Guidelines:Coding_PHP

(Many contents of this guideline has been taken from Moodle Coding Guidelines)

Any collaborative project needs consistency and stability to stay strong.

These coding guidelines are to provide a goal for all code to strive to. It's true that some of the older existing code falls short in a few areas, but it will all be fixed eventually. All new code definitely must adhere to these standards as closely as possible.

Specific rules

- Code submitted to repository CANNOT generate WARNINGS or NOTICES in regular basis. That's it, in standard operation should not generate any WARNING or NOTICE.
- All code files should use the .php extension.
- All template files should use the .html extension unless otherwise defined by a standard (like the translated files).
- All text files should use Unix-style text format (most text editors have this as an option).
- All php tags must be 'full' tags like <?php ?> ... not 'short' tags like <? ?>.
- Please *DO NOT* use shortags expression like <?=$value?> because most PHP renderers “out-of-the-box” doesn't support it, like SLES, and we want to make code running easily on the default plattforms.
- All existing copyright notices must be retained. You can add your own if necessary, remember to add yourself in AUTHORS file.
- Each file should check that the user is authenticated correctly for that function. The header of (almost) every file should look like this:

```
//Copyright notice

require_once ("include/config.php");

check_login ();

if (! give_acl ($config['id_user'], 0, "PM")) {
    audit_db ($config['id_user'], $REMOTE_ADDR, "ACL Violation",
        "Trying to access SNMP Group Management");
    require ("general/noaccess.php");
    exit;
}
```

- Use tabs, not multiple spaces for indents in all files unless impossible. This improves readability.
- Don't create or use global variables in functions except for the standard ones ($config) from index.php. Don't create functions in files. All functions should be defined in include/functions*.php

- Try not to use SQL in files. Unless it's a complicated query, almost all functions necessary have been defined or should be defined in the appropriate include/functions*.php files. Also never use mysql_query or call any of the mysql queries directly. They have been implemented in functions_db.php
- Delete and update functions should not be called from a page with <a href. They should be using a POST form. To make buttons look like an image, you can use CSS.
- All variables should be initialized or at least tested for existence using isset() or empty() before they are used. Array usage should have the same checks with is_array() and empty(). Warnings because inappropriate or non-existing checks are bad code.
- All strings should be translatable - We are now using gettext translatable strings. To display a translatable string do: echo __('String');, to pass it along as a variable, you can do the same ($test = __('Test'); or func_name (__('String'));. You should always use plain, context-correct English. There should be no space between the two underscores and the opening brackets (this is because another script scans the files for that pattern)
- Incoming data from the browser (sent via GET or POST) can be obtained using the function `get_parameter ('key')` or `get_parameter_get ('key')` and`get_parameter_post ('key')` if you it's relevant to fetch it from GET or POST respectively. All other raw data (from files or from databases) must be escaped with `safe_string()` function before inserting it into the database.
- User actions (any INSERT, UPDATE and DELETE) should be logged using the db_audit() function. These logs are used for activity reports and Logs.
- While creating dynamic HTML links, you should avoid the use of standard “&” character substitution (&) to help coders to read the code more easily.
- All other constructions should be made to optimize readability of the code. Eg. don't make lengthy (($statement) ? "string" : "string")) constructions unless there is no other option or it's short and simple enough to be on one line.
- Try to make sure you close all tags especially in your HTML and don't code browser-specific. Although we're currently on HTML Transitional which allows the browser to figure out when something needs closed, it can result in some unexpected output (especially when you're using <form> constructs in <table> and don't close the form or table tags). Try to use double quotes instead of single quotes in HTML constructs. Try to use the HTML function in functions_html.php as much as possible.
- Do not commit work in progress or untested code to SVN. All code committed to SVN should be (as far as you know) working as expected.

Comments style

It's imperative to use a normalized style of commenting functions. We will use in the future the PhpDoc format:

```
/**
 * Short description - used in indexlists
 *
 * Multiple line detailed description.
 * The handling of line breaks and HTML is up to the renderer.
 * Order: short description - detailed description - doc tags.
 *
 * @param int $postid The PHP type is followed by the variable name
 * @param array $scale The PHP type is followed by the variable name
```

```
 * @param array $ratings The PHP type is followed by the variable name
 * @return      array containing output (if present)
 **/
```

Comments should be added as much as is practical, to explain the code flow and the purpose of functions and variables. They are free !!, please USE IT.

- Every function (and class) should use the popular phpDoc format. This allows code documentation to be generated automatically.
- Inline comments should use the /* */ style or //, laid out neatly so that it fits among the code and lines up with it.

Coding style

We know it can be a little annoying to change your style if you're used to something else, but balance that annoyance against the annoyance of all the people trying later on to make sense of code with mixed styles. There are obviously many good points for and against any style that people use, but the current style just is, so please stick to it.

1. Use tabs for indenting. **Do NOT use blankspaces** when indenting your code.

2. Variable names should always be easy-to-read, meaningful lowercase English words (we're working translating old spanish variables to english). If you really need more than one word then put them together using "_" to separate words, but keep them short as possible. Use plural names for arrays of objects. **Do NOT use numbers in the variable name**, numbers do not offer an scope or meaning to the variables.

```
    GOOD: $quiz
    GOOD: $error
    GOOD: $sql
    GOOD: $assignments (for an array of objects)
    GOOD: $i (but only in little loops)
    GOOD: $database_backup
```

```
    BAD: $query1
    BAD: $row1
    BAD: $row2
    BAD: $Quiz
    BAD: $aReallyLongVariableNameWithoutAGoodReason
    BAD: $error_string
    BAD: $databasebackup
```

3. Constants should always be in upper case, and always start with the name of the module. They should have words separated by underscores.

```
define ("FORUM_MODE", 1);
```

4. Function names should be simple English lowercase words, and start with the name of the module to avoid conflicts between modules. Words should be separated by underscores. Parameters should always have sensible defaults if possible. Note there is a space between the function name and the following (brackets) and between operators, and “{” is after first line and closing block in a new line.

```
    function forum_set_display_mode ($mode = 0) {
        global $USER, $CFG;
        if ($mode) {
            $USER->mode = $mode;
        } else if (empty($USER->mode)) {
            $USER->mode = $CFG->forum_displaymode;
        }
    }
```

5. Blocks can be enclosed optionally in curly braces if they have only a line, but they should have braces if they have more than one level of code below or if it improves readability. For example:

```
    if ($quiz->attempts) {
        if ($numattempts > $quiz->attempts)
            error($strtoomanyattempts, "view.php?id=$cm->id");
    }
```

6. Strings should be defined using single quotes where possible, and concatenated with variables (it should become highlighted in an editor). Use double quotes only if it improves readability or any special character (\n \t, etc.) is needed.

```
    $var = 'some text without variables';
    $var = "with special characters like a new line \n";
    $var = 'a very, very long string with a '.$single.' variable in it";
    $var = 'some '.$text.' with '.$many.' variables '.$within.' it';
```

7. Tabs and newlines should be used liberally - don't be afraid to spread things out a little to gain some clarity. Generally, there should be one space between brackets and normal statements, between brackets and variables or functions:

```
    foreach ($objects as $key => $thing) {
        process ($thing);
    }
    if ($x == $y) {
        $a = $b;
    } else if ($x == $z) {
        $a = $c;
    } else {
        $a = $d;
    }
```

8. Please NEVER use inline coding style unless it's real easy. This is very hard to understand for other coders.

VERY BAD:

```
    function test ($a){ if ($a == 1){ echo "Goodbye world"; } else { echo
"Hello world"; }}
    (($test == 1 || $test == 2) ?
                   $blaat = 1;
                   echo "test";
    :
                  $whatever
    )
```

GOOD:

```
    function test ($a) {
        if ($a == 1) {
            echo "Goodbye world";
        } else {
            echo "Hello world";
        }
    }
```

BAD:

```
   (($test == 1) ? $var1 : $var2)
```

9. **Don't create code with more than 5 levels of indentation**. This kind of code is very hard to read/understand for other coders and usually could be written in a better / simple way. Use functions to make repetitive works and **DO NOT CODE USING COPY-PASTE METHODOLOGY**, reuse by creating functions and utilities, not by duplicating code.

9.bis **Don't create BIG functions**. If your functions is more like “a framework” than a simple function, and do a hell of different things with different scenarios and exceptions: SPLIT the function in several which makes simpler things.

10. Occam principle: Simple solution is (almost) always better.

11. Don't use PHP inside HTML. If you are including many HTML, please don't mix with PHP code. If you need to include large HTML data, please use string variables or external file inclusion (even template substitution), for example:

BAD

```
   <div class="datost"><strong><?php echo __('Source') ?> </strong></div>
```

GOOD

```
 echo '<div class="datost"><strong>';
 echo __('Source');
 echo '</b><br><br>';
```

GOOD ALSO

Or even in a single line, like:

```
 echo '<div
class="datost"><strong>'.$lang_label["source_agent"].'</strong></div>';
```

12. SQL queries **should** be created using sprintf() function to make them safer for SQL injections. SQL code syntax for long queries is to separate in many lines, using UPPERCASE in SQL keywords. For example:

```
 $sql = sprintf ('SELECT SUM(tworkunit.duration)
        FROM tworkunit, tworkunit_task
        WHERE tworkunit_task.id_task = %d
        AND tworkunit_task.id_workunit = tworkunit.id', $id_task);
```

13. Try to make fast, specific, indexed SQL queries and use the appropriate functions in functions_db.php. Don't start looping SQL statements if you can use subqueries and don't do SELECT * if you only want a subset of the table:

BAD:

```
$query1 = "SELECT * FROM table1 WHERE username = $uname";
$result = mysql_query ($sql);
while ($row = mysql_fetch_array($result)) {
      $query = "SELECT * FROM table2 WHERE id = $row[id]
}
```

BETTER:

```
$sql = "SELECT id FROM tagents WHERE username = $uname";
$tables = get_db_all_rows_sql ($sql);
foreach ($result as $row) {
       $sql = sprintf ('SELECT message, timestamp FROM table2 WHERE id =
%d', $row[id]);
}
```

BEST:

```
$sql = sprintf ('SELECT message, timestamp FROM table2 WHERE id = ANY(SELECT
id FROM table1 WHERE username = "%s"', $uname);
$result = get_db_all_rows_sql ($sql);
```

OR if you need information from both tables

```
$sql = sprintf ('SELECT table1.message, table1.timestamp, table2.fullname
FROM table1, table2 WHERE table1.id = table2.id AND username = "%s"',
$uname);
```

Database structures

To help you create tables that meet these guidelines.

1. Unless a table contains tuples to link 2 separate tables, every table with information must have an auto-incrementing id field (INT10) as primary index. The id field should not be named id but for a module: module_id, for an event: event_id. See rule 6.

2. The main table containing instances of each module must have the same name as the module (eg widget) and contain the following minimum fields:

- id - as described above
- course - the id of the course that each instance belongs to
- name - the full name of each instance of the module

3. Other tables associated with a module that contain information about 'things' should be named widget_things (note the plural).

4. Table and column names should avoid using reserved words in any database. Please check them before creation.

5. Column names should be always lowercase, in singular and short, following the same rules as for variable names. For some projects (Pandora FMS, Babel) they must begin with lowercase “t” from “table”.

6. Columns referencing primary key of other table should be called “id_tablename”. For example: “id_widget” references table “widget”.

7. Always define a default value for each field (and make it sensible) and specify whether or not it should be unsigned. All id's should be unsigned, all priority and status should be unsigned.

8. Never make database changes in the STABLE branches. If we did, then users upgrading from one stable version to the next would have duplicate changes occurring, which may cause serious errors.

Translation

- If the strings to translate is a constant string, don't worry use directly __("<constant string>"). But it is a string with variable data, like as “There are 400 agents in warning.” you must use sprintf function and don't cut the string in substrings. Because the volunteers translator don't understand a substring. And there are languages that the order of words it is diferent and for example the number of elements is in the end of sentence or in right to element instead in the left element.

VERY BAD:

```
if ($agents_critical >= $agents_warning) {
   $status_text = __(" in critical.");
   $agents_count = $agents_critical;
}
else if ($agents_warning >= $agents_normal) {
```

```
    $status_text = __(" in warning.");
    $agents_count = $agents_warning;
  }
  else {
    $status_text = __(" in normal.") . ">/b<";
    $agents_count = $agents_normal;
  }
  echo "<b>" . __("There are ") . $agents_count . $status_text . "</b>";
```

BAD:

```
  if ($agents_critical >= $agents_warning) {
    echo "<b>" . __("There are ") . $agents_critical . __(" in critical.")
. "</b>";
  }
  else if ($agents_warning >= $agents_normal) {
    echo "<b>" . __("There are ") . $agents_warning . __(" in warning.") .
"</b>";
  }
  else {
    echo "<b>" . __("There are ") . $agents_normal . __(" in normal.") .
"</b>";
  }
```

GOOD:

```
  if ($agents_critical >= $agents_warning) {
    echo "<b>" . __(sprintf("There are %d in critical.", $agents_critical))
. "</b>";
  }
  else if ($agents_warning >= $agents_normal) {
    echo "<b>" . __(sprintf("There are %d in warning.", $agents_warning)) .
"</b>";
  }
  else {
    echo "<b>" . __(sprintf("There are %d in normal.", $agents_normal)) .
"</b>";
  }
```

- Separate content of container. Because if the translation string have html tags, maybe the volunteers translator don't understand it or delete or change the tags in it.

BAD:

```
  echo __("<b>Error in the update</b>");
```

GOOD:

```
  echo "<b>" . __("Error in the update") . "</b>");
```