physical design choices (e.g., indexed fields) can be modeled in UML database diagrams.

UML's **component** diagrams describe storage aspects of the database, such as *tablespaces* and *database partitions*), as well as interfaces to applications that access the database. Finally, **deployment** diagrams show the hardware aspects of the system.

Our objective in this book is to concentrate on the data stored in a database and the related design issues. To this end, we deliberately take a simplified view of the other steps involved in software design and development. Beyond the specific discussion of UML, the material in this section is intended to place the design issues that we cover within the context of the larger software design process. We hope that this will assist readers interested in a more comprehensive discussion of software design to complement our discussion by referring to other material on their preferred approach to overall system design.

2.8 CASE STUDY: THE INTERNET SHOP

Start reading from here!

We now introduce an illustrative, 'cradle-to-grave' design case study that we use as a running example throughout this book. DBDudes Inc., a well-known database consulting firm, has been called in to help Barns and Nobble (B&N) with its database design and implementation. B&N is a large bookstore specializing in books on horse racing, and it has decided to go online. DBDudes first verifies that B&N is willing and able to pay its steep fees and then schedules a lunch meeting—billed to B&N, naturally—to do requirements analysis.

2.8.1 Requirements Analysis

The owner of B&N, unlike many people who need a database, has thought extensively about what he wants and offers a concise summary:

"I would like my customers to be able to browse my catalog of books and place orders over the Internet. Currently, I take orders over the phone. I have mostly corporate customers who call me and give me the ISBN number of a book and a quantity; they often pay by credit card. I then prepare a shipment that contains the books they ordered. If I don't have enough copies in stock, I order additional copies and delay the shipment until the new copies arrive; I want to ship a customer's entire order together. My catalog includes all the books I sell. For each book, the catalog contains its ISBN number, title, author, purchase price, sales price, and the year the book was published. Most of my customers are regulars, and I have records with their names and addresses.

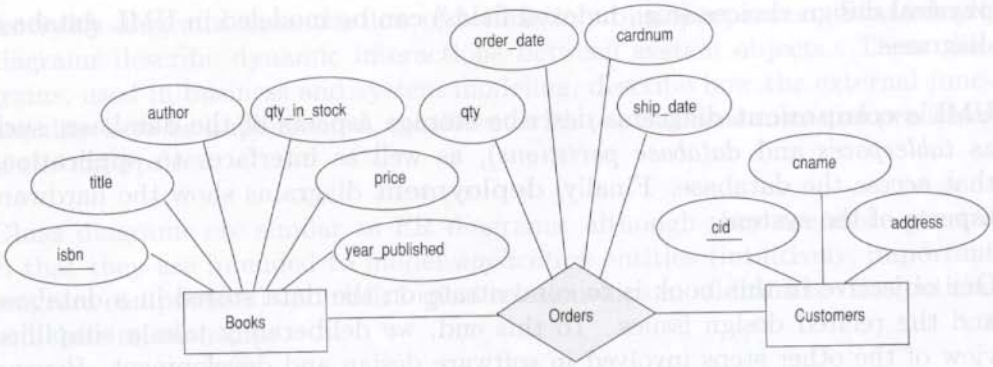

Figure 2.20 ER Diagram of the Initial Design

New customers have to call me first and establish an account before they can use my website.

On my new website, customers should first identify themselves by their unique customer identification number. Then they should be able to browse my catalog and to place orders online."

DBDudes's consultants are a little surprised by how quickly the requirements phase is completed—it usually takes weeks of discussions (and many lunches and dinners) to get this done—but return to their offices to analyze this information.

2.8.2 Conceptual Design

In the conceptual design step, DBDudes develops a high level description of the data in terms of the ER model. The initial design is shown in Figure 2.20. Books and customers are modeled as entities and related through orders that customers place. Orders is a relationship set connecting the Books and Customers entity sets. For each order, the following attributes are stored: quantity, order date, and ship date. As soon as an order is shipped, the ship date is set; until then the ship date is set to *null*, indicating that this order has not been shipped yet.

DBDudes has an internal design review at this point, and several questions are raised. To protect their identities, we will refer to the design team leader as Dude 1 and the design reviewer as Dude 2.

Dude 2: What if a customer places two orders for the same book in one day?
Dude 1: The first order is handled by creating a new Orders relationship and

the second order is handled by updating the value of the quantity attribute in this relationship.

Dude 2: What if a customer places two orders for different books in one day?

Dude 1: No problem. Each instance of the Orders relationship set relates the customer to a different book.

Dude 2: Ah, but what if a customer places two orders for the same book on different days?

Dude 1: We can use the attribute order date of the orders relationship to distinguish the two orders.

Dude 2: Oh no you can't. The attributes of Customers and Books must jointly contain a key for Orders. So this design does not allow a customer to place orders for the same book on different days.

Dude 1: Yikes, you're right. Oh well, B&N probably won't care; we'll see.

DBDudes decides to proceed with the next phase, logical database design; we rejoin them in Section 3.8.

2.9 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- Name the main steps in database design. What is the goal of each step? In which step is the ER model mainly used? **(Section 2.1)**
- Define these terms: *entity*, *entity set*, *attribute*, *key*. **(Section 2.2)**
- Define these terms: *relationship*, *relationship set*, *descriptive attributes*. **(Section 2.3)**
- Define the following kinds of constraints, and give an example of each: *key constraint*, *participation constraint*. What is a *weak entity*? What are *class hierarchies*? What is *aggregation*? Give an example scenario motivating the use of each of these ER model design constructs. **(Section 2.4)**
- What guidelines would you use for each of these choices when doing ER design: Whether to use an attribute or an entity set, an entity or a relationship set, a binary or ternary relationship, or aggregation. **(Section 2.5)**
- Why is designing a database for a large enterprise especially hard? **(Section 2.6)**
- What is UML? How does database design fit into the overall design of a data-intensive software system? How is UML related to ER diagrams? **(Section 2.7)**

```
ALTER TABLE Students
    ADD COLUMN maiden-name CHAR(10)
```

The definition of Students is modified to add this column, and all existing rows are padded with *null* values in this column. ALTER TABLE can also be used to delete columns and add or drop integrity constraints on a table; we do not discuss these aspects of the command beyond remarking that dropping columns is treated very similarly to dropping tables or views.

3.8 CASE STUDY: THE INTERNET STORE

The next design step in our running example, continued from Section 2.8, is logical database design. Using the standard approach discussed in Chapter 3, DBDudes maps the ER diagram shown in Figure 2.20 to the relational model, generating the following tables:

```
CREATE TABLE Books ( isbn            CHAR(10),
                     title           CHAR(80),
                     author          CHAR(80),
                     qty_in_stock    INTEGER,
                     price           REAL,
                     year_published  INTEGER,
                     PRIMARY KEY (isbn))

CREATE TABLE Orders ( isbn        CHAR(10),
                      cid         INTEGER,
                      cardnum     CHAR(16),
                      qty         INTEGER,
                      order_date  DATE,
                      ship_date   DATE,
                      PRIMARY KEY (isbn,cid),
                      FOREIGN KEY (isbn) REFERENCES Books,
                      FOREIGN KEY (cid) REFERENCES Customers )

CREATE TABLE Customers ( cid      INTEGER,
                         cname    CHAR(80),
                         address  CHAR(200),
                         PRIMARY KEY (cid)
```

The design team leader, who is still brooding over the fact that the review exposed a flaw in the design, now has an inspiration. The Orders table contains the field *order_date* and the key for the table contains only the fields *isbn* and *cid*. Because of this, a customer cannot order the same book on different days,

a restriction that was not intended. Why not add the *order_date* attribute to the key for the Orders table? This would eliminate the unwanted restriction:

```
CREATE TABLE Orders (    isbn          CHAR(10),
                         ...
                         PRIMARY KEY (isbn,cid,ship_date),
                         ...)
```

The reviewer, Dude 2, is not entirely happy with this solution, which he calls a 'hack'. He points out that no natural ER diagram reflects this design and stresses the importance of the ER diagram as a design document. Dude 1 argues that, while Dude 2 has a point, it is important to present B&N with a preliminary design and get feedback; everyone agrees with this, and they go back to B&N.

The owner of B&N now brings up some additional requirements he did not mention during the initial discussions: "Customers should be able to purchase several different books in a single order. For example, if a customer wants to purchase three copies of 'The English Teacher' and two copies of 'The Character of Physical Law,' the customer should be able to place a single order for both books."

The design team leader, Dude 1, asks how this affects the shippping policy. Does B&N still want to ship all books in an order together? The owner of B&N explains their shipping policy: "As soon as we have have enough copies of an ordered book we ship it, even if an order contains several books. So it could happen that the three copies of 'The English Teacher' are shipped today because we have five copies in stock, but that 'The Character of Physical Law' is shipped tomorrow, because we currently have only one copy in stock and another copy arrives tomorrow. In addition, my customers could place more than one order per day, and they want to be able to identify the orders they placed."

The DBDudes team thinks this over and identifies two new requirements: First, it must be possible to order several different books in a single order and second, a customer must be able to distinguish between several orders placed the same day. To accomodate these requirements, they introduce a new attribute into the Orders table called *ordernum*, which uniquely identifies an order and therefore the customer placing the order. However, since several books could be purchased in a single order, *ordernum* and *isbn* are both needed to determine *qty* and *ship_date* in the Orders table.

Orders are assigned order numbers sequentially and orders that are placed later have higher order numbers. If several orders are placed by the same customer

on a single day, these orders have different order numbers and can thus be distinguished. The SQL DDL statement to create the modified Orders table follows:

```
CREATE TABLE Orders ( ordernum     INTEGER,
                      isbn         CHAR(10),
                      cid          INTEGER,
                      cardnum      CHAR(16),
                      qty          INTEGER,
                      order_date   DATE,
                      ship_date    DATE,
                      PRIMARY KEY (ordernum, isbn),
                      FOREIGN KEY (isbn) REFERENCES Books
                      FOREIGN KEY (cid) REFERENCES Customers )
```

The owner of B&N is quite happy with this design for Orders, but has realized something else. (DBDudes is not surprised; customers almost always come up with several new requirements as the design progresses.) While he wants all his employees to be able to look at the details of an order, so that they can respond to customer enquiries, he wants customers' credit card information to be secure. To address this concern, DBDudes creates the following view:

```
CREATE VIEW OrderInfo (isbn, cid, qty, order_date, ship_date)
     AS SELECT O.cid, O.qty, O.order_date, O.ship_date
        FROM   Orders O
```

The plan is to allow employees to see this table, but not Orders; the latter is restricted to B&N's Accounting division. We'll see how this is accomplished in Section 21.7.

3.9 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What is a relation? Differentiate between a relation schema and a relation instance. Define the terms *arity* and *degree* of a relation. What are domain constraints? **(Section 3.1)**
- What SQL construct enables the definition of a relation? What constructs allow modification of relation instances? **(Section 3.1.1)**
- What are *integrity constraints*? Define the terms *primary key constraint* and *foreign key constraint*. How are these constraints expressed in SQL? What other kinds of constraints can we express in SQL? **(Section 3.2)**

6.6 CASE STUDY: THE INTERNET BOOK SHOP

DBDudes finished logical database design, as discussed in Section 3.8, and now consider the queries that they have to support. They expect that the application logic will be implemented in Java, and so they consider JDBC and SQLJ as possible candidates for interfacing the database system with application code.

Recall that DBDudes settled on the following schema:

Books(*isbn:* `CHAR(10)`, *title:* `CHAR(8)`, *author:* `CHAR(80)`,
qty_in_stock: `INTEGER`, *price:* `REAL`, *year_published:* `INTEGER`)
Customers(*cid:* `INTEGER`, *cname:* `CHAR(80)`, *address:* `CHAR(200)`)
Orders(*ordernum:* `INTEGER`, *isbn:* `CHAR(10)`, *cid:* `INTEGER`,
cardnum: `CHAR(16)`, *qty:* `INTEGER`, *order_date:* `DATE`, *ship_date:* `DATE`)

Now, DBDudes considers the types of queries and updates that will arise. They first create a list of tasks that will be performed in the application. Tasks performed by customers include the following.

- Customers search books by author name, title, or ISBN.
- Customers register with the website. Registered customers might want to change their contact information. DBDudes realize that they have to augment the Customers table with additional information to capture login and password information for each customer; we do not discuss this aspect any further.
- Customers check out a final shopping basket to complete a sale.
- Customers add and delete books from a 'shopping basket' at the website.
- Customers check the status of existing orders and look at old orders.

Administrative tasks performed by employees of B&N are listed next.

- Employees look up customer contact information.
- Employees add new books to the inventory.
- Employees fulfill orders, and need to update the shipping date of individual books.
- Employees analyze the data to find profitable customers and customers likely to respond to special marketing campaigns.

Next, DBDudes consider the types of queries that will arise out of these tasks. To support searching for books by name, author, title, or ISBN, DBDudes decide to write a stored procedure as follows:

```
CREATE PROCEDURE SearchByISBN (IN book_isbn CHAR(10))
    SELECT B.title, B.author, B.qty_in_stock, B.price, B.year_published
    FROM   Books B
    WHERE  B.isbn = book_isbn
```

Placing an order involves inserting one or more records into the Orders table. Since DBDudes has not yet chosen the Java-based technology to program the application logic, they assume for now that the individual books in the order are stored at the application layer in a Java array. To finalize the order, they write the following JDBC code shown in Figure 6.11, which inserts the elements from the array into the Orders table. Note that this code fragment assumes several Java variables have been set beforehand.

```
String sql = "INSERT INTO Orders VALUES(?, ?, ?, ?, ?, ?)";
PreparedStatement pstmt = con.prepareStatement(sql);
con.setAutoCommit(false);
try {
    // orderList is a vector of Order objects
    // ordernum is the current order number
    // cid is the ID of the customer, cardnum is the credit card number
    for (int i=0; i¡orderList.length(); i++)
        // now instantiate the parameters with values
        Order currentOrder = orderList[i];
        pstmt.clearParameters();
        pstmt.setInt(1, ordernum);
        pstmt.setString(2, Order.getIsbn());
        pstmt.setInt(3, cid);
        pstmt.setString(4, creditCardNum);
        pstmt.setInt(5, Order.getQty());
        pstmt.setDate(6, null);

        pstmt.executeUpdate();
    }
    con.commit();
catch (SQLException e){
    con.rollback();
    System.out.println(e.getMessage());
}
```

Figure 6.11 Inserting a Completed Order into the Database

DBDudes writes other JDBC code and stored procedures for all of the remaining tasks. They use code similar to some of the fragments that we have seen in this chapter.

- Establishing a connection to a database, as shown in Figure 6.2.
- Adding new books to the inventory, as shown in Figure 6.3.
- Processing results from SQL queries as shown in Figure 6.4.
- For each customer, showing how many orders he or she has placed. We showed a sample stored procedure for this query in Figure 6.8.
- Increasing the available number of copies of a book by adding inventory, as shown in Figure 6.9.
- Ranking customers according to their purchases, as shown in Figure 6.10.

DBDudes takes care to make the application robust by processing exceptions and warnings, as shown in Figure 6.6.

DBDudes also decide to write a trigger, which is shown in Figure 6.12. Whenever a new order is entered into the Orders table, it is inserted with ship_date set to NULL. The trigger processes each row in the order and calls the stored procedure 'UpdateShipDate'. This stored procedure (whose code is not shown here) updates the (anticipated) ship_date of the new order to 'tomorrow', in case qty_in_stock of the corresponding book in the Books table is greater than zero. Otherwise, the stored procedure sets the ship_date to two weeks.

```
CREATE TRIGGER update_ShipDate
        AFTER INSERT ON Orders                    /* Event */
FOR EACH ROW
BEGIN CALL UpdateShipDate(new); END               /* Action */
```

Figure 6.12 Trigger to Update the Shipping Date of New Orders

6.7 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- Why is it not straightforward to integrate SQL queries with a host programming language? **(Section 6.1.1)**
- How do we declare variables in Embedded SQL? **(Section 6.1.1)**

- `setName` and `getName`: We did not use these functions in our code fragment; they allow us to name the cookie.
- `setValue` and `getValue`: These functions allow us to set and read the value of the cookie.

The cookie is added to the `request` object within the Java servlet to be sent to the client. Once a cookie is received from a site (`www.bookstore.com` in this example), the client's Web browser appends it to all HTTP requests it sends to this site, until the cookie expires.

We can access the contents of a cookie in the middle-tier code through the `request` object `getCookies()` method, which returns an array of Cookie objects. The following code fragment reads the array and looks for the cookie with name 'username.'

```
Cookie[] cookies = request.getCookies();
String theUser;
for(int i=0; i < cookies.length; i++) {
    Cookie cookie = cookies[i];
    if (cookie.getName().equals("username"))
        theUser = cookie.getValue();
}
```

A simple test can be used to check whether the user has turned off cookies: Send a cookie to the user, and then check whether the `request` object that is returned still contains the cookie. Note that a cookie should never contain an unencrypted password or other private, unencrypted data, as the user can easily inspect, modify, and erase any cookie at any time, including in the middle of a session. The application logic needs to have sufficient consistency checks to ensure that the data in the cookie is valid.

7.8 CASE STUDY: THE INTERNET BOOK SHOP

DBDudes now moves on to the implementation of the application layer and considers alternatives for connecting the DBMS to the World Wide Web.

DBDudes begins by considering session management. For example, users who log in to the site, browse the catalog, and select books to buy do not want to re-enter their customer identification numbers. Session management has to extend to the whole process of selecting books, adding them to a shopping cart, possibly removing books from the cart, and checking out and paying for the books.

DBDudes then considers whether webpages for books should be static or dynamic. If there is a static webpage for each book, then we need an extra database field in the Books relation that points to the location of the file. Even though this enables special page designs for different books, it is a very labor-intensive solution. DBDudes convinces B&N to dynamically assemble the webpage for a book from a standard template instantiated with information about the book in the Books relation. Thus, DBDudes do not use static HTML pages, such as the one shown in Figure 7.1, to display the inventory.

DBDudes considers the use of XML as a data exchange format between the database server and the middle tier, or the middle tier and the client tier. Representation of the data in XML at the middle tier as shown in Figures 7.2 and 7.3 would allow easier integration of other data sources in the future, but B&N decides that they do not anticipate a need for such integration, and so DBDudes decide not to use XML data exchange at this time.

DBDudes designs the application logic as follows. They think that there will be four different webpages:

- `index.jsp`: The home page of Barns and Nobble. This is the main entry point for the shop. This page has search text fields and buttons that allow the user to search by author name, ISBN, or title of the book. There is also a link to the page that shows the shopping cart, `cart.jsp`.
- `login.jsp`: Allows registered users to log in. Here DBDudes use an HTML form similar to the one displayed in Figure 7.11. At the middle tier, they use a code fragment similar to the piece shown in Figure 7.19 and JavaServerPages as shown in Figure 7.20.
- `search.jsp`: Lists all books in the database that match the search condition specified by the user. The user can add listed items to the shopping basket; each book has a button next to it that adds it. (If the item is already in the shopping basket, it increments the quantity by one.) There is also a counter that shows the total number of items currently in the shopping basket. (DBDudes makes a note that that a quantity of five for a single item in the shopping basket should indicate a total purchase quantity of five as well.) The `search.jsp` page also contains a button that directs the user to `cart.jsp`.
- `cart.jsp`: Lists all the books currently in the shopping basket. The listing should include all items in the shopping basket with the product name, price, a text box for the quantity (which the user can use to change quantities of items), and a button to remove the item from the shopping basket. This page has three other buttons: one button to continue shopping (which returns the user to page `index.jsp`), a second button to update the shop-

ping basket with the altered quantities from the text boxes, and a third button to place the order, which directs the user to the page `confirm.jsp`.

- `confirm.jsp`: Lists the complete order so far and allows the user to enter his or her contact information or customer ID. There are two buttons on this page: one button to cancel the order and a second button to submit the final order. The cancel button empties the shopping basket and returns the user to the home page. The submit button updates the database with the new order, empties the shopping basket, and returns the user to the home page.

DBDudes also considers the use of JavaScript at the presentation tier to check user input before it is sent to the middle tier. For example, in the page `login.jsp`, DBDudes is likely to write JavaScript code similar to that shown in Figure 7.12.

This leaves DBDudes with one final decision: how to connect applications to the DBMS. They consider the two main alternatives presented in Section 7.7: CGI scripts versus using an application server infrastructure. If they use CGI scripts, they would have to encode session management logic—not an easy task. If they use an application server, they can make use of all the functionality that the application server provides. Therefore, they recommend that B&N implement server-side processing using an application server.

B&N accepts the decision to use an application server, but decides that no code should be specific to any particular application server, since B&N does not want to lock itself into one vendor. DBDudes agrees proceeds to build the following pieces:

- DBDudes designs top level pages that allow customers to navigate the website as well as various search forms and result presentations.
- Assuming that DBDudes selects a Java-based application server, they have to write Java servlets to process form-generated requests. Potentially, they could reuse existing (possibly commercially available) JavaBeans. They can use JDBC as a database interface; examples of JDBC code can be found in Section 6.2. Instead of programming servlets, they could resort to Java Server Pages and annotate pages with special JSP markup tags.
- DBDudes select an application server that uses proprietary markup tags, but due to their arrangement with B&N, they are not allowed to use such tags in their code.

For completeness, we remark that if DBDudes and B&N had agreed to use CGI scripts, DBDudes would have had the following tasks:

- Create the top level HTML pages that allow users to navigate the site and various forms that allow users to search the catalog by ISBN, author name, or title. An example page containing a search form is shown in Figure 7.1. In addition to the input forms, DBDudes must develop appropriate presentations for the results.
- Develop the logic to track a customer session. Relevant information must be stored either at the server side or in the customer's browser using cookies.
- Write the scripts that process user requests. For example, a customer can use a form called 'Search books by title' to type in a title and search for books with that title. The CGI interface communicates with a script that processes the request. An example of such a script written in Perl using the DBI library for data access is shown in Figure 7.16.

Our discussion thus far covers only the customer interface, the part of the website that is exposed to B&N's customers. DBDudes also needs to add applications that allow the employees and the shop owner to query and access the database and to generate summary reports of business activities.

Complete files for the case study can be found on the webpage for this book.

7.9 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What are URIs and URLs? **(Section 7.2.1)**
- How does the HTTP protocol work? What is a stateless protocol? **(Section 7.2.2)**
- Explain the main concepts of HTML. Why is it used only for data presentation and not data exchange? **(Section 7.3)**
- What are some shortcomings of HTML, and how does XML address them? **(Section 7.4)**
- What are the main components of an XML document? **(Section 7.4.1)**
- Why do we have XML DTDs? What is a well-formed XML document? What is a valid XML document? Give an example of an XML document that is valid but not well-formed, and vice versa. **(Section 7.4.2)**
- What is the role of domain-specific DTDs? **(Section 7.4.3)**
- What is a three-tier architecture? What advantages does it offer over single-tier and two-tier architectures? Give a short overview of the functionality at each of the three tiers. **(Section 7.5)**

Most inclusion dependencies in practice are *key-based*, that is, involve only keys. Foreign key constraints are a good example of key-based inclusion dependencies. An ER diagram that involves ISA hierarchies (see Section 2.4.4) also leads to key-based inclusion dependencies. If all inclusion dependencies are key-based, we rarely have to worry about splitting attribute groups that participate in inclusion dependencies, since decompositions usually do not split the primary key. Note, however, that going from 3NF to BCNF always involves splitting some key (ideally not the primary key!), since the dependency guiding the split is of the form $X \rightarrow A$ where A is part of a key.

19.9 CASE STUDY: THE INTERNET SHOP

Recall from Section 3.8 that DBDudes settled on the following schema:

Books(*isbn:* `CHAR(10)`, *title:* `CHAR(8)`, *author:* `CHAR(80)`,
qty_in_stock: `INTEGER`, *price:* `REAL`, *year_published:* `INTEGER`)
Customers(*cid:* `INTEGER`, *cname:* `CHAR(80)`, *address:* `CHAR(200)`)
Orders(*ordernum:* `INTEGER`, *isbn:* `CHAR(10)`, *cid:* `INTEGER`,
cardnum: `CHAR(16)`, *qty:* `INTEGER`, *order_date:* `DATE`, *ship_date:* `DATE`)

DBDudes analyzes the set of relations for possible redundancy. The Books relation has only one key, (*isbn*), and no other functional dependencies hold over the table. Thus, Books is in BCNF. The Customers relation also has only one key, (*cid*), and no other functional depedencies hold over the table. Thus, Customers is also in BCNF.

DBDudes has already identified the pair ⟨*ordernum, isbn*⟩ as the key for the Orders table. In addition, since each order is placed by one customer on one specific date with one specific credit card number, the following three functional dependencies hold:

ordernum → *cid*, *ordernum* → *order_date*, and *ordernum* → *cardnum*

The experts at DBDudes conclude that Orders is not even in 3NF. (Can you see why?) They decide to decompose Orders into the following two relations:

Orders(*ordernum*, *cid*, *order_date*, *cardnum*, and
Orderlists(*ordernum*, *isbn*, *qty*, *ship_date*)

The resulting two relations, Orders and Orderlists, are both in BCNF, and the decomposition is lossless-join since *ordernum* is a key for (the new) Orders. The reader is invited to check that this decomposition is also dependency-preserving. For completeness, we give the SQL DDL for the Orders and Orderlists relations below:

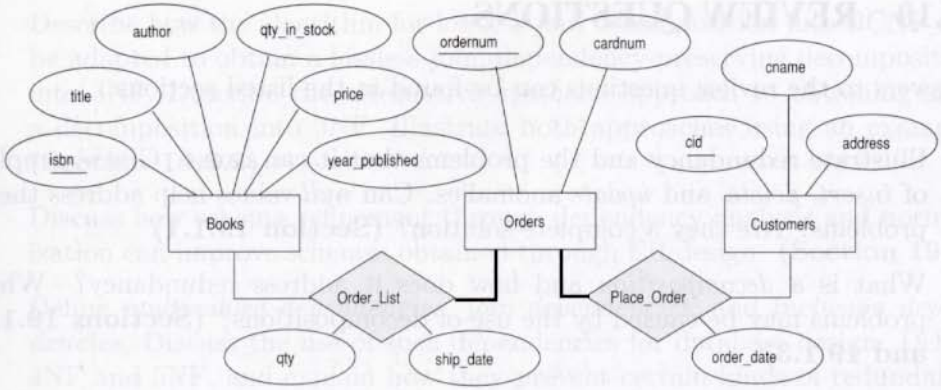

Figure 19.16 ER Diagram Reflecting the Final Design

```
CREATE TABLE Orders ( ordernum     INTEGER,
                      cid          INTEGER,
                      order_date   DATE,
                      cardnum      CHAR(16),
                      PRIMARY KEY (ordernum),
                      FOREIGN KEY (cid) REFERENCES Customers )

CREATE TABLE Orderlists ( ordernum     INTEGER,
                          isbn         CHAR(10),
                          qty          INTEGER,
                          ship_date    DATE,
                          PRIMARY KEY (ordernum, isbn),
                          FOREIGN KEY (isbn) REFERENCES Books)
```

Figure 19.16 shows an updated ER diagram that reflects the new design. Note that DBDudes could have arrived immediately at this diagram if they had made Orders an entity set instead of a relationship set right at the beginning. But at that time they did not understand the requirements completely, and it seemed natural to model Orders as a relationship set. This iterative refinement process is typical of real-life database design processes. As DBDudes has learned over time, it is rare to achieve an initial design that is not changed as a project progresses.

The DBDudes team celebrates the successful completion of logical database design and schema refinement by opening a bottle of champagne and charging it to B&N. After recovering from the celebration, they move on to the physical design phase.