



**DRAFT: Mambo Developers' Manual**

# Table of Contents

<b><u>DRAFT: Mambo Developers' Manual</u></b> .....	<b>1</b>
<u>Andrew Eddie, Lead Mambo Developer</u> .....	1
<u>Alex Kempkens, Mambo Developer</u> .....	1
<b><u>Preface</u></b> .....	<b>2</b>
<b><u>Chapter 1. Basic Templating and Site Design</u></b> .....	<b>3</b>
<u>Overview</u> .....	3
<u>The Layout File</u> .....	3
<u>CSS Stylesheets</u> .....	5
<u>The XML Setup File</u> .....	5
<u>The Thumbnail</u> .....	5
<b><u>Chapter 2. Advanced Templating</u></b> .....	<b>6</b>
<u>Overview</u> .....	6
<u>Hiding Modules</u> .....	6
<u>Using Class Suffixes</u> .....	6
<u>File and Function Reference</u> .....	6
<u>mosLoadComponents</u> .....	6
<u>mosCountModules</u> .....	6
<u>mosLoadModules</u> .....	6
<u>Administrator Templates</u> .....	7
<u>Module Support</u> .....	8
<u>mod_fullmenu</u> .....	8
<u>mod_components</u> .....	8
<u>mod_latest</u> .....	9
<u>mod_mosmsg</u> .....	9
<u>mod_online</u> .....	9
<u>mod_pathway</u> .....	9
<u>mod_popular</u> .....	9
<u>mod_stats</u> .....	9
<u>mod_toolbar</u> .....	9
<u>mod_unread</u> .....	9
<u>The Control Panel</u> .....	9
<b><u>Chapter 3. Modules</u></b> .....	<b>11</b>
<u>Overview</u> .....	11
<u>Writing a Module</u> .....	11
<u>Where do we start?</u> .....	11
<u>The Database Connector</u> .....	12
<u>Finishing Up</u> .....	12
<b><u>Chapter 4. Mambots</u></b> .....	<b>14</b>
<u>Overview</u> .....	14
<u>Writing a Content Mambot – The Old Way</u> .....	14
<u>Writing a Mambot – The New Way</u> .....	15
<u>An onSearch Mambot</u> .....	15
<u>An onPrepareContent Mambot</u> .....	16

# Table of Contents

<b><u>Chapter 4. Mambots</u></b>	
<u>An Editor Mambot</u> .....	18
<u>Extending Mambots</u> .....	18
<b><u>Chapter 5. Components</u></b> .....	<b>21</b>
<u>Overview</u> .....	21
<u>Scoping and Planning</u> .....	21
<u>Developing the Database Schema</u> .....	21
<u>Developing the Administrator Component</u> .....	22
<u>Adding Menu Items</u> .....	22
<u>Creating the Administrator Interface</u> .....	23
<u>The Component Toolbar</u> .....	24
<u>The Database Class Handlers</u> .....	26
<u>The Component Presentation Layer</u> .....	28
<u>The Component Event Handler</u> .....	30
<b><u>Chapter 6. Language Support</u></b> .....	<b>33</b>
<u>Overview</u> .....	33
<u>Installation and global configuration of languages</u> .....	33
<u>Creation of language files</u> .....	33
<u>Files within a language package</u> .....	33
<u>Manipulating the static text</u> .....	34
<u>Language configuration XML</u> .....	35
<b><u>Chapter 7. Packaging Custom Work</u></b> .....	<b>36</b>
<u>Overview</u> .....	36
<u>The XML Setup File</u> .....	36
<u>Parameters</u> .....	36
<u>XML Do's and Don'ts</u> .....	37
<u>A Mambot Setup File</u> .....	38
<u>A Module Setup File</u> .....	38
<u>A Component Setup File</u> .....	39
<u>A Template Setup File</u> .....	41
<b><u>Chapter 8. Access Control</u></b> .....	<b>42</b>
<u>Overview</u> .....	42
<u>phpGACL</u> .....	42
<u>The ACO</u> .....	42
<u>The ARO</u> .....	42
<u>The AXO</u> .....	43
<u>Inserting a New Group</u> .....	43
<b><u>Chapter 9. API Reference</u></b> .....	<b>44</b>
<b><u>Appendix A. Updates for 4.5.1</u></b> .....	<b>45</b>
<u>Mambots</u> .....	45
<u>Installer Parameters</u> .....	45
<u>Site Templates</u> .....	46

# Table of Contents

<b><u>Appendix A. Updates for 4.5.1</u></b>	
<u>Module containers</u>	46
<u>Pathway Arrows</u>	46
<u>CSS</u>	46
<u>Media installer tag</u>	46
<u>Modules</u>	46
<u>API Changes</u>	46
<u>Page Navigation</u>	46
<u>mosHTML</u>	47
<u>Tabs</u>	47
<u>Administrator</u>	47
<u>Templates and Modules</u>	47
<u>Banners</u>	47
<u>Help Component</u>	48
<u>Backup to mosDBTable</u>	48
<u>WYSIWYG Editors</u>	48
<u>Miscellaneous</u>	48
<u>Future Versions</u>	48
<u>Problems with XML Installer Files</u>	48
<b><u>Appendix B. Using Parameters</u></b>	<b>49</b>
<u>XML Definition</u>	49
<u>text</u>	49
<u>list</u>	49
<u>radio</u>	50
<u>mos_section</u>	50
<u>mos_category</u>	50
<u>mos_menu</u>	50
<u>mos_imagelist</u>	50
<u>textarea</u>	50
<u>The mosParameters Class</u>	51
<u>Extending Parameters</u>	51
<u>Using Parameters on the Site</u>	52
<u>Quick Fix for Old Modules</u>	53
<b><u>Appendix C. mosHTML Reference</u></b>	<b>54</b>
<u>mosHTML Helper Class</u>	54
<u>mosHTML::makeOption</u>	54
<u>mosHTML::selectList</u>	54
<u>mosHTML::integerSelectList</u>	55
<u>mosHTML::monthSelectList</u>	55
<u>mosHTML::treeSelectList</u>	56
<u>mosHTML::yesnoSelectList</u>	56
<u>mosHTML::radioList</u>	56
<u>mosHTML::yesnoRadioList</u>	56

## Table of Contents

<b><u>Appendix D. Code and Commenting Styles</u></b> .....	<b>57</b>
<u>File Header Comment</u> .....	57
<u>Documenting Classes</u> .....	57
<u>Documenting Functions</u> .....	58
<u>Miscellaneous Documentation</u> .....	58
<u>File Formats</u> .....	59
<u>Code Styles</u> .....	59
<b><u>Appendix E. Porting MiniXML to the DOMIT Library</u></b> .....	<b>60</b>

# DRAFT: Mambo Developers' Manual

**Andrew Eddie, Lead Mambo Developer**

**Alex Kempkens, Mambo Developer**

Copyright © 2004 Mambo Project. All rights reserved, The information in this publication is furnished for informational use only, and should not be construed as a commitment by the Mambo Project. The Mambo Project reserves the right to update or modify the contents. The Mambo Project assumes no responsibility or liability for any errors or inaccuracies that may appear in this publication.

## Revision History

Revision 3	26–Aug–2004	Revised by: aje
RC1 draft		
Revision 2	11–Aug–2004	Revised by: aje
Beta 4 draft		
Revision 1	10–Aug–2004	Revised by: aje
Initial draft		
Revision \$Revision: 1.2 \$	\$Date: 2004/08/24 04:25:39 \$	
CVS		

## List of Tables

5–1. [mos components Table Structure](#)

6–1. [File within the language package](#)

# Preface

Right up to the release of the 4.x versions of Mambo, writing components, modules and templates was a fairly convoluted affair. Files in scattered directories made it difficult to actually package an individual element; code was not organised into reusable API calls; it was generally hard work. The number of third-party addons around the time of the release of 4.0.x was testimony to this. There were maybe a half dozen templates and not many more components and modules.

The development of 4.5 sought to change all that. Code was modularised, files were reorganised and an installer made like just that bit easier. Today (over half way through 2004) we have literally hundreds of templates, and dozens of components, modules and mambots. Amazingly, most of the developers' of these addons have picked it up by trial and error and following the example of the main code base. To date, there has not been a great deal of quality developer documentation. The community spirit has had to come to the aid of many a fledgling developer crying "Help! How do I do this".

With the release of 4.5.1, we have realised that quality documentation, both for the user and for the developer, is of paramount importance. We have sought to lift both these areas with this release of Mambo.

This manual, while not yet complete, seeks to give you an insight into the workings of Mambo.

If you are new to Mambo, the chapters are set out roughly in order of easiest to hardest. We start with the Mambo template system which is extremely easy to learn. Modules then fit in nicely with your template work. Mambots are little multifunctional "things" that are life a Swiss Army Knife to Mambo. Then we look at building a Component.

If you are an old hand then we suggest you start in the "What's New" appendix to bring yourself up to speed with changes to the current version.

If there are any areas which you think are deficient then let us know. Suggestions and contributions are most welcome.

This is a draft manual. Download the latest copy from [http://mamboforge.net/docman/?group\\_id=5](http://mamboforge.net/docman/?group_id=5). Raw DocBook files may be downloaded from the CVS at [http://mamboforge.net/cgi-bin/cvsweb.cgi/mambo/4.5\\_manuals/english/](http://mamboforge.net/cgi-bin/cvsweb.cgi/mambo/4.5_manuals/english/).

If you who would like to submit corrections in spelling, grammar, syntax and/or content please use the bug tracker at MosForge at the following location: [http://mamboforge.net/tracker/?group\\_id=17](http://mamboforge.net/tracker/?group_id=17) You may also like to post comments on Mambo Forum at the following location: [http://mamboforge.net/forum/forum.php?thread\\_id=2474&forum\\_id=30](http://mamboforge.net/forum/forum.php?thread_id=2474&forum_id=30)

# Chapter 1. Basic Templating and Site Design

## Overview

The Mambo Template system is amongst the easiest to learn in the Content Management System family.

Templates are located in the `/templates` directory. The following shows a typical directory structure for a template:

```
/templates
/basic_template
/css
  template_css.css
/images
index.php
template_thumbnail.png
templateDetails.xml
```

This is the minimum set of files you need to make a template. The filenames must be adhered to as each one is expected by the core script. Note that while there are no images shown in the `/images` directory, this is typically where you would place any supporting images for your template, like backgrounds, banners, etc. Let's have a look at each of these files.

*index.php*: This is the template layout file.

*template\_css.css*: The CSS stylesheet for the template.

*templateDetails.xml*: This is a metadata file in XML format.

*template\_thumbnail.png*: A reduced screenshot of the template, usually around 140 pixels wide and 90 pixels high.

## The Layout File

While the template layout file is a PHP file, it is written mostly in HTML with only a few snippets of PHP. You do not have to be a master of PHP to write a template file. All you need to be able to do is learn where to place the key "hooks" into the Mambo templating engine.

A template layout is made up of mostly HTML. Within the HTML framework you place "windows" that look into the database behind your website. There are typically several small windows called Modules and usually one larger opening (like a frontdoor) for a Component.

You are encouraged to write templates in XHTML. While there is debate over whether XHTML *is* the way of the future, it is a well formed XML standard, whereas HTML is a loose standard. Future versions of Mambo will rely more and more on XML so it is wise to adopt this model now.

The `index.php` file for a typical 3-column layout would look like the following in a skeletal form:

```
1: <?php echo "<?xml version=\"1.0\"?>";
2: /** ensure this file is being included by a parent file */
3: defined( '_VALID_MOS' ) or die( 'Direct Access to this location is not allowed.' );
4: ?>
5: <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
6: <html xmlns="http://www.w3.org/1999/xhtml">
```

```

7: <head>
8: <title><?php echo $mosConfig_sitename; ?></title>
9: <meta http-equiv="Content-Type" content="text/html; <?php echo _ISO; ?>" />
10: <?php
11: if ($my->id) {
12:     include ("editor/editor.php");
13:     initEditor();
14: }
15: ?>
16: <?php include ("includes/metadata.php"); ?>
17: <link href="<?php echo $mosConfig_live_site;?>/templates/basic_template/css/template_css.css"
    rel="stylesheet" type="text/css" />
18: </head>
19: <body>
20: <table cellspacing="0" cellpadding="5" border="0">
21: <tr>
22: <td colspan="3">
23:     <?php echo $mosConfig_sitename; ?>
24: </td>
25: </tr>
26: <tr>
27: <td colspan="3">
28:     <?php mosLoadModules ( 'top', true ); ?>
29: </td>
30: </tr>
31: <tr>
32: <td width="20%" valign="top">
33:     <?php mosLoadModules ( 'left' ); ?>
34: </td>
35: <td width="60%" valign="top">
36:     <?php include_once("mainbody.php"); ?>
37: </td>
38: <td width="20%" valign="top">
39:     <?php mosLoadModules ( 'bottom' ); ?>
40: </td>
41: </tr>
42: </table>
43: </body>
44: </html>

```

Let's have a look at the main features. We are assuming you already know a bit about HTML pages so things like head tags, body tags, tables, etc will be skipped over.

Line 1: Defines the file as a valid XML file.

Line 3: Prevents direct access to this file. It is essential that you include this line in your template.

Lines 5–6: Set up the xHTML standard for the page.

Line 8: Prints out the Site Name configuration variable with the opening and closing title tags.

Line 9: `_ISO` is a special constant defining the character set to use. It is defined in your language file.

Line 10–15: `$my->id` is a script variable that is non-zero if a user is logged in to your site. If a user is logged in then the nominated WYSIWYG editor is pre-loaded. You may, if you wish, always pre-load the editor, but generally an anonymous visitor will not have the need to add content. This saves a little script overhead for normal browsing of your site.

Line 16: Inserts several metadata blocks.

Lines 17: Loads the CSS stylesheet. `$mosConfig_live_site` is a configuration variable that holds the absolute URL of your site.

Line 23: This prints the Site Name in a table cell (spanning the three columns).

Line 28: This loads any modules that are published in the "top" position. The "true" argument indicates that the modules are to be aligned horizontally.

Line 33: This loads any modules that are published in the "left" position. These modules will be displayed in a single column.

Line 36: This loads the component into your template. The component is set by the URL, for example, `index.php?option=com_content` will display the Content Component in this area.

Line 39: This loads any modules that are published in the "right" position. These modules will be displayed in a single column.

## CSS Stylesheets

TODO

## The XML Setup File

TODO

## The Thumbnail

When you have finished your template, publish it with the Template Manager in the Administrator. Preview the site and take a screenshot. Import the screen shot into you favourite graphic editing package and crop it down to contents of the browser's view port. Reduce the image down to around 140 pixels wide by 90 pixels high and save it in a PNG format to your template directory (that is, `/templates/basic_template`).

# Chapter 2. Advanced Templating

## Overview

This chapter includes some more advanced features such as hiding template columns and designing templates for the Administrator.

## Hiding Modules

Sometimes it is desirable to hide certain module areas if there are no modules assigned to that region. You can hide these areas by using the `mosCountModules` function.

```
<?php if (mosCountModules( "right" )) { ?>
  <td>
    <?php loadModules( "right" ); ?>
  </td>
<?php } ?>
```

If the `mosCountModules` function returns a value greater than 1, the table cell will be displayed. If there are no module defined for the "right" position for this particular page, then the cell will not be displayed. This is a good technique for increasing the horizontal screen width on certain page.

## Using Class Suffixes

TODO

## File and Function Reference

### `mosLoadComponents`

Syntax:

```
mosLoadComponents( $name )
```

Loads a component. For example "banners".

### `mosCountModules`

Syntax:

```
mosCountModules( $position_name )
```

Counts the number of modules that shown on the current page in the "position\_name" position.

### `mosLoadModules`

Syntax:

```
mosLoadModules( $position_name [, $style] )
```

Displays all modules that are assigned to the "position\_name" position for the current page. The "style" argument is optional but may be:

- 0 = (default display) Modules are displayed in a column. The following shows an example of the output:

```
<!-- Individual module -->
<table cellpadding="0" cellspacing="0" class="moduletable[suffix]">
  <tr>
    <th valign="top">Module Title</th>
  </tr>
  <tr>
    <td>
      Module output
    </td>
  </tr>
</table>
<!-- Individual module end -->
```

- 1 = Modules are displayed horizontally. Each module is output in the cell of a wrapper table. The following shows an example of the output:

```
<!-- Module wrapper -->
<table cellspacing="1" cellpadding="0" border="0" width="100%">
  <tr>
    <td align="top">
      <!-- Individual module -->
      <table cellpadding="0" cellspacing="0" class="moduletable[suffix]">
        <tr>
          <th valign="top">Module Title</th>
        </tr>
        <tr>
          <td>
            Module output
          </td>
        </tr>
      </table>
      <!-- Individual module end -->
    </td>
    <td align="top">
      <!-- ...the next module... -->
    </td>
  </tr>
</table>
```

- -1 = Modules are displayed as raw output and without titles. The following shows an example of the output

```
Module 1 OutputModule 2 OutputModule 3 Output
```

- -2 = Modules are displayed in X-Mambo format. The following shows an example of the output:

```
<!-- Individual module -->
<div class="moduletable[suffix]">
  <h3>Module Title</h3>
  Module output
</div>
<!-- Individual module end -->
```

Note in all cases that an optional class "suffix" can be applied via the module parameters.

## Administrator Templates

At this time the model for Administrator Templates is still being formed. It is intended that both the Site and Administrator templating systems will merge into a common API in a future version. However, some notes are provided here for reference.

## Module Support

You may include modules in your Administrator templates directly with `mosLoadAdminModule` or in groups, like for the site templates, with `mosLoadAdminModules`. For example:

```
<table width="100%" class="menubar" cellpadding="0" cellspacing="0" border="0">
  <tr>
    <td class="menubackgr"><?php mosLoadAdminModule( 'fullmenu' );?></td>
    <td class="menubackgr" align="right">
      <div id="wrapper1">
        <?php mosLoadAdminModules( 'header', 2 );?>
      </div>
    </td>
    <td class="menubackgr" align="right">
      <a href="index2.php?option=logout" style="color: #333333; font-weight: bold">Logout</a>
      <strong><?php echo $my->username;?>&nbsp;  </strong>
    </td>
  </tr>
</table>
```

The `mosLoadAdminModule` function takes one argument, the name of the module less the "mod\_" prefix. The first cell of the example table loads the Full Menu module (that is, `mod_fullmenu`).

In the second table cell, all the modules assigned to the "header" position are loaded. The second argument is a style setting:

- 0 = just output sequentially what the modules output
- 1 = display each module in a "Tab"
- 2 = display each module wrapped in a `<div>` tag

Formatting for the "header" modules is done completely via css. For example, the "wrapper1" style is defined as:

```
#wrapper1 div {
  border: 0px;
  margin: 0px;
  margin-left: auto;
  margin-right: auto;
  padding: 0px 5px 0px 5px;
  display: inline;
}
```

The modules are encased in plain div tags. To display modules in a column you may add a width attribute and changes the display attribute to "inline".

The following modules are available with the Mambo distribution.

### **mod\_fullmenu**

The Full Menu module displays the traditional DHTML Administrator menu. Content Sections and Components are dynamically added with the remainder of the menu being statically defined.

### **mod\_components**

The Components module displays a full list of the Components and sub-menu items. This is useful where many components are installed and the capacity of the DHTML menus are exceeded.

## **mod\_latest**

The Lastest Items module displays the most recently created content items.

## **mod\_mosmsg**

The Message module displays the message passed in the URL.

## **mod\_online**

The Users Online module displays the number of users logged in.

## **mod\_pathway**

The Pathway module displays an Administrator pathway.

## **mod\_popular**

The Most Popular module displays a list of the most "hit" content items.

## **mod\_stats**

The Menu Stats module shows some statistics about the menus.

## **mod\_toolbar**

The Toolbar module displays the icon toolbar.

## **mod\_unread**

The Unread Messages displays the number of unread private messages.

## **The Control Panel**

The Control Panel for the Administrator is a separate file, `cpanel.php`, that is included with the template. It is a separate file to allow for customisation of this area as different sites and users are likely to have different needs for this valuable piece of screen real estate.

The Control Panel file does not need to be included. In that event the Control Panel will simply display any Administrator Modules published in the "cpanel" position.

The `cpanel.php` file could be as simple as the following example:

```
<?php
/**
 * @version $ Id: cpanel.php,v 1.3 2004/08/12 08:29:21 rcastley Exp $
 * @package Mambo_4.5
 * @copyright (C) 2000 - 2004 Miro International Pty Ltd
 * @license http://www.gnu.org/copyleft/gpl.html GNU/GPL
 * Mambo is Free Software
 */

/** ensure this file is being included by a parent file */
defined( '_VALID_MOS' ) or die( 'Direct Access to this location is not allowed.' );
?>
<table class="adminform">
```

```
<tr>
  <td width="100%" valign="top">
    <?php mosLoadAdminModules( 'cpanel', 1 ); ?>
  </td>
</tr>
</table>
```

# Chapter 3. Modules

## Overview

Modules are small "windows", "blocks" or "portlets" that make up part of the Template. They are the home for menu systems and other small packets of function.

## Writing a Module

As an example, we will create a module that will list any content items that have keywords that match those in the one that is being viewed.

### Where do we start?

First decide on a module name. We are going to call this a Related Items module. The module name will be `mod_relcontent`. All module names must be prefixed with "mod\_" and the "relcontent" just stands for "related content" in our case.

In a scratch area on your file system, create a directory called `mod_relcontent`. In this directory, just create two empty files for the moment; one called `mod_relcontent.php` and the called `mod_relcontent.xml`. Let's build the xml file first. This is a definition file that tells the MOS installer, most importantly, what files are required and other metadata about the module. Copy and paste the following code into the xml file:

```
<?xml version="1.0" ?>
<mosinstall type="module">
  <name>Related Items</name>
  <creationDate>19/Aug/2003</creationDate>
  <author>Andrew Eddie</author>
  <copyright>This template is released under the GNU/GPL License</copyright>
  <authorEmail>eddieajau@users.sourceforge.net</authorEmail>
  <authorUrl></authorUrl>
  <version>1.0</version>
  <description>Shows related content items based on
    keywords in the meta key field</description>
  <files>
    <filename module="mod_relcontent">mod_relcontent.php</filename>
  </files>
</mosinstall>
```

The important tags here are:

name – The name used in menus.

files – There is only one file required for a module.

Save the xml file and move to the php file. Modules user to store their output in a `$content` variable. This is still supported but you can now either use echo statements or escape in and out of php to provide the output. The complete code is shown at the end of this article. Let's step through it.

The first line after the comment block is extremely important. This prevents direct and potentially malicious execution of the script.

Next, a couple of URL variables are collected. When you are writing your modules, never assume that variables you need from the URL are already available. This is not good programming practice and they simply may not be

within the scope of the code. For example, this module code is actually called from within a function, so many global variables simply are not visible. However, several code variables are made available, like `$database`.

The script then checks to see you are viewing a content item. If you are it selects the value of the ``metakey`` field from the item.

## The Database Connector

First you need to "initialise" the query. The main reason for this is that the query string you supply is parsed for a hash-underscore-underscore. This is replaced by the database prefix stored in the system configuration variables.

```
$database->setQuery( "SELECT metakey FROM mos_content WHERE id='$id'" );
```

OK, we've set up our query. This query returns only a single value. This is a really common exercise so we've provided a method called `loadResult()` just to grab that single value. Having got the value and checked that it contains something, we explode the string on commas. We then use arrays to help build a new database query that, in rough pseudo-code says "get all the id's of the content items where their metakey field is like `*this*` or `*that*`".

Now you'll see we have another common operation, that is, getting a list of results from a query. Here we use the database class method called `loadObjectList()` to return an array of rows where each row is stored as an object. The method returns null if the query fails to facilitate error checking.

Having received you're list of matched records, it's now a trivial exercise to loop through the array and print out a list of links.

## Finishing Up

Well, now we've got some code, how do we get it into Mambo. The module installer now requires a zip file of the php and xml file under it's parent directory (that is, in this case, `mod_relcontent`). Zip the two files up. Then, in the Mambo administrator, select `Modules->Install` from the menubar. At the bottom of the list you'll see an upload area. Browse to the zip file and upload it. Viola, all going well, your new module is installed and ready to use.

Go to `Modules->Manage Modules` to publish and select the 'side' and pages you want the module to appear on.

Now, in a couple of content items, put in a couple of different matching keywords in the Metadata tabs. From the front-end, view the items. You should get a list of other records that have matching keywords.

```
<?php
//Related Content//
/**
 * Related Content Module
 * @package Mambo
 * @Copyright (C) 2000 - 2003 Miro International Pty Ltd
 * @ All rights reserved
 * @ Mambo is Free Software
 * @ Released under GNU/GPL License : http://www.gnu.org/copyleft/gpl.html
 * @version $Revision: 1.3 $
 **/

defined( '_VALID_MOS' ) or die( 'Direct Access to this location is not allowed.' );

$option = trim( mosGetParam( $_REQUEST, 'option', null ) );
$task = trim( mosGetParam( $_REQUEST, 'task', null ) );
$id = intval( mosGetParam( $_REQUEST, 'id', null ) );

if ( $option == 'content' && $task == 'view' && $id ) {
```

```

// select the meta keywords from the item
$query = "SELECT metakey FROM mos_content WHERE id='$id'";
$database->setQuery( $query );

if ( $metakey = trim( $database->loadResult() ) ) {
    // explode the meta keys on a comma
    $keys = explode( ',', $metakey );
    $likes = array();

    // assemble any non-blank word(s)
    foreach ( $keys as $key ) {
        $key = trim( $key );
        if ( $key ) {
            $likes[] = $key;
        }
    }

    if ( count( $likes ) ) {
        // select other items based on the metakey field 'like' the keys found
        $query = "SELECT id, title"
            . "\nFROM mos_content"
            . "\nWHERE id<>$id AND state=1 AND access <=$my->gid AND (metakey LIKE '%";
        $query .= implode( "' OR metakey LIKE '", $likes );
        $query .= "%)";

        $database->setQuery( $query );
        if ( $related = $database->loadObjectList() ) {
            foreach ( $related as $item ) {
                echo "<a href='\"index.php?option=content&task=view&id=$item->id\">"
                    . "$item->title</a><br />";
            }
        }
        echo $database->getErrorMsg();
    }
}
?>

```

# Chapter 4. Mambots

## Overview

A Mambot is a small, task-oriented function that intercept content before it is displayed and manipulates it in some way. Mambo provides a number of Mambots in the core distribution.

`mosimage`: This mambot converts `{mosimage}` tags to html `img` tags.

`mospagebreak`: This mambot provides pagination and table of contents functionality with a page.

`moscode`: This mambot replaces the code with `{moscode}{/moscode}` tags with php syntax highlighted code.

`mosvote`:

## Writing a Content Mambot – The Old Way

This example is for writing version 4.5 mambots. You can still write mambots in this styles for 4.5.1 providing they are saved in the `/mambots` directory. These mambots are loaded every time a content item is displayed. So, for the frontpage component, the same file could be loaded five, ten or twenty times depending on the number of content items displayed which is extremely inefficient. You are encouraged to look at upgrading your mambots to gain better performance and flexibility. Additionally, you should upgrade as this method will be deprecated in future versions (after 4.5.1).

For our development example, we'll imagine we are creating a Mambot to replace smile text with smile images.

The file for the mambot will be called `mossmiles.php` and it is saved in `/mambots/mossmiles.php`. It looks like this:

```
if (!defined( '_MOS_CONVERT_SMILES_MAMBOT' )) {
// only define the function once
function MAMBOT_convert_smiles( &$row ) {
    $smiles_src = array(
        ':)',
        ':('
    );
    $prefix = '';
    $smiles_img = array(
        "{$prefix}happy.png{$suffix}",
        "{$prefix}sad.png{$suffix}"
    );
    $row->text = str_replace( $smiles_src, $smiles_img, $row->text );
}
define( '_MOS_CONVERT_SMILES_MAMBOT', 1 );
}
MAMBOT_convert_smiles( $row );
```

A number of things are set up for a mambot. The `mossmiles.php` has access to a number of variables:

`$row` – the current `mosContent` object. This is an object with all of the fields in the `mos_content` table.

`$mosConfig_absolute_path`

`$mask` – a variable holding masks for various display options

In addition, the \$row object has a property called text that the mambot will operate on to make changes to the text.

Row is passed by address to the MAMBOT\_convert\_smiles function so that any changes to the object made within the Mambot function are retained.

Because a mambot is called more than once during the code execution, you must ensure that the function is defined only once. Hence, a constant is defined the first time the mambot is loaded. On subsequent load, the function definition is ignored.

All modifications to the content text are made to the \$row->text property variable.

## Writing a Mambot – The New Way

In version 4.5.1, mambots are able to be triggered to perform at nominated locations in the execution of the Mambo script. At present these locations are few but will grow as the new framework for mambots matures. Mambot files are also loaded only once and a function is registered to be triggered on a particular event.

There are currently three documented event triggers for mambots:

- onPrepareContent
- onSearch
- onLoadEditor

All events require different arguments to be passed to them. This is explained in more detail below.

Mambots are also saved in groups under the /mambots directory. You will see all mambots relating to the searching are in the /mambots/search directory and those relating to content (mosimage, etc) are in the /mambots/content directory. When the search component is invoked, all the mambots in the 'search' group are loaded. Similarly, when content is to be displayed, all the mambots in the 'content' group are loaded.

Let's look at how you would write mambot for each of the supported events.

### An onSearch Mambot

Here is a code framework for a mambot that is triggered by the onSearch event (affectionately known as search bots).

```
<?php
/**
 * @version $Id $
 * @package Mambo_4.5
 * @copyright (C) 2000 - 2004 Miro International Pty Ltd
 * @license http://www.gnu.org/copyleft/gpl.html GNU/GPL
 * Mambo is Free Software
 */

/** ensure this file is being included by a parent file */
defined( '_VALID_MOS' ) or die( 'Direct Access to this location is not allowed.' );

$_PLUGINS->registerFunction( 'onSearch', 'botSearchContacts' );

/**
 * Search method
 * @param array Named 'text' element is the search term
 */
function botSearchContacts( $published, $text ) {
```

```

global $database;

$text = trim( $text );
if ( $text == '' || !$published ) {
    return array();
}

$database->setQuery( "SELECT name AS title,"
    . "\n '' AS created,"
    . "\n misc AS text,"
    . "\n 'Contact' AS section,"
    . "\n CONCAT('index.php?option=com_contact&task=view&id=',id) AS href,"
    . "\n '2' AS browsernav"
    . "\nFROM #__contact_details"
    . "\nINNER JOIN #__categories AS b ON b.id=a.catid AND b.access <='$my->gid'"
    . "\nLEFT JOIN #__sections AS u ON u.id = a.sectionid"
    . "\nWHERE name LIKE '%$text%' OR misc LIKE '%$text%'"
    . "\n AND published='1'"
    . "\nORDER BY name"
);

return $database->loadObjectList();
}
?>

```

Firstly you have the usual header and security gate.

`$_PLUGINS` is a variable exposed to the mambot file when it is included. You don't have to declare it as a global.

It has a method called `registerFunction` in the form:

```
$_PLUGINS->registerFunction( 'event_name', 'function_name' );
```

The event available for searching is 'onSearch' so we use this as our `event_name`.

The `function_name` is the function that we want the Mambo script to execute when it triggers the on search event. You can call it anything you like as long as you ensure it is unique. For this example we have named the function *botSearchContacts*. Functions called by the onSearch event have a required arguments list:

```
function function_name( integer 'published', string 'search_text' )
```

As you can see it accepts one integer argument and one string argument. The first argument indicates whether the Mambot is published. The second is the text to search for. The rest of the function simply queries the database and returns an array of results. The query must return rows with the following fields:

- *title*: A title for the search result
- *created*: The date of the row
- *section*: Where the row is from
- *href*: The href attribute for the link to the item
- *browsernav*: Set to 2 to open in the current window.

## An onPrepareContent Mambot

Here is a code framework for a mambot that is triggered by the onPrepareContent event (this is the traditional Mambot).

```

<?php
/**

```

```

* @version $Id $
* @package Mambo_4.5
* @copyright (C) 2000 - 2004 Miro International Pty Ltd
* @license http://www.gnu.org/copyleft/gpl.html GNU/GPL
* Mambo is Free Software
*/

/** ensure this file is being included by a parent file */
defined( '_VALID_MOS' ) or die( 'Direct Access to this location is not allowed.' );

$_PLUGINS->registerFunction( 'onPrepareContent', 'botMosLink' );

/**
* Link bot
*
* <b>Usage:</b>
* <code>{moslink id="the_id"}</code>
*/
function botMosLink( $published, &$row, $mask=0, $page=0 ) {
    global $mosConfig_absolute_path;

    if (!$published) {
        return true;
    }

    require_once( $mosConfig_absolute_path . '/includes/domit/xml_saxy_lite_parser.php' );

    // define the regular expression for the bot
    $regex = "#{moslink\s*(.*?)}#s";

    // perform the replacement
    $row->text = preg_replace_callback( $regex, 'botMosLink_replacer', $row->text );

    return true;
}

/**
* Replaces the matched tags an image
* @param array An array of matches (see preg_match_all)
* @return string
*/
function botMosLink_replacer( &$matches ) {
    $attrs = @SAXY_Lite_Parser::parseAttributes( $matches[1] );

    $id = @$attrs['id'];

    return '<a href="'.sefRelToAbs( 'index.php?option=com_content&task=view&id=' . $id ).'">Link</a>';
}
?>

```

Functions called by the onPrepareContent event have a required arguments list:

```
function function_name( int $published, object &$row, int $mask=0, int $page=0 )
```

- *published*: 1 if the mambot is published, 0 if not
- *row*: A variable reference to the content object
- *mask*: The current mask, default is 0
- *page*: The current page number, default is 0

The use of the built in `preg_replace_callback` function is a very efficient way of replacing the link tags. Once you define your regular expression, the nominated callback function is called. You simply return the string you want as a replacement for the regular expression.

You may wonder why we pass a 'published' argument. Some Mambots will need to do something if they are not published. For example, the mosimage Mambot needs to remove all of the {mosimage} tags from the text if it is not published.

## An Editor Mambot

The Editor mambots define pluggable WYSIWYG editors that can be made available. Here is a very simple example for no editor. The file none.php looks like:

```
/**
 * @version $Id $
 * @package Mambo_4.5
 * @copyright (C) 2000 - 2004 Miro International Pty Ltd
 * @license http://www.gnu.org/copyleft/gpl.html GNU/GPL
 * Mambo is Free Software
 */

/** ensure this file is being included by a parent file */
defined( '_VALID_MOS' ) or die( 'Direct Access to this location is not allowed.' );

$_MAMBOTS->registerFunction( 'onLoadEditor', 'botNoEditor' );

/**
 * No WYSIWYG Editor
 */
function botNoEditor( $published, $editorName='none' ) {
    if ( $published && $editorName == 'none' ) {
        function initEditor() {
        };
        function getEditorContents( $editorArea, $hiddenField ) {
        }
        // just present a textarea
        function editorArea( $name, $content, $hiddenField, $width, $height, $col, $row ) {
            echo "\n";
            echo '<textarea name="'. $hiddenField. '" id="'. $hiddenField. '" cols="'. $col. '" rows="'. $row. '" st';
            echo $content;
            echo '</textarea>';
        }
    }
}
```

Functions called by the onLoadEvent event have a required arguments list:

```
function function_name( int $published, string $editorName )
```

The editorName argument is the name of the current editor. The mambot function must check that it is the active editor before defining the functions.

The initEditor function is usually called in the <head> element of your template.

The editorArea method function is called to write the editor area.

The getEditorContents is called to retrieve the contents of the editor on a save event.

## Extending Mambots

Mambots and Mambot groups can be used for a variety of purposes. For example, suppose you want to include an HTML Templating system like patTemplate ([www.php-tools.de](http://www.php-tools.de)) and load it in your custom components that use it.

Here's a rough outline of the steps that you would take to set this up.

1. Create a directory called "patTemplate" under the /mambots directory.
2. Unpack the patTemplate distribution file such that patTemplate.php is in the new directory you created. All the patTemplate support files should be in 'another' patTemplate directory. Note that you will also need to include patError.php and patErrorManager.php in this directory. The /mambots/patTemplate directory should now look something like this:

```
/mambots
 /patTemplate
  patTemplate.php
  patError.php
  patErrorManager.php
 /patTemplate (many files provided under this directory)
```

3. Create an XML setup file for the Mambot listing all the files under the /mambot/patTemplate directory. It might look something like this:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mosinstall type="mambot" group="patTemplate" version="4.5.1">
  <name>patTemplate</name>
  <creationDate>01/09/2004</creationDate>
  <version>3.0 Beta 1</version>
  <author>Stephan Schmidt</author>
  <license>http://www.gnu.org/copyleft/lgpl.html GNU/LGPL</license>
  <authorEmail>schst@php.net</authorEmail>
  <authorUrl>www.php-tool.de</authorUrl>
  <description>Powerful templating engine</description>
  <files>
    <filename>patError.php</filename>
    <filename>patErrorManager.php</filename>
    <filename>patTemplate/Dump/Html.php</filename>
    <filename>patTemplate/Dump.php</filename>
    <filename>patTemplate/Function/Time.php</filename>
    <filename>patTemplate/Function.php</filename>
    <filename>patTemplate/InputFilter/StripComments.php</filename>
    <filename>patTemplate/InputFilter.php</filename>
    <filename>patTemplate/Modifier/HTML/Img.php</filename>
    <filename>patTemplate/Modifier/Wordwrapper.php</filename>
    <filename>patTemplate/Modifier.php</filename>
    <filename>patTemplate/Module.php</filename>
    <filename>patTemplate/OutputCache.php</filename>
    <filename>patTemplate/OutputFilter/Gzip.php</filename>
    <filename>patTemplate/OutputFilter/StripWhitespace.php</filename>
    <filename>patTemplate/OutputFilter.php</filename>
    <filename>patTemplate/Reader/File.php</filename>
    <filename>patTemplate/Reader/IT.php</filename>
    <filename>patTemplate/Reader/String.php</filename>
    <filename>patTemplate/Reader.php</filename>
    <filename>patTemplate/TemplateCache/File.php</filename>
    <filename>patTemplate/TemplateCache.php</filename>
    <filename mambot="patTemplate">patTemplate.php</filename>
    <filename>patTemplate.xml</filename>
  </files>
</mosinstall>
```

Notice that the file patTemplate.php is given the "mambot" attribute to signify that this is the file to be loaded by the Mambot handler.

4. Zip up the contents of the /mambo/patTemplate directory and install the Mambot. You should see it listed in the Mambot Manager.
5. To use this in your code, you would use something similar to the following:

```
global $_MAMBOTS;

if (!$_MAMBOTS->loadBotGroup( 'patTemplate' )) {
  die( 'This component requires the patTemplate Plugin' );
}
```

```
$tmpl =& new patTemplate( 'html' );
```

There are obviously many other applications for this technique.

# Chapter 5. Components

## Overview

Components are the main functional units that display in your template, like the content management system, contact forms, web links and the like.

This chapter serves to provide you with a number of useful examples that helps you learn how to create Components.

While we have gone to great lengths to make Mambo easy for content providers to use, we have equally spent a lot of time developing a flexible framework for developers to extend the capabilities of Mambo without having to touch the core code.

Please note that line numbers shown in code examples may not be sequential.

## Scoping and Planning

Let's work through the process of creating a customised component and supporting module.

A client has asked you for what you know to be a module to display in one of the side panels of there existing Mambo intranet. This module needs to be a list of particular users and show whether they are 'in' or 'out' of the office. The client also wants a small form under the list of users that allows you (the logged in user) to check out with a short text message of where you will be and the time you will be back.

Thinking astutely, you ask the client whether they would want to do anything else with the system down the track. The client asks whether this could be used to track overtime. You suggest that this can be done as a later stage of the commission but that you will take this into consideration while designing the first stage. You immediately think to yourself that the client may end up wanting to compile time sheets from the information.

## Developing the Database Schema

We now have a rough brief to work with. What do we have to consider and how are we going to build it in the Mambo framework.

Firstly what information do we need to store?

The users that will be displayed on the in/out board.

If a user checks out, when did they do it, where are they and when are they coming back?

The first thought is simply to add a few fields to the mos\_users table. OK, that's quick and easy but what happens if the client wants to upgrade their Mambo down the track? The Developers aren't going to be too sympathetic to thirdparties that hack into the core schema. So, it's obvious that we need to create a new table to support our Client's information without tampering with the core schema.

Let's call our new table mos\_com\_inout. This makes it obvious that the table is associated with the in/out component that we are going to build. Only users in this table will be displayed on the in/out board.

We will link to the mos\_users table via the id field (we shouldn't use the name field as this could change) as this is the primary key for the table. We need a few fields to store when the user checked out, where they are and when

they are coming back. So, the sql for the mos\_com\_inout table might look like:

```
CREATE TABLE `mos_com_inout` (  
  `id` int(10) unsigned NOT NULL auto_increment,  
  `user_id` int(10) unsigned NOT NULL default '0',  
  `time_out` datetime NOT NULL default '0000-00-00 00:00:00',  
  `time_in` datetime NOT NULL default '0000-00-00 00:00:00',  
  `location` varchar(100) NOT NULL default '',  
  `catid` int(10) unsigned NOT NULL default '0',  
  `checked_out` int(10) unsigned NOT NULL default '0',  
  `checked_out_time` datetime NOT NULL default '0000-00-00 00:00:00',  
  PRIMARY KEY (`id`)  
) TYPE=MyISAM;
```

We've included a primary key in this table, the `id` field, to help us edit specific records if we need to. We've also included a `catid` field. This is so we can link back to the categories table. You just have a hunch that your Client might ask you for a couple of different in/out boards for different groups of users. Using the Categories feature of Mambo will serve this purpose quite well without you having to write any additional code to support it.

You'll also see two other fields, `checked\_out` and `checked\_out\_time`. These two fields are required by the MOS framework to check in and out the records in the mos\_inout table when they are being edited (don't confuse this with the users checking in and checking out of the office).

So we have our schema planned, at least for stage 1. Next we need to think about what we need to provide in Mambo.

## Developing the Administrator Component

We need to develop a component in the Administrator in order for your Client to maintain who should be shown on the in/out board. We also need to plug into the Categories feature to enable us to provide different groups of users for different in/out boards.

So, where do we start?

### Adding Menu Items

Most components will have some sort of administration function. This is placed under the menu item called 'Components' on the menu bar. All these items are generated dynamically from the database. When you finally deploy your component, it will be delivered in a zipped file with an xml definition file telling the Component Installer what files to copy where, what tables to create and what menu items to provide. Well, it's a bit premature to do that during the development stage so we'll have to add the menu items by hand.

We need to add a few rows to the mos\_components table. Have a look at this table. It has the following fields:

**Table 5–1. mos\_components Table Structure**

id	the primary key for each record in the table
name	the proper-case name of the component
link	this is the link for the front-end (eg, option=com_inout)
menuid	has the component been attached to a main menu item
parent	this is the parent menu item in the components dropdown menu
admin_menu_link	the link for the menu item in the administrator

admin_menu_alt	the alt text for the graphic in the menu
admin_menu_image	the image associated with the menu text (that is, the name)
option	the component option, that is, the name of the component directory (eg, com_inout)
ordering	the order the component appears in the menu
iscore	true if this is a core component of Mambo
comdir	
admindir	

So, let's create a menu item called 'In/Out Board' and a fly-out menu with two item: 'Manages Users' and 'In/Out Categories'. The SQL to do this is:

```
INSERT INTO `mos_components`
VALUES
(500, 'In/Out Board', '', 0, 0, '', 'In/Out Board Management', 'com_inout', 0,
'js/ThemeOffice/module.png', 0, '', '');
INSERT INTO `mos_components`
VALUES
(501, 'Manage Users', '', 0, 500, 'option=com_inout', 'User Management',
'com_inout', 1, 'js/ThemeOffice/module.png', 0, '', '');
INSERT INTO `mos_components`
VALUES
(502, 'In/Out Categories', '', 0, 500, 'option=categories&section=com_inout',
'In/Out Categories', 'com_inout', 2, 'js/ThemeOffice/module.png', 0, '', '');
```

Execute these three lines of code in your database client. Refresh the Mambo Administrator and you should see the new entries in the menu.

## Creating the Administrator Interface

There are a number of files required to support a component in the MOS Administrator. First you need to create a directory under the /administrator/components directory. You need to prefix this directory with com\_. You also need four (possibly five) files; two to support the toolbar, two to support the component itself and maybe one other to define the table class.

Given that our component is called mrx\_inout, the directory structure will look like this:

```
/administrator
 /components
  /com_mrx_inout
   admin.mrx_inout.php
   admin.mrx_inout.html.php
   toolbar.mrx_inout.php
   toolbar.mrx_inout.html.php
 /components
  /com_mrx_inout
   mrx_inout.class.php
```

Each of these files has a specific function:

admin.mrx\_inout.php

This file is the event handler for the component.

admin.mrx\_inout.html.php

This file provides the html output to support the event handler.

`toolbar.mrx_inout.php`

This file is the event handler that controls the display in the toolbar.

`toolbar.mrx_inout.php`

This file provides the html output to support the toolbar.

`mrx_inout.class.php`

This file provides the class definition for the database table class(es) for the component. A single file is used that is used by the front–end and the administrator (in circumstances where the component is not used in the frontend at all, like for News Feeds, this file can be place under the administrator tree with the rest of the component files).

It is the custom in Mambo to, wherever possible, separate the event handlers from the presentation layer. As much code crunching as is practical is done in the events handler before handing over to output the html. Hence, you will generally see 'pairs' of files to handle and then display events.

## The Component Toolbar

You need two files to control the display of the toolbar in the MOS Administrator. One file is the event handler (`toolbar.*.php`) and one controls the html output (`toolbar.*.html.php`). This segregation of business logic and display is common throughout most of the Mambo codebase.

Let's have a look at the file that handles the html output first. Create a file called `toolbar.mrx_inout.html.php` in the `/administrator/components/com_mrx_inout` directory. Copy the following code into it and then we'll dissect what's happening.

```
1: <?php /* $Id $ */
2:
3: /**
4:  * In/Out Board Menubar HTML Writer
5:  * @package MOS-ROX
6:  * @license http://www.gnu.org/copyleft/gpl.html. GNU Public License
7:  * @version 4.5.1
8:  */
9:
10: // ensure this file is being included by a parent file
11: defined( '_VALID_MOS' ) or
12:     die( 'Direct Access to this location is not allowed.' );
13:
14: class TOOLBAR_mrx_inout {
15:     /**
16:     * Draws the menu to add or edit an item
17:     */
18:     function _EDIT() {
19:         mosMenuBar::startTable();
20:         mosMenuBar::save();
21:         mosMenuBar::cancel();
22:         mosMenuBar::spacer();
23:         mosMenuBar::endTable();
24:     }
25: }
26:
27: function _DEFAULT() {
28:     mosMenuBar::startTable();
29:     mosMenuBar::addNew();
30:     mosMenuBar::editList();
31: }
```

```
31: mosMenuBar::deleteList();
32: mosMenuBar::spacer();
33: mosMenuBar::endTable();
34: }
35: }?>
```

*Line 13:*

This is an essential part of the MOS security system that prevents a user from directly executing this script, for malicious purposes or otherwise.

*Line 15–35:*

These lines define a class to handle the output of the toolbar. The class has two methods, one to display the toolbar for editing a record (`_EDIT`), and another to display for any other task (`_DEFAULT`). Each of these methods are an assembly of API methods to create the toolbar. These methods are part of the `mosMenuBar` class. Because these are what might be called 'helper' or 'utility' methods of the class that don't interact with the properties of the class, we don't have to instantiate (this is, create) the class to use the methods. We simply call them via the `class_name::method_name()` syntax.

The `_EDIT` method will display a Save and a Cancel button. The `startTable` and `endTable` methods merely provide the html to open (`<table><tr>`) and close (`</tr></table>`) the table wrapping the toolbar. The `spacer` method provide a wide empty cell to keep the toolbar buttons bunched up of the left.

The `_DEFAULT` method will display a New, Edit and Delete button.

Now we need to build the event handler to display the correct toolbar. Create a file called `toolbar.mrx_inout.php` in the `/administrator/components/com_mrx_inout` directory. Copy the following code into it and then we'll again dissect what's happening.

```
<?php /* $Id: components.xml,v 1.6 2004/08/23 16:19:44 rcastley Exp $ */
2:
3: /**
4: * In/Out Board Menubar Handler
5: * @package MOS-ROX
6: * @license http://www.gnu.org/copyleft/gpl.html. GNU Public License
7: * @version 4.5.1
8: */
9:
10: // ensure this file is being included by a parent file
11: defined( '_VALID_MOS' ) or
12:     die( 'Direct Access to this location is not allowed.' );
13:
14: require_once( $mainframe->getPath( 'toolbar_html' ) );
15:
16: switch ( $task ) {
17:
18: case "new":
19: case "edit":
20:     TOOLBAR_mrx_inout::_EDIT();
21:     break;
22:
23: default:
24:     TOOLBAR_mrx_inout::_DEFAULT();
25:     break;
26: }
27: }
28: ?>
```

*Line 15:*

This line pulls in the previous file (`toolbar.mrx_inout.html.php`). The variable `$mainframe` is a class of utility methods. When it is initialised it, amongst other things, checks for the existence of toolbar and class files.

These could be in a number of locations as the method supports some legacy styles of coding. In our case, the `getPath` method would return:

```
/administrator/components/com_mrx_inout/toolbar.mrx_inout.html.php
```

The argument passed to the method is shorthand for asking the function to retrieve the path for the file that handling the html output for the toolbar. You could have just as easily written:

```
15: require_once( $mainframe->getPath(
' /administrator/components/com_mrx_inout/toolbar.mrx_inout.html.php ' ) );
```

*Line 17:*

Starts the switch block for the task.

*Line 19–21:*

If the task is either 'new' or 'edit' then we call the function to write the html for `EDIT_MENU`. Remember in the `toolbar.mrx_inout.html.php` file we created the `menu_mrx_inout` class with two methods. Again, as neither of these two functions interact with the properties of the class we can call them directly using the `class_name::method_name()` syntax. For reference, the double-colon (`::`) is called the scope indirection operator.

*Line 24–25:*

For any other task we call the function to write the `_DEFAULT` toolbar.

## The Database Class Handlers

In Mambo we provide a special class that relieves the developer of a lot of low level database work on tables. This special class is called `mosDBTable` and will have already been declared for you.

You don't use this class directly. In Object Oriented Programming (OOP) terms it's called an Abstract Class. We use this class to derive a new class which we will call `mosInOut`.

Create a file called `mrx_inout.class.php` in the `//components/com_mrx_inout` directory.

Copy the following code into it and then we'll look at what is happening in this file.

```
1: <?php /* $Id: components.xml,v 1.6 2004/08/23 16:19:44 rcastley Exp $ */
2:
3: /**
4: * In/Out Board Main Class
5: * @package MOS-ROX
6: * @license http://www.gnu.org/copyleft/gpl.html. GNU Public License
7: * @version 4.5.1
8: */
9:
10: // ensure this file is being included by a parent file
11: defined( '_VALID_MOS' ) or
12:     die( 'Direct Access to this location is not allowed.' );
13:
14: /**
15: * MRX InOut table class
16: */
```

```

18: class mosInOut extends mosDBTable {
19:     /** @var int Primary key */
20:     var $id=null;
21:     /** @var int */
22:     var $user_id=null;
23:     /** @var datetime */
24:     var $time_out=null;
25:     /** @var datetime */
26:     var $time_in=null;
27:     /** @var string */
28:     var $location=null;
29:     /** @var int */
30:     var $catid=null;
31:     /** @var int */
32:     var $checked_out=null;
33:     /** @var time */
34:     var $checked_out_time=null;
35:     /** @var int */
36:     var $ordering=null;
37:
38:     /**
39:      * @param database A database connector object
40:      */
41:     function mosInOut() {
42:         global $database;
43:         $this->mosDBTable( '#__mrx_inout', 'id', $database );
44:     }
46: }
48: ?>

```

*Line 18:*

This line shows us the syntax for deriving a class in PHP. This tells PHP to create a class called mosInOut based on the existing class mosDBTable. Our class will 'inherit' all the properties and methods in the mosDBTable class.

We could redefine all of them if we wanted to, but most have been written generically so that you don't. Only functions that work on information peculiar to your database table will be required to be 'overridden' (for example, the 'check' method).

*Line 19–36:*

All the comments in these lines are written in a format that an application called phpDocumentor can parse. We'll talk about this at the end of this book. In between the comments are a number of class variable declarations. What variables do I declare? That's easy. Declare a variable for each field in your database table. Make sure all the variables match the field names. Also make sure you set each variable to null otherwise problems will arise in some versions of 4.1.x of PHP.

*Lines 41–44:*

We need to create a class constructor method. This **MUST** be the same name as the class (you will often get errors or experience unusual behaviour if you forget).

*Line 43:*

To complete the construction of your class, you invoke the base class's constructor, passing it the name of your table, the primary key of your table and the database connector.

## The Component Presentation Layer

Here is where the first half of the serious work starts. We need to create a presentation layer to display the html for different events that are going to be triggered by our component. The presentation layer is a file that builds up as we develop additional functionality.

Let's start small. Let's examine what we need just to display a list of users allocated to an in/out board.

Create a file called `admin.mrx_inout.html.php` in the `/administrator/components/com_mrx_inout` directory. Copy the following code into it and then we'll dissect what's happening. Please don't worry about the fact that the line numbers are not sequential. There are just for reference purposes.

```
1: <?php /* $Id $ */
2:
3: /**
4:  * In/Out Board Main HTML Writer
5:  * @package MOS-ROX
6:  * @license http://www.gnu.org/copyleft/gpl.html. GNU Public License
7:  * @version 4.5.1
8:  */
9:
10: // ensure this file is being included by a parent file
11: defined( '_VALID_MOS' ) or
12:     die( 'Direct Access to this location is not allowed.' );
13:
14: class HTML_mrx_inout {
15:     /**
16:     * Display the list of users
17:     */
18:     function listUsers( &$rows, &$lists, &$pageNav, $option ) {
19:         global $my;
20:         ?>
21:         <form action="index2.php" method="post" name="adminForm">
22:         <table class="adminheading">
23:         <tr>
24:         <th>MRX In/Out Board - User List</th>
25:         <td nowrap="true">Display #</td>
26:         <td <?php echo $pageNav->getLimitBox(); ?> </td>
27:         <td width="right"> <?php echo $lists['catid']; ?> </td>
28:         </tr>
29:         </table>
30:
31:         <table class="adminlist">
32:         <tr>
33:         <th width="20">#</th>
34:         <th width="20"><input type="checkbox" name="toggle" value=""
35:             onclick="checkAll(<?php echo count( $rows ); ?>);" /></th>
36:         <th class="title" width="50%">Users</th>
37:         <th width="15%" align="left">Category</th>
38:         <th colspan="2">Reorder</th>
39:         </tr>
40:         <?php
41:         $k = 0;
42:         for ( $i=0, $n=count( $rows ); $i < $n; $i++ ) {
43:             $row = &$rows[$i];
44:             ?>
45:             <tr class="<?php echo "row$k"; ?>">
46:             <td width="20" align="center"><?php echo $i+$pageNav->
47:             >limitstart+1; ?></td>
48:             <td width="20">
49:             <?php if ( $row->checked_out && $row->checked_out != $my->id ) { ?>
50:             &nbsp;?>
51:             } else { ?>
```

```

51: <input type="checkbox" id="cb<?php echo $i;?>" name="cid[]"
value="<?php echo $row->id; ?>" onclick="isChecked(this.checked);" />
52: <?php } ?>
53: </td>
54: <td width="50%"><a href="#edit" onclick="return
listItemTask('cb<?php echo $i;?>', 'edit')"><?php echo $row->name; ?> </a></td>
55: <td width="50%"><?php echo $row->catname;?></td>
56: <td width="20">
57: <?php if ($i > 0 || ($i+$pageNav->limitstart > 0)) { ?>
58: <a href="#reorder" onclick="return listItemTask('cb<?php echo
$i;?>', 'orderup')">
59: 
60: </a>
61: <?php } ?>
62: </td>
63: <td width="20">
64: <?php if ($i < $n-1 || $i+$pageNav->limitstart < $pageNav->total-1) {
?>
65: <a href="#reorder" onclick="return listItemTask('cb<?php echo
$i;?>', 'orderdown')">
66: 
67: </a>
68: <?php } ?>
69: </td>
70: </tr>
71: <?php
72: $k = 1 - $k;
73: }
74: ?>
75: <tr>
76: <th align="center" colspan="10"> <?php echo $pageNav-
>writePagesLinks(); ?></th>
77: </tr>
78: <tr>
79: <td align="center" colspan="10"> <?php echo $pageNav-
>writePagesCounter(); ?></td>
80: </tr>
81: </table>
82:
83: <input type="hidden" name="option" value="<?php echo $option;?>" />
84: <input type="hidden" name="task" value="" />
85: <input type="hidden" name="boxchecked" value="0" />
86: </form>
87: <?php
88: }
143: }
144:
145: ?>

```

Looks a bit like spaghetti and meatballs? Well, it's not too hard. Let's walk you through.

*Line 15:*

We wrap the presentation methods in a class. You can call it what you like but the customary convention is `HTML_component_name`.

*Line 19:*

This is the first method we will build into the presentation layer. We've called it `listUsers`. We pass four arguments to the function. `&$rows` will be an array of objects that equate to rows in our database table.

Notice the ampersand. This means we are passing this array by its reference. If we didn't, then a copy of the array would be passed to the method. If this array took up, for example, 100kb of memory, we would have just added another 100kb to the memory stack. Passing by reference means we work on the original array and is far more memory efficient. `&$catlist` will be a string of preformatted html for a select list for categories. `&$pageNav` is an object that helps us with pagination of the list. `$option` is the option argument passed via the URL or a submitted form. We could hardcode it but, again, it is customary to pass the option around. This makes it easy to cut-and-paste code from other components already built without changing a lot of standard code.

*Lines 22 & 86:*

The entire list is wrapped in a HTML form named "adminForm". The toolbar buttons expect this form to exist.

*Line 23–30:*

This is a table that displays a heading, the display limit and a drop-down list to filter by category.

*Line 27:*

The pageNavigation class has a method called "writeLimitBox". This method handles the display of the drop-down list for the number of records to display on a page.

*Line 28:*

We will have already assembled the HTML for the category drop-down list using a number of utility functions (we'll come to that shortly). All you have to do is output the variable.

*Line 32–39:*

We have started another table to display the list of users. This code block defines the column headings. Line 35 contains a special checkbox that is used to select or unselect all of the items shown in the list.

## The Component Event Handler

This is where the other half of the serious work happens (actually it happens first). As for the presentation layer, event handler also builds up as we develop additional functionality. In fact, there are usually many more events handled than there are requirements to display them.

We've already created the code to display the user list and let's build the event handler for this first.

Create a file called `admin.mrx_inout.php` in the `/administrator/components/com_mrx_inout` directory. Copy the following code into it and then we'll dissect what's happening.

```
1: <?php /* $Id $ */
2:
3: /**
4:  * In/Out Board Main Handler
5:  * @package MOS-ROX
6:  * @license http://www.gnu.org/copyleft/gpl.html. GNU Public License
7:  * @version 4.5.1
8:  */
9:
10: // ensure this file is being included by a parent file
11: defined( '_VALID_MOS' ) or
12:     die( 'Direct Access to this location is not allowed.' );
13:
14: require_once( $mainframe->getPath( 'admin_html' ) );
```

```

16: require_once( $mainframe->getPath( 'class' ) );
17:
18: $cid = mosGetParam( $_REQUEST, 'cid', array() );
19:
20: switch ( $task ) {
48:
49: default:
50: listUsers( $option );
51: break;
52: }
53:
54: /**
55: * List users
56: * @param string The current GET/POST option
57: */
58: function listUsers( $option ) {
59: global $database;
60:
61: $catid = intval( mosGetParam( $_REQUEST, 'catid', 0 ) );
62: $limit = intval( mosGetParam( $_REQUEST, 'limit', 10 ) );
63: $limitstart = intval( mosGetParam( $_REQUEST, 'limitstart', 0 ) );
64:
65: // get the total number of records
66: $database->setQuery( "SELECT count(*) FROM #__mrx_inout"
67: . ( $catid ? "\nWHERE catid='$catid'" : "" )
68: );
69: $total = $database->loadResult();
70: echo $database->getErrorMsg();
71:
72: if ( $limit > $total ) {
73: $limitstart = 0;
74: }
75:
76: // get the subset (based on limits) of required records
77: $database->setQuery( "SELECT a.*, u.name, c.name AS catname"
78: . "\nFROM #__mrx_inout AS a"
79: . "\nINNER JOIN #__categories AS c ON c.id = a.catid"
80: . "\nINNER JOIN #__users AS u ON u.id = a.user_id"
81: . ( $catid ? "\nWHERE a.catid='$catid'" : "" )
82: . "\nORDER BY a.catid, a.ordering"
83: . "\nLIMIT $limitstart,$limit"
84: );
85:
86: $rows = $database->loadObjectList();
87: if ( $database->getErrorNum() ) {
88: echo $database->stderr();
89: return false;
90: }
91: $lists = array();
92: // get list of categories
93: $categories[] = mosHTML::makeOption( '0', 'Select Category' );
94: $categories[] = mosHTML::makeOption( '0', '- All Categories' );
95: $database->setQuery( "SELECT id AS value, title AS text FROM
96: #__categories"
97: . "\nWHERE section='$option' ORDER BY ordering" );
98: $categories = array_merge( $categories, $database->loadObjectList() );
99: $lists['catid'] = mosHTML::selectList( $categories, 'catid',
100: 'class="inputbox" size="1" onchange="document.adminForm.submit();" ',
101: 'value', 'text', $catid );
102: unset( $categories );
103:
104: require_once("includes/pageNavigation.php");
105: $pageNav = new mosPageNav( $total, $limitstart, $limit );
106: HTML_mrx_inout::listUsers( $rows, $lists, $pageNav, $option );

```

Yikes, that looks complicated at first glance! Well, let's try and make it a bit easier to follow:

Line 13:

Remember this from the toolbar. This is an essential part of MOS's security system. Don't forget to put this in all of your files.

--- INCOMPLETE ---

# Chapter 6. Language Support

## Overview

The core language support is focused on the static text which is used for the different presentation outputs. Most of the text which you see in the frontend presentation of your web site is defined within the language files and can be changed easily by adding new languages within the language manager (see administrator guide). The language support of Mambo is based on the requirement to have a web site in one certain language. With that requirement the feature enables you to control the following elements of the language presentation:

- Definition of language for static text elements in the frontend
- Definition of locale settings for output of dates, currencies, ... (setLocale setting)
- Definition of ISO character sets for the HTML header meta tags

The feature to provide a multi lingual web site will be supported by Mambo core in a future release (see roadmap). At the moment the support for the translation of dynamic content within your website is provided by a third party component (<http://mamboforge.net/projects/mambelfish>).

## Installation and global configuration of languages

The language file support is based on the language files stored in the /languages directory. The management of these files can be done thru the language manager within the administration or by copying the content of the language packages (language file and XML declaration for the language) into the /language directory.

Within Mambo each website has to have one active language which is English by default. To change this language you have to install a second language and either publish this language within the language manager or activate it in the global configuration. Both action require administrator rights within your administration and a writeable configuration.php file.

For further details of installation a language please see the administrator documentation.

## Creation of language files

To create your own language file it is necessary that you use exactly the same defines within the code. We suggest to copy an existing language and just translate the content of the defines.

## Files within a language package

The following files are used within a language package. These files should be provided for each language in order to install reliable within the language manager.

**Table 6–1. File within the language package**

<language_name>.php	Basic file containing the static text defines for the new language. The file name is normally the language name without any special chars and blanks. Also the filename should be all lower case in order to avoid problems with the filehandling on different systems.
<language_name>.ignor.php	Definitions of search words that will be ignored for this language.
<language_name>.xml	Language configuration file. This file contains all information about the language

handling and which filenames to be used within the package. In order to provide installation support thru the administrator this fill must be contain in the language package.

## Manipulating the static text

The following code is an extract from the `english.php` language file. In order to translate the static text of your site it is needed to change the english text. Everything else in the file (specially the define names) should stay like they are.

```
<?php
/**
 * @version $Id: languages.xml,v 1.3 2004/08/20 13:58:23 akede Exp $
 * @package Mambo_4.5
 * @copyright (C) 2000 - 2004 Miro International Pty Ltd
 * @license http://www.gnu.org/copyleft/gpl.html GNU/GPL
 * Mambo is Free Software
 */

/** ensure this file is being included by a parent file */
defined( '_VALID_MOS' ) or die( 'Direct Access to this location is not allowed.' );

/** common */
DEFINE( "_NOT_AUTH", "You are not authorized to view this resource." );
DEFINE( "_DO_LOGIN", "You need to login." );
DEFINE( '_VALID_AZ09', "Please enter a valid %s. No spaces, more than %d characters and contain 0-9,a-z" );
DEFINE( '_CMN_YES', "Yes" );
DEFINE( '_CMN_NO', "No" );
DEFINE( '_CMN_SHOW', "Show" );
DEFINE( '_CMN_HIDE', "Hide" );
```

The translated version (here into German) would look like:

```
<?php
// $Id: languages.xml,v 1.3 2004/08/20 13:58:23 akede Exp $
/**
 * Content code
 * @package Mambo Open Source
 * @Copyright (C) 2000 - 2003 Miro International Pty Ltd
 * @ All rights reserved
 * @ Mambo Open Source is Free Software
 * @ Released under GNU/GPL License : http://www.gnu.org/copyleft/gpl.html
 * @version $Revision: 1.3 $
 **/

defined( '_VALID_MOS' ) or die( 'Direkter Zugriff zu diesem Bereich ist nicht erlaubt.' );

// common
DEFINE( "_NOT_AUTH", "Du bist nicht berechtigt, diesen Bereich zu sehen." );
DEFINE( "_DO_LOGIN", "Du musst dich anmelden." );
DEFINE( '_VALID_AZ09', "%s ist nicht zul&auml;ssig. Bitte keine Leerzeichen, mindestens %d Stellen, 0-9" );
DEFINE( '_CMN_YES', "Ja" );
DEFINE( '_CMN_NO', "Nein" );

DEFINE( '_CMN_NAME', "Name" );
DEFINE( '_CMN_DESCRIPTION', "Beschreibung" );
DEFINE( '_CMN_SAVE', "Speichern" );
```

## Language configuration XML

The configuration XML is needed for the installation and basic language settings. In future this XML file will get more importance because it will contain language related information such as the usual charsets and ISO code references for the language. So please be sure to provide this XML file in any language package you distribute.

```
<xml version="1.0" encoding="iso-8859-1">
<mosinstall type="language">
  <name>English</name>
  <version>1.14</version>
  <creationDate>07/07/2004</creationDate>
  <author>Mambo Project</author>
  <authorEmail>admin@mamboserver.com</authorEmail>
  <files>
    <filename>english.php</filename>
    <filename>english.xml</filename>
  </files>
</mosinstall>
```

This XML file covers the standard installer tags and attributes.

# Chapter 7. Packaging Custom Work

## Overview

Mambo comes with an easy-to-use installer for modules, mambots, components and templates. Using the installer, the site administrator can add features and templates by uploading one zipped file. This topic describes how to create mambot, module, component and template psetup files that are used by the installer.

## The XML Setup File

All installations using the installer require a text file marked up in XML. The XML file describes the installation, its authors, and the files to be installed. Only a passing familiarity with XML is required. Each XML file begins with the prologue, `<?xml version="1.0" ?>`. Thereafter, several nested sections appear, all of which are nested in the root, `<mosinstall>`. The `mosinstall` tag has at least two attributes: "type" is the type of element you are installing and "version" is the version of Mambo that the element is written for. A third attribute called "client" is used for some Administrator elements.

```
<?xml version="1.0" ?>
<mosinstall type="installtype" version="4.5.1">
. . .
</mosinstall>
```

Tags that are common to all types of installation are listed below.

`<name>the name</name>` The name tag is required. It is used in menus, etc.

`<creationDate>01/08/2004</creationDate>` The creation date for the XML file or component.

`<author>Test Designer</author>` The author of the Component, Module or Template

`<copyright>This template is released under the GNU/GPL License</copyright>` Copyright Information

`<authorEmail>info@domain.com</authorEmail>` The email address of the author

`<authorUrl>www.domain.com</authorUrl>` The author's URL

`<version>1.0</version>` The version of the package

```
<files>
  <filename>filename.xxx</filename>
</files>
```

Optional, but if you don't have it, no files will be installed. The files section is used to tell the installer which files it shall install. There is no limit in the number of files you can have in this section. Depending on the the installation type, files are copied to `/modules`, `/components/[component_name]` or `/templates/[template_name]`.

## Parameters

Module and Component setup file may also have a block defining parameters, for example:

```
<params>
```

```
<param name="count" type="text" default="5" label="Number of items"
  description="The number of items to show" />
</params>
```

Refer to the appendix on parameters for more information.

## XML Do's and Don'ts

All XML files must be well-formed without exception. The XML parser used by Mambo will not parse malformed XML documents. While there are numerous rules as to what makes a well-formed XML document, here is a list of the major checkpoints:

1. Your xml file must not have any whitespace or character in front of the XML declaration. For example, this is legal if it shows the start of a file:

```
<?xml version="1.0" ?>
```

This is not legal:

```
<?xml version="1.0" ?>
```

Nor is this:

```
bad<?xml version="1.0" ?>
```

2. You must match every starting tag with the same ending tag. Note that tag names are case sensitive. in XML. For example:

```
<LeGaL>This is ok</LeGaL>
```

```
<illegal>This is not</ILLEGAL>
```

3. You may nest element tags but they may not overlap. For example, the following is malformed:

```
<name>Main <author>Harry</name></author>
```

4. There can only be exactly one root element. For example, the following is a malformed document because there are two root elements:

```
<?xml version="1.0" ?>
<mosinstall type="module">
<!-- the XML definition -->
</mosinstall>
<mosinstall type="component">
<!-- the XML definition -->
</mosinstall>
```

5. All attributes in a tag must be quoted. For example, the following is malformed because of the missing quotes, as you might see in a HTML document:

```
<mosinstall type=module>
```

6. You may not have more than one tag attribute with the same name. For example, the following is malformed:

```
<mosinstall type="module" type="component">
```

7. You must escape all < and & signs that occur within the character data or attributes of a tag. For example:

```
<menu link="option=com_foo&task=bar">Sub &lt; menu 1</menu>
```

8. To use HTML markup in the character data of a tag, you must enclose the text in a CDATA section. For example, the following example allows for the &copy; markup to be placed in the copyright tag:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mosinstall type="template">
  <name>How flung</name>
  <copyright><![CDATA[&copy; 2004 Me]]></copyright>
```

There are a number of tools to use to check that your xml is well-formed. One method is to open the xml file in a modern browser. If it is well-formed you will likely see a rendering like:

```
- <mosinstall type="module">
  <name>Archived Content</name>
  <author>Mambo Project</author>
  <creationDate>July 2004</creationDate>
  <copyright>(C) 2000 - 2004 Miro International Pty Ltd</copyright>
```

A malformed xml file may look like this:

**XML Parsing Error: mismatched tag. Expected: </param>.**  
**Location: file:///D:/mamboforge/4.5/modules/custom.xml**  
**Line Number 23, Column 4:**

```
-----^
      </params>
```

## A Mambot Setup File

The XML for a module installation could look like this:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mosinstall type="mambot" group="content" version="4.5.1">
  <name>Smiles converter</name>
  <creationDate>01/08/2004</creationDate>
  <author>Mambo</author>
  <copyright>(C) 2000 - 2004 Miro International Pty Ltd</copyright>
  <license>http://www.gnu.org/copyleft/gpl.html GNU/GPL</license>
  <authorEmail>admin@mamboserver.com</authorEmail>
  <authorUrl>www.mamboserver.com</authorUrl>
  <description>Displays a menu.</description>
  <files>
    <filename mambot="smiles.bot">smiles.bot.php</filename>
  </files>
</mosinstall>
```

The mosinstall element has an addition attribute called "group". This attribute places the mambot in a sub-directory of the name "group".

There need to be ONE and only one <filename> that have an attribute 'mambot'. This attribute indicates the file that is called when the mambot is loaded. The value of the attribute should be the name of the file less the ".php" extension.

There are two special (or reserved) groups in Mambo:

- *content*: The mambots in this group perform operations on displayed content.
- *search*: The mambots in this group provide for components to plug in the results to search engine.

Any parameters defined are displayed in the Mambot Edit form. The tag is optional. If omitted a simple text box will be provide.

## A Module Setup File

The XML for a module installation could look like this:

```

<?xml version="1.0" encoding="iso-8859-1"?>
<mosinstall type="module" version="4.5.1">
  <name>Main Menu</name>
  <author>Mambo</author>
  <copyright>(C) 2000 - 2004 Miro International Pty Ltd</copyright>
  <license>http://www.gnu.org/copyleft/gpl.html GNU/GPL</license>
  <authorEmail>admin@mamboserver.com</authorEmail>
  <authorUrl>www.mamboserver.com</authorUrl>
  <description>Displays a menu.</description>
  <files>
    <filename module="mod_mainmenu">mod_mainmenu.php</filename>
  </files>
  <params>
    <param name="menutype" type="text" default="" label="Menu Type"
      description="The name of the menu (default mainmenu)" />
    <param name="class_sfx" type="text" default="" label="Class Suffix"
      description="A suffix to be applied to the css class" />
    <param name="menu_style" type="list" default="vert_indent" label="Menu Style"
      description="The menu style">
      <option value="vert_indent">Vertical</option>
      <option value="horiz_flat">Horizontal</option>
    </param>
  </params>
</mosinstall>

```

There need to be ONE and only one <filename> that have an attribute 'module' this attribute is used as a system variable.

Any parameters defined are displayed in the Module Edit form. The tag is optional. If omitted a simple text box will be provide.

For administrator modules you must include a "client" attribute in the mosinstall tag, for example:

```

<mosinstall type="module" client="administrator">

```

## A Component Setup File

A component differ form a module in the way that it it's more complex, and can have extra features to maintain data.

The XML for a component installation could look like this:

```

<?xml version="1.0" ?>
<mosinstall type="component" version="4.5.1">
  <name>My component</name>
  <creationDate>01/08/2004</creationDate>
  <author>Test developer</author>
  <copyright>This component is released under the GNU/GPL License</copyright>
  <authorEmail>info@example.com</authorEmail>
  <authorUrl>www.example.com</authorUrl>
  <version>1.0</version>
  <description>This is a breif description of the component</description>
  <files>
    <filename>mycomponent.php</filename>
    <filename>mycomponent.html.php</filename>
    <filename>images/approve.png</filename>
  </files>
  <install>
    <queries>
      <query id="1"># create a table</query>
      <query id="2"># populate new table</query>
    </queries>
  </install>
</mosinstall>

```

```

</queries>
</install>
<uninstall>
  <queries>
    <query id="1"># delete the table</query>
  </queries>
</uninstall>
<installfile>install.mycomponent.php</installfile>
<uninstallfile>uninstall.mycomponent.php</uninstallfile>
<administration>
  <menu>My component</menu>
  <submenu>
    <menu act="sub1">Sub menu 1</menu>
    <menu act="sub2">Sub menu 2</menu>
  </submenu>
  <files>
    <filename>admin.mycomponent.php</filename>
    <filename>admin.mycomponent.html.php</filename>
    <filename>toolbar.mycomponent.php</filename>
    <filename>toolbar.mycomponent.html.php</filename>
  </files>
  <images>
    <filename>administrator/images/approve.png</filename>
  </images>
</administration>
</mosinstall>

```

As you can see it has more sections, the sections that differ from the common ones is described below.

There is no limit in the number of <filename> entries and they may refer to sub-directories which will be created by the installer.

As the nature of a component gets more complex you can use the <install> section to specify creation of tables and the inserting of default data. In the <install> section you can have <queries>. <queries> is where you can have all the sql statements that your component requires e.g. creation of new tables and adding data to these new tables. There is no limit in the number of <query> sections. Each query must have an id field so that it can be uniquely identified.

NOTE on queries:

If you have HTML in the queries you need to add <![CDATA[ in front and ]]> at the end. For example:

```

<query>
<![CDATA[ INSERT INTO mos_mytable VALUES
  (1, '<P>HTML tags in queries</P><br><a href="http://www.mysite.com">My Site</a>');]]>
</query>

```

The installfile tag is used to specify an additional file, where you can add additional code that will be called at the end of the installation. The file MUST have a com\_install() function in order to work. In the com\_install() function you can have additional functionality to make the component work. The com\_install() function must return a string with a status, which will be displayed to the user at the end of the installation.

The uninstallfile tag is used to specify an additional file, where you can add additional code that will be called at the end of the uninstallation. The file MUST have a com\_uninstall() function in order to work. In the com\_uninstall() function you can have additional functionality to do any cleaning up. The com\_uninstall() function must return a string with a status, which will be displayed to the user at the end of the un-install.

The administrator collectino tag is used to add functionality to the administrator part of Mambo. The <menu> section is used to specify the name of the menu for maintaining the component, it will appear under the

Components menu. If your component needs it, you can have sub menus, use the <submenu> section to add sub menus. The <menu> section of a sub menu have an additional attribute act, which is passed to your module. The act attribute is required for sub menus.

The administrator collectino tag also have <files> and <images>. These sections works the same way as described before, with the difference that files is copied to /administrator/components/[component\_name]/ and images to /administrator/components/[component\_name]/images/.

## A Template Setup File

The XML for a template installation could look like this:

```
<?xml version="1.0" ?>
<mosinstall type="template" version="4.5.1">
  <name>My template</name>
  <creationDate>01/08/2004</creationDate>
  <author>Test Designer</author>
  <copyright>This template is released under the GNU/GPL License</copyright>
  <authorEmail>info@domain.com</authorEmail>
  <authorUrl>www.domain.com</authorUrl>
  <version>1.0</version>
  <description>This is a breif description of the template</description>
  <files>
    <filename>index.php</filename>
    <filename>template_thumbnail.png</filename>
    <filename>css/template_css.css</filename>
    <filename>images/arrow.png</filename>
    <filename>images/mt_business_back.png</filename>
    <filename>images/mt_business_bottom.png</filename>
    <filename>images/mt_business_top.png</filename>
    <filename>images/mt_menu_back.jpg</filename>
  </files>
  <media>
    <filename>self-portrait.jpg</filename>
    <filename>fruit/melon.gif</filename>
  </media>
</mosinstall>
```

Files are copied to /templates/[template\_name]/ including any relative path except for those files in the <media> tag. Files under the <media> will be installed in the /images/stories directory.

For administrator templates you must include a "client" attribute in the mosinstall tag, for example:

```
<mosinstall type="template" client="administrator">
```

# Chapter 8. Access Control

## Overview

Mambo is implementing a powerful access control library that will enable both fine and course grained access control hierarchies to be devised to suit the needs of you site.

## phpGACL

The access control system uses a library called PHPgacl. For more information on the technical points of this library, refer to the site's home page at [phpgacl.sf.net](http://phpgacl.sf.net).

The library has been slightly modified to use the database abstraction layer used in Mambo as opposed to the ADODB library. Some additional functions have also been implemented. However, the format of the files has been honoured as much as possible so that side-by-side comparisons of the modified Mambo files can be made with any future versions of the PHPgacl library.

The schema has been slightly modified so that primary keys are slightly more descriptive (for example, id become aro\_id in the mos\_core\_acl\_aro table).

The PHPgacl has a good tutorial to help you work through the troubles of creating a robust ACL system. We won't regurgitate it all here but we will highlight some terminology and concepts that are relevant for developers'.

## The ACO

An ACO is an Access Control Object. In terms of Mambo it is an action that you want to perform, such a logging in, viewing, adding, editing, etc.

## The ARO

An ARO is an Access Request Object. In terms of Mambo this is "who" or "what" is asking for permission to do something. This will generally be a user, and the mos\_user table is synconises with the mos\_core\_acl\_aro table. However, there may be circumstance where system processes will be requesting permission to do something, possibly in the context of a workflow engine or the like.

ARO's are able to be assigned to a group. The default ARO groups provided in Mambo are:

```
ROOT
| - USERS
| - - Public Frontend
| - - - Registered
| - - - - Author
| - - - - - Editor
| - - - - - - Publisher
| - - Public Backend
| - - - Manager
| - - - - Administrator
| - - - - - Super Administrator
```

The first group is ROOT. This is really a placeholder group as there can only be one root group. The second group, USERS, is also a placeholder group. It collects all the ARO groups that pertain to users. As mentioned previously, other "things" may require access and these would all start with there own placeholder group (for example, WORKFLOW).

Next are the start of two branches, one for access to the frontend web site and one for access to the backend administration.

## The AXO

An AXO is an Access eXtension Object

The ACO, AXO and ACL database schemas are not yet implemented. They are emulated by hand in a simplistic fashion by the gacl class in /classes/gacl.php

## Inserting a New Group

The group mapping for ARO's and AXO's uses a pre-order tree traversal technique to enable more efficient record retrieval from the hierarchially related data. This means that you cannot simply add a row to the table and expect the hierarchial relationships to be maintained.

To add a new user group by hand you would use the following SQL:

```
SET @parent_name = 'Registered';
SET @new_name = 'Support';

-- Select the parent node to insert after
SELECT @ins_id := group_id, @ins_lft := lft, @ins_rgt := rgt
FROM mos_core_acl_aro_groups
WHERE name = @parent_name;

SELECT @new_id := MAX(group_id) + 1 FROM mos_core_acl_aro_groups;

-- Make room for the new node
UPDATE mos_core_acl_aro_groups SET rgt=rgt+2 WHERE rgt>=@ins_rgt;
UPDATE mos_core_acl_aro_groups SET lft=lft+2 WHERE lft>@ins_rgt;

-- Insert the new node
INSERT INTO mos_core_acl_aro_groups (group_id,parent_id,name,lft,rgt)
VALUES (@new_id,@ins_id,@new_name,@ins_rgt,@ins_rgt+1);
```

The parent\_name is the name of an existing group that you want to be the parent of the new group. The new\_name is the name of the new group.

The \_mos\_add\_acl method is also available for custom developers to provide for additional ACL checks.

For more information on pre-order tree traversal algorithms refer to the following sites:

<http://www.sitepoint.com/article/1105/>

[http://searchdatabase.techtarget.com/tip/1,289483,sid13\\_gci537290,00.html](http://searchdatabase.techtarget.com/tip/1,289483,sid13_gci537290,00.html)

# Chapter 9. API Reference

Mambo includes many stand alone utility functions and Object Oriented helper class to reduce the tedium of repetitive task and increase your productivity as a developer.

# Appendix A. Updates for 4.5.1

## Mambots

Mambots are changing format. The format for Mambo 4.5 is still supported but will be deprecated in the next version. See the chapter on Mambots for more information on performance and feature enhancements for event driven Mambots.

A notable addition is that searching is now done via Mambots. This allows any component to craft their own search bot and have it added to the results of the search component.

In this version, Mambots cannot be enabled or disabled with the Administrator. If they are present they will be functioning. To remove the functionality of a Mambot you must delete the file.

## Installer Parameters

You can now format your parameters in modules and components that appear as menu items. XML files are now copied with modules to support this feature.

Two types of parameter tags are currently supported, a textbox and a list box.

Let's use the Main Menu module as an example. The XML file looks like the following:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mosinstall type="module">
  <name>Main Menu</name>
  <author>Mambo</author>
  <copyright>(C) 2000 - 2004 Miro International Pty Ltd</copyright>
  <license>http://www.gnu.org/copyleft/gpl.html GNU/GPL</license>
  <authorEmail>admin@mamboserver.com</authorEmail>
  <authorUrl>www.mamboserver.com</authorUrl>
  <description>Displays a menu.</description>
  <files>
    <filename module="mod_mainmenu">mod_mainmenu.php</filename>
  </files>
  <params>
    <param name="menutype" type="text" default="" label="Menu Type"
      description="The name of the menu (default mainmenu)" />
    <param name="class_sfx" type="text" default="" label="Class Suffix"
      description="A suffix to be applied to the css class" />
    <param name="menu_style" type="list" default="vert_indent" label="Menu Style"
      description="The menu style">
      <option value="vert_indent">Vertical</option>
      <option value="horiz_flat">Horizontal</option>
    </param>
  </params>
</mosinstall>
```

The new params collection tag is highlighted. For each parameter you add a params tag. The following attributes are allowed:

*name*: The name of the parameter

*type*: The type of edit control. Refer to the Appendix on Parameters for a detailed list.

*default*: The default value for the parameter

*label*: The text to place to the left of the edit control

*description*: The text to place to the right of the edit control

Where the parameter type is a list, you may add any number of standard HTML option tags. You must provided a closing option tag!

## Site Templates

### Module containers

A hardcoded `<br/>` tag has been removed from the bottom of each module. To compensate for this, a `bottom-margin` of 15px has been added to the `table.moduletable` css style in the templates distributed with Mambo.

### Pathway Arrows

The pathway will look to see if an `images/arrow.png` file exists in the current template directory. If it finds that this file exists, it will use it as the separator for the pathway. If not, it default to the `>` character.

### CSS

Several new style tags are available: `frontpage`, `frontpageheader` and `blogpageheader`

### Media installer tag

A new `<media>` tag is available for templates setup files. Files list under this tag will be installed in the `/images/stories` directory.

## Modules

See the notes on module parameters above.

Modules written for version 4.5 that use parameters will not work correctly in version 4.5.1. See the appendix on Parameters for more information.

Modules are also supported for Administrator. Modules with an "iscore" field value of 2 or 3 and an access level of 99 are considered Administrator modules.

## API Changes

### Page Navigation

A new method called `getLimitBox` has been added to the `pageNav` class which returns the html for the limit box. The `writeLimitBox` method is still provided for backward compatibility but this method simply returns the return string from `getLimitBox`.

In the Administrator version of the Page Navigation class, there is a new method called `getListFooter`. This method returns an HTML table of the Paging links, the list limit selection box and the Page X – Y of Z results.

The methods `rowNumber`, `orderUpIcon` and `orderDownIcon` have also been added.

## mosHTML

Some extra methods have been added to the mosHTML class to support radio lists.

## Tabs

Following is a skeletal example of how to implement the new style of tabs:

```
<?php
// Construct the object
// Call class, set persistent (useCookies) to no (0) or yes (1)
$tabs = new mosTabs( 0 );

// Start a new pane
// Note, you can have more than one pane on a page (think wrapper)
$tabs->startPane( "module" );

// Start first tab with TabText of Details
// and the 2nd var can be anything UNIQUE
$tabs->startTab("Details", "Details-page");
?>
Put some Content here !!!!!!!!
<?php
// end this first tab
$tabs->endTab();

// Create a new tab!!
// Same vars as before (but different names)
$tabs->startTab("Parameters", "params-page");
?>
More content
<?php
//end this tab
$tabs->endTab();

// end this pane (wrapper)
$tabs->endPane();
?>
```

DON'T FORGET to remove legacy call as follows (normally right at the end of the file.

```
<script language="javascript" type="text/javascript">
    dhtml.cycleTab('tab1');
</script>
```

## Administrator

### Templates and Modules

The template and module support in the Administrator should be considered experimental and may change before the final release of 4.5.1. It is intended that in coming versions the templating engines for both the Site and the Administrator combine. By all means play with the current system and suggest improvements and point out strategic deficiencies. The active template may be changed by directly editing the configuration.php file.

### Banners

The useBanner configuration variable has been dropped from Global Configuration.

## Help Component

The help component has been dropped from the 4.5.1 distribution. A new system based on XML DocBook files has been implemented. Refer to the Documentation Manual for how you add compile help files for your applications.

## Backup to mosDBTable

A new option exists in the Database Backup screen. You may select a mosDBTable as a format. This is useful when designing new components. After you have created your database table, select this function and a skeleton PHP class will be provided for you.

## WYSIWYG Editors

Editors are now Mambots installable in the "editors" group.

## Miscellaneous

TODO

## Future Versions

The MENU\_Default class will be deprecated in the next version.

## Problems with XML Installer Files

DOMIT is more strict with XML parsing than was the MiniXML library. If you are receiving complaints that your package file cannot be install because an installer file cannot be found, it's likely that the file is not well formed. You can quickly test this by opening the XML file in any modern browser.

If you want to have html or other special characters in any field you will need to enclose the contents of the tag CDATA tags, for example:

```
<copyright><![CDATA[ &copy; Copyright Me - 2004 ]]></copyright>
```

# Appendix B. Using Parameters

This appendix describes how to use the parameter system in your Mambo Modules and Components. Parameters provided with components relate to the when they are added to menu items. The parameters will show when you edit the menu item.

## XML Definition

The setup file for Modules and Components setup files define the parameters. Parameters are wrapped in a `<params>` collection tag and then individual parameters are defined in a `<param>` tag. A typical parameter block may look like:

```
<params name="" description="">
  <param name="count" type="text" default="5" label="Number of items"
    description="The number of items to show" />
</params>
```

The following attributes are allowed for the `<params>` tag:

- *name*: The name of the group of parameters parameter
- *description*: A description for the group of parameters

Both these attributes are optional.

The following attributes are allowed for the `<param>` tag:

- *name*: The name of the parameter
- *type*: The type of form element.
- *default*: the default value for the parameter
- *label*: The text to place to the left of the edit control
- *description*: The text to place to the right of the edit control

The available types for parameters are as follows:

### text

The "text" type provides a simple text box, for example:

```
<param name="count" type="text" default="5" label="Number of items"
  description="The number of items to show" />
```

### list

The "list" type provides for an HTML select list. You provide the necessary HTML options as child elements of this tag in the same way as you would for an HTML select list, for example:

```
<param name="hide_author" type="list" default="" label="Hide Author"
  description="Show/Hide the item author - only affects this page">
  <option value="">Use Global</option>
  <option value="1">Hide</option>
  <option value="0">Show</option>
</param>
```

## radio

The "radio" type provides for an HTML radio group. You provide the necessary HTML options as child elements of this tag in the same way as you would for an HTML select list, for example:

```
<param name="show_leading" type="radio" default="1" label="Show Leading"
  description="Show leading items">
  <option value="0">No</option>
  <option value="1">Yes</option>
</param>
```

Radio list groups are formatted horizontally.

## mos\_section

The "mos\_section" type provides a list of published sections, for example:

```
<param name="section_id" type="mos_section" default="0" label="Section"
  description="A content section" />
```

The section id (primary key) is returned.

## mos\_category

The "mos\_category" type provides a list of published categories, for example:

```
<param name="catid" type="mos_category" default="0" label="Category"
  description="A content category" />
```

The lists displays items in "Section–Category" format. The category id (primary key) is returned.

## mos\_menu

The "mos\_menu" type provides a list of the defined menu types used in the site, for example:

```
<param name="menutype" type="mos_menu" default="mainmenu" label="Menu Type"
  description="The name of the menu (default mainmenu)" />
```

## mos\_imagelist

The "mos\_imagelist" type provides a list of images in the directory defined by the "directory" attribute, for example:

```
<param name="menu_image" type="imagelist" directory="/images/M_images" default=""
  label="Menu Image"
  description="A small image to be placed to the left or right of your menu item..." />
```

The directory is relative to the installation directory of Mambo.

## textarea

The "textarea" type provides a simple textarea, for example:

```
<param name="description_text" type="textarea" default="" label="Description Text" rows="30" cols="5"
description="Description for page, if left blank it will load _WEBLINKS_DESC from your language file
```

Note that rows and cols attributes are available for this parameter type.

## The mosParameters Class

A helper class is available for working with parameters called mosParameters.

The mosParameters class constructor takes two arguments, the first the textual parameter values retrieved from the database, and the second in the xml setup file where the parameters are defined. See the following example:

```
global $mainframe;

$option = 'com_content';

// get params definitions
$params =& new mosParameters( $menu->params,
    $mainframe->getPath( 'com_xml', $option ), 'component' );
```

To display the parameters pass the params variable to your display function and simply echo the text returned by the "render" method, for example:

```
<?php
function displayFoo( &$params ) {
    echo $params->render();
}
?>
```

Always pass the params variable by reference as this conserves memory.

## Extending Parameters

NOTE: Experimental

It is possible to add your own form elements by extending the parameters class. For example, to add a form element that lists users you might do something like the following:

```
class myParameters extends mosParameters {
    function _form_userlist() {
        global $database;

        $database->setQuery( "SELECT a.id AS value, a.name AS text"
            . "\nFROM #__users AS a"
            . "\nWHERE a.blocked='0'"
            . "\nORDER BY a.name"
        );
        $options = $database->loadObjectList();
        array_unshift( $options, mosHTML::makeOption( '0', '- Select User -' ) );
        return mosHTML::selectList( $options, "params[$name]", "class=\"inputbox\"",
            'value', 'text', $value );
    }
}

// get params definitions
$params =& new myParameters( $params_text, $xml_file );

echo $params->render();
```

# Using Parameters on the Site

Parameters replaces the bit-wise masking technique used in site components and modules.

The `mosParameters` class has three methods that you will use to access parameters:

```
get( 'name' [, 'default' ] )
```

This methods will return the value of a parameter if it exists or is set, otherwise it returns the 'default' value (or an empty string).

```
set( 'name', 'value' )
```

This method method sets the value of a parameter. It returns the value set.

```
def( 'name', 'default' )
```

This method combines both get and set. It will check to see if the parameter of 'name' exists. If it does it returns it. If it doesn't it sets it to 'default' and returns that value.

Here are some examples:

```
// Parameters
$menu =& new mosMenu( $database );
$menu->load( $Itemid );
$params =& new mosParameters( $menu->params );

$header = $params->get( 'header' );

$count = $params->def( 'count', 10 );

$params->set( 'readon', 1 );
```

A good example of how to use the `def` method is when you want to use a global state by default, for example, a parameter may be defined like:

```
<param name="hide_author" type="list" default="" label="Hide Author"
  description="Show/Hide the item author - only affects this page">
  <option value="">Use Global</option>
  <option value="1">Hide</option>
  <option value="0">Show</option>
</param>
```

Notice that the "Use Global" option is an empty string. You may then have the following code in you module or component:

```
$hide_author = $params->def( 'hide_author', $mosConfig_hideauthor );
```

If the "hide\_author" parameter is not defined, that is, is an empty string or not present at all, then the parameter will be set to the second argument, in this case the global setting for hiding author names, and that value will be returned by the method. If the parameters is not empty, that is, either "0" or "1" then the parameter will not be changed and the actual setting is returned by the method.

## Quick Fix for Old Modules

Modules written for version 4.5 that use parameters will not work correctly in version 4.5.1 or later.

Developers are encouraged to upgrade the method of using parameters as it is very simple to do so. However, to get things running quickly you may insert the following code near the head of you module, before the parameters are used:

```
$params = mosParseParams( $module->params );
```

# Appendix C. mosHTML Reference

## mosHTML Helper Class

mosHTML is a helper class for rendering list. Following are some examples of how to use the methods in this class.

### mosHTML::makeOption

```
makeOption( string $value [, string $text ] )
```

This method returns an object that can be passed in an array to another list handling method. The method takes two arguments, one for the value of the option tag and optionally one for the text to display. If the text is omitted, the single string is used for both the option value and the text.

Example: creating a list with hard coded values:

```
// Option value and text will be the same
$mylist1 = array();
$mylist1[] = mosHTML::makeOption( 'Good' );
$mylist1[] = mosHTML::makeOption( 'Bad' );
$mylist1[] = mosHTML::makeOption( 'Ugly' );

// Option value and text will be the different
$mylist2 = array();
$mylist2[] = mosHTML::makeOption( '0', 'Select Priority' );
$mylist2[] = mosHTML::makeOption( '1', 'Low' );
$mylist2[] = mosHTML::makeOption( '2', 'High' );
```

Example: creating a list from a database query:

```
// alias the 'value' and 'text' fields and the array will
// be in the correct format
$users = array();
$users[] = mosHTML::makeOption( '0', 'Select User' );
$database->setQuery( "SELECT id AS value, username AT text"
    . "\nFROM #__users" );

$users = merge_array( $users, $database->loadObjectList() );
```

The mosHTML method returns a stdClass object with class variables of 'value' and 'text'.

### mosHTML::selectList

```
selectList( array $values, string $tag_name, string $tag_attribs, string
$key, string $text, mixed $selected )
```

The selectList method builds an HTML select tag complete with options. It takes six arguments:

**\$values** – An array of objects that have been returned by a query or the mosHTML method (see above, the \$mylist1 and the \$mylist2 variables would be suitable).

**\$tag\_name** – The name attribute of the select tag.

**\$tag\_attributes** – An additional attributes you want to assign to the select tag.

**\$key** – The name of the class variable holding the option 'value'. Should generally be 'value'.

**\$text** – The name of the class variable holding the option 'text'. Should generally be 'text'.

**\$selected** – Either a string value for a single value select list or an array for a multiple value select list.

Example: a single value select list

```
// Creates a list select list of the users (see previous example)
// The option tag with the value of zero is selected

$html = mosHTML::selectList( $users, 'size="1" class="inputbox"',
    'value', 'text', 0 );

echo $html;
```

Example: a multiple value select list

```
// alias the 'value' and 'text' fields and the array will
// be in the correct format
$users = array();
$users[] = mosHTML::makeOption( '0', 'No User' );
$database->setQuery( "SELECT id AS value, username AT text"
    . "\nFROM #__users" );

$users = merge_array( $users, $database->loadObjectList() );

// Get the selected users from a fictitious table
// We only need the 'value' to lookup the selected options
$database->setQuery( "SELECT id AS value"
    . "\nFROM #__users_selected" );

$selected = $database->loadObjectList();

// Creates the html
$html = mosHTML::selectList( $users, 'size="10" class="inputbox" multiple="true"',
    'value', 'text', $selected );

echo $html;
```

## mosHTML::integerSelectList

```
integerSelectList( $start, $end, $inc, $tag_name, $tag_attribs, $selected,
    $format="" )
```

The `integerSelectList` method builds a list of number from `$start` to `$end` with an increment of `$inc`. The optional `$format` argument allows to apply a printf style format to the number.

Example:

```
$html = mosHTML::integerSelectList( -12, 12, 1, 'tzoffset',
    'size="1" class="inputbox"', 0, "%02d" );

echo $html;
```

## mosHTML::monthSelectList

```
monthSelectList( $tag_name, $tag_attribs, $selected )
```

A convenient method for producing a list of months.

```
$html = mosHTML::monthSelectList( 'month', 'class="inputbox"', '01' );
```

## **mosHTML::treeSelectList**

Todo.

## **mosHTML::yesnoSelectList**

```
yesnoSelectList( $tag_name, $tag_attribs, $selected )
```

A convenient method for producing a simple Yes/No select list.

```
$html = mosHTML::yesnoSelectList( 'show_banners', 'class="inputbox"', 0 );
```

## **mosHTML::radioList**

```
radioList( &$arr, $tag_name, $tag_attribs, $selected=null, $key='value',  
$text='text' )
```

## **mosHTML::yesnoRadioList**

```
yesnoRadioList( $tag_name, $tag_attribs, $selected )
```

A convenient method for producing a simple Yes/No radio button combination.

```
$html = mosHTML::yesnoRadioList( 'show_banners', 'class="inputbox"', 0 );
```

# Appendix D. Code and Commenting Styles

A small subset of the available commenting tags are shown in the following examples. For more details, see the phpDocumentor documentation.

## File Header Comment

All php files must have the following comment block after the opening php tag:

```
/**
 * This is a broad description of the file function
 * @version $ Id $
 * @package Mambo_4.5
 * @copyright (C) 2000 - 2004 Miro International Pty Ltd
 * @license http://www.gnu.org/copyleft/gpl.html GNU/GPL
 * Mambo is Free Software **/
```

This example is used for core files. Replace the description with a brief statement of what the file is for. Where files are stored in a CVS, the \$ Id \$ tag (remove the spaces around the Id text) otherwise use whatever version numbering system you care to employ. Replace the copyright and license with the appropriate details.

The package should be a single word (no spaces) that describes the application this file is a part of. For example, all core file are in the Mambo\_4.6 package. A third party element called Super Forum may be given the package name of SuperForum. When the documentation is compiled by phpDocumentor, it is groups into it's respective packages.

## Documenting Classes

Classes should be commented in the following way:

```
/**
 * A utility class
 */
class mosUtility extends mosAbstractUtility {
    /** @var string A temporary string */
    var $temp;
    /**
     * Constructor
     * @param string A string to store
     * @return boolean True if the string is not empty, false otherwise
     */
    function mosUtility( $temp ) {
        $this->temp = $temp;
        return strlen( $temp ) > 0;
    }
}
```

There are a number of comment blocks used here.

The first is placed before the class definition and should briefly describe what the class is. If the class has a tutorial document related to it, you can link this via the @tutorial tag.

For each class variable (for example, \$temp) provide a comment block on the line before with the @var tag. The syntax is:

@var data-type description

For each class method provide a comment block. The first line should be a brief description. For each argument in the function statement, provide a @param tag and, if the function returns a value, provide a @return tag. Both tags have the same syntax as for the @var tag.

Abstract classes should include the @abstract tag as follows:

```
/**
 * An abstract utility class
 * @tutorial utility.cls
 * @abstract
 */
class mosAbstractUtility {
    // this class does nothing
}
```

## Documenting Functions

Functions are documented in the same way as class methods, for example:

```
/**
 * Strips html tags from a string
 * @param string The source string
 * @return string
 */
function mosStripTags( $text ) {
    return strip_tags( $text );
}
```

## Miscellaneous Documentation

Additional comment throughout the code should be used "just enough" to help you remember what you were doing when you haven't looked at the code for three weeks. Too many comments just bulk the code and too few can reduce the effectiveness of a development team or even your own debugging.

You should include short notes before query statements describing what they are for (it's not always obvious just looking at the sql). You should also use comments to breakup lengthy functions or class methods into logical clumps of code (for example, query data, validate, prepare output, etc).

Always use C-style comments, not Perl style (hash). Here are some examples:

```
// print something out, this is acceptable for a short note
echo $var;

/*
Here is something that needs a little more explanation
over a few lines. This is acceptable.
*/
echo $var->complex();

## Try to avoid this style of comment
echo badPractice();

// -----
// Try to avoid the ascii equivalent of a page break unless they are really necessary
// The bytes add up over time
```

# File Formats

Files must be committed to the CVS in Unix format.

## Code Styles

The Mambo core scripts use a modified Pear standard.

Language control blocks, such as `if`, `while`, `switch`, etc, have a space before the opening bracket but no space around the arguments in the brackets. There is a space between the last bracket and the opening curly brace (this is not on a new line). Subsequent lines are indented with a tab (generally set to the equivalent of 4 spaces in your editor for looks). The closing brace is outdented to align with the opening statement. For example:

```
if ($less < $more) {
    echo 'Here';
}
```

Functions have no space between the name and the opening bracket but have space around the enclosed arguments. This distinguishes them from language constructs. Arguments are separated by a comma–space pair. For example:

```
echo myFunction( $arg1, $arg2 );

if (myCompare( $arg1, $arg2 )) {
    echo 'My compare is true';
}
```

Switch statements should have indented "case" statements followed by indented code including the "break" statement. For example:

```
switch ($task) {
    case 'edit':
        doEditFunction( $option );
        break;

    case 'view':
    default:
        doViewFunction( $option );
        break;
}
```

Classes should naturally indent their methods. For example:

```
class foo {
    function bar( $shah ) {
        return "humbug";
    }
}
```

# Appendix E. Porting MiniXML to the DOMIT Library

Mambo 4.5.1 has upgraded it's XML parser to DOMIT! (<http://www.engageinteractive.com/domit/>). This has been done to solve a conflict in the MiniXML library with PHP Version 4.2.2. DOMIT is also a far more mature library and development is ongoing where other libraries seem to have stagnated.

As a result, the MiniXML libraries have been deleted from the source distribution.

Any component or module that relied on the MiniXML libraries has two choices. Either include your own MiniXML source file, or upgrade to using DOMIT! The later is obviously recommended. It's quite easy. It just takes a few steps to convert a few functions over. Just following the following steps.

## *Finding elements by path*

```
Replace:
$e = &$xml->getElementByPath( 'mosinstall/name' );

With:
$e = &$xml->getElementsByPath( 'name', 1 );
```

## *Getting the contents of a text node*

```
Replace:
$e->getValue();

With:
$e->getText();
```

## *Getting the child nodes array*

```
Replace:
$e->getAllChildren( );

With:
$e->childNodes;
```

## *Loading the XML file*

Replace the following type of block

```
$xmlDoc = new MiniXMLDoc( $dirName . $xmlfile );
$xmlDoc->fromFile( $dirName . $xmlfile );

$element = &$xmlDoc->getElementByPath( 'mosinstall' );
mosDebugVar( $element );

if ( is_null( $element ) ) {
    continue;
}
if ( $element->attribute( "type" ) != "mosbot" ) {
    continue;
}
```

With the following type of code block

```
// Read the file to see if it's a valid MOSBot XML file
$xmlDoc =& new DOMIT_Lite_Document();
```

```
$xmlDoc->resolveErrors( true );  
if (!$xmlDoc->loadXML( $dirName . $xmlfile, false, true )) {  
    continue;  
}  
$element = &$xmlDoc->documentElement;  
  
if ($element->getTagName() != 'mosinstall') {  
    continue;  
}  
if ($element->getAttribute( "type" ) != "mosbot") {  
    continue;  
}  
}
```

### *Getting the root node*

```
Replace:  
$element = &$xmlDoc->getElementByPath( 'mosinstall' );  
  
With:  
$element = &$xmlDoc->documentElement;
```

---

Generated by the free version of [GemDoc](http://www.gemdoc.net). Purchase now at [www.gemdoc.net/purchase](http://www.gemdoc.net/purchase)  
DocBook Made Easy – A single source, Windows based, multiple format solution for your document needs.