ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

Mesures des sous-caractéristiques de la maintenabilité en génie logiciel

Rapport technique
Présenté à l'École de Technologie Supérieure dans le cadre du cours MGL804 - Réalisation et Maintenance de Logiciels

Été 2018

Prof. Alain April

Par,
ELWARTITI, Alkhalil
Code permanent: ELWA19129201

Montréal, le 16 juillet 2018

RÉSUMÉ

Mesures des sous-caractéristiques de la maintenabilité en génie logiciel

Le projet est un travail de recherche concernant les mesures de la maintenabilité en génie logiciel. Plus particulièrement, il s'agit de mesurer les sous-caractéristiques de l'attribut de qualité de maintenabilité tel que défini dans le standard ISO 25010 [3]. Il s'agit de documenter les mesures offertes par le standard ISO, ceci étant l'aspect théorique, puis d'étudier les mesures pouvant être effectuées au niveau pratique et ceci à l'aide d'outils logiciels.

Le standard ISO 25023 [7] est utilisé dans ce projet pour faire l'inventaire de mesures importantes pour chaque sous-caractéristique de la maintenabilité, et des outils populaires dans le monde du développement logiciel sont utilisés pour l'aspect pratique du projet, ces outils sont Sonarqube [4] et IntelliJ [11]. Afin de mieux exploiter les outils, les mesures sont faites sur un projet open-source écrit en Java.

Plusieurs mesures tant au niveau théorique que pratique sont compilées, et une comparaison est faite au final pour faire la liaison entre les deux. Les outils permettent de mesurer des attributs importants au niveau d'un logiciel en vue d'assurer sa qualité, et offrent une assistance au processus de développement en donnant un diagnostic du projet à chaque fois qu'un changement est apporté au logiciel.

Mots-clés :
Maintenance, Sonarqube, IntelliJ, mesure.

Table des matières

TABLES DES FIGURES

1. Introduction

Dans le cadre du cours de maintenance en génie logiciel MGL804 tenu à l'été 2018 à l'École de Technologie Supérieure, j'ai décidé de travailler sur l'aspect de la mesure de la qualité interne d'un logiciel mon projet de session. Ce sujet est très important, car il est un facteur déterminant au niveau de la gestion dans une entreprise qui applique un processus de maintenance mature.

En effet, lorsque les mesures de qualité sont disponibles, il est possible de préciser notre compréhension des problématiques au sujet de la qualité du code source et on lui procure une certaine tangibilité pour l'améliorer. À l'inverse, lorsqu'on ne peut pas mesurer la qualité interne d'un logiciel, nous en avons une connaissance limitée et la qualité en souffre.

D'un point de vue externe, au niveau d'une organisation, les gestionnaires sont souvent confrontés à ce genre de situation, où il n'y a pas de mesures à présenter pour appuyer le processus de maintenance. Conséquemment, on accorde moins d'importance qu'il faut pour mener à bien cette activité. La maintenance occupe la grande majorité du cycle de vie d'un logiciel. Robert L. Glass [2] estime ainsi que la maintenance occupe 40% à 80% du coût total d'un logiciel, donc on peut voir à quel point c'est important de pouvoir avoir un processus de gestion de la maintenance, qui soit mature.

Dans le livre de référence, Alain April et Alain Abran [1] présentent les trois problématiques qui représentent la vue externe de l'utilisateur lors de l'exécution du processus de maintenance:
- Le coût élevé.
- La lenteur du service.
- Le flou sur les activités faites en maintenance.

Pour adresser ces problématiques, la mesure est un outil incontournable, qui peut offrir des informations utiles au sein d'un département de maintenance.

Ce rapport est structuré de la manière suivante. Premièrement, quelques mesures sont présentées ici, qui permettent d'évaluer les sous-caractéristiques de l'attribut de qualité de maintenabilité, tel que décrit par le modèle de qualité défini dans la norme ISO 25010 [3]. Par la suite un inventaire des types de mesures qui sont offertes par des outils d'analyse de la qualité du code source, notamment SonarQube [4] est présenté. Finalement, une comparaison entre ce qui est présenté dans la littérature et ce qui peut être fait en pratique dans le cadre de la maintenance est discuté.

2. L’activité de la maintenance

Le standard ISO qui traite exclusivement la maintenance logicielle est le standard ISO 14764 [5]. Cette norme définit la maintenance comme étant les activités requises afin de fournir un support à un système informatique.

Lehman [12] a décrit, en 1969, la maintenance en termes d’évolution du système en 1969, et ses recherches au fil des années se sont conclus par huit lois qui se résument en : la maintenance est un processus évolutionnaire et les logiciels deviennent de plus en plus complexes au fur et à mesure qu’on y fait des changements ou qu’on y ajoute des fonctionnalités, stipulant qu’une bonne maintenance nécessite de contrôler et de réduire cette complexité.

À un niveau de détails encore plus élevé, Lienz et Swanson [6] ont défini quatre catégories qui caractérisent les activités de maintenance :
- **Adaptive:** il s’agit de l’ajout de nouvelles fonctionnalités au système, souvent à la demande du client.
- **Corrective:** c’est lorsqu’il y a une panne au niveau du système, cette catégorie est la plus prioritaire en maintenance.
- **Perfective:** c’est des améliorations faites au système, qui permettent d’augmenter la maintenabilité du système.
- **Preventive:** prendre en charge les défauts dits dormants, qui peuvent se développer en pannes dans le futur.

Plusieurs recherches ont conclu que c’est les activités de types adaptive et perfective qui représentent la grande majorité des coûts de la maintenance, et que les corrections ne représentent qu’une petite partie. Cela veut dire que ces types de maintenance permettent de garder le produit compétitif, en ajoutant toujours de nouvelles fonctionnalités et en faisant des améliorations.

3. Modèle de qualité – Maintenabilité

Pour faire de la mesure de la qualité du code source, durant le processus de maintenance, il est nécessaire d’identifier les caractéristiques qualité du logiciel à mesurer. Ces caractéristiques doivent faire ressortir l’état du logiciel afin qu’on puisse en tirer des informations pertinentes, permettant par la suite d’envisager des actions appropriées à effectuer. Le but est de mettre en place une stratégie de gestion de la qualité du code source.

Pour qualifier la maintenabilité d’un logiciel, la norme ISO 25010 [3] est utilisée. Ce standard définit la maintenabilité comme étant l'efficacité avec laquelle on peut effectuer des modifications sur un logiciel.

La maintenabilité est composée des sous-caractéristiques suivantes :

- Modularité;
- Réutilisabilité;
- Analysabilité;
- Modifiabilité;
- Testabilité.

La figure suivante présente le modèle de qualité issu du standard 25010.

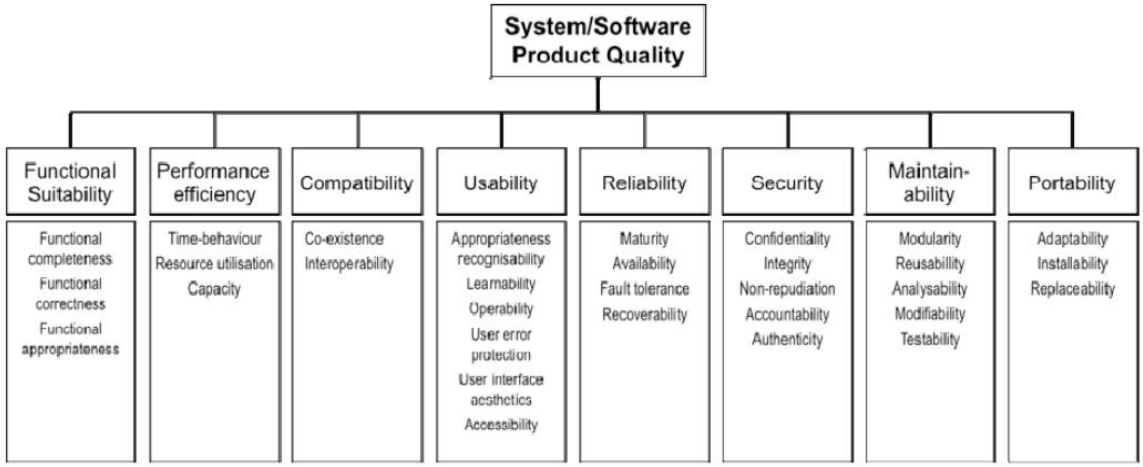

Figure 1. Modèle de qualité (ISO 25010 [3])

Il est important de comprendre ces sous-caractéristiques, afin d'en faire la mesure et arriver à choisir les mesures qui sont les plus appropriées et à partir desquelles on peut tirer des conclusions sur notre système. On en fait la définition ici, suivant le standard ISO 25010 [3].

- **Modularité:** La modularité est basée sur des composants. Une bonne modularité implique que les changements au niveau d'un composant ont un impact minimal sur les autres. Une structure modulaire applique le principe de la séparation des préoccupations en minimisant les interdépendances entre les composants;
- **Réutilisabilité:** niveau/facilité avec laquelle on peut réutiliser un composant dans un logiciel;
- **Analysabilité:** niveau avec lequel on peut faire de l'analyse d'impacts sur un logiciel lors d'un changement, ou bien lorsqu'on essaye de détecter les causes d'un défaut;
- **Modifiabilité:** facilité avec laquelle un logiciel peut subir un changement sans dégrader sa qualité et introduire des défauts;
- **Testabilité:** Ce concept représente la facilité avec laquelle un artéfact d'un logiciel peut être testé. Plus cette caractéristique est élevée pour un système, plus il est simple de trouver des défauts à l'aide de diverses activités de test.

4. La mesure des sous-caractéristiques de la maintenabilité

Si l'on veut évaluer les sous-caractéristiques de la maintenabilité présentées précédemment, il nous faut des mesures pertinentes et applicables dans un environnement réel. En général, les mesures sont très peu utilisées dans le domaine du logiciel, car c'est une activité récente.

Dans un premier temps, on énumère ici quelques mesures qui sont centrales pour évaluer la maintenabilité d'un logiciel. Ces mesures sont proposées dans le standard ISO 25023 [7] qui définit différents types de mesures en liaison avec les attributs de qualité qu'on peut voir sur la *figure 1*.

4.1. Modularité

La principale mesure pour cette caractéristique est le **couplage** entre les différents composants du système. Il est défini par :
X = A / B
A : Nombre de composants qui ne sont pas affectés directement par des changements effectués sur d'autres composants.
B : Nombre total de composants.

4.2. Réutilisabilité

Deux mesures sont définies pour cette caractéristique, la première est le **ratio de réutilisabilité** qui décrit combien d'éléments du système sont réutilisables:
X = A / B
A : Nombre total d'éléments réutilisables.
B : Nombre total d'éléments du système.

La seconde est la **conformance aux standards de programmation,** définie comme suit:
X = A / B
A : Nombre de modules conformes aux standards.
B : Nombre total des modules logiciels.

4.3. Analysabilité

On considère ici deux mesures, la **capacité de traçage** qui est la facilité avec laquelle on peut identifier l'opération spécifique qui a causé la panne, sa définition est la suivante:
X = A / B
A : Nombre d'objets opérationnels qui peuvent être enregistrés par le système durant l'opération.
B : Nombre d'objets opérationnels qui doivent être enregistrés par le système durant l'opération.

La deuxième mesure concerne la **suffisance des fonctions de diagnostics**, qui exprime le degré d'efficacité de ces fonctions. La définition est la suivante:
X = A / B
A : Nombre de fonctions de diagnostics implémentées.
B : Nombre de fonctions de diagnostics requises dans la spécification.

4.4. Modifiabilité

Cette sous-caractéristique comporte des mesures importantes, on cite premièrement l**'efficacité de modification** qui est la facilité avec laquelle un mainteneur peut apporter des modifications au système pour satisfaire des exigences, elle est définie comme suit:
X = A / B
A : Temps total de travail passé pour faire les modifications.
B : Nombre total de modifications.

Ensuite on a le **taux de réussite de la modification**, qui est le degré avec lequel on peut modifier un système sans qu'il y ait de pannes après.
X = 1 - B / A
A : Nombre de problèmes avant la modification dans un intervalle de temps.
B : Nombre de problèmes après la modification dans le même intervalle.

4.5. Testabilité

Le standard définit une mesure qui est intéressante pour la testabilité, qui est **la complétude fonctionnelle des fonctions de tests**, cette mesure est définie comme suit:
X = A / B
A : Nombre de fonctions de tests implémentées.
B : Nombre de fonctions de tests requises.

5. Mesurer la maintenabilité avec des outils

Dans la section précédente, des mesures de la maintenabilité d'un logiciel ont été présentées. Ces mesures sont tirées du standard ISO 25023 [7]. Mais qu'en est-il des mesures de la qualité logiciel pratiquement? Que permettent les outils d'analyse de la qualité du code source?

5.1. Outils considérés

Afin d'expérimenter ce qui se fait en termes de mesure de la qualité et d'analyse de code source pratiquement, deux outils populaires du domaine ont été choisis :
1) l'environnement de développement IntelliJ [11] et 2) l'outil d'inspection continue et d'analyse, SonarQube [4].

5.1.1. IntelliJ

Est un logiciel d'environnement de développement intégré, développé par *JetBrains.* Cet outil offre plusieurs fonctionnalités et plug-ins qui permettent de faire de l'analyse du code source. Il permet de produire quelques mesures de la qualité du code source intéressantes.
Il permet notamment d'analyser plusieurs aspects du code qu'on pourrait catégoriser de mauvais pratiques de programmation (c.-à-d. des "code smells"). Il permet aussi de faire une analyse des dépendances entre les composants du logiciel et effectuer une analyse du flux de données, dont l'objectif est de mieux comprendre le fonctionnement de parties du code qui sont complexes.

5.1.2. SonarQube

Outil très polyvalent pouvant être intégré dans plusieurs environnements. Il fournit l'analyse automatique pour plusieurs langages de programmation (c.-à-d. plus de 20 langages). Il permet aussi la détection des mauvaises pratiques de programmation (« code smells », des vulnérabilités et les défauts. SonarQube utilise des règles qui permettent de configurer l'analyse, et aussi un concept de profil de qualité pour le projet.

5.2. Quelques mesures disponibles dans ces outils

Afin d'avoir une meilleure idée de la nature des mesures que l'on peut retrouver en pratique, les outils présentés précédemment ont été appliqués sur un projet en Java. Ce projet s'appelle *pyramid* [8]. Pyramid est un cadriciel pour l'apprentissage machine écrit en langage Java. Il permet de faire, entre autres, de la régression, du clustering et de la classification.

Configuration de SonarQube avec Maven:

Le projet pyramid utilise Maven, et SonarQube possède une fonctionnalité pour utiliser les fichiers Maven. Il suffit simplement d'ajouter le profil de SonarQube dans le fichier *setting.xml* situé dans le dossier ./m2, et d'ajouter le « plug-in » correspondant dans le fichier *pom.xml* du projet. Par la suite, il faut s'assurer que la base de donnée et SonarQube sont démarrés. Pour notre étude de cas, la base de données *Postgresql* est utilisée.

Il ne reste plus qu'à lancer l'analyse du projet avec sonar, à l'aide de la commande : *mvn sonar:sonar*

Ceci étant fait, on peut consulter les résultats sur l'adresse *http://localhost:9000/projects* , suivant l'adresse du serveur spécifiée lors de l'installation de sonar.

La figure suivante montre l'interface d'accueil de SonarQube, on peut y voir listé les projets analysés et un "bilan" du projet en termes de concepts de qualité.

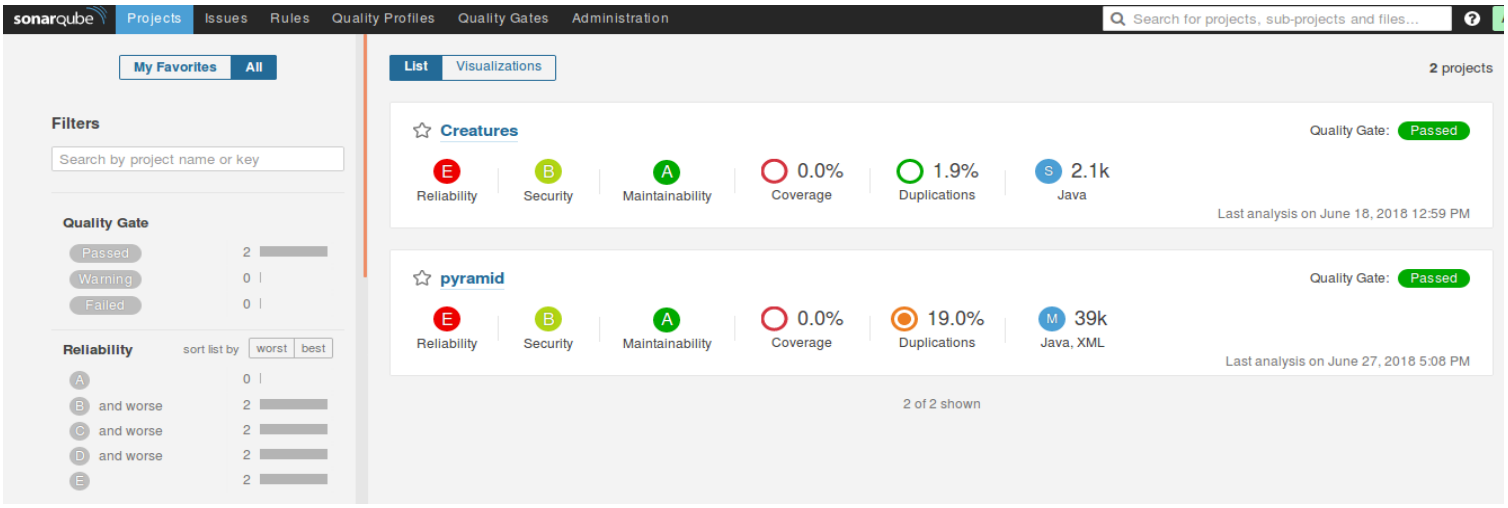

Figure 2. Interface de SonarQube

Les mesures disponibles sur SonarQube:

SonarQube offre sept catégories de mesures. Ces catégories sont présentées dans le tableau suivant, les termes sont laissés dans la langue d'origine (c.-à-d. l'Anglais):

Table 1. Mesures offertes par SonarQube

Reliability	• Reliability Remediation Effort • Reliability Remediation Effort on New Code • Reliability Rating on New Code
Security	• Security Remediation Effort • Security Remediation Effort on New Code • Security Rating on New Code
Maintainability	• Technical Debt • Added Technical Debt • Technical Debt Ratio • Technical Debt Ratio on New Code • Effort to Reach Maintainability Rating A • Maintainability Rating on New Code

Coverage	• Line Coverage • Uncovered Lines • Uncovered Lines on New Code • Uncovered Conditions on New Code • Lines to Cover on New Code • Lines to Cover
Duplications	• Duplicated Blocks • Duplicated Blocks on New Code • Duplicated Lines • Duplicated Lines on New Code • Duplicated Files
Size	• Lines • Lines on New Code • Statements • Functions • Classes • Files • Directories • Comment Lines • Comments (%)
Complexity	• Complexity / Function • Complexity / File • Complexity / Class • Cognitive Complexity

Comme on peut le voir au tableau précédent, il y a plusieurs mesures disponibles. Certaines d'entre elles, notamment la complexité, maintenabilité, fiabilité et la sécurité sont très utiles.

Calibration de l'outil SonarQube

On peut faire des calibrations au niveau de l'outil sur plusieurs aspects du logiciel. On présente ici des exemples de configurations de SonarQube qui sont intéressantes du point de vue du processus de développement logiciel.

L'une des calibrations est la « quality gate ». Elle permet de définir une politique de qualité au sein de l'organisation en établissant un standard pour le logiciel en développement. Ce standard se présente sous la forme de conditions à remplir par le logiciel lorsqu'il est analysé. SonarQube permet de définir plusieurs « quality gates » à la fois, et choisir quelle « gate » est assignée à quel logiciel.

Cette flexibilité représente bien la réalité étant donné qu'on a des exigences différentes suivant la nature du logiciel.

Pour illustrer le concept de « quality gate », on donne un exemple sur le projet étudié pyramid.

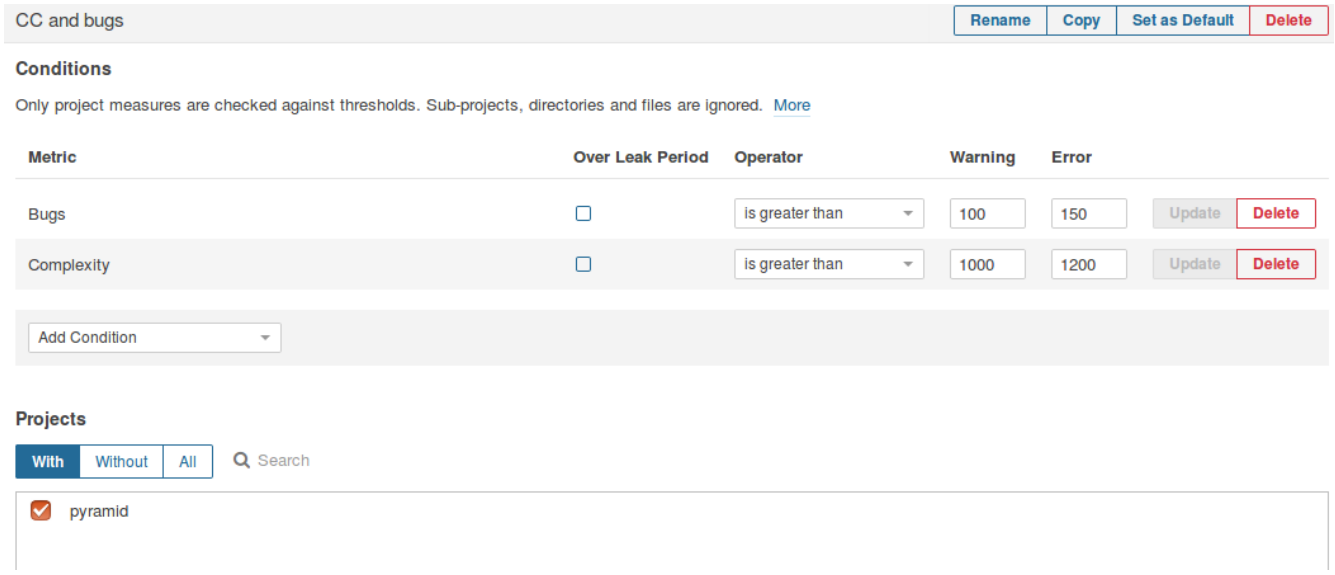

Figure 3. Quality gate configurée pour le projet pyramid

La *figure 3* montre un exemple de « quality gate » pour le logiciel pyramid. On peut y définir la mesure qu'on désire évaluer pour le logiciel et on y ajoute une condition booléenne. Dans l'exemple, on définit deux mesures qui stipulent que le nombre de défauts ne devrait pas dépasser le seuil de 150, et la complexité du projet devrait rester en dessous de 1200. À la fin, il faut lier cette « quality gate » au projet désiré. À partir de là, lorsqu'on lance une analyse sur le projet avec SonarQube, le logiciel sera testé contre cette « quality gate », et ce test représente la condition de passage en production du logiciel.

La figure suivante montre le résultat de l'analyse après avoir configuré la « quality gate ».

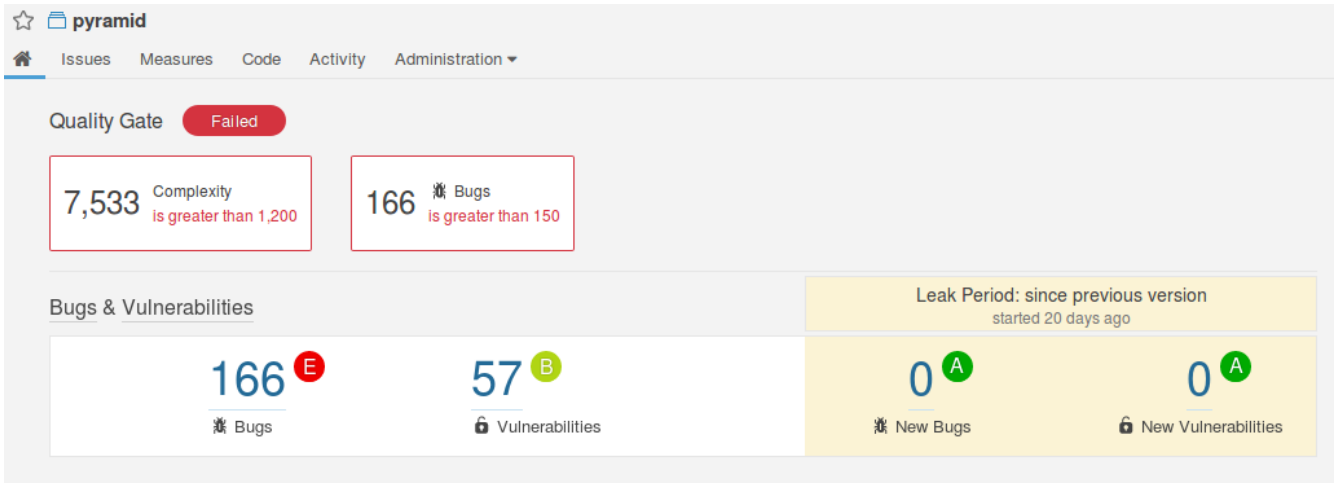

Figure 4. Test du logiciel pyramid contre la "quality gate"

Comme on peut le voir sur la *figure 4*, le résultat est « failed ». Le nombre de défauts et la complexité du logiciel dépassent les seuils qui représentent le standard que doit respecter le logiciel.

Un autre moyen de faire de la calibration sur SonarQube est de définir des profils de qualité (« Quality profile »). Le processus est similaire à celui de « quality gate », sauf qu'ici c'est les standards pour les règles.

SonarQube se base sur un concept de règles, c'est très important, car ça permet de configurer suivant ce qu'on veut analyser dans notre projet. Ça représente ce qui doit être considéré comme défaut, mauvaise pratique de programmation ou bien une vulnérabilité.

La figure suivante illustre cela.

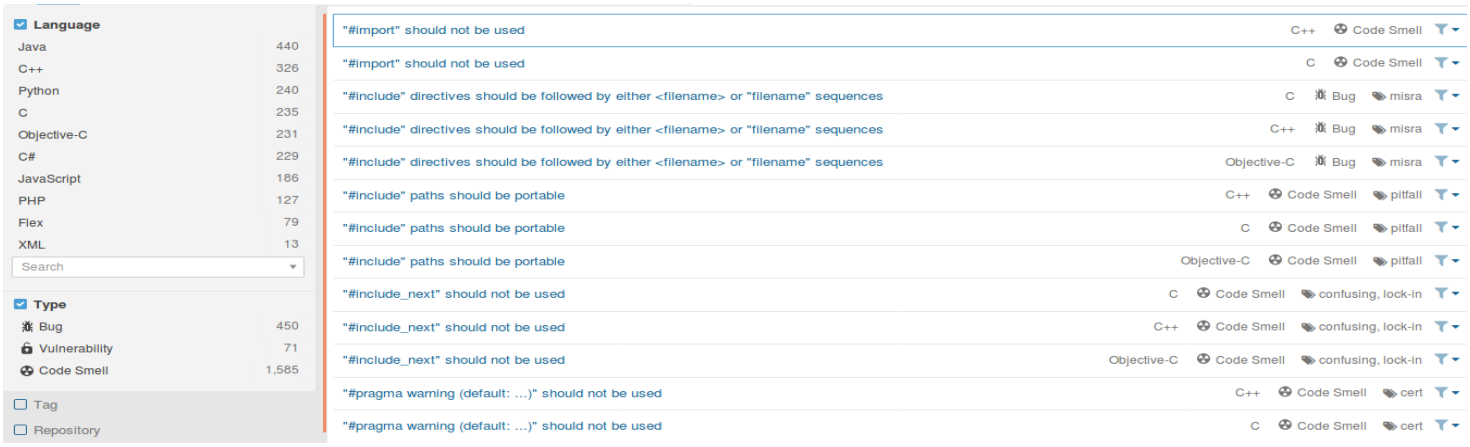

Figure 5. Règles d'analyse de SonarQube

On peut voir à la *figure 5* les différents types de règles, et les différents langages pour lesquels il y a des règles disponibles. On peut bien évidemment ajouter d'autres langages en installant de nouveaux "plug-ins" correspondants.

Les règles disponibles sur SonarQube sont définies par défaut dans un profil appelé « Sonar way ». Évidemment on peut créer notre propre profil, et c'est recommandé, car « Sonar way » n'est pas configurable. La figure suivante illustre ça.

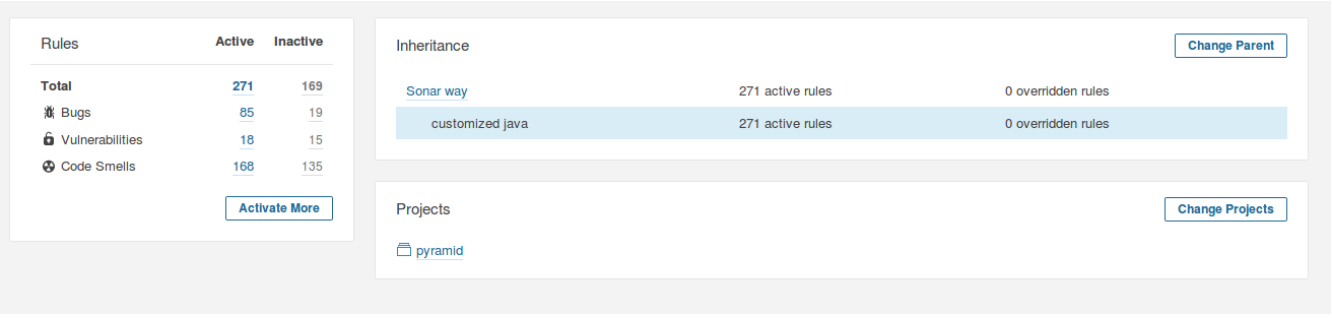

Figure 6. Profil de qualité sur SonarQube

SonarQube permet de créer facilement un profil à l'aide d'une fonctionnalité intéressante. On peut faire de l'héritage au niveau des règles. Comme on peut le voir sur la *figure 6* le profil nouvellement crée hérite des règles de « Sonar way » en ce qui concerne le langage Java. Par la suite, on peut configurer ces règles suivant les besoins du logiciel en développement.
Comme pour les « quality gate », il faut lier le profil à un projet en particulier.

Analyse à l'aide de SonarQube

Les trois attributs de qualité majeurs sont la fiabilité, la sécurité et la maintenabilité. Ces trois caractéristiques sont représentées sur SonarQube principalement par le nombre de défauts, le nombre de vulnérabilités ainsi que les code smells, respectivement. Aussi, pour chacune de ces trois mesures, l'effort pour remédier à ces problèmes est aussi estimé.

À titre d'exemple, on montre ici le résultat d'analyse pour la maintenabilité.

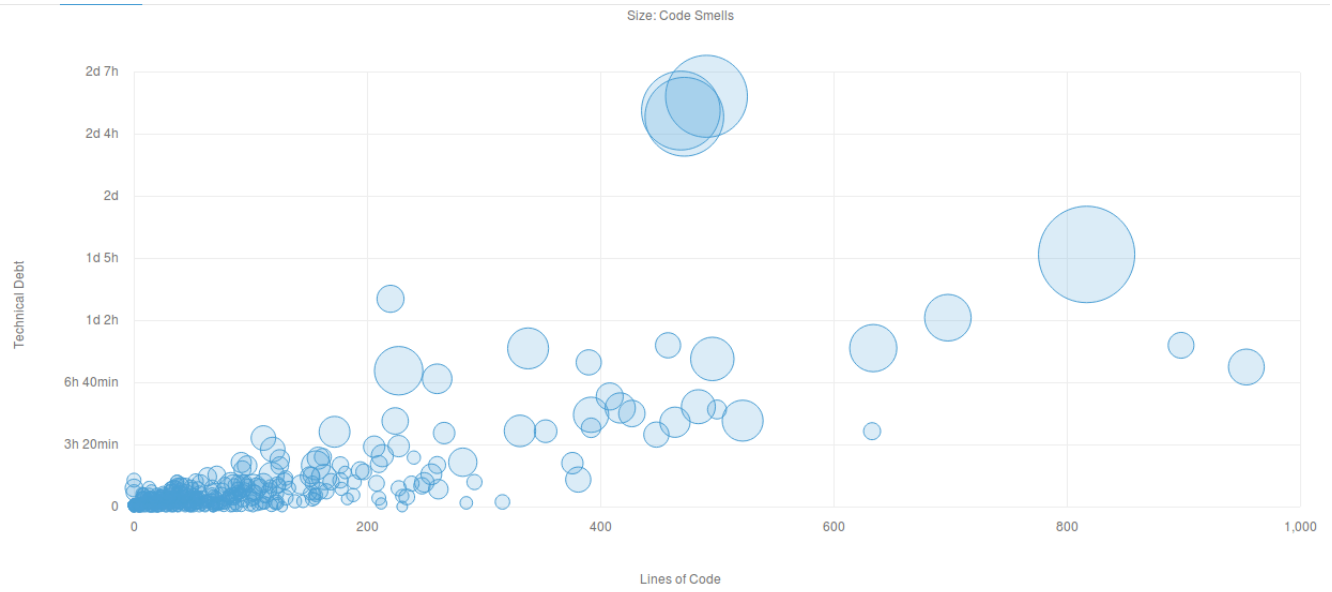

Figure 7. Analyse de la maintenabilité pour le projet pyramid (SonarQube)

Sur la *figure 7*, chaque cercle correspond à une classe logiciel du projet. Une classe comportant un nombre élevé de mauvaises pratiques (« code smells ») a un plus grand diamètre. On voit sur la figure ci-dessus qu'il y a trois classes dans le projet qui ont une dette technique assez élevée par rapport au reste.

Pour ce qui est des problèmes analysés, qu'ils soient des défauts, smells ou bien des vulnérabilités, ils sont catégorisés suivant leur sévérité jugée. Elles sont définies dans la documentation [9] comme suit:

Sévérité

- Bloquant;
- Critique;

- Majeur;
- Mineur;
- Info.

Dans la figure suivante, on peut voir les problèmes qui ont été relevés par SonarQube au niveau du projet. La grande majorité des problèmes sont de type “Majeur”.

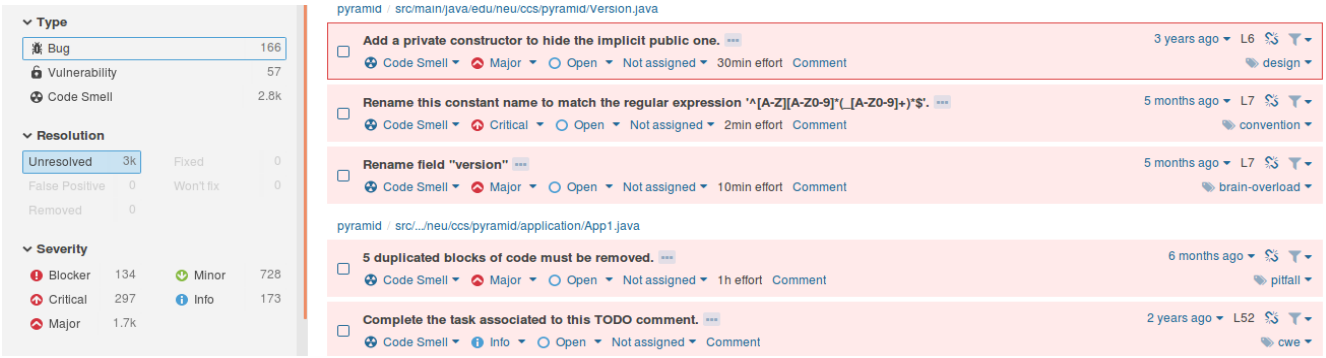

Figure 8. Problèmes dans SonarQube

La complexité

Une autre mesure importante en maintenance est la complexité. La complexité est définie dans [9] et représente le nombre de chemins qu’on peut trouver dans un module. SonarQube utilise la mesure de la complexité cyclomatique de McCabe [10].
Cette mesure est calculée d’une manière légèrement différente en passant d’un langage à un autre à cause de la différence dans les aspects techniques du langage, tels que ses mot-clés.
Pour ce qui est du langage Java, les mot-clés suivants augmentent la complexité : *if, for, while, case, catch, throw, return(qui n’est pas le dernier de la fonction), &&, ||, ?* [9].

La figure suivante montre les dix classes les plus complexes du logiciel étudié pyramid. Comme on peut le voir, beaucoup de classes représentent une complexité très élevée, et sont potentiellement des classes difficilement maintenables. Ceci peut être problématique au niveau de la maintenance de ce logiciel. Ceci devrait nous inciter à revoir le code de ces modules et tenter de les améliorer.

src/main/java/edu/neu/ccs/pyramid/dataset/DataSetUtil.java	188
src/main/java/edu/neu/ccs/pyramid/elasticsearch/ESIndex.java	130
src/main/java/edu/neu/ccs/pyramid/application/App1.java	130
src/main/java/edu/neu/ccs/pyramid/multilabel_classification/crf/CMLCRFElasticNet.java	117
src/main/java/edu/neu/ccs/pyramid/multilabel_classification/cbm/CBMNoiseOptimizerFixed.java	115
src/main/java/edu/neu/ccs/pyramid/multilabel_classification/crf/CRFF1Loss.java	112
src/main/java/edu/neu/ccs/pyramid/application/App2.java	112
src/main/java/edu/neu/ccs/pyramid/dataset/LabelGraph.java	110
src/main/java/edu/neu/ccs/pyramid/multilabel_classification/cbm/CBMUtilityOptimizer.java	110
src/main/java/edu/neu/ccs/pyramid/multilabel_classification/cbm/CBMOptimizer.java	110

Figure 9. Les dix classes les plus complexes dans pyramid

L'analyse des dépendances avec IntelliJ

Le second outil expérimenté pour mesure la maintenabilité utilise un « plug-in » disponible sur l'environnement de développement IntelliJ. Cet outil s'appelle **DSM Analysis** et permet de générer une matrice des dépendances entre les différents modules et classes du logiciel étudié. Cette représentation permet surtout d'évaluer l'effet d'un changement au logiciel. Si on devait apporter des modifications à une classe par exemple, on pourrait voir jusqu'où le changement va se propager (c.-à-d. Identifier les autres composants qui seront potentiellement impactés).

La *figure 10* présente la matrice générée à l'aide de l'outil DSM Analysis. Cette figure montre toutes les dépendances entre les différents modules de l'application, ainsi que les dépendances cycliques que l'on peut voir en rouge. On note que l'ellipsis dans la matrice se traduit en un nombre supérieur à 99 de dépendances.

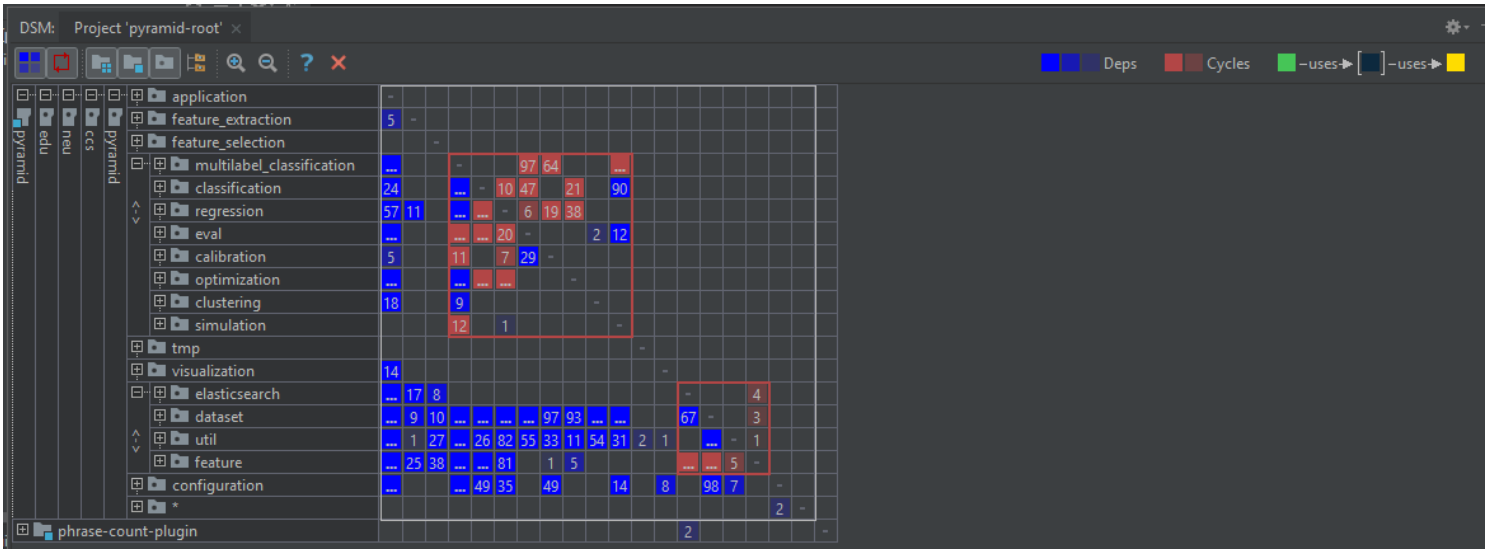

Figure 10. Dependence Structure Matrix du projet pyramid

Les dépendances cycliques sont les plus problématiques, car elles augmentent le couplage entre les modules et rendent impossible la réutilisation d'un module.

Le tableau suivant explique la *figure 10,* et montre le nombre de dépendances entre les différents modules du projet.

Table 2. Les dépendances entre les différents modules du projet pyramid

Module dépendant A	**Module dont dépent A**	**Nombre de dépendances**
multilabel_classification	eval	+99
multilabel_classification	calibration	11
multilabel_classification	simulation	12
classification	regression	+99
classification	eval	+99
classification	optimization	+99
regression	classification	10
regression	eval	20
regression	calibration	7
regression	optimization	+99
eval	multilabel_classification	97
eval	classification	47
eval	regression	6
calibration	multilabel_classification	64
calibration	regression	19
optimization	classification	21
optimization	regression	38
simulation	multilabel_classification	+99
elasticsearch	feature	+99
dataset	feature	+99
util	feature	5
feature	elasticsearch	4
feature	dataset	3
feature	util	1

6. Comparaison entre les mesures du standard et des outils

Nous avons vu beaucoup de mesures de la qualité dans ce rapport et ce, tant au niveau théorique, qu'au niveau pratique. Il reste maintenant à faire une synthèse de l'expérience effectuée.

La première observation qu'on pourrait faire est qu'il y a une similitude entre la sous-caractéristique de modifiabilité et les mesures de maintenabilité, fiabilité et sécurité de SonarQube. En effet, l'**efficacité de modification** ainsi que le **taux de réussite de modification**, peuvent être facilement mesurés pratiquement. SonarQube propose une estimation du nombre de problèmes dans le logiciel. Ceci est utile, car il est possible, après une période de corrections de défauts, de relancer une analyse sur le logiciel et voir combien de défauts on a pu résoudre.

La deuxième observation est qu'il y a une similitude entre la sous-caractéristique de testabilité et la complexité évaluée à l'aide de SonarQube. Si l'on considère **la complétude fonctionnelle des fonctions de tests,** elle est corrélée avec la complexité cyclomatique, car plus une fonction est complexe plus le nombre de chemins dans celle-ci est grand, ce qui rend le nombre de fonctions de tests requises plus élevé.

Le troisième lien est établi entre la **DSM** et la sous-caractéristique de modularité. Ce lien est évident, car la matrice dévoile toutes les dépendances entre les modules et classes et montre donc le **couplage** entre ces derniers.

Le quatrième lien est entre la **DSM** et la sous-caractéristique réutilisabilité. Comme il a été dit, la matrice de dépendances montre aussi les dépendances cycliques, et on sait qu'une dépendance cyclique rend la réutilisabilité d'un module impossible, car il ne peut être utilisé seul. On peut calculer le **ratio de réutilisabilité** à l'aide de la matrice.

Le tableau suivant est un récapitulatif des différents liens entre les mesures au niveau théorique et ceux au niveau pratique.

Table 3. Bilan des liens entre mesures théoriques et pratiques.

Mesure théorique	Mesure pratique
Modifiabilité	Nombre de smells, défauts et vulnérabilité (SonarQube)
Testabilité	Complexité cyclomatique (SonarQube)
Modularité	Dependance Structure Matrix
Réutilisabilité	Dependance Structure Matrix

Conclusion

Ce projet de session a offert une occasion d'expérimenter l'aspect de la mesure de la qualité interne du logiciel dans le processus de maintenance. On a vu en classe que l'activité de maintenance souffre d'un problème de gestion, et que les mesures peuvent contribuer à résoudre ce problème.

Dans ce projet il a été possible de faire un inventaire des mesures théoriques les plus importantes, à l'aide des standards ISO 25010 et 25023. Des mesures des sous-caractéristiques de la maintenabilité ont été explorées. Ensuite, les différentes mesures disponibles, à l'aide d'outils tels que SonarQube et DSM Analysis de IntelliJ ont été produites. Dans ces outils on a pu voir qu'il était possible d'analyser les problèmes reliés au code source au niveau d'un logiciel, notamment les code smells, les défauts et les vulnérabilités, et qu'il était facile d'analyser la complexité du code source. Une autre mesure intéressante vise l'analyse des dépendances cycliques entre les modules et classes, qui est très utile pour évaluer le couplage.

Ce genre de travail d'analyse est très intéressant, car il donne une bonne idée de ce qui peut se faire en réalité, et permet d'identifier de bons outils afin de s'attaquer aux problèmes de qualité dont souffre le domaine du logiciel.

Références

[1] April, A. et Abra, A. 2016 Améliorer la maintenance du logiciel, 2e édition Loze-Dion éditeur, 348 p.

[2] Glass, Robert L. 2001. Frequently Forgotten Fundamental Facts about Software Engineering. IEEE Software, vol. 18, no. 3 , pp, 112-111. doi: https://doi.org/10.1109/MS.2001.922739.

[3] ISO/IEC 25010: 2011. System and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, International Organization for Standardization, 34p.

[4] Sonarqube. The leading product for continuous code quality. SonarSource S.A., Switzeland. Consulté le 24 juin 2018. https://www.sonarqube.org/.

[5] ISO/IEC 14764:2006. Software Engineering - Software Life Cycle Processes –Maintenance. International Organization for Standardization, 44.

[6] Lientz, B. P, and Swanson, E. B. 1980. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2014p.

[7] ISO/IEC 25023:2016, System and software engineering – System and Software Quality Requirements and Evaluation (SQuaRE) – Measurement of system and software product quality, International Organization for Standardization, 45p.

[8] Cheng, L. 2014 Pyramid: Open Source Machine Learning Library written in Java. Java. Consulté le 27 juin 2018 : https://github.com/cheng-li/pyramid.

[9] Sonarqube. Metric Definitions. SonarSource S.A., Switzerland. Consulté le 27 juin 2018. https://docs.sonarqube.org/display/SONAR/Metric+Definitions.

[10] McCabe, T.J. 1976. "A Complexity Measure," IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308-320.doi: 10.1109/TSE.1976.233837.

[11] IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains. (n.d.). Consulté le 16 juillet 2018: https://www.jetbrains.com/idea/.

[12] *Lehman, M. M. 1980. "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle". Journal of Systems and Software.* ***1****: 213–221. doi:10.1016/0164-1212(79)90022-0.*