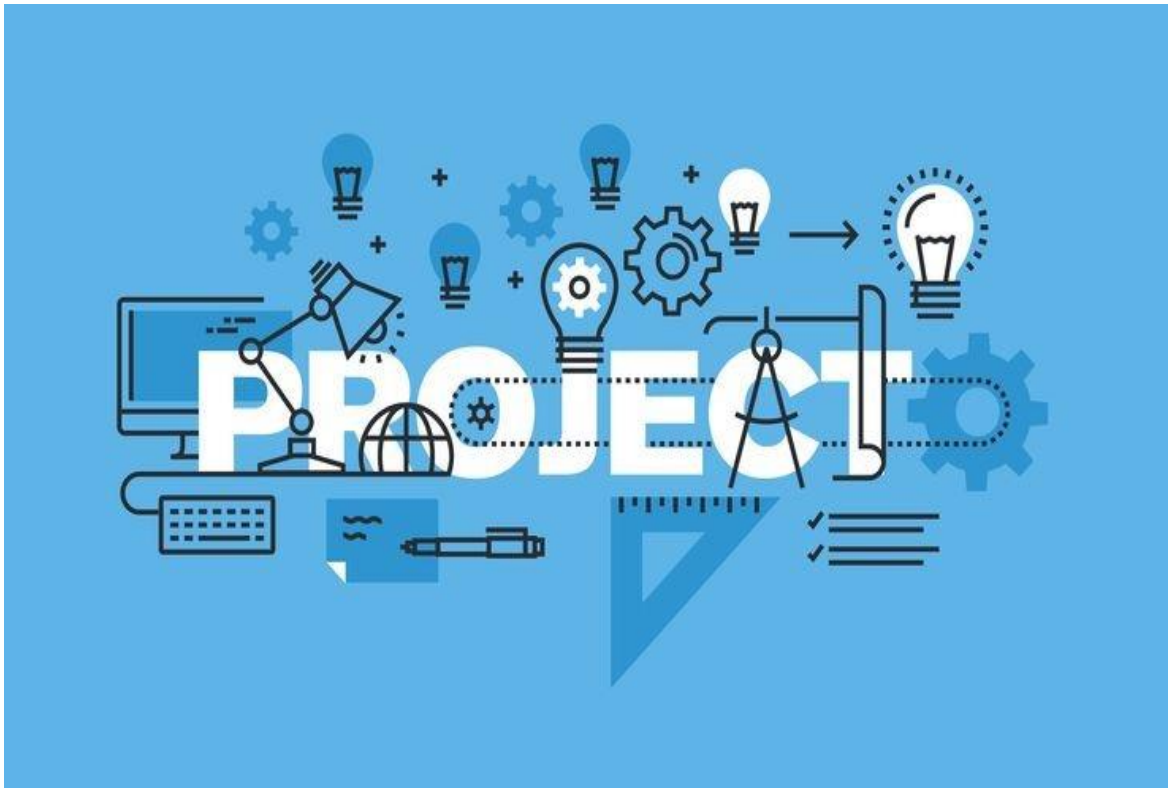

Rapport de Test Mesure et Qualité du Code d'un Système de gestion de projet

Membre du groupe :

MAMIE AWA WATT,

LATYR OMAR DIEDHIOU,

SEYDINA ISSA LAYE DIAGNE

Introduction

Dans le cadre du développement de notre application, il est essentiel de garantir la qualité du code afin d'assurer son bon fonctionnement, sa maintenabilité et sa robustesse. Pour ce faire, nous avons mis en place un processus rigoureux de test, de mesure et d'analyse de la qualité du code.

La première partie de ce processus consiste à rédiger des tests exhaustifs pour chaque méthode implémentée dans notre code. Nous utilisons la bibliothèque unittest pour créer une suite de tests qui vérifie le comportement attendu de chaque fonctionnalité. Cela nous permet de détecter rapidement les erreurs de programmation et de nous assurer que notre application fonctionne comme prévu.

Ensuite, nous utilisons un ensemble d'outils d'analyse de qualité pour évaluer différents aspects de notre code. Ces outils comprennent flake8, pylint, mypy, coverage, vulture, Black, radon et pyflakes. Chaque outil fournit des informations précieuses sur la conformité aux conventions de codage, les erreurs de programmation, les problèmes de typage, la couverture de code, les variables inutilisées, la complexité cyclomatique et les erreurs de syntaxe.

En intégrant ces étapes d'analyse dans notre processus de développement, nous visons à garantir que notre code Python est de haute qualité, facilement maintenable et conforme aux meilleures pratiques de développement. Ce rapport détaillera les résultats de chaque analyse et les actions prises pour améliorer la qualité du code.

Analyse du code avec les outils de mesure de qualité :

1. Vérification de la conformité aux conventions de codage PEP 8 :
   Outil Utilisé : flake8

   Résultat :

   Figure 1 : Résultat après l'exécution de la commande flake8 main.py

   La Figure 1 montre qu'il y a deux erreurs E203 et E225 qui respectivement nous signale qu'il devrait y avoir un espace avant les ' : ' et aussi un espace avant et après chaque opérateur.

   Résultat après correction :

   Figure 1.1 : Résultat après correction

   La Figure 1.1 montre qu'il y a plus d'erreur dans le code

2. Identifications des erreurs de programmation et les conventions de codage non respectées :
   Outil Utilisé : pylint

   Résultat :

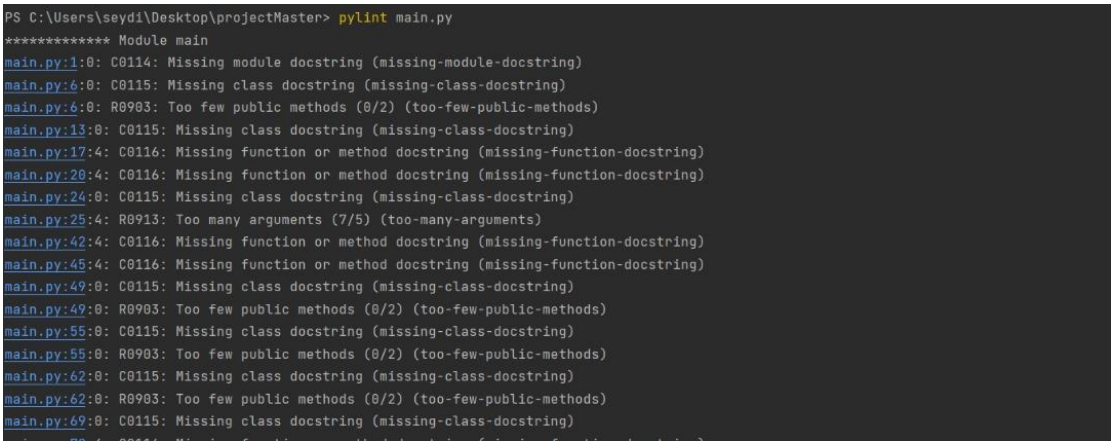

Figure 2 : Résultat après l'exécution de la commande pylint main.py

La Figure 2 montre qu'il y a les erreurs C0114, C0115, C0116 qui nous signale respectivement que les modules, les classes et les fonctions du code devrait être commenté et l'erreurs R0903 nous indiques que les classes doivent avoir au moins deux méthodes.

Résultat après correction :
L'erreur R0903 persiste parce que les méthodes n'ont pas été augmentées, conformément à ce qui était défini dans l'énoncé.

3. Vérification du typage statique et détection des erreurs de typage :

Outil Utilisé : mypy

Résultat :

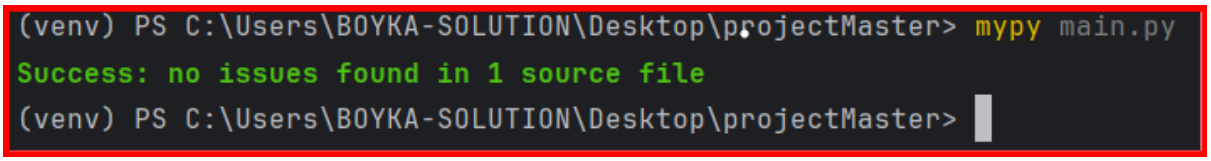

Figure 3 : Résultat après l'exécution de mypy

La Figure 3 montre que le typage statique et les erreurs de typage ne signalent aucune erreur.

4. Analyse de la couverture du code test :

Outil utilisé : coverage
Résultats :

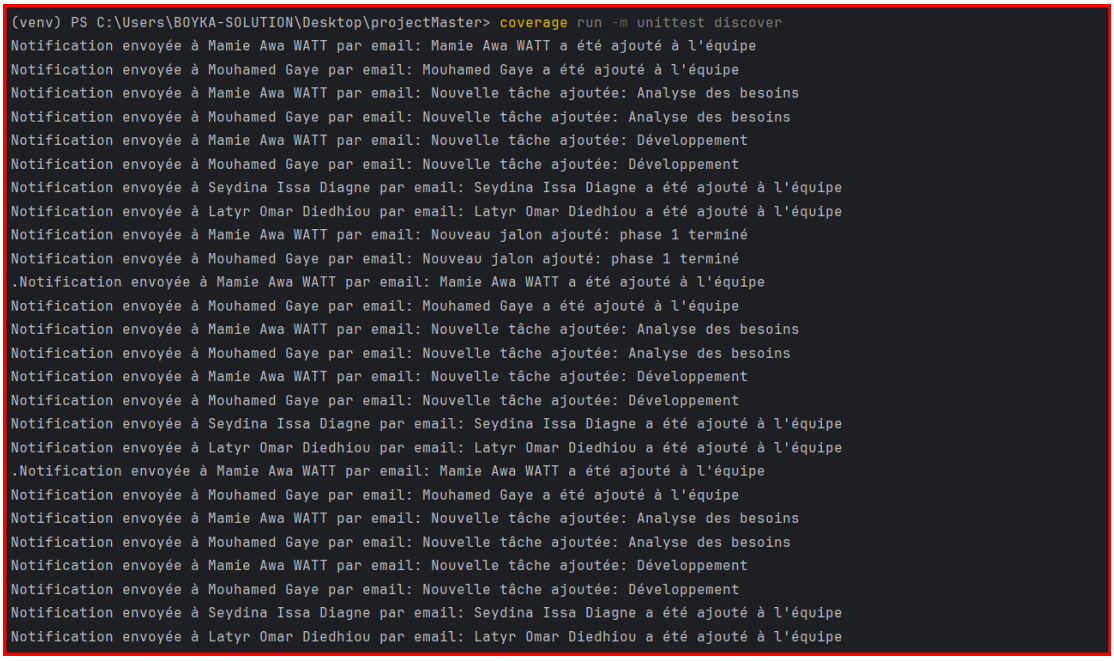

Figure 4 : Résultat après l'exécution de la commande coverage run -m unittest discover

La Figure 4 affiche le résultat après l'exécution automatique de tous les tests unitaires de la classe test : test_main.py.

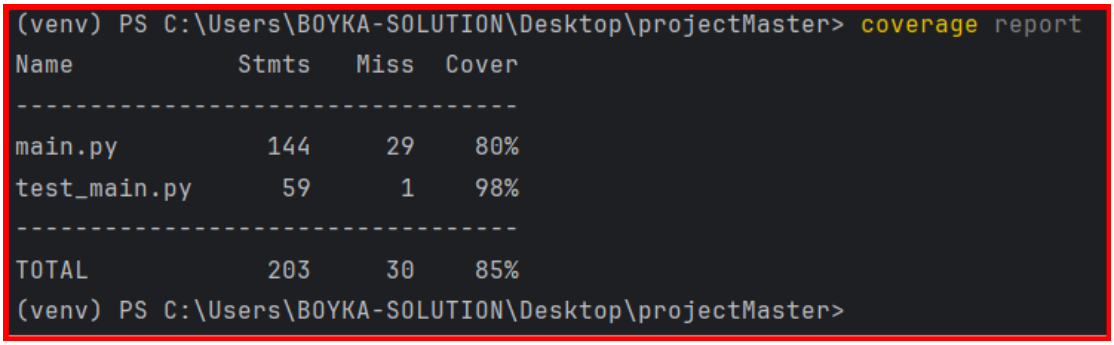

Figure 4.1 : Résultat après l'exécution de la commande coverage report

La Figure 4.1 montre certaines caractéristiques tel que :

Name : Nom du fichier source.

Stmts : Nombre total de déclarations dans le fichier.

Miss : Nombre de déclarations non couvertes par les tests.

Cover : Pourcentage de couverture des déclarations.

5. Détection des variables inutilisées :

Outils utilisés : vulture

Résultat :

```
(venv) PS C:\Users\BOYKA-SOLUTION\Desktop\projectMaster> vulture main.py
main.py:118: unused class 'SMSNotificationStrategy' (60% confidence)
```

Figure 5 : Résultat après l'exécution de la commande vulture main.py

La Figure 5 montre que la classe SMSNotificationStrategy qui n'est pas utilisé

```
(venv) PS C:\Users\BOYKA-SOLUTION\Desktop\projectMaster> vulture main.py
(venv) PS C:\Users\BOYKA-SOLUTION\Desktop\projectMaster>
```

Figure 5.1 : Résultat après correction

La Figure 5.1 montre que toutes les variables, classe, fonction ont été utilisés.

5. Reformater automatiquement le code selon les conventions PEP 8 :
   Outils utilisés : black
   Résultat :

```
(venv) PS C:\Users\BOYKA-SOLUTION\Desktop\projectMaster> black main.py
reformatted main.py

All done! ✨🍰 ✨
1 file reformatted.
```

Figure 6 : Résultat après l'exécution de la commande black main.py

La Figure 6 montre que le code a été bien formaté et qu'il respecte l'ensemble des règles de style cohérent.

6. Évaluation de la complexité cyclomatique et de la structuration globale du code :
   Outil utilisé : Radon
   Résultat sur la complexité cyclomatique :

Figure 7 : Résultat après l'exécution de la commande radon cc main.py -a

Interprétation des résultats du Figure 7 :

- M et C : Indiquent respectivement une méthode et une classe.
  - Chiffres après M ou C : Indiquent la ligne où la méthode ou la classe commence.
  - Nom de la méthode ou de la classe : Indique quelle méthode ou classe est analysée.
- Note de Complexité (A, B et C.) : Indique la complexité cyclomatique de la méthode ou de la classe :
  - A : Complexité faible, facile à comprendre et à maintenir.
  - B : Complexité modérée, légèrement plus complexe mais généralement acceptable.
  - C et au-delà : Complexité plus élevée, nécessite probablement une simplification.

Résultat sur la structuration globale du code :

```
(venv) PS C:\Users\BOYKA-SOLUTION\Desktop\projectMaster> radon hal main.py
main.py:
    h1: 2
    h2: 4
    N1: 2
    N2: 4
    vocabulary: 6
    length: 6
    calculated_length: 10.0
    volume: 15.509775004326936
    difficulty: 1.0
    effort: 15.509775004326936
    time: 0.861654166907052
    bugs: 0.005169925001442312
(venv) PS C:\Users\BOYKA-SOLUTION\Desktop\projectMaster>
```

Figure 7.1: Résultat après l'exécution de radon hal main.py

Interprétation des Résultats du Figure 7.1 :

- h1 : Nombre distinct d'opérateurs , 2.
- h2 : Nombre distinct d'opérandes ,4.
- N1 : Nombre total d'occurrences d'opérateurs, 2.
- N2 : Nombre total d'occurrences d'opérandes, 4.
- vocabulary : Vocabulaire du programme (h1 + h2), 6.
- length : Longueur du programme (N1 + N2), 6.
- calculated_length : Longueur calculée théorique du programme. C'est une estimation basée sur les opérateurs et les opérandes distincts, 10.0.
- volume: Volume du programme, calculé comme (length * log2(vocabulary)), 15.509775004326936.
- difficulty: Difficulté du programme, calculée comme (h1/2) * (N2/h2), 1.0.
- effort : Effort nécessaire pour comprendre ou écrire le programme, calculé comme (difficulty * volume), 15.509775004326936.
- time : Temps estimé pour écrire ou comprendre le programme, calculé comme (effort / 18) secondes, 0.861654166907052 secondes.
- bugs : Nombre estimé de bugs dans le programme, calculé comme (volume / 3000), 0.005169925001442312.

7. Vérification du code source pour les erreurs de syntaxe et les problèmes de style :

Outils utilisés : pyflakes

Résultat :

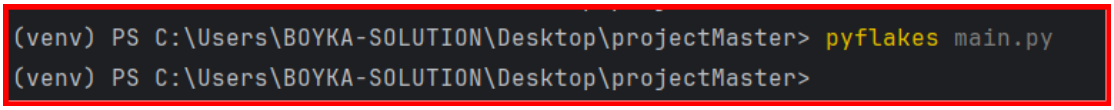

Figure 8 : Résultat après l'exécution de la commande pyflakes

La Figure 8 montre que la commande pyflakes n'a produit aucune sortie, cela signifie que Pyflakes n'a trouvé aucune erreur ou problème potentiel dans notre fichier main.py

Conclusion

La gestion de la qualité du code est une étape cruciale dans le processus de développement logiciel. À travers l'analyse détaillée effectuée avec différents outils, nous avons pu évaluer plusieurs aspects de notre code, allant de la conformité aux conventions de codage à la détection d'erreurs de programmation, en passant par la couverture de code, la complexité cyclomatique, et bien d'autres.

Les résultats de ces analyses ont permis d'identifier des zones d'amélioration et de prendre des mesures correctives pour garantir un code de qualité, facilement maintenable et conforme aux meilleures pratiques de développement. Par exemple, nous avons corrigé les erreurs de syntaxe et les problèmes de style détectés par flake8 et pyflakes, commenté les méthodes nécessaires pour respecter les conventions de codage selon pylint, et assuré une couverture de code satisfaisante grâce à l'exécution de tests unitaires.

Il est également important de noter que bien que notre projet ait obtenu une couverture de code de 85%, il est recommandé de viser une couverture encore plus élevée pour garantir la robustesse et la fiabilité du logiciel.

En conclusion, l'intégration de ces mesures de qualité dans notre processus de développement nous permet de maintenir un haut niveau de qualité du code tout au long du cycle de vie du projet, ce qui contribue à la satisfaction des utilisateurs et à la pérennité de l'application.