Contexte M2L

Normes de développement

Applications web écrites en PHP

Version : 1.0

Sommaire

Table des matières

1 Introduction

Ce document s'appuie sur différentes sources de règles de codage, en particulier du projet communautaire PEAR - "PHP Extension and Application Repository"[1] et du cadre Zend Framework[2] qui fournissent entre autres des règles de codage pour les scripts PHP. Le document s'inspire aussi des règles de codage issues d'autres langages tels que Java.

Les règles énoncées par le présent document comportent au minimum :

- une **description** concise de la règle,

Si nécessaire :

- des **compléments** par rapport à la description,
- des **exemples** illustrant la règle, et éventuellement des **exceptions,**
- une partie **intérêts** en regard des critères qualité**.**

NB : La version actuelle des règles de codage ne comporte pas de règles sur les notions de POO (classe, niveau d'accès, membres d'instance ou de classe, etc.).

1 http://pear.php.net/manual/fr/standards.php
2 http://framework.zend.com/manual/fr/coding-standard.html

2 Fichiers

Ce paragraphe a pour but de décrire l'organisation et la présentation des fichiers mis en jeu dans un site Web dynamique écrit en PHP.

2.1 Extension des fichiers

Description

Les fichiers PHP doivent obligatoirement se terminer par l'extension **.php** pour une question de sécurité. En procédant ainsi, il n'est pas possible de visualiser le source des fichiers PHP (qui contiennent peut-être des mots de passe), le serveur web les fait interpréter par PHP.
Les fichiers qui ne constituent pas des pages autonomes (des fichiers destinés à être inclus dans d'autres pages web) se terminent par l'extension **.inc.php**.
Les fichiers contenant uniquement des définitions de fonctions se terminent par l'extension **.lib.php**.
Un fichier contenant une classe se nommera **class.<nom de la classe>.inc.php**

2.2 Nom des fichiers

Description

Seuls les caractères alphanumériques, tirets bas et tirets demi-cadratin ("-") sont autorisés. Les espaces et les caractères spéciaux sont interdits.

2.3 Format des fichiers

Description

Tout fichier .php ou page .html doit :
- Etre stocké comme du texte ASCII
- Utiliser le jeu de caractères UTF-8
- Etre formaté Dos

Compléments

Le << formatage Dos >> signifie que les lignes doivent finir par les combinaisons de retour chariot / retour à la ligne (CRLF), contrairement au << formatage Unix >> qui attend uniquement un retour à la ligne (`LF`). Un retour à la ligne est représenté par l'ordinal `10`, l'octal `012` et l'hexa `0A`. Un retour chariot est représenté par l'ordinal `13`, l'octal `015` et l'hexa `0D`.

3 Préambule XML

Description

Les pages Web doivent se conformer à une des normes HTML ou XHTML.
Toute page Web devra donc débuter par la directive <!DOCTYPE précisant quelle norme est suivie.
Elles seront validées à l'aide du validateur en ligne http://validator.w3.org

Exemple

Pour exemple, voici l'en-tête d'un fichier XHTML 1.0 :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

3.1 Inclusion de scripts

L'inclusion de scripts peut être réalisée par plusieurs instructions prédéfinies en PHP : include, require, include_once, require_once. Toutes ont pour objectif de provoquer leur propre remplacement par le fichier spécifié, un peu comme les commandes de préprocesseur C `#include`.

Les instructions include et require sont identiques, hormis dans leur gestion des erreurs. include produit une alerte (warning) tandis que require génère une erreur fatale . En d'autres termes, lorsque le fichier à inclure n'existe pas, le script est interrompu. include ne se comporte pas de cette façon, et le script continuera son exécution.

La différence entre require et require_once (idem entre include et include_once) est qu'avec **require_once()**, on est assuré que le code du fichier inclus ne sera ajouté qu'une seule fois, évitant de ce fait les redéfinitions de variables ou de fonctions, génératrices d'alertes.

Description

Les inclusions seront réalisées à l'aide de l'instruction require_once.

Exemple

Script prem.inc.php

```
<?php
function uneFonction () {
      echo "Fonction définie dans prem.inc.php <br />";
}
?>
```

Script second.inc.php

```
<?php
require_once("prem.inc.php");
uneFonction();
echo "Pas de problème : uneFonction n'a pas été redéfinie 2 fois. Merci
require_once ! <br />";
?>
```

Script monscript.php

```
<?php
require_once(prem.inc.php);
require_once(second.inc.php);
echo "Tout va bien ! <br />";
?>
```

4 Présentation du code

4.1 Tag PHP

Description

Toujours utiliser `<?php ?>` pour délimiter du code PHP, et non la version abrégée `<? ?>`. C'est la méthode la plus portable pour inclure du code PHP sur les différents systèmes d'exploitation et les différentes configurations.

Intérêts

Portabilité

4.2 Séparation PHP/ HTML

Description

Les balises HTML doivent se situer au maximum dans les sections HTML et non incluses à l'intérieur du texte des messages de l'instruction d'affichage echo.

Exemples

Ne pas écrire

```
<?php
    echo "<select id=\"lstAnnee\" name=\"lstAnnee\">";
    $anCours = date("Y");
    for ( $an = $anCours - 5 ; $an <= $anCours + 5 ; $an++ ) {
        echo "<option value=\"" . $an ."\">" . $an . "</option>";
    }
    echo "</select>";
?>
</select>
```

Mais écrire

```
<select id="lstAnnee" name="lstAnnee">
<?php
    $anCours = date("Y");
    for ( $an = $anCours - 5 ; $an <= $anCours + 5 ; $an++ ) {
?>
       <option value="<?php echo $an ; ?>">echo $an; ?></option>
<?php
    }
?>
</select>
```

Intérêts :

Dans les sections HTML, l'éditeur de l'outil de développement applique la coloration syntaxique sur les balises et attributs HTML. Ceci accroît donc la lisibilité et la localisation des erreurs de syntaxe au niveau du langage HTML. Il est aussi plus aisé d'intervenir uniquement sur la présentation, sans effet de bord sur la partie dynamique.

Maintenabilité : lisibilité

Portabilité : indépendance

4.3 Caractères et lignes

Description

Chaque ligne doit comporter au plus une instruction.
Les caractères accentués ne doivent pas être utilisés dans le code source, excepté dans les commentaires et les messages texte.
Un fichier source ne devrait pas dépasser plus de **500 lignes**.

Exemples

Ne pas écrire

```
$i-- ; $j++ ;
```

Mais écrire

```
$i-- ;
$j++ ;
```

Intérêts

Maintenabilité : lisibilité et clarté du code.

4.4 Indentation et longueur des lignes

Description

Le pas d'indentation doit être **fixe** et correspondre à **4 caractères.** Ce pas d'indentation doit être paramétré dans l'éditeur de l'environnement de développement. L'indentation est stockée dans le fichier sous forme de 1 tabulation.

Il est recommandé que la longueur des lignes ne dépasse pas 100 caractères.
Lorsqu'une ligne d'instruction est trop longue, elle doit être coupée après une virgule ou avant un opérateur. On alignera la nouvelle ligne avec le début de l'expression de même niveau de la ligne précédente.

Exemples

Exemple 1 : découpage d'un appel de fonction

On découpe la ligne après une virgule :

```
$maVar = fonctionA(expressionLongue1, expressionLongue2,
                   fonctionB(expressionLongue3,
                             expressionLongue4));
```

Exemple 2 : découpage d'une expression arithmétique

La ligne est découpée avant un opérateur :

```
$maVar = expressionLongue1 * expressionLongue2
         + expressionLongue3 - expressionLongue4
```

Exemple 3 : découpage d'une expression conditionnelle

La ligne est découpée avant un opérateur :

```
if(((condition1 && condition2) || (condition3 && condition4))
        && !(condition5 && condition6)) {
    doSomething();
}
```

Intérêts

Maintenabilité : lisibilité.

4.5 Espacement dans les instructions

Description

1. Un mot-clé suivi d'une parenthèse ouvrante doit être séparé de celle-ci par un espace. Ce n'est pas le cas entre un identificateur de fonction et la parenthèse ouvrante.
2. Tous les opérateurs binaires, sauf l'opérateur « -> » doivent être séparés de leurs opérandes par un espace.
3. Les opérateurs unaires doivent être accolés à leur opérande.

Exemples :

1.
```
while (true) {
      ...
  }
```

2.
```
$totalTTC = $totalHT + ($totalHT * ($tauxTVA / 100));
$totalTTC = round($totalTTC, 2);
$existe = $unElt->hasAttribute();
```

3.
```
$nb = 0;
$fin = false;
while (!$fin) {

...  $nb++;
}
```

4.
```
for ($nbLignes = 1; $nbLignes < 4; $nbLignes++) {

}
```

Intérêts

Maintenabilité : lisibilité

4.6 Présentation des blocs logiques

Description

1. Chaque bloc logique doit être délimité par des accolades, même s'il ne comporte qu' une seule instruction (cf. exemple 1),
2. Dans une instruction avec bloc, l'accolade ouvrante doit se trouver sur la fin de la ligne de l'instruction ; l'accolade fermante doit débuter une ligne, et se situer au même niveau d'indentation que l'instruction dont elle ferme le bloc (cf. exemple 2),
3. Les instructions contenues dans un bloc ont un niveau supérieur d'indentation.

Exemples

Exemple 1 :

Ne pas écrire

```
if ($prime > 2000)
    $prime = 2000;
```

Mais écrire

```
if ($prime > 2000) {
    $prime = 2000;
}
```

Exemple 2 : écriture des instructions avec blocs :

Structures de contrôle conditionnelles

```
if (...) {
    ...
} elseif (...) {
    ...
} else {
    ...
}
switch (...) {
    case ... :
        ...
    case ... :
        ...
    default :
        ...
}
```

Définition de fonction

```
function uneFonction() {
    ...
}
```

Structures de contrôle itératives

```
for (...; ...; ...) {
    ...
}
while (...) {
    ...
}
do {
    …
}while (...);
```

Intérêts

La présence d'accolades ainsi que l'indentation facilitent la localisation des débuts et fins de blocs et réduit le risque d'erreur logique lors de l'ajout de nouvelles lignes de code.

Maintenabilité : lisibilité.

4.7 Appels de fonctions / méthodes

Description

Les fonctions doivent être appelées sans espace entre le nom de la fonction, la parenthèse ouvrante, et le premier paramètre ; avec un espace entre la virgule et chaque paramètre et aucun espace entre le dernier paramètre, la parenthèse fermante et le point virgule.
Il doit y avoir un espace de chaque côté du signe égal utilisé pour affecter la valeur de retour de la fonction à une variable. Dans le cas d'un bloc d'instructions similaires, des espaces supplémentaires peuvent être ajoutés pour améliorer la lisibilité.

Exemples

```
<?php
$total = round($total, 2);
?>

<?php
$courte         = abs($courte);
$longueVariable = abs($longueVariable);
?>
```

Intérêts

Maintenabilité : lisibilité

5 Documentations et commentaires

5.1 Introduction

La documentation est essentielle à la compréhension des fonctionnalités du code . Elle peut être intégrée directement au code source, tout en restant aisément extractible dans un format de sortie tel que HTML ou PDF. Cette intégration favorise la cohérence entre documentation et code source, facilite l'accès à la documentation, permet la distribution d'un code source auto-documenté. Elle rend donc plus aisée la maintenance du projet.

L'intégration de la documentation se fait à travers une extension des commentaires autorisés par le langage PHP. Nous utiliserons celle proposée par l'outil PHPDocumentor, dont les spécifications sont disponibles à l'URL http://www.phpdoc.org/.

Pour rappel, les commentaires autorisés par le langage PHP adoptent une syntaxe similaire aux langages C et C++ (`/* ... */` et `//`). Ils servent à décrire en langage naturel tout élément du code source.

L'extension de l'outil PHPDocumentor est la suivante `/** ... */`. Leur utilisation permettra de produire une documentation dans un ou plusieurs formats de sortie tels que HTML, XML, PDF ou CHM.

La syntaxe spécifique à l'outil PHPDocumentor sera utilisée au minimum pour les entêtes de fichiers source et les entêtes de fonctions. Des éléments sur l'installation, les spécifications et l'utilisation de l'outil PHPDocumentor sont fournis en annexe 1.

5.2 Entêtes de fichier source

Description

Chaque fichier qui contient du code PHP doit avoir un bloc d'entête en haut du fichier qui contient au minimum les balises phpDocumentor ci-dessous.

Compléments

Format d'entête de fichier source :

```
<?php
/**
 * Description courte des fonctionnalités du fichier

 * Description longue des fonctionnalités du fichier  si nécessaire
 * @author nom de l'auteur
 * @package default
 */
```

5.3 Entêtes de fonction

Description

Toute définition de fonction doit être précédée du bloc de documentation contenant au minimum :

- Une description de la fonction
- Tous les arguments
- Toutes les valeurs de retour possibles

Compléments

Format d'entête de fonction :

```
/**
 * Description courte de la fonction.

 * Description longue de la fonction (si besoin)
 * @param type nomParam1 description
 * ...
 * @param type nomParamn description
 * @return type description
 */
function uneFonction($nomParam1, ...) {
```

Exemple

```
/**
 * Fournit le compte utilisateur d'une adresse email.

 * Retourne le compte utilisateur (partie identifiant de la personne) de
 * l'adresse email $email, càd la partie de l'adresse située avant le
 * caractère @ rencontré dans la chaîne $email. Retourne l'adresse complète
 * si pas de @ dans $email.
 * @email string adresse email
 * @return string compte utilisateur
 */
function extraitCompteDeEmail ($email) {
  ...
}
```

5.4 Commentaires des instructions du code

Description

Il existe deux types de commentaires :
1. les commentaires mono ligne qui inactivent tout ce qui apparaît à la suite, sur la même ligne : //
2. les commentaires multi-lignes qui inactivent tout ce qui se trouve entre les deux délimiteurs, que ce soit sur une seule ligne ou sur plusieurs /* */

Il est important de ne réserver les commentaires multi-lignes qu'aux blocs utiles à PHPDocumentor et à l'inactivation de portions de code.
Les commentaires mono-ligne permettant de commenter le reste, à savoir, toute information de documentation interne relative aux lignes de code. Ceci afin d'éviter des erreurs de compilation dues aux imbrications des commentaires multi-lignes.

Exemples

1. Insertion d'un commentaire mono-ligne pour expliquer le comportement d'un code

Exemple 1 :

```
function extraitCompteDeEmail ($email) {
    // le traitement se fait en 2 temps : recherche de la position $pos dans
    // l'adresse de l'occurrence du caractère @, puis si @ présent,
    // extraction du morceau de chaîne du 1er caractère sur $pos caractères
    $pos = strpos($email, "@");
    if ( is_integer($pos) ) {
        $res = substr($email, 0, $pos);
    }
    else {
        $res = $email;
    }
    return $res;
}
```

Exemple 2 :

```
// page inaccessible si visiteur non connecté
if (!estVisiteurConnecte()) {
    header("Location: cSeConnecter.php");
}

// acquisition des données reçues par la méthode post
$mois = lireDonneePost("lstMois", "");
$etape = lireDonneePost("etape","");
```

2. Inactivation d'une portion de code pour débogage

```
/*
// page inaccessible si visiteur non connecté
if (!estVisiteurConnecte()) {
    header("Location: cSeConnecter.php");
}
*/
```

6 Nommage des identificateurs

Cette convention concerne les éléments suivants du langage :

- les fonctions,
- les paramètres formels de fonctions,
- les constantes,
- les variables globales à un script,
- les variables locales,
- les variables de session.

Pour l'ensemble de ces éléments, la clarté des identificateurs est conseillée. Le nom attribué aux différents éléments doit être aussi explicite que possible, c'est un gage de compréhension du code.

6.1 Nommage des fonctions

Description

L'identificateur d'une fonction est un verbe, ou groupe verbal.
Les noms de fonctions ne peuvent contenir que des caractères alphanumériques. Les tirets bas ("_") ne sont pas permis. Les nombres sont autorisés mais déconseillés.
Les noms de fonctions doivent toujours commencer avec une lettre en minuscule. Quand un nom de fonction est composé de plus d'un seul mot, la première lettre de chaque mot doit être mise en majuscule. C'est ce que l'on appelle communément la "notationCamel".

Exemples :
```
filtrerChaineBD(), verifierInfosConnexion(),estEntier()
```

Intérêts

Maintenabilité : lisibilité.

6.2 Nommage des variables et paramètres

Description

L'identificateur d'une variable ou paramètre indique le rôle joué dans le code ; c'est en général un nom, ou groupe nominal. Il faut éviter de faire jouer à une variable plusieurs rôles.
Les noms de variables et paramètres ne peuvent contenir que des caractères alphanumériques. Les tirets bas sont autorisés uniquement pour les membres privés d'une classe. Les nombres sont autorisés mais déconseillés.

Comme les identificateurs de fonctions, les noms de variables et paramètres adoptent la notation Camel.

Exemples :
```
$nomEtud, $login
```

Intérêts

Maintenabilité : lisibilité.

7 Algorithmique

7.1 Fonctions/méthodes

7.1.1 Nombre de paramètres des fonctions

Description

Les fonctions ne doivent pas comporter un trop grand nombre de paramètres. La limite de 5 à 6 paramètres est recommandée. Tout dépassement de cette limite doit être justifié.

Compléments

Cette règle s'applique, tout spécialement, dans le cadre de la programmation par objets qui permet justement de réduire le nombre de paramètres des fonctions.

Intérêts

Maintenabilité : lisibilité.

7.2 Instructions

7.2.1 Parenthésage des expressions

Description

Il est recommandé d'utiliser les parenthèses à chaque fois qu'une expression peut prêter à confusion.

Exemples

Il ne faut pas écrire ...

```
if ($nbLignes == 0 && $nbMots == 0)
```

...mais plutôt écrire

```
if (($nbLignes == 0) && ($nbMots == 0))
```

Intérêts

L'ajout de parenthèses dans les expressions comportant plusieurs opérateurs permet d'éviter des confusions sur leur priorité.

Maintenabilité : lisibilité.

7.2.2 Interdiction des instructions imbriquées

Description

Les instructions imbriquées doivent être évitées quand cela est possible. En particulier, les types d'instructions imbriquées suivantes sont à bannir :

- affectations multiples.

Compléments

Une expression ne doit donc contenir que :

- des variables,
- des constantes,
- des appels de fonctions dont les arguments ne sont pas eux-mêmes des éléments variables.

Exemples

Éviter les affectations dans les appels de fonctions :

```
uneFonction($nb = rand(10,20), $qte);
```

Éviter les affectations dans les expressions :

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

Éviter les affectations multiples :

```
$a = $b = $c = $i++;
```

Intérêts

La complexité des expressions peut donner lieu à des erreurs d'interprétation. Par exemple, l'affectation dans une condition peut être lue comme un test d'égalité.

Maintenabilité : lisibilité.

7.2.3 Limitation de l'utilisation des *break* et *continue* dans les itératives

Description

Utilisation modérée
Les ruptures de séquence `break` et `continue` doivent être utilisées avec modération dans les itératives.

Compléments

L'abus de ce type d'instructions peut rendre le code difficile à comprendre. Elles pourront toutefois être utilisées ponctuellement. Dans ce cas, un commentaire devra le signaler.

Intérêts

Limiter les instructions `break` et `continue` améliore la structuration du code. Ces instructions (qui sont des "goto" déguisés), lorsqu'elles sont utilisées fréquemment, peuvent en effet dénoter une mauvaise analyse des conditions d'itérations dans certains cas.

7.2.4 Écriture des *switch*

Description

1. Tout le contenu à l'intérieur de l'instruction "switch" doit être indenté avec 4 espaces. Le contenu sous chaque "case" doit être indenté avec encore 4 espaces supplémentaires.
2. Les structures `switch` doivent obligatoirement comporter une clause `default`.
3. Le niveau d'imbrication des `switch` ne doit pas dépasser 2.
4. Chaque cas ou groupe de cas doit se terminer normalement par une instruction break. Les cas ne se terminant par un saut **break** doivent spécifier un commentaire rappelant que l'exécution se poursuit.
5. L'instruction break est obligatoire à la fin du cas par défaut. Cela est redondant mais protège d'une erreur en cas de rajout d'autres cas par la suite.

Compléments

```
switch (choix) {
    case expression1 :
        instructions
        /* pas de break */
    case expression2 :
    case expression3 :
        instructions
        break;
    default :
        instructions;
        break;
}
```

Intérêts

Fiabilité : robustesse, clarté.

8 Gestion des formulaires HTML

8.1 Méthodes de soumission des formulaires

Les méthodes de soumission d'un formulaire sont au nombre de 2 : GET et POST. La première véhicule les noms et valeurs des champs dans l'URL de la requête HTTP, la seconde dans le corps de la requête HTTP.

Description

La méthode POST est à préférer pour des raisons de taille de données et de confidentialité. A noter que la confidentialité se résume ici à ne pas voir apparaître les noms et valeurs de champs dans la zone d'adresse du navigateur : les données sont, dans les 2 cas, transmises en clair sur le réseau dans le cas où le protocole applicatif utilisé reste HTTP.

Le choix de la méthode GET peut cependant se justifier s'il est souhaitable de pouvoir conserver les différentes soumissions d'un formulaire en favoris.