ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

DANS LE CADRE DU COURS
MGL804 - RÉALISATION ET MAINTENANCE DE LOGICIELS

ANALYSE DE LA MAINTENABILITÉ D'UN LOGICIEL

PAR ILYASS ARBAOUI

MONTRÉAL, 28 JUIN 2018

Résumé

Ce rapport présente un travail d'analyse de la maintenabilité d'un logiciel. Le logiciel à l'étude est une application Web de petite taille, développée à l'aide du langage de programmation Ruby on Rails. Ce logiciel est utilisé par un regroupement d'organismes, à but non lucratif, oeuvrant dans le domaine de la protection de la jeunesse. Il a été conçu afin de répondre à leurs besoins d'affaires.

Une des motivations premières de ce travail est d'établir une méthodologie d'analyse avec des outils modernes afin d'effectuer une analyse de maintenabilité compète sur un logiciel écrit en Ruby. Il faut préciser ici que le langage Ruby est un langage de programmation interprété. Ces langages sont plus délaissés par le marché des outils d'analyse du code source. Lors de l'analyse de la qualité du code source, une emphase particulière sera faite sur le respect des normes de programmation les plus récentes afin de rendre la méthodologie d'analyse la plus objective et reproductible possible. De plus, une sélection de mesures de la qualité les plus pertinentes sera faite et documentée.

Les outils utilisés, pour effectuer le travail d'analyse sont : 1) Codacy, un outil de revue de code source, 2) RSpec un cadriciel de tests; ainsi que 3) l'outil de visualisation de couverture de tests Simplecov. Avant d'effectuer l'analyse, à l'aide de ces outils, une configuration exhaustive est effectuée et présentée dans ce rapport de session du cours MGL804 tenu à l'été 2018 à l'École de Technologie Supérieure de Montréal.

Enfin, une discussion des résultats obtenus sur le logiciel RQRSDA est présentée et est suivie de recommandations et d'une interprétation de ces résultats.

TABLE DES MATIÈRES

Résumé	2
Liste des tableaux	4
Liste des figures	5
Liste des abréviations, sigles et acronymes	6
1. Introduction	7
2. Définition des étapes d’analyse	8
2.1 Motivation	8
2.2 Logiciel analysé	8
2.3 Critères pour l’analyse	9
2.3.1 Support du langage de programmation Ruby	9
2.3.2 Intégration facile et exécution rapide	9
2.3.3 Utiliser des standards reconnus	10
2.3.4 Facilité de communication des résultats	10
2.3.5 Permet de comparer deux versions du logiciel	10
2.4 Mesures de la qualité du code source	11
2.4.1 Facilité d’analyse	11
2.4.2 Facilité de modification	13
2.4.3 Stabilité	15
2.4.3 Facilité de test	15
2.5 Présentation des outils d’évaluation de qualité	17
2.6 Calibration des outils	18
3. Évaluation	21
3.1 Résultats des mesures obtenues	21
3.2 Analyse des résultats	25
4. Conclusion et recommandations	28
Liste des références bibliographiques	29

Liste des tableaux

Tableau 1 - Liste descriptive des mesures de volumes 12
Tableau 2 - Liste descriptive des mesures de duplication 12
Tableau 3 - Liste descriptive des mesures de taille d’unité 12
Tableau 4 - Liste descriptive des mesures de la couverture de tests 13
Tableau 5 - Liste descriptive des mesures de la complexité 14
Tableau 6 - Liste descriptive des outils utilisés par l’outil d’analyse de Codacy 18
Tableau 7 - Liste descriptive des outils utilisés pour les mesures de couverture de tests 18
Tableau 8 - Vue globale de la complexité du logiciel 22
Tableau 9 - Vue globale de la duplication du logiciel 23
Tableau 10 - Vue globale de la maintenabilité du logiciel 27

Liste des figures

Figure 1 - Configuration des valeurs limites du projet à l'étude 20
Figure 2 - Métriques de volume du logiciel à l'étude 21
Figure 3 - Métrique globale de complexité 22
Figure 4 - fichiers les plus complexes de l'application 23
Figure 5 - Métrique globale de duplication 23
Figure 6 - Métrique globale de couverture de tests 24
Figure 7 - Métrique de l'exécution de la suite de tests 24
Figure 8 - Métrique de couverture des tests par module 24

Liste des abréviations, sigles et acronymes

RQRSDA	Regroupement québécois des ressources de supervision des droits d'accès
CC	Complexité cyclomatique
RoR	Ruby on Rails
ISO	International Organization for Standardization
IEC	Internation Electrotechnical Commission
LOC	Lines of code
ASA	Arbre syntaxique abstrait
REST	Representational State Transfer. Architecture basée sur le protocole HTTP.
CRUD	Verbes d'action du protocole REST (Create, Read, Update, Delete)
JSON	Javascript Object Notation

1. Introduction

La maintenance est une étape importante du cycle de vie d'un logiciel. Malheureusement, les activités entourant celle-ci sont trop souvent effectuées informellement par manque de temps ou tout simplement par manque d'exemples de pratiques exemplaires. En effet, les activités de développement sont souvent séparées des activités de maintenance, ce qui peut créer des conflits entre les développeurs et les mainteneurs lors de la mise en production. La réalité des délais courts et de pression toujours de plus en plus grande pour livrer des nouvelles versions d'un logiciel crée inévitablement une dette technique. La dette technique est une analyse d'un logiciel qui identifie de mauvaises pratiques de conception et de programmation qui causent beaucoup de problèmes et des coûts élevés lors de la maintenance. Pour pallier à ce problème, il existe des outils intéressants d'analyse de la qualité du code source et de revue de code afin de déceler ces problèmes de maintenabilité des logiciels. Ces outils permettent de communiquer la dette technique à l'aide de mesures de la qualité du code source reconnu dans les standards modernes.

Toutefois, ce ne sont pas tous les langages de programmation qui ont la chance d'avoir un écosystème d'outils d'analyse de qualité disponible. La prochaine section introduit la méthodologie d'analyse.

2. Définition des étapes d'analyse

2.1 Motivation

La motivation derrière ce travail d'analyse consiste à configurer et utiliser des outils d'analyse modernes permettant de cerner les problèmes de maintenabilité d'un logiciel réel dans le but d'encadrer le processus de développement (c.-à-d. faire état de la qualité interne du logiciel avant sa mise en production et passage en maintenance). Ainsi, il sera possible d'évaluer la qualité interne du logiciel qui pourra plus facilement être maintenue. Ce travail vise un autre objectif. En effet, les outils plus populaires d'analyse statique ne sont pas compatibles avec tous les langages de programmation. Ainsi, certains langages interprétés sont laissés de côté. Ce travail vise particulièrement le langage de programmation *Ruby* et présente une étude de cas concrète pour illustrer l'utilisation d'outils pour cette technologie.

2.2 Logiciel analysé

Le logiciel analysé est une application Web, en cours de développement, qui a pour clientèle un regroupement d'organismes à but non lucratif oeuvrant dans le domaine de la protection de la jeunesse dans un contexte de séparation des parents. Ce regroupement québécois et connu sous le nom de RQRSDA [1]. Ainsi, cette application est responsable de la gestion des dossiers de famille et des ressources des organismes qui l'utilisent.

Cette application Web a été développée en utilisant le cadriciel Ruby on Rails (RoR) [2]. La persistance des données, qui transigent par cette application, est effectuée sur une base de données relationnelle PostgreSQL [3]. Enfin, les interfaces graphiques sont développées en utilisant les technologies Web populaires : HTML 5, CSS 3 et jQuery.

Le développement de ce logiciel a été effectué, en son entièreté, par moi-même et malgré le fait que j'ai activement tenté d'utiliser les meilleures pratiques de développement pour ce projet, j'aimerais effectuer ce travail d'analyse afin de m'assurer de la maintenabilité du logiciel. De plus, je connais déjà certains modules du logiciel qui sont plus complexes et qui devraient ressortir lors de l'analyse (c.-à-d. avoir besoin d'un travail de réingénierie pour les simplifier).

2.3 Critères pour l'analyse

Bien que l'objectif principal de ce travail est d'effectuer une analyse de la maintenabilité d'un logiciel réel, de la manière la plus objective et reproductible possible, il y a certains critères qui devront être respectés afin de permettre une analyse de qualité qui pourra être reproductible.

2.3.1 Support du langage de programmation Ruby

Un des objectifs de ce travail est de documenter la méthodologie d'analyse et la configuration des outils. Ainsi, l'environnement d'analyse, les outils d'analyse et les outils de test utilisés devront être configurés pour analyser les projets écrits en Ruby.

2.3.2 Intégration facile et exécution rapide

L'analyse de la maintenabilité n'est pas vouée à une exécution ponctuelle unique. L'objectif est de créer un cadre d'analyse documenté qui sera reproductible afin de permettre de répéter l'analyse au besoin. Dans cet ordre d'idée, il est primordial d'utiliser des outils et une méthodologie qui s'intègrent facilement avec la plupart des projets de développement afin de limiter les irritants qui pourraient ralentir l'adoption de ces outils. De plus, une fois configurés, les outils devront être en mesure de rapidement donner les résultats nécessaires afin d'aider à cibler rapidement les problèmes de maintenabilité à corriger. Une exécution ardue des tests et des activités d'analyse, par les développeurs et les mainteneurs, sont des bloquants majeurs pour leur utilisation régulière.

2.3.3 Utiliser des standards reconnus

Il est important de baser la méthodologie d'analyse sur des standards reconnus. Pour ce qui est de la maintenance logicielle, la norme *ISO/IEC 25010:2011* [5] est la norme à utiliser. Cette norme indique quatre attributs de la qualité interne du code source de la maintenabilité d'un logiciel: *facilité d'analyse, facilité de modification, testabilité* et *stabilité*. Ainsi, il est important de relier les activités d'analyse à ces attributs de la maintenabilité afin d'utiliser un cadre objectif et solide sur lequel se baser pour l'analyse et l'interprétation des résultats.

2.3.4 Facilité de communication des résultats

L'analyse de maintenabilité ne servira pas uniquement les ressources qui auront à travailler sur le développement et la maintenance d'un logiciel. Les caractéristiques de maintenabilité d'un logiciel sont intimement liées aux coûts et aux temps nécessaires pour effectuer des modifications ou pour effectuer des corrections sur un logiciel. Ainsi, ces informations représentent une réelle valeur ajoutée pour le client et la direction afin d'établir un plan de maintenance à long terme pour améliorer la situation. Il est donc important de pouvoir communiquer les résultats de l'analyse de la maintenabilité dans un format simple et convivial pour ces parties prenantes.

2.3.5 Permet de comparer deux versions du logiciel

Le cadre d'analyse doit aussi permettre de facilement comparer deux versions d'un même logiciel afin de générer des études comparatives de la maintenabilité du logiciel dans le temps (c.-à-d. est-ce que la situation s'améliore ou se détériore?). Un des intérêts de la répétition des analyses est de pouvoir contrôler la gestion des versions d'un logiciel afin d'identifier et potentiellement bloquer des ajouts ou des modifications qui viendraient dégrader la maintenabilité d'un logiciel en dessous des limites acceptables et préalablement établies. En effet, un tel suivi de la maintenabilité permet de visualiser l'impact de modifications sur la tendance de la maintenabilité du logiciel avant la mise en production.

2.4 Mesures de la qualité du code source

Les outils modernes, d'analyse de la qualité du code source, permettent de générer plusieurs mesures. Bien qu'attrayant, il est important de ne pas perdre son sens critique lorsque l'on utilise de tels outils. Dans cet ordre d'idée, certaines mesures ont été ciblées pour les besoins de l'analyse de maintenabilité selon les attributs reconnus par la norme *ISO/IEC 25010:2011*. Dans les sections qui suivent, des mesures sont sélectionnées afin de mesurer chacune des attributs qualité de la maintenabilité.

2.4.1 Facilité d'analyse

La norme *ISO/IEC 25010:2011* qui décrit l'évaluation de la qualité des logiciels à travers toutes les étapes de leur cycle de vie définit, pour l'étape de la maintenance, la facilité d'analyse comme la facilité d'analyse d'impact en termes d'impacts causés par des modifications demandées. Ainsi, cet attribut concerne la facilité d'estimer les dépendances et les liens entre les différents modules pour être en mesure de prévoir le travail qu'occasionnera une demande de modification dans un logiciel.

Afin d'évaluer cet attribut correctement, j'ai sélectionné, parmi les mesures disponibles, les mesures qui ont un lien direct avec la facilité d'analyse d'un logiciel. La norme *ISO/IEC 25010:2011* reprend une figure représentée dans l'ancienne version de la norme (norme *ISO/IEC 9126:2001*) [4] qui lie les mesures suivantes à la facilité d'analyse d'un logiciel :

Les mesures de volume

Les mesures de volume ont un lien direct avec la taille d'un composant logiciel. Cette propriété d'un logiciel (c.-à-d. le volume) impacte directement la facilité d'analyse de ce dernier puisque plus un logiciel est volumineux, plus l'effort nécessaire pour son analyse est important. Le tableau suivant illustre les sous-mesures qui seront analysées pour cette catégorie de mesure.

NOM	DESCRIPTION
LOC	Nombre de lignes de codes dans le projet
Classes	Nombre de classes dans le projet
Méthodes	Nombre de méthodes dans le projet

Tableau 1 - Liste descriptive des mesures de volumes

Les mesures de duplication

Les mesures de duplication sont intéressantes à considérer pour cette analyse puisqu'une des conséquences d'un trop haut degré de duplication dans un système est de créer un système qui est beaucoup plus gros qu'il ne devrait l'être. Ceci a comme effet d'augmenter les efforts d'analyse lorsqu'une modification doit avoir lieu dans un logiciel. Le tableau suivant illustre les sous-mesures qui seront analysées pour cette catégorie de mesure.

NOM	DESCRIPTION
Nombre de clones	Nombre de blocs de duplication dans un fichier ou dans le projet
LOC dupliquées	Nombre de lignes de code dupliquées dans un fichier ou dans le projet
% duplication	Pourcentage de duplication dans le projet

Tableau 2 - Liste descriptive des mesures de duplication

Les mesures de taille d'unité

Les mesures de taille d'unité sont excellentes pour donner un aperçu de la facilité d'analyse d'un module. En effet, il est intéressant d'analyser la distribution de taille dans un projet par module et par méthodes afin d'identifier les zones nécessitant un travail supplémentaire afin d'améliorer la facilité d'analyse de ces régions. Le tableau suivant illustre les sous-mesures qui seront analysées pour cette catégorie de mesure.

NOM	DESCRIPTION
M/C	Ratio du nombre de méthodes par nombre de classe
LOC/M	Ratio du nombre de lignes de code par nombre de méthodes

Tableau 3 - Liste descriptive des mesures de taille d'unité

Les mesures de couverture de tests unitaires

La présence d'une suite rigoureuse de tests unitaires améliore grandement la maintenabilité d'un logiciel. Dans le cas de la facilité d'analyse, les tests unitaires permettent de mieux comprendre un système et ainsi réduisent considérablement le temps et l'effort nécessaires pour prévoir les impacts de modifications sur un logiciel dans un but d'analyse. Le tableau suivant illustre les sous-mesures qui seront analysées pour cette catégorie de mesure.

NOM	DESCRIPTION
% Couverture	Pourcentage de couverture des lignes de codes par des tests unitaires.
Nombre de tests	Nombre total des tests individuels écrits
Nombre d'erreurs	Nombre total des tests individuels qui ne passent pas
Temps	Temps nécessaire pour exécuter la suite de tests

Tableau 4 - Liste descriptive des mesures de la couverture de tests

Il est important de noter que certaines de ses mesures ne sont pas utiles pour la facilité d'analyse. Par exemple, le temps nécessaire pour exécuter la suite de tests n'est pas utile pour savoir à quel point il est facile d'analyser le logiciel. Toutefois, cette mesure prend toute son importance dans l'attribut de facilité de test. Nous la listons ici afin que ce tableau devienne une référence pour les mesures liées à la couverture de tests. Le lecteur pourra s'y référer à la lecture de la sous-section *Facilité de tests* ou *Testabilité.*

2.4.2 Facilité de modification

Tel que mentionné précédemment, la norme *ISO/IEC 25010:2011* décrit les quatre attributs de la maintenabilité logicielle. Cette dernière décrit la facilité de modification ou la *modifiabilité* comme étant la qualité d'un logiciel à permettre des modifications en utilisant un effort et des ressources minimales. Afin d'évaluer cet attribut correctement, j'ai sélectionné, parmi les mesures disponibles, les mesures qui ont un lien direct avec

la facilité de modification d'un logiciel. La norme *ISO/IEC 25010:2011* qui lie les mesures suivantes à la facilité de modification d'un logiciel :

Les mesures de complexité

La complexité d'un logiciel et de ces modules rend les changements plus difficiles à amener dans un système. En effet, plus un logiciel évolue dans le temps, plus il devient lourd à maintenir et cette réalité est d'autant plus vraie dans le cas des modules complexes. Ainsi, les mesures de cette catégorie jouent un rôle important dans la méthodologie d'analyse de maintenabilité. Le tableau suivant illustre les sous-mesures qui seront analysées pour cette catégorie de mesure.

NOM	DESCRIPTION
Complexité cyclomatique	Nombre de branchements logiques dans un bloc logique de code
Complexité par classe	Complexité cyclomatique pour chaque classe

Tableau 5 - Liste descriptive des mesures de la complexité

Dans le cadre de l'analyse, la complexité cyclomatique [6] sera évaluée pour l'ensemble du projet en utilisant une moyenne ainsi que pour chaque classe logicielle du système afin d'être en mesure d'identifier les zones problématiques à cet égard.

Les mesures de duplication

Comme pour la facilité d'analyse, une duplication trop élevée (> 10%) [9] impact négativement la facilité à apporter des modifications sur un logiciel. En effet, l'effort nécessaire pour amener une modification devra être effectué pour chaque bloc dupliqué. Les mesures de duplication sont les mêmes que pour la sous-section traitée plus haut en lien avec la facilité d'analyse. Une liste descriptive des mesures appartenant à cette catégorie se trouve au tableau 2 du présent document.

2.4.3 Stabilité

Selon la norme ISO/IEC 25010:2011, la stabilité est l'attribut d'un logiciel qui lui permet de limiter les conséquences néfastes causées par une modification ou une correction. Ainsi, la stabilité est fortement liée au concept de modularité. En effet, un logiciel fortement cohésif et faiblement couplé permet une meilleure séparation des responsabilités, ce qui vient améliorer sa maintenabilité.

Afin d'évaluer cet attribut correctement, j'ai sélectionné, parmi les mesures disponibles, les mesures qui ont un lien direct avec la facilité de modification d'un logiciel. La norme *ISO/IEC 25010:2011* qui lie les mesures suivantes à la stabilité logicielle:

Les mesures de couverture de tests unitaires

La présence d'une suite rigoureuse de tests unitaires améliore grandement la maintenabilité d'un logiciel. Dans le cas de la stabilité, les tests unitaires permettent de voir l'effet d'une modification sur les autres modules à chaque exécution. Ainsi, lors de modifications ou de corrections, il est possible de voir si d'autres modules sont affectés si les tests ne passent plus. Il est également possible d'utiliser ces tests pour effectuer un travail de ré ingénierie afin d'améliorer la stabilité du logiciel tout en assurant une régression nulle au niveau fonctionnel.

Les mesures de couverture de tests sont les mêmes que pour la sous-section traitée plus haut en lien avec la facilité d'analyse. Une liste descriptive des mesures appartenant à cette catégorie se trouve au tableau 4 du présent document.

2.4.3 Facilité de test

Selon la norme ISO/IEC 25010:2011, la facilité de test, c'est-à-dire la *testabilité*, est l'attribut d'un logiciel qui lui permet de facilement et rapidement tester les différents modules qui le composent afin de valider un respect des exigences et le fonctionnement attendu. Il s'agit d'un aspect très important pour la maintenabilité d'un

logiciel, car les activités de maintenance peuvent être sérieusement affectées si les mainteneurs n'osent pas faire des changements par peur de briser un élément du logiciel et ne pas être en mesure de le tester avant la mise en production.

Afin d'évaluer cet attribut correctement, j'ai sélectionné, parmi les mesures disponibles, les mesures qui ont un lien direct avec la facilité de modification d'un logiciel. La norme *ISO/IEC 25010:2011* qui lie les mesures suivantes à la stabilité logicielle:

Les mesures de complexité

La complexité d'un logiciel et de ces modules rend le logiciel beaucoup plus difficile à tester. En effet, chaque branchement dans la logique d'un bloc doit être testé. Il est donc facile d'imaginer l'ampleur que peut prendre un système à tester compte tenu de la complexité de certains algorithmes.

Les mesures de couverture de tests sont les mêmes que pour la sous-section traitée plus haut en lien avec la stabilité. Une liste descriptive des mesures appartenant à cette catégorie se trouve au tableau 5 du présent document.

Les mesures de taille d'unité

Certains modules ou fonctions dans un projet peuvent avoir une taille plus importante et nécessiter plus de temps et d'efforts pour permettre une couverture de tests adéquate. Les mesures de taille d'unité présentées au tableau 3 du présent document prennent donc tout leur sens pour être en mesure de déceler les modules ou fonctions plus problématiques et voir si un travail est nécessaire pour les traiter.

Les mesures de couverture de tests

Il va sans dire que les mesures de couverture de tests sont importantes pour évaluer la facilité de test d'un système. Les mesures de cette catégorie illustrent l'écosystème de tests ainsi que le nombre et le temps d'exécution des tests. Ces informations sont importantes afin d'évaluer si un travail est nécessaire pour améliorer l'environnement de tests et les tests actuels. En effet, si l'exécution des tests est trop longue ou trop lourde

à faire, les gens ne la feront pas. Les mesures associées à cette catégorie peuvent être trouvées au tableau 4 du présent document.

2.5 Présentation des outils d'évaluation de qualité

Tel que mentionné précédemment, il y a actuellement un problème avec les outils d'analyse de code statique disponible sur le marché. En effet, beaucoup des outils populaires n'acceptent pas certains langages interprétés puisque ceux-ci ne sont pas compilés et n'ont donc ne peuvent pas générer l'arbre syntaxique abstrait (ASA) requis pour effectuer les mesures. Pour les langages interprétés, il est nécessaire de créer des outils particuliers afin de générer un remplacement à l'ASA pour chaque langage interprété. Pour cette étude, plusieurs outils d'évaluation et d'analyse ont été utilisés conjointement afin de remplacer des solutions disponibles librement comme *SonarQube* [7]. SonarQube ne fait pas l'analyse du langage Ruby.

Le premier outil utilisé est un outil d'analyse et de revue de code statique automatisé nommé *Codacy* [8]. En plus de supporter l'analyse de la syntaxe de *Ruby*, il permet d'utiliser un ensemble d'outils spécialisés afin de générer des mesures de la qualité du code source basées sur la norme ISO/IEC25010:2011.

Dans le cas du projet à l'étude, le tableau suivant résume les différents outils intégrés par l'analyse de Codacy ainsi que leur utilité.

Nom de l'outil	Langage	Utilité
Brakeman	Ruby	Outil disponible librement permettant de vérifier les vulnérabilités de sécurité dans des applications Ruby on Rails
CSSLint et SCSSLint	CSS/SASS	Outil qui vérifie le style et la syntaxe du code CSS et SCSS.
ESLint 4.15.0	Javascript	Outil permettant d'identifier des patrons et des erreurs de syntaxes dans le code Javascript.
PMD	Javascript	Outil qui analyse le code source du code Javascript. Il identifie les erreurs de programmation selon les meilleurs pratiques.

Nom de l’outil	Langage	Utilité
Rubocop	Ruby	Outil qui remplace PMD pour les projets Ruby. Il est entièrement configurable avec des centaines d’options afin d’évaluer la qualité du code source par rapport aux meilleurs pratiques de la communauté. Il permet aussi des vérification de styles similaires à ceux de l’outil Checkstyle.

Tableau 6 - Liste descriptive des outils utilisés par l’outil d’analyse de Codacy

Ainsi, la plupart des mesures identifiées à la section 2.4 seront mesurées à l’aide de l’outil de Codacy. Les seules mesures qui seront produites par un autre outil sont les mesures de couverture de tests.

Une intégration additionnelle a été faite, avec l’outil d’analyse de Codacy, afin de permettre une suite de tests automatiques et permettant de démontrer la couverture dans le journal de visualisation de *Codacy*. Les outils suivants ont été intégrés avec le projet pour permettre ces mesures.

Nom de l’outil	Langage	Utilité
RSpec	Ruby	Outil disponible librement pouvant être intégré aux application Ruby on Rails afin de permettre la création d’un environnement de tests automatisé et l’écriture de tests unitaires pour l’ensemble de l’application.
Simplecov	Ruby	Outil *open source* permettant de créer un rapport détaillé de la couverture des tests effectués par RSpec. Cet outil permet également d’acheminer les résultats des tests à la plateforme de *Codacy* afin de faire le pont avec cette dernière lors des analyses.

Tableau 7 - Liste descriptive des outils utilisés pour les mesures de couverture de tests

2.6 Calibration des outils

Utiliser un outil de revue ou d’analyse de code source sans calibration préalable n’est pas recommandé afin d’obtenir des résultats utiles et qui représentent vraiment la réalité du logiciel analysé. Afin de pallier à ce problème, une série de configurations ont étés effectuées sur les outils d’analyse décrits dans la section précédente afin d’obtenir

de s’assurer que les mesures obtenues représentent bien la réalité de l’application (c’est-à-dire qu’une inspection visuelle confirme que la mesure détecte bien le problème de qualité). En effet, une analyse préliminaire a été effectuée avec les valeurs de configurations par défaut. Une analyse de ces résultats a permis d’identifier les éléments à calibrer pour les analyses subséquentes. Les calibrations ci-dessous ont été effectuées dans ce but.

Filtrer les fichiers du cadriciel

Ce ne sont pas tous les fichiers du projet qui ont été utilisés pour l’analyse du logiciel. En effet, le logiciel analysé a été développé en utilisant le cadriciel *RoR.* Ainsi, il y a beaucoup de fichiers qui ont été générés automatiquement par ce cadriciel. Les fichiers automatiquement générés, qui n’ont pas été altérés ou écrits pendant le développement, ont été retirés de la portée de l’analyse. Cette décision est motivée par la volonté de ne pas diluer les cas problématiques dans un éventail de fichiers ou de code qui obtient de bons résultats pour ne pas fausser les pourcentages et les ratios globaux obtenus pour ce logiciel.

Filtrer les fichiers générés automatiquement et non modifiés

D’autres fichiers qui ont été retirés de la portée de l’analyse sont les fichiers qui sont générés automatiquement lors du développement et qui restent vides ou inchangés. Par exemple, lors du développement d’une application web avec *RoR*, un concept de *scaffolding* existe. Ce concept est une commande qui génère automatiquement tous les fichiers nécessaires pour effectuer les opérations standards de *CRUD* dans une architecture *REST*. Bien que cette commande soit utile afin d’accélérer le processus de développement, elle génère également des fichiers qui seront peut-être utiles dans le futur, mais qui ne sont pas nécessairement utilisés à court terme, tels que des fichiers permettant de répondre à un appel distant du contrôleur en format *JSON*. Ces fichiers n’ont pas été supprimés, car bien qu’ils ne soient pas utiles maintenant, ils pourraient le devenir prochainement si le besoin survient d’utiliser l’application comme une API en découplant les interfaces graphiques à une application client externe.

__Configuration des valeurs limites__

Les outils d'analyse utilisés créent des mesures jugées inacceptables par rapport à l'étendue des fichiers analysés. Ainsi, il est important de configurer les outils d'analyse afin d'évaluer ces mesures selon des valeurs limites acceptables pour le projet à l'étude. La figure suivante illustre les valeurs limites utilisées pour l'analyse de qualité.

Issues are over	20	%
Complexity of files is over	10	%
File is complex when over	10	value(s)
Duplication of files is over	10	%
File is duplicated when over	2	cloned block(s)
Coverage is under	80	%

Figure 1 - Configuration des valeurs limites du projet à l'étude

Afin de suivre les recommandations des experts, par rapport à l'évaluation de la complexité cylcomatique [6], et la maintenabilité d'un code source, nous avons opté pour une valeur limite de 10 plutôt que la valeur par défaut de 20. De plus, compte tenu du fait que le logiciel consiste majoritairement d'opérations *CRUD* sur des ressources et que la logique est répétée pour toutes celles-ci, le nombre de blocs dupliqués pour considérer un fichier comme étant dupliqué a aussi été calibré de 1 à 2.

Enfin, compte tenu du caractère interprété du langage utilisé pour ce logiciel, il est important d'assurer une bonne couverture de tests afin de ne pas avoir de surprise lors de la mise en production. En effet, même si une erreur flagrante est présente dans un fichier. Le logiciel va marcher jusqu'à ce qu'un utilisateur enclenche une action qui utilise la partie du code concerné. Il est donc primordial de maximiser la couverture du code. Ainsi, la valeur limite de la couverture a été mise à 80% au lieu du 60% par défaut.

Configuration des outils utilisés par Codacy

Enfin, la dernière calibration concerne l'éventail d'outils intégrés par l'outil d'analyse et de revue de code *Codacy*. Les paramètres de ces outils ainsi que chaque règle de style ont été revues afin d'assurer que les règles soient toutes appropriées pour le logiciel à l'étude. Les résultats obtenus à partir des valeurs par défauts de l'outil ont donné des résultats fiables puisque les valeurs par défaut des outils d'analyse sont déjà préconfigurés pour des applications *RoR* dans le profil de langage pré chargé par l'outil pour des projets *Ruby*.

3. Évaluation

Cette section présente les résultats de l'analyse effectuée par les outils intégrés à *Codacy* ainsi que les outils de développement qui ont été intégrés à l'environnement de développement et de test de l'application.

3.1 Résultats des mesures obtenues

Résultats des mesures de volume et de taille d'unité

La figure suivante représente les mesures documentées comme étant des mesures de volume.

Name	Lines	LOC	Classes	Methods	M/C	LOC/M
Controllers	2228	1561	22	210	9	5
Helpers	46	44	0	0	0	0
Models	885	582	27	57	2	8
Mailers	14	13	2	1	0	11
Javascripts	353	194	0	28	0	4
Libraries	120	77	0	5	0	13
Tasks	0	0	0	0	0	0
Mailer specs	36	25	1	0	0	0
Model specs	304	203	0	0	0	0
Request specs	187	169	0	0	0	0
Routing specs	654	501	0	0	0	0
Controller specs	2922	2291	0	0	0	0
Helper specs	17	1	0	0	0	0
Total	7766	5661	52	301	5	16

FIGURE 2 - MÉTRIQUES DE VOLUME DU LOGICIEL À L'ÉTUDE

Les fichiers responsables pour les vues ne sont pas inclus dans ce tableau récapitulatif. Un filtre a été placé sur les fichiers *HTML et CSS* afin de mettre l'emphase sur les fichiers qui gèrent une logique d'affaires. Ainsi, avec 5661 lignes de codes majoritairement écrites en *Ruby* et quelques scripts *front end* en JavaScript, 52 classes et 301 méthodes, le logiciel est catégorisé comme étant de petit volume.

De plus, le ratio moyen de lignes de code par méthodes ainsi que le ratio moyen de méthodes par classes sont très bas. En effet, un ratio moyen de 5 méthodes par classe ainsi qu'un ratio moyen de 16 lignes de code par méthode permet d'affirmer que le logiciel est de petite taille et que la distribution de ces modules est petite et uniforme. Bien que certaines classes et méthodes comportent des ratios plus élevés, ces ratios demeurent toujours très acceptables.

Résultats des mesures de complexité

La figure suivante représente le pourcentage de fichiers jugés complexes dans l'application.

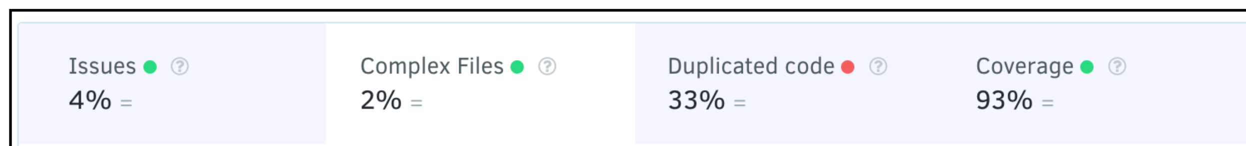

FIGURE 3 - MÉTRIQUE GLOBALE DE COMPLEXITÉ

Cette valeur indique le pourcentage de fichiers évalués « complexes » parmi l'ensemble des fichiers inclus dans la portée de l'analyse. Dans ce cas-ci, les fichiers évalués « complexes » sont les fichiers ayant une méthode avec une complexité cyclomatique supérieure à 10.

Le tableau suivant illustre les valeurs agrégées et unitaires de complexité cyclomatique.

Complexité cyclomatique totale	**873**
Complexité par méthode moyenne	**2.9**

Tableau 8 - Vue globale de la complexité du logiciel

Ces valeurs sont excellentes et illustrent que le logiciel a un très faible degré de complexité. Toutefois, on peut constater que certains fichiers comportent des niveaux de complexité plus importants.

GRADE	FILENAME	ISSUES	DUPLICATION	COMPLEXITY	COVERAGE
B	app/controllers/service_requests_controller.rb	1	2	15	93%
A	lib/utilities/event_utility.rb	1	0	14	-
C	app/models/service_request.rb	0	4	11	77%

FIGURE 4 - FICHIERS LES PLUS COMPLEXES DE L'APPLICATION

Ces fichiers ont un niveau moyen de complexité par méthode dépassant la limite établie de 10 pour la valeur de complexité cyclomatique par méthode.

Résultats des mesures de duplication

La figure suivante représente le pourcentage de fichiers comportant de la duplication dans l'application.

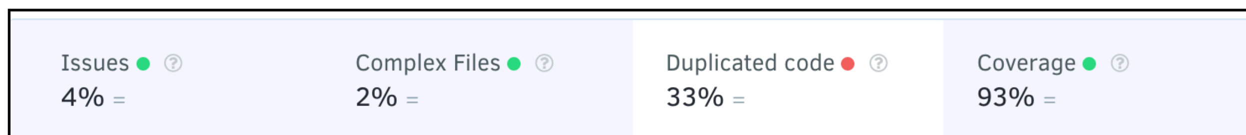

FIGURE 5 - MÉTRIQUE GLOBALE DE DUPLICATION

33% des fichiers inclus dans la portée de l'analyse comportent donc plus de deux blocs de duplication de code. Le tableau suivant illustre les valeurs agrégées de duplication dans l'application.

Pourcentage de duplication	**33%**
Lignes de code dupliquées	**2780**
Blocs dupliqués	**255**
Nombre de fichiers avec plus de deux blocs dupliqués	**49**

Tableau 9 - Vue globale de la duplication du logiciel

Les valeurs sont anormalement élevées. Toutefois, ce sujet sera traité dans la section d'analyse qui suivre la présentation des résultats.

Résultats des mesures de couverture de tests

La figure suivante représente le pourcentage de lignes de code testé par la suite de tests.

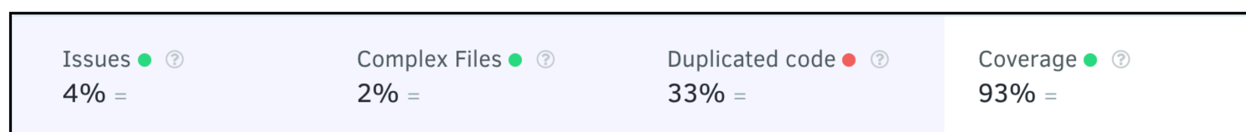

FIGURE 6 - MÉTRIQUE GLOBALE DE COUVERTURE DE TESTS

Ainsi, le logiciel obtient 93% de couverture de tests ce qui est nettement supérieure à la valeur limite de vérification spécifiée à 80%. La figure suivante représente le nombre de tests effectués, le nombre de tests qui résultent en une erreur ainsi que le temps nécessaire pour exécuter la suite de tests.

```
Finished in 20.32 seconds (files took 4.96 seconds to load)
396 examples, 0 failures
```

FIGURE 7 - MÉTRIQUE DE L'EXÉCUTION DE LA SUITE DE TESTS

Enfin, la figure suivante représente la distribution des tests avec un pourcentage de couverture pour chaque module de l'application.

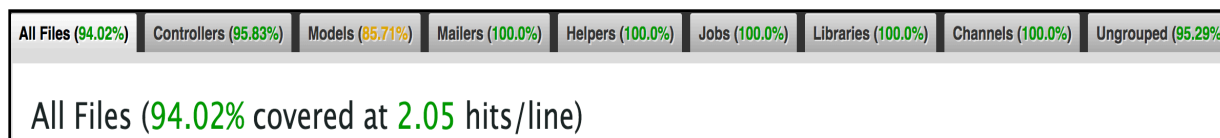

FIGURE 8 - MÉTRIQUE DE COUVERTURE DES TESTS PAR MODULE

Les résultats par rapport à la couverture des tests unitaires sont excellents. En plus, d'avoir une couverture de test intéressante pour l'application, la suite de tests est exécutée rapidement, ce qui est excellent d'un point de vue de maintenabilité.

3.2 Analyse des résultats

Pour demeurer objectif et rigoureux dans la méthodologie d'analyse ainsi que pour appuyer le travail d'analyse sur des normes et des standards solides, l'analyse des résultats se fera selon le modèle du SIG [4]. En effet, ce dernier se base sur les quatre caractéristiques de la maintenabilité telles que documentées dans la norme *ISO/IEC 25010:2011.*

Analyse des résultats du volume du logiciel

Les résultats obtenus pour les mesures de volume, pour ce logiciel, indiquent qu'il s'agit d'un petit logiciel. Cela lui donne donc une bonne note par rapport à la maintenabilité puisque les petits logiciels sont reconnus comme étant beaucoup plus maintenables d'un point de vue de facilité d'analyse. Ainsi, après analyse, la note selon le module du SIG est de **++**.

Analyse des résultats de complexité du logiciel

Le logiciel a d'excellents résultats par rapport à la complexité. En effet, le logiciel a une moyenne de 2.9 CC par méthode. De plus, en analysant le logiciel de plus prêt, seulement trois fichiers comportent des méthodes ayant des complexités cyclomatiques plus élevées que la valeur limite suggérée par le modèle SIG (10 CC). Ceci dit, la valeur de complexité demeure relativement faible et ne dépasse jamais la valeur de 20 CC pour les méthodes les plus complexes, ce qui est un bon point pour ce logiciel.

De plus, après analyse, on observe que les classes les plus complexes sont effectivement celles qui ont été identifiées comme telles lors du développement du logiciel. En effet, ces classes sont responsables d'une fonctionnalité importante, mais

complexe de gestion des calendriers communs entre plusieurs intervenants. Ce besoin crée donc une complexité connue dans l'application.

Le modèle SIG attribue une note de **++** aux applications n'ayant aucune méthode avec une valeur CC plus élevée que 21, ce qui est notre cas.

Analyse des résultats de duplication du logiciel

Ces résultats sont très intéressants. En effet, les valeurs obtenues de duplication représentent 33% de l'application ce qui excède nettement la limite maximale accordée par le modèle SIG. En effet, selon le modèle, la note obtenue de cette évaluation est de **- -**.

Toutefois, il est important d'effectuer une validation visuelle des mesures. Le logiciel analysé est une application Web développée avec le cadriciel *RoR*. Ce cadriciel permet de générer beaucoup de code automatiquement pour le développeur afin de minimiser l'effort à écrire du code redondant pour des actions standards, telles que les opérations *CRUD* sur une ressource ainsi que la création des interfaces graphiques standards. Conséquemment, le cadriciel, une fois intégré avec des outils de tests, génère également une suite de tests standards pour les fichiers préalablement générés. Certes, de la personnalisation est à faire, toutefois, le squelette demeure pour la majeure partie inchangée compte tenu de la nature hautement standardisée par convention de *RoR*. Ainsi, le résultat obtenu par rapport à la duplication, bien qu'étant mauvais d'un point de vue de la mesure effectuée, demeure acceptable après l'analyse visuelle.

Analyse des résultats de taille d'unité du logiciel

Le logiciel a une moyenne de 16 lignes de code par méthode. De plus, bien que cette moyenne soit relativement faible, les méthodes les plus volumineuses vont jusqu'à 60 lignes de code. Tel que mentionné précédemment, cette catégorie de mesure coïncide bien avec la mesure de complexité, car ce sont les mêmes méthodes qui ont les plus hautes valeurs de lignes de code. Ainsi, selon le modèle du SIG, la note est de **-**.

Analyse des résultats de couverture de tests du logiciel

La couverture du logiciel est élevée et le temps d'exécution des tests est bas. Toutefois, pour avoir une note parfaite du côté du SIG, il faut une couverture de tests excédant la barre des 95%. Ainsi, le résultat de l'analyse pour cette catégorie de mesure est **+**.

Le tableau suivant présente un résumé des résultats de l'analyse de maintenabilité du logiciel à l'étude.

	Volume	Complexité	Duplication	Taille d'unité	Couverture des tests	Résultat
Facilité d'analyse	++		- -	-	+	o
Facilité de modification		++	- -			o
Stabilité					+	+
Facility à tester		++		-	+	++

Tableau 10 - Vue globale de la maintenabilité du logiciel

4. Conclusion et recommandations

La méthodologie utilisée pour l'analyse a permis d'effectuer une analyse complète et objective de la maintenabilité d'un logiciel développé dans un langage de programmation peu supporté par les outils disponible librement d'analyse sur le marché (*Ruby*). De plus, les outils utilisés ont permis l'utilisation de mesures concrètes et reconnues par les experts et les standards reconnus. De plus, la méthodologie d'analyse a répondu aux critères énoncés au début du travail. En effet, la méthodologie d'analyse permet une analyse rapide et facile avec des outils qui s'intègrent facilement à l'environnement de développement et de maintenance. En plus de respecter les normes de maintenance et d'évaluation de la qualité logicielle en vigueur, les résultats obtenus sont clairs et peuvent être facilement être interprétés et communiqués aux différentes parties prenantes, et ce, peu importe leur niveau d'expertise technique.

Enfin, le logiciel à l'étude a obtenu de bons résultats lors de l'analyse de maintenabilité. Sa seule faiblesse se situe au niveau de la duplication qui semble beaucoup trop important. Il faudrait décider de l'importance de cette faiblesse, dans l'application, lors de la prochaine itération de maintenance. Lors de cette décision, il sera intéressant de débattre du niveau anormal de duplication qui peut être expliqué par la nature même du cadriciel de développement utilisé (c.-à-d. RoR) qui force cette philosophie de convention dans ses meilleures pratiques de la communauté. Ainsi, ce dernier génère beaucoup de code dupliqué qui fait réagir l'outil d'analyse de qualité de code source négativement.

Ceci dit, à la suite de cette expérimentation, il est clair que ces outils d'analyse seront davantage explorés et continueront à prendre de la popularité à l'avenir. Je crois fermement au contrôle de la dette technique en créant un environnement automatique d'analyse et de revue afin de surveiller la qualité logicielle lors du développement.

Liste des références bibliographiques

[1] RQRSDA. Regroupement québécois des ressources de supervision des droits d'accès. Consulté le 28 juin 2018. http://rqrsda.org/index.php

[2] RAILS. Ruby on Rails. Consulté le 28 juin 2018. https://rubyonrails.org/

[3] POSTGRESQL. The world's most advanced open source relational database. Consulté le 28 juin 2018. https://www.postgresql.org/

[4] Heitlager I., Kuipers T., and Visser J. (2007) "A Practical Model for Measuring Maintainability", 6th Int'l Conf. on the Quality of Information and Communications Technology (QUATIC '07), IEEE Computer Society Press.

[5] ISO/IEC (2011), 'ISO/IEC 25010 System and software quality models' , ISO/IEC .

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

[7] Sonarqube. The leading product for continuous code quality. Consulté le 28 juin 2018. https://www.sonarqube.org/

[8] Codacy. Automatic code review. Consulté le 28 juin 2018. https://www.codacy.com/

[9] Codacy. ISO 25010 Software Quality Model. Consulté le 28 juin 2018. https://blog.codacy.com/enterprise-software-a-summary-of-iso-25010-software-quality-model-7100575d6f6