When you are combining functions into a code library, you should be careful to maintain a balance between grouping related functions and including functions that are not often used. When you include a code library in a page, all of the functions in that library are parsed, whether you use them all or not. PHP's parser is quick, but not parsing a function is even faster. At the same time, you don't want to split your functions over too many libraries, so that you have to include lots of files in each page, because file access is slow.

Templating Systems

A *templating system* provides a way of separating the code in a web page from the layout of that page. In larger projects, templates can be used to allow designers to deal exclusively with designing web pages and programmers to deal (more or less) exclusively with programming. The basic idea of a templating system is that the web page itself contains special markers that are replaced with dynamic content. A web designer can create the HTML for a page and simply worry about the layout, using the appropriate markers for different kinds of dynamic content that are needed. The programmer, on the other hand, is responsible for creating the code that generates the dynamic content for the markers.

To make this more concrete, let's look at a simple example. Consider the following web page, which asks the user to supply a name and, if a name is provided, thanks the user:

```
<html>
  <head>
    <title>User Information</title>
  </head>

  <body>
    <?php if (!empty($_GET['name'])) {
      // do something with the supplied values
    ?>

    <p><font face="helvetica,arial">Thank you for filling out the form,
       <?php echo $_GET['name'] ?>.</font></p>
<?php }
else { ?>
      <p><font face="helvetica,arial">Please enter the
      following information:</font></p>

      <form action="<?php echo $_SERVER['PHP_SELF'] ?>">
        <table>
          <tr>
            <td>Name:</td>
            <td><input type="text" name="name" /></td>
          </tr>
        </table>
      </form>
```

```
        <?php } ?>
      </body>
    </html>
```

The placement of the different PHP elements within various layout tags, such as the `font` and `table` elements, are better left to a designer, especially as the page gets more complex. Using a templating system, we can split this page into separate files, some containing PHP code and some containing the layout. The HTML pages will then contain special markers where dynamic content should be placed. Example 13-1 shows the new HTML template page for our simple form, which is stored in the file *user.template*. It uses the {DESTINATION} marker to indicate the script that should process the form.

Example 13-1. HTML template for user input form

```
<html>
  <head>
    <title>User Information</title>
  </head>

  <body>
    <p><font face="helvetica,arial">Please enter the following
    information:</font></p>

    <form action="{DESTINATION}">
      <table>
        <tr>
          <td>Name:</td>
          <td><input type="text" name="name" /></td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

Example 13-2 shows the template for the thank you page, called *thankyou.template*, that is displayed after the user has filled out the form. This page uses the {NAME} marker to include the value of the user's name.

Example 13-2. HTML template for thank you page

```
<html>
  <head>
    <title>Thank You</title>
  </head>

  <body>
    <p><font face="helvetica,arial">Thank you for filling out the form,
      {NAME}.</font></p>
  </body>
</html>
```

Now we need a script that can process these template pages, filling in the appropriate information for the various markers. Example 13-3 shows the PHP script that uses these templates (one for before the user has given us information and one for after). The PHP code uses the `FillTemplate( )` function to join our values and the template files. This file is called *form_template.php*.

Example 13-3. Template script

```
$bindings['DESTINATION'] = $PHP_SELF;

$name = $_GET['name'];

if (!empty($name)) {
  // do something with the supplied values
  $template = "thankyou.template";
  $bindings['NAME'] = $name;
}
else {
  $template = "user.template";
}

echo FillTemplate($template, $bindings);
```

Example 13-4 shows the `FillTemplate( )` function used by the script in Example 13-3. The function takes a template filename (to be located in the document root in a directory called *templates*), an array of values, and an optional instruction denoting what to do if a marker is found for which no value is given. The possible values are: `"delete"`, which deletes the marker; `"comment"`, which replaces the marker with a comment noting that the value is missing; or anything else, which just leaves the marker alone. This file is called *func_template.php*.

Example 13-4. The FillTemplate() function

```
function FillTemplate($inName, $inValues = array( ),
                      $inUnhandled = "delete") {
  $theTemplateFile = $_SERVER['DOCUMENT_ROOT'] . '/templates/' . $inName;
  if ($theFile = fopen($theTemplateFile, 'r')) {
    $theTemplate = fread($theFile, filesize($theTemplateFile));
    fclose($theFile);
  }

  $theKeys = array_keys($inValues);
  foreach ($theKeys as $theKey) {
    // look for and replace the key everywhere it occurs in the template
    $theTemplate = str_replace("\{$theKey}", $inValues[$theKey],
                               $theTemplate);
  }

  if ('delete' == $inUnhandled ) {
    // remove remaining keys
    $theTemplate = eregi_replace('{[^ }]*}', '', $theTemplate);
```

Example 13-4. The FillTemplate( ) function (continued)

```
  } elseif ('comment' == $inUnhandled ) {
    // comment remaining keys
    $theTemplate = eregi_replace('{([^ }]*)}', '<!-- \\1 undefined -->',
                                 $theTemplate);
  }

  return $theTemplate;
}
```

Clearly, this example of a templating system is somewhat contrived. But if you think of a large PHP application that displays hundreds of news articles, you can imagine how a templating system that used markers such as {HEADLINE}, {BYLINE}, and {ARTICLE} might be useful, as it would allow designers to create the layout for article pages without needing to worry about the actual content.

While templates may reduce the amount of PHP code that designers have to see, there is a performance trade-off, as every request incurs the cost of building a page from the template. Performing pattern matches on every outgoing page can really slow down a popular site. Andrei Zmievski's *Smarty* is an efficient templating system that neatly side-steps this performance problem. *Smarty* turns the template into straight PHP code and caches it. Instead of doing the template replacement on every request, it does it only when the template file is changed. See *http://smarty.php.net/* for more information.

Handling Output

PHP is all about displaying output in the web browser. As such, there are a few different techniques that you can use to handle output more efficiently or conveniently.

Output Buffering

By default, PHP sends the results of `echo` and similar commands to the browser after each command is executed. Alternately, you can use PHP's output buffering functions to gather the information that would normally be sent to the browser into a buffer and send it later (or kill it entirely). This allows you to specify the content length of your output after it is generated, capture the output of a function, or discard the output of a built-in function.

You turn on output buffering with the `ob_start( )` function:

```
ob_start([callback]);
```

The optional *callback* parameter is the name of a function that post-processes the output. If specified, this function is passed the collected output when the buffer is flushed, and it should return a string of output to send to the browser. You can use this, for instance, to turn all occurrences of *http://www.yoursite.com/* to *http://www.mysite.com/*.