



[Rapport Qualité]

Analyse qualité Sonar/ EvalMetrics



Réalisé par :

BOUNIETE Safae

LAMSELLAK Hajar

Année Universitaire : 2017/2018

Sommaire

I. INTRODUCTION	3
II. PRESENTATION GENERALE DU PROJET	3
1. Objectif	3
2. Description des outils	3
1.1 Premier pas avec EvalMetrics.....	3
1.2 Premier pas avec Sonar	5
III. ANALYSE DU PROJET JUNIT PAR EVALMETRICS	9
1. Métriques, caractéristiques et interprétation	9
1.1 Métriques, caractéristiques et interprétation par projet	9
a. Analyse des caractéristiques	9
b. Analyse des sous caractéristiques	10
c. Analyse des propriétés	10
1.2 Métriques, caractéristiques et interprétation par classe.....	11
1.3 Métriques, caractéristiques et interprétation par méthode	11
2. Résultat de test sur la classe TestCase.....	12
3. Statistiques	12
3.1 Nombre des lignes	12
3.2 Méthodes	12
IV. ANALYSE DU PROJET JUNIT PAR SONAR.....	13
1. Introduction	13
2. Analyse des résultats obtenus.....	13
2.1 Par projet :	13
2.2 Par classe :	16
V. COMPARAISON DES DEUX OUTILS SONAR ET EVALMETRICS	18
VI. CONCLUSION	19

I. INTRODUCTION

En informatique et en particulier en génie logiciel, la **qualité logicielle** est une appréciation globale d'un logiciel, basée sur de nombreux indicateurs et plusieurs outils offrent une solution performante de son contrôle. Mais qu'est-ce **que la qualité d'un logiciel**, et en quoi est-il important de la **contrôler** ?

*En paraphrasant **Wikipédia**, la gestion de la qualité est l'ensemble des activités qui concourent à l'obtention de la qualité dans un cadre de production de biens ou de services (dans notre cas, d'un logiciel). Plus largement, c'est aussi un moyen que se donnent certaines organisations, dans des buts tels que la mise en conformité par rapport aux standards du marché.*

Dans le monde **informatique** en général, et en Java en particulier, la qualité d'une application va être directement liée à la **qualité du code**. De nombreux outils s'affairent à contrôler certains aspects de cette qualité du code : exécution de tests unitaires, analyse de la couverture du code par ces tests, vérifications du respect des règles de codage, etc. Il est donc possible de contrôler la qualité de son code grâce à ces outils, et d'avoir une confiance accrue en son application !

Le contrôle de la qualité va donc pousser l'équipe de développement à adopter et à respecter certains standards de développement. Le but de tout cela étant bien entendu de rendre le code plus sûr, mais de permettre d'y déceler les erreurs le plus rapidement possibles... et donc de les corriger !

II. PRESENTATION GENERALE DU PROJET

1. Objectif

Dans le présent projet, nous ferons une étude comparative entre les outils d'évaluation de la qualité, basé sur des modèles de qualité de collecte et calcule de la métrique pour analyser la qualité du code Junit.

En premier temps, nous allons faire une analyse avec Sonar et EvalMetrics, après nous présentons la comparaison entre ces outils, et on termine par une conclusion.

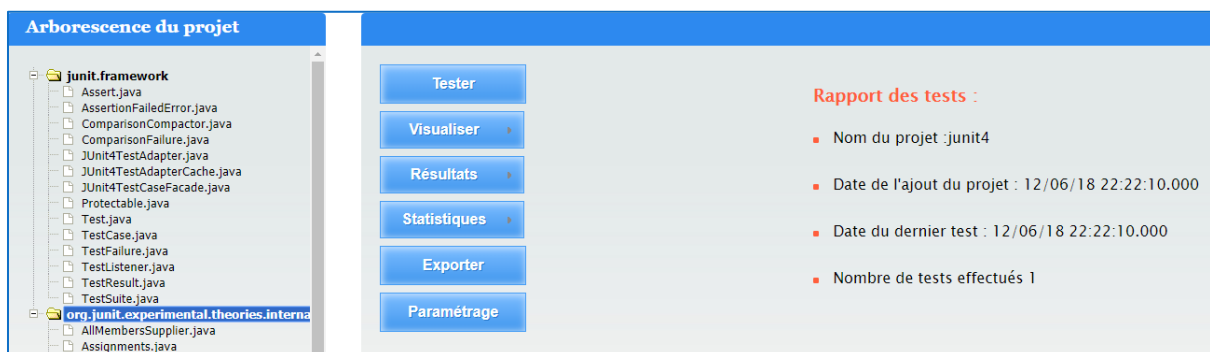
2. Description des outils

1.1 Premier pas avec EvalMetrics

a. Présentation

EvalMetrics est un outil de mesure de qualité open source, basé sur la norme ISO 25000 (ISO / CEI, 2015), est développé par les étudiants de l'école nationale des sciences appliquées de Oujda.

Le but principal de cet outil est d'évaluer le logiciel pendant son développement.



b. Principes fonctionnalités

En effet, ce logiciel a pour objectif d'évaluer un projet pendant son développement, en utilisant plusieurs mesures pour analyser sa qualité, à savoir :

- Taille du code (nombre de lignes par méthode, par classe ...)
- taille du commentaire (Densité des lignes de commentaire),
- code dupliqué (nombre de lignes dupliquées)
- complexité (complexité moyenne par méthode, par classe)
- dépendances (packages de dépendances de classe dépendances ...)
- règles de codage (Violations of Sun code conventions)
- la détection de modèles de conception.

c. Analyse d'un projet par EvalMetrics

EvalMetrics est capable d'analyser des projets réalisés avec java, en utilisant des métriques. Le développeur peut également déclencher une analyse pendant la phase de développement pour lui permettre d'anticiper la qualité et la corriger même avant la sortie.

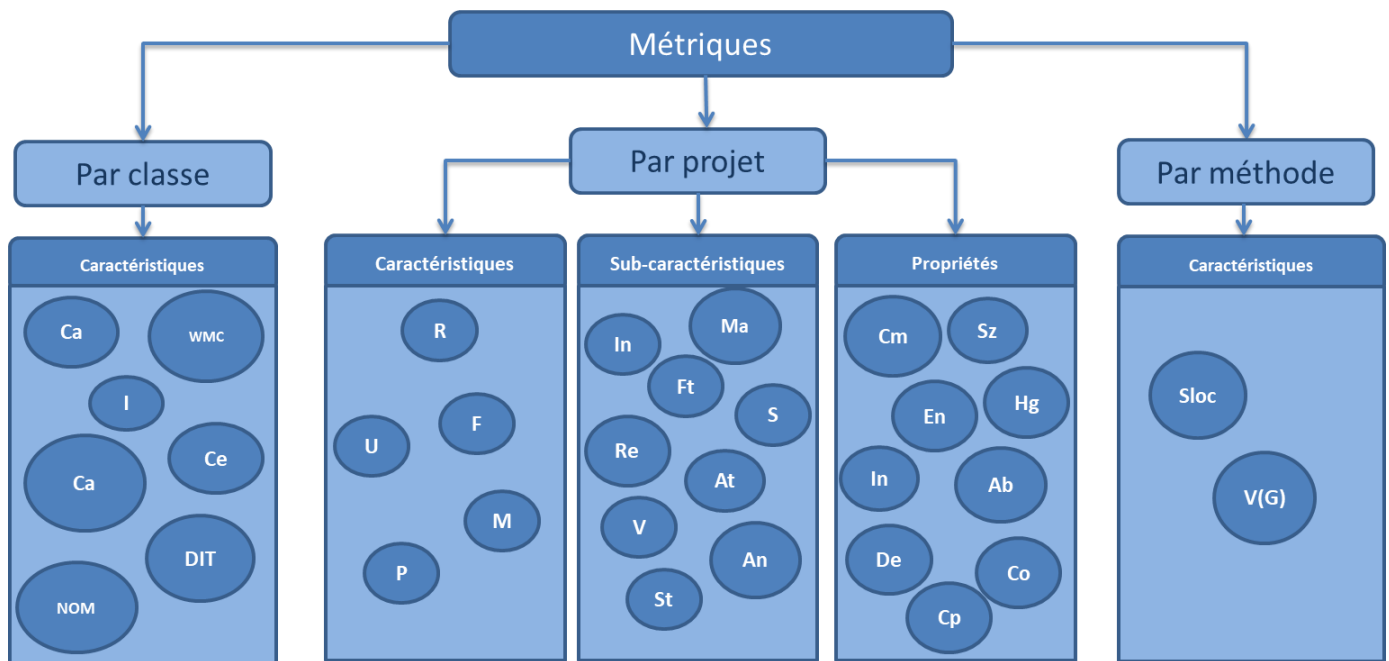
Concernant les résultats, EvalMetrics offre une possibilité d'extraire des rapports et des graphiques pour faciliter cette analyse.

d. Les règles d'EvalMetrics

EvalMetrics, étant basé sur la norme **ISO 25000**, il définit donc un modèle composé de six caractéristiques générales qui définissent la qualité globale d'une application, à savoir :

- **Fonctionnalité**
- **Fiabilité**
- **Utilisabilité Efficacité**
- **Maintenabilité**
- **Portabilité**

Cependant, pour calculer ces métriques, EvalMetrics propose une division de parties, la figure suivante répertorie les métriques disponibles dans cet outil :



SLOC : Le nombre total de lignes de codes dans la méthode.

V(G) (McCabe Cyclomatic Complexity) : La complexité cyclomatique d'une méthode.

WMC : La somme de la complexité cyclomatique de McCabe pour toutes les méthodes de la classe.

CA : Le nombre de classes hors d'un package qui dépendent d'une classe dans le package.

CE : Le nombre de classes dans un package qui dépendent d'une classe d'un autre package.

RMI (Instability) : Ce nombre vous donnera l'instabilité de votre projet. C'est-à-dire les dépendances entre les paquets.

DIT (Depth of Inheritance Tree): Distance jusqu'à la classe Object dans la hiérarchie d'héritage.

NOM (Number of Methods): Le nombre de méthodes dans l'élément sélectionné.

RMD (Distance) : Ce nombre devrait être petit, proche de zéro pour indiquer une bonne conception des parquets.

1.2 Premier pas avec Sonar

a. Présentation

Sonar est un outil open source dont le but principal est de fournir une analyse complète de la qualité d'une application en fournissant de nombreuses métriques sur ses projets qui permettent d'évaluer la qualité du code, et d'en connaître l'évolution au cours du développement.

Ceci permet aux développeurs d'accéder aux données d'analyse et découvrir les erreurs de style présentes, les éventuels bugs et défauts de code, la duplication de code, le manque de couverture de test et de la complexité excessive. Tout ce qui affecte notre base de code, des détails de style mineurs aux erreurs de conception critiques, est inspecté et évalué par Sonar.

b. Architecture

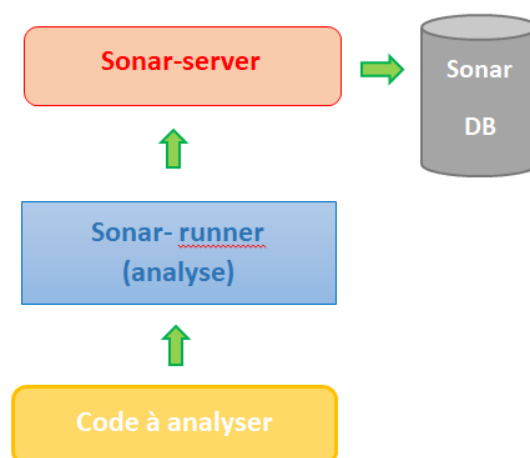
D'un point de vue architectural, Sonar est composé de trois couches principales:

- **Un exécuteur** basé sur Maven, Ant ou un exécuteur Java, dont le but sera de lancer un certain nombre

D'outils d'analyse, et d'en agréger les résultats.

- **Le serveur web** qui permet la navigation et

La consultation des analyses réalisées sur les projets.



c. Principes fonctionnalités

Sonar est capable d'analyser des projets de différents langages. Cependant, les résultats et les fonctionnalités proposées par cet outil sont généralement indépendants du langage.

Nous listons ici les principales fonctionnalités de l'outil Sonar qui seront décrites en détail dans la partie suivante de ce rapport :

- **Tableau de bord** complet des différents projets suivis.
- Détection rapide du **code à risque**.
- **Mesures quantitatives** : nombre de classes, duplication de code, etc.
- **Mesures qualitatives** : couverture et taux de réussite des tests, complexité du code, Respect des règles de codage...
- **Historiques des statistiques**, pour en voir l'évolution au cours du temps.

d. Profils, et Règles de Sonar

En ce qui concerne les métriques utilisées par Sonar, ils sont regroupés en catégories :

- ✓ Métriques de **complexité**

Nom	La description
Complexity	C'est la complexité calculée en fonction du nombre de chemins à travers le code.

- ✓ Métriques de **sévérité**

Sévérité	Description
Blocker	Ce problème peut rendre l'application entière instable en production.
Critical	Ce problème peut entraîner un comportement inattendu en production sans affecter l'intégrité de l'ensemble de l'application.
Major	Ce problème pourrait avoir un impact important sur la <i>productivité</i> .
Minor	Ce problème pourrait avoir un impact potentiel et mineur sur la <i>productivité</i> .
Info	Risque de sécurité inconnu ou pas encore bien défini ou impact sur la productivité.

✓ Métriques de **duplications**

Nom	La description
Duplicated blocks	Nombre de blocs de lignes dupliqués.
Duplicated files	Nombre de fichiers impliqués dans les duplications.
Duplicated lines	Nombre de lignes impliquées dans les duplications.
Duplicated lines (%)	Densité de duplication = lignes dupliquées / lignes * 100

✓ Métriques de **taille**

Métrique	Description
Classes	Nombre de classes (y compris les classes imbriquées, les interfaces, les énumérations et les annotations).
Comment lines	Nombre de lignes contenant un commentaire ou un code commenté.
Comments (%)	<p>Densité des lignes de commentaires = Lignes de commentaires / (Lignes de code + Lignes de commentaires) * 100</p> <p>Avec une telle formule:</p> <ul style="list-style-type: none"> 50% signifie que le nombre de lignes de code est égal au nombre de lignes de commentaires 100% signifie que le fichier ne contient que des lignes de commentaires
Directories	Nombre de répertoires
Files	Nombre de fichiers

Métrique	Description
Lines	Nombre de lignes physiques.
Lines of code	Nombre de lignes physiques contenant au moins un caractère qui n'est ni un espace ni une tabulation ni une partie d'un commentaire.
Lines of code per language	Lignes sans commentaire de code réparties par langue
Functions	Nombre de fonctions Selon la langue, une fonction est une fonction ou une méthode ou un paragraphe.
Projects	Nombre de projets dans une vue.
Statements	Nombre de déclaration

✓ Métriques de **tests**

Métrique	La description
Couverture	C'est un mélange de couverture de ligne et de couverture de condition . Son objectif est de fournir une réponse encore plus précise à la question suivante: Quelle part du code source a été couverte par les tests unitaires?
Couverture de ligne sur le nouveau code	Identique à la couverture de ligne mais limitée au code source nouveau / mis à jour.
Conditions non couvertes sur le nouveau code	Identique aux conditions Uncovered mais limité au code source nouveau / mis à jour.
Lignes non couvertes sur le nouveau code	Identique aux lignes Uncovered mais limité au code source nouveau / mis à jour.
Condition de couverture sur le nouveau code	Identique à la couverture Condition mais limitée au code source nouveau / mis à jour.
Couverture sur le nouveau code	Identique à la couverture mais limité au code source nouveau / mis à jour.
Résultats de couverture de ligne	Liste des lignes couvertes.
Résultats de couverture de condition	Liste des conditions couvertes.
Conditions couvertes par ligne	Nombre de conditions couvertes par ligne.
Conditions par ligne	Nombre de conditions par ligne.
Conditions non couvertes	Nombre de conditions qui ne sont pas couvertes par des tests unitaires.

Métrique	La description
Lignes découvertes	Nombre de lignes de code qui ne sont pas couvertes par les tests unitaires.
Lignes à couvrir	Nombre de lignes de code qui pourraient être couvertes par des tests unitaires (par exemple, les lignes vides ou les lignes de commentaires complètes ne sont pas considérées comme des lignes à couvrir).
Tests unitaires ignorés	Nombre de tests unitaires ignorés.
Échecs de test unitaire	Nombre de tests unitaires qui ont échoué avec une exception inattendue.
Erreurs de test unitaires	Nombre de tests unitaires qui ont échoué
Tests unitaires	Nombre de tests unitaires.
Couverture de ligne	Sur une ligne de code donnée, la couverture de Ligne répond simplement à la question suivante: Cette ligne de code a-t-elle été exécutée lors de l'exécution des tests unitaires? C'est la densité des lignes couvertes par des tests unitaires.
Couverture de condition	Sur chaque ligne de code contenant des expressions booléennes, la couverture de la condition répond simplement à la question suivante: 'Est-ce que chaque expression booléenne a été évaluée à la fois vrai et faux?'. C'est la densité des conditions possibles dans les structures de contrôle de flux qui ont été suivies pendant l'exécution des tests unitaires.
Densité de succès de test unitaire (%)	Test de densité de succès = (Tests unitaires - (Erreurs de test unitaires + échecs de test unitaire)) / Tests unitaires * 100
Durée des tests unitaires	Temps nécessaire pour exécuter tous les tests unitaires.

III. ANALYSE DU PROJET JUNIT PAR EVALMETRICS

Après avoir mesuré la qualité du code source JUnit, nous avons obtenus les résultats suivants :

1. Métriques, caractéristiques et interprétation

1.1 Métriques, caractéristiques et interprétation par projet

a. Analyse des caractéristiques

■ Résultat 1:

métriques , caractéristiques et interprétation par projet												
Analyse des caractéristiques, Sub-caractéristiques, propriétés,et métriques												
liste des caractéristiques												
	date	F	F Int	R	R I	U	U I	M	M I	P	P I	
S	P	2018-06-12	22.67	0	0.67	0	0.33	0	83.85	25.7965	4.31	2.3333
<< < 1 > >> Aller à la page: 1 Nombre d'enregistrement: 10 Afficher 1 à 1												

■ Interprétation des résultats :











L'image ci-dessus représente les caractéristiques par projet, qui sont la fonctionnalité, la réalisabilité, l'utilisabilité et la portabilité.

Dans le cas de notre projet JUnit, on remarque que les 4 premières caractéristiques sont marquées par des nuages avec la pluie, ce qui signifie une maintenance difficile aux niveaux des besoins fonctionnels exprimés, l'utilisation du logiciel, les efforts qu'il faut entreprendre pour le faire fonctionner, ainsi que les possibilités de faire évoluer Junit dans le cas de nouveaux besoins.

En effet, cette analyse a été effectuée juste pour une portion du code JUnit, et non pas le code entier, du coup EvalMetrics a détecté des problèmes au niveau de ces caractéristiques, puisqu'il manque des classes. D'autre part, on trouve que l'indice de portabilité est marqué avec un symbole de soleil, ce qui exprime que notre projet est portable.

b. Analyse des sous caractéristiques

■ Résultat 2:











Sub-caractéristiques - 2018-06-12T22:22:10																							
In	In I	S	S I	Ma	Ma I	FT	FT I	Re	Re I	At	At I	V	V I	An	An I	St	St I	In	In I				
113.34		2.25	 undefined	 undefined	1	 0	1	 0	1	 0	 undefined	162.32	 147.605	75.24	 1.212	0	 1						
<div><< < 1 > >> Aller à la page: 1 Nombre d'enregistrement: 10</div>																			Afficher 1 à 1 de				

■ Interprétation des résultats :

Cette image présente un ensemble des sous caractéristiques du projet, dont la plupart sont présentées avec des nuages et la pluie, sauf l'interopérabilité et la stabilité du code, cela est dû au fait que ces sous caractéristiques dépendent de la caractéristique globale.

c. Analyse des propriétés

■ Résultat 3:








































Propriétés - 2018-06-12T22:22:10																				
Cm	Cm I	Sz	Sz I	Hg	Hg I	En	En I	Ab	Ab I	In	In I	De	De I	Me	Me I	Co	Co I	Cp	Cp I	
16.7	 3	582.92	 3	1	 3	0.83	 2.4858	0.05	 0.15	0	 1	-0.95	 0.924	677	 1.5	0.61	 1.5	225.67	 1.5	
<div><< < 1 > >> Aller à la page: 1 Nombre d'enregistrement: 10</div>																			Afficher 1 à 1 de 1	

■ Interprétation des résultats :

EvalMetrics a ajouté au modèle un autre niveau qui contient les propriétés de qualité, qui présente un indicateur élémentaire d'un logiciel, dans notre cas la plupart des propriétés répondent aux besoins des fonctionnalités.

1.2 Métriques, caractéristiques et interprétation par classe

■ Résultat 1:

liste des classes																	+ Ajouter
Test	Nom Classe	WMC	interpretation	CA	interpretation	CE	interpretation	DF	interpretation	Distance	NOM	interpretation	DT	interpretation	date		
1	junit.framework.Assert	52	 1	0	 3	0	 3	0	 1	-0.95	39	 1	1	 3	2018-06-12		
1	junit.framework.AssertionFailedError	1	 3	0	 3	0	 3	0	 1	-0.95	3	 3	1	 3	2018-06-12		
1	junit.framework.ComparisonCompactor	16	 3	0	 3	0	 3	0	 1	-0.95	8	 3	1	 3	2018-06-12		
1	junit.framework.ComparisonFailure	3	 3	0	 3	0	 3	0	 1	-0.95	4	 3	1	 3	2018-06-12		
1	junit.framework.JUnit4TestAdapter	13	 3	0	 3	0	 3	0	 1	-0.95	12	 3	1	 3	2018-06-12		
1	junit.framework.JUnit4TestAdapterCache	14	 3	0	 3	0	 3	0	 1	-0.95	5	 3	1	 3	2018-06-12		
1	junit.framework.JUnit4TestCaseFacade	4	 3	0	 3	0	 3	0	 1	-0.95	5	 3	1	 3	2018-06-12		

■ Interprétation des résultats :

Concernant l'analyse par classe, la majorité des métriques des classes sont représentés par un symbole de soleil, ce qui signifie que notre projet JUnit maintenable.

- Prenons l'indice WCM (La somme de la complexité cyclomatique de McCabe), on trouve qu'il est inférieur à 30 dans la majorité des classes, donc on peut déduire que les méthodes sont facile à maintenir et à comprendre.
- Les classes ne dépend pas des autre classes dans d'autre packages (CA, CE)
- Le nombre des méthodes par classe est inférieur à 40, ce qui respecte les normes de qualité.
- Il n'y a pas des problèmes de l'héritage puisque la profondeur de la classe égale 1 dans l'arbre d'héritage.

1.3 Métriques, caractéristiques et interprétation par méthode

■ Résultat 3:

métriques et interprétation par méthodes										
liste des méthodes										
+ Ajouter										
Test	Nom de la Classe	type de méthode	Nom de méthode	SLOC	interpretation	V(G)	interpretation	date du test		
2	junit.framework.Assert	public static	assertTrue	5	 3	2	 3	2018-06-13		
2	junit.framework.Assert	public static	assertTrue	3	 3	1	 3	2018-06-13		
2	junit.framework.Assert	public static	assertFalse	3	 3	1	 3	2018-06-13		
2	junit.framework.Assert	public static	assertFalse	3	 3	1	 3	2018-06-13		
2	junit.framework.Assert	public static	fail	6	3	2	3	2018-06-13		

■ Interprétation des résultats :

On interprète l'image ci-dessus, toutes les méthodes sont marquées avec un symbole de soleil, c.-à-d que le nombre total de lignes de codes dans la méthode est respecté (inférieur à 5), ainsi que les chemins sont indépendant dans un module de travail.

2. Résultat de test sur la classe TestCase

- Résultat :

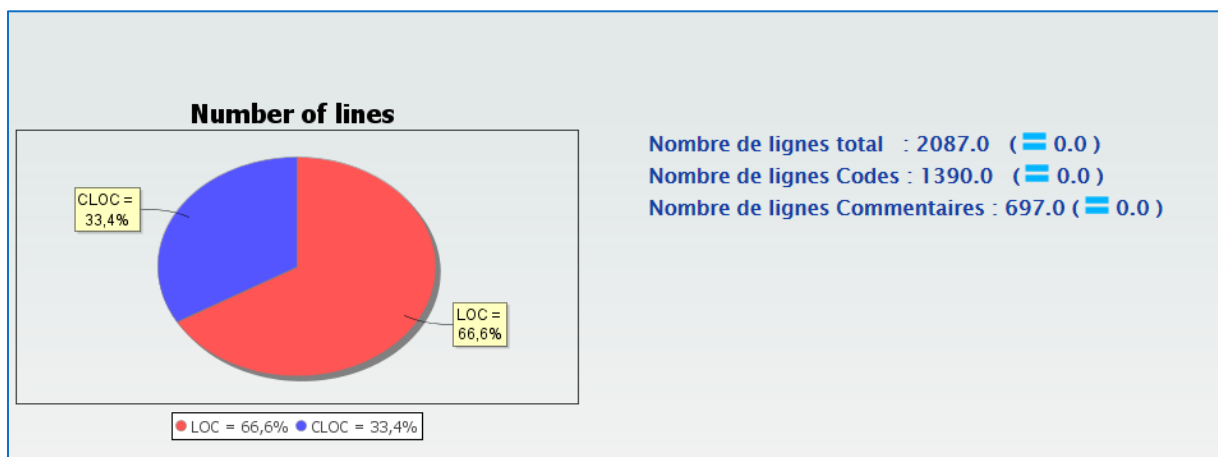
Résultats du test sur les métriques																		
MPF	WMC	cyclo	VG	NOC	LOC	DIT	CLOC	NOM	POF	MHF	AHF	RFC	APF	Abs	MPC	TCC	Inst	D
0.0784	0.0	61.0	1.0	0.0	192.0	1.0	263.0	51.0	0.0	0.0	1.0	107.0	0.0	1.0	107.0	1.0	0.0	0.0

La présente figure présente un ensemble des métriques calculée, de la classe TestCase.

3. Statistiques

3.1 Nombre des lignes

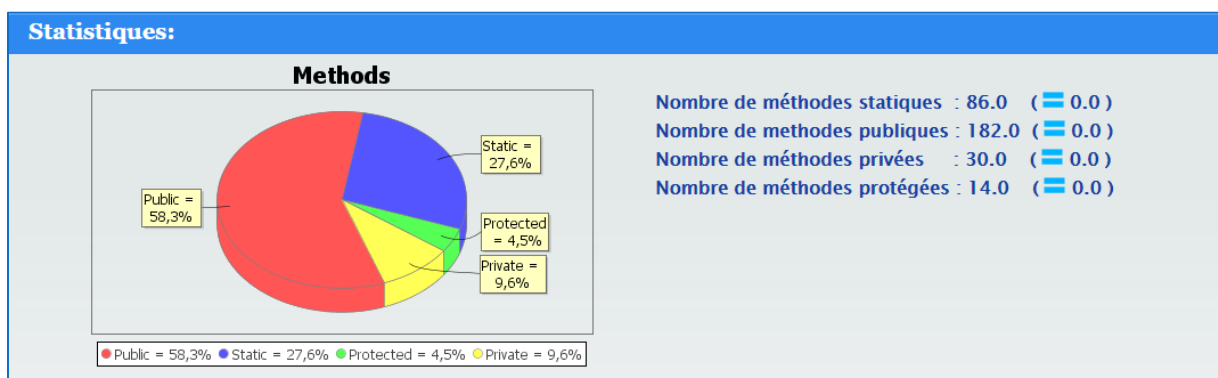
- Résultat 3:



- Interprétation des résultats :

EvalMetrics nous permet également de faire des statistiques sur le nombre des lignes du projet, en effet notre projet contient 33% de commentaires par rapport au code, cela respecte les normes de qualité.

3.2 Méthodes



■ Interprétation des résultats :

Le présent graphe nous donne des informations sur le pourcentage de différentes méthodes présentes dans notre projet, pour notre cas, plus que la moitié des méthodes qui constituent JUnit sont **public**, les méthodes **statiques** présentent un quart, et le reste sont des méthodes **protected** et **private**.

IV. ANALYSE DU PROJET JUNIT PAR SONAR

1. Introduction

Lorsqu'on se connecte sur la page d'accueil du serveur Sonar, et en accédant à notre projet, on visualise un tableau où sont affichées les principales informations résultantes des analyses de Sonar

	LINES OF CODE	DEBT	ISSUES	COVERAGE	DUPLICATIONS
My project	30k	59d	2.7k		2.0%
src/main/java/junit/extensions	134	47min	8		0.0%
src/main/java/junit/framework	933	6h	82		0.0%
src/main/java/junit/runner	264	4h	31		0.0%
src/main/java/junit/textui	221	4h	29		0.0%

Puisque Sonar nous garantit une consultation des statistiques pour :

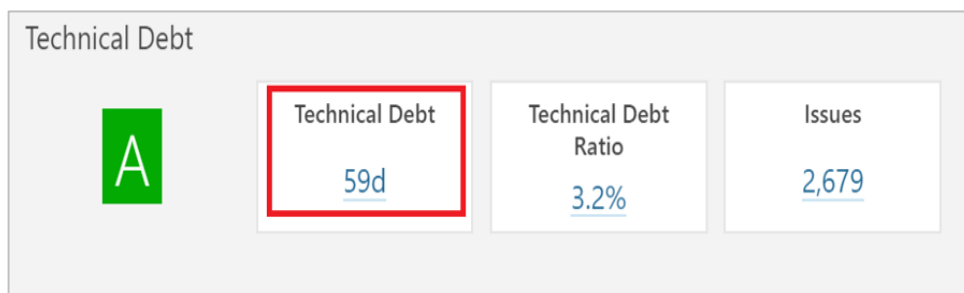
- **Tout le projet**
- **Un module du projet**
- **Un package Java**
- **Une classe Java**

On vous présente dans ce rapport des statistiques **globales** qui concerne **le projet entier**, et d'autres concernant **une classe** choisie du projet.

2. Analyse des résultats obtenus

2.1 Par projet :

■ Résultat 1 :



■ Description

L'image ci-dessus représente **une métrique de maintenabilité** nommée : « la dette technique ».

La dette technique est une dette que vous encourez à chaque fois que vous faites quelque chose de la mauvaise façon en laissant la qualité du code se détériorer. Tout comme les dettes financières, agir ainsi est plus facile sur le court terme. Mais au fil du temps, les "intérêts" que vous payez sur cette dette deviennent énormes, jusqu'à atteindre un point où une réécriture complète de l'application devient plus facile que de maintenir ou de modifier le code existant (Explications de **Martin Fowler** en anglais).

Cette **métrique** reflète donc l'**effort** pour résoudre tous **les problèmes de maintenabilité**.

Sa mesure est stockée en **minutes** dans la base de données, en considérant le jour est composé de 8 heures lorsque les valeurs sont affichées en jours.

- **Interprétation des résultats**

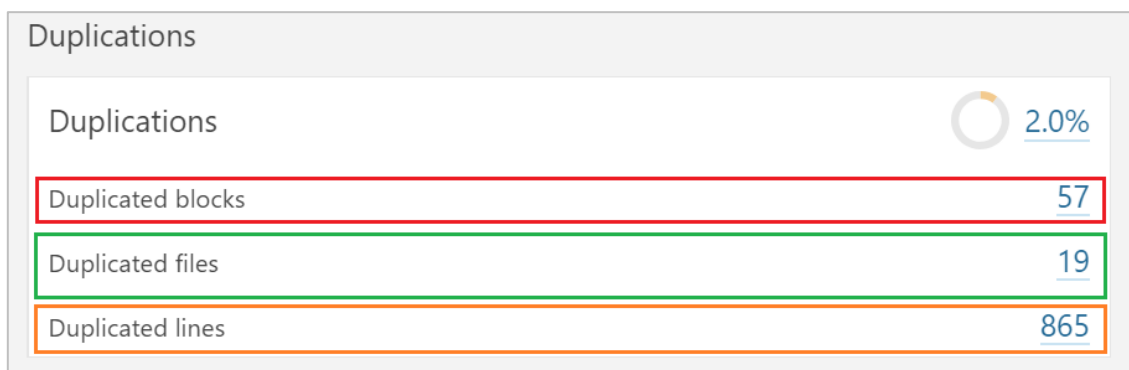
Le cas de notre projet demande **59 jours** pour résoudre les problèmes de **maintenabilité**, cette

Mesure est très couteuse en terme de temps (presque 2 mois).

Afin de **la diminuer** il faut contrôler un certain nombre de facteurs parmi lesquels on cite :

- Limitation des technologies qui entrent en jeu dans la construction de l'application.
- Les technologies sélectionnées devront avant tout être adaptées au besoin.
- Le code source doit suivre un modèle unique, adopté et appliqué systématiquement par l'ensemble des développeurs travaillant sur le projet.

- **Résultat 2 :**



■ Description :

La figure ci-dessus nous montre une **métrique de duplications** :

La duplication de code arrive à la suite de la **programmation par copier-coller**. C'est une erreur classique de débutants en programmation informatique. Cependant, cette erreur touche également les développeurs confirmés.

Le code dupliqué entraîne de graves problèmes de **maintenance** dont la gravité augmente avec la quantité de code dupliqué. Plus le code est dupliqué, plus il y a de code à **maintenir**. Par exemple :

- Si le code copié avait une erreur de programmation, alors tout le **code collé** dispose de la même erreur de programmation. **Corriger** l'erreur nécessite de **modifier tous les endroits** où le code est **dupliqué**.
- Si le code dupliqué doit **évoluer**, il faut alors modifier **tous les endroits** où le code est dupliqué pour y parvenir.
- Un code dupliqué peut **masquer des différences minimales**, mais essentielles, qui existent avec une autre portion de code similaire.

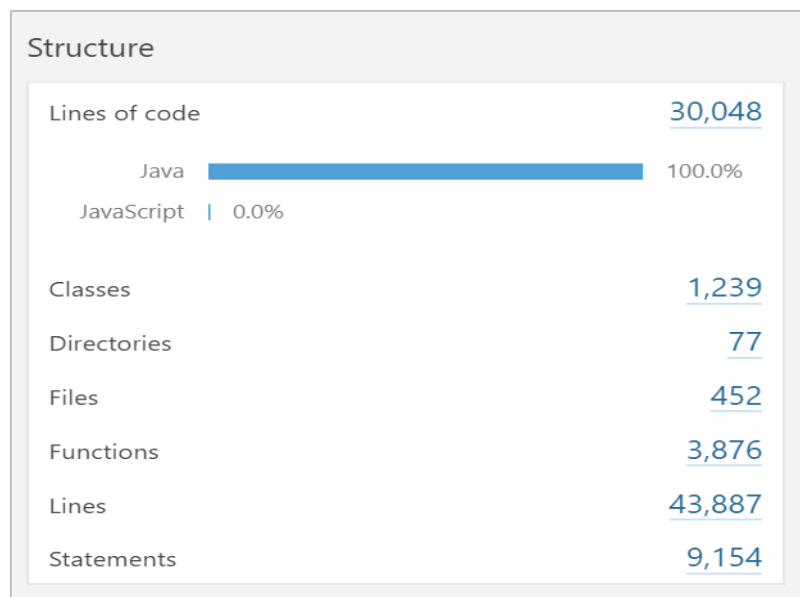
■ Interprétation des résultats

Cette mesure reflète **57 bloc dupliqué, 19 fichier dupliqué et 865 lignes dupliquées**.

Même si le pourcentage du code dupliqué dans ce cas n'est pas trop sévère **2%**.

Pour éviter la duplication du code il faut penser à : factoriser le code, faire des tests unitaires, ...

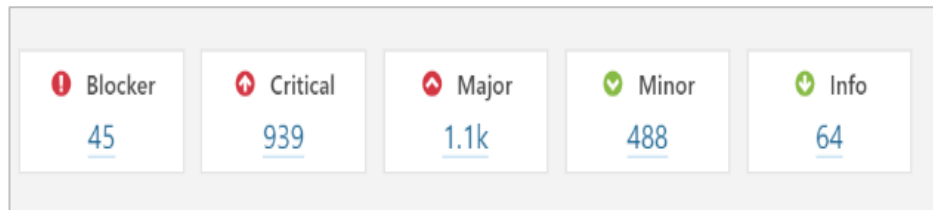
■ Résultat3



■ Description

Cette image la structure du projet mesuré, en montrant des statistiques principales, comme le nombre de lignes de code, de packages, de répertoires, de classes Java, etc.

▪ Résultat 4 :



Cette figure reflète les niveaux de sévérité des erreurs présentes dans notre projet : **45 problèmes bloquants, 939 critiques, 1100 problèmes majeurs et 488 problèmes mineurs.**

▪ Résultat 5 :



▪ Description :

Sonar nous permet ainsi d'avoir des blocs destinés à afficher les résultats de l'exécution des tests unitaires : couverture du code, pourcentage de réussite, etc.

2.2 Par classe :

▪ Résultat1 : (voir l'image dans la page suivante)

▪ Interprétation des métriques de lignes de code :

- La classe mesurée contient **70 lignes** de code :

Cette mesure correspond à la métrique traditionnelle de taille qui dit :

« *La longueur d'un fichier devrait contenir entre 4 et 400 lignes de programme* ».

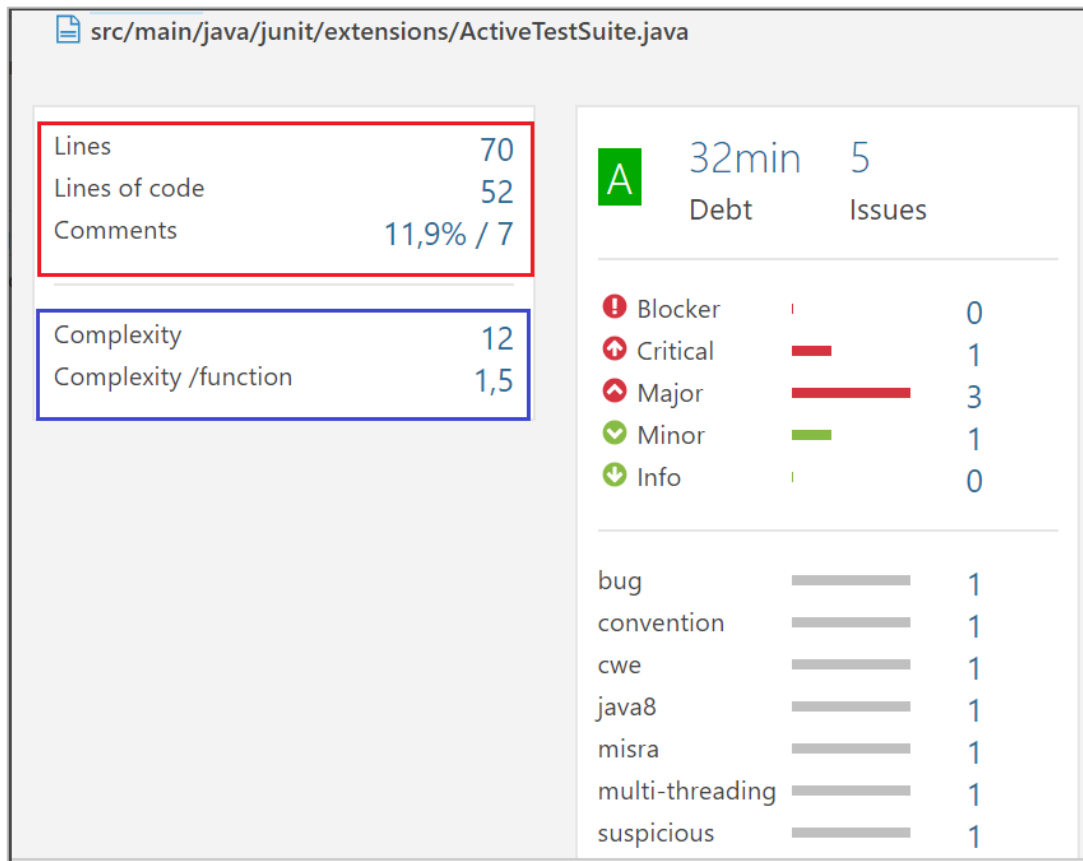
- Le pourcentage de commentaire dans cette classe est 12% ce qui est inférieur à 30% qui est la borne minimale nécessaire.

- Peu de commentaires reflètent le fait que le contenu de cette classe est très trivial
Sinon elle est pauvrement expliquée.

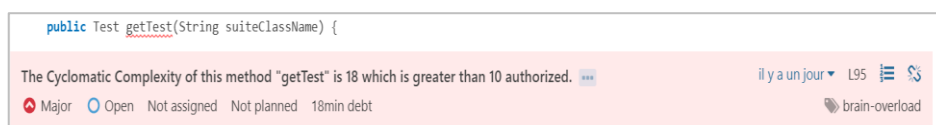
- **Interprétation des métriques de complexité :**

La complexité des fonctions de cette classe est 1.2.

Cette mesure est acceptable et n'est pas risqué car elle supérieure de la valeur minimale (qui est 1) de la complexité que doit avoir toute fonction.



- **Résultat 2 :**



- **Description et interprétation :**

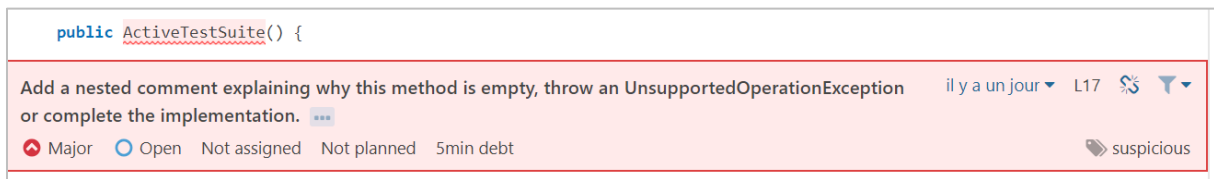
L'image ci-dessus traite la complexité cyclomatique de la classe mesurée.

Le résultat indique que la complexité de cette fonction dépasse la valeur permise qui est 10.

Cette violation de règle est considérée comme un problème **majeur** qui demande **18 min** d'effort en maintenabilité.

Pour diminuer cette complexité il faut augmenter le nombre de test sur cette fonction.

■ Résultat 3 :



Suite à une analyse de cette méthode **vide**.

L'outil Sonar indique que l'absence de l'implémentation est considérée comme **une erreur majeure**.

Pour la corriger il faut soit :

- Implémenter la méthode.
- Ajouter un commentaire expliquant pourquoi.

V. COMPARAISON DES DEUX OUTILS SONAR ET EVALMETRICS

En se basant sur **catégories des métriques** de chaque des outils et tenant compte **des résultats d'analyse générés** et donc les **métriques utilisées**, la comparaison entre Sonar et evalMetrics peut être résumée dans les points suivants :

- ✓ En prenant **les catégories des métriques adoptées** comme critère de comparaison, **Sonar** est plus orienté vers les **cas de test** en indiquant les blocs testés, les autres non testés, en annonçant les états couverts et non couverts ..., en termes **de métriques de code**, il se concentre sur **la taille** en mesurant La taille du **commentaire**, la taille du **fichier**, la taille du **paquet**, **class** ...
EvalMetrics couvre également toutes les catégories de métriques précédemment mentionnées **à l'exception de la catégorie de test**.
- ✓ Les deux outils utilisent **les plugins PMD** qui est un outil puissant pour détecter les défauts présents dans le code. Grace à cet outil d'analyse, Sonar et evalMetrics deviennent capable d'analyser le code source Java, rechercher **les bugs possibles**, **le code mort**, détecter le code dupliqué existant dans le programme et mesurer le nombre **de blocs**, **de lignes dupliqués**...
- ✓ Les deux outils sont aussi basés sur l'outil **Checkstyle** qui peut détecter jusqu'à 2228 problèmes liés aux **conventions de code Java** et aux **normes**. Son fonctionnement est basé sur **des règles de validation**, qui sont équivalentes dans la plupart des cas à des conventions de codage, de sorte que les **violations** de règles permettent de mesurer les violations des conventions de codage.
- ✓ Sonar est un outil très puissant pour faire une analyse **microscopique** sur un projet utilisant **un modèle de qualité**. Cela signifie que bien qu'il soit basé sur **ISO9126** que ce **modèle n'apparaît** pas dans le projet. L'outil se concentre principalement sur les métriques et

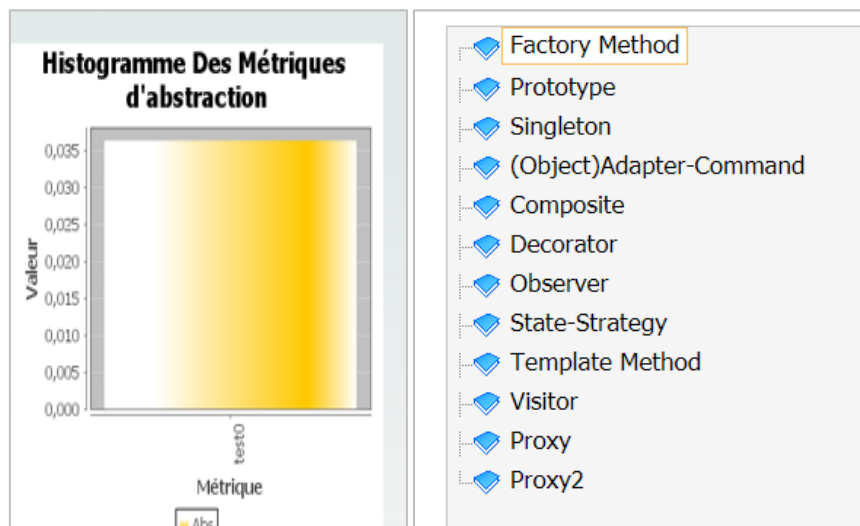
règles de codage et il les présente en général. Avec evalMetrics, une analyse à la fois **macroscopique** et **microscopique** est possible, en fait EvalMetrics est basé sur le modèle **ISO 25000** donc l'analyse du projet se fait suivant ce dernier, dans la mesure ou devient possible de voir les facteurs de qualité de notre modèle de projet ainsi que leur satisfaction, chose qui nous aide à décider et évaluer la qualité de l'ensemble du projet. En outre, evalMetrics donne aussi la possibilité **d'analyser chaque classe, méthode de classe** en montrant les métriques liées à chaque niveau.

VI. CONCLUSION

Ce rapport présente une étude comparative qui s'articule sur les deux outils de mesure de qualité **Sonar** et **evalMetrics**.

En se basant sur des critères fonctionnels présentés dans la section précédente, cette étude se résume à dire que **Sonar** est plus riche que **EvalMetrics** en ce qui concerne la présentation des tests effectués / non effectués ..., mais il montre sa faiblesse dans la mise en œuvre de la mesure, car il ne couvre pas toutes les propriétés de qualité et ne met pas en valeur le modèle sur lequel il s'est basé (absence de présentation du modèle dans les résultats).

EvalMetrics quant à lui, est similaire à Sonar en ce qui concerne les fonctionnalités qu'ils couvrent, même s'il ne couvre pas les tests, il met en œuvre de nombreuses mesures, il possède une partie dédiée à la détection des designs patterns et il est capable de tracer un histogramme de chaque facteur de qualité et de chaque mesure pour le premier test.



En conclusion de cette étude, on peut dire que Sonar est plus avantageux dans la présentation du test unitaire (Unité erreurs de test, temps de test, test effectué, état couvert ...) tandis que evalMetrics n'a pas. D'autre part evalMetrics dépasse Sonar en présentant le modèle qu'il met en œuvre.