Normes de programmation

Par **Jonathan Clavet-Grenier**

Version 1.0

Distribué le 26 janvier 2016

Déclaration de variables

Une majuscule pour chaque mot.

Les noms des types (Integer, String, etc) sont tous en minuscules. Les types fait par l'utilisateur, tels que des énumérés, des classes, sont écris telles qu'elles ont été écrites.

Il est **recommandé, mais non obligatoire,** d'aligner les déclarations ligne à ligne.

Exemple :

```
int                              IndTab;
int                              UneAutreVariable;
string                           UneAutreVariableVar;
UnEnumereAvecUnNomVraimentLong   UneVariable;
```

Sans la recommandation, voici ce que ça ferait :

```
int IndTab;
int UneAutreVariable;
string UneAutreVariableVar;
UnEnumereAvecUnNomVraimentLong UneVariable;
```

Il est donc toujours plus préférable d'aligner les variables pour une question de visiblité.

Affectation des variables

Il est **recommandé, mais non obligatoire,** d'aligner les lignes une à une comme dans l'exemple ci-dessus lors d'affectations des variables.

```
IndTab              = 1;
UneAutreVariable    = 5;
UneAutreVariableVar = '';
UneVariable         = UnEnumereAvecUnNomVraimentLong.ChoixA;
```

Nom des données membres et des paramètres des procédures

Une donnée membre **doit toujours** commencer par F. Donc une donnée membre ***Rayon*** devrait être appelée ***FRayon***. Pourquoi F? F représente Field Members en anglais et c'est l'un des termes les plus utilisés en entreprise. De plus, il est plus rapide écrire F que «m_ ».

En ce qui concerne les noms des paramètres des procédures et fonctions, nous mettons un _ devant chaque paramètre. Donc, pour un paramètre nommée ***ChaineConnexion***, celui-ci sera écrit ***_ChaineConnexion***.

Début d'une procédure/fonction/if/else/while/for/etc

À chaque début de procédure, fonction, if, else, while et tout autre, il faudra aligner le code à l'aide de la touche **Tab**.

À ÉVITER :

```
int RetournerUnNom()
{
int tati = 0;
return tati;
}
```

On devrait plutôt faire :

```
int RetournerUnNom()
{
  int tati = 0;
  return tati;
}
```

Les noms des procédures, fonctions et variables

Les noms des procédures, fonctions et variables sont en français.

Ils **doivent** tous être portés d'un nom significatif. Une procédure nommée nom() qui retourne le prénom et le nom d'un personnage devrait être plutôt nommée RetournerNomComplet. Il est inutile de préciser « RetournerNomCompletPersonnage », car la fonction est déjà dans une classe CPersonnage. Donc, on sait que la fonction appartient à la classe CPersonnage et c'est donc inutile de préciser.

On peut donc avoir une fonction avec 255 caractères de long, en autant qu'elle aille des noms significatifs et que je n'aille pas besoin d'aller voir les commentaires pour voir ce qu'elle fait.

Simplement avec son nom, je suis supposé de savoir qu'est-ce qu'elle fait.

En ce qui concerne les variables, une variable nommée _Chaine, par exemple, ne dit pas grand-chose. Une chaîne de quoi? De caractères? De connexion? De mot de passe? Dans le cas d'une chaîne de connexion, il faudrait écrire _ChaineConnexion.

Longueur des procédures et fonctions

La longueur des procédures et des fonctions **ne devraient pas** dépasser la longueur d'une page-écran. C'est-à-dire qu'on ne devrait pas « scroll » pour voir la procédure ou fonction au complet dans une résolution de 1980x1080. Si on doit scroll, c'est que généralement la fonction ou procédure est trop longue.

Bien sûr, il arrive qu'il aille certaines exceptions.

Du code suivi d'un while/for/if

Il est obligatoire de faire un espace avant et après le while/for/if. Exception : Lorsque la fonction finit immédiatement après le if, le while, etc.

À ÉVITER :

```
int tata()
{
  CPersonnage tati = null;
  tati = new CPersonnage();
  if (tati == null)
    return 0;
  else
    return 1;
}
```

On devrait plutôt faire cela :

```
int tata()
{
  CPersonnage tati = null;
  tati = new CPersonnage();

  if (tati == null)
    return 0;
  else
    return 1;
}
```

Dans le cas qu'il y aille un commentaire :

```
int tata()
{
  CPersonnage tati = null;
  tati = new CPersonnage();
```

```
  // Un commentaire ici. Notez l'espace entre tati et celui-ci.
  if (tati == null)
    return 0;
  else
    return 1;
}
```

Les opérateurs et l'appel des procédures et fonctions

Il doit avoir toujours un espace de chaque bord lorsqu'on utilise un opérateur ou entre les paramètres d'une procédure ou fonction.

À ÉVITER :

```
int tata=0;

bool toto==0;

UneProcedure(Parametre1,Parametre2,UneAutreFonctionQuiRetourneQuelq
ueChose(Param1,Param2));
```

Voici la bonne manière de le faire.

```
int tata = 0;

bool toto = (tata == 0);

UneProcedure(Parametre1, Parametre2,
UneAutreFonctionQuiRetourneQuelqueChose(Param1, Param2));
```

Lorsqu'il y a plusieurs conditions dans un if/while

Il est obligatoire de mettre des parenthèses pour chaque condition.

Ne pas créer 5 if un dans l'autre. Créer une approche plus claire au code.

Lorsqu'il y a trop de conditions pour aligner sur une seule ligne, alignez les conditions une en-dessous de l'autre comme ceci :

```
if (Condition) and

   (Condition) and

   (Condition)

{

  …

}
```

Lorsqu'il y a du code relativement long

Lorsqu'il y a du code relativement long, la question à se demander est : Si j'étais un autre programmeur et que je voyais ça, est-ce que je me casserais la tête pour comprendre?

Sincèrement, demandez-vous le et répondez-vous honnêtement. Si la réponse n'est pas instantanément « oui », alors commentez le code ligne par ligne s'il le faut. Il faut que ça soit **clair.**

On sait tous à quel point c'est démotivant et frustrant d'essayer de comprendre le code d'un autre. Alors SVP, respectez cette norme.

Procédures et fonctions

On utilise le /// au-dessus de la procédure et de la fonction pour le C#. Pour le reste, on utilise les commentaires manuellement.

On précise la description de la fonction, qu'est-ce qu'elle fait? Une fonction nommée EffectuerUnTour devrait avoir une description plus précise de ce qu'elle fait. « Effectue un tour » n'est pas un bon commentaire, mais plutôt : « Effectue un tour en faisant parcourir la liste des avions en vol dans la carte du monde. ».

On précise aussi les paramètres. Encore une fois, on doit avoir plus de détails sur ce qu'est le paramètre et non une « répétition » du nom. _ChaineConnexion : Représente la connexion. Ceci n'est pas un bon commentaire. Un bon commentaire serait plutôt : Représente une connexion vers la base de données ProgressQuest.

Finalement, dans le cas d'une fonction, on précise ce qu'elle retourne.

Note : On ajoute un « À noter » s'il y a quelque chose d'anormal que l'on doit noter tel que : Peut provoquer une custom exception. Voir le Try catch ci-dessous pour comprendre.

If/Else

Ne pas insérer de brackets pour une ligne de code dans un if.

Try catch

Ceci sont des cas rares. Mais je les écris au cas où que ça l'arrive. Il peut arriver que, parfois, qu'on écrive qui doit retourner Vrai ou Faux, mais que s'il y a une erreur qui se produise, que l'on doit retourner l'erreur. Votre premier réflexe serait de mettre un paramètre en « out » et de retourner une string ou un integer précisant le message d'erreur.

C'est une mauvaise pratique. Et en entreprise, si vous codez en orientée objet, vous allez comprendre pourquoi.

Imaginons que nous avons une fonction, qui peut avoir 10 messages d'erreurs différents, et qui doit seulement retourner Vrai ou Faux. Vrai si ça la fonctionné. Faux si ça n'a pas fonctionné. Imaginons que cette fonction est dans une classe CCommande et imaginons que cette fonction s'appelle Commander. Lorsqu'on l'appelle, la fonction tente de commander les objets, mais imaginons qu'elle n'aille plus accès à internet, comment fait-on pour retourner un message d'erreur avec un code d'erreur et son message d'erreur? Dans ce cas-ci, on le sait d'avance que si ça plante, c'est à cause d'internet. Donc, on créer une classe CCommandeException qui va hériter de la classe Exception.

https://msdn.microsoft.com/en-us/library/87cdya3t(v=vs.110).aspx

Ensuite, dans notre code, on va instancier un objet de type CCommandeException et en faisant appel au nouvel objet, on va pouvoir indiquer le code et le message d'erreur.

Celui qui va appeler la fonction, va être obligé de faire un *try catch* dans l'appel de la fonction.

```
Try
{
  Mafonction();
}
Catch e as Exception
{
  If (e == CCommandeException)
    MonInterface.AfficherMessage('Une erreur s'est produite lors de
la commande. Voici le message d'erreur' : + e.Message);
  else
  {
    MonInterface.AfficherMessage('Une erreur inconnue s'est
produite. Fermeture de l'application. Voici le message d'erreur :'
+ e.Message);
  Application.Exit();
 }
}
```

Évidemment, ceci est très général. L'avantage d'utiliser les exceptions, c'est que l'on permet au programmeur de savoir exactement ce qu'est l'erreur et ça sera à lui de savoir qu'est-ce qu'il

souhaite faire avec. De plus, contrairement à un code d'erreur que l'on recevait (la première manière que j'ai dit que c'était à éviter), il n'aura pas besoin d'aller voir le code d'erreur pour savoir c'est quoi l'erreur, car il aura déjà un objet expliquant déjà c'est quoi l'erreur!

À chaque création d'un objet

Il est très **recommandé, mais non obligatoire,** de mettre un try finally lors de la création des objets. Il peut arriver, que parfois, le programme plante ou qu'il se passe un bug pour une raison inconnue. Dans ces cas, si le try finally n'est pas là, les objets en mémoires ne sont jamais libérés et causant ainsi une fuite de mémoire. Donc, à chaque création d'un objet, on devrait mettre un try finally. S'il y en a plusieurs, les mettres dans un seul try finally comme dans l'exemple ci-dessus :

```
CPersonnage Hero    = null;
CPersonnage Monstre = null;

try
{
  Hero    = new CPersonnage();
  Monstre = new CPersonnage();
}
finally
{
  Hero.Free();
  Monstre.Free();

  Hero    = null;
  Monstre = null;
}
```

Évidemment, dans le langage C#, nous n'avons pas à se soucier de ça. Mais dans tous les autres langages, il serait préférable d'optimiser le plus possible le rendement en mémoire et le processeur pour donner une belle expérience à l'utilisateur qui utilise notre programme.