was for the last few chapters, but sometimes these problems sneak through to the bitter end.)

At this point, the database is in pretty good shape and you should be able to build it with some confidence that it can successfully fend off some serious data anomalies.

However, Figure 14-5 doesn't show the complete picture. It shows the table structures and lookup tables, but it doesn't show the many additional constraints that were identified in Chapters 11 and 13. Those must be implemented as field- and table-level check constraints.

The following chapters show how to build this database in the Access and MySQL relational database management systems. They show how to build the tables shown in Figure 14-5 and how to provide the necessary check constraints to really make the database robust.

Before you move on to Chapter 15, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

Exercises

For these exercises, consider the following aquarium show schedule. The show times with asterisks match with the shows with asterisks. For example, the 11:15 show in Sherman's Lagoon is ''Sherm's Shark Show'' and the 1:15 show is ''Meet the Rays.'' (Yeah, I think it's a strange way to list shows, too, but I saw a real schedule very similar to this one recently.)

Show	Venue	Seating	Times
Sherm's Shark Show / Meet the Rays*	Sherman's Lagoon	375	11:15, 1:15*, 3:00, 6:00*
Deb's Daring Dolphins / The Walter Walrus Comedy Hour*	Peet Amphitheater	300	11:00, 12:00, 2:00*, 5:27*, 6:30
Flamingo Follies / Wonderful Waterfowl*	Ngorongoro Wash	413	2:00, 3:00*

1. Explain why this table isn't in 1NF. Make a relational design that uses one table in 1NF. Show the data in the new table.
2. Explain why the solution to Exercise 1 isn't in 2NF. Make a relational design that fixes it. Show the data in the new tables.
3. Explain why the solution to Exercise 2 isn't in 3NF. Make a relational design that fixes it. Show the data in the new tables.
4. If you made the fewest changes possible while converting the original table into 1NF, 2NF, and 3NF, the new tables probably use show name, time, and venue name as primary keys. That bodes ill if you need to change a show's name (Pete Penguin holds out for equal billing in The Walter Walrus Comedy Hour), a time, or a venue's name (the Trustees decide to sell naming rights and change the name of ''Peet Amphitheater'' to ''Pampered Pet Cove'').

   Modify the design to make those kinds of changes easier. Show the data in the new tables.