Coding Style

Camille Raymond Jerôme Audoux Justine Ranc
Yann Pravossoudovitch

29 mai 2011

Chapitre 1

Introduction

Le coding style est un ensemble de règles utilisées lorsque l'on écrit le code source d'un programme pour que celui-ci soit harmonisé et uniformisé. Ce coding style est fortement inspiré de celui utilisé dans le noyau Linux (`http ://lxr.linux.no/linux/Documentation/CodingStyle`). On ajoute à ça quelques règles d'utilisation du BugTracker, du Code Repository et quelques notions de base sur les Unit Tests.

Chapitre 2

Indentation

L'indentation est cruciale pour la compréhension du code, de ce fait, tout code doit être indenté. On augmente le niveau d'indentation après chaque ouverture d'accolade, et il revient au niveau inférieur après la fermeture.

Une indentation correspond à **4 espaces matérialisés par une tabulation**. 4 et rien d'autre. L'indentation est là pour clarifier le code et faciliter la lecture en le découpant en bloc. De plus, si vous trouvez que le code est trop indenté, c'est sûrement qu'il doit être modifié. **Voyez ça comme un aide mémoire, pas comme un handicap.**

Une ligne ne doit pas dépasser **80 caractères**. Si pour une raison ou pour une autre cette règle doit être brisée, elle ne doit pas entravée la lisibilité du code. Idéalement, la ligne doit être découpée en morceau et alignée avec la ligne du dessus.

On ne fait pas plusieurs déclarations sur la même ligne.

Exemple :

```
if (meh)  lol
x,y = 0 ;
```

...sont à bannir.

Chapitre 3

Accolades, parenthèses, espaces et sauts de ligne

L'ouverture d'une accolade se fait toujours à la ligne et doit être placée **au même niveau d'indentation que le mot clé qui l'appel**. Même chose pour les fonctions. Placer l'accolade sur la même ligne que le mot-clé est utile lorsqu'on visualise le code sur un terminal de 80*24 et qu'il est nécessaire de gagner des lignes. Aujourd'hui, les écrans sont un peu plus grand, donc on peut se permettre de sauter des lignes pour **clarifier le code**. Bien sûr, ce point de vue peut être discuté longuement.

Toute condition doit être parenthèsée. Pas d'espace après l'ouverture et avant la fermeture des parenthèses.

On mettra un espace après les mots-clés suivant :

`if, switch, case, for, do, while, return`

On placera des espaces autour des opérateurs suivant :

On sautera une ligne avant et après chaque mot-clé (ci-dessus), sauf si ils ouvrent ou ferment un autre bloc. Ainsi, on sépare clairement les blocs de code qui suivent et précèdent des boucles ou conditions.

Exemple :

```
lol
{
    meh
    {
        var
        var

        fleh
        {
            return meh
        }
    }
}
```

On fera attention à ce qu'il ne reste **pas d'espace en fin de ligne**.

Chapitre 4

Nommage

Le nom des fonctions devra être écrit en **CamelCase avec la première lettre en minuscule**. On n'utilisera pas de dash ni d'underscore. Ils doivent être clairs et compréhensibles et écrit en **Anglais**. Le nom doit être descriptif et permettre d'avoir une vue simple de l'utilité de ce qu'il nomme tout en restant court. De plus, le 1er mot doit être un verbe (en anglais donc) qui décrit ce que la fonction fait. Exemple :
isLoutre
getMeh

On utilisera : is* pour les tests qui retournent une valeur booléenne ou qui effectue un test d'appartenance.
get* ou retrieve* quand on va chercher des données ou pour des calculs simples.

Chapitre 5

Fonctions

Une fonction doit être **courte et lisible**. **Elle ne doit faire qu'une seule chose et doit le faire bien**. On mettra en place le principe DRY (Don't Repeat Yourself). Si on doit réécrire une même partie de code 2 fois, envisagez d'en faire une fonction. Si ce comportement doit changer, ce sera plus facile à maintenir.

Une fonction ne doit pas avoir plus de 10 variables locales, boucles ou conditions (!). Si il y en a plus, envisagez de réécrire la fonction.

Une fonction ne doit pas dépasser 80 lignes (!). Si il y en a plus, envisagez de réécrire la fonction.

Si il est vraiment nécessaire d'utiliser une variable globale, et qu'il n'y a vraiment **pas d'alternative**, on doit préciser qu'elle est globale à chaque utilisation. On doit aussi noté à sa déclaration quelles fonctions l'utilisent dans un commentaire multilignes.

La valeur de retour et comment l'utiliser doit être préciser dans le commentaire au dessus de la fonction.

Les mots-clés doivent être écrit en minuscule. Tout malloc doit faire référence à un free. Il serait agréable de connaître l'emplacement (relatif) de l'un par rapport à l'autre lors de leur utilisation.

Une fonction ne doit avoir qu'un seul "return". Préférez une variable pour éviter les sorties multiples.

Chapitre 6

Commentaires

Les commentaires sont nécessaires à la compréhension du code. De ce fait, **tout code doit être commenté**. On utilisera seulement les commentaires **multilignes** (/* */). * *Ce que fait la fonction, et pourquoi (fonction mère et comportement éventuellement) *Nom de la ou les fonction(s) de test et fichier dans laquelle elle se trouve * *@param, type, utilité - pour chaque paramètre *@return, type, format - si la fonction retourne quelque chose */

On utilisera TODO lorsqu'il est nécessaire d'accomplir un travail supplémentaire sur la fonction.

Exemple :
Le code est fonctionnel mais pourrait être réécrit si il y a des problèmes de performance.
Le code n'est pas complet, il manque un cas à traiter, etc...

Les commentaires peuvent être écrit en Français ou en Anglais. Et doivent être collés au dessus de ce qu'ils documentent.

Chapitre 7

Unit Tests

Toute fonction doit avoir au minimum 1 Unit Test associé à elle, directement, ou indirectement. On précisera dans le commentaire de la fonction par quel Unit Test elle est testée.

Le commentaire d'un Unit Test doit être comme suit :

```
/*
*Fonction(s) testées
*
*Comportement attendu
*/
```

Si vous trouvez qu'il est fastidieux de faire un Unit Test pour certaines fonctions "simples", ajoutez au commentaire d'en-tête son comportement attendu.

Chapitre 8

Versioning

Chaque modification d'un fichier que vous considérez comme une étape, doit être soumise au gestionnaire de version.
1 commit = 1 feature/action = seulement les fichiers concernés.

Quand un commit ferme un ticket du BugTracker, on utilisera *fix #no_ticket*. Si il est simplement adressé à ce ticket, on utilisera *ref #no_ticket*. Si il réouvre un ticket (bug trouvé et TODO complété), on utilisera *reopen #no_ticket*.

Pour clarifier, on utilisera ces codes :

#fix
Si la modification fix un problème signalé dans le bugtracker

#add
Si c'est un ajout de fonction

#patch
Si c'est un dirty fix ou si c'est un essai fix. Si il s'avère efficace, il faudra fermer le ticket à la main.

#rem
Si on supprime une fonction ou du code en quantité importante (code obsolète, deprecated ou plus utilisé)

#rfctr ou #refactor
Si on réécrit ou nettoie une fonction.

#draft
Si on commit du code pour montrer aux autres membres mais qu'il n'est pas fonctionnel ou qu'il est en pseudo-code (auquel cas il doit être entouré de commentaire multiligne)

#test
Si elle a besoin d'être testé.

Avant de commencer à coder et avant de pusher les changements, pensez à récupérer les dernières versions des fichiers et le cas échéants les merger aux votre.

Avant de pusher, il faut faire tourner les Unit Tests. Si ils ne peuvent être complété dans leur totalité, pensez à utiliser #draft pour préciser que c'est du code qui ne marche pas.