Coder proprement

Par Dimitri PIANETA

Préface

- Ce cours a été crée par rapport au livre de Robert C. Martin « coder proprement ».
- Et aux annexes de ce même livre

Sommaire

Partie 1

Code propre

Du code et du mauvais code

→**Programmer** : Préciser des exigences à un niveau de détail qui permet à un machine de les exécuter.

→Pour améliorer un mauvais code, il est recommandé d'utiliser les **design pattern**.

Coût total du désordre

→**Mal programmer coût cher en maintenance dans un projet ou dans un prochaine reprise de code**.

→Les sources doivent être claire et selon un modèle prévus dans l'entreprise pour une maintenance pour de nouveaux développeurs.

Partie 2

Noms significatifs

Etat de l’art

→**Tout au long notre développement, nous nommons** les variables, les fonctions, les arguments, les classes et les paquetages.

→Nous nommons aussi les fichiers sources et les répertoires qui le contiennent.

Eviter la désinformation

- Il faut éviter les mots dont le sens établi varie du sens voulu.
- Eviter d'utiliser les mots clés des langages.
- Eviter de choisir des noms trop proches comme XYZController-ForEfficientHandlingsOfStrings dans un module et plus loin XYZControllerFor-EfficientHandlingsOfStrings dans un module.

Faire des distinctions significatives

- Par exemple, puisqu'il est impossible d'employer le même nom pour faire référence à deux choses différentes dans la même portée, vous pourriez être tenté de remplacer l'un des noms de manière arbitraire. Pour certains font une faute orthographe donc si le compilateur corrige le mot donc plus compilation .

Faire des distinctions significatives(2)

- L'emploi de mots de série comme (a1, a2, .., aN) va entraîné une mal compréhension.

Dans des boucles ou dans la signature d'une fonction ou d'une méthode.

- Les articles dans les noms sont des parasites comme (en anglais: a, an et the), en français(un, une, le, les).
- Nous ne devons pas employer dans notre nommage le mot variable.

Choisir des noms prononçables

- Nous pouvons utiliser des mots non prononçables sauf il faut une explication aux équipes.

Evider la codification

- Une époque, la taille des noms de variables étaient importants dues à la mémoire.
- On peut utiliser la notation hongroise ou autres notations.
- On peut maintenant préfixer les variables par m_, mais à éviter car cela peut être un fouillis.

Interface

- On peut préfixer une interface par I.

Noms de classes

- Pour les classes et les objets, nous devons choisir des noms ou des groupes nominaux.
- Un nom de classe ne doit pas être un verbe.

Noms de méthodes

- Choisir des verbes ou des groupes verbaux comme PostPayment, deletePage, ou save.
- Les accesseurs, les mutateurs, et les prédicats doivent être nommés d'après leur valeur et préfixe par get, set ou is conformément au standard du langage.

Choisir un mot par concept

- Les noms doivent être autonomes et cohérents. Par exemple, mettre des termes selon action.

Eviter les jeux de mots

- Eviter les mots qui ont deux sens différents.
- On respectant la règle « un mot par concept ».

Partie 3

Fonctions

Faire court

- Une fonction efficace doit utiliser des méthodes et des fonctions déjà existants.
- Le mauvais code semble justifier les commentaires. Mieux faire des fonctions par exemple pour une condition.
- Eviter de faire des fonctions avec trop de signatures.

Partie 4

Bonnes pratiques du codage

Indentation

- Utilisez les tabulations pour écrire le code.
- Souvent l'indentation est fabriqué automatiquement dans un IDE par espace.
- Si vous modifiez le code de quelqu'un d'autre essayez de garder le style de codage similaire.

```
for (i = 0; i < 10; i++){
      If  ( !quelque chose (i))
           continue;
       do_more ();
  }
```

Indentation(2)

- Les instructions de commutation ont la même casse que le commutateur:

```
switch (x) {
case ‘a’  :
        …
case ‘b’ :
        …
}
```

Performance et lisibilité

- Performance et lisibilité sont étroitement interdépendants et peuvent se résumer aux quatre axiomes qui suivent:
- Il est plus important d'être correct que d'être rapide.
- Un code rapide, difficile à maintenir, sera probablement ignoré.
- Un code bien écrit et documenté aura plus de chances d'être réutilisé.

Où mettre les espaces

- Utilisez un espace avant une parenthèse d'ouverture lors de l'appel de fonctions ou de l'indexation, comme ceci:

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

Bien :

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

Mauvais :

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

Où mettre les espaces (2)

- Mettre un espace entre les types génériques

Bien :

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

Mauvais :

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

Où mettre des accolades

- Dans un bloc de code, placez l'accolades sur la même ligne que l'instruction:

Bien :

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

Mauvais :

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

Où mettre des accolades(2)

- Evitez d'utiliser des accolades inutiles d'ouverture/fermeture, l'espace vertical est généralement limité :

Bien :

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

Mauvais :

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

Où mettre des accolades(3)

- A moins que plusieurs conditions hiérarchiques soient utilisées ou que la condition ne puisse pas être insérée dans une seule ligne.

Bien :

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

Mauvais :

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

Où mettre des accolades(4)

- Lors d'une définition d'une méthode, utilisez le style C pour le placement des accolades, c'est-à-dire utilisez une nouvelle ligne pour l'accolade, comme ceci :

Bien :

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

Mauvais :

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

Où mettre des accolades(5)

- Les propriétés et les indexeurs constituent une exception. Conservez les accolades sur la même ligne que la déclaration de propriété.

Bien :

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

Mauvais :

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

Où mettre des accolades(6)

- On peut compresser accesseur et mutateur:

Bien :

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

Où mettre des accolades(7)

- Si le else est structuré comme suivant :

Bien :

```
if (truc) {
        …
} else {
        …
}
```

Mauvais :

```
if (truc)
{
      …
}
else
{
      …
}
```

Mauvais :

```
if (truc)
 {
      …
 }
 else {
     …
 }
```

Où mettre des accolades(8)

- Les classes et les espaces de noms fonctionnent comme les instructions if, différemment des méthodes :

Bien :

```
namespace N {
        class X {
                …
        }
}
```

Mauvais :

```
namespace N
 {
        class X
        {
            …
        }
 }
```

Résumé

Déclaration	Position
Espace de noms	même ligne
Type	même ligne
Méthode	nouvelle ligne
Propriété	même ligne
Blocs de contrôle (si, pour, …)	même ligne
Types et méthodes anonymes	même ligne

Paramètres multilignes

- Lorsque vous devez écrire des paramètres sur plusieurs lignes, indiquez les paramètres sous ma ligne précédente :

Bien :

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

Les séparateurs vont à la fin:

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

Mauvais :

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

Utilisez des espaces pour plus de clarté

- Utilisez des espaces blancs dans les expressions, sauf en présence de parenthèses :

Bien :

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

Mauvais :

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

Entête de fichiers

- Pour tout nouveau fichier, pensez à utiliser une introduction descriptive, comme ceci :

```
//
//  System.Comment.cs    : commentaires intégrés au fichier de code
//
// Auteur :
//  Jean Martin (jeanmartin@adresse.com)
//
//  Copyright (c) Entreprise
```

Commentaires

- Voir la norme du langage

Enveloppe

- Les noms d'argument doivent utiliser les identificateurs adéquats, comme ceci :

Bien :

```
void Method (string myArgument)
```

Mauvais :

```
void Method (string lpsrtArgument)
void Method (string my_string)
```

Les champs d'instance doivent utiliser le soulignement comme séparateur :

Bien :

```
class X {
        string my_string;
        int very_important_value;
}
```

passable:

```
class X {
        string myString;
        int veryImportantValue;
}
```

Enveloppe (2)

Les champs d'instance doivent utiliser le soulignement comme séparateur :

Bien :

```
class X {
  string myString;
   int very_important_value;
   }
```

mauvais :

```
class X {
    string m_string;
     int _intVal;
      }
```

Référence

Cette partie s’est fortement inspiré du magazine Coding Magazine n°4#Mai/JUIN/JUILLET 2019.