eZ coding standards & guidelines

Author:	Graham Brookins
Contributors:	Bertrand Dunogier (original version author), Patrick Allaert
Contact:	graham [at] se7enx [dot] com
Revision:	4
Date:	2024-12-05
Copyright:	7x
License	Creative Commons

These coding standards are the foundations to the eZ consistent, readable code. They must be read, known and followed by every eZ Publish developer & contributor, internal as well as external.

They are based on the eZ Components coding standards (http://ezcomponents.org/contributing/coding_standards) as well as on the work realized by Patrick Allaert.

Note that they are mostly based on the current eZ Publish version (4.4/4.5 as of writing), and therefore don't reflect everything we will use in the close or remote future. They will evolve as our methodologies are updated, for instance when we introduce namespaces.

Table of Contents

Files

Naming

Files MUST be named in lower-case, and MUST NOT use any separators (dashes, spaces, underscores). PHP files MUST use the `.php` extension Template files MUST use the `.tpl` extension.

Indenting

Indenting MUST use 4 spaces. Real tab characters MUST NOT be used.

Directory structure

lib

To be used for library items. Library items don't use the database or filesystem. They can implement tools that provide a way to access a storage system (database or filesystem), but aren't related to specific content.

The following structure should be used:

```
lib/<libraryName>/classes/<libraryFile>.php
```

All new library files must be unit tested.
The following `@package` and `@subpackage` headers must be used:

```
* @package lib
* @subpackage libraryName
```

kernel/[private/]classes

The root level of this folder contains general kernel classes. Most of them will be eZPersistentObject implementations. Other eZ Publish, business layer related files will go there.

`kernel/private/classes` should be used for items that aren't APIU stable yet, and may therefore suffer API changes in the near future.

The following structure should be used:

```
kernel/[private/]classes/<classNameInLowerCase>.php
```

The following `@package` header must be used:

```
* @package kernel
```

kernel/classes/datatypes

This is where the default system datatypes are to be placed.

The following structure must be used:
`kernel/classes/datatypes/<datatypename>/<datatypename>type.php`

Other classes & files used by the datatype should be placed in the same folder. This for example applies to persistent object classes used by the datatype. Common libraries that could be re-used should be placed in `lib/`.

The following `@package` and `@subpackage` headers must be used:

```
@package kernel
@subpackage datatypes
```

kernel/classes/workflowtypes

This is where the default system workflow events are to be placed.

The following structure must be used:
`kernel/classes/workflowtypes/event/<eventName>/<eventName>type.php`

kernel/*

This is the structure for eZ Publish modules. The following structure has to be used:

```
kernel/<moduleName>/module.php // views definition
kernel/<moduleName>/function_definition.php // fetch functions
definition
kernel/<moduleName>/ez<moduleName>operationcollection.php // module
operations, if applicable
kernel/<moduleName>/ez<moduleName>functioncollection.php // fetch
functions implementations
```

The following `@package` and `@subpackage` headers must be used:

```
@package kernel
@subpackage <moduleName>
```

kernel/[private/]classes/clusterfilehandlers & kernel

This is where cluster file handlers are located.

@package and @subpackage:
@package kernel
@subpackage clustering

kernel/classes/notification/event
Notification events handlers.

@package and @subpackage:

```
@package kernel
@subpackage notification
```

kernel/classes/collaborationhandlers
Collaboration handlers.

@package and @subpackage:

```
@package kernel
@subpackage collaboration
```

kernel/classes/vathandlers
VAT handlers (used by the ezprice datatype and the webshop).

@package and @subpackage:

```
@package kernel
@subpackage vat
```

Encoding
All text files must be encoded as UTF-8 (without byte-order-mark).

Note about BOM: Byte Order Mark is a specific unicode character used to signify the end of the file without specific metadata or explicit marker (like XML). As UTF-8 is by design ASCII back compatible and the BOM isn't, it isn't recommended with this character set. More details on http://en.wikipedia.org/wiki/Byte_order_mark.

Headers
All files must start with the following header:
```
/**
 * File containing the eZAutoloadGenerator class.
 *
 * @copyright Copyright (C) 1999-2011 eZ Systems AS. All rights
reserved.
 * @license http://www.gnu.org/licenses/gpl-2.0.txt GNU General
Public License v2
 * @version //autogentag//
 * @package <Package>
 * @subpackage <SubPackage>
 */
```

`//autogentag//` will be automatically replaced by the build system with the matching, up-to-date strings.

Note that @license and @copyright are updated/sanitized during build:

- @license (.*) is replaced with “@license <correct license string>”
- @copyright Copyright (C) 1999-2011 is changed so that the second year is always up-to-date.

See the Documentation chapter for details about `@package` and `@subpackage`.

Code structure

This is a general example of what eZ code must be formatted like:

```
<?php
/**
 * File containing the ezpFormattingExample class.
 *
 * @copyright Copyright (C) 1999-2011 eZ Systems AS. All rights
reserved.
 * @license GNU General Public License v2.0
 * @version //autogentag//
 * @package Examples
 * @subpackage Formatting
 */

/**
 * Formatting example class
 *
 * This class shows how the most common code items are to be
 * formatted and documented.
 *
 * <code>
 * $codeFormatting = new ezpFormattingExample( true, ‘foobar’ );
 * // more code here
 * </code>
 *
 * @property string $x A dynamic read/write property
 * @property-read bool $isPropertlyFormatted
 *                 A dynamic read only property with a longer
 *                 documentation message
 * @property-write string $foo A dynamic write property
 * @package Examples
 * @subpackage Formatting
 */
class ezpFormattingExample
{
    /**
     * Constructs an ezpFormattingExample that is used in the
documentation

     * @param bool $parameterOne a boolean parameter
     * @param string $parameterTwo
     *        A string parameter that has a long documentation string
     *        that wouldn't fit on a single line
     * @throws ezpFormattingException if $parameterOne isn't valid
     */
    public function __construct( $parameterOne, $parameterTwo =
'value' )
    {
        if ( $parameterOne === '...' )
        {
            // do this
        }
        elseif ( $parameterOne === 'somestring' )
        {
            // do something else
        }
        else
        {
            throw new ezpFormattingExceptionb( $parameterOne );
        }
    }

    /**
     * Returns a boolean that indicates if the formatting is valid
     * @return bool
     */
    public function isValidFormatting()
    {
        return ( $conditionOne && $conditionTwo );
    }

    /**
     * Parses the array $array
     * @param array(int=>string) $array
     * @return string An example string
     */
    public function parseArray( $array )
    {
        if ( !empty( $array ) && isset( $array[3] ) )
        {
            return "This is a string";
        }
    }

    /**
     * The code
     *
     * @var string
     */
    public $code;
}
?>
```

Curly brackets: {}

Start and end brackets MUST BE placed on the same column:

```
function multiply( $x, $y )
{
    // function body
}

if ( $x === $y )
{
    ..
}
else if ( $x > $y )
```

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

class foo
{
    function foo( )
    {
        ..
    } // function foo( )
}
```

An extra comment reminding what opening item is matched MAY be added after a closing bracket when the opening matching one may not be easily viewed on one screen.

Parenthesis

A space MUST BE added after a start parenthesis and one space before an end parenthesis.

Example:

```
$foo = ezpFooProvider::factory(
    $config["bar"], $config["baz"], true
);
```

Commas

One space MUST BE placed after every comma.

Example:

```
$foo = ezpFooProvider::factory(
     $config["bar"], $config["baz"], true
);
```

square brackets

Square brackets ([]) MUST NOT have a space after the start bracket and before the end bracket.

Example:
```
$foo = ezpFooProvider::factory(
    $config["bar"], $config["baz"], true
);
```

Empty lines

One empty line SHOULD BE used to distinguish logical code entities. Variable names which is grouped should be indented to the same level.

RFC: There were some opposition (same motivation as for function arguments) to indent variables at the same level since it requires realigning when adding variables with a greater length. => bigger diffs => more conflicts => support more difficult.

Example (using temporary variables (cfr. below) here for the sake of the example):
```
$hostname = "localhost";
$port     = 3306;
$user     = "root";
$password = "pass";

$mysqli = new MySQLi( $hostname, $user, $password, 'db_name', $port
);
```

Empty lines should not only appear empty, but they MUST also be empty. All space characters MUST BE removed. Unnecessary trailing white-spaces MUST also be removed.

Extra parenthesis MAY be used when a logical expression may not make sense at first sight, as it may induce confusion for readers.

Strings

PHP overall provides three types of strings format: double quoted, single quoted, heredoc and nowdoc syntax. More informations about all three can be found in the official strings documentation.

Default: double quotes

By default, you SHOULD use double quoted strings:
```
"This is a simple string";
"This one is using expanded {$variable}!";
"SELECT * FROM table WHERE identifier = '{$identifier}' AND
language_id = {$languageId}";
```

Single quotes to avoid escaping

If your string contains double quotes, you MAY use single quoted strings to avoid escaping and thus maintain maximum readability:

```
'Identifiers like "foo", "bar" and "baz" are often used by
developers!';
```

Remember that variables are *not* expanded in single quoted strings.

heredoc for multiline strings

Rather than concatenating multiple strings, you SHOULD use the heredoc syntax to handle multiline strings:

```
if ( $foo )
{
  echo <<<'EOT'
Multiline string which will
expand $variable as well as convert
escaped characters like \t, \n, \r,...
EOT;
   }
```

Remember that variables as well as special characters (carriage return, line feed, tabulation...) are expanded/interpreted in heredoc code just like in double quote strings. Both single and double quotes can be used without any escaping.

Variables in strings

Variables in strings are encouraged and SHOULD be used as they are more readable than concatenated items. Curly brackets SHOULD be used around these variables to increase readability, especially around complex variables such as object properties. Remember that constants, class constants and methods can not be used this way:

```
// bad
"This is a bad string with an" .  $object->property . " in it";

// better
"This is a good string with an $object->property in it";

// best
"This is a good string with an {$object->property} in it";
```

Conclusion

Double quotes are overall preferred, even though single ones may be used. In any case, don't spend time changing quotes in existing code, as it won't make a huge performance difference.

Booleans and null

While PHP supports any case on the boolean types and the null type they MUST always be written in lowercase:

```
$isSet = true;
$canPrint = false;
$unset = null;
```

Boolean operators

While PHP supports both symbols and words based operators, the symbol based ones MUST be used:

```
if ( $condition1 && $condition2 )
{
    doFoo();
}
```

Control structures

These are examples of how control structures MUST be formatted.

Notice the space after the control structure name. This is different from functions, which DO NOT have a space after the function name.

PHP offers alternative syntaxes for these structures, but the alternatives SHOULD NOT be used.

if/else/elseif

Example:

```
// nested if/else statement
if ( $a === $b )
{
    ...
}
elseif ( $a === $c )
{
    ...
}
else
{
    ...
}
```

while loop

Example:

```
$i = 0;
while ( $i < 100 )
{
    ...
    $i++;
}
```

do...while loop

Example:

```
$i = 0;
do
{
    ...
    $i++;
}
while ( $i < 100 );
```

for loop

Example:

```
for ( $i = 0; $i < 100; $i++ )
{
    ...
}
```

Remember *not* to run code within the conditional (second) part of the for loop. This gets interpreted at each run:

```
// bad
for ( $i = 0; $i < count( $array ); $i++ )
{
    ...
}

// good
for ( $i = 0, $count = count( $array ); $i < $count; $i++ )
{
    ...
}
```

foreach loop

Example:

```
foreach ( $array as $value )
{
    ...
}
```

switch/case

Example:

```
switch ( $value )
{
    case 0:
        ...
        break;

    case 'text':
        ...
        break;

    default:
        ...
}
```

Documentation

Thorough documentation is required for every class, method and property. It is to be written using the PHPDoc format.

Headers

As explained in the Files chapter, every php file must start with a standard header that has a

@package tag, and possibly a @subpackage tag. These tags are used to group items belonging together in the automatically generated APIDocumentation.

They only allow letters, digits, _ and -.

Both tags can be page-level or class-level (RFC: link to the PHPDoc documentation page about these). A page level tag is located on top of the file. A class level tag is located right above the class definition. A file can contain both: if located at page-level, the tag applies to all the items in the file. If applied at class-leve, it applies to the class it is present for.

The subpackages list is a non exhaustive list, and suggestions for more are welcome.

@package

@package really represents a high-level package, and these should be limited in number. Currently accepted:

- @package kernel for anything in kernel/
- @package lib for anything in lib/
- @package <extensionName> for extensions

@subpackage

`@subpackage` is used to specify a second sorting level. It is useful for kernel & lib as both these folders are quite huge. Note that subpackages aren't cross-package, unlike what one would think. An item with `@package` Kernel and `@subpackage Datatypes` won't match another file with @package eZStarRating and `@subpackage Datatypes`.

Currently accepted @subpackage values:

- `@subpackage tests`
- `@subpackage datatypes`
- `@subpackage workflow`
- `@subpackage notification`
- `@subpackage clustering`
- `@subpackage content` for everything about the content engine that is not datatypes or workflows

Types

Type names should be written as described on this page: http://php.net/language.types. They are valid whenever a type is require (`@param`,`@var`, `@return`, `@property`, `@property-read`, `@property-write`)

Basically the allowed types are: - bool - int - float - string - array - object (use class name) - `resource` - `mixed` (SHOULD be avoided. If needed, use multiple return types as documented below) - `type1|type2|type3`

If the type is array, we must describe the requirements to the contents of the array. This is done by specifying the type for normal arrays or the expected key's and corresponding value type if it is a hash array.

Example of normal array of integers:
`array(int)`

Example of a hash array:
`array(string=>valueType)`

Default values are auto documented, never document the default unless it is not obvious what it does.

Classes

Required for all classes.

The following fields are required:

- A brief one line description of the class.
- An extensive description of the class. Use examples, embedded with <code></code>, unless it is obvious how to use the class (obvious to others, not you).
- `@package`

The following fields are optional:

- `@tutorial`, if there are relevant tutorials
- `@see`, if this class has related classes
- `@property`/`@property-read`/`@property-write` are used to document properties of normal and Option Class.
- `@subpackage`

Example:
```
/**
 * One line description of the class.
 *
 * Extensive documentation of the class. Feel free to use some
 * inline code. For example, the following code is "like this":
 * <code>
 * $archive = new ezcArchive( "/tmp/archive.tar.gz" );
 * $entry = $archive->getEntry();
 * print( "First entry in the archive: " . $entry->getPath() );
 * </code>
 *
 * Continue documentation.
 *
 * @see all_related_classes
 *
 * @package PackageName
 * @version //autogen//
 */
```

Properties

This describes a new way of documenting properties, something that phpDocumentor does not yet understand directly. However, our patched version does.

Properties are documented in the class' docblock, and not with the __set() and __get() methods. Documentation of properties goes as follows:

```
* @property        <type> $name  Description
* @property-read   <type> $name  Description
* @property-write  <type> $name  Description
```

Examples are:

```
* @property        string  $pass   password or null
* @property-read   int     $port   port, only values > 1024 are
allowed
* @property-write  array   $query  complete query string as an
associative array
```

They are to be used as follows:

- `@property`: properties that can be read from and written to,
- `@property-read`: read-only properties,
- `@property-write`: write-only properties.

Methods

Required for all methods and functions.

The following fields are required:

- A brief (one line) description of what the class does. We use the following wording conventions (snatched from the excellent Qt documentation)
    - First word of description should always be a verb.
    - "Constructs a/the" for all constructors
    - "Returns ...." for all functions returning something except if the returned value is not significant for the duty of the method.
- @throws, syntax: @throws ExceptionType if [your reason here], like:
  @throws ezcConfigurationIniFileNotWritable if the current location values cannot be used for storage.
- @param
- @return, but only if there is something returned from the method.

The following fields are optional:

- A longer description of what the function does. If natural, mention the various parameters. If used the description field must follow the brief description.
- @see, if there are other related methods/functions.

Example 1 (With object parameters):

```
/**
 * Returns the length of the line defined in the two dimensional
space
 * between $point1 and $point2.
 *
 * Example:
 * <code>
 * $point1 = new Point( 5, 10 );
 * $point2 = new Point( 15, 42 );
 *
 * $length = getLength( $point1, $point2 );
 * </code>
 *
 * @see getLength3D()
 *
 * @throws PointException if any of the points are imaginary.
 * @param Point $point1
 * @param Point $point2
 * @return int
 */
public function getLength2D( Point $point1, Point $point2 )
{
```

Note how the parameters are not documented since they are already mentioned in the description.

Example 2 (Same as above but with optional extra parameter and array arguments):
```
/**
 * Returns the length of the line defined in two dimensional space
 * between point1 and point2.
 *
 * @param array $point1     Format array( 'x' => int, 'y' => int )
 * @param array $point2     Format array( 'x' => int, 'y' => int )
 * @param int   $multiplier Multiplies the result by the given
factor.
 * @return int
 */
```

Note how the additional optional parameter is documented since it is not mentioned in the normal description. Of course in this case you could choose to mention it there instead.

Function parameter and return types

All parameters must be documented with at least their type and parameter name. A description is required unless the function/method description says what the parameter does or the parameter is *really* self explanatory. Do not assume something is self explanatory just because you know what it does at writing time.

All returns should be documented with at least their type. If it is not obvious what the return value/parameter does with the short description, it should be described in the long description of the method, a description must be written. Do not add long descriptions of obvious parameters/return types if they are explained in the description text. This is to avoid cluttering the documentation with obvious stuff.

Long parameter descriptions must be formatted this way:
```
/**
 * @param array(int) $somewhatLongerName
 *        A long description of myParameter and it doesn't fit
 *        with the 79 characters.
 */
```

Class variables

Class member variable should be documented like this, based on the types described above:

```
/**
 * Holds the properties of this class.
 *
 * @var array(string=>mixed)
 */
```

Private classes

If you are documenting a private class make sure to mark both the file and the class docblock with `@access private`. Documentation for these classes will not be generated for the end user documentation.

It is important that private classes are not exposed anywhere within the public classes.

phpDocumentor tags and required usage

@apichange

RFC: not determined yet. Would be interesting for the API, but needs rules.

@copyright

Mandatory in any pagelevel doc block.

Unless another copyright is explicitely required (it is the case for some extensions, the value must always be as follows:

```
* @copyright Copyright (C) 1999-2011 eZ Systems AS. All rights
reserved.
```

@deprecated

Must be used when an items is deprecated. Apply at the largest possible level: - method/property level if a method/property is deprecated - class level if a class is deprecated, but not the whole page (unlikely) - page level if the whole page is deprecated (more likely than class level)

The contents of the tag must give further indications of when an item was deprecated. The following format is to be used so that deprecations can easily be searched:
`@deprecated Deprecated since eZ Publish 4.5`

@license

Must be added to every page level documentation block with the following contents:

```
@license GNU General Public License v2.0
```

@version

Required in every file *and* class documentation. Use the following:

```
@version //autogentag//
```

Implementation guidelines

While this chapter is by no means a strict perimeter of what can and can't be used, a consitent implementation requires that the same problems are solved with the same solutions.

Exceptions

One class per error type. Every exception descends from the ezpException (RFC: write it !) class. The ezpBaseException class must not be used in eZ Publish.
Similar errors can be grouped in one abstract exception class:

```
ezpException
|
+ ezpFileException
| |
| + ezpFileNotWritableException ( $filename, $fileType )
| |
| + ezpFileNotReadableException ( $filename, $fileType )
| |
| + ezpFileCanNotCopyExecption ( $sourceName, $destinationName,
$fileType )
|
+ ezpDbException
  |
  + ezpDbNoConnectionException
```

Exceptions are thrown with only their parameters. The exception class is responsible for preparing and formatting the message.

RFC: ezpException, or ezpBaseException ? RFC: what about splException ? Use ?

Functions

Functions should be placed inside classes when possible in order to maximize the benefit of autoloading. This does *not* encourage class based programming. eZ publish is object oriented, not procedural.

The syntax of functions are showed in the snipped below:
```
/**
 * This is my function.
 * @param int $paramA My parameter A
 * @param string $paramB my parameter B
 * @return int The meaning of life
 */
function meaningOfLife( $paramA = 1, $paramB = 'hike' )
{
    return 42;
}
```

No functions SHOULD offer more than seven arguments.

When many arguments need to be passed to a function, you SHOULD split them over multiple lines. In this case, the first argument should appear on the next line with one additional level of indentation. Ending parentheses are aligned with the start of the related keyword:
```
$aResult = aClass::instance()
    ->function1(
        "First argument function",
        array(
            "Hello",
            "World"
        )
    )
    ->chainedFunction(
        "Hi!",
        embeddedCall()
    )
);
```

Arrays

PHP arrays are quite all purposes. They MAY serve as queues, stacks, sets, associative arrays, hashes, lists,...

It is important to use them accordingly to avoid suboptimal operations.

Plain arrays

Arrays are a collection of ordered values which can be iterated and accessed through numeric indices mostly known as keys.

The creation of such arrays are mostly done using the array() construct:
```
$numbers = array( "one", "two", "three" );
```

Adding elements SHOULD done with:
```
$numbers[] = "four";
```

This has the same effect as `array_push( $numbers, "four" );` but is way more readable.

Combining the data of two arrays is done with array_merge():
```
var_dump( array_merge(
    array( "a", "b", "c" ),
    array( "a", 1, 2, "1", "2" ) ) );

array(8) {
    [0]=> string(1) "a"
    [1]=> string(1) "b"
    [2]=> string(1) "c"
    [3]=> string(1) "a"
    [4]=> int(1)
    [5]=> int(2)
    [6]=> string(1) "1"
    [7]=> string(1) "2"
}
```

Common mistakes

Initializing arrays using square brackets

This syntax is counter productive:
```
$numbers = array();
$numbers[] = "one";
$numbers[] = "two";
$numbers[] = "three";
```

This is preferred:
```
$numbers = array( "one", "two", "three" );
```

Using in_array() / array_search() where a hash table is more appropriate

This has too high a complexity (0(n)):

```
$array = array ( "one", "two", "three" );
$isTwoInArray = in_array( "two", $array );
```

This is preferred:

```
$set = array ( "one" => true, "two" => true, "three" => true );
$isTwoInSet = isset( $set["two"] ); // O(1)
```

using array_key_exists() to check if a key has value

This has O(n) complexity:

```
$set = array( 1 => true, 2 => true, 2 => true, 2 => true, 3 => true,
3 => true );
if( array_key_exists( 2, $set ) )// O(n)
{
    return $set[2];
}
```

Use isset(), which is several times faster and more readable. This has O(1) complexity:

```
$set = array(
    1 => true, 2 => true, 2 => true,
);
if( isset( $set[2] ) )// O(1)
{
    return $set[2];
}
```

Importante note: isset() will return true if a key isset but contains the value null:

```
$set = array( 1 => null, 2 => null );
var_dump( isset( $set[1] );

=> bool( false )
```

If the array can contain the value null, use array_key_exists() instead, as isset() will return false if the value is null. Avoid setting array values to null if there is no explicit reason to do so.

using count() to see if array is empty or not instead of using empty()

count() should be used to actually *count* items in a list.

This is bad:

```
if( count( $set ) === 0 )
(..)
if ( count( $set ) > 0 )
```

Good, use empty() which is several times faster and more readable:
```
if( empty( $set ) )
(..)
if ( !empty( $set ) )
```

Associative arrays

Associative arrays (also called maps) are arrays for which the key has a meaning for the associated value:
```
$numbers = array( 0 => "zero", 5 => "five", 10 => "ten" );
$bases = array( "binary" => 2, "octal" => 8, "decimal" => 10,
"hexadecimal" => 16 );
```

Merging maps

Combining the data of two associative arrays (union) is done using the `+` array operator:
```
var_dump(
    array( "a" => 1, "b" => 2, "c" => 3 )
    + array( "a" => 2, "d" => 5 )
);

array(4) {
    ["a"]=> int(1)
    ["b"]=> int(2)
    ["c"]=> int(3)
    ["d"]=> int(5)
}
```

Value from the left operand will take precedence and won't be overriden. A good example of union is to apply default values to an associative array of parameters:
```
$params = array( "host" => "myhost", "pass" => "mypass" );
$params += array( "host" => "localhost", "port" => 3306,
    "user" => "root", "pass" => "" );

var_dump( $params );
array(4) {
    ["host"]=> string(6) "myhost"
    ["pass"]=> string(6) "mypass"
    ["port"]=> int(3306)
    ["user"]=> string(4) "root"
}
```

Common mistakes with maps

Using array_merge() on associative arrays

This is not efficient:

```
var_dump( array_merge(
    array( "a" => 2, "d" => 5 ),
    array( "a" => 1, "b" => 2, "c" => 3 )
);
```

output:

```
array(4) {
    ["a"]=> int(1)
    ["d"]=> int(5)
    ["b"]=> int(2)
    ["c"]=> int(3)
}
```

This has the same effect, but is several times faster, and ordering is more logical:

```
var_dump( array( "a" => 1, "b" => 2, "c" => 3) + array( "a" => 2, "d"
=> 5 ) );
```

output:

```
array(4) {
    ["a"]=> int(1)
    ["b"]=> int(2)
    ["c"]=> int(3)
    ["d"]=> int(5)
}
```

Stacks

Stacks are useful to implement a last in, first out (LIFO) container. You add (push) an element with:

```
$array[] = 'new element'; // or using: array_push()
```

and remove (pop) one with:

```
$element = array_pop( $array );
```

Note: Starting with PHP 5.3, SplStack <http://php.net/SplStack> is preferred (but can not be used in eZ Publish until we require 5.3):

```
$q = new SplStack();
```

```
$q[] = 1;
$q[] = 2;
$q[] = 3;

foreach ( $q as $elem )  {
    echo $elem."\n";
}
```

Prints:

```
3
2
1
```

Queues

Queues are useful to implement a first in, first out (FIFO) container. You add (push) an element with:

```
$array[] = 'new element'; // or using: array_push()
```

And remove one with:

```
$element = array_shift( $array );
```

Starting with PHP 5.3, SplQueue <http://php.net/SplQueue> is preferred (but can not be used in eZ Publish until we require 5.3):

```
$q = new SplQueue();

$q->enqueue( 1 );
$q->enqueue( 2 );
$q->enqueue( 3 );

var_dump(
    $q->dequeue(), // int(1)
    $q->dequeue()  // int(2)
);
```

Sets

Sets are useful to store values, without any particular order, and no repeated values. Common operations on sets are adding/removing elements, iterating over elements and checking about the presence of an element in the set.

Those operations are commonly achieved with PHP's array implementation and functions like:

array_search(), in_array() and array_unique(). However, those operations are O(n) which is *extremely expensive*.

Instead of storing values in the value part of PHP's array, it is possible to store them in the key part. This has the benefit that uniqueness of elements is automatic and that checking about the presence of an element or removing one is O(1):

```
// using a small piece of information != null in the value part
$set = array( 1 => true, 2 => true, 5 => true );

$set[10] = true; // adding 10 to the set
$set[5] = true; // adding 5 to the set (no duplicates inserted)
$isTwoPresent = isset( $set[2] ); // checking presence with isset
unset( $set[1] ); // removing 1 from the set
```

Important note: Because of the nature of PHP's array, the key can only store integers and strings and cannot handle values like booleans, float or objects unless they are converted to a string representation.

As of PHP 5.1, SplObjectStorage provides a mechanism that let you use it as a set of objects:

```
$s = new SplObjectStorage();

$o1 = new StdClass;
$o2 = new StdClass;
$o3 = new StdClass;

$s->attach($o1);
$s->attach($o2);

var_dump($s->contains($o1)); // true
var_dump($s->contains($o2)); // true
var_dump($s->contains($o3)); // false

$s->detach($o2);

var_dump($s->contains($o1)); // true
var_dump($s->contains($o2)); // false
var_dump($s->contains($o3)); // false
```

Best practices

Be strict

Be as strict and precise as possible when developing. Prefer `===` over `==`, when comparing strings, always use `===`.

Keep it compact

Avoid unnecessary if's:
```
if ( condition1 )
{
    if ( condition2 )
    {
        // ...
    }
}
```

could be simplified:
```
if ( condition1 && condition2 ) {
    // ...
}
```

Do not use the `@` operator

While it can be useful to hide an error we know might happen, it can have very nasty side effects:
```
$db = mysql_connect( ... );
if ( !$db )
{
    ...
}
```

While the test will ensure that no database calls are made if the connection fails, what happens if PHP isn't compiled with MySQL ? A silent fatal error will occur, and it will be hell to trace down.

Handling of PHP errors with exceptions can help catching these errors if required.

Avoid commented code

We have version control system to discover what has been removed, don't keep code by commenting it.

Blocks of commented code are confusing to those greping the code for class and function usage.

Avoid references

Lots of references can be found throughout legacy eZ Publish 3.x code. There were no alternatives at that time when writing OO code. PHP5 doesn't need them anymore in most situations.

Only use them if you actually *want* to pass parameter by reference (an array sorting callback, for instance), but limit their usage as much as possible.

Avoid ternary operator to return `true`/`false` or to achieve a cast

These are not required:

```
$value ? true : false;
$value ? 1 : 0;
```

Use this instead:

```
(bool) $value;
(int) $value;
```

Don't test when you can return !

This is totally useless:

```
if ( condition )
{
    return true;
}
else
{
    return false;
}
```

This makes much more sense:

```
return (bool)condition;
```

Limit usage of temporary variables. They don't always make code more readable

With temporary variables:

```
$lang = $db->escapeString( $lang );
$version = (int) $version;
```

```
$initialLanguage = $this->attribute( 'initial_language_id' );
$fields = "name, content_translation";
$table = "ezcontentobject_name";
$query= "SELECT $fields
         FROM $table
         WHERE contentobject_id = '$contentObjectID'
            AND content_version = '$version'
            AND ( content_translation = '$lang' OR language_id =
'$initialLanguage' )";
$result = $db->arrayQuery( $query );
$resCount = count( $result );
```

Without temporary variables:

```
$resCount = count(
    $db->arrayQuery(
        "SELECT name, content_translation
            FROM ezcontentobject_name
            WHERE contentobject_id = '$contentObjectID'
                AND content_version = '" . (int) $version . "'
                AND ( content_translation = '" . $db->escapeString(
$lang ) . "' OR
                    language_id = '" . $this->attribute(
'initial_language_id' ) . "' )"
                )
            );
```

The second example clearly indicates we are creating variable `$resCount`.

In the first pne, it's not obvious which variables comes out from the block. Extra searches on variables' name would be required to figure out the exact scope of each of them and in the case of $resCount not being needed anymore, removing it and its temporary variables would be a tedious task. Furthermore, even if PHP is using a copy-on-write mechanism, it helps keeping the memory usage low by avoiding GC work.

In order to increase readability, some temporary variables may be used in that way:

```
$version = (int) $version;
$langString = $db->escapeString( $lang );
$languageId = $this->attribute( 'initial_language_id' );

$resCount = count(
    $db->arrayQuery(
        "SELECT name, content_translation
            FROM ezcontentobject_name
            WHERE contentobject_id = '$contentObjectID'
                AND content_version = '$version'
                AND ( content_translation = '$langString' OR
                    language_id = '$languageId' )"
                )
            );
unset ( $langString, $languageId );
```

The most important part is to avoid as much as possible to create very large temporary variables. If they are required, they should be unset as soon as possible.

Use `return` with care

Usage of return can be debated over and over again. There are mainly two religions about this construct: return ASAP and one true return.
Since One true return is an archaic model, and it makes it harder to debug code, and degrades performance, return ASAP must be used:

```
if ( $condition )
{
  return false;
}
if ( $otherCondition )
{
  return true
}
foreach( $array as $item )
{
  if ( condition( $item ) )
  {
    return ‘somestring’;
  }
  $return[] = $item;
}

return $return;
```

Also remember that exceptions can and should be used instead of return and will usually give much better feedback regarding what is going on... the early return might in many cases very well be replaced with exceptions!

Limit dependencies

Dependencies make it harder to test and debug code. Pass objects as parameters or options to other objects and make them interact this way. The eZ Components with their TieIn approach are a very good example of loose coupling.

Do not use a class name while inside it

This is a big problem if you rename your class at some point, as you will have to update very occurence of the class name. It also doesn't bring much.

Examples:

```
class ezpFoo
{
    public function bar()
    {
        // bad
        ezpFoo::myStaticFunction();

        // good
        self::myStaticFunction();

        // bad
        return new ezpFoo();

        // good
        return new self;

        // bad
        eZDebug::writeDebug( "My message", "ezpFoo::bar" );

        // good
        eZDebug::writeDebug( "My message", __METHOD__ );
    }
}
```

Always have a debugger enabled while developing

They will highlight mistakes you would otherwise miss, and will help you debug them, improving your code quality and making maintenance easier.

Interactive debugging through it plus DBGp can also be a huge time saver and quality improvement.

eZ Systems uses & recommends xDebug, by Derick Rethans, although alternatives exist.

Recommended naming scheme

To ensure consistency throughout all components the following naming conventions should be used. If the names don't make much sense in a given context other names should be found. These recommendations can also be used as prefix/suffix, e.g. `$srcX`, `$srcY`.

- For file or directory paths use `path`.
- For filename without a path use `file`.
- For directory name without a path use `dir`.
- Use `load`, `save` for complete operations on the file system, for instance loading an entire INI file.
- Use `read`, `write` for partial operations of data stream, for instance storing 10 bytes to a file.
- Use `fetch`, `store` for remote operations, for instance fetching data from a web server or database.
- When adding elements the following naming should be used:
  - `add` when adding elements without a specific order;
  - `insert` when adding elements in a specific order, for instance at a given index or position;
  - `append` when adding elements to the end;
  - `prepend` when adding elements to the beginning.
- Use `create` for PHP object creation and `generate` for operations that generate text, code, SQL etc.
- Use `reset` for resetting elements in the object.
- Use `getInstance` to get an instance in f.e. a singleton pattern.
- When removing elements the following naming should be used:
  - Use `remove` when elements are no longer referenced but not actually deleted. For instance removing a file path from a list while still leaving the file on the file system.
  - Use `delete` when elements are no longer meant to exists. For instance unlinking a file from the file system or deleting a database record.
- Short names are advised when their context is very clear. An example is a `copy()``function in a File class which has a source and a destination, it is quite clear that in this context we are working with files and can use abbreviated forms, src and dest. ``copy($src, $dest)`. The order of source and destination is always source first.
- Some words are different in British English and American English. It is most common to use the American spelling and so all words should follow this. Some typical names are:

- initialize, finalize, color, grey

Specific elements

Class names

Classes are named using UpperCamelCase and are always prefixed with ezp which stands for eZ Publish. Do not use more than five words for your names and make sure they are not verbs but nouns.

RFC: what about eZ extensions prefix ? ezx ? What about autoloading then ?

All classes in one package should start with the same prefix, unless a class bundles multiple "main" classes into one package. Examples:

```
ezpContent
ezpContentCriteria
ezpContentClassCriteria
ezcContentDepthCriteria
```

Class names have the form:

```
"ezc"([A-Z][a-z]+)+
```

Other examples:

```
ezpLocation, ezpLanguageSwitcher, ezpRestClient
```

Exception classes

For exception classes we append "Exception" to the class name.

Abstract classes

Interfaces

The debate is still going on about how interfaces should be named:

- the ezComponents CS state that interfaces (and abstract classes + structs shouldn't be postfixed). It makes it harder to identify them.
- in our current code, we either use no prefix, or ezpNameInterface
- some say that interfaces should be named starting with a capital I, but this breaks our CS/Namespace
    - one argument is that it makes it easier to locate interfaces in an IDE with code search tools

- an alternative would be ezpIName, that doesn't break our CS/Namespace, and still helps locating interfaces in our classes
- the last alternative is to decide that all interfaces should end up with Interface

The same applies to abstract classes, and maybe structs.

Method names

Methods are named using lowerCamelCase and should not use more than three words. Methods which change an internal property should be named setXXX() and in addition the retrieval method must be named getXXX(). All methods must use a verb:
`printReport(), validateData(), publish(), isValid()`

Property names

Properties are named using lowerCamelCase and should not use more than two words. Properties must always use nouns and not verbs, except for booleans which start with is, has, etc. and form a question:
`$name, $path, $author`

Parameter names

Parameters are named using lowerCamelCase and should not use more than two words. Parameters must always use nouns and not verbs, the exception are booleans which start with is, has etc. and form a question:
`$name, $path, $isObject`

Constant names

Constant names should follow the UPPER_CASE_WORDS standard, where an underscore separates words.

Special functions

There are a couple of special functions in PHP, which you should not use parenthesis with. These functions are:

- `include`, `include_once`, `require`, `require_once`
- `print`, `echo`
- `instanceof`
- `break`, `continue`
- `clone`, `new`

Use them without parenthesis, like:
```
require_once 'file.php';
echo "foo\n";
clone $class;
break 2;
```

Prefer *`echo`* over `print`, and *`require_once`* over `include`, `include_once` and `require`.

Although none of the "inclusion" functions should be used at all in normal code due to autoloading.

Common patterns

Singletons

Note: it is being discussed whether we should or not use singleton and not dependency injection, as singleton makes testing *much* harder.

However, the following code should be used when implementing the singleton pattern:

```
/**
 * @param ezpExampleSingleton Instance
 */
static private $instance = null;

/**
 * Private constructor to prevent non-singleton use
 */
private function __construct()
{
}

/**
 * Returns an instance of the class ezpExampleSingleton
 *
 * @return ezpExampleSingleton Instance of ezpExampleSingleton
 */
public static function getInstance()
{
    if ( self::$instance === null )
    {
        self::$instance = new ezpExampleSingleton();
    }
    return self::$instance;
}
```