Xaraya
Request for Comments: 0044
Category: Best Current Practice

M.R. van der Boom
F.B. Besler
Xaraya Development Group
March 2003

Xaraya Coding standards

Status of this Memo

This document specifies a Xaraya Best Current Practices for the Xaraya Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited.

Copyright Notice

Abstract

This RFC lists all the coding standards applicable for the Xaraya project.

Table of Contents

1. Introduction

In this RFC we pull together the general PEAR coding standard[1] and the specific Xaraya standards into one document.

The RFC does not only contain coding standards as perceived in the normal way, but does also touch on standards for creating templates, themes, modules and general rules of engagement etc., which are not perceived as coding activities by some people. We think that any activity which touches the repository or influences the use of Xaraya needs to be considered as a coding activity.

[1] http://pear.php.net/manual/en/standards.php

2. BlockLayout Templating

See the BlockLayout Specification (RFC 10[2]) for detailed information.

This section focussed on general rules to follow when writing templates for the blocklayout engine for use in Xaraya.

No output generation in code

Any output generated, whether this is HTML, XML or any other output format used to render a page, block or other part of the output, goes through a template. Embedded output in code prevents overriding the output in a theme and is as such 'hard-coded'. For some areas of Xaraya it is difficult to comply with this at the moment (hooks, exceptions, xmlrpc), but eventually this will all go through a template.

For certain parts the obligation for output to go through templates generates additional complexity in code and to Xaraya in general. This is the price to be paid for a completely templated system. The additional flexibility during the deployment at (potentially lots of) user sites warrants this.

Minimize duplication

Try to use the <xar:template /> tag to organize the templates you are working on. If some "common" part of the output exists (like headers, footers, forms and menus for example), isolate those templates in a separate file and use <xar:template /> to pull them into the templates where the output is needed.

Minimize PHP usage

While possible, PHP usage in templates is highly discouraged and can often be solved in a templatable way. If a solution cannot be found, we will first look whether the problem can be genaralized, so we can adapt blocklayout to cater for the general problem and solve it like that. It is our expectation that we can catch 95% of the situations this way.

Keep in mind that PHP usage in blocklayout templates might be dissallowed in the future, as it leads to all kinds of problems (security related mostly) which are not easy to solve if PHP usage is allowed. There is enough knowledge in our group to help out on solving the issues which involve the use of PHP in templates.

Note: the PHP usage in certain blocklayout attributes is another matter. These are inherilty built into blocklayout (for example the condition attribute in the logic tags). The fact that these attributes are PHP expressions is merely an easy way to have expressions available, using the language used to write Xaraya. (We could have used another language in those attributes to express the conditions).

Templates must be self-containable.

```
            Wrong:
              <tr>
                <td>DATA</td>
              </tr>
```

```
            Better:
              <table>
                <tr>
                  <td>DATA</td>
                </tr>
              </table>
```

This rule is 'fuzzy' at least. The 'right' example is also not completely containable. (can't appear inside 'head' for example) The actual rule is dependent on the output format (for clear text, there is no such thing at all) and how that output format defines it's context rules. A more loose definition could be: "Maximize the scope of

[2] http://www.xaraya.com/documentation/rfcs/rfc0010.html

containability". The first example can *only* be embedded inside a template with a <table> tag surrounding it at its ancestor level, while the second example can be contained in any place where a block level element is valid.

Another way of saying the above would be: "Try to minimize the requirements on the container template within the context of the output domain".

Yet another way of saying it is: "Use common sense" (which is by the way the overall rule which applies to everything said in this RFC)

3. Headers

This section contains the standard headers we use

Template Header Recommendation

All Templates should include the name of the license and a link to it so as to avoid legal ambiguity. You may also wish to add the author's name as well.

```
            <xar:comment>License: GPL
http://www.gnu.org/copyleft/gpl.html</xar:comment>
```

Note: Use the <xar:comment> </xar:comment> form over the <!-- --> (or deprecated <!--- --->) form so that it isn't added to the output stream (BL will not output it)

File header

This header is for all php files in the codebases. The package name and copyright year (or a list of years) should be updated as necessary.

```
/**
* Short description of purpose of file
*
* @package unassigned
* @copyright (C) 2002-2005 by The Digital Development Foundation
* @license GPL {@link http://www.gnu.org/licenses/gpl.html}
* @link http://www.xaraya.com
*
* @subpackage module name
* @link  link to information for the subpackage
* @author author name <author@email> (responsible person)
*/
```

Make sure that the fileheader is immediately followed by another docblock, otherwise you'll get errors and the docblock is interpreted as the documentation for the first php code encountered in the file.

Function and class method header

The header below is used for all functions in the codebase including methods in class defintions:

```
/**
* Short description of the function
*
* A somewhat longer description of the function which may be
* multiple lines, can contain examples.
*
* @author  Author Name <author@email>
* @deprec  date since deprecated <(only if function is deprecated)>
* @access  public / private / protected
* @param   type param1 Description of parameter 1
* @param   type param2 Description of parameter 2
* @return  type to return description
* @throws  list of exception identifiers which can be thrown
* @todo    <devname> <#> todolist for module (RFC, wish or otherwise)
*/
```

Class header

The header below is used for all classes in the Xaraya codebase

```
/**
* Short class description
*
* A somewhat longer description of the class which
* may be on multiple lines and can contain examples
```

```
*
* @package unassigned - replace with packagename
* @author Author Name <author@email>
* @deprec date since deprecated <insert if class is deprecated>
* @todo  todo-item, specify each with new tag
*/
```

4. Xaraya coding style

This section deals with standards for coding php files in Xaraya

4.1 General

Use <?php ... ?> instead of <? ... ?>

Use the long form of identifying the start of a php section in a file. We follow the PEAR standard in this; it is also the most portable way to include PHP code on differing operating systems and setups.

Note:

Short tags will also cause php to choke on <?xml ?> tags if the short notation is allowed in the php setup.

Indenting

Use an indent of 4 spaces. Do not use the tab character for indentation. Most editors will have a configuration option for this.

Indentation is intended to convey structural meaning, not just to line up items on adjacent lines. Structural indentation should take priority over visual layout indentation. For example, this indentation style is just visual; the indentation happens to reflect the length of a function names, and nothing of the structure:

```
list($p1,
     $p2) = xarVarCleanFromInput('name',
                                 'module');
```

A better style would convey more meaning of the structure. Examples could include the following, depending on the numbers and sizes of the tokens. Note that braces and brackets are closed either on the same indentation level they are opened, or on the same line they are opened:

```
list($p1, $p2) = xarVarCleanFromInput('name', 'module');

list($p1, $p2) =
    xarVarCleanFromInput('name', 'module');

list(
    $p1, $p2, $p3, $p4
) = xarVarCleanFromInput(
    'name',
    'module',
    'a_really_long_token_indeed',
    'something_shorter'
);
```

File and line endings

Use unix file and line endings, not dos/windows ones. Most editors and IDEs on Windows will provide a choice to save a file as unix file. To some extent Monotone will provide the necessary conversions if necessary.

Comments

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

Non-documentation comments are strongly encouraged. A general rule of thumb is that if you look at a section of code and think "Wow, I don't want to try and describe that", you need to comment it before you forget how it works.

C style comments (/* */) and standard C++ comments (//) are both fine. Use of Perl/shell style comments (#) is discouraged.

In order to maintain consistency across the code base all comments should be in English where possible - feel free to ask for help with translating if you need it.

Including Code

The inclusion of files should in most cases be done with the API function:

```
              xarInclude($filename, $flags);
```

This function is part of the Xaraya API and tries hard to safely include certain files. For "normal includes" this function should be used.

If you cannot use xarInclude() for some reason, the normal php functions can also be used, but you are responsible for handling the exceptions or any other errors caused by the direct inclusion.

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()*. 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()*.

Note: include_once() and *require_once()* are statements, not functions. You don't need parentheses around the filename to be included.

4.2 Functions

Calling

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);
            ?>
```

Declarations

Function declarations follow the "one true brace" convention:

```
            <?php
            function fooFunction($arg1, $arg2 = '')
            {
                if (condition) {
                    statement;
                }
                return $val;
            }
            ?>
```

Arguments

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;
            }
            ?>
```

Naming

The following naming conventions are used in general. The module functions are required to have certain specific names if they are used in API parts or interface parts of that function. The specific rules for these are described in the "Xaraya Module Developers Guide" (need to have a permalink for this)

1. public functions start with "xar"
2. protected functions start with "xarFoo_" (one underscore)
3. private functions start with "xarFoo__" (two underscores)

Functions should have English based names where at all possible to maintain consistency across the code base.

Note that php (version 4 or lower) has no notion of 'public', 'protected', 'private' or 'abstract' access specifiers, i.e. php will NOT enforce that a private function is not called from a callee outside the 'private' scope.

4.3 Classes

Declarations

Class declarations follow the "one true brace" convention:

```
            <?php
            class fooClass extends barClass
            {
                function fooClass()
                {
                    statements;
                }
            }
            ?>
```

4.4 Variables

Consistant variable naming eases the readability of code. We recommend (fill in the blank)

Variables should have English based names where at all possible to maintain consistency across the code base.

Example:

```
            Example array key and variable usage
```

4.5 Control Structures

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

```
<?php
if ((condition1) || (condition2)) {
    action1;
} elseif ((condition3) && (condition4)) {
    action2;
} else {
    defaultaction;
}
?>
```

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

You are strongly encouraged to always use curly braces 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. For switch statements:

```
<?php
switch (condition) {
    case 1:
        action1;
        break;
    case 2:
        action2;
        break;
    default:
        defaultaction;
}
?>
```

5. Helper docs

5.1 Scenario status report

When working in a scenario it is necessary to give some status back to the group every now and then. A biweekly or monthly status report is the suggested schedule. Use the template below for submitting status reports.

```
Scenario status report
======================

1. Scenario
-----------
<Try to use the name as in roadmap>

2. Developers change
--------------------
2.1 Developer name (both Real name and irc nick)
2.2 Developer emailaddress
2.3 Added/Removed/Away till:__/__/____/

3. Status changes
-----------------
Date of effect:
Percentage complete:
Duration:
Dependency:

4. Remarks
----------
<Anything missed above >

Mail this template:
- to: xaraya_pmc@xaraya.com
- cc: xaraya_committers@xaraya.com
You will get notification when the roadmap has been
updated with the info, so you can check if we've
done it right.
```

5.2 Scenario assignment

When starting a new scenario (locally, in a group) you can use the template below to register it. This will trigger the following actions:

1. check in bugzilla if the appropriate components are available
2. allocate a repository area on the server, if applicable
3. creating the right accounts, if necessary
4. ...

```
Scenario assigment template
===========================

1. Scenario
-----------
<This name will be used in the roadmap>

2. Names of developers working in the scenario
----------------------------------------------
2.1 Developer name (both Real name and irc nick)
2.2 Developer emailaddress
<only team lead is necessary, rest is optional>

3. Scenario data
----------------
3.1 Description
<Give a short paragraph on the what and why>
```

```
3.2 start date
3.3 estimated amount of time needed in total
3.4 percentage complete
3.5 dependencies
(as in: which scenarios do you need to be completed
first before you can finish)

4. Infrastructure
-----------------
4.1 Server scenario needed: yes/no
4.2 Web interface needed: yes/no
4.3 Triggers needed: yes/no
>Specify if you have special wishes>

5. Remarks
----------
<Anything missed above >

Mail this template to: xaraya_pmc@xaraya.com
You will get notification whether we can include the scenario and
where the updated roadmap is available.
```

6. Version numbers

Typically in software development, you deal a lot with version numbers. Below are the rules for any version number used in Xaraya.

6.1 Format

1. All version numbers have exactly 3 positions. (1.2.3)
2. Each position is numeric
3. First position denotes major revision: Very significant additions/changes in design, architecture, or functionality, up to and including complete rewrite from the previous major version. For new works, a Major digit of 0 is suggested until the work reaches a feature-complete state as envisioned by the author(s).
4. Second position denotes minor revision: A new feature is added, a dependency is added/changed/removed, or a change to database structure is implemented. Regardless of how small the difference may be between successive versions, if dependencies or database structure change, a change in the minor digit is required. When the criteria for a Minor digit increase overlap (ie, a new feature includes a database change) the overlapping criteria are counted as a single change for the purposes of version number change.
5. Third position denotes bugfixes: A single bug is fixed, even if the bug is previously unknown. In that case, the bug should still be documented in the appropriate tracker, even posthumously.
6. No leading zero's are used (i.e. 1.01.04 is invalid, this should be 1.1.4)
7. Version numbering in one position does not overflow and is not limited to single digits; 345.300.14 is a valid version number, as is 1.10.11
8. The first two positions are 'active' positions. This means that based on the change of any of the first or second position an action could be triggered. Typically in modules this involves a forced upgrade or a reinitialisation. The third position is a 'passive' location, i.e. Xaraya just registers the version number and may perhaps do an automatic upgrade. The responsible developer should make sure that his object, module, theme or otherwise deals with this correctly.
9. Version suffixes (ie, RC2, b1, etc) are not significant to the numbering scheme and should be ignored by code that handles version numbers (except for display purposes). They may be used to indicate milestones of stability and/or feature completion while approaching a Major (x.0.0) version. If used, these indicators should be used in the following order: a[1-9] (alpha), b[1-9] (beta), RC[1-9] (release candidate). Version suffixes should be separated from the last numeric positon by a hyphen. Version suffixes should not affect the numeric digits in any way, or vice versa. The use of version suffixes is left to each author's preference, and is neither encouraged nor discouraged.

6.2 Version Number Changes

As development continues, version numbers change to reflect progress. Using the rules below, two version numbers can indicate at a glance the amount of change between the two versions.

1. Version numbers never decrease.
2. When multiple features or bugfixes are contained in a single revision, the corresponding posiiton is increased accordingly (ie, 1.0.0 + 2 bugfixes = 1.0.2)
3. When a position changes, all lower positions are reset to 0 (ie, 1.2.7 + new feature = 1.3.0, or 1.2.7 + major functionality = 2.0.0).
4. When a revision would affect more than one position, only the leftmost posiiton is changed, and the lower ones are reset to 0. For example, 1.2.7 + 2 new features and 3 bugfixes = 1.4.0.
5. At any time, an author may arbitrarily decide that the number of bugfixes since a minor version constitutes another minor version. For example, 1.3.15 may become 1.4 without any code changes. It is strongly suggested that this only be done if the number of bugfixes since the previous minor version equals or

exceeds 10, however this suggestion should not be interpreted as "10 bugfixes equals a minor version": the significance and severity of the bugs should be considered, not just the quantity.

6.3 Version Terminology

In practice documentation also can have a version number. The same rules apply as well. We recommend using the term 'editions' when refering to documentation and 'revisions' when refering to code.

7. Database

The following rules are used in database related things:

1. indexes are added on primary keys
2. indexes are added on foreign keys
3. indexes are added of fields which have explicit sorting clauses

8. To be included

Exception to direct output generation are some parts of the core (notably logger and exception subsystem)

Describe hook templates very clearly, as they are the one which always get contained in another template

9. Revision history

2007-12-14: Dracos - Broke section "Version Numbers" into subsections; added subsection "Version Number Changes"

2005-10-08: jojodee - updated header section, some small updates

2004-01-15: MrB - added class section

2003-10-13: MrB - reorganized a bit, added some verbosity.

2003-04-23: FB - added PEAR coding standard

2003-03-27: MrB - created