Une vue d'ensemble sur diverses Mesures de Test Orientées Objet.

*** Kara Nabiul, ** Hadj-Arab Adel, *** Benalia Mohamed**
*** Rachedi Abderahman, ** Kecira Abdraouf, *** Bechar Walid**

*Project pulirdisiplinaire 10, Deptt 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-Cles: Métriques de test orientées objet, programmation orientée objet, tests logiciels.

1.Introduction aux Métriques Orientées Objet:

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

Les métriques présentées dans cet article sont des métriques de test orientées objet. Le but de cet article 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. L'objectif de ce travail est de présenter une large revue de la littérature existante sur les mesures de test OO. Les métriques examinées sont proposées pour les systèmes OO afin de se concentrer sur les métriques de produit pouvant être appliquées à une conception avancée ou au code. Plusieurs auteurs ont formulé des suggestions qui doivent être prises en compte lors de la définition des métriques pour les logiciels. Les métriques 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. L'objectif de ce travail est de fournir aux chercheurs un aperçu de l'état actuel des métriques pour la programmation 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 de test pour la programmation orientée objet.

2. Metrics de Test Object-Oriented:

Nous allons maintenant présenter les propositions de métriques qui illustrent au mieux le contexte actuel des métriques de test pour la programmation 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 :

$$NORM(c) = |SUR_c|$$

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 cette héritage est public ou privé. Dans un langage tel que C++, 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 :

$$NMI(c) = |MET_c \cap MET_C|$$

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 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 :

$$NMA(c) = METc - |MET_c \cap MET_C|$$

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 surcharge 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.

2.5 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 : $$M_{available}(C) = M_{herited}(C) + M_{defined}(C)$$

2.6 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.

2.7 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.

2.8 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:

$$PMR(c) = \frac{Number\ of\ Publique\ Methods}{Number\ of\ Methods}$$

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:

$$IMR(c) = \frac{Number\ of\ Inherted\ Methods}{Total\ Methods}$$

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.

2.10 Method Hidden Factor (MHF) :

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

Méthode de calcul:

$$\frac{\sum_{i=1}^{TotalClasses} M_{hidden}(C_i)}{\sum_{i=1}^{TotalClasses} M_{defined}(C_i)}$$

tel que : $M_{defined}(C) = M_{hidden}(C) + M_{visible}(C)$

Impact :
Un MHF élevé peut indiquer une forte utilisation de méthodes privées dans le système, ce qui peut améliorer l'encapsulation et la sécurité du code. Cependant, cela peut également rendre le code plus difficile à comprendre et à maintenir, car les méthodes privées ne sont pas accessibles à l'extérieur de leur classe respective.

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 cachée hérite 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.

2.13 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: $\frac{\sum_{i=1}^{TC}\sum_{j=1}^{TC} is_{client}(C_i, C_j)}{TC^2 - TC}$

où : $\begin{cases} 1. ssi\, C_c \Rightarrow C_s\, C_C \neq C_s \\ 0, sinon \end{cases}$

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:

$$\text{PF} = \frac{\sum_{i=1}^{TC} M_0(C_i)}{\sum_{i=1}^{TC} M_n(C_i) \times DC(C_i)}$$

où : $M_d(C_i) = M_n(C_i) + M_0(C_i)$
Et : $M_n(C_i) = Nombre\, de\, nouvelles\, methodes$
$M_0(C_i) = Nombre\, de\, methodes\, redefinies$
DC (C_i = *Nombre de descendants*
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: $\frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_0(C_i)}$

où : $A_n(C_i) = A_d(C_i) + A_i(C_i)$
Et : $A_d(C_i)$= le nombre d'attributs déclarés dans une classe
$A_a(C_i)$ = le nombre d'attributs pouvant être invoqués en association avec C_i
$A_i(C_i)$ = le nombre d'attributs hérités (et non redéfinis dans C_i .
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: $\frac{\sum_{i=1}^{TC}\sum_{m=1}^{M_d(C_i)} \left(1 - V(M_{mi})\right)}{\sum_{i=1}^{TC} M_d(C_i)}$

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 : $isVisible(M_{mi}, C_i = \begin{cases} 1\, ssi\, j \neq i\, C_j\, peut\, appel\acute{e}\, M_{mi}) \\ 0, sinon \end{cases}$

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.

2.17 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: $\dfrac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$

où : $A_a(C_i) = A_d(C_i) + A_i(C_i)$

et : $A_d(C_i)$: le nombre d'attributs déclarés dans une classe.

$A_a(C_i)$: le nombre d'attributs pouvant être invoqués en association avec Ci.

$A_i(C_i)$: 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:

$CC = Nombre\ d'aretes - Nombre\ de\ noeuds\ de\ base + 2$

où :

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

Nombre d'aretes :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 est une somme de 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: $WMC(c) = \sum_{i=1}^{NM(c)} C(m_i)$

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. Ainsi, WMC peut être utilisé pour identifier les classes qui nécessitent une attention particulière lors du processus de développement et de maintenance du logiciel.

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:

$si\ |P| > |Q|\ \ LCOM(c)=|P|-|Q|$

$sinon\ \ LCOM(c)=0$

où :

$P=\{(Ii, Ij) \mid Ii \cap Ij = 0\}$

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

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

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

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

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

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.

2.21 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:

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

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

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

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 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:

$$S(p)=\frac{CE}{CE+CA}$$

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.

2.23 Abstractness (A):

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

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:

$$A(p)=\frac{Number\ of\ Abstact\ Class+Number\ of\ Interface}{Number\ of\ Class}$$

Impact :
- Une valeur d''abstractité 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''abstractité 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'abstractité et de la stabilité est égale à 1. Les valeurs proches de zéro indiquent un respect du SAP(Stable-Abstractions-Principle).

Méthode de calcul: $DMS=|A+S-1||$

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 abstractité et stabilité.

2.25 Number of attributes:

Cette métrique compte simplement le nombre total d'attributs (également appelés variables d'instance ou champs) au sein d 'une classe. Les attributs représentent l'état d'un objet et définissent ses caractéristiques ou propriétés.

Impact :
- Un grand nombre d'attributs dans une classe peut rendre celle-ci plus complexe et difficile à comprendre, ce qui peut conduire à une conception moins maintenable et à des difficultés lors de la modification du code.
- D'autre part, un nombre minimal d'attributs peut indiquer une classe plus cohérente et mieux encapsulée, ce qui facilite sa compréhension et sa modification ultérieure.

2.26 Number of static attributes:

Cette métrique compte le nombre total d'attributs statiques définis dans une classe. Les attributs statiques appartiennent à la classe elle-même et sont partagés entre toutes les instances de la classe.

Impact :
Un nombre élevé d'attributs statiques peut indiquer une forte dépendance de la classe sur des données partagées, ce qui peut entraîner des problèmes de concurrence et de cohérence dans le programme. D'autre part, un nombre minimal d'attributs statiques peut rendre la classe plus autonome et faciliter la gestion de son état interne.

2.27 Number of public attributes:

Cette métrique compte le nombre total d'attributs publics définis dans une classe. Les attributs publics peuvent être accessibles depuis l'extérieur de la classe et sont déclarés avec le modificateur d'accès public.

Impact :
Un nombre élevé d'attributs publics peut compromettre l'encapsulation des données, rendant la classe plus vulnérable aux modifications externes et aux erreurs de manipulation des données. Cela peut également rendre le code plus difficile à maintenir, car les dépendances externes sur les attributs peuvent se propager à travers le système.

2.28 Number of methods:

Cette métrique compte le nombre total de méthodes (y compris les méthodes d'instance et les méthodes statiques) définies dans une classe. Les méthodes définissent le comportement de la classe et peuvent manipuler l'état de l'objet ainsi que réaliser diverses tâches.

Impact :
Un grand nombre de méthodes peut rendre une classe plus complexe et difficile à comprendre, ce qui peut entraîner une baisse de la maintenabilité du code. Cependant, un nombre adéquat de méthodes, bien organisées et cohérentes, peut contribuer à une conception logicielle plus modulaire et plus facile à gérer.

2.29 Number of static methods:

Cette métrique compte le nombre total de méthodes statiques définies dans une classe. Les méthodes statiques appartiennent à la classe elle-même et peuvent être appelées directement sur la classe sans nécessiter la création d'une instance.

Impact :
Un grand nombre de méthodes statiques peut indiquer une forte dépendance de la classe sur des opérations globales, ce qui peut réduire la modularité du code et rendre la classe plus difficile à tester et à maintenir. En revanche, un nombre minimal de méthodes statiques peut favoriser une meilleure encapsulation des fonctionnalités et une conception plus orientée objet.

2.30 Number of classes

Cette métrique compte le nombre total de classes dans un système logiciel ou dans un module spécifique. Les classes servent de modèles pour créer des objets, encapsuler des données et définir des méthodes pour opérer sur ces données.

Impact :
Le nombre total de classes dans un système logiciel peut avoir un impact significatif sur sa complexité et sa maintenabilité. Un grand nombre de classes peut rendre le système plus complexe à comprendre et à gérer, tandis qu'un nombre minimal de classes peut simplifier la structure globale du système. Il est donc important de maintenir un équilibre approprié dans le nombre de classes pour assurer une conception logicielle cohérente et facile à maintenir.

2.31 Number Of Children (NOC):

Le nombre d'enfants d'une classe (NOC) est une mesure 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.

2.32 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.

2.33 Number Of Descendants (NOD):

Le nombre de descendants d'une classe (NOD) est le nombre de classes qui héritent directement ou indirectement de cette classe.

Méthode de calcul:
$$NOD = N_{direct\,subclasses} + N_{indirect\ subclasses}.$$

Impact :
- Un NOD élevé peut indiquer une classe qui a une influence étendue sur différentes parties du système, en raison de ses nombreux descendants.
- Cela peut nécessiter une attention particulière lors de la modification de la classe parente pour éviter des effets indésirables sur ses descendants.

2.34 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:
$$NOA = N_{direct\ superclasses} + N_{indirect\ superclasses}.$$

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.

2.35 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.

2.36 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. Nouvelle Métrique :

Cette Section est consacré aux nouvelles métriques découverte par notre équipe, une implementation de tous ses métrique est prévue et ont est déterminer à la réaliser.

3.1 Variable Naming Coherence (VNC) :

Calcule la coherence de denomination de tout les variable dans une program.

Méthode de calcul:

VNC = product{NameCoherenceFactor(name)}
telleque :
0<= NameCoherenceFactor(x) <= 1

impact :
-Ecrire des programmes robustes est facilement maintenable.

3.2 Methode Naming Coherence (MNC):

Calcule la coherence de denomination de tout les methode dans une program.

Méthode de calcul:

MNC=product{MethodCoherenceFactor(methode)}
tellque :
0<= MethodCoherenceFactor(x) <= 1

impact :
-Ecrire des programmes robustes est facilement maintenable.

3.3 Class Naming Coherence (CNC) :

calcule la coherence de denomination de tout les class dans un program.

Méthode de calcul:

VNC = product{ClassCoherenceFactor(Class)}
tellque
0<= ClassCoherenceFactor(x) <= 1

impact :
-Ecrire des programmes robustes est facilement maintenable.

3.4 Component naming coherence (CMNC):

calcule la coherence de denomination de tout les composent dans un program.

Méthode de calcul:

CMNC = CNC.MCN.VNC

impact :
-Ecrire des programmes robustes est facilement maintenable.

3.5 Nubmber of imports per class (NIC):

calcule le nombre total de class importer dans une class.

Méthode de calcul:

NIC = sum{imports}

impact :
-Estimation de la charge total sur le Linker(phase compilation).
-Un grand nombre d'imports signifie une grande charge sur le compilateur est un long temps de compilation.

3.6 Number of effectif imports per class (NEIC):

Calcule le nombre total de class importer effectivement utiliser dans un class.

Méthode de calcul:

NEIC = sum{UsedImports}

impact:
-Meilleure visibilité les imports nécessaire dans une class.

3.7 Number of non effectif imports (NNEIC) :

calcule le nombre d'imports non utiliser dans une class.

Méthode de calcul:

NNEIC =NIC - NEIC

impact:
-Elimination des tranches de code non necessaire, maintenabilite.

3.8 Time reading class (T) :

Calcule le temps approximative de la lecture de la class par le compilateur.

impact :
-Estimation du temps de compilation, maintenabilité , facilité de test et de debugage.
-Dépend aussi de l'equipement utiliser.

4. Conclusions et travaux futurs.

Le document présente une suite de métrique de base pour le Test orientée objet. Les données métriques offrent une rétroaction rapide aux concepteurs de logiciels et aux gestionnaires. L'analyse et la collecte de ces données peuvent prédire la qualité de la conception. Si elles sont utilisées de manière appropriée, elles peuvent conduire à une réduction significative des coûts de mise en œuvre globale et à des améliorations de la qualité du produit final. La qualité améliorée, à son tour, réduit les efforts de maintenance future. Utiliser des indicateurs de qualité précoce basés sur des preuves empiriques objectives est donc un objectif réaliste.

À l'avenir, nous concevrons un nouvel ensemble de métriques permettant de Tester nos programmes.

References

[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.*