4

Understanding User Needs

The previous chapters discussed databases in general terms. Chapters 1 and 2 explained the goals of database design and described some of the types of databases that are available. Chapter 3 described the most common type of database, relational databases, in slightly greater detail. With this basic understanding of databases, you're ready to take the first step in designing an actual database to solve a particular problem: understanding the user's needs.

Designing any custom product, whether it's a database, beach house, or case mod (see `www.neatorama.com/case-mod/index.php` for some amazing examples), is largely a translation process. You need to translate the customers' needs, wants, and desires from the sometimes fuzzy ideas floating around in their heads into a product that meets the customers' needs.

The first step in the translation process is understanding the user's requirements. Unless you know what the user needs, you cannot build it. Designing the best order processing database imaginable won't do you a bit of good if the customer really wants a circuit design database or an ostrich race handicapping system.

Just as the database design forms the foundation upon which the rest of the application's development stands, your understanding of the user's needs form the foundation of the database design. If you don't know what the user needs, how can you possibly design it?

If you don't understand the customer's needs thoroughly and completely, you may as well pack it in now. There's little satisfaction in wasting months of your life and a pile of your company's money to build something unusable. Make sure you're on the right road before you stomp on the accelerator and burn rubber down a dead-end alley.

This chapter explains techniques that you can use to learn about the customer's needs. It describes methods that you can use to record those needs in a concrete and verifiable way.

The sections that follow describe some of the steps you can take to better understand the customers' needs. In some projects, you may not need to follow all of these steps. For example, if your customer is a single person with very concrete ideas about what needs to be done, you may not need to spend much time learning who's who or brainstorming. If your customer works with government classified data, you may not be allowed to ''walk a mile in the user's shoes'' and you may have access to only some of the business's documentation.

I once knew a developer who was working on a classified project. He had clearance to see the source code but not the data, so every week his customer brought him a giant printout of the latest run with all of the data carefully clipped out with scissors. He would try to guess what was going on and make some suggestions so the customer's developers could try to fix the code. Then the cycle repeated the next week. What an odd way to work!

In other projects, the steps may work best in a different order. You may find it better to brainstorm before visiting the customers' site and watching them work.

These are just steps that I've found most useful in trying to understand the customers' situation. You'll have to adjust them as necessary to fit each of your projects.

In this chapter, you learn how to:

- ❑ Understand the customers' needs and motivations.
- ❑ Gather and document user requirements.
- ❑ Cull requirements from existing practices and information.
- ❑ Build use cases to understand the user's needs and to measure success or failure.
- ❑ Anticipate changes and future needs to build the most flexible database possible.

After you master these techniques, you'll be ready to move on to the next step and actually start designing the database.

Make a Plan

Though the steps described in this chapter sometimes occur in different orders, the following list summarizes the order that's most typical. Feel free to add, remove, and rearrange them as necessary.

- ❑ Bring a List of Questions
- ❑ Meet the Customers
- ❑ Learn Who's Who
- ❑ Pick the Customers' Brains
- ❑ Walk a Mile in the User's Shoes
- ❑ Study Current Operations
- ❑ Brainstorm
- ❑ Look to the Future
- ❑ Understand the Customers' Reasoning
- ❑ Learn What the Customers Really Need
- ❑ Prioritize
- ❑ Verify Your Understanding
- ❑ Write the Requirements Document

- ❑ Make Use Cases
- ❑ Decide Feasibility

This list isn't perfect but it makes a good meta-plan — a plan for making the project's plan. (Hopefully it won't be as useless as the traditional pre-meeting agenda planning meeting.)

Bring a List of Questions

From the very first day, you should start thinking of questions to ask the customers to get a better idea of the project's goals and scope.

The following sections list some questions that you can ask your customers to help understand their needs. You'll see many of them described in greater detail later in this chapter.

This list is by no means complete — the questions that you need to ask will depend to a large extent on the type of project. Use them only as a starting point. It's helpful to have something to work from when you start, however. Then you can then wander off in promising directions as the discussions continue.

Functionality

These questions deal with what the system is supposed to accomplish and, to a lesser extent, how. It is usually best to avoid deciding how the system should do anything until you thoroughly understand what it should do so you don't become locked into one idea too early, but it's still useful to record any impressions the customers have of how the system should work.

- ❑ What should the system do?
- ❑ Why are you building this system? What do you hope it will accomplish?
- ❑ What should it look like? Sketch out the user interface.
- ❑ What response times do you need for different parts of the system? (Typically, interactive response times should be under five seconds, whereas reports and other offline activities may take longer.)
- ❑ What reports are needed?
- ❑ Do the end users need to be able to define new reports?
- ❑ Who are the players? (ties to previous section)
- ❑ Do power users and administrators need to be able to define new reports?

Data Needs

These questions help clarify the project's data needs. Knowing what data is needed will help you start defining the database's tables.

- ❑ What data is needed for the user interface?
- ❑ Where should that data come from?

- How are those pieces of data related?
- How are these tasks handled today? Where does the data come from?

Data Integrity

These questions deal with data integrity. They help you define some of the integrity constraints that you will build into the database.

- What values are allowed in which fields?
- Which fields are required? (For example, does a customer record need a phone number? A fax number? An email address? One of those but not all of them?)
- What are the valid domains (allowed values) for various fields? What phone number formats are allowed? How long can customer names be? Addresses? Do addresses need extra lines for suite or apartment number? Do addresses need to handle U.S. ZIP Codes such as 12345? ZIP+4 Codes such as 12345-6789? Canadian postal codes such as T1A 6G9? Or other countries' postal codes?
- Which fields should refer to foreign keys? (For example, an address's State field might need to be in the States table and a CustomerID field might need to be in the Customers table. I've seen customers with a big list of standard comments and a Comments field can only take those values.)
- Should the system validate cities against postal codes? (For example, should it verify that the 10005 ZIP Code is in New York City, New York? That's cool but a bit tricky and can involve a lot of data.)
- Do you need a customer record before you can place orders?
- If a customer cancels an account, do you want to delete the corresponding records or just flag them as inactive?
- What level of system reliability is needed?
  - Does the system need 24/7 access?
  - How volatile is the data? How often does it need to be backed up?
  - How disastrous will it be if the system crashes?
  - How quickly do you need to be back up and running?
  - How painful will it be if you lose some data during a crash?

Security

These questions focus on the application's security. The answers to these questions will help you decide which database product will work best (different products provide different forms of security) and what architecture to use.

- Does each user need a separate password? (Generally a good idea.)
- Do different users need access to different pieces of data? (For example, sales clerks might need to access customer credit card numbers but order fulfillment technicians probably don't.)
- Does the data need to be encrypted within the database?

- ❑ Do you need to provide audit trails recording every action taken and by whom? (For example, you can see which clerk increased the priority of a customer who was ordering the latest iPod and then ask that clerk why that happened.)
- ❑ What different classes of users will there be?

I often use three classes of users. First, clerks do most of the regular work. They enter orders, print invoices, discuss the latest Oprah around the water cooler, and so forth. Second, supervisors can do anything that clerks can and they also perform managerial tasks. They can view reports, logs, and audit trails; assign clerks to tasks; grant bonuses; and so forth. Third, super users or key users can do everything. They can reset user passwords, go directly into database tables to fix problems, change system parameters such as the states that users can pick from dropdowns, and so forth. There should only be a couple of super users and they should usually log in as supervisors, not as super users, to prevent accidental catastrophes.

- ❑ How many of each class of user will there be? Will only one person need access to the data at a time? Will there be hundreds or even thousands (as is the case with some Web applications)?
- ❑ Is there existing documentation describing the users' tasks and responsibilities?

Environment

These questions deal with the project's surrounding environment. They gather information about other systems and processes that the project will replace or with which it will interact.

- ❑ Does this system enhance or replace an existing system?
  - ❑ Is there documentation describing the existing system?
  - ❑ Does the existing system have paper forms that you can study?
  - ❑ What features in the existing system are required? Which are not?
  - ❑ What kinds of data does the existing system use? How is it stored? How are different pieces of data related?
  - ❑ Is there documentation for the existing system's data?
- ❑ Are there other systems with which this one must interact?
  - ❑ Exactly how will it interact with them?
  - ❑ Will the new project send data to existing systems? How?
  - ❑ Will the new project receive data from existing systems? How?
  - ❑ Is there documentation for those systems?
- ❑ How does your business work? (Try to understand how this project fits into the bigger picture.)

Meet the Customers

Before you can start any project, you need to know what it is about. Are you building an inventory system, a supply chain model, or a stock price tracker and predictor (also called a random number generator)?

The best way to understand the system you need to design and build is to interrogate the customers. I use the rather unfriendly word ''interrogate'' because, to do the job right, you need much more than a simple chat over tea and crumpets. Learning about the customers' requirements can be a long and grueling process. It can take days or even weeks of cross-examination, studying existing practices, poring over dusty scrolls and other corporate documentation, and spying on the customers while they do their daily jobs.

When it's over, the customers shouldn't hate you outright but they might wish you would go away and leave them alone for a while. A good question and answer session should leave everyone exhausted but with the warm glow of satisfaction that comes with moving a lot of information from their brains to yours.

Customers who are truly dedicated to the company are usually willing to field even the most obtuse questions as long as you're willing to dish them out. Benjamin Disraeli once said, ''Talk to a man about himself, and he will listen for hours.'' Most customers are more than happy to share the ins and outs of their corner of the business universe with you for as long as you can stand it.

> *It may sound boring listening to customers drone on about their supply chains but I've found that once you dig deeply enough, almost any business can be pretty interesting. I've worked on projects spanning such topics as fuel tax collection, wastewater treatment, ticket sales, and school enrollment. Every time, after I'd learned enough, I discovered hidden complexity that I would never have imagined.*

The goal isn't to torture the customers (although it may sometimes seem like it to them) but to give you an absolute and complete understanding of the problem you're attempting to solve. You want as few surprises as possible after you're done researching the problem. Unexpected difficulties and feature requests are the biggest reasons why software projects finish late, come in over budget, or fail completely.

The sooner you can identify potential problems and the more completely you can identify the system's features, the easier it will be for you to plan for them and the less they will mess up your meticulously crafted plan. Your initial encounters with the customer give you your first chance to address these issues so they don't bite you later.

So when you first start a project, meet the customers. Get to know them and what they do. Even if the problem you are trying to solve is only a small part of their business, get a feel for the overall picture. Sometimes you'll find unexpected connections that may make your job easier or that may lead to surprising benefits in a completely unrelated area.

When you first meet the customers, it usually doesn't hurt to warn them that you're going to be a major pest for a while. This can also help you figure out who's who. Those who are committed to the project and are eager to succeed will take your warning well. Those who are less than dedicated may tip their hands at this point. This idea leads naturally to the next section.

Learn Who's Who

Ideally a project team works well together, everyone does the best possible job without conflict, and the project moves along smoothly to create a finished product that meets the customers' needs. In practice, however, it doesn't always work out that way. Like the bickering superheroes in an X-Men movie, everyone has his or her own personal abilities, agenda, and motivation that don't always coincide with those of the other team members.

As you get to know the customers (and your team members), it's important to realize that not everyone shares the same vision of the product. You need to figure out which customer is the leader, which are team players, which have little or no say in specifying the project, and which will be super villains.

> *No one wants a super villain on their project, but you should be aware that they are out there. I've worked on projects where customers ostracized members of the project team, tried to delete all of our project files, spread dark rumors among senior management, and even slashed tires. Hopefully you won't encounter any of these types but it's best that you know about these people as early as possible.*

The following list describes some of the roles that customers (and developers) often play in a project. Naturally these cannot categorize everyone, but they define some characteristics that you should look for.

- **Executive Champion:** This is the highest ranking customer driving the project. Often this person doesn't participate in the project's day-to-day trials and tribulations. The Executive Champion will fight for truth, justice, and getting you that extra laptop you need. In the end, the Executive Champion must be able to take on any bored super villains or you might be in trouble.
- **Customer Champion:** This person has a thorough understanding of the customers' needs. Lesser champions may help define pieces of the project but this is the person you run to when the others are unclear. For the purposes of this chapter (''Understanding User Needs''), this is the most important person on the project. This person must have enough time and resources (also known as ''people'') to help you define the project and answer your questions. Ideally this person also has enough clout to make decisions when the heroes start bickering over who has to fight Magneto and who gets to fly the invisible plane.
- **Customer Representative:** A Customer Representative is someone assigned to answer your questions and help define the project. Often these are people who do the day-to-day work of your customers' business. Sometimes they are experts in only parts of the business so you need more than one to cover all of the issues.
- **Stakeholder:** This is anyone who has an interest in the project. Some of these fall into other categories such as Customer Champion or Customer Representative. Others are affected by the outcome but have no direct say in the design of the system. For example, front-line clerks rarely get to toss in their two cents when you design a point-of-sales system. They are like the civilians whose fate is determined by the battling superheroes and who are easily crushed by falling debris and pieces of robot monsters. Though many of them have no direct power over the outcome, you should keep them in mind and try to minimize collateral damage. (In a really well-run company, these people have their own representatives to watch out for them.)
- **Sidekick/Gopher:** This is someone who can help you get the more mundane resources you need such as conference rooms, airline tickets, donuts, and kryptonite. Though this isn't a glamorous job, an effective Sidekick can make everything run more smoothly. (Sometimes they also provide comic relief. On one project, the Sidekick invited everyone out to a huge celebratory lunch on him, only to find that the restaurant didn't take credit cards, so we all had to pitch in. In all fairness, though, it could have happened to any of us.)
- **Short-Timer:** This is someone who is only going to be around for a short while. This may be someone who is about to be promoted to a new division, who will retire soon, or who is just plain fed up and about to walk. A dedicated short-timer can be a huge asset, particularly those who are about to retire and take a lifetime's worth of experience with them. Others don't care all that much whether the project succeeds or fails after they're gone. (These are like the red-shirts

on *Star Trek* who don't contribute much. When Kirk says, ''Spock, Bones, and Smith, meet me in the transporter room,'' guess who isn't coming back?)

- **Devil's Advocate:** This is an important role for avoiding groupthink. Left unchecked, some groups become irrationally optimistic and can make extremely poor decisions. A Devil's Advocate can help bring the hopeless dreamers back to earth and keep the project realistic... as long as the Devil's Advocate doesn't get out of hand. The purpose of the Devil's Advocate is to maintain a reality check, not to defeat the entire project. If this person shoots down every idea anyone comes up with, you might gently mention that eventually you need to decide on an approach and get something done.
- **Convert:** This is someone who originally is against the project but who you convert to your cause. Strangely, both finding and converting this person are usually surprisingly easy, at least for bigger projects. If you talk to the disenfranchised stakeholders (the front-line users who have no say in the matter), you can usually find some who are dead-set against the project, if for no other reason than it represents a change from the way they have always worked. Take one of these people who has a fair amount of experience and make him a Stakeholder Representative. Get him involved early in the process and take his suggestions very seriously. If you act on some of those suggestions, you'll show that you have the Stakeholders' interests in mind and you'll win his loyalty. He'll tell the rest of the Stakeholders and, if all goes well, you'll have more support than you can imagine. And who knows, you may build a better product with this person's input.
- **Generic Bad Guy:** These range from simple defeatists and layabouts to Arch Super Villains actively trying to sabotage the project. Try to identify these people early so you know what you're up against. (On one project, we had a super villain at the Vice Presidential level. We also had an Executive Champion at the same level, so we were able to hold our own, but it was pretty tough going. It's easy to get squashed when such heavy-hitters collide.)

Don't feel constrained by this list. These are just some of the characters that I've encountered and you may meet others.

I don't mean to imply that every project is subject to continual harassment, interference, and sabotage. I've worked on lots of projects where everyone really was pulling for the common good and we achieved impressive results. Just keep your eyes open. Identify the main players as quickly as possible so you know who to ask questions and where to run when the fighting erupts.

Try It Out — Know the Players

If you're familiar with the Dilbert comic strip, think about the main characters Dilbert, Alice, Wally, Asok the Intern, and the Pointy-Haired Boss. Assume they are your customers and you need to design them a database.

Who will play which customer roles? In particular, who will be:

- Executive Champion
- Customer Representative
- Sidekick
- Bad Guy

What are your chances for success?

How It Works

Unfortunately the only candidate for Executive Champion is the Pointy-Haired Boss. He's incompetent and unable to defend against any attacks from bad guys so you're in trouble from the start.

Alice and Dilbert generally know what's going on and try to do the right thing. They will be your best bets for Customer Representatives.

Asok means well and is competent but he's new to the company and doesn't know how everything works, so he won't be the best Customer Representative. He might make a good Sidekick, however.

Wally is a serious layabout. He actively seeks to avoid work even if doing the work would be easier. He's a bad guy, although on a minor scale. He won't destroy the project single-handedly but he may waste other people's time.

Your overall chances depend entirely on whether the project will face outside attack. If any serious bad guy appears, the Pointy-Haired Boss will crumble and the project will fail.

If no one else is interested in taking over or ruining the project, you might have a chance to finish before the Pointy-Haired Boss plays too active a role and messes everything up. (But then again, how long do things run without interference in a Dilbert cartoon?)

Pick the Customers' Brains

Once you figure out more or less who the movers and shakers are, you can start picking their brains. Sit down with the Customer Champion and find out what the customers think they need. Find out what they think the solution should look like. Find out what data they think it should contain, how that data will be presented, and how different parts of the data are related.

Get input from as many Stakeholders as you can. Always keep in mind, however, that the Customer Champion is the one who understands the customers' needs thoroughly and has the authority to make the final decisions. Though you should consider everyone's opinions, the Customer Champion has the final word.

Depending on the scope of the project, this can take a while. I've been on projects where the initial brain-picking sessions took only a few hours and I've been on others where we spent more than a week talking to the customers. One project was so complex that part of the project was still defining requirements after other parts of the project had been underway for months.

Take your time and make sure the customers have finished telling you what they think they need.

Walk a Mile in the User's Shoes

Often following the customers' day-to-day operations can give you some extremely helpful perspective. Ideally you could do the customers' jobs for them for a while to thoroughly learn what's involved. Unless your customers aren't in your industry (and if they are, why are they hiring you?), however, you probably aren't qualified to do their jobs.

I was once saddened to read an article about ice cream testers. I eat a lot of ice cream and thought I had a good sense of what tastes good and what tastes bad, but professional ice cream testers can isolate and identify individual flavors in recipes that include dozens of ingredients. I'm not even competent to eat ice cream professionally!

Though you may not be able to actually do the customers' jobs, you may be able to sit next to them while they do it. Warn them that you will probably reduce productivity slightly by asking stupid and annoying questions. Then ask away. Take notes and learn as much as you can. Sometimes your outsider's point of view can lead to ideas that the customers would never have discovered.

Another Point of View

On one project, we visited a billing center responsible for a couple million accounts. Every three days they processed 1/10th of their accounts and one of the things they did was print out a pile of paper almost three feet tall listing all of the accounts that owed money.

Unfortunately the accounts were arranged by ID, not balance, so they couldn't figure out which ones owed the most. In fact, by state law they were not allowed to do anything about accounts that owed less than $50, and those included the vast majority.

Because of our outsider computer nerd viewpoint, we knew there was a better approach. We installed a printer emulator (a program that looks like a printer to the system but actually captures the data instead of killing trees with it) and dumped the data into a file. We sorted the file by account balance and displayed the result to the user. The first two or three pages listed all of the accounts that needed action. (In fact, the first four or five accounts usually owed more than all of the other accounts combined.)

We were actually there looking at a different problem but when we saw this one we jumped all over it and in about a week we were heroes.

Take notes while you're watching the customers do their jobs. Draw pictures and diagrams if that helps you visualize what they're doing. Pictures can also be very helpful in asking the customers if you have the right idea. If the customers will let you, print screen shots and even take photographs. (However, keep in mind that many businesses are required to safeguard the privacy of their clients' data, so don't expect them to let you walk out with screen shots or photographs showing credit information, medical histories, or records of political contributions. Be sure you ask before you try to take any material away and ask before you even bring a camera in the building.)

Study Current Operations

After you've walked a mile or two in the customers' shoes, see if there are other ways that you can study the current operation. Often companies have procedure manuals and documentation that describes the customers' roles and responsibilities. (In fact, that kind of documentation is required for certain kinds of ISO certifications. Some bigger companies like to display huge banners that say things like ''ISO-9000

Certified.'' These may just be there to cover holes in the wall, but if they have such a banner they probably have more documentation than you can stomach.)

Look around for any existing databases that the customers use. Don't forget the lesson of Chapter 2 that there are many different kinds of databases. Don't just look for relational databases. Look also for note files, filing cabinets, boxes of index cards, tickler files (cubbies where they can place items that should be examined on a certain date), and so forth. Generally, snoop around and find out what information is kept where.

Figure out how that information is used and how it relates to other pieces of information. Often different physical databases contain redundant information and that forms a relationship. For example, a filing cabinet holding information about customers includes all of the customers' data. A pile of invoices also includes the customers' names, addresses, ID numbers, and other information that is duplicated in the customer files. Paper orders probably contain the same information. These are the sorts of pieces of data that tie the whole process together.

Brainstorm

At this point, you should have a decent understanding of the customers' business and needs. To make sure the customer hasn't left anything out, you can hold brainstorming sessions. Bring in as many Stakeholders as you can and let them run wild. Don't rule out anything yet. If a stakeholder says the database should record the color of customers' shoes when they make a purchase, write it down. If someone else says they need to track the number of kumquats eaten by assembly line workers, write it down.

Continue brainstorming until everyone has had their say and it's clear that no new ideas are appearing.

Occasionally extra creative people (sometimes known to management as ''troublemakers'') look like they're going to go on forever. Let them go for a while but if it's clear they really can't stem the flood of ideas, split up. Have everyone go off separately and write down anything else relevant that they can think of. Then come back and dump all of the ideas in a big pile.

Try not to let the Customer Champion suppress the others' creativity too early. Though the Customer Champion has the final say, the goal right now is to gather ideas, not to decide which ones are the best.

The goal at this point isn't to accept or eliminate anything as much as it is to write everything down. You want to be sure that everything relevant is considered before you start designing. Later, when you've started laying out tables and indexes and changes are more difficult to make, you don't want someone to step in and say, ''Owl voltages! Why didn't someone think of owl voltages?'' Hopefully you have owl voltages written down somewhere and crossed out so you can say they were considered and everyone agreed they were not a priority.

> *Different development shops take different approaches if this earth-shatteringly important requirement somehow got missed during brainstorming. I prefer to grudgingly add it to the requirements, while making sure that the customers understand that this sort of last minute change might affect the schedule. If you grumble a little, they usually take the hint and only insist on changes that really are important. Other shops simply say, ''Sorry, that wasn't in the original requirements and we're not doing it so there!'' Though this is technically correct, it increases the chances that the final product won't meet the customers' needs.*

Look to the Future

During the brainstorming process, think about future needs. Explicitly ask the customers what they might like to have in future releases. You may be able to include some of those ideas in the current project, but even if you can't it's nice to know where things are headed. That will help you design your database flexibly so you can more easily incorporate changes in the future.

For example, suppose your customer Paula Marble runs a plumbing supply shop but thinks some day it might be nice to add a little café and call the whole thing ''Paula's Plumbing and Pastries.'' After you hide your snickers behind a cough, think about how this might affect the database and the rest of the project.

Plumbing supplies are generally non-perishable, but pastries must be baked fresh daily and the ingredients that go into pastries are perishable. You may want to think about using separate inventory tables to hold information about non-perishable plumbing items that clients can purchase (gaskets, thread tape, pipe wrenches) and perishable cooking items that the clients won't buy directly (flour, eggs, raisins).

You might not even track quantity in stock for finished pastries (the clients either see them in the case or not) but you probably want to be able to record prices for them nonetheless. In that case, you will have entries in an inventory table that will contain prices but that will never hold quantities.

You don't necessarily need to start planning the future database just yet (after all, Paula may decide to go with ''Paula's Plumbing and Tattoo Palace'' instead), but you can keep these future changes in mind as you study the rest of the problem.

Understand the Customers' Reasoning

Occasionally you'll come across a customer who thinks he knows something about database design. He may say that you should use a particular table structure, an object-relational hierarchical data model, or an acute polar space modulator.

Sometimes these suggestions make perfect sense. Other times you'll think the customer clicked the Google ''I'm Feeling Lucky'' link and stumbled into the endless morass of techno-babble.

Even if the suggestions seem to make no sense whatsoever, don't dismiss them out of hand. Remember that the customer has a different perspective than you do. The customer knows a lot more than you about his particular business. He may or may not know anything about database design, but it's entirely possible that he has a reason for his obscure requests.

For example, suppose you're trying to design a sales and inventory system for Thor's Thimbles. The president and CEO Thor says he thinks you need to use a temporal database, although the way he pronounces it makes you think he probably doesn't understand what that means (or perhaps it's just his Scandinavian accent). You think, ''How hard can it be to sell thimbles?'' and ignore him.

After you spend a month building a really slick relational database you discover that old Thor isn't so naive after all. It turns out that the company sells hundreds of different models of thimbles made from such materials as stainless steel, anodized aluminum, gold, and platinum. The value of the more exotic models changes daily with precious metal prices. Almost as volatile are the collectors' models such as the

Great Scientists of History series and the Sports Immortals (the Pete Rose Hall of Fame model can bring up to $200 at auction).

Suddenly what you thought was a simple problem really does have hundreds of variables changing rapidly over time and you realize that you probably should have built a temporal database. You have egg on your face and Thor decides that his brother-in-law, who originally suggested the temporal database to Thor, might be able to do a better job than you.

Even if a customer's suggestion seems odd, take it seriously. Dig deeper to find out why the customer thinks that will be useful. Take the approach my doctor takes when I tell him that I think I have scurvy or the plague or some other nonsense. He keeps an absolutely straight face and asks, ''Why do you think that?'' I won't be right but the symptoms I used in my incorrect diagnosis may help him decide that I really have a cold. (I envision him with the other doctors sitting in the break room later laughing and saying, ''You'll never guess what my patient thought he had today! Ha, ha, ha!'')

Try It Out Who's Right?

Suppose you have a customer who says you should use an XML-enabled object-relation database. You look into the problem and don't think that makes any sense. You ask the customer and he gives you a bunch of half-justifications that don't really add up. In the end he says, ''Just do it.''

How should you respond?

How It Works

This is a tricky situation. Everyone dreads the customer who tells you point-blank to do something that you know doesn't make sense. Do you waste the customer's time and money to pursue the wrong course? Or do you tell the customer that you won't do it and risk getting fired?

Everyone has to make this call for him- or herself. You're the one who has to be able to sleep at night after making the decision.

My personal philosophy is that I put the customer's needs first. If I think the customer is telling me to do something incorrect, I'll say so. But if the customer insists and I think I can do what he wants, I'll go ahead and do my best. In the end, it's the customer's money after all. If I make a big deal out of it and get fired, he'll probably just go out and find someone less experienced who blunders in without seeing the consequences of following the misguided advice and will make matters worse than I would.

However, I've rarely come to this point with a customer. Usually if you can explain your concerns in terms that customers can understand, they'll either convince you that there's a reason to their madness or they'll realize that the issues you've raised make sense.

Learn What the Customers Really Need

Sometimes the customers don't really understand what they need. They think they do and they almost certainly understand the symptoms of their problems, but they don't always make the right cause-and-effect connections.

Sometimes customers think a database or a new computer program will magically increase their sales, reduce their costs, walk their dogs, and wash their cars. In fact, a well-designed database will increase consistency, reduce data entry errors, provide reports, and otherwise help the customers manage their data, but that won't necessarily translate into higher profits.

As you look over the customers' operation, keep in mind that their real goals may not be exactly what they think they are. Their real goals probably include things such as making bigger profits, making fewer mistakes so they don't get yelled at as much by managers and clients, and finishing their daily work in time to go watch junior's soccer practice.

Look for the real causes of the customers' problems and think about ways you can address them. If you can see a way to improve operations, suggest it (always keeping in mind that they probably know a whole lot more about their business than you do so there's a good chance that your idea won't fly).

> *By the way, never ever tell a customer, ''What you really need is a slap in the head and a better product.'' That sort of non-constructive criticism may be gratifying but usually generates an unfavorable response.*

Prioritize

At this point, you should have a fair understanding of the customers' business, at least the pieces that are relevant to your project. You should understand at least roughly which customers will be playing which roles during the upcoming drama. At a minimum, you should know who the Customer Champion and Customer Representatives are so you know who to ask questions.

You should also have a big list of desired features. This list will probably include a lot of unicorns and pixie dust — things that would be nice to have but that are obviously unrealistic. It may also include things that are reasonable but that would take too much time for your current project.

To narrow the wish list to manageable scope, sit down with the customers and help them prioritize. You'll need the Customer Representatives who understand what is needed so they can make the decisions. Sometimes you may need the Customer Champion either in the meeting or available for consultation to make the tough calls.

Group the features into three categories. Priority 1 (or release 1) features are things that absolutely must be in the version of the project that you're about to start building. This should be the bare-bones essentials without which the project will be a failure.

Priority 2 (or release 2) features are those that the customers can live without until the first version is in use and you have time to start working on the next version. If development goes well, you may be able to pull some of these features into the first release but the customers should not count on it.

Priority 3 (or release 3) features are those that the customers think would be nice but that are less important than the priority 1 and 2 features. This is where you put the unicorns and pixie dust so you can ignore them for now.

> *You don't need to tell the customers but the priority 3 features are unlikely to ever make it into production. By the time release 1 is finished, the customers will have thought of a plethora of other priority 1 and 2 features that they want in release 2 so the release 3 features will remain unimplemented in the next version, and so on forever by induction.*

This is another place where different development shops take different approaches. In the more flexible approach that I prefer, these categories are somewhat flexible. If, during development, you discover that some priority 2 feature would be really easy to implement, you can pull it into the current release. In contrast, if some priority 1 feature turns out to be unexpectedly hard, you might ask the customers how important it really is and suggest that it be bumped to the priority 2 list to avoid endangering the schedule.

To make this sort of shuffling easier, it can be helpful to further prioritize the items within each category. If an item is high up on the list of priority 1 items, it is not a likely candidate for deferral to the next release. Similarly, if an item is high up in the priority 2 list, you might be willing to spend a little extra effort to bring it into the first release.

In a hard-line development approach, the categories are fixed after the requirements phase ends and items never move from one category to another. This prevents the customers from promoting items from priority 2 to priority 1, so it can save you some trouble. However, this approach also makes it hard for you to downgrade a feature that turns out to be a real project albatross.

Verify Your Understanding

With your notebook (and brain) bursting at the seams with all of this information, it's almost time to move on to the next chapter and start building a data model. Before you do, you should verify one last time that you really understand the customers' needs. This may be your last chance to avoid a painful catastrophe, so be sure you've gotten it right.

Walk through your understanding of the system and explain it to the customers as if they were building the system for you and not the other way around. They should make comforting grunts and noises such as ''yup'' and ''uh huh.''

Watch out for words such as ''but,'' ''except,'' and ''sort of.'' When they use those words, make sure they're only emphasizing something that you already know and not adding a new twist to things. Often at this stage the customers think of new situations and exceptions that they didn't think of before.

Pay particular attention to exceptions — cases where things mostly work one way but occasionally work in another. Exceptions are the bane of database designers and, as you'll see in the following chapter, you need to handle them in a special way.

For example, suppose you need to allow for returns. (A client might decide that the Kathryn Janeway sculpture he ordered is too short or clashes with his Predator statue.) While reviewing your understanding of the project, you say, ''So the receiving clerk enters the RMA (Return Merchandize Authorization number) and clicks Done, right?'' Your customer representatives look sheepishly at each other and say, ''Well... usually but sometimes they don't have an RMA. Then they just write in 'None.''' This is an important exception that the customers didn't tell you about before and you need to write it down.

For another example, suppose your customers currently use paper order forms that have shipping and billing address sections. You say, ''So the form needs to hold one shipping address and one billing address?'' Your customer replies, ''Well, sometimes we need two shipping addresses because different parts of the order go to different addresses.'' Someone pulls out an order form where a second address and additional instructions have been scribbled in the margin.

This is a huge exception. It's easy enough to add little notations to a paper form but it's impossible to add more than one address value to a single set of fields in a database. You can work around the issue if you plan for it, but it can be a major headache if you don't learn about it ahead of time.

For a final example, suppose a customer record needs a billing address. While you're reviewing your understanding the customer says, ''Oh yeah, and a shipping address because sometimes they buy one as a gift.'' Now you have to wonder if sometime later someone will decide that you also need a contact address in case you have questions about the order. Or a corporate address where you can send legal correspondence. Or perhaps a whole slew of branch office addresses. Or an executive address where you can send golf clubs to bribe the client's executives.

When your customer expands a single field (or a group of fields such as an address), you should ask seriously whether it's going to happen again. If the record needs to hold many copies of the same field, you can easily pull them into a separate table if you plan ahead of time, but it can be hard to add new copies of fields to a table after you build it and its user interface. A single customer record can hold one or two addresses but not an ill-defined, ever-expanding number. It's better to know ahead of time and plan for an arbitrary number of related addresses.

Sometimes in database design it's better to only allow one or many related items. There's no such thing as two.

Write the Requirements Document

The *requirements document* describes the system that you are going to build. This document is sometimes called the *product requirements document* (PRD), the *requirements specification*, *specification*, or *spec*. As all of these names imply, this document specifies the project's requirements.

At a minimum, the requirements document needs to spell out what you're planning to build and what it will do. It needs to explain the problems that it will solve and it should describe how the customers will use it to solve their problems. It can also include any design or architecture that you've already done, and it can include (possibly as attachments or appendixes) summaries of the discussions you've had while deciding on the project's features.

The requirements document keeps everyone on track during later design and development. It can also prevent finger-pointing when someone starts yelling about how you forgot to include the telepathic user interface. You can simply point to the requirements document and say, ''Sorry but the telepathic interface isn't in here.'' In fact, if you considered this issue during brainstorming and dumped the telepathic interface into the priority 3 ''unicorns and pixie dust'' category, having it listed there will probably allow you to skip the whole argument. The potential wave-maker can see that the issue has been shelved for now and will probably not bother stirring up trouble on a dead issue.

(I've worked on some projects that had enormous requirements documents, sometimes running to 500 or more pages. In that case, it's hard for anyone to remember everything that's in there and you may end up revisiting some issues occasionally.)

The requirements document should define *deliverables* (also called *milestones*, not to be confused with millstones) that the customers can use to gauge the project's progress. These should be tasks that you complete along the way that you can show the customer and *that can be verified in some meaningful way*. It's important that they be verifiable. Saying you're 25 percent done thinking about the design doesn't

do the user any good. Saying that you will have a database built and you will have filled it with test data drawn from a legacy system is much more useful and verifies that the database can hold that kind of data.

If you make the database design a deliverable (usually a good idea), then you need to be able to somehow verify that the design meets the customers' needs. Usually that means an extensive review where a lot of people put their heads together and try to poke holes in your carefully crafted design.

Prototypes also make excellent deliverables. Customers can experiment with a prototype to better understand what the system will do and they can give you feedback if you're not heading in the right direction. If you're building a full-blown user interface for the database, you could mock-up some prototype screens (probably with no error checking and possibly with just a little concocted data) to give the customers a feel for the completed application.

Some of the deliverables defined by the requirements document should be *final deliverables*. These are deliverables that determine whether the project is finished. Like all of the other deliverables, they must be measurable to be useful.

A particularly useful technique for deciding when a project has met its goals is to create use cases. Use cases are described in the following section.

Make Use Cases

A *use case* is a script that the users can follow to practice solving a particular problem that they will face while using your finished product. These can range in complexity from the very simple such as logging in or closing the application, to the extremely complex such as scheduling a fleet of trucks to perform in-home dog grooming.

Depending on how complete the user interface design is when you are writing the use cases, these may be sketchy or extremely detailed. They may spell out every keystroke and mouse movement that the user must make or they may provide vague instructions such as, ''The user will use the Order Entry form to place a new order.''

When the project is finished, the customers should review all of the use cases and verify that the finished project can handle them all. (In self-defense, you should run through the use cases before you tell the customers that you're finished. That way you don't look silly when the product cannot handle simple chores during an executive dog and pony show.)

Some of the things that you might specify when writing up use cases include:

- **Goals:** A summary of what the use case should achieve.
- **Summary:** An executive overview that your Executive Champion can understand.
- **Actors:** Who will do what? This includes people, your finished system, other systems, and so forth. Anyone or anything that will do something.
- **Pre- and post-conditions:** The conditions that should be true before and after the use case is finished. For example, a pre-condition to placing a new order might be that the client placing the order already exists.

- **Normal Flow:** The normal steps that occur during the use case.
- **Alternative Flow:** Other ways the use case might proceed. For example, when a user tries to look up a customer, what happens if the customer isn't there?
- **Notes:** Just in case there are special considerations that the person following the use case needs to know.

Many developers like to draw use case diagrams to show what actors perform what tasks. These seem to usually work at one of two levels.

A higher level use case shows which actors perform which tasks. For example, the Student actor enrolls in a class and takes the class, the Instructor actor teaches the class and assigns grades, and so forth. This type of use case diagram provides little detail about how the actors accomplish their tasks. It's useful early on when you know what you want to do but don't yet know how the system will do it.

Figure 4-1 shows a high-level use case diagram. Actors are shown as stick figures, tasks are shown in ellipses, and lines connect actors to tasks. More elaborate use case diagrams use other kinds of arrows, lines, and annotations to provide more detail.

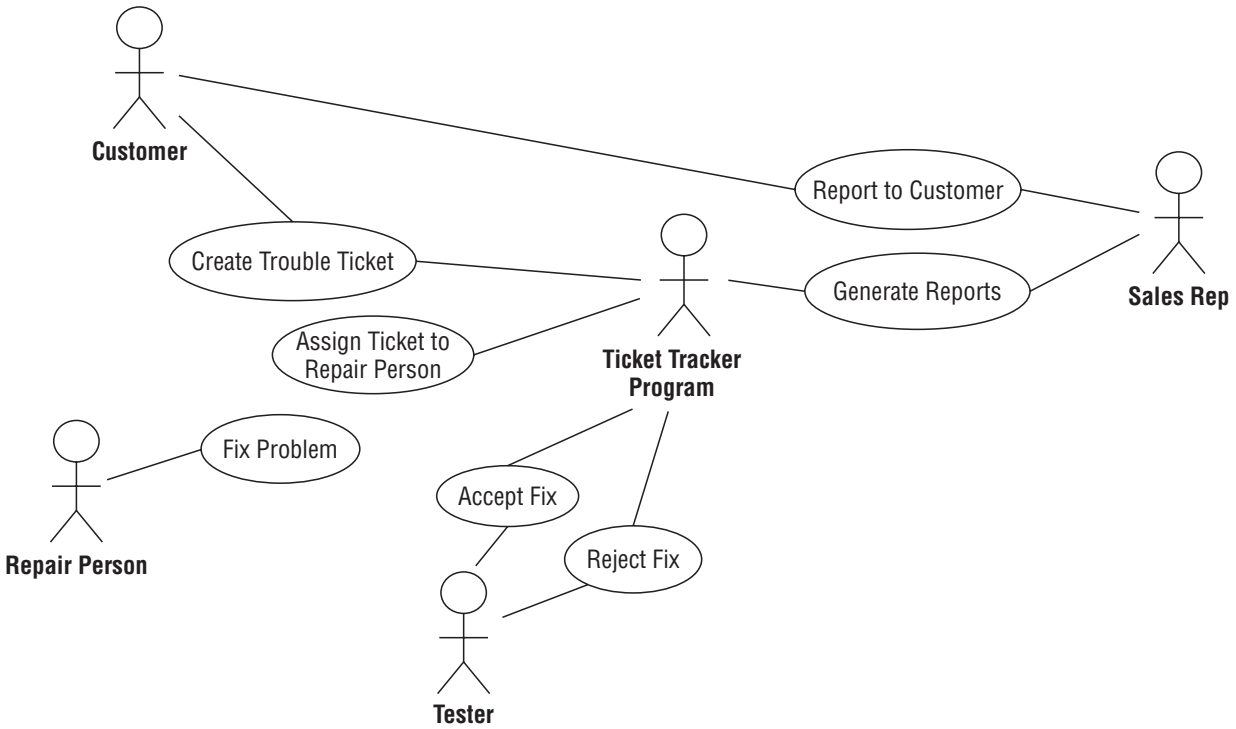

Figure 4-1

The second kind of use case lists more specific steps that actors take to perform a task, although the steps are still listed at a fairly high level.

Neither of these kinds of use case diagram provides enough detail to use as a script for testing, although they do list the cases that you must test. Because they are shown at such a high level, they are great for executive presentations. For more information on use case diagrams, look for books about UML (Universal Modeling Language), which includes use case diagrams, or search the Web for ''use case diagram.'' Two links that provide introductions are:

- `atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/use_case.htm`
- `www.developer.com/design/article.php/10925_2109801_1`

Typical use cases might include:

- The user logging in.
- The user logging out.
- Switching users (if the program allows that).
- Creating a new customer record.
- Editing a customer record.
- Marking a customer record as inactive.
- Creating a new order for an existing customer.
- Creating a new order for a new customer.
- Creating an invoice for an order.
- Sending out late payment notices.
- Creating a replacement invoice in case the customer lost one.
- Receiving a payment.
- Defining a new inventory item (when the CEO decides that you should start selling Rogaine for Dogs).
- Adding new items to inventory (for example, when you restock your fuzzy dice supply).
- Etc.

The list can go on practically forever. A large project can include hundreds of use cases and it may take quite a while to write them all down and then later verify that the finished project handles them all.

In addition to being measurable (you want to be able to tell whether the program can pull its weight), use cases should be as realistic as possible. There's no point in verifying that the program can handle a situation that will never occur in real life.

> *In one project, the program we were writing needed to be able to handle 20 simultaneous users. One customer performed a test where 20 people all sitting in the same room walked step-by-step through the same use case at the same time. They all typed in the same text and clicked the Find button at the same time. The program gave terrible performance because every user's computer tried to access the same database records at the same time. In a more realistic test, every user tried to access a different record and everything was fine.*

Try It Out Use Cases

Suppose you are building a program to let students log on over the Internet and enroll in classes. All enrollments are tentative until a specific date on which they are all processed. (That gives the school a chance to juggle schedules; for example, if a graduating student really needs a class, another student might get bumped for now.) To accommodate this flexibility, students should enter alternate choices.

For this exercise, make a list of database use cases that you could use to look for data that you have not built into the design and to later test to ensure that all of the data is present. You don't need to explain how a user will perform a certain task, just briefly describe the task and list the kinds of data that must be stored or accessed during that task. Add any questions that need further study or feedback from the customers.

How It Works

You should perform use cases covering every task that the final users of the system would perform. Here's the list that I've come up with.

Task	Data Needs
Log on successfully or unsuccessfully	Verify UserName and Password in Students table. (How do we generate these? How do we guarantee security?)
Enter desired schedule	Let students pick from dropdown lists so we don't need to verify that they typed meaningful choices. Refer to course schedule tables to give students choices. Save student selections in student selections tables. (Allow students to prioritize their selections?)
Generate final schedules	Refer to course schedule tables to get Capacity. Refer to global tables to learn minimum enrollment to not cancel a class. (Or does this vary by class? By department?) Process student selections tables, adding students to desired classes in the course tables. If a class fills, bump lower priority students, consult their selections, and assign a replacement course. If a class has too few students, notify the administrator to cancel the class. Consult the selections of any students in the class and assign replacement courses. When finished, review the course tables and copy student course assignments into student data tables. Check global tables (vary by department?) to learn minimum and maximum normal course load. If a student falls outside of those bounds, look up the student's counselor in the student tables and notify that counselor via email.
Send schedules to students	Get student schedule and email address from student tables. Email the schedule.

Task	Data Needs
Email course rosters	Get course roster data from course tables. Get the name and email address of each course's instructor from the course tables. Get the instructor's email address from the instructor tables. Email the class's roster to the instructor.
Manually adjust schedules	Allow administrators to manually adjust schedules to handle special circumstances. This will require free access to course tables, student data tables, and student course assignment tables.

Decide Feasibility

At some point, you should step back, take a deep breath, and decide whether the project is feasible. Is it even possible to design a database to do everything that the customer wants it to do?

Can you really build a database to hold records for 17 million customers, provide simultaneous access for 80 service representatives, log every transaction with timestamps and user IDs, give interactive responses to queries in less than 2 seconds 90 percent of the time, and still fit it all on a 16MB flash drive?

Okay, the last condition is pretty unrealistic but seriously, someone needs to think about the project's viability at some point. No one will be happy to hear that you can't solve all of the customers' problems, but everyone will be a lot happier if the project is canceled early instead of after you've waste a year of everyone's time and a king's ransom in funding.

If it really looks like you can't complete the project, make the tough call and ask everyone to rethink. Perhaps the customers can give up some features to make the project possible. Or perhaps everyone should just walk away and move on to a more realistic project.

Summary

Building any custom product is largely a translation process whether you're building a small database, a gigantic Internet sales system similar to the one used by Amazon, or a really tricked-out snowboard. You need to translate the half-formed ideas floating around in the minds of your customers into reality.

The first step in the translation process is understanding the customers' needs. This chapter explained ways you can gather information about the customers' problems, wishes, and desires so you can take the next step in the process.

In this chapter you learned how to:

- ❑ Try to decide which customers will play which roles.
- ❑ Pick the customers' brains for information.

- ❑ Look for documentation about user roles and responsibilities, existing procedures, and existing data.
- ❑ Watch customers at work and study their current operations directly.
- ❑ Brainstorm and categorize the results into priority 1, 2, and 3 items.
- ❑ Verify your understanding of the customers' needs.
- ❑ Write a requirements document with verifiable deliverables including use cases.

After you've achieved a good understanding of the customers' needs and expectations, you can start turning them into data models. The following chapter explains how to convert those needs into informal data models that help you better understand the database, and then how to convert the informal models into more formal ones that you can actually use to build a database.

Before you move on to Chapter 5, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to selected exercises in Appendix A.

Exercises

1. In Figure 4-2, draw lines connecting the customer roles with their corresponding descriptions.

Customer Role	Description
Convert	Someone who won't be around for long. May be helpful or may not care all that much.
Customer Champion	Answers your questions about the project.
Customer Representative	Anyone who has an interest in the project.
Devil's Advocate	Makes things generally run smoothly. Not glamorous but very useful.
Executive Champion	Provides a reality check and prevents groupthink.
Generic Bad Guy	Ranges from annoying naysayer to malicious saboteur/super villain.
Short-Timer	A user who originally was against your project that you include in the development process to bring them onto your side.
Sidekick/Gopher	The highest ranking customer driving the project. Willing to fight super villains.
Stakeholder	Thoroughly understands the customers' needs. Has the authority to make decisions that stick.

Figure 4-2

2. Which of the following does *not* describe a use case?

   a. A script for performing some task.
   b. Should describe a realistic operation.
   c. Should cover the customer's entire operation from start to finish.
   d. Should be verifiable.

3. Brainstorming sessions should ideally include:

   a. Customer Representatives.
   b. A Devil's Advocate.
   c. All interested Stakeholders.
   d. All of the above.

4. If a customer says you should use a hierarchical XML database, you should:

   a. Politely say, ''Thank you,'' and ignore this nugget of wisdom.
   b. Ask the customer why he thinks that.
   c. Do as the customer says. (It's his money.)
   d. Study the problem to see if that kind of database makes sense.

5. During a visit to view the customers' operation, you see someone repeatedly stamping the front of an order with the current date, turning the order over, turning it over again, and stamping the front with the date again. You should:

   a. Ask someone what that's all about.
   b. Suggest that the manager fire this crazy and possibly dangerous employee.
   c. Ignore the whole issue and stay focused on your own tasks.
   d. Avoid eye contact with this employee at all costs.

6. Look at the ZIP Code lookup form at `zip4.usps.com/zip4/welcome.jsp`. What are this form's data needs? Which fields are required? (How does the user know those fields are required?) What are the domains for the fields? Which *could* involve a foreign key validation?

7. Which of the following is *not* a security issue that you should consider when studying the project?

   a. The number of classes of users the database must support.
   b. Whether you need to provide audit trails to record changes to the data.
   c. The frequency with which you need to perform backups.
   d. Whether the users should have individual passwords.

8. You are called upon to design a database for a florist shop named ''Frank's Floral Fantasies.'' Frank thinks that he might want to track the medicinal and homeopathic properties of his plants because he thinks that might improve his sales of echinacea, St. John's Wort, and other plants. What priority should this requirement get?

a. Priority 1, definitely in this release.

b. Priority 2, probably in the next release.

c. Priority 3, with the unicorns and pixie dust.

d. It depends (you need more information).

9. Write a use case for logging in to your computer's operating system.

10. You're halfway finished designing your database when a Vice Presidential Super Villain says your project is doomed to failure because you didn't include a sufficient allowance for farbulistic granilation. You need to cancel the whole thing and start over with him in control. How should you handle this attack?

5

Translating User Needs into Data Models

Chapter 4 discussed ways you can work with customers to gain a full understanding of the problem at hand. The result should be a big pile of facts, goals, needs, and requirements that should be part of the new database and its surrounding ecosystem. You may already have made some connections among various parts of this information, but mostly it should be a big heap of requirements that doesn't say too much about the database's design and construction.

This kind of pile of information is sometimes called a *contextual list*. It's basically just a list of important stuff (although it may be fairly elaborate and include requirements documents, diagrams, charts, and all sorts of other supporting documentation).

The next step in turning the conceptual list into a database is converting it into a more formal model. You can compare the formal model to the contextual list and make sure that the model can handle all of your requirements.

You can also use the model to verify that you're on track. You can explain the model to the customers and see if they think it will handle all of their needs or if they forgot to mention something important while you were following the procedures described in Chapter 4.

> *Constantly verifying that you're on track is an important part of any project. It's much easier to hit a target if you're constantly checking the map and making any necessary adjustments. You wouldn't aim your car at a parking space, close your eyes, and step on the pedal, would you? It's much easier to park if you keep an eye on your progress, the other cars, the skateboarders slamming nosegrinds off the curb, kids riding on shopping carts, and everything else in the parking lot.*

After you build a data model (or possibly more than one), you can use it to build a relational model. The relational model is a specific kind of formal model that has a structure very similar to the one used by relational databases. That makes it relatively easy to convert the relational model into an actual database in Access, SQL Server, MySQL, or some other database product.

In this chapter you learn how to:

- Create user interface models
- Create semantic object models
- Create entity-relationship models
- Convert those types of models into relational models

After you master these techniques, you'll be ready to start pulling the models apart and rearranging the pieces to improve the design by making it lean and flexible.

What Are Data Models?

Despite what some managers occasionally seem to believe, a model isn't a silver bullet or enchanted wand that will magically make a project succeed. A model by itself doesn't do anything. It doesn't build a database, it isn't a piece of software (although there are software tools that can help you build a model), and the final user of your database never sees a model.

A model is a plan. It's a blueprint for building something, in this case a database. The purpose of the model isn't to do anything by itself. Instead it gives you a concrete way to think about the database that you are going to build. By studying the pieces of the model, you can decide whether it represents all of the data that you need to meet your customers' needs.

A model is also useful for ensuring that everyone on the project has the same understanding of what needs to be done. If everyone understands the model, then everyone should have the same ideas about what data should be stored, which tables should contain it, and how the tables are related. They should also agree on the business rules that determine how the data is used and constrained.

Note that it's important that everyone actually understands the model. I've seen developers build remarkably complicated models and then dump them on hapless end users, expecting those users to understand the models' every subtle nuance. The developers ended up walking the users through the models until the users' heads were spinning and the developers could have convinced them of just about anything. The models are for those who know how to understand them, not necessarily for everyone.

After you build a model, you can look at it and ask questions such as:

- Where do we store customer information?
- How many contact names can we store for a customer?
- Where do we store the contacts' favorite colors?
- What if we need to store multiple price points for the same product?
- How do we store the seventeen kinds of addresses we need for customers?
- Where do we store supplier information?
- If someone asks about an order they placed but haven't received, how can we figure out where it is?

- Where can we enter special instructions for an order?
- How do we know when we need to restock left-handed cable stretchers?

You should also work through any use cases or current scenarios and see if the model can handle them. You can't actually fill out insurance claim forms and look in the warehouse for missing orders yet, but you should be able to say, ''this table contains the data we need to do that.''

The end users can help a lot with this part. Though they may not understand the models, they do understand their business and can ask these sorts of questions while you and the other developers try to figure out if the model can handle them.

If the model cannot handle all of your (and the users') questions, you need to adjust the model. You might need to add fields or tables, change a field's data type, make new connections between tables, or make other changes to satisfy the requirements. In extreme cases, it may be easiest to start a new model from scratch.

This chapter discusses four kinds of models that grow successively closer to the final database implementation.

First, a user interface model views the database at a very high level as seen from the final user's point of view. Depending on how you are going to use the database, this might be as the user will view the database through forms on a computer screen. This model is very far from the final database implementation and it doesn't tell much about the database design. This model is useful for understanding what data is needed by the project and how you might use it to navigate through the user interface.

The second and third types of models described in this chapter are semantic object models and entity-relationship models. These are roughly at the same distance from the final database. They are at a slightly lower level than the user interface model and show relationships among data entities more explicitly. They are still at a moderately high logical level, however, and do not provide quite enough detail to build the final database.

The fourth type of model described in this chapter is the relational model. This model mimics the structure of a relational database closely enough that you can actually sit down and start building the database.

In a typical database design project, you might start with a user interface model. I like to start there because I figure if the user is going to see something, we better have a place for it in the database. Conversely, if the user isn't going to see it in some manner, do we really need it in the database? (But that's just me. I like designing user interfaces. Some people prefer to skip that and let someone else worry about the user interface.)

Next you use what you learned from the user interface model to build either a semantic object model or an entity-relationship model. These models serve the same purpose so you generally don't need to build them both. Work through this chapter and the exercises at its end and decide which one you prefer.

Finally, you convert the semantic object model or entity-relationship model into a relational model. Now you have something that could be turned into a database. There are still some steps to go as you refine the relational model to improve the final database's reliability and performance, but those are subjects for later chapters.

Remember, these models are intended to better your understanding of the data and the ways in which different bits are related, so with that in mind, anything that increases understanding is beneficial. Don't be afraid to add notes that clarify confusing issues. Feel free to modify the basic modeling techniques described here. There's some benefit to sticking close to standard notations because it lets others who have studied the same notation understand what you are doing, but if adding a number in a box by each link or a colored triangle helps you and your team get a better handle on the design, do it. Just be sure to make a note of your additions and changes so everyone is on the same page.

User Interface Models

In most database applications, a user will eventually see the data in some form. For example, an order entry and tracking application might use a series of screens where the user can perform such chores as entering orders, tracking orders, marking an order as paid, looking up available inventory, and so forth. Those screens form the database's user interface.

Some databases don't have their own user interfaces, at least not that a human will see. Some databases are designed to store data for other applications to manipulate. In that case, it is the interfaces that those other applications provide that the human user sees. If possible, you should consider what those applications will need to display and plan accordingly. Sometimes it is useful to build throwaway interfaces to view the data on forms, in spreadsheets such as Microsoft Excel, or in text files.

You should also consider how those other applications will get the data from your database. The way in which those applications interact with your database forms a non-human interface and you should plan for that one, too. For example, suppose you know that a dispatch system will need to fetch information about employees from your database and information about pending repair jobs from another system. You should think about the kinds of employee data that the dispatch system will need (things such as a repairperson's skills, equipment, assigned vehicle, and so forth). Then you can design your database to make fetching this data easy and efficient.

To build the user interface model, start by making rough sketches of the screens that the user will see. Often these first sketches can come directly from paper forms if any exist.

Include the fields with sample data to make it easier to understand what belongs on each screen. These sketches can be anything from crayon scribbles on bar napkins, to forms drawn with your favorite computerized drawing tool, to full user interface prototypes. Figure 5-1 shows a mocked-up Find Orders screen built with Visual Basic. This form holds only controls and doesn't include any code to do anything more than just sit there and look pretty.

In addition to the image in Figure 5-1, you should include text explaining what the various parts of the form do. In this case, that text might say:

- The user enters selection criteria in the upper part of the form and clicks the Search button.
- The program displays a list of matching order records in the bottom of the form.
- The user can select an order from the list and click Open to open that order's detail form.

At this level, the user probably thinks of each order as containing all of the information on this form. If you were to fill out an order on a piece of paper, that paper would include blanks for you to fill in customer name, customer ID, contact name, order date, and so forth. The order would also have a status,

although you might represent that by putting the order in boxes on your desk labeled Pending, Open, Closed, and so forth rather than by having a status box on the paper form.

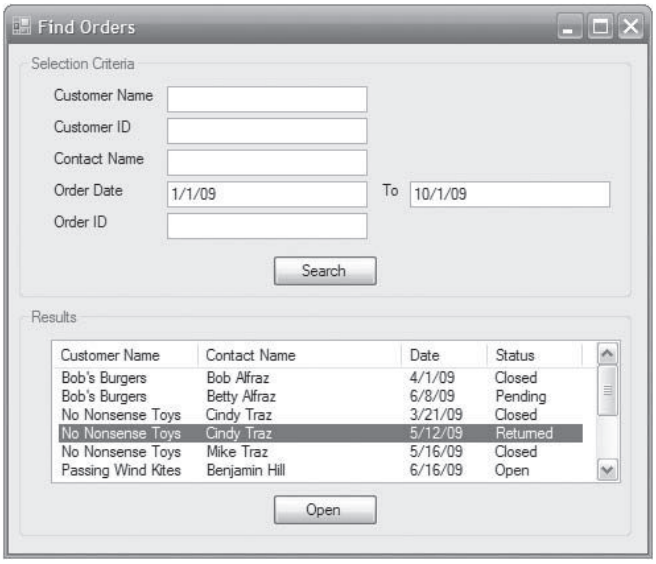

Figure 5-1

The form and its description also raise some important questions:

- What fields should be allowed as selection criteria?
- Should we index the selection criteria fields to make searching faster? Some or all fields?
- When the user selects an order and clicks Open, how does the program open the Orders record? (Searching for the exact combination of fields shown in the list would be slow and there might even be two entries with the same values if someone placed two orders on the same day. It might be wise to add an order ID field to make finding the record again easier.)

When you select an order from the form shown in Figure 5-1 and click Open, the program displays the form shown in Figure 5-2.

This form shows the fields that should be associated with an order. These include:

- Various dates such as the date the order was placed, the date the products were shipped, the date the customer paid, and so forth
- The order's current status
- The shipping method (Priority, Overnight, Armored Courier, and so forth)
- The billing method (credit card, invoice net 30)
- Various addresses such as the shipping and billing addresses
- Contact information for when we get confused (or want to send spam to the unsuspecting contact)
- The order's line items
- Subtotal, taxes, shipping, and grand total

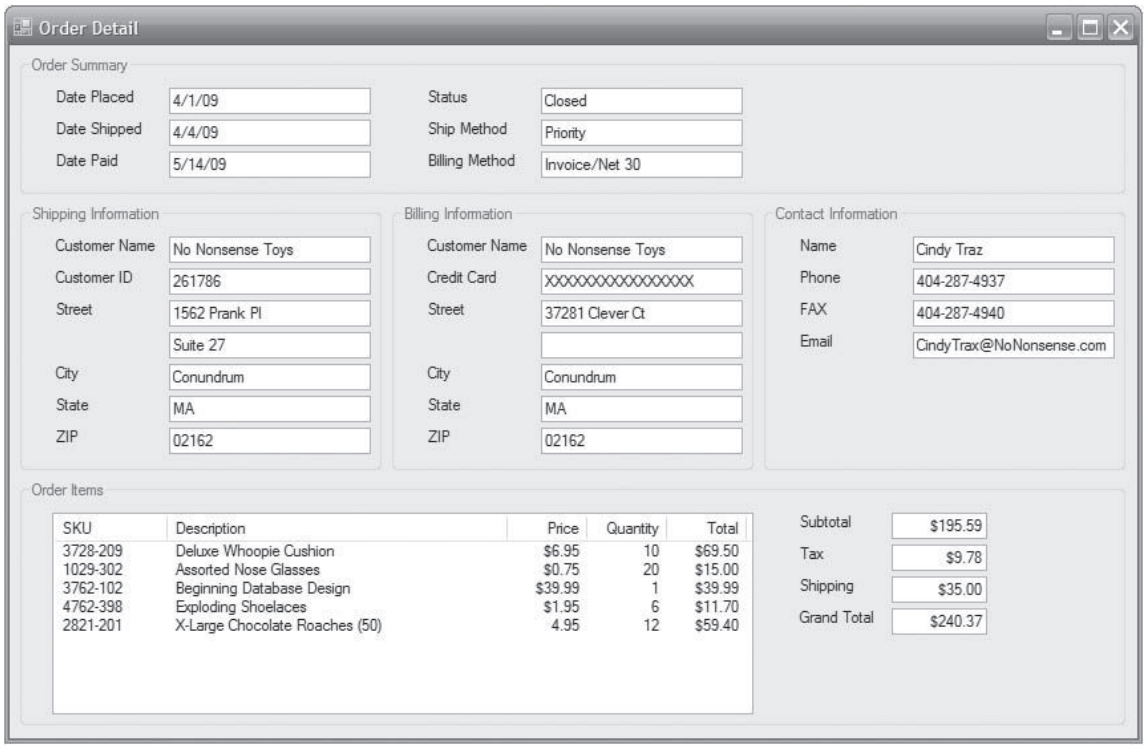

Figure 5-2

Both of these forms involve orders and both provide some information about the order data. The Order Detail form includes a lot of the fields that must be stored to represent an order. The Find Orders screen tells which order fields should be allowed as search criteria (and thus may make good keys) and which order fields should be displayed in the result list.

Each of these forms tells a little bit more about the order data. Other mocked-up forms would give even more information about the order data. For example, the application would need an order entry form and a form to update order information (such as changing the addresses or setting order status to Closed). Depending on how the work was divided among employees, there might be special forms for performing a single specific task. For example, an order fulfillment clerk (who puts things in a box and ships them) would need to be able to change an order's status to Shipped but probably doesn't need to be able to change credit card numbers. In fact, going through the screens and deciding which employees should be able to do which tasks gives you an initial indication of the application's security requirements.

Still other forms would give hints about other parts of the database. A full-fledged database for this application would need to include forms for managing inventory. (For example, how do we know there are any more whoopee cushions to sell and how do we know when to order more?) It might also include supplier information (who sells us our nose glasses?), employee information (who is assigned to pester delinquent customers this week?), advertising data (which spam campaigns gave us the most new contacts?), and so forth.

A large application might include dozens or even hundreds of forms, each of which gives only a partial glimpse of the information contained in the database. Together these mocked-up screens form a user interface model that shines spotlights into the data needed to support the application.

With the user interface model in hand, you are now ready to build a more formal model that shows the entities used by the application in greater detail. The first of those models discussed in this chapter is the semantic object model.

Try It Out User Interface Models

Sketch out a form where the user could enter shift information for employees. What data must be displayed on the form?

How It Works

Figure 5-3 shows a mocked-up employee shift form.

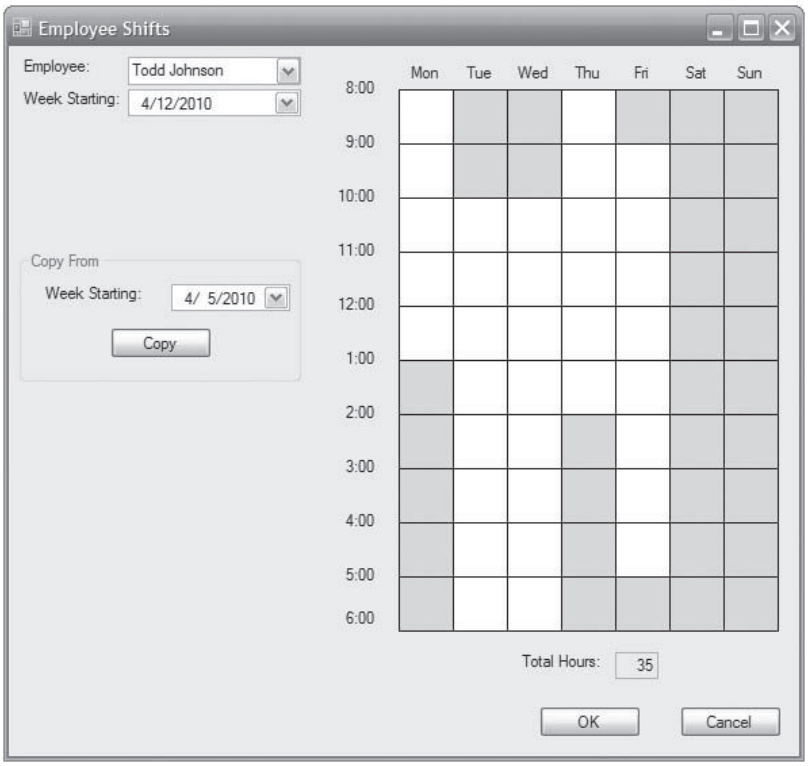

Figure 5-3

This form includes the following data:

- Employee name (selected from a combo box).
- The starting day of the week the user is viewing and editing for this employee (selected from a combo box). (Which weeks will we allow the user to pick? How far in the future?)
- The user should also be able to select past weeks (from a combo box) from which to copy.
- The hours that the employee is scheduled to work. These records (in the EmployeeShifts table?) will include employee, date, start time, and stop time.
- Total hours scheduled. This can be calculated from the shift data.

The form will also need to look up minimum and maximum normal hours so we can warn the user if something is unusual. For example, if the user is scheduled to work 70 hours in a week, the form can ask the user to verify before accepting the changes.

Semantic Object Models

A semantic object model (SOM) is intended to represent a system at a fairly high level. Though the ideas are somewhat technical, they still relate fairly closely to the way people think about things, so semantic object models are relatively understandable to users.

Classes and Objects

Intuitively a *semantic class* is a type of thing that you might want to represent in your system. This can include physical objects such as people, furniture, inventory items, and invoices. It can also include logical abstractions such as report generators, tax years, and work queues.

Technically a semantic class is a named collection of attributes that are sufficient to identify a particular entity. For example, a `PERSON` class might have `FirstName` and `LastName` attributes. If you can identify members of the `PERSON` class by using their `FirstName` and `LastName` attribute values, then that's good enough.

By convention, the names of semantic classes are written in ALL CAPS as in `EMPLOYEE`, `WORK_ORDER`, or `PHISHING_ATTACK`. Some prefer to use hyphens instead of underscores so the last two would be `WORK-ORDER` and `PHISHING-ATTACK`.

A *semantic object* (SO) is an instance of a semantic class. It is an entity instance that has all of the attributes defined by the class filled in. For example, an instance of the `PERSON` class might have `FirstName` ''David'' and `LastName` ''Letterman.''

Traditionally the attributes that define a semantic class and that distinguish semantic objects are written in mixed case as in `LastName`, `InvoiceDate`, and `DaysOfConfusion`.

Attributes come in three flavors: simple, group, and object.

A *simple attribute* holds a simple value such as a string, number, or date. For example, `LastName` holds a string and `EmployeeId` holds a number.

A *group attribute* holds a composite value — a value that is composed of other values. For example, an `Address` attribute might contain a `Street`, `Suite`, `City`, `State`, and `ZipCode`. You could think of these as separate attributes but that would ignore the structure built into an address. These values really go together so, to represent them together, you use a group attribute.

An *object attribute* represents a relationship with some other semantic object. For example, a relationship may represent logical containment. A `COURSE` class would have a `STUDENT` object attribute to represent the students taking the course. Similarly the `STUDENT` class would have a `COURSE` object attribute representing the courses that a student was taking. Each of these classes is related to the other so they are called *paired classes*. Similarly their related attributes are called *paired attributes*.

Cardinality

An attribute's *cardinality* tells how many values of that attribute an object might have. For example, at the start of some volleyball tournaments each team's roster must contain between 6 and 12 players.

You write the lower and upper bounds beside the attribute to which they apply separated by a period. The volleyball team roster's `Players` attribute would have cardinality 6.12. (I have no idea why it's a single period and not a dash or ellipsis.)

Usually the minimum cardinality is 0 if the value is optional or 1 if it is required.

The maximum cardinality is usually 1 if at most one value is allowed or N if any number of values is allowed.

Probably the most common cardinalities are:

- **1.1:** Exactly one value required. For example, suppose you are building a database to track restaurant orders. In the `ORDER` class, the `ServerName` attribute would have cardinality 1.1 because every order must have exactly one server.
- **1.N:** Any number of values but at least one required. For example, the `ORDER` class's `Item` attribute would hold the items ordered by the diners and would have cardinality 1.N. It wouldn't make sense to send an order to the kitchen if it didn't contain any items, but it could contain any number of items. (Although in practice I might double-check with the server if the kitchen received an order for 13,000 hamburgers.)
- **0.1:** An optional single value. For example, the server might want to record a comment to go with the order. (''Extra cheese on the milkshake.'')
- **0.N:** Any number of optional values. For example, a series of comments. (''Dressing on the side for salad 1. No mayo on burger 2. Recognize poor tipper, use day-old breadsticks.'')

Identifiers

An *object identifier* is a group of one or more attributes that the users will typically use to identify an object in the class.

An object identifier can include a single attribute such as `CustomerId` or a group of attributes such as `FirstName`, `MiddleName`, and `LastName`.

You indicate an identifier by writing the text ''ID'' to the left of its attributes. Often identifiers contain unique values so every item in the class will have different values for the identifier. For example, `CustomerId`, `SocialSecurityNumber`, and `Isbn` are unique identifiers for customers, employees, and books, respectively. You can indicate a unique identifier by underlining the ''ID'' to its left.

Sometimes non-unique identifiers are used to find groups of objects. For example, suppose the users of your system will want to find customers in a particular city. Then the `CUSTOMER` class's `City` attribute would be a non-unique identifier.

Putting It Together

Figure 5-4 shows a simple representation of a `CUSTOMER` class that demonstrates these notational features.

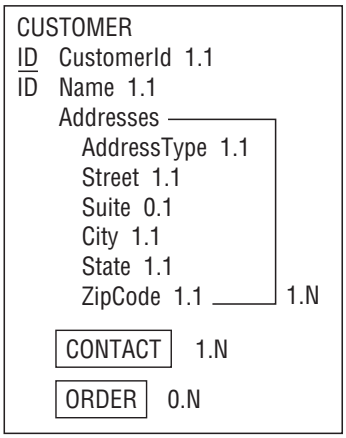

Figure 5-4

A big box surrounds the whole class definition. The class name, `CUSTOMER`, goes at the top.

`CustomerId` is a simple attribute that is used to identify customers so it gets the ID notation. `CustomerId` values are unique so the ID is underlined. This value is required and a customer can have only one ID so its cardinality is 1.1.

Users sometimes want to search for customers by name so the `Name` attribute is also an identifier. It is possible that two customers could have the same name, however, so here ID isn't underlined. (Duplicate customer names could also lead to a trademark battle if your customers are companies. Fortunately that's their problem, not yours.)

The `CUSTOMER` class includes address information stored in the `Addresses` attribute. Each address has the attributes `AddressType` (this will be something like `Shipping` or `Billing`), `Street`, `Suite`, `City`, `State`, and `ZipCode`. All of these except `Suite` are required and can hold only one value. The `Suite` attribute is optional. Lines show the attributes contained inside the `Addresses` value. The 1.N to the lower right of the group indicates that a `CUSTOMER` object must have one or more `Addresses` values (each containing a `Street`, `Suite`, `City`, `State`, and `ZipCode`).

Finally, the class has two object attributes named `CONTACT` and `ORDER`. The `CONTACT` attribute represents one or more contact people for the customer. The box around the attribute tells you that this is an object attribute. Its cardinality 1.N indicates that the `CUSTOMER` must have at least one contact.

The `ORDER` attribute represents the orders placed by this customer. You might think that this should have cardinality 1.N. After all, why would you need a customer who doesn't place any orders? However, when you first create a customer record it will have no associated orders. You might also want to be able to make a customer record in anticipation of future orders. For both of those reasons, this design sets the cardinality of `ORDER` to 0.N.

> *This is a design decision and in your application you could take the other route. You can look at the user interface model to see which would be more natural. Do you want to provide a screen where a user can create a customer record without an order or do you want to make the order entry screen allow for creating a new customer?*

Try It Out Semantic Object Model

Make a semantic object model for an `EMPLOYEE_WEEK` class that holds information about employees scheduled for a week. This class should have object identifier fields `EmployeeId` and `StartDate`. It should also have a group attribute named `Shift` that includes `StartTime` and `StopTime`, and it should hold one `Shift` for each of the seven days of the week.

How It Works

Figure 5-5 shows the semantic object model for the `EMPLOYEE_WEEK` class.

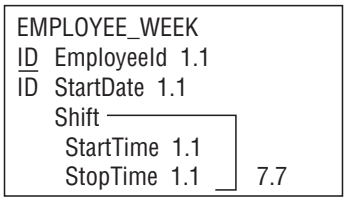

Figure 5-5

Semantic Views

Sometimes it is useful to define different views into the same data. For example, consider the kinds of information a company typically tracks for its employees. That information might include:

- Normal contact information such as name, address, phone number, and next of kin.
- Work-related contact information such as title, office number, extension, pager number, and locker number at the country club (if you're an executive).
- Confidential salary information including your complete salary and annual bonus history.
- Other confidential information such as your stock plan and 401K program participation, insurance selections, annual performance reviews, and golf handicap.

Some of this information, such as your name and title, is freely available to anyone who wants it.

Other semi-public information is available to anyone within the company but not outside the company. (Many companies worry that executive recruiters with the company phonebook could steal employees away with all of their valuable skills and the proprietary information locked inside their heads.) This information includes your office number, extension, project history, and birth date (excluding the year). It does not include your home address, annual performance reviews, salary history, or other financial data.

Other more sensitive information should be available to your manager and other superiors but not to the general population of coworkers. This information includes such things as your annual performance reviews and work history. However, your manager does not need to know how much you are having deducted for retirement contributions, whether you participate in the company stock plan, and whether

you are deducting the extra $750 a month for the dental plan. Those sorts of information should be hidden from your manager. (Depending on the way your company is structured, your manager might not even need to know your exact salary.)

The people in the Human Resources department are the ones who arrange to siphon money out of your paycheck for such perks as the stock plan and dental insurance so they obviously need to know that information. However, they probably don't need access to your annual performance reviews.

Figure 5-6 shows an `EMPLOYEE` class and four views that give access to different parts of the employee data. For simplicity I've shown each attribute as if it were a simple attribute when actually most of these are group or object attributes. For example, the `OfficeData` attribute is really a compound attribute including `Title`, `Office`, `Extension`, `BirthDate`, and so forth.

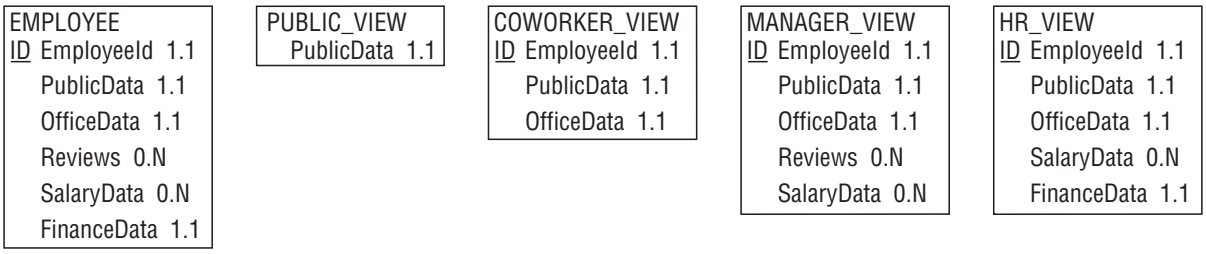

Figure 5-6

Defining these different views allows you to make data available only to those who need it. (This notion of *view* maps directly to the relational database concept of *view* so defining views now will help you later.)

After you finish building a complete semantic object model, you should check each of the views to ensure that they contain all of the information needed for each class of user and nothing else. For example, you should run through all of the use cases for managers and see if the `EMPLOYEE` class's `MANAGER_VIEW` provides enough information to handle those use cases. You should also check that every piece of data included in the `MANAGER_VIEW` is actually used. If something isn't used in some use case, then managers might not need it and it might not belong in the `MANAGER_VIEW`.

Class Types

The following sections describe some of the types of classes that you may need to use while building semantic object models. Some of these are little more than names for simple cases. Others such as association classes and derived classes introduce new concepts that are useful for building models.

Simple Objects

A *simple* or *atomic object* is one that contains only single-valued simple attributes. For example, an inventory item class might include the attributes `Sku`, `Description`, `UnitPrice`, and `QuantityInStock`. Each inventory item's data must include exactly one value for each of these attributes.

Figure 5-7 shows a simple `INVENTORY_ITEM` class.

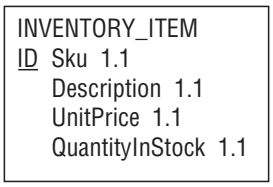

Figure 5-7

Composite Objects

A *composite object* contains at least one multi-valued, non-object attribute. For example, suppose you allow online customers to provide product reviews for inventory items. Then you could add a multi-valued `Reviews` attribute to the class shown in Figure 5-7 to get the composite object shown in Figure 5-8.

> *There's some difference among developers over these terms. Some call an object with a multi-valued, non-object attribute a ''complex object'' or ''complex type'' and use ''composite'' to mean an object that contains more than one data element. I think the terms defined here are more common but if there's any doubt in your discussion with other developers, you should agree on common definitions.*

Figure 5-8

Note that the multi-valued attribute need not be a simple attribute. For example, suppose you decide not to use a simple attribute to hold customer comments. Instead for each comment you store the customer's user name, a numeric rating, and comments. Figure 5-9 shows the revised `INVENTORY_ITEM` class.

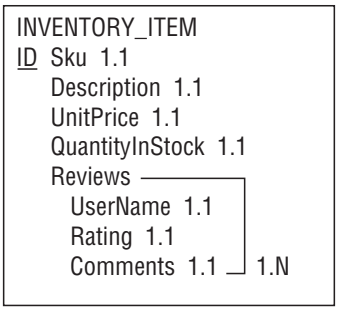

Figure 5-9

Compound Objects

A *compound object* contains at least one object attribute. For example, consider the `CUSTOMER` class shown in Figure 5-10. This class contains basic information such as a customer name and shipping

and billing addresses. Its CONTACT object attribute stores information about the person we should contact if we have a question about this customer. (This is also the person who gets our junk mail.) The SALES_REPRESENTATIVE object attribute refers to another object representing the sales representative who is charged with keeping this customer happy. (Okay, not too much junk mail.)

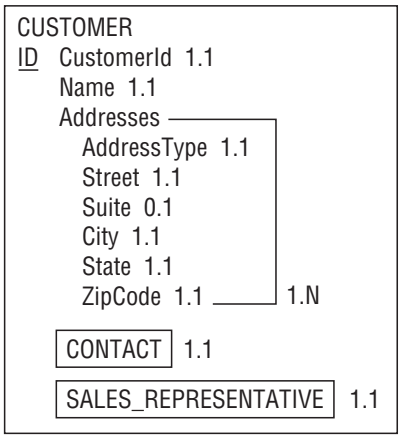

Figure 5-10

Hybrid Objects

A *hybrid object* contains a combination of the other kinds of attributes. For example, it might contain a multi-valued group that contains an object attribute. The ORDER class shown in Figure 5-11 contains a LineItems group attribute to represent the items in the order. Each LineItems entry contains an INVENTORY_ITEM object attribute that refers to an object of the type shown in Figure 5-9.

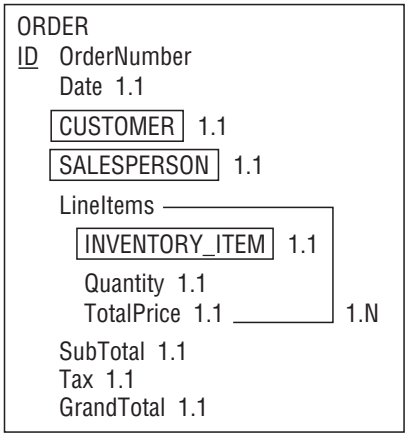

Figure 5-11

Association Objects

An *association object* represents a relationship between two other objects and stores extra information about the relationship.

Association objects are particularly useful for many-to-many relationships where an object of one class can be associated with many objects of a second and an object of the second class can be associated with many objects of the first.

For example, consider the `PROJECT` and `DEVELOPER` classes. A `PROJECT` may include many `DEVELOPER`s and a `DEVELOPER` may work on many `PROJECT`s, so the two classes have a many-to-many relationship. Figure 5-12 shows this relationship modeled with straightforward object attributes.

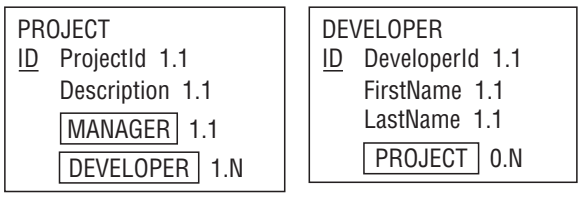

Figure 5-12

If this is all there is to the relationship, then this model is fine. However, if there is extra information that should be stored with the relationship, this model has no place to store that information.

For example, suppose developers play different roles in a project. A developer might be a technical lead, toolsmith, tester, writer, generic project member, or even the project's manager. In that case, there's no place to store this information in Figure 5-12. You cannot place it in the `PROJECT` class because data in that class applies to the project as a whole and not to a specific developer on the project. You cannot place the information in the `DEVELOPER` class because a developer might play different roles on different projects.

The solution is to create an association class to connect these classes and store the extra information. Figure 5-13 shows the new design. A `PROJECT_ROLE` object connects the `PROJECT` and `DEVELOPER` classes to represent the relationship that a particular developer has with a particular project. The `RoleName` attribute stores the information about the type of role that a particular developer plays in the project (technical lead, tester, and so forth).

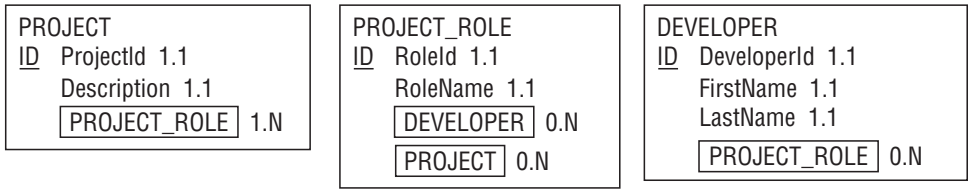

Figure 5-13

For a concrete example, consider Dr. Frankenstein's famous Build-a-Friend project. The following table shows this `PROJECT` object's attribute values.

ProjectId	Description	PROJECT_ROLE
Build-a-Friend	Make a friend out of spare parts.	Role1
		Role2

The following table shows the attribute values for the two DEVELOPER objects.

DeveloperId	FirstName	LastName	PROJECT_ROLE
Dr. Frankenstein	Ted	Frankenstein	Role1
Igor	Igor	Johnson	Role2

Finally, the following table shows the values for PROJECT_ROLE objects.

RoleId	RoleName	DEVELOPER	PROJECT
Role1	Mad Scientist	Dr. Frankenstein	Make-a-Friend
Role2	Flunky	Igor	Make-a-Friend

From this data, you can figure out which developers play which roles on what projects.

Try It Out Association Objects

Suppose you're putting together a database to record World of Warcraft adventures. You want to remember which player participated in which adventure. You also want to know what character they played during the adventure.

Make a semantic object model to record this information.

1. Create PLAYER and ADVENTURE classes.
2. Make a PLAYER_CHARACTER association class to fit between PLAYER and ADVENTURE. This class should store the character in addition to data linking the other two classes.

How It Works

1. Create PLAYER and ADVENTURE classes.

   The PLAYER class stores player information (PlayerId, FirstName, LastName, and so forth), plus an object attribute pointing to one or more PLAYER_CHARACTER objects. Those objects represent this player's characters in various adventures.

   The ADVENTURE class stores adventure information (AdventureId, Description), plus another object attribute pointing to one or more PLAYER_CHARACTER objects. Those objects represent all of the characters in the adventure.

2. Make a PLAYER_CHARACTER association class to fit between PLAYER and ADVENTURE. This class should store the character in addition to data linking the other two classes.

   The PLAYER_CHARACTER class stores the name of the character that the player used in this adventure. An object attribute points to the single PLAYER who played this character. Another object attribute points to the single ADVENTURE in which the player used this character.

Figure 5-14 shows the classes.

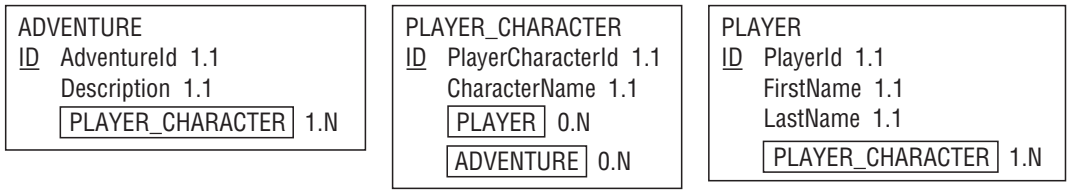

Figure 5-14

Inherited Objects

Sometimes one class might share most of the characteristics of another class but with a few differences.

For example, you've built a `CAR` class that has typical automobile attributes: `Make`, `Model`, `Year`, `NumberOfCupholders`, and so forth.

Now suppose you decide you need a `RACECAR` class. A racecar is a type of car so it has all of the same attributes that a car has. In addition, it has some racecar-specific attributes such as `ZeroTo60Time`, `ZeroTo100Time`, `TopSpeed`, and `QuarterMileTime`. You could build a whole new class that duplicates all of the `CAR` attributes but that would not only be extra work (something any self-respecting database designer should avoid), it also doesn't acknowledge the relationship between the two classes.

Instead you can make `RACECAR` a subclass or subtype of the `CAR` class. To denote a subclass in a semantic object model, create a `RACECAR` class that contains only the new attributes not included in `CAR`. Include an object attribute in `CAR` linking to the `RACECAR` class and using the notation `0.ST` in place of the cardinality to indicate that `RACECAR` forms an optional subtype for `CAR`. Then place an object attribute in the `RACECAR` class linking it back to the `CAR` class and using the notation `p` in place of the cardinality to indicate that the link refers to the parent class.

Figure 5-15 shows a `CAR` class and a `RACECAR` subclass. In this case, the `RACECAR` class is said to inherit from the `CAR` class. `CAR` is called the *parent class*, *superclass*, or *supertype*.

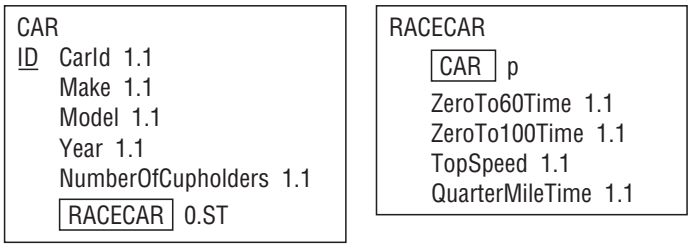

Figure 5-15

In more complicated models, a class can have multiple subclasses, nested subclasses, or multiple parent classes.

For example, suppose you decide you also want to store information about motorcycles. Motorcycles and cars share some information but one isn't really a special type of the other, so you create a new `VEHICLE`

class to hold the common features. You then pull the common attributes from the CAR class into VEHICLE and make both CAR and MOTORCYCLE subclasses of VEHICLE. In this example, you have multiple classes (CAR and MOTORCYCLE) inheriting from a common parent class (VEHICLE). You also have a nested class RACECAR inheriting from the CAR subclass.

Comments and Notes

Semantic object models are fairly good at capturing the basic classes involved in a project, and through object attributes they do a decent job of showing which classes are related to other classes. However, they don't capture every possible scrap of information about a project.

For example, semantic object models don't indicate an attribute's domain. There's nothing in Figure 5-15 that shows that the CAR class's Make attribute must take values from an enumerated list (Ford, GM, Yugo, De Lorean, and so forth), that Model must come from a list that depends on Make, and that NumberOfCupholders should be an integer between 0 and 99 (some of the bigger minivans may need three-digit numbers).

For an even stranger example, suppose you build a VOLLEYBALL_TEAM class to represent volleyball teams. Depending on the tournament, a volleyball team might have 2, 4, or 6 players but other values are not allowed. (Although I've seen some really weird formats including as the ''executive retreat'' event where as many 12 people wearing slacks and dress shirts but no shoes squeeze onto the court.) A semantic object model lets you specify a minimum and maximum for the PLAYER object attribute but it cannot handle the special case of 2, 4, or 6.

A semantic object model also doesn't necessarily capture all of the meaning of the relationships between classes. For example, suppose you build BAND and ARTIST classes to store information about your favorite heavy metal bands. You would like to make separate fields in the BAND class to represent lead vocal, lead guitar, lead trombone, and other key band members but, because these are all object attributes, you need to represent them in the model as ARTIST. You'd really like to make LeadVocal, LeadGuitar, and LeadTrombone attributes that have as their domain ARTIST objects.

Though you cannot make those kinds of attributes, you can jot down notes saying what each of the ARTIST objects in the BAND class represent. You can add them as a footnote to the class, in a separate document, or in any other way that will make it easy for you to remember the meanings of these associations.

> *Note that you can also work around this problem by making an association class* BAND_MEMBER *that has a* Role *attribute in addition to* BAND *and* ARTIST *object attributes. Then, for example, you could use a* BAND_MEMBER *object to associate the* BAND *Spiñal Tap with the* ARTIST *David St. Hubbins with* Role *set to* Lead Vocal.

Remember that the point of a semantic model (or any model for that matter) is to help you understand the problem. If the model alone doesn't capture the full scope of the problem, add comments, notes, attachments, video clips, dioramas, and other extras. The model can only do so much and if it's missing something, write it down. You may not need this information now to build the initial model, but you'll need it later to build the database so write it down.

Entity-Relationship Models

An *entity-relationship diagram* (ER diagram or ERD) is another form of object model that in many ways is similar to a semantic object model. It also allows you to represent objects and their relationships, although

it uses different symbols. ER diagrams also have a different focus, providing a bit more emphasis on relations and a bit less on class structure.

The following sections explain how to build basic ER diagrams to study the entities and relationships that define a project.

Entities, Attributes, and Identifiers

An *entity* is similar to a semantic object. It represents a specific instance of some thing that you want to track in the object model. Like semantic objects, an entity can be a physical thing (employee, work order, espresso maker) or a logical abstraction (appointment, discussion, excuse).

Similar entities are grouped into *entity classes* or *entity sets*. For example, the employee entities Bowb, Phrieda, and Gnick belong to the Employee entity set.

Like semantic objects, entities include attributes that describe the object that they represent.

There are a couple of different methods for drawing entity sets. In the first method, a set is contained within a rectangle. Its attributes are drawn within ellipses and attached to the set with lines. If one of the attributes is an identifier (also called a *key* or *primary key*), its name is underlined. Figure 5-16 shows a simple `Employee` entity set with three attributes. (Some developers write entity set names in ALL CAPS, whereas others use Mixed Case.)

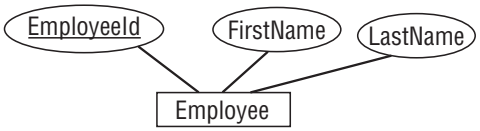

Figure 5-16

One problem with this notation is that it takes up a lot of room. If you add all of the attributes to the `Employee` class (`EmployeeId`, `FirstName`, `LastName`, `SocialSecurityNumber`, `Street`, `Suite`, `City`, `State`, `ZipCode`, `HomePhone`, `CellPhone`, `Fax`, `Email`, and so forth), you'll get a pretty cluttered picture. If you then try to add `Department`, `Project`, `Manager`, and other classes to the picture with all of their attributes, you can quickly build an incomprehensible mess.

A second approach is to draw entity sets in a manner similar to the one used by semantic object models and then place only the set's name in the ER diagram. Lines and other symbols, which are described shortly, connect the entity sets to show their relationships. This approach allows you greater room for listing attributes while removing them from the main ER diagram so it can focus on relationships.

Relationships

An ER diagram indicates a relationship with a diamond containing the relationship's name. The name is usually something very descriptive such as `Contains`, `Works For`, or `Deceives`, so often the relationship is perfectly understandable on its own. If the name isn't enough, you can add attributes to a relationship just as you can add them to entities: by placing the attribute in an ellipse and attaching it to the relationship with a line.

Normally entity names are nouns such as `Voter`, `Person`, `Forklift`, and `Politician`. Relationships are verbs such as `Elects`, `Drives`, and `Deceives`. When you see entities and relationships connected in an

ER diagram, they appear as easy-to-read caveman phrases such as `Voter Elects Politician`, `Person Drives Forklift`, and `Politician Deceives Voter`.

Figure 5-17 shows the `Person Drives Forklift` relationship.

Figure 5-17

Note that every relation implicitly defines a reverse relation. The phrase `Person Drives Forklift` implicitly defines the relation `Forklift IsDrivenBy Person`. Usually you can figure out the relation's direction from the context. You can help by drawing the relationships from left-to-right and top-to-bottom whenever possible.

I've also seen ER diagrams that include arrows above or beside a relationship to show its direction. For example, Figure 5-18 shows an ER diagram that includes three objects and two relationships. The arrows make it easier to see that `Customer Places Order` and `Shipper Ships Order`.

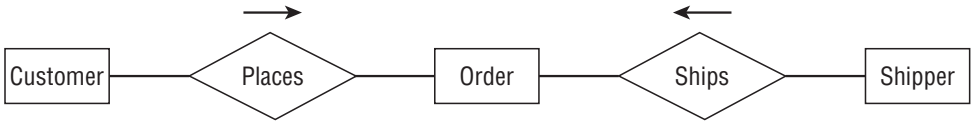

Figure 5-18

Cardinality

To add cardinality information, ER diagrams add one or more of three symbols to the lines leading in and out of entity sets. The three symbols are:

- **ring:** A ring (or circle or ellipse) means zero.
- **line:** A short line (or dash or bar) means one.
- **crow's foot:** A crow's foot (or teepee or whatever you call it) means many.

These aren't too hard to remember because the number 0 looks like a circle, the number 1 looks a line, and the crow's foot looks like several 1s.

If two of these symbols are present, they give the minimum and maximum number of entities that can be associated with the relation. For example, if the line entering an entity includes a circle and line, then zero or one of those items is associated with the relation.

For a concrete example, consider Figure 5-19. The relationship `Swallows` connects the classes `SwordSwallower` and `Sword`. The two lines beside `SwordSwallower` mean that the relationship involves between 1 and 1 `SwordSwallower`. In other words, the relationship requires exactly one `SwordSwallower`.

The circle and crow's foot beside `Sword` mean that the relationship involves between 0 and many swords. That means this is a one-to-many relationship.

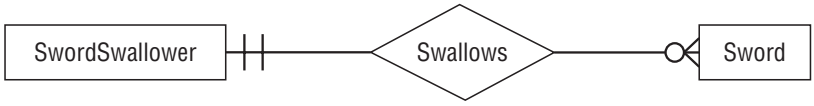

Figure 5-19

ER diagrams only have three symbols for representing three cardinalities: 0, 1, and many. (It reminds me of those primitive tribes that only have words for the numbers 1, 2, and many. I wonder if they played a role in developing ER diagrams?) This means you cannot specify cardinality as precisely as you can with semantic object models, which let you explicitly give upper and lower bounds.

For example, suppose you want to represent 2 to 4 jugglers juggling 5 or more flaming torches. (It's hardly juggling if two people just stand there holding four torches. Even I could do that, if they're not too heavy.) In a semantic object model, you would give the jugglers the cardinality 2.4 and the torches 5.N. Because ER diagrams don't have symbols for 2, 4, or 5, you're out of luck if you're building an ER diagram.

But wait! The point of these models is to gain an understanding of the system, not to rigidly follow the rules to their ridiculous conclusions, so I see no reason why you shouldn't merge the best of both systems and use ER diagrams that specify cardinality in the semantic object model style.

Figure 5-20 shows how I would model the jugglers. You won't find many people who use this combined notation on the Internet so you should understand the normal ER symbols, too, but this version seems easy enough to understand.

Figure 5-20

Inheritance

Like a semantic object model, an ER diagram can represent inheritance. An ER diagram represents inheritance as a special relationship named `IsA` (read as ''is a'') that's drawn inside a triangle. One point of the triangle points toward the parent class. Other lines leading into the triangle attach on the triangle's sides.

For example, a space shuttle crew contains several different kinds of astronauts including Commander, Pilot, Mission Specialist, and Payload Specialist. All of these have the common crew member attributes plus additional attributes that relate to their more specialized roles. For example, a Commander, Pilot, and Mission Specialist have special NASA space training (I'll call them ''space trained'').

A Payload Specialist is a doctor, physicist, database design book author, or other professional who comes along for the ride to perform some specific mission such as watching spiders spin webs in microgravity.

Figure 5-21 shows one way you might model this inheritance hierarchy in an ER diagram. The `PayloadSpecialist` inherits directly from `Astronaut`. `SpaceTrained` also inherits from `Astronaut`, although the relationship diagram probably will include only subclasses of `SpaceTrained` and not any `SpaceTrained` entities. `Commander`, `Pilot`, and `MissionSpecialist` inherit from `SpaceTrained`.

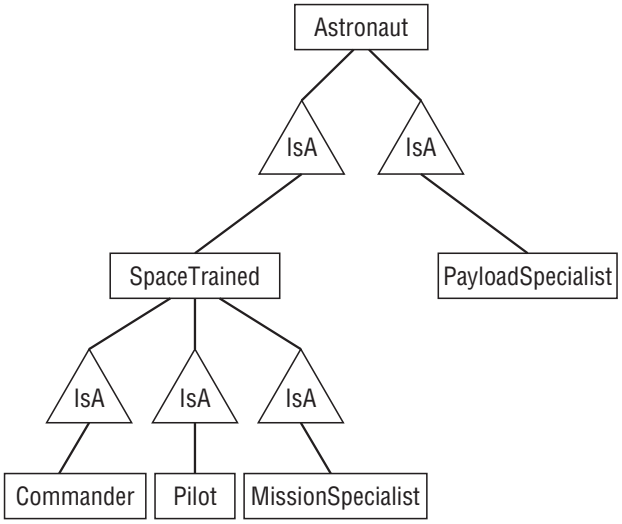

Figure 5-21

Sometimes you may see the `IsA` symbol shared by more than one inherited entity. The result implies a sibling relationship that probably doesn't mean much (for example, `SpaceTrained` and `PayloadSpecialist` are related only by the fact that they inherit from a common parent entity) but it does make the diagram less cluttered.

Figure 5-22 shows the same inheritance diagram shown in Figure 5-21 but with this new notation.

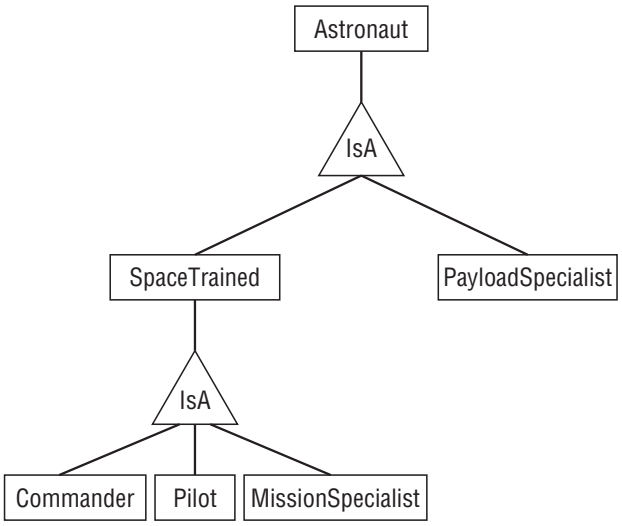

Figure 5-22

Try It Out ER Diagrams

Make an ER diagram to represent the `Passenger`, `Driver`, and `Car` entities.

1. Make a `Person` class with `PersonId`, `FirstName`, and `LastName` fields.
2. Show `Passenger` and `Driver` inheriting from `Person`.
3. Display the relationships between the `Driver` and `Passenger` classes and the `Car` class.

How It Works

1. Make a `Person` class with `PersonId`, `FirstName`, and `LastName` fields.

   Draw `Person` in a rectangle. Attach ellipses holding `PersonId` (underlined because it's the key), `FirstName`, and `LastName`.

2. Show `Passenger` and `Driver` inheriting from `Person`.

   Place a triangular `IsA` symbol below `Person`. Draw lines out of the bottom of that symbol to connect to the `Driver` and `Passenger` classes.

3. Display the relationships between the `Driver` and `Passenger` classes and the `Car` class.

   Connect `Driver` with `Car` via a `Drives` relationship. This relationship must involve exactly one `Driver` and one `Car`. (This model doesn't allow backseat drivers.)

   Connect `Passenger` with `Car` via a `Rides In` relationship. This relationship must involve exactly one `Car` but may involve any number of `Passengers` (even none).

   Figure 5-23 shows the finished diagram.

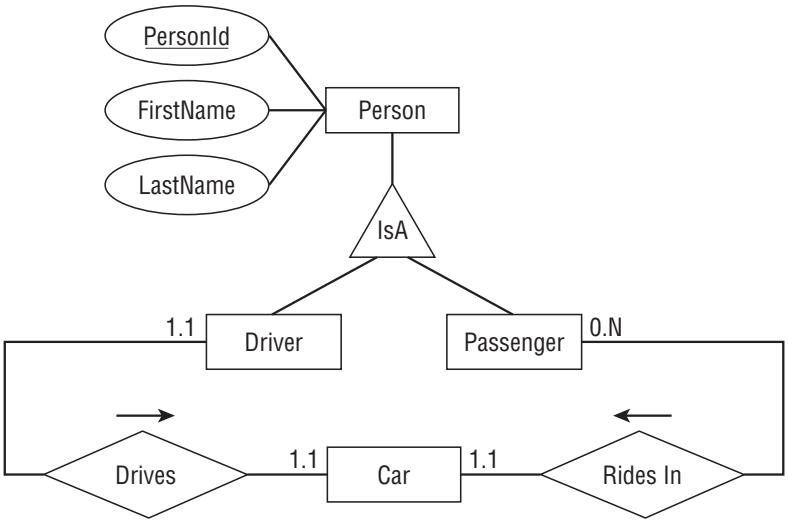

Figure 5-23

Additional Conventions

ER diagrams use a few other conventions to add fine shades of meaning to a model.

If every entity in an entity set *must* participate in the relationship, the diagram includes a thick or double line. This is called a *participation constraint* because each entity must participate.

For example, consider the `Pilot Flies Airplane` relationship. During flight, every airplane must have a pilot (otherwise it's called a ''smoking pile of metal'' instead of an ''airplane''). This is a participation constraint on the `Airplane` entity set because all entities in that set must participate in the relationship (that is, have a pilot).

If an entity can participate in *at most one* instance of the relationship set, the diagram uses an arrow to connect the entity to the relationship. This is called a *key constraint*. For example, during flight a pilot can fly at most one airplane so the `Pilot` entity set has a key constraint on the `Flies` relationship. (Although I suppose a pilot could throw a paper airplane while in the cockpit and thus fly two planes at the same time.)

If an entity must be involved in exactly one instance of a relationship set, it gets a thick or double arrow to indicate both participation and key constraints. For example, during flight an airplane must have one and only one pilot so it would get the thick or double arrow.

Figure 5-24 shows the `Pilot Flies Airplane` relationship. Each `Pilot` can fly at most one airplane so `Pilot` is connected to the relationship with an arrow (key constraint). A `Pilot` might sometimes be a passenger who's not flying the airplane so there's no participation constraint on `Pilot` for this relationship. On the other side of the relationship, the `Airplane` must have one and only one `Pilot` so it gets the double arrow to indicate both key and participation constraints. The cardinalities are between 1 and 1 for both entities because there's a one-to-one relationship between `Pilot` and `Airplane` (ignoring copilots) in this relationship.

Figure 5-24

A *weak entity* is one that cannot be identified by its attributes alone. For example, consider a database to store submarine race results. A `Race` entity holds information about particular race. A `Result` entity holds information about how a submarine performed in a race. The `Result` entity has attributes to store a reference to the `Race` entity, a reference to a `Sub` entity, and result information such as `Time`, `FinishPosition`, and `TorpedoesFired`.

Alone, there's no reasonable way to find a specific `Result` entity. There is no combination of `Result` attributes that really makes sense as a search key. You could search for a combination of `Time` and `FinishPosition` but that doesn't identify a particular `Result`.

Instead you would either search for a particular `Race` and use it to find its associated `Results`, or search for a particular `Sub` and use it to find its associated `Results`.

In an ER diagram, you draw a weak entity with a thick rectangle and connect it to its identifying relationship with a thick arrow. Figure 5-25 shows the `Race`, `Sub`, and `Result` entity sets and their relationships.

Figure 5-25

Comments and Notes

As is the case with semantic object models, you shouldn't be afraid to add notes, comments, scribbles, and anything else to make an ER diagram easier to understand. Annotate entity set definitions to show the domain and cardinality of an entity's attributes. Add notes to further explain confusing entities and relationships.

The purpose of an ER diagram is to help you understand a project, not to become a technically correct but uninformative doodle.

Relational Models

Chapter 3 explained basic concepts of relational databases such as tables, tuples, rows, and columns. (If you don't remember Chapter 3, go back and skim through it quickly to refresh your memory.)

Converting semantic object models and ER diagrams into a relational version isn't too difficult once you know how the concepts described in Chapter 3 map to those described so far in this chapter. The following table shows the how key terms from Chapter 3 map to the terms used in semantic object models and ER diagrams.

Theory	Database	File	SOM	ER
Relation	Table	File	Class	Entity Set
Tuple	Row	Record	Object	Entity
Attribute	Column	Field	Attribute	Attribute

To convert semantic object models and ER diagrams into relational models, you simply map the classes or entity sets to tables. You then figure out which columns in the tables form the foreign key relationships among the tables.

The following sections work through examples of converting SOM and ER models into relational ones.

Converting Semantic Object Models

Consider the simple semantic object model shown in Figure 5-26. A `CUSTOMER` object has one or more `Addresses`, one or more `CONTACT`s, and one or more `ORDER`s. The `CONTACT` class contains only simple attributes. The `ORDER` class contains a simple `Date` and a group attribute to hold information about `Item`s ordered.

This model leads immediately to three relational tables: Customers, Contacts, and Orders.

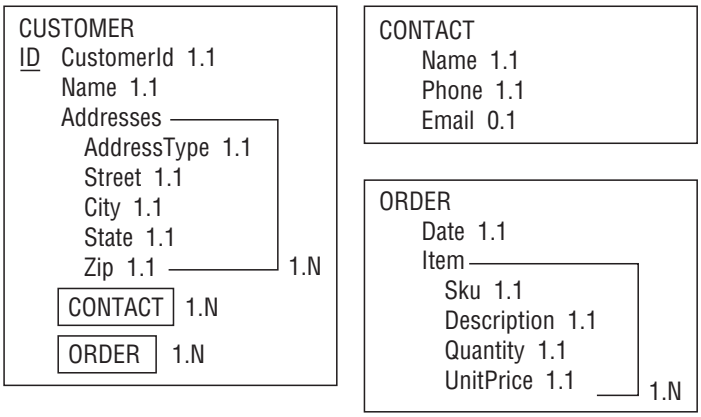

Figure 5-26

If the semantic object model includes inheritance relationships, build a table for each of the object sets. Use the parent class's primary key as a foreign key in the child class to connect the two in a one-to-one relationship. For example, if `CUSTOMER` inherits from `PERSON`, add a PersonId field in the Customers table to associate the corresponding records in the two tables.

The `CUSTOMER` class's `CONTACT` and `ORDER` attributes indicate that there should be a link from the Customers table to the Contacts and Orders tables. To do this, you can place foreign key fields in the Contacts and Orders tables to hold the CustomerId values of their corresponding Customer records. To make understanding the relational model easier, call those fields CustomerId so they match the name in the Customers table.

At this point, the relational model is practically finished. Only one little problem remains: a relational record cannot hold a potentially unlimited number of columns. In this case, a row in the Customers table cannot have an unlimited number of columns to hold multiple address values for every row. Similarly, the Orders table cannot have an unlimited number of columns to hold item data.

The solution is similar to the one used to allow a Customers record to correspond to multiple Contacts and Orders records. Create new tables to hold the repeated items. Then use foreign key fields to link those records back to their owning Customers and Orders records.

Figure 5-27 shows the resulting relational model.

Each table's primary key is underlined (only the Customers and Orders tables have primary keys).

Lines connect the fields that form foreign key relationships. The numbers at the ends of these lines give the numbers of items participating in the relationship (the infinity symbol ∞ means ''many''). In this example, all of the relationships are one-to-many relationships.

This diagram shows relationships among tables but doesn't show much other detail. In particular, it doesn't show the fields' data types or whether they are required. If you expand each table's representation, you can add some of this information. Figure 5-28 shows the same model with columns to show the fields' data types and whether each is required.

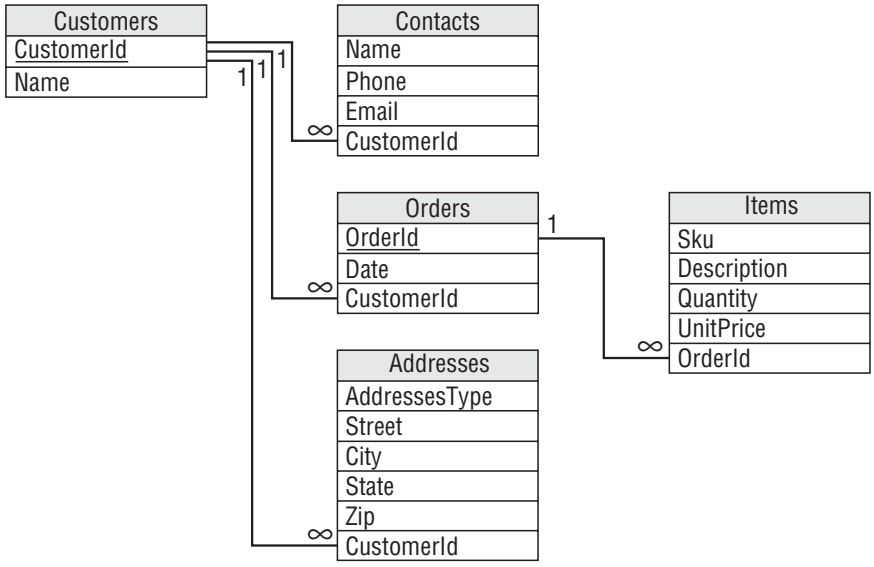

Figure 5-27

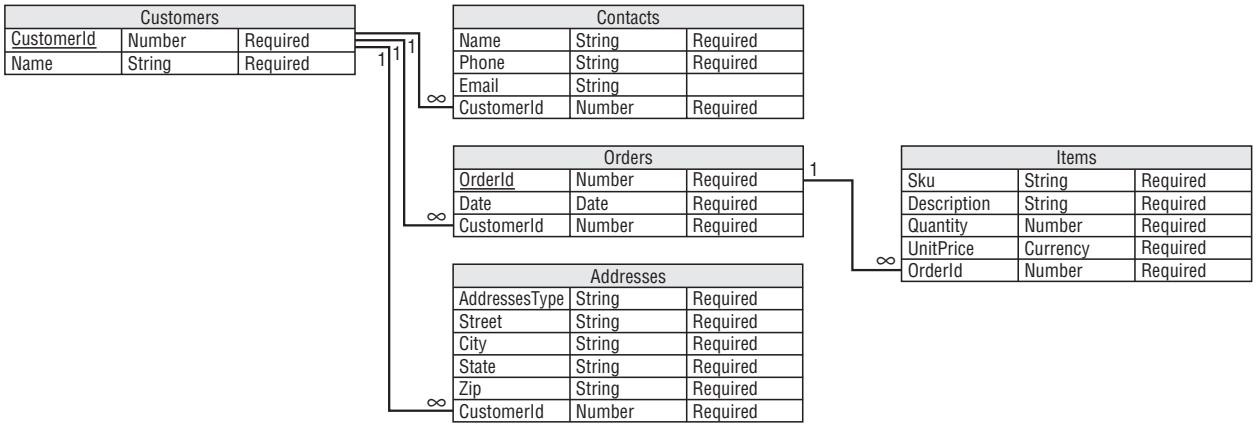

Figure 5-28

There's only so much information you can add to one of these diagrams, however. Even this relatively simple diagram is pretty big if you add data type and required data. Usually it's better to stick to the simpler version and put additional information in separate documents.

As is the case with all models, you should write down notes to record any information that is not fully captured by the diagram alone. For example, Figure 5-28 does not show which fields are required, their meanings (what does Sku mean, anyway?), more precise cardinalities (what if ''one-to-many'' should really be ''one-to-four''), and so forth.

Though the figure gives data types for each of the tables' fields, that does not necessarily completely specify the fields' domains. For example, the Zip field should contain a 5-digit ZIP Code or a Zip+4 Code

similar to 12345-5678, UnitPrice should be a positive monetary value, and the Email field should hold a properly formatted email address such as `comments@whitehouse.gov`.

You should write down all of these and any other constraints that are not obvious from the diagram. (In case you're curious, Sku stands for ''stock keeping unit'' and is pronounced ''skew.'' It's like a serial number you can use to identify products.)

Converting ER Diagrams

Figure 5-29 shows an ER diagram that covers a situation similar to the one modeled by semantic object model shown in Figure 5-26.

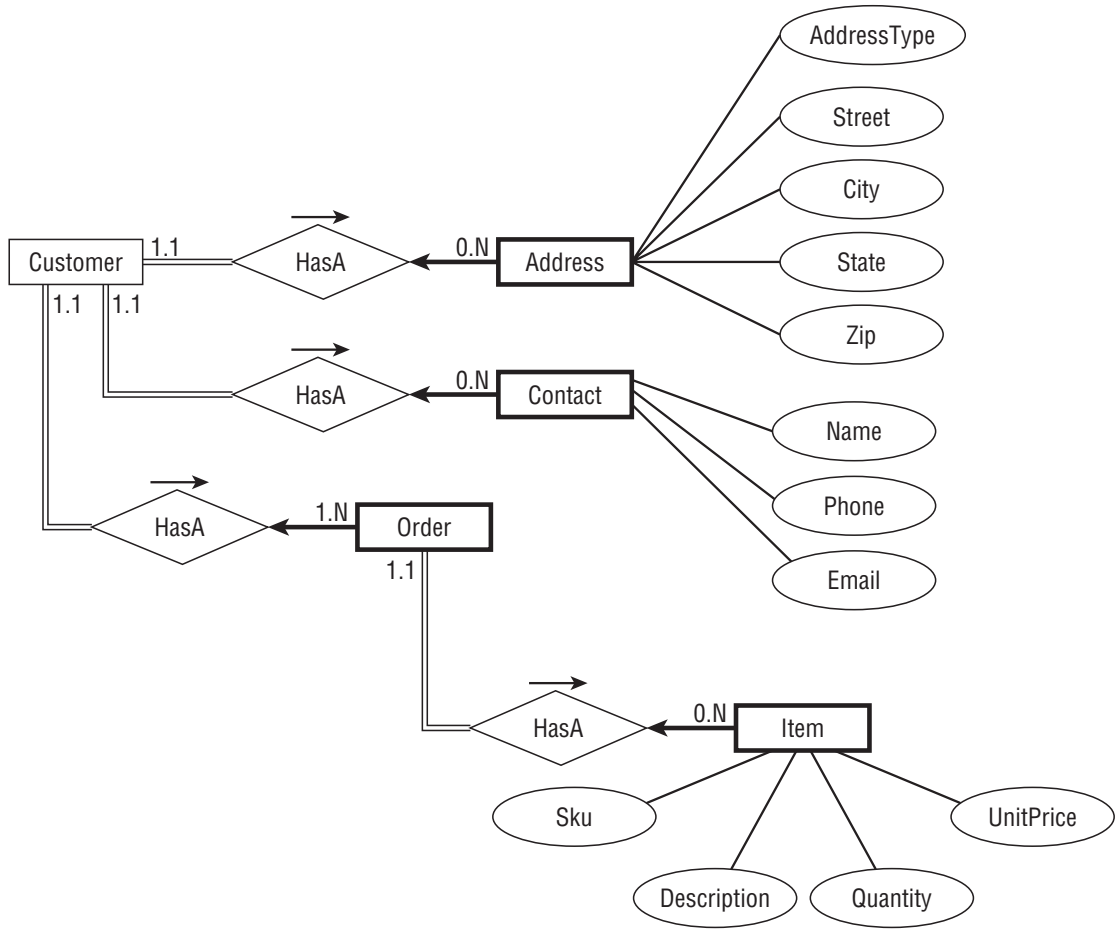

Figure 5-29

Each `Customer` entity has at least one `Address`, `Contact`, and `Order`. Those are all participation constraints so they are drawn with double lines.

The `Address`, `Contact`, and `Order` entities are accessed through their corresponding `Customer` entities. That makes them weak entities so they are drawn with thick rectangles and they have thick arrows

pointing to their identifying relationships. (If you want to allow the users to search for orders directly, perhaps by an `OrderId`, then `Order` would not be a weak entity.)

The `Order` entity must be associated with at least one `Item` so it has a participation constraint drawn with a double line. The `Item` entity is also weak so it is drawn with a thick rectangle and it uses a thick arrow to connect to its identifying relationship.

The entities in the ER diagram lead directly to the relational tables Customers, Addresses, Contacts, Orders, and Items.

To connect a weak entity with its owner, make sure the owner's table has a primary key. Then add a foreign key field to the weak entity's table that refers back to the owner's primary key.

The resulting relational model is the same as the one generated by the semantic object model and is shown in Figure 5-27.

You can handle inheritance the same way you did for semantic object models. Build a table for each of the entities. Use the parent class's primary key as a foreign key in the child class to connect the two in a one-to-one relationship. For example, if `Politician` inherits from `Weasel`, then add a WeaselId field in the Politicians table to link the corresponding records in the two tables.

As is the case when translating a semantic object model into a relational model, you will need to write down any extra conditions, constraints, or other information that is not completely captured by the model. See the end of the previous section for some examples of things you might want to write down.

Summary

Different kinds of models help define a problem. They identify the entities that are significant to the problem and they clarify the relationships among those entities. You can then use the models to test your understanding of the problem and to verify that the models provide the data you need to satisfy the problem's use cases and other requirements.

This chapter explained how to build different kinds of models.

In this chapter you learned how to:

- Build user interface models to learn what kind of data the database will need to store.
- Build semantic object models to study the objects that will interact while solving the problem.
- Build entity-relationship diagrams to study the entities that are involved in the problem and to examine their interactions.
- Convert semantic object models and entity-relationship diagrams into relational models.

After you've built a relational model, you can use it to start building a database. Before you begin, however, there are several techniques that you can use to make the model more efficient. The first of these techniques, extracting business rules, is described in the following chapter.

Before you move on to Chapter 6, 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

1. Draw a semantic object model for a small college with the classes `STUDENT`, `INSTRUCTOR`, `COURSE`, and `PROJECT`. The rules are:

   a. All students must be enrolled in at least one course or one project (or they're dropped).
   b. Similarly an instructor must teach at least one course or supervise at least one project.
   c. A student cannot be working on more than one project (they're too time-consuming).
   d. An instructor can teach any number of courses and supervise any number of projects.
   e. A project or course must have an instructor.
   f. A course must have at least 5 students (or it's canceled).
   g. A project must have between 1 and 5 students.
   h. `STUDENT` and `INSTRUCTOR` should be subclasses of a `PERSON` class that contains common elements such as name, address, and phone number.
   i. Student data must include past courses and projects, and grades for them.

   Write down any special conditions and features that the semantic object model cannot handle with its normal notation.

2. Draw two ER diagrams for the situation described in Exercise 1, one to show the inheritance relationships and one to show the main entity relationships. Write down descriptions of any constraints and any special conditions that are not represented by the diagram alone.

3. Convert either the semantic object model that you built for Exercise 1 or the ER diagram you built for Exercise 2 into a relational model.

4. Mike's Trikes sells tricycles. Not the little kiddie models, the giant motorized half-ton behemoths you occasionally see on the road that are somewhere between a motorcycle with an extra wheel and a car with one missing.

   Draw a semantic object model for Mike with the classes `CUSTOMER`, `SALESPERSON`, `MANAGER`, `CONTRACT`, `PAYMENT`, and `SHIFT`. Use the following assumptions:

   a. `CUSTOMER` and `SALESPERSON` are subclasses of the `PERSON` class that holds contact information (name, address, phone). `MANAGER` is a subclass of `SALESPERSON`.
   b. A salesperson sells a payment contract to a customer. The salesperson gets a commission so you need to keep track of who sold the contract.
   c. A customer doesn't have a record until that customer buys a contract.
   d. `SHIFT` objects track dates and times that a salesperson works.
   e. Customers make payments that should be subtracted from the customer's balance. A `PAYMENT` object should record the payment's date and amount, and the customer who made it.

f. You should be able to find all of the contracts that a particular salesperson sold.

g. You should be able to find all of the contracts that a particular customer purchased. You should also be able to check the customer's current balance.

Write down any special conditions and features that the semantic object model cannot handle with its normal notation.

5. Draw two ER diagrams for the situation described in Exercise 3, one to show the inheritance relationships and one to show the main entity relationships. Write down descriptions of any constraints and any special conditions that are not represented by the diagram alone.

6. Convert either the semantic object model that you built for Exercise 4 or the ER diagram you built for Exercise 5 into a relational model.

7. Suppose you want to make a database to represent your most expensive purchases. These include your house and vehicles so you make `HOUSE` and `VEHICLE` classes. You decide to expand the model to include `CAR` and `TRUCK` classes. Then you buy a camper. Because it shares attributes with both `HOUSE` and `TRUCK`, you decide that it should inherit from both of those classes.

   Draw a semantic object model showing these inheritance relations. Add a few additional non-object attributes of your choosing to each class.

8. Draw an ER diagram representing the inheritance hierarchy described in Exercise 7.

6

Extracting Business Rules

Chapter 5 explained how to build models to represent the entities involved in a database project and to study the interactions among those entities. The final kind of model described in that chapter, the relational model, has a structure that closely mimics the organization of a relational database. You can easily convert a relational model into a working relational database.

Before you do, however, you should optimize the relational model to make the final database as flexible and efficient as possible. Optimizing the model now is easier than reorganizing the database later, so it's worth taking some time to make sure you get the database design right the first time.

The first step in optimizing the database is extracting business rules. Keeping business rules separate from other database constraints and relations, at least logically, makes later changes to the database easier.

In this chapter you learn:

- Why business rules are important.
- How to identify business rules.
- How to modify a relational model to isolate business rules.

After you understand business rules, you'll be able to use them to make the database more flexible and easier to maintain.

What Are Business Rules?

Business rules describe the objects, relationships, and actions that a business finds important and worth writing down. They include rules and policies that define how a business operates and handles its day-to-operations. They generally help a business satisfy its goals and meet its obligations.

For example, some general business rules might be:

- The nearest clerk greets customers by saying ''Welcome to Cloud Nine'' when they enter the store.
- Clerks ask to see a customer's ID when writing a check for more than $20 or charging more than $50. No signature is required when charging less than $25.
- Whoever unlocks the door in the morning makes the first pot of coffee (or risks mutiny).
- Save the good scotch for the landlord.

Because this is a database design book, this chapter is only concerned with the database-related business rules. Some examples of those are:

- Don't make a Customer record until the customer buys something and has an associated order.
- Customer records must have first and last names. (If Bono, Everlast, or Madonna buys something, get an autograph and make up a last name.)
- If a student doesn't enroll in at least one class, change the Status field to Inactive.
- If a salesperson sells more than 10 hot tubs in one month, award a $200 bonus.
- All Contact records must have at least one phone number or email address.
- If an order totals more than $100 before taxes and shipping, give a 10 percent discount.
- If an order totals more than $50 before taxes and shipping, give free shipping.
- Employees get a 1 percent discount.
- If the in-stock quantity of an inventory item drops below the number of items sold during the last month, order more.

From a database point of view, business rules are constraints. Some are simple constraints such as:

- All orders must have a ContactPhoneNumber.

Simple rules such as this one map easily to the features provided by a relational database. It's easy to indicate that a field has a certain data type or that it is required (as in this case).

Other business rules may represent quite complex constraints such as:

- A student's number of course hours plus number of project hours must be between 1 and 14.

You can implement some of these more complex rules with check constraints or foreign key constraints. Recall from Chapter 3 that check constraints include field-level constraints that apply to a single field in a table, and table-level constraints can examine more than one field in the same record.

Still other business rules are even more complex:

- An instructor must have a combination of classes, labs, and office hours totaling at least 30 contact hours with up to 1/2 office hour per hour of class, 1 office hour per hour of lab, and thesis supervision counts as 2 hours.

This constraint may require you to gather data from several different tables. This kind of very complex check is probably best performed by code either in the database itself or in external software.

All of these rules are implemented as constraints in one form or another, whether as easy database features (requiring a field), as harder database features (check constraints and foreign keys), or in code (inside or outside of the database).

Identifying Key Business Rules

Writing down all of the business rules is worthwhile in its own right so you can make sure they all get implemented somehow in the database. It's also worth categorizing the business rules so you can build them properly.

How you implement a business rule depends not only on how tricky it is to build, but also on how likely it will be to change later. If a rule is likely to change later, you may be better off building it by using a more complicated but more flexible method.

For example, suppose you only ship orders to states where you have a warehouse and those include Wyoming, Nebraska, Colorado, and Kansas. A business rule requires that the State field in an order's shipping address must be WY, NE, CO, or KS. You can implement this as a simple field-level check constraint in the Orders table. Three minutes' work and you're a hero! No big deal.

But now suppose you open a new warehouse in Utah. To allow for this change, you'll need to edit this check constraint. This isn't the end of the world, but this change requires that you modify the structure of the database.

Now suppose the company policy changes so some warehouses are allowed to ship to certain other states. You'll need to change the database's check constraints again to allow for the change. This still isn't the end of the world, but once more you're required to change the structure of the database to accommodate a change to a business rule.

Now consider an alternative approach. Suppose instead of making this business rule a field-level check constraint on the State field, you make it a foreign key constraint. You create a ShippingStates table and fill it with the values WY, NE, CO, and KS. Then you make the Orders table's State field a foreign key referring to the new States table. Now the Orders table will accept only records that have a State value that is listed in the States table.

If you need to change the states that are allowed, you only need to add or remove records from the States table. Admittedly the difference in difficulty between this approach and the previous one is small. The previous approach required changing the database's structure, whereas the new approach only requires changing the data.

Not only does changing the data take a bit less effort, but it also requires less skill. This rule implemented as a check constraint might look like this:

```
State = 'WY' Or State = 'NE' Or State = 'CO' Or State = 'KS'
```

This isn't terribly difficult code, but it is code and only someone familiar with database programming will be able to make changes to it.

Data in the States table, however, is relatively easy to understand. Even your customers can add entries to this table (possibly with a few hints from you).

Placing the validation data in a separate table also allows the users to understand it more easily. Most users would be intimidated by the previous check constraint (even if they can find it), but they can easily understand a list of allowed values in a table.

To identify these key business rules, ask yourself two questions. First, how easy is it to change a rule? If a rule is very complex, it will be difficult to change without messing it up. If implementing the rule is as simple as making a field required or not in a table, you won't lose a huge amount of time if the customer later decides that the Lumberjacks table's PreferredAxe field isn't required after all.

Second, how likely is the rule to change? If a rule is likely to change frequently, it's probably worth some extra planning to make changing the rule easier.

Types of rules that make good candidates for extra attention include:

- **Enumerated values:** For example, allowed shipping states, order statuses (Pending, Approved, Shipped), and service names (Installation, Repair, Dog Washing).
- **Calculation parameters:** For example, suppose you give free shipping on orders over $50. Will you later change that to $75? $100?
- **Validity parameters:** For example, suppose full-time students must take between 8 and 16 credits. Will we ever make this 12 to 16 hours? 8 to 20 hours? Or suppose you require that all projects include between 2 and 5 students. Will you ever want to allow a single student to have a project? Or will you allow a bigger team if a group of friends wants to work together badly enough to bribe you with donuts and latte?
- **Cross-record and cross-table checks:** These kinds of checks are more complicated. For example, you might require that the date and time of a poker game be after the date the tournament started. (Although the Olympics schedules competitions before the opening ceremony. They probably use some sort of time-warp effect at international levels.)
- **Generalizable constraints:** If you think you may need to apply a similar constraint later, you should think about generalizing the constraint and moving it out of the database proper. For example, suppose your buyer slipped a decimal point and ordered 100 sets of crampons (those spiky things that ice climbers wear on their boots) instead of 10. To move the excess inventory, you offer a $50 bonus to any salesperson who can sell 10 pairs in a week. That's fine, but next month you might end up with an excess inventory of ice axes. After you fire your buyer, you might want to change the incentive to give a $30 bonus to any salesperson who sells 5 ice axes. You can make these changes easier if you pull the product name or ID, number of sales, bonus amount, and duration (weekly) out into another table and then use those parameters to calculate bonuses.

- **Very complicated checks:** Some checks are so complex that it's just easier to move them into code, either stored within the database or in external code modules. For example, suppose you can only register for the course Predicate Calculus (in the Mathematics department) if you have already taken (and passed) Propositional Calculus (in the Mathematics department) or Logic I and Logic II (in the Philosophy department). Or you have the instructor's permission. Or your advisor's. You can probably implement this as a table-level check constraint, but it may be worth thinking about moving this rule somewhere else, particularly because you may be able to generalize it to handle prerequisites for other courses.

Types of rules that are usually not worth special attention and can be just implemented directly in the database include:

- **Enumerated types with fixed values:** Though it might make sense to move allowed values for a State field into a new table, it probably doesn't make sense to do the same for a Handedness field. Unless you're planning to start marketing to octopi and squids, Left Handed and Right Handed are probably the only values you'll ever need for this field.
- **Data type requirements:** Requiring specific data types for a field is one of the bigger advantages to using a database. It hardly ever makes sense to use a very generic data type such as string because you're not sure whether the field will need to be a currency amount or a date. If you are that unsure, you probably need to study this field some more or split it into multiple fields.
- **Required values:** If a GolfRound record really needs a Caddie entry (so the golfer knows who to blame for using the 3-wood on that 124-yard par 3 hole), just make it required and worry about something more complicated.
- **Sanity checks:** For example, all inventory items should have a price of at least 0. You might want to allow 0 cost for loss leaders (or perhaps not) but if I ever find a store that sells a product for less than nothing (that is, pays me), I'm going down there with a dump truck and cleaning them out. (Now that I think about it, I've bought a few products that would have been overpriced at negative amounts. I might have to think a bit harder depending on what the product is.) If the sanity checks are so broad that they'll never need to be changed, just wire them in and don't worry about it.

Somewhere in the middle ground are business rules that have never changed in the past but that you cannot swear won't change in the future. They may be easy to implement as checks within the database but there still might be some advantage to extracting them to accommodate changes.

For example, suppose you require that Resident Advisors (RAs) have passed all of their general education requirements. It's been that way for five years, but before that the rule was different. Chances are the rule won't change again, at least not for a long time, but you never know. There has been talk about exempting RAs from the writing requirement ('cause riting ain't emportunt enuff).

In cases such as this one, you need to rely on the judgment of those who make the rules. Then when the unexpected happens, you can blame them.

So write down all of the business rules you can discover. Include the domains of every field and any simple bounds checks such as Price > 0 in addition to more complicated rules.

Group the rules by how likely they are to change and how hard they would be to change. Then take a closer look at the ones that are likely to change and that will be hard to change and see if you shouldn't pull them out of the database's structure.

Try It Out Find the Business Rules

Consider this partial list of business rules for a custom woodworking shop:

- ❑ Accept no new orders if the customer has an unpaid balance on completed work.
- ❑ All customer records must include daytime and evening phone numbers.
- ❑ Always wear proper eye protection.
- ❑ Clean the shop thoroughly at the end of the day.
- ❑ Create a customer record when the customer places his or her first order.
- ❑ Don't use non-portable power tools alone.
- ❑ Give a 10% discount if the customer pays in full in advance.
- ❑ If there is less than 1 pound of standard size screws, add them to the reorder list.
- ❑ If we have fewer than 3 bottles of glue, add it to the reorder list.
- ❑ Leave no power tool plugged in when not in use, even for a minute.
- ❑ No customer can ever have an outstanding balance greater than $10,000.
- ❑ No customers allowed in the painting area.
- ❑ Order 25% extra material for stock items.
- ❑ Replace a tool if you won't need it in the next hour. Replace all tools at the end of the day.
- ❑ Require half of an order's payment up front (so we can buy materials) if the order is more than $1,000.
- ❑ Walt is not allowed to use the nail gun. Ever!
- ❑ When we have fewer than 2 pounds of standard size nails, add them to the reorder list.

1. Identify the database-related rules and indicate when they would apply.
2. Identify the rules that are simple or that seem unlikely to change so they can be built into the database.
3. Identify the rules that may change or that are complicated enough to deserve special attention.

How It Works

1. The following are the database-related rules:
   - ❑ Accept no new orders if the customer has an unpaid balance on completed work.

     Applies when the customer tries to place a new order.
   - ❑ All customer records must include daytime and evening phone numbers.

     Applies when the customer places a new order.
   - ❑ Create a customer record when the customer places his or her first order.

     When the customer places a new order, try to look up the customer's record. If there is no record, create one.

- ❑ Give a 10% discount if the customer pays in full in advance.

  When the customer places a new order, give this discount if he or she pays in advance. (You should also mention the discount at this time.)

- ❑ If there is less than 1 pound of standard size screws, add them to the reorder list.

  When we use screws, check this. (In practice, we'll probably check weekly or whenever we notice we're running low.)

- ❑ If we have fewer than 3 bottles of glue, add it to the reorder list.

  When we use up a bottle of glue, check this.

- ❑ No customer can ever have an outstanding balance greater than $10,000.

  When the user tries to place a new order, check the outstanding balance and place the order on hold until the balance is under $10,000. Also when we receive a payment, check for orders on hold (so we can release them if the customer's balance is low enough).

- ❑ Order 25% extra material for stock items.

  When ordering supplies, see if an item is in stock and if so add 25% to the order.

- ❑ Require half of an order's payment up front (so we can buy materials) if the order is more than $1,000.

  When the user places a new order, check the cost and require this payment if necessary.

- ❑ When we have fewer than 2 pounds of standard size nails, add them to the reorder list.

  When we use nails, check this. (In practice, we'll probably check weekly or whenever we notice we're running low.)

2. The following rules are simple or seem unlikely to change, so they can be built into the database:

- ❑ Accept no new orders if the customer has an unpaid balance on completed work.

  This seems unambiguous and unlikely to change, although if we need an exception mechanism (for brother-in-law Frank), this rule cannot be built into the database's structure.

- ❑ All customer records must include daytime and evening phone numbers.

  This seems unambiguous and unlikely to change. (Will we ever need more than two phone numbers?)

- ❑ Create a customer record when the customer places his or her first order.

  This seems unambiguous and unlikely to change.

3. The following rules seem likely to change or are complicated enough to deserve special attention:

- ❑ Give a 10% discount if the customer pays in full in advance.

  The parameter ''10%'' might change.

- ❑ If there is less than 1 pound of standard size screws, add them to the reorder list.

  The parameter ''1 pound'' might change.

- ❑ If we have fewer than 3 bottles of glue, add it to the reorder list.

  The parameter ''3 bottles'' might change.

- No customer can ever have an outstanding balance greater than $10,000.

  The parameter ''$10,000'' might change. (Do we need an exception mechanism?)

- Order 25% extra material for stock items.

  The parameter ''25%'' might change.

- Require half of an order's payment up front (so we can buy materials) if the order is more than $1,000.

  The parameters ''half'' and ''$1,000'' might change.

- When we have fewer than 2 pounds of standard size nails, add them to the reorder list.

  The parameter ''2 pounds'' might change.

A few of these rules follow the pattern, ''If we have less than X, reorder.'' It might be worthwhile to generalize this rule and apply it to all inventory items. The item's record would have fields ReorderWhen (to indicate the quantity on hand that triggers a supply order) and ReorderQuantity (to indicate how much to order).

Extracting Key Business Rules

Now that you've identified the business rules that will be tricky to implement within the database or that may change frequently, pull them out of the database. There are a couple of standard approaches for doing that.

First, if the rule is a validation list, convert it into a foreign key constraint. Only shipping to a set of specific states is the perfect example. Simply make a States table, enter the allowed states, and then make the Orders table's State field be a foreign key referring to the States table.

Second, if the rule is a fairly straightforward calculation with parameters that may change, pull the parameters out and put them in a table. For example, if you want to give salespeople who sell at least $250,000 worth of cars in a month a $5 bonus, pull the parameters $250,000 and $5 out and put them in a table. In some businesses, you might even want to pull out the duration one month.

I've written several applications that had a special Parameters table containing all sorts of oddball parameters that were used to perform calculations, check constraints, and otherwise determine the system's behavior. The records had two fields: Name and Value. To see if a salesperson should get the bonus, you would look up the BonusSales parameter and see if his or her sales totaled at least that much. If so, you would look up the BonusAward parameter and give the salesperson that big a bonus. (This approach works particularly well when program code performs the checks. When the program starts, it can load all of the parameters into a collection. Later it can look up values in the collection without hitting the database.)

Third, if a calculation is complicated, extract it into code. That doesn't necessarily mean you need to write the code in C++, C#, Ada, or the latest programming language flavor-of-the-month. Many database products can store and execute stored procedures. A stored procedure can select and iterate through records, perform calculations, make comparisons, and do just about anything that a full-fledged programming language can.

So what's the point of moving checks into a stored procedure? Partly it's a matter of perception. Pulling the check out of the database's table structure and making it a stored procedure separates it logically from the tables. That makes it easier to divide up maintenance work on the database into structural work and programming work.

Of course you can also build the check into code written in a traditional programming language. You may be able to invoke that code from the database or you might use it in the project's user interface.

Finally, if you have a rule that you might want to generalize, well, you're going to have to use your judgment and imagination. For example, suppose an author of a database design book earns a 5% royalty on the first 5,000 copies sold, 7% on the next 5,000, and 10% on any copies after that. You could code those numbers into a stored procedure to calculate royalties but then later, when Steven Spielberg turns the book into a movie, you better believe the author will want better terms for the sequel.

Rather than writing these values into the code, put them in a table. In this case, those values are associated with a particular record in the Books table. You may want more or less than three percentage values for different royalty points so you'll need to pull the values into their own table (in ER diagram terms, the new table will be a weak entity with identifying relationship to the Books table).

Figure 6-1 shows a tiny part of the relational model for this database. To find the royalty rates for a particular book, you would look up the RoyaltyRates records for that book's BookId.

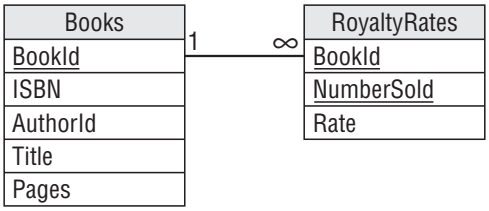

Figure 6-1

Now it will be a little more work calculating royalty payments than before (although you can still hide the details in a stored procedure), but it is easy to create new royalty schedules for future books.

Multi-Tier Applications

A multi-tier application uses several different layers to handle different data-related tasks. The most common form of multi-tier application uses three tiers. (The tiers are also often called layers, so you'll hear talk of three-layer systems.)

The first tier (often called the *user interface tier* or *user interface layer*) is the user interface. It displays data and lets the user manipulate it. It might perform some basic data validation such as ensuring that required fields are filled in and that numeric values are actually numbers, but it doesn't implement complicated business rules.

The third tier (often called the *data* or *database tier* or *layer*) is the database. It stores the data with as few restrictions as possible. Normally it provides basic validation (NumberOfRockets is required, MaximumTwistyness must be between 0.0 and 1.0) but it doesn't implement complicated business rules, either.

The middle tier (often called the *middle* or *business tier* or *layer*) is a service layer that moves data between the first and third tiers. This is the tier that implements all of the business rules. When the user interface tier tries to send data back to the database, the middle tier verifies that the data satisfies the business rules and either sends the data to the data tier or complains to the user interface tier. When it fetches data from the database, the middle tier may also perform calculations on the data to create derived values to forward to the user interface tier.

Figure 6-2 shows the three-tier architecture graphically.

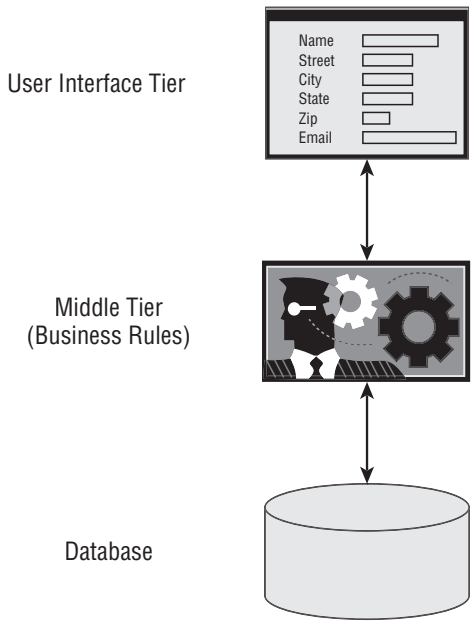

Figure 6-2

The main goal of a multi-tier architecture is to increase flexibility. The user interface and database tiers can work relatively independently while the middle tier provides any necessary translation. For example, if the user interface changes so a particular value must be displayed differently (perhaps in a dropdown instead of in a text box), it can make that change without requiring any changes to the database. If the database must change how a value is stored (perhaps as a string Small/Medium/Large instead of as a numeric size code), the user interface doesn't need to know about it. The middle tier might need to be adjusted to handle any differences but the first and third tiers are isolated from each other.

The middle tier also concentrates most of the business logic. The user interface and database perform basic validations but the middle tier does all of the heavy lifting.

Another advantage of multi-tier systems is that the tiers can run on different computers. The database might run one a computer at corporate headquarters, the middle tier libraries might run on a second computer (or even split across two other computers), and the user interface can run on many users' computers. Or all three tiers might run on the same computer. Separating the tiers lets you shuffle them around to fit your computing environment.

In practice, there's some benefit to placing at least some checks in the database tier so, if there's a problem in the rest of the application, the database has the final say. For example, if the user interface contains an obscure bug so customers who order more than 999 pencils on leap year day are charged $–32,767, the database can save the day by refusing that obviously harebrained price.

There's also some value to placing basic checks in the user interface so the application doesn't need to perform unnecessary round trips to the database. For example, it doesn't make a lot of sense to ship an entire order's data across the network to the corporate database only to have it rejected because the order is for –10 buggy whips. The user interface should be smart enough to know that customers cannot order less than zero of something.

Adding validations in both the user interface and the database requires some redundancy, but it's worth it. (Also notice that the user interface developers and database programmers can do their work separately so they can work in parallel.)

Although multi-tier architecture is far outside the scope of this book, it's worth knowing a little about it so you understand that there's another benefit to extracting complex business rules from the database's table structure. Even if you implement those rules in stored procedures within the database, you still get some of the benefits of a logical separation and flexibility almost as if you had a hidden extra tier.

Try It Out — Multi-Tier Applications

Consider the following database-related business rules:

- ❑ All Customers records must have at least one associated Orders record.
- ❑ All Orders records must have at least one associated OrderItems record.
- ❑ In a Customers record, Name, Street, City, State, and Zip are required.
- ❑ In an Orders record, Customer, Date, and DueDate are required.
- ❑ In an OrderItems record, Item and Quantity are required.
- ❑ In an OrderItems record, Quantity must be at least 1.
- ❑ In an OrderItems record, Quantity must normally (99% of the time) be no greater than 100. In very rare circumstances, it might be greater.
- ❑ Only one order can be assigned a particular DueDate. (We have special products that take an entire day to install and only enough staff to handle one installation per day.)

Decide where each of these rules should be implemented in a three-tier application.

1. Identify the rules that should be implemented in the database's structure.
2. Identify the rules that should be implemented in the middle tier.
3. Identify the rules that should be implemented in the user interface.

Note that there will be some overlap. For example, the user interface may validate a required field to avoid a round-trip to the database if the field is missing.

How It Works

1. Identify the rules that should be implemented in the database's structure.

   Whether a rule should be implemented in the database's structure depends on whether it will change. For example, if the users decide that it might be useful to create a Customers record with no associated Orders records after all, then that rule should not be implemented in the database layer. The following list shows the rules that seem extremely unlikely to change, so they can be implemented in the database's structure:

   - In Customers record, Name, Street, City, State, and Zip are required.
   - In an Orders record, Customer, Date, and DueDate are required.
   - In an OrderItems record, Item and Quantity are required.
   - In an OrderItems record, Quantity must be at least 1.
   - Only one order can be assigned a particular DueDate.

   The last one might be a bit iffy if the customer decides to add new staff so they can perform more than one installation per day. I'd check with the customer on this, but this rule seems to belong here.

2. Identify the rules that should be implemented in the middle tier.

   The rules in the middle tier are the most complicated and the most subject to change. The following list shows the rules that should probably be implemented in the middle tier:

   - All Customers records must have at least one associated Orders record.
   - All Orders records must have at least one associated OrderItems record.
   - In an OrderItems record, Quantity must normally (99% of the time) be no greater than 100. In very rare circumstances it might be greater.

3. Identify the rules that should be implemented in the user interface.

   Whether a rule should be implemented in the user interface depends mostly on the rule's simplicity and the likelihood that it will change. These rules can prevent unnecessary trips to the middle and database tiers so they can save time. However, the user interface shouldn't be unduly constrained so we have to make changes to it when business rules change. (Hint: if it's implemented in the middle tier, there's probably a reason, so you might not want to implement it here, too.) The following list shows the rules that probably should be enforced in the user interface:

   - In Customers record, Name, Street, City, State, and Zip are required.
   - In an Orders record, Customer, Date, and DueDate are required.
   - In an OrderItems record, Item and Quantity are required.
   - In an OrderItems record, Quantity must be at least 1.

   The following table summaries the places where these rules are implemented.

Rule	Database	Middle Tier	UI
All Customers records must have at least one associated Orders record.		X	
All Orders records must have at least one associated OrderItems record.		X	
In Customers record, Name, Street, City, State, and Zip are required.	X		X
In an Orders record, Customer, Date, and DueDate are required.	X		X
In an OrderItems record, Item and Quantity are required.	X		X
In an OrderItems record, Quantity must be at least 1.	X		X
In an OrderItems record, Quantity must normally (99% of the time) be no greater than 100. In very rare circumstances it might be greater.		X	
Only one order can be assigned a particular DueDate.	X		

The final rule demonstrates an unusual combination: a rule that is easy to implement in the database but hard to implement in the middle tier or user interface. Only the database has ready access to every order at all times so it can see if a new order's DueDate will conflict with a different order's DueDate.

Summary

Business rules are, quite simply, the rules that govern how a business runs. They cover everything from a list of acceptable attire for Casual Fridays to the schedule of performance bonuses.

As far as databases are concerned, business rules help define the data model. They define every field's domain (what values and ranges of values they can contain), whether fields are required, the fields' data types, and any special conditions on the fields.

Some rules are simple and unlikely to change so they can be easily implemented by using the database's features. Other rules are complex or subject to occasional change. You can make changing those rules easier by separating them from the database's structure either physically or logically.

In this chapter you learned how to:

- Understand business rules.
- Identify key business rules that may deserve special attention.
- Isolate key business rules physically or logically by extracting their data into tables, moving them into stored procedures, and moving them into a middle tier.

Separating business rules from the database's table structure is one way to make a database more efficient and flexible. Chapter 7 describes another important way to improve the database's behavior: normalization.

Before you move on to Chapter 7, 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 Exercises 1 through 3, consider Happy Sherpa Trekking, a company that sells and rents trekking equipment (boots, backpacks, llamas, yaks). They also organize guided adventure treks. Figure 6-3 shows a relational model for that part of the business.

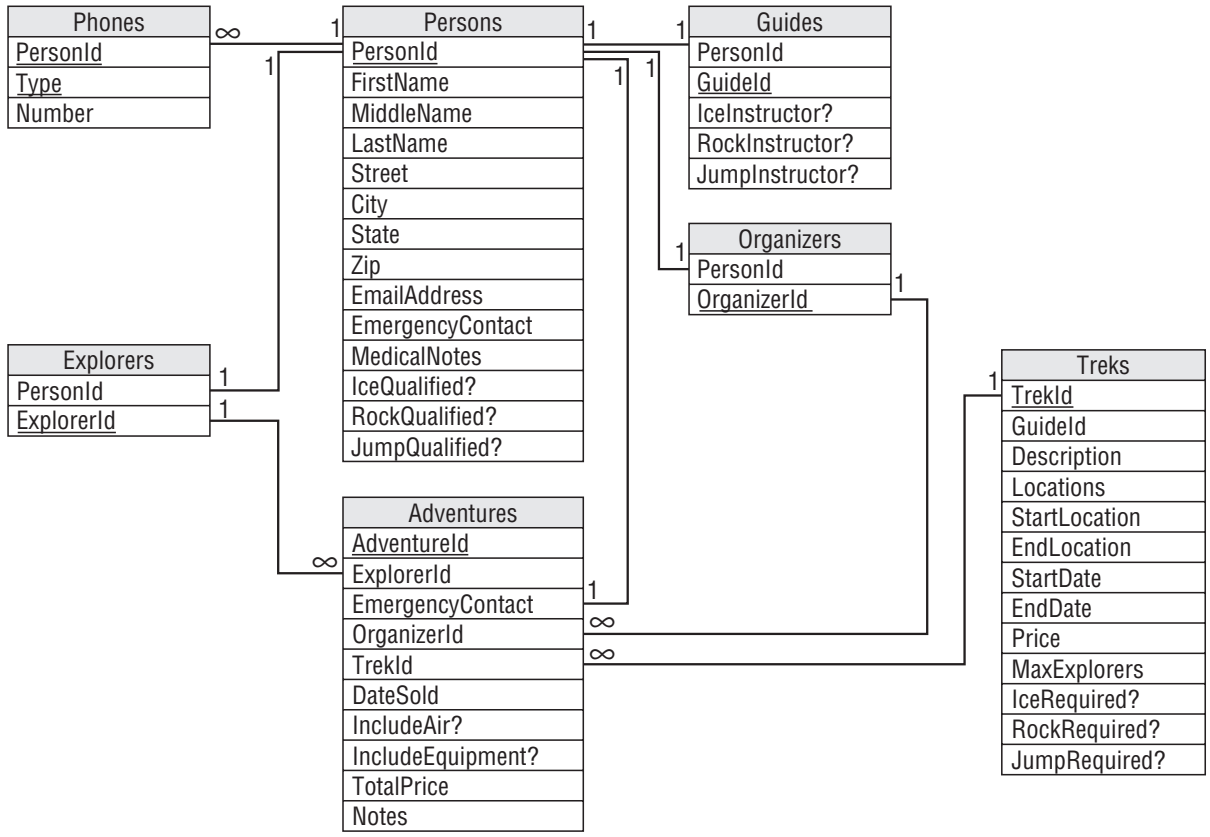

Figure 6-3

The company uses the following terminology to try to get everyone in an Indiana Jones mood:

- **Explorer:** A customer. Must be qualified to go on a trek.
- **Adventure:** A particular explorer's trek. This is basically the contract with the customer.
- **Guide:** The person who will lead the trek. Must be qualified and an instructor for any needed skills for a particular trek.

- **Organizer:** A salesperson who sells adventures to explorers.
- **Ice/Rock/Jump:** These refer to ice climbing, rock climbing, and parachute jumping skills.
- **Qualified?:** These indicate whether an explorer or guide has training in a particular skill. For example, if IceQualified? is Yes, then this person has ice climbing training.
- **Instructor?:** These indicate whether a guide is qualified to teach a particular skill. For example, if RockInstructor? is Yes, then this guide can teach rock climbing.
- **Required?:** These indicate whether a trek requires a particular skill. For example, if JumpRequired? is Yes, then this trek requires parachute jumping skills (for example, the popular ''Parachute into the Andes and Hike Out'' trek).

The company requires an emergency contact for all explorers and that contact cannot be going on the same trek (in case an avalanche takes out the whole group).

The company gives a 10% discount if the explorer purchases airline flights with the adventure. Similarly the company gives a 5% discount if the explorer rents equipment with the adventure. (During requirements gathering, one of the company's owners asks which gives the customer the biggest discount: applying a 10% discount followed by a 5% discount, applying a 5% discount followed by a 10% discount, or adding the two and applying a 15% discount. What do you think?)

If the explorer purchases airfare with the adventure, the organizer uses the Notes field to record flight information such as the explorer's starting airport and meal preferences.

1. For each of the database's tables, make a chart to describe the table's fields. Fill in the following columns for each field:
   - **Field:** The field's name.
   - **Required:** Enter Yes if the field is always required, No if it is not, or ? if it is sometimes required and sometimes not.
   - **Data Type:** The field's data type as in String or Yes/No.
   - **Domain:** List or describe the allowed values. If the allowed values are a list that might need to change, write ''List:'' before the list. You will extract these values into a new table. If the value must be in another table, list the foreign field as in Persons.PersonId.
   - **Sanity Checks:** List any basic sanity checks such as MaxExplorers > 0. Remember that these just verify extremely basic information such as a price is greater than $0.00. Don't enter more complex checks such as looking up a value in a list. Also don't enter data type validations such as the fact that an ID really is an ID or that a date is a date. (Note that this kind of sanity check has nothing to do with the explorers' sanity. If it did, we'd never get any customers to purchase the ''Tubing over Niagara Falls'' or ''August in Tampa'' packages.)
2. For each of the database's tables, list the related business rules that are unlikely to change (and that are not too horribly complicated) so they should be implemented in the table's field or table checks. (For example, before the database creates a new Adventures record, it should verify that the corresponding trek has space available.)

   Include any fields that can be validated against a list of values that will never change (such as Gender) and more complex format validations (such as phone numbers).

Do not include fields that are foreign key constraints because that validates them completely. For example, a Phones record's PersonId value must be in the Persons table so it needs no further validation.

3. List any business rules that are likely to change, that fit better in a lookup table, that are really complicated, or that are just too weird to build irrevocably into the database. Next to each, describe how you might extract that business rule from the database's structure.
4. List the new tables (and their fields) that you would create to implement these changes.