Règles de codage

Rédigé pour : IUT Informatique, Université d'Auvergne Clermont1
Rédigé par : Marc Chevaldonné, IUT Informatique, Université d'Auvergne Clermont1

20 septembre 2013
Version 5.0

Sommaire

1. Introduction

1.1. Contexte

L'objectif de ce document est de fournir un exemple de règles sur différents aspects de la production de code en C# pour les étudiants de l'IUT d'Informatique de Clermont-Ferrand (Université d'Auvergne Clermont1). Il propose tout d'abord des règles de formatage de votre code pour que tous les fichiers aient le même aspect visuel ; puis il décrit de quelle manière vous devez commenter votre code afin de produire une documentation utile ; enfin, il propose un patron d'organisation de fichiers et de dossiers qu'il est fortement conseillé de lire.

Ce document a été écrit pour accompagner mes cours de C# à l'IUT d'Informatique. Il reprend les règles communes et classiques, adoptées par la très grande majorité des développeurs .NET. Il n'a pour autant nullement l'intention de se substituer à d'autres règles de codage (proposées par d'autres groupes de développeurs, d'autres entreprises) et en particulier celles écrites pour d'autres langages de programmation.

1.2. Références

Pour plus d'informations, le lecteur peut se documenter lui-même en lisant les documents suivants :

Titre	Version	Lien
C# Programming Guide - XML Documentation Comments	2010	http://msdn.microsoft.com/en-US/library/b2s063f7%28v=VS.80%29.aspx
Quick Guide Doxygen	3.0	http://marc.chevaldonne.free.fr/ens_rech/DocumentsDivers_files/Quick_Guide_Doxygen.pdf
Doxygen Manual	1.7.1	http://www.stack.nl/~dimitri/doxygen/manual.html
Guick Guide Git TortoiseGit	1.1	http://marc.chevaldonne.free.fr/ens_rech/DocumentsDivers_files/Quick Guide Git TortoiseGit.pdf
Quick Guide TortoiseSVN	2.0	http://marc.chevaldonne.free.fr/ens_rech/DocumentsDivers_files/Quick Guide TortoiseSVN.pdf

1.3. Table des modifications

Auteur	Modification	Version	Date
Marc Chevaldonné	Écriture de ce document	1.0	24 novembre 2008
Marc Chevaldonné	Ajout de règles spécifiques au C#	2.0	08 septembre 2009
Marc Chevaldonné	Ajout du 4ème paragraphe et modifications mineures	3.0	06 août 2010
Marc Chevaldonné	Traduction du document en français + quelques modifications (sur les noms d'interfaces et de types génériques par exemple)	4.0	09 septembre 2012
Marc Chevaldonné	Modifications mineures (propriétés calculées, nommage en XAML)	5.0	20 septembre 2013


iut
Université d'Auvergne
UdA

2. Format et style de codage

Voici les règles de base à respecter lors de la production de code.

2.1. Règles générales d'écriture et de nommage

- Ne pas utiliser la notation Hongroise[1].
- Utiliser des propriétés pour accéder aux membres des classes (e.g. "`public string Name { get{...} set{...} }`", pour accéder à "`private string mName`") et n'utiliser que des champs privés ou protégés.
- Ne jamais utiliser de méthodes commençant par `Get` ou `Set`. Préférer les propriétés calculées (e.g. préférer "`public float Moyenne { get{...} }`" à "`public float GetMoyenne() {...}`" ; préférer "`public int Paramètre { set{...} }`" à "`public void SetParamètre(int value) {...}`").
- Casse
  - Espaces de nom : camelback notation[2], en utilisant un préfixe : par exemple "`gi`" dans le cadre des cours de C# (e.g. "`giConnect4`")
  - Classes : camelback notation[2], en démarrant avec une majuscule (e.g. "`MySuperClass`")
  - Interfaces : même règles que les classes mais doivent en plus commencer par un I majuscule (e.g. "`IMySuperInterface`")
  - Types génériques : dans une classe ou une méthode générique, le nom du type respecte les mêmes règles que pour les classes mais doit en plus commencer par un T majuscule (e.g. "`TMySuperType`")
  - Dossiers : utiliser un dossier par espace de nom avec le même nom que l'espace de nom associé (e.g. "`src/giConnect4`")
  - Enums : camelback notation[2] (e.g. "`PlayerType`")
  - Variables locales : utiliser, soit des minuscules et des underscores pour séparer les mots (e.g. `my_cool_variable`), soit la notation camelback mais en gardant une minuscule pour le premier mot (e.g. `myCoolVariable`)
- Membres des classes
  - Champs d'instance : les noms des variables commencent avec un '`m`' minuscule, puis le reste en notation camelback (e.g. "`int mNiceNumber;`")
  - Propriétés : camelback notation[2], en commençant avec une majuscule (e.g. "`MySuperProperty`"). Si la propriété est associée à un champ, elle doit avoir le même nom que le champ sans le '`m`' minuscule (e.g. `Name` for `mName`).
  - Méthodes de classes publiques : camelback notation en commençant avec une majuscule (e.g. "`bool IsFull();`"
- Accolades : BDS (en colonnes) style. Exemple :

```
if(something == true)
{
...
}
```

- Commentaires : utiliser la documentation automatique du C# .NET pour les espaces de nom, les classes, les champs, les propriétés et les méthodes.

---

[1] voir http://en.wikipedia.org/wiki/Hungarian_notation pour plus d'informations.

[2] Pas de séparation entre les mots, mais la première lettre de chaque mot est en majuscule (ThisIsCamelbackNotation). Voir http://en.wikipedia.org/wiki/Camelcase pour plus d'informations

2.2. Règles d'écriture et de nommage en XAML

Tous les éléments graphiques en XAML respectent les règles de nommage précédentes (le nom d'un membre de type TextBlock commencera également avec un m minuscule). Toutefois, pour les variables de type Control, le nom est composé de : `m` + le type du Control + le nom. Par exemple : `mTextBlockName` pour une `TextBlock` affichant un nom ; `mButtonStart` pour un bouton permettant de démarrer quelque chose, etc...

2.3. Espaces de noms

Le code doit être organisé en espaces de noms (*namespace*) afin de permettre une meilleure lisibilité et une meilleure compréhension du code. Suivez ces instructions à propos de la définition et de l'utilisation des espaces de noms.

Les espaces de noms doivent utiliser un préfixe de 2 ou 3 lettres en minuscules en lien avec l'application ou la bibliothèque de classes à laquelle ils sont liés. Dans le cadre des TP de C#, nous utiliserons par exemple le préfixe "`gi`" pour "Gestion Informatique". Si vous utilisez des espaces de nom hiérarchiques, le même préfixe doit être attribué aux enfants également.

Exemple de code

(Les commentaires ont été supprimés afin de mettre en évidence le style et la casse ; pour les considérations sur les commentaires, lire la section 3)

```
namespace giConnect4
{
        class MyCoolClass : BaseClass
        {
              public void CalcNumber(int value)
              {
                    ...
              }

              public int Name
              {
                    get
                    {
                          return mName;
                    }
                    set
                    {
                          mName = value;
                    }
              }
              private int mName;

              private int mMySpecialNumber;
        }
}
```

Exemple d'espace de nom (*namespace*)

(Les commentaires ont été supprimés afin de mettre en évidence le style et la casse ; pour les considérations sur les commentaires, lire la section 3)

```
namespace giGames
{
        namespace giConnect4
        {
              ...
        }
}
```

2.4. File header

Chaque fichier .cs (ainsi que n'importe quel autre fichier d'ailleurs...) devrait débuter avec un header regroupant une série d'informations, comprenant :

- ce sur quoi porte ce fichier (*Module*)
- le créateur du fichier (*Author*)
- la date de création du fichier (*Creation Date*)

Voici un patron de header que vous pouvez utiliser pour vos fichiers source.

```
// ============================================================================
//
// Copyright (C) 2008-2009 IUT CLERMONT1 - UNIVERSITE D'AUVERGNE
//                         www.iut.u-clermont1.fr
//
// Module       : Board - source file
// Author       : Marc Chevaldonné
// Creation date: 2008-11-25
//
// ============================================================================
```

Voici un autre patron de header que vous pouvez utiliser pour vos fichiers de type XML (XML, XSD, XSL...).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- ============================================================================

   Copyright (C) 2008-2009 IUT CLERMONT1 - UNIVERSITE D'AUVERGNE
                           www.iut.u-clermont1.fr

   Document     : Scores - XML schema
   Author       : Marc Chevaldonné
   Creation date: 2008-11-25

============================================================================
```

Il ne vous reste plus qu'à changer les informations par celles vous concernant.

3. Documenter le code

Cette section propose un guide de documentation de code C#. Documenter le code est important pour aider d'autres personnes à utiliser votre propre code, mais également pour vous aider à revoir ou modifier manuellement vos propres algorithmes et classes. Faites simplement l'effort de porter l'attention nécessaire à l'ajout de commentaires dans vos fichiers ; la qualité de l'ensemble de votre projet en dépend. Pour la production de documentation, je vous conseille d'utiliser le système de documentation automatique du C# .NET (voir http://msdn.microsoft.com/en-us/library/b2s063f7(VS.80).aspx pour plus de détails) et/ou Doxygen (voir http://www.stack.nl/~dimitri/doxygen/manual.html pour plus de détails). Ceci nécessite simplement d'étiqueter vos commentaires d'une manière spécifique pour qu'ils puissent être examinés proprement par le système de documentation automatique de .NET ou de Doxygen. Les sections suivantes décrivent comment étiqueter vos blocs de commentaires pour qu'ils soient correctement interprétés par ces systèmes.

3.1. Introduction

Il y a deux types de blocs de commentaires qui peuvent être utilisés dans vos fichiers source :

- un ***Documentation Comment***, qui est un bloc de commentaires spécial, qui est voué à être traité par le système de documentation .NET ou Doxygen et publié dans la documentation qui sera générée.
  Les *documentation comments* sont insérés en suivant les conventions spéciales suivantes :

```
///<xml_element>
/// ...
///</xml_element>
```

Exemple complet :

```
/// <summary>
/// Class level summary documentation goes here
/// </summary>
/// <remarks>
/// Longer comments can be associated with a type or member
/// through the remarks tag
/// </remarks>
public class TestClass
{
      /// <summary>
      /// Description for SomeMethod.
      /// </summary>
      /// <param name="s"> Parameter description for s goes here</param>
      /// <param name="f"> Parameter description for f goes here</param>
      /// <returns> Return results are described through the returns tag.
      /// </returns>
      /// <seealso cref="System.String">
      public int SomeMethod(string s, float f)
      {
            ...
      }
}
```

On distingue également deux types de *documentation content* :

- ***Reference Guide Documentation*** : également appelée *API Specification*, qui contient l'information minimale qu'un développeur peut attendre de votre documentation.
  Il s'agit d'explications détaillées sur ce qu'une bibliothèque, une classe ou une fonction peuvent faire et comment les utiliser.

- ***Programming Guide Documentation*** : ce qui sépare l'*API specifications* du *programming guide*, ce sont les exemples, la définition de termes, de certains concepts généraux (comme des métaphores), la description des bugs d'implémentations ou des raccourcis. Il est évident que ceux-ci contribuent grandement à la compréhension et aide à une utilisation adaptée plus rapide.

Dans le cadre des cours de C#, j'attends au moins un ***Reference Guide Documentation*** acceptable pour garantir un bon travail en équipe et une bonne maintenance de code. Toutefois, j'encourage également l'insertion de commentaires de type ***Programming Guide Documentation***, autant que faire se peut.

- un ***Code Comment*** a pour but d'aider les développeurs à comprendre comment et pourquoi le code a été écrit de telle ou telle manière. Ces commentaires ne sont pas traités par le générateur de documentation de .NET ou Doxygen et ne seront donc pas publiés dans la documentation générée.
  Les commentaires de code source sont mélangés avec les instructions et les définitions et ne suivent qu'une règle particulière.
  Les *code comments* sont insérés en utilisant les notations de programmations du langage pour les commentaires :

```
// ... text ...
```

  Utilisez les notations suivantes seulement pour désactiver un morceau de code (en mode debug par exemple) :

```
/* ... text ...*/
```

  ou

```
/*
... text ...
*/
```

3.2. Conseils pour l'écriture de *Documentation Comments*

Un *documentation comment* précède la déclaration d'un espace de nom (*namespace*), d'une classe ou la définition d'un membre, d'une méthode, d'une propriété, d'un indexeur... C'est une description mélangée avec quelques étiquettes spéciales de commentaires.

Pour chaque déclaration ou définition, il y a deux types de descriptions, qui ensemble forment la documentation : une brève description (***brief description***) et une description détaillée (***detailed description***). La première est obligatoire et la suivante optionnelle. Il n'est pas autorisé d'avoir plusieurs brèves descriptions ou plusieurs descriptions détaillées (dans plusieurs fichiers par exemple, dans le cas de classes partielles).
Pour la description brève, utilisez l'étiquette `///<summary>`, et pour la description détaillée, utilisez l'étiquette `///<remark>`. Une description brève ne doit pas dépasser une ligne.

Certaines déclarations ou définitions peuvent nécessiter des descriptions plus complètes. Pour ce faire, voici une liste non exhaustive des étiquettes obligatoires et recommandées :

Étiquettes de commentaires obligatoires	
`<summary>`	description brève d'un espace de nom, d'une classe, d'un membre, d'une propriété, d'une méthode (...) suivie du commentaire
`<param>`	description d'un paramètre ou d'un argument. Notez que cet élément à un attribut obligatoire **`name`** qui doit contenir le nom du paramètre.
`<returns>`	description de la valeur de retour d'une méthode.

Étiquettes de commentaires conseillées	
`<remark>`	description détaillée (et donc plus longue que **`<summary>`**) ou commentaire générale
`<exception>`	description d'une exception pouvant être lancée par la méthode ou la propriété
`<example>`	un exemple documenté
`<see>` `<seealso>`	une référence à un membre, une classe, une méthode, un espace de nom (...) accessible par le compilateur. Notez que **`<seealso>`** a un attribut obligatoire **`cref`** contenant le nom de la classe référée, la méthode, la propriété... **`<seealso>`** est notamment recommandé comme commentaire d'un membre privé associé à une propriété.

3.3. Ordre suggéré pour les étiquettes les plus communes

Légende :

`<obligatoire>`
`<commun>`
`<rare>`

```
///<summary>
/// ...
///</summary>
///<remark>
///...
///</remark>
///<param name="sthg">...</param> requis pour chaque paramètre, même si la
                                  description est évidente
///<returns>...</returns> requis pour chaque méthode retournant autre chose que
                          void, même si la description est redondante avec la
                          description de la méthode (Essayez autant que faire se
                          peut d'écrire quelque chose de non redondant (quelque
                          chose de plus spécifique par exemple))
///<seealso cref="...">...</seealso>
///<example>...</example>
```

3.4. Style guide

- **N'utilisez pas les parenthèses pour la forme générale des méthodes et des constructeurs**
  Lorsque vous faites référence à une méthode ou un constructeur ayant différentes réécritures, et que vous voulez faire référence à une de ces réécritures en particulier, utilisez les parenthèses et les types d'arguments. Par exemple, `MyClass` a 2 méthodes `GetSomething` : `GetSomething(int)` et `GetSomething(int, bool)`.
  Toutefois, si vous faites référence aux deux formes de la méthode, n'écrivez pas les parenthèses. Cela pourrait induire qu'on fait référence à une version de la méthode sans argument. L'objectif ici est de bien faire la distinction entre la forme générale de la méthode et ses réécritures particulières. Ajoutez le mot *method* pour bien faire la distinction avec un champ ou une propriété.
  - évitez : `the GetSomething() enables you to get something`
    You mean "all forms" of the `GetSomething` method
  - préférez : `the GetSomething method enables you to get something`
- **Utilisez des phrases courtes sans sujet à la place de phrases complètes, pour rester bref et concis**
  Ceci s'applique particulièrement pour les descriptions de `<param>` et `<returns>`
- **Les descriptions de méthodes commencent avec un verbe sans sujet**
  Une méthode implémente une opération, la description commence donc généralement par un verbe :
  - évitez : `This method gets the label of this button`
  - préférez : `Gets the label of this button`
- **Utilisez la 3ème personne plutôt que la 2ème**
  - évitez : `Get the label`
  - préférez : `Gets the label`
- **La description d'une classe, d'un membre, d'une propriété ou d'un argument peut omettre le sujet et simplement donner l'état de l'objet**
  - évitez : `This argument is a button label`
  - préférez : `A button label`
- **Un commentaire de membre de classe peut être omis s'il y a une propriété associée à ce champ**
  Cependant, vous pouvez utiliser à la place `<seealso cref="MyProperty"/>` pour faire référence à la propriété associée. Dans ce cas, choisissez de déclarer le champ privé sous la propriété, comme dans l'exemple suivant :

```
///<summary>
///description de ma propriété
///</summary>
public int Entier
{
    get
    {
        return mEntier;
    }
    private set
    {
        mEntier = value;
    }
}
///<seealso cref="Entier"/>
private int mEntier;
```

- **Utilisez “this” (ou «ce», «cet», «cette») à la place de “the” («le») quand vous faites référence à l’objet créé par la classe en cours de documentation**
  - évitez : `Gets the determinant of the matrix`
  - préférez : `Gets the determinant of this matrix`

- **Choisissez des noms auto-descriptifs et des descriptions allant au-delà de ces descriptions par nom**
  Les meilleures noms d’API sont «auto-documentés», c’est-à-dire qu’à leur simple lecture vous comprenez à quoi sert une classe, une méthode, une propriété, ou ce qu’elles font.
  Si la documentation ne fait que répéter le nom de l’API dans une phrase, elle n’apporte aucune information.
  Par exemple, si la description d’une méthode ne fait qu’utiliser les mots qui apparaissent dans son nom, vous n’obtenez pas d’information supplémentaire.
  Le commentaire idéal va au-delà du nom et doit toujours apporter une information supplémentaire qui n’était pas évidente à partir du nom de l’API.
  Par exemple, pour la méthode `SetToolTipText` :
  - évitez : `Sets the tool-tip text`
  - préférez : `Registers the text to display when the cursor lingers over the component`

3.5. Exemple complet

```
///<summary>
///contains classes composing the Connect4 game
///</summary>
namespace giConnect4
{
    ///<summary>
    ///contains data related to the connect4 board,i.e. the matrix and its content
    ///</summary>
    ///<remark>
    ///manages the matrix board of the Connect4 game and its evolution:
    ///allows inserting chips in columns, checking if a column or the complete
    ///board is full or not, checking if one player has made a n-chip line, etc.
    ///</remark>
    ///<seealso cref=”Game”/>
    class Board
    {
        ///<summary>
        ///inserts a new chip from one player in a particular column, if not full
        ///</summary>
        ///<remark>checks internally that the column is not full
        ///by calling IsColumnFull</remark>
        ///<param name=”column”>column in which the player inserts a chip</param>
        ///<param name=”playerId” cref=”Player”>the id of the player inserting
        ///a chip. It should match the mId member of a Player instance</param>
        ///<returns>true if the chip has been inserted correctly, false instead
        ///(mainly because the column is full)</returns>
        public bool InsertChip(int column, int playerId)
```

```
            {
                //checks if the column is full or no before inserting the chip
                if(IsColumnFull(column) == false)
                {
                    ...
                }
                ...
            }
            ...
            ///<summary>the matrix representing the board</summary>
            ///<remark>in each case, 0 stands for empty case, and 1 or 2 represent a
            ///chip of a player (the number is the player id)</remark>
            private int[,] mMatrix;
        }
    }
```

3.6. Activer la génération de documentation automatique de .NET

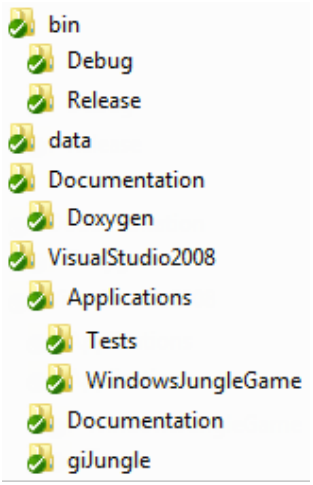
Afin de demander au compilateur de générer la documentation à la compilation, vous devez :

1. dans l'Explorateur de solutions, cliquer droit sur le projet puis cliquer sur Propriétés
2. ouvrir le dossier de configuration de propriétés et cliquer sur Générer
3. modifier la propriété Fichier de documentation XML en lui donnant par exemple la valeur XMLdocumentation.xml
4. dans le menu Générer, cliquer sur Générer. Le fichier de sortie XML sera dans le dossier de sortie (debug ou release)

3.7. Activer la génération de documentation par Doxygen

Pour utiliser Doxygen, via le Doxygen Wizard ou directement depuis VIsualStudio pour produire une documentation HTML, Latex, XML (...), vous pouvez vous référer à un autre de mes documents à succès : Doxygen - Quick Guide v2.x.

4. Fichiers et arborescence de dossiers



Cette section propose une organisation de vos dossiers et fichiers pour une meilleure gestion par un système de gestion de version et de production de code source pour un client. Le dossier racine doit contenir au moins les dossiers suivants :

- `bin` : contient deux dossiers : `Debug` and `Release`, tous deux contiennent les résultats de la compilation de vos projets (respectivement en mode debug et release), i.e. `*.dll`, `*.exe`.[3]

- `data` : contient des ressources, e.g. images, videos, xml files... à organiser en sous-dossiers

- `Documentation` : contient des fichiers utilisés pour la création de la documentation Doxygen dans le sous-dossier `Doxygen` (i.e. Doxyfile, footers, logos...). Peut contenir d'autres sous-dossiers comme : `UML` (qui contiendrait les diagrammes UML de vos projets), `Documents` (qui pourrait contenir le cahier des charges, les spécifications fonctionnelles, les tutoriels, les guides utilisateurs...), et bien entendu les sous-dossiers (non suivis par le système de contrôle de versions) contenant la documentation automatique produite.

- `VisualStudio2010` : contient tous les fichiers source (`*.cs` files), les projets (`*.csproj`) et les solutions (`*.sln`).

    - Les solutions `*.sln` et `*_all.sln` (e.g. `Jungle.sln` et `Jungle_all.sln`) apparaissent directement dans le dossier `VisualStudio2010`. Le `*.sln` ne contient que les projets indispensables à votre application. En revanche, le `*_all.sln` contient tout ce que contient `*.sln` ainsi que tous les projets de test.

    - Les différents projets de vos solutions sont séparés en différents dossiers nommés avec les espaces de noms (e.g. `giJungle` contient le projet qui compilera en l'assemblage `giJungle.dll` qui est l'assemblage principal de l'application).

    - `Documentation` contient le projet de type *makefile* pour la génération de la documentation via Doxygen (voir Doxygen - Quick Guide v2.0 pour plus de détails).

    - `Applications` contient les différents exécutables, i.e. ceux avec l'interface graphique (e.g. `WindowsJungleGame`) et tous les tests des différents projets dans le sous-dossier `Tests`.

Pour plus d'informations sur l'organisation en fichiers et dossiers et l'utilisation avec un système de contrôle de versions, vous pouvez vous référer à mon autre document à succès : Quick Guide Git TortoiseGit.

[3] Ces dossiers ne sont généralement pas suivis par les systèmes de contrôle de versions.