Assimiler

Qualité et documentation du code

Rédigé par

David ROUMANET
Professeur BTS SIO

Changement

Date	Révision

Sommaire

A Introduction....1
A.1 Impacts du code dans la vie courante....1
A.2 Recommandations....2
B Principes du "Secure by design"....3
B.1 KISS (Keep It Simple Stupid)....4
B.2 STUPID (Singleton, Tight Coupling, Untestability….)....4
B.3 SOLID (Single responsiblity, Open Principle, …)....5
B.4 Loi de Déméter (LoD)....5
B.5 YAGNI (You Aren't Gonna Need It)....6
C Gestion de la qualité du code avec SonarLint....7
C.1 Exemple de règles....7
C.2 Installer SonarLint....8
C.3 Mise en œuvre SonarLint avec Visual Studio....11
D Commentaires et documentation....14
D.1 Les commentaires....14
D.1.1 Entêtes de fichiers....14
D.1.2 Entêtes de fonctions....14
D.1.3 Détails d'opérations complexes....14
D.1.4 Javadoc en Java et XML en C#....15
D.1.4.a Javadoc....15
D.1.4.b XML documentation....15
D.1.5 Autres outils de commentaires....17
D.1.6 Génération de documentation....17
D.1.6.a Installation de l'extension Ghostdoc....17
D.1.6.b Utilisation de GhostDoc....18
D.1.6.c Génération d'une documentation....19
E Développement par les tests unitaires....20
E.1 Fonctionnement....20

Nomenclature :

- **Assimiler** : cours pur. Explication théorique et détaillée (globalement supérieur à 4 pages).
- **Décoder** : fiche de cours, généralement inférieure à 5 pages.
- **Découvrir** : Travaux dirigés. Faisable sans matériel.
- **Explorer** : Travaux pratiques. Nécessite du matériel ou des logiciels.
- **Mission** : Projet encadré ou partie d'un projet.
- **Voyager** : Projet en autonomie totale. Environnement ouvert : Vous êtes le capitaine !

A Introduction

La qualité et la fiabilité du code sont intimement liés.

Dans ce cours, nous étudierons les impacts de codes mal conçus, les concepts pour coder correctement et nous verrons quelques outils pour faciliter le travail des développeurs.

Enfin, nous étudierons les moyens pour documenter correctement une application, afin de ne pas avoir à générer une documentation séparée.

A.1 Impacts du code dans la vie courante

Récemment, le double crash de Boeing 737 max a montré que le développement informatique est devenu un élément critique de nos vies, tel que l'explique cet article Developpez.com↗.

> Des ingénieurs de longue date de Boeing pensent avoir la réponse à cette question. Ils affirment que le logiciel défaillant du 737 Max a été mis au point à un moment où Boeing congédiait des ingénieurs expérimentés et faisait pression sur les fournisseurs pour qu'ils réduisent leurs coûts. Selon eux, c'est la tendance de l'avionneur américain à faire sous-traiter certaines opérations auprès d'entrepreneurs moins bien rémunérés, notamment celles en relation avec le génie logiciel, qui l'a conduit dans cette impasse. Afin de mener à bien certaines opérations clés, notamment celles liées au développement et au test de ses solutions logicielles, Boeing et ses sous-traitants auraient pris l'habitude de faire appel à des travailleurs temporaires, gagnant à peine 9 dollars de l'heure, souvent issus de pays dépourvus de connaissances approfondies en aérospatiale, notamment en Inde.

À cause d'un bogue, les Airbus A350↗ sont réinitialisés toutes les 149H. Il s'agit d'un simple problème de débordement de mémoire, et pourtant…

Le dysfonctionnement de Pajemploi↗ qui est enfin corrigé a privé 6000 familles du complément de libre choix du Mode de Garde : un problème de transfert entre la CAF et Pajemploi.

Il est possible de parler du bogue Android↗ qui peut exposer tout le carnet d'adresse, sans oublier Apple↗ et les portiques de la SNCF, etc.

A.2 Recommandations

Le génie logiciel doit aujourd'hui repenser son fonctionnement mais surtout, éduquer les mentalités des développeurs.

Le magazine Programmez! Spécial automne 2023 rappelle en page 34, quelques principes du "secure by design" côté développeur.

À ce titre, les projets simulés en écoles et milieux scolaires devraient mieux intégrer la sécurité et la qualité du codage. **En tant qu'étudiant de BTS SIO, vous devez intégrer les bonnes pratiques**.

Ce premier support, apporte donc des éléments de réponses :

- bonne pratique
- gestion de la qualité du code
- développement par les tests (TDD)

B Principes du "Secure by design"

Le codage d'une application, par une équipe de développement implique des risques importants :

- Chacun utilise son propre éditeur (VS Code, Eclipse, Sublim Text, etc.)
  - Chaque éditeur génère des éléments supplémentaires dans un projet (fichiers d'initialisation)
  - Chaque développeur utilise ses propres préférences (symbole { après une déclaration ou bien avec un retour à la ligne ? Commentaires avec // ou avec /* */ ?)
- Chacun utilise ses propres règles de nommages
- Chacun code au niveau de ses capacités
  - Un bon codeur factorisera trop son code
  - Un codeur plus faible générera des lignes inutiles
- Chacun travaille pour son propre intérêt
  - Utilisation de codes et bibliothèques extérieures
  - Simplification des problèmes (en omettant la sécurité)
  - Report des responsabilités sur les autres

Bref, le travail d'équipe est une contrainte que peu de développeurs débutants acceptent. Le regard des autres, la peur de mal faire et d'avoir des remarques ou encore d'être non légitime sur le poste sont autant de motifs de préférer travailler seul.

Mais là encore, travailler seul ne garantit pas de générer un code fiable et exempt de problème de sécurité.

Avec le temps, un certain nombre de règles ont fait leur apparition, pour aider les développeurs, en voici quelques-unes.

B.1 KISS (Keep It Simple Stupid)

C'est une sorte de commandement du développeur : chaque fonction doit être simple et stupide. Il s'agit de ne pas créer une "usine à gaz" avec une fonction qui effectue plusieurs traitements séquentiels mais de découper chaque fonction en brique élémentaire, un peu comme avec un jeu Lego.

D'après Steve Mc Connel et Robert C. Martin (oncle Bob), une méthode ne doit faire qu'une chose : ce n'est pas une question de taille (nombre de lignes) mais de capacité.

B.2 STUPID (Singleton, Tight Coupling, Untestability….)

Cette fois, il s'agit des mauvaises pratique lors du codage. L'acronyme s'explique par...

- **S**ingleton : l'usage du design pattern "Singleton" n'est pas toujours mauvais, mais il y a un risque de limiter l'accès à d'autres éléments car il agit comme un goulet d'étranglement.
- **T**ight Coupling : lorsque les différents modules d'un programme sont trop imbriqués, cela implique qu'une modification sur l'un entraîne des modifications dans plusieurs autres.
- **U**ntestability : un code qui n'est pas testable, est un mauvais code. Le temps gagné à ne pas écrire de tests unitaires, sera probablement perdu lors des différentes évolutions ou lors des tests fonctionnels...
- **P**remature Optimization : ce n'est pas l'optimisation qui est mauvaise, mais c'est le fait de vouloir le faire trop tôt dans les différentes phases de développement. Les phases "alpha" ou "beta" sont des phases ou les performances ne sont pas importantes.
- **I**ndescriptive Naming : le mauvais nommage des méthodes, des classes et des attributs conduit souvent aux erreurs de fonctionnement et de compréhension (personnellement, en Purebasic, j'ai perdu beaucoup de temps à cause d'une variable appelée *cptp* que j'ai écrit *ctpp* dans une autre partie de code… pour abréger compteurProduction)
- **D**uplication : le copier/coller d'une partie de code signifie qu'il y a une erreur de conception. Il faut éviter la redondance, car il y a forcément un moment ou une des copies ne sera pas modifié par la suite. Appliquez le concept "DRY" (Don't Repeat Yourself. Dry signifie également sec).

B.3 SOLID (Single responsiblity, Open Principle, …)

Les bonnes pratiques de codage consistent donc à rendre votre programme solide. C'est aussi un moyen mnémotechnique pour…

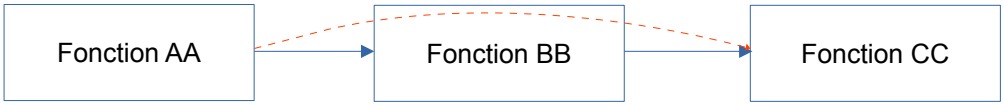
- **S**ingle Responsibility principle : une classe ne doit avoir qu'un domaine de responsabilité. Si vous avez besoin de créer de nouvelles fonctions, assurez-vous de respecter ce principe ! Ainsi, une évolution future ne touchera qu'une classe à la fois.
- **O**pen/closed principle : une classe doit être ouverte aux évolutions mais fermée aux modifications. Pour respecter ce principe, utilisez des portées privées pour vos attributs et publiques pour vos méthodes accessibles à l'extérieur.
- **L**iskov substitution principle : ce principe indique que lors des tests sur des classes héritées, les assertions doivent s'appliquer de la même manière que celles de la classe mère. L'exemple du principe de Liskov est généralement une classe carrée qui hérite d'une classe rectangle : dans la réalité, il faudra ré-écrire les setters, car dans un carré, la largeur est égale à la hauteur.
- **I**nterface Segregation principle : l'idée est de créer des interfaces qui n'implémentent pas ce qui n'est pas nécessaire. Si vous avez plusieurs rôles, il devrait y avoir autant de fenêtres pour chaque fonction similaire. Cela évite qu'un hacker puisse comprendre facilement le fonctionnement de l'application et limite les risques de dépendances.
- **D**ependency Inversion Principle : plutôt que d'utiliser un couplage fort, il est préférable d'utiliser une interface (une classe de type interface), car elle définit la syntaxe mais laisse les développeurs créer le contenu indépendamment.

Pour aller plus loin, vous pouvez consulter l'article de William Durand.

B.4 Loi de Déméter (LoD)

La loi de Déméter adresse particulièrement les portées des fonctions, pour éviter des intrications trop importantes.

Par exemple, une fonction AA peut utiliser une fonction BB. La fonction BB peut utiliser une fonction CC ; On évitera que la fonction AA puisse interagir directement avec la fonction CC. En effet, la fonction CC dépendrait des deux autres fonctions, ce qui rendrait son développement plus complexe par la suite.



Il s'agit d'une approche ascendante de la programmation objet.

B.5 YAGNI (You Aren't Gonna Need It)

Ce principe peut se traduire par "Vous n'en aurez pas besoin". L'objectif est de limiter le développement aux seules fonctions nécessaires et de ne pas trop anticiper sur les besoins futurs : cela complexifie le développement et augmente les coûts sans garantir un gain plus tard.

C Gestion de la qualité du code avec SonarLint

Le codeur devrait suivre un certain nombre de règles à la lettre et surtout, s'engager à les suivre par respect de ces collègues de travail.

Cependant, bien connaître son environnement de travail (IDE, RAD) et en connaître les bugs, les défaillances et les non-sens peut s'avérer chronophage.

Il existe donc des outils de vérification de la qualité des lignes écrites dans un programme, contenant de nombreuses règles, afin que le programmeur soit avertit des risques de son code.

SonarLint fait partie de ces outils. C'est une version locale de SonarQube qui est un outil plus puissant mais qui s'adresse aux gestionnaires qualités sur de gros projets (les gestionnaires peuvent créer, valider ou supprimer des règles).

C.1 Exemple de règles

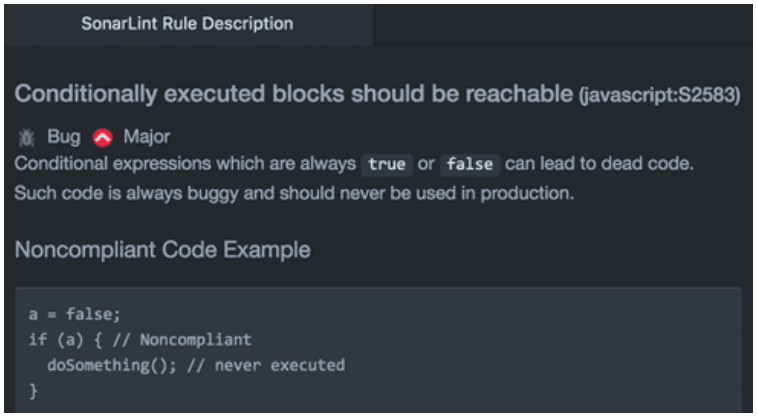
Il existe plusieurs niveaux de règles :

- vulnérabilités
- bugs
- code "nauséabond"
- zone de risque en sécurité

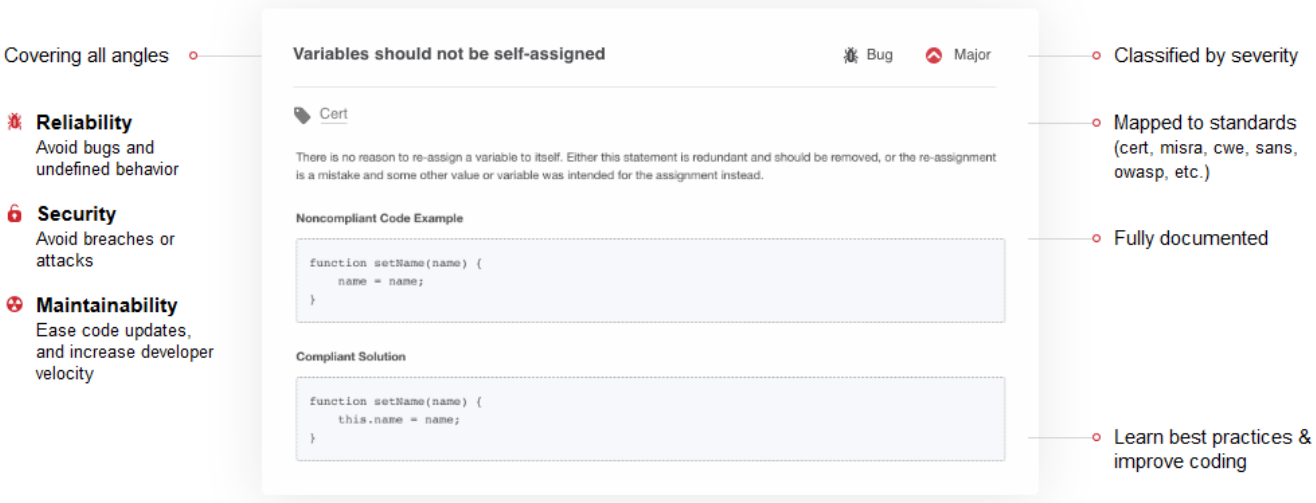
Selon le niveau, l'impact dans le programme sera plus ou moins important. La société SonarSource propose donc des règles classées par thème et par langage.

L'exemple JavaScript ci-contre paraît évident, car les lignes sont proches et permettent de voir l'erreur. Cependant, dans un programme de plusieurs centaines de lignes, ce cas peut se retrouver facilement.

La variable **a** étant toujours fausse, le code dans la condition ne sera **jamais** exécuté.



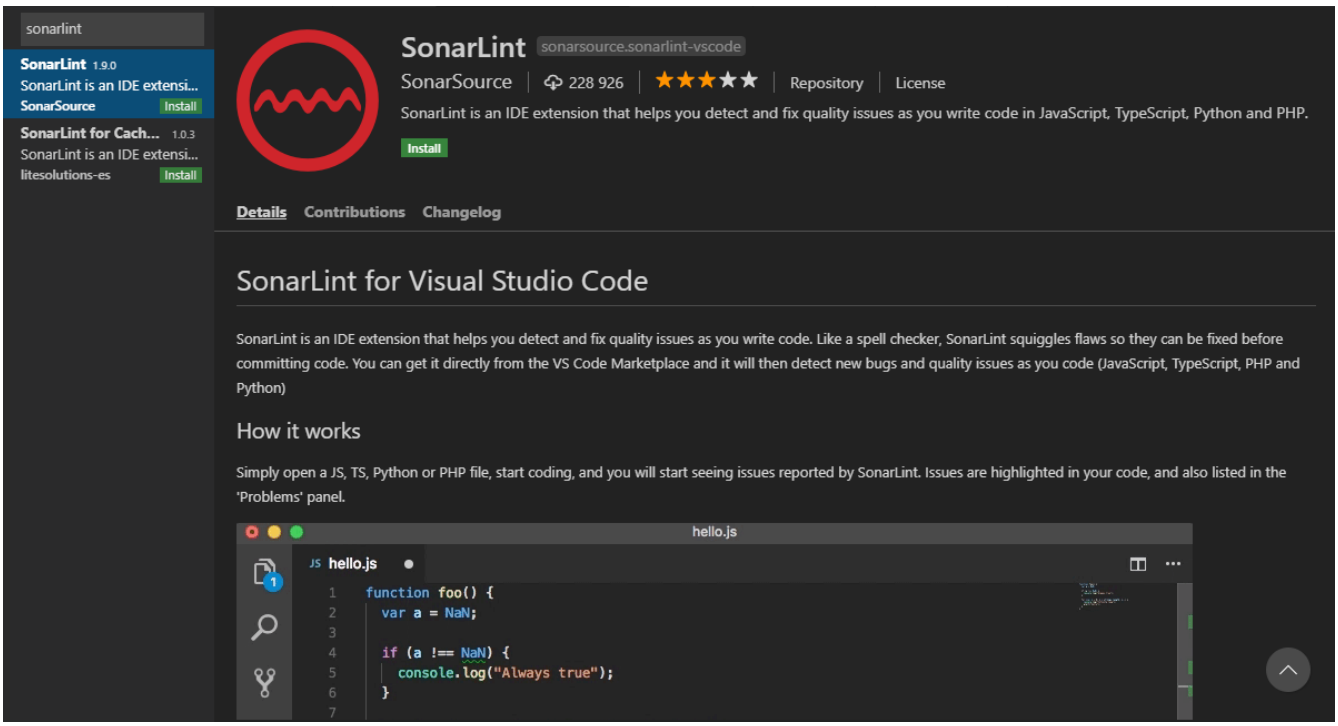
Voici comment lire les fiches de chaque règle :



C.2 Installer SonarLint

L'extension SonarLint se trouve dans le "market place" de visual Studio :

https://marketplace.visualstudio.com/items?itemName=SonarSource.SonarLintforVisualStudio2017



Cliquez sur le bouton vert [Install] et l'extension est prête à fonctionner.

Dans Visual Studio 2017+ le fonctionnement est similaire. Dans un nouveau projet en mode console () tapez le code suivant :

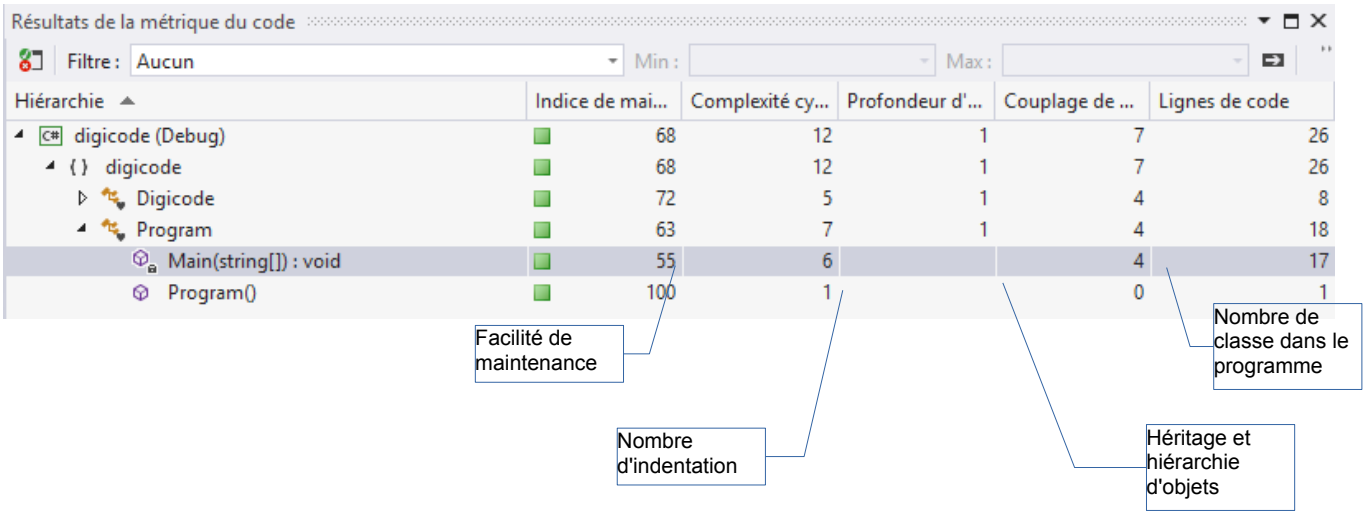
```
using System;
namespace ConsoleApp1 {
    class Program {
        static void Main(string[] args) {
            bool a = false;
            if (a) {
                Console.WriteLine("A est VRAI");
        }
    }
}
```

Si SonarLint fonctionne correctement, vous devriez voir le 'a' souligné en **vert**, dans la condition, et en passant la souris dessus, un message devrait vous avertir du risque dans ce code.

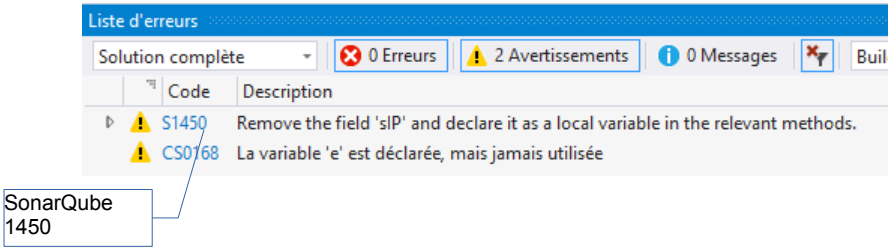
Les erreurs soulignées en **rouge**, sont signalées par l'IDE Visual Studio lui-même (ce n'est donc pas SonarLint, la preuve est que le programme **ne peut pas être compilé**).

Vous pouvez également lancer l'analyse du code :

Dans le menu **Analyser > Calculer les métriques du code > pour la solution**



De plus, vous pouvez aussi cliquer sur le lien donné dans la liste d'erreurs en bas de l'IDE :



C.3 Mise en œuvre SonarLint avec Visual Studio

Pour valider que SonarLint fonctionne bien, mais aussi que vous comprenez les potentielles erreurs qu'il détecte, corrigez le code suivant et rédigez un document PDF avec 3 colonnes : la première contient le numéro de lignes, la deuxième contient l'origine de l'erreur (IDE ou SonarLint) et la troisième est votre justification de l'erreur.

Réfléchissez et ne donnez pas le message d'erreur écrit bêtement : votre analyse compte.

Exemple :

N° ligne	IDE / SonarLint	Explications personnelles

Le code est sur la page suivante...

Voici le code en C# :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;

namespace digicode {
    class Digicode {
        /// <summary>
        /// La méthode SaisieChiffre permet de travailler sur un retour clavier
uniquement numérique (ou car. Escape)
        /// </summary>
        /// <param name="ch">Cette chaine permet d'afficher un texte avant la
saisie</param>
        /// <returns></returns>
        public char SaisieChiffre(String ch)
      {
          ConsoleKeyInfo saisie;
            Console.WriteLine(ch + " entre 0 et 9 ou touche [Echap]");
            do {
                saisie = Console.ReadKey(true);
                // Console.WriteLine("[" + saisie.KeyChar + "] soit la touche " +
saisie.Key);
                Debug.WriteLine("[" + saisie.KeyChar + "] soit la touche " +
saisie.Key);
            } while (saisie.Key != ConsoleKey.Escape && (saisie.KeyChar > 57 ||
saisie.KeyChar < 48));
            return saisie.KeyChar;
            return 'b';
        }
    }
    class Program {
        static void Main(string[] args)
      {
            Digicode monCode = new Digicode();
            int cpt;
            int[] caseCode = new int[4];
            char toucheCode;
            do {
                do {
                 cpt++;
                    toucheCode = monCode.SaisieChiffre("Code " +
Convert.ToString(cpt));
                 caseCode[cpt] == toucheCode;
                } while (toucheCode != 27 || cpt < 5)
                if (cpt < 5) {
                    Console.WriteLine("Bye!");
                 Console.RedKey();
                 System.Environment.Exit(0);
                } else {
                    Console.Write("Le code tapé est ");
                    for (int index=0; index < 5; index++) {
```

```
                        Console.Write(Convert.ToString(casCode[indx]);
                    }
                }
            } while (true != false);
        }
    }
}
}
```

Approfondissez vos connaissances. Allez sur le site SonarSource↗ et répondez aux questions suivantes :

Question	Réponse
Combien de règles existent pour JavaScript, concernant la sécurité ?	
En Python, par quoi faut-il remplacer le test d'inégalité <> ? (aide : Code Smell)	
Que recommande cette règle ? https://rules.sonarsource.com/csharp/RSPEC-1450	
Pourquoi cette règle est placée dans la catégorie "bug" ? https://rules.sonarsource.com/html/type/Bug/RSPEC-1100	
Quel langage contient le plus de règles ? Combien ?	

Désormais, appliquez ces règles dans vos développements.

D Commentaires et documentation

Outre la qualité du code, il est important de développer en ayant en tête les règles de bonnes pratiques. Parce que vous êtes en cours et que les projets sont rarement réels, cela peut induire de mauvais réflexes. Il faut bannir ce comportement et faire le choix d'une programmation vertueuse.

D.1 Les commentaires

Un code sans commentaires sera inutilisable après quelques semaines sans travail dessus. Personne d'autre que le développeur ne peut les rédiger. Il faut considérer 3 sortes de commentaires :

D.1.1 Entêtes de fichiers

Situé en début de fichier, il doit décrire ce que contient le code à suivre, les dates d'interventions, le(s) auteur(s) et de manière claire, à quoi il sert.

```
/* Palindrome.
 * cette application permet de vérifier si une phrase contient un palindrome.
 * Les caractères spéciaux sont éliminés, les accents transformés afin de pouvoir
 * appliquer une inversion de chaine et comparer le résultat.
 *
 * Auteur : David ROUMANET
 * Date   : 28/06/2019
 *
 */

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

D.1.2 Entêtes de fonctions

Les commentaires d'entête des fonctions (ou méthodes) sert pour expliquer les paramètres d'entrées et de sorties. Notamment les plages de valeurs par exemple.

```
static float Moyenne(float a, float b)
{
    // Moyenne renvoie la moyenne entre deux nombres
    // a, b : valeur numérique > 0 et < 20
    return (a + b) / 2;
}
```

D.1.3 Détails d'opérations complexes

Lorsqu'une opération est compliquée à lire ou comprendre rapidement.

```
Pmax = Power(Max(a,b), "w ")+Math.Cos(anglePhy)+"°"; // donne la puissance et le déphasage
```

D.1.4 Javadoc en Java et XML en C#

Les développeurs ont rapidement compris qu'il était contre-productif de rédiger des commentaires dans le code source, puis de rédiger une documentation traditionnelle (dans une suite bureautique).

On trouve donc une approche qui consiste à utiliser des mots-clés dans le code et utiliser un programme qui rédigera la documentation automatiquement. C'est le principe de Javadoc.

D.1.4.a Javadoc

Sans entrer dans les détails, Javadoc est un outil qui utilise les commentaires dans le programme pour décrire les méthodes et classes. Voici un exemple de la syntaxe Javadoc, détaillée sur Wikipedia↗

```
/**
 * Valide un mouvement de jeu d'Echecs.
 * @param leDepuisFile   abscisse de la pièce à déplacer
 * @param leDepuisRangée ordonnée de la pièce à déplacer
 * @param leVersFile     abscisse de la case de destination
 * @param leVersRangée   ordonnée de la case de destination
 * @return vrai (true) si le mouvement d'échec est valide ou faux (false) sinon
 */
boolean estUnDeplacementValide(int leDepuisFile, int leDepuisRangée, int leVersFile, int leVersRangée)
```

Les différents mots-clés sont précédés du caractère **@**, comme **@author**, **@version**, **@param** (décrit un paramètre de la méthode), **@return** (décrit la valeur de retour), **@exception**...

D.1.4.b XML documentation

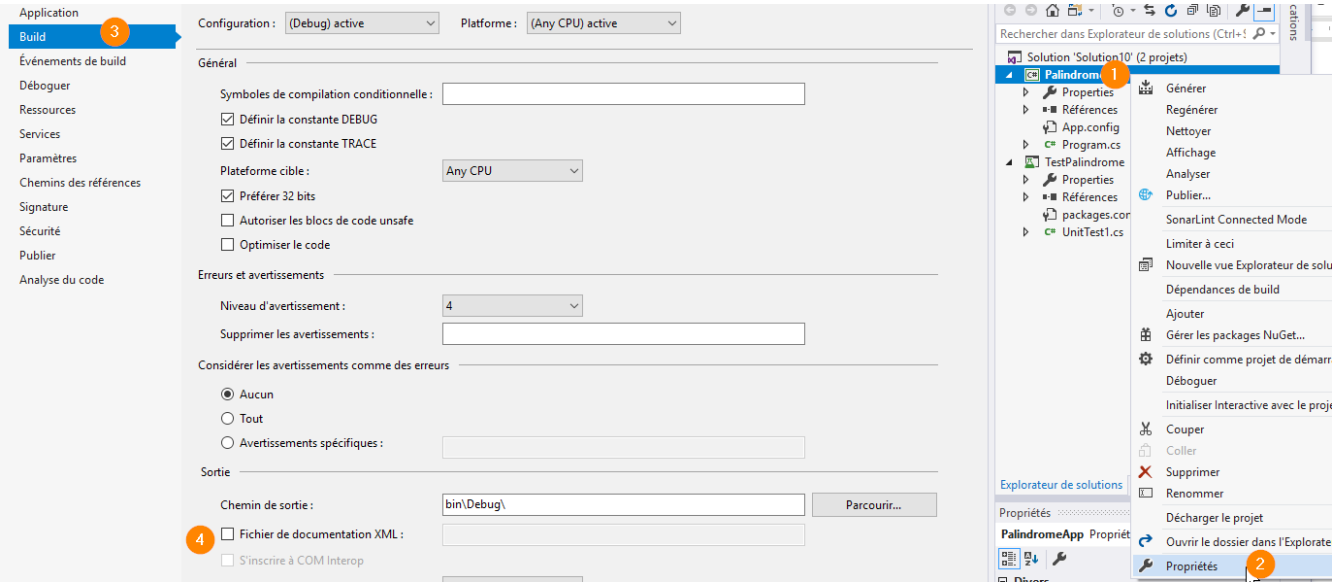
L'équivalent C# de Javadoc est la création d'un fichier au format XML. Les commentaires utilisent un triple **/** suivi de balises XML (pour rappel, encadrée par **<** et **>**). sur developpez.com↗, Vincent LAINE montre le fonctionnement de cette syntaxe.

```
/// <summary>
///  Description de la classe. on donne la fonction ainsi que les particularités
/// </summary>
public class MaClasse
{
    /// <summary>
    /// Une méthode qui ne fait rien ;-)
    /// </summary>
    /// <param name=a>parametre qui ne sert à rien du tout.</param>
    /// <returns>Valeur pnon modifiée car la fonction ne fait rien .</returns>
    public int UneMethode(int a)
    {
        return a;
    }
}
```

Les principales étiquettes (tags) sont (complétez le cours)

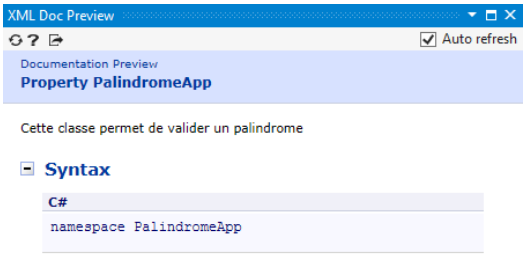
<summary>, <param name>, <value>, <remarks>, <exception>, <example>, <seealso>...

Dans Visual Studio, le fait de taper /// au-dessus d'une classe ou d'une méthode prépare les étiquettes utiles automatiquement. Pour créer la documentation il faut activer la génération de documentation, comme montré dans la page suivante :



clic droit sur le projet, sélectionner "propriétés", dans l'onglet [Build] activer la case à cocher pour "fichier de documentation XML".

Une fois le fichier XML disponible, il est possible d'utiliser d'autres outils comme Ndoc pour générer la documentation au format HTML.



Sous Visual Studio, une extension pratique est DocPreview.

Une fois activée, il faut aller dans le menu **Affichage** > **Autres fenêtres** > **XML Documentation Preview**

D.1.5 Autres outils de commentaires

Il existe d'autres outils, relativement indépendant de l'éditeur. Doxygen↗ en fait partie, comme ceux référencés sur la page de comparaison Wikipedia↗. Leurs syntaxes utilisent les mêmes principes, cependant, Doxygen reste l'un des plus pratiques car disponible sous Windows, Linux et Mac.

Extension Doxygen pour :

- Visual Studio Code↗
- Eclipse↗
- Netbeans↗
- etc.

Pour Visual Studio, il est malheureusement nécessaire d'installer Doxygen puis l'extension 1-Click Docs↗ pour pouvoir compiler la documentation.

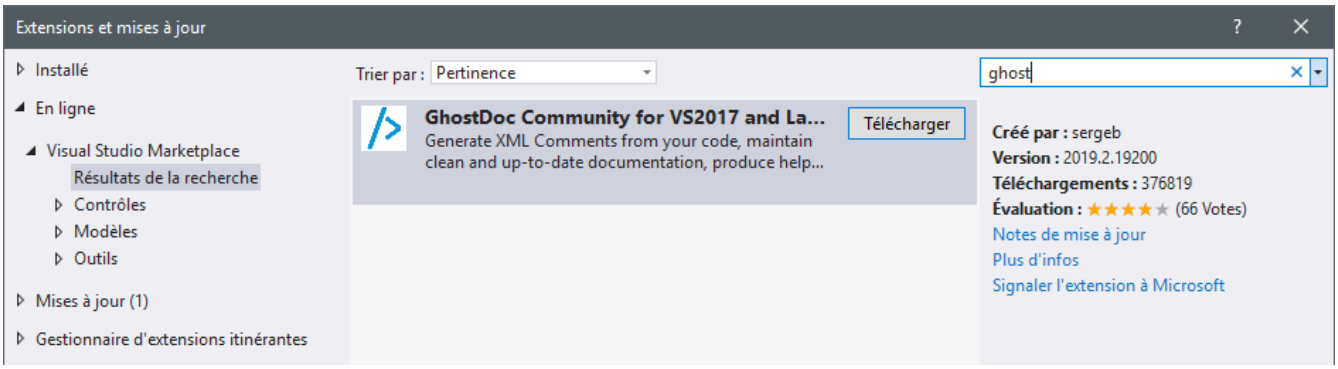
D.1.6 Génération de documentation

Pour générer une documentation à partir du format natif de Microsoft (XML), vous pouvez utiliser Le générateur de documentation HTML de Microsoft et l'extension GhostDoc (Community).

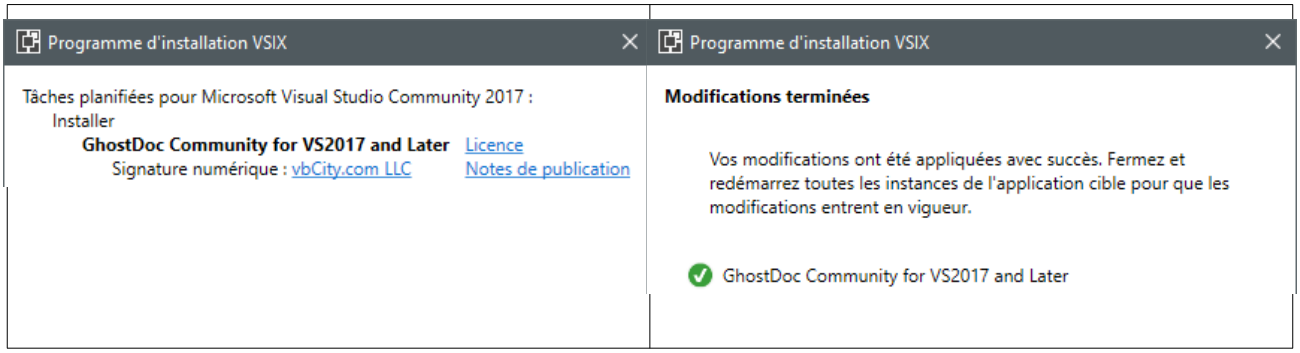
D.1.6.a Installation de l'extension Ghostdoc

Recherchez l'extension Ghostdoc dans le menu des extensions de Visual Studio 2017

https://marketplace.visualstudio.com/items?itemName=sergeb.GhostDoc



Redémarrez l'IDE pour que l'installation se fasse.



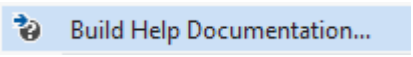
D.1.6.b Utilisation de GhostDoc

Chargez un projet ou une solution et commencez à commenter le code avec /// (ce qui déclenche une macro Visual Studio de préparation devant Ia méthode)

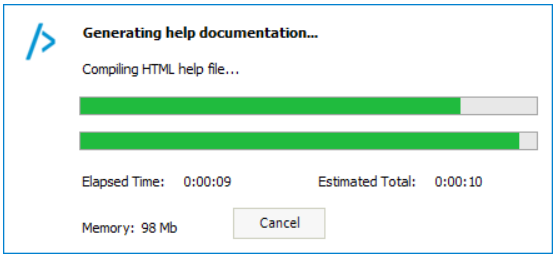
Dans le menu **Outils > GhostDoc** choisissez "**Document This...**"

D.1.6.c Génération d'une documentation

Dans le menu **Outils > ghostDoc**, cliquez sur "**Build Help Documentation...**"

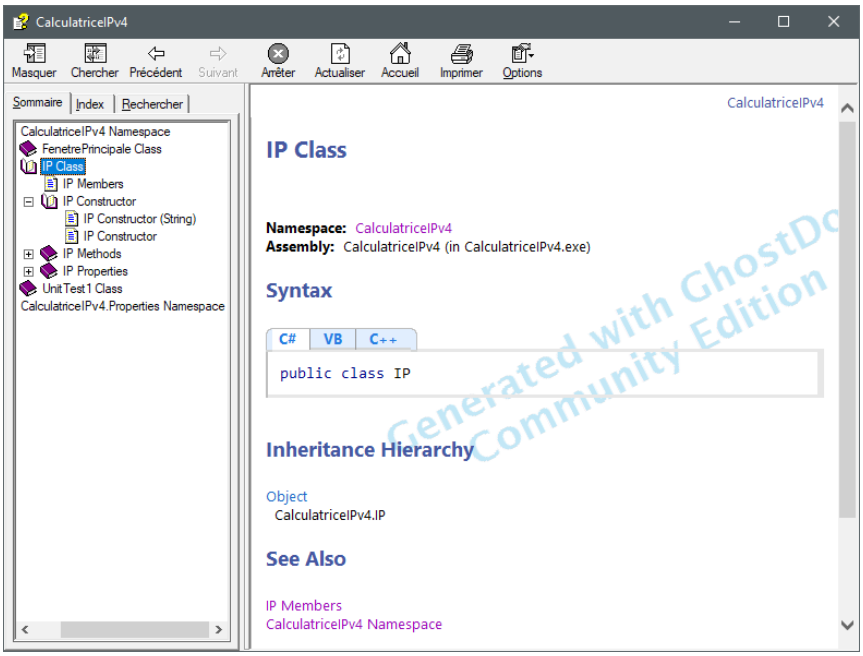


La version Community étant très restrictive, vous ne pouvez configurer que le chemin du fichier d'aide. L'outil génère la documentation et les liens automatiquement :



Vous obtenez un fichier lisible depuis n'importe quelle machine Windows.

Le fichier devrait ressembler à la capture suivante :



E Développement par les tests unitaires

Les tests unitaires permettent de développer d'abord la méthode de test, puis le codeur peut développer sa fonction et valider son fonctionnement.

On parle donc de **TDD**[1] (Test Driven Development) car le développeur créé ou modifie du code uniquement en cas d'échec. Évidemment, les tests unitaires peuvent être utilisés aussi à posteriori.

Bien entendu, pour de petites applications, cette méthode se révèle lourde, mais sur des projets où plusieurs développeurs collaborent, cela évite des régressions de code lourdes à déboguer. Il est fortement recommandé de prévoir les résultats attendus par une fonction, de rédiger les tests unitaires puis de développer la fonction.

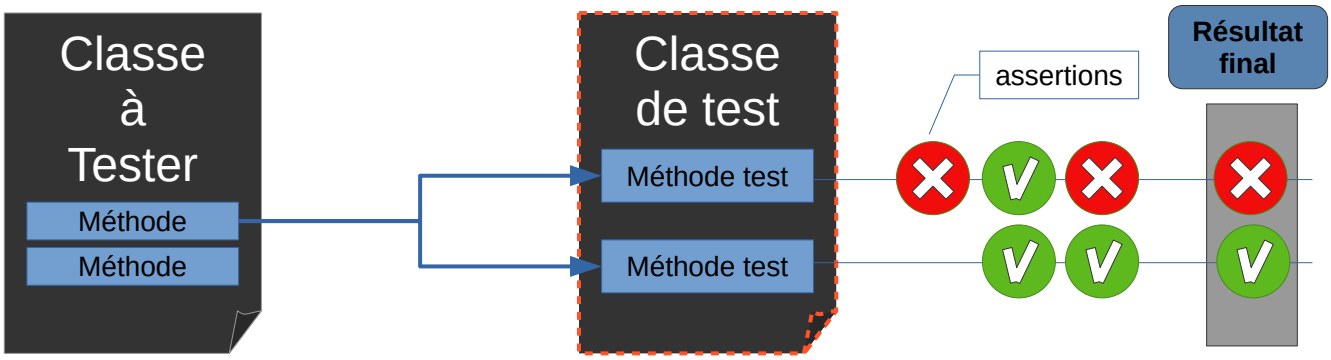
E.1 Fonctionnement

L'utilisation des tests unitaires se fait sur une classe (qu'on peut instancier). Dans la pratique, il s'agit d'un ensemble de fonctions qui appellent la fonction à tester et vérifient le résultat attendu.

Il faut donc instancier la classe à tester, puis tester la fonction avec une assertion. Cela signifie donc que la fonction testée doit renvoyer une valeur. En fonction de cette valeur, les assertions sont les suivantes :

assertion	signification
.isFalse()	l'expression doit renvoyer False pour que le test réussisse
.isTrue()	l'expression doit renvoyer True pour que le test réussisse
.isNotNull()	l'expression ne doit pas être nulle pour que le test réussisse
.isNull()	l'expression doit être nulle pour que le test réussisse
.isEquals()	l'expression doit être égale à un nombre pour que le test réussisse

L'IDE effectue chaque appel et vérifie que le résultat est cohérent avec celui attendu : si ce n'est pas le cas, le test échoue.



1 http://bruno-orsier.developpez.com/tutoriels/TDD/pentaminos/