# 2

# Entities and Data Relationships

In this chapter we will explore the fundamental concept behind all databases: There are things in a business environment about which we need to store data, and those things are related to one another in a variety of ways. In fact, to be considered a *database*, the place where data are stored must contain not only the data but also information about the relationships between those data.

The idea behind a database is that the user—either a person working interactively or an application program—has no need to worry about the way in which data are physically stored on disk. The user phrases data manipulation requests in terms of data relationships. A piece of software known as a *database management system* (DBMS) then translates between the user's request for data and the physical data storage.

The formal way in which you express data relationships to a DBMS is known as a *data model.* The relational data model, about which you will learn in this book, is just such a formal structure. However, the underlying relationships in a database environment are independent of the data model and therefore also independent of the DBMS you are using. Before you can design a database for any data model, you need to be able to identify data relationships.

> *Note: Most DBMSs support only one data model. Therefore, when you choose a DBMS, you are also choosing your data model.*

In this chapter we will explore data relationships and their characteristics. You will also learn a DBMS-independent technique for documenting those relationships known as the *entity–relationship diagram* (ER diagram).

## Entities and Their Attributes

An *entity* is something about which we store data. A customer is an entity, as is a merchandise item stocked by Lasers Only. Entities are not necessarily tangible. For example, an event such as a concert is an entity; an appointment to see the doctor is an entity.

Entities have data that describe them (their *attributes*). For example, a customer entity is usually described by a customer number, first name, last name, street, city, state, zipcode, and phone number. A concert entity might be described by a title, date, location, and name of the performer.

When we represent entities in a database, we actually store only the attributes. Each group of attributes that describes a single real-world occurrence of an entity acts to represent an *instance* of an entity. For example, in Figure 2-1, you can see four instances of a customer entity stored in a database. If we have 1000 customers in our database, then there will be 1000 collections of customer attributes.

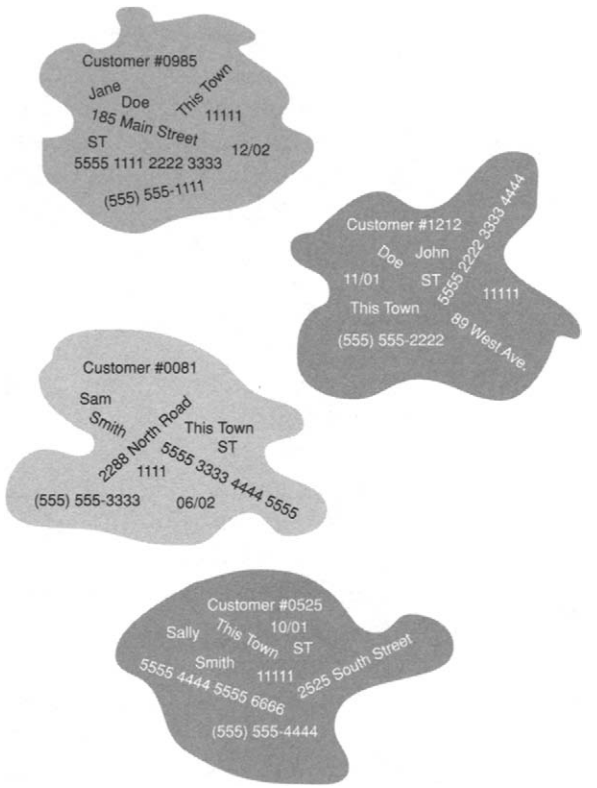



**Figure 2-1: Instances of a customer entity in a database**

> *Note: Keep in mind that we are not making any statements about how the instances are physically stored. What you see in Figure 2-1 is purely a conceptual representation.*

## Entity Identifiers

The only purpose for putting the data that describe an entity into a database is to retrieve the data at some later date. This means that we must have some way of distinguishing one entity from another so that we can always be certain that we are retrieving the precise entity we want. We do this by ensuring that each entity has some

attribute values that distinguish it from every other entity in the database (an *entity identifier*).

Assume, for example, that Lasers Only has two customers named John Smith. If an employee searches for the items John Smith has on order, which John Smith will the DBMS retrieve? In this case, both of them. Because there is no way to distinguish between the two customers, the result of the query will be inaccurate.

Lasers Only solved the problem by creating unique customer numbers. That is indeed a common solution to identifying instances of entities where there is no simple unique identifier suggested by the data themselves.

Another solution would be to pair the customer's first and last names with his or her telephone number. This combination of columns (a *concatenated identifier*) would also uniquely identify each customer. There are, however, two drawbacks to doing so. First, the identifier is long and clumsy; it would be easy to make mistakes when entering any of the parts. Second, if the customer's phone number changes, then the identifier must also change. As you read in Chapter 1, changes in an entity identifier can cause serious problems in a database.

Some entities, such as invoices, come with natural identifiers (the invoice number). We assign unique, meaningless numbers to others — especially accounts, people, places, and things. Still others require concatenated identifiers.

> *Note: We will examine the issue of what makes a good unique identifier more closely in Chapter 4, when we talk about "primary keys."*

When we store an instance of an entity in a database, we want the DBMS to ensure that the new instance has a unique identifier. This is an example of a *constraint* on a database, a rule to which data must adhere. The enforcement of a variety of database constraints helps us to maintain data consistency and accuracy.

## Single-Valued versus Multivalued Attributes

Because we are eventually going to create a relational database, the attributes in our data model must be *single-valued*. This means that for a given instance of an entity, each attribute can have only one value. For example, a customer entity allows only one telephone number for each customer. If a customer has more than one phone number and wants them all included in the database, then the customer entity cannot handle them.

> *Note: While it is true that the entity–relationship model of a database is independent of the formal data model used to express the structure of the data to a DBMS, we often make decisions on how to model the data based on the requirements of the formal data model we will be using. Removing multivalued attributes is one such case. You will also see an example of this when we deal with many-to-many relationships between entities.*

The existence of more than one phone number turns the phone number attribute into a *multivalued attribute*. Because an entity in a relational database cannot have multivalued attributes, you must handle those attributes by creating an entity to hold them.

In the case of the multiple phone numbers, we could create a phone number entity. Each instance of the entity would include the customer number of the person to whom the phone number belonged along with the telephone number. If a customer had three phone numbers, then there would be three instances of the phone number entity for the customer. The entity's identifier would be the concatenation of the customer number and the telephone number.

> *Note: There is no way to avoid using the telephone number as part of the entity identifier in the telephone number entity. As you will come to understand as you read this book, in this particular case there is no harm in using it in this way.*

What is the problem with a multivalued attribute? Multivalued attributes can cause problems with the meaning of data in the

database, significantly slow down searching, and place unnecessary restrictions on the amount of data that can be stored.

Assume, for example, that you have an Employee entity with attributes for the names and birthdates of dependents. Each attribute is allowed to store multiple values. How will you associate the correct birthdate with the name of the dependent to which it applies? Will it be by the position of a value store in the attribute (i.e., the first name is related to the first birthdate, and so on)? If so, how will you ensure that there is a birthdate for each name and a name for each birthdate? How will you ensure that the order of the values is never mixed up?

When searching a multivalued attribute, a DBMS must search each value in the attribute, most likely scanning the contents of the attribute sequentially. A sequential search is the slowest type of search available.

In addition, how many values should a multivalued column be able to store? If you specify a maximum number, what will happen when you need to store more than the maximum number of values? For example, what if you allow room for 10 dependents in the Employee entity just discussed and you encounter an employee with 11 dependents? Do you create another instance of the Employee entity to handle that person? Consider all the problems that doing so would create, particularly in terms of the unnecessary duplicated data.

> *Note: Although it is theoretically possible to write a DBMS that will store an unlimited number of values in an attribute, the implementation would be difficult and searching much slower than if the maximum number of values were specified in the database design.*

As a general rule, if you run across a multivalued attribute, this is a major hint that you need another attribute. The only way to handle multiple values of the same attribute is to create an entity of which you can store multiple instances, once for each value of the attribute. In the case of the Employee entity, we would need a Dependent

entity that could be related to the Employee entity. There would be one occurrence of the Dependent entity related to an occurrence of the Employee entity for each of an employee's dependents. In this way, there is no limit to the number of an employee's dependents. In addition, each occurrence of the Dependent entity would contain the name and birthdate of only one dependent, eliminating any confusion about which name was associated with which birthdate. Searching would also be faster because the DBMS could use fast search techniques on the individual Dependent entity occurrences, without resorting to the slow sequential search.

## Avoiding Collections of Entities

When you first begin to work with entities, the nature of an entity can be somewhat confusing. Consider, for example, the merchandise inventory handled by Lasers Only. Is "inventory" an entity? No. Inventory is a collection of the merchandise items handled by the store. The entity is actually the merchandise item. Viewing all of the instances of the merchandise item entity as a whole provides the inventory.

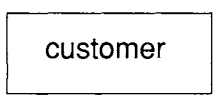
To make this a bit clearer, consider the attributes you would need if you decided to include an inventory entity: merchandise item number, item title, number in stock, retail price, and so on. But because you are trying to describe an entire inventory with a single entity, you need multiple values for each of those attributes. As you read earlier, however, attributes cannot be multivalued. This tells you that inventory cannot stand as an entity. It must be represented as a collection of instances of a merchandise item entity.

As another example, consider a person's medical history maintained by a doctor. Like an inventory, a medical history is a collection of more than one entity. A medical history is made up of appointments and the events that occur during those appointments. Therefore, the history is really a collection of instances of appointment entities and medical treatment entities. The "history" is an output that a database application can obtain by gathering the data stored in the underlying entity instances.

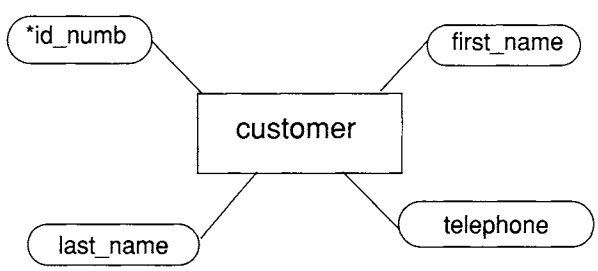
## Documenting Logical Data Relationships

Entity-relationship diagrams provide a way to document the entities in a database along with the attributes that describe them. There are actually several styles of ER diagrams. The two most commonly used styles are Chen (named after the originator of ER modeling, Dr. Peter P. S. Chen) and Information Engineering (IE), which grew out of work by James Martin and Clive Finkelstein. It does not matter which you use, as long as everyone who is using the diagram understands its symbols.

Both the Chen and Information Engineering models use rectangles to represent entities. Each entity's name appears in the rectangle and is expressed in the singular, as in

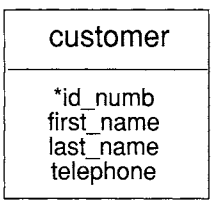



The original Chen model has no provision for showing attributes on the ER diagram itself. However, many people have extended the model to include the attributes in ovals:




The entity's identifier is the attribute preceded by an asterisk (*id_numb).

The Information Engineering model includes the attributes in the rectangle with the entity:




Because the Information Engineering model tends to produce a less cluttered diagram, we will be using it for most of the diagrams in this book, although you will be introduced to elements of both models throughout this chapter.

## Entities and Attributes for Lasers Only

The major entities and their attributes for the Lasers Only database can be found in Figure 2-2. As you will see, the design will require additional entities as we work with the relationships between those already identified. In particular, there is no information in Figure 2-2 that indicates which items appear on which orders because that information is a part of the logical relationship between orders and items.

The entities in Figure 2-2 and the remainder of the ER diagrams in this book were created with a special type of software known as a *CASE tool* (computer-aided software engineering). CASE tools provide a wide range of data and systems modeling assistance. You will find more detail on how CASE tools support the database design process in Chapter 9.

> *Note: The specific product used for these diagrams was MacA&D, which provides capabilities typical of most professional CASE tools.*

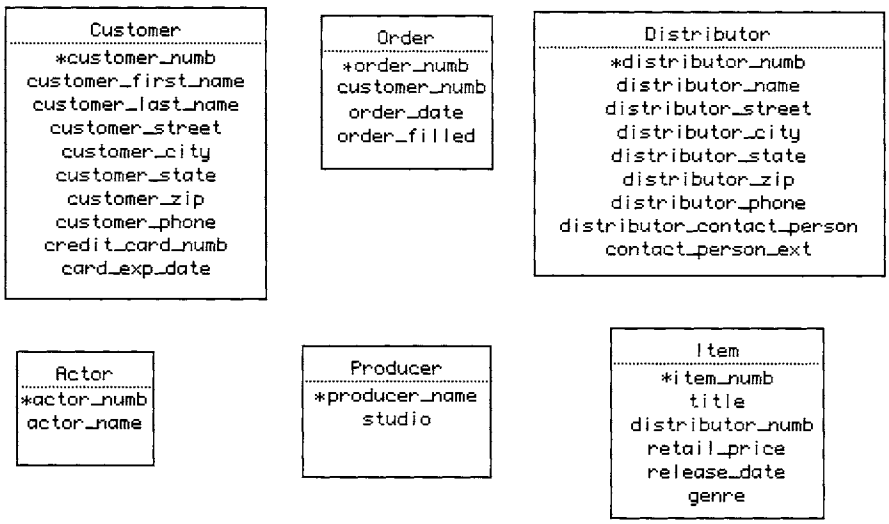



**Figure 2-2: Major entities and their attributes for the Lasers Only database**

# Domains

Each attribute has a *domain*, an expression of the permissible values for that attribute. A domain can be very small. For example, a T-shirt store might have a Size attribute for its merchandise items with the values L, XL, and XXL comprising the entire domain. In contrast, an attribute for a customer's first name is very large and might be specified only as "text" or "human names."

A DBMS enforces a domain through a *domain constraint*. Whenever a value is stored in the database, the DBMS verifies that it comes from its attribute's specified domain. Although in many cases we cannot specify small domains, at the very least the domain assures us that we are getting data of the right type. For example, a DBMS can prevent a user from storing 123x50 in an attribute whose domain is currency values. Most DBMSs also provide fairly tight domain checking on date and time attributes, which can help you avoid illegal dates such as February 30.

## Documenting Domains

The common formats used for ER diagrams do not usually include domains on the diagrams themselves, but store the domains in an associated document (usually a *data dictionary*, something about which you will read much more throughout this book). However, the version of the Chen method that includes attributes can also include domains by placing an expression of the domain underneath each attribute. Notice in Figure 2-3 that three of the domains are fairly general (integer and character), while the domain for the telephone number attribute includes a very specific format. Whether a domain can be constrained in this way depends on the DBMS.

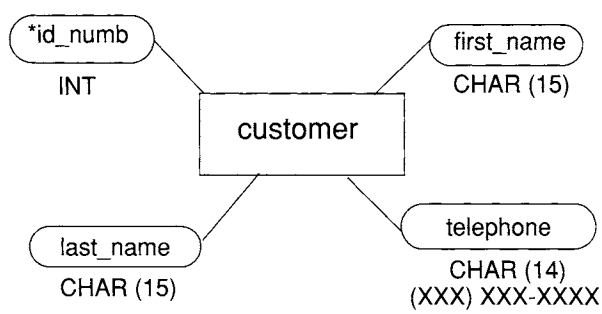



**Figure 2-3: Indicating domains on an ER diagram**

*Note: There is no specific syntax for indicating domains. However, if you know which DBMS you will be using, consider using the column data types supported by that product as domains in an ERD to simplify the later conversion to the DBMS's requirements.*

## Practical Domain Choices

The domains that Lasers Only chooses for its attributes should theoretically be independent of the DBMS that the company will use. In practical terms, however, it makes little sense to assign domains that you cannot implement. Therefore, the database designer

working for Lasers Only takes a look at the DBMS to see what column data types are supported.

Most relational DBMSs that use SQL as their query language provide the following among their column data types, any of which can be assigned as a domain to an attribute:

- CHAR: A fixed-length string of text, usually up to 256 characters.
- VARCHAR: A variable-length string of text, usually up to 256 characters.
- INT: An integer, the size of which varies depending on the operation system.
- DECIMAL and NUMERIC: Real numbers, with fractional portions to the right of the decimal point. When you assign a real number domain, you must specify how many digits the number can contain (including the decimal point) and how many digits should be to the right of the decimal point (the value's *precision*). For example, currency values usually have precision of two, so a number in the format XXX.XX might have a domain of DECIMAL (6,2).
- DATE: A date.
- TIME: A time.
- DATETIME: The combination of a date and a time.
- BOOLEAN: A logical value (either true or false).

Many of today's DBMSs also support a data type known as a BLOB (binary large object), which can store anything binary, such as a graphic.

Choosing the right domain can make a big difference in the accuracy of a database. For example, a U.S. zip code is made up of five or nine digits. Should an attribute for a zip code therefore be given a domain of INT? No, for two reasons. First, it would be nice to be able to include the hyphen in nine-digit zip codes. Second, and more important, zip codes in the northeast begin with a zero. If they are stored as a number, the leading zero disappears. Therefore, we always choose a CHAR domain for zip codes. Since we never do

arithmetic with zip codes, nothing is lost by using character rather than numeric storage.

By the same token, it is important to choose domains of DATE and TIME for chronological data. As an example, consider what would happen if the dates 01/12/2000 and 08/12/1999 were stored as characters. If you ask the DBMS to choose which date comes first, the DBMS will compare the character strings in alphabetical order, and respond that 01/12/2000 comes first, because 01 alphabetically precedes 08. The only way to get character dates to order correctly is to use the format YYYY/MM/DD, a format that is rarely used anywhere in the world. However, if the dates were given a domain of DATE, then the DBMS would order them properly. The DBMS would also be able to perform date arithmetic, finding the interval between two dates or adding constants (for example, 30 days) to dates.

# Basic Data Relationships

Once you have a good idea of the basic entities in your database environment, your next task is to identify the relationships among those entities. There are three basic types of relationships that you may encounter: one-to-one, one-to-many, and many-to-many.

Before turning to the types of relationships themselves, there is one important thing to keep in mind: The relationships that are stored in a database are between instances of entities. For example, a Lasers Only customer is related to the items he or she orders. Each instance of the customer entity is related to instances of the specific items ordered (see Figure 2-4).

> *Note: As you look at Figure 2-4, once again remember that this is a purely conceptual representation of what is in the database and is completely unrelated to the physical storage of the data.*

When we document data relationships, such as when we draw an ER diagram, we show the types of relationships among entities. We

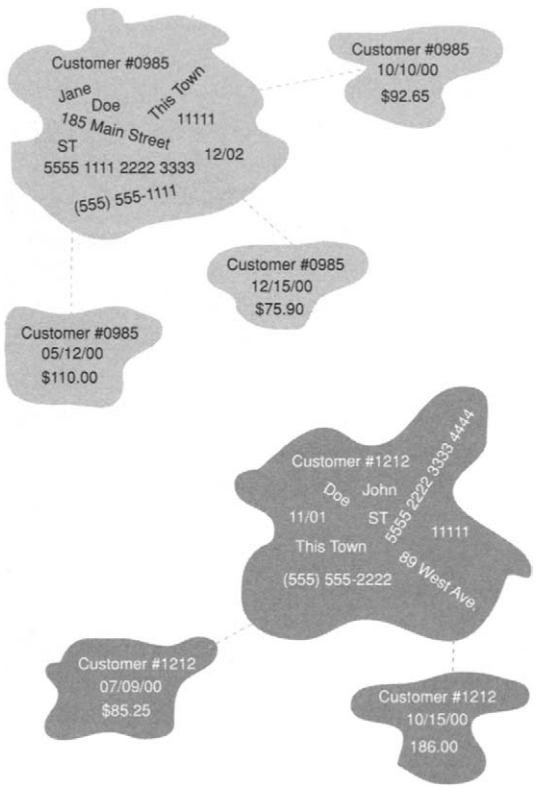



**Figure 2-4: Relationships between instances of entities in a database**

are showing the possible relationships that are allowable in the database. Unless we specify that a relationship is mandatory, there is no requirement that every instance of every entity be involved in every documented relationship. For example, Lasers Only could store data about a customer without the customer having any current orders to which it is related.

## One-to-One Relationships

Consider, for a moment, an airport in a small town and the town in which the airport is located, both of which are described in a database of small town airports. Each of these might be represented as

an instance of a different type of entity. The relationships between the two instances can then be expressed as "The airport is located in one and only one town and the town contains one and only one airport."

This is a true *one-to-one relationship* because at no time can a single airport be related to more than one town and no town can be related to more than one airport. (Although there are municipalities that have more than one airport, the towns in this database are too small for that to ever happen.)

If we have two instances of two entities (A and B) called $A_i$ and $B_i$, then a one-to-one relationship exists if at all times $A_i$ is related to no instances of entity B or one instance of entity B, and $B_i$ is related to no instances of entity A or one instance of entity A.

True one-to-one relationships are very rare in business. For example, assume that Lasers Only decides to start dealing with a new distributor of laser discs. At first, the company orders only one specialty title from the new distributor. If we peered inside the database, we would see that the instance of the distributor entity was related to just the one merchandise item instance. This would then appear to be a one-to-one relationship. However, over time Lasers Only may choose to order more titles from the new distributor, which would violate the rule that the distributor must be related to no more than one merchandise item. What we have is therefore not a true one-to-one relationship. (This is an example of a one-to-many relationship, which is discussed in the next section of this chapter.)

By the same token, what if Lasers Only created a special credit card entity to hold data about the credit cards that renters used to secure their rentals? Each customer has only one credit card on file with the store. There would therefore seem to be a one-to-one relationship between the instance of a customer entity and the instance of the credit card entity. However, in this case we are really dealing with a single entity. The credit card number, the type of credit card, and the credit card's expiration date can all become attributes of the customer entity. Given that only one credit card is stored for each

customer, the attributes are not multivalued; no separate entity is needed.

If you think you are dealing with a one-to-one relationship, look at it very carefully. Be sure that you are not really looking at a one-to-many relationship or that what you think is two entities should really be one.

## One-to-Many Relationships

The most common type of relationship is a *one-to-many relationship*. (In fact, most relational databases are constructed from the rare one-to-one relationship and numerous one-to-many relationships.) For example, Lasers Only typically orders many titles from each distributor and a given title comes from only one distributor. By the same token, a customer places many orders but an order comes from only one customer.

If we have instances of two entities (A and B), then a one-to-many relationship exists between two instances ($A_i$ and $B_i$) if $A_i$ is related to zero, one, or more instances of entity B and $B_i$ is related to zero or one instance of entity A.

Other one-to-many relationships include that between a daughter and her biological mother. A woman may have zero, one, or more biological daughters; a daughter has only one biological mother. As another example, consider a computer and its CPU. A CPU may not be installed in any computer or it may be installed in at most one computer; a computer may have no CPU, one CPU, or more than one CPU.

The example about which you read earlier concerning Lasers Only and the distributor from which the company ordered only one title is actually a one-to-many relationship where the "many" is currently "one." Remember that when we are specifying data relationships, we are indicating possible relationships and not necessarily requiring that all instances of all entities participate in every documented relationship. There is absolutely no requirement that a distributor be

related to any merchandise item, much less one or more merchandise items. (It might not make much sense to have a distributor in the database from whom the company did not order, but there is nothing to prevent data about that distributor from being stored.)

## Many-to-Many Relationships

*Many-to-many relationships* are also very common. There is, for example, a many-to-many relationship between an order placed by a Lasers Only customer and the merchandise items carried by the store. An order can contain multiple items; each item can appear on more than one order. The same is true of the orders placed with distributors. An order can contain multiple items and each item can appear on more than one order.

A many-to-many relationship exists between entities A and B if for two instances of those entities ($A_i$ and $B_i$), $A_i$ can be related to zero, one, or more instances of entity B and $B_i$ can be related to zero, one, or more instances of entity A.

Many-to-many relationships bring two major problems to a database's design. These issues and the way in which we solve them are discussed in the next major section of this chapter ("Dealing with Many-to-Many Relationships").

## Weak Entities and Mandatory Relationships

As we have been discussing types of data relationships, we have defined those relationships by starting each with "zero," indicating that the participation by a given instance of an entity in a relationship is optional. For example, Lasers Only can store data about a customer in its database before the customer places an order. Therefore, an instance of the customer entity does not have to be related to any instances of the order entity.

However, the reverse is not true in this database: An order *must* be related to a customer. Without a customer, an order cannot exist. An

order is therefore an example of a *weak entity*, one that cannot exist in the database unless a related instance of another entity is present and related to it. An instance of the customer entity can be related to zero, one, or more orders. However, an instance of the order entity must be related to one and only one customer. The "zero" option is not available to a weak entity. The relationship between an instance of the order entity and the customer is therefore a mandatory relationship.

Identifying weak entities and their associated mandatory relationships can be very important for maintaining the consistency and integrity of the database. Consider the effect, for example, of storing an order without knowing the customer to which it belongs. There would be no way to ship the item to the customer, causing a company to lose business.

By the same token, we typically define the relationship between an order and the order lines (the specific items on the order) as one-to-many because we don't want to allow an order line to exist in the database without it being related to an order. (An order line is meaningless without knowing the order to which it belongs.)

In contrast, we can allow a a merchandise item to exist in a database without indicating the supplier from which it comes (assuming that there is only one source per item). This lets us store data about a new item before we have decided on a supplier. In this case, the relationship between a supplier and an item is actually zero-to-many.

## Documenting Relationships

The Chen and Information Engineering methods of drawing ER diagrams have very different ways of representing relationships, each of which has its advantages in terms of the amount of information it provides and its complexity.

### The Chen Method

The Chen method uses diamonds for relationships and lines with arrows to show the type of relationship between entities. For example, in Figure 2-5 you can see the relationship between a Lasers Only customer and an order. The single arrow pointing toward the customer entity indicates that an order belongs to at most one customer. The double arrow pointing toward the order entity indicates that a customer can place one or more orders. The word within the relationship diamond gives some indication of the meaning of the relationship.

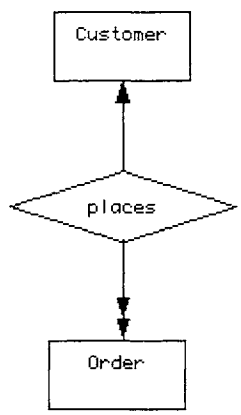



**Figure 2-5: Using the Chen method with relationship diamonds and arrows**

There are two alternative styles within the Chen method. The first (for example, Figure 2-6) replaces the arrows with numbers and letters. A "1" indicates that an order comes from one customer. The "M" (or an "N") indicates that a customer can place many orders.

The second alternative addresses the problem of trying to read the relationship in both directions when the name of the relationship is within the diamond. "Customer places order" makes sense, but "order places customer" does not. To solve the problem, this alternative removes the relationship name from the diamond and adds both the relationship and its inverse to the diagram, as in Figure 2-7. This version of the diagram can be read easily in either direction:

"A customer places many orders" and "An order is placed by one customer."

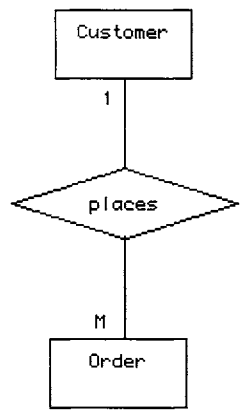



**Figure 2-6: A Chen method ER diagram using letters and numbers rather than arrows to show relationships**

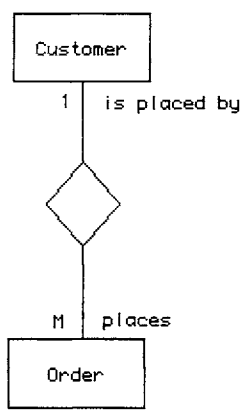
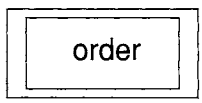



**Figure 2-7: Adding inverse relationships to a Chen method ER diagram**

There is one major limitation to the Chen method of drawing ER diagrams: There is no obvious way to indicate weak entities and mandatory relationships. For example, an order should not exist in the database without a customer. Therefore, order is a weak entity and its relationship with a customer is mandatory.

Some database designers have therefore added a new symbol to the Chen method for a weak entity — a double-bordered rectangle:




Whenever a weak entity is introduced into an ER diagram, it indicates that the relationship between that entity and at least one of its parents is mandatory. However, if the entity does happen to have multiple parents, then there is no way to determine simply by looking at the diagram which of the relationships are mandatory.

**The Information Engineering Method**

The Information Engineering (IE) method exchanges simplicity in line ends for added information. As a first example, consider Figure 2-8. This is the same one-to-many relationship we have been using to demonstrate the Chen method ER diagrams. However, in this case the ends of the lines indicate which relationships are mandatory.

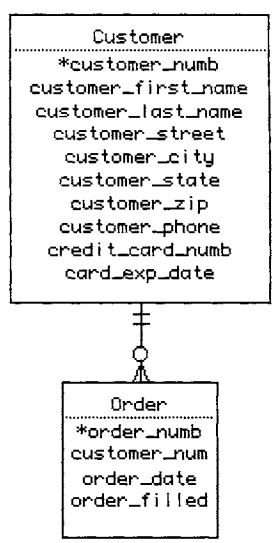



**Figure 2-8: A one-to-many relationship using the IE method**

The double line below the customer entity means that each order is related to one and only one customer. Because zero is not an option, the relationship is mandatory. In contrast, the 0 and three-legged teepee connected to the order entity mean that a customer may have zero, one, or more orders.

There are four symbols used at the ends of lines in an IE diagram:

- ||: One and one only (mandatory relationship)
- 0|: Zero or one
- >|: One or more (mandatory relationship)
- >0: Zero, one, or more

Although we often see the symbols turned 90 degrees, as they are in Figure 2-8, they are actually readable if viewed sideways as in the preceding list.

An IE method ER diagram often includes attributes directly on the diagram. As you can see in Figure 2-8, entity identifiers are marked with an asterisk.

## Basic Relationships for Lasers Only

The major entities in the Lasers Only database are diagrammed in Figure 2-9. You read the relationships in the following way:

- One customer can place zero, one, or more orders. An order comes from one and only one customer.
- An order has one or more items on it. An item can appear on zero, one, or more orders.
- An actor appears in zero, one, or more items. An item has zero, one, or more actors in it. (There may occasionally be films that feature animals rather than human actors; therefore it is probably unwise to require that every merchandise item be related to at least one actor.)
- Each item has zero, one, or more producers. Each producer is responsible for zero, one, or more items. (Although in practice you would not store data about a

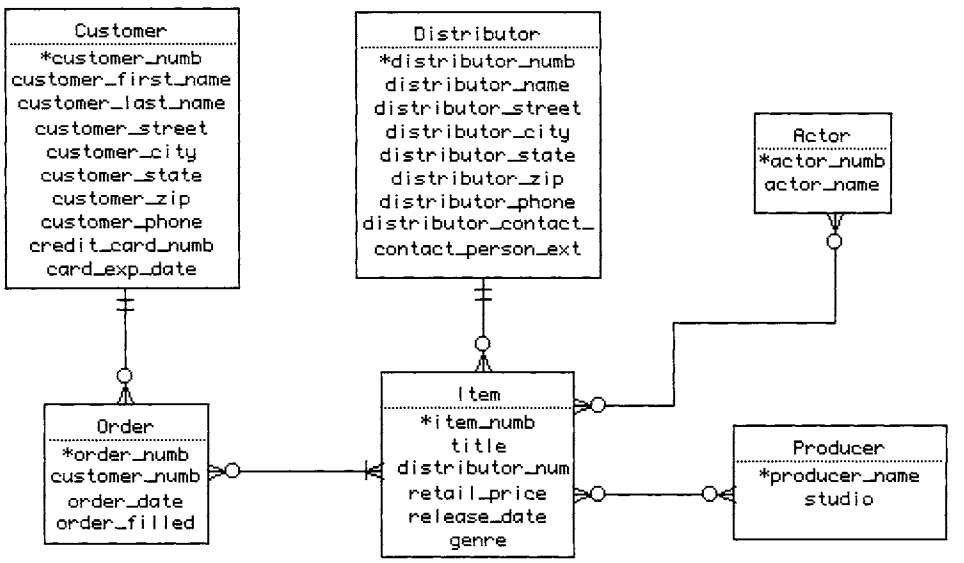



**Figure 2-9: The major entities and the relationships between them in the Lasers Only database**

producer unless that producer was related to an item, leaving the relationship between a producer and an item as optional means that you can store producers without items if necessary.)

The major thing to notice about this design is that there are three many-to-many relationships: order to item, actor to item, and producer to item. Before you can map this data model to a relational database, they must be handled in some way.

# Dealing with Many-to-Many Relationships

As you read earlier, there are problems with many-to-many relationships. The first is fairly straightforward: The relational data model cannot handle many-to-many relationships directly; it is limited to one-to-one and one-to-many relationships. This means that you must replace the many-to-many relationships that you have

identified in your database environment with a collection of one-to-many relationships if you want to be able to use a relational DBMS.

The second is a bit more subtle. To understand it, consider the relationship between an order Lasers Only places with a distributor and the merchandise items on the order. There is a many-to-many relationship between the order and the item because each order can be for many items and, over time, each item can appear on many orders. Whenever Lasers Only places an order for an item, the number of copies of the item varies, depending on the perceived demand for the item at the time the order is placed.

Now the question: Where should we store the quantity being ordered? It cannot be part of the order entity because the quantity depends on which item we are talking about. By the same token, the quantity cannot be part of the item entity because the quantity depends on the specific order.

What you have just seen is known as *relationship data*, data that apply to the relationship between two entities rather than to the entities themselves. Relationships, however, cannot have attributes. We therefore must have some entity to represent the relationship between the two, an entity to which the relationship data can belong.

## Composite Entities

Entities that exist to represent the relationship between two other entities are known as *composite* entities. As an example of how composite entities work, consider the relationship between an order placed by a Lasers Only customer and the items on the order. There is a many-to-many relationship between an item and an order: An order can contain many items and over time the same item can appear on many orders.

What we then need is an entity that tells us that a specific title appears on a specific order. If you look at Figure 2-10, you will see three order instances and three merchandise item instances. The first order for customer 0985 (Order #1) contains only one item

(item 09244). The second order for customer 0985 (Order #2) contains a second copy of item 02944 as well as item 10101. Order #3, which belongs to customer 1212, also has two items on it (item 10101 and item 00250).

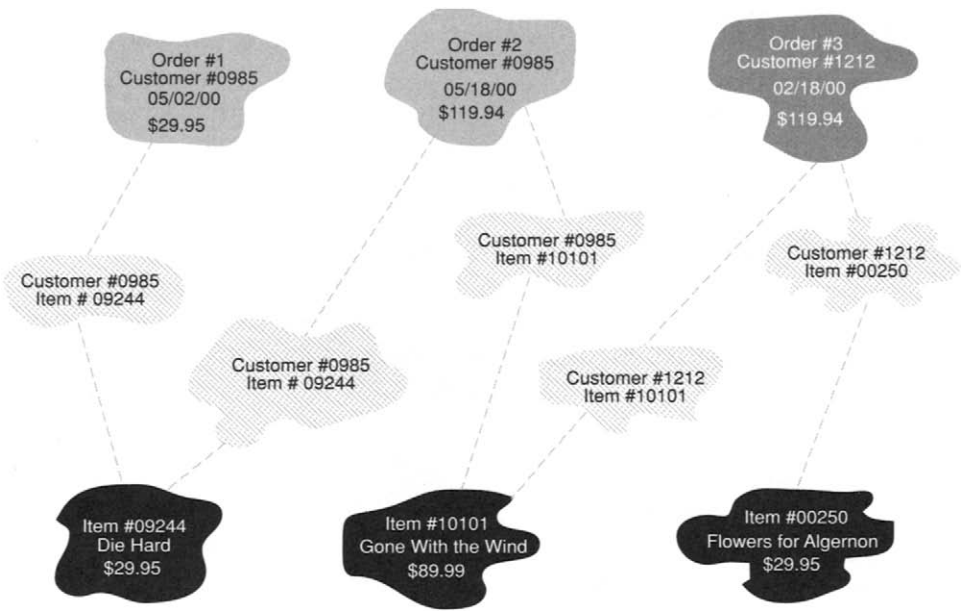



**Figure 2-10: Using instances of composite entities to change many-to-many relationships into one-to-many relationships**

There are five items ordered among the three orders. The middle of the diagram therefore contains five instances of a composite entity we will call a "line item" (thinking of it as a line item on a packing slip). The line item entity has been created solely to represent the relationship between an order and a merchandise item.

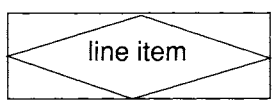
Each order is related to one line item instance for each item on the order. In turn, each item is related to one line item instance for each order on which it appears. Each line item instance is related to one and only one order; it is also related to one and only one merchandise item. As a result, the relationship between an order and its line items is one-to-many (one order has many line items) and the

relationship between an item and the orders on which it appears is one-to-many (one merchandise item appears in many line items). The presence of the composite entity has removed the original many-to-many relationship and turned it into two one-to-many relationships.

If necessary, the composite entity can be used to store relationship data. In the preceding example, we might include an attribute for the quantity ordered, a flag to indicate whether it has been shipped, and a shipping date.

## Documenting Composite Entities

In some extensions of the Chen method for drawing ER diagrams, the symbol for a composite entity is the combination of the rectangle used for an entity and the diamond used for a relationship:




The Information Engineering method, however, has no special symbol for a composite entity.

## Resolving Lasers Only's Many-to-Many Relationships

To eliminate Lasers Only's many-to-many relationships, the database designer must replace each many-to-many relationship with a composite entity and two one-to-many relationships. As you can see in Figure 2-11, the three new entities are as follows:

- Order lines: The order lines entity represents one item appearing on one order. Each order can have many "order lines," but an order line must appear on one and only one order. By the same token, an order line contains one and only one item but the same item can appear on many order lines, each of which corresponds to a different order.

- Performance: The performance entity represents one actor appearing in one film. Each performance is for one and only one film although a film can have many performances (one for each actor in the film). Conversely, an actor is related to one performance for each film in which he or she appears although each performance is in one and only one film.
- Production: The production entity represents one producer working on one film. A producer may be involved in many productions, although each production relates to one and only one producer. The relationship with item indicates that each film can be produced by many producers but that each production relates to only one item.

*Note: If you find sorting out the relationships in Figure 2-11 a bit difficult, keep in mind that if you rotate the up-and-down symbols 90 degrees, you will actually be able to read the relationships.*

Because composite entities exist primarily to indicate a relationship between two other entities, they must be related to both of their parent entities. This is why the relationship between each composite entity in Figure 2-11 and its parents is mandatory.

## Relationships and Business Rules

In many ways, database design is as much an art as a science. Exactly what is the "correct" design for a specific business depends on the business rules; what is correct for one organization may not be correct for another.

As an example, assume that you are creating a database for a retail establishment that has more than one store. One of the things you are being asked to model in the database is an employee's schedule. Before you can do that, you need to answer the question of the relationship between an employee and a store: Is it one-to-many or many-to-many? Does an employee always work at only one store—

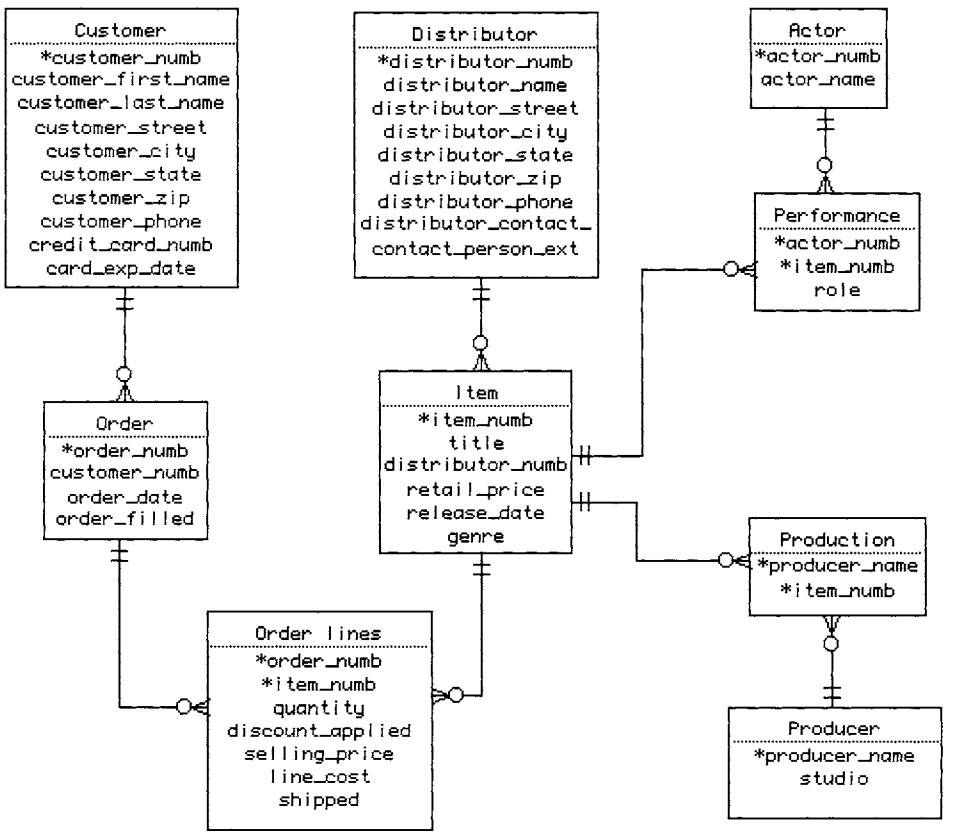



**Figure 2-11: The complete ER diagram for the Lasers Only database**

in which case the relationship is one-to-many—or can an employee split his or her time between more than one store, producing a many-to-many relationship? This is not a matter of right or wrong database design, but an issue of how the business operates.

The bottom line is that no matter how much you know about database design, you will not have a good database unless the relationships depicted in that database are an accurate reflection of the relationships in the database environment.

# Data Modeling versus Data Flow

One of the most common mistakes people make when they are beginning to do data modeling is to confuse data models with data flows. A *data flow* shows how data are handled within an organization, including who handles the data, where the data are stored, and what is done to the data. In contrast, a *data model* depicts the internal, logical relationships between the data, without regard to who is handling the data or what is being done with them.

Data flows are often documented in data flow diagrams (DFDs). For example, in Figure 2-12 you can see a top-level data flow diagram for Lasers Only. The squares represent the people who are handling the data. Circles represent *processes*, or things that are done with the data. A place where data are stored (a *data store*) appears as two parallel lines, in this example containing the words "Main database." The arrows on the lines show the way in which data pass from one place to another.

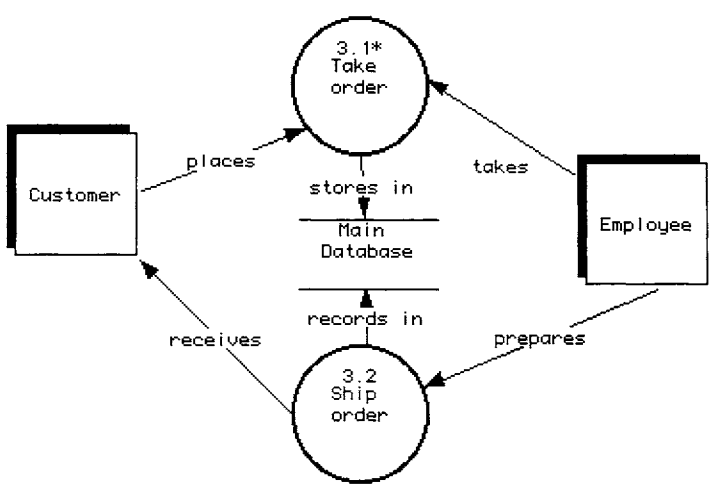



**Figure 2-12: A top-level data flow diagram for Lasers Only**

Data flow diagrams are often exploded to show more detail. For example, Figure 2-13 contains an explosion of the "Take Order"

process from Figure 2-12. You can now see that the process of taking an order involves two major steps: getting customer information and getting item information.

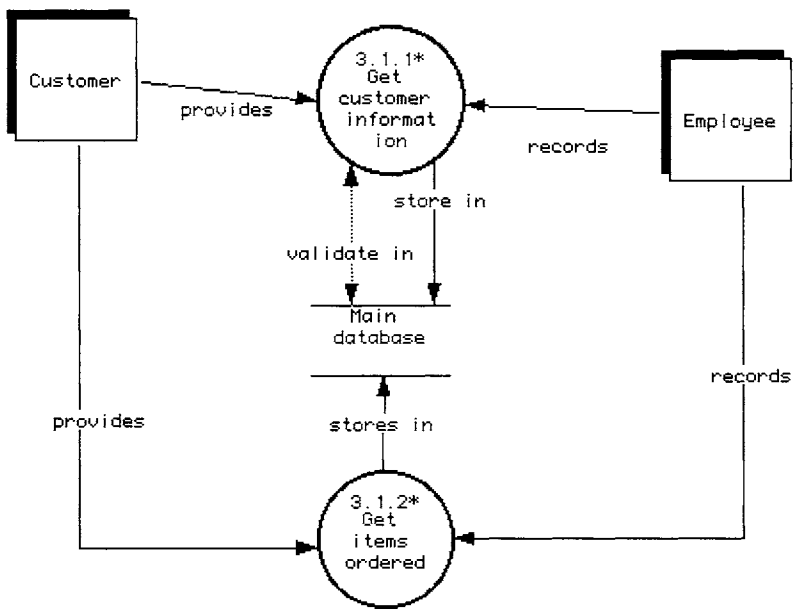



**Figure 2-13: An explosion of the Take order process from Figure 2-12**

Each of the processes in Figure 2-13 can be exploded even further to show additional detail (see Figure 2-14 and Figure 2-15). At this point, the diagrams are almost detailed enough so that an application designer can plan an application program.

Where do the database and its ER diagram fit into all of this? The entire ER diagram is buried inside the "Main database." In fact, most CASE tools allow you to link your ER diagram to a database's representation on a data flow diagram. Then, you can simply double-click on the database representation to bring the ER diagram into view.

> *Note: Don't forget that you can read a great deal more about using a CASE tool in Chapter 9.*

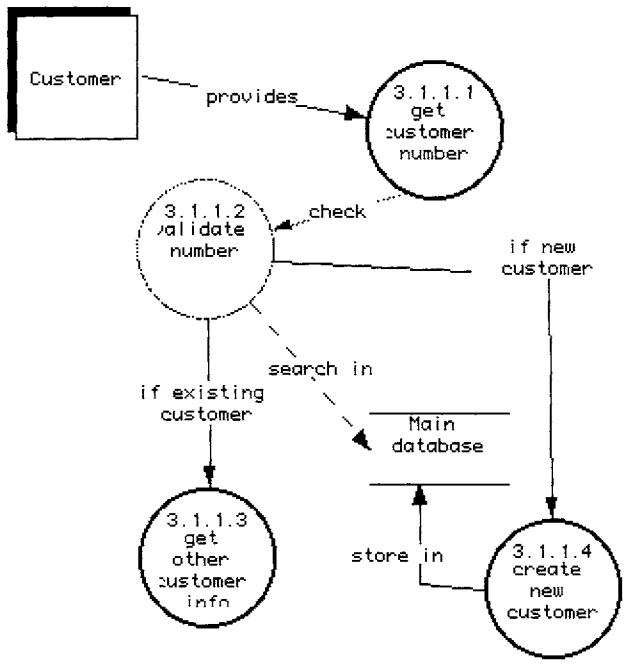



**Figure 2-14: An explosion of the "Get customer information" process from Figure 2-13**

There are a few guidelines you can use to keep data flows and data models separate:

- A data flow shows who uses or handles data. A data model does not.
- A data flow shows how data are gathered (the people or other sources from which they come). A data model does not.
- A data flow shows operations on data (the processes through which data are transformed). A data model does not.
- A data model shows how data entities are interrelated. A data flow does not.
- A data model shows the attributes that describe data entities. A data flow does not.

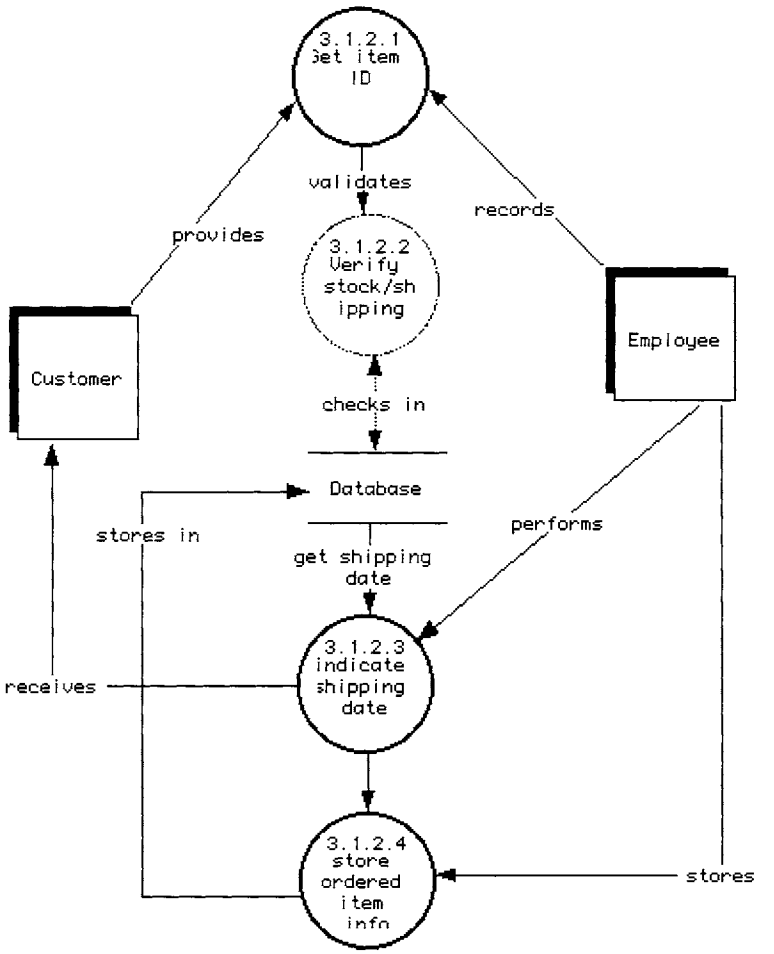



**Figure 2-15: An explosion of the "Get items ordered" process from Figure 2-13**

The bottom line is this: A data model contains information about the data being stored in a database (entities, attributes, and entity relationships). If data about an entity are not going to be stored in the database, then that entity should not be part of the database. For example, although the Lasers Only data flow diagram shows the Lasers Only employee who handles most of the data, no data about employees are going to be stored in the database. Therefore, there is no employee entity in the ER diagram.

# Schemas

A completed entity-relationship diagram represents the overall, logical plan of a database. In database terms, it is therefore known as a *schema*. This is the way in which the people responsible for maintaining the database will see the design. However, users (both interactive users and application programs) may work with only a portion of the logical schema. And both the logical schema and the users' views of the data are at the same time distinct from the physical storage.

The underlying physical storage, which is managed by the DBMS, is known as the *physical schema*. It is for the most part determined by the DBMS. (Only very large DBMSs give you any control over physical storage.) The beauty of this arrangement is that both database designers and users do not need to be concerned about physical storage, greatly simplifying access to the database and making it much easier to make changes to both the logical and physical schemas.

Because there are three ways to look at a database, some databases today are said to be based on a *three-schema architecture* (see Figure 2-16). Systems programmers and other people involved with managing physical storage deal with the physical schema. Most of today's relational DBMSs provide almost no control over the file structures used to store database data. However, DBMSs designed to run on mainframes to handle extremely large datasets do allow some tailoring of the layout of internal file storage.

> *Note: As you will see in Chapter 3, DBMSs based on earlier data models were more closely tied to their physical storage than relational DBMSs. Therefore, systems programmers were able to specify physical file structures to a much greater extent.*

Database designers, database administrators, and some application programmers are aware of and use the logical schema. End users working interactively and application programmers who are

creating database applications for them work with user views of the database.

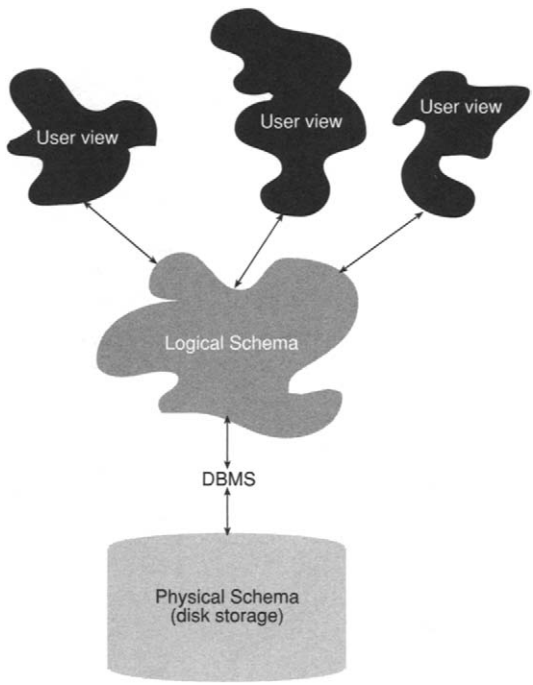



**Figure 2-16: The three-schema architecture**

Throughout the rest of this book we will be focusing on the design of the logical schema. You will also learn how to create and use database elements that provide users with limited portions of the database.

# For Further Reading

The entity–relationship model was developed by Dr. Peter P. S. Chen. If you want to learn more about its early forms and how the model has changed, see the following:

Chen, P. "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transactions on Database Systems*. Vol. I, No. 1, March, 1976.

Chen, P. *The Entity-Relationship Approach to Logical Database Design*. QED Information Sciences, Data Base Monograph Series, No. 6, 1977.

Chen, P. *Entity-Relationship Approach to Information Modeling*. E-R Institute, 1981.

The original work that described Information Engineering can be found in the following:

Martin, James. *Information Engineering, Book I: Introduction, Book II: Planning and Analysis, Book III: Design and Construction*. Prentice Hall, 1989.

Finkelstein, Clive. *An Introduction to Information Engineering*. Addison-Wesley, 1989.

For more recent, in-depth coverage of ER diagramming, you can consult either of the following:

Barker, Richard. *Case*Method: Entity Relationship Modelling*. Addison-Welsey, 1990.

Thalheim, Richard. *Entity-Relationship Modeling: Foundations of Database Technology*. Springer Verlag, 2000.