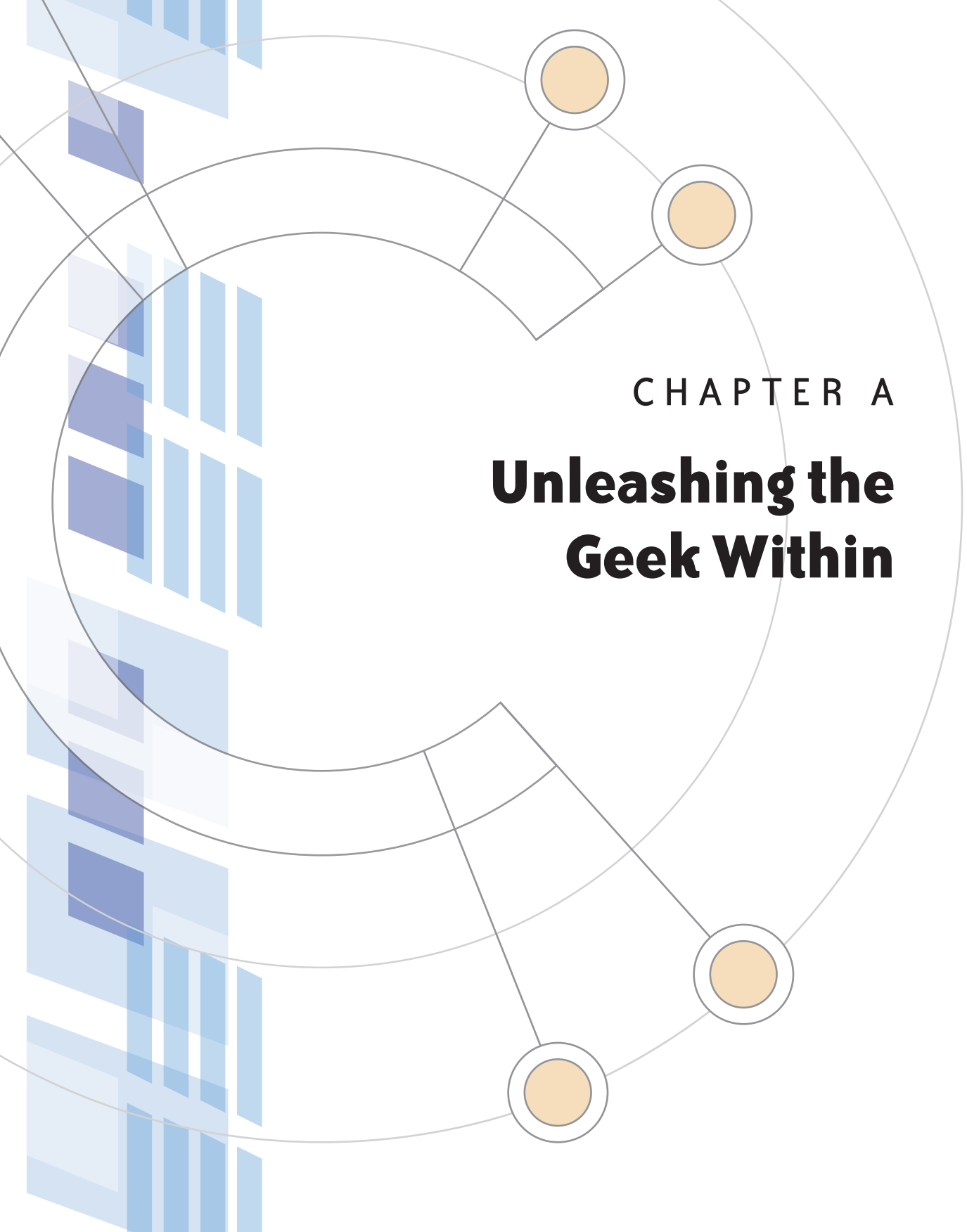


CHAPTER A

# Unleashing the Geek Within

Learning to write code is like learning a foreign language. At first, that language seems incomprehensible. Then you start building your vocabulary—the words that make up the language—and you start to learn the syntax—the order in which the nouns, verbs, and adjectives appear. Soon, you are stringing together sentences to convey your ideas and, eventually, communicating with ease.

Programming is no different. When you first see code, it looks like a bunch of hieroglyphics, but every programming language has its own vocabulary of words and symbols and its own equivalent of verbs, nouns, and adjectives (variables, functions, objects, and so on). In time, and with application, you can read code as easily as you read the words on this page.

Anyone can code. Coding is not a black art, but requires the same process as learning to write music, speak Spanish, or master any other technical skill. The same approach applies: learn the principles, deconstruct the work of others, and practice, practice, practice.

If you want to understand coding principles and start to learn how to program a Web site from front to back—from interface to database—*Codin' for the Web* is a good place to start. Hard-core programming books will certainly be more intelligible after you read this one.

Here, we'll briefly consider the creative process as it applies to Web site development and how the ideas for a site become code—code that defines the logic and processes that deliver the user experience of your design.

Some designers I know say "Oh, I'm right-brained, I could never program," with the deep-seated and, I think, erroneous view that programming is purely the left-brained application of logic and process and has nothing to do with the creativity that is required for design.

I have two comments in regard to this line of thinking.

First, if you really want to be an effective Web designer, code is the delivery mechanism of your vision, so you have to make the leap into understanding how code works if you are to reach your full potential in the world of the Web.

Second, programming is a creative endeavor in itself, as there are numerous ways to use code to solve any development problem, even though the end result is lines of text rather than a colorful comp.

That said, coding is, without doubt, an exercise in precision, patience, and perseverance, virtues that some designers may feel they lack. If a page layout you are designing has a graphic that is a pixel out of place, it probably won't affect the overall look, but one misplaced comma can bring your code to its knees. Partly because of this need for total accuracy, when your Web pages spring to life, populated by information that was pulled from a database on a distant server, you can't beat the sense of achievement.

So my advice is to strive to be more *whole brained* and realize that both user-experience-related tasks (information architecture and visual design) and programming tasks (database development and writing code) are creative and technical endeavors, and success in either is predicated on your technical skills.

# Form versus Function

To explore these broad ideas further, let's consider the tasks that designers and programmers undertake in the design process (**Table A.1**).

TABLE A.1 Design and Programming Tasks

| DESIGN-RELATED TASKS | PROGRAMMING-RELATED TASKS |
|---|---|
| Business requirements | Functional specifications |
| User experience | Process flow |
| Information architecture | Technical requirements |
| Aesthetics | Code development |
| Interface design | Testing |
| Usability | Quality assurance |

Web designers' tasks focus on creation of the overall design, or form, of the site: what features will the site offer, how will the user interact with it? With form comes nuance: some users may like one design comp, and others may prefer a second, and neither is inherently wrong. Your task is to develop a solution that works best for most users.

Programmers' tasks focus on creation of the site's functionality: what data is required to enable the user to interact with the site, and how will the code process the data obtained from this interaction?

Here's a simple example: Did the user submit a properly formed e-mail address? When the user submits an e-mail address, our code can test the following: that it starts with some text characters (but no symbols), which are followed by an @ character, then more text, and then a period followed by no more than four characters. For instance, charles@bbd.com would pass, but _someone. @@hello.world would not. We can create the test using a code structure called a regular expression, and if the address passes that test, it's formed correctly as an e-mail address. Here, as with some aspects of coding, there is no nuance—the test will simply respond TRUE or FALSE.

# Use Cases

Use cases can be used to describe the back-and-forth interactions between the user and the code for each of the site's functions. Use cases are a great bridge between the conceptual ideas and the actual code. The basic principle of use cases is to show how actors—your Web site's various users, in the scope of this discussion—interact with systems—the various functional parts of your Web site. By mapping these interactions for every task you want your site visitors to undertake, you not only think through exactly how these interactions will occur, but also lay the foundations for planning how the associated code will be written.

*If you want to learn more about use cases and how to write them, read* Writing Effective Use Cases *by Alistair Cockburn, published by Addison-Wesley.*

The Unified Modeling Language, known as UML, is a quite complex and formalized tool for creating use cases, and it can help you break down and convey to others the interactions of very complex systems.

To illustrate how effective use cases can be, let's look at a simple one-actor, one-system example with a basic back-and-forth set of steps. To use a non-Web analogy, the idea might be "Let's build a machine with a screen and buttons where users can deposit checks

and withdraw money!" but of course code supports the operation of an ATM.

Sketching out use cases is a great way to prepare for a meeting with a programming team. **Figure A.1** shows a simple use case for withdrawing cash from an ATM.

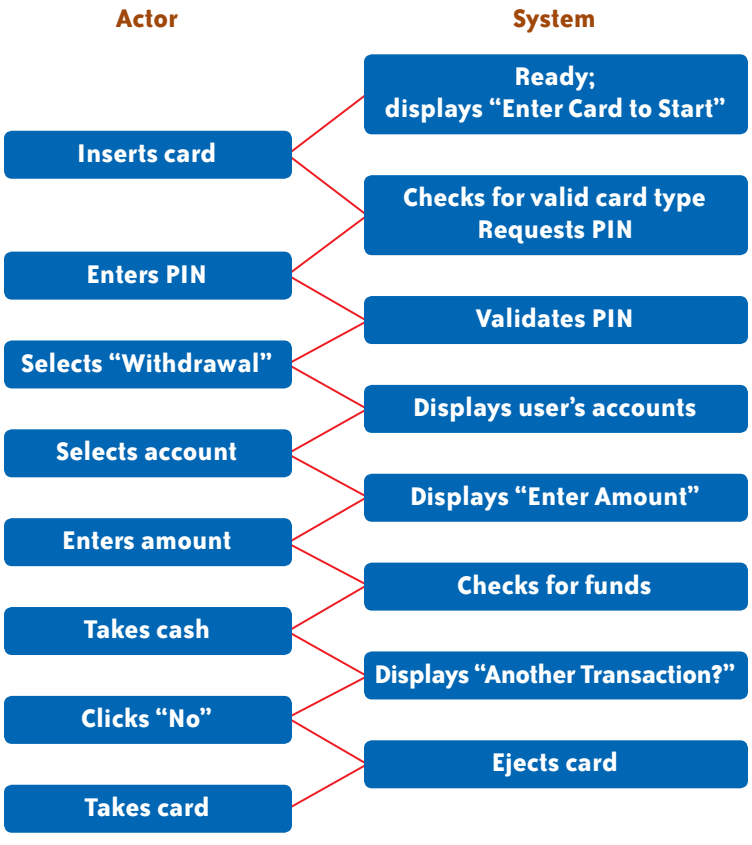



FIGURE A.1 A simple use case sketch: withdrawing cash from an ATM.

*For simplicity, Figure A.1 does not show edge cases—the things that usually don't happen, but sometimes do. In this example, edge cases would show what the system does if the card is expired or if the user has insufficient funds in the account.*

Sketching a use case for each process and reviewing the use cases with clients and other members of your production team early in the production cycle can save hours of time because everyone will understand in detail how each process will flow before coding begins.

Use cases are of huge value to programmers in determining what data is needed within the system and from the user, and what processes must act on that data, to support each of the user-system transactions. Programmers also look for pieces of functionality that occur repeatedly in the use cases (such as a testing of whether the user has the right login credentials to view a particular page)

and engineer these as stand-alone blocks of code called *functions* that can be accessed by many other pieces of the code, and therefore have to be written only once. Chapter 1 of *Codin' for the Web* explores functions in depth.

In short, the programmer's task is to break down the required functionality of the site into code components that work together to accept inputs (data generated by user actions) and provide outputs (data returned to the user from the system) so that the user can successfully complete the tasks that the site supports.

# Dynamic Web Sites

Many Web sites consist of *static* pages—pages that are unchanging or, to use a programmer's term, hard-coded; the code, and therefore its related page's appearance, is exactly the same each time it loads into the user's browser. These static-page sites limit the user to basic tasks like clicking links to navigate between pages and submitting simple forms. There is no way for the site to provide any significant degree of response to users' actions.

A Web site that is capable of accepting and then responding to user-supplied data is called a *dynamic* site. While *dynamic* is an overused word that has come to mean exciting or energetic, here we refer to the original meaning of the word: "changing over time." Dynamic sites are characterized by pages that can serve up variations of the site's available content based on user input. Perhaps the best-known and most illustrative example of a dynamic site is Amazon.com, which delivers customized pages based on users' searches, personalization, and previous browsing and shopping activity (**Figure A.2**).

Personalization like this requires that the server delivering the pages track each user's computer from visit to visit. This tracking is achieved through the use of *cookies*. A cookie is a small data file that a site stores on the user's computer and is typically created when you first visit a site or when you enter some personal information—when you make your first purchase, for example.

In the case of Amazon, the cookie contains a user ID that references your information in the Amazon databases, so Amazon can read the cookie's ID back to its servers each time you request a page and then instantly look up information such as your name and previous page views and purchases.

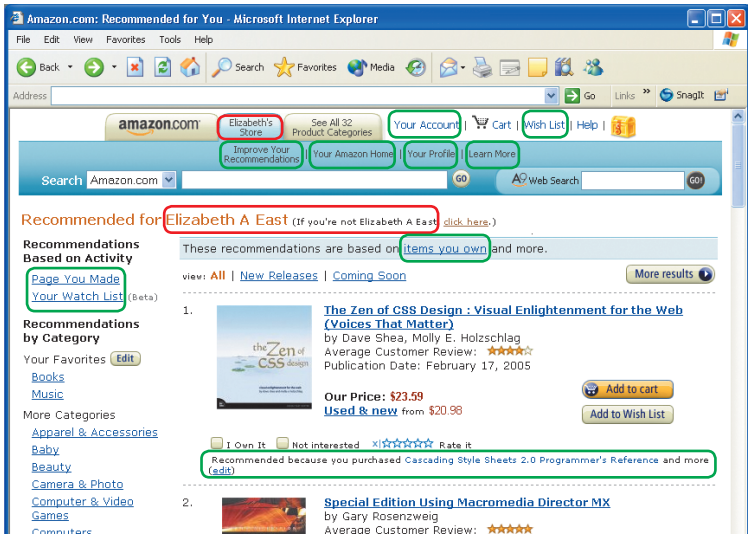



FIGURE A.2 A customized Amazon.com page. Note the numerous locations on this page that are customized (highlighted in red) for this visitor or that offer links to customized pages (highlighted in green).

The business rules Amazon has programmed into its site then use this data to cause the site to serve up features such as a page with your name, a link to recommendations based on your previous purchases, and perhaps an offer for some other items that might appeal to you—all within a few seconds of your typing `www.amazon.com` and pressing the Enter key on your keyboard.

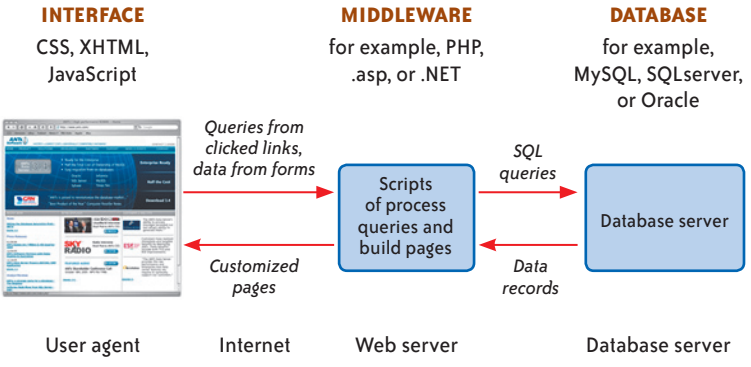
It's a bit like walking into your favorite store and being greeted by a salesperson who immediately says, "Welcome back, Kate! How did you like that flat-screen TV we sold you last week—check out this great DVR that works perfectly with it." Clearly, this kind of personalization is effective, and Amazon has built its entire site, and therefore its entire business, around personalization of the site for every visitor.

Virtually all dynamic sites are built on a simple principle that is explored in detail through the course of *Codin' for the Web*: three-tier architecture.

# Three-Tier Architecture

The three tiers of three-tier architecture are the interface, middleware, and database (**Figure A.3**).

FIGURE A.3 Three-tier




The database and middleware reside on Web servers at the Web site's hosting facility, be it an ISP or your company's data center, and the interface, while served from the Web server, is viewed in the user's browser.

We'll start with a quick overview of each tier; later in this chapter, we will examine the tiers and the coding languages and techniques associated with them in greater depth.

## Interface

The interface is the part of the site that the users sees and interacts with in the Web browser. The code that describes the interface is run (or *executed*) by the browser and enables the user to view a page's content and, through the use of links and forms, interact with your site. When you create the user interface, you will usually use three programming languages:

- **XHTML** (a syntactically stricter HTML variant) defines the page's structure.
- **CSS** defines the page's presentation.
- **JavaScript** defines the page's behavior.

*The interface can also deliver various media files such as QuickTime movies and Adobe Flash animations. Playback of these kinds of media are enabled by plug-ins, which is files that extend the browser's capabilities.*

## Middleware

Middleware is a generic term given to the software on the Web server that receives the user inputs and serves up the dynamic Web pages. You can think of it as sitting between the user interface and the database, controlling the processing of user queries and the transactions between the user interface and the database.

Middleware can further be divided into two parts: the platform and the scripts (pieces of code) it runs. The platform, as its name suggests, provides the basis for development. It defines the language or languages that may be used in the scripts and acts as the engine that processes the scripts. Common examples of middleware platforms are PHP (open source) and .NET, pronounced dot-net (Microsoft).

Middleware is where the real processing work of your Web site is done and where the greatest programming effort is required. It's the middleware that enables your pages to be dynamic.

For now, think of middleware as a system that does the following:

- Receives requests from the user interface.
- Makes calculations, comparisons, and database queries using scripts (pieces of code) written to handle each type of request.
- Writes the results to a Web page generated from a page template (we'll discuss page templates in more detail later in this chapter).
- Serves the finished Web page back to the browser.

There are also specialized applications that serve the role of middleware such as content management systems (CMSs) and customer relationship management (CRM) systems. Sometimes multiple middleware products such as these have to work together and jointly provide the data for the page that is delivered to the user.

*As suggested by Figure A.3, the middleware and database typically run on separate servers. This approach provides improved performance for the site, allows it to be scaled more easily to handle more traffic through the addition of more Web or database servers, and provides an additional level of security for the data, as users do not communicate directly with the database server—only the Web server can do that.*

## Database

The database holds the information relating to your Web site. It may contain users' e-mail and password information, the details of products in an e-commerce store, or the scores of the games users play on your site. Whenever your site collects or generates a piece of data that will be needed in the future, that data goes into the database.

Databases are made up of numerous related *tables*. These tables are each a grid of rows and columns, much like a Microsoft Excel spreadsheet, containing related pieces of data. The development of good database structure is essential to a site's performance (how quickly it responds to user requests) and its scalability (a measure of the ease of expanding the initial structure in the future).

# Anatomy of the Interface

The primary purposes of the user interface are to link the user to the content of your Web site and to enable the user to provide inputs to the system.

Our focus here is not on the aesthetics of design, so we will not get into a discussion of how to design interfaces here. Instead, we'll take a development-focused look at the user interface, because it's important to understand from a coding perspective the interface's components and underlying structure.

## Interface Components

Everything in a user interface can be considered to be part of one or more of five types of components (**Figure A.4**):

- **Content** is the generic term for the information that a site delivers: text, images, video, animations, sound, and downloadable files.
- **Orientation** describes the components that help users understand where they are in relation to the rest of the site: section titles above page titles, highlighted navigation links that indicate the current page or section, and breadcrumbs that list sequentially the pages that are traveled from the home page to the current page are all aids to orientation.

- **Directions** provide as-needed instruction to the user. Key examples are error messages and feedback. Directions clearly explain both successes and errors in tasks and offer options for what to do next: "Transaction complete! Shop more or check out?" or "Our database is busy right now—please wait a moment and try again." Good task feedback = easy-to-use site = happy users. Directions are a type of content, and you must develop a process to create them.
- **Inputs** are the means by which users can inject data and content into the system. Forms, image, and file uploading capabilities are examples of inputs.
- **Navigation** describes the interface components that enable the user to move through the content: links, clickable images, and menus that act as links are the primary navigation approaches.

It is purely coincidental that you can remember these five components by the acronym CODIN.

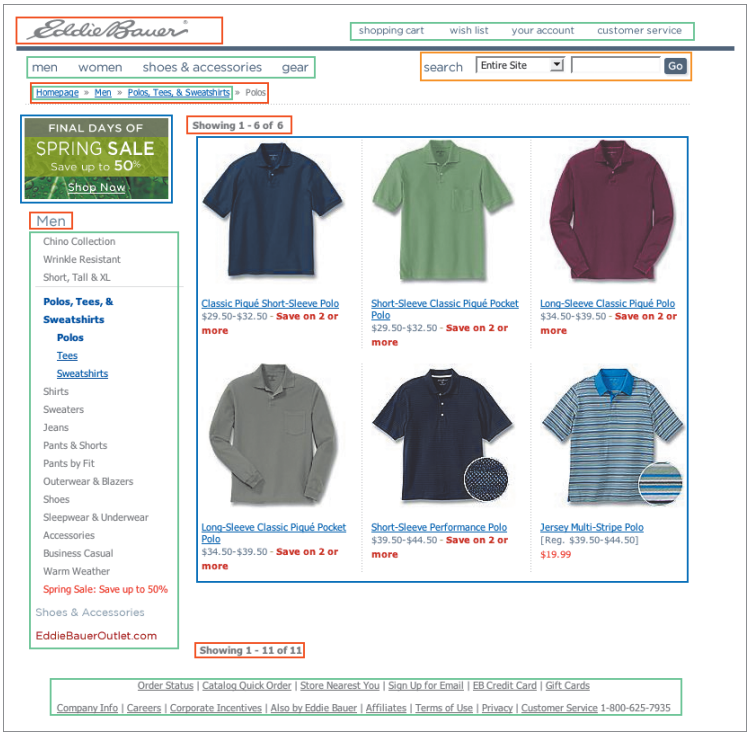



FIGURE A.4 The interface components on a page of the Eddie Bauer (eddiebauer.com) Web site: content (blue), orientation (red), navigation (green), and inputs (orange). Note that Eddie Bauer keeps the site clean by saving directions for when they really add value—for instance, the user does not need to be told "Click a link to view a product."

A key goal of user-focused programming (that is, programming with the user's needs always in mind) is to always provide relevant navigation and orientation context along with the content.

## Structure of the Interface: Hierarchical Page Structure

XHTML and CSS are not programming languages in the true sense of the word, as they cannot themselves process data. However, it is very important to understand the page structure that XHTML provides to a page and how CSS can modify the presentation of that structure, because the code we write will output XHTML to the user's browser, and CSS will determine how that XHTML will be presented.

XHTML provides structure to content. An XHTML page consists of elements that are defined through the use of XHTML tags, in either an enclosing form such as this:

```
<p>This is text enclosed in paragraph tags</p>
```

or a nonenclosing form, such as this:

```
<img src="images/my_dog" alt="My dog" />
```

If we look at the underlying structure of an XHTML page (xhtml_template.html in the Chapter B files at the *Codin'* Web site is a barebones example of such a file), we see something like this:

```
<html>
 <head>
   <title>My demo page</title>
 </head>
 <body>
  <div id="navigation">
   <ul>
      <li><a href="about_xhtml">About XHTML</a></li>
      <li><a href="about_css">About CSS</a></li>
   </ul>
 </div>
 <div id="content_area">
    <h1>About page structure</h1>
```

```
<p>It's important to realize that there is a hierarchy of
tags in every Web page, because this hierarchy is used
by CSS and JavaScript to locate and modify the elements'
appearance or content.</p>

    <a>Learn more</a>

  </div>

 </body>

</html>
```

The markup consists of two main sections: the `<head>`, which contains elements that help the browser understand how to display the page and provide metadata (data about the page's data, such as its title), and the `<body>`, which contains the actual content that the user sees. It doesn't really matter in this example what this page actually looks like in the browser; what you should understand here is that we can also look at the body of this page as shown in **Figure A.5**.

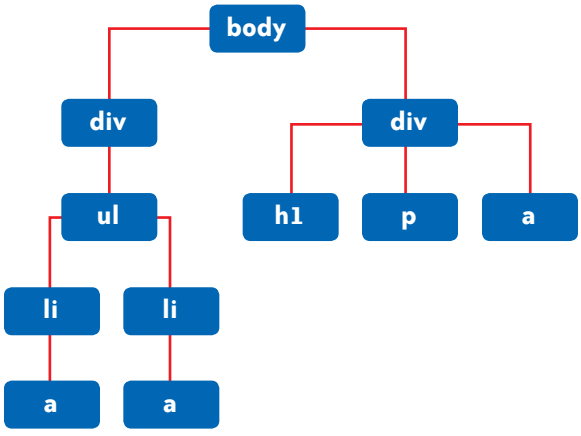



FIGURE A.5 Structural hierarchy diagram of the XHTML code.

There is a tree-like structure to the markup, with tags nested inside one another. For example, we could say that the unordered list (`ul` tag) is a child of the content area division (`div` tag), and the parent of the list items (`li` tags). The links (`a` tags) within the list items are the unordered list's descendants (in this case, grandchildren), and the `body` tag is its ancestor (in this case, its grandparent).

This hierarchical structure enables us to target CSS and JavaScript at specific tags. For example, the CSS rule `div#content_area a {color:red;}` would color the link in the content `div` area red, but the links in the navigation `div` area would be unaffected by this rule, because the rule states that the link must be a descendant of the content area `div` tag.

## Interface Summary

The user interface is the user's window into your Web site. The interface components support five primary functions: the display of content, navigation to other pages and sites, orientation to help the viewer understand the relationship of the current page to others, directions to keep the user on task, and input capabilities to allow the user to inject data into the system.

The interface is structured with XHTML markup of nested elements tags, and this structure allows the elements and their content to be targeted and modified by CSS and JavaScript.

# Middleware

Middleware is the brain of your Web site. It is the software that resides on your Web server that receives the data sent from the user interface, and, using the code you write, checks that the data is in a valid format, processes it, and, based on that processing, builds an appropriate XHTML page and sends it back to the browser.

Each task that you want your site to perform requires you to write a code script to process the data associated with that task.

For example, if we want a user to be able to sign up for our monthly e-mail newsletter, the script that will process the sign-up form needs to ensure that the e-mail address supplied is properly formatted, write the person's name and e-mail address in the database for future use, and then immediately e-mail the current edition of the newsletter to that person. Finally, the script builds a page thanking the user by name for signing up and returns the page to the user's Web browser.

*The Web site www.hotscripts.com is an excellent repository of scripts for all platforms.*

For the middleware scripts for your site, you can use something that is already written, or you can write the scripts yourself.

The advantage of writing the scripts yourself is that you can write whatever functionality you want, the way you want to write it. The disadvantage is obvious: it's a lot more work.

There are many off-the-shelf middleware scripts that may provide the functionality you need. For example, if you want to set up an e-commerce store, you might choose osCommerce (PHP based) or Miva Merchant (.NET or Unix-based). These products have all the basic online store functionality built in, such as features for setting

up customers, organizing products into categories, and providing a virtual shopping cart, and therefore require minimal custom coding. Of course, the downside is that you must accept the constraints that a preprogrammed solution imposes.

There is a middle ground between totally preprogrammed and do-it-yourself: open source code. With open source code, you get, as the name implies, the source code and are allowed to modify it as you see fit, so you can build on others' work while doing all the customization you want. Note, though, when using open source code in your project, that, depending on the code's license and your means of distributing your code, you too may be required to share your changes and additions with others.

If your site is going to provide a large library of content, you might select off-the-shelf middleware focused on content management. Numerous products—from the simple but effective CityDesk from FogCreek; through business-strength products from developers such as Ektron, RedDot, and PaperThin; all the way up to massive corporate solutions such as Documentum and Interwoven—are available.

My advice is, if you are building something that might have been built before, such as a store, a forum, or a blog, research the off-the-shelf solutions before embarking on the more demanding approach of building from scratch.

Many sites combine various middleware products to achieve the desired functionality. You can fairly confidently guess that the Dell site (www.dell.com) is built with a combination of e-commerce and content management middleware products, as the site provides sophisticated shopping options, where you can select exactly how you want your new computer configured, and also vast libraries of product information and technical documentation for you to browse as you make your purchasing decisions.

When these kinds of solutions are being developed, things can get very complex. Each piece of middleware used must be able to communicate with the others, and each may have its own database. There can be some thorny issues related to what data is stored where. How such databases exchange information can become a large engineering initiative in its own right.

Regardless of which of approach you choose, you will find two common concepts in virtually every middleware system: templates and include files.

### A Note to the Corporate Designer

Be aware when you are considering buying and integrating existing packages that there are some important business issues to examine, such as these: What platform (operating system) does each application run on? What language you have to use to extend the applications' functionality? Does each application have an application programming interface (API) to enable the establishment of real-time communication and data exchange between the applications and other systems in the enterprise? If not, is there at least a way to schedule data dumps to files that can be read by other components of the system, so that close-to-real-time (for instance, hourly or daily) synchronization of the data can be set up? The list goes on.

## The Concept of Templates

In the world of dynamic Web sites, each page delivered to the user can be unique. Let's return to the example of Amazon. It's unlikely that anyone else has looked at exactly the same selection of Amazon pages you have looked at in the past or bought exactly the same combination of products, so the items on the page of recommendations you see when you log into Amazon today may never have been presented to you or anyone else in quite that combination before, and after you make your next purchase or view another page, that precise combination may never be seen again.

However, clearly, there are significant similarities in every Amazon "Recommendations" page. Many elements, such as the top of the page, are essentially the same every time, the number of products in each row is the same, and the overall formatting of the page's text and layout is the same. Even the position and typeface of the "Welcome *(your name here)*, we have recommendations for you" line is the same—only the name changes when you, instead of someone else, view the page. In short, only the information that relates specifically to you is changed.

This customization of what is essentially the same page every time is achieved by the use of page *templates*. Think of a template as a recipe from which a page is cooked up. A template contains both XHTML code for the static elements of the page, which never change, and middleware code elements, which the middleware processes and replaces with the dynamic elements that change from user to user.

Here's a simple example of a template in action. Imagine that the user has typed the URL of your home page. A cookie on the user's

computer from when the user previously registered with the site enables the middleware to pull the user's name from the database and place it in a variable (a location in memory) called `$user_name`. The home page template's XHTML contains the following code:

```
<h2>Welcome back, <?php $user_name ?>, it's good to see you
again!</h2>
```

As you might guess from the code, this is an example from a site that uses PHP as its middleware. Instead of directly serving up the template, PHP creates a copy of it line by line, and as it does so, it looks for PHP tags in the XHTML. Whenever it encounters one, it processes it and replaces the PHP tag with the result of the process.

In this case, PHP looks up the variable `$user_name` and replaces the PHP tag with the string of text it finds there. Once PHP has processed the whole page, it serves up the customized copy to the user's browser, and the line of code now might read like this:

```
<h2>Welcome back, Suzie Q, it's good to see you again!</h2>
```

Even a massive site like Amazon may have only a few page templates, but, in the manner shown here, the middleware can create an infinite number of page variations from them—in fact, the fewer page templates your site has, the more consistent the overall look and feel of the site will appear to be.

### An Approach to Templates: Use Only as Many as You Really Need

In a typical production team, the design group is responsible for delivering the page templates, written in XHTML and CSS, to the programmers so they can incorporate these templates into the site. The programmers replace each piece of the designers' placeholder content with the appropriate middleware tag, as just described.

At the start of the site's development, the design team might first identify just three page templates—the Home Page template, the Main Section pages template for the pages that link directly to the home page, and a general page template that is used for all the other pages.

Each time a new type of content is identified, there is discussion around whether this content requires a new template to display it effectively, or whether it can be presented within the framework of one of the existing templates. The addition of another template is usually vetoed unless someone can make a valid case for it—more templates mean more code to maintain and yet another variation in the site's look and feel.

## The Concept of include Files

Imagine a site with static pages—pages are hard-coded in XHTML and have no dynamic elements. Each of these pages almost certainly has code that repeats on every page: for example, the page header with the logo and company name, or the general navigation links in a sidebar. Anyone who has had to make changes to these areas of such pages has experienced the tedium of opening page after page to make the same alteration, and also knows too well the hold-your-breath-and-pray moment that accompanies clicking the Replace All button when performing a global search-and-replace of large chunks of code across dozens of pages.

Middleware enables you to break out each chunk of code that is common to multiple pages and move it into a file known as an include. Then you simply add an `include` tag to the page in its place, using a line of code like this:

```
<?php

include("includes/main_nav.inc.php");

?>
```

In the same way as the `$user_name` tag was replaced with a user name in the code we discussed earlier, so an `include` tag in a page is replaced with the appropriate include file, which can contain many lines of code. Now the navigation element that appears on every page of our site is stored and maintained within a single file that every page references. **Figure A.6** illustrates the include file concept. You can see it in action in Chapter 6 of *Codin' for the Web*.

The include file concept opens a very efficient process for creating pages. If a template can be thought of as an abstraction of a Web page, then an include file can be thought of as an abstraction of a template.

We have seen that a template allows us to define dynamic areas within our pages. Include files allow us to define common areas within our templates. Pages are built from templates, and templates can be built, at least in part, from include files. For example, you may have five templates to present five different layouts or types of content, but each of those templates may have the same main navigation links and the same footer across the bottom of the page. Such common elements can each be stored in an include file and referenced by every template using an `include` tag (**Figure A.7**).

FIGURE A.6 By breaking out common elements of a site's pages into include files, these elements can be shared by multiple pages.

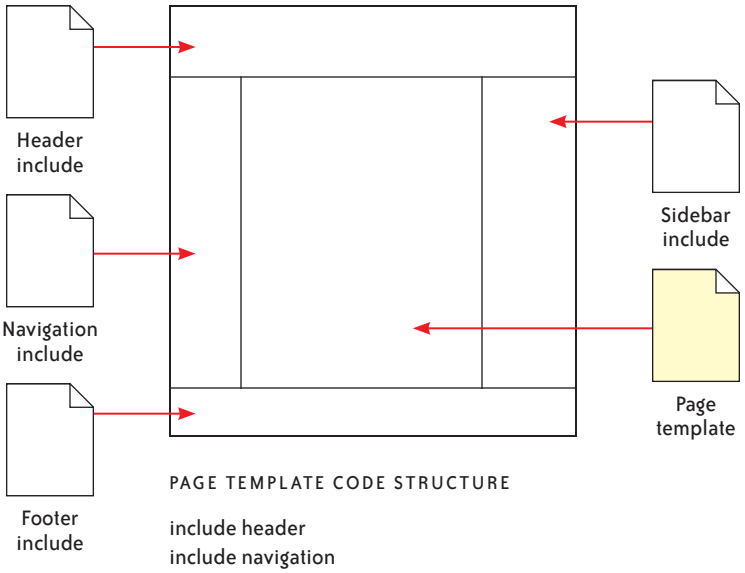
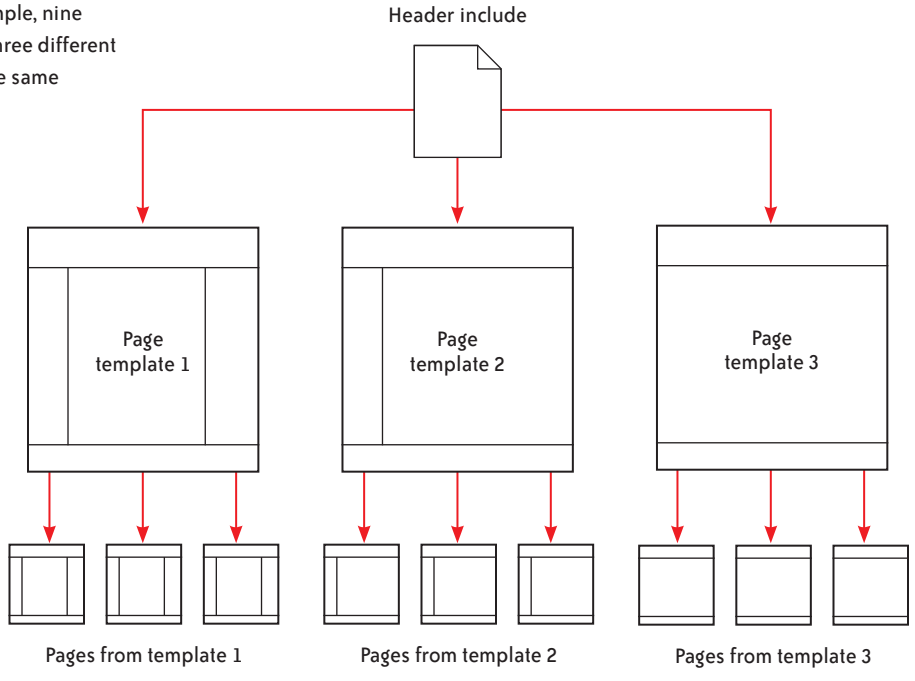



FIGURE A.7 In this example, nine pages, generated from three different templates, all contain the same header include file.



Such an approach to designing the structure of your site can yield huge efficiencies in its development and updating. This kind of modular thinking separates the excellent Web developers from the merely competent ones and is essential when developing large corporate sites like those of Amazon and Dell.

### Middleware Summary

Middleware can come as code in general-purpose languages such as PHP and C#, or it can come as specialized applications such as customer relationship management, content management, and e-commerce applications.

Templates containing variables and other code structures enable pages to be populated with content, information from databases, and the inputs provided by the user, based on the system's business rules, which are expressed in the site's middleware code.

Include files allow code blocks that are shared by the templates to be added to a page at the time that it is generated from a template.

## Database

The database is the storehouse for all the data associated with your Web site.

Various databases are available, and you usually decide which one to use based on the operating system you are using for your Web server. If you are building on the Windows operating system, you will probably use the Microsoft Internet Information Services (IIS) Web server, .NET middleware, and Microsoft SQL Server database. Oracle is a powerful database alternative for large enterprise applications running on Windows.

For situations with high contention (lots of users attempting to access the database at once, such as an instant messaging or airline reservations system), you might use a new, powerful contender in the database arena, the ANTs Data Server (which also works with Linux).

If you are using an open source operating system, such as Linux or Unix, you will probably use PHP middleware and the mySQL database, as PHP has been optimized to work with it. PostgreSQL is another database that works excellently with PHP and provides a number of features that mySQL doesn't offer.

## Structure of a Database

Databases are made up of tables. If you have ever used an Excel spreadsheet, you will be familiar with the grid-based method of organizing data used by database tables. The similarity ends there, though; database tables have their own data organization rules and relate to one another in ways that are very different from spreadsheets.

A Web site database usually needs to store many different kinds of data. For example, an e-commerce site needs to store customer information (name, mailing address, e-mail address, and so on), product information (product name, description, SKU, price, and so on), and the transactions that customers have made in the store.

A database is queried (asked to perform a task) using Structured Query Language (SQL), a language developed especially for this purpose.

SQL is used for two processes: *data definition*—creating, deleting, and editing the structure of tables—and *data manipulation*—adding, modifying, deleting, and retrieving the data in the tables. Data definition happens when the structure of the database is first built; data manipulation happens every time the middleware accesses the database.

Database design focuses on logically dividing the data to be managed into tables. By dividing the data into multiple tables, we create a *relational database*, where relationships between the different types of data can be defined. This approach greatly speeds access to the data, simplifies management of that data, and most important, avoids redundant (repeated) data. Redundant data is the enemy of good database organization.

*In reality, we would need lots more information, such as user names and passwords and product manufacturers and their information, but even this greatly simplified example has plenty of the kinds of problems that result from a nonrelational approach.*

Imagine an e-commerce store that sells software online; for each customer, we need the person's name and e-mail address so we can send notifications, and for each product, we need the product name, price, and current inventory. Let's start by doing things the wrong way: by shoving all this data into one big table.

With only one table to hold all the store's information, if we want to track which customers bought which products, we may end up with something like **Figure A.8**.

| first_nam | last_name | email | registered_date | product_nam | category | version | platform | price | inventory |
|---|---|---|---|---|---|---|---|---|---|
| John | Jones | jjones@abc.cor | 4/4/06 | Write-a Lot | word processor | 1.2 | Windows | 39.99 | 91 |
| Suzie | Q | sq@bcd.com | 4/4/06 | Write-a Lot | word processor | 1.2 | Windows | 39.99 | 90 |
| John | Jones | jjones@abc.cor | 4/4/06 | MasterArtMak | graphic design | 4 | Mac | 109 | 17 |
| Bill | Williams | bill@dogbreath. | 4/5/06 | Write-a Lot | word processor | 1.2 | Windows | 39.99 | 89 |
| Annie | Smith | annie@mysite.c | 4/5/06 | MusicMaster | music compositior | 2.3 | Windows | 399 | 9 |
| Suzie | Q | sq@bcd.com | 4/4/06 | MusicMaster | music compositior | 2.3 | Windows | 399 | 8 |

FIGURE A.8 An example of how *not* to organize data in a database table. Note the duplication of many pieces of information.

In Figure A.8, as in any database table, each horizontal row is a *record* (in this case, of a transaction), and each vertical column represents a *data type* associated with each of the records. At the intersection of the row and the column is the *value*—the piece of data of that data type associated with that record.

In Figure A.8, each record has the user's information first and the product information next. The first thing you will notice is how much information is repeated or not needed in this example of just six transactions: all of Suzie's and John's information appears twice, and information on two of the products appears twice. Also, we don't need to keep track of the inventory like this; all we need to know is the current status of a product, not how many we had in stock before the previous sale. And this example uses only four customers and five products; extrapolate this redundant data to thousands of customers and thousands of sales, and you can see that even updating someone's e-mail address might become a massive job. Also, what happens if one instance of the e-mail address gets changed but not another? It's a recipe for confusion and unhappy customers.

The solution to this problem is *normalization*: the formalized process of eradicating redundant data from a database. With no redundant data, if a change has to be made, it needs to be made in only one place. Let's normalize the data in Figure A.8.

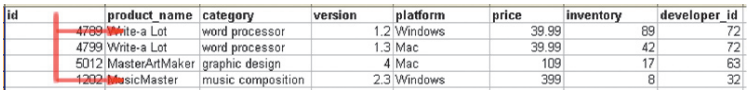
We really have two sets of data records here—the customer information and the product information—so let's split them into two separate tables. After that, we can look at the actual transactions—the sales of products to customers—which will need to be represented by some kind of relationship between the two tables.

As we create our Customers and Products tables, we will give each record a unique identifying number (ID), known in database-speak as a *primary* key. The primary key can be any field in the table that is guaranteed to never repeat. It is also possible to assign an auto-incrementing number to each row if no natural primary key exists. This is the case with our tables, as shown in **Figures A.9** and **A.10**.

FIGURE A.9 The Customers

| id | first_nam | last_name | email | registered_date |
|---|---|---|---|---|
| 103 | John | Jones | jjones@abc.com | 4/4/06 |
| 105 | Suzie | Q | sq@ucd.com | 4/4/06 |
| 108 | Annie | Smith | annie@mysite.com | 4/5/06 |
| 109 | Bill | Williams | bill@dogbreath.com | 4/5/06 |

FIGURE A.10 The Products



| id | product_name | category | version | platform | price | inventory | developer_id |
|---|---|---|---|---|---|---|---|
| 4789 | Write-a Lot | word processor | 1.2 | Windows | 39.99 | 89 | 72 |
| 4799 | Write-a Lot | word processor | 1.3 | Mac | 39.99 | 42 | 72 |
| 5012 | MasterArtMaker | graphic design | 4 | Mac | 109 | 17 | 63 |
| 1202 | MusicMaster | music composition | 2.3 | Windows | 399 | 8 | 32 |

*As you will see when we look at SQL in more detail, you can set up database tables so the database knows that a particular column contains primary keys, and it will automatically allocate a new and unique primary key to each record as it is created.*

The first column of each table contains the primary keys, which provide unique references to each record in the table. If we delete a record (for example, if a product is no longer available and has to be removed), we will *never* use its primary key again. One reason why we must never reuse a primary key is that in our archives of previous sales, that key will still be associated with the discontinued product, so if we reallocate that primary key to a new product, a future audit of sales could be thrown into confusion.

We are now in much better shape; if a customer's e-mail address changes, we need to change it in only one place. If we want to serve up a list of all our products, our middleware simply has to grab every value in the product_name column of our Products table.

Customers come to our store to buy products, and it's our business to keep track of those sales, so let's now create a table of the transactions: the individual sales of products to customers. To avoid redundancy, we can't put any actual customer or product information in our Transactions table, or we will return to the problems we had in the earlier table with all its redundant data. Instead, we just reference each of the primary keys of the customers and product records, as shown in **Figure A.1A**.

FIGURE A.11 The Transactions table.

| id | customer | product | sale_date | shipped_date |
|---|---|---|---|---|
| 1 | 103 | 4789 | 4/4/2006 | 4/5/2006 |
| 2 | 105 | 4789 | 4/4/2006 | 4/5/2006 |
| 3 | 103 | 5012 | 4/5/2006 | 4/6/2006 |
| 4 | 109 | 4789 | 4/5/2006 | 4/6/2006 |
| 5 | 108 | 1202 | 4/6/2006 | 4/6/2006 |
| 6 | 105 | 1202 | 4/6/2006 | 4/6/2006 |

Note that the first column of this table has a primary key for each transaction. The second and third columns list primary keys from the Customers and Products tables respectively. When we use the primary key of one table as a value in another table like this, it is referred to as a *foreign key*. We can think of each foreign key as a pointer to a record in another table, which indeed it is.

In our new Transactions table, record 1 shows that John Jones (customer 103) bought Write-a-Lot version A.2 for Windows (product 4789) on April 4, and it was shipped the following day. Take a close look at these three tables, and you will see that every piece of information is unique or is a foreign key that points to unique data. Note particularly that the non-foreign-key data in this table is unique to the transaction itself and not to any other data type.

## Structured Query Language: SQL

As mentioned earlier, we access the information in our carefully organized database tables using SQL. We can program our middleware to request data by sending SQL queries to the database, and the requested data will be located and put into variables (data containers in memory) that the middleware can then read.

We are going to explore how to write SQL queries in detail later, but here are a couple of examples to whet your appetite.

Let's say, for example, that Suzie Q has used her e-mail address to log in to our site. We want to greet her onscreen, so we need her first name. Here's what we need to do (or have SQL do for us): Go to the Customers table, search down the email column for Suzie's e-mail address, and when we find it, return the corresponding first_name value from that record row (**Figure A.12**).

FIGURE A.12 A simple lookup of a name based on an e-mail address.

| id | first_nam | last_name | email | registered_date |
|---|---|---|---|---|
| 103 | John | Jones | jjones@abc.com | 4/4/06 |
| 105 | Suzie | Q | sq@bcd.com | 4/4/06 |
| 108 | Annie | Smith | annie@mysite.com | 4/5/06 |
| 109 | Bill | Williams | bill@dogbreath.com | 4/5/06 |

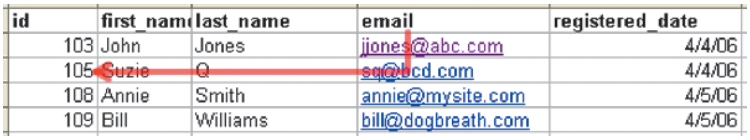
Here's the SQL query that will accomplish this task:

```
SELECT first_name from Customers WHERE e-mail LIKE
"sq@bcd.com"
```

As you can see, SQL is quite easy to read—at least for the more simpler queries. SQL can also handle more complex requests. Let's imagine that Suzie wants to see a list of the products she has previously bought from us. We know her e-mail address, so what we would need to do (if we had to do this manually and didn't have SQL to do it in a split-second for us) is the following:

1. Go to the Customers table and look down the email column for the record that contains Suzie's e-mail address. Then go across to the ID column to get Suzie's customer ID—her record's primary key (**Figure A.13**).

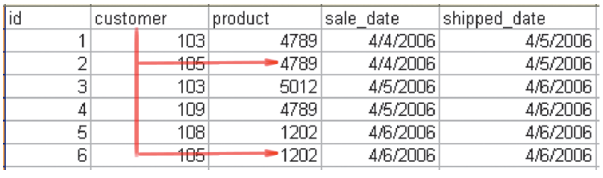
FIGURE A.13 Find the ID based on the e-mail address.



| id | first_nam | last_name | email | registered_date |
|---|---|---|---|---|
| 103 | John | Jones | jjones@abc.com | 4/4/06 |
| 105 | Suzie | Q | sq@ucd.com | 4/4/06 |
| 108 | Annie | Smith | annie@mysite.com | 4/5/06 |
| 109 | Bill | Williams | bill@dogbreath.com | 4/5/06 |

2. We have Suzie's ID; now we go to the Transactions table, find every record that has Suzie's ID in the customer column, and grab the corresponding product numbers for each record from the product column (**Figure A.14**).

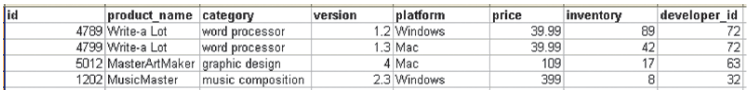
FIGURE A.14 Then find the associated transaction numbers.



| id | customer | product | sale_date | shipped_date |
|---|---|---|---|---|
| 1 | 103 | 4789 | 4/4/2006 | 4/5/2006 |
| 2 | 105 | 4789 | 4/4/2006 | 4/5/2006 |
| 3 | 103 | 5012 | 4/5/2006 | 4/6/2006 |
| 4 | 109 | 4789 | 4/5/2006 | 4/6/2006 |
| 5 | 108 | 1202 | 4/6/2006 | 4/6/2006 |
| 6 | 105 | 1202 | 4/6/2006 | 4/6/2006 |

3. With our list of product IDs, we head over to the Products table. We look down the primary key column for each matching product ID and return each matching record's product_name value (**Figure A.15**).

FIGURE A.15 Then look up the product names.



| id | product_name | category | version | platform | price | inventory | developer_id |
|---|---|---|---|---|---|---|---|
| 4789 | Write-a Lot | word processor | 1.2 | Windows | 39.99 | 89 | 72 |
| 4799 | Write-a Lot | word processor | 1.3 | Mac | 39.99 | 42 | 72 |
| 5012 | MasterArtMaker | graphic design | 4 | Mac | 109 | 17 | 63 |
| 1202 | MusicMaster | music composition | 2.3 | Windows | 399 | 8 | 32 |

4. Now we pass those product names to the middleware to display on a page.

```
Write-a lot

MusicMaster
```

That might sound like a lot of steps, but it can be achieved with a single SQL query:

```
SELECT Products.product_name FROM Products JOIN
Transactions JOIN Customers ON Products.id =
Transactions.product AND Customers.id = Transactions.
customer WHERE Customers.e-mail LIKE 'someguy@ithus.com'
```

SQL integrates very tightly with middleware products, so it's easy to build queries based on user actions, pass them to the database, and then incorporate the returned data into calculations and process it or simply write it to an XHTML page to serve up to the user, as shown on Chapter 2 of *Codin' for the Web*.

## Database Summary

A relational database is a collection of tables. Each table contains a different data set (customer information, product information, book titles, book authors, software languages, and so on) that you determine based on the needs of the application you are designing. Tables are made up of horizontal rows known as records. Each vertical column defines a type of data associated with the record. The item of data stored at the intersection of a record and a data type is called a value.

A defined procedure called normalization is used to ensure that tables do not contain redundant data—every data value, except foreign keys, is thereby unique.

Normalization requires that every record in every table have a primary key, which is a unique identifier, usually a number, for that record. Records in one table can be referenced from another table by their primary keys. A primary key used as a value in another table is, in that table, called a foreign key.

SQL enables the programmer to build, manage, and delete database tables and to add, edit, delete, and retrieve data from the tables.

# Summary

This completes our overview of three-tier architecture and the principles and software that are used to create it.