

## Chapter 1

# Designing ASP.NET 2.0 Applications

---

### *In This Chapter*

- ▶ Tracing the application-development life cycle
  - ▶ Getting a handle on systems analysis and design
  - ▶ Looking at layered architectures
  - ▶ Designing relational databases
  - ▶ Designing objects
- 

**A**SP.NET is Microsoft's platform for developing Web applications. With the new release of version 2.0, Microsoft has added powerful new features such as Master Pages and automatic site navigation, which make it one of the most powerful (yet easy-to-use) Web-development tools out there.

And it's inexpensive. Although the professional versions of Visual Studio will set you back some, Visual Web Developer Express Edition will cost you only about \$100 and can be used to develop sophisticated ASP.NET applications, using your choice of programming languages — Visual Basic or C#.

One way to learn ASP.NET is to buy a beginning ASP.NET book. There are plenty of good ones out there, including (in all due modesty) my own *ASP.NET 2.0 All-In-One Desk Reference For Dummies* (published by Wiley, of course). But this book takes a different approach. Instead of belaboring the myriad of details that go into ASP.NET programming, this book presents a series of complete popular applications, such as a shopping cart and a forum host, and explains in detail how these applications work. You can study these applications to see how real-world ASP.NET programming is done, and you can even copy them to give your own applications a running start.

You'll need to modify the applications, of course, to make them work for your own situation. Still, the samples presented in this book should provide an excellent starting point. Even so, before you base your app on any of the applications presented in this book, take a step back: Carefully analyze the problem the application is intended to solve — and design an appropriate

solution. This chapter presents a brief introduction to this process, known in software development circles as *analysis and design*. Along the way, you get a look at the basics of designing relational databases, as well as designing objects to work with an ASP.NET application.

## The Development Treadmill

Over the years, computer gurus have observed that computer projects have a life of their own, which goes through natural stages. The *life cycle* of an application-development project typically goes something like this:

1. **Feasibility study:** This is the conception phase, in which the decision to undertake a new computer system is made based on the answers to questions such as:
  - What business problem will the new system solve?
  - Will the new system actually be an improvement over the current system?
  - If so, can the value of this improvement be quantified?
  - Is the new system possible?
  - What will the new system cost to develop and run?
  - How long will the system take to develop?

The result of the feasibility study is a charter for the new project that defines the scope of the project, user requirements, budget constraints, and so on.

2. **Analysis:** This is the process of deciding exactly what a computer system is to do. The traditional approach to analysis is to thoroughly document the existing system that the new system is intended to replace, even if the existing system is entirely manual and rife with inefficiency and error. Then, a specification for a new system to replace the old system is created. This specification defines exactly what the new system will do, but not necessarily how it will do it.
3. **Design:** This process creates a plan for implementing the specification for a new system that results from the analysis step. It focuses on how the new system will work.
4. **Implementation:** Here's where the programs that make up the new system are coded and tested, the hardware required to support the system is purchased and installed, and the databases required for the system are defined and loaded.
5. **Acceptance testing:** In this phase, all pieces of the system are checked out to make sure that the system works the way it should.

6. **Production:** This is another word for “put into action.” If the system works acceptably, it’s put into production: Its users actually begin using it.
7. **Maintenance:** The moment the computer system goes into production, it needs maintenance. In this dreaded phase, errors — hopefully minor — that weren’t caught during the implementation and acceptance phases are corrected. As the users work with the system, they invariably realize that what they *really* need isn’t what they said they wanted, so they request enhancements — which are gradually incorporated into the system.



The biggest challenge of this phase is making sure that corrections and enhancements don’t create more problems than they solve.

8. **Obsolescence:** Eventually, the new system becomes obsolete. Of course, this doesn’t mean the system dies; it probably remains in use for years, perhaps even decades, *after* it becomes “obsolete.” Many obsolete COBOL systems are still in production today, and Web applications being built today will be in production long after ASP.NET becomes passé.

Only the most obsessive project managers actually lead projects through these phases step by step. In the real world, the phases overlap to some degree. In fact, modern development methods often overlap all phases of a highly *iterative* process where the approach is “try, hit a snag, make changes, try again with a new version.”



I omitted two important pieces of the computer-system-development puzzle because they should be integrated throughout the *entire* process: *quality assurance* and *documentation*. Quality needs to be built into each phase of development, and shouldn’t be tacked on to the end as an afterthought. Likewise, documentation of the system should be built constantly as the system is developed, to minimize confusion.

## Building Models

When it comes right down to it, computer system analysis and design is nothing more than glorified model-building. (Minus the glue fumes.)

Most engineering disciplines involve model-building. In fact, that’s what engineers do all day: sit around building fancy models of skyscrapers, bridges, freeway overpasses, culverts, storm drains, whatever.

These models usually aren’t the kind made of molded plastic parts and held together with cement (though sometimes they are). Instead, they’re conceptual models drawn on paper. Architects draw floor plans, electrical engineers draw schematic circuit diagrams, structural engineers draw blueprints; these are all nothing more than models.

The reason engineers build models is that they're cheaper to build (and break) than the real thing. It's a lot easier to draw a picture of a bridge and examine it to make sure it won't collapse the first time the wind blows too fast or the river is too full than it is to build an actual bridge and *then* find out.

The same holds true for computer-application design. Building a computer system is an expensive proposition. It's far cheaper to build a paper model of the system first, and then test the model to make sure it works before building the actual system.

## What Is an Application Design?

Glad you asked. An *application design* is a written model of a system that can be used as a guide when you actually construct a working version of the system. The components of an application design can vary, but the complete design typically includes the following:

- ✔ **A statement of the purpose and scope of the system:** This statement of purpose and scope is often written in the form of a *use case*, which describes the actors and actions (users and uses) that make up the system and shows what it's for. Sometimes the use case is a graphic diagram; most often it's plain text.
- ✔ **A data model:** Normally this is an outline of the database structure, consisting of a set of *Entity-Relationship Diagrams (ERDs)* or other diagrams. These describe the details of how the application's database will be put together. Each application in this book uses a database and includes an ERD, which describes how the database tables relate to each other.
- ✔ **Data Flow Diagrams (DFDs):** Some application designs include these diagrams, which show the major processes that make up the application and how data flows among the processes. The data flow is pretty straightforward for most of the applications presented in this book, so I don't include Data Flow Diagrams for them.
- ✔ **User Interface Flow Diagrams:** These are sometimes called *storyboards* and are often used to plan the application's user interface. I include a User Interface Flow Diagram for each application in this book so you can see how the application flows from one page to the next.

## Using Layered Architectures

One approach to designing Web applications is to focus on clearly defined layers of the application's architecture. This approach is similar to the way

an architect designs a building. If you've ever seen detailed construction plans for a skyscraper, you know what I'm talking about. The construction plans include separate blueprints for the foundation, frame, roof, plumbing, electrical, and other floors of the building.

With a layered architecture, specialists can design and develop the “floors” — called *layers* — independently, provided that the connections between the layers (the *interfaces*) are carefully thought out.

The layers should be independent of one another, as much as possible. Among other things, that means heeding a few must-dos and shalt-nots:

- ✔ **Each layer must have a clearly defined focus.** To design the layers properly, you must clearly spell out the tasks and responsibilities of each layer.
- ✔ **Layers should mind their own business.** If one layer is responsible for user interaction, only that layer is allowed to communicate with the user. Other layers that need to get information from the user must do so through the User Interface Layer.
- ✔ **Clearly defined protocols must be set up for the layers to interact with one another.** Interaction between the layers occurs only through these protocols.

Note that the layers are not tied directly to any particular application. For example, an architecture might work equally well for an online ordering system and for an online forum. As a result, layered architecture has nothing to do with the ERDs that define a database or the Data Flow Diagrams that define how the data flows within the application. It's a separate structure.

## *How many layers?*

There are several common approaches to application architecture that vary depending on the number of layers used. One common scheme is to break the application into two layers:

- ✔ **Application Layer:** The design of the user interface and the implementation of business policies are handled in this layer. This layer may also handle *transaction logic* — the code that groups database updates into transactions and ensures that all updates within a transaction are made consistently.
- ✔ **Data Access Layer:** The underlying database engine that supports the application. This layer is responsible for maintaining the integrity of the database. Some or all the transaction logic may be implemented in this layer.

In the two-layer model, the Application Layer is the ASP.NET Web pages that define the pages presented to the user as well as the code-behind files that implement the application's logic. The Data Access Layer is the database server that manages the database, such as Microsoft SQL Server or Oracle.

Note that ASP.NET 2.0 doesn't require that you place the application's logic code in a separate code-behind file. Instead, you can intersperse the logic code with the presentation code in the same file. However, it's almost always a good idea to use separate code-behind files to separate the application's logic from its presentation code. All of the applications presented in this book use separate code-behind files.

## Using objects in the Data Access Layer

One of the fundamental architecture decisions you need to make when developing ASP.NET applications is whether to create customized data classes for the Data Access Layer. For example, an application that accesses a Products database might incorporate a class named `ProductDB` that includes methods for retrieving, inserting, updating, and deleting data in the Products database. Then, the other layers of the application can simply call these methods to perform the application's data access.

Creating custom data-access classes like this has several advantages:

- ✔ The data-access code is isolated in a separate class, so you can assign your best database programmers to work on those classes.
- ✔ You can fine-tune the database performance by spending extra time on the data-access classes without affecting the rest of the application.
- ✔ If you need to migrate the application from one database server to another (for example, from SQL Server to Oracle), you can do so by changing just the data-access classes.
- ✔ You can design the data-access classes so they work with a variety of databases. Then,

you can let the user configure which database to use when the application is installed.

However, this flexibility isn't without cost. ASP.NET is designed to work with the data-source controls embedded in your `.aspx` pages. If you want to create your own data-access classes, you have basically two choices:

- ✔ Don't use the ASP.NET data sources, which means you can't use data binding. Then, you must write all the code that connects your user interface to your data-access classes. That's a lot of work.
- ✔ Use the new ASP.NET 2.0 object data sources, which are designed to let you bind ASP.NET controls to custom data-access classes. Unfortunately, this adds a layer of complexity to the application and often isn't worth the trouble.

The applications in this book don't use custom data-access classes. However, you should be able to adapt them to use object data sources if you want.

For more information about designing objects for ASP.NET applications, see the "Designing Objects" section, later in this chapter.

The division between the Application and Data Access layers isn't always as clear-cut as it could be. For performance reasons, transaction logic is often shifted to the database server (in the form of stored procedures), and business rules are often implemented on the database server with constraints and triggers. Thus, the database server often handles some of the application logic.

If this messiness bothers you, you can use a *three-layer architecture*, which adds an additional layer to handle business rules and policies:

- ✓ **Presentation Layer:** This layer handles the user interface.
- ✓ **Business Rules Layer:** This layer handles the application's business rules and policies. For example, if a sales application grants discounts to certain users, the discount policy is implemented in this layer.
- ✓ **Data Access Layer:** The underlying database model that supports the application.

Creating a separate layer for business rules enables you to separate the rules from the database design and the presentation logic. Business rules are subject to change. By placing them in a separate layer, you have an easier task of changing them later than if they're incorporated into the user interface or database design.

## Model-View-Controller

Another common model for designing Web applications is called *Model-View-Controller (MVC)*. In this architecture, the application is broken into three parts:

- ✓ **Model:** The *model* is, in effect, the application's business layer. It usually consists of objects that represent the business entities that make up the application, such as customers and products.
- ✓ **View:** The *view* is the application's user interface. In a Web application, this consists of one or more HTML pages that define the look and feel of the application.
- ✓ **Controller:** The *controller* manages the events processed by the application. The events are usually generated by user-interface actions, such as the user clicking a button or selecting an item from a drop-down list.

In a typical ASP.NET application, the `.aspx` file implements the view; the model and controller functions are combined and handled by the code-behind file. Thus, the code-behind file can be thought of as the *model-controller*.



You can, of course, separate the model and controller functions by creating separate classes for the business entities. For simplicity, the applications in this book keep the model and controller functions combined in the code-behind file.

## Designing the user interface

Much of the success of any Web application depends on the quality of its user interface. As far as end-users are concerned, the user interface *is* the application: Users aren't interested in the details of the data model or the design of the data-access classes.

In an ASP.NET Web application, the user interface consists of a series of `.aspx` pages that are rendered to the browser using standard HTML. Designing the user interface is simply a matter of deciding which pages are required (and in what sequence) — and populating those pages with the appropriate controls.

Standard HTML has a surprisingly limited set of user-input controls:

- ✓ Buttons
- ✓ Text boxes
- ✓ Drop-down lists
- ✓ Check boxes
- ✓ Radio buttons

However, ASP.NET offers many controls that build on these basic controls. For example, you can use a GridView control to present data from a database in a tabular format.



All ASP.NET controls are eventually rendered to the browser, using standard HTML. As a result, even the most complicated ASP.NET controls are simply composites made of standard HTML controls and HTML formatting elements (such as tables).

Designing the user interface can quickly become the most complicated aspect of a Web application. Although user interface design has no hard-and-fast rules, here are a few guidelines you should keep in mind:

- ✓ Consider how frequently the user will use each page and how familiar he or she will be with the application. If the user works with the same page over and over again all day long, try to make the data entry as efficient as possible. However, if the user will use the page only once in a while, err on the side of making the page self-explanatory so the user doesn't have to struggle to figure out how to use the page.
- ✓ Remember that the user is in control of the application and users are pretty unpredictable. Users might give up in the middle of a data-entry sequence, or unexpectedly hit the browser's Back button.
- ✓ Some users like the mouse, others like the keyboard. Don't force your preference on the user: make sure your interface works well for mouse as well as keyboard users.

- ✓ Review prototypes of the user-interface design with *actual users*. Listen to their suggestions seriously. They probably have a better idea than you do of what the user interface should look like and how it should behave.
- ✓ Study Web sites that you consider to have good interfaces.

## Designing the Business Rules Layer

*Business rules* are the portion of a program that implements the business policies dictated by the application. Here are some examples of business rules:

- ✓ Should a customer be granted a credit request?
- ✓ How much of a discount should be applied to a given order?
- ✓ How many copies of Form 10432/J need to be printed?
- ✓ How much shipping and handling should be tacked onto an invoice?
- ✓ When should an inventory item that is running low on stock be reordered?
- ✓ How much sick leave should an employee get before managers wonder whether he or she has been skiing rather than staying home sick?
- ✓ When should an account payable be paid to take advantage of discounts while maximizing float?

The key to designing the business-rules portion of an application is simply to identify the business rules that must be implemented and separate them as much as possible from other parts of the program. That way, if the rules change, only the code that implements the rules needs to be changed.

For example, you might create a class to handle discount policies. Then, you can call methods of this class whenever you need to calculate a customer's discount. If the discount policy changes, the discount class can be updated to reflect the new policy.



Ideally, each business rule should be implemented only once, in a single class that's used by each program that needs it. All too often, business policies are implemented over and over again in multiple programs — and if the policy changes, dozens of programs need to be updated. (That even hurts to think about, doesn't it?)

## Designing the Data Access Layer

Much of the job of designing the Data Access Layer involves designing the database itself. Here are some pointers on designing the Data Access Layer:

- ✓ For starters, you must decide what database server to use (for example, SQL Server or Oracle).

- ✔ You'll need to design the tables that make up the database and determine which columns each table will require. For more information about designing the tables, refer to the section "Designing Relational Databases," later in this chapter.
- ✔ You must also decide what basic techniques you'll use to access the data. For example, will you write custom data-access classes that access the database directly, or will you use ASP.NET's `SqlDataSource` control to access the database? And will you use stored procedures or code the SQL statements used to access the data directly in the application code?

## Designing Relational Databases

Most ASP.NET applications revolve around relational databases. As a result, one of the keys to good application design is a good database design.

Database design is the type of process that invites authors to create step-by-step procedures, and I certainly don't want to be left out. So what follows is an ordered list of steps you can use to create a good database design for your ASP.NET application. (Keep in mind, however, that in real life most designers manage to do many, if not all, of these steps at once.)

### *Step 1: Create a charter for the database*

Every database has a reason for being, and you'll be in a much better position to create a good database design if you start by considering why the database needs to exist and what will be expected of it.

Database designers sometimes fall into one of two traps: Assuming that the data exists for its own sake, or assuming that the database exists for the sake of the Information Technology (IT) department. Of course, the database exists for its users. Before designing a database, you'd better find out why the users need the database — and what they expect to accomplish with it.

You can think of this purpose statement as a *mission statement* or a *charter* for the database. Here's an example of a charter for a database for a store that sells supplies for pirates:

The purpose of the Pirate Supply Store database is to keep track of all the products sold at the Acme Pirate Supply store. The database should include detailed information about each product and should enable us to categorize the products into one of several categories. It should also allow us to add new categories later on if we decide to sell additional types of products. And it should provide a way to display a picture of each product on our Web page. It should also keep track of our customers and keep track of each sale.

For a more complicated application, the charter will probably be more detailed than this. But the key point is that the charter should identify the unique capabilities that the user expects from the database. In this case, the flexibility to add new product categories down the road and the ability to show pictures on the Web site are key features that the user wants.

An important part of this step is examining how the data is currently being stored and to uncover the weaknesses in the status quo. If the data is currently stored in an Excel spreadsheet, carefully examine the spreadsheet. If paper forms are used, study the forms to see what kind of data is included on them. If the data is scribbled on the back of napkins, collect the napkins and scrutinize them.

## *Step 2: Make a list and check it twice*

Once you're sure you understand the purpose of the database, sit down with a yellow pad and a box of freshly sharpened #2 pencils and start writing. (You can use a word processor if you prefer, but I like to have something I can crumple up when I change my mind.) Start by listing the major tables that the database includes.

When creating and fiddling with the lists of tables and data items, it helps to think in terms of *entities*: tangible, real-world objects that the database needs to keep track of, such as people and things. For the `Pirate Supply Store` database mentioned in Step 1, you might list the following entities:

- ✓ Products
- ✓ Categories
- ✓ Customers
- ✓ Orders

After you identify the major tables in the database, list the data elements that fall under each one. For example:

```
Products
  Name
  Category Name
  Description
  Vendor name
  Vendor address
  Vendor phone number
  Price
  Image file name

Category
```

```
Name
Customers
  Last Name
  First Name
  Address
  City
  State
  Zip Code
  Phone Number
  E-mail
  Credit Card Number
Order
  Order number
  Date
  Customer
  Product
  Quantity
  Price
  Subtotal
  Shipping
  Tax
  Total
```



Don't be afraid to crumple up the paper and start over a few times. In fact, if you're doing this step right, you'll end up with wads of yellow paper on your floor. You can clean up when you're done.

For example, you may realize that the vendor information stored in the `Products` table should actually be its own table. So you break the `Products` table into two tables, `Products` and `Vendors`:

```
Products
  Name
  Category Name
  Description
  Price
  Image file name
Vendor
  Name
  Address
  City
  State
  Zip Code
  Phone Number
  E-mail
```

As you design the database, creating additional tables like this will become a regular occurrence. You'll discover tables that need to be split because they have data for two distinct entities, or you'll discover entities that you simply forgot to include. The number of tables in a database rarely goes *down* as you refine the design.

Note that the `Orders` table has several problems in its current form. For example, how do you identify which customer is associated with an order? And, more importantly, what if more than one product is ordered? We'll solve these problems in subsequent steps.

## Step 3: Add keys

In an SQL database, every table should have a column or combination of columns that uniquely identifies each row in the table. This column (or combination of columns) is called the *primary key*. In this step, you revisit all the entities in your design and make sure each one has a useful primary key.

Selecting the primary key for a table is sometimes a challenge. For example, what field should you use as the primary key for the `Customers` table? Several choices come to mind:

- ✔ **Last Name:** This works fine until you get your *second* customer named Smith. It can also be a problem when you get a customer named Zoldoske. Every time you type this name, you'll probably spell it differently: Zoldosky, Soldoskie, Zaldosky, and so on. (Trust me on this one. My wife's maiden name is Zoldoske; she's seen it spelled each of these ways — and many more.)
- ✔ **Last and First Name combined:** This works better than Last Name alone, but you still may have *two* Lucy McGillicuddys who want to buy your stuff.
- ✔ **Phone Number:** Everyone has a unique phone number, but some phone numbers are shared by several individuals (say, roommates or family members). And when people move, they often change their phone numbers.
- ✔ **E-mail Address:** This isn't too bad a choice; people rarely share e-mail addresses and don't change them nearly as often as phone numbers.

If no field in the table jumps out as an obvious primary key, you may need to create an otherwise meaningless key for the table. For example, you could add a `Customer Number` to the `Customers` table. The `Customer Number` would be a unique number that has no meaning other than as an identifier for a specific customer. You can let the user enter a unique value for the key field, or you can let the database automatically generate a unique value. In the latter case, the key is known as an *identity column*.

In the Pirate Supply Store database, I decided to use the E-mail Address field for the primary key of the Customers table. For the Products table, I added a Product ID field that represents a unique product code determined by the users. I did the same for the Categories table. For the Orders table, I used the Order Number column and designated it as an identify column so it will be automatically generated.

As you add primary keys to your tables, you can also add those primary keys columns as *foreign keys* in related tables. For example, a Vendor ID column could be added to the Products table so each product is related to a particular vendor.

After the key columns have been added, the list looks like this:

```
Products
  Product ID (primary key)
  Name
  Category ID (foreign key)
  Category Name
  Description
  Price
  Image file name
  Vendor ID (foreign key)

Vendor
  Vendor ID (primary key)
  Name
  Address
  City
  State
  Zip Code
  Phone Number
  E-mail

Category
  Category ID (primary key)
  Name

Customers
  Last Name
  First Name
  Address
  City
  State
  Zip Code
  Phone Number
  E-mail (primary key)
  Credit Card Number

Order
  Order number (primary key)
  Date
```

```
Customer ID (foreign key)
Product ID (foreign key)
Quantity
Price
Subtotal
Shipping
Tax
Total
```

## Step 4: Normalize the database

*Normalization* refers to the process of eliminating redundant information and other problems in the database design. To normalize a database, you identify problems in the design and correct them, often by creating additional tables. After normalizing your design, you almost always have more tables than you had when you started.

Five different levels of normalization exist, known as the *five normal forms*. You'll find a list of all five of these normal forms (which actually look sort of monstrous) in the sidebar at the end of this chapter, "The Five Abby-Normal Forms."

To normalize the `Pirate Supply Store` database, I made several changes to the design:

- ✔ I changed all the table names to plural. Before, I had a mixture of singular and plural names. (This is just a consistency issue.)
- ✔ I broke the `Orders` table into two tables: `Orders` and `Line Items`. When a customer places an order, one row is created in the `Orders` table for the entire order, and one row is created in the `Line Items` table for each product ordered. This allows the customer to order more than one product in a single order.
- ✔ I removed the `Category Name` field from the `Products` table because this data is contained in the `Categories` table.
- ✔ I removed the `Subtotal` column from the `Orders` table. The `Line Items` table contains an `Item Total` column, and the subtotal for an order can be calculated by adding up the item totals for each line item that belong to the order.
- ✔ I designated the `Item Total` column in the `Line Items` table as a calculated value. Rather than being stored in the table, this value is calculated by multiplying the quantity times the price for the row being retrieved.

- ✔ While interviewing the users, I discovered that some of the products are available from two or more vendors. Thus, the Products↔Vendors relationship isn't many-to-one, but many-to-many. As a result, I added a new table named `Product_Vendor` to implement this relationship. Each row in this table represents a vendor that supplies a particular product.

The resulting design now looks like this:

```
Products
  Product ID
  Name
  Category ID
  Description
  Price
  Image file name

Vendors
  Vendor ID
  Name
  Address
  City
  State
  Zip Code
  Phone Number
  E-mail

Categories
  Category ID
  Name

Customers
  E-mail
  Last Name
  First Name
  Address
  City
  State
  Zip Code
  Phone Number
  Credit Card Number

Orders
  Order number
  Date
  Customer E-mail
  Shipping
  Tax
  Total

Line Items
  Order number
  Product ID
```

```
Quantity
Price
Item Total

Product Vendor
Product ID
Vendor ID
```

Even though I did mention at the beginning of this section that five degrees of normality exist (It's a good thing these apply to databases and not to people, because some of us would be off the chart.), most database designers settle for the first through third normal forms. That's because the requirements of the fourth and fifth normal forms are a bit picky. As a result, I don't go into the fourth and fifth normal forms here. However, the following sections describe the first three normal forms.

### ***First normal form (1NF)***

A database is in 1NF when each table row is free of repeating data. For example, you might be tempted to design the `Orders` table like this:

```
Orders
  Order number
  Date
  Customer ID
  Product ID 1
  Quantity 1
  Price 1
  Product ID 2
  Quantity 2
  Price 2
  Product ID 3
  Quantity 3
  Price 3
  Subtotal
  Shipping
  Tax
  Total
```

This design allows the customer to purchase as many as three different products on a single order. But what if the customer wants to purchase four products? The solution is to create a separate table for the line items. The `Line Items` table uses a foreign key to relate each line item to an order.

### ***Second normal form (2NF)***

Second normal form applies only to tables that have composite keys — that is, a primary key that's made up of two or more table columns. When a table has a composite key, every column in the table must depend on the entire key, not just on part of the key, for the table to be in second normal form.

For example, consider the following table, in which the primary key is a combination of the Order Number and Product ID columns:

```
Line Items
  Order Number
  Product ID
  Name
  Quantity
  Price
```

This table breaks 2NF because the Name column depends solely on the Product ID, not on the combination of Order Number and Product ID. The solution is to remove the Name column from the Line Items table, and retrieve the product name from the Products table whenever it's required.

You might wonder whether the Price column also violates second normal form. The answer depends on the application's requirements. A product's price can change over time, but the price for a given order should be the price that was effective when the order was created. So in a way, the price *does* depend on the order number. Thus, including the Price column in the Line Items table doesn't violate 2NF.

### ***Third normal form (3NF)***

A table is in *third normal form* if every column in the table depends on the entire primary key, and none of the non-key columns depend on each other.

## **The Five Abby-Normal Forms**

No, this stuff didn't come from an abnormal brain in a jar; it only seems that way. In case you're interested (and just to point out how esoteric these things can be), here's a list of the original definitions of the five normal forms, in the original Greek, as formulated by C. J. Date in his classic book, *An Introduction to Database Systems* (Addison-Wesley, 1974):

**First Normal Form (1NF):** A relation *R* is in *first normal form* (1NF) if and only if all underlying domains contain atomic values only.

**Second Normal Form (2NF):** A relation *R* is in *second normal form* (2NF) if and only if it is in 1NF and every nonkey attribute is fully dependent on the primary key.

**Third Normal Form (3NF):** A relation *R* is in *third normal form* (3NF) if and only if it is in 2NF and every nonkey attribute is nontransitively dependent on the primary key.

**Fourth Normal Form (4NF):** A relation *R* is in *fourth normal form* (4NF) if and only if, whenever there exists an MVD in *R*, say  $A \twoheadrightarrow B$ , then all attributes of *R* are also functionally dependent on *A* (that is,  $A \twoheadrightarrow X$  for all attributes *X* of *R*).

(An MVD is a *multivalued dependence*.)

**Fifth Normal Form (5NF):** A relation *R* is in *fifth normal form* (5NF) — also called projection-join normal form (PJ/NF) — if and only if every join dependency in *R* is implied by the candidate keys of *R*.

Suppose the store gives a different discount percentage for each category of product, and the `Products` and `Categories` tables are designed like this:

```
Product
  Product ID
  Category ID
  Name
  Price
  Image file
  Discount Percent

Categories
  Category ID
  Name
```

Here, the `Discount Percent` column depends not on the `Product ID` column, but on the `Category ID` column. Thus the table is not in 3NF. To make it 3NF, you'd have to move the `Discount Percent` column to the `Categories` table.

## Step 5: Denormalize the database

What?! After all that fuss about normalizing the data, now I'm telling you to *de-normalize* it? Yes — sometimes. Many cases occur in which a database will operate more efficiently if you bend the normalization rules a bit. In particular, building a certain amount of redundancy into a database for performance reasons is often wise. Intentionally adding redundancy back into a database is called *denormalization* — and it's perfectly normal. (Groan.)

Here are some examples of denormalization you might consider for the `Pirate Supply Store` database:

- ✓ Restoring the `Subtotal` column to the `Orders` table so the program doesn't have to retrieve all the `Line Items` rows to calculate an order total.
- ✓ Adding a `Name` field to the `Line Items` table so the program doesn't have to retrieve rows from the `Products` table to display or print an order.
- ✓ Adding the customer's name and address to the `Orders` table so that the application doesn't have to access the `Customers` table to print or display an order.
- ✓ Adding the `Category Name` to the `Products` table so the application doesn't have to look it up in the `Categories` table each time.

In each case, deciding whether to denormalize the database should depend on a specific performance tradeoff — updating the redundant data in several places versus improving the access speed.

## *Step 6: Pick legal SQL names*

All through the data-design process, I use names descriptive enough that I can remember exactly what each table and column represents. However, most SQL dialects don't allow tables with names like `Line Items` or columns with names like `Product ID` or `Discount Percent`, because of the embedded spaces. At some point in the design, you'll have to assign the tables and columns actual names that SQL allows. When picking names, stick to these rules:

- ✓ No special characters, other than \$, #, and \_.
- ✓ No spaces.
- ✓ No more than 128 characters.

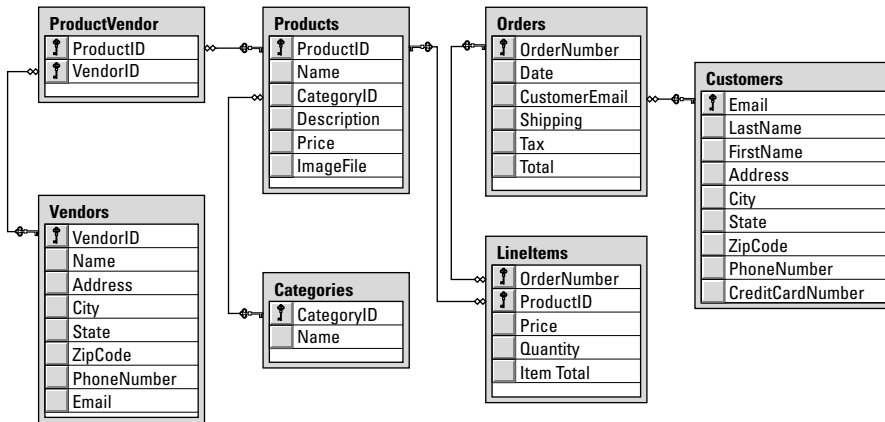
Shorter names are better, as long as the meaning is preserved. Although you can create names as long as 128 characters, I suggest you stick to names with 15 or fewer characters.

## *Step 7: Draw a picture*

Computer professionals love to draw pictures, possibly because it's more fun than real work, but mostly because (as they say) a picture is worth 1,024 words. So they often draw a special type of diagram — an *Entity-Relationship Diagram (ERD)* — when creating a data model. Figure 1-1 shows a typical ERD. Visual Studio 2005 includes a handy feature that automatically creates these diagrams for you.

The ERD shows each of the tables that make up a database and the relationships among the tables. Usually you see the tables as rectangles and the relationships as arrows. Sometimes, the columns within each table are listed in the rectangles; sometimes they aren't. Arrowheads are used to indicate one-to-one, one-to-many, many-to-one, and many-to-many relationships. Other notational doodads may be attached to the diagram, depending on which drawing school the database designers attended — and whether they're using UML (more about that shortly).

That's it for the steps needed to design relational databases. In the next section, I describe another important aspect of application design: designing the various objects that will make up the application.



**Figure 1-1:**  
A typical  
ERD.

## Designing Objects

The Microsoft .NET Framework is inherently object-oriented, so all ASP.NET applications are object-oriented applications. At minimum, each Web page that makes up the application is represented as two classes, as described by the Model-View-Controller (MVC) pattern:

- ✓ The *view* defines the appearance of the page.
- ✓ The *model-controller* represents the methods called to handle events, such as when the user clicks a button or selects an item from a drop-down list.

Many ASP.NET applications need additional classes to represent other types of objects. As a result, you might find yourself defining objects that represent business objects, or even some that implement business rules. Then you can write C# or VB code to implement those objects.

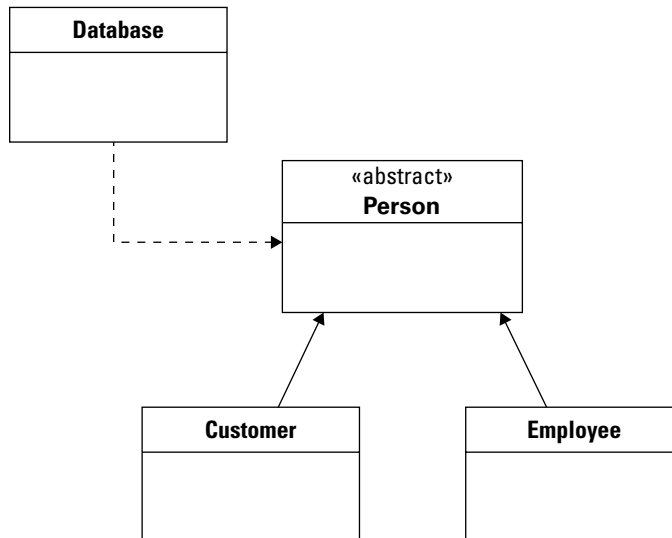
The task of designing these objects boils down to deciding what classes the application requires and what the public interface to each of those classes must be. If you plan your classes well, implementing the application is easy; plan your classes poorly, and you'll have a hard time getting your application to work.

## Diagramming Classes with UML

Since the beginning of computer programming, programmers have loved to create diagrams of their programs. Originally they drew *flowcharts*, graphic representations of a program's procedural logic (the steps it took to do its job).

Flowcharts were good at diagramming procedures, but they were way too detailed. When the Structured Programming craze hit in the 1970s, programmers started thinking about the overall structure of their programs. Before long, they switched from flowcharts to *structure charts*, which illustrate the organizational relationships among the modules of a program or system.

Now that object-oriented programming is the thing, programmers draw *class diagrams* to illustrate the relationships among the classes that make up an application. For example, the simple class diagram shown in Figure 1-2 shows a class diagram for a simple system that has four classes. The rectangles represent the classes themselves; the arrows represent relationships among classes.



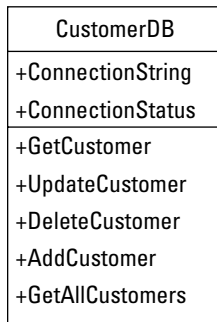
**Figure 1-2:**  
A simple  
class  
diagram.

You can draw class diagrams in many ways, but most programmers use a standard diagramming approach called *UML* (which stands for *Unified Modeling Language*) to keep theirs consistent. The class diagram in Figure 1-2 is a simple example of a UML diagram; they can get much more complicated.

The following sections describe the details of creating UML class diagrams. Note that these sections don't even come close to explaining all the features of UML. I include just the basics of creating UML class diagrams so that you can make some sense of UML diagrams when you see them, and so that you know how to draw simple class diagrams to help you design the class structure for your applications. If you're interested in digging deeper into UML, check out *UML 2 For Dummies* by Michael Jesse Chonoles and James A. Schardt (Wiley).

## Drawing classes

The basic element in a class diagram is a *class* — drawn as a rectangle in UML. At minimum, the rectangle must include the class name. However, you can subdivide the rectangle into two or three compartments that can contain additional information about the class, as shown in Figure 1-3.



**Figure 1-3:**  
A class.

The middle compartment of a class lists the class variables; the bottom compartment lists the class methods. You can precede the name of each variable or method with a *visibility indicator* — one of the symbols listed in Table 1-1 — although actual practice commonly omits the visibility indicator and lists only those fields or methods that have public visibility. (*Visibility* refers to whether or not a variable or method can be accessed from outside of the class.)

**Table 1-1** Visibility Indicators for Class Variables and Methods

<i>Indicator</i>	<i>Description</i>
+	Public
-	Private
#	Protected

If you want, you can include type information in your class diagrams — not only for variables, but for methods and parameters as well. A variable's type is indicated by adding a colon to the variable name and then adding the type, as follows:

```
connectionString: String
```

A method's return type is indicated in the same way:

```
getCustomer(): Customer
```

Parameters are specified within the parentheses; both the name and type are listed, as in this example:

```
getCustomer(custno: int): Customer
```

**Note:** The type and parameter information are often omitted from UML diagrams to keep them simple.



Interfaces are drawn pretty much the same way as classes, except the class name is preceded by the word *interface*, like this:

```
<<interface>>  
ProductDB
```

**Note:** The word *interface* is enclosed within a set of double-left and double-right arrows. These double arrows are often called *chevrons* and can be accessed in Microsoft Word via the Insert Symbol command.

## Drawing arrows

Besides rectangles to represent classes, class diagrams also include arrows that represent relationships among classes. UML uses various types of arrows; this section shows a basic set of them.

A solid line with a hollow, closed arrow at one end represents inheritance:



The arrow points to the base class.

A dashed line with a hollow, closed arrow at one end indicates that a class implements an interface:



The arrow points to the interface.

A solid line with an open arrow indicates an association:



An *association* simply indicates that two classes work together. It may be that one of the classes creates objects of the other class, or that one class requires an object of the other class to perform its work. Or perhaps instances of one class contain instances of the other class.

You can add a name to an association arrow to indicate its purpose. For example, if an association arrow indicates that instances of one class create objects of another class, you can place the word `Creates` next to the arrow.