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.

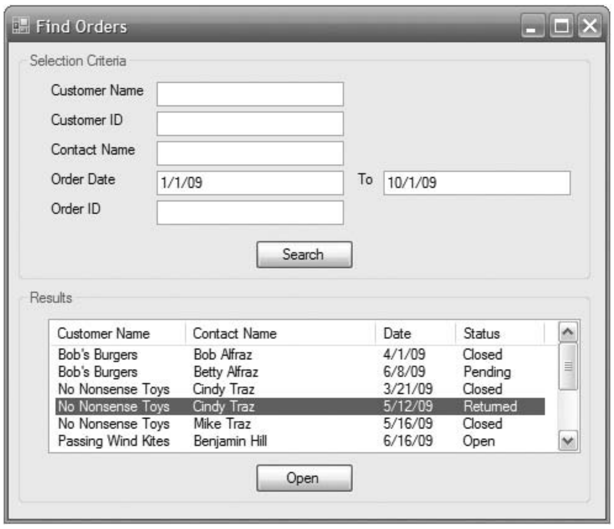
To build the user interface model, start by making rough sketches of the screens that the user will see. Often these first sketches can come directly from paper forms if any exist.

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.



Customer Name			
Customer ID			
Contact Name			
Order Date	1/1/09	To	10/1/09
Order ID			

Customer Name	Contact Name	Date	Status
Bob's Burgers	Bob Alfraz	4/1/09	Closed
Bob's Burgers	Betty Alfraz	6/8/09	Pending
No Nonsense Toys	Cindy Traz	3/21/09	Closed
No Nonsense Toys	Cindy Traz	5/12/09	Returned
No Nonsense Toys	Mike Traz	5/16/09	Closed
Passing Wind Kites	Benjamin Hill	6/16/09	Open

Figure 5-1

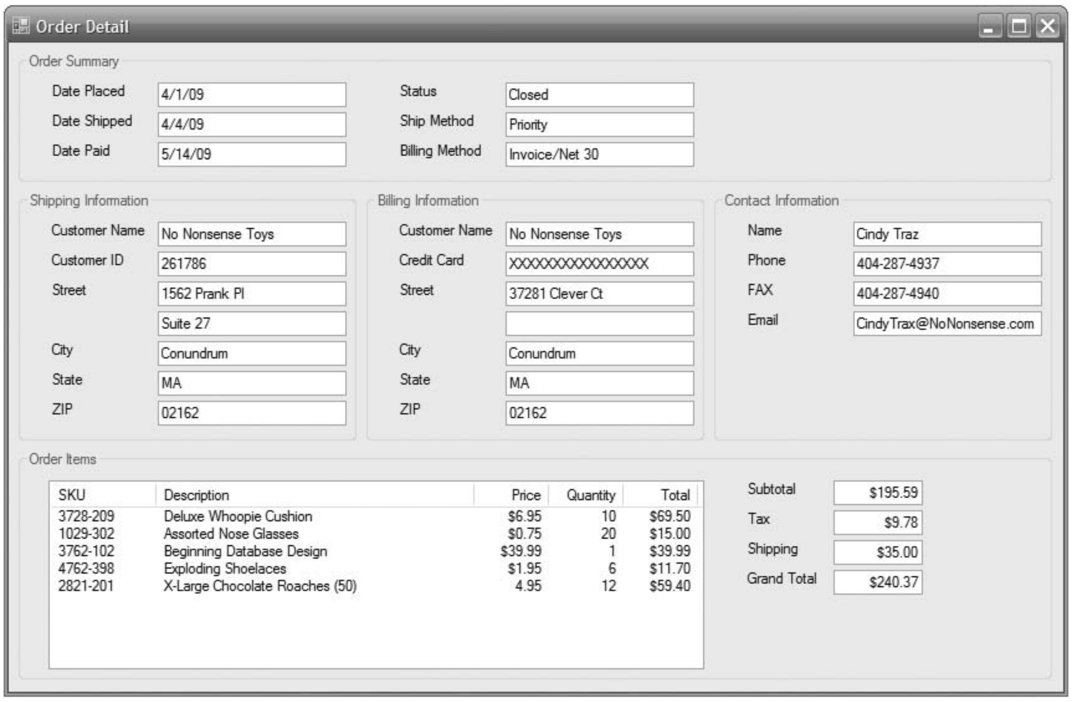
The form and its description also raise some important questions:

- ❑ 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



Order Detail

Order Summary

Date Placed	4/1/09	Status	Closed
Date Shipped	4/4/09	Ship Method	Priority
Date Paid	5/14/09	Billing Method	Invoice/Net 30

Shipping Information

Customer Name	No Nonsense Toys
Customer ID	261786
Street	1562 Prank Pl
	Suite 27
City	Conundrum
State	MA
ZIP	02162

Billing Information

Customer Name	No Nonsense Toys
Credit Card	XXXXXXXXXXXXXXXX
Street	37281 Clever Ct
City	Conundrum
State	MA
ZIP	02162

Contact Information

Name	Cindy Traz
Phone	404-287-4937
FAX	404-287-4940
Email	CindyTrax@NoNonsense.com

Order Items

SKU	Description	Price	Quantity	Total
3728-209	Deluxe Whoopie Cushion	$6.95	10	$69.50
1029-302	Assorted Nose Glasses	$0.75	20	$15.00
3762-102	Beginning Database Design	$39.99	1	$39.99
4762-398	Exploding Shoelaces	$1.95	6	$11.70
2821-201	X-Large Chocolate Roaches (50)	4.95	12	$59.40

Subtotal	$195.59
Tax	$9.78
Shipping	$35.00
Grand Total	$240.37

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.

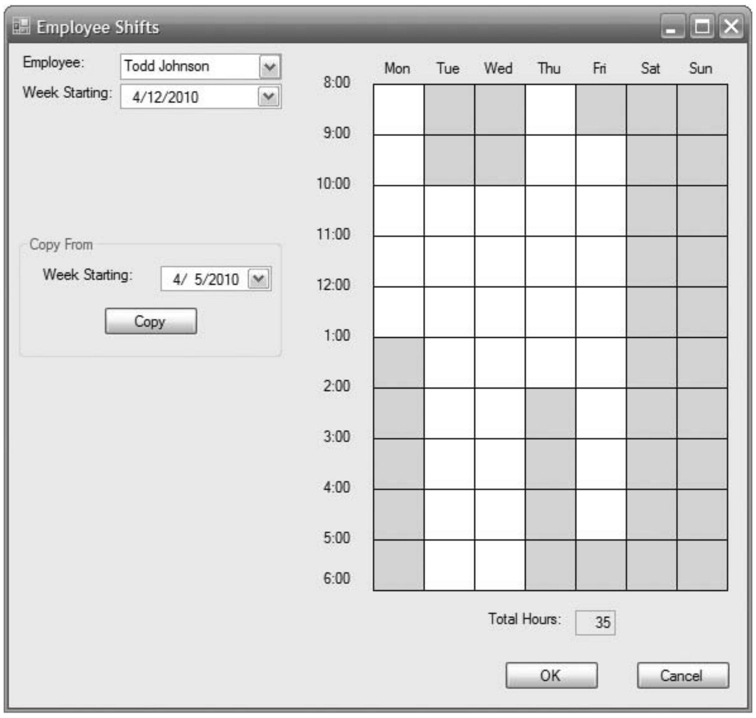

Employee: Todd Johnson
Week Starting: 4/12/2010
Copy From
Week Starting: 4/ 5/2010
Total Hours: 35

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.

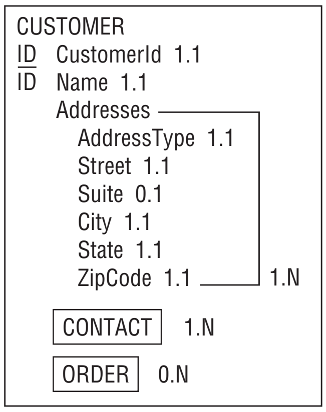


Figure 5-4

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.

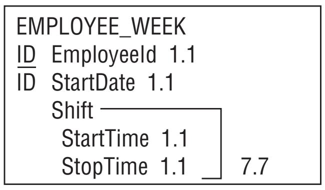


Figure 5-5

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.

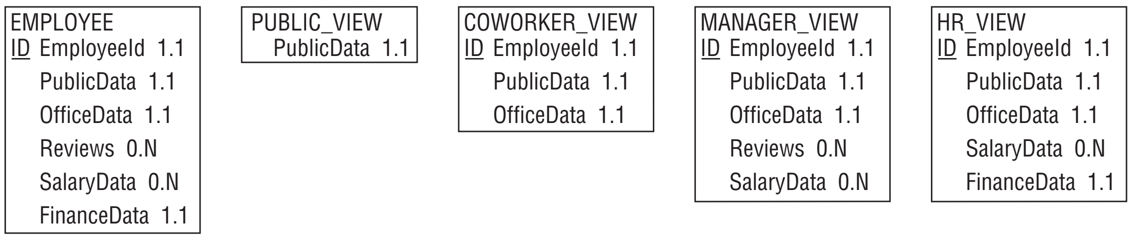


Figure 5-6

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.

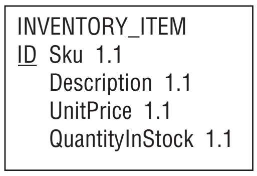


Figure 5-7

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.

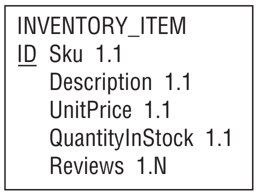


Figure 5-8

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.

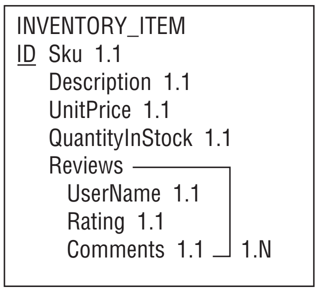


Figure 5-9

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.)

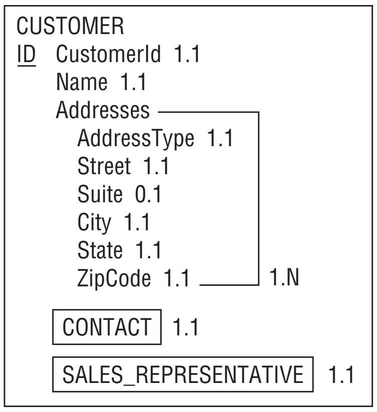


Figure 5-10

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.

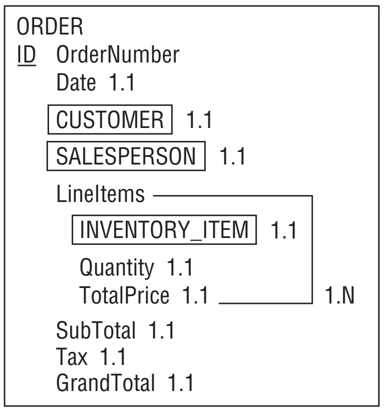


Figure 5-11

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 DEVELOPERs and a DEVELOPER may work on many PROJECTs, so the two classes have a many-to-many relationship. Figure 5-12 shows this relationship modeled with straightforward object attributes.

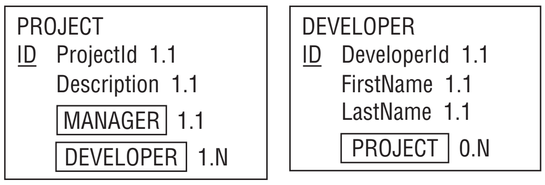


Figure 5-12

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).

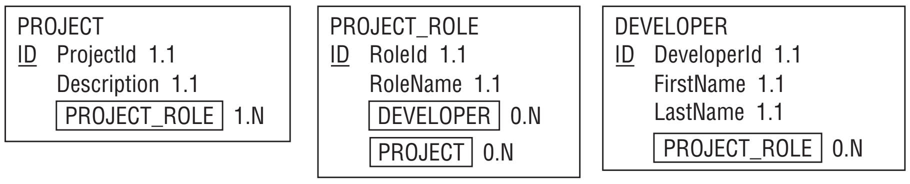


Figure 5-13

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.

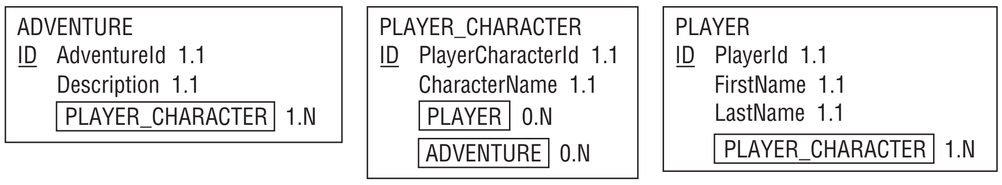


Figure 5-14

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*.

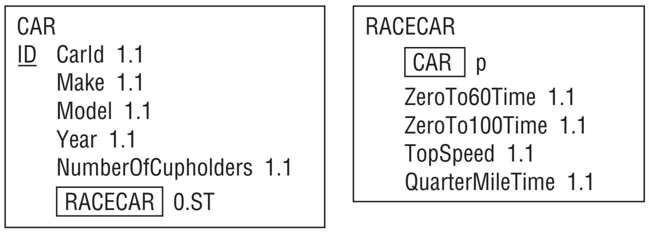


Figure 5-15

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.)

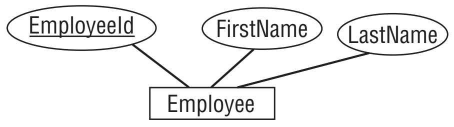


Figure 5-16

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.

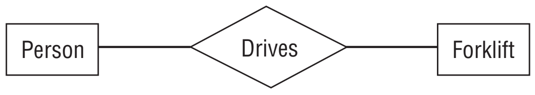


Figure 5-17

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`.

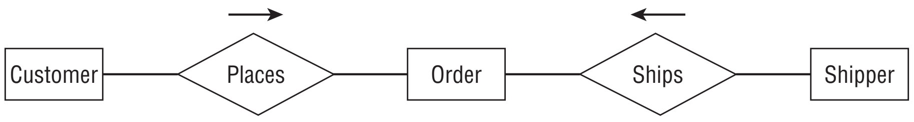


Figure 5-18

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.

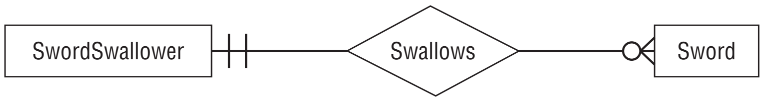


Figure 5-19

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.

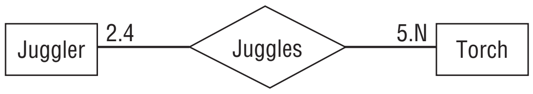


Figure 5-20

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`.

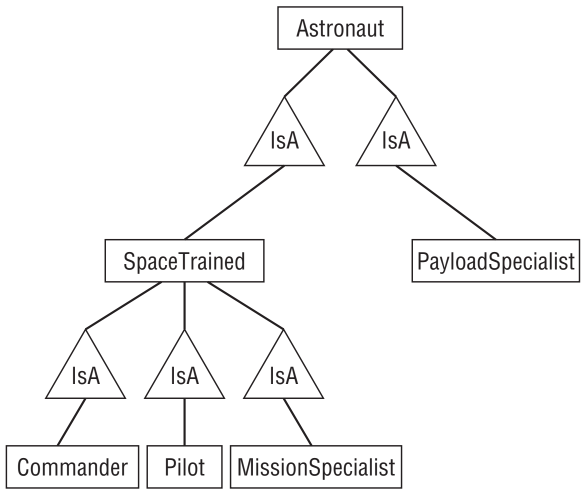


Figure 5-21

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.

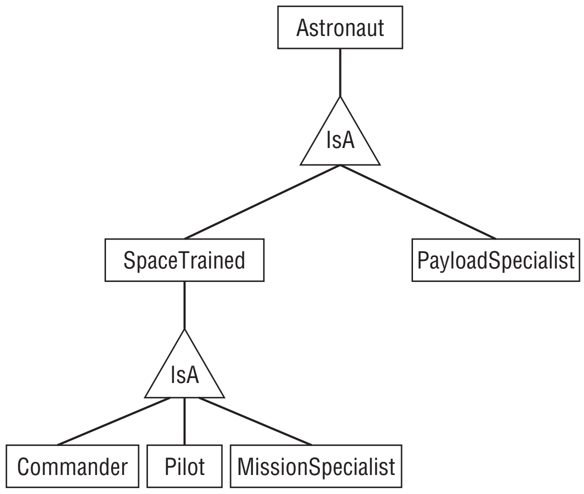


Figure 5-22

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.

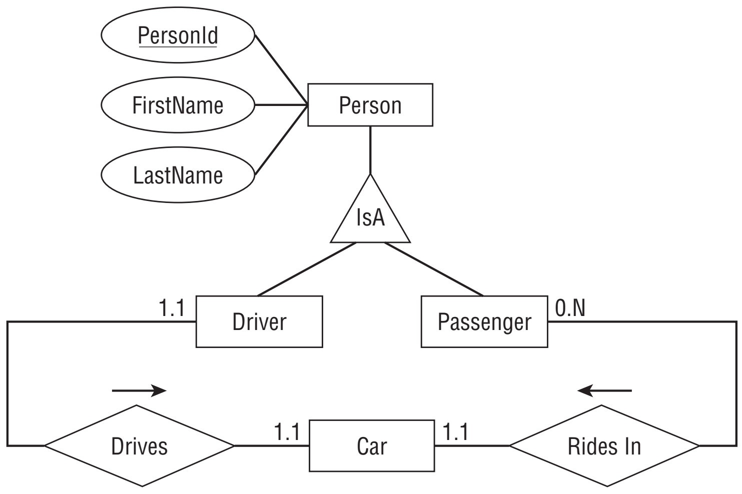


Figure 5-23

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.

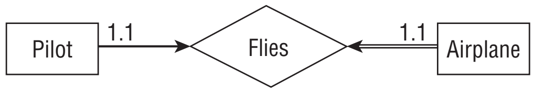


Figure 5-24

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.

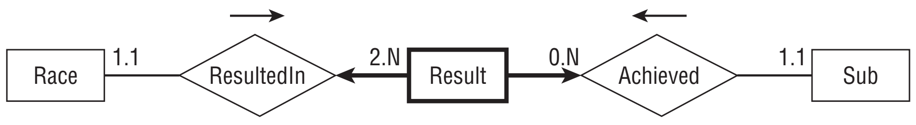


Figure 5-25

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 `Items` ordered.

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

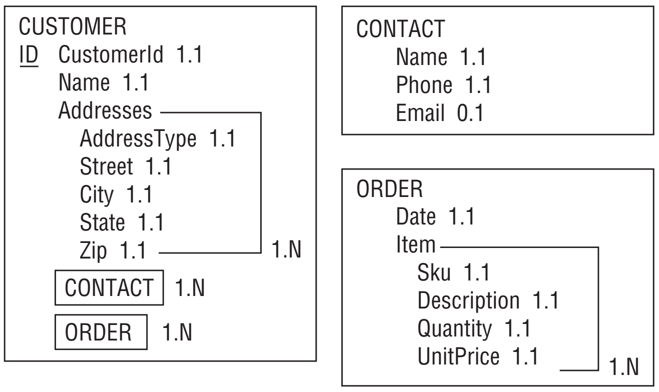


Figure 5-26

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.

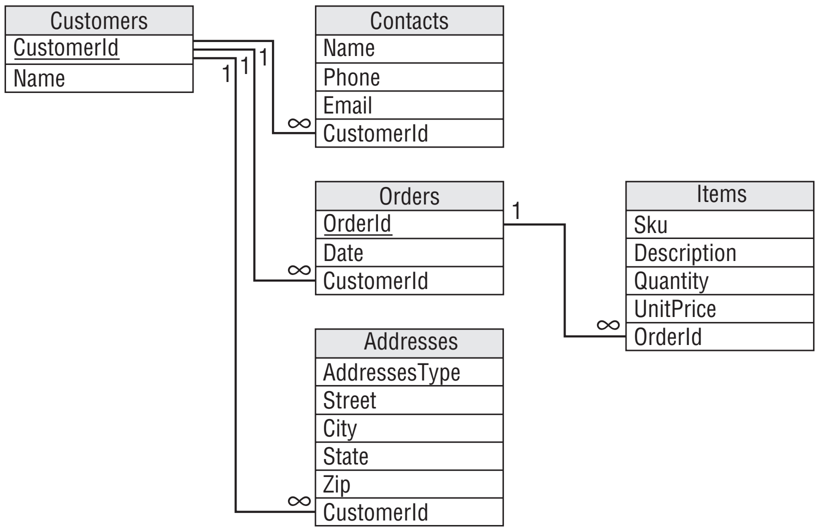


Figure 5-27

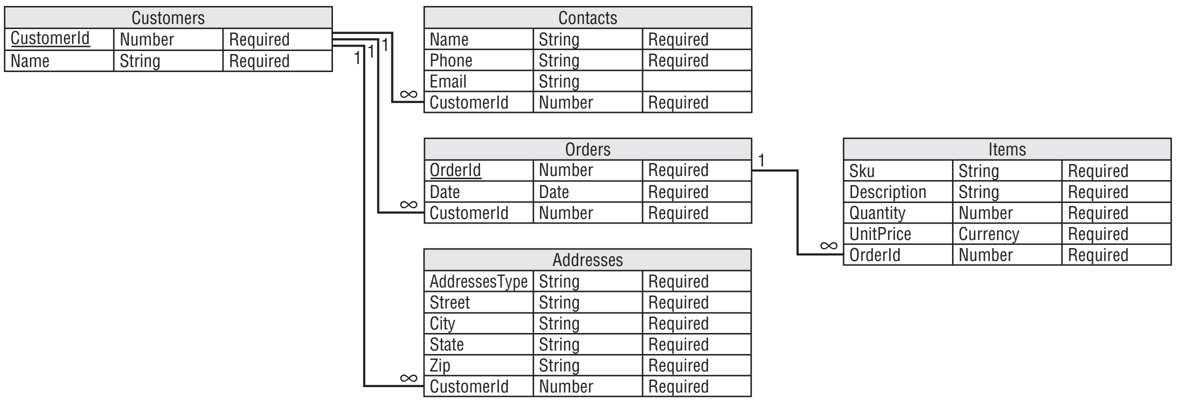


Figure 5-28

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.

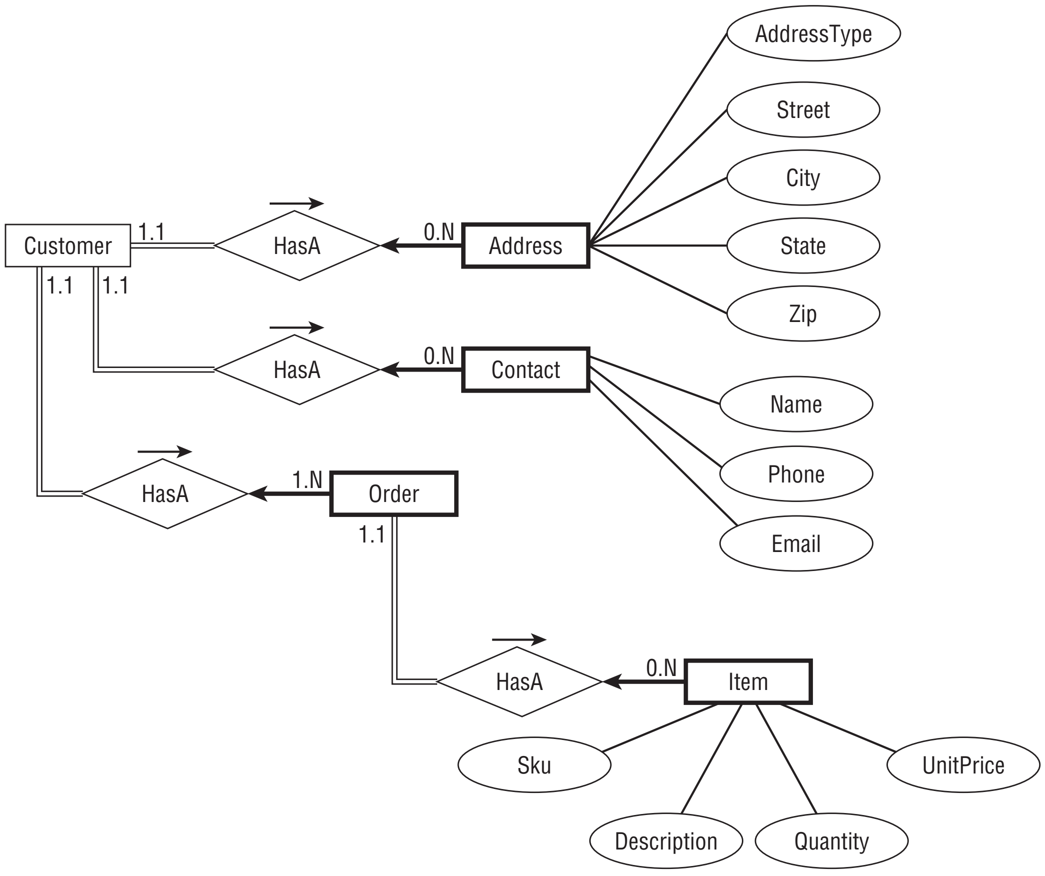


Figure 5-29

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.