The Presentation Layer

The presentation layer of an application that uses multi-tier software architecture is concerned with providing the public fact of that application. It is the programs of the presentation layer that generate interfaces of various kinds.

An interface may be a human interface, such as a command-line syntax, or a graphical user interface (GUI). Most often, these will take the form of hypertext markup language (HTML) pages. Alternately, a GUI can be in the form of some sort of special client that’s particular to your operating system. Either way, it’s important to separate, as fully as possible, the people who do the accessor- and logic-layer work from the designers who concern themselves primarily with the visual appearance of data. Lots of tools exist to help you achieve this goal, notably the PHP-based Smarty templating engine (read its nutshell tutorial here: http://smarty.php.net/crashcourse.php).

Alternately, the presentation layer may provide an interface that’s suitable for interpretation only by a machine. It’s possible that you would create a presentation layer that generates extensible markup language (XML) documents for consumption by some remote computing resource.

In this chapter, we’ll endow Currawong Accounting with a front end in the form of a collection of HTML pages. These documents will deliver information to and extract information (and commands) from the application’s user.

9.1 Frameworks and Resources

In creating an HTML-based presentation layer, we’ll need to create a couple of resources up front. First among these is a basic system of HTML frames, one of which acts as a navigation bar with which to call up the various presentation-layer programs (it also acts as a sort of
"to-do" list as you work toward implementing all of the features you want your application to have). We'll also need to create a library that makes it easier to include standard HTML features, such as list boxes containing the names of all recorded payees, in our pages.

9.1.1 An HTML Display Framework

The first thing we're going to need is an HTML framework in which to display our information. The basic design is a single window with two frames: a small navigation strip down the left side and the remainder of the window for displaying the results of PHP programs. Setting this up isn't hard; the job requires only a single master framing document, a links document to fill the navigation strip, and a default page that displays when the master framing document is loaded (i.e., before any links in the navigation strip have been clicked).

The master framing document is called app.html, and it looks like this:

app.html

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Currawong Accounting</title>
</head>
<frameset COLS="30%,*">
<frame name="controls" src="acctLinks.html">
<frame name="display" src="welcome.html">
</frameset>
</html>
```

The key bit is the frameset element, which defines two columnar frames in a 30–70 allocation. Then, it defines the two frames themselves, giving them each a name and some initial contents: acctLinks.html and welcome.html. It's important to note that the name of the right-hand frame is display, because we'll have to make PHP programs render there in a moment.

With those columns established, let's look at the documents they're defined as containing. Here's acctLinks.html:

acctLinks.html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Currawong Accounting</title>
</head>
<body>

<H4>Currawong Accounting</H4>

<P> <A HREF="welcome.html" TARGET="display">Home</A></P>

<P>Transactions</P>
<UL>
  <LI><A HREF="/acct/presentation/enterTransaction.php" TARGET="display">Add a Transaction</A>
  <LI><A HREF="/acct/presentation/viewTransactions.html" TARGET="display">View/Edit/Delete Transactions</A>
</UL>

<P>Reporting</P>
<UL>
  <LI><A HREF="/acct/bl/blAccountsReport.php" TARGET="display">Table: Summary of Accounts</A>
  <LI><A HREF="/acct/presentation/viewBarGraphSingleAccountWeekly.php" TARGET="display">Graph: Single Account (Native Currency)</A>
  <LI><A HREF="/acct/presentation/viewBarGraphSingleAccountWeeklyWithMA.php" TARGET="display">Graph: Single Account (Native Currency) with Moving Average</A>
  <LI><A HREF="/acct/presentation/viewBarGraphMultiAccountWeeklyUSD.php" TARGET="display">Graph: Several Accounts (USD)</A>
</UL>

<P> View/Add/Edit</P>
<UL>
  <LI> <A HREF="/acct/presentation/enterAccount.php" TARGET="display">View/Add/Edit Accounting Category</A>
  <LI> <A HREF="/acct/presentation/enterBankAccount.php" TARGET="display">View/Add/Edit Bank Account</A>
</UL>
TARGET="display">View/Add/Edit Bank Account</A>  
<LI> <A HREF="/acct/presentation/enterAcctType.php" TARGET="display">View/Add/Edit Bank Account Type</A>  
<LI> <A HREF="/acct/presentation/enterCurrency.php" TARGET="display">View/Add/Edit Currency</A>  
<LI> <A HREF="/acct/presentation/enterInstitution.php" TARGET="display">View/Add/Edit Institution</A>  
<LI> <A HREF="/acct/presentation/enterPayee.php" TARGET="display">View/Add/Edit Payee</A>  
<LI> <A HREF="/acct/presentation/enterTransType.php" TARGET="display">View/Add/Edit Transaction Type</A>  
</UL>  
<P> Remote Updates  
<UL>  
<LI> <A HREF="/acct/elsewhere/updateCurrencies.php" TARGET="display">Update Currencies</A>  
</UL>  
<br><br><br><br><br> <center>  
<IMAGE SRC="littleCurrawong.jpg">  
</center>  
</body>  
</html>  

Again, not even close to rocket science. The only interesting bits (and that is only if you're kind of out of touch with HTML) are the TARGET attributes of the A elements (the links). Notice that each of the links looks like this example:

<A HREF="/acct/presentation/enterPayee.php" TARGET="display">View/Add/Edit Payee</A>  

The TARGET="display" part of that line tells the browser that the document to which the link points should not be rendered in the frame that contains the link (the navigation strip), but rather in the frame named display. We defined the larger frame as having that name in the master framing document (app.html). It's a simple but important effect.  

By default (because it's specified in app.html), the large frame contains a file called welcome.html. This is essentially the splash screen of Currawong Accounting, containing
the applications name and a photo of Strepera graculina, the currawong. It's not a complex HTML file:

```
welcome.html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Currawong Accounting</title>
</head>
<body>
<center>
<h2>Currawong Accounting</h2>
<h3>Multi-Currency Bookkeeping for Small Business</h3>
<img src="currawong.jpeg">
</center>
</body>
</html>
```

Figure 9.1 illustrates the default appearance of Currawong Accounting.

### 9.1.2 Generating List Boxes

In generating HTML pages on the presentation layer, there will be a number of points at which we'll want to generate lists representing the contents of database tables. When presenting the user with the opportunity to record a transaction, for example, there will need to be list boxes containing the names of all existing bank accounts, all existing payees, and so on.

In HTML, list boxes look like this:

```
<select name='bankAccount'>
<option value='1'>Middleburg Bank Savings</option>
<option value='2'>Business A</option>
<option value='3'>Business B</option>
```
Figure 9.1: Currawong Accounting looks like this when it's first opened.

```html
<option value='4'>Lloyds Retirement</option>
</select>
```

That's an actual example of a list box that's required for the Currawong Accounting presentation layer. Key parts of it include:

- The name, which in this case is "institution." A name allows JavaScript to manipulate the list box, among other things.
- The text enclosed by the option tags (such as "Middleburg Bank"), which is actually visible in the list box when it's rendered.
- The value attributes of the option elements, which are what can be manipulated programmatically and what goes to the server when the form is submitted.

In the case of Currawong Accounting, the text enclosed by option tags represents a value (usually the name or description column) from a row in a database table. The value attribute of the same option is the value of the primary key column of the same row (the primary key column is always the id column, in the database we set up in Chapter 6). That way, when the form is submitted, there's no need to resolve the name (or description) back to the id value. There's one less place for a mistake to appear.

In some cases, we'll want an option other than the first to be preselected. This will be the case, for example, when offering users the opportunity to edit something that already exists. In editing a transaction, say, the user should be able to change the account on which
a check was drawn (it may have been entered in error originally), but should be able to see the details of the transaction as they were originally recorded). The HTML for this is straightforward:

```html
<select name='bankAccount'>
<option value='1'>Middleburg Bank Savings</option>
<option value='2'>Business A</option>
<option value='3' SELECTED>Business B</option>
<option value='4'>Lloyds Retirement</option>
</select>
```

In that list box, the Hang Seng Bank option is shown selected when the list box is originally rendered.

A third possibility is a list box in which multiple elements may be selected. Such a list box looks like this:

```html
<select MULTIPLE name='bankAccount'>
<option value='1'>Middleburg Bank Savings</option>
<option value='2'>Business A</option>
<option value='3' SELECTED>Business B</option>
<option value='4'>Lloyds Retirement</option>
</select>
```

The MULTIPLE attribute makes it possible to choose multiple elements.

For the Currawong application, we need a way to quickly generate HTML code like the above. We want to be able to import a library into our presentation-layer programs and use lines like this:

```javascript
menuBankAccount('bankAccounts', 3, 0);
```

That would cause a list box of all existing bank accounts to be generated. It would have a name attribute of bankAccount, and the option with a value attribute equal to 3 would be shown selected. The final parameter, 0, indicates that it should not have a MULTIPLE attribute (a 1 in place of the 0 would indicate that a MULTIPLE attribute was required). In other words, we'd get precisely the same HTML code as was shown earlier:

```html
<select name='bankAccount'>
<option value='1'>Middleburg Bank Savings</option>
<option value='2'>Business A</option>
<option value='3' SELECTED>Business B</option>
<option value='4'>Lloyds Retirement</option>
</select>
```

By the way, if you don't want a particular option preselected, you could call the function like this:

```javascript
menuBankAccount('bankAccounts', 0, 0);
```
The library containing the code for generating selection lists is called listboxes.php. It contains seven functions, listed here:

- `menuBankAccounts`
- `menuAccount`
- `menuPayees`
- `menuTransTypes`
- `menuInstitutions`
- `menuAcctTypes`
- `menuCurrencies`

Most take two arguments (a name value and the id value of the element to be pre-selected) and all, except `menuBankAccounts()`, return selection lists with the value attribute equal to the id column and the visible text equal to the name column. In the case of `menuBankAccounts()`, the rendered text comes from the description column, which allows for the possibility that the user would have multiple accounts at the same bank. It's also the case that `menuBankAccounts()` takes a third parameter to allow the inclusion of the `MULTIPLE` attribute.

Because listboxes.php will have to make reference to the accessor layer, it has to import the usual libraries:

```php
require_once('nusoap-0.6/nusoap.php');
require_once('configuration.php');
```

After that, the functions are defined. A typical example is `menuBankAccounts()`:

```php
function menuBankAccounts($listName, $selectedIndex, $multiple) {

    global $accessorHost;

    $serviceURL = 'http://' . $accessorHost . '/acct/accessor/getBankAccounts.php';
    $functionName = "getBankAccounts";

    $soapclient = new soapclient($serviceURL);

    $parameters = array();

    $result = $soapclient->call($functionName, $parameters);

    $datatype = gettype($result);

    if ($multiple==1)
{  
echo "<select name='$listName' MULTIPLE>
";  
}  
else  
{  
echo "<select name='$listName'>
";  
}  

foreach ($result as $key => $subarray)  
{  
$id = $subarray['id'];  
$identifier = $subarray['description'];  
if ($id == $selectedIndex)  
{  
echo "<option value='$id' selected>$identifier</option>
";  
}  
else  
{  
echo "<option value='$id'>$identifier</option>
";  
}  

}  
echo "</select>";
}  

That function makes a call to a program on the accessor layer (getBankAccounts.php, in this case) and receives an array back. The array contains the contents of the ACCT_bankAccounts table (see Chapter 6 for more about the design of the database). The following line picks out each returned row as an array, and the code contained in its block pulls out the required elements by name.

foreach ($result as $key => $subarray)  
{  
$id = $subarray['id'];  
$identifier = $subarray['description'];  
if ($id == $selectedIndex)  
{  
echo "<option value='$id' selected>$identifier</option>
";  
}  
else  
{  
echo "<option value='$id'>$identifier</option>
";  
}  

}  
echo "</select>";
}  

An if-else block determines if the element is one that's meant to be shown preselected, and the end result is that the HTML that represents the selection list is written with a series of echo commands.

The other six functions lack the capacity to add a MULTIPLE attribute to the generated list box. Function menuAccount() is typical:

function menuAccount($listName, $selectedIndex)  
{  

global $accessorHost;
```php
$serviceURL = "http://" . $accessorHost . "/acct/accessor/getAccounts.php";
$functionName = "getAccounts";

$soapclient = new soapclient($serviceURL);

$parameters = array();

$result = $soapclient->call($functionName,$parameters);

$datatype = gettype($result);

echo "<select name="$listName">
";

foreach ($result as $key => $subarray)
{
    $id = $subarray['id'];
    $identifier = $subarray['name'];
    if($id == $selectedIndex)
    {
        echo "<option value='$id' selected>$identifier</option>
";
    }
    else
    {
        echo "<option value="$id" >$identifier</option>
";
    }
}

echo "";
```

The other five functions in listboxes.php are virtually identical to menuAccount(), and so are not listed here.

### 9.2 Viewing and Adding—Everything but Transactions

A key functional element of Currawong Accounting is the ability to view information stored in the database. The user needs to be able to extract lists of existing (i.e., already stored) currencies, payees, bank accounts, institutions, and the other business entities that are
involved in managing the bookkeeping system. The user also has to be able to add to these lists, adding new payees, currencies, and so on. In other words, the user needs to be able to insert new rows into the tables in the database.

Specifically, we need user interfaces for the following purposes:

- Viewing and adding accounting categories,
- Viewing and adding bank accounts,
- Viewing and adding bank account types,
- Viewing and adding currencies,
- Viewing and adding institutions,
- Viewing and adding payees, and
- Viewing and adding transaction types.

As it happens, the programs in this category are very similar to one another. For that reason, this book does not include full listings of each one. There is a full listing of enterAccount.php, which is the presentation-level program used to view accounting categories, add new ones, and spawn the subwindow with which you edit existing categories. There’s a full commentary on how enterAccount.php works, as well.

For the other programs in this section, you’ll find only the parts that are significantly different (meaning, not the decorative HTML that surrounds the PHP code) listed here.

There is also a need for an interface with which to view and add transactions. However, because that requirement is handled differently, it’s covered in a separate section at the end of this chapter.

9.2.1 Viewing and Adding Accounting Categories

The same presentation-layer program handles viewing and adding accounting categories. It’s contained in a file called enterAccount.php.

Here’s a listing of that file:

```php
<html>
<head>
<title>Enter Account</title>
<SCRIPT LANGUAGE="JavaScript">

function openEditWindow(id)
{
```
var url = "editAccount.php?id" + id;

var newWindow = window.open(url, "child", "HEIGHT=200, WIDTH=300");

</SCRIPT>
</head>

<body>

<H1>Enter Accounting Category</H1>

require_once('listboxes.php');
require_once('nusoap-0.6/nusoap.php');
require_once('configuration.php');

$parameters = array();

$soapclient = new soapclient('http://' . $accessorHost . '/acct/accessor/getAccounts.php');

$result = $soapclient->call('getAccounts',$parameters);

 echo "<P><B>Existing Accounts</B><BR>

echo '<TABLE BORDER="1" CELLPADDING="5">

foreach ($result as $key => $subarray)
{
  echo '<TR>' ;
  echo '<TD>' ;
  echo $subarray['name'] ;
  echo '</TD>' ;
  echo '</TR>' ;
}
In large part, this program is concerned with generating table code. The table is filled with data retrieved directly from the accessor layer, specifically getAccounts.php. The first interesting bit of code in the presentation-layer program presented here has to do with contacting getAccounts.php:

```php
$parameters = array();
$soapclient = new soapclient('http://' . $accessorHost .
```

Those three lines of code create a simple object access protocol (SOAP) client object (which is possible because the NuSOAP library is imported), which accesses the getAccounts.php file and sends it $parameters, an empty array (the accessor-layer program is defined as taking no parameters, as discussed in Chapter 7). After the call, $result contains a two-dimensional array representing the output of getAccounts().

That variable, $results, is used to generate the table of existing accounts. By using this construction:

```php
foreach ($result as $key => $subarray) {
    echo '<TR>';
    echo '<TD>' . $subarray['name'] . '</TD>';
    echo '<TD>' . '<INPUT TYPE="BUTTON" onClick="openEditWindow(' . $subarray['id'] . ')" VALUE="Edit">' . '</TD>' . '</TR>';
}
```

In the loop, $subarray['name'] represents the contents of the name element in that particular row (see the accessor-layer program that created the array in the first place for further explanation). The name element from each returned row gets inserted into a table.

The other interesting line in that block is this one:

```php
echo '<INPUT TYPE="BUTTON" onClick="openEditWindow(' . $subarray['id'] . ')" VALUE="Edit">';
```

It puts an Edit button in the table next to a row of data. When clicked, it calls a JavaScript function called openEditWindow(). To that function, it sends a single parameter: The value of the id column of the current row. That way, the JavaScript function knows what element it's working with.

The JavaScript function, openEditWindow(), is short. It looks like this:

```javascript
function openEditWindow(id) {
    var url = "editAccount.php?rowToEdit=" + id;
    var newWindow = window.open(url, "child", "HEIGHT=200,WIDTH=300");
}
```
It takes the id value as an argument (note that JavaScript variables aren't preceded by $) and assembles a string using it. The string looks like this (using the id value 3 as an example):

`editAccount.php?rowToEdit=3`

That's a hypertext transport protocol (HTTP) GET statement. Indeed, this uniform resource locator (URL), when invoked by the final line of `openEditWindow()`, opens a new window containing HTML generated by `editAccount.php` (which is discussed in the editing section later in this chapter). That last line, by the way, opens a new window (using the `window.open` method) and calls the new window child. It's necessary to give the new window a name in JavaScript. Note also the unusual syntax of the HEIGHT and WIDTH specifications. These are different for the various child windows covered later in this chapter.

The second form generated by `enterAccount.php` is used to add a new row to the ACCT_accounts table, by way of `insertAccount.php` on the accessor layer. The call is made, however, to the business-logic layer:

```
    echo '<FORM name="addAcct" METHOD="POST"
ACTION="http://'.$blHost.'/acct/bl/blEnterAccount.php">';
```

In Chapter 8, you saw that `blEnterAccount.php` receives the HTTP POST packet generated by the Submit button of this form and passes it on to the accessor layer via SOAP. Figure 9.2 shows `enterAccount.php`.

**Figure 9.2:** Viewing and adding accounting categories.
9.2.2 Viewing and Adding Bank Accounts

The presentation-layer work of viewing and adding bank accounts takes place in enterBankAccount.php. Its design is very similar to that of enterAccount.php. However, the Web service call is different:

```php
$parameters = array();
$soapclient = new soapclient("http://" . $accessorHost . "/acct/accessor/getBankAccounts.php");
$result = $soapclient->call('getBankAccounts', $parameters);
```

Also, the table-generation code is a bit more elaborate, because more pieces of data come back from the accessor layer:

```php
foreach ($result as $key => $subarray) {
    echo '<TR>' .
    echo '  <TD>' .
    echo $subarray['institution'] .
    echo '  </TD>' .
    echo '  <TD>' .
    echo $subarray['number'] .
    echo '  </TD>' .
    echo '  <TD>' .
    echo $subarray['description'] .
    echo '  </TD>' .
    echo '  <TD>' .
    echo $subarray['currency'] .
    echo '  </TD>' .
    echo '  <TD>' .
    echo $subarray['type'] .
    echo '  </TD>' .
    echo '  <TD>' .
    echo '<INPUT TYPE="BUTTON" onClick="openEditWindow( . $subarray['id'] . ")" VALUE="Edit">' .
    echo '</TD>' .
    echo '</TR>' .
}
```

The table is just more of the same, though—all of the required returned elements are accessed as $subarray['columnName']. Plus, the "add new" form is submitted to a different program on the business-logic layer:

```html
    echo "<FORM name='addBankAccount' METHOD='POST' ACTION='http://" . $blHost . "/acct/bl/blEnterBankAccount.php'>";
```

Figure 9.3 shows enterBankAccount.php, fully rendered.
9.2.3 Viewing and Adding Bank Account Types

The presentation-layer program involved with viewing and adding bank account types is enterAcctType.php. It's similar in design to other programs covered in more detail earlier in this section.

The call to the accessor layer (to retrieve rows for display) looks like this:

```php
$parameters = array();
$soapclient = new soapclient('http://'. $accessorHost .
   '/acct/accessor/getAcctTypes.php');
$result = $soapclient->call('getAcctTypes',$parameters);
```

The table-generation code looks like this, involving only one column from the database:

```php
foreach ($result as $key => $subarray) {
    echo '<TR>';
    echo '<TD>'; echo $subarray['name']; echo '</TD>';
    echo '<TD>'; echo '<INPUT TYPE="BUTTON" onClick="openEditWindow(' . $subarray['id'] . '"
        VALUE="Edit">'; echo '</TD>';
}
```
Finally, the "add" form calls blEnterAccountType.php:

```php
    echo '<FORM name="addTransType" METHOD="POST"
ACTION="http://' . $blHost . '/acct/bl/blEnterAcctType.php">';
```

Figure 9.4 shows enterAcctType.php in action.

### 9.2.4 Viewing and Adding Currencies

The code for viewing and adding currencies is contained in enterCurrency.php. Its call to the accessor layer is unique:

```php
    $parameters = array();
    $soapclient = new soapclient('http://' . $accessorHost . '/acct/accessor/getCurrencies.php');
    $result = $soapclient->call('getCurrencies',$parameters);
```

Its table-generator uses standard techniques to generate a custom look:

```php
    foreach ($result as $key => $subarray) {
        echo '<TR>';
        echo '<TD>';
        echo $subarray['abbreviation'];
        echo '</TD>';
    }
```
Finally, the "Add Currency" form submits to a special program on the business layer:

```php
echo '<FORM name="addCurrency" METHOD="POST" ACTION="http://' . $blHost . '/acct/bl/blEnterCurrency.php">';
```

Figure 9.5 shows enterCurrency.php as it fits into the rest of the Currawong Accounting interface.

### 9.2.5 Viewing and Adding Institutions

The presentation-layer code for viewing and adding financial institutions is contained in enterInstitution.php. Its call to the accessor layer refers to getInstitutions.php:

```php
$parameters = array();
$soapclient = new soapclient("http://" . $accessorHost . "/acct/accessor/getInstitutions.php");
$result = $soapclient->call('getInstitutions', $parameters);
```

The institution-related table-generation code is fairly complicated, but it uses the standard echo $subarray['columnName'] technique repeatedly:

```php
foreach ($result as $key => $subarray) {
    echo '<TR>';
    echo '<TD>';
    echo $subarray['name'];
    echo '</TD>';
    echo '<TD>';
    echo $subarray['xRate'];
    echo '</TD>';
    echo '<TD>';
    echo $subarray['updated'];
    echo '</TD>';
    echo '<TD>';
    echo '<INPUT TYPE="BUTTON" onClick="openEditWindow(' . $subarray['id'] . ') " VALUE="Edit">';
    echo '</TD>';
    echo '</TR>';
}
```
**Figure 9.5:** Viewing and adding currencies.

```php
echo '</TD>';
echo '<TD>';
echo $subarray['streetAddress'];
echo '</TD>';
echo '<TD>';
echo $subarray['city'];
echo '</TD>';
echo '<TD>';
echo $subarray['state'];
echo '</TD>';
echo '<TD>';
echo $subarray['postcode'];
echo '</TD>';
echo '<TD>';
echo $subarray['country'];
echo '</TD>';
echo '<TD>';
echo '<INPUT TYPE="BUTTON" onClick="openEditWindow(' . $subarray['id'] . '')" VALUE="Edit">';
echo '</TD>';
echo '</TR>';
}
```
9.2 Viewing and Adding—Everything but Transactions

![Image of a web page with a table and form]

*Figure 9.6: Viewing and adding institutions.*

Its "Add Institution" form submits to blEnterInstitution.php:

```php
    echo '<FORM name="addInstitution" METHOD="POST" ACTION="http://'.$blHost.'/acct/bl/blEnterInstitution.php">';
```

Figure 9.6 shows enterInstitution.php ready for use.

### 9.2.6 Viewing and Adding Payees

The job of viewing and adding payees on the presentation layer falls to enterPayee.php. Its SOAP client looks like this:

```php
    $parameters = array();
    $soapclient = new soapclient("http://".$accessorHost.
        "/acct/accessor/getPayees.php");
    $result = $soapclient->call('getPayees',$parameters);
```

Its table generator is moderately long but not complex:

```php
    foreach ($result as $key => $subarray) {
        echo '<TR>
            echo '<TD>';
            echo '<TD>';
            echo $subarray['name'];
```
And, its "Add Payee" form submits to a specialized program on the business-logic layer:

```
    echo "<FORM name='addPayee' METHOD='POST' ACTION='http://" . $blHost . "acct/bl/blEnterPayee.php'>";
```

Figure 9.7 shows enterPayee.php at work within the rest of the bookkeeping application.

### 9.2.7 Viewing and Adding Transaction Types

The portion of the presentation layer concerned with viewing and adding transaction types is contained in enterTransType.php. The call to the accessor layer (to retrieve existing transaction types) looks like this:

```
$parameters = array();
$soapclient = new soapclient("http://" . $accessorHost . "/acct/accessor/getTransTypes.php");
$result = $soapclient->call('getTransTypes',$parameters);
```

The code that displays the retrieved transaction types is simple:

```
foreach ($result as $key => $subarray)
{
    echo '<TR>';
```
9.3 Editing—Everything but Transactions

In the programs having to do with viewing and adding business entities (dealt with in the preceding section), you saw that each view/add page included an Edit button for each
Figure 9.8: Viewing and adding transaction types.

Element extracted from the database. The Edit buttons, in each case, triggered a JavaScript function, which in turn spawned a child window of some specified size. The child windows displayed the results of PHP programs. Those PHP programs are the subject of this discussion.

The editing programs on the presentation layer are concerned with extracting a particular row from a particular table in the database, and presenting the data in that row in a way that makes editing possible.

There are seven editing programs, each concerned with a different table. They correspond exactly to the seven view/add programs. They are:

1. Editing accounting categories,
2. Editing bank accounts,
3. Editing bank account types,
4. Editing currencies,
5. Editing institutions,
6. Editing payees, and
7. Editing transaction types.

Because the seven programs are so similar, this section explains only the first in detail, then highlights the slight differences of the others.
9.3.1 Editing Accounting Categories

In editing accounting categories, Currawong Accounting is concerned with extracting the values contained in a specified row of the ACCT_accounts table. The whole idea of multi-tier software design is that we don’t need to know the name of the table, so we refer only to the relevant accessor-layer program: getAccounts.php.

The work of extracting a database row, displaying it in a useful form, and allowing the user to submit a change is handled by editAccount.php. Here is a full listing:

```php
<?php

$parameters = array();

foreach ($_GET as $key => $value)
{
    $parameters[$key] = $value;
}

$soapclient = new soapclient('http://' . $accessorHost .
    '/acct/accessor/getSpecifiedAccount.php');

$result = $soapclient->call('getSpecifiedAccount',$parameters);

echo '<?FORM name="addTransType" METHOD="POST"
ACTION="http://' . $blHost . '/acct/bl/blUpdateAccount.php">

});
```
echo '<P>Modify Existing Accounting Account';
echo "\n";
echo '<P><INPUT TYPE="HIDDEN" NAME="id" VALUE="" .
$result[0][\'id\'] . '" SIZE=15>';
echo "\n";
echo '<P>Name:<INPUT TYPE="TEXT" NAME="name" VALUE="" .
$result[0][\'name\'] . '" SIZE=15>';
echo "\n";
echo '<P><INPUT TYPE="SUBMIT" NAME="submit" VALUE="Submit">';
echo "\n";
echo '</FORM>';

?>
</body>
</html>

Remember that editAccount.php is called with a full HTTP GET statement, such as this:

editAccount.php?rowToEdit=3

That means that the rowToEdit value is available within the program. Depending on which version of PHP Currawong is running under, the value submitted as part of the GET statement is accessible in one of two ways:

1. As $_GET[\'rowToEdit\']. This works only under PHP 4.3 and newer. Note that the $_GET array is a "superglobal" array that does not need to be declared in order to be available within functions.

2. As an ordinary global variable with a name that matches the value in the GET statement: $rowToEdit.

If you're using a version of PHP prior to version 4.3, you will need to set register_globals equal to On in php.ini, and edit the code so that it refers to the GET variables by name rather than through $_GET. It's probably easier to upgrade, if you can.

The $_GET array is used to populate the $parameters array, which is used in a call to the accessor layer:

$parameters = array();

foreach ($_GET as $key => $value) {
    $parameters[$key] = $value;
}
$soapclient = new soapclient('http://'. $accessorHost . '/acct/accessor/getSpecifiedAccount.php');

$result = $soapclient->call('getSpecifiedAccount',$parameters);

With that call made, $result holds a two-dimensional array of only one row—it's effectively an array containing a single subarray. That's why the form-generation code pulls values from it like this:

```php
echo '<P><INPUT TYPE="HIDDEN" NAME="id" VALUE="'. $result[0]['id'] . '" SIZE=15>;
```

All results are referred to as $result[0][\'columnName\']. Note also that this form submits to a program on the business-logic layer that's specific to updating accounts:

```php
echo '<FORM name="addTransType" METHOD="POST"
ACTION="http://'. $blHost . '/acct/bl/blUpdateAccount.php">';
```

There's a screen shot of editAccount.php in Figure 9.9.

### 9.3.2 Editing Bank Accounts

For editing bank accounts, Currawong Accounting relies on editBankAccount.php. It's very similar to editAccount.php. Its call to the accessor layer looks like this:

```php
$parameters = array();
```
foreach ($_GET as $key => $value) {
    $parameters[$key] = $value;
}

$soapclient = new soapclient('http://'. $accessorHost . '/acct/accessor/getSpecifiedBankAccount.php');

$result = $soapclient->call('getSpecifiedBankAccount',$parameters);

Its table-generation code looks like this:

```php
echo '<TR>';
echo '<TD>';
menuInstitutions("institution",$result[0]["institutionId"]);
echo '</TD>';
echo '<TD>';
echo '<INPUT TYPE="TEXT" NAME="number" VALUE="'. $result[0]["number"] .'" SIZE=20>}';
echo '</TD>';
echo '<TD>';
echo '<INPUT TYPE="TEXT" NAME="description" VALUE="'. $result[0]["description"] .'" SIZE=30>';
echo '</TD>';
echo '<TD>';
menuCurrencies("currency",$result[0]["currencyId"]);
echo '</TD>';
echo '<TD>';
menuAcctTypes("type",$result[0]["acctTypeId"]);
echo '</TD>';
echo '</TR>';
```

The interesting elements of the table-generation code are the calls to the functions in the listbox.php library. These make use of the second argument to specify which element in the generated list box should be shown and preselected. A typical example is this:

```php
menuAcctTypes("type",$result[0]["acctTypeId"]);
```

That tells the menuAcctTypes function (in listboxes.php) to generate a list of available account types, with the account type whose id value is equal to $result[0]["acctTypeId"] selected. That is, the list box should show the current bank account’s account type preselected. Figure 9.10 shows how this editing script looks in the wild.
9.3.3 Editing Bank Account Types

For editing bank account types, users of Currawong accounting use editAcctType.php. It works like most of the other editing programs. Its call to the accessor layer, in which an HTTP GET value is used as a parameter, looks like this:

```php
$parameters = array();

foreach ($_GET as $key => $value)
{
    $parameters[$key] = $value;
}

$soapclient = new soapclient('http://' . $accessorHost . '/acct/accessor/getSpecifiedAcctType.php');

$result = $soapclient->call('getSpecifiedAcctType', $parameters);
```

Its table-generation code looks like this:

```php
echo '<FORM name="addTransType" METHOD="POST" ACTION="http://' . $blHost . '/acct/bl/blUpdateAcctType.php">' ;
echo "\n" ;
echo '<P>Modify Existing Account Type';
echo "\n" ;
echo '<P><INPUT TYPE="HIDDEN" NAME="id" VALUE="'. $result[0]["id"] . '" SIZE=15>' ;
echo "\n" ;
echo 'Name:<INPUT TYPE="TEXT" NAME="name" VALUE="'. $result[0]["name"] . '" SIZE=15>' ;
echo "\n" ;
```

Figure 9.10: Editing bank accounts.
Figure 9.11: Editing bank account types.

```php
echo '<p><INPUT TYPE="SUBMIT" NAME="submit" VALUE="Submit">';
echo "\n";
echo '</FORM>'; 
```

Note that it submits the form contents to blUpdateAcctType.php, a special-purpose program on the business-logic layer. Figure 9.11 shows how this program looks when rendered in a browser window.

### 9.3.4 Editing Currencies

The program contained in editCurrency.php handles the work of modifying the characteristics of a currency. Functionally, this program is much the same as others.

To retrieve details from the accessor layer, the program makes a SOAP client call:

```php
$parameters = array();

foreach ($_GET as $key => $value) {
    $parameters[$key] = $value;
}

$soapclient = new soapclient('http://' . $accessorHost . '/acct/accessor/getSpecifiedCurrency.php');

$result = $soapclient->call('getSpecifiedCurrency',$parameters);```
It then uses the results, literally, the array $result, to generate a form containing the existing values:

```php
echo '<FORM name="editCurrency" METHOD="POST"
ACTION="http://'. $blHost . '/acct/bl/blUpdateCurrency.php">';
echo '<TR>';
echo '<TD>' . $result[0]['abbreviation'] . '</TD>' . 
echo '<TD>' . $result[0]['country'] . '</TD>' . 
echo '<TD>' . $result[0]['name'] . '</TD>' . 
 echo '<TD>' . $result[0]['xRate'] . '</TD>' . 
echo '</TR>' . 
```

Figure 9.12 shows how Microsoft Internet Explorer renders this program.

### 9.3.5 Editing Institutions

To edit the attributes of an institution that's already been recorded in the database, Currawong Accounting uses editlnstitution.php. Like other editing programs, editlnstitution.php makes a call to the accessor layer:

```php
$parameters = array();
```
foreach ($_GET as $key => $value)
{
    $parameters[$key] = $value;
}

$soapclient = new soapclient('http://' . $accessorHost . '/acct/accessor/getSpecifiedInstitution.php');

$result = $soapclient->call('getSpecifiedInstitution', $parameters);

It then generates a form in a table, using the contents of $result to populate the fields with existing values:

    echo "<FORM name='modifyExistingInstitution' METHOD='POST'
    ACTION='http://' . $blHost . '/acct/bl/blUpdateInstitution.php'>";

    echo '<TR>,'
    echo '<TD>',
    echo '<INPUT TYPE="TEXT" NAME="name" VALUE="' . $result[0]['name'] . '" SIZE=15>',
    echo '</TD>,'
    echo '<TD>',
    echo '<INPUT TYPE="TEXT" NAME="streetAddress" VALUE="' . $result[0]['streetAddress'] . '" SIZE=20>',
    echo '</TD>,'
    echo '<TD>',
    echo '<INPUT TYPE="TEXT" NAME="city" VALUE="' . $result[0]['city'] . '" SIZE=20>',
    echo '</TD>,'
    echo '<TD>',
    echo '<INPUT TYPE="TEXT" NAME="state" VALUE="' . $result[0]['state'] . '" SIZE=10>',
    echo '</TD>,'
    echo '<TD>',
    echo '<INPUT TYPE="TEXT" NAME="postcode" VALUE="' . $result[0]['postcode'] . '" SIZE=10>',
    echo '</TD>,'
    echo '<TD>',
    echo '<INPUT TYPE="TEXT" NAME="country" VALUE="' . $result[0]['country'] . '" SIZE=10>',
    echo '</TD>,'
    echo '</TR>';

Figure 9.13 shows how this looks when rendered in a browser.
9.3.6 Editing Payees

The code in editPayee.php is involved in editing existing payees. Its call to the accessor layer looks like this:

```php
$parameters = array();

foreach ($_GET as $key => $value)
{
    $parameters[$key] = $value;
}

$soapclient = new soapclient('http://' . $accessorHost . '/acct/accessor/getSpecifiedPayee.php');

$result = $soapclient->call('getSpecifiedPayee',$parameters);
```

Its form-generation code looks like this:

```html
echo "<FORM name='modifyExistingPayee' METHOD='POST' ACTION='http://' . $blHost . '/acct/bl/blUpdatePayee.php'>";

echo '<P><INPUT TYPE="HIDDEN" NAME="id" VALUE="' . $result[0]['id'] . '" SIZE=15>' ;

<TR>
TD>

<INPUT TYPE="TEXT" NAME="name" VALUE="' . $result[0]['name'] . '" SIZE=15>'

</TD>' ;

</TR>' ;

echo '</TD>' ;
```

**Figure 9.13:** Editing institutions.
When rendered in a browser, editPayee.php looks as shown in Figure 9.14.

9.3.7 Editing Transaction Types

To give the user the ability to edit existing transaction types, Currawong Accounting provides editTransType.php. It makes a SOAP call to getSpecifiedTransType.php on the accessor layer:

```php
$parameters = array();
```
foreach ($_GET as $key => $value)
{
    $parameters[$key] = $value;
}

$soapclient = new soapclient('http://'. $accessorHost . '/acct/accessor/getSpecifiedTransType.php');

$result = $soapclient->call('getSpecifiedTransType', $parameters);

It also generates a form containing the returned values and allows the user to submit changes to a dedicated program on the business-logic layer:

echo '<FORM name="addTransType" METHOD="POST" ACTION="http://'. $blHost . '/acct/bl/blUpdateTransType.php">';
echo '\n';
echo '<P>Modify Existing Transaction Type';
echo '\n';
echo '<P><INPUT TYPE="HIDDEN" NAME="id" VALUE="'. $result[0]['id'] . '" SIZE=15>,'
    echo '\n';
echo '<P>Name:<INPUT TYPE="TEXT" NAME="name" VALUE="'. $result[0]['name'] . '" SIZE=15>,'
    echo '\n';
echo '<P><INPUT TYPE="SUBMIT" NAME="submit" VALUE="Submit">','
    echo '\n';
echo '</FORM>);

Rendered, editTransType.php is shown in Figure 9.15.

![Edit Transaction Type](https://example.com/edit_trans_type.png)

**Figure 9.15**: Editing transaction types.