CHAPTER 15

Database Design Case Study #3: SmartMart

Abstract

This chapter presents the third of three major case studies in the book. It presents a large retail environment that includes multiple stores and warehouses. The design examines additional features of database design, including reference entities, circular relationships, and mutually-exclusive relationships. It also revisits the issue of true one-to-one relationships.

Keywords

database design
database case study
ER diagrams
SQL CREATE
reference entities
circular relationships
mutually exclusive relationships
one-to-one relationships

Many retail chains today maintain both a Web and a brick-and-mortar presence in the marketplace. Doing so presents a special challenge for inventory control because the inventory is shared between physical stores and Web sales. The need to allow multiple shipping addresses and multiple payment methods within a single order also adds complexity to Web selling. Online shopping systems also commonly allow users to store information about multiple credit cards.

To familiarize you with what is necessary to maintain the data for such a business, we'll be looking at a database for SmartMart, a long-established retailer with 325 stores across North America that has expanded into Web sales. SmartMart began as a local grocery store, but over the years has expanded to carry clothing, sundries, home furnishings, hardware, and electronics. Some stores still have grocery departments; others carry just "dry" goods.

In addition to the retail stores, SmartMart maintains four regional warehouses that supply the stores as well as ship products ordered over the Web.

The Merchandising Environment

SmartMart has three major areas for which it wants an integrated database: in-store sales, Web sales, and some limited Human Resources needs. The sales data must be compatible with accounting systems to simplify data transfer. In addition, both the in-store sales and Web sales applications must use the same data about products.

Product Requirements

The products that SmartMart sells are stocked throughout the company's stores, although every store does not carry every product. The database must therefore include data about the following:

- Products
- Stores
- Departments within stores
- Products stocked within a specific department
- Current sales promotions for a specific product

The store and department data will need to be integrated with the database's Human Resources data.

In-Store Sales Requirements

The data describing in-store sales serve two purposes: accounting and inventory control. Each sale (whether paid with cash or credit) must decrement inventory and provide an audit trail for accounting.

Retaining data about in-store sales is also essential to SmartMart's customer service reputation. The company offers a 15-day return period, during which a customer can return any product with which he or she is not satisfied. Store receipts therefore need to identify entire sales transactions, in particular which products were purchased during a specific transaction.

Because the company operates in multiple US states, there is a wide variety of sales tax requirements. Which products are taxed varies among states, as well as the sales tax rates. The database must therefore include sales tax where necessary as a part of an in-store transaction.

The database must distinguish between cash and credit transactions. The database will not store customer data about cash transactions, but must retain card numbers, expiration dates, and customer names on credit sales.

retain card numbers, expiration dates, and customer names on credit sales.

Web Sales Requirements

Web sales add another layer of complexity to the SmartMart data environment. The Web application must certainly have access to the same product data as the in-store sales, but must also integrate with a shopping cart application.

To provide the most flexibility, SmartMart wants to allow customers to store multiple shipping addresses, to ship to multiple addresses on the same order, and to store multiple credit card data from which a customer can choose when checking out. In addition, customers are to be given a choice as to whether to pick up their order or have it shipped. The Web application must therefore have access to data about which stores are in a customer's area and which products are available at each store, the same inventory information used by in-store applications.

Finally, the Web application must account for backorders and partial shipments. This means that a shipment is not the same as a Web order, whereas an in-store sale delivers its items at one time. (Should an in-store customer want to purchase an item that is not in stock, the item will be handled as if it were a Web order.)

Personnel Requirements

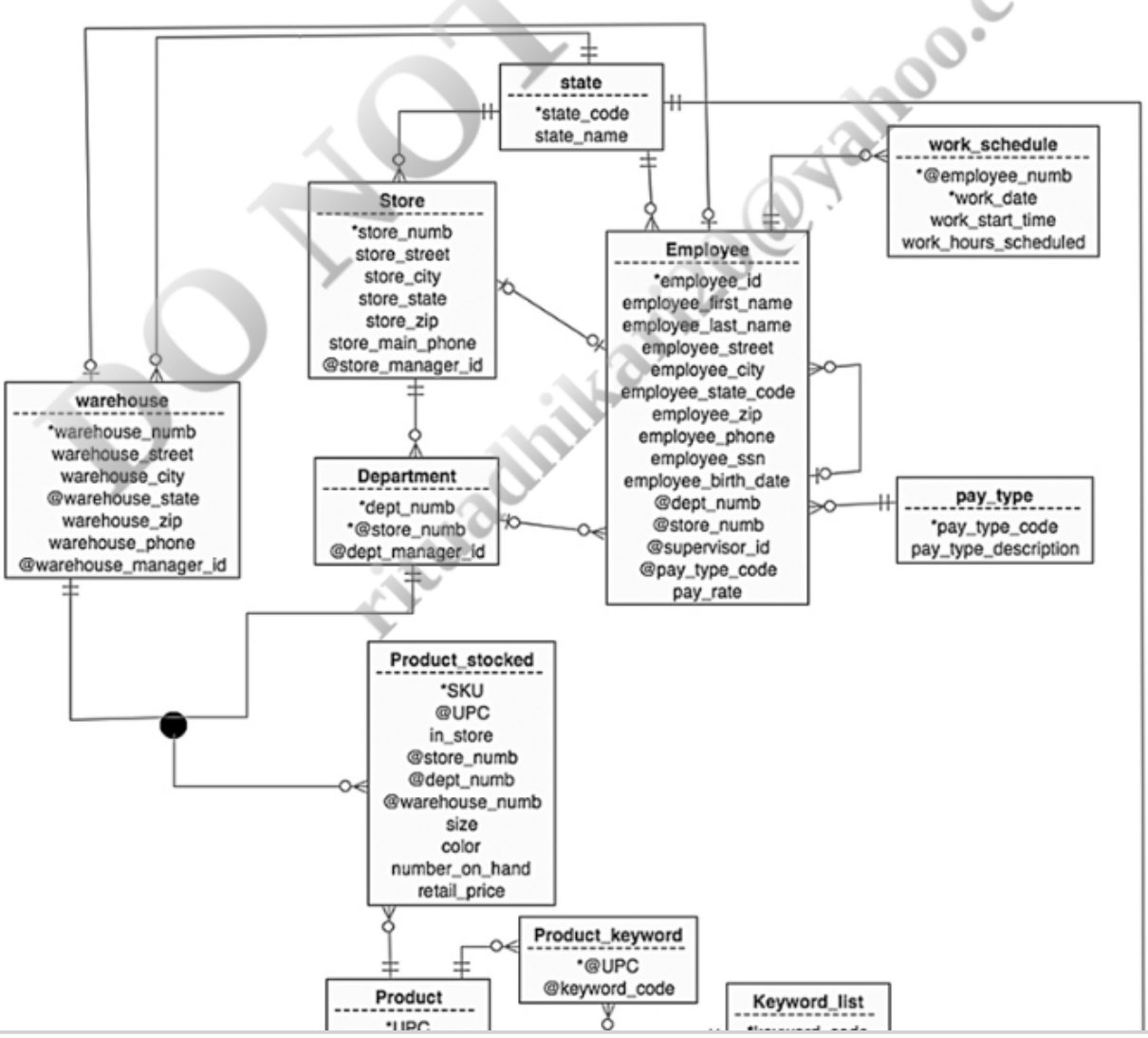
Although a complete personnel database is beyond the scope of this case, SmartMart's management does want to be able to integrate some basic HR functions into the database, especially the scheduling of "sales associates" to specific departments in specific stores. The intent is to eventually be able use an expert system to analyze sales and promotion data to determine staffing levels and to move employees among the stores in a region as needed.

Putting Together an ERD

As you might expect, the SmartMart ERD is fairly large. It therefore has been broken into three parts to make it easier to see and understand.

Stores, Products, and Employees

As you can see in Figure 15.1 (the first third of the ERD), the SmartMart database begins with four "foundation" entities (entities that are only at the "one" end of 1:M relationships): *employee*, *store*, *warehouse*, and *product*.



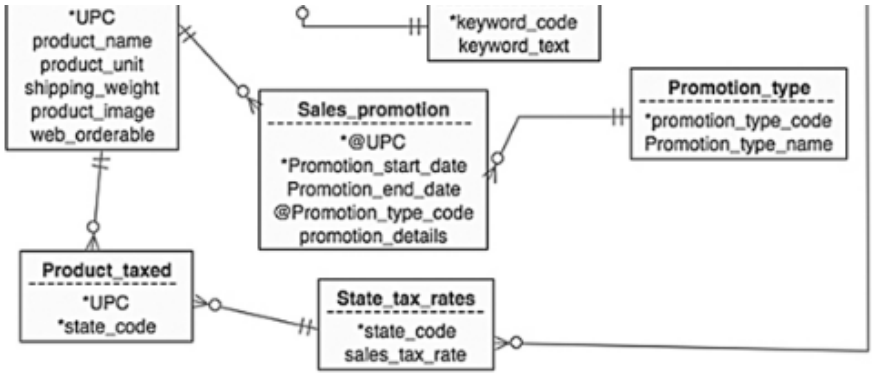


FIGURE 15.1 **The SmartMart ERD (part 1).**

The *store* and *warehouse* entities, at least at this time, have exactly the same attributes. It certainly would be possible to use a single entity representing any place products were kept. This would remove some of the complexity that arises when locating a product. However, there is no way to be certain that the data stored about a store and a warehouse will remain the same over the life of the database. Separating them into two entities after the database has been in use for some time would be very difficult and time consuming. Therefore, the choice was made to handle them as two distinct entities from the beginning.

Reference Entities

There are also several entities that are included for referential integrity purposes (*state*, *keyword_list*, *pay_type*, *promotion_type*). These entities become necessary because the design attempts to standardize text descriptions by developing collections of acceptable text descriptions and then using a numbering scheme to link the descriptions to places where they are used. There are two major benefits to doing this.

First, when the text descriptions, such as a type of pay (for example, hourly versus salaried), are standardized, searches will be more accurate. Assume, for example, that the pay types haven't been standardized. An employee who is paid hourly might have a pay type of "hourly," "HOURLY," "hrly," and so on. A search that retrieves all rows with a pay type of "hourly" will miss any rows with "HOURLY" or "hrly," however.

Second, using integers to represent values from the standardized list of text descriptions saves space in the database. Given the relative low price of disk storage, this usually isn't a major consideration.

The drawback, of course, is that when you need to search on or display the text description, the relation containing the standardized list and the relation using the integers representing the terms must be joined. Joins are relatively slow activities, but, in this case, the reference relation containing the master list of text descriptions will be relatively small; the join uses integer columns, which are quick to match. Therefore, unless for some reason the reference relation becomes extremely large, the overhead introduced by the join is minimal.

Circular Relationships

If you look closely at the *employee* entity in Figure 15.1, you'll see a relationship that seems to relate the entity to itself. In fact, this is exactly what that circular relationship does. It represents the idea that a person who supervises other employees is also an employee: The *supervisor_id* attribute is drawn from the same domain as *employee_id*. Each supervisor is related to many employees and each employee has only one supervisor.

There is always a temptation to create a separate *supervisor* entity. Given that a supervisor must also be an employee, however, the *supervisor* entity would contain data duplicated from the *employee* entity. This means that we introduce unnecessary duplicated data into the database and run a major risk of data inconsistencies.

Note: To retrieve a list of supervisors and who they supervise, someone using SQL would join a copy of the employee table to itself, matching the supervisor_id column in one table to the employee_id column in the other. The result table would contain data for two employees in each row (the employee and the employee's supervisor) that could be manipulated—in particular, sorted—for output, as needed.

Mutually Exclusive Relationships

There is one symbol on the ERD in Figure 15.1 that has not been used before in this book: the small circle that sits in the middle of the relationships between a stocked product, a department (in a store), and a warehouse. This type of structure indicates a *mutually exclusive* relationship. A given product can be stocked in a store or in a warehouse, but not both. (This holds true for this particular data environment because a product stocked represents physical items to be sold.)

The structure of the *product_stocked* entity reflects its participation in this type of relationship. In particular, it contains a Boolean column (*in_store*) that holds a value of true if the product is stocked in a store; a value of false indicates that the product is stocked in a warehouse. The value of the *in_store* attribute will then tell a user whether to use the *warehouse_numb* column or the concatenation of the *store_numb* column with the *dept_numb* attribute to find the actual location of an item.

One-to-one Relationships

Earlier in this book, you read that true one-to-one relationships are relatively rare. There are, however, three of them visible in Figure 15.1. All involve employees that manage something: a store, a department within a store, or a warehouse. A corporate unit may have one manager at a time, or it may have no manager; an employee may be the manager of one corporate unit or the manager of none. It is the rules of this particular database environment that make the one-to-one relationships valid.

In-store Sales

The second part of the ERD (Figure 15.2) deals with in-store sales. The data that are common to cash and credit sales are stored in the *in_store_sale* entity. These data are all that are needed for a cash sale. Credit sales, however, require data about the credit card used (the *credit_sale_details* entity). Notice that there is therefore a one-to-one relationship between *in_store_sale* and *credit_sale_details*. The two-entity

credit_sale_details entity). Notice that there is therefore a one-to-one relationship between *in_store_sale* and *credit_sale_details*. The two-entity design is not required for a good database design, but has been chosen for performance reasons with the assumption that although currently there are more credit/debit than cash transactions, the data retrieval is very skewed, requiring customer data far more frequently than credit card details. It therefore is a way of providing horizontal partitioning for an access pattern that is "unbalanced." In other words, a small proportion of the queries of sales data will need credit data.

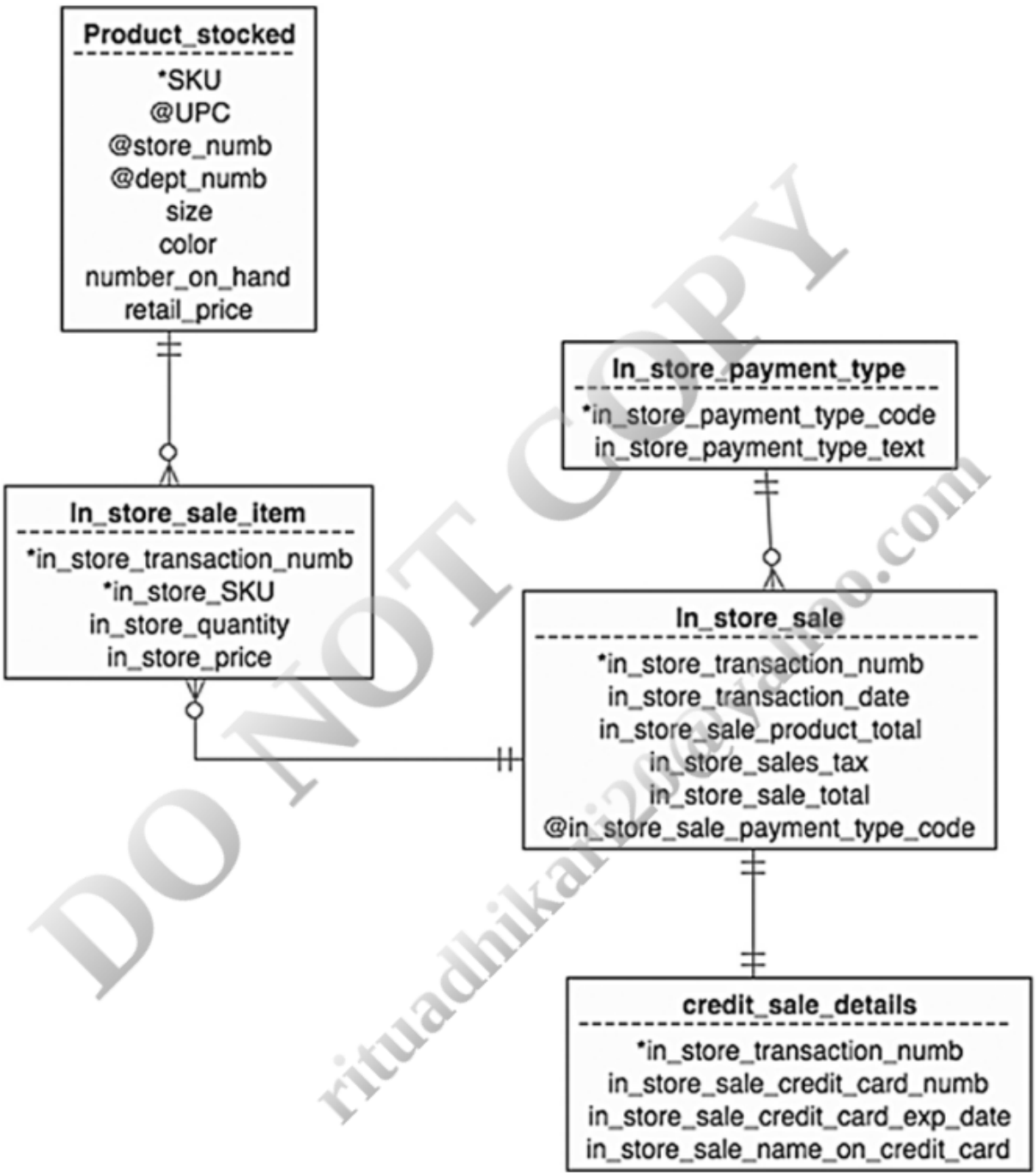


FIGURE 15.2 **The SmartMart ERD (part 2).**

After SmartMart's database has been in production for some time, the database administrator can look at the actual proportion of retrievals that need the credit card details. If a large proportion of the retrievals include the credit/debit data, then it may make sense to combine *in_store_sale* and *credit_sale_details* into a single entity and simply leave the credit detail columns as null for cash sales. Although some space will be wasted, the combined design avoids the need to perform a lengthy join when retrieving data about a credit sale.

Web Sales

The third portion of the ERD (Figure 15.3) deals with Web sales. Each Web sale uses only one credit card, but a Web customer may keep more than one credit card number within SmartMart's database. A customer may also keep more than one shipping address; multiple shipping addresses can be used within the same order. (Multiple credit cards for a single order are not supported.)

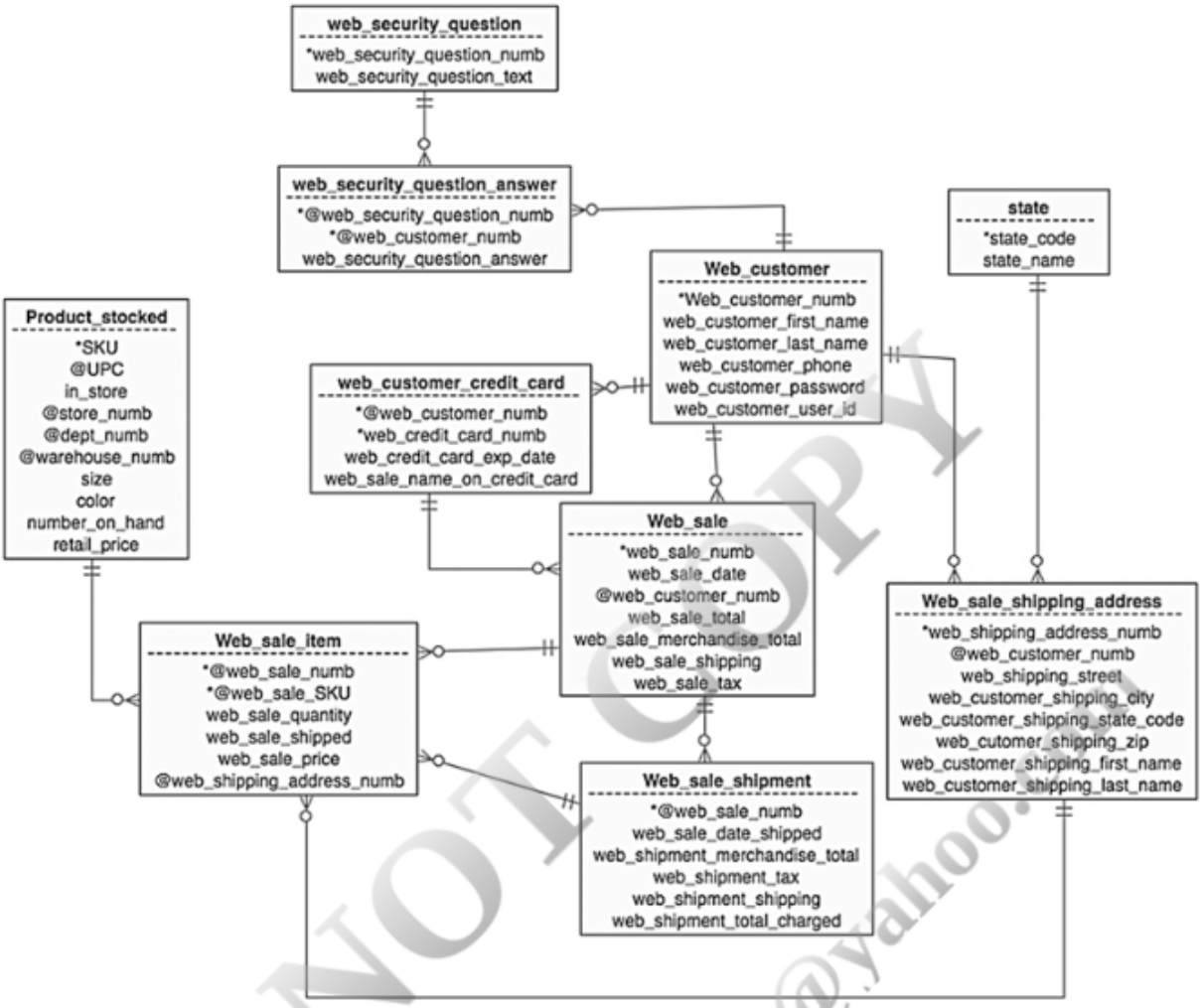


FIGURE 15.3 **The SmartMart ERD (part 3).**

At first glance, it might appear that the three relationships linking *web_customer*, *web_sale*, and *web_customer_credit_card* form a circular relationship. However, the meaning of the relationships is such that the circle does not exist:

- The direct relationship between *web_customer* and *web_customer_credit_card* represents the cards that a web customer has allowed to be stored in the SmartMart database. The credit card data are retrieved when the customer is completing a purchase. He or she chooses one for the current order.
- The relationship between *web_customer* and *web_sale* connects a purchase to a customer.
- The relationship between *web_customer_credit_card* and *web_sale* represents the credit card used for a specific order.

It can be difficult to discern such subtle differences in meaning from a simple ERD. There are two solutions: define the meaning of the relationships in the data dictionary or add relationship names to the ERD.

The *web_sale_shipping_address* entity is used to display addresses from which a user can choose. However, because items within the same shipment can be sent to different addresses, there is a many-to-many relationship between *web_sale* and *web_sale_shipping_address*. The *web_sale_item* resolves that many-to-many relationship into two one-to-many relationships.

The relationship between *web_sale*, *web_sale_item*, and *web_sale_shipment* is ultimately circular, as it appears in the ERD. However, at the time the order is placed, there are no instances of the *web_sale_shipment* entity because no items have shipped. The circle is closed when items actually ship. SmartMart's database can handle multiple shipments for a single order. However, each item on an order is sent to the customer in a single shipment. How do we know this? Because here is a 1:M relationship between a web sale and a web sale shipment (a sale may have many shipments) and a 1:M relationship between a shipment and an item on the order (each item is shipped as part of only one shipment).

Creating the Tables

The ERDs you have just seen produce the following tables, listed in alphabetical order:

```
credit_sale_details (in_store_transaction_numb,
    in_store_sale_credit_card_numb, in_store_exp_date,
    in_store_sale_name_on_credit_card)
```

```
department (dept_numb, store_numb, dept_manager_id)

employee (employee_id, employee_first_name, employee_last_name,
     employee_street, employee_city, employee_state_code,
     employee_zip, employee_phone, employee_ssn,
     employee_birth_date, dept_id, store_numb, supervisor_id,
     pay_type_code, pay_rate)

in_store_payment_type (in_store_payment_type_code,
    in_store_payment_type_text)

in_store_sale (in_store_transaction_numb,
     in_store_transaction_date, in_store_sale_product_total,
     in_store_sales_tax,in_store_total,
     in_store_payment_type_code)

in_store_sale_item (in_store_sale_transaction_numb,
     in_store_SKU, in_store_quantity, in_store_price)

keyword_list (keyword_code, keyword_text)

pay_type (pay_type_code, pay_type_description)

product (UPC, product_name, product_unit, shipping_weight,
     product_image, web_orderable)

product_keyword (UPC, keyword_code)

product_stocked (SKU, UPC, in_store, store_numb, dept_numb,
     warehouse_numb, size, color, number_on_hand, retail_price)

product_taxed (UPC, state_code)

promotion_type (promotion_type_code, promotion_type_name)

sales_promotion (UPC, promotion_start_date, promotion_end_date,
     promotion_type_code, promotion_details)

state (state_code, state_name)

state_tax_rates (state_code, sales_tax_rate)

store (store_numb, store_street, store_city, store_state_code,
     store_zip, store_main_phone, store_manager_id)
```

```
warehouse (warehouse_id, warehouse_street, warehouse_city,
     ware_house_state_code, warehouse_zip, warehouse_phone,
     warehouse_manager_id)

web_customer (web_customer_numb, web_customer_first_name,
     web_customer_last_name, web_customer_phone,
     web_customer_password, web_customer_user_id)

web_customer_credit_card (web_customer_numb,
     web_credit_card_numb, web_credit_card_exp_date,
     web_sale_name_on_credit_card)

web_sale (web_sale_numb, web_sale_date, web_customer_numb,
     web_sale_total, web_sale_merchandise_total,
     web_sale_shipping,web_sale_tax)

web_sale_item (web_sale_numb, web_sale_SKU,
     web_sale_quantity, web_sale_shipped, web_sale_price,
     web_shipping_address_numb)

web_sale_shipment (web_sale_numb, web_sale_date_shipped,
     web_shipment_merchandise_total, web_shipment_tax,
     web_shipment_shipping, web_shipment_total_charged)

web_sale_shipping_address (web_shipping_address_numb,
     web_customer_numb, web_shipping_street,
     web_shipping_city, web_shipping_state_code,
     web_customer_zip, web_customer_shipping_first_name,
     web_customer_shipping_last_name)

web_security_question (web_security_question_numb,
     web_security_question_text)

web_security_question_answer (web_security_question_numb,
     web_customer_numb, web_security_answer_text)

work_schedule (employee_id, work_date, work_start_time,
     work_hours_scheduled)
```

Because of the circulation relationship between a supervisor, who must be an employee, and an employee being supervised, the employee table contains a foreign key that references the primary key of its own table: *supervisor_id* is a foreign key referencing *employee_id*. There is nothing wrong with this type of design. The definition of a foreign key states only that the foreign key is the same as the primary key of some table; it does not rule out the foreign key referencing the primary key of its own table.

Generating the SQL

The case tool that generated the SQL misses some very important foreign keys—the relationships between employees and various other entities—because the two columns don't have the same name. Therefore, the database designer must add the constraints manually to the foreign key tables' CREATE TABLE statements:

```
employee table: FOREIGN KEY (supervisor_id)
     REFERENCES employee (employee_id)

warehouse table: FOREIGN KEY (warehouse_manager_id)
     REFERENCES employee (employee_id)

department table: FOREIGN KEY (department_manager_id)
     REFERENCES employee (employee_id)

store table: FOREIGN KEY (store_manager_id)
     REFERENCES employee (employee_id)
```

The manual foreign key insertions could have been avoided, had the manager IDs in the warehouse, department, and store tables been given the name *employee_id*. However, it is more important in the long-run to have the column names reflect the meaning of the columns.

There is also one foreign key missing of the two needed to handle the mutually exclusive relationship between a product stocked and either a department (in a store) or a warehouse. The foreign key from *product_stocked* to department is present, but not the reference to the warehouse table. The database designer must, therefore, add the following to the *product_stocked* table:

```
FOREIGN KEY (warehouse_numb)
     REFERENCES warehouse (warehouse_numb)
```

Won't there be a problem with including the two constraints, if only one can be valid for any single row in *product_stocked*? Not at all. Remember that referential integrity constraints are only used when the foreign key is not null. Therefore, a product stocked in a warehouse will have a value in its *warehouse_numb* column, but null in the *store_numb* and *dept_numb* columns. The reverse is also true: A product stocked in a store will have values in its *store_numb* and *dept_numb* columns, but null in the *warehouse_numb* column.

The foreign key relationships to the *state* reference relation must also be added manually, because the foreign key columns do not have the same name as the primary key column in the *state* table. Whether to use the same name throughout the database (*state_name*) is a design decision. If all tables that contain a state use the same name, foreign keys will be added automatically by the CASE tool. However, depending on the way in which the states are used, it may be difficult to distinguish among them, if the column names are all the same (Figure 15.4).