**Les Métriques Orientées Objet**

**KARA Nabil, HADJ-ARAB Adel, BENALIA Mohamed**

**RACHEDI Abderrahmane, KECIRA Abderraouf , BECHAR Walid**

Project pluridisciplinaire 10, Département d’informatique, USTHB, Algérie

**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 .**
2. **Analyse des métriques orientées objet et leur impact sur la qualité logicielle.**
3. **Les nouvelles métriques proposées .**
4. **Contributions au delà du projet.**
5. **Conclusions et travaux futurs .**
6. **Références .**

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.

1. **Analyse des métriques orientées objet et leur impact sur la qualité logicielle:**

Les métriques présentées dans cet article sont orientées objet. Le but de notre travail n'est pas de mentionner toutes les métriques existantes ni de les présenter entièrement, mais plutôt de sensibiliser le lecteur à leur existence et d'offrir des références pour des lectures complémentaires. Nous visons de présenter une large revue de la littérature existante sur les mesures OO. Plusieurs auteurs ont formulé des suggestions qui doivent être prises en compte lors de la définition des métriques. Elles doivent être définies en poursuivant un objectif clair et leur calcul doit être facile ; il est préférable que leur extraction soit automatisée par un outil. Dans ce travail nous fournissons un aperçu de l'état actuel des métriques orientée objet, en mettant l'accent sur les forces et les faiblesses de chaque proposition existante. Ainsi, les chercheurs peuvent avoir une vue d'ensemble approfondie du travail déjà accompli et de celui qui reste à réaliser dans le domaine des métriques orientée objet.

**2. Quelques métriques orientée objet :**

Nous allons maintenant présenter les propositions de métriques qui illustrent au mieux le contexte actuel des métriques orientée objet.

**2.1 Number Of Overridden Methods (NORM):**

Elle permet de calculer le nombre de méthodes

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 suivante :

**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 cette 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 suivante :

**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 Number Of Methods Added To Inheritance (NMA):**

Une méthode est définie comme une méthode ajoutée dans une sous-classe s'il n'existe aucune méthode du même nom dans aucune de ses superclasses.

**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 NMA pour la classe c sera donnée par la cardinalité de la différence entre METc et l’intersection de l’ensemble METc et METC

Ce calcul est défini par l’équation suivante :

**Impact :**

Un NMI élevé peut contribuer à une meilleure spécialisation et à une extension des fonctionnalités, mais il peut également augmenter la complexité et les risques associés à la modification et à la maintenance du code.

**2.4 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 l’hiérarchie

d’heritage.

.METc: l’ensemble des méthodes de c.

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

Il est défini par l’équation suivante :

**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.

* 1. **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:**

tel que :

* 1. **Number of Public Methods Defined (PMd):**

C'est une métrique de niveau classe qui retourne le nombre de toutes les méthodes publiques déclaré par une classe donnée.

**Impact:**

Un PMd élevé peut indiquer une interface plus étendue exposée par la classe, ce qui pourrait rendre la classe plus complexe et difficile à maintenir. Cependant, cela pourrait également refléter une conception plus riche et une meilleure encapsulation si les méthodes sont bien organisées et cohérentes.

* 1. **Number of Public Methods inherited (NPMi) :**

C'est une métrique de niveau héritage qui retourne le nombre de toutes les méthodes publiques hérite par la classe parent d'une classe donnée.

**Impact :**

Un NPMi élevé peut indiquer une forte dépendance de la classe sur les fonctionnalités héritées, ce qui peut affecter la flexibilité et la maintenance du code.

* 1. **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:**

**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.

**2.9 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:**

**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.

* 1. **Method Hidden Factor (MHF) :**

C'est une métrique de niveau système qui est le quotient du nombre des méthodes cachées dans chaque classe du système sur le nombre des méthodes déclarées.Une méthode cachée est une méthode privée.

**2.11 Number of Hidden Methods defined(NHMd):**

C'est une métrique de niveau classe qui retourne le nombre de toutes les méthodes cachées déclaré par une classe donnée.

**Impact :**

Un NHMd élevé peut indiquer une forte utilisation de méthodes privées dans la classe, favorisant ainsi une encapsulation efficace. Cependant, cela peut rendre le code plus difficile à tester et à maintenir.

**2.12 Number of Hidden Methods inherited (NHMi):**

C'est une métrique de niveau héritage qui retourne le nombre de toutes les méthodes cachées hérité par la classe parent d'une classe donnée.

**Impact :**

Un NHMi élevé peut indiquer une dépendance significative de la classe sur les méthodes cachées héritées, renforçant ainsi l'encapsulation et la sécurité du code. Cependant, cela peut également complexifier la compréhension du code, surtout si ces méthodes ne sont pas bien documentées ou utilisées de manière inappropriée.

**2.14 Coupling factor :**

CF est défini comme le rapport entre le nombre maximal possible de couplages dans le système et le nombre réel de couplages non imputables à l'héritage. Autrement dit, cette métrique compte le nombre de communications entre classes.

**Méthode de calcul:**

**où** :

}

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.

**Impact :**

Un CF élevé peut indiquer une forte interdépendance entre les classes du système, ce qui peut augmenter la complexité et la difficulté de maintenance. D'autre part, un CF bas peut indiquer une meilleure modularité et une plus grande facilité de maintenance, car les classes sont moins fortement couplées entre elles.

**2.14 Polymorphism Factor (PF):**

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 =

**où** :

Et :

DC (

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.

**2.15 Attribute Inheritance Factor (AIF) :**

AIF est défini comme le rapport de la somme des attributs hérités dans toutes les classes du système en question au nombre total d'attributs disponibles (définis localement plus hérités) pour toutes les classes.

**Méthode de calcul:**

**où :**

**Et :** = le nombre d'attributs déclarés dans une classe.

= le nombre d'attributs pouvant être invoqués en association avec

= le nombre d'attributs hérités (et non redéfinis dans ).

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

**Impact :**

Un Attribute Inheritance Factor (AIF) élevé peut indiquer une forte dépendance des classes sur les attributs hérités, ce qui peut simplifier la réutilisation du code et favoriser une conception plus cohérente. Cependant, cela peut également rendre le système plus complexe à comprendre et à maintenir, en particulier si les attributs hérités sont mal documentés ou mal utilisés.

**2.16 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:**

**où**

est le nombre de méthodes déclarées dans une classe, et est le nombre de méthodes déclarées dans une classe,

**et :**

**et :**

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 :

**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.

* 1. **Attribute Hiding Factor (AHF):**

AIF est défini comme le rapport de la somme des attributs hérités dans toutes les classes du système en question au nombre total d'attributs disponibles (définis localement plus hérités) pour toutes les classes.

**Méthode de calcul:**

**où**

**et :**  : le nombre d'attributs déclarés dans une classe.

 : le nombre d'attributs pouvant être invoqués en association avec Ci.

 : le nombre d'attributs hérités (et non redéfinis dans Ci).

TC : le nombre total de classes dans le système en question.

**Impact :**

Un AHF élevé peut indiquer une forte utilisation d'attributs hérités dans le système, ce qui peut simplifier la conception et favoriser la réutilisabilité du code. Cependant, cela peut également rendre le système plus dépendant des classes parentes et plus difficile à maintenir si les attributs hérités ne sont pas correctement documentés ou utilisés de manière incohérente.

**2.18 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:**

**où**

nombre total de points de décision dans le code.

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.

**2.19 Weighted Methods Per Class (WMC):**

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:**

**où**

: le nombre de méthodes dans une classe.

 : la complexité de la méthode mi.

**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.

**­­­2.20** **Lack Of Cohesion Methods (LCOM):**

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:**

**où**

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

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

**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.

**2.21 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:**

.: un package, :  : nombre de classe dans .

**où**

représente le nombre d'entités externes qui dépendent de la classe du package.

représente le nombre d'entités externes dont dépend la classe 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 peut révéler une forte dépendance du package vis-à-vis d'autres parties du système. Cela peut rendre le package plus difficile à réutiliser et à maintenir, car il peut être fortement lié à d'autres composants, augmentant ainsi la complexité du système dans son ensemble.

**2.22 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:**

**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.

**2.23 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:**

**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.

**2.24 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:** |

**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.

**2.25 Number Of Children (NOC):**

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.

**2.26 Number Of Parents (NOPa):**

Le nombre de parents d'une classe (NOPa) est le nombre de classes dont elle hérite directement.

**Impact:**

- Un NOPa élevé peut indiquer une classe qui hérite de multiples fonctionnalités, ce qui peut entraîner une complexité accrue et une plus grande dépendance par rapport à d'autres parties du système.

- Cela peut nécessiter une analyse plus approfondie de la hiérarchie d'héritage pour s'assurer que la classe reste cohérente et bien définie dans son rôle.

**2.27 Number Of Ancestors (NOA):**

Le nombre d'ancêtres d'une classe (NOA) est le nombre de classes à partir desquelles elle hérite directement ou indirectement.

**Méthode de calcul:**

**Impact :**

- Un NOA élevé peut indiquer une classe qui hérite de nombreuses fonctionnalités et qui peut être influencée par divers aspects de la conception du système.

- Cela peut rendre la classe plus complexe à comprendre et à maintenir, nécessitant une gestion plus minutieuse de ses interactions avec ses ancêtres.

**2.28 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.

* 1. **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.

1. **Les nouvel****les métriques proposées :**

Dans le domaine du développement logiciel, l'évaluation de la qualité et de la performance des applications orientées objet revêt une importance capitale. Notre équipe s'est penchée sur cette problématique en développant une série de métriques orientées objet visant à mieux appréhender la complexité, la maintenabilité et la robustesse des systèmes logiciels. Ces métriques, bien que modestes dans leur ambition, représentent une avancée significative dans notre quête de méthodes d'évaluation plus précises et adaptées à l'environnement orientés objet.

* 1. **Number of imports per class (NIC):**

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:**

**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):**

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:**

**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 Implemented Interfaces (NII) :**

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.

**3.4 Number Of Handled Exceptions (NHE) :**

**Définition :**

C’est le nombre d’exceptions gerees dans une classes. On Java om 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.

**3.5** **Total Number Of Possible Exceptions :**

**Définition :**

C’est le nombre total des exceptions qui peuvent être générées dans un programme. Une mesure directe ne peut pas être fournie dans tout 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.

**3.6 Facteur d'Exceptions (FE) :**

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

**Méthode de Calcul :**

**Impact :** Un FE 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.

**3.7 Class Modification Frequency (CMF) :**

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.

**3.8 Number of Attribute Uses Relative to Number of Methods:**

**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 :**

**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.

**3.9 Number of instanciable attributes :**

**Définition :** C’est le nombre d’attributs instanciables dans une classe, c’est-à-dire le nombre d’attributs de type ‘Object’ (Non primitive).

**Impact :**

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

Un nombre d’attributs 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 attributs instanciables disponibles dans une classe donnée, surtout s'ils sont répartis sur plusieurs niveaux de la hiérarchie.

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

**3.10 Number of imports from the web (NIW) :**

**Définition :**

La métrique "Nombre d'Importations depuis le Web" 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 .

**Impacts:**

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.

**3.11** **Number of Used Imported Methods (NUIM) :**

**Définition :**

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

**Impact :**

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.

1. **C****ontributions 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.

1. **Conc****lusions 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.

1. ***Référen******ces:***
2. 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.
3. 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.
4. Applying and Interpreting Object Oriented Metrics, By Dr. Linda H. Rosenberg :Track 7 – Measures/Metrics*.*
5. 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.
6. Metrics For Object Oriented Design (MOOD) To Asses Java Programs ,Prof. JUBAIR J. AL-JA'AFER & KHAIR EDDIN M. SABRI, University of Jordan.
7. An Overview of, Object-Oriented Design Metrics, Daniel Rodriguez, Rachel Harrison, RUCS/2001/TR/A, March 2001.