UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES

PAR
OUSSAMA HACHEMANE

ÉVALUATION DE L'IMPACT DU REFACTORING BASÉ SUR LES CLONES SUR LA QUALITÉ (MAINTENABILITÉ-TESTABILITÉ) DES SYSTÈMES ORIENTÉS OBJET

DÉCEMBRE 2015

REMERCIEMENT

Je tiens à exprimer ma profonde gratitude envers notre Créateur, pour sa clémence et pour m'avoir donné le courage, la volonté, l'espoir et surtout la santé pour réaliser ce mémoire.

J'exprime mes sincères remerciements à mes respectueux directeur et directrice de recherche Mr Mourad Badri et Mme Linda Badri. Je vous remercie de m'avoir accueilli, conseillé, encouragé, soutenu depuis le début et à chaque moment et d'avoir mis en valeur le présent travail par vos propositions, remarques et corrections considérables. Trouvez ici l'expression de ma profonde gratitude.

À mes chers parents Mouloud et Hasna, à mes frères et sœurs Souheil, Serine, Abdelmalek et Oumaima, qui m'ont vraiment encouragé, soutenu et aidé. Je ne vous remercierai jamais assez de m'avoir accompagné par votre soutien affectif, moral, matériel et par vos encouragements continuels tout au long de mon parcours. Que le présent travail soit le reflet de mon immense reconnaissance et amour.

Je voudrais exprimer ma reconnaissance envers tous mes amis et collègues qui m'ont apporté leur support moral et intellectuel tout au long de mon travail.

Mes sincères remerciements vont également à tous les professeurs du Département de mathématiques et d'informatique de l'Université du Québec à Trois Rivières qui m'ont enseigné et qui étaient toujours disponibles pour m'aider.

RÉSUMÉ

Le refactoring (remaniement) représente une démarche continue, qui vise à pallier activement aux problèmes d'évolution logicielle comme la présence de clones dans le code source. Le refactoring vise essentiellement à améliorer la qualité (structurelle) et les caractéristiques non fonctionnelles d'un programme en modifiant son code source sans altérer son comportement.

La qualité se dégrade en présence de clones (ou de code dupliqué) en rendant la maintenance plus difficile. Le développeur se trouve obligé (par exemple) de vérifier tous les fragments similaires pour corriger les fautes suite à une défaillance donnée.

Les travaux traitant de l'impact du refactoring basé sur les clones ne sont pas très nombreux. Dans ce mémoire, nous visons à conduire une étude empirique en vue d'évaluer l'impact du refactoring orienté objet basé sur les clones sur la qualité des logiciels (du point de vue de la testabilité en particulier). Nous nous intéressons à la testabilité des classes sous la perspective particulière des tests unitaires (l'effort requis pour les écrire). Dans ce contexte, nous avons utilisé deux systèmes open source développés en Java (Ant et Archiva). Dans notre analyse, nous avons utilisé plusieurs classes de ces applications avant et après le refactoring. Pour quantifier l'effort de test unitaire pour chacune des classes considérées, nous avons utilisé trois métriques de test. Les classes de test correspondantes aux classes analysées ont été générées en utilisant l'outil CodePro. Nous avons également utilisé un ensemble de métriques orientées objet pour mesurer les différents attributs de code source (couplage, complexité, taille, cohésion et héritage), attributs reliés à la testabilité unitaires des classes. En vue d'évaluer l'impact du refactoring orienté objet des clones sur la testabilité unitaire des classes, nous avons utilisé des tests statistiques, des corrélations et la régression linéaire simple et multiple. Les résultats suggèrent que : la testabilité unitaire des classes a été améliorée après l'application du refactoring basé sur les clones, les variations des attributs du code

source ont un impact sur les attributs du code des tests unitaires, et ces derniers sont plus influencés par les variations de la complexité et la taille des classes comparativement aux autres attributs du code source.

ABSTRACT

This study aims at empirically investigating the effect of refactoring based on clones on the unit testability of classes in object-oriented software. We investigate unit testability of classes from the perspective of unit testing effort, and particularly from the perspective of unit test case construction. We focus on the effort required to write the code of unit test cases. We also investigated unit testability of classes from the perspective of source code attributes.

We performed an empirical analysis using data collected from two well-known open source Java software systems (ANT and ARCHIVA). We used, in fact, two versions of each refactored class: the class before refactoring, and the class after refactoring. To capture the unit testing effort of classes, we used three metrics to quantify various perspectives related to the code of corresponding JUnit test cases. JUnit test cases have been generated using a tool (CodePro). We also used object-oriented metrics to measure various source code attributes (coupling, cohesion, inheritance, complexity and size), attributes that are related to unit testability of classes. In order to investigate the effect of refactoring based on clones on the unit testability of classes, we used statistical tests, correlation analysis and linear regression analysis. Results provide evidence that the unit testability of the refactored classes has been improved.

TABLE DES MATIÈRES

Liste des tableaux

Liste des figures

Chapitre 1 – Introduction

Introduction

Les systèmes logiciels orientés objets sont de plus en plus volumineux et complexes. Les exigences en termes de qualité ont également augmentées. Par ailleurs, la présence de clones dans la plupart des systèmes développés nuit à la qualité de ces derniers [1, 2, 3]. Ce problème est devenu un enjeu primordial qu'il faut absolument résoudre ou maîtriser.

Les clones en tant qu'éléments importants dans la création des logiciels représentent une mauvaise pratique de programmation et un des facteurs qui contribuent à la dégradation de la qualité des logiciels [4]. Il s'agit de copier des fragments de code existants et de les insérer dans divers endroits du code, avec ou sans modifications, ce qui peut donner lieu à différents types de clones. Les clones, du point de vue des chercheurs, représentent un sujet très délicat vu leur impact direct sur la qualité et la maintenance des logiciels [1, 2, 4, 5]. György Hegedus et al. [6] ont cité l'influence négative possible de la présence des clones dans le code source sur les deux attributs de qualité : analysabilité et changeabilité, ainsi que sur leurs sous caractéristiques. Francesca Arcelli Fontana et al. [14] ont présenté à travers une analyse l'effet positif sur la plupart des métriques de qualité après élimination du code dupliqué en utilisant le refactoring.

Afin de résoudre cette problématique liée au code dupliqué et à la dégradation de la qualité, le refactoring a été sélectionné comme une activité d'ingénierie logicielle qui consiste à modifier le code source d'une application sans altérer son comportement. Il permet aussi de réduire les coûts de la maintenance en améliorant la qualité en termes de performance et de fiabilité [3]. Le refactoring de clones vise à éliminer les duplications dans le code source afin d'améliorer certaines de ses caractéristiques [7]. L'objectif principal de notre travail est de voir les effets possibles du refactoring sur la testabilité des logiciels.

Introduction

Plusieurs travaux portant sur le refactoring orienté objet et les clones ont été réalisés afin d'améliorer la qualité des systèmes logiciels. Chaque attribut de qualité est influencé par certaines propriétés du code source comme le nombre de ligne de code, la complexité et le couplage.

Notre travail consiste à évaluer l'impact du refactoring des clones sur la testabilité unitaires des classes. Ce travail passe d'abord par l'élaboration d'un cadre permettant d'évaluer la testabilité sous la perspective particulière de la construction des tests unitaires. Des systèmes logiciels orientés objet développés avec Java sont analysés afin d'extraire les classes ayant subi un refactoring des clones. Une évaluation en utilisant des tests statistiques est faite sur les classes des systèmes sélectionnés avant et après le refactoring en vue d'identifier les métriques (attributs du code source) ayant le plus d'effet (impact) sur la testabilité unitaires des classes suite à l'élimination des clones.

2.1 Introduction

Dans ce chapitre, nous allons présenter trois axes principaux reliés à cette étude: les clones, le refactoring (le remaniement), et la qualité (sous la perspective de la testabilité) des systèmes orientés objet.

La présence de clones est l'un des facteurs qui peut nuire à la qualité des logiciels. Le code dupliqué ou les clones, comme façon très courante dans la création des logiciels, représente une mauvaise pratique de programmation et parmi l'un des facteurs importants conduisant à la dégradation de la qualité [4]. La détection des bogues liés à un clone lors de l'exécution des programmes, oblige le programmeur à vérifier tous les fragments similaires pour éviter les mêmes bogues. Des problèmes de défaillance de logiciels, des difficultés d'entretien et l'augmentation considérable de la taille du projet peuvent être liés à cette façon de programmer (utilisation de clones) [1, 2, 7].

Le refactoring comme processus faisant face aux défis imposés par les différents changements à travers les activités de maintenance, s'est avéré une alternative très intéressante pour améliorer la qualité d'un logiciel donné. L'ensemble des chercheurs le catégorise parmi les solutions efficaces pour éliminer les codes dupliqués [3, 6, 7, 8].

Ce chapitre présente une revue sommaire de la littérature portant sur les clones dans les systèmes orientés objet, ainsi que le refactoring et son impact sur la qualité des systèmes logiciels.

2.2 Les clones dans les systèmes orientés objet

Les clones sont considérés comme une mauvaise pratique de programmation dans l'ingénierie de logiciel. Ils représentent toutes sortes de redondances, réplications et duplications de code. Les clones se créent généralement par la copie des fragments de code existants et de leur réutilisation (collage) avec ou sans modification. Il existe plusieurs définitions de clones dans la littérature. Baxter et al. [1] définissent les clones comme des fragments de code similaires en absence de définition exacte de la similitude. Par contre, Kamiya et al. [2] les définissent comme toutes parties identiques ou similaires à d'autres parties d'un fichier source.

Les clones du point de vue des chercheurs sont traités comme un sujet très délicat vu leur impact direct sur la qualité, en particulier sur la maintenance qui représente l'une des phases les plus coûteuses du processus de développement [1, 2, 4, 5, 9]. La maintenance est définie comme la modification du produit logiciel après sa livraison afin de corriger les erreurs résiduelles, améliorer sa performance ou l'adapter à un autre environnement [9].

Plusieurs travaux portent sur la présence des clones dans le code source, leur importance et leur impact possible. Hotta et al. [10] ont conduit une étude empirique sur l'influence de la présence du code dupliqué. La mesure d'impact a été réalisée à travers la définition d'un nouvel indicateur appelé «fréquence de modifications». Cet indicateur mesure l'impact de la présence des clones comparativement à leur élimination avec la fréquence de modifications. Ils ont conclu que les clones affectent l'évolution du logiciel et sa stabilité, car les codes dupliqués sont susceptibles d'avoir plusieurs modifications pendant l'évolution.

Kim et al. [8], ont conduit une étude avec le même objectif, mais d'une façon différente. Ils ont développé une définition formelle de l'évolution des clones, ainsi qu'un outil de

généalogie des clones. Selon ce groupe de chercheurs, cet outil permet d'extraire l'historique des clones à partir des dépôts de code source. Le refactoring des clones était déduit comme la solution idéale pour régler les problèmes de présence de clones dans le code source, mais ce n'est pas toujours le cas. Le refactoring de clones n'est pas préférable sur des clones volatils (les clones qui disparaissent avec l'évolution du logiciel) ou sur certains clones à long terme (les clones qui ne sont pas susceptibles de subir de refactoring).

2.2.1 Les raisons de l'utilisation des clones

Il existe plusieurs facteurs qui forcent le programmeur à créer des clones dans un système. Mais parfois, ils se créent accidentellement [3]. La liste est longue [4], mais dans ce qui suit, nous allons citer les cas les plus importants:

- Le premier cas s'inscrit parmi les stratégies de développement. Cette catégorie représente le recours à la duplication de code en vue de le réutiliser dans un autre endroit du même projet. Il se fait soit par un copier/coller simple, soit par le Forking, un terme introduit et utilisé par Kapser et al. [5]. Il représente la réutilisation des solutions similaires en espérant qu'elles vont diverger avec l'évolution du système [3].

- Fusionner deux ou plusieurs systèmes contenant des procédures identiques dans le but de créer un nouveau système s'inscrit parmi les cas de la deuxième catégorie. Cette opération donne une forte probabilité d'avoir un nouveau logiciel avec un nombre important de codes dupliqués [3].

2.2.2 Les types de clones et l'évolution des systèmes

Chanchal Kumar et al. [4], citent deux genres de similarité entre deux fragments de code : la similitude textuelle, et la similitude fonctionnelle.

- ✓ **La similarité textuelle** : Elle est basée sur la similarité des textes de code, en copiant des fragments et en les collant dans d'autres emplacements:
  - Type I: des fragments de code identiques, sauf pour les variations de blancs et de commentaires.
  - Type II: des fragments identiques structurellement / syntaxiquement à l'exception de variations des identifiants, des littéraux, des types, et des commentaires.
  - Type III: des fragments copiés avec d'autres modifications; Les déclarations peuvent être modifiées, ajoutées où supprimées en plus des variations dans les identificateurs, les littéraux, les types, et les commentaires.
- ✓ **La similarité fonctionnelle**: ce sont les cas où les fonctionnalités de deux fragments de code sont identiques ou similaires:
  - Type IV: il inclut deux ou plusieurs fragments de code qui effectuent le même calcul, mais ils sont différents en termes de variantes syntaxiques.

La création de clones dans les systèmes orientés objet peut dépendre fortement de l'évolution de ces derniers [11]. Il peut y avoir des clones au sein d'un système, mais qui disparaissent avec l'évolution du système en raison de travaux de maintenance. Ce genre de clones s'appelle : **les clones volatils**. L'autre type de clones, **les clones à long terme**, sont des cas considérés en général comme des clones qui ne sont pas susceptibles de subir de refactoring. Donc, ils vont rester dans toutes les versions du système.

2.3 Le refactoring orienté objet

Le refactoring est une activité d'ingénierie logicielle qui consiste à modifier le code source d'une application de manière à améliorer sa qualité sans altérer son comportement (du point de vue de ses utilisateurs) [3]. Cette activité permet de réduire les coûts de maintenance et d'améliorer la qualité [3, 7, 9]. Par ailleurs, la maintenance (corrective et évolutive) concerne l'évolution d'un logiciel. Il s'agit de répondre à l'urgence et d'être réactif. Elle permet de faire la modification directe sur le comportement d'un logiciel, en termes de correction de bogues et d'ajout ou d'amélioration des fonctionnalités [3]. Ce qui est complètement différent du refactoring.

Le refactoring représente une démarche continue, qui vise à pallier activement aux problèmes d'évolution. Son application a été proposée en parallèle avec la maintenance dès lors qu'il est compatible avec ses contraintes de réactivité, car plus un refactoring est effectué tôt, moins il coûte cher [3]. Le refactoring étant un processus délicat qui améliore les caractéristiques non fonctionnelles comme la changeabilité et la stabilité. Il permet aussi d'assurer que [12]:

- Toute information nécessaire est disponible.
- Toute information redondante sera supprimée.
- Simplification des algorithmes et des méthodes.
- La complexité de chaque classe sera limitée.
- Le nombre de classes sera limité.

Selon leur complexité, trois types de refactoring ont été proposés : chirurgical, tactique et stratégique [3]:

- Pour le refactoring chirurgical, c'est la réalisation d'opérations de refactoring limitées et localisées dans le code source en remaniant quelques composantes sans altérer leurs relations avec le reste du logiciel.

- Par contre, le refactoring tactique c'est la refonte complète de quelques composants repensés tant dans leur fonctionnement interne que dans leurs relations avec le reste du logiciel. Par exemple, introduire des designs patterns dans le code source.

- Pour le refactoring stratégique, c'est la remise à plat de la conception du logiciel afin de l'adapter aux présentes et futures exigences des utilisateurs.

2.3.1 Bénéfices du refactoring

Le refactoring [13] est l'une des meilleures méthodes utilisées pour améliorer la qualité du logiciel en général et la maintenabilité en particulier, car il mène au développement de bonnes applications. Par conséquent, la conception de code déjà en place va être améliorée sans altérer le comportement de l'application :

- Le refactoring [14] représente des transformations sécuritaires de code source afin d'améliorer la qualité.
- Le refactoring [14] peut résoudre les problèmes liés aux mauvaises conceptions, car elles sont difficiles à maintenir et à tester.
- Le refactoring [15] vise principalement à améliorer les métriques logicielles représentant les propriétés internes de la qualité comme le nombre de lignes de code et la complexité des méthodes, et par conséquent les attributs externes comme la maintenabilité et la testabilité.
- Il existe [16] plusieurs méthodes de refactoring, et chacune a un effet particulier sur les attributs internes et externes.
- Le refactoring [17] facilite la compréhension ainsi que la localisation des bogues dans le code source.

2.3.2 Quand et où est ce qu'il faut appliquer le refactoring

La meilleure conception de logiciel [3, 13] correspond à la meilleure façon d'anticiper le refactoring afin d'offrir plus de flexibilité pour affronter et faire face aux évolutions. Pour cela, il est nécessaire d'utiliser les meilleures pratiques en la matière, formalisées notamment sous forme de design patterns. Il est important de séparer au maximum les préoccupations métiers et techniques, de manière à limiter les effets de bord lors d'un refactoring.

Quand on parle de refactoring, on se demande quand est ce qu'il faut l'appliquer. Martin Fowler [67] cite dans son livre, que le refactoring est une opération qui ne doit pas être faite périodiquement mais plutôt quand on :

- Ajoute une nouvelle fonctionnalité : l'ajout de nouvelles fonctionnalités [13] dans un code (peu clair) est très difficile et risqué. Il faut donc essayer d'améliorer le design du code en utilisant le refactoring afin de faciliter l'ajout.

- Trouve des bogues dans le code : un code clair [13] est plus facile à comprendre et aide à mieux détecter les bogues.

La détection des bad smells [3, 13, 17] est une technique très répandue permettant d'identifier les endroits où il faut appliquer le refactoring. Ils sont considérés comme des points noirs et ambigus du point de vue de la qualité (les approches et outils seront cités dans les sections 2.3.3 et 2.3.4).

2.3.3 Approches d'anticipation du refactoring basées sur les clones

Il existe différentes approches d'anticipation du refactoring. Ces dernières offrent des opportunités pour rechercher les clones :

- **Approches basées sur le texte** : Ces approches [5, 18, 19] considèrent que le programme cible est une séquence de lignes de code. Il existe une étape de filtrage de code avant l'analyse de similarité. Elle comporte la suppression des commentaires, la suppression des espaces, la normalisation et le remplacement des identificateurs par des caractères spéciaux. Ces approches comparent les fragments de code en termes de textes et de chaînes de caractères. Simian [19] propose un outil pour la détection de clones basé sur le texte. Ce dernier est capable de détecter des clones dans plusieurs langages de programmation.

- **Approches basées sur les jetons** : Ces approches [2, 5, 18] utilisent un analyseur syntaxique pour placer les jetons dans le code. Elles font ensuite la recherche sur la base de ces jetons, sachant que la même série de jetons apparaît dans les fragments similaires. Elles renomment également les identifiants et elles ignorent les espaces. CCFinder [2] et CP-Miner [5] sont deux exemples importants de l'approche basée sur les jetons.

- **Approches basées sur les AST** : Les arbres syntaxiques abstraits sont une manière de représenter le code source avec toutes ses caractéristiques. Ces approches [1, 5, 20] représentent le code source sous forme d'un arbre syntaxique abstrait (AST). Elles utilisent ensuite des algorithmes différents afin de retourner les sous arbres similaires. Baxter et al. [1] proposent un détecteur de clones basé sur les AST. Il détecte les clones en trouvant des sous arbres identiques.

- **Approches basées sur la similarité sémantique**: Ce sont des approches [5, 18, 21] basées sur les graphes de dépendances dans un programme (PDG). Elles cherchent des représentations de grande abstraction du code source. Un PDG contient le flux de contrôle et les flux de données d'un programme. Une fois qu'un ensemble de PDGs est obtenu à partir d'un programme donné, l'algorithme isomorphe de la correspondance des sous-graphes est appliqué pour trouver des sous-graphes similaires qui seront renvoyés comme des clones.

- **Approches basées sur les métriques** : Ces approches [5, 22] calculent pour chaque fragment à analyser un certain nombre de métriques. Ces dernières vont être comparées afin de détecter les clones dans le code source sachant que chaque métrique donne une façon de mesurer une caractéristique du code. Mayrand et al. [23] ont développé un outil basé sur cette approche permettant de comparer deux fragments afin de détecter les clones. Cet outil calcule les métriques à partir de l'arbre syntaxique abstrait du code source en question. Elles sont calculées sur les expressions et le flux de contrôle des fonctions.

2.3.4 Outils d'anticipation du refactoring basé sur les clones

Après avoir vu les différentes approches de refactoring, nous allons présenter les outils les plus utilisés:

- PMD[1] est parmi les outils d'anticipation du refactoring basé sur les clones. Il permet d'analyser le code source Java, et de détecter les codes dupliqués, les bogues, les boucles vides, les expressions complexes et les codes morts dans un programme. Il vient avec une série d'outils complémentaires comme CPD[2]. PMD permet la détection de clones en utilisant les approches de jetons. Il se configure à l'aide d'un fichier XML ; fichier contenant un ensemble de règles pour faciliter la détection des clones.

[1] http://www.pmd.sourceforge.net
[2] Copie/Paste detector

Pour définir une règle, l'ensemble des données suivantes doit être défini :

- **Le nom** : le nom de la règle (exemple : useSingleton).
- **La classe** : la classe Java implémentant le contrôle de la règle.
- **Le message** : le message associé à une infraction à la règle.
- **La description** : comporte le détail de la règle.
- **Une priorité** : allant de 1 à 5 pour qualifier l'importance de l'infraction.

Avec PMD, chaque portion de code copiée au moins une fois est considérée comme une occurrence.

- BAUHAUS[3] est un autre outil qui analyse principalement le comportement du programme en se basant sur l'approche AST. Il peut être utilisé à tous les différents niveaux d'analyse du programme. Par exemple, la récupération d'architecture logicielle en se basant sur l'identification de composantes réutilisables et l'estimation de l'impact de changement. Parmi ses fonctionnalités, l'identification de clones dans un code source, l'extraction des mesures de qualité et l'identification automatique des erreurs de programmation. Cet outil cherche plusieurs types de clones.

- CCFinder est un détecteur de clones implémenté en C++ [2]. Il est compatible avec différents langages de programmation comme C, C++, Java et COBOL. Il permet de détecter les différents types de clones (I, II, III, IV) [24]. CCfinderX est une version plus récente de CCfinder, totalement remodelée [2] avec les caractéristiques suivantes:
  - ✓ Le prétraitement est basé sur les AST.
  - ✓ Les utilisateurs peuvent adapter l'outil à des langages où à des dialectes de programmation différents.
  - ✓ Analyse en utilisant différentes métriques.

[3] http://bauhaus-stuttgart.de

- CodePro de Google est un outil multitâches permettant d'analyser le code source Java, de calculer des métriques, de générer les tests unitaires JUnit, d'analyser les dépendances et de détecter les codes dupliqués. Cet outil offre plusieurs types de recherche sur les clones [7].

Le tableau 1 représente les outils cités précédemment, les langages supportés et les algorithmes utilisés :

Outil	Langage supporté	Algorithme
PMD	Java,C, C++,Jsp,Php, Ruby, Fortran	Karp-Rabin's
Bauhaus	C, C++, C#, Java, Ada	Baxter's variation on AST
CodePro	Java	//

Tableau 1: Les outils de détection de clones.

Les outils PMD, Bauhaus, et CodePro sont classés parmi les meilleurs outils de détection de clones [7].

2.3.5 Le processus de refactoring

Selon Mens et al. [25], pour pouvoir appliquer le refactoring sur le code source de la meilleure façon et pour ne pas avoir des effets de bord, il faut passer par les étapes suivantes : (1) Identification des parties à remanier ; (2) Détermination du type de refactoring; (3) Validation du refactoring; (4) Évaluation de l'impact du refactoring sur la qualité ; (5) Synchronisation du code source, élaboration de la documentation et des tests.

(1) **Identification des parties à remanier** : avant de procéder à l'identification des parties à remanier, il faut d'abord mettre en place une infrastructure pour gérer les changements (comme la gestion de configuration et la gestion des tests) que le

logiciel va subir pendant son évolution. La détection des bad smells [26] dans le code est considérée parmi les meilleures approches d'anticipation du refactoring sachant que la présence de ces derniers dans le code a un impact négatif sur la maintenabilité. Ducasse et al. [27] ont proposé une approche d'anticipation du refactoring basé sur la détection des clones dans le code source, ainsi que les techniques de refactoring qui peuvent les éliminer et résoudre cette problématique.

Une autre alternative proposée afin d'identifier les zones à remanier consiste en une analyse à la fois quantitative et qualitative basée sur les métriques [3]. L'analyse quantitative consiste à calculer les différentes métriques relatives au logiciel pour identifier les zones problématiques. L'analyse qualitative, se base principalement sur les résultats trouvés précédemment. En utilisant la liste des zones à remanier, elle évalue chacune d'elles en calculant le coût, le gain en termes de maintenabilité, évolutivité et risques. Simon et al. [28] proposent d'utiliser des métriques orientées objet afin d'anticiper le refactoring en identifiant les bad smells, ainsi que les méthodes de refactoring adéquates.

(2) **Détermination du type de refactoring** : La méthode de refactoring qui va être appliquée doit respecter un seul critère. Ce dernier est de s'assurer que les modifications que le code source va subir n'altèrent pas le fonctionnement (comportement) du logiciel [25, 26]. De ce fait, avant toute modification du code, il est nécessaire de mettre au point une batterie de tests sur le code original (si elle n'existe pas déjà). Ils seront utiles pour la phase de validation. Lorsque les tests de non-régression sont validés sur le code original, on peut procéder au refactoring. Pour mieux valider cette phase, il est recommandé de réaliser une validation unitaire sur chaque opération de refactoring.

(3) **Validation de refactoring** : Cette étape [25] est le fruit du processus global. Elle se base principalement sur le critère qui vérifie si le respect de la contrainte de non-

modification du logiciel était satisfaisant ou non. Pour le faire, elle se base sur la batterie des tests mise en place pour le refactoring. Il est donc important que ces tests soient les plus exhaustifs possible afin de valider le plus de cas de figures possibles.

(4) **Évaluation de l'impact du refactoring sur la qualité** : Les méthodes de refactoring sont classifiées selon les attributs externes de la qualité qu'ils affectent [25]. Ceci nous permet d'améliorer la qualité par l'application de la bonne méthode de refactoring sur le bon fragment de code. Certaines méthodes de refactoring éliminent les clones, d'autres améliorent la réutilisabilité, etc. L'effet peut être estimé à travers l'observation de l'impact au niveau des attributs internes de la qualité (la taille, la complexité, le couplage, etc.). Pour évaluer l'impact du refactoring, il existe plusieurs techniques comme : l'évaluation empirique, les métriques et les tests statistiques. Kataoka et al. [29] proposent des métriques de couplage comme une méthode d'évaluation d'impact de refactoring sur la maintenabilité.

(5) **Synchronisation du code source, documentation et tests** : Cette étape est la dernière dans le processus de refactoring. Elle permet de mettre à jour le code source, la batterie de tests et la documentation après avoir appliqué le refactoring [25].

2.3.6 Différentes techniques de refactoring

Plusieurs techniques ont été proposées dans la littérature [3, 6, 7]. Les plus importantes seront citées en incluant celles relatives aux clones :

Méthode de renommage: ou Rename Method (Figure 1), cette méthode [6] consiste à modifier le nom d'un ou de plusieurs éléments du code. Pour Java, les éléments susceptibles d'être renommés sont les suivants : les packages, les classes ou les interfaces, les méthodes, les attributs de classe, les variables locales, et les paramètres

d'une méthode. Le principe à respecter est que le nom doit être représentatif de l'entité qu'il représente, par exemple en Java le nom d'une classe doit commencer par une majuscule. Ce type de méthode est détectable par PMD ou Checkstyle.

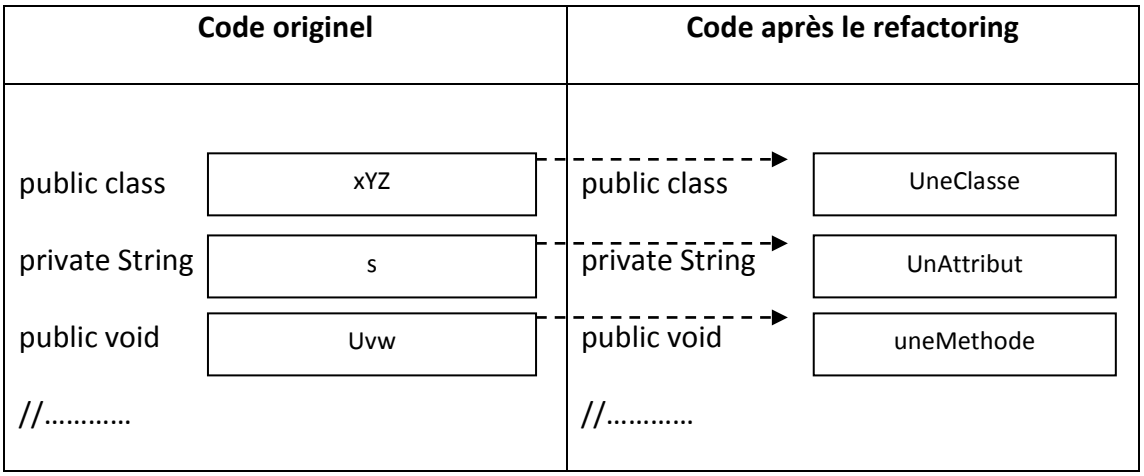

Figure 1. Méthode de renommage [6].

- **Méthode de déplacement**: ou move method, elle permet de créer une nouvelle méthode avec le même corps dans la classe la plus utilisée. L'ancienne méthode pourrait être transformée en une simple délégation [6].

- **Extraction de méthode**: ou Extract Method (Figure 2), cette méthode [6, 7] consiste à sélectionner un bloc de code consistant et à le transformer en une méthode en remplaçant le code déplacé par un appel de méthode. L'extraction d'une méthode est utile principalement dans les deux cas suivants: la méthode contenant le bloc de code à extraire est complexe où longue, où dans le cas où le bloc de code constitue un élément réutilisable au sein de la méthode source (ou au niveau de la classe).

Code originel	Code après le refactoring

Public class UneClass {
Public void uneMethode () {
blocDeCode1 ;
blocDeCode2 ;
}}
	Public class UneClass {
	Private void methode2(){
	blocDeCode2 ;}
	Public void uneMethode(){
	blocDeCode1 ;
	methode2() ; }}

Figure 2. Méthode d'extraction [6].

- **Méthode de remplacement** : ou Replace Method, cette méthode [6] permet de chercher les redondances de codes similaires par rapport à une méthode sélectionnée et les remplacer ensuite par des appels à cette même méthode.
- **Pull Up Method**: Elle permet de déplacer les méthodes qui ont le même corps de la classe fille à la classe mère. Cette méthode permet d'éliminer les comportements en double. Bien que deux méthodes dupliquées puissent fonctionner très bien, cela risque d'être problématique dans le futur [6].

Extract Method, Replace Method et Pull up Method sont considérées aussi comme des méthodes de refactoring de clones.

2.4 Refactoring basé sur les clones, qualité et travaux connexes

Cette section est consacrée à la présentation des principaux travaux de recherche portant sur le refactoring (surtout ceux basés sur les clones), ainsi que leur impact sur les attributs de qualité, la testabilité en particulier.

Gyorgy Hegedus et al. [6] ont travaillé sur l'effet du refactoring orienté objet sur la testabilité et autres aspects de la maintenabilité. Ils ont montré que toutes les caractéristiques de qualité sont influencées par la taille du code; en particulier entre les

propriétés du code source (complexité et taille) et l'effort de test unitaire avec les différents attributs de la maintenabilité. Cependant, une corrélation négative entre la changeabilité, la testabilité et la stabilité avec la complexité du code source est démontrée (les codes complexes sont difficiles à tester). Par ailleurs, la présence du code dupliqué dans le code a une influence directe sur l'analysabilité et la changeabilité. Dans une deuxième partie, ils ont résumé les impacts de plusieurs méthodes de refactoring sur les propriétés du code source, les clones et l'effort de test. Premièrement, la plupart des méthodes de refactoring dans les mêmes catégories ont presque les mêmes effets sur les attributs internes de la qualité. Les deux méthodes de refactoring Pull up et Extract superclass sont les seules ayant un impact sur les clones, ou leur objectif principal est d'éliminer les comportements similaires, et par conséquent, ils diminuent le nombre de clones (un impact positif), le nombre de lignes de code ainsi que l'effort de test. La méthode d'extraction a une relation directe avec le couplage, ou il augmente suite à un refactoring. Pour la méthode Inline, elle a une corrélation positive avec la complexité et la taille, et une corrélation négative avec le couplage et l'effort de test unitaire.

Dans une dernière étape, ils ont essayé de fusionner les résultats trouvés auparavant afin de prédire les impacts du refactoring sur la maintenabilité et ses caractéristiques. L'effet d'EM (extract method) sur la testabilité repose sur de nombreux facteurs. Par exemple, EM diminue la complexité provoquant un impact positif sur la testabilité. Donc, EM a un effet positif sur la testabilité causée par la diminution de la complexité. Les résultats obtenus dans cette étude ont montré que le refactoring selon le cas et selon la méthode utilisée peut avoir des impacts différents sur les attributs internes de la qualité.

L'étude de Francesca Arcelli Fontana et al. [7] vise principalement à présenter des expériences faites avec trois outils de détection de clones cités auparavant (comme PMD et Bauhaus) en décrivant les différences entre ces derniers. Ces outils ont été appliqués sur cinq versions de deux logiciels open source (Ant et Ganttproject) suite à une analyse

d'évolution de clones sur leurs différentes versions. Ensuite, une étape d'anticipation de refactoring de clones en se basant sur les métriques a été réalisée. IntelliJ IDEA et MetricsReloades ont été choisis respectivement comme outil de refactoring et de calcul des métriques. Le constat tiré est que le code dupliqué augmente en parallèle avec l'augmentation de la taille du système, car le nombre de lignes du code dupliqué a augmenté pour chaque nouvelle version analysée. Les différences principales entre les trois outils se résument dans le nombre de clones trouvés et la grandeur de ces fragments. Ces différences sont liées aux différents algorithmes utilisés pour détecter les codes dupliqués. Pour évaluer l'impact du refactoring orienté objet sur les clones, une liste de métriques a été utilisée : LOC (lignes de code), NOM (nombre de méthodes), CC (Complexité cyclomatique), CF (le facteur de couplage), et DMS (la distance depuis la séquence Main) comme métriques de paquets. LCOM (Lack of cohesion of methods), WMC (Weighted Method per Class) et RFC (response for a class) ont été sélectionnées pour évaluer l'impact au niveau des classes. Ensuite, les métriques ont été calculées avant et après le refactoring sur les versions sélectionnées.

L'évaluation d'impact se résume dans la comparaison des valeurs de métriques avant et après le refactoring (basé sur les clones). Un impact positif au niveau de la métrique LOC dans les deux systèmes a été déduit. Ce dernier s'exprime par une diminution du nombre de lignes de code après l'application du refactoring. Par rapport à la métrique NOM, c'est pareil pour Ant, mais une stabilité a été observée pour Ganttproject. Ces résultats se résument par le fait que la plupart des méthodes de refactoring se basent sur la suppression ou l'extraction des méthodes. Un autre impact positif sur la métrique CC après l'application de la méthode de remplacement dans les deux systèmes. Un impact positif pour la métrique de couplage CF a été constaté avec Ant et une stabilité avec Ganttproject. Un impact positif a été observé pour les trois métriques de classes dans les deux systèmes.

2.5 Conclusion

Dans ce chapitre, nous avons vu les clones comme une mauvaise pratique de programmation ainsi que les conséquences de leur utilisation. La présence de clones ou les bad smells dans le code source représente un facteur important dans la dégradation de la qualité. Ceci provoque des difficultés dans l'entretien et l'augmentation de la taille du système. Nous avons vu aussi le refactoring comme activité d'ingénierie logicielle qui consiste à améliorer le code source sans altérer le comportement ainsi que les bénéfices apportés par cette activité, les différentes approches, outils, processus et techniques de refactoring. Finalement, une revue de la littérature a été faite sur les principaux travaux portant sur le refactoring des clones.

Dans le prochain chapitre, nous allons voir la qualité logicielle selon la norme ISO/IEC 9126. L'objectif sera l'élaboration d'un modèle pour évaluer la testabilité en utilisant l'approche GQM.

3.1 Introduction

D'une manière générale, un logiciel de qualité est considéré comme un logiciel capable de répondre de façon satisfaisante aux attentes des clients [30]. La qualité est déterminée comme un ensemble de règles et de principes à suivre au cours du processus de développement [9, 31,32]. Les modèles les plus connus actuellement sont des modèles hiérarchiques qui recensent les principes de la qualité. Ceci en partant des exigences globales et des principes, pour descendre vers les métriques qui permettent de les mesurer. Ces dernières impliquent que la mesure de la qualité ne peut débuter qu'une fois le modèle totalement spécifié. Les premiers résultats ne peuvent être obtenus qu'une fois la collecte de données est suffisante [33].

Dans ce chapitre, nous allons voir la qualité selon la norme ISO/IEC 9126. Le but est d'élaborer un modèle pour évaluer la testabilité après l'application du refactoring basé sur les clones.

3.2 Les indicateurs de qualité selon ISO/IEC 9126

L'International Standards Organisation (ISO) a publié un consensus sur les terminologies des caractéristiques de qualité pour l'évaluation des produits logiciels. ISO / IEC IS 9126 : Software Product Evaluation - Quality Characteristics and Guidelines a été développé par ISO entre 2001 et 2004. Il comporte les standards internationaux et les rapports techniques, comme IS 9126-1 pour les modèles de qualité (Quality Model), TR 9126 - 2 et - 3 pour les métriques internes et externes (External and Internal Metrics) et TR 9126 - 4 pour les métriques de qualité en cours d'utilisation (Quality in Use Metrics) [31].

Deux types de qualités ont été déduits, les qualités internes et externes et les qualités en cours d'utilisation [34]. Les qualités internes de logiciels sont celles qui peuvent être mesurées sans exécution. Les qualités externes sont les qualités de logiciels qui peuvent être mesurées lors de l'exécution. Les qualités en cours d'utilisation, sont les qualités de logiciels qui peuvent être mesurées pendant le lancement des opérations par un utilisateur, ou lors de la maintenance.

La norme ISO semble être une bonne approche pour déterminer la qualité d'un logiciel dans son ensemble et fournir une vue globale satisfaisante. Cependant, la norme ne précise pas de manière explicite comment mesurer les caractéristiques de qualité définies et comment les relier aux métriques de bas niveau.

Les modèles de qualité ISO permettent d'identifier les caractéristiques de qualité selon des cadres de référence et de terminologie standardisés qui facilitent la communication concernant la qualité [30].

ISO/IEC 9126-1 [35] est une norme parmi les normes standards internationales. Elle vise à évaluer la qualité. Cette norme évolue pour être enrichie et intégrée dans la norme SquaRE (Software product Quality Requirement and Evaluation). Elle est composée de six caractéristiques générales qui définissent la qualité globale d'une application [36]: la capacité fonctionnelle, la fiabilité, l'efficacité, la maintenabilité, la facilité d'usage, et la portabilité [36]:

a. **La capacité fonctionnelle :** c'est le degré pour lequel un produit logiciel répond aux besoins implicites et explicites des usagers. Elle se résume par des notions comme la pertinence, l'exactitude, l'interopérabilité, la sécurité et la conformité.

b. **La fiabilité :** c'est le degré pour lequel un produit logiciel peut maintenir un niveau de performance dans des conditions précises (tolérance aux pannes, conditions de remise en service, maturité, etc.).

c. **L'efficacité :** cette caractéristique se résume par le rapport entre le service produit par un produit logiciel et l'effort pour le faire fonctionner.

d. **La maintenabilité :** c'est l'évaluation d'un produit logiciel dans le cas d'un nouveau besoin comme une modification, une correction, une amélioration ou une adaptation à un nouvel environnement.

e. **La facilité d'usage :** cette caractéristique comporte un certain nombre d'attributs permettant de définir un degré représentant l'effort nécessaire à un utilisateur potentiel pour utiliser le système : exploitation ou encore attraction, et facilité de compréhension et d'apprentissage.

f. **La portabilité :** cette caractéristique se résume dans la facilité de transfert d'une composante logicielle d'un environnement à un autre.

Toutes ces caractéristiques sont considérées comme des caractéristiques de qualité soit internes ou externes. Chacune d'elles est décomposée en sous caractéristiques (27 sous caractéristiques).

3.3 La Maintenabilité comme un attribut de qualité

Le tableau 2 illustre la hiérarchisation des notions de qualité d'un produit logiciel interne et externe selon la norme ISO 9126. Cette répartition est décomposée en six principales caractéristiques subdivisées en vingt-sept sous caractéristiques [30]. Dans le contexte de notre travail, nous allons focaliser sur la maintenabilité et ses caractéristiques.

Qualité interne et externe					
La capacité fonctionnelle	**La fiabilité**	**L'efficacité**	**La maintenabilité**	**La facilite d'usage**	**La portabilité**
pertinence précision interopérabilité sécurité	maturité tolérance aux fautes recouvrabilitée	comportement dans le temps ressource utilisation	**analysabilité** **changeabilité** **stabilité** **testabilité**	compréhensibilité apprenabilitée opérabilité attractivité	adaptabilité installabilitée co- existence remplaçabilitée

Tableau 2. Qualités interne et externe [30].

La maintenabilité est l'une des six caractéristiques de qualité selon le modèle de qualité ISO 9126 - 1, avec l'analysabilité, la changeabilité, la stabilité et la testabilité comme sous caractéristiques [30, 31] :

- **Analysabilité** : c'est la facilité pour diagnostiquer la défaillance d'un système ou pour identifier les parties qui ont besoin d'être modifiées.

- **Changeabilité** : c'est la facilité pour faire changer un système.

- **Stabilité** : c'est le degré de facilité pour garder un système dans un état consistant durant une modification.

- **Testabilité** : c'est le degré de facilité pour tester un système.

Pour notre travail, la qualité sera examinée essentiellement d'un point de vue de la maintenabilité, sous la perspective particulière de la testabilité.

3.4 La testabilité comme sous-caractéristique de la maintenabilité

La testabilité est la facilité de tester et de détecter les fautes d'un produit logiciel [30, 31]. Elle représente un enjeu primordial, et un domaine de recherche très actif en raison de son importance dans le processus de développement d'un logiciel orienté objet [37]. Elle permet d'augmenter la fiabilité et surtout de réduire les coûts des tests d'un produit logiciel [25]. Elle doit être définie selon le contexte et les facteurs qui l'influencent [38].

Mouchawrab et al. [39] ont défini la relation entre la testabilité et quelques attributs classés selon les activités qui l'influencent dans un Framework générique. Jungmayr [38] fait l'évaluation de la testabilité en se basant sur les dépendances. Il part du principe que si le nombre de dépendances dans un produit logiciel augmente, la testabilité du produit logiciel baisse (un impact négatif). Il définit aussi un ensemble de métriques pour évaluer la testabilité à travers les dépendances. L'ensemble de métriques définies a permis à l'auteur de proposer une technique d'identification de dépendances et des pistes de refactoring possibles. Blinder [40] a proposé un modèle déterminant la testabilité avec les attributs qui l'influencent. La testabilité comme attribut externe de la qualité, peut être mesurée en utilisant un ou plusieurs attributs internes de la qualité comme la taille, la complexité, etc. Les attributs internes les plus utilisés dans la littérature sont présentés dans le tableau 3 :

Attributs / Études	Complexité	Taille	Couplage	Cohésion	Héritage	Test unitaire
Bruntink and van Deursen [41]	x	x	x			
Mouchawrab et al. [42]		x	x	x	x	x
Bruntink et Van Deursen [41]	x	x	x	x		
Heitlager et al. [30]	x	x				x
Hegedus et al. [53]	x	x	x			x
Kout et al. [45]						x
Badri et al. [46]	x	x	x	x	x	
Zhou et al. [47]	x	x		x	x	x

Tableau 3. Attributs internes représentatifs de la testabilité.

Pour pouvoir mesurer les attributs internes présentées à la section précédente, il faut les relier à des métriques de code source. Il existe un nombre considérable de métriques et certaines d'entre elles ne sont pas calculées de la même manière. On peut toutefois classer les métriques en deux grandes catégories : (1) Les métriques primitives, qui sont les métriques mesurant des propriétés de base du code source, telles que le nombre de lignes de code et la complexité cyclomatique. (2) Le deuxième type comporte les métriques de design, ce sont les métriques qui déterminent si le code source respecte les principes de conception préalablement définis. Il s'agit de métriques telles que les métriques de couplage, de cohésion, et les métriques d'architecture de paquets (comme les métriques de couplage efférent de Martin) [48].

Le tableau 4 présente les métriques les plus utilisées dans la littérature pour mesurer les différents attributs internes.

Attributs	Métriques	Étude source
Taille	LOC	[30, 41, 46, 49]
	LOCC	[41]
	Size1, Size2	[50, 51]
	TNOS (total number of statements)	[52]
	NOM	[41, 49, 50]
	NOO, NOA, NI (nombre d'instructions), Number of instructions, Number of overloaded operations.	[42, 45, 46, 49]
	NOF	[41, 49]
	SLOC, Stmt, SMTSEXE, NAIMP, NMIMP, NMpub, NMNpub.	[47]
	NumPara, NOS (nb of statement), NFMA (Number of Foreign Methods Accessed).	[53]
	LLOC (logical ligne of code), NMD (nombre de méthodes déclarées), NAD (nombre d'attributs déclarés), Number of declared Operator.	[54]
Complexité	RFC	[41, 46, 49, 55]
	WMPC1, CIE, CCE	[43, 46, 47, 49, 50, 51, 54]
	Cyclomatic complexity per unit	[43]
	CDE	[30,54]
	WOC	[56]
Test unitaire	TRFC, TNOO, TASSERT, TWMPC, TLOC	[45, 47]
	LOCC, NOTC	[41]
	Unit test coverage, Number of assert	[30]
Couplage	CBO	[42, 45, 46, 54, 56]
	FOUT	[41, 49, 50]
	NICMIC, NIMMIC, NIIC (NIIC=NICMIC+NIMMIC)	[52]
	MPC (Message Passing Coupling)	[51]
	DAC	[50]
	DCMED, DCAEC	[54]
	NOI (Number of Outgoing Invocations)	[53]
	CBO_IUB(is used by)	[57]
	CA (Afferent coupling), CE (Efferent coupling)	[55]
	RFC	[42]
	Fan-in	[30]
	CM, CHC	[58]
	MIC	[59]
Cohésion	LCOM	[41, 49, 50]
	LCOM1, LCOM2, LCOM3, LCOM5	[42, 45, 46, 50, 54]
	ICH	[46]
	Co, Co(h), CAMC, CAMCs, iCAMCs, SNHDs, iSNHDs	[62]
	(N)(P)(O)RCI	[60]
	Loose Class Cohesion (LCC)	[58]
	TCC	[63]
Héritage	OVO, NMA, SPD, DPD	[32]
	DIT	[42, 45, 46, 50, 54]
	NOC	[42, 51]
	ABS	[54]
	Number of inherited operation, Number of inherited attributes, Number of inherited association, Number of inherited dependency, Number of inherited interface, Width of inheritance hierarchy	[42]
Duplication	Blocks overs 6 lines	[30]

Tableau 4. Les métriques pour mesurer les attributs de la testabilité.

3.5 Cadre d'évaluation final de la Maintenabilité – Testabilité en utilisant GQM

L'approche GQM (Goal - Quality- Metrics) est une alternative très courante pour résoudre les problèmes d'évaluation de différents attributs externes de la qualité. Selon Roche et al. [30], elle permet de construire des cadres d'évaluation pour la qualité. Elle offre une souplesse dans l'identification et la validation des métriques logicielles pour des environnements particuliers. C'est une approche dirigée par des axes superposés : Objectif, Questions, Métriques. Elle a été développée par Basili et ses coauteurs dans Basili et al. [64].

Les questions sont dérivées des objectifs afin de les associer aux métriques qui sont les attributs mesurables répondants aux questions et qui représentent le dernier niveau de l'échelle [65]. Cette approche propose donc un modèle de trois niveaux afin de résoudre les problématiques posées [65] :

- **Un niveau conceptuel (buts)** : pour préciser les objectifs à atteindre. Les objectifs peuvent être décomposés en d'autres buts afin d'obtenir une meilleure description.

    - **But** : Notre objectif est d'évaluer l'impact du refactoring de clones sur un attribut externe de la qualité : la testabilité.

- **Un niveau opérationnel (questions)** : les questions permettent de définir l'objet de la mesure et le point de vue de l'évaluation.

    - **Question 1**: Comment le refactoring orienté objet basé sur les clones affecte les différents attributs internes de la testabilité ?

- **Un niveau quantitatif (Métriques)** : les métriques représentent la mesure quantitative dérivée à partir des questions pour l'évaluation des objectifs associés.

- **Métriques** : le refactoring orienté objet basé sur les clones affecte la testabilité en affectant ses différents attributs internes cités dans le tableau 3 (la taille, la complexité, le couplage, l'héritage, la cohésion, les tests unitaires). La réponse à notre question va être donnée par les métriques que nous avons choisies afin de mesurer les attributs internes ayant une influence sur la testabilité (résumé dans le tableau 5).

Selon les études que nous avons analysées dans la section 2.4, chaque technique de refactoring a une influence sur au moins deux attributs internes de la qualité que nous avons choisis pour évaluer la testabilité.

TESTABILITE					
Taille	**complexité**	**couplage**	**héritage**	**cohésion**	**Test unitaire**
LOC NOA NOO	CC RFC WMPC1 WOC	CBO CE DAC CM CHC FAN-IN FO MIC	DIT NOC	LCOM1 LCOM3 TCC	TLOC TNOO TWMPC1

Tableau 5. Le cadre final d'évaluation de la testabilité des classes.

Nous résumons les métriques sélectionnées pour l'étude empirique. Plusieurs études dans la littérature ont montré que ces dernières ont une relation significative avec les attributs internes de la qualité et elles ont été utilisées dans plusieurs études empiriques (résumé dans les tableaux 3 et 4). Nous avons sélectionné vingt métriques de code source et trois des classes de test. Les métriques choisies vont être citées et définies selon leur appartenance (métriques de couplage, métriques de complexité, etc.).

Métriques de code source

Le couplage

- **CBO** : ou Coupling between Objects, mesure de dépendance qui compte le nombre de classes auxquelles une classe est couplée [1].
- **CE** : ou Efferent Coupling, cette métrique mesure le nombre de classes dont une classe donnée dépend. Elle est considérée comme une mesure de dépendance (sortante, Efferent = outgoing) de paquets [2].
- **DAC** : ou Data Abstraction Coupling, cette métrique mesure la complexité du couplage causée par les types de données abstraits (TDAs). Li et Henry définissent (DAC) comme: DAC = nombre de TDAs définie dans une classe [3].
- **CM:** ou Changing Methods, cette métrique mesure le nombre de méthodes distinctes dans le système qui seraient potentiellement touchées par les changements opérés dans la classe mesurée [4].
- **FAN-IN :** cette métrique mesure le nombre d'appels entrants pour une classe donnée [6].
- **ChC:** ou Changing class, cette métrique mesure le nombre de classes clientes qui vont être changées suite à un changement dans une classe donnée [4].
- **FO**: ou FanOut, cette métrique mesure le nombre d'appels sortants d'une classe donnée [6].
- **MIC**: ou Method Invocation Coupling, cette métrique indique le nombre relatif de classes auxquelles une classe donnée envoie des messages [5].

La complexité

- **CC :** ou Cyclomatic complexity, cette métrique mesure le nombre de chemins linéairement indépendants à travers le code source en utilisant le graphe qui décrit le flux de contrôle du programme [12].
- **RFC**: ou Response for a class, c'est le nombre de méthodes dans l'ensemble de toutes les méthodes qui peuvent être invoquées en réponse à un message envoyé à un objet d'une classe [1].

- **WMPC1**: ou Weighted Methods Per Class1, cette métrique mesure la somme des complexités de toutes les méthodes d'une classe, ou chaque méthode est pondérée par sa complexité cyclomatique [66].
- **WOC**: ou Weight of Class, cette métrique est définie par le nombre de méthodes publiques fonctionnelles dans une classe sur le nombre total des membres publiques [67].

La cohésion

- **LCOM1**: cette mesure a été introduite dans la suite de métriques de Chidamber & Kemerer [68, 69]. Elle représente le nombre de méthodes prises deux à deux (paires de méthodes) ne partageant pas de variables d'instances dans une classe. Plus le nombre de paires de méthodes partageants des variables d'instances dans la classe est faible, moins la classe est cohésive. Si cette valeur est négative, LCOM 1 est fixée (selon la définition des auteurs) à zéro:
  - LCOM1 =0 indiques que la classe est cohésive.
  - LCOM1 > 0 indiques que la classe est moins cohésive.
- **LCOM3:** ou Lack of cohesion of Methods 3, LCOM3 est une métrique qui appartient à un ensemble de métriques de cohésion qui mesurent la dissimilarité dans une classe donnée [36] :

LCOM3= (m – Sum (m.A) / a) / m-1 ……. (1)

M: le nombre de méthodes dans une classe.

A: le nombre de variables dans une classe.

m.A: le nombre de méthodes qui accèdent à une variable.

Sum (m.A): le nombre de méthodes qui accèdent à plusieurs variables.

a: variables qui peuvent être partagées ou non.

- **TCC**: Tight Class Cohesion, comme métrique de cohésion normalisée, elle est définie comme le nombre relatif de méthodes directement connectées. Deux

méthodes sont directement connectées si elles accèdent à une variable commune de la classe [61]. Une valeur élevée de la cohésion reflète la bonne modélisation, donc elle est toujours souhaitée.

La taille

- **LOC**: ou Lines of code, cette métrique compte le nombre de lignes de code d'une classe donnée [70].
- **NOA**: ou Number of Attributs, cette métrique compte le nombre total d'attributs définis dans une classe [61].
- **NOO:** ou Number of Operations, cette métrique compte le nombre de méthodes dans une classe [70].

L'héritage

- **DIT**: ou Depth of Inheritance Tree, elle est définie comme la longueur maximale du nœud à la racine de l'arbre. Elle est mesurée par le nombre de classes ancêtres [71].
- **NOC :** ou Number of Children, cette métrique mesure le nombre de descendants immédiats d'une classe donnée [72].

Métriques de classes tests

- **TLOC :** cette métrique mesure le nombre de lignes de code d'une classe de test donnée.
- **TNOO :** cette métrique mesure le nombre de méthodes d'une classe de test donnée.
- **TWMPC1 :** cette métrique mesure la somme des complexités de toutes les méthodes d'une classe de test donnée.

Ces trois dernières ont été utilisées afin de quantifier l'effort de test requis (la taille, le nombre de méthodes et la complexité de la classe de test) [45, 47].

3.6 Conclusion

Dans ce chapitre, nous nous sommes penchés sur la qualité du logiciel afin de définir notre cadre conceptuel de la testabilité. Ce dernier va nous permettre d'évaluer l'impact du refactoring orienté objet (basé sur les clones) sur la testabilité des classes. Selon la littérature, la taille, la complexité, le couplage, l'héritage, la cohésion et les tests unitaires sont les caractéristiques représentant le mieux la testabilité.

L'étude empirique que nous avons menée est présentée dans le prochain chapitre. Nous allons présenter les résultats obtenus en termes d'impact de refactoring des clones sur la testabilité des classes. Cette dernière est mesurée dans les systèmes en utilisant les classes de code source et les classes test correspondantes (avant et après le refactoring). Les tests statistiques sur les classes avant et après le refactoring permettent d'observer l'impact sur les attributs internes de la qualité ainsi que sur la testabilité (du point de vue des tests unitaires). La collecte de données, les tests statistiques, les résultats et les interprétations sont détaillés dans ce chapitre (chapitre 4).

4.1 Introduction

Ce chapitre résume la partie empirique de notre travail portant sur l'évaluation de l'impact du refactoring des clones sur la qualité des systèmes orientés objet. Tel que mentionné précédemment, la qualité sera examinée essentiellement d'un point de vue de la maintenabilité, sous la perspective particulière de la testabilité. La testabilité, comme caractéristique importante de la maintenabilité, sera abordée du point de vue de l'effort de test unitaire. Une étude empirique a été réalisée en se basant sur des analyses statistiques effectuées sur des données collectées sur deux systèmes implémentés en Java : Ant et Archiva.

4.2 Objectif de l'étude

Pour atteindre notre objectif principal, qui se résume en une étude empirique portant sur l'évaluation de l'impact du refactoring des clones sur la testabilité des systèmes orientés objet, nous focalisons sur les points suivants :

- l'impact du refactoring des clones sur les différents attributs du code source, attributs liés à la testabilité unitaire des classes.

- l'impact du refactoring des clones sur les différents attributs des classes de test.

- la relation entre les variations observées après le refactoring des clones sur les métriques de code source et les métriques de classes de test.

4.3 Environnement et collecte de données

Les deux systèmes sélectionnés pour cette étude sont des systèmes Java, bien connus, sur lesquels le refactoring basé sur les clones a été effectué :

- **Ant**: est un logiciel crée par la fondation Apache. Son nom est un acronyme pour « Another Neat Tool ». Ce dernier vise à automatiser principalement la construction de projets en langage Java, et d'autres opérations qui se répètent durant le développement de logiciel comme : la génération de pages HTML de documents (javadoc), la compilation, la génération de rapports, etc.

- **Archiva** : Apache Archiva est un logiciel qui offre plusieurs fonctionnalités telles que la gestion d'accès de sécurité, l'indexation et les rapports d'utilisation. Archiva est un outil de gestion référentiel qui permet de s'occuper des dépôts. Il est considéré comme le compagnon idéal des outils de construction comme : Maven et Ant.

Pour comparer les deux implémentations des deux systèmes (avant et après refactoring) que nous avons choisis, nous prenons des classes des deux systèmes avant et après le refactoring de clones selon le processus décrit par la figure 3.

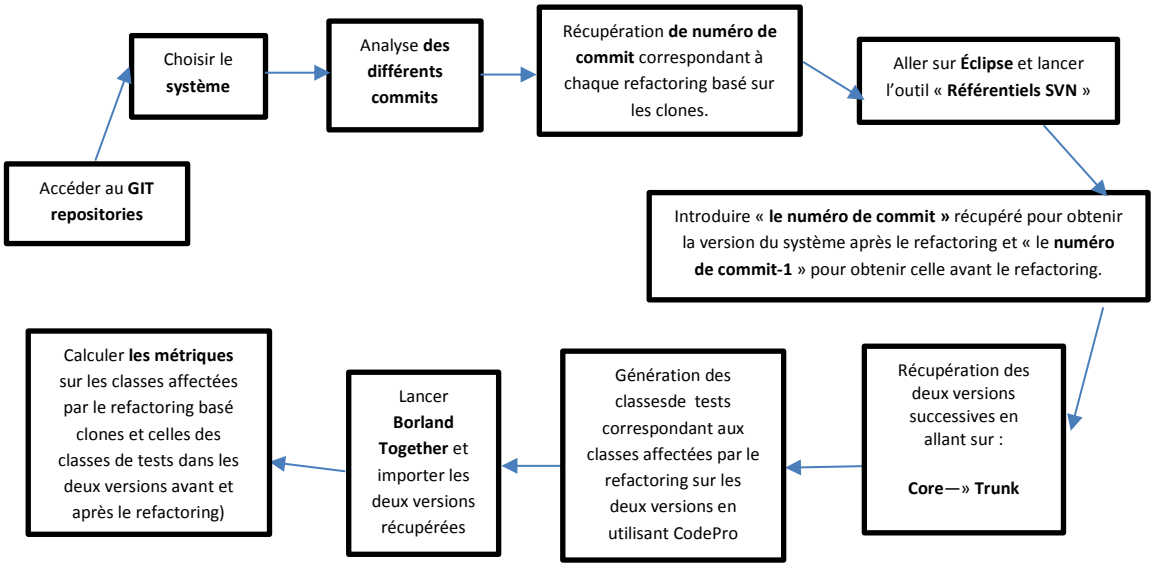

Figure 3. Processus de collecte de données.

Système	Versions	Nombre de classes touchées par le refactoring	Type de refactoring
Ant	1.5 1.6 1.7	42 cas	Pull up method : 9 cas Replace method : 33 cas
Archiva	1.0 1.0 alpha 1 1.0 alpha 2 1.2 1.4	40 cas	Pull up Method : 4 cas Replace Method : 13 cas Move Method : 23 cas

Tableau 6. Les détails sur les systèmes choisis.

42 cas de refactoring basé clones sont identifiés (dans le tableau 6) sur trois versions différentes pour Ant (42 classes touchées) ainsi que 40 cas sur cinq versions pour Archiva (40 classes touchées). Pull up, Replace et Move methods sont les méthodes de refactoring utilisées afin d'éliminer les clones dans les deux systèmes.

Le processus de collecte de données est bien détaillé dans la figure 3, en passant par le Git Repositories. Git est un outil de gestion de versions permettant aux développeurs de travailler en parallèle sur le même projet, et fusionner les commits des modifications apportées avec le projet initial. Apache met tous les commits des différents projets dans Git Repositories (http://ant.apache.org/git.html) où on trouve plusieurs informations, le type de la modification, l'auteur, la classe avant et après la modification, etc.

Après avoir accédé au Git repositories, le système en question sera sélectionné afin d'obtenir ses commits. Une analyse de chaque commit est faite afin d'identifier ceux qui concernent le refactoring basé sur les clones. Le numéro de commit permet de récupérer la version du système correspondante au refactoring et la classe avant la modification en utilisant le référentiels SVN sur Éclipse.

La récupération des versions avant et après le refactoring permet le passage à l'étape de création des classes de test correspondantes aux classes touchées par le refactoring. Cette étape a été réalisée en utilisant CodePro[4]. Ce dernier a été utilisé comme outil pour générer, pour chaque classe sélectionnée, la classe de test correspondante.

La dernière étape du processus de collecte de données consiste à calculer les métriques du code source pour les classes affectées par le refactoring avant et après l'opération de refactoring en utilisant l'outil Borland Together[5]. Les métriques de test pour les classes de test correspondantes aux classes affectées par le refactoring sont également calculées.

4.4 Résultats et interprétations

4.4.1 Statistiques descriptives

Les tableaux suivants (tableaux 7, 8, 9 et 10) listent les statistiques descriptives pour les métriques de code source et celles des classes de test pour ANT avant et après le refactoring basé sur les clones.

[4] https://developers.google.com/java-dev-tools/CodePro/
[5] http://www.borland.com/

Attributs	Statistique	Nb. d'observations	Nb. de valeurs manquantes	Minimum	Maximum	Moyenne	Coefficient de variation
Couplage	**CBO**	42	0	2	38	8,833	0,935
	CE	42	0	2	34	9,429	0,81
	DAC	42	0	0	8	1,357	1,418
	CM	42	0	0	684	16,905	6,165
	FAN-In	42	0	0	271	6,5	6,355
	ChC	42	0	0	271	7,524	5,469
	FO	42	0	0	26	6,548	1,044
	MIC	42	0	0	250	73,238	0,909
Complexité	**CC**	42	0	2	178	29,524	1,02
	RFC	42	0	16	253	77,095	0,632
	WMPC1	42	0	2	174	27,881	1,017
	WOC	42	0	45	267	130,095	0,422
Cohésion	**LCOM1**	42	1	0	4489	209,683	3,538
	LCOM3	42	2	0	97	73,025	0,409
	TCC	42	0	0	67	10,333	1,363
Taille	**LOC**	42	0	9	778	143,952	0,903
	NOA	42	0	0	20	5,667	0,841
	NOO	42	0	1	98	14,024	1,188
Héritage	**NOC**	42	0	0	2	0,095	4,472
	DIT	42	0	0	6	3,167	0,33

Tableau 7. Statistiques descriptives pour Ant avant le refactoring (métriques de classes).

Statistique	Nb. d'observations	Nb. de valeurs manquantes	Minimum	Maximum	Moyenne	Coefficient de variation
TLOC	42	0	28	2781	381,024	1,3
TNOO	42	0	5	196	31,5	0,991
Twmpc1	42	0	7	198	32,881	0,945

Tableau 8. Statistiques descriptives pour Ant avant le refactoring (métriques de classes de test).

Attributs	Statistique	Nb. d'observations	Nb. de valeurs manquantes	Minimum	Maximum	Moyenne	Coefficient de variation
Couplage	**CBO**	42	0	0	38	8,19	1,022
	CE	42	0	1	33	8,714	0,886
	DAC	42	0	0	8	1,238	1,594
	CM	42	0	0	684	16,881	6,174
	FAN-In	42	0	0	271	6,5	6,355
	ChC	42	0	0	271	7,548	5,452
	FO	42	0	0	26	6	1,191
	MIC	42	0	1	250	60,405	0,92
Complexité	**CC**	42	0	1	177	24,714	1,222
	RFC	42	0	13	253	74,857	0,634
	WMPC1	42	0	1	173	23,262	1,226
	WOC	42	0	0	267	109,881	0,568
Cohésion	**LCOM1**	42	11	0	4497	248,032	3,28
	LCOM3	42	12	0	97	77,867	0,311
	TCC	42	5	0	100	11,432	1,694
Taille	**LOC**	42	0	5	774	122,619	1,081
	NOA	42	0	0	19	4,643	1,073
	NOO	42	0	0	98	12,786	1,276
Héritage	**NOC**	42	0	0	2	0,095	4,472
	DIT	42	0	0	6	3,286	0,327

Tableau 9. Statistiques descriptives pour Ant après le refactoring (métriques de classes).

Statistique	Nb. d'observations	Nb. de valeurs manquantes	Minimum	Maximum	Moyenne	Coefficient de variation
TLOC	42	0	21	2781	343,381	1,423
TNOO	42	0	4	196	26,714	1,175
Twmpc1	42	0	6	198	28,048	1,118

Tableau 10. Statistiques descriptives pour Ant après le refactoring (métriques de classes de test).

Les résultats obtenus pour les métriques de code source et celles des classes de test avant et après le refactoring des clones (pour le système Ant) sont présentés dans les tableaux 7, 8, 9 et 10. Toutes les moyennes pour les métriques de code source ont diminuées après l'application de ces refactorings, sauf pour ChC, DIT, LCOM1, LCOM3 et TCC où les valeurs ont augmenté. Pour FAN-IN et NOC, elles n'ont pas changées.

WMPC1 (+):

- La moyenne avant le refactoring : 27,881
- La moyenne après le refactoring : 23,262

LCOM1 (-):

- La moyenne avant le refactoring : 209,262
- La moyenne après le refactoring : 248,032

NOC (=):

- La moyenne avant le refactoring : 0,095
- La moyenne après le refactoring : 0,095

Les différentes techniques de refactoring utilisées afin d'éliminer les clones dans les classes (pour le système Ant) ont provoqué des impacts positifs (les moyennes ont diminué) sur la plupart des métriques liées aux attributs de code source tel que le couplage, la complexité et la taille, ainsi que des impacts négatifs (les moyennes ont augmenté) sur les métriques de cohésion et d'héritage (DIT) et sans impact sur FAN-IN et NOC. Le tableau 11 résume les résultats trouvés précédemment dans les tableaux 7 et 9 sous forme de signes. La diminution de la moyenne après le refactoring est représentée par un signe (+), sinon un signe (-) est assigné, et s'il n'existe pas d'effets, des (0) sont indiqués.

Attributs	Statistique	Impact
Couplage	CBO	+
	CE	+
	DAC	+
	CM	+
	FAN-In	0
	ChC	-
	FO	+
	MIC	+
Complexité	CC	+
	RFC	+
	WMPC1	+
	WOC	+
Cohésion	LCOM1	-
	LCOM3	-
	TCC	-
Taille	LOC	+
	NOA	+
	NOO	+
	NOC	0
Héritage	DIT	-

Tableau 11. Le résumé des statistiques descriptives pour ANT (métriques de classes).

Les deux tableaux 8 et 10 contiennent les résultats pour les métriques de classes de test. Une baisse remarquable (impact positif) au niveau de toutes les moyennes concernées après l'application du refactoring. Comme pour TLOC, la moyenne avant le refactoring est : 381,024 et la moyenne après le refactoring est : 343,381.

Le tableau 12 résume les résultats les tableaux 8 et 10 sous forme de signes.

Statistique	Impact
TLOC	+
TNOO	+
TWMPC1	+

Tableau 12. Le résumé des statistiques descriptives pour ANT (métriques de classes de test).

Les tableaux 13, 14, 15 et 16 listent les statistiques descriptives pour les métriques de code source et celles des classes de test pour Archiva avant et après le refactoring basé sur les clones.

Attributs	Statistique	Nb. d'observations	Nb. de valeurs manquantes	Minimum	Maximum	Moyenne	Coefficient de variation
Couplage	**CBO**	40	0	0	41	14,825	0,736
	DAC	40	0	0	12	3,05	1,013
	CE	40	0	0	44	15,175	0,794
	CM	40	0	0	4	0,2	3,905
	FAN-In	40	0	0	0	0	
	ChC	40	0	0	4	1,225	0,957
	MIC	40	0	0	200	32,45	1,618
	FO	40	0	0	35	10,425	0,753
Complexité	**CC**	40	0	0	84	18,65	0,859
	RFC	40	0	11	182	54,1	0,694
	WMPC1	40	0	0	84	17,725	0,841
	WOC	40	0	0	400	116,975	0,583
Cohésion	**LCOM1**	40	4	0	1451	88,778	2,775
	LCOM3	40	9	0	100	71,742	0,457
	TCC	40	1	0	100	30,487	1,229
Taille	**LOC**	40	0	11	463	139,375	0,742
	NOA	40	0	0	23	5,05	1,148
	NOO	40	0	1	58	10,3	1,025
Héritage	**DIT**	40	0	0	3	1,675	0,572
	NOC	40	0	0	2	0,05	6,245

Tableau 13. Statistiques descriptives pour Archiva avant le refactoring (métriques de classes).

Statistique	Nb. d'observations	Nb. de valeurs manquantes	Minimum	Maximum	Moyenne	Coefficient de variation
TLOC	40	0	13	857	197,75	0,811
TNOO	40	0	3	48	20,225	0,553
TWMPC1	40	0	3	48	20,25	0,55

Tableau 14. Statistiques descriptives pour Archiva avant le refactoring (métriques de classes de test).

Attributs	Statistique	Nb. d'observations	Nb. de valeurs manquantes	Minimum	Maximum	Moyenne	Coefficient de variation
Couplage	**CBO**	40	0	0	34	11,875	0,848
	DAC	40	0	0	9	2,65	1,049
	CE	40	0	0	36	12,2	0,897
	CM	40	0	0	4	0,2	3,905
	FAN-In	40	0	0	0	0	
	ChC	40	0	0	4	1,05	1,106
	MIC	40	0	0	300	35,5	1,905
	FO	40	0	0	21	8,25	0,834
Complexité	**CC**	40	0	0	81	14,725	0,988
	RFC	40	0	3	186	47,35	0,727
	WMPC1	40	0	0	81	14,45	0,999
	WOC	40	0	0	400	96,125	0,769
Cohésion	**LCOM1**	40	8	0	1289	77,75	2,908
	LCOM3	40	11	0	100	76,828	0,422
	TCC	40	5	0	100	16,457	1,763
Taille	**LOC**	40	0	6	449	114,225	0,877
	NOA	40	0	0	22	4,375	1,173
	NOO	40	0	0	55	8,45	1,156
Héritage	**DIT**	40	0	0	3	1,725	0,534
	NOC	40	0	0	2	0,05	6,245

Tableau 15. Statistiques descriptives pour Archiva après le refactoring (métriques de classes).

Statistique	Nb. d'observations	Nb. de valeurs manquantes	Minimum	Maximum	Moyenne	Coefficient de variation
TLOC	40	0	13	857	163,7	0,975
TNOO	40	0	3	44	17,125	0,657
TWMPC1	40	0	3	44	17,15	0,657

Tableau 16. Statistiques descriptives pour Archiva après le refactoring (métriques de classes de test).

Une amélioration remarquable sur toutes les moyennes de valeurs de métriques de code source sauf pour MIC, LCOM3 et DIT, où il y a une augmentation au niveau des moyennes. CM, FAN-IN et NOC sont stables.

NOO (+):

- La moyenne avant le refactoring : 10.3
- La moyenne après le refactoring : 8.45

LCOM 3 (-):

- La moyenne avant le refactoring : 71.742
- La moyenne après le refactoring : 76.828

CM (=):

- La moyenne avant le refactoring : 0.2
- La moyenne après le refactoring : 0.2

Les différentes techniques de refactoring utilisées afin d'éliminer les clones dans les classes (pour le système Archiva) ont provoqué une amélioration au niveau des métriques liées aux attributs de code source comme le couplage (comme : CBO et DAC), la taille (LOC, NOA et NOO), la complexité (comme : RFC et WMPC1) et la cohésion (LCOM3) sauf pour l'héritage (DIT). CM, FAN-IN et NOC n'ont pas changé. Le tableau 17 résume les résultats trouvés précédemment dans les tableaux 13 et 15 sous forme de signes.

Attributs	Statistique	Impact
Couplage	CBO	+
	DAC	+
	CE	+
	CM	0
	FAN-In	0
	ChC	+
	MIC	-
	FO	+
Complexité	CC	+
	RFC	+
	WMPC1	+
	WOC	+
Cohésion	LCOM1	+
	LCOM3	-
	TCC	+
Taille	LOC	+
	NOA	+
	NOO	+
Héritage	DIT	-
	NOC	0

Tableau 17. Le résumé des statistiques descriptives pour Archiva (métriques de classes).

Les tableaux 14 et 16 présentent les résultats pour les métriques de classes test (pour le système Archiva). Des impacts positifs ont été observés sur toutes les moyennes correspondant aux valeurs des métriques de test après l'application de refactoring. Comme pour TWMPC 1, la moyenne avant le refactoring est : 20.25 et la moyenne après le refactoring est : 17.15.

Le Tableau 18 résume les résultats les tableaux 14 et 16 sous forme de signes.

Statistique	Impact
TLOC	+
TNOO	+
TWMPC1	+

Tableau 18. Le résumé des statistiques descriptives pour Archiva (métriques de classes de test).

La différence des résultats obtenus s'explique par l'utilisation de différentes techniques de refactoring dans les deux systèmes afin d'éliminer les clones existants dans le code source. Les résultats obtenus doivent être validés.

4.4.2 Corrélations entre métriques de code source et métriques de classes test

Dans cette section, nous présentons les différentes corrélations entre les métriques de code source et les métriques de classes de test. Les différentes corrélations ont été observées en testant l'hypothèse suivante :

Hypothèse : il existe une relation significative entre les métriques de code source et celles des classes de test. L'hypothèse nulle est : il n'existe pas une relation significative entre les métriques de code source et celle des classes de test.

Afin de valider l'hypothèse, nous allons analyser les corrélations entre chaque métrique de code source et les métriques de classes de test. Les corrélations de Spearman et Pearson ont été utilisées dans cette étude. Ces techniques seront utilisées pour mesurer la relation entre deux variables. Le coefficient de corrélation prendra une valeur entre -1 et +1.

Une corrélation positive signifie que les deux variables augmentent ensemble et diminuent ensemble. Une corrélation négative signifie que si l'une des variables

augmente l'autre diminue. Une corrélation de +1 ou -1 montre que la relation entre les deux variables est linéaire. Une corrélation proche de 0 signifie qu'il n'y a pas de corrélation linéaire entre les deux variables. La présence de fortes et de faibles corrélations entre les variables, pousse la concentration de l'étude sur certaines métriques et l'exclusion de certaines d'autres. XLSTAT[6] a été utilisé comme outil pour faire notre analyse. Le seuil de signification a été fixé à α=0.05.

Spearman					Pearson		
Variables	**TLOC**	**TNOO**	**Twmpc1**		**TLOC**	**TNOO**	**Twmpc1**
CBO	**0,710**	**0,589**	**0,583**		**0,724**	**0,678**	**0,673**
CE	**0,784**	**0,683**	**0,677**		**0,570**	**0,540**	**0,534**
DAC	**0,516**	**0,335**	**0,334**		**0,744**	**0,637**	**0,634**
FO	**0,730**	**0,640**	**0,635**		**0,625**	**0,581**	**0,574**
MIC	-0,267	-0,156	-0,146		-0,139	-0,119	-0,114
CC	**0,800**	**0,854**	**0,862**		**0,926**	**0,948**	**0,948**
RFC	**0,630**	**0,470**	**0,458**		**0,802**	**0,744**	**0,741**
WMPC1	**0,805**	**0,876**	**0,883**		**0,937**	**0,970**	**0,970**
WOC	-0,034	-0,169	-0,158		-0,063	-0,097	-0,091
LCOM3	**0,497**	**0,562**	**0,573**		0,300	**0,334**	**0,327**
TCC	-0,034	0,172	0,148		-0,181	-0,104	-0,116
LOC	**0,765**	**0,795**	**0,808**		**0,898**	**0,925**	**0,927**
NOA	**0,707**	**0,741**	**0,746**		**0,791**	**0,697**	**0,692**
NOO	**0,764**	**0,836**	**0,848**		**0,931**	**0,970**	**0,971**
DIT	0,001	-0,051	-0,069		-0,180	-0,235	-0,242

Tableau 19. Les corrélations pour Ant avant le refactoring.

[6] www.xlstat.com/fr/

Spearman					Pearson		
Variables	**TLOC**	**TNOO**	**TWMPC1**		**TLOC**	**TNOO**	**TWMPC1**
CBO	**0,544**	**0,605**	**0,619**		**0,568**	**0,520**	**0,529**
DAC	**0,523**	**0,595**	**0,596**		**0,526**	**0,472**	**0,471**
CE	**0,461**	**0,530**	**0,544**		**0,519**	**0,472**	**0,481**
ChC	-0,184	-0,041	-0,041		-0,160	-0,026	-0,027
MIC	0,311	0,349	0,343		-0,049	0,079	0,077
FO	**0,510**	**0,597**	**0,611**		**0,502**	**0,550**	**0,557**
CC	**0,816**	**0,827**	**0,834**		**0,872**	**0,610**	**0,613**
RFC	**0,559**	**0,666**	**0,675**		**0,673**	**0,554**	**0,557**
WMPC1	**0,789**	**0,791**	**0,798**		**0,885**	**0,565**	**0,568**
WOC	-0,082	0,029	0,041		-0,218	-0,149	-0,146
LCOM1	**0,521**	**0,534**	**0,536**		**0,779**	0,268	0,269
LCOM3	0,080	0,081	0,086		0,219	0,152	0,154
TCC	0,045	-0,023	-0,029		-0,132	-0,064	-0,070
LOC	**0,770**	**0,816**	**0,826**		**0,849**	**0,701**	**0,705**
NOA	**0,423**	**0,464**	**0,468**		**0,663**	**0,425**	**0,426**
NOO	**0,594**	**0,628**	**0,631**		**0,836**	**0,445**	**0,446**
DIT	0,043	0,141	0,141		0,106	0,123	0,122

Tableau 20. Les corrélations pour Archiva avant le refactoring.

Les tableaux 19 et 20 montrent les résultats de l'analyse des corrélations obtenus selon Spearman et Pearson (pour les deux systèmes Ant et Archiva) avant le refactoring basé sur les clones. Le premier constat que l'on peut faire, c'est que les corrélations sont presque les mêmes aussi bien pour Spearman que pour Pearson à quelques exceptions près comme TLOC-LCOM3 pour Ant**,** ou avec Spearman, la corrélation est significative (r=**0,497)**, et ne l'est pas avec Pearson (r=**0,300**). La même remarque pour les corrélations entre TNOO et TWMPC1 avec LCOM1 pour Archiva. Par ailleurs, avec Pearson, on observe les différentes corrélations entre les métriques de code source et des classes de test; TLOC (pour Ant) est positivement corrélée (fortes corrélations : r > **0 .7**) avec CBO, CE, FO, CC, WMPC1, LOC et NOO. Ceci peut expliquer le fait que les valeurs de ces dernières augmentent ensemble et diminuent ensemble.

TLOC a d'autres corrélations modérées (**0.3** < r < **0.7**) comme avec DAC, RFC et LCOM3. Nous avons trouvé que TLOC n'est pas fortement corrélée (r < **0.4**) avec chacune des

métriques de code source suivantes : MIC, WOC, TCC et DIT (r < **0,4**). Pour TNOO et TWMPC1 (pour Ant), il y a de fortes corrélations (corrélations positives) avec CC, WMPC1, LOC et NOO.

Pour Archiva, nous avons observé aussi que les métriques des classes de test sont fortement corrélées avec chacune des dernières métriques citées (CC, WMPC1, LOC et NOO), ce qui explique l'importance de ces dernières. Les fortes corrélations positives déduites des deux derniers tableaux montrent que la croissance ou la décroissance du nombre de lignes de code, du nombre de méthodes et de la complexité des classes de test dépendent fortement de la complexité, du nombre de méthodes et du nombre de lignes de code des classes du code source.

Spearman					**Pearson**		
Variables	TLOC	TNOO	Twmpc1		TLOC	TNOO	Twmpc1
CBO	**0,724**	**0,631**	**0,607**		**0,698**	**0,679**	**0,667**
CE	**0,756**	**0,657**	**0,639**		**0,523**	**0,511**	**0,501**
DAC	**0,663**	**0,531**	**0,507**		**0,751**	**0,669**	**0,656**
FO	**0,726**	**0,661**	**0,639**		**0,571**	**0,558**	**0,544**
MIC	-0,228	0,021	0,031		-0,150	-0,120	-0,114
CC	**0,728**	**0,835**	**0,829**		**0,921**	**0,948**	**0,946**
RFC	**0,812**	**0,670**	**0,656**		**0,831**	**0,772**	**0,764**
WMPC1	**0,727**	**0,856**	**0,852**		**0,930**	**0,969**	**0,968**
WOC	0,320	0,128	0,124		0,034	0,015	0,015
LCOM3	0,222	**0,403**	**0,411**		0,267	0,295	0,297
TCC	-0,353	-0,144	-0,165		-0,252	-0,206	-0,217
LOC	**0,708**	**0,834**	**0,825**		**0,888**	**0,932**	**0,930**
NOA	**0,582**	**0,686**	**0,680**		**0,755**	**0,656**	**0,650**
NOO	**0,677**	**0,819**	**0,821**		**0,926**	**0,968**	**0,967**
DIT	0,324	0,214	0,205		-0,027	-0,151	-0,159

Tableau 21. Les corrélations pour ANT après le refactoring.

Spearman					Pearson		
Variables	**TLOC**	**TNOO**	**TWMPC1**		**TLOC**	**TNOO**	**TWMPC1**
CBO	**0,524**	**0,469**	**0,469**		**0,522**	**0,450**	**0,450**
DAC	**0,421**	**0,408**	**0,408**		0,351	0,293	0,290
CE	**0,493**	**0,452**	**0,451**		**0,485**	**0,406**	**0,406**
ChC	-0,095	-0,053	-0,053		-0,133	-0,016	-0,017
MIC	**0,403**	0,374	0,368		-0,042	0,089	0,087
FO	**0,513**	**0,468**	**0,468**		**0,443**	**0,464**	**0,464**
CC	**0,852**	**0,797**	**0,794**		**0,897**	**0,568**	**0,567**
RFC	**0,580**	**0,598**	**0,601**		**0,733**	**0,514**	**0,514**
WMPC1	**0,838**	**0,777**	**0,775**		**0,893**	**0,546**	**0,545**
WOC	0,093	0,110	0,109		-0,121	-0,052	-0,050
LCOM1	**0,475**	**0,419**	**0,425**		**0,780**	0,262	0,262
LCOM3	-0,231	-0,237	-0,246		0,044	-0,038	-0,038
TCC	0,349	0,354	0,360		0,056	0,156	0,155
LOC	**0,845**	**0,808**	**0,806**		**0,841**	**0,675**	**0,673**
NOA	**0,383**	**0,376**	0,374		**0,610**	0,324	0,320
NOO	**0,647**	**0,609**	**0,607**		**0,833**	**0,417**	**0,415**
DIT	-0,065	-0,009	-0,009		0,041	0,001	-0,001

Tableau 22. Les corrélations pour ARCHIVA après le refactoring.

Les tableaux 21 et 22 montrent les résultats de l'analyse de corrélations pour les deux systèmes Ant et Archiva après le refactoring basé sur les clones. Nous avons repéré quelques différences entre les résultats obtenus selon Spearman et Pearson tels que: TNOO et TWMPC1 avec LCOM3 pour Ant (par exemple entre TNOO et LCOM3, Spearman: r = **0,403**, Pearson: r = **0,295**). Pour archiva, entre DAC et les différentes métriques de classes de test, les résultats étaient significatifs avec Spearman et ne l'étaient pas avec Pearson, ainsi qu'entre TLOC et MIC, entre TNOO et TWMPC1 avec LCOM1 et aussi entre TNOO et NOA.

En analysant les tableaux et en se référant aux corrélations obtenues par Pearson, nous avons observé toujours de fortes corrélations entre les métriques de classes de test et les métriques de code source : CC, WMPC1 et LOC dans les deux systèmes, ou TLOC est fortement corrélée aussi avec RFC et NOO (pour Ant et Archiva).

Ces fortes corrélations positives montrent que les valeurs de métriques de code source affectent les valeurs de métriques de classes de test, ce qui explique que suite à la croissance ou la décroissance d'une valeur de métrique, la valeur d'une ou de plusieurs autres métriques change.

Pour les deux systèmes, à partir des tableaux 23, 24, 25 et 26, nous pouvons observer que les métriques CC, WMPC1 et LOC sont les variables qui ont les degrés de corrélations les plus élevés avec les métriques de classes de test.

4.4.3 Analyse de corrélation entre les métriques de code source et les métriques de classes de test en utilisant les différences

Dans cette section, nous analysons la relation entre les différences observées dans les valeurs de métriques (code source et test) avant et après le refactoring basé sur les clones. Les formules de calcul sont citées ci-dessous :

Δ Métrique de code source = la valeur de métrique avant le refactoring – la valeur de métrique après le refactoring.

Δ Métrique de classes test = la valeur de métrique de classes test avant le refactoring – la valeur de métrique de classes test après le refactoring.

L'hypothèse à tester est la suivante :

Hypothèse : il existe une relation significative entre les différences de valeurs de métriques de code source (**Δ métrique du code source**) avec les métriques de classes de

test (**Δ métrique de classes tests**). L'hypothèse nulle est: il n'existe pas une relation significative entre les métriques de code source et celle des classes de test.

Afin de valider cette hypothèse, nous allons analyser la relation entre les différences de valeurs de métriques de code source avant et après le refactoring avec les différences de valeurs de métriques des classes de test. XLSTAT est utilisé pour calculer les corrélations (Spearman et Pearson). On fixe le seuil α=0.05 afin de pouvoir observer les corrélations significatives.

Spearman					Pearson		
Variables	**Δ TLOC**	**Δ TNOO**	**Δ Twmpc1**		**Δ TLOC**	**ΔTNOO**	**Δ Twmpc1**
ΔCBO	**0,529**	**0,534**	**0,686**		**0,515**	**0,855**	**0,869**
ΔCE	**0,462**	**0,633**	**0,610**		**0,509**	**0,863**	**0,860**
ΔDAC							
ΔFO	**0,521**	**0,637**	**0,641**		**0,460**	**0,739**	**0,738**
ΔMIC	-0,140	0,022	-0,004		0,298	**0,419**	**0,413**
ΔCC	**0,423**	**0,387**	**0,521**		**0,504**	**0,911**	**0,921**
ΔRFC	0,299	**0,489**	**0,393**		0,349	**0,669**	**0,661**
ΔWMPC1	**0,450**	**0,605**	**0,494**		**0,530**	**0,937**	**0,931**
ΔWOC	**0,517**	**0,637**	**0,635**		0,267	**0,502**	**0,499**
ΔLCOM3	**0,536**	**0,662**	**0,646**		0,150	**0,375**	**0,372**
ΔTCC	-0,149	-0,288	-0,302		-0,110	-0,296	-0,294
ΔLOC	**0,455**	**0,420**	**0,540**		**0,475**	**0,853**	**0,857**
ΔNOA	**0,617**	**0,649**	**0,625**		**0,502**	**0,833**	**0,829**
ΔNOO	**0,645**	**0,664**	**0,644**		**0,528**	**0,908**	**0,906**
ΔDIT	**-0,455**	-0,322	-0,290		**-0,411**	**-0,390**	**-0,385**

Tableau 23. Les corrélations pour Ant.

Spearman					Pearson		
Variables	**ΔTLOC**	**ΔTNOO**	**ΔTWMPC1**		**ΔTLOC**	**ΔTNOO**	**ΔTWMPC1**
ΔCBO	**0,568**	**0,581**	**0,581**		**0,586**	**0,500**	**0,504**
ΔDAC	0,324	0,293	0,249		0,371	0,215	0,211
ΔCE	**0,596**	**0,605**	**0,600**		**0,562**	**0,471**	**0,475**
ΔChC	**0,381**	**0,391**	0,352		0,230	0,154	0,152
ΔMIC	**0,556**	**0,497**	**0,417**		0,333	0,255	0,249
ΔFO	**0,533**	**0,543**	**0,540**		**0,561**	**0,457**	**0,453**
ΔCC	**0,432**	**0,551**	**0,583**		**0,693**	**0,658**	**0,656**
ΔRFC	**0,596**	**0,662**	**0,645**		**0,716**	**0,696**	**0,690**
ΔWMPC1	**0,433**	**0,552**	**0,584**		**0,761**	**0,802**	**0,805**
ΔWOC	0,109	0,117	0,133		**0,467**	**0,565**	**0,536**
ΔLCOM1	0,334	0,327	**0,401**		**0,434**	0,271	0,276
ΔLCOM3	0,168	0,138	0,228		-0,202	-0,293	-0,262
ΔTCC	-0,108	-0,029	-0,086		-0,215	-0,171	-0,207
ΔLOC	**0,666**	**0,675**	**0,722**		**0,721**	**0,670**	**0,673**
ΔNOA	**0,375**	0,334	0,310		**0,485**	0,339	0,341
ΔNOO	**0,516**	**0,545**	**0,537**		**0,746**	**0,669**	**0,673**
ΔDIT	0,012	-0,037	-0,170		0,079	0,040	-0,066

Tableau 24. Les corrélations pour Archiva.

Les deux tableaux 23 et 24 montrent les résultats de l'analyse de corrélations (pour les deux systèmes Ant et Archiva) pour les différences de valeurs des métriques avant et après le refactoring des clones. Les métriques CBO, CE, FO, CC, WMPC1, LOC et NOO sont positivement corrélées avec les différentes métriques de classes de test et l'effort de test unitaire dans les deux systèmes, ce qui explique que les changements de ces dernières dépendent des changements de métriques de code source déjà citées.

Test statistique

Le test Z est appliqué sur les valeurs de métriques obtenues avant et après le refactoring basé sur les clones afin de déduire les changements significatifs (amélioration ou dégradation) et non significatifs. Ce test est considéré comme un test paramétrique permettant de comparer les moyennes de deux échantillons. Ces derniers sont les valeurs de métriques avant et après le refactoring basé sur les clones. Ce genre de test est utilisé afin de déduire la significativité (positif ou négatif) des changements comme le refactoring en question. Nous avons opté pour ce test parce que les valeurs des métriques utilisées sont indépendantes et l'ensemble de données est grand. La différence entre les métriques avant et après le refactoring est significative si |Z| > Z critique. Nous allons appliquer le seuil typique (alpha = 0.05) pour décider si les différences entre les valeurs de métriques sont significatives.

Attributs		Différence	z (Valeur observée)	z (Valeur critique)	p-value (bilatérale)	Alpha
Couplage	**CBO**	8,833	6,845	1,96	< 0,0001	0,05
	CE	9,429	7,901	1,96	< 0,0001	0,05
	DAC	1,357	4,514	1,96	< 0,0001	0,05
	CM	16,905	1,039	1,96	0,299	0,05
	FAN-IN	6,5	1,008	1,96	0,314	0,05
	ChC	7,524	1,171	1,96	0,242	0,05
	FO	6,548	6,133	1,96	< 0,0001	0,05
	MIC	73,238	7,045	1,96	< 0,0001	0,05
Complexité	**CC**	29,524	6,275	1,96	< 0,0001	0,05
	RFC	77,095	10,135	1,96	< 0,0001	0,05
	WMPC1	27,881	6,294	1,96	< 0,0001	0,05
	WOC	130,095	15,187	1,96	< 0,0001	0,05
Cohésion	**LCOM1**	209,683	1,788	1,96	0,074	0,05
	LCOM3	73,025	15,27	1,96	< 0,0001	0,05
	TCC	10,333	4,697	1,96	< 0,0001	0,05
Taille	**LOC**	143,952	7,091	1,96	< 0,0001	0,05
	NOA	5,667	7,616	1,96	< 0,0001	0,05
	NOO	14,024	5,39	1,96	< 0,0001	0,05
Héritage	**DIT**	3,167	19,41	1,96	< 0,0001	0,05
	NOC	0,095	1,432	1,96	0,152	0,05

Tableau 25. Le test Z pour ANT (métriques de classes).

	Différence	z (Valeur observée)	z (Valeur critique)	p-value (bilatérale)	Alpha
TLOC	381,024	4,926	1,96	< 0,0001	0,05
TNOO	31,5	6,46	1,96	< 0,0001	0,05
TWMPC1	32,881	6,774	1,96	< 0,0001	0,05

Tableau 26. Le test Z pour ANT (métriques de classes de test).

Attributs		Différence	z (Valeur observée)	z (Valeur critique)	p-value (bilatérale)	Alpha
Couplage	**CBO**	14,825	8,485	1,96	< 0,0001	0,05
	DAC	3,05	6,164	1,96	< 0,0001	0,05
	CE	15,175	7,863	1,96	< 0,0001	0,05
	CM	0,2	1,599	1,96	0,11	0,05
	FAN-IN	-	-	-	-	-
	ChC	1,225	6,526	1,96	< 0,0001	0,05
	MIC	32,45	3,859	1,96	0	0,05
	FO	10,425	8,289	1,96	< 0,0001	0,05
Complexité	**CC**	18,65	7,268	1,96	< 0,0001	0,05
	RFC	54,1	9,002	1,96	< 0,0001	0,05
	WMPC1	17,725	7,422	1,96	< 0,0001	0,05
	WOC	116,975	10,705	1,96	< 0,0001	0,05
Cohésion	**LCOM1**	88,778	2,132	1,96	0,033	0,05
	LCOM3	71,742	11,992	1,96	< 0,0001	0,05
	TCC	30,487	5,014	1,96	< 0,0001	0,05
Taille	**LOC**	139,375	8,415	1,96	< 0,0001	0,05
	NOA	5,05	5,441	1,96	< 0,0001	0,05
	NOO	10,275	6,063	1,96	< 0,0001	0,05
Héritage	**DIT**	1,675	10,909	1,96	< 0,0001	0,05
	NOC	0,05	1	1,96	0,317	0,05

Tableau 27. Le test Z pour Archiva (métriques de classes).

	Différence	z (Valeur observée)	z (Valeur critique)	p-value (bilatérale)	Alpha
TLOC	197,75	7,704	1,96	< 0,0001	0,05
TNOO	20,225	11,296	1,96	< 0,0001	0,05
TWMPC1	20,25	11,364	1,96	< 0,0001	0,05

Tableau 28. Le test Z pour Archiva (métriques de classes de test).

Les tableaux 25 et 27 montrent les résultats obtenus après l'application du test statistique Z sur les données que nous avons obtenu avant et après le refactoring (les métriques de classes). Le test n'est pas significatif si les p-values sont supérieures à 0.05 ($\alpha > 0.05$) :

- **ANT** : ce test est significatif pour toutes les métriques de code source sauf CM, FAN-IN, ChC, LCOM1 et NOC.
- **Archiva** : ce test est significatif pour toutes les métriques de code source sauf CM, FAN-IN, et NOC.

Pour les valeurs de métriques de classes de test, les changements sont significatifs dans les deux systèmes et les p-value sont inférieures au seuil défini.

Pour ANT, a partir du tableau 25, nous pouvons observer que les valeurs de la plupart des metriques de code source (p-value correspondant au test Z) prises en particulièrement sont significatives apres l'application du refactoring. Particulirement pour CBO, CE, DAC, FO, MIC, CC, RFC, WMPC1, WOC, LCOM3, TCC, LOC, NOA, NOO et DIT. Ces métriques correpondent respectivement aux attributs internes suivants : le couplage, la complexité, la cohésion, la taille et l'héritage. Seulement quelques changements non significatifs (p-value > 0.05) sont observés. Ces métriques correspondent respectivement au couplage (CM, ChC et FAN-IN), la cohésion (LCOM1) et l'héritage (NOC). Ces résultats laissent déduire que les améliorations apportées par le refactoring basé sur les clones peuvent affecter significativement les métriques de code source et les attributs internes de la qualité ainsi que la testabilité unitaire des classes.

Pour ANT, à partir du tableau 26, nous pouvons observer que les valeurs des trois métriques de classes de test (TLOC, TNOO et TWMPC1) sont significatives après l'application du refactoring des clones. Ces résultats laissent deduire que les améliorations apportées par ce dernier peuvent affecter significativement la testabilité unitaire des classes.

Pour Archiva, à partir du tableau 27, nous pouvons observer que les valeurs de la plupart des métriques de code source (p-value correspondant au test Z) prises en considération sont significatives après l'application du refactoring. particulièrement pour CBO, CE, ChC, DAC, FO, MIC, CC, RFC, WMPC1, WOC, LCOM3, TCC, LOC, NOA, NOO et DIT. Ces métriques correspondent respectivement aux attributs internes suivants : le couplage, la complexité, la cohésion, la taille et l'héritage. Seulement quelques changements non significatifs (p-value > 0.05) sont observés. Ces métriques correspondent respectivement au couplage (CM, FAN-IN), la cohésion (LCOM1) et l'héritage (NOC). Ces résultats laissent déduire que les améliorations apportées par l'application du refactoring peuvent affecter significativement les métriques de code source et les attributs internes de la qualité ainsi que sur la testabilité unitaire des classes.

Pour Archiva, à partir du tableau 28, nous pouvons observer que les valeurs des trois métriques de classes de test (TLOC, TNOO et TWMPC1) sont significatives après l'application du refactoring basé sur les clones. Nous pouvons conclure que le refactoring basé sur les clones peut affecter significativement la testabilité unitaire des classes (les métriques de code source et des classes de test).

4.4.4 L'impact de variations de métriques de code source sur l'effort de test en utilisant la régression linéaire

Dans la première partie de cette section, nous nous penchons sur l'impact individuel des variations de chaque métrique (les différences) de code source (comme variables dépendantes) sur les variations de chaque métrique (les différences) de classes de test (comme variables indépendantes) en utilisant la régression linéaire simple.

Cette technique va nous permettre: (1) l'identification des métriques de code source qui sont reliées significativement avec l'effort de test unitaire en utilisant les variations de chaque métrique (les différences). (2) la recherche des métriques de code source qui

peuvent prédire où qui ont plus d'impact sur les métriques de classes de test après le refactoring basé sur les clones.

La régression linéaire simple est une technique statistique très utilisée. Elle permet d'établir ou de chercher le lien entre deux variables, une variable dépendante (Y) et une autre variable indépendante (X), afin de faire les prédictions sur (Y) lorsque (X) est mesurée où : $Y=\beta_0+ \beta_1X$.

L'objet de cette section est d'appliquer cette technique afin de prédire l'impact individuel possible des variations de métriques de code source sur l'effort de test en général et sur les métriques de classes de test en particulier. Nous visons à extraire les métriques qui ont les impacts les plus élevés sur l'effort de test des classes, ces dernières sont considérées comme les meilleurs prédicteurs. Comme parmi une série de logiciels qui offre la possibilité d'analyser les données, notre choix s'est posé sur XLSTAT pour la qualité des tableaux qu'il présente.

Les tableaux fournis par XLSTAT suite à l'application de la régression linéaire simple ou multiple sont essentiels pour faire des prévisions, simulations ou pour faire des comparaisons. Ces tableaux contiennent les coefficients d'ajustement des modèles.

Le R^2 qui est le coefficient de détermination donnant une idée du pourcentage de variabilité de la variable à modéliser par la variable explicative. Plus ce coefficient est proche de 1, meilleur est le modèle. Ils contiennent aussi une colonne « Pr > F » qui est le test de Fisher. Il est utilisé afin de tester si la moyenne de la variable à modéliser suffirait à décrire les résultats obtenus. Il permet aussi de conclure si la variable explicative apporte une quantité d'information significative au modèle. Donc, c'est à ce niveau qu'on décide, c'est le modèle est à prendre ou à rejeter.

		ΔTLOC	ΔTNOO	ΔTWMPC1
	R^2	0,236	0,707	0,699
ΔCBO	Beta	0,486	0,809	0,828
	P-value	0,001	< 0,0001	< 0,0001
	R^2	0,209	0,680	0,678
ΔCE	Beta	0,457	0,785	0,788
	P-value	0,002	< 0,0001	< 0,0001
	R^2	0,004	0,000	0,000
ΔDAC	Beta	0,066	-0,010	-0,012
	P-value	0,677	0,952	0,940
	R^2	0,193	0,457	0,455
ΔFO	Beta	0,440	0,676	0,675
	P-value	0,004	< 0,0001	< 0,0001
	R^2	0,103	0,113	0,112
ΔMIC	Beta	0,321	0,336	0,334
	P-value	0,038	0,029	0,031
	R^2	0,162	0,956	0,959
ΔCC	Beta	0,402	0,899	0,904
	P-value	0,008	< 0,0001	< 0,0001
	R^2	0,287	0,004	0,004
ΔRFC	Beta	0,522	-0,061	-0,065
	P-value	0,002	0,699	0,684
	R^2	0,172	0,964	0,962
ΔWMPC1	Beta	0,414	0,924	0,927
	P-value	0,006	< 0,0001	< 0,0001
	R^2	0,129	0,008	0,009
ΔWOC	Beta	0,340	-0,091	-0,094
	P-value	0,063	0,566	0,555
	R^2	0,150	0,141	0,137
ΔLCOM3	Beta	0,311	0,375	0,372
	P-value	0,137	0,041	0,043
	R^2	0,006	0,130	0,131
ΔTCC	Beta	-0,078	0,361	0,362
	P-value	0,647	0,028	0,028
	R^2	0,432	0,883	0,884
ΔLOC	Beta	0,619	0,830	0,835
	P-value	< 0,0001	< 0,0001	< 0,0001
	R^2	0,291	0,888	0,887
ΔNOA	Beta	0,544	0,942	0,942
	P-value	0,001	< 0,0001	< 0,0001
	R^2	0,203	0,862	0,861
ΔNOO	Beta	0,450	0,936	0,939
	P-value	0,003	< 0,0001	< 0,0001
	R^2	0,288	0,000	0,000
ΔDIT	Beta	-0,499	-0,021	-0,019
	P-value	0,005	0,894	0,906

Tableau 29. Résultats de la régression linéaire simple sur ANT.

Le tableau 29 présente les résultats de la régression linéaire simple pour ANT:

- Les modèles linéaires pour prédire **Δ TLOC** sont tous significatifs sauf les modèles basés sur les métriques suivantes: DAC, WOC et TCC. Car, les R^2 sont faibles et les p-value associées au test de Student (Pr > t) sont supérieures à 0.05 (comme pour DAC : p-value=**0,677 et** R^2= **0,004).** Donc, les variations de métriques de code source (les métriques jugées significatives) ont un impact significatif sur les variations de la métrique de classes de test TLOC.

- Les modèles linéaires pour prédire **Δ TNOO** et **Δ TWMPC1** sont tous significatifs sauf pour les métriques suivantes: DAC, RFC, WOC et DIT.

- Il s'est avéré que les modèles de régression linéaire basés sur les métriques de code source ont les valeurs de R^2 les plus élevées avec les deux métriques de classes de test TNOO et TWMPC1. Le R^2 en tant qu'indicateur permettant de juger la qualité d'une régression linéaire, il peut prendre des valeurs entre 0 et 1. Il permet de mesurer l'adéquation entre le modèle et les données observées. Plus la valeur de R^2 est proche de 1, plus le modèle est bon.

		ΔTLOC	ΔTNOO	ΔTWMPC1
	R^2	0,352	0,282	0,287
ΔCBO	Beta	0,594	0,531	0,535
	P-value	< 0,0001	0,000	0,000
	R^2	0,114	0,224	0,222
ΔDAC	Beta	0,338	0,397	0,394
	P-value	0,033	0,038	0,039
	R^2	0,354	0,280	0,284
ΔCE	Beta	0,595	0,529	0,533
	P-value	< 0,0001	0,000	0,000
	R^2	0,140	0,127	0,124
ΔMIC	Beta	0,374	0,356	0,352
	P-value	0,017	0,024	0,026
	R^2	0,254	0,181	0,180
ΔFO	Beta	0,504	0,426	0,424
	P-value	0,001	0,006	0,006
	R^2	0,369	0,33	0,331
ΔCC	Beta	0,607	0,574	0,576
	P-value	< 0,0001	0,000	0,000
	R^2	0,573	0,540	0,535
ΔRFC	Beta	0,757	0,735	0,732
	P-value	< 0,0001	< 0,0001	< 0,0001
	R^2	0,698	0,713	0,717
ΔWMPC1	Beta	0,771	0,761	0,759
	P-value	< 0,0001	< 0,0001	< 0,0001
	R^2	0,247	0,299	0,291
ΔWOC	Beta	0,483	0,516	0,507
	P-value	0,009	0,004	0,005
	R^2	0,185	0,266	0,270
ΔLCOM1	Beta	0,430	0,525	0,524
	P-value	0,014	0,008	0,007
	R^2	0,04	0,084	0,067
ΔLCOM3	Beta	-0,201	-0,290	-0,259
	P-value	0,296	0,127	0,176
	R^2	0,023	0,016	0,026
ΔTCC	Beta	-0,152	-0,127	-0,162
	P-value	0,383	0,467	0,354
	R^2	0,4471	0,405	0,410
ΔLOC	Beta	0,669	0,636	0,640
	P-value	< 0,0001	< 0,0001	< 0,0001
	R^2	0,155	0,285	0,289
ΔNOA	Beta	0,394	0,478	0,479
	P-value	0,012	0,010	0,010
	R^2	0,175	0,135	0,140
ΔNOO	Beta	0,419	0,368	0,375
	P-value	0,007	0,019	0,017
	R^2	0,151	0,1131	0,279
ΔDIT	Beta	0,389	0,337	0,191
	P-value	0,0132	0,033	0,340

Tableau 30. Résultats de la régression linéaire simple pour Archiva.

Le tableau 30 présente les résultats de la régression linéaire simple pour Archiva:

- Les modèles linéaires pour prédire **ΔTLOC** sont tous significatifs sauf le modèle basé sur la métrique suivante: LCOM3. Donc, les variations de métriques de code source (les métriques jugées significatives) ont un impact significatif sur les variations de la métrique de classes de test TLOC.

- Les modèles linéaires pour prédire **ΔTNOO** sont tous significatifs sauf le modèle basé sur la métrique suivante: LCOM3.

- Les modèles linéaires pour prédire **ΔTWMPC1** sont tous significatifs sauf les modèles basés sur les métriques suivantes: LCOM3 et DIT.

- Nous avons observé que les variations des métriques de code source LOC, WMPC1 et RFC ont les impacts les plus élevés sur les trois métriques de classes de test: TLOC, TNOO et TWMPC1.

Le tableau 31, présente un résumé de la régression linaire simple appliquée sur les deux systèmes en question. Le signe (-) indique que le modèle est non significatif pour la prédiction, (+) indique que le modèle est significatif pour la prédiction, (++) indique que la variation de la métrique de code source a un impact élevé sur la métrique de classes de test. Selon les résultats obtenus dans le tableau 31 les variations de métriques de code source ont un impact important sur les deux métriques de classes de test TNOO et TWMPC1.

		Δ TLOC	Δ TNOO	Δ TWMPC1
		Pr > F	Pr > F	Pr > F
Δ CBO	**Ant**	+	++	++
Δ CBO	**Archiva**	++	+	+
Δ CE	**Ant**	+	++	++
Δ CE	**Archiva**	++	+	+
Δ DAC	**Ant**	-	-	-
Δ DAC	**Archiva**	+	+	+
Δ FO	**Ant**	+	++	++
Δ FO	**Archiva**	+	+	+
Δ MIC	**Ant**	+	+	+
Δ MIC	**Archiva**	+	+	+
Δ RFC	**Ant**	+	-	-
Δ RFC	**Archiva**	++	++	++
Δ WMPC1	**Ant**	+	++	++
Δ WMPC1	**Archiva**	++	++	++
Δ WOC	**Ant**	-	-	-
Δ WOC	**Archiva**	+	+	+
Δ LCOM3	**Ant**	+	++	++
Δ LCOM3	**Archiva**	-	-	-
Δ TCC	**Ant**	-	+	+
Δ TCC	**Archiva**	+	+	+
Δ LOC	**Ant**	++	++	++
Δ LOC	**Archiva**	++	++	++
Δ NOA	**Ant**	+	++	++
Δ NOA	**Archiva**	+	+	+
Δ NOO	**Ant**	+	++	++
Δ NOO	**Archiva**	+	+	+
Δ DIT	**Ant**	+	-	-
Δ DIT	**Archiva**	+	+	-
Δ LCOM1	**Archiva**	+	+	+

Tableau 31. Résumé des résultats de la régression linéaire simple.

La réalisation des régressions simples nous a permis de faire la présélection des métriques de code source probablement adéquates pour l'explication de (ou des) la variable dépendante (métriques de classes de test et leurs variations). Pour éviter le problème de la multi-colinéarité entre les variables dépendantes, et éviter les biais que cela peut induire dans la régression multiple, il est donc préférable de sélectionner ou d'introduire dans le modèle multiple seulement les métriques fortement indépendantes, et cela pour s'assurer de l'indépendance des métriques explicatives. Les variables sont multi-colinéaires, s'il existe une relation linéaire entre ces variables.

Afin de résoudre ce problème, nous avons opté pour l'application de l'analyse en composante principale (ACP) sur chaque ensemble de métriques représentant un même attribut (métriques de couplage, métriques de taille, etc.). Cette alternative nous permettra de réduire le nombre de métriques de sortes qu'on garde l'information nécessaire. Donc, au lieu de prendre les cinq métriques de couplage, nous allons appliquer l'ACP sur cet ensemble et en se référant aux tableaux de contributions des variables avec les vecteurs propres, nous allons garder seulement les meilleures. F1, F2, F3, F4 et F5 sont les facteurs (les composantes principales). Le vecteur F1 explique la plus grande part d'information (inertie), le deuxième vecteur F2 explique la plus grande part de l'inertie restante, etc. On ne sélectionne pas deux variables qui ont les valeurs propres très élevées avec le même vecteur propre, parce qu'il y a une forte probabilité qu'elles soient en multi-colinéarité.

Les résultats sont présentés dans les tableaux de 32 à 36.

	F1	F2	F3	F4	F5
CBO	30,227	0,591	1,200	17,611	50,371
CE	29,538	3,696	0,001	18,437	48,328
DAC	7,164	58,193	30,552	2,890	1,200
FO	23,047	17,314	1,217	58,391	0,032
MIC	10,024	20,206	67,029	2,671	0,070

Tableau 32. Contribution des variables de couplage pour Ant.

	F1	F2	F3	F4
CC	49,841	0,024	0,169	49,965
RFC	0,019	52,531	47,440	0,011
WMPC1	49,891	0,036	0,054	50,018
WOC	0,249	47,409	52,337	0,006

Tableau 33. Contribution des variables de complexité pour Ant.

	F1	F2	F3	F4	F5	F6
CBO	25,311	0,455	0,018	8,565	11,248	54,403
DAC	21,560	0,995	0,042	77,356	0,002	0,045
CE	25,393	0,110	0,100	6,535	23,243	44,619
CM	0,000	0,000	0,000	0,000	0,000	0,000
FAN-In	0,000	0,000	0,000	0,000	0,000	0,000
ChC	1,354	57,090	40,982	0,152	0,375	0,046
MIC	2,077	38,677	58,646	0,052	0,548	0,001
FO	24,306	2,674	0,213	7,339	64,583	0,886

Tableau 34. Contribution des variables de couplage pour Archiva.

	F1	F2	F3	F4
CC	33,550	2,636	0,460	63,354
RFC	32,318	0,023	55,616	12,043
WMPC1	32,674	0,344	43,539	23,443
WOC	1,458	96,997	0,384	1,161

Tableau 35. Contribution des variables de complexité pour Archiva.

	F1	F2	F3
LCOM1	22,794	63,522	13,684
LCOM3	32,351	36,147	31,502
TCC	44,855	0,331	54,814

Tableau 36. Contribution des variables de cohésion pour Archiva.

Pour Ant, les métriques suivantes ont été sélectionnées:

- CBO, DAC et MIC comme métriques de couplage.
- WMPC1, WOC et RFC comme métriques de complexité.
- LOC comme métrique de taille.
- LCOM 3 comme métrique de cohésion.
- DIT comme métrique d'héritage.

Et pour archiva, les métriques suivantes ont été sélectionnées :

- CBO, ChC et MIC comme métriques de couplage.
- RFC et WOC comme métriques de complexité.
- LOC comme métrique de taille.
- LCOM3 et TCC pour la cohésion.
- DIT pour l'héritage.

Dans la deuxième partie de cette section, les résultats de la régression linéaire multiple vont être présentés. L'objectif visé est d'explorer l'effet de combiner les métriques du code source (leur variation) pour prédire la variation de la testabilité unitaire des classes.

La régression linéaire multiple est une technique statistique permettant de prédire une variable (Y) dite variable dépendante en utilisant une combinaison de variables dites: variables indépendantes, où :

$$Y_i = a_0 + a_1 x_{i,1} + a_2 x_{i,2} + \cdots + a_p x_{i,9+\varepsilon_i;\ i=1,\ldots,n}. \ldots\ldots\ldots (2)$$

XLSTAT a été utilisé afin de faire cette analyse. Le recours à cette technique permet de déduire les meilleurs prédicteurs de la testabilité unitaire des classes.

ANT		ΔTLOC	ΔTNOO	ΔTWMPC1
	R^2	0,494	0,985	0,981
	P-value	< 0,0001	< 0,0001	< 0,0001
	Beta	LOC 0,534 RFC 0,298	WMPC1 1,745 LOC -0,798	WMPC1 1,700 LOC -0,751

Tableau 37. Résultats de la régression linéaire multiple pour ANT.

Dans cette section, nous présentons les résultats obtenus en utilisant la régression linéaire multivariée. Le but principal est d'examiner l'effet de combiner plusieurs métriques afin de prédire la testabilité unitaire des classes (l'effort de test unitaire). Nous avons utilisé l'ensemble de métriques déjà cité. Le tableau 37 résume les résultats de la régression linéaire multivariée avec la méthode de sélection «stepwise» pour Ant :

- Le meilleur modèle pour la métrique de test TLOC, comporte les deux métriques LOC (taille) et RFC (complexité) avec un R^2 = 0.494 et p-value < 0,0001. Ces résultats laissent déduire que les changements de la métrique TLOC sont influencés par les changements des métriques de code source LOC et RFC (la taille et la complexité des classes).
- Pour TNOO et TWMPC1, les métriques suivantes ont été obtenues : WMPC1 (la complexité) et LOC (taille) avec un R^2 = 0.985 et p-value < 0,0001 pour le modèle de TNOO et un R^2 = 0.981 et p-value < 0,0001 pour le modèle de TWMPC1. Ce qui explique que les variations des valeurs de WMPC1 et LOC provoque des variations significatives sur les valeurs des deux métriques TWMPC1 et TNOO.
- La métrique de classes de test TNOO est la plus prédictible, car son R^2 est le plus élevé comparativement aux autres métriques de test (TLOC et TWMPC1).
- En plus, les valeurs de R^2 obtenues pour ces modèles sont significativement plus élevées que ceux obtenus en utilisant la régression linéaire univariée.

- Les résultats obtenus montrent que les métriques TLOC, TNOO et TWMPC1 sont influencées par les variations de la complexité et la taille des classes (LOC et RFC, WMPC1 sont les meilleures prédicteurs de la testabilité unitaire).

ARCHIVA		ΔTLOC	ΔTNOO	ΔTWMPC1
	R^2	0,705	0,694	0,694
	P-value	< 0,0001	< 0,0001	< 0,0001
	Beta	LOC 0,777 WOC 0,288	LOC 0,729 WOC 0,333	LOC 0,730 WOC 0,324

Tableau 38. Résultats de la régression linéaire multiple pour ARCHIVA.

Le tableau 38 résume les résultats de la régression linéaire multivariée avec la méthode de sélection «stepwise» pour Archiva :

- Le meilleur modèle obtenu pour TLOC comporte : LOC (taille) et WOC (complexité) comme des meilleurs prédicteurs avec un $R^2 = 0.705$ et p-value < 0,0001 pour le modèle. Ce qui explique que les changements de ces derniers ont les plus grands impacts sur TLOC.
- Pour TNOO et TWMPC1, WOC (complexité) et LOC, ont été obtenues avec un $R^2 = 0.694$ et p-value < 0,0001 pour la variable TNOO et un $R^2 = 0.694$ et p-value < 0,0001 pour TWMPC1.
- LOC a plus d'impact sur les trois modèles, car la beta est plus grande que celles de la métrique WOC.
- Les résultats obtenus montrent que les métriques TLOC, TNOO et TWMPC1 sont influencées par les variations de la complexité et la taille des classes.
- Les modèles obtenus en combinant les métriques de code source et en utilisant la régression linéaire multivariée sont meilleurs que ceux obtenus en utilisant la régression linéaire univariée, car les R^2 ont significativement augmentées (Plus la valeur de R^2 est proche de 1, plus le modèle est bon).
- Les résultats obtenus viennent confirmer les hypothèses citées auparavant, ainsi que les fortes corrélations entre les métriques de code source et celles de classes de test, comme entre LOC et WMPC1 avec TLOC (pour Ant).

La régression linéaire nous a permis d'identifier les meilleurs prédicteurs de la testabilité unitaire des classes (l'effort de test unitaire) pour les métriques TLOC, TNOO et TWMPC1 représentant respectivement le nombre de lignes de code, le nombre de méthodes et la somme des complexités pour les classes de test. Les variations de ces dernières métriques dépendent des variations de métriques citées comme meilleurs prédicteurs, qui sont des métriques très explicites en matière d'effort de test unitaire. Les résultats viennent confirmer certaines fortes corrélations trouvées auparavant comme entre LOC et TLOC, TNOO et TWMPC1 (pour Ant).

4.5 Conclusion

Pour capturer la testabilité unitaire des classes dans les deux systèmes en question, nous avons utilisé les trois métriques: TLOC, TWMPC1, TNOO. Dans le but d'évaluer la relation entre les métriques de code source et celles des classes de test, des corrélations et la régression linéaire simple et multiple ont été utilisées.

Nous avons utilisé la régression linéaire simple pour faire la présélection des métriques adéquates pour l'explication des métriques dépendantes qui sont respectivement TLOC, TNOO, TWMPC1.

Finalement, nous avons appliqué la régression linéaire multiple pour explorer l'effet de combiner les métriques de code source sur la prédiction de la testabilité des classes. Les meilleurs prédicteurs de la testabilité unitaire sont : RFC, WMPC1, WOC (des métriques de complexité) et LOC (une métrique de taille).

Chapitre 5 – Conclusion générale

Chapitre 5 – Conclusion générale

Dans ce mémoire, nous avons mené une étude empirique portant sur l'évaluation de l'impact du refactoring des clones sur la testabilité. Cette dernière a été abordée (en particulier) du point de vue de l'effort de test unitaire (effort lié à l'écriture des suites de tests JUnit). Ce travail présente une étude quantitative comparant la testabilité unitaire des classes avant et après le refactoring OO basé sur les clones. Notre méthodologie pour analyser l'effet du refactoring des clones sur l'effort de test unitaire a été décomposée en deux niveaux: (1) le premier, est basé sur la relation entre les attributs de qualité internes (les métriques de code source) et la testabilité unitaire des classes, et (2) le deuxième, est basé sur les différents attributs de tests unitaires (les métriques de classes de test), qui ont une relation avec l'effort requis pour écrire le code des tests unitaires.

Notre étude a été réalisée grâce à plusieurs versions de deux systèmes open source développés en Java (Ant et Archiva). Nous avons focalisé seulement sur la testabilité des classes avant et après le refactoring basé sur les clones. Pour capturer la testabilité, nous avons utilisé un ensemble de trois métriques pour quantifier la suite de tests JUnit correspondante. Cette dernière a été générée en utilisant CodePro. Nous avons aussi utilisé un autre ensemble de métrique OO afin de mesurer les attributs du code source (comme le couplage et la taille) des classes avant et après le refactoring (attributs reliés à la testabilité unitaire des classes). Nous avons comparé comment les valeurs de ces métriques changent avant et après l'application du refactoring. Nous avons utilisé des tests statistiques, des corrélations et la régression linéaire simple et multiple afin d'extraire les meilleurs prédicteurs de la testabilité unitaire des classes.

Ce mémoire a été décomposé en plusieurs parties. Dans la première partie (chapitre 2), nous avons fait une description détaillée de tous les aspects liée au refactoring orienté objet basé sur les clones. Nous avons expliqué les raisons d'utilisations des clones, ainsi que les différents types de clones. De plus, nous avons donné les bénéfices, les approches, les outils, le processus et les différentes techniques de refactoring. Pour conclure cette partie, nous avons fait une analyse globale de plusieurs travaux portant sur l'impact de refactoring des clones sur la qualité afin d'en tirer des avantages.

Dans la deuxième partie (chapitre 3), le but était d'élaborer un modèle conceptuel de la testabilité. Au début, nous avons décrit les indicateurs de qualité selon le ISO/IEC 9126, la maintenabilité comme un attribut qualité, ainsi que la testabilité comme une sous caractéristique. Nous avons donné une description détaillée de tous les aspects liés afin de faciliter la compréhension du modèle proposé. Une synthèse de plusieurs articles portant sur la qualité, et sur la testabilité en particulier a été faite. A la fin de cette partie, nous avons présenté notre cadre final, qui a été utilisé pour évaluer l'impact du refactoring orienté objet (basé sur les clones) sur la testabilité des classes.

Dans la troisième partie (chapitre 4), a présenté notre étude empirique portant sur l'évaluation d'impact de refactoring des clones sur la testabilité. Tout d'abord, nous avons rappelé l'objectif de l'étude, la collecte de données et les résultats ainsi que leurs interprétations. Les résultats ont été présentés en quatre parties : (1) La statistique descriptive, (2) Les corrélations entre les métriques de code source et les métriques de classes de test, (3) L'analyse de corrélation se basant sur les différences et (4) L'analyse d'impact de variations des métriques de code source sur l'effort de test en utilisant la régression linéaire simple et multiple. Pour conclure cette étape, nous avons présenté les meilleurs prédicteurs de la testabilité unitaire des classes.

A travers les études de cas présentées et à la lumière de l'évaluation faite dans le chapitre 4, les résultats obtenus sont satisfaisants, encourageants et représentent tout de même une validation des techniques adaptées.

Dans le cadre d'une amélioration continue et futures perspectives de ce projet, nous pourrons focaliser nos efforts sur les axes suivants : (1) Élargir notre cadre d'évaluation de la testabilité unitaire en incluant d'autres caractéristiques et d'autres métriques de code source, (2) Utiliser d'autres méthodes de refactoring basé sur les clones, (3) Évaluer l'impact du refactoring en utilisant d'autres techniques comme les méthodes d'apprentissage automatique, (4) Répliquer notre travail sur d'autres systèmes OO afin d'avoir des résultats plus généraux.

Références

[1] I. Baxter, A. Yahin, L. Moura and M. Sant Anna, Clone Detection Using Abstract Syntax Trees. In Proceedings of the 14th International Conference on Software Maintenance (ICSM'98), pp. 368-377, Bethesda, Maryland, November 1998.

[2] T. Kamiya and S. Kusumoto, CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. Transactions on Software Engineering, Vol. 28(7): 654- 670, July 2002.

[3] JP. Retaillé, Refactoring des applications Java/J2EE. ISBN: 2-212-11577-6, 2005.

[4] C. Kumar and R. Cordy, A Survey on Software Clone Detection Research. Technical Report No. 2007-541, Queen's University at Kingston, 2007.

[5] C. Kapser and M. Godfrey, Clones considered harmful. In Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06), pp. 19-28, Benevento, Italy, October 2006.

[6] G. Hegedus and G. Hrabovszki, Effect of Object Oriented Refactoring on Testability, Error Proneness and other Maintainability Attributes. University of Szeged, Szeged, Hungary.

[7] A. Fontana, M. Zanoni, Ranchetti and D. Anchetti, Software Clone Detection and Refactoring. University of Milano-Bicocca, Italy, 2013.

[8] M. Kim, V. Sazawal, D. Notkin and C. Murphy, An empirical study of code clone genealogies. Proc. ESEC-FSE, 2005, pp. 187–196.

[9] M. Monzur, R. Arifur and A. Salah Uddin, A Literature Review of Code Clone Analysis to Improve Software Maintenance Process. Department of Computer Science, American International University-Bangladesh, Carleton University-Canada, SCICON & Tiger HATS-Bangladesh.

[10] K. Hotta, Y. Sano, Y. Higo and S. Kusumoto, Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution? An Empirical Study on Open Source Software. Proc. EVOL/IWPSE, 2010, pp. 73–82.

[11] K. Miryung and M. Gail, An Empirical Study of Code Clone Genealogies. In Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/SIGSOFT FSE 2005 '05), pp. 187-196, Lisbon, Portugal, September 2005.

[12] D. Advani, Y. Hassoun and S. Consell, Extracting Refactoring Trends from Open-Source Software and a Possible Solution to the Related Refactoring Conundrum. SAC, 2006.

[13] S. Dahmane, Étude et automatisation de refactorings pour le langage Delphi. Institut informatique, Université de Mons-Hainaut, 2004-2005.

[14] N. Kumari and S. Anju, Effect of refactoring on software quality. USICT, Dwarka, Delhi, India.

[15] M. Alshayeb, Empirical Investigation Of Refactoring Effect On Software Quality. Volume 51, Issue 9, Pages 1319-1326, 2009, Elsevier.

[16] Elish, O. Karim and M. Alshayeb, Investigating the Effect of Refactoring on Software Testing Effort. In Software Engineering Conference, APSEC'09, Asia-Pacific, pp. 29-34, IEEE, 2009.

[17] M. Fowler, Refactoring: Improving the Design of Existing Code.

[18] R. Dhavleesh, B. Rajesh and S. Maninder, Software clone detection: A systematic review.

[19] Tool Simian <http://www.harukizaemon.com/simian/index.html> (accessed April 2012).

[20] L. Jiang, G. Misherghi, Z. Su and S. Glondu, DECKARD: scalable and accurate tree-based detection of code clones. In: ICSE ’07: proceedings of the 29th international conference on software engineering. IEEE Computer Society, Los Alamitos, pp 96–105, 2009.

[21] J. Krinke, Identifying similar code with program dependence graphs. In: WCRE ’01: proceedings of the eigth working conference on reverse engineering (WCRE 2001). ACM, New York, pp 301–309, 2001.

[22] G. Casazza, G. Antoniol, U. Villano, E. Merlo and MD. Penta, Identifying clones in the linux kernel. In: First IEEE international workshop on source code analysis and manipulation, IEEE Computer Society Press, Los Alamitos, pp 92–100.2001.

[23] J. Mayrand, C. Leblanc and M. Merlo, Experiment on the automatic detection of function clones in a software system using metrics. in: Proceedings of the 12th International Conference on Software Maintenance (ICSM’96), Monterey, CA, USA, 1996, pp. 244–253.

[24] B. Wix and H. Balzert, Software wartung. Angewandte Informatik. BI Wissenschaftsverlag, 1988.

[25] T. Mens and T. Tourwe, A Survey of Software Refactoring. Transactions on Software Engineering, vol. 30, n° 2, p. 126-162, February, 2004.

[26] K. Allem and T. Mens, Refactoring des modèles : concepts et défis.

[27] S. Ducasse, M. Rieger and S. Demeyer, A language independent approach for detecting duplicated code. In: Proceedings ICSM’99: international conference on software maintenance, IEEE Computer Society Press, Los Alamitos, pp 109–118, 1999.

[28] F. Simon, F. Steinbruckner and C. Lewerentz, Metrics Based Refactoring. Proc. European Conf. Software Maintenance and Reeng, pp. 30-38, 2001.

[29] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, A quantitative evaluation of maintainability enhancement by refactoring. Software Maintenance, Proceedings International Conference, pp.576-585, 2002.

[30] I. Heitlager, T. Kuipers and J. Visser, A Practical Model for Measuring Maintainability. Software Improvement Group the Netherlands.

[31] P. Abran, R. Bourque and L. Tripp, Guide to the software engineering body of knowledge (ironman version). Technical report, IEEEComputer Society, 2004.

[32] K. Dubey and A. Rana, Assessment of maintenability Metrics for object oriented Software Systems. ACM SIGSOFT Engineering Notes septembre 2011 volume 36 num 5.

[33] M. Reichling and S. Cherfi, Qualité des Applications Web. Une approche centrée sur l'accessibilité. Université Paris 1.

[34] M. O'Keeffe and M. Cinneide, Search-based refactoring: an empirical study. Journal of Software Maintenance and Evolution: Research and Practice, 20(5):345–364, 2008.

[35] International Standards Organization. Software engineering - product quality - part 1: Quality model, ISO/IEC 9126-1 edition, 2001.

[36] A. Motoei, Applying ISO/IEC 9126-1 Quality Model to Quality Requirements Engineering on Critical Software. Department of Industrial and Management Systems Engineering School of Science and Engineering Waseda University.

[37] B. Baudry and Y. Traon, Measuring design testability of a uml class diagram. Inf.Softw.Technol.47 (13):859-879, 2005.ISSN:09505849. doi:http://dx.doi.org/ 10.1016/ j.infsof. 2005.01.006.

[38] S. Jungmayr, Testability measurement and software dependencies. Informatik fernuni, Universitätsstraße 1, Hagen, Germany 2002.

[39] S. Mouchawrab and Y. Labiche, A measurement frame-work for object-oriented software testability. Inf. Soft. Technol., 47(15):979{997, 2005. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2005.09.003.

[40] V. Robert and Binder, Design for testability in object-oriented systems. Commun. ACM, 37(9):87-101, 1994.ISSN 0001-0782. Doi: http//doi.acm.org/10.1145/182987.184077.

[41] M. Bruntink and A. van Deursen, Predicting class testability using object-oriented metrics, In SCAM, pages 136–145, 2004.

[42] S. Mouchawrab and Y. Labiche, A measurement framework for object-oriented software testability. Inf. Softw. Technol., 47(15):979–997, 2005.

[43] M. Chaumun, H. Kabaili, R. Keller and Lustman, F., "A change impact model for changeability assessment in object-oriented software systems. Science of Computer Programming, 45(2–3):155 – 174, 2002.

[44] R. Marinescu, Detecting Design Flaws via Metrics in Object-Oriented Systems. Politehnica university of Timisoara, Department of Computer Science, Romania.

[45] A. Kout, F. Toure and M. Badri, An empirical analysis of a testability model for object-oriented programs. SIGSOFT Softw. Eng. Notes, 36(4): 1–5, 2011.

[46] M. Badri, A. Kout and L. Badri, On the Effect of Aspect-Oriented Refactoring on Testability of Classes: A Case Study.

[47] Y. Zhou, H. Leung, A. Song, J. Zhao, H. Lu, L. Chen and B. Xu, An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems. SCIENCE CHINA Information Sciences, 55(12): 200– 215, 2012.

[48] C. Robert, “Stability” (1997). www.objectmentor.com.

[49] M. Bruntink and A. van Deursen, An empirical study into class testability. J. Syst. Softw., 79(9):1212–1232, 2006.

[50] Y. Dash, K. Dubey and A. Rana, Maintainability Prediction of Object Oriented Software System by Using Artificial Neural Network Approach.

[51] Y. Zhou and L. Hareton, Predicting object-oriented software maintainability using multivariate adaptive regression splines.

[52] M. Dagpinar and H. Jahnke, Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison.

[53] P. Hegedus and T. Bakota, Source Code Metrics and Maintainability: a Case Study. DOI 10.1007/978-3-642-27207-3_28, Online ISBN: 978-3-642-27207-3.

[54] S. Vaucher and H. Sahraoui, Étude de la changeabilité des systèmes orientés-objet. In Langages ET Modèles à Objets, 2008.

[55] Y. Ayalew and K. Mguni, An assessment of changeability of open source software. Computer and Information Science, 6(3):68–79, 2013.

[56] R. Marinescu. Measurement and quality in object-oriented design. Politehnica University of Timisoara, 2002.

[57] M. Ajrnal Chaumun, H. Kabaili, R. Keller, F. Lustman and G. Saint-Denis, Design properties and object-oriented software changeability. In Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European, pages 45–54, 2000.

[58] S. Mazeiar, L. Shimin and T. Ladan, A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. Department of Electrical and Computer Engineering University of Waterloo, Waterloo, Ontario.

[59] M. Badri, L. Badri and F. Touré, Empirical Analysis of Object-Oriented Design Metrics: Towards a New Metric Using Control Flow Paths and Probabilities. Software Engineering Research Laboratory, Department of Mathematics and Computer Science, University of Quebec at Trois-Rivieres, Québec, Canada.

[60] H. Kabaili, R. Keller and F. Lustman, Class cohesion as predictor of changeability: An empirical study. 2001.

[61] K. Aggarwal, Y. Singh, K. Arvinder and M. Ruchika, Empirical Study of Object-Oriented Metrics. School of Information Technology, GGS Indraprastha University, Delhi 110006, India.

[62] K. Dubey and A. Rana, Assessment of maintenability Metrics for object oriented Software Systems. ACM SIGSOFT Engineering Notes septembre 2011 volume 36 num 5.

[63] S. Dahmane, Étude et automatisation de refactorings pour le langage Delphi. Institut informatique, Université de Mons-Hainaut, 2004-2005.

[64] V. Basili, G. Caldiera and H. Rombach, The goal question metric approach. Encyclopedia of Software Engineering, pp. 528-532, John Wiley & Sons, Inc., 1994.

[65] H. Wang, Mémoire présenté comme exigence partielle de la maîtrise en informatique, les métriques appliquées dans la construction de logiciel. Université du Québec à Montréal.

[66] M. Alshayeb, Empirical Investigation Of Refactoring Effect On Software Quality. Volume 51, Issue 9, Pages 1319-1326, 2009, Elsevier.

[67] R. Marinescu, Detecting Design Flaws via Metrics in Object-Oriented Systems. Politehnica university of Timisoara, Department of Computer Science, Romania.

[68] S. Chidamber and C. Kemerer, A Metrics Suite for Object-Oriented Design. In IEEE Transactions on Software Engineering 20, 6(1994), pages 476–493, 1994.

[69] F. Toure, Indicateur de qualité pour les systèmes orientes objet: vers un modèle unifiant plusieurs métriques. Université du Québec à Trois-Rivières. Canada.

[70] M. Fowler, Refactoring: Improving the Design of Existing Code.

[71] A. Kaur, S. Singh, S. Kahlon and S. Parvinder, Empirical Analysis of CK & MOOD Metric Suit. International Journal of Innovation, Management and Technology, Vol. 1, No. 5, December 2010.