This chapter gets straight to the essence of multi-tier design under PHP by walking through the design and implementation of a complete application. Though the application is simple in some ways—its database consists of a single table, for example, and there is no elsewhere layer—it illustrates some important points. The idea is that you'll get a taste of these key concepts here, and explore them more fully in later chapters.

5.1 Examining the Problem

The requirement we aim to solve in this chapter is one of navigation. Our customer, an airline, wants us to provide an application that calculates the distance between various world cities. For example, a pilot should be able to open up our application—delivered via one or more Web pages, naturally—and specify an origin city and a destination city. The application should then return a value, in kilometers, that represents the shortest possible air route between the two cities. This value will be used for fuel calculations and other planning purposes.

The Earth, as you may have heard, is roughly spherical. The shortest distance between two points on the surface of a sphere is along a circle whose edge includes the two points, and whose center, furthermore, is the center of the Earth. Such a route is called a Great Circle route. The Great Circle route between two nearby points is nearly indistinguishable from a straight line, but a Great Circle path between two widely spaced points can seem odd to people accustomed to thinking in terms of flat surfaces. This is why the shortest path between New York and Hong Kong, for example, goes nearly over
the North Pole. Federal Express has a freight hub in Anchorage, Alaska, for a reason: That city is roughly equidistant from Tokyo, New York, and London.

In any case, the customer wants us to develop an application that takes latitude and longitude values for a given city from a database and uses them to calculate the Great Circle distance between the two places. The solution to the problem, as well as the interface that allows the user to specify the two cities of interest, should be one or more browser-renderable pages.

5.1.1 Sketching Out the Layers

Not surprisingly, the customer's requirement represents an excellent opportunity to use multi-tier software architecture.

- The database. The customer wants latitude and longitude information about various cities to be stored in a database. They don't specify a particular kind of database, but we know that the odds are good that at some point in the life of our application the airline will migrate from one database platform to another. At the very least, we want the ability to do our development and testing under one kind of database before connecting our application to the airline's database server. It seems like a good idea to treat our database and its server as a discrete unit—the database layer.

- The accessors. With a database layer that's distinct from everything else, we need a collection of software elements that act as interfaces between the database and the software that relies on its data. Structured query language (SQL) is the specific go-between, but we want to abstract the SQL statements a bit. This is the function of the objects in the accessor layer. The accessor layer should be easily reconfigured to connect to a new kind of database server, as well. For that reason, it will make use of the PEAR DB classes, which will be discussed further later in this chapter, and in Chapter 7 on database access.

- The business logic. The customer provided us with a statement of the business logic for this application. The application is to calculate Great Circle distances. How? Based on the latitudes and longitudes of the two endpoint cities. If we can find a mathematical formula that takes such input and provides the required output, and can encode that algorithm into the PHP language, we have a business logic layer.

- The presentation mechanism. The user interface shouldn't be too complicated, and it may even be possible to fit it all into one piece of software. The presentation layer has to first present the user with lists of possible origin and destination cities and allow him or her to select one of each. Upon submission of those values, the user interface should present the calculated Great Circle distance, and make it easy for the user to do another calculation with a different pair of cities.

It's all modular, and it'll be easier to build, maintain, and modify because of that characteristic. Multi-tier design seems like the right design choice for this application. Figure 5.1 shows the proposed architecture for our solution.
5.1.2 Communication Between the Layers

How, though, will the layers communicate with one another? Communication requires a protocol, either a custom-designed and -implemented one or a standard one that's adopted. What characteristics will our interlayer communication protocol need? Here's a list:

- Support for intermachine communications. It's possible that the various layers of this application will reside on different hardware platforms. It's safe to assume that they'll be connected by a TCP/IP network, and that standard protocols, such as domain name service (DNS), hypertext transport protocol (HTTP), and simple mail transport protocol (SMTP), will exist on the network.

- Support for the exchange of simple and complex data types. Considering the problem to be solved here, it seems likely that we'll need to move complex data structures, such as indexed and associative arrays, between layers. Our job will be much simpler if the communications protocol lets us move such structures without drama.
Ease of implementation. We want to spend our time as designers and, to a lesser extent, implementors of a software solution. The application we create should be notable as one that solves its assigned problem efficiently, not as one that uses a particular interlayer communications protocol.

Later chapters deal with the question of why, but the quick answer to the question of which interlayer communication protocol to use is: simple object access protocol (SOAP). SOAP satisfies all of our performance requirements, and it's easy to implement because (as is often the case in PHP) there's a freely available library that will do the heavy lifting for us. We just whack a reference to the library into our software and start using SOAP communication.

5.2 The Database Layer

The persistence layer (assuming that it's a database, anyhow) speaks SQL. It receives SQL queries in, and replies with SQL result sets (or messages that indicate success or failure). In the larger scheme of multi-tier architecture, the persistence layer receives incoming SQL queries from the PHP programs in the accessor layer. As a matter of principle, nothing but programs on the accessor layer should read from or write to the database. That's a very important idea in multi-tier application architecture.

5.2.1 Creating the Table

Our application has a very simple back end that comprises only one table, which has only four columns. There are no real issues of database design to be worked through in this case—no foreign keys, dependencies, or relationships to be established and verified.

The table, to be called cities, needs only to hold the name of each city to which the airline flies and the latitude and longitude of each. Because it's good form—something you should always include in your tables—there will be an automatically incremented id column, as well, just in case there is ever a need to establish relationships with this table.

Here's the SQL code that establishes the table we need in the MySQL database server:

```
cities.sql

DROP TABLE IF EXISTS cities;

CREATE TABLE 'cities' (  
   'id' int(11) NOT NULL auto_increment,  
   'cityName' varchar(30) NOT NULL default ",  
   'latitude' double NOT NULL default '-9999',  
   'longitude' double NOT NULL default '-9999',  
   PRIMARY KEY ('id')  
)  
TYPE=MyISAM;
```
Note that the default values given for the latitude and longitude columns are obviously invalid. If we’d used 0 instead, we’d have run the risk of confusion because 0,0 is a real location off the west coast of Africa.

To run that code on a typical MySQL server (assuming you had not created a database to contain the cities table), you’d use a sequence of commands beginning with these:

```
mysql
mysql>create database cities;
Query OK, 1 row affected (0.05 seconds)
mysql>quit
Bye.
```

Those lines logged you in, created a database called cities (the database and the table will have the same name), and logged you out. Then, at the operating system’s command line, you’d enter this:

```
mysql cities < cities.sql
```

That runs the SQL statements in cities.sql against the database called cities.

Continuing, there are some rules about the data that is to be stored in the cities table:

- The latitude column contains a decimal representation of degrees latitude, with 0 being the equator, positive numbers being north latitudes, and negative numbers being south latitudes.
- The longitude column contains a decimal representation of degrees longitude, with 0 being the prime meridian through Greenwich, positive numbers being west longitudes, and negative numbers being east longitudes.

Both latitude and longitude values are therefore between –180 and 180.

### 5.2.2 Populating the Table

To make our table useful, we need to populate it with some city names, latitudes, and longitudes. These SQL statements fill the cities table with data (note that the id column is populated automatically):

```
citiesPopulator.sql
```

```
INSERT INTO cities (cityName, latitude, longitude)
values('Sydney', -33.87, -151.22);

INSERT INTO cities (cityName, latitude, longitude)
values('Darwin', -12.47, -130.83);

INSERT INTO cities (cityName, latitude, longitude)
values('Hong Kong', 22.28, -114.15);
```
INSERT INTO cities (cityName, latitude, longitude) values('Mumbai', 19, -72.8);

INSERT INTO cities (cityName, latitude, longitude) values('Shanghai', 31.23, -121.47);

INSERT INTO cities (cityName, latitude, longitude) values('Seoul', 37.55, -126.97);

INSERT INTO cities (cityName, latitude, longitude) values('Copenhagen', 55.67, -12.58);

INSERT INTO cities (cityName, latitude, longitude) values('Paris', 42.87, -2.33);

INSERT INTO cities (cityName, latitude, longitude) values('Cairo', 30.05, -31.25);

INSERT INTO cities (cityName, latitude, longitude) values('Cape Town', -33.92, -18.22);

INSERT INTO cities (cityName, latitude, longitude) values('Washington', 38.9, 77.03);

INSERT INTO cities (cityName, latitude, longitude) values('San Francisco', 37.78, 122.42);

INSERT INTO cities (cityName, latitude, longitude) values('Santiago', -33.45, 70.67);

INSERT INTO cities (cityName, latitude, longitude) values('Recife', -8.05, 34.87);

To run that code, use the same syntax as before at the operating system's command line:

mysql cities < citiesPopulator.sql

You can see the effects of the populator script by entering the SQL query,

SELECT * from cities

which should yield results like this:

<table>
<thead>
<tr>
<th>id</th>
<th>cityName</th>
<th>latitude</th>
<th>longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sydney</td>
<td>-33.87</td>
<td>-151.22</td>
</tr>
<tr>
<td>2</td>
<td>Darwin</td>
<td>-12.47</td>
<td>-130.83</td>
</tr>
</tbody>
</table>
The accessor layer is the first layer at which we find software written in the PHP language. Essentially, the accessor layer exists to keep SQL statements out of the business logic layer. That's done for reasons of security and improved reliability.

Functions in the accessor layer typically are accessor and mutator functions (otherwise called "getter" and "setter" functions), which retrieve and modify values in the database, respectively. With such functions in place, software lower in the application—in the logic layer, specifically—can (indeed, should) refer to the getter and setter functions and not include a single line of SQL.

The purpose of the accessor layer in this application is to provide two sets of data, each under a different condition. First, when the application is generating the user interface, the accessor layer needs to provide a list, in other words, an array, of the cities whose details are included in the database. Second, the accessor layer needs to be able to take a city name as input and, in response, return an array containing that city's latitude and longitude values. The accessor layer is going to include two significant functions.

5.3.1 Isolating the Database Details

Because the accessor layer is the interface between the database and the software that makes use of it, much of the code in the accessor layer is concerned with connecting to the database server. Furthermore, because there's only one table supporting this simple application, all PHP programs in the accessor layer will connect to it.

This program will use the PEAR DB class for database connectivity, largely because PEAR DB makes it easy to change database server types (say, from MySQL to Oracle) if the need arises. PEAR DB requires certain information when it establishes a connection, including the hostname of the machine on which the database runs, the username and password to be used in establishing a connection, and the name of the table to be queried.

<table>
<thead>
<tr>
<th></th>
<th>Hong Kong</th>
<th>22.28</th>
<th>-114.15</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Mumbai</td>
<td>19</td>
<td>-72.8</td>
</tr>
<tr>
<td>5</td>
<td>Shanghai</td>
<td>31.23</td>
<td>-121.47</td>
</tr>
<tr>
<td>6</td>
<td>Seoul</td>
<td>37.55</td>
<td>-126.97</td>
</tr>
<tr>
<td>7</td>
<td>Copenhagen</td>
<td>55.67</td>
<td>-12.58</td>
</tr>
<tr>
<td>8</td>
<td>Paris</td>
<td>42.87</td>
<td>-2.33</td>
</tr>
<tr>
<td>9</td>
<td>Cairo</td>
<td>30.05</td>
<td>-31.25</td>
</tr>
<tr>
<td>10</td>
<td>Cape Town</td>
<td>-33.92</td>
<td>-18.22</td>
</tr>
<tr>
<td>11</td>
<td>Washington</td>
<td>38.9</td>
<td>77.03</td>
</tr>
<tr>
<td>12</td>
<td>San Francisco</td>
<td>37.78</td>
<td>122.42</td>
</tr>
<tr>
<td>13</td>
<td>Santiago</td>
<td>-33.45</td>
<td>70.67</td>
</tr>
<tr>
<td>14</td>
<td>Recife</td>
<td>-8.05</td>
<td>34.87</td>
</tr>
</tbody>
</table>
Furthermore, PEAR DB requires a specific string that identifies the type of database server being connected to the application. A list of various servers’ key strings appears in the PEAR DB documentation and in Chapter 7, but for now just know that the identifying string that corresponds to a MySQL server is mysql.

Each program in the accessor layer will need all of these details. Because it’s possible that they’ll change—that the server will move to a different machine, or that the username and password will change—it makes sense to isolate all of the details in a single file. The file can then be imported (using a require_once() statement, typically) into any piece of software that will access the database.

Here is a listing of the most important lines from dbDetails.php:

```php
$phptype = 'mysql'; // Type of database server (PEAR-DB standard name string).
$hostspec = 'db2'; // Hostname of database server (or IP address).
$dbdatabase = 'cities'; // Database name.
$username = 'access'; // Database username.
$password = 'php'; // Database password.
```

The configuration details are all there and commented, ready for modification.

### 5.3.2 Getting City Names from the Database

The first requirement of this application’s accessor layer is that it provide an array containing all the cities whose geographic details appear in the cities table. This is a straightforward matter of an SQL statement based on the SELECT command, but there’s a fair bit of PHP packaging that has to go on around it.

The application uses getCities.php to satisfy the city-listing requirement. That file, like all files discussed in this book, is available on the Internet site. It’s also listed fully here, but interspersed with explanatory text to clarify what’s going on.

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

The first order of business is to import some libraries. The first line imports Dietrich Ayala’s NuSOAP library and the second brings in the simple declarations of database details that we moved to a separate file for ease of maintenance.

```php
// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('getCities');
$s->service($HTTP_RAW_POST_DATA);
```

These lines refer to NuSOAP classes and establish a SOAP server to which other layers can connect. The SOAP server is set up to catch HTTP requests, and the getCities() function, to be declared momentarily, is made available for external access.

```php
function getCities() {
```
// Make globals from dBDetails.php available within function scope.

global $phptype;  // Type of database server.
global $hostspec; // Hostname of database server.
global $database; // Database name.
global $username; // Database username.
global $password; // Database password.

// Assemble Data Source Name (DSN) and connect, allowing for errors...

$dsn = "$phptype://$username:$password@$hostspec/$database";
$db = DB::connect($dsn);

if (DB::isError($db)) {
    die ($db->getMessage());
}

The first part of the getCities() function concerns itself with connecting to the database whose details are defined by the variables in dBDetails.php. The variables have to be made available inside the function (that's what the five lines beginning with global are for). Then, they can be used to assemble a datasource name (DSN), which is a uniform resource locator (URL)-like string that PEAR DB uses to establish a connection to a database. An attempt to connect is made; the attempt results in an object that has to be examined to see if it's an error, in which case everything is off.

// Assemble and send SQL statement, allowing for errors...

$sql = "SELECT cityName FROM cities ORDER BY cityName";
$result = $db->query($sql);

if (DB::isError($result))
{
    $errorMessage = $result->getMessage();
    die ($errorMessage);
}

With the connection to the database established, the function proceeds to set a variable equal to an SQL query string (the decision to put the ORDER BY instruction here, rather than in the presentation layer, certainly is debatable). The query then gets sent to the database, which results in an object. That object has to be checked to see if it's an error. If it's not an error, it's an object representing the results of the query.

    // Extract rows from query results, fitting pieces of data into
// $returnArray (an associative array) for returning.

while ($row = $result->fetchRow())
{
  $returnArray[$i]=$row[0];
  ++$i;
}

This loop exists to use MySQL's fetchRow() function against every row in the result object, thus extracting it. Because we know the result set has only one column (the SQL statement requested only cityName), we can take the first element of every row array ($row[0]—the only element) and put it into another array, $returnArray.

// Disconnect from database and return $returnArray...

$db->disconnect();

return $returnArray;

}

Having extracted all returned rows, the function terminates the database connection and returns $returnArray. Because this function is exposed as a SOAP service, $returnArray could be sent out across the network via the SOAP protocol. It's not a problem; SOAP handles the transmission of arrays without any hassle.

5.3.3 Getting Latitudes and Longitudes from the Database

The other function of the Great Circle application's accessor layer is to get latitude and longitude of a specified city from the database. This piece of accessor software has to take a city name as a parameter and use it in extracting coordinate data from the cities table.

This functionality appears in getLatLong.php, which is listed and commented upon here:

require_once('nusoap-0.6/nusoap.php');
require_once('dbDetails.php');

// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('getLatLong');
$s->service($HTTP_RAW_POST_DATA);
The program opens much like getCities.php, but with the getLatLong() function being exposed this time.

```php
function getLatLong($city) {

    // Make globals from dbDetails.php available within function scope.

    global $phptype; // Type of database server.
    global $hostspec; // Hostname of database server.
    global $database; // Database name.
    global $username; // Database username.
    global $password; // Database password.

    // Assemble Data Source Name (DSN) and connect, allowing for errors...

    $dsn = "$phptype://$username:$password@$hostspec/$database";
    $db = DB::connect($dsn);

    if (DB::isError($db)) {
        die ($db->getMessage());
    }

    // Assemble and send SQL statement, allowing for errors...

    $sql = "SELECT latitude,longitude FROM cities WHERE cityName='$city';";
    $result = $db->query($sql);

    if (DB::isError($result)) {
        $errorMessage = $result->getMessage();
        die ($errorMessage);
    }

    // Extract rows from query results, fitting pieces of data into
    // $returnArray (an associative array) for returning.

    while ($row = $result->fetchRow()) {
```

Again, the sole function uses PEAR DB to hook up to a database, assembling a DSN out of the pieces defined in the dbDetails.php file.

```php
    // Assemble and send SQL statement, allowing for errors...

    $sql = "SELECT latitude,longitude FROM cities WHERE cityName='$city';";

    $result = $db->query($sql);

    if (DB::isError($result)) {
        $errorMessage = $result->getMessage();
        die ($errorMessage);
    }
```

The SQL query includes $city (note the quotation marks around the variable reference), which came into the function as a parameter.

```php
    // Extract rows from query results, fitting pieces of data into
    // $returnArray (an associative array) for returning.

    while ($row = $result->fetchRow()) {
```
$latitude = $row[0];
$longitude = $row[1];

$returnArray['city'] = $city;
$returnArray['latitude'] = $latitude;
$returnArray['longitude'] = $longitude;
}

This function differs from getCities() in that it uses the query results, which comprise three columns, to create an associative array. The code inside the while loop relies on the fact that the sequence of the columns in the result set is known—a situation that results from the explicit listing of column names in the SELECT statement.

    // Disconnect from database and return $returnArray...

$db->disconnect();

return $returnArray;

}  

When the work of querying the database and transferring the results to an associative array is done, the function shuts down the database connection and returns the associative array containing coordinate information.

5.4 The Business Logic Layer

The business logic layer is where you tackle the problems your program was created to solve. In the logic layer, classes decide what information they need in order to solve their assigned problems, request that information from the accessor layer, manipulate that information as required, and return the ultimate results to the presentation layer for formatting.

The business logic layer in this application does the spherical geometry—the Great Circle calculation itself. The calculation program, calcGreatCircle.php, uses an algorithm that assumes the Earth is perfectly spherical, which it isn't. The calculation is not the point here, so if you're thinking about using this application as a serious navigation tool, think again.

In terms of software requirements, calcGreatCircle.php is interesting because it has to be both a SOAP server and a SOAP client. As a SOAP server, it takes city name pairs from the presentation layer. As a SOAP client, it forwards those city names to the accessor layer, getLatLong.php, specifically, in order to retrieve the latitude and longitude of each.

Here are the contents of calcGreatCircle.php, the sole element of the Great Circle application's business logic layer:
require_once('nusoap-0.6/nusoap.php');

// Establish NuSOAP soap_server object
// and register function as Web Service...

$s = new soap_server;
$s->register('calculateGreatCircle');
$s->service($HTTP_RAW_POST_DATA);

Only the NuSOAP library is imported; there’s no need for the database stuff here. The calculateGreatCircle() function is exposed as a SOAP service.

function toRad($degrees) {
    // Converts $degrees to equivalent value in radians.

    $radians = $degrees * (pi()/180);
    return $radians;
}

The function toRad() is a utility function that calculateGreatCircle() makes use of. It converts a value expressed in degrees into an equivalent value expressed in radians. This function is not exposed as a SOAP service; it’s accessed only by calculateGreatCircle().

function calculateGreatCircle($city1, $city2) {
    // Calculates Great Circle distance (in km) between $city1 and $city2
    // Establish $parameters array and call Web Service to get latitude and longitude for $city1...

    $parameters = array('city'=> $city1);
    $soapclient = new soapclient('http://db2/greatCircle/getLatLong.php');
    $returnedArray = $soapclient->call('getLatLong',$parameters);

    // Populate simple variables for clarity...

    $lat1 = $returnedArray[latitude];
    $long1 = $returnedArray[longitude];

    // Establish $parameters array and call Web Service to get latitude and longitude for $city2...

    $parameters = array('city'=> $city2);
$soapclient = new soapclient('http://db2/greatCircle/getLatLong.php');
$returnedArray = $soapclient->call('getLatLong',$parameters);

// Populate simple variables for clarity...
$lat2 = $returnedArray[latitude];
$long2 = $returnedArray[longitude];

// Convert degrees to radians
$lat1 = toRad($lat1);
$long1 = toRad($long1);
$lat2 = toRad($lat2);
$long2 = toRad($long2);

// Calculate distance...
$theta = $long2 - $long1;
$distance = acos((sin($lat1) * sin($lat2)) + (cos($lat1) * cos($lat2) *
                          cos($theta)));

if ($distance < 0) {
    $distance = $distance + pi();
}

// Multiply by constant to get kilometers...
$distance = $distance * 6371.2;
return $distance;
}

The rest of calcGreatCircle.php has to do with the Great Circle calculation itself, which isn't remarkable except for its extensive use of PHP's trigonometry functions.

### 5.5 The Presentation Layer

The presentation layer exists for the purpose of providing a user interface, whether the user is a machine or a human being. If the user is a human being, the user interface will likely take the form of an hypertext markup language (HTML) document. They will likely include text boxes, buttons, and selection lists—all the usual elements we see when we use our computers. The details of user interface design (how to arrange your program's interface elements, how your commands should behave, what sort of feedback
your users should get, and so on) make up an elaborate field of programming specialty. They're largely beyond the scope of this book, so we'll deal only with the characteristics of the user interface that have to do with communicating with the rest of the application.

If the user is a machine, another software application of some kind, our program should probably generate an extensible markup language (XML) document as output. The beauty of the multi-tier architecture comes through when you consider that it would be just as easy to provide XML, rather than HTML, documents at the presentation layer. With XML results being generated, the application becomes, broadly speaking, more of a Web service (to be used by other machines rather than by people) than a business application. That means someone else could use your whole application as a module in his or her project.

The presentation layer of the Great Circle application comprises a single page, which perhaps not ideally, contains a combination of PHP and HTML code. Its "life cycle" has two parts. First, it displays lists of candidate origin and destination cities, each with a corresponding radio button, and a master Submit button at the bottom of the page. That's shown in Figure 5.2.

When the user chooses cities and clicks the Submit button, the page changes to include the calculated distance between the two previously selected cities, as depicted in Figure 5.3.

Here's a commentary on the presentation layer program, greatCircle.php (the name, which makes no mention of the presentation layer, was chosen because the user may have to type this filename as part of a URL):

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

// Extract from $_POST array to allow for register_globals being off

$city1 = $_POST['origin'];
$city2 = $_POST['destination'];

Variables $_POST['origin'] and $_POST[destination] correspond to the name attributes of the two sets of radio buttons in the HTML that appear later in greatCircle.php. When a form is submitted to this program (it's submitted to itself, a process that's explained later in this section), the value attribute of the selected radio button from each group is the value of $_POST['origin'] and $_POST[destination]. Note that it's no longer good practice to refer to $origin and $destination directly, as was reasonable with the older versions of PHP that shipped with the register_globals option (in php.ini) on bydefault. Modern versions of PHP have register_globals off, so we must approach form contents via the superglobal $_POST (or $_GET) array. See Chapter 4 for more information on HTTP POST and HTTP GET operations.

// Establish $parameters array and call Web Service to get distance

$parameters = array('city1'=> $city1, 'city2'=> $city2);
$soapclient = new soapclient('http://db2/greatCircle/calcGreatCircle.php');
```

The beautify of the multi-tier architectur comes through when you consider that it would be just as easy to provide XML, rather than HTML, documents at the presentation layer. With XML results being generated, the application becomes, broadly speaking, more of a Web service (to be used by other machines rather than by people) than a business application. That means someone else could use your whole application as a module in his or her project.

The presentation layer of the Great Circle application comprises a single page, which perhaps not ideally, contains a combination of PHP and HTML code. Its "life cycle" has two parts. First, it displays lists of candidate origin and destination cities, each with a corresponding radio button, and a master Submit button at the bottom of the page. That's shown in Figure 5.2.

When the user chooses cities and clicks the Submit button, the page changes to include the calculated distance between the two previously selected cities, as depicted in Figure 5.3.

Here's a commentary on the presentation layer program, greatCircle.php (the name, which makes no mention of the presentation layer, was chosen because the user may have to type this filename as part of a URL):

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

// Extract from $_POST array to allow for register_globals being off

$city1 = $_POST['origin'];
$city2 = $_POST['destination'];

Variables $_POST['origin'] and $_POST[destination] correspond to the name attributes of the two sets of radio buttons in the HTML that appear later in greatCircle.php. When a form is submitted to this program (it's submitted to itself, a process that's explained later in this section), the value attribute of the selected radio button from each group is the value of $_POST['origin'] and $_POST[destination]. Note that it's no longer good practice to refer to $origin and $destination directly, as was reasonable with the older versions of PHP that shipped with the register_globals option (in php.ini) on bydefault. Modern versions of PHP have register_globals off, so we must approach form contents via the superglobal $_POST (or $_GET) array. See Chapter 4 for more information on HTTP POST and HTTP GET operations.

// Establish $parameters array and call Web Service to get distance

$parameters = array('city1'=> $city1, 'city2'=> $city2);
$soapclient = new soapclient('http://db2/greatCircle/calcGreatCircle.php');
```
Great Circle Calculator

Choose the origin.

- Cairo
- Cape Town
- Copenhagen
- Darwin
- Hong Kong
- Mumbai
- Paris
- Recife
- San Francisco
- Santiago
- Seoul
- Shanghai
- Sydney
- Washington

Choose the destination.

- Cairo
- Cape Town
- Copenhagen
- Darwin
- Hong Kong
- Mumbai
- Paris
- Recife
- San Francisco
- Santiago
- Seoul
- Shanghai
- Sydney
- Washington

Figure 5.2: The Great Circle presentation layer, before submission of a city pair.

Figure 5.3: The Great Circle presentation layer, after submission of a city pair and ready for another challenge.

$distance = $soapclient->call('calculateGreatCircle', $parameters);
$distance = round($distance);

The function makes a call to the business logic layer, sending the two city names as parameters for calcGreatCircle to process. The round() function strips the fractional portion from the result, because the algorithm used isn’t accurate enough to merit such precision.

// Establish $parameters array and call Web Service to get list of cities...

$parameters = array();
$soapclient = new soapclient('http://db2/greatCircle/getCities.php');
$cities = $soapclient->call('getCities',$parameters);

Another SOAP call, this time directly to the accessor layer, secures an array of city names. This array is used twice in generating the HTML interface.

```html
<html>
<head>
<title>Great Circle Calculator</title>
</head>
<body>

<H2>Great Circle Calculator</H2>

```php
if ($city1 != "" AND $city2 != ":") {
    echo "The great circle distance between $city1 and $city2 is $distance kilometers."
;
}
```php

This conditional statement ensures that the calculated Great Circle distance is displayed only if such a calculated distance exists. The result isn't shown the first time the page is loaded, in other words.

```html
<form action="<?php $PHP_SELF; ?>" method="post">

<H3>Choose the origin.</H3>

```php
foreach ($cities as $city) {
    echo "<input type='RADIO' name='origin' value='$city'>$city<br>
"
;
}
```php

A simple PHP loop goes through the $cities array and generates the HTML radio buttons representing origin cities.

```html
<H3>Choose the destination.</H3>

```php
foreach ($cities as $city) {
    echo "<input type='RADIO' name='destination' value='$city'>$city <br>
"
;
}
```php

```html
</form>
```
A nearly identical loop, also operating on the `$cities` array, generates the HTML radio buttons representing destination cities.

```html
<P>
<input type="SUBMIT" name="submitButton" value="Calculate Great Circle">
</form>
</body>
</html>
```

The remainder of the program is simple HTML code.

### 5.6 Questions and Exercises

Here are some questions and exercises meant to further your thinking about the material contained in this chapter.

1. This chapter didn't implement the elsewhere layer, which was introduced at the beginning of the chapter. How might an elsewhere layer be put to advantageous use in the Great Circle application?

2. If you were to expand the scope of the Great Circle application to enable the user to specify a city name manually, rather than choosing one from a list, what problems might this introduce? How could you deal with them?

3. If you didn't want to use SOAP for interlayer communication, what other strategies could you use?

4. See if you can redesign the presentation layer of the Great Circle application so that the PHP code is completely separated from the HTML code.