Chapter 2

Database Design

Chapter Outline

What You Will Learn in This Chapter

- What is database design and why is it important?
- Why are models important in designing systems?
- How do you begin a database project?
- How do you know what data to put in the database?
- What is a class diagram (or entity-relationship diagram)?
- Is there an easier way to get started with database design?
- How are some common business associations handled in class diagrams?
- Are more complex diagrams different?
- What are the different data types?
- What are events, and how are they described in a database design?
- How are teams organized on large projects?
- How does UML split a big project into packages?
- What is an application?
- What process is followed when starting a project?

A Developer's View

Miranda: Well, Ariel, you were right as usual. A database seems like the right tool for this job.

Ariel: So you decided to take the job for your uncle's company?

Miranda: Yes, it's good money, and the company seems willing to let me learn as I go. But, it's only paying me a small amount until I finish the project.

Ariel: Great. So when do you start?

Miranda: That's the next problem. I'm not really sure where to begin.

Ariel: That could be a problem. Do you know what the application is supposed to do?

Miranda: Well, I talked to the manager and some workers, but there are a lot of points I'm not clear about. This project is bigger than I thought. I'm having trouble keeping track of all the details. There are so many reports and terms I don't know. And one salesperson started talking about all these rules about the data—things like customer numbers are five digits for corporate customers but four digits and two letters for government accounts.

Ariel: Maybe you need a system to take notes and diagram everything they tell you.

Getting Started

Begin by identifying the data that needs to be stored. Group the data into entities or classes that are defined by their attributes. It is often easiest to start with common entities such as Customers, Employees, and Sales, such as Customer(CustomerID, LastName, FirstName, Phone, …). Identify or create primary key columns. Look for one-to-many or many-to-many relationships and use key columns to specify the "many" side. Use the online DBDesign to create a diagram of the entities and relationships. Add a table and decide which attributes (columns) belong in that table. A database design is a model of the business and the tables, relationships and rules must reflect the way the business is operated.

Introduction

What is database design and why is it important? Database management systems are powerful tools but you cannot just push a button and start using them. Designing the database—specifying exactly what data will be stored—is the most important step in building the database. The table is the fundamental concept in a relational database. A table represents entities or classes of objects in the business world (Customer, Employee, Sale, Merchandise, and so on). Its columns define the properties. So the developer's main goal is to identify all of the business entities and their properties which can be turned into tables. As you will see, the actual process of creating a table in a DBMS is relatively easy. The hard part is identifying exactly what columns are needed in each table, determining the primary keys, and determining relationships among tables.

Even the process of defining business entities or classes is straightforward. If you use the DBDesign tool (highly recommended), it is easy to define a business class and add columns to it. The real challenge is that your database design has to match the business rules and assumptions. Every business has slightly different needs, goals, and assumptions. Your design should reflect these rules. Consequently, you first have to learn the individual business rules. Then you have to figure out how those rules affect the database design. In a real-world project, you will need to talk with users and managers to learn the rules. A database represents a model of the organization. The more closely your model matches the original, the easier it will be to build and use the application. This chapter shows how to build a visual model that diagrams the business entities and relationships. Chapter 3 discusses these concepts in more detail and defines specific rules that tables need to follow.

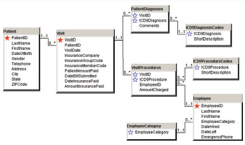
To be successful, any information system has to add value for the users. You need to identify the users and then decide exactly how an information system can help them. Along the way, you identify the data needed and the various business rules. This process requires research, interviews, and cross-checking.

Initially, the main things to look for are: 1. Business entities (Customer, Merchandise, Employee, and so on), 2. Primary keys that identify the entities (CustomerID, SKU, EmployeeID), and 3. Associations or relationships among the entities. In particular, focus on the degree of the association: Can customers place one order or many orders? Is an order assigned to one employee or can many employees be involved? The answers to these questions can depend on the specific business and they will change the overall database design.

Two-Minute Chapter

Relational databases are powerful tools to build business applications. One of the strengths is the way data is separated into tables. A table is a set of data that has certain properties, but essentially, data in a table represents a single business object (or entity). The columns of a table represent the attributes of the object, such as Name, and Phone number for a Customer. Each row is a single instance of the object. Defining the tables needed for a business application is the key goal of this chapter (and the next). A project often begins with a collection of forms and reports that the users need. You need to identify the primary objects on those forms and reports. Business objects typically including things such as Customers, Employees, Items, and Vendors. Other objects arise because of events, such as Sales and Purchases. More complex objects arise from repeating sections on forms (subforms) and to link tables together.

A critical feature of a table is that it must have a primary key—which is a column or set of columns that uniquely identify each row. In base cases, an ID value can be generated by the DBMS. For example, a CustomerID column often contains generated values for Customers to ensure the ID is unique. But, avoid using generated keys for every table. Some tables need multiple columns as part of the key. These columns represent many-to-many relationships. The common business example is SaleItems—a table that lists items that were purchased on a given sale. It contains two columns as part of the key: SaleID + ItemID, because each Sale can contain many Items being sold, and each ItemID can be sold many times. When trying to decide which columns should be part of the primary key, write them down and ask: For each of the other columns (Sale), can there be one or many of these items? If the answer is “many,” then set the new column as key.



Patient(PatientID, LastName, FirstName, DateOfBirth, ...)
Visit(VisitID, PatientID, VisitDate, InsuranceCompany, ...)
PatientDiagnoses(VisitID, ICD9Diagnosis, Comments)
VisitProcedures(VisitID, ICD9Procedure, EmployeeID, AmountCharged)
ICD9DiagnosisCodes(ICD9Diagnosis, ShortDescription)
ICD9ProcedureCodes(ICD9Procedure, ShortDescription)
Employee(EmployeeID, LastName, FirstName, EmployeeCategory, ...)
EmployeeCategory(EmployeeCategory)

User views of data.	Conceptual data model.	Implementation (relational) data model.	Physical data storage.
	Class diagram that shows business entities, relationships, and rules.	List of nicely-behaved tables. Use data normalization to derive the list.	Indexes and storage methods to improve performance.

Figure 2.1

Design models. The conceptual model records and describes the user views of the system. The implementation model describes the way the data will be stored. The final physical database may utilize storage techniques like indexing to improve performance.

Then turn the question around and ask it again: Can each Item appear on one or many Sales? If the answer is many, add that column as part of the key. Getting the correct primary key is a critical step in designing a relational database.

This chapter focuses on a visual approach to design where each table is defined to represent a single object. Primary keys are defined for each table and highlight one-to-many and many-to-many relationships.

Models

Why are models important in designing systems? Small projects that involve a few users and one or two developers are generally straightforward. However, you still must carefully design the databases so they are flexible enough to handle future needs. Likewise, you have to keep notes so that future developers can easily understand the system, its goals, and your decisions. Large projects bring additional complications. With many users and several developers, you need to split the project into smaller problems, communicate ideas between users and designers, and track the team's progress. Models and diagrams are often used to communicate information among user and developers.

An important step in all development methodologies is to build models of the system. A model is a simplified abstraction of a real-world system. In many cases the model consists of a drawing that provides a visual picture of the system. Just as contractors need blueprints to construct a building, information system developers need designs to help them create useful systems. As shown in Figure 2.1, conceptual models are based on user views of the system. Implementation models are based on the conceptual models and describe how the data will be stored. The implementation model is used by the DBMS to store the data.

Three common types of models are used to design systems: process models, class or object models, and event models. Process models are displayed with a **collaboration diagram** or a data flow diagram (DFD). They are typically explained in detail in systems analysis courses and are used to redesign the flow of

1. Identify the exact goals of the system.
2. Talk with the users to identify the basic forms and reports.
3. Identify the data items to be stored.
4. Design the classes (tables) and relationships.
5. Identify any business constraints.
6. Verify the design matches the business rules.

Figure 2.2

Initial steps in database design. A database design represents the business rules of the organization. You must carefully interview the users to make sure you correctly identify all of the business rules. The process is usually iterative, as you design classes you have to return to the users to obtain more details.

information within an organization. Class diagrams or the older entity-relationship diagrams are used to show the primary entities or objects in the system. Event models such as a sequence or statechart diagram are newer and illustrate the timing of various events and show how messages are passed between various objects. Each of these models is used to illustrate a different aspect of the system being designed. A good designer should be able to create and use all three types of models. However, the class diagrams are the most important tools used in designing and building database applications.

The tools available and the models you choose will depend on the size of the project and the preferences of the organization. This book concentrates on the class diagrams needed for designing tables. You can find descriptions of the other techniques in any good systems analysis book. You can also use the online DBDesign system to help create and analyze the class diagrams used in this book.

Getting Started

How do you begin a database project? Today's DBMS tools are flashy and alluring. It is always tempting to jump right in and start building the forms and reports that users are anxious to see. However, before you can build forms and reports, you must design the database correctly. If you make a mistake in the database design it will be hard to create the forms and reports, and it will take considerable time to change everything later.

Before you try to build anything, determine exactly what data will be needed by talking with the users. Occasionally, the users know exactly what they want. Most times, users have only a rough idea of what they want and a vague perception of what the computer is capable of producing. Communicating with users is a critical step in any development project. The most important aspect is to identify (1) exactly what data to collect, (2) how the various pieces of data are related, and (3) how long each item needs to be stored in the database. Figure 2.2 outlines the initial steps in the design process.

Once you have identified the data elements, you need to organize them properly. The goal is to define classes and their attributes. For example, a Customer is defined in terms of a CustomerID, LastName, FirstName, Phone number and so on. Classes are related to other classes. For example, a Customer participates in a Sale. These relationships also define business rules. For instance, in most cases, a Sale can have only one Customer but a Customer can be involved with many Sales. These business rules ultimately affect the database design. In the example,

if more than one customer can participate in a sale, the database design will be different. Hence, the entire point of database design is to identify and formalize the business rules.

To build business applications, you must understand the business details. The task is difficult, but not impossible and almost always interesting. Although every business is different, many common problems exist in the business world. Several of these problems are presented throughout this book. The patterns you develop in these exercises can be applied and extended to many common business problems.

Designing Databases

How do you know what data to put in the database? A database system has to reflect the rules and practices of the organization. You need to talk with users and examine the business practices to identify the rules. And, you need a way to record these rules so you can verify them and share them with other developers. System designs are models that are used to facilitate this communication and teamwork. Designs are a simplification or picture of the underlying business operations.

Identifying User Requirements

One challenging aspect of designing a system is to determine the requirements. You must thoroughly understand the business needs before you can create a useful system. A key step is to interview users and observe the operations of the firm. Although this step sounds easy, it can be difficult—especially when users disagree with each other. Even in the best circumstances, communication can be difficult. Excellent communication skills and experience are important to becoming a good designer.

As long as you collect the data and organize it carefully, the DBMS makes it easy to create and modify reports. As you talk with users, you will collect user documents, such as reports and forms. These documents provide information about the basic data and operations of the firm. You need to gather four basic pieces of information for the initial design: (1) the data that needs to be collected, (2) the data type (domain), (3) the amount of data involved, and (4) rules about the object relationships.

Business Objects

Database design focuses on identifying the data that needs to be stored. Later, queries can be created to search the data, input forms to enter new data, and reports to retrieve and display the data to match the user needs. For now, the most important step is to organize the data correctly so that the database system can handle it efficiently.

All businesses deal with entities or objects, such as customers, products, employees, and sales. From a systems perspective, an **entity** is some item in the real world that you wish to track. That entity is described by its **attributes** or **properties**. For example, a customer entity has a name, address, and phone number. In modeling terms, an entity listed with its properties is called a **class**. In a programming environment, a class can also have **methods** or functions that it can perform, and these can be listed with the class. For example, the customer class might have a method to add a new customer. Database designs seldom need to describe methods, so they are generally not listed.

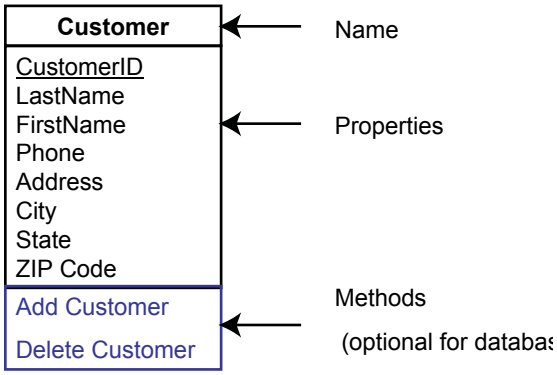


Figure 2.3

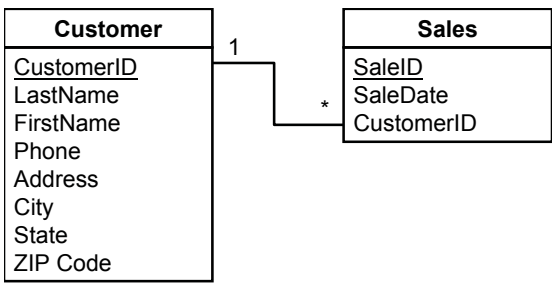
Class. A class has a name, properties, and methods. The properties describe the class and represent data to be collected. The methods are actions the class can perform, and are seldom used in a database design.

Database designers need some way to keep notes and show the list of classes to users and other designers. Several graphical techniques have been developed, but the more modern approach (and easiest to read) is the class diagram. A **class diagram** displays each class as a box containing the list of properties for the class. Class diagrams also show how the classes are related to each other by connecting them with lines. Figure 2.3 shows how a single class is displayed.

When drawing a class diagram, you often begin by identifying the major classes or entities. As you create a class, you enter the attributes that define this object. These attributes represent the data that the organization needs to store. In the Customer example, you will always need the customer name, and probably an address and phone number. Some organizations also might specify a type of customer (government, business, individual, or something else).

Figure 2.4

Relationships. The Sales tables needs CustomerID to reveal which customer participated in the sale. Putting the rest of the customer data into the Sales table would waste space and cause other problems. The relationship link to the Customer table enables the database system to find all of the related data based on just the CustomerID value.



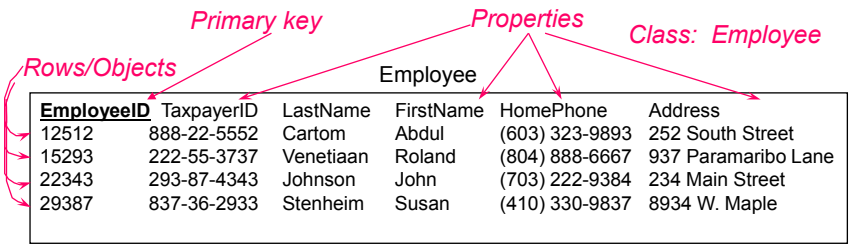


Figure 2.5

Basic database definitions. Codd has more formal terms and mathematical definitions, but these are easier to understand. One row in the data table represents a single object, a specific employee in this situation. Each column (attribute) contains one piece of data about that employee.

Tables and Relationships

Classes will eventually be stored as tables in the database system. You have to be careful what columns you include in each table. Chapter 3 describes specific rules in detail, but they also apply when you create the class diagram. One of the most important aspects is to avoid unnecessary duplication of data. Figure 2.4 shows a simple example. The Sales table needs to identify which customer participated in a sale. It accomplishes this task by storing just the primary key CustomerID in the Sales table. An alternative would be to store all of the Customer attributes in the Sales table. However, it would be a waste of space to repeat all of the customer data every time you sell something to a customer. Instead, you create a CustomerID primary key in the Customer table and place only this key value into the Sales table. The database system can then retrieve all of the related customer data from the Customer table based on the value of the CustomerID.

Notice the 1 and the * annotations in the diagram. These represent the business rules. Most companies have a policy that only one (1) customer can be listed on a sale, but a customer can participate in many (*) different sales.

Definitions

To learn how to create databases that are useful and efficient, you need to understand some basic definitions. The main ones are shown in Figure 2.5. Codd created formal mathematical definitions of these terms when he defined relational databases and these formal definitions are presented in the Appendix to Chapter 3. However, for designing and building business applications, the definitions presented here are easier to understand.

A **relational database** is a collection of carefully defined tables organized for a common purpose. A **table** is a collection of columns (attributes or properties) that describe an entity. Individual objects are stored as rows (tuples in Codd's terms) within the table. For example, EmployeeID 12512 represents one instance of an employee and is stored as one row in the Employee table. An attribute (property) is a characteristic or descriptor of an entity. Two important aspects to a relational database are that (1) all data must be stored in tables and (2) all tables must be carefully defined to provide flexibility and minimize problems. **Data normalization** is the process of defining tables properly to provide flexibility, minimize redundancy, and ensure data integrity. The goal of database design and data normal-

ization is to produce a list of nicely behaved tables. Each table describes a single type of object in the organization.

Primary Key

Every table must have a primary key. The **primary key** is a column or set of columns that identifies a particular row. For example, in the customer table you might use customer name to find a particular entry. But that column does not make a good key. What if eight customers are named John Smith? In many cases you will create new key columns to ensure they are unique. For example, a customer identification number is often created to ensure that all customers are correctly separated. The relationship between the primary key and the rest of the data is one-to-one. That is, each entry for a key points to exactly one customer row. To highlight the primary key, the names of the columns that make up the key will be underlined. The DBDesign system uses a star in front of primary key column names because it is easier to see. You can use either approach (or both) if you draw class diagrams by hand.

In some cases there will be several choices to use as a primary key. In the customer example you could choose name or phone number, or create a unique CustomerID. If you have a choice, the primary key should be the smallest set of columns needed to form a unique identifier.

Some U.S. organizations might be tempted to use Social Security numbers (SSN) as the primary key. Even if you have a need to collect the SSN, you will be better off using a separate number as a key. One reason is that a primary key must always be unique, and with the SSN you run a risk that someone might present a forged document. More important, primary keys are used and displayed in many places within a database. If you use the SSN, too many employees will have access to your customers' private information. Because SSNs are used for many financial, governmental, and health records, you should protect customer privacy by limiting employee access to these numbers. In fact, you should encrypt them to prevent unauthorized or accidental release of the data.

The most important issue with a primary key is that it can never point to more than one row or object in the database. For example, assume you are building a database for the human resource management department. The manager tells you that the company uses names of employees to identify them. You ask whether or not two employees have the same name, so the manager examines the list of employees and reports that no duplicates exist among the 30 employees. The manager also suggests that if you include the employee's middle initial, you should never have a problem identifying the employees. So far, it sounds like name might be a potential key. But wait! You really need to ask what the possible key values might be in the future. If you build a database with employee name as a primary key, you are explicitly stating that no two employees will *ever* have the same name. That assumption is almost guaranteed to cause problems in the future. It is far safer to use the database to generate a key number—your application can always provide the ability to search by name, but internally, the DBMS will not mix up two people with the same name.

Class Diagrams: Introduction

What is a class diagram (or entity-relationship diagram)? The DBMS approach focuses on the data. In many organizations data remains relatively stable. For example, companies collect the same basic data on customers today that they

Term	Definition	Pet Store Examples
Entity	Something in the real world that you wish to describe or track.	Customer, Merchandise, Sales
Class	Description of an entity that includes its attributes (properties) and behavior (methods).	Customer, Merchandise, Sale
Object	One instance of a class with specific data.	Joe Jones, Premium Cat Food, Sale #32
Property	A characteristic or descriptor of a class or entity.	LastName, Description, SaleDate
Method	A function that is performed by the class.	AddCustomer, UpdateInventory, ComputeTotal
Association	A relationship between two or more classes.	Each sale can have only one customer

Figure 2.6

Basic definitions. These terms describe the main concepts needed to create a class diagram. The first step is to identify the business entities and their properties. Methods are less important than properties in a database context, but you should identify important functions or calculations.

collected 20 or 30 years ago. Basic items such as name, address, and phone number are always needed. Although you might choose to collect additional data today (cell phone number and e-mail address for example), you still utilize the same base data. On the other hand, the way companies accept and process sales orders has changed over time, so forms and reports are constantly being modified. The database approach takes advantage of this difference by focusing on defining the data correctly. Then the DBMS makes it easy to change reports and forms. The first step in any design is to identify the things or entities that you wish to observe and track.

Classes and Entities

Figure 2.6 shows some examples of the entities and relationships that will exist in the Pet Store database. Note that these definitions are informal. Each entry has a more formal definition in terms of Codd's relational model and precise semantic definitions in the **Unified Modeling Language (UML)**. However, you can develop a database without learning the mathematical foundations.

A tricky problem with database design is that your specific solution depends on the underlying assumptions and business rules. The design process becomes easier as you learn the common business rules. But, any business can have different rules, so you always have to verify the assumptions. For example, consider an employee. The employee is clearly a separate entity because you always need to keep detailed data about the employee (date hired, name, address, and so on). But what about the employee's spouse? Is the spouse an attribute of the Employee entity, or should he or she be treated as a separate entity? If the organization only cares about the spouse's name, it can be stored as an attribute of the Employee

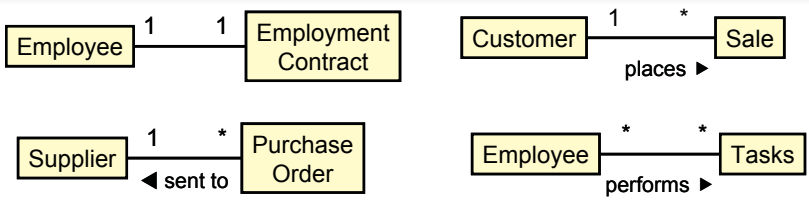


Figure 2.7

Associations. Three types of relationships (one-to-one, one-to-many, and many-to-many) occur among entities. They can be drawn in different ways, but they represent business or organizational rules. Avoid vague definitions where almost any relationship could be classified as many-to-many. They make the database design more complex.

entity. On the other hand, if the organization wants to keep additional information about the spouse (e.g., birthday, occupation, or health records), it might be better to create a separate Spouse entity with its own attributes. Your first step in designing a database is to identify the entities and their defining attributes. The second step is to specify the relationships among these entities.

Associations and Relationships

An important step in designing databases is identifying associations or relationships among entities. Details about these relationships represent the business rules. **Associations** or **relationships** represent business rules. For example, it is clear that a customer can place many orders. But the relationship is not as clear from the other direction. How many customers can be involved with one particular order? Many businesses would say that each order could come from only one customer. Hence there would be a one-to-many relationship between customers and orders. On the other hand, some organizations (such as home sales) might have multiple customers on one order, which creates a many-to-many relationship.

Associations can be named: UML refers to the **association role**. Each end of a binary association may be labeled. It is often useful to include a direction arrow to indicate how the label should be read. Figure 2.7 shows how to indicate that one customer places many sales orders.

UML uses numbers and asterisks to indicate the **multiplicity** in an association. As shown in Figure 2.7, the asterisk (*) represents many. So each supplier can receive many purchase orders, but each purchase order goes to only one supplier. Some older entity-relationship design methods used multiple arrowheads or the letters M and N to represent the "many" sides of a relationship. Correctly identifying relationships is important in properly designing a database application.

Class Diagram Details

A class diagram is a visual model of the classes and associations in an organization. These diagrams have many options, but the basic features that must be included are the class names (entities) in boxes and the associations (relationships) connecting them. Typically, you will want to include more information about the classes and associations. For example, you will eventually include the properties of the classes within the box.

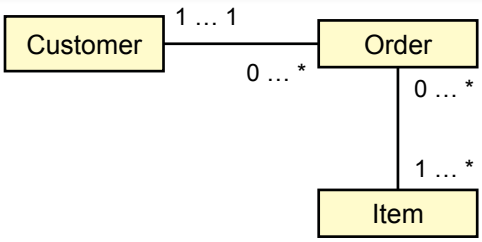


Figure 2.8

Class diagram or entity-relationship diagram. Each customer can place zero or many orders. Each sale must come from at least one and no more than one customer. The zero (0) represents an optional item, so a customer might not have placed any orders yet.

Associations also have several options. One of the most important database design issues is the multiplicity of the relationship, which has two aspects: (1) the maximum number of objects that can be related, and (2) the minimum number of objects, if any, that must be included. As indicated in Figure 2.8, multiplicity is shown as a number for the minimum value, ellipses (…), and the maximum value. An asterisk (*) represents an unknown quantity of "many." In the example in Figure 2.8, exactly one customer (1…1) can be involved with any sale.

Most of the time, a relationship requires that the referenced entity must be guaranteed to exist. For example, what happens if you have a sale form that lists a customer (CustomerID = 1123), but there is no data in the Customer table for that customer? There is a referential relationship between the sales order and the customer entity. Business rules require that customer data must already exist before that customer can make a purchase. This relationship can be denoted by specifying the minimum value of the relationship (0 if it is optional, 1 if it is required). In the Customer-Sales example, the annotation on the Customer would be 1…1 to indicate that a CustomerID value in the Sales table points to exactly one customer (no less than one and no more than one).

Be sure to read relationships in both directions. For example, in Figure 2.8, the second part of the customer/sales association states that a customer can place from zero to many sales orders. That is, a customer is not required to place an order. Some might argue that if a person has not yet placed a sale, that person should not be considered a customer. But that interpretation is getting too picky, and it would cause chicken-and-the-egg problems if you tried to enforce such a rule. Consider the Customer table to include potential customers who have signed up but not purchased anything yet.

Moving down the diagram, note the many-to-many relationship between Sale and Item (asterisks on the right side for both classes). A sale must contain at least one item (empty sales orders are not useful in business), but the firm might have an item that has not been sold yet.

Quick Start

Is there an easier way to get started with database design? In the end, details are important in designs. But often it helps to focus on the bigger picture first and fill in the details later. The sections after this one examine some common patterns

1. Identify the primary classes and data elements.
2. Create the easy classes.
3. Create generated keys if necessary.
4. Add tables to split many-to-many relationships.
5. Check primary keys.
6. Verify relationships.
7. Verify data types.

Figure 2.9

Steps to create a class diagram. Primary keys often cause problems. Look for many-to-many relationships and split them into new tables.

that arise in designing business databases, but it is easy to get lost in the details. Eventually, you need to understand those details, but first you need to start thinking in terms of a few basic concepts. The purpose of this section is to show you some ways to begin learning database design.

Creating a Class Diagram

This section summarizes how you begin a class diagram and highlights the issue of primary keys. Figure 2.9 outlines the major steps. The real trick is to start with the easy classes. Look for the base entities that do not depend on other classes. For instance, most business applications have relatively simple classes for Customers, Employees, and Items. These tables often use a generated key as the primary key column, and the data elements are usually obvious. A generated key is one that is created by the DBMS and guaranteed to be unique within that table.

Figure 2.10

Basic Sales form. Look through the form and see if you can identify the basic business objects. You should be able to easily find three objects.

Sale ID					Date
Customer First Name Last Name Address City, State ZIPCode					
ItemID	**Description**	**List Price**	**Quantity**	**QOH**	**Value**
					Total

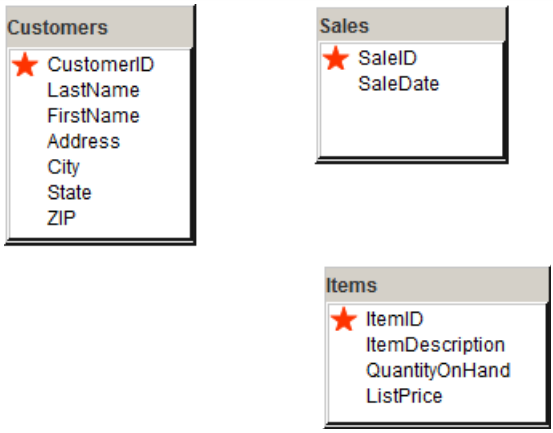


Figure 2.11

Initial objects for the Sales form: Customers, Items, and Orders. Each has a single generated column as the primary key because rows are created independently. Notice that they are not yet related to each other.

Consider the basic Sales form. Figure 2.10 shows a simplified version. Look over the form and see if you can identify the main objects it contains. Think about standard business sales for a second and you should be able to identify at least three common objects contained on the form: Customers, Items, and Sales. Data for the Customer object seems to be displayed in its own section of the form so that is a useful indicator, but people are often represented as a separate object so you should become comfortable with creating that table. The Items are a little trickier to see, but think about the business transaction and you can see the importance of treating merchandise items as a separate object. The Sales object is slightly trickier because it records an event. But in most cases, the overall form itself will become a table—because it ties together all of the other objects. In this case, Sales represent the integration of Customers and Items.

Each of these three tables (Customers, Items, and Sales) can stand alone as a base table. Customers are clearly defined in terms of standard properties such as name and address. Items have properties such as a description and list price. Sales take place on a specified date. Every table needs a primary key, but each of these three tables can benefit from using a generated key. Some companies might rely on the marketing department to create unique CustomerID and ItemID values, but database systems are good at creating unique numbers, so it is far easier to let the DBMS generate ID values as new customers, items, and sales are entered. Figure 2.11 shows these three tables in DBDesign, but you could also draw them by hand or write them with the main columns.

Notice that these three tables are not yet connected to each other. Eventually, to be able to recreate the Sales form, all of the tables must be related or connected somehow. But it is worth examining exactly how these tables might be connected. Begin by focusing on the Customers and Sales tables. Three possibilities exist: 1. Put the SaleID into the Customers table, 2. Put the CustomerID into the Sales table, or 3. Put both ID values into a third table. Before trying to find the answer, understand that each of those three possibilities could be correct—the answer depends on the specific business rules. That is, each possibility represents a different set of business rules.

CustomerID SaleID

Each customer can place many Sales (key SaleID).
Each order comes from one customer (do not key CustomerID).

*SaleID
CustomerID

Figure 2.12

Identifying primary keys. Write down the potential key columns. Ask if each of the first entity (Customer) can have one or many of the second entity (Sale). If the answer is many, key the second item. Reverse the process to see if one of the second items can be associated with one or many of the first items.

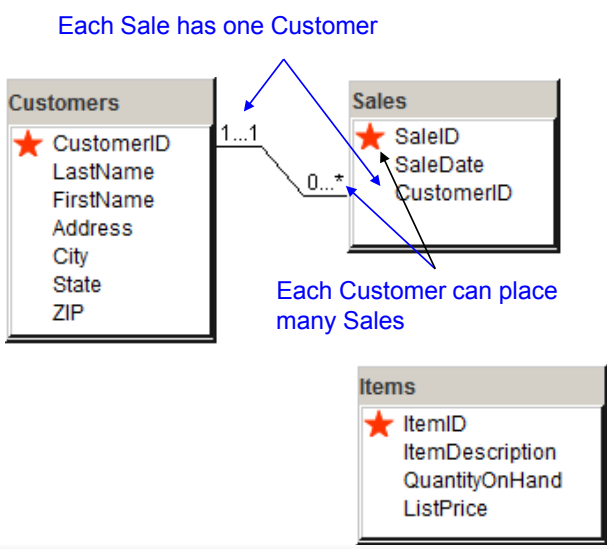
Primary Keys and Relationships

Primary keys and many-to-many relationships are often difficult for students. The trick is to remember that any column that is part of the primary key represents a "many" relationship. Consider the classic Customers-Sales relationship.

When you are not certain how to identify the keys in a table, Figure 2.12 shows a process for identifying the class relationship. Write down the columns you want to study with no key indicators. Ask yourself if each customer (first column) can place one or many Sales (second column). If the answer is many Sales, add a key indicator (underline) to the SaleID. Reverse the process and ask if a specific Sale can come from one Customer or many. The standard business rule says only one

Figure 2.13

Relationship between Customers and Sales. CustomerID belongs in the Sales table but CustomerID is not part of the key. Each Sale has one Customer—so CustomerID is not keyed in the Sales table. Each Customer can have multiple Sales. Read keyed columns as "many" and non-keyed columns as "one."



SaleID ItemID

Each Sale can have many Items (key ItemID).
Each Item can be sold many times (key SaleID).

Need a table with both SaleID and ItemID as keys

*SaleID
*ItemID

Figure 2.14

Identifying primary keys. Write down SaleID and ItemID and identify the associations. The repeating section in the original sales form shows that a Sale can list many Items. When Items are represented as SKUs (such as cans of dog food), the item can be sold many times. So a table is needed with both SaleID and ItemID as keys. Because this table does not yet exist, it must be created: SaleItems.

customer is responsible for a sale, so do not key CustomerID. The result says that SaleID is keyed but CustomerID is not. So you need to put CustomerID into a table with only SaleID as the key column. That would be the Sales table.

If you had put the SaleID into the Customers table the relationships would be reversed. With CustomerID keyed but not SaleID the design would be saying that each customer can never participate in more than one sale, and every sale could involve many customers. Read the keyed column as "many" and non-keyed columns as "one." Figure 2.13 shows the resulting design by placing CustomerID into the Sales table where it is not part of the key. From a more mechanical perspective, CustomerID could never be keyed in the Sales table. The SaleID is already a generated key so it is guaranteed to be unique. No other column would ever need to be keyed in that table. Generated key columns always stand alone.

The design is closer, but notice that the Items table is still not connected to any of the others. Thinking about the business associations, it seems that Items and Sales should be related somehow—because the Items are shown on the original Sales Form. So try the same key process. Figure 2.14 shows the process. Write down SaleID and ItemID and ask yourself if a Sale can contain one or many Items. If you are uncertain, look back at the original sales form and notice the repeating section that can list many items, so the answer is Many. Mark ItemID as a key. Working the other direction: Can an Item be sold more than once? If the items are standardized, such as cans of dog food, the answer is also "many" times. Yes, a given, specific can is sold only once, but that particular ItemID representing a brand and flavor of dog food can be sold many times. So also key SaleID.

Hence, you need a table that contains both SaleID and ItemID as a key. Look at your work so far and you will see that no such table exists. So create a new table that contains those two keys. Figure 2.15 shows the resulting table and how it ties the Sales and Items tables together. Many-to-many relationships are always handled with this third table. In DBDesign, note that a blue star is used to represent the keys in the SaleItems table. Remember that a filled red star is only used for generated keys—in the table where the values are created. So a new SaleID value will be created when a Sale is added to the Sale table, and that value will be inserted into the SaleItems table. Similarly, an existing ItemID from the Items table will be inserted into the SaleItems table to indicate which item is being sold.

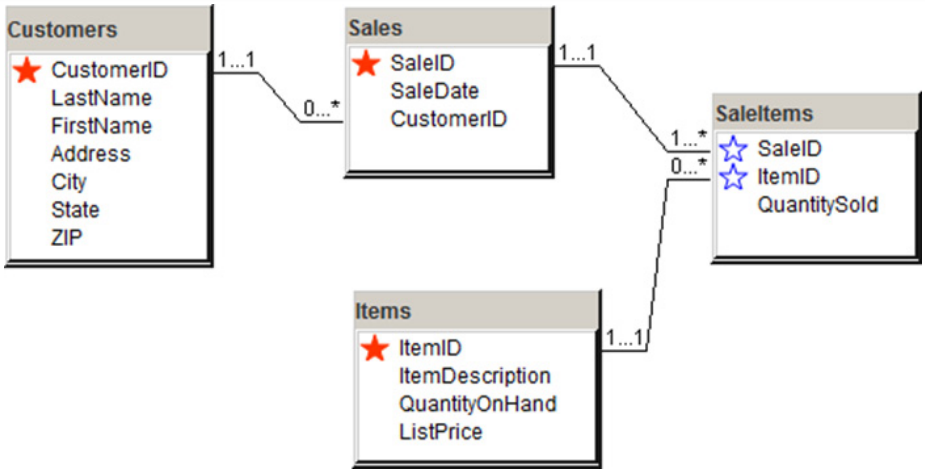


Figure 2.15

Two columns as part of the primary key. Sales and Items have a many-to-many relationship which is handled as a new table (SaleItems) with both of the columns keyed.

Think of it as a bar code scanner that reads the existing ItemID and inserts it into a new row of the SaleItems table.

DBDesign uses a special symbol to show where keys are generated to remind you that: (1) In a generating table, the generated key can be the only key column, (2) A generated key can be generated only once, and (3) You can never have a relationship that ties two generated keys together (because it would never make sense to link two randomly generated numbers).

Getting the primary keys right is critical at this stage of the design. In many ways, the keys identify the objects and tables. In the example, the generated keys CustomerID, ItemID, and SaleID uniquely identify each related object. The composite key: SaleID+ItemID identifies the many-to-many relationship between Sales and Items. From this point, you then assign the other columns as properties of the correct objects. For instance, Last Name, First Name, and Address are attributes of Customers. ListPrice is an attribute of Items because each item has one list price, if we do not worry about changes over longer periods of time. And QuantitySold is an attribute of the SaleItems because it represents the amount of a specific Item on a given Sale.

Class Diagrams: Details

How are some common business associations handled in class diagrams?

Class diagrams are useful to visualize the business entities and the underlying relationships. Many business entities can be represented by simple classes (Customers, Employees, Merchandise, Sales, and so on). However, some common business problems can lead to relatively complex relationships. Many of these situations are tied to events, such as sales or as-

Note: It is possible to temporarily skip this section and return to it once the student is more familiar with the basic design issues.

sembly. A couple of tricky concepts evolved from object-oriented design require special handling in class diagrams, and are trickier to handle within relational databases. Two classic situations are composition (objects built from other objects), and inheritance (objects defined as extensions of parent objects). This section examines how to diagram these relatively complex topics.

Association Details: N-ary Associations

Many-to-many associations between classes cause problems in the database design. They are acceptable in an initial diagram such as Figure 2.16, but they will eventually have to be split into one-to-many relationships. This process is explained in detailed in Chapter 3.

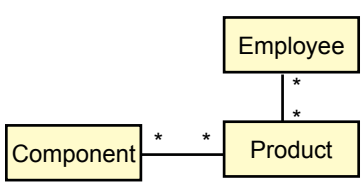
In a related situation, as shown in Figure 2.16, entities are not always obvious. Consider a basic manufacturing situation in which employees assemble components into final products. At first glance, it is tempting to say that there are three entities: employees, components, and products. This design specifies that the database should keep track of which employees worked on each product and which components go into each product. Notice that two many-to-many relationships exist.

To understand the problem caused by the many-to-many relationships, consider what happens if the company wants to know which employees assembled each component into a product. To handle this situation, Figure 2.17 shows that the three main entities (Employee, Product, and Component) are actually related to each other through an Assembly association. When more than two classes are related, the relationship is called an **n-ary association** and is drawn as a diamond. This association (actually any association) can be described by its own class data. In this example an entry in the assembly list would contain an EmployeeID, a ComponentID, and a ProductID. In total, many employees can work on many products, and many components can be installed in many products. Each individual event is captured by the Assembly association class. The Assembly association solves the many-to-many problem, because a given row in the Assembly class holds data for one employee, one component, and one product. Ultimately, you would also include a Date/Time column to record when each event occurred.

According to the UML standard, multiplicity has little meaning in the n-ary context. The multiplicity number placed on a class represents the potential number of objects in the association when the other n-1 values are fixed. For example, if ComponentID and EmployeeID are fixed, how many products could there be? In other words, can an employee install the same component in more than one

Figure 2.16

Many-to-many relationships cause problems for databases. In this example, many employees can install many components on many products, but we do not know which components the employee actually installed.



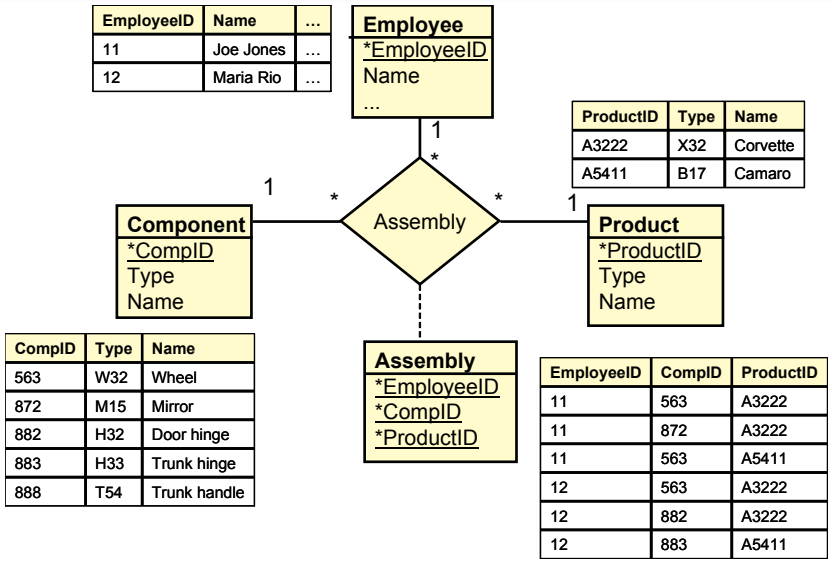


Figure 2.17

Many-to-many associations are converted to a set of one-to-many relationships with an n-ary association, which includes a new class. In this example each row in the Assembly class holds data for one employee, one component, and one product. Notice that the Assembly class (box) is connected to the Assembly association (diamond) by a dashed line.

product? In most situations the answer will be yes, so the multiplicity will generally be a "many" asterisk.

Eventually to create a database, all many-to-many relationships must be converted to a set of one-to-many relationships by adding a new entity. Like the Assembly entity, this new entity usually represents an activity and often includes a date/time stamp.

As a designer you will use class diagrams for different purposes. Sometimes you need to see the detail; other times you only care about the big picture. For large projects, it sometimes helps to create an overview diagram that displays the primary relationships between the main classes. On this diagram it is acceptable to use many-to-many relationships to hide some detail entities.

Association Details: Aggregation

Some special types of associations arise often enough that UML has defined special techniques for handling them. One category is known as an **aggregation** or a collection. For example, a Sale consists of a collection of Items being purchased. As shown in Figure 2.18, aggregation is indicated by a small diamond on the association line next to the class that is the aggregate. In the example, the diamond is next to the Sale class. Associations with a many side can be ordered or unordered. In this example, the sequence in which the Items are stored does not matter. If order did matter, you would simply put the notation {ordered} underneath the association. Be sure to include the braces around the word. Aggregations are rarely marked separately in a database design.

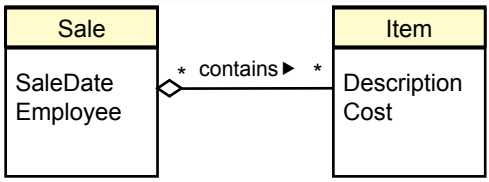


Figure 2.18

Association aggregation. A Sale contains a list of items being purchased. A small diamond is placed on the association to remind us of this special relationship.

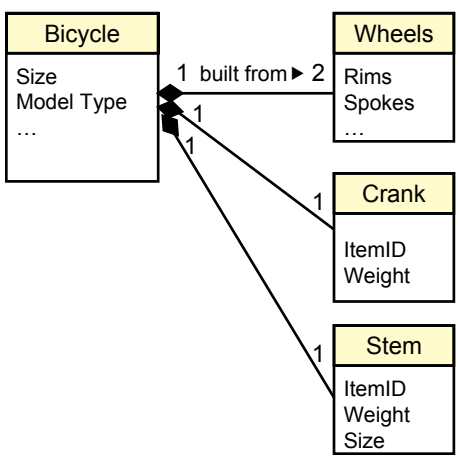
Association Details: Composition

The simple aggregation indicator is not used much in business settings. However, **composition** is a stronger aggregate association that does arise more often. In a composition, the individual items become the new object. Consider a bicycle, which is built from a set of components (wheels, crank, stem, and so on). UML provides two methods to display composition. In Figure 2.19 the individual classes are separated and marked with a filled diamond. An alternative technique shown in Figure 2.20 is to indicate the composition by drawing the component classes inside the main Bicycle class. It is easier to recognize the relationship in the embedded diagram, but it could get messy trying to show 20 different objects required to define a bicycle. Figure 2.20 also highlights the fact that the component items could be described as properties of the main Bicycle class.

The differences between aggregation and composition are subtle. The UML standard states that a composition can exist only for a one-to-many relationship. Any many-to-many association would have to use the simple aggregation indicator. Composition relationships are generally easier to recognize than aggregation relationships, and they are particularly common in manufacturing environments.

Figure 2.19

Association composition. A bicycle is built from several individual components. These components no longer exist separately; they become the bicycle.



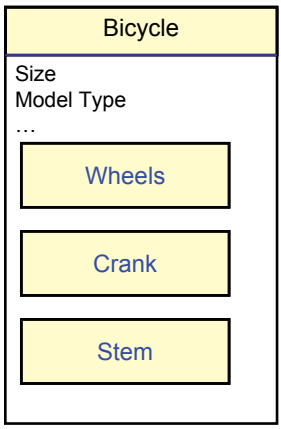


Figure 2.20

Association composition. It is easier to see the composition by embedding the component items within the main class.

Just remember that a composition exists only when the individual items become the new class. After the bicycle is built, you no longer refer to the individual components.

Association Details: Generalization

Another common association that arises in business settings is **generalization**. This situation generates a class hierarchy. The most general description is given at the top, and more specific classes are derived from it. Figure 2.21 presents a sample from Sally's Pet Store. Each animal has certain generic properties (e.g., DateBorn, Name, Gender, ListPrice), contained in the generic Animal class. But specific types of animals require slightly different information. For example, for a mammal (perhaps a cat), buyers want to know the size of the litter and whether or not the animal has claws. On the other hand, fish do not have claws, and customers want different information, such as whether they are fresh- or saltwater fish and the condition of their scales. Similar animal-specific data can be collected for each species. There can be multiple levels of generalization. In the pet store example, the Mammal category could be further split into Cat, Dog, and Other.

A small, unfilled triangle is used to indicate a generalization relationship. You can connect all of the subclasses into one triangle as in Figure 2.21, or you can draw each line separately. For the situation in this example, the collected approach is the best choice because the association represents a disjoint (mutually exclusive) set. An animal can fall into only one of the subclasses.

An important characteristic of generalization is that lower-level classes inherit the properties and methods of the classes above them. Classes often begin with fairly general descriptions. More detailed classes are **derived** from these base classes. Each lower-level class inherits the properties and functions from the higher classes. **Inheritance** means that objects in the derived classes include all of the properties from the higher classes, as well as those defined in their own class. Similarly, functions defined in the related classes are available to the new class.

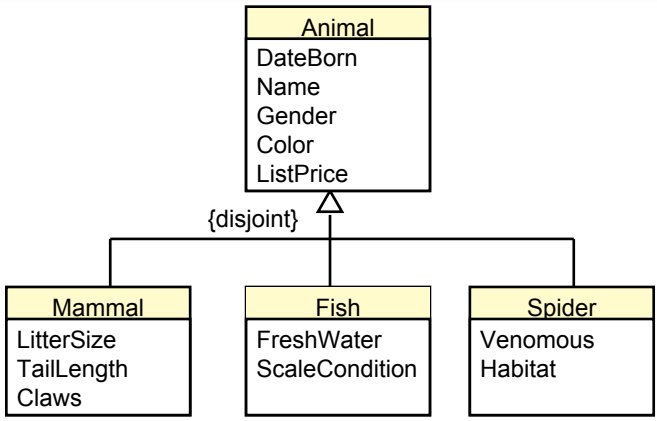


Figure 2.21

Association generalization. The generic Animal class holds data that applies to all animals. The derived subclasses contain data that is specific to each species.

Consider the example of a bank accounting system displayed in Figure 2.22. A designer would start with the basic description of a customer account. The bank is always going to need basic information about its accounts, such as AccountID, CustomerID, DateOpened, and CurrentBalance. Similarly, there will be common functions including opening and closing the account. All of these basic properties and actions will be defined in the base class for Accounts.

New accounts can be derived from these accounts, and designers would only have to add the new features—saving time and reducing errors. For example, Checking Accounts have a MinimumBalance to avoid fees, and the bank must track the number of Overdrafts each month. The Checking Accounts class is derived from the base Accounts class, and the developer adds the new properties and functions. This new class automatically inherits all of the properties and functions from the Accounts class, so you do not have to redefine them. Similarly, the bank pays interest on savings accounts, so a Savings Accounts class is created that records the current InterestRate and includes a function to compute and credit the interest due each month.

Additional classes can be derived from the Savings Accounts and Checking Accounts classes. For instance, the bank probably has special checking accounts for seniors and for students. These new accounts might offer lower fees, different minimum balance requirements, or different interest rates. To accommodate these changes, the design diagram is simply expanded by adding new classes below these initial definitions. These diagrams display the **class hierarchy** which shows how classes are derived from each other, and highlights which properties and functions are inherited. The UML uses open diamond arrowheads to indicate that the higher-level class is the more general class. In the example, the Savings Accounts and Checking Accounts classes are derived from the generic Accounts class, so the association lines point to it.

Each class in Figure 2.22 can also perform individual functions. Defining properties and methods within a class is known as **encapsulation**. It has the advantage of placing all relevant definitions in one location. Encapsulation also provides some security and control features because properties and functions can be protected from other areas of the application.

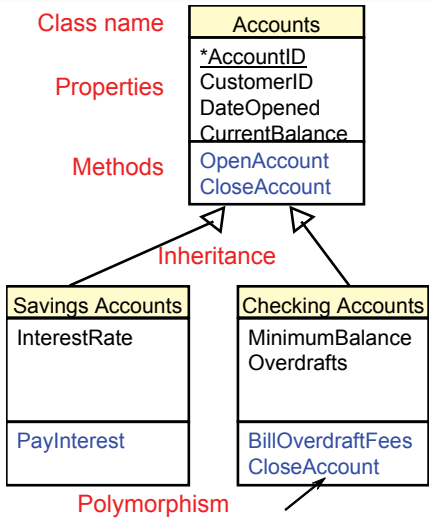


Figure 2.22

Class inheritance. Object classes begin with a base class (e.g., Accounts). Other classes are derived from the base class. They inherit the properties and methods, and add new features. In a bank, all accounts need to track basic customer data. Only checking accounts need to track overdraft fees.

Another interesting feature of encapsulation can be found by noting that the Accounts class has a function to close accounts. Look carefully, and you will see that the Checking Accounts class also has a function to close accounts (CloseAccount). When a derived class defines the same function as a parent class, it is known as **polymorphism**. When the system activates the function, it automatically identifies the object's class and executes the matching function. Designers can also specify that the derived function (CloseAccount in the Checking Accounts class) can call the related function in the base class. In the banking example, the Checking Account's CloseAccount function would cancel outstanding checks, compute current charges, and update the main balance. Then it would call the Accounts CloseAccount function, which would automatically archive the data and remove the object from the current records.

Polymorphism is a useful tool for application builders. It means that you can call one function regardless of the type of data. In the bank example you would simply call the CloseAccount function. Each different account could perform different actions in response to that call, but the application does not care. The complexity of the application has been moved to the design stage (where all of the classes are defined). The application builder does not have to worry about the details.

Note that in complex situations, a subclass can inherit properties and methods from more than one parent class. In Figure 2.23, a car is motorized, and it is designed for on-road use, so it inherits properties from both classes (and from the generic Vehicle class). The bicycle situation is slightly more complex because it could inherit features from the On-Road class or from the Off-Road class, depending on the type of bicycle. If you need to record data about hybrid bicycles, the Bicycle class might have to inherit data from both the On-Road and Off-Road classes.

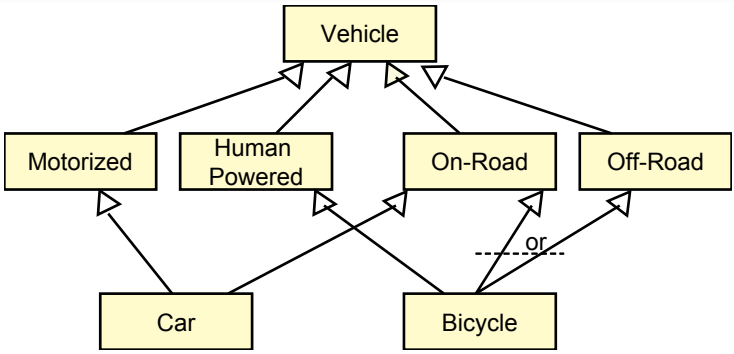


Figure 2.23

Multiple parent classes. Classes can inherit properties from several parent classes. The key is to draw the structure so that users can understand it and make sure that it matches the business rules.

Association Details: Reflexive Association

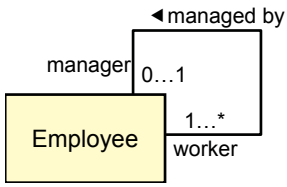
A reflexive relationship is another situation that arises in business that requires special handling. A **reflexive association** is a relationship from one class back to itself. The most common business situation is shown in Figure 2.24. most employees (worker) have a manager. Hence there is an association from Employee (the worker) back to Employee (the manager). Notice how UML enables you to label both ends of the relationship (manager and worker). Also, the "◀managed by" label indicates how the association should be read. The labels and the text clarify the purpose of the association. Associations may not need to be labeled, but reflexive relationships should generally be explained so other developers understand the purpose.

Sally's Pet Store Class Diagram

Are more complex diagrams different? It takes time to learn how to design databases. It is helpful to study other examples. Remember that Sally, the owner of the pet store, wants to create the application in sections. The first section will track the basic transaction data of the store. Hence you need to identify the primary entities involved in operating a pet store.

Figure 2.24

Reflexive relationship. A manager is an employee who manages other workers. Notice how the labels explain the purpose of the relationship.



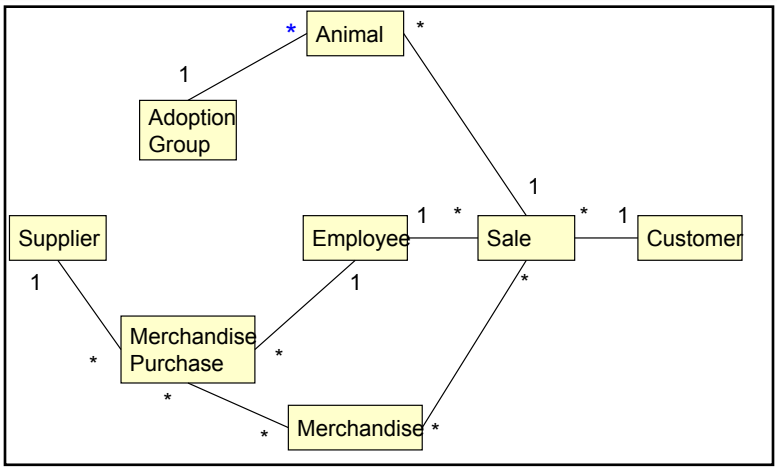


Figure 2.25

Initial class diagram for the PetStore. Animal purchases and sales are tracked separately from merchandise because the store needs to monitor different data for the two entities.

The first step in designing the pet store database application is to talk with the owner (Sally), examine other stores, and identify the primary components that will be needed. After talking with Sally, it becomes clear that the Pet Store has some features that make it different from other retail stores. The most important difference is that the store must track two separate types of sales: animals are handled differently from products. For example, the store tracks more detailed information on each animal. Also, products can be sold in multiple units (e.g., six cans of dog food), but animals must be tracked individually. Figure 2.25 shows an initial class diagram for Sally's Pet Store that is based on these primary entities. The diagram highlights the two separate tracks for animals and merchandise. Note that animals are also adopted instead of sold. Because each animal is unique and is adopted only once, the transfer of the animal is handled differently than the sale of merchandise.

While talking with Sally, a good designer will write down some of the basic items that will be involved in the database. This list consists of entities for which you need to collect data. For example, for the Pet Store database you will clearly need to collect data on customers, suppliers, animals, and products. Likewise, you will need to record each purchase and each sale. Right from the beginning, you will want to identify various attributes or characteristics of these entities. For instance, customers have names, addresses, and phone numbers. For each animal, you will want to know the type of animal (cat, dog, etc.), the breed, the date of birth, and so on.

The detailed class diagram will include the attributes for each of the entities. Notice that the initial diagram in Figure 2.25 includes several many-to-many relationships. All of these require the addition of an intermediate class. Consider the MerchandiseOrder class. Several items can be ordered at one time, so you will create a new entity (OrderItem) that contains a list of items placed on each MerchandiseOrder. The AnimalOrder and Sale entities will gain similar classes.

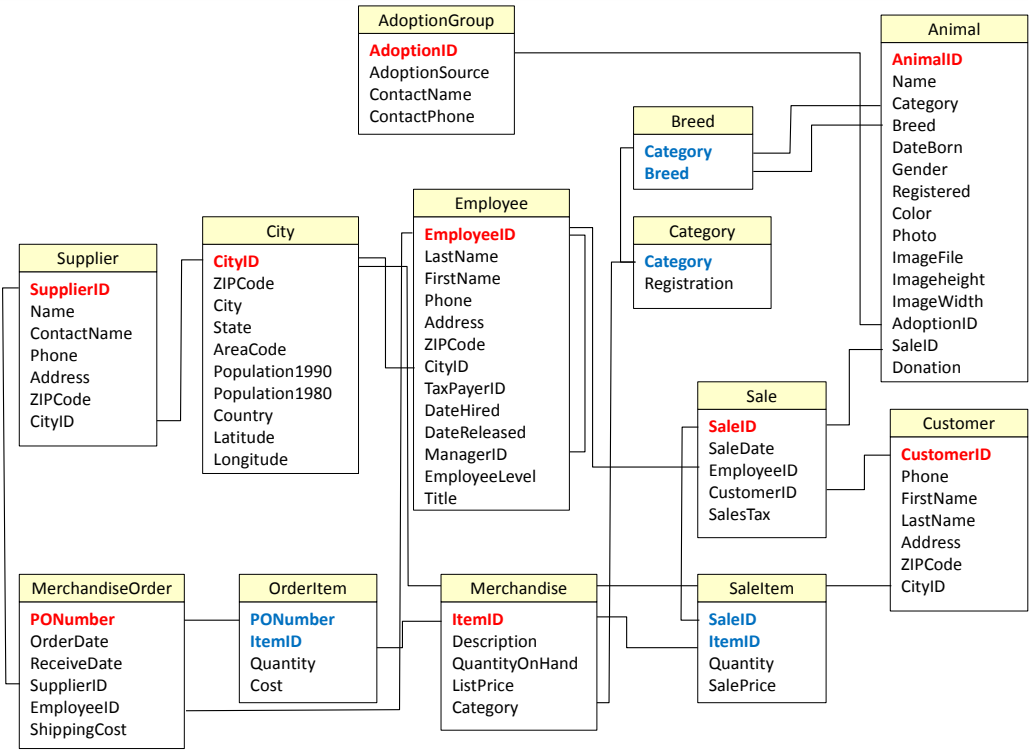


Figure 2.26

Detailed class diagram for the pet store. Notice the tables added to solve many-to-many problems: OrderItem, AnimalOrderItem, SaleItem, and SaleAnimal. The City table was added to reduce data entry. The Breed and Category tables were added to ensure data consistency. Users select the category and breed from these tables, instead of entering text or abbreviations that might be different every time.

Figure 2.26 shows the more detailed class diagram for the Pet Store with these new intermediate classes. It also contains new classes for City, Breed, and Category. Postal codes and cities raise issues in almost every business database. There is a relationship between cities and postal codes, but it is not one-to-one. One simple solution is to store the city, state, and postal code for every single customer and supplier. However, for local customers, it is highly repetitive to enter the name of the city and state for every sale. Clerks end up abbreviating the city entry and every abbreviation is different, making it impossible to analyze sales by city. A solution is to store city and postal code data in a separate class as a lookup table. Commonly used values can be entered initially. An employee can select the desired city from the existing list without having to reenter the data.

The Breed and Category classes are used to ensure consistency in the data. One of the annoying problems of text data is that people rarely enter data consistently. For example, some clerks might abbreviate the Dalmatian dog breed as Dal, others might use Dalma, and a few might enter the entire name. To solve this problem, you want to store all category and breed names one time in separate

classes. Then employees simply choose the category and breed from the list in these classes. Hence data is pulled from these lookup-tables and entered exactly the same way every time.

Both the overview and the detail class diagrams for the Pet Store can be used to communicate with users. Through the entities and relationships, the diagram displays the business rules of the firm. For instance, the separate treatment of animals and merchandise is important to the owner. Similarly, capturing only one customer per each sale is an important business rule. This rule should be confirmed by Sally. If a family adopts an animal, does she want to keep track of each member of the family? If so, you would need to add a Family class that lists the family members for each customer. The main point is that you can use the diagrams to display the new system, verify assumptions, and get new ideas.

Data Types (Domains)

What are the different data types? As you list the properties within each class, you should think about the type of data they will hold. Each attribute holds a specific **data type** or data domain. For example, what is an EmployeeID? Is it numeric? At what value does it start? How should it be incremented? Does it contain letters or other alphanumeric characters? You must identify the domain of each attribute or column. Figure 2.27 identifies several common domains. The most common is text, which holds any characters.

Note that any of the domains can also hold missing data. Users do not always know the value of some item, so it may not be entered. Missing data is defined as a **null** value.

Text

Text columns generally have a limited number of characters. SQL Server and Oracle both cut the limit in half for Unicode (2-byte) characters. Microsoft Access is the most limited at 255 characters. Some database management systems ask you to distinguish between fixed-length and variable-length text. Fixed-length strings always take up the amount of space you allocate and are most useful to improve speed in handling short strings like identification numbers or two-letter state abbreviations. Variable-length strings are stored so they take only as much space as needed for each row of data.

Memo or long-text columns are also used to hold large variable-length text data. The difference from variable-length text is that the database can allocate more space for as it is needed. The exact limit depends on the DBMS and the computer used, but long text can often include tens of thousands or millions of characters in one database column. Long text columns are often used for long comments or even short reports. However, some systems limit the operations that you can perform with these columns, such as not allowing you to sort the column data or apply pattern-matching searches.

Numbers

Numeric data is also common, and computers recognize several variations of numeric data. The most important decision you have to make about numeric data columns is choosing between integer and floating-point numbers. Integers cannot hold fractions (values to the right of a decimal point). Integers are often used for counting and include values such as 1; 2; 100; and 5,000. Floating-point numbers can include fractional values and include numbers like 3.14159 and 2.718.

Generic	Access	SQL Server	Oracle
Text			
fixed	NA	char	CHAR
variable	Short Text	varchar	VARCHAR2
Unicode	Short Text	nchar, nvarchar	NVARCHAR2
Long text	Long Text	nvarchar(max)	LONG
XML	NA	XML	XMLType
Number			
Byte (8 bits)	Byte	tinyint	INTEGER
Integer (16 bits)	Integer	smallint	INTEGER
Long (32 bits)	Long	int	INTEGER
(64 bits)	NA	bigint	NUMBER(127,0)
Fixed precision	Decimal	decimal(p,s)	NUMBER(p,s)
Float	Float	real	NUMBER, FLOAT
Double	Double	float	NUMBER
Currency	Currency	money	NUMBER(38,4)
Yes/No	Yes/No	bit	INTEGER
Date/Time	Date/Time	datetime smalldatetime	DATE
Interval	NA	interval year...	INTERVAL YEAR...
Image	OLE Object	varbinary(max)	LONG RAW, BLOB
AutoNumber	AutoNumber	Identity rowguidcol	SEQUENCES ROWID

Figure 2.27

Data types (domains). Common data types and their variations in three database systems. The text types in SQL Server and Oracle beginning with an "N" hold Unicode character sets, particularly useful for non-Latin based languages.

The first question raised with integers and floating-point numbers is, Why should you care? Why not store all numbers as floating-point values? The answer lies in the way that computers store the two types of numbers. In particular, most machines store integers in 2 (or 4) bytes of storage for every value; but they store each floating point number in 4 (or 8) bytes. Although a difference of 2 bytes might seem trivial, it can make a huge difference when multiplied by several billion rows of data. Additionally, arithmetic performed on integers is substantially faster than computations with floating-point data. Something as simple as adding two numbers together can be 10 to 100 times faster with integers than with floating-point numbers. Although machines have become faster and storage costs keep declining, performance is still an important issue when you deal with huge databases and a large customer base. If you can store a number as an integer, do it—you will get a measurable gain in performance.

Most systems also support long integers and double-precision floating-point values. In both cases the storage space is doubled compared to single-precision data. The main issue for designers involves the size of the numbers and precision that users need. For example, if you expect to have 100,000 customers, you cannot use an integer to identify and track customers (a key value). Note that only 65,536 values can be stored as 16-bit integers. To count or measure larger values, you need to use a long integer, which can range between +/- 2,000,000,000. Similarly,

Data Types	Size		
	Access	**SQL Server**	**Oracle**
Text (characters) fixed variable long text XML	 255 64 KB	 8K, 4K 8K, 4K 2 G, 1G 2 G	 2 K 4 K 2 G
Numeric Byte (8 bits) Integer (16 bits) Long (32 bits) (64 bits) Fixed precision Float Double Currency Yes/No	 255 +/- 32767 +/- 2 B NA p: 1-28 +/- 1 E 38 +/- 1 E 308 +/- 900.0000 tril. 0/1	 255 +/- 32767 +/- 2 B 18 digits p: 1-38 +/- 1 E 38 +/- 1 E 308 +/- 900.0000 tril. 0/1	 38 digits 38 digits 38 digits p: 38 digits s: -84-127, p: 1-38 38 digits 38 digits 38 digits
Date/Time	1/1/100 - 12/31/9999 (1 sec)	1/1/1753 - 12/31/9999 (3 ms) 1/1/1900 - 6/6/2079 (1 min)	1/1/-4712 - 1/31/9999 (sec)
Image	1 GB	2 GB	2 GB, 4 GB
AutoNumber	Long (4 B)	4 B or 18 digits with bigint	38 digits max.

Figure 2.28

Data sizes. Make sure that you choose a data type that can hold the largest value you will encounter. Choosing a size too large can waste space and cause slow calculations, but if in doubt, choose a larger size.

floating point numbers can support about six significant digits. Although the magnitude (exponent) can be larger, no more than six or seven digits are maintained. If users need greater precision, use double-precision values, which maintain 14 or more significant digits. Figure 2.28 lists the maximum sizes of the common data types.

Many business databases encounter a different problem. Monetary values often require a large number of digits, and users cannot tolerate round-off errors. Even if you use long integers, you would be restricted to values under 2,000,000,000 (20,000,000 if you need two decimal point values). Double-precision floating-point numbers would enable you to store numbers in the billions even with two decimal values. However, floating-point numbers are often stored with round-off errors, which might upset the accountants whose computations must be accurate to the penny. To compensate for these problems, database systems offer a currency data type, which is stored and computed as integer values (with an imputed decimal point). The arithmetic is fast, large values in the trillions can be stored, and round-off error is minimized. Most systems also offer a generic fixed-precision data type. For example, you could specify that you need 4 decimal digits of precision, and the database will store the data and perform computations with exactly 4 decimal digits.

Dates and Times

All databases need a special data type for dates and times. Most systems combine the two into one domain; some provide two separate definitions. Many beginners try to store dates as string or numeric values. Avoid this temptation. Date types have important properties. Dates (and times) are actually stored as single numbers. Dates are typically stored as integers that count the number of days or seconds from some base date. This base date may vary between systems, but it is only used internally. The value of storing dates by a count is that the system can automatically perform date arithmetic. You can easily ask for the number of days between two dates, or you can ask the system to find the date that is 30 days from today. Even if that day is in a different month or a different year, the proper date is automatically computed. Although most systems need 8 bytes to store date/time columns, doing so removes the need to worry about any year conversion problems.

A second important reason to use internal date and time representations is that the database system can convert the internal format to and from any common format. For example, in European nations, dates are generally displayed in day/month/year format, not the month/day/year format commonly used in the United States. With a common internal representation, users can choose their preferred method of entering or viewing dates. The DBMS automatically converts to the internal format, so internal dates are always consistent.

Databases also need the ability to store time intervals. Common examples include a column to hold years, months, days, minutes, or even seconds. For instance, you might want to store the length of time it takes an employee to perform a task. Without a specific interval data type, you could store it as a number. However, you would have to document the meaning of the number—it might be hours, minutes, or seconds. With a specified interval type, there is less chance for confusion.

Binary Objects

A different type of domain is a category for objects or **binary large object (BLOB)**. It enables you to store any type of object created by the computer. A useful example is to use a BLOB to hold images and files from other software packages. For example, each row in the database could hold a different spreadsheet, picture, or graph. An engineering database might hold drawings and specifications for various components. The advantage is that all of the data is stored together, making it easier for users to find the information they need and simplifying backups. Similarly, a database could hold several different revisions of a spreadsheet to show how it changed over time or to record changes by many different users.

On the other hand, BLOBs can quickly eat up space in the database. The free versions of commercial software all place limits on the size of the database files. This limit tends to be around 2 gigabytes. If your application loads thousands of BLOBs into the database, it will quickly reach this upper limit; requiring you to move up to a paid version of the DBMS. Increasing the size of the data files also complicates the backup process and might slow down other operations. So, many developers store binary files as regular operating system files and store the filename within the DBMS—which requires only a few dozen text characters for each file.

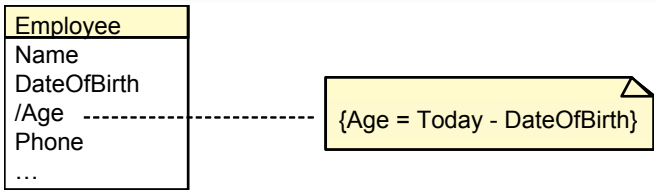


Figure 2.29

Derived values. The Age attribute does not have to be stored, since it can be computed from the date of birth. Hence, it should be noted on the class diagram. Computed attribute names are preceded with a slash.

Computed Values

Some business attributes can be computed. For instance, a sales form typically computes SalePrice times Quantity. Or an employee's age can be computed as the difference between today's date and the DateOfBirth. At the design stage, you should indicate which data attributes could be computed. The UML notation is to precede the name with a slash (/) and then describe the computation in a note. For example, the computation for a person's age is shown in Figure 2.29. The note is displayed as a box with folded corner. It is connected to the appropriate property with a dashed line.

User-Defined Types (Domains/Objects)

A relatively recent object-relational feature is supported by a few of the larger database systems. You can build your own domain as a combination of existing types. This domain essentially becomes a new object type. The example of a geocode is one of the easiest to understand. You can define a geographic location in terms of its latitude and longitude. You also might include altitude if the data is available. In a simple relational DBMS, this data is stored in separate columns. Anytime you want to use the data, you would need to look up and pass all values to your code. With a user-defined data type, you can create a new data type called *geolocation* that includes the desired components. Your column definition then has only the single data type (geolocation), but actually holds two or three pieces of data. These elements are treated by the DBMS as a single entry. Note that when you create a new domain, you also have to create functions to compare values so that you can sort and search using the new data type.

Events

What are events, and how are they described in a database design? Events are another important component of modern database systems that you need to understand and document. Many database systems enable you to write programming code within the database to take action when some event occurs. In general, three different types of events can occur in a database environment:

1. Business events that trigger some function, such as a sale triggering a reduction in inventory.
2. Data changes that signal some alert, such as an inventory that drops below a preset level, which triggers a new purchase order.
3. User interface events that trigger some action, such as a user clicking on an icon to send a purchase order to a supplier.

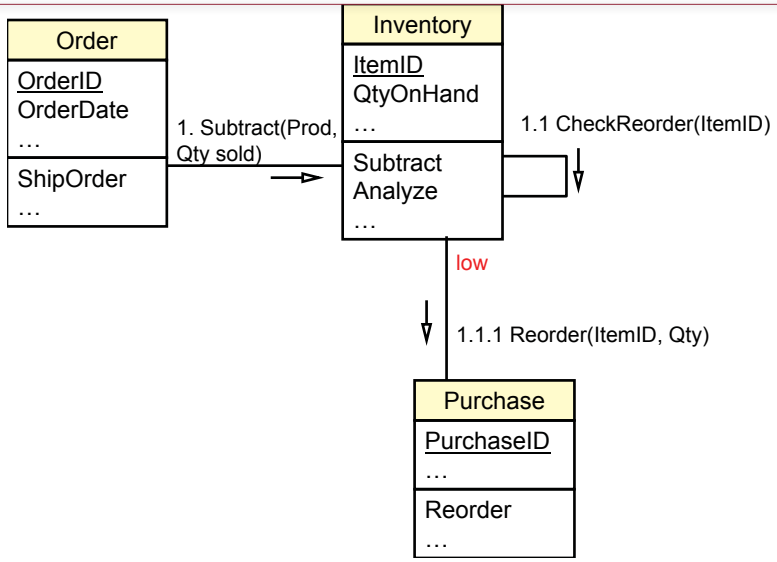


Figure 2.30

Collaboration diagram shows inventory system events. An Order shipment triggers a reduction of inventory quantity on hand which triggers an reorder-point analysis routine. If necessary, the analysis routine triggers a new purchase order for the specified item.

Events are actions that are dependent on time. UML provides several diagrams to illustrate events. The collaboration diagram is the most useful for recording events that happen within the database. Complex user interface events can be displayed on sequence diagrams or statechart diagrams. These latter diagrams are beyond the scope of this book. You can consult an OO design text for more details on how to draw them.

Database events need to be documented because (1) the code can be hard to find within the database itself, and (2) one event can trigger a chain that affects many tables and developers often need to understand the entire chain. Handling business inventory presents a useful example of the issues. Figure 2.30 is a small collaboration diagram that shows how three classes interact by exchanging messages and calling functions from other classes. Note that because the order is important, the three major trigger activities are numbered sequentially. First, when a customer places an order, this business event causes the Order class to be called to ship an order. The shipping function triggers a message to the Inventory class to subtract the appropriate quantity. When an inventory quantity changes, an automatic trigger calls a routine to analyze the current inventory levels. If the appropriate criteria are met, a purchase order is generated and the product is reordered.

The example represents a linear chain of events, which is relatively easy to understand and to test. More complex chains can be built that fork to alternatives based on various conditions and involve more complex alternatives. The UML sequence diagram can be used to show more detail on how individual messages are handled in the proper order. The UML statechart diagrams highlight how a class/object status varies over time. Details of the UML diagramming techniques are covered in other books and online tutorials. For now you should be able to draw simple collaboration diagrams that indicate the primary message events.

```
ON (QuantityOnHand < 100)
THEN Notify_Purchasing_Manager
```

Figure 2.31

Sample trigger. List the condition and the action.

In simpler situations you can keep a list of important events. You can write events as triggers, which describe the event cause and the corresponding action to be taken. For example, a business event based on inventory data could be written as shown in Figure 2.31. Large database systems such as Oracle and SQL Server support triggers directly. Microsoft Access added a few data triggers with the introduction of the 2010 version. You define the event and attach the code that will be executed when the condition arises. These triggers can be written in any basic format (e.g., pseudocode) at the design stage, and later converted to database triggers or program code. UML also provides an Object Constraint Language (OCL) that you can use to write triggers and other code fragments. It is generic and will be useful if you are using a tool that can convert the OCL code into the database you are using.

Large Projects

How are teams organized on large projects? If you build a small database system for yourself or for a single user, you will probably not take the time to draw diagrams of the entire system. However, you really should provide some documentation so the next designer who has to modify your work will know what you did. On the other hand, if you are working on large projects involving many developers and users, everyone must follow a common design methodology. What is a large project and what is a small project? There are no fixed rules, but you start to encounter problems like those listed in Figure 2.32 when several developers and many users are involved in the project.

Figure 2.32

Development issues on large projects. Large projects require more communication, adherence to standards, and project monitoring.

- Design is harder on large projects.
  - Communication with multiple users.
  - Communication between IT workers.
  - Need to divide project into pieces for teams.
  - Finding data/components.
  - Staff turnover-retraining.
- Need to monitor design process.
  - Scheduling.
  - Evaluation.
- Build systems that can be modified later.
  - Documentation.
  - Communication/underlying assumptions and model.

Computer-Aided Software Engineering
- Diagrams (linked)
- Data dictionary
- Teamwork
- Prototyping
  - Forms
  - Reports
  - Sample data
- Code generation
- Reverse engineering

Figure 2.33

CASE tool features. CASE tools help create and maintain diagrams. They also support teamwork and document control. Some can generate code from the designs or perform reverse engineering.

Methodologies for large projects begin with diagrams such as the class and collaboration diagrams described in this chapter. Then each company or team adds details. For example, standards are chosen to specify naming conventions, type of documentation required, and review procedures.

The challenge of large projects is to split the project into smaller pieces that can be handled by individual developers. Yet the pieces must fit together at the end. Project managers also need to plan the project in terms of timing and expenses. As the project develops, managers can evaluate team members in terms of the schedule.

Several types of tools can help you design database systems, and they are particularly useful for large projects. To assist in planning and scheduling, managers can use project-planning tools (e.g., Microsoft Project) that help create Gantt and PERT charts to break projects into smaller pieces and highlight the relationships among the components. Computer-assisted software engineering (CASE) tools (like IBM's Rational set) can help teams draw diagrams, enforce standards, and store all project documentation. Additionally, groupware tools (like SharePoint or Lotus Notes/Domino) help team members share their work on documents, designs, and programs. These tools annotate changes, record who made the changes and their comments, and track versions.

As summarized in Figure 2.33, CASE tools perform several useful functions for developers. In addition to assisting with graphical design, one of the most important functions of CASE tools is to maintain the data repository for the project. Every element defined by a developer is stored in the data repository, where it is shared with other developers. In other words, the data repository is a specialized database that holds all of the information related to the project's design. Some CASE tools can generate databases and applications based on the information you enter into the CASE project. In addition, reverse-engineering tools can read files from existing applications and generate the matching design elements. These CASE tools are available from many companies, including Rational Software, IBM, Oracle, and Sterling Software. CASE tools can speed the design and development process by improving communication among developers and through generating code. They offer the potential to reduce maintenance time by providing complete documentation of the system.

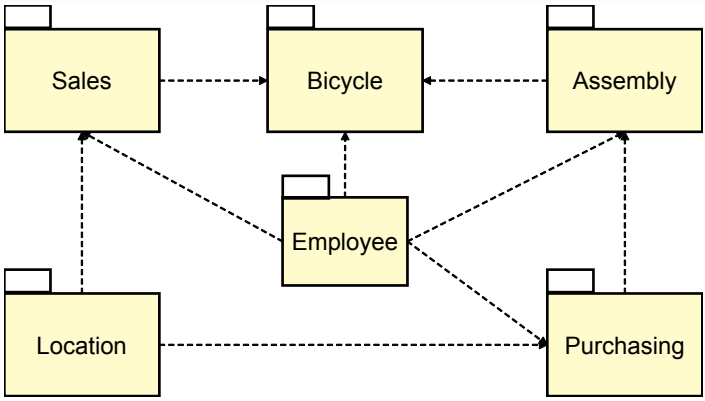


Figure 2.34

Rolling Thunder Bicycles—top-level view. The packages are loosely based on the activities of the firm. The goal is for each package to describe a self-contained collection of objects that interacts with the other packages.

Good CASE tools have existed for several years, yet many firms do not use them, and some that have tried them have failed to realize their potential advantages. Two drawbacks to CASE tools are their complexity and their cost. The cost issue can be mitigated if the tools can reduce the number of developers needed on a given project. But their complexity presents a larger hurdle. It can take a developer several months to learn to use a CASE tool effectively. Fortunately, some CASE vendors provide discounts to universities to help train students in using their tools. If you have access to a CASE tool, use it for as many assignments as possible.

Rolling Thunder Bicycles

How does UML split a big project into packages? The Rolling Thunder Bicycle case illustrates some of the common associations that arise in business settings. Because the application was designed for classroom use, many of the business assumptions were deliberately simplified. The top-level view is shown in Figure 2.34. Loosely based on the activities of the firm, the elements are grouped into six packages: Sales, Bicycles, Assembly, Employees, Purchasing, and Location. The packages will not be equal: some contain far more detail than the others. In particular, the Location and Employee packages currently contain only one or two classes. They are treated as separate packages because they both interact with several classes in multiple packages. Because they deal with independent, self-contained issues, it makes sense to separate them.

Each package contains a set of classes and associations. The Sales package is described in more detail in Figure 2.35. To minimize complexity, the associations with other packages are not displayed in this figure. For example, the Customer and RetailStore classes have an association with the Location::City class. These relationships will be shown in the Location package. Consequently, the Sales package is straightforward. Customers place orders for Bicycles. They might use a RetailStore to help them place the order, but they are not required to do so. Hence the association from the RetailStore has a (0…1) multiplicity.

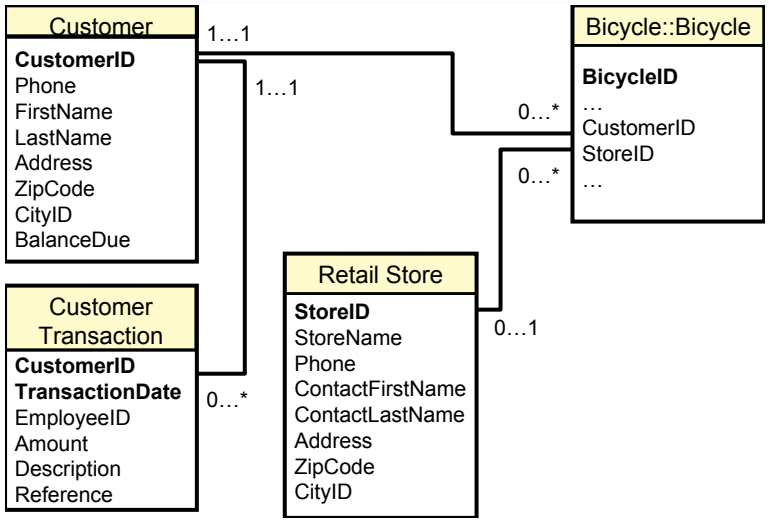
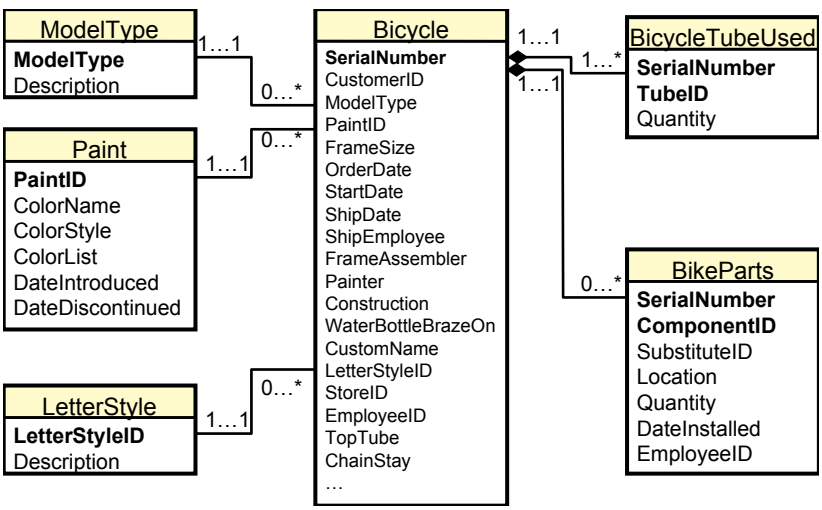


Figure 2.35

Rolling Thunder Bicycles—Sales package. Some associations with other packages are not shown here. (See the other packages.)

Figure 2.36

Rolling Thunder Bicycles—Bicycle package. Note the composition associations into the Bicycle class from the BikeTubes and BikeParts classes. To save space, only some of the Bicycle properties are displayed.



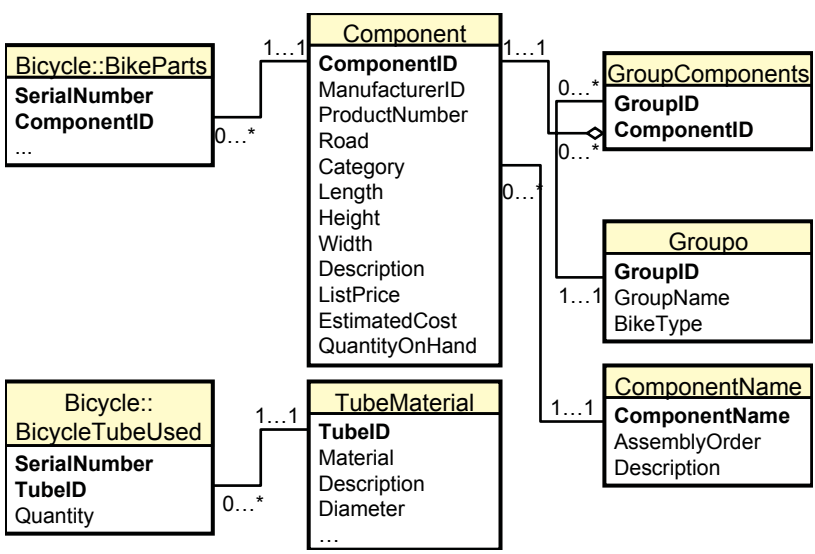
The Bicycle package contains many of the details that make this company unique. To save space, only a few of the properties of the Bicycle class are shown in Figure 2.36. Notice that a bicycle is composed of a set of tubes and a set of components. Customers can choose the type of material used to create the bicycle (aluminum, steel, carbon fiber, etc.). They can also select the components (wheels, crank, pedals, etc.) that make up the bicycle. Both of these classes have a composition association with the Bicycle class. The Bicycle class is one of the most important classes for this firm. In conjunction with the BicycleTubeUsed and BikeParts classes, it completely defines each bicycle. It also contains information about which employees worked on the bicycle. This latter decision was a design simplification choice. Another alternative would be to move the ShipEmployee, FrameAssembler, and other employee properties to a new class within the Assembly package.

As shown in Figure 2.37, the Assembly package contains more information about the various components and tube materials that make up a bicycle. In practice, the Assembly package also contains several important events. As the bicycle is assembled, data is entered that specifies who did the work and when it was finished. This data is currently stored in the Bicycle class within the Bicycle package. A collaboration diagram or a sequence diagram would have to be created to show the details of the various events within the Assembly package. For now, the classes and associations are more important, so these other diagrams are not shown here.

All component parts are purchased from other manufacturers (suppliers). The Purchase package in Figure 2.38 is a fairly traditional representation of this activity. Note that each purchase requires the use of two classes: PurchaseOrder and PurchaseItem. The PurchaseOrder is the main class that contains data about the

Figure 2.37

Rolling Thunder Bicycles—Assembly package. Several events occur during assembly, but they cannot be shown on this diagram. As the bicycle is assembled, additional data is entered into the Bicycle table within the Bicycle package.



order itself, including the date, the manufacturer, and the employee who placed the order. The PurchaseItem class contains the detail list of items that are being ordered. This class is specifically included to avoid a many-to-many association between the PurchaseOrder and Component classes.

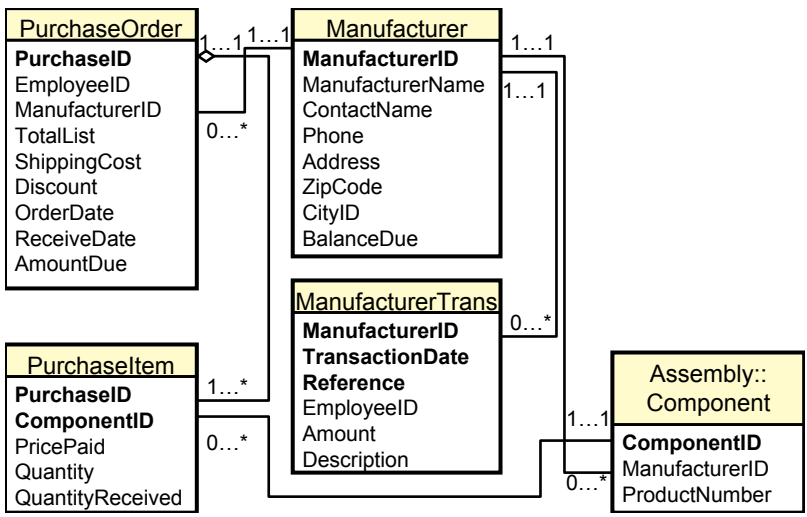
Observe from the business rules that a ManufacturerID must be included on the PurchaseOrder. It is dangerous to issue a purchase order without knowing the identity of the manufacturer. Chapter 10 explains how security controls can be imposed to provide even more safety for this crucial aspect of the business.

An additional class (ManufacturerTransactions) is used as a transaction log to record each purchase. It is also used to record payments to the manufacturers. On the purchase side, it represents a slight duplication of data (AmountDue is in both the PurchaseOrder and Transaction classes). However, it is a relatively common approach to building an accounting system. Traditional accounting methods rely on having all related transaction data in one location. In any case the class is needed to record payments to the manufacturers, so the amount of duplicated data is relatively minor.

The Location package in Figure 2.39 was created to centralize the data related to addresses and cities. Several classes have address properties. In older systems it was often easier to simply duplicate the data and store the city, state, and ZIP code in every class that referred to locations. Today, however, it is relatively easy to obtain useful information about cities and store it in a centralized table. This approach improves data entry, both in speed and data integrity. Clerks can simply choose a location from a list. Data is always entered consistently. For example, you do not have to worry about abbreviations for cities. If telephone area codes or ZIP codes are changed, you need to change them in only one table. You can also store additional information that will be useful to managers. For example, the

Figure 2.38

Rolling Thunder Bicycles—Purchasing package. Note the use of the Transaction class to store all related financial data for the manufacturers in one location.



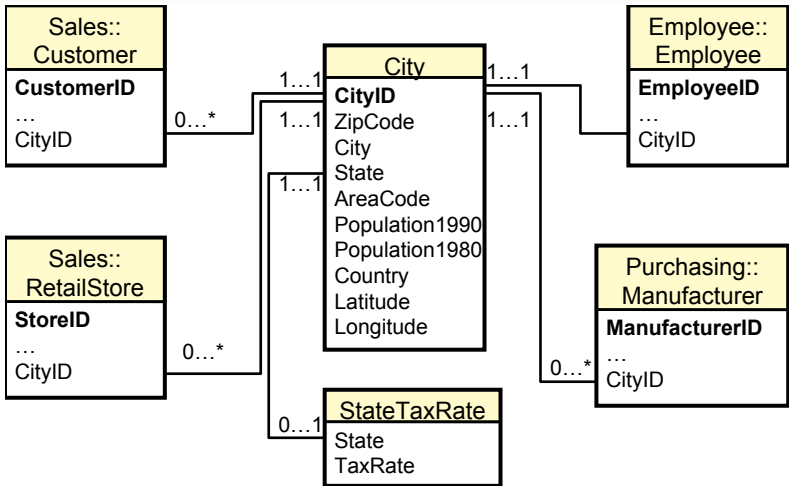


Figure 2.39

Rolling Thunder Bicycles—Location package. By centralizing the data related to cities, you speed clerical tasks and improve the quality of the data. You can also store additional information about the location that might be useful to managers.

population and geographical locations can be used to analyze sales data and direct marketing campaigns.

The Employee package is treated separately because it interacts with so many of the other packages. The Employee properties shown in Figure 2.40 are straightforward. Notice the reflexive association that denotes the management relationship. For the moment there is only one class within the Employee package. In actual practice this is where you would place the typical human resources data and associations. For instance, you would want to track employee evaluations, assignments, and promotions over time. Additional classes would generally be related to benefits such as vacation time, personal days, and insurance choices.

A detailed, combined class diagram for Rolling Thunder Bicycles is shown in Figure 2.41. Some associations are not included—partly to save space. A more important reason is that all of the drawn associations are enforced by Microsoft Access. For example, once you define the association from Employee to Bicycle, Access will only allow you to enter an EmployeeID into the Bicycle class that already exists within the Employee class. This enforcement makes sense for the person taking the order. Indeed, financial associations should be defined this strongly. On the other hand, the company may hire temporary workers for painting and frame assembly. In these cases the managers may not want to record the exact person who painted a frame, so the association from Employee to Painter in the Bicycle table is relaxed.

Application Design

What is an application? The concept of classes and attributes seems simple at first, but can quickly become complicated. Practice and experience make the process easier. For now, learn to focus on the most important objects in a given project. It is often easiest to start with one section of the problem, define the basic

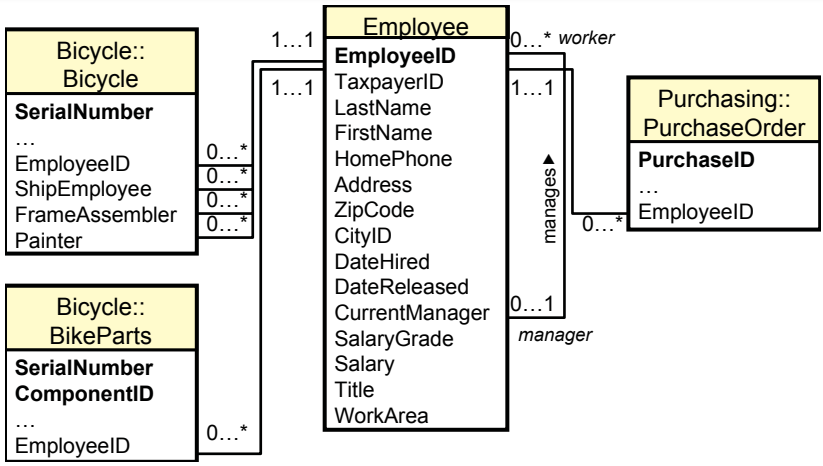


Figure 2.40

Rolling Thunder Bicycles—Employee package. Note the reflexive association to indicate managers.

elements, add detail, then expand into other sections. As you are designing the project, remember that each class becomes a table in the database, where each attribute is a column, and each row represents one specific object.

You should also begin thinking about application design in terms of the forms or screens that users will see. Consider the simple form in Figure 2.42. On paper, this form would simply have blanks for each of the items to be entered. Eventually, you could build the same form with blanks as a database form. In this case, you might think only one table is associated with this form; however, you need to think about the potential problems. With blank spaces on the form, people can enter any data they want. For example, do users really know all of the breed types? Or will they sometimes leave it blank, fill in abbreviations, or misspell words? All of these choices would cause problems with your database. Instead, it will be better to give them a selection box, where users simply pick the appropriate item from a list. But that means you will need another table that contains a list of possible breeds. It also means that you will have to establish a relationship between the Breed table and the Animal table. In turn, this relationship affects the way the application will be used. For example, someone must enter all of the predefined names into the Breed table before the Animal table can even be used.

At this point in the development, you should have talked with the users and collected any forms and reports they want. You should be able to sketch an initial class diagram that shows the main business objects and how they relate to each other, including the multiplicity of the association. You should also have a good idea about what attributes will be primary keys, or keys that you will need to create for some tables. You also need to specify the data domains of each property.

Corner Med

What process is followed when starting a project? Before you can design tables and relationships, you need to talk with the users and determine what data needs to be collected. It is easier to understand the users if you have some knowledge of

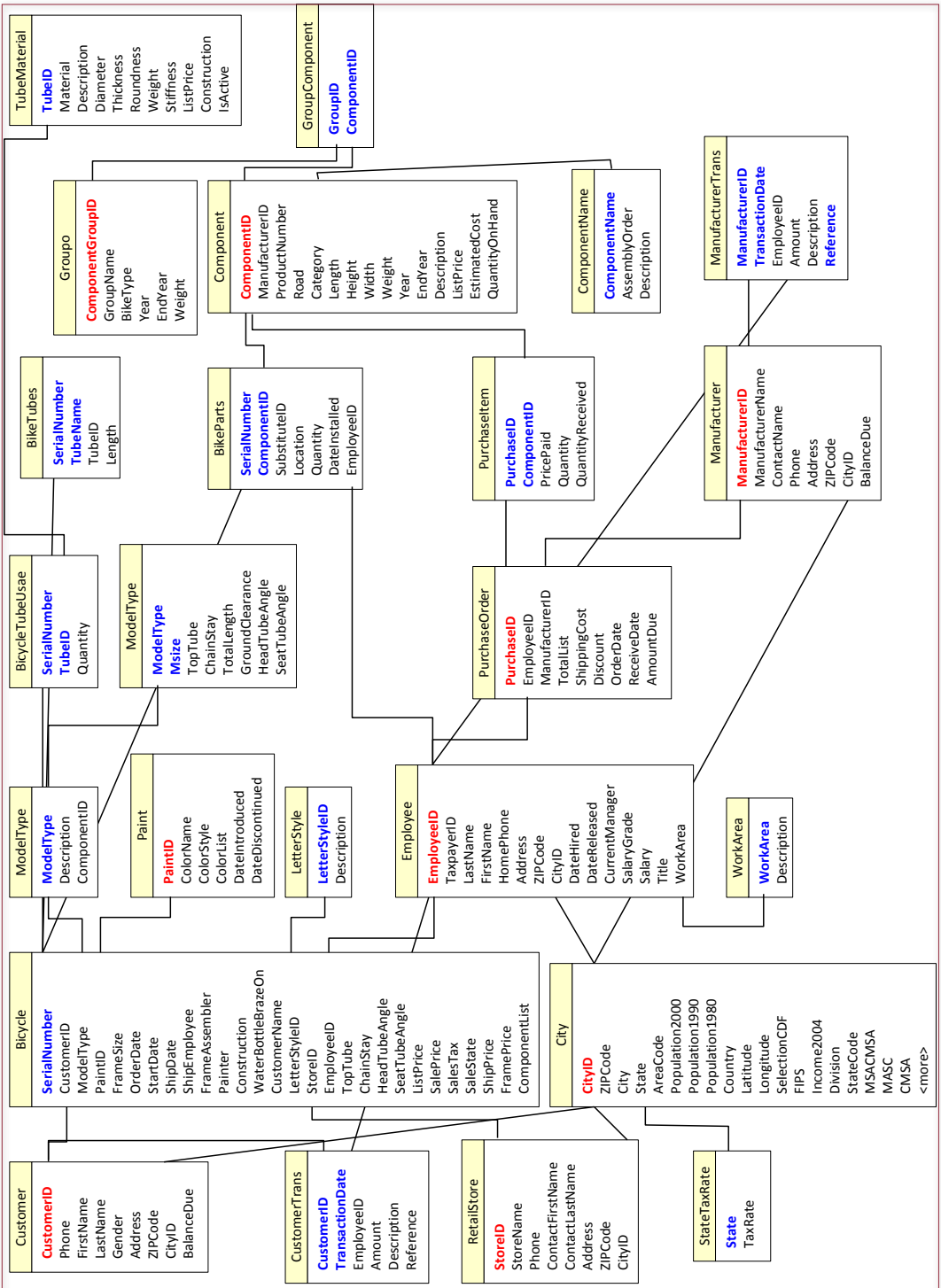


Figure 2.41

Rolling Thunder detailed class diagram. The detail class diagram is a nice reference tool for understanding the organization, but for many organizations this diagram will be too large to display at this level of detail.

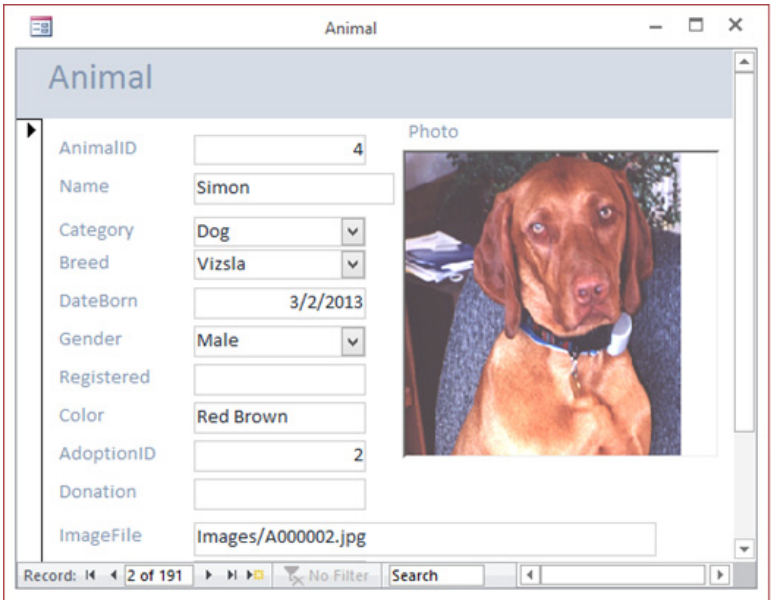


Figure 2.42

Basic Animal form. Initially this form seems to require one table (Animal). But to minimize data-entry errors, it really needs a table to hold data for Category and Breed, which can be entered via a selection box.

their field. You probably do not need a medical degree to build a business system for physicians; however, you will have to learn some of the basic terminology to understand the various data relationships. This is a good place to point out that the sample Corner Med database is merely a start of an application. None of the components should be used in an actual medical situation. It is designed purely as a demonstration project to highlight some of the issues in database design.

In a family-practice physician office, the patient visit is going to be a key element in any business administration system. Figure 2.43 shows a simple version of a form to record data about a patient visit. The first thing to note is that the main form contains two subforms. Note that each subform represents repeating data or a one-to-many relationship. A useful question to ask the managers at this point would be to confirm the insurance data. In particular, can patients have more than one insurance plan? If this data is important, the form would have to be modified to add a repeating section for insurance data. It might be tempting to argue that almost all data could potentially be repeating, so perhaps there should be dozens of repeating sections on the form. Given the state of health insurance in the U.S., it is possible that you will need to add this repeating section. However, be cautious with other items. One-to-many relationships add flexibility to collecting and storing data, but they make the data form considerably more complex. If patients rarely have more than one insurance provider, it will be cumbersome for the clerks to deal with the extra repeating section when it is rarely used. On the other hand, the patient diagnoses and treatments sections are required because most patient visits will require multiple entries. The patient visit form also illustrates one of the key steps in starting a database project: Collect input forms and reports from users so you can identify the data that needs to be stored.

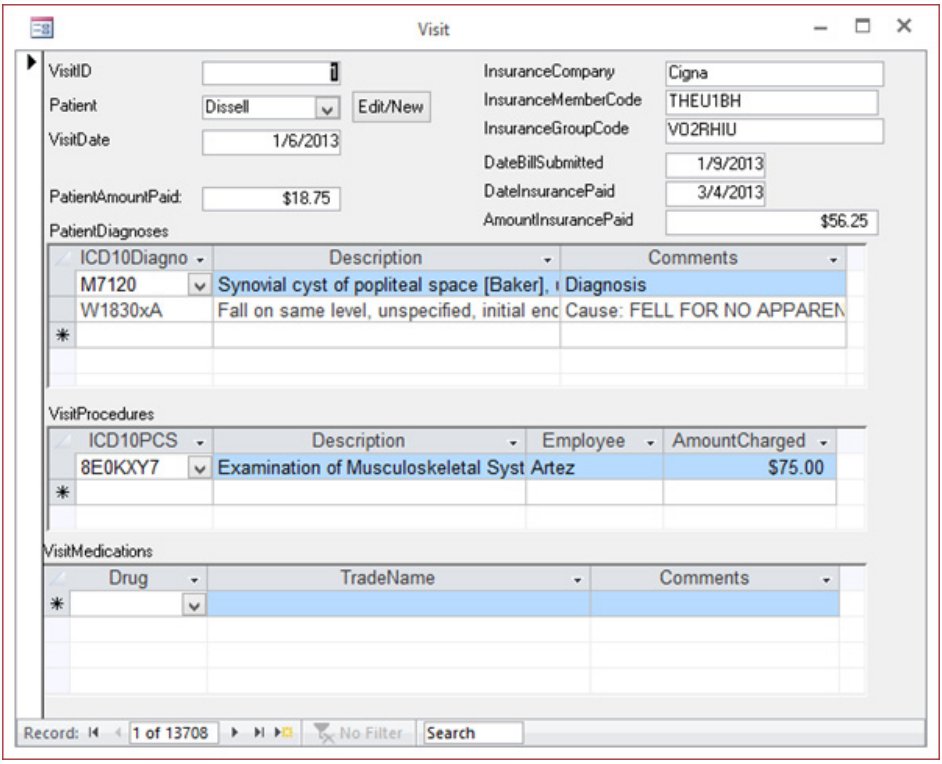


Figure 2.43

Patient Visit form. This form has two repeating sections: One for diagnoses and one for treatment. Many more details can be added but it is possible to start with these key data elements.

When you look at the patient visit form, you should start thinking about the tables that will be needed to hold the data. At the start, you should quickly identify three starting tables: (1) PatientVisit, a table that represents the form itself, (2) PatientDiagnoses, a table that arises because of the first repeating section, and (3) PatientProcedures, a table representing the second repeating section.

When you identify a new table, you should also think about the possible key columns. The PatientVisit table will most likely need a generated key—a value that the DBMS will create whenever a new visit is added to the database. Call the column VisitID. It is the best way to guarantee a unique value for every visit. A generated VisitID value also makes it easier to identify the keys for the repeating sections. Each of these tables will need two key columns. For instance, VisitID, ICD10Diagnosis will be the two key columns for the PatientDiagnoses table. It is easy to verify that both columns need to be keyed because on a specific visit, a patient could be diagnosed with many different problems, requiring ICD10Diagnosis to be part of the key. In reverse, a specific diagnosis could be applied to many different visits (either for one patient or different patients), requiring VisitID to be keyed. The same analysis reveals the two keys required for the PatientProcedures table.

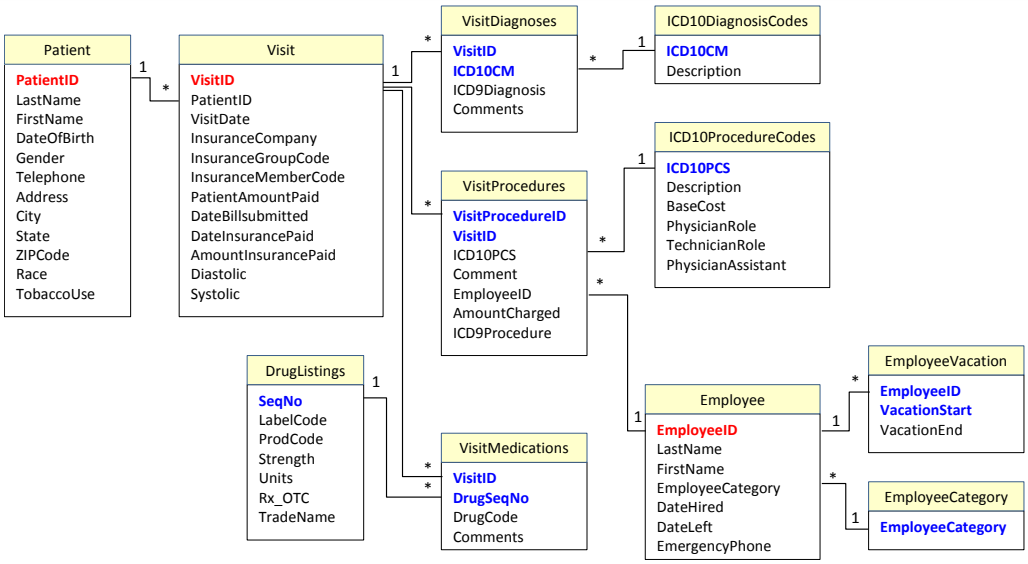


Figure 2.44

Corner Med basic tables. Ultimately, all of these tables will contain more data columns.

The next step is to ask where the ICD10Diagnosis and ICD10Procedure columns will be defined. These are slightly trickier in the context of the medical world. Ideally, you would create a table of standard codes for each of these values. The best approach would be to purchase a complete list of codes. For example, you could buy the current ICD10 codes from the United Nation's World Health Organization. Enabling physicians to pull the codes from a standard list will reduce errors. However, it would also require physicians to become familiar with the codes and to take the time to read through the list to find the specific code for every diagnosis and procedure. In practice, large healthcare institutions find it more efficient to have physicians enter written descriptions of diagnoses and procedures and hire medical coders to identify the specific codes later. This decision is an example of a complex business problem that you will have to solve early in the design process. In many cases, you will have to outline the options and present them to senior management for the final decision.

Figure 2.44 shows the basic tables used for the Corner Med case. In a real case, all of these tables will contain more data columns. However, the strength of the relational data model is that the basic structure will remain the same. It is relatively easy to add more columns to each table later. Notice that data for all employees is handled in a single class. That is, physician, nurse, and clerical data are all stored in the same table. The employees are identified by EmployeeCategory which is stored in a lookup list. However, you might want to think about this decision. The company might want to keep considerably more data for physicians. This data could be highly specialized, such as license number and date. If the amount of data gets large, it will be more efficient to store data for physicians in a table separate from the other employees. Otherwise, you will waste space and complicate the data-entry form for employees where you do not need this extra data.

By now, you should be able to make a first pass at creating a class diagram for a specific problem. You should also recognize that the final structure of the diagram depends on the business rules and assumptions. You can often resolve these questions by talking with users, but some decisions have to be passed up to senior management.

Summary

Managing projects to build useful applications and control costs is an important task. The primary steps in project management are the feasibility study, systems analysis, systems design, and implementation. Although these steps can be compressed, they cannot be skipped.

The primary objective is to design an application that provides the benefits needed by the users. System models are created to illustrate the system. These models are used to communicate with users, communicate with other developers, and help us remember the details of the system. Because defining data is a crucial step in developing a database application, the class diagram is a popular model.

The class diagram is created by identifying the primary entities in the system. Entities are defined by classes, which are identified by name and defined by the properties of each entity. Classes can also have functions that they perform.

Associations among classes are important elements of the business design because they identify the business rules. Associations are displayed as connecting lines on the class diagram. You should document the associations by providing names where appropriate, and by identifying the multiplicity of the relationship. You should be careful to identify special associations, such as aggregation, composition, generalization, and reflexive relationships.

Designers also need to identify the primary events or triggers that the application will need. There are three types of events: business events, data change events, and user events. Events can be described in terms of triggers that contain a condition and an action. Complex event chains can be shown on sequence or collaboration diagrams.

Designs generally go through several stages of revision, with each stage becoming more detailed and more accurate. A useful approach is to start with the big picture and make sure that your design identifies the primary components that will be needed in the system. Packages can be defined to group elements together to hide details. Detail items are then added in supporting diagrams for each package in the main system diagram.

Models and designs are particularly useful on large projects. The models provide a communication mechanism for the designers, programmers, and users. CASE tools are helpful in creating, modifying, and sharing the design models. In addition to the diagrams, the CASE repository will maintain all of the definitions, descriptions, and comments needed to build the final application.

A Developer's View

Like any developer, Miranda needs a method to write down the system goals and details. The feasibility study documents the goals and provides a rough estimate of the costs and benefits. The class diagram identifies the main entities and shows how they are related. The class diagram, along with notes in the data dictionary, records the business rules. For your class project, you should study the case. Then create a feasibility study and an initial class diagram.

Key Terms

aggregation
association
association role
attribute
binary large object (BLOB)
class
class diagram
class hierarchy
collaboration diagram
composition
data normalization
data type
derived class
encapsulation
entity
generalization
inheritance
method
multiplicity
n-ary association
null
polymorphism
primary key
property
rapid application development (RAD)
reflexive association
relational database
relationship
table
Unified Modeling Language (UML)

Review Questions

1. How do you identify user requirements?
2. What is the purpose of a class diagram (or entity-relationship diagram)?
3. What is a reflexive association and how is it shown on a class diagram?
4. What is multiplicity and how is it shown on a class diagram?
5. What are the primary data types used in business applications?
6. How is inheritance shown in a class diagram?
7. How do events and triggers relate to objects or entities?
8. What problems are complicated with large projects?
9. How can computer-aided software engineering tools help on large projects?
10. What is an application?

Exercises

1. Most medical practices use turn-key systems to handle billing data and basic electronic medical records. But a local physician wants a system to help him perform deeper statistical analyses on basic patient data. Specifically, he wants to track basic test results for patients and wants to compare them by families. For example, he wants to see if families where parents have high blood pressure also affect the blood pressure of the children and at what age those values change. Initially, he just wants to track basic medical values including heart rate, pressure, and basic blood test. Notes:

Patient Last name, First Name Date of Birth Gender Race Tobacco y/n Alcohol y/n Marital Status Phone, e-mail Address, City, State, ZIP	Father: Mother: Family history notes, particularly if data is not available.
Test date: Last meal time:	

Test	Meas.	Value	Low	High	Comments
Albumin	*g/dL*		*3.9*	*5.0*	
Alkaline phosphatase	*IU/L*		*44*	*147*	
ALT	*IU/L*		*8*	*37*	
AST	*IU/L*		*10*	*34*	
BUN	*mg/dL*		*7*	*20*	
Calcium	*mg/dL*		*8.5*	*10.9*	
Chloride	*mmol/L*		*96*	*106*	
CO2	*mmol/L*		*20*	*29*	
Creatinine	*mg/dL*		*0.8*	*1.4*	
Glucose	*mg/dL*		*100*		
Potassium	*mEq/L*		*3.7*	*5.2*	
Sodium	*mEg/L*		*136*	*144*	
Total bilirubin	*mg/dL*		*0.2*	*1.9*	
Total protein	*g/dL*		*6.3*	*7.9*	
Blood Pressure-systolic	*mm Hg*		*90*	*140*	
Pressure-diastolic	*mm Hg*		*60*	*90*	
Heart rate	*bpm*		*50*	*90*	

2. A local store that sells household appliances wants a database to track special orders. Most of the items ordered are large and from high-end vendors so they are too expensive to stock in the display room. Also, customers tend to order them when they are remodeling their houses so they do not want the items immediately. Instead, they need to be ordered and scheduled for delivery on a specified date. Of course, construction delays are common so the managers also need the ability to delay delivery by a few days or weeks when necessary. Most of the items are ordered directly from the manufacturers and they are good at scheduling deliveries, but sometimes highly-customized items need to be tracked down at other stores across the nation.

Customer Name Address (delivery location) City, State, ZIP Salesperson Deposit Amount: $	Order Date Desired Delivery Date Comments

Item	Manufac/Loc.	ModelID	Color	Descrip/Size	Price	Actual Deliv.

Delivery comments and changes:

Contact Date	Item (or All)	Employee	Comments	New Deliv. Date

If the item is not available from the manufacturer, the location is specified in terms of the store and contact information. Typically, it is ordered immediately and held in storage until needed; because it is too hard to find a new version so safer to buy it now.

3. A friend of yours lives in a town with many older houses and he repairs antique lights for homeowners. Many of the lights use crystals and colored panels that were designed by artists. Fortunately, most of the electrical components are relatively standard and compatible with today's parts. The most common problems involve the wiring, because old wires used paper and plastic insulation that tends to crack and disintegrate over time. He often has to rewire the entire light and he usually replaces the bulb sockets at the same time. When possible, he takes down the lights and brings them to his shop, other times he has to set up appointments to do the work in place because the light cannot be removed easily. He needs an application to track appointments, the work done, and a basic billing system that includes his time and the parts used.

Contact Last Name, First Name Phone Address City, State, ZIP	Contact Date Referred by: Date Paid:
Description of problem Description of light, style, est. year In-place or shop	

Date	Work Performed	Hours

Part No.	Description	Quantity	Cost	Source

Total Hours: _____ Total Parts: _____ Amount Due: _____	Rate: _____ Value: _____

4. A local day spa wants you to build an application that can be used to track services provided by the various employees. You do not need to handle reservations and scheduling—which are currently handled by an online provider based primarily on number of slots available during the day. Instead, you want to focus on billing and payments for an application that will be used when clients arrive. In the past, a paper card would be created for guests and services listed on the cards. Payment methods often include gift cards. Staff members often write up comments regarding treatments of clients so they can refer to them when the client returns in the future. Most treatments have a set amount of time, such as 50 minutes for a massage. At the end of the visit, clients are asked to evaluate the staff members in terms of the service quality, knowledge, and friendliness. For most people, the owner simply talks to the guests and then fills out a form later. In a modern twist, the staff members are also asked to rate the clients—largely in terms of dealing with special requests; which might lead to changes in the treatment offerings.

Guest Name Cell phone, Address, City, State, ZIP Health issues or concerns	Date Payment method

Room & Time	Treatment	Staff Specialty, Phone	Staff comments	Amount & Tip

	Subtotal Tax Total

Guest, Gender, Approx. Age Facility comments/overall	Comments

Staff member	Treatment	Quality	Knowledge	Friendliness	Staff rate Guest

Suggested changes in treatments

Treatment	Change	Est. Time

5. A small company makes winter gloves for men and women. Originally, the gloves were woven wool, but recently the company has also added leather gloves and might consider synthetic materials in the future. The woolen gloves come in a variety of colors. Sizes are typically small, medium, and large which are slightly different for men and women (largely in terms of finger length). The factory also produces different styles which tend to be variations in length of the glove, cuffs, or designs in the stitching or emblems. The company needs a database to track production and shipments.

   Production runs emphasize a single style, material, and size. Changes in yarn or material color do not require reconfiguring the machine so colors are tracked within the same production run. The IDs from the input material are tracked and an employee inspects the output batch and adds any comments.

Production Run ID Machine ID Material Glove Style Men/Women Size	Date Start Time Employee Last/First Name Job Title InspectorID

Color	Quantity Made	Material Batch	Qty Rejected	Comments

Order ID Customer/Store Contact Person Address City, State, ZIP	Order Date Ship Date

ItemID	Description	Size	Color	Style	Gender	Quantity	Sale Price

6. A start-up company is assembling customized stereo ear-buds. Instead of shipping dozens of different sizes and shapes of in-ear pieces, the company distributes a clear piece of plastic with various marks. Customers hold the piece on their ear and take a photograph. A system at the company reads the markings and determines the best ear-piece for each customer. The assembly team then builds the custom ear-buds for the customer allowing them to set additional specifications including colors, logos, number of "armatures" or speaker cones, and control switches for android or Apple devices. The company needs a database to track orders, assembly, and shipping.

Customer Last name, First name Phone, e-mail Address City, State, Postal Code Country		Referrer/Source
Measure Photo Horizontal Distance Vertical Distance Depth	Left Ear	Right Ear
Color Wire color/type Logo Armatures Apple/Android		

Assembly Date
Work station #
Employee Name, Date Hired, Title

Customer Order ID	Item ID	Quantity	Comments	Start Time End Time
Comments/ Changes				

Inspector ID, Supervisor

Test	Value	Pass/Fail
Audio low		
Audio high		
Stretch		

Inspection outcome:

7. A rich uncle owns about a dozen classic automobiles. He keeps them in several garages around town and actually drives them for different events. At a recent family holiday get-together, he mentioned that he struggles to remember the maintenance schedules for all of the vehicles. He has records for all of the service work, including oil changes, tune-ups, and other repairs. But currently, they are just paper receipts and he needs a way to track and schedule the maintenance so that any of the cars will be available for use when he wants it. He also would like to plan his usage so that he doesn't have to get all of the cars serviced at the same time. You suggested that it would be a good application for a database. Actually, you are really hoping he will let you borrow one of them for a date; but first you have to design the database.

Car Make, Model, Year, Color VIN Storage Location, Name, Address Size, Heat/Cool Monthly Cost	Date Acquired Exterior Condition Interior Condition Amount Paid Source

Manufacturer Service Intervals

Miles Interval *3000*	*Oil change*
	Grease frame fittings
	Tire pressure
15000	*Replace spark plugs,…, Tune-up*
	Rotate tires
	Grease door fittings

Actual Service Records

Miles	Date	Location	Service	Comments	Cost	
					Labor	Parts

8. Experience exercise: Talk to a friend, relative, or local manager to identify a basic job and create a class diagram for the problem.
9. Identify the typical relationships between the following entities. Write down any assumptions or comments that affect your decision. Be sure to include minimum and maximum values. Use the Internet to look up terms and examples.
   a) Company, CEO
   b) Restaurant, Cook
   c) TV Show, commercial ad
   d) E-mail address, computer user
   e) Item, List price
   f) Car, Car wash
   g) House, Painter
   h) Dog, Owner
   i) Manager, Worker
   j) Doctor, Patient

10. For each of the entities in the following list (left side), identify whether each of the items on the right should be an attribute of that entity or a separate entity.

a)	Employee	Name, Date Hired, Manager, Spouse, Job
b)	Factory	Manager, Address, Supplier, Machine, Size
c)	Boat	Dock, Length, Passenger, Captain, Weight
d)	Dentist	Patient, Graduate School, Emergency Phone, Drill
e)	Library	Book, Librarian, Number of Books, Visitor

Sally's Pet Store

11. Do some initial research on retail sales and pet stores. Identify the primary benefits you expect to gain from a transaction processing system for Sally's Pet Store. Estimate the time and costs required to design and build the database application.
12. Extend the class diagram by adding comments about each animal, beginning with adoption group remarks and including comments by employees and customers.
13. Write classes for the pet store case to track special sales events. Every couple of months the store has clearance sales and places specific items on sale. Eventually, Sally wants to evaluate the sales data to see how customers respond to the reduced prices.

14. Extend the pet store class diagram to include scheduling of appointments for pet grooming.

Rolling Thunder Bicycles

15. The Bicycle table includes entries for several employees who worked on the bike. The advantage to this approach is that it leaves all the work in one table and identifies the work performed, making it easier to enter the data. The drawback is that it is more difficult to query (and would require several links to the Employee table). Redesign the table to eliminate these problems.

16. Rolling Thunder Bicycles is thinking about opening a chain of bicycle stores. Explain how the database would have to be altered to accommodate this change. Add the proposed components to the class diagram.

17. If Rolling Thunder Bicycles wants to add a Web site to sell bicycles over the Internet, what additional data needs to be collected? Extend the class diagram to handle this additional data.

Corner Med

18. One of the first things Corner Med needs for the database is the ability to enter multiple numbers for the physicians, such as pager and cell phone. Add the necessary class.

19. Corner Med needs more information about insurance companies. Each company requires claims to be submitted to a specific location. Today, much of the data can be submitted electronically, so there will be an electronic address as well as a physical address. There will also be an account number and password, as well as a phone number and contact person. Add these elements to the class diagram.

20. In theory, prescriptions could be handled as ICD10 procedures. However, because of various drug laws, including pharmacy verification and tracking needs, it is easier to store the data separately. Add the class(es) to the diagram to handle drug prescriptions. Be sure to include the drug name, the dosage, instructions for taking the drug, and the time period. Note that you do not need to add a Drug table because it would be too large and change too often; although the physicians might want to add the Physician's Desk Reference (PDR) on CD later.

Web Site References

http://www.rational.com/uml/	The primary site for UML documentation and examples.
http://www.iconixsw.com	UML documentation and comments.
http://docs.oracle.com/cd/B28359_01/server.111/b28318/datatype.htm	Oracle data type description.
http://msdn.microsoft.com/en-us/library/ms187752(SQL.90).aspx	SQL Server data types.
http://msdn.microsoft.com/en-us/library/ms130214.aspx	SQL Server Books Online documentation.
http://JerryPost.com/DBDesign	Database design system.

Additional Reading

Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, 13 no. 6, (1970), pp. 377-387. [The paper that initially described the relational model.]

Constantine, L., "Under Pressure," Software Development, October 1995, pp. 111-112. [The importance of design.]

Constantine, L., "Re: Architecture," Software Development, January 1996, pp. 87-88. [Update on a design competition.]

McConnell, S., Rapid Development: Taming Wild Software Schedules, Redmond: Microsoft Press, 1996. [An excellent introduction to building systems, with lots of details and examples.]

Penker, M. and H. Eriksson, Business Modeling with UML: Business Patterns at Work, New York: John Wiley & Sons, 2000. [Detailed application of UML to business applications.]

Silverston, Len, The Data Model Resource Book, Vol 1 and 2, 2001, New York: John Wiley & Sons. [A collection of sample models for a variety of businesses.]

Appendix: Database Design System

Many students find database design to be challenging to learn. The basic concept seems straightforward: define a table that represents one basic entity with columns that describe the properties to hold the necessary data. For example, a Customer table will have columns for CustomerID, LastName, FirstName, and so on. But it is often difficult to decide exactly which columns belong in a table. It is also difficult to identify the key columns, which are used to establish relationships among tables. The design is complicated by the fact that the tables reflect the underlying business rules, so students must also understand the business operations and constraints in order to create a design that provides the functionality needed by the business.

In addition to reading Chapters 2 and 3 closely, one of the most important steps in learning database design is to work as many problems as possible. The catch is that students also need feedback to identify problems and improve the design. An online expert system is available to instructors and students to provide this immediate feedback. This online system is available at: http://JerryPost.com/DBDesign. This appendix uses the DB Design system to highlight a graphical approach to designing a database. However, even if you do not use the DB Design system, this appendix provides a useful summary of how to approach database design.

The design process in this appendix is illustrated with a generic sales order form. If you are unfamiliar with order forms and the entire ordering process, check out the Universal Business Language on the Oasis Web site at http://docs.oasis-open.org/ubl/cd-UBL-1.0. This organization has defined a generic purchasing process that applies to any organization. The goal is to create a standard means of transferring data among businesses. The specification includes several XML schema definitions. Because the goal is to create a generic format, the specification is considerably more complex than the example presented here, but the document also defines the common terms, processes, and business rules.

Sample Problem: Customer Orders

It is easiest to understand database design and the DB Design system by following an example. Customer orders are a common situation in business databases, so

Figure 2.1A

Typical order form. Each order can be placed by one customer but can contain multiple items ordered as shown by the repeating section.

Order Form

Order # Date
Customer
First Name, Last Name
Address
City, State ZIP

Item	Description	List Price	Quantity	QOH	Value
			Order total:		

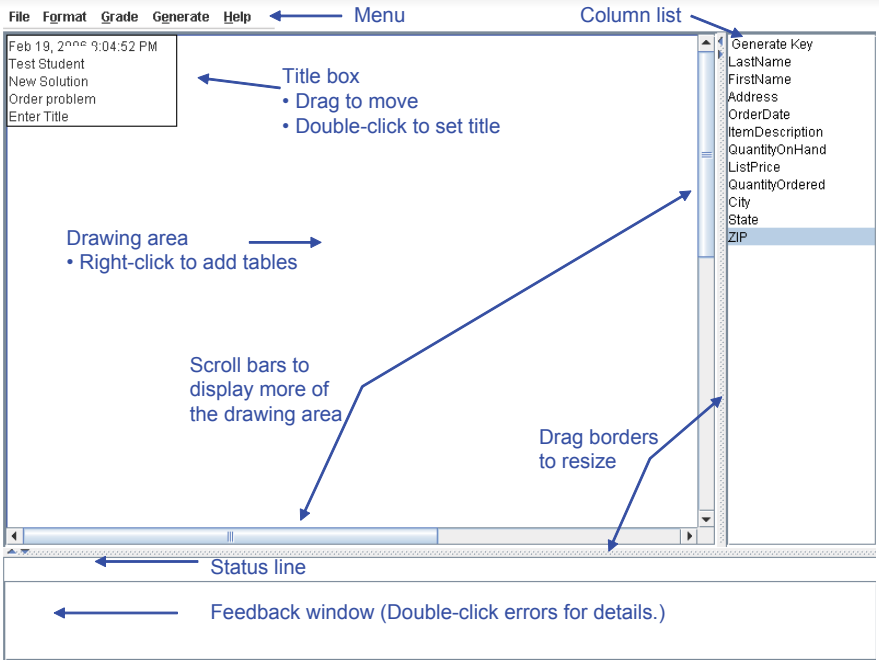


Figure 2.2A

DB Design screen. Once you log in, use the menu option File/Open to choose the Order Problem. The Help menu has an option to View the Problem. The right-hand window contains a list of the available columns that will be placed into tables. Selecting the Grade menu option generates comments in the feedback window.

consider the simple sales order form displayed in Figure 2.1A. The layout of the form generally provides information about the business rules and practices. For example, there is space for only one customer on the order, so it seems reasonable that no more than one customer can participate in an order. Conversely, the repeating section shows multiple rows to allow several items to be ordered at one time. These one-to-many relationships are important factors in the database design.

Getting Started: Identifying Columns

One of the first steps in creating the database design is to identify all of the properties or items for which you need to collect data. In the example, you will need to store customer first name, last name, address, and so on. You will also need to store an order number, order date, item description, and more. Basically, you identify each item on the form and give it a unique name. Note that some items can be easily computed and will not need to be stored. For instance, value is list price times quantity, and the order total is the sum of the value items. In a business environment, you will have to identify these items yourself and write them down. The DBDesign system handles this step for you and displays all of the columns in a list.

As shown in Figure 2.2A, after you have opened a problem, the DB Design system provides you with a list of items from the form. This list is presented in the

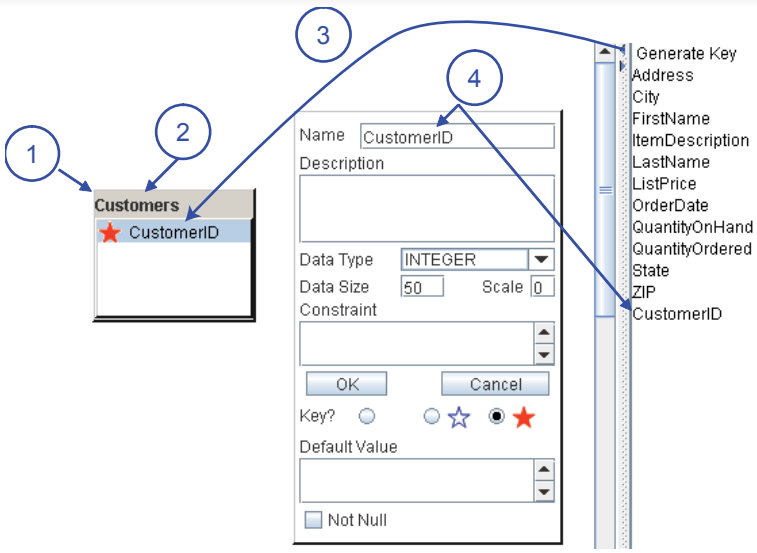


Figure 2.3A

Adding a table and key. (1) Right click and select Add table. (2) Enter a new name (Customer) in the title box. (3) Drag the Generate Key item onto the table. (4) Enter a new name (CustomerID) in the edit box and click the OK button.

right-hand column. The list of columns is the foundation for the database design. Your job is to create tables and then select the columns that belong in each table. You can rename the columns by right clicking the column name and selecting the Rename option, but be careful to use names that represent the data. Also, key columns should have unique names. To get a better grasp of the columns available, you can sort the list by right clicking the list and selecting the Sort option. You can also double-click a column to see more details about it, including a brief description. If two columns have the same name (such as LastName), you will have to look at the description to see which entity it refers to (such as employee or customer).

Creating a Table and Adding Columns

The main objective is to create tables and specify which columns belong in each table. It is fairly clear that the sale order problem will need a table to hold customer data, so begin by right-clicking the main drawing window and selecting the option to add a table. The system enters a default name for the table, but you should change it by typing in a new name. Later, you can change the name by right-clicking the name and selecting the rename option. For this demonstration, enter "Customer" to provide the new name.

Each table must have a primary key—one or more columns that uniquely identify each row in the table. Looking at the order form and the column list you will not see a column that can be used as a primary key. You might consider using the customer phone number, but that presents problems when customers change their numbers. Instead, it is best to generate a new column called CustomerID. To ensure each customer is given a different ID value, the data for this column will

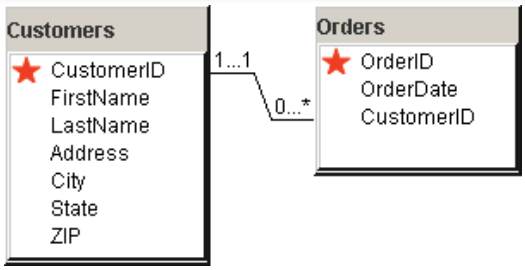


Figure 2.4A

Two tables. Each table represents a single entity, and all columns are data collected for that entity. The Orders table contains the CustomerID, which provides a method to obtain the matching data in the Customer table. Build the Customer table first, followed by the Orders table and the relationship line will probably be added automatically for you.

be generated by the DBMS whenever a new customer is added. To create a new key column that is generated by the DBMS, drag the Generate Key item from the column list and drop it on the Customers table. The column-edit form will pop up with a temporary name. Type a new name for the column (CustomerID). You can enter a description if you want. Click the OK button when you are ready. Notice that CustomerID will be displayed in the Customers table and as a new column in the column list. Also, notice in Figure 2.3A that the CustomerID is marked with a filled red star to indicate that it is part of the primary key in the Customers table. You can edit a column name and description later by double-clicking the column name.

A star in the DB Design system indicates that a column is part of the primary key for a table. But, there are two types of stars: (1) a filled red star, or (2) an open blue star. Both indicate that the column is part of the primary key. The filled red star additionally notes that the key values are generated in that table whenever a row is added. Because generated values must always be unique, any table that contains a generated key column can only have that column as the primary key. You can change the key attribute by opening the column-edit form or by double-clicking the space in front of a column name. As you double-click the space, the key indicator will rotate through the three choices: blank (no key), blue star (key), red star (generated key).

Now that the table and primary key are established, you can add other columns to the table. But which columns? The Customers table should contain columns that identify attributes specifically about a customer. So, find each column that is strictly identified by the new primary key CustomerID and drag it onto the Customers table.

Relationships: Connecting Tables

Almost all database problems will need multiple tables. In the sales order problem, it is fairly clear that the database design will need an Orders table. Add a new table, name it "Orders," and generate a key for OrderID. Once again, you need to identify the columns that belong in the Order table. Looking at the Order form, you should add the OrderDate column. Notice that the order form also contains

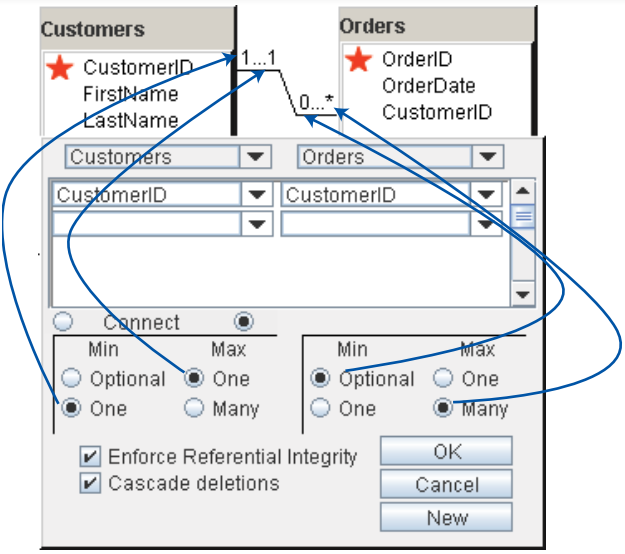


Figure 2.5A

Relationships. Drag the CustomerID column from the Customer table and drop it onto the CustomerID column in the Orders table. Then set the minimum and maximum values for each side of the relationship. An order must have exactly one customer, and a customer can place from zero to many orders.

customer information. But it would seem to be a waste of effort to require clerks to enter a customer's name and address for every order. Instead, you need to add only the CustomerID in the Order table.

When you add the CustomerID to the Orders table, as shown in Figure 2.4A, the system will create a relationship back to the Customers table. It will even try to get the multiplicity correct. Actually, your instructor can turn off the automatic relationship and the multiplicity options, so there is a small chance that you will have to create the relationship by hand. You can delete a relationship by right-clicking the sloping line and choosing the Delete option. You edit a relationship by double-clicking the connecting line. You create a new relationship by dragging a column from one table and dropping it onto the matching column in a second table.

Remember that CustomerID will not be a primary key in the Order table, because for each order, there can be only one customer. If it were keyed, you would be indicating that more than one customer could take part in an order.

You often need to edit the multiplicity values when you create a relationship. If all key columns are specified correctly, the system does a good job of setting the values automatically. But, read that "if" condition again and you quickly realize that you will have to edit multiplicity values for many of your relationships. Double-click the connection line to open the relationship edit window. Figure 2.5A shows how the selections are displayed. Your form might be slightly different from the one shown because the form is dynamic. It looks at the diagram and displays the left-most table on the left. If your layout is different, the table names will change positions to match your diagram. Every relationship has four values: a minimum and maximum on each end of the relationship. These values are set with

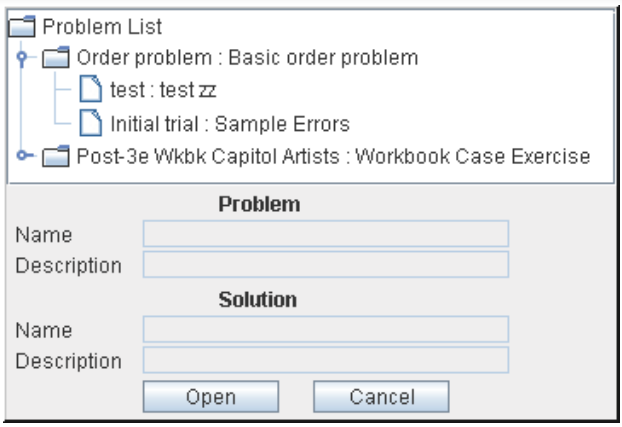


Figure 2.6A

Opening solutions. You can save multiple versions or solutions for any problem. To open a saved solution, you have to expand the list by clicking the handle icon in front of the problem name.

the option buttons. In this case, an order can be placed by exactly one customer, so the minimum customer value is one and the maximum value is also one. On the other side of the relationship, each customer can place from zero to many orders. Some might argue that if a customer has not placed any orders, then he or she is only a potential customer, but the difference is not critical to the database design.

The relationship-edit form has a couple of other options. The Connect option box is useful when two tables are displayed vertically (above and below, instead of the left and right used here). It enables you to specify the preferred side for the relationship line (left or right). Look at the boxes containing the CustomerID values, and you can use the drop-down lists to change the column matches if you made a mistake when you dropped a column while building the relationship. You can also create a relationship that connects tables on multiple columns by moving to a new row and choosing the matching columns. For more complex cases, you can click the New button to create multiple relationships between two tables. For example, you might need to connect a City.CityID column to both an Order.DeliveryCity and an Order.BillingCity column. These would be two separate, independent relationships. None of these more complicated options are needed for this example, but it is good to know they exist.

Saving and Opening Solutions

Be sure to save your work as you go. If you wait too long, the Internet connection will time-out and you might lose your changes. In most cases, if you lose your session, you can log in and try again. The first time you save your solution, you will be asked to give it a name and a brief description. You can use File/Save to create copies with different names—enabling you to save multiple versions of your work. Generally, you will only need this approach for complex problems.

Even if you save only one version of your solution, you need to understand the File/Open box shown in Figure 2.6A. First, note that you can resize the box by dragging its lower right-hand corner. This trick is useful when you have a long list of problems or solutions. Second, the list is stored and displayed in a tree hi-

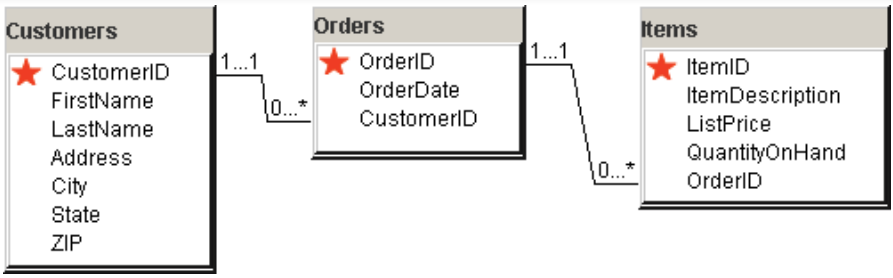


Figure 2.7A

Creating errors. To demonstrate a potential problem, add the OrderID column to the Items table and then link it to the Orders table.

erarchy that starts by listing each problem available to you. If you double-click a problem (or select one and click the Open button), you will get a blank problem where you start over. Sometimes this approach is useful if you really messed up an earlier solution. In most cases, you will want to click the handle icon in front of the problem name to open the list of solutions you saved for that problem. You can open any of the solutions you have saved.

Grading: Detecting and Solving Problems

You will repeat these same steps to create the database design: add a table, set the primary key, add the data columns, and link the tables. The DB Design system makes the process relatively easy, and you can drag tables around to display them conveniently. You can save your work and come back at a later time to retrieve it and continue working on the problem. However, you still do not know if your design is good or bad.

Consider adding another table to the sample order problem. Add a table for Items and generate a new key column called ItemID. Add the columns for ItemDescription, ListPrice, and QuantityOnHand. The problem you face now is that you need to link this new table with the Orders table. But, so far, they do not have any related columns. So, as an experiment, try placing the OrderID column into the Items table and build a relationship from Items to Orders by linking the OrderID columns, as shown in Figure 2.7A.

At any time, you can ask the server to grade the current design to see if there are problems. In fact, it is a good idea to check your work several times as you create the tables, so you can spot problems early. Use the Grade option on the menu to Grade and Mark the diagram. This option generates a list of comments in the bottom window. The Grade to HTML option generates the same list organized by tables in a separate window. Both options automatically save your work, so you do not need to worry about saving your solution as long as you continue to grade it.

As shown in Figure 2.8A, when you grade this problem, you get a reasonably good score (88.1). However, there are several important comments. When you select (click) a comment, the system highlights the error in the diagram whenever possible. Notice the first grade comment about the unused column. If you had others, they would also be listed in that message. Clicking that message will cause all of the column names to be highlighted in the right-hand side list—making them easy to find.

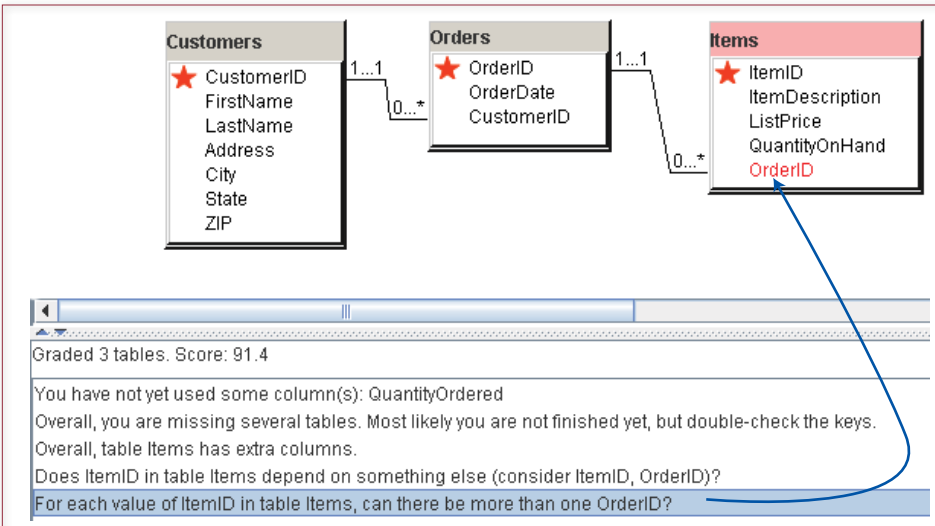


Figure 2.8A

Grading the exercise. Click a comment to highlight the table and column causing problems. In this case, each ItemID can appear in many Orders, but OrderID is not part of the key. Double-click an error message to see more information about the error.

You use the error messages to help improve the design. In this case, most of the comments indicate there is a problem with the Items table. In particular, the OrderID column is presenting a problem. The first couple of questions ask whether the key values are correct. The question highlighted at the bottom is important because it tells you how to solve the problem. It is asking whether an item can be sold on more than one order. Currently, since OrderID is not part of the key, any item can be sold only one time. This assumption is extremely restrictive and probably wrong. The system is telling you that you need a table where both ItemID and OrderID are key columns.

At this point, you really should stop and think about this entire section of the design. But, see what happens if you just look at the one comment and leap ahead. Just make OrderID a key along with ItemID. Figure 2.9A shows the result of this change. First, notice that the score actually decreased! The DB Design system is still pointing out problems with the keys. In particular, note that ItemID was created as a generated key, so it is always guaranteed to be unique. If that is true, then you would never need a second key column in the same table. As a side note, observe that you can use the Ctrl+click approach to highlight several error messages at once. The basic problem is that you cannot include the OrderID column in the Items table.

The solution is to realize that a relational database cannot support a direct many-to-many relationship between two tables (Orders and Items). Instead, you must insert a new table between the two. In this case, call it an OrderItems table. Then be sure to add the key columns from both of the linked tables (OrderID and ItemID). As shown in Figure 2.10A, add both relationships as one-to-many links.

As indicated by the score, this four-table solution is the best database design for the typical order problem. The Customers table holds data about each customer.

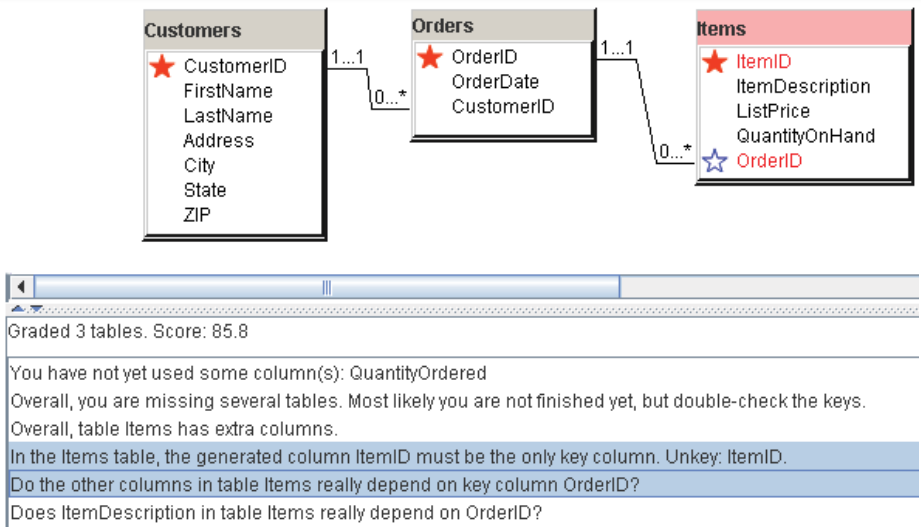


Figure 2.9A

Trying to fix the problems. You could try making OrderID part of the key, but notice that the score decreased, so the fix actually made the situation worse. The problems with the OrderID and the relationship have not been solved. You can use Ctrl+Click to highlight several errors at the same time.

The Items table contains rows that describe each item for sale. The Orders table provides the order number, date, and a link to the customer placing the order. The OrderItems table represents the repeating section of the order form and lists the multiple items being purchased on each order. You should verify that all of the data items from the initial form appear in at least one of the tables.

Specifying Data Types

You need to perform one additional step before the database design is complete. Eventually, this design will be converted into database tables. When you create the tables, you will need to know the type of data that will be stored in each column. For example, names are text data, and key columns are often 32-bit integers. Make sure that all dates and times are given the Date data type. Be careful to check when you need floating point versus integer values: use single or double depending on how large the maximum value will be. Figure 2.11A shows that you set the data type by double-clicking to open the column-edit form.

The default value is text since it is commonly used. Consequently, many columns such as customer name will not need changes. Although there are standard names for data types, every DBMS uses its own terms. You can control which terms are displayed by setting the target DBMS under the Generate menu command. This choice makes it easier for you to choose the exact data type for a particular DBMS. Internally, the DB Design system assigns a generic definition. You can use the generic definitions, the SQL standard names, or switch to one of the common DBMSs to fine-tune the choice.

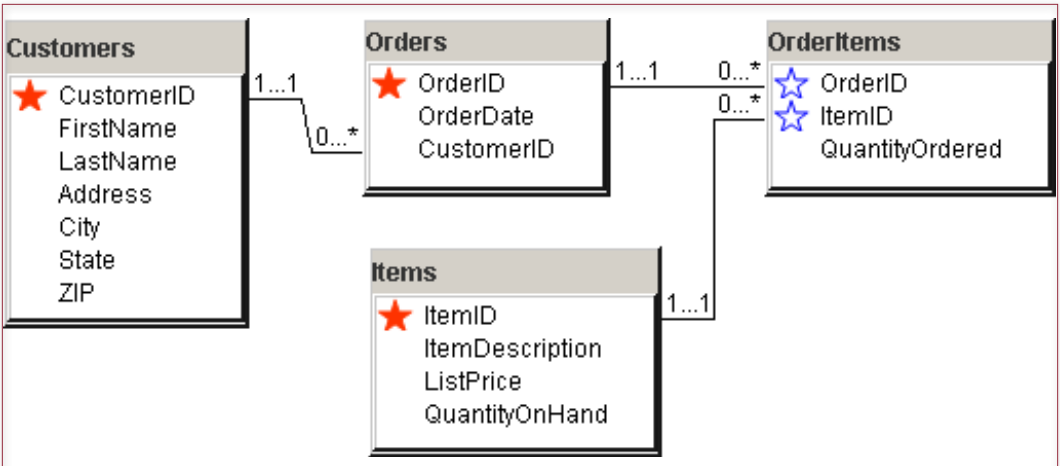


Figure 2.10A

A solution. Add the intermediate table OrderItems and include keys from both tables (OrderID and ItemID). Use one-to-many relationships to link it to both tables. Notice the difference in the key indicators. The solid red star shows where a key value is generated.

You can also set default values and constraint rules for the form. Default values are fairly standard, but the syntax of constraint rules depends heavily on the specific target DBMS. These options are provided primarily for when you want to generate complete table descriptions. Until you gain experience with your target DBMS, you should leave them blank.

Generating Tables

Once you are satisfied with your design, you can use the system to help create the tables in your DBMS. Almost all DBMSs support the SQL CREATE TABLE command. When you ask DB Design to generate tables, it writes a SQL script that you can run to generate the tables within your DBMS. Note that DB Design does not actually create the tables inside itself. You need to copy the SQL script and run it on your database server.

Use the Generate/Generate Tables menu command to open a new browser window with the SQL script. As shown in Figure 2.12A, you can scroll to the bottom of the window and change some of the options. For example, you might want to change the target DBMS. When you are satisfied with the script, click within the script window, press Ctrl+A to select all of the lines, and Ctrl+C to copy the text. Open a text editor or a script edit window in your DBMS management tool. Paste (Ctrl+V) the script and save it or execute it. If necessary, you can edit the script to fine-tune some DBMS-specific options.

If you are using Microsoft Access, read the notes at the top of the script file. While Access supports the CREATE TABLE command it does not support script files (at least through Office 2010). Consequently, you can only run one CREATE TABLE command at a time. Also, you need to hand-edit all of the final relationships inside Access because it does not support the cascade options.

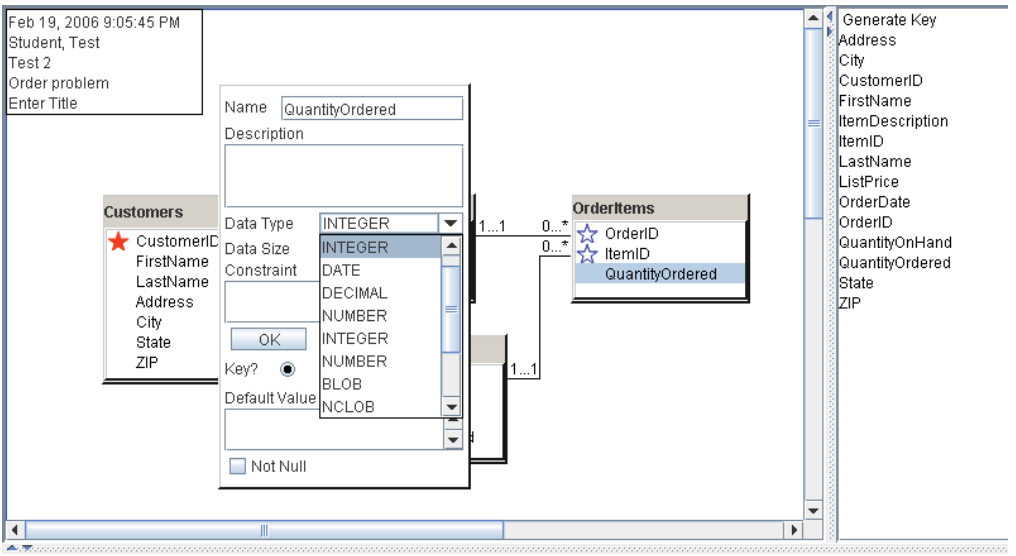


Figure 2.11A

Data types. Double-click a column name to open the edit window. Set the data type. The default is Text, so you do not have to change common columns like the customer name. You can also add a description and a default value setting. The Constraint setting has to match the format of the target DBMS.

The Generate form contains some additional options. The name delimiter is straightforward. You are not allowed to use reserved words or characters in table and column names. For instance, column names cannot include spaces. However, current DBMSs will allow you to violate these rules if you enclose the name in special delimiters. The delimiters vary by DBMS. For example, Microsoft Access and SQL Server use square brackets, while Oracle uses double quotes. If you enter a delimiter in the box (such as [or "), the generator will apply it to all table and column names. Why does the generator not apply delimiters by default? The answer is because delimiters sometimes have other consequences. In particular, if you use the double-quote delimiter (") in Oracle, the table and column names become case-sensitive. From that point, every time you reference a table or column name, you are required to enclose it in quotation marks. When you type SQL statements by hand, it is annoying to type all of those quotation marks, so it is easier to use well-formed names and avoid the delimiter completely.

Although it is not shown here, a checkbox option has been added to exclude the descriptive comments. Most of the time, you should keep the comments as documentation of the database design. However, if the comments are excessive or intrusive, you can tell the generator to leave them out.

The prefix option needs more explanation than the others. It is included because of the way DB Design works. In particular, since DB Design displays all of the columns in one list, it is helpful to ensure that the names are unique. For instance, if you see several columns called LastName, it is not immediately clear which entity or table is referenced. Consequently, it is helpful to add a prefix to the names to make them unique. For instance, you could have Emp_LastName and Cust_LastName. However, when you generate the tables in the DBMS, the

```
CREATE TABLE Orders
(
  OrderID                          INTEGER,
  OrderDate                        DATE,
  CustomerID                       INTEGER,
     CONSTRAINT pk_Orders PRIMARY KEY (OrderID),
     CONSTRAINT fk_Orders_Customer FOREIGN KEY (CustomerID)
        REFERENCES Customer (CustomerID)
        ON DELETE CASCADE
)
;
COMMENT ON COLUMN Orders.OrderDate IS 'The date the order was placed';
CREATE SEQUENCE sq_Orders INCREMENT BY 1 START WITH 1;

CREATE TABLE OrderItems
(
  OrderID                          INTEGER,
  ItemID                           INTEGER,
  QuantityOrdered                  INTEGER,
     CONSTRAINT pk_OrderItems PRIMARY KEY (OrderID, ItemID),
     CONSTRAINT fk_OrderItems_Orders FOREIGN KEY (OrderID)
        REFERENCES Orders (OrderID)
        ON DELETE CASCADE,
     CONSTRAINT fk_OrderItems_Items FOREIGN KEY (ItemID)
        REFERENCES Items (ItemID)
        ON DELETE CASCADE
)
;
```

Remove column prefix [ ] Delimiter for names [ ] DBMS [Oracle] [Rebuild]

Figure 2.12A

Generate Tables. Choose the Generate/Generate Tables menu option to create a set of SQL commands that can be run on your DBMS to build the tables created in the diagram. You can choose the target DBMS before running the command or select it on the generated page. Use Ctrl+A to select the entire text in the window, then open a text editor or a SQL editor and paste the commands with Ctrl+V.

column will gain the context of the table and the prefix is superfluous and just something extra to type (such as Employee.Emp_LastName). If you adopt a consistent naming convention, the generator can automatically remove the prefix. The easiest approach is to use an abbreviation of the entity followed by an underscore (e.g., Emp_Address, Cust_Address). When you enter the underscore character (_) into the prefix box and generate the SQL script, the generator will examine every column name and remove all characters that appear before the first underscore (and the underscore).