Les Métriques Orientées Objet

KARA Nabil, HADJ-ARAB Adel, BENALIA Mohamed

RACHEDI Abderrahmane, KECIRA Abderraouf , BECHAR Walid

Introduction :

La programmation orientée objet (POO) est une méthodologie de développement logiciel influente, mettant en avant les objets pour favoriser la modularité et la réutilisabilité du code. Dans ce contexte, l'évaluation des métriques orientées objet est devenue cruciale pour mesurer la qualité, la complexité et la performance des logiciels. Ce rapport se concentre sur l'exploration des métriques orientées objet, en mettant l'accent sur les patrons de conception et leur impact dans le développement logiciel. En plus des métriques traditionnelles, de nouveaux indicateurs ont été créés pour mieux appréhender les défis contemporains du développement logiciel. Il examine les différentes métriques, leur utilité pratique, et leur contribution à l'amélioration des processus de développement en identifiant les zones à risque et en guidant l'optimisation du code. En somme, ce rapport vise à souligner l'importance des métriques orientées objet dans la création de logiciels robustes et maintenables.

Mots-Clés : Métriques de test orientées objet, programmation orientée objet, tests logiciels, qualité, bugs, erreurs

Table des matières :

1- Introduction :

Les métriques de test orientées objet sont des mesures utilisées pour évaluer la qualité et la performance des logiciels orientés objet. Elles sont spécifiquement conçues pour analyser les caractéristiques des programmes écrits en utilisant des principes de programmation orientée objet (POO). Ces métriques se concentrent souvent sur des aspects tels que l'encapsulation, l'héritage, la polymorphie, la cohésion et le couplage dans le code source. L'objectif principal des métriques de test orientées objet est d'identifier les zones potentielles de faiblesses ou d'améliorations dans le code, afin d'améliorer sa qualité, sa maintenabilité et sa robustesse.

2- Quelques métriques orientée objet :

2.1 Number Of Overridden Methods (NORM):

Elle permet de calculer le nombre de méthode surchargées, pour une classe donnée.

Méthode de calcul:

.c: une classe, SURc: l'ensemble des méthodes surchargées de c.

La valeur de la métrique NORM pour la classe c sera donnée par la cardinalité de l'ensemble SURc.

Ce calcul est défini par l'équation : $NORM(c) = |SUR_c|$

Impact :

-Un NORM élevé peut signaler une complexité accrue et une maintenance difficile, tandis qu'un NORM bas peut indiquer une conception plus simple mais potentiellement moins flexible.

2.2 Number Of Inherited Methods (NMI):

Cette métrique mesure le nombre de méthodes héritées par une sous-classe. Aucune mention n'est faite quant à savoir si cet héritage est public ou privé. Dans un langage tel que Java, nous devons considérer la possibilité

que l'héritage soit privé. Ainsi, toute classe utilisant des méthodes d'une sous-classe n'aurait pas nécessairement accès à toutes les méthodes héritées.

Méthode de calcul :

c : une classe, C: superclasse de c.

METc : l'ensemble des méthodes de c.

METC : l'ensemble des méthodes de C.

La valeur de la métrique NMI pour la classe c sera donnée par la cardinalité de l'intersection de l'ensemble METc et METC

Ce calcul est défini par l'équation : $NMI(c) = |MET_c \cap MET_C|$

Impact:

-Un NMI élevé peut indiquer une forte dépendance sur les méthodes héritées, ce qui peut rendre le code plus complexe et difficile à maintenir (encapsulation fragile dans le cas d'héritage privée).

-Un NMI élevé peut également indiquer une bonne réutilisabilité du code, car les méthodes héritées peuvent être utilisées dans plusieurs parties du programme.

2.3 Specialization Index (SIX). [Lorenz and Kidd, 1994]

L'indice de spécialisation mesure dans quelle mesure les sous-classes remplacent le comportement de leurs superclasses.

Méthode de calcul:

.c: une classe, H(c): la position de c dans la hiérarchie d'héritage.

METc: l'ensemble des méthodes de c.

NORM(c): nombre de méthode surchargé de c.

Il est défini par l'équation suivante $SIX(c) = \frac{|NORM(c)| \cdot H(c)}{|METc|}$

Impact :

-Le SIX peut indiquer qu'il y a un excès de méthodes surchargées, ce qui pourrait signifier que l'abstraction initiale n'était peut-être pas appropriée. (Qualité de la sous-classification).

-L&K suggèrent une valeur de 15% pour aider à identifier les classes pour lesquelles une action corrective pourrait être appropriée.

2.4 Method Inheritance Factor (MIF):

C'est une métrique de niveau système qui est le quotient de la somme des méthodes hérité dans chaque classe du système sur la somme des méthodes total (déclaré + hérité) de chaque classe du système.

Méthode de calcul : $MIF = \frac{\sum_{i=1}^{TotalClasses} M_{herited}(C_i)}{\sum_{i=1}^{TotalClasses} M_{available}(C_i)}$, tel que : $M_{available}(C) = M_{herited}(C) + M_{defined}(C)$

2.5 Public Methods Ratio (PMR) :

C'est une métrique de niveau classe qui retourne le quotient du nombre de méthodes publique sur le nombre de méthode globale de la classe.

Méthode de calcul: $PMR(c) = \frac{NumberofPubliqueMethods}{NumberofMethods}$

Impact :

Un PMR élevé peut rendre la classe plus complexe et difficile à maintenir en raison d'une exposition excessive de son interface publique. Cependant, cela peut également faciliter l'utilisation et la compréhension de la classe si les méthodes publiques sont bien documentées et encapsulées.

2.6 Inherited Methods Ratio (IMR) :

C'est une métrique de niveau classe qui retourne le quotient du nombre de méthodes hérité sur le nombre de méthode globale de la classe

Méthode de calcul: $IMR(c) = \frac{NumberofInhertedMethods}{TotalMethods}$

Impact :

Un IMR élevé peut indiquer une forte dépendance de la classe sur les méthodes héritées, ce qui peut compromettre l'indépendance et la modularité de la classe.

2.7 Method Hidden Factor (MHF) :

MHF est défini comme le rapport de la somme des invisibilités de toutes les méthodes définies dans toutes les classes au nombre total de méthodes définies dans le système en question.

L'invisibilité d'une méthode est le pourcentage des classes totales à partir desquelles cette méthode n'est pas visible.

En d'autres termes, MHF est le rapport des méthodes cachées - méthodes protégées ou privées - au total des méthode

Méthode de calcul: $\frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1-V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)}$

où : $M_d(C_i)$ est le nombre de méthodes déclarées dans une classe, et est le nombre de méthodes déclarées dans une classe,

Et : $V(M_{mi}) = \frac{\sum_{j=1}^{TC} isVisible(M_{mi},C_i)}{TC-1}$

Et : $isVisible\begin{pmatrix} M_{mi}, C_i = \{1ssi\ j \neq i\ C_j\ peut\ appelé\ M_{mi} \\ 0, sinon\ \} \end{pmatrix}$

TC = Nombre total de classes dans le système en question.

Dans des langages où il existe le concept de méthode protégée, la méthode est comptée comme une fraction entre 0 et 1 : $V(M_{mi}) = \frac{DC(C_i)}{TC-1}$

Impact :

Un MHF élevé peut indiquer une forte utilisation de méthodes cachées, ce qui peut renforcer l'encapsulation et la sécurité du code. Cependant, cela peut rendre le code plus difficile à comprendre et à maintenir, en particulier si les méthodes cachées ne sont pas bien documentées ou si elles sont utilisées de manière incohérente.

2.8 Coupling Between Objects (CBO) : [Chidamber and Kemerer]

Définition :

Le CBO mesure le nombre de classes auxquelles une classe particulière est couplée. En d'autres termes, il compte combien de classes différentes une classe donnée dépend ou interagit.

Impact :

CBO élevé : Indique que la classe est fortement interdépendante avec d'autres classes, ce qui peut rendre le système plus complexe et plus difficile à maintenir.

CBO faible : Suggère que la classe est relativement indépendante, ce qui la rend généralement plus facile à réutiliser et à maintenir.

2.9 Coupling Factor (CF) :

Définition :

Le Facteur de Couplage est une métrique à l'échelle du système qui mesure le degré de couplage entre les classes dans l'ensemble de l'application. Il offre une vue d'ensemble de l'interdépendance des classes dans tout le système.

Méthode de calcul : $\frac{\sum_{i=1}^{TC} \sum_{j=1}^{TC} is_{client}(C_i,C_j)}{TC^2-TC}$ **où** : $\{1, ssiC_c \Rightarrow C_s\ C_C \neq C_s\ 0, sinon\ \}$

La relation client-fournisseur (Cc => Cs) signifie que la classe cliente, Cc, contient au moins une référence non héritée à une fonctionnalité de la classe fournisseur, Cs.

TC = Nombre total de classes dans le système en question.

Où : n est le nombre total de classes dans le système,

CBO_i est la valeur de CBO pour la i -ème classe.

Impact :

CF élevé : Indique que le système présente un niveau élevé de dépendances entre les classes, ce qui peut compliquer la maintenance et réduire la modularité.

CF faible : Suggère que les classes du système sont relativement indépendantes, ce qui renforce la modularité et simplifie la maintenance et les tests.

2.10 Afferent – Efferent Coupling (CA-CE) :

Couplage afférent (CA) : Il représente le nombre d'entités externes qui dépendent de l'entité courante.

Couplage efférent (CE) : Il représente le nombre d'entités externes dont dépend l'entité courante.

Méthode de calcul : p: un package, : n : nombre de classe dans p.

$CA(p) = \sum_{i=1}^{n} CA_i$ $\quad\quad$ $CE(p) = \sum_{i=1}^{n} CE_i$

Où : CAi représente le nombre d'entités externes qui dépendent de la classe i du package.

CEi représente le nombre d'entités externes dont dépend la classe i du package.

Impact :

- Un CA élevé peut indiquer une forte dépendance des autres parties du système à l'égard du package étudié. Cela peut rendre le package plus sensible aux changements externes, augmentant ainsi sa vulnérabilité et sa complexité.

- Un CE élevé indique : Une complexité accrue due à une forte interdépendance entre les composants logiciels.

-Un CE faible indique : Un design plus simple et modulaire avec une moindre interdépendance.

2.11 Polymorphism Factor (PF) : [Chidamber and Kemerer]

PF est défini comme le rapport entre le nombre réel de situations polymorphiques différentes possibles pour la classe Ci et le nombre maximum de situations polymorphiques distinctes possibles pour la classe Ci.

PF est le nombre de méthodes qui redéfinissent des méthodes héritées, divisé par le nombre maximum de situations polymorphiques distinctes possibles.

Méthode de calcul: PF = $\frac{\sum_{i=1}^{TC} M_0(C_i)}{\sum_{i=1}^{TC} M_n(C_i) \times DC(C_i)}$

où : $M_d(C_i) = M_n(C_i) + M_0(C_i)$

Et : $M_n(C_i) = Nombre\ de\ nouvelles\ méthodes\ M_0(C_i) = Nombre\ de\ méthodes\ redéfinies$

DC $(C_i) = Nombrededescendants$

TC = Nombre total de classes dans le système en question.

Impact :

Un Polymorphism Factor (PF) élevé indique une forte utilisation du polymorphisme dans la conception, ce qui peut améliorer la flexibilité et l'extensibilité du système.

2.12 Cyclomatic Complexity (CC):

La complexité cyclomatique est une mesure de la complexité d'un programme informatique. Elle est calculée en comptant le nombre de chemins linéaires indépendants à travers le code source. Chaque chemin linéaire représente une série d'instructions qui peuvent être exécutées sans interruption du début à la fin de la fonction ou de la méthode.

Une décision est définie comme une occurrence de mots-clés tels que : « while », « for », « foreach »,"continue", "if", "case", "goto", "try" et "catch" dans la fonction. La complexité cyclomatique est la somme de ces constructions.

Méthode de calcul: $CC = Nombred'aretes - Nombredenoeudsdebase + 2$

où :

$Nombre\ de\ noeuds\ de\ base$: nombre total de points de décision dans le code.

$Nombre\ d'arêtes$: Nombre total de chemins possibles à travers le graphe de contrôle du programme.

Impact :

Une complexité cyclomatique élevée peut indiquer une fonction complexe avec de nombreuses branches de décision, ce qui peut rendre la fonction difficile à comprendre, à tester et à maintenir. Une complexité plus faible est souvent préférable, car elle indique un code plus simple et plus clair.

2.13 Weighted Methods Per Class (WMC) : [Chidamber and Kemerer]

La métrique Méthodes pondérées par classe (WMC) est la somme des complexités de méthodes définies

dans une classe. Il représente la complexité d'une classe dans son ensemble et peut être utilisé pour indiquer l'effort de développement et de maintenance pour une classe.

Méthode de calcul: $WMC(c) = \sum_{i=1}^{NM(c)} \quad C(m_i)$

où :

$NM(c)$: le nombre de méthodes dans une classe.

$C(m_i)$: la complexité de la méthode m_i.

Impact :

Une valeur élevée de WMC peut indiquer une classe complexe nécessitant une attention particulière en termes de conception et de maintenance. Cela peut signifier que la classe a un nombre élevé de méthodes ou que les méthodes individuelles sont complexes, ce qui peut rendre la classe plus difficile à comprendre, tester et maintenir. En revanche, une valeur plus basse peut indiquer une classe plus simple et plus facile à gérer.

2.14 Lack Of Cohesion Methods (LCOM) : [Chidamber and Kemerer]

La cohésion d'une classe est caractérisée par l'étroitesse des relations entre les méthodes locales et les attributs locaux. Le manque de cohésion (LCOM) mesure la dissemblance des méthodes dans une classe par variable d'instance ou attributs. Un module hautement cohérent doit être autonome, une cohésion élevée indique une bonne subdivision des classes.

Méthode de calcul: $si\ |P| > |Q|\ LCOM(c) = |P| - |Q|$, $sinon\ LCOM(c) = 0$

où : $P = \{(Ii, Ij) | Ii \cap Ij = 0\}$

P : Ensemble des paires de méthodes de la classe sans variable d'instance en commun, indiquant leur indépendance.

$$Q = \{(Ii, Ij) | Ii \cap Ij \neq 0\}$$

Q : Ensemble des paires de méthodes de la classe avec au moins une variable d'instance en commun, indiquant leur interdépendance.

$Si\ tous\ les\ ensembles\ \{I_1, \dots, I_n\} valent\ 0$, $alors\ P = 0$.

$(I_x)\ représente\ l'ensemble\ de\ toutes\ les\ variables d'instance\ utilisées\ par\ la\ méthode M_i$.

Impact :

- Une LCOM élevée indique une faible cohésion, ce qui rend la classe moins autonome et plus difficile à maintenir.

- Une LCOM élevée peut rendre la classe plus difficile à maintenir et augmenter le risque d'effets secondaires non intentionnels lors de modifications du code.

- Une LCOM basse rend le code plus cohérent et logique, facilitant ainsi sa compréhension et sa maintenance.

2.15 Stability (S):

Cette mesure indique la résilience du paquet au changement, avec une plage de valeurs entre 0 et 1. Un **S=0** signifie un paquet complètement stable, tandis qu'un **S=1** indique un paquet totalement instable.

Méthode de calcul : $S(p) = \frac{CE}{CE+CA}$

Impact :

- Un indice de stabilité (S) proche de 0 indique une forte stabilité du package, ce qui signifie qu'il est moins susceptible d'être affecté par les changements externes. Cela peut favoriser une architecture robuste et facile à maintenir.

- À l'inverse, un indice de stabilité (S) proche de 1 indique une forte instabilité du package, ce qui signifie qu'il est fortement influencé par les changements externes. Cela peut rendre la maintenance et l'évolution du code plus complexes et moins prévisibles.

2.16 Abstractness (A):

Le rapport entre le nombre de classes abstraites (et d'interfaces) dans le package analysé sur le nombre total de classes.

Cette mesure évalue le degré d'abstraction du package, avec une plage de valeurs entre 0 et 1. Un A=0 indique un package complètement concret, tandis qu'un A=1 indique un package complètement abstrait.

Méthode de calcul: $A(p) = \frac{NumberofAbstactClass+NumberofInterface}{NumberofClasses}$

Impact : Une valeur d'abstraction proche de 0 indique un package constitué principalement de classes concrètes. Cela oriente le code vers des implémentations spécifiques, ce qui peut rendre le package moins adaptable aux changements futurs. Cependant, cela peut également rendre le code plus accessible aux développeurs novices.

- Une valeur d'abstraction proche de 1 indique un package principalement composé de classes abstraites et d'interfaces. Cela favorise une conception plus abstraite, offrant une flexibilité accrue, une réutilisabilité et une maintenabilité améliorées. Cependant, cela peut rendre la compréhension et la navigation du code plus complexes pour les développeurs.

2.17 Normalized Distance From Main Sequence (DMS):

La distance de la séquence principale d'un package mesure à quel point il se situe par rapport à la "main sequence" théorique où la somme de l'abstraction et de la stabilité est égale à 1. Les valeurs proches de zéro indiquent un respect du SAP(Stable-Abstractions-Principle).

Méthode de calcul: $DMS = |A + S - I||$

Impact :

Cette métrique permet d'évaluer à quel point un package respecte le SAP, avec des valeurs proches de zéro indiquant un bon équilibre entre l'abstraction et la stabilité.

2.18 Number Of Children (NOC): [Chidamber and Kemerer]

Le nombre d'enfants d'une classe (NOC) est une métrique qui indique le nombre de sous-classes directes héritant d'une classe parente dans une hiérarchie d'héritage.

Impact :

- Un NOC élevé peut indiquer une classe centrale dans la hiérarchie d'héritage, ayant une influence significative sur la conception globale du système.

- Il peut nécessiter des tests supplémentaires pour valider les méthodes héritées dans chaque sous-classe.

- Un NOC élevé peut également indiquer une mauvaise abstraction de la classe parente, nécessitant une révision de la conception pour une meilleure distribution des responsabilités.

2.19 Number Of Links (NOL): Le nombre de liens d'une classe (NOL) est le nombre total de relations de dépendance (agrégation, composition, association, etc.) qu'elle a avec d'autres classes.

Impact:

- Un NOL élevé peut indiquer une classe qui est fortement liée à d'autres parties du système, ce qui peut augmenter la complexité et la dépendance entre les composants.
- Cela peut rendre la classe plus difficile à isoler et à réutiliser, nécessitant une gestion attentive de ses interactions avec d'autres classes.

2.20 Nested Block Depth (NBD):

La profondeur des blocs imbriqués (NBD) est le nombre maximal de niveaux d'imbrication de blocs de code à l'intérieur d'une fonction ou d'une méthode.

Impact :

-Une NBD élevée peut rendre le code source difficile à lire, à comprendre et à maintenir.

-Une profondeur excessive des blocs imbriqués peut indiquer une complexité excessive du code, ce qui peut augmenter le risque d'erreurs et de bogues.

-Une NBD élevée peut également rendre le code plus difficile à tester, car il peut y avoir plusieurs chemins d'exécution à travers les blocs imbriqués, augmentant ainsi la complexité des tests unitaires.

3- Les nouvelles métriques proposées :

3.1 Number of imports per class (NIC) (Métrique de code) :

La métrique NIC évalue le nombre total de classes importées dans une classe donnée. Elle fournit des indications sur la complexité et la charge de dépendances d'une classe.

Méthode de calcul: $NIC = \sum imports$

Impact:

Un grand nombre d'imports peut augmenter la complexité du code et prolonger le temps de compilation, affectant ainsi la maintenabilité et les performances du logiciel.

3.2 Number of effective imports per class (NEIC) (Métrique de code) :

La métrique NEIC évalue le nombre total de classes importées effectivement utilisées dans une classe donnée. Elle fournit des indications sur la pertinence et l'efficacité des dépendances d'une classe.

Méthode de calcul: $NEIC = \sum Used\ Imports$

Impact:

Une meilleure visibilité des imports nécessaires dans une classe améliore sa compréhension et sa maintenabilité, réduisant ainsi la complexité inutile du code.

3.3 Number of Used Imported Methods (NUIM) (Métrique de code) :

Définition :

C’est le nombre des méthodes utilisées depuis les importations.

Impact :

Un nombre plus grand de méthodes utilisées depuis les importations peut indiquer la dépendance de la classe sur d’autres classes externes ce qui la rend fragile et affecte par les changements dans les classes importes.

Un nombre plus grand de méthodes utilisées depuis les importations augmente le nombre total de methodes dans la classe ce qui donne une valeur plus élevée de la métrique ‘Number of methods’.

Si le nombre de méthodes utilisées est faible par rapport au nombre total de méthodes disponibles dans les importations, cela peut indiquer une sur-importation de bibliothèques, ce qui signifie que le projet importe des bibliothèques plus larges que nécessaire.

Un grand nombre de méthodes importées mais non utilisées peut ralentir la compilation et augmenter le temps d'exécution.

Lorsqu'il est nécessaire de mettre à jour ou de remplacer une bibliothèque, un nombre limité de méthodes utilisées facilite le processus d'adaptation aux nouvelles versions ou alternatives.

3.4 Number of Implemented Interfaces (NII) (Métrique de code):

Cette métrique mesure le nombre d'interfaces qu'une classe implémente. L'implémentation d'une interface nécessite la redéfinition de ses méthodes abstraites (en excluant les méthodes par défaut).

Impact :

Cela conduit à une augmentation du nombre de méthodes par classe, ce qui peut entraîner une complexité accrue du code et un nombre potentiellement plus élevé de lignes de code. Une classe avec un nombre élevé d'interfaces implémentées peut nécessiter une attention particulière lors du développement et de la maintenance pour garantir un comportement correct et une gestion appropriée des méthodes héritées.

3.5 Number Of Handled Exceptions (NHE) (Métrique de code) :

Définition :

C'est le nombre d'exceptions gérées dans une classe. On Java on compte le nombre de 'try' et 'throw'.

Impact : Un nombre élevé d'exceptions gérées indique une attention portée à la gestion des erreurs et des cas exceptionnels.

Une gestion adéquate des exceptions peut contribuer à prévenir les attaques par déni de service ou les tentatives d'exploitation de vulnérabilités par le biais d'erreurs inattendues.

3.6 Total Number Of Possible Exceptions (NPE) (Métrique de code) :

Définition :

C'est le nombre total des exceptions qui peuvent être générées dans un programme. Une mesure directe ne peut pas être fournie dans tous les cas (par exemple les erreurs en temps d'exécution) Mais cette métrique estime le taux des exceptions possibles dans le code, et indique les parties du code critiques qui peuvent la générer.

Impact :

Un grand nombre d'exceptions potentielles identifie les points critiques où des erreurs peuvent survenir, ce qui permet aux développeurs de concentrer leurs efforts sur la gestion appropriée de ces cas.

Un grand nombre d'exceptions potentielles peut indiquer une complexité excessive dans le code, ce qui peut rendre le code plus difficile à comprendre, à maintenir et à déboguer.

3.7 Exceptions Factor (EF) :

Le FE représente le pourcentage d'exceptions possibles gérées par le développeur dans son code.

Méthode de Calcul : $\frac{NHE}{NPE}$

Impact : Un EF élevé indique une prise en charge étendue des erreurs dans le code, ce qui peut contribuer à améliorer la robustesse et la fiabilité du logiciel.

Nous proposons cette métrique comme solution pour évaluer la qualité de la gestion des erreurs dans le code. Un faible facteur d'exceptions peut indiquer une gestion insuffisante des erreurs, ce qui peut conduire à des comportements imprévus ou à des pannes dans le logiciel.

Cette métrique peut être implémentée sous forme de plugin dans les environnements de développement intégrés (IDE).

3.8 Class Modification Frequency (CMF) (Métrique de processus) :

La fréquence de modification d'une classe mesure à quelle fréquence elle subit des changements pendant le développement logiciel, reflétant ainsi sa stabilité dans le temps.

Impact :

Une fréquence élevée de modifications peut signaler des problèmes de conception ou des exigences changeantes, compliquant la maintenance et la compréhension du code. Une classe stable, avec peu de modifications, est souvent considérée comme bien conçue et facile à maintenir.

Si une classe est modifiée fréquemment, il peut être un signe que cette classe a trop de responsabilités ou qu'elle dépend de trop d'autres classes, ce qui va à l'encontre des bonnes pratiques de conception.

Une fréquence élevée de modifications d'une classe peut indiquer des violations des principes SOLID, tels que SRP (trop de responsabilités), OCP (difficulté d'extension sans modification) et DIP (couplages forts), compromettant la stabilité et la maintenabilité du code.

3.9 Number of Attribute Uses Relative to Number of Methods (Métrique de code) :

Définition : Cette métrique évalue la fréquence à laquelle les attributs d'une classe sont utilisés par rapport au nombre total de méthodes définies dans cette classe. Elle offre un aperçu de l'interaction entre les attributs et les méthodes, ce qui peut influencer la complexité et la qualité du code.

Méthode de Calcul : $\frac{\sum \textit{Attributs utilisés}}{\sum \textit{Méthodes}}$

Impact : Un quotient élevé peut indiquer une forte dépendance des méthodes par rapport aux attributs, ce qui peut rendre la classe plus difficile à comprendre et à maintenir. D'autre part, un ratio bas peut indiquer une sous-utilisation des attributs ou une mauvaise encapsulation, ce qui peut affecter la flexibilité et la réutilisabilité du code. Une évaluation équilibrée de cette métrique peut contribuer à une meilleure conception et à une meilleure qualité du logiciel.

Nous avons suggéré cette métrique comme solution pour identifier les classes avec des responsabilités excessives ou des dépendances excessives entre les attributs et les méthodes. Cela permet de détecter des problèmes tels que la fragilité du code, la difficulté de maintenance et de compréhension, ainsi que des obstacles à la modification et à l'extension du logiciel.

3.10 Number of instanciable variables (Métrique de code) :

Définition : C'est le nombre de variables instanciables dans une classe, c'est-à-dire le nombre de variables de type 'Object' (Non primitive).

Impact : Lorsqu'on utilise des objets polymorphes, il peut être difficile de gérer correctement l'état de l'objet, surtout si les classes dérivées ont des états supplémentaires non présents dans la classe de base.

Un nombre de variables instanciables plus grand, implique des dépendances entre classes, ce qui augmente la valeur de la métrique 'Coupling Factor'.

Dans des hiérarchies de classes profondes, il peut devenir difficile de suivre et de comprendre tous les variables instanciables disponibles dans une classe donnée, surtout s'ils sont répartis sur plusieurs niveaux de la hiérarchie.

Plus une classe a de variables, plus il est probable que certains d'entre eux puissent être nul à un moment donné, nécessite une gestion rigoureuse de l'initialisation et de la vérification des attributs pour éviter les 'NullPointerException'.

3.11 Number of external imports (NXI) (Métrique de code):

Définition :

La métrique "Nombre d'Importations externes" quantifie le nombre de dépendances de code externe qu'un projet importe à partir de sources en ligne telles que GitHub, des bibliothèques externes, des frameworks ou des API.

Ces importations peuvent inclure des bibliothèques open-source, des modules tiers et d'autres ressources de code hébergées et intégrées au projet.

Impact : Les bibliothèques externes peuvent introduire des vulnérabilités de sécurité. En surveillant le nombre d'importations, les développeurs peuvent évaluer les risques associés et s'assurer que les dépendances sont régulièrement mises à jour et vérifiées. Importer du code depuis des sources non fiables peut introduire des erreurs ou des comportements inattendus.

Nous suggérons cette métrique comme solution pour évaluer le degré de dépendance du code à des ressources externes.

Cette métrique, inspirée par un problème rencontré lors de l'implémentation de nos métriques, découle de l'utilisation d'un parseur Java provenant d'un dépôt GitHub en tant que dépendance. Nous avons découvert que cette bibliothèque présentait une vulnérabilité notée à 7.1 sur 10, ce qui est considéré comme élevé, reporté sous le code CVE-2023-2976 .

3.12 Variable Naming Coherence (VNC) (Métrique de processus) :

La métrique VNC évalue la cohérence des noms de variables dans un programme. Elle vise à garantir une désignation uniforme et compréhensible des variables, ce qui contribue à la lisibilité et à la maintenance du code.

Méthode de calcul: $VNC = \prod NameCoherenceFactor(name)$, $0 \leq NameCoherenceFactor(x) \leq 1$

Impact:

Cette métrique met en évidence la cohérence dans le choix des noms des variables à travers un projet de logiciel. Nous proposons cette métrique comme solution pour évaluer la qualité du nommage des variables, car un manque de cohérence peut rendre le code plus difficile à comprendre et à maintenir. Des noms de variables incohérents peuvent entraîner des confusions et des erreurs lors de la lecture du code, ce qui nuit à la productivité des développeurs et augmente les risques d'erreurs

Nous avons inspiré cette métrique lors de l'implémentation de nos métriques, car des déclarations de variables incohérentes ont ralenti notre travail.

4- Contributions au-delà du projet :

- Conception d'un site web servant de vitrine pour présenter le travail réalisé, offrant plusieurs fonctionnalités telles que la consultation et les tests.

- Conception d'une interface graphique Java permettant de calculer les scores obtenus pour différentes métriques proposées pour un fichier Java.

- Organisation des implémentations dans un logiciel basé sur le patron MVC, permettant d'ajouter dynamiquement les implémentations des nouvelles métriques et de les tester.

Captures d'écrans :

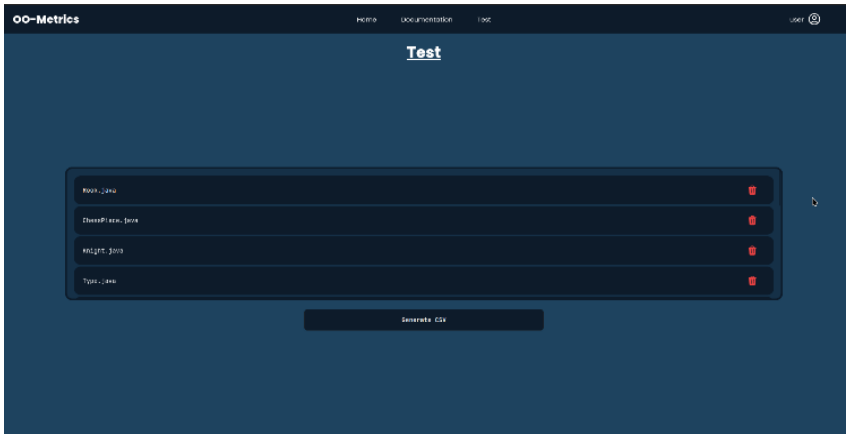

Figure 1: Page de choix et de calcul

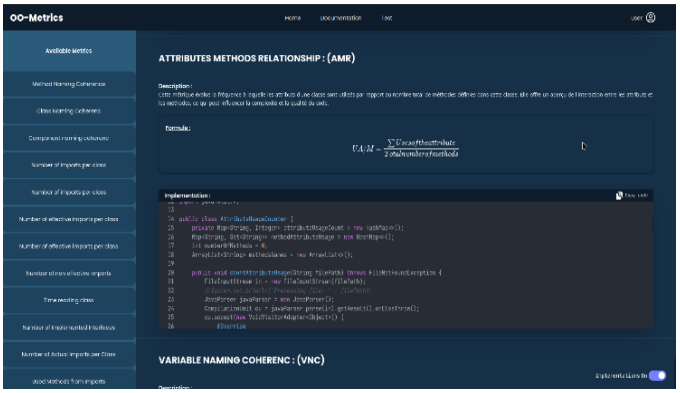

Figure 2: Page de documentation

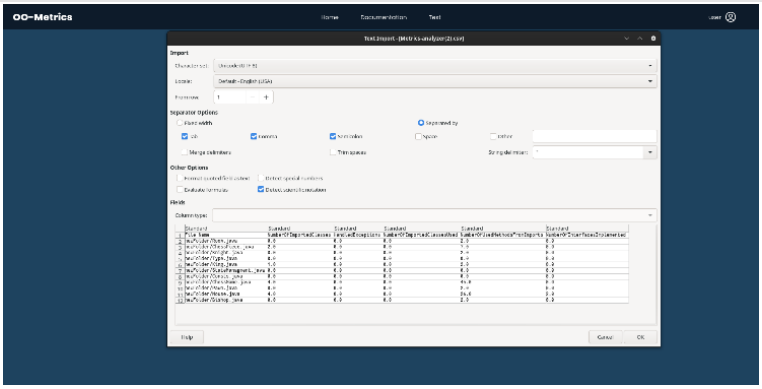

Figure 3: Page des résultats

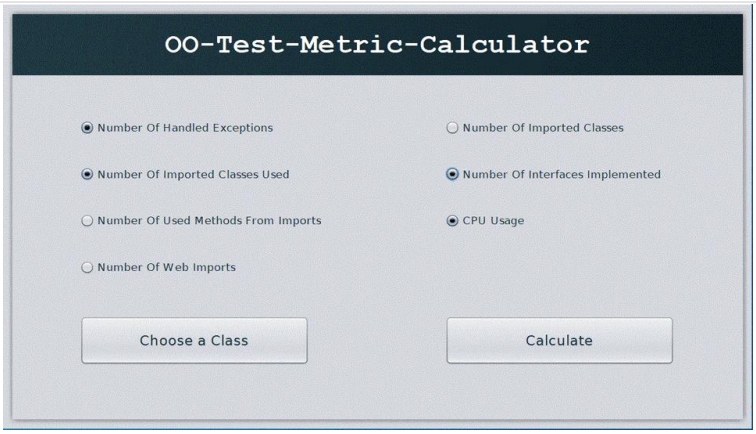

Figure 4 : sélection dans la calculatrice

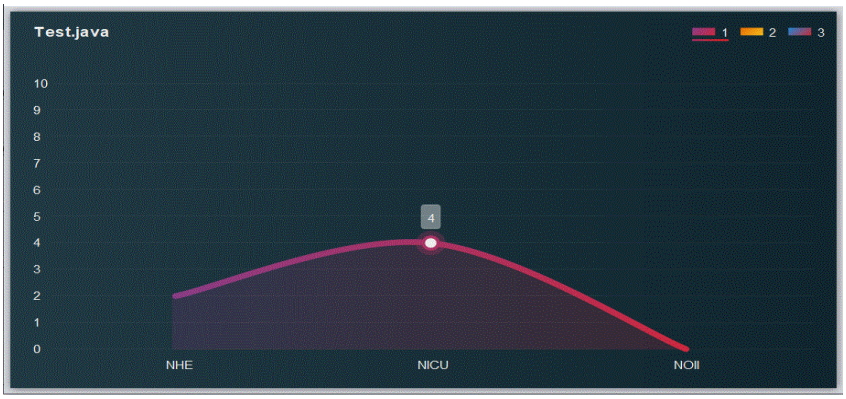

Figure 5: Résultats des calculs

5- Conclusions et travaux futurs :

Ce rapport présente une série de métriques orientées objet, offrant ainsi une rétroaction précieuse aux concepteurs de logiciels et aux gestionnaires. L'analyse et l'utilisation de ces métriques peuvent prédire la qualité de la conception, conduisant potentiellement à une réduction significative des coûts de mise en œuvre et à des améliorations de la qualité du produit final. En mettant l'accent sur des indicateurs de qualité précoces basés sur des données empiriques objectives, nous visons à favoriser une meilleure qualité logicielle et à réduire les efforts de maintenance future. À l'avenir, nous envisageons d'implémenter la métrique 'Number Of Possible Exceptions' - en construisant des modèles d'apprentissage automatique et les entraîner, par exemple, avec des arbres de décision, des forêts aléatoires, des réseaux neuronaux- , nous visons aussi à développer un nouvel ensemble de métriques, renforçant ainsi notre engagement envers l'excellence et l'amélioration continue dans le domaine du développement logiciel.

6- Références :

1- An Overview of Various Object Oriented Metrics, By Brij Mohan Goel & Prof. Pradeep Kumar Bhatia, International Journal of Information Technology & Systems, Vol. 2; No. 1: ISSN: 2277-9825.

2- Analysis of Object Oriented Metrics on a Java Application, By D.I. George Amalarethinam & P.H. Maitheen Shahul Hameed, International Journal of Computer Applications (0975 – 8887), Volume 123 – No.1, August 2015.

3- Applying and Interpreting Object Oriented Metrics, By Dr. Linda H. Rosenberg :Track 7 – Measures/Metrics.

4- Empirical Study of Object-Oriented Metrics, By K.K.Aggarwal & Yogesh Singh & Arvinder Kaur & Ruchika Malhotra, School of Information Technology, GGS Indraprastha University, Delhi 110006, India.

5- Metrics For Object Oriented Design (MOOD) To Asses Java Programs ,Prof. JUBAIR J. AL-JA'AFER & KHAIR EDDIN M. SABRI, University of Jordan.

6- An Overview of, Object-Oriented Design Metrics, Daniel Rodriguez, Rachel Harrison, RUCS/2001/TR/A, March 2001.