The Bloxx Coding Standards

Authors:
Telmo Menezes (telmo@cognitiva.net)

Created on:
17 February 2005

Version:
1.1

Document Changelog

Date	Ver	Description	Author
17 Feb 2005	1.0	Document Creation	Telmo Menezes (telmo@cognitiva.net)
23 Feb 2005	1.1	– Curly braces allways below line; – implode() use recommendation droped; – draft's email corrected	Silas Francisco (draft@dog.kicks-ass.net) Tiago Baptista (tiago@baptista.net)

Table of Contents

The Bloxx Coding Standards 1
1.Introduction 2
2.General Recommendations 2
3.Indenting and Line Length 3
4.Control Structures 3
5.Variables 4
6.Strings 4
7.Function/Method Calls 5
8.Function/Method Definitions 5
9.Class Definitions 6
10.Comments 7
11.Including Code 7
12.PHP Code Tags 7
13.Header Comment Blocks 8
14.Using CVS 9
15.Example URLs 10
16.Naming Conventions 10
1General 10
2Classes 11
3Functions and Methods 11
4Constants 11

5Global Variables..... 11
17.File Formats..... 12
18.Sample File..... 12

1.Introduction

This document specifies the PHP coding conventions to be used by Bloxx developers.

The purpose of this document is to define one style of programming in PHP that must be used by all Bloxx developers, in order to keep all the code easy to correct, inspect and maintain by all the developers in the Bloxx team.

In order to reach these goals, the programs should:

Have a consistent style,
Be easy to read and understand,
Be free of common types of errors,
Be maintainable by different programmers.

The Bloxx Coding Standards apply to code that is part of the official Bloxx distribution (Available for download from the Bloxx sourceforge.net project repository).

This collection of rules should be seen as a dynamic document; suggestions for improvements are encouraged. Suggestions can also be made via e-mail to one of the following addresses:

telmo@cognitiva.net
draft@dog.kicks-ass.net

This document is based on the PEAR Coding Standards for PHP (http://pear.php.net/manual/en/standards.php), also borrowing concepts from some industry time-tested C/C++ coding standards adapted to PHP and some other PHP sources when relevant. It contains many similarities but also important differences from the PEAR Coding Standards, so if you are familiar with them please study this document anyway before starting to develop for the Bloxx project.

You are free to use these standards on your own non-Bloxx PHP project if you like.

2.General Recommendations

☑ Optimize code only if you know that you have a performance problem. Think twice before you begin.

☑ Various tests have demonstrated that programmers generally spend a lot of time optimizing code that is never executed. If your program is too slow, determine the exact nature of the problem before beginning to optimize.

☑ Use standard resources.

☑ Use the CVS version control system.

☑ Changes in the code due to bug fixes or enhancements should be marked in the history.txt file on the project root dir, under the current version under development and optionaly nearby the lines of code affected.

☑ Avoid the use of numeric values in code; use symbolic values instead.

3.Indenting and Line Length

☑ Use an indent of 4 spaces, with no tabs.

☑ It is recommended that you break lines at approximately 75-85 characters. There is no standard rule for the best way to break a line, use your judgment.

☑ Branching constructs and looping blocks should be indented.

4.Control Structures

These include if, for, foreach, while, do-while, switch. Here is an example if statement, since it is the most complicated of them:

```
<?php
if ((condition1) || (condition2))
{

    action1;
}
else if ((condition3) && (condition4))
{

    action2;
}
else
{

    defaultaction;
}
?>
```

☑ Control statements should have one space between the control keyword and opening parenthesis, to distinguish them from function calls.

☑ Curly braces must be used even in situations where they are technically optional. Having them increases readability and decreases the likelihood of logic errors being introduced when new lines are added.

☑ The code following a case or a group of adjacent case label must always be terminated by a break statement.

☑ A switch statement must always contain a default branch which handles unexpected

cases.

```
<?php
switch (condition)
{
case 1:
    action1;
    break;

case 2:
    action2;
    break;

default:
    defaultaction;
    break;
}
?>
```

- ☑ The choice of loop construct (for, foreach, while or do-while) should depend on the specific use of the loop.
- ☑ Never use the alternative syntax for control structures. PHP offers an alternative syntax for some of its control structures; namely, if, while, for, foreach, and switch. In each case, the basic form of the alternate syntax is to change the opening brace to a colon (:) and the closing brace to endif;, endwhile;, endfor;, endforeach;, or endswitch;, respectively. Never use this syntax.
- ☑ Always use inclusive lower limits and exclusive upper limits.
- ☑ Avoid the use of continue.
- ☑ Use break to exit a loop if this avoids the use of flags.
- ☑ Use parentheses to clarify the order of evaluation for operators in expressions.
- ☑ Ensure that the values of expressions do not depend on the order of evaluation.

5.Variables

- ☑ Variables are to be declared with the smallest possible scope.
- ☑ Every variable that is declared is to be given a value before it is used.

6.Strings

PHP has two different quotes " and ' which are used to define a text string. Single quote does not process the string or does variable replacement.

If you write "$myVariable\n" and '$myVariable\n' the first will print the contents of the variable followed by a newline character. The latter will print: $myVariable\n, with no aditional processing.

☑ Use single quotes rather than double quotes because this will be quicker.

Below is samples of how you should use single quotes:

```
$myHash['key'];
$str = 'This is a string';
$combinedString = implode( '', array( 'The color of the ball is ', $color, ' and
it costs: ', $price ) );

if ( $variable == 'string' )
{
    ..
}
```

7.Function/Method Calls

☑ Functions should be called with no spaces between the function name, the opening parenthesis, and the first parameter; spaces between commas and each parameter, and no space between the last parameter, the closing parenthesis, and the semicolon.

Here's an example:

```
<?php
$var = foo($bar, $baz, $quux);
?>
```

☑ As displayed above, there should be one space on either side of an equals sign used to assign the return value of a function to a variable. In the case of a block of related assignments, more space may be inserted to promote readability:

```
<?php
$short         = foo($bar);
$long_variable = foo($baz);
?>
```

☑ Minimize recursive function calls.

8.Function/Method Definitions

☑ Function declarations use this format:

```
<?php
function fooFunction($arg1, $arg2 = '')
{
```

```
...
}
?>
```

☑ Arguments with default values go at the end of the argument list. Always attempt to return a meaningful value from a function if one is appropriate. Here is a slightly longer example:

```
<?php
function connect(&$dsn, $persistent = false)
{
    if (is_array($dsn))
    {

        $dsninfo = &$dsn;
    }
    else
    {

        $dsninfo = DB::parseDSN($dsn);
    }

    if (!$dsninfo || !$dsninfo['phptype'])
    {

        return $this->raiseError();
    }

    return true;
}
?>
```

☑ When declaring functions, the leading parenthesis and the first argument (if any) are to be written on the same line as the function name. If space permits, other arguments and the closing parenthesis may also be written on the same line as the function name. Otherwise, each additional argument is to be written on a separate line (with the closing parenthesis directly after the last argument).

☑ Always write the left parenthesis directly after a function name.

☑ Avoid long and complex functions.

☑ Avoid functions with many arguments.

☑ Check the arguments for consistency before everything else in the function's

☑ body.

☑ In functions with a relatively large size, use a single exit point.

9.Class Definitions

☑ The public and private methods of a class are to be declared in that order.

☑ By placing the public section first, everything that is of interest to a user is gathered in the beginning of the class definition. The private section contains details that should have the least general interest.

10.Comments

☑ Inline documentation for classes should follow the PHPDoc convention, similar to Javadoc. More information about PHPDoc can be found here: http://www.phpdoc.org/

☑ Comments should have the same indentation as the object being described.

☑ All comments are to be written in English.

☑ Write a descriptive header before every function.

☑ Comments are often said to be either strategic or tactical. A strategic comment describes what a function or section of code is intended to do, and is placed before this code. A tactical comment describes what a single line of code is intended to do, and is placed, if possible, at the end of this line. Unfortunately, too many tactical comments can make code unreadable. For this reason, it is recommended to primarily use strategic comments, unless trying to explain very complicated code. If the characters // are consistently used for writing comments, then the combination /* */ may be used to make comments out of entire sections of code during the development and debugging phases.

☑ Perl/shell style comments (#) are no te be used.

11.Including Code

☑ Anywhere you are unconditionally including a class file, use require_once().

☑ Anywhere you are conditionally including a class file (for example, factory methods), use include_once().

☑ If you are including a Bloxx Module class, allways use the include_module_once() Bloxx global function.

Either of these will ensure that class files are included only once. They share the same file list, so you don't need to worry about mixing them - a file included with require_once() will not be included again by include_once(). The same is true for include_module_once().

12.PHP Code Tags

☑ Always use `<?php ?>` to delimit PHP code, not the `<? ?>` shorthand. This is the most portable way to include PHP code on differing operating systems and setups.

13.Header Comment Blocks

☑ All source code files in the core Bloxx distribution should contain the following comment block as the header:

```
<?php
// Bloxx - Open Source Content Management System
//
// Copyright (c) 2002 - 2005 The Bloxx Team. All rights reserved.
//
// Bloxx is free software; you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation; either version 2 of the License, or
// (at your option) any later version.
//
// Bloxx is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.

// You should have received a copy of the GNU General Public License
// along with Bloxx; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
//
// Authors: Original Author <author@example.com>
//          Your Name <you@example.com>
//
// $Id:$
?>
```

☑ There's no hard rule to determine when a new code contributor should be added to the list of authors for a given source file. In general, their changes should fall into the "substantial" category (meaning somewhere around 10% to 20% of code changes). Exceptions could be made for rewriting functions or contributing new logic.

☑ Simple code reorganization or bug fixes would not justify the addition of a new individual to the list of authors.

☑ Files not in the core Bloxx repository should have a similar block stating the copyright, the license, and the authors. All files should include the modeline comments to encourage consistency.

☑ All class declarations should be preceded by a comment block with the following format (PHPDoc):

```
/**
  * Class purpose and description
  *
  * @package   Bloxx_Core
  * @version   $Id:$
  * @category  Core/Module/Auxiliar
  * @copyright Copyright &copy; 2002-2005 The Bloxx Team
  * @license   The GNU General Public License, Version 2
  * @author    First Author <author1@example.com>
```

```
  * @author     Second Author <author2@example.com>
  */
```

☑ All function/method declarations should be preceded by a comment block with the following format (PHPDoc):

```
/**
  * Function/Method purpose and description
  *
  * @param resource $result    query result identifier
  * @param array    $arr       (reference) array where data from the row
  *                            should be placed
  * @param int      $fetchmode how the resulting array should be indexed
  * @param int      $rownum    the row number to fetch
  *
  * @return mixed DB_OK on success, null when end of result set is
  *               reached or on failure
  *
  * @see DB_result::fetchInto()
  * @access private
  */
```

14. Using CVS

☑ Include the Id CVS keyword in each file. As each file is edited, add this tag if it's not yet present (or replace existing forms such as "Last Modified:", etc.).

The rest of this section assumes that you have basic knowledge about CVS tags and branches.

☑ CVS tags are used to label which revisions of the files in your package belong to a given release.

Below is a list of the required and suggested CVS tags:

RELEASE_*n_n_n*
(required) Used for tagging a release. If you don't use it, there's no way to go back and retrieve your package from the CVS server in the state it was in at the time of the release.

QA_*n_n_n*
(branch, optional) If you feel you need to roll out a release candidate before releasing, it's a good idea to make a branch for it so you can isolate the release and apply only those critical fixes before the actual release. Meanwhile, normal development may continue on the main trunk.

MAINT_*n_n_n*
(branch, optional) If you need to make "micro-releases" (for example 1.2.1 and so on after 1.2.0), you can use a branch for that too, if your main trunk is very active and you want only minor changes between your micro-releases.

Only the RELEASE tag is required, the rest are recommended for your convenience.
Below is an example of how to tag the 1.2.0 release of the **Money_Fast** package:

```
$ cd bloxx/Money_Fast
$ cvs tag RELEASE_1_2_0
T Fast.php
T README
T package.xml
```

Here's an example of how to create a QA branch:

```
$ cvs tag QA_2_0_0_BP
...
$ cvs rtag -b -r QA_2_0_0_BP QA_2_0_0
$ cvs update -r QA_2_0_0
$ cvs tag RELEASE_2_0_0RC1
...and then the actual release, from the same branch:
$ cvs tag RELEASE_2_0_0
```

The `QA_2_0_0_BP` tag is a "branch point" tag, which is the start point of the tag. It's always a good idea to start a CVS branch from such branch points. MAINT branches may use the RELEASE tag as their branch point.

15.Example URLs

☑ Use `example.com`, `example.org` and `example.net` for all example URLs and email addresses, per RFC 2606.

16.Naming Conventions

1 General

☑ Do not use names that differ only by the use of uppercase and lowercase letters.

☑ Names should not include abbreviations that are not generally accepted.

Example of ambiguous names:

```
function termMessage(); // Terminate message or terminal message?
```

☑ Choose variable names that suggest the usage.

One rule of thumb is that a name, which cannot be pronounced, is a bad name. A long name is normally better than a short, cryptic name, but the truncation problem must be taken into consideration. Abbreviations can always be misunderstood.

Example: Choice of names

```
var $groupID; // instead of var $grpID
var $nameLength; // instead of $namLn
```

2 Classes

☑ Classes should be given descriptive names. Avoid using abbreviations where possible. Class names should always begin with Bloxx_ followed by an uppercase letter.

Examples of correct class names are:

```
Bloxx_Module
Bloxx_PollVote
Bloxx_DBObject
```

3 Functions and Methods

☑ Functions and methods should be named using the "studly caps" style (also referred to as "bumpy case" or "camel caps"). The initial letter of the name (after the prefix) is lowercase, and each letter that starts a new "word" is capitalized.

Some examples:

```
connect()
getData()
buildSomeWidget()
```

☑ Private class members (meaning class members that are intented to be used only from within the same class in which they are declared are preceded by a single underscore.

For example:

_sort()
_initTree()

4 Constants

☑ Constants should always be all-uppercase, with underscores to separate words.

Note: The `true`, `false` and `null` constants are excepted from the all-uppercase rule, and must always be lowercase.

5 Global Variables

☑ If your package needs to define global variables, their name should start with G_ and be all upercase.

☑ Global variables should be avoided as much as possible.

17.File Formats

☑ All scripts files contributed to Bloxx must be stored as ASCII text.

☑ All scripts files contributed to Bloxx must use ISO-8859-1 character encoding.

☑ All scripts files contributed to Bloxx must be Unix formatted.

"Unix formatted" means two things:

1) Lines must end only with a line feed (`LF`). Line feeds are represented as ordinal `10`, octal `012` and hex `0A`. Do not use carriage returns (`CR`) like Macintosh computers do or the carriage return/line feed combination (`CRLF`) like Windows computers do.
2) There should be *one* line feed after the closing PHP tag (`?>`). This means that when the cursor is at the very end of the file, it should be *one* line below the closing PHP tag.

☑ Script files must allways use the ".php" extension.

☑ If the script file contains a class definition, the file name should be the same as the class name but all lowercase. Ex: class Bloxx_ModuleManager on file bloxx_modulemanager.php.

☑ Script files should contain a single class definition.

18.Sample File

TBD!!!