UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES HOUARI BOUMEDIENE FACULTÉ D'INFORMATIQUE

RAPPORT PROJET PLURIDISCIPLINAIRE

Nouvelles métriques en programmation orientée objet

Encadrant(e) :
Dr MEKAHLIA Fatma Zohra

Réalisé par :
- **ALLOUN Anis**
- **LOUTANI Meriem**
- **AGRANE Rym**
- **HIMRANE Sid Ahmed**
- **HAMDOUCHE Abdallah Samy**

Année Universitaire : 2023 – 2024

TABLE DES MATIERES :

1-Introduction ***.....3***

2-Problématique ***.....3***

3-Solution ***.....3***

4-Étape de réalisation du projet ***.....4***

5-Métriques existantes dans la littérature ***.....4***

a. Les métriques de couplage ***.....4***

- *CBO (Coupling Between Objects)* **5**
- *RFC (Response for Class)* **5**

b. Les métriques de tailles ***.....5***

- *NMO (Number of Methods)* 5
- *NOM (Number of Attributes)* 6

c. Les métriques d'héritages ***.....6***

- *DIT (Depth of Inheritance Tree)*.....6
- *NOC (Number of Children)* 6

d. Les métriques de complexité ***.....6***

- *WMC (Weighted Methods per Class)* 6
- *McCabe's cyclomatic complexity*.....7

6-Nouvelles métriques ***.....7***

- *ReusabilityMetric*.....7
- *Métrique de détection de code mort (DCM)*.....9
- Indice de Structuration du Code ISC.....10
- Taille de la Redondance Logique (TRL).....11
- Indice de Complexité Méthodologique (ICM).....12

7-Conclusion ***.....14***

Introduction :

La programmation orientée objet (POO), est un modèle de programmation qui se base sur le concept de classes et d'objets pour organiser le code de manière modulaire et réutilisable. Les classes sont des structures qui définissent des propriétés (attributs) et des comportements (méthodes), et les objets sont des instances concrètes de ces classes. Ce paradigme permet de représenter les concepts du monde réel de façon intuitive et structurée, ce qui facilite la gestion, l'évolution et la compréhension des logiciels complexes. En utilisant la POO, les développeurs peuvent créer des programmes plus facilement maintenables et extensibles.

Problematique:

Dans le domaine du développement logiciel, la qualité du code est une préoccupation majeure. Un code de haute qualité est essentiel pour garantir la fiabilité, la maintenabilité et la performance des applications. C'est pourquoi l'une des étapes critiques du cycle de vie du développement logiciel est la phase de test. Cependant cette phase est souvent très couteuse et plus longue que les phases de spécification, conception et réalisation réunies.

Solution :

Pour répondre à cette exigence et aider à l'améliorer, les métriques ont été créées. Une métrique est une mesure quantifiable utilisée pour évaluer un aspect particulier d'un système ou d'un processus. Dans le contexte du développement logiciel, les métriques jouent un rôle crucial dans l'évaluation de la qualité du code en mesurant des aspects tels que la complexité, la cohésion, le couplage et la taille d'un programme. Elles permettent d'identifier les parties du code susceptibles de contenir des erreurs ou qui sont difficiles à maintenir, ce qui améliore globalement la qualité logicielle. De plus, des modèles de prédiction des fautes utilisent ces métriques pour estimer la probabilité qu'une classe ou un module contienne des erreurs, permettant de concentrer les efforts de test sur les parties les plus risquées et de réduire ainsi le temps et les ressources nécessaires.

Les métriques aident également à améliorer la conception du programme en identifiant les défauts de conception, comme une classe avec un couplage élevé, qui peut être refactorisée pour réduire ses dépendances. Cela améliore la modularité et la maintenabilité du programme. En outre, elles permettent de comparer des programmes de taille et de complexité similaires, aidant à identifier ceux qui sont les plus efficaces et performants. Enfin, les métriques permettent de suivre l'évolution d'un programme au fil du temps, mesurant l'impact des modifications apportées au code et s'assurant que la qualité du programme ne se dégrade pas. Cependant, il est crucial de noter que les métriques ne sont que des mesures quantitatives et ne peuvent pas remplacer l'analyse humaine du code. Il est important de choisir les métriques appropriées pour le problème à résoudre. Une mauvaise utilisation des métriques peut conduire à des conclusions erronées. Les métriques doivent être utilisées comme des outils complémentaires à l'expertise humaine pour offrir une évaluation complète et précise de la qualité logicielle.

Etape de réalisation du projet :

Dans le cadre de ce projet, nous avons suivi plusieurs étapes méthodiques pour rechercher, concevoir, implémenter et évaluer de nouvelles métriques pour les programmes orientés objet :

1. **Recherche et Compréhension du Concept des Métriques**
    - Rechercher et comprendre le concept des métriques.
2. **Recherche de Métriques Existantes**
    - Effectuer une revue de la littérature pour identifier les métriques couramment utilisées dans le domaine de la programmation orientée objet.
    - Consulter des articles de recherche, des thèses et des standards de l'industrie.
    - Analyser les métriques en documentant leur définition, leurs avantages, leurs inconvénients et les contextes d'application.
3. **Étude de l'Implémentation des Métriques Existantes**
    - Analyser techniquement comment ces métriques sont implémentées, en étudiant des outils open source et des descriptions techniques.
4. **Création de Nouvelles Métriques Inspirées des Existantes**
    - Concevoir de nouvelles métriques adaptées à des besoins spécifiques en utilisant les principes et méthodes identifiés.
5. **Implémentation des Nouvelles Métriques en Java**
    - Développer le code Java pour implémenter les nouvelles métriques, en assurant une bonne structure de code, modularité et réutilisabilité.
    - Créer des tests unitaires et d'intégration pour valider le bon fonctionnement des nouvelles métriques.
    - Comparer les résultats avec des cas d'utilisation réels pour garantir leur pertinence.

Ces étapes nous ont permis de garantir une approche méthodique et complète pour l'amélioration de la qualité logicielle à travers l'utilisation et la création de nouvelles métriques pour la poo.

Métriques Existante dans la littérature :

Il existe une multitude de types de métriques pour évaluer différents aspects des programmes orientés objet, et parmi les plus couramment utilisées, on retrouve :

Les métriques de couplage :

CBO (Coupling Between Objects) :

Référence bibliographique : Chidamber, S. R., & Kemerer, A Metrics Suite for Object-Oriented Design, **1994,** IEEE Transactions on Software Engineering, 20(6), 476-493

Description : Représente le nombre de classes couplées à une classe donnée, Le couplage entre classes se produit lorsque des classes interagissent les unes avec les autres à travers des relations comme l'héritage, l'appel de méthodes, l'utilisation de variables ou de paramètres, etc.

Formule de calcul: $CBO = |C1, C2, \ldots, Cn|$

C est la classe donnée.

C1, C2, ...,Cn sont les classes distinctes avec lesquelles la classe C . Interagit directement.

$|C1$, □2, ..., □□| : représente le nombre total de ces classes distinctes.

RFC (Response for Class):

Reference bibliographies:

Briand, L. C., Daly, J. W., & Wüst, J, A unified framework for coupling measurement in object-oriented systems, **1999,** IEEE Transactions on Software Engineering, 25(1), 91-121.

Description :

Mesure le nombre total de méthodes qui peuvent être potentiellement invoquées (directement ou indirectement) à partir d'une instance de la classe donnée. En d'autres termes, RFC représente le nombre de méthodes différentes auxquelles une classe peut potentiellement répondre.

Elle est utilisée pour évaluer la complexité et la taille d'une classe en programmation orientée objet. Une RFC élevée peut indiquer une classe complexe avec de nombreuses responsabilités ou dépendances, ce qui peut rendre la classe plus difficile à maintenir, à tester et à comprendre.

Formule de calcul :

La formule de calcul de la métrique RFC (Response For Class) peut être définie comme suit :

$\mathrm{RFC} = \mathrm{NOM} + \mathrm{CBO}$ Où :

NOM (Number of Methods) : Nombre total de méthodes définies dans la classe.

CBO (Coupling Between Object classes) : Nombre de classes couplées à la classe donnée.

Les métriques de tailles :

NMO (Number of Methods) :

Reference bibliographies:

Chriram R. Chidamber and Chris F. Kemerer, A Metrics Suite for Object-Oriented Design-1994-IEEE Transactions on Software Engineeri IEEE Computer Society Press-477-491

Description :

Métrique logicielle qui mesure la complexité d'une classe en comptant le nombre de méthodes qu'elle définit.

Formule de Calcul :

Le calcul de NMO est relativement simple. Il suffit de compter le nombre de méthodes définies dans une classe.

Voici les étapes à suivre :

1. Identifier toutes les classes dans le système logiciel.
2. Pour chaque classe, compter le nombre de méthodes qu'elle définit.
3. Le NMO de la classe est le nombre de méthodes comptées

NOM (Number of Attributes) :

Référence bibliographique : S. R. Chidamber et C. F. Kemerer- Metrics Suite for Object Oriented Design 1994 IEEE Transactions on Software Engineering

Description :

Mesure de la complexité d'une classe dans un programme orienté objet,Elle est définie comme le nombre d'attributs déclarés dans une classe,Elle peut être utilisée pour comparer des classes et

identifier les classes complexes.Il est important de tenir compte des limites de la métrique NOM lors de son utilisation.

Quelques avantages de la métrique NOM : Simple à calculer, Facile à comprendre, Peut être utilisé pour comparer des classes de différents langages de programmation

Quelques inconvénients de la métrique NOM : Ne prend pas en compte la complexité des attributs,Ne prend pas en compte la relation entre les attributs, Peut être trompeur pour les classes avec des attributs simples

Les métriques d'heritages :

DIT (Depth of Inheritance Tree) :

Référence bibliographique:

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering, 22(10), 751-761.

Description :

La métrique DIT (Depth of Inheritance Tree) mesure la profondeur maximale d'une classe dans la hiérarchie d'héritage. En d'autres termes, elle indique le nombre de niveaux de classes parents entre une classe donnée et la classe racine (généralement object en Java).

NOC (Number Of Children) :

Référence bibliographique :

Basili,V. R., Briand, L.C., & Melo,W. L. (1996). A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering, 22(10), 751-761.

Description :

La métrique NOC (Number Of Children) mesure le nombre direct de sous-classes d'une classe donnée. En d'autres termes, elle indique combien de classes héritent directement de cette classe.

Les métriques de complexité :

WMC (Weighted Methods per Class):

Référence bibliographique :

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6), 476-493.

Description :

Mesure la complexité d'une classe en additionnant les pondérations de ses méthodes. Une pondération est attribuée à chaque méthode selon sa complexité. Une WMC élevée indique une classe complexe et potentiellement difficile à maintenir, tandis qu'une WMC basse suggère une classe plus simple et facile à comprendre

Formule de calcul :

$$WMC = \sum_{ni=1} Ci$$

McCabe's cyclomatic complexity:

Référence bibliographique : McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering, (4), 308-320.

Description :

La complexité cyclomatique est une métrique importante des logiciels. Elle fait référence au nombre de chemins d'exécution possibles dans un morceau de code donné, par exemple une fonction. Plus on utilise de structures de décision, plus il y a de branches possibles pour votre code.

Formule de calcul :

Pour un programme composé de n nœuds de décision (N), de branches conditionnelles (B), et de boucles (L), la complexité cyclomatique (V(G)) est calculée selon la formule suivante :

$V(G)=B-N+2L$

Il existe encore une multitude d'autres métriques, telles que les métriques de cohésion (LCOM, TCC, LCC...), les Métriques basées sur les modifications de l'état (NDC, DSM, CSD…), et bien d'autres, chacune apportant une perspective unique et enrichissante sur la qualité, la maintenabilité, et la structure des systèmes de code orienté objet.

Nouvelles métriques :

Après avoir mené des recherches approfondies et étudié les métriques existantes, nous nous en sommes inspirés pour concevoir et développer cinq nouvelles métriques. Celles-ci sont présentées comme suit :

1. ReusabilityMetric :

Description :

La classe "**ReusabilityMetric**"est conçue pour évaluer la réutilisabilité d'une classe Java en calculant un score basé sur trois aspects clés : la cohésion, le couplage et l'utilisation des interfaces. Voici une description détaillée de cette métrique :

1. Calcul du Score de Réutilisabilité :

La méthode principale de cette classe est "calculateReusability" qui combine les scores de cohésion, de couplage et d'utilisation des interfaces pour calculer un score global de réutilisabilité.

Score de Cohésion (cohesionScore) : Mesure à quel point les méthodes de la classe travaillent ensemble vers un objectif commun.

Score de Couplage (couplingScore) : Mesure le degré de dépendance de la classe par rapport aux classes externes.

Score d'Utilisation des Interfaces (interfaceUsageScore) : Évalue la flexibilité et l'extensibilité de la classe en mesurant son utilisation des interfaces.

Le score global est calculé comme la somme des scores de cohésion et d'utilisation des interfaces divisée par le score de couplage plus un.

Formule de calcul : $R = (C + I) (K + 1)$

Où C : est le score de cohésion

I : est le score d'utilisation des interfaces

K : est le score de couplage

2. Calcul du Score de Cohésion :

La méthode “calculateCohesionScore” évalue la cohésion de la classe en comptant les méthodes pertinentes qui ne sont ni des getters, ni des setters, ni des constructeurs.

Nombre de Méthodes Pertinentes (relevantMethodsCount) : Méthodes qui ne sont pas des getters, des setters ou des constructeurs.

Score de Cohésion (cohesionScore) : évalue des méthodes pertinentes par rapport au nombre total de méthodes déclarées.

Formule de calcul : CohesionScore = (NBR de méthodes) / (NBR total de méthodes)

3. Calcul du Score de Couplage :

La méthode “calculateCouplingScore” mesure le couplage en comptant les champs publics et les types de paramètres des méthodes qui ne font pas partie des packages standard de Java.

Nombre de Dépendances Externes (externalDependenciesCount) : Champs publics et types de paramètres des méthodes non standards.

Score de Couplage (couplingScore) : évalue des dépendances externes par rapport au nombre total de champs et de méthodes.

Formule de calcul :

CouplingScore = (NBR de dépendances externes) / (NBR total de champs et de méthodes)

4. Calcul du Score d'Utilisation des Interfaces :

La méthode “calculateInterfaceUsageScore” évalue l'utilisation des interfaces par la classe.

Nombre d'Interfaces Implémentées (implementedInterfacesCount) : Interfaces implémentées par la classe.

Score d'Utilisation des Interfaces (interfaceUsageScore) : évalue des interfaces implémentées par rapport au total (avec +1 pour éviter la division par zéro).

Formule de calcul :

InterfaceUsageScore = (NBR d’interfaces implémentées) / (NBR_total d’interfaces)

Objectif :

La métrique de réutilisabilité, implémentée par la classe “ReusabilityMetric”, a pour objectif d'évaluer quantitativement la facilité avec laquelle une classe Java peut être réutilisée dans différents contextes. Elle combine trois aspects essentiels de la conception du code pour fournir un score global de réutilisabilité : la cohésion, le couplage et l'utilisation des interfaces.

Conclusion :

Cette métrique de réutilisabilité fournit une évaluation quantifiée de la réutilisabilité des classes en Java en intégrant des aspects de conception essentiels tels que la cohésion, le couplage et l'utilisation des interfaces. Les résultats peuvent aider à identifier les classes qui

sont bien conçues pour être réutilisées, ainsi que celles qui pourraient bénéficier d'améliorations pour augmenter leur réutilisabilité.

2. Métrique de détection de code mort (DCM) :

Description :

La métrique DCM, ou Détection de Code Mort, est un processus d'analyse statique du code source visant à identifier les parties inutilisées d'une application logicielle, communément appelées "code mort". Ces fragments de code peuvent résulter d'erreurs de conception, de refactoring inapproprié, de fonctionnalités abandonnées, de commentaires désactivés ou de blocs conditionnels inutilisés.

L'objectif principal de la métrique DMC est de repérer ces segments de code superflus afin de les éliminer, améliorant ainsi la qualité, la maintenabilité et les performances globales du logiciel.

En scrutant le code source, la métrique DCM permet aux développeurs d'identifier les zones de complexité inutile et de risques potentiels, facilitant ainsi la décision de refactoriser et d'optimiser le code. En intégrant cette détection de codes inutilisés dans leur processus de développement, les équipes logicielles peuvent garantir un code plus propre, plus efficace et plus robuste, réduisant ainsi les risques de bugs et améliorant l'expérience utilisateur finale.

Objectifs et utilités de la Métrique DCM :

- Optimisation du Code : En identifiant les parties non utilisées, la métrique DCM encourage les développeurs à nettoyer et à optimiser le code, réduisant ainsi la complexité inutile et améliorant sa lisibilité.

- Réduction de la Taille du Code : En éliminant le code mort, la taille du code source diminue, réduisant ainsi le temps de compilation, de déploiement et la consommation de mémoire.

- Détection d'Erreurs Potentielles : Les sections de code inutilisées peuvent contenir des erreurs non détectées, la métrique DMC aide à les repérer, améliorant ainsi la fiabilité globale du logiciel.

- Meilleure Compréhension du Système et du Code : En éliminant le code mort, les développeurs obtiennent une vision plus claire du système, facilitant l'intégration de nouvelles fonctionnalités et la maintenance du logiciel.

Conclusion :

La métrique DMC est un outil essentiel pour les équipes de développement, contribuant à la création de programmes plus propres, optimisés et évolutifs. En éliminant le code mort, elle améliore la qualité et la maintenabilité du logiciel, conduisant à une meilleure expérience utilisateur et à une réduction des risques de bugs. Ainsi, la métrique DCM joue un rôle crucial dans le processus de développement logiciel, garantissant la production de logiciels robustes et fiables.

3. Indice de Structuration du Code ISC :

Description

L'Indice de Structuration du Code (ISC) est une mesure composite conçue pour évaluer la qualité de la structuration du code source en tenant compte de plusieurs aspects essentiels : la cohésion, le couplage et la modularité. Ces aspects sont mesurés à l'aide de métriques standard telles que LCOM (Lack of Cohésion in Methods), FANIN (Fan-in), FANOUT (Fan-out) et CBO (Coupling Between Objects). L'ISC fournit une évaluation globalisée de la qualité du code, facilitant l'identification des points forts et des faiblesses structurelles.

Objectif

L'objectif de l'ISC est de fournir aux développeurs et aux équipes de développement un outil de mesure unique et normalisé pour évaluer la qualité structurelle du code. En combinant plusieurs métriques en une seule, l'ISC permet de mieux comprendre la maintenabilité, la modularité et le niveau de couplage du code. Une haute valeur d'ISC indique une bonne structuration du code, facilitant la maintenance et l'évolution du logiciel, tandis qu'une basse valeur met en lumière des besoins potentiels en refactorisation.

Formule et Explication de la Normalisation :

Formule de l'ISC : ISC = ((NC+ NM) - (NCBO+1)) / 2

Explication et Normalisation des Composantes :

- ***LCOM (Lack of Cohesion in Methods):*** Mesure la cohésion d'une classe en évaluant si les méthodes partagent ou non des variables. Une faible LCOM indiquerait une forte cohésion, ce qui est important pour notre métrique d'ISC.

Normalisation : NC (Normalized Cohésion) =1-LCOM/MAXLCOM

Cette fomule assure que la cohésion est comprise entre 0 (faible cohésion) et 1 (haute cohésion).

- ***FANIN (Fan-in) et FANOUT (Fan-out) :***

FANIN (Fan-in) : Mesure le nombre de modules qui utilisent un module donné. Cela peut nous aider à évaluer la réutilisabilité du code et l'impact de chaque module sur le reste de system. ***FANOUT (Fan-out) :*** Mesure le nombre de modules utilisés par un module donné. Cela peut nous aider à évaluer la complexité et la dépendance de chaque module.

Modularité : NM (Normalized Mod) = (FANIN+FANOUT)/2*MAX (FANIN, FANOUT)

Ces formules assurent que la modularité est comprise entre 0 (faible modularité) et 1 (haute modularité).

- ***CBO (Coupling Between Objects) :*** Mesure le nombre de classes sur lesquelles une classe donnée dépend. Cela peut nous aider à évaluer le niveau de dépendance entre les différentes parties du code.

Normalisation : NCBO (Normalized Coupling) =1-CBO/MAXCBO

Cette formule inverse le couplage pour que 0 représente un couplage maximal et 1 représente un couplage minimal (meilleur couplage), alignant ainsi avec l'objectif de faible couplage.

Plages de valeurs pour l'ISC :

Haute valeur d'ISC (proche de 1) :

- **Cohésion élevée** : Méthodes et attributs bien liés.
- **Modularité élevée** : Bonne gestion des dépendances et responsabilités.
- **Couplage faible** : Peu de dépendances, facilitant la maintenance.

Valeur moyenne d'ISC (proche de 0.5) :

- **Cohésion moyenne** : Méthodes et attributs modérément liés.
- **Modularité moyenne** : Certaines améliorations possibles.
- **Couplage modéré** : Quelques dépendances.

Basse valeur d'ISC (proche de 0) :

- **Cohésion faible** : Méthodes et attributs peu liés.
- **Modularité faible** : Mauvaise gestion des responsabilités et dépendances.
- **Couplage élevé** : Nombreuses dépendances, compliquant la maintenance.

Conclusion :

L'ISC est une métrique puissante pour évaluer la qualité de la structuration du code. En combinant les métriques de cohésion, modularité et couplage, il fournit une mesure globale de la qualité du code. Cependant, il est important de comprendre que l'ISC est une métrique agrégée et ne capture pas tous les aspects de la qualité du code. Il doit être utilisé en conjonction avec d'autres métriques et techniques d'analyse pour obtenir une image complète de la qualité du code.

4. Taille de la Redondance Logique (TRL) :

Description de la Métrique (TRL) :

La Taille de la Redondance Logique (TRL) est une métrique conçue pour quantifier la fréquence de répétition des structures de contrôle logiques au sein d'un code source. Elle mesure la diversité et l'unicité des motifs logiques En fournissant une indication de la redondance logique globale, cette métrique aide à identifier les sections de code où des structures de contrôle similaires sont utilisées de manière répétée. De plus, la TRL permet de repérer les segments de code qui accomplissent des tâches identiques ou très similaires, mais avec des implémentations légèrement différentes. Cela peut révéler des opportunités pour rationaliser et unifier le code, améliorant ainsi la maintenabilité et la performance globales.

Objectifs de la Métrique TRL :

- **Réduire les Redondances :** Identifier et minimiser les répétitions inutiles de motifs logiques pour améliorer la clarté et l'efficacité du code.
- **Diminuer la Complexité :** Favoriser l'utilisation de structures logiques diversifiées pour rendre le code moins complexe et plus facile à comprendre.

- **Réduire la Taille du Code** : Éliminer les redondances permet de réduire la taille globale du code source, rendant le projet plus léger et plus facile à maintenir.
- **Faciliter le Refactoring :** Aider les développeurs à repérer les opportunités de refactorisation pour unifier et optimiser le code, rendant ainsi le projet plus cohérent et performant.

Formule de Calcul de TRL :

$$TRL = \left(\frac{Nombretotaldemotifs\log iques}{Nombredemotifs\log iques\ distincts}\right) \cdot 100\ \%$$

Étapes de Calcul :

1. Identification des Motifs Logiques Distincts : Analyser le code source pour identifier les différents types de structures de contrôle logiques utilisées (ex. `if`, `for`, `while`, `switch`, `Try`, `catch`, etc.).
2. Comptage des Occurrences Totales de Motifs Logiques : Compter le nombre total d'occurrences de toutes les structures de contrôle logiques dans le code.
3. Calcul du Pourcentage de Redondance Logique :

. Diviser le nombre de motifs logiques distincts par le nombre total de motifs logiques.

. Multiplier le résultat par 100 pour obtenir le pourcentage de la TRL.

Conclusion :

En résumé, la Taille de la Redondance Logique (TRL) offre une évaluation fine de la répétition des structures de contrôle logiques dans le code source. En réduisant les redondances, la complexité et la taille du code, elle favorise une maintenance plus aisée et une meilleure performance. Par l'identification des motifs logiques récurrents, elle guide les développeurs vers des pratiques de codage plus efficaces, améliorant ainsi la clarté et la robustesse du logiciel.

5. Indice de Complexité Méthodologique (ICM) :

Description :

L'Indice de Complexité Méthodologique (ICM) évalue la complexité d'une classe en tenant compte de la distribution et de la diversité de ses méthodes. Il combine plusieurs facteurs tels que le nombre total de méthodes, les modificateurs d'accès, les propriétés spécifiques des méthodes (comme static, final, abstract), et les relations d'héritage pour fournir une mesure unique de complexité.

Composants :

1.Nombre Total de Méthodes (NTM)

Description : Nombre total de méthodes dans une classe, y compris les méthodes héritées.

2.Complexité des Modificateurs d'Accès (CMA)

Formule : $CMA = \frac{PM \times wPM + PRM \times wPRM + PTM \times wPTM + DAM \times wDAM}{NTM}$

- **PM :** Nombre de méthodes publiques.
- **PRM :** Nombre de méthodes privées.
- **PTM :** Nombre de méthodes protégées.

- **DAM :** Nombre de méthodes avec accès par défaut.
- Où Wpm, Wprm, Wptm, Wdam sont des poids attribués respectivement aux méthodes publiques, privées, protégées et par défaut. Les poids peuvent être ajustés en fonction de l'importance ou de la visibilité des méthodes dans le contexte du projet

3.Propriétés Spécifiques des Méthodes (PSM)

Formule : $PSM = \frac{FM \times wFM \times SM + wSM + AM \times wAM}{NTM}$

FM : Nombre de méthodes finales.
SM : Nombre de méthodes statiques.
AM : Nombre de méthodes abstraites.
Où Wfm, Wsm, Wam sont des poids attribués respectivement aux méthodes finales, statiques et abstraites.

4.Indice d'Héritage (IH)

Formule : $IH = \frac{IM + OMA}{2}$

IM : Nombre de méthodes héritées.
OM : Nombre de méthodes surchargées.

5.Diversité Méthodologique (DM)

Formule : $DM = 1 - \frac{1}{1 + e^{-\left(\frac{NTM - \mu}{\sigma}\right)}}$

e : Constante mathématique (approx. 2.71828).
mu : Moyenne du nombre total de méthodes dans l'ensemble des classes du projet.
sigma : Écart type du nombre total de méthodes dans l'ensemble des classes du projet.

Formule Finale :

L'Indice de Complexité Méthodologique (ICM) est calculé en combinant tous les composants mentionnés ci-dessus pour obtenir une mesure unique de la complexité.

$$ICM = NTM \times (CMA + PSM + IH) \times DM$$

Utilisation Pratique et Indications des Valeurs :

1-Valeur Minima (ICM < 3) : Classe extrêmement simple, probablement sous-développée ou avec très peu de méthodes.

2-Valeur Basse (3 < ICM < 10) : Classe relativement simple, avec peu de méthodes et une faible complexité.

3-Valeur Modérée (10 ≤ ICM < 50) : Classe de complexité moyenne, bien équilibrée en termes de méthodes et de modificateurs d'accès.

4-Valeur Élevée (50 ≤ ICM < 250) : Classe complexe avec un grand nombre de méthodes et une diversité élevée. Peut nécessiter une refactorisation pour améliorer la maintenabilité.

5-Valeur Très Élevée (ICM ≥ 250) : Classe très complexe. Forte probabilité de problèmes de maintenabilité et de besoin urgent de refactorisation.

Les formules choisies visent à fournir une évaluation équilibrée et complète de la complexité des méthodes d'une classe. Voici les raisons spécifiques :

NTM : Indicateur simple de la taille brute de la classe, utile pour identifier les classes potentiellement surchargées.

CMA : Balance l'impact des différents modificateurs d'accès. Une classe bien conçue aura une distribution équilibrée de méthodes avec différents niveaux d'accès, reflétant une bonne encapsulation.

PSM : Les propriétés spécifiques des méthodes (final, static, abstract) influencent la flexibilité et la réutilisabilité. Une classe avec trop de méthodes de ces types peut être rigide ou difficile à tester.

IH : L'héritage est fondamental en programmation orientée objet, mais une mauvaise gestion peut introduire une complexité inutile. Cette métrique aide à évaluer cet aspect.

DM : La diversité méthodologique donne une idée de la spécialisation d'une classe par rapport au projet global, évitant une mesure trop centrée sur des valeurs extrêmes.

Cas d'Utilisation :

- **Vue d'ensemble** : En combinant différentes mesures, l'ICM fournit une vue d'ensemble complexité de la classe, en tenant compte à la fois du nombre et de la nature des méthodes et de l'influence de l'héritage.
- **Analyse Avancée de la Complexité du Code :** Permet de comparer la complexité de différentes classes au sein du même projet ou entre différents projets, ce qui aide à identifier les classes particulièrement complexes ou potentiellement problématiques.
- **Planification de la Maintenance :** Des valeurs d'ICM plus élevées peuvent indiquer des classes qui peuvent nécessiter plus d'efforts pour être comprises, maintenues et modifiées, aidant les développeurs à prioriser le refactorage ou les efforts de documentation supplémentaires.
- **Optimisation de la Conception du Code :** Aider à évaluer et à améliorer la conception des classes pour assurer un bon équilibrage entre fonctionnalité et maintenabilité.
- **Audit de Qualité du Code :** Utiliser l'ICM comme indicateur de la qualité du code pour les revues de code et les audits de qualité.

Conclusion :

L'intégration de nouvelles métriques telles que la **ReusabilityMetric**, la **Détection de Code Mort (DCM)**, l'**Indice de Structuration du Code (ISC)** et la **Taille de la Redondance Logique (TRL)** enrichit notre approche d'évaluation de la qualité du code.

ReusabilityMetric(RM) permet d'évaluer la réutilisabilité d'une classe Java en analysant la cohésion, le couplage et l'utilisation des interfaces. Cette métrique aide à identifier les classes qui sont facilement réutilisables, favorisant ainsi un code modulaire et durable.

La **métrique de Détection de Code Mort (DCM)** identifie les fragments de code inutilisés ou obsolètes. En supprimant ces segments superflus, nous améliorons la maintenabilité, la performance et la clarté du code, réduisant ainsi les risques de bugs.

L'**Indice de Structuration du Code (ISC)** offre une vue d'ensemble de la qualité structurelle du code en tenant compte de la cohésion, du couplage et de la modularité. Cette analyse aide à repérer les points faibles structurels, facilitant les efforts de refactoring pour obtenir un code plus robuste et maintenable.

La **Taille de la Redondance Logique (TRL)** mesure la fréquence des motifs logiques répétitifs dans le code source. En identifiant les sections de code redondantes, la TRL permet de rationaliser et d'unifier le code, améliorant ainsi la maintenabilité et la performance globale.

Enfin, L'Indice **de Complexité Méthodologique (ICM**) évalue la complexité des classes en prenant en compte divers aspects comme le nombre de méthodes, les modificateurs d'accès, les propriétés spécifiques des méthodes, l'héritage et la diversité méthodologique. Cette métrique fournit une mesure unique de la complexité, aidant les développeurs à évaluer la qualité du code de manière équilibrée et détaillée.

En conclusion, l'utilisation de ces nouvelles métriques fournit une évaluation complète et détaillée de la qualité du code source. Elles permettent d'identifier les points faibles et les opportunités d'amélioration. En intégrant ces métriques dans notre processus de développement, nous pouvons garantir un logiciel plus propre, performant et facile à maintenir, répondant mieux aux exigences des utilisateurs et du marché.