

# PHP Coding Standard

***Last Modified: 2008-11-12***

The PHP Coding Standard is with permission **based on** [Todd Hoff's C++ Coding Standard](#).  
Rewritten for PHP by [Fredrik Kristiansen](#) / [DB Medialab](#), Oslo 2000-2003.  
Rewritten and updated By Kris Sherrerd, 2008

**Using this Standard.** If you want to make a local copy of this standard and use it as your own you are perfectly free to do so. That's why we made it! If you find any errors or make any improvements please [e-mail me](#) the changes so I can merge them in. [Recent Changes](#).

**Before you start please verify that you have the most [recent document](#).**

You can also download a this standard as a [word document](#) (maintained by [Chris Hubbard](#)).

## Contents

Not Written yet.

---

## Introduction

### ***Standardization is Important***

It helps if the standard annoys everyone in some way so everyone feels they are on the same playing field. The proposal here has evolved over many projects, many companies, and literally a total of many weeks spent arguing. It is no particular person's style and is certainly open to local amendments.

### **Good Points**

When a project tries to adhere to common standards a few good things happen:

- programmers can go into any code and figure out what's going on
- new people can get up to speed quickly
- people new to PHP are spared the need to develop a personal style and defend it to the death
- people new to PHP are spared making the same mistakes over and over again
- people make fewer mistakes in consistent environments
- programmers have a common enemy :-)

## Bad Points

Now the bad:

- the standard is usually stupid because it was made by someone who doesn't understand PHP
- the standard is usually stupid because it's not what I do
- standards reduce creativity
- standards are unnecessary as long as people are consistent
- standards enforce too much structure
- people ignore standards anyway

## Discussion

The experience of many projects leads to the conclusion that using coding standards makes the project go smoother. Are standards necessary for success? Of course not. But they help, and we need all the help we can get! Be honest, most arguments against a particular standard come from the ego. Few decisions in a reasonable standard really can be said to be technically deficient, just matters of taste. So be flexible, control the ego a bit, and remember any project is fundamentally a team effort.

## *Interpretation*

### Conventions

The use of the word "shall" in this document requires that any project using this document must comply with the stated standard.

The use of the word "must" is similar to "shall": in that it designates a requirement that must be followed.

The use of the word "should" directs projects in tailoring a project-specific standard, in that the project must include, exclude, or tailor the requirement, as appropriate.

The use of the word "may" is similar to "should", in that it designates optional requirements.

### Standards Enforcement

First, any serious concerns about the standard should be brought up and worked out within the group. Maybe the standard is not quite appropriate for your situation. It may have overlooked important issues or maybe someone in power vehemently disagrees with certain issues :-)

In any case, once finalized hopefully people will play the adult and understand that this standard is reasonable, and has been found reasonable by many other programmers, and therefore is worthy of being followed even with personal reservations.

Failing willing cooperation it can be made a requirement that this standard must be followed to pass a code inspection.

Failing that the only solution is a massive tickling party on the offending party.

## ***Accepting an Idea***

1. It's impossible.
2. Maybe it's possible, but it's weak and uninteresting.
3. It is true and I told you so.
4. I thought of it first.
5. How could it be otherwise.

If you come to objects with a negative preconception please keep an open mind. You may still conclude objects are bunk, but there's a road you must follow to accept something different. Allow yourself to travel it for a while.

---

## **Names**

### ***Make Names Fit***

Names are the heart of programming. Some stories have us believing knowing someone's true name give us magical powers over them. If you can think up the true name for something, you give yourself and the people coming after power over the code. Unlike the stories it **WILL** give you power over the code! Don't laugh!

A name is the result of a long deep thought process about the ecology it lives in. Only a programmer who understands the system as a whole can create a name that "fits" with the system. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected. Take you time thinking of names, and make then as short and descriptive as possible.

Names like Thing and DoIt should never appear and if they do you need to refactor your design and your code.

### **Class Names**

- Name the class after what it is. If you can't think of what it is that is a clue you have not thought through the design well enough.
- Compound names of over three words are a clue your design may be confusing various entities in your system. Revisit your design and see if you classes have more responsibilities then they need.
- Avoid the temptation of bringing the name of the class a class derives from into the derived class's name. A class should stand on its own. It doesn't matter what it derives from.
- Suffixes are sometimes helpful. For example, if your system uses agents then naming something DownloadAgent conveys real information.

#### **Format:**

- Must use upper case letters as word separators, lower case for the rest of a word
- Must have the first character in a name is upper case
- Shall not use all caps words
- Shall have no underscores ('\_')

## Justification

- Easy to read, easy to follow

## Example

```
class NameOneTwo  
  
class Name
```

## Method and Function Names

- Usually every method and function performs an action, so the name should make clear what it does: checkForErrors() instead of errorCheck(), dumpDataToFile() instead of dataFile(). This will also make functions and data objects more distinguishable.
- Suffixes are sometimes useful:
  - *Max* - to mean the maximum value something can have.
  - *Cnt* - the current count of a running count variable.
  - *Key* - key value.

For example: retryMax to mean the maximum number of retries, retryCnt to mean the current retry count.

- Prefixes are sometimes useful:
  - *Is* - to ask a question about something. Whenever someone sees *Is* they will know it's a question.
  - *Get* - get a value.
  - *Set* - set a value.

For example: IsHitRetryLimit.

### Format:

- Must use upper case letters as word separators, lower case for the rest of a word
- Must have the first character in a name is upper case
- Shall not use all caps words
- Shall have no underscores ('\_')

## Justification

- Again, easy to read, easy to understand

## Example

```
class NameOneTwo  
{  
    function CreateDirtyMag() {};  
    function HandleError() {};  
}
```

## Class Library Names

- Now that name spaces are becoming more widely implemented, name spaces should be used to prevent class name conflicts among libraries from different vendors and groups.
- When not using name spaces, it's common to prevent class name clashes by prefixing class names with a unique string. Two characters is sufficient, but a longer length is fine.

### Format:

- Must use upper case letters as word separators, lower case for the rest of a word
- Must have the first character in a name is upper case
- Shall not use all caps words
- Shall have no underscores ('\_')

## Example

John Johnson's complete data structure library could use *JJ* as a prefix, so classes would be:

```
class JjLinkList
{
}
```

---

## Variable Names

These should be descriptive, a noun if at all possible to describe what the variable is.

### Format:

- Must use all lower case letters
- Must use '\_' as the word separator

## Justification

- With this approach the scope of the variable is clear in the code.
- Now all variables look different and are identifiable in the code.

## Example

```
function HandleError($errorNumber)
{
    $error = new OsError;
    $time_of_error = $error->GetTimeOfError();
    $error_processor = $error->GetErrorProcessor();
}
```

## Class Variable Names

These names should have the word my in front of them, in order to distinguish that they are a part of the actual class when it is created. They also need to describe what information they will be holding if any. These are NOT to be used as a temp variable in any function or method.

### **Format:**

- my should be in front of any variables with no \_ following it and the beginning of the variable name
- Must use all lower case letters
- Must use '\_' as the word separator

### **Example**

```
class NameOneTwo
{
    var $myladder
    var $myhappy_place
}
```

### ***Method Argument Names***

These should be descriptive of what is coming in.

### **Format:**

- The first character should be lower case.
- Should use upper case letters and \_ as word separators, lower case for the rest of a word

### **Justification**

- You can always tell which variables are passed in variables.

### **Example**

```
class NameOneTwo
{
    function StartYourEngines(&$some_Engine, &$another_Engine) {
        $this->mysome_engine = $some_Engine;
        $this->myanother_engine = $another_Engine;
    }

    var $mysome_engine;
    var $myanother_engine;
}
```

### ***Array Element***

Array element names follow the same rules as a variable.

### **Format:**

- Should use all lower case letters

- Should use '\_' as the word separator
- Shall have no magic numbers (Numbers that have no meaning, place them in a meaningful variable!)

## Justification

- Arrays elements are just another type of variable in reality

## Example

```
$myarr['foo_bar'] = 'Hello';
print "$myarr[foo_bar] world"; // will output: Hello world
```

## Single or Double Quotes

- Should access an array's elements with single quotes
- Shall never use quotes within magic quotes

## Justification

- Some PHP configurations will output warnings if arrays are used without quotes except when used within magic quotes

## Example

```
$myarr['foo_bar'] = 'Hello';
$element_name = 'foo_bar';
print "$myarr[foo_bar] world"; // will output: Hello world
print "$myarr[$element_name] world"; // will output: Hello world
print "$myarr['$element_name'] world"; // parse error
print "$myarr['$element_name'] world"; // parse error
```

---

## ***Reference Variables and Functions Returning References***

You should when ever possible NOT return references, however if you must use the following format so that everyone will know that 1. the item is referenced in to a method, and 2. the data in that referenced variable could change.

### **Format:**

- Must follow function and variable naming formats
- Should place a rx in front of the variables that will be referenced (yes include the underscore as defined)
- Should place a Rx in front of functions that modify referenced variables

## Justification

- The difference between referenced and non-referenced variable types is clarified

- It establishes the difference between a method that modifies an object and the same method name returning a non-modifiable object.

## Example

```
class Test
{
    var $myrx_status;
    function DoSomething(&$RxStatus) {};
    function &RxStatus() {};
}
```

---

## ***Global Variables***

Global variables are bad, and should be used as little as possible.

### **Format:**

- Must follow variable naming formats
- Global variables should be prepended with a 'gx'.
- If stuck trying to determine what should come first in naming, Globals should always go first (ie for a referenced value)

## **Justification**

- It's important to know the scope of a variable.

## Example

```
global $gx_log;
global &$gx_rx_log;
```

## ***Define Names / Global Constants***

Constants can be a good thing. If it's a variable that will never change, consider making it a constant. Again if you want to make it a global constant, be careful globals can become more hassle than they are worth.

### **Format:**

- All word(s) must be in caps.
- All words must be use '\_' to separator words.

## **Justification**

It's tradition for global constants to named this way. You must be careful to not conflict with other predefined globals.



## Example

```
define("A_GLOBAL_CONSTANT", "Hello world!");
```

---

## Static Variables

### Format:

- Must follow variable naming formats
- Static variables may be prepended with 'sx'.

## Justification

- It's important to know the scope of a variable.

## Example

```
function test()
{
    static $sx_status = 0;
}
```

# Error Return Check Policy

- Must check every system call for an error return, unless you know that some errors should be ignores.
  - Include the system error text for every system error message.
- 

# Braces {} Policy

Of the three major brace placement strategies one of the two are acceptable methods must be used, with the first one listed being preferable:

- Place brace under and inline with keywords:

|   |  |
|---|--|
| <pre>if (\$condition) {     ... }</pre> | <pre>while (\$condition) {     ... }</pre> |
|---|--|

- Traditional Unix policy of placing the initial brace on the same line as the keyword and the trailing brace inline on its own line with the keyword:

|                               |                                  |
|-------------------------------|----------------------------------|
| <pre>if (\$condition) {</pre> | <pre>while (\$condition) {</pre> |
|-------------------------------|----------------------------------|

```
} ... }
```

## ***Justification***

- Another religious issue of great debate solved by compromise. Either form is acceptable, many people, however, find the first form more pleasant. Why is the topic of many psychological studies.

There are more reasons than psychological for preferring the first style. If you use an editor (such as vi) that supports brace matching, the first is a much better style. Why? Let's say you have a large block of code and want to know where the block ends. You move to the first brace hit a key and the editor finds the matching brace. Example:

```
if ($very_long_condition && $second_very_long_condition)
{
    ...
}
else if (...)
{
    ...
}
```

To move from block to block you just need to use cursor down and your brace matching key. No need to move to the end of the line to match a brace then jerk back and forth.

---

# **Indentation/Tabs/Space Policy**

- Indent using 4 spaces for each level.
- Do not use tabs, use spaces. Most editors can substitute spaces for tabs.
- Indent as much as needed, but no more. There are no arbitrary rules as to the maximum indenting level. If the indenting level is more than 4 or 5 levels you may think about factoring out code.

## ***Justification***

- When people using different tab settings the code is impossible to read or print, which is why spaces are preferable to tabs.
- Most PHP applications use 4 spaces.
- Most editors use 4 spaces by default.
- As much as people would like to limit the maximum indentation levels it never seems to work in general. We'll trust that programmers will choose wisely how deep to nest code.

## ***Example***

```
function func()
{
    if (something bad)
    {
        if (another thing bad)
        {
```

```
        while (more input)
        {
            }
    }
}
```

---

## Parens () with Key Words and Functions Policy

- Do not put parens next to keywords. Put a space between.
- Do put parens next to function names.
- Do not use parens in return statements when it's not necessary.

### *Justification*

- Keywords are not functions. By putting parens next to keywords keywords and function names are made to look alike.

### *Example*

```
if (condition)
{
}

while (condition)
{
}

strcmp($s, $s1);

return 1;
```

---

## Constructors and Destructors

Since this document should be updated for PHP 5 it is necessary to define how and why Constructors and Destructors should be made and called.

### **Format:**

- Should use `__construct($variables)` to create the constructor the the class.
- Should use `__destruct ()` to destruct a class.
- Because parent constructors are not called implicitly if the child class defines a constructor you must run a parent constructor, a call to **`parent::__construct()`** within the child constructor in order to construct the parent class. (assuming `__construct` is defined).
- The same is true for the destructor.

## ***Justification***

- PHP5 looks for this first, there is no reason not to you use it and save some cycles
- It's easy to see where the constructor for the class is
- It's easy to see where the destructor is.

## ***Example***

```
class DeathDevice
{
    function __construct()    { /* initialize and other stuff */ }
    function __destruct()    { delete this; }
};
```

# **Do Not do Real Work in Object Constructors**

Must not do any real work in an object's constructor. Inside a constructor initialize variables only and/or do only actions that can't fail.

If needed create another method(ie Open()) for an object which completes construction. This method should be called after object instantiation.

## ***Justification***

- Constructors can't return an error
- Constructors are just that, the real work needs to be down though methods and functions

## ***Example***

```
class Device
{
    function __construct()    { /* initialize and other stuff */ }
    function Open()    { return FAIL; }
};

$dev = new Device;
if (FAIL == $dev->Open()) exit(1);
```

---

# **Make Functions Reentrant**

Functions should not keep static variables that prevent a function from being reentrant.

---

# *If Then Else* Formatting

## **Layout**

It's up to the programmer. Different bracing styles will yield slightly different looks. One common approach is:

```
if (condition)                // Comment
{
}
else if (condition)           // Comment
{
}
else                          // Comment
{
}
```

If you have *else if* statements then it is usually a good idea to always have an else block for finding unhandled cases. Maybe put a log message in the else even if there is no corrective action taken.

## **?:**

The trouble is people usually try and stuff too much code in between the ? and :. Here are a couple of clarity rules to follow:

- Put the condition in parens so as to set it off from other code
- If possible, the actions for the test should be simple functions.
- Put the action for the then and else statement on a separate line unless it can be clearly put on one line.
- If it looks complex to you, it probably is, might be better to use the first type of if then else statement(see above) in order to clarify for you and anyone else reading your code what is going on.

## **Example**

```
(condition) ? funct1() : func2();
```

or

```
(condition)
? long statement
: another long statement;
```

## **Condition Format**

Always put the constant on the left hand side of an equality/inequality comparison. For example:

```
if ( 6 == $errorNum ) ...
```

One reason is that if you leave out one of the = signs, the parser will find the error for you. A second reason is that it puts the value you are looking for right up front where you can find it instead of buried at the end of your expression. It takes a little time to get used to this format, but then it really gets useful.

---

## *switch* Formatting

- Falling through a case statement into the next case statement shall be permitted as long as a comment is included.
- The *default* case should always be present and trigger an error if it should not be reached, yet is reached.
- If you need to create variables put all the code in a block.

### **Example**

```
switch (...)
{
    case 1:
        ...
        // FALL THROUGH

    case 2:
    {
        $v = get_week_number();
        ...
    }
    break;

    default:
}
```

---

## Use of *continue*, and *break*

### **Continue and Break**

Continue and break are really disguised gotos so they are covered here.

Continue and break like goto should be used sparingly(or never) as they are magic in code. With a simple spell the reader is beamed to god knows where for some usually undocumented reason.

The two main problems with continue are:

- It may bypass the test condition
- It may bypass the increment/decrement expression

Consider the following example where both problems occur:

```
while (TRUE)
{
    ...
    // A lot of code
    ...
}
```

```

if (/* some condition */) {
    continue;
}
...
// A lot of code
...
if ( $i++ > STOP_VALUE) break;
}

```

Note: "A lot of code" is necessary in order that the problem cannot be caught easily by the programmer.

From the above example, a further rule may be given: Mixing continue with break in the same loop is a sure way to disaster.

## ***Alignment of Declaration Blocks***

- Block of declarations should be aligned.

## ***Justification***

- Clarity.
- Similarly blocks of initialization of variables should be tabulated.
- The ‘&’ token should be adjacent to the type, not the name.

## ***Example***

```

var      $mDate
var&     $mrDate
var&     $mrName
var      $mName

$mDate   = 0;
$mrDate  = NULL;
$mrName  = 0;
$mName   = NULL;

```

---

# **One Statement Per Line**

There should be only one statement per line unless the statements are very closely related.

---

# **Short Methods**

- Methods must limit themselves to a single page of code.

## Justification

- The idea is that the each method represents a technique for achieving a single objective.
  - Most arguments of inefficiency turn out to be false in the long run.
  - True function calls are slower than not, but there needs to a thought out decision (see premature optimization).
  - The longer the method the more likely that it's doing more then needed
- 

## Document Null Statements

Always document a null body for a for or while statement so that it is clear that the null body is intentional and not missing code.

```
while ($dest++ = $src++)  
    ;           // VOID
```

---

## Do Not Default If Test to Non-Zero

Do not default the test for non-zero, i.e.

```
if (FAIL != f())  
is better than
```

```
if (f())
```

even though FAIL may have the value 0 which PHP considers to be false. An explicit test will help you out later when somebody decides that a failure return should be -1 instead of 0. Explicit comparison should be used even if the comparison value will never change; e.g., **if (!(\$bufsize % strlen(\$str)))** should be written instead as **if (0 == (\$bufsize % strlen(\$str)))** to reflect the numeric (not boolean) nature of the test. A frequent trouble spot is using strcmp to test for string equality, where the result should *never ever* be defaulted.

The non-zero test is often defaulted for predicates and other functions or expressions which meet the following restrictions:

- Returns 0 for false, nothing else.
  - Is named so that the meaning of (say) a **true** return is absolutely obvious. Call a predicate IsValid(), not CheckValid().
-



# The Bull of Boolean Types

Do not check a boolean value for equality with 1 (TRUE, YES, etc.); instead test for inequality with 0 (FALSE, NO, etc.). Most functions are guaranteed to return 0 if false, but only non-zero if true. Thus,

```
if (TRUE == func()) { ...
```

must be written

```
if (FALSE != func()) { ...
```

---

## Usually Avoid Embedded Assignments

There is a time and a place for embedded assignment statements. In some constructs there is no better way to accomplish the results without making the code bulkier and less readable.

```
while ($a != ($c = getchar()))
{
    process the character
}
```

The ++ and -- operators count as assignment statements. So, for many purposes, do functions with side effects. Using embedded assignment statements to improve run-time performance is also possible. However, one should consider the tradeoff between increased speed and decreased maintainability that results when embedded assignments are used in artificial places. For example,

```
$a = $b + $c;
$d = $a + $r;
```

should not be replaced by

```
$d = ($a = $b + $c) + $r;
```

even though the latter may save one cycle. In the long run the time difference between the two will decrease as the optimizer gains maturity, while the difference in ease of maintenance will increase as the human memory of what's going on in the latter piece of code begins to fade.

---

## Reusing Your Hard Work and the Hard Work of Others

Reuse across projects is almost impossible without a common framework in place. Objects conform to the

services available to them. Different projects have different service environments making object reuse difficult.

Developing a common framework takes a lot of up front design effort. When this effort is not made, for whatever reasons, there are several techniques one can use to encourage reuse:

### ***Don't be Afraid of Small Libraries***

One common enemy of reuse is people not making libraries out of their code. A reusable class may be hiding in a program directory and will never have the thrill of being shared because the programmer won't factor the class or classes into a library.

One reason for this is because people don't like making small libraries. There's something about small libraries that doesn't feel right. Get over it. The computer doesn't care how many libraries you have.

If you have code that can be reused and can't be placed in an existing library then make a new library. Libraries don't stay small for long if people are really thinking about reuse.

If you are afraid of having to update makefiles when libraries are recomposed or added then don't include libraries in your makefiles, include the idea of **services**. Base level makefiles define services that are each composed of a set of libraries. Higher level makefiles specify the services they want. When the libraries for a service change only the lower level makefiles will have to change.

### ***Keep a Repository***

Most companies have no idea what code they have. And most programmers still don't communicate what they have done or ask for what currently exists. The solution is to keep a repository of what's available.

In an ideal world a programmer could go to a web page, browse or search a list of packaged libraries, taking what they need. If you can set up such a system where programmers voluntarily maintain such a system, great. If you have a librarian in charge of detecting reusability, even better.

Another approach is to automatically generate a repository from the source code. This is done by using common class, method, library, and subsystem headers that can double as man pages and repository entries.

---

## **Comments on Comments**

### ***Comments Should Tell a Story***

Consider your comments a story describing the system. Expect your comments to be extracted by a robot and formed into a man page. Class comments are one part of the story, method signature comments are another part of the story, method arguments another part, and method implementation yet another part. All these parts should weave together and inform someone else at another point of time just exactly what you did and why.

## ***Document Decisions***

Comments should document decisions. At every point where you had a choice of what to do place a comment describing which choice you made and why. Archeologists will find this the most useful information.

## ***Use Headers for functions***

Use something to describe the function and what it does. This way people reading over your code can quickly look and see exactly what it should be doing

These headers are structured in such a way as they can be parsed and extracted. They are not useless like normal headers. So take time to fill them out. If you do it right once no more documentation may be necessary.

Example:

```
/*#####  
Function: MyFunction  
Needs: Int $number Boolean $rx_show_this  
Modified Referenced Values: $rx_show_this  
returns: int $wow  
what does it do: Take the number and determines if a series of wow numbers should be shown.  
#####*/  
function MyFunction($number &$rx_show_this)  
{  
  //..  
}
```

## ***Comment Layout***

Each part of the project has a specific comment layout.

## ***Make Gotchas Explicit***

Explicitly comment variables changed out of the normal control flow or other code likely to break during maintenance. Embedded keywords are used to point out issues and potential problems. Consider a robot will parse your comments looking for keywords, stripping them out, and making a report so people can make a special effort where needed.

## ***Gotcha Keywords***

- **:TODO: topic**  
Means there's more to do here, don't forget.
- **:BUG: [bugid] topic**  
means there's a Known bug here, explain it and optionally give a bug ID.
- **:KLUDGE:**

When you've done something ugly say so and explain how you would do it differently next time if you had more time.

- **:TRICKY:**  
Tells somebody that the following code is very tricky so don't go changing it without thinking.
- **:WARNING:**  
Beware of something.
- **:PARSER:**  
Sometimes you need to work around a parser problem. Document it. The problem may go away eventually.
- **:ATTRIBUTE: value**  
The general form of an attribute embedded in a comment. You can make up your own attributes and they'll be extracted.

## Gotcha Formatting

- Make the gotcha keyword the first symbol in the comment.
- Comments may consist of multiple lines, but the first line should be a self-containing, meaningful summary.
- The writer's name and the date of the remark should be part of the comment. This information is in the source repository, but it can take a quite a while to find out when and by whom it was added. Often gotchas stick around longer than they should. Embedding date information allows other programmer to make this decision. Embedding who information lets us know who to ask.

## Example

```
// :TODO: tmh 960810: possible performance problem
// We should really use a hash table here but for now we'll
// use a linear search.

// :KLUDGE: tmh 960810: possible unsafe type cast
// We need a cast here to recover the derived type. It should
// probably use a virtual method or template.
```

## Note:

One thing about LARGE amounts of comments, while these are nice they can point to bugs, code that should be refactored or worse, the comments could be wrong. In a perfect world people would update comments but they don't. Keep comments simple, and keep your code clear so that you only have to comment things that are truly complex.

# Interface and Implementation Documentation

There are two main audiences for documentation:

- Class Users

- Class Implementors

With a little forethought we can extract both types of documentation directly from source code.

## ***Class Users***

Class users need class interface information which when structured correctly can be extracted directly from a header file. When filling out the header comment blocks for a class, only include information needed by programmers who use the class. Don't delve into algorithm implementation details unless the details are needed by a user of the class. Consider comments in a header file a man page in waiting.

## ***Class Implementors***

Class implementors require in-depth knowledge of how a class is implemented. This comment type is found in the source file(s) implementing a class. Don't worry about interface issues. Header comment blocks in a source file should cover algorithm issues and other design decisions. Comment blocks within a method's implementation should explain even more.

---

# **File Documentation**

Every file should have a overall header comment that covers:

- the purpose of the file and what it contains
- direct people to related resources:
  - directories of source
  - online documentation
  - paper documentation
  - design documentation
- anything else that might help someone

In addition depending on the project size it may also be useful to have one or more ReadMe.txt files(the more would mean that each directory should have one) that cover the following:

- the purpose of the directory and what it contains
- a one line comment on each file. A comment can usually be extracted from the NAME attribute of the file header.
- cover build and install directions
- direct people to related resources:
  - directories of source
  - online documentation
  - paper documentation
  - design documentation
- anything else that might help someone

Consider a new person coming in 6 months after every original person on a project has gone. That lone scared explorer should be able to piece together a picture of the whole project by traversing a source directory tree and reading README files, and source file headers.

---

# Open/Closed Principle

The Open/Closed principle states a class must be open and closed where:

- open means a class has the ability to be extended.
- closed means a class is closed for modifications other than extension. The idea is once a class has been approved for use having gone through code reviews, unit tests, and other qualifying procedures, you don't want to change the class very much, just extend it.

The Open/Closed principle is a pitch for stability. A system is extended by adding new code not by changing already working code. Programmers often don't feel comfortable changing old code because it works! This principle just gives you an academic sounding justification for your fears :-)

In practice the Open/Closed principle simply means making good use of our old friends abstraction and polymorphism. Abstraction to factor out common processes and ideas. Inheritance to create an interface that must be adhered to by derived classes.

---

## Server configuration

This section contains some guidelines for PHP/Apache configuration.

## HTTP\_\*\_VARS

HTTP\_\*\_VARS are either enabled or disabled. When enabled all variables must be accessed through \$HTTP\_\*\_VARS[key]. When disabled all variables can be accessed by the key name.

- use HTTP\_\*\_VARS when accessing variables.
- use enabled HTTP\_\*\_VARS in PHP configuration.

### *Justification*

- HTTP\_\*\_VARS is available in any configuration.
  - HTTP\_\*\_VARS will not conflict with existing variables.
  - Users can't change variables by passing values.
- 

## PHP File Extensions

There is lots of different extension variants on PHP files (.html, .php, .php3, .php4, .phtml, .inc, .class...).

- Must always use the extension `.php`.
- Must always use the extension `.php` for your class and function libraries.

## ***Justification***

- The use of `.php` makes it possible to enable caching on other files than `.php`.
  - The use of `.inc` or `.class` can be a security problem. On most servers these extensions aren't set to be run by a parser. If these are accessed they will be displayed in clear text.
- 

## **Miscellaneous**

This section contains some miscellaneous do's and don'ts.

- Don't use floating-point variables where discrete values are needed. Using a float for a loop counter is a great way to shoot yourself in the foot. Always test floating-point numbers as `<=` or `>=`, never use an exact comparison (`==` or `!=`).
- Do not rely on automatic beautifiers. The main person who benefits from good program style is the programmer him/herself, and especially in the early design of handwritten algorithms or pseudo-code. Automatic beautifiers can only be applied to complete, syntactically correct programs and hence are not available when the need for attention to white space and indentation is greatest. Programmers can do a better job of making clear the complete visual layout of a function or file, with the normal attention to detail of a careful programmer (in other words, some of the visual layout is dictated by intent rather than syntax and beautifiers cannot read minds). Sloppy programmers should learn to be careful programmers instead of relying on a beautifier to make their code readable. Finally, since beautifiers are non-trivial programs that must parse the source, a sophisticated beautifier is not worth the benefits gained by such a program. Beautifiers are best for gross formatting of machine-generated code.
- Accidental omission of the second ```=` of the logical compare is a problem. The following is confusing and prone to error.

```
if ($abool= $bbool) { ... }
```

Does the programmer really mean assignment here? Often yes, but usually no. The solution is to just not do it, an inverse Nike philosophy. Instead use explicit tests and avoid assignment with an implicit test. The recommended form is to do the assignment before doing the test:

```
$abool= $bbool;  
if ($abool) { ... }
```

---

# Use if (0) to Comment Out Code Blocks

Sometimes large blocks of code need to be commented out for testing. The easiest way to do this is with an if (false) block:

```
function example()
{
    great looking code

    if (false) {
        //lots of code
    }

    // more code
}
```

You can't use `/**/` style comments because comments can't contain comments and surely a large block of your code will contain a comment, won't it?

## Test Till you break it, then FIX IT!

One thing that is used in Agile programming is testing, and while it should be considered already happening I want to make it a part of the standard coding. If you don't write tests before you write code I recommend that you start.

- If you should choose to code with tests you must always write the tests before you write any code.
- If you should choose to code with tests you must always run the tests when ever you have modified code.
- If you should choose to code with tests you must always write 1 test package per class and keep it with the class so that when you code is modified the test can be run to ensure everything still works!

### ***Justification***

- If you write the test before your write the code you already have clear idea of what you must write. All you have to do is write the code to pass the test.
- Find possible problems NOW not later. By running these tests you can find out now that you broke something and get it fixed. Which is much MUCH better then releasing the code and finding out through your users.

---

## Different Accessor Styles

### ***Implementing Accessors***

There are two major idioms for creating accessors.



## Get/Set

```
class X
{
    function GetAge()          { return $this->mAge; }
    function SetAge($age)     { $this->mAge = $age; }
    var $mAge;
};
```

Get/Set is ugly. Get and Set are strewn throughout the code cluttering it up.

But one benefit is when used with messages the set method can transparently transform from native machine representations to network byte order.

## Attributes as Objects

```
class X
{
    function      Age()          { return $this->mAge; }
    function      Name()         { return $this->mName; }

    var           $mAge;
    var           $mName;
}

$x = new X;

// Example 1
$age = $x->Age();
$r_age = &$x->Age(); // Reference

// Example 2
$name = $x->Name();
$r_name = &$x->Name(); // Reference
```

Attributes as Objects is clean from a name perspective. When possible use this approach to attribute access.

---

## Layering

Layering is the primary technique for reducing complexity in a system. A system should be divided into layers. Layers should communicate between adjacent layers using well defined interfaces. When a layer uses a non-adjacent layer then a layering violation has occurred.

A layering violation simply means we have dependency between layers that is not controlled by a well defined interface. When one of the layers changes code could break. We don't want code to break so we want layers to work only with other adjacent layers.

Sometimes we need to jump layers for performance reasons. This is fine, but we should know we are doing it and document appropriately.

---

# Code Reviews

If you can make a formal code review work then my hat is off to you. Code reviews can be very useful. Unfortunately they often degrade into nit picking sessions and endless arguments about silly things. They also tend to take a lot of people's time for a questionable payback.

My god he's questioning code reviews, he's not an engineer!

Not really, it's the form of code reviews and how they fit into normally late chaotic projects is what is being questioned.

First, code reviews are **way too late** to do much of anything useful. What needs reviewing are requirements and design. This is where you will get more bang for the buck.

Get all relevant people in a room. Lock them in. Go over the class design and requirements until the former is good and the latter is being met. Having all the relevant people in the room makes this process a deep fruitful one as questions can be immediately answered and issues immediately explored. Usually only a couple of such meetings are necessary.

If the above process is done well coding will take care of itself. If you find problems in the code review the best you can usually do is a rewrite after someone has sunk a ton of time and effort into making the code "work."

You will still want to do a code review, just do it offline. Have a couple people you trust read the code in question and simply make comments to the programmer. Then the programmer and reviewers can discuss issues and work them out. Email and quick pointed discussions work well. This approach meets the goals and doesn't take the time of 6 people to do it.

One thing of note: even with this document as your ultimate standard it is possible that some things just won't be seen the same way. Keep in the following in mind when doing a code review for some one (or getting one done by someone else) "Don't sweat the small stuff". If you have a personal style not covered here nor used by you, that's fine. If the code works, is readable, and is to the best of your (and their) knowledge correct, be happy!

---

## Create a Source Code Control System Early and Not Often

A common build system and source code control system should be put in place as early as possible in a project's lifecycle, preferably before anyone starts coding. Source code control is the structural glue binding a project together. If programmers can't easily use each other's products then you'll never be able to make a good reproducible build and people will piss away a lot of time. It's also hell converting rogue build environments to a standard system. But it seems the right of passage for every project to build their own custom environment that never quite works right.

Some issues to keep in mind:

- Shared source environments like CVS or SVN usually work best in largish projects.

- If you use CVS use a *reference tree* approach. With this approach a master build tree is kept of various builds. Programmers checkout source against the build they are working on. They only checkout what they need because the make system uses the build for anything not found locally. Using the *-I* and *-L* flags makes this system easy to setup. Search locally for any files and libraries then search in the reference build. This approach saves on disk space and build time.
- Get a lot of disk space. With disk space as cheap it is there is no reason not to keep plenty of builds around.
- Make simple things simple. It should be dead simple and well documented on how to:
  - check out modules to build
  - how to change files
  - how to add new modules into the system
  - how to delete modules and files
  - how to check in changes
  - what are the available libraries and include files
  - how to get the build environment including all compilers and other tools

Make a web page or document or whatever. New programmers shouldn't have to go around begging for build secrets from the old timers.

- On check in's log comments should be useful. These comments should be collected every night and sent to interested parties.

Recommended Systems:

These are a few popular open source systems.

CVS - [http://ximbiot.com/cvs/wiki/Main%20Page#CVS\\_Downloads](http://ximbiot.com/cvs/wiki/Main%20Page#CVS_Downloads)

\*SVN - <http://subversion.tigris.org/>

\*Used in all my projects.

## Create a Bug Tracking System Early and Not Often

The earlier people get used to using a bug tracking system the better. If you are 3/4 through a project and then install a bug tracking system it won't be used. You need to install a bug tracking system early so people will use it.

Programmers generally resist bug tracking, yet when used correctly it can really help a project:

- Problems aren't dropped on the floor.
- Problems are automatically routed to responsible individuals.
- The lifecycle of a problem is tracked so people can argue back and forth with good information.
- Managers can make the big schedule and staffing decisions based on the number of and types of bugs in the system.
- Configuration management has a hope of matching patches back to the problems they fix.
- QA and technical support have a communication medium with developers.

Not sexy things, just good solid project improvements.

Source code control should be linked to the bug tracking system. During the part of a project where source is frozen before a release only checkins accompanied by a valid bug ID should be accepted. And when code is changed to fix a bug the bug ID should be included in the checkin comments.

## Sources

Bugzilla <http://www.bugzilla.org/>

# Honor Responsibilities

Responsibility for software modules is scoped. Modules are either the responsibility of a particular person or are common. Honor this division of responsibility. Don't go changing things that aren't your responsibility to change. Only mistakes and hard feelings will result.

Face it, if you don't own a piece of code you can't possibly be in a position to change it. There's too much context. Assumptions seemingly reasonable to you may be totally wrong. If you need a change simply ask the responsible person to change it. Or ask them if it is OK to make such-n-such a change. If they say OK then go ahead, otherwise holster your editor.

Every rule has exceptions. If it's 3 in the morning and you need to make a change to make a deliverable then you have to do it. If someone is on vacation and no one has been assigned their module then you have to do it. If you make changes in other people's code try and use the same style they have adopted.

Programmers need to mark with comments code that is particularly sensitive to change. If code in one area requires changes to code in another area then say so. If changing data formats will cause conflicts with persistent stores or remote message sending then say so. If you are trying to minimize memory usage or achieve some other end then say so. Not everyone is as brilliant as you.

The worst sin is to flit through the system changing bits of code to match your coding style. If someone isn't coding to the standards then ask them or ask your manager to ask them to code to the standards. Use common courtesy.

Code with common responsibility should be treated with care. Resist making radical changes as the conflicts will be hard to resolve. Put comments in the file on how the file should be extended so everyone will follow the same rules. Try and use a common structure in all common files so people don't have to guess on where to find things and how to make changes. Checkin changes as soon as possible so conflicts don't build up.

As an aside, module responsibilities must also be assigned for bug tracking purposes.

---

## PHP Code Tags

PHP Tags are used for delimit PHP from html in a file. There are several ways to do this. `<?php ?>`, `<? ?>`, `<script language="php"> </script>`, `<% %>`, and `<?=$name?>`. Some of these may be turned off in your PHP settings.

- Use `<?php ?>`

## Justification

- `<?php ?>` is always available in any system and setup.

## Example

```
<?php print "Hello world"; ?> // Will print "Hello world"
```

## No Magic Numbers

A magic number is a bare-naked number used in source code. It's magic because no-one has a clue what it means including the author inside 3 months. For example:

```
if (22 == $foo) { start_thermo_nuclear_war(); }
else if (19 == $foo) { refund_lotso_money(); }
else if (16 == $foo) { infinite_loop(); }
else { cry_cause_im_lost(); }
```

In the above example what do 22 and 19 mean? If there was a number change or the numbers were just plain wrong how would you know?

Heavy use of magic numbers marks a programmer as an amateur more than anything else. Such a programmer has never worked in a team environment or has had to maintain code or they would never do such a thing.

Instead of magic numbers use a real name that means something. You should use `define()`. For example:

```
define("PRESIDENT_WENT_CRAZY", "22");
define("WE_GOOFED", "19");
define("THEY_DIDNT_PAY", "16");

if (PRESIDENT_WENT_CRAZY == $foo) { start_thermo_nuclear_war(); }
else if (WE_GOOFED == $foo) { refund_lotso_money(); }
else if (THEY_DIDNT_PAY == $foo) { infinite_loop(); }
else { happy_days_i_know_why_im_here(); }
```

Now isn't that better?

---

## Refactor Refactor Refactor!!!

Ok, now on to reality, If you have used this standard 100% and you know your code is the best in the universe, there may still be “bad smells” in it. If you have not refactored at all in your project chances are you have lots of “bad smells”, some may be an issue of taste (nothing wrong with them, just not pretty code), some of them may actually be things that can be made better. The personal taste we can do nothing about, but the other, we can refactor it till it shines!

## Justification

- Makes code more efficient (usually), more readable, easier to update.

## Recent Changes

1. 2003-02-17. Modified indent rule.
2. 2002-03-04. Some changes in PHP File Extensions section. Only .php extensions is now recommended.
3. 2002-01-23. I've added Array Element.
4. 2001-08-13. The Variable Names example is now compatible with this PHP standard. Added word version of this document submitted by Chris Hubbard.
5. 2001-01-23. Method Argument Names example code fix. Parts of Different Accessor Styles has been deprecated because there was no support in PHP for these.
6. 2000-12-12. HTTP\_\*\_VARS added
7. 2000-12-11. Indentation/Tabs/Space Policy has been changed  
PHP Code Tags added
8. 2000-12-05. Method Argument Names has been updated