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
CATHERINE BÉLIVEAU

PRÉDICTION ET PRIORISATION DE L'EFFORT DE TEST DES SYSTÈMES ORIENTÉS OBJET À PARTIR DES MODÈLES

DÉCEMBRE 2020

Université du Québec à Trois-Rivières

Service de la bibliothèque

Avertissement

L'auteur de ce mémoire ou de cette thèse a autorisé l'Université du Québec à Trois-Rivières à diffuser, à des fins non lucratives, une copie de son mémoire ou de sa thèse.

Cette diffusion n'entraîne pas une renonciation de la part de l'auteur à ses droits de propriété intellectuelle, incluant le droit d'auteur, sur ce mémoire ou cette thèse. Notamment, la reproduction ou la publication de la totalité ou d'une partie importante de ce mémoire ou de cette thèse requiert son autorisation.

REMERCIEMENTS

Je tiens tout d'abord à exprimer de sincères remerciements à mon directeur de recherche Mourad Badri et à ma codirectrice de recherche Linda Badri pour avoir cru en moi et m'avoir épaulée tout au long de ce projet de maîtrise. Je les remercie pour leur grande disponibilité, leur encadrement et leur exceptionnel soutien autant sur les plans moral et professionnel qu'académique. Ils ont su me transmettre avec générosité leurs connaissances, leur expertise et ont su m'enseigner la rigueur et la détermination.

Je tiens également à remercier le conseil de recherche en sciences naturelles et en génie du Canada (CRSNG) qui a financé ce projet et m'a ainsi soutenue financière durant toute la première année de ma maîtrise.

Je tiens finalement à remercier tous les membres de ma famille pour leur soutien et leur compréhension et à exprimer spécialement ma reconnaissance à mon conjoint sans qui ce projet de maîtrise réalisé en parallèle à la fondation de notre famille n'aurait pas été possible.

RÉSUMÉ

Les tests logiciels constituent une étape cruciale (en termes d'assurance qualité) dans le processus de développement de logiciels. La phase de test demande cependant beaucoup d'efforts, de temps et de ressources. Il est donc important pour les développeurs de prédire, le plus tôt possible dans le processus de développement, l'effort requis pour tester un logiciel permettant ainsi d'effectuer une bonne planification des activités de test et une gestion optimale des ressources.

Les techniques de priorisation des cas de test sont des techniques permettant de gérer les ressources et de réduire le temps attribuable à la phase de test d'un logiciel. Ces techniques permettent l'organisation du développement des tests selon une priorité accordée aux cas de test basée sur un certain critère de priorisation. Il existe plusieurs objectifs à la priorisation des tests, toutefois le but premier de celle-ci est l'accélération de la phase de test (minimisation des efforts) et une augmentation du taux de détection des fautes.

Malheureusement, très peu de travaux ont été effectués sur la prédiction et la priorisation de l'effort de test en comparaison avec les nombreux travaux effectués sur la prédiction de l'effort de développement. Tout comme l'estimation de l'effort de développement, l'estimation et la priorisation de l'effort de test ne sont pas des tâches faciles. L'effort de test est une notion complexe qui est affectée par plusieurs facteurs. Dans ce projet, l'effort de test est considéré principalement sous la perspective de la construction des cas de test en termes de taille des suites de test JUnit.

Dans ce projet, nous proposons une approche permettant de prédire et de prioriser l'effort de test des systèmes orientés objet à partir de métriques simples et objectives extraites de modèles UML (essentiellement le modèle des cas d'utilisation et le diagramme de classes). Trois études exploratoires ont donc été menées ayant pour objectifs : (1) la priorisation des tests basée sur les cas d'utilisation, (2) la prédiction et la priorisation de l'effort de test à partir des modèles UML (diagrammes de classes UML en particulier), et (3) l'exploration de l'extension éventuelle de ces deux approches (1 et 2) aux systèmes orientés aspect. Plusieurs méthodes de modélisation ont été utilisées pour la construction des modèles de prédiction et de priorisation (algorithmes de régression et d'apprentissage d'ordonnancement).

ABSTRACT

Software testing is a crucial step (in terms of quality assurance) in the software development process. However, the software testing phase requires a lot of effort, time and resources. It is therefore important for developers to predict, as early as possible in the development process, the amount of effort required to test the software. This allows a better planning of the testing activities and an optimal management of resources.

The test case prioritization techniques are techniques that optimize resource management and reduce time attributable to the software testing phase. These techniques allow to organize tests development according to a prioritization criterion. There are several objectives for prioritizing tests, however, the primary goal of the prioritization is to accelerate the testing phase (reducing efforts) and increase the fault detection rate.

Unfortunately, little work has been done on test effort prediction and prioritization in comparison to the numerous works done on development effort prediction. Much like effort development estimation, estimating and prioritizing test efforts is not an easy task. Testing effort is a complex concept that is affected by several factors. In this project, testing effort is considered mainly from the perspective of building test cases in terms of the size of JUnit test suites.

In this project, the main goal is the development of an approach to predict and prioritize the testing effort in object-oriented systems based on simple and objective metrics extracted from explored UML models (essentially the use case model and the UML class diagram model). Three exploratory studies were conducted: (1) test prioritization based on use cases, (2) testing effort prediction and prioritization from UML models (particularly the UML class diagrams), and (3) the exploration of the possible extension of these techniques (1 and 2) to aspect-oriented systems. Several modeling methods were used for the construction of the prediction and prioritization models (regression and learning to rank algorithms).

TABLE DES MATIÈRES

1 Introduction 1

1.1. Problématique 1

1.2. Objectifs du projet 4

1.3. Organisation du mémoire 4

2 Priorisation des tests basé sur les cas d'utilisation 6

2.1 État de l'art 6

2.1.1 Introduction 6

2.1.2 Les techniques de priorisation basées sur les cas d'utilisation 7

2.2 Objectif 14

2.3 Technique de priorisation de l'effort de test à partir des métriques des DCU : étude exploratoire 15

2.3.1 Méthodologie 15

2.3.2 Limitations 20

2.4 Conclusion 20

3 Prédiction et priorisation de l'effort de test basé sur le diagramme de classes UML 22

3.1 État de l'art 22

3.1.1 Introduction 22

3.1.2 Les métriques du diagramme de classes (MDC) 22

3.1.3 Techniques de prédiction de l'effort de développement à partir du MDC 24

3.1.4 Techniques de prédiction de l'effort des tests à partir du DC 25

3.1.5 Techniques de priorisation des tests à partir à partir du DC 26

3.2 Objectif 26

3.3 Technique de priorisation de l'effort de test à partir des MDC : étude exploratoire 26

3.3.1 Métriques orientés objet du DC (MDC) 27

3.3.2 Métriques de taille des tests (MT) 27

3.3.3 Étude expérimentale 28

3.4 Conclusion 50

4 Prédiction et priorisation de l'effort de test étendues à la programmation orientée aspect 52

4.1 État de L'art 52

4.2 Objectif 53

4.3 Méthodologie 50

4.3.1 Démarche expérimentale 54

4.3.2 Métriques compilées 57

4.3.3 Hypothèses 59

4.4 Résultats 60

4.4.1 Résultats concernant les métriques globales avant et après implémentation de la POA 60

4.4.2 Résultats concernant les métriques unitaires avant et après implémentation de la POA 62

4.5 Conclusion 63

5 Conclusions 66

Liste de références bibliographiques 69

TABLEAUX

Tableau 2.1 MDCU utilisées pour l'estimation de l'effort de développement des SOO 14

Tableau 3.1 Statistiques concernant les 6 études de cas 28

Tableau 3.2 Corrélations de Spearman entre les MDC et MT des classes testées 31

Tableau 3.3 Corrélations de Kendall entre les MDC et MT des classes testées 31

Tableau 3.4 Corrélations de Spearman entre les MDC et MT des classes testées et non testées 32

Tableau 3.5 Corrélations de Kendall entre les MDC et MT des classes testées et non testées 32

Tableau 3.6 Résultats de l'ACP portant sur les MDC 33

Tableau 3.7 Résultats de l'ACP portant sur les MT 34

Tableau 3.8 Résultats des analyses de RL simples entre les MDC et MT d'intérêts 35

Tableau 3.9 Analyses de RL multiples concernant les MDC et MT d'intérêts 37

Tableau 3.10 Corrélations de Spearman et Kendall entre les scores et les métriques d'effort de test avec validation 10-folds 40

Tableau 3.11 Corrélations de Spearman et Kendall entre les scores et les métriques d'effort de test avec validation inter-projet 41

Tableau 3.12 Résultats d'un classement (1 fold) avec l'algorithme RecursiveLeastSquare sur les classes du projet IO avec validation 10-folds 46

Tableau 3.13 Résultats d'un classement (1 fold) avec l'algorithme OrdinaryLeastSquare sur les classes des projets Apache avec validation 10-folds 46

Tableau 3.14 Résultats d'un classement (1 fold) avec l'algorithme OrdinaryLeastSquare sur les classes du projet Jaqlib avec validation 10-folds 47

Tableau 3.15 Résultats du classement avec l'algorithme LASSO sur les classes du projet Nextgen avec validation inter-projet iValidator-Nextgen 48

Tableau 3.16 Corrélations de Spearman et Kendall entre les scores et les métriques d'effort de test avec validation inter-projet IO-iValidator 48

Tableau 4.1 DSS représentant le CU CreerNouvelleVente de l'étude de cas NextGen_OO 56

Tableau 4.2 DSS représentant le CU CreerNouvelleVente de l'étude de cas NextGen_OA 57

Tableau 4.3 Résultats concernant les métriques globales de modèles du système Nextgen avant et après aspectualisation 60

Tableau 4.4 Résultats concernant les métriques globales de classes du système Nextgen avant et après aspectualisation 61

Tableau 4.5 Résultats concernant les métriques globales de modèles du système Nextgen avant et après aspectualisation 61

Tableau 4.6 Résultats concernant les métriques de modèles unitaires du système Nextgen avant aspectualisation 62

Tableau 4.7 Résultats concernant les métriques de modèles unitaires du système Nextgen après aspectualisation 63

LISTE DES FIGURES ET ILLUSTRATIONS

Figure 2.1 Schématisation des DCU et de leurs DS sous forme d'arbre de la méthode de priorisation CoWTeSt 8
Figure 2.2 Exemple d'attribution de poids aux nœuds d'un CU de la méthode de priorisation CoWTeSt. 8
Figure 2.3 Attribution des poids des côtés de la technique de priorisation de Gantait et al. 10
Figure 3.1 Taux de classes testées (1) vs de classes non testées (0) 29

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ACP	Analyse en composantes principales
CT	Cas de test
CU	Cas d'utilisation
DC	Diagramme de classe
DCU	Diagramme de cas d'utilisation
DS	Diagramme de séquence
DSS	Diagramme de séquence système
GLOG	Laboratoire de Génie Logiciel, DMI, UQTR
LTR	*Learning to rank*
MDC	Métriques du diagramme de classes
MDCU	Métriques du diagramme de cas d'utilisation
MT	Métriques de test
OA	Orienté aspect
OO	Orienté objet
POA	Programmation orientée aspect
POO	Programmation orientée objet
PU	Processus unifié
RL	Régression linéaire
SOA	Systèmes orientés aspect
SOO	Systèmes orientés objet
TPC	Techniques de priorisation des cas de tests
UQTR	Université du Québec à Trois-Rivières
UML	*Unified Modeling Language*

CHAPITRE 1
INTRODUCTION

Ce chapitre d'introduction présente la problématique générale entourant la prédiction et la priorisation de l'effort de test des systèmes orientés objet (SOO) basées sur les modèles du langage de modélisation unifié UML. On présente, également, les différents objectifs de ce projet de maîtrise ainsi qu'une brève description des études menées dans le domaine. Enfin, l'organisation du présent mémoire y sera présentée.

1.1. Problématique

Dans le domaine du développement logiciel, une forte augmentation de la taille et de la complexité des logiciels a été constatée dans le temps [1]. De plus, un désir de diminution des délais (et coûts) de développement ainsi qu'une augmentation des exigences de qualité ont été observés [1]. Le *Rational Unified Process* (RUP) est l'implémentation la plus largement utilisée à ce jour du processus unifié (PU) pour le développement de logiciels orientés objet (OO) proposée par Ivar Jacobson, Grady Booch et James Rumbaugh [2]. Cette représentation comprend plusieurs phases dont une phase de test faisant suite à la phase d'implémentation du code et ayant comme objectif la détection des comportements fautifs d'un logiciel. Cette phase est l'une des phases les plus importantes (et coûteuses) du processus, et joue un rôle crucial dans le processus d'assurance qualité. Elle nécessite, toutefois, beaucoup d'efforts et de coûts en termes de temps et de ressources. En effet, les coûts engendrés par l'analyse et la conception (et conduite) des tests peuvent atteindre jusqu'à 50% de l'effort de développement total d'un système [1]. Ces faits ont amené certains chercheurs dans le domaine du génie logiciel à essayer de prédire le plus tôt possible les efforts (en termes de temps, ressources, lignes de codes, etc.) attribuables à cette phase, d'une part, et à gérer les tests dans le but de faire une détection des fautes la plus rapide et la moins coûteuse possible, d'autre part. La prédiction de l'effort de test tôt dans le processus de développement est pertinente du fait qu'elle permet de connaître d'une manière précoce les coûts associés à cette phase. Plus tôt les développeurs ont une idée des coûts associés à la phase de test, plus tôt il leur sera possible de mettre en place des stratégies de développement (et de test) afin de répondre aux exigences (coûts, qualité, temps, etc.). Malheureusement, il existe très peu de modèles (approches) développés à ce jour permettant de prédire l'effort de test tôt dans le processus de développement [3]. En fait,

plusieurs études de la littérature cherchent plus à prédire l'effort total de développement d'un système et non l'effort de test uniquement [3].

Les Techniques de Priorisation des Cas de test (TPC) sont des techniques permettant, entre autres, de réduire le temps et mieux gérer les ressources attribuables à la phase de test d'un logiciel [4]. Ces techniques permettent l'organisation du développement des tests en accordant une priorité aux Cas de Test (CT) selon certains critères de priorisation [5]. Ces critères peuvent être de différentes natures, par exemple, la couverture des tests, le taux de détection des fautes, l'historique des fautes, etc. Il existe, aussi, plusieurs objectifs à la priorisation des tests : la détection de fautes le plus tôt possible, une plus grande couverture de test le plus rapidement possible, la détection de fautes le plus rapidement possible, etc. Toutefois, le but premier de cette technique est l'accélération de la phase de tests et une augmentation du taux de détection des fautes.

Les approches d'estimation de l'effort de test et de priorisation peuvent être divisées en deux catégories de haut niveau : les approches basées sur le code source et les autres approches (basées sur les modèles, sur les humains (experts), etc.). Au niveau de la littérature, la majorité des approches de priorisation développées sont basées sur le code [6]. Les approches basées sur les modèles appartenant à la seconde catégorie sont toutefois très intéressantes du fait qu'elles sont généralement simples et utilisées tôt dans le processus de développement. Ces approches peuvent être, à leur tour, divisées en deux catégories : les approches basées sur les modèles du langage de modélisation UML (Unified Modeling Language) et les approches basées sur les modèles d'autres langages de modélisation (EFSM (Extended Finite State Machine), SDL (Specification and Description Language), etc.). Les approches basées sur les modèles UML sont particulièrement intéressantes, car le langage UML est le langage de modélisation le plus utilisé dans l'industrie du développement logiciel [7].

Parmi les modèles UML, le diagramme de Cas d'Utilisation (DCU) proposé par Jacobson [2] est un modèle particulièrement intéressant pour l'estimation de l'effort de test et la priorisation des tests. Ce diagramme est un modèle simple permettant la représentation des besoins fonctionnels d'un système. Le DCU ainsi que les diagrammes conceptuels directement dérivés de celui-ci (diagramme de séquence système, diagrammes d'interactions, diagramme d'états-transitions, diagramme d'activités) sont intéressants pour les techniques d'estimation de l'effort de test et de priorisation des tests, car ils sont disponibles très tôt dans le cycle de vie d'un logiciel. Le DCU est

encore plus intéressant sachant qu'il est le premier modèle à développer avec la modélisation métier et qu'il est le point central du développement dans le processus unifié, processus de développement le plus largement utilisé à ce jour. Le DCU guide ainsi tout le reste du processus de développement.

Le diagramme de classes UML (DC) semble aussi être une avenue intéressante pour la prédiction de l'effort de test et la priorisation des tests. En effet, il est le diagramme le plus utilisé par les entreprises de développement logiciel [8]. Ce digramme est la représentation graphique des objets (classes) d'un SOO ainsi que de leurs relations et attributs. De plus, la version conceptuelle de ce diagramme est le tout premier diagramme à être produit dans le PU [2]. Cette version est ensuite revue, en phase de conception, pour devenir une version objet permettant de guider l'implémentation du code.

Bien que ces deux modèles UML semblent d'un grand intérêt pour le développement de techniques de prédiction de l'effort de test et de priorisation des tests, peu de techniques de prédiction et de priorisation ont été développées à ce jour à partir de ces derniers.

Au niveau de la littérature, il est connu que la programmation orientée objet (POO) a apporté de grands bénéfices au domaine de la programmation logicielle [9]. La POO permet plusieurs avantages tels que la réutilisation du code, la réduction des délais et des coûts de développement et de maintenance. Toutefois, dans certains cas précis, elle ne fournit pas de solution satisfaisante en termes de réutilisation. Ce problème, comme décrit par Gregor Kiczales et son équipe [10], est rencontré lorsque l'on est en présence de préoccupations transverses dans un programme. Ces dernières sont des extensions comportementales enchevêtrées ou dispersées dans le code d'un système telles que la journalisation, l'authentification ou la synchronisation. La dispersion se produit quand le code d'une même fonctionnalité est éparpillé dans plusieurs classes d'un système tandis que l'enchevêtrement se produit lorsque plusieurs contraintes d'intégrité sont contenues dans une même classe. Dans les deux cas, le code impliqué dans ces préoccupations transverses est difficilement modulable et réutilisable en POO. Pour pallier cette faiblesse, Gregor Kiczales et son équipe ont développé une nouvelle technique de programmation dans les années 1990, la programmation orientée aspect (POA). Cette technique de programmation permet l'encapsulation des préoccupations transverses dans des modules réutilisables appelés *aspects*. Ces modules sont ensuite greffés à des endroits précis dans le code d'un système OO. Depuis son apparition, cette technique de programmation a suscité beaucoup d'intérêt au niveau de la recherche

dans le domaine du génie logiciel. Toutefois, à notre connaissance, il n'existe pas de modèle d'estimation de l'effort de test et de priorisation développés pour cette technique de programmation.

1.2. Objectifs du projet

Les faits énoncés dans la problématique ci-dessus démontrent que la prédiction de l'effort de test et la priorisation des tests à partir des modèles UML sont des sujets intéressants et peu explorés dans le domaine du génie logiciel. C'est dans ce contexte que s'inscrivent les objectifs de notre maîtrise. Le but ultime de ce projet est le développement d'une approche de prédiction et de priorisation de l'effort de test des SOO à partir de modèles UML (les diagrammes de cas d'utilisation et les diagrammes de classes).

Ce projet comprend 3 objectifs plus précis : (1) le développement d'une technique de priorisation des tests basée sur les cas d'utilisation (CU), (2) le développement d'une technique de prédiction et de priorisation de l'effort de test à partir du diagramme de classes UML, et (3) l'exploration de l'extension éventuelle de ces techniques aux systèmes orientés aspect (SOA).

1.3. Organisation du mémoire

Ce mémoire est composé de 5 chapitres. Le premier chapitre est une introduction générale relative à la problématique qui nous intéresse à savoir la prédiction et à la priorisation des tests dans les SOO.

Les chapitres qui suivent présentent les travaux de recherche effectués, nous permettant ainsi de mieux situer nos objectifs. Ils sont organisés comme suit : le chapitre 2 présente l'étude menée sur la priorisation des tests à partir des CU. Le chapitre 3 présente l'étude menée sur la prédiction et priorisation de l'effort de test à partir du diagramme de classes UML. Le chapitre 4 présente l'étude préliminaire portant sur l'exploration de l'extension des deux approches précédentes aux SOA et enfin, le chapitre 5 présente les conclusions de ce mémoire.

Les chapitres 2, 3 et 4, portant sur les études menées, sont tous subdivisés en 4 sous-sections identiques soit : l'état de l'art, les objectifs spécifiques, les expérimentations effectuées ainsi que leurs résultats et les conclusions tirées.

CHAPITRE 2
PRIORISATION DES TESTS BASÉE SUR LES CAS D'UTILISATION

2.1 État de l'art

2.1.1 Introduction

Les TPC sont des techniques d'organisation du développement des tests avec pour objectif de réduire les coûts relatifs à la phase de test d'un logiciel. Ces techniques permettent d'accorder une priorité aux CT selon certains critères de priorisation.

Plus spécifiquement, le problème de la priorisation des tests a été défini par Rothermel et al. [5] comme ceci :

$$\text{Identifier } T' \in PT \; tel \; que \; (\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$$

T dans l'expression précédente, représente une suite de tests, PT l'ensemble des permutations de T, donc l'ensemble de cas de test et f une fonction.

Tel que mentionné dans l'introduction de ce mémoire, il existe plusieurs objectifs pouvant être atteints par la priorisation des tests, toutefois, le but ultime de cette technique est l'accélération de la phase de tests.

Plusieurs techniques de priorisation des cas de test ont été développées au fil du temps par différents chercheurs. La majorité de ces techniques sont basées sur le code et peuvent être classifiées selon un certain nombre de caractéristiques [4, 6, 11-16]. Une récente étude de Catal et al. [17] propose la catégorisation de ces techniques en 15 classes : les approches à réseau bayésien, les approches basées sur le changement, la complexité, le coût, la couverture, le client, la distribution, les fautes, les algorithmes génétiques, les graphes, les algorithmes gloutonnes, l'historique, les modèles, les besoins et les autres approches.

Parmi ces approches, la catégorie basée sur les modèles est particulièrement intéressante car elle permet une priorisation tôt dans le processus de développement. Elle est, par ailleurs, généralement compréhensible en raison de l'utilisation des modèles. Toutefois, peu de techniques ont été développées dans cette catégorie au niveau de la littérature. Ces techniques représentent

effectivement environ 12% des techniques étudiées dans la littérature entre les années 2000 et 2012 selon Catal et al. [6].

La majorité des approches basées sur les modèles dans la littérature sont basées sur les modèles UML. Toutefois, quelques approches ont aussi été développées à partir d'autres modèles n'appartenant pas au langage UML, par exemple [12, 13].

Comme mentionné précédemment, le DCU et les digrammes conceptuels directement dérivés de celui-ci sont des modèles UML particulièrement intéressants pour les techniques d'estimation de l'effort de test et de priorisation des tests, car ils sont disponibles très tôt dans le cycle de vie d'un logiciel.

2.1.2 Les techniques de priorisation basées sur les cas d'utilisation

Selon une étude portant sur une revue systématique de la littérature (non publiée) menée dans le contexte de ce travail, 12 articles scientifiques ont été publiés à ce jour portant sur des techniques de priorisation des CT basées sur les DCU ou sur les modèles dérivés de ceux-ci. Les librairies et moteurs de recherche utilisés afin de mener cette étude sont les suivants : IEEE Xplore, ACM Digital Library, ScienceDirect, Web of Science et Google Scholar. Un résumé de ces 12 différentes approches est présenté dans ce qui suit.

Basanieri et al. [18] proposent une stratégie pour la sélection et la priorisation des CT nommée CoWTeSt (*Cost Weighted Test Strategy*). Ils utilisent, comme l'illustre la figure 2.1, un DCU du système, organisé en arbre afin de produire les différents diagrammes de séquence (DS) organisés aussi en arbres. La méthode UIT (*Use Interaction Test*) est ensuite utilisée afin d'obtenir les différents CT. Afin de prioriser les CT obtenus, leur effort unitaire est calculé à partir du poids de la somme des nœuds. La valeur du poids d'un nœud est une valeur entre 0 et 1, donnée selon son importance (en termes de complexité, fréquence d'utilisation, etc.). Cette valeur est donnée de manière à ce que la somme de tous les enfants d'un nœud soit égale à 1. La figure 2.2 de ce chapitre illustre un exemple d'attribution de poids à 6 nœuds selon la méthode précédemment décrite. Les CT sont ensuite classés selon ces poids.

Figure 2.1

Schématisation des DCU et de leurs DS sous forme d'arbre de la méthode de priorisation CoWTeSt

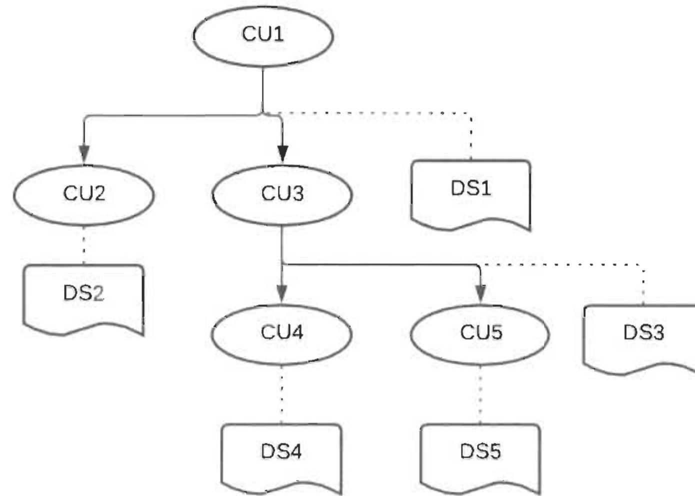
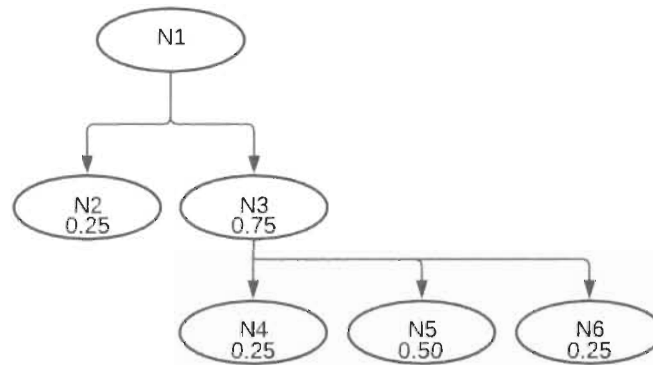


Figure 2.2

Exemple d'attribution de poids aux nœuds d'un CU de la méthode de priorisation CoWTeSt



Kundu et al. [19] ont développé l'approche STOOP (*System Testing for Object-Oriented systems with test case Prioritization*) permettant de générer et de prioriser les CT d'un système à partir d'un Graphe de Séquences GS bâti à partir des différents DS d'un système. Ce graphe orienté est formé d'arcs et de nœuds. La priorisation est faite selon trois métriques extraites du GS : le poids des messages (*message weight*), le poids d'une arête (*edge weight*) et la somme des poids des messages (*Sum of message weights*). Le poids des messages est associé à une arête du graphe et correspond au nombre de messages entre ses nœuds de début et de fin. Le poids d'une arête

correspond au nombre de chemins (*paths*) qui incluent cette arête. La somme des poids des messages correspond à la sommation des poids des messages de toutes les arêtes impliquées dans un chemin.

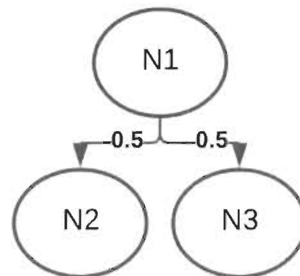
Sapna et al. [20] présentent une technique de priorisation à deux niveaux : la priorisation des CU à partir du DCU et la priorisation des scénarios de test à partir du diagramme d'activité. Les métriques utilisées par la première priorisation sont les suivantes : le poids des acteurs, le poids de la priorité du client et le poids de la priorité technique. Le poids de la priorité des acteurs est une mesure subjective donnée par le développeur, le poids de la priorité du client est une mesure subjective donnée par le client, le poids de la priorité technique est calculé à partir de plusieurs métriques techniques (nombre d'acteurs (Na), nombre de fois que le CU apparaît dans le modèle (Nt), nombre de CU inclus dans le CU (Nin), nombre de CU qui incluent ce CU (Nind), nombre de CU étendus (Nex), nombre de CU qui étendent le CU (Nexd) et nombre de CU qui héritent du CU (Ninh)). Pour le second niveau de priorisation, les diagrammes d'activité sont convertis en un graphe à l'aide de l'algorithme de recherche en profondeur. Ensuite, les poids de tous les nœuds et arrêtes sont calculés. Une priorité de 3 est accordée aux nœuds de bifurcation (fork) et d'union (join), de 2 pour les nœuds de fusion (merge) et de 1 pour les actions. Le poids des arrêtes est calculé selon le nombre de dépendances entrantes et sortantes des nœuds de début et de fin. Finalement, le poids de chaque scénario correspond à la sommation du poids des nœuds et arrêtes impliqués dans celui-ci.

Gantait [21] propose une approche de génération et de priorisation des CT basée sur les diagrammes d'activité. Les flux d'activité des diagrammes d'activité représentent les différents CT. Leur technique de priorisation propose de prioriser les flux d'activité en fonction de leur couverture de transitions dans le diagramme d'activité. Leur technique consiste, dans un premier temps, à trouver le nombre minimum de flux couvrant toutes les arrêtes. S'il y a plus d'une combinaison, le poids des flux est calculé. Ils définissent le poids d'un flux par la probabilité d'exécution d'un flux pouvant être donnée par un expert dans le domaine ou en le calculant à partir du poids des arrêtes. Le poids d'un côté p représente un nombre décimal entre 0 et 1 qui indique la probabilité de la transition. De ce fait, si un nœud a 1 côté sortant le poids du côté sortant est de 1. Si un nœud à n côté sortant, le poids de chaque côté sortant est de 1/n. La figure 2.3 présent un

exemple d'attribution de poids aux côtés sortant d'un nœud. Le poids d'un flux se calcul donc en multipliant tous les poids des arrêtes.

Figure 2.3

Attribution des poids des côtés de la technique de priorisation de Gantait et al.



Kashyap et al. [22] présentent un modèle d'approche permettant de générer les CT à partir d'un diagramme d'états-transitions structuré à l'aide du processus MMMP (*Markov Modulated Markov Process*) qui modélise l'utilisation des données du système (actions des utilisateurs et réponses du système). Ils proposent ensuite de prioriser les CT générés en les triant à partir d'une fonction de vraisemblance (*likelihood function*).

Kaur et al. [23] proposent une approche de priorisation des scénarios de test générés à partir du diagramme d'activité. Ils proposent de convertir le diagramme d'activité en graphe de flux de contrôle et de générer ensuite les scénarios de test à partir de celui-ci. La priorisation de ces scénarios est basée sur la complexité des chemins en utilisant les concepts suivants : longueur des chemins, métrique d'information de flux, nœud prédicat et conditions de couverture.

Sharma et al. [24] ont développé une technique pour la priorisation des CT dérivés des diagrammes d'activité et des diagrammes d'états-transitions. Leur méthodologie est une extension de leurs précédents travaux [25]. Ils proposent de transformer respectivement les diagrammes d'activité et d'états-transitions en graphe de flux de contrôle et graphe de dépendance. Leur technique de priorisation consiste à utiliser ces seconds graphes ainsi que les concepts de métriques de flux d'information de base (*basic information flow metrics*), de pile et d'algorithme génétique afin de calculer le poids de chaque CT et de les trier selon ce poids.

Jena et al. [26] proposent une technique de génération et priorisation des CT à partir des diagrammes de séquence et d'interactions. Ils utilisent ces diagrammes afin de produire deux

seconds diagrammes : un graphe d'interaction (IG) et un graphe de séquence de dépendances (MSDG). Ces graphes sont ensuite combinés afin de produire un graphe intermédiaire, le graphe de séquence d'interactions (SIG). C'est à partir de ce dernier graphe, que les différents CT sont produits. Pour la priorisation des CT, ils transforment à nouveau le graphe SIG en un graphe de dépendance (DG). Ensuite, ils assignent un poids à chaque nœud du graphe en calculant l'impact de celui-ci à l'aide d'un algorithme utilisant le concept de *backward slicing* [27]. Le poids des arrêtes est calculé à partir de la règle 80-20, c'est-à-dire que le coût de 4 (80%) est attribué à une décision vraie et le coût de 1 (20%) est attribué à une décision fausse. Le poids de chaque chemin est calculé par la sommation du poids de ses nœuds et arrêtes. La priorisation est ensuite effectuée à partir du poids de ces chemins.

Swain et al. [28] ont développé une technique de génération et priorisation dans laquelle les CT sont d'abord générés à partir d'un arbre de flux de tests (TFT), lui-même généré à partir des diagrammes de communication et d'activités du système. Ensuite, le TFT est converti en arbre COMMACT (*COMMunication ACTivity tree*) et la priorisation des CT est faite à partir du poids des chemins de cet arbre. Le poids des chemins est calculé par la sommation du poids des arrêtes et des nœuds. Le poids des nœuds est basé sur la complexité de ceux-ci. Les poids 2 et 1 sont donnés respectivement aux nœuds de fusion (merge) et aux nœuds d'action/activité. Le poids des arrêtes correspond au produit des dépendances entrantes du nœud de départ et des dépendances sortantes du nœud de fin.

Rhmann et al. [29] utilisent le concept d'algorithme génétique afin de prioriser les CT obtenus à partir d'un graphe de flux d'activité (AFG). Une valeur (*fitness value*) calculée à partir de la sommation des informations de flux de chaque nœud et de la couverture des nœuds de décision est attribuée à chaque CT, qui sont ensuite priorisés par rapport à cette valeur.

Wang et al. [30] proposent une approche utilisant le diagramme d'activité et un algorithme génétique hybride (HGA) qui combine l'algorithme génétique avec l'optimisation PSO (*Particle Swarm Optimization*) afin de prioriser les scénarios de test. L'approche développée permet plus précisément de trouver le chemin critique à partir d'un diagramme d'activités devant être testé en premier. Les différentes étapes utilisées pour identifier le chemin critique sont : convertir le diagramme d'activité en diagramme de flux de contrôle (CFG), générer tous les chemins

indépendants et non-redondants en utilisant la méthode de recherche en profondeur (DFS) et finalement trouver le chemin critique en utilisant l'algorithme HGA.

Bhuyan et al. [31] proposent une approche de priorisation des scénarios de test à partir des diagrammes de CU et d'activité. Les différentes étapes de leur approche sont les suivantes : 1) construire le DCU, 2) spécifier les chemins principaux et alternatifs, 3) identifier les scénarios, 4) générer le diagramme d'activités, 5) convertir le diagramme d'activité en diagramme de flux de contrôle (CFG), 6) générer les chemins indépendants à partir du CFG, 7) dériver la complexité de chaque chemin, et 8) prioriser les scénarios à partir du poids des chemins, de la complexité des chemins et de la couverture des nœuds. Le poids des chemins est calculé à partir de la sommation des poids de leurs arrêtes. Ces poids sont attribués selon le type de couplage des arrêtes. La complexité d'un chemin représente la somme des valeurs de complexité (CV) de ses nœuds. La valeur de complexité représente la valeur de contribution d'un nœud divisée par le nombre de nœuds affectés. La valeur de contribution d'un nœud correspond au nombre de fois que le nœud est rencontré.

En observant les différentes études citées ci-haut, on constate que peu de techniques de priorisation utilisent le DCU et que la majorité des techniques de priorisation développées à ce jour sont basées sur le diagramme d'activité. Ces constats concordent avec les résultats d'une récente revue de littérature portant sur les techniques de priorisation des tests basées sur les modèles [32]. Dans cette étude, les auteurs ont déterminé que 35% des techniques de priorisation utilisant les modèles développés en 2018 étaient basées sur le diagramme d'activité et que seulement 6% de ces techniques étaient basées sur le DCU.

2.1.3 Métriques des DCU (MDCU)

Peu de métriques simples extraites des DCU ont été utilisées à ce jour pour l'estimation de l'effort de test ou la priorisation des CT. Toutefois, plusieurs recherches ont été menées au fil des années portant sur des techniques d'estimation de l'effort de développement à partir des DCU. Le tableau 2.1 comprend ces différentes études. En analysant ces études, il est possible de constater que plusieurs métriques complexes et/ou subjectives ont été utilisées pour l'estimation de l'effort de développement. Le poids des acteurs (Wactors), le poids du cas d'utilisation (WUseCases), le

facteur environnemental (Ef) et le facteur de complexité technique présentés dans l'étude de Karner 1993 [33] et repris dans plusieurs études en sont des exemples. Toutefois, certaines métriques objectives et simples ont été aussi utilisées à plusieurs reprises par ces études. Le nombre de transactions (NT) et le nombre d'objets (*Entity Objects*, *Objects*) sont ceux le plus souvent utilisés.

Une récente étude de Badri et al. [34], présentée dans le tableau 2.1, propose une technique d'estimation de l'effort de développement des SOO à partir de métriques objectives et simples de DCU. Elle porte plus précisément sur la prédiction de la taille du code source en termes de lignes de code à partir des seules métriques de CU et sur la comparaison de celle-ci par rapport à la technique bien connue Use Case Points (UCP) [33]. La plupart des métriques du DCU, utilisées dans leur étude, ont été inspirées de leurs études antérieures [3] portant sur la prédiction de la taille des suites de test JUnit à partir des métriques du DCU. JUnit est un Framework permettant le développement et l'exécution de tests unitaires (TU) pour les classes Java. Ils ont utilisé différentes techniques de modélisation afin de construire des modèles de prédiction. Plus précisément, les régressions logistique univariée et linéaire simple ont tout d'abord été utilisées pour évaluer l'effet de chacune des métriques sur SLOC (nombre de lignes de code source). Ensuite, les régressions logistiques multivariée et linéaire multiple ont été utilisées pour vérifier l'effet combiné des MDCU sur SLOC. Des algorithmes d'apprentissage automatique ont également été utilisés. Les résultats de leur étude, avec les modèles de régression (logistique univariée et multivariée ainsi que linéaire simple et multiple) et les modèles utilisant des algorithmes d'apprentissage automatique (k-NN, Naïve Bayes, Perceptron, C4.5 et random forest), démontrent que les MDCU sont plus précis pour la prédiction de SLOC que la méthode UCP.

Tableau 2.1

MDCU utilisées pour l'estimation de l'effort de développement des SOO

Auteur	Année	Métriques
Karner	1993	UCP(Wactors, WUseCases(NT), Ef, TCF)
Mohagheghi et al	2005	AUCP(Wactors, WUseCases(NT), Ef, TCF, nb nouveaux acteurs, nb modifications cas d'utilisation)
Edward Caroll	2005	IUCP(Wactors, WUseCases(NT), Ef, TCF, Tlevel, Elevel)
Anda et al.	2005	NT
Sergey Diev	2006	UCPm(UAW, UUCW(NT), TCF, EF, BSC)
Braz and Vergilio	2006	USP(USP actor, USP Precondition, USP Exception, USP postcondition, USP scenario, TF, EF)
Braz and Vergilio	2006	FUSP(fuzzy numbers for USP)
Issa et al.	2006	Object point extraction using use case model (OP)
Robiolo et al.	2007	NT, Entity objects
Robiolo et al.	2008	NT, Entity objects
Robiolo et al.	2009	NT, Paths
Wang et al	2009	EUCP(UCP + fuzzy sets + BNN for effort)
Periyasamy et al.	2009	e-UCP(AW(NA), UCW(NA/NT), UCNP(Input, output, predicats, actions in successful scenario, exceptions), TF, EF)
Ochodek et al.	2010	SUCP(UCP(NT, semantic transactions, steps, TCF, EF))
Ochodek et al.	2011	TTpoints(Core_Actions, Objects, Actors)
Kirmani, M.M. and Wahid, A	2015	Re-UCP(UUCP(NT), TCF, ECF)
Badri et al.	2013	NS, NEO, NIM, NIC
Badri et al.	2017	NIC, NEO, NS, NT

2.2 Objectif

À la lumière des informations données précédemment, on comprend que le développement d'une technique de priorisation de l'effort de test pour les systèmes OO à partir des CU serait très pertinent et utile pour les développeurs d'applications. Notre objectif, dans ce contexte, est donc l'exploration et la proposition d'une technique simple de priorisation des tests basée sur les CU. Plus précisément, l'objectif de nos travaux consiste à étendre la méthodologie de Badri et al., présentée précédemment, afin de proposer une technique de priorisation basée sur les MDCU.

2.3 Technique de priorisation de l'effort de test à partir des métriques des DCU : étude exploratoire

2.3.1 Méthodologie

2.3.1.1 Métriques utilisées

2.3.2.1.1 MDCU

Les MDCU utilisées dans la méthodologie de cette étude exploratoire sont celles utilisées dans l'étude de Badri et al. 2017 :

- *NIC (nombre de classes impliquées) :* cette métrique correspond au nombre de classes conceptuelles impliquées dans un CU.
- *NEO (nombre d'opérations externes) :* cette métrique correspond au nombre d'opérations systèmes liées au CU. Ces opérations sont générées par des sources externes au système. Elles sont facilement identifiables à partir du diagramme de séquence système et sont généralement assignées à des classes contrôleurs.
- *NS (nombre de scénario) :* cette métrique correspond au nombre de scénarios du CU incluant le scénario principal.
- *NT (nombre de transactions) :* cette métrique correspond au nombre total de transactions comprises dans un CU. Une transaction correspond à un évènement se produisant entre un acteur et le système.

Ces métriques peuvent être directement extraites des modèles de CU de la phase d'analyse du cycle de développement logiciel. La métrique NIM, introduite dans l'étude de Badri et al. 2013, a été exclue de la méthodologie de ce chapitre tout comme pour l'étude de Badri et al. 2017, car cette métrique peut seulement être extraite lors de la phase de conception du cycle de développement.

2.3.2.1.2 Métriques de test (MT)

Les MT explorées dans cette étude sont, tout d'abord, deux métriques utilisées dans Badri et al. 2013 pour quantifier la taille des suites de test JUnit (TLOC et TASSERT) ainsi qu'une autre métrique (TINVOK) utilisée dans [35] et permettant de capturer la dimension d'interactions entre les classes/objets d'un système.

Voici la description de chacune de ces MT :

- *TLOC (nombre de lignes de code de test)* : cette métrique correspond au nombre de lignes de code de test d'un CT JUnit correspondant à un CU.
- *TASSERT (nombre de méthodes* d'assertion*)* : cette métrique correspond au nombre de méthodes d'assertion utilisées dans un CT correspondant à un CU. Une méthode d'assertion est une méthode de vérification utilisée à l'intérieur d'un TU.
- *TINVOK (nombre de méthodes invoquées)* : cette métrique correspond au nombre de méthodes invoquées dans un CT correspondant à un CU. Elle permet de capturer les dépendances d'un CT.

2.3.1.2 Analyses statistiques

La première étape de la méthodologie de cette étude exploratoire correspond à des analyses statistiques.

Tout d'abord, cette étape consiste à calculer les quatre MDCU et trois MT de plusieurs études de cas afin d'effectuer deux types d'analyses statistiques : des analyses de corrélations et des analyses de régression. Toutefois, contrairement à la méthodologie de Badri et al. 2017, les sept métriques calculées seront extraites à partir de véritables diagrammes provenant de la phase d'analyse des études de cas et non par des diagrammes obtenus par rétro-ingénierie. L'objectif de cette dernière pratique est d'éviter un biais possible dans les modèles de la phase d'analyse; modèles obtenus par rétro-ingénierie (proximité avec le code source).

2.3.2.2.1 Analyses en Composantes Principales (ACP)

L'ACP est une technique fréquemment utilisée en génie logiciel afin d'explorer des jeux de données multidimensionnelles et d'évaluer la contribution des métriques dans ces dimensions.

Une ACP sur les MDCU calculées est suggérée afin de vérifier ses résultats sur le jeu de données calculé.

De plus, dans notre étude exploratoire, une nouvelle MT (TINVOK) a été introduite. Une ACP est alors nécessaire pour l'ensemble des MT choisies pour cette étude exploratoire. Le but est de vérifier si les MT du nouvel ensemble sont indépendantes ou si certaines capturent les mêmes dimensions.

Ces deux ACP sont effectuées dans le but d'orienter les phases suivantes de notre approche.

2.3.2.2.2 Analyses de corrélations

Les relations entre les MDCU et les MT peuvent être évaluées à l'aide d'analyses de corrélations. Ces analyses de corrélations peuvent être effectuées facilement à l'aide du logiciel XLSTAT [36]. L'objectif final de ces analyses est la vérification des hypothèses suivantes pour chaque couple de métriques $<MDCU_i, MT_i>$:

- Hypothèse : Il existe une relation entre la métrique $MDCU_i$ et la métrique MT_i.
- Hypothèse nulle : Il n'existe pas de relation entre la métrique $MDCU_i$ et la métrique MT_i.

Afin de vérifier ces hypothèses, deux techniques d'analyse de corrélations seront utilisées : Spearman et Pearson.

2.3.2.2.3 Analyses de régression

Le second objectif des analyses statistiques consiste à vérifier si les MDCU combinés ou non sont de bons prédicteurs de l'effort de test. Des analyses de régression simples et multiples sont nécessaires afin de vérifier le potentiel de prédiction d'une métrique spécifique à partir d'une ou plusieurs autres métriques. La régression logistique univariée ainsi que la régression linéaire

simple sont donc nécessaires afin de vérifier le potentiel de prédiction de chaque métrique individuelle. Par ailleurs, la régression logistique multivariée ainsi que la régression linéaire multiple sont également retenues pour vérifier le potentiel de prédiction des combinaisons de métriques.

En ce qui concerne les régressions logistiques, il est nécessaire que les métriques dépendantes (MT) soient transformées en des versions binaires, c'est-à-dire qu'elles peuvent avoir seulement deux valeurs différentes (0 ou 1). Pour ce faire, la technique de Badri et al. est suggérée [3]. Cette dernière technique consiste à classifier les métriques dépendantes en deux catégories :

Complexe : Cette catégorie inclut les métriques pour lesquelles : MT_i .>= valeur moyenne de. MT_i.

Simple : Cette catégorie inclut les métriques pour lesquelles : MT_i .<= valeur moyenne de. MT_i.

L'objectif de la régression simple est de vérifier les hypothèses suivantes :

- Hypothèse : La variable $MDCU_i$ permet de prédire la variable MT_i.
- Hypothèse nulle : La variable $MDCU_i$ ne permet pas de prédire la variable MT_i.

Les hypothèses à vérifier par la régression multiple sont les suivantes :

- Hypothèse : La combinaison de variables {$MDCU_i$, $MDCU_j$,..} permet de prédire la variable MT_i.
- Hypothèse nulle : La combinaison de variables {$MDCU_i$, $MDCU_j$,..}ne permet pas de prédire la variable MT_i.

2.3.2.2.4 Algorithmes d'apprentissage d'ordonnancement

L'apprentissage d'ordonnancement (*learning to rank* (LTR)) est un domaine de recherche récent. Il consiste à optimiser une fonction d'ordonnancement de manière automatique à l'aide d'algorithmes d'apprentissage. Son principal domaine d'application est la recherche d'informations [37]. L'apprentissage d'ordonnancement est le principe responsable de l'indexage

lors d'une recherche sur le web. Bien que l'apprentissage d'ordonnancement soit très utilisé dans le domaine de la recherche d'informations, il est aussi de plus en plus utilisé dans d'autres domaines ayant recours à l'ordonnancement. Cet algorithme a, entres autres, été utilisé dans le domaine des langues pour l'ordonnancement des traductions potentielles d'une phrase traduite automatiquement [38], dans l'ordonnancement d'appariement de questions [37] et dans le domaine du génie logiciel pour la localisation de fautes [39].

Les méthodes d'apprentissage d'ordonnancement sont divisées en trois catégories : les approches par points, les approches par paires et les approches par listes.

Les méthodes par points utilisent chaque document en entrée du système d'apprentissage de manière indépendante. Les principales approches appartenant à cette catégorie sont : MART [40], Prank [41], OC SVM [42], Subset Ranking [43], McRank [44].

Les méthodes par paires utilisent plutôt les documents par paires en entrée du système d'apprentissage. L'objectif est de comparer le résultat d'une requête d'un document par rapport à un autre document. Les principales approches appartenant à cette catégorie sont les suivantes : Ranking SVM [45], RankBoost [46], RankNet [47], IR SVM [48], GBRank [49] et LambdaMART [50].

La dernière catégorie de méthodes utilise plutôt une liste ordonnée de résultats, associée à chaque requête d'entrée. Cette liste est considérée comme la référence à prédire. ListNet [51], AdaRank [52], SVM MAP [53], Coordinate Ascent [54], ListMLE [55] et SoftRank [56] sont les principaux algorithmes appartenant à cette catégorie.

La méthodologie de cette étude exploratoire propose d'utiliser les MDCU les plus prometteuses pour la prédiction de l'effort de test suite aux étapes d'analyses précédentes et de les utiliser dans des algorithmes automatiques d'ordonnancement dans le but de construire un modèle de priorisation des tests à partir des MDCU. Plus précisément, deux types d'algorithmes d'apprentissage automatique sont proposés : des algorithmes de régression tels que la régression linéaire par les moindres carrées (*Ordinary Least Squares),* la régression Ridge (*ridge regression*), la régression linéaire par les moindres carrés récursifs (*Recursive Least Squares*), la régression Lasso [57] et la régression avec *gradient boosting* [58] et des algorithmes LTR tels que MART

[40], Ranknet [47] et Coordinate Ascent [54]. Ces algorithmes peuvent être implémentés en langage Java à l'aide des librairies Smile [59] et Ranklib [60]. Dans notre cas, ces algorithmes ont été implémentés par un étudiant du GLOG (Laboratoire de Génie Logiciel, DMI, UQTR) et utilisés lors de l'étude exploratoire présentée dans le chapitre 3 de ce mémoire. Certains algorithmes semblent prometteurs concernant la priorisation de l'effort de test à partir de métriques extraites à partir des modèles.

2.3.2 Limitations

La méthodologie que nous avons proposée dans ce chapitre, n'a malheureusement pas été validée empiriquement, en raison d'un certain nombre de problème rencontrés. L'un de nos objectifs était d'utiliser de véritables modèles provenant de la phase d'analyse d'étude de cas pour l'extraction des MDCU afin d'exclure le biais provenant de la création des modèles par rétro-ingénierie. Une recherche rigoureuse à partir des librairies web de projets libres (*open source projects*) a été effectuée afin d'obtenir des études de cas de qualité comprenant des modèles de CU et modèles de séquences créés par les auteurs de ces études en phase d'analyse. Toutefois, cette recherche n'a malheureusement pas été fructueuse. Effectivement, une seule étude de cas a été retenue au cours de cette recherche. En majorité, les autres études de cas analysées ne comprenaient aucun des deux diagrammes nécessaires à notre étude; l'un ou l'autre des deux diagrammes était absent, ou bien l'un ou l'autre ou les deux diagrammes n'étaient pas complets ou de qualité.

Cette limitation majeure a fait en sorte qu'il n'a pas été possible de valider la méthodologie proposée dans ce chapitre.

2.4 Conclusion

Dans cette partie, une méthodologie très intéressante a été présentée pour la priorisation de l'effort de test dans les SOO utilisant des techniques déjà explorées et des métriques testées pour la prédiction de l'effort de test. Toutefois, une grande limitation a aussi été rencontrée au cours de cette étude au niveau du faible nombre d'études de cas libres comprenant des modèles d'analyses de qualité. La méthodologie de ce projet n'a, de ce fait, pas pu être validée.

Dans le futur, il serait intéressant de valider l'approche présentée sur différents systèmes de domaines et complexités différentes. Afin de pallier au manque d'études de cas issues de librairies libres comprenant les deux diagrammes utilisés par notre méthodologie, il serait intéressant de demander à des entreprises appartenant à différents domaines des études de cas de qualité comprenant une phase d'analyse complète. Toutefois, cette démarche pourrait être longue due soit à des délais de communication, soit à des refus ou bien à une faible quantité probable d'études comprenant des diagrammes complets et de qualité.

CHAPITRE 3
PRÉDICTION ET PRIORISATION DE L'EFFORT DE TEST BASÉES SUR LE DIAGRAMME DE CLASSES UML

3.1 État de l'art

3.1.1 Introduction

Comme mentionné précédemment, le Diagramme de Classes (DC) UML est un modèle d'un grand intérêt pour la prédiction de l'effort de test et la priorisation des tests, car c'est un diagramme d'une grande popularité au niveau des entreprises. De plus, celui-ci fournit beaucoup d'informations sur les classes d'un système et sur le système lui-même. Il est, par ailleurs, utilisé relativement tôt dans le processus de développement d'un logiciel, avant l'implémentation du code. Le DC peut être facilement extrait par des outils de rétro-ingénierie sans grande conséquence de validité due à la différence d'abstraction entre le modèle et le code. Ceci a entraîné un engouement de la part des chercheurs pour le développement de techniques de prédiction de l'effort de développement à partir de ce diagramme [61-66]. Récemment, certains chercheurs se sont aussi intéressés au développement de techniques de prédiction de l'effort de test [67] en utilisant ce diagramme.

3.1.2 Les métriques du diagramme de classes (MDC)

Les MDC sont des métriques fort intéressantes pour le développement d'une approche d'estimation et de priorisation de l'effort de test. Elles sont simples et objectives. De plus, la plupart de ces métriques sont disponibles à la phase d'analyse du processus de développement d'un logiciel [68, 69]. Ces métriques ont de ce fait été largement utilisées dans la littérature pour l'estimation de la taille du code source dans les systèmes orientés objet [61-66].

Les MDC peuvent être classifiés en deux catégories selon Zhou et al. [70] : les métriques de taille et les métriques de complexité structurelle.

Voici une brève description de chacune des métriques utilisées.

MDC de taille :

- *Nombre de classes (NC) :* Cette métrique correspond au nombre de classes d'un système orienté objet.
- *Nombre d'attributs (NA) :* Cette métrique correspond au nombre d'attributs. Elle peut être utilisée au niveau d'un système orienté objet (NA total d'un système) ou d'une classe (NA d'une seule classe).
- *Nombre de méthodes (NM) :* Cette métrique correspond au nombre de méthodes. Elle peut aussi être utilisée au niveau d'un système orienté objet (NM total d'un système) ou d'une classe (NM d'une seule classe).

MDC de complexité structurelle :

- *Nombre d'associations (Nassoc) :* Cette métrique correspond au nombre d'associations d'une classe. Plus précisément, une classe A à une association avec une classe B si elle contient un attribut de type B. Le terme association est un terme général pouvant inclure : association, ou agrégation ou bien composition.
- *Nombre d'agrégations (Nagg) :* Cette métrique correspond au nombre de relations d'agrégation de chaque classe. Plus précisément, une classe A à une relation d'agrégation avec une classe B si un attribut qu'elle contient remplit une des conditions suivantes: (1) son type est un tableau et le type des éléments de ce tableau est de la classe B (ex : B nomTableau[]), (2) son type est un type paramétré et le type de paramètre est la classe B (ex : ArrayList<B> nomListe).
- *Nombre de compositions (Ncomp) :* Cette métrique correspond au nombre de relations de composition. Plus précisément, si une classe A contient un attribut dont le type est la classe B, alors il existe une relation de composition entre A et B.
- *Nombre de dépendances (Ndep) :* Cette métrique correspond au nombre de relations de dépendance d'une classe. Plus précisément, si une classe A contient une méthode ayant un paramètre dont le type est la classe B, alors il existe une relation de dépendance entre A et B (A dépend de B).
- *Nombre de généralisations (Ngen) :* Cette métrique correspond au nombre de généralisations d'une classe. Plus précisément, si une classe A hérite d'une classe B, alors il y a une relation de généralisation entre A et B. Au niveau de la littérature,

deux métriques sont utilisées concernant les généralisations, Ngen et NgenH. La métrique NGen correspond au nombre total de relations de généralisation dans un système (somme de toutes les relations mère-fille) tandis que la métrique NGenH correspond au nombre total de généralisations hiérarchiques d'un système (somme des mères).

- *Profondeur de l'arbre d'héritage (MaxDIT et DIT) :* Cette métrique correspond au niveau de profondeur d'héritage. Elle peut s'appliquer à un système OO (niveau de profondeur maximum du système (MaxDIT)) ou à une classe seule (niveau de profondeur de la classe dans le système (DIT)).

3.1.3 Techniques de prédiction de l'effort de développement à partir du MDC

Comme mentionné précédemment, plusieurs techniques d'estimation de l'effort de développement des SOO basées sur les MDC existent à ce jour. Toutefois, la plupart de ces techniques ont été développées afin d'analyser l'effort global d'un système. Ce qui rend ces techniques moins intéressantes comme base pour le développement d'une approche de priorisation.

En 2014, Zhou et al. [70] ont mené une étude concernant la comparaison de la précision de 6 types de métriques provenant des diagrammes de classes pour l'estimation de l'effort de développement. Ils ont utilisé plusieurs techniques de modélisation afin de construire un modèle de prédiction permettant d'étudier les métriques suivantes : CDM (MDC), POP (*Predictive object points*), OOPS (*Object-oriented project size points*), FS_CP (*fast&&serious class points*), O_CP (*objective class points*), OOFP (*object-oriented function points*) [61, 71-76]. Les métriques utilisées dans leur modèle de prédiction sont les suivantes : NC, NA, NM, NAssoc, NAgg, NComp, NDep, NGen, NGenH, et MaxDIT. Il est à noter que le nombre d'associations (NAssoc) collectées correspond plutôt à la définition suivante : nombre d'associations de compositions/agrégations bidirectionnelles d'une classe. Afin de mener leur étude, ils ont collecté ces six types de métriques pour 100 systèmes Java open-source et ont ensuite utilisé leur modèle de prédiction afin d'investiguer la précision de chacune de ces types de métriques. Concernant les métriques MDC, ils ont tout d'abord constaté que toutes les métriques, exceptée MaxDIT, sont corrélées à l'effort de développement et plus particulièrement NA et NM. Finalement, leur conclusion est que les

métriques MDC ainsi que les métriques OOPS et O_CP sont de bons prédicteurs de la taille du code source des systèmes OO.

À la lumière de ces informations, les MDC semblent très intéressantes pour l'exploration du développement d'une approche d'estimation de l'effort de test unitaire et de priorisation à partir de celles-ci.

3.1.4 Techniques de prédiction de l'effort des tests à partir du DC

L'estimation de l'effort de test à partir du DC, à notre connaissance, a été peu étudiée à ce jour. Toutefois, plusieurs études ont été menées sur la testabilité ou l'effort de test à partir du code source [77-80]. Ces techniques donnent des résultats souvent très intéressants, cependant elles ne peuvent être utilisées tôt dans le processus de développement. Comme mentionné précédemment, plusieurs études ont aussi été menées à partir des DCU et des diagrammes directement dérivés de ceux-ci (diagramme de séquence et diagramme d'activités entre autres) pour l'estimation de l'effort de test [21-27,30-32]. Toutefois, l'utilisation de ces diagrammes pose un problème de validation au niveau de la recherche dû à la faible quantité d'études de cas pour lesquelles ces diagrammes sont disponibles.

Une étude récente [67] s'est toutefois intéressée à l'estimation de l'effort de test à partir du DCU et du DC. Ces auteurs proposent une technique permettant de calculer et de suivre l'effort total d'un système à partir des calculs de l'effort de test individuel basés sur le DCU et sur le DC. Le calcul de l'effort de test à partir du DCU se fait à partir de la sommation du poids de chaque CU (nombre de transactions dans le scénario normal, nombre de transactions dans les scénarios exceptionnels, nombre d'acteurs), la sommation du poids des acteurs (type de communication, nombre d'interactions avec des CU), d'un facteur de maturité technique et environnementale propre à l'organisation qui développe les tests (TEI) et de la moyenne de productivité de l'équipe de test (Prod). Le calcul de l'effort de test à partir du DC se fait quant à lui à partir du poids de chaque classe (nombre d'attributs, nombre de méthodes encapsulées, nombre de services utilisés). Cette approche d'estimation de l'effort de test est très intéressante, toutefois les auteurs ne semblent pas avoir validé leur approche.

3.1.5 Techniques de priorisation des tests à partir du DC

Comme mentionné précédemment, la majorité des études portant sur la priorisation des cas de test sont basées sur le code source. La priorisation des tests à partir des modèles apporte toutefois de grands avantages tels qu'énumérés dans le chapitre 2 de ce mémoire. En effet, le DC est un diagramme UML intéressant pour la priorisation des tests. Malheureusement, à notre connaissance, aucune technique de priorisation des tests n'a toutefois été développée à ce jour à partir de celui-ci.

3.2 Objectif

À la lumière des informations données précédemment, il est possible de comprendre que le développement d'une technique d'estimation et de priorisation de l'effort de test simple et objective pour les systèmes OO à partir du DC serait très intéressante et utile pour les développeurs d'applications.

Notre objectif dans ce contexte, est donc le développement d'une approche simple et objective de prédiction et priorisation de l'effort de test unitaire pour les systèmes OO à partir du diagramme de classes UML.

3.3 Technique de priorisation de l'effort de test à partir des MDC : étude exploratoire

Afin de remplir l'objectif de ce chapitre, une étude exploratoire a été menée portant sur le développement d'une technique de priorisation de l'effort de test à partir des MDC. Cette section de chapitre présente les différentes parties de cette étude, soit la présentation des MDC utilisées dans cette étude, la présentation des métriques de taille de test (MT) utilisées, l'étude expérimentale menée avec la description des études de cas sélectionnées, la démarche expérimentale de la méthodologie et les résultats de chaque partie de la démarche expérimentale.

3.3.1 Métriques orientés objet du DC (MDC)

Dans cette étude exploratoire, les métriques visées sont des métriques simples pouvant être extraites du DC d'une manière unitaire, c'est-à-dire pouvant être extraites de chaque classe (CL_i) du DC. Huit métriques unitaires ont donc été sélectionnées soit NA, NM, Nassoc, Ncomp, NDep, NGen, NOC et DIT. Les métriques NA, NM, Nagg, Ncomp, Ndep, NGen et DIT ont été utilisées telle qu'elles. Nassoc a toutefois été utilisée telle qu'utilisée par Zhou et al. [70], soit comme étant le nombre d'associations de compositions/agrégations bidirectionnelles d'une classe. Nous avons de plus ajouté la métrique NOC (*number of childrens*) qui correspond au nombre de classes filles qu'a une classe mère dans les cas de relation d'héritage. Cette métrique correspond à la métrique NgenH de l'étude de Zhou et al., mais sous forme unitaire.

3.3.2 Métriques de taille des tests (MT)

Afin de mesurer l'effort de test unitaire requis par une classe, trois métriques connues et utilisées en littérature [35, 81] provenant des tests unitaires JUnit ont été utilisées : TLOC, TASSERT et TINVOK.

En théorie, à chaque classe (CL_i) d'un programme devrait correspondre un cas de test JUnit (CT_i) comprenant plusieurs tests permettant de tester chaque méthode de la classe en question. Dans cette étude, la relation entre les différentes caractéristiques d'une classe d'un programme pouvant être tirées du DC et les différentes caractéristiques de son cas de test correspondant a été analysée. Pour ce faire, les métriques de chaque paire $<CL_i, CT_i>$ ont été quantifiées.

Voici la définition de chacune des MT telles qu'utilisées dans cette étude :

- *Nombre de lignes de code de tests (TLOC) :* cette métrique correspond au nombre de lignes de code de test d'un CT JUnit.
- *Nombre d'assertions (TASSERT):* cette métrique correspond au nombre de méthodes d'assertions JUnit comprises dans une CT. Les méthodes d'assertions JUnit sont utilisées par les testeurs à l'intérieur des méthodes de test afin de faire des comparaisons permettant de vérifier que le comportement d'une classe est bien celui attendu.

- *Nombre d'invocations (TINVOK)* : cette métrique correspond au nombre d'invocations de méthodes comprises dans une CT. Elle permet de capturer les dépendances des CT.

3.3.3 Étude expérimentale

3.3.3.1 Études de cas sélectionnées

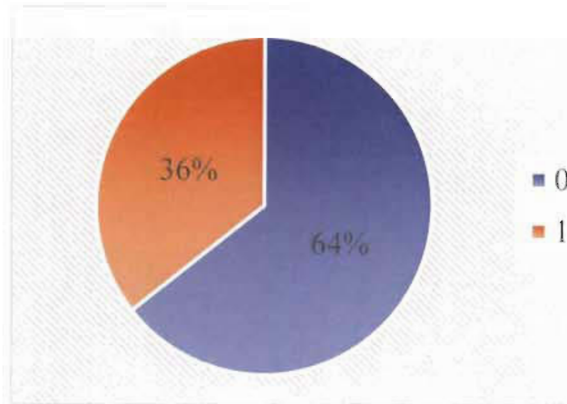
Afin d'évaluer la relation entre les MT et les MDC, 6 études de cas Java provenant de différents domaines ont été utilisées : Nextgen, Exec, Email, IO, Jaqlib et iValidator. Chacune de ces études de cas comprend le code source et les CT associées. Toutefois, aucune de ces études de cas ne comprenait de DC disponible publiquement.

Le tableau 3.1 comprend les statistiques générales relatives à ces études de cas. Il comprend le nombre de classes, le nombre de CT, le nombre de méthodes et le nombre de lignes de code. Au total, 430 classes et 152 CT ont été prises en compte pour cette expérience. Il est à noter que certaines classes ont été écartées si leur CT était hors normes (non fait de manière unitaire, pas d'utilisation de méthodes d'assertions, etc.). Par ailleurs, en observant la figure 3.1, on constate qu'environ 36% des classes conservées ont été testées par les testeurs, toutes études de cas confondues.

Tableau 3.1
Statistiques concernant les 6 études de cas

	NC	NCT	NM	TLOC
Nextgen	10	8	47	408
Email	14	5	95	1577
Exec	25	18	128	1617
IO	95	62	684	14944
iValidator	117	38	563	4440
Jaqlib	169	21	671	1250

Figure 3.1
Taux de classes testées (1) vs de classes non testées (0)



3.3.3.2 Démarche expérimentale

Les différentes étapes (5) de notre démarche expérimentale sont :

1. Rétro-ingénierie du code
2. Collecte des MDC
3. Collecte des MT
4. Analyses statistiques des données
5. Utilisation d'algorithmes d'apprentissage automatique pour la classification

3.3.3.2.1 Rétro-ingénierie du code

Afin de pallier à la non-disponibilité des DC des études de cas utilisées dans notre approche, la première étape de notre démarche scientifique consistait en la rétro-ingénierie du code source des études de cas en DC. Ainsi, pour chaque étude de cas (EC_i), un diagramme de classes (DC_i) a été produit. Pour ce faire, l'outil Understand a été utilisé. Ce puissant outil de rétro-ingénierie a été utilisé dans plusieurs études scientifiques [70, 82-86].

3.3.3.2.2 Collecte des données

À la suite de la rétro-ingénierie, les MDC et MT ont été collectées. Les 7 MDC ont été collectées à partir des DC produits par l'outil Understand. Les métriques de test ont été collectées à l'aide de deux techniques différentes. La métrique TLOC a été collectée directement à partir de l'outil Understand. Les deux autres métriques de test, soient TASSERT et TINVOK, ont été calculées à partir du code source des CT grâce à des requêtes de recherche dans l'outil Understand.

3.3.3.2.3 Analyses statistiques des données

3.3.3.2.3.1 Analyses de corrélations

Afin d'évaluer l'existence d'une relation entre les MDC et les MT, une analyse de corrélations a été effectuée à l'aide du logiciel XLSTAT [36]. L'objectif de cette analyse est la vérification des hypothèses suivantes pour chaque couple de métriques $<MDC_i, MT_i>$:

- Hypothèse : Il existe une relation de corrélation entre la métrique MDC_i et la métrique MT_i.
- Hypothèse nulle : Il n'existe pas de relation de corrélation entre la métrique MDC_i et la métrique MT_i.

Afin de vérifier ces hypothèses, deux techniques d'analyse de corrélation ont été utilisées : la corrélation de Spearman et la corrélation de Kendall. Ces deux techniques ont été choisies, car elles ne tiennent pas compte de la distribution des données contrairement à la technique de corrélation de Pearson qui nécessite deux variables linéaires distribuées de façon normale.

Pour les deux techniques de corrélation, le seuil de signification alpha choisi est le seuil standard 0.05.

Ces deux techniques d'analyse de corrélation ont tout d'abord été utilisées pour analyser les données des 152 classes des études de cas retenues et dont les tests ont été développés. Les tableaux 3.2 et 3.3 présentent respectivement les corrélations de Spearman et de Kendall entre les MDC et les MT pour cet ensemble de données. Parmi les constats intéressants pouvant être faits, on remarque que : les métriques NA et NM sont corrélées positivement à TLOC, TASSERT et TINVOK, que la métrique NDep est corrélée positivement à TLOC et TINVOK, que la métrique

NGen est négativement corrélée à TLOC, TASSERT et TINVOK et enfin que les trois MT sont fortement corrélées positivement ensembles. Il est aussi possible de constater que NM est la MDC la plus fortement corrélée aux trois métriques de test.

Tableau 3.2
Corrélations de Spearman entre les MDC et MT des classes testées

Var.	**NA**	**NM**	**NASSOC**	**NAGG**	**NCOMP**	**NDEP**	**NGEN**	**NOC**	**DIT**	**TLOC**	**TASSERT**	**TINVOK**
NA	**1**	**0,329**	-0,099	**0,199**	**0,318**	**0,282**	-0,137	0,005	**-0,173**	**0,391**	**0,294**	**0,336**
NM	**0,329**	**1**	-0,005	0,019	0,095	**0,234**	**-0,210**	0,092	**-0,186**	**0,696**	**0,554**	**0,667**
NASSOC	-0,099	-0,005	**1**	-0,125	**0,414**	**-0,187**	0,016	-0,053	0,052	**-0,165**	-0,079	-0,084
NAGG	**0,199**	0,019	-0,125	**1**	0,062	**0,245**	-0,029	-0,083	-0,055	0,122	-0,025	0,063
NCOMP	**0,318**	0,095	**0,414**	0,062	**1**	**0,365**	**-0,245**	-0,040	**-0,249**	0,014	-0,045	0,065
NDEP	**0,282**	**0,234**	**-0,187**	**0,245**	**0,365**	**1**	**-0,235**	-0,022	**-0,198**	**0,243**	0,146	**0,313**
NGEN	-0,137	**-0,210**	0,016	-0,029	**-0,245**	**-0,235**	**1**	0,027	**0,920**	**-0,174**	**-0,176**	**-0,303**
NOC	0,005	0,092	-0,053	-0,083	-0,040	-0,022	0,027	**1**	0,092	0,079	-0,035	-0,041
DIT	**-0,173**	**-0,186**	0,052	-0,055	**-0,249**	**-0,198**	**0,920**	0,092	**1**	-0,145	-0,124	**-0,288**
TLOC	**0,391**	**0,696**	**-0,165**	0,122	0,014	**0,243**	**-0,174**	0,079	-0,145	**1**	**0,767**	**0,741**
TASSERT	**0,294**	**0,554**	-0,079	-0,025	-0,045	0,146	**-0,176**	-0,035	-0,124	**0,767**	**1**	**0,646**
TINVOK	**0,336**	**0,667**	-0,084	0,063	0,065	**0,313**	**-0,303**	-0,041	**-0,288**	**0,741**	**0,646**	**1**

Tableau 3.3
Corrélations de Kendall entre les MDC et MT des classes testées

Var.	**NA**	**NM**	**NASSOC**	**NAGG**	**NCOMP**	**NDEP**	**NGEN**	**NOC**	**DIT**	**TLOC**	**TASSERT**	**TINVOK**
NA	**1**	**0,257**	-0,085	**0,171**	**0,272**	**0,219**	**-0,120**	0,004	**-0,146**	**0,292**	**0,226**	**0,257**
NM	**0,257**	**1**	-0,004	0,017	0,074	**0,174**	**-0,177**	0,075	**-0,148**	**0,523**	**0,407**	**0,505**
NASSOC	-0,085	-0,004	**1**	**-0,122**	**0,394**	**-0,162**	0,016	-0,051	0,050	**-0,135**	-0,066	-0,069
NAGG	**0,171**	0,017	**-0,122**	**1**	0,059	**0,213**	-0,028	-0,080	-0,053	0,100	-0,021	0,052
NCOMP	**0,272**	0,074	**0,394**	0,059	**1**	**0,309**	**-0,232**	-0,037	**-0,230**	0,014	-0,037	0,052
NDEP	**0,219**	**0,174**	**-0,162**	**0,213**	**0,309**	**1**	**-0,205**	-0,018	**-0,164**	**0,179**	**0,109**	**0,231**
NGEN	**-0,120**	**-0,177**	0,016	-0,028	**-0,232**	**-0,205**	**1**	0,027	**0,889**	**-0,143**	**-0,147**	**-0,251**
NOC	0,004	0,075	-0,051	-0,080	-0,037	-0,018	0,027	**1**	0,084	0,062	-0,030	-0,031
DIT	**-0,146**	**-0,148**	0,050	-0,053	**-0,230**	**-0,164**	**0,889**	0,084	**1**	**-0,115**	-0,101	**-0,231**
TLOC	**0,292**	**0,523**	**-0,135**	0,100	0,014	**0,179**	**-0,143**	0,062	**-0,115**	**1**	**0,598**	**0,580**
TASSERT	**0,226**	**0,407**	-0,066	-0,021	-0,037	**0,109**	**-0,147**	-0,030	-0,101	**0,598**	**1**	**0,502**
TINVOK	**0,257**	**0,505**	-0,069	0,052	0,052	**0,231**	**-0,251**	-0,031	**-0,231**	**0,580**	**0,502**	**1**

Les deux mêmes techniques de corrélations ont aussi été utilisées afin de vérifier les mêmes hypothèses sur l'ensemble de toutes les classes testées et non testées (sur les 430 classes totales). Les tableaux 3.4 et 3.5 présentent respectivement ces résultats. En analysant les résultats de ces deux nouveaux tableaux, il est possible de faire les mêmes constats que pour les classes testées concernant les corrélations entre les MDC NA, NM et NGEN et les MT. Cependant, ces

corrélations sont toutes plus faibles que les premières corrélations obtenues. De plus, la métrique NDEP n'est plus corrélée significativement à TLOC et TINVOK. Ces deux observations s'expliquent propablement par le fait que plusieurs classes parmi l'ensemble des classes analysées comprennant plusieurs attributs, méthodes et dépendances n'ont pas été testées par les testeurs.

Tableau 3.4
Corrélations de Spearman entre les MDC et MT des classes testées et non testées

Var.	**NA**	**NM**	**NASSOC**	**NAGG**	**NCOMP**	**NDEP**	**NGEN**	**NOC**	**DIT**	**TLOC**	**TASSERT**	**TINVOK**
NA	**1**	**0,324**	-0,055	**0,225**	**0,386**	**0,316**	**-0,265**	**0,128**	**-0,246**	**0,317**	**0,319**	**0,317**
NM	**0,324**	**1**	-0,021	0,006	0,087	**0,250**	**-0,251**	**0,116**	**-0,250**	**0,469**	**0,440**	**0,472**
NASSOC	-0,055	-0,021	**1**	0,025	**0,380**	**-0,109**	0,025	-0,068	0,008	0,049	0,020	0,055
NAGG	**0,225**	0,006	0,025	**1**	**0,120**	**0,207**	-0,082	-0,063	**-0,108**	0,055	0,010	0,048
NCOMP	**0,386**	0,087	**0,380**	**0,120**	**1**	**0,408**	**-0,134**	0,094	**-0,129**	0,028	0,010	0,038
NDEP	**0,316**	**0,250**	**-0,109**	**0,207**	**0,408**	**1**	**-0,101**	0,057	-0,033	-0,009	-0,007	0,011
NGEN	**-0,265**	**-0,251**	0,025	-0,082	**-0,134**	**-0,101**	**1**	-0,075	**0,934**	**-0,144**	**-0,163**	**-0,173**
NOC	**0,128**	**0,116**	-0,068	-0,063	0,094	0,057	-0,075	**1**	-0,065	0,020	0,002	0,006
DIT	**-0,246**	**-0,250**	0,008	**-0,108**	**-0,129**	-0,033	**0,934**	-0,065	**1**	**-0,124**	**-0,133**	**-0,154**
TLOC	**0,317**	**0,469**	0,049	0,055	0,028	-0,009	**-0,144**	0,020	**-0,124**	**1**	**0,947**	**0,975**
TASSERT	**0,319**	**0,440**	0,020	0,010	0,010	-0,007	**-0,163**	0,002	**-0,133**	**0,947**	**1**	**0,928**
TINVOK	**0,317**	**0,472**	0,055	0,048	0,038	0,011	**-0,173**	0,006	**-0,154**	**0,975**	**0,928**	**1**

Tableau 3.5
Corrélations de Kendall entre les MDC et MT des classes testées et non testées

Var.	**NA**	**NM**	**NASSOC**	**NAGG**	**NCOMP**	**NDEP**	**NGEN**	**NOC**	**DIT**	**TLOC**	**TASSERT**	**TINVOK**
NA	**1**	**0,256**	-0,048	**0,199**	**0,338**	**0,250**	**-0,236**	**0,111**	**-0,205**	**0,264**	**0,268**	**0,265**
NM	**0,256**	**1**	-0,018	0,005	**0,070**	**0,192**	**-0,216**	**0,095**	**-0,206**	**0,385**	**0,358**	**0,386**
NASSOC	-0,048	-0,018	**1**	0,025	**0,361**	**-0,094**	0,025	**-0,065**	0,007	0,044	0,019	0,050
NAGG	**0,199**	0,005	0,025	**1**	**0,112**	**0,179**	**-0,081**	-0,060	**-0,102**	0,050	0,009	0,044
NCOMP	**0,338**	**0,070**	**0,361**	**0,112**	**1**	**0,342**	**-0,127**	**0,086**	**-0,116**	0,025	0,008	0,033
NDEP	**0,250**	**0,192**	**-0,094**	**0,179**	**0,342**	**1**	**-0,088**	0,048	-0,025	-0,007	-0,006	0,009
NGEN	**-0,236**	**-0,216**	0,025	**-0,081**	**-0,127**	**-0,088**	**1**	**-0,073**	**0,891**	**-0,132**	**-0,151**	**-0,158**
NOC	**0,111**	**0,095**	**-0,065**	-0,060	**0,086**	0,048	**-0,073**	**1**	-0,061	0,018	0,002	0,006
DIT	**-0,205**	**-0,206**	0,007	**-0,102**	**-0,116**	-0,025	**0,891**	-0,061	**1**	**-0,109**	**-0,119**	**-0,136**
TLOC	**0,264**	**0,385**	0,044	0,050	0,025	-0,007	**-0,132**	0,018	**-0,109**	**1**	**0,880**	**0,901**
TASSERT	**0,268**	**0,358**	0,019	0,009	0,008	-0,006	**-0,151**	0,002	**-0,119**	**0,880**	**1**	**0,850**
TINVOK	**0,265**	**0,386**	0,050	0,044	0,033	0,009	**-0,158**	0,006	**-0,136**	**0,901**	**0,850**	**1**

3.3.3.2.3.2 Analyses en composantes principales

Les résultats de l'analyse de corrélation ci-haut suggèrent que certaines des MDC et MT sont liées entre elles. Ce phénomène apporte de la redondance d'information et pourrait surestimer les résultats d'un algorithme de prédiction de l'effort de test. Afin de limiter certaines de ces redondances, deux ACP ont été effectuées dans cette étude : une ACP portant sur les MDC et une autre portant sur les MT.

3.3.3.2.3.2.1 ACP portant sur les MDC

Une première ACP a donc été menée à l'aide du logiciel XLSTAT sur les MDC retenues soit : NA, NM et NDEP. Le tableau 3.6 présentent ces résultats. En les analysant, on constate que l'ACP n'a pas réussi à diminuer le nombre de facteurs par rapport au nombre de métriques original. Effectivement, les deux premiers facteurs trouvés ne capturent qu'environ 78% de la variance des données. Les 3 MDC considérées ont donc été conservées pour la suite des expérimentations.

Tableau 3.6
Résultats de l'ACP portant sur les MDC

	F1	F2	F3
Valeur propre	1,565	0,772	0,662
% cumulé	52,182	77,921	100,000
NA	**36,983**	3,313	**59,704**
NM	33,514	31,845	34,641
NDEP	29,503	**64,842**	5,655

3.3.3.2.3.2.2 ACP portant sur les MT

Une seconde ACP a été menée sur les MT soient les métriques TLOC, TASSERT et TINVOK afin de réduire la redondance éventuellement présente dans ce jeu de données. Le tableau 3.7 présente ces résultats. En les analysant, nous pouvons constater que les deux premiers facteurs permettent de capturer plus de 90% de la variance du jeu de données. En analysant la contribution des métriques, nous pouvons remarquer que les métriques TLOC, TASSERT et TINVOK ont une

bonne contribution pour le premier facteur. TLOC contribue un peu plus fortement. Ce facteur représente probablement la taille. En ce qui concerne le facteur 2, la métrique TINVOK à une plus grande contribution, suivi de la métrique TASSERT. Ce facteur représente probablement la notion de dépendance.

Tableau 3.7
Résultats de l'ACP portant sur les MT

	F1	F2	F3
Valeur propre	2,437	0,356	0,207
% cumulé	81,238	93,096	100,000
TLOC	**35,514**	0,607	**63,879**
TASSERT	**32,670**	**43,289**	24,041
TINVOK	**31,816**	**56,104**	12,080

3.3.3.2.3.3 Analyses de régression linéaire (RL)

Afin de vérifier l'effet de chacune des métriques MDC sur les métriques MT, deux techniques de régression ont été utilisées dans cette étude : la régression linéaire simple et la régression linéaire multiple. Ces techniques de régression ont aussi été effectuées à l'aide du logiciel XLSTAT.

La RL est un algorithme d'apprentissage supervisé permettant de modéliser une variable dépendante quantitative Y à partir d'une combinaison linéaire d'une ou de plusieurs autres variables quantitatives explicatives X_i.

3.3.3.2.3.3.1 Analyses de RL simples

La RL simple est tout simplement une RL modélisée à partir d'une seule variable quantitative explicative X. Ce modèle est représenté par l'équation suivante dans le contexte de cette étude: $MT_i = aMDC_i + e$, e étant l'erreur du modèle.

Grâce à la régression simple, chaque couple de métriques $<MDC_i, MT_i>$ a donc été analysé pour toutes les MDC et MT d'intérêts (NA, NM, NDEP,TLOC et TINVOK).

Les hypothèses à vérifier par les régressions simples menées dans cette étude sont les suivantes :

- Hypothèse : La variable MDC_i permet de prédire la variable MT_i par une relation linéaire.
- Hypothèse nulle : La variable MDC_i ne permet pas de prédire la variable MT_i par une relation linéaire.

Le tableau 3.8 présente les résultats des analyses de RL simples. En observant les résultats des tests de Fisher (Pr > f) du tableau 3.8, on constate que pour tous les modèles de RL simples, toutes les variables explicatives apportent une quantité d'informations significative. Ces résultats sont tous inférieurs ou égaux à 0,005. En observant les modèles du tableau 3.8, on constate que TINVOK augmente respectivement de 16.764, 19.068, 10.155 tandis que TLOC augmente respectivement de 7.694, 8.879 et 6.938, lorsque NA, NM et NDEP augmentent respectivement de 1. On constate également, que NA, NM et NDEP sont responsables respectivement de 6.5%, 72% et 6.8% de la variabilité de TLOC et de 5.2%, 58.7% et 12% de la variabilité de TINVOK.

Tableau 3.8
Résultats des analyses de RL simples entre les MDC et MT d'intérêts

		a	**e**	**R^2**	**Pr > f**
TLOC	**NA**	16.764	81.724	6.5%	0.001
	NM	19.068	-23.158	72%	<0.0001
	NDEP	10.155	94.938	6.8%	0.001
TINVOK	**NA**	7.694	25.211	5.2%	0.005
	NM	8.879	-23.948	58.7%	<0.0001
	NDEP	6.938	23.549	12%	<0.0001

3.3.3.2.3.3.2 Analyses de RL multiples

La RL multiple est une RL modélisée à partir de plusieurs variables quantitatives explicatives X_i. Dans cette étude, les variables explicatives utilisées dans la RL multiples sont les MDC - NA, NM et NDEP et les variables expliquées sont les MT - TLOC et TINVOK. Les modèles

testés dans cette étude peuvent être représentés par l'équation suivante : MT_i = aNA+ bNM + cNDEP + e.

Les hypothèses à vérifier par les régressions multiples menées dans cette étude sont les suivantes :

- Hypothèse : Les variables NA, NM et NDEP permettent de prédire la variable MT_i par une relation linéaire.
- Hypothèse nulle : Les variables NA, NM et NDEP ne permettent pas de prédire la variable MT_i par une relation linéaire.

Le tableau 3.9 présente les résultats respectifs des analyses de RL multiples pour les métriques TLOC et TINVOK. En l'observant et grâce aux résultats des tests de Fisher (Pr > f), on constate que les variables explicatives apportent une quantité d'informations significative concernant les deux MT. Effectivement, les deux valeurs de ce test sont < 0.0001. On constate également que le modèle proposé pour la variable TLOC est responsable de 72.3% de la variabilité totale de TLOC, tandis que celui proposé pour la variable TINVOK est responsable de 61.3% de la variabilité totale de TINVOK. Les modèles du tableau 3.9 montrent que lorsque la métrique NA augmente de 1, les poids de TLOC et TINVOK diminuent respectivement de 1.944 et 1.910. Par ailleurs, lorsque les métriques NM et NDEP augmentent de 1, les poids de TLOC et TINVOK augmentent respectivement de 19.002 et 1.843 pour TLOC et de 8.581 et 3.351 pour TINVOK. Il est aussi possible de constater que la métrique NM (b) est la variable qui contribue le plus significativement aux deux modèles.

Tableau 3.9
Analyses de RL multiples concernant les MDC et MT d'intérêts

	Var.	Val.	σ	t	Pr > |t|	R^2	Pr > f
TLOC	a	-1.944	3,057	-0,636	0,526	72.3%	< 0.0001
	b	19.002	1,043	18,224	< 0,0001		
	c	1.843	1,775	1,038	0,301		
	e	-23.357	13,757	-1,698	0,092		
TINVOK	a	-1.910	1,863	-1,025	0,307	61.3%	< 0.0001
	b	8.581	0,635	13,503	< 0,0001		
	c	3.351	1,082	3,097	0,002		
	e	-27.498	8,384	-3,280	0,001		

3.3.3.2.4 Algorithmes d'apprentissage d'ordonnancement

Des algorithmes d'apprentissage d'ordonnancement ont été utilisés dans cette étude pour la construction de modèles de priorisation de l'effort de test. Afin de pallier le faible nombre de classes testées provenant des études de cas choisies et d'augmenter la taille de l'ensemble de données utilisées pour construire notre modèle de priorisation, l'intégration des classes non testées à un modèle de priorisation a aussi été évaluée. Dans cette étude expérimentale, plusieurs algorithmes d'apprentissage d'ordonnancement ont donc été utilisés dans un but ultime de vérifier et valider la justesse et la validité d'un modèle de priorisation des tests construit à partir de classes testées et non testées des différentes études de cas sélectionnées.

3.3.3.2.4.1 Hypothèses

Plus précisément, deux hypothèses ont été vérifiées dans cette section étude. La première concerne la significativité des modèles de priorisation de l'effort de test conçus à partir d'algorithmes d'apprentissage. Les hypothèses concernant chaque modèle de priorisation sont les suivantes :

- Hypothèse : Le modèle de priorisation de l'effort de test mod_i conçu à partir de l'algorithme alg_i d'apprentissage d'ordonnancement et de la méthode de validation val_i permet de classifier l'effort de test relatif aux classes d'un système OO de manière significative.
- Hypothèse nulle : Le modèle de priorisation de l'effort de test mod_i conçu à partir de l'algorithme alg_i d'apprentissage et de la méthode de validation val_i ne permet pas de classifier l'effort de test relatif aux classes d'un système OO de manière significative.

La seconde hypothèse vérifiée concerne l'ordonnancement des classes non testées. Voici les hypothèses à propos de celui-ci :

- Hypothèse : Les classes non testées sont classées majoritairement dans les plus hauts rangs
- Hypothèse nulle : Les classes non testées ne sont pas classées majoritairement dans les plus hauts rangs

3.3.3.2.4.2 Démarche expérimentale

Dans cette démarche expérimentale, un programme informatique créé par un étudiant du GLOG a été utilisé afin d'étudier la possibilité d'ordonnancer les classes d'un système OO à partir d'un algorithme d'apprentissage automatique prédisant l'effort de test. Dans cette étude, deux types d'algorithmes d'apprentissage automatique ont été utilisés pour la priorisation : des algorithmes de régression et des algorithmes d'apprentissage d'ordonnancement (LTR). Plus précisément, cinq algorithmes de régression, soient la régression linéaire par les moindres carrées (*Ordinary Least Squares*), la régression Ridge (*ridge regression*), la régression linéaire par les moindres carrés récursifs (*Recursive Least Squares*), la régression Lasso [57] et la régression avec *gradient boosting* [58], et trois algorithmes de LTR soient MART [40], Ranknet [47], et Coordinate Ascent [54]. Les librairies Smile [59] et Ranklib [60] ont respectivement été utilisées pour l'implémentation des algorithmes de régression et de LTR.

Tous les modèles de priorisation ont été bâtis à l'aide des mêmes MDC retenues, c'est-à-dire les métriques NA, NM et NDEP afin de tenter de prédire individuellement les métriques de test TLOC et TINVOK.

De plus, deux méthodes de validation différentes ont été utilisées et évaluées afin d'estimer la qualité des modèles : la k-fold cross-validation et la validation inter-projet. La k-fold cross-validation consiste à former k sous-ensembles et à utiliser un de ces sous-ensembles afin de valider le modèle. Les autres sous-ensembles sont utilisés pour entraîner le modèle. Dans cette étude, 10 folds ont été choisis pour ce type de validation. La validation inter-projet consiste, quant à elle, à entraîner un modèle à l'aide des données d'un projet choisi et de l'évaluer sur un second projet.

La validation des modèles a été évaluée pour les deux méthodes de validation en calculant, dans un premier temps, les scores de chaque algorithme pour chacune des classes du projet testé. Ensuite, les corrélations de Spearman et Kendall ont été calculées entre ces scores et la métrique réelle de l'effort de test (TLOC ou TINVOK).

3.3.3.3 Résultats

Comme les études de cas de cette étude sont de taille variable, seulement certaines de ces études comprenant un nombre relativement important de classes ont été sélectionnées afin d'entraîner les modèles avec la technique de validation 10-folds : IO, Jaqlib et iValidator. À la suite de l'analyse des résultats de cette technique, seules les études IO et iValidator ont été retenues pour la validation inter-projet. Certaines études ont aussi été combinées afin d'entraîner les modèles pour les deux techniques de validation. Un premier ensemble combinant les études provenant de la librairie Apache, et un second ensemble combinant toutes les classes de toutes les études de cas ont été formés (tous les projets).

Le tableau 3.10 présente les résultats des corrélations entre le score calculé par chaque algorithme d'apprentissage et une métrique de test (TLOC ou TINVOK) avec la technique de validation 10-folds. Le tableau 3.11, quant à lui, présente les résultats des corrélations entre les mêmes métriques avec la technique de validation inter-projet.

Tableau 3.9
Corrélations de Spearman et Kendall entre les scores et les métriques d'effort de test avec validation 10-folds

		Spearman TLOC	Kendall TLOC	Spearman TINVOK	Kendall TINVOK
IO	MART	0,52411	0,43646	0,46880	0,39063
	RankNet	0,46583	0,36335	0,26407	0,23834
	Coordinate Ascent	0,46537	0,35242	0,40633	0,33033
	OrdinaryLeastSquare	0,56795	0,47021	0,43927	0,32966
	GradientTreeBoost	0,55345	0,41753	0,54481	0,42641
	RidgeRegression	0,57577	0,47826	0,34977	0,27983
	LASSO	0,57799	0,48164	0,47999	0,38524
	RecursiveLeastSquare	0,59923	0,47913	0,37149	0,30269
iValidator	MART	0,51160	0,46334	0,50632	0,44258
	RankNet	0,23984	0,20368	0,43908	0,36208
	Coordinate Ascent	-0,15451	-0,13062	0,24085	0,20214
	OrdinaryLeastSquare	0,46320	0,37927	0,47421	0,40146
	GradientTreeBoost	0,36309	0,30637	0,49582	0,42531
	RidgeRegression	0,45506	0,38296	0,45768	0,38912
	LASSO	0,48188	0,39617	0,46112	0,39079
	RecursiveLeastSquare	0,50048	0,42281	0,50823	0,42828
Jaqlib	MART	NaN	NaN	NaN	NaN
	RankNet	NaN	NaN	0,16602	0,14290
	Coordinate Ascent	NaN	NaN	NaN	NaN
	OrdinaryLeastSquare	0,30254	0,25809	NaN	NaN
	GradientTreeBoost	NaN	NaN	0,16754	0,15876
	RidgeRegression	NaN	NaN	NaN	NaN
	LASSO	0,27486	0,23147	NaN	NaN
	RecursiveLeastSquare	NaN	NaN	0,23477	0,20029
Apache	MART	0,45819	0,39655	0,49519	0,41090
	RankNet	0,53993	0,40856	0,07217	0,05642
	Coordinate Ascent	-0,00792	-0,00535	0,57951	0,46832
	OrdinaryLeastSquare	0,61400	0,48219	0,53255	0,40537
	GradientTreeBoost	0,57172	0,44311	0,57042	0,46370
	RidgeRegression	0,57528	0,45671	0,54645	0,42204
	LASSO	0,57881	0,48064	0,52087	0,40289
	RecursiveLeastSquare	0,56348	0,45437	0,49866	0,38524
Tous les projets	MART	0,40237	0,34995	0,44417	0,37607
	RankNet	0,45848	0,36656	0,44045	0,35234
	Coordinate Ascent	0,02552	0,02092	0,41884	0,33760
	OrdinaryLeastSquare	0,46200	0,37331	0,44624	0,35402
	GradientTreeBoost	0,47694	0,41023	0,44931	0,38327
	RidgeRegression	0,42885	0,34105	0,45657	0,36333
	LASSO	0,45040	0,36333	0,46197	0,36880
	RecursiveLeastSquare	0,46368	0,37243	0,44468	0,35730

Tableau 3.10
Corrélations de Spearman et Kendall entre les scores et les métriques d'effort de test avec validation inter-projet

		Spearman TLOC	Kendall TLOC	Spearman TINVOK	Kendall TINVOK
IO-Exec	MART	0,52675	0,42267	0,47110	0,39247
	RankNet	0,56862	0,43845	0,63698	0,48194
	Coordinate Ascent	0,44797	0,34798	0,60529	0,51161
	OrdinaryLeastSquare	0,54501	0,43114	0,41673	0,31920
	GradientTreeBoost	0,54501	0,43114	0,41673	0,31920
	RidgeRegression	0,54501	0,43114	0,41673	0,31920
	LASSO	0,54501	0,43114	0,41673	0,31920
	RecursiveLeastSquare	0,54501	0,43114	0,41673	0,31920
IO-Nextgen	MART	0,80029	0,67700	0,91065	0,82486
	RankNet	0,54268	0,38636	0,58716	0,36784
	Coordinate Ascent	0,16977	0,18835	0,88685	0,73568
	OrdinaryLeastSquare	0,54268	0,38636	0,36697	0,18392
	GradientTreeBoost	0,54268	0,38636	0,36697	0,18392
	RidgeRegression	0,54268	0,38636	0,36697	0,18392
	LASSO	0,54268	0,38636	0,36697	0,18392
	RecursiveLeastSquare	0,54268	0,38636	0,36697	0,18392
IO-Email	MART	0,63473	0,57417	0,63473	0,57417
	RankNet	0,51597	0,40020	0,42356	0,31445
	Coordinate Ascent	0,32123	0,25873	0,60865	0,49727
	OrdinaryLeastSquare	0,46618	0,36141	0,46618	0,36141
	GradientTreeBoost	0,46618	0,36141	0,46618	0,36141
	RidgeRegression	0,46618	0,36141	0,46618	0,36141
	LASSO	0,46618	0,36141	0,46618	0,36141
	RecursiveLeastSquare	0,46618	0,36141	0,46618	0,36141
IO-iValidator	MART	0,48870	0,37195	0,50793	0,40961
	RankNet	-0,57114	-0,42470	-0,00596	-0,00073
	Coordinate Ascent	0,05849	0,04204	0,57705	0,45483
	OrdinaryLeastSquare	0,61748	0,47835	0,58600	0,45483
	GradientTreeBoost	0,61748	0,47835	0,58600	0,45483
	RidgeRegression	0,61748	0,47835	0,58600	0,45483
	LASSO	0,61748	0,47835	0,58600	0,45483
	RecursiveLeastSquare	0,61748	0,47835	0,58600	0,45483
iValidator-Exec	MART	0,45071	0,37016	0,51964	0,42831
	RankNet	0,38570	0,29926	0,59921	0,49591
	Coordinate Ascent	-0,68382	-0,54981	0,55555	0,44488
	OrdinaryLeastSquare	0,64573	0,53009	0,61904	0,49591
	GradientTreeBoost	0,64573	0,53009	0,61904	0,46107

	RidgeRegression	0,64573	0,53009	0,61904	0,46107
	LASSO	0,64573	0,53009	0,61904	0,46107
	RecursiveLeastSquare	0,64573	0,53009	0,61904	0,46107
iValidator-Nextgen	**MART**	0,77037	0,61546	0,80274	0,68483
	RankNet	0,39634	0,20455	0,63609	0,45980
	Coordinate Ascent	-0,66463	-0,56818	0,80790	0,69268
	OrdinaryLeastSquare	0,73780	0,56818	0,78288	0,64372
	GradientTreeBoost	0,73780	0,56818	0,78288	0,64372
	RidgeRegression	0,73780	0,56818	0,78288	0,64372
	LASSO	0,73780	0,56818	0,78288	0,64372
	RecursiveLeastSquare	0,73780	0,56818	0,78288	0,64372
iValidator-Email	**MART**	-0,05075	-0,03636	0,50448	0,40000
	RankNet	0,48003	0,37162	0,44923	0,34303
	Coordinate Ascent	-0,71106	-0,60031	0,48487	0,38252
	OrdinaryLeastSquare	0,43527	0,33250	0,43527	0,33250
	GradientTreeBoost	0,43527	0,33250	0,43527	0,33250
	RidgeRegression	0,43527	0,33250	0,43527	0,33250
	LASSO	0,43527	0,33250	0,43527	0,33250
	RecursiveLeastSquare	0,43527	0,33250	0,43527	0,33250
iValidator-IO	**MART**	0,48870	0,37195	0,50793	0,40961
	RankNet	0,62499	0,48155	0,56352	0,41687
	Coordinate Ascent	0,05849	0,04204	0,57705	0,44613
	OrdinaryLeastSquare	0,61748	0,47835	0,58600	0,45483
	GradientTreeBoost	0,61748	0,47835	0,58600	0,45483
	RidgeRegression	0,61748	0,47835	0,58600	0,45483
	LASSO	0,61748	0,47835	0,58600	0,45483
	RecursiveLeastSquare	0,61748	0,47835	0,58600	0,45483
Apache-Email	**MART**	0,44468	0,34751	0,43233	0,34653
	RankNet	0,56781	0,42393	0,18751	0,13687
	Coordinate Ascent	0,11866	0,07853	0,59603	0,46402
	OrdinaryLeastSquare	0,62450	0,48222	0,60760	0,47465
	GradientTreeBoost	0,62450	0,48222	0,60760	0,47465
	RidgeRegression	0,62478	0,48222	0,60760	0,47465
	LASSO	0,62450	0,48222	0,60760	0,47465
	RecursiveLeastSquare	0,62450	0,48222	0,60760	0,47465
Apache-Exec	**MART**	0,32519	0,25664	0,27940	0,21982
	RankNet	-0,47960	-0,34507	0,14763	0,10251
	Coordinate Ascent	-0,09074	-0,05508	0,60569	0,45803
	OrdinaryLeastSquare	0,61326	0,47874	0,60177	0,47412
	GradientTreeBoost	0,61326	0,47874	0,60177	0,47412

	RidgeRegression	0,61326	0,47874	0,60177	0,47412
	LASSO	0,61326	0,47874	0,60177	0,47412
	RecursiveLeastSquare	0,61326	0,47874	0,60177	0,47412
Apache-Nextgen	**MART**	0,41892	0,32163	0,33331	0,25207
	RankNet	0,30353	0,22082	0,06687	0,04472
	Coordinate Ascent	-0,56243	-0,41314	0,59368	0,46065
	OrdinaryLeastSquare	0,61327	0,47611	0,59368	0,46065
	GradientTreeBoost	0,61327	0,47611	0,59575	0,46610
	RidgeRegression	0,61327	0,47611	0,59368	0,46065
	LASSO	0,61327	0,47611	0,59368	0,46065
	RecursiveLeastSquare	0,61327	0,47611	0,44062	0,33014
Tous les projets -Exec	**MART**	0,21391	0,18295	0,19877	0,16433
	RankNet	-0,45616	-0,35350	0,24745	0,19486
	Coordinate Ascent	-0,01844	-0,01204	0,45162	0,35855
	OrdinaryLeastSquare	0,45517	0,36403	0,47094	0,37766
	GradientTreeBoost	0,45517	0,36403	0,47094	0,37766
	RidgeRegression	0,45517	0,36403	0,47094	0,37766
	LASSO	0,45517	0,36403	0,47094	0,37766
	RecursiveLeastSquare	0,45517	0,36403	0,47094	0,37766
Tous les projets -Nextgen	**MART**	0,35236	0,28853	0,29337	0,24028
	RankNet	-0,44099	-0,34741	0,31552	0,24333
	Coordinate Ascent	-0,10891	-0,08283	-0,09489	-0,07424
	OrdinaryLeastSquare	0,45547	0,36255	0,48506	0,38711
	GradientTreeBoost	0,45547	0,36255	0,48506	0,38711
	RidgeRegression	0,45547	0,36255	0,48241	0,38538
	LASSO	0,45547	0,36255	0,48506	0,38711
	RecursiveLeastSquare	0,45547	0,36255	0,48506	0,38711
Tous les projets -Email	**MART**	0,30776	0,26190	0,31407	0,26852
	RankNet	0,09550	0,07252	-0,02473	-0,01782
	Coordinate Ascent	0,08638	0,06508	0,39446	0,32071
	OrdinaryLeastSquare	0,48098	0,38256	0,48558	0,38800
	GradientTreeBoost	0,48098	0,38256	0,48558	0,38800
	RidgeRegression	0,48127	0,38277	0,48546	0,38791
	LASSO	0,48098	0,38256	0,48558	0,38800
	RecursiveLeastSquare	0,48098	0,38256	0,48558	0,38800
Tous les projets - IO	**MART**	0,43571	0,39375	0,40674	0,35905
	RankNet	0,19551	0,15533	-0,26152	-0,20771
	Coordinate Ascent	0,22744	0,18296	0,37848	0,30159
	OrdinaryLeastSquare	0,41813	0,33772	0,34216	0,27130
	GradientTreeBoost	0,41813	0,33772	0,34216	0,27130
	RidgeRegression	0,41813	0,33772	0,34216	0,27130
	LASSO	0,41813	0,33772	0,34216	0,27130
	RecursiveLeastSquare	0,41813	0,33772	0,34216	0,27130
Tous les projets - iValidator	**MART**	0,39832	0,33859	0,39223	0,33828
	RankNet	-0,45060	-0,34833	0,15393	0,11453

	Coordinate Ascent	0,10455	0,07784	0,38072	0,30794
	OrdinaryLeastSquare	0,46954	0,37284	0,47038	0,37319
	GradientTreeBoost	0,46954	0,37284	0,47038	0,37319
	RidgeRegression	0,46954	0,37284	0,47038	0,37319
	LASSO	0,46954	0,37284	0,47038	0,37319
	RecursiveLeastSquare	0,46954	0,37284	0,47038	0,37319
Tous les projets - Jaqlib	**MART**	0,29841	0,25101	0,28277	0,21643
	RankNet	0,50845	0,39363	0,49236	0,38297
	Coordinate Ascent	0,50625	0,38588	-0,20376	-0,15544
	OrdinaryLeastSquare	0,54812	0,42811	0,52339	0,41437
	GradientTreeBoost	0,54812	0,42811	0,52339	0,41437
	RidgeRegression	0,54812	0,42811	0,52339	0,41437
	LASSO	0,54812	0,42811	0,52339	0,41437
	RecursiveLeastSquare	0,54812	0,42811	0,52339	0,41437

En observant ces tableaux, on constate que les algorithmes de régression donnent des corrélations plus stables que les algorithmes de LTR sur l'ensemble des projets et par rapport aux deux techniques de validation. Effectivement, les corrélations obtenues par ces algorithmes sont toutes positives et sont supérieures à 0,3 pour la majorité des projets.

Concernant les algorithmes de LTR, l'algorithme MART est celui donnant les résultats les plus stables et les plus élevés. Les résultats concernant la validation 10-folds obtenus sont toutefois plus stables que pour la validation inter-projet. Par exemple, les corrélations de Spearman concernant les métriques TLOC et TINVOK sont toutes positives et supérieures à 0,39 pour la validation 10-fods. Toutefois, pour la validation inter-projet, elles sont parfois hautes comparativement aux modèles de regression (P.ex 0,80029 comparativement à 0,54268 pour la corrélation de Spearman concernant le TLOC du projet IO-Nextgen), parfois basses comparativement aux modèles de regression (P.ex 0,21391 comparativement à 0,45517 pour la corrélation de Spearman concernant le TLOC du projet Tous les projets - Exec) et parfois négatives (-0,05075 pour la corrélation de Spearman concernant le TLOC du projet iValidator-Email). Les algorithmes Ranknet et Coordinate Ascent ne semblent pas être de bons algorithmes à utiliser avec de tels jeux de données. Effectivement, leurs corrélations sont parfois élevées et positives, parfois plus faibles et d'autres fois négatives. Par exemple, l'algorithme Ranknet a déterminé une corrélation de 0,56862 pour la métrique TLOC du projet Exec entraîné à l'aide des données du projet IO, de -0,57114 pour le projet IValidator entraîné à l'aide des données du projet IO et de 0,09550 pour le projet Email entraîné à l'aide de tous les projets.

En observant les deux précédents tableaux de corrélations (3.10 et 3.11), on constate toutefois que la force des corrélations retenues (algorithmes de régression et MART) varie d'un projet à l'autre pour les deux techniques de validation (P.ex, une corrélation de Spearman de 0,73780 pour TLOC iValidator-Nextgen vs de 0,41813 pour TLOC de Tous les projets - IO). Ceci peut probablement s'expliquer par les différences de style et de qualité des tests des différents projets. On constate, par ailleurs, que la majorité des algorithmes retenus combinés à la validation 10-folds ne donnent pas de résultats concluants pour le projet Jaqlib (voir tableau 3.10). Ceci s'explique probablement par la grande quantité de classes non testées dans ce projet combinée à la qualité des tests. Cette dernière raison pourrait aussi expliquer en partie les plus faibles corrélations des algorithmes comprenant ce projet dans les données d'entraînement (projets impliquant la combinaison « Tous les projets » du tableau 3.11).

Entre tous les modèles de priorisation à l'essai dans cette exploration, les algorithmes de régression combinés à la validation inter-projet pour des projets d'une même librairie (librairie Apache) sont les modèles qui donnent les résultats les plus intéressants avec des corrélations stables et positives dépassant 0,6 (0,62450 pour Apache-Email, 0,61326 pour Apache-Exec, 0,61327 pour Apache-Nextgen). Ces résultats sont très intéressants et laissent penser que cette technique pourrait être utilisée par les programmeurs dans leurs différents projets afin de prioriser l'effort de test dans un nouveau projet.

Par la suite, et afin de vérifier les deux hypothèses de cette étude, les résultats des priorisations des différents modèles ont été étudiés. Les tableaux 3.12, 3.13, 3.14, 3.15 et 3.16 présentent les classements résultants des classes de certains projets retenus.

Le tableau 3.12 présente les résultats d'un des classements (1-fold) pour le projet IO avec l'algorithme RecursiveLeastSquare et la validation 10-folds. Le tableau 3.13 présente, quant à lui, les résultats d'un des classements (1-fold) pour les projets Apache combinés avec l'algorithme OrdinaryLeastSquare et la validation 10-folds. Dans ces tableaux, on retrouve le TLOC réel, le score obtenu par l'algorithme, le classement réel (correct), le classement obtenu par l'algorithme (rank), la distance entre ces deux classements et les valeurs de MDC pour chacune des classes d'un projet (NA, NM, NDEP).

Tableau 3.11
Résultats d'un classement (1 fold) avec l'algorithme RecursiveLeastSquare sur les classes du projet IO avec validation 10-folds

Classe	TLOC	score	correct	rank	Dist.	NA	NM	NDEP
BoundedInputStream	50	220,867	2	0	2	5.0	12.0	0.0
LockableFileWriter	168	169,536	0	1	-1	2.0	10.0	2.0
CharSequenceReader	86	135,407	1	2	-1	3.0	8.0	0.0
FileEntry	0	118,482	5	3	2	8.0	6.0	0.0
ThreadMonitor	43	54,248	3	4	-1	2.0	4.0	0.0
DelegateFileFilter	0	35,033	5	5	0	2.0	3.0	1.0
NameFileFilter	0	35,033	5	5	0	2.0	3.0	2.0
AbstractFileFilter	0	26,431	5	7	-2	0.0	3.0	1.0
NullOutputStream	19	26,431	4	7	-3	0.0	3.0	0.0
CanReadFileFilter	0	-11,998	5	9	-4	0.0	1.0	5.0

Tableau 3.12
Résultats d'un classement (1 fold) avec l'algorithme OrdinaryLeastSquare sur les classes des projets Apache avec validation 10-folds

Classe	TLOC	score	correct	rank	Dist.	NA	NM	NDEP
FileUtils	1876	1013,652	0	0	0	0.0	73.0	11.0
BOMInputStream	285	225,816	2	1	1	8.0	13.0	2.0
DefaultExecutor	577	189,982	1	2	-1	7.0	11.0	10.0
ProxyInputStream	0	151,218	8	3	5	0.0	12.0	0.0
Registre	141	131,476	4	4	0	3.0	9.0	7.0
FileEntry	0	126,848	8	5	3	8.0	6.0	0.0
Tailer	157	104,176	3	6	-3	5.0	6.0	2.0
StringBuilderWriter	90	102,223	5	7	-2	1.0	8.0	0.0
ByteArrayDataSource	0	67,365	8	8	0	2.0	5.0	0.0
FileDeleteStrategy	85	45,670	6	9	-3	1.0	4.0	1.0
DelegateFileFilter	0	39,089	8	10	-2	2.0	3.0	1.0
CompositeFileComparator	47	24,951	7	11	-4	2.0	2.0	1.0
EmailAttachment	0	19,346	8	12	-4	5.0	0.0	0.0
WildcardFilter	0	9,836	8	13	-5	0.0	2.0	2.0

En observant ces deux premiers tableaux, on constate que la plupart des classes avec un TLOC élevé ont aussi un score élevé. Toutefois, quelques exceptions de classes n'ayant pas été testées (valeur de TLOC à 0) possèdent des scores relativement élevés. Ces deux phénomènes peuvent être observés pour tous les projets, algorithmes et méthodes de validation testées à l'exception du projet Jaqlib.

Les résultats du projet Jaqlib ont été analysés plus en détail. Le tableau 3.14 présente les résultats d'un des classements (1-fold) des classes de ce projet avec l'algorithme OrdinaryLeastSquare et la validation 10-folds.

Tableau 3.13
Résultats d'un classement (1 fold) avec l'algorithme OrdinaryLeastSquare sur les classes du projet Jaqlib avec validation 10-folds

Classe	TLOC	score	correct	rank	Dist.	NA	NM	NDEP
StandardValueObjectFactory	0	38,887	1	0	1	5.0	20.0	18.0
Condition	0	32,962	1	1	0	1.0	19.0	18.0
ShortTypeHandler	0	9,534	1	2	-1	3.0	4.0	6.0
DateTypeHandler	0	8,070	1	3	-2	0.0	5.0	0.0
RowIdTypeHandler	0	5,073	1	4	-3	2.0	2.0	1.0
CharacterSaveConversion	0	5,073	1	4	-3	2.0	2.0	6.0
WhereClause	50	5,073	0	4	-4	2.0	2.0	4.0
DefaultsDelegate	0	5,073	1	4	-3	2.0	2.0	4.0
SingleConnectionDataSource	0	5,073	1	4	-3	2.0	2.0	0.0
InClause	0	4,018	1	9	-8	1.0	2.0	7.0
ExceptionUtil	0	3,371	1	10	-9	2.0	1.0	5.0
DbUtil	0	2,962	1	11	-10	0.0	2.0	1.0
IsGreaterThanOrEqualTo	0	2,962	1	11	-10	0.0	2.0	1.0
QueryCache	0	2,962	1	11	-10	0.0	2.0	1.0
JdkProxy	0	2,315	1	14	-13	1.0	1.0	3.0
DbInsertDataSource	0	0,612	1	15	-14	1.0	0.0	0.0
AbstractQueryFactory	0	0,443	1	16	-15	0.0	0.0	1.0

En analysant ces résultats, on constate, comme suspecté, que le haut taux de classes non testées semble être en cause dans la faiblesse du classement des classes de ce projet. Les résultats des autres classements (folds) concernant cette expérimentation sont tous semblables à ceux du tableau 3.14.

Les tableaux 3.15 et 3.16 présentent les résultats concernant deux modèles basés sur la validation inter-projet. Le tableau 3.15 présente le classement des classes du projet Nextgen selon TLOC calculé à partir d'un modèle utilisant l'algorithme LASSO et entraîné avec le projet iValidator. Le tableau 3.16 présente le classement des classes du projet iValidator selon TLOC calculé à partir d'un modèle utilisant l'algorithme RigdeRegression et entraîné avec le projet IO.

Tableau 3.14
Résultats du classement avec l'algorithme LASSO sur les classes du projet Nextgen avec validation inter-projet iValidator-Nextgen

Classe	TLOC	score	correct	rank	dist.	NA	NM	NDEP
Systeme	97	166,343	1	0	1	6.0	22.0	2.0
Registre	141	67,353	0	1	-1	3.0	9.0	7.0
Vente	50	49,755	2	2	0	2.0	7.0	2.0
LogSingleton	27	21,160	4	3	1	3.0	2.0	3.0
SingleLog	0	12,359	8	4	4	1.0	2.0	1.0
LigneArticles	24	10,161	5	5	0	2.0	1.0	1.0
SpecificationProduit	33	7,962	3	6	-3	3.0	0.0	0.0
NoLog	0	7,959	8	7	1	0.0	2.0	1.0
CatalogueProduits	22	5,760	6	8	-2	1.0	1.0	1.0
Magasin	14	5,760	7	8	-1	1.0	1.0	1.0

Tableau 3.15
Corrélations de Spearman et Kendall entre les scores et les métriques d'effort de test avec validation inter-projet IO-iValidator

Classe	TLOC	score	corr.	rank	dist.	NA	NM	NDEP
FileUtils	1876	476,083	0	0	0	0.0	73.0	11.0
IOUtils	1150	399,177	1	1	0	2.0	60.0	3.0
FileFilterUtils	0	280,500	63	2	61	2.0	42.0	17.0
FilenameUtils	983	245,324	2	3	-1	0.0	38.0	1.0
EndianUtils	222	192,579	11	4	7	0.0	30.0	0.0
BOMInputStream	285	115,712	8	5	3	8.0	13.0	2.0
NullInputStream	164	104,717	17	6	11	7.0	12.0	0.0
ByteArrayOutputStream	94	102,506	24	7	17	5.0	13.0	1.0
NullReader	161	98,123	18	8	10	7.0	11.0	0.0
FileAlterationObserver	287	98,104	6	9	-3	4.0	13.0	5.0
BoundedInputStream	50	95,913	40	10	30	5.0	12.0	0.0
DirectoryWalker	365	93,702	4	11	-7	3.0	13.0	5.0
SwappedDataInputStream	94	93,682	24	12	12	0.0	15.0	2.0
FileSystemUtils	456	87,109	3	13	-10	3.0	12.0	4.0
CollectionFileListener	0	84,937	63	14	49	7.0	9.0	2.0
IOCase	286	80,516	7	15	-8	3.0	11.0	1.0
ProxyReader	91	80,496	27	16	11	0.0	13.0	0.0
ProxyWriter	49	80,496	41	16	25	0.0	13.0	0.0
ThresholdingOutputStream	0	73,922	63	18	45	3.0	10.0	0.0
ProxyInputStream	0	73,903	63	19	44	0.0	12.0	0.0
FileEntry	0	69,560	63	20	43	8.0	6.0	0.0
LockableFileWriter	168	69,520	15	21	-6	2.0	10.0	2.0
WriterOutputStream	84	62,947	32	22	10	5.0	7.0	0.0
FileCleaningTracker	230	62,947	10	22	-12	5.0	7.0	3.0
CharSequenceReader	86	60,736	29	24	5	3.0	8.0	0.0
NullWriter	126	60,716	21	25	-4	0.0	10.0	0.0
CopyUtils	18	60,716	57	25	32	0.0	10.0	0.0
DeferredFileOutputStream	240	58,565	9	27	-18	7.0	5.0	2.0
Tailer	157	56,354	19	28	-9	5.0	6.0	2.0
LineIterator	293	54,143	5	29	-24	3.0	7.0	0.0

FileWriterWithEncoding	90	51,932	28	30	-2	1.0	8.0	0.0
StringBuilderWriter	165	51,932	16	30	-14	1.0	8.0	2.0
FileAlterationMonitor	105	49,761	22	32	-10	5.0	5.0	1.0
ReaderInputStream	72	47,570	35	33	2	6.0	4.0	0.0
ByteOrderMark	77	47,550	33	34	-1	3.0	6.0	0.0
ProxyOutputStream	30	47,530	48	35	13	0.0	8.0	0.0
OrFileFilter	214	45,339	12	36	-24	1.0	7.0	3.0
AndFileFilter	214	45,339	12	36	-24	1.0	7.0	3.0
XmlStreamWriter	92	43,168	26	38	-12	5.0	4.0	1.0
FileCleaner	11	38,746	61	39	22	1.0	6.0	2.0
ExtensionFileComparator	27	34,383	52	40	12	6.0	2.0	4.0
BrokenInputStream	130	32,153	20	41	-21	1.0	5.0	1.0
CountingInputStream	65	32,153	37	41	-4	1.0	5.0	0.0
TeeOutputStream	0	32,153	63	41	22	1.0	5.0	1.0
TeeInputStream	65	29,962	37	44	-7	2.0	4.0	1.0
ThreadMonitor	43	29,962	44	44	0	2.0	4.0	0.0
FileDeleteStrategy	85	25,560	30	46	-16	1.0	4.0	1.0
CountingOutputStream	67	25,560	36	46	-10	1.0	4.0	1.0
PrefixFileFilter	0	23,369	63	48	15	2.0	3.0	1.0
HexDump	16	23,369	59	48	11	2.0	3.0	1.0
DelegateFileFilter	185	23,369	14	48	-34	2.0	3.0	0.0
SuffixFileFilter	0	23,369	63	48	15	2.0	3.0	2.0
NameFileFilter	0	23,369	63	48	15	2.0	3.0	2.0
YellOnFlushAndCloseOutputStream	0	23,369	63	48	15	2.0	3.0	2.0
TaggedIOException	0	23,369	63	48	15	2.0	3.0	1.0
NotFileFilter	85	18,967	30	55	-25	1.0	3.0	2.0
TaggedOutputStream	49	18,967	41	55	-14	1.0	3.0	0.0
BrokenOutputStream	0	18,967	63	55	8	1.0	3.0	2.0
TaggedInputStream	98	18,967	23	55	-32	1.0	3.0	2.0
XmlStreamReaderException(input.)	0	16,795	63	59	4	5.0	0.0	0.0
MagicNumberFileFilter	47	16,775	43	60	-17	2.0	2.0	1.0
AgeFileFilter	0	16,775	63	60	3	2.0	2.0	2.0
SizeFileFilter	0	16,775	63	60	3	2.0	2.0	1.0
FalseFileFilter	0	16,775	63	60	3	2.0	2.0	1.0
CompositeFileComparator	0	16,775	63	60	3	2.0	2.0	1.0
AbstractFileComparator	62	14,565	39	65	-26	0.0	3.0	2.0
NullOutputStream	0	14,565	63	65	-2	0.0	3.0	3.0
WildcardFileFilter	0	14,565	63	65	-2	0.0	3.0	0.0
AutoCloseInputStream	0	14,565	63	65	-2	0.0	3.0	1.0
AbstractFileFilter	19	14,565	55	65	-10	0.0	3.0	0.0
SizeFileComparator	30	12,373	48	70	-22	1.0	2.0	3.0
ReverseComparator	30	12,373	48	70	-22	1.0	2.0	3.0
PathFileComparator	0	12,373	63	70	-7	1.0	2.0	1.0
NameFileComparator	42	12,373	45	70	-25	1.0	2.0	3.0
LastModifiedFileComparator	38	10,182	46	74	-28	2.0	1.0	2.0
TrueFileFilter	0	7,971	63	75	-12	0.0	2.0	2.0
DirectoryFileComparator	24	7,971	53	75	-22	0.0	2.0	2.0
WildcardFilter	0	7,971	63	75	-12	0.0	2.0	1.0
RegexFileFilter	21	5,780	54	78	-24	1.0	1.0	0.0
ClassLoaderObjectInputStream	75	5,780	34	78	-44	1.0	1.0	2.0
EmptyFileFilter	12	1,378	60	80	-20	0.0	1.0	0.0
CloseShieldOutputStream	0	1,378	63	80	-17	0.0	1.0	2.0
CloseShieldInputStream	17	1,378	58	80	-22	0.0	1.0	2.0
CanReadFileFilter	0	1,378	63	80	-17	0.0	1.0	3.0

ClosedOutputStream	0	1,378	63	80	-17	0.0	1.0	3.0
FileFileFilter	0	1,378	63	80	-17	0.0	1.0	2.0
YellOnCloseInputStream	29	1,378	51	80	-29	0.0	1.0	2.0
CanWriteFileFilter	0	1,378	63	80	-17	0.0	1.0	3.0
ClosedInputStream	7	1,378	62	80	-18	0.0	1.0	0.0
DefaultFileComparator	0	1,378	63	80	-17	0.0	1.0	1.0
HiddenFileFilter	0	1,378	63	80	-17	0.0	1.0	5.0
DirectoryFileFilter	32	1,378	47	80	-33	0.0	1.0	2.0
IOExceptionWithCause	19	0,813	55	92	-37	1.0	0.0	0.0
XmlStreamReaderException(compatibility.)	0	0,813	63	92	-29	1.0	0.0	1.0
FileExistsException	0	-5,215	63	94	-31	0.0	0.0	0.0

En observant les résultats de ces tableaux, on constate que les classes demandant un plus grand effort sont classées en premier (System, Registre et Vente dans iValidator-Nextgen ainsi que FileUtils, IOUtils et FilenameUtils dans IO-iValidator). On remarque également que la plupart des classes avec un TLOC de 0 sont classées dans des rangs assez élevés, mais qu'il y a toutefois des exceptions.

3.4 Conclusion

Concernant la première hypothèse portant sur la validation de la significativité des modèles de priorisation, on peut affirmer que les modèles bâtis à partir des algorithmes de régression et de la validation 10-folds donnent tous des résultats significatifs sur les projets testés. On peut également affirmer que tous les modèles bâtis à partir des algorithmes de régression et de la validation inter-projet donnent des résultats significatifs sur les projets sur lesquels ils ont été testés. Toutefois, il serait intéressant de tester ces modèles sur plus de projets afin de voir si ces résultats peuvent être généralisés.

Concernant les algorithmes LTR, les modèles bâtis à partir de l'algorithme MART donnent des résultats stables et significatifs pour la validation 10-folds. Cependant, ce même algorithme donne des résultats beaucoup moins stables pour la validation inter-projet. Il est possible que cet algorithme soit plus sensible aux variations de style et de qualité entre les projets. Il serait intéressant de vérifier cette hypothèse dans un travail futur. Les deux autres algorithmes, soit Ranknet et Coordinate Ascent, donnent des résultats instables pour les deux types de validations.

Concernant l'hypothèse 2 de cette étude, soit celle en lien avec les classes non testées, on peut conclure que celles-ci sont classées majoritairement dans les plus hauts rangs pour tous les

algorithmes utilisés dans cette étude. Certaines exceptions sont toutefois observées où des classes non testées se retrouvent dans de faibles rangs.

CHAPITRE 4
PRÉDICTION ET PRIORISATION DE L'EFFORT DE TEST ÉTENDUES À LA PROGRAMMATION ORIENTÉE ASPECT

4.1 État de L'art

Plusieurs recherches ont été menées sur l'impact de la POA sur les systèmes POO [87-92] aussi bien au niveau qualitatif que quantitatif. La POA possède plusieurs avantages notamment l'amélioration de la modularité des programmes, l'augmentation de leur productivité, la réduction de leur complexité et l'amélioration de plusieurs critères de qualité tels que la maintenabilité, la traçabilité, la réutilisabilité et la compréhensibilité en sont des exemples.

Les programmes orientés aspects (OA) et les programmes OO ont plusieurs similarités. Ils partagent des structures similaires comme les classes, interfaces, méthodes, packages, etc. Toutefois, la POA apporte de nouveaux éléments structuraux (aspects, greffons, points de jonction, etc.) et de nouvelles relations entre les éléments d'un programme.

Plusieurs chercheurs se sont donc intéressés aux défis que pose la POA pour les tests logiciels. Alexander et al.[93] ont été les premiers à discuter des défis posés par la POA au niveau des tests logiciels. Ensuite, plusieurs chercheurs ont essayé de relever ces défis. Selon Ferrari et al. [94], la recherche sur les tests des systèmes OA a porté principalement sur les trois aspects suivants : la caractérisation des types de défauts et des modèles de bogues, la définition de modèles de tests et sélection des critères de test et la création d'outils automatisés de génération de tests. D'après ces chercheurs, la POA est généralement utilisée afin de refactoriser (activité de refactoring, remaniement) les SOO existants afin d'obtenir une meilleure modularisation des comportements entrelacés dans un système.

Bien que plusieurs chercheurs se sont intéressés aux tests logiciels des SOA, peu d'études ont été menées à ce jour concernant l'effort de développement des tests dans les SOA.

Ferrari et al. [94] ont aussi étudié les difficultés liées au test des programmes OA et se sont intéressés, entre autres, à l'effort d'adaptation des suites de tests en passant du paradigme OO au paradigme OA. Leurs conclusions sont que la construction de modèles de test OA est plus complexe que celle pour les tests OO, mais que les suites résultantes atteignent des niveaux de

qualité similaires en termes de couverture du code. Ces constats peuvent toutefois dépendre de la taille et des caractéristiques des applications testées.

Munoz et al. [95] se sont intéressés à la testabilité des SOA. La testabilité est une notion complexe décrite par la règle ISO/IEC 25010 comme étant le degré d'efficacité et d'efficience à établir les critères de test d'un système [96, 97]. Munoz et al. ont comparé différentes métriques de testabilité relatives aux implémentations OO et OA d'un même système. Leurs résultats sont que, pour ce système, la POA réduit la taille des modules de test, augmente leur cohésion, mais augmente le couplage global introduisant un impact négatif sur la testabilité du système testé.

Badri et al. ont mené une récente étude [98, 99] portant aussi sur la testabilité des SOA. Leur étude était basée sur une autre étude exploratoire antérieure [98], menée par ces mêmes chercheurs portant sur l'effet de la refactorisation d'un programme OO en programme OA, sur la testabilité des classes logicielles. Dans leurs deux études, ils se sont intéressés à la testabilité unitaire des classes. Dans leur seconde étude, ils se sont intéressés, plus particulièrement, à l'impact de la refactorisation sur les attributs du code source d'une part, sur l'impact de la refactorisation sur les attributs des TU d'autre part, et aux relations entre les variations observées après refactorisation au niveau des attributs du code source et des attributs du code des TU. Les résultats, portant sur trois études de cas, suggèrent que l'effort nécessaire à la construction des CT des classes refactorisées est réduit. Par ailleurs, les variations des attributs du code source ont plus d'impact sur l'invocation des méthodes entre les CT et les variations des attributs du code des TU sont plus influencées par la variation de la complexité des classes refactorisées que par les autres attributs de classe.

4.2 Objectif

La construction de modèles de tests OA a été un sujet de recherche important dans les dernières années, ce qui n'a pas été, malheureusement, le cas de l'effort de développement des tests dans ce domaine. Peu d'études ont porté sur ce sujet et les résultats de ces différentes études ne concordent pas tous. De plus, aucune étude ne semble s'être intéressée jusqu'à présent à la prédiction de l'effort de test pour les SOA ni à la priorisation de l'effort de test dans les SOA. Toutefois, la prise en compte des préoccupations transversales et la prédiction de l'effort de test tôt

dans le cycle de développement de ces systèmes serait un grand avantage pour les développeurs utilisant ce paradigme.

L'objectif de cette étude est l'exploration de la possibilité de la prédiction et de la priorisation de l'effort de test des SOA à partir des modèles. Plus précisément, il s'agit de vérifier dans cette étude préliminaire l'impact de la refactorisation de programmes OO en programmes OA sur les métriques des modèles de CU. Les différentes sections de ce rapport présentent la description de la méthodologie utilisée, les résultats obtenus et une conclusion.

4.3 Méthodologie

4.3.1 Démarche expérimentale

Les différentes étapes (8) de notre démarche expérimentale sont :

1. Modélisation par rétro-ingénierie du système OO
2. Modélisation du système OA
3. Refactorisation du SOO en SOA
4. Développement des tests du SOA
5. Collecte des métriques de modèles
6. Collecte des métriques OO
7. Collecte des métriques de test
8. Analyse des données

4.3.1.1 Études de cas

4.3.1.1.1 Études de cas OO

Afin de vérifier l'impact produit sur les métriques du DCU et de l'effort de test par la refactorisation de programmes OO en programmes OA, une étude de cas Java (Nextgen_OO) a été utilisée à des fins d'exploration. La programmation OO et les tests existants de cette étude de cas ont été développés par des étudiants à la maîtrise du département de Mathématiques et Informatique de l'UQTR (activités du laboratoire GLOG). Le code source ainsi que les tests ont toutefois été adaptés pour les besoins de cette étude afin que le comportement de certaines préoccupations transversales soit développé et testé.

4.3.1.1.2 Études de cas OA

Une refactorisation du programme OO NextGen en programme OA a été effectuée dans ce cadre à l'aide de l'extension AspectJ [100]. AspectJ est une extension au langage Java pour la POA. Certains comportements transversaux et éparpillés dans le code OO ont été remplacés par des aspects qui sont des unités modulaires permettant d'encapsuler certaines préoccupations transverses d'un programme. Deux aspects ont été développés pour le système Nextgen.

4.3.1.2 Stratégie de développement des tests du SOA

La stratégie de test unitaire que nous avons utilisée consiste à tester de manière isolée les plus petites entités d'un système [101]. Les méthodes et les greffons ont donc été considérés comme étant les plus petites unités des SOA. Les TU des classes déjà existantes ont donc été modifiés afin que les classes affectées par la refactorisation OA soient à nouveau testées d'une manière unitaire. Ensuite, chaque aspect du système a aussi été testé de manière unitaire (méthodes et greffons).

Dans cette étude, toutes les parties du greffon ont été considérées : greffon (before, after, around), points de coupure, paramètres et code des greffons.

4.3.1.3 Modélisation des systèmes OO

Les modèles utilisés pour l'étude de cas Nextgen OO sont les diagrammes de séquences système (DSS). Plus précisément, une forme de DSS simplifiée par un tableau représentant tous les CU ainsi que leurs opérations a été utilisée. Le DSS de l'étude Nexgen_OO a été initialement produit par rétro-ingénierie à partir du code source OO puis adapté, dans le cadre de notre approche, afin de tenir compte des comportements non-fonctionnels ajoutés ou complétés.

Le tableau 4.1 présente un exemple de DSS du CU creerNouvelleVente de l'étude de cas NextGen_OO. Les quatre colonnes de ce tableau présentent respectivement le nom du CU, les différentes classes objets, le point de départ du CU et les différentes méthodes impliquées. Les méthodes comprenant un seul chiffre en préfixe sont des méthodes externes faisant donc partie de l'interface du CU tandis que les méthodes comprenant plusieurs chiffres en préfixe sont des sous

méthodes appelées par les méthodes externes ou par d'autres méthodes internes. Les méthodes portant le nom « create » représentent des appels de constructeurs. Les expressions « > Branch » et « > Canfail » parfois suivies d'un multiplicateur (ex : x2) peuvent suivre une méthode. Canfail représente un cas d'erreur qui cause la fin prématurée du CU tandis que Branch représente un branchement dans le scénario du CU signifiant ainsi deux chemins possibles.

Tableau 4.1
DSS représentant le CU CreerNouvelleVente de l'étude de cas NextGen_OO

CreerNouvelleVente	*Registre*	X	1.creerNouvelleVente()
	Systeme		1.1.authentification() > Branch
	Log		1.1.1.write()
	Vente		1.2.create
	Log		1.3.write()

4.3.1.4 Modélisation des systèmes OA

Le modèle produit concernant le système Nextgen après aspectualisation est une seconde version du DSS simplifié intégrant les aspects comme étant des objets impliqués dans l'exécution de certains CU.

Le tableau 4.2 comprend un exemple de DSS du CU creerNouvelleVente de l'étude de cas NextGen_OA. Il est possible de remarquer que des aspects ont été ajoutés à ce DSS. Afin de faire la distinction entre les classes objet et les aspects, les notations O et A ont été ajoutées à la suite du nom de ces éléments. Les lettres B et A ont aussi été ajoutées aux préfixes de certaines méthodes. La lettre B correspond aux greffons de type « *before* » et la lettre A correspond aux greffons de type « *after* ». Dans cette étude, seulement ces deux types de greffons ont été utilisés. Dans le tableau suivant, les méthodes ayant un seul chiffre en préfixe sont donc aussi des méthodes externes et les méthodes ayant plusieurs chiffres en préfixes sont aussi des méthodes internes. Toutefois, les méthodes comprenant les lettres B et A sont des méthodes faisant intervenir des aspects comprenant des greffons intervenant avant ou après la méthode. Certains aspects font aussi intervenir des méthodes internes (p.ex : Authentication_A – 1.B.authentification()) pouvant appartenir à l'aspect lui-même ou à une classe objet.

Tableau 4.2
DSS représentant le CU CreerNouvelleVente de l'étude de cas NextGen_OA

CreerNouvelleVente	*Registre_O*	X	1.creerNouvelleVente()
	Authentication_A		1.B.authentification()> Branch
	Journalisation_A		1.B.A.write()
	Vente_O		1.1.create
	Journalisation_A		1.1.A.write()

4.3.2 Métriques compilées

Afin d'explorer l'impact de la POA sur les métriques de classes, sur les modèles et sur l'effort de test et d'explorer la suite de métriques pouvant encapsuler certaines caractéristiques propres à la POA, trois différentes classes de métriques ont été utilisées dans ce contexte: des métriques de modèles, des métriques de classes et des métriques de test.

De plus, deux sommations de métriques distinctes ont été effectuées. Une première sommation concerne toutes les métriques appartenant aux trois types de métriques calculées pour le système global. Ainsi, le nombre de classes impliquées, par exemple, correspond au nombre de classes impliquées dans le système complet (pour tous les CU à l'étude). La seconde concerne les métriques de modèle et la métrique TLOC calculées de manière unitaire à chaque CU.

4.3.5.1 Métriques de modèles

Les métriques de modèles compilées dans cette étude sont les suivantes :

- *NCI (Nombre de classes impliquées)* : cette métrique correspond au nombre de classes impliquées dans le CU.
- NAI *(Nombre d'aspects impliqués)* : cette métrique correspond au nombre d'aspects impliqués dans le CU.
- NEI (*Nombre d'éléments impliqués*) : cette métrique correspond au nombre d'éléments (classes et aspect) impliqués dans un CU après aspectualisation.
- *NOE (Nombre d'opérations externes)* : cette métrique correspond au nombre d'opérations externes d'un CU, c'est-à-dire au nombre d'opérations faisant partie de l'interface du CU.

- *NMI_OO (Nombre de méthodes impliquées orientées objet)* : cette métrique correspond au nombre de méthodes orientées objet impliquées d'un CU. Les constructeurs n'ont pas été considérés comme des méthodes et ne sont donc pas compilés dans cette métrique.
- *NMAI (Nombre de méthodes aspect impliquées)* : cette métrique correspond au nombre de méthodes aspects impliquées dans un CU.
- *NMI_OA (Nombre de méthodes impliquées orientées aspect)* : cette métrique correspond à la somme du nombre de méthodes impliquées orientées objet (NIM_OO) et du nombre de méthodes aspect impliquées (NMA_I) d'un CU.
- *NGREFI (Nombre de greffons impliqués)* : cette métrique correspond au nombre de greffons impliqués dans un CU.
- *NOPI (Nombre d'opérations impliquées)* : cette métrique correspond au nombre d'opérations impliquées dans un CU après aspectualisation. Les méthodes de classes et d'aspects ainsi que les greffons sont les trois types d'opérations calculées dans cette étude.
- *NS (Nombre de scénarios)* : cette métrique correspond au nombre de scénarios du CU incluant le scénario principal.

Ces métriques ont été compilées manuellement à partir des DSS à l'aide de la méthode du test-retest, c'est-à-dire que les calculs ont été effectués plus d'une fois pour une même métrique afin de s'assurer de sa justesse.

4.3.5.2 Métriques de code source

Les métriques de classes sont les suivantes :

- *SLOC _OO (Nombre de lignes de code orienté objet)* : cette métrique correspond au nombre de lignes de code source des classes OO.
- *SLOC_OA (Nombre de lignes de code orienté aspect)* : cette métrique correspond au nombre de lignes de code source des classes OO et des aspects combinés.
- *NM (Nombre de méthodes)* : cette métrique correspond au nombre de méthodes OO.

- *NOP_OA (Nombre d'opérations)* : cette métrique correspond au nombre d'opérations d'un système OA. Les méthodes de classes, les méthodes d'aspects ainsi que les greffons sont considérés comme des opérations.
- *SLOC/M (Nombre de lignes de code moyen / méthode)* : cette métrique correspond au nombre de lignes de code moyen par méthode OO.

L'outil CodePro Analytix a été utilisé pour compiler ces métriques de code lorsqu'il était possible de l'utiliser.

4.3.5.2 Métriques de test

Les métriques de test compilées pour chaque système de cette étude sont :

- *TLOC_OO (nombre de lignes de test orienté objet) :* cette métrique correspond au nombre de lignes de code de test des CT liées aux classes OO.
- *TLOC_OA (nombre de lignes de test orienté aspect) :* cette métrique correspond au nombre de lignes de code de test des CT liées aux classes OO et aux aspects.
- *TLOC/M (nombre de lignes de test moyen / méthode de test) :* cette métrique correspond au nombre de lignes de code de test moyen par méthode.

L'outil CodePro Analytix a aussi été utilisé pour compiler ces métriques lorsqu'il était possible de l'utiliser.

4.3.3 Hypothèses

Deux hypothèses ont été vérifiées dans cette section d'étude.

La première hypothèse concerne l'impact de la refactorisation OA sur chacune des métriques du système global (MS_i) et a été vérifiée pour tous les types de métriques (T_j).

Voici les hypothèses à propos de celui-ci :

- Hypothèse : La métrique MS_i appartenant au type T_j n'est pas impactée par la refactorisation OA du système
- Hypothèse nulle : La métrique MS_i appartenant au type T_j est impactée par la refactorisation OA du système

La seconde hypothèse concerne l'impact de la refactorisation OA sur les métriques unitaires des CU d'un système (MU_i) et a été vérifiée pour les métriques de modèles et pour la métrique de test TLOC. Voici les hypothèses à propos de celui-ci :

- Hypothèse : La métrique MU_i n'est pas impactée par la refactorisation OA du système
- Hypothèse nulle : La métrique MU_i est impactée par la refactorisation OA du système

4.4 Résultats

Afin de pouvoir analyser les différences entre les différentes métriques avant et après aspectualisation des études de cas, la lettre B ou A (B pour before et A pour after) a été ajoutée à la fin de toutes les métriques calculées avant et après aspectualisation. Ainsi, par exemple, la métrique NCI_B signifie le nombre de classes impliquées avant aspectualisation et la métrique NCI_A signifie le nombre de classes impliquées après aspectualisation.

4.4.1 Résultats concernant les métriques globales avant et après implémentation de la POA

Les résultats concernant les métriques globales des systèmes Nextgen_OO et Nextgen_OA ont été compilés dans 3 tableaux distincts (tableaux 4.3, 4.4 et 4.5) : respectivement les métriques de modèles, de classes et de test avant et après aspectualisation.

Tableau 4.3
Résultats concernant les métriques globales de modèles du système Nextgen avant et après aspectualisation

NCI_B	7	**NOE_B**	12	**NMI_OO_B**	33	**NS_B**	19
NCI_A	6	**NOE_A**	12	**NMI_OO_A**	15	**NS_A**	19
NAI	2			**NMAI**	18		
NEI	8			**NMI_OA**	33		
				NGREFI	18		
				NOPI	51		

Tableau 4.4
Résultats concernant les métriques globales de classes du système Nextgen avant et après aspectualisation

SLOC_OO_B	318	**NM_B**	53	**SLOC/M_B**	4,26
SLOC_OO_A	257	**NM_A**	46	**SLOC/M_A**	3,72
SLOC_OA	313	**NOP_OA**	57		

Tableau 4.5
Résultats concernant les métriques globales de modèles du système Nextgen avant et après aspectualisation

TLOC_OO_B	299	**TLOC/M_B**	5,73
TLOC_OO_A	253	**TLOC/M_A**	5,26
TLOC_OA	401		

Dans le tableau 4.3, on constate que le nombre de classes (NCI) a diminué après aspectualisation, toutefois le nombre d'éléments total après aspectualisation (NEI) est plus grand que celui avant aspectualisation (correspondant au NCI_B). On constate aussi que le nombre de méthodes OO (NMI_OO) a diminué après aspectualisation, cependant le nombre de méthodes impliquées OA (NMI_OA) est le même que le nombre de méthodes impliquées avant aspectualisation (NMI_OO_B). Si on ajoute à ce nombre le nombre de greffons impliqués (NGREFI), le nombre total d'éléments impliqués (NEI) augmente de beaucoup après aspectualisation. Le nombre d'opérations externes (NOE) ainsi que le nombre de scénarios (NS) reste toutefois identiques après aspectualisation.

Dans le tableau 4.4, on constate tout d'abord que le nombre de lignes de code OO (SLOC_OO) diminue considérablement après aspectualisation. Le nombre de lignes de code total après aspectualisation (SLOC_OA) diminue aussi, mais plus faiblement. On constate aussi que le nombre de méthodes OO (NM) diminue aussi avec l'aspectualisation, toutefois le nombre d'opérations total (NOP) augmente. Enfin, le nombre moyen de lignes de code par méthodes (SLOC/M) diminue aussi.

Dans le tableau 4.5, on constate que le nombre de lignes de code de test OO (TLOC_OO) ainsi que le nombre de lignes de code de test par méthode (TLOC/M) diminuent avec

l'aspectualisation. Toutefois, le nombre total de lignes de code de test après aspectualisation (TLOC_OA) a augmenté.

4.4.2 Résultats concernant les métriques unitaires avant et après implémentation de la POA

Les résultats concernant les métriques unitaires de modèles des systèmes Nextgen_OO et Nextgen_OA ont été compilés dans 2 tableaux distincts (tableaux 4.6 et 4.7) présentant respectivement les métriques de modèle unitaires avant et après aspectualisation.

Tableau 4.6
Résultats concernant les métriques de modèles unitaires du système Nextgen avant aspectualisation

CU	NCI_B	NOE_B	NMI_OO_B	NS_B	TLOC_B
CU1	1	1	1	1	13
CU2	2	1	2	1	48
CU3	4	1	4	2	96
CU4	2	1	2	1	53
CU5	2	1	2	1	53
CU6	2	1	4	2	71
CU7	2	1	2	1	46
CU8	5	1	4	3	81
CU9	3	1	3	1	69
CU10	2	1	3	2	71
CU11	2	1	3	2	71
CU12	2	1	3	2	71

Tableau 4.7
Résultats concernant les métriques de modèles unitaires du système Nextgen après aspectualisation

CU	NCI_A	NAI	NEI	NOE_A	NMI_OO_A	NMAI	NMI_OA_A	NADVI	NOI	NS_A	TLOC_A
CU1	1	0	1	1	1	0	1	0	1	1	13
CU2	1	1	2	1	1	1	2	0	3	1	64
CU3	2	2	4	1	1	3	4	3	7	2	103
CU4	1	1	2	1	1	1	2	1	3	1	78
CU5	1	1	2	1	1	1	2	1	3	1	78
CU6	1	2	3	1	1	3	4	3	7	2	79
CU7	1	1	2	1	1	1	2	1	3	1	46
CU8	4	1	5	1	3	1	4	1	5	3	129
CU9	2	1	3	1	2	1	3	1	4	1	91
CU10	1	2	3	1	1	2	3	2	5	2	99
CU11	1	2	3	1	1	2	3	2	5	2	99
CU12	1	2	3	1	1	2	3	2	5	2	99

Dans les tableaux 4.6 et 4.7, on constate que les mêmes conclusions formulées au niveau du système peuvent être émises aussi au niveau unitaire des cas d'utilisation. En effet, le nombre de classes impliquées (NCI) pour les CU impactés par l'aspectualisation est plus faible. Toutefois, le nombre d'éléments impliqués (NEI) a augmenté pour certains CU impactés. Le nombre de méthodes impliquées OO diminue aussi pour tous les CU impactés, toutefois le nombre de méthodes OA reste le même. On observe, également, une augmentation du nombre d'opérations impliquées au niveau des CU impactés et une augmentation du nombre de lignes de code de test pour tous les CU sauf un. Par ailleurs, on constate que les métriques de modèle OO qui semblent influencer le plus TLOC sont les métriques NCI, NMI et NS. Les métriques OA qui semblent influencer le plus TLOC sont plutôt NEI, NOPI et NS.

4.5 Conclusion

Cette étude constituait une première étape à l'exploration de l'extension de techniques de priorisation à partir des métriques de modèles. Dans cette étude, nous avons analysé l'impact de la refactorisation d'un système OO en un système OA à travers trois familles de métriques (modèle DSS, code source et code de test). Nous avons aussi évalué l'impact de cette refactorisation sur les métriques du modèle DSS au niveau unitaire. De nouvelles métriques de modèle DSS relatives au paradigme OA ont été introduites pour mieux évaluer l'effort en termes de test.

Les résultats obtenus concernant les métriques des modèles de DSS globaux des systèmes avant et après aspectualisation reflètent un comportement identique (NOE et NS identiques). On observe, également, une baisse de complexité OO (NCI et NMI) et une augmentation de la complexité totale (NEI et NOPI).

Les résultats obtenus concernant les métriques de code source globales des systèmes avant et après aspectualisation montrent une diminution faible de l'effort de développement avec des méthodes de classes et d'aspects plus légères. Toutefois, les résultats concernant les opérations totales (NOP) après aspectualisation dénotent une augmentation de la complexité du système.

Les résultats obtenus concernant l'impact de cette refactorisation sur les métriques de code de test globales du système montrent un plus faible effort de test attribué à chaque test individuel du système touché par l'aspectualisation, mais un plus grand effort de test total pour le système global dû à la complexité importante du système OA.

Les résultats obtenus concernant les métriques de modèle DSS unitaire avant et après aspectualisation montrent des DSS plus légers en termes d'éléments OO (classes et méthodes OO), mais plus complexes en termes d'éléments OA (classes, aspects, méthodes OO et OA et advice). De plus, ces résultats montrent une augmentation de l'effort de test au niveau des CU impactés par la refactorisation.

Les métriques de modèle DSS qui semblent le plus influencer l'effort de développement dans le contexte OO sont les métriques NCI, NMI et NS et dans le contexte OA sont NEI, NOPI et NS.

Les résultats obtenus dans cette étude préliminaire laissent croire que l'extension à la POA de technique de priorisation de l'effort à partir des métriques de modèles serait possible. Les résultats ne peuvent toutefois pas être généralisés étant donné qu'ils concernent une seule étude de cas peu complexe. Plusieurs des résultats obtenus dans cette étude peuvent dépendre des comportements refactorisés, taille et complexité des systèmes, technique de refactorisation utilisée par les développeurs et enfin technique de test utilisée. Il serait donc intéressant dans un futur travail de considérer d'autres études comportant des diagrammes complets et appartenant à différents domaines afin de reproduire cette approche. Il serait intéressant de trouver des études de cas déjà refactorisées pour lesquelles il est possible de construire des modèles de prédiction et de

priorisation de l'effort de test pour les CU des SOA. Toutefois, tel que mentionné dans le chapitre 2, une telle démarche pourrait être longue due à des délais de communication, parfois à des refus et au faible nombre d'études de cas refactorisées comprenant des diagrammes complets et de qualité.

Il serait aussi intéressant, dans une étude future, d'évaluer l'impact de la refactorisation OA sur les MDC dans un but de produire une technique de priorisation étendue à la POA à partir des MDC et MDCU.

CHAPITRE 5
CONCLUSIONS

Ce mémoire visait 3 objectifs principaux : (1) le développement d'une technique de priorisation des tests basée sur les CU, (2) le développement d'une technique de prédiction et de priorisation de l'effort de test à partir du diagramme de classes UML et (3) l'exploration de l'extension éventuelle de ces techniques aux systèmes orientés aspect. Afin d'atteindre ces différents objectifs, 3 études exploratoires ont été menées dans ce contexte.

La première de ces 3 études consistait en une étude exploratoire portant sur le développement d'une technique de priorisation des tests basée sur les CU. Dans ce contexte, nous avons proposé une méthodologie basée sur la méthode de prédiction de l'effort de test à partir du DCU de Badri et al. [3] et sur les algorithmes d'apprentissage d'ordonnancement. Toutefois, une grande limitation a été rencontrée dans cette étude, ce qui a empêché la validation empirique de la technique proposée. Cette limitation est due au faible nombre d'études de cas disponibles comprenant des modèles d'analyse complets et ayant un niveau de qualité acceptable. Les chercheurs dans ce contexte doivent composer avec cette limitation et doivent se tourner vers soit la rétro-ingénierie pour créer les modèles, soit vers des études de cas développées par eux-mêmes ou bien être contraints d'utiliser très peu d'études de cas. Toutes ces solutions posent des problèmes dans la validation de leur recherche.

La seconde approche concernait une étude exploratoire portant sur le développement d'une technique de priorisation de l'effort de test à partir des métriques du DC. Dans cette étude, des techniques d'analyses statistiques ont été utilisées afin de déterminer les MDC les plus prometteuses pour la prédiction de l'effort de test et qui sont : le nombre d'attributs (NA), le nombre de méthodes (NM) et le nombre de dépendances (NDEP). Différents algorithmes d'apprentissage d'ordonnancement ont été utilisés afin d'étudier la possibilité d'ordonnancer les classes d'un système OO en termes de prédiction de l'effort de test. Les résultats obtenus montrent que les modèles de priorisation bâtis à partir des algorithmes de régression considérés et validation 10-folds, des algorithmes de régressions et validation inter-projet, et, l'algorithme MART et validation 10-folds sont de bons modèles retenus pour la priorisation de l'effort de test des classes. Il serait toutefois intéressant de tester ces modèles sur plusieurs projets afin de voir si ces résultats peuvent être généralisés.

La dernière démarche consistait en une étude préliminaire exploratoire concernant la possibilité d'étendre la prédiction et la priorisation de l'effort de test à un contexte orienté aspect. Précisément, l'analyse de l'impact de la refactorisation de programmes OO en programmes OA sur les métriques des modèles de CU a été étudiée. De nouvelles métriques de modèle DSS relatives au paradigme OA ont aussi été introduites. Dans ce contexte, trois familles de métriques relatives au modèle de DSS, au code source et au test ont été compilées pour un système avant et après refactorisation. Les métriques de modèles du DSS ont été également calculées unitairement pour chaque CU avant et après aspectualisation ainsi que la métrique TLOC. Les résultats obtenus concernant l'impact de cette refactorisation au niveau des métriques globales de modèles montrent une baisse de la complexité OO (en termes de taille) et une augmentation de la complexité totale OA (taille et complexité). Les résultats obtenus concernant les métriques globales du code source démontrent une diminution faible de l'effort de développement (SLOC), des méthodes de classes et d'aspects, et une augmentation de la complexité du système OA pour les métriques du code source. Au niveau des métriques globales de test, les résultats démontrent un plus faible effort de test attribué à chaque test individuel du système touché par l'aspectualisation, mais un plus grand effort de test total pour le système global OA pour les métriques de test. Les résultats obtenus concernant les métriques de modèles globales sont identiques aux résultats obtenus au niveau unitaire. De plus, l'analyse des résultats de ces métriques unitaires laisse supposer une augmentation de l'effort de test au niveau des CU impactés par la refactorisation. NCI, NMI et NS sont les métriques de modèle OO qui semblent le plus influencer l'effort de test tandis que pour le modèle OA se sont les métriques NEI, NOPI et NS.

Les résultats obtenus dans ces trois études, démontrent que les métriques simples de modèles (DCU et DC) sont des métriques très intéressantes pour le développement de techniques de priorisation des tests. Toutefois, beaucoup de travail reste encore à faire avant d'atteindre une approche valide et fiable de priorisation des tests à partir des modèles de CU. Dans de futurs travaux, il serait tout d'abord intéressant de valider les approches proposées dans ce mémoire concernant les métriques de modèles de CU. Comme mentionné à plusieurs reprises dans ce mémoire, la meilleure manière de réaliser cette validation serait de contacter des entreprises afin d'obtenir des études de cas complètes et de qualité. Il serait intéressant de vérifier, également, la possibilité d'utiliser des algorithmes d'apprentissage automatique pour pallier le faible taux

d'études de cas obtenues, le cas échéant. Les résultats obtenus concernant la troisième étude de ce mémoire laissent croire que l'extension à la POA de technique de priorisation de l'effort à partir des métriques de modèles serait possible. Dans un second temps, il serait donc intéressant de vérifier, aussi, la possibilité de produire une technique de priorisation qui combine les DCU et DC et voir comment étendre cette technique à la POA.

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

1. Pressman, R.S., *Software engineering: A practitioner's approach*. 2015, McGraw-Hill Education.
2. Larman, C. *UML 2 et les design patterns: analyse et conception orientées objet*.2005.
3. Badri, M., L. Badri, and W. Flageol. *Predicting the size of test suites from use cases: An empirical exploration*. in *IFIP International Conference on Testing Software and Systems*. 2013. Springer.
4. Do, H., et al., *The effects of time constraints on test case prioritization: A series of controlled experiments.* IEEE Transactions on Software Engineering, 2010. **36**(5): p. 593-617.
5. Rothermel, G., et al. *Test case prioritization: An empirical study*. in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. 1999. IEEE.
6. Catal, C. and D. Mishra, *Test case prioritization: a systematic mapping study.* Software Quality Journal, 2013. **21**(3): p. 445-478.
7. Störrle, H. *How are conceptual models used in industrial software development?: A descriptive survey*. in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. 2017. ACM.
8. Reggio, G., et al. *What are the used UML diagrams? A Preliminary Survey*. in *EESSMOD@ MoDELS*. 2013.
9. LaLonde, W.R., J. McGugan, and D. Thomas. *The real advantages of pure object-oriented systems or why object-oriented extensions to C are doomed to fail*. in *[1989] Proceedings of the Thirteenth Annual International Computer Software & Applications Conference*. 1989. IEEE.
10. Kiczales, G., et al. *Aspect-oriented programming*. in *European conference on object-oriented programming*. 1997. Springer.
11. Yoo, S. and M. Harman, *Regression testing minimization, selection and prioritization: a survey.* Software Testing, Verification and Reliability, 2012. **22**(2): p. 67-120.
12. Korel, B., G. Koutsogiannakis, and L.H. Tahat. *Model-based test prioritization heuristic methods and their evaluation*. in *Proceedings of the 3rd international workshop on Advances in model-based testing*. 2007. ACM.
13. Khandelwal, E. and M. Bhadauria, *Various techniques used for prioritization of test cases.* International Journal of Scientific and Research Publications, 2013. **3**(6): p. 1879-1883.
14. Roongruangsuwan, S. and J. Daengdej, *Test case prioritization techniques.* Journal of theoretical and applied information technology, 2010. **18**(2): p. 45-60.
15. Solanki, K. and Y. Singh, *Novel Classification of Test Case Prioritization Techniques.* International Journal of Computer Applications, 2014. **975**: p. 8887.
16. Do, H., G. Rothermel, and A. Kinneer, *Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis.* Empirical Software Engineering, 2006. **11**(1): p. 33-70.
17. Catal, C., *A Survey of Test Case Prioritization Techniques*. 2013. p. 45-74.
18. Basanieri, F., A. Bertolino, and E. Marchetti. *CoWTeSt: A cost weighed test strategy*. in *Proceeding of ESCOM-SCOPE*. 2001.
19. Kundu, D., et al., *System testing for object-oriented systems with test case prioritization.* Software Testing, Verification and Reliability, 2009. **19**(4): p. 297-333.
20. Sapna, P. and H. Mohanty. *Prioritizing use cases to aid ordering of scenarios*. in *2009 Third UKSim European Symposium on Computer Modeling and Simulation*. 2009. IEEE.
21. Gantait, A. *Test case generation and prioritization from uml models*. in *2011 Second International Conference on Emerging Applications of Information Technology*. 2011. IEEE.
22. Kashyap, A., et al. *A model driven approach for system validation*. in *2012 IEEE International Systems Conference SysCon 2012*. 2012. IEEE.

23. Kaur, P., P. Bansal, and R. Sibal. *Prioritization of test scenarios derived from UML activity diagram using path complexity*. in *Proceedings of the CUBE International Information Technology Conference*. 2012. ACM.
24. Sharma, C., S. Sabharwal, and R. Sibal, *Applying genetic algorithm for prioritization of test case scenarios derived from UML diagrams.* arXiv preprint arXiv:1410.4838, 2014.
25. Sabharwal, S., R. Sibal, and C. Sharma. *Prioritization of test case scenarios derived from activity diagram using genetic algorithm*. in *2010 International Conference on Computer and Communication Technology (ICCCT)*. 2010. IEEE.
26. Jena, A.K., S.K. Swain, and D.P. Mohapatra, *Test case generation and prioritization based on UML behavioral models.* Journal of Theoretical & Applied Information Technology, 2015. **78**(3).
27. Weiser, M. *Program slicing*. in *Proceedings of the 5th international conference on Software engineering*. 1981. IEEE Press.
28. Swain, R.K., et al., *Prioritizing test scenarios from UML communication and activity diagrams.* Innovations in Systems and Software Engineering, 2014. **10**(3): p. 165-180.
29. Rhmann, W., T. Zaidi, and V. Saxena, *Use of genetic approach for test case prioritization from UML activity diagram.* International Journal of Computer Applications, 2015. **115**(4).
30. Wang, X., X. Jiang, and H. Shi. *Prioritization of test scenarios using hybrid genetic algorithm based on UML activity diagram*. in *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. 2015. IEEE.
31. Bhuyan, P., A. Ray, and M. Das, *Test Scenario Prioritization Using UML Use Case and Activity Diagram*, in *Computational Intelligence in Data Mining*. 2017, Springer. p. 499-512.
32. Shafie, M.L. and W.M.N. Kadir, *Model-based test case prioritization: A systematic literature review.* Journal of Theoretical and Applied Information Technology, 2018. **96**: p. 4548-4573.
33. Karner, G., *Resource estimation for objectory projects.* Objective Systems SF AB, 1993. **17**: p. 1-9.
34. Badri, M., et al., *Source code size prediction using use case metrics: an empirical comparison with use case points.* Innovations in Systems and Software Engineering, 2017. **13**(2-3): p. 143-159.
35. Toure, F., M. Badri, and L. Lamontagne. *Towards a Unified Metrics Suite for JUnit Test Cases*. in *SEKE*. 2014.
36. Fahmy, T. and A. Aubry, *XLstat.* Société Addinsoft SARL, 1998. **40**.
37. Pradel, C., et al. *L'apprentissage d'ordonnancement pour l'appariement de questions*. 2016.
38. Li, H., *Learning to rank for information retrieval and natural language processing.* Synthesis Lectures on Human Language Technologies, 2014. **7**(3): p. 1-121.
39. B Le, T.-D., et al. *A learning-to-rank based fault localization approach using likely invariants*. in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016. ACM.
40. Friedman, J.H., *Greedy function approximation: a gradient boosting machine.* Annals of statistics, 2001: p. 1189-1232.
41. Crammer, K. and Y. Singer. *Pranking with ranking*. in *Advances in neural information processing systems*. 2002.
42. Shashua, A. and A. Levin. *Ranking with large margin principle: Two approaches*. in *Advances in neural information processing systems*. 2003.
43. Cossock, D. and T. Zhang. *Subset ranking using regression*. in *International Conference on Computational Learning Theory*. 2006. Springer.
44. Li, P., Q. Wu, and C.J. Burges. *Mcrank: Learning to rank using multiple classification and gradient boosting*. in *Advances in neural information processing systems*. 2008.
45. Herbrich, R., *Large margin rank boundaries for ordinal regression.* Advances in large margin classifiers, 2000: p. 115-132.
46. Freund, Y., et al., *An efficient boosting algorithm for combining preferences.* Journal of machine learning research, 2003. **4**(Nov): p. 933-969.

47. Burges, C., et al. *Learning to rank using gradient descent*. in *Proceedings of the 22nd International Conference on Machine learning (ICML-05)*. 2005.
48. Cao, Y., et al. *Adapting ranking SVM to document retrieval*. in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. 2006. ACM.
49. Zheng, Z., et al. *A general boosting method and its application to learning ranking functions for web search*. in *Advances in neural information processing systems*. 2008.
50. Wu, Q., et al., *Adapting boosting for information retrieval measures.* Information Retrieval, 2010. **13**(3): p. 254-270.
51. Cao, Z., et al. *Learning to rank: from pairwise approach to listwise approach*. in *Proceedings of the 24th international conference on Machine learning*. 2007. ACM.
52. Xu, J. and H. Li. *Adarank: a boosting algorithm for information retrieval*. in *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*. 2007. ACM.
53. Yue, Y., et al. *A support vector method for optimizing average precision*. in *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*. 2007. ACM.
54. Metzler, D. and W.B. Croft, *Linear feature-based models for information retrieval.* Information Retrieval, 2007. **10**(3): p. 257-274.
55. Xia, F., et al. *Listwise approach to learning to rank: theory and algorithm*. in *Proceedings of the 25th international conference on Machine learning*. 2008. ACM.
56. Taylor, M., et al. *Softrank: optimizing non-smooth rank metrics*. in *Proceedings of the 2008 International Conference on Web Search and Data Mining*. 2008. ACM.
57. Tibshirani, R., *Regression shrinkage and selection via the lasso.* Journal of the Royal Statistical Society: Series B (Methodological), 1996. **58**(1): p. 267-288.
58. Friedman, J., *Greedy function approximation: a gradient boosting machine. Department of Statistics.* University of Stanford: Stanfors, CA, USA, 1999.
59. Li, H., *Smile-Statistical Machine Intelligence & Learning Engine*. 2016.
60. Dang, V., *The lemur project-wiki-ranklib.* Lemur Project,[Online]. Available: http://sourceforge. net/p/lemur/wiki/RankLib.
61. Mišić, V.B. and D.N. Tešić, *Estimation of effort and complexity: An object-oriented case study.* Journal of Systems and Software, 1998. **41**(2): p. 133-143.
62. Antoniol, G., R. Fiutem, and C. Lokan, *Object-oriented function points: An empirical validation.* Empirical Software Engineering, 2003. **8**(3): p. 225-254.
63. Del Bianco, V. and L. Lavazza. *An empirical assessment of function point-like object-oriented metrics*. in *11th IEEE International Software Metrics Symposium (METRICS'05)*. 2005. IEEE.
64. Chen, Y., et al. *An empirical study of eServices product UML sizing metrics*. in *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE'04.* 2004. IEEE.
65. Tan, H.B.K. and Y. Zhao, *Sizing data-intensive systems from ER model.* IEICE transactions on information and systems, 2006. **89**(4): p. 1321-1326.
66. Tan, H.B.K., Y. Zhao, and H. Zhang, *Conceptual data model-based software size estimation for information systems.* ACM Transactions on Software Engineering and Methodology (TOSEM), 2009. **19**(2): p. 4.
67. Sahoo, P. and J. Mohanty, *Early test effort prediction using UML diagrams.* Indonesian Journal of Electrical Engineering and Computer Science, 2017. **5**(1): p. 220-228.
68. Genero, M., et al., *Building measure-based prediction models for UML class diagram maintainability.* Empirical Software Engineering, 2007. **12**(5): p. 517-549.

69. Genero, M., M. Piattini, and C. Calero, *Early measures for UML class diagrams.* L'objet, 2000. **6**(4): p. 489-505.
70. Zhou, Y., et al., *Source code size estimation approaches for object-oriented systems from UML class diagrams: A comparative study.* Information and Software Technology, 2014. **56**(2): p. 220-237.
71. Carbone, M. and G. Santucci. *Fast&&Serious: a UML based metric for effort estimation*. in *Proceedings of the 6th ECOOP workshop on quantitative approaches in object-oriented software engineering (QAOOSE'02)*. 2002.
72. Jahan, M.V. and R. Sheibani. *A new method for software size estimation based on UML metrics*. in *The national conference on software engineering*. 2001.
73. Kim, S., W.M. Lively, and D.B. Simmons. *An Effort Estimation by UML Points in Early Stage of Software Development*. in *Software Engineering Research and Practice*. 2006.
74. Minkiewicz, A., *Measuring object oriented software with predictive object points.* PRICE Systems, LLC, 1997: p. 609-866.
75. Brodine, D., *Oops, there it is: object-oriented project size estimation.* Enterprise Systems Journal, 2000. **15**(3): p. 55-57.
76. Antoniol, G., et al., *A function point-like measure for object-oriented software.* Empirical Software Engineering, 1999. **4**(3): p. 263-287.
77. Singh, Y., A. Kaur, and R. Malhotra. *Predicting testing effort using artificial neural network*. in *Proceedings of the World Congress on Engineering and Computer Science (WCECS 2008) San Francisco, USA. Newswood Limited*. 2008.
78. Badri, M. and F. Toure, *Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes.* Journal of Software Engineering and Applications, 2012. **5**(7): p. 513.
79. Badri, L., M. Badri, and F. Toure. *Exploring empirically the relationship between lack of cohesion and testability in object-oriented systems*. in *International Conference on Advanced Software Engineering and Its Applications*. 2010. Springer.
80. Badri, L., M. Badri, and F. Toure, *An empirical analysis of lack of cohesion metrics for predicting testability of classes.* International Journal of Software Engineering and Its Applications, 2011. **5**(2): p. 69-85.
81. Bruntink, M. and A. Van Deursen. *Predicting class testability using object-oriented metrics*. in *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. 2004. IEEE.
82. Griffith, I., et al. *The correspondence between software quality models and technical debt estimation approaches*. in *2014 Sixth International Workshop on Managing Technical Debt*. 2014. IEEE.
83. Nguyen, T.H., B. Adams, and A.E. Hassan. *Studying the impact of dependency network measures on software quality*. in *2010 IEEE International Conference on Software Maintenance*. 2010. IEEE.
84. Li, B., et al. *A scenario-based approach to predicting software defects using compressed C4. 5 model*. in *2014 IEEE 38th Annual Computer Software and Applications Conference*. 2014. IEEE.
85. Younis, A., Y.K. Malaiya, and I. Ray, *Assessing vulnerability exploitability risk using software properties.* Software Quality Journal, 2016. **24**(1): p. 159-202.
86. Cotroneo, D., R. Natella, and R. Pietrantuono. *Is software aging related to software metrics?* in *2010 IEEE Second International Workshop on Software Aging and Rejuvenation*. 2010. IEEE.
87. Bartsch, M. and R. Harrison, *An exploratory study of the effect of aspect-oriented programming on maintainability.* Software Quality Journal, 2008. **16**(1): p. 23-44.
88. Coelho, R., et al. *Assessing the impact of aspects on exception flows: An exploratory study*. in *European Conference on Object-Oriented Programming*. 2008. Springer.
89. Figueiredo, E., et al. *Evolving software product lines with aspects*. in *2008 ACM/IEEE 30th International Conference on Software Engineering*. 2008. IEEE.

90. Garcia, A., et al., *Modularizing design patterns with aspects: a quantitative study*, in *Transactions on Aspect-Oriented Software Development I*. 2006, Springer. p. 36-74.
91. Greenwood, P., et al. *On the impact of aspectual decompositions on design stability: An empirical study*. in *European Conference on Object-Oriented Programming*. 2007. Springer.
92. Hoffman, K. and P. Eugster. *Towards reusable components with aspects: an empirical study on modularity and obliviousness*. in *Proceedings of the 30th international conference on Software engineering*. 2008.
93. Alexander, R.T., J.M. Bieman, and A.A. Andrews, *Towards the systematic testing of aspect-oriented programs*. 2004, Technical Report CS-4-105, Department of Computer Science, Colorado State ....
94. Ferrari, F.C., et al., *Testing of aspect-oriented programs: difficulties and lessons learned based on theoretical and practical experience.* Journal of the Brazilian Computer Society, 2015. **21**(1): p. 20.
95. Munoz, F., et al., *Usage and testability of aop: an empirical study of aspectj.* Information and Software Technology, 2013. **55**(2): p. 252-266.
96. ISO/IEC, *ISO/IEC 25010: 2011 Systems and software engineering--Systems and software Quality Requirements and Evaluation (SQuaRE)--System and software quality models*. 2011, CH: ISO Geneva.
97. Commission, I.O.f.S.I.E., *ISO/IEC 9126–Software Engineering–Product Quality*. 2001, Geneve.
98. Badri, M., A. Kout, and L. Badri. *On the effect of aspect-oriented refactoring on testability of classes: A case study*. in *2012 International Conference on Computer Systems and Industrial Informatics*. 2012. IEEE.
99. Badri, M., A. Kout, and L. Badri, *Investigating the effect of aspect-oriented refactoring on the unit testing effort of classes: an empirical evaluation.* International Journal of Software Engineering and Knowledge Engineering, 2017. **27**(05): p. 749-789.
100. Kiczales, G., et al. *An overview of AspectJ*. in *European Conference on Object-Oriented Programming*. 2001. Springer.
101. Lemos, O.A.L. and P.C. Masiero, *Integration testing of aspect-oriented programs: a structural pointcut-based approach.* 22nd SBES, 2008: p. 49-64.