Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Prof. Dr. W.-P. de Roever

Jan Waller - PHP Gruppe 1

Inhaltsverzeichnis

1 Coding style

This section is based on an article by *Tim Perdue* on *www.phpbuilder.com*. I didn't ask for his permission, but most of this is just copy pasted... Some Additions done by me...

1.1 Indenting

All your code should be properly indented. This is the most fundamental thing you can do to improve readability. Even if you don't comment your code, indenting will be a big help to anyone who has to read your code after you.

```
while ($x < $z) {
  if ($a == 1) {
    echo 'A was equal to 1';
  } else {
    if ($b == 2) {
      //do something
    } else {
      //do something else
    }
  }
}
```

Use only spaces for indenting code, never tabs. Tab display width is not standardized enough, and anyway it's easier to manually adjust indentation that uses spaces.
As a standard I propose a width of 2.
Stay within 80 columns, the width of a minimal standard display window.

1.2 Braces

If you use conditional expressions (`IF` statements) without braces, not only is it less readable, but bugs can also be introduced when someone modifies your code.

Bad Example:

```
if ($a == 1) echo 'A was equal to 1';
```

That's pretty much illegible. It may work for you, but the person following after you won't appreciate it at all.

Less Bad Example:

```
if ($a == 1)
  echo 'A was equal to 1';
```

Now that's at least readable, but it's still not maintainable. What if I want an additional action to occur when `$a==1`? I need to add braces, and if I forget to, I'll have a bug in my code.

Correct:

```
if (($a == 1) && ($b==2)) {
  echo 'A was equal to 1';
  //easily add more code
} elseif (($a == 1) && ($b==3)) {
  //do something
}
```

Notice the space after the `if` and `elseif` - this helps distinguish conditionals from function calls.

1.3 Function Calls

Functions should be called with no space between between the function name and the parentheses. Spaces between params and operators are encouraged.

```
$var = myFunction($x, $y);
```

1.4 Functions

Function calls are braced differently than conditional expressions. This is a case where I'll have to change my personal coding style in order to be kosher, but I guess it needs to be done.

```
function myFunction($var1, $var2 = '')
{
  //indent all code inside here
  return $result;
}
```

Notice again there is no space between the function name and the parens, and that the params are nicely spaced. All your code inside the function will be at least 4 spaces indented.
Another important principle when coding functions is that they should always return instead of print directly. Remember, if you print directly inside your function call, whomever calls your function cannot capture its output using a variable.

1.5 Comments

Borrowing - almost in its entirety - from the `JavaDoc` spec, I strongly encourages the use of `PHPDoc` comment style. JavaDoc is rather clever because you format your code comments in such a manner that they can be parsed by a doc-generating tool, essentially creating *self commenting* code. You can then view the resulting docs using a web browser.

```
/**
 *  short description of function
 *
 *  Optional more detailed description.
 *
 *  @param $paramName - type - brief purpose
 *  @param ...
 *  ...
 *  @return type and description
 */
```

1.6 Including Code

PHP4 introduced a pretty useful new feature: `include_once()`. I've seen a lot of questions on the mailing lists and discussion boards caused by people including the same files in multiple places. This can cause conflicts when the included files include functions, which can be defined only once per script.
The simple solution is to replace all your `include()` and `require()` calls with corresponding `include_once()` and `require_once()`. Both use the same file list, so a file included with `require_once()` will not be later included with an `include_once()` call. (technically, `require_once` and `include_once` are *operators*, not *functions* so parens are not necessary).
I never use `include`, or any of these `*_once` functions. IMHO, well-designed applications include files in one place that is easy to maintain and keep track of. Others will disagree and say that each included file should `require_once()` every file that it depends on. This seems like extra overhead and maintenance headaches to me personally. Either way, the days of conflicting includes are probably over.

1.7 PHP Tags

You should always use the full `<?php ?>` tags to enclose your code, not the abbreviated `<? ?>` tags, which could conflict with XML or other languages. ASP-style tags, while supported, are messy and discouraged.

1.8 Strings

In PHP of course, double quotes (") are parsed, but single quotes (') are not. That means PHP does extra work (magic really) on double-quoted strings that it doesn't do on single-quoted strings. So there is a (subtle) performance difference between the two, especially in code that is iterated or called a large number of times.

Best:

```
$associative_array['name'];
$var='My␣String';
$var2='Very...␣long...␣string...␣' . $var . '␣...more␣string...␣';
$sql="INSERT␣INTO␣mytable␣(field)␣VALUES␣('$var')";
```

Acceptable:

```
$associative_array["name_$x"];
$var="My␣String␣$a";
```

The first example does not include any `vars` that need to be appended or parsed into the strings, so single quotes (') are definitely the best way to go. The second example includes very short strings and has variables that need to be parsed into the strings, so it is acceptable to use double quotes. If the strings were long, it would be best to use the append (.) operator. For consistency's sake, I always use single quotes around all my strings, except SQL statements, which almost always contain apostrophes and variables that must be parsed into the strings.

2 Writing log messages

This section is again almost entirely copied (this time my source is the `papidSVN` Hacking Guide...

Certain guidelines should be adhered to when writing log messages:

- Make a log message for every change. The value of the log becomes much less if developers cannot rely on its completeness. Even if you've only changed comments, write a log that says *Doc fix.* or something.
- Use full sentences, not sentence fragments. Fragments are more often ambiguous, and it takes only a few more seconds to write out what you mean. Fragments like *Doc fix*, *New file*, or *New function* are acceptable because they are standard idioms, and all further details should appear in the source code.
- The log message should name every affected function, variable, macro, makefile target, grammar rule, etc, including the names of symbols that are being removed in this commit. This helps people searching through the logs later. Don't hide names in wildcards, because the globbed portion may be what someone searches for later. For example, this is bad:

```
* twirl.cpp
  (twirling_baton_*): Removed these obsolete structures.
```

```
    (handle_parser_warning): Pass data directly to callees, instead
    of storing in twirling_baton_*.

  * twirl.h: Fix indentation.
```

Later on, when someone is trying to figure out what happened to 'twirling_baton_fast', they may not find it if they just search for "_fast". A better entry would be:

```
  * twirl.cpp
    (twirling_baton_fast, twirling_baton_slow): Removed these
    obsolete structures.
    (handle_parser_warning): Pass data directly to callees, instead
    of storing in twirling_baton_*.

  * twirl.h: Fix indentation.
```

The wildcard is okay in the description for 'handle_parser_warning', but only because the two structures were mentioned by full name elsewhere in the log entry.

Note how each file gets its own entry, and the changes within a file are grouped by symbol, with the symbols are listed in parentheses followed by a colon, followed by text describing the change. Please adhere to this format – not only does consistency aid readability, it also allows software to colorize log entries automatically.

- If your change is related to a specific issue in the issue tracker, then include a string like ïssue #Nïn the log message. For example, if a patch resolves issue 1729, then the log message might be:

```
Fix issue #1729:

* get_editor.cpp
  (frobnicate_file): Check that file exists first.
```

- For large changes or change groups, group the log entry into paragraphs separated by blank lines. Each paragraph should be a set of changes that accomplishes a single goal, and each group should start with a sentence or two summarizing the change. Truly independent changes should be made in separate commits, of course.

- One should never need the log entries to understand the current code. If you find yourself writing a significant explanation in the log, you should consider carefully whether your text doesn't actually belong in a comment, alongside the code it explains. Here's an example of doing it right:

```
(consume_count): If 'count' is unreasonable, return 0 and don't
advance input pointer.
```

And then, in 'consume_count' in 'cplus-dem.cpp':

```
while (isdigit ((unsigned char)**type))
{
  count *= 10;
```

```
count += **type - '0';
/* A sanity check.  Otherwise a symbol like
  '_Utf390_1__1_9223372036854775807__9223372036854775'
  can cause this function to return a negative value.
  In this case we just consume until the end of the string.  */
if (count > strlen (*type))
{
  *type = save;
  return 0;
}
```

This is why a new function, for example, needs only a log entry saying *New Function* — all the details should be in the source.

- There are some common-sense exceptions to the need to name everything that was changed:
  - If you have made a change which requires trivial changes throughout the rest of the program (e.g., renaming a variable), you needn't name all the functions affected, you can just say *All callers changed*.
  - If you have rewritten a file completely, the reader understands that everything in it has changed, so your log entry may simply give the file name, and say Rewritten".
  - If your change was only to one file, or was the same change to multiple files, then there's no need to list their paths in the log message (because ßvn logcan show the changed paths for that revision anyway). Only when you need to describe how the change affected different areas in different ways is it necessary to organize the log message by paths and symbols, as in the examples above.
- In general, there is a tension between making entries easy to find by searching for identifiers, and wasting time or producing unreadable entries by being exhaustive. Use your best judgment — and be considerate of your fellow developers. (Also, run ßvn logto see how others have been writing their log entries.)

3 SVN specific stuff

- If you add any files to the project that might be opened on more than one operating system then give the file the native eol-style property. This ensures that all files downloaded via svn arrive with the line endings native to that OS. Here is how you add the property:
  `% svn propset eol-style native new_file.php`
- If you add images to the project make sure you apply the correct mime-type property. Example for a XPM:
  `% svn propset svn:mime-type image/x-xpm`