CHAPTER 17

The Shopping Cart

OVERVIEW

The shopping cart applications are one of the more popular CGI applications available. They allow a company to put its inventories online, where clients can browse the items using a simple Web-based interface.

A shopping cart is one of the more demanding CGI applications to manage, because it integrates many CGI routines into one system. For example, in order for each client to maintain a unique set of shopping cart items, the application must keep track of each client and the cart that corresponds to him or her. *Maintaining state*, as this is called, is difficult because HTTP is a "connectionless" protocol: every time a Web browser requests the attention of your server, it is considered a new request, unrelated to any other requests fulfilled by the server. Because each request is considered independently of others, the server has no way to keep track of what clients have been doing in the past. If you want

to keep track of a client's activities from page to page (perhaps keeping a list of items that the client has ordered), your CGI application must find a way to "remember" these transactions, because the server cannot.

Furthermore, the cart itself is a database file that is built and modified on the fly based on the client's needs. The client must be able to add and delete items as well as change the quantities of items ordered. The maintenance of the cart demands a full-fledged database manager.

Yet the script must do more than just display and modify shopping cart items that are contained in the datafile. It must also manipulate the database fields to perform price calculations, such as subtotaling and generating grand totals from subtotals.

Finally, the application must enable the client to browse throughout the store. They must have easy access to the inventory, and the script must provide the means of guiding them through.

In this version of the shopping cart script, we use an HTML-based method of storing information about items for sale. Product pages are built by the store administrator using plain HTML coding. A database version of this script is available at the following URL:

```
http://www.eff.org/~erict/Scripts/
```

INSTALLATION AND USAGE

The script should expand into a predesignated directory structure beginning with the root directory **Html_web_store**. Figure 17.1 shows the directory structure and the correct permissions for files and directories.

Html_web_store, the root directory, contains two files (**html_web_store.cgi** and **html_web_store.setup**) and three subdirectories (**Html**, **Library**, and **User_carts**). It must be readable and executable by the Web server.

html_web_store.cgi, the main script for the application, should be readable and executable by the Web server. It will be discussed in greater detail in the "Design Discussion" section.

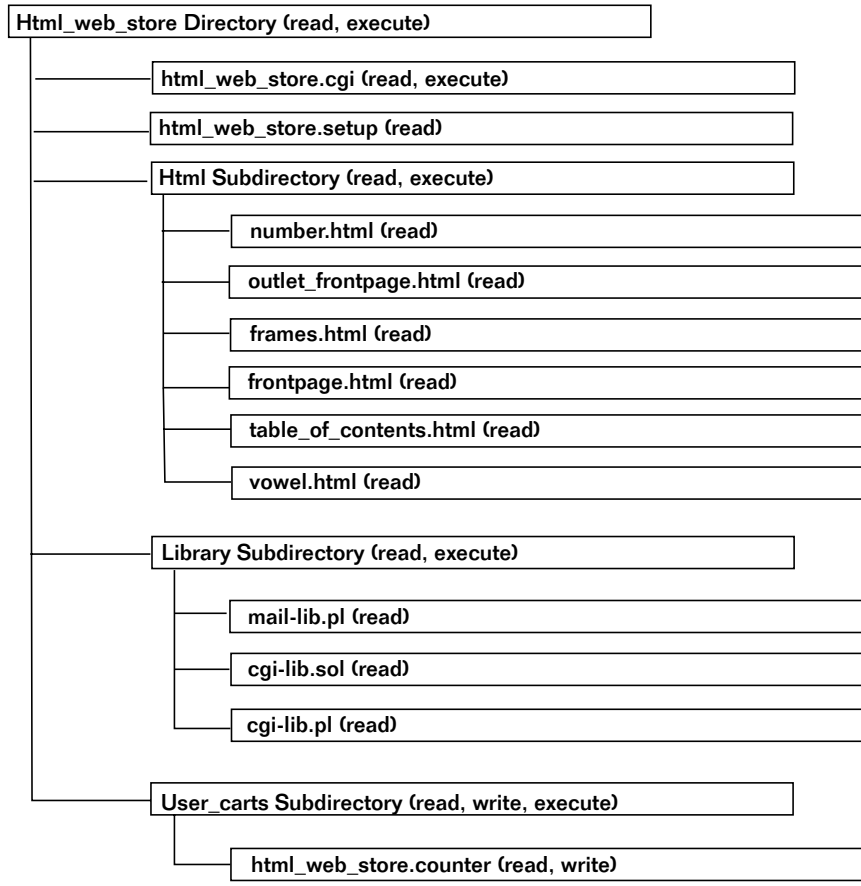

Figure 17.1 *HTML Web store directory structure.*

html_web_store.setup is the file you use to set all the server-specific variables and options. This file should be readable by the Web server and is discussed in greater detail in the "Setting Server-Specific Setup and Options" section.

Html contains all the miscellaneous HTML files used by the script. In the accompanying CD-ROM, this directory initially contains seven files: **number.html**, **outlet_frontpage.html**, **outlet_order_form.html**, **vowel.html**, **frames.html**, **frontpage.html**, and **table_of_contents.html**. These seven files are examples of how to maintain the HTML aspects of the shopping

cart and will be discussed individually in the "Server-Specific Setup and Options" section. The subdirectory itself must be readable and executable by the Web server, and all files in it must be readable.

The **Library** subdirectory contains the supporting library files. This application uses **cgi-lib.pl** to read and parse form data, **cgi-lib.sol** for the lock file and counter routines, and **mail-lib.pl** for mailing orders. Each of these libraries is discussed in Part Two and should be readable by the Web server. The **Library** subdirectory must be executable and readable by the Web server.

User_carts is the subdirectory that **html_web_store.cgi** uses to hold the shopping carts used by clients as well as a counter file and a few temporary files used to manipulate the carts. This directory should initially contain only **html_web_store.counter**, which is used by **html_web_store.cgi** to keep track of unique shopping cart item ID numbers; when clients wish to modify their carts, the script uses these ID numbers to identify unique items in their carts. The counter file should be readable and writable by the Web server. The script also uses this subdirectory to store the shopping cart files for each client. When the subdirectory is set to be readable, writable, and executable by the Web server, the script fills it with user carts and temporary files and prunes it when it no longer needs them.


NOTE

As you will see in the next section, the store administrator can define how long to keep old shopping carts and how often pruning takes place.

Server-Specific Setup and Options

The Set-up File

The setup file is used to configure the generic script for your own needs. It defines the following variables.

`$counter_file` is the location of the counter file used to assign unique numbers to every item added to the shopping cart.

`$main_script` is the local path of **html_web_store.cgi**.

`$main_script_url` is the URL of **html_web_store.cgi**.

`$order_form` is the location of the HTML order form that clients use to place their orders.

`$admin_email` is the E-mail address of the person who should receive the submitted orders.

`$email_subject` is the subject of the E-mail regarding an order.

`$cart_directory` is the location of the subdirectory where all the clients' shopping carts are stored.

`$html_directory` is the full path to the root directory where your HTML documents start. As archived on the accompanying CD-ROM, all HTML files are contained in the subdirectory where **html_web_store.cgi** is located, but it does not matter where your HTML files are as long as you include the entire path relative to the Web server's environment.

`$cgi_lib_pl` is the location of **cgi-lib.pl**.

`$cgi_lib_sol` is the location of **cgi-lib.sol**.

`$mail-lib.pl` is the location of **mail-lib.pl**.

`$frontpage_file` is the location of the very first HTML page that you want displayed when a new client enters the store.

`@display_fields` is the array containing all the shopping cart database fields that you want displayed when the user presses the view or modify button.

`@display_numbers` contains the corresponding array indexes to the elements specified in `@display_fields`. Thus, if `price` is the second element in `@display_fields` but you want it to be displayed first to the client, `@display_numbers` should have its first element set to `1`. Remember that array counting starts at zero and not at 1.

`$price_number` is the array index of the `price` element within `@display_fields`. To make price calculation, the script must be able to identify this index.

`@email_numbers` are the indexes to the elements in the `@display_fields` array that are sent as part of the E-mail to the store administrator so that he or she knows which items have been ordered.

%order_form_array is the associative array that contains the variable names and their associated display names. Note that the preceding numbers (01-, 02-, 03-) are used to "force" an order. Because associative arrays use hash tables, the orders cannot be determined unless you sort them with numbers as shown here. This is not mandatory, but appropriate naming of variables will make the screen display more intelligible to users.

The following is the setup file for you to examine:

```
$counter_file = "./User_carts/html_web_store.counter";
$main_script = "./html_web_store.cgi";
$main_script_url = "html_web_store.cgi";
$order_form = "outlet_order_form.html";
$admin_email = "selena\@foobar.com";
$email_subject = "Html Web Form Order";
$cart_directory = "./User_carts";
$html_directory = "./Html";
$cgi_lib_pl = "./Library/cgi-lib.pl";
$cgi_lib_sol = "./Library/cgi-lib.sol";
$mail_lib_pl = "./Library/mail-lib.pl";
$frontpage_file = "./Html/outlet_frontpage.html";
@display_fields = ("Catalog Number", "Price After
      Options", "Description", "Options");
@display_numbers = ("0", "4", "2", "3");
$price_number = "4";
@email_numbers = ("0", "4", "2", "3");
%order_form_array = ( '01-name', 'Name',
     '02-b_street_address', 'Billing Address Street',
     '03-b_city', 'Billing Address City',
     '04-b_state', 'Billing Address State',
     '05-b_zip', 'Billing Address ZIP',
     '06-b_country', 'Billing Address Country',
     '07-m_street_adress', 'Mailing Address Street',
     '08-m_city', 'Mailing Address City',
     '09-m_state', 'Mailing Address State',
     '10-m_zip', 'Mailing Address ZIP',
     '11-m_country', 'Mailing Address Country',
     '12-phone', 'Phone Number',
     '13-fax', 'Fax Number',
     '14-e-mail', 'Email',
     '15-URL', 'URL',
     '16-link', 'Link',
     '17-type_of_card', 'Type of Card',
     '19-cardname', 'Name Appearing on Card',
     '20-card_number', 'Card Number',
     '21-ex_date', 'Card Expiration');
```

The Front Page: outlet_frontpage.html

Next is an example of the sample front page HTML file included on the accompanying CD-ROM.

```
<HTML>
<HEAD>
<TITLE>Selena Sol's Electronic Outlet</TITLE>
</HEAD>
<BODY BGCOLOR = "FFFFFF" TEXT = "000000">
<B>Welcome to....</B>
<H2>Selena Sol's Electronic Outlet</H2>
where you will find the most up-to-date letters and numbers available.
All of our products are guaranteed for your lifetime, and our prices
are always competitive.
```

So far, all this is standard HTML. However, what follows is not. For the script to process your HTML pages, you need to follow a few standards. You should feel free to create any type of GUI you want for your store, but be sure to follow these standards in your HTML code. Specifically, any link that you create to another page must use the following format:

```
<A HREF = "html_web_store.cgi?page=xxx.html&cart_id=">XXX</A>
```

The shopper never sees a page directly. Instead, the shopper asks **html_web_store.cgi** to display a requested HTML document (in this case, **xxx.html**). This is why we do not have a link such as the following:

```
<A HREF="xxx.html">XXX</A>
```

The CGI script is actually a filter. It displays whatever HTML document that the page variable is set equal to.

The main reason we use this technique is that the script must "remember" a great deal of information, such as the name of the client using the script. Because the Web is connectionless, the Web protocols have no way to give the server or script the ability to remember information about a client. As far as the Web server is concerned, each new request is unrelated to any other request. Without these filtered variables, the script would not be able to remember whose cart was whose.

The trick is to pass such information as hidden form variables when a form is appropriate, or tagged on to the end of URL strings when a hyperlink is appropriate. In this case, we use a hyperlink that calls **html_web_store.cgi** and passes it the variable page. This page is set to the name of the HTML page that you want displayed.

The name that you set page **equal to must include the location relative to the variable** $html_directory. **Thus, if you had a subdirectory of your root HTML directory called Parts, with a file called carb.html, you would set** page **to** Parts/carb.html.

The script finds the requested HTML document and displays it for the client. The accompanying CD-ROM includes two examples—**vowels.html** and **numbers.html**—which you should read thoroughly.

In the example hyperlink tag, what is cart_id? It appears to be equal to nothing! Actually, this is a special tag that **html_web_store.cgi** will recognize. When it sees that tag, it automatically inserts the value of the user's cart ID number.

That is the real reason we run all HTML pages through **html_web_store.cgi** as a filter. The page variable tells the script which page to filter, and the cart_id variable is what is actually filtered. Because the Web is connectionless, the only way to remember information such as unique shopping cart ID numbers is to append them to the URL string.

But the HTML is not smart enough to do it on its own. The HTML is predesigned and static. How can it know in advance what the cart_ids will be? Thus, the script serves as a liaison between the static HTML document and the interactive client. The rest of the front page appears as follows:

```
<UL>
<LI><A HREF = "html_web_store.cgi?page=vowel.html&cart_id=">Vowels</A>
<LI><A HREF =
"html_web_store.cgi?page=number.html&cart_id=">Numbers</A>
```

The following hyperlink references a "frames" version of the Web store, which is included in the accompanying CD-ROM. We'll discuss how the

frames version works in the section titled "An Alternative Approach: Using Frames." For now, focus on the first two lines.

```
<LI><A HREF = "html_web_store.cgi?page=frames.html&cart_id=">Frames
    Version</A>
</UL>
</BODY></HTML>
```

On the Web, the front page will look like the one in Figure 17.2.

Figure 17.2 *The Web store front page.*

THE PRODUCT PAGE: VOWEL.HTML

Next is an example of a product page. Each product page begins with a standard header. You can change this header to reflect the specifics of your own site, but be absolutely sure that the form call in every page points to **html_web_store.cgi**.

```
<HTML>
<HEAD>
<TITLE>Vowel Page</TITLE>
</HEAD>
<BODY BGCOLOR = "FFFFFF" TEXT = "000000">
<FORM METHOD = "post" ACTION = "html_web_store.cgi">
```

The body of the product page appears below the header. You can customize this display in any way you would like. However, each item for sale must have an `<INPUT TYPE = "text">` field accompanying it. This text input field will record the quantity of the corresponding item and will have a `NAME` argument that contains all the item-specific information so that the script can identify the items in the client's cart.

Along similar lines, each item may also have user-configured options. For example, the shopper could choose a variety of colors or sizes.

```
<H2><CENTER>Vowel Mart!</CENTER></H2>
<TABLE>

<TR>
<TH>Quantity</TH>
<TH></TH>
<TH>Description</TH>
</TR>

<TR>
<TD COLSPAN = "3"><HR></TD>
</TR>
<TR>
```

The use of options will be discussed in greater detail later. It should be noted that in adding options you must be very careful not to make typos.

Defining an Item

Now comes the heart of the product page—the first occurrence of an item for sale. Notice first that it has a `TEXT` input field associated with it. This text field is three characters long and provides a maximum of four

characters. This field corresponds to the quantity that the user wants to order and is set to a maximum of 9999. You can change this limit if you want to.

Second, notice that the `NAME` argument of the `INPUT` field is `0001|15.98|The letter A`. This is where you define the information that gets added to the user's shopping cart. It is crucial that this list correspond to the list defined in line 199 of **html_web_store.cgi**:

```
($item_quantity, $item_id_number, $item_price, $item_description) =
split (/\|/, $item_ordered_with_options);
```

Notice that `$item_id_number` equals `0001`, `$item_price` equals `15.98`, and `$item_description` equals `The letter A`. `$item_quantity` will be defined by the user, so you need not worry much about that one. However, the others must correspond exactly. If you customize this script to your own inventory, take care to modify **html_web_store.cgi** as well as the setup file.

This information is also hard-coded into the arrays `@display_fields`, `@display_numbers`, `@email_numbers`, and the variable `$price_number` in the **html_web_store.setup** file.

The moral of this explanation is that if you want to add your own fields, you need to make sure that everything matches. You must change your HTML to reflect a new field as well as change line 199 and the lines in the setup file. Be very careful about doing this, because everything must be perfect for it to work!

```
<TD>
<INPUT TYPE = "text" NAME = "0001|15.98|The letter A"
       SIZE = "3" MAXLENGTH = "4"></TD>
<TD><IMG SRC = "http://www.eff.org/~erict/Graphics/Scripts/a.jpg"
ALIGN = "left"></TD>
<TD>You got it, the world-renowned letter "A" for the
low base price of $15.98<BR>
```

Use of Options

Now let's look at the use of options. We need a way to communicate to the script what an option is, which item it belongs to, and the value set by the user.

The first step is to make sure that we associate options with items for sale. We do this by using the NAME argument. Following is an example of using a select menu for options. In this case, the name syntax breaks down as follows:

- An option tag, which tells the script that the incoming data is an option and not an item.
- A unique sequence number for the option. Each item for sale may have several options associated with it. It is essential that each one gets its own number. If all the options for item #0001 were called option|0001, it would be impossible to parse them separately. So we will name each option uniquely, such as option|1|0001, option|2|0001, and option|3|0001.
- The ID of the item that the option is associated with. Notice that this ID is the same as the one that was used in the NAME argument for the quantity text box. This is deliberate and essential. Options must be associated with the items they modify.

```
<P><B>Available Options<B><P>
Font: <SELECT NAME = "option|1|0001">
<OPTION VALUE = "Times New Roman|0.00">Times New Roman (No
charge)
<OPTION VALUE = "Arial|1.50">Arial (+ $1.50)
<OPTION VALUE = "Chicago|2.00">Chicago (+ $2.00)
</SELECT>
<P>
```

The following is an example of using a radio button to create an option. It uses the same naming conventions as the <SELECT> tag.

```
Color: <BR>
<INPUT TYPE = "radio" NAME = "option|2|0001"
       VALUE = "Red|0.00" CHECKED>Red<BR>
<INPUT TYPE = "radio" NAME = "option|2|0001"
       VALUE = "Blue|.50">Blue (+ $1.00)
</TD>
</TR>
<TR>
<TD COLSPAN = "3"><HR></TD>
```

```
</TR>
<TR>
<TD>
<INPUT TYPE = "text" NAME = "0002|12.98|The letter E"
       SIZE = "3" MAXLENGTH = "4"></TD>
<TD><IMG SRC = "http://www.eff.org/~erict/Graphics/Scripts/e.jpg"
ALIGN = "left"></TD>
<TD>You got it, the world-renowned letter "E" for the low base price
of $12.98<BR>
<P><B>Available Options<B><P>
Font: <SELECT NAME = "option|1|0002">
<OPTION VALUE = "Times New Roman|0.00">Times New Roman (No charge)
<OPTION VALUE = "Arial|1.50">Arial (+ $1.50)
<OPTION VALUE = "Chicago|2.00">Chicago (+ $2.00)
</SELECT>
<P>
Color: <BR>
<INPUT TYPE = "radio" NAME = "option|2|0002"
       VALUE = "Red|0.00" CHECKED>Red<BR>
<INPUT TYPE = "radio" NAME = "option|2|0002"
       VALUE = "Blue|.50">Blue (+ $1.00)
</TD>
</TR>
<TR>
<TD COLSPAN = "3"><HR></TD>
</TR>
```

Simple Item Definition

Finally, we have provided an example of an item with no options. This example is much simpler, because all you need to worry about is the one `NAME` argument of each item for sale.

```
<TR>
<TD>
<INPUT TYPE = "text" NAME = "0004|18.98|The letter I"
       SIZE = "3" MAXLENGTH = "4"></TD>
<TD><FONT SIZE = "+9"><B>I</B></FONT></TD>
<TD>Yep, this is the letter I am selling for the low price of $18.98.
No options are available on this model.</TD>
</TR>
<TR>
<TD COLSPAN = "3"><HR></TD>
</TR>
</TABLE>
```

Next, we see the standard page footer that must appear on every HTML page. The first two hidden input tags are specially marked with the double percent (%%) sign so that the script can identify the tags and substitute them for the actual values associated with the client. You should make sure to have these tags on every page.

```
<P>
<CENTER>
<INPUT TYPE = "hidden" NAME = "cart_id"
       VALUE = "%%cart_id%%">
<INPUT TYPE = "hidden" NAME = "page"
       VALUE = "%%page%%">
```

The remaining input tags, which are used by the client to select the operation to be performed, can have their VALUE argument modified but not their NAME argument.

```
<INPUT TYPE = "submit" NAME = "add_to_cart"
       VALUE = "I'd like to buy a vowel">
       VALUE = "View/Modify Cart">
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
       VALUE = "Return to Front page">
<INPUT TYPE = "submit" NAME = "order_form"
       VALUE = "Checkout Stand">
,/FORM></BODY></HTML>
```

On the Web, the vowel page looks like Figure 17.3.

Alternative Approach: Using Frames

It is also possible to use frames as a navigational tool throughout your store. To demonstrate this, we have included three extra HTML files that use frames: **frames.html**, **frontpage.html**, and **table_of_contents.html**.

If you recall from the discussion of **outlet_frontpage.html**, we noted that the following line references a frames version of the Web store:

```
<LI><A HREF = "html_web_store.cgi?page=frames.html&cart_id=">Frames
Version</A>
```

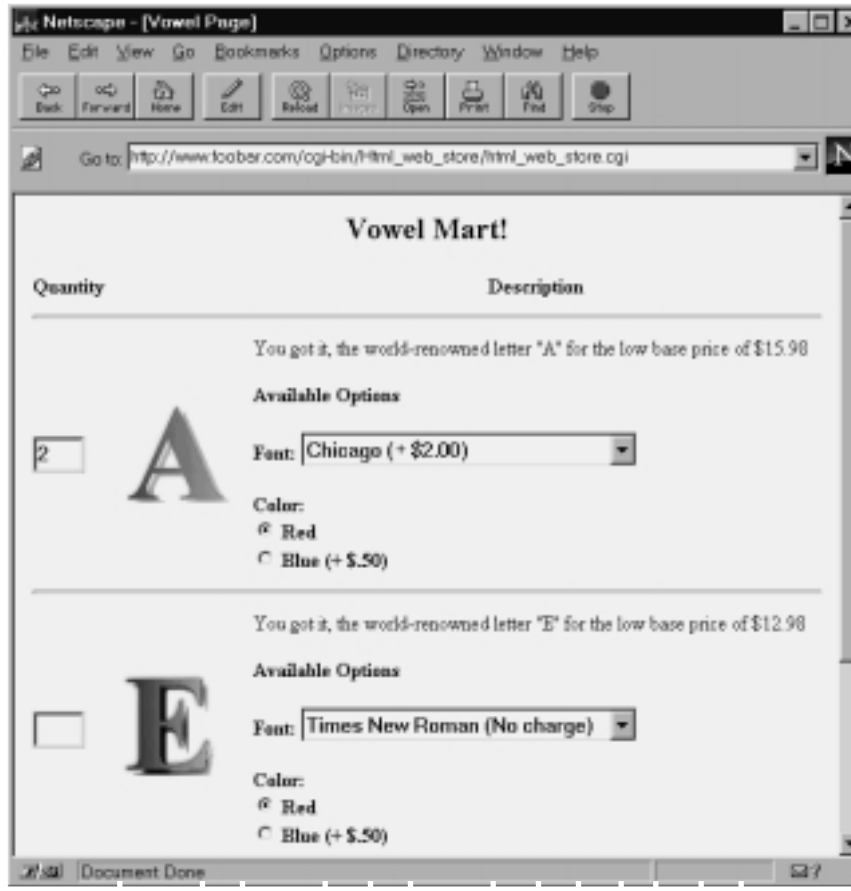

Figure 17.3 *The vowel order page.*

You can easily run the Web store with a frames interface by using this example. **frames.html** is the file that defines the windows. Here is the example of **frames.html** as it exists on the CD-ROM:

```
<HTML>
<HEAD>
<TITLE>Selena Sol's Public Domain CGI Script Archive and Resource
Library: Shopping Cart Frames Example</TITLE>
```

```
</HEAD>
<FRAMESET COLS = "111, 80%">
<FRAME SRC = "html_web_store.cgi?page=table_of_contents.html"
SCROLLING = "auto">
<FRAME NAME = "main" SRC = "html_web_store.cgi?page=frontpage.html">
</FRAMESET></BODY></HTML>
```

As you can see, in this case we used two windows. Figure 17.4 depicts the frames interface.

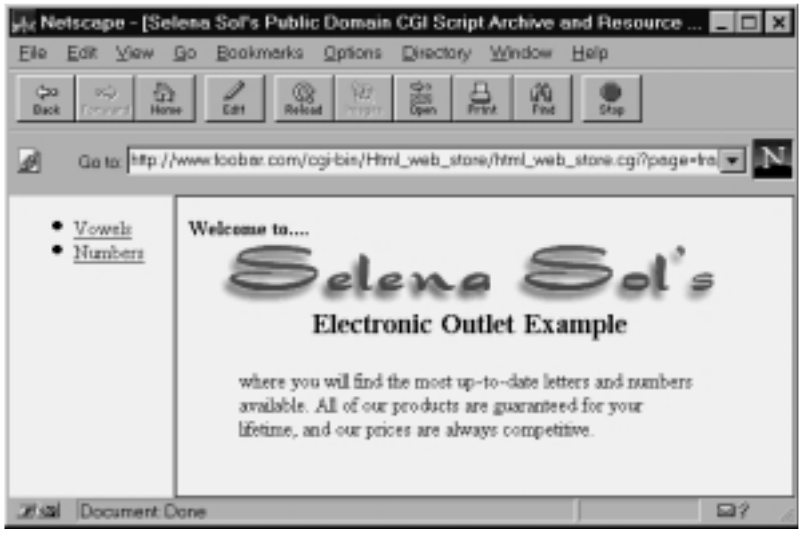

Figure 17.4 *The frames interface.*

The first window, **table_of_contents.html**, lists the different product pages that the client may wish to browse. This page is similar to the front page in the nonframes version and includes the urlencoded information that **html_web_store.cgi** must filter.

```
<HTML>
<HEAD>
<TITLE>Selena Sol's Electronic Outlet</TITLE>
</HEAD>
<BODY BGCOLOR = "FFFFFF" TEXT = "000000">
<UL>
<LI><A HREF="html_web_store.cgi?page=vowel.html&cart_id=" TARGET =
```

```
"main">Vowels</A>
<LI><A HREF="html_web_store.cgi?page=number.html&cart_id=" TARGET =
"main">Numbers</A> </UL>
</BODY></HTML>
```

The second window, **frontpage.html**, is used to generate a front page store banner. We have chosen to include no navigational links in this window, because the **table_of_contents.html** window is used for navigation. **frontpage.html** is simply a front banner for the store. Until the client begins shopping and the frame is used to display products, it serves the simple purpose of filling the main frame. Here is the text of **frontpage.html**:

```
<HTML>
<HEAD>
<TITLE>Selena Sol's Electronic Outlet</TITLE>
</HEAD>
<BODY BGCOLOR = "FFFFFF" TEXT = "000000">
<B>Welcome to....</B><BR>
<CENTER>
<IMG SRC =
"http://www.eff.org/~erict/Graphics/Scripts/web_store_frontpage.gif">
</CENTER>
<BLOCKQUOTE>
where you will find the most up-to-date letters and numbers available.
All of our products are guaranteed for your lifetime, and our prices
are always competitive.
</BLOCKQUOTE></BODY></HTML>
```

The window with **frontpage.html** is the "main" window, in which the various pages defined in the **table_of_contents.html** window are displayed. Because both pages are initially loaded through **frames.html**, if you wished to include navigational links in the **frontpage.html** window, they would also be filtered by **html_web_store.cgi**.

Because the pages are filtered through **html_web_store.cgi**, they are filtered as any other page and hence are able to maintain state.

The Order Form: outlet_order_form.html

The accompanying CD-ROM also includes an example of an order form. There is no magic to this form; it is just a standard HTML form. It refer-

ences **html_web_store.cgi**, which is used to process the order, and all variable names are prefixed with a number and a hyphen (-). The numbers represent the order in which you want the variables displayed. The numbers begin at 01- and count upward.

```
<HTML>
<HEAD>
<TITLE>Outlet Order Form!!</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<FORM ACTION = "html_web_store.cgi" METHOD = "post">
<CENTER>
<TABLE WIDTH = "90%" BORDER="3" CELLPADDING="2">
<TR><TD COLSPAN=2><FONT SIZE=+1>Personal Information:</FONT></TD></TR>

<TR><TD>Name(First &amp;amp; Last)</TD>
<TD><INPUT TYPE = "text" NAME = "01-name" SIZE = "30"
           MAXLENGTH = "30"></TD></TR>

<TR>
<TD COLSPAN = "2"><FONT SIZE = "+1">Billing Address:</FONT></TD>
</TR>

<TR><TD>Street:</TD>
<TD><INPUT TYPE="text" NAME="02-b_street_address" SIZE="30"></TD></TR>

<TR><TD COLSPAN=2>City:
<INPUT TYPE="text" NAME="03-b_city" SIZE="10">
State: <INPUT TYPE="text" NAME="04-b_state" SIZE="2" MAXLENGTH="8">
ZIP: <INPUT TYPE="text" NAME="05-b_zip" SIZE="5" MAXLENGTH="5">
Country: <INPUT TYPE="text" NAME="06-b_country" SIZE="10"
MAXLENGTH="20">
</TD></TR>

<TR><TD COLSPAN=2><FONT SIZE=+1>Mailing Address (If
Different):</FONT></TD></TR>

<TR><TD>Street:</TD>
<TD><INPUT TYPE="text" NAME="07-m_street_adress" SIZE="30"></TD></TR>

<TR>
<TD COLSPAN=2>City:
```

```
<INPUT TYPE="text" NAME="08-m_city" SIZE="10"> State:
<INPUT TYPE="text" NAME="09-m_state" SIZE="2" MAXLENGTH="8">ZIP:
<INPUT TYPE="text" NAME="10-m_zip" SIZE="5" MAXLENGTH="5">Country:
<INPUT TYPE="text" NAME="11-m_country" SIZE="10" MAXLENGTH="20">
</TD></TR>

<TR><TD>
Phone: <INPUT TYPE="text" NAME="12-phone" SIZE="10"
MAXLENGTH="12"></TD><TD> Fax: <INPUT TYPE="text" NAME="13-fax"
SIZE="10" MAXLENGTH="12">
</TD></TR>

<TR><TD>E-Mail:</TD><TD>
<INPUT TYPE="text" NAME="14-e-mail" MAXLENGTH="30">
</TD></TR>

<TR><TD>URL:</TD><TD>
<INPUT TYPE="text" NAME="15-URL" MAXLENGTH="30"> Request Link: <INPUT
TYPE="checkbox" NAME="16-link" VALUE="on">
</TD></TR>

<TR><TD COLSPAN=2><FONT SIZE=+1>Credit Card
Information:</FONT></TD></TR>

<TR><TD COLSPAN=2>
<INPUT TYPE="radio" NAME="17-type_of_card" VALUE="visa">Visa <INPUT
TYPE="radio" NAME="17-type_of_card" VALUE="mastercard">Mastercard
<INPUT TYPE="radio" NAME="17-type_of_card"
VALUE="discover">Discover</TD></TR>

<TR><TD>Name on Card:</TD><TD>
<INPUT TYPE="text" NAME="19-cardname" SIZE="30"
MAXLENGTH="30"></TD></TR>

<TR><TD>Number:</TD><TD><INPUT TYPE="text" NAME="20-card_number"
MAXLENGTH="20"></TD></TR>

<TR><TD>Exp. Date:</TD><TD>
<INPUT TYPE="text" NAME="21-ex_date" SIZE="10"
MAXLENGTH="10"></TD></TR>

</TABLE>
<P>
```

```
Allow 3-4 weeks for delivery.  Shipping prices and delivery times
may vary when shipping to cities outside the continental U.S.
<P>
<INPUT TYPE=reset>
<INPUT TYPE=submit NAME = "send_in_order"
       VALUE = "Submit Secure Order"><BR>
</CENTER>
</FORM>
</BODY></HTML>
```

On the Web, the order form looks like Figure 17.5.

Figure 17.5 *The Web store order form.*

Running the Script

Once you have configured the setup file to the specifics of your local server setup, you can take a trip to the store. To access the script, create a hyperlink to the location of the main script as follows:

```
<A HREF = "http://www.foobar.com/cgi-
bin/Html_web_store/html_web_store.cgi">Html Web Store</A>
```

Because the front page HTML file is specified in the setup, you need not pass the script any extra information initially.

Design Discussion

The logic of the application is depicted in Figure 17.6.

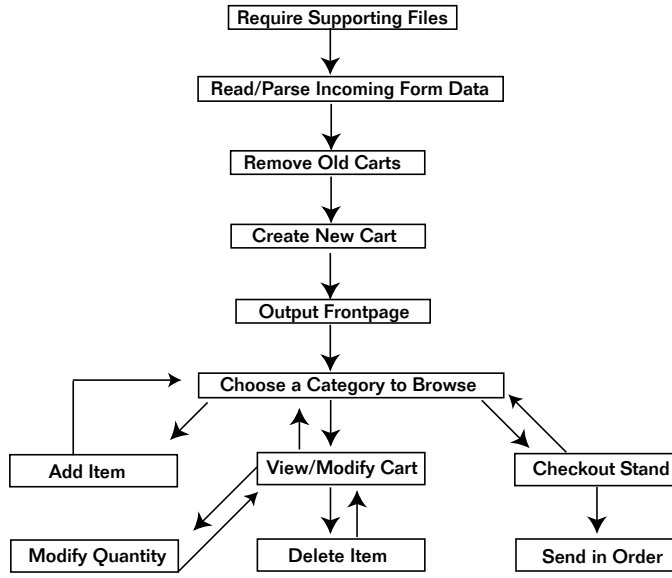

Figure 17.6 *Script logic.*

html_web_store.cgi begins by starting the Perl interpreter and printing the http header.

```
#!/usr/local/bin/perl
print "Content-type: text/html\n\n";
```

Loading the Libraries

Next, the script loads the supporting files.

```
require "./html_web_store.setup";
require "$cgi_lib_pl";
require "$cgi_lib_sol";
require "$mail_lib_pl";
```

Reading and Parsing Form Data

Next, the script uses **cgi-lib.pl** to parse the incoming form data.

```
&ReadParse(*form_data);
```

Removing Old Carts

Then the script cleans up the **User_carts** subdirectory; there is no need to keep old carts around once the clients are gone. The script first opens the **User_carts** subdirectory. Then it reads the subdirectory contents and uses grep to grab every CART. Then the script closes the directory. Thus, the `@carts` array will contain every cart file as an element.

```
opendir (USER_CARTS, "$cart_directory") ||
&open_error($cart_directory);
@carts = grep(/.cart$/,readdir(USER_CARTS));
closedir (USER_CARTS);
```

For every cart in the directory, the script checks its age. If the cart is older than half a day, it is deleted.

```
foreach $cart (@carts)
 {
```

```
if (-M "$cart_directory/$cart" > .5)
  {
  unlink("$cart_directory/$cart");
  }
}
```

Generating a New Shopping Cart

Next, the script creates a new cart for the new client. It first determines whether it has received a shopping cart ID number as form data. If it has not, the script knows that the client has not yet received a unique shopping cart. So it assigns the client a unique shopping cart file that is used while the client browses the store.

To generate a unique cart ID, the script first generates a random (`rand`) eight-digit (10000000) integer (`int`) and then adds to that the current process ID (`$$`). It then appends the process ID to the end of the integer. `srand`, which must be called before the `rand` command is used, is seeded with the process ID and the time, a technique that provides a better randomizing algorithm.

```
if ($form_data{'cart_id'} eq "")
 {
 srand (time|$$);
 $cart_id = int(rand(10000000));
 $cart_id .= ".$$";
```

Once the unique number has been generated, the script creates a new cart for the shopper using the value of `$cart_id` as the name of the file.

```
open (CART, ">$cart_directory/$cart_id.cart") ||
&open_error("$cart_directory/$cart_id.cart");
 }
```

If there was a shopping cart ID number coming in as form data, on the other hand, the script simply reads the `form_variable` into the `$cart_id` Perl variable.

```
else
  {
```

```
$cart_id = "$form_data{'cart_id'}";
}
```

Once the client has received a cart, that cart number must be passed from screen to screen as url-encoded information or as a hidden form variable so that the client does not lose the cart.

Displaying the Front Page

Now that the client has received a cart, it is time to display the first page of the store. We will assume that the site administrator has created a front page and has set that location to `$frontpage_file` in the store-specific setup file.

There are two cases in which the script should display the front page: first, if the user has specifically asked to return to the front page and, second, if no product page has been assigned. The main purpose of the front page is to query the client for the product page the client is interested in browsing. The front page presents all the options and creates hyperlinks that specify the product pages for the script. (Recall the `page` variable in our HTML documents.)

```
if ($form_data{'page'} eq "" ||
$form_data{'return_to_frontpage'} ne "")
{
```

If it has been asked for the front page, the script opens it and displays it line by line.

```
open (FRONTPAGE, "$frontpage_file") ||
&open_error("$frontpage_file");
while (<FRONTPAGE>)
{
```

However, the script must do a couple of things to the front page before it is displayed. First, it must make sure that the client's unique cart ID gets passed from the front page to any successive pages. We do this by urlencoding. We have the user click on a link that says

```
<A HREF ="web_store.cgi?page=foo.html&cart_id=yyy">.
```

Notice two things in that line. First, we define a page in those links. Second, we add the other half to the previously half-completed variable `userid`. As discussed in the "Installation and Usage" section, this tag, `userid=`, is a tag that this script is looking for. Any time (`/g`) it sees the tag, the script substitutes (`s/`) occurrences of `cart_id=` with `cart_id= the actual cart id of the user`. Because we do not know the cart ID ahead of time, we cannot hard-code it into the HTML. Thus, we create the `cart_id=` tag so that this script can finish the job on the fly:

```
s/cart_id=/cart_id=$cart_id/g;
print "$_";
}
```

Then the script closes the front page file and quits.

```
  close (FRONTPAGE);
  exit;
} # End of if ($form_data{'page_to_view'} eq "")
```

An example of the front page was shown in Figure 17.2.

Adding an Item to the Client's Cart

Once the client has decided to purchase an item from one of the product pages, he or she adds a quantity to the text box and presses **submit**. The resulting displayed page is printed using routines we'll explain later. First, the script adds a new item to the cart.

```
if ($form_data{'add_to_cart'} ne "")
{
```

The script begins by using the `%form_data` array passed to it by **cgi-lib.pl**. All the keys of the `%form_data` associative array are dropped into the `@incoming_data` array. A key is like a variable name, whereas a value is the value associated with that variable name. Each of the text fields where the client can enter quantities is associated with the item IDs of the items (as

defined in the displayed category view at the end of this script), so the HTML should read `<INPUT TYPE = "text" NAME = "1234"...>` for the item with database ID `1234`, and `<INPUT TYPE = "text" NAME = "5678"...>` for item `5678`. If the client orders two of `1234` and nine of `5678`, then `@incoming_data` will be a list of `1234` and `5678` such that `1234` is "associated" with the value of 2 in `%form_data` and `5678` is associated with the value of 9.

```
@items_ordered = keys (%form_data);
```

Next, the script begins going through the list of incoming items one by one.

```
foreach $item (@items_ordered)
{
```

However, there are some incoming items—`cart_id`, `page`, and `add_to_cart`—that the script will not care about. These are administrative values set internally by this script and are not part of the client's order. These values come in as form data, and the script will need them for other things, but they're not used to fill the user's cart. Similarly, it will not need to worry about any items that have empty values. If the shopper did not enter a quantity for a product, then the script should not add it to the cart.

```
if ($item ne "cart_id" && $item ne "page" &&
    $item ne "add_to_cart" && $form_data{$item} ne "")
{
```

Next, the script separates the ordered items from the options that modify those items. If `$item` begins with the word `option`, which we set specifically in the HTML file, the script adds (`pushes`) that item to the array called `@options`.

```
if ($item =~ /^option/i)
  {
  push (@options, $item);
  }
```

On the other hand, if `$item` is not an option, the script adds it to the array `@items_ordered_with_options`, but it adds both the item and its

value. The value will be a quantity, and the item will be something like `0001|12.98|The letter A`, as defined in the HTML file.

```
else
  {
  push (@items_ordered_with_options,
            "$form_data{$item}\|$item\|");
  }
 } # End of if ($item ne "cart_id"....
} # End of foreach $item (@items_ordered)
```

Then the script goes through the array `@items_ordered_with_options` one item at a time.

```
foreach $item_ordered_with_options
        (@items_ordered_with_options)
{
```

Next, the script initializes a few variables that it will use for each item. `$options` is used to keep track of all the options selected for any given item. `$option_subtotal` is used to determine the total cost of each option. `$option_grand_total` is used to calculate the total cost of all ordered options. `$item_grand_total` is used to calculate the total cost of the item ordered.

```
$options = "";
$option_subtotal = "";
$option_grand_total = "";
$item_grand_total = "";
```

Next, the script `splits` the `$item_ordered_with_options` into its fields.

```
($item_quantity, $item_id_number, $item_price,
 $item_description) =
 split (/\|/, $item_ordered_with_options);
```

For every option in `@options`, the script `splits` each option into its fields.

```
foreach $option (@options)
{
```

```
($option_marker, $option_number,
 $option_item_number) = split
     (/\|/, $option);
```

Then the script compares the name of the item with the name associated with the option. If they are the same, the script knows that this is an option that was meant to modify this item.

```
if ($option_item_number eq "$item_id_number")
     {
```

Because the script must apply this option to the item, it splits out the value associated with the option and appends it to `$options`. Once it appends all the options, using ".=," the script will have one big string containing all the options so that we can print them.

```
($option_name, $option_price) =
split (/\|/,$form_data{$option});
$options .= "$option_name $option_price,";
```

The script also calculates the cost with options:

```
$unformatted_option_grand_total = $option_grand_total +
                                  $option_price;
$option_grand_total = sprintf ("%.2f\n",
                      $unformatted_option_grand_total);

}
```

We use the `sprintf` function to format the price only to two decimal places. The `sprintf` function returns a string formatted by the `printf` conventions. The string to be formatted contains an embedded field specifier (%) into which the formatted conversion (`.2f`) of a string (`$unformatted_option_grand_total`) is placed. The `.2f` option tells `sprintf` to format the string to only two decimal places. The `sprintf` function is also covered in depth in the UNIX man pages.

It also removes the last comma in `$options`. If you scan up a few lines, you will see that a comma is added to the end of each option. The last option does not need that last comma, so we strip it off.

```
chop $options;
```

Next, the script adds a space after each comma so that the display looks nicer.

```
$options =~ s/,/, /g;
```

Then it uses the subroutine `counter` in **cgi-lib.sol**, sending it the location of the counter file defined in the setup file. This routine returns one variable, `$item_number`, that we use to identify a shopping cart item absolutely. In this way, when we modify and delete items we know exactly which items to affect. The counter file is protected by using the lock file routines in **cgi-lib.sol**.

```
&GetFileLock("$counter_file.lock");
&counter ($counter_file);
&ReleaseFileLock("$counter_file.lock");
```

Finally, the script performs the last price calculations and appends every ordered item to `$cart_row`.

```
$unformatted_item_grand_total = $item_price +
                                $option_grand_total;
$item_grand_total = sprintf ("%.2f\n",
                    $unformatted_item_grand_total);
chop $item_grand_total;
$cart_row .=
"$item_quantity\|$item_id_number\|$item_price\|$item_description\|$opt
ions\|$item_grand_total\|$item_number\n";
}
```

When it has finished appending all the items to `$cart_row`, the script opens the user's shopping cart and adds the new items.

```
open (CART,
">>$cart_directory/$form_data{'cart_id'}.cart")
|| &open_error ("$cart_directory/$form_data{'cart_id'}.cart");
print CART "$cart_row";
close (CART);
```

It then sends the client back to the product page using the `display_items_for_sale` subroutine at the end of the script.

```
&display_items_for_sale;
exit;
}
```

Displaying the View/Modify Cart Screen

A client may decide to reduce the quantities of some of the chosen items or even to delete them from the cart. Or perhaps the client just wants to see what has been placed in the cart so far.

Figure 17.7 *Cart modification interface.*

The script's first step in such circumstances is to send an HTML form that includes a display of the client's current cart. We use the subroutine `display_cart_contents` at the end of this script.

```
if ($form_data{'modify_cart'} ne "")
{
&display_cart_contents
}
```

Figure 17.7 shows the cart interface on the Web.

Generating a Change Quantity Form

If the client asks to make a quantity modification, the script displays a form where the changes can be specified.

```
if ($form_data{'change_quantity'} ne "")
{
```

The script first outputs the header as it did before.

```
&page_header("Change Quantity");
print <<"end_of_html";
<CENTER>
<TABLE BORDER = "1">
<TR>
<TH>New Quantity</TH>
end_of_html
```

It then builds the table headers.

```
foreach $field (@display_fields)
  {
  print "<TH>$field</TH>\n";
  }
print "<TH>Quantity</TH>\n<TH>Subtotal</TH>\n</TR>\n";
```

Next, the client's cart is opened and the script begins going through it line by line.

```
open (CART, "$cart_directory/$form_data{'cart_id'}.cart")
|| &open_error("$cart_directory/$form_data{'cart_id'}.cart");
while (<CART>)
{
@cart_fields = split (/\|/, $_);
```

This time, however, the script pops out both the cart-specific item ID number and the store-specific item ID number and then pushes them back onto the array once it has assigned the values to some variables. Using the shift operator, the script also grabs the quantity, but it does not unshift it back into the array; we will not need it in the array, because we have assigned it to the variable $quantity. Finally, the script chops off the newline character for $cart_row_number.

```
$cart_row_number = pop(@cart_fields);
$db_number = pop(@cart_fields);
push (@cart_fields, $db_number);
push (@cart_fields, $cart_row_number);
$quantity = shift(@cart_fields);
chop $cart_row_number;
```

Then the script begins creating a table-based form for quantity changes. First, it creates an input text field where users can enter a new quantity. In this case, the script assigns the $cart_row_number to the name in the input field so that it will later remember which item was associated with the quantity change.

```
print "<TD ALIGN = \"center\">\n";
print "<INPUT TYPE = \"text\"
              NAME = \"$cart_row_number\"
              SIZE =\"3\">\n";
```

The script then creates the table rows and page footer as it did before.

```
foreach $display_number (@display_numbers)
  {
  if ($cart_fields[$display_number] eq "")
    {
    $cart_fields[$display_number] = "<CENTER>-
                                     </CENTER>";
    }
  print "<TD ALIGN =
  \"center\">$cart_fields[$display_number]</TD>\n";
  } # End of foreach $display_number (@display_numbers)

print "<TD ALIGN = \"center\">$quantity</TD>\n";
$unformatted_subtotal = ($quantity *
                         $cart_fields[$price_number]);
$subtotal = sprintf ("%.2f\n", $unformatted_subtotal);
$unformatted_grand_total = $grand_total + $subtotal;
$grand_total = sprintf ("%.2f\n",
               $unformatted_grand_total);
print "<TD ALIGN = \"center\">\$$subtotal</TD>\n";
print "</TR>\n";
} # End of while (<CART>)
close (CART);

print <<"end_of_html";
</TABLE>
<P>
Grand Total = \$$grand_total
<P>
<INPUT TYPE = "submit" NAME = "submit_change_quantity"
       VALUE = "Submit Quantity Changes">
<INPUT TYPE = "submit" NAME = "continue_shopping"
       VALUE = "Continue Shopping">
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
       VALUE = "Return to Front page">
<INPUT TYPE = "submit" NAME = "order_form"
       VALUE = "Checkout Stand">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the change quantity screen looks like Figure 17.8.

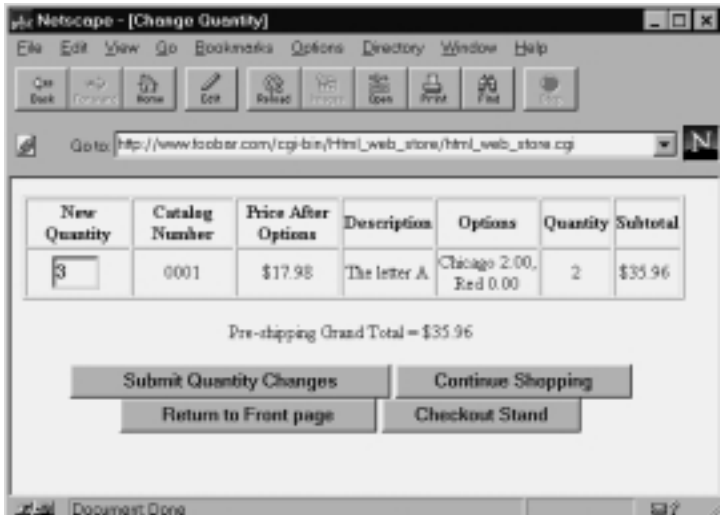

Figure 17.8 *Change quantity interface.*

Changing the Quantity of an Item in the Client's Cart

Once the client has typed some quantity changes and has submitted the information using the form generated in the previous section, the script makes the modifications to the database.

```
if ($form_data{'submit_change_quantity'} ne "")
{
```

First, it gathers the keys as it did for the front page generation earlier, checking to make sure the user entered a positive integer.

```
@incoming_data = keys (%form_data);
foreach $key (@incoming_data)
  {
  if ($key =~ /[\d]/ && $form_data{$key} =~ /[\D]/)
    {
    &bad_order_note;
    }
```

As it did earlier, the script creates an array (`@modify_items`) of valid keys.

```
unless ($key =~ /[\D]/ && $form_data{$key} =~ /[\D]/)
  {
  if ($form_data{$key} ne "")
    {
    push (@modify_items, $key);
    }
  } # End of unless ($key =~ /[\D]/...
} # End of foreach $key (@incoming_data)
```

Next, it opens the client's cart and goes through it as usual.

```
open (CART,
"$cart_directory/$form_data{'cart_id'}.cart")
 || &open_error ("$cart_directory/$form_data{'cart_id'}.cart");
while (<CART>)
  {
  @database_row = split (/\|/, $_);
  $cart_row_number = pop (@database_row);
  push (@database_row, $cart_row_number);
  $old_quantity = shift (@database_row);
  chop $cart_row_number;
```

Then the script checks to see whether the item number submitted as form data is equal to the number of the current database row.

```
 foreach $item (@modify_items)
  {
  if ($item eq $cart_row_number)
    {
```

If it is, the script creates the `$shopper_row` variable and begins creating the modified row. The script replaces the old quantity with the quantity submitted by the client (`$form_data{$item}`). Recall that `$old_quantity` has already been `shift`ed off the array.

```
$shopper_row .= "$form_data{$item}\|";
```

Now the script adds the rest of the database row to `$shopper_row` and sets two flag variables. `$quantity_modified` is used to notify the script that the current row has had a quantity modification. `$invalid_submission` does basically the same thing but is used to make sure that the client submitted a valid quantity change.

```
foreach $field (@database_row)
  {
  $quantity_modified = "yes";
  $invalid_submission = "no";
  $shopper_row .= "$field\|";
  }
 chop $shopper_row; Get rid of last |
 } # End of if ($item eq $cart_row_number)
} # End of foreach $item (@modify_items)
```

If the script gets here and if `$quantity_modified` has not been set to `yes`, the script knows that the preceding routine was skipped, because the item number submitted from the form was not equal to the current database ID number. The script now knows that no quantity in the current row is being changed and the row can be added to `$shopper_row` as is. Remember, we want to add the old rows as well as the new, modified ones.

```
if ($quantity_modified ne "yes")
  {
  $shopper_row .= $_;
  }
 $quantity_modified = "";
} # End of while (<CART>)
close (CART);
```

Next, the script checks to make sure that somewhere along the line the client made a valid submission. If the submission was not made, the script executes the `bad_order_note` subroutine.

```
if ($invalid_submission ne "no")
  {
  &bad_order_note
  }
```

If the client made a valid submission, on the other hand, the script overwrites the old cart with the new information and sends the client back to the previous category page.

```
open (CART, ">
$cart_directory/$form_data{'cart_id'}.cart")
      || &open_error
```

```
("$cart_directory/$form_data{'cart_id'}.cart");
print CART "$shopper_row";
close (CART);
&display_cart_contents;
exit;
}
```

Generating the Delete Item Form

Now suppose that the client asked to delete an item rather than modify the quantity.

```
if ($form_data{'delete_item'} ne "")
{
```

In this case, the script prints the usual page and table headers.

```
&page_header("Delete Item");
print <<"end_of_html";
<CENTER>
<TABLE BORDER = "1">
<TR>
<TH>Delete Item</TH>
end_of_html
foreach $field (@display_fields)
  {
  print "<TH>$field</TH>\n";
  }
print "<TH>Quantity</TH>\n<TH>Subtotal</TH>\n</TR>\n";
```

It then opens the cart and this time adds a check box instead of an input text field so that the client can specify multiple items to delete. As it did in the routines used for modification, the script assigns the `NAME` equal to the `cart_row_number`.

```
open (CART,
"$cart_directory/$form_data{'cart_id'}.cart")
      || &open_error ("$cart_directory/$form_data{'cart_id'}.cart");
while (<CART>)
  {
  @cart_fields = split (/\|/, $_);
```

```
  $cart_row_number = pop(@cart_fields);
  $db_number = pop(@cart_fields);
  push (@cart_fields, $db_number);
  push (@cart_fields, $cart_row_number);
  chop $cart_row_number;
  $quantity = shift(@cart_fields);
  print "<TD ALIGN = \"center\">\n";
  print "<INPUT TYPE = \"checkbox\"
                NAME = \"$cart_row_number\">\n";
```

Next, the script displays the table rows as before.

```
foreach $display_number (@display_numbers)
  {
  if ($cart_fields[$display_number] eq "")
    {
    $cart_fields[$display_number] = "<CENTER>-</CENTER>";
    }
 print "<TD ALIGN =
 \"center\">$cart_fields[$display_number]</TD>\n";
 } # End of foreach $display_number (@display_numbers)

 print "<TD ALIGN = \"center\">$quantity</TD>\n";
 $unformatted_subtotal = ($quantity *
                          $cart_fields[$price_number]);
 $subtotal = sprintf ("%.2f\n", $unformatted_subtotal);
 $unformatted_grand_total = $grand_total + $subtotal;
 $grand_total = sprintf ("%.2f\n",
                $unformatted_grand_total);
print "<TD ALIGN = \"center\">\$$subtotal</TD>\n";
print "</TR>\n";
} # End of while (<CART>)
close (CART);
```

The script also outputs the footer.

```
print <<"end_of_html";
</TABLE>
<P>
Grand Total = \$$grand_total
<P>
<INPUT TYPE = "submit" NAME = "submit_deletion"
       VALUE = "Submit Deletion">
<INPUT TYPE = "submit" NAME = "continue_shopping"
       VALUE = "Continue Shopping">
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
```

```
       VALUE = "Return to Front page">
<INPUT TYPE = "submit" NAME = "order_form"
       VALUE = "Checkout Stand">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

On the Web, the delete interface looks like Figure 17.9.

Figure 17.9 *Delete interface.*

Deleting an Item from the Client's Cart

If the client submits items for deletion, the script processes the following routines.

```
if ($form_data{'submit_deletion'} ne "")
{
```

Again, the script checks for valid entries. This time, though, it needs to make sure only that we filter out the extra form keys. It need not make sure that we have a positive integer value.

```
@incoming_data = keys (%form_data);
foreach $key (@incoming_data)
  {
  unless ($key =~ /[\D]/)
    {
    if ($form_data{$key} ne "")
      {
      push (@delete_items, $key);
      }
    } # End of unless ($key =~ /[\D]/...
  } # End of foreach $key (@incoming_data)
```

The script then opens the cart and gets the `$cart_row_number`, `$db_id_number`, and `$old_quantity` as it did for modification.

```
open (CART,
"$cart_directory/$form_data{'cart_id'}.cart")
     || &open_error ("$cart_directory/$form_data{'cart_id'}.cart");
while (<CART>)
{
  @database_row = split (/\|/, $_);
  $cart_row_number = pop (@database_row);
  $db_id_number = pop (@database_row);
  push (@database_row, $db_id_number);
  push (@database_row, $cart_row_number);
  chop $cart_row_number;
  $old_quantity = shift (@database_row);
```

Unlike modification, however, the deletion script only needs to check whether the current database row matches a submitted item for deletion. If it does not match, the script adds it to `$shopper_row`. Otherwise, the row is not added. Thus, all the rows are added except the ones that should be deleted.

```
$delete_item = "";
foreach $item (@delete_items)
  {
  if ($item eq $cart_row_number)
    {
    $delete_item = "yes";
    } # End of if ($item eq $cart_row_number)
  } # End of foreach $item (@add_items)
if ($delete_item ne "yes")
  {
```

```
  $shopper_row .= $_;
  }
} # End of while (<CART>)
close (CART);
```

Then the script overwrites the old cart with the new information and sends the client back to the category page.

```
open (CART, ">
$cart_directory/$form_data{'cart_id'}.cart")
      || &open_error ("$cart_directory/$form_data{'cart_id'}.cart");
print CART "$shopper_row";
close (CART);
&display_cart_contents;
exit;
}
```

Displaying the Order Form

Our hope is that the client fills the cart and heads to the cash register. Online, the cash register is analogous to the online order form.

```
if ($form_data{'order_form'} ne "")
{
```

The script prints an order form by reading the prepared form in the **Html** subdirectory.

```
open (ORDER_FORM, "$html_directory/$order_form")
      || &open_error("$html_directory/$order_form");
while (<ORDER_FORM>)
{
if ($_ =~ /<FORM/)
{
print <<"end_of_html";
<FORM METHOD = "post" ACTION = "$main_script_url">
<INPUT TYPE = "hidden" NAME = "page"
       VALUE = "$form_data{'page'}">
<INPUT TYPE = "hidden" NAME = "cart_id"
       VALUE = "$form_data{'cart_id'}">
<TABLE BORDER = "1"><TR>
end_of_html
```

To make this easier, we have created a subdirectory that contains the order form and front page HTML files. This way, it will be easier to modify them to your own needs. The preceding routine outputs your predesignated order form, but, as we did for the front page, we will insert our hidden variables.

Then the script prints the current cart contents using the previous routines.

```
foreach $field (@display_fields)
  {
  print "<TH>$field</TH>\n";
  }
print "<TH>Quantity</TH><TH>Subtotal</TH></TR>";

open (CART, "$cart_directory/$form_data{'cart_id'}.cart")
      || &open_error("$cart_directory/$form_data{'cart_id'}.cart");
while (<CART>)
  {
  @cart_fields = split (/\|/, $_);
  $quantity = shift(@cart_fields);
  foreach $display_number (@display_numbers)
    {
    if ($cart_fields[$display_number] eq "")
      {
      $cart_fields[$display_number] = "<CENTER>-</CENTER>";
      }
    print "<TD ALIGN
=\"center\">$cart_fields[$display_number]</TD>\n";
    }
  print "<TD ALIGN = \"center\">$quantity</TD>";
  $unformatted_subtotal =
     ($quantity*$cart_fields[$price_number]);
  $subtotal = sprintf ("%.2f\n", $unformatted_subtotal);
  $unformatted_grand_total = $grand_total + $subtotal;
  $grand_total = sprintf ("%.2f\n",
     $unformatted_grand_total);
  print "<TD ALIGN = \"center\">\$$subtotal</TD>";
  print "</TR>";
  } # End of while (<CART>)
 print <<"end_of_html";
</TABLE>
<P><CENTER>
```

```
Grand Total = \$$grand_total
<P></CENTER>
end_of_html
close (CART);
} # End of if ($_ =~ /<FORM/)
```

Once it has output the cart contents, the script continues outputting the predesignated order form.

```
  print "$_";
  } # End of while (<ORDER_FORM>)
close (ORDER_FORM);
exit;
}
```

Finalizing the Order

Finally, the script sends the order to whomever handles the order requests and sends the client a note thanking them for the order.

```
if ($form_data{'send_in_order'} ne "")
{
```

The script begins by sending the header and sorting the incoming form variables. We do this because the Perl `keys` function will not give us the form variables in any specific order due to hash routines. By encoding each form variable in your HTML document with a preceding number, such as `01-name` or `05-city`, you give yourself the ability to sort them in the order you want them to appear.

Take a look at the HTML in the accompanying CD-ROM. You will see that all the `NAME` values are ordered this way. The number corresponds to its order when all the variables are printed. The `sort` function takes all the unsorted variables in `@variable_names`, sorts them by this number, and then adds them to `@sorted_variable_names`.

```
&page_header("Your Order Has Been Sent");
@variable_names = keys (%form_data);
@sorted_variable_names = sort (@variable_names);
```

Once the variables are sorted, the script displays information the client entered on the order form and uses `$email_body` to begin creating the body of the E-mail message that it will send to the store administrator.

```
print "<H2><CENTER>
You submitted the following information</CENTER></H2>";
$email_body .= "Personal Information\n\n";
print "<TABLE>";
```

Then it prints the form variable and its associated value for every form field the client submitted. The script does not print administrative hidden variables used by this script, because the client does not care about them.

If you add other hidden variables, be sure to escape them here.

The script also adds the list to the `$email_body` variable.

```
foreach $variable (@sorted_variable_names)
  {
  if ($form_data{$variable} ne "" &&
      $variable ne "cart_id" &&
      $variable ne "page" &&
      $variable ne "send_in_order")
    {
    print "<TR><TH ALIGN =
    \"left\">$order_form_array{$variable}</TH>\n";
    print "<TD>$form_data{$variable}</TD></TR>\n";
    $email_body .= "$order_form_array{$variable} =
                    $form_data{$variable}\n";
    }
  } # End of foreach $variable (@sorted_variable_names)
print "</TABLE>";
```

Then the current cart contents as ordered are displayed.

```
print "<H2><CENTER>
You ordered the following items</CENTER></H2>";
print "<TABLE BORDER = \"1\"><TR>";
```

```
foreach $field (@display_fields)
  {
  print "<TH>$field</TH>\n";
  }
print "<TH>Quantity</TH><TH>Subtotal</TH></TR>";
```

This time, however, the script does something a little bit different when it comes to mailing the store administrator. First, the script opens the cart again and reads it line by line.

```
$email_body .= "\nOrder Information\n\n";
open (CART,  "$cart_directory/$form_data{'cart_id'}.cart")
      || &open_error ("$cart_directory/$form_data{'cart_id'}.cart");
while (<CART>)
  {
  @cart_fields = split (/\|/, $_);
  $quantity = shift(@cart_fields);
  $cart_quantity = $cart_quantity + $quantity;
```

Because the store administrator does not need to have every single database row included with orders, the script includes only certain essential fields. First, the script notes the quantity of every item.

```
$email_body .= "$quantity";
```

Then, for every item that we have set to display in our setup file with the `@display_numbers` array, the script prints the whole row for the client on the Web.

```
foreach $display_number (@display_numbers)
  {
if ($cart_fields[$display_number] eq "")
  {
  $cart_fields[$display_number] = "<CENTER>-</CENTER>";
  }
print "<TD ALIGN=\"center\">$cart_fields[$display_number]</TD>";
```

But the store administrator receives a different list, which is defined in the setup file (`@email_numbers`). The administrator gets the values corresponding to the elements of `@email_numbers`, a sublist of `@display_numbers`

that includes only those fields that the store administrator needs to see. For example, whereas the client may want to see the description, name, price, item number, size, and picture, the administrator may want to see only the item number, name, and quantity. Thus, the script sends the store administrator only those fields configured in the setup file.

```
foreach $display_field (@email_numbers)
  {
  if ($display_number eq $display_field)
    {
    $email_body .= "    $cart_fields[$display_number]";
    }
  }
} # End of foreach $display_number (@display_numbers)
```

The four spaces before `$cart_fields[$display_number]` are just for formatting, as you will see when you get the E-mail.

The script adds a newline for every cart item.

```
$email_body .= "\n";
```

It then includes the footer information for the client on the Web and calculates costs.

```
 print "<TD ALIGN = \"center\">$quantity</TD>";
 $unformatted_subtotal =
    ($quantity*$cart_fields[$price_number]);
 $subtotal = sprintf ("%.2f\n", $unformatted_subtotal);
 $unformatted_grand_total = $grand_total + $subtotal;
 $grand_total = sprintf ("%.2f\n",
    $unformatted_grand_total);
 print "<TD ALIGN = \"center\">\$$subtotal</TD>";
 print "</TR>";
} # End of while (<CART>)
```

The following routines are cost calculations for the example order form. They demonstrate how shipping costs might be figured.

```
if ($cart_quantity eq "1")
  {
  $shipping_cost = 2.50;
  }
if ($cart_quantity eq "2")
  {
  $shipping_cost = 3.50;
  }
if ($cart_quantity ne "2")
  {
  $cart_quantity = $cart_quantity - 2;
  $unformatted_shipping_cost = "3.50 + $cart_quantity";
  $shipping_cost = sprintf ("%.2f\n",
                   $unformatted_shipping_cost);
  }
  $unformatted_final_total = $grand_total +
                             $shipping_cost;
  $final_total = sprintf ("%.2f\n",
                 $unformatted_final_total);
```

Next, the script prints the footer for both the client and the store administrator, including total price.

```
print <<"end_of_html";
</TABLE>
<P><CENTER>
Item Total = \$$grand_total
<P>
Shipping and Handling: \$$shipping_cost
<P>
Grand Total: \$$final_total
<P>
</CENTER>
end_of_html
close (CART);
$email_body .= "\nGrand Total = \$$final_total\n";
print "$_";
close (ORDER_FORM);
```

Finally, the script uses the `send_mail` routine in **mail-lib.pl** to send E-mail to the store administrator with the client's ordering information.

```
&send_mail("$admin_email","$admin_email",
     "$email_subject", "$email_body");
 exit;
}
```

On the Web, the client receives a page like that shown in Figure 17.10.

Displaying the Products for Sale

If it has advanced through all the preceding `if` tests, the script knows that the client simply wanted to see a page display. So it displays the current items within the page selected by the client.

It uses the `display_items_for_sale` subroutine:

```
else
  {
  &display_items_for_sale;
  exit;
  }
```

Figure 17.10 *A successful order.*

The display_items_for_sale Subroutine

This subroutine outputs the client-selected product page that has been prepared by the store administrator. Using the same filtering method used to output the front page, this routine loads the selected page, formats it with `cart_id` and `page` information, and outputs it to the client on the Web.

```
sub display_items_for_sale
  {
  open (HTML_FILE, "$html_directory/$form_data{'page'}")
      ||
  &open_error("$html_directory/$form_data{'page'}");
  while (<HTML_FILE>)
    {
    s/cart_id =/cart_id = $cart_id/g;
    s/%%cart_id%%/$form_data{'cart_id'}/g;
    s/%%page%%/$form_data{'page'}/g;
    print "$_";
    } # End of while (<HTML_FILE>)
   close (HTML_FILE);
  }
```

The bad_order_note Subroutine

The `bad_order_note` subroutine is used in case the client has typed an invalid character in one of the input fields. After all, we do not want to allow the client to enter a negative quantity and perhaps sneakily affect the grand total!

```
sub bad_order_note
{
&page_header("Woooopsy");
print <<"end_of_html";
<CENTER><H2>Woooopsy</H2></CENTER>
<BLOCKQUOTE>
I'm sorry, it appears that you did not enter a valid numeric quantity
(whole numbers greater than zero) for one or more of the items you
ordered and I am not allowed to modify your cart unless you do so.
Would you try again?  Thanks<P>
<CENTER><INPUT TYPE = "submit" NAME = "try_again"
               VALUE = "Try Again">
</CENTER></BLOCKQUOTE></BODY></HTML>
end_of_html
exit;
}
```

The page_header Subroutine

page_header prints the HTML page header. We pass the routine one argument, which differentiates pages that use this routine when it prints the HTML header.

```
sub page_header
  {
  local($type_of_page) = @_;
  print <<"end_of_html";
  <HTML>
  <HEAD>
  <TITLE>$type_of_page</TITLE>
  </HEAD>
  <BODY>
  <FORM METHOD = "post" ACTION = "$main_script_url">
  <INPUT TYPE = "hidden" NAME = "page"
         VALUE = "$form_data{'page'}">
  <INPUT TYPE = "hidden" NAME = "cart_id"
         VALUE = "$form_data{'cart_id'}">
  end_of_html
  }
```

The display_cart_contents Subroutine

The display_cart_contents subroutine is used by the script to display the current items in the client's cart.

```
sub display_cart_contents
{
```

The subroutine first uses the page_header subroutine to output an HTML header, passing it one parameter relating to how the page title should be modified.

```
&page_header("View/Modify Cart");
```

Next, it begins to create a table to display the current cart contents.

```
print <<"end_of_html";
<CENTER>
```

```
<TABLE BORDER = "1">
<TR>
end_of_html
```

The script then creates the table headers. For every item in our special list of database items to be displayed, the script creates a header cell.

```
foreach $field (@display_fields)
  {
  print "<TH>$field</TH>\n";
  }
```

Now the script adds one cell for quantity and one for subtotal, because the client's cart has those "extra" fields.

```
print "<TH>Quantity</TH>\n<TH>Subtotal</TH>\n</TR>\n";
```

Next, it opens the client's cart again and goes through it one line at a time.

```
open (CART,  "$cart_directory/$form_data{'cart_id'}.cart")
|| &open_error ("$cart_directory/$form_data{'cart_id'}.cart");
while (<CART>)
  {
  @cart_fields = split (/\|/, $_);
  $cart_id_number = pop (@cart_fields);
  $quantity = shift(@cart_fields);
```

`@display_numbers`, defined in the database-specific setup file, gives the script the array numbers associated with the fields to be displayed on this table. `@display_numbers` gets the value associated with the field from the `@cart_fields` array.

```
foreach $display_number (@display_numbers)
{
if ($cart_fields[$display_number] eq "")
     {
     $cart_fields[$display_number] = "<CENTER>-</CENTER>";
     }
print "<TD ALIGN = \"center\">$cart_fields[$display_number]</TD>";
} # End of foreach $display_number (@display_numbers)
```

Using the quantity value it shifted earlier, the script fills the next table cell. After using the `$price_number` variable defined in the setup file, the script calculates the subtotal for that database row and fills the final cell. Next, it closes the table row and, once it has gone all the way through the cart file, also closes it.

```
print "<TD ALIGN = \"center\">$quantity</TD>";
$unformatted_subtotal = ($quantity*$cart_fields[$price_number]);
$subtotal = sprintf ("%.2f\n", $unformatted_subtotal);
$unformatted_grand_total = $grand_total + $subtotal;
$grand_total = sprintf ("%.2f\n", $unformatted_grand_total);
print "<TD ALIGN = \"center\">\$$subtotal</TD>";
print "</TR>";
} # End of while (<CART>)
close (CART);
```

Finally, the script prints the footer.

```
print <<"end_of_html";
</TABLE>
<P>
Grand Total = \$$grand_total
<P>
<INPUT TYPE = "submit" NAME = "change_quantity"
      VALUE = "Change Quantity">
<INPUT TYPE = "submit" NAME = "delete_item" VALUE = "Delete Items">
<INPUT TYPE = "submit" NAME = "continue_shopping"
 VALUE = "Continue Shopping">
<INPUT TYPE = "submit" NAME = "return_to_frontpage"
 VALUE = "Return to Front page">
<INPUT TYPE = "submit" NAME = "order_form" VALUE = "Checkout Stand">
</CENTER></FORM></BODY></HTML>
end_of_html
exit;
}
```

The open_error Subroutine

The `open_error` subroutine is used by this script to provide useful debugging information for the script administrator.

```
sub open_error
{
```

The subroutine begins by assigning the `filename` variable that was sent from the main routine to the local variable, `$filename`.

```
local ($filename) = @_;
```

It then sends back an error message and exits.

```
print "I am really sorry, but for some reason I was unable to open
<P>$filename<P>Please make sure that the filename is correctly defined
in your setup, actually exists, and has the right permissions relative
to the Web browser. Thanks!";
exit;
}
```