Les normes de qualité du code

V1.0

PROJET:

Plateforme d'identification,valorisation et formation de l'artisanat marocain

Module: Conduite de projet

ENCADRÉ PAR : M.RAKRAK Said

MEMBRES DE L'ENTREPRISE SECURE STEP:

- KHOUDRAJI OUIAM
- EL ANSARI AMINA
- EL MOUMNAOUI KAOUTAR
- ELORF LAHCEN

Deuxième année du cycle d'ingénieur :Ingénierie en Réseaux et Systèmes d'Information.

Année universitaire: 2022-2023

VERSION	DATE	MODIFICATIONS
1.0	Le 11/03/2023	

SOMMAIRE :

INTRODUCTION :

La qualité du code est un aspect clé dans le développement de logiciels de qualité. Les normes de qualité de code peuvent aider les développeurs à produire du code qui est plus lisible, plus maintenable, plus testable, plus sécurisé et plus performant.

Les normes de qualité de code ne sont pas seulement importantes pour les développeurs, mais aussi pour les utilisateurs finaux de l'application. Les applications qui ont été développées avec des normes de qualité de code élevées sont plus fiables, ont moins de bogues, sont plus facilement maintenables et peuvent être mises à jour plus facilement.

Dans ce document, nous allons passer en revue plusieurs normes de qualité de code qui ont été largement acceptées par la communauté des développeurs. Ces normes peuvent inclure des pratiques de codage telles que l'utilisation de noms de variables significatifs, la documentation appropriée du code, la gestion des erreurs, la séparation de la logique de l'interface utilisateur, l'utilisation de tests unitaires, et bien d'autres encore.

NORMES DE QUALITÉS DU CODE :

• Les variables

- Les variables sont généralement nommées en utilisant une notation camelCase (la première lettre de chaque mot est en majuscule, sauf pour la première lettre du premier mot).
- Les noms de variables doivent être descriptifs et utiliser des noms de mots complets plutôt que des abréviations ou des acronymes

```
#Exemple de déclaration des variables:
firstName = "firstName"
lastName = "lastName"
```

• Les constantes

- *Les constantes sont généralement nommées en utilisant une notation UPPER_CASE_WITH_UNDERSCORES (tout en majuscules, les mots séparés par des underscores)*
- *Évitez d'utiliser des caractères spéciaux tels que les tirets, les espaces ou les caractères de ponctuation.*
- *Les noms de constantes doivent être descriptifs et utiliser des noms de mots complets plutôt que des abréviations ou des acronymes.*

```
#Exemple de déclaration des constantes:
MAX_SIZE = 100
PI = 3,14
```

• Les fonctions

- *Les fonctions sont généralement nommées en utilisant une notation camelCase, tout comme les variables.*
- *Les noms de fonctions doivent être descriptifs et utiliser des noms de mots complets plutôt que des abréviations ou des acronymes.*
- *Les noms de fonctions doivent également commencer par un verbe d'action qui décrit ce que fait la fonction.*
- *Chaque fonction doit avoir un commentaire qui explique son comportement*

```
#Exemple des fonctions:
    def calculateTotal():
        total = 100
        return total
```

- **Les classes**
  - Utilisez des noms descriptifs pour vos classes qui reflètent leur objectif ou leur comportement.
  - Les classes sont généralement nommées en utilisant une notation PascalCase (la première lettre de chaque mot est en majuscule, sans espaces ni underscores).
  - Les noms de classes doivent être descriptifs et utiliser des noms de mots complets plutôt que des abréviations ou des acronymes.
  - Les noms de classes doivent également commencer par un nom qui décrit la nature de la classe.

```
22  #Exemple des classes:
23  class Car:
24      #statement1 ...
25
26      #statement2 ...
27
28      #statement3 ...
```

- **Les commentaires**
  - Évitez les commentaires redondants
  - Les commentaires doivent être suffisamment brefs pour être facilement compréhensibles, mais suffisamment détaillés pour fournir des informations utiles.
  - les commentaires de type **''' ... '''** ou **/* ... */** doivent être au début de chaque fichier du code source on mettant dedans une petite description qui montre ce qui a dans le fichier
  - **#** ou **//** pour commenter une ligne
  - **#?** ou **//?** pour laisser une question aux autres développeur
  - **#!** ou **//!** pour spécifier une partie de code à revoir
  - Remarque:
    - **#** et **''' ... '''** pour python
    - **//** et **/* ... */** pour les autres langages

GIT & GITHUB :

La bonne gestion de qualité du code est une étape cruciale dans notre projet, d'où la nécessité d'utiliser une plateforme web d'hébergement de code source **Github** qui utilise **Git** pour la gestion de versions.

• Les branches

Création d'une branche pour chaque fonctionnalité en respectant la convention suivante pour le nom:

- **feature/nouvelle-fonctionnalite**
- Utilisez des noms en minuscules pour les branches.
- Utilisez des tirets (-) pour séparer les mots dans le nom de la branche.
- Utilisez des noms courts et descriptifs qui indiquent clairement l'objectif de la branche.
- Évitez d'utiliser des caractères spéciaux ou des espaces dans le nom de la branche.
- Utilisez des préfixes pour identifier le type de branche :
  - "**feature/**" pour une fonctionnalité,
  - "**bugfix/**" pour une correction de bug,
  - "**hotfix/**" pour une correction urgente, etc.).
- Évitez d'utiliser des noms génériques tels que "master", "develop", ou "test".

• Les commits

les commits doivent respecter les normes suivantes:

- le message de chaque commit doit être descriptif, il ne faut pas dépasser 50 caractères.
- Les types de commit peuvent être feat (nouvelle fonctionnalité), fix (correction de bogue), docs (documentation), style (mise en forme), refact (refactorisation du code), perf (optimisation des performances), test (ajout ou modification de tests), ou build (mise à jour des scripts de build).
- le message de la commit doit être sous la forme suivante:
  - "type de commit: description courte "

- **Les permissions**
  - chaque membre ne peut faire un push que vers sa branche dans le dépôt distant
  - la validation de chaque nouvelle fonctionnalité nécessite un Pull-request
  - la branche "master" est protégé seul le chef de qualité du code qui peut mixer les branches après la validation finale selon les normes du premier axe.

LES OUTILS DU TEST

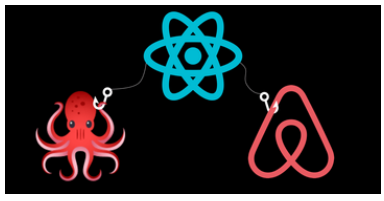
Les tests sont une pratique courante en développement logiciel qui consiste à exécuter des morceaux de code de manière automatique et systématique pour vérifier s'ils fonctionnent correctement. Les tests sont essentiels pour garantir la qualité du code et prévenir les bugs, les erreurs et les régressions.

Pour Java Script :

Il existe de nombreux outils de test disponibles pour **ReactJS**, certains d'entre eux étant spécifiquement conçus pour tester des fonctionnalités **React** spécifiques. Voici quelques-uns des outils les plus populaires :

- **Jest :** Jest est un framework de test JavaScript développé par Facebook. Il est particulièrement adapté pour tester des applications React, car il offre un support intégré pour les tests de composants React et utilise un système de snapshot pour s'assurer que les changements apportés à votre application ne cassent pas les tests existants.

- **React Testing Library :** La bibliothèque de test React Testing Library est conçue pour tester les applications React en simulant le comportement des utilisateurs et en vérifiant que l'interface utilisateur fonctionne comme prévu. Elle utilise une approche centrée sur l'utilisateur et se concentre sur les tests d'accessibilité, ce qui peut être très utile pour s'assurer que votre application est utilisable par tous.



LES OUTILS DU TEST

Exemple :

Tester un composant React "**Button**" en utilisant **JTes**t

```jsx
import React from 'react';

const Button = ({ onClick, text }) => (
  <button onClick={onClick}>{text}</button>
);

export default Button;
```

Le teste:

```jsx
import React from 'react';
import { render, fireEvent } from '@testing-library/react';
import Button from './Button';

test('renders button with correct text', () => {
  const onClick = jest.fn();
  const text = 'Click me';
  const { getByText } = render(<Button onClick={onClick} text={text} />);
  const buttonElement = getByText(text);
  expect(buttonElement).toBeInTheDocument();
});

test('calls onClick prop when button is clicked', () => {
    const onClick = jest.fn();
    const text = 'Click me';
    const { getByText } = render(<Button onClick={onClick} text={text} />);
    const buttonElement = getByText(text);
    fireEvent.click(buttonElement);
    expect(onClick).toHaveBeenCalledTimes(1);
  });
```

Pour Python :

- **unittest :**

C'est un framework intégré à Python qui permet d'écrire des tests unitaires. Il offre une variété d'assertions pour vérifier les résultats attendus de votre code. Vous pouvez également utiliser des annotations pour indiquer les tests que vous souhaitez exécuter.

- **coverage**

C'est un module qui permet de mesurer la couverture de code de vos tests. Il vous permet de savoir quelles parties de votre code ont été testées et lesquelles ne l'ont pas été.

Exemple :

```python
def division(a, b):
    if b == 0:
        raise ZeroDivisionError("division par zéro")
    return a / b
#test de la fonctionnalité division(a,b)
import unittest

class TestDivision(unittest.TestCase):
    def test_division(self):
        result = division(10, 2)
        self.assertEqual(result, 5)

    def test_division_by_zero(self):
        with self.assertRaises(ZeroDivisionError):
            division(10, 0)
```

Conclusion :

En conclusion, pour assurer la fiabilité, la stabilité et la maintenabilité d'un logiciel, il est primordial de se concentrer sur la qualité du code. Cela implique de respecter les critères tels que la lisibilité, la modularité, la performance, la sécurité, la maintenabilité et la testabilité. De plus, l'utilisation d'un outil de gestion de version tel que Git permet d'optimiser le travail en équipe, de faciliter la collaboration et de garantir l'intégrité du code.