Coding Guidelines

Talemul Islam
16 September, 2011

Conventional

Table of Contents

1. Avoidance of Direct Constants 4
2. Avoidance of Long Function/Method/Procedure 4
3. Documentation 4
4. Structured Code 6
5. Error/Exception Handling 7
6. Initialize and Exit 7
7. Logging 8
8. Source Safe Maintenance 8
9. General Rules for Functions 8
10. Use of Whitespace 9
11. Naming Conventions 10
12. Declarations 10
13. Code Commenting 10
14. ISMP Service Development 11

Version	Version date
1	16 September, 2011
1.1	28 October , 2014

1. Avoidance of Direct Constants

Avoid uses of direct constants. For example instead of using
a=10;
use
#define INIT_VALUE 10
a=INIT_VALUE;
This will enable easy change of the INIT_VALUE in future

2. Avoidance of Long Function/Method/Procedure

- Avoid using long functions. Maximum length of functions should be kept at such a size so that it can be reviewed easily.
- Group lines of codes to a function or method or procedure rather than keeping them in a continuous code flow. This practice will enable reusability and manageability.

3. Documentation

Parameters, Return Values and Purpose of a function should be documented inside the codes. Any default value should be documented. This documentation can be avoided or minimized if the names are self-explanatory.

A commit messages following the rules:

[TASK] Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the two together.

Write your commit message in the present tense: "Fix bug" and not "Fixed bug." This convention matches up with commit messages generated by commands like git merge and git revert.

Code snippets::

should be written in Re Structured Text compatible format for better highlighting

Further paragraphs come after blank lines.

* Bullet points are okay, too
* An asterisk is used for the bullet, it can be preceded by a single space. This format is rendered correctly by Forge (red mine)
* Use a hanging indent

Resolves: #123
Resolves: #456
Related: #789
Releases: master, 1.1, 1.0

Examples of good and bad subject lines:

Introduce xyz service	// BAD, missing code prefix
[BUGFIX] Fixed bug xyz	// BAD, subject should be written in present tense
[WIP][!!!][TASK] A breaking change	// BAD, subject has to start with [!!!] for breaking changes
[BUGFIX] Session Manager removes expired sessions	// GOOD, the line explains what the change does, not what the bug is about (this should be explained in the following lines and in the related bug tracker ticket)

Standard documentation block:

```
/**
 * First sentence is short description. Then you can write more, just as you like
 *
 * Here may follow some detailed description about what the class is for.
 *
```

```
 * Paragraphs are separated by an empty line.
 */
class SomeClass {
 ...
}
```

Standard variable documentation block:

```
/**
 * A short description, very much recommended
 *
 * @var string
 */
protected $title = 'Untitled';
```

Standard method documentation block:

```
/**
 * A description for this method
 * * Paragraphs are separated by an empty line.
 *
 * @param \TYPO3\Blog\Domain\Model\Post $post A post
 * @param string $someString This parameter should contain some string
 * @return void
 */
public function addStringToPost(\TYPO3\Blog\Domain\Model\Post $post, $someString) {
 ...
}
```

Standard testcase documentation block:

```
/**
 * @test
 */
public function fooReturnsBarForQuux() {
 ...
}
```

4. Structured Code

- Maintain structured code – every function and code should strictly maintain Declaration and Body. Declaration of variables throughout the code should be avoided all the time.
- DO NOT connect to multiple DB's in a single php. This will ensure low coupling in the code.

- Common Libraries should be included from a central location. And ideally it should not be included in the distribution. Development will use common libraries from their site.

- All unused files should be deleted from the distribution.

- When passing many parameters to another url, use variables. For example, here we are using variables for few parameters to give the idea:
    1. $URL_PREFIX=' http://localhost/CMS/';
    2. $USERID = 'tareq';
    3. $USERPWD='1234';
    4. $url=$URL_PREFIX ."ContentList.php?Reftype=CategoryID&RefValue=".$RefValue."&NoOfItem=".$NoOfItem."&SortColumn=ContentID&SortType=ASC&ListFormat=".$ListFormat."&Channel=IVR&ServiceID=".$ServiceID."&UserID=."$USERID."&Password.$USERPWD."&LogType=preview&RequestSourceID=".$RequestSourceID;

5. Error/Exception Handling

There should be single function to handle all the errors/exception. And whenever errors/exception is occurred it should be passed to that handler only.

6. Initialize and Exit

- There should be one single function/block (that may call other functions) to initialize the application. And that function should take calling source as an input.

- There should be one single function/block (that may call other functions) to exit from the application. And that function should take calling source as an input.

- Whenever something is opened it must be closed. It can be file, connection socket anything.

7. Logging

There should be one single function to write/print logs or notifications. And that function should take calling source as an input.

8. Source Safe Maintenance

- Whenever you are checking in a code to source safe proper comments should be provided.
- Always keep restore points while working and check in the stable versions to the source safe frequently so that you can fall back easily without losing much of works.
- Consider sharing violations and multiuser scenarios seriously
- Always use locks while accessing same variables from multiple threads/processes

9. General Rules for Functions

- Avoid writing codes like
  for(i=0; i<strlen(str);i++) {Code Block}
  it is better to write
  n=strlen(str); for(i=0;i<n;i++) {Code Block}
  This will be time efficient.
- Avoid writing codes like X=a*b/function1(x)*function2(y);
  it is always better to write
  V1=function1(x);
  V2=function2(y);
  X=a*b/V1*V2;
  This will make it easy to debug in case of problems.
- Avoid multiple function call in same line.
- Always handle function return values unless you are sure that you don't need them.
- Always use parameters to control application logics and try to read the parameters from a place outside your application.
- Always keep a provision so that runtime parameters can be changed.
- Be extra cautious while a bulk portion or function is being copied and minor changes are made to create a new procedure or function. Most likely you are introducing extra overhead to change and there is a better way to do it. The possible ways to avoid are – use default values in parameters and add extra parameters with default values, add a wrapper function, modify or extend the function to accommodate the new requirements.

Examples of good and bad comparisons:

```
if ($template)            // BAD
if (isset($template))     // GOOD
if ($template !== NULL))  // GOOD
if ($template !== ''))    // GOOD

if (strlen($template) > 0) // BAD! strlen("-1") is greater than 0
if (is_string($template) && strlen($template) > 0) // BETTER

if ($foo == $bar)         // BAD, avoid truthy comparisons
if ($foo != $bar)         // BAD, avoid falsy comparisons
if ($foo === $bar))       // GOOD
if ($foo !== $bar))       // GOOD
```

10. Use of Whitespace

- Maintain indentation in coding. Use paragraph (blank lines) for code structuring. It makes the code readable.
- Use vertical and horizontal whitespace generously. Indentation and spacing should reflect the block structure of the code. A long string of conditional operators should be split onto separate lines. For example:

```
if (foo->next==NULL && number < limit && limit <=SIZE &&
node_active(this_input)) {...
```

might be better as:

```
if (foo->next == NULL
    && number < limit && limit <= SIZE
    && node_active(this_input))
{
...
```

- Similarly, elaborate **for loops** should be split onto different lines:

```
for (curr = *varp, trail = varp;
    curr != NULL;
```

```
        trail = &(curr->next), curr = curr->next )
    {
    ...
```

- Other complex expressions, such as those using the ternary **?:** operator, are best split on to several lines, too.

```
z = (x == y)
    ? n + f(x)
    : f(y) – n;
```

11. Naming Conventions

- Names with leading and trailing underscores are reserved for system purposes and should not be used for any user-created names. Convention dictates that:

#define constants should be in all CAPS.

enum constants are Capitalized or in all CAPS

- Function, typedef, and variable names, as well as struct, union, and enum tag names should be in lower case.

12. Declarations

- All external data declaration should be preceded by the **extern** keyword.
- The "pointer" qualifier, '*', should be with the variable name rather than with the type.

char *s, *t, *u; instead of char* s, t, u;

The latter statement is not wrong, but is probably not what is desired since 't' and 'u' do not get declared as pointers.

13. Code Commenting

Do not keep commented code. Use comments for adding notes only.

14. ISMP Service Development

1. Do not directly invoke SelectValue or similar php from visio call flow. Use a meaningful wrapper instead.
2. DO NOT pass ano, bno, mid, channel etc. from visio to php services. Php can get this by Request parameters. Also DO NOT assign values to these parameter in the php code. The values will always be fixed with the visio environment values.
3. The name of the visio config file should be config.txt and it should be kept in the same folder as in the visio file.
4. Use storedata shape in visio to avoid inserting directly into the database.
5. There should be only one single configuration file for each service
6. The services should generate some sort of log or meaningful error messages. Errors should be handled from a central place using a central function e.g. raiseError($errorcode) and then that function should raise all errors.
7. Error Reporting Level should be included in the configuration file. During development it can be all error but before packaging it should be set to no error reporting e.g. error_reporting(0)
8. For all the services test with mssql_connect and odbc_connect as DBType for all the services from development itself.

15. Style Guideline

Indentation

Use tabs (and configure your IDE to show a size of 8 spaces for them) for writing your code (hopefully we can keep this consistent). If you are modifying someone else's code, try to keep the coding style similar.

Since we are using 8-space tabs, you might want to consider the Linus Torvalds trick to reduce code nesting. Many times in a loop, you will find yourself doing a test, and if the test is true, you will nest. Many times this can be changed. Example:

```
for (i = 0; i < 10; i++) {
   if (something (i)) {
      do_more ();
   }
}
```

This takes precious space, instead write it like this:

```
for (i = 0; i < 10; i++) {
   if (!something (i))
      continue;
```

```
    do_more ();
}
```

Switch statements have the case at the same indentation as the switch:

```
switch (x) {
case 'a':
    ...
case 'b':
    ...
}
```

Performance and Readability

It is more important to be correct than to be fast.

It is more important to be maintainable than to be fast.

Fast code that is difficult to maintain is likely going to be looked down upon.

Where to put spaces

Use a space before an opening parenthesis when calling functions, or indexing, like this:

```
method (a);
b [10];
```

Do not put a space after the opening parenthesis and the closing one, ie:

good:

```
method (a);
array [10];
```

bad:

```
method ( a );
array[ 10 ];
```

Do not put a space between the generic types, ie:

good:

```
var list = new List<int> ();
```

bad:

```
var list = new List <int> ();
```

Where to put braces

Inside a code block, put the opening brace on the same line as the statement:

good:

```
if (a) {
   code ();
   code ();
}
```

bad:

```
if (a)
{
   code ();
   code ();
}
```

Avoid using unnecessary open/close braces, vertical space is usually limited:

good:

```
if (a)
   code ();
```

bad:

```
if (a) {
   code ();
}
```

Unless there are either multiple hierarchical conditions being used or that the condition cannot fit into a single line.

good:

```
if (a) {
   if (b)
      code ();
}
```

bad:

```
if (a)
```

```
   if (b)
      code ();
```

When defining a method, use the C style for brace placement, that means, use a new line for the brace, like this:

good:

```
void Method ()
{
}
```

bad:

```
void Method () {
}
```

Properties and indexers are an exception, keep the brace on the same line as the property declaration.

Rationale: this makes it visually simple to distinguish them.

good:

```
int Property {
   get {
      return value;
   }
}
```

bad:

```
int Property
{
   get {
      return value;
   }
}
```

Notice how the accessor “get” also keeps its brace on the same line.

For very small properties, you can compress things:

ok:

```
int Property {
```

```
   get { return value; }
   set { x = value; }
}
```

Empty methods: They should have the body of code using two lines, in consistency with the rest:

good:

```
void EmptyMethod ()
{
}
```

bad:

```
void EmptyMethod () {}
void EmptyMethod () {
}
void EmptyMethod ()
{}
```

If statements with else clauses are formatted like this:

good:

```
if (dingus) {
	...
} else {
	...
}
```

bad:

```
if (dingus)
{
	...
}
else
{
	...
}
```

bad:

```
if (dingus) {
	...
```

```
}
else {
        ...
}
```

Classes and namespaces go like if statements, differently than methods:

good:

```
namespace N {
   class X {
       ...
   }
}
```

bad:

```
namespace N
{
   class X
   {
       ...
   }
}
```

So, to summarize:

Statement	Brace position
Namespace	same line
Type	same line
Method (including ctor)	**new line**
Properties	same line
Control blocks (if, for…)	same line
Anonymous types and methods	same line

Multiline Parameters

When you need to write down parameters in multiple lines, indent the parameters to be below the previous line parameters, like this:

Good:

```
WriteLine (format, foo,
        bar, baz);
```

If you do not want to have parameters in the same line as the method invocation because you ar running out of space, you can indent the parameters in the next line, like this:

Good:

```
WriteLine (
      format, moved, too, long);
```

Comma separators go at the end, like a good book, never at the beginning:

Good:

```
WriteLine (foo,
        bar,
        baz);
```

Atrocious:

```
WriteLine (foo
        , bar
        , baz);
```

Use whitespace for clarity

Use white space in expressions liberally, except in the presence of parenthesis:

good:

```
if (a + 5 > method (blah () + 4))
```

bad:

```
if (a+5>method(blah()+4))
```