

MYSOURCE MATRIX

CODING STANDARDS

TABLE OF CONTENTS

1. INTRODUCTION	04
→ 1.1. Why Coding Standards?	04
2. CONTROL STRUCTURES	05
→ 2.1. BREAK Statement Alignment (CS05) v1.0	05
→ 2.2. Condition Spacing (CS02) v1.0	05
→ 2.3. Control Structure Declarations (CS01) v2.1	05
→ 2.4. ELSE IF Spacing (CS03) v1.0	07
→ 2.5. Inline IF statement (CS06) v1.1	08
→ 2.6. ELSE Spacing (CS04) v1.0	08
3. LOGICAL OPERATORS	09
→ 3.1. Use of Logical Operators (LO01) v1.0	09
4. ARRAY DECLARATIONS	10
→ 4.1. Multi-line Array Declarations (AD02) v2.0	10
→ 4.2. Simple Array Declarations (AD01) v1.0	11
5. CLASS DECLARATIONS	12
→ 5.1. Opening Class Brace (CD01) v1.0	12
6. FUNCTION DECLARATIONS	13
→ 6.1. Default Function Argument Spacing (FD02) v1.1	13
→ 6.2. Opening Function Brace (FD01) v1.0	13

TABLE OF CONTENTS

7. FUNCTION CALLS	14
→ 7.1. Assigning Referenced Return Values (FC01) v1.0	14
8. VARIABLE ASSIGNMENT	15
→ 8.1. Instantiating New Objects (VA02) v1.0	15
→ 8.2. Assignment by Reference (VA01) v1.0	15
9. STRINGS AND QUOTES	16
→ 9.1. Quoting Strings (SQ01) v1.0	16
→ 9.2. Heredoc Syntax (SQ02) v1.0	16
→ 9.3. String Concatenation (SQ03) v1.0	17
10. WHITESPACE	18
→ 10.1. Whitespace at End of Line (WS01) v1.0	18
→ 10.2. Whitespace Following End of File (WS03) v1.0	18
→ 10.3. Whitespace Proceeding Start of File (WS02) v1.0	19
→ 10.4. Function and Method Spacing (WS04) v1.0	20
11. NAMING CONVENTIONS	21
→ 11.1. Naming Class Methods (NC01) v1.0	21
→ 11.2. PHP Keywords (NC02) v1.0	21
12. DATABASE QUERIES	22
→ 12.1. Asserting Valid DB Results (MY02) v1.0	22
→ 12.2. Database Transactions (MY03) v1.1	22

TABLE OF CONTENTS

13. MIXING PHP AND HTML	23
→ 13.1. PHP Code Tags (PH01) v1.0	23
14. COMMENTING	24
→ 14.1. Inline Comments (CM01) v1.0	24
→ 14.2. Function Comments (CM02) v1.0	24
→ 14.3. Debug Functions (MY01) v1.0	26

1. INTRODUCTION

→ 1.1. WHY CODING STANDARDS?

Being an open source project, [Squiz.net](https://www.squiz.net) strives to ensure that all code produced for MySource Matrix meets a defined coding standard, be it developed by one of our own developers or contributed by another member of the open source community. As a general rule, if you can look through the code base and recognise a particular developer's code, the coding standards have not been met. All code should look as if it has been produced by the same person.

When contributing code for MySource Matrix, be it a patch for the core or an external module, please ensure you meet the MySource Matrix Coding Standards.

2. CONTROL STRUCTURES

→ 2.1. BREAK STATEMENT ALIGNMENT (CS05) V1.0

Case statements that contain a **break** keyword should have the break keyword indented to the same column as the matching **case** or **default** keyword.

Valid: break indented correctly	Invalid: break indented past case keyword
<pre>switch.(\$foo).{ > case.\$bar: > > ... > break; }</pre>	<pre>switch.(\$foo).{ > case.\$bar: > > ... > >> break; }</pre>

→ 2.2. CONDITION SPACING (CS02) V1.0

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

Valid: single space after keyword	Invalid: no space before opening parenthesis
<pre>foreach (\$foo.as.\$bar).{ > if (\$bar).{ > > ... > } }</pre>	<pre>foreach(\$foo.as.\$bar).{ > if(\$bar).{ > > ... > } }</pre>

→ 2.3. CONTROL STRUCTURE DECLARATIONS (CS01) V2.1

With the exception of some IF statements, all control structure declarations should use curly braces to begin and end the code block. This applies to the following control structure keywords:

if, else, for, foreach, while, do, switch

→ 2.3. CONTROL STRUCTURE DECLARATIONS (CS01) V2.1 (CONTINUED)

Valid: curly braces used	Invalid: defined inline
<pre>foreach (\$foo.as.\$bar) { > \$bar = .false; }</pre>	<pre>foreach (\$foo.as.\$bar) \$bar = .false;</pre>

When control structure declarations do use curly braces to begin and end the code block, each curly brace must be the last PHP code token on the line. The only exception to this rule is for closing curly braces that are followed by another control structure declaration, such as **DO/WHILE**, **IF/ELSE**, or **IF/ELSE IF**.

Valid: curly braces not followed by PHP code	Invalid: opening brace followed by PHP code
<pre>foreach (\$foo.as.\$bar) { > \$bar = .false; } //.end.of.block.comments.are.not.counted.as //.being.PHP.tokens.and.so.the.following.code //.block.is.also.valid.under.this.standard foreach (\$foo.as.\$bar) { > \$bar = .false; } //end.foreach</pre>	<pre>foreach (\$foo.as.\$bar) { \$bar = .false; }</pre>

Valid: ELSE is part of the same statement	Invalid: second IF is a new statement
<pre>if (empty(\$foo)) { > \$bar = .false; } else { > \$bar = .true; }</pre>	<pre>if (empty(\$foo)) { > \$bar = .false; }; if (!empty(\$foo)) { > \$bar = .true; }</pre>

IF statements **must** be declared inline (without curly braces and on the same line) if the following conditions are satisfied:

1. it does not have an ELSE clause
2. it does not contain the boolean operators **&&**, **&**, **//** or **|** in the condition
3. the total length of the condition and action block is less than or equal to **55 characters**

Valid: has ELSE clause and uses braces	Invalid: has ELSE clause - must use braces
<pre>if (empty(\$foo)) { > \$bar = .false; } else { > \$bar = .true; }</pre>	<pre>if (empty(\$foo)) \$bar = .false; else \$bar = .true;</pre>

→ 2.3. CONTROL STRUCTURE DECLARATIONS (CS01) V2.1 (CONTINUED)

Valid: simple statement	Invalid: boolean AND used - must use braces
<pre>if.(empty(\$foo)).\$bar.=.false;</pre>	<pre>if.(empty(\$foo).&&.\$bar).\$bar.=.false;</pre>

Valid: short IF defined inline	Invalid: short IF should not use braces
<pre>if.(empty(\$foo)).\$bar.=.false;</pre>	<pre>if.(empty(\$foo)).{ > \$bar.=.false; }</pre>

```
//.this.is.not.valid,.the.length.of.the.statement.is.greater.than.55.characters  
//(excess.characters.are.highlighted)  
if.(!empty($error)).trigger_error($error,.E_USER_WARNING);  
  
//.this.is.correctly.formatted  
if.(!empty($error)).{  
> trigger_error($error,.E_USER_WARNING);  
}
```

Commenting If the control structure is longer than 30 lines of codes, it should have the end of block comment

```
if.(TRUE).{  
> ...more.than.30.lines.of.codes.here  
} //end.if.[Optional.comments.after.the.type.of.control.structure]
```

→ 2.4. ELSE IF SPACING (CS03) V1.0

Conditional control statements that contain an **else if** keyword should have one space between the **else** and **if** keywords.

Valid: single space between ELSE and IF	Invalid: no space between ELSE and IF
<pre>if.(\$foo).{ > ... }.else if.(\$bar).{ > ... }</pre>	<pre>if.(\$foo).{ > ... }.elseif.(\$bar).{ > ... }</pre>

→ 2.5. INLINE IF STATEMENT (CS06) V1.1

Inline IF statements should contain spacing around the THEN and ELSE operators.

Valid: spaces around operators	Invalid: missing spaces around operators
<pre>\$foo.=.(TRUE) .?.'hello' .:.'world';</pre>	<pre>\$foo.=.(TRUE)?'hello':'world';</pre>

→ 2.6. ELSE SPACING (CS04) V1.0

Conditional control statements that contain an **else** keyword should have one space between the **else** keyword and each of the opening and closing braces.

Valid: single space before and after ELSE	Invalid: no space before or after ELSE
<pre>if.(\$foo){ > ... } else { > ... }</pre>	<pre>if.(\$foo){ > ... }else{ > ... }</pre>

3. LOGICAL OPERATORS

→ 3.1. USE OF LOGICAL OPERATORS (LO01) V1.0

When creating logic expressions, use the lazy operators **&&** and **//** instead of the lower precedence operators **and** and **or**. The difference between the two sets of operators is the order of precedence in which they are executed. Parentheses should be used instead of **and** and **or** to achieve the same effect.

Valid: parentheses used to set precedence	Invalid: lower precedence operators used
<pre>if.(\$a && (\$b \$c)).{ > ... }</pre>	<pre>if.(\$a and \$b \$c).{ > ... }</pre>

4. ARRAY DECLARATIONS

→ 4.1. MULTI-LINE ARRAY DECLARATIONS (AD02) V2.0

This standard covers all array declarations that span multiple lines, regardless of the number and type of values contained within the array.

The first array key must begin on the line after the **Array** keyword.

Valid: first key on second line	Invalid: first key on same line
<pre>\$array = .Array(> > > 'key1'> => . 'value1', > > > 'key2'> => . 'value2', > > .);</pre>	<pre>\$array = .Array('key1'> => . 'value1', > > > 'key2'> > > => . 'value2', > > .);</pre>

All array keys must be aligned at the first tab stop after the start of the **Array** keyword. Array keys cannot be aligned directly under the **Array**.

Valid: keys aligned at first tab stop	Invalid: keys aligned incorrectly
<pre>\$array = .Array(> > > 'key1'> => . 'value1', > > > 'key2'> => . 'value2', > > .);</pre>	<pre>\$array = .Array(> > > .. 'key1'> => . 'value1', > > > 'key2'> => . 'value2', > > .);</pre>

All double arrow symbols are aligned at the first tab stop after the longest array key. Alignment must be achieved using tabs.

Valid: aligned using tabs	Invalid: spaces used and no alignment
<pre>\$array = .Array(> > > 'key1'> => . 'value1', > > > 'key2'> => . 'value2', > > .);</pre>	<pre>\$array = .Array(> > > 'key1' .=> . 'value1', > > > 'key2' .=> . 'value2', > > .); \$array = .Array(> > > 'key1' .=> . 'value1', > > > 'key22' .=> . 'value2', > > .);</pre>

→ 4.1. MULTI-LINE ARRAY DECLARATIONS (AD02) V2.0 (CONTINUED)

All array values must be followed by a comma, including the final value.

Valid: comma after each value	Invalid: no comma after last value
<pre>\$array = .Array(> > > 'key1'> => . 'value1', > > > 'key2'> => . 'value2', > > > 'key3'> => . 'value3', > > .);</pre>	<pre>\$array = .Array(> > > 'key1'> => . 'value1', > > > 'key2'> => . 'value2', > > > 'key3'> => . 'value3' > > .);</pre>

The closing parenthesis must be aligned exactly under the **A** in the **Array** keyword. Both tabs and spaces should be used to achieve the alignment.

Valid: closing parenthesis aligned	Invalid: closing parenthesis not aligned
<pre>\$array = .Array(> > > 'key1'> => . 'value1', > > > 'key2'> => . 'value2', > > .);</pre>	<pre>\$array = .Array(> > > 'key1'> => . 'value1', > > > 'key2'> => . 'value2', > > >);</pre>

→ 4.2. SIMPLE ARRAY DECLARATIONS (AD01) V1.0

When defining a new array, capitalise the **a** as if creating a new object.

Valid: uppercase A	Invalid: lowercase A
<pre>\$array = .Array(); \$array = .Array(> > > 'key1'> => . 'value1', > > > 'key2'> => . 'value2', > > .);</pre>	<pre>\$array = .array(); \$array = .array(> > > 'key1'> => . 'value1', > > > 'key2'> => . 'value2', > > .);</pre>

5. CLASS DECLARATIONS

→ 5.1. OPENING CLASS BRACE (CD01) V1.0

The opening brace for a class should be placed on the line after the class definition.

Valid: opening brace on next line	Invalid: opening brace on same line
<pre>class foo { ... ></pre>	<pre>class foo { > ... ></pre>

6. FUNCTION DECLARATIONS

→ 6.1. DEFAULT FUNCTION ARGUMENT SPACING (FD02) V1.1

Arguments with default values go at the end of the argument list. There should be no spaces between a default value, the equals sign, and the argument name.

Valid: no space before or after equals sign

```
function.getLink($id,.$side='major')....
```

Invalid: space before and after equals sign

```
function.getLink($id,.$side='major')....
```

→ 6.2. OPENING FUNCTION BRACE (FD01) V1.0

Function declarations follow the "one true brace" convention. The opening brace for a function should be placed on the line after the function definition.

Valid: opening brace on next line

```
function.foo()
{
> ...
```

Invalid: opening brace on same line

```
function.foo(){
> ...
```

7. FUNCTION CALLS

→ 7.1. ASSIGNING REFERENCED RETURN VALUES (FC01) V1.0

When a function returns by reference, the value should be assigned by reference.

```
class Foo
{
>   function &getBar().{
>   >   ...
>   }
}
```

Valid: the value is assigned properly

```
$foo = &.new.Foo();
$bar = &.$foo->getBar();
```

Invalid: return by reference ignored

```
$foo = &.new.Foo();
$bar = .$foo->getBar();
```

8. VARIABLE ASSIGNMENT

→ 8.1. INSTANTIATING NEW OBJECTS (VA02) V1.0

New objects should always be assigned to variables by reference, or returned by reference from functions.

Valid: assigned by reference	Invalid: missing '&' after '='
<code>\$foo.=&.new.Bar();</code>	<code>\$foo.=.new.Bar();</code>
Valid: function returns by reference	Invalid: missing '&' before foo()
<pre>function.&foo().{ > return.new.Bar(); }</pre>	<pre>function.foo().{ > return.new.Bar(); }</pre>

→ 8.2. ASSIGNMENT BY REFERENCE (VA01) V1.0

Assignment by Reference operator should be located after '=' without a space.

Valid: correct spacing	Invalid: incorrect spacing
<code>\$foo.=&.\$this->db;</code>	<code>\$foo.=.&\$this->db;</code>

9. STRINGS AND QUOTES

→ 9.1. QUOTING STRINGS (SQ01) V1.0

Strings should be quoted using **single quotes** unless the string contains variables, newline characters, tab characters, or single quotes. Only in these cases may strings be quoted using **double quotes**.

Valid: simple string uses single quotes	Invalid: double quotes used
<pre>\$string.='This is a string';</pre>	<pre>\$string.="This is a string";</pre>

```
//.the.following.are.valid.uses.of.double.quoted.strings

//.this.example.uses.a.variable.within.the.string
$string.="This is a string with $num_vars variables";

//.tabs.and.newlines.are.used.in.this.example
$string.="Column.1\t\tColumn2\tColumn3\n\n";
```

→ 9.2. HEREDOC SYNTAX (SQ02) V1.0

The use of heredoc syntax for specifying strings is prohibited within MySource Matrix to ensure all strings are assigned in a standard format. If you are attempting something complex and heredoc syntax seems like your only option, try thinking a bit harder and cleaning up your code.

Valid: standard string assignment	Invalid: heredoc syntax used
<pre>\$format.='<tr> > > > > <td> > > > > > Cell.1 > > > > </td> > > > > <td> > > > > > Cell.2 > > > > </td> > > > > <td> > > > > > Cell.3 > > > > </td> > > > </tr>';</pre>	<pre>\$format.=<<<HTML > > > > <tr> > > > > > <td> > > > > > > Cell.1 > > > > > </td> > > > > > <td> > > > > > > Cell.2 > > > > > </td> > > > > > <td> > > > > > > Cell.3 > > > > > </td> > > > > > </tr> HTML;</pre>

→ 9.3. STRING CONCATENATION (SQ03) V1.0

Strings should be concatenated using **periods (".")** without whitespace on either side, unless the string leading or trailing the period is on a different line.

Valid: period used for concatenation	Invalid: whitespace around the period
<code>\$string3.=.\$string1.string2;</code>	<code>\$string3.=.\$string1. .string2;</code>
Valid: period trails a newline	Invalid: whitespace following the period
<code>\$string3.=.\$string1 > > ...string2;</code>	<code>\$string3.=.\$string1 > >string2;</code>

10. WHITESPACE

→ 10.1. WHITESPACE AT END OF LINE (WS01) V1.0

Unrequired whitespace should not appear at the end of any line. Blank lines should also not contain any whitespace. Please ensure your text editor is able to display whitespace so you are able to identify any unrequired tabs or spaces present within the file.

Valid: no unrequired whitespace	Invalid: whitespace at end of line
<pre>if.(\$foo).{ > echo.'valid'; } //.this.is.a.comment if.(\$bar).{ > echo.'valid'; }</pre>	<pre>if.(\$foo).{ > echo.'invalid'; } > //.this.is.a.comment. if.(\$bar).{ > echo.'invalid'; }</pre>

→ 10.2. WHITESPACE FOLLOWING END OF FILE (WS03) V1.0

No whitespace should appear after the closing PHP tag at the end of a file. The closing tag should end at the last character of the last line of the file unless the following lines are HTML.

```
47....
48.//.this.is.the.bottom.of.the.file
49.return.$foo;
50.??>
```

The following example shows a PHP file with HTML following the closing PHP tag. In this case, the closing PHP tag does not have to be on the last line.

```
47.> > ...
48.> > //.this.is.the.bottom.of.the.file
49.> > return.$foo;
50.> > ?>
51.> </body>
52.</html>
```

→ 10.2. WHITESPACE FOLLOWING END OF FILE (WS03) V1.0 (CONTINUED)

Notice that that the last line in the next example is blank. Here, the closing PHP tag should be on the last line instead of the second last.

```
47....  
48.//.this.is.the.bottom.of.the.file  
49.return.$foo;  
50.?.>  
51.
```

→ 10.3. WHITESPACE PROCEEDING START OF FILE (WS02) V1.0

No whitespace should appear before the opening PHP tag at the start of a file. The PHP tag should start at the first character of the first line of the file unless the proceeding lines are HTML.

```
1.<?php  
2.//.this.is.the.top.of.the.file  
3.if.($foo)....
```

The following example shows a PHP file with HTML preceding the opening PHP tag. In this case, the opening PHP tag does not have to be on the first line.

```
1.<html>  
2.> <head></head>  
3.> <body>  
4.> > <?php  
5.> > //.this.is.also.acceptable  
6.> > if.($foo)....
```

Notice that that the first line in the next example is blank. Here, the opening PHP tag should be on the first line instead of the second.

```
1.  
2.<?php  
3.//.here,.the.PHP.tag.is.not.on.the.first.line  
4.if.($foo)....
```

→ 10.4. FUNCTION AND METHOD SPACING (WS04) V1.0

Each function or class method should have two blank lines above the function comment and two blank lines below the function definition.

```
class .test
{
>
>
>  /**
>   * .Constructor
>   */
>  function.test()
>  {
>  >   ...
>
>  } //end.constructor
>
>
>  /**
>   * .Comment.for.function.one
>   */
>  function.one()
>  {
>  >   ...
>
>  } //end.one()
>
} //end.class
```

Each function should also have one blank lines above the funtion closing bracket.

```
function.one()
{
>   ...
>   One.blank.line.here
} //end.one()
```

11. NAMING CONVENTIONS

→ 11.1. NAMING CLASS METHODS (NC01) V1.0

Methods should be named using the "studly caps" style (also referred to as "bumpy case" or "camel caps"). The initial letter of the name is lowercase, and each letter that starts a new "word" is capitalised. Leading underscores are allowed for private functions but not anywhere else within the method name.

Acceptable method names	Unacceptable method names
<pre>getAsset() getAssetLinks() _privateFunction()</pre>	<pre>get_asset() getassetlinks()</pre>

→ 11.2. PHP KEYWORDS (NC02) V1.0

PHP Keywords should be written in upper case only.

Valid: TRUE	Invalid: true
<pre>\$foo = TRUE;</pre>	<pre>\$foo = true;</pre>
Valid: FALSE	Invalid: false
<pre>\$foo = FALSE;</pre>	<pre>\$foo = false;</pre>
Valid: NULL	Invalid: null
<pre>\$foo = NULL;</pre>	<pre>\$foo = null;</pre>

12. DATABASE QUERIES

→ 12.1. ASSERTING VALID DB RESULTS (MY02) V1.0

After retrieving a result from the database, the result must be checked for error before being used. A global assertion function, **`assert_valid_db_result()`**, is provided for this purpose. All database results must be passed to this function before being used. If the database result is not valid, this function will print the error and stop script execution.

```
$result.=.$db->getOne($sql);
assert_valid_db_result($result);

$result.=.$GLOBALS['SQ_SYSTEM']->db->getOne($sql);
assert_valid_db_result($result);
```

→ 12.2. DATABASE TRANSACTIONS (MY03) V1.1

When executing an INSERT, UPDATE or DELETE database query, the statement issuing the query must be wrapped in a database transaction by issuing a BEGIN statement before the query and a COMMIT or ROLLBACK statement after it. The transaction must never be left open.

```
$GLOBALS['SQ_SYSTEM']->doTransaction('BEGIN');

$sql.='UPDATE
> > > sq_lock
> > SET
> > > expires.='db_extras_todate($db,$date).'
```

13. MIXING PHP AND HTML

→ 13.1. PHP CODE TAGS (PH01) V1.0

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.

Valid: var output with standard echo function

Invalid: shorthand notation used

```
<td><?php echo $var; ?></td>
```

```
<td><?=$var ?></td>
```

14. COMMENTING

→ 14.1. INLINE COMMENTS (CM01) V1.0

There should be single space before the start of inline comment.

Valid: correct spacing	Invalid: missing a space
<code>// Comment.starts.here</code>	<code>// Comment.starts.here</code>

→ 14.2. FUNCTION COMMENTS (CM02) V1.0

Function comments must follow the format shown below.

```
/**
 * .This is a short description.....//.Compulsory
 *
 * .Optional longer description which can wrap....//.Optional
 * .across multiple lines if necessary. .This
 * .long description should respect the max
 * .chars per line standard
 *
 * .@param.string>   $a>   desc.....//.Compulsory for each function parameter
 * .@param.object>  $b>   desc
 *
 * .@return.mixed.void|object|int.....//.Compulsory
 * .@access.public.....//.Compulsory
 * .@see.variable.....//.Optional
 */
function.foo($a,.$b)
{
>   ...
}
```

→ 14.2. FUNCTION COMMENTS (CM02) V1.0 (CONTINUED)

Function descriptions Each function comment must have a short description. Optionally, it can have a longer multi-line description if the function is complex enough to require it.

```
/**
 * .This is a short description.....//.Compulsory
 *
 * .Optional longer description which can wrap....//.Optional
 * .across multiple lines if necessary. .This
 * .long description should respect the max
 * .chars per line standard
```

@param Tags Each argument must have its own @param tag. Indentation must be done carefully, with a single space before the @param tag and argument type. All other indentation should use tabs.

```
*.@param.string>   $a>   single.line.description
*.@param.object>  $b>   longer.description.that.spans.across.multiple
*>   >   >   >   >   >   lines
*....
function.foo($a,.$b){
```

@return Tag Every function must have its' return type specified. Only specific return types are allowed, which include [int|string|boolean|array|object|void|null]. The 'mixed' keyword can also be used to list multiple return types using '|' as a separator.

```
*.@return.array
*.@return.mixed.object|int
```

@access Tag Specify the access control for the function. It can have one of the following values:

```
*.@access.public
*.@access.private
*.@access.protected
```

@see Tags This tag(s) is optional, but is useful for other developers reading your code. It can have one of the following values:

```
*.@see.Class::function()
*.@see.Class
*.@see.function()
*.@see.variable
```

→ 14.2. FUNCTION COMMENTS (CM02) V1.0 (CONTINUED)

@static Tags Use the @static tag to declare a method or class to be static. Static elements can be called without reference to an instantiated class variable, as in class::method().

```
*.@static
```

HTML Element HTML tags inside function comment should use only lower cases to be compatible with XHTML standard.

Valid: lower case tag

Invalid: upper case tag

```
<pre></pre>
```

```
<PRE></PRE>
```

→ 14.3. DEBUG FUNCTIONS (MY01) V1.0

No MySource Matrix debug functions (such as bam, minibam, speed_check etc) should be left in committed PHP files.

```
bam('Got here');//.Debug.function.call.should.be.removed
```

No commented out code should be left in committed PHP files. Our code versioning system keeps track of all the code that is removed, so you don't have to keep it around.

```
foreach($items as $item){  
> //.if(is_null($item)).continue;  
> ...  
}
```